Accurate and Ultra-Fast Launch-Time Validation of Idempotency for GPU Kernels

Mingcong Han, Weihang Shen, Rong Chen*, Haibo Chen

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

We discovered that a GPU kernel can have both idempotent and non-idempotent instances depending on the input. These kernels, called *conditionally-idempotent*, are common in real-world GPU applications—490 out of 547 from six popular applications. This finding reveals a limitation in previous work that statically classifies GPU kernels as idempotent or non-idempotent, potentially compromising the correctness and effectiveness of idempotence-based systems. This paper presents PICKER, the first launch-time analysis system for instance-level idempotency validation. PICKER accurately validates the idempotency of GPU kernel instances before execution by utilizing launch arguments. Several optimizations are proposed to reduce validation latency to microseconds. Evaluations using representative GPU applications (547 kernels and 18,217 instances) show that PICKER accurately identifies idempotent instances with zero false positives and an 18.54% false-negative rate. The launch-time validation completes in under 5 µs for all instances (about 90% under 1 µs). Through integration, PICKER reduces checkpoint costs to less than 4% in fault-tolerant systems and decreases preemption latency by 84.2% in scheduling systems.

CCS Concepts • **Software and its engineering** → **Software verification and validation;** • **Computer systems organization** → **Heterogeneous hardware**

Keywords: GPU kernel idempotency, launch-time validation

ACM Reference Format:

Mingcong Han, Weihang Shen, Rong Chen, and Haibo Chen. 2026. Accurate and Ultra-Fast Launch-Time Validation of Idempotency for GPU Kernels. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3767295.3803597

*Rong Chen is the corresponding author (rongchen@sjtu.edu.cn).

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04

https://doi.org/10.1145/3767295.3803597

1 Introduction

Idempotent GPU kernels promise to produce the same output upon re-execution, without causing any side effects, no matter how many times they are interrupted at any point. Such idempotence property effectively overcomes performance bottlenecks in many aspects of GPU systems, including fault tolerance [51, 101], task preemption [32, 49, 70], and memory persistence [57, 100]. For instance, idempotence-based optimizations can reduce GPU kernel preemption latency in multitask scheduling by an order of magnitude [32, 49, 70, 78], greatly improving the responsiveness of time-sensitive applications (e.g., autonomous driving [6, 41, 50], virtual reality [72, 99], speech recognition [37, 95], and healthcare [20, 28]).

However, given the presence of non-idempotent GPU kernels, idempotence-based optimizations require **prior** knowledge of kernel idempotency (i.e., knowing before execution). This idempotency information must be **correct**, ensuring that no non-idempotent kernels are incorrectly identified as idempotent (i.e., no false positives); otherwise, the system may erroneously re-execute a non-idempotent kernel from an inconsistent state. Moreover, it is also important for the idempotency information to be **effective**, identifying as many idempotent functions as possible (i.e., fewer false negatives), thereby improving system performance by leveraging the benefits of idempotency. Existing systems, however, determine the idempotency of a GPU kernel by statically analyzing either the source code [49, 51, 70] or historical traces [32, 78].

New findings. We discovered that a GPU kernel can have both idempotent and non-idempotent instances, where an instance refers to the invocation of a GPU kernel with a specific input state and arguments. These *conditionally-idempotent* (abbreviated as cond-idempotent) GPU kernels are common in real-world GPU applications. For instance, 490 out of 547 kernels we have studied are cond-idempotent (see Table 1).[1] This prevalence occurs because GPU kernels usually have multiple pointer arguments that can be freely configured by the host program. If the memory used for input and output overlaps, the GPU kernel instance is likely to be non-idempotent as the input data may be overwritten during execution. For example, Fig. 1 shows a cond-idempotent GPU kernel (`vectorAdd`) and its two kinds of instances. Consequently, the idempotency of a GPU kernel instance depends on not only its device code, but also how the host invokes it.

However, prior work statically classifies a GPU kernel as either fully idempotent or non-idempotent at the **kernel**

[1] We also discovered four cond-idempotent GPU kernels in their public application traces from Rodinia [14] and PyTorch [71] (see §3 for details).

level [32, 49, 51, 70, 78], and assumes that the idempotency of all instances of a kernel are the same. This approach overlooks the existence and prevalence of cond-idempotent GPU kernels. Consequently, classifying cond-idempotent kernels as purely idempotent compromises the correctness of idempotence-based systems. Conversely, treating cond-idempotent kernels as purely non-idempotent severely sacrifices the performance of idempotence-based optimizations, reducing their applicability by around 90% (see §3). This **correctness-performance dilemma** has prevented idempotency-based optimizations from being deployed in practice, despite their potential for impressive performance improvements.

Fortunately, our study further reveals that, despite most GPU kernels being cond-idempotent, nearly 80% of instances are idempotent according to public traces (see Table 2). This finding motivates us to identify idempotency at the **instance level**, as it can satisfy both correctness and effectiveness. However, existing approaches for analyzing GPU kernel characteristics encounter fundamental limitations in instance-level validation. Static analysis techniques [54, 56] (e.g., model checking [12]), performed at compile-time, cannot handle dynamically created instances, particularly in modern deep learning frameworks where kernels are pre-compiled without knowing their calling contexts. These static approaches cannot access critical instance-level runtime information, such as the concrete address of memory pointers. On the other hand, dynamic analysis techniques [47, 104] (e.g., instrumentation [94]), performed at runtime, can access instance-level information, but this comes too late for idempotence-based systems that need to determine idempotency before execution.

Key insight. The GPU kernel instance has a unique time window between its submission and execution—a *launch-time* period when arguments are already bound but execution has not yet commenced. This launch-time window overcomes the limitations of existing approaches. Unlike static analysis, it has access to instance-level information, including memory pointers and thread configurations. Unlike dynamic analysis, it occurs before executing the instance. Therefore, the launch-time validation opens up an opportunity to identify idempotency at the instance level.

Unique challenges. Launch-time validation faces two fundamental challenges. First, the information available at launch time is limited—while kernel arguments are accessible, the actual contents of GPU memory are not. The validation must produce accurate results by analyzing only kernel arguments. Second, the launch-time window is extremely short, typically just a few microseconds before execution begins. The validation process must be ultra-lightweight to complete within this narrow timeframe without significant performance impact.

Our approach. To tackle these challenges, we present PICKER, the first launch-time validation system for instance-level idempotency of GPU kernels, with three novel aspects.

First, PICKER proposes a hybrid architecture that combines static analysis (symbolic execution) and dynamic analysis (runtime validation) to effectively leverage the limited instance-level information available during the microsecond-scale launch time. In an offline phase, PICKER performs static analysis on the GPU kernel binary to generate lightweight validation functions. These functions take only the kernel arguments as inputs and determine whether an instance is idempotent. At launch time, PICKER dynamically intercepts GPU kernel launch calls and runs these pre-generated validation functions with the launch arguments.

```
# Idempotent GPU kernel
_global_ void vectorSet(A):
  idx = bid * bdim + tid
  A[idx] = Val

# Non-idem GPU kernel
_global_ void vectorInc(A):
  idx = bid * bdim + tid
  A[idx] = A[idx] + 1
```

```
# Conditionally-idem GPU kernel
_global_ void vectorAdd(A, B, C):
  idx = bid * bdim + tid
  A[idx] = B[idx] + C[idx]

                gdim    bdim
# Idempotent GPU kernel instance
vectorAdd<<<32, 64>>>(x, y, z)
# Non-idem GPU kernel instance
vectorAdd<<<32, 64>>>(x, y, x)
```

Fig. 1: *The simplified version of three GPU kernel types from CUDA [66]: an idempotent kernel, a non-idempotent kernel, and a conditionally-idempotent kernel with its two instances. The variables* `bid`, `tid`, `gdim`, *and* `bdim` *stand for block index, thread index, grid dimension, and block dimension, respectively.*

Second, PICKER ensures correctness through a conservative validation approach. It transforms the validation of instance-level idempotency into determining whether any memory is both read and written by the instance. PICKER identifies an instance as idempotent only when it can definitively prove that no memory location experiences both read and write accesses during execution. Any uncertainty (e.g., non-parameter variables) automatically results in classifying the instance as non-idempotent, thus satisfying the correctness requirement of idempotence-based systems.

Third, GPU kernel instances typically use thousands of threads that collectively access billions of memory addresses—far too many to validate within microseconds at launch time. To address this scaling challenge, PICKER employs range abstractions instead of individual addresses, to represent memory accesses across all GPU threads at the same instruction. By leveraging the validated monotonicity between thread IDs and memory access patterns, PICKER only needs to calculate addresses for threads with minimum and maximum IDs in most cases. This optimization dramatically reduces computational complexity while maintaining validation accuracy.

We implemented PICKER to directly analyze GPU kernel assembly code (i.e., SASS code for NVIDIA GPU [33, 63]), making it applicable to production-level GPU applications—including those with closed-source or dynamically generated kernels. We evaluated PICKER using applications from well-known GPU benchmarks (i.e., Rodinia [14] and Parboil [85]) and popular deep learning frameworks (i.e., TVM [91], PyTorch [71], TensorRT [64], and FasterTransformer [67]). Our experimental results show that PICKER successfully identifies 11,745 idempotent instances from 18,217 total instances, achieving zero false positives with only an 18.54% false-

negative rate. Impressively, all GPU instances were validated within 5 µs, with about 90% completing in under 1 µs. Furthermore, by utilizing the correct and effective idempotency information provided by PICKER, a GPU fault-tolerant system [51] reduced the checkpoint cost of error-free execution from over 115% to under 4%, while a GPU scheduling system [70] achieved an 84.2% reduction in preemption latency.

Contributions. We summarize our contributions as follows.

- The discovery of the prevalence of cond-idempotent GPU kernels in real-world applications (§2), with an in-depth study on idempotency that reveals the necessity of instance-level idempotency validation (§3).
- A new system architecture for launch-time validation that combines static analysis and dynamic validation (§4).
- An approach to accurately validate the instance-level idempotency by checking memory access overlapping (§5), with optimizations enabling microsecond-scale validation (§6).
- A prototype implementation and an evaluation that demonstrates the efficacy and efficiency of PICKER (§7).

PICKER, including tools and workloads, is open-source and available at https://github.com/XpuOS/picker.

2 Background

2.1 Idempotence-Based GPU Systems

Idempotent GPU kernels consistently produce the same output no matter where their execution is interrupted and then restart. This has been widely used to accelerate GPU applications where kernel execution may face either unintentional or intentional interruptions, as illustrated in the following cases.

Case 1: GPU checkpointing. Checkpointing is a fundamental mechanism for GPU systems that need to capture and restore execution state, enabling critical functionalities like fault recovery [26, 62, 69, 96, 97] and task migration [80, 97, 98]. However, conventional checkpointing imposes a substantial performance penalty owing to the massive amount of data that must be saved. Therefore, prior work [51] leverages the idempotency of GPU kernels to mitigate this overhead by skipping the checkpointing of idempotent kernels and instead using re-execution for their restoration. This reduction in checkpointing overhead is particularly vital for latency-critical applications with strict deadlines, such as embodied AI pipelines [105].

Case 2: preemptive scheduling. Latency-sensitive GPU applications, such as packet processing [30, 46] and real-time model inference [29, 38], demand a fast kernel preemption mechanism for scheduling to meet the service level objective (SLO) of latency. However, the conventional preemption mechanism—context switching—introduces high preemption latency on the GPU (typically tens of microseconds) due to the large size of on-chip context (e.g., 20 MB for NVIDIA V100 [65]). This may severely harm the responsiveness and safety of latency-critical GPU systems [58, 70]. Fortunately, researchers have recently leveraged the idempotency of GPU kernels to drastically reduce preemption latency [32, 49, 70], since idempotent kernels can be instantly killed (in about 1 µs) and then restarted without saving their context.

Case 3: soft error correction. Soft errors in unprotected GPU ALUs pose a critical reliability concern, particularly in emerging large-scale workloads like LLM training [35, 60, 76, 89, 90, 102]. For instance, during the pre-training of models such as Gemini and Llama 3, silent data corruptions (SDCs) caused by soft errors were observed one to two times per week [89, 90]. A single error can halt an entire training job that spans thousands of GPUs, forcing a costly rollback and wasting thousands of GPU hours. While mechanisms to detect these errors exist [59, 92], correcting them efficiently and cost-effectively remains a challenge. Idempotent kernels offer an elegant solution: upon detecting a soft error, these kernels can be safely discarded and re-executed to produce the same output without causing any side effects [103].

An important prerequisite for these idempotence-based systems to work efficiently is that they need to accurately know the idempotency of GPU kernels. We summarize three key requirements for the provided idempotency information.

R0: practicality (knowing before execution). The idempotency of a GPU kernel instance should be identified before actually executing it, enabling systems to make appropriate decisions about how to handle each kernel.

R1: correctness (no false positives). A GPU kernel instance that is non-idempotent should not be mistakenly classified as idempotent, avoiding incorrect results if systems re-execute them from inconsistent states.

R2: effectiveness (fewer false negatives). A GPU kernel instance that is idempotent should not be mistakenly classified as non-idempotent, preventing systems from missing opportunities to leverage their idempotent nature.

2.2 Kernel-Level Idempotency

Prior work statically classifies a GPU kernel as either idempotent or non-idempotent [49, 51], and treats all instances of a GPU kernel as idempotent or non-idempotent, respectively. However, we discovered that a GPU kernel can have both idempotent and non-idempotent instances, depending on the instance's input.

An illustrative example. Fig. 1 shows a simplified version of the `vectorAdd` GPU kernel from CUDA samples [66] that calculates the sum of two vectors. When the kernel is invoked with three distinct pointers (i.e., the first instance with arguments `x`, `y` and `z`), the GPU kernel instance is idempotent because the input buffers (i.e., `y` and `z`) are not overwritten during execution. However, the second instance where the parameters `A` and `C` have the same value (i.e., `x`) is non-idempotent, because the input vector pointed by `x` is modified during execution, and the result of repeatedly executing this instance is different from that of only executing it once.

Table 1: *The kernel-level idempotency of the evaluated GPU kernels. The columns ●, ○, and ◐ indicate that the GPU kernel is idempotent, non-idempotent, and cond-idempotent, respectively. The columns $◐_T$ indicates that the GPU kernel has both idempotent and non-idempotent instances from their public application traces.*

GPU Apps	Code	#Kernels	●	○	◐	$◐_T$
Rodinia [14]	Source	40	7	12	21	2
Parboil [85]	Source	25	4	12	9	0
TVM [91]	Source	308	0	0	308	0
PyTorch [71]	Binary	66	3	1	62	2
TensorRT [64]	Binary	58	0	2	56	0
FT [67]	Binary	50	9	7	34	0
All		547	23	34	490	4

Definition: kernel-level idempotency. Considering that different instances of the same GPU kernel may vary in their idempotency (i.e., idempotent or non-idempotent), the idempotency of a GPU kernel can no longer be categorized as a simple binary distinction. In this paper, we propose a new categorization of kernel-level idempotency for GPU kernels based on the idempotency of their instances, as follows:

- ● **Idempotent** GPU kernel: All instances of the kernel are idempotent.
- ○ **Non-idempotent** GPU kernel: All instances of the kernel are not idempotent.
- ◐ (NEW) **Conditionally-idempotent** GPU kernel: Some instances of the kernel are idempotent, while others are not.

For example, as shown in Fig. 1, the `vectorSet` kernel is idempotent because the value of `A[idx]` remains the same no matter how many times it is re-executed. In contrast, the `vectorInc` kernel is non-idempotent because the value of `A[idx]` is incremented by one every time it is re-executed. Lastly, the `vectorAdd` kernel is conditionally-idempotent, as it can have both idempotent and non-idempotent instances.

3 A Study of Idempotency in Applications

To gain a better understanding of idempotency in real-world GPU applications, we conducted a comprehensive study on well-known GPU benchmarks (i.e., Rodinia [14] and Parboil [85]) and popular DL frameworks, including TVM [91], PyTorch [71], TensorRT [64], and FasterTransformer [67] (abbreviated as FT). These experiments were performed on an NVIDIA GV100 GPU.[2] More details can be found in §7.1.

An important aspect of our study is that it covers both open-source kernels and production kernels distributed only as binaries. In particular, although PyTorch and FasterTransformer are open-source at the framework level, their execution frequently invokes closed-source GPU kernels from vendor libraries (e.g., cuDNN and TensorRT), and may include JIT-generated kernels, which motivates binary-level analysis and tracing. Moreover, framework- or source-level reasoning alone cannot reliably capture low-level pointer aliasing and in-place updates performed within these kernels.

Instance-level idempotency tracing. We build an idempotency tracing tool[3] using GPU dynamic instrumentation [94]. The tool records all accessed addresses on the GPU memory during execution. We analyze the addresses to detect the presence of a clobber anti-dependency access pattern, which specifically refers to a write-after-read without a prior read-after-write on the same address [21]. If no clobber anti-dependency is identified, the instance is categorized as idempotent, otherwise non-idempotent.

Kernel-level idempotency labeling. It is non-trivial to correctly and effectively identify idempotent or non-idempotent kernels, as it requires knowing the idempotency of all its instances. However, it is straightforward to identify a cond-idempotent kernel once both an idempotent and a non-idempotent instance are observed. Therefore, we adopt a simple yet conservative approach to identify cond-idempotent kernels. We first try our best to find both idempotent and non-idempotent instances for a kernel to confidently label it as cond-idempotent. If this approach fails, we roughly label it as either idempotent or non-idempotent, although it is still possible that the kernel is actually cond-idempotent.

Specifically, we first run the applications with public traces (i.e., traced instances) and use our tracing tool to obtain the instance-level idempotency. Then, we construct additional instances (i.e., generated instances) that have opposite idempotency to their instances from the trace. The construction is performed by changing the value of the pointers in the launch arguments. If a traced instance is idempotent, we try to generate a non-idempotent instance by assigning the same value to the write pointers in the launch arguments. Conversely, if a traced instance is non-idempotent, we try to generate an idempotent instance by assigning distinct values to all pointers in the launch arguments. We use both traced and generated instances to label the kernel-level idempotency.

The main results for the idempotency of GPU kernels and instances are reported in Table 1 and Table 2, respectively. We outline three important observations as follows.

O1: the existence of cond-idempotent kernels. The column $◐_T$ in Table 1 displays the number of kernels that include both idempotent and non-idempotent instances from their public application traces. Upon analysis, we discovered four kernels with such instances, providing conclusive evidence of the existence of cond-idempotent kernels.[4] For example, one of these kernels closely resembles the straightforward `vectorAdd` kernel depicted in Fig. 1, performing an element-wise tensor addition. These real-world cases highlight the risk of misclassifying non-idempotent instances as

[2]We also conducted these experiments on an NVIDIA RTX 3080 GPU, obtaining similar results that further confirmed our findings, which can be found in Appendix B.

[3]The tool is available at https://github.com/XpuOS/picker.

[4]Detailed information on these four cond-idempotent GPU kernels can be found in Appendix A.

Table 2: *The instance-level idempotency of the traced instances. The kernel-level idempotency is the same as that in Table 1. The columns ● / ★ (○ / ☆) indicates the idempotent (non-idempotent) instances of idempotent (non-idempotent) kernels. The columns ◐ / ★ and ◐ / ☆ indicate the idempotent and non-idempotent instances of cond-idempotent kernels, respectively.*

GPU Apps	#Instances	● / ★	○ / ☆	◐ / ★	◐ / ☆
Rodinia [14]	4,527	85	78	4,334	30
Parboil [85]	1,033	103	738	192	0
TVM [91]	609	0	0	609	0
PyTorch [71]	1,570	151	1	1,131	287
TensorRT [64]	478	0	8	465	5
FT [67]	10,000	229	988	7,119	1,664
All	18,217	568	1,813	13,850	1,986

idempotent (i.e., false positives) when treating conditionally-idempotent kernels optimistically. Consequently, the correctness of idempotence-based systems can be compromised, directly contravening **R1**.

O2: the majority of kernels are cond-idempotent. The column ◐ in Table 1 identifies a total of 490 cond-idempotent kernels, accounting for 89.6% of all kernels. Among them, 486 kernels are labeled as cond-idempotent because we constructed instances that have the opposite idempotency to their instances from the trace. This indicates that simply modifying the input pointer of an instance can alter its idempotency, further highlighting the prevalence of cond-idempotent kernels. The fundamental issue arises from the separation between host-side caller and device-side callee, which enforces all kernel arguments to be passed via pointers. Although GPU kernels can use `restrict` qualifiers internally to prevent pointer aliasing, these constraints do not extend across the host-device boundary. This allows host code to freely pass the same pointer to multiple kernel arguments, leading to overlapping memory regions and altering the idempotency of an instance. These findings illustrate the necessity of a cautious approach, treating cond-idempotent GPU kernels as non-idempotent. Although the correctness of idempotence-based systems would remain intact, most GPU kernels and instances would be erroneously classified as non-idempotent (i.e., false negatives). This would compromise the effectiveness of idempotence-based systems, contradicting **R2**.

Collectively, the observations presented above indicate that the existing approaches [49, 51, 70] that statically categorize kernels as either idempotent or non-idempotent and assume that idempotency for all instances is the same as the kernel-level idempotency, can greatly compromise the correctness (**R1**) or the effectiveness (**R2**) of idempotence-based systems.

O3: the majority of instances are idempotent. Table 2 illustrates that out of all traced instances, 14,418 (79.1%) are classified as idempotent (★). Remarkably, even for cond-idempotent kernels, 13,850 out of 15,836 instances are idempotent (◐/★), accounting for 87.5% of their total instances. Consequently, it is evident that although cond-idempotent kernels can have both idempotent and non-idempotent instances, the majority are indeed idempotent. This indicates that determining the idempotency of each instance individually is promising to satisfy the effectiveness (**R2**) of idempotence-based systems without compromising the correctness (**R1**).

Based on these observations, we argue that *idempotency must be determined at the instance level for idempotence-based systems*. Unfortunately, existing techniques for analyzing GPU kernel properties fall short when directly applied to instance-level idempotency determination.

Static analysis (e.g., GKLEE [54]) can only work on instances that are compiled with constant parallelism parameters and explicitly provided launch arguments (e.g., the instances in Fig. 1). However, in practice, GPU kernels are often compiled into a shared library (e.g., `cuDNN`) or even just a JIT-compiled binary (e.g., GPU kernels in PyTorch), and invoked with dynamically generated launch arguments. Therefore, most GPU kernels must be validated with instance-level information rather than through offline static analysis.

Dynamic analysis (e.g., iGUARD [47]) can use code instrumentation to trace GPU kernel instance execution and then determine instance properties based on the observed execution behaviors. Although it does produce accurate results, it cannot be applied to idempotence-based systems because the results are only available after execution, contradicting **R0**.

4 Approach and Overview

Opportunity: kernel launch. A GPU kernel instance has a specific *launch time* that occurs between its submission and execution. During this period, all kernel arguments are bound while the instance has not yet been sent to the GPU. The launch time presents a unique opportunity to validate instance-level idempotency using runtime information (e.g., memory pointers and thread configurations) before execution.

Our approach: launch-time validation. The key idea behind PICKER is to leverage launch arguments to validate the idempotency of each GPU kernel instance. This launch-time validation overcomes the limitations of existing approaches for idempotence-based systems. Unlike static analysis, which lacks runtime information about GPU kernels, our approach can accurately examine the actual memory addresses that will be accessed. Unlike dynamic analysis, which captures runtime behaviors during execution, our approach can timely get the necessary information before executing the instance.

Challenge. A straightforward approach for launch-time validation would entail applying existing analysis techniques directly to the GPU kernel code at launch time. However, this intuitive solution encounters two severe limitations. First, the information available at launch time is restricted to the kernel arguments, with no access to the actual data in GPU memory. Second, the launch-time window is extremely tight, typically just a few microseconds, making complex code

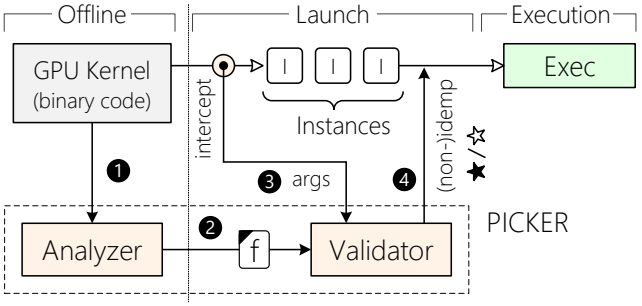

Fig. 2: *Architecture of* PICKER *and the workflow of integrating* PICKER *to run GPU kernels on idempotence-based systems.*

analysis infeasible. Collectively, these constraints present a daunting technical challenge: *how to perform accurate validation within microseconds using only the limited information available at launch time.*

Overview of PICKER. PICKER is the first launch-time validation system for instance-level idempotency of GPU kernels. To overcome the aforementioned challenges, it proposes a hybrid architecture that combines static analysis (symbolic execution) and dynamic analysis (runtime validation). Fig. 2 illustrates the architecture of PICKER and its integration with an idempotence-based system. PICKER consists of two main components: an offline analyzer and a launch-time validator. Rather than attempting complex code analysis at launch time, PICKER conducts a comprehensive offline analysis of GPU kernel binary code (❶) without any instance-level information. This analysis extracts and compresses idempotency-related information into compact validation functions, which are compiled into dynamic shared libraries (❷). At launch time, the validator loads these pre-compiled libraries and intercepts GPU kernel instances, validating them by invoking the pre-generated functions with actual launch arguments (❸). These functions return a boolean value indicating the idempotency of the instance (❹). The instance-level idempotency information is passed to the idempotence-based system, enabling it to take appropriate actions (e.g., checkpointing) in advance. The validation functions are carefully designed to produce accurate results (see §5) within microseconds (see §6).

5 Accurate Launch-Time Validation

The idempotency information provided to the idempotence-based systems must be accurate, meaning that there should be no false positives (**R1**) and less false negatives (**R2**), so the idempotence-based system can correctly take appropriate actions based on the idempotency information. This section details the design of the validation function generated by PICKER to satisfy both **R1** and **R2**.

5.1 Memory-Address-Based Validation Condition

We first establish a condition that can be validated prior to the execution of a GPU kernel instance (**R0**), and is *sufficient* for an instance to be considered idempotent.

An idempotent instance ensures that, when re-executed with the same inputs as its initial execution, the output will be consistent with the exactly-once execution. Therefore, a common approach is to check for modifications to the *input memory* of an instance during execution. Input memory refers to the memory region that was first accessed as a read. Specifically, an instance is considered as non-idempotent if there exists any write-after-read access patterns on its input memory [22, 49]. However, analyzing the memory access order in a GPU kernel instance, which utilizes multiple concurrent GPU threads and has interleaved memory accesses, is extremely complex, if not impossible.

Our approach: validating R/W overlaps. PICKER enforces a more stringent condition to validate the idempotency of a GPU kernel instance, where an instance is considered non-idempotent if there exists any overlap in the read and write addresses of all potential memory accesses, regardless of the access order. On the other hand, an instance is classified as idempotent if PICKER can ensure that each byte of GPU memory is either read-only or write-only. Although this may occasionally lead to false negatives (e.g., a read-after-write pattern misclassified as non-idempotent), it upholds the requirement of no false positives (**R1**).

Up to this point, the problem of validating instance-level idempotency has transformed into the task of accurately predicting the memory addresses that the instance will access. To satisfy **R1**, the predicted addresses must cover all the really accessed addresses. To satisfy **R2**, the predicted addresses should minimize the inclusion of non-accessed addresses.

5.2 Strawman

We start by presenting a strawman solution, to demonstrate the fundamental design of PICKER. The basic idea is to analyze the GPU kernel code offline to extract the expressions that determine accessed memory addresses, and then calculate the concrete addresses at launch time using the launch arguments.

Static analyzer. The analyzer in PICKER first analyzes the binary code of GPU kernels via symbolic execution. The symbolic execution engine records the memory addresses when it encounters load/store instructions on GPU global memory. Each recorded address represents a symbolic expression, referred to as a *symbolic address*. The variables inside a symbolic address typically consist of the parameters of the GPU kernel (including the parallelism parameters), as well as the block and thread IDs of the GPU threads that execute the instruction. Moreover, each symbolic address is associated with a list of path conditions that must be satisfied to traverse the path during symbolic execution. Finally, the analyzer saves the symbolic addresses, which will be loaded by the validator at runtime. Additionally, the analyzer also generates a *global condition* that contains some preconditions (explained later).

Launch-time validator. The validator in PICKER is responsible for validating the idempotency of instances at launch time, using the symbolic addresses generated by the analyzer. Fig. 3

```
# Instance-level idempotency validation
bool Validator::is_idempotent(kern_info, args, dims) {
      # check the global condition
1     if not kern_info.global_cond(args, dims)
2       return FALSE    # not satisfied as non-idempotent
      # calculate all possible concrete R/W addresses
3     for path_cond, sym_addr in kern_info.sym_addrs
4       for bid in {0, 1 .. dims.gdim-1}
5         for tid in {0, 1 .. dims.bdim-1}
            # check the path condition
6           if not path_cond(args, dims, bid, tid) then
7             continue
            # calculate concrete address
8           conc_addr = sym_addr(args, dims, bid, tid)
9           mem_addrs.extend(conc_addr)
      # validate R/W overlaps
10    for addr in mem_addrs.read_addrs
11      if addr in mem_addrs.write_addrs then
12        return FALSE    # overlap exists: non-idempotent
13    return TRUE    # no overlap: idempotent
14 }
```

Fig. 3: *Pseudocode of launch-time idempotency validation.*

shows the simplified pseudocode of the runtime idempotency validation. For each GPU kernel instance, the validator first validates the global condition (Lines 1–2), and then calculates the concrete addresses of all memory accesses (Lines 3–9). For each symbolic address, the validator iterates through all block and thread IDs to calculate the concrete address accessed by each GPU thread (Lines 4–5). For simplicity but without loss of generality, we only consider that the block and thread IDs are one-dimensional. Within each iteration, the validator confirms whether the path condition of the symbolic address is satisfied (Lines 6–7), and then calculates the concrete address (Line 8). Finally, the validator determines the idempotency of the instance by checking the overlap between the read and write addresses (Lines 10–12).

In order to satisfy **R1**, PICKER must analyze all possible memory addresses an instance may access. This requires the analysis to cover all possible control flows and data flows. Because GPU kernels typically have no complex behaviors [47, 61], the analysis can be gracefully completed most of the time. However, there are still some cases where the analysis may fail to cover all flows, e.g., due to indirect function calls. For these cases, PICKER conservatively assumes that the kernel is non-idempotent for all its instances.

Handling unbounded loops. The symbolic execution may fail due to unbounded loops. When encountering an unbounded loop, the analyzer executes the loop body a fixed number of times, and then breaks the loop. If the loop is actually executed more times than the analyzer expected, the instance should be treated as non-idempotent. The analyzer achieves this by generating a global condition that the instance must satisfy if the loop is executed more times than expected. The validator then checks whether the condition is satisfied. If not satisfied, the instance is classified as non-idempotent.

Handling non-parameter variables. A symbolic address may

```
# Element-wise B=ReLU(A)                                  GPU Kernel Code
_global_ void elemwise_relu(A, B, N):
  for i in N
    idx = (bid * bdim + tid) * N + i
    B[idx] = max(A[idx], 0)
```

```
Global Condition: N <= 32                                          Analyzer
```

Condition	Symbolic Address	Bytes	R/W
N > 0	A + 4 * ((bid*bdim+tid)*N+ 0)	4	R
N > 0	B + 4 * ((bid*bdim+tid)*N+ 0)	4	W
N > 1	A + 4 * ((bid*bdim+tid)*N+ 1)	4	R
N > 1	B + 4 * ((bid*bdim+tid)*N+ 1)	4	W
...	...	.	.
N > 31	A + 4 * ((bid*bdim+tid)*N+31)	4	R
N > 31	B + 4 * ((bid*bdim+tid)*N+31)	4	W

```
Instance Args: A = 0x10000, B = 0x20000, N = 16                    Validator
Instance Dims: gdim = 0x20, bdim = 0x20
READ ADDRs   : {0x10000, 0x10001 ... 0x1ffff}
WRITE ADDRs  : {0x20000, 0x20001 ... 0x2ffff}
Idempotent   : True
```

Fig. 4: *An example for idempotency validation using our strawman solution.* `elemwise_relu` *is a simplified version of a cond-idempotent GPU kernel mentioned in §3 from PyTorch [71].*

consist of variables that are not from the parameters, dimensions or IDs, called *non-parameter variables*. For example, dereferencing a double pointer (e.g., `A[0][0]`) may generate a symbolic address that consists of a variable read from the GPU memory. It is worth noting that the validator can only get the concrete values of the parameters and dimensions of the instance, but not the variables from the GPU memory. Therefore, PICKER conservatively assumes that these variables could be any concrete value. Most of the time, a symbolic address with non-parameter variables is considered to be overlapped with any other address, and thus classified as non-idempotent.

Idempotent/non-idempotent GPU kernels. PICKER makes a simple analysis of the symbolic addresses to detect kernels whose instances are always idempotent or non-idempotent. For example, if a kernel has no load or store instructions, it can be safely treated as idempotent. While, if a kernel reads and writes the same symbolic address, or contains an atomic instruction, it is also directly classified as non-idempotent. The validator directly returns kernel-level idempotency without performing the validation for these kernels.

Example. Fig. 4 illustrates a kernel performing the ReLU activation function on each element of the input array, with each GPU thread responsible for computing `N` elements. The analyzer symbolically executes the code and records the addresses of all memory accesses. Each symbolic address consists of the parameter variables (i.e., `A`, `B`, and `N`), the dimension variables (i.e., `bdim` and `gdim`), alongside the block and thread IDs (i.e., `bid` and `tid`). Because the kernel includes an unbounded loop, the analyzer only executes the loop body for a predefined number of times (i.e., 32 times in the example), and generates a global condition (`N<=32`)

to identify instances that execute the loop more times. At runtime, the validator first validates the global condition. It then enumerates all `bid` and `tid` to calculate the addresses. Ultimately, the validator identifies no overlap in the read and write addresses, classifying the instance as idempotent.

6 Ultra-Fast Launch-Time Validation

Our experimental results (see §7.3) show that the strawman solution takes up to 50 ms to validate a GPU kernel instance, which exceeds the microsecond-scale launch-time window by four orders of magnitude, thereby introducing substantial overhead to application performance.

Upon examining the pseudocode in Fig. 3, we can formulate an estimation of the validation latency as the product of three critical factors: the number of symbolic addresses within the kernel ($N_{symAddr}$), the total GPU threads for the instance (N_{Thrd}), and the average execution time required to calculate the concrete value of a symbolic address (T_{Instr}). Each of these factors significantly influences the validation latency. Therefore, we propose several optimizations to reduce these factors individually, with the aim of achieving the microsecond-scale validation latency.

6.1 Reducing N_{Thrd}: Range-based Address Model

Our strawman solution performs concrete address calculations for each GPU thread, resulting in the execution time being proportional to the total number of GPU threads (`bdim`×`gdim`). However, an instance typically comprises thousands of GPU threads, leading to lengthy computation times. Therefore, it is necessary to reduce this factor to a constant value that is independent of the number of GPU threads.

Representing addresses with ranges. We observed that the memory addresses accessed by different GPU threads at the same load/store instruction are often *consecutive*. This pattern is recommended to enhance memory access efficiency by coalescing the memory accesses of GPU threads [15, 40]. Consequently, instead of enumerating all the individual addresses, PICKER represents the memory addresses accessed by all GPU threads at the same load/store instruction using a *range*, which is defined by two values: the upper bound and the lower bound. If we can calculate the two bounds of the range in constant time, the execution time of the validation would be independent of the factor N_{Thrd}. However, without enumerating all the GPU thread IDs, computing the lower bound, for instance, requires solving the analytic form of the bound function in advance. This function, denoted as $LB(\texttt{args}) = \min_{\texttt{bid},\texttt{tid}} Addr(\texttt{bid},\texttt{tid},\texttt{args})$, represents a non-trivial challenge as it involves solving bound functions for symbolic addresses of arbitrary form.

Estimating range bounds with monotonicity. We further discovered that in most cases (82% symbolic addresses of the evaluated kernels), the values of symbolic addresses increases monotonically with respect to GPU thread IDs (`bid` and `tid`). Taking the symbolic address `A+4×(bid×bdim+tid)×N` in Fig. 4 as an example, with concrete values, the symbolic address becomes `0x10000+4×(bid×0x20+tid)×16`, which demonstrates a monotonically increasing trend with respect to both `bid` and `tid`. In other words, the lowest address is accessed by the thread with the smallest IDs (`bid` and `tid`), and the highest by the largest. This monotonicity allows us to calculate the address range using only the smallest and largest thread IDs, making validation latency independent of N_{Thrd}.

Checking monotonicity. However, the monotonicity of symbolic addresses is not always guaranteed. Before using the smallest and largest GPU thread IDs to calculate the range, PICKER statically checks the monotonicity of the symbolic addresses on the variables of GPU thread IDs offline. To check the monotonicity of $Addr(\texttt{bid},\texttt{tid},\texttt{args})$ on `bid`, PICKER exploits the capabilities of the SMT solver [23] to prove that $Addr(\texttt{x},\texttt{tid},\texttt{args}) \geq Addr(\texttt{y},\texttt{tid},\texttt{args})$ holds true whenever $\texttt{x} > \texttt{y}$. If the solver cannot find a counterexample after exhausting the entire search space, we say that the symbolic address is monotonically increasing with respect to `bid`. In contrast, if the solver finds a counterexample or times out, the symbolic address is considered to be non-monotonic. The checking is also performed on `tid` in the same way.

In fact, the monotonicity is often satisfied conditionally. For instance, in the first symbolic address in Fig. 4, if `A` is a very large address (e.g., $2^{64}-1$), the address may overflow when `bid` and `tid` increase, making it not monotonic. But real-world applications rarely encounter such overflow, because the address space is much smaller than 2^{64}. Therefore, PICKER makes prior constraints on the value of the launch arguments for the monotonicity checking. These constraints are also added to the *global condition* to be validated at runtime. For example, PICKER may assume $\texttt{A} < 2^{56}$, $\texttt{N} < 2^5$, $\texttt{bdim} < 2^{10}$, and $\texttt{gdim} < 2^{10}$, which are reasonable constraints for real-world applications and makes symbolic addresses monotonic. If an instance violates these constraints, PICKER would conservatively treat it as non-idempotent.

The constraints can be either inferred by PICKER or manually specified by the programmer. Currently, PICKER can automatically generate constraints for pointer-typed parameters. Other types of parameters must be specified manually.

Handling non-monotonic symbolic addresses. PICKER attempts to transform a non-monotonic symbolic address into a monotonic form by replacing certain subexpressions with new independent variables. For example, the symbolic address `A+tid%10` is non-monotonic with respect to `tid`, but it can be transformed into `A+var` where `var` is a new variable constrained to be in the range $[0, 9]$. This guarantees the transformed address's range is a subset of the original, avoiding false positives (**R1**), but may lead to false negatives.

The transformation involves two steps. First, PICKER traverses the expression tree of the symbolic address bottom-up to find non-monotonic subexpressions. Then, it uses SMT

solver to determine the minimum and maximum values of these non-monotonic subexpressions, and replaces them with new variables constrained to the range. The new symbolic address is then checked for monotonicity, and the process is repeated until the it is monotonic. For most of the cases, the transformation can stop after eliminating periodic functions like `tid`%10 and `tid`&7. In the worst case, the whole symbolic address may be replaced with a new variable, which is almost equivalent to classifying the kernel as non-idempotent.

Leveraging constraints on GPU thread IDs. Using the smallest and largest GPU thread IDs to calculate the range of a monotonic symbolic address may overestimate the actual accessed address range if the instruction is not executed by all GPU threads. For instance, if a symbolic address has a path condition of `tid`<10, but is launched with `bdim`=32, the address should only be calculated with `tid` in the range $[0, 9]$, not $[0, 31]$. Therefore, PICKER should leverage these path conditions to narrow down the estimated range.

PICKER utilizes pattern matching to identify path conditions that can narrow down the range. It first looks for conditions that take the form of a comparison between a variable part (an expression with GPU thread IDs) and a constant part (an expression consisting solely of launch arguments and constants). It then replaces the variable part with a new variable whose range is restricted by the constant part. For example, the path condition `tid`<`N` matches this pattern, where the variable part is `tid` and the constant part is `N` which is a launch argument. The range of the variable part can be confined to $[0, min(\texttt{N}-1, \texttt{bdim}-1)]$, with the comparison between `N`−1 and `bdim`−1 left to the launch-time validator.

6.2 Reducing $N_{symAddr}$: Range Compaction

The presence of loops causes load/store instructions to be symbolically executed multiple times, generating a large number of symbolic addresses that increases with the number of loop iterations, leading to a significant value of $N_{symAddr}$. PICKER reduces $N_{symAddr}$ by merging symbolic addresses generated from the same instruction within a loop, resulting in a single merged symbolic address to calculate at runtime.

We observed that addresses of the load/store instructions inside loops in most kernels consist of only loop-invariant variables and induction variables. Induction variables are variables that are incremented or decremented by a fixed step in each loop iteration [5]. For example, all kernels generated by TVM [91] that we evaluated match this pattern. Our strawman solution which unrolls the loop, is essentially equivalent to enumerating the values of induction variables. However, it is unnecessary to enumerate the induction variables, as their range can be symbolically analyzed using the initial value, step value, and the loop iterations. Thus, we can treat induction variables as another type of independent variable similar to GPU thread IDs that fit the range-based address model.

Specifically, PICKER performs dataflow analysis to identify the induction variables and loop-invariant expressions offline. If a symbolic address consists of only induction variables and loop-invariant expressions, and the range of induction variables can be represented using only launch arguments, PICKER unrolls the loop body once and retains the induction variables in the symbolic addresses. These variables are then included in the monotonicity checking and range calculation.

For example, consider the loop in Fig. 4 with a read of the address of `A`[(`bid`×`bdim`+`tid`)×`N`+`i`]. Here, `i` is as an induction variable incrementing by 1 in each loop iteration, while the rest variables are loop-invariant. PICKER can statically determine that `i` iterates from 0 to `N`−1. Our strawman solution unrolls the loop and generates 32 symbolic addresses for this instruction. However, PICKER retains `i` as an independent variable in the symbolic address, allowing it to further participate in the monotonicity checking and in calculating the concrete range using its bound values (i.e., 0 and `N`−1). This optimization reduces the number of symbolic addresses generated for the instruction to only one, and for the entire kernel to only two, significantly reducing $N_{symAddr}$.

6.3 Reducing T_{Instr}: Compiled Execution

A straightforward approach to calculate the concrete ranges of symbolic addresses is to directly use the *substitute* function of the symbolic engine. This function replaces symbolic variables with concrete values and evaluates the expressions. Nevertheless, the interpretive evaluation of symbolic expressions is considerably slow. In contrast, PICKER converts the symbolic addresses into native code, specifically C language, and subsequently compiles it into a dynamic shared library.

PICKER also applies several common optimizations to the native code execution. For example, the loop over the symbolic addresses is unrolled, and the functions that calculate the ranges are all inlined. This enables the compiler to perform powerful compilation optimizations, such as common expression extraction, to further reduce validation latency.

7 Evaluation

We implemented PICKER with 13 K lines of Python for the static analyzer and 1 K lines of C++ for the launch-time validator. PICKER is designed to analyze GPU kernel assembly code rather than source code, allowing it to handle closed-source kernels (e.g., from cuDNN [16]) and dynamically generated ones (e.g., from PyTorch [71]). PICKER currently supports the ISA of NVIDIA Volta and Ampere GPUs (e.g., GV100 and RTX 3080 in our experiments). We omit similar results on RTX 3080 due to space limitation. The symbolic execution engine is based on claripy [79] and Z3 [23].

7.1 Experimental Setup

Testbed. All experiments were conducted on a PC equipped with one Intel i7-13700 CPU, 32 GB of DRAM, and one NVIDIA GV100 GPU. We ran the experiments in a container based on an official Docker image from NVIDIA, which had CUDA 11.4.2, cuDNN 8.0, and Ubuntu 20.04 installed.

Table 3: *The launch-time validation results of idempotency for GPU kernels and instances in GPU applications. The columns ●, ○, and ◐ indicate that the GPU kernel is idempotent, non-idempotent, and conditionally-idempotent, respectively. The columns ★ and ☆ indicate that the GPU kernel instance is idempotent and non-idempotent, respectively.* **F+** *and* **F–** *indicate false positives and false negatives, respectively.*

GPU Apps	#Kernels	#Instances	Manual			Kernel-level			Truth		Instance-level			
			●	○	◐	●	○	◐	★	☆	★	☆	F– (%)	F+
Rodinia [14]	40	4,527	7	12	21	4	23	13	4,419	108	4,033	494	386 (8.74)	0
Parboil [85]	25	1,033	4	12	9	1	15	9	295	738	222	811	73 (24.75)	0
TVM [91]	308	609	0	0	308	0	0	308	609	0	596	13	13 (2.13)	0
PyTorch [71]	66	1,570	3	1	62	3	22	41	1,282	288	824	746	458 (35.73)	0
TensorRT [64]	58	478	0	2	56	0	17	41	465	13	267	211	198 (42.58)	0
FT [67]	50	10,000	9	7	34	8	19	23	7,348	2,652	5,803	4,197	1,545 (21.03)	0
All	547	18,217	23	34	490	16	96	435	14,418	3,799	11,745	6,472	2,673 (18.54)	0

Table 4: *The breakdown of GPU kernels that* PICKER *identifies as non-idempotent for various reasons. The columns ○ indicates that the kernel is non-idempotent.* **IF**, **PE**, **NA**, *and* **SO** *are abbreviations for "indirect function call", "path explosion", "non-parameter address", and "symbolic overlap", respectively.*

GPU Apps	#Kernels	○	IF	PE	NA	SO
Rodinia [14]	40	23	7	2	3	11
Parboil [85]	25	15	1	1	3	10
TVM [91]	308	0	0	0	0	0
PyTorch [71]	66	22	0	16	5	1
TensorRT [64]	58	17	0	12	4	1
FT [67]	50	19	0	12	4	3
All	547	96	8	43	19	26

Table 5: *The breakdown of GPU kernel instances that* PICKER *incorrectly validates as non-idempotent (false negatives) for various reasons.* **F–** *indicates false negatives in the validation of GPU kernel instances.* **IF**, **PE**, **NA**, **NC**, *and* **RO** *are abbreviations for "indirect function call", "path explosion", "non-parameter address", "non-parameter condition", and "range overestimation", respectively.*

GPU Apps	F–	IF	PE	NA	NC	RO
Rodinia [14]	386	44	15	71	1	255
Parboil [85]	73	0	12	59	0	2
TVM [91]	13	0	0	3	0	10
PyTorch [71]	458	0	119	314	1	24
TensorRT [64]	198	0	117	62	0	19
FT [67]	1,545	4	263	1,214	64	0
All	2,673	48	526	1,723	66	310

Workloads. We evaluated the correctness and efficiency of PICKER using GPU kernels from two GPU benchmarks, Rodinia [14] and Parboil [85], and four DL frameworks, TVM [91] v0.11.0, PyTorch [71] v1.12.1, TensorRT [64] v7.2.3.4, and FasterTransformer [67] v5.3 (referred to as FT). For Rodinia, we use kernels and instances from all applications, except for `mummergpu`, `cfd`, `particlefilter`, `srad`, and `streamcluster`, as these cannot run on our testbed. For Parboil, we use kernels and instances of the CUDA-based implementation. For TVM, PyTorch, and TensorRT, we use kernels and instances for inference tasks of five DNN models, ResNet [34], MobileNet [38], Inception [87], DenseNet [39], and VGG [83]. As for FasterTransformer, we use kernels and instances for an inference task of GPT-2 [74].

PICKER settings. In the static analysis phase, we set the maximum number of unroll iterations to 32 (see §5.2). As for the preconditions of the kernel parameters utilized for monotonicity checking (see §6.1), we establish bounds for each kernel parameter by utilizing the minimum and maximum values observed in the evaluated instances.

7.2 Validation Accuracy

Methodology. We evaluate the accuracy of PICKER in classifying the idempotency of GPU kernel instances (see Table 3). First, we employ the idempotency tracing tool (refer to §3) to gather the ground truth idempotency (**Truth**) and instances' launch arguments. We then label the kernel-level idempotency (**Manual**) as the same way mentioned in §3. Next, we utilize PICKER to statically analyze the kernel-level idempotency of each GPU kernel (**Kernel-level**). Finally, we employ PICKER to dynamically validate the idempotency of instances using the launch arguments provided in the trace and find incorrect classifications relative to the ground truth (**Instance-level**).

Ground truth. We totally evaluate 547 GPU kernels with 18,217 instances (using their public traces), where 14,418 instances are idempotent. All frameworks have both idempotent and non-idempotent GPU kernel instances, except for TVM.[5]

Kernel-level idempotency. As shown in the group **Kernel-level** of Table 3, most kernels (435 out of 547) are classified as cond-idempotent. PICKER classifies only 16 kernels as idempotent, as it requires a kernel to be write-only on GPU memory (e.g., for initialization), which is uncommon in practice. In contrast, 96 kernels are classified as non-idempotent by PICKER, surpassing the manually identified number of

[5]The TVM community has recently introduced support for in-place updates [19], enabling TVM to generate non-idempotent instances, though our evaluation encountered only idempotent instances.

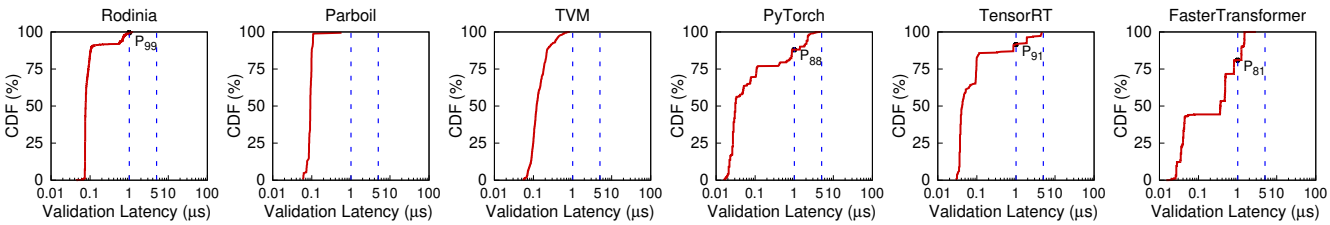

Fig. 5: *The CDF of validation latency for instances in various GPU benchmarks and DL frameworks.*

34. This suggests that some cond-idempotent kernels are pessimistically classified as non-idempotent.

Kernel-level classification breakdown. As shown in Table 4, we provide a detailed breakdown of why specific kernels are classified as non-idempotent, according to four reasons. Specifically, 26 kernels are normally classified as non-idempotent because PICKER can statically identify that they have load and store instructions accessing the same symbolic address (SO). There are 51 kernels classified as non-idempotent due to the failure of symbolic execution, including 8 kernels due to indirect function calls (IF), and 43 kernels due to path explosion (PE). Path explosion is frequently observed with irregular unbounded loops. The majority of these kernels are matrix multiplication kernels from cuDNN [16] and CUTLASS [24], featuring unbounded loops with complex control flows that PICKER cannot effectively optimize. Additionally, 19 kernels are classified as non-idempotent because they have symbolic addresses containing non-parameter variables, called non-parameter addresses (NA). Note that not all kernels with non-parameter addresses are classified as non-idempotent. Only those that have no path conditions to be validated at runtime are classified as non-idempotent, as these addresses will always overlap with others during validation.

Instance-level idempotency. PICKER successfully identifies 11,745 idempotent instances (with no false positives), and all non-idempotent instances. However, 2,673 instances are misclassified as non-idempotent, resulting in a false-negative rate of 18.54%. Instances from TVM exhibit the lowest false-negative rate, at 2.13%. This is attributed to the simplicity of the kernels produced by TVM, which has no unbounded loops that complicate analysis. On the other hand, other deep learning frameworks have significantly higher false-negative rates. For example, PyTorch has a false-negative rate of 35.73% and TensorRT has a rate of 42.58%.

Instance-level classification breakdown. We further analyze the detailed reasons for false negatives, reported in Table 5.[6] There are 574 false negative instances whose kernels are statically classified as non-idempotent due to the failure of static analysis (i.e., IF and PE). The primary cause of false negatives is non-parameter address (NA), accounting for 1,723 instances. As a reminder, PICKER adopts a conservative approach by assuming that a non-parameter variable can be any value, causing non-parameter addresses to always overlaps with other addresses during validation. Non-parameter variables can also exist in path conditions (non-parameter conditions, NC). PICKER conservatively assumes such conditions are always satisfied. This causes the predicted address set to exceed the actual one accessed by the instance, resulting in 66 false negatives. The range-based model in PICKER causes the remaining 310 false negatives, which can be correctly classified using our strawman solution. There are two main types of overestimation. The first type occurs when the concrete addresses of a symbolic address are discrete but PICKER considers them as consecutive ranges. The second type occurs when the range bounds are not tight enough.

[6]Detailed explanations of false negatives in PICKER can be found in §8.

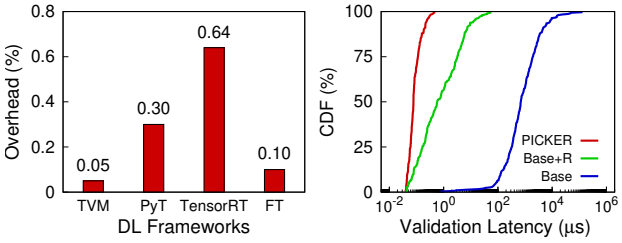

Fig. 6: *(a) The average overhead of launch-time validation on DL application performance, and (b) The CDF of validation latency for each instance in TVM using different optimizations.*

7.3 Validation Latency

Methodology. We measured validation latency for each instance, focusing only on kernels that PICKER classified as cond-idempotent. We excluded unconditionally idempotent and non-idempotent kernels since their validation functions simply return true or false without computation. For each instance, we ran its validation function 1,000 times and recorded the maximum execution time as the final validation latency.

Launch-time validation latency. Fig. 5 illustrates the CDF of the validation latency for each benchmark and framework. It is worth noting that PICKER is capable of validating all instances across the evaluated benchmarks and frameworks in under 5 µs, successfully achieving our ideal validation latency design goal (a few microseconds). Specifically, in Rodinia, PyTorch, TensorRT, and FasterTransformer, 99%, 88%, 91%, and 81% of the instances, respectively, and all instances in Parboil and TVM, can be validated within 1 µs.

Application overhead. We further evaluate the overhead to the application caused by PICKER, using inference tasks from four DL frameworks. The results, as shown in Fig. 6(a),

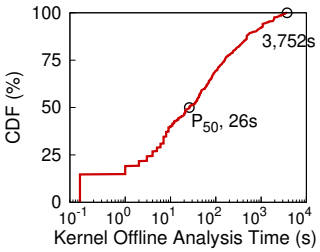

Fig. 7: *The CDF of offline analysis time for 547 total GPU kernels.*

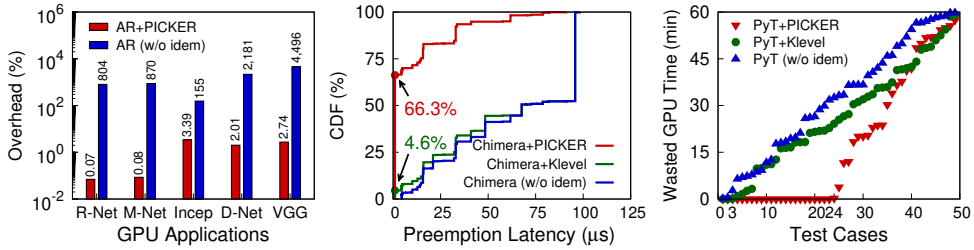

Fig. 8: *(a) The performance overhead of a checkpoint-based fault tolerant system, (b) The CDF of preemption latency in a preemptive scheduling system, and (c) wasted GPU time from recovering soft errors during GPT-2 training (60-minute checkpoint interval), with and without leveraging the idempotency information provided by* PICKER.

demonstrate that the overhead in terms of end-to-end execution time is less than 1% for all cases. This is because the end-to-end execution time of tasks is dominated by the execution time of kernels, and the launch-time validation of PICKER is sufficiently fast to overlap with it.

Optimization breakdown. To evaluate the efficacy of the optimizations proposed in §6, we measure the validation latency for three versions of PICKER: a fully optimized version (PICKER), a version without range compaction (Base+R), and a version without both range compaction and range-based address model (Base). All versions enable the compiled execution optimization. In this experiment, we specifically use instances from TVM, as all its kernels have no unbounded loops, allowing for effective analysis with our strawman solution. Fig. 6(b) illustrates the CDF of validation latency with different optimizations. The Base version, which enumerates all thread IDs for each symbolic address, generates excessive concrete addresses, causing 42% of instances to have validation latencies over 1 ms. Adding the range-based address model (§6.1) reduces each symbolic address to just two concrete address calculations, bringing the maximum validation latency below 100 µs. Finally, by applying range compaction (§6.2), PICKER further reduces the number of symbolic addresses, achieving a maximum validation latency under 1 µs.

Offline analysis time. Offline analysis time varies significantly across kernels, primarily depending on control-flow complexity and the size of the symbolic address space. Across all evaluated kernels, offline analysis time ranges from seconds to an hour (see Fig. 7). Aggregated over all kernels in our six evaluated applications, the total offline analysis time is 36 CPU-hours (one-time per kernel, amortized across all future runs). The static analysis phase of PICKER averaged 4 minutes per kernel across all evaluations. The most time-consuming case—a matrix multiplication kernel from cuDNN—required 62 minutes to analyze due to its complex control flow. This offline analysis is a one-time cost and does not impact the runtime performance.

7.4 Case Studies

We study three typical cases that improve the performance of idempotence-based systems using PICKER (see §2.1).

Case 1: checkpoint-based fault tolerant system. We implemented a simplified version of Asymmetric Resilience [51] (AR), a checkpoint-based GPU fault-tolerant system that leverages the idempotency to reduce the checkpointing overhead. Specifically, AR checkpoints the input buffer of every GPU kernel instance to the host memory before executing the instance. If a transient fault (e.g., a bit flip in the registers) occurs during the execution, the system is recovered by copying back the saved input data and re-executing the instance. For idempotent instances, AR avoids memory checkpointing since it can be recovered by re-execution.

Fig. 8(a) shows the performance overhead of AR on five TVM-backed DNN inference applications (as mentioned in §7.1). Without exploiting idempotency (AR w/o idem), the performance overhead is an impractical 155% to 4,496%. By integrating PICKER, the overhead drops to under 4% across all cases. For ResNet and MobileNet, PICKER correctly identifies all instances as idempotent, eliminating the need for checkpointing and resulting in minimal overhead (0.08%) from launch-time validation alone. The other three applications have misclassified instances (up to 6 in Inception), resulting in a maximum end-to-end overhead of 3.39%.

Case 2: preemptive scheduling of inference serving. We implemented an extended version of Chimera [70], a GPU multitasking system that leverages the idempotency information to accelerate preemptive scheduling. Specifically, if a kernel instance is non-idempotent, it is preempted by saving the execution context, generally resulting in preemption latency of tens of microseconds [58, 88]. In contrast, idempotent instances can be immediately preempted by killing them and discarding their execution context, thereby reducing the preemption latency to less than 1 µs [32, 49, 70, 78].

To evaluate the necessity of instance-level idempotency, we compare PICKER against a conservative kernel-level baseline (Klevel). Klevel treats a kernel as idempotent only if all its possible instances are idempotent; if a kernel is labeled as cond-idempotent, Klevel must conservatively treat every

instance as non-idempotent to avoid state corruption.

Fig. 8(b) shows the CDF of the preemption latency for GPT-2 inference requests. Each request consists of 10,000 instances involving 50 kernels [67]. PICKER identifies 5,803 instances as idempotent. The latency to kill a running instance is assumed to be 1 µs, following prior work [32, 49, 70]. Without exploiting idempotency (Chimera w/o idem), preemption latency ranges from 4 µs to 98 µs, depending on the context size of preempted instances, with an average of 64.3 µs. Because most GPT-2 kernels are labeled cond-idempotent at the kernel level, Chimera+Klevel provides negligible improvement, as it must save the context for nearly all instances. By integrating PICKER, 66.3% of instances are identified as idempotent, reducing the average latency by 84.2% to 10.2 µs.

Case 3: soft error correction. To evaluate PICKER in reducing recovery time from soft errors, we trained a GPT-2 model on an NVIDIA GV100 GPU using PyTorch and simulated soft errors by intercepting CUDA API calls and returning errors immediately after kernel launches. The faulty launch was selected randomly from all launches within a 60-minute checkpoint interval. By default, PyTorch handles such errors by terminating the process and rolling back to the most recent checkpoint, potentially wasting up to 60 minutes of computation. In contrast, PICKER determines whether the failed instance can be safely re-executed, allowing training to continue without a full rollback when the instance is idempotent.

A single training iteration in our GPT-2 setup consists of 61 distinct kernels with 1,289 kernel instances. Only 4 kernels can be manually annotated as idempotent at the kernel level, covering just 15 instances. In contrast, PICKER identified 803 instances as idempotent through launch-time validation. We evaluated three recovery approaches over 50 trials each. As shown in Fig. 8(c), vanilla PyTorch wastes a uniformly distributed amount of GPU time between 0 and 60 minutes due to rollback, while using only kernel-level idempotency information (PyTorch+Klevel) enables immediate recovery in only 3 of 50 trials. With the instance-level idempotency information provided by PICKER (PyTorch+PICKER), 24 out of 50 injected errors were corrected without rollback, significantly reducing wasted GPU time.

8 Discussions and Future Extensions

Analysis granularity: source vs. binary. A key design decision in PICKER is to perform analysis at the binary level (SASS) rather than the source level (CUDA C++). This approach addresses practical constraints in modern GPU ecosystems, where source-level analysis is often infeasible. High-performance applications increasingly rely on proprietary vendor libraries, such as cuDNN or TensorRT, for which source code is not publicly available. Furthermore, the prevalence of Just-In-Time (JIT) compilation in frameworks like PyTorch generates executable kernels at runtime, making static source-level verification impractical. By operating on kernel binary, PICKER achieves broad applicability across the entire software stack.

```
# Indirect func calls (IF)
*f(); # non-inline func

# Non-param address (NA)
p = *pp; # read param
*p = 0; # write non-param

# Non-param condition (NC)
if (*cp) # read non-param
  ...

# Range overestimation (RO)
# read p[1], p[3], ...
v = p[tid*2+1];
# write p[0], p[2],⋯
p[tid*2] = v;

# Memory access orders (MO)
*p = 0; # WRITE first
x = *p; # READ after WRITE
```

Fig. 9: *Examples of false-negative cases.*

More importantly, binary-level analysis is more faithful to the actual hardware behavior. High-level languages often rely on abstract memory models (e.g., `__restrict__`-qualifiers) that may be optimized or even disregarded by the compiler, leading to subtle pointer aliasing that is only visible in the final SASS instructions. Although binary analysis introduces complexities such as control-flow recovery, it is a necessary trade-off to satisfy our strict "no false positive" requirement (R1) where the source code is either absent or semantically disconnected from the actual device execution.

Nevertheless, source-level information can provide profound semantic insights to accelerate the offline analysis. For open-source kernels, developer-provided annotations (e.g., constant qualifiers) and compiler-generated metadata could be leveraged to prune symbolic execution paths or provide tighter bounds for induction variables. Integrating such source-level hints to further reduce false negatives remains a promising direction for our future work.

Multi-kernel idempotency. Although this paper focuses on single GPU kernel instances, PICKER can be extended to validate idempotency across multiple kernels. To support such multi-kernel validation at runtime, PICKER can intercept all kernel launches and maintain a table of running and pending instances together with their predicted read/write ranges. When a new instance is submitted, PICKER checks its predicted ranges against those of in-flight instances; any potential overlap is treated conservatively as non-idempotent for system-level actions that rely on idempotency. This conservative approach preserves correctness even when concurrency relationships are dynamic (e.g., overlapping CUDA streams). In addition, the host CPU typically interacts with GPU memory through explicit data-movement operations (e.g., `cudaMemcpyAsync`), which expose clear read/write ranges (source, destination, and size). PICKER can apply the same overlap checking by intercepting these operations, thereby accounting for host-side accesses in system-wide idempotency decisions.

False negatives and mitigation strategies. PICKER guarantees no false positives, though it may conservatively produce false negatives in certain cases. Fig. 9 summarizes representative patterns.

Indirect function call (IF). PICKER performs symbolic exe-

cution on GPU kernels offline, which currently does not support device functions invoked indirectly (e.g., via function pointers). A potential direction is to resolve indirect calls by enumerating feasible callees and analyzing each target, which we leave for future work.

Non-parameter address and condition (NA/NC). PICKER is most effective when memory addresses and key path conditions can be expressed as functions of launch arguments and thread identifiers. When an address or a condition depends on non-parameter state (e.g., values loaded from global/shared/constant memory), PICKER conservatively assumes the widest possible range, which may introduce false overlaps. One mitigation is to allow developers or frameworks to provide admissible value ranges for such variables, turning conservative unknowns into checkable launch-time constraints.

Range overestimation and memory-access orders (RO/MO). Range abstraction can over-approximate discrete address sets, and overlap checking ignores memory access order. Improving range precision (e.g., recognizing strided/discrete patterns) and incorporating ordering constraints under synchronization are promising directions to reduce false negatives without sacrificing correctness.

Non-GPU-memory states. While PICKER currently targets GPU kernels that only access GPU memory, it can be extended to support other system states, including CPU memory [43], persistent memory [68], and disk [82]. Since all these system states are also accessed through the virtual memory system, PICKER can handle them in a similar way.

Extensions of launch-time analysis. The ability to accurately and efficiently predict memory addresses of GPU kernel instances at launch time enables several powerful extensions. Our recent research, MSched [77] and PhoenixOS [97], leverage launch-time memory prediction to optimize data movement and persistence. MSched [77] utilizes these predictions to implement proactive memory page scheduling, significantly reducing the overhead of on-demand swapping in Unified Memory systems [9, 45]. PhoenixOS [97] identifies which memory pages will be modified (i.e., "dirty" pages) before a kernel runs, allowing the system to selectively checkpoint only the necessary data. Furthermore, PICKER can also be used to detect potential memory safety violations (e.g., buffer overflows [27, 31]) before kernel execution. This is particularly valuable in GPU environments, where memory errors often cause silent data corruption.

9 Related Work

Systems using idempotence. The idempotence property has been widely used in many systems, such as serverless computing [25, 42, 44, 73, 84, 101], persistency model [57, 100], storage and distributed systems [36, 48, 75, 81, 106] and intermittent computing [86, 93]. Flux [25] is one of these systems that are most related to PICKER, which can automatically verify the idempotency for a group of concurrent stateful serverless functions. The key difference between serverless functions and GPU kernels is the way they access state. Serverless functions access external databases using the table name and the record key, with the table name usually being a constant value, thereby making it possible to statically determine the idempotency. On the other hand, GPU kernels access GPU memory using pointers, with the base address typically being a parameter of the kernel and unknown before invocation. Therefore, PICKER chooses to dynamically validate the idempotency.

GPU program analysis. One of the key techniques behind PICKER is the automatic program analysis of GPU kernels. There are many existing research efforts on GPU program analysis for different purposes, such as bug detection [17, 53, 55], race detection [10–13, 18, 47, 56], memory coalescing detection [7, 8], test generation [52, 54] and security [61]. Honeycomb [61] is one of these systems most closely related to PICKER, a software-based TEE for GPU applications. Like PICKER, Honeycomb leverages static program analysis to validate the security of GPU kernels. Both PICKER and Honeycomb focus on validating memory address access at the instance level. However, there are two major technical differences. First, Honeycomb only validates that memory accesses fall within specified memory regions, while PICKER validates whether read and write accesses overlap. Second, Honeycomb uses a polyhedral model to represent potential memory addresses for instructions, while PICKER leverages monotonicity, which offers higher accuracy but requires an SMT solver.

10 Conclusion

This paper reveals that the majority of GPU kernels are conditionally-idempotent, meaning they possess both idempotent and non-idempotent instances depending on their input arguments. This finding exposes a critical limitation in prior work that statically classifies kernels, which either compromises system correctness or sacrifices performance.

To address this, we presented PICKER, the first launch-time validation system for instance-level idempotency. By combining offline symbolic execution with high-performance runtime validation, PICKER accurately identifies idempotent instances within microseconds. Our evaluations across 547 kernels and 18,217 instances demonstrate that PICKER achieves zero false positives with a low false-negative rate.

Acknowledgments

We sincerely thank our shepherd Minhui Xie and the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by the National Natural Science Foundation of China (No. 62432010, 62272291) and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM122).

References

[1] PyTorch Github Repository. https://github.com/pytorch/pytorch/blob/664058fa83f1d8eede5d66418abff6e20bd76ca8.

[2] Rodinia Website. https://rodinia.cs.virginia.edu/doku.php.

[3] TensorFlow Pre-trained MobileNetV1 Models. https://github.com/tensorflow/models/blob/505f9bf352d35700bf2596f7d9ce908881f81a07/research/slim/nets/mobilenet_v1.md.

[4] TorchVision. https://pytorch.org/vision/stable/models.html.

[5] *Induction Variables*, pages 35–58. Springer US, Boston, MA, 1995.

[6] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Timing of autonomous driving software: Problem analysis and prospects for future solutions. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 267–280, 2020.

[7] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. Gpudrano: Detecting uncoalesced accesses in gpu programs. In *International Conference on Computer Aided Verification*, 2017.

[8] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. Static detection of uncoalesced accesses in gpu programs. *Formal Methods in System Design*, 60:1 – 32, 2021.

[9] Pratheek B., Guilherme Cox, Ján Veselý, and Arkaprava Basu. Suv: Static analysis guided unified virtual memory. *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 293–308, 2024.

[10] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for gpu kernels. In *International Conference on Computer Aided Verification*, 2014.

[11] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for gpu kernels. *ACM Trans. Program. Lang. Syst.*, 37:10:1–10:49, 2015.

[12] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for gpu kernels. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2012.

[13] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of cuda programs. In *Third Workshop on Software Tools for MultiCore Systems*, volume 33, 2008.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[15] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2011.

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *ArXiv*, abs/1410.0759, 2014.

[17] Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamaric. Formal analysis of gpu programs with atomics via conflict-directed delay-bounding. In *NASA Formal Methods*, 2013.

[18] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering*, 40:710–737, 2014.

[19] TVM Community. The discussion of inplace update in dataflow block. https://discuss.tvm.apache.org/t/discuss-inplace-update-in-dataflow-block/14669, 2023.

[20] Alexander Craik, Yongtian He, and José Luis Contreras-Vidal. Deep learning for electroencephalogram (eeg) classification tasks: a review. *Journal of neural engineering*, 16 3, 2019.

[21] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 140–151, 2011.

[22] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[23] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2008.

[24] CUTLASS developers. Cutlass. https://github.com/NVIDIA/cutlass, 2021.

[25] Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. Automated verification of idempotence for stateful serverless applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 887–910, Boston, MA, 2023. USENIX Association.

[26] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[27] Christopher Erb, Mike Collins, and Joseph L. Greathouse. Dynamic buffer overflow detection for gpgpus. *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 61–73, 2017.

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

[29] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a machine learning-assisted kernel with lake. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023.

[30] Younghwan Go, Muhammad Asim Jamshed, Younggyoun Moon, Changho Hwang, and KyoungSoo Park. Apunet: Revitalizing gpu as packet processing accelerator. In *Symposium on Networked Systems Design and Implementation*, 2017.

[31] Yanan Guo, Zhenkai Zhang, and Jun Yang. GPU memory exploitation for fun and profit. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4033–4050, Philadelphia, PA, August 2024. USENIX Association.

[32] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[33] Ari B. Hayes, Fei Hua, Jin Huang, Yan-Hao Chen, and Eddy Z. Zhang. Decoding cuda binary. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 229–241, 2019.

[34] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[35] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama K. Govindaraju, Nishant Patil, and Yanjing Li. Understanding and mitigating hardware failures in deep learning training systems. *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023.

[36] Pat Helland. Idempotence is not a medical condition. *Communications of the ACM*, 55:56 – 65, 2012.

[37] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdelrahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[38] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

[39] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[40] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David R. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22:105–118, 2011.

[41] Won-Seok Jang, Hansaem Jeong, Kyungtae Kang, Nikil D. Dutt, and Jong-Chan Kim. R-tod: Real-time object detector with minimized end-to-end delay for autonomous driving. *IEEE Real-Time Systems Symposium (RTSS)*, pages 191–204, 2020.

[42] Abhinav Jangda, Donald Pinckney, Samuel Baxter, Breanna Devore-McDonald, Joseph Spitzer, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3:1 – 26, 2019.

[43] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023.

[44] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[45] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. Overlapping host-to-device copy and computation using hidden unified memory. *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.

[46] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using gpus in software packet processing. In *Symposium on Networked Systems Design and Implementation*, 2015.

[47] Aditya K. Kamath and Arkaprava Basu. iguard: In-gpu advanced race detection. *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[48] Seon Wook Kim, Chong liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst.*, 28:942–965, 2006.

[49] Hyeonsu Lee, Hyunjune Kim, Cheolgi Kim, Hwansoo Han, and Euiseong Seo. Idempotence-based preemptive gpu kernel scheduling for embedded systems. *IEEE Transactions on Computers*, 70:332–346, 2021.

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

[51] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran Monfort, Pradip Bose, Quan Chen, Minyi Guo, and Vijay Janapa Reddi. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 44–57, 2020.

[52] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh K. Gupta, Ranjit Jhala, and Sorin Lerner. Verifying gpu kernels by test amplification. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[53] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Fast Software Encryption Workshop*, 2010.

[54] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. Gklee: concolic verification and test generation for gpus. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2012.

[55] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in cuda programs. *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[56] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical symbolic race checking of gpu programs. *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 179–190, 2014.

[57] Zhen Lin, Mohammad A. Alshboul, Yan Solihin, and Huiyang Zhou. Exploring memory persistency models for gpus. *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 311–323, 2019.

[58] Zhen Lin, L. Nyland, and Huiyang Zhou. Enabling efficient preemption for simt architectures with lightweight context switching. *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 898–908, 2016.

[59] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr. *ACM Trans. Embed. Comput. Syst.*, 16(2), December 2016.

[60] Jeffrey Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. Understanding silent data corruption in llm training, 2025.

[61] HaoHui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. Honeycomb: Secure and efficient GPU executions via static validation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 155–172, Boston, MA, 2023. USENIX Association.

[62] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.

[63] NVIDIA. NVIDIA GPU Instruction Set Reference. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-reference.

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

[65] NVIDIA. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[66] NVIDIA. CUDA Samples. https://github.com/NVIDIA/cuda-samples, 2023.

[67] NVIDIA. FasterTransformer. https://github.com/NVIDIA/FasterTransformer, 2023.

[68] Shweta Pandey, Aditya K. Kamath, and Arkaprava Basu. Gpm: leveraging persistent memory from a gpu. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[69] Konstantinos Parasyris, Kai Keller, Leonardo Arturo Bautista-Gomez, and Osman Sabri Unsal. Checkpoint restart support for heterogeneous hpc applications. *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 242–251, 2020.

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

[71] Adam Paszke, S. Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, N. Gimelshein, L. Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[72] Reid Pinkham, Andrew Berkovich, and Zhengya Zhang. Near-sensor distributed dnn processing for augmented and virtual reality. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021.

[73] Sheng Qi, Xuanzhe Liu, and Xin Jin. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[74] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[75] G. Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. *Principles of Programming Languages (POPL)*, January 2013.

[76] Elvis Rojas, Diego Pérez, and Esteban Meneses. A characterization of soft-error sensitivity in data-parallel and model-parallel distributed deep learning. *J. Parallel Distrib. Comput.*, 190(C), August 2024.

[77] Weihang Shen, Yinqiu Chen, Rong Chen, and Haibo Chen. Towards fully-fledged gpu multitasking via proactive memory scheduling, 2026.

[78] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. XSched: Preemptive Scheduling for Diverse XPUs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 671–692, Boston, MA, July 2025. USENIX Association.

[79] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[80] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav S. Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads. *CoRR*, abs/2202.07848, 2022.

[81] Helgi Sigurbjarnarson, James Bornholt, Nicolas Christin, and Lorrie Faith Cranor. Push-button verification of file systems via crash refinement. In *USENIX Annual Technical Conference*, 2016.

[82] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. In *TOCS*, 2013.

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[84] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[85] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. 2012.

[86] Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/o dependent idempotence bugs in intermittent systems. *Proceedings of the ACM on Programming Languages*, 3:1 – 31, 2019.

[87] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[88] I. Tanasić, Isaac Gelado, Javier Cabezas, A. Ramírez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, 2014.

[89] Gemini Team. Gemini: A family of highly capable multimodal models, 2025.

[90] Llama Team. The llama 3 herd of models, 2024.

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

[92] Gaurang Upasani, Xavier Vera, and Antonio González. Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 37–48. IEEE Press, 2014.

[93] Joel van der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[94] Oreste Villa, Mark W. Stephenson, David W. Nellans, and Stephen W. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[95] Yuxuan Wang, RJ Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, et al. Tacotron: Afully end-to-end text-to-speech synthesis model. *arXiv preprint arXiv:1703.10135*, 2017.

[96] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[97] Xingda Wei, Zhuobin Huang, Tianle Sun, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. Phoenixos: Concurrent os-level gpu checkpoint and restore with validated speculation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 996–1013, New York, NY, USA, 2025. Association for Computing Machinery.

[98] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

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

[100] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. Scalable and fast lazy persistency on gpus. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 252–263. IEEE, 2020.

[101] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[102] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022.

[103] Yida Zhang and Changhee Jung. Featherweight soft error resilience for gpus. *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–262, 2022.

[104] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems*, 25:104–115, 2014.

[105] Wenxin Zheng, Bin Xu, Jinyu Gu, and Haibo Chen. Save: software-implemented fault tolerance for model inference against gpu memory bit flips. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[106] Siyuan Zhuang, Stephanie Wang, Eric Liang, Yi Cheng, and Ion Stoica. ExoFlow: A universal workflow system for Exactly-Once DAGs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 269–286, Boston, MA, 2023. USENIX Association.

Table 6: *The kernel-level idempotency of the evaluated GPU kernels using NVIDIA RTX 3080 GPU. The columns ●, ○, and ◐ indicate that the GPU kernel is idempotent, non-idempotent, and cond-idempotent, respectively. The columns ◐$_T$ indicates that the GPU kernel has both idempotent and non-idempotent instances from public application traces.*

GPU Apps	Code	#Kernels	●	○	◐	◐$_T$
Rodinia [14]	Source	40	7	12	21	2
Parboil [85]	Source	25	4	12	9	0
TVM [91]	Source	306	0	0	306	0
PyTorch [71]	Binary	43	5	3	35	5
TensorRT [64]	Binary	61	1	2	58	4
All		475	17	29	429	11

Table 7: *The instance-level idempotency of GPU kernel instances based on public application traces using NVIDIA RTX 3080 GPU. The columns ● / ★ (○ / ☆) indicates the idempotent (non-idempotent) instances of idempotent (non-idempotent) GPU kernels. The columns ◐ / ★ and ◐ / ☆ indicate the idempotent and non-idempotent instances of cond-idempotent GPU kernels, respectively. The kernel-level idempotency of each GPU kernel is the same as that in Table 6.*

GPU Apps	#Instances	● / ★	○ / ☆	◐ / ★	◐ / ☆
Rodinia [14]	4,527	85	78	4,334	30
Parboil [85]	1,033	103	738	192	0
TVM [91]	609	0	0	609	0
PyTorch [71]	1,796	95	18	1,238	445
TensorRT [64]	482	37	6	432	7
All	8,447	320	840	6,805	482

Appendix A Cond-idempotent GPU Kernels

We provide detailed information on four cond-idempotent GPU kernels, as shown in the column ◐$_T$ of Table 1. These GPU kernels have both idempotent and non-idempotent instances based on their public application traces.

A.1 Rodinia Benchmark

Version: Rodinia v3.1 [2]

A.1.1 Case 1: `bfs`

Application: bfs

Kernel name: `Kernel2`

Kernel source: `rodinia-root/cuda/bfs/kernel2.cu`

Execution method: We ran the application using the command in `rodinia-root/cuda/bfs/run`.

Number of idempotent instances: 1

Number of non-idempotent instances: 11

A.1.2 Case 2: `heartwall`

Application: heartwall

Kernel name: `kernel`

Kernel source: `rodinia-root/cuda/heartwall/kernel.cu`

Execution method: We ran the application using the command in `rodinia-root/cuda/heartwall/run`.

Number of idempotent instances: 19

Number of non-idempotent instances: 1

A.2 PyTorch Framework

Version: PyTorch v1.12.0 [1]

DL Models in ONNX Format: ResNet50, InceptionV3, VGG19, and DenseNet121 exported from TorchVision [4], MobileNetV1 exported from TensorFlow Model Garden [3].

Execution method: We ran the inference of the five models using an input tensor of zero values and a batch size of 32.

A.2.1 Case 3: `elementwise_kernel`

Kernel name: `elementwise_kernel`

Kernel source: `pytorch-root/aten/src/ATen/native/cuda/CUDALoops.cuh\#L132`

Number of idempotent instances: 5

Number of non-idempotent instances: 252

A.2.2 Case 4: `vectorized_elementwise_kernel`

Kernel name: `vectorized_elementwise_kernel`

Kernel source: `pytorch-root/aten/src/ATen/native/cuda/CUDALoops.cuh\#L58`

Number of idempotent instances: 1

Number of non-idempotent instances: 31

Appendix B A Study on RTX 3080 GPU

We also conducted the same experiments using an NVIDIA RTX 3080 GPU, with the same setup and workload as that of the §3 in our paper, except that FasterTransformer is not evaluated due to some limitations of our tracing tool. The main results are reported in Table 6 and Table 7. The experimental results on the two GPU benchmarks (Rodinia and Parboil) are the same as those using GV100 GPU in the paper. The main difference is that the DL frameworks (TVM, PyTorch, and TensorRT) use different GPU kernels for different hardware architectures.

Specifically, we found 5 more cond-idempotent GPU kernels in PyTorch, and 4 more cond-idempotent GPU kernels in TensorRT, further confirming the existence of cond-idempotent GPU kernels in real-world applications. Moreover, the observation of the prevalence of cond-idempotent GPU kernels and idempotent instances in real-world applications still holds on the RTX 3080 GPU, thereby further reinforcing the generality of our study results.