Hunting CUDA Bugs at Scale with cuFuzz

MOHAMED TAREK IBN ZIAD, NVIDIA, USA
CHRISTOS KOZYRAKIS, NVIDIA, USA and Stanford University, USA

GPUs play an increasingly important role in modern software. However, the heterogeneous host-device execution model and expanding software stack make GPU programs prone to memory-safety and concurrency bugs that evade static analyses. While fuzz-testing, combined with dynamic error checking tools, offers a plausible solution, it remains underutilized for GPUs. In this work, we identify three main obstacles limiting prior GPU fuzzing efforts: (1) kernel-level fuzzing leading to false positives, (2) lack of device-side coverage-guided feedback, and (3) incompatibility between coverage and sanitization tools. We present cuFuzz, the first CUDA-oriented fuzzer that makes GPU fuzzing practical by addressing these obstacles.

cuFuzz uses whole program fuzzing to avoid false positives from independently fuzzing device-side kernels. It leverages NVBit to instrument device-side instructions and merges the resultant coverage with compiler-based host coverage. Finally, cuFuzz decouples sanitization from coverage collection by executing host- and device-side sanitizers in separate processes. cuFuzz uncovers 43 previously unknown bugs (19 in commercial libraries) across 14 CUDA programs, including illegal memory accesses, uninitialized reads, and data races. cuFuzz achieves significantly more discovered edges and unique inputs compared to baseline approaches, especially on closed-source targets. Moreover, we quantify the execution time overheads of the different cuFuzz components and add persistent-mode support to improve the overall fuzzing throughput. Our results demonstrate that cuFuzz is an effective and deployable addition to the GPU testing toolbox. cuFuzz is publicly available at https://github.com/NVlabs/cuFuzz.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Software security engineering*; • **Computer systems organization** → Heterogeneous (hybrid) systems.

Additional Key Words and Phrases: CUDA, GPU, coverage-guided fuzzing, memory safety, data races

ACM Reference Format:
Mohamed Tarek Ibn Ziad and Christos Kozyrakis. 2026. Hunting CUDA Bugs at Scale with cuFuzz. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 123 (April 2026), 28 pages. https://doi.org/10.1145/3798231

1 Introduction

Graphics Processing Units (GPUs) play a critical role in modern computing. They serve as core components across diverse fields, from medical image processing [25] to large language models [39] and quantum computing [41]. As GPUs become larger and more complex, programming them becomes increasingly challenging and error-prone. Recent studies show that memory safety issues, plaguing CPUs for decades [42], also affect GPUs [11, 32, 36, 37]. While there are excellent tools to find memory safety bugs in GPU programs either statically [2, 3] or dynamically [18, 29, 43], these tools have their own limitations. Static tools suffer from high false positive rates whereas dynamic tools have limited scope—they only detect errors triggered by specific program inputs. When error-triggering inputs are not exercised, dynamic tools miss bugs entirely. A natural solution to this problem is fuzz-testing.

Authors' Contact Information: Mohamed Tarek Ibn Ziad, mtarek@nvidia.com, NVIDIA, Westford, MA, USA; Christos Kozyrakis, ckozyrakis@nvidia.com, NVIDIA, Santa Clara, CA, USA and Stanford University, Stanford, CA, USA.

ACM 2475-1421/2026/4-ART123
https://doi.org/10.1145/3798231

Fuzz-testing (or simply fuzzing) is a well-established CPU testing technique spanning decades [27]. It automatically generates random inputs to trigger unexpected program behaviors like crashes and assertions. Fuzzing proves highly effective for uncovering bugs in C/C++ programs [8], yet remains largely unexplored for GPUs. Traditional wisdom viewed GPUs as simple accelerators executing well-structured code requiring minimal testing. This perspective no longer holds. Modern GPUs support numerous features, primitives, and operations, making GPU workloads increasingly error-prone. However, naively applying fuzzing to GPUs faces three key challenges:

① **Fuzzing granularity**. GPU programs consist of host-side functions (running on the CPU) and device-side kernels (running on the GPU). Fuzzing can target whole programs or individual kernels. Prior work explored fuzzing individual device kernels by permuting input arguments to expose runtime bugs [22] or increase coverage [34]. However, kernel-level fuzzing generates false alarms by creating invalid input combinations that would never occur in complete program execution. For example, a host-side check can invalidate certain device-side input combinations, which kernel-level fuzzing cannot capture.

② **Lack of device-side coverage**. Coverage information is essential for guiding the fuzzer. State-of-the-art CPU fuzzers (like AFL++ [6]) use compiler-level instrumentation to collect edge coverage from control flow transitions in the target program. For GPUs, host-side coverage alone is insufficient as it misses critical device-side kernel behaviors. Further, obtaining device-side coverage presents two challenges. First, host and device code use different compilation toolchains (gcc/clang for host, nvcc/ptxas for device). Second, GPUs heavily rely on closed-source libraries [31] that cannot be recompiled for coverage instrumentation. A unified approach collecting coverage from both host and device sides is essential for effective GPU fuzzing.

③ **Tool incompatibility**. Effective fuzzing requires sanitizers for runtime error detection. While Google's AddressSanitizer (ASan) [38] dominates CPU testing, no ASan equivalent exists for CUDA due to closed-source GPU libraries and compilers. NVIDIA's Compute Sanitizer [29] serves as the alternative, using dynamic binary instrumentation to detect memory safety violations, data races, and uninitialized memory usage. Unfortunately, these sanitizers are neither compatible with each other nor compatible with the coverage collection tools due to shared GPU runtime API dependencies. Hence, coverage collection and sanitization cannot coexist in the same process. This incompatibility severely limits fuzzing effectiveness for GPU code. For example, prior GPU fuzzing work omits sanitizers entirely [22, 34].

Our work. We propose cuFuzz, the first end-to-end CUDA-oriented fuzzer (Figure 1). cuFuzz addresses the aforementioned challenges as follows. For challenge ①, cuFuzz operates at the whole program level, avoiding false alarms from kernel-level input permutation while preserving inter-kernel and host-device dependencies. For challenge ②, cuFuzz leverages NVBit [45], a dynamic binary instrumentation tool, to collect device-side coverage at runtime. cuFuzz merges this with host-side coverage to guide comprehensive fuzzing. For challenge ③, cuFuzz decouples sanitization from coverage collection by executing sanitizers in separate processes within the fuzzing loop.

Implementation. We implement cuFuzz using AFL++ [6] and NVIDIA's NVBit [45]. We evaluate cuFuzz on 14 diverse CUDA programs from HeCBench [15, 16] and CUDALibrarySamples [30]. Our workloads span open-source programs and closed-source libraries (nvTIFF, nvJPEG, nvJPEG2000, cuDNN, cuBLAS). We perform a thorough evaluation of cuFuzz to assess its bug finding capabilities, edge and input coverage, performance, sanitization, and efficiency.

Results. From a bug-finding perspective, at the time of writing, cuFuzz discovered 43 previously unknown bugs, including 19 in production libraries. These bugs range from memory safety violations (host-side heap overflows, device-side illegal memory accesses) to uninitialized memory reads and data races. We reported all bugs to the corresponding developers who acknowledged them. So far, 40 of 43 bugs have been fixed by the libraries' maintainers in latest releases.

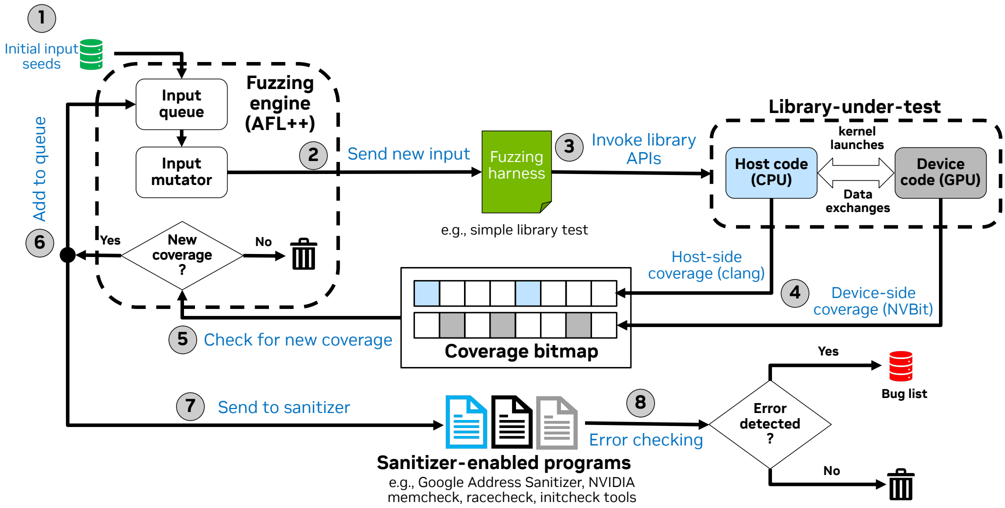


Fig. 1. cuFuzz high-level overview showing the integration of AFL++ with NVBit for device-side coverage and various sanitizers (e.g., Compute Sanitizer's memcheck, racecheck, initcheck) for GPU error detection.

To better understand cuFuzz's individual component contributions, we conduct isolated fuzzing campaigns with only a subset of cuFuzz's components enabled. We observe that cuFuzz achieves significantly more discovered edges and unique inputs compared to vanilla AFL++, especially on closed-source libraries. Additionally, cuFuzz's ability to run various incompatible sanitizers allows it to discover more bugs than AFL++.

Furthermore, we quantify cuFuzz's execution overheads by measuring the throughput of its individual components. We observe that device-side coverage collection (NVBit) reduces throughput by 67% whereas host-side coverage collection and sanitization have negligible impact on throughput. On the other hand, device-side sanitization has a different impact on throughput depending on the enabled tool with memcheck, racecheck, and initcheck reducing the throughput by 39%, 66%, and 32%, respectively. Based on this analysis, we share the following recommendations. First, since device-side coverage most benefits closed-source libraries, we recommend disabling it when whole-program recompilation is feasible to avoid the NVBit overheads. Second, to reduce the execution overheads of device-side sanitization, we leverage the insights of a recent work [20] to only run the sanitizers on a subset of inputs (e.g., inputs triggering unique execution paths).

Our experiments show that CUDA fuzzing is throughput limited. To alleviate this problem, we add support for persistent mode fuzzing in cuFuzz. Persistent mode improved performance (i.e., achieved higher coverage in less time) for 50% of the workloads, discovered the second-highest number of bugs (35 out of 43), and found 16 of them faster than all other cuFuzz configurations. Finally, on the nine open-source benchmarks where kernel-level fuzzing is applicable, cuFuzz's whole-program approach demonstrates clear advantages: kernel-level fuzzing found only 6 of 14 device-side bugs (missing 8 due to violated host-enforced invariants) while producing 16 false positives, whereas cuFuzz found all 14 device-side bugs plus 10 additional host-side bugs with zero false positives. These results demonstrate cuFuzz's value as a practical GPU testing tool.

2 Background

This section provides background on GPU architecture and fuzzing. While our work applies to any GPU and fuzzing engine, we focus on NVIDIA GPUs and the American Fuzzy Lop (AFL) due to their widespread adoption.

2.1 GPU Architecture

NVIDIA GPUs use the CUDA programming model, which has the following characteristics.

2.1.1 Programming Model. A CUDA program consists of host-side code (running on the CPU) and device-side code (running on the GPU). Host-side code comprises regular C++-like functions, while device-side code consists of special GPU functions called kernels. GPU kernels spawn thousands of threads—the smallest execution units on the GPU. Each thread has its own thread ID and execution context. Threads are organized into warps of 32 threads that execute in Single Instruction, Multiple Threads (SIMT) style on streaming multiprocessors (SMs), the primary programmable units on NVIDIA GPUs. Each SM provides a large register file (e.g., 256 KB) and an L1 cache (e.g., 96 KB) configurable as on-chip shared memory, accessible only to threads on that SM. All SMs connect to a unified L2 cache, which links to multiple memory controllers for high-bandwidth DRAM access.

2.1.2 Memory Spaces. GPUs have multiple memory spaces: `local` memory (available only to individual threads), `shared` memory (shared between threads on the same SM), and `global` memory (accessible to all GPU threads). Additional memory spaces include constant memory and texture memory, both limited to read-only access and used for specialized data access patterns and graphics rendering, respectively.

2.2 Fuzzing and Sanitization

Fuzzing is a widely used software testing method. It generates inputs (based on pre-defined rules or random strategies) and feeds them to the target program. The program executes with these inputs while being monitored for crashes or abnormal outputs. This process repeats with newly generated inputs for extended periods (hours or days). Inputs causing crashes are saved for analysis to identify vulnerability root causes. Fuzzing can be divided into three main categories: *black-box fuzzing* (no knowledge of program internals, monitoring only outputs), *white-box fuzzing* (complete program knowledge, using techniques like taint analysis or symbolic execution), and *grey-box fuzzing* (partial program knowledge, using code coverage to guide input generation).

2.2.1 AFL++. AFL++ is the state-of-the-art coverage-guided grey-box fuzzer [6]. This community-driven fuzzer has been extensively used in academia and industry. Starting with a fuzzing harness and initial seed inputs, AFL++ uses multiple mutation strategies (byte flips, arithmetic operations, input merging, dictionary-based mutations) to mutate program inputs. AFL++ uses compiler instrumentation to track branch (edge) coverage between basic blocks. At runtime, it stores executed edges for each input in a coverage bitmap. This bitmap is compared against a reference bitmap storing all previously seen edges to identify inputs triggering new code paths. Inputs that lead to new edges are saved in the fuzzing queue for further mutation. While AFL++ has a QEMU mode for closed-source CPU library fuzzing, it lacks CUDA program support, making host-side code instrumentation the only feasible option when fuzzing GPU programs.

2.2.2 Sanitizers. Fuzzing typically involves running error checking tools (also known as sanitizers) to detect errors that do not cause program crashes. State-of-the-art sanitizers include Google's AddressSanitizer (ASan [38]), a compiler-based tool checking for memory safety violations like buffer overruns and use-after-free in C/C++ code, and NVIDIA's Compute Sanitizer [29], a dynamic binary instrumentation tool detecting various device-side errors. The tools supported by Compute Sanitizer include memcheck for catching memory safety errors, initcheck for detecting uninitialized memory accesses, and racecheck for detecting data races, mainly in shared memory space.

Listing 1. CUDA motivating example.

```
 1  // Host-side validation function
 2  bool validateInputs(int n, float* a, float* b) {
 3      if (n <= 0 || n > MAX_VECTOR_SIZE) return false;
 4      if (!a || !b) return false;
 5      return true;
 6  }
 7
 8  // Device kernel with off-by-one bounds bug
 9  __global__ void vectorAdd(float* a, float* b, float* c, int n) {
10      int idx = blockIdx.x * blockDim.x + threadIdx.x;
11      if (idx <= n) { // BUG: off-by-one allows idx == n
12          c[idx] = a[idx] + b[idx];
13          // Complex device-side logic with edge cases
14          if (c[idx] > 1e6) {
15              // Edge case: large values trigger special processing
16              c[idx] = processLargeValue(c[idx]);
17          }
18      }
19  }
20
21  __global__ void clamp(float* x, int n, float lo, float hi) {
22      int idx = blockIdx.x * blockDim.x + threadIdx.x;
23      if (idx < n) {
24          float v = x[idx];
25          if (v < lo) v = lo;
26          if (v > hi) v = hi;
27          x[idx] = v;
28      }
29  }
30
31  __global__ void scaleInPlace(float* x, int n, float factor) {
32      int idx = blockIdx.x * blockDim.x + threadIdx.x;
33      if (idx < n) {
34          x[idx] *= factor;
35      }
36  }
37
38  __global__ void checksumKernel(const float* x, int n,
39      unsigned long long* out) {
40      int idx = blockIdx.x * blockDim.x + threadIdx.x;
41      if (idx < n) {
42          // Simple checksum: sum of absolute integer parts
43          unsigned long long v =
44                  (unsigned long long)fabs((double)x[idx]);
45          atomicAdd(out, v);
46      }
47  }
48  // Program entry point
49  int main(int argc, char** argv) {
50      // Read program inputs (n, and host buffers a, b). Details elided.
51      int n = /* read from input */ 0;
52      float *a = /* read host buffer */ nullptr;
53      float *b = /* read host buffer */ nullptr;
54      float *c = (float*)malloc((size_t)n * sizeof(float));
55
56      // Host-side validation
57      if (!validateInputs(n, a, b)) {
58          return 1;
59      }
60
61      // Allocate device memory
62      float *d_a, *d_b, *d_c;
63      cudaMalloc(&d_a, (size_t)n * sizeof(float));
64      cudaMalloc(&d_b, (size_t)n * sizeof(float));
65      cudaMalloc(&d_c, (size_t)n * sizeof(float));
66
67      unsigned long long *d_checksum;
68      cudaMalloc(&d_checksum, sizeof(unsigned long long));
69      cudaMemset(d_checksum, 0, sizeof(unsigned long long));
70
71      // Copy data to device
72      cudaMemcpy(d_a, a, (size_t)n * sizeof(float),
            cudaMemcpyHostToDevice);
73      cudaMemcpy(d_b, b, (size_t)n * sizeof(float),
            cudaMemcpyHostToDevice);
74
75      // Launch a sequence of kernels
76      int blockSize = 256;
77      int gridSize = (n + blockSize - 1) / blockSize;
78      // Preprocess inputs on device
79      clamp<<<gridSize, blockSize>>>(d_a, n, -1e3f, 1e3f);
80      scaleInPlace<<<gridSize, blockSize>>>(d_b, n, 0.5f);
81      // Core computation
82      vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
83      // Postprocess/validate result
84      checksumKernel<<<gridSize, blockSize>>>(d_c, n, d_checksum);
85      // Copy result back
86      cudaMemcpy(c, d_c, (size_t)n * sizeof(float),
            cudaMemcpyDeviceToHost);
87
88      // Cleanup
89      cudaFree(d_a); cudaFree(d_b);
90      cudaFree(d_c); cudaFree(d_checksum);
91      free(c);
92      return 0;
93  }
```

3 Motivation

This section motivates the need for GPU-aware fuzzing by highlighting the key challenges in GPU-based testing. We illustrate these challenges using a simple CUDA example (Listing 1) containing four device-side kernels (`clamp`, `scaleInPlace`, `vectorAdd`, `checksumKernel`) and one host-side function (`validateInputs`). The `main` function accepts two data buffers and their sizes as inputs. It validates inputs using `validateInputs`, allocates device memory, launches the four kernels, and copies the final checksum back to the host. This example demonstrates GPU program complexity and why traditional fuzzing approaches prove insufficient.

3.1 Kernel-Level versus Whole-Program Fuzzing

Prior work explored fuzzing individual device-side kernels by permuting input arguments. For example, the `vectorAdd` kernel can be fuzzed by independently permuting buffers `a`, `b`, and argument `n`. However, this approach generates false alarms by creating invalid input combinations that never occur in complete program execution. This can be demonstrated by two examples: (1) Providing NULL pointers causes illegal memory access in line 12, but host-side validation (line 4) already prevents this. (2) Providing negative n values causes out-of-bounds access, but input validation (line 3) blocks this scenario. In short, kernel-level fuzzers, which ignore host-side logic, will incorrectly flag issues that never occur in full programs. Conversely, kernel-level fuzzing misses bugs whose root cause lies in host-device interactions, e.g., when unchecked CUDA API failures cause device-side errors that cannot be reproduced in isolation.

A whole-program fuzzer is essential for uncovering realistic bugs in CUDA programs while avoiding false positives from kernel-level input permutation.

3.2 Lack of Device-Side Coverage

Coverage information plays a crucial role in guiding fuzzing. While host-side coverage can indicate device-side execution (when host functions launch device kernels), it may not capture all unique device-side executions. Consider the `vectorAdd` kernel in Listing 1. Random inputs with valid `n`, `a`, `b` values exercise most host-side code and will accidentally cover some device-side conditions (e.g., line 11). However, lines 14–17 contain conditional logic, `processLargeValue()`, which is only executed under specific GPU conditions. Host-side fuzzing alone cannot exercise these GPU-only paths since it is unaware of the condition at line 14.

An efficient GPU fuzzer must gather coverage data (such as executed edges) from both host- and device-side code.

3.3 Incompatibility Between Different Fuzzing Components

Sanitizers are key fuzzing components that uncover issues missed during baseline execution. The `vectorAdd` kernel in Listing 1 contains a bug in line 11: `if (idx ≤ n)` should be `if (idx < n)`. This creates an off-by-one error in line 12. The overflow is too small to trigger a GPU runtime segmentation fault, so it goes unnoticed even with Google's AddressSanitizer, which only validates host-side memory accesses. However, NVIDIA Compute Sanitizer's memcheck tool easily captures this overflow by tracking exact buffer bounds and validating all device memory accesses. Unfortunately, enabling device-side sanitizers is challenging since they depend on the same runtime APIs as other GPU tools (NVBit, cuda-gdb). Only one tool can be enabled simultaneously.

Running device-side sanitizers during fuzzing is strongly recommended to maximize benefits, but they conflict with each other and with coverage collection tools.

3.4 Throughput

While whole-program fuzzing offers significant benefits, it incurs substantial costs. CUDA runtime initialization creates a major bottleneck, adding overhead to each input trial. Additionally, enabling multiple sanitizers within the fuzzing loop further increases runtime costs, resulting in very low throughput as will be demonstrated in our experiments.

An efficient GPU fuzzer should maximize throughput while strategically enabling sanitizers to catch more errors.

4 cuFuzz High-Level Design

This section presents cuFuzz, the first CUDA-oriented fuzzer. The description here is tool-agnostic as implementation details are provided in Section 5.

4.1 Overview

Figure 1 shows cuFuzz's high-level overview. cuFuzz comprises four main components: (a) a fuzzing harness (the fuzzing entry point), (b) a fuzzing engine (generates and selects new inputs), (c) coverage collection (gathers coverage information from the program under test), and (d) sanitization (performs error detection). While these components exist in any fuzzing framework, cuFuzz uniquely positions them to address the challenges discussed in Section 3.

4.2 Whole Program Fuzzing

To avoid false positives from permuting individual kernel inputs without considering inter-kernel dependencies or host-side code, cuFuzz operates at the whole program level. Our fuzzing harness targets entire CUDA programs rather than single kernels. For CUDA applications, we use the application's main function as the harness. For closed-source CUDA libraries, we use accompanying examples as harnesses (e.g., the nvTIFF-Decode-Encode example from CUDALibrarySamples [30]).

The fuzzing harness is instrumented according to the fuzzing engine (e.g., AFL++ or LibFuzzer) to update coverage information. This instrumentation occurs at compile time (when source code is available) or execution time (for closed-source libraries). Using whole programs as harnesses enables capturing critical bugs arising from CUDA API interactions or shared kernel state (e.g., races occurring when memory accesses from different kernels touch the same memory location with one of them being a write).

4.3 Device-Side Coverage Collection

For closed-source library fuzzing, the harness only contains the API calls without information about the library internals. For instance, the nvTIFF library example has a single encode API triggering multiple kernel launches: `compactStrips_k`, `exsumMax_1blk_k`, `compressStrips_k`, and `batchedCopyLittleEndian_k`, as revealed by dynamic binary instrumentation tools [45]. Consequently, host-side coverage alone cannot adequately guide fuzzing. cuFuzz collects device-side coverage at runtime using NVBit [45], a dynamic binary instrumentation tool for GPUs.

For host-side coverage, cuFuzz relies on standard AFL++ compile-time instrumentation. However, device-side collection poses challenges due to inherent differences between host and device execution models. Host code's single-threaded nature enables simple coverage data structure updates via write operations. On the device side, thousands of threads may execute the same branch instruction simultaneously, attempting to update the same coverage entry. cuFuzz adopts several optimizations for this scenario, discussed in Section 5. Additionally, collisions may occur between host-side coverage entries (assigned at compile time) and device-side entries (assigned at dynamic load time). cuFuzz addresses this problem by assigning disjoint regions in the coverage data structure for host and device code.

4.4 Decoupling Host- and Device-Side Sanitization

To address incompatibility issues between tools involved in fuzzing (e.g., NVBit and Compute Sanitizer), cuFuzz decouples sanitization from coverage collection. We run separate processes for each desired sanitizer within the fuzzing loop, in addition to the main fuzzing harness collecting coverage information. While this approach increases runtime costs (due to multiple processes), overheads are reduced by selectively running sanitizers on input subsets (e.g., inputs with unique execution paths), as will be shown in Section 7.

4.5 Putting It All Together

We now put all components together by examining the complete fuzzing loop. Starting from Figure 1's top left, in step ❶, users provide a CUDA library to test, a sample program invoking it, and seed inputs (raw data files or images). Next, ❷ the fuzzing engine adds initial inputs to a queue and begins mutating them using its mutation strategies. ❸ Mutated inputs are passed to the fuzzing harness, which invokes library APIs.

At runtime, ❹ cuFuzz collects host-side coverage (executed edges) through traditional compile-time instrumentation and device-side coverage through custom dynamic binary instrumentation, storing both in the coverage data structure (coverage bitmap). In step ❺, the fuzzing engine compares the per-input coverage bitmap against a local copy, which tracks all previously explored edges. If the per-input bitmap contains no new entries, the input is discarded. However, if at least one new host- or device-side edge is detected, ❻ the corresponding input is added to the queue for further mutation. ❼ We also send this input to sanitizer-enabled versions of the fuzzing harness according to a pre-configured sanitization strategy. ❽ Finally, error-triggering inputs are saved to the crashes folder for inspection while benign inputs are discarded.

5 Implementation

This section describes cuFuzz's implementation details.

5.1 Fuzzing Loop

While cuFuzz's design from Section 4 can be built on top of any CPU fuzzer, our proof-of-concept implementation uses AFL++ as the fuzzing engine for input mutation. We also use AFL++'s compiler to build the fuzzing harness and insert necessary edge coverage instrumentation. For simplicity, we focus on mutating file-based inputs.

For example, when fuzzing a `medianfilter` program accepting a ".ppm" image and iteration count, we fix the iteration count at two and permute only the .ppm file contents. For programs not accepting input files (e.g., `convolution3D`, which accepts six integers: batch size, input channels, output feature maps, input width, input height, and kernel size), we modify the harness interface to read six integers from a file as a series. This preserves AFL++'s interface without modifications.

5.2 Coverage Collection

cuFuzz uses a two-pronged approach for coverage collection: standard AFL++ instrumentation for host-side code and a custom NVBit tool for device-side code. Host-side coverage requires no custom implementation as AFL++'s compiler inserts edge-tracking instrumentation during compilation. For device-side coverage, we built an NVBit tool to collect edge-coverage information from device-side code. NVBit is a dynamic binary instrumentation framework that patches device-side kernels at load time to enable dynamic instrumentation of device-side GPU code [45]. When the CUDA context is initialized (typically with the first kernel launch), our NVBit tool captures the AFL++ coverage bitmap address and allocates GPU memory buffers to track per-thread executed edges, similar to AFL++'s host-side coverage tracking.

At kernel load time, our NVBit tool replaces the first instruction of each basic block on the device-side with a jump instruction pointing to trampoline code. In the trampoline, NVBit saves the thread context, executes our coverage-collection instrumentation function (`cufuzz_cov_edge`), restores the program context, executes the original instruction, and jumps back. At the final CUDA context execution, our NVBit tool copies the device-side bitmap to the host and merges it with the host-side bitmap. Due to the inherent differences between host-side and device-side code, our tool addresses the following three challenges.

5.2.1 Handling Concurrent Updates. Listing 2 shows the host-side coverage collection instrumentation function pseudo-code, which XORs the current edge identifier with the previous edge identifier and updates the coverage data structure. Using the same function on the device side causes significant instability since multiple threads can execute it simultaneously. We address this issue using atomic updates as shown in line 16 of Listing 3.

Listing 2. AFL++ host-side instrumentation.

```
extern unsigned char* __afl_area_ptr; /* points to shared bitmap */
__thread uint32_t __afl_prev_loc = 0; /* per-thread edge history */

static inline void afl_cov_edge(uint32_t cur_loc) {

    uint32_t idx = __afl_prev_loc ^ cur_loc;

    /* never-zero increment of 8-bit counter */
    uint8_t v = __afl_area_ptr[idx];
    v++;
    __afl_area_ptr[idx] = v ? v : 1;

    __afl_prev_loc = cur_loc >> 1;
}
```

Listing 3. cuFuzz device-side instrumentation.

```
 1 extern "C" __device__ __noinline__ void cufuzz_cov_edge(
 2   int cur_loc, uint64_t __afl_area_ptr, uint64_t prev_loc) {
 3   const int active_mask = __ballot_sync(__activemask(), 1);
 4   const int laneid = get_laneid();
 5   const int first_laneid = __ffs(active_mask) - 1;
 6
 7   uint64_t tid = threadIdx.x + blockIdx.x * blockDim.x
 8     + threadIdx.y * blockDim.x * gridDim.x
 9     + blockIdx.y * blockDim.x * blockDim.y * gridDim.x
10     + threadIdx.z * blockDim.x * blockDim.y * gridDim.x * gridDim.y
11     + blockIdx.z * blockDim.x * blockDim.y * blockDim.z * gridDim.x *
           gridDim.y;
12   int idx = (MAP_SIZE/2) + ((prev_loc[tid] ^ cur_loc) % (MAP_SIZE/2));
13   prev_loc[tid] = cur_loc >> 1;
14   /* only the first active thread will perform the atomic update*/
15   if (first_laneid == laneid) {
16     atomicAdd((unsigned int*)&__afl_area_ptr[idx], 1);
17   }
18 }
```

Fig. 2. cuFuzz coverage instrumentation: (Left) AFL++ host-side edge tracking via compile-time instrumentation; (Right) NVBit device-side edge tracking via runtime instrumentation with warp-aware atomic updates.

5.2.2 Handling Bitmap Collisions. Our current prototype uses a 64 KB coverage bitmap where each host- or device-side edge is represented by a single byte. To avoid collisions between host- and device-side bitmap entries, host-side coverage entries use the first 32 KB while device-side coverage entries use the second 32 KB. We achieve this by starting device-side edges at `MAP_SIZE/2`, as shown in line 12 of Listing 3.

5.2.3 Handling Thread Count. In AFL++ terminology, the byte representing host-side edges in the coverage bitmap not only indicates whether an edge is executed or not, but also tracks execution frequency per input (enabling the identification of edges executed within loops). This count is bucketed to powers of two to avoid path explosion. An input is considered interesting (saved to the queue) if it explores at least one new bucket for an edge. We use the same approach on the device side by bucketing thread counts (how many threads executed the edge). To account for potentially massive GPU thread numbers, we use larger power-of-two buckets and warp-level (instead of thread-level) counts. As shown in line 16 of Listing 3, a value of one is used to increment the bitmap entry counter for the entire warp.

5.3 Sanitization

Inspired by recent work, SAND [20], we decouple coverage collection from host- and device-side sanitization in cuFuzz's fuzzing loop, avoiding incompatibility issues between NVBit and Compute Sanitizer. The original SAND implementation, now merged with AFL++, assumes each sanitizer-enabled program is instrumented at compile time with sanitizer instrumentation in addition to AFL++ server instrumentation. Since our main device-side sanitizer, NVIDIA Compute Sanitizer, relies on binary instrumentation, we create a wrapper around the program under test that can be compiled with SAND (to add AFL++ server instrumentation) and used to launch both Compute Sanitizer and the original program. While this approach increases execution time costs (due to multiple processes), overheads are amortized by selectively running sanitizers on input subsets during fuzzing, as will be quantified in Section 7.5.

5.4 Persistent Mode

AFL++ runs one test case per process by default. This avoids test case interference and improves error attribution. However, this creates large overhead for GPU fuzzing due to CUDA runtime's substantial initialization cost. One potential solution is to use AFL++'s persistent mode, allowing

Listing 4. Harness persistent mode modifications.

```
1  __AFL_FUZZ_INIT();
2
3  __global__ void cufuzz_notification_kernel(int signal_id) {
4      /* Empty kernel – just for NVBit signaling */
5      ;
6      /* Signal ID can be used to identify different events */
7  }
8
9  // Fuzzing harness entry point
10 main() {
11     /* program initialization. */
12     /* ... */
13
14     #ifdef __AFL_HAVE_MANUAL_CONTROL
15     __AFL_INIT();
16     #endif
17     unsigned char *buf = __AFL_FUZZ_TESTCASE_BUF;
18
19     while (__AFL_LOOP(1000)) {
20         int len = __AFL_FUZZ_TESTCASE_LEN;
21         if (len < 512) continue; /* check for a useful min. length */
22
23         /* write to output file for sanitization */
24         FILE *fp = fopen(fname, "wb");
25         if (fp != NULL) {
26             fwrite(buf, 1, len, fp); fclose(fp);
27         }
28         cufuzz_notification_kernel<<<1, 1>>>(1234); /* iteration start */
29         /* ... */
30         /* regular fuzzing harness code */
31         /* ... */
32         cufuzz_notification_kernel<<<1, 1>>>(5678); /* iteration end */
33     }
34     return 0;
35 }
```

Listing 5. cuFuzz persistent mode support in NVBit.

```
void nvbit_at_cuda_event(CUcontext ctx, int is_exit, nvbit_api_cuda_t
        cbid, const char *name, void *params, CUresult *pStatus) {
    /* Identify all the possible CUDA launch events */
    if (cbid == API_CUDA_cuLaunch ||
        cbid == API_CUDA_cuLaunchKernel_ptsz ||
        cbid == API_CUDA_cuLaunchGrid ||
        cbid == API_CUDA_cuLaunchGridAsync ||
        cbid == API_CUDA_cuLaunchKernel) {
        cuLaunch_params *p = (cuLaunch_params *)params;
        std::string kernel_name = nvbit_get_func_name(ctx, p->f, 1);
        if (kernel_name == "_Z19cufuzz_notification_kerneli" &&
                afl_persistent && !is_exit) {
            /* Check for a specific notification kernel */
            if (cbid == API_CUDA_cuLaunchKernel_ptsz ||
                cbid == API_CUDA_cuLaunchKernel) {
                cuLaunchKernel_params* p_kernel = (
                        cuLaunchKernel_params*)params;
                if (p_kernel->gridDimX == 1 &&
                    p_kernel->gridDimY == 1 &&
                    p_kernel->gridDimZ == 1 &&
                    p_kernel->blockDimX == 1 &&
                    p_kernel->blockDimY == 1 &&
                    p_kernel->blockDimZ == 1) {
                    uint32_t magic_value = *((uint32_t*)p_kernel->
                            kernelParams[0]);
                    if (magic_value == MAGIC_VALUE_START) {
                        /* reset device-side coverage bitmap */
                    } else if (magic_value == MAGIC_VALUE_END) {
                        /* merge device and host coverage bitmaps */
                    }
                }
            }
        }
    }
}
```

Fig. 3. cuFuzz persistent mode: (Left) Harness structure with AFL++ loop processing multiple inputs per process; (Right) State reset between iterations in NVBit.

the target program to run in a loop, executing one test case per iteration. The pseudo-code for fuzzing harness changes required for AFL++ persistent mode appears in Listing 4.

To enable persistent mode for cuFuzz, we have to address two challenges. First, we need a method to notify NVBit of loop boundaries so our custom tool can differentiate between test cases and update the device-side coverage bitmap accordingly. Second, we need to pass inputs from the persistent run to the separately running sanitizer processes.

We address the first challenge by calling an empty kernel `cufuzz_notification_kernel` (lines 28 and 32 in Listing 4) with a unique identifier at the beginning and end of the persistent loop. We modify our NVBit coverage tool to intercept these unique kernel invocations, check the input parameter, and take appropriate actions (i.e., reset the device bitmap at loop beginning and merge it with the host bitmap at loop end). We address the second challenge by writing the newly generated buffer `buf` to a file that is read by our sanitizer-enabled wrapper. We evaluate persistent mode coverage and bug finding benefits in Section 7.

6 Experimental Methodology

6.1 Benchmarks

We evaluate 14 programs from the heterogeneous computing benchmark suite HeCBench [15, 16] and CUDA library samples CUDALibrarySamples [30].[1] These benchmarks span diverse GPU application domains: *Machine Learning* (`cuDNN`, `attention`), *Math* (`cuBLAS`, `lud`), *Image Processing*

[1] For HeCBench, we use commit number: 37507304619620b716977d63829871b658465986. For CUDA libraries, we use the official build versions shown in Table 1.

Table 1. Summary of our benchmarks.

Fuzzing harness	Target library	Domain	Input format	Static binary size
attention	attention	Machine learning	4 integers	1.3 MB
boxfilter	boxfilter	Image processing	ppm	1.7 MB
crs	crs	Data encoding	2 integers	1.6 MB
dxtc2	dxtc2	Data compression	ppm	1.5 MB
lud	lud	Math	1 integer	1.2 MB
medianfilter	medianfilter	Image processing	ppm	1.4 MB
recursiveGaussian	recursiveGaussian	Image processing	ppm	1.5 MB
seam-carving	seam-carving	Image processing	jpg	2.0 MB
urng	urng	Random number generation	bmp	1.2 MB
nvtiff_example	libnvtiff-0.4.0.62	Image processing	tif	4.0 MB
nvjpegDecoder	libnvjpeg-12.4.0.33	Image processing	jpg	8.8 MB
nvjpeg2k_example	libnvjpeg2k-0.8.0.38	Image processing	jp2, j2k	6.4 MB
convolution3D	cuDNN-9.11.0	Neural Networks	6 integers	29 MB
blas-gemm	cuBLAS-12.9.1.4	Math	4 integers	465 MB

(`nvTIFF`, `nvJPEG`, `nvJPEG2000`, `boxfilter`, `medianfilter`, `recursiveGaussian`, `seam-carving`), *Data Compression/Encoding* (`dxtc2`, `crs`), and *Random Number Generation* (`urng`). Table 1 summarizes our benchmarks, including fuzzing harness, application or library name, domain, and input format. The table includes static binary size as a proxy for program complexity.

6.2 Tools

Our implementation uses AFL++ 4.31c, NVBit 1.7.5, and NVIDIA Compute Sanitizer 2025.2.0.0. All fuzzing harnesses are statically compiled using NVIDIA CUDA compiler (nvcc) 12.9.36 with clang 14 as the host compiler. For closed-source libraries (lower half of Table 1), we statically link fuzzing harnesses with the official pre-built versions shown in the table.

6.3 Infrastructure

We run experiments on two servers, each equipped with Intel(R) Xeon(R) Platinum 8362 CPU (64 cores, 2 threads per core) and 1008 GB memory. Each server contains eight NVIDIA A40 GPUs with 48 GB memory each. The servers run Ubuntu 22.04-x86_64-standard-uefi with NVIDIA driver 570.144 and CUDA 12.9 toolkit.

To minimize variability across runs, we set CPU clock speed to 3.2 GHz and GPU logic and memory clock speeds to 1740 MHz and 7200 MHz, respectively. We use separate Docker containers for each benchmark, with each container having exclusive access to 16 CPU cores, 120 GB memory, and a single A40 GPU. We run exactly one fuzzing harness per GPU to avoid context switching.

6.4 Fuzzing Configurations

To evaluate our tool's effectiveness, we run and compare the following configurations:

- *AFL++*: baseline approach with only host-side coverage collection enabled (through "afl-clang-fast" compiler instrumentation).
- *cuFuzz*: our comprehensive approach including AFL++ plus device-side coverage collection (through NVBit) and full sanitization (through Google's AddressSanitizer and NVIDIA Compute Sanitizer's memcheck, racecheck, and initcheck tools).
- *cuFuzz-noDeviceCoverage*: cuFuzz variant with device-side coverage collection disabled.
- *cuFuzz-noSanitizer*: cuFuzz variant with sanitization disabled (only host- and device-side coverage collection enabled).
- *cuFuzz-persistent*: cuFuzz variant with persistent mode enabled, including device-side coverage collection and sanitization.

We run these five configurations (plus three more described in Section 7.5) on our 14 benchmarks three times for 24 hours each, totaling 8,064 GPU hours, and report the best per-configuration result across the three runs. We use the same initial seed corpus per benchmark for all configurations. Seed counts range from 1 to 5, depending on each benchmark and its input format.

For sanitizer-enabled configurations (cuFuzz, cuFuzz-noDeviceCoverage, and cuFuzz-persistent), we feed all fuzzer-generated inputs to the fuzzing harness and only a subset of these inputs to the four sanitizers. The inputs we feed to the sanitizers are the ones that trigger at least one new edge in either host or device-side code. We compare different strategies for selecting the inputs to feed to the sanitizers in Section 7.5.

7 Evaluation

This section describes our cuFuzz evaluation. We first present our research questions, then detail the experimental results. Finally, we analyze our experimental results and answer each research question. Our evaluation was guided by the following research questions:

- **RQ1: Bug finding.** How effective is cuFuzz in finding bugs in the target libraries?
- **RQ2: Coverage.** How does cuFuzz's coverage compare to baseline AFL++ fuzzer coverage?
- **RQ3: Performance.** How is cuFuzz's performance impacted by its individual components?
- **RQ4: Persistent mode.** Does persistent mode improve cuFuzz's performance?
- **RQ5: Sanitization.** How does input selection for sanitization impact cuFuzz's coverage?
- **RQ6: Efficiency.** Which cuFuzz configuration provides the best time to exposure of bugs?
- **RQ7: Kernel-level fuzzing.** How does cuFuzz compare to kernel-level fuzzing approaches?

7.1 Bug Finding

A fuzzer's most important metric is its ability to find bugs in tested programs. We assess cuFuzz's bug-finding capability by running it for 24 hours on target programs from Table 1. Table 2 summarizes bugs found by cuFuzz, including target library, bug type, file or kernel name, bug category (host or device), used sanitizer, and status (pending, confirmed, fixed). We record issues as "confirmed" only when developers can reproduce them and as "fixed" when a pull request containing the bug fix is merged into the main branch as a result of cuFuzz's findings.

Figure 4 shows the distribution of bug types found by cuFuzz. cuFuzz discovers a wide range of bugs in both host- and device-side code: 8 heap buffer overflows, 1 stack buffer overflow, 2 floating-point exceptions, 2 segmentation faults, 13 out-of-bounds device accesses, 5 shared memory data races, and 12 uninitialized device reads. Notably, 19 of 43 bugs occur in commercial libraries, highlighting cuFuzz's effectiveness in bug detection. Next, we analyze some of the bugs found by cuFuzz in more detail.

7.1.1 Bug Sample (Id #2). Bug #2 is an invalid device-side global-memory read in the col_kernel function of the `boxfilter` benchmark. This benchmark accepts a ppm image as input and performs simple arithmetic operations on pixel values. Listing 6 shows the error reported by Compute Sanitizer's memcheck tool. As shown in Listing 7, the root cause is that col_kernel accesses the input image buffer (line 11) using a global index (line 8) without checking if the index is within buffer bounds. The fix adds a bounds check (line 9) ensuring only threads with indices smaller than the image width access the image buffer.

7.1.2 Bug Sample (Id #23). Bug #23 is another case of invalid device-side global memory reads. It impacts the compute_costs_kernel function of the `seam-carving` benchmark. This benchmark accepts a jpg image as input and performs the seam carving algorithm. Listing 8, which shows the error reported by Compute Sanitizer's memcheck tool, reveals that the invalid access is in fact

Table 2. List of bugs found by cuFuzz.

Bug#	Target library	Bug type	File/kernel name	Category	Sanitizer	Status
1	attention	Heap buffer overflow	reference.h:13	host	ASan	fixed
2	boxfilter	Invalid global read of size 4 bytes	col_kernel	device	memcheck	fixed
3	boxfilter	Uninitialized global read of size 4 bytes	row_kernel	device	initcheck	fixed
4	boxfilter	Uninitialized global read of size 4 bytes	col_kernel	device	initcheck	fixed
5	boxfilter	Heap buffer overflow	reference.cpp:121	host	ASan	fixed
6	crs	Floating-point exception	main.cu:81	host	crash	fixed
7	dxtc2	Invalid global read of size 4 bytes	compress	device	memcheck	fixed
8	dxtc2	Invalid global write of size 8 bytes	compress	device	memcheck	fixed
9	dxtc2	Data race in shared memory	compress	device	racecheck	fixed
10	dxtc2	Heap buffer overflow	shrUtils.cu:1122	host	ASan	fixed
11	lud	Invalid global read of size 4 bytes	lud_diagonal	device	memcheck	fixed
12	lud	Stack buffer overflow	common/common.cu:161	host	ASan	fixed
13	medianfilter	Invalid global write of size 4 bytes	ckMedian	device	memcheck	fixed
14	medianfilter	Uninitialized global read of size 4 bytes	ckMedian	device	initcheck	fixed
15	medianfilter	Data race in shared memory	ckMedian	device	racecheck	fixed
16	medianfilter	Heap buffer overflow	MedianFilterHost.cu:70	host	ASan	fixed
17	medianfilter	Heap buffer overflow	MedianFilterHost.cu:85	host	ASan	fixed
18	recursiveGaussian	Heap buffer overflow	shrUtils.cu:1302	host	ASan	fixed
19	recursiveGaussian	Invalid global read of size 4 bytes	RecursiveRGBA	device	memcheck	fixed
20	recursiveGaussian	Uninitialized global read of size 4 bytes	RecursiveRGBA	device	initcheck	fixed
21	recursiveGaussian	Uninitialized global read of size 4 bytes	Transpose	device	initcheck	fixed
22	recursiveGaussian	Heap buffer overflow	RecursiveGaussianHost.cu:180	host	ASan	fixed
23	seam-carving	Invalid global read of size 4 bytes	compute_costs_kernel	device	memcheck	fixed
24	urng	Heap buffer overflow	SDKBitMap.h:368	host	ASan	fixed
25	nvTIFF	Invalid global write of size 1 bytes	batchedCopyLittleEndian_k	device	memcheck	fixed
26	nvTIFF	Invalid global write of size 2 bytes	reshapeStrilesRGBInterleaved_k	device	memcheck	fixed
27	nvTIFF	Floating-point exception	StripedTiffImageFile	host	crash	fixed
28	nvTIFF	Race condition in shared memory	batchedLZWDecompress_k	device	racecheck	fixed
29	nvTIFF	Uninitialized global read of size 1 byte	compressStrips_k	device	initcheck	fixed
30	nvJPEG	Segmentation fault	libnvjpeg_static	host	crash	fixed
31	nvJPEG	Invalid global write of size 1 bytes	ycbcr_to_format_kernel_roi	device	memcheck	fixed
32	nvJPEG2000	Invalid global write of size 1 bytes	idwt	device	memcheck	fixed
33	nvJPEG2000	Invalid global write of size 1 bytes	lossy_mct_levelshift	device	memcheck	fixed
34	nvJPEG2000	Invalid global write of size 1 bytes	lossless_mct_levelshift	device	memcheck	fixed
35	nvJPEG2000	Uninitialized global read of size 4 bytes	idwt	device	initcheck	fixed
36	nvJPEG2000	Uninitialized global read of size 4 bytes	inv_quantize_partial	device	initcheck	fixed
37	nvJPEG2000	Uninitialized global read of size 4 bytes	tier1decodemultiple_channels_k	device	initcheck	fixed
38	nvJPEG2000	Uninitialized global read of size 4 bytes	htTier1MelAndVlcDecodeMultiple	device	initcheck	fixed
39	nvJPEG2000	Race condition in shared memory	tier1decodemultiple_channels_k	device	racecheck	fixed
40	nvJPEG2000	Segmentation fault	tier2Decode::decodeCPRL	host	crash	fixed
41	cuDNN	Race condition in shared memory	scudnn_winograd_128x128	device	racecheck	confirmed
42	cuDNN	Uninitialized global read of size 4 bytes	conv2d_grouped_direct_kernel	device	initcheck	confirmed
43	cuDNN	Uninitialized global read of size 4 bytes	implicit_convolve_sgemm	device	initcheck	confirmed

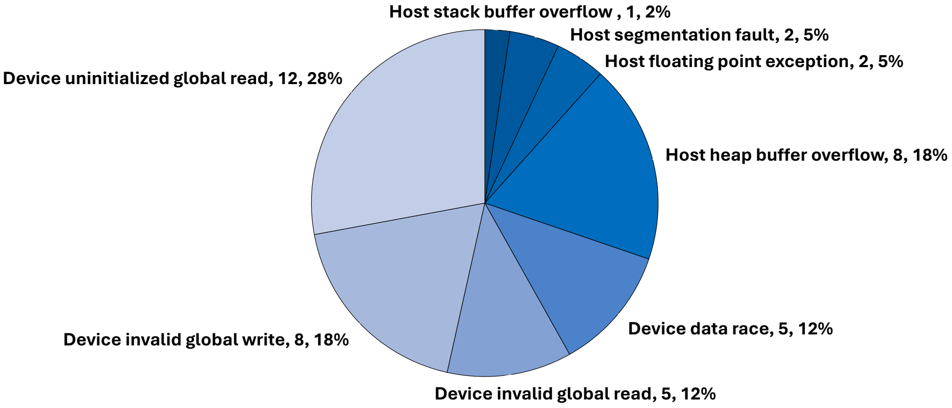


Fig. 4. Distribution of bug types found by cuFuzz.

Listing 6. Bug #2 memcheck report.

```
========= Invalid __global__ read of size 4 bytes
========= at col_kernel(const unsigned int *, unsigned int *, unsigned int, unsigned int, int, float)+0x490
========= by thread (1,0,0) in block (0,0,0)
========= Access at 0x7f71b9e01360 is out of bounds
========= and is 1 bytes after the nearest allocation at 0x7f71b9e00a00 of size 2400 bytes
========= Saved host backtrace up to driver entry point at kernel launch time
========= Host Frame: cudaLaunchKernel [0x86c87] in main
========= Host Frame: BoxFilterGPU(uchar4*, unsigned int*, unsigned int*, unsigned int, unsigned int, int, float, int) in main.cu:187 [0xbe55] in main
========= Host Frame: main in main.cu:236 [0xb23c] in main
```

Listing 7. Bug #2 location in col_kernel function.

```
__global__ void col_kernel (
    const uint* __restrict__ uiSource, uint* __restrict__ uiDest,
    const int uiWidth,
    const int uiHeight,
    const int iRadius,
    const float fScale)
{
    int globalPosX = blockIdx.x * blockDim.x + threadIdx.x;
    if (globalPosX >= uiWidth) return; // the added fix

    const uint* uiInputImage = &uiSource[globalPosX]; // bug here
    uint* uiOutputImage = &uiDest[globalPosX];
    // ...
```

Fig. 5. Bug #2 (invalid global read in col_kernel from `boxfilter`) root cause analysis.

Listing 8. Bug #23 memcheck report.

```
========= Program hit cudaErrorMemoryAllocation (error 2) due to "out of memory" on CUDA API call to cudaMalloc.
========= Saved host backtrace up to driver entry point at error
========= Host Frame: main [0x9208] in main
=========
========= Program hit cudaErrorInvalidValue (error 1) due to "invalid argument" on CUDA API call to cudaMemcpy.
========= Saved host backtrace up to driver entry point at error
========= Host Frame: main [0x8d9d] in main
=========
========= Invalid __global__ read of size 4 bytes
========= at compute_costs_kernel(const uchar4 *, short *, short *, short *, int, int, int)+0x110
========= by thread (1,4,0) in block (0,0,0)
========= Access at 0x3fc84 is out of bounds
========= and is 140660749108092 bytes before the nearest allocation at 0x7fee22000000 of size 1137491208 bytes
========= Saved host backtrace up to driver entry point at kernel launch time
========= Host Frame: main [0x92fa] in main
=============================================
```

Listing 9. Bug #23 location.

```
1  #define CUDA_CHECK(call) \
2  do { \
3      cudaError_t err_ = call; \
4      if (err_ != cudaSuccess) { \
5          fprintf(stderr, "CUDA error at %s:%d code=%d(%s) \"%s\" \n", \
6                  __FILE__, __LINE__, err_, cudaGetErrorString(err_), #call); \
7          exit(EXIT_FAILURE); \
8      } \
9  } while (0)
10 // ...
11  cudaMalloc((void**)&d_pixels, img_bytes);
12  cudaMemcpy(d_pixels, h_pixels, img_bytes
13  , cudaMemcpyHostToDevice);
14  CUDA_CHECK(cudaMalloc((void**)&d_pixels, img_bytes));
15  CUDA_CHECK(cudaMemcpy(d_pixels, h_pixels, img_bytes
16  , cudaMemcpyHostToDevice));
17 // ...
```

Fig. 6. Bug #23 (invalid global read in compute_costs_kernel from `seam-carving`) root cause analysis.

preceded by a memory allocation error (out-of-memory) in cudaMalloc. The problem is that the original code (lines 11-13 in Listing 9) never checks CUDA API return values at runtime. As a result, the program continues execution and launches device-side kernels even after an API failure. This bug manifests only with large input images that request allocations exceeding available GPU memory, causing `cudaMalloc` to fail silently. One potential fix is to encapsulate all CUDA API calls with checks (lines 1–9) to ensure proper application exit if any CUDA API call fails. This bug demonstrates the importance of considering host- and device-side code interaction when identifying and fixing bugs.

7.1.3 Bug Sample (Id #24). Bug #24 is a heap-buffer overflow in the SDKBitMap::load function of the `urng` benchmark. This benchmark loads an input ppm image and generates uniform random noise. Listing 10 shows the error reported by Google's AddressSanitizer for the out-of-bounds read. The root cause is that the original code (Listing 11) assumes the "size" and "offset" fields read from the input image header are trusted. Thus, it never compares these fields with the actual input image size. The fix is to add a validation check (lines 5–8) ensuring the application exits properly if the input image is not a valid bitmap file. This bug demonstrates cuFuzz's versatility in catching bugs using different sanitizers.

Listing 10. Bug #24 address sanitizer report.

```
==8121==ERROR: AddressSanitizer: heap−buffer−overflow on address 0
        x7fcfffdff800 at pc 0x55c89ad345cd bp 0x7ffe7e99f860 sp 0
        x7ffe7e99f858
READ of size 1 at 0x7fcfffdff800 thread T0
    #0 0x55c89ad345cc in SDKBitMap::load(char const*) src/urng−cuda
            /../include/SDKBitMap.h:368:39
    #1 0x55c89ad30e44 in main src/urng−cuda/main.cu:38:13

0x7fcfffdff800 is located 0 bytes to the right of 786432−byte region [0
        x7fcfffd3f800,0x7fcfffdff800)
allocated by thread T0 here:
    #0 0x55c89ad2e69d in operator new[](unsigned long) (src/urng−
            cuda/main+0xe369d) (BuildId: 19
            d629d31da22263bff3ab5f79c6f821c59d7085)
    #1 0x55c89ad3353d in SDKBitMap::load(char const*) src/urng−cuda
            /../include/SDKBitMap.h:320:28

SUMMARY: AddressSanitizer: heap−buffer−overflow src/urng−cuda/../
        include/SDKBitMap.h:368:39 in SDKBitMap::load(char const*)
```

Listing 11. Bug #24 location.

```
void load(const char * filename){
    //...
    // Allocate buffer to hold all pixels
    unsigned int sizeBuffer = size − offset;
     if (width * height * (bitsPerPixel  8) != sizeBuffer) {
      printf("This is not a valid bitmap file.n");
      fclose(fd); return;
     }
    unsigned char * tmpPixels = new unsigned char[sizeBuffer];
    if (tmpPixels == NULL)
    {
        delete colors_;
        colors_ = NULL;
        fclose(fd); return;
    }
    // Read pixels from file, including any padding
    val = fread(tmpPixels, sizeBuffer * sizeof(unsigned char), 1, fd);
```

Fig. 7. Bug #24 (heap-buffer-overflow in SDKBitMap::load from urng) root cause analysis.

> ***Answer to RQ1:*** *cuFuzz discovered 43 previously-unknown bugs across the tested programs, including 19 in commercial libraries, demonstrating its bug-finding capability.*

7.2 Coverage

Coverage is the second key fuzzer evaluation metric. It poses the following question: starting with few input seeds covering limited library portions, how deeply can the fuzzer explore? More precisely, how many new edges can a fuzzer discover in a given time period? We evaluate cuFuzz's coverage by comparing its edge and unique input coverage to baseline AFL++. Figure 8 and Figure 9 compare edge and unique input coverage of different cuFuzz configurations from Section 6.4 to vanilla AFL++ across all benchmarks. An input is considered unique if it reaches new edges or increases hit counts enough to cross logarithmic bucket boundaries: AFL++'s host-side buckets (1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+) or our device-side buckets (1, 2, 3+, 512+, 4096+, 16384+, 65536+) which use coarser granularity to accommodate GPU thread counts.

To ensure fair comparison, we first run all configurations for the same duration and collect interesting findings in corresponding queues. We then rerun all queue entries per configuration using a modified "afl-showmap" tool to collect edge coverage information from both host and device sides (even for configurations not using device-side information like AFL++). The hypothesis is that even AFL++ might accidentally hit device-side edges that need quantification, though they are not part of fuzzer feedback. In Figure 8 and Figure 9, we show edge and unique input coverage on the main y-axis (solid lines) and total executions on the secondary y-axis (dashed lines).

Our results reveal several key observations. First, for benchmarks with available source code, vanilla AFL++ achieves better coverage (or reaches the same coverage faster) than cuFuzz. This stems from two reasons: (a) AFL++ runs faster without sanitization and device-side coverage slowdowns, and (b) host-side coverage suffices for these benchmarks since they rely on simple kernels where host-side edges indicate device-side execution well. Second, for benchmarks invoking closed-source libraries, cuFuzz achieves better coverage than AFL++ due to NVBit benefits. Third, coverage of cuFuzz and cuFuzz-noSanitizer is nearly identical with only small delays from sanitizer slowdowns. The same observation applies to AFL++ and cuFuzz-noDeviceCoverage. Finally, while a single cuFuzz execution is slower than AFL++'s single execution, cuFuzz achieves the same coverage as AFL++ with fewer executions. This occurs because cuFuzz leverages device-side coverage to guide the fuzzing process, generating more effective mutations in fewer executions.

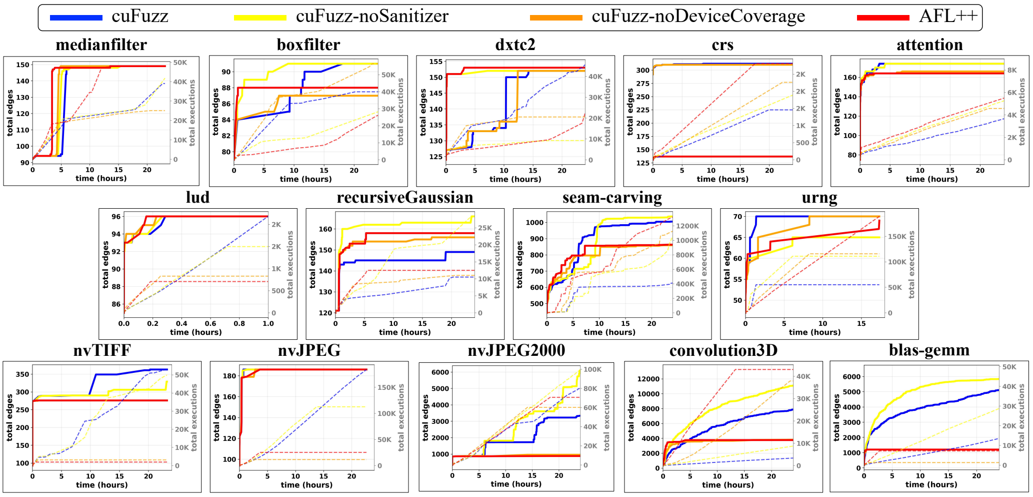


Fig. 8. Host- and device-side edge coverage over 24 hours for all 14 benchmarks. Solid lines show cumulative edges discovered; dashed lines show total executions.

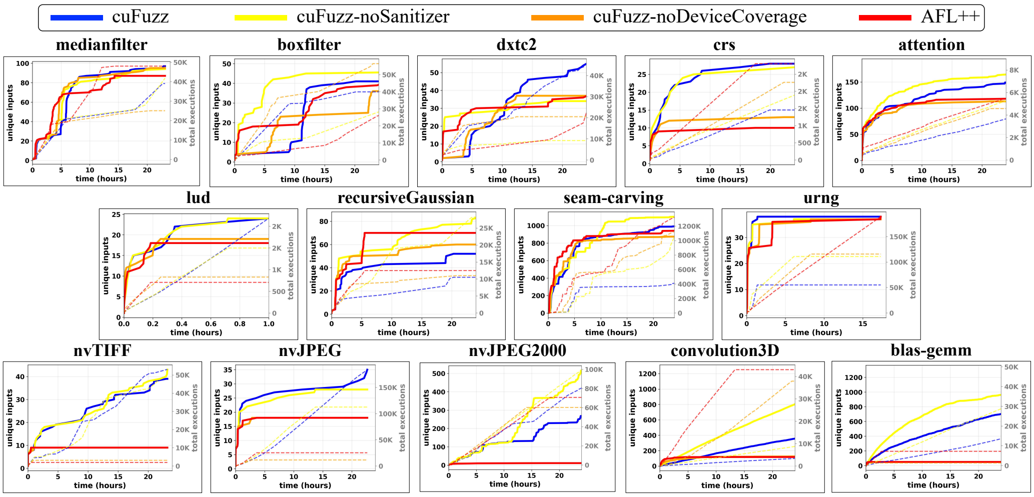


Fig. 9. Unique inputs discovered over 24 hours for all 14 benchmarks. An input is unique if it triggers new edges or crosses hit-count bucket boundaries.

To further confirm our observation regarding closed-source libraries, Figure 10 reports device-side edge progress during fuzzing for the five closed-source applications, comparing cuFuzz with cuFuzz-noDeviceCoverage. Each stacked bar shows the cumulative device-side edges discovered, decomposed by kernel. As shown, leveraging device-side coverage through NVBit enables cuFuzz to discover significantly more device-side edges compared to cuFuzz-noDeviceCoverage, with improvements ranging from 9% (`nvTIFF`) to 289% (`blas-gemm`). Open-source benchmarks are omitted since both configurations reach the same edges (per Figure 8). Closed-source libraries benefit more because their pre-compiled binaries lack host-side instrumentation.

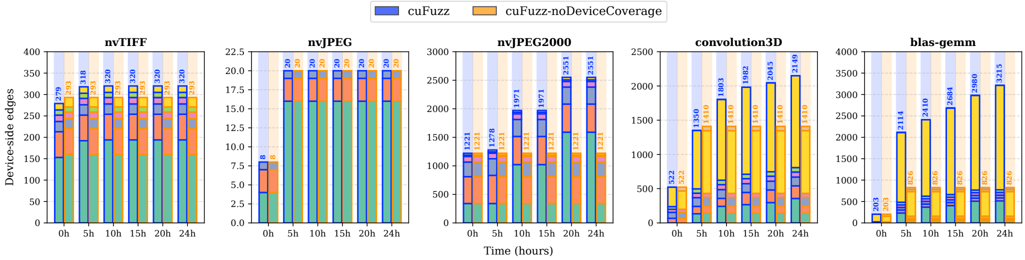


Fig. 10. Device-side edge coverage over time for closed-source libraries, comparing cuFuzz (left bars, blue labels) vs. cuFuzz-noDeviceCoverage (right bars, orange labels). Numbers above bars indicate total device-side edges at each time point.

> ***Answer to RQ2:*** *Enabling device-side coverage in cuFuzz (blue and yellow) yields substantially higher coverage than baseline AFL++ (red) for closed-source libraries, because their pre-compiled binaries lack host-side instrumentation and device-side coverage via NVBit is essential for guiding exploration. Open-source programs show similar coverage across configurations since host-side instrumentation suffices to guide device-side exploration.*

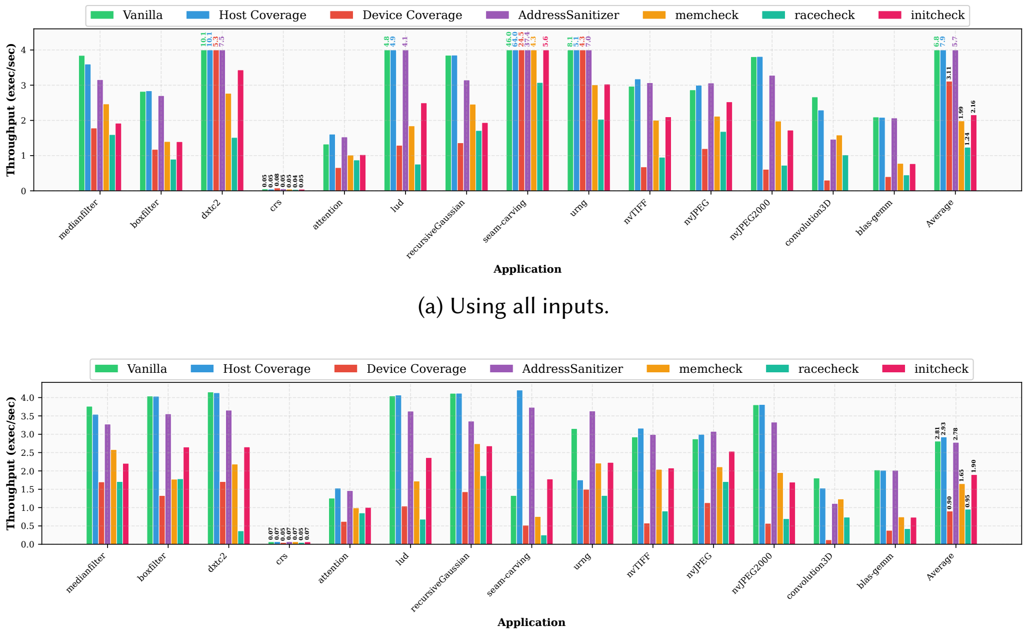
7.3 Performance

To understand the performance impact of different cuFuzz components, we perform an ablation study. We select 100 unique generated inputs per benchmark and run them using "hyperfine" [35] on individual cuFuzz components, reporting median throughput per second (1000/execution time). The goal is to identify speed-of-light throughput when each component is disjointly enabled over the same input set, since comparing throughput across different fuzzing runs introduces significant randomness. Figure 11 shows our ablation study results.

Considering all 100 inputs regardless of execution pattern (Figure 11a), we observe comparable average throughput for vanilla execution, host-side coverage, and AddressSanitizer. This is expected since these three variants rely on compile-time instrumentation with similarly low overhead. Conversely, we observe 55% throughput reduction for device-side coverage due to NVBit's runtime overheads. Similarly, we observe 71% throughput reduction for memcheck and 82% for racecheck due to Compute Sanitizer's runtime overheads.

While average results are expected, Figure 11a contains outliers where vanilla throughput significantly exceeds other workloads. For example, `seam-carving`'s vanilla throughput is 46 exec/sec (24 exec/sec with NVBit). To understand this behavior, we investigated per-input throughput and noticed a striking difference between inputs rejected early by host-side code and inputs triggering at least one kernel execution.

For device-triggering inputs, `seam-carving` vanilla throughput is only 1.2 exec/sec (0.5 exec/sec with NVBit), comparable to other workloads. We define device-triggering inputs as the inputs that actually manage to execute at least one device-side kernel. Other inputs were too shallow, failing host-side sanity checks and causing early program termination. This explains why vanilla and NVBit throughput were notably high when no device-side execution occurred. Conversely, Compute Sanitizer tools always incur the same overhead regardless of input, likely due to runtime loading and initialization costs of the sanitizer itself. We report only device-triggering input throughput in Figure 11b. On average, NVBit reduces throughput by 67% and memcheck by 39%, while host-side sanitization has negligible throughput impact.



(a) Using all inputs.

(b) Using device-triggering inputs only.

Fig. 11. Throughput (executions/second) of individual cuFuzz components measured using 100 unique inputs per benchmark. (a) All inputs; (b) Device-triggering inputs only.

> ***Answer to RQ3:*** *Device-side coverage collection is cuFuzz's primary throughput bottleneck, reducing throughput by 67% for inputs that trigger device execution. The throughput impact of device-side sanitization varies with the specific tool enabled.*

7.4 Persistent Mode

To understand persistent mode's impact on cuFuzz, Figure 12 compares cuFuzz's edge coverage in persistent mode (black line) versus regular single-input-per-process cuFuzz from Figure 8 (blue line). We observe that cuFuzz-persistent achieves higher coverage than cuFuzz in the same time for seven applications, while both approaches achieve the same coverage for four applications. For the remaining three programs (`seam-carving`, `urng`, and `nvTIFF`), cuFuzz-persistent does not improve fuzzing throughput because the harness requires expensive reinitialization inside the loop. For `nvTIFF` and `seam-carving`, we must call `cudaMalloc` and `cudaFree` for large buffers in every iteration to handle variable-sized inputs and properly reset state between runs. Moving these allocations outside the loop causes crashes when subsequent inputs require different buffer sizes. For `urng`, the kernel execution itself is so short-lived (sub-millisecond) that the overhead of the persistent loop bookkeeping negates any benefit from avoiding process restart.

> ***Answer to RQ4:*** *In 11 of 14 programs, cuFuzz-persistent delivered equal or better coverage than single-input-per-process cuFuzz, highlighting its potential.*

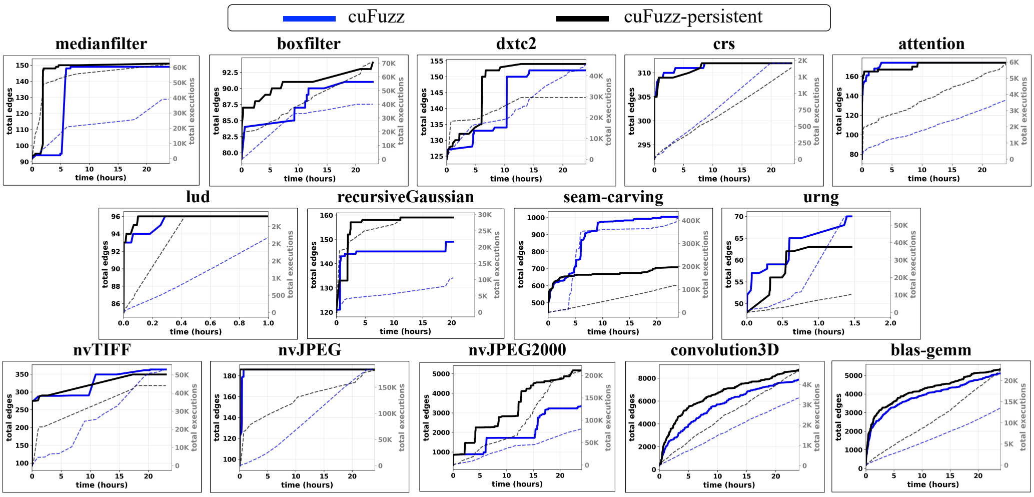


Fig. 12. Edge coverage comparison: cuFuzz-persistent (black) vs. regular cuFuzz (blue) over 24 hours.

7.5 Sanitization

Given Compute Sanitizer tools' large impact on fuzzing throughput (Figure 11), running all sanitizers on every input would be suboptimal. Inspired by prior work [20], we evaluate four strategies for subsetting inputs fed to sanitizers: (1) "all-trace": runs all sanitizers on every input, (2) "unique-trace": runs all sanitizers on inputs with unique execution paths, (3) "simple-trace": runs sanitizers on inputs with unique execution paths without considering thread/edge count, and (4) "coverage-increase": runs sanitizers on inputs causing fuzzing engine coverage increases. Figure 13 summarizes cuFuzz's edge coverage with each strategy.

As the most sensitive strategy, all-trace has the lowest performance. While it achieves the same coverage as other strategies for some benchmarks (e.g., `medianfilter`), it is generally slower. Conversely, coverage-increase is the most performant strategy as the least sensitive one, running only inputs that are part of the fuzzing queue through sanitizers. The unique-trace and simple-trace strategies balance sensitivity and runtime overheads; thus, the latter (simple-trace) is enabled by default in our main experiments (Section 6.4). We justify this decision in the following section.

> ***Answer to RQ5:*** *While all strategies progress similarly over time, the less sensitive input selection methods "simple-trace" and "coverage-increase" run faster than the others because they incur lower device-sanitization overhead.*

7.6 Time to Exposure

To understand device-side sanitization's impact on cuFuzz's bug-finding capability and to quantify the effectiveness of the different sanitization strategies, we collected bug exposure times for different cuFuzz configurations. Table 3 shows the results with the shortest exposure time highlighted in blue. After three 24-hour fuzzing experiments, we collect unique bugs found by each configuration. We applied Google's AddressSanitizer and Compute Sanitizer's memcheck, racecheck, and initcheck tools to all crash folder contents, collecting crashes from generated test inputs per configuration. We identified unique crash bugs based on collected error messages, first removing crashes with identical stack traces, then manually identifying crashes with different stack traces but suspected shared root causes. We identify the following observations.

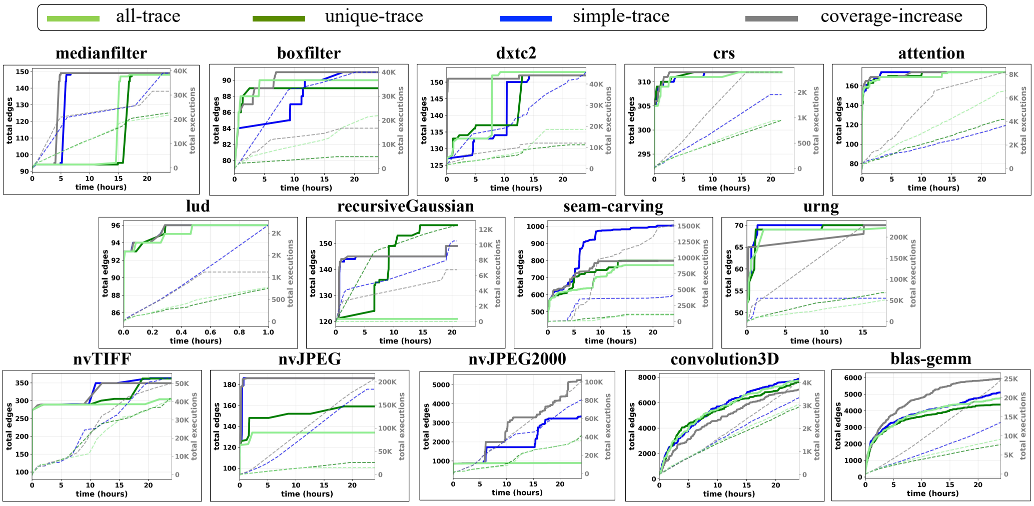


Fig. 13. cuFuzz edge coverage in various sanitization strategies. "simple-trace" is the default cuFuzz strategy.

First, without device-side sanitization (AFL++ and cuFuzz-noSanitizer columns in Table 3), the fewest bugs are found (11 and 9 out of 43, respectively). Only host-side (H) bugs causing application-level crashes are captured. Enabling device-side sanitization (even without device-side coverage, such as cuFuzz-noDeviceCoverage) significantly increases bugs found (30 out of 43).

Second, comparing sanitization strategies, simple-trace strategy (cuFuzz column) finds the most bugs (36 out of 43), followed by coverage-increase strategy (35 out of 43). Despite running every input through sanitizers, all-trace strategy finds only 28 bugs, mainly because the 24-hour limit expired before exploring sufficient paths. Thus, we use simple-trace strategy by default in our main experiments (Section 7.2 and Section 7.4). Finally, enabling persistent mode identifies 35 bugs, including three bugs captured only in cuFuzz-persistent runs. It also has the largest number of bugs with the least time to exposure (16 out of 43).

Answer to RQ6: *The "simple-trace" cuFuzz configuration found the most bugs (83%), while its persistent-enabled variant achieved the shortest time to exposure for the largest bugs share (37%).*

7.7 Comparison with Kernel-Level Fuzzing

Prior work on GPU fuzzing [22, 34] proposed kernel-level fuzzing, which isolates individual GPU kernels and permutes their input parameters independently. To understand the trade-offs between kernel-level and whole-program fuzzing, we implemented kernel-level harnesses for 22 kernels across nine open-source benchmarks (closed-source libraries are excluded since kernel names and arguments are hidden). Each harness allocates buffers, initializes the CUDA runtime, and invokes a single target kernel with independently permuted arguments (grid/block dimensions, buffer sizes, and scalar parameters) read from fuzzer-generated input files. For fair comparison, we ran these harnesses using the same cuFuzz capabilities (AFL++ mutation, host-side coverage, NVBit device-side coverage in simple-trace sanitization mode) for 24 hours, repeated three times.

Table 4 summarizes the results for the 24 open-source bugs from Table 2. Of these, kernel-level fuzzing found only six bugs (✓), missed eight device-side bugs (✗), and could not detect the ten host-side bugs by design (N/A). Focusing on the 14 device-side bugs where kernel-level fuzzing is applicable, it achieves only 43% recall (6/14) compared to cuFuzz's whole-program approach which found all 14.

Table 3. Time to exposure of bugs for different cuFuzz configurations.

Bug id	Library	AFL++	cuFuzz-noSanitizer	cuFuzz-noDevCov	cuFuzz	cuFuzz-persistent	cuFuzz-all-trace	cuFuzz-unique-trace	cuFuzz-cov-increase
1	attention (H)	20:57:35	-	21:19:16	-	-	-	-	04:39:38
2	boxfilter (D)	-	-	00:26:03	00:34:30	00:00:22	00:29:20	00:30:31	00:36:38
3	boxfilter (D)	-	-	03:27:40	04:35:14	00:15:15	00:38:39	00:39:15	00:38:21
4	boxfilter (D)	-	-	05:27:18	06:00:03	01:09:55	00:42:34	00:44:07	06:33:34
5	boxfilter (H)	00:19:36	00:33:02	04:24:21	06:00:03	00:15:45	00:37:47	00:38:26	00:37:37
6	crs (H)	00:07:04	00:05:35	00:07:47	00:17:19	00:41:32	00:15:20	00:17:01	00:37:54
7	dxtc2 (D)	-	-	08:12:47	10:45:04	00:49:27	07:50:27	04:44:39	00:04:23
8	dxtc2 (D)	-	-	10:17:03	09:39:31	00:51:51	07:48:09	05:01:39	00:04:02
9	dxtc2 (D)	-	-	00:00:33	00:00:33	00:16:24	00:00:34	00:00:33	00:04:02
10	dxtc2 (H)	-	-	18:52:06	12:58:30	02:12:57	07:48:09	20:34:14	-
11	lud (D)	-	-	00:00:04	00:00:11	00:00:05	00:00:17	00:00:12	00:00:09
12	lud (H)	00:00:38	00:00:41	00:00:12	00:00:48	00:00:17	00:01:09	00:01:01	00:00:51
13	medianfilter (D)	-	-	04:54:32	06:43:55	-	22:31:06	15:31:42	06:03:30
14	medianfilter (D)	-	-	04:22:53	05:26:24	01:10:59	15:07:54	15:13:37	04:31:17
15	medianfilter (D)	-	-	04:54:32	05:42:48	-	15:29:11	15:19:32	04:36:40
16	medianfilter (H)	02:01:07	04:26:50	04:54:32	05:42:48	03:59:49	17:28:04	15:31:42	04:36:40
17	medianfilter (H)	02:31:45	04:40:24	04:37:16	05:53:37	01:11:39	15:14:51	15:19:32	04:44:09
18	recursiveGaussian (H)	-	-	06:49:16	04:03:00	-	-	06:48:36	14:43:22
19	recursiveGaussian (D)	-	-	-	02:29:48	-	-	06:27:51	02:00:43
20	recursiveGaussian (D)	-	-	00:29:34	00:31:04	00:27:49	00:30:28	00:26:45	00:30:53
21	recursiveGaussian (D)	-	-	01:04:51	01:37:08	-	13:43:10	07:30:35	01:04:13
22	recursiveGaussian (H)	01:11:02	01:07:51	01:04:51	01:37:08	00:28:13	05:26:07	01:04:13	01:04:26
23	seam-carving (D)	-	-	04:11:35	13:52:39	-	-	-	-
24	urng (H)	00:02:17	00:03:02	00:03:21	00:03:23	00:19:16	00:22:36	00:32:40	00:02:48
25	nvTIFF (D)	-	-	00:34:13	06:49:08	01:01:19	-	-	-
26	nvTIFF (D)	-	-	01:15:46	-	-	10:05:52	11:02:27	-
27	nvTIFF (H)	00:15:34	06:57:32	00:33:26	06:48:51	01:01:00	-	-	06:39:07
28	nvTIFF (D)	-	-	-	00:59:35	01:15:58	01:15:13	01:17:39	00:53:19
29	nvTIFF (D)	-	-	-	-	11:04:36	-	-	-
30	nvJPEG (H)	00:08:53	00:16:29	00:10:37	02:10:21	00:17:52	01:02:40	01:55:11	01:33:27
31	nvJPEG (D)	-	-	-	00:44:31	00:00:02	00:04:46	00:05:36	00:38:50
32	nvJPEG2000 (D)	-	-	-	06:04:46	02:06:56	13:06:34	-	05:50:11
33	nvJPEG2000 (D)	-	-	-	05:41:34	12:27:11	-	-	05:36:59
34	nvJPEG2000 (D)	-	-	-	-	12:15:39	-	-	-
35	nvJPEG2000 (D)	-	-	-	-	07:25:03	-	-	09:02:03
36	nvJPEG2000 (D)	-	-	-	-	07:07:57	-	-	-
37	nvJPEG2000 (D)	-	-	-	05:49:55	02:10:20	-	-	11:54:45
38	nvJPEG2000 (D)	-	-	-	13:15:16	12:29:05	-	-	06:03:28
39	nvJPEG2000 (D)	-	-	-	-	12:27:11	-	-	05:57:24
40	nvJPEG2000 (H)	00:31:08	-	00:38:11	06:29:29	03:36:02	-	-	-
41	cuDNN (D)	-	-	-	08:56:58	00:25:47	03:02:12	02:36:43	03:29:11
42	cuDNN (D)	-	-	00:31:27	00:18:48	00:17:52	00:39:01	00:24:18	00:22:12
43	cuDNN (D)	-	-	00:04:31	00:02:06	00:01:13	00:01:52	00:01:10	00:07:40
	Total findings	11	9	30	36	35	28	29	35

More critically, kernel-level fuzzing generated 16 *false positives*, i.e., spurious errors that cannot occur in the original program (Table 5). These false alarms arise because independent parameter permutation violates host-enforced invariants. For example, in `seam-carving`'s `compute_M_step1` kernel, harnesses fuzz `gridSize` independently when the original program derives it from data dimensions (i.e., `gridSize = ⌈current_w/128⌉`). Similarly, in `lud`'s `lud_perimeter` kernel, compile-time constants like `BLOCK_SIZE = 16` become fuzzed runtime variables, causing shared memory sizing mismatches. These violations trigger memory errors and race conditions that are impossible under valid host-side orchestration.

The eight false negatives (✗) stem from fundamental limitations of kernel isolation. Harnesses abstract away host-side logic that is often the *root cause* of device-side bugs: (1) **file parsing failures**, where malformed images cause garbage dimensions, but harnesses always initialize buffers correctly; (2) **error handling flaws**, where programs continue after failed `cudaMalloc` calls, but harnesses explicitly check return codes; (3) **host-device interface mismatches**, where

Table 4. Kernel-level fuzzing results. ✓ = found, ✗ = missed (device-side), N/A = not applicable (host-side).

Bug#	Target	Bug type	Kernel/File	Found?	Reason
1	attention	Heap buffer overflow	reference.h:13	N/A	Host-side bug
2	boxfilter	Invalid global read	col_kernel	✓	Missing kernel-internal bounds check
3	boxfilter	Uninitialized global read	row_kernel	✗	Harness always initializes buffers
4	boxfilter	Uninitialized global read	col_kernel	✓	Found by malformed input parameters
5	boxfilter	Heap buffer overflow	reference.cpp:121	N/A	Host-side bug
6	crs	Floating-point exception	main.cu:81	N/A	Host-side bug
7	dxtc2	Invalid global read	compress	✗	Host-device interface mismatch
8	dxtc2	Invalid global write	compress	✗	Host-device interface mismatch
9	dxtc2	Data race (shared memory)	compress	✓	Structural synchronization bug
10	dxtc2	Heap buffer overflow	shrUtils.cu:1122	N/A	Host-side bug
11	lud	Invalid global read	lud_diagonal	✓	Found with a different trigger
12	lud	Stack buffer overflow	common.cu:161	N/A	Host-side bug
13	medianfilter	Invalid global write	ckMedian	✓	Missing kernel-internal bounds check
14	medianfilter	Uninitialized global read	ckMedian	✗	Harness correctly provisions buffers
15	medianfilter	Data race (shared memory)	ckMedian	✓	Harness violates pitch invariant
16	medianfilter	Heap buffer overflow	MedianFilterHost.cu:70	N/A	Host-side bug
17	medianfilter	Heap buffer overflow	MedianFilterHost.cu:85	N/A	Host-side bug
18	recursiveGaussian	Heap buffer overflow	shrUtils.cu:1302	N/A	Host-side bug
19	recursiveGaussian	Invalid global read	RecursiveRGBA	✗	Malformed PPM leading to dimension mismatch
20	recursiveGaussian	Uninitialized global read	RecursiveRGBA	✗	Malformed PPM leading to dimension mismatch
21	recursiveGaussian	Uninitialized global read	Transpose	✗	Malformed PPM leading to dimension mismatch
22	recursiveGaussian	Heap buffer overflow	RecursiveGaussianHost.cu:180	N/A	Host-side bug
23	seam-carving	Invalid global read	compute_costs_kernel	✗	Host-side error handling flaw
24	urng	Heap buffer overflow	SDKBitMap.h:368	N/A	Host-side bug

Table 5. False positives from kernel-level fuzzing. All stem from violating host-enforced parameter invariants.

#	Target	Kernel	Error type	Violated invariant
FP1	attention	kernel1_blockReduce	Invalid global read	`gridSize = n`
FP2	attention	kernel2_blockReduce	Invalid global read	`gridSize = d`
FP3	attention	kernel2_blockReduce	Invalid global write	`gridSize = d`
FP4	crs	gcrs_m_1_w_4_coding_dotprod	Invalid global read	index derived from buffer bounds
FP5	crs	gcrs_m_2_w_4_coding_dotprod	Invalid global read	index derived from buffer bounds
FP6	boxfilter	row_kernel	Invalid shared read	`iRadiusAligned ≥ iRadius`
FP7	boxfilter	row_kernel	Invalid global read	`gridY = uiHeight`
FP8	urng	noise_uniform	Invalid global read	`gridSize = imageSize / blockSize`
FP9	urng	noise_uniform	Uninitialized global read	`gridSize = imageSize / blockSize`
FP10	lud	lud_perimeter	Race condition	`BLOCK_SIZE = 16` (compile-time)
FP11	lud	lud_perimeter	Invalid global read	`gridSize` derived from `matrix_dim`
FP12	lud	lud_internal	Race condition	`BLOCK_SIZE = 16` (compile-time)
FP13	lud	lud_internal	Invalid global read	`gridSize` derived from `matrix_dim`
FP14	seam-carving	compute_M_step1	Invalid global read	`base_row < h`
FP15	seam-carving	min_reduce	Invalid global write	`gridSize` derived from output size
FP16	seam-carving	compute_M_step1	Race condition	`gridSize = ⌈current_w/128⌉`

buggy host-side buffer sizing calculations cannot be expressed when harnesses derive buffer sizes from kernel parameters. Bug #23 (Figure 6) exemplifies this: the kernel error is triggered by a host-side allocation failure that kernel-level fuzzing could not reproduce.

We additionally ran kernel-level harnesses with the all-trace sanitization strategy (running all sanitizers on every input) but observed no new findings beyond those reported above, confirming that the false negative gap stems from harness design limitations rather than sanitization coverage.

Answer to RQ7: *Kernel-level fuzzing found only 6 of 14 device-side bugs (43% recall) while generating 16 false positives due to violated parameter invariants. In contrast, cuFuzz's whole-program approach found all 14 device-side bugs plus 10 additional host-side bugs with zero false positives, demonstrating the importance of preserving host-device context.*

8 Discussion

8.1 Differences Between CPU and GPU Fuzzing

Our experience with cuFuzz reveals multiple key differences between CPU and GPU fuzzing. Most importantly, throughput differs significantly between CPU and GPU workloads. While CPU fuzzing achieves tens to hundreds of exec/sec on a single thread, GPU fuzzing is typically limited to tens of exec/sec due to CUDA runtime initialization overhead. Second, GPU fuzzing is more susceptible to context switching due to limited GPU nodes per system (8 GPUs versus 128 cores per server). Both factors limit the number and duration of fuzzing campaigns.

Potential solutions for improving GPU fuzzing throughput include using techniques to co-execute multiple GPU processes on the same device (e.g., NVIDIA Multi-Process Service, MPS [28]) or radically modifying AFL++ and the application under test to process inputs in batches rather than individually. Both solutions depend on the GPU utilization of the program-under-test.

8.2 GPU Utilization During Fuzzing

Our experiments reveal that GPU utilization during fuzzing depends heavily on both the program-under-test and input characteristics. Some programs invoke multiple kernels with high thread counts, fully utilizing GPU resources, while others use only a fraction of available SMs and shared memory. Similarly, some inputs trigger deep functionality while others fail early, causing utilization to vary widely for the same program. Future work could leverage these utilization patterns to co-execute low-utilization workloads and improve throughput.

8.3 Thread-Interleaving Coverage Limitations

Our current coverage mechanism collapses thread variations—two executions with identical control flow but different thread interleavings produce the same coverage signature. Consequently, the races cuFuzz finds are "surface-level," exposed simply by reaching the code location where `racecheck` detects the conflict. We verified that all 5 `racecheck`-detected bugs in our evaluation were reproducible across multiple runs, suggesting they are relatively deterministic races triggered by the code structure rather than timing. However, cuFuzz may miss race conditions dependent on precise thread-scheduling timing. Developing thread-interleaving-aware coverage is a promising future direction, though naive implementation would lead to state space explosion on GPUs where thousands of threads execute concurrently. We leave this extension to future work.

9 Related Work

Fuzzing has long been a cornerstone of software reliability and security testing. Tools like AFL++ [6] and LibFuzzer [26] demonstrate the effectiveness of coverage-guided input mutation for CPU programs. However, GPU fuzzing remains relatively underexplored due to architectural and tooling challenges unique to heterogeneous systems. Next, we review related work in GPU fuzzing, dynamic sanitizers, and other fuzzing directions.

Table 6. Comparison of GPU fuzzing approaches.

Tool	Target	Execution mode	Instrumentation	Feedback	Sanitizers	Closed-source support
CLFuzz [34]	OpenCL kernels	Physical GPU	Source/IR instrumentation	Branch (kernel)	✗	✗
CUDA-emulation [40]	CUDA kernels	CPU emulation	Source/IR translation	Edge (CPU Mapped)	✓ (CPU ASan)	✗
Fuzz4Cuda [48]	CUDA libraries	Physical GPU	Debugger (CUDA-GDB)	Basic block	✗ (Crash only)	✓
CUDAsmith [14]	CUDA compilers	CPU	Random code generation	None (Black-box)	✗	N/A
Moneta [17]	GPU drivers	Simulated	Driver state recording	State/API	✗	✓
DL Fuzzers [10, 46, 47]	DL frameworks	API level	Differential testing	API/model coverage	✗	✓
cuFuzz (Ours)	CUDA programs and libraries	Physical GPU	Compile instrumentation (host) & runtime instrumentation (device)	Edge (host+device)	✓ (CPU ASan and Compute Sanitizer)	✓

9.1 GPU Fuzzing

Table 6 summarizes the key differences between cuFuzz and related GPU fuzzing approaches. Early GPU fuzzing efforts focused on kernel-level input permutation. CVFuzz [22] targets OpenCL kernels by generating pathological inputs to expose complexity vulnerabilities. Similarly, CLFuzz [34] leverages SMT solvers to generate inputs for uncovered branches in OpenCL kernels, aiming to increase kernel-level code coverage. These approaches suffer from false positives due to the lack of host-device context awareness.

Concurrently, Singh et al. propose hardening CUDA programs by transforming them into CPU-executable code [40]. That approach maps GPU thread hierarchies to CPU loops to leverage mature CPU fuzzing ecosystems (e.g., AFL++, ASan). However, relying on emulation introduces significant performance overheads, with reported throughputs often falling below 1 execution per second for complex kernels, and inherently fails to capture hardware-specific behaviors. In contrast, our work performs fuzzing on physical GPU hardware. This not only achieves significantly higher throughput by running kernels natively but also ensures fidelity to the actual GPU memory consistency model and warp scheduling. Moreover, by leveraging binary instrumentation, our approach supports closed-source libraries and driver interactions that are inaccessible to transformation-based methods requiring source code or intermediate representation [40].

Another concurrent work, Fuzz4Cuda [48], proposes a framework that leverages the CUDA-GDB interface to fuzz GPU libraries. While it addresses initialization latency via a persistent loop, it relies on software breakpoints to track basic block coverage, which is inherently less sensitive to logic errors than the transition-aware edge coverage utilized by our approach. This coarse granularity, combined with the inability to employ runtime sanitizers due to debugger conflicts, significantly limits its effectiveness. For instance, in a month-long campaign, Fuzz4Cuda identified only 5 bugs (primarily in nvJPEG), missing silent errors like data races and uninitialized reads. In contrast, our lightweight binary instrumentation and decoupled sanitization enabled the discovery of 43 unique vulnerabilities across 14 diverse programs and libraries in under 24 hours.

CUDAsmith [14] introduces a fuzzing framework for CUDA compilers. While effective for compiler validation and uncovering compiler bugs, CUDAsmith does not address runtime errors in CUDA applications or library APIs. Another orthogonal line of work focuses on GPU driver fuzzing. Moneta [17] represents the most relevant work, statefully fuzzing GPU drivers by recalling past in-vivo driver execution states at scale. In contrast, cuFuzz focuses on fuzzing user-land applications.

Finally, while cuFuzz focuses on fuzzing programs with file-based inputs, other tools target deep learning (DL) libraries/frameworks, such as TensorFlow [1] and PyTorch [33]. DL library fuzzers [4, 5, 10, 46, 47] primarily depend on differential testing to expose bugs in the DL kernel and API implementations running on CPUs and/or GPUs. To the best of our knowledge, none of these tools deploy heterogeneous code coverage and sanitizers.

9.2 GPU Dynamic Sanitizers

Besides Compute Sanitizer [29], there exist other tools for catching GPU code errors at runtime. Examples include iGUARD [18], a dynamic binary instrumentation tool based on NVBit for catching global memory data races not currently covered by Compute Sanitizer's racecheck. BinFPE [21] and GPU-FPX [23] identify floating-point exceptions in device code, both based on NVBit.

Compiler-based tools like cuCatch [43] and HiRace [13] use compiler analysis and instrumentation to capture device-side memory safety errors and data races, respectively. All these tools can be easily integrated into cuFuzz's fuzzing loop without runtime compatibility concerns.

9.3 Other Fuzzing Directions

There has been a large body of work in improving file-based fuzzers. The baseline fuzzer used in this paper, AFL++ [6], aggregates many research proposals built on top of the original AFL [9]. However, several recent proposals are not yet integrated into mainstream AFL++. For example, data-flow-aware coverage has shown promise for improving coverage-guided fuzzing [7, 12, 19, 24]. Such techniques could be incorporated into cuFuzz to enhance host- and device-side coverage beyond the default control-flow edge coverage used in this paper.

10 Threats to Validity

10.1 Threats to External Validity

One external validity threat concerns the benchmark programs used in our evaluation. To mitigate this threat, we used a diverse set of programs from multiple domains that accept different input formats. Another threat involves seed selection's impact on outcomes. We addressed this by selecting inputs from each benchmark's testing suite and using identical initial seeds across all configurations. Using more seeds might yield different results (i.e., better coverage or more bugs) but would ultimately require more than 24 hours to converge given CUDA fuzzing's low throughput.

10.2 Threats to Internal Validity

The main internal validity threat is randomness in the fuzzing process, which could introduce experimental result variability. We addressed this by repeating all experiments three times, using identical initial seeds across all configurations, and employing the "-Z" flag to force sequential exploration of the fuzzing input queue.

11 Conclusion

The rapid adoption of GPU computing in domains ranging from high-performance computing to safety-critical systems has amplified the importance of robust GPU testing. However, the unique characteristics of GPU architectures, such as massive parallelism, heterogeneous memory hierarchies, and complex host-device interactions, introduce distinct challenges for identifying memory- and thread-safety vulnerabilities. Fuzzing is a proven automated testing method for identifying these vulnerabilities on CPUs, yet it remains underutilized for GPUs.

cuFuzz demonstrates that practical and effective fuzzing of CUDA-based GPU programs is feasible. By rethinking fuzzing granularity, integrating device-side coverage, and decoupling sanitization from coverage collection, cuFuzz overcomes multiple barriers in GPU testing. Its ability to uncover 43 previously unknown bugs—including 19 in widely deployed production libraries—highlights the inadequacy of existing testing pipelines. As GPU workloads continue to expand into privacy-sensitive applications, robust testing tools like cuFuzz will be essential.

Data-Availability Statement

The cuFuzz artifact is available on Zenodo [44]. The artifact includes the source code, usage instructions, and evaluation scripts to reproduce the main experiments in this paper.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback that helped improve this paper. We are grateful to Zheming Jin for promptly addressing the HeCBench bug reports uncovered by cuFuzz, and to the maintainers of NVIDIA's CUDA-accelerated libraries for handling the reported bugs. We also thank Aamer Jaleel, Mark Stephenson, Sana Damani, and the members of the Architecture Research Group at NVIDIA Research for helpful technical discussions.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI'16: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. Savannah, GA, USA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *OOPSLA'12: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. Tucson, AZ, USA, 113–132. doi:10.1145/2384616.2384625

[3] Black Duck. 2025. Coverity Documentation 2025.6.2. https://documentation.blackduck.com/bundle/coverity-docs/page/webhelp-files/help_center_start.html.

[4] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P. Kemerlis. 2023. IvySyn: automated vulnerability discovery in deep learning frameworks. In *USENIX Security'23: Proceedings of the 32nd USENIX Conference on Security Symposium*. USENIX Association, Anaheim, CA, USA, Article 134, 18 pages. https://www.usenix.org/conference/usenixsecurity23/presentation/christou

[5] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *ICSE'24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon, Portugal, Article 70, 13 pages. doi:10.1145/3597503.3623343

[6] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: combining incremental steps of fuzzing research. In *WOOT'20: Proceedings of the 14th USENIX Conference on Offensive Technologies*. USENIX Association, USA, Article 10, 1 pages. https://www.usenix.org/conference/woot20/presentation/fioraldi

[7] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security'20: Proceedings of the 29th USENIX Security Symposium*. San Antonio, TX, USA, 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[8] Google. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://google.github.io/oss-fuzz/.

[9] Google. 2019. american fuzzy lop: a security-oriented fuzzer. https://github.com/google/AFL

[10] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2021. Audee: automated testing for deep learning frameworks. In *ASE'21: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Virtual Event, Australia, 486–498. doi:10.1145/3324884.3416571

[11] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *USENIX Security'24: Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, Philadelphia, PA, USA, 4033–4050. https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

[12] Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2023. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Transactions on Software Engineering and Methodology* 32, 5, Article 132 (July 2023), 31 pages. doi:10.1145/3587156

[13] John Jacobson, Martin Burtscher, and Ganesh Gopalakrishnan. 2024. HiRace: Accurate and Fast Data Race Checking for GPU Programs. In *SC'24: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, Atlanta, GA, USA, Article 36, 14 pages. doi:10.1109/SC41406.2024.00042

[14] Bo Jiang, Xiaoyan Wang, W. K. Chan, T. H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. 2020. CUDAsmith: A Fuzzer for CUDA Compilers. In *COMPSAC'20: 2020 IEEE 44th Annual Computers, Software, and Applications Conference*. Madrid, Spain, 861–871. doi:10.1109/COMPSAC48688.2020.0-156

[15] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *ISPASS 2023: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. Raleigh, NC, USA, 325–327. doi:10.1109/ISPASS57527.2023.00041

[16] Zheming Jin and Jeffrey S. Vetter. 2023. HeCBench: heterogeneous computing benchmark suite. https://github.com/zjin-lcf/HeCBench.

[17] Joonkyo Jung, Jisoo Jang, Yongwan Jo, Jonas Vinck, Alexios Voulimeneas, Stijn Volckaert, and Dokyung Song. 2025. Moneta: Ex-Vivo GPU Driver Fuzzing by Recalling In-Vivo Execution States. In *NDSS 2025: Proceedings of the 2025 Network and Distributed System Security Symposium*. San Diego, CA, USA. doi:10.14722/ndss.2025.230218

[18] Aditya K. Kamath and Arkaprava Basu. 2021. IGUARD: In-GPU advanced race detection. In *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Virtual Event, Germany, 49–65. doi:10.1145/3477132.3483545

[19] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In *USENIX Security'23: Proceedings of the 32nd USENIX Security Symposium*. Anaheim, CA, USA, 4931–4948. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[20] Ziqiao Kong, Shaohua Li, Heqing Huang, and Zhendong Su. 2025. SAND: Decoupling Sanitization from Fuzzing for Low Overhead. In *ICSE'25: Proceedings of the 47th International Conference on Software Engineering*. Ottawa, Ontario, Canada, 255–267. doi:10.1145/2830772.2830818

[21] Ignacio Laguna, Xinyi Li, and Ganesh Gopalakrishnan. 2022. BinFPE: accurate floating-point exception detection for GPU applications. In *SOAP'22: Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. San Diego, CA, USA, 1–8. https://doi.org/10.1145/3520313.3534655

[22] Wentao Li, Zhiwen Chen, Xin He, Guoyun Duan, Jianhua Sun, and Hao Chen. 2022. CVFuzz: Detecting complexity vulnerabilities in OpenCL kernels via automated pathological input generation. *Future Generation Computer Systems* 127 (2022), 384–395. doi:10.1016/j.future.2021.09.006

[23] Xinyi Li, Ignacio Laguna, Bo Fang, Katarzyna Swirydowicz, Ang Li, and Ganesh Gopalakrishnan. 2023. Design and Evaluation of GPU-FPX: A Low-Overhead tool for Floating-Point Exception Detection in NVIDIA GPUs. In *HPDC'23: Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. Orlando, FL, USA, 59–71. https://doi.org/10.1145/3588195.3592991

[24] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *SP'22: Proceedings of the 2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 1–17. https://doi.org/10.1109/SP46214.2022.9833594

[25] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. 2017. A survey on deep learning in medical image analysis. *Medical Image Analysis* 42 (2017), 60–88. doi:10.1016/j.media.2017.07.005

[26] LLVM. 2016. LibFuzzer: a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[27] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. doi:10.1145/96267.96279

[28] NVIDIA. 2020. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.

[29] NVIDIA. 2025. Compute Sanitizer v2025.2.0. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html.

[30] NVIDIA. 2025. CUDA Library Samples. https://github.com/NVIDIA/CUDALibrarySamples/tree/master.

[31] NVIDIA. 2025. NVIDIA CUDA-X Libraries. https://developer.nvidia.com/gpu-accelerated-libraries.

[32] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security* 102 (2021), 102–115. doi:10.1016/j.cose.2020.102115

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Red Hook, NY, USA. https://dl.acm.org/doi/10.5555/3454287.3455008

[34] Chao Peng and Ajitha Rajan. 2020. Automated test generation for OpenCL kernels using fuzzing and constraint solving. In *GPGPU '20: Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. San Diego, CA, USA, 61–70. doi:10.1145/3366428.3380768

[35] David Peter. 2023. *hyperfine*. https://github.com/sharkdp/hyperfine

[36] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *EuroSec '25: Proceedings of the 18th European Workshop on Systems Security*. Rotterdam, Netherlands, 40–48. doi:10.1145/3722041.3723099

[37] Sihyun Roh, Woohyuk Choi, Jaeyoung Chung, Yoochan Lee, Suhwan Song, and Byoungyoung Lee. 2026. GHost in the SHELL: A GPU-to-Host Memory Attack and Its Mitigation. In *SP'26: Proceedings of the 2026 IEEE Symposium on Security and Privacy*. IEEE Computer Society, San Francisco, CA, USA, 883–898. doi:10.1109/SP63933.2026.00047

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *ATC '12: Proceedings of the 2012 USENIX Annual Technical Conference*. Boston, MA, USA, 10 pages.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019). https://doi.org/10.48550/arXiv.1909.08053

[40] Saurabh Singh, Ruobing Han, Jaewon Lee, Seonjin Na, Yonghae Kim, Taesoo Kim, and Hyesoon Kim. 2026. CuFuzz: Hardening CUDA Programs through Transformation and Fuzzing. *arXiv preprint arXiv:2601.01048* (2026). https://arxiv.org/abs/2601.01048

[41] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, Yuria Hiraga, Masahiro Nakadai, Jiabao Chen, Ken M. Nakanishi, Kosuke Mitarai, Ryosuke Imai, Shiro Tamiya, Takahiro Yamamoto, Tennin Yan, Toru Kawakubo, Yuya O. Nakagawa, Yohei Ibe, Youyuan Zhang, Hirotsugu Yamashita, Hikaru Yoshimura, Akihiro Hayashi, and Keisuke Fujii. 2020. Qulacs: a fast and versatile quantum circuit simulator for research purpose. https://arxiv.org/abs/2011.13524.

[42] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *S&P '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 48–62. doi:10.1109/SP.2013.13

[43] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proceedings of the ACM on Programming Languages, Issue. PLDI* 7, 111 (June 2023), 24 pages. doi:10.1145/3591225

[44] Mohamed Tarek Ibn Ziad and Christos Kozyrakis. 2026. Reproduction Package for Article 'Hunting CUDA Bugs at Scale with cuFuzz'. Zenodo. doi:10.5281/zenodo.18727045

[45] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *MICRO'19: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA, 372–383. doi:10.1145/3352460.3358307

[46] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: fuzzing deep-learning libraries from open source. In *ICSE'22: Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh, Pennsylvania, 995–1007. doi:10.1145/3510003.3510041

[47] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *ISSTA'22: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual, South Korea, 176–188. doi:10.1145/3533767.3534220

[48] Yuhao Zhou, Peng Jia, Jiayong Liu, and Ximing Fan. 2026. Fuzz4Cuda: A fuzzing framework for GPU library vulnerability detection. *Computers & Security* 161 (2026). doi:10.1016/j.cose.2025.104754

Received 10 October 2025; revised 03 February 2026; accepted 17 February 2026