USENIX Security '26 Artifact Appendix: Fuzzing Open-Source GPU Hardware with SIMT Program Generation

Zibo Gao[†§], Jie Wang[†§], Qihang Zhou[†§], Lixiao Shan[†§], Junjie Hu[†§], Xiaoqi Jia [†§], and Zhiqiang Lv[†§]
[†]Institute of Information Engineering, Chinese Academy of Sciences.
[§]School of Cyber Security, University of Chinese Academy of Sciences.
{gaozibo,wangjie2024,zhouqihang,shanlixiao,hujunjie,jiaxiaoqi,lvzhiqiang}@iie.ac.cn

A Artifact Appendix

A.1 Abstract

This artifact implements FuzzGPU, a GPU fuzzing framework for automated vulnerability discovery at the Register Transfer Level (RTL). It uses a generator to produce valid SIMT programs with diverse control and data flows, stressing GPU execution behavior. It also handles microarchitectural nondeterminism (e.g., caches and memory timing) and enables trace-based comparison against an ISA oracle for instruction-level bug localization. Evaluation shows FuzzGPU finds new RTL and ISS bugs. To the best of our knowledge, no publicly available GPU hardware fuzzer exists for comparison. We therefore adapted Cascade and DiveFuzz to GPUs as baselines that generate instruction-level data and control flows. Experimental results show that FuzzGPU achieves higher coverage and throughput than the GPU ports of Cascade and DiveFuzz. We also package the Devices Under Test (DUTs), including Vortex and Ventus OpenGPGPU. Overall, this artifact demonstrates FuzzGPU's functionality as presented in the paper and enables reproduction of the main results that motivate its design.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

FuzzGPU targets RTL designs and runs in simulation, and thus not compromises the host system executing the experiments. Broader ethical and safety considerations are discussed in the paper.

A.2.2 How to access

The artifact is available on Zenodo (https://doi.org/10.5281/zenodo.20321391) and can also be cloned via Git:

```
git clone --depth 1 --recursive \
https://github.com/cassuto/fuzzgpu.git
```

A.2.3 Hardware dependencies

To reproduce the performance results, we recommend using an Intel(R) Xeon(R) 6982P-C system with 192 vCPUs and 768 GB of RAM.

A.2.4 Software dependencies

Our artifact requires a Linux distribution, `make`, and `Docker v29.5.3` as prerequisites. The Dockerfile automatically installs the remaining dependencies. Experiments are designed to run inside the Docker container.

A.2.5 Benchmarks

FuzzGPU was evaluated on two open-source GPUs, Vortex and Ventus OpenGPGPU. We also compare its performance against GPU ports of two CPU fuzzers: Ported-Cascade and Ported-DiveFuzz.

A.3 Set-up

A.3.1 Installation

Clone the Git repository and build the Docker image:

```
make docker
```

This step performs clean installation. It downloads, builds, and installs all dependencies and configures the metadata. A stable Internet connection is required to avoid missing-file errors. Proxy options are provided in the Makefile and Dockerfile. This step may take up to 24 machine-hours and consume ~200 GB of disk space.

A.3.2 Basic Test

Run smoking tests:

```
scripts/run_smoking_tests.sh --all
```

The expected successful output is:

```
================================================
>>> SMOKING SUITE RESULT: PASS
>>> units_total=6 units_pass=6 units_fail=0
================================================
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FuzzGPU significantly improves instruction completion rate. This is shown in Figure 10.
(C2): FuzzGPU generates memory relation edges, including *co*, *coe*, *rf*, *rfe*, *rfi*, *fr*, and *ppo*. This is shown in Table 2.
(C3): FuzzGPU achieves higher generation throughput than Ported-Cascade and Ported-DiveFuzz. FuzzGPU also achieves higher effective throughput than Ported-DiveFuzz and comparable performance to Cascade. This is shown in Figures 11 and 12.
(C4): The instruction generation time of FuzzGPU accounts for a smaller fraction than RTL emulation time in the per-test fuzzing time breakdown. This is shown in Figure 14.
(C5): FuzzGPU improves coverage over Ported-Cascade and Ported-DiveFuzz. This is shown in Figure 15.
(C6): Ablation shows that removing any component (generation stack, barrier planner, or taint propagation) reduces program metrics and coverage. This is shown in Section 5.4.
(C7): FuzzGPU finds most of the bugs very fast. This is shown in Figure 16.

A.4.2 Experiments

(E1): [5 human-minutes + 24 machine-hour + 200GB disk]: Build the Docker image, refer to A.3.
How to: Build image using the provided Dockerfile.
Preparation: None.
Execution: In the repository, execute `make docker`.
Results: The Docker image will ultimately be built successfully, as confirmed by Docker's output.
(E2): [5 human-minutes + 1 machine-hour]: Instruction completion rate (validating C1).
How to: Measure the instruction completion rate.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E2.sh`.
Results: `results/plots/instr_complete_rate.pdf` reproduces Figure 10.
(E3): [5 human-minutes + 8 machine-hour]: Throughput measurement (validating C3).
How to: Measure and compare the generation and effective throughput.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E3.sh`.
Results: `results/plots/gen_throughput_vortex_xlen64.pdf` reproduces Figure 11. `results/plots/effective_throughput_vortex_xlen64.pdf` reproduces Figure 12.
(E4): [5 human-minutes + 5 machine-hour]: Coverage measurement (validating C5).
How to: Measure the coverage.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E4.sh`.
Results: `results/plots/coverage_comparison_ventus_x32_combined.pdf` reproduces Figure 15.
(E5): [5 human-minutes + 2 machine-hour]: Ablation study (validating C6).
How to: Run the ablation study evaluation.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E5.sh`.
Results: It reproduces Section 5.4.
(E6): [5 human-minutes + 5 machine-hour]: Fuzzing campaign (validating C2, C4, and C7).
How to: Run fuzzing campaign.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E6.sh`.
Results: It reproduces Table 2, Figure 14 (`results/plots/time_breakdown_fuzzgpu.pdf`), and Figure 16 (`results/plots/bug_discover_time.pdf`).

A.5 Notes on Reusability

- We provide reduced-parameter guidance in `README.md`. In particular, this file explains how to scale down `scripts/run_E2.sh` – `scripts/run_E6.sh` by editing the top-level parameters in the corresponding `scripts/run_*.sh` files, such as the number of seeds, runtime limits, target architectures, methods, coverage types, and ablation configurations. These reduced runs are intended to complete faster while preserving the qualitative trends.

- For fuzzer extension, Doxygen generates API documentation at `docs/html/index.html`, covering the generator, architecture-support, and difftest interfaces. Using Graphviz `dot`, it also visualizes inheritance, collaboration, include, and directory graphs to help identify extension points for new RTL designs, ISS backends, and generator modules.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.