GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs

Dian-Lun Lin
University of Utah
Salt Lake City, UT, USA
dian-lun.lin@utah.edu

Yanqing Zhang
Nvidia Corporation
Santa Clara, CA, USA
yanqingz@nvidia.com

Haoxing Ren
Nvidia Corporation
Austin, TX, USA
haoxingr@nvidia.com

Brucek Khailany
Nvidia Corporation
Austin, TX, USA
bkhailany@nvidia.com

Shih-Hsin Wang
University of Utah
Salt Lake City, UT, USA
shwang@math.utah.edu

Tsung-Wei Huang
University of Utah
Salt Lake City, UT, USA
tsung-wei.huang@utah.edu

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247942

***Abstract*—Hardware fuzzing has emerged as a promising automatic verification technique to efficiently discover and verify hardware vulnerabilities. However, hardware fuzzing can be extremely time-consuming due to compute-intensive iterative simulations. While recent research has explored several approaches to accelerate hardware fuzzing, nearly all of them are limited to single-input fuzzing using one thread of a CPU-based simulator. As a result, we propose Gen-Fuzz, a GPU-accelerated hardware fuzzer using a genetic algorithm with multiple inputs. Measuring experimental results on a real industrial design, we show that GenFuzz running on a single A6000 GPU and eight CPU cores achieves 80× runtime speed-up when compared to state-of-the-art hardware fuzzers.**

I. Introduction

The ever-increasing complexity of hardware design has put significant strain on System-on-Chip (SoC) designers and system integrators to detect and evaluate hardware vulnerabilities within the design stage [1]–[3]. As SoC complexity continues to grow, industry-quality functional verification signoff typically requires a significant and growing amount of engineering effort to generate and simulate many thousands of test cases on the same Design-Under-Test (DUT) for converging on coverage closure and avoiding bug escape from corner cases. Much research over the past decades has focused on automatic *constrained random verification* (CRV) [4] to relieve the increasing strain for hardware engineers. CRV tests a DUT by randomly combining manually-defined inputs (i.e., instructions plus compiled RTL stimulus) into transaction sequences. However, since CRV relies only on randomly generated inputs, it suffers from zero knowledge of coverage and becomes inefficient when verifying large designs.

Coverage-guided verification, also known as *hardware fuzzing*, has emerged as a promising automatic hardware verification technique to efficiently discover and verify hardware vulnerabilities [1], [5]. Unlike CRV that randomly combines inputs, hardware fuzzing generates inputs by mutating previously interesting inputs (i.e., the inputs that increase coverage) to effectively discover unknown hardware behaviors. However, since this process requires time-consuming feedback analysis, hardware fuzzing often takes hours or days to finish.

To alleviate the long runtime, recent research has explored several approaches to speed up the single-input, single-threaded, per fuzzing iteration time. RFUZZ [1] proposes a mux-coverage metric that treats the select signal of each 2:1 multiplexer as a coverage point. However, RFUZZ cannot scale to large designs since their runtime grows significantly as the number of multiplexers increases. DirectFuzz [2] extends RFUZZ to generate test inputs that maximize the coverage of a specific block. Compared to RFUZZ, DirectFuzz can improve performance on small designs. However, the speedup on complex designs is insignificant (e.g., 1.08× on the Sodor1Stage RISC-V processor). Also, their work only targets RFUZZ's mux coverage and is not generalizable to other coverage metrics. DIFUZZRTL [5] introduces a reg-coverage metric to monitor value changes of control registers connected to mux control signals. While DIFUZZRTL's reg coverage shows capability for large designs, their fuzzing technique requires many hours or days to achieve high coverage.

TheHuzz [6] explores processor states using multiple coverage metrics. However, it induces over 70% runtime overhead when collecting coverage data since it must access multiple metrics per fuzzing iteration. Hw-Fuzzing [4] converts a hardware-description-language (HDL) model into an equivalent software model using Verilator, and performs fuzzing on the software code using software coverage metrics. Although it shows software coverage metrics are comparable with HDL line coverage, other hardware coverage metrics such as finite-state-machine (FSM) coverage cannot be easily added. Other research has leveraged FPGAs to accelerate hardware fuzzing [1], [5]. However, there are three drawbacks of FPGA-based hardware fuzzing: 1) It suffers from a complicated compilation setup. 2) It does not provide visibility to internal signals, complicating bug detection. 3) Instrumenting a design on an FPGA is challenging [1], [5].

While all these approaches have shown coverage or runtime improvements, *nearly all of them are limited to single-input*

	RFUZZ	DirectFuzz	DIFUZZRTL	TheHuzz	Hw-Fuzzing	**GenFuzz**
FPGA	Y	N	Y	N	N	N
CPU	single thread	single thread	single thread	single thread	single thread	**multiple threads**
GPU	N	N	N	N	N	**Y**
#Inputs	1	1	1	1	1	**multiple**

Fig. 1. Comparison between GenFuzz and existing hardware fuzzers.

fuzzing using one thread of simulation on a CPU architecture. Recently, RTL simulation research has achieved significant performance improvement by leveraging GPUs to simulate multiple inputs simultaneously [3]. This result inspires us to accelerate hardware fuzzing by exploring data parallelism, which we refer to as *multi-input hardware fuzzing*, using CPU-GPU heterogeneous computing. However, multi-input hardware fuzzing is extremely challenging for three reasons. Firstly, existing works focus on speeding up single-input fuzzing using sequential mutation frameworks. Multi-input hardware fuzzing requires a new CPU-GPU task decomposition strategy to benefit from heterogeneous parallelism. Secondly, multi-input hardware fuzzing needs an effective mutation algorithm to find the best previous inputs as seeds and generate multiple new inputs of interest. Lastly, fuzzing multiple inputs in parallel can introduce inefficiencies from highly overlapped coverage within a fuzzing iteration. We need to rule out unwanted inputs that cause redundant overlaps.

To overcome these challenges, we propose GenFuzz, a GPU-accelerated hardware fuzzer using a genetic algorithm (GA) with multiple inputs. Figure 1 compares the key differences between GenFuzz and existing hardware fuzzers. To the best of our knowledge, this is the first GPU-accelerated hardware fuzzing using GA in the literature. We summarize three key contributions as follows:

- We design an efficient **multi-input hardware fuzzer** to fuzz multiple inputs simultaneously using both CPU and GPU parallelisms.
- We design an effective **GA-based framework** to iteratively produce multiple inputs of interest that are most likely to extend the coverage.
- We design a novel **coverage-maximization algorithm** to avoid overlapped coverage for both inter- and intra-fuzzing iterations.

We have evaluated GenFuzz on real designs and demonstrated its promising performance compared to the state-of-the-art DIFUZZRTL [5] and RFUZZ [1]. GenFuzz using one A6000 GPU and eight CPU cores outperforms DIFUZZRTL on single CPU thread with up to 80× speed-up for BOOMCore design using reg coverage. We have also shown that GenFuzz achieves 2.1× more coverage points when the same number of instructions are fuzzed. We will make GenFuzz open-source to benefit the community and inspire hardware fuzzing research with heterogeneous parallelism.

II. Background

A. Conventional Hardware Fuzzing

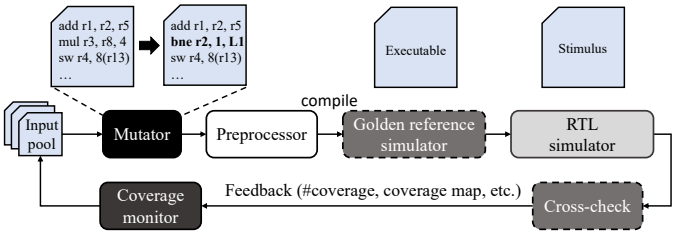

Fig. 2. Conventional single-input hardware fuzzing flow.

Figure 2 shows the flow of conventional single-input hardware fuzzing [5]. We start by randomly choosing one input from the input pool that maintains a set of interesting inputs. The mutator generates a new input by mutating instructions from the selected input. The preprocessor compiles the input into a stimulus and an executable for simulation. Depending on different hardware fuzzers and coverage metrics, we may need a reference simulator to validate an input. We then use an RTL simulator to simulate the DUT with the mutated stimulus and collect the coverage data. The input that discovers new hardware states (i.e., new coverage points) is considered interesting and is saved back to the input pool for future fuzzing. The fuzzing iteration continues until we cannot explore new states for a maximum number of iterations. Finally, we use assertions or cross-check RTL simulation results against results from the reference simulator to detect bugs.

B. Genetic Algorithm

GA [7] is a search heuristic that reflects the process of natural selection. The best individuals are selected to produce good offsprings for the next generation. Listing 1 gives an example. The code wraps all individuals with a population and applies GA iteratively. A quantitative fitness function evaluates the quality of each individual in each iteration. We then select individuals with better fitness as parents to produce superior offsprings. For each parent pair, we perform crossover to exchange genes between parents to generate offsprings. The new offsprings become a new population for the next GA iteration. The iteration continues until we cannot find a better solution after a maximum number of iterations.

```
Population pop;              // construct a population
Individuals pars;            // construct parents
Individuals offs;            // construct offsprings
pop.initialize();            // initialize a population
while(iter < MAX_ITER) {
  pop.caculate_fitness(). // calculate  fitness
  pars = pop.select();       // select good parents
  offs = pars.crossover();// generate new offsprings
  offs.mutate();             // mutate to add diversity
  pop.replace(offs);         // replace old population
}
```

Listing 1. A common C++ genetic algorithm.

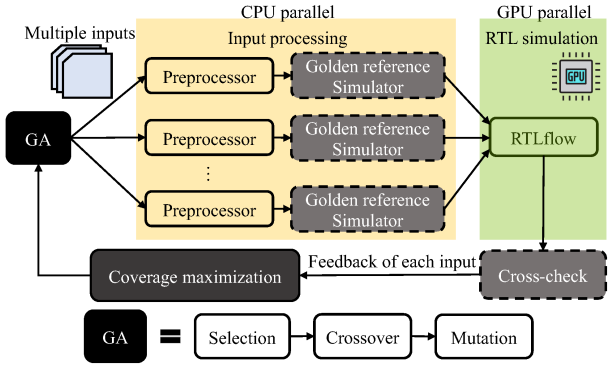

Fig. 3. Overview of GenFuzz.

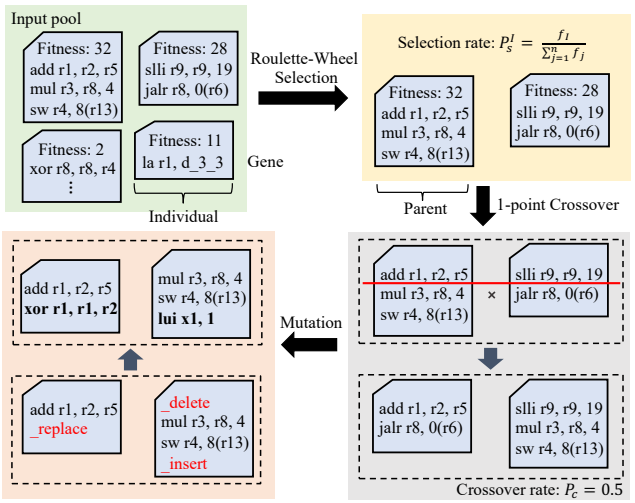

Fig. 4. Overview of our GA-based fuzzing framework.

III. GenFuzz

A. Multi-input Hardware Fuzzing

At a high level, GenFuzz efficiently discovers new coverage by fuzzing multiple inputs in parallel at each fuzzing iteration. Figure 3 shows the overview of GenFuzz. We define the multi-input hardware fuzzing workload as a parallel task dependency graph that iterates five stages: *GA*, *input processing*, *RTL simulation*, *cross-check* and *coverage maximization*. At the GA stage, GenFuzz incorporates three GA processes, *selection*, *crossover*, and *mutation*, to generate inputs that are most likely to extend coverage. Based on the results from the previous fuzzing iteration, we select the best inputs as parents to generate new inputs. The input processing stage involves CPU-intensive tasks including the compilation of simulation inputs and file I/O. Without data parallelism, RTL simulation needs to wait until we process all inputs, thus incurring significant overhead. To improve runtime performance, we evenly distribute inputs across different CPUs to process each input in parallel.

We integrate the state-of-the-art RTL simulator, RTLflow [3], into GenFuzz to enable GPU acceleration for multi-stimulus RTL simulation. Unlike existing hardware fuzzers that typically use Verilator or ModelSim to simulate one stimulus at a time, RTLflow achieves high-throughput RTL simulation by running multiple stimuli in parallel using GPU. After the RTL simulation, we cross-check results derived from the reference simulator and RTLflow. We do not parallelize cross-check since this stage is fast (e.g., a few seconds to finish). Our coverage-maximization algorithm scores each input by analyzing the feedback of each input. Finally, we send each input with its fitness to GA for selection. The fuzzing iteration continues until GA cannot find new coverage after a maximum number of iterations.

B. GA-based Framework

The goal of our GA-based framework is to produce new inputs that can maximally extend coverage using results from the previous fuzzing iteration. Our GA consists of *selection*, *crossover*, and *mutation*. Unlike existing mutation approaches that only select one input to mutate, our GA framework exchanges interesting instructions between two parents and passes instructions to newborn inputs. Furthermore, we adopt variable-length individual representation. Each input can have a different number of instructions, allowing us to efficiently explore coverage on the time dimension.

Figure 4 shows the overview of our GA-based framework. Each gene represents an instruction, and each individual consists of multiple genes to form an input. Our GA framework can be applied to arbitrary coverage metrics and inputs. For example, to perform fuzzing on the RTL level using mux coverage [1], we can map the input value of each input pin as a gene. Each individual thus concatenates multiple genes to form a stimulus. Since the mux coverage is collected after RTL simulation, it can be transformed to fitness for the selection process.

1) Selection: The selection process chooses individuals with higher fitness from the input pool for later reproduction. We apply Roulette-Wheel Selection (RWS) for our framework such that the probability of choosing an individual is proportional to its fitness. Compared to truncation selection which directly eliminates a fixed percentage of the weakest candidates, RWS can still select individuals with lower fitness to create more diversity and to avoid quick convergence. We define the selection rate P_s^I of input I as $P_s^I = \frac{f_I}{\sum_{j=1}^{n} f_j}$ where f_I is the fitness of input I and n is the number of input.

2) Crossover: The crossover process allows parents to exchange instructions for producing the next generation of individuals. We choose two parents from selected individuals and apply one-point crossover. Since individuals can have different lengths of genes, we randomly choose a crossover point based on the parent with smaller length. As shown in Figure 4, genes of both parents after the crossover point (red line) are interchanged. We choose a crossover rate $P_c = 0.5$ to randomly determine whether two parents occur crossover. If crossover does not happen, the two parents are considered as newborn inputs and passed into the mutation process.

3) Mutation: The mutation process provides a mechanism for newborn inputs to escape from local regions and to create

more diversity. During the mutation phase, we iterate each gene in an individual and decide if the gene is mutated using the mutation rate P_m. Our mutation contains three operators: *insert*, *delete*, and *replace* as shown in Figure 4. Once mutation occurs, we randomly choose one operator to mutate the gene.

The mutation rate P_m plays an important role in the mutation process. If the mutation rate is minimal, there will be too many similar individuals. On the other hand, having a large mutation rate can easily direct GA toward random search. To have a better mutation rate for effective GA search, we use a time-dependent mutation rate proposed by [7]:

$$P_m = \begin{cases} 0.6\left[1-(\frac{t}{T})^2\right], & 0 \le t \le 0.2T \\ 0.2\left[0.1(\frac{t-T}{T})^2\right]+0.05, & 0.2T < t \le T \end{cases}$$

where T is the total number of fuzzing iterations and t is the t^{th} iteration.

C. Coverage-Maximization Algorithm

Multi-input hardware fuzzing can introduce inefficiencies due to highly overlapped coverage within a fuzzing iteration. Moreover, selecting inputs with large overlap as parents causes newborn inputs to inherit the same overlap. The overlap grows significantly as we increase the number of fuzzing iterations. To overcome this problem, a greedy solution is to find the top-k inputs based on the maximum coverage problem, defined as follows:

Definition 1. *Let $\{I_j\}_{j=1}^{n}$ be the sequence of inputs where I_j is the $j-$th input and n is the number of these inputs. Let $C(I_j)$ represent the set of coverage discovered by input I_j. Then, given a positive number $k \le n$, the goal of the maximum coverage problem is to find a subsequence $\{I_{j_l}\}_{l=1}^{k} = \{I_{j_1}, I_{j_2}, ..., I_{j_k}\}$, called the top-k inputs in $\{I_j\}_{j=1}^{n}$, such that their total coverage $|\bigcup_{l=1}^{k} C(I_{j_l})|$ is maximized.*

Unfortunately, this problem is NP-hard [8]. Also, existing greedy algorithms that choose one input with the largest uncovered coverage at a time cannot be used out of the box. Specifically, after selecting the best input, greedy algorithms require remaining inputs to re-calculate uncovered coverage by iterating all coverage for each input. The time complexity of greedy algorithms is thus $O((cov\_size * n)^2)$ where cov_size is the coverage size in a design and n is the number of inputs per fuzzing iteration. Since the number of coverage points is in the millions for large designs, such greedy algorithms are extremely time-consuming.

To reduce the time complexity, we introduce two coverage metrics, *delta* and *progressive* coverage, as fitness for GA to select inputs:

Definition 2. *The delta coverage $C_d(I_j)$ measures how much new coverage is discovered by I_j compared to the total coverage explored in previous fuzzing iterations.*

Definition 3. *The progressive coverage $C_p(I_j)$ measures how much new coverage is discovered by I_j compared to the total coverage explored by all inputs before I_j.*

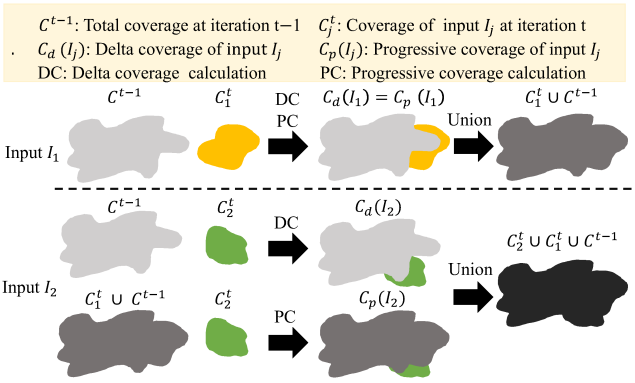

Fig. 5. The proposed progressive coverage and delta coverage calculation. The progressive coverage and delta coverage of the first input is identical.

Figure 5 shows an example of calculating delta and progressive coverage for two inputs I_1, I_2. We represent the total explored coverage at iteration $t-1$ by C^{t-1} and the coverage of input I_j at iteration t by C_j^t.

Since all inputs before the first input are in previous fuzzing iterations, $C_d(I_1) = C_p(I_1)$ are both given by taking the complement of C^{t-1} w.r.t. C_1^t. Similar to $C_d(I_1)$, we calculate $C_d(I_2)$ by taking the complement of C^{t-1} w.r.t. C_2^t. On the other hand, we take the union of C^{t-1} and C_1^t to get the total coverage explored by all the inputs before I_2. Finally, we calculate $C_p(I_2)$ by taking the complement of $C^{t-1} \cup C_1^t$ w.r.t. C_2^t.

Based on our delta and progressive coverage metrics, we define the fitness function as follows:

$$Fitness(I_j) = Norm(C_d(I_j)) + Norm(C_p(I_j))$$

where $Norm$ represents mix-max normalization that transforms two coverage metrics to the same scale.

Algorithm 1 shows our coverage-maximization algorithm. Each input owns a coverage map of size cov_size to record its coverage. The $total\_cov\_map$ represents total coverage by taking a union of all inputs. In the beginning, we initialize $total\_cov$ and $total\_cov\_map$ by using coverage explored in previous fuzzing iterations (lines 3-4). During delta coverage calculation, each input iterates its coverage map and compares coverage results with $prev\_total\_cov\_map$ (lines 6-14). If an input discovers a new coverage that was not explored in previous fuzzing iterations, we increment the delta coverage of the input by one (lines 10-12). During progressive coverage calculation, each input iterates its coverage map again and compares coverage results with $total\_cov\_map$ (lines 16-26). If an input discovers a new coverage that was not toggled in $total\_cov\_map$, we increment the progressive coverage of the input and $total\_cov$ by one (lines 22 and 23). Also, we toggle the corresponding coverage in $total\_cov\_map$ to one (line 21). In this case, the next input that discovers the same coverage cannot increment its progressive coverage. After coverage calculation, we normalize both delta and progressive coverage (line 27). Finally, we calculate the fitness for each input by summing up its delta and progressive coverage (lines 29-31).

Algorithm 1: Coverage-maximization algorithm

Input: NUM_INPUTS: Number of inputs
Input: COV_SIZE: Coverage size
Input: $prev\_total\_cov$: Total coverage at previous iter
Input: $prev\_total\_cov\_map$: Total coverage map at previous iter
Input: cov_maps: Array of coverage maps
Output: $total\_cov$: Total coverage
Output: $total\_cov\_map$: Total coverage map
Output: $fitness$: Array of fitness

$delta\_covs$.initialize(0)
$prog\_covs$.initialize(0)
$total\_cov \leftarrow prev\_total\_cov$
$total\_cov\_map \leftarrow prev\_total\_cov\_map$
$i, c \leftarrow 0$
/* delta coverage caculation */
while $i++ < NUM\_INPUTS$
  while $c++ < COV\_SIZE$
    $result \leftarrow cov\_maps[i][c]$
    $prev\_result \leftarrow prev\_total\_cov\_map[c]$
    if $prev\_result == 0$ **and** $result == 1$ **then**
      $delta\_covs[i]++$
    end
  end
end
$i, c \leftarrow 0$
/* progressive coverage calculation */
while $i++ < NUM\_INPUTS$
  while $c++ < COV\_SIZE$
    $result \leftarrow cov\_maps[i][c]$
    $prev\_result \leftarrow total\_cov\_map[c]$
    if $prev\_result == 0$ **and** $result == 1$ **then**
      $total\_cov\_map[c] = 1$
      $prog\_covs[i]++$
      $total\_cov++$
    end
  end
end
min_max_norm($delta\_covs$, $prog\_covs$)
$i \leftarrow 0$
while $i++ < NUM\_INPUTS$
  $fitness[i] \leftarrow delta\_covs[i] + prog\_covs[i]$
end

We can clearly see the time complexity of our algorithm is $O(cov\_size * n)$.

IV. Experimental Results

We conducted our experiments on a 64-bit CentOS Linux machine with one NVIDIA RTX A6000 GPU and eight Intel i7-11700 CPU cores at 2.5 GHz. We compiled our programs with CUDA NVCC 11.6 on a GCC 8.3.0 host compiler and enabled optimization flag `-O3` and C++17 standard `-std=c++17`. Each fuzzing iteration has an input size of 1024. For each run, we used 1024 GPU threads for RTL simulation and 8 CPU cores for all host operations. We used Taskflow [9], [10] to parallelize our task dependency graph. All data is an average of five runs.

We consider two state-of-the-art hardware fuzzers, RFUZZ [1] and DIFUZZRTL [5], as our baselines. RFUZZ and DIFUZZRTL each proposed coverage metrics (i.e., mux coverage and reg coverage) to measure the number of discovered design states. We compare GenFuzz with each fuzzer using their coverage metrics. DIFUZZRTL performs fuzzing on its CPU input format at the instruction level. It combines several instructions into a word and performs per-word mutations. On the other hand, RFUZZ directly fuzzes at the RTL level. It concatenates all input pins as an input vector to represent input values in one test cycle. To explore coverage on the time dimension, it further concatenates several single-cycle stimuli to form a multi-cycle stimulus. For a fair comparison, we perform fuzzing at the same level as DIFUZZRTL and RFUZZ. We use Verilator as the RTL simulator for RFUZZ and DIFUZZRTL. For single-input hardware fuzzing, RTLflow is slower than Verilator due to little data parallelism [3]. To demonstrate our efficiency, we evaluate GenFuzz on five real designs, BoomCore1, BoomCore2, RocketCore, Sodor3Stage, and Sodor5Stage, provided by RFUZZ and DIFUZZRTL.

A. Overall Performance Comparison

Table I compares the overall runtime between GenFuzz and DIFUZZRTL on RocketCore, BoomCore1, and BoomCore2. We run DIFUZZRTL for 48 hours to derive the total number of coverage (#coverage), and compare runtime of GenFuzz and DIFUZZRTL for achieving 50%, 70%, and 100% #coverage. GenFuzz outperforms DIFUZZRTL in all scenarios. For achieving 100% #coverage, GenFuzz is 61× faster on RocketCore and is 80× faster on BOOMCore1. The significant improvement on runtime demonstrates the promise of our multi-input hardware fuzzing techniques. The speedup of achieving 100% #coverage is larger than 50% and 70% #coverage. When discovered coverage becomes large, the coverage of DIFUZZRTL starts to saturate. On the other hand, our genetic algorithm keeps finding new inputs of interest and thus efficiently increases the coverage. Figure 6 compares the coverage throughput among GenFuzz, DIFUZZRTL, and RFUZZ on different designs. To demonstrate the efficiency of coverage throughput, we use the number of words and cycles as the x-axis for different coverage metrics. Compared to DIFUZZRTL using reg coverage, GenFuzz achieves 2.1× speed-up using the same amount of words. The throughput gap continues to grow as we increase the number of inputs. Compared to RFUZZ using mux coverage, GenFuzz achieves 1.6× speed-up using the same amount of cycles. The number of coverage quickly saturates since both Sodor3Stage and Sodor5Stage are small designs that do not have many coverage points to discover.

B. Performance Result of Coverage-maximization Algorithm

In this section, we study the performance benefit of our coverage-maximization algorithm. Figure 7 shows total coverage over increasing numbers of fuzzing iterations for GenFuzz, random (GenFuzz^{r}), and GenFuzz without coverage-maximization algorithm (GenFuzz^{-cm}). In GenFuzz^{r}, we set the mutation rate to 1 to achieve random testing. In GenFuzz^{-cm}, each input uses the coverage number achieved by itself as fitness. We can clearly see the performance advantage of our coverage-maximization algorithm. GenFuzz achieves a higher coverage number than other implementations in almost all scenarios. The performance gap continues to

TABLE I

OVERALL PERFORMANCE COMPARISON BETWEEN DIFUZZRTL AND GENFUZZ ON DIFFERENT BENCHMARKS FOR ACHIEVING 50%, 70%, AND 100% COVERAGE USING REG COVERAGE. THE BENCHMARK STATISTICS SHOW VERILOG LINES OF CODE (VERILOG LOC) AND NUMBER OF ACHIEVED COVERAGE (#COVERAGE) BY RUNNING DIFUZZRTL FOR 48 HOURS. BOLD TEXT REPRESENTS SPEED-UP.

			Runtime (s) for achieving K% #Coverage					
			K=50		K=70		K=100	
Benchmark	Verilog LOC	#Coverage	DIFUZZRTL	GenFuzz	DIFUZZRTL	GenFuzz	DIFUZZRTL	GenFuzz
RocketCore	81883	110509	6218s	578s (**10.8**×)	18972s	1216s (**15.6**×)	172800s	2835s (**61.0**×)
BoomCore1	193689	518506	7503s	484s (**15.5**×)	22278s	1180s (**18.9**×)	172800s	2160s (**80.0**×)
BoomCore2	239282	684202	9208s	835s (**11.0**×)	24555s	1428s (**17.2**×)	172800s	2690s (**64.2**×)

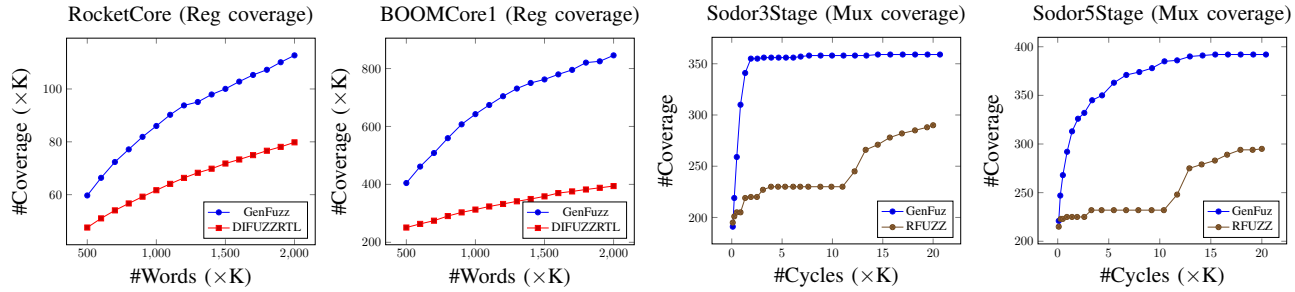

Fig. 6. Comparison of coverage throughput among GenFuzz, DIFUZZRTL, and RFUZZ on RocketCore, BOOMCore, Sodor3Stage, and Sodor5Stage. The x-axis uses the number of words and the number of cycles.

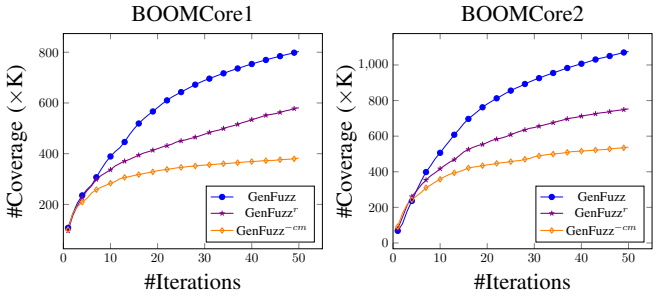

Fig. 7. Coverage growth over increasing numbers of iterations for GenFuzz, GenFuzz with mutation rate $P_r = 1$ (GenFuzz^r), and GenFuzz without coverage-maximization algorithm (GenFuzz^{-cm}) on BOOMCore using reg coverage with 256 inputs.

enlarge as we increase the number of fuzzing iterations. Without our coverage-maximization algorithm, both GenFuzz^r and GenFuzz^{-cm} fail to distinguish overlapped coverage within each fuzzing iteration, thus misguiding GA to select inputs that are less likely to extend coverage. The coverage growth of GenFuzz^{-cm} is even slower than GenFuzz^r because GenFuzz^{-cm} does not consider coverage improvement at each fuzzing iteration. GA converges within 20 iterations and hardly discovers new coverage points.

V. CONCLUSION

In this paper, we have introduced GenFuzz, a GPU-accelerated hardware fuzzer using a novel genetic algorithm to speed up hardware fuzzing with multiple inputs. GenFuzz introduces a multi-input hardware fuzzer, a genetic algorithm-based framework, and a coverage-maximization algorithm to accelerate hardware fuzzing to a new performance milestone. Measuring experimental results on real industrial designs, GenFuzz achieves up to 80× runtime speed-up when compared to DIFUZZRTL and RFUZZ.

ACKNOWLEDGMENT

We are grateful for the support of four National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523 (CAREER), OAC-2209957, and TI-2229304.

REFERENCES

[1] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *IEEE ICCAD*, 2018.
[2] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *ACM/IEEE DAC*, 2021.
[3] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus," in *ACM ICPP*, 2022.
[4] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *USENIX SECURITY*, 2022.
[5] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *IEEE SP*, 2021.
[6] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *USENIX SECURITY*, 2022.
[7] K. Tan, C. Goh, Y. Yang, and T. Lee, "Evolving better population distribution and exploration in evolutionary multi-objective optimization," *European Journal of Operational Research*, 2006.
[8] D. S. Hochba, "Approximation algorithms for np-hard problems," *SIGACT News*, 1997.
[9] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE TPDS*, 2022.
[10] D.-L. Lin and T.-W. Huang, "Efficient gpu computation using task graph parallelism," in *Euro-Par*, 2021.