Leveraging Hardware Probes and Optimizations for Accelerating Fuzz Testing of Heterogeneous Applications

Jiyuan Wang
University of California, Los Angeles
USA
wangjiyuan@cs.ucla.edu

Qian Zhang
University of California, Riverside
USA
qzhang@cs.ucr.edu

Hongbo Rong
Intel Lab
USA
hongbo.rong@intel.com

Guoqing Harry Xu
University of California, Los Angeles
USA
harryxu@cs.ucla.edu

Miryung Kim
University of California, Los Angeles
USA
miryung@cs.ucla.edu

ABSTRACT

There is a growing interest in the computer architecture community to incorporate heterogeneity and specialization to improve performance. Developers can create *heterogeneous applications* that consist of both *host* code and *kernel* code, where compute-intensive kernels can be offloaded from CPU to hardware accelerators. Testing such applications on *real* heterogeneous architectures is extremely challenging as kernels are black boxes, providing no information about the kernels' internal execution to diagnose issues such as silent hangs or unexpected results. Additionally, inputs for heterogeneous applications are often large matrices, leading to a vast search space for identifying bug-revealing inputs.

We propose a novel fuzz testing technique, HFuzz, to enable efficient testing on real heterogeneous architectures. HFuzz aims to increase both the observability of hardware kernels and testing efficiency through a three-pronged approach. First, HFuzz automatically generates test guidance by inserting device-side in-kernel hardware probes in addition to host-side software monitors. Second, it performs rapid input space exploration by offloading compute-intensive input mutations to hardware kernels. Third, HFuzz parallelizes fuzzing and enables fast on-chip memory access, by utilizing four FPGA-level optimizations including loop unrolling, shannonization, data preloading, and dynamic kernel sharing.

We evaluate HFuzz on seven open-source OneAPI subjects from Intel. HFuzz speeds up fuzz testing by 4.7× with HW-accelerated input space exploration. By incorporating HW probes in tandem with SW monitors, HFuzz finds 33 defects within 4 hours and reveals 25 unique, unexpected behavior symptoms that could not be found by SW-based monitoring alone. HFuzz is the first to design hardware optimizations to accelerate fuzz testing.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computer systems organization** → **Heterogeneous**.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616318

KEYWORDS

Fuzz Testing, Heterogeneous Applications

ACM Reference Format:
Jiyuan Wang, Qian Zhang, Hongbo Rong, Guoqing Harry Xu, and Miryung Kim. 2023. Leveraging Hardware Probes and Optimizations for Accelerating Fuzz Testing of Heterogeneous Applications. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616318

1 INTRODUCTION

There has been a growing interest in developing specializable hardware accelerators for domain-specific workloads for various performance and energy benefits [11, 13, 16]. As an example, FPGA can be easily customized to accelerate applications across a wide variety of domains [9, 14] at lower power and higher performance than general-purpose CPUs [10, 23, 42]. Major hardware vendors are offering or plan to offer packages that include both CPUs and FPGAs [1, 18]. Such hardware packages have also been made into all major clouds to accelerate various analytic and learning tasks.

In recent years, fuzz testing has emerged as an effective test generation technique for large software systems [40]. Most fuzzing techniques, such as AFL [55], start from a seed input, generate new inputs by mutating the previous input, and add new inputs to the queue if they improve a given guidance metric such as branch coverage. In this paper, we focus on *fuzz testing* (i.e. *fuzzing*) of applications on a heterogeneous platform with a CPU host and an FPGA device. Such a *heterogeneous application* consists of *host* code and *kernel* code, and the host code offloads compute-intensive kernels from the CPU to the FPGA to run. Despite the potential benefits of FPGAs and their commercial availability to a broad user base, programming FPGAs is notoriously difficult in practice. Ensuring the correctness of FPGA programs, even seemingly-simple kernels, could take a substantial amount of time in terms of months [46]. As such, FPGA programming can be done by only a small handful of hardware experts [3, 35, 47]. Automatic fuzz testing of heterogeneous applications, together with root cause analysis of failures, can greatly simplify FPGA programming, thereby making FPGAs accessible to the masses.

There has been significant effort to ease the development of heterogeneous applications with FPGAs. The most successful effort

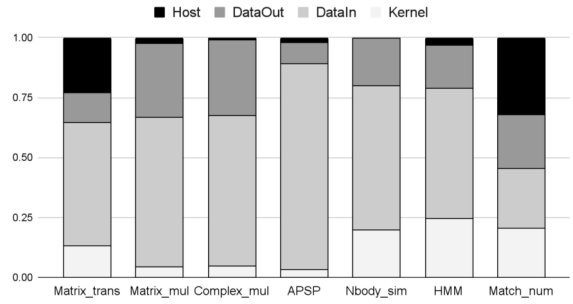

Figure 1: Latency breakdown of running applications on heterogeneous architectures. On average, data transfer into kernels takes 60% of execution time, highlighted in gray.

is *high-level synthesis* (HLS) [15]. HLS raises the level of programming abstraction from hardware description languages (such as Verilog) to C/C++ dialects (such as SYCL/DPC++[25]), enabling C/C++ developers on FPGAs. Even when heterogeneous applications are written in HLS languages, debugging and testing these heterogeneous applications can remain a significant challenge due to the following reasons:

Lack of Observability. FPGA is a device of massive parallelism. Little debugging support exists to help high-level programmers. Kernels run on an FPGA device as black boxes, and it often confuses programmers, e.g., when the kernels silently deadlock. General-purpose FPGA debugging [17, 33] works at the gate level. Even when in-circuit debugging information is available, it is difficult to correlate low-level gate signals with high-level variables in HLS programs.

Consider a scenario where an application multiplies two matrices `A` and `B` to create a new matrix `M`: M=A×B and then applies a reciprocal transformation on each element of `M`. This application has two kernels offloaded to FPGA: (1) `matrix_multiply` and (2) `transformer`. To transfer the intermediate result `M` from the first to the second kernel, a pipe is established to facilitate data transfer. For each element in the matrix `M`, the first kernel writes its computed value to the designated pipe, and the second kernel `transformer` reads it from the pipe, computes the reciprocal, and transfers the final result back to the host. With FPGA emulation, the application works as expected because both kernels run at the same speed. However, when run on an actual FPGA, the speed of the first kernel generating a value can be different from the speed of the second kernel consuming it. The developer should check the size of the pipe, delay writing if it is full, or delay reading if it is empty. If such check is not done, the pipe would be saturated or depleted, resulting in data loss and wrong reciprocal outcomes. Currently, due to a lack of observability into the dynamic usage of the pipe, the developer may find it difficult to diagnose the root cause.

Costly Transfer of Data with High Redundancy. Traditional iterative fuzzing techniques often mutate a small part of a seed input to generate new inputs. While this approach works well for many CPU programs, it is extremely ineffective for applications that are run on heterogeneous architectures. Inputs of heterogeneous applications are often large matrices and tensors, leading to significant data access and transfer overheads—the host, which mutates the matrices, must send newly mutated matrices (e.g., with only a few elements modified) to the device. Figure 1 illustrates the latency breakdown of running applications on Intel's heterogeneous architecture. On average, data transfer from CPU to hardware kernels takes 60% of the execution time. For a 100k×100k matrix, a single process of offloading the new generated matrix from the fuzzer to the device would take 2 minutes, prohibiting fast fuzzing on heterogeneous architectures.

Overlooked Opportunities for FPGA-level Optimizations. Fuzzing heterogeneous applications may be approached in a naïve manner by treating hardware kernel invocations as analogous to software function calls and repeatedly invoking them from an iterative input mutation loop. However, this approach ignores the potential optimizing capability of FPGA, as the mutations often consist of independent tasks that can be parallelized efficiently when offloaded to the FPGA side. In other words, the nature of fuzzing (i.e., iterative input generation and program invocation) unlocks new micro-architecture level performance optimizations. Indeed, we can treat the domain of heterogeneous applications, not only as a new target domain, but as a new enabler for accelerating automated test generation. When software-style matrix input mutation is offloaded to FPGA and is then combined with subsequent kernel invocation, many micro-architecture level optimizations such as loop unrolling, data preloading, shannonization, and dynamic kernel sharing are now applicable for further performance speed-up.

HFuzz. We developed HFuzz, a novel fuzz testing tool that aims to quickly reveal bugs in heterogeneous applications. Our key insights are elaborated below:

First, to improve error observability during testing, HFuzz injects *hardware probes inside the kernels* in tandem with software monitors inside the host. This is different from prior approaches that consider an FPGA kernel as a black box and inject software monitors only [58]. In HFuzz, both software monitors and hardware probes are designed to effectively detect overflows caused by intermediate variables within the FPGA kernel, as well as pipe saturation errors that may occur during data transfer between different devices. These hardware probes are injected through source-to-source transformation and then synthesized for FPGA. With timely execution feedback from the hardware probes, HFuzz prioritizes inputs that provide a new behavior signal at the FPGA execution level. For example, HFuzz monitors the saturation of a communication pipe between two FPGA kernels and retains the inputs that lead to a new maximum pipe saturation level for further mutations.

Second, HFuzz offloads input mutations into FPGA kernels to reduce unnecessary data transfer. For a `vector-add` example, instead of repeatedly transferring a mutated input vector of size 10^6, HFuzz retains the initial input vector in the FPGA buffer and mutates the elements of the vector within the FPGA kernel. For another example, the host-side mutation of a seed matrix with 10,000 elements for 1,000 times takes 9.1 seconds, in our evaluation, while in-kernel input mutation takes only 2.1 seconds.

Third, HFuzz implements four types of FPGA-level optimizations to speed up fuzzing. For example, one such optimization is *dynamic kernel sharing in parallel fuzzing loops*, which enables a more effective search space exploration when utilizing multiple input generators, each with its own seed queue. HFuzz then invokes the target kernel function using a mutated input selected from one of

the seed queues and dynamically increases the probability of choosing that input generator if the input yields new behavior signals at the hardware execution level. The other three micro-architecture level optimizations are *loop unrolling* which enables parallel iteration, *shannonization* which precomputes operations and reduces the latency of critical paths, and *data pre-loading* for fast memory access by moving data from global memory to local memory. HFuzz is the first to directly leverage the performance enhancing power of FPGA for automated testing of heterogeneous applications on an FPGA device.

We evaluate HFuzz's effectiveness on seven programs. These programs are from Intel's OneAPI benchmarks for heterogeneous applications with FPGA kernels [24]. We compare HFuzz against four alternatives: (Alternative 1: AFL-like) an AFL-like grey-box fuzzing tool that uses branch coverage as feedback and runs on the host entirely, (Alternative 2: HeteroFuzz) the state-of-the-art testing tool for heterogeneous applications using software monitors only, (Alternative 3: NoKernelMutation) HFuzz with CPU-side input mutation without offloading it to FPGA, and (Alternative 4: NoHWOptimization) HFuzz without FPGA-level optimizations. It took HFuzz much less time (i.e., 7%, 9.7%, 21.3%, and 29.4% of the time used by the four alternatives) to find the same number of defects. Given the same time budget (4 hours), HFuzz found 11×, 4.13×, 2.36×, and 1.03× more defects than the four alternatives. We tried longer time (24 hours) but no more defect is found after 4 hours. Per the open science policy, we make HFuzz's artifacts, benchmark programs, and datasets available at *https://github.com/UCLA-SEAL/HFuzz*.

In summary, this work makes the following contributions:

- To our knowledge, HFuzz is the first fuzz testing technique that uses hardware probes in tandem with software monitors to guide test input generation for heterogeneous applications.
- HFuzz is the first to unlock new micro-architecture level performance optimizations for fuzz testing by mapping both iterative input mutation and kernel invocation to FPGA-side computation. It implements four FPGA-level optimizations and accelerates fuzzing by 3.4×.
- HFuzz accelerates fuzz testing by 4.7× by directly synthesizing input mutations within kernels on FPGA. This also reduces the host-device data transfer overhead by 66%.
- With a 4-hour budget on seven benchmarks, HFuzz was able to discover 33 defects, while traditional coverage-guided fuzzing only uncovered 3 defects. Out of these 33 defects, 25 could not have been found without the use of device-side feedback.

2 BACKGROUND

2.1 Heterogeneous Applications with FPGA

Driven by performance and energy benefits, heterogeneous computing applications [7] contain code that is executed on different kinds of processors such as CPU, GPU, and FPGA.

FPGAs are field programmable gate arrays. Modern FPGAs include millions of look-up tables (LUTs), thousands of embedded block memories (BRAMs), thousands of digital signal processing blocks (DSPs), and millions of flip-flop registers (FFs) [52]. Intel provides CPU+FPGA multi-chip packages; with its recent acquisition of Altera, such integration is expected to be even tighter in the future. FPGA has made its way into modern data centers, including Microsoft's Azure, Amazon F1, and Intel DevCloud [2, 26, 54].

```
1  for(int s = 1; s <= nsteps; ++s) {
2    ...
3  // Kernel: calculate velocity
4     h.parallel_for(n, [=](item<1> i){
5       acc0=0; acc1=0; acc2=0;
6       #pragma unroll factor=2
7       for(int j=0; j<n; j++) {
8         if (j==i) {continue};
9         int8 dx, dy, dz;
10        dx = p[j].pos[0]-p[i].pos[0];
11        dy = p[j].pos[1]-p[i].pos[1];
12        dz = p[j].pos[2]-p[i].pos[2];
13        int8 sqr=dx*dx+dy*dy+dz*dz;
14        acc0+=(kG*p[j].mass/sqr)*dx; //calculate acceleration
15        acc1+=(kG*p[j].mass/sqr)*dy;
16        acc2+=(kG*p[j].mass/sqr)*dz;}
17        p[i].vel[0]+=acc0*dt; //calculate velocity
18        p[i].vel[1]+=acc1*dt;
19        p[i].vel[2]+=acc2*dt;});});
```

Figure 2: Nbody-simulation: a heterogeneous version with DPC++ high-level synthesis.

A heterogeneous application typically consists of *host* code executed on the CPU and *kernel* code to be synthesized and executed on FPGA or GPU. Host code initializes the device, allocates the device memory, transfers data to the device, and invokes the compute-intensive kernel on the device side. After the execution, it transfers the kernel output back to the host and deallocates the memory.

To simplify kernel development, high-level-synthesis (HLS) [15, 21] lifts the abstraction of hardware development by automatically generating register-transfer level (RTL) descriptions from code written in C-like dialects. One example of HLS C/C++ dialects is Intel's Data Parallel C++ (DPC++), a cross-platform abstraction layer that enables code to be targeted to different CPUs, GPUs, and FPGAs [44, 45]. With DPC++, users can specify which hardware platform to implement a kernel on. For example, a user may use a compiler flag `-Xsboard=intel_s10sx_pac` to select Intel's FPGA S10. The user can develop a kernel function `f`, calling `h.parallel_for(n,f)` with a job handler `h`. This handler executes `f` with `n` degree parallelism on FPGA S10. Consider the example in section 2.2.

2.2 An Illustrating Example: Nbody-simulation

Figure 2 illustrates the simulation of n particles moving over a sequence of `nsteps`. Lines 10-12 calculate the distance between particles, while Lines 14-16 calculate the acceleration. In lines 17-19, the program subsequently updates the particles' velocities based on the acceleration. These computations are extracted as compute-intensive kernels and offloaded to an FPGA. To enable parallelism and speed up the velocity calculation, the developer uses `h.parallel_for` and loop unrolling `#pragma unroll factor=2` (highlighted in red) at Lines 4 and 6.

When writing a heterogeneous application, a user must conservatively estimate the limit of hardware resources and specify bitwidths for custom types and the size of buffers and pipes because all hardware resources are finite. Due to the need to finite hardware resources, a heterogeneous application often contains defects that cannot be detected statically via static analysis. This is a problem that universally exists with all HLS languages. To illustrate, consider the real defects in the Nbody-simulation.

Divide By Zero in Nbody-Simulation. For code in Figure 2, with the input `p.pos=[(1,2,4),...,(1,2,4)]`, the velocity calculation on an FPGA A10 device produces absurdly large numbers `p.vel=[(-214748364,..),..]`. This is because, when the kernel inputs contain two particles with the same position, a divide-by-zero may happen inside the kernel in Lines 14-16 due to `sqr=0` at Line 13.

Overflow in Nbody-Simulation. When the kernel calculates the acceleration of two particles in Figure 2, an in-kernel overflow could occur if two particles are close to each other (i.e., `sqr`≈`0` at Line 13). This is because when `sqr` is close to zero, `acc` becomes large. When the inputs `p.pos=[(81,0,0),(81,1,0),(81,0,1),...]` are sent to the kernel, it produces a small value `sqr=1`, leading to overflow for the variables `acc1`; finally, the wrong result is sent back to the host.

State-of-the-Art. Grey-box fuzzing [58] generates program inputs based on per-iteration execution feedback. Suppose that a user uses grey-box fuzzing to monitor the value range of the inputs and outputs of kernels on the host-side (CPU) code. For the divide-by-zero bug that could occur in Figure 2, because `sqr` is an in-kernel variable and does not appear in the host code, software-side grey-box fuzzing [58] cannot easily reveal defects that originate from the inside of the kernel.

HFuzz addresses the limitations of existing work by utilizing hardware probes to monitor the intermediate states of kernels. HFuzz identifies the in-kernel local variable `sqr` at Line 13 and inserts hardware probes to track its value range. The input generation process is then optimized by prioritizing inputs that result in new minimum or maximum values of `sqr`. As a result, HFuzz is able to effectively detect overflow when `sqr` reaches the small value `sqr=1` and divide-by-zero defects when `sqr` reaches its minimum value 0.

3 APPROACH

HFuzz aims to find inputs that can trigger both in-kernel errors and host-side errors for heterogeneous applications written in Intel's DPC++ HLS [25]. HFuzz contains three novel components that work in concert: (1) in tandem monitoring of software and hardware feedback by injecting software monitors and in-kernel probes (Section 3.1); (2) offloading input mutations to hardware kernels (Section 3.2), and (3) FPGA-level optimizations to speed up iterative input generation and kernel invocation (Section 3.3). HFuzz's design builds on two key insights. First, hardware-level parallelism can bring notable performance enhancement for iterative fuzzing, which is often characterized by independent task-level parallelism. Second, grey-box fuzzing's effectiveness can be significantly improved by observing feedback signals from both hardware and software.

The Fuzzing Process. The overall workflow of HFuzz is shown in Algorithm 1. HFuzz takes as input a program p written in Intel's DPC++ and produces concrete inputs that trigger defects in p. HFuzz first applies a source-to-source transformation to p to produce an instrumented version $p\prime$, by inserting in-kernel probes and software monitors that can guide fuzz testing. HFuzz selects an input generator G from a set of generator S. It then randomly offloads a random seed input in' from G's seed queue into the kernels. To generate new inputs, HFuzz creates a new mutation kernel job in addition to the original kernel, and utilizes parallelism within FPGAs to

Algorithm 1: Fuzzing workflow.

```
   Input: program p, input generator set S, mutation operator set O
1  FuzzingLoop(p, S)
2  begin
3      p' = INSTRUMENT(p);
4      Feedback = ∅;
5      for 1..MAX do
6          G = S.select_input_generator();
7          in' = RANDOM_SELECT(G);
8          F_HW, F_SW =
             p'.host, in_kernel_mutate_execute(in', O);
9          for F ∈ {F_HW ⋃ F_SW} do
10             if F ∉ Feedback then
11                 INCREASE_PROB(S, G);
12                 good_input = REGENERATE(F.m, in');
13                 G = G ⋃ {good_input};
14                 Feedback = Feedback ⋃ {f};
15             end
16         end
17     end
18 end
   Input: kernel_input k_s, mutation_ops_set O
   Output: F_HW is a queue of triples (f, m, out) where f is
           kernel-feedback, m is mutation, and out is kernel output
19 In_Kernel_Mutate_Execute(k_s, O)
20 begin
21     for i = 1..MAX do
22         operator o = SELECT_OP(O);
23         start s = RANDOM_GENERATE();
24         end e = RANDOM_GENERATE();
25         mutation m = {(o, s, e)};
26         In_queue = In_queue ⋃ MUTATE_INPUT(o, s, e, k_s);
27     end
28     foreach in ∈ In_queue do
29         (f, m, out)=EXECUTEONDEVICE(in) ;
30         F_HW = F_HW ⋃ (f, m, out) ;
31     end
32     return F_HW
33 end
```

mutate the input locally. The target function directly accesses the new input from local memory. In this process of input mutation and target execution, HFuzz incorporated four FPGA level optimizations for performance efficiency. As shown in Algorithm 1 at Lines 10-15, inputs that advance either software or hardware feedback are saved to the input queue as *good_input* for the next fuzzing iteration. If a new input generated by generator G results in new feedback, G will be considered a favored generator and its activation probability will be increased with $INCREASE\_PROB(S, G)$ at Line 11.

3.1 Injecting HW Probes in addition to SW Monitors

HFuzz, *for the first time*, directly introduces application-specific observability to hardware kernels by inserting hardware probes. It leverages these kernel probes in tandem with software-level monitors to form effective feedback signals to stretch heterogeneous application behavior.

Table 1: Mutations accelerated by hardware.

Category	Description	SW Mutations	In-kernel Mutations	Average Speedup
M1 Sparsity Mutation	Replace non-zeros with zeros from index *s* to *e*, or do the opposite	`for i in s..e do {vector[i]=0}`	`# pragma unroll for i in s..e{vector[i]=0});`	4.31×
M2 Copy Mutation	Replace each element from index *s* to *e* with element at *s*	`for i in s..e do {vector[i]=vector[s]}`	`# pragma unroll for i in s..e{vector[i]=vector[s]});`	3.98×
M3 Addition Mutation	Add constant *a* to each element from index *s* to *e*	`for i in s..e do {vector[i]+=a}`	`# pragma unroll for i in s..e{vector[i]+=a});`	3.21×
M4 Bit Mutation	Mutate an element with binary XOR given a constant *x*	`for i in s..e do {vector[i]^= (1«x)}`	`# pragma unroll for i in s..e {vector[i]^= (1«x)});`	4.42×

```
1  //First kernel...
2    h.parallel_for(range(M, P), [=](auto index) {
3      int sum = 0;
4      #pragma unroll factor=2
5      for (int i = 0; i < num_element; i++) {
6        sum += a[index[0]][i] * b[i][index[1]];
7        if (min_sum>sum) min_sum=sum;
8        if (max_sum<sum) max_sum=sum;}
9      bool flag;
10     KToKPipe::write(sum, flag);
11     KToKPipeSize++;//Pipe usage Probe
12     DeviceToHostKToKPipe::write(KToKPipeSize);
13     DeviceToHostMax_sum::write(max_sum);//sum's Value Range Probe
14     DeviceToHostMin_sum::write(min_sum);});});
15 //Second kernel...
16   h.single_task([=]() {
17   for (size_t i = 0; i < number_element; ++i) {
18     out[i] = KToKPipe::read();
19     KToKPipeSize--;//Pipe usage Probe
20     DeviceToHostKToKPipe::write(KToKPipeSize);
21     out[i] = reciprocalTransform(output[i]); }});});
22 for(int i=0; i<number_element; i++) {//SW monitor for kernel output
23   outmin=min(outmin, output[i]);
24   outmax=max(outmax, output[i]);}
```

Figure 3: Matrix transform: inserted Value Range Probes are in the green rectangle. InsertedPipe Usage Probes are in the red rectangles. Inserted SW Monitors are in the orange rectangle.

Hardware Probes. While OS virtualization could provide the appearance of unbounded resources for the code executed on traditional CPUs, kernel functions are physically mapped to *resource-limited* heterogeneous architectures. This distinction leads to unique failures that are often induced by *resource limitations* on the device-side, which are not easily detectable when running software simulators. For example in Figure 2, a local variable `sqr` customizes regular integers to 8-bit integers for resource efficiency. Overflow conditions can occur if the variable's value exceeds its customized bitwidth. As another example, pipe saturation between two consecutive kernel functions can lead to read and write failures. In fact, such incorrect *intermediate computation states* within hardware kernels have been identified as the primary reason for hardware-originated bugs. HFuzz takes advantage of this observation, identifies local variables within kernels that hold intermediate states, and injects hardware probes to expose potential failures in kernel.

HFuzz automates the process of hardware probe insertion through source to source transformation, creating an instrumented kernel. From such instrumented kernel, intermediate states in the HW device are sent directly to the host code using dedicated host-kernel communication channels. The channels are implemented as global FIFO buffers and can be accessed from both the host and the kernel. The kernel side writes hardware feedback into the channels, while the host side reads information from the channels. Both read and write operations are non-blocking, in order to minimize any additional overhead to the original kernel logic. To expose intermediate computation states, HFuzz identifies in-kernel local variables and pipe usage via a C/C++ AST analysis [4]. As shown in Figure 3, in-kernel variable `sum` is highlighted in green, and pipe usage is highlighted in red. With a focus on in-kernel local variable and pipe monitoring, HFuzz aims to uncover the two most commonly seen errors in custom hardware accelerators: overflows resulting from the resource and bitwidth finitization, as well as read/write failures caused by communication pipe saturations.

- *Value Range Probe*: HFuzz creates a value range monitor that checks the maximum and minimum value for each in-kernel variable. In Figure 3, HFuzz inserts probes on the intermediate variable `sum` which saves the cumulative sum of the product `a[index[0]][i]*b[i][index[1]]`. These probes monitor the minimum and maximum value of `sum`. HFuzz also constructs channels `DeviceToHostMax_sum` and `DeviceToHostMin_sum` to send these captured values back to the host at Line 13-14.
- *Pipe Usage Probe*: HFuzz creates a pipe usage monitor for each communication pipe. Consider the same example in Figure 3. HFuzz uses an AST analysis tool [4] to identify the locations of two kernel functions: `matrix_multiply` at Line 1-14 and `transformer` Line 16-21. We identify the variable name, `KToKPipe` used for pipe-based data transfer between the two kernels. By using `KToKPipe::write()` and `KToKPipe::read()`, the first kernel writes its result `sum` at Line 10 and the second kernel reads the value from this pipe at Line 18 in Figure 3. HFuzz applies source to source transformation to inject a counter-based usage monitor for this pipe and update the counter `KToKPipeSize` at Line 11 and Line 19 in Figure 3. Then HFuzz sends this counter value to the host by creating another direct communication channel, called `DeviceToHostKToKPipe` at Line 12 and Line 20.

Software Monitors. In addition to in-kernel probes, HFuzz inserts a set of software monitors on the host side, specialized to the custom FPGA accelerator synthesized on the device. We monitor: (1) the number of loop iterations, because it is related to pipelining and loop unrolling, common optimizations for parallelization implementation on FPGA; (2) the value range of each kernel input and output; (3) the kernel execution time, as hang or unexpectedly slow execution could be an indicator of failures. HFuzz retrieves the time and loop unrolling information from the HLS compilation

report generated by DPC++. Besides, to monitor the value range of each kernel input and output, HFuzz inserts a value range monitor before and after each kernel, as shown in Lines 22-24 of Figure 3.

3.2 Offloading Input Mutations to Kernels

The traditional fuzzing process involves repeatedly mutating seed inputs and feeding them into a target program. The implicit assumption underlying such mutations is that seed inputs can be mutated and sent to the target program fast. Unfortunately, this assumption does not hold true for heterogeneous applications. Inputs to heterogeneous applications are often large matrices, leading to significant data transfer overheads between CPU and FPGA. We observe that *local* data transfer—data transfer within FPGAs, consumes less than 89% of the time required for data transfer between the fuzzer and the kernel. Additionally, in the process of fuzzing, a variety of *independent* mutation operations are frequently employed on small segments of the same seeds with the aim of exploring the input space. Thus, we can avoid repetitive data transfer by offloading the seed inputs to hardware kernels and mutating them directly within FPGAs. To achieve this, HFuzz creates a dedicated kernel for mutations *in parallel to* the original kernel, as well as a segment of on-chip memory for the storage of seeds and newly generated inputs. The mutation kernel and the original kernel function are both synthesized to the FPGA hardware concurrently. Table 1 shows four supported mutation operators. Because mutation operators are all order-independent and deterministic, HFuzz modifies all elements in the seed input at once. A resulting input can be re-generated given the seed and a concrete instance of mutation.

Consider Figure 3 as an example. The first kernel code computes the matrix product with two input matrices. We show how HFuzz tracks the feedback and mutates the input step by step in Table 2. With the initial seed input offloaded to the kernel, HFuzz tracks hardware feedback from the in-kernel variable `sum` at Line 2 by the inserted in-kernel probes in the green rectangle (column *Hardware Probes* in Table 2). After we apply the *M3 Addition Mutation* with loop unrolling optimization, from the starting offset `s=1` to the ending offset `e=4` on array `a`, a greybox fuzzer that only monitors the value range for the kernel interface variables `a` and `b` would discard the input `[-20,5,7,7,9,20]` because it does not achieve a new value spectra at the software level. However, HFuzz saves the corresponding mutation information, since this input registers a new feedback at the hardware level for the in-kernel variable `sum`.

3.3 FPGA Optimizations for Fuzzing

Traditional fuzz testing can be naïvely applied to heterogeneous applications by treating hardware kernel invocations as equivalent to software function calls. However, such straightforward application of software-style fuzzing results in severe performance inefficiencies. In heterogeneous applications, there is a distinct *opportunity* to utilize hardware micro-architecture level optimizations to accelerate the traditional fuzzing process. Both iterative matrix mutations and target executions involve independent tasks, enabling task-level parallelism.

HFuzz applies four FPGA optimizations to accelerate iterative matrix mutations and target execution, including loop unrolling, shannonization, local memory access, and dynamic kernel sharing.

```
1  for (int i = s; i < e; i++) {
2    if (A[i]==0) {A[i] = generate_number(seed);}}
```

(a) Original mutation

```
1  int local_A[e-s];
2  #pragma unroll factor=4
3  for (int i = 0; i < e-s; i++) {local_A[i] = A[i+s];}
4  int t = generate_number(seed);
5  for (int i = 0; i < e-s; i++) {
6    if (local_A[i]==0) {
7    local_A[i] = t;
8    t = generate_number(seed);}}
9  #pragma unroll factor=4
10 for (int i = 0; i < e-s; i++) {A[i+s] = local_A[i];}
```

(b) Optimized mutation in kernel

Figure 4: Sparsity mutation: replace the zero elements to non-zero elements from index `s` to index `e`.

These optimizations are not specific to HFuzz or Intel's heterogeneous architecture, and thus also are applicable to other applications on other FPGAs. For instance, loop unrolling is a technique that can be used to optimize iterative computations that do not have significant data dependencies between iterations, and it can be applied independently of the specific FPGA platform.

1. Dynamic Kernel Sharing. In traditional fuzzing, the difficulty of testing often arises from the need to explore deep branches within the program. However, when testing heterogeneous applications, errors tend to occur due to variations in the range of values for in-kernel variables and resource usage. This presents a significant challenge of rapid input space exploration especially when inputs are large matrices.

We propose a dynamic, probabilistic kernel-sharing method to interleave the exploration of input search space originating from multiple seeds in heterogeneous applications. To implement this method, HFuzz employs four input generators that share the same target kernel and each has its own seed queue. These input generators start with different seed inputs and, during each iteration, one generator is chosen based on an activation probability array. The selected generator then picks a seed input from its queue, mutates it within the kernel, and sends the generated input to the target kernel function via on-chip memory on the device. If the generated input results in new feedback, it is saved in the generator's seed queue for use in future fuzzing iterations.

HFuzz utilizes an adaptive approach to input generation by selecting an input generator and its associated seed queue based on an activation probability array. The selection process involves evaluating the performance of each generator and adjusting its probabilities accordingly. For instance, if a new input generated by generator G results in new feedback, it will be considered a favored generator and its activation probability will be increased. Otherwise, it will be labeled as an inactive generator and its activation probability will be decreased. This approach allows for efficient input space exploration and ensures that the test generation is

Table 2: Example execution of input generator G.

ID	Mutation Operator	Kernel Inputs	Variable	Hardware Probes Min	Hardware Probes Max	Software Monitors Min	Software Monitors Max	New Value Range	Overflow	Save Input	Memorization HW Range	Memorization SW Range	P_G
Seed	N/A		sum	-56	168			N/A	No	N/A	[-56,168]		0.25
		a[][1]=[-20, 2, 4, 4, 6, 20]	a			-20	20	N/A				[-20, 20]	
		b[1][]=[1, -10, -4, -14, 28, 0]	b			-14	28	N/A				[-14, 28]	
1	**M3**		sum	-202	54			**Yes**	No	**Yes**	**[-202, 168]**		0.3
	start s=1	a[][1]=[-20, 5, 7, 7, 9, 20]	a			-20	20	No				[-20, 20]	
	end e=4	b[1][]=[1, -10, -4, -14, 28, 0]	b			-14	28	No				[-14, 28]	
2	**M2**		sum	-70	140			No	No	No	[-202, 168]		0.25
	start s=1	a[][1]=[-20, 5, 5, 5, 5, 20]	a			-20	20	No				[-20, 20]	
	end e=4	b[1][]=[1, -10, -4, -14, 28, 0]	b			-14	28	No				[-14, 28]	
3	**M3**		sum	20	-140			**No**	**Yes**	**Yes**	**[-202, 168]**		0.3
	start s=1	a[][1]=[-20, 8, 10, 10, 12, 20]	a			-20	20	No				[-20, 20]	
	end e=4	b[1][]=[1, -10, -4, -14, 28, 0]	b			-11	28	No				[-14, 28]	

focused on areas that are likely to yield new feedback:

$$P_G = \begin{cases} P_G + \alpha & \text{if } G \text{ is chosen and HFuzz gets new feedback} \\ P_G - \frac{\alpha}{l-1} & \text{if } G \text{ is not chosen and HFuzz gets new feedback} \\ P_G - \alpha & \text{if } G \text{ is chosen and HFuzz gets no new feedback} \\ P_G + \frac{\alpha}{l-1} & \text{if } G \text{ is not chosen and HFuzz gets no new feedback} \end{cases} \quad (1)$$

In our experiment, we set the number of generators l to be 4. The initial activation probability for each generator P_G is set to $1/l = 0.25$. The update factor α is predefined as 0.05. In Table 2, in the second execution (ID 2), inputs generated by generator G increased the hardware monitor range. As a result, HFuzz increases the activation probability of G from 0.25 to $0.25+\alpha = 0.3$.

2. Data Preloading [28]. Matrix mutation on large matrices requires a significant amount of data read and write operations. To improve efficiency, it is crucial to minimize memory access time for input vectors or matrices. Many heterogeneous computing systems, such as Intel oneAPI, have both *global memory* that can be accessed by both kernel and host code, and on-chip *local memory* that is only accessible by kernel code. Accessing local memory within the kernel typically has a shorter latency than accessing global memory. We thus apply data preloading to transfer data from global memory to local memory.

In Figure 4b, HFuzz reduces memory access costs (highlighted in red) by transferring data from array `A` to the local array `local_A`. This results in a reduction of memory access cost as seen at Lines 6-7 in the optimized code, compared to the original code in Figure 4a at Line 2. This optimization leads to a 1.31x speedup in the mutation process.

3. Shannonization [27]. Sparsity mutation replaces zero elements with non-zero elements. It necessitates the implementation of a null check for each element in the matrix. As shown in Line 2 of Figure 4a, an `if` statement is added to accomplish this. However, this `if` statement induces extra hardware overhead, as it increases the delay in the critical path. Each time the `if` condition is satisfied (i.e. `A[i]==0`), the operation `generate_number` needs to be computed, which can slow down the overall performance.

Shannonization improves performance by precomputing operations within a loop and removing them from the critical path. In this example, HFuzz applies shannonization (highlighted in green in Figure 4b) by precomputing the operation `generate_number` at Line 4, and removing it from the critical path inside the branch at Line 6. Then HFuzz precomputes the next value of `t = generate_number` at Line 8 for a later iteration of the loop to use when required (that is, the next time `local_A[i]==0`). This precomputation can be done simultaneously within the loop, allowing for a reduction in the critical path delay and leading to a 1.24x speedup in the sparsity mutation process.

4. Loop Unrolling [29]. Software-style mutations on large vectors and matrices are often performed by modifying one or some particular elements. Line 2 in Figure 4a shows an example mutation based on a `for` loop. Such direct application of loops on hardware neglects the potential for hardware parallelism, resulting in inefficient use of hardware resources.

Loop unrolling improves performance by creating multiple copies of the loop body, thus the required number of iterations is reduced. In the example shown in Figure 4b, the `#pragma unroll` directive (highlighted in orange) causes the kernel to unroll the loop by a factor of 4, as specified by the `factor=4` argument. The compiler then expands the pipeline by quadrupling the number of operations and loading three times more data. This results in a 4x speedup of the loop process.

4 EVALUATION

We evaluate the following research questions:

RQ1 How much improvement in defect detection capability is achieved by incorporating both device-side feedback and host-side feedback in HFuzz?

RQ2 How much speed-up is achieved by in-kernel input mutations?

RQ3 How much speed-up is achieved by FPGA-level optimizations for fuzzing?

RQ4 How much overhead is incurred by injecting hardware probes in HFuzz?

To assess the improvement in defect detection and fuzzing acceleration, we compare HFuzz against four baselines.

(1) Alternative 1 AFL-like: This option uses branch-coverage guided fuzzing similar to AFL and performs input mutations on CPU side.

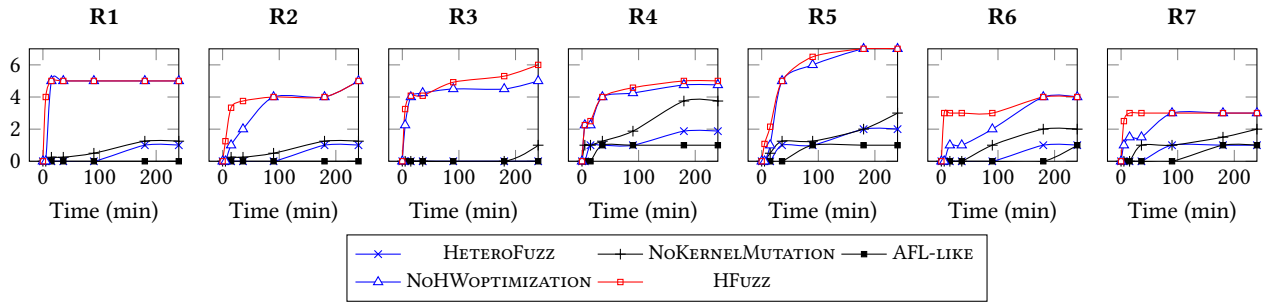

Figure 5: # Number of Defects

(2) Alternative 2 HeteroFuzz: This option is a replication of the state-of-art work HeteroFuzz [58] for Intel DPC++. Compared to HFuzz, it does not have in-kernel probes on FPGA devices and considers only software monitoring feedback.
(3) Alternative 3 NoKernelMutation: This option disables in-kernel mutations and performs input mutations on the CPU.
(4) Alternative 4 NoHWOptimization: This option disables hardware optimizations and only uses one input queue instead.

Benchmarks. We choose seven applications from Intel's OneAPI GitHub repositories [24]: (R1) Matrix-transform. It has two kernels—one for matrix multiplication M=A*B and the other for reciprocal transformation on each element of M; (R2) Matrix-mul: multiplication of two matrices; (R3) Complex-mul: multiplication of two vectors of complex numbers in parallel; (R4) APSP: the Floyd-Warshall algorithm to find the shortest path between the pairs of vertices in a graph; (R5) Nbody-sim: Simulation of a dynamical system of particles under the influence of gravity; (R6) Hidden-Markov-model: a statistical model using a Markov process; (R7) Match-num: reading data from the host and sending the numbers that match a set of pre-defined constants back to the host.

These benchmarks are widely used in hardware acceleration literature [46] and cover a representative set of optimizations used in kernels (e.g., custom bitwidth, loop unrolling, etc.) and exhibit different memory usage patterns (e.g., buffer memory and unified shared memory for kernel input and output, kernel-to-kernel pipe and kernel-to-host pipe, local memory for in-kernel variables, etc.). Testing difficulties for heterogeneous applications do not depend on the code size; rather, it depends on how hardware resources are synthesized (e.g., in-kernel variables, loop unrolling) and the communication channel details between software and hardware and between hardware kernels. These benchmarks' kernels are widely used and their code size is similar to commercial HLS benchmarks. They are complex in both optimizations and memory arrangements and hard to get right.

Experimental Environment. All experiments were conducted on Intel DevCloud A10 nodes [26]. The automated kernel probe insertion was implemented using DPC++ compiler and Pycparser [4]. The refactored programs were synthesized to RTL and targeted to Intel Arria 10 GX FPGA [30]. We also tried HFuzz on other FPGAs like Intel Stratix 10 SoC FPGA [31] and achieved similar results.

Table 3: Example symptoms of kernel defects in R1.

ID	Symptom	Description	HeteroFuzz Find
S1	Kernel Runtime Overflow	The value of intermediate variables `sum` at line 2 of Figure 3 exceeds its bitwidth capacity, leading to a wrong result.	✓
S2	Pipe Write Failure	Pipe write failure happens when FPGA attempts to write into a pipe when the pipe is full.	×
S3	Pipe Read Hang	Pipe read hang happens when FPGA attempts to read synchronously from an empty pipe.	×
S4	Division by Zero	`sum` in line 5 of Figure 3 equals 0, leading to divide by zero at line 21.	×
S5	Incorrect Loop Unrolling	CPU and FPGA produce different results when the input array size `num_element` is not multiple of 2.	✓

4.1 Defect Detection by HW and SW Feedback

We assess the effectiveness of HFuzz's feedback guidance by comparing the number of defects detected through combined hardware probes and software monitors to that of HeteroFuzz, which relies solely on software monitors. For each benchmark, we generate test inputs using HFuzz and HeteroFuzz for 4 hours. We tried longer time (24 hours) but no more defect is found after 4 hours. Using the generated inputs, we then perform differential testing between CPU-only executions and CPU+FPGA executions and measure the number of defects (i.e., diverging outcomes) found.

Figure 5 shows the average experimental results from ten runs. HFuzz is able to detect 3.1× more defects than HeteroFuzz. For example, for R5 Nbody-simulation, without monitoring in-kernel variable `sqr`, HeteroFuzz cannot find the divide-by-zero error we mentioned in Section 2.2 at Lines 16-18 in Figure 2. When using HeteroFuzz, the value range of kernel inputs does not reflect the change in the square of distance between particles `sqr`. HFuzz, instead, directly monitors the value range of in-kernel variable `sqr`, and finds the defects when `sqr` reaches its minimum value 0. In total, HeteroFuzz finds 8 unique defects in 16.5 hours, while HFuzz finds the same defects in 1.6 hours—almost 90% reduction in the testing time.

Table 3 lists five defects found by HFuzz in R1 `Matrix-transform`.

First, S1 shows an overflow occurred in the FPGA execution due to the in-kernel variable `sum` at Line 3 in Figure 3. It happens when

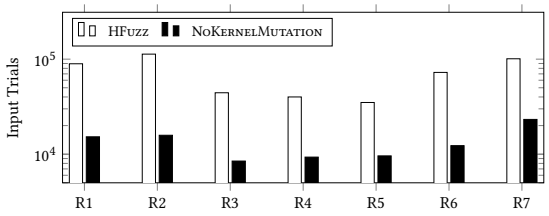

Figure 6: Number of Input Trials

the input vector a includes a large number such as 2090401586. By monitoring in-kernel variable sum's value range, HFuzz increases the chance of generating a new vector with large numbers.

Second, two kernels in R1 use a 128-byte pipe to facilitate direct data transfer. As mentioned in Section 1, when the first kernel produces results faster than the second kernel can consume, the pipe may become saturated. Consequently, a pipe write failure occurs silently and the newly written value is lost, shown as S2 in Table 3. This may further lead to another defect S3: pipe read hang. The second kernel in Figure 3 reads values from the pipe for `number_elements` times. However, if the number of values successfully written to the pipe is less than `number_elements`, the second kernel will hang at this pipe read. Both defects cannot be detected by prior work HeteroFuzz because host-side software monitors cannot detect the saturation of commutation pipes.

Third, S4 depicts a divide-by-zero error caused by the intermediate result sum in the second kernel `reciprocalTransform` at Line 21 in Figure 3. It happens when both two input matrices are sparse matrices. On CPU, this execution may raise a division-by-zero exception; however, it silently returns an unexpected number on FPGA instead. By monitoring sum's value range, HFuzz triggers this defect by generating inputs using *Sparsity Mutation*.

Fourth, since R1 makes two copies of the loop body at Line 4 in Figure 3 by using `#pragma unroll factor=2`, a wrong result happens if the number of loop iterations `num_elements` is not a multiple of the unroll factor 2.

> HFuzz achieves 10.3× speed-up and finds 25 new defects compared to HeteroFuzz, demonstrating the combined benefit of hardware probes and software monitors.

4.2 Speed-up from In-kernel Input Mutations

To assess speed-up enabled by offloading input mutations to FPGA devices, we compare HFuzz with a downgraded version NoKernelMutation. We measure the number of generated inputs and defects found within the same 4-hour budget.

Figure 6 reports the average number of input trials within 4 hours. For example, in R7, NoKernelMutation generates 23225 inputs, while HFuzz generates 100918 inputs (5.3× speed-up) by avoiding redundant data transfer and parallelizing input mutations. In R2, NoKernelMutation and HFuzz enumerate 15824 and 112940 inputs respectively, leading to 7.1× speed-up. R2 achieves higher speedup than R7 because its performance is more dominated by data transfer as shown in Figure 1.

Figure 5 shows the number of defects found by NoKernelMutation. While NoKernelMutation reports 14 unique defects in 24 hours, HFuzz detects the same defects in 5.1 hours, which translates to 4.7× speed-up in defect detection. These defects are not found by NoKernelMutation, because it wastes time in sequentially mutating inputs in CPU and sending the large data to the kernel.

> HFuzz reduces the need for data transfer by offloading mutations into kernels and thus speeds up fuzzing by 4.7×.

4.3 Speed-up from FPGA-level Optimizations

To evaluate the effectiveness of FPGA-level optimizations for input generation, we created a downgraded version of our tool NoHWoptimization, which disables this feature. We evaluated the time taken to find the same defects. The results are shown in Figure 5. Compared to NoHWoptimization, HFuzz finds the same 33 bugs 3.4x faster, taking only 8.3 hours as opposed to 28 hours.

In R1 (e.g., Figure 3), the detected defects include (1) a divide-by-zero error when the kernel takes as input two sparse matrices and (2) an overflow error when the kernel takes as input two dense matrices with large elements. Because inputs leading to these defects are distinct from each other, traditional mutational fuzzers with a single input queue may be inefficient to find them. In fact, it takes 2 hours to mutate two sparse matrices into dense ones. HFuzz uses one hardware optimization technique, called dynamic kernel sharing, to enable simultaneous exploration of input subspaces originating from different seeds. For that, HFuzz utilizes multiple input generators. One generator *A* starts with dense matrices and another generator *B* starts with sparse matrices. HFuzz can detect these two bugs by interleaving generator *A* and generator *B* based on runtime feedback. For example, when generator *A* reaches its maximum value and triggers an overflow, it can no longer provide any new feedback. HFuzz will switch to generator *B* and detect the divided-by-zero error. HFuzz reduces the detection time to 5 mins.

> HFuzz achieves 3.4× speed-up in the detection of detects by implementing hardware optimizations. Loop unrolling, shannaization, and fast memory access directly speed up the mutation process. Dynamic kernel sharing enables efficient input space exploration.

4.4 Probe Overhead

Inserting hardware probes into the original kernels may cause extra overhead on hardware resources, as reported in Table 4. We measure four types of hardware resource, including ALUT (a lookup table implementing the boolean function), FF (flip flops for storing temporary data), RAM (random access memory blocks), and DSP (a digital signal processing unit for common fixed-point and floating-point arithmetic). The inserted kernel probes incur a relatively large overhead for a simple kernel because the inserted probes significantly increase kernel logic complexity compared to the original kernel. In R2, compared to the original kernel with 9592 ALUTs and 14466 FFs, inserted probes used 22% more ALUTs and 33% more FFs. For a relatively complex kernel R4, the overhead is 6% ALUT and 10% FFs. The extra resource usage mainly comes from (1) the probe computation including read and write, and (2) the kernel dispatch logic establishes the communication between kernel and host.

Table 4: Resource overhead from injecting hardware probes.

ID/Program		#LUT	#FF	#RAM	#DSP	Freq /MHz
R1/	Orig	15932	25088	137	4.5	247
Matrix_trans	Probe	17905	34320	192	4.5	246
R2/	Orig	9592	14466	492	16	259
Matrix_mul	Probe	12032	19443	492	16	247
R3/	Orig	11545	18494	106	6	273
Complex_mul	Probe	11203	27117	106	6	253
R4/	Orig	60468	92249	555	195	221
APSP	Probe	64327	101229	558	195	212
R5/	Orig	23642	44352	309	34	270
Nbody_sim	Probe	27612	50549	317	34	260
R6/	Orig	48706	64987	395	67	257
HMM	Probe	56562	87392	491	67	247
R7/	Orig	2239	1357	67	12	279
Match_num	Probe	3828	2033	73	12	259

Such overhead could be further reduced by manual optimizations. For example, Curreri [17] performs resource sharing by using the same FIFO probe for multiple feedback signals.

> Hardware probe insertion uses 24% extra LUT, 29% extra FF, and 8% extra RAM, and reduces frequency by 5% on average. However, it enables an overall 10.3× speed-up in defect detection by providing hardware feedback.

5 THREATS TO VALIDITY

We discuss the threats to validity as follows.

Device Dependence. Our experiments run all kernel executions on two prominent FPGA cards: S10 and A10 [30, 31], which are among the most widely used FPGAs currently. This specific configuration may constrain the applicability of our results to other devices, such as Intel's Altera, because the divergence symptoms detected could differ across different platforms. While the absolute values of execution time and symptoms depend on configurations, we believe that HFuzz will preserve its overall advantages in terms of acceleration and divergence-detection capability when extended to various platforms.

Time Limit. We empirically set four hours as the time limit for fuzzing. Longer execution time may expose more divergence errors or more execution paths as suggested in [32]; however, this time limit is reasonable, as we did not see any increase in new types of divergence errors with a higher time limit for subjects R1-R7.

Scalability. The insertion of our probes relies on the static analysis of heterogeneous programs and often necessitates human intervention to address potential transformation errors. This process can become challenging, particularly for complex in-kernel logic within large programs. Further experimentation is essential to validate the scalability of our method. However, our benchmarks may look small in size from the software engineering perspective, but they are sizable in the hardware community. Rosseta benchmarks [59] and heterogeneous applications in Intel Devcloud are comparable in size (i.e., hundreds of lines of code.) Testing complexity for heterogeneous applications do not depend on the lines of code size. Instead, they depend on factors such as how hardware resources are synthesized (e.g., in-kernel variables, loop unrolling), as well as the nuanced details of the communication channels between software and hardware, as well as among hardware kernels.

6 RELATED WORK

Fuzz Testing. Traditional fuzzing starts from a seed input, runs the program on the selected input, generates new inputs by mutating the previous input, and adds new inputs to the queue if they improve a given guidance metric such as branch coverage. Instead of using coverage as guidance, several techniques use custom guidance mechanisms. UAFL [50] incorporates typestate properties and information flow analysis to detect the use-after-free vulnerabilities. BigFuzz [57] monitors dataflow operator coverage in tandem with branch coverage for dataflow-based analytics. For example, MemLock [51] employs both coverage and memory consumption metrics. AFLgo [5] extends AFL to direct fuzzing towards user-specified target sites. SiliFuzz [48] finds CPU defects by fuzzing software proxies, like CPU simulators or disassemblers, and then executing the accumulated test inputs (known as the corpus) on actual CPUs on a large scale. PerfFuzz [36] uses the execution counts of exercised instructions together with branch coverage to identify inputs revealing pathological performance. HeteroFuzz [58] generates concrete test inputs for heterogeneous applications to perform differential testing between CPU vs. CPU+FPGA. Unlike HFuzz, HeteroFuzz treats the kernels as black boxes and performs software-level monitoring only. All these techniques rely on pure software-level feedback either at the level of code coverage or using custom monitors. None leverages hardware probes in tandem with software monitors to guide test input generation, like HFuzz.

A fuzzing loop consists of multiple invocations of a target program with different inputs in an independent manner; thus, it provides a natural opportunity for parallelism. AFL++ [20] injects a fork server, which tells the target to fork itself to run, and thus realizes parallel fuzzing across multiple CPU cores or across a fleet of systems. For example, P-Fuzz [49] distributes unique seeds to run fuzzing in parallel, and PAFL [38] maintains global and local guiding information for synchronizing parallel fuzzing jobs. These techniques accelerate fuzz testing via distributed computation on CPU, unlike HFuzz, none accelerates fuzzing by using FPGAs. HFuzz pushes iterative input mutation directly to an FPGA kernel, and benefits from the massive hardware parallelism intrinsic to FPGA during iterative testing of heterogeneous applications.

Coverage-guided greybox fuzzing adds test cases into the set of seeds if they exercise the new path or new behavior. However, most seeds exercise the same "high-frequency" paths. To explore more paths with the same number of tests, researchers develop strategies to select seeds wisely. AFLFast [6] models coverage-based greybox fuzzing as a Markov chain, and assigns different selection probabilities for different seeds. EcoFuzz [53] improves AFLFast's Markov chain model and presents a variant of the Adversarial Multi-Armed Bandit model. EcoFuzz sets three states of the seeds set and develops a unique adaptive scheduling algorithm. While these techniques select seeds based on probabilities, none of them leverages FPGA-level optimizations to speed up seed selection with dynamic kernel sharing.

High Level Synthesis & In-Circuit Debugging. To ease the development of heterogeneous applications, HLS tools automatically generate RTL descriptions from C/C++ programs. To help debugging HLS-generated circuits, *Inspect* [8] introduces software debugger-like capabilities, including gdb-like breakpoints, step, and data inspection. It tracks file names and line numbers in HLS code, so that HW probes at the level of wires and registers could be linked to specific lines in the HLS code. A user can monitor each variable for its data width and the number of elements in an array. Monson and Hutchings [41] design a debugger for HLS-generated FPGA-based circuits via source instrumentation by connecting C expressions to top-level ports that serve as debug signals. HLScope [12] is a performance debugger that traces the cause of stalls for HLS-generated circuits. Curreri et al. realize in-circuit assertions for timing analysis and stall-relate bugs [17]. While these debuggers and HFuzz leverage a similar mechanism of injecting HW probes, HFuzz's goal is different—it improves the effectiveness of grey-box fuzzing for heterogeneous applications by designing meaningful monitors at both software and hardware levels.

In the hardware design community, *circuit verification*, including formal verification and runtime verification, has been used to validate code written in hardware description languages (Verilog, VHDL, etc.). For example, RFUZZ [34] is a circuit-level input generator for FIRRTL IR (UC Berkeley's RTL variant). RFUZZ invents a notion of *MUX toggle coverage* for circuit testing at the gate level and employs a rapid memory resetting on FPGA for RTL circuit verification. However, their monitors are gate-level and not application-specific. Qin and Mishra present a scalable test generation technique [43] for hardware kernels in Verilog by interleaving concrete and symbolic execution to bridge the gap between model checking and testing. Kourfali and Stroobandt [33] exploit parameterization of LUTs and routing infrastructures in an FPGA to create a virtual debugging overlay network inside circuits. These circuit testing and verification techniques find bugs in kernels at RTL level, while HFuzz targets *end-to-end testing of heterogeneous applications written in HLS*. In other words, it is not feasible to directly compare HFuzz against these in-circuit verification techniques.

FPGA Performance Optimizations. Ma et al. explored various loop optimization techniques, such as loop tiling, loop interchange, and loop unrolling to reduce memory consumption and data movement when mapping deep convolutional neural networks [39] to FPGA. Zhang et al. adopt data buffering techniques to hide the memory access latency and interconnects, avoiding data transfer overhead from the global memory to FPGAs on-chip memory [56]. Li et al. [37] use pipeline optimizations when mapping layer-by-layer computation to multiple FPGAs resources. Pipelining can increase hardware utilization and achieve high throughput by preventing the computing engines to become idle due to imbalanced computation speed across layers. Other widely used kernel optimizations include I/O optimization by sharing resources among computation tasks at different time stamps. Another optimization is *retiming*, which moves edge-triggered registers across combinatorial gates or LUTs to improve timing while ensuring identical behavior, etc [22]. Inspired by these FPGA-level performance optimizations, HFuzz designs four unique FPGA-level optimizations to accelerate the combined computation of input generation and kernel invocation: dynamic kernel sharing, shannonization, loop unrolling, and data buffering. HFuzz is a pioneering tool—the first to embody FPGA-level optimizations to enhance fuzzing efficiency and effectiveness for heterogeneous applications.

SNAP [19] leverages the existing CPU pipeline and hardware features to optimize the bitmap update required for coverage-guided testing. As opposed to SNAP that targets fuzzing traditional programs running on a CPU and simply uses existing hardware features as a black box acceleration aid, HFuzzHFuzz designs new FPGA-level optimizations for mapping input generation and kernel invocation to FPGAs and empirically demonstrates significant fuzzing speed-up from these optimizations (3.4×).

7 DATA AVAILABILITY

Per the open science policy, we make HFuzz's artifacts, benchmark programs, and datasets available at *https://github.com/UCLA-SEAL/HFuzz*.

8 CONCLUSION

In recent years, performance improvement in CPU has slowed significantly to only a few percent—due to challenges in power supply scaling, heat dissipation, space and cost. This trend necessitates the needs to embrace heterogeneous computer architectures such as GPU and FPGA. In particular, FPGA is a promising, *reprogrammable* alternative for improving performance and energy efficiency. However, due to the lack of observability into FPGA execution and complex interaction between CPU and kernel execution on FPGA, developing and testing heterogeneous applications is extremely inaccessible to regular software engineers.

HFuzz is the first grey-box testing approach leverages the *capability of heterogeneous hardware* for testing *heterogeneous applications*. In particular, HFuzz injects hardware probes in addition to injecting software monitors to better guide input generation and offloads iterative input generation to hardware accelerators. HFuzz speeds up fuzzing by offloading input mutations to FPGAs by 4.7× without sacrificing any defect detection capability. It speeds up testing 10.3× on average by gathering meaningful signals from hardware execution directly by injecting in-kernel probes. This work fits the domain of software testing, as it targets HLS C/C++ dialects and it has the potential to significantly improve correctness in the new era of *heterogeneous computing*, where regular software developers write code in HLS C/C++ to exploit custom hardware acceleration.

ACKNOWLEDGMENTS

The participants of this research are in part supported by NSF grants 1956322, 1764077, 1460325, 2106383, 2106404, Amazon gift, Samsung contract, and Regents Faculty Fellowship offered by UCR Academic Senate.

REFERENCES

[1] Paul Alcorn. 2022. AMD to Fuse FPGA AI Engines Onto EPYC Processors, Arrives in 2023. https://www.tomshardware.com/news/amd-to-fuse-fpga-ai-engines-onto-epyc-processors-arrives-in-2023.
[2] Amazon.com. 2021. Amazon EC2 F1 Instances: Run Custom FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1.
[3] David F. Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Commun. ACM* 56, 4 (apr 2013), 56–63. https://doi.org/10.1145/2436256.2436271
[4] E Bendersky. 2012. PyCParser C Parser and AST Generator Written in Python.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, David Evans, Tal Maklin, and Dongyan Xu (Eds.). Association for Computing Machinery (ACM), United States of America, 2329–2344. https://doi.org/10.1145/3133956.3134020 ACM Conference on Computer and Communications Security 2017
, CCS 2017 ; Conference date: 30-10-2017 Through 03-11-2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[7] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. 2010. State-of-the-art in heterogeneous computing. *Scientific Programming* 18, 1 (2010), 1–33.

[8] Nazanin Calagar, Stephen D. Brown, and Jason H. Anderson. 2014. Source-level debugging for FPGA high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927496

[9] Jared Casper and Kunle Olukotun. 2014. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/2554688.2554787

[10] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[11] Andrew A Chien, Allan Snavely, and Mark Gahagan. 2011. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *Procedia Computer Science* 4 (2011), 1987–1996.

[12] Young-Kyu Choi and Jason Cong. 2017. HLScope: High-Level Performance Debugging for FPGA Designs. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 125–128. https://doi.org/10.1109/FCCM.2017.44

[13] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2593069.2596667

[14] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. 2018. SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for DNA Sequencing. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 206–206. https://doi.org/10.1109/FCCM.2018.00040

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[16] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. 2011. Customizable Domain-Specific Computing. *IEEE Design Test of Computers* 28, 2 (2011), 6–15. https://doi.org/10.1109/MDT.2010.141

[17] John Curreri, Greg Stitt, and Alan D. George. 2010. High-level synthesis techniques for in-circuit assertion-based verification. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1–8. https://doi.org/10.1109/IPDPSW.2010.5470747

[18] Ian Cutress. 2018. Intel Shows Xeon Scalable Gold 6138P with Integrated FPGA, Shipping to Vendors. https://www.anandtech.com/show/12773/intel-shows-xeon-scalable-gold-6138p-with-integrated-fpga-shipping-to-vendors.

[19] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. 2021. Hardware Support to Improve Fuzzing Performance and Precision. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2214–2228.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. *AFL++: Combining Incremental Steps of Fuzzing Research.* USENIX Association, USA.

[21] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. 2012. *High—Level Synthesis: Introduction to Chip and System Design.* Springer Science & Business Media.

[22] Philippe Garrault and Brian Philofsky. 2006. HDL coding practices to accelerate design performance. *Xilinx White Paper* 231 (2006), 1–22.

[23] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 127–135. https://doi.org/10.1109/FCCM.2019.00027

[24] Intel. 2021. Dense Linear Algebra. https://github.com/oneapi-src/oneAPI-samples/tree/6901f7203b549a651911fec694ffefad82ed0b35/DirectProgramming/C%2B%2BSYCL/DenseLinearAlgebra.

[25] Intel. 2021. DPC++ Reference. https://oneapi-src.github.io/DPCPP_Reference/.

[26] Intel. 2022. Devcloud. https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html.

[27] Intel. 2022. FPGA Optimization Guide for Intel® oneAPI Toolkits - Shannonization to Improve FMAX/II. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/single-work-item-kernels/loops/shannonization-to-improve-fmax-ii.html.

[28] Intel. 2022. FPGA Optimization Guide for Intel® oneAPI Toolkits - Transfer Loop-Carried Dependency to Local Memory. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/single-work-item-kernels/loops/transfer-loop-carried-dependency-to-local-memory.html.

[29] Intel. 2022. FPGA Optimization Guide for Intel® oneAPI Toolkits - Unroll Loops. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/single-work-item-kernels/loops/unroll-loops.html.

[30] Intel. 2022. Intel® Arria® 10 GX FPGA Overview. https://www.intel.com/content/www/us/en/products/details/fpga/arria/10/gx/products.html.

[31] Intel. 2022. Intel® Stratix® 10 GX FPGA Overview. https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[33] Alexandra Kourfali and Dirk Stroobandt. 2020. In-Circuit Debugging with Dynamic Reconfiguration of FPGA Interconnects. *ACM Trans. Reconfigurable Technol. Syst.* 13, 1, Article 5 (jan 2020), 29 pages. https://doi.org/10.1145/3375459

[34] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 8 pages. https://doi.org/10.1145/3240765.3240842

[35] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (sep 2021), 39 pages. https://doi.org/10.1145/3469660

[36] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[37] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–9.

[38] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 809–814. https://doi.org/10.1145/3236024.3275525

[39] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 45–54.

[40] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* PP (10 2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[41] Joshua S. Monson and Brad Hutchings. 2015. Using source-to-source compilation to instrument circuits for debug with High Level Synthesis. In *2015 International Conference on Field Programmable Technology (FPT)*. 48–55. https://doi.org/10.1109/FPT.2015.7393129

[42] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2016. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *Commun. ACM* 59, 11 (Oct. 2016), 114–122. https://doi.org/10.1145/2996868

[43] Xiaoke Qin and Prabhat Mishra. 2014. Scalable Test Generation by Interleaving Concrete and Symbolic Execution. In *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems (VLSID '14)*. IEEE Computer Society, USA, 104–109. https://doi.org/10.1109/VLSID.2014.25

[44] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature.

[45] Ruyman Reyes and Victor Lomüller. 2016. SYCL: Single-source C++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 673–682.

[46] Hongbo Rong. 2017. Programmatic Control of a Compiler for Generating High-performance Spatial Hardware. *CoRR* abs/1711.07606 (2017). arXiv:1711.07606 http://arxiv.org/abs/1711.07606

[47] Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. 2011. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC*. 1102–1105. https://doi.org/10.1109/ASICON.2011.6157401

[48] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. Silifuzz: Fuzzing cpus by proxy. *arXiv preprint arXiv:2110.11519* (2021).

[49] Congxi Song, Xu Zhou, Qidi Yin, Xinglu He, Hangwei Zhang, and Kai Lu. 2019. P-Fuzz: A Parallel Grey-Box Fuzzing Framework. *Applied Sciences* 9, 23 (2019). https://doi.org/10.3390/app9235100

[50] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 999–1010. https://doi.org/10.1145/3377811.3380386

[51] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MEMLOCK: Memory Usage Guided Fuzzing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 765–777. https://doi.org/10.1145/3377811.3380396

[52] Xilinx. 2021. UltraScale Architecture and Product Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.

[53] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2307–2324.

[54] Mohamed Zahran. 2017. Heterogeneous computing: Here to stay. *Commun. ACM* 60, 3 (2017), 42–45.

[55] Michał Zalewski. 2021. American Fuzz Loop. http://lcamtuf.coredump.cx/afl/.

[56] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 161–170.

[57] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics using Framework Abstraction. In *The 35th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3324884.3416641

[58] Qian Zhang, Jiyuan Wang, and Miryung Kim. 2021. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 242–254.

[59] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. , 10 pages. https://doi.org/10.1145/3174243.3174255