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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:


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, FPGAs 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
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 but 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 and 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 = AB 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 x 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 naive 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 of FPGA optimizations, 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

![Figure 1: Latency breakdown of running applications on heterogeneous architectures. On average, data transfer into kernels takes 60% of execution time, highlighted in gray.](image-url)
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 defects is found after 4 hours. Per the open science policy, we make HFuzz’s artifacts, benchmark programs, and datasets available with this submission (uploaded with this submission).

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. 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].

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 following example.

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 Line 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 statically specify hardware resources, a heterogeneous application often contains defects that cannot be detected statically via a compiler analysis.
This is a problem that universally exists with all HLS languages. To illustrate, consider the real defects in the N-body-simulation.

**Divide By Zero in Nbody-Simulation.** For code in Figure 2, with the input \([p.\text{pos}]=[(1,2,4),\ldots, (1,2,4)]\), the velocity calculation on an FPGA A10 device produces absurdly large numbers \([p.\text{vel}]=[(214748364,\ldots),\ldots]\). 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 \(\text{sqr=0}\) 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., \(\text{sqr=0}\) at Line 13). This is because when \(\text{sqr}\) is close to zero, \(\text{acc}\) becomes large. When the inputs \([p.\text{pos}]=[(81,0,0),(81,1,0),(81,0,1),\ldots]\) are sent to the kernel, it produces a small value \(\text{sqr=1}\), leading to overflow for the variables \(\text{acc1}\); 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 \(\text{sqr}\) 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 \(\text{sqr}\) 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 \(\text{sqr}\). As a result, HFuzz is able to effectively detect overflow when \(\text{sqr}\) reaches the small value \(\text{sqr=1}\) and divide-by-zero defects when \(\text{sqr}\) 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 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 \(p’\), 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 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 Line 10-15, inputs that advance either software or hardware feedback are saved to the input queue for the next fuzzing iteration.

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.

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

```cpp
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    DeviceToHostMax_sum::write(max_sum);//sum's Value Range Probe
13    DeviceToHostMin_sum::write(min_sum);}};
14 //Second kernel...
15 h.single_task([=]() {
16    for (size_t i = 0; i < number_element; ++i) {
17      out[i] = KToKPipe::read();
18      out[i] = reciprocalTransform(output[i]);
19      KToKPipeSize--;//Pipe Usage Probe
20      DeviceToHostKToKPipe::write(KToKPipeSize);
21    }
22    for(int i=0; i<num_elements; i++) {//SW monitor for kernel output
23      outmin=min(outmin, output[i]);
24      outmax=max(outmax, output[i]);
25  });}
```

Figure 3: Matrix transform: inserted value range probes are in the green rectangle. Inserted pipe 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.
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 MS 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 $[-26,5,7,9,28]$ 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.

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...
We thus apply data preloading to transfer data from global memory which can slow down the overall performance. A

\[ \begin{align*}
\alpha &= \text{update factor} \\
A[i] &= 0 \\
\end{align*} \]

It necessitates the implementation of a delay in the critical path. Each time the if statement induces extra hardware overhead, as it increases the null check for each element in the matrix. As shown in Line 2 of Figure 4a at Line 2. This optimization leads to a 1.31x speedup 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 the 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 a branch-coverage guided fuzzing similar to AFL and performs input mutations on CPU side.
Benchmarks. We choose seven applications from Intel’s OneAPI GitHub repositories [24]: (R1) Matrix-transform. It has two kernels—one for matrix multiplication $M = A \times B$ 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 predefined 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, 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.

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 3.1x more defects than HeteroFuzz. For example, for R5 Nbody-simulation, without monitoring in-kernel variable $sqr$, HeteroFuzz cannot find divide-by-zero error we mentioned in Section 2.2 at Line 16-18 in Figure 3. When using HFuzz, 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 the input vector $a$ includes a large number such as 2090401586. By

![Figure 5: # Number of Defects](https://example.com/figure5)

<table>
<thead>
<tr>
<th>ID</th>
<th>Symptom</th>
<th>Description</th>
<th>HeteroFuzz Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Kernel</td>
<td>The value of intermediate runtime variables $sum$ at line 2 of</td>
<td>✓</td>
</tr>
<tr>
<td>S2</td>
<td>Pipe</td>
<td>Pipe write failure happens when FPGA attempts to write</td>
<td>×</td>
</tr>
<tr>
<td>S3</td>
<td>Pipe</td>
<td>Pipe read hang happens when FPGA attempts to read</td>
<td>×</td>
</tr>
<tr>
<td>S4</td>
<td>Division by Zero</td>
<td>Equals 0, leading to divide by zero at line 21.</td>
<td>×</td>
</tr>
<tr>
<td>S5</td>
<td>Incorrect Loop</td>
<td>Results when the input array size $num_{elements}$ is not multiple of 2.</td>
<td>✓</td>
</tr>
</tbody>
</table>
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 communication 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 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 10.3x 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 (5.3x speed-up) by avoiding redundant data transfer and parallelizing input mutations. In R2, NoKernelMutation and HFuzz enumerate 15824 and 112940 inputs respectively, leading to 7.1x 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 4.7x 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 speeds up fuzzing by 4.7x.

### 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 3.4x speed-up in the detection of defects by implementing hardware optimizations. Loop unrolling, shannazation, and fast memory access directly speed up the mutation process. Dynamic kernel sharing enables efficient input space exploration.

---

**Table 4: Resource overhead from injecting hardware probes.**

<table>
<thead>
<tr>
<th>ID/Program</th>
<th>#LUT</th>
<th>#FF</th>
<th>#RAM</th>
<th>#DSP</th>
<th>Freq (/MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1/Orig</td>
<td>15392</td>
<td>25088</td>
<td>137</td>
<td>4.5</td>
<td>247</td>
</tr>
<tr>
<td>Matrix_trans/Probe</td>
<td>17905</td>
<td>34320</td>
<td>192</td>
<td>4.5</td>
<td>246</td>
</tr>
<tr>
<td>R2/Orig</td>
<td>9592</td>
<td>14466</td>
<td>492</td>
<td>16</td>
<td>259</td>
</tr>
<tr>
<td>Matrix_mul/Probe</td>
<td>12032</td>
<td>19443</td>
<td>492</td>
<td>16</td>
<td>247</td>
</tr>
<tr>
<td>R3/Orig</td>
<td>11545</td>
<td>18494</td>
<td>106</td>
<td>6.5</td>
<td>273</td>
</tr>
<tr>
<td>Complex_mul/Probe</td>
<td>11203</td>
<td>27117</td>
<td>106</td>
<td>6.5</td>
<td>246</td>
</tr>
<tr>
<td>R4/Orig</td>
<td>60468</td>
<td>92249</td>
<td>555</td>
<td>195</td>
<td>221</td>
</tr>
<tr>
<td>APSP/Probe</td>
<td>64327</td>
<td>101229</td>
<td>558</td>
<td>195</td>
<td>212</td>
</tr>
<tr>
<td>R5/Orig</td>
<td>23642</td>
<td>44352</td>
<td>309</td>
<td>34</td>
<td>270</td>
</tr>
<tr>
<td>Nbody_sim/Probe</td>
<td>27612</td>
<td>50549</td>
<td>317</td>
<td>34</td>
<td>260</td>
</tr>
<tr>
<td>R6/Orig</td>
<td>68706</td>
<td>64987</td>
<td>395</td>
<td>67</td>
<td>273</td>
</tr>
<tr>
<td>HMM/Probe</td>
<td>56562</td>
<td>87392</td>
<td>491</td>
<td>67</td>
<td>247</td>
</tr>
<tr>
<td>R7/Orig</td>
<td>2239</td>
<td>1357</td>
<td>67</td>
<td>12</td>
<td>279</td>
</tr>
<tr>
<td>Match num/Probe</td>
<td>3828</td>
<td>2033</td>
<td>73</td>
<td>12</td>
<td>259</td>
</tr>
</tbody>
</table>
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 arithmetics). In general, the overhead depends on the complexity of the original kernels. 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.

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, 8\% extra RAM, and reduces frequency by 5\% on average. However, it enables an overall 10.3\times 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], both of which are currently among the most widely used FPGAs. This specific configuration may constrain the applicability of our results to other devices, as the divergence symptoms detected could differ across different platforms, such as Intel’s Altera. Although the absolute values of execution time and symptoms are contingent upon the particular configuration, we believe that HFuzz will preserve its overall advantages in terms of acceleration and divergence-detection capability when adapted to diverse 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 rectify potential transformation errors. This insertion process can become challenging, particularly when dealing with large heterogeneous programs with complex in-kernel logic. 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. Rosetta benchmarks [59] and heterogeneous applications in Intel Devcloud are comparable in size (i.e., hundreds of lines of code.) Testing difficulties 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., inner kernel variables, loop unrolling), as well as the nuanced details of the communication channels between both the software and hardware and 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. PerFuzz [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. While 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...
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. Kourofali and Stroobandt [35] 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, HFuzz leverages new FPGA-level optimizations for mapping input generation and kernel invocation to FPGAs and empirically demonstrates significant fuzzing speedup from these optimizations (3.4×).

### 7 DATA AVAILABILITY

Per the open science policy, we make HFuzz’s artifacts, benchmark programs, and datasets available on [https://github.com/wjy99-c/HFuzz](https://github.com/wjy99-c/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 need 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 4.7x 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.

### 9 ACKNOWLEDGEMENT

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.


