LLM-Guided Differential Fuzzing for Detecting Platform-Specific Bugs in Scientific Applications

Manish Motwani, Aakash Kulkarni[§], and Yunhan Qiao[§]
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, USA
{motwanim, kulkaraa, qiaoy}@oregonstate.edu

Matthew Davis[§] and Ziyan Chen[§]
School of Computer Science
Georgia Institute of Technology
Atlanta, GA
{mdavis438, zchen910}@gatech.edu

***Abstract*—Modern scientific and AI-based applications run complex workloads across heterogeneous platforms, such as CPUs and accelerators (e.g., GPUs). These applications often use platform-specific libraries to move data between compute units, which can introduce heterogeneous bugs, where applications behave differently when running on heterogeneous systems (combination of CPUs and accelerators) compared to CPU-only systems. Such bugs are especially difficult to detect with traditional software testing techniques because they overlook platform-specific characteristics.**

To address this challenge, we present HeteroBugDetect, an automated framework that leverages large language models (LLMs) to drive platform-aware testing of scientific applications, including AI-based workloads. HeteroBugDetect's novelty lies in combining the power of modern LLM, grammar-based kernel-sensitive fuzzing, and differential testing into a unified framework for uncovering platform-dependent bugs. We demonstrate the effectiveness of HeteroBugDetect through a case study on LAMMPS, a widely used molecular dynamics simulator. HeteroBugDetect automatically detected 8 of 20 known heterogeneous bugs and uncovered 2 previously unknown bugs, improving LAMMPS's correctness and reliability across diverse platforms.

***Index Terms*—Platform-specific software testing, LLM, Fuzz testing, Differential testing, heterogeneous computing**

I. Introduction

Scientific applications are essential tools for researchers across diverse domains. They simulate complex real-world phenomena, allowing scientists to test hypotheses and analyze intricate systems before undertaking costly and time-consuming physical experiments. The unprecedented rise in parallelism and heterogeneity places an unsustainable burden on developers of such scientific applications, who must ensure their software meets stringent performance, correctness, and reliability requirements for scientific simulations and analyses. While the specialized accelerators such as GPUs, programming models (e.g., OpenMP [1], CUDA [2], OpenCL [3]), and performance portable libraries (e.g., Kokkos[4] and Raja [5]) enable developers to efficiently manage and perform computations across heterogeneous devices, they introduce a new class of errors known as *heterogeneous* or *portability bugs* [6]. As future computing systems become even more heterogeneous and complex, the already challenging process of testing scientific applications is increasingly becoming more difficult. For instance, bugs may arise when developers misuse library APIs to copy data between the CPU and GPU, where parallel computations are performed. These bugs are missed by developer-written tests because they can only be identified by comparing application outputs for the same inputs across different platforms. While several automated testing techniques (e.g., HeteroFuzz [7], HFuzz [8], and Klokkos [6]) have been developed to detect heterogeneous bugs, they are typically tailored to specific domains or libraries and may not scale to complex, real-world scientific applications. HeteroBugDetect addresses this limitation by combining grammar-based fuzzing [9] using *kernel sensitive* metrics, differential testing [10], and state-of-the-art large language model (LLM) [11]. The primary goal of HeteroBugDetect is to improve the trustworthiness and reliability of real-world scientific applications considering their inherent complexity and the diverse platforms they operate in.

The HeteroBugDetect has three key components that enable the bug detection process. First, it uses an LLM to generate diverse inputs for the scientific software under test (SUT) from the natural language user documentation of the software. This enables developers to use HeteroBugDetect for applications that do not provide extensive suite of examples to run the software. Second, for generating test inputs that are more likely to expose heterogeneous bugs, HeteroBugDetect uses a grammar-based, *kernel-sensitive* fuzzing technique that mutates inputs based on changes in parallel executions and variables' properties that interact across heterogeneous platforms. Third, HeteroBugDetect uses a differential driver that executes platform-specific systems on the same fuzzed inputs and compares outputs, flagging discrepancies that exceed user-specified threshold as potential heterogeneous bugs. This tailored approach improves the efficiency and likelihood of detecting the heterogeneous bugs that may remain hidden when using traditional software testing methodologies.

As no benchmark of heterogeneous bugs currently exists, we implement a prototype of HeteroBugDetect targeting C++-based scientific applications and construct our own bug dataset through a case study on LAMMPS [12], a widely used open-source HPC application. We analyze user-reported bug reports and common developer mistakes to create a dataset of

[§]These authors contributed equally to the work.

20 heterogeneous bugs grouped into 7 categories. While our evaluation focuses on LAMMPS, the design of HeteroBugDetect is general and applicable to other scientific applications with platform-dependent behavior. HeteroBugDetect detects 8 (40%) of the 20 bugs across five categories and uncovers 2 previously unknown bugs, demonstrating its potential to improve the reliability of real-world scientific applications.

The main contributions of this paper are:

- HeteroBugDetect, an automated approach to detect heterogeneous bugs in HPC scientific applications.
- HeteroBugs, the first dataset of real-world heterogeneous bugs in a widely used scientific application.
- A prototype implementation of HeteroBugDetect supporting multiple backends and applications.
- A replication package for our empirical evaluation available at https://doi.org/10.7910/DVN/VHO9RH.

II. Background and Motivation

High-performance computing (HPC) has transformed scientific research by enabling large-scale simulations across disciplines like physics, chemistry, and biology. Modern HPC systems integrate supercomputers, specialized accelerators (e.g., GPUs), programming models (OpenMP [1], CUDA [2], OpenCL [3]), and performance-portable libraries (e.g., Kokkos [4], RAJA [5]) to deliver scalable computational power. Scientific applications including those using AI-based models built on these platforms allow researchers to simulate and analyze complex phenomena with unprecedented precision. For example, LAMMPS [12] is an open-source molecular dynamics simulator used widely for modeling atomic and molecular systems. Designed for heterogeneous platforms, LAMMPS supports execution on GPUs and CPUs. However, its scale and complexity also expose it to *heterogeneous bugs*–errors caused by incorrect data handling between compute units. These bugs can lead to inconsistent results or application crashes, making them difficult to detect and critical to address for ensuring simulation correctness.

For example, top of Figure 1 shows a HeteroBugDetect-generated simulation script for a physics simulation where we model how copper atoms move and interact inside a small 3D box. The script is run using LAMMPS that executes the simulation for 100 timesteps according to the laws of physics and records thermodynamic properties such as temperature, energy, pressure, and volume at every step. Interestingly, when we run this simulation on a CPU-only system and then again on a GPU, we get different results. The bottom of Figure 1 presents partial differences in the output logs generated by executing this script on two platforms: a Kokkos-enabled GPU version of LAMMPS using the CUDA backend, and the default CPU version of LAMMPS. As shown, the input script executes successfully for 100 steps on the CPU; however, it crashes on the GPU after step 73. Notably, even before the crash, key output values (Temp, PotEng, KinEng, TotEng, Press) differ significantly between platforms. Running the same simulation for 70 steps still triggers bug detection by HeteroBugDetect, as the L1 norm of output differences exceeds the user-defined threshold (0.05), as shown in Figure 2.

```
units            metal
dimension        3
boundary         p p p
atom_style       charge
lattice          fcc 5.43
region           box block 0 10 0 10 0 10
create_box       1 box
create_atoms     1 box
mass             1 12.01
set              group all charge 0.0
pair_style       comb
pair_coeff       * * ffield.comb Cu
timestep         0.005
thermo 1
thermo_style     custom step temp pe ke etotal press vol
thermo_modify    flush yes
velocity         all create 300 12345 dist gaussian
fix              1 all nve
run 100
```

```
--- log.lammps.gpu
+++ log.lammps.cpu
-Step  Temp       PotEng      KinEng      TotEng      Press       Volume
-69   14909911   20159.334   7707101.9   7727261.2   51659006    160103.01
-70   15984876   18722.673   8262763.4   8281486.1   55360299    160103.01
-71   17085107   25251.3     8831485.2   8856736.5   59193083    160103.01
-72   18474970   21554.438   9549921     9571475.5   63963371    160103.01
-73   19594644   21974.55    10128693    10150668    67828632    160103.01
-74   20766721   23096.653   10734553    10757650    71880139    160103.01
...
-99    54546022   26167.913   28195456    28221624    1.883785e+08  160103.01
-100   56408135   27665.155   29158003    29185669    1.9481599e+08 160103.01
- Loop time of 40.3395 on 1 procs for 100 steps with 4000 atoms
+Step  Temp       PotEng      KinEng      TotEng      Press       Volume
+69   14627191   21494.718   7560960.4   7582455.1   50696568    160103.01
+70   15870552   21310.699   8203667.8   8224978.5   54980363    160103.01
+71   17165522   22659.26    8873052.5   8895711.7   59446629    160103.01
+72   18622021   22725.965   9625933.2   9648659.2   64473752    160103.01
+73   20033276   20781.372   10355427    10376208    69323575    160103.01
+ ERROR: Lost atoms: original 4000 current 3999 (src/thermo.cpp:491)
+ Last command: run 100
```

Fig. 1. A HeteroBugDetect-generated COMB simulation script produces divergent results on the GPU (Kokkos) vs. CPU version of LAMMPS.

```
COMPARING Temp ✗
L1= 2.01402e+06, ref_L1= 300, diff= 2.01372e+06 ≥ 0.05
COMPARING PotEng ✗
L1= 4877.97, ref_L1= 0, diff= 4877.97 ≥ 0.05
COMPARING KinEng ✗
L1= 1.04107e+06, ref_L1= 155.073, diff= 1.041e+06 ≥ 0.05
COMPARING TotEng ✗
L1= 1.04336e+06, ref_L1= 155.073, diff= 1.0432e+06 ≥ 0.05
COMPARING Press ✗
L1= 6.99396e+06, ref_L1= 1034.56, diff= 6.9931e+06 ≥ 0.05
COMPARING Volume ✓
L1= 160103, ref_L1= 160103, diff= 0
HETEROGENEOUS BUG DETECTED, RESULTS DON'T MATCH!
```

Fig. 2. The HeteroBugDetect-generated COMB simulation script showin in Figure 1 still detects divergent results when executed for 70 steps.

Real users have also reported such bugs. For instance, in LAMMPS GitHub issue[1], developers discovered that a variable "*wasn't being updated on GPUs correctly, and somehow this slipped through all the regression tests.*" Despite extensive testing, the bug went unnoticed until a user compared simulation results across different platforms. As scientific workflows increasingly integrate AI surrogates, GPUs, and hybrid systems, ensuring cross-platform consistency becomes critical—highlighting the need for tools that can detect, explain, and help fix such heterogeneity-induced bugs, aligning with broader advances in automatic program repair [13], [14]. HeteroBugDetect detects such bugs by generating and comparing simulation logs across different platforms.

[1]https://github.com/lammps/lammps/pull/3541

III. Related Work

HeteroFuzz [7] is a testing technique designed to efficiently detect platform-dependent divergence in heterogeneous applications that use CPUs and FPGAs. HFuzz [8] builds on top of HeteroFuzz to speed up the fuzzing process via FPGA kernel probes and accelerating input mutations by offloading them to hardware, using FPGA optimizations like loop unrolling and dynamic kernel sharing. However, both techniques are limited to FPGA-based systems and cannot directly test applications like LAMMPS, which primarily utilize CPU/GPU architectures and do not use FPGA-specific pragmas or hardware simulation flows. Additionally, unlike HeteroBugDetect these techniques detect FPGA-specific issues (e.g., pipeline stalls, bitwidth errors) rather than generic heterogeneous bugs for arbitrary platforms. FuzzyFlow [15] is a framework that accelerates the detection of optimization bugs in HPC applications by using parametric dataflow representations to isolate optimization side effects, extract minimal test cases, and validate semantic equivalence through differential testing and gray-box fuzzing. However, FuzzyFlow cannot detect heterogeneous bugs as it targets optimization-induced errors within a single dataflow framework (e.g., DaCe) rather than cross-platform discrepancies. Its reliance on predefined dataflow representations and input-configuration fuzzing overlooks platform-specific runtime behaviors that drive portability failures.

Recent work such as TitanFuzz [16] and FuzzGPT [17] use LLMs to generate inputs for fuzzing deep learning libraries, focusing on valid test case generation and improved bug discovery. However, they do not address heterogeneous bugs caused by platform-dependent behaviors in scientific applications. Finally, Klokkos [6] detects heterogeneous bugs in Kokkos-based parallel programs using static analysis and symbolic execution. Unlike Klokkos, our approach requires no compiler instrumentation or internal IR representations, making it suitable for a broader range of applications..

IV. HeteroBugDetect Approach

Figure 3 shows HeteroBugDetect architecture. It takes three inputs: the codebase of the scientific software under test (SUT), a natural language description of a supported simulation, and a manually constructed grammar of valid commands and arguments. The output includes potential heterogeneous bugs along with the simulation scripts (test inputs) that trigger them. The following sections describe the three core components of HeteroBugDetect.

A. Seed Input Generator (SIG)

Scientific applications often require domain-specific simulation scripts with precise commands and arguments, which traditional input generators (e.g., EvoSuite [18], Swami [19]) cannot produce. While some developers provide example scripts or documentation, these are typically incomplete and may not cover the full feature space. As a result, relying solely on developer-provided scripts would significantly limit HeteroBugDetect's applicability. To address this challenge, HeteroBugDetect automatically generates valid simulation scripts using the power of modern large language model (LLM) ("seed input generator" in Figure 3). Specifically, HeteroBugDetect uses OpenAI's *gpt-3.5-turbo* due to the model's top performance in general natural language processing tasks and within scientific research compared to other LLMs [20]. HeteroBugDetect generates diverse, valid simulation scripts for the HPC scientific software-under-test (SUT) from the natural language descriptions of the simulations supported by the SUT.

Figure 4 shows the prompt template HeteroBugDetect uses to generate diverse, valid simulation scripts via an LLM. The top presents the general template; the bottom shows an example instantiated for generating a LAMMPS script using the COMB potential [21]. Since LLMs may not always produce valid scripts on the first try, HeteroBugDetect applies a chain-of-thought refinement process [22], using feedback from parsing script execution errors to iteratively improve the output. To enhance diversity, HeteroBugDetect also varies prompt parameters when querying the LLM. This component, called the *Seed Input Generator (SIG)*, produces *seed inputs*—the initial simulation scripts used by later components.

The SIG takes as input: (1) a natural language description of a simulation, (2) the name of the SUT, (3) the executable binary of the SUT, (4) a generative LLM, (5) a set of parameters for the LLM, and (6) the maximum number of attempts allowed to generate a valid simulation script. It begins by initializing a prompt using the simulation description and SUT name. For each combination of LLM parameters (e.g., *temperature* for *gpt-3.5-turbo*), SIG prompts the LLM up to 100 times to generate a valid simulation script.

Each LLM output is parsed using regular expressions and executed on the SUT to check for errors. If the script runs successfully and within a specified timeout, it is saved along with metadata (attempt number and parameter values). If the script fails or times out, the corresponding error messages or a directive to reduce execution time are appended to the prompt, and the generation is retried. This iterative process produces a diverse set of valid simulation scripts for the SUT.

B. Grammar-Based Kernel-Sensitive Fuzzer

While traditional fuzzing [23] effectively detects security bugs, it is insufficient for uncovering heterogeneous bugs, which require valid inputs and cross-platform execution. To address this, HeteroBugDetect employs a grammar-based, kernel-sensitive fuzzer ("fuzzer" in Figure 3) that mutates seed inputs from SIG using the syntax of supported commands and arguments encoded in the manually constructed grammar.

HeteroBugDetect prioritizes mutations to commands likely to impact parallel computation, specifically those that affect *kernel-sensitive variables* that are accessed from GPU during computation. For LAMMPS, two authors manually constructed a JSON-based grammar from developer documentation. This grammar encodes command structure, allowed arguments, optional fields, and dependencies, enabling semantically valid mutations. While building such grammars requires manual effort, **it is a one-time, incremental task**. For

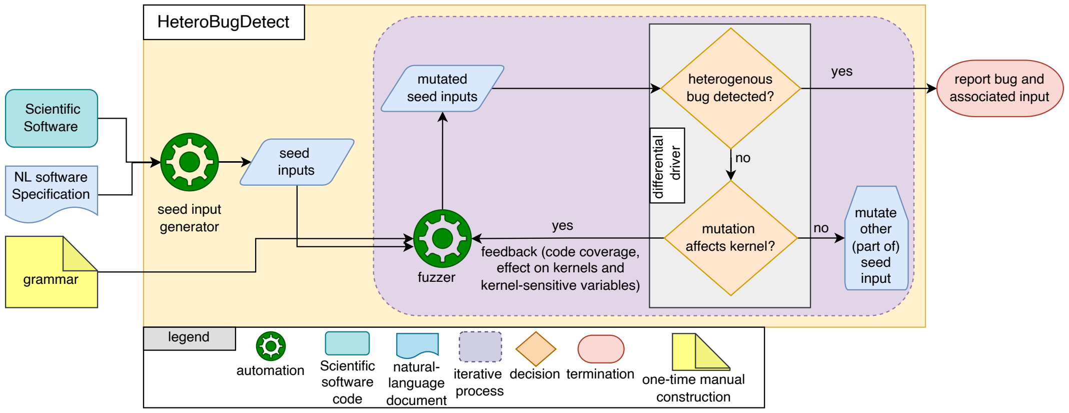
HeteroBugDetect
Scientific Software
NL software Specification
grammar
seed input generator
seed inputs
mutated seed inputs
fuzzer
feedback (code coverage, effect on kernels and kernel-sensitive variables)
yes
differential driver
heterogenous bug detected?
no
mutation affects kernel?
yes
no
mutate other (part of) seed input
report bug and associated input
legend
automation
Scientific software code
natural-language document
iterative process
decision
termination
one-time manual construction

Fig. 3. The HeteroBugDetect architecture.

Prompt template: You are a helpful assistant who understands how to use <SUT name> for scientific simulation. Generate a simulation script that can be executed on <SUT name> to simulate <simulation description>. Do not provide any explanations and do not add any comments and blank lines in the script.
Example prompt: You are a helpful assistant who understands how to use LAMMPS for scientific simulation. Generate a simulation script that can be executed on LAMMPS to simulate models using the COMB potential. Do not provide any explanations and do not add any comments and blank lines in the script.
Generated simulation script:

```
units           metal
dimension       3
boundary        p p p
atom_style      charge
lattice         fcc 5.43
region          box block 0 10 0 10 0 10
create_box      1 box
create_atoms    1 box
mass            1 12.01
set             group all charge 0.0
pair_style      comb
pair_coeff      * * ffield.comb Cu
timestep        0.005
thermo          1
thermo_style    custom step temp pe ke etotal
        press vol
thermo_modify   flush yes
velocity        all create 300 12345 dist
        gaussian
fix             1 all nve
run 10
```

Fig. 4. Prompt template used by HeteroBugDetect to generate diverse simulation scripts. The example shows the template instantiated for simulating the COMB potential in LAMMPS, along with a successfully generated script.

LAMMPS, the initial grammar was created in a few hours and can be easily extended. Figure 5 shows a partial view of the LAMMPS grammar, including how keywords like `No change` prevent mutation and `style_arg` defines valid argument choices. For instance, when mutating `create_atoms`, the first argument is drawn from `[1, 2, 3]`, while subsequent arguments are randomly selected from `style_arg` options.

While this grammar-based approach produces valid inputs, uniform random mutation is still suboptimal—only some commands affect kernel behavior. Thus, HeteroBugDetect dynamically prioritizes commands based on their observed impact on parallel computation. After each mutation, HeteroBugDetect computes a feedback signal in the range $[-0.5, 0.5]$ using six metrics: (1) **Memory Leak Fraction (ML)**: This measures the fraction of kernel-sensitive variables whose allocated memory space is not completely freed after executing the mutated input. This is computed by measuring the difference between the sum of memory allocated and de-allocated for each variable throughout the program execution and computing the fraction of variables showing memory leaks over all variables. The value of this metric varies from 0 to 1. (2) **Parallel Loop Count (PL)**: This measures the number of parallel loops executed during the execution of the mutated input. The value of this metric is a positive integer ≥ 0. (3) **Parallel Scan Count (PS)**: This measures the number of parallel scan events (e.g. prefix sum, prefix max) triggered during the execution of the mutated input. The value of this metric is a positive integer ≥ 0. (4) **Parallel Reduction Count (PR)**: This measures the number of parallel reduce events (e.g., sum, max) triggered during the execution ofthe mutated input. The value of this metric is a positive integer ≥ 0. (5) **Fence Execution Count (FE)**: This measures the number of fences (barriers between execution of different kernels) triggered during the execution of the mutated input. The value of this metric is a positive integer ≥ 0. (6) **Deep Copy Count (DC)**: This measures the number of deep copy events (copying data of some variable across heterogeneous devices e.g., CPU to GPU) triggered during the execution of the mutated input. The value of this metric is a positive integer ≥ 0.

Initially, all script commands have equal mutation probability ($1/n$ for n commands). After each iteration, HeteroBugDetect mutates a command, executes the script, and compares metric values with the previous iteration. The signal is computed as the average normalized difference across all metrics (scaled to $[-0.5, 0.5]$). A positive signal indicates that the mutated command increased parallel activity and should be prioritized; a negative signal suggests reduced parallelism. The mutation probability of a command is then updated as:

$$probability = probability * (1 + signal)$$

```
"create_box": {
    "arg_1": [1, 2, 3],
    "arg_2": ["No change"]
},
"create_atoms":{
    "arg_1": [1, 2, 3],
    "style_arg": {
        "box": [],
        "region": ["No change"],
        "random": {
            "arg_1": ["integer"],
            "arg_2": ["integer"],
            "arg_3": ["No change"]
        }
    }
},
"mass": {
    "arg_1": [1, 2, 3, "C", "*"],
    "arg_2": ["float"]
},
```

Fig. 5. Partial view of the manually-constructed grammar in JSON format used by HeteroBugDetect to generate diverse simulation scripts for LAMMPS.

Finally, probabilities are normalized to sum to 1, guiding the fuzzer to favor impactful mutations over time.

C. Differential Driver

Traditional fuzzing methods are limited in their ability to detect heterogeneous bugs because they operate within a single execution environment and do not account for platform-specific behaviors. Furthermore, scientific simulations can yield minor discrepancies in results across different platforms due to inherent randomness and variations in floating-point precision. Consequently, simply performing a `diff` on output logs from different platforms and flagging any differences as heterogeneous bugs would generate numerous false positives.

To address these challenges, HeteroBugDetect has the differential driver component that reformulates traditional fuzzing to report heterogeneous bugs instead of crashes ("differential driver" in Figure 3). To enable coverage-guided fuzzing while executing the HPC scientific software-under-test (SUT) on different platforms *within the same process*, differential driver compiles the SUT as a static library and invokes the SUT's functions to launch the same simulation multiple times using a different set of arguments specifying the input simulation script and the underlying platform. This allows the fuzzer to measure the code coverage on the SUT considering the execution on multiple platforms in a single process. For example, when testing LAMMPS, the differential driver imports LAMMPS compiled as a static library and invokes the LAMMPS functions on the same fuzzer-generated input script using CPU-only, and CPU + GPU platforms.

To filter out false-positives that arise due to minor differences in the simulation results produced using different platforms, the differential driver implements various error norms (L1 [Manhattan], L2 [Euclidean], and Max [Infinity] norms) that are typically used in scientific simulations with an error threshold that can be set by the user or developer to make HeteroBugDetect report a bug only if the differences observed using error norm are greater than pre-defined threshold. If HeteroBugDetect generates a script whose simulation results across different platforms are more than the specified threshold, it reports that observation as a heterogeneous bug along with the input script that causes divergent outputs.

bug category	**bug IDs**	**# of bugs**
Incorrect synchronization between host and device variables	1, 4, 6, 11, 15, 16, 18, 19	8
Missing synchronization between host and device variables	7, 8, 10, 12, 13, 17	6
Accessing memory in device space from host	2	1
Missed copying variable data from host to device or vice versa	5	1
Incorrect copying of variables from host to device or vice versa	20	1
Use of stale data	9, 14	2
Concurrent modification to shared variable from host and device	3	1
total		20

Fig. 6. Types of heterogeneous bugs and their associated bug IDs and count in the HeteroBugs dataset. *host* refers to CPU while *device* refers to GPU.

V. Evaluation

This section describes the dataset, metrics, implementation, and evaluation for HeteroBugDetect.

A. Dataset

As there is no existing dataset of heterogeneous bugs in real-world scientific applications, we introduce HeteroBugs, a collection of 20 bugs grouped into seven categories (see Figure 6). We constructed the dataset by mining GitHub issues reported by LAMMPS users and consulting experienced developers of HPC applications such as LAMMPS, Kokkos, and ArborX [24]. Bug categories were defined through analysis of public issue reports, developer discussions, and insights from prior work [6].

For each bug, HeteroBugs includes (1) a buggy source file, (2) the corresponding patch, and (3) an input script to reproduce the defect. Of the 20 bugs, 4 are user-reported and 16 are carefully synthesized based on common real-world bug patterns. Each bug is tested independently using separate builds and LAMMPS binaries to ensure isolation and avoid interference.

B. Evaluation Metrics

To evaluate HeteroBugDetect, we use the following two metrics: (1) **Unknown bug count (UBC)**: Measures unknown bugs in the latest stable version of an application. For a highly tested application such as LAMMPS, we expect UBC to be zero. (2) **Benchmark performance (BP):** Measures how many of the **20 known bugs** from 7 different categories in HeteroBugs dataset are detected. BP ensures a meaningful evaluation of bug-detection effectiveness even if UBC is zero.

C. Implementation and Experiment Procedure

We implemented HeteroBugDetect's SIG component using GPT-3.5 [25] to generate initial seed inputs. The Fuzzer component is built on AFL++ v4.10c [26], incorporating grammar-based fuzzing capabilities. HeteroBugDetect extends the *MemoryEvents* and *KernelLogger* tools from Kokkos Tools [27]

```
units           metal
dimension       3
boundary        p p p
atom_style      full
atom_modify     map array sort 0 0.0
read_data       polyethylene.data
pair_style      airebo 3.0
pair_coeff      * * CH.airebo C H
bond_style      harmonic
bond_coeff      1 2.59 4.12
angle_style     harmonic
angle_coeff     1 2.68 4.24
thermo_style    custom step temp etotal press density
thermo          10
neighbor        2.0 bin
neigh_modify    delay 0 every 8 check yes
fix             1 all nve
run             20
```

```
--- log.lammps.gpu
+++ log.lammps.cpu
-Step Temp          TotEng         Press          Density
-0    0             587.40916      12957.368      0.014556699
-10   271999.75     718.6126       32305.355      0.014556699
-ERROR on proc 0: Bond atoms 1 3 missing on proc 0 at step 16 (src/
       ntopo_bond_all.cpp:59)
+Step Temp          TotEng         Press          Density
+0    0             587.40916      12957.368      0.014556699
+10   271999.75     718.6126       32305.355      0.014556699
+20   2.2184837e+11 1.5463623e+09 -9.9300032e+10 0.014556699
+ERROR on proc 0: Atoms have moved too far apart (-2000.5306571379392) for
       minimum image (src/domain.cpp:986)
```

Fig. 7. A HeteroBugDetect-generated AIREBO simulation script produces divergent results on the GPU (Kokkos) vs. CPU version of LAMMPS.

for kernel-sensitive profiling and computing the signal. The differential driver component, implemented in C++, employs error norms (L1, L2, Max) to detect divergences in simulation results across different platforms using the thresholds defined by LAMMPS developers.

All experiments were executed on an Ubuntu 24.04 server with 2 Nvidia A40 GPUs. We evaluated HeteroBugDetect on LAMMPS using 58 examples from [21], generating natural language descriptions for each. For the 40 examples where seed inputs were successfully generated, we executed HeteroBugDetect with 10 random seeds, using a 30-minute timeout per seed. Each example ran for 7 hours to detect heterogeneous bugs, leveraging three different HeteroBugDetect configurations as part of the ablation study. In total, the experiments required approximately 2 weeks of wall clock time, with an additional week dedicated to analysis and result computation.

D. *Results*

This section describes our evaluation results in terms of the three research questions we ask.

1) RQ1: How effective is HeteroBugDetect in detecting heterogeneous bugs?: Considering unknown bug count (UBC), despite being heavily tested, HeteroBugDetect uncovered two previously unknown heterogeneous bugs in LAMMPS by generating simulation scripts that produce divergent outputs on different platforms.

The first unknown bug is shown in Figure 1 and described in Section II. The second unknown bug was identified using a script generated for the AIREBO potential. The top of Figure 7 shows the HeteroBugDetect-generated script that simulates a polyethylene system using the AIREBO potential with harmonic bond and angle interactions. The script runs for 20 timesteps in the NVE ensemble, using periodic boundaries

technique	# Valid	# Invalid	UBC	BP
AFL++ (Random)	7	11	0	0
HeteroBugDetect	958	0	1	8

Fig. 8. Comparison of HeteroBugDetect and random fuzzing on LAMMPS, showing valid scripts generated and defects detected from the HeteroBugs benchmark, averaged over 6 fuzzing runs. HeteroBugDetect outperforms random fuzzing in both script generation and defect detection.

and tracking temperature, energy, pressure, and density. The bottom of the figure shows partial differences in the output logs generated by executing this script on LAMMPS across two platforms: a Kokkos-enabled GPU version of LAMMPS using the CUDA backend, and the default CPU version of LAMMPS. As shown, the input script causes the simulation to crash after step 10 on the CPU, whereas on the GPU, it crashed after step 20. Notably, the error messages and the corresponding source files where the errors originated are significantly different between the two environments.

To compute benchmark performance (BP), we analyze how many of the 20 known bugs in the HeteroBugs benchmark HeteroBugDetect can detect. HeteroBugDetect detected 8 (40%) bugs (IDs 3, 10, 14, 15, 16, 18, 19, and 20) out of 20. These bugs fall into five categories: (1) incorrect synchronization between host and device variables, (2) concurrent modifications to shared variables, (3) missing synchronization between host and device, (4) use of stale data, and (5) incorrect copying of variables.

Notably, these 20 bugs require highly specific inputs or domain knowledge to trigger; detecting 8 bugs across five categories without domain-specific tuning demonstrates HeteroBugDetect 's effectiveness in identifying diverse, complex heterogeneous bugs, highlighting its value for improving reliability in real-world scientific applications.

HeteroBugDetect uncovered two previously unknown heterogeneous bugs and successfully detected 8 out of 20 bugs spanning five distinct categories in LAMMPS.

2) RQ2: How does HeteroBugDetect's performance compare to baseline?: To our knowledge, no existing technique detects heterogeneous bugs in real-world scientific applications such as LAMMPS. Therefore, we use random fuzzing as a baseline to evaluate HeteroBugDetect. Random fuzzing, implemented with AFL++ v4.10c [26], generates diverse inputs through mutation operations like bit-flipping, byte insertion, and altering integers in various ways. By comparing HeteroBugDetect to random fuzzing, we establish a reference point for evaluating its performance. We executed random fuzzing using the seed inputs generated by HeteroBugDetect's SIG component for the 40 examples on LAMMPS codebase using the same 10 randomly generated seeds and the same timeout of 30 min per seed as we used while executing HeteroBugDetect.

To ensure a fair and meaningful comparison, we compared HeteroBugDetect against random fuzzing using only the 6 runs where random fuzzing successfully generated results. Figure 8 shows the results of the comparison in terms of

(1) number of valid scripts generated, (2) number of invalid scripts generated, (3) the number of unknown bugs detected, and (4) the number of bugs detected out of the 20 known bugs in the HeteroBugs benchmark, averaged over 6 runs. As shown, HeteroBugDetect significantly outperforms the random fuzzing technique. The random fuzzing technique generated only 7 valid inputs with zero UBC and BP. In contrast, HeteroBugDetect generated 958 valid inputs and successfully identified 1 unknown bug along with 8 bugs from the HeteroBugs benchmark. These findings highlight the superior effectiveness of HeteroBugDetect in generating a much larger number of valid inputs and identifying more defects. The results suggest that HeteroBugDetect's targeted fuzzing approach by using grammar-based kernel-sensitive fuzzing significantly outperforms random fuzzing in terms of defect detection and overall efficiency.

The HeteroBugDetect surpasses the baseline by delivering higher input efficiency and successfully identifying bugs, making it a more effective tool for scientific software testing and heterogeneous bug detection.

3) RQ3: How do different components of HeteroBugDetect contribute to its performance?: To answer this research question, we analyze the effect of Seed Input Generator (SIG), the grammar-based fuzzing (Custom), and the kernel-sensitive metrics on the HeteroBugDetect performance.

SIG effectiveness: To evaluate the effectiveness of HeteroBugDetect's seed input generation, we measured how many example simulations SIG successfully produced valid scripts for LAMMPS. The more examples SIG can generate valid scripts for, the more effective HeteroBugDetect is in generating diverse test cases. Out of the 58 examples for which LAMMPS can be executed using Kokkos on GPUs and sequentially on CPUs, SIG successfully generated valid input scripts for 40 (69%) examples, averaging three attempts per example. These scripts were used in HeteroBugDetect's subsequent components. Of the 40 examples, 27 (67.5%) required no modifications, while the remaining 13 (32.5%) needed minor manual adjustments that included specifying paths to domain-specific data files (e.g., for simulations using the EIM potential, the required data file was provided by developers[2]) or correcting the order of commands in the generated scripts. The number of distinct scripts generated for each of these 40 examples ranged from 1 to 15, with an average of 4 scripts per example. In total, SIG generated 170 unique seed inputs for these 40 examples.

Grammar-based kernel-sensitive fuzzing effectiveness: To evaluate this, we created two variants: HeteroBugDetect$_R$ that uses random fuzzing and HeteroBugDetect$_C$ that uses grammar-based custom fuzzing without metrics. The top two rows of Figure 9 show their performance. Comparing HeteroBugDetect$_R$ to HeteroBugDetect (third row in Figure 9),

[2]https://github.com/lammps/lammps/blob/develop/potentials/ffield.eim

technique	**#Valid**	**#Invalid**	**UBC**	**BP**
HeteroBugDetect$_R$	2273.9	1460.1	0	3
HeteroBugDetect$_C$	963.9	0	2	8
HeteroBugDetect	959.8	0	2	8

Fig. 9. Ablation study results showing the effect of using grammar-based fuzzing and kernel-sensitive metrics on HeteroBugDetect's performance. The results are averaged across 10 runs.

we observe that random fuzzing generated more valid inputs (2273.9, on average) but failed to detect any unknown bugs (UBC), identifying only 3 known defects (BP) from the HeteroBugs benchmark. In contrast, HeteroBugDetect produced fewer valid inputs (959.8) but successfully uncovered 2 unknown bugs and detected 8 BP issues. This highlights that while random fuzzing increases input quantity, grammar-based, kernel-sensitive fuzzing leads to more effective and targeted bug detection. Comparing HeteroBugDetect$_C$ to HeteroBugDetect isolates the effect of kernel-sensitive metrics. As shown, both detected the same number of bugs (2 UBC and 8 BP), but HeteroBugDetect did so with slightly fewer valid inputs (959.8 vs. 963.9). This suggests that incorporating kernel-sensitive metrics improves input generation efficiency without sacrificing bug detection capability.

Each component of HeteroBugDetect enhances its performance. The SIG produced 170 diverse scripts from 40 examples. Grammar-based kernel-sensitive fuzzing detected significantly more bugs than random fuzzing.

VI. Discussion and Threats to Validity

We recognize that HeteroBugDetect is not sound or complete and can incur false positives and false negatives. This limitation is inherent to dynamic analysis approaches, which, while effective at uncovering bugs during execution, cannot guarantee exhaustive coverage or absolute correctness, especially in the context of complex, highly templated C++ codes in HPC applications such as LAMMPS where static analysis is impractical due to template instantiation occurring only at build time. False positives also depend on error thresholds. Although we adopt the thresholds used by LAMMPS developers, they can vary based on the simulation script (automatically generated by the LLM) and require domain expertise to configure appropriately. This introduces some subjectivity into bug detection.

To demonstrate practical effectiveness, we evaluate HeteroBugDetect on 20 known bugs across 7 categories. While not complete, our approach identifies a broad range of real-world heterogeneous bugs. Expanding the diversity of natural language inputs and using improved LLMs can further enhance coverage and utility, reinforcing the practicality and extensibility of HeteroBugDetect for complex HPC applications.

To mitigate threats to internal validity, we executed HeteroBugDetect ten times using 10 random seeds and have made our code and data publicly available for transparency and reproducibility. HeteroBugs, though currently limited to

20 LAMMPS bugs, is the first benchmark of heterogeneous bugs in real-world HPC scientific software. Creating such benchmarks is challenging due to the required domain knowledge and platform expertise. We acknowledge that applying HeteroBugDetect to other languages and architectures may pose challenges. To improve generalizability, we incorporated insights from developers of other HPC applications while designing HeteroBugDetect architecture. We are actively expanding HeteroBugs and plan to release it as an open-source resource to support broader community contributions.

VII. Conclusion and Future Work

We presented HeteroBugDetect, a dynamic-analysis-based approach for detecting heterogeneous bugs in complex, real-world scientific applications. By combining LLM, fuzzing, and differential testing, HeteroBugDetect overcomes the limitations of traditional methods and effectively detects diverse heterogeneous bugs. We also introduced HeteroBugs, the first dataset of heterogeneous bugs, to support reproducibility and community engagement. By evaluating HeteroBugDetect on LAMMPS, a large and widely used scientific application, we provided a rigorous and realistic assessment, and future work will extend the evaluation to other scientific applications. This work sets a foundation for improving the reliability of scientific software and advancing heterogeneous bug detection.

References

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[2] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. Piscataway, NJ, USA: IEEE (Institute of Electrical and Electronics Engineers), 2008, pp. 836–838.

[3] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 341–352.

[4] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, 2022, pp. 805–817.

[5] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Piscataway, NJ, USA, 2019, pp. 71–81.

[6] V. Kale, H. Yan, S. Mukherjee, J. Mayo, K. Teranishi, R. Rutledge, and A. Orso, "Toward automated detection of portability bugs in kokkos parallel programs," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 180–188.

[7] Q. Zhang, J. Wang, and M. Kim, "Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, New York, NY, USA, 2021, p. 242–254.

[8] J. Wang, Q. Zhang, H. Rong, G. H. Xu, and M. Kim, "Leveraging hardware probes and optimizations for accelerating fuzz testing of heterogeneous applications," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, New York, NY, USA, 2023, p. 1101–1113.

[9] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, no. 6, p. 206–215, jun 2008.

[10] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[11] F. Barreto, L. Moharkar, M. Shirodkar, V. Sarode, S. Gonsalves, and A. Johns, "Generative artificial intelligence: Opportunities and challenges of large language models," in *International Conference on Intelligent Computing and Networking*, Springer. Singapore: Springer Nature Singapore, 2023, pp. 545–553.

[12] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022.

[13] M. Motwani, "High-quality automated program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 309–314.

[14] ——, "High-quality automatic program repair," Ph.D. dissertation, University of Massachusetts Amherst, USA, Sept 2022. [Online]. Available: https://doi.org/10.7275/30288519

[15] P. Schaad, T. Schneider, T. Ben-Nun, A. Calotoiu, A. N. Ziogas, and T. Hoefler, "Fuzzyflow: Leveraging dataflow to find and squash program optimization bugs," in *ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2023.

[16] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435.

[17] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. New York, NY, USA: Association for Computing Machinery, 2024.

[18] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, New York, NY, USA, 2011, p. 416–419.

[19] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *41st International Conference on Software Engineering*, ser. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, p. 188–199.

[20] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2025. [Online]. Available: https://arxiv.org/abs/2303.18223

[21] LAMMPS Documentation Team, "Examples - lammps molecular dynamics simulator," n.d., accessed: 2025-04-20. [Online]. Available: https://docs.lammps.org/Examples.html

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, brian ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Red Hook, NY, USA: Curran Associates, Inc., 2022, pp. 24 824–24 837.

[23] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, "A systematic review of fuzzing," *Soft Comput.*, vol. 28, no. 6, pp. 5493–5522, oct 2023.

[24] D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery, "Arborx: A performance portable geometric search library," *ACM Trans. Math. Softw.*, vol. 47, no. 1, Dec. 2020.

[25] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, J. Zhou, S. Chen, T. Gui, Q. Zhang, and X. Huang, "A comprehensive capability analysis of gpt-3 and gpt-3.5 series models," 2023.

[26] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, Aug. 2020.

[27] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 2013, pp. 18–24.