Postprint of article in *Proceedings of the IEEE 44th Annual Computers, Software and Applications Conference* (*COMPSAC '20*), IEEE Computer Society (2020)

CUDAsmith: A Fuzzer for CUDA Compilers

Bo Jiang, Xiaoyan Wang
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiangbo,wxy}@buaa.edu.cn

W. K. Chan
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.cn

T. H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Na Li
New Navigation Technology Institue
CAST
Beijing, China
linacaas@163.com

Yongfeng Yin
School of Reliability and Sytems Engineering
Beihang University
Beijing, China
yyf@buaa.edu.cn

Zhenyu Zhang
Institute of Software
Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

***Abstract*—CUDA is a parallel computing platform and programming model for Graphics Processing Unit (GPU) of NVIDIA. With CUDA programming, General Purpose computing on GPU (GPGPU) is possible. However, the correctness of CUDA programs relies on the correctness of CUDA compilers, which is hard to test due to its complexity. In this work, we propose CUDAsmith, a fuzzing framework for CUDA compilers. The CUDAsmith tool can randomly generate deterministic and valid CUDA kernel code with several different strategies. Moreover, it adopts random differential testing and EMI testing techniques to solve the test oracle problems of CUDA compiler testing. In particular, we lift live code injection to CUDA compiler testing to help generate EMI variants. Our fuzzing experiments with both the NVCC compiler and the LLVM compiler for CUDA have detected thousands of failures and compiler developers have already confirmed several of them. Finally, the cost-effectiveness of CUDAsmith is also thoroughly evaluated in our fuzzing experiment.**

***Index Terms*—fuzzing, compiler, CUDA, GPGPU, EMI testing, differential testing**

I. Introduction

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for General Purpose computing on Graphical Processing Units (GPGPUs). With CUDA, developers can dramatically speed up computing applications by harnessing the power of GPUs. The CUDA programming model has successfully accelerated computation in different domains such as computational chemistry, machine learning, bioinformatics, data science, etc.

However, the correctness of the CUDA applications relies on the correctness of the underlying CUDA compiler. A bug in compiler can either lead to compile-time errors where the compiler fails to generate executable output. Or it may also lead to runtime errors to silently produce wrong executable code. In fact, developers have reported CUDA compiler bugs in CUDA developer forums [1] as well as in NVIDIA Developer website [2]. The CUDA compiler bugs may lead to build failures [3], [4], wrong code generation failures [5] and timeout failures [6], [7]. This can seriously affect the productivity of the CUDA application developers. Therefore, extensive testing of CUDA compiler implementations is crucial for the flourish of CUDA ecosystem. In this work, we are focusing on fuzz testing (i.e. fuzzing) techniques to generate random valid kernels to test CUDA compilers for failures.

However, there are two challenges for fuzzing CUDA compilers. The first challenge is to generate effective and deterministic kernel code as inputs to CUDA compilers. For a CUDA compiler, it must support the memory hierarchies, the vector data types and the synchronization primitives specified by CUDA programming model. A good fuzzing input generator must generate kernel code with those features to fully exercise the compiler under test. The second challenge is to determine the test oracle when fuzzing CUDA compiler. Indeed, the output of the compiler is binary code, whose conformance with the source code is hard to verify manually or automatically. Although differential testing techniques and EMI (Equivalence Modulo Inputs) testing techniques are reported to be effective to solve the test oracle problem for compiler testing in previous works [8] [9], there is still no work on the adaptation of these techniques into CUDA compiler testing context. The idea of EMI [9] is to take existing real-world code and transform it in a systematic way to produce different, but equivalent variants of the original code for the same input. In this way, the test oracle problem with compiler testing can be mitigated.

In previous work, Lidbury et al. [8] proposed the CLsmith tool to fuzz OpenCL compilers. CLsmith used a stochastic grammar approach to generate valid OpenCL kernels for fuzzing. Furthermore, they proposed to use differential testing and dead code mutation (a kind of EMI) to address the test oracle problem for compiler testing. At one side, OpenCL and CUDA share similar computing architectures targeted at GPU devices. At the other side, there are still nontrivial differences between their execution models. Therefore, CLSmith cannot be used for CUDA compiler testing directly. Moreover, we

The work is supported in part by the NSFC of China under Project 61772056, Project U1633125, and Project 61690202.

want to further realize live code mutation strategies (another kind of EMI) to better address the test oracle problem of compiler testing.

In this work, we propose CUDAsmith, a fuzzing framework for CUDA compilers. Due to the similarity of CUDA and OpenCL computation architecture, we choose to adapt the kernel code generation logic of CLsmith to CUDA execution model. The CUDAsmith tool can randomly generate deterministic and valid CUDA kernel code based with a scholastic grammar approach. To solve the test oracle problem of compiler testing, in addition to the differential testing and dead code mutation techniques, CUDAsmith tool also lifts the live code injection mechanisms into CUDA compiler testing context. Moreover, we have performed a large scale fuzzing on several versions of NVCC and Clang compilers for CUDA with different optimization levels. The experimental results showed that CUDAsmith were effective to detect compiler errors based on random differential testing. Furthermore, we have also evaluated CUDAsmith in EMI testing mode, which is also effective to expose CUDA compiler bugs.

The contributions of this work are three fold. First, we have proposed an effective fuzzing tool for CUDA compilers by generating deterministic and valid CUDA kernel code. Second, we have enabled live code injection techniques to perform EMI testing on CUDA compilers. Third, we have performed the first comprehensive fuzzing campaign on CUDA compilers with different optimization levels, our CUDAsmith tool has found thousands of CUDA compiler failures in both NVCC and Clang. And the NVCC and Clang compiler developers have already confirmed several of our reported bugs.

The organization of the remaining sections is as follows. In Section 2, we will present our tool CUDAsmith in detail, including both its input generation strategies and test oracle checking strategies. In section 3, we will perform a comprehensive experimental study to evaluate the effectiveness of CUDAsmith in detecting compiler failures. Then we will analyze some NVCC and Clang compiler bugs confirmed by developers in section 4 followed by the related work in Section 5. Finally, we conclude our work in Section 6.

II. CUDAsmith: The CUDA Compiler Fuzzer

In this section, we first present the general workflow of our CUDAsmith tool. Then we will present its kernel function generation strategies, its differential testing strategy, and its EMI variants generation strategies.

A. The general workflow of CUDAsmith

The general workflow of CUDAsmith is shown in Figure 1. At first, the CUDA kernel generator of CUDAsmith tool will randomly generate a pool of CUDA kernel functions. CUDAsmith can generate CUDA kernels in different mode with different language features. The default configuration of CUDAsmith is to use the All mode to generate kernels with the most comprehensive features. Some of these generated kernel functions will be fed into the EMI variants generator

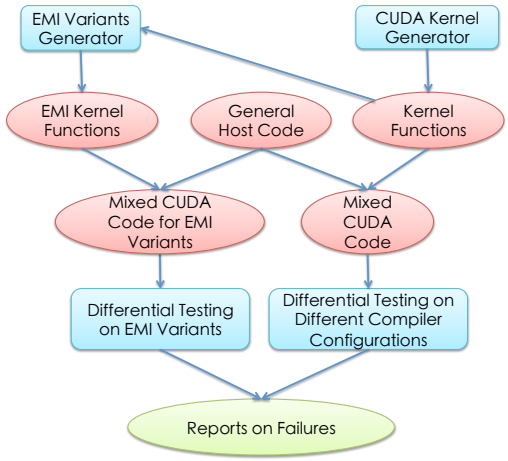

Fig. 1. Overview of CUDAsmith Workflow

to generate EMI variants of the kernel function with dead-code injection and live-code injection. Both the CUDA kernel functions and the EMI kernel function variants are respectively merged with a general host code to generate a mixed host and device code ready for compilation. Then CUDAsmith will use the mixed CUDA code to perform differential testing on different compiler configurations (i.e., different compiler versions and optimization options). Furthermore, CUDAsmith will also use the mixed CUDA code of EMI variants to perform EMI testing. Finally, CUDAsmith will analyze the fuzzing logs to report build failures, wrong code failures, and timeout failures. In summary, CUDAsmith combines CUDA program generation techniques, differential testing techniques, and EMI testing techniques to address the CUDA compiler testing problem.

B. The Host Code of CUDAsmith

As shown in the Table I below, the host code of CUDAsmith follows the basic design pattern of the host-side code of CUDA. It first parse of arguments for launching the kernel (line 1). Then it queries the attributes of CUDA device, its capabilities and memory size (line 2). After that, it creates the CUDA context and checks whether the number of threads and the block size are within the maximum limit of the device (line 3 and 4). Then, it allocate and initializes host and device memory based on the mode of kernel generation (line 5). At line 5 to 7, the kernel launch performs its main task: copy data from host to device, invoke the kernel function, and copy data from device to host. Finally, the launcher calculates the results and releases the memory.

Within this workflow, the API functions used by CUDAsmith and CLsmith are syntactically different. We have listed the mappings of the key API functions between CUDA and OpenCL used by host code as shown in Table II. The API functions in Table II involves device information query, memory operations, and kernel launch functions. In particular, the memory manipulation functions and kernel launch functions must be adapted to CUDA specification. Moreover, the

TABLE I
THE WORKFLOW OF HOST CODE

1	Parse arguments.
2	Query device, capability and memory.
3	Create Context.
4	Check number of threads and block number.
5	Allocate host and device memory for different mode.
6	CuMemcpyHtoD(...) //copy data from host to device
7	Kernel<<< ... >>> () //invoke kernel function
8	CuMemcpyDtoH(...) //copy data from device to host
9	Calculate results.
10	Release Memory.

TABLE II
THE MAPPING OF MAJOR API FUNCTIONS IN HOST CODE

CUDA	OpenCL
cudaGetDeviceProperties()	clGetDeviceInfo()
cudaMalloc()	clCreateBuffer()
cudaMemcpy()	clEnqueueRead(Write)Buffer()
cudaFree()	clReleaseMemObj()
Kernel<<< ... >>> ()	clEnqueueNDRangeKernel()

TABLE III
MAPPING OF EXECUTION MODEL

CUDA	OpenCL
Grid	NDRange
Thread Block	Work Group
Thread	Work Item
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
blockIdx*blockDim+threadIdx	get_global_id()
gridDim*blockDim	get_global_size()

TABLE IV
MAPPING OF MEMORY MODEL

CUDA	OpenCL
Host memory	Host memory
Global or Device memory	Global memory
Local memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Registers	Private memory

CUDAsmith tool also provides different options when running different types of kernels. In the host code, the main function must also parse the options specified by the users to launch the kernel accordingly.

C. The CUDA kernel function generator

In this section, we will present the technical details of the CUDA kernel function generator. In general, based on the syntactic characteristics of the kernel functions generated, the CUDA kernel function generator can work in different modes. More specifically, CUDAsmith can generate kernels in basic mode, vector mode, barrier mode, atomic mode, and the all mode. The design of different kernel generation modes follows that of the CLsmith [8] but adapted for CUDA context. The basic and the vector mode both generate embarrassingly parallel random kernels. The difference is that the vector mode will generate and use vector data types within the kernel. The atomic and barrier modes will generate communicating yet deterministic kernels. Finally, the all mode is the combination of the features of all different modes. Each mode will enable different syntactic characteristics in the generated kernels, which we will present below in details.

1) Basic mode: Since the execution model of CUDAsmith and CLsmith is similar, we follow the workflow of the basic mode of CLsmith to build the basic mode of CUDAsmith. In basic mode, each thread independently executes the same logic in the kernel and writes its own result (aggregated with CRC calculation) into an array called *result* with its own linear global id as index. The *result* array holds the final result after kernel execution. Therefore, the basic mode represents the most simple form of data-parallel computing problems.

On the other side, the execution model of CUDA and OpenCL also has many differences. Fortunately, there is a mapping between them which we summarizes in Table III. For example, the hierarchy of computing unit of CUDA includes Grid, Thread Block, and Thread, which corresponds to NDRange, Work Group, and Work Item in OpenCL, respectively. Similarly, the variables describing the size and indices of computing unit in CUDA also have their counterparts in OpenCL. So we borrow the workflow of the basic mode of CLsmith, and then we build the basic mode of CUDAsmith by referencing the syntax mapping between CUDA and OpenCL.

Another major difference between CUDA and OpenCL is their memory model. However, because the memory hierarchies of CUDA and OpenCL are both abstractions of the same set of memory resources (i.e., host memory, graphics memory, registers, etc), there is also a mapping between CUDA and OpenCL in terms of memory model as shown in Table IV. Based on this memory mapping, we manage to adapt each memory related kernel code generation in CLsmith into CUDA coding convention. For example, kernels are decorated with __global__ in CUDA and with __kernel in OpenCL. And for functions containing device code, CUDA uses __device__ modifier while OpenCL uses no explicit modifier.

After carefully handling the execution model and the memory model mappings between OpenCL and CUDA (as well as some other small syntactic differences), CUDAsmith is able to generate kernels in basic mode conforming to CUDA programming paradigm.

2) Vector mode: For data-parallel problems, vectors data types with different dimensions (1D, 2D, and 3D) are frequently used by CUDA programs for parallel processing on computation units. Therefore, it is important to enhance the basic mode with vector data types and operations support. So in vector mode, CUDAsmith aims at generating kernels with valid vector data types and corresponding operations.

There are several differences between OpenCL and CUDA in vector support. OpenCL supports signed and unsigned char, short, int and long vectors that can be declared with length 2, 3, 4, 8 and 16. While CUDA also provides a similar set of vector data types, the supported vector length only ranges from 1 to 4. Another difference between CUDA and

OpenCL vector data type is their constructor. CUDA provides a constructor function of the form $make\_\langle typename \rangle$. For example, "$make\_int2(int\ x, int\ y)$" will creates a vector of type int2 with $value(x, y)$. For OpenCL kernels, the constructor is usually in the form of vector literals such as $int2(int\ x, int\ y)$. So when adapting the kernel generation logic of vector mode from CLsmith, we must not only control the length of the vector data types for CUDA, but we also need to change the vector constructor statements from OpenCL style to CUDA style.

Moreover, OpenCL also supports built-in operators and functions for vector datatype while CUDA has no such support. So we must also handle appropriately in the generated CUDA kernel in vector mode. Finally, to avoid undefined behaviors arising from vector computations, we still reused the well-defined "safe math" vector macros realized within CLsmith in CUDAsmith.

3) Barrier mode: The basic mode and the vector mode only generate the embarrassingly parallel kernels. But in reality, different threads often needs to communicate or synchronize after each iteration of computation. The CUDAsmith in barrier and atomic mode tries to generate such communicating yet deterministic code.

The basic pattern for barrier mode is simple: the kernel uses a barrier function (i.e., __synchthreads) just before redistributing the ownership of elements within global or shared array among threads in a block. The barrier primitive can ensure race-freedom and deterministic result. An A_offset value is used by the thread to index into a shared array A for computation. And a 2D array called *permutations* provides each thread its unique A_offset value identified by its linear local id and a random number. The generated code for the *i*th such synchronization point is as follows:

```
__syncthreads();
A_offset=permutations[rnd_i][linear_local_id()];
```

In this way, each time after thread synchronization, a thread get a new A_offset with its own linear thread id and a new random number for the *i*th round. The barrier primitive ensures race freedom upon offset redistribution.

OpenCL uses barrier primitive to synchronize work-items within a work-group. Similarly, CUDA also uses barrier primitive to synchronize threads within a thread block. The differences is that CUDA uses $\_\_syncthreads()$ while OpenCL uses $barrier()$ as the barrier primitive. The $barrier()$ primitive in OpenCL has arguments specifying the memory address space to perform synchronization. For CUDA, the $\_\_syncthreads()$ has no arguments, but it works for both global and shared variables. So CUDAsmith uses $\_\_syncthreads()$ to replace $barrier(cl\_mem\_fence\_flags\ flags)$ for thread synchronization and declares both global and shared variables to synchronize on.

4) Atomic mode: The atomic functions are important instruments for threads to synchronize with each other for many-core computing. So the atomic mode of CUDAsmith aims at generating communicating yet deterministic kernels with those atomic functions. Both OpenCL and CUDA provide a rich set of atomic functions. The mapping between them are shown in Table V. Each pair of atomic functions in OpenCL and CUDA provides similar atomic operations, the difference mainly lies in their function signature.

TABLE V
MAPPING OF ATOMIC FUNCTIONS

CUDA	OpenCL
atomicAdd	atomic_add
atomicSub	atomic_sub
atomicCAS	atomic_cmxchg
atomicMin	atomic_min
atomicMax	atomic_max
atomicInc	atomic_inc
atomicDec	atomic_dec
atomicAnd	atomic_and
atomicOr	atomic_or
atomicXor	atomic_xor

In particular, the atomic functions in OpenCL explicitly differentiate the global and local variables. While the atomic functions for CUDA are applicable to both global and shared variable in CUDA. Therefore, when generating atomic function code in CUDAsmith, we must take special care of the variable types.

Within the kernel generated by CUDAsmith, we use shared variables as communication vehicles across threads. To avoid unintended compiler optimizations of shared variable into registers (accessible by one thread only), we must declare these shared variable as volatile. However, these atomic functions cannot directly accept pointers of volatile shared variables as arguments. Therefore, we choose to write wrapper functions to perform typecasts on pointers of volatile shared variables for those atomic functions.

Similar to CLsmith, the atomic mode of CUDAsmith has two sub-modes: the atomic section mode and the atomic reduction mode, which we describe in detail as follows.

The workflow of the code generated in the atomic section mode is similar to that of CLsmith [8]. Basically, CUDAsmith will randomly insert several atomic sections within the kernel code. The *i*th atomic section is shown in the code snippet below. The rnd_i is a random literal value for atomic section *i*. And *input* is a shared volatile *uint* value. One and only one thread within a block will execute the atomic section guarded by the conditional containing $atomicInc$ based on thread schedule. Within the atomic section, the thread will execute the *statements* and store the result in the variable *result*. And the *result* will be added to another shared variable *s* of type volatile *uint*. After the atomic sections, one of the threads will finally output the result on behalf of the thread block. Furthermore, different thread block will use different variables such that they are independent from each other. So whatever the thread schedule is, the results of the kernel execution is deterministic.

```
if(atomicInc(input) == rnd_i){
    /*statements*/
    atomicAdd(s, result);
}
```

For atomic reduction mode, the threads within a thread block first perform a atomic reduction on a volatile shared variable *v* and expression *i* using one of the commutative and associative atomic operations supported by CUDA: add, min, max, or, and xor. Then the threads synchronize via barrier primitives and one of the threads with local id equal to 0 calculates the sum for all threads. Due to the commutability and associativity of atomic operations, the relative order of the threads execution will not affect the final result. So the results is still deterministic.

```
atomicOp_i(&v, expr_i);
__syncthreads();
if(get_linear_local_id()==0) {sum+=v;}
__syncthreads();
```

5) All mode: Finally, the all mode of CUDAsmith is the combinations of all previous modes for kernel generation. Therefore, in all mode, we can generate a deterministic kernel with most of the CUDA programming features enabled with a certain probability. Compared with other modes, the kernel generated by all mode has a higher probability to cover compiler code and trigger compiler failures. Note that the occurrence of different program features are controlled by probabilities configurable within CUDAsmith. In this way, CUDAsmith can be configured to generate kernel with highlights on different features. It would be interesting to study the impact of different configurations on the fault detection ability of the generated kernels in future work.

D. EMI testing with CUDAsmith

To solve the test oracle problem, we further enabled EMI testing techniques with CUDAsmith. To generate EMI variants, we inject code into a kernel function generated by CUDAsmith with two strategies. One strategy is to inject always false conditional block (FCB, i.e., dead code mutation) and the other strategy is to inject always-true guard (TG, i.e., live code mutation). The dead code mutation is borrowed from CLsmith while the live code mutation is newly supported in this work.

1) Injecting always False Conditional Block (FCB): To generate always False Conditional Block, we follow the strategy realized in CLsmith [9] to inject dead code into existing kernels rather than prune existing dead code from the kernel. This is because recording CUDA kernel coverage is hard and dead code is rare in real world kernel.

For an initial kernel, CUDAsmith first randomly generates and injects a set of EMI-FCB blocks into it. And then it prunes the EMI-FCB blocks according to a set of probabilities to produce variants of the kernel. This strategy is also used in previous works in EMI testing [8], [9]. We follows the pruning strategies of [8] to perform the pruning.

To construct a always false condition block, CUDAsmith equips a kernel with an additional array parameter called *dead* and randomly inserts into the kernel a number of EMI-FCB blocks, where the *ith* FCB block to be generated has the following form:

```
if(dead[rnd_{i,1}] < dead[rnd_{i,2}]){
 /* Any statements */
}
```

The runtime values of elements of *dead* are initialized in the host application so that $dead[j] = j$, which is unknown to the CUDA compiler. The predicate of the *if* statement is designed to be false: where $rnd_{i,1}$ and $rnd_{i,2}$ are selected randomly during program generation to ensure $rnd_{i,1} > rnd_{i,2}$. In this way, CUDAsmith can ensure the statements within the EMI block are dynamically unreachable.

2) Injecting always True Guard (TG): To inject always True Guard (TG) into an existing kernel, CUDAsmith follows one of the strategy proposed by [10]. For an existing executed statement *s* in the original program, CUDAsmith introduces an *if* statement to guard *s*, of which the predicate *p* is always true, (i.e., $if(p)\ s;$). This strategy injects live code while still preserving the original semantics. The construction of predicate *p* is the similar to the idea of EMI-FCB but with an array called *live*:

And the array *live* is also initialized in host code so that live[j] = j. The only difference between FCB and TG is that $rnd_{i,1}$ and $rnd_{i,2}$ are selected randomly during program generation to ensure $rnd_{i,1} < rnd_{i,2}$. In this way, CUDAsmith can ensure the *if* statements inserted is always true. Different form FCB, to ensure simplicity, CUDAsmith will generate only one TG variant for each original kernel. To ensure the validity of the kernel code after live code injection, when choosing the statement *s* to add always true *if* statement, CUDAsmith tries to avoid choosing conditional statements as *s*.

```
if(live[rnd_{i,1}] < live[rnd_{i,2}]){
    s; /*s is an executed statement*/
}
```

Finally, for both dead code injection and live code injection, we follow [8] to perform filtering on base kernels to avoid injection on dead code. Basically, we invert the values of *dead* or *live* array and check whether the execution results of kernel is affected to perform filtering. This can help filter out many ineffective EMI base kernels.

III. Experiment and Results Analysis

In this section, we present the details of our experiment as well as the results analysis.

A. Research Questions

- RQ1: Is differential testing effective to detect CUDA compiler bugs?
- RQ2: Are EMI techniques effective to detect CUDA compiler bugs?
- RQ3: Which activity in fuzzing consumes most of the time during compiler testing?

For RQ1, we want to evaluate the effectiveness of differential testing techniques on detecting CUDA compiler bugs. For RQ2, we want to evaluate the fault detection ability of the two EMI techniques: dead-code injection and live-code

TABLE VI
COMPILER CONFIGURATIONS IN DIFFERENTIAL TESTING

Compiler	Version	Opt. Options	Total Config.
NVCC	8.0	O0, O1, O2, O3	12
	9.0	O0, O1, O2, O3	12
	9.2	O0, O1, O2, O3	12
Clang	6.0	O0, O1, O2, O3	12
	7.0(trunk)	O0, O1, O2, O3	8

injection. Finally, the compiler testing process involves test case generation, invalid kernel filtering (for EMI), compilation, and execution. For RQ3, we want to understand which of these activities consumes most of the time.

B. Experiment setup

We performed our compiler fuzzing experiment on two workstations and one desktop. Both workstations are equipped with Intel(R) Xeon(R) CPU E5-2620 and Quadro K2200 while the desktop is equipped with Intel Core i7-6700 and GeForce GTX 1060. For operating system, we used Ubuntu 16.04.2 LTS on the workstations and the desktop. We also installed different versions of NVCC and Clang compilers on these machines for differential testing.

C. Experiment procedure and compiler configurations

When performing differential testing, we combine compiler versions with compiler optimization options to form the basic compilation configurations for comparison. As shown in Table VI, we use NVCC and Clang compilers in differential testing, each with different versions and different optimization options. For NVCC, we use its official released version 8.0, 9.1 and 9.2. Moreover, there are 4 optimization levels for ptxas, its PTX optimization assembler. For Clang, we use its official version 6.0 and the current trunk version (which is the 7.0 version to release). For Clang, the optimization of kernel code is mainly performed by the LLVM IR optimizer [11] and the corresponding optimization options is integrated with the optimization options of Clang. When combined these versions and options together, we have 12 compiler configurations for NVCC and 8 compiler configurations for Clang. So for each kernel generated, we have 20 compilation configurations used for differential testing in total.

For the FCB mode of EMI testing, we used the EMI module to generate 40 EMI variants (kernels) for each valid base kernel generated by CUDAsmith in ALL mode. In total, we generated 1726 valid kernels with CUDAsmith after filtering invalid ones, and finally had 69040 EMI variants (kernels) for testing. When performing EMI testing on each group of 40 EMI variants, we used optimization level O0 for NVCC 9.2 and optimization level O3 for Clang.

For the TG mode of EMI testing, CUDAsmith only generates one TG variant for each kernel generated by all mode. So we generated 76111 valid base kernel for TG after filtering. For each of the 76111 base kernel, we generated a TG variant for it to conduct EMI testing. For NVCC 9.2, we used optimization level O0. And for the Clang trunk version we used optimization level O3.

When performing fuzzing, we wanted to use the same input source files for NVCC and Clang compilers. NVCC supports both whole program compilation and separate compilation [12] while Clang only supports the compilation of the whole program containing mixed host and kernel code [11]. So we built a general host code and merged it with the kernels generated by CUDAsmith to form a whole program as the uniform input to the compilers under test.

For each compiler configurations or EMI variant, we first compiled the mixed CUDA code and compared their compilations results to detect any build errors at compilation time. Then we executed the generated binary files and compared their outputs. If there were discrepancies among their outputs, we had successfully triggered a wrong code failure in the compiler under test. Since the kernels generated by CUDAsmith normally took only a few seconds to execute, we set a timeout of one minute for execution. If the execution of a kernel took more than one minute, we marked the kernel as triggering a timeout failure. After that, we manually inspected and re-executed the same test case to confirm the failure. Finally, we reduced the failure-triggering kernel for reporting to NVCC or Clang compiler developers.

D. Answering RQ1

In this section, we first summarize the results for differential testing. The differential testing results for the NVCC compiler and the Clang compiler are shown in Table VII and Table VIII, respectively.

As shown in Table VII, the first three columns show the compiler configurations including the compiler under test, the compiler version and the optimization level used during fuzzing. While the last four columns show the number of build failures, the number of wrong code failures, the number of timeout failures, and the total number of valid cases for each configuration. When generating test cases, the total number of test cases were the same for different configurations. However, a small amount of the test cases (less than 0.5% on average) were syntactically invalid and removed after compilation. The last row shows the average number of build failure, wrong code, timeout failure and valid test cases over all configurations. We can find that on average, the majority of failures detected by differential testing are of type wrong code, followed by timeout failures. And no build failures are detected on NVCC compilers. The average percentage of wrong code and timeout failures are 6.1% and 0.3% on NVCC compiler. In general, the results show that the wrong code failures are most prevalent in NVCC compilers. Furthermore, the timeout failure is also an important problem to pay attention to since it may seriously affect the performance of the CUDA application.

When comparing different compiler versions, there were more wrong code failures for NVCC 8.0 and NVCC 9.2 than for NVCC 9.0. In contrast, the number of timeout failures exposed in NVCC 9.0 and NVCC 9.2 are much higher than NVCC 8.0. Therefore, it seems different compiler versions have different distributions of failure types. It seems that the NVCC version 9.0 is relatively more stable among the

TABLE VII
RESULTS FOR DIFFERENTIAL TESTING ON NVCC COMPILER

Version	Opt. Level	Build Failure	Wrong Code	Timeout	Total
8.0	O0	0	1583	11	19429
	O1	0	1583	11	19430
	O2	0	1576	11	19430
	O3	0	1576	11	19430
9.0	O0	0	579	88	19473
	O1	0	885	95	19473
	O2	0	153	96	19473
	O3	0	161	95	19473
9.2	O0	0	1602	87	19940
	O1	0	1598	87	19922
	O2	0	1599	87	19924
	O3	0	1602	87	19924
Average	/	0	1208	64	19610

TABLE VIII
RESULTS FOR DIFFERENTIAL TESTING ON CLANG COMPILER

Version	Optimization Level	Build Failure	Wrong Code	Timeout	Total
6.0	O0	1	595	67	19940
	O1	1	519	523	19922
	O2	1	500	85	19924
	O3	1	497	85	19924
trunk (7.0)	O0	0	1930	118	19941
	O1	1	1702	138	19920
	O2	1	1603	99	19729
	O3	1	1621	82	19925
Average	/	1	1120	149	19903

three versions under test. And the version 9.2 of NVCC seems to have introduced some new bugs during its feature enhancement.

When we focus on the optimization options, within each compiler version, different optimization options in general have similar number of failures for a specific failure type. For example, for NVCC 9.2, the four optimization levels (O0, O1, O2, O3) have 1602, 1598, 1599, 1602 wrong code failures, which are quite close to each other. There is an exception for NVCC 9.0, where optimization O0 and O1 have much more wrong code failures than O2 and O3.

As shown in Table VIII, the meanings of the different columns for Clang is the same as that in Table VII. On average, 1120 wrong code failures, 149 timeout failures, and 1 build error are exposed on Clang compilers for each compiler configuration. The average percentage of wrong code and timeout failures are 5.6% and 0.7% for Clang compiler, respectively. The percentage of wrong code failure of Clang is close to that of NVCC but the timeout failure is doubled.

When comparing different compiler versions, the trunk version in general has much more wrong code failures than Clang 6.0. This results is consistent with the common development scenarios: the trunk version usually introduces more bugs while adding new features. This is also true for timeout bugs except for the configuration Clang 6.0 with optimization level O1, which has much more timeouts than any other configuration.

When we focus on the optimization options, within each compiler version, different optimization levels in general also have similar number of failures for a specific failure type. For example, for Clang trunk version, the four optimization levels (O0, O1, O2, O3) have 1930, 1702, 1603, 1621 wrong code failures, which are quite close. For timeout failures, the Clang 6.0 with optimization level O0, O2, and O3 have similar numbers, with the exception of O1. For the trunk version of Clang, the optimization level O0 and O1 have higher number of timeouts than the other two optimization level, however, the difference is small.

Since the number of valid test cases for Clang and NVCC are close to each other (19903 v.s. 19610), we proceed to compare them in terms of the number failures exposed. On average, CUDAsmith exposed similar number of wrong code failures in NVCC (1208) and Clang (1120). Furthermore, the number of build failures exposed on the two compilers are both very small. However, CUDAsmith exposed 149 timeout failures on Clang, which is more than two times the number of timeout failures (64) exposed on NVCC.

To summarize, the differential testing mode of CUDAsmith can detect a significant number wrong code failures and timeout failures on both Clang and NVCC compilers. Therefore, we can answer RQ1 that differential testing is an effective way to trigger CUDA compiler failures.

E. Answering RQ2

In this section, we will evaluate the effectiveness of EMI testing on detecting CUDA compiler bugs. For EMI testing, we have two modes, i.e., FCB and TG. To control the scale of fuzzing, we have to fix the compiler configurations to specific setting. For each mode, we perform fuzzing with both the trunk-version of Clang (Clang 7.0.7) as well as NVCC version 9.2 (i.e., the newest release version of CUDA Tookit at the time of experiment). Furthermore, We use the optimization level of O3 for Clang and optimization level O0 for NVCC, respectively.

The EMI fuzzing results are shown in Table IX. The first two columns show the Compiler and EMI testing modes while the following four columns have the same meaning as the table for differential testing. When comparing TG mode and FCB mode, we can find that FCB mode can expose more failures than TG mode on both Clang and CUDA. The last row shows the average number of failure exposed over all compiler and EMI configuration. We can see that on average, EMI detected 19 build failure, 14 wrong code failure, and 17 timeout failure with around 72377 test cases for each compiler version and EMI mode configuration. At one side, the failure detection rate of EMI testing is relatively small when compared with differential testing. At the other side, the EMI testing approach can detect different failures from differential testing. Indeed, EMI detects much more build failures than differential testing. And a manual analysis on the wrong code failures and timeout failures detected by EMI and differential testing shows that the most of the bugs found are also different.

To summarize, the EMI testing mode of CUDAsmith is also effective to detect different types of CUDA compiler bugs.

TABLE IX
RESULTS FOR EMI TESTING

Compiler	Mode	Build Failure	Wrong Code	Timeout	Total
CUDA	TG	0	13	17	76111
Clang	TG	3	5	9	71877
CUDA	FCB	0	27	32	69040
Clang	FCB	72	16	12	72480
Average	/	19	15	20	72377

And the differential testing mode and EMI testing mode of CUDAsmith are complement to each other in terms of failure detection.

F. Answering RQ3

In this section, we want to understand which activity during the fuzzing process consumes most of the time. Since the compilation and execution time are highly dependent on the hardware configuration, we will show the time cost of each activity on two hardware configurations separately. One hardware configuration is our workstation, which is equipped with Intel(R) Xeon(R) CPU E5-2620 and GPU model Quadro K2200. The other is a desktop equipped with Intel Core i7-6700 and GeForce GTX 1060. For each hardware configuration, we measured the mean time of test case generation, test case compilation, test case execution, and test case filtering for 10000 test cases. During our fuzzing process, we set a one minute timeout for the execution of one test case. In another word, we will stop the execution process when it takes more than one minute.

The time cost of different fuzzing activity is shown in Figure 2. The y-axis shows different activities of fuzzing on the two hardware configurations named with their GPU models, respectively. The x-axis shows the time cost of different activities for fuzzing 10000 test cases in minutes. In general, we can see that on both GPU model, the time costs for test case generation are both very small. Furthermore, the time costs for compilation and execution are both much higher than test case generation. Finally, the filtering cost of EMI FCB mode is small while the filtering cost of EMI TG mode is relative large. This is because, for the EMI FCB mode, we only need to verify the base kernel to check the validity of all kernels generated. Since we generate 40 kernels for each base kernel, the cost of filtering for FCB mode is approximately 1/20 of the time cost of executing all the kernels. For EMI TG mode, each base kernel will only generate two kernels. Therefore, its filtering cost is close to the execution cost, which is a little bit high.

For the desktop with GeFore GTX 1060, the compilation cost is much higher than the execution or filtering cost. In contrast, for the desktop with Quadro K2200, the compilation cost is smaller than execution or filtering cost. This difference results from the difference in their computation abilities.

On our workstation, it roughly takes 650 minutes to finish the fuzzing of 10,000 test cases on CUDA compiler in differential testing (note filtering is not needed). Combining

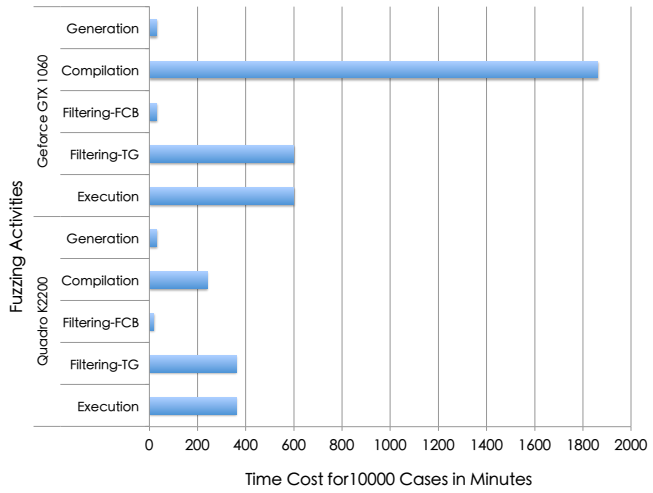

Fig. 2. Time Cost of Different Fuzzing Activities

the 6.1% and 0.3% failure rate for wrong code failures and timeout failures in differential testing, about 9 wrong code failures and 0.46 timeout failures can be exposed in every 10 minutes of fuzzing by CUDAsmith on average, which is cost-effective.

However, compared with the time to perform fuzzing, reducing the kernel to reproduce the failure (i.e., the first step in debugging) is in fact the most time-consuming activity. The kernels generated by CUDAsmith are generally large in size and complex to understand. To make it possible for developer to debug the compiler bugs, it is crucial to reduce the failure-triggering kernel as much as possible while ensuring to reproduce the same failure. However, there is still no practical tool to do this. We have tried the Berkeley delta tool to reduce the kernel. However, the tool is hard to terminate during reduction due to the complexity of the kernel generated. Therefore, we have to manually reduce the kernels for reporting. Based on our experiences, it may takes more than 14 hours to reduce a single kernel manually, which is tedious and frustrating. Therefore, we consider it is crucial to build an effective reducer for failure-triggering CUDA kernels, which we left as a future work.

To sum up, the time cost of the CUDAsmith tool is reasonable on our workstation. Considering its failure-triggering ability, the CUDAsmith tool is in general cost-effective for practical use on compiler testing. On the other hand, an effective test case reduction tool for reducing the failure triggering kernels is desired for debugging.

IV. ANALYSIS OF COMPILER BUGS DETECTED

In this section, we will analyze some of the confirmed compiler bugs by NVCC or Clang developers detected by CUDAsmith.[1]

[1]The links to the confirmed compiler bugs on the Web are removed for double-blind review.

A. Wrong code bug with vector data type

During differential testing, we have identified one wrong code bug related to logical operations on vector data type on CUDA toolkit version 9.0 and 9.1 when performing differential testing. The results on CUDA toolkit version 8.0 and on Clang were the correct. With non-trivial manual effort, we managed to reduce the failure-triggering test case into a small code snippet for filing bugs to CUDA development community. The NVIDIA developers later confirmed the bug. And they planned to fix the bug in future releases.

The reduced failure triggering code snippet is shown in the listing below. At line 1, the variable val is assigned with a value. Then at line 2, we build a 2D vector $< 10, 2 >$ with $make\_int2$ and perform OR operation with val. The expected value for a is 1 since y is 2. However, during our fuzzing, the value of a is 2. At line 3, we build another 2D vector $< 10, 0 >$ with make_int2 and perform OR operation with val. Since the y component of the vector is 0, the value of b should be dependent on val. However, the value of b is always 0 when the code is compiled with NVCC version 9.0 and 9.1. We have also tested 4D vectors with make_int4 with the same operation, and similar bugs also manifested on the y, z, w component of the 4D vector. The workaround provided by NVIDIA experts was to first store the vector value into a non-const temporary variable and then perform logical operation on the variable.

```
1. int val = SOME_VAL;
2. int a = make_int2(10,2).y || val;
   //the expected value of a is 1.
   //However, the actual value is 2.
3. int b = make_int2(10,0).y || val;
   //The value of b is always 0.
   //However, when val is 1,
   //b is expected to be 1.
```

B. Wrong code with ptxas tool within NVCC 8.0

We also identified a bug with ptxas tool within NVCC 8.0. When performing differential testing on Clang compiler front end, CUDAsmith triggered a wrong code bug with ptxas at optimization level "-O1". Note the Clang compiler just provides the front end to compile CUDA source code into PTX intermediate code, while the PTX intermediate code is in turn translated and optimized by the ptxas tool provided in NVCC toolchain into the final binary code. So the whole compilation process is the joint effort of Clang front-end and ptxas back end provided CUDA toolkit.

Initially, we thought it is a bug in Clang front end related to optimization. But it turns out to be a bug with the ptxas tool provided by CUDA Toolkit version 8.0. If we update the CUDA Toolkit to version 9.0 or 9.1, the bug will disappear. Both NVIDIA and LLVM engineers confirmed the bug.

As discussed by LLVM engineers, the problem is with the ptxas tool in CUDA-8. The PTX code generated by Clang is identical for both CUDA-8 and CUDA-9. However, the SASS generated by ptxas tool from CUDA-8 and CUDA-9 is significantly different. With CUDA-9 the code is straightforward and there are two writes, 8 bytes apart, both with the same value. With CUDA-8, func3 messes up the store to *l_302 and writes zero to the should_not_change field instead.

C. Wrong code with printf output in NVCC

We also identified a wrong code bug related to the debug flag and printf option in NVCC version 9.2. When performing reduction on a kernel, we found that the debug flag (the "-G" option) or the printf statement can independently affect the result when inspecting or outputting the value of one variable. The kernel function itself has no illegal operations, however, the result is always wrong. But once we begin to observe (with debug option) or output the value of a variable (with printf), the results will be correct. This bug can make the life of developers very hard as it may seriously affect their judgement during debugging and coding. We have reported this bug to NVCC engineers. They confirmed this bug and planned to fix it in future releases.

D. Timeout Failures of NVCC Compiler

During our fuzzing, we also identify many timeout failures in NVCC and Clang compilers. In particular, our EMI-TG testing mode triggered one timeout failure on both version 9.2 and 10.0 of NVCC compiler with GeFore 940MX. The kernel with an always true condition (always true guard) inserted should have generated the same result as the original kernel. However, the kernel with always true guard returns an error "CUDA_ERROR_LAUNCH_TIMEOUT = 702". Based on the CUDA_Driver_API documentation, this means that the kernel took tool long to execute. In contrast, the original kernel executes normally without exceeding time limit. We manually reduced the kernel and filed the bug to the NVCC compiler developers. The NVCC developers have no GeFore 940MX at hand, so they try reproducing the case with both GeForce GTX 980 and GeForce GTX 1080 Ti. Although they did not get the "CUDA_ERROR_LAUNCH_TIMEOUT = 702" error message, they still got a surprising result on performance: the execution time on the more powerful GeFore GTX 1080Ti is significantly slower than GeFore GTX 980. And the execution time also exceeded our one minutes timeout criteria. Therefore, the developers also confirmed our bug report on this timeout issue. Considering the large number timeout failures exposed in our experiment, we believe the performance issues with CUDA compilers is also an important problem to solve for compiler developers in the future.

V. Related Works

In this section, we briefly review related works on compiler testing and the test oracle problem of software testing.

Csmith [13], [14] is a well-known testing tool for C compiler. Csmtih randomly generates deterministic C programs as test cases containing complex code that covers a large subset of C while avoiding the undefined and unspecified behaviors. Using this tool, The Csmith found more than 325 bugs in mainstream Compilers including GCC, LLVM and commercial tools. In [15], Regehr et al. further proposed test case reduction

techniques to further reduce the failure triggering test cases such that the following up debugging activities could be made easier.

Lidbury et al. [8] proposed CLSmtih to find OpenCL compiler bugs. Based on Csmith, CLsmith can randomly generate deterministic, communicating, and feature-rich OpenCL kernels. CLsmith used both random differential testing and EMI testing to handle the test oracle problem. In contrast, our CUDAsmith tool is adapted and enhanced based on the CLsmith tool in many aspects to handle the CUDA programming model. CUDAsmith also introduces live-code mutation to handle the test oracle problem.

The random differential testing has proved successful at hunting compiler bugs. A deterministic program should always have a unique and well-defined result so that random differential testing can circumvent the test oracle problem with majority voting. The random testing has been widely used in the domain of compiler testing, e.g. C compiler [14], C++ compiler [16], JavaScript and PHP interpreter [17]. Eide [18] proposed a tool called *randprog*, which was a significantly enhanced version of a program generator written by Brian Turner [19]. The *randprog* tool can detect the miscompilations of volatiles via generating random c programs that contain volatile variables. In [20], Chen et al. proposed an approach to differential testing JVM implementations. They adopted mutation testing strategy and code coverage information to guide the class files selection process. Finally, the selected class files are used as inputs to differential testing.

Equivalence Modulo Inputs (EMI) is a recent promising approach for compiler validation proposed by Le et al. [9]. They have developed many tools such as Orion [9], Athena [21] using this approach to find bugs in compilers. Both Orion and Athena relied on deleting code from or inserting code into code regions which are not executed under the inputs. Then they further proposed a novel technique [10] that allowed mutation in the live code regions. Using this approach, they effectively found 168 confirmed bugs in GCC and LLVM in 13 months. The CLsmith tool [8] introduced an injection of dead-by-construction code mechanism that enabled EMI testing of OpenCL compilers.

VI. Conclusion

CUDA is one of the most popular general-purpose parallel computing platform and programming model. The correctness of CUDA compilers is the basis for the correctness of CUDA applications. In this work, we propose CUDAsmith, a practical CUDA compiler fuzzing tool to generate grammatically valid CUDA kernels, to perform differential testing, and to conduct EMI testing. Our CUDAsmith tool has successfully triggered thousands of compiler failures (including build failures, wrong code, and timeout) in both the NVCC CUDA compiler and the LLVM CUDA compiler with reasonable time cost. Furthermore, CUDAsmith identified failures not only in the trunk version of the CUDA compilers, but also the previous stable releases of the CUDA compilers. Finally, the NVIDIA and LLVM compiler developers have confirmed several compiler bugs based on our test report. For future of works, we will work on automatic test case reduction tools on CUDA kernels to facilitate more efficient bug reporting and debugging.

VII. Acknowledgment

We must thank the anonymous reviewers for their valuable comments on improving this work.

References

[1] NVIDIA. (2018) Cuda compiler bug caveats. [Online]. Available: https://devtalk.nvidia.com/default/topic/465682/compiler-bugs-caveats/

[2] NVIDA. (2018) Submit a new bug at nvidia developer website. [Online]. Available: https://developer.nvidia.com/nvidia_bug/add

[3] B. Peterson. (2017) Nvcc bug with kokkos. [Online]. Available: https://github.com/kokkos/kokkos/issues/1306

[4] Dbermond. (2017) Incompatibility of cuda comiler and glibc. caffe2. [Online]. Available: https://github.com/facebookarchive/caffe2/issues/1194

[5] Robobegkget8. (2018) Nvcc bug leading to render glitch. [Online]. Available: https://devtalk.nvidia.com/default/topic/1038585/cuda-programming-and-performance/cuda-9-2-9-2-148-update1-nvcc-compiler-bug/

[6] Daniel. (2018) Nvcc compilation timeout. Daniel. [Online]. Available: https://stackoverflow.com/questions/13041186/nvcc-takes-ages-to-compile-simple-code

[7] PhilippHRO. (2018) Slow to run tensorflow with gpu. [Online]. Available: https://github.com/tensorflow/tensorflow/issues/18652

[8] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 65–76. [Online]. Available: https://doi.org/10.1145/2737924.2737986

[9] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[10] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 849–863. [Online]. Available: https://doi.org/10.1145/2983990.2984038

[11] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. A. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: an open-source GPGPU compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, 2016, pp. 105–116. [Online]. Available: https://doi.org/10.1145/2854038.2854041

[12] NVIDIA. (2018) The cuda compilation trajectory. NVIDIA. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#cuda-compilation-trajectory

[13] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Z. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 197–208.

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[15] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[16] R. Morisset, P. Pawan, and F. Z. Nardelli, "Compiler testing via a theory of sound optimisations in the C11/C++11 memory model," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 187–196. [Online]. Available: https://doi.org/10.1145/2491956.2491967

[17] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[18] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, 2008, pp. 255–264. [Online]. Available: https://doi.org/10.1145/1450058.1450093

[19] B. Turner. (2007) Random c program generator. Bryan Turner. [Online]. Available: http://brturn.googlepages.com/randomcprogramgenerator

[20] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 85–99. [Online]. Available: https://doi.org/10.1145/2908080.2908095

[21] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015, pp. 386–399. [Online]. Available: https://doi.org/10.1145/2814270.2814319