DataRaceOnAccelerator – A Micro-benchmark Suite for Evaluating Correctness Tools Targeting Accelerators

Adrian Schmitz[1(✉)], Joachim Protze[1], Lechen Yu[2], Simon Schwitanski[1], and Matthias S. Müller[1]

[1] IT Center, RWTH Aachen University, Aachen, Germany
{a.schmitz,protze,schwitanski,mueller}@itc.rwth-aachen.de
[2] Georgia Institute of Technology, Atlanta, USA
lechen.yu@gatech.edu

Abstract. The advent of hardware accelerators over the past decade has significantly increased the complexity of modern parallel applications. For correctness, applications must synchronize the host with accelerators properly to avoid defects. Considering concurrency defects on accelerators are hard to detect and debug, researchers have proposed several correctness tools. However, existing correctness tools targeting accelerators are not comprehensively and objectively evaluated since there exist few available micro-benchmarks that can test the functionality of a correctness tool.

In this paper, we propose DataRaceOnAccelerator (DRACC), a micro-benchmark suite designed for evaluating the capabilities of correctness tools for accelerators. DRACC provides micro-benchmarks for common error patterns in CUDA, OpenMP, and OpenACC programs. These micro-benchmarks can be used to measure the precision and recall of a correctness tool. We categorize all micro-benchmarks into different groups based on their error patterns, and analyze the necessary runtime information to capture each error pattern. To demonstrate the effectiveness of DRACC, we utilized it to evaluate four existing correctness tools: ThreadSanitizer, Archer, GPUVerify, and CUDA-MEMCHECK. The evaluation results demonstrate that DRACC is capable of revealing the strengths and weaknesses of a correctness tool.

Keywords: Micro-benchmark Suite · Error classification · Accelerator

1 Introduction

Hardware accelerators are becoming increasingly popular within high performance computing area since the last decade. On the Top500 list, six out of the top ten most powerful supercomputers are equipped with GPGPU or many-core co-processors[1]. To leverage accelerators when developing parallel applications,

[1] https://www.top500.org/.

U. Schwardmann et al. (Eds.): Euro-Par 2019 Workshops, LNCS 11997, pp. 245–257, 2020.
https://doi.org/10.1007/978-3-030-48340-1_19

programmers utilize parallel programming models such as CUDA, OpenACC, and OpenMP. Those programming models ease the access to accelerators by their built-in APIs and compiler directives, while exposing enough low-level details to help tuning parallel applications. Nevertheless, the increasing complexity of programs results in higher chances of concurrency defects caused by incorrect usage of underlying programming models. Due to the lack of suitable correctness tools considering accelerators, concurrency defects may remain undetected in well-tested parallel applications. As an example, our group just recently identified and reported a mapping bug in the SPEC ACCEL OMP benchmark application 503.postencil [8], which we condensed to the reproducer in Listing 1. The code mimics an iterative solver with a dynamic break condition and works on two arrays where the output of one iteration is the input for the next iteration. In this code, the swap in line 6 has no effect on the `map` clause and therefore for the map-from at the end of the target data region. The code always maps the array originally addressed by `p1` back to the host; for odd numbers of iterations, `p2` points to that array afterwards, while `p1` points to the unmodified original `p2` array. Because of the pointer swap, the code expects `p1` to point to the result of this kernel.

Over the past years, a handful of correctness tools targeting concurrency defects on accelerators were presented, for example, GPUVerify [3], BARRACUDA [5], CUDA-MEMCHECK, and CURD [15]. Those correctness tools demonstrate the feasibility of detecting concurrency defects on accelerators. In this paper, we present *DataRaceOnAccelerator* (DRACC), a micro-benchmark suite designed to evaluate correctness tools objectively. DRACC focuses on possible concurrency defects in CUDA, OpenACC, and OpenMP programs. It covers common error patterns of concurrency defects incurred by conflicting memory accesses.

In summary, we make the following contributions:

- We present the micro-benchmark suite DRACC to evaluate correctness tools targeting accelerators;
- We thoroughly analyze the coverage of error patterns in DRACC by describing the mapping between micro-benchmarks to error pattern classifications proposed in previous work [11,14] and extending upon them by introducing mapping defects and a new categorization;

```
1 #pragma omp target data map(to:p2[0:N]) map(tofrom:p1[0:N])
2   { do {
3 #pragma omp target parallel for
4       for (int i = 0; i < N; i++)
5         p2[i] = 42 + p1[i];
6       std::swap(p1, p2); // executed on the host
7     } while (!done());
8   } // end of target data region: map(from:p1[0:N])
```

Listing 1. Mapping bug found in a SPEC ACCEL benchmark

– We introduce five levels of available information to understand the requirements and possibilities of analyzing each error pattern;
– We used DRACC to evaluate existing correctness tools: ThreadSanitizer [16, 17] and Archer [1], GPUVerify, and CUDA-MEMCHECK.

2 DataRaceOnAccelerator

Liao et al. [9,10] developed a benchmark suite, DataRaceBench, for data races in OpenMP. DataRaceBench is designed to test data race detectors for their capabilities of finding data races in OpenMP programs. This benchmark suite has been widely applied in the development and evaluation of OpenMP data race detectors [2,7].

Inspired by DataRaceBench, DRACC covers common memory driven defects on heterogeneous systems in CUDA, OpenMP, and OpenACC programs. DRACC provides a group of micro-benchmarks, developed upon following programming models and compilers: CUDA 9.1 with NVCC, OpenACC 2.6 with PGI compiler 18.4 and OpenMP 4.5 with Clang 7.0. The micro-benchmarks for DRACC are synthesized instances of the error patterns discussed in Sect. 3.

The complete micro-benchmark suite is available at Github[2] and can be compiled with the given Makefile for each programming model. All micro-benchmarks are designed based on specifications of abovementioned programming models. Thus, erroneous runtime implementations violating specifications may lead to unexpected results.

Listing 2 shows three kernels encountering an atomicity violation on the accelerator. Each of these kernels implements the same error pattern leading to an undefined value of the `countervar` variable. This failure is caused by the concurrent increment of the same variable `countervar/d_countervar`. For OpenMP and OpenACC the variable is globally accessible on the accelerator, causing a data race among the individual steps of the increment: read to a register, increment in a register, write to global memory. A further explanation of the error pattern is presented in Sect. 3.

The CUDA implementation in Listing 2 behaves differently from the OpenMP and OpenACC implementations. The device variable `d_countervar` is not explicitly defined as a global variable for CUDA, thus, each thread creates a thread-private copy of the variable, which is incremented accordingly. Due to CUDA's memory model, the result from each thread will then be copied back to the original variable. This results in a data race between the copy operations and a result of exactly `d_countervar = 1` for each execution regardless of grid and block dimension. Similar to the OpenMP and OpenACC kernel, the CUDA kernel also implements an atomicity violation.

[2] https://github.com/RWTH-HPC/DRACC.

```
__global__ void count_kernel(int *d_countervar){
    d_countervar[0]++;
}
void count(){//Launch CUDA Kernel
    count_kernel<<<100,512>>>(d_count);}
```

```
void count(){//OpenACC Kernel
    #pragma acc parallel copy(countervar) num_workers(256)
    #pragma acc loop gang worker
    for (int i=0; i<N; i++)
        countervar++;}
```

```
void count(){//OpenMP Kernel
    #pragma omp target map(tofrom:countervar) device(0)
    #pragma omp teams distribute parallel for
    for (int i=0; i<N; i++)
        countervar++;}
```

Listing 2. Examples of an atomicity violation on the accelerator in CUDA, OpenACC and OpenMP.

3 Classification

To understand the coverage of micro-benchmarks in DRACC, in this section we introduce a defect classification for application errors on heterogeneous systems. The classification shown in Fig. 1 is based on the study of Shan Lu et al. on concurrency defects [11] and the error classification by Münchhalfen et al. [14]. The classification focuses on common application errors to provide a foundation for future tool support on accelerators. Additionally, to differentiate between cause and effect of an error, we utilize the notation by A. Zeller [19] that *failure* is the manifestation of an error, e.g., non-deterministic results or a blocking application; and *defect* is the source of an error, e.g., incorrect source code.

The classification is designed to cover defects for application-programming purposes, especially regarding CUDA, OpenACC, and OpenMP programs. Syntactic correctness of the code as well as the validation of the programming model implementation, i.e., compiler and runtime, are out of scope for this work.

3.1 Overview

Using parallel programming paradigms can introduce new kinds of defects which are finally observed as failures. These are either segmentation faults or non-deterministic results. In Fig. 1 an overview of the defect classification is presented. All accelerator application defects belong to one of the following categories:

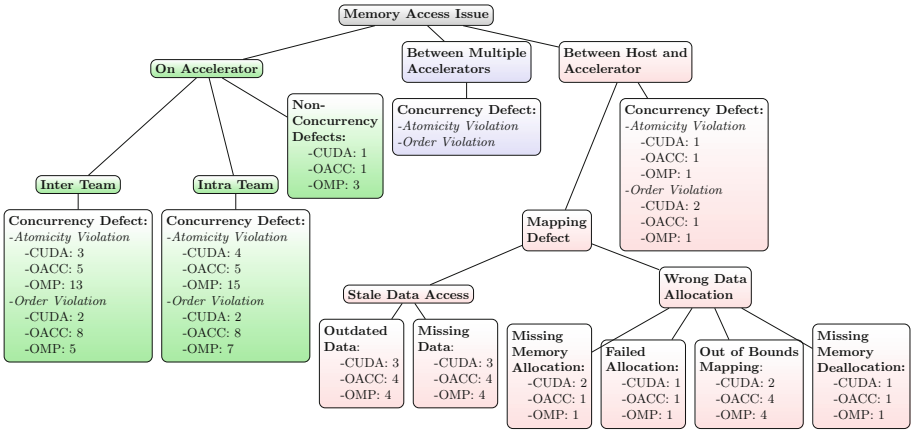


Fig. 1. A classification of common memory access defects in heterogenous computing. The number behind CUDA, OACC and OMP describes the number of micro-benchmarks that expose the corresponding defect class for the given programming model. Each micro-benchmark contains only a single defect.

1. *On Accelerator:* defects on the accelerator, independent of any other device.
2. *Between Host and Accelerator:* defects within the communication between host and device.
3. *Between Multiple Accelerators:* defects within the communication between multiple devices.

In this paper, we focus on the first two categories. Figure 1 provides an overview of the number of micro-benchmarks for each defect pattern per programming model. Since OpenMP allows the use of critical sections and locks on the device, we find more different error patterns for OpenMP.

3.2 Concurrency Defects

Concurrency defects as defined in [11], classify defects caused by concurrency. A non-deadlock concurrency defect can either be an *atomicity violation* or an *order violation*. Atomicity violation means that the intended atomicity of a group of memory accesses is not enforced. Order violation means that the intended order of two groups of memory accesses is potentially inverted, i.e., the intended order is not enforced. In this context memory accesses include memory reads, writes, allocations, and deallocations.

3.3 On Accelerator

Accelerator programming models typically provide similar high-level abstractions for program execution. An application is executed by a number of threads which are further divided into multiple *groups*. Threads belonging to the same

group can synchronize with each other, while threads from distinct groups execute independently. In this paper, we use the term *team* from OpenMP to refer to the notion of group (OpenMP *team*, OpenACC *gang*, and CUDA *thread block* express the same notion as group).

For defects on accelerators we define three classes: *Intra team concurrency defects* are bugs that occur within the same team of threads. In contrast, *inter team concurrency defects* are bugs between multiple teams of threads. Finally, *non-concurrency defects* are those defects that are not caused by concurrency, e.g., stack buffer overflows.

In general, defects occurring on CPUs may also happen on accelerators. Considering the overall architectures of hardware accelerators are vastly different from CPUs, we distinguish inter and intra team concurrency defects to clarify differences of corresponding failures.

3.4 Between Host and Accelerator

Defects between host and accelerator include order violations in the synchronization between host and accelerator, and atomicity violations for asynchronous kernels. In addition, for OpenMP atomicity violations may reside in synchronous kernels due to memory abstraction. The memory abstraction might hide the actual usage of unified or separate memory. Therefore an application relying on a specific implementation of OpenMP runtime may encounter atomicity violations.

Another defect class between host and accelerator are mapping defects. Mapping defects cover all defects related to data movement between host and accelerator. In different paradigms data movement is expressed by API functions or clauses for copy or mapping.

Some defects can be classified into both stale data access and wrong data allocation, for example, an asynchronous data movement conflicting with concurrent memory accesses. For those defects we treat it as concurrency defects.

Stale Data Access. Defects where data was not copied to or updated at the desired destination are defined as stale data access. Therefore, on host or accelerator data is missing. We distinguish *missing data*, where necessary data is not copied to the device; and *outdated data*, where data is changed on one device but not updated to the other device before accessing the data there. The example in Listing 1 shows the latter pattern, although it is not the root cause. Concurrent access to the same memory by both sides can be understood as outdated data in case of separate memory or concurrency defect in case of unified memory.

Wrong Data Allocation. Defects related to the allocation or deallocation of memory, that are not already covered by concurrency defects, are defined as wrong data allocation. We identified four different kinds of defects. When the application misses to check for a failed allocation and tries to use this memory

afterwards we call this *failed allocation*. We find *out of bounds mapping*, when the memory allocation on either side is smaller than the requested size for the mapping or copy. On *missing memory deallocation*, allocated memory is not deallocated at the end of execution which results in memory leaks. We call it *missing memory allocation* if no memory is allocated for transferred data before the first access on the device.

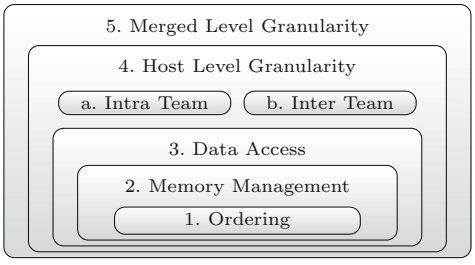
4 Information Usage

To analyze and detect the various defects described in the previous section and exposed in the provided micro-benchmarks, more or less detailed information is necessary. In this section, we discuss different levels of information which can be observed by an analysis tool. Although this might be obvious to tool developers, we believe, the following might help tool users to better understand the possibilities, limitations, and runtime overhead of specific tools. The different levels of information come with different runtime overhead and different impact on the execution. Furthermore, we classify the previously introduced defects for their necessary levels of observable information.

4.1 Five Levels of Observable Information

The five levels of observable information are a classification of the information needed to detect the defined defect classes from the prior section. An overview of the levels is given in Fig. 2. Each level consists of information about events in a program, whereby an *event* can be any observed instruction during the execution of a program. The different levels build up a hierarchy, i.e., each higher level includes all information of the corresponding lower levels. The five levels cover the following information:

1. *Ordering:* Information on the causality of events is available such that a tool can derive a happened-before relation for the events of an execution.
2. *Memory Management:* Memory allocations and data movement are tracked; this includes source, destination, and size (if applicable).



Tool	Level
CUDA-MEMCHECK	1 - 4a[a]
Archer	1 - 4
ThreadSanitizer	1 - 4
GPUVerify	None[b]

[a] shared memory only
[b] static tool

Fig. 2. The five levels of observable information and their dependencies on the left. Supported levels for all evaluated tools on the right.

3. *Data Access:* Memory location accesses are tracked on host and accelerator (read or write).
4. *Host Level Granularity:* In this level we distinguish *Intra Team Granularity*, when events within a group of threads (thread block/team/gang) can be attributed to the individual thread; and *Inter Team Granularity* when events within different groups of threads can be attributed to the group of threads.
5. *Merged Level Granularity:* The system is monitored as a whole allowing the differentiation between all threads on all devices when accessing the unified memory space.

4.2 Pattern Identification

In most cases, a tool will not be able to identify the concrete defect in the code, but in the best case pinpoint its location. The different defects result in different suspicious behavior which a tool is able to detect. This section presents for each level of information which of the previously discussed error patterns it can detect.

1. Ordering. Information on the causality of events enables a tool to detect simple order violations between host and device events. An example could be a program moving data from the host to device before any memory is allocated on the device. Since there is no further information on the address and size of the allocated memory regions, the detection capabilities of a tool with this restricted information are limited.

2. Memory Management. In case a tool tracks memory allocations and data movement in addition to just the ordering of events, it can detect all subclasses belonging to the *Wrong Data Allocation* class.

A tool can identify *Missing Memory Allocation* defects by testing if transferred memory at the destination has been allocated before the actual data movement. If the corresponding memory is only allocated afterwards, an order violation between host and accelerator could be diagnosed. *Missing Data Deallocation* defects can be detected by testing if the memory is released before the connection to the accelerator is closed. *Failed Allocation* defects can be detected by tracking if memory allocations result in errors. Subsequent null pointer access could be diagnosed as a potentially unhandled failed allocation. *Out of bounds mapping* defects can be detected by comparing the size of the data to be transferred, the size of allocated memory on source and destination, respectively.

3. Data Access. Tracking all memory accesses including their ordering on both host and device allows a tool to identify all patterns of the *Stale Data Access* class: *Missing data* defects can be diagnosed if data is read on the accelerator which is neither initialized nor copied from the host. If either is done after the access, an order violation is observed. *Outdated data* defects can be detected, when data is altered (write access) on one side but not updated to the other side

before access. In both cases, information from the data access level is necessary to detect stale data access.

The data access level is also sufficient to detect *order violations* related to data mappings between host and accelerator, because no attribution to certain threads or groups of threads is required to identify this issue.

4. Host Level Granularity. In case of concurrency defects, namely *atomicity violations* and *order violations*, it is not enough to just track memory location accesses on the accelerator: A tool also has to attribute them to the originating thread within a group of threads (intra team) or to attribute them to the group of threads in case of multiple teams (inter team). If this information is available and additionally all kinds of possible synchronization constructs are tracked (e.g., exclusive accesses), then *atomicity violations* and *order violations* on the accelerator can be detected. If a concurrency defect within a group of threads should be detected, then *intra team* granularity is required. If a concurrency defect between groups of threads should be identified, then *inter team* granularity is required.

5. Merged Level Granularity. This granularity level is only required for atomicity violations and computation related order violations in case of unified memory between host and accelerator. Since any thread on any device can be synchronized to another thread on another device, differentiation of memory accesses and synchronization between all threads on all devices accessing the unified memory space must be possible.

5 Tool Evaluation

To understand the support level of correctness tools for accelerator programming, we used DRACC to evaluate a set of existing tools, namely: ThreadSanitizer [16,17] delivered with LLVM 7.0, Archer [1] in a development version compatible with LLVM 7.0[3], GPUVerify [3] in version 2016-03-28[4], and CUDA-MEMCHECK[5] as delivered with CUDA 9.1. We carried out the experiments on Tesla P100 graphic cards on the CLAIX cluster at the RWTH Aachen University. Considering the supported programming models of these tools, we tested ThreadSanitizer and Archer with OpenMP micro-benchmarks, GPUVerify with CUDA micro-benchmarks, and CUDA-MEMCHECK with all three groups of micro-benchmarks. The supported levels of observable information for each tool are presented in Fig. 2.

Table 1 gives an overview of the evaluation results. It lists the counts of correct alerts (true positives, TP), false alerts (false positives, FP), error free

[3] https://github.com/PRUNERS/openmp/tree/archer_70 (303a691).
[4] http://multicore.doc.ic.ac.uk/tools/GPUVerify/download.php.
[5] https://docs.nvidia.com/cuda/cuda-memcheck/index.html.

(true negatives, TN), and omission (false negative, FN). Based on these counts, we calculated the metrics Precision ($P = \frac{TP}{TP+FP}$) and Recall ($R = \frac{TP}{TP+FN}$).

Since ThreadSanitizer and Archer do not support data race analysis on the accelerator, we force the OpenMP target regions in the OpenMP micro-benchmarks to be executed on the host. For both tools we compile the benchmark with the flag *-fsanitize=thread*, and for Archer we additionally load the Archer runtime library during execution. 20 out of 50 OpenMP error patterns are detected by both Archer and ThreadSanitizer, and they identify the same group of error patterns. The LLVM OpenMP implementation decides to only run a single team for the teams construct, therefore no issues in the inter team concurrency micro-benchmarks can be observed. Mapping defects are not understood by the tools, but can lead to segmentation faults. No false alerts are reported by either tool. The error patterns detected by the tools are the kind of errors which would also be detected for host code, which could be derived by removing the target regions from these micro-benchmarks.

CUDA-MEMCHECK does not detect any defect in the CUDA version of DRACC, although some micro-benchmarks result in CUDA errors. Thus, 0 of the 26 CUDA pattern implementations are detected by this tool. According to the documentation this tool can detect data races in shared device memory. For the same reason, the tool cannot detect most data races in OpenMP or OpenACC micro-benchmarks of DRACC. However, for OpenMP tests CUDA-MEMCHECK detects a generic defect during the initialization of the target region, which is disregarded in Table 1. CUDA-MEMCHECK can detect out of bounds memory mapping from the device to the host in OpenMP and OpenACC.

Since GPUVerify supports two usage modes, *-findbugs* and *-verify*, we tested these two modes on DRACC irrespectively. In both two usage modes, GPUVerify correctly detected 7 out of 26 CUDA error patterns, reported one false alarm, and failed to tackle the remaining 18 CUDA error patterns. For intra team and inter team atomicity violations, GPUVerify pinpointed these concurrency defects and generated a counter example for each concurrency defect. When analyzing the micro-benchmark for stack overflow, GPUVerify reported a false alarm that the micro-benchmark may encounter null-pointer memory access. A possible explanation for this false alarm is that GPUVerify does not model memory accesses in recursive function invocations correctly. For the remaining 18 CUDA error patterns, GPUVerify reported internal errors when analyzing the corresponding micro-benchmarks. The reason for internal errors is these CUDA error patterns are related to stale data access and wrong data allocation, while GPUVerify currently only checks data races and barrier divergence. In addition, some micro-benchmarks use new atomic operations introduced in CUDA 8.0. Since the 2016-03-28 version of GPUVerify is released earlier than CUDA 8.0, GPUVerify cannot recognize these atomic operations, which leads to internal errors.

In summary, DRACC successfully evaluated the functionality of four correctness checking tools. The observed result matches our expectation based on the description in the documentation.

Table 1. Analysis results of DRACC on four different tools, values given for CUDA/OpenACC/OpenMP

Tool	TP	FP	TN	FN	P[%]	R[%]
ThreadSanitizer	-/-/26	-/-/0	-/-/7	-/-/23	-/-/100	-/-/53
Archer	-/-/26	-/-/0	-/-/7	-/-/23	-/-/100	-/-/53
GPUVerify	7/-/-	1/-/-	3/-/-	18/-/-	87.5/-/-	28/-/-
CUDA-MEMCHECK	0/2/3	0/1/0	3/9/7	26/31/46	0/67/100	0/6/6

6 Related Work

Münchhalfen et al. [14] published an error classification of OpenMP programs and solutions for the detection. The main focus of their work is OpenMP. Offloading with OpenMP is also considered in their classification, as part of data transfer errors and data races. The classification in our work covers offloading with OpenMP, OpenACC, and CUDA.

Friedline et al. [6] developed a test suite to validate OpenACC implementations and corresponding features in OpenACC 2.5. Their work provides a validation test suite for compiler architects and programmers. This validation test suite is designed for multiple hardware architectures to test the portability of OpenACC code between these architectures.

Similar test suites for OpenMP have been developed by Müller et al. for OpenMP 2.0 [12] and OpenMP 2.5 [13]. These two test suites aim to cover the complete standard and valid combinations of OpenMP constructs. Wang et al. [18] developed a validation test suite for OpenMP 3.1. For OpenMP 4.5 Diaz and Pophale et al. [4] provided a validation test suite. These two validation test suites can verify the correctness of runtime implementation according to the specification.

7 Conclusion and Future Work

This paper introduced DRACC, a micro-benchmark suite containing common concurrency defects in CUDA, OpenACC, and OpenMP programs. DRACC was designed as a test suite for correctness tools to evaluate their functionalities. To cover as many error patterns as possible, DRACC was developed based on error pattern classifications from previous studies on concurrency defects. The evaluation of existing correctness tools demonstrates that DRACC can reveal the strengths and limitations of a correctness tool being tested. The evaluation further shows that proper tools supporting different levels of observable information are required to detect the discussed error patterns in accelerator programming.

For future work, we plan to extend DRACC to other parallel programing models which also support accelerators, for example, Kokkos and OpenCL. Furthermore, we also plan to test correctness tools on other accelerators in addition to Nvidia GPUs to conduct a more comprehensive evaluation.

References

1. Atzeni, S., Gopalakrishnan, G., et al.: ARCHER: effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS, pp. 53–62 (2016)
2. Atzeni, S., Gopalakrishnan, G., et al.: SWORD: a bounded memory-overhead detector of OpenMP data races in production runs. In: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, pp. 845–854 (2018)
3. Betts, A., Chong, N., et al.: GPUVerify: a verifier for GPU kernels. ACM SIGPLAN Not. **47**, 113–132 (2012)
4. Diaz, J.M., Pophale, S., Hernandez, O., Bernholdt, D.E., Chandrasekaran, S.: OpenMP 4.5 validation and verification suite for device offload. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) IWOMP 2018. LNCS, vol. 11128, pp. 82–95. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_6
5. Eizenberg, A., Peng, Y., et al.: BARRACUDA: binary-level analysis of runtime RAces in CUDA programs. SIGPLAN Not. **52**(6), 126–140 (2017)
6. Friedline, K., Chandrasekaran, S., Lopez, M.G., Hernandez, O.: OpenACC 2.5 validation testsuite targeting multiple architectures. In: Kunkel, J.M., Yokota, R., Taufer, M., Shalf, J. (eds.) ISC High Performance 2017. LNCS, vol. 10524, pp. 557–575. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67630-2_39
7. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for OpenMP programs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, pp. 61:1–61:12. IEEE (2018)
8. Juckeland, G., Grund, A., Nagel, W.E.: Performance portable applications for hardware accelerators: lessons learned from SPEC ACCEL. In: IEEE International Parallel and Distributed Processing Symposium Workshop (2015)
9. Liao, C., Lin, P.H., et al.: DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017. ACM (2017)
10. Liao, C., Lin, P., et al.: A semantics-driven approach to improving DataRaceBench's OpenMP standard coverage. In: Evolving OpenMP for Evolving Architectures (IWOMP 2018, Barcelona, Spain), pp. 189–202 (2018)
11. Lu, S., Park, S., et al.: Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. ACM SIGOPS Oper. Syst. Rev. **42**(2), 329–339 (2008)
12. Müller, M., Neytchev, P.: An OpenMP validation suite. In: Fifth European Workshop on OpenMP, Aachen University, Germany (2003)
13. Müller, M.S., Niethammer, C., et al.: Validating OpenMP 2.5 for Fortran and C/C++. In: Sixth European Workshop on OpenMP (2004)
14. Münchhalfen, J.F., Hilbrich, T., Protze, J., Terboven, C., Müller, M.S.: Classification of common errors in OpenMP applications. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 58–72. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_5
15. Peng, Y., Grover, V., et al.: CURD: a dynamic CUDA race detector. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 390–403. ACM (2018)
16. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA 2009, pp. 62–71. ACM (2009)

17. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 110–114. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_9
18. Wang, C., Chandrasekaran, S., Chapman, B.: An OpenMP 3.1 validation testsuite. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 237–249. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_18
19. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Elsevier, Oxford (2009)