

ESBMC-GPU

A Context-Bounded Model Checking Tool to Verify CUDA Programs

No Author Given

No Institute Given

Abstract The Compute Unified Device Architecture (CUDA) is a programming model used for exploring the advantages of Graphics Processing Unit (GPU) devices, through parallelization and specialized functions and features. However, as in other development platforms, errors may occur, due to traditional software creation processes, which may even compromise the execution of an entire system. In order to address such a problem, ESBMC-GPU was developed, as an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC). In summary, ESBMC processes input code through ESBMC-GPU and an abstract representation of the standard CUDA libraries, with the goal of checking a set of desired properties. Experimental results showed that ESBMC-GPU was able to correctly verify 82% of the chosen benchmarks and also overcame other existing GPU verifiers.

1 Introduction

The Compute Unified Device Architecture (CUDA) is a development framework that makes use of the architecture and processing power of Graphics Processing Units (GPUs) [?]. Indeed, CUDA is also an Application Programming Interface (API), through which a GPU's parallelization scheme and tools can be accessed, with the goal of executing kernels. However, source code is still written by human programmers, which may result in arithmetic overflow, division by zero, and other violation types. In addition, given that CUDA allows parallelization, problems related to the latter can be identified, due to thread scheduling [?].

In order to address that problem, an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [?] was developed, named as ESBMC-GPU [?, ?, ?], with the goal of verifying CUDA-based programs (available online at www.esbmc-gpu.org). ESBMC-GPU consists of an extension for parsing CUDA source code (*i.e.*, a front-end to ESBMC) and a CUDA operational model (COM), which is an abstract representation of the standard CUDA libraries (*i.e.*, the native API) that conservatively approximates their semantics.

A distinct difference of ESBMC-GPU, when compared with other approaches [?, ?, ?, ?], is the use of Bounded Model Checking (BMC) allied to Satisfiability Modulo Theories (SMT), with explicit state-space exploration [?]. Concurrency problems are tackled, up to an unwinding bound, while each interleaving itself is symbolically handled. However, even with BMC, space-state exploration may become a very time-consuming task, which is alleviated through state hashing and

Monotonic Partial Order Reduction (MPOR) [?]. As a consequence, redundant interleavings are eliminated, without ignoring a program’s behavior.

Finally, existing GPU verifiers normally ignore some aspects related to memory leak, data transfer, and overflow, which are normally present in CUDA programs. The proposed approach, in turn, explicitly addresses them, through an accurate checking procedure, which even considers data exchange between main program and kernel. Of course, that results in higher verification times, but more errors can then be identified and later corrected, in another development cycle.

Existing GPU Verifiers. GPUVerify [?] is based on Synchronous Delayed Visibility, which focus on detecting data race and barrier divergence, while reducing kernel verification procedures to the analysis of sequential programs. Symbolic Executor with Static Analysis (SESA) [?] and GPU + KLEE (GKLEE) [?] are concrete plus symbolic (*concolic*) execution tools. Initially, GKLEE modeled every original thread, which was later improved by taking into account symmetry, and it considers both kernel and main function, while checking barrier synchronization and data race, among others. SESA combines equivalent flows, which results in great performance improvement, focus on data race detection, and does not verify the main function. Prover of User GPU Programs (PUG) [?] is based on SMT solvers and applies symbolic static analysis to each kernel, but requires user annotations. CIVL [?] generates an abstract syntax tree and supports analysis and transformations. Besides, it also verifies concurrent programs with partial order reduction, which makes it very similar to the solution proposed here. Given that its support to CUDA libraries is still under development, CIVL is an interesting approach for checking simple CUDA programs.

2 Architecture and Implementation

ESBMC-GPU builds on ESBMC, which is an open source context-bounded model checker based on SMT solvers [?] for ANSI-C/C++ programs. ESBMC-GPU adds four essential models to ESBMC, as described below.

CUDA Operational Model. An operational model for CUDA libraries, in conjunction with ESBMC, provides support to CUDA functionalities, as shown in Fig. ???. Such an approach, which was previously attempted in the verification of C++ programs [?,?], consists in an abstract representation that reliably approximates the CUDA library’s semantics. However, COM incorporates pre- and post-conditions as assertions into verification processes, as shown in Fig. ??, which enables ESBMC-GPU to verify specific properties (cf. Sec. ??). Indeed, COM allows the necessary control for performing code analysis, where both CUDA operation and knowledge for model checking its properties are available.

ESBMC was designed to handle multi-threaded software, through the use of the Portable Operating System Interface (POSIX – ISO/IEC 9945). Thus, ESBMC-GPU applies a combination between processing methods used by Central Processing Units (CPUs) and the POSIX library, where thread instructions can interleave to create execution paths. Particularly, COM simulates the behavior of kernel calls using pthread functions (*e.g.*, `pthread_create`) and combines

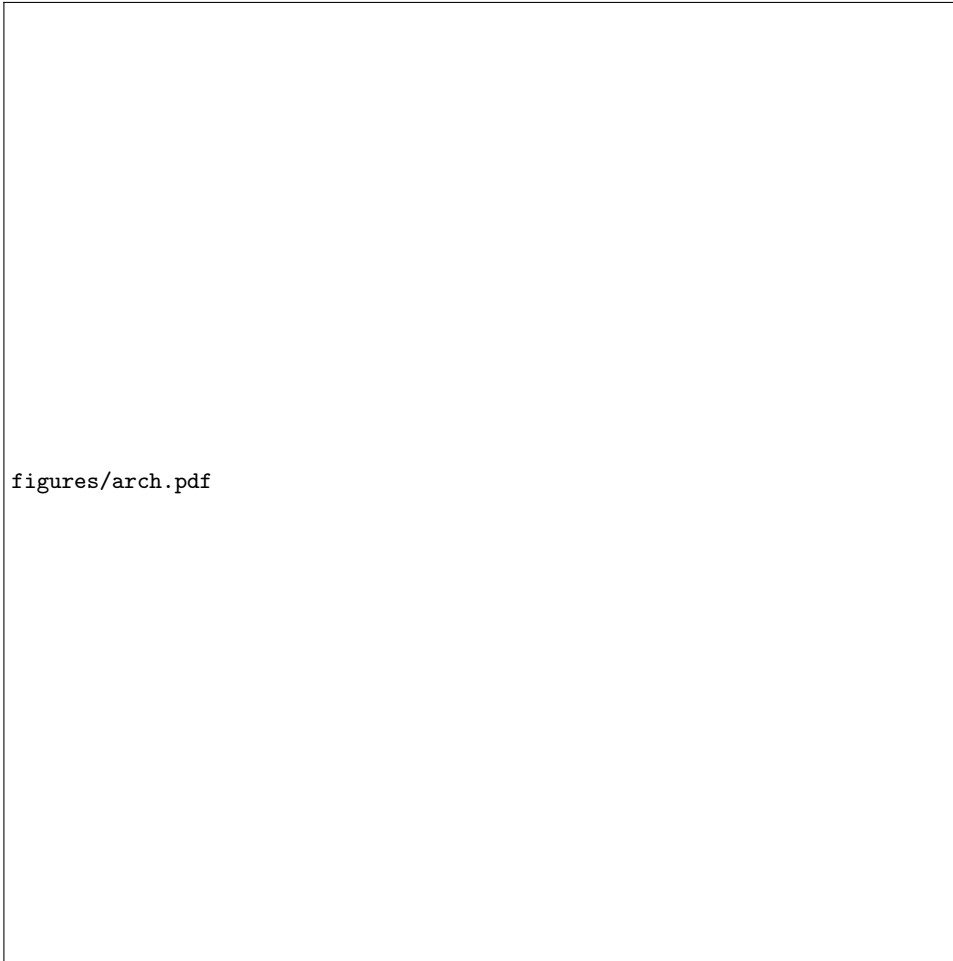
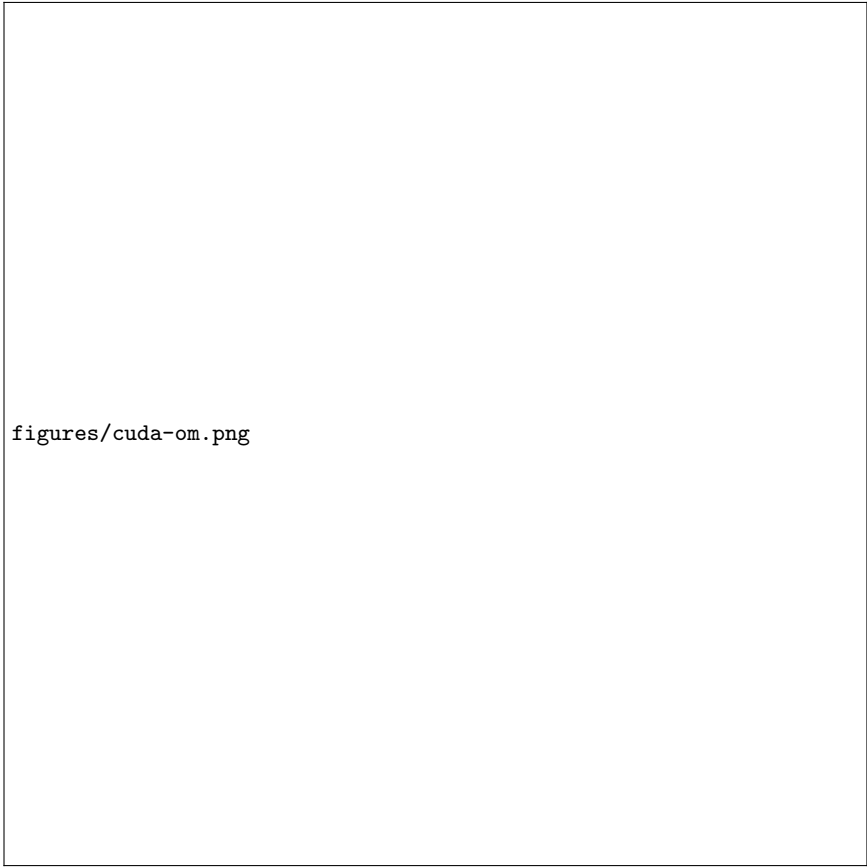


Figure 1: Overview of ESBMC-GPU's architecture.

that with ESBMC, in order to check data race and specific C/C++ programming language failures (*e.g.*, array out-of-bounds and pointer safety).

Two-threads Analysis. Similarly to GPUVerify [?] and PUG [?], ESBMC-GPU also reduces the number of threads, during the verification of CUDA programs, to two, considering a NVIDIA Fermi GPU architecture, in order to improve verification time and avoid the state-space explosion problem. In CUDA programs, whilst threads execute the same parametrized kernel, only two threads are necessary for conflict check. Thus, such an analysis ensures that errors (*e.g.*, data races) detected between two threads, in a given subgroup and due to unsynchronized accesses to shared variables, are enough to justify a property violation.



figures/cuda-om.png

Figure 2: CUDA operational model development process.

State Hashing. ESBMC-GPU applies state hashing to further eliminate redundant interleavings and reduce state space, based on SHA256 hashes.

Monotonic Partial Order Reduction. MPOR is used to reduce the number of thread interleavings, by classifying transitions inside a program as dependent or independent. As a consequence, it is possible to determine whether interleaving pairs always lead to the same state and then remove duplicates in a reachability tree, without ignoring any aspect of a program’s behavior.

3 Functionalities

Through the integration of COM into ESBMC, ESBMC-GPU is able to analyze CUDA programs and verify the following properties:

- **Data race.** ESBMC-GPU checks data race conditions, in order to detect if multiple threads perform unsynchronized access to shared-memory locations.

- **Pointer safety.** ESBMC-GPU also ensures that *(i)* a pointer offset does not exceed object bounds and *(ii)* a pointer is neither NULL nor invalid.
- **Array bounds.** ESBMC-GPU performs array-bound checking, in order to ensure that any variable, used as an array index, is within known bounds.
- **Arithmetic under- and overflow.** ESBMC-GPU checks whether a sum or product exceeds the memory limits that a variable can handle.
- **Division by zero.** ESBMC-GPU analyzes whether denominators, in arithmetic expressions, lead to a division by zero.
- **User-specified assertions.** ESBMC-GPU considers all assertions specified by users, which is essential to a thorough verification process.

4 Usage Aspects

In order to verify CUDA code, users must slightly modify one specific parameter, in a given program. In particular, the call

```
kernel<<< blocks, threads >>>(kernel-parameters)
```

must be transformed to the following ESBMC-GPU intrinsic function: called `ESBMC_verify_kernel()`, which takes as arguments: the kernel to be verified, the number of blocks, the number of threads, and the kernel parameters, respectively.

```
ESBMC_verify_kernel(<kernel>, <blocks>,
                   <threads>, <kernel-parameters>),
```

where `<kernel>` is the kernel to be verified, `<blocks>` is the number of blocks, `<threads>` is the number of threads, and `<kernel-parameters>` is the kernel function's parameters. After the required amendments, the user must call the ESBMC-GPU v2.0 command-line version, using

```
esbmc-gpu <file>.cu --unwind <k> --context-switch <c>
--state-hashing -I <path-to-CUDA-OM>,
```

where `<file>.cu` is the CUDA program to be verified, `<k>` is the maximum loop unrolling, `<c>` is a context-switch bound among all threads, `--state-hashing` reduces redundant interleavings, and `<path-to-CUDA-OM>` is the location of the COM library. In addition, the flag `--data-races-check` must be added at the end of the aforementioned command line, in order to check data races.

In summary, if no bug is found, up to a k -depth unwinding and a c context-bound, ESBMC-GPU reports `VERIFICATION SUCCESSFUL`. Conversely, it reports `VERIFICATION FAILED`, along with a counterexample, which contains all necessary information for identifying and reproducing the respective error. It is worth mentioning that counterexamples are not only recipients that contain the necessary information for reaching an error state, but it can also be used for implementing proper code correction, even automatically.

5 Experimental Evaluation

The chosen benchmark suite comprises 20 CUDA kernels from NVIDIA GPU Computing SDK v2.0, 20 CUDA kernels from Microsoft C++ AMP Sample Projects, and 104 CUDA-based programs that explore a wide range of CUDA functionalities [?]. In summary, the chosen suite contains 47.4% bug-free and 52.6% buggy benchmarks, which tackle data race, arithmetic operations, pointer assignment, `_device_` function calls, general ANSI-C functions (*e.g.*, `memset`), general CUDA functions (*e.g.*, `cudaMemcpy`), general libraries in CUDA (*e.g.*, `curand.h`), and the ability to work with variables, type modifiers (*e.g.*, `unsigned`), pointers, type definitions, and intrinsic CUDA variables (*e.g.*, `uint4`). The present experiments answer two research questions:

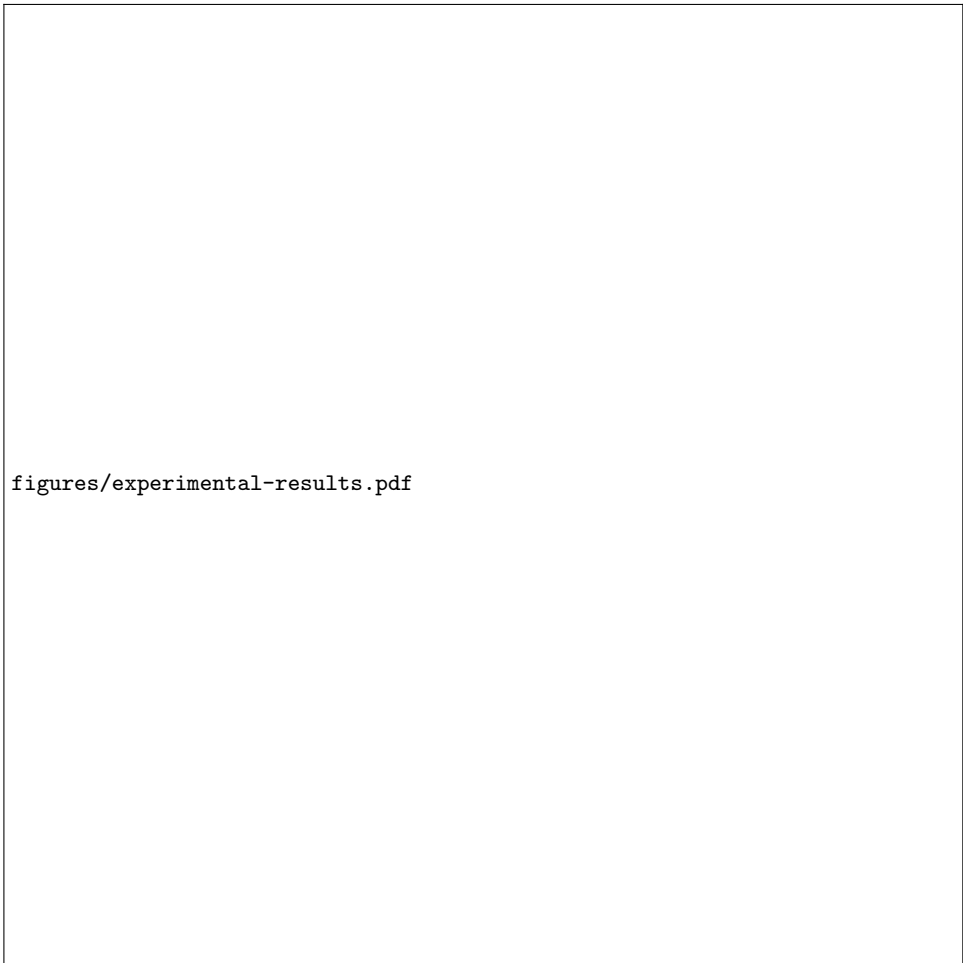
1. How accurate is ESBMC-GPU when verifying the chosen test suite?
2. How does ESBMC-GPU compare to other existing verifiers?

In order to answer both questions, all benchmarks were verified with 5 GPU verifiers (ESBMC-GPU, GKLEE, GPUVerify, PUG, and CIVL), on an otherwise idle Intel Core i7-4790 CPU 3.60 GHz, with 16 GB of RAM and a Linux OS. All times were measured by the UNIX time command, in seconds. An overview of the experimental results is shown in Fig. ??, where *True* represents bug-free benchmarks, *False* represents buggy benchmarks, *Not supported* represents benchmarks that could not be verified, *Correct* represents the percentage of benchmarks correctly verified, and *Incorrect* represents the percentage of benchmarks incorrectly verified. As one may notice, the present experimental results show that ESBMC-GPU reached a successful verification rate of approximately 82%, while GKLEE, GPUVerify, PUG, and CIVL reported 70%, 57%, 35%, and 31%, respectively. More precisely, ESBMC-GPU support the verification of benchmarks related to array bounds (3%), assertive statements (5%), data race (11%), NULL pointers (3%), and other specific CUDA functionalities (60%).

Limitations. ESBMC-GPU was unable to correctly verify 27 benchmarks, which are related to constant memory access (2%), the use of CUDA’s specific libraries (4.5%), and the use of pointers to functions, structures, and `char` type variables, when used as kernel call arguments (8.5%). In addition, ESBMC-GPU only reported 1% of false positive and 2% false negative results.

Performance. MPOR resulted in a performance improvement of approximately 80%, by decreasing the verification time from 16 to 3 hours, while the two-threads analysis further reduced that to 811 seconds. Although such techniques have considerably improved the ESBMC-GPU’s performance, it still takes longer than the other evaluated tools: GKLEE (128 sec), GPUVerify (147 sec), PUG (12 sec), and CIVL (158 sec). This is due to thread interleavings, which combine symbolic model checking with explicit state-space exploration. In addition, even though ESBMC-GPU reports the highest verification time, it still presents the highest accuracy, with less than 6 seconds per benchmark.

Availability of Data and Tools. The performed experiments are based on a set of publicly available benchmarks. All benchmarks, tools, and results, associated with the current evaluation, are available on <http://esbmc-gpu.org/>.



figures/experimental-results.pdf

Figure 3: Experimental evaluation of ESBMC-GPU against other verifiers.

6 Conclusions and Future Work

ESBMC-GPU is an SMT-based BMC tool that recognizes CUDA directives, further simplifies verification models, and provides fewer incorrect results, if compared with GKLEE, GPUVerify, PUG, and CIVL. In addition, it presents an improved ability to detect array out-of-bounds and data race violations. Future work aims to support stream interleaving and implement further techniques to reduce the number of thread interleavings by taking into account the GPU symmetry.

References

1. Cheng, J., Grossman, M., and McKercher, T. *Professional CUDA C Programming*. John Wiley and Sons, Inc. 2014.
2. Betts, A., Chong, N., Donaldson, A., Qadeer, S., and Thomson, P. *GPUVerify: A Verifier for GPU Kernels*. OOPSLA 2012; 113–132.
3. Cordeiro, L. and *et al.* *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. IEEE Trans. Software Eng. 2012; 38(4):957–974.
4. Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Santos, V., Ferreira, R., and Cordeiro, L. *Verificação de Kernels em Programas CUDA usando Bounded Model Checking*. WSCAD-SSC 2015; 24–35.
5. Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Santos, V., Ferreira, R., and Cordeiro, L. *Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking*. SAC SVT track 2016; 1648–1653.
6. Pereira, P., Albuquerque, H., Silva, I., Marques, H., Monteiro, F., Ferreira, R., and Cordeiro, L. *SMT-Based Context-Bounded Model Checking for CUDA Programs*. Concurrency Computat.: Pract. Exper. 2016; (to appear).
7. Li, G. and Gopalakrishnan, G. *Scalable SMT-based Verification of GPU Kernel Functions*. FSE 2010; 187–196.
8. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., and Rajan, S. *GKLEE: Concolic Verification and Test Generation for GPUs*. PPOPP 2012; 215–224.
9. Zheng, M., Rogers, M., Luo, Z., Dwyer, M., and Siegel, S. *CIVL: Formal Verification of Parallel Programs*. ASE 2015; 830–835.
10. Kahlon V, Wang C, Gupta A. *Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique*. CAV 2009; 398–413.
11. Li, P., Li, G., and Gopalakrishnan, G. *Practical Symbolic Race Checking of GPU Programs*. SC 2014; 179–190.
12. Ramalho, M., Freitas, M., Sousa, F., Marques, H., Cordeiro, L., and Fischer, B. *SMT-Based Bounded Model Checking of C++ Programs*. ECBS 2013; 147–156.
13. Garcia, M., Monteiro, F., Cordeiro, L., de Lima Filho, E. *ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications*. SPIN 2016; 97–103.

Appendix

The presentation plan of our work consists mainly of three parts. First, a brief motivation about general bugs in CUDA-based programs and how our present approach can assist CUDA developers to detect those bugs. Second, an overview about the ESBMC-GPU implementation and its verification approach. Third, a quick demonstration of ESBMC-GPU command-line version, using CUDA kernels from the developed benchmark test suite.

1. Introduction and Motivation (20%)

- Brief introduction about CUDA, from the perspective of a parallel computing platform and API model:
 - Present the motivation as described by NVIDIA to configure GPUs;
 - Show initial applications in graphical processing in games applications (especially those that require high computational power);
 - Demonstrate further applications to biomedicine, air traffic control, and weather simulation.
- Typical Programming Errors in CUDA:
 - Show that arithmetic under- and overflow, buffer overflow, pointer safety, and division by zero are also common source of errors in CUDA programs;
 - Emphasize that data race conditions, shared memory, and barrier divergence play an important role in CUDA-based program verification.
 - (a) Demonstrate that all those errors lead to incorrect results during the program execution; and
 - (b) They are hard to detect due to the parallel operations.
- Ensure code correctness in (critical) GPU-based applications.
 - Exploit SMT-based context-BMC to verify CUDA-based programs.
 - (a) Present an operational model for the CUDA platform, which consists of pre- and post-conditions and simulation features;
 - (b) Apply SMT-based context-bounded model checking to CUDA-based programs using MPOR, state-hashing, and two-thread analysis;
 - (c) Show the experimental results of ESBMC-GPU if compared to other state-of-art software verifiers for CUDA programs.

2. ESBMC-GPU Implementation Aspects (20%)

- Describe all properties that are currently supported by the ESBMC-GPU tool, which are:
 - Data race;
 - Pointer safety (enabled by default);
 - Memory leaks;

- Array bounds (enabled by default);
 - Arithmetic under- and overflow;
 - Division by zero (enabled by default);
 - User-specified assertions.
- ESBMC-GPU implementation and architecture:
- CUDA Operational model;

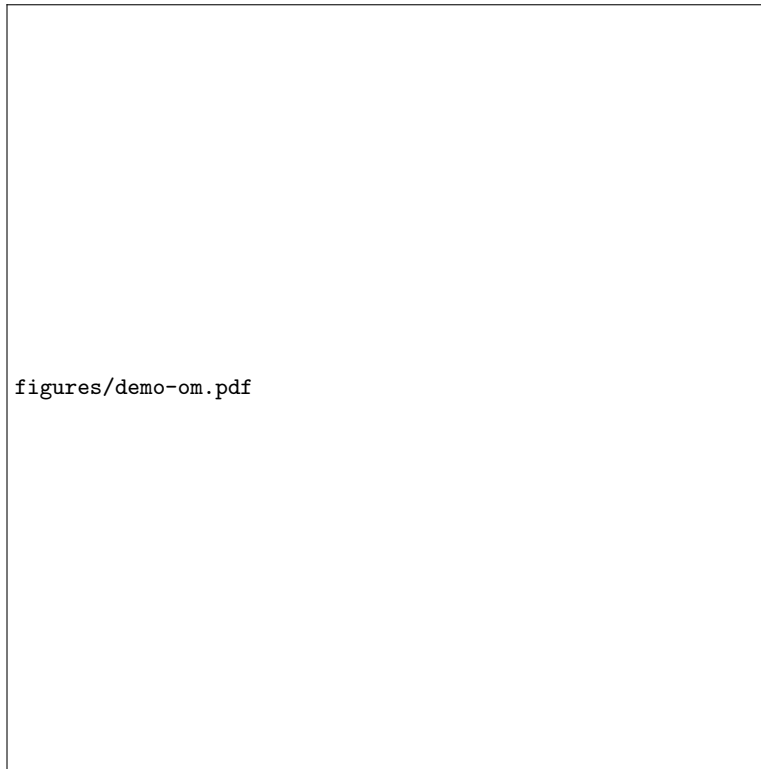


Figure 4: CUDA operational model scheme.

- (a) As shown in Fig. ??, operational models add assertions to the verification process, in order to check for specific properties;
 - (b) Verification model adopts the CPU parallel processing based on the POSIX Threading Library;
- Two-threads Analysis;
 Fig. ?? demonstrates the thread representation, which accesses the GPU shared memory and is based on NVIDIA Fermi GPU architecture. First of all, the numbered white boxes represent the shared memory positions to be accessed, t_r^n are the thread-readers, and t_w^n

are the thread-writers, where n is the index of each thread. In the first case of Fig. ?? (*i.e.*, S_1), the threads, whose kernel is correctly implemented, do not present a data race condition, once they access different memory positions. In the second case of Fig. ?? (*i.e.*, S_2), the threads execution of a kernel results in a data race condition, where several threads read or write in the same memory position, at the same lock-step. Finally, the third case of Fig. ?? is quite similar to the situation occurred in S_2 , however, instead of using multiple threads to analyze data-race conditions, ESBMC-GPU ensures that such errors can be detected through the analysis of the behavior of two threads that are operating over the same memory location (*i.e.*, S_3).



Figure 5: Two-threads analysis.

Note that the two-threads analysis affects mostly the data race verification, where program states must be analyzed with respect to their possible thread interleavings, which lead to an execution order of statements that results in error. In our benchmarks, we observed that the consideration of solely two threads attains substantial improvement in performance, whilst keeping true incorrect results at low rates.

- **State Hashing;**
Our symbolic state hashing approach computes a summary for a particular state that has been explored, and then index the set of explored state to reduce the generation of redundant states. In particular, given any state computed during the symbolic execution of a specific CUDA kernel, we simply summarize it and efficiently look up whether it has been explored before, along a different computation path; if that happens during the ESBMC-GPU symbolic execution, then the current computation path does not need to be further ex-

plored in the reachability tree. So if we reach a state in ESBMC-GPU, where a context switch can be taken (*e.g.*, before a global variable or synchronization primitive), and all shared/local variables together with the program counters are similar to another explored node, then ESBMC-GPU just considers one identical node to be further explored, since reachability subtrees associated to identical nodes are also similar.

- Monotonic Partial Order Reduction;

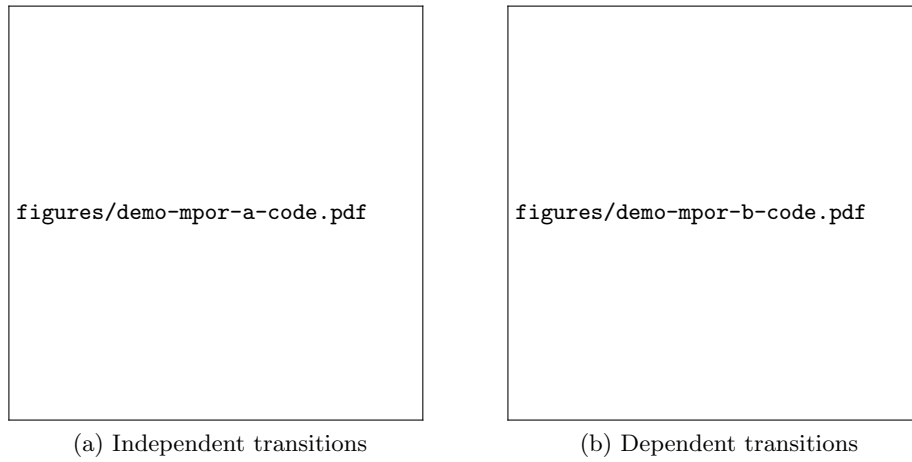


Figure 6: Code snippets of CUDA kernels.

In order to understand MPOR algorithm, one can observe the code snippets shown in Fig. ???. In addition, Fig. ??? shows the application of MPOR to a CUDA kernel, where the global variable a is written in a position relative to the thread ID; in particular, this variable determines the program execution state. On the first interleaving, thread t_1 reaches v_1 resulting in $a = [0, 0]$. Thread t_2 runs and reaches v_2 resulting in $a = [0, 1]$. The MPOR algorithm then checks whether v_2 does not exist and whether transition $t_1 \rightarrow t_2$ is defined as independent. Both conditions do not hold, therefore transition $t_1 \rightarrow t_2$ is defined as dependent. ESBMC-GPU thus performs all possible interleavings, and the next execution starts with thread t_2 from v_0 , which changes the array's content to $a = [0, 1]$. Thread t_1 runs and reaches v_4 , which is similar to v_2 . MPOR algorithm then checks whether thread t_1 has reached a redundant state. As thread t_1 has only reached v_1 , then the algorithm concludes that the transition $t_2 \rightarrow t_1$ is independent, disregarding it (represented in Fig. ??? by dotted lines).

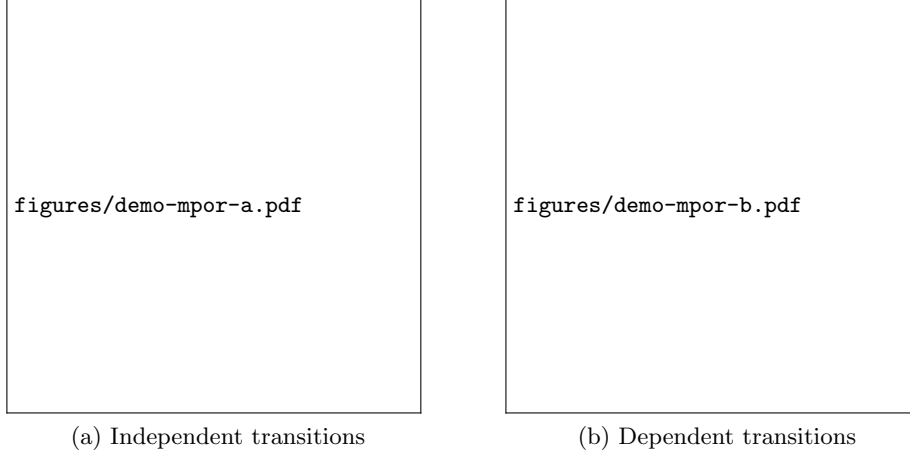


Figure 7: MPOR applied to a kernel with independent and dependent transitions.

Fig. ?? shows another example in which the thread execution results in different states. On the first interleaving, the thread execution t_1 reaches v_1 and the thread execution t_2 reaches v_2 . Since the content of $a[1]$ is different in both reachability tree (RT) nodes, the condition does not hold. On the second interleaving, thread t_2 modifies the array a to $a = [0, 1]$ and thread t_1 accidentally writes to $a[2]$. MPOR algorithm then checks whether v_4 exists in the RT. Since the condition does not hold, transition $t_2 \rightarrow t_1$ is then defined as dependent. In this particular case, both interleavings are considered and those two thread execution orders result in an array out-of-bounds violation since the array a is of length 2 (*e.g.*, v_4 in dotted lines).

3. ESBMC-GPU Demonstration (60%)

In this part, we will demonstrate the ESBMC-GPU usage, based on the CUDA program shown in Fig. ?. First of all, the user must replace the default kernel call (line 25) to an intrinsic function of ESBMC-GPU (line 26), as explained in Sec. ?. Then, the modified CUDA program can be passed to the command-line version of ESBMC-GPU as follows:

```
esbmc-gpu <file>.cu --unwind <k> --context-switch <c>
--state-hashing -I <path-to-CUDA-OM>,
```

As one may see in Fig. ?, this CUDA-based program has 1 block and 2 threads, and a kernel (lines 12 to 14), which assigns thread's index values to an array passed as an input argument. Indeed, its goal is to instantiate array positions according to the thread index. However, there is a mistake in the array index, as the value 1 is accidentally added to the thread index (line

```

1 #include <call_kernel.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <cuda.h>
6 #include <cuda_runtime_api.h>
7 #include <assert.h>
8
9 #define BLOCKS 1
10 #define THREADS 2
11
12 __global__ void kernel(int *A) {
13     A[threadIdx.x + 1] = threadIdx.x;
14 }
15
16 int main(){
17     int *a;
18     int *dev_a;
19     int size = THREADS*sizeof(int);
20     a = (int*)malloc(size);
21     cudaMalloc((void**)&dev_a, size);
22     for (int i = 0; i < THREADS; i++)
23         a[i] = 0;
24     cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
25     // kernel<<<BLOCKS,THREADS>>>(dev_a);
26     ESBMC_verify_kernel(kernel, BLOCKS, THREADS, dev_a);
27     for (int i = 0; i < THREADS; i++)
28         assert(a[i]==i);
29     cudaFree(dev_a);
30     return 0;
31 }

```

Figure 8: Illustrative CUDA code example.

13). As shown in the main function, array positions are assigned with value 0 (line 23), and after the kernel call (line 26), it is expected that $a[0] == 0$ and $a[1] == 1$.

In this example, ESBMC-GPU detects an array out-of-bounds violation. Indeed, this CUDA-based program retrieves a memory region that has not been previously allocated, thus, when $\text{threadIdx.x} = 1$, the program tries to access the position $a[2]$. Importantly, the `cudaMalloc()` function operational model has a precondition that checks if the memory size to be allocated is greater than zero. In addition, an assertion checks if the result matches to the expected postcondition (line 28). Therefore, the verification of this program through ESBMC-GPU produces 34 successful and 6 failed interleavings. For instance, one possible failed interleaving is represented by the threads executions $t_0 : a[1] = 0$; $t_1 : a[2] = 1$, where $a[2] = 1$ represents an incorrect access to the array index a .

It is worth noticing that ESBMC-GPU and GKLEE are able to detect this array out-of-bounds violation, but GPUVerify and PUG fail, as they report a true incorrect result (missed bug). Furthermore, ESBMC-GPU has the following additional command-line options:

- `--memory-leak-check`: to enable memory leak check;
- `--overflow-check`: to enable arithmetic over- and underflow check;
- `--data-races-check`: to enable data races check;
- `-DGPU_threads=<number>`: to define the total number of threads by kernel.

The tool, operational model, and all experiments results are available for downloading at: <http://esbmc-gpu.org/>.

Video demonstration at <https://www.youtube.com/watch?v=Q7p0Ar6zY3c>.