ESBMC-GPU
A Context-Bounded Model Checking Tool to Verify CUDA Programs

Felipe R. Monteiro[1], Erickson H. da S. Alves[1], Isabela S. Silva[1], Hussama A. Ismail[1], Lucas C. Cordeiro[1,2], and Eddie B. de Lima Filho[1,3]

[1] *Faculty of Technology, Federal University of Amazonas, Brazil*
[2] *Department of Computer Science, University of Oxford, United Kingdom*
[3] *FPF Tech, Brazil*

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. Nonetheless, 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 85% of the chosen benchmarks and it also overcame other existing GPU verifiers.

Keywords: GPU verification, formal verification, model checking, CUDA

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) [1]. 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. Nonetheless, 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 also occur, due to thread scheduling [2].

In order to address the mentioned issues, an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [4] was developed, named as ESBMC-GPU [5, 6, 7], with the goal of verifying CUDA-based programs (available online at `http://esbmc.org/gpu`). 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 feature of ESBMC-GPU, when compared with other approaches [2, 8, 9, 10], is the use of Bounded Model Checking (BMC) [11] allied to Satisfiability Modulo Theories (SMT) [12], with explicit state-space exploration [3, 4]. In summary, 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) [13]. As a consequence, redundant interleavings are eliminated, without ignoring a program's behavior.

Finally, existing GPU verifiers often 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. Obviously, it results in higher verification times, but more errors can then be identified and later corrected, in another development cycle.

2. Architecture and Implementation

ESBMC-GPU builds on top of ESBMC, which is an open source context-bounded model checker based on SMT solvers for ANSI-C/C++ programs [3, 4], and adds four essential models, as described below.

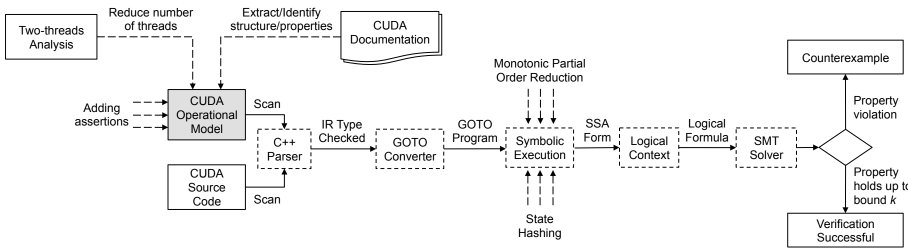

Figure 1: Overview of ESBMC-GPU's architecture.

1. CUDA Operational Model. An operational model for CUDA libraries that provides support to CUDA functionalities, in conjunction with ESBMC, as shown in Fig. 1. Such an approach, which was previously attempted in the verification of C++ programs [14, 15, 16, 17], consists in an abstract representation that reliably approximates the CUDA library's semantics; however, COM incorporates pre- and post-conditions into verification processes, which enables ESBMC-GPU to verify specific properties (cf. Sec. 3). 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 an API called Portable Operating System Interface (POSIX – ISO/IEC 9945) [18]. Thus, ESBMC-GPU applies a combination of 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 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).

2. Two-threads Analysis. Similarly to GPUVerify [2] and PUG [8], ESBMC-GPU also reduces the number of threads (to only two elements), during the verification of CUDA programs, by considering a NVIDIA Fermi GPU architecture, in order to improve verification time and avoid the state-space explosion problem. Besides, in CUDA programs, whilst threads execute the same parametrized kernel, only two of them 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 [7].

3. State Hashing. ESBMC-GPU applies state hashing to further eliminate redundant interleavings and also reduce state space, based on SHA256 hashes [19]. In particular, its symbolic state hashing approach computes a summary for a particular state that has already been explored and then indexes the resulting set, in order to reduce the generation of redundant states. Given any state computed during the symbolic execution of a specific CUDA kernel, ESBMC-GPU simply summarizes it and efficiently determines whether it has been explored before or not, along a different computation path. When this behavior is confirmed, which happens during the ESBMC-GPU's symbolic-execution procedure, then the current computation path does not need to be further explored in the associated reachability tree (RT). This way, if ESBMC-GPU reaches such a state, *i.e.*, where a context switch can be taken (*e.g.*, before a global variable or synchronization primitive) and all shared/local variables and program counters are similar to another explored node, then ESBMC-GPU just considers that an identical node to be further explored, since reachability subtrees associated to them are also similar [7, 20].

4. 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 program's behavior [20].

3. Functionalities

Through the integration of COM into ESBMC (*i.e.*, ESBMC-GPU), one is able to analyze CUDA programs and verify the following properties: data-race conditions, in order to detect if multiple threads perform unsynchronized access to shared-memory locations; pointer safety, *i.e.*, whether *(i)* a pointer offset does not exceed object bounds and *(ii)* a pointer is neither `NULL` nor invalid; array bounds, in order to ensure that array indices are within known bounds; arithmetic under- and overflow, which happens when a sum or product exceeds the memory limits that a variable can handle; division by zero, which takes place when denominators, in arithmetic expressions, lead to a division by zero; and user-specified assertions, *i.e.*, all assertions specified by users, which is essential to a thorough verification process.

In order to check the aforementioned properties, ESBMC-GPU explicitly explores the possible interleavings (up to the given context bound) and calls the single-threaded BMC procedure on each one, whenever it reaches an RT leaf node. Then, the mentioned procedure will stop if it finds a bug or when all possible RT interleavings has been systematically explored [7].

4. Illustrative Example

In this part, ESBMC-GPU usage is demonstrated, by using the CUDA program shown in Fig. 2. First of all, users must replace the default kernel call (line 16) by an intrinsic function of ESBMC-GPU (line 17). Then, the resulting 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>`, where <file>`.cu` is the CUDA program, <*k*> is the maximum loop unrolling, <*c*> is a context-switch bound, `--state-hashing` reduces redundant interleavings, and <`path-to-CUDA-OM`> is the location of the COM library.

In the mentioned 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, *i.e.*, when `threadIdx.x = 1`, the program tries to access $a[2]$. Importantly, the `cudaMalloc()` function's 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 19). Therefore, the verification of this program through ESBMC-GPU produces 54 successful and 3 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 CIVL, ESBMC-GPU, and GKLEE are also able to detect this array out-of-bounds violation, but GPUVerify fails, as it reports a true incorrect result (missed bug).

```
#include <...>
#define BLOCKS 1
#define THREADS 2
__global__ void kernel(int *A) {
  A[threadIdx.x + 1] = threadIdx.x;
}
int main(){
  int *a;
  int *dev_a;
  int size = THREADS*sizeof(int);
  a = (int*)malloc(size);
  cudaMalloc((void**)&dev_a, size);
  for (int i = 0; i < THREADS; i++)
    a[i] = 0;
  cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
  // kernel<<<BLOCKS,THREADS>>>(dev_a);
  ESBMC_verify_kernel(kernel, BLOCKS, THREADS, dev_a);
  for (int i = 0; i < THREADS; i++)
    assert(a[i]==i);
  cudaFree(dev_a);
  return 0;
}
```

Figure 2: Illustrative CUDA code example.

5. Experimental Evaluation

In order to evaluate ESBMC-GPU's precision and performance, benchmarks[1] were extracted from the available literature (*i.e.*, NVIDIA GPU Computing SDK v2.0 [21] and Microsoft C++ AMP Sample Projects [22]), which covers basic functions commonly used by real CUDA applications. The present experiments answer two research questions: *(i)* How accurate is ESBMC-GPU when verifying the chosen benchmarks? *(ii)* How does ESBMC-GPU's performance compare to other existing verifiers?

In order to answer both questions, all benchmarks were verified with 4 GPU verifiers (ESBMC-GPU $v2.0$, GKLEE $v2012$, GPUVerify $v1811$, and CIVL $v1.7.1$), on an otherwise idle Intel Core i7-4790 CPU 3.60 GHz, with 16 GB of RAM, running Ubuntu 14.04 OS. Importantly, all presented execution times are actually CPU times, *i.e.*, only the elapsed time periods spent in the allocated CPUs, which was measured with the `times` system call (POSIX system). An overview of the experimental results is shown in Fig. 3, 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 (*i.e.*, a verification tool reports an unexpected result). As one may notice, the present experimental results show that ESBMC-GPU reached a successful

[1] A detailed description of all benchmarks is available at `http://esbmc.org/gpu/`

verification rate of approximately 85%, while GKLEE, GPUVerify, and CIVL reported 72%, 50%, and 35%, respectively[2]. More precisely, ESBMC-GPU supports the verification of benchmarks related to array bounds (3%), assertive statements (5%), data race (11%), `NULL` pointers (3%), and other specific CUDA functionalities (63%).

Limitations. ESBMC-GPU was unable to correctly verify 24 benchmarks, which are related to constant memory access (2%), CUDA's specific libraries (4.5%), and the use of pointers to functions, structures, and `char` type variables, when passed as kernel call arguments (4.5%). In addition, it only reported 3% of incorrect true and 1% of incorrect false 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 789.6 sec. Although such techniques have considerably improved the ESBMC-GPU's performance, it still takes longer than the other evaluated tools: GPUVerify (98.36 sec), GKLEE (108.32 sec), and CIVL (708.52 sec). This is due to thread interleavings, which combine symbolic model checking with explicit state-space exploration [7]. In addition, ESBMC-GPU still presents the highest accuracy, with less than 6 seconds per benchmark.

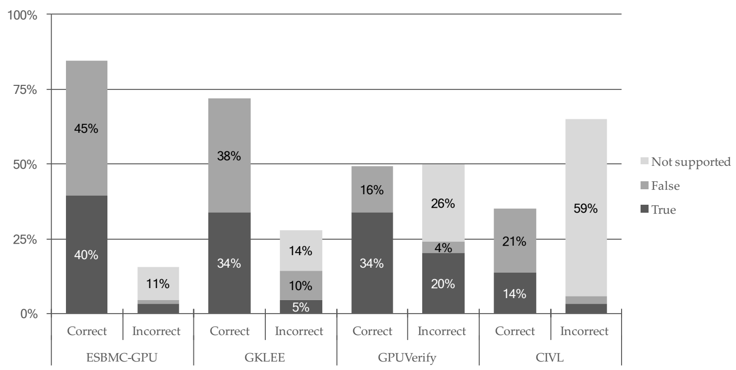

Figure 3: Experimental evaluation of ESBMC-GPU against other verifiers.

6. Conclusions and Future Work

ESBMC-GPU marks the first application of an SMT-based context-BMC tool that recognizes CUDA directives [7]. Besides, it further simplifies verification models and provides fewer incorrect results, if compared with GKLEE, GPUVerify, and CIVL. Finally, it presents 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 GPU symmetry.

[2] All experimental results are available at `http://esbmc.org/gpu/`

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 Fischer, B. *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking*. ICSE 2011; 331–340.

[4] Cordeiro, L. and, Fischer, B., and Marques-Silva, J. *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. IEEE Trans. Software Eng. 2012; 38(4):957–974.

[5] 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.

[6] 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.

[7] 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).

[8] Li, G. and Gopalakrishnan, G. *Scalable SMT-based Verification of GPU Kernel Functions*. FSE 2010; 187–196.

[9] 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.

[10] Zheng, M., Rogers, M., Luo, Z., Dwyer, M., and Siegel, S. *CIVL: Formal Verification of Parallel Programs.* ASE 2015; 830–835.

[11] Biere, A. *Bounded Model Checking. Handbook of Satisfiability.* IOS Press 2009; 457–481.

[12] Barrett C, Sebastiani R, Seshia S, Tinelli C. *Satisfiability Modulo Theories. Handbook of Satisfiability.* IOS Press 2009; 825–885.

[13] Kahlon V, Wang C, Gupta A. *Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique.* CAV 2009; 398–413.

[14] 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.

[15] Monteiro, F., Cordeiro, L., and de Lima Filho, E. *Bounded Model Checking of C++ Programs Based on the Qt Framework.* GCCE 2015; 179–447.

[16] Garcia, M., Monteiro, F., Cordeiro, L., and de Lima Filho, E. *ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications.* SPIN 2016; 97–103.

[17] Monteiro, F., Garcia, M., Cordeiro, L., and de Lima Filho, E. *Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework.* Software Testing, Verification & Reliability. 2017; (*to appear*).

[18] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004).* IEEE 2008.

[19] Federal Information Processing Standard 180-2. *Secure Hash Standard.* National Institute of Standards and Technology, 2002.

[20] Morse, J. *Expressive and Efficient Bounded Model Checking of Concurrent Software.* University of Southampton, PhD Thesis 2015.

[21] NVIDIA. *CUDA Toolkit Release Archive.* `https://developer.nvidia.com/cuda-toolkit-archive`, 2015.

[22] Microsoft Corporation. *C++ AMP sample projects for download (MSDN blog).* `blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx`, 2012.

Required Metadata

Current executable software version

Ancillary data table required for sub version of the executable software.

Nr.	(executable) Software metadata description	Please fill in this column
S1	Current software version	2.0
S2	Permanent link to executables of this version	`http://esbmc.org/gpu/`
S3	Legal Software License	Apache v2.0
S4	Computing Operating System	Ubuntu Linux OS
S5	Installation requirements & dependencies	GNU Libtool; Automake; Flex & Bison; Boost C++ Libraries; Multiprecision arithmetic library developers tools (libgmp3-dev package); SSL development libraries (libssl-dev package); CLang 3.8; LLDB 3.8; GNU C++ compiler (multilib files); libc6 and libc6-dev packages
S6	Link to user manual	`http://esbmc.org/gpu/`
S7	Support email for questions	`lucas.cordeiro@cs.ox.ac.uk`

Table 1: Software metadata (optional)

Current code version

Ancillary data table required for subversion of the codebase.

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	$v2.0$
C2	Permanent link to code/repository used for this code version	`https://github.com/ssvlab/esbmc-gpu`
C3	Legal Code License	GNU Public License
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	C++
C6	Compilation requirements, operating environments & dependencies	GNU Libtool; Automake; Flex & Bison; Boost C++ Libraries; Multi-precision arithmetic library developers tools (libgmp3-dev package); SSL development libraries (libssl-dev package); CLang 3.8; LLDB 3.8; GNU C++ compiler (multilib files); libc6 and libc6-dev packages
C7	Link to developer documentation	`http://esbmc.org/gpu`
C8	Support email for questions	`lucas.cordeiro@cs.ox.ac.uk`

Table 2: Code metadata (mandatory)