Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking

Phillipe A. Pereira
Federal University of Amazonas
apphillipe@gmail.com

Higo F. Albuquerque
Federal University of Amazonas
enghigo@gmail.com

Hendrio M. Marques
Federal University of Amazonas
hendriomm@gmail.com

Isabela S. Silva
Federal University of Amazonas
isabelassilva@gmail.com

Celso B. Carvalho
Federal University of Amazonas
celso@ufam.edu.br

Lucas C. Cordeiro
Federal University of Amazonas
lucascordeiro@ufam.edu.br

ABSTRACT

We present ESBMC-GPU, an extension to the ESBMC model checker that is aimed at verifying GPU programs written for the CUDA framework. ESBMC-GPU uses an operational model for the verification, *i.e.*, an abstract representation of the standard CUDA libraries that conservatively approximates their semantics. ESBMC-GPU verifies CUDA programs, by explicitly exploring the possible interleavings (up to the given context bound), while treating each interleaving itself symbolically. Experimental results show that ESBMC-GPU is able to detect more properties violations, while keeping lower rates of false results.

CCS Concepts

•**Software and its engineering** → **Model checking; Software verification;** *Formal software verification;* Parallel programming languages;

Keywords

Symbolic and Explicit Model Checking, GPU Kernels, CUDA Programs, MPOR

1. INTRODUCTION

The Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface model created by NVIDIA [1], which extends C/C++ and Fortran to create a computational model that aims to harness the computational power of Graphical Processing Units (GPUs) [2]. As in other programming languages, errors in CUDA programs eventually occur, in particular, array bounds, arithmetic overflow, and division by zero violations. Additionally, since CUDA is a platform that deals with parallel programming, specific concurrency errors related to data race and barrier divergence can be exposed due to the non-deterministic behaviour of the threads interleavings [3].

Here, we describe and evaluate an approach for verifying CUDA programs based on the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [4, 5, 6], named ESBMC-GPU, using an operational model, which is an abstract representation of the standard CUDA libraries (*i.e.*, the native application programming interface) that conservatively approximates their semantics. We describe the implementation of our operational model for the CUDA libraries, its preconditions and simulation features (*e.g.*, how the elements values of the libraries are stored), and how these are applied to verify CUDA applications. In contrast to previous attempts [3, 7, 8], we combine symbolic model checking, based on Bounded Model Checking (BMC) and Satisfiability Modulo Theories (SMT) techniques, with explicit state space exploration, similar to Cordeiro *et al.* [4]. In particular, we explicitly explore the possible interleavings (up to the given context bound), while we treat each interleaving itself symbolically with respect to a given property.

To prune the state-space exploration, we apply Monotonic Partial Order Reduction (MPOR) [9] to CUDA programs, which eliminates all redundant interleavings without missing any behavior that can be exhibited by the program. Since CUDA kernels typically manipulate one element of the array at a given position, the application of MPOR leads to substantial performance improvements. Thus, using operational models that simulate CUDA libraries, together with MPOR implementation in ESBMC-GPU, we achieve significant (correct) results of CUDA kernels verification, primarily when compared to other state-of-the-art GPU verifiers. Additionally, the proposed approach considers low-level aspects related to dynamic memory allocation, data transfer, memory de-allocation, and overflow, which are typically present in CUDA programs, but they are routinely ignored by most GPU verifiers; we thus provide a more precise verification than other existing approaches, considering soundness of data passed by the main program to the kernel, with the drawback of leading to a higher verification time.

We make four major contributions. First, we extend benefits of SMT-Based Context-Bounded Model Checking for CUDA programs, in the context of parallel programming for GPUs, to detect more failures than other existing approaches, while keeping lower rates of false results. Second, this work marks the first application of MPOR to CUDA programs to identify array accesses, which are independent, leading to sig-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2016,April 04-08, 2016, Pisa, Italy

Copyright 2016 ACM 978-1-4503-3739-7/16/04…$15.00

http://dx.doi.org/10.1145/2851613.2851830

nificant performance improvements. Third, we provide an effective and efficient tool implementation (ESBMC-GPU) to support the checking of several CUDA programs. Fourth, we provide an extensive experimental evaluation of our approach against GKLEE [8], GPUVerify [3], and PUG [7] using standard CUDA benchmarks, which are extracted from the literature [1, 3, 10]. Experimental results show that our present approach outperforms all existing GPU verifiers with respect to the number of correct results.

2. PRELIMINARIES

2.1 ESBMC

ESBMC is a context-bounded model checker for C/C++ programs based on SMT solvers, which allows the verification of single- and multi-threaded software with shared variables and locks [4, 5]. ESBMC can verify programs that make use of bit-level, arrays, pointers, structs, unions, memory allocation, and fixed-point arithmetic. It can reason about arithmetic overflows, pointer safety, memory leaks, buffer overflows, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions.

In ESBMC, the program is modelled as a state transition system $M = (S, R, s_0)$, which is extracted from the control-flow graph (CFG). S represents the set of states, $R \subseteq S \times S$ represents the set of transitions, and $s_0 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to *pc*. We identify each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system M, a safety property ϕ, a context bound C and a bound k, ESBMC builds a reachability tree (RT) that represents the program unfolding for C, k, and ϕ. We then derive a VC ψ_k^π for each given interleaving $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth k that is exhibited by π. ψ_k^π is given by the following logical formula:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{1}$$

I characterizes the set of initial states of M and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents executions of M of length i and ψ_k^π can be satisfied if and only if for some $i \leq k$ there exists a reachable state along π at time step i in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counterexample. A counterexample for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use this to check all paths. However, ESBMC combines symbolic model checking with explicit state space exploration; in particular, it explicitly explores the possible interleavings (up to the given context bound) while it treats each interleaving itself symbolically. ESBMC simply traverses the RT depth-first, and calls the single-threaded BMC procedure on the interleaving whenever it reaches an RT leaf node. It stops when it finds a bug, or has systematically explored all possible RT interleavings.

2.2 CUDA Programming Language

CUDA is a general-purpose parallel computing platform and was developed by NVIDIA to represent a programming model for GPUs [1, 10]. In the CUDA programming model, the kernel concept is used for a function that runs n copies at the same time in the GPU, where n is the product between the number of blocks and threads. A kernel is defined by a `__global__` specifier and uses the notation `kernel<<< B,T >>>`, where B and T are the number of blocks and threads per block, respectively. Each kernel runs in the GPU as thread and each thread receives an unique identifier (ID), which is formed by thread and block numbers. The thread ID is used to index its tasks (*i.e.*, memory positions and cooperation); and threads are typically organized by blocks in the GPU. Inside a block, the thread hierarchy is defined by a variable called *threadIdx*. This variable is a vector of three components, which allows the use of uni-, two-, and three-dimensional indexes [10].

Blocks can also be defined in three dimensions, where each dimension can be accessed by the *blockIdx* variable. This variable is also composed by three components that allow us to use uni-, two-, and three-dimensional blocks. The maximum number of threads per block depends on the hardware platform, but it usually ranges from 1024 to 2048 [10]. Blocks have a feature that permits them to be executed in any order; and they can also be allocated in any processor. As a result, a kernel may also be executed by multiple blocks, and the total number of threads represents the number of blocks multiplied by the number of threads per block.

In the CUDA programming model, the GPU is refereed as a *device* and the Central Processing Unit (CPU) is refereed as a *host*. *__device__* is a specifier for functions, which are executed and called only by the GPU, while *__host__* is a specifier for functions, which are executed and called only by the CPU. The data allocation in the *device* is carried out by the *host*, using the *cudaMalloc*, *cudaFree*, and *cudaMemcpy* functions; these are essential functions for CUDA programs, to transfer data from the *host* to the *device* and vice-versa.

2.3 Existing GPU Verifiers

GPUVerify [3] uses semantics to verify kernels (*synchronous, delayed visibility*), aiming to detect data race and barrier divergence. GPUVerify uses Boogie verification system [11] to generate verification conditions, which are solved by Z3 [12] or CVC4 [13] SMT solvers. GPUVerify accepts only kernel function as input, and it disregards main functions, which thus exposes incorrect results for verifying low-level aspects of CUDA programs.

SESA (Symbolic Executor with Static Analysis) [14] and GKLEE (GPU + KLEE) [8] are based on the *concolic* execution (concrete but symbolic) of CUDA programs, but they use different approaches to determine symbolic variables. While SESA performs an automatic evaluation, GKLEE needs inputs from the user to define those variables. SESA checks for real applications using original configuration of the number of threads and its focus is on data race detection, but it presents inconclusive results regarding access to memory positions. GKLEE supports checks related to barrier synchronization, functional correctness, performance,

and data race. SESA does not verify the main function, while GKLEE considers both kernel and main functions.

PUG (Prover of User GPU Programs) [7] analyzes kernels automatically using SMT solvers and it detects data race, barrier synchronization, and conflicts on shared memory. PUG faces problems with invariant derivation for loops, which can lead to incorrect results, and thus requires the user to provide those invariants. Problems are also found in arithmetic operations of pointers and advanced C++ features.

3. VERIFYING CUDA PROGRAMS

3.1 Operational Models for CUDA Libraries

In the present approach, operational models are developed to simulate the behavior of the CUDA libraries. In particular, our operational model consists of an abstract representation of a set of methods and data structures, which conservatively approximate the CUDA libraries semantics; every method simulates the library's real behavior, including pre- and post-conditions. Thus, the operational model contains only methods for verification, ignoring irrelevant calls (*e.g.*, screen-printing methods), where there is no relevant property to be checked in terms of software. As a result, our verification focuses on the operational model of the CUDA libraries, and how it is used to verify real-world CUDA programs; this simplifies significantly the model verification and consequently reduces the verification time. The operational model also includes built-in assertions, which check for specific properties (*e.g.*, division by zero, array and arithmetic overflow, pointer safety, and data races).

Figure 1 shows an example of an operational model developed for the *cudaMalloc* function, which abstracts the GPU memory hierarchy and accepts as input arguments, a pointer to allocate memory on the device (*i.e.*, *devPtr*) and the size in *bytes* needed for memory allocation. The function *malloc* represents the memory allocation on *device*, checking whether there is a successful allocation (lines 8 to 12). If so, the function returns `CUDA_SUCCESS`; otherwise, it returns an error `CUDA_ERROR_OUT_OF_MEMORY`. The variable *lastError* is global and stores the last `cudaError_t` value to be used in the *cudaLastError(). cudaMalloc()* has, as precondition, a positive memory size allocation; line 5 of Fig. 1 includes an assertion in which the size to be allocated must be greater than zero. If there is a violation in this precondition, then ESBMC-GPU returns an error message.

```
1  cudaError_t cudaMalloc(void ** devPtr,
2                         size_t size) {
3    cudaError_t tmp;
4    //pre-conditions
5    __ESBMC_assert(size > 0, ``Size must be
6    greater than zero'');
7    *devPtr = malloc(size);
8    if (*devPtr == NULL) {
9      tmp = CUDA_ERROR_OUT_OF_MEMORY;
10     exit(1);
11   } else {
12     tmp = CUDA_SUCCESS;
13   }
14   lastError = tmp;
15   return tmp;
16 }
```

Figure 1: *cudaMalloc* implementation.

Figure 2 presents the *cudaMemcpy*() function operational model. It checks, as precondition, the memory size to be copied (line 4). Two local variables *cdst* and *csrc* (lines 6 and 7) are used to receive arguments, which represent destination and source of the data copying. This model defines the number of *bytes* to be copied (line 8); the data copy is actually performed (in lines 9 and 10) between *device* and *host*. Finally, *cudaMemcpy* function returns value *CUDA_SUCCESS*.

Note that the operational models are implemented according to the functions operation, as described in the NVIDIA Programming Guide [10]. The behavior of these functions can be represented in C/C++ programming languages, using native functions that are already supported by ESBMC (*e.g.*, *malloc*, *free*, *assert*). As example, the *cudaMalloc* function operates similarly to the *malloc* function, which accepts as input argument the size of the variable to be allocated; the behavior of this function is in compliance with the C semantics. However, the conceptual difference for CUDA programs is that the memory allocation is carried out in the GPU, which is abstracted since neither hardware functions nor memory hierarchy are included into our operational model, as also done by [3, 7, 8]. The *cudaMemcpy* function is implemented similar to *Memcpy* function; the only difference is one additional parameter, which determines whether the operation is from *device* to *host* or vice-versa. *cudaFree* is just a small (program) routine that calls the *free* function.

```
1  cudaError_t cudaMemcpy(void *dst,
2        const void *src, size_t size,
3            enum cudaMemcpyKind kind) {
4    __ESBMC_assert(size > 0, ``Size must be
5   greater than zero'');
6    char *cdst = (char *) dst;
7    const char *csrc = (const char *)src;
8    int numbytes = count/(sizeof(char));
9    for (int i = 0; i < numbytes; i++)
10     cdst[i] = csrc[i];
11   lastError = CUDA_SUCCESS;
12   return CUDA_SUCCESS ;
13 }
```

Figure 2: *cudaMemcpy* implementation.

3.2 Monotonic Partial Order Reduction

To reduce the number of threads interleavings in CUDA programs, ESBMC-GPU implements the Monotonic Partial Order Reduction (MPOR), which was initially proposed by Kahlon *et al.* [9]. This algorithm classifies transitions inside a multi-threaded program as dependent or independent, which determines whether interleaving pairs always compute the same state, thus removing duplicate states in the RT [15]. For dependent transitions, MPOR considers possible thread execution orders to ensure that all program states are reached. If one transition is independent, then the MPOR algorithm considers only one order, because the program state is the same for other execution orders.

For CUDA programs, MPOR is applied to identify accesses to different positions in (shared) arrays. Typically, threads access unique positions in those (shared) arrays, which do not have dependency between them, thus allowing us to remove redundant states that are generated by the possible thread execution orders. Multiple accesses to specific memory positions in CUDA programs happen due to its concurrent nature, based on the linearized configuration of threads

and blocks [1]. The application of MPOR to that programs ensures a dramatic performance improvement.

As example, Fig. 3(a) shows the application of MPOR to a CUDA kernel, where the shared variable a is written in a position relative to the thread ID. Each node in the RT of Fig. 3 is represented as a tuple $\nu = (A_i, C_i, s_i)$ for a given time step i, where A_i represents the currently active thread; C_i represents the context switch number; and s_i represents the current state. Considering two threads, two possible interleavings can be observed, which result in the same program state (*i.e.*, v_2 and v_4 of Fig 3(a) show the same values for array a positions). MPOR thus identifies no dependency in the array accesses and disregards the redundant states, which are represented by dotted lines. In the kernel of Fig. 3(b), however, array a is of length 2 and two threads are executed; those two thread execution orders result in different program states. In particular, one interleaving shows an access out-of-bounds violation in array a (*i.e.*, v_4 of Fig 3(b)). The algorithm thus identifies the execution order dependency and two interleavings must be considered, leading to a property violation.

3.3 Two-threads Analysis

GPU architectures are composed by multiprocessors built upon processing elements (PE) sets [1, 10]. Those PEs are typically arranged in subgroups, which run in the same lock-step, ensuring that threads inside those PEs can synchronize using barriers, while threads from one subgroup run independently [3] to threads from another subgroup.

Similar to GPUVerify [3] (for checking race- and divergence-freedom) and PUG [7], we also reduce the CUDA program verification to two threads for improving verification time and avoiding the state-space explosion problem. Since CUDA kernels typically manipulate one element of the array, and for each element one thread is used, the two-threads analysis ensures that, errors (*e.g.*, data races) that are detected between two threads in a given subgroup, due to unsynchronized accesses to shared variables, are enough to justify the property violation in the program.

The two-threads analysis affects mostly the data race verification, where program states must be analyzed with respect to their possible threads interleavings, which lead to an execution order of statements that results in error. In our benchmarks, however, we observed a substantial improvement in performance considering only two threads, while keeping the number of true incorrect results at low rates.

3.4 Illustrative Example

The code fragment shown in Fig. 4 has 1 block and 2 threads, *i.e.*, $M = 1$ and $N = 2$, respectively. This CUDA program has a kernel (lines 3 to 5), which assigns thread's index values to an array passed as an input argument. The goal is to instantiate array positions, according to the thread index. Despite that, there is a mistake in the array index, as the value 1 is accidentally added to the thread index (in line 4). As shown in the main function, array positions are assigned with value 0 (line 11), and after the kernel call (line 14), it is expected by the programmer that $a[0] == 0, a[1] == 1$.

In this example, however, ESBMC-GPU detects an array out-of-bounds violation. Indeed, this CUDA program retrieves a memory region that has not been previously allocated, so that when $threadIdx.x = 1$, the program tries to access the position $a[2]$. Analysing the *cudaMalloc()* function operational model, there is a precondition that checks if the memory size to be allocated is greater than zero. Assertions check if the result matches the expected postconditions (line 16). The verification of this specific program produces 34 successful and 6 failed interleavings in ESBMC-GPU. One possible failed interleaving is represented by the threads executions $t0 : a[1] = 0$; $t1 : a[2] = 1$, where $a[2] = 1$ represents an incorrect access to the array index a.

```
1  #define M 1
2  #define N 2
3  __global__ void kernel(int *A) {
4    A[threadIdx.x + 1] = threadIdx.x;
5  }
6  int main(){
7    int *a; int *dev_a;
8    int size = N*sizeof(int);
9    a = (int*)malloc(size);
10   cudaMalloc((void**)&dev_a, size);
11   for (int i = 0; i < N; i++) a[i] = 0;
12   cudaMemcpy(dev_a,a,size,
13              cudaMemcpyHostToDevice);
14   ESBMC_verify_kernel(kernel, M, N, dev_a);
15   for (int i = 0; i < N; i++)
16     assert(a[i]==i);
17   ...
18 }
```

Figure 4: Fragment of a program to index *array*.

ESBMC-GPU and GKLEE are able to detect this array out-of-bounds violation, but GPUVerify and PUG fail to detect such violation, presenting a true incorrect (missed bug).

4. EXPERIMENTAL EVALUATION

4.1 Experimental Setup

This section describes experiments to investigate ESBMC-GPU performance for verifying CUDA programs. We also compare ESBMC-GPU to GKLEE [8], GPUVerify [3], and PUG [7]. In particular, we evaluate ESBMC-GPU's ability to verify 154 benchmarks[1], which are extracted from [3, 1, 10]; we added a *main* function to those benchmarks that do not contain it. The kernels typically exploit the support for: arithmetic operations, pointer assignment, *__device__* function calls, general C functions (*e.g.*, *memset*, *assert*), general CUDA functions (*e.g.*, *atomicAdd*, *cudaMemcpy*, *cudaMalloc*, *cudaFree*, *__syncthreads*), general libraries in CUDA (*e.g.*, curand.h, *curand_kernel.h*, *curand_mtgp32_host.h*) and the ability to work with variables *int*, *float*, *char* as well as type modifiers (*e.g.*, *long* and *unsigned*), pointers to that variables, function pointers, type definitions, and intrinsic CUDA variables (*e.g.*, *uint4*).

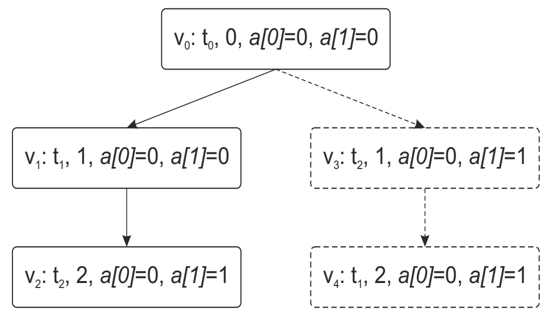
Our experiments answer two research questions: RQ1 (sanity check) which results does ESBMC-GPU obtain upon verifying benchmarks that compose the specified suite? RQ2 (comparison with other tools) what is ESBMC-GPU performance when compared to GKLEE, GPUVerify, and PUG?

To answer RQ1, ESBMC-GPU is executed with: `--force-malloc-success`, which considers that there is always enough memory available in the device; `--context-switch C`[2], which

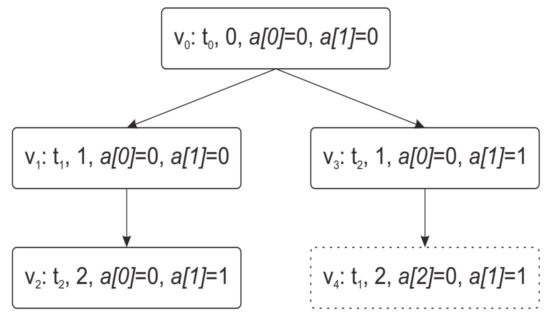
[1]ESBMC-GPU and benchmarks are available at http://esbmc-gpu.org
[2]The value of C ranges from 2 to 4 context switches.

```
1 __global__ void kernel1(int *a) {
2   a[threadIdx.x] = threadIdx.x;
3 }
```

```
1 __global__ void kernel2(int *a) {
2   if(a[1]==1){
3     a[threadIdx.x+2] = threadIdx.x;
4   }
5   else { a[threadIdx.x] = threadIdx.x; }
6 }
```



(a)



(b)

Figure 3: MPOR applied to a kernel with independent (Fig. 3(a)) and dependent (Fig. 3(b)) transitions.

considers a context-switch bound among all threads; and `-I libraries`, which specifies the library directory. We also replace the kernel call by the respective *ESBMC_verify_kernel* function using one block and two threads per block. As example, we call ESBMC-GPU as: `esbmc file.cu --force-malloc-success --context-switch 2 --I libraries`.

To answer RQ2, we apply GKLEE, GPUVerify and PUG to the ESBMC-GPU benchmarks suite. With GKLEE, we use two commands: *gklee-nvcc* and *gklee*. The first one checks the file to be verified, with the extension ".cu". When this command is executed, two new files are generated: a ".cpp" file and a runnable file (without extension). Then, the second command is used with the generated runnable file. We call GKLEE as: `gklee-nvcc file.cu; gklee file`.

To the verification of GPUVerify, the following modifications are required: (a) remove the main function; (b) check if the variable initialization performed by the main function is responsible for controlling some conditional declaration inside the kernel; if so, such variable must be initialized by `__requires()` function; (c) check if there is any assertion in the kernel; if so, this assertion must be replaced by `__assert()`; (d) check if there are specific functions to C/C++ libraries; if so, they must be removed, as they are not supported by GPUVerify. To run GPUVerify, two options must be used: `--gridDim=M` and `--blockDim=N` to assign the number of blocks and threads (per block), respectively. We call GPUVerify as: `gpuverify file.cu --blockDim=2 --gridDim=1`.

Some additional changes to the benchmarks are necessary to use PUG: (a) the file extension is changed from ".cu" to ".c"; (b) given that PUG is unable to verify main functions, these must be removed, in order to keep the kernel function only; (c) PUG's proprietary libraries *my_cutil.h* and *config.h* must be called inside the ".c" file. The first library has definitions of structs, qualifiers, and datatypes. The second one defines the number of blocks and threads (per block); (d) The kernel function's name has to be renamed to "kernel". As example, we call PUG as: `pug kernel.c`.

Additional options for ESBMC-GPU and GPUVerify are necessary to check for data races and array out-of-bounds, respectively. All experiments were conducted on an otherwise idle Intel Core i7-4790 CPU 3.60 GHz, 16 GB of RAM, and Linux OS. All times given are wall clock time in seconds as measured by the unix time command.

4.2 Experimental Results

Table 1 shows the results of ESBMC-GPU, GKLEE, GPUVerify, and PUG; each row means: tool name (Tool); total number of benchmarks in which the program was analyzed to be free of errors (True Correct); total number of benchmarks in which the error in the program was found and an error path was reported (False Correct); total number of benchmarks in which the program had an error but the verifier did not find it (True Incorrect); total number of benchmarks in which an error is reported for a program that fulfills the specification (False Incorrect); total number of benchmarks which are not supported (Not Supported); verification time, in seconds, for all benchmarks (Time).

ESBMC-GPU is able to correctly verify 75.9% of the benchmarks, GKLEE 70.1%, GPUVerify 57.7%, and PUG 35.7%. Note that ESBMC-GPU produces 2 true incorrect results, while GKLEE produces 14, GPUVerify produces 10, and PUG produces 7. With ESBMC-GPU, this result is due to incorrectly detected assertion (1) and null pointer access (1). With GKLEE, errors are due to failure in detecting data race (10), unsuccessful detection of attempts to modify constant memory (2), incorrect detected assertion (1), and null pointer access (1). GPUVerify does not detect data race (5), and it is unable to return values of the *__device__* function to the kernel, where they are called (3); additionally, it does not detect array bounds violation (2). PUG does not detect access to null pointer (1), data race (4), array bounds violation (1), and incorrect detected assertion (1).

ESBMC-GPU generated 3 false incorrect results, due to assertions included in the kernel, which should return true, but it fails (2), and the partial coverage of the *cudaMalloc* function for copies of float-type variables (1). GKLEE generated 8 false incorrect results, which are caused by incorrectly detected assertions (5), data-races (1), array-bounds (1) and solver call failure (1). GPUVerify generated 8 false incorrect results, due to incorrect detected assertion (2) and data-races (6). PUG produces 10 false incorrect results due to data races incorrectly detected.

Table 1: Results of ESBMC-GPU, GKLEE, GPUVerify, and PUG

Result/Tool	**ESBMC-GPU**	**GKLEE**	**GPUVerify**	**PUG**
True Correct	55	53	**59**	39
False Correct	**62**	55	30	16
True Incorrect	**2**	14	10	7
False Incorrect	**3**	8	8	10
Not Supported	32	**24**	47	82
Time (s)	657	125	148	**11**

ESBMC-GPU had 32 benchmarks that were not supported. This is related to constant memory access (3), use of CUDA's specific functions (*__mul24()*, *__threadfence()*) (2), the use of CUDA's specific libraries (*curand.h* and *math_functions.h*) (9), and the use of pointers to functions, structures, and char type variables used as kernel call arguments (18). GKLEE has 24 benchmarks that were not supported, which are due to the use of the *memset* function, which is a specific function of the C/C++ libraries (3); incorrectly detected assertions (3); pointers, either used as kernel arguments or as in any other parts of the CUDA program (9); specific CUDA libraries *curand.h* (7), and *switch-case* functions (2).

GPUVerify did not support 47 benchmarks. Since it does not support main functions, this explains most cases (36); it also does not support the use of *memset* function (3), while other cases are explained by the absence of support to function pointers, either as kernel function arguments or as in any other part of the CUDA program (8). PUG does not support 82 benchmarks. As GPUVerify, PUG does not verify main functions and this explains most unsupported cases (31), while others are explained by the lack of support to *__syncthreads* function (12), function pointers (9), and the *curand.h* library (7); additionally, PUG does not support the use of unsigned type modifier as argument to the function *atomicAdd* (6); changes in variables stored in constant memory (3), and inability to handle structs (2), variables with *__device__* qualifier (2), and *size_t* type (1), in addition to other cases that PUG aborted by returning a false null pointer access (7) or because it did not recognize the NULL identifier (2).

MPOR produced an improved performance of approximately 80% in our benchmarks; it reduced the verification time from 16 to 3 hours. With two-threads analysis, we further reduced the verification time from 3 hours to 650 seconds. However, ESBMC-GPU still takes more (verification) time than all other tools. This is due to the actual execution of the threads interleavings (which combines symbolic model checking with explicit state space exploration), while in GPUVerify the analysis is fully symbolic, performed only in the kernel level, without considering threads interleavings with the main thread. PUG lower verification time is due to the two-threads analysis and because it does not perform the *main function* verification. GKLEE presents a low verification time due to its directed state/path reduction method.

5. CONCLUSIONS

ESBMC-GPU is able to verify CUDA programs using SMT-based context-bounded checking and operational models, which recognize CUDA directives and further simplify the verification model. This work marks the first application of symbolic model checking with explicit state space exploration using MPOR for verifying CUDA programs. In particular, MPOR led to 80% of performance improvement in our benchmarks. ESBMC-GPU also presents an improved ability to detect array out-of-bounds and data race violations if compared to GKLEE, GPUVerify, and PUG. Additionally, ESBMC-GPU provides fewer incorrect results than all other existing GPU verifiers. Experimental results show that ESBMC-GPU presents a successful verification rate of 75.9%, compared to 70.1% of GKLEE, 57.7% of GPUVerify, and 35.7% of PUG. For future work, we will detect barrier divergence, improve support to argument types of kernel functions, and apply techniques to reduce the number of threads interleavings.

Acknowledgements. This research project was supported by the Technology Development Institute (INDT) and by CNPq 475647/2013-0 (UNIVERSAL).

6. ADDITIONAL AUTHORS

Vanessa de S. Santos (Federal University of Amazonas) and Ricardo dos S. Ferreira (Federal University of Viçosa).

7. REFERENCES

[1] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. John Wiley and Sons, Inc., 2014.

[2] D. Kirk and W. Hwu. *Programming Massively Parallel Processors*. Elsevier Inc., 1st edition, 2010.

[3] A. Betts *et al.* GPUVerify: A Verifier for GPU Kernels. In *OOPSLA*, pp. 113–132, 2012.

[4] L. Cordeiro and B. Fischer. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In *ICSE*, pp. 331–340, 2011.

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

[6] M. Ramalho *et al.* SMT-Based Bounded Model Checking of C++ Programs. In *ECBS*, pp. 147–156, 2013.

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

[8] G. Li *et al.* GKLEE: Concolic Verification and Test Generation for GPUs. In *PPoPP*, pp. 215–224, 2012.

[9] V. Kahlon, C. Wang, and A. Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*, *LNCS* 5643, pp. 398–413, 2009.

[10] NVIDIA. *CUDA C Programming Guide*. NVIDIA Corporation, v7.0 edition, 2015.

[11] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie Verification Debugger. In *SEFM*, *LNCS* 7041, pp. 407–414, 2011.

[12] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, *LNCS* 4963, pp. 337–340, 2008.

[13] C. Barrett *et al.* CVC4. In *CAV*, pp. 171–177, 2011.

[14] P. Li, G. Li, and G. Gopalakrishnan. Practical Symbolic Race Checking of GPU Programs. In *SC*, pp. 179–190, 2014.

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