CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–25
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

SMT-Based Context-Bounded Model Checking for CUDA Programs

P. A. Pereira[1], H. F. Albuquerque[1], I. S. da Silva[1], H. M. Marques[1], F. R. Monteiro[1], R. dos S. Ferreira[2], and L. C. Cordeiro[1*]

[1] *Faculty of Technology, Federal University of Amazonas, Amazonas, Brazil*
[2] *Department of Informatics, Federal University of Viçosa, Minas Gerais, Brazil*

SUMMARY

We present ESBMC-GPU tool, an extension to the Efficient SMT-Based Context-Bounded Model Checker (ESBMC), which is aimed at verifying Graphics Processing Unit (GPU) programs written for the Compute Unified Device Architecture (CUDA) platform. ESBMC-GPU uses an operational model, *i.e.*, an abstract representation of the standard CUDA libraries, which conservatively approximates their semantics, in order to verify CUDA-based programs. It then explicitly explores the possible interleavings (up to the given context bound), while treats each interleaving itself symbolically. Additionally, ESBMC-GPU employs the monotonic partial order reduction and the two-thread analysis to prune the state space exploration. Experimental results show that ESBMC-GPU can successfully verify 82% of all benchmarks, while keeping lower rates of false results. Going further than previous attempts, ESBMC-GPU is able to detect more properties violations than other existing GPU verifiers due to its ability to verify errors of the program execution flow and to detect array out-of-bounds and data race violations. Copyright © 2016 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: model checking; GPU; CUDA programs; formal verification;

1. INTRODUCTION

Compute **U**nified **D**evice **A**rchitecture (CUDA) is a parallel computing platform and **A**pplication **P**rogramming **I**nterface (API) model created by NVIDIA [1], which extends C/C++ and Fortran, in order to create a computational model that aims to harness the computational power of **G**raphical **P**rocessing **U**nits (GPUs) [2]. As in other programming languages, errors in CUDA programs eventually occur, in particular, due to array out-of-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].

*Correspondence to: 3000 Av. Rodrigo Otávio, 69077–000, Electronic and Information Research Center at Federal University of Amazonas, Manaus, Amazonas, Brazil.
E-mail: lucascordeiro@ufam.edu.br

Here, we describe and evaluate an approach for verifying CUDA programs based on the **E**fficient **S**MT-Based Context-**B**ounded **M**odel **C**hecker (ESBMC) [4, 5, 6], named as ESBMC-GPU, using 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. We describe the implementation of COM, its pre- and postconditions, and simulation features (*e.g.*, how elements are manipulated), and how these are applied to verify CUDA applications. In contrast to previous attempts [3, 7, 8, 9], we combine symbolic model checking, based on **B**ounded **M**odel **C**hecking (BMC) [10] and **S**atisfiability **M**odulo **T**heories (SMT) [11] 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 w.r.t. a given property.

To prune the state-space exploration, we apply **M**onotonic **P**artial **O**rder **R**eduction (MPOR) [12] to CUDA programs, which eliminates redundant interleavings without missing any behavior that can be exhibited by the program. We have modified the MPOR algorithm to identify transitions between threads that accessed different memory locations in the same array. Since CUDA kernels typically produce regular and independent access to explore the benefits of the GPU execution model, the application of MPOR routinely leads to substantial performance improvements in most benchmarks. 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 [3, 8, 7, 9]. Additionally, the present approach considers low-level aspects related to dynamic memory allocation, data transfer, memory deallocation, and overflow. Such violations are typically present in CUDA programs, however, they are routinely ignored by most GPU verifiers. Thus, we 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.

1.1. Contributions

We make four major contributions:

- 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; although SMT-based context-bounded model checking is not a novel technique, we have not seen in the literature its application to verify CUDA programs.
- this work marks the first application of MPOR to CUDA programs to identify array accesses, which are independent, leading to significant performance improvements.
- we provide an effective and efficient tool implementation (ESBMC-GPU) to support the checking of several CUDA programs. ESBMC-GPU tool and all benchmarks used during the evaluation process are available at `http://esbmc-gpu.org`.
- we provide an extensive experimental evaluation of our approach against GKLEE [8], GPUVerify [3], PUG [7], and CIVL [9] using standard CUDA benchmarks, which are extracted from the literature [1, 3, 13]. Experimental results also show that our present approach outperforms all existing GPU verifiers with respect to the number of correct results.

This paper extends our previous work [14, 15]. The version of ESBMC-GPU described and evaluated here has been optimized and extended. It now expands our operational model to support more CUDA libraries (*e.g.*, math functions) and includes new SMT solvers as back-end for ESBMC-GPU (*e.g.*, Boolector, Yices, MathSAT, and CVC4). We also provide additional details about our operational model, the application of MPOR to CUDA programs, and the two-thread analysis. We have also significantly expanded the experimental basis, including a new GPU verifier (CIVL), and evaluate ESBMC-GPU with (respectively, compare it against) the most recent stable versions of the GPU verifiers.

1.2. Organization of this work

The remainder of the paper is organized as follows: In Section 2, we first give a brief introduction to the CUDA programming language, existing verifiers for GPU, and the ESBMC verifier. In Section 3, we describe our operational models for CUDA libraries, the application of MPOR to CUDA programs, and our two-threads analysis. In Section 4, we present the results of our experiments using several publicly available benchmarks. In Section 5, we conclude and describe future work.

2. PRELIMINARIES

2.1. CUDA Programming Language

CUDA is a general-purpose parallel computing platform (developed by NVIDIA) to represent a programming model for GPUs [1, 13]. In the CUDA programming model, the kernel concept is used for a function that runs n copies concurrently 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 [13].

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 CUDA programs 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 [13]. In addition, 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 **C**entral **P**rocessing **U**nit (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. *__global__* is a function specifier that operates as an entry point for executing the kernels. The data allocation in the *device* is carried out by the *host*, using the *cudaMalloc*, *cudaFree*, and *cudaMemcpy* functions. Importantly, these are essential functions for CUDA programs, to transfer data from the *host* to the *device* and vice-versa.

2.2. *Existing GPU Verifiers*

GPUVerify [3] proposes a new operational semantics, known as **S**ynchronous **D**elayed **V**isibility (SDV) semantics to verify kernels, which aims to detect data race and barrier divergence. It uses the Boogie verification system [16] to generate verification conditions, which are solved by Z3 [17] or CVC4 [18] SMT solvers. GPUVerify accepts only kernel function as input, and it disregards the main function, which thus exposes incorrect results for verifying low-level aspects of CUDA programs.

Symbolic **E**xecutor with **S**tatic **A**nalysis (SESA) [19] and GPU + KLEE (GKLEE) [8] are verifiers based on the *concolic* (concrete, but symbolic) execution of CUDA programs; nevertheless they use different approaches to determine symbolic variables. While SESA performs an automatic evaluation, GKLEE needs inputs from the user to define those variables. On the one hand, SESA checks 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. On the other hand, GKLEE supports checks related to barrier synchronization, functional correctness, performance, and data race. Importantly, SESA does not verify the main function, while GKLEE considers both kernel and main functions.

Prover of **U**ser **G**PU Programs (PUG) [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. In addition, problems are also found in arithmetic operations of pointers and advanced C++ features.

CIVL [9] is a framework for static analysis and concurrent program verification, which supports MPI, POSIX, OpenMP, and CUDA. CIVL recognizes programs written in C++11, CUDA–C, and CIVL–C, which is an intermediate language that represents ANSI-C programs based on the aforementioned libraries. CIVL is the tool that is closer to ESBMC-GPU in terms of verification techniques, verifying the concurrent program with partial order reduction, in order to eliminate unnecessary interleavings. The verified properties include: user-specified assertions, deadlocks, memory leaks, invalid pointer dereference, array out-of-bounds, and division by zero. The support of CIVL for CUDA libraries is still in progress, being effective and efficient in programs with kernels that do not use many CUDA functionalities.

2.3. *ESBMC*

ESBMC is an open source, permissively licensed, award-winning context-bounded model checker for C/C++ programs based on SMT solvers [4, 5, 20, 21]. It is able to verify single- and multi-threaded software with shared variables and locks using the Phtread/POSIX. In particular, ESBMC is one of the most prominent BMC tools for verifying ANSI-C programs according to the last editions of the International **Comp**etition on **S**oftware **V**erification (SV-COMP) [22, 23]. ESBMC can verify programs that contain bit-level, arrays, structs/unions, pointers, dynamic memory allocation, as well as fixed-point arithmetic. It can reason about arithmetic overflows, pointer safety, memory leaks, array out-of-bounds, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions.

In ESBMC, ANSI-C/C++ programs are typically represented as a state transition system $M = (S, R, s_0)$, which are extracted from the **C**ontrol-**F**low **G**raph (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*. Each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} is identified with a logical formula $\gamma(s_i, s_{i+1})$, which captures constraints on the corresponding values of the program counter and the program variables.

In particular, consider the transition system *M*, a safety property ϕ, a context bound C, and a bound k, ESBMC constructs a **R**eachability **T**ree (RT), which represents the program unfolding for C, k, and ϕ. A **v**erification **c**ondition (VC) ψ_k^π is then derived for each given interleaving $\pi = \{\nu_1, \ldots, \nu_k\}$, such that ψ_k^π is satisfiable *iff* ϕ has a counterexample of depth *k*, which is exhibited by π. ψ_k^π is given as follows

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

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 the program variables values can be extracted, in order to construct a counterexample. A counterexample for a particular 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, one 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.

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.

2.3.1. Software Architecture Fig. 1 shows the ESBMC architecture. Rounded white rectangles represent input and output; Squared gray rectangles represent the verification steps.

ESBMC offers two front-ends to parse and generate the **I**ntermediate **R**epresentation (IR) of the program. On the old front-end, based on one from CBMC [24], ESBMC would first parse the program using flex/bison [25] to generate an **A**bstract **S**yntax **T**ree (AST), which is converted to IR. The IR is then type-checked, which is language-specific. During the type-checking, the code is statically analysed, including assignment checks, type-cast checks, pointer initialisation checks, function call checks and template instantiation.

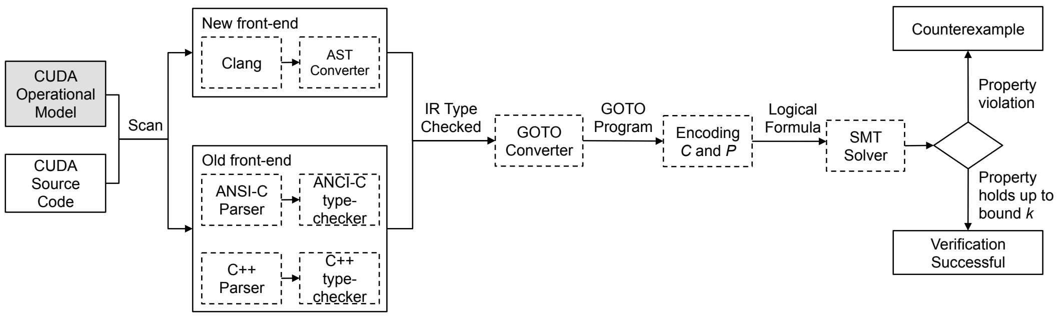


Figure 1. ESBMC architecture.

The new front-end offers a much simpler and stronger alternative, but it does not support C++ yet. In particular, the new front-end uses clang [26] to generate an AST, which is then converted to an IR. Clang is a state-of-art compiler for C/C++/ObjectiveC/ObjectiveC++ that it is already heavily used in industry [27]. The compiler offers an industry level static analyser and an API to access and traverse its internal AST, which is used by ESBMC to generate its IR. By using clang, we avoid the need to maintain our own front-end (a real challenge, given that the ANSI-C and C++ standards are rapidly evolving nowadays), and can focus on the main objective of the tool, the formal verification of a program. ESBMC currently can only convert the AST generated for ANSI-C programs.

The subsequent steps are the same as already described us in previous papers [5]. Regardless of the chosen front-end, the output is the IR that will be used by the GOTO converter to generate the GOTO program, which is a simplified version of the original program. The symbolic execution executes the GOTO program (unrolling loops up to bound *k*) and converts the expressions to **S**tatic **S**ingle **A**ssignments (SSA) [28]. Finally, two sets of quantifier-free formula are created based on the SSA expressions (*C* for the constraints and *P* for the properties), and used as input for an SMT solver that will produce a counterexample if there is a violation of a given property, or an unsatisfiable answer (*i.e.*, verification successful) if the property holds.

3. VERIFYING CUDA PROGRAMS

3.1. Operational Models for CUDA Libraries

In order to correctly support CUDA functionalities, operational models are developed to reliably simulate the behavior of the CUDA libraries. Such approach was previously attempted in the formal verification of C++ programs [6], Qt-based applications [29, 30], and Android mobile applications [31, 32]. 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 by means of assertions to ensure the correct operation. 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 specific properties (*e.g.*, array out-of-bounds, arithmetic overflow, pointer safety, and data races). As shown in Fig. 2, the operational model is passed to ESBMC's front-end together with the source code to be verified, in order to produce reliable AST with all relevant information for a trustful verification process.

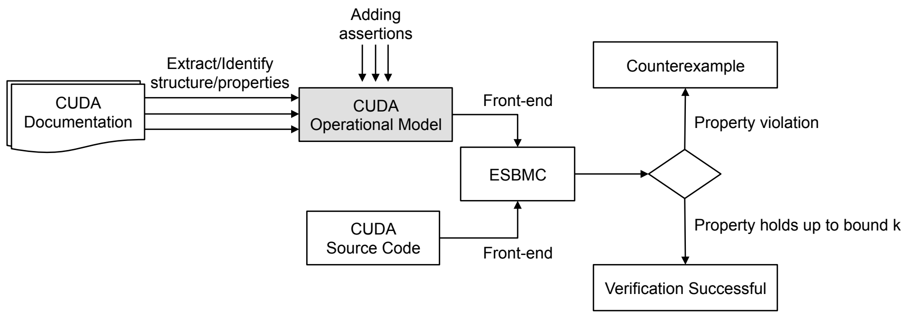


Figure 2. Link between CUDA operational model and ESBMC.

In our CUDA operational model (COM), methods, data types, qualifiers, and CUDA directives were implemented in C++. As an example, Figure 3 shows the implementation of the data type `dim3`, which is represented by a `struct` with three variables: x, y, and z. The `dim3` model has a default constructor, a constructor that receives a variable `uint3` as parameter (this variable is modelled by a `struct` with three `uint` variables), and a constructor that is initialized by a `dim3` variable. In addition to the (typical) primitive data types of ANSI-C/C++, COM supports specific data types of CUDA (*e.g.*, `char1`, `short2`, and `float2`).

```
struct __dim3
{
  unsigned int x, y, z;
  __dim3(unsigned int vx=1, unsigned int vy=1, unsigned int vz=1){
    x=vx; y=vy; z=vz;
  }
  __dim3(uint3 v){
    x=v.x; y=v.y; z=v.z;
  }
  __dim3(__dim3 d){
    x=d.x; y=d.y; z=d.z;
  }
  operator uint3(void){
    uint3 t; t.x=x; t.y=y; t.z=z;
  return t;
  }
};
typedef struct __dim3 dim3;
```

Figure 3. `dim3` implementation.

COM also models methods typically used to develop real CUDA applications. In this sense, we support most CUDA driver and runtime APIs; in particular, ESBMC-GPU supports the CUDA call kernel (*device_launch_parameters* and *vector_types*), driver_types (*math_functions*, *cuda_runtime_api*, and *host_definitions*), sm_atomic_functions (*vector_types*), and curand_kernel (*curand*). Figure 4 shows the CUDA operational model organization (internally) implemented in ESBMC-GPU.

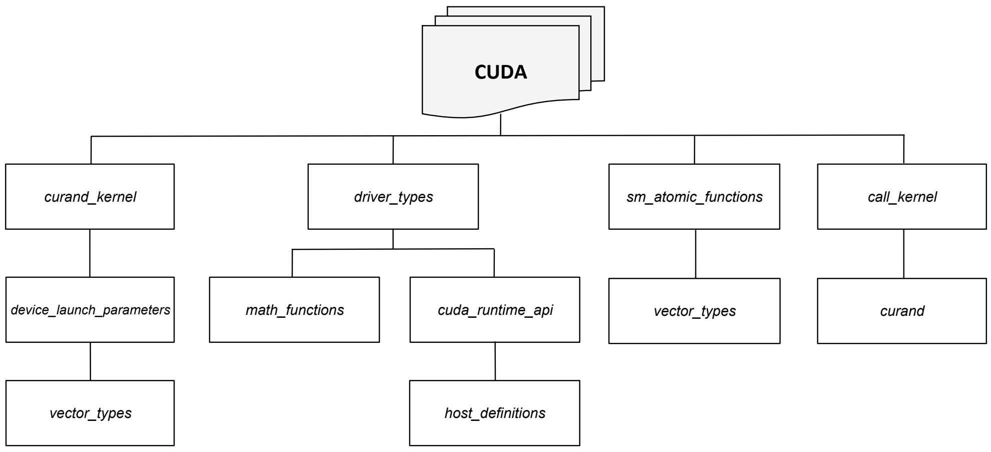


Figure 4. CUDA Operational Model Organization.

Here, we describe three essential functions in CUDA programs: *cudaMalloc*, *cudaMemcpy*, and *cudaFree*. Algorithm 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 (*i.e.*, $size$). In order to allocate memory for *devPtr* pointer (in step 3), this algorithm uses the *malloc* function, which represents the memory allocation on *device*, checking whether there is a successful allocation (in step 4). If so, the function returns `CUDA_SUCCESS`; otherwise, it returns an error specified by `CUDA_ERROR_OUT_OF_MEMORY`. The variable *lastError* is global and stores the last `cudaError_t` value to be used in the *cudaLastError()*. In addition, *cudaMalloc()* has, as precondition, a positive memory size allocation; step 2 of Algorithm 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, specifying the error together with the counterexample. As postcondition, we check whether the memory has been successfully allocated (in step 7).

Algorithm 1 Algorithm of *cudaMalloc* operation.

```
1: function CUDAMALLOC(void ** devPtr, size_t size)
2:     Check precondition: $size$ must be greater than zero
3:     Allocate a block of memory, which its magnitude is equal to $size$, for the $devPtr$ pointer
4:     If $*devPtr$ is equal to NULL, go to step 5, otherwise go to step 6
5:     Assign CUDA_ERROR_OUT_OF_MEMORY status to the global variable $lastError$
6:     Assign CUDA_SUCCESS status to the global variable $lastError$
7:     Check postcondition: $devPtr$ must be different than NULL
8:     Return $lastError$ and terminates
9: end function
```

Algorithm 2 presents the $cudaMemcpy()$ function operational model. It checks, as precondition, the memory size to be copied (in step 2). Then, two local variables named as $cdst$ and $csrc$ are used to receive arguments, which represent destination and source of the data copying (in steps 3 and 4). This model defines the number of *bytes* to be copied (in step 5); the data copy is actually performed (in step 6) between *device* and *host*. As postcondition, we check whether $cdst$ and $csrc$ contain the same data (in step 7). Finally, *cudaMemcpy* function returns $CUDA\_SUCCESS$ value.

Algorithm 2 Algorithm of *cudaMemcpy* operation.

```
1: function CUDAMEMCPY(void *dst, const void *src, size_t size, cudaMemcpyKind kind)
2:     Check precondition: $size$ must be greater than zero
3:     Initialize $cdst$ variable with $dst$'s content
4:     Initialize $csrc$ variable with $src$'s content
5:     Initialize $numbytes$ variable with the number of bytes to be copied (i.e., $size$)
6:     Copy $numbytes$ positions from $csrc$ to $cdst$
7:     Check postcondition: $cdst$ and $csrc$ must contain the same data
8:     Assign CUDA_SUCCESS status to the global variable $lastError$
9:     Return $lastError$ and terminates
10: end function
```

Similarly to *cudaMalloc* and *cudaMemcpy*, we also provide the implementation of the *cudaFree* function. Algorithm 3 shows the operational model, where a pointer to a variable is passed as an input argument so that the allocated memory is released. As precondition, we check whether the pointer $devPtr$, which points to the memory block previously allocated with *malloc*, is not NULL in order to avoid double free. Additionally, as we can see (in step 3), the *cudaFree* function is referenced to the *free* function of the ANSI-C programming language, however, in this case the memory is freed by the GPU. Importantly, this situation does not affect the verification results, because the memory allocation and deallocation model is represented by a tuple in the background theories of the (supported) SMT solvers, and properties that are analyzed do not take into account the memory hierarchy [33]. Finally, we assign *CUDA_SUCCESS* to *lastError*, which has the same functionality as in *cudaMalloc*.

Algorithm 3 Algorithm of *cudaFree* operation.

```
1: procedure CUDAFREE(void *devPtr)
2:     Check precondition: $devPtr$ must be different than NULL
3:     Call $free$ ANSI-C function to deallocate the memory from $devPtr$ pointer
4:     Assign CUDA_SUCCESS status to the global variable $lastError$
5:     Return $lastError$ and terminates
```

Note that the operational models are implemented according to the NVIDIA Programming Guide [13]. The behavior of those functions can be represented in C/C++ programming languages using native functions, which are already supported by ESBMC (*e.g.*, *malloc*, *free*, *assert*). The soundness proof for those native functions supported by ESBMC can be found in Cordeiro *et al.* [5]. In particular, 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 ANSI-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.

3.1.1. Correctness of the CUDA Operational Model (COM) The idea of making an environment operational model to perform model checking of real applications has been done before [29, 30, 31, 32], and the correctness of such model to trust in the verification results is actually the major issue. Consequently, the usefulness of our approach relies on the fact that COM correctly represents the original CUDA libraries. In that sense, all developed COM modules were manually verified and exhaustively compared to the original ones to ensure the same behavior; they all contain precondition and postcondition to ensure that a (given) predicate holds before and after the execution of a (given) function, respectively.

Additionally, although COM is a new implementation, it consists in (reliably) constructing a simplified model of the related CUDA libraries, using the same language and by means of the original code and documentation (pre- and postconditions are tested using assertions within the code itself), which thus tends to decrease the resulting number of errors. Note further that the behavior of the CUDA libraries functions are actually represented in C/C++ programming languages using native functions (*e.g.*, *malloc*, *free*, *assert*). The soundness proof for those native functions, which are already supported by ESBMC, can be found in Cordeiro *et al.* [4, 5]. Although proofs regarding the soundness of the entire COM could be carried out, it represents a hard task due to the (adopted) memory model [34]. To further improve the correctness of COM, conformance testing could be employed in practice [35, 36]; however, that option is not available in the present case, although it is an interesting possibility for future work.

3.2. *Modeling Kernel with Pthread Functions*

The ESBMC's architecture is designed to handle concurrent programs in the C/C++ programming languages using the Pthread/POSIX library [38]. Thus, ESBMC's verification model is guided by the processing method used by the CPU using that library, where thread instructions can interleave to form (different) execution paths. To apply this methodology to CUDA kernel verification, it is necessary to apply code transformations to the kernel call using ESBMC's intrinsic functions. In particular, the kernel call provides the thread/block configuration in a CUDA program; the parameters used by the (intrinsic) function are obtained from the same *struct* in that kernel call. As a result, the first step to verify a CUDA kernel is to create a new function *ESBMC_verify_kernel* using templates, in order to support different data types and parameters.

Algorithm 4 Algorithm of *ESBMC_verify_kernel* operation.

```
1: procedure ESBMC_VERIFY_KERNEL(RET *kernel, BLOCK blocks, THREAD threads, T1 arg1, T2 arg2, T3 arg3)
2:     Check precondition: $kernel$ must be different than NULL, $blocks$ and $threads$ must be greater than zero
3:     Initialize $gridDim$ variable with threads' dimension using $dim3$ function
4:     Initialize $blockDIM$ variable with blocks' dimension using $dim3$ function
5:     Call $ESBMC\_verify\_kernel\_wta$ function in order to determine the type of the $*kernel$
6:     Call $pthread\_join$ ANSI-C function to each GPU thread and terminates
```

Algorithm 4 shows the implementation of *ESBMC_verify_kernel* function, which supports six input parameters. The *kernel* parameter is a pointer to the original program kernel function, the *blocks* and *threads* parameters receive the blocks and threads configuration, which can be of type *int* or *dim3*, and the *arg1*, *arg2*, and *arg3* parameters correspond to values sent to the (respective) function. As preconditions, we check whether $kernel$ is a valid pointer and $blocks$ and $threads$ are greater than zero. Internally, there are two variables, *gridDim* (step 3) and *blockDim* (step 4), which receive the output of the *dim3* constructor used to configure the blocks and threads dimension. The function *ESBMC_verify_kernel_wta* (in step 5) particularizes the argument type to *int* and determines the type of the kernel function pointer. Lastly, there is a loop (in step 6), which synchronizes by means of the *pthread_join* function, the threads that are created at the end of the kernel process. The *ESBMC_verify_kernel* function is implemented to operate with a real number of threads in the program. However, with the reduction to the two-threads analysis (see Section 3.4), the respective loop in step 6 is bounded to two unwindings.

Algorithm 5 Algorithm of *ESBMC_verify_kernel_wta* operation.

```
1: procedure ESBMC_VERIFY_KERNEL_WTA(void *(*kernel)(int*,int*,int*), int blocks, int
   threads, void *arg1, void *arg2, void *arg3)
2:     Allocate a block of memory for the $threads\_id$ pointer, in order to store thread's ids
3:     Allocate a block of memory for each component of $dev\_three$
4:     Check precondition: $threads\_id$ and $dev\_three$ must be different than NULL
5:     Assign $arg1$, $arg2$, and $arg3$ to each component of $dev\_three$, respectively
6:     Assign $*kernel$ to the function pointer component of $dev\_three$
7:     Call $assignIndexes$ function
8:     Call $pthread\_create$ ANSI-C function to create each GPU thread and terminates
```

Algorithm 5 shows the reasoning behind the *ESBMC_verify_kernel_wta* function, which is responsible for translating GPU threads into POSIX threads. The *threads_id* pointer (in step 2) stores the thread IDs of *pthread_t* type, the *dev_three* variable (in steps 3 to 6) is an instance of *struct* and stores a pointer to the kernel function, and the *assignIndexes* function (in step 7) calculates the thread position in the grid using the ID of a thread (pthread) and a configuration of blocks and threads in the CUDA program; such values are stored in a *dim3* vector and are statically calculated to reduce paths generated during the verification. Lastly, there is a loop (in step 8), where threads are created by means of *pthread_create* function call and the function that will correspond to the thread is called via *ESBMC_execute_kernel_t*.

Algorithm 6 Algorithm of *ESBMC_execute_kernel_t* operation.

```
1: procedure ESBMC_EXECUTE_KERNEL_T(void *args)
2:     Check precondition: $args$ must be different than NULL
3:     Call function pointer component of $dev\_three$ (i.e., $dev\_three.func$) and terminates
```

Algorithm 6 shows the implementation of *ESBMC_execute_kernel_t* function. It checks the original kernel function using its parameters, which are passed as arguments by the *struct dev_three*. ESBMC-GPU then verifies this function and interleaves its possible execution paths using its native functions.

3.2.1. Thread Synchronization Note that in our proposed approach, each CUDA thread is actually mapped to a thread using the internal representation of ESBMC (via Pthread library) [4]. In ESBMC, a thread t is a sublist of commands between *begin_thread* and *end_thread*, which represent just scoping constructs and do not contribute to the expansion of the RT. Threads are created via asynchronous procedure calls (*start_thread*), which return an integer that can be used as thread identifier for synchronization (*join_thread*); hence, dynamic thread creation is allowed. Thus, the thread representation of ESBMC follows a similar approach to the official CUDA scheduler [1, 2, 3].

One feature of CUDA related to thread synchronization supported by ESBMC-GPU is the `__syncthreads()` intrinsic function, which specifies synchronization points in the GPU kernel. Such alternative guarantees that all threads in a certain block wait before any of them is allowed to proceed [1]. As described in Algorithm 7, we implement this feature in our operational model using Pthread functions. First of all, a $count$ global variable is initialized to control how many times the `syncthreads` function is actually called. In its first call, a mutex is initialized and a condition variable is set up (steps from 2 to 5). Then, on each subsequent call, the $count$ variable is incremented (step 6) and the calling thread is put on hold until it receives a signal (step 8). When all GPU threads are on hold, the $pthread\_cond\_signal$ function sends a signal to all threads waiting on a specific condition variable (step 9) and the mutex is then unlocked (step 10).

Algorithm 7 Algorithm of *__syncthreads* intrinsic function.

```
 1: procedure __SYNCTHREADS
 2:     If $count$ is equal to 0, execute steps 3, 4, and 5; otherwise go to step 6
 3:     Call $pthread\_mutex\_init$ function to initializes a mutex
 4:     Call $pthread\_cond\_init$ function to initializes a condition variable
 5:     Call $pthread\_mutex\_lock$ function to obtain a lock on the indicated mutex
 6:     Increment the $count$ variable
 7:     If $count$ is different than the number of GPU threads, go to step 8, otherwise go to step 9
 8:     Call $pthread\_cond\_wait$ function to block the indicated thread until it receives a signal
 9:     Call $pthread\_cond\_signal$ function for all GPU threads
10:     Call $pthread\_mutex\_unlock$ function and terminates
```

3.2.2. Illustrative Example The code fragment shown in Fig. 5 has 1 block and 2 threads, *i.e.*, $M = 1$ and $N = 2$, respectively. This CUDA program has a kernel (lines 4 to 6), 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 5). As shown in the main function, array positions are assigned with value 0 (line 14), and after the kernel call (line 16), 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 $a[2]$. Analyzing 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 19). 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.

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), and CIVL does not support this benchmark since it does not handle the identifier *threadIdx*.

The *cudaMalloc* function (line 12) verifies the precondition, in order to check whether *size* is greater than zero. Internally, the allocation of this memory area is converted to a *malloc* function, which is interpreted by ESBMC-GPU as an ANSI-C function. The *cudaMemcpy* function (lines 15

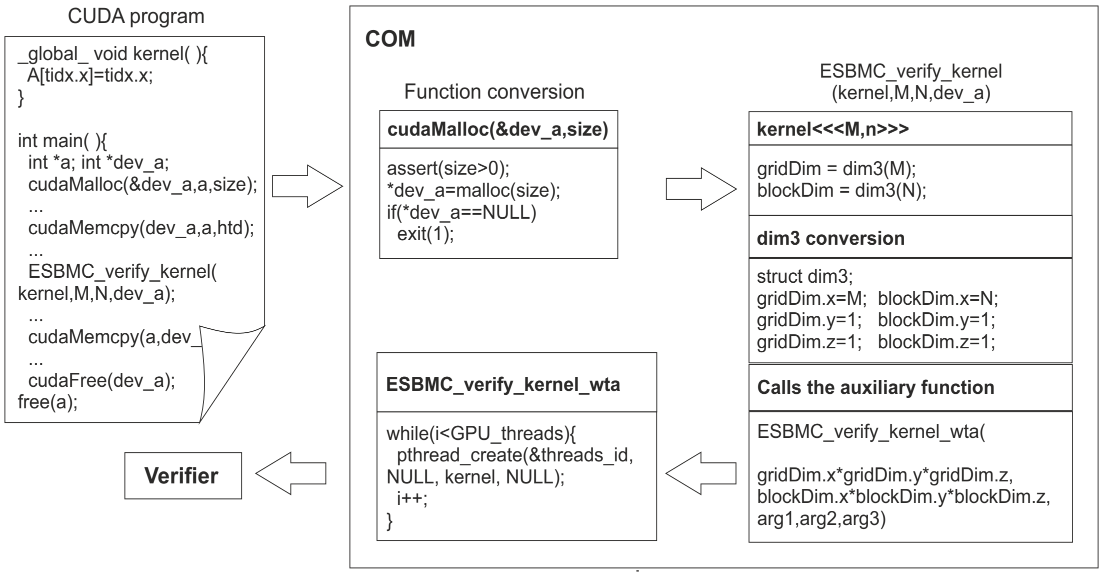
```
#define M 1
#define N 2

__global__ void kernel(int *A) {
  A[threadIdx.x + 1] = threadIdx.x;
}

int main(){
  int *a; int *dev_a;
  int size = N * sizeof(int);
  a = (int*) malloc(size);
  cudaMalloc((void**)&dev_a, size);
  for (int i = 0; i < N; i++)
     a[i] = 0;
  cudaMemcpy(dev_a,a,size, cudaMemcpyHostToDevice);
  ESBMC_verify_kernel(kernel, M, N, dev_a);
  cudaMemcpy(a,dev_a,size, cudaMemcpyDeviceToHost);
  for (int i = 0; i < N; i++)
     assert(a[i]==i);
  cudaFree(dev_a);
  free(a);
}
```

Figure 5. Code snippet to index *array*.

and 17) also checks whether *size* is greater than zero and, if so, it performs a copy using the *memcpy* function.



```
_global_ void kernel( ){
  A[tidx.x]=tidx.x;
}

int main( ){
  int *a; int *dev_a;
  cudaMalloc(&dev_a,a,size);
  ...
  cudaMemcpy(dev_a,a,htd);
  ...
  ESBMC_verify_kernel(
kernel,M,N,dev_a);
  ...
  cudaMemcpy(a,dev_
  ...
  cudaFree(dev_a);
free(a);
```

```
assert(size>0);
*dev_a=malloc(size);
if(*dev_a==NULL)
  exit(1);
```

```
gridDim = dim3(M);
blockDim = dim3(N);
```

```
struct dim3;
gridDim.x=M;  blockDim.x=N;
gridDim.y=1;  blockDim.y=1;
gridDim.z=1;  blockDim.z=1;
```

```
ESBMC_verify_kernel_wta(

gridDim.x*gridDim.y*gridDim.z,
blockDim.x*blockDim.y*blockDim.z,
arg1,arg2,arg3)
```

```
while(i<GPU_threads){
  pthread_create(&threads_id,
NULL, kernel, NULL);
  i++;
}
```

Figure 6. COM conversion steps.

The *ESBMC_verify_kernel* function receives as parameter the kernel function name (*kernel*), the number of blocks (*M*) and threads (*N*), and the *int* parameter *dev_a*. This function runs the grid configuration, creating a *dim3* representation and saving it in the *gridDim* and *blockDim* variables. In that particular point, the *assignIndexes* function is executed to create a vector with preprocessed thread IDs, avoiding state overgrowth at the verification paths. A loop triggers those threads with *pthread_create* and, after concluding its execution, the *pthread_join* thread synchronizes and finalizes the *ESBMC_verify_kernel* execution. The *assert* function checks whether there is any unexpected value in the kernel. Lastly, the *cudaFree* function releases the memory allocated by variables, verifying the occurrence of errors (*e.g.*, doublefree).

Figure 6 shows the steps executed by the ESBMC-GPU tool, in order to verify the program shown in Figure 5. Firstly, the *main* function instructions are modeled and converted to C/C++ native functions together with COM. C/C++ variables and functions are also recognized by the ESBMC-GPU without the need for converting them (*e.g.*, lines 10 and 11). The CUDA functions *cudaMalloc*, *cudaMemcpy*, *cudaFree*, and *ESBMC_verify_kernel* are part of the COM, which are thus converted to C/C++ and POSIX threads.

3.3. Monotonic Partial Order Reduction

To reduce the number of threads interleavings in CUDA programs, ESBMC-GPU implements the **M**onotonic **P**artial **O**rder **R**eduction (MPOR), which was initially proposed by Kahlon *et al.* [12]. In particular, the implementation of MPOR in ESBMC-GPU is inherited from its predecessor (ESBMC) [37]. 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. 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. As MPOR depends on the next state, ESBMC-GPU then verifies the previous state from the current one to verify whether any dependency between threads exists.

In particular, the MPOR implementation is applied to identify accesses to different positions in (shared) arrays. Typically, threads access unique positions in those global (global) arrays, which do not have dependency between them, thus allowing us to remove redundant states that are generated by the possible thread execution orders. Based on this observation, we have extended the MPOR algorithm in ESBMC to identify transitions in which threads access different array locations. This type of access results in the same state independently of the thread execution order. 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].

Next we describe the MPOR algorithm to identify transitions in which threads access different array locations and also outline its execution using a running example. Let Π be the RT, each node ν in Π be 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. Algorithm 8 shows the main steps of MPOR in ESBMC-GPU.

Algorithm 8 MPOR algorithm to identify accesses to different positions in shared arrays.

```
1: function MPOR($\nu$, $\Pi$)
2:     Check whether $s_i$ exists in $\Pi$; otherwise, go to step $4$.
3:     Check whether $A_i$ has read/write access to $s_i$; otherwise, go to step $5$
4:     Analyze whether $\gamma\,(s_{i-1}, s_i)$ is independent on $\Pi$; otherwise, go to step $6$
5:     Return "independent" on $\Pi$ and terminates.
6:     Return "dependent" on $\Pi$ and terminates.
7: end function
```

As example, Fig. 7(a) 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 (step 2) and whether transition $t_1 \rightarrow t_2$ is defined as independent (step 4). Both conditions do not hold and transition $t_1 \rightarrow t_2$ is defined as dependent (step 6). 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. Algorithm 8 then checks whether thread t_1 has reached a redundant state (step 2). 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. 7(a) by dotted lines).

Fig. 7(b) 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 RT nodes, the condition (step 2) 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 $a[2]$. Algorithm 8 then checks whether v_4 exists on Π (step 2). Since the condition (step 4) 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).

In our experimental evaluation, we observed that the application of MPOR to CUDA programs leads to a substantial performance improvement in ESBMC-GPU. The main reason for this improvement is that our MPOR implementation symbolically encodes the interleavings only if transitions are defined as dependent. In particular, when SMT formulae are built, ESBMC-GPU disregards those interleavings that lead to redundant states and only pass to the SMT solver the ones that have dependent transitions, which thus lead to new program states. With the MPOR analysis, we firstly define the first interleaving and only consider further interleavings if they are not redundant (the proof is described in Kahlon *et al.* [12]).

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

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

3.3.1. Differences from MPOR to other interleaving reduction methods There are software verifiers that apply interleaving reduction methods to CUDA program verification. As example, GKLEE implements a technique named as "canonical scheduling", which records all reads and writes that are run by each thread within barrier interval before continuing to the next thread. Records are related in pairs that can have conflicts and, by means of an SMT solver query, those conflicts are checked to detect whether data race condition exists. If conflicts are not detected, then the canonical

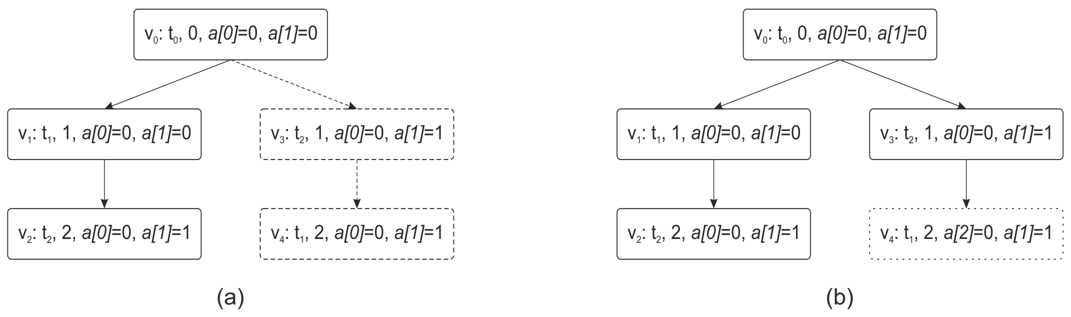

v₀: t₀, 0, a[0]=0, a[1]=0
v₁: t₁, 1, a[0]=0, a[1]=0
v₃: t₂, 1, a[0]=0, a[1]=1
v₂: t₂, 2, a[0]=0, a[1]=1
v₄: t₁, 2, a[0]=0, a[1]=1
(a)
v₀: t₀, 0, a[0]=0, a[1]=0
v₁: t₁, 1, a[0]=0, a[1]=0
v₃: t₂, 1, a[0]=0, a[1]=1
v₂: t₂, 2, a[0]=0, a[1]=1
v₄: t₁, 2, a[2]=0, a[1]=1
(b)

Figure 7. MPOR applied to a kernel with independent (Fig. 7a) and dependent (Fig. 7b) transitions.

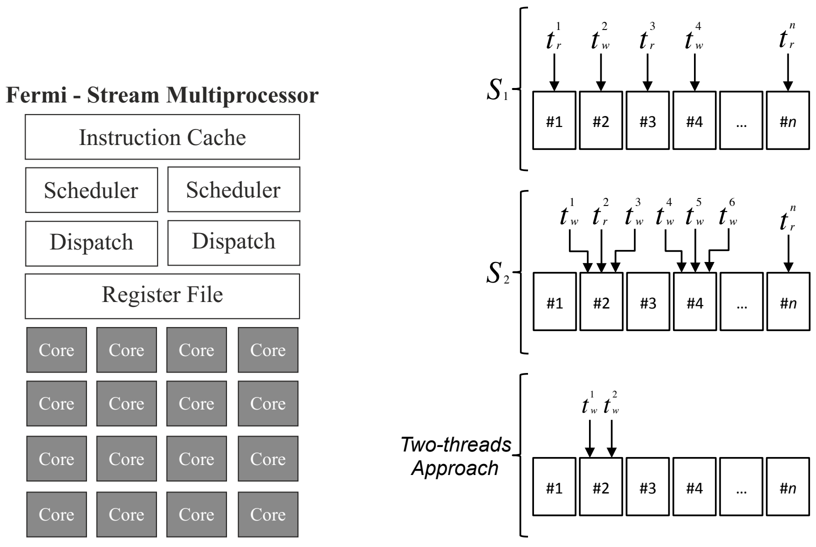
schedule is defined to represent the barrier interval for the verification of other properties. The MPOR algorithm performs its analysis and reduction on verification time and checks the properties violation on-the-fly, by encoding (small) SMT formulae. GKLEE first checks data race condition to reduce schedules and then verifies other properties. We can observe that both approaches call several times its back-end verification engine, being GKLEE with canonical scheduling and ESBMC-GPU with BMC and Pthread using MPOR.

3.4. *Two-threads Analysis*

GPU architectures are composed by multiprocessors built upon **P**rocessing **E**lements (PE) sets [1, 13]. Those PEs are typically arranged in subgroups (named as warps), which run in the same lockstep, 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.

In order to demonstrate the reasoning upon two-threads analysis, a NVIDIA Fermi GPU [13] composed by 32 processing cores is shown in Figure 8. Importantly, Figure 8a shows the architecture model of that GPU with 16 processing cores (highlighted in grey boxes), because the thread processing is performed in half-warps [39], *i.e.*, 16 threads of the same warp are scheduled, ensuring synchronization when a barrier occurs. Figure 8b demonstrates the thread representation, which accesses the GPU shared memory. 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; it is worth noticed that the threads are executed on a streaming multiprocessor. In the first case of Figure 8b (*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 Figure 8b (*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 Figure 8b is quite similar to the situation occurred in S_2, however, instead of using multiple threads to analyse data-race conditions, ESBMC-GPU ensures that such errors can



(a) NVIDIA Fermi GPU (b) Two-threads approach

Figure 8. Comparison between the NVIDIA Fermi GPU architecture and the two-threads approach used by used to handle data.

be detected through the analysis of the behaviour of two threads that are operating over the same memory location.

Note that 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, we observed a substantial performance improvement, considering only two threads, while keeping true incorrect results at low rates.

4. EXPERIMENTAL EVALUATION

4.1. Experimental Setup

This section describes the experiments to investigate ESBMC-GPU performance for verifying CUDA programs. We also compare ESBMC-GPU to GKLEE [8], GPUVerify [3], PUG [7], and CIVL [9]. In particular, we evaluate ESBMC-GPU's ability to verify 154 benchmarks[†], which are extracted from [1, 3, 13]; 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 ANSI-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*).

Note that we used benchmarks from a variety of sources to evaluate ESBMC-GPU's precision and performance, which include CUDA benchmarks suite to check basic functions that are commonly

[†]ESBMC-GPU and benchmarks are available at http://esbmc-gpu.org

used by real CUDA applications. Our benchmarks suite comprises of 20 CUDA kernels from NVIDIA GPU Computing SDK v2.0 [44], 20 CUDA kernels from Microsoft C++ AMP Sample Projects [45], and 104 CUDA programs that explore a wide range of CUDA functionalities [1, 13]. Although there are some CUDA benchmarks that do not implement real CUDA applications, they are still valuable to analyze to which extent ESBMC-GPU is able to handle and detect (known) errors; in particular, those benchmarks have been previously used for comparing the performance and precision of different GPU verifiers [3]. It is worth notice that the benchmarks related to OpenCL [46] were not applied to our experimental evaluation, once ESBMC-GPU does not support it yet.

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, PUG, and CIVL?

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`[‡], which 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, PUG, and CIVL 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 -libc=uclibc` or `gklee-nvcc file.cu -libc=klee; 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`.

Additional options for ESBMC-GPU and GPUVerify are necessary to check for data races and array out-of-bounds, respectively.

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`.

[‡]The value of C ranges from 2 to 4 context switches.

To verify CUDA programs with CIVL, we use the command `civl verify file.cu`. If the benchmark contains the *malloc* and *free* functions, we must manually add the standard ANSI-C library *stdlib.h* to the CUDA program, although it is not mandatory to declare it according to the CUDA documentation [1].

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. *Evaluating SMT Solvers for Verifying CUDA Programs*

ESBMC-GPU now supports five different SMT solvers: Z3 v4.0 [17], Boolector v2.0.1 [40], Yices v2.3.1 [41], MathSAT v5 [42], and CVC4 [43]. The flag −−solver (*e.g.*,−−boolector) is used to define an SMT solver, which is different from the standard ESBMC-GPU configuration.

The solvers' performance is shown in Table I, where the "tool" column describes the tool name, "true correct" shows the result in which no error has been found for bug-free benchmarks, "false correct" shows the results in which the tool correctly detected a bug, "true incorrect" shows the results in which the tool did not detect a bug, "false incorrect" shows the results in which the tool incorrectly detected a bug, "not supported" shows the results which are not supported by the respective tool, and "time" shows the total verification time.

Table I shows that Z3, Boolector, Yices, and MathSAT produce exactly the same number of true correct results (56). Additionally, Z3, Boolector, and Yices present 63 false correct results, while MathSAT produced only 61. These results ensure that Z3, Boolector, and Yices are the most effective SMT solvers for verifying CUDA programs in ESBMC-GPU, where correctness is an important parameter for evaluating the solvers performance. Additionally, if we consider verification time, then Z3 has an improved performance of $3,6\%$ over Yices. Thus, we defined Z3 as the (default) SMT solver to verify CUDA programs in ESBMC-GPU.

4.3. *Experimental Results*

Table II shows the results of ESBMC-GPU v1.25, GKLEE (LLVM v3.2), GPUVerify 2, PUG (ROSE v0.9) and CIVL v1.6; 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 82.5% of the benchmarks, GKLEE 71.4%, GPUVerify 57.1%, PUG 35.1% and CIVL 30.5%. Note that ESBMC-GPU produces 1 true incorrect results, while GKLEE produces 14, GPUVerify produces 9, PUG produces 7 and CIVL does not produce any true incorrect result. With ESBMC-GPU, this result is due to a null pointer access. 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 (2). GPUVerify does not detect data race (7), does not detect an array bounds violation (1), and does not detect an assertion violation (1).

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 7 false incorrect results, which are caused by incorrectly detected assertions (4), data-races (1), array out-of-bounds (1) and solver call failure (1). GPUVerify generated 8 false incorrect results, due to incorrectly detected assertion (2) and data-races (6). PUG produces 11 false incorrect results due to data races incorrectly detected. CIVL produces 3 false incorrect results due to detection of memory allocation error (2) and assertions (1).

ESBMC-GPU had 23 benchmarks that were not supported. These are related to constant memory access (3), the use of CUDA's specific libraries (*e.g., curand.h*) (7), and the use of pointers to functions, structures, and char type variables used as kernel call arguments (13). GKLEE has 22 benchmarks that were not supported, which are due to incorrectly detected assertions (2); function pointers (11), either used as kernel arguments or as in any other parts of the CUDA program; specific CUDA libraries *curand.h* (7), and *switch-case* functions (2).

GPUVerify did not support 49 benchmarks. Since it does not support main functions, this explains most cases (39); it also does not support the use of *memset* function (2), the array bounds detection (2), the use of *math_functions* (1), 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 (5).

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).

CILV does not support 104 benchmarks, this is caused by the use of *atomic functions* (18), *cudaThreadSynchronize* (10), *threadIdx* (10), function *typedef* (8), *curand functions* (7), *dim3* (6), *math_functions* (5), *uint4* (5), *__constant__ variables* (5), *__attribute__* (5), *__restrict* (2), structs (2), scanf (2), boolean functions (e.g AND and OR) (2), *uint* (2), *extern* C (1), *__threadfence* (1), typecast (1). Other benchmarks could not be verified due to the existence of data-race in benchmarks (11) and undefined cases (1).

4.3.1. ESBMC-GPU effectiveness over existing GPU verifiers MPOR produced an improved performance of approximately 80% in our benchmarks; in particular, it reduced the verification time from 16 to 3 hours. With two-threads analysis, we further reduced the verification time from 3 hours to 811 seconds. Although the application of these techniques has considerably improved the ESBMC-GPU performance, it still takes more (verification) time than all other existing 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.

Additionally, in order to improve the verification performance, ESBMC-GPU uses an abstract representation of the CUDA libraries that properly reflects their semantics. This abstract representation includes pre/post-conditions and simulation features that consider only relevant behavior to be explored from the verification perspective, *i.e.*, irrelevant code fragments that unnecessarily complicate the generated VCs are actually abstracted. As a result, during the ESBMC-GPU verification process, the COM replaces the real CUDA libraries, which thus improves its effectiveness to detect more properties violations than other existing GPU verifiers, as shown in our experimental evaluation.

5. CONCLUSIONS

ESBMC-GPU is able to verify CUDA programs using SMT-based context-bounded model 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, PUG, and CIVL. Additionally, ESBMC-GPU provides fewer incorrect results than all other existing GPU verifiers. Experimental results also show that ESBMC-GPU presents a successful verification rate of 82.5%, compared to 71.4% of GKLEE, 57.1% of GPUVerify, 35.1% of PUG, and 30.5% of CIVL. For future work, we will improve our support to argument types of kernel functions, support stream interleaving, and apply techniques to reduce the number of threads interleavings. Besides, conformance testing procedures will be developed for further validating COM, which could also be applied to CUDA modules.

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

REFERENCES

1. Cheng J, Grossman M, McKercher T. *Professional CUDA C Programming*. John Wiley and Sons, Inc. 2014.
2. Kirk D, Hwu W. *Programming Massively Parallel Processors*. Elsevier Inc. 2010.
3. Betts A, Chong N, Donaldson AF, Qadeer S, Thomson P. *GPUVerify: A Verifier for GPU Kernels*. OOPSLA 2012; 113–132.
4. Cordeiro L, Fischer B. *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking*. ICSE 2011; 331–340.
5. Cordeiro L, Fischer B, Marques-Silva J. *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. IEEE Trans. Software Eng. 2012; 38(4):957–974.
6. Ramalho M, Freitas M, Sousa F, Marques H, Cordeiro L, Fischer B. *SMT-Based Bounded Model Checking of C++ Programs*. ECBS 2013; 147–156.
7. Li G, 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, Rajan SP. *GKLEE: Concolic Verification and Test Generation for GPUs*. PPoPP 2012; 215–224.
9. Zheng M, Rogers M, Luo Z, Dwyer M, Siegel S. *CIVL: Formal Verification of Parallel Programs*. ASE 2015.
10. A. Biere. *Bounded Model Checking. Handbook of Satisfiability*. IOS Press 2009; 457–481.

11. Barrett C, Sebastiani R, Seshia S, Tinelli C. *Satisfiability Modulo Theories. Handbook of Satisfiability*. IOS Press 2009; 825–885.
12. Kahlon V, Wang C, Gupta A. *Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique*. CAV 2009; 398–413.
13. NVIDIA. *CUDA C Programming Guide*. NVIDIA Corporation 2015.
14. Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verificação de Kernels em Programas CUDA usando Bounded Model Checking.* WSCAD-SSC 2015; 24–35.
15. Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking.* SAC SVT track 2016; 1648–1653.
16. Le Goues C, Leino KRM, Moskal M. *The Boogie Verification Debugger.* SEFM 2011; 407–414.
17. Moura L, Bjorner N. *Z3: An Efficient SMT Solver*. TACAS 2008; 337–340.
18. Barrett C, Conway C, Deters M, Hadarean L, Jovanović D, King T, Reynolds A, Tinelli C. *CVC4.* CAV 2011; 171–177.
19. Li P, Li G, Gopalakrishnan G. *Practical Symbolic Race Checking of GPU Programs*. SC 2014; 179–190.
20. Morse J, Cordeiro L, Nicole D, Fischer B. *Context-Bounded Model Checking of LTL Properties for ANSI-C Software.* SEFM 2011; 302–317.
21. Morse J, Cordeiro L, Nicole D, Fischer B. *Model checking LTL properties over ANSI-C programs with bounded traces.* Software and System Modeling 2015; 14(1): 65–81.
22. Beyer, D. *Status report on software verification.* TACAS 2014; 373–388.
23. Beyer, D. *Software verification and verifiable witnesses.* TACAS 2015; 401–416.
24. Clarke E, Kroening D, Lerda F. *A Tool for Checking ANSI-C Programs*. TACAS 2004; 168–176.
25. Levine J, John L. *Flex & Bison*. 1st Edition, O'Reilly Media, Inc., 2009.
26. Lopes B, Auler R *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.
27. M. Cade. *Why Apples swift language will instantly remake computer programming*. http://www.wired.com/2014/07/apple-swift/, 2016.
28. Cytron R, Ferrante J, Rosen B, Wegman M, Zadeck F. *An Efficient Method of Computing Static Single Assignment Form* POPL 1989; 25–35.
29. Monteiro F, Cordeiro L, de Lima Filho E. *Bounded Model Checking of C++ Programs Based on the Qt Framework*. GCCE 2015; 179–447.
30. 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.
31. Heila van der Merwe, Brink van der Merwe, and Willem Visser. *Verifying android applications using Java PathFinder* SIGSOFT Softw. Eng. Notes 37, 6 2012; 1-5.
32. Heila van der Merwe, Oksana Tkachuk, Brink van der Merwe, and Willem Visser. *Generation of Library Models for Verification of Android Applications* SIGSOFT Softw. Eng. Notes 40, 1 2015; 1-5.
33. Morse J, Ramalho M, Cordeiro L, Nicole D, Fischer B. ESBMC 1.22 - (Competition Contribution). TACAS 2014; 405–407.
34. Mehtaa F, Nipkowb T. *Proving pointer programs in higher-order logic*. Information and Computation 2005; 199(1), 200–227.
35. Cámara P, Castro J, Gallardo M, Merino P. *Verification support for ARINC-653-based avionics software*. JSTVR 2011; 21(4): 267–298.
36. Cámara P, Gallardo M, Merino P, Sanán D. *Checking the reliability of socket based communication software*. STTT 2009; 11(5): 359-374.
37. Morse J. *Expressive and Efficient Bounded Model Checking of Concurrent Software*. University of Southampton, PhD Thesis 2015.
38. 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.
39. NVIDIA. *CUDA Compute Architecture: Fermi*. NVIDIA Corporation 2009.
40. Brummayer R, Biere A. *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*. TACAS 2009; 174–177.
41. B. Dutertre. *Yices 2.2*. CAV 2014; 737–744.
42. Cimatti A, Griggio A, Schaafsma B, Sebastiani R. *The MathSAT5 SMT Solver*. TACAS 2013; 93–107.
43. Deters M, Reynolds A, King T, Barrett C, Tinelli C. *A tour of CVC4: How it works, and how to use it*. FMCAD 2014; 7.
44. NVIDIA. *CUDA Toolkit Release Archive*. urlhttps://developer.nvidia.com/cuda-toolkit-archive, 2015.
45. Microsoft Corporation. *C++ AMP sample projects for download (MSDN blog)*. urlblogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx, 2012.

46. Khronos OpenCL Working Group. *The OpenCL Specification - version 1.1*. Document Revision: 44, 2011.

Table I. Results of the SMT Solvers Performance. **True Correct** represents the number of flawless benchmarks correctly verified, while **True Incorrect** represents the ones verified incorrectly. Similarly, **False Correct** represents the number of faulty benchmark verified correctly, while **False Incorrect** represents the ones verified incorrectly. Not Supported represents the number of benchmarks unsuccessfully verified, **Time** represents the total verification time for each solver, and bold numbers represent the best results in each category.

Result/Tool	**Z3 4.0**	**Boolector 2.0.1**	**Yices 2.3.1**	**CVC 4**	**MathSAT 5**
True Correct	**56**	**56**	**56**	35	**56**
False Correct	**63**	**63**	**63**	60	61
True Incorrect	**2**	**2**	**2**	4	**2**
False Incorrect	5	5	4	28	**2**
Not supported	28	28	29	**27**	33
Time (s)	**1828**	4614	1894	3326	8398

Table II. Results of ESBMC-GPU, GKLEE, GPUVerify, PUG, and CIVL. **True Correct** represents the number of flawless benchmarks correctly verified, while **True Incorrect** represents the ones verified incorrectly. Similarly, **False Correct** represents the number of faulty benchmarks verified correctly, while **False Incorrect** represents the ones verified incorrectly. **Not Supported** represents the number of benchmarks unsuccessfully verified, **Time** represents the total verification time for each verifier, and bold numbers represent the best results in each category.

Result/Tool	**ESBMC-GPU**	**GKLEE**	**GPUVerify**	**PUG**	**CIVL**
True Correct	**60**	53	58	39	23
False Correct	**67**	57	30	15	24
True Incorrect	1	14	9	7	**0**
False Incorrect	**3**	8	8	11	**3**
Not Supported	23	**22**	49	82	104
Time (s)	811	128	147	**12**	158