SMT-Based Context-Bounded Model Checking for CUDA Programs

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	CPE-16-0056
Editor Selection:	Special Issue Submission
Wiley - Manuscript type:	Special Issue Paper
Date Submitted by the Author:	29-Jan-2016
Complete List of Authors:	Pereira, Phillipe; Universidade Federal do Amazonas Albuquerque, Higo; Universidade Federal do Amazonas da Silva, Isabela; Universidade Federal do Amazonas Marques, Hendrio; Universidade Federal do Amazonas Ferreira, Ricardo; Universidade Federal de Vicosa Cordeiro, Lucas; Universidade Federal do Amazonas
Keywords:	Symbolic and Explicit Model Checking, GPU kernels, CUDA Programs, MPOR

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–19
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], R. dos S. Ferreira[2], 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, 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 than other existing GPU verifiers, while keeping lower rates of false results. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: 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, in order 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 a CUDA operational model (COM), which is an abstract representation of the standard CUDA libraries (*i.e.*, the native application programming interface) that conservatively approximates their

*Correspondence to: Av. Rodrigo Otavio, 3000 - CETELI, Universidade Federal do Amazonas, Amazonas, Brasil. Email: apphillipe@gmail.com semantics. We describe the implementation of COM, 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, 11], 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 redundant interleavings without missing any behavior that can be exhibited by the program. 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 [8, 3, 7, 11]. Additionally, the proposed approach considers low-level aspects related to dynamic memory allocation, data transfer, memory deallocation, 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 significant 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], PUG [7], and CIVL [11] 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.

This paper extends our previous work [12, 13]. 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. 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. The remainder of the paper is organized as follows: We first give a brief introduction to ESBMC, CUDA, and existing verifiers for GPU. 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. 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 is one of the most prominent BMC tools for verifying C programs according to the last editions of the International Competition on Software Verification (SV-COMP) [14, 15]. 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 (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 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 [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. *__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; 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 [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.

SESA (Symbolic Executor with Static Analysis) [19] 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.

CIVL [11] is a framework for static analysis and concurrent program verification, which supports MPI, POSIX, OpenMP, and CUDA. CIVL recognizes programs written in C11, CUDA-C, and CIVL-C, which is an intermediate language that represents 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.

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 specific properties (*e.g.*, array out-of-bounds, arithmetic overflow, pointer safety, and data races).

In our CUDA operational model (COM), methods, data types, qualifiers, and CUDA directives were implemented in C++. As an example, Figure 1 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 modeled 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 C/C++, COM supports specific data types of CUDA (*e.g., char1, short2, float2*).

COM also models methods, which are typically used to develop CUDA programs. As example, we show three essential functions in CUDA programs: *cudaMalloc*, *cudaMemcpy*, and *cudaFree*. Figure 2 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 (from line 6 to 11). 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

```
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 1. *dim3* implementation.

positive memory size allocation; line 4 of Fig. 2 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 together with the counterexample.

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

Figure 2. *cudaMalloc* implementation.

Figure 3 presents the $cudaMemcpy()$ function operational model. It checks, as precondition, the memory size to be copied (line 3). Two local variables *cdst* and *csrc* (lines 4 and 5) 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 6); the data copy is actually performed (in lines 7 and 8) between *device* and *host*. Finally, *cudaMemcpy* function returns value $CUDA\_SUCCESS$.

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

Figure 3. *cudaMemcpy* implementation.

Similarly to *cudaMalloc* and *cudaMemcpy*, we also provide the implementation of the *cudaFree* function. Figure 4 shows the implemented model, where a pointer to a variable is passed as an input argument so that the allocated memory is released. As we can see (in line 2), the *cudaFree* function is referenced to the *free* function of the C programming language, with the only difference that the memory is freed by the GPU. 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 employed hardware [20]. Variables *lastError* and *CUDA_SUCCESS* have the same functions as in *cudaMalloc*.

```
cudaError_t cudaFree(void *devPtr) {
  free(devPtr);
  lastError = CUDA_SUCCESS;
  return CUDA_SUCCESS;
}
```

Figure 4. *cudaFree* implementation.

Note that the operational models are implemented according to the NVIDIA Programming Guide [10]. The behavior of those functions can be represented in C/C++ programming languages, using native functions that are already supported by ESBMC (*e.g.*, *malloc*, *free*, *assert*). 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 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. 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 [22]. 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.

Figure 5 shows the implementation of *ESBMC_verify_kernel* function with support for six parameters. The parameter *kernel* (line 2) is a pointer to the original program kernel function. The parameters *blocks* and *threads* (line 2) receive the blocks and threads configuration, which can be of type *int* or *dim3*. The parameters *arg1*, *arg2*, and *arg3* (line 2) correspond to values sent to the (respective) function. Internally, there are two variables, *gridDim* (line 4) and *blockDim* (line 5), which receive the output of the *dim3* constructor used to configure the blocks and threads dimension. The function *ESBMC_verify_kernel_wta* (line 6) particularizes the argument type to *int*

```
 1 template<class RET, class BLOCK, class THREAD, class T1, class T2, class T3>
 2   void ESBMC_verify_kernel(RET *kernel,BLOCK blocks,THREAD threads,
 3     T1 arg1,T2 arg2,T3 arg3){
 4       gridDim = dim3(blocks);
 5       blockDim = dim3(threads);
 6       ESBMC_verify_kernel_wta((voidFunction_t)kernel,
 7   gridDim.x * gridDim.y * gridDim.z,
 8   blockDim.x * blockDim.y * blockDim.z,
 9   (void*)arg1,(void*)arg2,(void*)arg3);
10       for (int i=0; i<GPU_threads; i++)
11   pthread_join(threads_id[i], NULL);
12 }
```

Figure 5. *ESBMC_verify_kernel* template implementation.

and determines the type of the kernel function pointer. Lastly, there is a loop *for* (line 10), which synchronizes by means of the *pthread_join* function (line 11), 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 3.3), the loop *for* is bounded to two unwindings.

Figure 6 shows the implementation of the *ESBMC_verify_kernel_wta* function. This function is responsible for translating GPU threads into POSIX threads. The *threads_id* variable (line 3) stores the thread IDs of *pthread_t* type. *The dev_three* variable (from line 4 to 10) is a *struct* and stores a pointer to the kernel function. The *assignIndexes* function (line 12) 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. These values are stored in a *dim3* vector and are statically calculated to reduce paths generated during the verification. Lastly, in the loop *while* (line 13), 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*.

```
 1 void ESBMC_verify_kernel_wta(void *(*kernel)(int*,int*,int*),
 2     int blocks,int threads,void* arg1,void* arg2,void* arg3) {
 3   threads_id = (pthread_t *) malloc(GPU_threads * sizeof(pthread_t));
 4   dev_three.a = (int*) malloc(GPU_threads * sizeof(int));
 5   dev_three.b = (int*) malloc(GPU_threads * sizeof(int));
 6   dev_three.c = (int*) malloc(GPU_threads * sizeof(int));
 7   dev_three.a = (int*) arg1;
 8   dev_three.b = (int*) arg2;
 9   dev_three.c = (int*) arg3;
10   dev_three.func = kernel;
11   int i = 0, tmp;
12   assignIndexes();
13   while (i < GPU_threads) {
14     pthread_create(&threads_id[i],NULL,ESBMC_execute_kernel_t,NULL);
15     i++; }
16 }
```

Figure 6. *ESBMC_verify_kernel_wta* function implementation.

Figure 7 shows the implementation of *ESBMC_execute_kernel_t* function, where it checks the original kernel function using the parameters, which are passed as argument by the *struct dev_three*. ESBMC-GPU then verifies this function and interleaves its possible execution paths.

```
void *ESBMC_execute_kernel_t(void *args) {
  dev_three.func(dev_three.a, dev_three.b, dev_three.c);
  return NULL;
}
```

Figure 7. *ESBMC_execute_kernel_t* function implementation.

3.1.2. Illustrative Example The code fragment shown in Fig. 8 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 13), 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 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, cudaMemcpyHostToDevice);
13     ESBMC_verify_kernel(kernel, M, N, dev_a);
14     cudaMemcpy(a,dev_a,size, cudaMemcpyDeviceToHost);
15     for (int i = 0; i < N; i++)
16       assert(a[i]==i);
17     cudaFree(dev_a);
18     free(a);
19  }
```

Figure 8. 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), and CIVL does not support this benchmark since it does not handle the identifier *threadIdx*.

The *cudaMalloc* function (line 10) 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 a C function. The *cudaMemcpy* function (lines 12 and 14) also checks whether *size* is greater than zero and, if so, it performs a copy using the *memcpy* function.

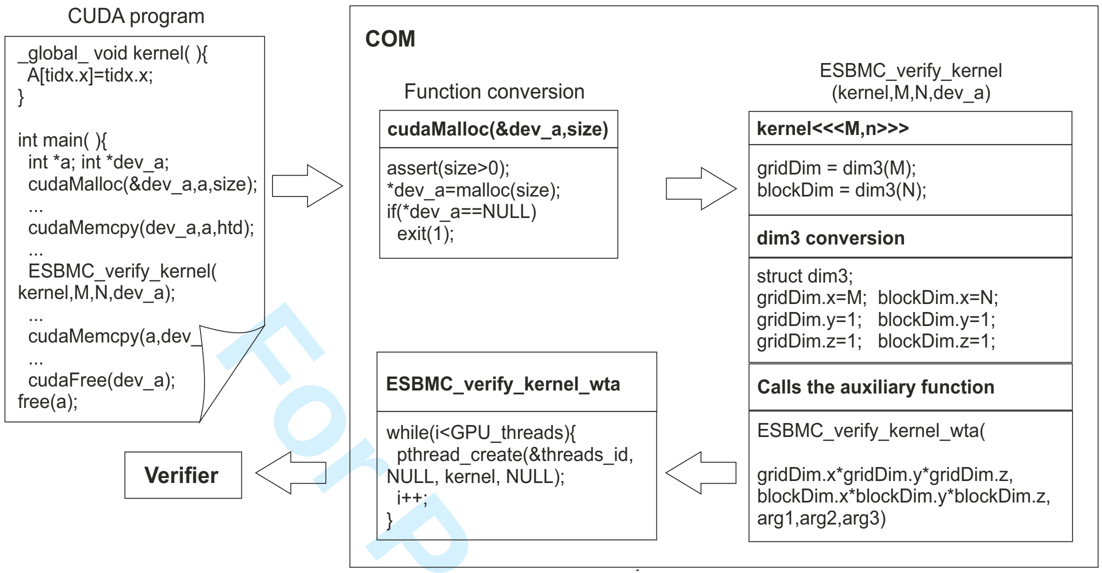


Figure 9. COM conversion steps.

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 9 shows the steps executed by the ESBMC-GPU tool, in order to verify the program shown in Figure 8. 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 7 and 8). 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.

3.2. 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]. In particular, the implementation of MPOR in ESBMC-GPU is inherited from its predecessor (ESBMC) [21]. 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 [21]. 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. 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]. 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 1 shows the main steps of MPOR in ESBMC-GPU.

Algorithm 1 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. 10(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. 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 10(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. 10(b), however, array a is of length 2 and two threads are executed; those two thread execution orders result in different program states. One interleaving shows an access out-of-bounds violation in array a (*i.e.*, v_4 of Fig 10(b)). The algorithm identifies execution order dependency and two interleavings must be considered, leading to a violation.

GPU architectures are composed by multiprocessors built upon processing elements (PE) sets [1, 10]. Those PEs are typically arranged in subgroups (named 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.

In our experimental evaluation, we observed that the application of MPOR to CUDA programs ensures a dramatic performance improvement.

3.3. Two-threads Analysis

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.

```
__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; }
}
```

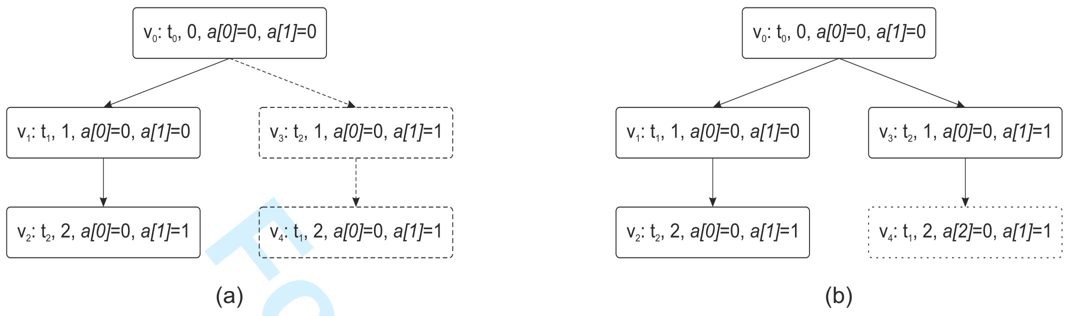

v₀: t₀, 0, a[0]=0, a[1]=0
v₁: t₁, 1, a[0]=0, a[1]=0
v₃: t₂, 1, a[0]=0, a[1]=1
v₂: t₂, 2, a[0]=0, a[1]=1
v₄: t₁, 2, a[0]=0, a[1]=1
(a)
v₀: t₀, 0, a[0]=0, a[1]=0
v₁: t₁, 1, a[0]=0, a[1]=0
v₃: t₂, 1, a[0]=0, a[1]=1
v₂: t₂, 2, a[0]=0, a[1]=1
v₄: t₁, 2, a[2]=0, a[1]=1
(b)

Figure 10. MPOR applied to a kernel with independent (Fig. 10a) and dependent (Fig. 10b) transitions.

Figure 11 shows the reasoning upon two-threads analysis. As example, a NVIDIA Fermi GPU is composed by 32 processing cores. Figure 11(a) shows the architecture model of that GPU with 16 processing cores, because the thread processing is performed in half-warps [23], *i.e.*, 16 threads of the same warp are scheduled, ensuring synchronization when a barrier occurs. These processing cores are highlighted in grey. Figure 11(b) shows the thread representation, accessing the GPU shared memory. The arrows represent threads being executed on Streaming Multiprocessor. The gray arrows represent threads whose kernel is correctly implemented and there is no data race condition, because they access different memory positions. Arrows in white represent threads of a kernel resulting in a race condition, where several threads read or write in the same memory position, at the same lock-step. Arrows in black present the same situation of the white ones, except that they represent the way in which ESBMC-GPU detects the race condition error, assuring that the error can be detected either with the two-threads analysis or the real number of threads.

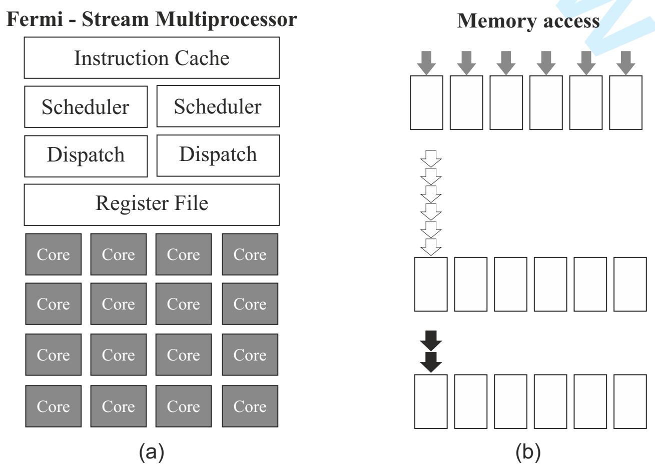


Figure 11. Fermi Architecture Model (Fig. 11a) and Memory Access (Fig. 11b).

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 experiments to investigate ESBMC-GPU performance for verifying CUDA programs. We also compare ESBMC-GPU to GKLEE [8], GPUVerify [3], PUG [7], and CIVL [11]. In particular, we evaluate ESBMC-GPU's ability to verify 154 benchmarks[†], which are extracted from [1, 3, 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, 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; 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

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

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`.

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 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, Boolector v2.0.1, Yices v2.3.1, MathSAT v5, and CVC4. 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 70.8%, 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 (1). GPUVerify does not detect data race (7), does not detect an array bounds violation (1), and does not detect an assertion (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-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 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 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).

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 811 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 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 70.8% of GKLEE, 57.1% of GPUVerify, 35.1% of PUG, and 30.5% of CIVL. For future work, we will detect barrier divergence, improve support to argument types of kernel functions, support stream interleaving, and apply techniques to reduce the number of threads interleavings.

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):957–974z.
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. Kahlon V, Wang C, Gupta A. *Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique*. CAV 2009; 398–413.

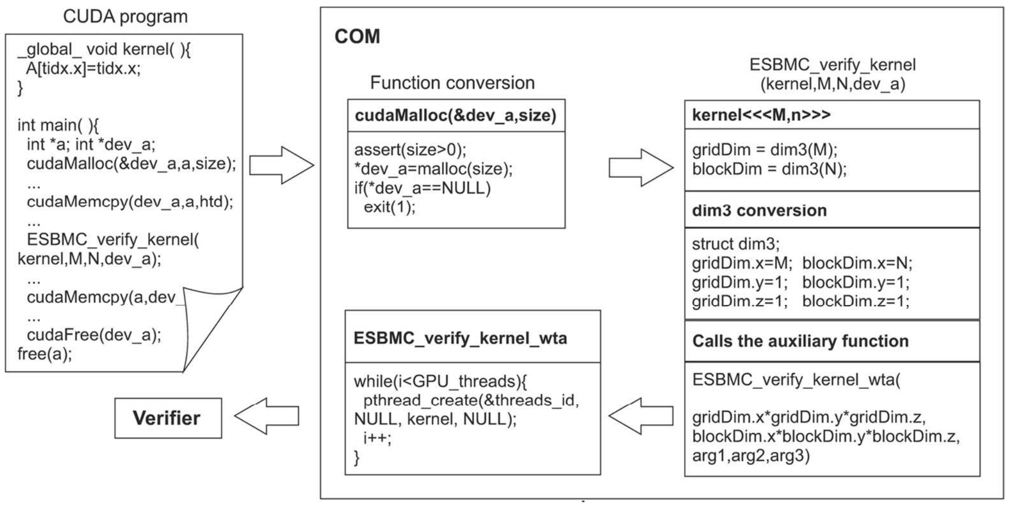
10. NVIDIA. *CUDA C Programming Guide*. NVIDIA Corporation 2015.
11. Zheng M, Rogers M, Luo Z, Dwyer M, Siegel S. *CIVL: Formal Verification of Parallel Programs*. ASE 2015.
12. Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verificação de Kernels em Programas CUDA usando Bounded Model Checking.* WSCAD-SSC 2015; 24-35.
13. 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 (to appear).
14. Beyer, D. *Status report on software verification.* Tools and Algorithms for the Construction and Analysis of Systems 2014; 373–388.
15. Beyer, D. *Software verification and verifiable witnesses.* Tools and Algorithms for the Construction and Analysis of Systems 2015; 401–416.
16. Le Goues C, Leino KRM, Moskal M. *The Boogie Verification Debugger.* SEFM 2011; 407–414.
17. De Moura L, Bjørner 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, Ramalho M, Cordeiro L, Nicole D, Fischer B. ESBMC 1.22 - (Competition Contribution). TACAS 2014; 405–407.
21. Morse J. *Expressive and Efficient Bounded Model Checking of Concurrent Software*. University of Southampton, PhD Thesis 2015.
22. Institute of Eletrical and Eletronics 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.
23. NVIDIA. *CUDA Compute Architecture: Fermi*. NVIDIA Corporation 2009.

Table I. Results of the SMT Solvers Performance.

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

Result/Tool	**ESBMC-GPU**	**GKLEE**	**GPUVerify**	**PUG**	**CIVL**
True Correct	**60**	53	58	39	23
False Correct	**67**	56	30	15	24
True Incorrect	1	14	9	7	**0**
False Incorrect	**3**	7	8	11	**3**
Not Supported	**23**	24	49	82	104
Time (s)	811	128	147	**12**	158



```
_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);
```

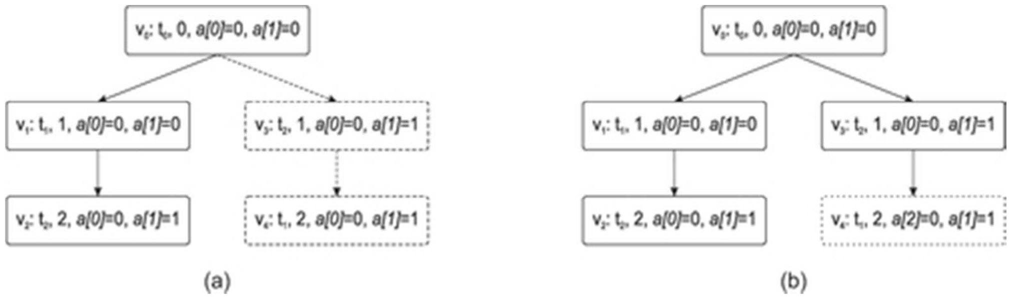
```
gridDim = dim3(M);
blockDim = dim3(N);
```

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

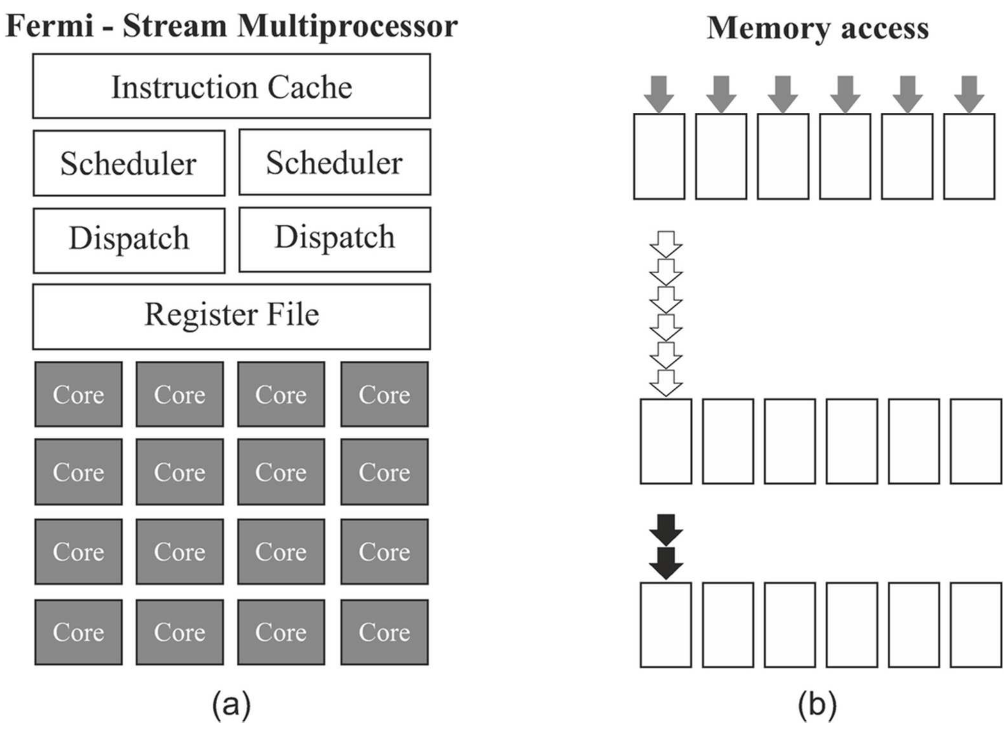
```
ESBMC_verify_kernel_wta(

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++;
}
```

COM conversion steps.
123x61mm (300 x 300 DPI)


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)

MPOR applied to a kernel with independent (Fig. 10a) and dependent (Fig. 10b) transitions.
42x12mm (300 x 300 DPI)



Fermi Architecture Model (Fig. 11a) and Memory Access (Fig. 11b).
91x64mm (300 x 300 DPI)