11-2014

GPU Accelerated counterexample generation in LTL model checking

Zhimin WU

Yang LIU

Yun LIANG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Citation

WU, Zhimin; LIU, Yang; LIANG, Yun; and SUN, Jun. GPU Accelerated counterexample generation in LTL model checking. (2014). *Proceedings of the 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, November 3–5*. 413-429. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4988

GPU Accelerated Counterexample Generation in LTL Model Checking

Zhimin Wu[1], Yang Liu[1], Yun Liang[2], and Jun Sun[3]

[1] Nanyang Technological University, Singapore
[2] Peking University, China
[3] Singapore University of Technology and Design, Singapore

Abstract. Strongly Connected Component (SCC) based searching is one of the most popular LTL model checking algorithms. When the SCCs are huge, the counterexample generation process can be time-consuming, especially when dealing with fairness assumptions. In this work, we propose a GPU accelerated counterexample generation algorithm, which improves the performance by parallelizing the Breadth First Search (BFS) used in the counterexample generation. BFS work is irregular, which means it is hard to allocate resources and may suffer from imbalanced load. We make use of the features of latest CUDA Compute Architecture-*NVIDIA Kepler GK110* to achieve the dynamic parallelism and memory hierarchy so as to handle the irregular searching pattern in BFS. We build dynamic queue management, task scheduler and path recording such that the counterexample generation process can be completely finished by GPU without involving CPU. We have implemented the proposed approach in PAT model checker. Our experiments show that our approach is effective and scalable.

1 Introduction

The LTL model checking problem is known as the emptiness checking of the product between $\mathcal{M}$ and $\mathcal{A}_{\neg\varphi}$, where $\mathcal{M}$ represents the model and $\mathcal{A}_{\neg\varphi}$ represents the Büchi automaton that expresses the negation of an LTL property φ. The emptiness checking is to detect if there exists an execution path in the product that can be accepted by the Büchi automaton. There are two main streams of LTL model checking approaches: nested Depth First Search (NDFS) and Strongly Connected Component (SCC) search, where the latter one is more suitable to handle fairness assumptions.

SCC based verification algorithms aim to find an SCC with at least one accepting state. If such SCC exists, it means that there is a run that can be accepted by the $\mathcal{A}_{\neg\varphi}$, i.e., the violation of the LTL property φ. To generate a counterexample in such case is to produce an infinite path $\pi = \rho_1\rho_2\rho_3$, which consists of three parts: a path ρ_1 from the initial state to a state s in the SCC, a path ρ_2 from the s to an accepting state a in the SCC and a loop ρ_3 that starts and ends at a. To generate such a path, currently, some algorithms [12,6] work on DFS-related solution with high complexity. Some work on BFS-related solution, such as in [5], which focus on building the minimal size counterexample to deal with the memory constraint.

In this paper, we propose an approach that has the potential to accelerate the counterexample generation process using GPU. The problem here is equivalent to building a solution to improve the performance of BFS with path recording. Compared with multi-core CPU architecture, GPU typically has a lot more cores and high memory bandwidth, which potentially provides high parallelism. Because the number of nodes in each layer of BFS is changing, it makes the resource allocation and the load balancing a challenging task in in GPU-based BFS searching. In the CUDA programming model, CPU will launch the kernel in GPU with static grid and block structures, which result in the lack or waste of compute resources. In previous research such as CUDA IIIT-BFS [7], it is necessary to launch the kernel each time when the BFS starts a new layer. It is costly and even slower than CPU-BFS in some cases. To deal with this problem, CUDA UIUC-BFS [9] has been proposed based on a hierarchical memory management solution. It builds a three-level queue for BFS to avoid consequent kernel launching, which offers certain speedup. But it is still a static method that cannot adjust according to the task size. Furthermore, there is no load balance approach in it.

In this work, we propose an almost CPU-free BFS based path generation process by leveraging on the new dynamic parallelism feature of CUDA. The key problem addressed is the number of tasks during BFS based path generation is dynamically changing. In this paper we propose four contributions. (1) Compared to related works of parallelizing BFS for model checking problems, our approach is totally CPU-Free. Existing related works allocate GPU resources in a static way. The resources can be re-allocated only by CPU when the execution of a kernel ends and launches a new kernel. For irregular graphs, it is costly and not flexible. Our approach presents a runtime resource adjustment approach for BFS and can be tailored for model checking problems. (2) We propose an approach to build dynamic parent-child relationship and a dynamic hierarchical task scheduler for dynamic load balancing. (3) We develop a three-level queue management to fit the dynamic parallelism and dynamic BFS layer expanding. Based on it, we propose a dynamic path recording approach, which helps duplicate elimination in BFS at the same time. Hierarchical memory structure of GPU is fully utilized for data accessing. (4) We implement our approach in PAT model checker and evaluate them to show the effectiveness of our approach.

Related Works. In the area of model checking, as the verification problem can be transformed to a graph search problem, there have been many works on accelerating model checking algorithms with CUDA. [3] focuses on the duplicate detection in external memory model checking. It utilizes GPU to accelerate sorting process in duplicate detection in BFS and builds a delayed duplicated detection on GPU. In [1], the authors propose a design of maximal accepting predecessors algorithm for accelerating LTL model checking in GPU. [4] accelerates the state space exploration for explicit-state model checking by utilizing GPU to do the breadth-first layered construction. [2] shows how the LTL model checking algorithms can be redesigned to fit on many-core GPU platforms so as to accelerate LTL model checking. [13] focuses on the on-the-fly state space exploration in GPU and proposes several options to implement this. All these research has proved CUDA compute architecture can be well utilized in solving model checking problems. In this paper, different from previous research in which most are based on a static way to allocate computing resource in advance and involve CPU frequently, we build an approach for counterexample generation which can completely put

Algorithm 1. Counterexample Generation Algorithm

```
Input: $init, SCC, \rightarrow$
Output: $\pi_{ce}$
1 $\pi_{ce} \leftarrow$ Init2SCCBFS$(init, SCC, \rightarrow)$;
2 $\pi_{ce} \leftarrow \pi_{ce} \frown$ Path2AccBFS$(\pi_{ce}, SCC, \rightarrow)$;
3 $\pi_{ce} \leftarrow \pi_{ce} \frown$ SelfLoopBFS$(\pi_{ce}, SCC, \rightarrow)$;
```

the work to execute in GPU and dynamically fit the feature of BFS. Then the dynamic parallelism and memory hierarchy from latest CUDA Architecture-*Kepler GK110* and its corresponding GPU device serve as the basis of our design.

2 Background

LTL Model Checking and Counterexample Generation. LTL model checking is to verify that a model satisfies a property expressed in LTL, which has been shown to be equivalent to checking the non-emptiness of the product between a Büchi automaton (which is the negation of the LTL property) and a system model. A Büchi automaton can be defined as a tuple $\mathcal{A} = (B, \mathcal{T}, b_i, \mathcal{F})$ where B represents a finite set of states; $\mathcal{T} \subseteq B \times B$ represents the set of transition between states; $b_i \in B$ represents the initial state, and $\mathcal{F} \subset B$ is a set of accept states. An infinite input sequence can be accepted by a Büchi Automaton if there exists an execution path that will visit an accept state infinitely often. Let AP be a set of atomic propositions. The system model can be represented with a Kripke Structure $\mathcal{M} = (S, \mathcal{I}, \mathcal{R}, \mathcal{L})$ where S is a finite set of states; $\mathcal{I} \subset S$ is the set of initial states, $\mathcal{R} \subseteq S \times S$ is the transition set and L is a labeling function: $L : S \rightarrow 2^{AP}$. Given a Büchi automaton $\mathcal{A}$ and a Kripke Structure $\mathcal{M}$, their product is defined as $\mathcal{P} = (B \times S, \rightarrow, \{b_i\} \times \mathcal{I})$, where $\rightarrow \subseteq (B \times S) \times (B \times S)$ is the product of $\mathcal{T}$ and $\mathcal{R}$.

Based on the definitions above, the non-emptiness checking is to search whether there exists an infinite run π such that π reaches a state (b, s) infinitely often and $b \in \mathcal{F}$. This is equivalent to detecting if a run contains a loop and (b, s) is included in the loop. For SCC based LTL model checking, the process is to detect an SCC containing an accepting state. When such an SCC is detected, it can be concluded that the model violates the LTL property. Counterexample generation process then start to produce a trace to reflect the errors in the model.

There are many counterexample generation algorithms [12,6,5], mostly using BFS searching to find the shortest counterexamples. Therefore, a way to accelerate BFS, combined with counterexample generation requirement, will work for these solutions. In this paper, we choose the counterexample generation algorithm (Algorithm 1) as the basis for our design. There are three inputs, *init* is an initial state in $\mathcal{P}$; *SCC* is a list that contains all nodes belong to the SCC; $\rightarrow$ is the outgoing transition relation of each node in $\mathcal{P}$. Strictly speaking, this transition relation is made up of the current explored transitions during the SCC searching process. Algorithm 1 contains three steps. (1) *Init2SCCBFS* is to find the path from *init* to any state in the *SCC* using BFS. (2) *Path2AccBFS* is to find the path from the SCC state found with *Init2SCCBFS* to the nearest accepting state. (3) *SelfLoopBFS* is to find a loop that starts from the accepting

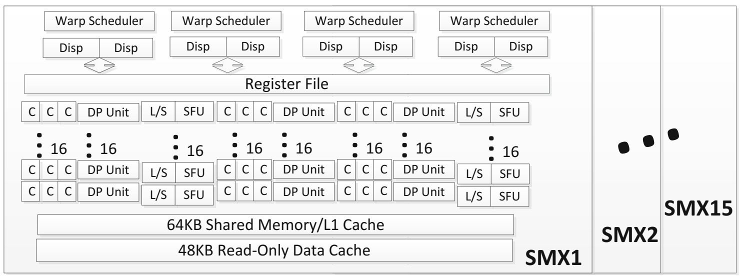

Fig. 1. Kepler-CUDA Hardware Model

state. π_{ce} is the returned counterexample run, which is the concatenation of the three path during the process. All these three steps are BFS based. In this paper, we will deal with accelerating these three steps in GPU with CUDA and merging them into one algorithm.

GPU and CUDA Architecture-Kepler. With its high parallel computational capability and wide memory bandwidth, GPU can speedup large scale data processing. CUDA is a parallel computing platform and programming model [11] designed for NVIDIA GPUs. As shown in Fig. 1, *NVIDIA Kepler GK*110 is the new GPU Computing Architecture.

On the hardware level, a GPU consists of tens of streamed multiprocessors (SMX), each of which contains a lot of streamed processors (CUDA cores, marked as C in Fig. 1), instruction units and hierarchical memory. Streamed processor is the most basic processing unit in GPU. The hierarchical memory design is common in CUDA architecture, which contains Global Memory (GM), Constant Memory (CM), Texture Memory (TM), Shared Memory (SM) and Local Memory (LM) (i.e., registers). The access rates of these memories are in the descending order: GM<CM/TM<SM<LM. SM can be used to exchange data within an SMX, and GM is used to exchange data among SMXs. For the use of the other memories, readers can refer to [11]. The hierarchical memory is critical for GPU programming as it determines the data access cost.

On the software level, applications developed with CUDA are launched by CPU and running on GPU. The running application is called *kernel*. Each kernel runs the same program in many independent data-parallel light weight threads [10]. In CUDA runtime environment, threads are organized into three levels:*warp, block and grid*. Warp is the most basic execution and scheduling unit in CUDA. A warp usually contains 32 threads. A streamed processor only handles one warp at a time. A block contains several warps, which must be executed in the same SMX. Grid is the combination of blocks. The block size and the grid size are configured when launching the grid from CPU.

Compared with previous versions, *Kepler GK110* comes with four significant updates. (1) The new multiprocessor architecture. *Kepler GK110* owns 15 SMX units in general. Each SMX contains 192 CUDA cores. The number of warp schedulers increases to 4, which means 4 warp threads can be started together. (2) The Hyper-Q, which is to enable multiple parallel CPU tasks to launch work in a single GPU simultaneously. Hyper-Q can dramatically increase GPU utilization and reduce CPU idle

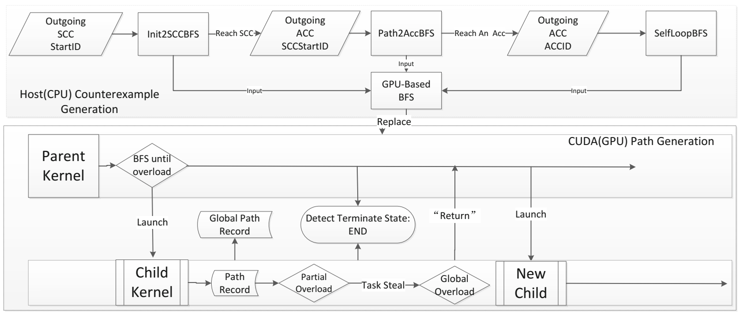

Fig. 2. Overall Design

time [10]. (3) Dynamic parallelism, which is the key feature we utilize in this paper, enables the kernel running in GPU to launch a new kernel to finish other works without involving CPU. Its presentation can be referred to *page 143* in [11]. The kernel launched from CPU is called *parent kernel*. One thread in parent grid can launch a *Child Kernel*. When the execution of the child kernel completes, it will stop and return to its parent. With this feature, the application running on GPU can make full use of the resource by dynamically launching new kernels. It can help developers to put the whole application to GPU for execution, which is efficient and cost effective. (4) The updated memory hierarchy. It introduces a 48KB cache for data known to be read-only during the execution.

3 CUDA Accelerated Counterexample Generation

The overall design of our approach is presented in Fig. 2. *Host (CPU) counterexample generation* represents the process in Algorithm 1. We build a general path generation approach to reach the target of function *Init2SCCBFS*, *Path2AccBFS* and *SelfLoopBFS* in Algorithm 1 based on different input. Our approach for handling the BFS based path generation is presented as *CUDA (GPU) Path Generation*. The complete counterexample generation process in Algorithm 1 can be replaced by executing GPU-based BFS for three times.

Our approach consists of two parts: *Parent Kernel* and *Child Kernel*. The overall process is described as follows. CPU launches the *Parent Grid* to execute *Parent Kernel*. *Parent Kernel* starts the BFS based path generation to generate one or more new layers of tasks. When the task size exceeds the thread number in *Parent Grid*, it launches *Child Grid* to execute *Child Kernel*. *Child Kernel* starts to do path generation and records path data. After generating a layer, the task scheduler will check whether any warp or block being overload. We define **overload** as the number of tasks exceeds the thread number in the GPU or no more tasks can be added to the BFS queue. Tasks rescheduling will start to do load balancing within the *Child Kernel*. If the whole *Child Grid* is **overloaded**, it will return to *Parent*. *Child Kernel* stops running. Resources of *Child Grid* are released. *Parent Kernel* reallocates tasks, launches a new *Child Grid* to execute *Child Kernel* and distributes tasks to it. The process continues until the "goal" being reached. The "goal" means terminating condition. The relationship between *Parent Grid* and *Child Grid* is dynamically adjusted according to the number of tasks. In order to maximize the parallelization, in default, each thread is asked to do the BFS and path recording for one state in BFS queue. It means that the number of tasks in each layer of BFS will decide the number of threads needed, so as to decide the structure of *Child Grid*. This dynamic relation ends at the time the process of our approach ends.

The dynamic parallelism is used to deal with the dynamic task size so as to make the execution flexible. Other features of CUDA programming model and *Kepler GK110*, such as the hierarchy memory, are integrated into each part. Our solution utilizes the latest GPU features to provide a novel counterexample generation solution.

3.1 Detailed Approach

We present Algorithm 2 and Algorithm 3 in this section for *Parent Kernel* and *Child Kernel* based on the process in Fig. 2. They follow the CUDA dynamic parallelism programming model presented in *pages 141 to 159* in [11]. Note that in CUDA, there are build-in objects *blockIdx* and *threadIdx* to record the ID of block and the ID of

Algorithm 2. CudaParentCounterexampleGeneration Algorithm

```
   Input: $init, TerminatingCondition, \rightarrow$
1  $inblocktid = threadIdx.x; inwarptid = inblocktid\%32$;
2  Define: $WarpQueue$, $WarpPathQueue$ in SM;
3  if $inblocktid = 0$ then
4      $WarpQueue[0].enqueue(init)$; $WarpPathQueue[0].enqueue((-1, init))$;
5  CUDA-API:$\_\_synthreads()$;
6  while TRUE do
7      if $WarpQueue[inwarptid] \neq \emptyset$ then
8          $S \leftarrow WarpQueue[inwarptid].Dequeue()$;
9          Shared Code with $MemoryOption = SM$
10     if $inwarptid = 0$ then
11         if $|WarpQueue| >$ WARPQUEUE_SIZE then
12             Intra_Warp_task_transfer;
13     if $inblocktid = 0$ then
14         if $|TasksInBlock| > InitialT$ then
15             break;
16         else
17             Inter_Warps_task_transfer;
18 CUDA-API:$\_\_synthreads()$;
19 if $\neg TerminatingCondition(anyState)$ then
20     if $inblocktid = 0$ then
21         $ChildSizeCalculation($EXPAND_LEVEL$)$;
22         write $WarpPathQueue$ to GM;
23         write $WarpQueue$ to GM with Duplicate Elimination;
24     while $\neg TerminatingCondition(anyState)$ do
25         if $inblocktid = 0$ then
26             Generate Tasks Distribution Offset;
27             Launch ChildKernel, Transfer tasks in GM to Child Grid;
28             CUDA-API:$cudaDeviceSynchronize()$: //If Child returns to Parent;
29 CUDA-API:$\_\_synthreads()$;
```

thread in each block. But there is no object to represent the ID of threads in warp. It can be calculated directly as the warp is built in sequence, means that threads with index $0 \sim 31$ will be warp 1.

To simplify the presentations of the two algorithms, we abstract the common part in both of two algorithms in List. 1.1. It corresponds to lines 9 in Algorithm 2 and lines 8. The details will be introduced together with the algorithms.

Algorithm 2 corresponds to the *Parent Kernel* executed in *Parent Grid*, named by *CudaParentCounterexampleGeneration*. It focuses on the task schedule and parent-child relation management. In Algorithm 2, the input variable $init$ means the initial state. $TerminatingCondition$ is a Boolean function which decides whether the algorithm should terminate at the current state. The condition in our approach means that the path generation process reaches any target state in the target states set, which can be be an SCC or an accept state list based on the input of each process in Algorithm 1. Line 1 presents two types of thread ID mentioned above. Line 2 presents the two types of queues used in the algorithm. $WarpQueue$ is an array of queues that represents the task queue for each thread. $WarpPathQueue$ has the same structure, which is to record the path to the target state. They are all allocated dynamically in SM. The details structure and operation rules can be referred to Sec. 3.2 and 3.4. Lines 3 and 4 are the first step of path generation. The initial state and initial path record are added to the queue of the first thread in the block. Here the path record is a tuple with two components: $(Predecessor, StateID)$. The function shown in lines 5, 18 and 29 is CUDA build-in API for intra-block synchronization [11]. The loop from line 6 to line 17 is the major path generation process in *Parent Kernel*. The condition to break the loop is that *Parent Grid* being overloaded. Line 7 means that the thread works when its queue is not empty. In line 8, the thread will get task S from its queue, then line 9 mentions the *Shared Code*, which is the abstraction of the BFS and counterexample generation related work. The *Shared Code* is presented in List. 1.1.

Listing 1.1. Shared Code

```
1      if($TerminatingCondition(anyState)$){
2          write $WarpPathQueue[inwarptid]$ to GM;
3          broadcast to other threads through $MemoryOption$;
4          $Iterativebacktracking \rightarrow FullPath$;
5          break;
6      }
7      $S_{new} = NewLayerTaskGeneration(S)$;
8      if ($|WarpPathQueue[inwarptid]| = WARPPATHQUEUE\_SIZE$){
9          write $WarpPathQueue$ to GM;
10         $WarpPathQueue[inwarptid].enqueue(\{S, S_{new}\})$;
11     }
12     $WarpQueue[inwarptid].enqueue(S_{new})$;
13     if($inwarptid = 0$){
14         Transfer tasks among queues in $WarpQueue$;
15    }
```

In List. 1.1, line 1 is the target state detection. When the path generation of any thread reaches any state in the target states set, path records stored in $WarpPathQueue$ in SM will be copied back to GM (using atomic operation *atomicAdd*) in line 2 and this information will be broadcasted through $MemoryOption$ in line 3. For Algorithm 2, $MemoryOption$ is set to *SM*. Then other threads will stop running and the thread which detects the terminating condition will deal with backtracking to generate the full path in line 4. Successors generation in line 7 is based on S and $\rightarrow$. In line 12, new successors S_{new} will be added to the queue of corresponding thread in $WarpQueue$. Lines 8 to 10 are to record path information. Path records will be stored in $WarpPathQueue$ in SM firstly, when the element number in the queue exceeds the constant *WARPPATHQUEUE_SIZE*, it will be copied back to GM (using atomic operation $atomicAdd$). The record in GM can also work as the preparation for future duplicate elimination, which will be detailed in Sec. 3.4. At the beginning, only thread 0 has tasks in its queue. So lines 13 and 14 are to involve other threads in the same warp by transferring tasks to threads with empty queue, which is done in central mode by the first thread in a warp.

Back to Algorithm 2, lines 10 to 12 perform load balancing within a warp. The constant *WARPQUEUE_SIZE* means the configured size of each queue in the array $WarpQueue$. Lines 13 to 17 are the inter-warps load balancing and the checking of *Parent Grid* being overload. The constant *INITIAL_T* means the thread number in *Parent Grid*. Lines 20 to 23 work on the calculation of *Child Grid* size and transfer data from SM to GM so as to transfer data from *Parent Kernel* to *Child Kernel*. Note that line 23 shows that a duplicate elimination approach takes action when copying back the content in task queue from SM to GM, which utilizes the path record information in GM. Details are also shown in Sec. 3.4. Line 26 shows that *Parent Kernel* needs to calculate the task distribution offset, which records the tasks storage index in GM for each block in *Child Grid*. Constant *EXPAND_LEVEL* means the times of *INITIAL_T* threads for *Child Grid*. Finally, lines 27 and 28 are the process to launch *Child Grid*. The loop from lines 24 to 28 is the loop in which *Parent Kernel* working as a scheduler to reallocate *Child Grid* to execute *Child Kernel* iteratively. This loop breaks only when the path generation detects any target state.

Algorithm 3 corresponds to the *Child Kernel* executed in *Child Grid* in Fig. 2. Functionally, it works on the path generation and the task schedule approach is also implemented in it. In Algorithm 3, variables or functions with the same name as in Algorithm 2 have the same meaning. The tasks in GM and the $Distributionoffset$ generated in Algorithm 2 are the inputs. In line 1, $globaltid$ represents the first thread among all blocks. Line 2 defines two variables in GM for communication among threads in different blocks. A loop from lines 3 to 32 is the major executing process. The break conditions of the loop are that path generation detects any terminal states or the whole *Child Grid* being overloaded. Lines 4 and 5 are for each warp to get its own tasks and push to the queue of each thread in balance. This is based on the $Distributionoffset$. Lines 6 to 8 are shared code with $MemoryOption$ being *SM+GM*. The full path generated will be "returned" to *Parent Kernel* through GM. Lines 10 to 22 are the intra warp and inter-warps load balancing. Lines 23 to 32 are the process to check if the whole *Child Grid* being overloaded and whether inter-blocks load balancing is needed. These three load balancing approaches make up the complete hierarchical task scheduling. And they can be regarded as three levels schedule: Warp level, means task adjustment among threads in a warp; Block level, means task adjustment among blocks of *Child Grid*; Grid level means returning the control to parent. Block level and Grid level need to copy the content in task queue to GM with the duplicate elimination approach. It decides at which level task scheduling will be taken dynamically. Lines 24 and 32

Algorithm 3. CudaChildCounterexampleGeneration Algorithm

```
Input: $Tasks, DistributionOffset, TerminatingCondition, \rightarrow$
$globaltid = blockDim.x * blockIdx.x + threadIdx.x$;
Define $WarpQueue, WarpPathQueue$ in SM $Child\_return2Parent, ChildSynNeed$ in GM;
while $\neg TerminatingCondition(anyState)$ or $Child\_return2Parent$ do
    if $inwarptid = 0$ and $interblockstaskschedulehappens$ then
        $WarpQueue[0...31].enqueue(GetTasks(Tasks, DistributionOffset))$;
    while $WarpQueue[inwarptid] \neq \emptyset$ do
        $S = WarpQueue[inwarptid].dequeue()$;
        Shared Code with $MemoryOption = SM + GM$
    CUDA-API:$\_\_synthreads()$;
    if $inwarptid = 0$ then
        if $|WarpQueue[0...31]| > WarpQueueSize$ then
            $InWarpadjustment = true$;
        if $TasksInWarp > 32$ then
            $InBlockadjustment = true$;
            $Ats = AvailableTaskSize$;
    if $inblocktid = 0$ then
        if $TasksInBlock > ThreadNumInBlock$ then
            $ChildSynNeed = TRUE$;
        else if $TasksInBlock \leq ThreadNumInBlock$ and $InWarpadjustment = true$ then
            Intra_Warp_task_transfer;
        else
            Inter_Warps_task_transfer$\Leftarrow Ats$;
    $CudaInterBlocksSyn()$;
    if $ChildSynNeed = TRUE$ then
        if $globaltid = 0$ then
            if $TasksInChild > ThreadNumInChild$ then
                $Child\_return2Parent = TRUE$;
                write $WarpQueue$ to GM;
            else
                write $WarpQueue$ to GM;
                Inter Blocks Task Scheduler;
        $CudaInterblocksSyn()$;
```

represent the invocation of the inter blocks synchronization interface. It is not CUDA built-in API. This will be described in following parts.

Specifically, Algorithm 2 is designed to be executed among threads in a block, while Algorithm 3 is to be executed among threads in multi blocks. This is because *Parent Kernel* focuses on task rescheduling while *Child Kernel* focuses on the path generation.

Some other functions are cited for Algorithm 2 and 3: function *CudaQuicksort* utilizes the dynamic parallelism feature of CUDA [11] to do quick sort for preprocessing the target states set. *CudaInterBlocksSyn* refers to the algorithm mentioned in [14]. It is for inter-blocks synchronization as CUDA does not supply API for this.

Synchronization and Atomic Operation. In our algorithm, synchronization happens in each layer expanding by default as the algorithm need to do load balancing. After any task scheduling, synchronization is needed to make sure that each thread gets its own tasks correctly. In previous algorithms, the atomic operation can be used to work as the

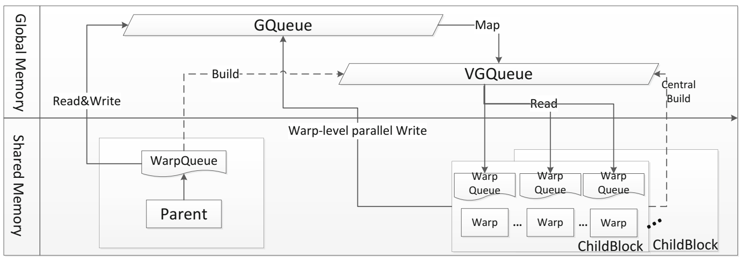

Fig. 3. Dynamic Three-level Queue Management

a $lock$. When some threads want to write the same memory address at the same time, only the first one which calls the lock will get the access right and others will discard their write operations and continue their executing.

3.2 Dynamic Three-Level Queue Management

As discussed in Sec. 2, GM can be read or written by all blocks running in different SMX, and SM is just available to blocks running in the same SMX. Read or write operations in SM cost much less than operations in GM. But the size of SM is much smaller than GM. Since our algorithm refers to huge data size, we cannot avoid accessing GM. However, as our tasks are distributed to the parallel threads, we can utilize SM to accelerate local data accessing. Considering that our algorithm is building dynamic Parent-Child relationship, we need a dynamic task distribution. We build a dynamic hierarchical queue to utilize the hierarchical memory. In order to fit our dynamic parallelism design, we build a three-level queue management approach, shown in Fig. 3. The first level queue is stored in SM, i.e., $WarpQueue$ in Algorithm 2 and 3. The second level queue is stored in GM, denoted as $GQueue$. The third level queue is also stored in GM, named Virtual Global Queue, denoted as $VGQueue$. For simplicity, we denote GQueue and VGQueue as GM in Algorithm 2 and 3.

Here, as there are many threads working together, the problem of read-write conflict when parallel threads write or read at the same time is necessary to be considered in the queue structure and the design of task schedule approach. One potential solution is to use lock or atomic operation to prevent conflict, which will lead to a huge cost with frequently write requests at the same time. Another potential solution is to use lock-free structure is preferred. We take two types of lock-free structures into consideration: first, as mentioned, the Kepler GK110 contains four warp schedulers in a single SMX, i.e., 4 warps can run in parallel. We build a lock-free queue with 4 sub-queues so as to avoid the conflict. However, it is hardly feasible because the warp scheduling in GPU is not visible to us. Therefore we adopt the design as showed in Fig. 3. In each block, no matter in *Parent Grid* or *Child Grid*, we make the first-level queue in SM a dynamic sub-queue set based on the warp number in one block. As shown in Fig. 4 part A, each $WarpQueue$ consists of 32 queues, which is due to the size of warp so as to make it lock-free. As we want to guarantee one thread holds only one expanding task, if the task size in a block exceeds the number of threads, the tasks will be re-scheduled and transferred to $GQueue$ in GM.

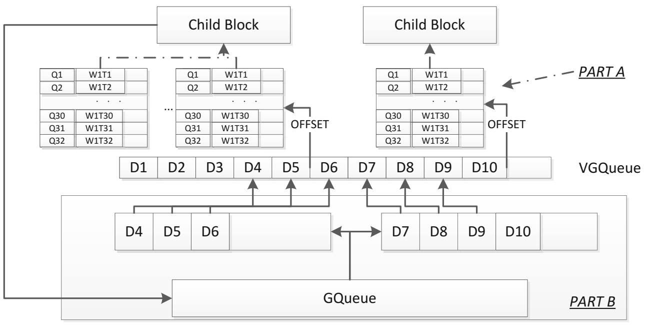

Fig. 4. Structure of WarpQueue, GQueue and VGQueue

In Fig. 3, $GQueue$ is built at the first time when *Parent Grid* launches a *Child Grid*, it is also a group of array shown in Fig. 4, part B. As the *Parent Grid* communicates with *Child Grid* via GM, which is also the way blocks communicate with each other, it is used to transfer tasks to *Child Grid* and used by *Child Kernel* to execute. In following execution, $GQueue$ stores the tasks when blocks being overloaded or the task reschedule among the blocks in *Child Grid* is needed. As in the global view, the tasks stored in the $GQueue$ is not continuous, $VGQueue$, shown in Fig. 4, is dynamically built as the third level and it is used for sequential accessing tasks data. This three-level queue follows the rules of dynamic parallelism, aiming at building a flexible way of data accessing and improving the performance. It works for the task schedule and can completely match the *Parent-Child* structure.

3.3 Dynamic Hierarchical Task Schedule

As the task size during the execution dynamically changes, unbalanced load or overload will happen frequently, especially for an irregular graph. Launching kernel is an expensive work. So we cannot rearrange the structure of *Child Grid* at each time that the unbalanced load happens. Flexible task scheduling methods are necessary. Combined with our path generation problem, there are several conditions that the program needs to do task scheduling in hierarchical level. Algorithm 2 lines 10 to 17 and Algorithm 3 lines 10 to 32 are related to these:

- The first time to launch *Child Grid* from *Parent Grid*. When *Parent Kernel* finishes some layers of BFS-related path generation and makes that *Parent Grid* cannot hold more tasks, the *Parent Grid* needs to launch *Child Grid* and schedules initial tasks to *Child Grid* and used by *Child Kernel*.
- The inside warp task transfer to make each thread has tasks in its queue. When each warp begins the execution after getting tasks, it needs to guarantee that each thread is involved in the path generation procedure.
- When the whole tasks in a warp make a warp overload, it needs to do inter warps task transfer. This is similar to the inside warp data transfer.
- When the tasks in a block make it overload, inter blocks task rescheduling will occur.

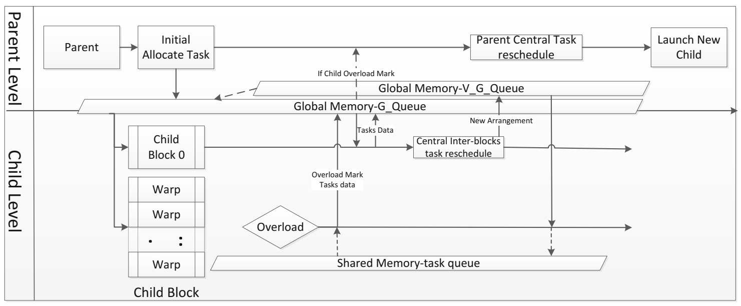

Fig. 5. Dynamic Hierarchical Task Schedule-block and grid level

– When the whole tasks in the *Child Grid* make it overloaded, *Child Kernel* will stop executing and the control will return to *Parent Grid* to rearrange the *Child grid* so as to reschedule the tasks. This and the inter blocks one are shown in Fig. 5. Both the inter blocks schedule and the *Parent Grid* schedule utilize $GQueue$ and $VGQueue$ GM to redistribute tasks. While inter warps or inside warp schedule is based on SM.

These make up a hierarchical fine-grained task scheduling. As many steps are in SM, it can make full use of the fast access feature. And only *Child Grid* being overloaded will cause the structure of *Child Grid* to be rearranged. In common, we will arrange the grid size of child bigger than needed at the beginning, to set the constant *EXPAND_LEVEL* so as to make the size of grid and block bigger than required, i.e., *INITIAL_T* $\times$ *EXPAND_LEVEL* in Algorithm 3. The *EXPAND_LEVEL* will based on the restriction of GPU architecture, which will be mentioned in Sec. 4. It is to make a compromise between resource cost and rescheduling cost. As the decision to do which level task rescheduling is due to the runtime task size, our design is a Dynamic Hierarchical Task Schedule method.

Note that after each layer of path generation, the overload detection will occur. This, together with the terminating condition detection, are in a central mode. This is to get rid of frequent communication among threads. When the whole block is overloaded and needs to copy tasks in each $WarpQueue$ back to $GQueue$, each warp will do its own transfer, makes it a parallel data transfer. Here, the targets of task scheduling are to balance workload in each warp/block and to allocate enough resources for future execution.

3.4 Dynamic Duplicate Eliminated Path Recording

Our algorithm is to deal with the counterexample generation, where path recording is necessary. Path recording should also be parallelized. As our approach performs BFS, the counterexample path is updated in each layer. Note that our path recording is to record the visited state ID and its first *Predecessor*. The "first Predecessor" means the firstly recorded predecessor. In fact, our algorithm is to find *a* path to reach the target set, So one predecessor for one state is enough to generate a complete path. Take Fig. 6

as example, record $(2, 4)$ and record $(3, 4)$ will not be recorded together, just $(3, 4)$ is recorded as it is reached earlier.

Combined with our previous design, the path recording is happening in two levels: (1) warp level in SM, each warp owns a $WarpPathQueue$, which is mentioned in Sec. 3.1. (2) block level in GM, path recording will be taken under three conditions: When the number of records in $WarpPathQueue$ exceeds the configured *WARPPATHQUEUE_SIZE*, it is executed independently in each warp and mentioned in line 9 in List. 1.1. Another two conditions are that path recording is taken before the task being copied back to GM or after the terminating condition being detected, mentioned in lines 22 in Algorithm 2 and line 9 in List. 1.1. The structure for path recording in this level is two arrays. One is the $path\_recording\_array$, the index of array represents the ID of state and the value represents the predecessor. The other is the $predecessor\_visited\_array$, different from the first array, its value represents if the corresponding state is visited. The example of this procedure can be shown in Fig. 6. When the record $(1, 2)$ is copied back to GM, it will be recorded as $path\_recording\_array[2] = 1$ then $predecessor\_visited\_array[1] = true$. And for record $(n, 3)$, as $predecessor\_visited\_array[3] = true$, this record will be discarded. But $predecessor\_visited\_array[n]$ will be marked $true$. Atomic operations are used for writing these two arrays.

We call this approach *Dynamic Duplicate Eliminated Path Recording*. The duplicate elimination here does not mean duplicate path record elimination. It is for duplicate BFS tasks elimination. When the tasks being copied back to $GQueue$, it should first detect if the corresponding value of task state in $predecessor\_visited\_array$ is true. If so, this state will not be copied back to GM for following task reschedule. So it reaches the duplicate elimination target to some extent. It is mentioned in Algorithm 2 and 3 when the algorithms proceed to *write* $WarpQueue$ *to GM*.

When the terminal states being detected, we need to generate the full path, which is mentioned in line 4 in List. 1.1. The process start from the target state reached by path generation process, marked as s. The iteration is started to find predecessor of s by getting value $prec(s) = predecessor\_visited\_array[s]$. This terminates when $predecessor\_visited\_array[s] = Init$. We generate the full path by recording each $prec(s)$ during the iteration. Atomic operation is also needed in getting the full path as we only need one path. Overall, our path recording also fits the idea of dynamic parallelism.

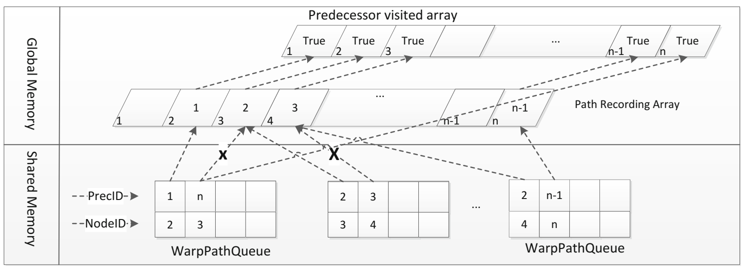

Fig. 6. Block-level Path Recording

Table 1. Parameters in the Algorithms

Parameter	Meaning	Default Value
INITIAL_T	The thread number of parent	32
WARPQUEUE_SIZE	The length of queue in WarpQueue	32
WARPPATHQUEUE_SIZE	The length of queue in WarpPathQueue	32
EXPAND_LEVEL	The times of thread number to expand compared to statistic requirement	2

4 Experiments and Evaluation

We evaluate our algorithms in two aspects. Firstly, we test the performance of our dynamic CUDA counterexample generation with models in different size and structures. Secondly, we analyze the effects of GPU parameters to our algorithms and discuss the limitation of the algorithms. We also propose two optimization options. The basic implementation of our algorithm uses C++. The system model is from PAT model checker [8]. Our experiments are conducted using a PC with Intel(R) Xeon(R) CPU E5-2620 @ 2.00GHz and a Tesla $K20c$ GPU @ 2.6 GHz with 5GB global memory, 13 SMXs and totally 2496 CUDA cores.

Performance Analysis. To analyze the performance, we choose the classic dinning philosophers problem (DP) as the input model. We use different process number to get different SCC size. The four GPU parameters used in the algorithms and their default value are shown in Table 1. The value of the parameters should be controlled in a fixed range based on hardware specifications. Their influence on our task schedule and their restrictions will be discussed in next section.

Based on the default configuration, our algorithms succeed in generating the counterexample for the verification of DP model in sizes from 5 to 8. We record the execution time for each process in *Parent Kernel* (Algorithm. 2), as well as the execution time of *Child Kernel* (Algorithm. 3). Firstly, Fig. 7 shows the distribution of the execution time for each BFS work in Algorithm. 1. *Init2SCCBFS*, *Path2AccBFS* and *SelfLoopBFS* are the three steps mentioned in Sec. 2. We can see that the first path generation costs more than the other two. This is because the $\rightarrow$ (outgoing transition table) for the first path generation is bigger as it contains all transitions generated during the model verification. So schedule, dynamic expanding and data transfer cost more. When doing $scc \rightarrow acc \rightarrow accloop$, the $\rightarrow$ is much smaller as we are preprocessing to eliminate non-SCC states in the $\rightarrow$.

We choose the data from the *Parent Kernel* execution of *Init2SCCBFS* path generation, as well as the total cost of *Child Kernel* execution. We get the results of the execution time percentage of each part, as shown in Table 2: *Schedule* means the task schedule; *Search* means the BFS with path recording; *Prepare* means the queue build-up for launching *Child Grid*; And *Child* means the execution time of *Child Kernel*. we can see *Child Kernel* will take charge of the highest percentage during the counterexample generation. In Parent, its major cost is on the initial schedule and the preparation for the child expanding. We can see the costs of each part are balanced among different size of tasks. The experimental results match the design of our algorithms.

Evaluation and Limitation. As shown in Table 1, there are four constants which affect the performance. We mentioned their meaning in Sec. 3.1. Firstly, The value of

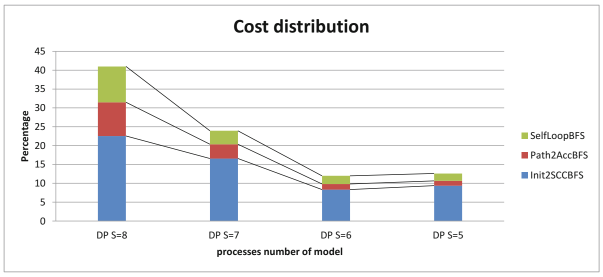

Fig. 7. Distribute of cost in three path generation

Table 2. Performance Analysis

Processes	TotalSize	SCCSize	AccSize	Schedule	Search	Prepare	Child	DataTrans	Total
5	348	120	36	20.7%	20.6%	20.5%	34.8%	0.57	9.3
6	1013	508	112	21.9%	22%	23.3%	30.8%	0.6	8.3
7	3420	2047	365	20.7%	20.6%	22.6%	35.5%	0.64	16.2
8	12339	7980	1195	24.4%	24.2%	25.4%	26.8%	1.01	19.7

INITIAL_T is due to the structure of the state space of the model. If the model's width is always short, setting a large *INITIAL_T* can reduce the chance to launch *Child Grid*. Too large value will waste a lot of resources when the Child is working on the major process. Secondly, based on our algorithm design, the hierarchical task scheduler is based on the grid size, means the number of threads. However, as the task size of each layer during the path generation of an irregular graph is unknown, if the size of one layer is larger than the remaining space of the *WarpQueue*, the task reschedule may occur, which is costly. So for the models with irregular state space, *WARPQUEUE_SIZE* will affect the performance. Thirdly, as the path records in warp level need to be copied back to GM when the $|WarpPathQueue|$ exceeds the *WARPPATHQUEUE_SIZE*. So set a large value to *WARPPATHQUEUE_SIZE* will definitely reduce the cost. At last, *EXPAND_LEVEL*, as we mentioned, if we just set the exact size of *Child Grid* according to the realistic requirement, it may cause the Child being overloaded soon and the Parent do rescheduling again. *EXPAND_LEVEL* is to make the compromise. It decides how much more resources to be allocated to the Child.

However, the size of the queue needs to be bounded. All above are restricted by the size of SM in each SMX. As described in Sec. 2, the size of chip memory in each SMX is 64KB. According to max SM per multiprocessor, only 48KB are available for SM. Before we launch the kernel, we need to decide how many SMs a block can use. In our algorithms, each item in the queue is an int. We represent the total shared memory cost as $MemC$, as defined below:

$$\begin{aligned} MemC =& sizeof(int) \times (|WarpQueue| + |WarpPathQueue|) \\ & \times PG(\textit{INITIAL\_T}) \times \textit{EXPAND\_LEVEL} \end{aligned} \tag{1}$$

It requires $MemC <$ 48KB. $PG(\textit{INITIAL\_T})$ denotes the statistic required size of resources (thread number), which starts from *INITIAL_T*. This is a dynamic variable so we combine the $(PG(\textit{INITIAL\_T})\times$ *EXPAND_LEVEL*$)$ to be $MaxWarpsize$. We can learn from equation (1) that these parameters are conditioned by each other.

Considering the restriction of CUDA architecture, the available size of SM is set before launching the kernel. If queue size described above is too large, the number of threads in one block will be restricted. During the execution, data in *WarpQueue* is flushed in each layer as old data being visited, and the size of data in *WarpPathQueue* is increasing all the time. so the constants *WARPPATHQUEUE_SIZE* and *WARPQUEUE_SIZE* will decide the extra GM accessing times. In fact, the value of all these parameters should be decided based on the structure of model.

The values of these parameters are also related to the grid level task schedule (*Parent Grid launch Child Grid*) in our design. We take the default setting in Table. 1 as an example. Suppose the total task size currently is $\mathcal{T}_{total}$. The structure of *Child Grid*, means blocks number in grid, is marked as $\mathcal{B}_c$. In default, we guarantee each block in *Child Grid* starts with tasks $\mathcal{T}_s = 32$, equals to the thread number in a warp. Then $\mathcal{B}_c = \mathcal{T}_{total} \div \mathcal{T}_s(+1)$. With the setting, each block will owns $ExpandLevel \times \mathcal{T}_s = 64$ threads, means 2 warps. With these, total shared memory cost in a block $\mathcal{TC}_B$ will be: $\mathcal{TC}_B = 64 \times (WarpQueueSize + WarpPathQueueSize) \times sizeof(int) = 16384 bytes$. Compared with $MemC$, it means two more warps can be added in a single block. and $\mathcal{B}_c \leq 13(SMX\ number)$. So the max threads number available under this setting will be 1664, means if any layer in a graph contains more than 1664 states, the schedule cannot work.

In summary, due to the restriction of SM size in CUDA, our approach does not work well for graph with large branching nodes. A solution to this problem can be using more global memory, or building united memory space with host memory, which has been proposed in the new CUDA 6.0.

Optimization Options. The experiments show that our approach is scalable in dealing with the counterexample generation problem. In our CUDA Dynamic Path Generation algorithm, the task schedule, as well as the queue building, take a substantial on the total cost. Based on this, we present two optimization options as follows. (1) According to [4], GPU works fast on short data. So building a compact graph representation to represent the model can improve the performance significantly. (2) Reduce the times of scheduling and global memory accessing. These can be done by applying latency task schedule, making each thread hold more tasks and performing the load balance after several layer expanding. The low cost intra block and warp level task schedule should take majority parts, means to increase the threshold to do inter-blocks or parent level schedule. These potential optimizations are important in the improvement of our algorithm and can be easily supported based on current design.

5 Conclusion

In this work, we proposed a CUDA Dynamic Counterexample Generation approach for SCC-based LTL model checking. We designed the dynamic queue management, hierarchical task scheduler and the dynamic parent-relation, path recording scheme by adopting the new features of dynamic parallelism of CUDA. The experiments show that our algorithm can be scalable in solving the counterexample generation problem.

In future work, we plan to optimize this algorithm to build a space-efficient encoding for the task data and path record data in order to save resources.

Acknowledgement. This work is supported by "Formal Verification on Cloud" project under Grant No: M4081155.020 and "Verification of Security Protocol Implementations" project under Grant No: M4080996.020.

References

1. Barnat, J., Brim, L., Ceska, M., Lamr, T.: CUDA Accelerated LTL Model Checking. In: ICPADS, pp. 34–41. IEEE (2009)
2. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-core GPUs. In: JPDC, pp. 1083–1097 (2012)
3. Edelkamp, S., Sulewski, D. Model Checking via Delayed Duplicatedetection on The GPU. In Technical Report 821. Dekanat Informatik, Univ. (2008)
4. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 106–123. Springer, Heidelberg (2010)
5. Gastin, P., Moro, P.: Minimal Counterexample Generation for SPIN. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 24–38. Springer, Heidelberg (2007)
6. Gastin, P., Moro, P., Zeitoun, M.: Minimization of Counterexamples in SPIN. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 92–108. Springer, Heidelberg (2004)
7. Harish, P., Narayanan, P.J.: Accelerating Large Graph Algorithms on the GPU Using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007)
8. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
9. Luo, L., Wong, M., Hwu, W.-M.: An Effective GPU Implementation of Breadth-first Search. In: DAC, pp. 52–55. ACM (2010)
10. Nvidia Corporation. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 (2012)
11. Nvidia Corporation. Nvidia CUDA C Programming Guide 5.5 (2013)
12. Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
13. Wijs, A., Bošnački, D.: GPUexplore: Many-core on-the-fly state space exploration using gPUs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 233–247. Springer, Heidelberg (2014)
14. Xiao, S., Feng, W. Inter-block GPU Communication via Fast Barrier Synchronization. In: IPDPS, pp. 1–12. IEEE (2010)