GPU Accelerated On-the-fly Reachability Checking

Zhimin Wu*, Yang Liu*, Jun Sun†, Jianqi Shi‡, Shengchao Qin§
*Nanyang Technological University, Singapore
†Singapore University of Technology and Design, Singapore
‡East China Normal University, China
‡Shenzhen University & Teesside University, United Kindom

***Abstract*—Model checking suffers from the infamous state space explosion problem. In this paper, we propose an approach, named GPURC, to utilize the Graphics Processing Units (GPUs) to speed up the reachability verification. The key idea is to achieve a dynamic load balancing so that the many cores in GPUs are fully utilized during the state space exploration.To this end, we firstly construct a compact data encoding of the input transition systems to reduce the memory cost and fit the calculation in GPUs. To support a large number of concurrent components, we propose a multi-integer encoding with conflict-release accessing approach. We then develop a BFS-based state space generation algorithm in GPUs, which makes full use of the GPU memory hierarchy and the latest dynamic parallelism feature in CUDA to achieve a high parallelism. GPURC also supports a parallel collaborative event synchronization approach and integrates a GPU hashing method to reduce the cost of data accessing. The experiments show that GPURC can give significant performance speedup (average 50X and up to 100X) compared with the traditional sequential algorithms.**

I. Introduction

Model checking is an automatic technique for verifying finite state systems. As the number of state variables or processes increases, the size of the state space grows exponentially and results in the *state space explosion* problem [6]. On-the-fly verification is one of the most widely used approaches to deal with the state space explosion problem [12]. Traditionally, on-the-fly verification is DFS-based for its memory efficiency. However it is known that DFS is hard to be parallelized. BFS is widely used in multi-core and many-core based verification algorithm design due to the fact that the state space can be easily partitioned and distributed independently. Because the state space is unknown during the verification process, the key challenge in these researches is to achieve a fully distribution of the state space so that the different cores can be fully utilized for maximum parallelization.

GPUs have been widely applied to accelerate computation in many areas, including model checking problems [13], [3], [9]. The challenges of effectively utilizing GPU for model checking are the redesign of the data structure and the algorithm mechanism to fit the architecture and compute model for GPUs (e.g., memory hierarchy and single instruction multiple data (SIMD)[1]). In this paper, we propose an on-the-fly reachability checking approach in GPU, which is realized in a tool named GPURC. The core algorithm is a parallel GPU accelerated BFS-based state space generation algorithm performed on the compacted GPU encoding of system models. The parallelism during the state space generation process is dynamically adjusted so as to fully utilize the many cores resources for GPUs to improve the performance. The execution process can be flexibly adjusted according to different system features, e.g., global variables and event synchronization.

Our key contributions of this work are as follows. 1) We propose a compact GPU encoding of concurrent system models with the support of global variables and event synchronization. Our approach can support systems with a large number of concurrent components using a multiple integer encoding. 2) We develop a BFS-based searching algorithm in GPU with dynamic load balancing without CPU involvement, using the latest dynamic parallelism feature of the Kepler architecture. 3) Our approach incorporates an efficient hierarchical hash structure to store the state space and uses parallel state space generation to achieve event synchronization. A conflict-release accessing model is used to support the multi-integer encoding. 4) Experimental results show that our approach can achieve up to around 100X speedup on benchmark examples compared with the sequential BFS-based and DFS-based algorithm. When exploring the complete state space is necessary, our approach can give up to 8X speedup.

Related Work GPUs have already been used for solving model checking problems, e.g., state space exploration and duplicate elimination problems. [10] presents a smooth interplay of a bitvector state space representation and the GPU accelerated BFS based on bitvector. It also integrates perfect hashing, but it is not suitable for an on-the-fly verification process. [18] accelerates the state space exploration for explicit-state model checking by utilizing GPU to do the breadth-first layered construction. [11] proposes how to use GPUs in the SPIN model checker. The closest work to ours is [4], which focuses on utilizing GPUs to construct the state space on-the-fly. And their journal version [5], which expands [4] for safety verification. It does not support the system models with global variables. The compute models in [10], [18], [4] are all based on CPU-GPU collaboration, which are only for state space generation instead of verification, while we support the compute model that purely based on GPU, which is also achieved by our previous work [21] for GPU-based counterexample generation.

II. Background

II-A System Modeling and Reachability Verification

In this work, a component is modeled as a labelled transition system (LTS).

Definition 1: A *labelled transition system* (LTS) is a tuple $M = (S, Act, s_0, \longrightarrow, AP, L)$ where S is a set of states, Act

[1]It describes devices with multiple processing elements that perform the same operation on multiple data points simultaneously

978-1-4673-8581-7/15 $31.00 © 2015 IEEE
DOI 10.1109/ICECCS.2015.21

is a set of actions (events), $s_0 \in S$ is the initial state, $\longrightarrow \subseteq S \times Act \times S$ is a transition relation, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function.

A transition system is *finite* if S, Act, and AP are finite. For convenience, we write $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \longrightarrow$ where $s, s' \in S$ and $\alpha \in Act$. A concurrent system may consist of multiple components running in parallel, each of which is a component LTS. The behavior of the composed system can be represented by the parallel composition of all component transition systems.

Definition 2: Given two LTSs $M_i = (S_i, Act_i, s_0^i, \longrightarrow_i, AP_i, L_i)$ for $i \in \{1, 2\}$, the *parallel composition* of them is an LTS: $M_1 \parallel M_2 = (S_1 \times S_2, Act_1 \cup Act_2, (s_0^1, s_0^2), \longrightarrow, AP_1 \cup AP_2, L)$ such that $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ where $s_1 \in S_1$, $s_2 \in S_2$. The transition relation $\longrightarrow$ is the smallest transition relation which satisfies the following:

$$\begin{cases} (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2') & \text{if} \quad s_1 \xrightarrow{\alpha} s_1' \wedge s_2 \xrightarrow{\alpha} s_2' \\ (s_1, s_2) \xrightarrow{\alpha} (s_1', s_2) & \text{if} \quad s_1 \xrightarrow{\alpha} s_1' \wedge \alpha \notin Act_2 \\ (s_1, s_2) \xrightarrow{\alpha} (s_1, s_2') & \text{if} \quad s_2 \xrightarrow{\alpha} s_2' \wedge \alpha \notin Act_1 \end{cases}$$

The process of generating the global state space is to compute the parallel composition of all the components. Given a concurrent system with n components $M_1, M_2, \ldots, M_n$, a state is defined as a *state vector* $\overline{sv} \in S_1 \times S_2 \times \ldots \times S_n$ where S_i is the state set of M_i for $i \in \{1, 2, \ldots, n\}$. On-the-fly reachability verification is to search the target during the state space generation, which can be applied in safety verification. e.g., deadlock verification. On-the-fly verification can avoid generating the complete state space.

II-B GPU Preliminaries

GPUs have been widely used to accelerate scientific, engineering and industry computations. CUDA is a parallel computing platform and programming model for NVIDIA GPUs. In this paper, we utilize the GPUs with compute capability no less than 3.5, e.g., Nvidia Geforce Titan with *Kepler GK110* architecture. Generally, GPUs is made up of a fixed number of streaming multiprocessors (SMX) e.g., 15 SMX in a full version *Kepler GK110*. Fig. 1 shows the architecture of one SMX, each of which contains a fixed number of streaming processors (*C:core* in Fig. 1). The execution model for these streaming processors is called single instruction multiple data (SIMD) [7], which makes GPUs have simple control hardware but imposes heavy cost in flow control.

One of the most important features of GPU is the memory hierarchy [16]. The hierarchical memory structure consists of Global Memory (GM), Constant Memory (CM), Texture Memory (TM), Shared Memory (SM) and Local Memory (Registers). The access performance of these memories in the descending order are as follows: GM<CM/TM<SM<LM. SM is independent for each SMX. It is shared among all stream processors in a SMX. GM is the large size board memory. e.g., 5GB in Geforce Titan. The slow access to the GM is always the major aspect that affects the performance of GPU computation. The most effective global memory access could be achieved when the same instruction for all threads in a warp accesses global memory locations that are physically adjacent. In this case, the hardware coalesces all memory accesses into a consolidated one to consecutive DRAM locations [20].

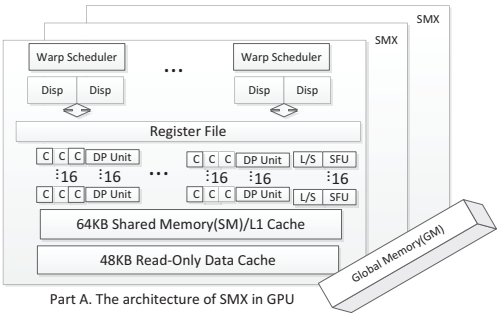


Figure 1. Architecture of SMX

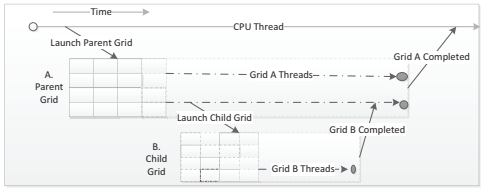


Figure 2. Dynamic Parallelism

In software perspective, the CUDA architecture defines three levels of threads organizing units: *Grid*, *Block* and *Warp*. A warp consists of 32 threads, which is the basic scheduling unit. Threads inside a warp is synchronized. A block contains a limited number of threads (e.g., 1024) and can only execute in a SMX. Grid is the organization of blocks. Threads inside a warp or block communicate through SM and threads in different blocks communicate through GM. The application running on GPUs is called *Kernel*. Note that the divergence execution in multiple threads inside a warp may follow different paths of execution, and all these paths are executed sequentially instead of in parallel, which is called *Warp Divergence*.

Since *Kepler GK110*, GPUs with compute capability no less than 3.5 support the new feature: *Dynamic Parallelism*. Different from the previous Fermi architecture, it gives kernels the ability to launch new tasks in GPU from itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths. This feature makes GPU computation fully independent from CPU. In CUDA 5+ architecture, the dynamic parallelism is described as in Fig. 2. The kernel launched from CPU is defined as *Parent Kernel*, while the corresponding execution environment is *Parent Grid*. The kernel launched from *Parent Kernel* is *Child Kernel*, with the execution environment *Child Grid*.

Defining a parallel-friendly data structure is an important GPU research topic [15]. The most important aspect for designing this kind of data structure should be the efficient update and access with millions of elements. In CPU, hash table is the most widely used data structure to fit these requirements. However, hash table based techniques used in CPU often cannot be translated directly in GPU [1]. Recently, *Cuckoo hashing* has been adopted, e.g., [1], [2], [4], by integrating multiple hash functions to provide each element a fixed number and random store positions. It avoids collisions by moving

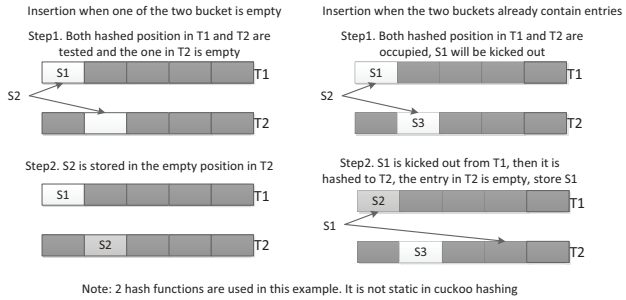


Figure 3. Cuckoo Hashing

elements around instead of fixing them in their initial positions. This technique can easily fit the memory access requirements in GPU. As it just takes few steps to read or update the elements, the number of uncoalesced memory accesses are reduced. An example is presented in Fig. 3 to describe the process of *Cuckoo hashing*.

III. GPURC Overview

This section presents the overview of GPURC design, which aims to achieve the following goals in parallelizing the reachability verification in GPU: 1) Be able to design an efficient algorithm to fully parallels in the GPUs. 2) Be able to reduce the memory cost for large scale state space. 3) Be able to support LTS models with different features.

The core of GPURC is a BFS-based on-the-fly state space generation algorithm, as shown in Algorithm 1. We parallelize the BFS process in GPU by distributing the expanded states to a large number of threads in different blocks. The whole verification process is on the fly such that if a target state (a goal state in terms of the property to check) is found by any thread in any block, the verification terminates.

The input of the algorithm is a succinct representation of the global transition system M and the reachability condition ϕ. The output is the reacheability verification result. A number of blocks[2] are started on GPUs and each block has a number of concurrent threads. In this work, we introduce the notion of *thread group* [4] as the logic grouping of threads for the purpose of concurrent generation of the outgoing transitions. Algorithm 1 is executed on all the threads in the GPU concurrently. The corresponding block ID (bid), thread group ID (gid) and thread ID (tid) are identified in line 1. The number of threads in each thread group equals to the number of component LTS in M. The number of thread groups is the thread number per block divided by the thread group number in a Warp as the synchronization of threads inside a warp can be maintained all the time. For example, if a warp consists of 32 threads, the number of components is $|M_i| = 5$, and a block has 512 threads, then the number of thread groups is $(512 \div 32) \times (32 \div |M_i|) = 80$.

Each block has a global working list Ω_{bid}, which is stored in GM and can be accessed by all other blocks. Ω_{bid} is initially empty for all blocks, except block 0. Block 0 starts the BFS from the initial state s_0; all other blocks are waiting until some states are inserted in their global working list.

Algorithm 1: PBFS: Breadth-First Search for GPU

input : $M = (S, Act, s_0, \longrightarrow, AP, L)$: the global transition system. ϕ: the reachability condition
output: Yes/No

1 Let bid, gid and tid be the current block ID, thread group ID and thread ID respectively;
2 Let Ω_{bid} be a working list, which contains the initial state s_0 for block with $bid = 0$;
3 **while** $true$ **do**
4 Let Ω be the thread group queue for breadth-first search;
5 Initialize Ω using the states in Ω_{bid} based on gid ;
6 **while** Ω *is not empty* **do**
7 Remove a state s from Ω;
8 **if** s *is not visited* **then**
9 Mark s as visited;
10 **else**
11 continue;
12 **if** $s \in \phi$ **then**
13 **return** Yes;
14 **foreach** *state* s' *such that* $s \xrightarrow{\alpha}_{tid} s'$ *and* s' *is not visited* **do**
15 **if** $s' \in S^{\phi}$ **then**
16 Insert s' into Ω;
17 **else**
18 continue;
19 Mark Ω_{bid} as empty: Define $Status[]$ and $Status[bid] = true$;
20 $Status[0..|M|]$: break when no element in array is $false$;
21 **return** *No*;

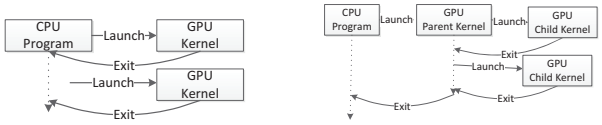
Each thread group has a private local queue Ω in SM, which is initialized using one state in Ω_{bid} as shown in line 5. If Ω_{bid} has more states than the maximum number of thread groups in a block, the extra states remain in Ω_{bid} for next round execution. From lines 6 to 18, each thread group performs the BFS searching for reachability detection. Here we use a global hash table (refer to Section IV-B) to check whether a state has been visited. The hash table is stored in GM and can be accessed by all blocks. Each thread in the thread group is in charge of one LTS in M for outgoing transition generation. This is the reason why the number of threads in each thread group equals to the number of component transition systems in M[3]. At line 14, each thread generates transitions for the corresponding LTS based on the transition relations. The event synchronization is handled differently as explained in Section IV-B.

This BFS is an iterative process. If the global working list Ω_{bid} is not empty and the local working list Ω has space, states are transferred to Ω. In line 19, when both Ω_{bid} and Ω become empty, we define a Boolean array $Status$ in GM, and the running block marks $Status[bid]$ as true. Then in line 20, the block reduces all elements in $Status$ to find if there is a false value. The whole algorithm terminates when all elements in $Status$ are true, otherwise the block continues to wait for new states.

During the BFS-based process, the workload is changing all the time, so we adopt a dynamic BFS process to adjust the parallelism such that we can help make full use of the many cores in GPU. To fit both Fermi and Kepler architecture,

[2]Grid is another way to organize threads for matrix computation, which is not used in this paper.

[3]For systems with concurrent components more than 32, each thread takes charge of the successor generation of more than one component LTS.



(a) Execution Model in Fermi (b) Execution Model in Kepler

Figure 4. Execution Model in Fermi and Kepler

we integrate the BFS-based process in two execution models. Fig. 4(a) shows the normal execution model in Fermi, which is a CPU-GPU collaborative process and is the most widely used in current research. Fig. 4(b) shows the execution model in Kepler, which utilizes the new dynamic parallelism feature to be a GPU-pure process without the involvement of CPU, which integrates the concept of Parent Kernel and Child Kernel. Details are shown in the first paragraph of Section IV.

To fit the computation in GPUs, it requires us to build a compact encoding for both the input models and the generated state space. During the state space generation process, there are a large number of memory accesses that affect the performance. We integrate the efficient GPU hash structures such that we can access the location to store the state within a fixed number of steps. Duplicate states elimination can also be handled by hash, mentioned in Algorithm 1. In addition, we integrate an efficient data transfer approach for the process to transfer data together with the adjustment of parallelism. Details are shown in Section IV-B.

IV. GPU Accelerated On-the-fly State Space Generation for Reachability Verification

In this section, we explain the GPU searching process in Algorithm 1. Our searching process can dynamically adjust the parallelism, which can be integrated into two different execution models as shown in Fig. 4. In this work, we focus on the integration of our approach to the one in Fig. 4(b) with the latest GPU technique. We present the GPU computation process in Fig. 5, which is a two-level scheduling model based on the dynamic parallelism of CUDA and GPU memory hierarchy. Both the *Parent Kernel* (presented in Algorithm 3) and the *Child Kernel* (presented in Algorithm 4) are the extended runtime implementation of the BFS searching process in Algorithm 1.

Different types of memory in GPU hierarchical memory structure have differences in access rate and size. Therefore, we propose to use hierarchical hash structures for data accessing and storage. As shown in Fig. 5, *GlobalHash* and *LocalHash* are created in GM and SM as the GPU implementation for Ω_{bid} and Ω in Algorithm 1, respectively. The details about how they work and the hash function are introduced in Section IV-B.

The GPU computation starts from ①. The *Parent Kernel* is launched from CPU to start the verification with the initial state s_0. *Parent Kernel* is concurrently executed by many thread groups. In ②, each thread group independently proceeds the successor generation and synchronization. Generated new states are stored in *LocalHash*. If a collision happens, defined as the overflow of *LocalHash*, a *Child Kernel* is launched from the *Parent Kernel* to allocate more computation resources, as shown in ③. The unvisited states are transferred from *LocalHash* of *Parent Kernel* to *GlobalHash* so as to be

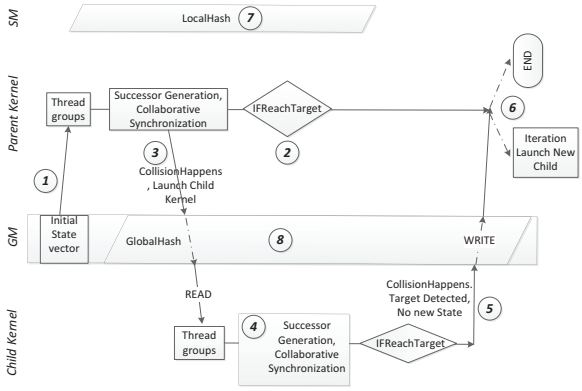


Figure 5. GPUDV with Dynamic Parallelism

transferred to *Child Kernel*, shown in ⑧. The *Child Kernel* performs the same computation as the *Parent Kernel*, as shown in ④, which is also executed by many thread groups. In runtime, when a target state is detected, all thread groups in the same block are notified through a *bool* mark in SM, and these thread groups terminate. If there is no target state, an array in SM works for the status recording of all thread groups, which has the same function as $Status[]$ defined in line 19 of Algorithm 1. During the execution, *Parent Kernel* terminates after all *Child Kernel*s finish their execution, which occurs in three conditions, as shown in ⑤:a) a collision happens in *LocalHash*, b) a target state is found and c) no more new state, i.e., the state space being completely generated and there is no target state. In ⑥, *Parent Kernel* continues if *Child Kernel* terminates in condition a), and finishes its execution if *Child Kernel* terminates in condition b) and c).

In our approach, we bring in the efficient GPU hashing design and combine it with the latest features in GPU architectures. We build a parallel collaborative synchronization and data transferring approach to improve the performance of the state space generation. Note that our approach is a BFS-based, but the search is not strictly layer by layer. We describe more details in Section IV-B. These approaches are independent to the GPU execution models.

IV-A System Encoding in GPU

The performance of graph traversal in GPU is highly affected by the memory access pattern. Each component LTS can be considered as a sparse graph with imbalanced structure. Hence adjacency matrix is not the suitable data structure for GPU computation. A compact encoding to represent the LTS is necessary. To this end, we build a minimal bit-cost encoding for LTSs.

Assume the global state space is the parallel composition of n LTSs denoted as $M_1 \parallel M_2 \parallel \cdots \parallel M_n$, and $M_i = (S_i, Act_i, s_0^i, \longrightarrow_i, AP_i, L_i)$ for $1 \leq i \leq n$. We encode the global state space as the composition of all component LTSs

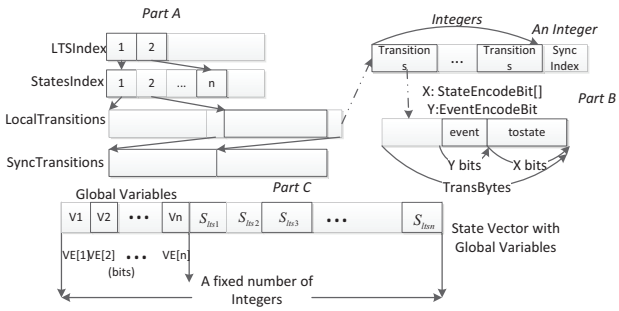


Figure 6. Component LTS Encoding

M_i, which is a four-layer integer array as explained in the following. The states in the global state space are in the form of $S_1 \times S_2 \times \ldots \times S_n$, which is encoded as a state vector. If the system model contains global variables, we encode them into the state vector with additional bits. An intuitive view on the four-array encoding can be observed in Fig. 6

Firstly, the encoding for the system model M is composed of multiple encoded M_i. To encode M_i for state space generation, we encode all states in S_i and all events in Act_i. We encode them as outgoing transitions in array *LocalTransitions* and *SyncTransitions*. An outgoing transition of a state is encoded as the montage of event in Act_i and its *tostate* in S_i. They are compact encoded with minimum number of bits. An event in Act_i is encoded with a fixed number of bits, which is equal to $\log|Act|$, where $|Act|$ is the number of all events. A state in S_i is encoded with $\log|S_i|$ bits, where $|S_i|$ is the number of states of M_i. The encoding of a transition should be aligned to a fixed number of bytes, shown in *Part B* in Fig. 6. In this way, an integer can be used to encode multiple transitions. Two types of transition, *transitions* and *transitions with synchronized events* are encoded separately in array *LocalTransitions* and *SyncTransitions*. There is an integer index from *LocalTransitions* to *SyncTransitions* in order to build the complete list of outgoing transitions from a state. Encoded transitions in *SyncTransitions* should be in order by the event ID of the transition.

The array *StatesIndex* is introduced to record the starting offset for the outgoing transitions of a state in array *LocalTransitions*. The array *LTSIndex* is used to record the starting offset for each M_i in array *StatesIndex*. The values in these four encoding arrays are static during the verification and can be bind to *TM* for fast random access.

Secondly, a state vector is encoded with a fixed number of 64bits integers. Global variables are encoded together with the state vector, which locates at the head of each state vector, the number of bits required to encode global variables is based on the number of variables and the range of their value, shown in *Part C* in Fig. 6. Different types of global variables are transferred to integers. e.g, for boolean type, 1 means true, 0 means false.

We introduce the concept of *thread group* in Section III, which consists of threads inside the same warp. As the synchronization of threads inside a warp can be maintained all the time, and tasks are independently proceeded inside a thread group so the runtime execution has *Coarse-grained* parallelism, which reduces the cost for synchronization.

IV-B State Space Generation in GPU

Based on the proposed system encoding, we explain the state space generation process in this section.

State Space Hashing and Duplicate Elimination For on-the-fly verification, the size of the state space is unknown. It is important to find an effective way to store the state vectors. In our previous work [21], we build arrays in both GM and SM to store data, but it is hard to define the size of the arrays with the load balancing problem among all threads, which always results in a sparse storage and is also not efficient for random access. In this work, we adopt hash tables to solve this problem.

Fig. 5 shows that our hash structures consist of *LocalHash* in SM and *GlobalHash* in GM. The hash methods for the two hash structures are cuckoo hashing [17] combined with linear probing hashing. Generated new state vectors are firstly stored in *LocalHash*. There are two global hash tables inside *GlobalHash*: *GlobalVisitedHash* and *GlobalOpenHash*. *GlobalVisitedHash* stores the visited state vectors, which is used in line 8, 9 and 14 in Algorithm 1. *GlobalOpenHash* is used to store the generated but unvisited state vectors, i.e., Ω_{bid} in Algorithm 1. Cuckoo hashing in our approach is described in Fig. 3. The cuckoo hashing uses multiple hash functions with the form: $hash(k) = (a*k+b)\%P\%TableSize$, where P is a prime number. a and b are a set of values which are generated randomly.

Duplicate elimination works on *LocalHash*, *GlobalOpenHash* to avoid storing duplicated unvisited state vectors and works on *GlobalVisitedHash* to avoid the successor generation for a visited state vector. As each state vector has its own hash value, the same state vectors have the same hash value. However, the hash value is not unique to a state vector, which means two different state vectors may also have the same hash value. These can cause more work to do duplicate detection as we need to compare the value of state vectors instead of just comparing their hash value. And we also integrate the linear probing if there is no available hash position to store the state vector. With these, the duplicate elimination cannot completely avoid duplicates. It should be noticed that the duplicate detection results can be *true-negative* but never be *false-positive*. So there is no missing state vector.

State Space Generation with Dynamically adjusted Parallelism State space generation is a BFS searching process based on the compact system encoding. Each thread group generates the successor states by decoding the transition relations in the four encoding arrays. Newly generated state vectors are stored in *LocalHash*, in which each thread group gets new state vectors to handle. Once a thread group finishes the successor generation, the state vector handled by it is stored in *GlobalVisitedHash*. Based on our design in Section III, successor generation process is scheduled dynamically and non-deterministically : 1) The parallelism for successor generation is dynamically adjusted based on the collision circumstance shown in Fig. 5. 2) The successor generation is not restricted to a BFS. As generated states are randomly distributed in the *LocalHash*, there is no guarantee that thread group gets states

Algorithm 2: Collaborative Synchronization

```
   Input: SyncTransition
1  Define Shared : SyncEInterC[], SyncSInterC[], SyncMark[];
2  index = 0, m = numof(M_i), tgid = threadindexinthreadgroup;
3  while true do
4      GetMinSyncT(&SyncEInterC[], &SyncSInterC[]);
5      if SyncEInterC[] = 0 then
6          break;
7      if
       SyncEInterC[tgid] ≤ SyncEInterC[0...tgid−1, tgid+1...m]
       then
8          leqthanall++;
9      if leqthanall = m − 1 then
10         while i < m do
11             if SyncEInterC[tid] = SyncEInterC[i], i ≠ tid then
12                 SyncSVec(SyncSInterC[tgid], SyncSInterC[i]);
13                 SyncMark[tgid] := true;
14         Eliminate duplicate sync result;
15         index++;
```

layer by layer by following the BFS mechanism. The benefit is that it can work like a DFS to some extent. It has potential to reach the target (e.g., deadlock state) faster.

No matter which execution model we use, i.e., Fig 4(a) or Fig 4(b), data transferring occurs with the parallelism adjustment. For *Read* operation from *GlobalOpenHash* to the local working list of thread groups. i.e., Ω in Algorithm 1, all threads access the *GlobalOpenHash* in GM in order for coalesced access, then use atomic operation to fill up the local working list for each thread group. For *Write* operation from *LocalHash* to *GlobalVisitedHash* and *GlobalOpenHash*, we use the hash operation mentioned previously.

Collaborative Synchronization Synchronization operation occurs during the interleaving state space generation process. Different from [4], which uses a central mode - one thread sorts all the time to synchronize and other threads just wait based on the warp divergence of GPU, we design a parallel collaborative synchronization approach. Instead of using several bits to mark the synchronized events, we encode all transitions with synchronized events in the same array, mentioned in Section IV-A. The process is shown in Algorithm 2. We define the shared space inter a thread group a tuple (*SyncEInterC*, *SyncSInterC*) in line 1 in Algorithm 2, Which stores a state's outgoing transition: the ID of event in Act_i and its tostate in S_i. The synchronization process starts with the ordered *SyncTransitions*, where the ID of event stored in *SyncEInterC* represents the smallest event ID. It is an iterative process that all threads are involved in the judgement if their event id stored in *SyncEInterC* is the smallest among all threads, shown in line 8. If so, the threads try to find the transition to synchronize by searching each threads' *SyncEInterC* in line 9 to 13. It may generate several same state vectors but we guarantee only one being stored in *LocalHash* in line 14. After that, each thread reads the next outgoing transition to *SyncEInterC* and *SyncSInterC*. For a intuitive view, we also propose the structure of collaborative synchronization in the Fig. 7.

The synchronization process in Algorithm 2 occurs only inside each thread group. Thread synchronization among all threads in a thread group is guaranteed. Each thread in a thread group owns a shared space for communication with

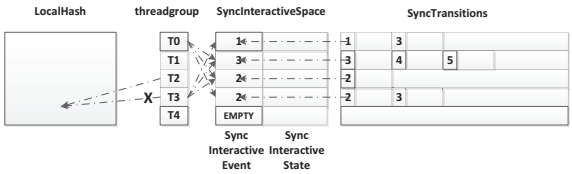


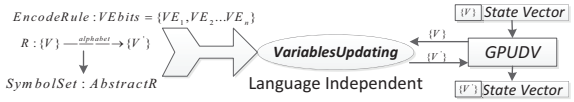
Figure 7. Collaborative Synchronization

other threads in the same thread group. Each thread reads the synchronization event in ascending order, it guarantees that the events with a minimum ID appear at the same iteration. So the synchronization works in an ascending order with the event ID. The conditions for the process to move to next event are that 1) no synchronized event exists in other components 2) it finishes the synchronization based on current event.

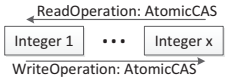
IV-C Supporting Global Variables and Large Number of Concurrent Components

For system models with global variables, the value set of the global variables is updated during the state space generation process. The value set updates based on the alphabets of the transition systems and the updating rules vary from different systems models. For general usage, we build the interface to support global variables independently to the state space generation process.

We define the structure for global variables as a tuple $GVS = (V, E, L, R, AR)$ where V is the set of global variables, E is the set of alphabets (the event ID) for all LTSs M_i. R is the set of formulas that define the rules to update the value of variables based on alphabets. AR is named from abstracted R, which is a set of symbols and encoded with bits. To generate the new value set of global variables V' for the successor is to perform mapping and analysing operation, $V' = L(V) : AR \leftarrow E, R$. The mapping and analyzing operation L is specific to different system models. Then we make our GPURC work as a common framework and supply two interfaces to support global variables: 1) *EncodeRule*. We mention in Section IV-A that the value set of global variables is encoded at the head of each state vector. This interface gets V as the input and calculates the bits needed to encode each global variables and builds an array to store the information for GPURC. 2) *VariablesUpdating*. Shown in Fig. 8(a), this interface performs the function of L. It decodes AR and works on the current value set of global variables to output the new value set. This is completely language or platform independent. The key idea is to construct AR, which is built by simplifying the formula of R, which represents the value change of variables with alphabets, to a fixed number of symbols so that it is easy to store and transfer to GPU for execution. e.g., $S_i \xrightarrow{a} S_j, V_i = V_i + 1, (V_i \leq Max)$ is abstracted as i, a, Max, and in *VariablesUpdating* we recognise these symbols as variables should be more close to Max, then the variables should plus one to finish this operation. In addition, for systems in which the event id can be matched to specific rules regularly, the construction of AR can be simplified more by updating global variables just based on event id. e.g., $ReaderWriter$ model in Section V-A, in which the events



(a) VariablesUpdating



(b) Conflict Release R/W

Figure 8. Support Global Variables and Large Number of Concurrent Components

with $eid\%2 == 0/eid\%2 == 1$ indicate the same rules to update global variables.

Besides supporting the system models with global variables, the other challenge is to support a large number of concurrent components, which means the number of components exceeds a threshold that a single 64bits integer is not enough to encode a state vector. So multiple integers are required. However, encoding a state vector with more than 1 64bits integer means the hash store operation cannot finish in one step as an atomic operation can handle at most a 64bit integers at a time. During the GPU state space generation process, the high parallelism results in a random write/read order. If we write/read operate on multiple integers with the same order as we do on single integer, it has the high probability to cause conflict and inconsistency. We deal with this problem by two approaches: 1) Read and Write operate in the opposite order with *atomicCAS* operation, shown in Fig 8(b). Only when the atomic operation succeeds in reading/writing the position to get/store the last/first integer of the state vector, the read/write operation continues. 2) The hashing method we used originally, the cuckoo hashing, should be changed to a double hashing together with linear probing. This change is to get rid of the inconsistency caused by the data exchange operation in cuckoo hashing, which requires several times' exchanging as a state vector is encoded in more than one integer.

IV-D Algorithms

In this section, we present the algorithms for the process in Fig 5. We describe the algorithm of *Parent Kernel* and *Child Kernel* based on dynamic parallelism. In fact, both of the parent kernel and child kernel handle the same work. Their relationship is based on the parallelism adjustment. And the algorithms shown can be easily transferred to the execution model in Fermi in Fig 4(a).

Parent Kernel is described in Algorithm 3 and *Child Kernel* is described in Algorithm 4. Note that we use bid, gid and tid as the block ID, thread group ID and the thread ID respectively as in Algorithm 1. We define $warpid$ to represent thread ID inside a *Warp*, and define $tgid$ as the thread index inside a thread group.

In Algorithm 3, the input is the encoded transition system M. s_0 is the set of initial state vectors. $|M_i|$ is the number of LTSs. In line 2, $GroupStore[]$ is the local working list of *thread group* to store the state vector it works on. The first thread in each thread group x transfers an unvisited state vector from I to its $GroupStore[x]$. Lines 3 to 28 are the major process. In line 4, each thread y in a thread group decodes and gets the state S of LTS_y based on $GroupStore[x]$ and its $tgid$. In line 5, the function $GetAllSucc$ initializes $SuccIdx$, which is a tuple ($LTbeginInt$, $LTendInt$, $STbeginInt$, $STendInt$) to index all outgoing transitions of S in *LocalTransitions* and *SyncTransitions*. In line 7, the first thread in a thread group detects if the $GroupStore[x]$ has been visited by accessing *GlobalVisitedHash* with *cuckoo hashing*. Then it enters the process in line 9 and line 10 if necessary, which is the process mentioned in Section IV-B and IV-C.

As the size of the *LocalHash* is limited, during the process of successor generation, collisions happen when the saturation of the *LocalHash* overflows, which means the *LocalHash* can not hold more insertions. If there is no collision, the process continues from line 26 to 20. All threads attend to get new state vectors randomly from *LocalHash* to fill up the $GroupStore$ for all thread groups, and start a new iteration.

If collisions have happened, no matter which execution model is integrated, all data in *LocalHash* is hashed to *GlobalOpenHash* in line 14. Current grid works as a *Parent Grid*. In the loop from line 16 to 20, the first thread in the block launches the *Child Grid* with more blocks and distributed data in *GlobalOpenHash* to *Child Kernel*.

In Dynamic Parallelism, the CUDA interface *cudaDeviceSynchronize* is used to synchronize between *Parent Kernel* and *Child Kernel*. In line 18, the *Child Kernel* finishes its execution and synchronizes with *Parent Kernel* for the information about 1) *IfTargetDetected*. 2) Whether the collision happens in *Child Kernel*. 3) Whether there is no unvisited state vector. Based on this information, *Parent Kernel* decides to either exit or calculate and allocate new size of resources to launch new *Child Kernel*. The interface in line 19 is integrated from [16], which is a high performance reduction approach. We use it to calculate the number of unvisited state vectors in *GlobalOpenHash*.

The function and process of Algorithm 4 are similar to Algorithm 3. So we ignore the duplicate description in line 7. The differences between Algorithm 3 and Algorithm 4 are: 1) In line 5, each thread group gets state vector from *GlobalOpenHash*. 2) In lines 9 to 10, after transfer data back to *GlobalOpenHash*, *Child Kernel* exits execution. 3) In line 11, if no more new state vectors exist, *Child Kernel* exits the execution.

It can be shown from the algorithms that the *Parent Kernel* needs to iteratively launch the *Child Kernel* and should not terminate until all *Child Kernels* terminate. Parallelism of *Parent Kernel* is static while it is flexible for *Child Kernel* to adjust parallelism. These motivate us that in our BFS-based state space exploration, we allocate little scale parallelism for *Parent Kernel* and let the *Child Kernel* achieve the high parallelism to finish the tasks as soon as possible.

V. Evaluation

GPURC is developed in CUDA C++ with two variants: 1) GPURC-GC: implemented in the CPU-GPU collaborative execution model in Fig. 4(a). 2) GPURC: implemented in the GPU-pure execution model in Fig. 4(b). It is the implementation of Algorithm 3 and Algorithm 4.We evaluate

Algorithm 3: Parent Kernel Algorithm

Input: Compact Encoding of M, s_0, $|M_i|$, GVS

```
if tgid = 0 then
    GroupStore[gid] = s0;
while ¬ifDs do
    S ← GetStateinV(tgid, GroupStore[gid]);
    ifanyOutgoing ← GetAllSucc(S, &SuccIdx);
    if tgid = 0 then
        ifdup ← DuplicateElimination(GroupStore[gid]);
    if ifanyOutgoing and ¬ifdup then
        Successor generation,Collaborative Synchronization
        → ifcollision;
        OPTION:GVS for system models with global variables.
    IFTargetDetected() → ifDs;
    if ¬ifDs then
        if ifcollision then
            cuckoo hash store&linear
            probing:LocalHash → GlobalOpenHash;
            if tid = 0 then
                while ¬ifDs do
                    LaunchChildKernel;
                    CUDA-API:CudaDeviceSynchronize();
                    cudahighperformanceReduce → NoS
                    CUDA-API:CudaDeviceSynchronize();
                    Adjust Parallelism based on the NoS: number of
                    state vectors in GlobalOpenHash;
    else
        break;
    CUDA-API:__syncthreads();
    index = 0;
    if LocalHash[tid]! = NULL then
        atomicAdd(Index);
        Index < |threadgroups|?LocalHash[tid] →
        GroupStore[Index] : donothing;
    Index = 0?GlobalOpenHash → GroupStore : continue;
```

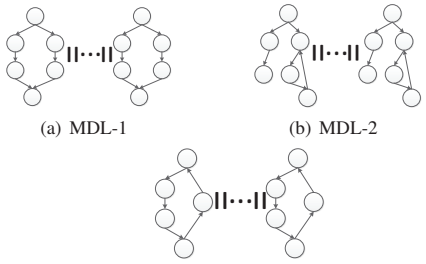
Algorithm 4: Child Kernel Algorithm

Input: $GlobalOpenHash$, Compact Encoding of M, $|M_i|$, GVS

```
S ← Read(GlobalOpenHash[tgid + |threadgroups| * bid]);
Index = 0;
if S! = NULL then
    atomicAdd(Index);
    Index < |threadgroups|?S → GroupStore[Index] :
    donothing;
while ¬IFTargetDetected do
    The process is same to Parent Kernel;
    if ¬ifcollision then
        cuckoo hash store&linear probing: LocalHash to
        GlobalOpenHash;
        "return" to parent kernel;
    Get new state vector from LocalHash,same to Parent Kernel
    Index = 0?GlobalOpenHash → GroupStore : exit;
```

the performance of GPURC-GC and GPURC by comparing them with the traditional sequential state space generation for deadlock verification algorithm, which is implemented based on the PAT model checker [19] and named as **DFS** for DFS-based algorithm and **BFS** for BFS-based algorithm. **SPUP** means the speedup.

Our experiments are conducted on a PC with two Intel(R) Xeon(R) CPU E5 − 2670, 2.60GHz, 16GB RAM and *Geforce Titan Black* GPU with 13SMX, 6GB GM and 48KB SM in each SMX. The compute capability of the GPU is 3.5 based on Kepler GK110 architecture. The execution model in Fermi in Fig.4(a) is common for all GPU.



(a) MDL-1 (b) MDL-2

(c) MNoDL

Figure 9. The Manual System Model for Evaluation

V-A Performance Evaluation

We take three sets of experiments. All experiments are taken several times and the **Speedup** means the average speedup. In the experiments, we assign 512 threads per block. Note that the parallelism (the number of GPU blocks) of GPURC is dynamically adjusted during the execution. We define **MPblocks** as the maximum parallelism, i.e., the maximum number of blocks in GPU that can be allocated during the execution, which in all experiments is set to 5000 blocks. We build two types of system models manually: *ManualDeadlock* **(MDL)**, which consists of two types **MDL-1** and **MDL-2**, and *ManualNoDeadlock* **(MNoDL)**, separately shown in Fig. 9. Both of them are composed of multiple component LTSs. Component LTS M_i in *ManualDeadlock* contains a deadlock state at the bottom layer. Component LTS M_i in *ManualNoDeadlock* is a circle structure. The **features** of all models for the experiment are shown in Table I.

In the first set of experiments, we take deadlock state as the target and all the models have deadlock state. We use different sizes of *Dinning Philosopher (DP)* model and our two types of *ManualDeadlock* models. During the execution, the parallelism is adjusted dynamically based on the unvisited state vectors in *GlobalOpenHash*. The realistic parallelism started in this set of experiments ranges from 1500 to 5000. Experiment results are shown in Table II, where we can see that in this set of experiments, compared with the traditional BFS-based sequential algorithm, our approach can give up to 70X speedup. Based on the result of *MDL-2*, we can see that compared with DFS-based sequential algorithm, our approach can bring up to 20X speedup.

The results show that GPU based approaches can quickly spread the threads to search for different parts of the state space, and hence give good speedup compared with traditional approach. This phenomenon comes from the high parallelism in GPU and the random data accessing in our approach which is not restricted to layer by layer. On the other hand, *MDL-1* and *MDL-2* contain the same number of states and transitions. But the cost of DFS-based algorithm on *MDL-1* can be ignored. We can see that the performance of DFS on reachability verification depends on the structure of models. The good performance on both *MDL-1* and *MDL-2* shows our approach is not restricted to the structure. GPURC-GC and GPURC give similar results because the number of CPU-GPU coordination is small and gives less overhead. Further, in the last row of Table II, we start 8000 blocks for the experiments. The BFS/DFS gets *out of memory exception* that cannot handle the *MDL-2* with 11 processes while the GPU

Table I. FEATURES OF TEST MODELS

Model	Proc	Ints	States	Trans
DP	13	2	$1.1 * 10^6$	$7 * 10^6$
DP	14	2	$3.42 * 10^6$	$2.34 * 10^7$
DP	15	2	$1.06 * 10^7$	$7.7 * 10^7$
MDL-1	8	1	$1.67 * 10^6$	$1.34 * 10^7$
MDL-1	9	1	$1.01 * 10^7$	$9.07 * 10^7$
MDL-1	10	1	$6.04 * 10^7$	$5.23 * 10^8$
MDL-2	8	1	$1.67 * 10^6$	$1.34 * 10^7$
MDL-2	9	1	$1.01 * 10^7$	$9.07 * 10^7$
MDL-2	10	1	$6.04 * 10^7$	$5.23 * 10^8$
MDL-2	11	1	$3.62 * 10^8$	–
DP-Free	11	2	$5.1 * 10^5$	$3.6 * 10^6$
DP-Free	12	2	$1.68 * 10^6$	$1.29 * 10^7$
DP-Free	13	2	$5.56 * 10^6$	$4.62 * 10^7$
MNoDL	9	1	$1.95 * 10^6$	$2.1 * 10^7$
MNoDL	10	1	$9.76 * 10^6$	$1.17 * 10^8$
MNoDL	11	1	$4.88 * 10^7$	$5.25 * 10^8$
RW	100	4	$5.4 * 10^5$	$1.6 * 10^6$
RW	120	4	$9.3 * 10^5$	$2.7 * 10^6$
RW	140	5	$1.46 * 10^6$	$4.3 * 10^6$
RW	160	6	$2.16 * 10^6$	$6.4 * 10^6$
RW	180	6	$3.06 * 10^6$	$9.09 * 10^6$
SP	12	1	$3.6 * 10^5$	$2.1 * 10^6$
SP	13	1	$1.06 * 10^6$	$6.6 * 10^6$
SP	14	1	$3.05 * 10^6$	$2.05 * 10^7$

Table II. PERFORMANCE EVALUATION FOR REACHABILITY VERIFICATION(TIME IN SEC)

Model	GPURC-GC	GPURC	BFS	DFS	SPUP(B/D)
DP/13	2.5	2.2	25	-	11X/-
DP/14	3.4	3	104	-	35X/-
DP/15	68	62	367.7	-	60X/-
MDL-1/8	1.2	1.4	15.3	-	12X/-
MDL-1/9	3.1	2.8	112.8	-	40X/-
MDL-1/10	9	8.3	623.3	-	70X/-
MDL-2/8	1.25	1.5	13.9	7.2	11X/5X
MDL-2/9	5.2	5.7	102.3	61.1	20X/11X
MDL-2/10	26.1	24.3	604	474	30X/20X
MDL-2/11	15.5/8000	17.5/8000	EX	EX	-/-

approach works, which is based on the compact encoding and with high parallelism for exploration, we could reduce the size of state space required to generate before reaching the target states.

In the second set of experiment, we want to see the speedup of GPURC when the complete state space is explored. We use different size of *Dinning Philosopher Deadlock Free (DP-Free)* model and our *ManualNoDeadlock* model. During the execution of the experiments, the parallelism always reaches 5000. Results are shown in Table III. Compared with traditional BFS-based algorithm, our approach can achieve up to 8X speedup. We can conclude that our approach is also available for the complete state space generation. But compare with results in Table II, we can conclude that the biggest advantage of GPURC is to handle the on-the-fly state space generation for the target searching. Note that $6GB \approx 6.44 * 10^9 bytes$ memory can store approximately $8.05 * 10^8$ 64bit integers. In our experiments, at most 6 integers are used to encode a state vector, which means at least $1.34 * 10^8$ state vectors can be stored in GPU.

In the third set of experiment, we aim to test the performance of GPURC for supporting global variables and large numbers of concurrent components. We use different sizes of *ReaderWriter* **(RW)** model and *Semaphore* **(SP)** model in PAT. Both of them contain global variables and there is no deadlock state. We do *non-executive* verification with **RW** and do the *RechabilityTest* verification with **SP**. *non-executive* verification requires to search the complete state

Table III. PERFORMANCE EVALUATION FOR COMPLETE STATE SPACE GENERATION (TIME IN SEC)

Model	GPURC-GC	GPURC	BFS	SPUP
DP-Free/11	10.1	9.5	11.4	1.2X
DP-Free/12	25	22	50.1	2X
DP-Free/13	48	42	192.5	5X
MNoDL/9	10.3	8.4	24.9	3X
MNoDL/10	44	36.3	145.4	4.5X
MNoDL/11	89	72	538.4	8X

Table IV. PERFORMANCE EVALUATION FOR SYSTEMS WITH DIFFERENT FEATURES (TIME IN SEC)

Model	GPURC-GC	GPURC	BFS	DFS	SPUP(B/D)
RW/100	33	32	36.9	-	1.1X/-
RW/120	57	55.1	72.2	-	1.5X/-
RW/140	63	61.3	112	-	2X/-
RW/160	71.5	69	177	-	3X/-
RW/180	88	83.2	234	-	3X/-
SP/12	0.25	0.22	15.6	2	70X/9X
SP/13	0.6	0.56	50.5	6	100X/10X
SP/14	1.21	1.1	151.3	14	150X/11X

space. *RechabilityTest* verification is to verify if the target state vector**RW** contains a large number of concurrent components. As shown in Table IV, We use *Ints* to represent the number of 64bit integers required to encode the state vector. For the *non-executive* verification, each state vector is encoded with multiple integers so during the state space generation process, we need to access the memory for multiple times to finish the read/write operation, which is costly and affects the performance speedup. We mention that a thread group cannot exceed a warp. In this experiment, the whole warp is a thread group. One thread in a thread group needs to take charge of several component LTSs, which is a sequential process in all threads. For the *RechabilityTest*, we always finish the searching with $Parallelism \approx 450 blocks$. We can see compared with **SeqB**, our approach can reach up to 150X speedup. Compared with **SeqD**, the speedup can reach up to around 10X. It can be concluded our approach can handle models with different features well.

Finally, the change of performance with different size of state space in GPU can be concluded from Table III. We can see that for sequential algorithm, the time cost increases almost linearly with the increasing size of state space. But for GPU algorithm, it is not increasing as fast as sequential algorithm, which reflects that we can get more speedup for larger state space. On the other hand, we can see the increment rate of speedup is decreasing, which is the feature of memory intensive GPU algorithm as the benefit of parallelism can be neutralized by the costly memory access.

Based on all our experiments, we can conclude that GPURC is efficient for the reachability test during the on-the-fly state space generation, which can be applied in safety verification. Our approach supports different types of system models and can be generally integrated to deal with other state space searching problems. Furthermore, in our experiments, for some system models with larger size than what we show in tables, the sequential algorithm will result in *OutOfMemoryException* while our approach can handle. This benefits from our compact encoding in Section IV-A.

V-B Discussion

Four points in our approach should be noticed: 1) The parallelism utilized in GPU. Theoretically, the value of *MP-blocks* defines the maximum parallelism and may have impact on the performance. In fact, we test different parallelism and

find there is no much difference among different *MPblocks*. The reason is the huge I/O cost neutralize the benefit of a larger parallelism. 2) The size of hash table used in our approach, which is restricted to the GPU memory hierarchy. When the state vector is encoded with multiple integers, the size of *LocalHash* should not exceed a threshold as the SM that available for a block is limited. e.g., $1536 * 64bit$ integers for state space encoded with single 64bit integer, $256 * 64bit$ integers for state vector encoded with > 4 64bit integers. 3) Our approach works well with both the CPU-GPU collaborative and the GPU-Pure execution models. But the performance gap between utilizing this two execution models here is not too obvious. Based on researches [8] on the *dynamic parallelism*. The *Clustering* problems with high data dependency during the iterations can benefit much from the GPU-pure execution model as the large size of data can be directly used inside the execution in GPU to avoid the cost of copy all data back to CPU memory. The observation in our approach is because the size of data required to cluster for launching a new kernel is small. But GPU-Pure execution model makes the algorithm development more flexible. e.g., in our approach, we can use the data in global memory directly for the resource reallocation to launch a new kernel (child kernel) instead of copying some data back to CPU to calculate. And with dynamic parallelism, we can allocate resources to launch a new kernel based on runtime results at any threads, without terminating all threads to return the control to CPU. In addition, based on [14], with dynamic parallelism, *nested parallel* problem can also be handled completely in GPU and the solution for recursive problems with dynamic parallelism shows the complexity simplify. 4) There is a limitation in the approach. Although we support large number of concurrent components, we haven't support the event synchronization with more than 32 components, which is due to that our synchronization occurs inside a warp (32 threads). This can be solved with our current work by integrating a sort operation for each thread to sort all synchronized events. The rest process can be similar to our existing algorithm. We plan to expand our approach with this in future work.

VI. Conclusion and Future Work

In this paper, we propose an approach to accelerating the reachability verification with support to LTS Models with different features in GPU. We propose the compact encoding that supports global variables, efficient hierarchical hash structure and parallel state space generation with collaborative synchronization, which can be generally used in other state space exploration problems. Our approach supports system models with global variables and large number of concurrent components. The experiments have shown that the design of GPURC significantly enhance the performance of dealing with the on-the-fly reachability verification problem. Meanwhile, our approach is flexible and scalable according to the evaluation results. In our future work, we plan to expand our approach by integrating some delay detection approaches, e.g., delay event synchronization, which is to delay the synchronization so as to reduce the number of states being explored temporarily. And it has potential to help improve the performance of reachability verification in our approach.

Acknowledgement This work is supported by project under Grant No.M4011178, NNSFC project Grant No.61373033 and SZSTI project Grant No.JCYJ201418193546117.

References

[1] D. Alcantara, S. Andrei, A. Fatemeh, S. Shubhabrata, M. Michael, J. Owens, and A. Nina. Real-time parallel hashing on the GPU. *TOG*, 28(5):154, 2009.

[2] D. Alcantara, V. Vasily, S. Shubhabrata, M. Michael, J. Owens, and A. Nina. Building an efficient hash table on the GPU. *GPU Computing Gems*, 2:39–53, 2011.

[3] W. Anton and B. Dragan. Improving gpu sparse matrix-vector multiplication for probabilistic model checking. In *Model Checking Software*, pages 98–116. 2012.

[4] W. Anton and B. Dragan. GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In *TACAS*, pages 233–247. 2014.

[5] W. Anton and B. Dragan. Many-core on-the-fly model checking of safety properties using GPUs. *International Journal on Software Tools for Technology Transfer*, pages 1–17, 2015.

[6] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. 2012.

[7] N. David and S. Sartaj. Data broadcasting in SIMD computers. *TC*, 100(2):101–107, 1981.

[8] J. DiMarco and M. Taufer. performance impact of dynamic parallelism on different clustering algorithms. In *SPIE Defense, Security, and Sensing*, pages 87520E–87520E. International Society for Optics and Photonics, 2013.

[9] B. Dragan, E. Stefan, S. Damian, and W. Anton. Parallel probabilistic model checking on general purpose graphics processors. *STTT*, 13(1):21–35, 2011.

[10] S. Edelkamp and D. Sulewski. Parallel state space search on the gpu. In *SoCS*, 2009.

[11] B. Ezio, D. Richard, and S. S. A. Towards a GPGPU-parallel SPIN model checker. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 87–96. ACM, 2014.

[12] C. Jean-Michel. On-the-fly verification of linear temporal logic. In *FM*, pages 253–271. 1999.

[13] B. Jiří, B. Petr, B. Luboš, and Č. Milan. Designing fast LTL model checking algorithms for many-core GPUs. *JPDC*, 72(9):1083–1097, 2012.

[14] S. Jones. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation S*, volume 338, 2012.

[15] A. Lefohn, S. Shubhabrata, K. Joe, S. Robert, and J. Owens. Glift: Generic, efficient, random-access GPU data structures. *TOG*, 25(1):60–99, 2006.

[16] Nvidia Corporation. Cuda c programming guide 6.0. 2014.

[17] P. Rasmus and R. F. Friche. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[18] E. Stefan and S. Damian. Efficient Explicit-state Model Checking on General Purpose Graphics Processors. In *SPIN*, pages 106–123. 2010.

[19] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.

[20] J. W.Davidson and J. Sanjay. Memory access coalescing: a technique for eliminating redundant memory accesses. In *ACM SIGPLAN Notices*, volume 29, pages 186–195, 1994.

[21] Z. Wu, Y. Liu, Y. Liang, and J. Sun. GPU Accelerated Dynamic Counterexample Generation in LTL Model Checking. In *ICFEM*, pages 413–429, 2014.