H

Zhimin Wu[1] and Yang Liu[1]

[1] NTU, NTU, SG,
zwu010@e.ntu.edu.sg,
WWW home page: http://users/~iekeland/web/welcome.html
[2] Université de Paris-Sud, Laboratoire d'Analyse Numérique, Bâtiment 425,
F-91405 Orsay Cedex, France

Abstract. The zhimin ...

Keywords: cs

1 Introduction

2 Counterexample Generation in LTL Model Checking

3 GPU Architecture and CUDA Dynamic Parallelism

4 CUDA Counterexample Generation

In this chapter, we will describe our CUDA Counterexample Generation proposal. The Key point of Counterexample Generation, as we mentioned, is a BFS-Related Path Generation work. The important properties of a parallel BFS contain the task queue management and the load balance during the search level by level. We take all these into consideration in our design. Based on the CUDA Dynamic Parallelism and GPU programming model, firstly, we propose the overall design of CUDA Counterexample Generation Algorithm. Then we explain in detail about the Two-Level Queue Management and Two-level Task Schedule in our design. Finally we propose the Path Record during the search of counterexample.

4.1 CUDA Dynamic Path Generation Algorithm

The original Counterexample Generation algorithm is shown in **??**. To design the CUDA Counterexample Generation proposal, we should focus on the key point, which means CUDA accelerated BFS-related algorithm in GPU. In normal CUDA program, host will call the kernel to run in GPU with static grid and block structure. And in the previous research CUDA IIIT-BFS [], it need to launch the kernel each time when a level of the graph is explored, which is costly and slower than CPU-BFS. To deal with this problem, research CUDA UIUC-BFS [] propose a hierarchical memory management proposal. It build three level

queue for BFS to avoid consequently launch kernel, which result in a certain speedup. But it is still a static method that can not adjust according to the task size; and there is no load balance plan in it. In addition, the above research does not focus on model checking.

There is also some research on BFS-Related CUDA accelerated model checking algorithms.(to be continued...)

To deal with these problems. Our solution to CUDA Counterexample Generation is to utilizes new features of CUDA, new GPU architecture and refers to some previous research to do our own expanding. Then we got a new CUDA Path Generation Algorithm to replace the XXX**??** in original Counterexample Generation algorithm **??**. Totally Speaking, the feature we focus on is he amount of tasks during the execution of BFS is dynamically changing. Our algorithm contains four parts:

- We utilize the CUDA Dynamic Parallelism to dynamically fork Child threads to fit the changing of tasks. We build a dynamic Parent-Child relationship.
- We build a Three-Level Queue Management proposal to fit the dynamic parallelism and dynamic task expanding.The three-level is made up of two physical level and a virtual level.
- We build a Two-level Dynamic Task Schedule to fit the dynamic Parent-Child relationship control and load balance.
- We utilize the hierarchical Memory in GPU to do the Two-Level Path Recording and duplicate elimination to some extent.

We present the design of our algorithm in Algorithm 1, Algorithm 2 and Algorithm 3: Algorithm 1 propose the *CudaParentPathGeneration* Algorithm. It is the kernel launched by host.It contains the initial steps of Path Generation and will finish a certain amount of BFS-related tasks. It major focus on the Parent-level task schedule, Queue Management, fork Child-threads and distributed tasks to them. It controls the Parent-Child Relationship during the execution. Line X to Line X is

Algorithm 2 propose the *CudaChildPathGeneration* Algorithm. It is the kernel launched by parent thread, regarded as Child kernel. It will finish the major tasks of Path Generation. It contains the Child-level Queue Management, Child-level task schedule and maintain the Parent-Child Relationship during the execution. Line X

Algorithm 3 propose the *CudaQuicksort*,*Scc&AccReach* and *CudaBlocksSyn* Algorithm. *CudaQuicksort* algorithm utilize the Dynamic Parallelism feature of CUDA to do quick sort for SCC node list.This refer to **??**. *Scc&AccReach* use the Binary search to find if the path generation execution has reached SCC. *CudaInterBlocksSyn* Algorithm refers to the algorithm mentioned in **??**. It choose the *simple_blocks_synchronize* when the number of blocks is small. And the *tree_blocks_synchronize* is for large number of blocks.

From the overall introduction of algorithms, the structure of our solution to the CUDA counterexample generation is clearly. Totally speaking, we utilize the dynamic parallelism technique from latest CUDA. But the key point is that we

finish a complete scheme to make our algorithm available to utilize this support from cuda, as well as fit the new GK110 architecture. Then we reach the ultimate goal of working out a novel CUDA counterexample generation scheme. In the following parts, more detail will be introduced.

4.2 Dynamic Parent-Child Relationship

Data-dependent parallel work can be generated inline with CUDA Dynamic Parallelism**??**. So building the relationship between *Parent* and *Child* is a data-driven work. In our algorithm design, in order to maximize the utilization of parallelization, each thread is asked to just do the breadth search and path recording of one node. It means the number of nodes in each level of BFS will decide the needed threads number. Therefore, with the node number uncertainty in BFS-related Path Generation, the best way is to allocate threads during runtime. That is what our algorithms do. And in order to save the cost, our *Parent*, which is launched by host, should not be a large scale grid. The major tasks of path generation should be done by *Child* and the *Parent* focus on the initial steps of path generation and the *Parent-Child* relation management. In our algorithm, *Parent* will always be just one *block*. As the minimum source schedule unit in GPU is *warp*, the GK110 can support 4 *warps* running at the same time, we always choose $4 - warps \times \alpha$ as the threads number of *Parent* grid.

We show the relationship in 1, The *Parent-Child* relationship is not static. when *Parent* launch *Child*, it will base on the current tasks to decide the structure of *Child*, the future tasks can not be forecasted but it is certainly that *Parent* should not set a very huge *Child* grid to prevent task overload. So in most case, *Parent-Child* relationship is dynamically adjusted. In our algorithm, when the total tasks overload in *Child*,it should return to *Parent* for reallocating(shown in **??**)). However, launch *Child* for a lot of time is also costly. so we should make a compromise.It will be mentioned in the Dynamic Two-level Task Schedule part.

Beside the total tasks overload in *Child*, when the *generated path* in any thread in *Child* reach the SCCACC ($SCCnodelist$ $Accnodelist \cup Nodelist_i n_p ath \neq \emptyset$), all threads in *Child* should synchronize and send result to *Parent*. As we mentioned in section XXX, The only way of communication between *Parent* and *Child*, as well as among *Childblocks* is on *Global Memory*, so the thread that find SCC-reach will record the generated path and a mark in *Global Memory* then synchronise among all Child blocks. As a result, *Parent* will got the mark and result and exit the execution.

In total, The *Parent-Child* relationship is dynamic according to the task overload and SCC Reach. This is very flexible and can fully utilize the dynamic parallelism.It makes our CUDA Path Generation a on-the-fly algorithm.

4.3 Dynamic Three-level Queue Management

As we mentioned in 3, the architecture of Memory in GPU contains Global memory and Shared Memory. Global memory can be readwrite by all blocks running

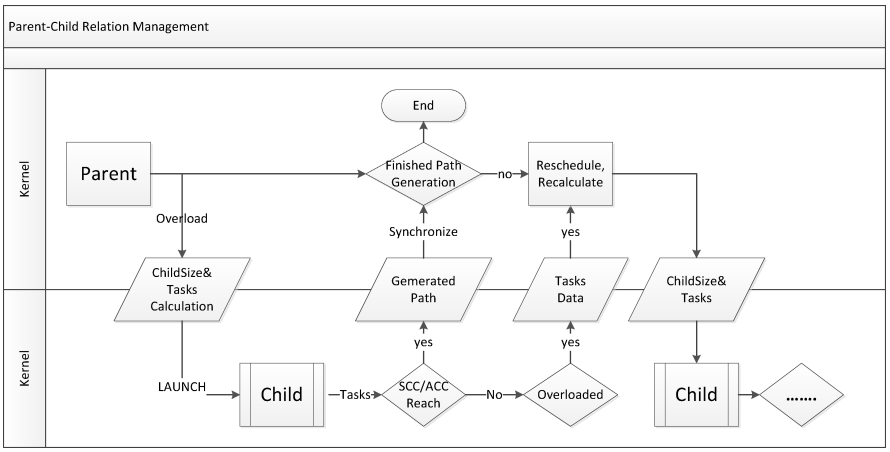

Fig. 1. Parent-Child Relationship

in all SM and Shared Memory is just available to blocks in same SM. ReadWrite operation in Shared Memory cost much less than operation in Global Memory. We should utilize this feature to improve the performance of our algorithm. But the size of shared memory is much lower than Global Memory. For our algorithm that refer to huge data size, we can not get rid of visiting Global memory. Considering the point in dynamic parallelism that only when tasks overload then call Child, 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 scheme. The structure can be shown in 2

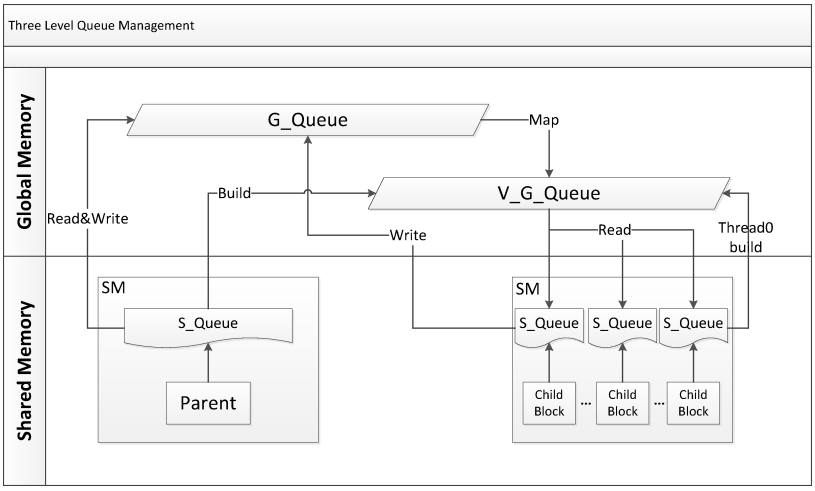

Fig. 2. Dynamic Three-level Queue Structure

The first level queue is stored in Shared Memory, marked as *S_Queue*, It is shared among blocks in 1 SM. The second level queue is stored in Global Memory, marked as *G_Queue*. The third level queue is also stored in Global Memory, called Virtual Global Queue, marked as *V_G_Queue*.

S_Queue is the first task storage queue. When the BFS-related path generation begin, the expanded node ID will firstly be pushed into the *S_Queue* in corresponding SM. Then for the next level expanding, threads do not need to visit Global Memory, it improve the data access speed. Here, the problem is it may cause readwrite conflict when parallel threads write/read at the same

time. Then if we set lock or use atomic operation to prevent conflict, it will cost much with frequently write request at the same time. So based on the GPU SM architecture and new Warp schedule in $KeplerGK110$,we build the lock-free *S_Queue* in structure showed in 3. As mentioned, the kernel is actually executed in groups of 32 threads,called a *warp*. In $KeplerGK110$, it can support 4 *warps* executing at the same time in reality. So for each block, we make the *S_Queue* a four queue-group with 32 queues each. So even the threads executed in a SM is much more than 4 *warps*, the number to write into queue at the same time is limited. *S_Queue* is the queue directly accessed by threads during execution. As one thread only hold 1 task in each level search, if the tasks in *S_Queue* exceed the number of threads executed in one block, the tasks should be re-scheduled and then it need to be transferred to *G_Queue* in Global Memory.

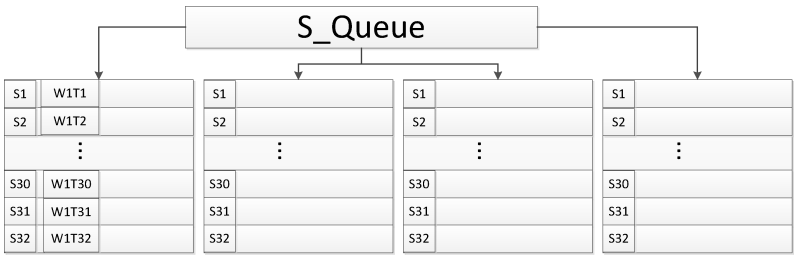

Fig. 3. Structure of S_Queue

G_Queue is build at the first time *Parent* call *Child*, it is a group of array. As dynamic allocate memeory and visit global memory is costly, this queue won't change after being build. As Global Memory is the way *Parent* communicate with *Child*,it work as the way to transfer tasks to *Child* when the first time launching *Child*. Then in following execution, it stores the tasks when blocks overload and copy the content in *S_Queue* to it. It is only visible for writing

and *Child* threads will never directly read from it. The way to write *G_Queue* is shown in 4.

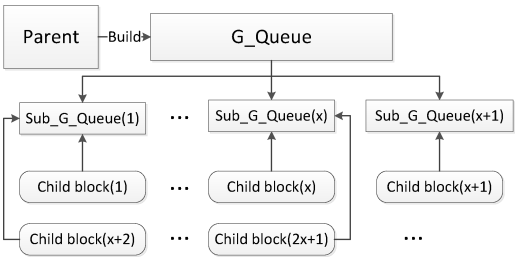

Fig. 4.

The Virtual Global Queue is the third level and it is a pointer array. As shown in **??**, in global view, the tasks store in the *G_Queue* is not continuous. So it is not convenient for tasks reschedule as it will request a lot of computing on the task range for each thread. Then *V_G_Queue* is to build a continuous list for task schedule. It is only visible for reading by threads. And only *V_G_Queue* is dynamic built during execution. It can be shown in 5.

This three-level Queue follows the rules of dynamic parallelism, aiming at building a flexible way of data access and improving the performance. It can work well with the *Parent-Child* structure.

4.4 Dynamic Two-level Task Schedule

In previous sections, what we mentioned contains two conditions that program need to do task reschedule:

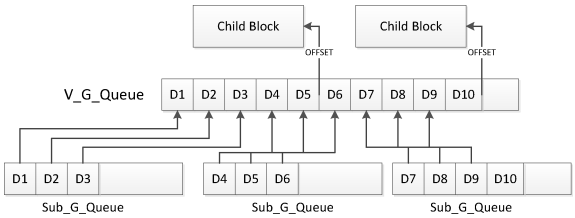
Child Block
Child Block
OFFSET
OFFSET
V_G_Queue
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
Sub_G_Queue
Sub_G_Queue
Sub_G_Queue

Fig. 5.

– when *Parent* finish some initial steps of BFS-related path generation and can not hold more tasks, it need to call *Child* and schedule tasks to *Child* the first time.
– when the whole tasks make *Child* blocks overloaded, it need to return to *Parent* to rearrange the {Child grid so as to reschedule the tasks

In this two conditions, tasks schedule is done by *Parent*. And launching kernel is an expensive work. If just one or a few blocks in *Child* is overloaded, it is obviously that the execution should not return to *Parent*. Instead, inside the *Child*, it should also have a *Child level inter blocks* task schedule. That, combined with the *Parent* level task schedule, we build the two-level task schedule. It can be shown in 6.

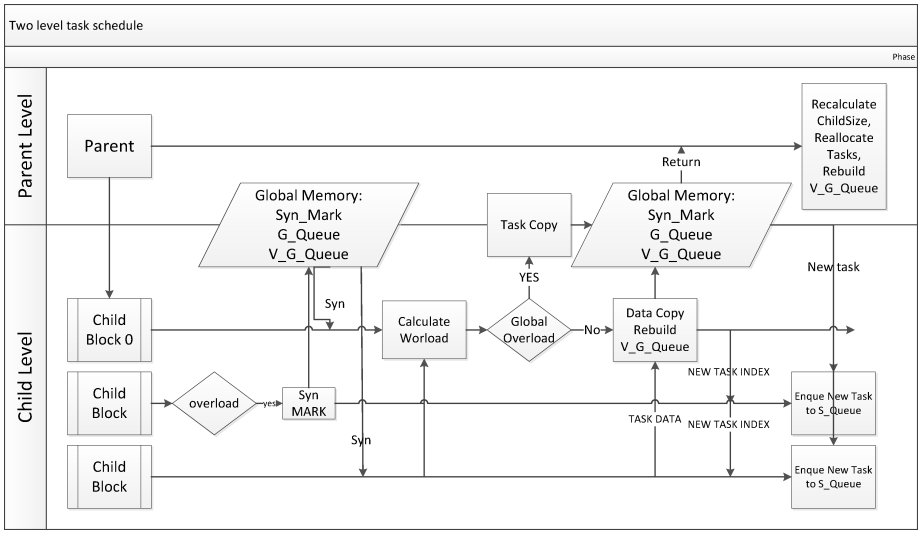

Fig. 6. Dynamic Two-level Task Schedule

In *Child* blocks, after each level of path generation, each block will decide if overloaded. Then if so, block will copy tasks in its own *S_Queue* back to *G_Queue*. Then mark *Child_syn_needed* in order to make all other blocks enter the *Child level* tasks schedule. In our algorithm, the *Child level* tasks schedule will balance the tasks among all blocks by build new *V_G_Queue*. This behavior has no reference to *Parent*. Here, in order to make each block has enough resource

to do future expanding, the schedule should not make the new tasks in each block exceed a *threshold*, Or the algorithm will conclude that the *Child* is overloaded.

Then it will come to *Parent level* task schedule. *Parent* will calculate new *Childsize-Child blocks number*, build new *V_G_Queue* and calculate tasks offset for each *Child block*. Here it also comes to the problem that if *Childsize* is not appropriate, then the *Child* will return to *Parent* frequently, which is costly. To make a compromise, in operation line X in our algorithm, when calculating the *Childsize*, we guarantee that tasks allocated to each Child should not beyond the number $warpsize = 32$ and the threads number in each *Child block* should be $warpsize \times \alpha$.

4.5 Two-level Path Recording and Duplicate Elimination

Our algorithm is to deal with the Path Generation so as to get Counterexample. So path recording is necessary. Since the path generation is a BFS-related work, the path record is updated in each search level. Take 7 as an example, the nodes in each level will inherit their precusor's path and then add themselves in the path. When a node has more than one precursor, it will inherit more than one path. However, our algorithm is to find a path to reach $SCCACCnodelist$, only to inherit one path is enough and this will save the storage space. Then we build the Path Recording scheme according to this and also the dynamic parallelism feature inclued in all other parts.

We call our scheme *Two-level Path Recording*. One is Shared Memory level path recording and the other is Global Memory level. Firstly, a simple structure is defined:

$$PathRecordNodenodeid, pathqueue$$

Then we define a *PathRecordNodeList* array in each block that stored in Shared Memory (called *S_PRN_List*) and one more array in Global Memory (called *G_PRN_List*. On a whole, during the path generation, threads will update the path in corresponding node in Shared Memory following the inherit rules mentioned. And the updating of path in Global Memory happens when the overload happens in order to fit the dynamic tasks schedule design previously mentioned. In details, the steps to do the path recording are as follows, all these steps are executed after the *IfReachSCC&ACC* detection of current visiting node:

- when a node is being visited, update the corresponding *PathRecordNode* in the *S_PRN_List* by push itself to the path
- when a node is being visited, copy its updated path to each of its successor's corresponding path queue If it is empty. If not, it means the successor has inherit a path and there is no need to inherit another path. Here, it also means the successor has been put into task queue and it can eliminate the duplication inside a block to some extent

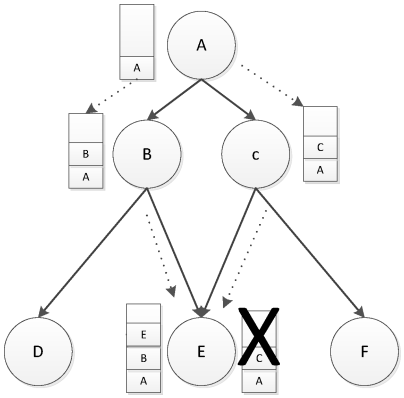

Fig. 7.

- when overload happen, we do the path copy together with copy tasks from *S_Queue* to *G_Queue*. Only the path data that with ID contained in *S_Queue* are copied. Here, with the same rule, if the corresponding *PathRecordNode* has unempty path queue, the copy operation will be aborted and this node ID will also be aborted in task copy.
- these steps continue until *IfReachSCC&ACC* detection return TRUE.

Here, the reason we say it can just eliminate duplication to some extent is that after once $S\_PRN\_List \Rightarrow G\_PRN\_List$ copy, All path queue in S_PRN_List will be empty. So in following steps, the visited node may have possibility to be visited again and it will store the path again. But when copy back to *G_PRN_Queue*, this will be detected.

Another problem is the storage space problem. (need expanding, to be continued...)

In addition, when a thread updating the path queue in *PathRecordNode*, atomic operation is needed to prevent conflict. And it should not block other threads. This will be discussed in the following part.

4.6 Synchronization and Atomic Operation

In our algorithm, synchronization contains two parts: intra block synchronization and inter block synchronization. For all threads inside a block, the intra block synchronization should be taken in each level expanding as the algorithm need to judge if overload happen. This intra block synchronization has been supported by CUDA with $CudaThreadSync()$. What we should focus is the inter block synchronization. It requires all blocks to attend, So a low-cost way to do this is necessary. We chose the inter block synchronization method in [] that has been proved to be high efficiency. the data that *Child* blocks need to communicate with each other contains *IfSCCAccReach, If overload and If_return2Parent*. These are stored in Global Memory and will be checked by each thread in each level expanding. If the corresponding event happens being detected by any thread, it will utilize atomic operation to update the data in Global Memory and then called the *CudaInterBlocksSyn*. After other blocks detect the update, they will also call *CudaInterBlocksSyn* so as to synchronise among all blocks.

Here we mention the atomic operation. It is used to build a unblock lock. When some threads want to write the same memory address at the same time, only the first one call the lock will get the access right and the other will abort their write operation and continue their executing. The design of this lock is shown in 4

With all these above, we build the complete CUDA Path Generation algorithm for Counterexample Generation. All content make full use of the dynamic parallelism and the feature of new Kepler $GK110$ architecture.

Algorithm 1 CudaParentPathGeneration Algorithm

Input: $Start\_ID, SCCnodelist/ACCnodelist, OutgoingRelation$

$Define\_shared\_S\_queue, Queuesize, Ifexpanded, IfSCCAccReach$;
if $inblockthreadindex = 0$ **then**
  $Three - levelQM : S\_queue.enqueue(Start\_ID)$;
  $Path - Recording : path\_record\_queueinsideStartNode$;
end if
$Intra\_block_syn()$;
while $Ifexpanded \neq TRUE$ **do**
  if $inblockthreadindex < Queuesize$ **then**
    $Three - levelQM : S\_queue.ParallelDequeue()$;
    $CudaScc\&AccReach : SCC\&AccReachdetection$;
    $BFS, OutgoingRelation \Rightarrow NEXTlevel$;
    $PathRecording : Inherit\&Update(path\_record\_queueofeachNode)$;
    $ThreelevelQM : S\_queue.ParallelEnqueue()$;
  end if
  if $inblockthreadindex = 0$ **then**
    $Queuesize \Leftarrow S_queue.size$;
    if $Queuesize > THRESHOLD$ **then**
      $Ifexpanded := TRUE$;
    end if
  end if
end while
$Intra\_block_syn()$;
if $inblockthreadindex = 0$ **then**
  $Preparefor parallelismexpanding :$
  $PathRecording : copybacktoGlobalmemory$
  $Two - levelTS : Parent - levelInitialTaskSchedule$;
  $Three - LevelQM : Build\_device\_G_queue$;
  $Update(Expanding\_ChildSize), Initial(\_device\_G\_IfSCC\&AccReach)$;
end if
$Intra\_block_syn()$;
while $!IfSCC\&AccReachAND!G_IfSCC\&AccReach$ **do**
  if $inblockthreadindex = 0$ **then**
    $Three - levelQM : Build(\_device\_V\_G\_Queue)$;
    $Two - levelTS : ParellelCal(Child\_Tasks\_Offset\_V\_G\_Queue)$;
  end if
  if $inblockthreadindex = 0$ **then**
    $ChildKernel <<< >>> (V\_G\_Queue, Child\_Tasks\_Offset, ...)$;
    $CudaDeviceSyn() : IfChildsNOTfittasksORfinished, returntoParent$;
    $ThenContinueloop : Two - levelTS : ParentRescheduletasks, Childre - expanding$;
  end if
end while
$Intra\_block\_syn()$;

Algorithm 2 CudaChildPathGeneration Algorithm

Input: $\_device\_V\_G\_Queue, Child\_Tasks\_offset,$
$SCCnodelistAccNodelist, OutgoingRelation$
$Pre-define\_device\_(Child\_Expandedtask, Child\_syn\_need,$
$Child\_need\_back2parent, G\_IfSCC\&AccReach)$
$Define\_shared\_(Child\_S\_Queue, queuesize, IfSCC\&Accreach);$
$Getinblockthreadindex, globalthreadindex;$
if $inblockthreadindex = 0$ **then**
  $Initialize(queensize, IfSCC\&Accreach);$
end if
if $globalthreadindex = 0$ **then**
  $Initialzie(all\_device\_variables);$
end if
$CudaInterBlockSyn();$
while $!G_IfSCC\&AccReachAND!Child\_need\_back2parent$ **do**
  $BlockTasks := Tasksoffsets[blockindex + 1]Taskoffsets[blockindex];$
  $Three-levelQM : Enqueue(GetthreadtaskFROMV\_G\_Queue);$
  $Queuesize \Leftarrow blocktasks.num;$
  $Intra\_block\_syn();$
  if $inblockthreadindex < queuesize$ **then**
    $Three-levelQM : Child\_S_Queue.ParallelDequeue();$
    $CudaScc\&AccReach : SCC\&AccReachdetection;$
    if $ReachSCC$ **then**
      $G\_IfSCC\&AccReach := TRUE, RecordPath;$
      $BREAK;$
    end if
    $BFS, OutgoingRelation \Rightarrow NEXTlevel;$
    $Path-Recording : Inherit\&Update(path\_record\_queueofeachNode);$
    $IFDuplicate \Leftarrow Path-Recording();$
    if $IFDuplicate = TRUE$ **then**
      $CONTINUE;$
      $Three-levelQM : S\_queue.ParallelEnqueue();$
    end if
    $Intra\_block\_syn();$
    if $inblockthreadindex = 0$ **then**
      $queuesize \Leftarrow Child\_S\_Queue.size;$
      if $queuesize > blockThreshold)$ **then**
        $Child\_syn\_neede := TRUE;$
      end if
    end if
    $CudaInterBlocksSyn();$
    if $child\_syn\_needed = TRUE$ **then**
      $Three-levelQM : Parallel\_Update(G\_Queue);$
      if $globalthreadindex = 0$ **then**
        $Build(V\_G\_Queue);$
        $Two-levelTS : Inter\_Childblocks\_Tasks\_Reschedule;$
        if $Whole\_blocks\_overloaded = TRUE$ **then**
          $Child\_return2Parent := TRUE;$
        end if
      end if
      $CudaInterblocksSyn();$
    end if

    $CudaInterblocksSyn();$

Algorithm 3 CudaQuicksort, Scc&AccReach and CudaInterBlocksSyn Algorithm

1: $CudaQuickSort(data, pivot, left, right)$
2: $Definenleft, nright$;
3: $Partition(data + left, data + right, pivot, nleft, nright)$;
4: **if** $left < nright$ **then** $CreatCudastream(s1)$; $CudaQuickSort$ <<< $...s1$ >>> $(data, left, nright)$;
5: **end if**
6: **if** $nleft < right$ **then** $CreateCudasteam(s2)$; $CudaQuickSort$ <<< $...s2$ >>> $(data, nleft, right)$;
7: **end if**
8:
9: $Scc\&AccReach()$
10: $NormalBinarySearch$
11:
12: $CudaInterBlocksSyn()$
13: **if** $blocks.num < THRESHOLD$ **then** $simple\_blocks\_synchronize()$;
14: **else**$tree\_blocks\_synchronize()$;
15: **end if**=0

Algorithm 4 Unblock lock with atomic operation

1: **while** $Iffinished = TRUE$ **do**
2: **if** $Atomic(mutex, 1)$ **then**
3: write operation;
4: $Iffinished := True$;
5: $Atomic(mutex, 0)$;
6: **else**
7: $abort = TRUE$;
8: $break$;
9: **end if**
10: **end while**