Model checking using multiple GPUs

Qi, Y.

Award date:
2018

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Model checking using multiple GPUs

Master Thesis

Yuyang Qi

Supervisor:
dr. ing. Anton Wijs

Committee members:
dr. ir. Tim Willemse
dr. Andrei Jalba

Eindhoven, November 2018

Abstract

Model checking using GPUs has seen increased popularity over the last years. Many model checkers try to increase their performance by deploying partial or complete exploration algorithms on GPUs. It has been proved that model checking using GPUs can lead to significant speed-ups. However, because GPUs have only a limited amount of memory, only small to medium-sized systems can be verified. Hence, the main problem of GPU model checkers are spacial rather than temporal.

In the thesis, we fixed two bugs in the tool GPUexplore and extended it so that it can efficiently runs on two GPUs installed in one machine. Benchmarks show that in average GPUexplore running on two GPUs can store 67% more states with similar or shorter runtime compared to GPUexplore running on one GPU. For larger models, GPUexplore running on two GPUs can achieve at most 1.7 speed-ups. When the tool runs on the state-of-the-art NVIDIA GPUs (2x GTX Titan Xp), it can construction a state-space at 17 million state/second speed.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overview

In software and hardware design of complex systems, more time and efforts are spent on verification than on construction [10]. The purpose of verification is to identify system defects at early stage of the development process, which prevents catastrophic outcomes due to the defects. One of the verification techniques is model checking [10]. Model checking is a technique that checks whether a finite-state system holds a formal property. It explores all possible system states and checks the validity of the given formal properties on-the-fly or after exploration. As the systems become more complex, the number of states of a system grows accordingly. Hence, for large systems, model checking is computationally intensive and requires significant amount of memory to store states, which often can be prohibitive in practical applications [3].

In order to obtain more computation power, multi-core and many-core architectures are popular in fields that require complex computations like machine learning and artificial intelligence. In model checking, many popular model checkers are provided with multi-core implementations like LTSmin [1], SPIN [18] and DiVinE [6]. Apart from multi-core architectures, there are some model checkers (see Section 1.3.2) take the advantage of many-core architectures. Many-core architectures can mainly be found in Graphic Processing Units (GPUs). Although GPUs have been aimed at rendering graphics when they were developed, now they are also suitable for general tasks. This is called general-purpose GPU (GPGPU) programming. One of the most popular Application Programming Interfaces (API) for GPGPU is Compute Unified Device Architecture (CUDA)[1]. A number of applications were developed using CUDA including model checking applications such as [19][24][25].

1.2 Problem statement

Despite GPU model checking having achieved significant speed-up for a GPU based model checker GPUexplore [4], it is limited by the memory of the GPU. The amount of memory of GPUs is often smaller than the main memory. As a result, traditional model checkers using main memory can often check larger models. The main reason for significant memory usage is state-space explosion [13]. The explosion is due to many possible interleaving of actions of concurrent processes. The memory usage of a model checker grows exponentially as the system grows linearly. One way to scale up GPU model checking is by using multiple GPUs to store the state-space.

The goal of the thesis is to increase the size of the state-space that GPUexplore can store by using multiple GPUs.

[1] `https://developer.nvidia.com/about-cuda`

By using multiple GPUs we can check larger models because more GPU memory is available. The thesis is to provide multi-GPU support for a specific GPU based model checker - GPUexplore [3]. The scope of the project is to use GPUs installed in a single machine. Support for GPUs installed in multiple machines is scheduled for future work.

1.3 Related work

There are three popular parallel explicit state model checking tools that runs on multi-core systems, namely LTSmin, SPIN and DiVinE. And there is a explicit state model checking tool that completely runs on a GPU which is GPUexplore [2].

1.3.1 Multi-core model checking

Laarman et al. [1] provided LTSmin [22] with a multi-core solution for checking safety properties to increase performance. The implementation uses `pthreads` library and works on shared-memory multi-core architectures. The implementation provides several new contributions in workload balancing, state hashing and state compression. Their state hashing technique allows DVE2 (from DiVinE) to explore state space at 10 million state/sec with 16 cores.

Holzmann [18] extended SPIN [17] with multi-core support on shared-memory architectures. The parallelization mainly focuses on the verification processes for safety properties which is based on a breadth-first search algorithm. He also applies Partial Order Reduction (POR) technique to reduce state-space size. The implementations has shown that the performance of the new parallel breadth-first search algorithm scales reasonably well with increasing numbers of CPU cores.

Barnat et al. [6] extended DiVinE tool [7] with multi-core and multi-cpu support for shared-memory architectures. The extension is based on distributed-memory exploration algorithms reimplemented specifically for multi-core and multi-cpu environments using shared-memory. The parallelized DiVinE tool can reached a speed-up of 12 with 16 threads.

1.3.2 GPU model checking

GPGPU techniques have been applied in model checking in various ways.

Bartocci et al. [8] implemented a CUDA version of SPIN. The state-space generation are done by the GPU and the states are stored using cuckoo hashing described in [5]. The implementation suffers significant overhead for smaller models while performing well for medium-size models.

Wijs and Bošnački developed a on the fly explicit state model checker that completely runs on a GPU using CUDA [3]. The tool is called GPUexplore. It can generate state-space, check for the absence of deadlocks and check safety properties. The performance of GPUexplore is similar to LTSmin running on about 10 threads. Then Wijs et al. [4] improved the performance of GPUexplore with enhanced lock-less hashing of the states and improved thread synchronizations. The newer version of GPUexplore running on state-of-the-art hardware (GTX Titan X) can be more than 100 times faster than a sequential implementation for large models. After that, Neele et al. [23] provided GPUexplore with POR support. The implementation can achieve a reduction similar to or better than POR on a multi-core platform.

Wu et al [25] extended the PAT model checker with a CUDA implementation of counter-example generation. When the CPU generates the state-space and reports an error state, then the GPU explores the state-space to find the smallest path to that error state. The implementation applies dynamic parallelism to balance the workload for processors in the GPU. There are no performance comparison with other works in the paper.

wu et al [24] also implemented a model checker that completely running on GPU using CUDA. They adopted several techniques from GPUexplore and added dynamic parallelism mechanism for state-space generation. The performance gain from the dynamic parallelism is minimal. The implementation shows good speed-up compared with sequential state-space exploration but no comparisons to other model checking tools are conducted.

1.4 Research questions

The goal of the project is to solve state-space explosion using multiple GPUs for GPUexplore (It was named GPUexplore 2.0 in [4]). We formulate the main research question as the following:

How to extend GPUexplore to use multiple GPUs so that it can construct larger state-space while maintaining or achieving better performance?

In order to analyse the main research question in a step-by-step manner, we formulated the following sub questions:

RQ1. What modifications to GPUexplore are needed so that it can runs on multiple GPUs?

RQ2. How much capacity increment can multi-GPU GPUexplore provide compared to GPUexplore?

RQ3. What would be the runtime difference between multi-GPU GPUexplore and GPUexplore?

RQ4. What can be done to improve the performance of multi-GPU GPUexplore?

1.5 Thesis outline

The rest of this report are structured as follows: Chapter 2 explains theories related to the development of multi-GPU GPUexplore; Chapter 3 illustrates the bugs in GPUexplore and the means to fix them; Chapter 4 gives the design and the implementation of the tool; Chapter 5 shows the optimizations for the tool; Chapter 6 provides the results of experiments of both version of the tool; Chapter 7 discusses the limitations of the tool and schedules future work; finally Chapter 8 concludes the results of the thesis.

Chapter 2

Background

This chapter first gives an introduction to the architecture of NVIDIA GPUs, then it explains several features of multi-GPU programming. After that, the basic theory of transition systems is illustrated. Finally, some methodologies of model checking are explained.

2.1 NVIDIA GPU architecture

CUDA is a parallel computing platform and an API model created by NVIDIA. It enables general purpose computing on NVIDIA GPUs and is widely used. CUDA is an extension to C/C++ and Fortran. CUDA code defines the behaviour of a single *thread*, and the code is executed by multiple threads in parallel. The threads are normally grouped into several *blocks*, and blocks are grouped into a *grid*. Threads and blocks can be grouped in a 1, 2 or 3-dimensional manner. For example, a grid of $10 * 6$ blocks, in which each block contains $8 * 8$ threads, has in total 60 blocks and $60 * (8 * 8) = 3840$ threads. Figure 2.1 shows the relation between the grid, blocks and threads.

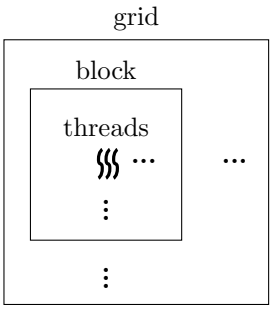

Figure 2.1: Relation between the grid, blocks and threads

From the hardware's perspective, each GPU consists of several streaming multiprocessors (SM), which contains a number of CUDA cores. For example, NVIDIA TITAN Xp has 60 SMs and each SM has 64 CUDA cores, which results in a total of 3840 CUDA cores. GPUs in the system are regarded as *devices*, each GPU has its own device number, which starts from 0 and is incremented by 1. The CPU that runs the CUDA code is regarded as the host.

Each GPU contains a global memory that has high latency and capacity, and a number of shared memories that has low latency and capacity. It can be accessed by all threads running on the GPU. However, the threads in a GPU would not access global memory directly, instead, they access it through L1 and L2 cache or texture cache. The working principle of the cache is similar to that of CPUs. Shared memory resides on SMs and has low latency and capacity. Each SM has

its own shared memory, and the memory is divided among the blocks that are assigned to this SM. Each partition of the memory assigned to a block can only be accessed by the threads in that block. Figure 2.2 shows the memory hierarchy of the GPU.

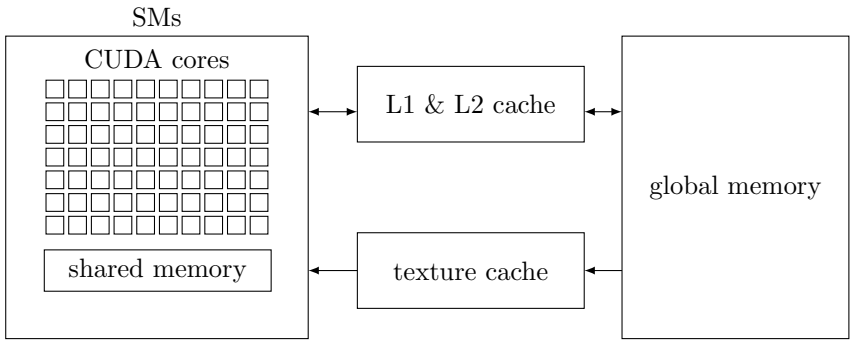

Figure 2.2: GPU memory hierarchy

From the perspective of threads execution, the grid is the parameter needed to launch a *kernel*. A kernel is a function that, when called, is executed by multiple threads in the GPU. Each block in the grid will be assigned to and executed by one SM, and one SM can be assigned with many blocks. The blocks assigned to the same SM can be executed in parallel. The number of parallel blocks depends on the model of GPU (for GTX 1080 it is between 1 and 32) and the launching configurations.

The threads in a block are executed in parallel, while the parallelization happens for each 32 threads, and each 32 threads is called a *warp*. If the threads in a warp cannot be executed in parallel (e.g., branch divergence and/or uncoalesced memory accesses), then they will be executed sequentially.

2.2 Multi-GPU programming

There are several situations where a single GPU is not appropriate for the tasks. The situations may be the lack of computation power or insufficient GPU memory. In our case, the main reason for using multi-GPU programming is to use more GPU memory so that we can check larger models. In order to take the advantage of multiple GPUs, we need to consider inter-GPU communications and the programming model.

For inter-GPU communications, there are two general cases. One is GPUs within a single network node and the other is GPUs across network nodes. GPUs within a single network node can communicate with each other through PCIe interconnect with the help of CPU; GPUs across network nodes can communicate with each other through Message Passing Interface (MPI) using CUDA-aware MPI library[1]. The scope of this thesis is using GPUs within a single network node.

Like single-GPU CUDA applications, GPUs in a multi-GPU CUDA application are also managed by a single CPU thread. We should consider the following matters when programming a multi-GPU application:

1. CUDA APIs are all issued to current GPU.
2. Current GPU can be selected using `cudaSetDevice()`.
3. All memory manipulation API calls (e.g. `cudaMemcpy()` and `cudaMalloc()`) need to manipulate memory that belongs to current GPU.

[1] `https://developer.nvidia.com/mpi-solutions-gpus`

2.2.1 Indirect GPU communication

Typically, GPUs in the same machine are connected by PCIe bus. Therefore, GPUs can communicate with each other through PCIe bus by the help of CPU. This is done by using the CUDA API `cudaMemcpy()` to copy the required data from the source GPU to the host memory via PCIe bus, and then the data in the host memory are copied to the destination GPU with the same API.

2.2.2 Unified memory

In 2013, NVIDIA brought up a new programming model named *Unified Memory* for multi-GPU programming in a single network node. This model combines GPUs memory and the host memory into one unified memory that eases the difficulty of multi-GPU programming. Unified memory, as the name implies, unifies all memories in the system into one unified memory. By doing this, it changes how developers view the memories. Figure 2.3 shows the unified memory for a system with multiple GPUs. Unified memory is a managed memory that is shared by the CPU and GPUs. It is accessible to CPU and GPUs. For Pascal and later generation GPUs running in a Linux kernel system, the unified memory is divided into pages and those pages are dynamically migrated between host or devices (GPUs) memory with the purpose of increasing access speed. Because the migration is dynamically adjusted and is invisible to the programmers, it is possible that some applications may have higher runtime because of page faults. Page faults are situations where CPU or GPUs are trying to read/write data in a unified memory that is not physically located in its memory. And this is solved by page migration, where the page containing that data will be moved to the device that requires read/write.

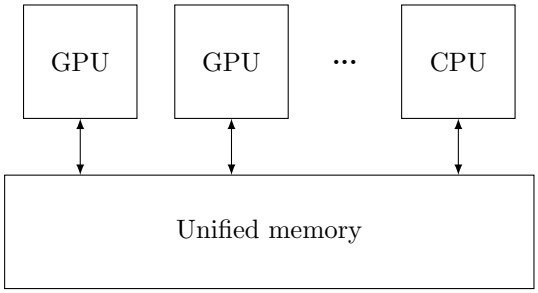

Figure 2.3: Unified Memory is a single memory address space accessible from any processor in a system.

2.2.3 Peer access

Apart from unified memory, another feature from NVIDIA that helps multi-GPU programming is peer-to-peer memory access. It allows devices of compute capability 2.0 and higher to dereference a pointer to the memory of the other GPU. In this way, one GPU can communicate with the other GPU without copying data to the host. This peer-to-peer memory access feature is supported between GPUs if `cudaDeviceCanAccessPeer()` returns true for all of them. By using peer access feature, one can launch a CUDA kernel and use pointers pointing to other GPUs memories as parameters.

2.3 Transition systems

A Labelled Transition System (LTS) is a directed graph in which the vertices represent states, and the edges represent transitions between states [3]. Each transition contains a labelled action that connects the states. Moreover, an LTS must have a initial state which is depicted graphically by a small incoming arrow. According to [20], the definition of an LTS is the following.

Definition 2.3.1 (**Labelled Transition System**). An LTS is a tuple $A = (S, Act, \longrightarrow, s)$ where

- S is the set of states.
- Act is the set of actions.
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation between states. Each transition is labelled with an action.
- $s \in S$ is the initial state.

It is common to write a transition from state q_0 to state q_1 with action a as $q_0 \xrightarrow{a} q_1$ for $(q_0, a, q_1) \in \longrightarrow$ [20]. The successors of a state q with action α is defined as $succ(q, \alpha) = \{q' | q \xrightarrow{\alpha} q'\}$. The successors of a state q with all actions in Act considered is defined as $succ(q) = \bigcup_{\alpha \in Act} succ(q, \alpha)$. We say q_n is reachable from q_0 if there exists a sequence of actions and states such that $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} ... \xrightarrow{a_n} q_n$. Finally, we can define $enabled(q) = \{\alpha | \exists q'.q \xrightarrow{\alpha} q'\}$ as the set of actions that are enabled in state q.

In order to give a proper definition of concurrent system that has finite state-space which are modelled as LTSs, we define a network of LTSs based on [14].

Definition 2.3.2 (**Vector**). For $i \in 1..n$, a vector of length n over set S is a list $\vec{t}$ where $\vec{t}[i]$ denotes the i-th element of $\vec{t}$ and $\vec{t}[i] \in S$. For example, a vector $\vec{t}$ of length 3 over set $\{0, 1\}$ can be $\vec{t}[1] = 1$, $\vec{t}[2] = 0$, $\vec{t}[3] = 1$ or in another form $\vec{t} =< 1, 0, 1 >$.

Definition 2.3.3 (**Network of LTSs**). A network of LTSs is a tuple $N = (L, V)$ where

- L is a vector of n LTS. For each $i \in 1, ..., n$, we write $L[i] = \langle S[i], Act[i], \longrightarrow [i], s[i].\rangle$
- V is a vector of synchronizations rules. A synchronization rule is a tuple $(\vec{t}, a)$ where a is an action and $\vec{t}$ is a vector of length n over set $\{0, 1\}$. In set $\{0, 1\}$, 1 represents that the corresponding process synchronizes on action a, and 0 represents the opposite. Note that $\vec{t}[i]$ can only have one element of $\{0, 1\}$.

Example 1. An example of a network of LTSs can be found in Figure 2.4. The example network N contains a list of LTSs L that has length 4. Take $L[1]$ = Producer, $L[2]$ = Observer, $L[3]$ = Consumer1 and $L[4]$ = Consumer2. After Producer generates a work (action *gen*), then the work is sent to either Consumer1 or Consumer2 via action *trans*. When Consumer1 or Consumer2 receives a work, it will process it (action *work*) and then get ready to receive next work. Observer will continuously do action *observe* and do not synchronize with other LTSs.

Given above information we can derive the list of synchronization rules $V[1] = (\vec{t_1}, gen)$, $V[2] = (\vec{t_2}, trans)$, $V[3] = (\vec{t_3}, trans)$, $V[4] = (\vec{t_4}, work)$, $V[5] = (\vec{t_5}, observe)$. Because $L[1]$, $L[3]$ and $L[4]$ synchronize on action *trans* and only one of $L[3]$ and $L[4]$ can proceed, we can derive two rules: $\vec{t_2} =< 1, 0, 1, 0 >$ and $\vec{t_3} =< 1, 0, 0, 1 >$. And because no LTSs synchronize on action *gen*, *observe* and *work*, $\vec{t_1} = \vec{t_4} = \vec{t_5} =< 0, 0, 0, 0 >$.

In the example, the order of which Consumer gets the work is unknown. Then it is necessary to consider all possibilities when constructing the state-space of this network of LTSs. When there is a significant number of LTSs that synchronizes on the same action, then the number of possible states that can be reached can be large. As a result, the size of state-space of the network can be enormous.

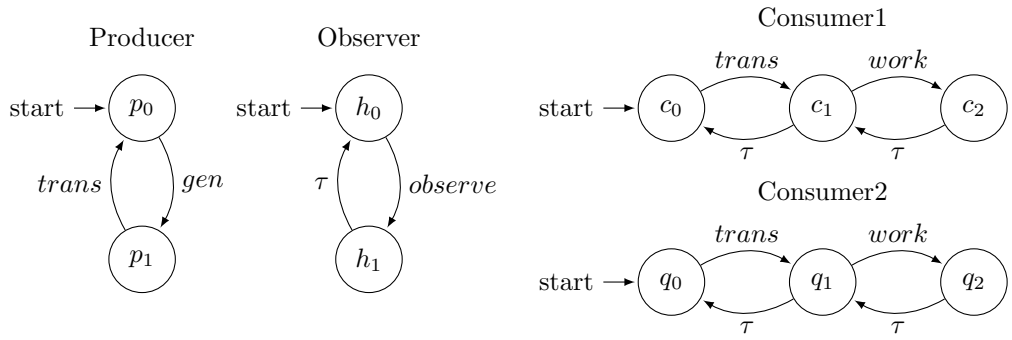

Figure 2.4: An example of a network of LTSs

2.4 Model checking

This chapter explains the state-space exploration algorithm, the distributed state-space exploration algorithm and a on-the-fly explicit-state GPU based model checker - GPUexplore.

2.4.1 State-space exploration

In order to check error states or to solve a reachability problem, we need to construct a LTS that combines the network of LTSs and synchronization rules, we call it the behaviour of the network. According to [23], the definition of it is the following.

Definition 2.4.1. Let $N = (L, V)$ be a network of LTSs. $A_N = (S, Act, \longrightarrow, s)$ is the LTS describing all potential behaviours of N where

- $S = S[1] \times ... \times S[n]$ is the cross-product of all state of the LTSs in the network.
- $Act = Act[1] \cup ... \cup Act[n]$ is the union of all action sets in the network.
- $\longrightarrow = \{(\langle q_1, ..., q_n \rangle, a, \langle q'_1, ..., q'_n \rangle) | \exists_{(\vec{t},a)\in V} \forall_{i\in\{1,...,n\}} (\vec{t}[i] = 1 \implies q_i \xrightarrow{a} q'_i \in \longrightarrow [i] \wedge \vec{t}[i] = 0 \implies q_i = q'_i)\}$ is the set of transition relations that follows from each of the LTSs and the synchronization rules.
- $s = s[0] \cup ... \cup s[n]$ is the union of initial states of all LTSs in the network.

The state-space exploration algorithm is shown in Algorithm 1. The Data required for the algorithm is two sets of states. Set *Open* includes states that still need exploration and *Closed* contains states that have been/are being explored. On Line 2 and Line 3 state q is moved from *Open* to *Closed* for exploration. Then on Line 4 all successors of q are generated and those states which are not explored are added to *Open* Line 6). The algorithm terminates when $Open = \emptyset$.

Algorithm 1: State-space exploration algorithm

```
   Data: Open ← {s}, Closed ← ∅
1  while Open ≠ ∅ do
2      Open ← Open\{q} for some q ∈ Open;
3      Closed ← Closed ∪ {q};
4      foreach q' ∈ succ(q) do
5          if q' ∉ Closed then
6              Open ← Open ∪ {q'};
```

2.4.2 Distributed state-space exploration

Since there is a growing number of people using clusters of machines to do research, it is reasonable to improve the state-space algorithm specifically for clusters of machines so that the exploration can be done faster and more memory can be used. A distributed state-space exploration algorithm is introduced in [9]. A simplified version of it is proposed in this report. The algorithm divides the state-space into distinct parts and uses multiple machines to explore its corresponding parts concurrently. Assume there is a cluster of N machines communicating by message passing. The method to distribute states to different machines is using a hash function $h : S \to \{0, ..., N-1\}$. Then, $h(s)$ will be the owner of state s. The responsible part of the state-space for process i is defined as $S_i = \{s | h(s) = i\}$ for $i \in \{0, ..., N-1\}$.

The algorithm uses two types of processes: A **coordinator** and N **worker** processes. The coordinator, a light process that can be executed on any worker, is responsible for sending initial states to workers and terminating all workers when the exploration is complete. The workers are the processes that explore the state-space, each worker is executed on one machine and all workers execute the same code. The task of a worker i is to generate the state-space that belongs to it, which is S_i. The algorithm is described in Algorithm 2.

Algorithm 2: Distributed state-space exploration (`dist(`i`)`)

```
   Data: Network of LTSs N
1  while Waiting ≠ ∅ do
2      Waiting ← Waiting\{s} for some s ∈ Waiting;
3      S_i ← S_i ∪ {s};
4      foreach s' ∈ succ(s) do
5          if h(s') = i then
6              S_i ← S_i ∪ {s'}; // New state s' belongs to current worker
7          else
8              send(h(s'), s'); // New state s' belongs to other worker
```

In the algorithm, $Waiting$ contains the states that need exploration, it will be fill by other workers via `messHandler()` and the coordinate with initial states. At each iteration of the while-loop, a state s will be removed from $Waiting$ and stored in S_i, and then generate its successors. And if the successors of state s belong to the current worker, they will be stored in S_i; if $h(s)$ suggests otherwise, they will be sent to other workers via `send()`.

Upon receiving a state, the worker process will be preempted[2] and a handler function `messHandler()` will be executed. The excerpt of this function is described in Algorithm 3 which indicates the operations executed when a state is received by worker i.

Algorithm 3: Message handler (`messHandler(`i`)`)

```
1  ...
2  case Message.Type = STATE do
3      s ← Message.state;
4      Waiting ← Waiting ∪ {s};
5      dist(i);
6  ...
```

[2]Preempted/preemtion means the current task is paused with the intention of resuming the task at a later time

Algorithm 2 is executed repeatedly until a `TERMINATE` message is received from the coordinator process. The termination occurs when all workers have finished handling theirs states and there is no state in transition.

2.4.3 Existing design - GPUexplore

According to [3], GPUexplore is a GPU based explicit state model checker that checks deadlocks and safety properties on-the-fly for models expressed in LTSs defined in Definition 2.3.1. GPUexplore constructs the synchronous product of the LTSs in the network using many threads in a Breadth-First-Search-based exploration based on Algorithm 1. All computations of the model checker are preformed on the GPU.

The input network is stored in the texture cache with the purpose of increasing access speed. GPUexplore implements *Open* and *Closed* as a single hash table located in GPU's global memory. All states are encoded as *state vectors* in the hash table and there is 1 bit per state to indicate whether the state has been explored or not. The hash table is divided into buckets of 32 integers (for coalesced memory accesses), and each of them may contain multiple state vectors. State vectors are concatenations of encodings of process LTS states. Each state vector may contains n states, which are assigned to a group of n threads for parallel successor generation (For detailed successor generation we refer to [3]). Before the threads work on state vectors, the vectors are copied from the global memory to *workTile* which is located in the shared memory. Each block has its own *workTile*. The n threads are grouped into a *vector group*, and the size of the group never exceeds warp boundaries unless $n > 32$. For each vector s, threads (with identifier *vgtid*) from the assigned vector group (*vgid*) are responsible for the successor generation. The generated states (except duplicated states) are stored in the *cache* which is located in shared memory. And if a state is identified as new, then it will be copied to the corresponding bucket according to the used hash function in global memory and be marked as *old* in the cache for duplication detection.

A high-level view of the algorithm is shown in Algorithm 4. The variables used in this algorithm are *table* (used by `findOrPutWarp()`), *cache* (used by `storeInCache()`) and *workTile* (to store the results from `gatherWork()`). *table* locates in GPU's global memory and others locate in shared memory in the GPU. On line 1, *vgid* is assigned with value that represents the index of the vector group. On line 2, *vgtid* is assigned with value that represents the id of the current thread in the group. From line 3 to line 13 is the kernel that will be executed `NumIterations` times, and the value of iterations is defined by the user. On line 4, *workTile* will store the results from `gatherWork()`. The function of `gatherWork()` is to search unexplored states in global memory and copy those states to the given location. Because `gatherWork()` will be executed in parallel between threads, it is necessary to synchronize the threads (line 5) so that all writes to *workTile* are finished before executing next line of code. After synchronization, each vector group takes a state from *workTile* (line 6), the state will be marked old in line 7, then the threads in that group will generate its successors in parallel (line 8). The generated states are stored into *cache* via `storeInCache()` if there are no duplications (line 9) in there. When all threads are done with successor generations (line 10) they will copy new states (identified by `isNew()`) into global memory via `findOrPutWarp()` (line 12-13). Then the copied states are marked old in the cache for local duplicate detection later on (line 14). The algorithm is executed repeatedly until there is no unexplored states.

Note that Algorithm 4 does not address any low level optimizations such as only allowing blocks with unexplored state to scan, saving last searching location or parallel duplication detection.

Algorithm 4: GPUexplore exploration algorithm

```
   Data: __global__tabel[ ], __shared__cache[ ], workTile[ ]
 1 vgid ← threadId.x/nrOfLts; // index of the vector group
 2 vgtid ← threadId.x mod nrOfLts; // id of the thread in the group
 3 foreach i ∈ 0..NumIterations do
 4     workTile ←gatherWork();
 5     __syncthreads();
 6     s ← workTile[vgid];
 7     markOld(s);
 8     foreach t ∈ succ_vgtid(s) do
 9         storeInCache(t);
10     __syncthreads();
11     foreach t ∈ cache do
12         if isNew(t) then
13             findOrPutWarp(t);
14             markOld(t);
```

Chapter 3

Bugs fixed

There were two bugs in GPUexplore, one is the hash table inaccuracy problem, the other is the inaccurate error reporting problem. This chapter explains the reasons for the bugs and the means to fix them.

3.1 Hash table inaccuracy problem

It is reported by Wijs and Bošnački [2][3] that due to the lack of atomic actions for multiple integers, the hash table may be inaccurate. Then [4] solved the inaccuracy problem by adding a padding in the middle of a bucket to prevent state slots from spanning more than a half-bucket. However, the inaccuracy problem still exists in the platform used in this project. The problem happens more frequently on the GPU (GTX 1080) that has slower global memory access speed.

The problem happens when a state is being stored in the hash table. Recall that states in GPUexplore are represented by state vectors, which is concatenations of several 32-bit integers. The hash table are divided into *buckets*, each bucket contains 32 *slots* and each slot can store a 32-bit integer.

The state vectors are stored by the collaboration of the threads in a warp. Take the number of integers used for a state vector as n, when a warp is trying to store a state vector it first calculates the destination bucket of this state, then each thread in the warp checks the occupation of its corresponding slot in that bucket; after that, the occupation result of each thread is broadcast to other threads (all threads in the warp will know the occupation status of this bucket); if there are empty and enough slots to store the state vector, the thread with smallest slot index will store the 1^{st} integer of the state vector while next $n-1$ threads store the subsequent integers. We leave out the storing procedures for full bucket situation for clarity. The inaccuracy problem happens when a warp is storing a state vector at a bucket while another warp is checking the occupancy of the same bucket.

To ensure correct, lock-less hashing, it is important that element insertion can be done atomically. The storing are done by CUDA atomic operation *compare-and-swap*. And because the operation only supports individual integers (either 32-bit or 64-bit), when the state vectors requires more than 64 bits, multiple operations are needed. The *compare-and-swap* itself is atomic but the group of *compare-and-swap* used for the state vector is not atomic. Since the writes to the global memory (where the hash table is located in) are not immediately visible to the threads, it can happen that only one *compare-and-swap* is visible to other threads while the rest *compare-and-swap* are invisible. In this case, if there is another warp checking the occupancy of this bucket, it may consider the invisible slots as empty, which will result in a part of the state vector being overwritten.

The means to fix this problem is to check the number of occupied slot before storing state vectors. Since the number of integers for all state vectors are same. If the number of occupied slots in a bucket is not 0 and not a multiple of n, then there is a state vector is being stored in the bucket but not all integers of it are visible at this moment. Hence, the warp should re-check the number of occupied slots of the same bucket until it is a multiple of n before it starts to store the state vector. In this way, state vectors will not be overwritten. The test results of this solution is illustrated in Section 6.2.

3.2 Inaccurate error reporting

GPUexplore will produce wrong number of states and report that there is no enough memory for the hash table. However, GPUexplore sometimes will fail to report even if the number of states is wrong. The problem is caused by WAW hazards in *contBFS*.

There are three types of data hazards, namely read after write (RAW), write after read (WAR) and write after write (WAW). RAW occurs when a thread is trying to read a value before it is written by another thread; WAR occurs when a thread is trying to write a value before it is read by another thread; WAW occurs when a thread is trying to write a value before it is written by another thread.

Located in the global memory, *contBFS* indicates whether more explorations are needed. Below are the meanings of the variable with different values:

1. *contBFS*= 0: No exploration is needed.
2. *contBFS*= 1: More explorations are needed.
3. *contBFS*= 2: Error happened during explorations.

We define exploration kernel as the kernel that runs Algorithm 4. There are two places where the value of *contBFS* will be modified in the exploration kernel. One is during the exploration, if a block has scanned an unexplored state, it will set *contBFS* to 1; the other is when a block failed to store a state in the hash table (e.g. because of full hash table), it will set *contBFS* to 2. Because synchronizations in the exploration kernel are targeting threads inside a block, the executions of blocks are asynchronous. As a result, a block could set *contBFS* to 1 even if it was already set to 2 by another block (WAW hazard). In this case, GPUexplore will fail to report the full hash table error.

Because the writes to the global memory are not immediately visible to the threads, we cannot get the up-to-date value of *contBFS*. Hence, using the value of *contBFS* as a condition to specify whether it can be set to 1 or not is infeasible. To make sure all full hash table errors are reported, we introduce an extra variable *fullGPU* to specify whether a hash table is full. The WAW hazards happen for *fullGPU* too but they are harmless. Because all value modifications to *fullGPU* are setting it to 1 (when errors occur), and all blocks are synchronized at the end of the exploration kernel, all writes will success and one of the writes will not be overwritten. In other words, if at least one block sets *fullGPU* to 1, at the end of the exploration kernel *fullGPU* will equal to 1. In this way, GPUexplore will report all errors correctly.

Chapter 4

Tool design

GPUexplore is developed for shared-memory many-core architectures, typically aiming for NVIDIA GPUs. However, with extra GPUs, the architecture becomes distributed-memory many-core architecture. This chapter illustrates two approaches to develop multi-GPU GPUexplore. One approach is using unified memory to create unified hash tables, the other is using a small amount of memory as a buffer in each GPU to store states for other GPUs.

4.1 Unified hash table approach

Recall Algorithm 4, in GPUexplore all threads first store generated states in shared memory (cache), then they store the states in the global memory (hash table). The storing of states requires duplication detection, which is done by checking the existence of the state in the target memory location. Duplication detection in the shared memory can reduce the number of accesses to the global memory by storing each state in each shared memory no more than once. Duplication detection in the global memory can reduce the number of total state by storing each state only once.

However, with extra GPUs, the shared-memory many-core architecture becomes distributed-memory many-core architecture since one GPU cannot directly access other GPUs' memory. Considering the enormous number of generated states, the indirect accessing method (see Section 2.2) is too time-costly for duplication detection in the global memory. One way to solve this problem is using unified memory to combine the memories from GPUs. In that way, the multiple GPUs can be regarded as a shared-memory many-core architecture.

As mentioned in Section 2.2.2, unified memory is the combination of system's main memory and GPU memories. For data declared using unified memory, one GPU can dereference pointers from other GPUs without host's help. The approach is combining the hash tables from all GPUs into one unified hash table, all GPUs work cooperatively as if they are a shared-memory many-core architecture. This approach works very similar to Algorithm 4, the differences between them locate in *table* and `gatherWork()`. The changes are the following:

- *table*: The data type of *table* will be **__managed__** (changed from **__global__**), which is the key word for declaring variable using unified memory.
- `gatherWork()`: Each GPU searches a evenly-distributed-part of the unified hash table (changed from searching the whole table). The function has an extra parameter *gpuId* to indicate the searching range. Hence, the function call in this approach is `gatherWork(gpuId)`.

It is important to prevent data hazards (see Section 3.2) when programming for parallel threads because the hazards may affect the output. The hazards are solved with CUDA atomic function `atomicCAS(address,compare,val)`. Atomic function means the function calls to this function are queued, only one thread will be executing the function at a time. Atomic function

`atomicCAS(address,compare,val)` reads value *old* located at `address` and compares it with `compare`. If *old* equals to `compare`, then the function stores `val` to `address`. The function returns *old*, which can be used to check whether a write is successful.

RAW and WAW hazards could happen without using atomic actions. The hazards may happen in `findOrPutWarp()` (line 13 in Algorithm 4), where several warps are trying to store a state at the same bucket in the global memory. When a warp tries to store a state, it first checks whether the target slots already contains states; if yes, the warp will store the state at other slots; if not, the warp will try to store the state a number of times (specified by user) at different slots/buckets until it succees or it report that they it to store the state. The explanation of why hazards could happen and how they are solved is the following:

- **RAW**: When a warp is checking the occupation status of a slot where another warp is writing on that slot, RAW happens. Because reads/writes are pipelined in NVIDIA GPU, the order of reads/writes is nondeterministic. Hence, a warp may take a slot as empty while in reality it is not, which situation will cause two or more warps writing on the same location (WAW).
- **WAW**: When two or more warps are writing on the same memory location, WAW happens. By using `atomicCAS(address,compare,val)` we can assure that only one warp will write successfully, the failed warps will change their slots/buckets and retry writing.

However, this check-and-retry method in unified hash table approach is infeasible. The reason is that `atomicCAS(address,compare,val)` is a device wide function that ensures atomicity among the threads within one device (GPU). But in our case, the threads may belong to different devices (GPUs). Although NVIDIA has a system wide `atomicCAS_system(address,compare,val)` that can ensure atomicity among all threads between GPUs and CPUs, using this API will introduce extra page faults. We failed to figure out the reasons for the page faults because no document of the implementation of `atomicCAS_system()` can be found. Theses extra page faults may slow down the application more than 100 times so that the application will have no runtime advantages. Hence, until NVIDIA provides a system wide atomic function that will not introduce page faults, the unified hash table approach is considered infeasible.

4.2 Store in buffer approach

Another approach is to use a small amount of memory as a buffer in each GPU to store states for other GPUs.

4.2.1 Methodology

Since combining the hash tables to make the GPUs work like a shared-memory architecture failed, we move on to distributed model checking which is aiming for distributed-memory architectures. Recall the distributed state-space exploration algorithm (see Algorithm 2), states belong to other nodes (machines) are sent asynchronously and stored in a waiting list. When a node receives a state, the node is preempted. It will first receive and explore states from the waiting list in a first-in-last-out (FILO) manner, then explore states generated by itself.

In order to apply this algorithm to GPUexplore we need to address the following difficulties:

- **There is no preemption support for NVIDIA GPUs at the moment.**
  Instead of using preemption, we can let the GPUs receive states at the end of exploration kernel. As long as all states are explored, the exploration moment for each state is irrelevant to the correctness of the state-space.

- **The sending of states is asynchronous, but indirect inter-GPU memory access needs synchronizations.**
  Instead of indirect inter-GPU memory access, we can use unified memory to make the waiting list accessible to all GPUs. In that way, if the waiting list is implemented as a buffer, the sending of states can be asynchronous respect to the receiving.

In the FILO buffer, there is a read index and a write index which indicate the memory location to read/write. The indexes should be accessible to all GPUs so that all of them can have the correct index to read/write. The indexes are updated using atomic actions, so each thread in the GPUs will get a unique index which prevents WAW hazard.

However, despite the fact that the indexes are using unified memory, which makes it accessible to all GPUs, the coherence of the indexes is not guaranteed. Listing 4.1 is a fragment of code that demonstrates this problem. In the example, GPU1 will continuously and atomically increase the unified index by 1; GPU2 will start to print "Hello world!" to console if the index is greater than 10. In some cases, GPU2 will never print "Hello world!" to console even though the value of index reported by GPU1 exceeds 10. By debugging the application we found that GPU2 will keep getting the same value of index rather than getting the most recent value. As a result of this coherence problem, the concept of using FILO buffer is infeasible.

```
__managed__ int index = 0;
if (gpuId == 1) {
  while (1) {
    atomicAdd_system(&index, 1);
  }
}
if (gpuId == 2){
  while (index > 10) {
    printf("Hello world!\n");
  }
}
```

Listing 4.1: Index coherence problem example

Given that the asynchronous sending is infeasible, we need to synchronize the sending and receiving actions. The idea is to store all generated states that belong to other GPUs (we name them *vehicle states*) in a buffer located in the GPU's own global memory during explorations. At the end of each iteration of exploration, vehicle states in the buffer are sent to the corresponding GPU and are explored in the next exploration kernel. In this way, a buffer will only be accessed by one GPU at a time, which prevents data hazards between GPUs.

The main activities of the buffer is insertion and deletion. The time complexity of both actions depend on the data structure of the buffer. Because GPU is single instruction multiple threads architecture, all threads can access memory simultaneously if they are accessing different and coalesced memory. As a result, regardless of the data structure of the buffer, the time complexity of deletion is $O(1)$. Hence, we investigate the time complexity of insertion for various data structures. Generally, insertion in a hash table has $O(1)$ time complexity in average and $O(n)$ in the worst case. In our case, due to the limited memory per GPU, the buffer size should be as small as possible. As a result of small hash table, the chance of collisions increases rapidly as the buffer is filled up, which leads to worse case insertion time. Therefore, we choose to implement the buffer as an array buffer where the storage index is incremental. The incremental index ensures the insertion of the array buffer has time complexity of $O(1)$. And an array buffer will not waste storage space, which is impossible for other data structures.

Because the experiment platform for this project has two GPUs, the implementation of multi-GPU version of GPUexplore supports up to two GPUs at the moment. We name this version of GPUexplore as GPUexplore-2-GPU. The working principle of GPUexplore-2-GPU is illustrated in Figure 4.1. Note that all retrieving/storing of a state are a collaboration of threads in a warp, the number of threads in collaboration depends on the length of the state vectors. For simplicity

we say the length of the state vectors is 1, then there will be only one thread working on a state. The flowchart represents the workflow of each thread in the application. Firstly, the thread checks if there are more states that need exploration, if this is true then the thread will retrieve an unexplored state and store it into shared memory, otherwise it will count how many states have been explored in cooperation with other threads; after retrieving a state, if the state belongs to the other GPU (this is due to buffer overflow which will be explained in Section 7.1.2) the thread will store the state into the buffer, otherwise it will start to generate the successors of the state in cooperation with other threads in the same vector group; after successor generation, for all successors of a state, states belonging to the other GPU will be stored in the buffer while others will be stored in the hash table; after all states have been explored and stored in buffer/hash table, the contents in the buffers of the 2 GPUs are swapped and then stored in the hash table; then the next iteration of the application starts.

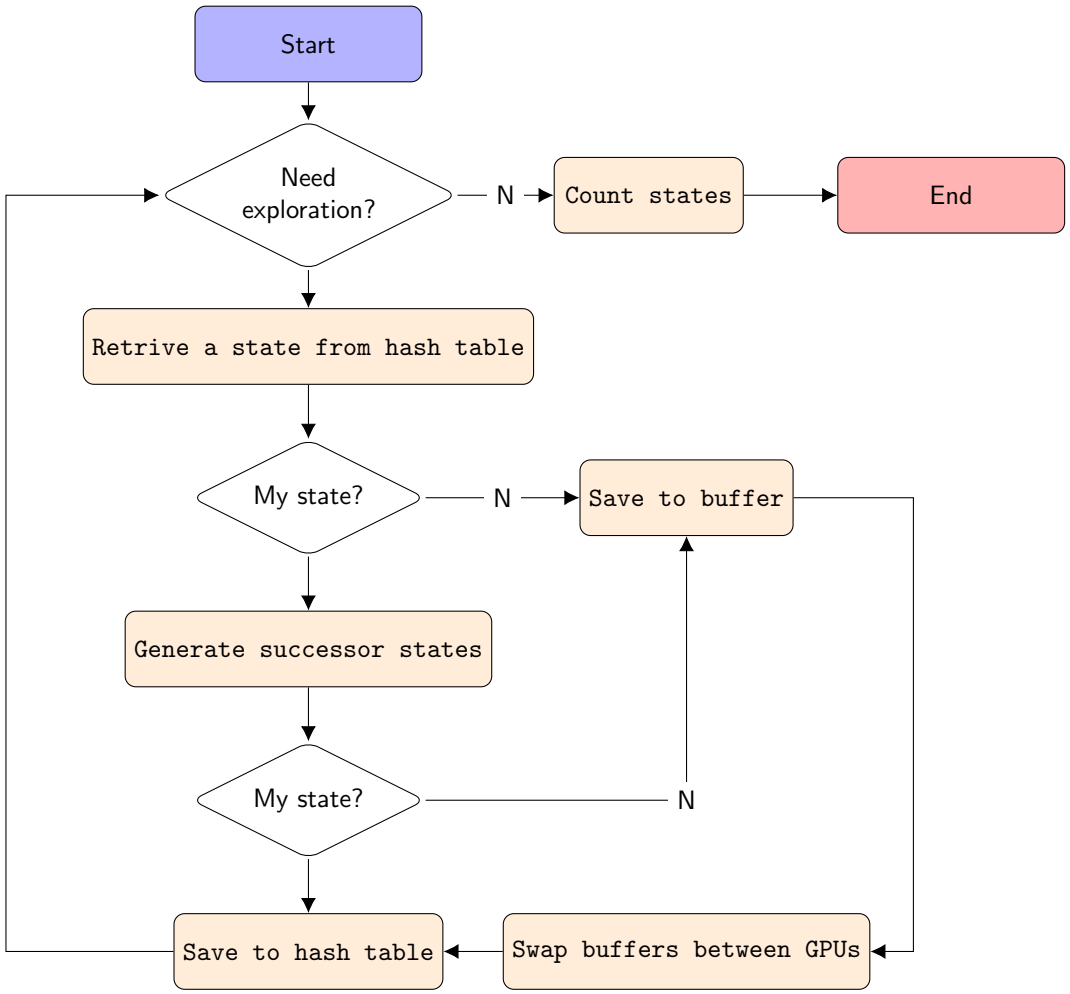

Figure 4.1: Workflow of GPUexplore-2-GPU

4.2.2 Implementation

We propose GPUexplore-2-GPU host algorithm in Algorithm 5. The host algorithm means the algorithm is executed by the CPU, it determines the timings for the GPUs to launch exploration kernels, to exchange buffer contents and to terminate. The GPU id is determined by NVIDIA which starts from 0 and is incremented by 1 for each extra GPU. Firstly, the host lets both GPUs

start exploration by consecutively setting the current GPU (line 2-4); after both GPUs finished exploration they synchronize on line 6 (`cudaMemcpy()` has a built-in `cudaDeviceSynchronize()`). Then in line 6 and line 7 the data in GPU buffers are copied to host buffers; then, the vehicle states from GPU2 are copied to the buffer in GPU1 (line 9), and then GPU1 starts to store vehicle states from the buffer to its hash table (line 10); the same procedure is repeated for GPU2 (line 11 to line 13); at last, the CPU takes the continue flags from both GPUs to determine whether next iteration of exploration is needed (line 14 and 15).

Algorithm 5: GPUexplore-2-GPU host algorithm

```
   Data: __global__gpu1Buffer[ ], __global__gpu2Buffer[ ] // buffers in GPUs
   Data: hostBuffer1[ ], hostBuffer2[ ] // temperately hold data in GPU buffers
   Data: contBFS1, contBFS2 // flags for GPU kenel termination
   Data: hostContBFS1, hostContBFS2
 1 while hostContBFS = 1 ∨ hostContBFS2 = 1 do
 2     cudaSetDevice(0);
 3     GPUexplore2GPU(gpu1Buffer, contBFS1);
 4     cudaSetDevice(1);
 5     GPUexplore2GPU(gpu2Buffer, contBFS2);
 6     cudaMemcpy(hostBuffer1, gpu1Buffer); // copy vehicle states from GPU to host
 7     cudaMemcpy(hostBuffer2, gpu2Buffer);
 8     cudaSetDevice(0);
 9     cudaMemcpy(gpu1Buffer, hostBuffer2); // copy vehicle states from host to GPU
10     bufferToTable(gpu1Buffer, contBFS1);
11     cudaSetDevice(1);
12     cudaMemcpy(gpu2Buffer, hostBuffer1);
13     bufferToTable(gpu2Buffer, contBFS2);
14     cudaMemcpy(hostContBFS1, contBFS1); // copy flags from GPU to host
15     cudaMemcpy(hostContBFS2, contBFS2);
```

As an extension of Algorithm 4, Algorithm 6 describes the exploration kernel for 2 GPUs. The algorithm will set *contBFS* to 0 when it terminates and no unexplored state is found. The variables used in this algorithm are *table* (used by `findOrPutWarp()`), *cache* (used by `storeInCache()`) and *workTile* (to store the results from `gatherWork()`). *table* locates in GPU's global memory and others locate in shared memory in the GPU.

On line 1, *vgid* is assigned with value that represents the index of the vector group (responsible for successors generation of a state vector). On line 2, *vgtid* is assigned with value that represents the id of the thread in the group. Line 3 to line 25 is the exploration procedure that will be executed `NumIterations` times, which value is defined by the user. On line 4, *workTile* will store the results from `gatherWork()`. The function of `gatherWork()` is to search unexplored states in global memory and copy those states to the given location, and if an unexplored and not-vehicle-state has been found, it will set *contBFS* to 1 indicating more iterations of exploration are needed. For each state s in *workTile* (line 6), the responsible GPU is calculated via `h()`. If s belongs to current GPU (line 7), all successors of it will be generated (line 8) and stored in *cache* via `storeInCache()` (line 9). Then state s will be set to old indicating the state has been explored (line 10). If s belongs to the other GPU (line 11) and if the buffer is not full (line 12), state s will be stored in the buffer via `storeInBufferWarp()`. If the buffer is full, state s will be stored in the buffer again (line 12-14) in the next iteration. Because the writes to *cache* are in parallel, it is to necessary to synchronize the threads (line 15) to ensure all writes to *cache* are visible to other threads. After synchronization, for all state t in *cache* (line 16), if t is a new state and belongs to current GPU (line 18), it will stored in the hash table via `findOrPutWarp()` (line 19). If t is a new state and belongs to the other GPU (line 20), the state will be stored in the hash table if the

buffer is full (line 21 and 22). And if the buffer is not full (line 23), t will be stored in the buffer via `storeInBufferWarp()` (line 24). At last, if t is new state, it will be marked as old (line 25).

Algorithm 6: `GPUexplore2GPU(`$gpuBuffer[]$`,` $contBFS$`)`

```
   Data: __global__tabel[ ], __shared__cache[ ], workTile[ ]
 1 vgid ← threadId.x/nrOfLts; // index of the vector group
 2 vgtid ← threadId.x mod nrOfLts; // id of the thread in the group
 3 foreach i ∈ 0..NumIterations do
 4     workTile ←gatherWork();
 5     __syncthreads();
 6     s ← workTile[vgid];
 7     if h(s)=gpuId then
 8         foreach t ∈ succ_vgtid(s) do
 9             storeInCache(t); // State belongs to current GPU
10         markOld(s);
11     else
12         if !isFullBuffer() then
13             storeInBufferWarp(s); // State belongs to the other GPU
14             markOld(s);
15     __syncthreads();
16     foreach t ∈ cache do
17         if isNew(t) then
18             if h(t)=gpuId then
19                 findOrPutWarp(t);
20             else
21                 if isFullBuffer() then
22                     findOrPutWarp(t); // Store in current GPU
23                 else
24                     storeInBufferWarp(t);
25             markOld(t);
```

Recall that the array buffer uses incremental index for insertion to ensure $O(1)$ time complexity. Algorithm 7 describes the warp version implementation of the buffer. Recall that the states are encoded as state vectors, each state vector contains several 32-bit integers. Each integer in a state vector is stored by one thread. The algorithm ensures each state vector has a unique index for storing, which prevents WAW hazard. On line 1, each thread is assigned with *entryId* which equals to the thread position in its state vector. On line 2, only the thread that is the head of a state vector will update the index. Other threads will wait for it until it starts to execute line 7. From line 3 to 5, all heads of state vector threads will keep trying to increase *stCount* until they succeed. On line 4, the heads of state vector threads will copy the value of *stCount* (all threads in the block can access *stCount*) to *index* (each thread has its own *index*). Then they will try to increase *stCount* by `#IntStateVector` (line 5). As aforementioned, when multiple threads are using `atomicCAS()` at the same memory location, only one will succeed. Hence, on line 5, only one thread will increase *stCount* by `#IntStateVector` and then **break** the while-loop and start to execute line 7. On line 7, all threads copy the value of *index* from the head of state vector thread. After that, the threads store their part of state vector in the buffer (line 8) at location $blockId * sizePerBlock + index + entryId$ where *blockId* is the id of blocks defined in the grid and *sizePerBlock* equals to total buffer size divided by the number of blocks.

Algorithm 7: `storeInBufferWarp(`s`)`

```
   Data: __global__gpuBuffer[ ], index
   Data: __shared__stCount // number of states stored in gpuBuffer
1  entryId ← i^th thread in a state vector;
2  if entryId = 0 then
3      while true do
4          index ← stCount;
5          if atomicCAS(addressOf(stCount),index,index+#IntStateVector)=success
            then
6              break;
7  index ← __shfl_sync(FULLMASK,index, head of state vector);
8  gpuBuffer[blockId * sizePerBlock + index + entryId] ← s;
```

The buffer is divided into a number of distinct parts equal to the number of blocks (defined when launching exploration kernel). All threads in a block will only store states in its corresponding part. If the buffer is not divided, the consequences can be the following:

1. All threads in a GPU will try to read/update *stCount*, which results in higher runtime due to `atomicCAS()`.
2. Some threads may not **break** the while-loop in Algorithm 7. Because memory coherence for global variables inside a **while** loop is not guaranteed, they will keep getting an old copy of *stCount*, which results in `atomicCAS()` will never succeed.

Every time when both GPUs finished one exploration iteration, they will exchange the states in the buffers. Then, they will copy the exchanged states into their own hash tables. The copy procedure is described in Algorithm 8. The variable used in the algorithm is *table* which is the hash table used for storing states. On line 1, all states in *gpuBuffer* are copied to the hash table via `findOrPutWarp()` (line 2). Then on line 3, the exploration indication flag *contBFS* will be set to 1 (more exploration needed). After all states have been stored in the hash table, all threads synchronize on line 4, and they reset the *gpuBuffer* to empty slots cooperatively on line 5.

Algorithm 8: `bufferToTable(`$gpuBuffer[]$`, `$contBFS$`)`

```
   Data: __global__tabel[ ]
1  foreach s ∈ gpuBuffer do
2      findOrPutWarp(s);
3      contBFS←1;
4  __syncthreads();
5  resetBuffer(gpuBuffer);
```

Chapter 5

Optimizations

This chapter introduces the methods to improve the performance of GPUexplore-2-GPU. The improvements take the advantage of unified memory and peer-to-peer access CUDA features to reduce communication time between GPUs.

5.1 Increase data transfer speed

The performance of GPUexplore-2-GPU is worse than GPUexplore. In average GPUexplore-2-GPU without any optimizations is 2 to 70 times slower than GPUexplore (the runtime can be seen in Figure 6.4). The worse performance is the result of considerable amount of communications between the GPUs.

One way to reduce communication time between GPUs is increasing data transfer speed. We consider a transfer is complete when the data is visible to the destination GPU. The two GPUs are connected via PCI Express (PCI-e) 3.0 bus with 8 lanes each. The PCI-e with 8 lanes has a fixed bandwidth of 7.88GB/s in each direction (each lane has two channels). Hence, we can only increase data transfer speed by reducing communication overhead or reducing the number of communications. Recall Algorithm 5, the GPUs communicate with each other with the help of CPU. Firstly, the states in buffers are copied to main memory via PCI-e bus, then the states are copied to the correspond GPU through the same bus. We can reduce the number of communications by using the peer-to-peer communication feature of CUDA.

5.1.1 Peer-to-peer communication

Since CUDA 4.0[1], NVIDIA GPUs with Fermi (or later) architecture can use peer-to-peer communication feature. The feature allows GPUs to communicate with each other without the help of CPU and no CUDA APIs are needed. By applying this feature to GPUexplore-2-GPU, each state in the buffer can be copied from one GPU to another GPU directly, which will reduce nearly 50% time spent on PCI-e bus. Furthermore, because peer-to-peer communication does not need CUDA APIs, it will also reduce the overhead time introduced by the APIs.

The optimized GPUexplore-2-GPU host algorithm is shown in Algorithm 9. Firstly, the host lets both GPUs start exploration by consecutively setting the current GPU (line 2-4); after both GPUs finished exploration they synchronize on line 6 via `cudaDeviceSynchronize()`; then each GPU directly takes the states stored in another GPU's buffer using peer-to-peer communication (line 7 to 10); at last, the CPU takes the continue flags from both GPUs to determine whether next iteration of exploration is needed (line 11 and 12).

[1] https://developer.nvidia.com/cuda-toolkit-40

Algorithm 9: GPUexplore-2-GPU host algorithm with peer-to-peer communication

```
   Data: __global__gpu1Buffer[ ], __global__gpu2Buffer[ ] // buffers in GPUs
   Data: contBFS1, contBFS2 // flags for GPU kenel termination
   Data: hostContBFS1, hostContBFS2
1  while hostContBFS = 1 ∨ hostContBFS2 = 1 do
2      cudaSetDevice(0);
3      GPUexplore2GPU(gpu1Buffer, contBFS);
4      cudaSetDevice(1);
5      GPUexplore2GPU(gpu2Buffer, contBFS2);
6      cudaDeviceSynchronize();
7      cudaSetDevice(0);
8      bufferToTable(gpu2Buffer, contBFS);
9      cudaSetDevice(1);
10     bufferToTable(gpu1Buffer, contBFS2);
11     cudaMemcpy(hostContBFS1, contBFS1); // copy flags from GPU to host
12     cudaMemcpy(hostContBFS2, contBFS2);
```

5.1.2 Peer-to-peer communication limitation

Despite that using peer-to-peer communication reduces the time spent on PCI-e busby nearly 50%, there are some cases where this feature will not improve the performance. When a GPU starts dereferencing a pointer pointing to the memory belongs to another GPU, it tends to copy all allocated memory for that pointer into its own global memory. In this case, when the buffer is not full of states, the more empty slots in the buffer, the more unnecessary time is spent on copying. However, the number of vehicle states varies for different models, it is impossible to know the optimal buffer size for each model before exploration. As a result, we experimentally choose the buffer size based on the runtime results for all test models.

5.2 Reduce data transfer size

Another way to reduce communication time between GPUs is reducing data transfer size. The original GPUexplore-2-GPU will transfer all states (even empty slots) in the buffer, but only vehicle states are useful. Recall Algorithm 7 where each block stores the number of generated vehicle states in *stCount*, we can use this value to improve the transfer process by only transferring vehicle states.

5.2.1 Keep track of generated states

Because each block has the number of generated vehicle states, and those states are stored in the corresponding region in the buffer, we can pinpoint the memory location of vehicle states generated by each block. For a block with *blockId*=i and the number of generated vehicle states *stCount*=a, the vehicle states generated by this block will be stored in $[i * BufferSizePerBlock, i * BufferSizePerBlock+a]$ in the buffer. As a result, for each block in a GPU, it only needs to transfer *stCount[i]* states to the other GPU. Based on this information, we optimized GPUexplore-2-GPU so that it only transfers vehicle states instead of the complete buffer to the other GPU. The optimized GPUexplore-2-GPU is described in Algorithm 10.

The newly introduced variables *gpu1StCount* and *gpu2StCount* are array variables that stores all *stCount* from all blocks in the target GPU. *hostStCount1* and *hostStCount2* are temporary variables to hold values in *gpu1StCount* and *gpu2StCount*. On line 1, the number of blocks in the GPU is assigned to *nblocks*; and the buffer size for each block is assigned to *bs* to calculate the memory location for vehicle states; after that, the host lets both GPUs start exploration

by consecutively setting the current GPU (line 4-7); after both GPUs finished exploration they synchronize on line 8 (`cudaMemcpy()` has a built-in `cudaDeviceSynchronize()`); then in line 8 and 9, the *stCount* from all blocks (stored in *gpu1StCount/gpu2StCount*) are copied to the host buffers; from line 10 to 14, the host copies vehicle states from GPU buffers to host buffers from block 0 to *nblocks* according to the number of generated states, and then swap the states in the buffers; after that, GPU1 starts to store vehicle states from the buffer to its hash table (line 15 and 16); the same procedure is repeated for GPU2 (line 17 and 18); at last, the CPU takes the continue flags from both GPUs to determine whether next iteration of exploration is needed (line 19 and 20).

Algorithm 10: GPUexplore-2-GPU host algorithm with state tracking

```
   Data: __global__gpu1Buffer[ ], __global__gpu2Buffer[ ] // buffers in GPUs
   Data: __global__gpu1StCount[ ], __global__gpu2StCount[ ] // state counters in GPUs
   Data: hostBuffer1[ ], hostBuffer2[ ] // temperately hold data in GPU buffers
   Data: hostStCount1[ ], hostStCount2[ ]
   Data: contBFS1, contBFS2 // flags for GPU kenel termination
   Data: hostContBFS1, hostContBFS2
1  nblocks ← Number of blocks in the GPU;
2  bs ← Buffer size per block;
3  while hostContBFS = 1 ∨ hostContBFS2 = 1 do
4      cudaSetDevice(0);
5      GPUexplore2GPU(gpu1Buffer, contBFS1);
6      cudaSetDevice(1);
7      GPUexplore2GPU(gpu2Buffer, contBFS2);
8      cudaMemcpy(hostStCount1, gpu1StCount); // copy state counts from GPU to host
9      cudaMemcpy(hostStCount2, gpu2StCount);
10     for i = 0; i < nblocks; i++ do //Swap vehicle states between GPUs
11         cudaMemcpy(hostBuffer1[i * bs], gpu1Buffer[i * bs], hostStCount1[i]);
12         cudaMemcpy(hostBuffer2[i * bs], gpu2Buffer[i * bs], hostStCount2[i]);
13         cudaMemcpy(gpu1Buffer[i * bs], hostBuffer2[i * bs], hostStCount2[i]);
14         cudaMemcpy(gpu2Buffer[i * bs], hostBuffer1[i * bs], hostStCount1[i]);
15     cudaSetDevice(0);
16     bufferToTable(gpu1Buffer, contBFS1);
17     cudaSetDevice(1);
18     bufferToTable(gpu2Buffer, contBFS2);
19     cudaMemcpy(hostContBFS1, contBFS1); // copy flags from GPU to host
20     cudaMemcpy(hostContBFS2, contBFS2);
```

State tracking optimization can greatly reduce the number of communications if the buffer size is large and most of them are empty. However, depending on the number of blocks, the overhead introduced by CUDA APIs could lead to performance penalties.

5.3 Combined approach - unified buffer

GPUs with compute capabilities 3.0 or higher (Kepler class or newer) can benefit from unified memory, which not only simplifies GPU programming but also automatically migrate data to increase access speed. In our case, we can combine Algorithm 9 and Algorithm 10 with the help of unified memory. One drawback of the Algorithm 10 is the overhead introduced by thousands of CUDA API calls. Because variables using unified memory do not need explicit commands to be migrated, we can declare buffers to use unified memory so the thousands of APIs are no longer needed. As a result, we propose Algorithm 11. When not considering the

implementations, the working principle of this algorithm is the same as the one using peer-to-peer communication (Algorithm 9). The differences are located in the declaration of buffers and `bufferToTable()`. Buffers in Algorithm 11 are declared using key word **__managed__**, which represents the variable is using unified memory; `bufferToTable()` in this algorithm takes an extra argument *gpu1StCount*/*gpu2StCount* which guarantees only vehicle states (instead of the complete buffer) will be copied.

Algorithm 11: GPUexplore-2-GPU host algorithm with unified memory

```
   Data: __managed__gpu1Buffer[ ], __managed__gpu2Buffer[ ] // buffers in GPUs
   Data: __global__gpu1StCount[ ], __global__gpu2StCount[ ] // state counters in GPUs
   Data: contBFS1, contBFS2 // flags for GPU kenel termination
   Data: hostContBFS1, hostContBFS2
 1 while hostContBFS = 1 ∨ hostContBFS2 = 1 do
 2     cudaSetDevice(0);
 3     GPUexplore2GPU(gpu1Buffer, contBFS);
 4     cudaSetDevice(1);
 5     GPUexplore2GPU(gpu2Buffer, contBFS2);
 6     cudaDeviceSynchronize();
 7     cudaSetDevice(0);
 8     bufferToTable(gpu2Buffer, gpu2StCount, contBFS);
 9     cudaSetDevice(1);
10     bufferToTable(gpu1Buffer, gpu1StCount, contBFS2);
11     cudaMemcpy(hostContBFS1, contBFS1); // copy flags from GPU to host
12     cudaMemcpy(hostContBFS2, contBFS2);
```

5.3.1 Unified buffer analysis

The performance of the unified buffer depends on the number of page fault (definition see Section 2.2.2). More page faults will lead to longer runtime. When a page fault happens, the page will be migrated to the device requiring the page to prevent further page faults. The page size is determined by GPU's built-in migration mechanism, which cannot be modified manually. After a number of experiments, the results show that page sizes are often 64KB or more (up to 2MB); and *BufferSizePerBlock* larger than 40KB often proves to be a waste of memory. Combining the experiment results we can find that when page size is greater than *BufferSizePerBlock*, extra page faults will be introduced.

An example helps to understand the scenarios where extra page faults are introduced. Take *PageSize* = 64KB, *BufferSizePerBlock* = 40KB. At the end of an exploration iteration, when block 0 in GPU2 is accessing *gpu1Buffer[0]*, states starting in *gpu1Buffer[0]* and the first 24KB of *gpu1Buffer[1]* will be migrated from GPU1 to GPU2. If all slots in *gpu1Buffer[1]* are empty, then at the next iteration, when GPU1 is trying to store vehicle states in *gpu1Buffer[1]* a page fault will happen. This page fault will not happen if *PageSize* = *BufferSizePerBlock*, because in that case, no states in *gpu1Buffer[1]* will be migrated.

Since at the moment we cannot manually set page size, we need to consider other methods to reduce the number of page faults. One way to eliminate the introduced page faults is to prevent the state being migrated away. CUDA provides access control over unified memory, we can simply add one line of code to specify that no data migrations are needed.

In conclusion, this unified buffer approach reduces communications in the following ways:

1. Reduced around 50% time spent on PCI-e bus through direct GPU memory accesses.

2. Reduced overhead due to CUDA APIs through direct GPU memory accesses.
3. Reduced number of states being transferred by neglecting empty slots.

5.4 Inconsecutive state transfer

There are some situations where a state transfer is not needed. For example, at the end of an exploration iteration, if only 10% of the buffer is filled with vehicle states, then it is unlikely that the buffer will be full at the end of the next iteration. So it is reasonable to skip the state transfer process for this iteration.

Hence, we added an option for users to specify the number of iterations between two state transfer processes. Moreover, to ensure all GPUs are not idle, we force state transfer to happen if one or two GPUs have no states to explore.

This method can be applied to GPUexplore-2-GPU plus one of the three aforementioned methods while those three are mutually exclusive. Because in GPUexplore-2-GPU every vehicle state will be transferred to the destination GPU eventually, the inconsecutive state transfer method only reduces communication time spent on empty slots. As a result, different combinations of this method with the other three, affect the performance of GPUexplore-2-GPU differently:

1. Inconsecutive + peer-to-peer: Since peer-to-peer communication will transfer empty states, a large buffer with long state transfer periods can reduce the communication time spent on empty slots.
2. Inconsecutive + state tracking: Since there are four CUDA API calls for each *block* when transferring states, a large buffer with long state transfer periods can eliminate the overhead introduced by the API calls.
3. Inconsecutive + unified buffer: Since unified buffer will not spend time on empty slots and will not introduce API overhead, this combination will not improve the performance.

Therefore, a large buffer and long state transfer periods are needed to improve the performance. However, on the other hand, long state transfer periods sometimes has negative effect on the performance. That is due to insufficient workload. Compared to consecutive state transfer, GPUs with inconsecutive state transfer have lighter workload during the exploration because of fewer unexplored states in the hash table. For example, if we explore the same model twice using consecutive and inconsecutive state transfers, it could happen that at some points consecutive state transfer has just enough states (workload) to make use of all resources, which implies that inconsecutive state transfer will have insufficient states (workload) to make use of all resources.

Another negative effect is due to buffer overflow problem (see Section 7.1.2). Since the buffers are reset at the end of state transfer, as the state transfer period increases then the chance for a full buffer also increases. And buffer overflow problem will not only introduce extra memory accesses to the global memory, but also takes up some spaces in the hash table for storing the vehicle states.

In conclusion, in order to benefit from this method, it is essential to choose a reasonable state transfer period so that all GPUs can make use of all resources and prevent buffer overflow. The effect on the performance for various state transfer periods are tested in Chapter 6.

Chapter 6

Experiments

This chapter presents the test results for the hash table inaccuracy problem, the occupancy of the hash table for both versions of GPUexplore, the workload distribution of GPUexplore-2-GPU and the runtime of both versions of GPUexplore.

6.1 Benchmark settings

The models used in the benchmarks (21 in total) have different origins. Five EXP models are taken from CADP toolkit [15], namely, `cache`, `sieve`, `odp`, `transit` and `asyn3`. The `leader_election`, `lamport8`, `lann6`, `lann7`, `peterson7` and `szymanski5` are models come from the BEEM database[1] and have been translated from DVE to EXP. Model `1394`, `acs` and `wafer_stepper.1` are from mCRL2 [11] and are also translated into EXP. Model `gasstation11` and `gasstation12` are models from [16] with 11 and 12 consumers. Finally, model `broadcast` is created by Wijs and Bošnački [3]. Models with .1 suffix are enlarged versions of the original models. Table 6.1 shows the details of the models.

The experiments are conducted on two NVIDIA GTX 1080s. The GTX 1080 has 20 SMs each with 128 CUDA cores, which results in a total of 2560 CUDA cores. Each SM contains 96 KB shared memory and the global memory has a size of 8 GB. For experiments with GPUexplore we allocate 5 GB of the global memory for the hash table. And for experiments with GPUexplore-2-GPU we allocate 2.5 GB of the global memory for the hash table and 40 MB for the buffer in each GPU. As mentioned in Section 2.1, threads are grouped into blocks. Different combinations of threads per block (T) and number of blocks (B) will affect the performance as well as the number of parallel blocks in each SM. To achieve best performance, we need to find out the combination of T and B and the number of parallel blocks in SM that lead to best performance.

First we fix T to 512 because any other value gives longer runtime. Then we determine that each SM can only run two blocks simultaneously. This is because setting it to one will cause not all registers (65535 in total) are used, which results in performance loss; and setting it to other values will cause insufficient registers, which makes some variables are stored in the global memory. And accesses to the global memory is much slower than accesses to registers. Hence, each SM should only run two blocks simultaneously.

6.1.1 Parameter for block

Since GTX1080 has 20 SMs and we determined each SM should only run two block simultaneously, we vary B from 40 to 10240. The results are plotted in Figure 6.1. From the data we can see that for models smaller than `asyn3` (except `transit`), they achieve best runtime between B = 160 and

[1] `http://paradise.fi.muni.cz/beem/`

Table 6.1: Overview of the models used in the benchmarks

Model	#state	#integer per state vector	State space size (MB)
cache	616	1	0.002
leader_election	4,261	3	0.049
acs	4,764	2	0.036
sieve_10	23,627	5	0.451
odp	91,394	1	0.349
1394	198,692	2	1.516
acs.1	200,317	2	1.528
transit	3,763,192	2	28.711
wafer_stepper.1	3,772,753	2	28.784
odp.1	7,699,456	2	58.742
1394.1	10,138,812	2	77.353
asyn3	15,688,570	2	179.541
lamport8	62,669,317	2	475.077
szymanski5	79,518,740	2	606.680
broadcast	105,413,504	2	804.241
peterson7	142,471,098	3	1630.452
lann6	144,151,628	2	1099.790
lann7	160,025,986	2	1220.901
asyn3.1	190,208,728	3	2176.766
gasstation11	243,104,733	2	1854.742
gasstation12	934,450,425	2	7129.291

B = 640, and the performance decreases as B increases. Take P as the number of blocks that leads to best performance, when $B < P$ the performance decreases because insufficient resources are used due to low B; when $B > P$ the performance decreases because B is too large, some blocks may have no work to do but they still need to spend time on communications.

Models larger than `1394.1` achieve good performance when B = 2560. And as B increases the performance slightly improves. This is because for a large model, a higher number of blocks will result in a fine-grained work scanning, which increases the chance for a block to find an unexplored state. And those together increase the utilization of GPU resources. Because we use 40 MB for the buffer, the *bufferSizePerBlock* will decrease as B increases. Small *bufferSizePerBlock* will increase the chance of buffer overflow, which will decrease the performance. And since models smaller than `broadcast` take fewer than 10 seconds to be explored, their runtimes have low priority. Hence, we conducted all later experiments with 2560 blocks, because it allows models larger than `1394.1` achieve good performance while keeping large buffer size per block.

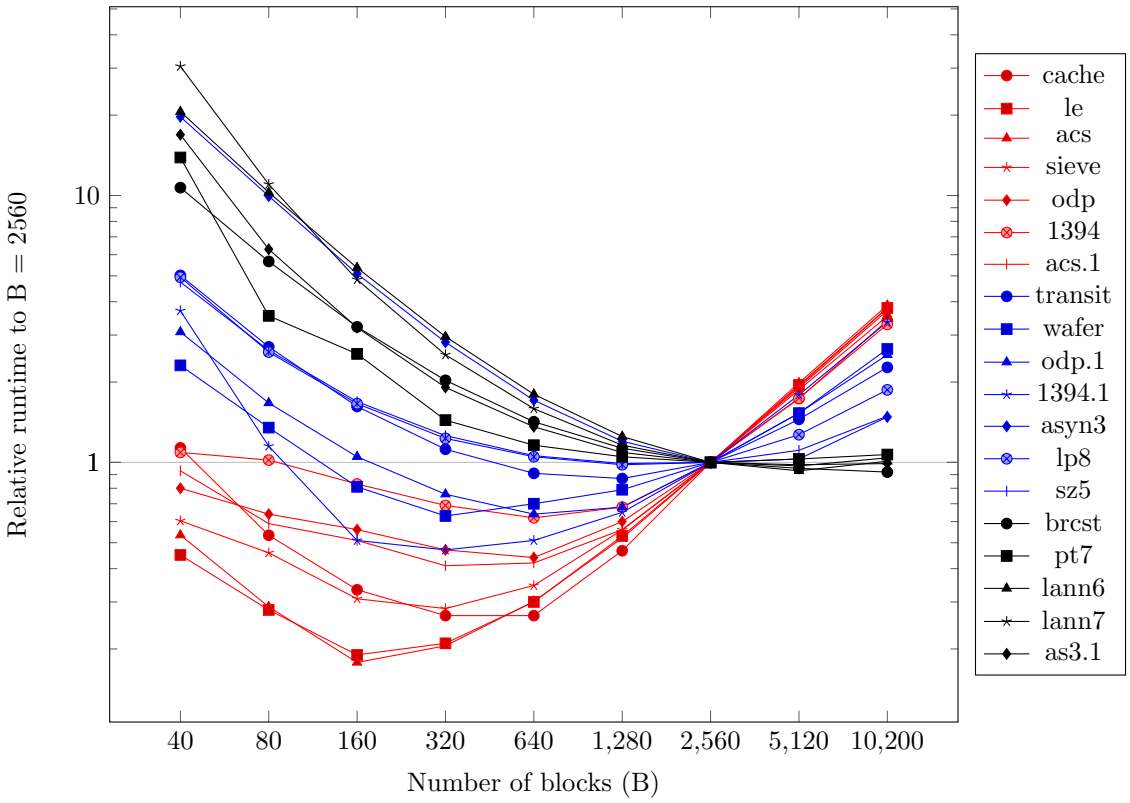

Figure 6.1: Runtime of GPUexplore-2-GPU with different amount of blocks

6.2 Hash table inaccuracy test

In Section 3.1, the hash table inaccuracy problem is explained. We have implemented the proposed method to fix the problem and the result is positive. The best candidate model for the test is the model that has the highest number of integers per state vector and has the highest amount of storing. Hence, despite `sieve_10` has the highest number of integers per state factor, it is not suitable for the test because of few storing. On the other hand, `asyn3.1` is suitable, since it has the highest amount of storing in models that has 3 integers per state vector.

The test consist of 50 runs of GPUexplore and the fixed GPUexplore to explore `asyn3.1` with 512 threads and 2560 blocks. There are 27 out of 50 GPUexplore runs that suffer the inaccuracy problem, which results in a inaccuracy rate of 54% for `asyn3.1`. Then we did another 50 runs on the fixed GPUexplore and all runs provided correct number of states. And no performance loss was observed.

6.3 Hash table occupancy test

There is a macro `NR_HASH_FUNCTIONS` to determine number of retries for element insertions to the hash table. The default value of it is 8. It means that when an element failed to find an empty position after eight attempts, it will report that the hash table is full. We define a occupancy threshold as the value of state-space size divided by the minimum memory required for the hash table(s). The occupancies reflect the memory efficiency of the hash table by indicating the percentage of memory used for storing states for a hash table. We conducted several experiments to find out the occupancy thresholds for `NR_HASH_FUNCTIONS` equals to eight and one hundred[2]. During the experiments we observed similar runtime for models smaller than `peterson7` with different `NR_HASH_FUNCTIONS`, and 5 to 10 percent shorter runtime for models larger than `broadcast` with `NR_HASH_FUNCTIONS` equals to 100. The shorter runtime is because a larger `NR_HASH_FUNCTIONS` will lead to a smaller minimum memory required, and a smaller memory needs fewer scanning procedures.

The experiment results are shown in Table 6.2. From the results we can see that with a larger `NR_HASH_FUNCTIONS`, occupancies for all models increases. Hash tables with `NR_HASH_FUNCTIONS` equals to 8 has an average occupancy of 71.49%, and the occupancy is increased to 95.95% with `NR_HASH_FUNCTIONS` equals to 100. Note that `gasstation12` failed to be stored in the hash table with `NR_HASH_FUNCTIONS` equals to 8 because of insufficient memory of GTX1080. But it can be stored with `NR_HASH_FUNCTIONS` equals to 100.

It is not recommenced to set `NR_HASH_FUNCTIONS` to a very large number since it also specifies the number of hash functions stored in the shared memory. Except the hash functions, the shared memory is also used for storing states locally for duplication detection. The available shared memory for storing states decreases as `NR_HASH_FUNCTIONS` increases. And as the shared memory for storing states decreases, fewer duplications can be found in the shared memory, then the number of duplication detection in the hash table increases. Since the accesses to the global memory (hash table) are much slower than the accesses to the shared memory, a very large `NR_HASH_FUNCTIONS` could result in a longer runtime. Therefore, since `NR_HASH_FUNCTIONS` with value 100 already enables hash tables to reach 90% occupancy for all models without performance penalties, a value larger than 100 for `NR_HASH_FUNCTIONS` is not recommended.

6.3.1 Occupancy comparison

In order to find out the capacity increment of GPUexplore-2-GPU over the original GPUexplore, we need to find out the occupancies for all models using GPUexplore-2-GPU. The experiments were conducted with the same settings as experiments with `NR_HASH_FUNCTIONS`. The results are shown in Table 6.3. From the results we can see that all occupancies of GPUexplore-2-GPU are lower than that of GPUexplore. GPUexplore has average 95.95% occupancy while GPUexplore-2-GPU has 80.16%. This is because the workload of GPUexplore-2-GPU is not always uniformly distributed. For each model, the more uneven the distribution is, the lower occupancy will it has. Only when the workload is uniformly distributed, can it has similar occupancy to that of GPUexplore.

[2]We chose 100 because that value allows the occupancy of `peterson7` to reach 90%

Table 6.2: Occupancies for GPUexplore with different `NR_HASH_FUNCTIONS`

Model	State space size (MB)	Occupancy (NR=8) %	Occupancy (NR=100) %
cache	0.002	88.00	96.25
leader_election	0.049	73.05	92.76
acs	0.036	86.62	99.15
sieve_10	0.451	71.60	91.58
odp	0.349	87.07	98.80
1394	1.516	81.93	98.36
acs.1	1.528	81.76	98.44
transit	28.711	77.59	98.00
wafer_stepper.1	28.784	77.79	98.12
odp.1	58.742	73.33	97.58
1394.1	77.353	75.66	97.72
asyn3	179.541	58.11	90.51
lamport8	475.077	73.26	97.30
szymanski5	606.680	72.29	96.39
broadcast	804.241	55.48	96.71
peterson7	1630.452	50.88	90.00
lann6	1099.790	72.08	97.73
lann7	1220.901	58.19	97.58
asyn3.1	2176.766	52.84	90.00
gasstation11	1854.742	62.33	97.63
gasstation12	7129.291	n/a*	94.39
Average		**71.49**	**95.95**

* - `Gasstation12` with `NR_HASH_FUNCTIONS` = 8 requires more memory than a single GTX 1080 can provide (8GB).

Table 6.3: Occupancy for both versions of GPUexplore with `NR_HASH_FUNCTIONS = 100`

Model	GPUexplore Occupancy %	GPUexplore-2-GPU Occupancy %
cache	96.25	80.21
leader_election	92.76	67.99
acs	99.15	60.36
sieve_10	91.58	48.62
odp	98.80	94.42
1394	98.36	98.36
acs.1	98.44	96.87
transit	98.00	86.51
wafer_stepper.1	98.12	94.32
odp.1	97.58	93.90
1394.1	97.72	51.21
asyn3	90.51	58.39
lamport8	97.30	95.80
szymanski5	96.39	96.39
broadcast	96.71	84.67
peterson7	90.00	89.60
lann6	97.73	73.73
lann7	97.58	54.43
asyn3.1	90.00	69.59
gasstation11	97.63	85.30
gasstation12	94.39	82.69
Average	**95.95**	**80.16**

6.4 GPUexplore-2-GPU memory usage

Based on the results in Table 6.3, we can calculated how many extra states can GPUexplore-2-GPU store compared to GPUexplore. Take the available memory of one GPU as M and the average occupancy threshold of GPUexplore as $O_{GPUexplore}$ then the capacity of GPUexplore is:

$$C_{GPUexplore} = M * O_{GPUexplore}$$

The same applies to GPUexplore-2-GPU, then the capacity of GPUexplore-2-GPU is:

$$C_{GPUexplore-2-GPU} = 2 * M * O_{GPUexplore-2-GPU}$$

Hence, the average capacity of GPUexplore-2-GPU compared to GPUexplore is:

$$\frac{C_{GPUexplore-2-GPU}}{C_{GPUexplore}} = \frac{2 * M * O_{GPUexplore-2-GPU}}{M * O_{GPUexplore}} = 167.08\%$$

6.5 Benchmark results

This section includes the workload distributions for two different hash functions, the memory usage of both versions of GPUexplore and runtime results for GPUexplore-2-GPU with different optimization configurations. For all experiments for GPUexplore-2-GPU (except experiments with `gasstation12`) we allocate 2.5 GB of the global memory for the hash table and 40 MB for the buffer in each GPU; for experiments with `gasstation12` we allocate 6 GB of the global memory instead of 2.5 GB. For all experiments for the original GPUexplore we allocate 5 GB of the global memory for the hash table. In this way, we can compare the runtime difference between both versions of GPUexplore since they have the same hash table size.

6.5.1 Workload balance

We tested two hash functions for determining the destination of a state. The hash function $h1$ determines the destination of a state according to the last 4 bits of an integer, which is the reversal of the penultimate non-zero integer in a state vector. The function returns 0 if the value is less than 7, and returns 1 if the opposite. The function is an experimental function that has good workload balance for models larger than `lann7`. This function is also the default hash functions for determining the destination of a state.

Another hash function $h2$ determines the destination of a state has been introduced to investigate the effects of different functions. This function determines the destinations according to the last bit of the first integer in a state vector. The function returns 0 if the aforementioned bit is 0, and return 1 if the opposite.

Distribution

We tested the two hash functions on GPUexplore-2-GPU with same configurations, the results for workload distribution are shown in Table 6.4. From the results we can see that model `sieve_10`, `1394.1` and `lann7` using $h1$ have uneven workload distributions, and model `asyn3` and `asyn3.1` using $h2$ have uneven workload distributions. Workload distributions higher than 89% are marked in bold. In average, $h2$ has better workload distributions (average 6.96% difference for the two hash tables) than $h1$ (average 13.38% difference for the two hash tables). Note that $h2$ has bad workload balance for `asyn3.1`, and state-space size of `asyn3.1` ranks the second among all test model.

Table 6.4: Workload distributions for the two hash functions

Model	Table 1 ($h1$) %	Table 2 ($h1$) %	Table 1 ($h2$) %	Table 2 ($h2$) %
cache	59.42	40.58	59.42	40.58
leader_election	32.62	67.38	18.28	81.75
acs	38.31	61.69	37.15	62.85
sieve_10	5.36	**94.64**	47.52	52.48
odp	52.68	47.32	52.68	47.32
1394	50.41	49.59	47.20	52.80
acs.1	49.07	50.93	67.15	32.85
transit	56.32	43.77	49.93	50.07
wafer_stepper.1	52.13	47.87	61.70	38.30
odp.1	47.57	52.43	50.34	49.66
1394.1	5.49	**94.51**	52.44	47.56
asyn3	22.02	77.98	**96.74**	3.26
lamport8	49.19	50.81	11.21	88.79
szymanski5	49.03	50.97	54.08	45.92
broadcast	42.86	57.14	71.43	28.57
peterson7	49.57	50.43	40.53	59.47
lann6	33.63	66.37	52.29	47.71
lann7	10.52	**89.48**	41.75	58.25
asyn3.1	35.22	64.78	**96.08**	3.92
gasstation11	42.72	57.28	57.28	42.72
gasstation12	42.03	57.97	57.97	42.03
Average	41.31	58.69	53.48	46.52

Runtime

The results for runtime with $h1$ and $h2$ are shown in Table 6.5. Models marked in bold in Table 6.4 are also marked bold in this table. From the results of this table we can see that except `sieve_10` and `asyn3`, models with more uneven workload distributions tends to have higher runtime. Model `sieve_10` and `1394.1` do not follow the trend because the state-space of them are too small, majority of the time is spent on communications rather than exploration. Model `asyn3` does not follow the trend because 97% of the states are store in GPU1 (hash table 1) when using $h2$, and 78% of the states are store in GPU2 (hash table 2) when using $h1$, and GPU1 has higher global memory access speed than GPU2.

Except models with uneven workload distributions, model `1394`, `wafer_stepper.1` and `lann6` also have big runtime differences. The runtime differences are caused by the number of generated vehicle states. Depending on the model, the percentage of new unexplored states in generated vehicle states per iteration is different for $h1$ and $h2$. A higher percentage will result in a fewer total exploration iterations, and shorter execution time per iteration. Hence, those models have big runtime differences. In average, the runtime of $h1$ is longer then that of $h2$ by 0.63 second.

Table 6.5: Runtime in seconds for the two hash functions

Model	Runtime $h1$	Runtime $h2$
cache	0.01	0.01
leader_election	0.07	0.07
acs	0.04	0.05
sieve_10	**0.4**	**0.46**
odp	0.05	0.06
1394	0.37	0.66
acs.1	0.20	0.14
transit	0.57	0.42
wafer_stepper.1	1.28	0.68
odp.1	0.75	0.93
1394.1	**1.75**	**3.07**
asyn3	**3.32**	**2.97**
lamport8	3.11	4.03
szymanski5	4.44	4.36
broadcast	15.69	18.5
peterson7	19.94	21.71
lann6	13.04	6.88
lann7	**17.08**	**11.39**
asyn3.1	**23.13**	**30.23**
gasstation11	36.24	31.23
gasstation12	145.85	136.25
Average	13.68	13.05

6.5.2 Runtime

This section includes the runtime results of GPUexplore-2-GPU for each optimization as well as their combinations, and then we compared the runtime of GPUexplore and GPUexplore-2-GPU with optimal optimizations.

Below is the list of optimization options of GPUexplore-2-GPU:

1. **Number of kernel iterations (K):** Number of iterations inside per kernel launch.
2. **Inconsecutive state transfer iterations (S):** Number of iterations to transfer vehicle states.
3. **Peer-to-peer communication:** Use peer-to-peer communication technology.
4. **State tracking:** Keep track of generated states.
5. **Unified buffer:** Use unified memory for the buffers.

Note that peer-to-peer communication and state tracking options are mutual exclusive and the unified buffer option need state tracking to work properly.

Number of kernel iterations

We conducted several experiments for GPUexplore-2-GPU to analyse the effect of K on different models. We vary K from 1 to 32 and measured the relative runtime to K = 1. The results are shown in Figure 6.2. When $K \leq 8$, all models except `odp` and `transit` achieve a better performance than that when $K = 1$. The reason for the speed-up is that a higher K will allow each exploration kernel to explore more states, which leads to a reduction of total exploration kernels. And because each exploration kernel is followed by a state transfer process, as the number of total exploration kernels decreases, the number of total state transfer kernels decreases. Thus, communication time decreases.

However, it can been seen that the runtime of some models start to increase when $K > 4$ or $K > 8$. It is because the writes to the global memory is not immediate visible to other threads, and they are guaranteed to be visible when the kernel is terminated. When K is large enough, there may be insufficient unexplored states in the global memory (many of them are still invisible). Then many blocks will fail to scan an unexplored state and wait for other blocks. In this case, a new exploration kernel can achieve higher performance than a new kernel iteration because the former one makes all writes to the global memory visible to other threads.

Because states in `odp` and `transit` are, to some extent, uniformly distributed in the hash table, the number of blocks participate in the writes to the global memory is large. As a results, the number of invisible writes is large, which causes the performance loss as K increases.

Inconsecutive state transfer iterations

We conducted several experiments for GPUexplore-2-GPU to analyse the effect of S on different models. We vary S from 1 to 32 and measured the relative runtime to S = 16. The results are shown in Figure 6.3. When varying S from 1 to 16, all models except `transit`, `odp`, `acs.1` and `wafer_stepper.1` have shorter runtime as S increases. When $S > 16$, several models will benefit a little from high S while others will have worse performance. The negative effect on the performance is the result of fewer state transfer processes. It could happen that at some points, there are insufficient unexplored states in the hash tables and many of them are in the buffers. At those situations, state transfer will lead to a higher utilization of GPU resources, which will speed-up the exploration despite the extra communication time introduced.

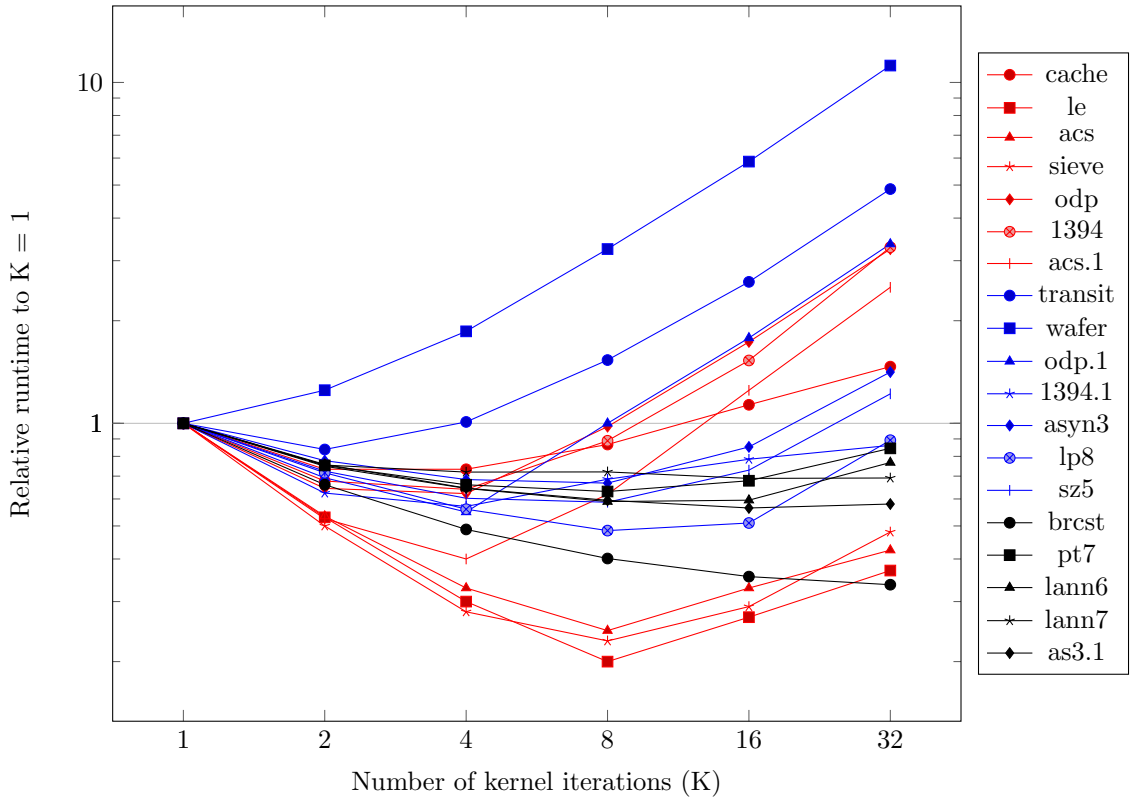

Figure 6.2: Runtime of GPUexplore-2-GPU with different kernel iterations (2560 blocks)

Model `transit` achieves best performance when S = 2 because inconsecutive state transfer reduces communication time. The performance gets worse as S increases. Because starting from S = 3, buffer will overflow (see Section 7.1.2). When a buffer overflow happens, the vehicle states will be stored in its own hash table, and they will be stored in the buffer at later iterations. This procedure will introduce at least one extra write and one extra read operations to the global memory.

Model `odp` and `acs.1` achieve best performance when $S = 8$. The performance becomes worse when $S > 8$ because of the buffer overflow problem. Model `wafer_stepper.1` achieves best performance when S = 1, and the performance remains the same when $S \geq 2$. One possible reason for the unchanging performance is because there is an exception for inconsecutive state transfer. The exception is: if at least one GPU does not have unexplored states in the hash table, a state transfer process is needed. Hence, it might be that this exception happens for every two exploration iterations for `wafer_stepper.1`, then the state transfer process happens for every two exploration iterations. As a result, the performance remains the same when $S \geq 2$.

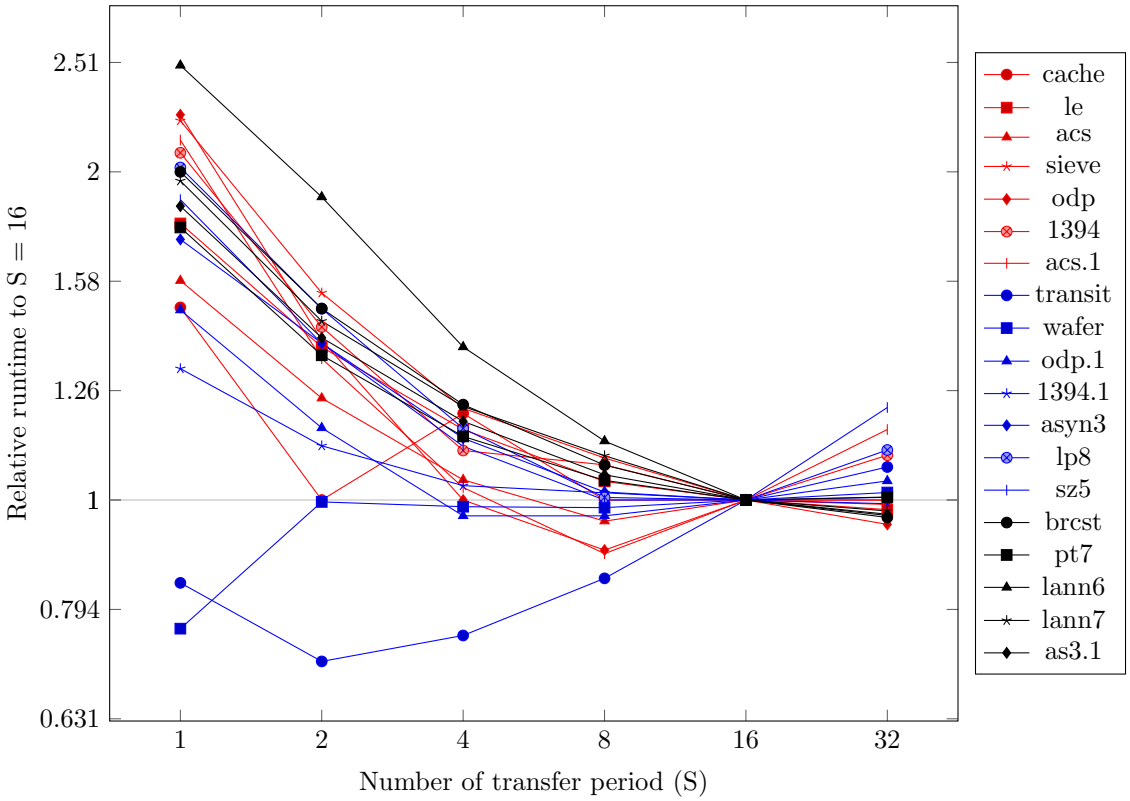

Figure 6.3: Runtime of GPUexplore-2-GPU with different inconsecutive state transfer iterations (2560 blocks)

Runtime comparison

Recall that the buffer overflow can lead to a worse performance and the chance for it increases as K or S increase. We need to make sure the combinations of K and S will not cause buffer overflow. The optimal combination of K and S that leads to the best performance is $K = 4$ and $S = 4$. The combination is the result of massive tests on all models and is only applicable to GTX 1080 using 40 MB for buffers.

Figure 6.4 shows the runtime results of GPUexplore-2-GPU with different optimization options. The experiments include the runtime for GPUexplore-2-GPU with no optimizations, only peer-

to-peer communication, only inconsecutive state transfer, inconsecutive state transfer with peer-to-peer communication, inconsecutive state transfer with state tracking, and unified buffer with state tracking optimizations.

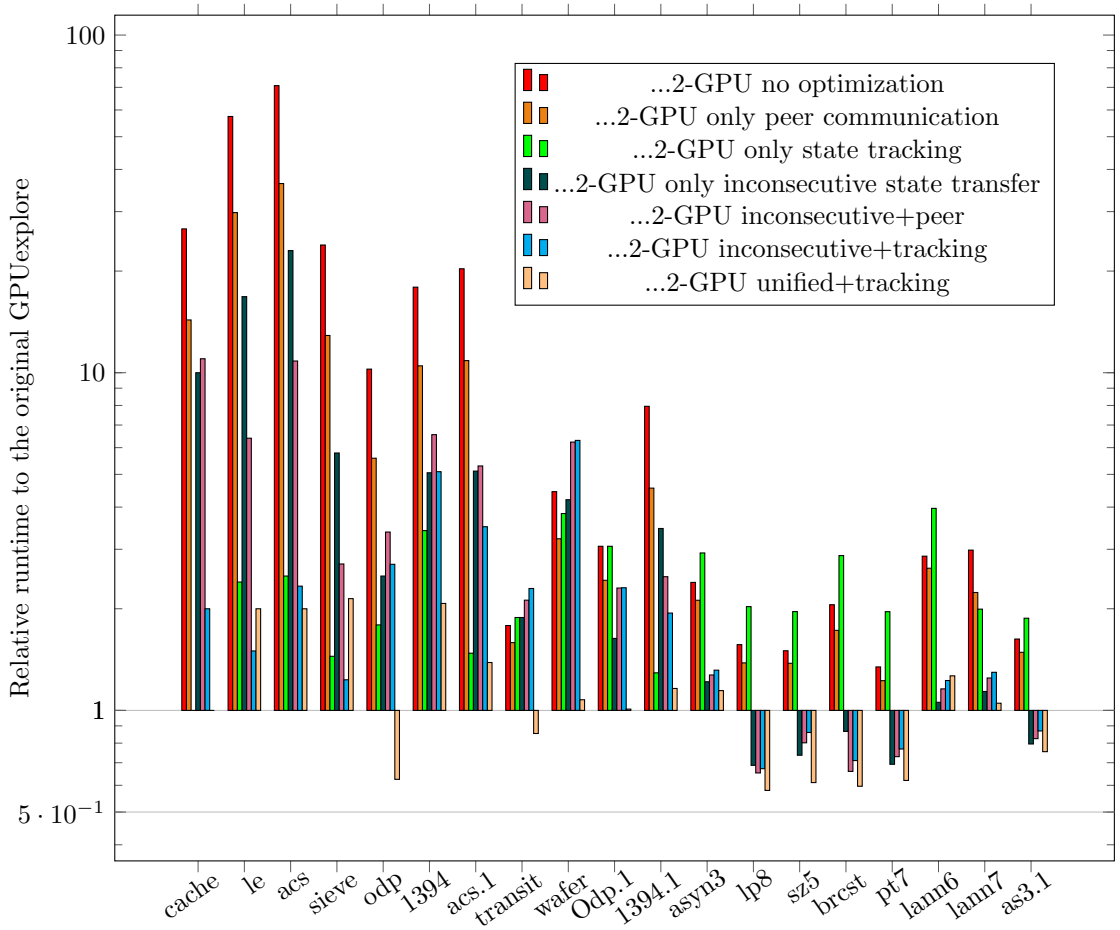

Figure 6.4: Runtime of GPUexplore and GPUexplore-2-GPU with different optimization options

When comparing the runtime of different optimization options for GPUexplore-2-GPU we can see that all optimizations and their combinations achieve better results than no optimizations. The unified buffer with state tracking always achieves best performance. Apart from the unified buffer, inconsecutive state transfer achieves best performance for models larger than `1394.1`; and state tracking tends to achieve best performance for models smaller than `asyn3`. Other optimizations and their combinations affect the performance differently depending on the model.

The runtime of GPUexplore-2-GPU and GPUexplore are shown in Table 6.6. In the table, we can observe that for GPUexplore-2-GPU, model `lamport8` has the greatest speed-up (1.73) and `1394` has the worst speed-up (0.48). In general, models smaller than `lamport8` (except `odp` and `transit`) has worse performance than GPUexplore. The reason is that when models are small, the utilization of GPU resources is low and more time are spent on communications. For models larger than `asyn3` (except `lann6` and `lann7`) some speed-ups can be observed. Model `lann6` has a worse performance because it generates too many vehicle states, whose number almost equals to the state-space of `lann6`; and `lann7` has a worse performance because the workload for one of the GPUs is around 90%, meaning the other GPU often remains idle.

In average GPUexplore-2-GPU has a 1.04 speed-up over GPUexplore. For models larger than `acs.1`, GPUexplore-2-GPU tends to have similar or shorter runtime.

Table 6.6: Runtime in seconds for GPUexplore and GPUexplore-2-GPU using unified buffer and state tracking optimizations

Model	Original GPUexplore	...2-GPU unified+tracking	Speed-up
cache	0.01	0.01	1.00
leader_election	0.03	0.07	0.50
acs	0.02	0.04	0.50
sieve_10	0.19	0.40	0.47
odp	0.08	0.05	1.60
1394	0.18	0.37	0.48
acs.1	0.15	0.20	0.72
transit	0.66	0.57	1.17
wafer_stepper.1	1.19	1.28	0.93
odp.1	0.75	0.75	0.99
1394.1	1.51	1.75	0.86
asyn3	2.90	3.32	0.87
lamport8	5.37	3.11	1.73
szymanski5	7.26	4.44	1.64
broadcast	26.31	15.69	1.68
peterson7	32.14	19.94	1.61
lann6	10.30	13.04	0.79
lann7	16.26	17.08	0.95
asyn3.1	30.64	23.13	1.32
Average			**1.04**

Chapter 7

Limitations and future work

This chapter explains the limitations of GPUexplore-2-GPU regarding the workload balancing, buffer overflow and full hash table. Directions for further optimizations of the tool as well as another research topic are suggested.

7.1 Limitations

One of the key aspects that is influencing the performance of GPUexplore-2-GPU is the workload balancing. The workload is determined by a hash function `h()` and the model itself.

7.1.1 Workload balance

According to the experiment results in Table 6.4, there exist some models that the number of explored states varies greatly for the GPUs. Many model checkers with distributed model checking method (using a static hash function) use a prime number to determine the destination of a state. This method is effective when the number of destination are often greater than two (more destinations lead to more even distributions). In our case, we only have 2 destinations. Because of the capacity of a machine, the number of destinations is unlikely to be greater than 4 when considering GPUs in one machine. Hence, the workload balance of GPUexplore-2-GPU highly depends on the model itself.

7.1.2 Buffer overflow

Recall the procedures when a thread tries to store a state, it first checks the occupancy of the destination bucket, if the bucket is full then the thread will try to store the state at next bucket which is determined by hashing the state by the hash functions stored in the shared memory.

When buffer overflow happens, the vehicle states failed to be stored in the buffer will be stored in the hash table. When these vehicle states succeeded to be stored in the buffer, they will be marked as old rather than be removed. If the states are removed, the removal of a state from the hash table will leave a hole at its location. This hole will threatens the correctness of duplication detection.

We give an example of a hole that threatens the correctness of duplication detection. Take a full bucket (bucket meant for state s) $B1$ with a vehicle state t, its next bucket $B2$ with state s store in $B2$. If a warp generated another s, it will first check if $s \in B1$. And because $B1$ is full, the warp will check if $s \in B2$. Then the warp will report duplication detected. However, if we remove t from $B1$, then $B1$ is not full. Then the next time a warp generates s, it will report $s \notin B1$, and then store s in $B1$. In this case, both $B1$ and $B2$ will have state s.

The example shows the removal of a state will influence the correctness of GPUexplore-2-GPU. The current mark-as-old method takes little memory (a few bytes in a GPU) if the buffer size is chosen properly.

One way to deal with these situations is to rearrange vehicle states in the hash table. When a warp tries to store a state in a full bucket with vehicle states, instead of storing the state in the next bucket, the warp can remove the vehicle states in the current bucket and store them in the next bucket. After that the warp can store the state in the current bucket.

We give an example of this rearranging mechanism. Take a full bucket (bucket meant for state s) $B1$ with a vehicle state t, its next bucket $B2$ with state s store in $B2$. If a warp generated another s, it will first remove t in $B1$ and store t in a bucket determined by the hash functions stored in the shared memory. And then the warp can store s in $B1$. This mechanism is leaved to future work.

7.1.3 Full hash table

GPUexplore-2-GPU will stop to explore if one of the GPUs has a full hash table. We investigated whether it is then possible to transfer all work to the other GPU. The result is if all work are transferred, we will have problems with duplication detection. Because the unified hash table method is considered infeasible at the moment, GPUs cannot directly access another GPU's hash table. As a result, when a GPU is full, it takes significant time to do duplication detection for each state belonging to that GPU. Moreover, duplication detection across GPUs need major changes to the application. Because of the thesis time period, we decided not to investigate cross GPU duplication detection.

7.1.4 Partial order reduction

Due to the limited time of the thesis period, the support for POR for GPUexplore-2-GPU is not provided. However, we believe that POR can be applied to GPUexplore-2-GPU straightforwardly. Because the destination of a state is determined, hence the states in a hash table will not be moved to another hash table. As a result, each GPU can apply POR on its own states independently.

7.2 Future work

This section points out some aspects that can improve the performance of GPUexplore-2-GPU, and a future research direction for a multi-machine GPUexplore.

7.2.1 Increase state transfer speed

The two GPUs in the development platform of the thesis are communicating via PCI-e 3.0 in x8 mode, which has a dual way bandwidth[1] of 15.8 GB/s. Some types of NVIDIA GPUs can also communicate with NVlink. NVlink 1.0 has dual way bandwidth of 160 GB/s and NVlink 2.0 has 300 GB/s. GPUexplore-2-GPU running on GPUs that communicate with NVlink can spend less time on communications.

7.2.2 Better GPU synchronization mechanism

In GPUexplore-2-GPU, GPUs synchronize at the end of each exploration iteration. In practice, there will always be a GPU idle, waiting for the other GPU to finish exploration. Investigations on asynchronous exploration kernel launches can be conducted to eliminate GPU idle time. Note that asynchronous exploration kernel launches need inconsecutive state transfer optimization applied.

[1] https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2

7.2.3 Hash function for destinations

The hash function for choosing the destination of a state can be improved. Current hash function only chooses destinations based on the raw value of a state. Recall that a state vector is a concatenation of LTS states. Hence, most of the time the raw values of successors of a state s hash to the same value as s.

It is observed from the experiments that lots of vehicle states are generated during the explorations. Sometimes the number of total vehicle states exceeds the number of total states in both GPUs. Hence, the hash function can be improved by addressing the similar raw values phenomenon to reduce the number of vehicle states.

Another approach is to using dynamic hash function to determine the destinations. In that way, we need to investigate the trade-off between the workload balance and the performance penalties due to the hashing.

7.2.4 Multi-machine GPUexplore

Further research on GPUexplore-2-GPU can be conducted on the multiple machine version. There are several research on the distributed state-space construction using distributed memory. Such as a distributed-memory version of SPIN [21] and a parallel version of Murϕ using message passing [12]. The approaches share a common idea: each machine in the network is responsible for exploring a part of the state-space, and using message passing to communicate. The differences locate at the data type used for storing states and the partitioning functions (a static or a dynamic one). Since GPUexplore-2-GPU already implemented the concept of distributed state-space exploration, we only need to implement message passing in order to develop multi-machine GPUexplore. And because CUDA has support for MPI[2], the implementation of multi-machine GPUexplore is straightforward.

[2]https://developer.nvidia.com/mpi-solutions-gpus

Chapter 8

Conclusions

In this thesis we solved the hash table inaccuracy problem and the inaccurate error reporting problem without introducing performance penalties. The GPUexplore will construct the complete state-space of a model if it is given enough memory, and reports full hash table error when it occurs.

Two approaches were discarded during the implementation of GPUexplore-2-GPU. Unified hash table approach was considered to be infeasible because the system wide atomic compare-and-swap function can slow down the application by 100 times. Simultaneous read/write FILO buffer was considered to be infeasible because the coherency of a unified variable inside a while-loop is not guaranteed. The feasible approach uses a array buffer to store states belong to other GPUs. The memory used for the buffer are typically 40MB in one GPU while the rest of the memory can be used for the hash table.

We showed that the average capacity increment of GPUexplore-2-GPU is 67% compared to GPUexplore. GPUexplore-2-GPU generally gives worse or similar runtime for smaller models. But it gives similar or better runtime for larger models. It can achieve a maximum of 1.73 speedup for larger models compared to GPUexplore. Four optimizations are available for GPUs with different compute capabilities, which optimizations are inspired by NVIDIA unified memory and peer accessing features. The implementation preserves all functionalities of GPUexplore except POR.

8.1 Answers to the research questions

RQ1. What modifications to the GPUexplore are needed so that it can runs on multiple GPUs?
Using multiple GPUs will changed the architecture of the system. A system with one GPU is a shared-memory architecture while a system with two or more GPUs is a distributed-memory architecture. Since GPUexplore is aiming at shared-memory architectures, we can either combine the memory of the GPUs into a shared memory and apply the same exploration method; or we can convert the exploration method that is aiming at shared memory architectures to that aims at distributed-memory architectures. Since the former approach proved to be infeasible at the moment, we need to convert the exploration method.

RQ2. How much capacity increment can multi-GPU GPUexplore provide compared to GPUexplore?
According to the occupancy test results in Table 6.3, we can see that the average occupancy is 95.95% for GPUexplore and 80.16% for GPUexplore-2-GPU. From these results we can calculate (see Section 6.4) the average capacity of GPUexplore-2-GPU is 67% more than GPUexplore.

RQ3. What can be done to improve the performance of multi-GPU GPUexplore?
Since the storing of states and successor generations (two main operations in model checking) of GPUexplore and GPUexplore-2-GPU are the same, the performance penalties of GPUexplore-2-GPU are all due to inter-GPU communications. The improvements are focusing on increasing data transfer speed and reducing data transfer size. Four optimizations (see Chapter 5) and their combinations are available for GPUexplore-2-GPU, which are peer-to-peer communication, keep track of generated states, unified buffer and inconsecutive state transfer.

RQ4. What would be the runtime difference between GPUexplore and the multi-GPU GPUexplore?
According to the runtime results in Figure 6.4 we can see that GPUexplore has shorter or similar runtime for models smaller than `lamport8` compared to GPUexplore-2-GPU with any optimizations. GPUexplore has longer runtime for models larger than `asyn3` (except `lann6` and `lann6`) compared to GPUexplore-2-GPU with at least two optimizations. In average, GPUexplore-2-GPU with optimal optimizations can achieve a 1.04 speed up compared to the original GPUexplore. Note that most of the speed-ups are be observed for larger models that takes a dozen of seconds to explore.

Main RQ: How to extend GPUexplore to use multiple GPUs so that it can construct larger state-space while maintaining or achieving better performance?

In order to develop such kind of GPUexplore we first need to changed the exploration method so that it can works on a distributed-memory many-core system (e.g. multiple GPUs). Then we need to minimize the memory used for this new exploration method to save more memory for the hash table (to construct larger state-space). At last, we need to reduce the communication time of GPUexplore-2-GPU with the intention of improving the performance (maintain or achieve better performance).

Bibliography

[1] A. Laarman, J. Pol and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011*, pages 506–511. Springer Verlag, 7 2011. 1, 2

[2] A. Wijs and D. Bošnački. Gpuexplore: Many-core on-the-fly state space exploration using gpus. *Springer Berlin Heidelberg*, pages 233–247, 2014. 2, 13

[3] A. Wijs and D. Bošnački. Many-core on-the-fly model checking of safety properties using GPUs. *International Journal on Software Tools for Technology Transfer*, 18:169–185, 2016. 1, 2, 7, 11, 13, 29

[4] A. Wijs and T. Neele and D. Bošnački. GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. *Springer International Publishing*, page 694701, 2016. 1, 2, 3, 13

[5] D. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. Owens, and N. Amenta. Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM. 2

[6] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill. Model checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis (ATVA 2017)*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017. 1, 2

[7] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine – a tool for distributed verification. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, pages 278–281, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 2

[8] E. Bartocci, R. DeFrancisco, and S. Smolka. Towards a gpgpu-parallel spin model checker. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 87–96, New York, NY, USA, 2014. ACM. 2

[9] M. Boukala and L. Petrucci. Distributed ctl model-checking and counterexample search. In *Proceedings of the Third International Conference on Verification and Evaluation of Computer and Communication Systems*, VECoS'09, pages 48–59, Swindon, UK, 2009. BCS Learning & Development Ltd. 10

[10] C. Baier and J. Katoen. *Principles of model checking.* MIT Press, 2008. 1

[11] S. Cranen, J. Groote, J. Keiren, F. Stappers, E. Vink, W. Wesselink, and T. Willemse. An overview of the mcrl2 toolset and its recent advances. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 29

[12] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*, pages 522–525, Oct 1992. 45

[13] E.M. Clarke and W. Klieber and M. Nováček and P. Zuliani. *Model Checking and the State Explosion Problem.* Springer-Verlag Berlin Heidelberg, 2012. 1

[14] F. Lang. Refined interfaces for compositional verification. *FORTE 2006*, 4229:159–174, 2006. 8

[15] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, Apr 2013. 29

[16] D. Heimbold and D. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985. 29

[17] G. Holzmann. *The Spin Model Checker: primer and reference manual.* AddisonWesley, 2004. 2

[18] G. Holzmann. Parallelizing the spin model checker. In *SPIN*, 2012. 1, 2

[19] J. Barnat and L. Brim and M. Češka and T. Lamr. CUDA Accelerated LTL Model Checking. *ICPADS 2009, Proceedings*, pages 34–41, 2009. 1

[20] J.F. Groote and M.R. Mousavi. *Modelling and Analysis of Communicating Systems.* Eindhoven University of Technology, 2013. 7, 8

[21] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 45

[22] S. Blom and J. Pol and M. Weber. Ltsmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, pages 354–359, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 2

[23] T. Neele and A. Wijs and D. Bošnački and J. Pol. Partial-Order Reduction for GPU Model Checking. *ATVA 2016, Proceedings*, pages 357–374, 2016. 2, 9

[24] Z. Wu and Y. Liu and J. Sun and J. Shi and S. Qin. GPU Accelerated On-the-Fly Reachability Checking. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 100–109, Dec 2015. 1, 3

[25] Z. Wu and Y. Liu and L. Yun and J. Sun. GPU Accelerated Counterexample Generation in LTL Model Checking. In *Formal Methods and Software Engineering*, pages 413–429, Cham, 2014. Springer International Publishing. 1, 2