Parallel Analysis of Parallelism

Verifying Concurrent Software System Designs Using GPUs

GTC 2015

Anton Wijs
Correctness of Concurrent Systems

- Distributed, concurrent systems common-place, but very difficult to develop
  - network applications, communication protocols, multi-threaded applications
- Systems may contain bugs such as deadlocks and livelocks
  - Deadlock: computation not finished, but system cannot progress
  - Livelock: system repeats computation steps without progressing
- Given a model of a concurrent system, these, and other functional properties can be checked using model checking
  - All states in which the system (design) can end up are inspected
  - It is automatic
  - Provides useful feedback (counter-examples)
Model Checking

- Exhaustively interpret all potential functional behaviour of a model of a (concurrent) system, and automatically check whether that behaviour meets a given specification
  - Deadlock freedom
  - Race freedom
  - ... safety and liveness properties

- Formal models describe **hardware or software designs**, requirements specified using **temporal logics** (*CTL, LTL, mu-calculus*)

Safety:
“two processes can never simultaneously access the same critical section”

Liveness:
“When a message has been sent, it is eventually received”

2007: pioneers **E.M. Clarke, J. Sifakis, E.A. Emerson** (early 80’s) receive **Turing award**
Model Checking

(Dining Philosophers Problem)

State Space is a map showing all possible system states and transitions between them

(Produced with the LTSview tool of the mCRL2 toolset)

[True] <True> True

(Dining Philosophers System can deadlock!)

(Deadlock freedom as mu-calculus formula)
Model Checking Success Stories

- **Deadlocks** detected in airline reservation systems
- Modern e-commerce protocols have been verified
- Studies of IEEE standards for in-house communication of appliances has led to **significant improvements**
- Errors found in *Deep Space 1 spacecraft controller* model (’98)
- **TU/e with mCRL2**: Control software of the Compact Muon Solenoid Experiment at the *Large Hadron Collider*: 27,500 finite state machines, **livelock** and **reachability bugs** found
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system
Drawback: state space explosion

Running example: Traffic light control system

27 states
Drawback: state space explosion

Running example: Traffic light control system

81 states
Drawback: state space explosion

Running example: Traffic light control system

13 traffic lights 1.59 million states
14 traffic lights 4.78 million states
15 traffic lights 14.35 million states
16 traffic lights 43.05 million states
17 traffic lights 129.14 million states

Linear growth of model leads to exponential growth of state space!

Current state-of-the-art (explicit-state) model checking: reason about ~ 3 billion states
Common operations in model checking:

- **Generating** state spaces (+ **on-the-fly checking properties**)
- **Analysing the structure** of states spaces (e.g., **strongly connected components**, relevant for more complex properties)
- Comparing states and transitions
  - **Minimising** state spaces for more efficient analysis

Can GPUs be used for this?

- Yes, but far from trivially
Harnessing the power of GPUs for model checking

Anton Wijs
Eindhoven University of Technology

On-The-Fly State Space Exploration

- Construct a state space, given a model of a concurrent system [3]
- New hash-table design for GPUs, with fine-grained parallelism
- Threads work in groups to generate state successors
- Block-local shared memory used for state caches

10-100x speedup

State Space Structural Analysis

- Decompose explicit graph into Strongly Connected Components
- Decomposition based on Forward/Backward Breadth-First Search
- Check equivalence of states for state space comparison and minimisation
- Both operations use a new pivot selection procedure for each region / block

10-79x speedup

Probability Computations

- Perform numerical computations for probabilistic model checking [1, 4]
- Solving systems of linear equations and performing matrix-vector multiplication
- Novel restructuring of input ensures coalesced memory access by threads

20-35x speedup

References

[1] Parallel Probabilistic Model Checking on General Purpose Graphics Processors
   D. Bosch, S. Edelkamp, D. Sulewski, and A.J. Wijs

[2] GPU Accelerated Strong and Branching Bisimilarity Checking
   A.J. Wijs
   in Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17), accepted for publication (2017)

[3] GPUexplore: Many-Core On-The-Fly State Space Exploration Using GPUs
   A.J. Wijs and D. Bošnački

[4] Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking
   A.J. Wijs and D. Bošnački

[5] GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components
   A.J. Wijs, J.-P. Katoen, and D. Bošnački

Tools available at http://www.win.tue.nl/~awijs
Harnessing the power of GPUs for model checking

Anton Wijs
Eindhoven University of Technology

On-The-Fly State Space Exploration

Construct a state space, given a model of a concurrent system [3]

New hash-table design for GPUs, with fine-grained parallelism

Elements are placed in buckets using warp-the-line technique

Threads work in groups to generate state successors

Parallelism at state-level

Block-local shared memory used for state caches

Local duplicate detection reduces global hash table access

Work forwarding per block from one search iteration to the next

Speed-up fetching new work for the next iteration

10-100x speedup

State Space Structural Analysis

Decompose explicit graph into Strongly Connected Components

Decompose graph of Markov Decision Process into Maximal End Components [5]

Decomposition based on Forward/Backward Breadth-First Search

Novel combined forward/backward thread kernel with trimming of trivial SCCs

Check equivalence of states for state space comparison and minimisation [2]

Efficiently checks strong and branching bisimilarity of states

Equivalence determined via many-core partition refinement

Both operations use a new pivot selection procedure for each region / block

10-79x speedup

Probability Computations

Perform numerical computations for probabilistic model checking [1, 4]

Solving systems of linear equations and performing matrix-vector multiplication

Parallel matrix-vector multiplication used in Jacobi method for solving equation systems

Parallel termination checking achieves significant speedup

Fast checking if next iteration is needed

Stochastic states are grouped in segments of 16 and 32 states

Covers a half and a full warp of threads

20-35x speedup

References

[1] Parallel Probabilistic Model Checking on General Purpose Graphics Processors
D. Bošnački, S. Edelkamp, D. Sulewski, and A.J. Wijs

[2] GPU Accelerated Strong and Branching Bisimilarity Checking
A.J. Wijs
in Proceedings of the 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’13), accepted for publication (2015)

[3] GPUexplore: Many-Core On-The-Fly State Space Exploration Using GPUs
A.J. Wijs and S. Edelkamp

[4] Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking
A.J. Wijs and D. Bošnački

[5] GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components
A.J. Wijs, J.-P. Katoen, and D. Bošnački

Tools available at http://www.win.tue.nl/~awijs

The central image in “State Space Structural Analysis” shows the state space of a Bounded Retransmission Protocol model, and was created using the LTSview tool of the mCRL2 toolset (http://www.mcrl2.org)
State Space Generation

- **Graph traversal** is a very important operation
  - Much work on GPU graph traversal (also at GTC 2015)
- **However**, for model checking, many approaches are not suitable, since the graph (state space) is not known *a priori*
  - Number of states and transitions *not known*
- Traffic light system with a pedestrian process:

  - **Key aspects:**
    - Next-state computation (compute new state vectors)
    - Keeping track of which state vectors have been visited / explored
Model encoding

- In addition: synchronisation rules are encoded as bit sequences

**Process LTSs**
- ProcOffsets: \(\ldots 67\)
- StateOffsets: \(\ldots 201, 206\)
- TransArray: \(\ldots t_1, \ldots, t_6\)

**State vector**
- \(s[4] \quad s[3] \quad s[2] \quad s[1]\)
- \(\ldots 101\)

**Transition**
- \(T_{s_2} \quad T_{s_1} \quad T_{s_0} \quad T_{aT_s}\)

**Input** has a known size, and never changes: can be stored in **texture memory**
• Block fetches **unexplored vectors** from global to shared memory
• Threads are placed in *groups* of size \( n \) (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are *immediately processed*
• *For synchronisations*: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory
• Threads are placed in groups of size $n$ (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are **immediately processed**
• *For synchronisations*: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory
• Threads are placed in *groups* of size $n$ (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are *immediately processed*
• *For synchronisations*: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory
• Threads are placed in *groups* of size $n$ (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are *immediately processed*
• *For synchronisations*: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory

• Threads are placed in **groups** of size $n$ (= state vector length)

• Each thread fetches transition entries of its process / state
  - independent transitions are **immediately processed**

• **For synchronisations**: all transitions of next label are fetched, group leader manages progress
- Block fetches unexplored vectors from global to shared memory
- Threads are placed in groups of size $n$ (= state vector length)
- Each thread fetches transition entries of its process / state
  - independent transitions are immediately processed
- For synchronisations: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory
• Threads are placed in *groups* of size $n$ (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are *immediately processed*
• For *synchronisations*: all transitions of next label are fetched, group leader manages progress
• Block fetches unexplored vectors from global to shared memory
• Threads are placed in groups of size $n$ (= state vector length)
• Each thread fetches transition entries of its process / state
  - independent transitions are immediately processed
• For synchronisations: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory
• Threads are placed in **groups** of size \( n \) (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are **immediately processed**
• For synchronisations: all transitions of next label are fetched, group leader manages progress
• Block fetches **unexplored vectors** from global to shared memory

• Threads are placed in *groups* of size $n$ (= state vector length)

• Each thread fetches transition entries of its process / state
  
  • independent transitions are *immediately processed*

• For *synchronisations*: all transitions of next label are fetched, group leader manages progress

---

**Diagram Description:**

- **Global Memory:** Block fetches unexplored vectors from global to shared memory.

- **Thread Groups:** Threads are placed in groups of size $n$.

- **Transition Processing:** Each thread fetches transition entries of its process/state.
  
  - Independent transitions are immediately processed.

- **Synchronisation:** For synchronisations, all transitions of the next label are fetched.
  
  - Group leader manages progress.

---

**Diagram Nodes:**

- **R:** Red
- **G:** Green
- **Y:** Yellow

**Transitions:**

- **Delay:** Delay
- **Stop:** Stop
- **Cross:** Cross
- **Approach:** Approach
- **Wait:** Wait

**Labels:**

- **<R,3>...**

**Notes:**

- Store lists of reachable states via transitions labelled “cross” and combine.
• Block fetches unexplored vectors from global to shared memory
• Threads are placed in groups of size $n$ (= state vector length)
• Each thread fetches transition entries of its process / state
  • independent transitions are immediately processed
• For synchronisations: all transitions of next label are fetched, group leader manages progress

Store lists of reachable states via transitions labelled “cross” and combine

$<R,3>$ ...

$<G, 1>$
Property checking

• Add another automaton to the model network representing the property
• Example: mutual exclusion property
State storage

In a warp, random memory access \(\Rightarrow\) bad for performance

Worse when elements consist of >1 integers

Cuckoo hashing on GPUs (Alcantara et al.):
- moving around of elements may lead to duplicate entries
- Drastically more so when element insertion and lookup is not atomic
- Need of another hash table design
State storage

- Hash table with linear probing
- Buckets of 32 integers fits in cache line
- Scanning bucket content in parallel
  - warp-the-line (nod to walk-the-line [Laarman et al.,’10])
State storage

- Hash table with linear probing
- Buckets of 32 integers fits in cache line
- Scanning bucket content in parallel
  - **warp-the-line** (nod to walk-the-line [Laarman et al.,’10])

Assumes vector size < 32, limitation can be overcome
State storage

- Hash table with linear probing
- Buckets of 32 integers fits in cache line
- Scanning bucket content in parallel
  - **warp-the-line** (nod to walk-the-line [Laarman et al.,’10])

Assumes vector size < 32, limitation can be overcome

But if groups of $n$ threads generate a vector, how to employ 32 threads for storing it?
State storage

- Shared memory hash table used for temporary storage
  - block-local partial duplicate detection
State storage

32 32 32 32 32

19
State storage

32  32  32  32  32
State storage

- Warp scans shared memory
- Warp stores new vectors in buckets
State storage

- Warp scans shared memory
- Warp stores new vectors in buckets
• Warp scans shared memory
• Warp stores new vectors in buckets
Data races

• For vectors in multiple integers
  • Warp W1 can be writing vector \( v \) while warp W2 reads
• False positives
  • W2 concludes that \( v \) is not in hash table
• However: results in redundant work, not in ignoring states
  • On average 2% redundant work
State retrieval

- Global hash table also serves for state retrieval
  - Requires scanning hash table for work
- Work claiming:
  - When a group generates new vector, it is claimed by block for next iteration
Model representation

Next-state computation

Block-local state caches

Global hash table

Multiprocessor 1

- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP

Shared memory

128B

L1 & L2 cache

Texture cache

Global memory

Multiprocessor $N$

- SP
- SP
- SP
- SP
- SP
- SP
- SP
- SP

Shared memory

128B
Parameter experiments - blocks
Runtimes - property checking
Further material

• **GPUexplore, GPUdecompose, GPUreduce** tools online
  
  • [http://www.win.tue.nl/~awijs/software.html](http://www.win.tue.nl/~awijs/software.html)

• **Publications Model Checking & GPUs:**


• **Poster P5185 - Harnessing the Power of GPUs for Model Checking**
Structure of the talk

• Automatic formal verification: what is it and why use it?
  • *State space generation and analysis*
• GEM Toolbox: Model Checking on GPUs
  • What does it offer?
  • How is it implemented?
    • Range of techniques specifically designed for state space structures
  • What speedups can it achieve?
Dining Philosophers Problem

- 5 Philosophers at a dining table
- A philosopher needs two forks to eat (on the right and left)
- *Can a philosopher starve?*
- *Can all philosophers starve?*

- Try out possibilities or ...

- Make a formal specification of the situation (what is there and what can happen?)
- Automatically check all possible events and states of the system
- *Model checking*
- Allows you to check all kinds of properties
• *State space*: involves all possible *states* of system, and *transitions* between those states

• Image of the state space of a **Bounded Retransmission Protocol** model

• Model checking can guarantee that a system is correct or can reach undesired states (*the dining philosophers can starve*)

• But...

• Model checking is computationally very demanding, due to *state space explosion problem*
  
  • Linear growth of model tends to lead to exponential growth of state space