More CUDA Accelerated LTL Model Checking

Jiří Barnat, Petr Bauch, Luboš Brim, and **Milan Češka**

Faculty of Informatics, Masaryk University
Brno, Czech Republic

D3S Seminar based on

CUDA accelerated LTL Model Checking
[Barnat et al. ICPADS'09]

DiVinE-CUDA – A Tool for GPU Accelerated LTL Model Checking
[Barnat et al. PDMC'09]

Employing Multiple CUDA Devices to Accelerate LTL Model Checking
[Barnat et al. ICPADS'10]

CUDA Accelerated LTL Model Checking – Revisited
[Bauch et al. MEMICS'10]

ParaDiSe
Parallel & Distributed
Systems Laboratory

18.1, 2011

1. Introduction
   - Motivation and Contribution
   - LTL Model Checking
2. Redesign of the Maximal Accepting Predecessor Algorithm
   - Parallelization of MAP algorithm
   - Reformulation of MAP algorithm
3. Parallel Construction of CSR Representation
4. Overcoming Memory Limitation
   - Partitioning of CSR representation
   - Multiple CUDA MAP algorithm
5. Redesign of One-Way-Catch-Them-Young Algorithm
   - Reformulation of OWCTY Algorithm
6. Experimental Evaluation
7. Conclusion and Future work

Motivation

Formal verification

- critical system - any mistake may have fatal consequences
- testing is insufficient - can not guarantee correctness
- formal methods can prove or disprove correctness of the system

Motivation

Formal verification

- critical system - any mistake may have fatal consequences
- testing is insufficient - can not guarantee correctness
- formal methods can prove or disprove correctness of the system

Model Checking

- fully automated approach to the formal verification
- state space explosion problem
- possible solution:
  - symbolic representation
  - reduction techniques
  - **platform-dependent verification**

ParaDiSe Parallel & Distributed Systems Laboratory

Platform-dependent Verification

DiVinE

- Explicit Parallel LTL Model Checker
- Focuses on full utilization of available HW power

Platform-dependent Verification

DiVinE

- Explicit Parallel LTL Model Checker
- Focuses on full utilization of available HW power

Successful stories on the following parallel platforms

- Clusters, Grids
  DiVinE Cluster
- Multi-core workstations
  DiVinE Multi-Core
- Cluster of Multi-core workstations
  DiVinE 2.x

ParaDiSe Parallel & Distributed Systems Laboratory

Platform-dependent Verification

DiVinE

- Explicit Parallel LTL Model Checker
- Focuses on full utilization of available HW power

Successful stories on the following parallel platforms

- Clusters, Grids
  DiVinE Cluster
- Multi-core workstations
  DiVinE Multi-Core
- Cluster of Multi-core workstations
  DiVinE 2.x

Many-core architectures

- Shared-memory setting, many computing cores
- Parallel computing platform of the future?
- Widely accessible due to GP GPU devices

ParaDiSe
Parallel & Distributed Systems Laboratory

Many-core Architecture

NVIDIA CUDA Technology

- Many-core architecture
- Hundreds of computing cores
- HW support for thousands of computing threads
- Requires quite specific memory-usage pattern
- Incompatible with random memory access (hashing)
- Limited size of the GPU memory

ParaDiSe
Parallel & Distributed
Systems Laboratory

Many-core Architecture

NVIDIA CUDA Technology

- Many-core architecture
- Hundreds of computing cores
- HW support for thousands of computing threads
- Requires quite specific memory-usage pattern
- Incompatible with random memory access (hashing)
- Limited size of the GPU memory

Need for new many-core algorithms

- Straightforward adaptation of distributed and multi-core algorithms to many-core architecture is inefficient
- Necessity of expensive phase of construction of data structures
- Partitioning of data structures for multiple device computation
- Also the case of parallel LTL Model Checking

ParaDiSe Parallel & Distributed Systems Laboratory

CUDA Accelerated LTL Model Checking

Previous solution

- successful redesign of MAP algorithm – significant GPU acceleration [Barnat et al. ICPADS'09]
- DiVinE-CUDA – tool for CUDA Accelerated LTL Model Checking [Barnat et al. PDMC'09]

Two weaknesses of our approach

- expensive phase of encoding the state space into the suitable representation
- limited to the middle-size instances that can fit the memory of a single CUDA device

Solution of the weaknesses

[Barnat et al. ICPADS'10]

- multi-core acceleration of expensive encoding the state space into the CSR representation
- overcoming of single GPU memory limitations
- verification of much larger model checking problems
- preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm

Solution of the weaknesses

[Barnat et al. ICPADS'10]

- multi-core acceleration of expensive encoding the state space into the CSR representation
- overcoming of single GPU memory limitations
- verification of much larger model checking problems
- preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm

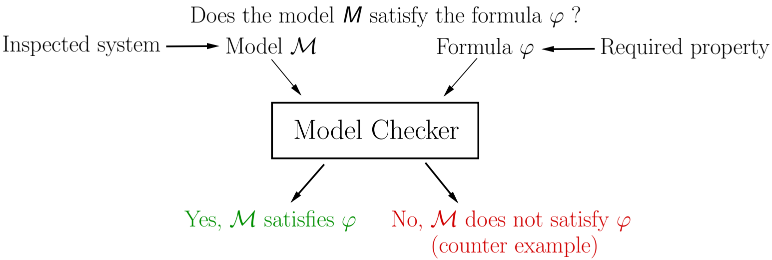
[Bauch et al MEMICS'10]

- redesign of the OWCTY algorithm
- significantly outperform the original CUDA MAP algorithm
- robust to improper ordering in the input representation

ParaDiSe Parallel & Distributed Systems Laboratory

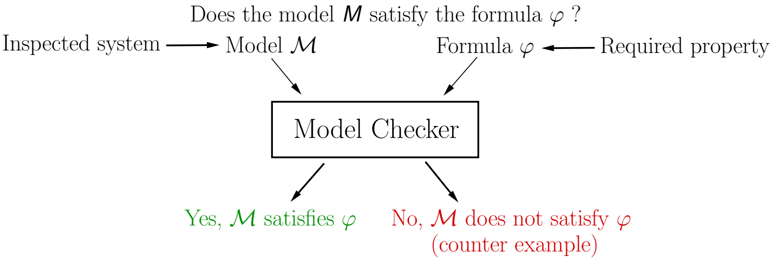
LTL Model Checking

Model Checking



LTL Model Checking

Model Checking

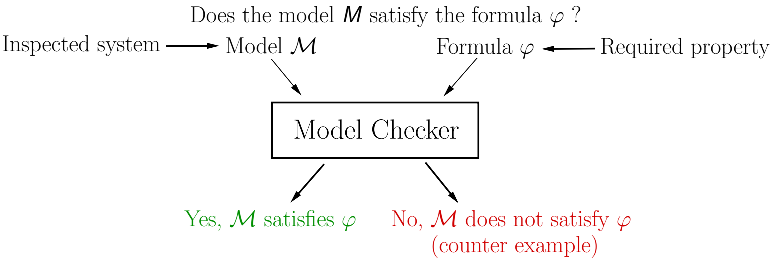


LTL Model Checking

- required property $\rightarrow$ formula in Linear Temporal Logic (LTL)
  - examples: $\varphi = \mathrm{FG}(p)$ – from certain moment p always holds
    $\varphi = \mathrm{G}(p \Rightarrow \mathrm{F}(q))$ – response

LTL Model Checking

Model Checking



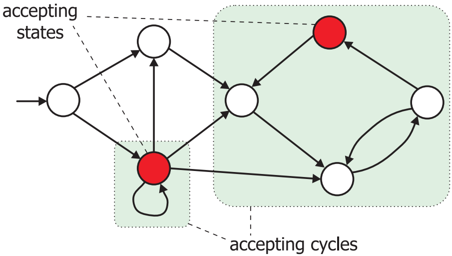
LTL Model Checking

- required property $\rightarrow$ formula in Linear Temporal Logic (LTL)
  - examples: $\varphi = \mathrm{F}\mathrm{G}(p)$ – from certain moment p always holds
    $\varphi = \mathrm{G}(p \Rightarrow \mathrm{F}(q))$ – response

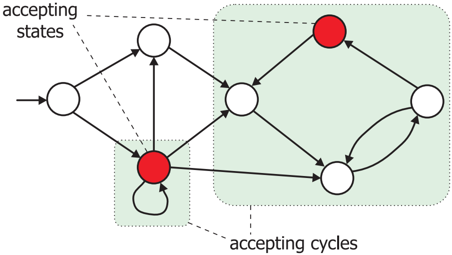
Solution

- automata based approach
- reduction on **accepting cycle detection**

Algorithms for Accepting Cycle Detection



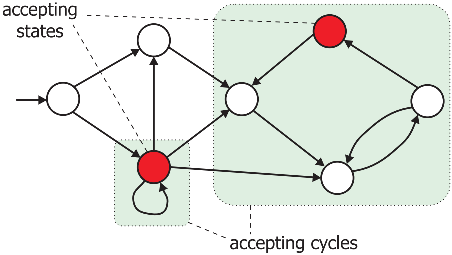
Algorithms for Accepting Cycle Detection



Optimal time complexity but hard to parallelize:

- Nested DFS
- Tarjan's SCC decomposition

Algorithms for Accepting Cycle Detection



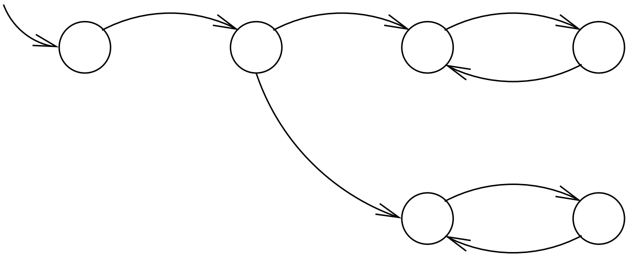
Optimal time complexity but hard to parallelize:

- Nested DFS
- Tarjan's SCC decomposition

Unoptimal time complexity but easy to parallelize:

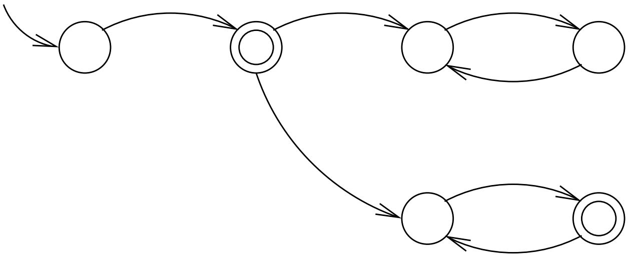
- One-Way-Catch-Them-Young (OWCTY)
- Maximal accepting predecessor (MAP)

Algorithm MAP



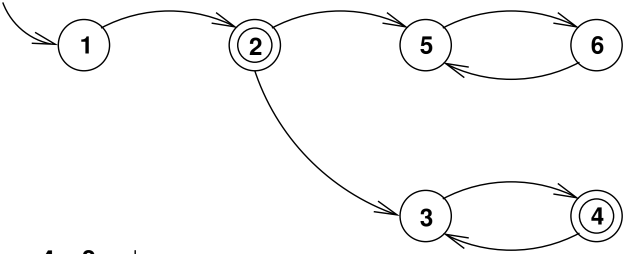
Graph corresponding to the state space.

Algorithm MAP



Accepting vertices, accepting cycle.

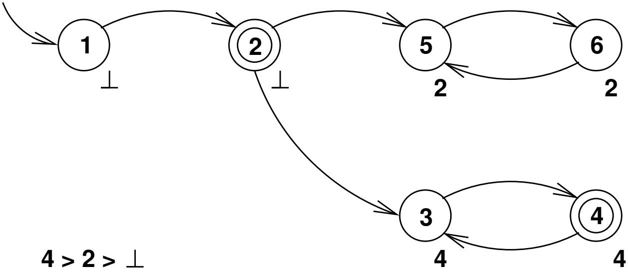
Algorithm MAP



4 > 2 > $\perp$

Vertex ordering.

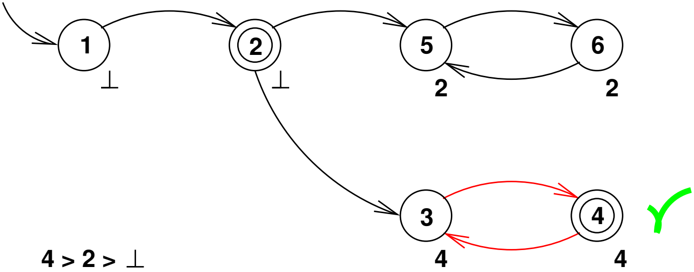
Algorithm MAP



Maximal Accepting Predecessor (MAP)

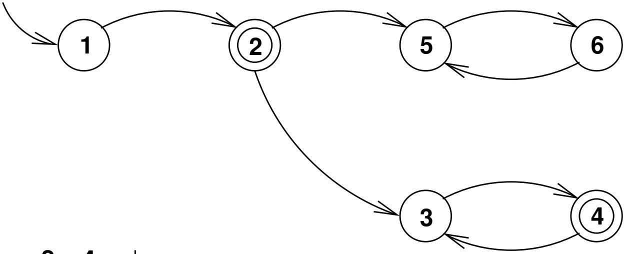
$$map(v) = \max\{\perp, u \mid (u, v) \in E^+ \wedge \mathcal{A}(u)\}$$

Algorithm MAP



$$map(v) = v \quad \Longrightarrow \quad \text{accepting cycle}$$

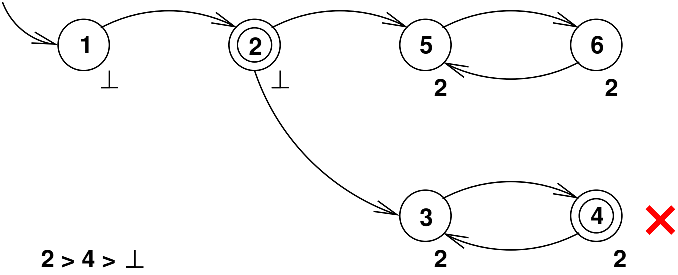
Algorithm MAP



2 > 4 > ⊥

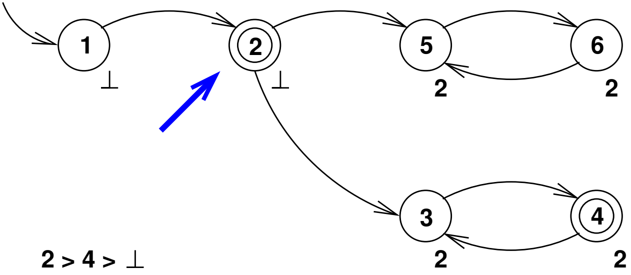
What if $2 > 4$?

Algorithm MAP



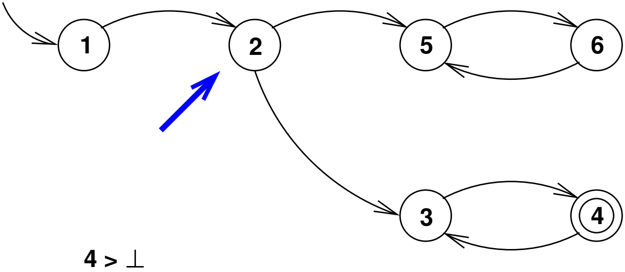
Accepting cycle undetected.

Algorithm MAP



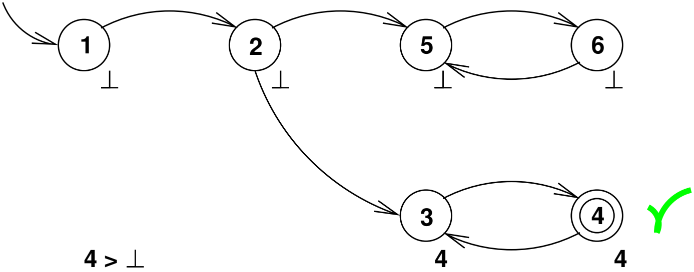
If no accepting cycle is found, then maximal accepting vertices cannot be part of an accepting cycle.

Algorithm MAP



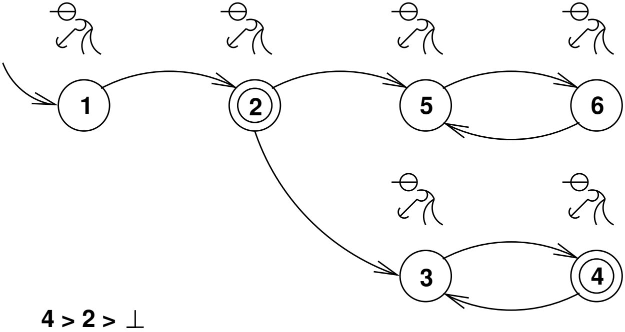
Maximal accepting vertices marked as non-accepting.

Algorithm MAP



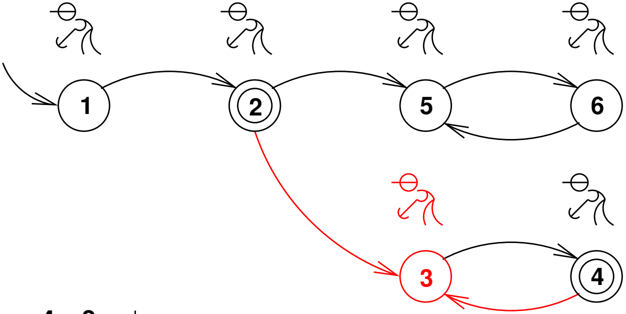
Repeat until accepting cycle is found or there are no accepting vertices.

Computing Values of MAP – Many-cores



One thread per vertex.

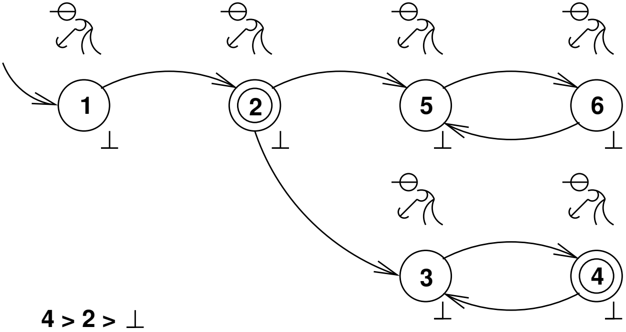
Computing Values of MAP – Many-cores



4 > 2 > $\perp$

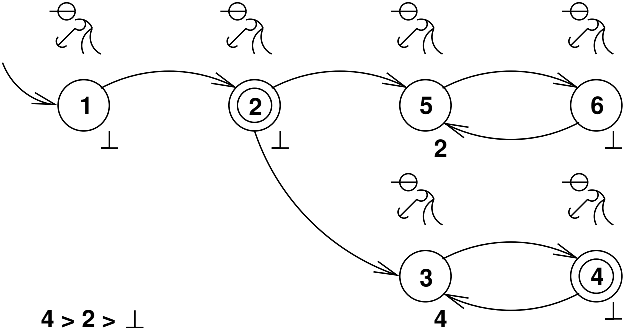
Each thread processes all incoming edges.

Computing Values of MAP – Many-cores



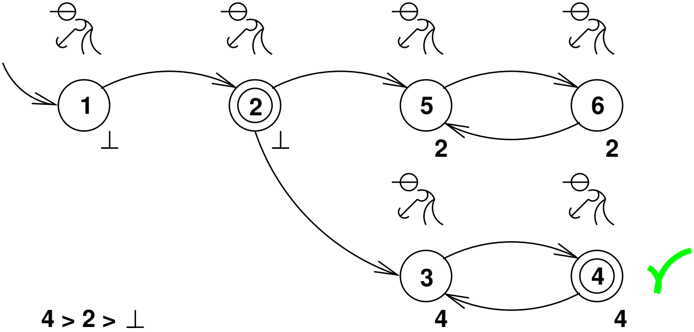
Threads proceed simultaneously.

Computing Values of MAP – Many-cores



Threads proceed simultaneously.

Computing Values of MAP – Many-cores

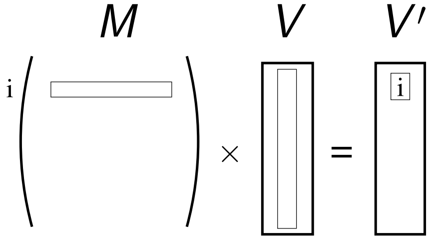


Accepting cycle found.

MAP Algorithm as Matrix-Vector Product

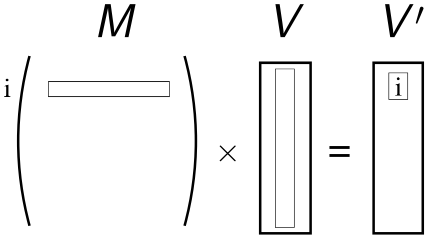
$$\overset{M}{\begin{pmatrix} \quad & \quad & \quad \\ & & \\ & & \end{pmatrix}} \times \overset{V}{\boxed{}} = \overset{V'}{\boxed{}}$$

MAP Algorithm as Matrix-Vector Product



$$V'[i] = \sum_{0\leq j\leq n} M[i][j] \cdot V[j]$$

MAP Algorithm as Matrix-Vector Product



$$V'[i] = \sum_{0\leq j\leq n} M[i][j] \cdot V[j]$$

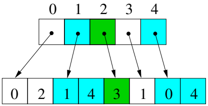
$$V'[i] = \max_{0\leq j\leq n} M[i][j] \cdot \mathrm{maxacc(V[j],i)}$$

$$maxacc(u,v) = \begin{cases} \max\{map(u), map(v), u\} & \text{if } \mathcal{A}(u) \\ \max\{map(u), map(v)\} & \text{otherwise.} \end{cases}$$

Two Weaknesses of our Approach

1. **Expensive construction of the CSR representation**
   - handling full matrices of predecessors is memory inefficient
   - Compact Sparse Row (CSR) representation of the graph

$$\begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$



Mc=(0 2 1 4 3 1 0 4)
Mr=(0 2 4 5 6)

   - CSR construction takes 93% of total verification time

Parallel Construction of CSR Representation

Basic idea

1. Multi-core parallel state space generation
   - hash-based partitioning of the graph
   - local storage for local vertices
   - non-local vertices are hand out to the owning threads
   - vertices exchange – contention and lock-free queue structures

Parallel Construction of CSR Representation

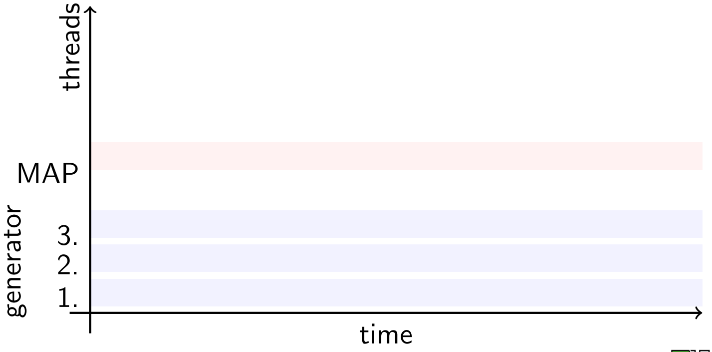
Basic idea

1. Multi-core parallel state space generation
   - hash-based partitioning of the graph
   - local storage for local vertices
   - non-local vertices are hand out to the owning threads
   - vertices exchange – contention and lock-free queue structures
2. Concurrent parallel construction of CSR representation
   - computation of a unique integer number between 1 and $|V|$ for each vertex
   - when a cross transition is generated and stored to the CSR representation the number of target vertex is unknown
   - avoiding of multiple state space traversal

DiVinE-CUDA Workflow

On-the-fly model checking computation

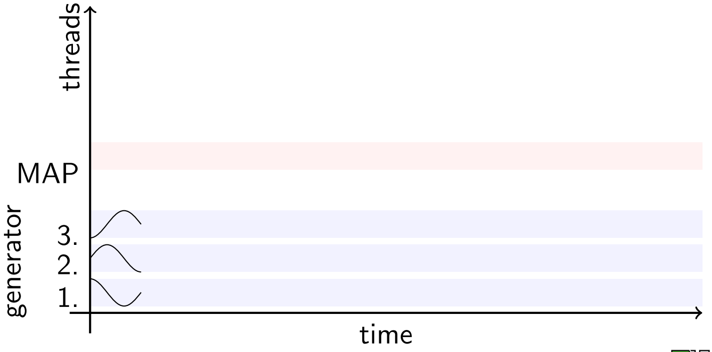
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

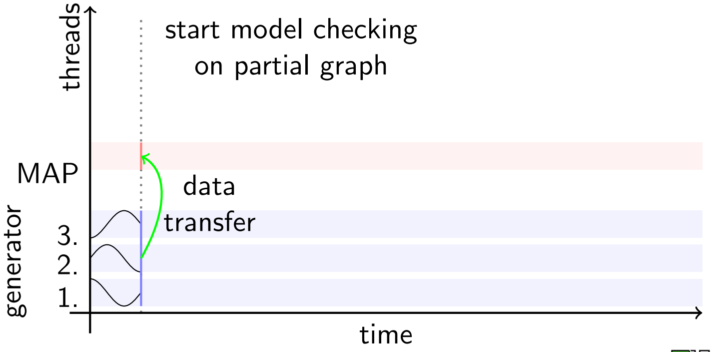
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

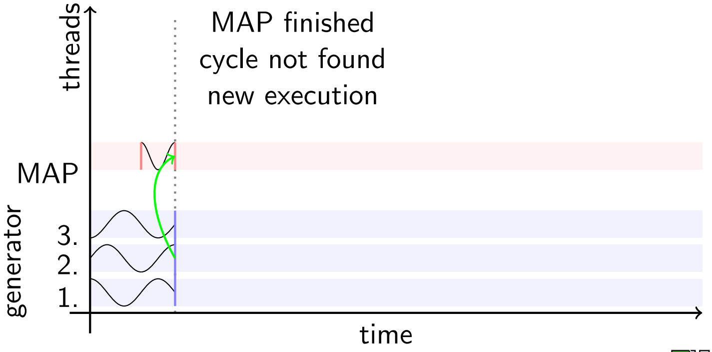
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

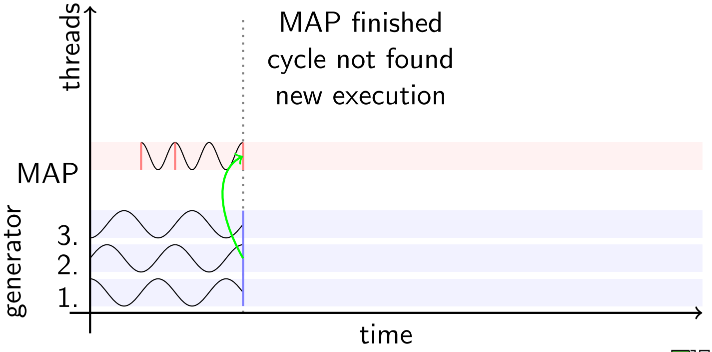
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

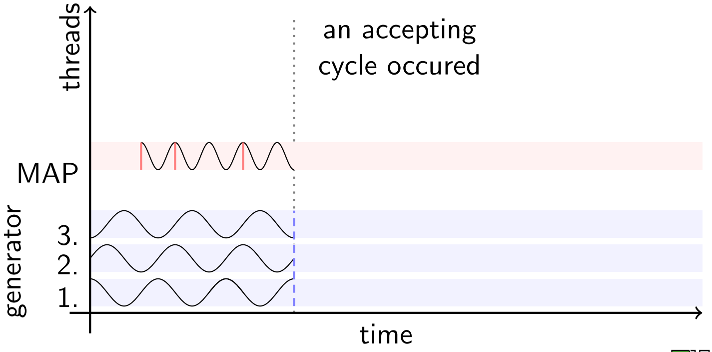
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

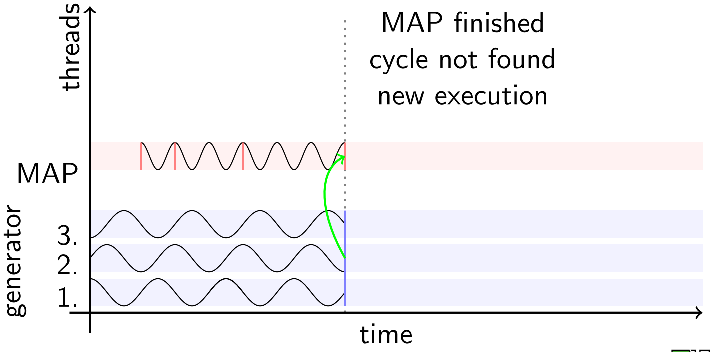
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

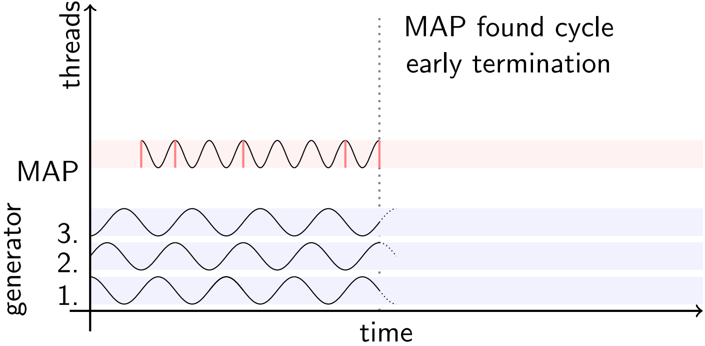
- multi-core state space generation
- one core oversees the communication with GPU device



DiVinE-CUDA Workflow

On-the-fly model checking computation

- multi-core state space generation
- one core oversees the communication with GPU device

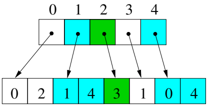


Two Weaknesses of our Approach

2. Limited only to middle-size model checking problems

- CSR representation + MAP vectors has to fit to GPU memory

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$



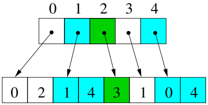
Mc=(0 2 1 4 3 1 0 4)
Mr=(0 2 4 5 6)

Two Weaknesses of our Approach

2. Limited only to middle-size model checking problems

- CSR representation + MAP vectors has to fit to GPU memory

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$



Mc=(0 2 1 4 3 1 0 4)
Mr=(0 2 4 5 6)

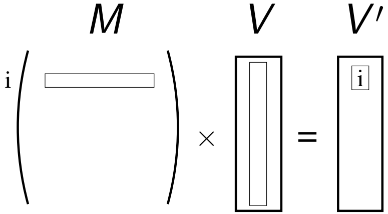
- GPU Memory consumption
  - Data stored on GPU
    - 12B per vertex (4B due to CSR + 8B due to MAP vectors)
    - 4B per edge (CSR)
  - 1 GB of GPU memory
    - 30 M of vertices, 150 M of edges (avg. outdegree 5)
    - 50 M of vertices, 100 M of edges (avg. outdegree 2)

ParaDiSe Parallel & Distributed Systems Laboratory

Employing Multiple GPU to Overcome Memory Limitation

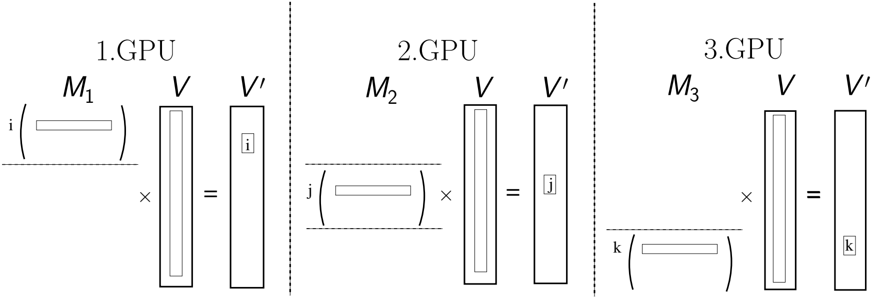
Basic idea

- verification of the problems that fit the aggregate memory of multiple GPU devices
- split and distribute two data structures
  1. CSR representation of the graph
  2. vector of values associated with individual vertices

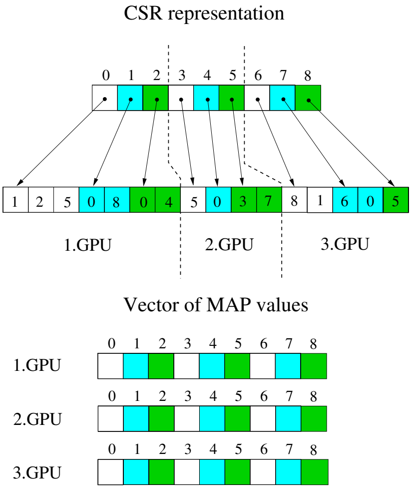


Two Proposed Partitioning

1. **Only the CSR representation of the graph is partitioned**
   - every GPU device keeps:
     - one part of the CSR representation of the graph
     - complete vector of values associated to individual vertices

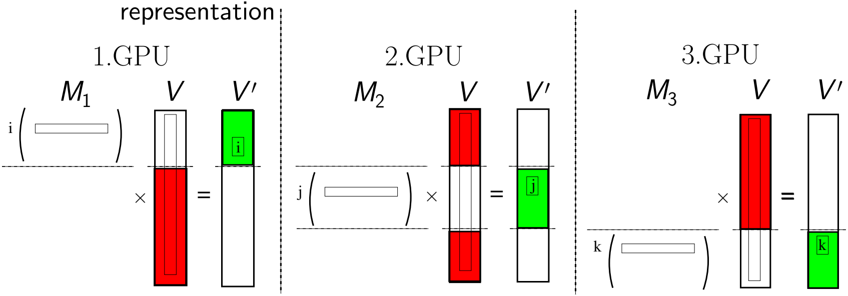


Two Proposed Partitioning

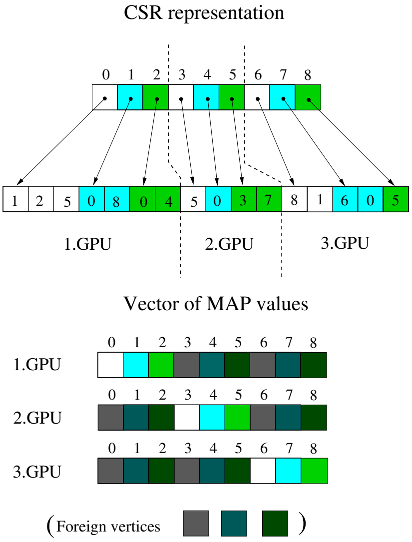


Two Proposed Partitioning

1. **Only the CSR representation of the graph is partitioned**
   - every GPU device keeps:
     - one part of the CSR representation of the graph
     - complete vector of values associated to individual vertices
   - foreign vertices – all vertices stored in foreign parts of the CSR representation



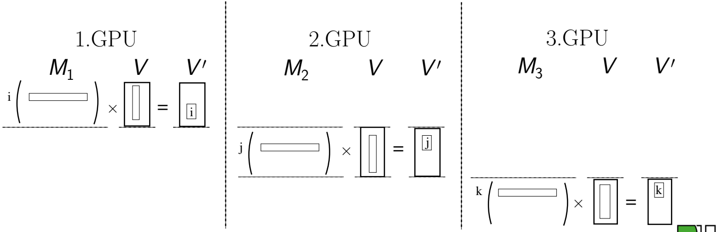
Two Proposed Partitioning



ParaDiSe Parallel & Distributed Systems Laboratory

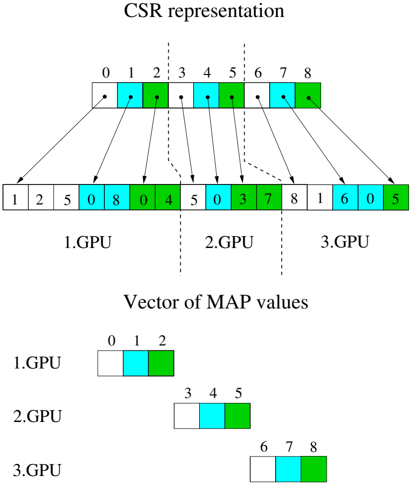
Two Proposed Partitioning

1. **Only the CSR representation of the graph is partitioned**
   - every GPU device keeps:
     - one part of the CSR representation of the graph
     - complete vector of values associated to individual vertices
2. **Also vector of values are partitioned**
   - every GPU device keeps reduced vector
     - contains the values for all vertices that appear in the local CSR representation part



ParaDiSe Parallel & Distributed Systems Laboratory

Two Proposed Partitioning

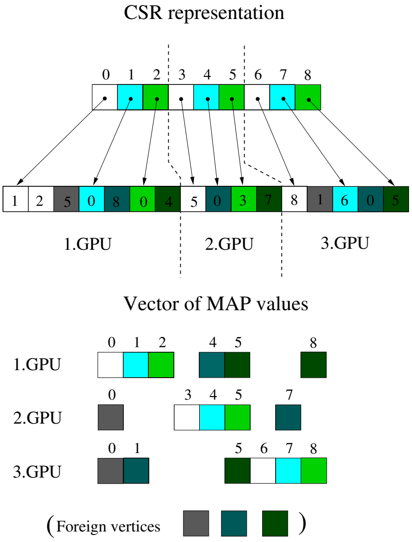


Two Proposed Partitioning

1. **Only the CSR representation of the graph is partitioned**
   - every GPU device keeps:
     - one part of the CSR representation of the graph
     - complete vector of values associated to individual vertices
   - foreign vertices – all vertices stored in foreign parts of the CSR representation
2. **Also vector of values are partitioned**
   - every GPU device keeps reduced vector
     - contains the values for all vertices that appear in the local CSR representation part
   - every GPU device keeps foreign vertices
     - target vertices of cross edges whose source vertices are stored in the local CSR representation part

ParaDiSe Parallel & Distributed Systems Laboratory

Two Proposed Partitioning

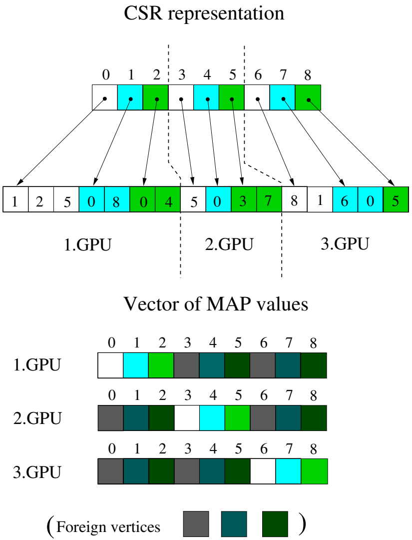


Preparing Foreign Vertices Vectors

Synchronisation during MAP computation

- requires to exchange the values of the foreign vertices
- communication among GPU devices is realized through the host memory – maintains the complete vector of values
- first partitioning approach – efficient sequential read/write

Preparing Foreign Vertices Vectors

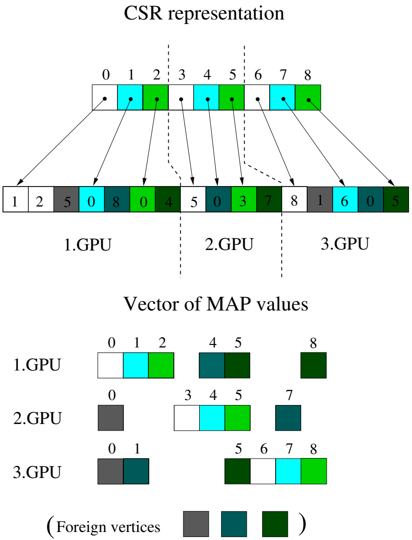


Preparing Foreign Vertices Vectors

Synchronisation during MAP computation

- requires to exchange the values of the foreign vertices
- communication among GPU devices is realized through the host memory – maintains the complete vector of values
- first partitioning approach – efficient sequential read/write
- second partitioning approach – inefficient scattered read/write

Preparing Foreign Vertices Vectors



Preparing Foreign Vertices Vectors

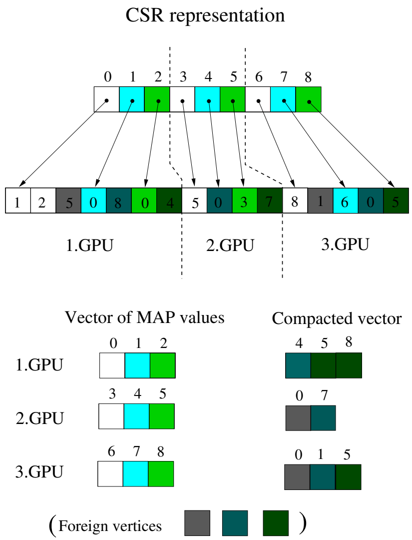
Synchronisation during MAP computation

- requires to exchange the values of the foreign vertices
- communication among GPU devices is realized through the host memory – maintains the complete vector of values
- first partitioning approach – efficient sequential read/write
- second partitioning approach – inefficient scattered read/write

Solution

- duplicate the values of the foreign vertices in the host memory
  - separate compacted vectors containing values for foreign vertices for particular CUDA devices
- create compacted vectors of values for foreign vertices on particular CUDA devices

Preparing Foreign Vertices Vectors



Preparing Foreign Vertices Vectors

Synchronisation during MAP computation

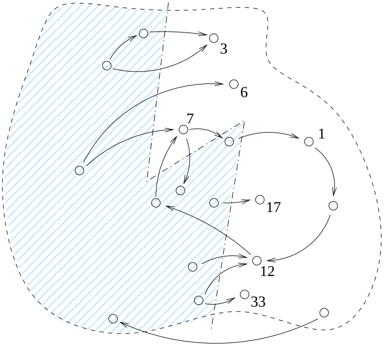
- requires to exchange the values of the foreign vertices
- communication among GPU devices is realized through the host memory – maintains the complete vector of values
- first partitioning approach – efficient sequential read/write
- second partitioning approach – inefficient scattered read/write

Solution

- duplicate the values of the foreign vertices in the host memory
  - separate compacted vectors containing values for foreign vertices for particular CUDA devices
- create compacted vectors of values for foreign vertices on particular CUDA devices
  - map the foreign vertices in the CSR representation with their counterparts in the compacted vector
  - efficient compaction procedure on CUDA

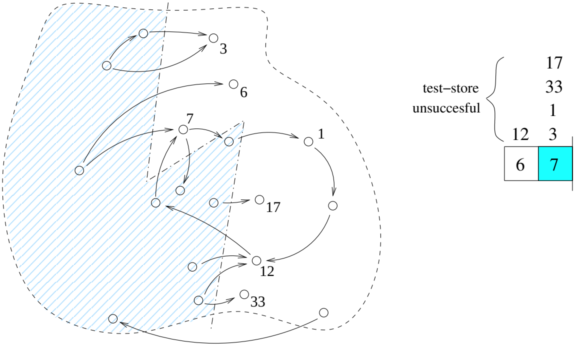
ParaDiSe Parallel & Distributed Systems Laboratory

Compaction Procedure - Illustration



- allocate a vector of size 2^i (i is a small integer)
- CUDA kernels performing iteratively the following operations:

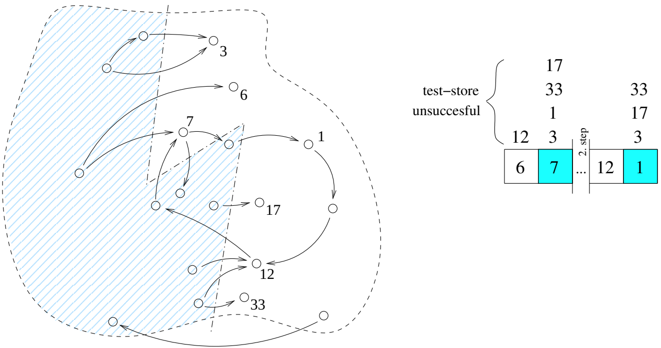
Compaction Procedure - Illustration



1.step

- store every foreign vertex v on the position $v\&(2^{i-1}-1)$
- in case of conflicts for multiple vertices on some positions, we keep only the first vertex stored

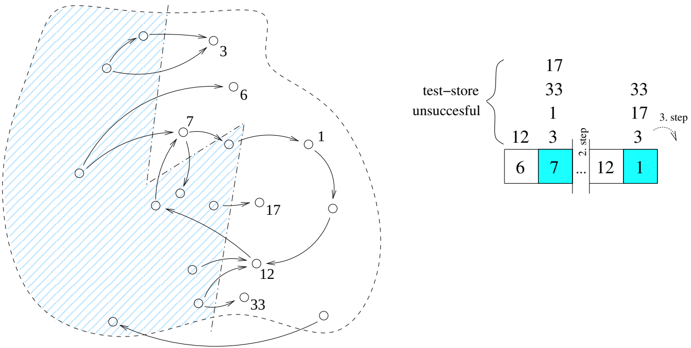
Compaction Procedure - Illustration



2.step

- store conflicting vertices v on the position $2^{i-1} + v\&(2^{i-1} - 1)$
- in case of conflicts for multiple vertices on some positions, we keep only the first vertex stored

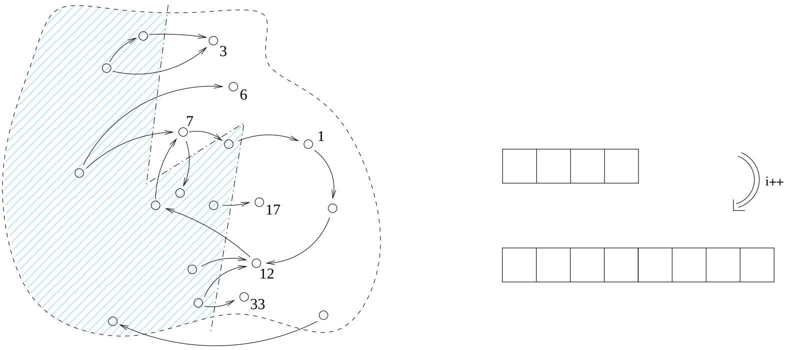
Compaction Procedure - Illustration



3.step

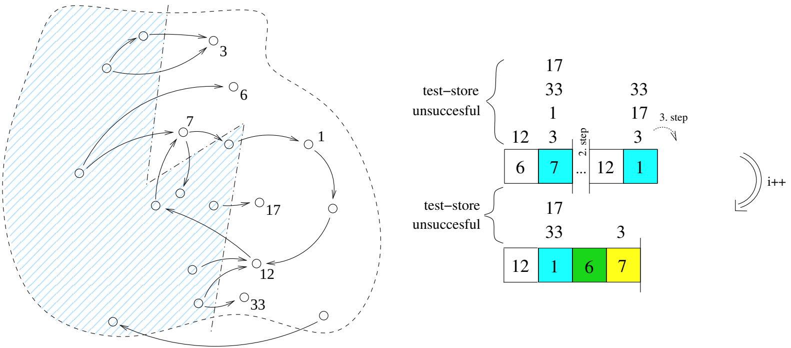
- in case of conflicts we sequentially look for empty position from $2^{i-1} + v\&(2^{i-1} - 1) + 1$ to $2^{i-1} + v\&(2^{i-1} - 1) + i$
- if there are conflicts after $\mathcal{O}(i)$ steps, we increment i and repeat the procedure

Compaction Procedure - Illustration



- allocate a vector of size 2^i
- CUDA kernels performing iteratively the following operations:

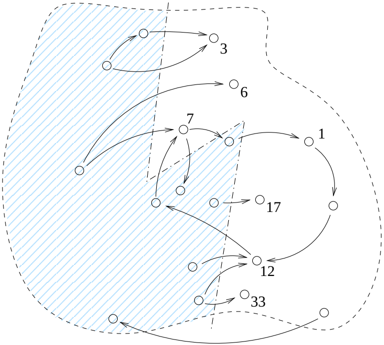
Compaction Procedure - Illustration

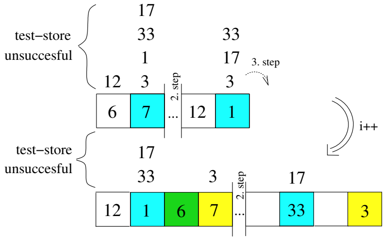


1.step

- store every foreign vertex v on the position $v\&(2^{i-1} - 1)$
- in case of conflicts for multiple vertices on some positions, we keep only the first vertex stored

Compaction Procedure - Illustration



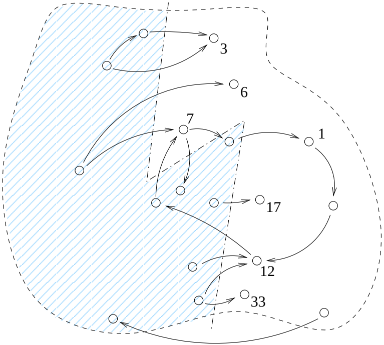


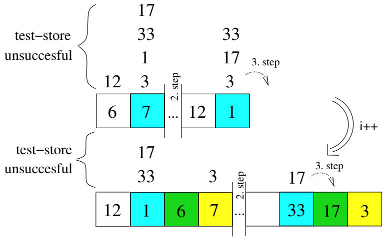
2.step

- store conflicting vertices v on the position $2^{i-1} + v\&(2^{i-1} - 1)$
- in case of conflicts for multiple vertices on some positions, we keep only the first vertex stored

ParaDiSe Parallel & Distributed Systems Laboratory

Compaction Procedure - Illustration

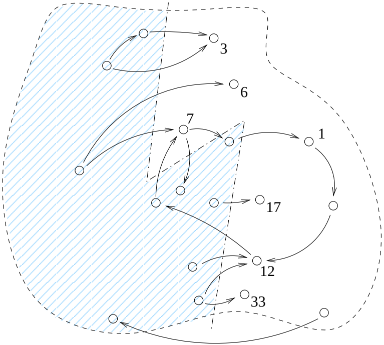




3.step

- in case of conflicts we sequentially look for empty position from $2^{i-1} + v\&(2^{i-1} - 1) + 1$ to $2^{i-1} + v\&(2^{i-1} - 1) + i$
- we have a compacted vector of the size 2^i containing all foreign vertices exactly once

Compaction Procedure - Illustration



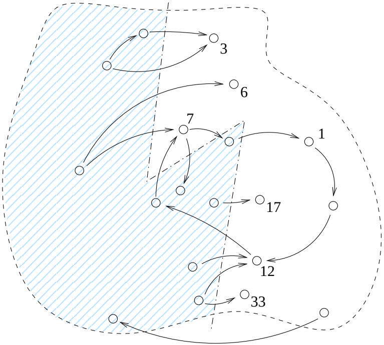
	1	3	6	7	12	17	33

Sorting the vector

- values of allocated vector are initialized on zeroes
- external sorting procedure

ParaDiSe Parallel & Distributed Systems Laboratory

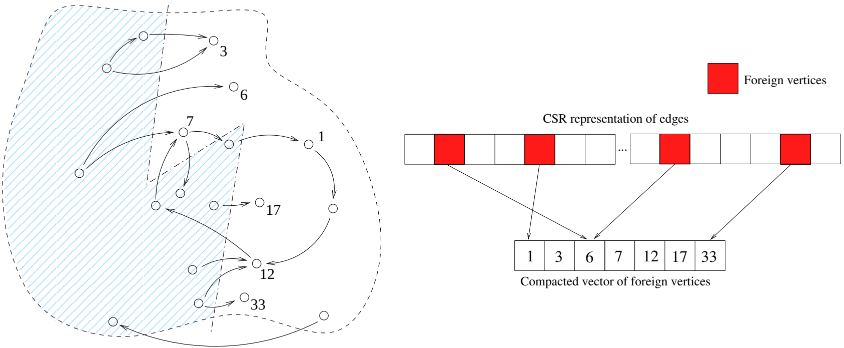
Compaction Procedure - Illustration



1	3	6	7	12	17	33

Cutting off the prefix of zeroes

Compaction Procedure - Illustration



- map the foreign vertices with their counterparts in the sorted compacted vector
- CUDA implementation of binary search

MAP Computation Algorithm

Algorithm 1 *MAP computation*

```
while globalChange ∧ ¬acc_found do
    foreignMAPs ← Download()
    localChange ← false
    while repropagate ∧ ¬acc_found do
        repropagate ← false
        mapKernel(G, localMAPs, foreignMAPs)
        Check(repropagate, acc_found)
        localChange ← localChange ∨ repropagate
    end while
    Upload(localMAPs)
    VoteIn(localChange)
    Rendezvous()
    globalChange ← VoteOut()
end while
```

MAP values of foreign vertices are received during the synchronisation with all the other GPU devices

MAP Computation Algorithm

Algorithm 1 *MAP computation*

```
while globalChange ∧ ¬acc_found do
    foreignMAPs ← DOWNLOAD()
    localChange ← false
    while repropagate ∧ ¬acc_found do
        repropagate ← false
        MAPKERNEL(G, localMAPs, foreignMAPs)
        CHECK(repropagate, acc_found)
        localChange ← localChange ∨ repropagate
    end while
    UPLOAD(localMAPs)
    VOTEIN(localChange)
    RENDEZVOUS()
    globalChange ← VOTEOUT()
end while
```

Every single CUDA device computes the local fix-point using the mutable MAP values of local vertices and the constant MAP values of foreign vertices

ParaDiSe Parallel & Distributed Systems Laboratory

MAP Computation Algorithm

Algorithm 1 *MAP computation*

```
while globalChange ∧ ¬acc_found do
    foreignMAPs ← Download()
    localChange ← false
    while repropagate ∧ ¬acc_found do
        repropagate ← false
        mapKernel(G, localMAPs, foreignMAPs)
        Check(repropagate, acc_found)
        localChange ← localChange ∨ repropagate
    end while
    Upload(localMAPs)
    VoteIn(localChange)
    Rendezvous()
    globalChange ← VoteOut()
end while
```

These steps are repeated until a global fix-point is found or accepting cycle is found.

Parallel & Distributed Systems Laboratory

MAP Computation Algorithm

Algorithm 1 *MAP computation*

```
while globalChange ∧ ¬acc_found do
    foreignMAPs ← Download()
    localChange ← false
    while repropagate ∧ ¬acc_found do
        repropagate ← false
        mapKernel(G, localMAPs, foreignMAPs)
        Check(repropagate, acc_found)
        localChange ← localChange ∨ repropagate
    end while
    Upload(localMAPs)
    VoteIn(localChange)
    Rendezvous()
    globalChange ← VoteOut()
end while
```

If the local fix-point is found in zero iterations (no change after synchronisation step) workers vote for global termination

MAP Computation Algorithm

Algorithm 1 *MAP computation*

while *globalChange* $\wedge$ $\neg$*acc_found* **do**
    foreignMAPs $\leftarrow$ DOWNLOAD()
    localChange $\leftarrow$ `false`
    while *repropagate* $\wedge$ $\neg$*acc_found* **do**
        repropagate $\leftarrow$ `false`
        MAPKERNEL(*G*, *localMAPs*, *foreignMAPs*)
        CHECK(*repropagate*, *acc_found*)
        localChange $\leftarrow$ *localChange* $\vee$ *repropagate*
    end while
    UPLOAD(*localMAPs*)
    VOTEIN(*localChange*)
    RENDEZVOUS()
    globalChange $\leftarrow$ VOTEOUT()
end while

If after a barrier operation the vote for termination is unanimous the algorithm terminates.

Redesign of OWCTY – Motivation

- OWCTY algorithm is more efficient than MAP algorithm on models without accepting cycle

ParaDiSe
Parallel & Distributed Systems Laboratory

Redesign of OWCTY – Motivation

- OWCTY algorithm is more efficient than MAP algorithm on models without accepting cycle
- experiments show that reversed BFS ordering in the CSR representation provides the best times
  - decrease of repropagation

Redesign of OWCTY – Motivation

- OWCTY algorithm is more efficient than MAP algorithm on models without accepting cycle
- experiments show that reversed BFS ordering in the CSR representation provides the best times
  - decrease of repropagation
- proposed methods affect this ordering
  - parallel construction of the CSR representation
  - partitioning of the CSR representation

Redesign of OWCTY – Motivation

- OWCTY algorithm is more efficient than MAP algorithm on models without accepting cycle
- experiments show that reversed BFS ordering in the CSR representation provides the best times
  - decrease of repropagation
- proposed methods affect this ordering
  - parallel construction of the CSR representation
  - partitioning of the CSR representation
- can lead to significant slowdown of the CUDA MAP algorithm
  - different number of calls to CUDA kernels
  - different memory access pattern

ParaDiSe Parallel & Distributed Systems Laboratory

Redesign of OWCTY – Motivation

- OWCTY algorithm is more efficient than MAP algorithm on models without accepting cycle
- experiments show that reversed BFS ordering in the CSR representation provides the best times
  - decrease of repropagation
- proposed methods affect this ordering
  - parallel construction of the CSR representation
  - partitioning of the CSR representation
- can lead to significant slowdown of the CUDA MAP algorithm
  - different number of calls to CUDA kernels
  - different memory access pattern
- OWCTY algorithm should prove more resistant to any improper ordering in CSR representation.

Reformulation of OWCTY Algorithm

- OWCTY algorithm comprises of alternating execution of
  - forward reachability
  - backward elimination – elimination of vertices without immediate predecessors

Reformulation of OWCTY Algorithm

- OWCTY algorithm comprises of alternating execution of
  - forward reachability
  - backward elimination – elimination of vertices without immediate predecessors
- data-parallel versions forward reachability
  - naive solution – trivial
  - more advance methods – [Barnat et al. IPDPS'11]

Reformulation of OWCTY Algorithm

- OWCTY algorithm comprises of alternating execution of
  - forward reachability
  - backward elimination – elimination of vertices without immediate predecessors
- data-parallel versions forward reachability
  - naive solution – trivial
  - more advance methods – [Barnat et al. IPDPS'11]
- data-parallel versions backward elimination
  - problems with backwards edges – space vs time
  - without backward edges elimination requires two steps
    - 1.step – set the flag to all successors
    - 2.step – eliminate vertices without flag

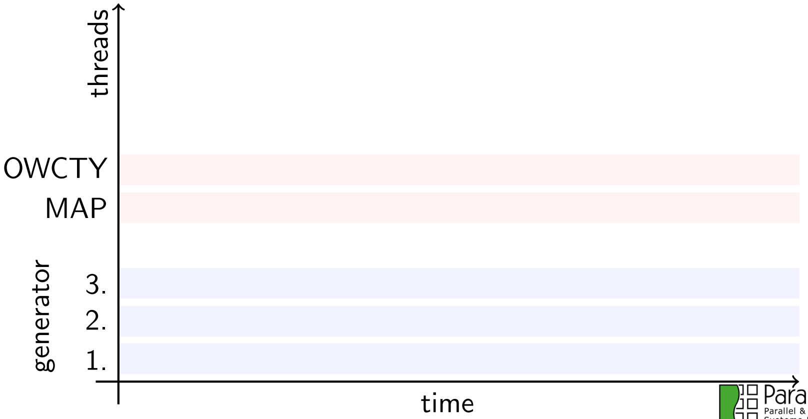
ParaDiSe
Parallel & Distributed Systems Laboratory

Reformulation of OWCTY Algorithm

- OWCTY algorithm comprises of alternating execution of
  - forward reachability
  - backward elimination – elimination of vertices without immediate predecessors
- data-parallel versions forward reachability
  - naive solution – trivial
  - more advance methods – [Barnat et al. IPDPS'11]
- data-parallel versions backward elimination
  - problems with backwards edges – space vs time
  - without backward edges elimination requires two steps
    - 1.step – set the flag to all successors
    - 2.step – eliminate vertices without flag
- Reversed OWCTY algorithm
  - forward elimination – elimination of vertices without immediate successors (require just one step)
  - back reachability – less efficient (slower propagation)

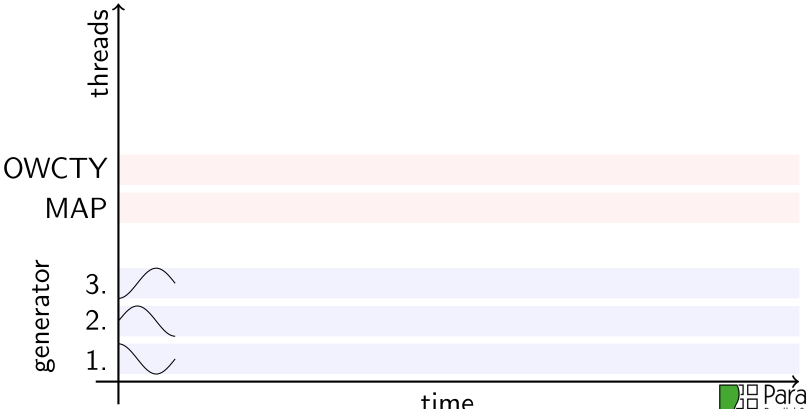
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



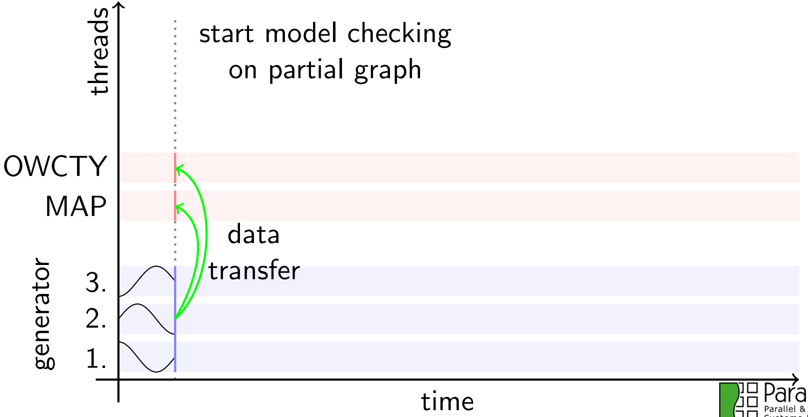
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



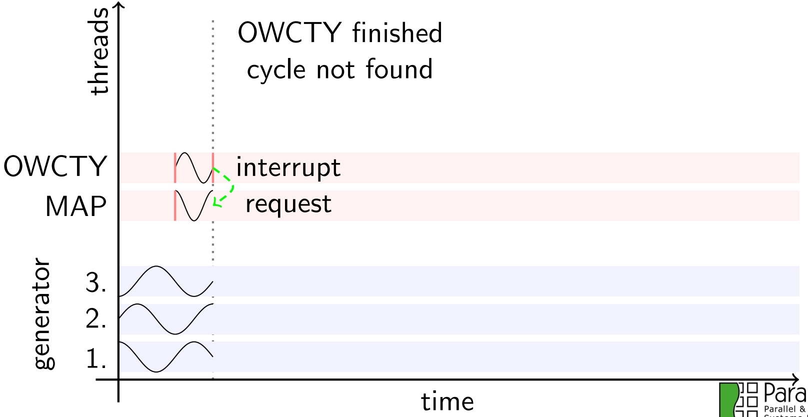
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



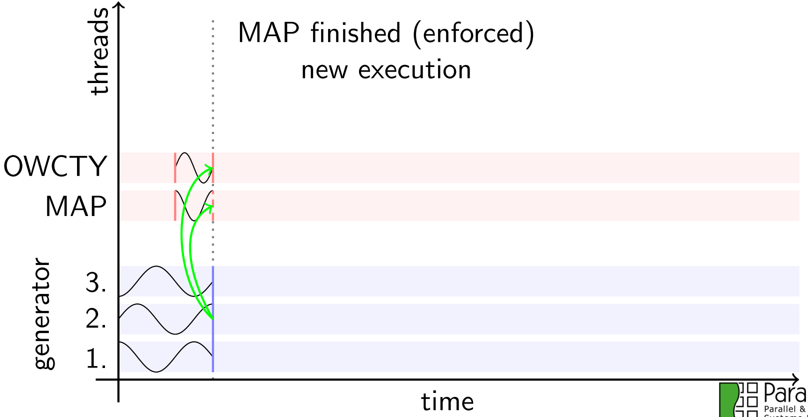
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



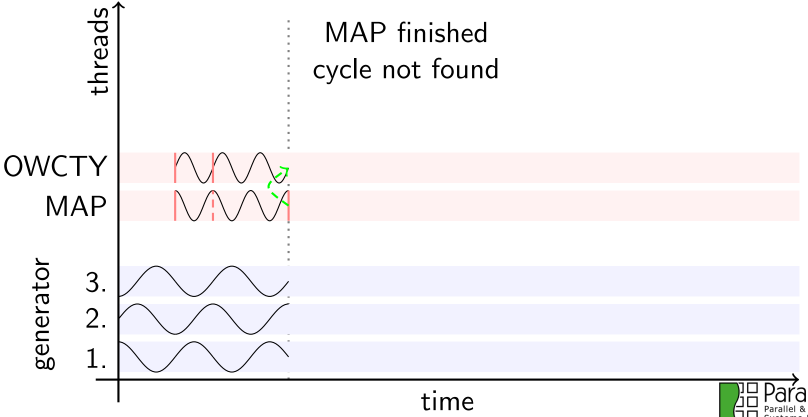
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



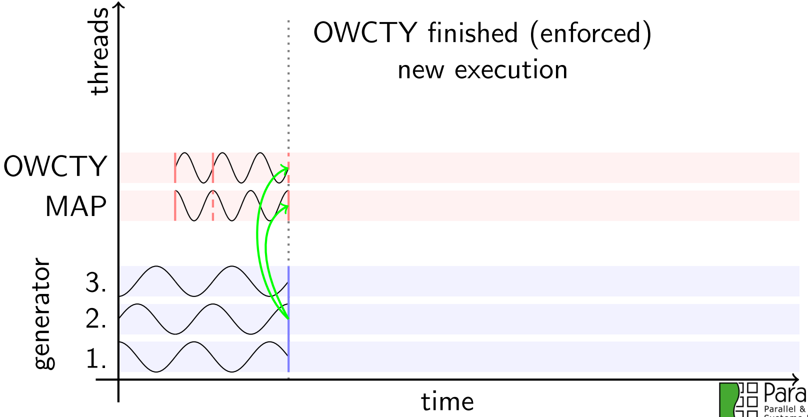
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



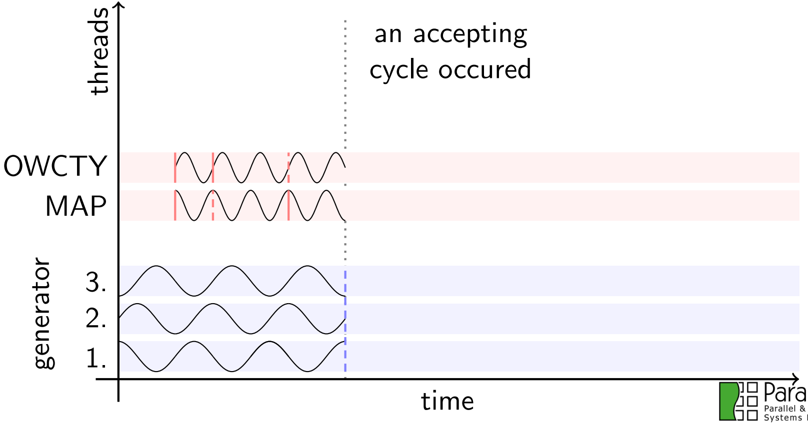
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



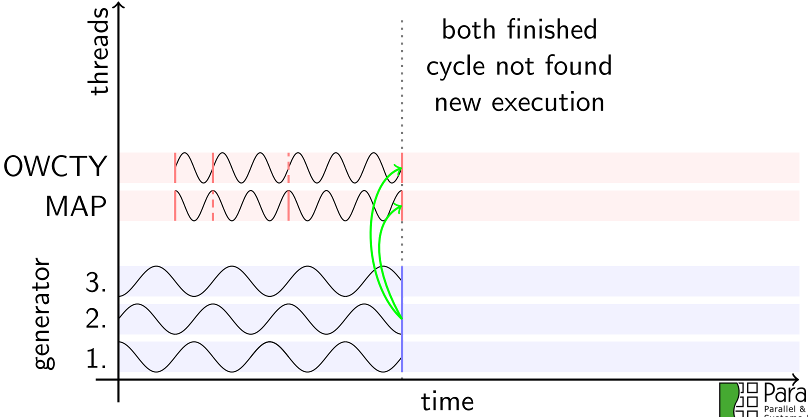
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



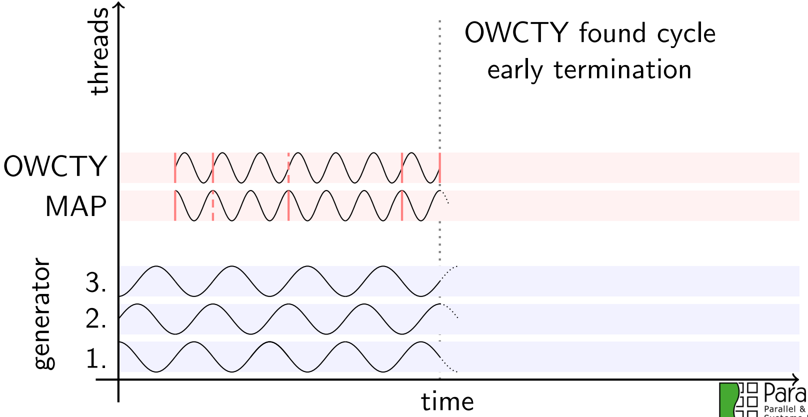
DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



DiVinE-CUDA Workflow

- multi-core state space generation
- on-the-fly model checking computation
- two CUDA devices: concurrent execution of both algorithms



Experimental Setting

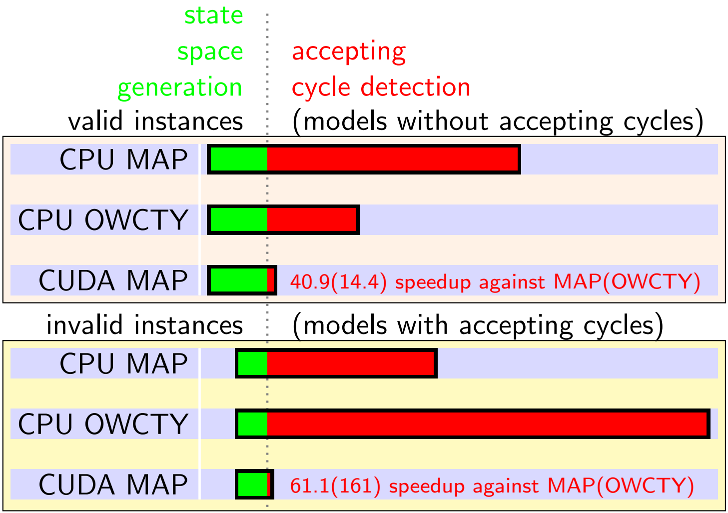
Linux workstation with

- quad core AMD Phenom(tm) II X4 940 Processor @ 3GHz
- 8 GB DDR2 @ 1066 MHz RAM
- two NVIDIA GeForce GTX 280 GPU's with 1GB of memory

- all the run-times in seconds
- CSR representation for models indicated by stars was created on a workstation with 32 GB RAM
- one core oversees the communication with CUDA device

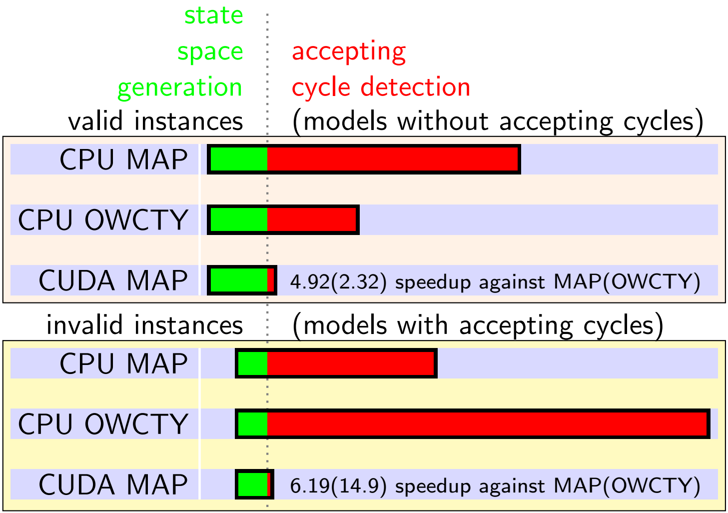
ParaDiSe
Parallel & Distributed
Systems Laboratory

Comparison of DIVINE and DIVINE-CUDA



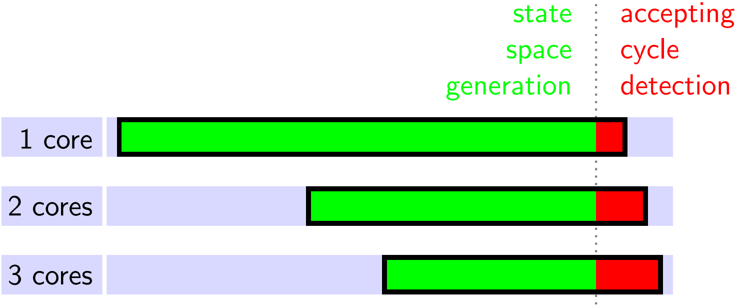
ParaDiSe Parallel & Distributed Systems Laboratory

Comparison of DiVinE and DiVinE-CUDA



ParaDiSe Parallel & Distributed Systems Laboratory

Multi-core Acceleration of CSR Construction



- parallel CSR construction affects the ordering in CSR representation
- can lead to significant slowdown of the MAP algorithm
  - different number of calls to CUDA kernels
  - different memory access pattern

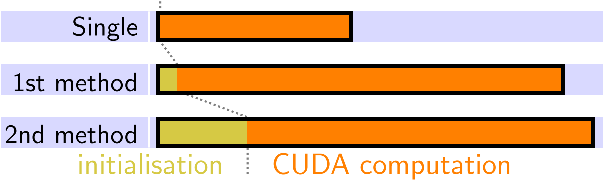
Employing Multiple CUDA Devices

verification of much larger model checking instances

- cannot be verified using the original CUDA algorithm
- fits the aggregate memory of multiple CUDA devices

slowdown of CUDA computation

- multiple CUDA computation requires the synchronisations
- 2nd method needs more time for initial phases



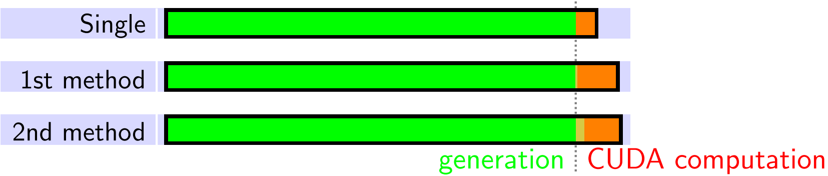
Employing Multiple CUDA Devices

verification of much larger model checking instances

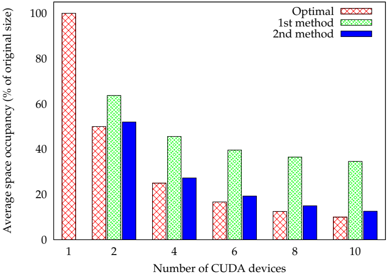
- cannot be verified using the original CUDA algorithm
- fits the aggregate memory of multiple CUDA devices

slowdown of CUDA computation

- multiple CUDA computation requires the synchronisations
- 2nd method needs more time for initial phases
- negligible with respect to the whole model checking procedure



Space Efficiency of Two Proposed Partitioning



- illustrate ability to efficiently utilise space when increasing number of CUDA devices is employed
- average over all tested models
- 2nd method is significantly better for partitioning of a wide variety of graphs

Comparison of CUDA MAP and CUDA OWCTY

Invalid instances (models with accepting cycles)

- algorithms have almost same times on most of the instances
- on some instances (e.g. `peterson 2`) the MAP algorithm falls behind both the OWCTY algorithms significantly

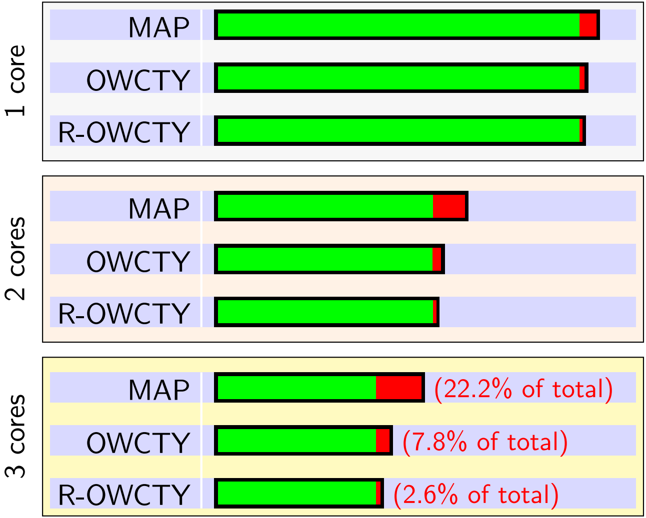
Models (seq. total time: MAP/OWCTY)	CPU cores	*CSR time*	CUDA MAP		CUDA OWCTY		CUDA OWCTY REVERSE	
			CUDA time	*total time*	*CUDA time*	*total time*	*CUDA time*	*total time*
peterson 2 (173/404)	1	25.7	4.0	30.5	0.4	26.9	0.3	26.8
	2	17.4	4.3	22.5	0.6	18.8	0.8	19.0
	3	12.5	0.6	13.8	1.2	14.4	1.0	14.2

- CUDA accelerated MAP algorithm deeply depends on the ordering in CSR representation
- CUDA accelerated OWCTY algorithm is more resistant to slowdown caused by improper ordering

ParaDiSe Parallel & Distributed Systems Laboratory

Comparison of CUDA MAP and CUDA OWCTY

Valid instances (models without accepting cycles)



Conclusion

Successful redesign of MAP algorithm

- significant GPU acceleration of LTL Model Checking
- DiVinE-CUDA – tool for CUDA Accelerated Model Checking

ParaDiSe
Parallel & Distributed Systems Laboratory

Conclusion

Successful redesign of MAP algorithm

- significant GPU acceleration of LTL Model Checking
- DiVinE-CUDA – tool for CUDA Accelerated Model Checking

Multi-core acceleration of CSR construction

- significant speed-up of expensive phase of CSR construction
- utilization of modern multi-core machines with CUDA

Conclusion

Successful redesign of MAP algorithm

- significant GPU acceleration of LTL Model Checking
- DiVinE-CUDA – tool for CUDA Accelerated Model Checking

Multi-core acceleration of CSR construction

- significant speed-up of expensive phase of CSR construction
- utilization of modern multi-core machines with CUDA

Overcoming of single GPU memory limitations

- verification of much larger model checking problems
- preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm
- designed methods can be used for others graph problems

Conclusion

Successful redesign of MAP algorithm

- significant GPU acceleration of LTL Model Checking
- DiVinE-CUDA – tool for CUDA Accelerated Model Checking

Multi-core acceleration of CSR construction

- significant speed-up of expensive phase of CSR construction
- utilization of modern multi-core machines with CUDA

Overcoming of single GPU memory limitations

- verification of much larger model checking problems
- preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm
- designed methods can be used for others graph problems

Successful redesign of the OWCTY algorithm

- significantly outperform the original CUDA MAP algorithm
- robust to improper ordering in the input representation

ParaDiSe Parallel & Distributed Systems Laboratory

Conclusion

Successful redesign of MAP algorithm

- significant GPU acceleration of LTL Model Checking
- DiVinE-CUDA – tool for CUDA Accelerated Model Checking

Multi-core acceleration of CSR construction

- significant speed-up of expensive phase of CSR construction
- utilization of modern multi-core machines with CUDA

Overcoming of single GPU memory limitations

- verification of much larger model checking problems
- preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm
- designed methods can be used for others graph problems

Successful redesign of the OWCTY algorithm

- significantly outperform the original CUDA MAP algorithm
- robust to improper ordering in the input representation

Exhaustive experimental evaluation

ParaDiSe Parallel & Distributed Systems Laboratory

Current and Future Work

Improve the performance of multi CUDA algorithms

- on-the-fly property
- new techniques allowing efficient utilization of GPU clusters

Current and Future Work

Improve the performance of multi CUDA algorithms

- on-the-fly property
- new techniques allowing efficient utilization of GPU clusters

CUDA accelerated state space generation

- crucial for CUDA accelerated model checking
- [Edelkam et al. SPIN'10]
  - CUDA accelerated
    - checking of transitions enabledness
    - generating the successors
  - nonsignificant speed up due to CPU duplicate detection
- necessity of CUDA accelerated duplicate detection

Current and Future Work

Improve the performance of multi CUDA algorithms

- on-the-fly property
- new techniques allowing efficient utilization of GPU clusters

CUDA accelerated state space generation

- crucial for CUDA accelerated model checking
- [Edelkam et al. SPIN'10]
  - CUDA accelerated
    - checking of transitions enabledness
    - generating the successors
  - nonsignificant speed up due to CPU duplicate detection
- necessity of CUDA accelerated duplicate detection

CUDA acceleration of other graph problems

- strongly connected component decomposition [Barnat et al. IPDPS'11]
- mean weight cycles

ParaDiSe Parallel & Distributed Systems Laboratory

The End

Thank you for your attention.

More information:

http://www.fi.muni.cz/paradise

xceska@fi.muni.cz

ParaDiSe
Parallel & Distributed
Systems Laboratory