Employing Multiple CUDA Devices to Accelerate LTL Model Checking

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

Faculty of Informatics, Masaryk University
Brno, Czech Republic

MEMICS 2010

based on

[Barnat et al. ICPADS'10]

23.10, 2010
Motivation

Model Checking

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

DiVinE

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

Many-core architectures

- Parallel computing platform of the future?
- Widely accessible due to GP GPU devices
Motivation

Model Checking

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

DiVinE

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

Many-core architectures

- Parallel computing platform of the future?
- Widely accessible due to GP GPU devices
Motivation

Model Checking

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

DiVinE

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

Many-core architectures

- Parallel computing platform of the future?
- Widely accessible due to GP GPU devices
Acceleration of model checking process by full utilization of modern massively parallel architectures

- successful redesign of Maximal Accepting Predecessor algorithm allowing for 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:

1. expensive phase of encoding the state space into a suitable and compact representation
2. limited to the middle-size instances that can fit the memory of a single GPU device
CUDA Accelerated LTL Model Checking

Acceleration of model checking process by full utilization of modern massively parallel architectures

- successful redesign of Maximal Accepting Predecessor algorithm allowing for 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:

1. expensive phase of encoding the state space into a suitable and compact representation
2. limited to the middle-size instances that can fit the memory of a single GPU device
Acceleration of model checking process by full utilization of modern massively parallel architectures

- successful redesign of Maximal Accepting Predecessor algorithm allowing for 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:

1. expensive phase of encoding the state space into a suitable and compact representation
2. limited to the middle-size instances that can fit the memory of a single GPU device
LTL Model Checking

Does the model $M$ satisfy the formula $\varphi$?

Inspected system $\rightarrow$ Model $M$ $\rightarrow$ Formula $\varphi$ $\leftarrow$ Required property

Model Checker

Yes, $M$ satisfies $\varphi$  
No, $M$ does not satisfy $\varphi$ (counter example)

- required property $\rightarrow$ formula in Linear Temporal Logic (LTL)
- reduction on accepting cycle detection
Maximal Accepting Predecessor (MAP) Algorithm

Graph corresponding to the state space.
Maximal Accepting Predecessor (MAP) Algorithm

Accepting vertices, accepting cycle.
Maximal Accepting Predecessor (MAP) Algorithm

Vertex ordering.
Maximal Accepting Predecessor (MAP) Algorithm

\[ \text{map}(v) = \max\{\perp, u \mid (u, v) \in E^+ \land \mathcal{A}(u)\} \]
Maximal Accepting Predecessor (MAP) Algorithm

\[ map(v) = v \quad \implies \quad \text{accepting cycle} \]
What if $2 > 4$?
Maximal Accepting Predecessor (MAP) Algorithm

Accepting cycle undetected.
If no accepting cycle is found, then maximal accepting vertices cannot be part of an accepting cycle.
Maximal Accepting Predecessor (MAP) Algorithm

Maximal accepting vertices marked as non-accepting.
Repeat until accepting cycle is found or there are no accepting vertices.
Computing Values of MAP – Multi-cores

Graph partitioning.
Computing Values of MAP – Multi-cores

Each core processes own vertices.
Computing Values of MAP – Multi-cores

Each core processes own vertices.
Computing Values of MAP – Multi-cores

Non local vertices are sent to the owners.
Computing Values of MAP – Multi-cores

Parallel processing of vertices.
Computing Values of MAP – Multi-cores

Parallel processing of vertices.
Computing Values of MAP – Multi-cores

Parallel processing of vertices.
Computing Values of MAP – Multi-cores

Accepting cycle found.
Computing Values of MAP – Many-cores

One thread per vertex.
Computing Values of MAP – Many-cores

4 > 2 > 1

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

\[ \begin{pmatrix} M \end{pmatrix} \times \begin{pmatrix} V \end{pmatrix} = \begin{pmatrix} V' \end{pmatrix} \]
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

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

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

\[
\text{maxacc}(u, v) = \begin{cases} 
\max\{\text{map}(u), \text{map}(v), u\} & \text{if } A(u) \\
\max\{\text{map}(u), \text{map}(v)\} & \text{otherwise}
\end{cases}
\]
Compact Representation

Size of the GPU memory is limited

- handling full matrices of predecessors is memory inefficient

Matrix of predecessors is sparse

- matrix is made of 1 and 0 only
- Compact Sparse Row (CSR) representation

\[
\begin{pmatrix}
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

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

- hard to compute – matrix of successors
- adjacency list represented as two arrays
Memory Limitations

GPU Memory consumption

- Data stored on GPU
  - 12B per vertex (4B due to CSR + 8B due to \(map, oldmap, A, r\))
  - 4B per edge (CSR)

\[
\begin{pmatrix}
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

- 1 GB of GPU memory
  - 30 millions of vertices, 150 millions of edges (avg. outdegree 5)
  - 50 millions of vertices, 100 millions of edges (avg. outdegree 2)
Memory Limitations

GPU Memory consumption

- Data stored on GPU
  - 12B per vertex (4B due to CSR + 8B due to $map, oldmap, A, r$)
  - 4B per edge (CSR)

\[
\begin{pmatrix}
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

- 1 GB of GPU memory
  - 30 millions of vertices, 150 millions of edges (avg. outdegree 5)
  - 50 millions of vertices, 100 millions of edges (avg. outdegree 2)
Two Weaknesses of our Approach

1. Expensive construction of the CSR representation
   - CSR construction takes 93% of total verification time
Paral·lel Construction of CSR Representation I

Basic idea

1. Parallel multi-core 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 exchanging – contention and lock-free queue structures

2. Concurrent parallel construction of CSR representation
   - assigning a unique number between 1 and $|V|$ to 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
Basic idea

1. Parallel multi-core state space generation
   - hash-based partitioning of the graph
   - local storage for local vertices
   - non-local vertices are handed out to the owning threads
   - vertices exchanging – contention and lock-free queue structures

2. Concurrent parallel construction of CSR representation
   - assigning a unique number between 1 and $|V|$ to 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
Parallel Construction of CSR Representation II

Problem solution

- vertex number is composed of two parts
  - thread-unique *thread_id* (4 bits)
  - per thread-unique *vertex_number* (28 bits)

- local vectors allowing insertion in two different ways
  - direct insertion for local transitions
  - two-phase (allocation, storage) insertion for cross transitions

- special handling of cross transitions
  - preallocated space is sent together with the vertex
  - owning thread assigns a number to the vertex and stores it to the preallocated space remotely

- local vectors are concatenated into a single system-wide vector

- CUDA kernels translate the pairs of numbers into continuous range of integers
Problem solution

- vertex number is composed of two parts
  - thread-unique *thread_id* (4 bits)
  - per thread-unique *vertex_number* (28 bits)
- local vectors allowing insertion in two different ways
  - direct insertion for local transitions
  - two-phase (allocation, storage) insertion for cross transitions
- special handling of cross transitions
  - preallocated space is sent together with the vertex
  - owning thread assigns a number to the vertex and stores it to the preallocated space remotely
- local vectors are concatenated into a single system-wide vector
- CUDA kernels translate the pairs of numbers into continuous range of integers
Parallel Construction of CSR Representation II

Problem solution

- vertex number is composed of two parts
  - thread-unique \( thread\_id \) (4 bits)
  - per thread-unique \( vertex\_number \) (28 bits)
- local vectors allowing insertion in two different ways
  - direct insertion for local transitions
  - two-phase (allocation, storage) insertion for cross transitions
- special handling of cross transitions
  - preallocated space is sent together with the vertex
  - owning thread assigns a number to the vertex and stores it to the preallocated space remotely
- local vectors are concatenated into a single system-wide vector
- CUDA kernels translate the pairs of numbers into continuous range of integers
Problem solution

- vertex number is composed of two parts
  - thread-unique \textit{thread\_id} (4 bits)
  - per thread-unique \textit{vertex\_number} (28 bits)
- local vectors allowing insertion in two different ways
  - direct insertion for local transitions
  - two-phase (allocation, storage) insertion for cross transitions
- special handling of cross transitions
  - preallocated space is sent together with the vertex
  - owning thread assigns a number to the vertex and stores it to the preallocated space remotely
- local vectors are concatenated into a single system-wide vector
- CUDA kernels translate the pairs of numbers into continuous range of integers
Parallel Construction of CSR Representation II

Problem solution

- vertex number is composed of two parts
  - thread-unique \textit{thread\_id} (4 bits)
  - per thread-unique \textit{vertex\_number} (28 bits)
- local vectors allowing insertion in two different ways
  - direct insertion for local transitions
  - two-phase (allocation, storage) insertion for cross transitions
- special handling of cross transitions
  - preallocated space is sent together with the vertex
  - owning thread assigns a number to the vertex and stores it to
    the preallocated space remotely
- local vectors are concatenated into a single system-wide vector
- CUDA kernels translate the pairs of numbers into continuous
  range of integers
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

![Diagram showing the workflow of DiVinE-CUDA with multi-core state space generation and one core overseeing communication with GPU device.](image)
DiVinE-CUDA Workflow

On-the-fly model checking computation

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

start model checking on partial graph

data transfer
On-the-fly model checking computation

- multi-core state space generation
- one core oversees the communication with GPU device
On-the-fly model checking computation

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

MAP finished
cycle not found
new execution
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

1. Expensive construction of the CSR representation
   - CSR construction takes 93% of total verification time

2. Limited only to middle-size model checking problems
   - CSR representation has to fit to GPU memory
Basic idea

- splitting the data structures into parts and distributing them among multiple GPU devices
- allowing verification of the problems that fit the aggregate memory of multiple GPU devices
- two data structures has to be considered for partitioning
  1. CSR representation of the graph
  2. vector of values associated with individual vertices

\[
\begin{align*}
  M_i \\ V \\ V' \\
\end{align*}
\]
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
     - respects successors of individual vertices

\[\begin{align*}
1.\text{GPU} & \quad M_1 \quad V \quad V' \\
2.\text{GPU} & \quad M_2 \quad V \quad V' \\
3.\text{GPU} & \quad M_3 \quad V \quad V'
\end{align*}\]
Two Proposed Partitioning

CSR representation

Vector of MAP values

1.GPU

2.GPU

3.GPU
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
     - respects successors of individual vertices
     - foreign vertices – all vertices stored in foreign parts of the CSR representation

![Diagram showing partitioning of the graph]
Two Proposed Partitioning

CSR representation

Vector of MAP values

( Foreign 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
     - respects successors of 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

\[
\begin{align*}
\text{1.GPU} & : \quad M_1 \times V = V' \\
\text{2.GPU} & : \quad M_2 \times V = V' \\
\text{3.GPU} & : \quad M_3 \times V = V'
\end{align*}
\]
Two Proposed Partitioning

CSR representation

Vector of MAP values

1. GPU

2. GPU

3. GPU
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
     - respects successors of 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
Two Proposed Partitioning

CSR representation

Vector of MAP values

(Foreign vertices)
Preparing Foreign Vertices Vectors

Synchronisation during MAP computation

- requires to exchange the values of the foreign vertices
- communication between 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

CSR representation

Vector of MAP values

Foreign vertices

Milan Češka et al.  
Employing Multiple CUDA Devices to Accelerate LTL Model Checking  
23.10, 2010 18 / 30
Preparing Foreign Vertices Vectors

**Synchronisation during MAP computation**

- requires to exchange the values of the foreign vertices
- communication between 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

CSR representation

Vector of MAP values

(Foreign vertices)
Preparing Foreign Vertices Vectors

Synchronisation during MAP computation

- requires to exchange the values of the foreign vertices
- communication between 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 GPU devices
- create compacted vectors of values for foreign vertices on particular GPU devices
Preparing Foreign Vertices Vectors

CSR representation

Vector of MAP values

Compacted vector

(Foreign vertices)
Preparing Foreign Vertices Vectors

Synchronisation during MAP computation

• requires to exchange the values of the foreign vertices
• communication between 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 GPU devices
• create compacted vectors of values for foreign vertices on particular GPU devices
  • map the foreign vertices in the CSR representation with their counterparts in the compacted vector
  • efficient compaction procedure on CUDA
Compaction Procedure on CUDA

- mark all foreign vertices in the vector of edges
- some foreign vertices may appear in the structure repeatedly
- **compute number of unique foreign vertices**
  - efficient solution using reduce requires a copy of the whole vector of vertices – problem due to space limitation
  - memory efficient solution requires `atomicCAS` – time expensive
  - heuristics to obtain a safe but close overapproximation
- sort the compacted vector
- remove unused part of the compacted vector
- use a binary search to map the foreign vertices in the CSR representation with their counterparts in the compacted vector
Compaction Procedure on CUDA

• mark all foreign vertices in the vector of edges
• some foreign vertices may appear in the structure repeatedly
• **compute number of unique foreign vertices**
  • efficient solution using reduce requires a copy of the whole vector of vertices – problem due to space limitation
  • memory efficient solution requires atomicCAS – time expensive
  • heuristics to obtain a safe but close overapproximation
• sort the compacted vector
• remove unused part of the compacted vector
• use a binary search to map the foreign vertices in the CSR representation with their counterparts in the compacted vector
Compaction Procedure on CUDA

- mark all foreign vertices in the vector of edges
- some foreign vertices may appear in the structure repeatedly
- **compute number of unique foreign vertices**
  - efficient solution using reduce requires a copy of the whole vector of vertices – problem due to space limitation
  - memory efficient solution requires `atomicCAS` – time expensive
  - heuristics to obtain a safe but close overapproximation
- sort the compacted vector
- remove unused part of the compacted vector
- use a binary search to map the foreign vertices in the CSR representation with their counterparts in the compacted vector
Compaction Procedure - Illustration

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

- store every foreign vertex \( v \) on the position \( v \& (2^i - 1) \)
- 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 $O(i)$ 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:
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

- test–store unsuccessful
- test–store unsuccessful
- \( i++ \)
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
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

Sorting the vector
- values of allocated vector are initialized on zeroes
- external sorting procedure
Compaction Procedure - Illustration

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**

1. while $\text{globalChange} \land \neg \text{acc\_found}$ do
2.     $\text{foreignMAPs} \leftarrow \text{DOWNLOAD}()$
3.     $\text{localChange} \leftarrow \text{false}$
4.     while $\text{repropagate} \land \neg \text{acc\_found}$ do
5.         $\text{repropagate} \leftarrow \text{false}$
6.         $\text{MAPKernel}(G, \text{localMAPs}, \text{foreignMAPs})$
7.         $\text{Check}(\text{repropagate}, \text{acc\_found})$
8.         $\text{localChange} \leftarrow \text{localChange} \lor \text{repropagate}$
9.         end while
10. $\text{Upload}(\text{localMAPs})$
11. $\text{VoteIn}(\text{localChange})$
12. $\text{Rendezvous}()$
13. $\text{globalChange} \leftarrow \text{VoteOut}()$
14. end while

MAP values of foreign vertices are received during the synchronisation with all the other GPU devices.
Algorithm 1 MAP computation

1: while globalChange ∧ ¬acc_found do
2:   foreignMAPs ← DOWNLOAD()
3:   localChange ← false
4: while repropagate ∧ ¬acc_found do
5:   repropagate ← false
6:   MAPKernel(G, localMAPs, foreignMAPs)
7:   Check(repropagate, acc_found)
8:   localChange ← localChange ∨ repropagate
9: end while
10: Upload(localMAPs)
11: VoteIn(localChange)
12: Rendezvous()
13: globalChange ← VoteOut()
14: end while

Every single GPU device computes the local fix-point using the mutable MAP values of local vertices and the constant MAP values of foreign vertices.
Algorithm 1 MAP computation

1: while \( \text{globalChange} \land \neg \text{acc\_found} \) do
2: \hspace{1em} \text{foreignMAPs} \leftarrow \text{DOWNLOAD}()
3: \hspace{1em} \text{localChange} \leftarrow \text{false}
4: \hspace{1em} while \( \text{repropagate} \land \neg \text{acc\_found} \) do
5: \hspace{2em} \text{repropagate} \leftarrow \text{false}
6: \hspace{2em} \text{MAPKernel}(G, \text{localMAPs}, \text{foreignMAPs})
7: \hspace{2em} \text{CHECK}(\text{repropagate, acc\_found})
8: \hspace{2em} \text{localChange} \leftarrow \text{localChange} \lor \text{repropagate}
9: \hspace{1em} end while
10: \hspace{1em} \text{UPLOAD}(\text{localMAPs})
11: \hspace{1em} \text{VOTEIN}(\text{localChange})
12: \hspace{1em} \text{RENEZVOUS}()
13: \hspace{1em} \text{globalChange} \leftarrow \text{VOTEOUT}()
14: end while

These steps are repeated until a global fix-point or an accepting cycle is found.
Algorithm 1 MAP computation

1: while $globalChange \land \neg acc\_found$ do
2:   $foreignMAPs \leftarrow \text{DOWNLOAD}()$
3:   $localChange \leftarrow \text{false}$
4:   while $repropagate \land \neg acc\_found$ do
5:     $repropagate \leftarrow \text{false}$
6:     $\text{MAPKernel}(G, localMAPs, foreignMAPs)$
7:     $\text{CHECK}(repropagate, acc\_found)$
8:     $localChange \leftarrow localChange \lor repropagate$
9:   end while
10:  $\text{UPLOAD}(localMAPs)$
11:  $\text{VOTEIN}(localChange)$
12:  $\text{RENEZVOUS}()$
13:  $globalChange \leftarrow \text{VOTEOUT}()$
14: end while

If the local fix-point is found in zero iterations (no change after synchronisation step) workers vote for global termination.
Algorithm 1  *MAP computation*

1: while $\text{globalChange} \land \neg \text{acc\_found}$ do
2:    foreignMAPs $\leftarrow \text{DOWNLOAD}()$
3:    localChange $\leftarrow$ false
4:    while $\text{repropagate} \land \neg \text{acc\_found}$ do
5:        repropagate $\leftarrow$ false
6:        MAPKernel($G$, localMAPs, foreignMAPs)
7:        Check(repropagate, acc\_found)
8:        localChange $\leftarrow$ localChange $\lor$ repropagate
9:    end while
10:   Upload(localMAPs)
11:   VoteIn(localChange)
12:   Rendezvous()
13:   $\text{globalChange} \leftarrow \text{VoteOut}()$
14: end while

If after a barrier operation the vote for termination is unanimous the algorithm terminates.
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 GPU device
### Multi-core Acceleration of CSR Construction

<table>
<thead>
<tr>
<th>Model</th>
<th>acc. cycle</th>
<th>1 core</th>
<th>2 cores</th>
<th>3 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSR time</td>
<td>CUDA time</td>
<td>total time</td>
<td># kernel calls</td>
</tr>
<tr>
<td>elevator 1</td>
<td>N</td>
<td>24.5</td>
<td>6.0</td>
<td>31.6</td>
</tr>
<tr>
<td>leader</td>
<td>N</td>
<td>86.0</td>
<td>0.1</td>
<td>87.4</td>
</tr>
<tr>
<td>peterson 1</td>
<td>N</td>
<td>97.9</td>
<td>3.5</td>
<td>102.3</td>
</tr>
<tr>
<td>anderson</td>
<td>N</td>
<td>30.6</td>
<td>1.5</td>
<td>33.2</td>
</tr>
<tr>
<td>lamport*</td>
<td>N</td>
<td>1118.1</td>
<td>ofm</td>
<td>ofm</td>
</tr>
<tr>
<td>elevator 2</td>
<td>Y</td>
<td>27.2</td>
<td>0.6</td>
<td>28.7</td>
</tr>
<tr>
<td>phils</td>
<td>Y</td>
<td>45.2</td>
<td>&lt;0.1</td>
<td>46.1</td>
</tr>
<tr>
<td>peterson 2</td>
<td>Y</td>
<td>25.7</td>
<td>4.0</td>
<td>30.5</td>
</tr>
<tr>
<td>bakery</td>
<td>Y</td>
<td>22.1</td>
<td>&lt;0.1</td>
<td>23.2</td>
</tr>
<tr>
<td>brp</td>
<td>Y</td>
<td>1.2</td>
<td>&lt;0.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

- significant speedup of the CSR construction when more CPU cores are used
### Multi-core Acceleration of CSR Construction

<table>
<thead>
<tr>
<th>Model</th>
<th>acc. cycle</th>
<th>1 core</th>
<th>2 cores</th>
<th>3 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CSR time</td>
<td>CUDA time</td>
<td>total time</td>
</tr>
<tr>
<td>elevator 1</td>
<td>N</td>
<td>24.5</td>
<td>6.0</td>
<td>31.6</td>
</tr>
<tr>
<td>leader</td>
<td>N</td>
<td>86.0</td>
<td>0.1</td>
<td>87.4</td>
</tr>
<tr>
<td>peterson 1</td>
<td>N</td>
<td>97.9</td>
<td>3.5</td>
<td>102.3</td>
</tr>
<tr>
<td>anderson</td>
<td>N</td>
<td>30.6</td>
<td>1.5</td>
<td>33.2</td>
</tr>
<tr>
<td>lamport*</td>
<td>N</td>
<td>1118.1</td>
<td>ofm</td>
<td>ofm</td>
</tr>
<tr>
<td>elevator 2</td>
<td>Y</td>
<td>27.2</td>
<td>0.6</td>
<td>28.7</td>
</tr>
<tr>
<td>phils</td>
<td>Y</td>
<td>45.2</td>
<td>&lt; 0.1</td>
<td>46.1</td>
</tr>
<tr>
<td>peterson 2</td>
<td>Y</td>
<td>25.7</td>
<td>4.0</td>
<td>30.5</td>
</tr>
<tr>
<td>bakery</td>
<td>Y</td>
<td>22.1</td>
<td>&lt; 0.1</td>
<td>23.2</td>
</tr>
<tr>
<td>brp</td>
<td>Y</td>
<td>1.2</td>
<td>&lt; 0.1</td>
<td>2.1</td>
</tr>
</tbody>
</table>

- 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
### Experiments

<table>
<thead>
<tr>
<th>Model</th>
<th>acc. cycle</th>
<th>single CUDA</th>
<th>dual CUDA 1st method</th>
<th>dual CUDA 2nd method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td># kernel calls</td>
<td># synchs</td>
</tr>
<tr>
<td>elevator 1</td>
<td>N</td>
<td>6.0</td>
<td>459</td>
<td>145</td>
</tr>
<tr>
<td>leader</td>
<td>N</td>
<td>0.1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>peterson 1</td>
<td>N</td>
<td>3.5</td>
<td>132</td>
<td>48</td>
</tr>
<tr>
<td>anderson</td>
<td>N</td>
<td>1.5</td>
<td>114</td>
<td>13</td>
</tr>
<tr>
<td>lamport*</td>
<td>N</td>
<td>ofm</td>
<td>ofm</td>
<td>49</td>
</tr>
<tr>
<td>elevator 2</td>
<td>Y</td>
<td>0.9</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>phils*</td>
<td>Y</td>
<td>&lt; 0.1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>peterson 2</td>
<td>Y</td>
<td>4.5</td>
<td>188</td>
<td>20</td>
</tr>
<tr>
<td>bakery*</td>
<td>Y</td>
<td>0.1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>brp*</td>
<td>Y</td>
<td>ofm</td>
<td>ofm</td>
<td>1</td>
</tr>
</tbody>
</table>

- **CUDA time** includes the initialisation phase (**Init time**)
- on-the-fly property was not incorporated in the multi CUDA algorithm – whole state space had to be generated
- two times slowdown of dual CUDA computation
### Employing Multiple CUDA Devices

<table>
<thead>
<tr>
<th>Model</th>
<th>acc. cycle</th>
<th>single CUDA</th>
<th>dual CUDA</th>
<th>1st method</th>
<th>2nd method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CUDA time</td>
<td># kernel calls</td>
<td># synchs</td>
<td>CUDA time</td>
</tr>
<tr>
<td>elevator 1</td>
<td>N</td>
<td>6.0</td>
<td>459</td>
<td>145</td>
<td>14.3</td>
</tr>
<tr>
<td>leader</td>
<td>N</td>
<td>0.1</td>
<td>3</td>
<td>4</td>
<td>0.9</td>
</tr>
<tr>
<td>peterson 1</td>
<td>N</td>
<td>3.5</td>
<td>132</td>
<td>48</td>
<td>6.4</td>
</tr>
<tr>
<td>anderson</td>
<td>N</td>
<td>1.5</td>
<td>114</td>
<td>13</td>
<td>2.5</td>
</tr>
<tr>
<td>lamport*</td>
<td>N</td>
<td>ofm</td>
<td>ofm</td>
<td>49</td>
<td>29.5</td>
</tr>
<tr>
<td>elevator 2</td>
<td>Y</td>
<td>0.9</td>
<td>42</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>phils*</td>
<td>Y</td>
<td>&lt; 0.1</td>
<td>1</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>peterson 2</td>
<td>Y</td>
<td>4.5</td>
<td>188</td>
<td>20</td>
<td>8.5</td>
</tr>
<tr>
<td>bakery*</td>
<td>Y</td>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>brp*</td>
<td>Y</td>
<td>ofm</td>
<td>ofm</td>
<td>1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- some models require significant number of synchronisations
- computation of individual iterations on particular devices involves less number of vertices – number of calls to CUDA kernels are smaller
### Employing Multiple CUDA Devices

<table>
<thead>
<tr>
<th>Model</th>
<th>acc. cycle</th>
<th>single CUDA</th>
<th>dual CUDA</th>
<th>1st method</th>
<th>2nd method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CUDA time</td>
<td># kernel calls</td>
<td># synchs</td>
<td>CUDA time</td>
</tr>
<tr>
<td>elevator 1</td>
<td>N</td>
<td>6.0</td>
<td>459</td>
<td>145</td>
<td>14.3</td>
</tr>
<tr>
<td>leader</td>
<td>N</td>
<td>0.1</td>
<td>3</td>
<td>4</td>
<td>0.9</td>
</tr>
<tr>
<td>peterson 1</td>
<td>N</td>
<td>3.5</td>
<td>132</td>
<td>48</td>
<td>6.4</td>
</tr>
<tr>
<td>anderson</td>
<td>N</td>
<td>1.5</td>
<td>114</td>
<td>13</td>
<td>2.5</td>
</tr>
<tr>
<td>lamport*</td>
<td>N</td>
<td>ofm</td>
<td>ofm</td>
<td>49</td>
<td>29.5</td>
</tr>
<tr>
<td>elevator 2</td>
<td>Y</td>
<td>0.9</td>
<td>42</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>phils*</td>
<td>Y</td>
<td>&lt; 0.1</td>
<td>1</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>peterson 2</td>
<td>Y</td>
<td>4.5</td>
<td>188</td>
<td>20</td>
<td>8.5</td>
</tr>
<tr>
<td>bakery*</td>
<td>Y</td>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>brp*</td>
<td>Y</td>
<td>ofm</td>
<td>ofm</td>
<td>1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- example of models which cannot be verified using the original CUDA algorithm due to space limitation
### Employing Multiple CUDA Devices

<table>
<thead>
<tr>
<th>Model</th>
<th>acc. cycle</th>
<th>single CUDA</th>
<th>dual CUDA</th>
<th>1st method</th>
<th>2nd method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td># kernel calls</td>
<td># synchs</td>
<td>CUDA time</td>
</tr>
<tr>
<td>elevator 1</td>
<td>N</td>
<td>6.0</td>
<td>459</td>
<td>145</td>
<td>14.3</td>
</tr>
<tr>
<td>leader</td>
<td>N</td>
<td>0.1</td>
<td>3</td>
<td>4</td>
<td>0.9</td>
</tr>
<tr>
<td>peterson 1</td>
<td>N</td>
<td>3.5</td>
<td>132</td>
<td>48</td>
<td>6.4</td>
</tr>
<tr>
<td>anderson</td>
<td>N</td>
<td>1.5</td>
<td>114</td>
<td>13</td>
<td>2.5</td>
</tr>
<tr>
<td>lamport*</td>
<td>N</td>
<td>ofm</td>
<td>ofm</td>
<td>49</td>
<td>29.5</td>
</tr>
<tr>
<td>elevator 2</td>
<td>Y</td>
<td>0.9</td>
<td>42</td>
<td>2</td>
<td>2.1</td>
</tr>
<tr>
<td>phil*</td>
<td>Y</td>
<td>&lt; 0.1</td>
<td>1</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>peterson 2</td>
<td>Y</td>
<td>4.5</td>
<td>188</td>
<td>20</td>
<td>8.5</td>
</tr>
<tr>
<td>bakery*</td>
<td>Y</td>
<td>0.1</td>
<td>1</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>brp*</td>
<td>Y</td>
<td>ofm</td>
<td>ofm</td>
<td>1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- 2nd method needs more time for initial phases
- total times spent on CUDA computation are very similar
- relative size of CSR representation in 2nd method is significantly smaller – leads to better space efficiency
Space Efficiency of Two Proposed Partitioning

- illustrate ability to efficiently utilise space when increasing number of GPU devices is employed
- average over all tested models
- 2nd method is significantly better for partitioning of a wide variety of graphs
Comparison of Overall Run-times and Speedup

Overall run-times and speedup from [ICPADS’09]

<table>
<thead>
<tr>
<th>Models</th>
<th>CUDA MAP</th>
<th>CPU MAP</th>
<th>CPU OWCTY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total time</td>
<td>total time</td>
<td>CUDA MAP speedup</td>
</tr>
<tr>
<td>non-accepting</td>
<td>276</td>
<td>1357</td>
<td>4.92</td>
</tr>
<tr>
<td>accepting</td>
<td>139</td>
<td>860</td>
<td>6.19</td>
</tr>
<tr>
<td>both</td>
<td>415</td>
<td>2173</td>
<td>5.24</td>
</tr>
</tbody>
</table>

Overall run-times and speedup from [ICPADS’10]

- performance of CUDA computation was slightly improved
- on-the-fly property was not incorporated in the multi CUDA algorithm – comparison only for models without accepting cycle

<table>
<thead>
<tr>
<th>non-accepting models</th>
<th>total time</th>
<th>CPU OWCTY</th>
<th>CPU MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time</td>
<td>-</td>
<td>1357</td>
<td>639</td>
</tr>
<tr>
<td>Single CUDA MAP</td>
<td>255</td>
<td>5.3</td>
<td>2.5</td>
</tr>
<tr>
<td>Dual CUDA MAP</td>
<td>267</td>
<td>5.1</td>
<td>2.3</td>
</tr>
</tbody>
</table>
Conclusion

Multi-core acceleration of CSR construction
- significant speed-up of expensive phase of CSR construction
- utilization of modern multi-core machines equipped with GPU device

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

Exhaustive experimental evaluation
CUDA accelerated OWCTY algorithm

- next talk by Petr Bauch

On-the-fly property in the multi CUDA algorithms

- improve the performance of multi CUDA algorithms

Combination of multi-core acceleration of CSR construction and multi CUDA algorithms

- full utilization of modern multi-core machines equipped with multiple GPU devices

Extend the experimental evaluation

- new generation of Fermi architecture
  - new memory hierarchy, doubling of the number of cores

CUDA acceleration of other graph problems

- strongly connected component decomposition
- mean weight cycles
Current and Future Work

CUDA accelerated OWCTY algorithm
  • next talk by Petr Bauch

On-the-fly property in the multi CUDA algorithms
  • improve the performance of multi CUDA algorithms

Combination of multi-core acceleration of CSR construction and multi CUDA algorithms
  • full utilization of modern multi-core machines equipped with multiple GPU devices

Extend the experimental evaluation
  • new generation of Fermi architecture
    • new memory hierarchy, doubling of the number of cores

CUDA acceleration of other graph problems
  • strongly connected component decomposition
  • mean weight cycles
Current and Future Work

CUDA accelerated OWCTY algorithm
- next talk by Petr Bauch

On-the-fly property in the multi CUDA algorithms
- improve the performance of multi CUDA algorithms

Combination of multi-core acceleration of CSR construction and multi CUDA algorithms
- full utilization of modern multi-core machines equipped with multiple GPU devices

Extend the experimental evaluation
- new generation of Fermi architecture
  - new memory hierarchy, doubling of the number of cores

CUDA acceleration of other graph problems
- strongly connected component decomposition
- mean weight cycles
Current and Future Work

CUDA accelerated OWCTY algorithm

- next talk by Petr Bauch

On-the-fly property in the multi CUDA algorithms

- improve the performance of multi CUDA algorithms

Combination of multi-core acceleration of CSR construction and multi CUDA algorithms

- full utilization of modern multi-core machines equipped with multiple GPU devices

Extend the experimental evaluation

- new generation of Fermi architecture
  - new memory hierarchy, doubling of the number of cores

CUDA acceleration of other graph problems

- strongly connected component decomposition
- mean weight cycles
Current and Future Work

CUDA accelerated OWCTY algorithm

- next talk by Petr Bauch

On-the-fly property in the multi CUDA algorithms

- improve the performance of multi CUDA algorithms

Combination of multi-core acceleration of CSR construction and multi CUDA algorithms

- full utilization of modern multi-core machines equipped with multiple GPU devices

Extend the experimental evaluation

- new generation of Fermi architecture
  - new memory hierarchy, doubling of the number of cores

CUDA acceleration of other graph problems

- strongly connected component decomposition
- mean weight cycles
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
The End

Thank you for your attention