FI MU

Faculty of Informatics
Masaryk University Brno

CUDA accelerated LTL Model Checking

by

Jiří Barnat, Luboš Brim, Milan Češka, and Tomáš Lamr

FI MU Report Series **FIMU-RS-2009-05**

 June 2009

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**

CUDA accelerated LTL Model Checking*

Jiří Barnat, Luboš Brim, Milan Češka, and Tomáš Lamr
Faculty of Informatics, Masaryk University,
Botanicka 68a, 60200 Brno, Czech Republic
{xbarnat, brim, xceska, xlamr}@fi.muni.cz

Abstract

Recent technological developments made available various many-core hardware platforms. For example, a SIMD-like hardware architecture became easily accessible for many users who have their computers equipped with modern NVIDIA GPU cards with CUDA technology. In this paper we redesign the maximal accepting predecessors algorithm [7] for LTL model checking in terms of matrix-vector product in order to accelerate LTL model checking on many-core GPU platforms. Our experiments demonstrate that using the NVIDIA CUDA technology results in a significant computation speedup.

1 Introduction

Model-checking [1] is a wide-spread technique for automated formal verification. Given a formal description of a system and desired system property, the goal of the model-checking procedure is to analyze reachable system configurations in order to decide whether the system satisfies the property or not. In this paper we deal with LTL model checking, which is the case when the property to be verified is given as a formula of Linear Temporal Logic (LTL). In LTL model checking, the question of satisfaction of the property can be reduced to the problem of detection of an accepting cycle (cycle through at least one vertex denoted as accepting vertex) in a directed graph.

*This work has been supported in part by the Czech Grant Agency grants No. 201/09/P497, 201/09/1389, 102/09/H042 and the Academy of Sciences grant No. 1ET408050503.

All the know algorithms for accepting cycle detection can be divided into two classes. Algorithms such as Nested DFS [10] or algorithms based on Tarjan's SCC decomposition algorithm [18, 17] exhibit optimal (linear) time complexity, but are incompatible with parallel processing. This is because they strongly rely on the so called depth-first search (DFS) postorder for computation of which no scalable parallel algorithm is known [16]. The other group of algorithms for accepting cycle detection are algorithms such as OWCTY [12, 9] or MAP [7] that avoid DFS postorder, but exhibit unoptimal time complexity. However, it has been demonstrated that the unoptimality is easily outweighted by parallel processing [19, 2]. As a result, the unoptimal algorithms are actually faster than the optimal sequential algorithms if contemporary parallel hardware is used.

Moreover, the graph to be analyzed tends to be very large for realistic systems and it is handled only with difficulties by a single memory-limited machine. Consequently, optimal utilization of resources of various hardware platforms have got much attention by the model checking community. As most modern hardware platforms are actually parallel platforms, the desire for full utilization of the power available rendered all sequential algorithms obsolete.

Modern graphics processing units (GPUs) have emerged as a revolutionary technological opportunity due to their tremendous massive parallelism, floating point capability, low cost, and ubiquitous presence in commodity computer systems. Many key computational kernels have been redesigned to exploit the performance of this modern hardware. The key to effective utilization of GPUs for scientific computing is the design and implementation of efficient data-parallel algorithms that can scale to hundreds of tightly coupled processing units.

In this paper we show how one of the parallel algorithms for accepting cycle detection, namely the MAP algorithm, can be effectively accelerated on GPU if the input data are given in an appropriate format. We demonstrate that the GPU platform has enough potential to significantly outperform non-GPU computation.

The rest of the paper is organized as follows. In Section 2 we briefly recall basics of automata-theoretic approach to LTL Model Checking. Sections 3, and 6 describe how we adapted the algorithm MAP to GPU processing and what enhancements we did to achieve good performance. In Sections 5 we recapitulate NVIDIA CUDA hardware platform a show how our adapted algorithm can be implemented on CUDA. Section 7 reports on an experimental evaluation of our approach, and finally, Section 8 summarizes achieved results and plots some future directions.

2 LTL Model Checking

For LTL model checking purposes, the system to be analyzed has to be described in some modeling language, `ProMeLa` [14] for example, and the property to be checked has to be given as formula of Linear Temporal Logic (LTL) [1]. To answer the LTL model checking question, tools, such as `SPIN` [14] or `DiVinE` [5], employ automata-theoretic approach to reduce the model checking problem to the problem of non-emptiness of Büchi automata. In particular, the model of a system S is viewed as a finite automaton A_S describing all possible behaviors of the system. The property to be checked (LTL formula φ) is negated and translated into Büchi automaton $A_{\neg\varphi}$ describing all the behaviors violating φ. In order to check whether the system violates φ, a synchronous product $A_S \times A_{\neg\varphi}$ of A_S and $A_{\neg\varphi}$ is constructed describing those behaviors of the system that violates φ, i.e. $L(A_S \times A_{\neg\varphi}) = L(A_s) \cap L(A_{\neg\varphi})$. The automata A_S, $A_{\neg\varphi}$, and $A_S \times A_{\neg\varphi}$ are referred to as *system*, *property*, and *product* automata, respectively. System S satisfies formula φ if and only if the language of the product automaton is empty, which is if and only if there is no reachable accepting cycle in the underlying graph of the product automaton. The LTL model checking problem is thus reduced to the problem of the detection of an accepting cycle in the product automaton graph.

There are several parallel algorithms for accepting cycle detection. One of them is the algorithm MAP [7] which we now briefly introduce in its "successor" version. Let $G = (V, E, v_0, \mathcal{A})$ be the graph of the product automaton, where V is a finite set of vertices, E is a set of edges, v_0 is an initial vertex, and $\mathcal{A}$ is a vertex predicate indicating whether a state is accepting or not. Let $<$ be a linear ordering of the set of vertices, given e.g. by the vertex numbering. We extend the ordering to the set $V \cup \{\bot\}$ ($\bot \notin V$) and put $\bot < v$ for all $v \in V$. Furthermore, let $map :: V \to V \cup \{\bot\}$ is a function returning the maximal accepting successor of a given vertex or $\bot$ if it does not exist, i.e. $map(u) = \max\{\bot, v \mid (u, v) \in E^+ \wedge \mathcal{A}(v)\}$.

The idea of the algorithm to detect an accepting cycle is as follows. If a vertex u is its own maximal accepting successor, i.e. $u = map(u)$, the presence of an accepting cycle is guaranteed. If there is an accepting cycle in the graph, but for none of its vertices $u = map(u)$, then the maximal accepting successor of all the vertices of the cycle must be the same, must lie outside the cycle and can thus be marked as non-accepting. The idea of the algorithm is to process the graph in a few iterations so that each iteration computes map values for all the vertices. If no accepting cycle is discovered, all maximal accepting successors that occur in $map(u)$ for some u are marked as non-accepting for all the following iterations. The algorithm

Algorithm 1 Algorithm MAP

Input: directed graph $G = (V, E, v_0, \mathcal{A})$ of $A_{S\times\neg\varphi}$
linear ordering $<$ on V

Output: `true`, if $A_{S\times\neg\varphi}$ contains accepting cycle
`false`, otherwise

1: **while** $(\exists v \in V : \mathcal{A}(v) = \texttt{true})$ **do**
2: COMPUTEALLMAPS(G,<)
3: **if** $(\exists u \in V : u = map(u))$ **then**
4: **return** `true`
5: **end if**
6: $\mathcal{A}(v) \leftarrow$ `false`
7: **end while**
8: **return** `false`

iterates until an accepting cycle is found or the set of accepting vertices becomes empty. See the pseudo-code in Algorithm 1.

A key procedure of the algorithm is COMPUTEALLMAPS() that is responsible for computing the values of the function map for all the vertices reachable from the initial vertex. Initially, the values of $map(u)$ are set to $\perp$ for all $u \in V$. These values are then repeatedly updated until a global fix-point is reached, i.e. no update can be done for any value of $map(u)$. Suppose a directed edge (u, v) from u to v, the new value of $map(u)$, the so called *update* along the edge (u, v), is computed using function $maxacc(u, v)$ as follows:

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

Henceforward, we also refer to the iterations of the while loop of Algorithm MAP given in Algorithm 1 as of *outer* iterations, and the iterations of the while loop of procedure COMPUTEALLMAPS given in Algorithm 2 as of *inner* iterations.

The practical performance of the basic algorithm may be further enhanced if the graph to be checked for the presence of an accepting cycle is partitioned into subgraphs so that no cycle of the original graph maps to multiple partitions. In that case the *inner* iterations as performed in procedure COMPUTEALLMAPS may be prevented from propagating values of map along

Algorithm 2 COMPUTEALLMAPS(G,<)

Input: directed graph $G = (V, E, v_0, \mathcal{A})$
linear ordering $<$ on V

Output: value of $map(u)$ for all $u \in V$

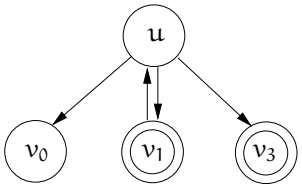
```
1: for all (u ∈ V) do
2:    map(u) =⊥
3: end for
4: while (¬ fix-point) do
5:    for all ((u, v) ∈ E) do
6:       if (v ∈ A) then
7:          map(u) ← max{map(u), map(v), v}
8:       else
9:          map(u) ← max{map(u), map(v)}
10:      end if
11:   end for
12: end while
13: return map
```

edges that cross partition boundaries. This brings no complexity improvement, but it generally reduces the number of inner iterations needed to achieve the fix-point.

One technique to partition the product automaton graph is part of the algorithm itself. It builds upon the fact that if two vertices differ in their values of map, they cannot lie on the same cycle. Therefore, the propagation in procedure COMPUTEALLMAPS may be localized to those edges (u, v) for which the values of $map(v)$ and $map(u)$ computed in the previous outer iteration are the same. The values of map function from the previous outer iteration are referred to as $oldmap$ values.

3 Reformulation of MAP algorithm

In order to accelerate the MAP algorithm on CUDA we reformulate it as a matrix-vector multiplication algorithm.



$$
\begin{array}{c}
\begin{array}{cc} & \begin{array}{cccc} v_0 & v_1 & & v_3 \end{array} \\ u & \left(\begin{array}{cccc} 0&0&0&0 \\ 0&0&1&0 \\ 1&1&0&1 \\ \hline 0&0&0&0 \end{array}\right) \end{array}
\begin{array}{c} \begin{array}{ccc} \vec{m} & \vec{o} & \vec{\mathcal{A}} \end{array} \\ \times \\ \begin{array}{ccc} \bot & \bot & 0 \\ \bot & \bot & 1 \\ \bot & \bot & 0 \\ \bot & \bot & 1 \end{array} \end{array}
= \begin{array}{c} \begin{array}{ccc} \vec{m} & \vec{o} & \vec{\mathcal{A}}\,\vec{r} \end{array} \\ \begin{array}{cccc} \bot & \bot & 0 & 0 \\ \bot & \bot & 1 & 0 \\ 3 & \bot & 0 & 1 \\ \bot & \bot & 1 & 0 \end{array} \end{array}
\qquad
\begin{array}{ccc} m & o & \mathcal{A} \\ \bot & \bot & 0 \\ 3 & \bot & 1 \\ 3 & \bot & 0 \\ \bot & \bot & 1 \end{array}
\Longrightarrow
\begin{array}{ccc} m & o & \mathcal{A} \\ \bot & \bot & 0 \\ 3 & \bot & 1 \\ 3 & \bot & 0 \\ \bot & 3 & 0 \end{array}
\Longrightarrow
\begin{array}{ccc} m & o & \mathcal{A} \\ \bot & \bot & 0 \\ \bot & 3 & 1 \\ \bot & 3 & 0 \\ \bot & 3 & 0 \end{array}
\\[2ex]
\begin{array}{cc} & \begin{array}{cccc} v_0 & v_1 & & v_3 \end{array} \\ v_1 & \left(\begin{array}{cccc} 0&0&0&0 \\ 0&0&1&0 \\ \hline 1&1&0&1 \\ 0&0&0&0 \end{array}\right) \end{array}
\begin{array}{c} \begin{array}{ccc} \vec{m} & \vec{o} & \vec{\mathcal{A}} \end{array} \\ \times \\ \begin{array}{ccc} \bot & \bot & 0 \\ \bot & \bot & 1 \\ 3 & \bot & 0 \\ \bot & \bot & 1 \end{array} \end{array}
= \begin{array}{c} \begin{array}{ccc} \vec{m} & \vec{o} & \vec{\mathcal{A}}\,\vec{r} \end{array} \\ \begin{array}{cccc} \bot & \bot & 0 & 0 \\ 3 & \bot & 1 & 1 \\ 3 & \bot & 0 & 0 \\ \bot & \bot & 1 & 0 \end{array} \end{array}
\qquad
\left(\begin{array}{cccc} 0&0&0&0 \\ 0&0&1&0 \\ \hline 1&1&0&1 \\ 0&0&0&0 \end{array}\right)
\times
\begin{array}{ccc} \bot & \bot & 0 \\ \bot & 3 & 1 \\ \bot & 3 & 0 \\ \bot & 3 & 0 \end{array}
= \begin{array}{ccc} \bot & \bot & 0\,0 \\ \bot & 3 & 1\,1 \\ \hline 1 & 3 & 0\,0 \\ \bot & 3 & 0\,0 \end{array}
\\[2ex]
\left(\begin{array}{cccc} 0&0&0&0 \\ 0&0&1&0 \\ 1&1&0&1 \\ 0&0&0&0 \end{array}\right)
\times
\begin{array}{ccc} \bot & \bot & 0 \\ 3 & \bot & 1 \\ 3 & \bot & 0 \\ \bot & \bot & 1 \end{array}
= \begin{array}{ccc} \bot & \bot & 0\,0 \\ 3 & \bot & 1\,0 \\ 3 & \bot & 0\,0 \\ \bot & \bot & 1\,0 \end{array}
\qquad
\left(\begin{array}{cccc} 0&0&0&0 \\ \hline 0&0&1&0 \\ \hline 1&1&0&1 \\ 0&0&0&0 \end{array}\right)
\times
\begin{array}{ccc} \bot & \bot & 0 \\ \bot & 3 & 1 \\ 1 & 3 & 0 \\ \bot & 3 & 0 \end{array}
= \begin{array}{ccc} \bot & \bot & 0\,0 \\ \hline 1 & 3 & 1\,0 \\ \hline 1 & 3 & 0\,0 \\ \bot & 3 & 0\,0 \end{array}
\end{array}
$$

Figure 1. Matrix vector computation of MAP.

Let us assume the graph G of the product automaton $A_{S\times\neg\varphi}$ is represented as an adjacency matrix M. The matrix keeps value 1 at row u and column v for every directed edge (u, v). See Figure 1. Additional data to be stored with every vertex of the graph are not stored directly in the matrix, but they are rather organized in separate vectors. Namely, vector $\vec{m}$ of *map* values, vector $\vec{o}$ of *oldmap* values, vector $\vec{\mathcal{A}}$ of values of the predicate $\mathcal{A}$, and an output vector $\vec{r}$ of bits indicating a recent update to the value of map. Vector $\vec{r}$ is used to detect the fix-point of computation of inner iterations, i.e. the situation when no update to map function occurs in two successive inner iterations.

The algorithm proceeds as illustrated in Figure 1. Initially, all *map* and *oldmap* values are set to $\bot$, see the column vectors $\vec{m}$ and $\vec{o}$. The algorithm repeatedly updates values of the vector $\vec{m}$ until a fix-point is reached (the three topmost matrix-vector product equations). Since $map(v) \neq v$ for all vertices v, the maximal accepting vertex v_3 cannot be part of an accepting

cycle ($map(v_3) < v_3$). The algorithm resets its accepting status, copies map values to $oldmap$ values and sets all values of map to $\perp$. Note that there are two subgraphs to be further processed identified by the $oldmap$ values. Vertex v_3 is part of one of the subgraphs, but since it is not accepting anymore, it cannot influence any future values of map computed for vertices within the subgraph. Then the next outer iteration proceeds. The algorithm detects (using two inner iterations) that $map(v_1) = v_1$ and so it terminates reporting accepting cycle through vertex v_1.

The key observation is that the vector of map values computed in each inner iteration is computed as a matrix-vector product by substituting $\texttt{max}$ for the standard $+$ operation, and $\texttt{maxacc}$ for the standard $\cdot$ operation. The result, however, considers only those summands for which $oldmap$ values are the same as $oldmap$ value of the updated vertex. For example, the resulting value of $\vec{m}[u]$ is computed as

$$\vec{m}[u] = max_{1<i\leq 4} \left(M[u][i] \cdot \texttt{old} \cdot \texttt{maxacc}(i, u)\right)$$

where $\texttt{old}$ equals to 1 if $\vec{o}[u] = \vec{o}[i]$ and equals to 0 in the other case. Also note that values of $\vec{A}$ are accessed within $\texttt{maxacc}$ operation and that 0 is used to encode $\perp$.

4. CUDA Architecture

The Compute Unified Device Architectures (CUDA) [11], developed by NVIDIA, is parallel programing model and software environment providing general purpose programming on Graphics Processing Units. At the hardware level, GPU device is a collection of multiprocessors each consisting of eight scalar processor cores, instruction unit, on-chip shared memory, and texture and constant memory caches. Every core has a large set of local 32-bit registers but no cache. See the structure as depicted in Figure 2. The multiprocessors follow the SIMD architecture, i.e. they concurrently execute the same program instruction on different data. Communication among multiprocessors is realized through the shared device memory that is accessible for every processor core.

On the software side, the CUDA programming model extends the standard C/C++ programming language with a set of parallel programming supporting primitives. A CUDA program consist of a *host* code running on a CPU and a *device* code running on the GPU. The device code is structured into so called *kernels*. A kernel executes the same scalar sequential program in many *data independent parallel threads*. Within the kernel, threads are organized into

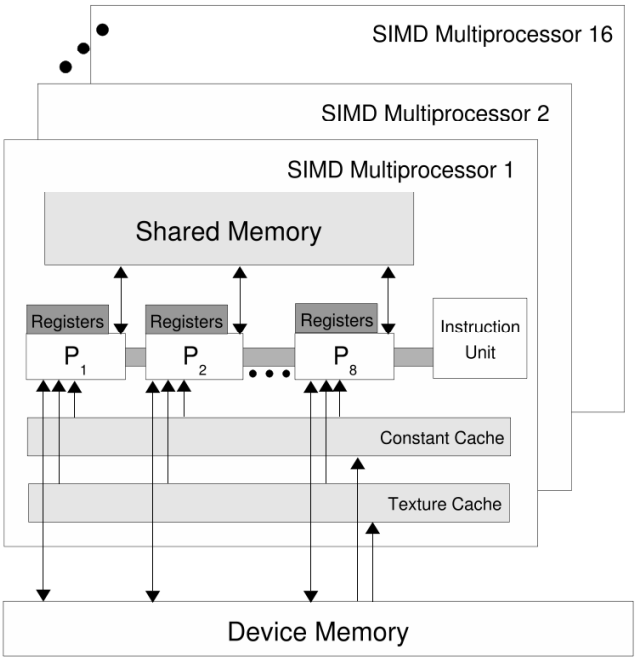


Figure 2. CUDA hardware model

thread blocks forming a grid of one or more blocks, see Figure 3. Each thread is given a unique index within its block *threadIdx* and each block is given a unique index *blockIdx* within the grid. The threads of a single block are guaranteed to be executed on the same multiprocessor, thus, they can easily access data stored in shared memory of the multiprocessor. The programmer specifies both the number of blocks and number of threads per block to be created before a kernel is launched. These values are available to the kernel as *gridDim* and *blockDim* values, respectively.

Using CUDA to accelerate the computation is easily exemplified on a vector summation problem. Suppose two vectors of length n to be summed. In the standard imperative programming language, a programmer would use a for loop to sum individual vector elements successively. Using CUDA, however, the vector elements can be summed concurrently in a single *kernel* call populated with n threads, each responsible for summation of a single pair of vector elements at the position given be the thread index.

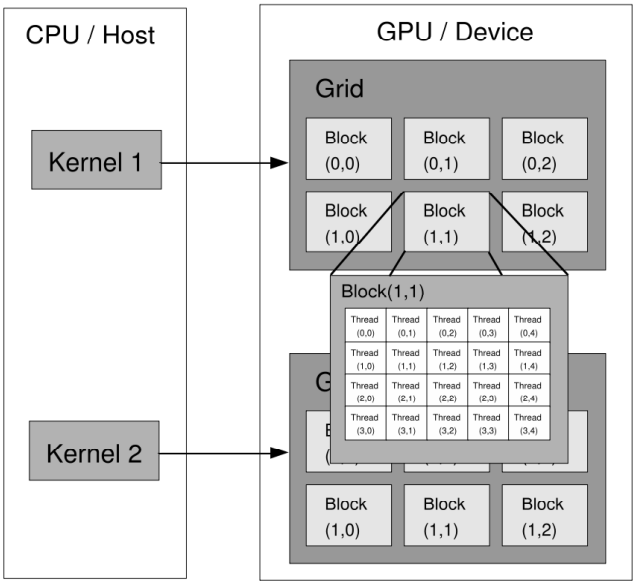


Figure 3. CUDA programming model

$$M = \begin{pmatrix} a & b & c & 0 \\ 0 & d & e & 0 \\ 0 & 0 & 0 & 0 \\ f & 0 & g & h \end{pmatrix}$$

Row 0 Row 1 Row 3

$$M_r[8] = \{\, a\ b\ c\ d\ e\ f\ g\ h \,\}$$

$$M_c[8] = \{\, 0\ 1\ 2\ 1\ 2\ 0\ 2\ 3 \,\}$$

$$M_n[4] = \{\, 0\ 3\ 5\ 8 \,\}$$

Figure 4. A sparse matrix and its CSR representation.

5. CUDA Accelerated Algorithm MAP

Data structures used for CUDA accelerated computation must be designed with care. First, they have to allow independent thread-local data processing so that the CUDA hardware can employ massive parallelism. And second, they have to be small so that the high latency device-memory access and limited device-memory bandwidth are not large performance bottlenecks. As for the algorithm MAP, it is the matrix representation of the graph $A_{S\times\neg\varphi}$ to be encoded appropriately at the first place. Note that uncompressed matrix or dynamically linked adjacency lists violate the requirements and as such they are inappropriate for CUDA computing.

We decided to encode the matrix of the product automaton graph as a sparse matrix using *compressed sparse row* (CSR) format. In this format a sparse matrix is encoded using three one-dimensional arrays M_r, M_c, and M_n as follows. All the non-zero elements of a matrix M

Algorithm 3 CUDA MAP Algorithm - host code

Input: directed graph $G = (V, E, v_0, \mathcal{A})$ of $A_{S\times\neg\varphi}$

Output: `true`, if $A_{S\times\neg\varphi}$ contains accepting cycle
`false`, otherwise

1: CREATE_CSR_REPRESENTATION$(G, M_c, M_n, \vec{m})$
2: *acc_cycle_found* $\leftarrow$ `false`
3: *repropagate* $\leftarrow$ `true`
4: *unmarked* $\leftarrow$ `true`
5: **copy** $(M_c, M_n, \vec{m})$ **to** GPU $(gM_c, gM_n, g\vec{m})$
6: **while** *unmarked* $\wedge\neg$*acc_cycle_found* **do**
7: **while** *repropagate* $\wedge\neg$*acc_cycle_found* **do**
8: *repropagate* $\leftarrow$ `false`
9: *map_Kernel*$(gM_c, gM_n, g\vec{m}, $*acc_cycle_found*$)$
10: *check_repropagate_Kernel*$(g\vec{m}, $*repropagate*$)$
11: **end while**
12: *unmarked* $\leftarrow$ `false`
13: *unmark_acc_vertices_Kernel*$(g\vec{m}, $*unmarked*$)$
14: **end while**
15: **return** *acc_cycle_found*

are stored in the array M_r in left-to-right and top-to-bottom order. The array M_c keeps the corresponding column indices for every element in M_r, while the array M_n keeps positions of first elements of rows of M in arrays M_n and M_r. See Figure 4. Note that in our case all the non-empty elements of the matrix are the same, hence, the array M_r is redundant for our purposes and we do not maintain it at all.

The other data structures are organized as vectors, which is compatible with CUDA processing. The values of *map*, *old map*, $\mathcal{A}$ predicate, and repropagation bit `r` for vertex `i` are available in the pseudo-code as `m[i].map`, `m[i].old`, `m[i].acc` and `m[i].repropagate`, respectively. Since the values of `map` and `oldmap` are technically pointers, we were able to store the two other bits of information into unused pointer bits reducing thus the space needed to record all the data for one vertex to two times 4 Bytes.

Algorithm 4 device code - map_Kernel

proc *map_Kernel*($\mathrm{gM_c}$, $\mathrm{gM_n}$, $\mathrm{g\vec{m}}$, *acc_cycle_found*)

1: *row* $\leftarrow$ blockIdx $*$ blockDim $+$ threadIdx
2: **if** *row* $<$ $|\mathrm{g\vec{m}}|$ **then**
3: *row_begin* $\leftarrow$ $\mathrm{gM_n}$[*row*]
4: *row_end* $\leftarrow$ $\mathrm{gM_n}$[*row* $+$ *1*]
5: $\mathrm{u} \leftarrow \mathrm{gm}$[*row*]
6: *propagate* $\leftarrow \bot$
7: *u.updated* $\leftarrow$ `false`
8: **for** *column* $\leftarrow$ *row_begin* **to** *row_end* $-$ 1 **do**
9: $\mathrm{v} \leftarrow \mathrm{gm}[\mathrm{gM_c}$[*column*]]
10: **if** *u.map* $=$ *v.old* $\wedge$ *u.old* $\neq$ *v.old* **then**
11: *u.old* $\leftarrow$ *v.old*
12: *u.map* $\leftarrow \quad \bot$
13: *propagate* $\leftarrow \bot$
14: *u.repropagate* $\leftarrow$ `true`
15: **break**
16: **else if** *u.old* $=$ *v.old* **then**
17: *propagate* $\leftarrow$ *max*(*propagate*, *maxacc*(u, v))
18: **end if**
19: **end for**
20: **if** *propagate* $=$ *row* **then**
21: *acc_cycle_found* $\leftarrow$ `true`
22: **end if**
23: **if** *propagate* $>$ *u.map* **then**
24: *u.map* $\leftarrow$ *propagate*
25: *u.repropagate* $\leftarrow$ `true`
26: **end if**
27: gm[*row*] $\leftarrow \mathrm{u}$
28: **end if**

end

Algorithm 5 device code - *check_repropagate_Kernel*

proc*check_repropagate_Kernel*($\mathtt{g}\vec{\mathtt{m}}$, *repropagate*)

1: *row* $\leftarrow$ $\mathtt{blockIdx} * \mathtt{blockDim} + \mathtt{threadIdx}$
2: **if** *row* $< |\mathtt{g}\vec{\mathtt{m}}|$ **then**
3: $\quad \mathtt{u} \leftarrow \mathtt{gm}[row]$
4: $\quad$ **if** *u.repropagate* **then**
5: $\quad\quad$ *repropagate* $\leftarrow$ `true`
6: $\quad$ **end if**
7: **end if**

end

Algorithm 6 device code - *unmark_acc_vertices_Kernel*

proc *unmark_acc_vertices_Kernel*($\mathtt{g}\vec{\mathtt{m}}$, *unmarked*)

1: *row* $\leftarrow$ $\mathtt{blockIdx} * \mathtt{blockDim} + \mathtt{threadIdx}$
2: **if** *row* $< |\mathtt{g}\vec{\mathtt{m}}|$ **then**
3: $\quad \mathtt{u} \leftarrow \mathtt{gm}[row]$
4: $\quad$ **if** *u.acc* $\wedge$ *u.map* $<$ *row* **then**
5: $\quad\quad$ *u.map* $\leftarrow \perp$
6: $\quad\quad$ *u.old* $\leftarrow$ *row*
7: $\quad\quad$ *u.acc* $\leftarrow$ `false`
8: $\quad\quad$ $\mathtt{gm}[row] \leftarrow \mathtt{u}$
9: $\quad\quad$ *unmarked* $\leftarrow$ `true`
10: $\quad$ **end if**
11: **end if**

end

As explained in Section 3, the major computation part of the algorithm MAP can be formulated in terms of matrix-vector product. Given CSR matrix representation and a column vector, an efficient CUDA accelerated matrix-vector product procedure was described in [13, 6]. The idea of the procedure is to map every row of the matrix to one thread. Since in our case the edges of the graph are more or less uniformly spread in the matrix, this approach leads to a satisfactory balanced load of CUDA cores.

The pseudo-code of the CUDA accelerated algorithm MAP follows. Algorithm 3 lists the overall host code, i.e. the part that is executed on the CPU. The inner and outer while loops listed in the pseudo-code correspond with the inner and outer iterations as introduced in Section 2.

There are three kernel functions called from the host code. The most important one, *map_Kernel*, is listed as Algorithm 4. Every call to *map_Kernel* performs one matrix-vector product operation, i.e. it propagates the $\mathtt{map}$ values once along every edge (see lines 17 and 23-26 of Algorithm 4). Note however, that the very first call to the kernel in every outer loop does a slightly different job. In particular, it copies $\mathtt{map}$ values to $\mathtt{oldmap}$ values to decompose the graph according to $\mathtt{map}$ values from the previous outer iteration (see lines 10-15 of Algorithm 4). If no accepting cycle is found and *map_Kernel* returns, *check_repropagate_Kernel* is called to detect a fix-point. *check_repropagate_Kernel* is listed as Algorithm 5. If there is no $\mathtt{map}$ value to be further propagated, the outer iteration is completed by a call to *unmark_acc_vertices_Kernel* to unset accepting predicate for accepting states proven to be outside an accepting cycle. *unmark_acc_vertices_Kernel* is listed as Algorithm 6.

6. On-The-Fly Verification

The last not-yet-discussed but quite essential procedure of the whole verification process is the transformation of the input data as given to the model checker into the form suitable for CUDA accelerated computation. In the model checking process, the graph to be searched for accepting cycles is given implicitly. Implicit definition of a graph involves a function to enumerate initial vertices, a function to enumerate edges emanating from a given vertex, and a function to check for accepting status of a given vertex. In order to use our CUDA accelerated accepting cycle detection algorithm, we have to turn the implicit definition of the graph into an explicit one. This process is generally referred to as *state space generation*. In addition to explicit state space construction we also build its CSR representation.

A distinguished property of the MAP algorithm is that it can be altered to work on-the-fly [7]. An on-the-fly algorithm can detect the presence of an accepting cycle before the state space generation procedure completes its task. We were able to adapt our implementation to mimic this behavior as well. In particular, we let the CPU to perform state space generation during which we let the GPU to apply CUDA accelerated MAP algorithm on partially con-

Models	Model description	Inspected LTL properties
elevator	the elevator controller	1: if level 1 is requested, it is served eventually
		2: if level 1 is requested, it is served as soon as the cab passes the level 1
peterson	Peterson's mutual exclusion algorithm	1: infinitely many times someone is in the critical section
		2: if process 0 is not in the critical section then it will eventually reach it
leader	leader election algorithm based on filters	eventually leader will be elected
anderson	Anderson's queue lock mutual exclusion algorithm	for each process holds that if the process is active infinitely often then it is in the critical section infinitely often
bakery	Bakery mutual exclusion algorithm	for each process holds that if the process is active infinitely often and starts wait then it waits until reaches the critical section and eventually reaches it
phils	dining philosophers problem	infinitely many times someone eats

Table 1. The used experimental models.

structed graph. If the part of the graph constructed so far contains an accepting cycle, CUDA accelerated MAP algorithm simply reveals it before the state space generation is complete.

To further accelerate CUDA computation, we employed another technique to decompose the product automaton graph [15, 4]. The idea is to decompose the property automaton into strongly connected components and then project this decomposition to the final graph. Moreover some parts of the product automata graph are known to be without accepting vertices in advance and may be omitted when constructing CSR representation of the graph. This technique significantly reduced the size of the matrix as well as number of repropagations needed.

7. Experimental evaluation

We have implemented the algorithm as a part of the DiVinE-Cluster model checker version 0.8.2 [5]. We compared the performance of the CUDA implementation against the algorithms MAP and OWCTY as provided by the model checker. To order vertices as required by the algo-

Model	*# generated states*	*# stored states*	*# generated transitions*	*# stored transitions*	*accepting cycle*	*# MAP iterations*	*# kernel executions*	*avg. kernel time [ms]*
elevator 1	5 015 528	1 722 344	63 110 616	20 483 544	N	14	539	18
leader	26 302 351	26 302 351	84 124 038	84 124 038	N	2	3	58
peterson 1	18 995 033	9 497 514	124 897 292	41 457 112	N	10	160	40
anderson	10 728 476	6 170 260	46 795 735	26 328 440	N	4	223	27
elevator 2	6 645 826	3 354 971	76 052 914	32 562 797	Y	1	42	22
phils	6 976 798	2 278 932	63 492 002	7 470 054	Y	1	1	12
peterson 2	5 797 524	2 933 213	38 297 450	12 943 640	Y	4	525	11
bakery	6 986 289	4 333 229	37 438 316	18 145 482	Y	1	1	23

Table 2. The statistic of CUDA MAP algorithm.

		CUDA MAP			CPU MAP				CPU OWCTY	
Model	*accepting cycle*	*CSR time*	*CUDA time*	*total time*	*1st iter. time*	*other iter. time*	*total time*	*# iter.*	*reachability time*	*total time*
elevator 1	N	26	7	34	44	56	100	16	24	41
leader	N	87	1	90	97	600	697	17	90	297
peterson 1	N	105	6	113	175	270	445	16	110	188
anderson	N	31	7	39	64	51	115	5	33	113
elevator 2	Y	33	1	35	50	–	50	1	41	177
phils	Y	45	1	47	295	102	397	5	180	576
peterson 2	Y	25	5	31	173	–	173	1	114	404
bakery	Y	24	1	26	240	–	240	1	219	907

Table 3. The run-times in seconds.

Models	CUDA MAP	CPU MAP		CPU OWCTY	
	total time	*total time*	*CUDA MAP speedup*	*total time*	*CUDA MAP speedup*
non-accepting	276	1357	4.92	639	2.32
accepting	139	860	6.19	2064	14.87
both	415	2173	5.24	2730	6.51

Table 4. The overall run-times in seconds, and speedup of the whole model checking procedure.

rithm, we employed inverse ordering on row numbers since with this ordering the numbers of inner iterations were very small. For the details on how the ordering influences performance of the algorithm, see [8].

To compare the CUDA algorithm with the existing algorithms implemented in the DiVinE Cluster model checker, we used DiVinE native models as listed in Table 1. All the experiments were run on a Linux workstation equipped with two AMD Phenom(tm) II X4 940 Processors @ 3MHz, 8 GB DDR2 @ 1066 MHz RAM and NVIDIA GeForce GTX 280 GPU with 1GB of GPU memory.

Table 2 captures various statistics of our experiments. The difference between *stored* and *generated* states illustrates how much of the state space is made of subgraphs without accepting states. Note that if the graph contains an accepting cycle, the reported numbers refer to numbers of states and transitions generated and stored before the accepting cycle was discovered. *#MAP iterations* reports the number of outer iterations, *#kernel executions* gives the total number of calls to CUDA kernels, and *avg kernel time* gives an average time a single call to a CUDA kernel took.

Table 3 provides details on run-times of individual algorithm parts. As for the CUDA MAP algorithm, the total run-time includes the initialization time (not reported in the table), CSR construction time (*CSR time*), and time spent on CUDA computation (*CUDA time*). Note that the first iteration of CPU MAP is actually slower than construction of the CSR representation. This is because the first iteration of the CPU MAP not only generates the state space, but also computes first stable values of *map*. Also note the different number of outer iterations in CUDA MAP (reported in Table 2) and CPU MAP. The difference is a result of employing maximal accepting predecessors in CPU MAP and maximal accepting successors in CUDA MAP.

The number of iterations of CUDA MAP is consistently smaller, for which we have no good explanation yet. Algorithms MAP and OWCTY were running on a single core.

Finally, Table 4 gives a comparison of overall run-times for both valid and invalid model checking instances. We can see that if the whole model checking procedure is considered, the speedup is not that impressive. This is obviously due to the CSR representation preparation. Though, the speedup is still significant.

8. Conclusions

We demonstrated successful reformulation of the LTL model checking algorithm MAP in terms of matrix-vector product that allows for significant GPU accelerated model checking process. The main bottleneck of the whole approach is the costly procedure of preparation of data structures that are necessary for efficient acceleration. Though we put significant effort in designing accelerated CSR representation computation, we did not achieve a procedure with consistent speed-up. Therefore, we consider GPU accelerating of the data structures preparation to be the next challenge for model checking community.

We are aware of other representations that could be used for CUDA efficient matrix-vector product, the other representations even exhibit better CUDA performance, however, their preparation is generally more complex, hence not very suitable for our domain.

In the future we would like to accelerate slow CSR representation preparation at least by means of multi-core processing, which we believe may bring similar speed-up as in the case of state space generation [3]. Another problem we are aware of if the limited memory size of a single CUDA device. We intend to overcome this limit by employing multiple CUDA devices for which we already have some initial thoughts.

References

[1] C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[2] J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *Model Checking Software (SPIN 2007)*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[3] J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

[4] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *Proceedinfs of the 3rd International Workshop on Verification and Computational Logic (VCL'02)*, pages 1–10. University of Southampton, UK, Technical Report DSSE-TR-2002-5 in DSSE, 2002.

[5] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification (CAV 2006)*, volume 4144/2006 of *LNCS*, pages 278–281. Springer, 2006.

[6] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[7] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

[8] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. *ENTCS*, 132(2):3–18, 2006.

[9] I. Černá and R. Pelánek. Distributed Explicit Fair Cycle Detection (Set Based Approach). In *Model Checking Software (SPIN'03)*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[10] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.

[11] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 2.0,. `http://www.nvidia.com/object/cuda_develop.html`, June 2009.

[12] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.

[13] M. Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th annual conference on Design automation (DAC'08)*, pages 2–6. ACM, 2008.

[14] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[15] A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical Report 00176, Institut für Informatik, University Freiburg, Germany, July 2002.

[16] J. Reif. Depth-first search is inherrently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[17] S. Schwoon and J. Esparza. A Note on On-The-Fly Verification Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.

[18] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, Januar 1972.

[19] K. Verstoep, H. Bal, J. Barnat, and L. Brim. Efficient Large-Scale Model Checking. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2009)*. IEEE, 2009.