Designing Fast LTL Model Checking Algorithms for Many-Core GPUs

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

Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic

---

Abstract

Recent technological developments made various many-core hardware platforms widely accessible. These massively parallel architectures have been used to significantly accelerate many computation demanding tasks. In this paper we show how the algorithms for LTL model checking can be redesigned in order to accelerate LTL model checking on many-core GPU platforms. Our detailed experimental evaluation demonstrates that using the NVIDIA CUDA technology results in a significant speedup of the verification process. Together with state space generation based on shared hash-table and DFS exploration, our CUDA accelerated model checker is the fastest among state-of-the-art shared memory model checking tools.

The effective utilisation of CUDA technology, however, is quite often reduced by the costly preparation of suitable data structures and limited to small or middle-sized instances due to space limitations, which is also the case of our CUDA-aware LTL Model Checking solutions. Hence, we further suggest how to overcome these limitations by multi-core construction of the compact data structures and by employing multiple CUDA devices for acceleration of fine-grained communication-intensive parallel algorithms for LTL Model Checking.

Keywords:

Parallel Model Checking; Linear Temporal Logic; Massively Parallel Architectures; CUDA Technology; Multiple CUDA Devices

---

1. Introduction

Model checking [1] is a wide-spread technique for automated formal verification of software and hardware systems. For a given formal description (finite-state model) of a system and desired system property, the goal of the model checking procedure is to systematically analyse all reachable configurations in order to decide whether the model satisfies the property or not. The model checking technique generally suffers from the so called *state space explosion problem* that makes a wide gap between the complexity of systems the current model checking tools can handle and the complexity of systems built in practice. As a result, the applicability of the model checking method to large, hence realistic, industrial systems is rather limited unless additional techniques are employed.

Several clever techniques have thus been introduced to fight the state space explosion problem in model checking. The most successful techniques are the partial order reduction [2, 3] and symbolic representation [4] that both aim at the reduction of the computational resources needed for a verification task. However, in the last decade a different approach to fight the state space explosion problem has attracted the model checking community – using multi-core or multiple computers for parallel and distributed model checking. The primary goal here is to extend the available memory to handle larger verification problems. Nevertheless, generating and analysing large state spaces calls for parallel algorithms in order to obtain the desired level of performance.

It is the recent shift in the hardware architecture design towards multi-cores with large amounts of local RAM that has intensified research pertaining to *shared memory* paradigm. Multi-core extension to the existing sequential model checker SPIN [5] has been introduced [6], the LTSmin tool [13] now uses a multi-core version of the nested depth-first search [12] and there has also been a dedicated multi-core branch [7, 8] of parallel model checker DiVinE [10]. For share memory architectures, linear speedup of model checking is the primary goal [11]. Besides multi-core and multi-CPU systems, many-core hardware accelerators have received a lot of attention as well. Modern Graphics Processing Units (GPU), in particular, have emerged as a revolutionary technological opportunity in recent years. Due to their tremendous massive parallelism, floating point capability, low cost, and ubiquitous presence in commodity computer systems they became a popular mean of parallel acceleration of many computationally demanding tasks. NVIDIA's CUDA technology [15] has started an avalanche of prototypes and research results demonstrating application of massive parallelism in many scientific fields, model checking being not an exception. Recent model checking research results related to the use of CUDA technology describe CUDA accelerated state space generation [16, 17, 18], model checking of probabilistic systems [19], or CUDA accelerated Linear Temporal Logic (LTL) Model Checking [20].

1.1. Contribution

This journal paper provides a survey of latest advancements in GPU acceleration of LTL model checking including, among others, our own results recently published in several conference and workshop proceedings [20, 21, 22, 23, 24] and exhaustive experimental evaluation. The paper recapitulates how accepting cycle detection algorithms MAP [25] and OWCTY [26] can be accelerated by means of massive parallel processing in order to speedup the LTL model checking procedure. It also describes techniques enabling efficient usage of multiple GPUs for acceleration of a single model checking task as well as techniques that allow for parallel multi-core preprocessing of input data into the form required by the GPU devices for an efficient computation.

The specific contributions of this journal paper are manyfold: first, a new common-ground presentation of our previously published results, second, an extension of CUDA accelerated OWCTY algorithm allowing for an efficient utilisation of multiple GPU devices, and third, a novel use of state space generation based on shared hash-tables and DFS-bases exploration combined with GPU accelerated accepting cycle detection. Moreover, we report on a completely new set of experiments providing detailed evaluation of all the techniques using a pair of new NVIDIA's Fermi architecture based GPUs. The experimental evaluation has been extended with a completely new set of experiments that position our GPU-accelerated DiVinE-CUDA as the fastest among the state-of-the-art parallel model checkers using an unbiased selection of model checking instances. The experiments demonstrate even better parallel efficiency and end-to-end performance than reported in previous publications which can be partially attributed to the use of shared hash-table and DFS-based exploration in state space generation.

2. LTL Model Checking

For LTL model checking purposes, the system to be analysed has to be described in some modeling language, `ProMeLa` [5] 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` [5], `DiVinE` [10], or LTSmin [11], 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 behaviours of the system. The property to be checked (LTL formula φ) is negated and translated into a Büchi automaton $A_{\neg\varphi}$ describing all the behaviours 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 digraph of the product automaton. An accepting cycle is a

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: $\begin{cases} \texttt{true} & A_{S\times\neg\varphi} \text{ contains accepting cycle} \\ \texttt{false} & \text{otherwise} \end{cases}$

```
1 while ∃v ∈ V : A(v) do
2     map ← computeAllMaps(G, <)
3     foreach u ∈ V do
4         if u = map(u) then
5             return true
6         else
7             A(map(u)) ← false
8 return false
```

cycle that contains an accepting state. The LTL model checking problem is thus reduced to the problem of deciding existence of a reachable accepting cycle in the product automaton graph (*accepting cycle detection* problem).

2.1. Maximal Accepting Predecessor Algorithm

There are several parallel algorithms for accepting cycle detection. One of them is the algorithm MAP [25]. Let $G = (V, E, v_0, \mathcal{A})$ be the graph of the product automaton where V is the finite set of vertices, E is the set of edges (E^+ its transitive closure), v_0 is the initial vertex, and $\mathcal{A}$ is the 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 $V \cup \{\perp\}$ ($\perp \notin V$) by requesting that for all $v \in V$ we have $\perp < v$. Furthermore, let $map : V \to V \cup \{\perp\}$ be a function returning the maximal accepting successor of a given vertex or $\perp$ if it does not exist, i.e. $map(u) = \max\{\perp, v \mid (u, v) \in E^+ \wedge \mathcal{A}(v)\}$.

The idea of MAP algorithm for detection of an accepting cycle is as follows. If an accepting vertex u is its own maximal accepting successor, i.e. $u = map(u)$, the presence of an accepting cycle is guaranteed. If there are accepting cycles in the graph, but for none of the accepting vertices $u = map(u)$, then the maximal accepting successors as computed for vertices on accepting cycles must always lie outside a cycle. An accepting vertex lying outside a cycle can be safely marked as non-accepting as it cannot be reason for the existence of an accepting cycle. The idea of the iterative algorithm is to process the graph so that each iteration computes map values for all 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 the rest of computation. The algorithm iterates until an accepting cycle is found or the set of accepting vertices becomes empty. For details see the pseudo-code in Algorithm 1. The proof of the correctness can be found in [25].

A key procedure of the algorithm is computeAllMaps() (called *a propagation*) that is responsible

Algorithm 2: COMPUTEALLMAPS($G, <$)

```
  Input  : directed graph G = (V, E, v0, A) of A_{S×¬φ},
           linear ordering < on V
  Output: map function
1 foreach u ∈ V do map(u) ←⊥                    // also prevMap ≠ map
2 while map ≠ prevMap do
3 |  prevMap ← map
4 |  foreach (u, v) ∈ E do
5 |  |_ map(u) ← maxacc(u, v)
  |_
6 return map
```

for computing the values of the function *map* for all the vertices reachable from the initial vertex (see the pseudo-code in Algorithm 2). 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 *outer* iterations, and the iterations of the while loop of procedure COMPUTEALLMAPS given in Algorithm 2 as *inner* iterations. The time complexity of Algorithm MAP is $\mathcal{O}(|V|^2 \cdot (|V| + |E|))$ [25] since in the worst case the algorithm performs $|V|$ outer iterations and each outer iteration takes $|V| \cdot (|V| + |E|)$ time (at most $|V|$ propagations has to be done in procedure COMPUTEALLMAPS). The ordering of the set of vertices has a significant impact on the practical performance of the algorithm [25]. To obtain an optimal ordering (ensuring $\mathcal{O}(|V| + |E|)$ time complexity of the algorithm) is at least as hard as to detect the presence of an accepting cycle in the graph, therefore, only simple ordering heuristics are applied [25].

The practical performance of the basic algorithm may be further improved 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 may be prevented from propagating values of *map* along 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 localised 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.

Algorithm 3: Algorithm OWCTY

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

Output: $\begin{cases} \texttt{true} & \text{if } A_{S\times\neg\varphi} \text{ contains accepting cycle} \\ \texttt{false} & \text{otherwise} \end{cases}$

```
1 S ← REACHABILITY(v0)
2 old ← ∅
3 while S ≠ old do
4 |   old ← S
5 |   S ← REACHABILITY({s|s ∈ S ∧ A(s)})
6 |   S ← ELIMINATION(S)
7 return S ≠ ∅
```

Algorithm 4: Function Reachability

Input : directed graph $G = (V, E)$ and set of vertices S

Output: set of vertices $R = \{v \in V \mid \exists s \in S : (s, v) \in E^+\}$

```
1 R ← ∅
2 while S ≠ ∅ do
3 |   pick and remove s from S
4 |   foreach v such that (s, v) ∈ E do
5 |   |   if v ∉ R then
6 |   |   |   S ← S ∪ {v}
7 |   |   |   R ← R ∪ {v}
8 return R
```

2.2. One-Way-Catch-Them-Young Algorithm

Another parallel algorithm for accepting cycle detection is One-Way-Catch-Them-Young (OWCTY) algorithm [26]. The key idea of the algorithm is maintaining an approximating set of vertices that may lie on an accepting cycle in the graph G. The algorithm repeatedly refines the approximating set by locating and removing vertices that cannot lie on any accepting cycle. The algorithm employs two rules to remove vertices from the approximating set: First, a vertex is removed from the approximation set if it cannot be reached from an accepting vertex in the set, second, a vertex is removed from the approximation set if it has zero in-degree in the set.

The basic scheme of the OWCTY algorithm is given in Algorithm 3. The function REACHABILITY(S) (Algorithm 4) computes the set of all vertices that are reachable from the set S. The function ELIMINATION(S) (Algorithm 5) successively eliminates those vertices that have zero in-degree in S. The assignment on line 5 removes from the graph the vertices according to the first rule. The assignment on line 6 removes vertices according to the second one. The **while** loop terminates when a fix-point of the approximation is reached. If the approximating set is nonempty, the presence of an accepting cycle is guaranteed. The proof of the correctness can be found in [26]. Moreover, we can weaken the termination condition in the following way:

Algorithm 5: Function Elimination

Input : directed graph $G = (V, E)$ and set of vertices S
Output: set of vertices $R = \{r \in S \mid \exists\, c_0, c_1, \ldots, c_{n-1} \in S : \forall\, i\, (0 \leq i < n)\ (c_i, c_{(i+1) \bmod n}) \in E$
$\wedge \exists\, j\, (0 \leq j < n)\, (c_j = r \vee (c_j, r) \in E^+)\}$

```
1 R ← S
2 old ← ∅
3 elim ← {e ∈ R | ∄r ∈ R : (r, e) ∈ E}
4 while old ≠ elim do
5 |   old ← elim
6 |   R ← R ∖ elim
7 |_  elim ← {e ∈ R | ∄r ∈ R : (r, e) ∈ E}
8 return R
```

Proposition 1. ELIMINATION$(S) = S$ *is a correct termination condition of Algorithm 3.*

PROOF. Let us assume that $S' :=$ REACHABILITY$(S \cap F) =$ ELIMINATION(S) and let $\rightsquigarrow$ denote reachability relation. Then if $S' \neq \emptyset$ we have: 1) $\forall u \in S'.\exists v \in F : u \rightsquigarrow v$, 2) $\forall v \in S'.\exists u \in S' : (u, v) \in E$. Hence there is an infinite sequence $\pi := u_1, v_1, u_2, v_2, \ldots : u_i \in F, (v_i, u_i) \in E, u_i \rightsquigarrow v_{i-1}$. And since F is finite, we may conclude that π contains an accepting cycle.

The time complexity of Algorithm OWCTY is $\mathcal{O}(|V| \cdot (|V| + |E|)$ [26] since the algorithm eliminates at least one vertex in each iteration of the main while loop on line 3 (otherwise it terminates) and both REACHABILITY and ELIMINATION takes $\mathcal{O}(|V| + |E|)$ time.

In practice, the number of iterations of the while loop needed to compute the fixpoint is very small. This can be supported in theory by the fact that the number of iterations needed is bound by the height of quotient graph of G. The *quotient graph* of $G = (V, E)$ is a graph $G' = (V', E')$ such that V' is the set of strongly connected components of G and $(C_1, C_2) \in E'$ if and only if $C_1 \neq C_2$ and there exist $c_1 \in C_1$ and $c_2 \in C_2$ such that $(c_1, c_2) \in E$. The *height* of the graph G is the length of the longest path in the quotient graph of G (note that the quotient graph is acyclic). Let h be a height of the input graph, then the more precise complexity of Algorithm OWCTY is $\mathcal{O}(h \cdot (|V| + |E|))$ [26]. Moreover, the algorithm takes only $\mathcal{O}(|V| + |E|)$ time for an important subclass of LTL properties so-called *weak* LTL properties (only one iteration of the main while loop is required) [9].

3. Model Checking on CUDA

The Compute Unified Device Architectures (CUDA) [15], developed by NVIDIA, is a parallel programming model and a software environment providing general purpose programming on Graphics Processing Units (GPUs). At the hardware level, a GPU device is a collection of multiprocessors, each consisting of eight scalar processor cores, an instruction unit, on-chip shared memory, and texture and constant memory

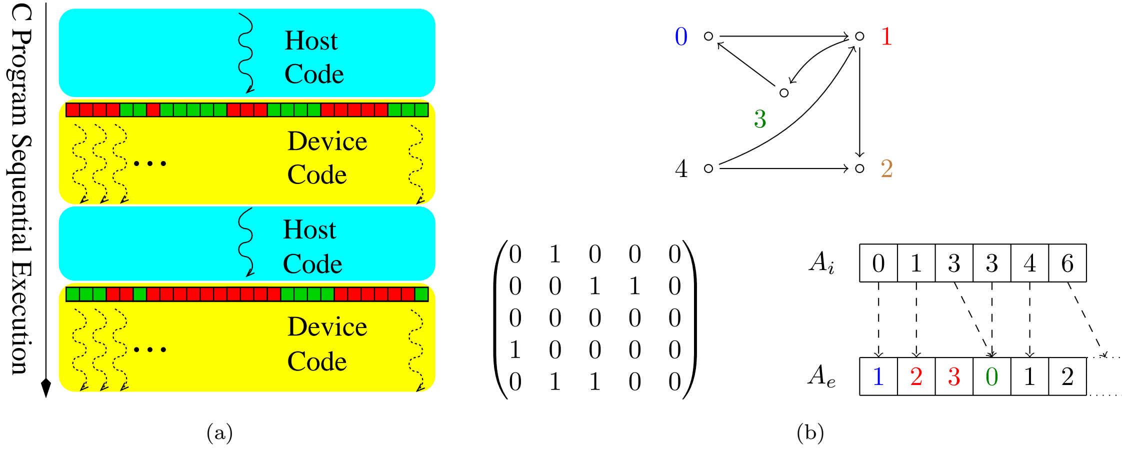


Figure 1: a) Sequential heterogeneous computation workflow with CUDA. b) Matrix and adjacency list representation: a graph $G = (V, E)$ (top) is stored as matrix (left) and two arrays (right): A_i of size $|V| + 1$ and A_e of size $|E|$.

caches. Every core has a large set of local 32-bit registers but no or a very small cache (L1 cache has configurable size of 16-48KB). The multiprocessors follow the SIMD architecture, i.e. they concurrently execute the same program instruction on different data. Communication among multiprocessors is realised 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 consists of a *host* code running on the CPU and a *device* code running on the GPU. The device code is structured into the so called *kernels*. A kernel executes the same scalar sequential program in many *independent data-parallel threads*.

Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment, a group of threads called a *warp* executes on the multiprocessors in a lock-step manner. When several warps are scheduled on multiprocessors, memory latencies and pipeline stalls are hidden primarily by switching to another warp. Overall the combination of out-of-order CPU and data-parallel processing GPU allows for heterogeneous computation as illustrated in Figure 1a, where sequential host code and parallel device code are executed in turns.

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. In our case, it is representation of the graph $A_{S\times\neg\varphi}$ to be encoded appropriately in the first place. Note that uncompressed matrix or dynamically linked adjacency list graph representations violate these requirements, and as such they are inappropriate for CUDA accelerated computation.

Algorithm 6: FORWARDREACHABILITYKERNEL – device code (run in parallel $\forall v \in V$)

```
   Input  : gA_e, gA_i, gFlags, fixPointFound
 1 tid ← blockId.x ∗ blockDim.x + threadId.x                    // tid ≡ v
 2 myVertex ← gFlags[v]
 3 if myVertex.expanded ∨ ¬myVertex.reached then
 4     return
 5 first ← gA_i[v]
 6 last ← gA_i[v + 1]
 7 localFixPoint ← true
 8 foreach index ∈ first, . . . , last do
 9     targetVertex ← gA_e[index]
10     mySucc ← gFlags[targetVertex]
11     if mySucc.reached then
12         continue
13     mySucc.reached ← true
14     gFlags[targetVertex] ← mySucc
15     localFixPoint ← false
16 if ¬localFixPoint then
17     fixPointFound ← false
18 myVertex.expanded ← true
19 gFlags[v] ← myVertex
```

3.1. CUDA Accelerated Graph Algorithms

To realise efficiently a CUDA-aware graph algorithm the graph needs to be represented in a compact, preferably vector-like, fashion [27]. *Compressed sparse row* representation of the matrix of the graph has proven to be an option for efficient CUDA graph encoding [28]. This graph encoding uses two one-dimensional arrays A_i and A_e, to encode a directed graph as depicted in Figure 1b. For a vertex v_j array A_i keeps the sum of the number of edges emanating from vertices v_0 to v_j. The number of edges emanating from a vertex v_j can be computed as $A_i[j] - A_i[j-1]$. The idea of this encoding is such that the value of $A_i[j]$ serves as an index to the second array A_e. Array A_e is a concatenation of ordered lists of target vertices of edges emanating from individual graph vertices. Sizes of arrays A_i and A_e correspond to the sizes of sets of vertices and edges of the graph, respectively. Should other data be needed by a CUDA kernel computation, it is encoded in similar way.

Having properly encoded the graph the standard CUDA accelerated graph traversal procedure (reachability procedure) is given as Algorithm 7. The general work-flow of the algorithm is that the CPU runs the main loop of the algorithm and performs calls to CUDA kernels that run on the GPU (CUDA kernel for the reachability procedure is listed as Algorithm 6). This general work-flow is shared among all other graph procedures presented in this paper.

To perform graph reachability procedure two additional data structures are required to keep the track of vertices being reached and vertices being reached but not yet expanded. To that end the CUDA kernel

Algorithm 7: CUDA Forward Reachability – host code

Input : directed graph $G = (V, E, v_0,)$
Output: set of vertices $R = \{v \in V \mid \exists s \in S : (s, v) \in E^+\}$

1 $\textsc{createRepresentation}(G, A_e, A_i, Flags)$
2 fixPointFound ← `false`
3 $\textsc{copyToGPU}((A_e, A_i, Flags) \rightarrow (gA_e, gA_i, gFlags))$
4 **while** ¬fixPointFound **do**
5 | fixPointFound ← `true`
6 | $\textsc{forwardReachabilityKernel}(gA_e, gA_i, gFlags,$ fixPointFound$)$
7 $\textsc{copyToCPU}(gFlags \rightarrow Flags)$
8 $R \leftarrow \{v \in V \mid Flags[v].reached = \texttt{true}\}$
9 **return** R

employs two vertex labels: *reached* and *expanded*, respectively. Each single call to the CUDA kernel then explores edges emanating from *reached* but not *expanded* vertices and updates vertex labels accordingly. The CUDA kernel is invoked repeatedly until vertex labels change.

In the case of graph algorithms CUDA processing allows for very fine granularity of parallelism [28]. In particular, a separate thread is executed for every vertex of the graph (each item of Array A_i). This approach may naturally lead to uneven work load distribution and performance degradation if the out-degree (number of edges emanating from a vertex) vary significantly among vertices of the graph. Several solutions to the problem of irregular graphs have been proposed, discussed and evaluated [29, 24, 30].

According to our experience the out-degree of vertices in model checking graphs tend to have minor variations only, so the work load imbalance is not an issue in this application domain. However, even if the out-degree do not vary much there is still strong correlation among the average out-degree of a vertex and the overall performance of a general data-parallel CUDA accelerated graph algorithm [24]. This is because the reachability procedure performs in linear time with respect to the radius of the graph. The radius tends to grow for a fixed-size graph as the average out-degree of a vertex decreases. Note that the radius of the graph determines how many times the CUDA kernel must be called in order to achieve the fixpoint in the computation, which is not only relevant to reachability procedure, but applies to all other graph procedures presented later on in this paper.

3.2. Limitations of CUDA Acceleration

The general work-flow as presented in the previous Subsection requires to copy the complete representation of the graph to the GPU device memory. This step of the algorithm limits the applicability of the approach to those problems whose compact graph representation fits the memory of GPU. Provided the graph representation fits, the transfer costs have only a negligible impact on the overall performance as the graph representation and other data structures are copied only once at the beginning (and possibly second time at the end) of computation. Calls to individual CUDA kernels require transfer of small amount

of information only, such as a bit indication whether a fixpoint has been reached. In the case that the graph representation is too large to fit the GPU memory, multiple GPU devices might be used. In that case the transfer costs are more significant as the computation on two or more GPU devices demands data synchronisation among those devices. For more details on multi-GPU acceleration see Section 4.2 of this paper.

To evaluate how efficiently can CUDA handle suggested way of data-parallel processing of graph algorithms we designed and implemented a simple CUDA application mimicing the typical behaviour of a data-parallel graph algorithm. The goal was to explore how different memory access patterns and work-load patterns affect the acceleration achieved by employing many-core GPUs and to validate that the suggested approach is justified. The idea is based on observation that our algorithms employ only a few primitive data-parallel graph operations such as forward or backward reachability. Since the graphs are stored as two one-dimensional arrays we mimic the graph procedures by performing multiple iterations of scanning or updating numbers in two vectors. This corresponds to the operation where each vertex scan its immediate successors (predecessors) and updates its own value (*the read-many pattern*), or the operation where the value associated with is spread among the successors (predecessors) (*the write-many pattern*). The computation of values to be stored is trivial allowing thus the memory access pattern to have the most significant impact on the overall performance.

A CUDA kernel of a graph algorithm typically access the graph representation in a two-level nested way. The first level access goes to a vector position given by a thread id for which the GPU is highly optimised (coalescence). This type of memory access pattern is also suitable for sequential CPU computation as it has good space locality and can be efficiently handled with CPU cache system. The second level access depends on the indexes of the individual successors (predecessors) that are scanned or updated and, therefore, it depends on the structure of the graph. Note that we first create the compact array representation of the graph, so we could partially alter its structure. However, very sophisticated changes would imply a non-trivial time overhead during the computation of the representation that would kill any benefit achieved. We are thus limited to only simple techniques that we described in Section 4.1.

An important aspect of efficient CUDA computing is the impact of different memory access patterns on the performance of the computation. We have therefore measured the performance of the application if it is executed on CPU, GPU without the use of GPU shared memory, and on GPU with the use of shared memory. Note that in most cases the data-parallel graph operations do not allow to efficiently utilise the shared memory as without quite expensive preprocessing the random structure of the graph breaks the data locality with respect to the adjacent threads within the block running on a CUDA multiprocessor. Even though the irregularity of out-degrees is not an issue in model checking graphs, we have measured to the sake of completeness the impact of uneven work-load to threads by setting randomly the number of loads/stores a thread execute.

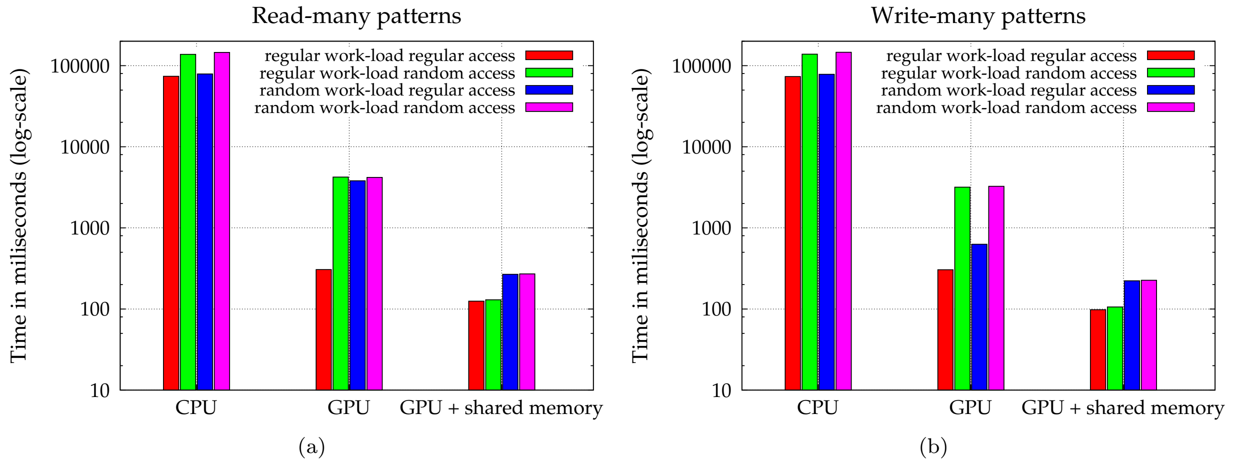


Figure 2: Evaluation of different memory access patterns and work-load patterns for the model of general graph algorithms (1000 iterations, average outdegree 6, size of the graph 1 000 000) : a) Read-many patterns b) Write-many patterns.

The results (runtimes) of our measurements are plotted in Figure 2. Each column reports on time needed to complete one thousand of operations in a given memory pattern simulating a graph with a million of vertices and average out-degree six. As for the CPU computation, we can observe that the random memory access causes twofold slowdown while different work-load patterns do not have much effect. On the other hand, the varying memory access patterns and work-load patterns have a significant impact on performance of the GPU computation. The random memory access leads to fourteen-fold slowdown in the case of read-many pattern and to ten-fold slowdown in the case of write-many pattern compared to regular memory access. The random work-load pattern is twelve times slower in read-many pattern, but surprisingly only two times slower in write-many pattern. The combination of random memory access and random work-load pattern brings no additional slowdown for read-many pattern and additional five-fold slowdown in write-many pattern. Finally, we can see that the use of GPU shared memory reduces the slowdown caused by random memory access and random work-load pattern.

To sum it up, in the case of regular memory access pattern, the regular work-load pattern and efficient utilisation of shared memory, the GPU computation is about two orders of magnitude faster than the sequential CPU computation on suggested graph representation. Even in the case of irregular memory pattern and uneven work-load the GPU computation is about thirty times faster, which clearly justify our motivation to use massively parallel GPU architecture to accelerate the parallel model checking algorithms.

3.3. CUDA Accelerated MAP Algorithm

As discussed in the previous sections, to achieve good acceleration with a CUDA device, the input data must be represented in an appropriate, preferably vector-like, fashion. This is easily achievable in algorithm

Algorithm 8: CUDA MAP Algorithm – host code

Input : directed graph $G = (V, E, v_0, \mathcal{A})$ of $A_{S\times\neg\varphi}$
Output: $\begin{cases} \texttt{true} & A_{S\times\neg\varphi} \text{ contains accepting cycle} \\ \texttt{false} & \text{otherwise} \end{cases}$

```
1  createRepresentation(G, A_e, A_i, Maps)
2  accCycleFound, fixPointFound, unmarked ← false, false, false
3  copyToGPU((A_e, A_i, Maps) → (gA_e, gA_i, gMaps))
4  while unmarked ∧ ¬accCycleFound do
5      while ¬fixPointFound ∧ ¬accCycleFound do
6          fixPointFound ← true
7          mapKernel(gA_e, gA_i, gMaps, accCycleFound, fixPointFound)
8      unmarked ← false
9      unmarkAccKernel(gMaps, unmarked)
10 return accCycleFound
```

Algorithm 9: mapKernel – device code (run in parallel $\forall v \in V$)

Input : $gA_e, gA_i, gMaps$, accCycleFound, fixPointFound

```
1  myVertex, candidate ← gMaps[v], ⊥
2  foreach u ∈ succ(v) do          // succ(v) = {gA_e[gA_i[v]], ..., gA_e[gA_i[v + 1]]}
3      mySucc ← gMaps[u]
4      if myVertex.oldMap = mySucc.oldMap then
5          candidate ← max{candidate, maxacc(v, u)}
6  if candidate = v then
7      accCycleFound ← true
8  if candidate > myVertex.map then
9      myVertex.map, fixPointFound ← candidate, false
10 gMaps[v] ← myVertex
```

MAP as the additional data required by the algorithm are associated with vertices, hence, they can be stored in vectors. In particular, MAP algorithm maintain array $Maps$ that keeps the values of *map*, $oldMap$ and $\mathcal{A}$ predicate for each vertex.

The main computation demanding procedure of algorithm MAP is procedure computeAllMaps, see Section 2. This procedure can be parallelised in the way similar to the previous case of forward reachability procedure. Algorithm 8 lists the algorithm MAP host code. 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, mapKernel, is listed as Algorithm 9. Every call to mapKernel updates the *map* values along every edge (see lines 5 and 8-9 of Algorithm 9). If no accepting cycle is found a fixpoint is detected in mapKernel using the variable fixPointFound. If there is no *map* value to be further propagated, the outer iteration is completed by a call to unmarkAccKernel to unset accepting predicate for accepting states proven to be outside

Algorithm 10: UNMARKACCKERNEL – device code

```
   Input   : gMaps, unmarked
 1 myVertex, change ← gMaps[v], false
 2 if A(v) ∧ myVertex < v then
 3   A(v), unmarked, change ← false, true, true
 4 if myVertex.map ≠ myVertex.oldMap then
 5   myVertex.oldMap, myVertex.map, change ← myVertex.map, ⊥, true
 6 if change then
 7   gMaps[v] ← myVertex
```

Algorithm 11: ELIMINATION Procedure

```
   Input   : gA_e, gA_i, gFlags, accCycleFound, fixPointNotFound
 1 changeFound ← true
 2 while changeFound do
 3   progressKernel(gA_e, gA_i, gFlags)
 4   changeFound, accCycleFound ← false, false
 5   checkKernel(gFlags, changeFound, accCycleFound)
 6   fixPointNotFound ← changeFound ? true : fixPointNotFound

   kernel progressKernel(gA_e, gA_i, gFlags)
 7 if ¬gFlags[v].elim then
 8   foreach u ∈ succ(v) do
 9     if ¬gFlags[u].elim ∧ gFlags[u].elimPrep then
10       gFlags[u].elimPrep ← false

   kernel checkKernel(gFlags, changeFound, accCycleFound)
11 if ¬gFlags[v].elim then
12   if gFlags[v].elimPrep then
13     gFlags[v].elim, changeFound ← true, true
14   else
15     gFlags[v].elimPrep, accCycleFound ← true, true
```

an accepting cycle. Pseudo-code of UNMARKACCKERNEL is listed as Algorithm 10. Note that *map* and *oldMap* values are handled so that *oldMap* values partition the graph with respect to the previous outer iteration.

Performance of the MAP algorithm deeply depends on the vertex ordering. This property is inherited by the CUDA accelerated version of the algorithm [20]. With inappropriate vertex ordering the execution of CUDA MAP algorithm may be significantly slower. Unfortunately, the ordering of vertices is partially determined by the way in which the graph representation is computed. Furthermore, we should point out that we are actually computing minimal accepting successors. Considering successors allows us to store only the forward edges in the graph representation.

Algorithm 12: CUDA OWCTY Algorithm – host code

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

Output: $\begin{cases} \texttt{true} & A_{S\times\neg\varphi} \text{ contains accepting cycle} \\ \texttt{false} & \text{otherwise} \end{cases}$

```
1  createRepresentation(G, A_e, A_i, Flags))
2  copyToGPU((A_e, A_i, Flags) → (gA_e, gA_i, gFlags))
3  visAcceptingKernel(gA_e, gA_i, gFlags)
4  fixPointNotFound, accCycleFound ← true, false
5  while fixPointNotFound do
6  │  reachability(gA_e, gA_i, gFlags)
7  │  testSetKernel(gFlags)
8  │  fixPointNotFound ← false
9  └  elimination(gA_e, gA_i, gFlags, accCycleFound, fixPointNotFound)
10 return accCycleFound
```

3.4. CUDA Accelerated OWCTY Algorithm

The non-CUDA version of OWCTY algorithm comprises of alternating execution of forward reachability and *backward elimination* (Algorithm 3). In the current context we denote elimination of vertices without immediate predecessors as backward elimination. These two operations will similarly be the building blocks of our CUDA accelerated implementation of algorithm OWCTY.

Implementation of reachability was given sufficient space in Section 3 and advance techniques were proposed in [29, 24, 30]. Therefore, in the following we will focus on describing in more detail the implementation of backward elimination and subsequently the whole OWCTY algorithm. Given the fact that the algorithm disposes of only the forward edges we were unable to follow the most obvious implementation procedure, i.e. to eliminate a vertex if all its predecessors were already eliminated. The option of providing also the backward edges would be overly complex both in time and space.

Our backward elimination hence needed to consist of two steps (see Algorithm 11). The first step is performed by the CUDA kernel progressKernel, starting at line 7. This kernel has the purpose of propagating the property of not to be eliminated to its successors. Followed by the second kernel checkKernel which eliminates vertices without this property. Finally, the flags *elim* and *elimPrep* are stored as bits in a piece of memory assigned to every vertex, which allows their change to be performed very fast even on simple GPU processing units.

Having described the building blocks, we may proceed to the actual OWCTY algorithm implementation (see Algorithm 12). The basic layout is similar to the original implementation. The CUDA kernel visAcceptingKernel sets all accepting vertices to visited. Having considered the Proposition 1, we do not need to test if reachability visited all vertices. Only its effect, the elimination of non-visited vertices is necessary (via kernel testSetKernel). The elimination proceeds as described above. Furthermore, if no vertex is eliminated (line 6 of Algorithm 11) the algorithm terminates with resulting value stored in variable

accCycleFound. It is observable that accCycleFound keeps track of existence of the not eliminated vertices thus providing correct answer once the main cycle terminated.

The dual version of OWCTY algorithm, here referred to as *reversed* OWCTY, may seem to present equivalent obstacles as far as the CUDA implementation is concerned. However, as stated in [22], backward reachability via forward edges is tractable (with certain slowdown), which allows us to implement elimination in the trivial way sketched above. The rest of the algorithm remains unchanged.

In [23] we show that the reversed variant of CUDA accelerated OWCTY algorithm has better times that the standard variant. The reason behind it is that in reversed OWCTY the elimination was implemented more efficiently to the detriment of the reachability procedure. And since in most of the tested models the reachability needed considerably less iteration, it was the reversed version that thrived. We will compare the performance of CUDA MAP and CUDA reversed OWCTY algorithms (further referred only as CUDA OWCTY) in the Section 5.

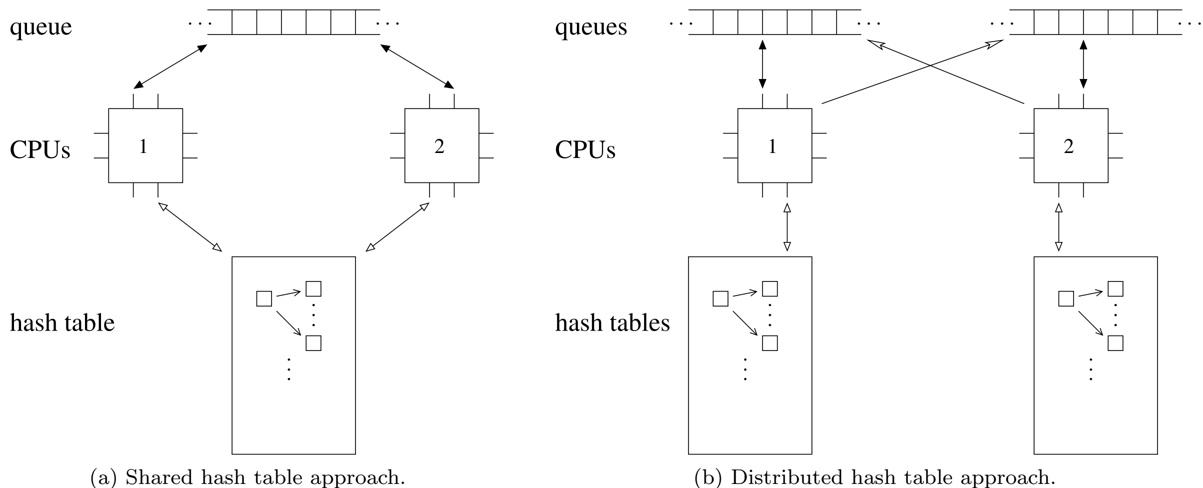
4. Problems Related to Model Checking on CUDA

As far as accepting cycle detection on CUDA is concerned, the previous section could serve as a complete description. However, in model checking as a whole, accepting cycle detection is only one part of the solution. To fully solve the model checking problem using GPUs there are other issues that need to be tackles, e.g. transformation from implicit graph representation to a form suitable for GPU computation. The solution of problems related to and further optimisation of CUDA accelerated model checking will form the content of this chapter.

4.1. Computation of Adjacency List Representation

The crucial procedure of the whole verification process is the transformation of the input data as given to the model checker, into a form suitable for CUDA accelerated computation (line 1 of Algorithm 8 and line 1 of Algorithm 12). In the model checking process the graph is given implicitly by 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 the CUDA accelerated algorithm, we have to turn the implicit definition of the graph into an explicit one. This process is generally referred to as the *state space generation*. In addition to the explicit state space construction we also have to build the corresponding adjacency list representation of the graph.

The main bottleneck of the whole CUDA accelerated approach to LTL model checking is the costly procedure of the construction of the adjacency list representation [20]. In order to alleviate the burden of the transformation of the implicit definition of the graph into the adjacency list representation, we have devised a multi-core parallel procedure for it. The procedure builds upon the multi-core parallel state space exploration that is reported to achieve up to ten-fold speed-up on a 16-core machine [7, 8].



(a) Shared hash table approach.

(b) Distributed hash table approach.

Figure 3: Comparison of the workflow of the two state-of-the-art approches to state space generation. a) uses a single shared hash table (storing closed vertices) and a shared queue (vertices to be expanded). Without any distribution this approach is limited to shared memory parallelism. In b) every thread has its own hash table and queue, but vertices from other graph partitions must be sent to their owners.

In the parallel state space generation procedure the vertices of the graph are assigned to parallel workers using a hash-based partition function. Each parallel thread has a local storage to keep track of the generated vertices assigned to the thread. Vertices that are new (have not been stored yet) are stored and their immediate successors are generated. Non-local vertices are handed out to the owning threads according to the partition function. When none of the threads has new vertices to be processed the state space generation terminates. The crucial aspect regarding the scalability is the way the threads exchange vertices to be explored. In this approach we rely on contention and lock-free queue structures (FIFOs).

Note that recently another method allowing for more efficient multi-core acceleration of state space generation has been proposed [11]. In contrast to the aforementioned method, they are using shared storage based on a lock-free hash table and therefore this approach can benefit from low communication costs. No vertex has a predefined owner and all new vertices are sent to a shared queue. Communication with the queue has to be protected by locking and to minimise the number of these locks it was further proposed that the elements of the queue are not single vertices but rather whole *chunks* of them. The two methods can be distinguished on the illustration of their work-flow in Figure 3.

The key extension required from the state space generation procedure is the computation of a unique integer number for each vertex. In particular, we require a mapping of vertices into integer numbers between 0 and $|V| - 1$. This is slightly tricky if we want to avoid multiple state space traversal procedures. In particular, when a transition is generated and stored to the adjacency list representation the number of the target vertex is unknown to the generating thread. Hence we need to alter the hash table to also contain the number associated with each state. Note that in the case that a counterexample generation (described

in Section 4.3) was required, we also store in the hash table a pointer to the predecessor state to create *the parent graph*.

Yet even if the hash table contains with every state a unique identifier, there is the possibility that the target vertex is not stored in the hash table. To solve this problem we have designed a write-only vector data structure that allows insertion of data in two different ways. A vector element can be inserted either directly by calling *void* `push_back`(T *elem*), or in a two-phase manner, where we first allocate space for the element by calling T ∗ `push_empty`() and then we store it using the returned pointer.

The parallel adjacency list construction procedure works as follows. The number that a thread assigns to a vertex is composed of two parts, these are the thread-unique *thread_id* (4 bits) and per thread-unique *vertex_number* (28 bits). When a local vertex is generated by a thread it is given the lowest per-thread fresh number – stored directly in the vector using `push_back` function. The specific solution differs for shared and distributed hash table approach. In distributed approach, when a non-local vertex is generated, a space for it is preallocated using `push_empty` function and the address of the preallocated space is sent together with the vertex to the owning thread that assigns a number to the vertex and stores it to the preallocated space remotely.

In shared hash table approach, a tuple (vertex, address) is enqueued on the shared queue and the thread that removes the tuple is responsible for storing the vertex in the hash table with unique number and for writing this number to the dequeued address. The additional adjacency list construction contributes nontrivially to the amount of work done during state space generation. Apart from other aspects, now every threads has to enqueue all vertices that are not yet in the hash table (before the thread could store these vertices itself). Even though the construction almost doubled the generation time our experiments show that the speedup gained from using compact representation enables to overcome the effect of slower generation.

As soon as the whole state space is generated, the local vectors are concatenated into a single system-wide vector and processed with two CUDA kernels to translate the pairs of numbers into continuous range of integers. Note that due to the parallel processing, the final ordering of vertices in adjacency list representation is not computed deterministically.

Finally, to accelerate CUDA computation, we employed a decomposition technique to shrink the product automaton graph [31, 32]. The idea is to decompose the property automaton into strongly connected components and then project this decomposition to the final graph. Since some parts of the product automata graph are known to be without accepting vertices in advance and may be omitted when constructing the adjacency list representation of the graph. This technique significantly reduced the size of the representation as well as the number of repropagations needed (see the Section 5).

Efficient utilisation of many-core GPUs to accelerate state space generation is another logical direction of research in the community of parallel and distributed model checking. Edelkamp et al. have proposed the

acceleration of state space generation by executing complex operations on GPUs [18]. They have achieved significant speedup of transition enabledness checking and successors generation. However, duplicate detection, the most crucial part of the state space generation, has not been parallelised on GPU yet. Thus the overall speedup of the whole state space generation was moderate.

4.2. Overcoming Memory Limitation – Multi GPU Model Checking

The size of the compact representation can be estimated to approximately $8|V|+4|E|$ bytes for OWCTY and $12|V|+4|E|$ bytes for MAP, which is considerably less that the amount of memory consumed by a model checking procedure that stores the whole states. Despite the fact that both our algorithms use a compact representation, the model checking graphs tend to be exceedingly large, an thus the scope of the proposed algorithms is diminished due to GPU memory limitation. In this section we describe two methods to overcome the memory limitation of a single CUDA device in the context of CUDA accelerated LTL model checking. Both approaches build upon the idea of splitting the data structures into parts and distributing them among multiple CUDA devices. Such a strategy allows to process verification instances that do not fit the memory of a single GPU device, but fit the aggregate memory of multiple CUDA devices. There are two primary data structures to be partitioned. First, the adjacency list representation of the graph, and second, the vector of values associated with individual vertices ($Maps$ for MAP algorithm and $Flags$ for the OWCTY).

Ideal partitioning would be to split both data structures into a number of even-sized pieces in such a way that processing these pieces in parallel would require no interaction among parallel threads. This is possible only if no accepting cycle crosses the boundaries of a single graph partition. Unfortunately, such a partitioning generally does not exist, and even if it does, it is computationally expensive to be obtained. As a result we do not aim at computing a partitioning that would preserve cycle locality, but rather at the partitioning that allows for uniform data structure distribution while being aware of the necessity of interaction during the parallel computation.

The two suggested partitionings are as follows. In the first approach, we partition only the adjacency list representation of the graph, i.e. every CUDA device keeps one part of the adjacency list representation and the complete vector of values associated to individual vertices. We do such a partitioning of the adjacency list representation in order for all edges emanating from a single vertex to be positioned to the same partition. The second approach extends the first one. In addition to the adjacency list representation partitioning it also introduces distribution of the vector of values. In particular, every CUDA device keeps one part of the adjacency list representation, and a reduced vector of values. The reduced vector keeps the values for all vertices that appear in the local adjacency list representation part. Note that some of those vertices are the so called *foreign vertices*, i.e. vertices whose edges are kept in another (foreign) part of the adjacency list representation, but are listed as end-points of some edges in the local part of the adjacency

Algorithm 13: Parallel MAP Algorithm (inner fix-point) – host code

```
   Input  : localG, localMaps, accCycleFound, fixPointFound
 1 globalChange ← true
 2 while globalChange ∧ ¬accCycleFound do
 3 |   foreignMaps, localChange ← DOWNLOAD(), false
 4 |   while ¬fixPointFound ∧ ¬accCycleFound do
 5 |   |   fixPointFound ← true
 6 |   |   MAPKERNEL(localG, localMaps, foreignMaps, accCycleFound, fixPointFound)
 7 |   |_  localChange ← localChange ∨ ¬fixPointFound
 8 |   UPLOAD(localMaps)
 9 |   VOTEIN(localChange)
10 |   RENDEZVOUS()
11 |_  globalChange ← VOTEOUT()
```

list representation.

4.2.1. Synchronisation

Here we first explain how to utilise multiple CUDA devices to accelerate the MAP algorithm and then we discuss how to generalise this approach so that it would be possible to apply it to other graph algorithms such as OWCTY.

Having the edges of the graph distributed among multiple CUDA devices the local computation of the algorithm comes across the necessity to exchange the *map* values of foreign vertices, the so called *synchronisation*. In the distributed MAP algorithm (Algorithm 13) every single CUDA device computes the local fix-point as in the single CUDA computation, but then it synchronises on the values of foreign vertices with all the other CUDA devices. The local fix-point is thus achieved using solely the mutable *map* values of local vertices and the constant *map* values of foreign vertices received from the synchronisation. These two subsequent steps repeat until a global fix-point is found. Since the *map* values of foreign vertices are constant throughout the local fix-point search, the UPLOAD is conditioned by a change detected after considering these values for the first time within the inner cycle. If the global fix-point is found in zero iterations, the individual parallel CUDA workers vote for global termination. Then if after a barrier operation the vote for termination is unanimous (lines 9-11), the algorithm terminates.

In the case of a more complicated graph algorithms such as OWCTY with alternating execution of forward reachability and backward elimination, we have to perform the synchronisation and the vote for termination once the local fix-points of each phase is found. Apart from small adjustments of the terminating condition this was the only change necessary for the OWCTY algorithm to be extended to work on multiple CUDA devices. We have duplicated the multi-device experiments from [22] (for MAP algorithm) using our new OWCTY algorithm and we direct the reader to Section 5 to see the comparison of the two implementations.

4.2.2. Preparing Foreign Vertices Vectors

The synchronisation procedure requires to exchange only the values associated with foreign vertices, however, the communication between CUDA devices is realised through the host memory, where the complete vector of values is maintained. This does not make any problem as regards the first partitioning approach. However, if a CUDA device wanted to read only the values associated with the foreign vertices from the host memory as in the case of the second approach, it would have to perform a scattered read, which is not very efficient.

Our solution to this problem is that after the partitioning of the graph representation, we compute a list of foreign vertices for every participating CUDA device. We duplicate values stored for the foreign vertices in a separate compacted vector, i.e. a vector containing only the values for foreign vertices for a particular CUDA device and use this vectors for efficient communication between host and device memory. Note that individual adjacency list representations need to be modified so that all occurrences of foreign vertices are replaced and accompanied with a special bit indicating that the number is not the local number but an index to the vector of values of foreign vertices.

The compaction procedure is done as follows. First, we employ a CUDA kernel to mark all foreign vertices in the vector of edges in the partitioned adjacency list representation. This can be done due to the static and uniform nature of the partitioning. Then we create a compacted vector containing all foreign vertices exactly once. This is slightly tricky due to the space limitation since we cannot afford to create the copy of the whole vector of vertices. We proposed the solution of this problems in [22].

To finish the preparation of the data for multiple CUDA computation we map the foreign vertices with their counterparts in the compacted vector. This is carried out by a single kernel implementing binary search.

4.3. Early Termination

A key property of some model checking algorithms is that they can be altered to provide early termination. It means that they can detect the presence of an accepting cycle before the state space generation procedure completes its task. We were able to adapt our implementation of both CUDA accelerated algorithms to mimic this behaviour as well. The idea is very similar as in our previous papers [22, 20]. In particular, we let the CPU perform (parallel) state space generation while having the GPU apply CUDA accelerated algorithms on partially constructed graph. If the part of the graph constructed so far contains an accepting cycle, CUDA accelerated algorithms simply reveals it before the state space generation is complete.

To further extend the potential efficiency of the proposed model checking method we allow for both the MAP and OWCTY algorithm to be executed concurrently in the background of the state space generation. This work flow, though requiring two CUDA devices, provides the best result of the two algorithms whether or not was the early termination available (and with negligible impact on their stand-alone performance).

4.4. Counterexample Generation

An important part of the model checking procedure is counterexample generation. If the given model does not satisfy the inspected property and thus an accepting cycle is found, the tool has to provide a counterexample, i.e. an example of behaviour violating the property. In the case of LTL model checking, the counterexample consists of states forming the accepting cycle and states on a path from the initial state to that cycle. In order to efficiently generate the counterexample we have to consider the state space representation and also the algorithm for the accepting cycle detection. In contrast to traditional approaches, our GPU accelerated algorithms are using compact representation of the part of state space where the path from the initial state to the cycle is not necessarily stored (see Section 4.1). Moreover, the states in the compact representation are stored only as unique numbers that do not contain necessary information for the user, and thus a translation back to the full state description is required. Therefore the counterexample generation is more involved in our case and includes two phases.

In the first phase the algorithms identify the states in the compact representation that form the found cycle. Algorithm MAP provides an accepting state on the cycle and marks the part of state space where the cycle is located. Therefore the cycle can be identify using a forward reachability that is launched from the accepting state and restricted to the marked part, followed by the backward reachability launched from the accepting state. The backward reachability follows the parent graph in which each state keeps only one predecessor and that was created during the foregoing forward reachability and stored in $oldMaps$. We can clearly see that the first phase of the counterexample generation has linear time complexity.

Algorithm OWCTY only marks the part of the state space where the cycle is located and therefore we have to first find an accepting state on the cycle. In order to do this we repetitively pick an accepting state from this part of the state space and run a restricted forward reachability to detect whether the state lies on the cycle. If it does not we can safely remove the state and all the other states that have been completely searched during the reachability and continue. This ensure that each state is visited in the reachabilities only once and the first phase of the counterexample generation keeps linear time complexity [26]. Since for both algorithms the first phases includes only restricted forward and backward reachability it can be efficiently accelerated by the GPU as we already described.

In the second phase we select a state on the cycle (represented only as the unique number) and obtain its full state description. To efficiently get this description we have to scan the hash table where both the full state description and the corresponding unique number are stored. This scanning can be easily parallelised by multi-core CPU since each thread can independently scan a part of the hash table. Once we get the full description of the state we can run the restricted CPU forward and backward reachability on the explicit representation (containing the full state description of the entire state space) to obtain the full state description of states forming the cycle and states forming the path to the cycle, respectively. The forward reachability is navigated by states on the cycle and backward reachability is navigated by the parent graph

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 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	If the process is active infinitely often then it is in the critical section infinitely often
bakery	Bakery mutual exclusion algorithm	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
lamport	Lamport's fast mutual exclusion algorithm	Infinitely many times someone is in the critical section.
brp	Bounded retransmission protocol	If the producer sends message, it will eventually get some acknowledgment from the sender process.

Table 1: Models used in the experiments with properties they are expected to have.

that is created during the initial state space generation. Since this phase includes only one forward and backward reachability and one linear scanning of the hash table, it has also linear time complexity. Although the scanning is not performed during the standard approach where only the full explicit representation of the state space is employed, it has a negligible impact on the overall practical performance of the tool.

5. Experimental Evaluation

We have implemented the described algorithms and methods as a part of the DiVinE-CUDA [21]. We compared the performance of the CUDA implementation against CPU implementations, employing the same state space generator, and against the state-of-the-art parallel model checkers. We used DiVinE native models as listed in Table 1. Moreover, we provide an example of models which cannot be verified using the original (single CUDA) algorithms because of space limitation. We show that the employment of the methods for multiple CUDA devices (described in Section 4.2) allows verification of these models.

All the experiments were run on a Linux workstation with two quad core Intel Xeon E5335 Processors @ 2GHz, 8 GB DDR2 @ 1066 MHz RAM and two NVIDIA GeForce GTX 480 GPU with 1.5 GB of GPU memory. In the case of models indicated by stars, whose explicit representation did not fit in the main memory, we first had its adjacency list representation created on a workstation with 32 GB RAM and then we finished the experiments on our CUDA-equipped workstation. Our previous results reported in [20, 23, 22] were all measured on a workstation with the preceding generation of GPUs (GTX 280). The main difference from current generation lies in doubling the number of parallel cores. This hardware upgrade resulted in almost twofold speedup in CUDA computation, yet in the run-times of the whole model checking procedure it was hardly noticeable.

Model	vertices		edges		RAM	accepting
	explored	stored	explored	stored	cons.	cycle
elevator 1	5.0 mil	1.7 mil	63.1 mil	20.5 mil	2.4 GB	N
leader	26.3 mil	26.3 mil	84.1 mil	84.1 mil	3.8 GB	N
peterson 1	19.0 mil	9.5 mil	124.9 mil	41.5 mil	4.6 GB	N
anderson	10.7 mil	6.2 mil	46.8 mil	26.3 mil	2.1 GB	N
lamport*	74.4 mil	35.8 mil	422.8 mil	129.7 mil	21.3 GB	N
elevator 2	0.23 mil	0.18 mil	1.84 mil	1.56 mil	741 MB	Y
phils	0.2 mil	0.17 mil	1.72 mil	1.47 mil	774 MB	Y
peterson 2	0.94 mil	0.74 mil	4.35 mil	3.55 mil	786 MB	Y
bakery	0.24 mil	0.2 mil	1.0 mil	0.89 mil	794 MB	Y
brp*	84.5 mil	42.3 mil	263.2 mil	87.4 mil	22.1 GB	Y

Table 2: Spacial complexity of the models and existence of accepting cycles.

Table 2 captures various statistics of the models. The difference between *stored* and *explored* vertices (edges) illustrates how much of the state space consists of subgraphs without accepting states and therefore how much the technique proposed in Section 4.1 reduces the size of the graph representation. The overall CPU memory consumtion (column *RAM cons.*) does not necessarily relate to the respective sizes of the models since the states stored in hash tables may have different sizes for every model. The column *accepting cycle* depicts whether the model contains the accepting cycle (invalid instance) or not (valid instance). Note that if the graph contains an accepting cycle, the reported numbers refer to the state when the accepting cycle was discovered (see Section 4.3 for more details).

5.1. Comparison with State-of-the-art Model Checkers

There are two leading model checkers in the paradigm of shared memory parallelism and these are DiVinE [14] and LTSmin [12] (though technically DiVinE combines distributed and shared memory parallelism). Considering that DiVinE-CUDA also requires shared memory state space generator, it seemed reasonable to compare its performance with DiVinE and LTSmin. For this comparison to be as fair as possible we have selected among all BEEM models those whose checking by DiVinE lasted for more than 10 seconds, so that we would be able to observe also the scalability of the tools. Another criterion was that the computation does not run out of the operating memory.

Using these criteria we found appropriate 10 incorrect and 6 correct models (it is important to distinguish the two classes because of the on-the-fly capability of all the tools). During our experiments we have observed that LTSmin is considerably more effective on incorrect models. Having rightly attributed this phenomenon to the DFS ordering in which LTSmin explores the state space, we have adapted our own state space generator to partially imitate this behaviour. First by using shared stack instead of a shared queue and then by *reversing* the order in which vertices were taken from chunks. Experiments with reversed ordering are marked with letter R. Also note that due to nondeterminism of the computation the run-times of particular

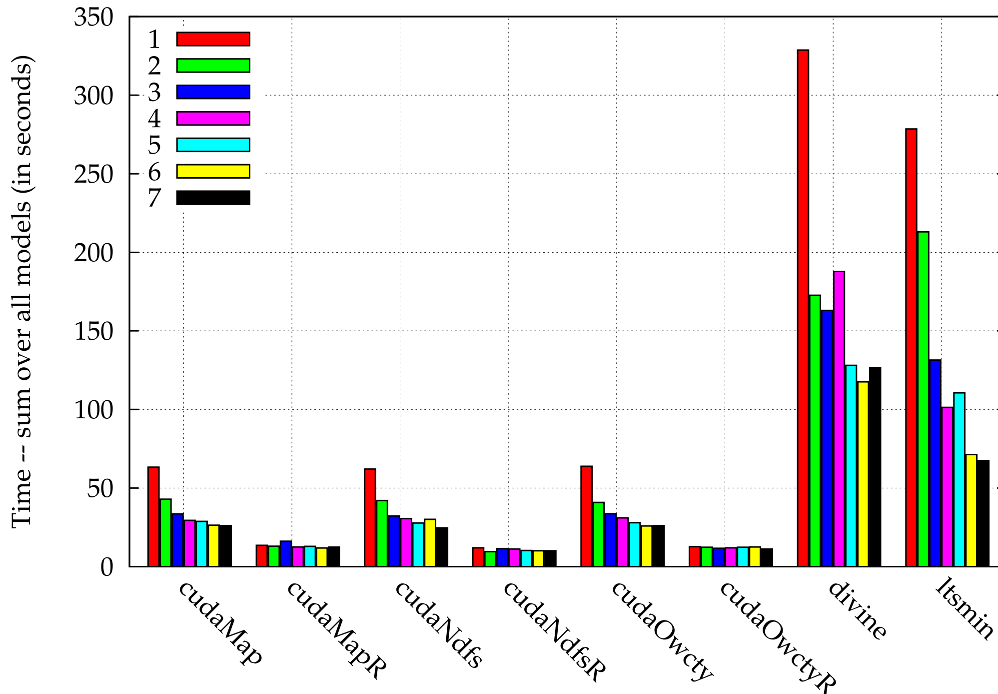


Figure 4: Effectivity of state-of-the-art model checkers on incorrect models (1..7 cores)

runs can vary greatly and thus every experiment was run 5 times and we use the median of the 5 runs in our plots.

In Figures 4 and 5 we depict the overall run-time on correct and incorrect models. To make the view complete we have implemented the sequential nested-dfs algorithm for the same compact representation as used in CUDA computation (denoted as *cudaNdfs*). Though LTSmin usually thrives on incorrect models there were two models in our set on which LTSmin performed very poorly (and had we removed these two models LTSmin would have the best times). Yet overall the algorithms using compact representation clearly dominate the incorrect models. Especially with the reversed order where most of the computation time is spend in the common part in which the representation is prepared.

In much smaller scale this is also true for the experiments on correct models in Figure 5 where it is always necessary to generate the whole state space. Here it is also worth noting that while the reversed generation should have no little positive effect on the run-times it can have a negative effect on the MAP algorithm, because of its dependency on ordering of vertices. The exact run-times of the algorithms on some selected models are summarised in Table 3. Note that the computation of the algorithms which ran out of memory is marked in the table by n/a.

Scalability of model checking algorithm (see Figure 6) is more complicated to measure because it is not always clear if the speedup was caused by non-determinism of state space generation or by the actual

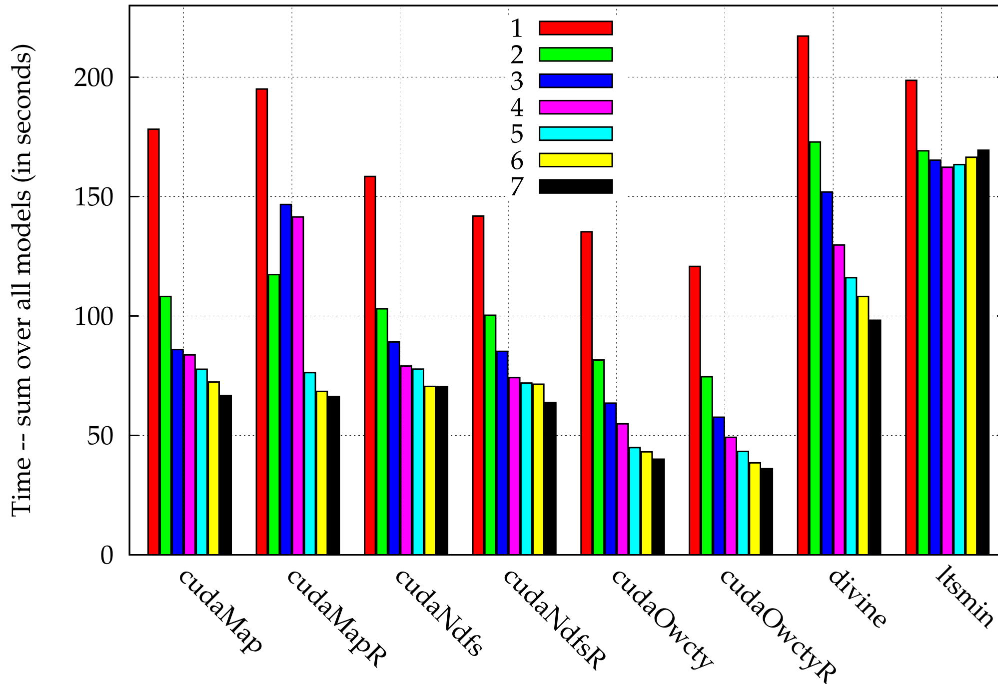


Figure 5: Effectivity on correct models (1..7 cores)

efficiency of the accepting cycle detection algorithm. Hence we have decided to present only the data collected from verification of correct models where this non-determinism is irrelevant. As we can see, in terms of scalability there is hardly any difference between the reversed and normal ordering in state space generation. On the other hand, the difference between shared and distributed hash tables is quite pronounced as represented by the difference between DiVinE and CUDA algorithms.

5.2. Comparison of Algorithms on Compact Representation

Table 3 provides details on run-times of individual CUDA accelerated algorithm parts and gives the comparison to CPU algorithm running on up to 7 core. The table first reports for all 3 algorithms on compact representation the state space generation times (gen.), then it gives the time for creating the compact representation (prep.) and the times spent on CUDA computation (comp.); it finally states the total run-times (tot.) of all algorithms. As for the CUDA algorithms, the total run-time also includes certain initialisation overhead not reported in the table. We can observe that in CUDA accelerated OWCTY algorithm the time for preparation of adjacency list representation significantly dominates to the whole model checking procedure.

We can further see that the CUDA accelerated OWCTY algorithm significantly outperforms the original CUDA accelerated MAP algorithm on most valid model checking instances (without accepting cycle). The

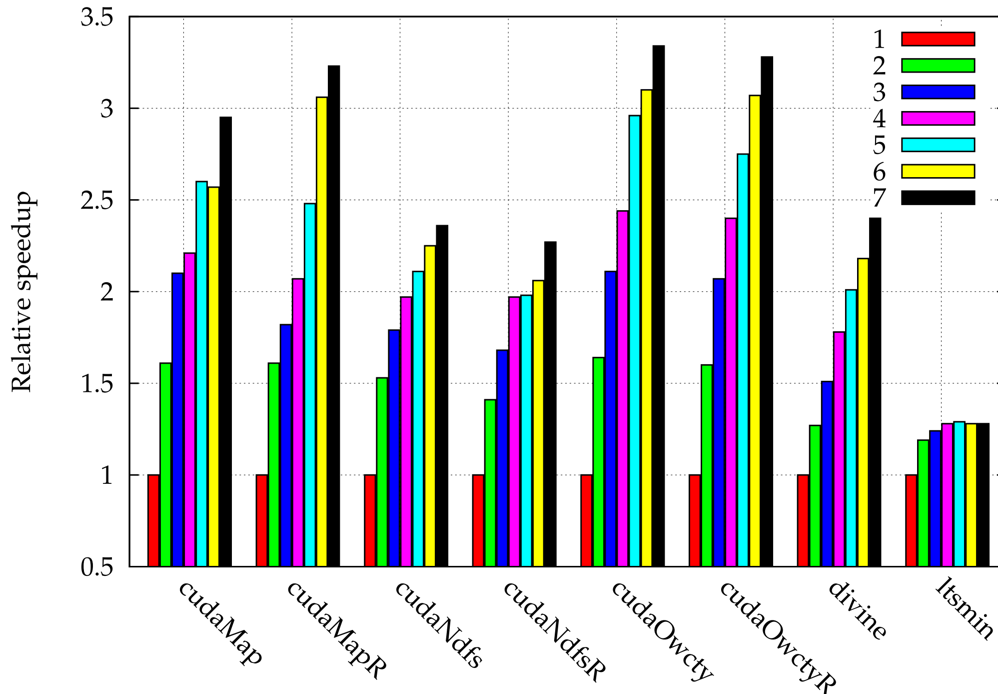


Figure 6: Scalability of state-of-the-art model checkers on correct models (1..7 cores)

results for invalid instances (with accepting cycle) speak also in favour of the OWCTY algorithm, though the gain is considerably less impressive.

From Figure 7 we may observe how effective is the multi-core acceleration of the state space generation – proposed in Section 4.1. We have summed the respective run-times over all tested models and plot them in each bar in the following order: the times of state space generation (red), compact representation preparation (green) and the actual CUDA computation (blue). We can see a steady speedup of the adjacency list construction when more CPU cores are used. However, as we have already explained, the parallel construction usually affects the ordering in adjacency list representation. This leads to different number of calls to CUDA kernels (see [22] for more details) and to different times spent on CUDA computation. Note that during the whole computation of the algorithm, one core oversees the communication with CUDA device and thus cannot be efficiently used in the adjacency list construction.

The experiments also show that the performance of the CUDA OWCTY algorithm does not depend on the ordering in adjacency list representation as much as the CUDA MAP algorithm (see [23] for more details) and therefore the CUDA OWCTY algorithm has better run-times on almost all models also in the case when multi-core acceleration of adjacency list construction is utilised. All together it seems that when multi-core acceleration of adjacency list computation is utilised the OWCTY algorithm is clearly a winner for CUDA computation. The superiority is even more pronounced that it was in our previous papers [22, 23]

Models	#cores	DiVinE-CUDA												LTSmin	DiVinE
		MAP				OWCTY				NDFS					
		gen.	prep.	comp.	tot.	gen.	prep.	comp.	tot.	gen.	prep.	comp.	tot.		
elevator 1	1	18.7	0.5	19.7	39.0	19.5	0.5	0.6	20.6	18.6	0.5	3.5	22.7	45.6	32.3
	2	10.6	0.5	19.9	31.2	11.1	0.5	0.5	12.3	10.8	0.5	3.5	14.9	39.7	29.7
	4	6.0	0.5	20.5	27.1	6.4	0.5	0.5	7.5	6.0	0.5	3.6	10.2	37.8	25.4
	7	3.9	0.5	18.0	22.5	4.1	0.5	0.5	5.3	3.9	0.5	3.6	8.1	39.4	19.2
leader	1	44.7	0.9	3.6	49.3	45.9	0.9	0.2	47.0	43.7	0.9	19.1	63.8	77.2	180.1
	2	26.1	0.9	3.3	30.5	27.0	0.9	0.2	28.2	25.8	0.9	19.5	46.3	39.8	142.5
	4	14.8	0.9	3.0	18.8	15.4	0.9	0.1	16.5	14.9	0.9	19.6	35.5	21.7	n/a
	7	9.4	1.0	2.7	13.2	9.7	1.0	0.1	11.0	9.4	0.9	19.7	30.1	14.8	n/a
peterson 1	1	45.7	1.1	35.7	82.6	47.4	1.1	0.7	49.3	46.2	1.1	11.8	59.2	111.1	109.1
	2	26.5	1.1	26.2	54.0	27.7	1.1	0.7	29.6	27.1	1.1	12.2	40.5	97.4	93.8
	4	15.1	1.1	22.8	39.1	16.0	1.1	0.7	17.9	15.4	1.1	12.4	29.1	95.8	74.2
	7	9.7	1.2	21.2	32.1	10.4	1.2	0.7	13.3	9.8	1.1	12.7	23.7	101.1	55.1
anderson	1	21.9	0.5	2.8	25.3	22.7	0.5	0.2	23.4	21.7	0.4	4.8	27.1	47.1	63.6
	2	12.5	0.5	2.1	1.51	12.9	0.5	0.2	13.7	12.6	0.4	4.9	18.1	34.4	51.0
	4	7.1	0.5	1.8	9.5	7.4	0.5	0.2	8.1	7.1	0.4	5.0	12.7	30.9	41.2
	7	4.6	0.5	1.8	7.0	4.8	0.5	0.2	5.5	4.5	0.4	5.0	10.0	33.0	32.3
elevator 2	1	1.0	0.0	0.2	1.3	0.7	0.0	0.0	0.8	0.6	0.0	0.1	0.7	0.1	21.2
	2	4.0	0.1	1.6	5.9	0.8	0.0	0.0	0.9	1.5	0.0	0.2	1.8	0.2	66.0
	4	1.6	0.1	0.8	2.6	1.8	0.1	0.2	2.2	0.6	0.0	0.1	0.8	0.1	52.7
	7	1.6	0.1	0.7	2.6	1.0	0.1	0.1	1.2	0.8	0.0	0.2	1.1	0.2	39.7
phils	1	0.7	0.0	0.0	0.8	0.7	0.0	0.0	0.8	0.5	0.0	0.0	0.5	217.8	24.8
	2	0.7	0.1	0.0	0.8	0.8	0.1	0.0	0.9	0.5	0.0	0.0	0.6	216.2	77.9
	4	0.8	0.1	0.0	0.9	0.9	0.1	0.1	1.1	0.5	0.0	0.0	0.6	n/a	24.4
	7	0.9	0.2	0.0	1.1	1.0	0.1	0.1	1.3	1.9	0.3	0.2	2.4	n/a	13.8
peterson 2	1	5.1	0.1	2.2	7.5	1.6	0.1	0.2	1.9	1.3	0.0	0.1	1.5	0.1	136.6
	2	3.4	0.1	1.0	4.7	0.8	0.0	0.0	0.9	1.5	0.0	0.2	1.8	0.1	104.4
	4	8.5	0.3	5.7	14.5	3.9	0.2	1.3	5.5	5.0	0.3	1.4	6.8	0.1	79.3
	7	10.5	0.7	46.2	57.5	2.9	0.3	1.0	4.2	3.7	0.3	1.5	5.6	0.2	59.4
bakery	1	0.7	0.0	0.0	0.8	0.7	0.0	0.0	0.8	0.5	0.0	0.0	0.5	198.5	19.5
	2	0.7	0.0	0.0	0.8	0.7	0.0	0.0	0.8	0.5	0.0	0.0	0.5	125.6	2.6
	4	0.7	0.1	0.0	0.9	0.8	0.1	0.0	0.9	0.5	0.0	0.0	0.5	61.9	2.9
	7	0.8	0.1	0.0	1.0	0.8	0.1	0.0	1.0	0.5	0.0	0.0	0.6	53.3	5.4

Table 3: The comparison of LTL model checking tools (run-times in seconds).

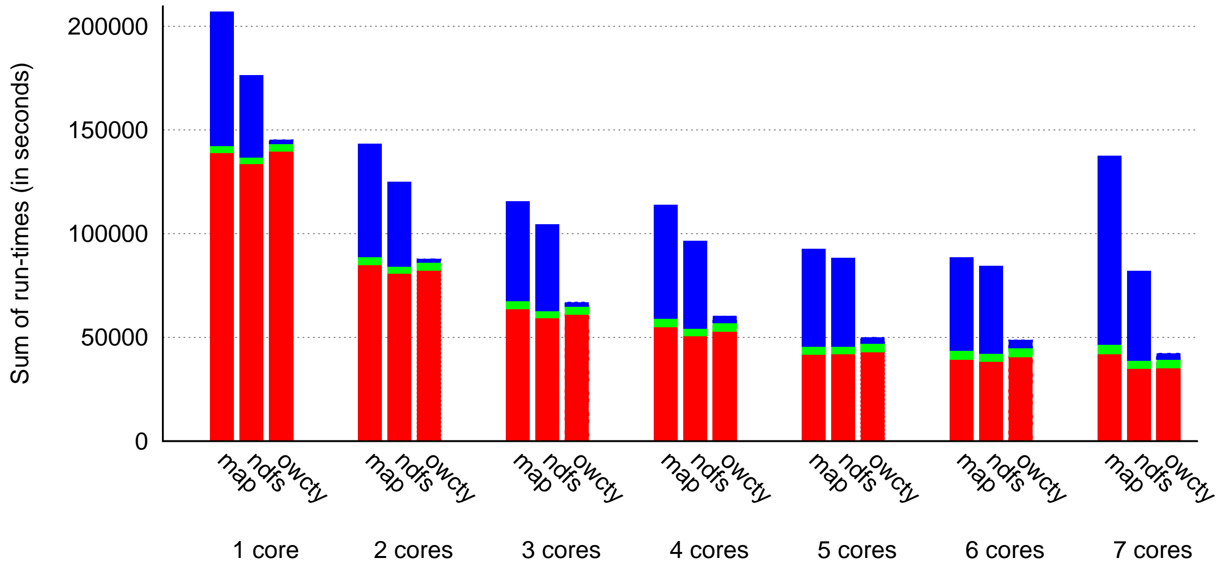


Figure 7: The efficiency of combination of multi-core state space generation and many-core accepting cycle detection. Red bar is for *state space generation*, green for *compact representation preparation* and blue represents the *CUDA Computation*.

which is most likely caused by the usage of shared hash table approach to state space generation which also alters the vertex ordering.

5.3. Experiments on Multi GPU Algorithms

Figures 8a and 8b provide detailed comparison of the relative size of adjacency list representation and the space efficiency for both described methods (see Section 4.2) of multi CUDA computation. Figure 8a depicts the comparison of space efficiency of the two proposed methods on the example of *lamport* model. The first method plainly fails to scale with the number of employed CUDA devices (given the fact that every card has to keep the whole vector of MAP values). The second method scales considerable better, though showing increasing dispersal as the number of devices grows (some cards require more space than others, either for the representation itself or for supplementary arrays in the allocation part – again see Section 4.2 for more details). This phenomenon can be moderated by allowing the preparation phase to be more space and less time efficient.

Figure 8b further illustrates the difference of the two proposed methods with respect to their ability to efficiently utilise space when increasing number of CUDA devices is employed. Considering the fact that the represented average is taken over all tested models, we can conclusively state that the 2nd method can be used to partition a wide variety of graphs, not necessarily limiting its potential competence to model checking graphs. Since we are executing our experiments on a machine with two CUDA devices, the two Figures 8a and 8b represent only the state space partitioning part of the model checking. As they only

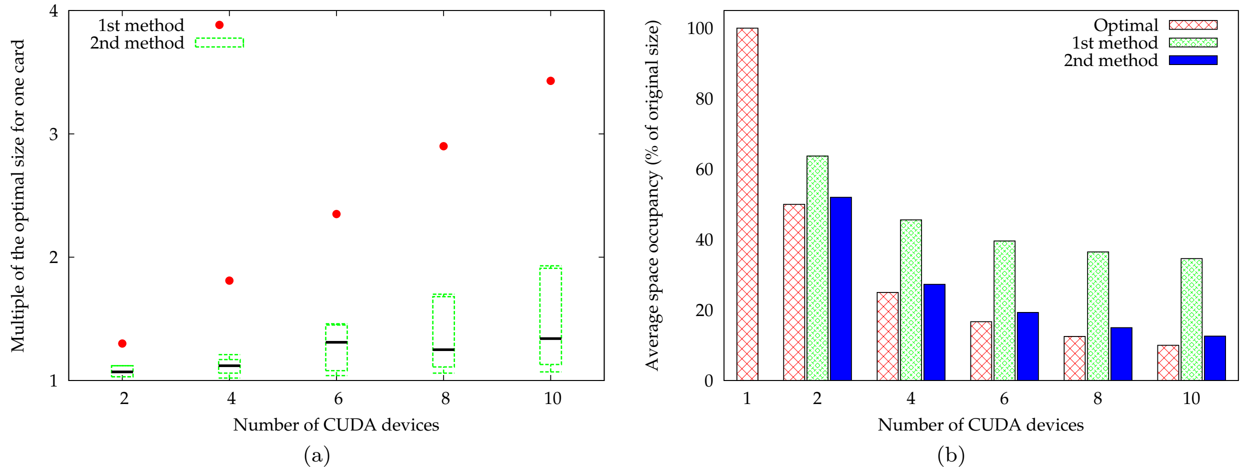


Figure 8: a) The space efficiency for the *lamport* model: depicting the space complexity of respective methods including the variable per-card complexity of the 2nd method. b) The average space efficiency for all models: depicting the average space occupancy per-card of respective methods.

speak about space complexity, however, this is only a noteworthy comment.

In Figure 9 we only provide run-times of algorithms containing the second methods of graph partitioning as it was shown (in [22]) to be only negligibly slower while considerably more space efficient. In the figure there are both dual CUDA algorithms compared to each other and to their single device counterparts. We have detached the adjacency list preparation from the comparison for two reason: to make the differences more apparent and since the state space of some of the models could not be generated on our CUDA-equipped workstation.

The reader should also be aware that the initialisation time of every run contains certain non-trivial overhead (approximately 5 seconds). We have observed that this overhead is caused by serialisation of allocation requests among the two devices. With this knowledge it seems reasonable to state that the slowdown (of dual CUDA computation) caused by inter-CUDA communication is acceptable, especially considering how much time the adjacency list preparation takes.

Unlike OWCTY algorithm, whose multi-device version requires seemingly constant and small number of synchronisation between devices, the MAP algorithm needs considerable more synchronisations for completion. This observation illustrates that not only is OWCTY algorithm insensitive to vertex ordering, it also has a potential to effectively ignore partitioning of the graph. The only exception from this conclusion is the *brp* model on which the OWCTY algorithm had to perform an exceeding amount of eliminations before reaching the fix-point.

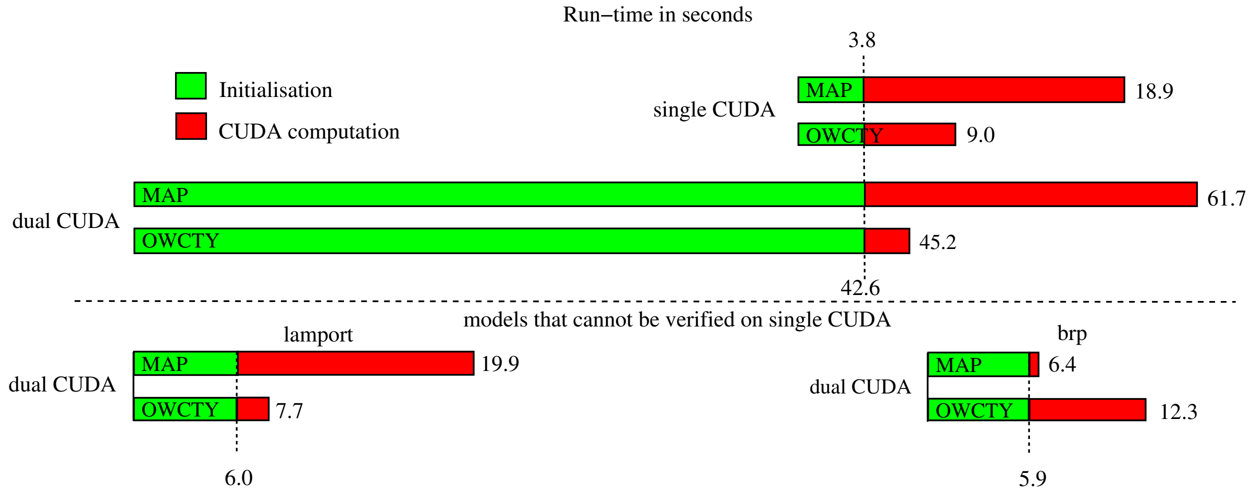


Figure 9: The comparison of single CUDA and dual CUDA algorithms on all models (two of which cannot be verified using a single device).

6. Conclusions

We provided a summary of the latest advancements in GPU acceleration of the LTL model checking. We briefly discussed principal points behind acceleration of parallel accepting cycle detection algorithms (MAP and OWCTY) by means of massivelly parallel processing. Furthermore, we examined the two main bottlenecks of the proposed methods. The first one is that the preparation of adjacency list representation of the models is overly costly thus preventing effective acceleration. The sond one is that the size of the models is constrained by the limited GPU memory. Subsequently, we demonstrated how to overcome these weaknesses.

We designed a parallel multi-core construction of the adjacency list allowing for significant efficienty improvement of the proposed CUDA LTL model checking algorithm. We also established successful employment of multiple CUDA devices to verify considerably larger instances of model checking problems while preserving significant speedup. We showed that the expensive communication among particular CUDA devices and CPU imposed by individual synchronisations leads to only negligible slowdown. These new approaches can be effectively employed on modern multi-core machines equipped with multiple CUDA devices.

Furthermore, we provided a detailed experimental evaluation of our approach and comparison with state-of-the-art model checkers. The experiments show that in the case when model checking is used for “falsification” (the model is invalid, i.e. an accepting cycle is present in the graph) the methods based on DFS exploration of the state space are thriving. The DFS exploration usually locates the part of the state space where an accepting cycle is present much earlier than BFS exploration. Therefore, a significantly smaller part of the state space has to be generated. This leads to substantial acceleration of the whole model checking procedure. On the other hand, this limits the potential of GPU acceleration of accepting

cycle detection (the other part of the model checking procedure), since the detection is executed on a small input graph and thus forms only a negligible part of the overall computation.

In the case model checking is used for "verification" (i.e. no accepting cycle is present in the graph), the exploration strategy has no impact since the whole state space has to be generated. For this reason the performance of accepting cycle detection plays a role of equal importance to that of state space generation. Hence, the GPU acceleration of accepting cycle detection has a chance to significantly speedup the computation. Also, the experiments show that the performance of the GPU accelerated MAP algorithm deeply depends on the ordering in the adjacency list representation and thus it is not as suitable for model checking as the GPU accelerated OWCTY algorithm.

All together it seems that the multi-core state space generation based on shared hash-tables and DFS exploration together with GPU accelerated OWCTY algorithm for accepting cycle detection leads to the best result among the state-of-the-art shared memory model checkers. Even though there were many models in our experiments on which the LTSmin exceeded the performance of DiVinE-CUDA, they were exclusively instances of invalid models. If the intended use of the model checker is verification of correctness of the system instead of falsification, the reported results suggest to employ either DiVinE or DiVinE-CUDA, based on the hardware the user has available.

In the future we would like to put significant effort in designing GPU accelerated state space generation and adjacency list computation which can lead to additional speedup model checking and which we consider to be the next challenge for the parallel model checking community.

Acknowledgement

This research has been supported by the Czech Grant Agency grants No. GP201/09/P497, GA201/09/1389, GAP202/11/0312, and by Artemisia iFEST project grant No. 100203.

References

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

[2] D. Peled, All from One, One from All: on Model Checking Using Representatives, in: Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93), Vol. 697 of LNCS, Springer-Verlag, 1993, pp. 409–423.

[3] D. Bosnacki, S. Leue, A. L. Lafuente, Partial-Order Reduction for General State Exploring Algorithms, International Journal on Software Tools for Technology Transfer 11 (1) (2009) 39–51.

[4] K. L. McMillan, Symbolic Model Checking: an Approach to the State Explosion Problem, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1992).

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

[6] G. J. Holzmann, D. Bosnacki, The Design of a Multicore Extension of the SPIN Model Checker, IEEE Transactions on Software Engineering 33 (10) (2007) 659–674.

[7] J. Barnat, L. Brim, P. Ročkai, Scalable Multi-core LTL Model-Checking, in: Proceedings of the 14th International SPIN Conference on Model Checking Software (SPIN'07), Vol. 4595 of LNCS, Springer, 2007, pp. 187–203.

[8] J. Barnat, L. Brim, P. Ročkai, DiVinE Multi-Core – A Parallel LTL Model-Checker, in: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08), Vol. 5311 of LNCS, Springer, 2008, pp. 234–239.

[9] J. Barnat, L. Brim, P. Ročkai, A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties, in: Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM 2009), Vol. 5885 of LNCS, Springer, 2009, pp. 407–425.

[10] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, P. Šimeček, DiVinE – A Tool for Distributed Verification (Tool Paper), in: Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), Vol. 4144/2006 of LNCS, Springer, 2006, pp. 278–281.

[11] A. Laarman, J. van de Pol, M. Weber, Boosting Multi-Core Reachability Performance with Shared Hash Tables, in: Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD'10), IEEE Computer Society, 2010.

[12] A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, A. Wijs, Multi-Core Nested Depth-First Search, in: Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11), 2011.

[13] S. Blom, J. C. van de Pol, M. Weber, LTSmin: Distributed and Symbolic Reachability, in: Proceedings of the 20th Internation Conference on Computer Aided Verification (CAV'10), Vol. 6174 of LNCS, Springer, 2010, pp. 354–359.

[14] J. Barnat, L. Brim, M. Češka, P. Ročkai, DiVinE: Parallel Distributed Model Checker (Tool paper), Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC'10), IEEE, 2010, pp. 4–7.

[15] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 3.2, `http://developer.nvidia.com/object/cuda_3_2_downloads.html` (February 2011).

[16] S. Edelkamp, D. Sulewski, Model Checking via Delayed Duplicate Detection on the GPU., Tech. Rep. Technical Report 821, TU Dortmund, presented on the 22nd Workshop on Planning, Scheduling, and Design (PuK'08) (2008).

[17] S. Edelkamp, D. Sulewski, Parallel State Space Search on the GPU, in: International Symposium on Combinatorial Search (SoCS'09), 2009.

[18] S. Edelkamp, D. Sulewski, Efficient Explicit-state Model Checking on General Purpose Graphics Processors, in: Proceedings of the 17th International SPIN Conference on Model Checking Software (SPIN'10), Springer-Verlag, 2010, pp. 106–123.

[19] D. Bosnacki, S. Edelkamp, D. Sulewski, Efficient Probabilistic Model Checking on General Purpose Graphics Processors, in: Proceedings of the 16th International SPIN Conference on Model Checking Software (SPIN'09), Vol. 5578 of LNCS, Springer, 2009, pp. 32–49.

[20] J. Barnat, L. Brim, M. Češka, T. Lamr, CUDA accelerated LTL Model Checking, in: Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS'09), IEEE Computer Society, 2009, pp. 34–41.

[21] J. Barnat, L. Brim, M. Češka, DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking, Electronic Proceedings in Theoretical Computer Science (PDMC'09) 14 (2009) 107–111.

[22] J. Barnat, P. Bauch, L. Brim, M. Češka, Employing Multiple CUDA Devices to Accelerate LTL Model Checking, in: Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS'10), IEEE Computer Society, 2010, pp. 259–266.

[23] J. Barnat, P. Bauch, L. Brim, M. Češka, CUDA Accelerated LTL Model Checking - Revisited, in: Proceedings of the 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10), NOVPRESS Brno, 2010, pp. 11–19.

[24] J. Barnat, P. Bauch, L. Brim, M. Češka, Computing Strongly Connected Components in Parallel on CUDA, in: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11), To appear in IEEE Computer Society, 2011.

[25] L. Brim, I. Černá, P. Moravec, J. Šimša, Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking, in: Proceedings of 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04), Vol. 3312 of LNCS, Springer, 2004, pp. 352–366.

[26] I. Černá, R. Pelánek, Distributed Explicit Fair Cycle Detection (Set Based Approach), in: Proceedings of the 10th International SPIN Conference on Model Checking Software (SPIN'03), Vol. 2648 of LNCS, Springer, 2003, pp. 49–73.

[27] A. Lefohn, J. M. Kniss, J. D. Owens, Implementing Efficient Parallel Data Structures on GPUs, in: GPU Gems 2, Addison-Wesley, 2005, pp. 521–545.

[28] P. Harish, P. J. Narayanan, Accelerating Large Graph Algorithms on the GPU Using CUDA, in: Proceedings of the 14th International Conference on High Performance Computing (HiPC'07), Vol. 4873 of LNCS, Springer, 2007, pp. 197–208.

[29] S. Hong, S. K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA Graph Algorithms at Maximum Warp, in: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11), To appear in ACM Press, 2011.

[30] P. Harish, V. Vineet, P. J. Narayanan, Large Graph Algorithms for Massively Multithreaded Architectures, Tech. Rep. IIIT/TR/2009/74, Center for Visual Information Technology, International Institute of Information Technology Hyderabad, INDIA (2009).

[31] A. L. Lafuente, Simplified Distributed LTL Model Checking by Localizing Cycles, Tech. Rep. 00176, Institut für Informatik, University Freiburg, Germany (2002).

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