CUDA Accelerated LTL Model Checking – Revisited*

Petr Bauch[1] **and Milan Češka**[2]

1 **Faculty of Informatics, Masaryk University**
Brno, Czech Republic
`xbauch@fi.muni.cz`

2 **Faculty of Informatics, Masaryk University**
Brno, Czech Republic
`xceska@fi.muni.cz`

Abstract

Recently, the massively parallel architecture has been used to significantly accelerate many computation demanding tasks. For example, in [2, 5] we have shown how CUDA technology can be employed to accelerate the process of Linear Temporal Logic (LTL) Model Checking. In this paper we redesign the One-Way-Catch-Them-Young (OWCTY) algorithm [7] in order to devise a new CUDA accelerated OWCTY algorithm that will significantly outperform the original CUDA accelerated algorithm and will be resistant to slowdown caused by improper ordering of the input data representation.

Digital Object Identifier 10.4230/OASIcs.MEMICS.2010.1

1 Introduction

Model checking [1] is a wide-spread technique for automated formal verification of parallel and distributed software and hardware systems. For a given formal description of a system and desired system property, the goal of the model checking procedure is to analyse reachable system configurations in order to decide whether the system satisfies the property or not. The model checking technique generally suffers from the so called *state space explosion problem* that makes 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 industrial systems is rather limited.

A possible way to reduce the delay due to the formal verification process is to accelerate computation of verification tools using contemporary parallel hardware. Hardware platforms such as multi-core multi-CPU systems or many-core hardware accelerators have recently received a lot of attention in this aspect. At the leading edge of this class of massively parallel chip architectures are the modern Graphics Processing Units (GPU). 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 utilisation 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 target LTL model checking, where the property to be verified is given as a formula of Linear Temporal Logic (LTL). The problem of LTL model checking can be

* This work has been supported in part by the Czech Grant Agency grants No. 201/09/1389 and 102/09/H042.

reduced to the problem of detection of an accepting cycle (cycle containing vertex denoted as accepting) in a directed graph. In our previous work [5] we have redesigned the maximal accepting predecessors (MAP) algorithm [6] for detection of an accepting cycle 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 speedup of verification process. The proposed method exhibits two weaknesses. First, it is the very expensive phase of preparation of data structures for consecutive CUDA processing, and second, the limited size of the state space that can fit the memory of a single CUDA device.

Further we have shown [2] that the expensive phase of encoding the state space into the appropriate representation can be itself accelerated by means of multi-core parallel processing followed by a few CUDA operations and second, we have shown how to employ multiple CUDA devices to overcome the memory limitations of a single device. Although preserving a decent efficiency of our inter-CUDA communication intensive parallel algorithm for LTL model checking, the proposed methods may affect the ordering of the representation which subsequently causes significant slowdown of the overall CUDA computation of the MAP algorithm.

In this paper we redesign the One-Way-Catch-Them-Young (OWCTY) algorithm [7] in order to devise a new CUDA accelerated OWCTY algorithm, both superior to the previous MAP algorithm in speed and robust to improper ordering in the representation.

2 Preliminaries

2.1 LTL Model Checking

To answer an LTL model checking question, the model checking tools, such as `SPIN` [9] or `DiVinE` [3], employ the automata-theoretic approach to LTL model checking, which allows to reduce the LTL 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 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 behaviours 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. In [5] we have adapted the MAP algorithm [6] to allow for CUDA accelerated LTL model checking. The main idea behind this algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor. The algorithm computes a single representative accepting predecessor for each vertex. We presuppose a linear ordering $<$ of vertices (given e.g. by their memory representation) and choose the maximal accepting predecessor. If a vertex is its own maximal accepting predecessor the presence of an accepting cycle is guaranteed. If there is an accepting cycle in the graph, but none of the vertices is its own maximal accepting successor, then the maximal accepting predecessor 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 algorithm iteratively computes the maximal accepting predecessor for all the vertices until an accepting cycle is found or the set of accepting vertices becomes empty.

Another parallel algorithm for accepting cycle detection is One-Way-Catch-Them-Young (OWCTY) algorithm [7]. The key idea of the algorithm is maintaining an approximating set of states that may lie on an accepting cycle in the graph G. The algorithm repeatedly refines the approximating set by locating and removing states that cannot lie on any accepting cycle. The algorithm employs two rules to remove vertices from the approximating set: 1. vertices not reachable from any accepting vertex (vertices in the set F) and 2. vertices having zero in-degree.

The basic scheme of the OWCTY algorithm is given in Algorithm 1. The function Reachability(S) computes the set of all vertices that are reachable from the set S. The function Elimination(S) successively eliminates those vertices that have zero in-degree. The assignment on line 5 removes from the graph the vertices according to the 1. rule. The assignment on line 6 removes from the graph the vertices according to the 2. rule. The `while` loop terminates when fixpoint of the approximating set is reached. In the case that the approximating set is nonempty the presence of an accepting cycle is guaranteed. Moreover, we can weaken the termination condition in the following way:

Algorithm 1 *OWCTY*

```
proc OWCTY($G = (V, E), F \subseteq V, init\_state \in V$)
1: $S \leftarrow$ Reachability($init\_state$)
2: $old \leftarrow \emptyset$
3: while $S \neq old$ do
4:     $old \leftarrow S$
5:     $S \leftarrow$ Reachability($S \cap F$)
6:     $S \leftarrow$ Elimination($S$)
7: end while
8: return $S \neq \emptyset$
```

▶ Proposition 1. Elimination(S) $= S$ is a correct termination condition of Algorithm 1.

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. ◀

2.2 CUDA Architecture

The Compute Unified Device Architectures (CUDA) [8], developed by NVIDIA, is a parallel programming 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. 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

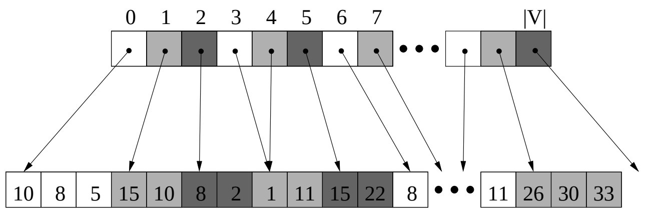

Figure 1 Adjacency list representation: $G = (V, E)$ is stored as two arrays of sizes $|V| + 1$ and $|E|$.

program consists of a *host* code running on the 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*.

Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment, a group of threads called a *warp* execute 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.

2.3 CUDA Accelerated MAP Algorithm

To realise efficiently any CUDA-aware graph algorithm needs the graph to be represented in a compact, preferably vector-like, fashion. The MAP algorithm employs a variant of adjacency list representation, resembling *Compressed Sparse Row* (CSR) representation as illustrated in Figure 1. See [5] for more details. The key idea of the acceleration of the MAP algorithm lies in the parallel computation of the maximal accepting predecessor for all the vertices. We have devised a CUDA kernel that updates the values of the maximal accepting predecessors along the corresponding outgoing edges simultaneously for all vertices in the graph. See [5] for more details. Besides the data structure for representing the graph, the CUDA algorithm has to maintain another data structure to store the MAP values – a vector. Data manipulation thus resembles a sparse matrix (graph) vector (values of maximal accepting predecessor) multiplication pattern, which is known to be convenient for CUDA acceleration.

Our CUDA accelerated approach to LTL model checking exhibited certain weaknesses as already mentioned in [5]. Among other aspects it was the costly preparation of data structures for consecutive CUDA processing. Though we have diminished the size of this problem considerably by means of multi-core parallelisation [2], a new flaw consequently emerged. The altered ordering in the CSR representation has shown less efficient for the MAP algorithms. To understand why, we should point out that we are actually computing minimal accepting successors. Considering successors allows us to store only the forward edges and preferring smaller values inverts the BFS ordering enforced by generation (actual BFS ordering provided significantly worse results). This observation can then be explained by existence of paths going out of accepting cycles: prolonging search for maximal successor and preventing termination when one is found. While avoided by order inversion, this aspect seems to be partially restored when generation is done concurrently. The following CUDA accelerated OWCTY algorithm should prove more resistant to any improper ordering in CSR representation.

3 CUDA Accelerated OWCTY Algorithm

The non-CUDA version of OWCTY algorithm comprises of alternating execution of forward reachability and *backward elimination* (Algorithm 1). 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 new implementation. Their data-parallel versions to be precise.

Implementation of reachability was given sufficient space in [2] (where referred to as closure computation). We will thus in the following concentrate 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 2). The first step is performed by the CUDA kernel PROGRESS, 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 CHECK which eliminates vertices without this property. Finally, the operations ELIM, SETELIM, etc. are low-level bitwise operations on a piece of memory assigned to every vertex, which allows them 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 3). The basic layout is equivalent to the original implementation. The CUDA kernel VISACCEPTING sets all accepting vertices to visited. Having considered the Proposition 1, we need not to test if REACHABILITY visited all vertices. Only its effect, the elimination of non-visited vertices is necessary (via kernel TESTSET). The elimination proceeds as described above. Furthermore, if no vertex is eliminated (line 5) the algorithm terminates with resulting value stored in variable $found$. It is observable that $found$ keeps track of existence of 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. Though as stated in [2] backward reachability via forward edges is securable (with certain slowdown), allowing us to implement elimination in the trivial way as sketched above. The rest of the algorithm remains the same and the resulting efficiency of both implementation is compared in Section 5.

4 Early Termination and Combination of Algorithms

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 CUDA accelerated OWCTY algorithm to mimic this behaviour as well. The idea is very similar as in our previous papers [2, 5]. In particular, we let the CPU perform (parallel) state space generation while having the GPU apply CUDA accelerated OWCTY algorithm on partially constructed graph. If the part of the graph constructed so far contains an accepting cycle, CUDA accelerated OWCTY algorithm 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 back-

Algorithm 2 *Backward Elimination*

```
1: while $change$ do
2:    Progress($V$)
3:    $change, found \leftarrow$ false
4:    Check($V, change, found$)
5:    $result \leftarrow change$ ? true : $result$
6: end while

kernel Progress($V$) // run in data-parallel fashion on all $v \in V$ at once
7: if $\neg$Elim($v$) then
8:    for all $u \in$ Succ($v$) do
9:       if $\neg$Elim($u$) $\wedge$ ElimPrep($u$) then
10:         UnsetElimPrep($u$)
11:      end if
12:   end for
13: end if

kernel Check($V, change, found$) // again on all $v \in V$ at once
14: if $\neg$Elim($v$) then
15:    if ElimPrep($v$) then
16:       SetElim($v$)
17:       $change \leftarrow$ true
18:    else
19:       SetElimPrep($v$)
20:       $found \leftarrow$ true
21:    end if
22: end if
```

Algorithm 3 *CUDA OWCTY*

```
1: VisAccepting($V$)
2: while $result$ do
3:    Reachability($V$)
4:    TestSet($V$)
5:    $result \leftarrow$ false
6:    Elimination($V, found, result$)
7: end while
8: return $found$
```

ground 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).

5 Experimental Evaluation

We have implemented both variants of CUDA accelerated OWCTY algorithm as a part of DiVinE-CUDA [4]. We compared the performance of these algorithms against the original CUDA accelerated MAP algorithm [2, 5].

All the experiments were run on a Linux workstation with a quad core AMD Phenom(tm) II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066 MHz RAM and two NVIDIA GeForce GTX 280 GPU's with 1GB of GPU memory.

Table 1 provides details on run-times of the algorithms. The total run-time includes the

Models (seq. total time: MAP/OWCTY)		CPU cores	*CSR time*	CUDA MAP		CUDA OWCTY		CUDA OWCTY REVERSE	
				CUDA time	*total time*	*CUDA time*	*total time*	*CUDA time*	*total time*
without accepting cycle	elevator 1 (100/41)	1	24.5	6.0	31.6	0.7	26.3	0.2	**25.8**
		2	15.2	5.8	22.3	0.8	17.3	0.3	**16.8**
		3	12.1	6.1	19.2	1.2	14.3	0.3	**13.4**
	leader (697/297)	1	86.0	0.1	**87.4**	1.1	88.4	0.8	88.1
		2	49.1	4.2	54.5	2.2	52.5	1.2	**51.5**
		3	35.4	9.3	45.2	4.3	40.2	1.3	**37.2**
	peterson 1 (445/188)	1	97.9	3.5	102.3	1.0	99.8	0.5	**99.3**
		2	58.3	9.5	69.6	1.8	61.9	0.7	**60.8**
		3	41.5	10.0	52.7	2.1	44.8	0.8	**43.5**
	anderson (115/113)	1	30.6	1.5	33.2	0.5	32.2	0.2	**31.9**
		2	19.5	1.6	22.4	0.5	21.3	0.3	**21.2**
		3	15.5	4.4	21.6	1.2	18.4	0.4	**17.6**
with accepting cycle	elevator 2 (50/177)	1	27.2	0.6	28.7	1.2	29.3	0.5	**28.6**
		2	19.5	0.9	21.5	1.8	22.4	0.6	**21.2**
		3	14.6	0.9	16.4	2.0	17.5	0.5	**16.0**
	phils (397/576)	1	45.2	< 0.1	46.1	< 0.1	46.1	< 0.1	46.1
		2	29.6	< 0.1	30.3	0.1	30.4	< 0.1	30.3
		3	20.8	< 0.1	21.6	0.1	21.7	< 0.1	21.6
	peterson 2 (173/404)	1	25.7	4.0	30.5	0.4	26.9	0.3	**26.8**
		2	17.4	4.3	22.5	0.6	**18.8**	0.8	19.0
		3	12.5	0.6	**13.8**	1.2	14.4	1.0	14.2
	bakery (240/907)	1	22.1	< 0.1	**23.2**	0.4	23.6	0.2	23.4
		2	13.5	< 0.1	**14.4**	0.5	14.9	0.3	14.7
		3	6.2	< 0.1	**7.3**	0.8	8.1	0.1	7.4

Table 1 The overall run-times of the algorithms in seconds.

initialisation time (not reported in the table), CSR construction time (*CSR time*) and time spent on CUDA computation (*CUDA time*). 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 CSR construction.

We have extended the table presented in [2] by the times for both variants of CUDA accelerated OWCTY algorithm. We can see that the reversed variant of CUDA accelerated OWCTY algorithm has better times that the standart 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 can further see that both variants of CUDA accelerated OWCTY algorithm significantly outperform the original CUDA accelerated MAP algorithm on most valid model checking instances (without accepting cycle). Also on most of the invalid instances (with accepting cycle) the reversed OWCTY algorithm has slightly better times than the MAP algorithm. Moreover, on `peterson 2` the MAP algorithm falls behind both the OWCTY algorithms significantly. The reason is that the performance of CUDA accelerated MAP algorithm deeply depends on the ordering in CSR representation which directly affects the number of calls to CUDA kernels [2, 5].

The improper ordering in CSR representation is even more crucial in case of multi-core acceleration of CSR representation. The parallel CSR construction usually affects the ordering and can lead to slowdown of CUDA computation as in the case of `leader`. The experiments show that the performance of the OWCTY algorithms does not depend on the ordering in CSR representation as much as the MAP algorithm. All together it seems that when the multi-core acceleration of CSR representation is utilised the reversed variant of the OWCTY algorithm is clearly a winner for CUDA computation.

6 Conclusions

We have demonstrated that the new CUDA accelerated OWCTY algorithms outperform the original MAP algorithm on valid instances of model checking problems. Moreover, the reversed variant of OWCTY algorithm has slightly better times also on invalid instances. The experiments also show that in opposite to MAP algorithm the OWCTY algorithm is resistant to improper ordering in CSR representation. This is particularly important when the order affecting multi-core acceleration of data preparation is applied. In the future we would like to include also the state space generation in CUDA acceleration thus allowing the whole model checking procedure to fully utilise the parallel potential of many-core architectures.

References

1 Christel Baier and Joost P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

2 J. Barnat, P. Bauch, L. Brim, and M. Češka. Employing Multiple CUDA Devices to Accelerate LTL Model Checking. In *16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, pages 259–266. IEEE Computer Society, 2010.

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

4 J. Barnat, L. Brim, and M. Češka. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. *EPTCS (PDMC 2009)*, 14:107–111, 2009.

5 J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA accelerated LTL Model Checking. In *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 34–41. IEEE Computer Society, 2009.

6 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.

7 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.

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

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