MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Designing Data-Parallel Graph Algorithms for Model Checking

RNDr. Milan Češka

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy at
the Faculty of Informatics, Masaryk University

Brno, Czech Republic, April 2012

Acknowledgement

I would like to thank all the people who have helped me to finish my PhD studies. First, I would very much like to thank my supervisor prof. RNDr. Luboš Brim, CSc. for his guidance and help throughout my PhD research. Second, I would like to thank my consultant doc. RNDr. Jiří Barnat, PhD for many pieces of helpful advice and fruitful discussions.

Also I thank all members of ParaDiSe laboratory at the Faculty of Informatics, Masaryk University, for willingness to help me whenever problems with this work arose and the people I met during may stay in Bamberg and had the pleasure to work with. In particular, I would like to thank Jana Tůmová and Petr Bauch for inventive and lengthy discussions on various kinds of subjects, for helping me with the implementation works and for reading the draft of my thesis.

And many great thanks go also to my family, girlfriend and all friends for supporting me during the work on this thesis and for their nearly unlimited patience.

Abstract

Model checking 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 analyze a graph of all reachable configurations in order to decide whether the model satisfies the property or not. The model checking techniques generally suffer from the so-called *state space explosion problem* that causes the graphs to be very large for realistic systems. Therefore, the performance of fundamental graph algorithms (breadth-first search, shortest path detection, strongly connected component decomposition, accepting cycle detection, etc.) that form building blocks of the model checking algorithms is crucial. However, sequential implementations of these algorithms become impractical for extremely large graphs to be processed. As a result, parallel graph algorithms have been devised to efficiently use computer clusters and multi-core architectures. Even though the algorithmic shift to parallel processing is possible, sequential code needs to be rewritten to take proper advantage of parallel architectures. This especially applies to recently introduced massively parallel general purpose graphics processing units (GPUs). These devices contain hundreds of arithmetic units and can be harnessed to provide tremendous acceleration for many computation-intensive scientific applications. The key to effective utilization of GPUs for scientific computing is the design and implementation of efficient data-parallel algorithms that can scale to hundreds of tightly coupled processing units.

In this thesis, we primarily focus on data-parallel graph algorithms for model checking. However, our approach is more general and applicable to other graph algorithms. We discuss and propose how to design efficient data-parallel algorithms for accepting cycle detection, strongly connected component decomposition, optimal cycle detection and graph-based resolution of boolean equation systems. In particular, we design basic data-parallel graph primitives and show how the aforementioned algorithms can be built from them.

While the raw computing power of massively parallel GPU devices is tremendous, its effective utilization is, however, quite often reduced by the costly preparation of suitable data structures and limited to small or middle-sized instances due to space restrictions. Hence, we further suggest how to overcome these limitations using multi-core construction of the compact data structures and employing multiple GPU devices for acceleration of fine-grained communication-intensive parallel algorithms.

In order to evaluate the efficiency of the proposed techniques we experimentally demonstrate the performance of the suggested data-parallel graph algorithms and compared them with best sequential counterparts. Our experiments show that the proposed GPU acceleration results in a significant speedup of the graph algorithms. Moreover, the experimental evaluation positions our GPU accelerated DiVinE-CUDA tool (LTL model checker based on our data-parallel algorithm for accepting cycle detection and on a novel multi-core construction of the compact state space representation) as the fastest among the state-of-the-art parallel LTL model checkers using an unbiased selection of model checking instances.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Computer-aided technologies have become ubiquitous in most areas of our lives and have been frequently applied to control areas where any failure or mistake may have fatal consequences. Examples of such technologies are computer systems controlling aircrafts, high speed trains, space stations, or critical industrial systems such as nuclear power plants or electricity distribution networks. Consequences of failures of these systems can be illustrated by a number of real accidents (the failure of the maiden flight of Ariane 5 in 1996, four NASA Mars missions failing between 1997 and 2004, the US Northeast blackout in 2003, or the malfunction of the autopilot of a Boeing 777 aircraft during a regular passenger flight in 2005).

Therefore there is a strong pressure on developing formal verification methods for automated discovery of errors in computer systems and/or for proving their correctness. This is reflected in many basic research projects supported by various American and European grant agencies, in the existence of research teams focusing on formal verification within many leading companies and institutions (such as Microsoft, IBM, or NASA) as well as in the emergence of various spin-off companies working in the area of automated verification (e.g., Coverity, GrammaTech, or AbsInt). A successful application of formal verification to industrial systems can be illustrated by Intel Core i7 processor execution engine validation, where testing has been replaced by verification techniques [73].

Techniques of formal verification employed in the current verification tools, typically include static analysis, model checking, and theorem proving. In this thesis we primarily focus on model checking [6], a technique for automated formal verification of software and hardware systems. For a given formal description (represented as a finite-state model) of a system and desired system property (expressed as a formula of a temporal logic), the goal of the model checking procedure is to decide whether the system satisfies the property or not by systematically analyzing a graph of reachable system configurations.

Although model checking has been effectively employed in the design process of several real industrial systems (e.g., model checking of complex avionics systems reported in [92]), the model checking techniques generally suffer from the so-called *state space explosion problem*. This causes that the gap between the complexity of systems built in practice and the complexity of systems the current formal verification tools can handle is still quite wide.

The reason behind the gap is that the graphs to be analyzed tend to be very large for realistic systems. As a result, sequential implementations of fundamental graph algorithms (breadth-first search, shortest path detection, strongly connected component decomposition, accepting cycle detection, etc.) that form the building blocks of the model checking procedure become impractical for extremely large graphs to be processed.

A lot of research has aimed at developing methods that can reduce the size of the resulting graphs. The most successful techniques are the partial order reduction [98, 34], symbolic representation [90], or restricting the exploration of the state space only to relevant traces using on-the-fly generation [104]. However, these methods are often limited to a specific class of systems and they do not provide sufficient reduction in the size of the graph.

In the last decade a different approach to fight the state space explosion problem has attracted the model checking community – scalable distributed computation of the fundamental graph algorithms using contemporary parallel hardware platforms that can improve the performance of verification tools. Therefore, many techniques using distributed and parallel processing have been developed to deal with the computational complexity of model checking procedure. The primary goal here is to extend the available memory to handle larger verification problems. Nevertheless, generating and analyzing large state spaces calls for acceleration using parallel algorithms in order to obtain the desired level of performance. Hence new parallel graph algorithms have been designed [39, 35, 77, 78] and incorporated into the model checking tools [16, 12, 28, 1].

These algorithms were designed assuming the parallelism as provided by multi-core shared-memory or distributed-memory architectures. As such they are based on coarse-grain parallelism mapping multiple independent task to individual cores. The aim is to deliver higher performance by exploiting modestly parallel workloads. However, a related architectural trend is the growing prominence of massively parallel, throughput-oriented hardware architectures that arise from the assumption that they will be presented with fine-grain workloads in which parallelism is abundant [59]. At the leading edge of this class of massively parallel architectures there are the modern Graphics Processing Units (GPUs). GPUs have emerged as a revolutionary technological opportunity due to their tremendous massive parallelism, floating point capability, low cost, and ubiquitous presence in commodity computers. Many fundamental algorithms have been redesigned to exploit the performance of this hardware. The reason is that programming massively parallel, throughput-oriented processors requires much more emphasis on parallelism and scalability than in the case of multi-core or sequential processors. The key to utilization of GPUs for scientific computing is the design and implementation of efficient *data-parallel algorithms* that can scale to hundreds of tightly coupled processing units.

NVIDIA's CUDA technology [49] has started an avalanche of prototypes and research results demonstrating the application of massive parallelism in many scientific fields. Recently, GPUs have been successfully used to accelerate many compute-intensive and high-performance applications in different areas such as image processing, simulation and modeling of biological systems, numerical calculations and many others. Also some fundamental computational building blocks such as basic graph algorithms [62, 91], sorting [103, 83], and sparse matrix-vector multiplication [43] were successfully adapted to massively parallel architectures the modern GPUs offer.

In the field of model checking the application of massive fine-grain parallelism needs to be considered as well in order to push the limits further towards industrial strength verification techniques, allowing to deal with larger state spaces and providing rapid feedback to the developer. Therefore in this thesis, we aim at developing methods that allow to efficiently employ massively parallel architectures in order to significantly speed up the model checking techniques. In particular, we design fast data-parallel graph algorithms that form the building blocks for our GPU accelerated model checking algorithms.

A successful utilization of modern GPUs in the context of model checking have been recently demonstrated in [32, 33]. Authors have shown how to significantly accelerate the computation of linear equation systems in order to speed up model checking of probabilistic systems. Another application of massive parallelism closely related to our work has been recently published in [54], where GPUs are used to accelerate state space generation.

1.1 Contribution of the Thesis

This thesis contributes to the area of designing fast data-parallel graph algorithms in the context of model checking. It presents new methods and algorithms that enable to efficiently utilize modern massively parallel architectures in order to significantly accelerate the model checking process. The thesis is mainly based on our results recently published in several journal, conference and workshop papers (see Sections 1.2 and 1.3 for more details). The thesis provides a new common-ground presentation of our previously published results. The contributions of this thesis are manyfold.

- We present new efficient data-parallel algorithms for accepting cycle detection, strongly connected component decomposition, optimal cycle detection and graph-based resolution of boolean equation systems.

- We describe new techniques that allow to efficiently utilize the data-parallel algorithms in the context of model checking. We present a new multi-core construction of the compact data structures that overcomes their costly preparation. We also successfully demonstrate how to employ multiple GPU devices for acceleration of fine-grained communication-intensive parallel algorithms for LTL model checking that overcomes the space limitation of the current GPUs.

- We present a new approach for designing fast data-parallel graph algorithms. We keep the provably correct layout of the existing algorithms and redesign the algorithms to enable vector processing. In particular, we reformulate the recursion present in the algorithms by means of iterative procedures, design basic data-parallel graph primitives and show how the aforementioned algorithms can be built from them. This approach is more general and applicable to other graph algorithms. It aims at enabling implementation of algorithms on various vector models of computation and propounds a way how could other graph algorithms be altered to benefit from massively parallel architecture.

- We experimentally evaluate the performance of the designed data-parallel algorithms and compare them with the best sequential counterparts. Our experiments demonstrate that the proposed GPU accelerated algorithms significantly outperform the best sequential counterparts and thus in total speed up the solution of the inspected graph problems.

- We deliver the DiVinE-CUDA tool that implements the designed GPU accelerated algorithms for LTL model checking and experimentally evaluate its performance. The experimental evaluation positions our DiVinE-CUDA tool as the fastest among the state-of-the-art parallel LTL model checkers using an unbiased selection of model checking instances.

1.2 Author's Contribution

The overall contribution of the author to the research in computer science is summarized in this section. Note that not all published results are mentioned in this thesis.

Journal papers

[10] J. Barnat, P. Bauch, L. Brim, and **M. Češka**. Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. *To appear in Journal of Parallel and Distributed Computing*, 2012. [Author's contribution: 60%]

Conference papers

[9] J. Barnat, P. Bauch, L. Brim, and **M. Češka**. Computing Strongly Connected Components in Parallel on CUDA. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, pages 541–552. IEEE Computer Society, 2011. [Author's contribution: 60%]

[7] J. Barnat, P. Bauch, L. Brim, and **M. Češka**. Computing Optimal Cycle Mean in Parallel on CUDA. In *Proceedings of 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'11)*, volume 72 of *Electronic Proceedings in Theoretical Computer Science*, pages 68–83, 2011. [Author's contribution: 50%]

[26] P. Bauch and **M. Češka**. CUDA Accelerated LTL Model Checking - Revisited. In *Proceedings of the 6th International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10) – Selected Papers*, volume 16 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–8, 2011. [Author's contribution: 70%]

[8] J. Barnat, P. Bauch, L. Brim, and **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)*, pages 259–266. IEEE Computer Society, 2010. [Author's contribution: 60%]

[21] J. Barnat, L. Brim, **M. Češka**, and T. Lamr. CUDA accelerated LTL Model Checking. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS'09)*, pages 34–41. IEEE Computer Society, 2009. [Author's contribution: 50%]

[18] J. Barnat, L. Brim, I. Černá, **M. Češka**, and J. Tůmová. Local Quantitative LTL Model Checking. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *LNCS*, pages 53–68. Springer, 2008. [Author's contribution: 30%]

Tool papers

[22] J. Barnat, L. Brim, **M. Češka**, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Proceedings of the 9th International Workshop on Parallel and Distributed Methods in verifiCation and the 2nd International Workshop on High Performance Computational Systems Biology (PDMC/HiBi'10)*, pages 4–7. IEEE Computer Society, 2010. [Author's contribution: 30%]

[20] J. Barnat, L. Brim, and **M. Češka**. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. In *Proceedings of 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'09)*, volume 14 of *Electronic Proceedings in Theoretical Computer Science*, pages 107–111, 2009. [Author's contribution: 50%]

[19] J. Barnat, L. Brim, I. Černá, **M. Češka**, and J. Tůmová. ProbDiVinE-MC: Multi-core LTL Model Checker for Probabilistic Systems. In *Proceedings of the 5th International Conference on Quantitative Evaluation of Systems (QEST '08)*, pages 77–78. IEEE Computer Society, 2008. [Author's contribution: 30%]

[17] J. Barnat, L. Brim, I. Černá, **M. Češka**, and J. Tůmová. ProbDiVinE: A Parallel Qualitative LTL Model Checker. In *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST'07)*, pages 215–216. IEEE Computer Society, 2007. [Author's contribution: 20%]

Work in progress reports

[11] J. Barnat, L. Brim, I. Černá, **M. Češka**, and J. Tůmová. Distributed Qualitative LTL Model Checking of Markov Decision Processes. In *Proceedings of 5th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'06)*, pages 1–15. University of Bonn, 2006. [Author's contribution: 20%]

This thesis is mainly based on the results published in [10, 9, 7, 26, 8, 21, 20]. These results arose during PhD research of the author of this thesis who also has the major contribution to these publications. The contribution includes formulation of key ideas of the designed algorithms and methods, significant part of implementation works, analysis of experimental evaluations, formulation of conclusions and significant part of writing. However, the results would not come into existence without advisor prof. L. Brim and consultant doc. J. Barnat who initiated and motivated this research. Important contributions to these results have been done by master's students T. Lamr and P. Bauch who participated on the prototype implementations of the designed methods and algorithms.

Other published results are based on the joint work by the authors of the publications who equally contributed to these results. This work closely relates to the thesis since it presents new parallel algorithms and methods for LTL model checking and for analysis of probabilistic systems. The work also provided to the author of this thesis an important background and motivation for the research in the area of the parallel formal verification.

1.3 Outline of the Thesis

This thesis is organized as follows:

Chapter 2 provides a background of this thesis. It presents key characteristics of modern massively parallel Single Instruction Multiple Data (SIMD) architectures that form target hardware platforms for proposed data-parallel algorithms. The chapter also introduces selected graph algorithms and model checking techniques whose parallelization is studied in the context of this thesis.

Chapter 3 describes a general work-flow of data-parallel graph algorithms. It introduces suitable data representation and heterogeneous computation work-flow that are demonstrated on GPU accelerated graph traversal. It also discusses the main limitations of the parallelization and studies recently proposed methods that try to overcome these limitations. Finally, it provides a simple model of data-parallel processing of graph algorithms that enables to evaluate how efficiently can modern GPUs handle the proposed way of data-parallel processing of graph algorithms. The model also justifies our motivation to accelerate the graph algorithms on massively parallel SIMD architectures. This chapter is partly based on our work published in [10, 9, 21].

Chapter 4 presents data-parallel algorithms for accepting cycle detection, namely the data-parallel versions of the MAXIMAL-ACCEPTING-PREDECESSOR and ONE-WAY-CATCH-THEM-YOUNG algorithms. These algorithms provide building blocks for GPU accelerated LTL model checking. This chapter summarizes our results published in [26, 20, 21].

Chapter 5 shows how to employ the proposed data-parallel algorithms for accepting cycle detection in order to accelerate the LTL model checking process on modern many-core GPUs. It describes an efficient transformation from implicit graph representation to a form suitable for GPU computation and for overcoming of the GPU memory limitation. Finally, it provides an experimental comparison of our GPU accelerated model checker with others state-of-the-art tools. This chapter contains our results published in [10, 8, 20, 21].

Chapter 6 describes a design of data-parallel algorithms for strongly connected component decomposition. It shows how to decompose the existing algorithms into primitive data-parallel graph operations and how to reformulate the recursion present in these algorithms by means of iterative procedures. It also describes a new data-parallel procedure for pivot selection and provides an experimental evaluation of the proposed algorithms and their comparison with optimal inherently sequential TARJAN'S algorithm. This chapter is based on the results published in [9].

Chapter 7 presents a data-parallel algorithm for the optimal cycle mean problem, namely data-parallel version of HOWARD'S algorithm. It evaluates all existing classes of algorithms for this problem with respect to their predisposition for vector processing. It also provides an experimental comparison with an optimal inherently sequential algorithm given by Young Tarjan and Orlin in the context of performance verification based on the optimal cycle mean detection. This chapter is based on the results published in [7].

Chapter 8 studies parallel resolution of Boolean Equations Systems (BESs) in the context of model checking of alternation-free μ-calculus. It describes a data-parallel fixpoint computation of the resolution of BESs and provides an experimental evaluation of both multi-core and many-core implementations and their comparison with optimal algorithm based on the *chasing one* technique. This chapter summarizes the unpublished result that has been done in cooperation with A. Ditter and prof. G. Lüttgen and that is submitted for publication.

Chapter 9 concludes the thesis and discusses the future work in this area.

Chapter 2

Background

In this chapter, we first briefly discuss the key characteristics of modern massively parallel architectures that form the target architectures for proposed data-parallel graph algorithms. Afterwards we provide the necessary definition from graph theory and introduce selected graph algorithms. Finally, we shortly present and motivate the model checking problem and the problem of performance verification and describe specific approaches for these problems based on the introduced graph algorithms.

2.1 Massively Parallel Architectures

The potential of Single Instruction Multiple Data (SIMD) parallelism (first efficiently employed in the Connection Machines [65]) was recently rediscovered with the entry of affordable massively parallel Graphics Processing Units (GPUs). The modern GPUs provide off-the-shelf data-parallel computation capability which has been soon utilized by the academic community and has led to acceleration of various scientific computations, concisely reported for example in the GPU gems series [57, 99, 94].

The modern GPUs represent a successful example of the massively parallel architectures where a massive number of relatively simple in-order processors perform a sequence of instruction on many data elements. The overall approach to parallelism is considerably distinct from the one assumed by CPU computation. While in shared memory parallelism are complex out-of-order processors potentially communicating via the shared memory it is not the case in SIMD computation.

The Compute Unified Device Architecture (CUDA) [49], developed by NVIDIA, further improves the SIMD approach by establishing a hierarchy of threads prior to the actual computation. Within this hierarchy, threads are arranged in *blocks*. The threads are hardwired into groups of 32 called *warps* that form a basic scheduling unit. All threads within a block may communicate via the shared memory and also an efficient barrier synchronization (`syncthreads`) is limited to the given block. The aforementioned improvement on the SIMD approach lies in its evolution into *single instruction multiple thread* (SIMT) approach. Since to every warp a thread scheduler is assigned (or two for the latest generations of devices), allowing the threads in other warps to perform another sequence of instructions, thus effectively improving the robustness of the previously much more restricted applicability of data-parallelism. Furthermore, if a sufficient number of threads is dispatched, the thread scheduler can issue execution of another warp, while the current warp is accessing the global memory to hide the extensive latency of this operation. Figure 2.1 illustrates the GeForce architecture which is one of the-state-of-the-art generation of the CUDA architecture.

The other aspect of the CUDA architecture is its software side, i.e. the modification of the C programming language, the *nvcc* compiler for device code, the CUDA debugger, etc. As

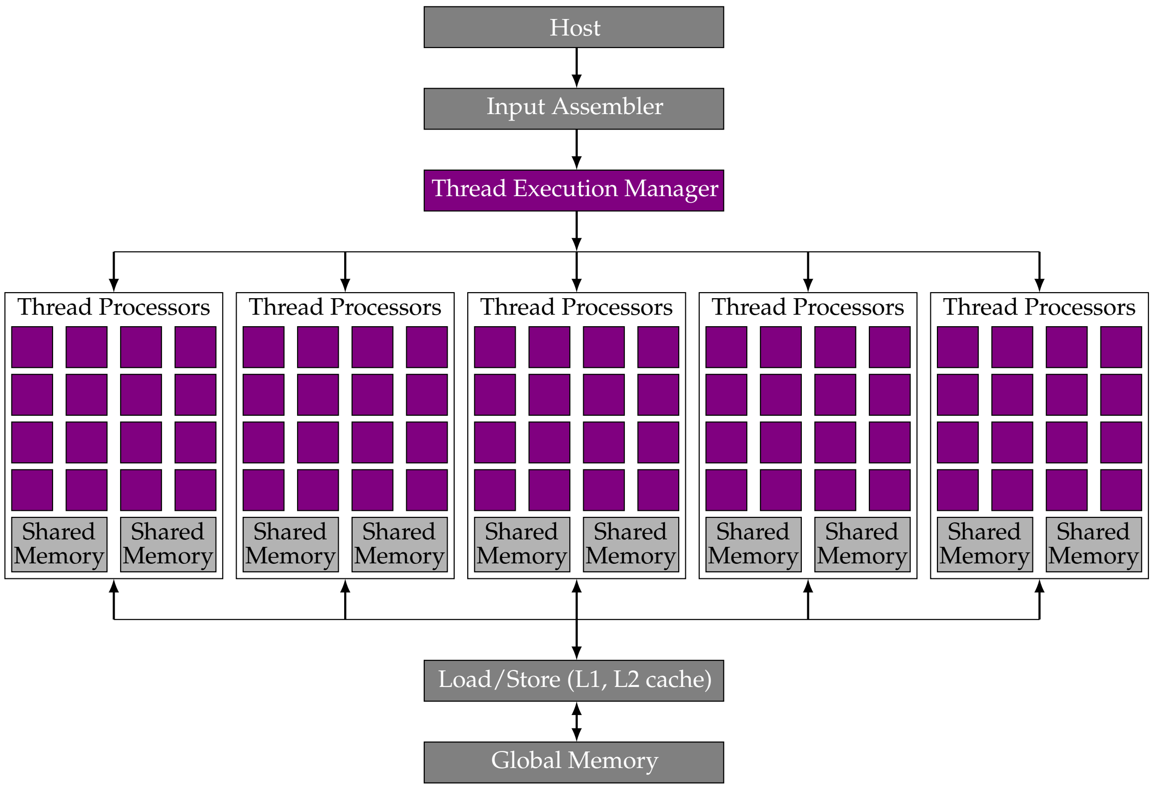
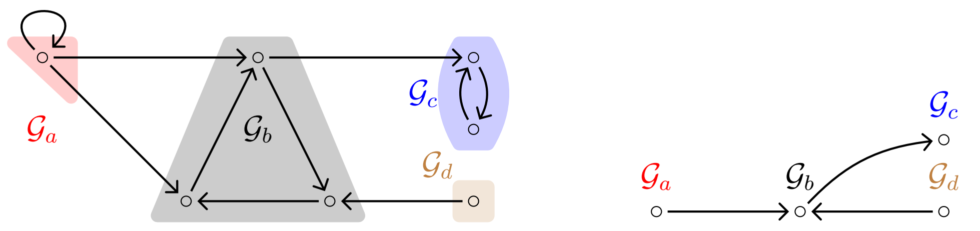


Figure 2.1: *Nvidia GeForce graphics processor architecture. The particular GeForce GTX 480 has 15 streaming multiprocessors (SM) each containing 32 streaming processors (SP) and 8 special function units (SFU). Hence, it provides 480 computing cores. Each SP can perform one single precision operation per cycle and each SFU can fulfill four SF operations per cycle, summing up the theoretical peak performance to approximately 1345 GFLOPs. The card possesses 1.5 GB of global memory. Each SM has 64 KB of on-chip memory that can be configured as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache and 768 KB of L2 cache.*

far as the modification of C is concerned it is to be seen as both an extension and restriction. The language is extended with the option to specify what parts of code are going to be executed on the GPU (called *kernel* functions) and for specification of the thread hierarchy to be applied. The restriction lies mainly in the kernels, where only declaration of variables; arithmetical, logical and bitwise operations and the branching and iteration operations can be applied.

2.2 Graphs and Selected Graph Algorithms

A directed graph G is a pair (V, E), where V is a set of vertices, and $E \subseteq V \times V$ is a set of directed edges. If $(u, v) \in E$, then v is called immediate successor of u, and u is called immediate predecessor of v. The *in-degree* and *out-degree* of a vertex v is the number of immediate predecessors and successors of v, respectively. Given a graph G, we use n and m to denote the number of vertices and edges in G, respectively. $G^T = (V, E^T)$, the *transposed graph* of $G = (V, E)$, is the graph G with all edges reversed, i.e., $E^T = \{(u, v) \mid (v, u) \in E\}$. A subgraph of a directed graph $G = (V, E)$ given by a set of vertices $V' \subseteq V$ is a directed graph $G' = (V', E \cap (V' \times V'))$.



(a) *Graph G and its strongly connected components.* **(b)** *The component graph of G.*

Figure 2.2: *Example of strongly connected component decomposition.*

Let $G = (V, E)$ be a directed graph. A *path* in G is a non-empty sequence of edges $\pi =< e_1, \ldots, e_n >$ such that $\forall 1 \leq i \leq n : e_i = (v_i - 1, v_i) \in E$. The length of path π is denoted as $|\pi|$ and for $\pi =< e_1, \ldots, e_n >$ equals to n. A path for which $v_0 = v_n$ is called a *cycle*. The set of all cycles of graph G is denoted with Z_G. We say that a vertex $t \in V$ *is reachable* from a vertex $s \in V$ if there is a path from s to t or $s = t$. A graph is *rooted* if there is an vertex $s_0 \in V$ such that all vertices in V are reachable from s_0.

For $v \in W \subseteq V$, the *forward closure* of v in W is the set of reachable states from v in the subgraph of G given by W. If W is not specified, $W = V$. The forward closure of $S \subseteq W$ in W is the union of forward closures in W over all vertices from S. Finally, the *backward closure* is defined as the forward closure in the graph G^T.

2.2.1 Strongly Connected Component Decomposition

A set of vertices $C \subseteq V$ is *strongly connected*, if for any two vertices $u, v \in C$, we have that v is reachable from u. A *strongly connected component* (SCC) is a *maximal* strongly connected set $C \subseteq V$, i.e. such that no C' with $C \subsetneq C' \subseteq V$ is strongly connected. A maximal strongly connected component C is *trivial* if C is made of a single vertex c and $(c, c) \notin E$, and is *non-trivial* otherwise. Furthermore, C is called *leading* or *terminal* if $(V \times C) \cap E = \emptyset$ or $(C \times V) \cap E = \emptyset$, respectively. We say that a subgraph $G' = (V', E')$ of G respects strongly connected components of G (is *SCC-closed*) if for every strongly connected component C of G we have $C \cap V' \neq \emptyset \implies C \subseteq V'$. Graph $G_c = (V_c, E_c)$ is a *component graph* of G if V_c is the set of all strongly connected components of G and $e = (G_1, G_2)$ is in $E_c \subseteq V_c \times V_c$ if there is an edge in G between a vertex from G_1 and a vertex from G_2. In Figure 2.2 there is an example of a graph decomposed into its strongly connected components and the corresponding component graph.

To decompose a graph into SCCs means to classify vertices of the graph according to the strongly connected component they belong to. The standard sequential algorithmic solution to the problem is due to Tarjan [105] who gave an optimal $O(n + m)$ depth-first traversal procedure (further referred as TARJAN'S algorithm) to output the list of all SCCs for a given directed graph. An annotated pseudo-code for TARJAN'S algorithm is listed as Algorithm 1. A key idea of the algorithm is a detection of a *root* vertex for each SCC. The root vertex is the first vertex of the SCC that is visited during the depth-first search traversal. Once recursive calls on root's successors has finished, all vertices on the stack from root upwards form an SCC. In order to identify the root nodes the algorithm uses a depth-search *index* that numbers the nodes consecutively in the depth-first search order in which they are discovered. The algorithm for each vertex v also keeps a value *lowlink*, that is equal to the smallest index among vertices reachable from v. Therefore v is the root of an SCC if and only if $v.lowlink = v.index$.

Algorithm 1: CPU TARJAN'S Algorithm

Input : directed graph $G = (V, E)$
Output: strongly connected component decomposition of G

```
1  index ← $0$
2  stack ← emptyStack
3  foreach $v \in V$ do
4      if $v.index = \bot$ then                      // $v$ has not yet been visited
5          DECOMPOSE($v$)

procedure DECOMPOSE($v$)
6  $v.index$ ← index
7  $v.lowlink$ ← index
8  index ← index $+ 1$
9  stack.$push(v)$
10 foreach $u \in succ(v)$ do              // for each immediate successor of $v$
11     if $u.index = \bot$ then                      // $u$ has not yet been visited
12         DECOMPOSE($u$)
13         $v.lowlink \leftarrow min\{v.lowlink, u.lowlink\}$
14     else if $u \in$ stack then // $u$ is in the stack and thus in the current SCC
15         $v.lowlink \leftarrow min\{v.lowlink, u.index\}$
16 if $v.lowlink = v.index$ then                     // $v$ is a root vertex of an SCC
17     start a new SCC
18     repeat
19         $w \leftarrow S.pop()$
20         add $w$ to the current SCC
       until $w = v$
```

2.2.2 Accepting Cycle Detection

A directed graph with *accepting vertices* G_A is a triple (V, E, A), where a set $A \subseteq V$ specifies the accepting vertices. We say that a graph G_A contains *an accepting cycle* if there exists an accepting vertex $a \in A$ such that there is a path from a to a (i.e. there is a cycle containing the vertex a). *Accepting cycle detection* is a problem of deciding whether a graph G_A contains an accepting cycle reachable from a given vertex $v_0 \in V$. The standard and optimal sequential algorithm for the accepting cycle detection is based on Nested Depth-First Search [48] (further referred as the NDFS algorithm) that runs in $O(v + m)$ time. The NDFS algorithm (listed as Algorithm 2) employes two procedures based on depth-first search graph traversal. The DFSBLUE procedure which forms the main loop of the algorithm, allows for marking each newly visited state as blue. The DFSRED procedure tries to find a path back to a given accepting vertex v and marks all visited as red. It is sufficient to find a path to a vertex v' that is on DFSBLUE stack (line 14) since the path from v' to v is guaranteed. If the DFSRED procedure detects that some accepting state is reachable from itself, the algorithm reports the existence of an accepting cycle, otherwise the graph does not contain any accepting cycle.

Algorithm 2: CPU NDFS algorithm

```
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  foreach $v \in V$ do
2      $v.blue \leftarrow$ false
3      $v.red \leftarrow$ false
4  DFSBLUE($v_0$)
5  return false

procedure DFSBLUE($v$)
6  $v.blue \leftarrow$ true
7  foreach $u \in succ(v)$ do
8      if $u.blue =$ false then
9          DFSBLUE($u$)
10 if $v \in (A)$ then
11     DFSRED($v$)

procedure DFSRED($v$)
12 $v.red \leftarrow$ true
13 foreach $u \in succ(v)$ do
14     if $u$ is on DFSBLUE stack then
15         return true
16     else if $u.red =$ false then
17         DFSRED($u$)
```

2.2.3 Optimal Cycle Mean Problem

Let $G = (V, E)$ be a graph. A *weight* function is a function $w : E \to \mathbb{R}$ that assigns a real weight to every edge of G. We speak of weighted graph if G and w are given. Weight function naturally extends to paths as a sum of the weights of all the edges on the path, i.e. $w(\pi) \overset{df}{=} \sum_{i=1}^{n} w(e_i)$, where $\pi =< e_1, \ldots, e_n >$. Let π be a cycle in a graph G weighted with a weight function w. We define *cycle mean* of cycle π as $\mu(\pi) \overset{df}{=} \frac{w(\pi)}{|\pi|}$. *Minimal cycle mean* for a given graph G and weight function w is then denoted with $\mu^*(G, w)$, where $\mu^*(G, w) = min\{\mu(\pi) \mid \pi \in Z_G\}$. Henceforward, we will safely drop the graph and weight function from the notation of minimal cycle mean and will refer to minimal cycle mean simply as to *optimal cycle mean* that will be denoted by μ^*. *OCM problem* is for a given graph G and weight function w to find the minimal cycle mean. Many different approaches to compute OCM have been studied. For example Dasdan et al. [51] gave a comprehensive list of algorithms. From practical point of view the most efficient sequential algorithmic solution to OCM problem is given by Young, Tarjan and Orlin [112] (further referred as the YTO algorithm) that runs in $\mathcal{O}(nm + n^2 \log n)$ time. We describe different approaches to OCM problem in Section 7.1.

2.3 Model Checking

In the last several years computer systems have been frequently applied to control areas where any failure or mistake may have fatal consequences. Examples of such systems can be air traffic control systems, medical instruments, banking applications, and many others. In these cases the traditional techniques of verification such as *simulation* or *testing* [93] are insufficient because they do not discover all errors, and thus, they can not guarantee that the system is correct. Therefore, the techniques of *formal verification*, i.e. formal methods proving or disproving the correctness of the system, are widely studied.

Model checking [6] is a wide-spread technique for automated formal verification of software and hardware systems. For a given formal description of a system represented as a finite-state model and inspected system property, the goal of the model checking procedure is to decide whether the model satisfies the property or not. There exist several approaches to model checking. They differ in specification of the system and the system property. In this thesis, we focus on two specific approaches, namely LTL model checking [109] and model checking of the alternation-free μ-calculus [76] using boolean equation systems [86].

2.3.1 LTL Model Checking

For LTL model checking purposes, the system to be analyzed has to be described in some modeling language, `ProMeLa` [67] for example, and the property to be checked has to be given as a formula of Linear Temporal Logic (LTL) [6]. To answer the LTL model checking question, tools, such as `SPIN` [67], `DiVinE` [16], or LTSmin [79], employ automata-theoretic approach to reduce the model checking problem to the problem of non-emptiness of Büchi automata [108]. In particular, the model of a system S is viewed as a finite automaton A_S describing all possible behaviors of the system. The property to be checked (LTL formula φ) is negated and translated into a Büchi automaton $A_{\neg\varphi}$ describing all behaviors violating φ. In order to check whether the system violates φ, a synchronous product $A_S \times A_{\neg\varphi}$ of A_S and $A_{\neg\varphi}$ is constructed describing those behaviors of the system that violate 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. The size of the product automaton is linear to the size of the system and exponential to the size of the formula [108]. 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. The LTL model checking problem is thus reduced to the problem of deciding existence of a reachable accepting cycle in the product automaton graph.

2.3.2 Model Checking Using Boolean Equation Systems

A Boolean Equation Systems (BES) is a finite sequence of the least and the greatest fixpoint equations, where each right-hand side of an equation is a proposition formula. Each equation has the form $\sigma X = \varphi$, where $\sigma \in \{\mu, \nu\}$ is the least or the greatest fixpoint operator, *left hand side* (LHS) X is a propositional variable and *right hand side* (RHS) φ is a proposition formula of the from $\varphi ::= \top \mid \bot \mid X \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$.

In the case of model checking of alternation-free μ-calculus (the inspected temporal property is expressed by a formula in alternation-free μ-calculus [76]), BESs are the result of the interpretation of a μ-formula over a *labeled transition system* (LTS) (see Figure 2.3 for the

Property - Deadlock freedom: $\nu X.([-]X \wedge \langle - \rangle \texttt{true})$

LTS_{unsat}

$$\begin{tikzcd} X_1 \arrow[r, bend right] & X_2 \arrow[l, bend right] \arrow[r] & X_3 \end{tikzcd}$$

LTS_{sat}

$$\begin{tikzcd} X_1 \arrow[r, bend right] & X_2 \arrow[l, bend right] \arrow[r] & X_3 \arrow[ll, bend left] \end{tikzcd}$$

Resulting BES

$$\begin{aligned} \nu X_1 &= X_2 \wedge \top &&= X_2 \\ \nu X_2 &= X_1 \wedge X_3 \wedge \top &&= X_2 \wedge X_3 \\ \nu X_3 &= \top \wedge \bot &&= \bot \end{aligned}$$

Resulting BES

$$\begin{aligned} \nu X_1 &= X_2 \wedge \top &&= X_2 \\ \nu X_2 &= X_1 \wedge X_3 \wedge \top &&= X_2 \wedge X_3 \\ \nu X_3 &= X_1 \wedge \top &&= X_1 \end{aligned}$$

Figure 2.3: *Interpretation of μ-formula over LTSs.*

illustration). As the formula has to be verified in every state of the LTS the resulting BES is of size $|LTS|$ x $|\varphi|^k$, i.e., the size of the BES is proportional to the size of the LTS and exponential in the complexity of the μ-formula, where k is the number of nested fixpoint operators [86]. Each fixpoint operator of the formula is represented by a so-called *block* in the resulting BES, containing the set of equations connected to this operator.

While equations may be reordered arbitrarily within a block, this is not the case for the ordering in which the blocks are solved as it might lead to the computation of a wrong fixpoint. The order in which blocks have to be processed is defined by their nesting within the μ-formula.

The solution of BESs can be computed in various ways, using methods such as *Gaussian Elimination* [86], *chasing ones* [3], or simply a fixpoint iteration – see [75] for a comprehensive summary on the topic.

2.4 Performance Verification Using Optimal Cycle Mean

High quality implementation of complex computer systems, e.g. complex embedded systems, is a major challenge today and the computer industry struggles with how to efficiently engineer these systems. Implementations of these systems raise complex parallelism and scheduling issues, which are in practice solved by hand or, at best, by using emerging tools that address only a limited set of applications with favorable properties, such as static nests of loops. One way to tackle this challenge is to use model-driven engineering.

Model-driven engineering is a very active academic domain, driving many studies and prototype tools. Model-driven performance analysis introduces performance analysis in the early design phases leading to design of more reliable and optimal system.

An inspection of graph cycles is one of the possible means to deal with performance prediction. For example assume that the actions of the system are modeled by transitions labeled with resource consumption. Then by finding the *maximal cycle mean* of a graph representing the system it is possible to approximate the worst sustainable load—the amount of resources consumed—under which the system will operate.

Analogously, the inspection of properties of critical cycles, and especially the computation of optimal cycle mean (OCM), allows to analyze performance of a large number of systems. It has been shown that the system to be analyzed can be modeled as a Petri net [100], Process graph [88] or e.g. a Data flow graph [71]. When an appropriate modeling formalism for the given real-world system is used, the enumeration of a specific cycle property facilitates performance evaluation of asynchronous systems [37], delay intensive and latency-insensitive systems [85]; rate analysis and scheduling of embedded real-time systems [88]; time-separation analysis of concurrent systems [38] and many others.

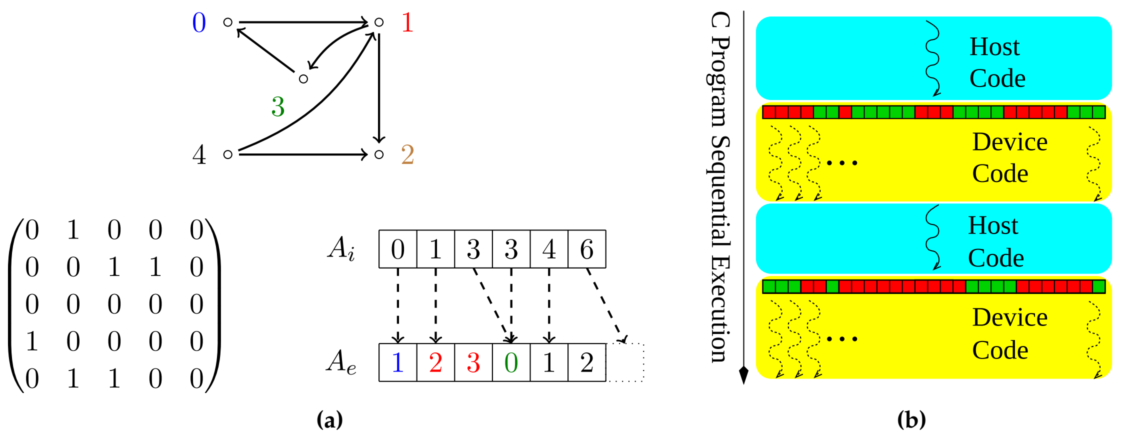
Chapter 3

Data-Parallel Graph Algorithms

This chapter provides basic ideas behind designing data-parallel graph algorithms. In particular, we present an efficient graph representation, describe a data-parallel algorithm for graph traversal that introduces the basic concept that all the other presented data-parallel algorithms build on and discuss limitations of the data-parallel algorithm. Finally, we introduce a simple model of data-parallel graph algorithms that helps us to evaluate how efficiently can modern GPUs handle suggested way of data-parallel processing.

3.1 Graph Representation

Data structures used for GPU 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 the representation of the graph 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 GPU accelerated computation.



0
1
2
3
4
$\begin{pmatrix} 0&1&0&0&0\\ 0&0&1&1&0\\ 0&0&0&0&0\\ 1&0&0&0&0\\ 0&1&1&0&0 \end{pmatrix}$
A_i
0 1 3 3 4 6
A_e
1 2 3 0 1 2
Host Code
Device Code
Host Code
Device Code
...
C Program Sequential Execution
(a)
(b)

Figure 3.1: *a) 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 $|E|$. b) Sequential heterogeneous computation work-flow with CUDA.*

To realize efficiently a CUDA-aware graph algorithm the graph needs to be represented in a compact, preferably vector-like, fashion [82]. *Compressed sparse row* representation of the adjacency matrix of the graph has proven to be an option for efficient CUDA graph encoding [62]. This graph encoding uses two one-dimensional arrays A_i and A_e, to encode

Algorithm 3: CPU ForwardReachability procedure

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 $R \leftarrow \emptyset$
2 while $S \neq \emptyset$ do
3     pick and remove $s$ from $S$
4     foreach $v$ such that $(s, v) \in E$ do
5         if $v \notin R$ then
6             $S \leftarrow S \cup \{v\}$
7             $R \leftarrow R \cup \{v\}$
8 return $R$
```

a directed graph as depicted in Figure 3.1a. 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. If other data associated to a vertex are needed by a CUDA kernel algorithm, then they are organized in vectors as well. Since the device memory is limited we try to store other bits of information into unused pointer bits (the values of A_i are technically pointers) reducing thus the space needed to record all the data for one vertex. The efficient representation of other data will be further described together with individual algorithms.

3.2 Parallelization of Graph Traversal

One of the most fundamental graph primitives is graph traversal procedure (further referred as the ForwardReachability procedure), see Algorithm 3 for CPU pseudo-code. Having properly encoded the graph the standard CUDA accelerated ForwardReachability procedure is given as Algorithm 4. The general work-flow of the algorithm is the combination of out-of-order CPU and data-parallel processing GPU that allows for heterogeneous computation as illustrated in Figure 3.1b, where sequential host code and parallel device code are executed in turns. The CPU runs the main loop of the algorithm and performs calls to CUDA kernels that run on the GPU (the CUDA kernel for the ForwardReachability procedure is listed as Algorithm 5). In the case of graph algorithms CUDA processing allows for very fine granularity of parallelism [62]. In particular, a dedicated thread is executed for every vertex of the graph (each item of Array A_i). This general work-flow is shared among all other graph procedures presented in this thesis.

To perform the 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 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 as long as vertex labels change. Note that in order to reduce the space

Algorithm 4: GPU FORWARDREACHABILITY procedure – host code

```
Input  : directed graph $G = (V, E, S)$
Output: set of vertices $R = \{v \in V \mid \exists s \in S : (s, v) \in E^+\}$
1  CREATEREPRESENTATION($G, A_e, A_i, Flags$)
2  fixPointFound ← false
3  COPYTOGPU(($A_e, A_i, Flags$) → ($gA_e, gA_i, gFlags$))
4  while ¬fixPointFound do
5      fixPointFound ← true
6      FORWARDREACHABILITYKERNEL($gA_e, gA_i, gFlags$, fixPointFound)
7  COPYTOCPU($gFlags \rightarrow Flags$)
8  $R \leftarrow \{v \in V \mid Flags[v].reached =$ true$\}$
9  return $R$
```

Algorithm 5: GPU FORWARDREACHABILITY kernel – 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 $\equiv v$
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  foreach index ∈ first, . . . , last do
8      targetVertex ← $gA_e$[index]
9      mySucc ← $gFlags$[targetVertex]
10     if mySucc.$reached$ then
11         continue
12     mySucc.$reached$ ← true
13     $gFlags$[targetVertex] ← mySucc
14     fixPointFound ← false
15 myVertex.$expanded$ ← true
16 $gFlags[v]$ ← myVertex
```

complexity of the FORWARDREACHABILITY procedure these two labels associated with each vertex v can be stored into two unused pointer bits of $gA_i[v]$. Afterwards all operations on $gFlags[targetVertex]$ can be replaced by bit manipulation on $gA_i[targetVertex]$ making thus Array $gFlags$ redundant.

3.2.1 Backward Graph Traversal

There are two options to perform the backward graph traversal (further referred as the BACKWARDREACHABILITY procedure). Either we can compute the representation of the transposed graph and employ the FORWARDREACHABILITY kernel, or we can devise a sep-

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

```
Input   : $gA_e, gA_i, gFlags,$ fixPointFound
tid ← $blockId.x * blockDim.x + threadId.x$        // tid $\equiv v$
myVertex ← $gFlags[v]$
if myVertex.$reached$ then
    return
first ← $gA_i[v]$
last ← $gA_i[v+1]$
foreach index $\in$ first, ..., last do
    targetVertex ← $gA_e$[index]
    mySucc ← $gFlags$[targetVertex]
    if mySucc.$reached$ then
        myVertex.$reached$ ← true
        fixPointFound ← false
        break
$gFlags[v]$ ← myVertex
```

arate kernel in which each thread checks the presence bits of immediate successors of its vertex and then if some of them are in the closure set, it sets the presence bit for its own vertex. While obviously the latter solution is more space efficient, our experiments have shown almost exclusive performance dominance of the first solution. For the difference between the approaches, see pseudo-codes as listed in Algorithms 5 and 6. Note that, in the case of the backward graph traversal the algorithm needs to keep only the label *reached*.

3.3 Limitations of Data-Parallel Graph Algorithms

A common drawback of most CUDA kernels for graph procedures is that many threads read some data from memory, but after evaluating them they do not write any data back. For example, in the case of the reachability procedures each thread accesses the vector of presence bits and if it reads zero for its corresponding vertex, it terminates without making any update to the vector. As a result, the CUDA hardware has to perform a lot of useless and expensive memory read operations. A possible solution [63] to the problem is to reorganize threads so that only those threads are deployed that actually do some update to the memory. However, this preprocessing is quite an expensive procedure and does not lead to a consistent speedup. Therefore, we have devised a different solution to the problem. We maintain an additional vector of $\lceil |V|/32 \rceil$ elements where we keep an information which warps (32 consecutive threads) will perform an update to the memory in the succeeding iteration. Namely, if all vertices processed within a single warp are not part of the closure set (all have the presence bit set to zero) no update to memory will occur due to this warp. By employing special *broadcast* operation available in CUDA we can thus replace (potentially up to) two 128-byte and one 64-byte data transactions with a single 32-byte memory read operation followed by the broadcast to all threads in the warp. According to our experiments this approach led to an observable speedup in many cases while introducing minimal slowdown in the other ones.

The very fine granularity of parallelism 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. A solution to the problem of irregular graphs have been recently proposed, discussed and evaluated in [69] where rather than threads, warps are mapped to vertices. The authors present a virtual warp-centric programming method based on warp-wise task allocation. This method enables to make the necessary trade-off between SIMD underutilization caused by the warp-wise task allocation and work load imbalance. It significantly boosts the performance on the highly irregular graphs. Since the method is more general it can be also applicable to other graph algorithms. Hence it has a potential to further improve the performance of data-parallel algorithms that are designed in this thesis.

The out-degree of vertices and its variability deeply depend on application domain. For example 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. However, even if the out-degree does 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. This is because the reachability procedure performs in linear time with respect to the diameter of the graph. The diameter tends to grow for a fixed-size graph as the average out-degree of a vertex decreases. We should emphasize that the diameter of the graph determines how many times the CUDA kernel must be called in order to achieve the fixpoint in the computation. Therefore the described parallelization performs in the worst case a quadratic amount of work (it inspects every edge or, at a minimum, every vertex during every kernel call). Note that this problem is not only relevant to the reachability procedure, but applies to all other graph procedures presented later on in this thesis.

Although the aforementioned methods (the reduction of useless memory operations and the warp-based mapping) also reduce the amount of work in every kernel call (not all edges and vertices are fully inspected), *a quadratic parallelization* can be very inefficient on graphs with high diameter as it has been recently shown in [91]. The authors have designed *a linear parallelization* of breadth-first search (BFS). They focused on fine-grained task management that achieves an asymptotically optimal $\mathcal{O}(m+n)$ work complexity. To realize this complexity each iteration should examine only the vertices that were first visited in the previous iteration (*a vertex-frontier*). For each iteration, tasks are mapped to unexplored vertices in the input queue maintaining the vertex-frontier. Their neighbors are inspected and the unvisited ones are placed into the output queue. In the proposed parallelization the neighbors are expanded in parallel and the local prefix sum [42] is used to compute enqueue offsets where each thread should start writing its output vertices. This approach can drastically improve the performance of the graph traversal for high diameter graphs. Moreover, it can also lead to an observable speedup for low diameter graphs. Therefore, this parallelization can significantly boost the performance of algorithms that heavily utilize the graph traversal procedure. However, the applicability to more complicated graph algorithm is questionable and will require further research. We should also remark that it has significantly higher space complexity compared to the quadratic parallelization which can be limiting factor for many application such as model checking.

Other important aspect of the GPU accelerated computation is a limited device-memory. The general work-flow as presented in the previous section 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 indicating 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 synchronization among those devices. For more details on multi-GPU acceleration see Section 5.2 of this paper. The multi-GPU acceleration of graph traversal based on a similar idea have been also described in [91].

3.4 Simple Model for CUDA Accelerated Graph Algorithms

To evaluate how efficiently can CUDA handle suggested way of data-parallel processing of graph algorithms we design and implement a simple CUDA application mimicking the typical behavior 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 the forward or the 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 scans 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*). Note that, the FORWARDREACHABILITY procedure (Algorithm 5) illustrates the write-many pattern while the BACKWARDREACHABILITY procedure (Algorithm 6) illustrates the read-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 optimized (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 indices 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 5.1.

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 utilize the shared memory as without quite expensive preprocessing the random structure of the graph breaks the data locality

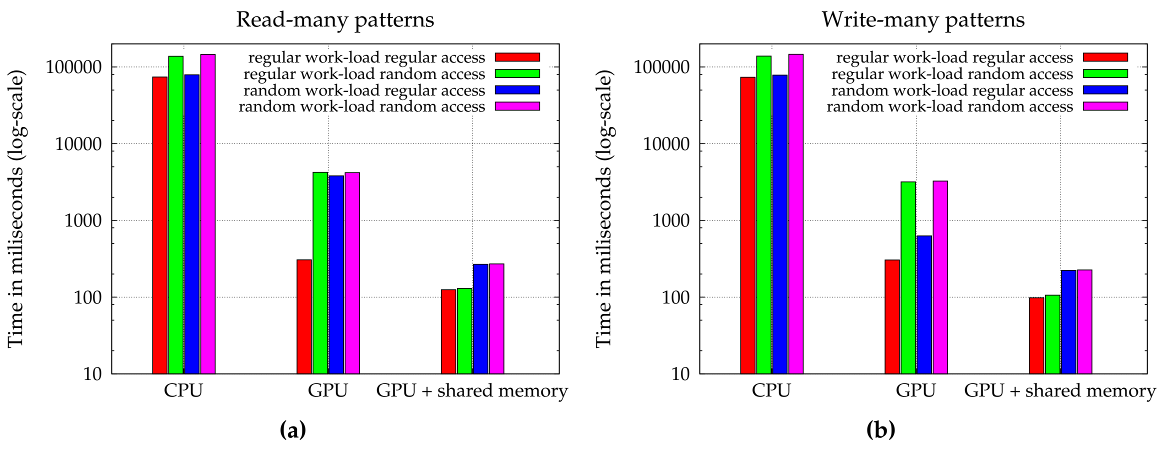


Figure 3.2: *Evaluation of different memory access patterns and work-load patterns for the model of general graph algorithms (1000 iterations, average outdegree 6, size of the graph 1 000 000) : a) Read-many patterns b) Write-many patterns.*

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, for the sake of completeness, the impact of uneven work-load to threads by setting randomly the number of loads/stores a thread executes.

The results (runtimes) of our measurements are plotted in Figure 3.2. Each column reports on time needed to complete one thousand of operations in a given memory pattern simulating a graph with a million vertices and average out-degree six. As for the CPU computation, we can observe that the random memory access causes two-fold 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 utilization 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 justifies our motivation to use massively parallel GPU architecture to accelerate the graph algorithms.

3.5 Conclusion

In this chapter, we introduced the general work-flow of data-parallel graph algorithms. This work-flow is shared among all other graph procedures presented in this thesis and allows to design complex data-parallel graph algorithms that enables to efficiently utilize modern

SIMD architectures. In particular, we showed how to encode the graph representation in order to employ massive parallelism and we described the heterogeneous computation based on the combination of out-of-order CPU and data-parallel processing GPU. We illustrated the general work-flow on the CUDA accelerated graph traversal.

We also discussed the main limitation of data-parallel graph algorithms such as inappropriate memory access and work-load patterns caused by the structure of the given graphs, non-optimal amount of work that has to be done for high diameter graphs, and memory limitation. We also mentioned several methods that can reduce or overcome these limitations, namely the reduction of memory read operations, the warp-centric programming [69], the linear parallelization [91], and multiple GPU computation. Especially, the recently proposed warp-centric approach and the linear parallelization have great potential to further significantly accelerate the data-parallel algorithms that we design in this thesis and thus they open an interesting area of our future work.

Finally, we designed and implemented the simple model for CUDA accelerated graph algorithms that mimics the typical behavior of the data-parallel algorithms. It allows us to evaluate how efficiently can CUDA handle the suggested way of data-parallel processing of graph algorithms. This evaluation clearly justifies our motivation to use massively parallel GPU architecture to accelerate graph algorithms.

Chapter 4

Data-Parallel Algorithms for Accepting Cycle Detection

In this chapter, we design data-parallel algorithms for the accepting cycle detection. These algorithms provide the building blocks for GPU accelerated LTL model checking and thus their performance is crucial. The standard and optimal sequential solution to the accepting cycle detection is based on the NDFS algorithm that was described in Section 2.2.2. The algorithm employes the depth-first search traversal that is inherently sequential and thus unsuitable for SIMD acceleration. However, recently two versions of multi-core NDFS algorithms have been introduced [55, 77]. They have a potential to scale beyond two threads (the previous multi-core implementation of the NDFS algorithm can efficiently utilize only two threads [68]) while maintaining the original time complexity. Moreover, the experiments in [78] have shown that the combination of these algorithms can successfully compete with other parallel algorithms that are not based on the depth-first search traversal. Although a lot of research has been put into the parallelization of accepting cycle detection data-parallel algorithms allowing for efficient utilization of SIMD architectures have not been considered so far. We experimentally compare the performance of different approaches to parallelization of accepting cycle detection in the context of LTL model checking in Section 5.5.

4.1 Parallel Algorithms for Accepting Cycle Detection

There exist several parallel algorithms for accepting cycle detection that are not based on the depth-first search traversal. We focus on the MAXIMAL-ACCEPTING-PREDECESSOR (MAP) algorithm [35] and the ONE-WAY-CATCH-THEM-YOUNG (OWCTY) algorithm [39] since the previous research has shown that they have the best practical performance [12, 13, 22]. These algorithms were designed assuming parallelism provided by shared-memory multi-core or distributed-memory architectures, hence, they need to be revised to benefit from SIMD parallelism the modern GPUs offer.

4.1.1 MAXIMAL-ACCEPTING-PREDECESSOR Algorithm

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 $map : V \rightarrow V \cup \{\perp\}$ 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 the 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

Algorithm 7: CPU MAP algorithm

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

Algorithm 8: CPU COMPUTEALLMAPS procedure

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

1 foreach $u \in V$ do $map(u) \leftarrow \perp$          // also $prevMap \neq map$
2 while $map \neq prevMap$ do
3     $prevMap \leftarrow map$
4     foreach $(u, v) \in E$ do
5         $map(u) \leftarrow maxacc(u, v)$
6 return $map$
```

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 the 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 7. The proof of the correctness can be found in [35].

A key procedure of the algorithm is COMPUTEALLMAPS (called *a propagation*) that is responsible for computing the values of the function map for all the vertices reachable from the initial vertex (see the pseudo-code in Algorithm 8). Initially, the values of $map(u)$ are set to $\perp$ for all $u \in V$. These values are then repeatedly updated until a global fixpoint 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 the MAP algorithm given in Algorithm 7 as *outer* iterations, and the iterations of the while loop of the COMPUTEALLMAPS procedure given in Algorithm 8 as *inner* iterations. The time complexity of the MAP algorithm is $\mathcal{O}(|V|^2 \cdot (|V| + |E|))$ [35] 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 the COMPUTEALLMAPS procedure). The ordering of the set of vertices has a significant impact on the practical performance of the algorithm [35]. 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 [35].

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 fixpoint.

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 the COMPUTEALLMAPS procedure may be localized to those edges (u, v) for which the values of $map(v)$ and $map(u)$ computed in the previous outer iteration are the same. The values of map function from the previous outer iteration are referred to as $oldMap$ values.

4.1.2 ONE-WAY-CATCH-THEM-YOUNG Algorithm

The key idea of the OWCTY 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 within the set.

The basic scheme of the OWCTY algorithm is given in Algorithm 9. The FORWARD-REACHABILITY(S) procedure (Algorithm 3) computes the set of all vertices that are reachable from the set S. The ELIMINATION(S) procedure (Algorithm 10) 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 fixpoint 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 [39]. Moreover, we can weaken the termination condition in the following way:

Proposition 4.1.1. ELIMINATION(S) $= S$ *is a correct termination condition of Algorithm 9.*

Algorithm 9: CPU OWCTY algorithm

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 $S \leftarrow$ ForwardReachability($v_0$)
2 $old \leftarrow \emptyset$
3 while $S \neq old$ do
4     $old \leftarrow S$
5     $S \leftarrow$ ForwardReachability($\{s | s \in S \wedge \mathcal{A}(s)\}$)
6     $S \leftarrow$ Elimination($S$)
7 return $S \neq \emptyset$
```

Algorithm 10: CPU Elimination procedure

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 $R \leftarrow S$
2 $old \leftarrow \emptyset$
3 $elim \leftarrow \{e \in R \mid \nexists r \in R : (r, e) \in E\}$
4 while $old \neq elim$ do
5     $old \leftarrow elim$
6     $R \leftarrow R \setminus elim$
7     $elim \leftarrow \{e \in R \mid \nexists r \in R : (r, e) \in E\}$
8 return $R$
```

Proof. Let us assume that $S' :=$ ForwardReachability$(S \cap F) =$ 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 the OWCTY algorithm is $\mathcal{O}(|V|\cdot(|V|+|E|)$ [39] since it eliminates at least one vertex in each iteration of the main while loop on line 3 (otherwise it terminates) and both ForwardReachability and Elimination take $\mathcal{O}(|V| + |E|)$ 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 the component graph of G. The *height* of the graph G is the length of the longest path in the component graph of G (note that the component graph is acyclic). Let h be a height of the input graph, then the more precise complexity of the OWCTY algorithm is $\mathcal{O}(h\cdot(|V|+|E|))$ [39]. 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) [14].

Algorithm 11: GPU 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 $\leftarrow$ false, false, false
3  COPYTOGPU$((A_e, A_i, Maps) \rightarrow (gA_e, gA_i, gMaps))$
4  while unmarked $\wedge$ $\neg$accCycleFound do
5      while $\neg$fixPointFound $\wedge$ $\neg$accCycleFound do
6          fixPointFound $\leftarrow$ true
7          MAPKERNEL$(gA_e, gA_i, gMaps,$ accCycleFound, fixPointFound$)$
8      unmarked $\leftarrow$ false
9      UNMARKACCKERNEL$(gMaps,$ unmarked$)$
10 return accCycleFound
```

4.2 Data-Parallel Version of MAXIMAL-ACCEPTING-PREDECESSOR Algorithm

As discussed in the previous sections, to achieve good acceleration on massively parallel architectures, the input data must be represented in an appropriate, preferably vector-like, fashion. This is easily achievable in the MAP algorithm as the additional data required by the algorithm are associated with vertices, hence, they can be stored in vectors. In particular, the MAP algorithm maintains array $Maps$ that keeps the values of *map*, *oldMap* and $\mathcal{A}$ predicate for each vertex.

The main computation demanding part of the MAP algorithm is the COMPUTEALLMAPS procedure, see Subsection 2.3.1. This procedure can be parallelized in the way similar to the previous case of the forward reachability procedure. Algorithm 11 lists the host code of the MAP algorithm. The inner and outer while loops listed in the pseudo-code correspond with the inner and outer iterations as introduced in Subsection 2.3.1.

There are three kernel procedures called from the host code. The most important one, MAPKERNEL, is listed as Algorithm 12. Every call to MAPKERNEL updates the *map* values along every edge (see lines 5 and 8-9). 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 the accepting predicate for accepting states proven to be outside an accepting cycle. A pseudo-code of UNMARKACCKERNEL is listed as Algorithm 13. The values of *map* and *oldMap* are handled so that the *oldMap* values partition the graph with respect to the previous outer iteration.

The performance of the MAP algorithm deeply depends on the vertex ordering [36]. With inappropriate vertex ordering the execution of the 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. We should point out that in the context of model checking we are actually computing minimal accepting successors. Considering successors allows us to store only the forward edges in the graph representation and preferring smaller values inverts the BFS ordering enforced by generation (actual BFS ordering provided significantly worse results). This observation can be explained by existence of paths

Algorithm 12: GPU MapKernel – device code (run in parallel $\forall v \in V$)

```
Input   : $gA_e, gA_i, gMaps$, accCycleFound, fixPointFound
1  myVertex, candidate ← $gMaps[v], \bot$
2  foreach $u \in succ(v)$ do          // $succ(v) = \{gA_e[gA_i[v]], \ldots, gA_e[gA_i[v+1]]\}$
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
```

Algorithm 13: GPU UnmarkAccKernel – device code (run in parallel $\forall v \in V$)

```
Input   : $gMaps$, unmarked
1  myVertex, change ← $gMaps[v]$, false
2  if $\mathcal{A}(v) \wedge$ myVertex $< v$ then
3      $\mathcal{A}(v)$, unmarked, change ← false, true, true
4  if myVertex.$map \neq$ myVertex.$oldMap$ then
5      myVertex.$oldMap$, myVertex.$map$, change ← myVertex.$map$, $\bot$, true
6  if change then
7      $gMaps[v]$ ← myVertex
```

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 proved more resistant to any improper ordering in the representation.

4.3 Data-Parallel Version of One-Way-Catch-Them-Young Algorithm

The non-CUDA version of OWCTY algorithm comprises of alternating execution of forward reachability and *backward elimination* (Algorithm 9). 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 the OWCTY algorithm.

Implementation of reachability was given sufficient space in Section 3.2. Therefore, 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.

Algorithm 14: GPU 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 $gFlags[v].elim$ = false then
8      foreach $u \in succ(v)$ do                // $succ(v) = \{gA_e[gA_i[v]], \ldots, gA_e[gA_i[v+1]]\}$
9          if $gFlags[u].elim$ = false $\wedge$ $gFlags[u].elimPrep$ = true then
10             $gFlags[u].elimPrep$ ← false

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

Our backward elimination hence needed to consist of two steps (see Algorithm 14). 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.

Having described the building blocks, we may proceed to the actual OWCTY algorithm implementation (see Algorithm 15). The basic layout is similar to the original implementation. The CUDA kernel VISACCEPTINGKERNEL sets all accepting vertices (that have not yet been eliminated) to visited. Having considered the Proposition 4.1.1, we do not need to test if FORWARDREACHABILITY visited all vertices. Only its effect, the elimination of non-visited vertices is necessary (via the kernel TESTSETKERNEL). Afterwards this kernel also sets all vertices to unvisited to prepare the next iteration of the main loop. The elimination proceeds as described above. Furthermore, if no vertex is eliminated (line 6 of Algorithm 14) 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 the 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 Subsection 3.2.1, 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.

Algorithm 15: GPU 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(($A_e, A_i, Flags$) $\rightarrow$ ($gA_e, gA_i, gFlags$))
3  VisAcceptingKernel($gFlags$)
4  fixPointNotFound, accCycleFound $\leftarrow$ true, false
5  while fixPointNotFound do
6      reachabilityFixPointFound $\leftarrow$ false
7      while $\neg$reachabilityFixPointFound do
8          reachabilityFixPointFound $\leftarrow$ true
9          ForwardReachabilityKernel($gA_e$, $gA_i$, $gFlags$, reachabilityFixPointFound)
10     TestSetKernel($gFlags$)
11     fixPointNotFound $\leftarrow$ false
12     Elimination($gA_e, gA_i, gFlags$, accCycleFound, fixPointNotFound)
13     VisAcceptingKernel($gFlags$)
14 return accCycleFound
kernel VisAcceptingKernel($gFlags$)
15 if $gFlags[v].acc = \texttt{true} \wedge gFlags[v].elim = \texttt{false}$ then
16     $gFlags[v].visited \leftarrow \texttt{true}$
kernel TestSetKernel($gFlags$)
17 if $gFlags[v].visited = \texttt{false}$ then
18     $gFlags[v].elim \leftarrow \texttt{true}$
19 $gFlags[v].visited \leftarrow \texttt{false}$
```

In [26] we show that the reversed variant of the 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 the CUDA MAP and CUDA reversed OWCTY algorithms (further referred only as CUDA OWCTY) and the optimal sequential NDFS algorithm in the context of LTL model checking in Section 5.5.

4.4 Conclusion

In this chapter, we introduced two selected parallel algorithms for accepting cycle detection, namely the MAP and OWCTY algorithms that have been shown to have the best practical performance. Since these algorithms were designed assuming parallelism provided by shared-memory multi-core or distributed-memory architectures, we reformulated the algorithms to benefit from SIMD architecture parallelism the modern GPUs offer. In particular, we showed how to build the efficient data-parallel versions of these algorithms from the basic graph primitives. This allows to keep the provably correct layout of the existing algorithms but significantly accelerates their computation on many-cores GPUs.

The experimental evaluation of the particular algorithms for accepting cycle detection in the context of LTL model checking is presented in Section 5.5.

In our future work, we plan to employ the recently proposed methods such as the warp-centric programming [69] and the linear parallelization [91] (see Chapter 3 for more details) in order to further accelerate the accepting cycle detection on the many-core GPUs.

Chapter 5

Fast LTL Model Checking Algorithms for Many-Core GPUs

In this chapter we show how the proposed data-parallel algorithms for accepting cycle detection can be employed in order to accelerate the LTL model checking process on modern many-core GPU platforms.

The model checking techniques generally suffer 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 [98, 34] and symbolic representation [90] 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 analyzing 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 LTL model checker SPIN [67] has been introduced [68], the LTSmin tool [29] now uses a multi-core version of the nested depth-first search [77, 78] and there has also been a dedicated multi-core branch [12, 13] of parallel model checker DiVinE [16]. For share memory architectures, linear speedup of model checking is the primary goal [79]. Besides multi-core and multi-CPU systems, many-core hardware accelerators have received a lot of attention as well. Recent model checking research results related to the use of CUDA technology describe CUDA accelerated state space generation [52, 53, 54] or model checking of probabilistic systems [32].

As we describe in Section 2.3.1, the LTL model checking problem can be reduced to the problem of deciding existence of a reachable accepting cycle in the product automaton graph. As far as GPU accelerated accepting cycle detection is concerned, Chapter 4 could serve as a complete description. However, in LTL 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 tackle, e.g. transformation from implicit graph representation to a form suitable for GPU computation and overcoming of GPU memory limitation. The solution of problems related to and further optimization of GPU accelerated model checking will form the content of this chapter.

5.1 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 11 and line 1 of Algorithm 15). 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.

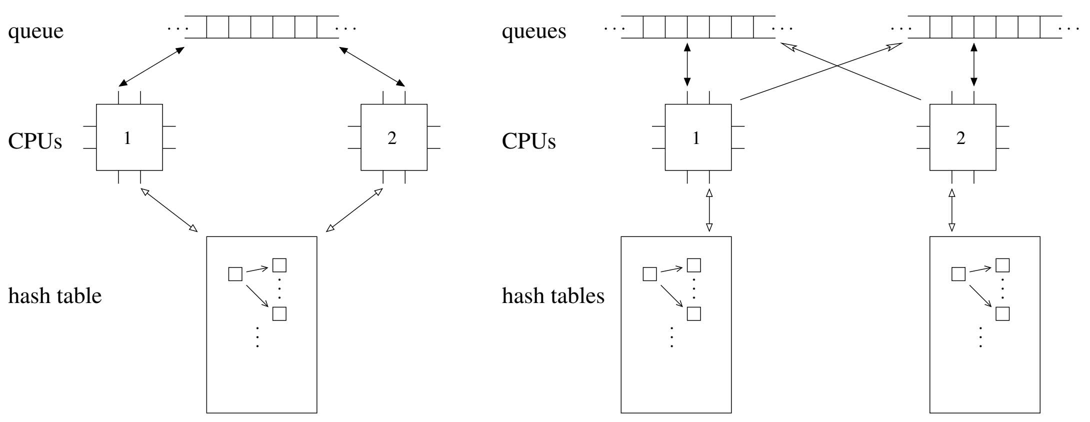
The main bottleneck of the whole CUDA accelerated approach to LTL model checking is the costly procedure of the construction of the adjacency list representation [21]. In order to alleviate the burden of the transformation of the implicit definition of the graph into the adjacency list representation, 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 [12, 13].

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 [79]. 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 minimize 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 5.1.

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. 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 5.3) 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.



(a) *Shared hash table approach.* **(b)** *Distributed hash table approach.*

Figure 5.1: *Comparison of the work-flow of the two state-of-the-art approaches 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.*

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 thread 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 [80, 15]. 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 they 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 5.5).

Efficient utilization 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 [54]. They have achieved significant speedup of transition enablement checking and successors generation. However, duplicate detection, the most crucial part of the state space generation, has not been parallelized on GPU yet. Thus the overall speedup of the whole state space generation was moderate.

5.2 Multi GPU Model Checking

The size of the compact representation can be estimated to approximately $8|V| + 4|E|$ bytes for the OWCTY algorithm and $12|V| + 4|E|$ bytes for the MAP algorithm, 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 GPU 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 GPU 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 GPU 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 (see Section 4.2).

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 GPU device keeps one part of the adjacency list representation and the complete vector of values associated with 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. It also introduces distribution of the vector of values. In particular, every GPU 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 list representation.

5.2.1 Synchronization

Here we explain how to utilize multiple GPU devices to accelerate the MAP algorithm and then we discuss how to generalize this approach to other graph algorithms such as OWCTY.

Algorithm 16: Multi GPU MAP algorithm (inner fixpoint) – host code

```
Input : $localG, localMaps$, accCycleFound, fixPointFound
1  globalChange $\leftarrow$ true
2  while globalChange $\wedge$ $\neg$accCycleFound do
3      $foreignMaps$, localChange $\leftarrow$ DOWNLOAD(), false
4      while $\neg$fixPointFound $\wedge$ $\neg$accCycleFound do
5          fixPointFound $\leftarrow$ true
6          MAPKERNEL($localG, localMaps, foreignMaps$, accCycleFound, fixPointFound)
7          localChange $\leftarrow$ localChange $\vee$ $\neg$fixPointFound
8      UPLOAD($localMaps$)
9      VOTEIN(localChange)
10     RENDEZVOUS()
11     globalChange $\leftarrow$ VOTEOUT()
```

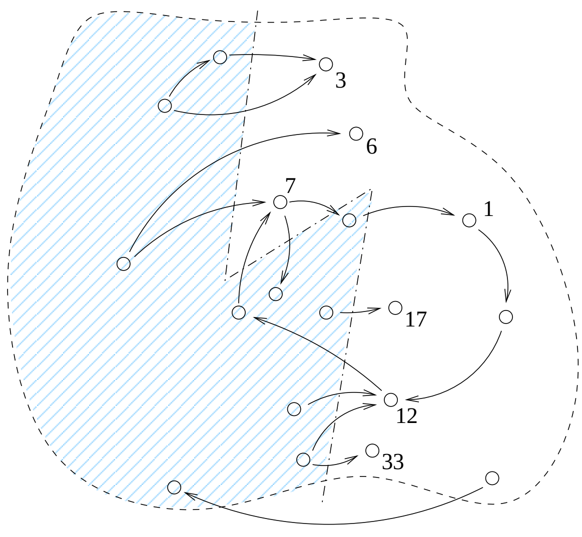
Having the edges of the graph distributed among multiple GPU devices the local computation of the algorithm comes across the necessity to exchange the map values of foreign vertices, the so-called *synchronization*. In the distributed MAP algorithm (Algorithm 16) every single GPU device computes the local fixpoint as in the single CUDA computation, but then it synchronizes on the values of foreign vertices with all the other CUDA devices. The local fixpoint is thus achieved using solely the mutable map values of local vertices and the constant map values of foreign vertices received from the synchronization. These two subsequent steps repeat until a global fixpoint is found. Since the map values of foreign vertices are constant throughout the local fixpoint search, the UPLOAD is conditioned by a change detected after considering these values for the first time within the inner cycle. If the global fixpoint 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 the forward reachability and the backward elimination, we have to perform the synchronization and the vote for termination once the local fixpoints of each phase are found. Apart from small adjustments of the termination condition this was the only change necessary for the OWCTY algorithm to be extended to work on multiple GPU devices. We direct the reader to Section 5.5 to see the comparison of both algorithms.

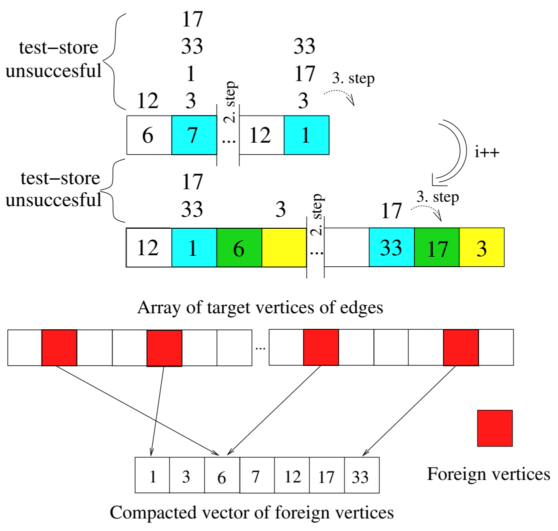
5.2.2 Preparing Foreign Vertices Vectors

The synchronization procedure requires to exchange only the values associated with foreign vertices, however, the communication between GPU devices is realized 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 GPU 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 GPU device. We duplicate values



(a)



(b)

Figure 5.2: *Example of the compacted vector construction for foreign vertices. a) Graph partitioning. b) The compact vector allocating procedure: it fails for $i = 2$ and is successful for $i = 3$; mapping foreign vertices on their counterparts in the compacted vector.*

stored for the foreign vertices in a separate compacted vector, i.e. a vector containing only the values for foreign vertices for a particular GPU 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.

In order to sum the number of unique foreign vertices effectively we could mark the first occurrence of each foreign vertex using the reduce [64] and `atomicCAS` operations. This would, however, require a copy of the whole vector of vertices which we cannot afford due to the space limitation. To get a safe but close overapproximation of the number of foreign vertices we therefore apply the heuristics as illustrated in Figure 5.2.

Let i be a small integer number. We allocate a vector of size 2^i and process it by CUDA kernels performing iteratively the following operations. First, we try to store every foreign vertex v on the position $v\&(2^{i-1} - 1)$, if there are conflicts for multiple vertices on some positions, we keep only the first vertex stored on the position and try to store those other conflicting vertices v on the position $2^{i-1} + v\&(2^{i-1} - 1)$, if there are still conflicts, we sequentially look for empty position from $2^{i-1} + v\&(2^{i-1} - 1) + 1$ to $2^{i-1} + v\&(2^{i-1} - 1) + i$ to store every conflicting vertex v. If the call of CUDA kernel is unable to resolve all the conflicts within $\mathcal{O}(i)$ steps, we increment i and repeat the procedure.

After that we have a compacted vector of the size 2^i containing all foreign vertices exactly once. We finish the preparation of the vector by sorting the vector and cutting off the prefix of zeroes. 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.

5.3 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 behavior as well. 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 the OWCTY algorithm to be executed concurrently in the background of the state space generation. This work-flow, though requiring two GPU 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).

5.4 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 behavior 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 5.1). 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 both the MAP and the OWCTY algorithm identify the states in the compact representation that form the found cycle. The MAP algorithm provides an accepting state on the cycle and marks the part of state space where the cycle is located. Therefore, the cycle can be easily identify using the 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 in the case of the MAP algorithm the first phase of the counterexample generation has linear time complexity.

The OWCTY algorithm 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 [39].

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 to wait then it waits until reaches the critical section and it eventually reaches the critical section.
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 5.1: *Models used in the experiments with properties they are expected to have.*

Since for both algorithms the first phases includes only the 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 parallelized 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 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.

5.5 Experimental Evaluation

We have implemented the designed algorithms and methods as a part of the DiVinE-CUDA tool [20]. 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 5.1. 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 GPU devices (described in Section 5.2.2) allows verification of these models.

Model	vertices		edges		RAM cons.	accepting cycle
	explored	stored	explored	stored		
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 5.2: *Spacial complexity of the models and existence of accepting cycles.*

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 [21, 26, 8] 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 runtimes of the whole model checking procedure it was hardly noticeable.

Table 5.2 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 5.1 reduces the size of the graph representation. The overall CPU memory consumption (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 5.3 for more details).

5.5.1 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 [22] and LTSmin [77] (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.

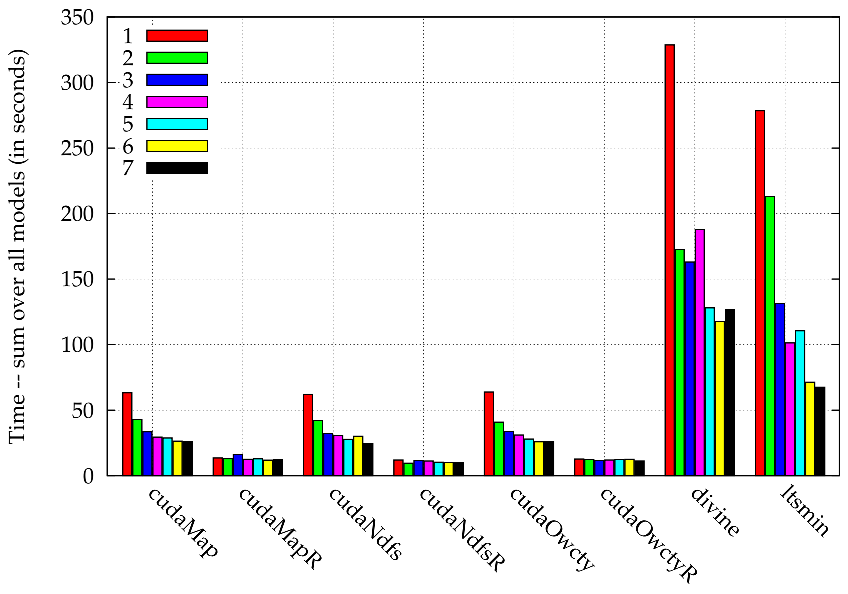


Figure 5.3: *Effectivity of state-of-the-art model checkers on incorrect models (1..7 cores)*

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 behavior. 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 runtimes of particular 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 5.3 and 5.4 we depict the overall runtime 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 5.4 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 positive effect on the runtimes it can have a negative effect on the MAP algorithm, because of its dependency on ordering of vertices. The exact runtimes of the algorithms on some selected models are summarized in Table 5.3. 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 5.5) 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 efficiency of the accepting cycle detection algorithm. Hence we

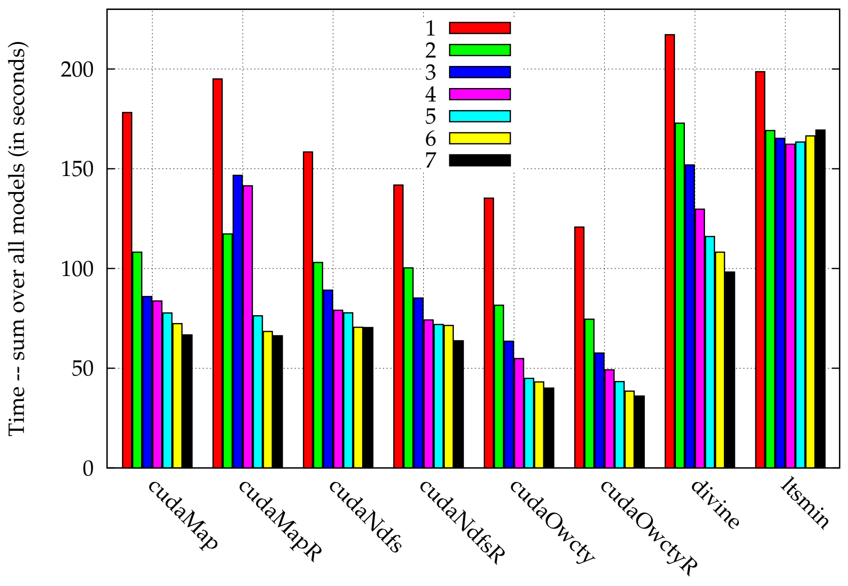


Figure 5.4: *Effectivity of state-of-the-art model checkers on correct models (1..7 cores)*

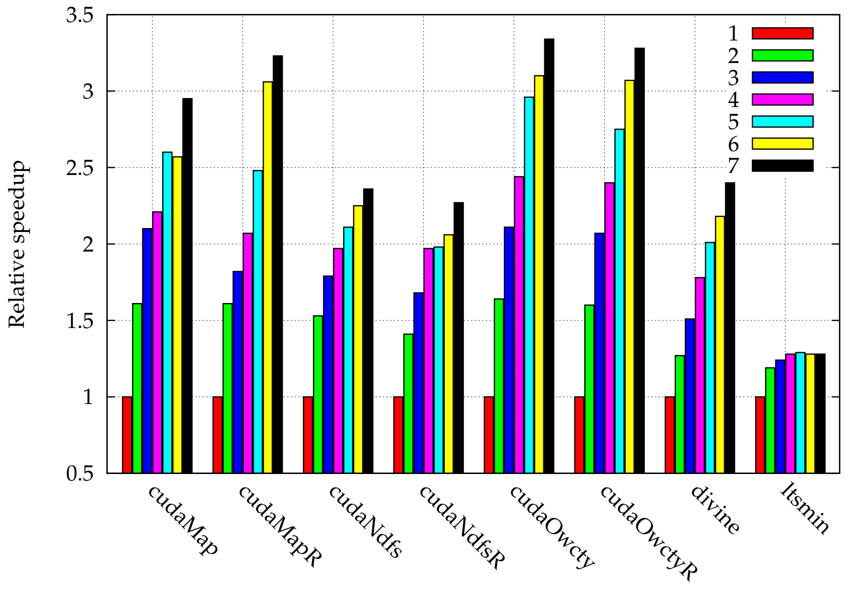


Figure 5.5: *Scalability of state-of-the-art model checkers on correct models (1..7 cores)*

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.

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 5.3: *The comparison of LTL model checking tools (runtimes in seconds).*

5.5.2 Comparison of Algorithms on Compact Representation

Table 5.3 provides details on runtimes of individual CUDA accelerated algorithm parts and gives the comparison to the 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 runtimes (tot.) of all algorithms. As for the CUDA algorithms, the total runtime also includes certain initialization overhead not reported in the table. We can observe that in the CUDA accelerated OWCTY algorithm the time for preparation of adjacency list representation significantly dominates to the whole model checking procedure.

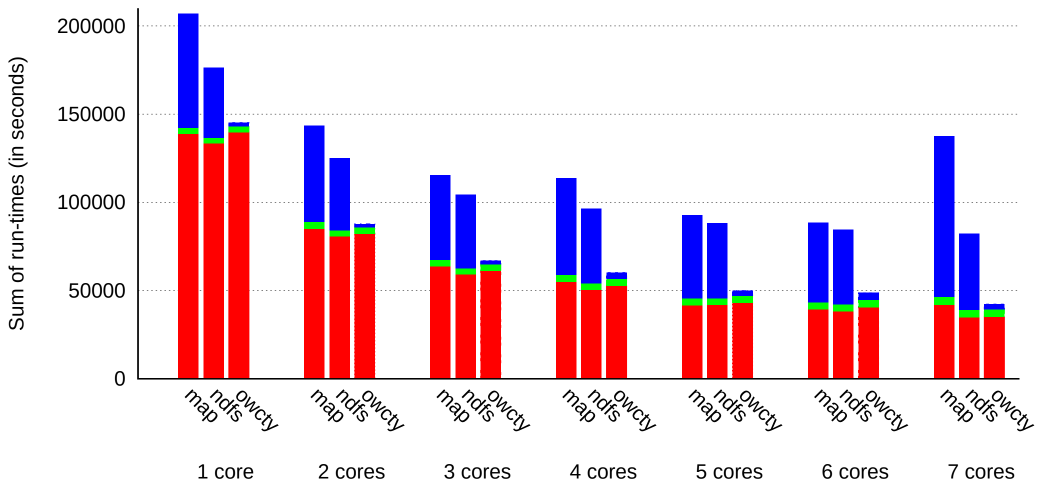


Figure 5.6: *The efficiency of combination of multi-core state space generation and many-core accepting cycle detection. Red bar is for* state space generation, *green for* compact representation preparation *and blue represents the* CUDA Computation.

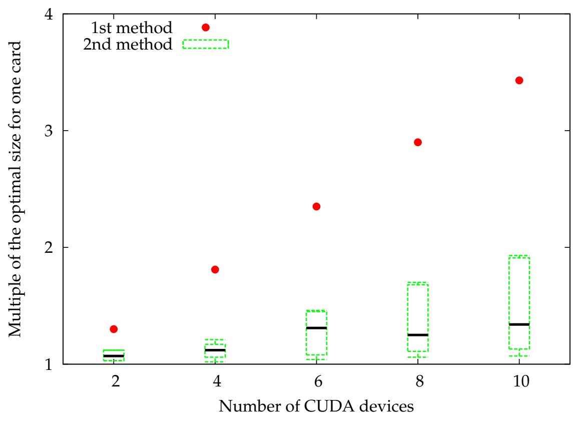
We can also see that the CUDA accelerated OWCTY algorithm significantly outperforms the original CUDA accelerated MAP algorithm on most valid model checking instances (without accepting cycle). The results for invalid instances (with accepting cycle) speak also in favor of the OWCTY algorithm, though the gain is considerably less impressive.

From Figure 5.6 we may observe how effective is the multi-core acceleration of the state space generation – proposed in Section 5.1. We have summed the respective runtimes 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 [8] 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 GPU device and thus cannot be efficiently used in the adjacency list construction.

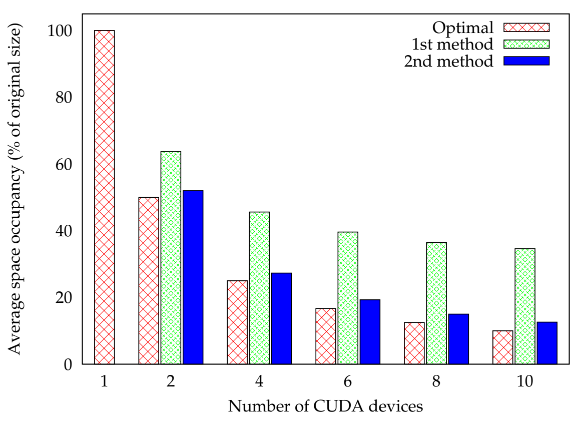
The experiments also show that the performance of the CUDA OWCTY algorithm does not depend on the ordering in adjacency list representation as much as the CUDA MAP algorithm (see [26] for more details) and therefore the CUDA OWCTY algorithm has better runtimes on almost all models also in the case when multi-core acceleration of adjacency list construction is utilized. All together it seems that when multi-core acceleration of adjacency list computation is utilized the OWCTY algorithm is clearly a winner for CUDA computation. The superiority is even more pronounced that it was in our previous papers [8, 26] which is most likely caused by the usage of shared hash table approach to state space generation which also alters the vertex ordering.

5.5.3 Experiments on Multi GPU Algorithms

Figures 5.7a and 5.7b provide detailed comparison of the relative size of adjacency list representation and the space efficiency for both described methods (see Section 5.2.2) of multi



(a)



(b)

Figure 5.7: *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.*

GPU computation. Figure 5.7a 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 GPU 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 5.2.2 for more details). This phenomenon can be moderated by allowing the preparation phase to be more space and less time efficient.

Figure 5.7b further illustrates the difference of the two proposed methods with respect to their ability to efficiently utilize space when increasing number of GPU 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 GPU devices, the two Figures 5.7a and 5.7b represent only the state space partitioning part of the model checking. As they only speak about space complexity, however, this is only a noteworthy comment.

In Figure 5.8 we only provide runtimes of algorithms containing the second method of graph partitioning as it was shown in [8] to be only negligibly slower while considerably more space efficient. The figure compares both dual CUDA algorithms and 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 initialization time of every run contains certain non-trivial overhead (approximately 5 seconds). We have observed that this overhead is caused by serialization 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.

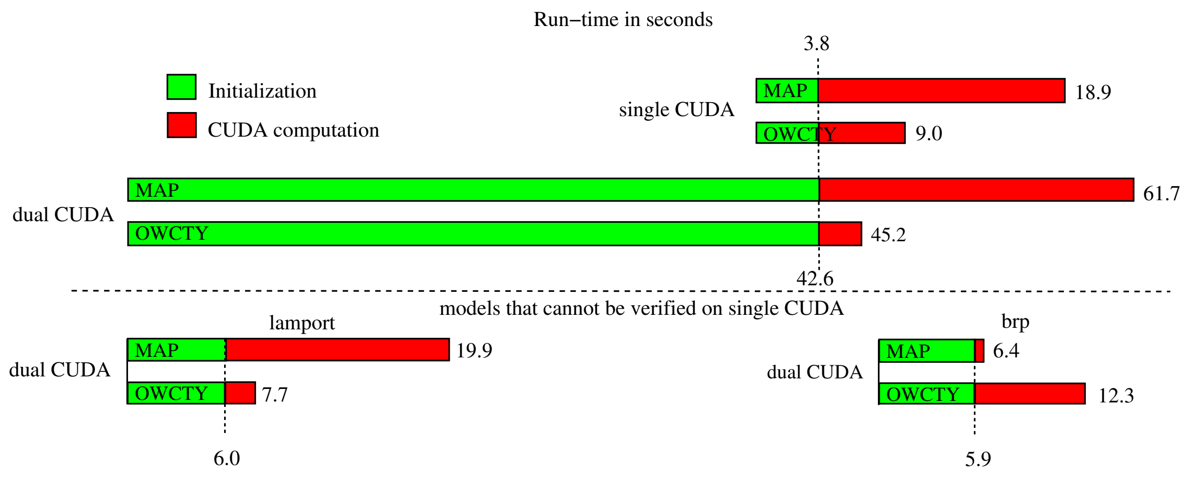


Figure 5.8: *The comparison of the single CUDA and the dual CUDA algorithms on all models (two of which cannot be verified using a single device).*

Unlike the OWCTY algorithm, whose multi-device version requires seemingly constant and small number of synchronization between devices, the MAP algorithm needs considerable more synchronizations for completion. This observation illustrates that not only is the 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 fixpoint.

5.6 Conclusions

In this chapter, we provided a summary of the latest advancements in GPU acceleration of the LTL model checking. 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 second 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 efficiency improvement of the proposed CUDA LTL model checking algorithm. We also established successful employment of multiple GPU devices to verify considerably larger instances of model checking problems while preserving significant speedup. We showed that the expensive communication among particular GPU devices and CPU imposed by individual synchronizations leads to only negligible slowdown. These new approaches can be effectively employed on modern multi-core machines equipped with multiple GPU 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 hashtables and DFS exploration together with the 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 work we would like to put significant effort in designing GPU accelerated state space generation and adjacency list computation which can lead to additional speedup of model checking procedure and which we consider to be the next challenge for the parallel model checking community.

Chapter 6

Data-Parallel Decomposition into Strongly Connected Components

In this chapter we focus on the problem of decomposing a directed graph into its strongly connected components (*SCC decomposition*). This problem has many applications leading to very large graphs and requiring high performance processing. One example is the web analysis based on web archives, such as topic tracking, time-frequency analysis of blog postings, and web community extraction. A particular application we also have in mind is automated verification of software (model checking, dataflow analysis, bad cycle detection, etc.), where SCC decomposition is typically used as a sub-procedure and its fast performance is crucial for assuring overall efficiency of verification tools. Several parallel algorithms for SCC decomposition have been proposed [58, 96, 25, 24], however data-parallel algorithms allowing for efficient utilization of SIMD architectures have not been considered so far.

Parallel SCC decomposition is a particularly tricky problem. The reason is that the optimal sequential algorithm strongly relies on the depth-first search post ordering of vertices whose computation is known to be P-complete [101] and thus, difficult to be computed in parallel. Hence, different approaches suitable for parallel processing have been considered. See e.g. [60, 47, 2] for algorithm that works in $\mathcal{O}(log^2 n)$ time, but requires $\mathcal{O}(n^{2.376})$ parallel processors, or [111] for randomized parallel algorithm for the problem.

We show how existing parallel SCC decomposition algorithms can be modified in order to be accelerated on a vector processing SIMD architecture. In particular, we decompose the algorithms into primitive data-parallel graph operations, design a new CUDA-aware procedure for pivot selection and reformulate the recursion present in the algorithms by means of iterative procedures.

We experimentally demonstrate that with a single GTX 480 GPU we were able to gain speedup ranging from 5 to 40 when compared to the optimal sequential TARJAN'S algorithm.

6.1 Parallel Algorithms for Strongly Connected Component Decomposition

First, we describe in more details the basic ideas behind the parallel SCC decomposition algorithms. These algorithms (similarly as parallel algorithm for accepting cycle detection) were also designed assuming parallelism provided by shared-memory multi-core or distributed-memory architectures, and therefore they need to be redesigned in order to efficiently utilized from SIMD architecture parallelism. We refer readers to original papers for the proofs of correctness and time complexity of the algorithms.

6.1.1 FORWARD-BACKWARD Algorithm

The FORWARD-BACKWARD (FB) algorithm [58] introduces the basic concept that all the other presented algorithms build on. The algorithm proceeds as follows. A vertex called *pivot* is selected and the strongly connected component the pivot belongs to is computed as the

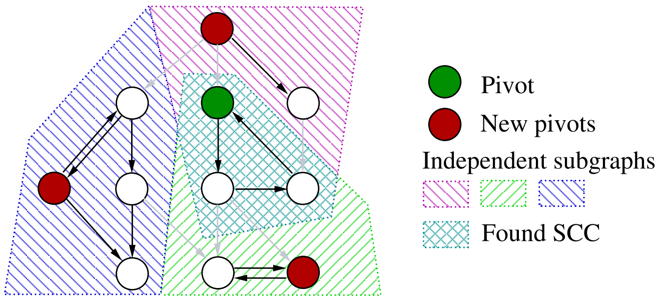


Figure 6.1: FB *decomposition of the graph into the independent subgraphs*

Algorithm 17: CPU FB algorithm

Input : directed graph $G = (V, E)$
Output: strongly connected component decomposition of G

```
1  Procedure FB($V$)
2  begin
3      pivot ← PivotSelection($V$)
4      F ← ForwardReachability(pivot, $V$)
5      B ← BackwardReachability(pivot, $V$)
6      F ∩ B is SCC
7      in parallel do
8          FB(F \ B)
9          FB(B \ F)
10         FB($V$ \ (F ∪ B))
```

intersection of the forward and backward closure of the pivot. Computation of the closures divides the graph into four subgraphs that all respect strongly connected components. These subgraphs are 1) the strongly connected component with the pivot, 2) the subgraph given by vertices in the forward closure but not in the backward closure, 3) the subgraph given by vertices in the backward closure but not in the forward closure, and 4) the subgraph given by vertices that are neither in the forward nor in the backward closure (see Figure 6.1). The subgraphs that do not contain the pivot form three independent instances of the same problem, and therefore, they are recursively processed in parallel with the same algorithm. The pseudo-code of the algorithm is listed as Algorithm 17.

The time complexity of the FB algorithm is $\mathcal{O}(n \cdot (m + n))$ since it performs $\mathcal{O}(m + n)$ work to detect a single strongly connected component. Practical performance of the algorithm may be further improved by performing elimination of leading and terminal trivial strongly connected components – so called *trimming* [89]. The TRIMMING procedure builds upon a topological sort elimination. The key idea is as follows. A vertex cannot be part of a non-trivial strongly connected component if its in-degree (out-degree) is zero. Therefore, such a vertex can be safely removed from the graph as a trivial SCC, before the pivot vertex is selected and forward and backward closures are computed. The removal of a vertex may render another vertex or vertices to have zero in-degree (out-degree). Therefore, the elimination is iteratively repeated until no more vertices with zero in-degree (out-degree) exist. Only after that, the pivot is selected and the algorithm proceeds as stated above. Note that the elimination procedure is also referred as the OWCTY elimination, see Subsection 4.1.2.

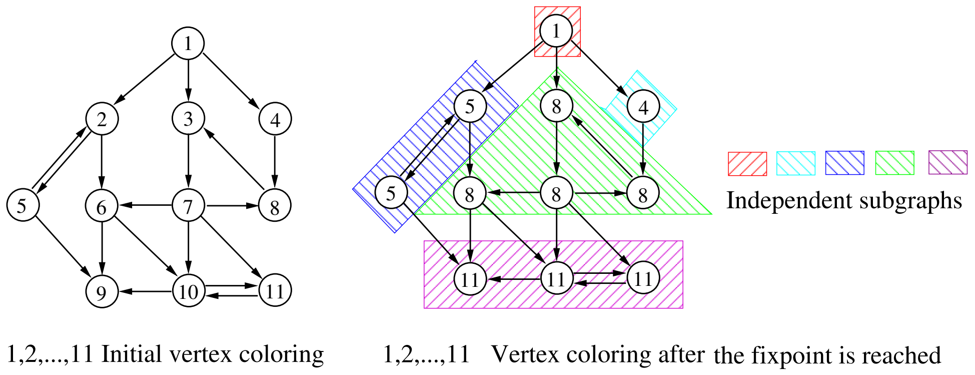

1,2,...,11 Initial vertex coloring
1,2,...,11 Vertex coloring after the fixpoint is reached
Independent subgraphs

Figure 6.2: COLORING *decomposition of the graph into the independent subgraphs*

Algorithm 18: CPU COLORING algorithm

Input : directed graph $G = (V, E)$
Output: strongly connected component decomposition of G

```
1 Procedure COLORING($V$)
2 begin
3     if $V \neq \emptyset$ then
4         maxColorList, $(V_k)_{k \in \text{maxColorList}} \leftarrow$ FORWARDREACHABILITYMAXCOLOR($V$)
5         for $k \in$ maxColorList in parallel do
6             $B_k \leftarrow$ BACKWARDREACHABILITY($k$, $V_k$)
7             $B_k$ is SCC
8             COLORING($V_k \setminus B_k$)
```

6.1.2 COLORING Algorithm

The main limitation of the FB algorithm is that it performs $\mathcal{O}(m+n)$ work to detect a single SCC. This may be rather expensive strategy if the graph contains many small but non-trivial components. Completely opposite approach is taken in the COLORING algorithm [96]. It is capable of detecting many strongly connected components in a single recursion step, however, for the price of $\mathcal{O}(n\cdot(m+n))$ procedure. Therefore, the time complexity of the algorithm is $\mathcal{O}((l+1)\cdot n\cdot(m+n))$ where l is the longest path in the component graph of G. The idea of the algorithm relies on the propagation of unique and totally ordered identifiers (colors) associated with vertices. Initially, each vertex keeps its own color. The colors are then iteratively propagated along edges of the graph (line 4) so that each vertex keeps only the maximum color among the initial color and colors that have been propagated into it (maximal preceding color). After a fixpoint is reached (no color update is possible), the colors associated with vertices partition the graph into multiple component respecting subgraphs (see Figure 6.2). All vertices of a subgraph are reachable from the vertex whose color is associated with vertices in the subgraph, and this vertex lies in the leading strongly connected component of the subgraph. Therefore, the related component can be identified by computing a backward closure of the vertex restricted to the subgraph. This is what the algorithm does for all the subgraphs in parallel prior the recursion step (see Algorithm 18). Note that the propagation procedure is rather expensive if there are multiple large components in the graph [23].

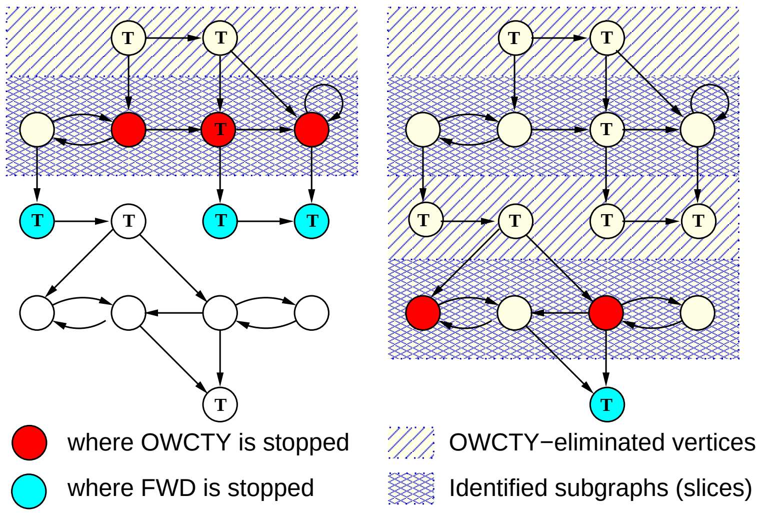


Figure 6.3: OBF *decomposition of the graph into the independent subgraphs*

6.1.3 RECURSIVE OBF Algorithm

Similarly to the COLORING algorithm, also the OBF procedure [25] aims at decomposing the graph in more than three component respecting subgraphs within a single recursion step. However, unlike the COLORING algorithm, the price of the OBF procedure is $\mathcal{O}(m+n)$. The name OBF is an acronym of the three procedures the algorithm comprises of: the OWCTY elimination, the BACKWARD reachability, and the FORWARD reachability.

To identify the independent subgraphs (*OBF slices* in terminology of the RECURSIVE OBF algorithm) of a rooted graph the procedure iteratively employs the following three steps until the whole graph is processed (see Figure 6.3 and Algorithm 19):

O Apply the OWCTY elimination to remove leading trivial strongly connected components (TRIMMING), and return vertices that were not eliminated, but some of their immediate predecessors were.

B Compute backward closure of vertices returned in the previous O step, vertices in the closure form a subgraph (slice) denoted by B.

F Compute forward closures of vertices returned in the previous O step within the subgraph given by B in order to remove the slice B from the graph. The immediate successors of vertices in B that are outside B are identified as new initial states (*Seeds*) for the rest of the graph.

Should the subgraphs be processed recursively by the RECURSIVE OBF algorithm [24, 23] they first need to be split into rooted subgraphs. In the main loop of the algorithm (see Algorithm 20) a vertex v from the set V is picked and computed its forward closure $range$ in the set V. Afterwards the OBF procedure (Algorithm 19) is executed on the vertex v and the set $range$ in parallel (line 9 of Algorithm 20). Vertices from $V \setminus range$ are processed in the next iteration of the main loop.

Before the main loop of the OBF procedure is started the size of the set $range$ is stored to the variable $originalRange$ in order to determine whether the found slice forms the SCC and thus whether the recursion stops. The OWCTY elimination is executed to repeatedly eliminate vertices with in-degree 0 reachable from $seeds$. Each of eliminated vertices forms trivial

Algorithm 19: CPU OBF procedure

```
1  Procedure OBF-X(seeds, range)
2  begin
3      originalRange ← |range|
4      while range $\neq \emptyset$ do
5          eliminated, reached, range ← OWCTY(seeds, range)
6          All elements of eliminated are trivial SCCs
7          B ← BackwardReachability(reached, range)
8          if $|B|$ = originalRange then
9              B is SCC
10         else
11             in parallel do
12                 Recursive-OBF(B)
13             seeds ← ForwardReachabilitySeeds(B, $Range$)
14         range ← range \ B
```

Algorithm 20: CPU Recursive-OBF algorithm

```
Input  : directed graph $G = (V, E)$
Output: strongly connected component decomposition of $G$
1  Procedure Recursive-OBF($V$)
2  begin
3      while $V \neq \emptyset$ do
4          Pick a vertex $v \in V$
5          range ← ForwardReachability($v$, $V$)
6          seeds ← $\{v\}$
7          $V \leftarrow V \setminus$ range
8          in parallel do
9              OBF(seeds, range)
```

SCC and thus they are also removed from the set $range$. The OWCTY elimination returns the set $reached$ containing the vertices where the elimination has been stopped (vertices with non-zero in-degree). Afterwards the backward closure B that forms a slice is computed from the vertices in the set $reached$. If $|B| = originalRange$ the slice B forms the SCC and the OBF procedure ends. Otherwise the Recursive OBF algorithm is applied on the slice B recursively. This nested invocation can run in parallel and thus it can increase parallelism. The set of $seeds$ for next iteration of the main loop of the OBF procedure is computed by the ForwardReachabilitySeeds procedure (a trivial modification of the Forward reachability). It returns all vertices from $range$ that are immediate successors of vertices in the slice B but do not belong to B. Vertices from $range \setminus B$ are processed in the next iteration of the main loop. The time complexity of the Recursive OBF algorithm is $\mathcal{O}((l+1) \cdot (m+n))$ where l is the longest path in the component graph of G.

Algorithm 21: GPU TrimmingKernel – device code (run in parallel $\forall v \in V$)

```
Input  : $gA_e, gA_i, gElim$, fixPointFound
1 if $gElim[v] = $ false then
2     eliminate ← true
3     if $\exists u \in succ(v) : gElim[u] = $ false then
      // $succ(v) = \{gA_e[gA_i[v]], \ldots, gA_e[gA_i[v+1]]\}$
4         eliminate ← false
5     if eliminate = true then
6         $gElim[v]$, fixPointFound ← true, false
```

6.2 Data-Parallel Graph Primitives

Instead of trying to devise a completely new algorithm for SCC decomposition that would be primarily suited for the CUDA architecture, we decided for a different course of action. And that not to change the provably correct layout of the existing parallel algorithms, but instead force both the underlying graph representation and the incorporated primitive graph operations to assume and enable vector processing. Hence once the representation and graph primitives (we are adopting both concept and name from numerical vector primitives from [27] for graph related setting) are prepared, we may start building the respective algorithms with relative ease.

The core graph primitive used in all of the algorithms is the computation of the forward and the backward reachability. The data-parallel version of the forward and the backward reachability was described in Section 3.2. Note that for the backward reachability we employ the method based on the graph transposition that has larger memory demands but it is more time efficient. Assume that the result of the computation of a reachability procedure is a vector *visited* of $|V|$ bits indicating which of the vertices have been visited. Initially, only the bits for the vertices of which reachability should be computed are set to one.

As explained in the previous section, some algorithms employ the Trimming procedure to efficiently deal with leading and terminal trivial SCCs. The goal of the procedure is to iteratively identify vertices of the underlying subgraph that have no immediate predecessors (in the case of leading components) or immediate successors (in the case of terminal components). Such vertices may be iteratively removed from the subgraph as trivial SCCs. Since we store both forward and backward edges the data-parallel version of the Trimming procedure is very simple (in contrast to the data-parallel version of the Elimination procedure with the scope of the OWCTY algorithm where only the forward edges are stored 4.3). The host code of the Trimming procedure is quite similar to the host code of forward reachability (listed as Algorithm 4) and therefore we list only the Trimming kernel for iterative elimination of vertices with no immediate successors, see Algorithm 21. Note that in the case of iterative elimination of vertices with no immediate predecessors the algorithm uses the backward edges. Again we can assume that the result of the procedure is a vector *eliminated* of $|V|$ bits indicating which of the vertices have been eliminated. Also note that the Trimming procedure can be easily augmented also to eliminate SCCs made of a single vertex with a self-loop by simply ignoring the self-loop edges.

Quite often, the computation of the reachability or the elimination needs to be restricted to a subgraph. To that end we denote each subgraph with a unique number and use other date structures of size $\mathcal{O}(|V|)$ to identify the subgraph each vertex belongs to. The thread in the kernel then updates the presence bit of a successor only if it is a part of the same subgraph as the source vertex. In the rest of the chapter we will use an equivalence relation $\sim$ to denote that two vertices are part of the same subgraph, and $[x]$ to denote vertices equivalent to a vertex x, i.e. $[x] = \{v \in V \mid x \sim v\}$.

6.3 Pivot Selection

There are several stages of the algorithms that require a single vertex to be chosen within a processed subgraph – the so-called pivot. Since pivot selection is a key graph primitive it plays significant role in practical performance of the algorithms. As a good heuristics to pivot selection the algorithms typically rely on a pseudo-random number generator. In our approach, we not only need to select a single pivot, but since we share kernels for graph procedures over multiple subgraphs, we need to choose a number of pivots, one for each subgraph. To that end, the usage of a random number generator seems inappropriate as we cannot guarantee that after a repeated random selection, the selected vertices will satisfy the desired distribution.

We have, therefore, opted for a different solution. The basic idea of our pivot selection is to let all vertices of a subgraph concurrently write their own unique identifiers to a single memory location. After that the location keeps a single value that identifies the pivot. Surprisingly, the most challenging problem when implementing the idea was where to define/store the memory location for a subgraph. Note that within a single kernel we may select pivots for quite a large number of subgraphs.

To solve the problem we employed the observation that a subgraph when defined is fully contained within a *parent* subgraph (the part of the graph from whose subgraphs the pivots are voted). For our first solution to the problem suppose now that the pivot of the parent subgraph has an extra space allocated to it. Then all the child subgraphs may be learnt about the pivot of their parent subgraph and thus they may use the extra space allocated to the parent subgraph pivot as the memory location they need for their own pivot selection. If there are multiple child subgraphs of one parent subgraph then they are serialized for the usage of the memory location. Since we do not know in advance which vertices become pivots, we reserve extra space for every vertex. This requires at least $|V|size(v)$ additional space, where $size(v)$ is the space necessary for identification of a single vertex.

In our second approach to the problem we have allocated a single shared vector of memory locations and make sure that every computed subgraph gets a unique pointer to the vector. If each recursive step defines bounded number of subgraphs, we can compute the unique number of a child subgraph from the unique number associated with the parent subgraph. For example, in the case of the FB algorithm, the bound is equal to three, so the three new subgraphs of a parent subgraph with unique number i will get numbers $3i + 0$, $3i + 1$, and $3i + 2$. An obvious problem of the second solution is that the number of subgraphs is unknown in advance, hence the unique numbers associated with the subgraphs may grow beyond the size of the preallocated vector. Note that if that happens a lot of unique numbers of subgraphs that were parent subgraphs before, are unused. We therefore postpone the computation of the algorithm and run a heuristics that renumbers active subgraphs so that they get numbers somewhere at the beginning of the vector. To compute new unique numbers of active subgraphs we employ hash function. Collisions due to the hash function are relatively rare, and they are handled sequentially after the renumbering by the hash function.

6.4 Processing of Independent Subgraphs and Kernel Sharing

Within the scope of SCC decomposition the computation of forward or backward closures are typically restricted to a particular component respecting subgraph of the original graph. As soon as the algorithm is deeper in its recursion, the same procedures are typically executed over different subgraphs. If each operation such as the computation of a forward closure, is implemented as a CUDA kernel, we can easily mimic the recursion as suggested by the algorithms within the host code (we let the host code call a separate kernel for each graph operation over every subgraph in every recursion branch). However, if a kernel is executed in this approach over a whole matrix, a lot of CUDA threads, namely those that are deployed for vertices out of the processed subgraph, are idling or performing useless work. We can avoid this inefficiency if we deploy only the threads for vertices of the subgraph, but to be able to do so we would have to renumber the vertices of the graph so that the vertices of the subgraph are well-distributed in the vector of vertices, i.e. at least in a number of continuous blocks. This renumbering would of course kill any benefit the preprocessing might have brought.

We therefore proceed in a different way and share the calls to the kernels that are made for the same operation over different subgraphs in different recursion branches. In particular, if we synchronize the recursion of the algorithm so that in the second recursion step, let us say, the computation of a forward closure is executed simultaneously over multiple independent subgraphs, we can employ a single CUDA kernel to compute all the forward closures at the same time. This synchronization over recursion deepening and kernel sharing principles allow us to reformulate the recursion present in the algorithms by means of iterative procedures (while loops). This is exemplified on pseudo-code for the FB algorithm listed in Algorithm 22 (though the idea is common to all implemented algorithms). According to our experience the penalty for explicit synchronization due to loop iterations is easily outweighed by performance gain achieved due to the kernel sharing.

6.5 Designing CUDA Accelerated Algorithms Using Graph Primitives

We have prepared all the representation details and graph primitives to be applied on graphs divided potentially into multiple subgraphs. Thus the idea foreshadowed above of mapping the recursive nature of the presented algorithms into iterative processing, enables us to implement the algorithms on various vector models of computation, e.g. on the heterogeneous model (presented in Section 2.1) that all CUDA-equipped off-the-shelf computers possess.

As we describe in Section 3.2 the general work-flow of GPU accelerated graph algorithms is the combination of out-of-order CPU and data-parallel processing GPU and requires to create the complete representation of the graph. In the context of SCC decomposition we assume that the complete representation of the graph G and the transposed graph G^T is given in advance. For the sake of simplicity we do not list in the following pseudo-codes the initial transfer of the representation to GPU and final transfer of the result back to CPU. All procedures that are called transfer only a bit indication whether a fixpoint has been reached (see Section 3.2 for more details).

Algorithm 22: GPU FB algorithm - host code

```
Input  : directed graph $G = (V, E)$
Output: strongly connected component decomposition of $G$
Data   : $u \sim v \Leftrightarrow$ range$[u]$ = range$[v]$
1 INIT(pivots, range, visited, eliminated, terminate)
2 while terminate = false do
3     FORWARDREACHABILITY($G$, pivots, visited.$f$)
4     BACKWARDREACHABILITY($G$, pivots, visited.$f$)
5     TRIMMING($G$, eliminated)
6     PIVOTSELECTION(pivots, range, visited, eliminated)
7     UPDATE(range, visited, eliminated, terminate)
```

Algorithm 23: GPU COLORING algorithm - host code

```
Input  : directed graph $G = (V, E)$
Output: strongly connected component decomposition of $G$
Data   : $u \sim v \Leftrightarrow$ oldMaps$[u]$ = oldMaps$[v]$
1 INIT(pivots, maps, oldMaps, visited, terminate)
2 while terminate = false do
3     COLORINGPROPAGATION($G$, pivots, maps, oldMaps)
4     BACKWARDREACHABILITY($G$, pivots, maps, visited)
5     UPDATE(pivots, maps, oldMaps, visited, terminate)
```

6.5.1 CUDA Accelerated FORWARD-BACKWARD Algorithm

Once the algorithm is given as iterative procedure, the adaptation for CUDA environment is rather straightforward. See the pseudo-code as listed in Algorithm 22. We are using the vector *visited* to indicate which of the vertices belong to the forward respectively backward closure (*visited.f*, *visited.b*), vector *elim* to keep the eliminated vertices and vector *pivots* to determine the pivots for next iteration of the algorithm. Finally, the vector *range* is used to identify the subgraph that each vertex belongs to. The UPDATE kernel recomputes the *range* vector (the relation $\sim$) according to the vectors *visited* and *elim*. Moreover, it sets the variable *terminate* to `true` if all vertices from previous iteration were either visited by both the forward and backward reachability procedure, or were eliminated.

6.5.2 CUDA Accelerated COLORING Algorithm

Likewise the FB algorithm, also the COLORING algorithm can be formulated as an iterative algorithm. In such a case every loop iteration consists of two procedures: the color-propagation procedure that partitions the graph into multiple subgraphs, and the backward reachability procedure that identifies and removes the leading component of every subgraph. We list pseudo-codes of the host code of the COLORING algorithm and the CUDA kernel for the color propagation, see Algorithm 23 and Algorithm 24, respectively. Note that the host code of the color propagation has the similar structure to the host code of the

Algorithm 24: GPU ColoringKernel – device code (run in parallel $\forall v \in V$)

```
Input : $gA_e, gA_i, gMaps$, inner
1 $gMaps[v] \leftarrow \max\{v,\ gMaps[v]\}$
2 foreach $u \in succ(v)$ do                // $succ(v) = \{gA_e[gA_i[v]], \ldots, gA_e[gA_i[v+1]]\}$
3     if $v \sim u \ \wedge\ gMaps[v] < gMaps[u]$ then
4         $gMaps[v]$, inner $\leftarrow gMaps[u]$, true
5 if $gMaps[v] = v$ then
6     $gPivots[v] \leftarrow$ true
```

Algorithm 25: GPU OBF algorithm - host code

```
Input : directed graph $G = (V,\ E)$
Output: strongly connected component decomposition of $G$
Data : $u \sim v \Leftrightarrow$ range$[u]$ = range$[v]$
1 Init(phase, range, eliminated, terminate)
2 while terminate = false do
3     while Interruption($i$) = false do
4         OBFKernel(O, B, F, $V$)
5     if phase$[i] \in$ O then
6         UpdateO($i$)
7     if phase$[i] \in$ F then
8         UpdateF($i$)
9     if phase$[i] \in$ B then
10        UpdateB($i$)
11    Update(range, eliminated, terminate)
```

ForwardReachability procedure listed as Algorithm 4. Also note that the color propagation procedure computes the vertex (pivot) for the succeeding backward reachability procedure, and that variable *inner* is used to detect that no fixpoint has been reached yet.

6.5.3 CUDA Accelerated Recursive OBF Algorithm

Unlike the FB and the Coloring algorithms, the adaptation of the OBF algorithm to the CUDA environment was more involved. In our final solution, we have decided not to use three independent CUDA kernels for individual phases (O, B, and F), but instead we have devised a single CUDA kernel that performs all three phases at the same time. Every vertex keeps extra information to know in which phase it is currently processed. The host code and the device code of the OBF algorithm are listed as Algorithm 25 and 26, respectively.

OBF-kernel proceeds until one of the phases terminates, which is detected by the Interruption procedure. After the termination, vertex i is returned to identify the subgraph of the phase that has terminated and caused the interruption. An update procedure is then executed according to the type of the phase:

Algorithm 26: GPU OBFKERNEL – device code (run in parallel $\forall v \in V$)

```
Input  : $gA_e$, $gA_i$, $gTransA_e$, $gTransA_i$, $gReach$, $gElim$, $gPhase$
1  if $gPhase[v] = \mathsf{O} \wedge gReach.o[v] = \texttt{true}$ then
2      if $\forall u \in pred(v).u \sim v : gElim[u] = \texttt{true}$ then
       // $pred(v) = \{gTransA_e[gTransA_i[v]], \ldots, gTransA_e[gTransA_i[v+1]]\}$
3          foreach $w \in succ(v)$ do        // $succ(v) = \{gA_e[gA_i[v]], \ldots, gA_e[gA_i[v+1]]\}$
4              $gReach.o[w] \leftarrow \texttt{true}$
5          $gElim[v] \leftarrow \texttt{true}$
6  else if $gPhase[v] = \mathsf{B}$ then
7      foreach $u \in pred(v)$ do
8          $gReach.b[u] \leftarrow \texttt{true}$
9      else
10         foreach $u \in succ(v)$ do
11             $gReach.f[u] \leftarrow \texttt{true}$
```

- UPDATE_O updates not eliminated vertices of $[i]$ to be processed by the next phase (B).
- UPDATE_B checks whether the reached part of $[i]$ is rooted. If so, *range* of vertices in the reached part is set to a common unique value and the part is eliminated as a SCC. If the reached part was not rooted, we select a pivot and execute a forward closure to get a rooted subgraph. The phase of the rest of vertices in $[i]$ is set back to O. We also set $reach.o[v]$ for every vertex v that is a successor of a reached vertex in $[i]$.
- UPDATE_F selects a pivot from the not reached part of $[i]$ to start a new forward reachability there. Simultaneously the phase is set to O for the reached vertices and the pivot of $[i]$ is the only one set to reached. Finally, the two parts (reached and not reached) are separated (within $\sim$) by setting the range of the reached vertices to a new unique value.

The UPDATE procedure merely checks whether all the vertices were eliminated (either by OWCTY or when found to be in a rooted subgraph by the UPDATE_B procedure) and sets the $terminate$ variable accordingly.

It is clear to see that OBF-KERNEL forces the individual threads to perform different tasks if their vertices fall into different sets. Which is in the opposition to one of the principles of CUDA programming since all threads within any *warp* should at one time perform the same instruction. This is of a little problem when the sets O, B, F (containing vertices in the respective phases) are large and consist of consecutive vertices (meaning they are stored in an uninterrupted row in the adjacency matrix representation), but as they grow smaller or less compact it might entail considerable slowdown.

We have tried to at least partially eliminate this problem by opting out subgraphs that are too small. Once the size of a subgraph drops below a given threshold, we employed the COLORING algorithm to finish the decomposition of the subgraph. In order to do so, we had to adapt the OBF algorithm to compute the sizes of the produced subgraphs. Once all the subgraphs are small enough we actually stop the OBF algorithm and continue with the

Coloring algorithm. Despite the inevitable overhead of the size computation, this strategy often lead to significant improvement according to our experimental measurements. Furthermore, OBF can be augmented with an initial call to the Trimming procedure in order to avoid the costly OBF-like computation on all the leading and terminal trivial components.

6.5.4 Employing Multiple CUDA Devices

CUDA technology provides tremendous computing power due to the massive parallelism, however, with limited memory resources. As such its applicability to SCC decomposition is limited by the size of the graph that can fit the memory of a single CUDA device.

In Section 5.2 we show how to efficiently employ multiple CUDA devices to solve the accepting cycle detection problem. More precisely, we were able to distribute computation of various elementary graph algorithms among multiple GPUs and alter the underlying graph representation to successfully overcome the space limitation. And even though our parallel algorithm was quite inter-CUDA communication intensive, we were able to preserve a decent time efficiency of the whole parallel system.

It is worthy to observe that accepting cycle detection and SCC decomposition algorithms both use the same elementary graph procedures like reachability computation or unique and totally order identifiers propagation. Thus is seems tractable, due to the idea of building SCC decomposition algorithms from graph primitives, to follow the same process of devising multi GPU implementation as presented in Section 5.2.

6.6 Experimental Evaluation

We compare the performance of the described CUDA algorithms with the CPU implementation of Tarjan's algorithm that is considered to be the best sequential algorithm for SCC decomposition. For this purpose we have implemented our own highly optimized version of Tarjan's algorithm using identical representation of adjacency list as the one used for CUDA computation. Our implementation of Tarjan's algorithm outperforms (2 times) the Boost [31] implementation.

We have also implemented multi-core versions of the algorithms for the standard parallel shared-memory platforms. To that end we have experimented with two implementations. In the first variant, we basically let CPU cores perform the CUDA version of each algorithm without employing CUDA device. In the second version, we took the approach of parallel distributed-memory graph traversal procedures, see e.g. [23], and we applied it to shared-memory environment. For the shared-memory message passing we used lock-free FIFO data structures, as suggested in [12]. Unfortunately, none of our implementations were able to outperform Tarjan's algorithm using quad-core architecture, which can be explained by extremely cache-efficient representation of the graph used for Tarjan's algorithm that was used on relatively small graphs (we have experimented with graphs whose representation fitted 1.5 GB of RAM of our CUDA GPU card). All the experiments were run on a Linux workstation with an AMD Phenom(tm) II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066 MHz RAM and NVIDIA GeForce GTX 480 GPU providing 480 cores and 1.5 GB of GPU memory. In order to explore the scalability of the data-parallel algorithms we also use the previous generation of GPU i.e. GeForce GTX 240 GPU providing 240 cores and 1 GB of memory.

To evaluate the algorithms we used input graphs as generated by Georgia Tech. graph generator (GTgraph) [5] containing: Scalable Synthetic Compact Applications (SSCA) bench-

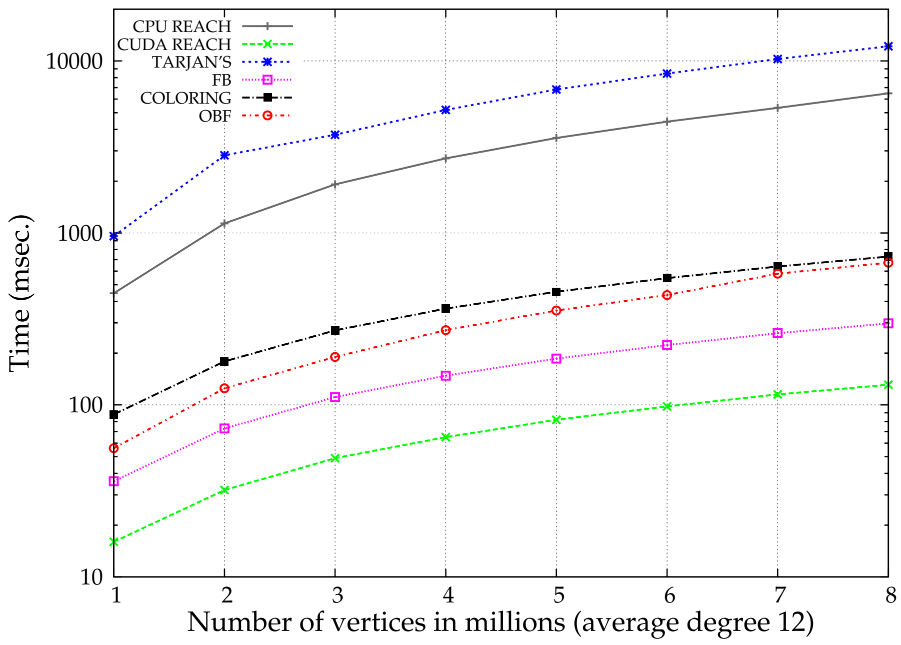


Figure 6.4: *Runtimes for Random graphs in milliseconds.*

mark suite [4], Recursive Matrix (R-MAT) generator [40], Erdös-Rényi random graph generator; and graphs as produced by the enumerative model checker DiVinE [16]. Since all algorithms use the identical representation of the graphs, we do not take the time of its construction into account for the experimental evaluation.

We provide comparison of the following algorithms: the serial CPU-based forward reachability denoted by CPU REACH, the CUDA-based forward reachability denoted by CUDA REACH, Tarjan's algorithm, the CUDA-based FB algorithm (+ Trimming), the CUDA-based Coloring algorithm, and the CUDA-based OBF algorithm (+ Trimming, + Coloring, + Trimming and Coloring). Table 6.1 lists runtimes of the algorithms if executed on the three types of synthetic graphs with average degree set to twelve and scaled up by the number of vertices. Table 6.2 gives runtimes of the algorithms for graphs corresponding to model checking problems. The runtimes are also plotted in Figures 6.4, 6.5, 6.6, and 6.7 using the best time available among versions of individual parallel algorithms.

Note that the computation of the algorithms which has reached 50 seconds time limit has been aborted and is marked in the tables by '-'. Also note that the reported runtimes for CUDA-based forward reachability exhibit similar values to the values reported in [62, 63].

Table 6.3 enumerates the difference in runtimes of the algorithms when executed on the previous and current generation of NVIDIA graphics cards, i.e. GTX 280 and GTX 480. Again, the dashes mean that the algorithm did not finish in time (50s) on some instances. Examining the hardware specifications of the respective cards we see that the frequency of individual computation cores was increased rather modestly (from 602 to 700 MHz) when compared to doubling of the number of cores (from 240 to 480 cores). Bearing this in mind, we may consider the observed speedup (closing to threefold in some instances) to be a witness of effective scalability of the presented data-parallel algorithms. The seeming superlinear speedup is to be accredited to both the higher frequency and unprecedented cached memory hierarchy.

Random Graphs

Algorithm	Number of vertices in milions (Number of SCCs)								Total
	1M (16)	2M (31)	3M (30)	4M (48)	5M (61)	6M (72)	7M (97)	8M (106)	
CPU REACH	446	1135	1915	2712	3563	4440	5331	6479	26021
CUDA REACH	16	32	49	65	82	98	115	131	588
Tarjan's	957	2825	3722	5195	6822	8443	10265	12169	50398
FB	39	85	127	183	243	309	384	456	1826
FB + Trim	36	73	111	148	186	223	261	298	1336
Coloring	88	179	271	363	455	546	638	729	3269
OBF	56	125	190	272	354	435	580	690	2702
OBF + Col	62	129	196	284	372	459	620	741	2863
OBF + Trim	81	165	249	334	418	502	586	671	3006
OBF + Col + Trim	81	166	251	336	421	506	590	676	3027

R-MAT Graphs

Algorithm	Number of vertices in milions (Number of SCCs)								Total
	1M (0.48M)	2M (0.97M)	3M (0.97M)	4M (1.9M)	5M (1.0M)	6M (2.0M)	7M (2.9M)	8M (3.9M)	
CPU REACH	280	744	1484	1910	3060	3504	3921	4428	14903
CUDA REACH	16	31	49	64	83	99	114	128	584
Tarjan's	785	1851	3230	4332	6171	7365	8529	9738	42001
FB	-	-	-	-	-	-	-	-	-
FB + Trim	36	74	134	158	241	281	305	329	1558
Coloring	87	180	324	367	548	659	660	745	3570
OBF	-	-	-	-	-	-	-	-	-
OBF + Col	-	-	-	-	-	-	-	-	-
OBF + Trim	83	203	343	427	595	702	796	887	4036
OBF + Col + Trim	80	197	305	405	513	618	728	827	3673

SSCA#2 Graphs

Algorithm	Number of vertices in milions (Number of SCCs)								Total
	1M (576)	2M (1.1K)	3M (1.7K)	4M (2.2K)	5M (2.8K)	6M (3.4K)	7M (4.0K)	8M (4.4K)	
CPU REACH	350	790	1274	1794	2319	2866	3451	4141	16985
CUDA REACH	31	65	110	135	195	223	261	316	1336
Tarjan's	601	1313	2116	2973	3721	4565	5513	6377	27179
FB	299	833	2089	3003	5168	7702	8455	11483	39032
FB + Trim	72	155	284	352	538	661	851	1046	3959
Coloring	1646	3483	6532	9373	14095	15511	17352	27020	95012
OBF	281	913	2008	3092	4872	6939	9401	11528	39034
OBF + Col	316	1025	2269	3536	5574	7964	10855	13223	44762
OBF + Trim	143	310	532	724	989	1257	1539	1930	7424
OBF + Col + Trim	140	294	481	646	859	1035	1328	1650	6433

Table 6.1: *Runtimes for synthetic graphs in milliseconds. Average degree of the graphs is 12*

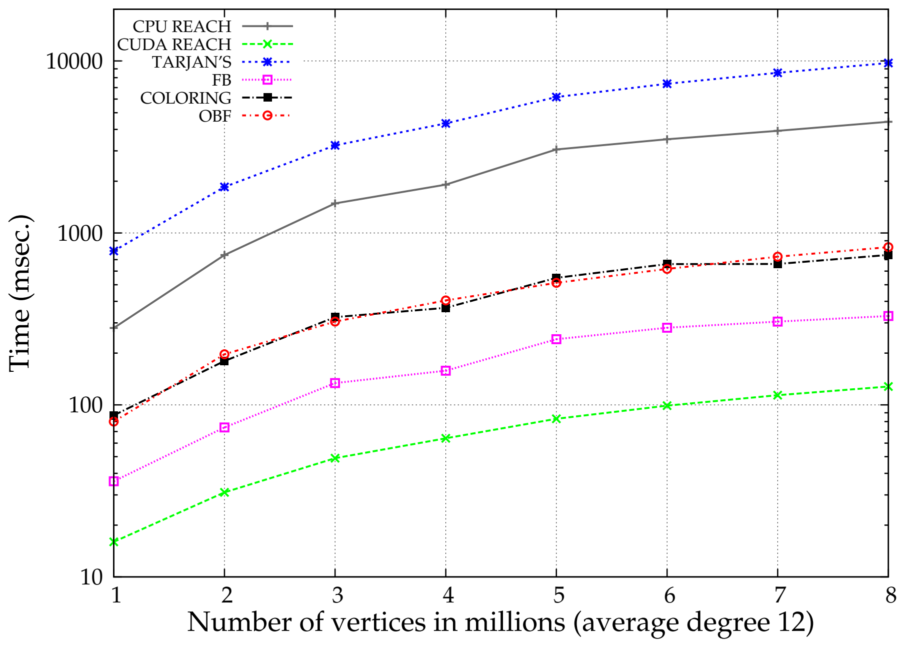


Figure 6.5: *Runtimes for R-MAT graphs in milliseconds.*

Model	Algorithm									
	CPU REACH	CUDA REACH	TARJAN'S	FB	FB + TRIM	COLORING	OBF	OBF + COL	OBF + TRIM	OBF + COL + TRIM
leader-2 (0.7M, 3.8M, 0.7M)	23	6	197	383	37	2	18	19	28	28
phils (0.7M, 6.0M, 59K)	29	3	287	20344	31	57	54	34	52	29
fisher (2.5M, 13.8M, 81K)	84	7	597	18206	54	106	175	76	185	84
anderson-2 (3.1M, 13.4M, 1.6M)	100	20	774	-	83	459	115	132	90	93
leader-1 (3.6M, 26.6M, 3.6M)	129	30	582	7504	324	17	363	84	379	403
elevator-2 (6.4M, 83.3M, 1)	323	48	2437	145	147	3441	199	200	251	249
anderson-1 (8.9M, 47.7M, 4.3M)	325	43	2738	-	600	537	415	420	312	389
peterson (9.5M, 42.0M, 18K)	387	35	2740	13466	166	487	211	224	266	279
elevator-1 (8.6M, 89.4M, 2.0M)	400	54	2933	-	1049	5969	1336	1370	1384	1375
Total	1800	246	13285	-	2491	13075	2886	2559	2947	2929

Table 6.2: *Runtimes for model checking graphs in milliseconds. For each model we list in brackets the number of vertices, edges and SCCs, respectively.*

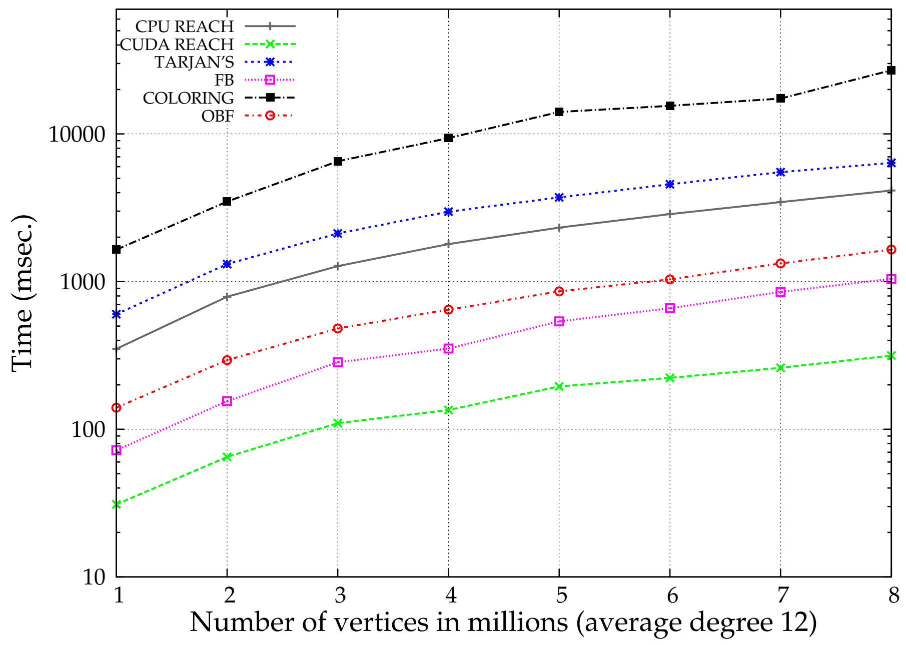


Figure 6.6: *Runtimes for SCCA#2 graphs in milliseconds.*

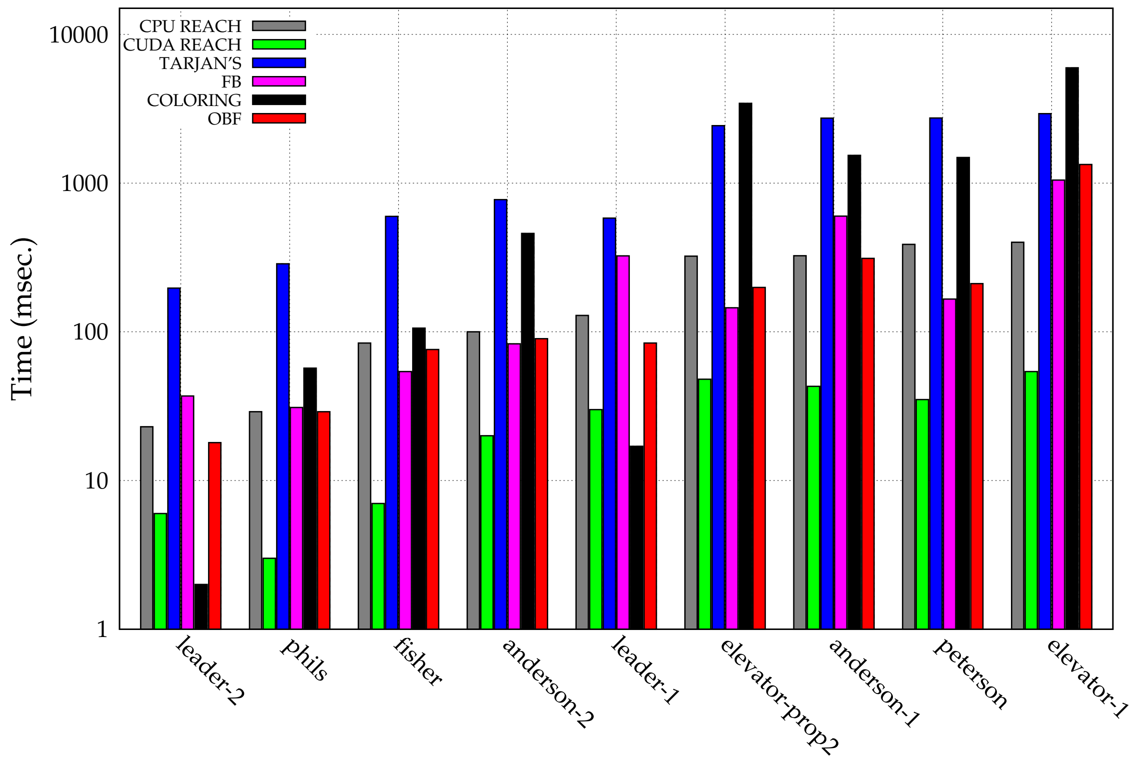


Figure 6.7: *Runtimes for model checking graphs in milliseconds.*

Finally, Table 6.4 gives the overall achieved speedup when the best time available among versions of individual parallel algorithms is considered.

Graph type	GPU device	Algorithm							
		CUDA REACH	FB	FB + TRIM	COLORING	OBF	OBF + COL	OBF + TRIM	OBF + COL + TRIM
Random	GTX 280	1501	3980	2888	5205	5447	6274	5754	5763
	GTX 480	588	1826	1336	3269	2702	2863	3006	3027
R-MAT	GTX 280	1651	-	3578	7496	-	-	7120	7190
	GTX 480	584	-	1558	3570	-	-	4036	3673
SSCA#2	GTX 280	3923	78892	10245	135003	68040	102546	16625	15186
	GTX 480	1336	39032	3959	95012	39034	44762	7424	6433
Model Checking	GTX 280	526	-	5188	23152	4604	4151	4344	4555
	GTX 480	246	-	2491	13075	2886	2559	2947	2929

Table 6.3: *Runtimes in milliseconds summed over respective graph types for the previous and current generation of GPUs.*

Graph type	GPU device	Speedup against CPU REACH	Speedup against TARJAN'S		
		CUDA REACH	FB	COLORING	OBF
Random	GTX 280	17.3	17.5	9.7	9.2
	GTX 480	44.3	37.7	15.4	18.7
R-MAT	GTX 280	9.0	11.7	5.6	5.9
	GTX 480	25.5	27.0	11.8	11.4
SSCA#2	GTX 280	4.3	2.7	0.2	1.8
	GTX 480	12.7	6.9	0.3	4.2
Model Checking	GTX 280	3.4	2.6	0.6	3.8
	GTX 480	7.3	5.3	1.0	5.6

Table 6.4: *Overall achieved speedup for the previous and current generation of GPUs.*

We have observed that the performance of CUDA-based algorithms deeply depends on the average degree of the vertices in the graph. Simple reachability procedure (forward closure) performs in linear time with respect to the diameter of the graph, which tends to expand as the average degree decreases. For graphs with low degree, the performance of the reachability procedure may be improved using our heuristics to reduce the number of memory loads, see Subsection 3.2. Generally, we observe that the scalability and efficiency of the parallel reachability procedure effectively limits scalability and efficiency of the SCC decomposition algorithms. We can conclusively state that in most experiments our algorithms were able to reach this limit.

Other observations are as follows. For Random graphs, where most of the vertices have similar degree, all algorithms significantly outperform (40 times) TARJAN'S algorithm as they can effectively exploit the parallelism. R-MAT graphs have uneven degree distribution with most vertices of rather a small degree. These graphs expand slowly in each iteration and exhibit uneven load balancing and thus the performance of algorithms based on the computation of the forward reachability is very poor. However, adding the TRIMMING phase drastically improves their performance and leads to overall twentyseven-fold speedup. SSCA graphs exhibit similar degree distribution to the R-MAT graphs, but they typically contain large number of small non-trivial components, which limits the efficiency of the Trimming procedure (overall seven-fold speedup). For model checking graphs, the average degree of a vertex is rather small compared to the synthetic graphs, therefore the runtimes are not as good as in the case of synthetic graphs (overall six-fold speedup).

For synthetic graphs, the FB algorithm with Trimming has the best times. This is because the graphs usually contain small number of large components and large number of trivial or very small components. Such a structure of a graph causes the whole decomposition process to boil down to a few invocations of the forward and backward reachability interleaved with the Trimming procedure. On the other hand model-checking graphs contain in general bigger number of large components. For graphs with such a structure the OBF algorithm has the potential to significantly outperform the other ones. Finally, we observe that the Coloring algorithm exhibits rather unstable performance. While thriving on highly disconnected graphs or graphs with many small components, its performance degrades as the size of the components in the graph grows.

6.7 Conclusions

In this chapter, we demonstrated successful redesign of several parallel algorithms for SCC decomposition. The redesigned versions allow for computation acceleration on massively parallel hardware platforms such as modern GPUs. In particular, we designed a new massively parallel procedure for pivot selection and reformulated the parallel SCC decomposition algorithms in order to outperform the optimal but inherently sequential Tarjan's algorithm.

Thus while not proposing a strictly speaking new parallel algorithm for SCC decomposition we instead suggest methods allowing to map the recursive nature of the presented algorithms into iterative procedures. Hence instead of devising an ad-hoc solution for SCC decomposition on CUDA we tried to establish more general approach to irregular graph algorithms. This approach aims at enabling implementation of algorithms on various vector models of computation and propounds how could other graph algorithms be altered to benefit from SIMD architecture.

We also carried out an extensive experimental evaluation of the known algorithms on several types of graphs proving that with single GTX 480 GPU card we can easily outperform the optimal serial algorithm. The results imply that the FB algorithm is to be declared a winner among the particular algorithms having speedup up to forty-fold and rarely going below six-fold. Though Coloring reaches surprisingly good results on some instances, there are only a few of them. And finally despite achieving rather steady performance, the OBF algorithm seems to fall behind the other algorithms. This is because the random graph generators as used in our study fail to provide graphs with significant amount of nontrivial and large components. Whether this is the case of all application domains is, however, questionable.

In our future work, we intend to apply the recently proposed methods such as the warp-centric programming [69] and the linear parallelization [91] (briefly described in Chapter 3) to further accelerate the SCC decomposition on the massively parallel SIMD architectures the modern GPUs offer.

Chapter 7

Data-Parallel Algorithms for Optimal Cycle Mean Problem

In this chapter, we aim at optimal cycle mean (OCM) problem. As we discuss in Section 2.4, inspection of graph cycles is one of the possible means to deal with performance prediction. It is imperative that the process of finding the optimal cycles is not overly expensive since many of these applications require the critical cycle to be found repeatedly [50]. To further emphasize the necessity of having efficient algorithmic solution to this problem we present two practical observations. The graphs representing even a small system can be exceedingly large, containing millions of vertices. Moreover, the number of cycles can be exponential with respect to the number of vertices thus making the trivial inspection of all cycles in the graph impractical. Yet the asymptotic complexity of even the best sequential algorithms is very high, which renders the applicability of OCM-based performance analysis limited to small systems.

To the best of our knowledge, there exists only distributed-memory parallelization of OCM algorithms [41]. Since distributed computation has led to rather moderate results, we intend to improve the runtime of OCM computation and consequently the applicability of OCM-based performance analysis by employment of SIMD parallelism offered by modern GPUs.

7.1 Algorithms for Optimal Cycle Mean Problem

First, we describe in more details the basic ideas behind the algorithms for OCM problem. We focus on parallelization of these algorithms and we choose the best candidate for our new data-parallel algorithm.

In order to describe further details of some OCM algorithms we introduce the notion of parametric weight functions. Given a weight function w and a real number Λ, we define *parametric weight function* $w_\Lambda \overset{df}{=} w - \Lambda$. We say that Λ is *feasible* for a graph G if no cycle in the graph has negative weight with respect to w_Λ.

Now we will prove two very simple propositions that provide theoretical basis for OCM algorithms. Recall that *optimal cycle mean* for a given graph G and weight function w is denoted as μ^*.

Proposition 7.1.1. *Λ is feasible $\Leftrightarrow \Lambda \leq \mu^*$.*

Proof. $\Rightarrow$ Let ζ_c be any of the cycles with the optimal mean in G. Then $w_\Lambda(\zeta_c) = w(\zeta_c) - \Lambda|\zeta_c| \geq 0$ hence $w(\zeta_c) \geq \Lambda|\zeta_c|$ and $\frac{w(\zeta_c)}{|\zeta_c|} = \mu^* \geq \Lambda$.

$\Leftarrow$ Let ζ_m be an arbitrary cycle in G such that $w_\Lambda(\zeta_m) < 0$. Then $w(\zeta_m) - \Lambda|\zeta_m| < 0$ and thus $\Lambda > \frac{w(\zeta_m)}{|\zeta_m|} = \mu^*$, which contradicts the assumption that $\Lambda \leq \mu^*$. □

Proposition 7.1.2. *Minimal cycle mean of G is equal to the smallest of minimal cycle means among the strongly connected components of G.*

Proof. First we observe that the component graph $G_c = (E_c, V_c)$ of G is acyclic: Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2) \in V_c$ such that there is a cycle in G_c starting and ending in G_1 and passing G_2. Since every vertex is part of some strongly connected component, the sets V_1 and V_2 are nonempty. Let v_1, v_2 be arbitrary vertices from V_2, V_2, respectively and let $< e_1, \ldots, e_n >$ be the path between G_1 and G_2 in G_c. The definition of strong connectivity implies existence of a path $< e_{01}, \ldots, e_{0i_0}, e_1, \ldots, e_n, e_{n1}, \ldots, e_{ni_n} >$ in G between v_1 and v_2. Dually, it is possible to show that there is also a path from v_2 to v_1. Hence $V_1 \cup v_2$ together with $E \cap (V_1 \cup v_2) \times (V_1 \cup v_2)$ form a strongly connected component of G which contradicts the maximality of V_1.

Thus it is obvious that cycles in G are only within its strongly connected components and by finding the smallest among the minimal mean cycles of the strongly connected components we also find minimal mean cycle of G. □

Through the course of study of optimal properties of graph cycles a plethora of algorithms emerged. These algorithms, while not sharing similar concepts, can be divided into several groups according to what graph property they use to find critical cycles. Since our goal in this section is to choose one of the algorithms which would be most suitable for data-parallel implementation we will describe all three categories of OCM algorithms and consider their aptness for our purpose.

There are various limitations imposed on the potential algorithms should they be even considered for SIMD acceleration. First of all, the data structures used must be to a very large extend in form of vectors: stacks, queues or heaps are not suitable for effective SIMD parallelization. Then the kernels should prevalently address the whole set of vertices (or edges): limiting the computation to an insufficiently small subset would prevent utilization of the computational power. Yet the vector-wide operations are rather costly even on many-core architectures and the complexity of the algorithm is practically measurable in their number. Finally, there is also a non-negligible overhead of kernel calls and thus even a very fast kernel should not be run excessively.

A strong relation can be observed between the OCM and the *Shortest Path Feasibility* (SPF) problems [50]. This relation should be much more apparent once we formulate the OCM problem as a linear programming problem: μ^* is the optimal solution of

$$\begin{aligned} &\textbf{max } r \textbf{ subject to} \\ &\quad d(v) \le d((u) + (w((u,v)) - r) \\ &\quad \forall e = (u,v) \in E, \end{aligned} \tag{7.1}$$

where d stand for *distance*, i.e. minimal-cost path from the source node to v. According to the previous formulation, we can equivalently search for the maximal parameter r such that G with w_r contains no negative cycle. Following Proposition 7.1.1 such r is exactly the optimal cycle mean. As a result, existence of an efficient implementation of an SPF algorithm enables an efficient solution to the OCM problem. Furthermore, any future improvements to an SPF procedure automatically translates to improvements in this OCM problem solution.

All SPF algorithms have a common basic step: the *scanning method* [44]. This method assumes we maintain for every vertex u its potential $\pi(u)$, parent $p(u)$, and a label $S(u) \in \{unreached, found, scanned\}$. Initially, only the root vertex is labeled $found$, all other vertices are $unreached$, and potential of all vertices is set to zero. A single $found$ vertex is then repeatedly scanned using Algorithm 27.

Algorithm 27: CPU Scanning Method of SPF algorithms

```
Input: A $found$ vertex $u$
1 foreach $e = (u, v) \in E$ do
2     if $\pi(u) + w(e) < \pi(v)$ then
3         $\pi(v), p(v), S(v) \leftarrow \pi(u) + w(e), u, found$
4 $S(u) \leftarrow scanned$
```

The scanning method repeats until there is no $found$ vertex or until the algorithm finishes n *passes*. Recall that $n = |V|$. Passes of an SPF algorithm are defined inductively:

0-th	pass is the initialization,
i-th	pass scans all vertices labeled $found$ during the $(i-1)$st pass.

If all n passes are performed, the graph inevitably contains a negative cycle.

7.1.1 Cycle-Based

The arguably most straightforward application of shortest path feasibility solution to the OCM problem is the *cycle-based* approach. The idea is to maintain an upper bound Λ of the minimal cycle mean, i.e. $\Lambda \geq \mu^*$, and a cycle C such that $\Lambda = \mu(\mathsf{C})$. If $\Lambda > \mu^*$ then new better upper bound Λ' of minimal mean cycle can be detected with the SPF algorithm provided that it uses parametric weight function w_Λ. The newly computed upper bound Λ' is used in the next iteration of the algorithm as Λ. The whole procedure repeats until no improvement of upper bound can be found, which indicates that $\Lambda = \mu^*$.

A classical implementation of the cycle-based approach is Howard's algorithm [70], which further improves the approach by altering the SPF procedure. At the end of each pass it checks the parent graph, induced by edges $(p(v), v)$, for cycles. Existence of a cycle allows to restate Λ to a value that sets to zero the weight of the most negative cycle. Should there be no negative cycle the improved SPF algorithm either terminates, if all reduced weights are non-negative, or it continues with the next pass.

The asymptotic time complexity of cycle-based algorithms is exponential in the worst case, e.g. Howard's algorithm runs in time $\mathcal{O}(nmN)$ [46], where N is the number of simple cycles in G. In practice, however, these algorithms often surpass algorithms with known polynomial bound on their time complexity.

The cycle-based approach seems to be fairly compatible with the SIMD computation. Namely, the SPF subroutine is a vector-wide propagation of values, the minimal cycle location on the parent graph is feasible to parallelize, and most importantly sequential experiments suggest that the algorithm typically performs only very few passes of the underlying SPF subroutine.

7.1.2 Binary Search

The *binary search* approach is slightly more involved. It maintains both upper and lower bounds $\Lambda_1 \leq \mu* \leq \Lambda_2$ together with a cycle C such that $\mu(\mathsf{C}) = \Lambda_2$. The SPF subroutine is repeatedly called with parametric weight function w_Λ, where Λ, as the name suggests, is set to $\frac{\Lambda_1+\Lambda_2}{2}$. In case a negative cycle ζ is found, we set C and Λ_2 to ζ and $\mu(\zeta)$, respectively. Since we did not use Λ_2 as a parameter we cannot be certain of the value of the optimal cycle mean in either of SPF answers, and thus if no negative cycle is found we set Λ_1 to Λ. The termination criterion for binary search approach is $\Lambda_2 - \Lambda_1 < \epsilon$. If ϵ is chosen sufficiently small, C will be the critical cycle.

The well-known implementation of binary search is due to Lawler [81], who also proved the runtime of his algorithm to be in $\mathcal{O}(nm \lg(W/\epsilon))$, where W is the maximal edge weight. Structurally, there is no apparent reason why Lawler's algorithm should be inappropriate for data-parallelization, yet the fact that the SPF subroutine requires up to n passes (and given the idea of binary search it often is necessary to carry out all n passes) renders this particular approach unusable. This hypothesis was experimentally confirmed once we have implemented Lawler's algorithms and executed preliminary tests.

7.1.3 Tree-Based

While the previous two approaches used full SPF, the *tree-based* approach uses the shortest path feasibility subroutine only partially. Here only the lower bound Λ is maintained, initially small enough to guarantee that all edges have positive weight under w_Λ. Λ is progressively increased throughout the algorithm in correctly chosen increments, until a cycle ζ is found such that $w_\Lambda(\zeta) = 0$. Apart from Λ we also maintain the shortest path tree $\mathbf{T}$, with respect to the current w_Λ. As we are working with the augmented graph we may initiate $\mathbf{T}$ to consist of the edges from s to every other vertex.

The increments of Λ must be chosen with care, otherwise minimal mean cycle could be missed. A safe strategy is to set new value of Λ to the smallest $\lambda \geq \Lambda$ such that there is a different $\mathbf{T}$ for w_λ. To this end we assign to every vertex u a *threshold*, the smallest λ that would force u to change parent. Finding smallest among all thresholds is facilitated by a priority queue.

Thus the known implementations mainly differ in the specific priority queue they use: there is a heap-based implementation due to Karp and Orlin [74] running in $\mathcal{O}(nm \log n)$ and a Fibonacci heap-based implementation due to Young, Tarjan and Orlin [112] running in $\mathcal{O}(nm + n^2 \log n)$ and further referenced as the YTO algorithm.

Again this approach is unsuitable for SIMD acceleration for several reasons. The usage of priority queues (either heaps or Fibonacci heaps) is particularly problematic and would most likely be implemented as a simple vector, with the minimum operation as parallel reduction. Much larger problem was found during experiments with sequential version demonstrating that there are simply too many iteration of the algorithm that must be carried out one after another.

7.1.4 Howard's Algorithm

Since it is Howard's algorithm that appears to be the one most suitable for parallelization, we will now provide its detailed description. First, we should stress that the algorithm works on strongly connected graphs only. There exist two approaches how to overcome this restriction. First, we can decompose the given graph to its strongly connected components and then process the graph one component at a time. In the sequential case we can use Tarjan's algorithm [105] based on the depth-first traversal procedure which outputs the list of all strongly connected component in $\mathcal{O}(n + m)$ time. Hence there is asymptoti-

Algorithm 28: CPU Howard's algorithm

```
Input  : A directed, strongly-connected graph $G = (V, E, w),\ w : E \to \mathbb{Q}$
Output: $\lambda \in \mathbb{Q} : \lambda = \mu^*(G)$
1  foreach $v \in V$ do
2      $\mathsf{val}_0(v),\ \pi(v) \leftarrow 0,\ nil$
3  $\mathsf{improved} \leftarrow$ true
4  $\mathsf{i} \leftarrow 0$
5  $\lambda \leftarrow 0$
6  while improved do
7      $\mathsf{improved} \leftarrow$ false
8      foreach $v \in V$ do
9          $\mathsf{val}_{((\mathsf{i}+1) \mod 2)}(v) \leftarrow \min_{u \in \mathtt{succ}(v)} \{\mathsf{val}_{(\mathsf{i} \mod 2)}(u) + w(v,u) - \lambda\}$
10         if $\pi(v) = nil \vee (\mathsf{val}_{(\mathsf{i} \mod 2)}(\pi(v)) + w(v,\pi(v)) - \lambda > \mathsf{val}_{((\mathsf{i}+1) \mod 2)}(v))$ then
11             $\pi(v) \leftarrow u | \mathsf{val}_{(\mathsf{i} \mod 2)}(u) + w(v,u) - \lambda = \mathsf{val}_{((\mathsf{i}+1) \mod 2)}(v)$
12             $\mathsf{improved} \leftarrow$ true
13     $\mathsf{i} \leftarrow \mathsf{i} + 1$
14     if improved then
15         $\mathsf{c} \leftarrow$ MinMeanWeightCycle($G_\pi$)        // $G_\pi.|E| = |V|, \forall v \in V : deg(v) = 1$
16         $\lambda \leftarrow \mu_{G(\mathsf{c})}$
17         break all other cycles than c in $G_\pi$ so that all vertices have path to c
18         $\mathsf{s} \leftarrow$ MinVertex(c)
19         $\mathsf{val}_{(\mathsf{i} \mod 2)}(\mathsf{s}) \leftarrow 0$
20         $q.push(\mathsf{s})$
21     while $\neg q.empty$ do
22         $\mathsf{v} \leftarrow q.pop$
23         foreach $u \in$ pred(v) do
24             if $u \neq s \wedge \pi(u) = \mathsf{v}$ then
25                 $\mathsf{val}_{(\mathsf{i} \mod 2)}(u) \leftarrow \mathsf{val}_{(\mathsf{i} \mod 2)}(\mathsf{v}) + w(u,\mathsf{v}) - \lambda$
26                 $q.push(u)$
27 return $\lambda$
```

cally no difference in complexity of the algorithm, although practically the difference can be quite substantial. The second approach suggests to modify the underlying graph by adding a Hamiltonian cycle $\zeta_H =< (v_0, v_1), \dots (v_{n-1}, v_0) >$ to the graph. With this modification the graph becomes strongly connected and provided that weights of newly added edges are sufficiently large, the optimal cycle mean of the graph will remain unchanged.

As stated in the description of the cycle-based approach, Howard's algorithm (see Algorithm 28, adopted from [41]) extends the shortest path feasibility algorithm. Indeed the main cycle on lines 6–26 up to line 13 is in fact the SPF algorithm. Yet the output of the cycle on lines 8–12 is not the shortest path tree, since we are approaching the optimal cycle mean from top and hence there are cycles in our shortest path graph. To remain consistent with the established notation we will call the graph induced on edges $(v, \pi(v))$ the *policy graph*.

Apart from the successor in the policy graph $\pi(v)$ that must exist since we are working with strongly-connected graphs only, we also store two values $\mathsf{val}_0(v)$ and $\mathsf{val}_1(v)$ with every vertex. In these two values we keep information about the current and the following parametric length for a given vertex. In every iteration we check whether there was a change in the policy graph, and if not we interrupt the main cycle and return λ as the optimal cycle mean. Each λ that is a parameter for the feasibility computation, is actually the mean weight of a specific cycle in both the original and the policy graph. Hence, after every iteration of the SPF algorithm part we inspect the policy graph, locate all cycles inside it and choose the one with minimal mean weight (line 15). Upon finding the minimal cycle (or after choosing one of the minimal cycles), we modify the policy graph in such a way that every vertex has a path to the minimal cycle (line 17).

Lines 15 and 17 would perhaps require more detailed explanation. From the property of the policy graph (every vertex has exactly one outgoing edge), we know that each of its connected components consists of one cycle and potentially several paths leading to this cycle. Finding all cycles can thus be done in linear time simply by following the successor path, marking all visited vertices. Next we need to rebuild the policy graph so that the selected cycle would be the only cycle there and every vertex has a path to that cycle. Moreover, it is required for the component of the minimal cycle to remain unchanged, otherwise the SPF subprocedure would always detect an improvement. This can be achieved by two consecutive backward reachabilities: one to demarcate the component of the minimal cycle and the other one to connect also the remaining vertices to the minimal cycle.

Subsequently, we choose one vertex (line 18) on the minimal cycle and set its val to zero. It is necessary that we always select the same vertex, assuming the same cycle is found minimal. Consequently, after the modification of the policy graph and selecting new λ, it is also necessary to modify the val values for other vertices accordingly. This process is started by setting the val of s to zero on line 19 and carried out by the cycle on lines 21–26, performing backward reachability from s along the edges of the policy graph.

7.2 Data-Parallel Version of Howard's Algorithm

The actual description of our data-parallel implementation of Howard's algorithm will be conducted in several steps. We start by proposing a high-level work-flow, where we attempt to preserve the provably correct layout. Concurrently proposing graph primitive operations that would perform actions functionally equivalent to those of the original algorithm, but, wherever possible, addressing the whole vector of values at a time. CUDA-specific implementation of these graph primitives will be detailed extensively in the following section. Finally, we propose an extension to Howard's algorithm which prepends a parallel decomposition to strongly connected components to the algorithm. Then we let the algorithm perform the OCM computation on all components concurrently.

For the sake of simplicity we do not list in the following pseudo-code the initial transfer of the representation to GPU. All procedures that are called transfer only a bit indicating whether a fixpoint or a termination condition has been reached (see Section 3.2).

7.2.1 High-Level Description

The proposed host code of our implementation is listed as Algorithm 29. It is apparent that lines 15 and 17 of Algorithm 28 that rebuild the policy graph, require much more attention

Algorithm 29: GPU Howard's algorithm – host code

```
Input  : A directed, strongly-connected graph $G = (V, E, w), w : E \to \mathbb{Q}$
Output: $\lambda \in \mathbb{Q} : \lambda = \mu^*(G)$
1  while true do
2      terminate ← SPFPassIter($G, gVal, \lambda, G_\pi$, it)
3      if terminate = true then
4          break
5      it ← it + 1
6      GpiPreprocess($G_\pi, gPredInfo$)
7      Elimination*($G_\pi, gPredInfo$)
8      CycleIndetification($G_\pi, gCycles$)
9      Reduce_min($gCycles$, minCycle)
10     $\lambda$ ← minCycle.$mean$
11     SetMinCycle($G_\pi, gCycles$, minCycle)
12     MarkMinComponent*($G_\pi, gPredInfo$)
13     ConnectGpi*($G, G_\pi$)
14     $gVal$[it&1][minCycle.$minIndex$] ← 0
15     GpiPreprocess($G_\pi, gPredInfo$)
16     ValuePropagate*($G^T, G_\pi, gVal$[it&1], $\lambda$, minCycle.$minIndex, gPredInfo$)
17 return
```

in the SIMD environment as it is the place most susceptible to inefficient processing. These two lines of CPU pseudo-code span from line 6 to line 13 in our GPU implementation. We first describe this part of the algorithm postponing the SPF subroutine for later discussion.

There are two calls to the GpiPreprocess kernel (on lines 6 and 15) and they both serve the same purpose to gather information about predecessors in the policy graph. This step is merely an optimization speeding up the kernels that perform the backward reachability (or its modification). It would be possible to omit this kernel for the same reasons as it is possible to perform backward reachability using only forward edges (see Section 3.2.1). Yet the speedup gained from employing this kernel is quite considerable even though we have to call it twice as the graph is rebuild in the ConnectGpi kernel. The first call is because of the Elimination and the MarkMinComponent kernel, the second call is because of the ValuePropagate kernel.

From the description of sequential Howard's algorithm we know that the policy graph consists of *weakly connected* components, each containing a cycle and several paths leading to this cycle. In order to be able to find all cycles in the policy graph in as few parallel steps as possible, we first apply Elimination to remove all vertices that do not lie on a cycle. This kernel is called iteratively (every call removes the vertices with no predecessors) until a fixpoint is found, i.e. until there are no such vertices. There are more fixpoint kernels in Algorithm 29 all marked with an asterisk. The elimination allows localization of cycles and computation of their means in a straightforward manner (line 8). The minimal among them can subsequently be found by employing parallel reduction [42] with `min` operation.

Upon finding the cycle with minimal mean (which has to be agreed on by all vertices on line 11) we can start rebuilding the graph. With the first backward reachability (line 12) we undo the elimination of vertices within the component of the minimal cycle, hence the second backward reachability (line 13) is started from this component and is iteratively applied until all vertices are connected, one breadth-first search layer at a time.

Finally, the SPF subprocedure is easy to parallelize. Using two val vectors (denoted as $gVal$) allows us to perform all updates of values in a single parallel step (as there is no danger of race between threads, see line 2). Also realization of the VALUEPROPAGATE kernel on line 16 that propagates the change of val from the vertex on minimal cycle with the smallest index (*source*), was only a minor modification of backward reachability procedure.

7.2.2 Graph Primitives and Data Structures

Most of known OCM algorithms require to access predecessors of a given vertex in order to perform a kind of backward reachability. Storing backward edges together with their forward versions causes additional nontrivial memory requirements, which might be a problem as the size of CUDA memory is limited. We have shown how to carry out the backward reachability using only forward edges with time overhead in Section 3.2.1. However, OCM algorithms often require the reachability procedure to perform only on a given subgraph of the whole graph. Unfortunately, augmenting the original procedure in order to follow only selected edges resulted in considerable slow down in computation. For that reason we were forced to explicitly store both forward and backward edges of the graph.

As mentioned before the amount of memory on a CUDA device may limit application of CUDA accelerated algorithms to graphs of which representation fits the memory of the GPU. Multiple CUDA-aware GPUs can be used to effectively extend available memory (see 5.2) for the price of extensive modification of the source code and a certain slow down. Fortunately, the memory limitation is not that restricting in the case of OCM algorithms as the high asymptotic complexity of individual algorithms results in practical issues dealing with long runtimes rather than a lack of memory space.

While most of our kernels are only minor modifications of data-parallel graph primitives presented in the previous chapters, they are crucial for the overall efficiency of our data-parallel implementation, and therefore, we describe some of those modifications in detail. Prior to that we first focus on the data structures that are used during the computation. The graph representation itself has been described in Section 3.1. In order to keep low space profile, we store the policy graph G_π (denoted as gA_π in the VALUEPROPAGATE kernel) in a vector of n elements containing indices of Array A_e that uniquely specifies what edge leads to the successor of a given vertex. Also the first few bits of every element are reserved to flags. For example there is a flag marking what vertices have been removed during elimination. Cycles are stored in $gCycles$ using two 32-bit values, one for the index of its source vertex and second for the mean weight. Finally, the vector $gPredInfo$ is used to store a partial information about the predecessors within one 32-bit value. The first 16-bits are for the number of predecessors and the last 16-bits for the *local* index of the first predecessor.

GPIPREPROCESS primitive was devised for a simple reason: there is no efficient way to propagate along backward edges in the policy graph. It is approximately equally inefficient to use forward edges (since we have to deploy as many threads as there are unseen vertices) as to search among backward edges those in the policy graph (since there are often many edges than do not belong to the policy graph). The improvement we have proposed first passes the whole graph G storing correctly the information about predecessors into $gPredInfo$. This preprocessing allows to virtually skip inspection of vertices with no prede-

Algorithm 30: GPU VALUEPROPAGATE kernel – device code (run in parallel $\forall v \in V$)

```
Input : $gTransA_e,\ gTransA_i,\ gA_\pi,\ gVal,\ \lambda,$ source, $gPredInfo$
prop ← false
counter ← $gPredInfo[v]$.getNum()
pred ← $gTransA_i[v + gPredInfo[v]$.getFirst()$]$
while counter $> 0$ do
    edge ← $gTransA_e$[pred]
    if $v = gTransA_e[gA_\pi$[edge.to]$]$ then
        counter ← counter $- 1$
        if edge.to $\neq$ source then
            $gVal$[edge.to] ← $gVal[v]$ + edge.weight $- \lambda$
            prop ← true
    pred ← pred $+ 1$
if prop then
    fixPoint ← false
```

cessors in the policy graph and also jump at the first edge that belongs to the policy graph as can be observed from the pseudo-code of VALUEPROPAGATE in Algorithm 30. There we store in the local variable $counter$ the number of predecessor of the vertex assigned to this thread (line 2) and can skip the cycle if $counter$ is zero. Together with the jump to the first actual predecessor (see line 3) this improvement alone has led to five-fold speedup of VALUEPROPAGATE including the cost of preprocessing.

Details on remaining kernels are as follows. The ELIMINATION kernel is actually the TRIMMING kernel (see Algorithms 21) augmented similarly as the VALUEPROPAGATE kernel with the information about predecessors. A simple while loop (it is executed only on vertices on some cycle) for identification of cycle source and its mean is implemented in the CYCLEIDENTIFICATION kernel. And finally the CONNECTGPI kernel performs backward reachability from the component of the minimal cycle, and it utilizes a flag $propagate$ which marks the currently active breath-first layer and thus less threads needs to be dispatched.

7.2.3 SCC Decomposition Extension

There are several reasons why to prepend SCC decomposition before a CUDA accelerated OCM algorithm. First of all, the algorithm requires the input graph to be strongly connected and thus we have to add the Hamiltonian cycle. Not only is this operation costly, it also adds more edges into the graph, further prolonging the computation. Furthermore, even though the parallel algorithms for SCC decomposition have rather high asymptotic complexity ($\mathcal{O}(n(n+m))$), we were able to implement data-parallel SCC decomposition which is considerably faster than the optimal sequential algorithm (see Chapter 6). And most importantly, it allows the computation to be executed concurrently on all SCC components, which further improves the running time provided that the components are much smaller than the whole graph.

The technique to run a kernel on multiple *regions* within a graph and to restrict its effect to respective regions was thoroughly described in Section 6.4 and we will thus concentrate

Algorithm 31: CPU Region-specific minimum voting

```
myCycle ← (source, mean = (weight/length))
while true do
    comCycle ← $cycles$[ myRegion]
    if comCycle.mean ≤ myCycle.mean then break
    atomicCAS(&($cycles$[ myRegion]), comCycle, myCycle)
```

on the parts related to computation over decomposed graphs specific for our OCM algorithm. First of all the Proposition 7.1.2 states that the optimal cycle mean of the whole graph needs to be found as the minimal among all components. This observation raises two problems, first, how to choose the minimum, and second, where to store the component-specific λ values during the computation (which also has to be agreed on).

Selecting the minimal cycle mean during the computation on multiple regions is particularly problematic as the vertices of one component are not clustered together. It would actually require first to *split* [97] the vector according to the region identification and then perform *segmented* reduction [42], both complicated and expensive operations. Fortunately, we have observed that often very few cycles are found and consequently only a few values are candidates for the minimum. Thus we were able to use the `atomicCAS` operation as shown in Algorithm 31.

Also the termination needs to be modified to work on two levels. The global termination occurs when computation is finished on all strongly connected components. But it is also important to prevent execution of kernels on components where we have already found the OCM. Since then less threads needs to be deployed and less candidates compete in the minimum voting. For that purpose we have inserted a kernel that unsets the flag *work* for all vertices in inactive components between lines 2 and 5 of Algorithm 29. Finally, we have optimized the overall amount of work by unsetting the *work* flag of all single vertex components prior to the actual OCM computation.

7.3 Experimental Evaluation

Since the prime objective of our research was to accelerate performance analysis based on computation of optimal cycle mean, we have compared sequential and data-parallel algorithms mainly on models of communicating distributed systems. The state space of all possible configuration of a given system forms a graph with cost function (representing for example resource consumption) labeling edges of that graph. Both modeling of the system and generation of its state space was facilitated by the enumerative model checker DiVinE [22], extended with the capability to analyze performance of input models.

The sequential OCM Algorithms HOWARD'S and YTO were also implemented within the DiVinE tool. Thus the evaluation of our CUDA accelerated HOWARD'S algorithms is conducted by comparing its running time against these two sequential algorithms (which are often considered to be the fastest [50]). The graph representation, while primarily targeting the vector processing architecture of GPU, is also particularly suitable for CPU due to its cache-efficient characteristics. Hence we do not take the time of graph construction into account for the evaluation.

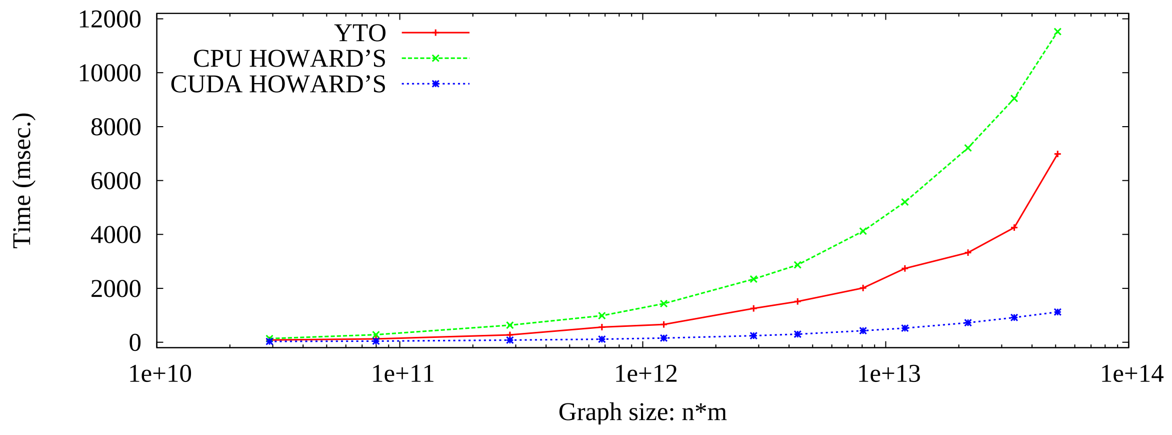


Figure 7.1: *Plot for the server-free system.*

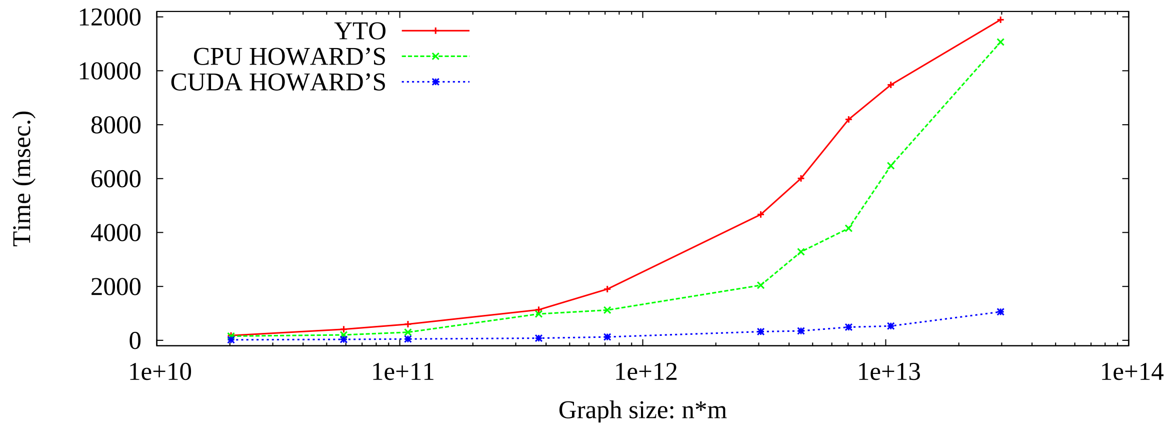


Figure 7.2: *Plot for the system with a server.*

All experiments were executed on a CUDA-equipped Linux workstation with an AMD Phenom II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066MHz RAM and NVIDIA GeForce GTX 480 GPU with 1.5 GB of GDDR5 memory.

Our approach to performance analysis allows us to compute quantitative characteristics of distributed systems where clients comply to a distinct sets of behavioral patterns (scenarios of expected behavior). Furthermore, we can state how many clients of that particular scenario appear in the system and thus we can estimate the load of a part of the system (a server for example) in an execution based solely on the information of how many clients of what scenario there are.

Using this formalism we have constructed two distributed client/server system *templates* from which a user can generate system models by deciding on the number of client for each scenario. In the first system, there is no server present and the actions of clients are left to be interleaved nondeterministically. The second system contains a server and the clients have to communicate with the server (and only one client can access the server at a time). The OCM of such systems then represents the average system load inflicted on the environment.

We have performed various tests on both templates to measure scalability of all the algorithms and plotted the findings in two figures. In Figure 7.1 there are the results for system without a server and in Figure 7.2 the results for system with a server. The x axis is in logarithmic scale and represents the size of the graph in $n * m$ as all the OCM algorithms are asymptotically in $\mathcal{O}(nm)$ or worse. The plots show very clearly that while both the YTO and sequential HOWARD'S algorithm struggle with preserving their running times as the graphs grow bigger, our CUDA accelerated implementation is capable of performing the OCM computation in reasonably small time. On larger instances the GPU rarely fails to provide a five-fold speedup compared to the better of the two sequential algorithms. It is worth noting that while the CPU algorithms were deterministic and in different runs exhibited indistinguishable behavior, the GPU implementation behaves nondeterministically due to the nature of the massive parallelism. For that reason we executed every test ten times and the result displayed in the plots is the median of all trials.

Although primarily targeting acceleration of OCM computation for performance analysis we feel obliged to admit that on graphs from other applications was our data-parallel implementation much less successful. We have conducted several experiments with US traffic network graphs, random graphs and various graphs from the DIMACS challenge and were never able to outperform the YTO algorithm. On these graphs the YTO algorithm performed only a very few iterations which we attach to the fact that the OCM of these algorithms was often very close to the minimal edge weight.

7.4 Conclusion

In this chapter, we proposed data-parallel acceleration of an OCM algorithm within several consecutive steps. First we have evaluated all existing classes of OCM algorithms with respect to their predisposition for vector processing. Subsequently, we described thoroughly HOWARD'S algorithm which was found most appropriate and devised its data-parallel version. Specifics of the implementation together with selected data-parallel graph primitives were then detailed, e.g. the incorporation of the SCC decomposition and the concurrent execution of the OCM algorithm on all strongly connected components.

The primary motivation behind the proposed data-parallel OCM algorithms was the acceleration of performance analysis of distributed communication systems. That we have evaluated experimentally by constructing two scalable client/server systems based on distinct scenarios of the clients finding our data-parallel algorithm capable of providing performance analysis in negligible time. Although competitiveness of our CUDA accelerated HOWARD'S algorithm on other types of graphs is questionable, we have reported a steady five-fold speedup on performance analysis against all other algorithms.

In our future work we plan (similarly as in the case of the accepting cycle detection and the SCC decomposition) to employ the recently proposed methods such as the warp-centric programming [69] and the linear parallelization [91] (see Chapter 3 for more details) in order to further improve the performance of our data-parallel algorithm for OCM problem.

Chapter 8

Parallel Algorithms for Resolution of Boolean Equation Systems

In this chapter, we design a data-parallel algorithm for the resolution of Boolean Equation Systems (BESs). Our goals are (i) to evaluate the scalability of our parallel approach with respect to an increasing number of parallel *processing units* (PUs), and (ii) to prove its competitiveness in comparison with the optimal sequential algorithm, which we implemented analog to the description in [3].

As shown in [86] and briefly described in Section 2.3.2 model checking of alternation-free μ-calculus [76] can be encoded by a BES, where the solution of the BES is equivalent to the solution of the underlying model checking problem. This BES is obtained by combining the given *labeled transition system* (LTS) and the inspected property. The model checking technique based on the resolution of BES also suffers from the so-called *state space explosion problem* that makes the size of the corresponding BES to be exceedingly large for real-life industrial systems. As a result, the applicability of this technique to systems built in practice is rather limited. Therefore a lot of research has focused on the development of methods that can reduce the size of the resulting BES. Similarly as in the case of LTL model checking these methods are based on partial order reduction [98, 34], symbolic representation of the state space [90] or limiting the exploration of the state space only to relevant traces using on-the-fly generation [104].

In contrast to this, our approach does not aim at reducing the problem size, but exploits modern SIMD architectures in oder to speed up the model checking procedure. In the context of this chapter we want to restrict ourselves to algorithms for parallel resolution of BESs. However, in the scope of the model checking procedure the construction of BES from the given model and the inspected formula also plays significant role. In the model checking process the model (represented as LTS) is given implicitly (by functions to enumerate initial state and transitions emanating from a given state) or explicitly (by an adjacency list representation of the LTS). The construction of the BES is based on traversal of the corresponding LTS and on interpretation of the formula over all visited states [86]. In addition to the BES construction we also have to transform the BES into a form suitable for parallel processing. There exist several efficient approaches for parallel traversal of transition systems and for construction of the compact data representations in context of model checking. See Section 5.1 and [12, 79] for more details. These methods can be easily modified to build the efficient representation of BES and thus we concentrate only on designing data-parallel algorithms for resolution of BESs.

Besides the parity game-based approach presented in [107], which performs a parallel resolution of μ-formulae on shared-memory CPU-based systems, we are aware of only distributed implementations of the resolution. The aim of these distributed approaches is to increase the total amount of memory, rather than to increase the performance, as the network latency degrades the overall performance significantly. One algorithm for checking the full μ-calculus, based on the distributed evaluation of sub-formulae, is proposed in [61]. In [30] and [84] two algorithms employing distributed game graphs are presented that perform a parallel coloring of the graph in order to solve the underlying model checking problem. Another implementation and experimental evaluation was carried out in [66] using message passing on a *network of workstations*.

Only two other approaches, focusing on the parallel resolution of BESs, are known to us. The first one, described in [72], is tailored to distributed systems aiming at the resolution of extremely large BES instances rather than improving runtime performance. The second one [102] employing *Gaussian Elimination* as proposed in [86], turned out not to be viable in practice, due to its exponential space complexity.

We, however, present a parallel, shared-memory approach to model checking of alternation-free μ-calculus, by employing the parallel resolution of BESs. This is explicitly targeted at large state spaces in order to exploit the power of parallel, throughput oriented architectures. We primarily aim at the well-established VLTS benchmark suite [110], which provides 40 LTSs originating from academia and industry that can be checked for deadlocks and livelocks. For the sake of completeness we want to mention two sequential tools, namely *evaluator*, which we used to generate the BESs from the benchmark's LTSs and *bes_solve* to verify our results, both from the CADP toolset [87].

As our experiments show, we can confirm the scalability results presented in [107] for the multi-core implementation – but on a much larger benchmark suite. Furthermore, our data-parallel algorithm also scales up to many-core architectures and outperforms the optimal sequential algorithm on most examples from the benchmark suite by almost an order of magnitude.

8.1 Algorithms for Resolution of Boolean Equation Systems

There exist several approaches for resolution of BESs. They are using methods similar to *Gaussian Elimination* [86], on-the-fly techniques such as *chasing ones* [3], which we used as the sequential baseline in our experimental evaluation, or simply a fixpoint iteration. See [75] for a comprehensive summary on the topic. In order to be able to conduct a fair evaluation of our parallel implementations in terms of competitiveness, we have implemented an optimal, sequential, CPU-based algorithm, in the style of chasing ones as proposed in [3] (further referenced as the CHASING-ONE algorithm). The internal representation of BESs is the same as in our CUDA accelerated implementation (see Section 8.4). The high-level pseudo-code is listed as Algorithm 32.

As we briefly described in Section 2.3.2 each fixpoint operator of the inspected μ-formula is represented by a so-called *block* in the resulting BES, containing the set of equations connected to this operator. The order in which the individual blocks of the BESs have to be processed is defined by their nesting within the formula. Since we consider only alternation-free μ-formulae the dependencies within the blocks form a tree [76, 86]. This tree can be easily constructed and gives us the proper order (from the leaves to the root) in which the individual blocks have to be solved. For the sake of simplicity we assume that the outer for loop on line 1 processes the blocks in this ordering.

The resolution of block B starts at those equations evaluating to, or being directly assigned a terminal value (i.e., *true* or *false*). The corresponding LHS variables that have the terminal values are pushed into the queue (via the INIT procedure on line 2). The algorithm further propagates this information within the block B. It repeatedly takes a variable from

Algorithm 32: CPU CHASING-ONE algorithm

```
Input  : BES
Output: Resolution of BES
1  foreach block $B$ do                // blocks are processed in the right order
2      INIT(queue)
3      while queue is not empty do
4          LHSVariable $\leftarrow$ queue.$pop()$
5          foreach equation $E$ : LHSVariable $\in$ RHS$(E)$ do
6              if $E \in B$ then
7                  EVAL$(E)$
8                  if LHS$(E)$ is changed then
9                      queue.$push$(LHS$(E)$)
10             else
11                 POSTPONE$(E)$
```

the queue and evaluates each equation E whose *right hand side* (RHS) contains this variable and belongs to the block B. If the equation does not belong to this block its evaluation is postponed until the corresponding block is processed (via the POSTPONE procedure). If the value of the equation E has changed the corresponding LHS variable is pushed into the queue. The resolution of block B terminates (the while loop on lines 3–11) if the queue is empty and thus no further propagation can be done. The whole algorithm finishes if the block corresponding to the root of the tree is resolved.

In order to efficiently identify which equations have to be evaluated in each iteration also the backward dependencies has to be stored. Note that our data-parallel algorithm based on the fixpoint iteration needs to store only the forward dependencies.

The space and time complexity of the CHASING-ONE algorithm is linear with respect to the size of BES [3]. As the complexity is optimal for this problem the algorithm is well suited to be a baseline for comparison with our data-parallel algorithm.

8.2 Parallelization Strategy

While a lot of effort has been invested into the development and optimization of sequential model checking algorithms, in order to fight the computational complexity caused by the state space explosion, our aim was to investigate whether a parallel approach can effectively scale up to massively parallel hardware architectures. In order to choose a suitable parallelization strategy for our problem we explore structural properties and dependencies of the BESs from the benchmark suite. The structure and data dependencies within the resulting BES are closely related to the structure of the LTS it was generated from. The average branching factor, i.e., the average number of outgoing edges per vertex, over all 40 LTSs in the benchmark is 5.73. With respect to parallelization, this number can be interpreted as the upper bound for potential parallelism that is given by an LTS, as in our setting information needs to be propagated along the edges. For work set-based producer-consumer parallelizations this means that (i) for each processed work item only few new work items are expected

Algorithm 33: CPU FIXPOINT algorithm

```
Input  : block $B$ of BES
Output: Resolution of block $B$
1 do
2     variablesChanged ← false()
3     for equation $E \in B$ in parallel do        // equation order does not matter
4         EVAL($E$)
5         if LHS($E$) is changed then
6             variablesChanged ← true()
7 while variablesChanged
```

to be added, and (ii) synchronization is needed in order to maintain consistency of the dynamic data structure used to store the work items.

Due to this, our approach is not based on the producer-consumer paradigm (required by the CHASING-ONE algorithm), but on a fixpoint iteration. This promises a much higher potential for the utilization of parallel hardware as it does not rely on dynamic data structures. In our particular setting the operations on the data structure can even be implemented lock-free, as the fixpoint iteration is used to solve a monotonic function. Furthermore, we do not have to populate a work set, as we propagate all possible changes during an iteration, for the price of some computational overhead, which seems negligible considering the ever growing number of parallel processing units.

The fixpoint iteration is suitable for fine-grained parallelism where a dedicated thread is executed for every equation of the BESs. This approach allows us to design data-parallel algorithm for resolution of BESs that can efficiently utilize modern SIMD architectures.

8.3 Data-Parallel Fixpoint Computation

In this section we first present the algorithmic background of our approach based on the fixpoint iteration, followed by the concepts for our multi-core and many-core implementations.

As we describe in Section 8.1 the individual blocks of the BES have to be resolved in the order given by the tree that corresponds to the structure of the inspected formula. We start with the blocks that form the leaves of the tree and follow a *bottom-up* approach. It means that once all leaves are resolved we remove them and continue with the next level of the tree. In the context of parallelization the shape of the tree plays an important role since all blocks that form one (the lowest) level of the tree can be resolved in parallel (there are no dependencies within the blocks on the same level). The efficient utilization of this property is rather straightforward (we only need to build the tree and identify the individual levels) and thus we further discuss only the parallel resolution of a single block.

The listing of Algorithm 33 illustrates the basic idea of the data-parallel fixpoint computation for a single block (further referenced as the FIXPOINT algorithm). It consists of two nested loops. The purpose of the outer one (on line 1) is to identify the fixpoint of the resolution of the block. The inner for loop (on line 3) computes the value of the LHS for each equation within the block according to the evaluation of its respective RHS. In the beginning, all LHSs are initialized, depending on the fixpoint operator, as *false* in the case $\sigma = \mu$

and *true* in the case $\sigma = \nu$. This is *initial approximation* derived from the Knaster-Tarski fixpoint theorem [106], where $\mu f = \bigsqcup\{f^i(false); i \in \mathbb{N}\}$ and $\nu f = \bigsqcap\{f^i(true); i \in \mathbb{N}\}$. The termination of the fixpoint computation is detected on line 7 by a marker variable, indicating whether one or more LHS variables have changed during an iteration.

The sequential time complexity of Algorithm 33 is quadratic with respect to the size of the block since in the worst case only one LHS is changed in each iteration of the main loop (the number of iterations is equal to the number of equations) and each iteration performs linear amount of work. Thus Algorithm 33 provides *the quadratic parallelization* for the resolution of BESs.

The core idea for parallelization of the basic fixpoint algorithm is the parallel execution of the inner loop (on line 3) of Algorithm 33, computing the LHS value of an equation, as the order in which equations are processed does not matter. Considering the fact that this operation has to be executed for all equations and during each iteration step, our approach exposes much potential for parallel computation, even within one iteration step, as we expect the number of equations to be very large. The soundness of this approach is guaranteed by the fact that BESs represent monotonic functions, i.e., even if the computation of an LHS depends on several other LHS variables – which in a parallel setting are potentially modified concurrently – the updated value of each LHS is available and thus can be propagated in the subsequent iteration.

8.3.1 Multi-Core Implementation of Data-Parallel Fixpoint Computation

Before we show how to employ the data-parallel fixpoint computation in order to accelerate the resolution of BESs on the SIMD architectures offered by modern many-core GPUs, we focus on multi-core CPU architecture. Our goal is to explore the scalability of multi-core CPU implementation of Algorithm 33 and motivate the utilization of the parallel processing in the context of the resolution of BESs.

To design an efficient multi-core implementation of data-parallel algorithms the choice of the framework is very important, as it most significantly influences the overhead connected to context switches. The overhead of manual thread maintenance is not negligible, since the amount of productive work per thread invocation is very limited. Therefore, the direct use of multi-threading environments, such as PThreads [95], is very likely to nullify the gain we expect from the parallelization itself. For this reason we chose Intel's Cilk Plus framework [45], which offers a work stealing-based thread pool and internally employs efficient scheduling and load balancing mechanisms. The scheduling of workers is not explicit and more lightweight than context switches using threads. It provides three simple yet powerful extensions for parallelization to the C and C++ programming languages, namely: *cilk_spawn*, *cilk_sync* and *cilk_for*.

We chose Cilk Plus framework because it is well suited for problems with fine grained data-parallelism and irregular structure [56], as is the case in our setting. This stems from its work stealing approach which utilizes a pool of workers, each of which is mapped to a thread during execution. This is in contrast to having to create, manage and delete threads manually, thereby inducing a much higher overhead.

Data structures for multi-core systems have to follow two main objectives. On one hand they have to provide good data locality, i.e., data necessary for a computation should be closely grouped so it can, ideally, be stored in the same cache line of a CPU. On the other hand, unrelated data should be separated in such a way that it does not interfere with each

BES

$$\begin{aligned} x_0 &= x_1 \wedge x_3 \\ x_1 &= x_3 \vee x_2 \\ x_2 &= \texttt{false} \\ x_3 &= x_4 \wedge x_1 \\ x_4 &= x_1 \vee x_2 \end{aligned} \longrightarrow$$

Adjacency Matrix

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

CSR Format

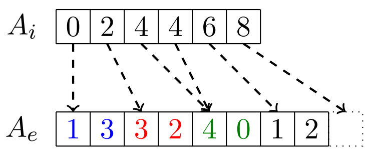


Figure 8.1: *Generation of adjacency list representation from BES.*

other in order to avoid harmful effects, such as cache thrashing. Due to these two factors and the structure of our input data (variables $\in$ equation(s) $\in$ block(s) $\in$ BES) we have decided to use a nested data structure, where each aforementioned component is modeled by a structured type. In this layout all data needed to evaluate one equation – the most frequent operation in our algorithm – is stored in a single structure resembling an equation, thus, accounting for good data locality. Clearly, this also provides good separation and to improve this would require machine dependent optimization.

The key idea of our CPU-based implementation is based on the parallelization of the inner for loop of the basic fixpoint algorithm by employing Cilk Plus' parallel for-loop version, *cilk_for*. The reasons why we do not have to explicitly lock or modify the computations are (i) the monotonicity of the Boolean function as mentioned before and (ii) the fact that the variable indicating the change of LHS variables is set and reset outside the parallel loop (line 2 of Algorithm 33) and modified uniformly (only set to *true*) inside the parallel loop (line 6 of Algorithm 33), and thus cannot cause any inconsistency.

8.4 CUDA Accelerated Resolution of BESs

In this section we design CUDA accelerated resolution of BESs based on the proposed data-parallel fixpoint computation. As we discussed in the previous chapters data structures used for CUDA accelerated computation must be designed specifically for this purpose. They must support independent thread-local data processing, and, at the same time, they must also be compact enough to enable good data locality. This is to avoid high latency device-memory access and to generally reduce the usage of device-memory bandwidth, which might otherwise become a performance bottleneck [82].

A key observation is that a BES can also be interpreted as a directed graph where the LHS variables form the vertices and the dependencies on the RHSs represent the edges. In Section 3.1 we show how the adjacency matrix representation of directed graphs can be encoded using two arrays (A_i and A_e) in *compressed sparse row* (CSR) format (see Figure 8.1). This data structure has been demonstrated to be efficient for graph algorithms in the context of CUDA accelerated computation [21, 9, 62]. In our case, each vertex is stored in Array A_i and keeps the following information: an index to Array A_e (where the target vertex of the first emanating edge is stored), its Boolean value, a flag indicating whether the value is already computed, and the *type* of operator (conjunction or disjunction). Since the GPU memory is limited we store these information into unused pointers bits (the values of the vertices are technically pointers) reducing thus the total space requirements to 4 bytes per vertex.

The general work-flow of our CUDA accelerated fixpoint computation builds on the same principles as CUDA accelerated graph algorithms described in Section 3.2. It combines

Algorithm 34: GPU Fixpoint algorithm – host code

```
Input  : block $B$ of BES
Output: Resolution of block $B$
1  CreateRepresentation($B$, $A_e$, $A_i$)
2  fixPointFound $\leftarrow$ false
3  CopyToGPU(($A_e$, $A_i$) $\rightarrow$ ($gA_e$, $gA_i$))
4  while $\neg$fixPointFound do
5      fixPointFound $\leftarrow$ true
6      FixpointKernel($gA_e$, $gA_i$, fixPointFound)
7  CopyToCPU($gA_i \rightarrow A_i$)
```

Algorithm 35: GPU Fixpoint kernel – device code (run in parallel for every LHS variable)

```
Input  : $gA_e, gA_i,$ fixPointFound
1  tid $\leftarrow$ $blockId.x * blockDim.x + threadId.x$
2  myVertex $\leftarrow$ $gA_i$[ tid]
3  if myVertex.$solved$ then
4      return
5  first $\leftarrow$ myVertex.$index$
6  last $\leftarrow$ $gA_i$[ tid $+ 1$].$index$
7  foreach index $\in$ first, . . . , last do
8      targetVertex $\leftarrow$ $gA_e$[ index]
9      mySucc $\leftarrow$ $gA_i$[targetVertex]
10     if mySucc.$value \neq$ myVertex.$type$ then          // $type$ $\vee \equiv 0$ and $type$ $\wedge \equiv 1$
11         break
12 if myVertex.$value \neq$ mySucc.$value$ then
13     myVertex.$solved$ $\leftarrow$ true
14     myVertex.$value$ $\leftarrow$ mySucc.$value$
15     $gA_i$[ tid] $\leftarrow$ myVertex
16     fixPointFound $\leftarrow$ false
```

the out-of-order CPU and data-parallel processing GPU and allows for very fine granularity of parallelism [62], where a dedicated thread is executed for every equation within the block that is processed. Note that if more than one block is processed in parallel (see Section 8.3) then the dedicated thread is executed for every equation in all of these blocks.

The CPU host code (listed as Algorithm 34) first creates the representation of the BES suitable for CUDA computation and then runs the main while loop of Algorithm 33. It performs calls to CUDA kernel that are executed on the GPU. The kernel (listed as Algorithm 35) is responsible for evaluation of the RHSs of all equations. The kernel is invoked repeatedly as long as vertex values change.

Each thread first loads the vertex from Array A_i (stored in GPU global memory) into a local copy (line 2) and checks if the corresponding variable has already been solved (line 3).

Then, it processes all immediate successors (loop on line 7), representing the RHS of the corresponding equation. The algorithm employs a lazy evaluation of the equations. In the case that the value of the inspected RHS variable and equation type immediately determines the value of the equation (i.e. the RHS is a purely disjunctive term where at least one variable is *true* or the RHS is purely conjunctive and at least one variable is *false*), the loop is broken (line 11). Next, the kernel checks whether the evaluation of an RHS (stored in $mySucc.value$) changes the value of its LHS (line 12). In that case the value of the respective vertex is updated and written back to Array A_i (line 15), and the fixpoint flag is set to false indicating that the fixpoint is not yet reached.

We further design and implement two optimizations of the CUDA accelerated fixpoint computation. The first one, our so-called *intra-warp fixpoint iteration*, which is based on the observation that all threads within a warp have to load the required data from global memory into local copies. All operations are performed on the local copies, which are written back to global memory at the end of the execution of the warp. This means that updated LHSs do not become visible to other threads until the next iteration step and thus, changes can only be propagated one step per iteration. The intra-warp fixpoint iteration is intended to increase the number of propagations, by performing multiple iterations on the equations bundled in a warp and thereby propagating changes of LHSs within this warp.

The second optimization is an extension to the aforementioned intra-warp fixpoint iteration. It utilizes the GPU's shared memory, which provides a fast local memory for single thread or warp, allowing the intermediate storage of data. We use this shared memory to optimize the execution of the kernel by reading the LHSs contained in an RHS, from global to shared memory. When the data of an LHS is required by the kernel the copy in shared memory is utilized instead of the one in global memory. When the kernel returns, the copy is written back from shared to global memory. However, the assignment on line 9 potentially requires further LHSs, this data can either be read from global memory as before, or also be copied to shared memory. This reduces access to global memory, but requires additional load and store operations before and after each thread invocation.

8.5 Experimental Evaluation

As we already mentioned, we focus only on the parallelization of the algorithms for resolution of BESs. Therefore, we do not consider the construction of BES from the given model and the inspected μ-formula in our experimental evaluation.

Two main goals of our experiments are to evaluate i) the scalability of our approach and ii) the overall competitiveness of our approach as compared to the existing optimal sequential algorithm, that seems ill suited for parallelization. Furthermore, we evaluate structure and density of BESs generated from the benchmark suite and, in order to provide an outlook on the generality of our results, extend this evaluation with randomly generated BESs. Besides the runtime-based comparison we provide important insights to the specifics of BESs in the context of model checking and present heuristics that yield a significant speedup for this particular setting. As all experiments and results are based on and obtained by the evaluation of our BES resolution algorithms they are well suited for direct comparison.

In order to be able to conduct a fair evaluation of our approach in terms of competitiveness, we have implemented the sequential Chasing-One algorithm, as proposed in [3] using the identical representation of the BES as the one used for GPU computation. Since the

Property	μ-formula
No Deadlock	$\nu X.([-]X \land \langle - \rangle true)$
Livelock	$\mu X.(\langle - \rangle X \lor \nu Y.(\langle \tau \rangle Y))$

Table 8.1: *μ-formulae of inspected properties*

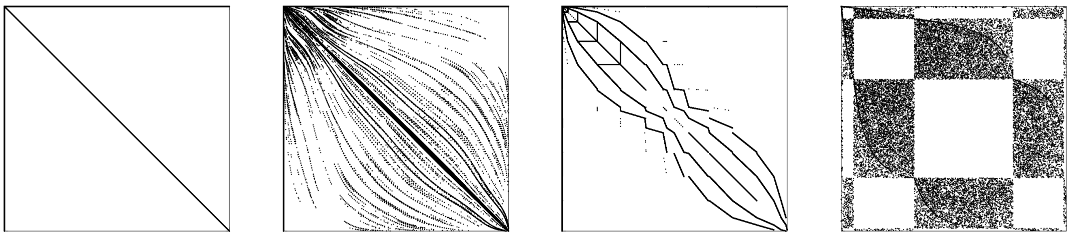

(a) Example 10 (b) Example 22 (c) Example 34 (d) Rnd2

Figure 8.2: *Visualization of benchmark examples as adjacency matrices.*

space and time complexity of this approach is optimal it seems to be a suitable candidate for comparison with our parallel GPU-based implementation.

Our experiments were conducted using the well established VLTS Benchmark Suite [110] compiled within a joint project of CWI and INRIA. It consists of 40 examples from academia and industry, given as labeled transition systems with numbers of states ranging from 289 up to 33,949,609.

The background of the benchmark examples varies greatly, thus, a variety of different properties could be checked for individual examples. For our evaluation we use two representative properties, namely no deadlock and livelock, that can be checked for all examples of the benchmark suite – see Table 8.1 for their formalization. Also, the results for those properties are provided by the authors of the benchmark, allowing a direct verification of the correctness of the results of our implementations.

The images of some exemplary BESs, as depicted in Figure 8.2, show the significant variance in structure and density of the LTSs provided in the benchmark. The images are visualizations of the adjacency matrices of the respective BESs, with the origin on the top left.

In contrast to intuition, our experiments show that this information about structure and density does not usefully correlate with the scalability and/or runtime performance of our approach. This is the case for the following reasons: (i) runtime depends on whether the property, which the LTS is checked for, is fulfilled or violated, (ii) the fact that our approach does not favor local propagation of changing variables, but globally propagates all possible changes during an iteration and (iii) the fact that our approach performs best in those cases that expose large numbers of concurrent changes – rather than sequential chains of changes. Unfortunately none of these factors can be estimated and extracted from a BES's structure.

Our experiments were carried out on different hardware platforms for (i) the CPU and (ii) the GPU version of the implementation: (i) Two interconnected Intel XEON E7-4830 Processors @ 2,13 GHz, each with 8 physical cores and Hyper-Threading enabled (i.e., a total of 32 logical PUs) and 64 GB DDR3 RAM @ 1333 MHz, running Windows 7 64-bit, and (ii) one AMD Phenom II X4 940 Processor @ 3,0 GHz, 8 GB DDR2 RAM @ 1066 MHz along with (a) one NVIDIA GeForce GTX 280 GPU with 1 GB of global memory, 16KB of shared memory per multiprocessor, providing 240 CUDA cores and (b) one NVIDIA GeForce GTX 480 GPU with 1.5 GB of global memory, 48KB of shared memory per multiprocessor, providing 480 CUDA cores, running Linux 64-bit.

Algorithm		Benchmark Example										Random	
		10	21	22	31	32	33	34	35	38	39	Rnd1	Rnd2
CPU	(i) sequential	1	19	18	573	475	737	1	704	806	901	3891	7801
	(ii) parallel	2538	77	611	1564	1786	2764	279	4325	7729	8170	7966	41K
GPU GTX 280	(iii) simple	1336	17	68	217	113	359	51	242	256	290	350	1840
	(iv) intra-warp	104	22	69	320	149	528	52	404	339	344	493	2594
GPU GTX 480	(iii) simple	703	6	33	75	46	105	6	98	114	125	178	992
	(iv) intra-warp	40	7	28	109	63	157	6	152	141	158	248	1391
	(v) shared mem.	38	40	59	659	341	862	48	190	203	227	315	1800

Table 8.2: *Runtimes for CPU-based and GPU-based implementations in milliseconds*

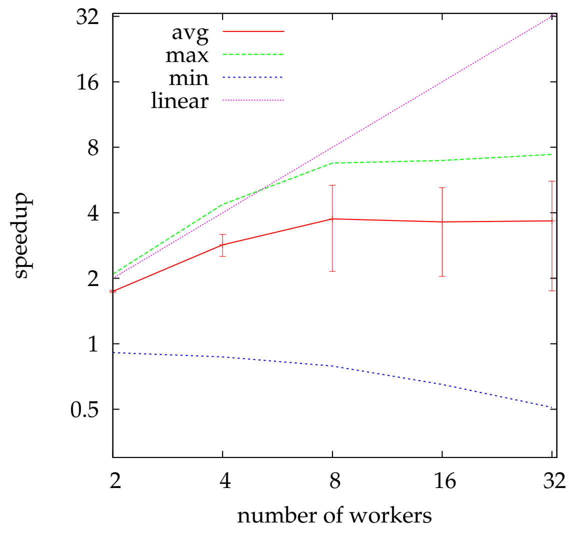
Although the systems use different CPU types this fact does not significantly affect our results as we compare our data-parallel FIXPOINT algorithm and the optimal CHASING-ONE algorithm on the same system equipped with the GPUs.

Table 8.2 provides a comparison of runtimes (in milliseconds) of the following implementations: (i) the sequential CPU implementation of the optimal CHASING-ONE algorithm, (ii) the parallel Cilk-based CPU implementation of the FIXPOINT algorithm and the GPU implementations of the FIXPOINT algorithm including (iii) the GPU implementation without any optimization, (iv) the GPU implementation with intra-warp iterations, and (v) the GPU implementation utilizing shared memory. Note that in the case of parallel CPU implementation we list the best runtimes available among the numbers of cores that have been utilize. Also note that in the case of the GTX 280 GPU, we omit the results for (iv), the shared memory implementation, as this GPU does not provide a sufficient amount of shared memory to efficiently employ this optimization.

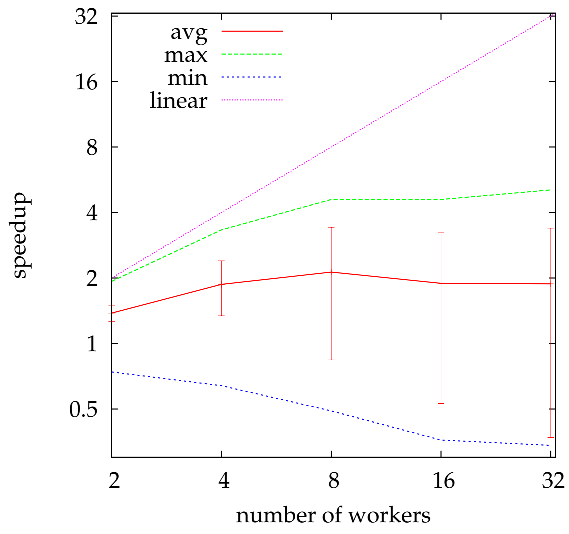
We restrict selection of benchmark examples in Table 8.2 to those for which the runtime of the GPU implementation is sensibly measurable. Nonetheless we conducted our experiments for the entire benchmark suite. The numbering of the benchmark examples refers to their position in the table provided on the VLTS website [110], which is sorted in ascending order relative to the number of states of the LTS, thus, Example 10 is *vasy_25_25*, Example 21 is *vasy_166_651*, Example 22 is *cwi_214_684*, Example 31 is *vasy_2581_11442*, Example 32 is *vasy_4220_13944*, Example 33 is *vasy_4338_15666*, Example 34 is *vasy_6020_19353*, Example 35 is *vasy_6120_11031* and Example 39 is *vasy_12323_27667*. In this naming scheme the first number is the number of states divided by 1000, and the second number is the number of transitions divided by 1000.

Furthermore, all examples in Table 8.2 are checked for the deadlock freedom property as only eight of the 40 LTSs contain livelocks. Nonetheless, our general statements about scalability and competitiveness have been evaluated and are valid for the entire benchmark suite. In order to extrapolate our results and obtain more insight to the specific structure of the benchmark's LTSs, we extend our evaluation to randomly generated BESs. We evaluate a total of five random examples with the number of states ranging from 1 to 10 million; Rnd1 and Rnd2 are two representatives illustrating our general observations for this class of BESs.

We omit an explicit measure of the memory consumption of our implementations as (i) our parallel versions operate on a static data structure that is linear in the size of the input



(a) *Deadlock freedom*



(b) *Livelock*

Figure 8.3: *Scalability of multi-core implementation.*

BES (ranging from approximately 90 KB up to 4.5 GB) and (ii) in the scope of this paper it is not our aim to optimize memory efficiency, especially since all benchmark examples easily fit our systems memory.

The results in Table 8.2 clearly show that our multi-core implementation is outperformed significantly by the optimal sequential baseline. The reason is that the parallel FIXPOINT algorithm has to perform quadratic amount of work in the worst case in contrast to the optimal CHASING-ONE algorithm. Since the total number of parallel PUs is low (32 logical cores), the computational overhead of the fixpoint iteration is too large compared to the amount of productive work and cannot be compensated by parallel processing power. This observation is supported by the two graphs in Figure 8.3, which show the overall scalability of our CPU-based approach for an increasing number of parallel workers. This result is in accordance with [107] and extends their results to our much larger benchmark suite.

The data for the two graphs in Figure 8.3 is averaged over all 40 benchmark examples, but separately evaluated for the two properties: no deadlock (Figure 8.3a) and livelock (Figure 8.3b). It shows that the average scalability is below linear for both properties, but observable for up to eight workers, which corresponds to the number of physical cores of one CPU in our system. It is important to remark that the shape of the two graphs, suggesting better scalability for LTSs that have been checked for the no deadlock property, is affected by the fact that there are 20 examples containing deadlocks, while only 8 examples contain livelocks. In the case of the trivial examples, i.e., those that do not contain deadlocks/livelocks, the algorithm needs to perform only one iteration, which has the significant impact on the scalability.

The superlinear speedup in (Figure 8.3a) can be explained by the parallel execution of workers. As the Cilk Plus framework may schedule the evaluation order of equations differently from the original one in the BES, this may lead to a faster propagation of updated LHSs, requiring less iterations and, thus, result in the seemingly superlinear boost in performance.

The evaluation of our many-core implementation is aimed more at the competitiveness of our approach when compared to the optimal sequential baseline than at its scalability.

The scalability analysis of our many-core implementation is much more difficult than for the multi-core implementation as we had to use different GPU devices that are not comparable with respect to some important specifications. Not only did the number of CUDA cores double from the GTX 280 to the GTX 480, but also the clock rate and the available amount of memory increased significantly. For this reason we did not evaluate the scalability aspect beyond the scope provided in Table 8.2, which shows a significant boost in performance for the GTX 480. We may consider the observed speedup (closing to three-fold in some instances) to be a witness of effective scalability of the presented data-parallel algorithms.

The main limitation of the GPU parallelization is the length of the chain of propagations of LHS values. The example 10 in the benchmark suite contains an artificially long chain of dependencies from the initial state to the last state (see Figure 8.2(a)). For this example, the number of iterations is equal to the number of states for the unoptimized version of our many-core implementation, yet it is a prime candidate to benefit from the intra-warp iteration as the changes can be propagated ideally within the equations of a warp.

However, the remaining benchmark examples do not have such an extreme structure and therefore the intra-warp iteration, on average, does not provide any advantage, but rather induces overhead as the comparison of runtimes in Table 8.2 shows.

Since the efficiency of our shared memory optimization is tightly coupled to the intra-warp iteration, it can only improve the performance of the many-core implementation in those cases in which the intra-warp iteration actually works. Due to this reason, the results for this optimization in Table 8.2 are, not surprisingly, even worse than for the intra-warp iteration because the transfer times from and to shared memory degrade the runtime performance even further. Moreover, in order to use the shared memory, the required data (the part of a BES corresponding to a block) has to fit the limited size of the GPUs shared memory. The size of the data that has to be stored in the shared memory is given by the block size - a number of vertices in one block and by the number of their successors. In the case the average out-degree (the average number of RHS variables per equations) is high, we have to decrease the group size. This can lead to underutilization or low occupancy of the individual multiprocessors and thus significantly reduces the performance of the algorithm.

As documented in Table 8.2 we have shown that our GPU implementations of the resolution of BESs provide significant speedup for most cases of the benchmark examples and especially for the randomly generated BESs. Surprisingly, the GPU implementation with no optimizations yields the best results, since in most of cases the structure of the inspected BESs does not allow to benefit from the designed optimizations.

Table 8.3 provides a comprehensive overview of the total number of iterations for those examples of the benchmark, that have been checked for deadlocks, where the initial approximation is not equal to the final solution, i.e., the total number of iterations is larger than one. Even though the available number of PUs continuously increases with each hardware generation, it is still far from the point where a full iteration step can be computed completely in parallel. Thus, the order of equations processing within a block has a significant influence on the total number of iterations needed to compute the fixpoint. Yet, our evaluation yields an interesting insight for an ideal *vectorized* parallelization, assuming that a fully parallel iteration step is possible. We model this by delaying the visibility of a changed LHSs until the next iteration step. Our evaluation shows that this limitation does not increase the total number of iteration significantly, compared to the *original* ordering, where equations are evaluated in their initial order and changes of LHSs are directly visible in the following computations of the iteration. This result shows that the penalty for a fully parallel computation

Heuristic	Benchmark Example																			
	4	5	7	10	15	16	18	19	21	22	25	27	30	31	32	33	35	37	38	39
Forward	64	19	7	25K	19	33	23	18	33	208	24	7	56	32	23	34	33	20	29	29
Vectorized	64	19	7	25K	23	37	23	19	37	213	24	7	56	37	25	37	34	20	29	29
Reverse	2	4	3	2	7	8	8	5	8	8	10	2	3	7	4	6	4	5	5	5
Random	20	9	5	25K	10	12	6	11	11	63	7	3	6	8	5	8	16	10	9	9

Table 8.3: *Impact of heuristics on total number of iterations*

is negligible, with respect to the total number of iterations needed to reach the fixpoint.

As the application of advanced heuristics would require costly preprocessing of the data – causing a potentially high computational overhead – we restrict our evaluation to two simple cases that do not introduce any overhead. The first heuristic, to carry out the evaluation of equations, within a BES-block, in reversed order, was similarly proposed in [107], and yields a significant improvement with respect to the total number of iterations needed to compute the fixpoint. Yet, according to our observations this heuristic only works for real world examples generated from the benchmark's LTSs, but not for randomly generated BESs.

The second heuristic is the randomized evaluation of equations within a BES-block. In our observations this heuristic leads to a decrease in the number of iterations needed to solve a BESs when compared to the *original* ordering. This result is of practical relevance as our parallel implementations rely on parallelizations in which the order of RHS evaluations is not under our control, but it is determined by the runtime environment of CUDA and Cilk Plus. Thus, we expect an additional performance boost rather than a degradation, due to the parallelization frameworks.

8.6 Conclusion

We implemented and evaluated a parallelization approach for BES resolution on multi- and many-core systems, with respect to scalability and competitiveness in comparison to the optimal sequential algorithm. The results of our experimental evaluation confirm the scalability results from [107] for the CPU-based implementation, yet its overall performance is not competitive, compared to our sequential implementation of the optimal algorithm. The utilization of many-core hardware yields a significant speedup, compared to multi-core systems. Our GPU-based implementation can compete and even outperform the optimal algorithm for most instances of the benchmark. Scalability, with respect to an increasing numbers of PUs, has been shown in our evaluation by employing cards with 240 and 480 CUDA cores, respectively.

As BESs are not restricted to model checking and we observed our approach to work very well on randomly generated BESs, in our future work we intend to evaluate input BESs from other applications, such as data-flow analysis. Furthermore, the recently proposed parallelization of the graph algorithms [69, 91], should be evaluated with respect to its suitability and potential impact on our work.

Chapter 9

Conclusion

9.1 Summary

The goal of this thesis was to design methods that can help to narrow the gap between the complexity of systems built in practice and the complexity of systems the current model checking tools can handle. The reason behind the gap is the so-called *state space explosion problem*. This causes the graphs of reachable system configurations, that have to be systematically analyzed, to be very large for realistic systems. Generating and analyzing large graphs calls for acceleration using parallel algorithms in order to obtain the desired level of performance. In this thesis we aimed at designing new methods and data-parallel graph algorithms that enable to efficiently utilize the massively parallel SIMD architectures the modern GPUs offer in order to significantly speed up the model checking process.

First, we presented general work-flow of data-parallel graph algorithms that is shared among all other graph procedures and that enables to efficiently employ modern SIMD architectures. In particular we introduced a compact graph representation and heterogeneous computation work-flow allowing for vector processing. In order to evaluate how efficiently can modern GPUs handle the proposed way of data-parallel processing of graph algorithms, we designed a simple model of data-parallel graph algorithms. The evaluation clearly justifies our motivation to use massively parallel GPUs to accelerate the graph algorithms.

Afterwards, we discussed the main drawbacks of the proposed parallelization and suggest several methods that can reduce their impact on performance of the algorithms. We also studied recently published methods [69, 91] that have the potential to further improve the performance of our data-parallel algorithms.

We further described a new approach for designing fast data-parallel graph algorithms. We primarily focused on design of new efficient data-parallel algorithms for accepting cycle detection, strongly connected component decomposition, optimal cycle detection and graph-based resolution of boolean equation systems that form the building blocks of the model checking procedure. However, our approach is more general and can be applied to design of other data-parallel graph algorithms.

We kept the provably correct layout of the existing algorithms and redesigned the algorithms to enable vector processing. In particular, we reformulated the recursion present in the algorithms by means of iterative procedures, designed basic data-parallel graph primitives and showed how can the algorithms be built from them. Note that, some algorithms are inherently sequential (e.g. TARJAN'S algorithm based on depth-first search) and thus inappropriate for SIMD parallelization. In these cases we have to consider another algorithm typically with worse time complexity that has a potential to outperform the optimal but inherently sequential algorithm using parallel processing.

Another key aspect of designing data-parallel graph algorithms is a suitable graph representation enabling the vector processing. While the raw computing power of massively parallel architectures is tremendous, its utilization in model checking is, however, quite often reduced by the costly preparation of suitable data structures and limited to small or middle-sized instances due to space restrictions. Hence, we described new techniques that allow to efficiently employ the data-parallel graph algorithms in the context of the model checking procedure. In particular, we presented a new multi-core construction of the compact data structures and new fine-grained communication-intensive parallel algorithms allowing for multiple GPUs processing.

In order to experimentally evaluate our approach we provided CUDA implementations of the designed data-parallel algorithms and compared them with implementations of the best sequential counterparts. Our experiments demonstrated that the proposed GPU accelerated algorithms significantly outperform the best sequential counterparts and thus, in total, speed up the solution of the inspected graph problems. However, the experiments also showed that the performance of the data-parallel graph algorithms deeply depends on the structure of the input graphs. Especially in the case of graphs with high diameter our parallelization tends to be inefficient.

Finally, we delivered the DiVinE-CUDA tool that implements the designed GPU accelerated algorithms for LTL model checking. We experimentally demonstrated that our methods result in a significant speedup of the model checking process. Moreover, the experimental evaluation positions our DiVinE-CUDA tool as the fastest among the state-of-the-art parallel LTL model checkers using an unbiased selection of model checking instances.

9.2 Conclusions

The main conclusions based on the research presented in this thesis are the following:

- A class of fundamental graph algorithms can be accelerated by efficient utilization of massively parallel SIMD architectures the modern GPUs offer. Resulting data-parallel algorithms significantly outperform the best sequential counterparts and thus in total speed up the solution of the corresponding graph problems. However, the acceleration is quite often limited by the structure of the input graphs and particularly the parallelization described in this thesis tends to be unsuitable for the graphs with high diameter.

- An effective and successful approach for designing fast data-parallel graph algorithms can be based on the following steps: evaluation of all existing algorithms for the given problem with respect to their predisposition for vector processing, selection of the most suitable algorithm, preservation of the provably correct layout of the algorithm, reformulation of the recursion present in the algorithm by means of iterative procedures, decomposition of the algorithm into primitive graph operations and design of their efficient data-parallel versions.

- A crucial part of designing fast data-parallel graph algorithms for model checking is the costly preparation of suitable graph representation that enables the vector processing. This preparation can be significantly accelerated by multi-core construction of the compact data structures. However, the parallel construction may affect the ordering of the input data representation which subsequently causes significant slowdown of some data-parallel graph algorithms. Therefore, the algorithms resistant to slowdown caused by improper ordering have to be preferred.

- Effective utilization of modern GPUs is limited to small or middle-sized instances due to space restrictions. However, the limitation can be overcome by fine-grained communication-intensive parallel algorithms that allow to efficiently employ multiple GPUs.

9.3 Future Work

There are several promising directions of possible future work. First, we can try to employ two recently proposed methods in order to reduce the main limitations of our parallelization. In particular we can extend the warp-centric programming method presented in [69] to improve the performance of our data-parallel algorithms on highly irregular graphs. The second method based on linear parallelization of the graph traversal [91] significantly outperforms the standard quadratic parallelization used in our data-parallel algorithms on the graphs with high diameter. Therefore this method has a great potential to drastically boost the performance of our data-parallel algorithms that heavily utilize the graph traversal procedure. However, the applicability to more complicated graph algorithms is questionable and will require further research.

Another line of work is to put significant effort in designing GPU accelerated state space generation and adjacency list computation which can lead to additional speedup of the model checking procedure. Edelkamp et al. have recently proposed acceleration of the state space generation by executing complex operations on GPUs [54]. However, duplicate detection, the most crucial part of the state space generation, has not been parallelized on GPU yet. Thus the overall speedup of the whole state space generation was moderate. We consider an efficient massive fine-grained parallelization of the duplicate detection to be another logical direction of research in the community of parallel and distributed model checking.

From a practical view of point the third vision for a promising future work is to consider real industrial applications of our data-parallel algorithms. In particular, we plan to study the performance of the data-parallel algorithms with respect to the structure of real word graphs and to design heuristics that can further improve the applicability of the algorithms.

Bibliography

[1] M. AlTurki and J. Meseguer. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *Proceedings of the 4th International Conference on Algebra and Coalgebra in Computer Science (CALCO'11)*, pages 386–392. Springer, 2011.

[2] N. Amato. Improved Processor Bounds for Parallel Algorithms for Weighted Directed Graphs. *Information Processing Letters*, 45(3):147–152, 1993.

[3] H. R. Andersen. Model Checking and Boolean Graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.

[4] D. A. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05)*, volume 3769 of *LNCS*, pages 465–476. Springer, 2005.

[5] D. A. Bader and K. Madduri. GTgraph: A Synthetic Graph Generator Suite. Technical Report GA 30332, Georgia Institute of Technology, Atlanta, 2006.

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

[7] J. Barnat, P. Bauch, L. Brim, and M. Češka. Computing Optimal Cycle Mean in Parallel on CUDA. In *Proceedings of 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'11)*, volume 72 of *Electronic Proceedings in Theoretical Computer Science*, pages 68–83, 2011.

[8] J. Barnat, P. Bauch, L. Brim, and 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)*, pages 259–266. IEEE Computer Society, 2010.

[9] J. Barnat, P. Bauch, L. Brim, and M. Češka. Computing Strongly Connected Components in Parallel on CUDA. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, pages 541–552. IEEE Computer Society, 2011.

[10] J. Barnat, P. Bauch, L. Brim, and M. Češka. Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. *To appear in Journal of Parallel and Distributed Computing*, 2012.

[11] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Distributed Qualitative LTL Model Checking of Markov Decision Processes. In *Proceedings of 5th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'06)*, pages 1–15. University of Bonn, 2006.

[12] J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *Proceedings of the 14th International SPIN Conference on Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 187–203. Springer, 2007.

[13] J. Barnat, L. Brim, and 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)*, volume 5311 of *LNCS*, pages 234–239. Springer, 2008.

[14] J. Barnat, L. Brim, and 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 '09)*, pages 407–425. Springer, 2009.

[15] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *Proceedings of the 3rd International Workshop on Verification and Computational Logic (VCL'02)*, pages 1–10. University of Southampton, 2002.

[16] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

[17] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE: A Parallel Qualitative LTL Model Checker. In *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST'07)*, pages 215–216. IEEE Computer Society, 2007.

[18] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. Local Quantitative LTL Model Checking. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *LNCS*, pages 53–68. Springer, 2008.

[19] J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE-MC: Multi-core LTL Model Checker for Probabilistic Systems. In *Proceedings of the 5th International Conference on Quantitative Evaluation of Systems (QEST '08)*, pages 77–78. IEEE Computer Society, 2008.

[20] J. Barnat, L. Brim, and M. Češka. DiVinE-CUDA: A Tool for GPU Accelerated LTL Model Checking. In *Proceedings of 8th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'09)*, volume 14 of *Electronic Proceedings in Theoretical Computer Science*, pages 107–111, 2009.

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

[22] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Proceedings of the 9th International Workshop on Parallel and Distributed Methods in verifiCation and the 2nd International Workshop on High Performance Computational Systems Biology (PDMC/HiBi'10)*, pages 4–7. IEEE Computer Society, 2010.

[23] J. Barnat, J. Chaloupka, and J. V. D. Pol. Distributed Algorithms for SCC Decomposition. *Journal of Logic and Computation*, 21(1):23–44, 2011.

[24] J. Barnat, J. Chaloupka, and J. C. van de Pol. Improved Distributed Algorithms for SCC Decomposition. In *Proceedings of 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'07)*, volume 198 of *Electronic Notes in Theoretical Computer Science*, pages 63 – 77, 2008.

[25] J. Barnat and P. Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Proceedings of Formal Methods: Applications and Technology, the 11th International Workshop on Formal Methods for Industrial Critical Systems and the 5th International Workshop on Parallel and Distributed Methods in verifiCation (FMICS/PDMC'06)*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

[26] P. Bauch and M. Češka. CUDA Accelerated LTL Model Checking - Revisited. In *Proceedings of the 6th International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10) – Selected Papers*, volume 16 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–8, 2011.

[27] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[28] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, pages 354–359. Springer, 2010.

[29] S. Blom, J. van de Pol, and M. Weber. LTSMIN: Distributed and Symbolic Reachability. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174, page 354. Springer, 2010.

[30] B. Bollig, M. Leucker, and M. Weber. Local Parallel Model Checking for the Alternation-Free μ-Calculus. In *Proceedings of the 9th International SPIN Conference on Model Checking Software (SPIN'09)*, pages 128–147. Springer, 2002.

[31] Boost: C++ Libraries. `http://www.boost.org/`, February 2012.

[32] D. Bošnački, S. Edelkamp, and 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)*, volume 5578 of *LNCS*, pages 32–49. Springer, 2009.

[33] D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs. Parallel Probabilistic Model Checking on General Purpose Graphics Processors. *STTT*, 13(1):21–35, 2011.

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

[35] L. Brim, I. Černá, P. Moravec, and 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)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

[36] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, volume 132 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–18, 2006.

[37] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Intitute of Technology, Pasadena, 1991.

[38] S. M. Burns, H. Hulgaard, T. Amon, and G. Borriello. An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems. *IEEE Transaction on Computers*, 44(11):1306–1317, 1995.

[39] I. Černá and R. Pelánek. Distributed Explicit Fair Cycle Detection (Set Based Approach). In *Proceedings of the 10th International SPIN Conference on Model Checking Software (SPIN'03)*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[40] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 4th International Conference on Data Mining (SDM'04)*, pages 442–446. SIAM, 2004.

[41] J. Chaloupka. *Distributed Algorithms for the Minimum Mean Weight Cycle Problem*. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2006.

[42] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan Primitives for Vector Computers. In *Proceedings of the 2nd International Conference for High Performance Computing, Networking, Storage and Analysis (SC'90)*, pages 666–675. IEEE Computer Society, 1990.

[43] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors (SASP'08)*, pages 101–107. IEEE Computer Society, 2008.

[44] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck. Shortest Path Feasibility Algorithms: An Experimental Evaluation. *Journal of Experimental Algorithmics*, 14:118–132, 2010.

[45] Intel Cilk Plus Framework. `http://software.intel.com/en-us/articles/intel-cilk-plus/`, February 2012.

[46] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. M. Gettrick, and J.-P. Quadrat. Numerical Computation Of Spectral Elements In Max-Plus Algebra. In *In Proceedings of the 5th IFAC Conference on System Structure and Control (SSC'98)*, 1998.

[47] R. Cole and U. Vishkin. Faster Optimal Parallel Prefix Sums and List Ranking. *Information and Computation*, 81(3):334–352, 1989.

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

[49] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide Version 4.0, February 2012. `http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[50] A. Dasdan and R. K. Gupta. Faster Maximum and Minimum Mean Cycle Algorithms for System Performance Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:889–899, 1997.

[51] A. Dasdan, S. S. Irani, and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 37–42. ACM, 1999.

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

[53] S. Edelkamp and D. Sulewski. Parallel State Space Search on the GPU. In *International Symposium on Combinatorial Search (SoCS'09)*, 2009.

[54] S. Edelkamp and 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)*, LNCS, pages 106–123. Springer, 2010.

[55] S. Evangelista, L. Petrucci, and S. Youcef. Parallel Nested Depth-First Searches for LTL Model Checking. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis (ATVA'11)*, pages 381–396. Springer, 2011.

[56] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. To Parallelize or to Optimize? *Journal of Logic and Computation*, 21:85–120, 2011.

[57] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[58] L. K. Fleischer, B. Hendrickson, and A. Pinar. On Identifying Strongly Connected Components in Parallel. In *Proceedings of the 15 Workshops on Parallel and Distributed Processing (IPDPS'00)*, volume 1800 of *LNCS*, pages 505–511. Springer, 2000.

[59] M. Garland and D. B. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53:58–66, 2010.

[60] H. Gazit and G. L. Miller. An Improved Parallel Algorithm That Computes the BFS Numbering of a Directed Graph. *Information Processing Letters*, 28(2):61–65, 1988.

[61] O. Grumberg, T. Heyman, and A. Schuster. Distributed Symbolic Model Checking for μ-Calculus. *Formal Methods in System Design*, 26:197–219, 2005.

[62] P. Harish and 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)*, volume 4873 of *LNCS*, pages 197–208. Springer, 2007.

[63] P. Harish, V. Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical Report IIIT/TR/2009/74, Center for Visual Information Technology, International Institute of Information Technology Hyderabad, INDIA, 2009.

[64] M. Harris. Optimizing Parallel Reduction in CUDA, March 2010. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[65] W. D. Hillis. The Connection Machine. *Scientific American*, 256(6):108–115, 1987.

[66] F. Holmén, M. Leucker, and M. Lindström. UppDMC: A Distributed Model Checker for Fragments of the mu-Calculus. In *Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, volume 128 of *Electronic Proceedings in Theoretical Computer Science*, pages 91–105, 2005.

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

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

[69] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 267–276. ACM, 2011.

[70] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[71] K. Ito and K. K. Parhi. Determining the Minimum Iteration Period of an Algorithm. *The Journal of VLSI Signal Processing*, 11(3):229–244, 1995.

[72] C. Joubert and R. Mateescu. Distributed Local Resolution of Boolean Equation Systems. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 264–271. IEEE Computer Society, 2005.

[73] R. Kaivola, R. Ghughaland, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, pages 414–429. Springer, 2009.

[74] R. M. Karp and J. B. Orlin. Parametric Shortest Path Algorithms with Application to Cyclic Staffing. *Discrete Applied Mathematics*, 3:37–45, 1981.

[75] M. Keinänen. Techniques for Solving Boolean Equation Systems. Technical Report A105, Helsinki University of Technology, 2006. Doctoral dissertation.

[76] D. Kozen. Results on the Propositional mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[77] A. Laarman, R. Langerak, J. Van De Pol, M. Weber, and A. Wijs. Multi-Core Nested Depth-First Search. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis (ATVA'11)*, pages 321–335. Springer, 2011.

[78] A. Laarman and J. van de Pol. Variations on Multi-Core Nested Depth-First Search. In *Proceedings of 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'11)*, volume 72 of *Electronic Proceedings in Theoretical Computer Science*, pages 13–28, 2011.

[79] A. Laarman, J. van de Pol, and 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.

[80] A. L. Lafuente. Simplified Distributed LTL Model Checking by Localizing Cycles. Technical Report 00176, Institut für Informatik, University Freiburg, Germany, 2002.

[81] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart, and Winston, New York, NY, 1976.

[82] A. Lefohn, J. M. Kniss, and J. D. Owens. Implementing Efficient Parallel Data Structures on GPUs. In *GPU Gems 2*, pages 521–545. Addison-Wesley, 2005.

[83] N. Leischner, V. Osipov, and P. Sanders. GPU Sample Sort. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*, pages 1–10. IEEE Computer Society, 2010.

[84] M. Leucker, R. Somla, and M. Weber. Parallel Model Checking for LTL, CTL*, and L_μ^2. In *Proceedings of 2nd International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 4–16, 2003.

[85] R. Lu and C. Koh. Performance Analysis of Latency-Insensitive Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):469–483, 2006.

[86] A. H. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technische Universität München, 1997.

[87] R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-free Boolean Equation Systems. *STTT*, 8(1):37–56, 2006.

[88] A. Mathur, A. Dasdan, and R. K. Gupta. Rate Analysis for Embedded Systems. *ACM Transaction on Design Automation of Electronic Systems*, 3(3):408–436, 1998.

[89] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[90] K. L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[91] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pages 117–128. ACM, 2012.

[92] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Communication of the ACM*, 53(2):58–64, 2010.

[93] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[94] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, 2007.

[95] B. Nichol, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[96] S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Free University of Amsterdam, 2004.

[97] S. Patidar and P. J. Narayanan. *Scalable Split and Gather Primitives for the GPU*. Technical Report IIT/TR/2009/99, Centre for Visual Information Technology, Hyderabad, INDIA, July 2009.

[98] 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)*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.

[99] M. Pharr and F. Randima. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[100] C. V. Ramamoorthy and G. S. Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transaction of Software Engineering*, 6(5):440–449, 1980.

[101] J. Reif. Depth-First Search is Inherrently Sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[102] A. Sailer. Utilizing And-Inverter Graphs in the Gaussian Elimination for Boolean Equation Systems. Master's thesis, Hochschule Regensburg, 2011.

[103] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 23th IEEE International Parallel & Distributed Processing Symposium (IPDPS'09)*, IPDPS '09, pages 1–10. IEEE Computer Society, 2009.

[104] S. Schwoon and J. Esparza. A Note on On-The-Fly Verification Algorithms. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.

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

[106] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Math*, 5(2):285–309, 1955.

[107] J. van de Pol and M. Weber. A Multi-Core Solver for Parity Games. In *Proceedings of 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'08)*, volume 220 of *Electronic Notes in Theoretical Computer Science*, pages 19–34, 2008.

[108] M. Vardi and P. Wolper. Reasoning about Infinite Computation Paths. *Proceedings of 24th IEEE Symposium on Foundation of Computer Science*, pages 185–194, 1983.

[109] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS '86)*, pages 332–344. IEEE Computer Society, 1986.

[110] VLTS Benchmark Suite. `http://www.inrialpes.fr//vasy/cadp/resources/benchmark\_bcg.html`, February 2012.

[111] S. Warren. Finding Strongly Connected Components in Parallel Using O(log2n) Reachability Queries. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*, pages 146–151. ACM, 2008.

[112] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster Parametric Shortest Path and Minimum Balance Algorithms. *NETWORKS*, 21(2):1–17, 1991.