# Solving Parity Games on the GPU*

Philipp Hoffmann and Michael Luttenberger

Institut für Informatik, Technische Universität München
{hoffmaph,luttenbe}@model.in.tum.de

**Abstract.** We present our GPU-based implementations of three well-known algorithms for solving parity games. Our implementations are in general faster by a factor of at least two than the corresponding implementations found in the widely known PGSolver collection of solvers. For benchmarking we use several of PGSolver's benchmarks as well as arenas obtained by means of the reduction of the language inclusion problem of nondeterministic Büchi automata to parity games with only three colors [3]. The benchmark suite of http://languageinclusion.org/CONCUR2011 was used in the latter case.

## 1 Introduction

The term "graphics processing units", short GPUs, was introduced in 1999 by Nvidia when they included hardware on the graphics chip specialized for processing triangles and lighting computations. In the last years, GPUs have constantly gained both computational power and versatility. In particular, GPUs excel at "embarrassingly parallel problems" which can be easily split into a large number of mostly independent parallel tasks, e.g. matrix-vector multiplication. Today, the fastest supercomputers combine both traditional multi-core processors and graphics processing units. Accordingly, there has been an ever growing amount of research on how to take advantage of the computational power of GPUs in general-purpose computing. Recently, Barnat et al. [1] have shown how to take advantage of GPUs in LTL model checking.

In this paper we present GPU-enabled implementations for solving parity games. Solving these games is a problem of great interest because if its applications in model checking as well as synthesis. The algorithms we use are the small-progress-measure (SPM) algorithm by Jurdzinski [5], the recursive algorithm due to Zielonka [9] and a variant of the strategy iteration (SI) algorithms of [2,7,8] described in [6]. We use the GPU for solving, roughly spoken, weighted min-max systems underlying all three algorithms. To our best knowledge, solving parity games using the GPU was not previously studied in literature.

We implemented all three algorithms using the Nvidia specific CUDA tool kit. Implementations using the vendor independent OpenCL framework will be

* This work was partially founded by the DFG project "Polynomial Systems on Semirings: Foundations, Algorithms, Applications" and by the DFG Graduiertenkolleg 1480 (PUMA).

released at a later point of time; so far we only have a first OpenCL-version of the SI-algorithm. While the CUDA-based implementations make better use of our hardware (but can only be run on Nvidia GPUs) and thus gives a better impression of the attainable speed-up, the advantage of OpenCL, besides being vendor independent, is that it can be executed both on GPUs and also on multi-core CPUs commonly used in todays desktop PCs. This allows us to assess the speedup obtained by moving from the CPU to the GPU.

Furthermore we compare our current implementation to the PGSolver by Friedmann and Lange [4] in order to assess its absolute speed. PGSolver has been in development for several years, therefore we deem it a reasonable choice for evaluating the speed of our own implementation. As benchmarks we use randomly generated arenas, arenas generated from LTL verification, and arenas obtained via the reduction by Etessami et. al. [3] of the language inclusion problem of nondeterministic Büchi automata to parity games.

The current implementations and benchmarks are available at `www.model.in.tum.de/tools/gpupg`.

## 2 GPU-specific implementation

Due to the page limit, we have to assume that the reader is familiar with parity games and cannot discuss GPU programming in detail. For more informations on the algorithms we refer the reader to the respective articles [5,9,6]; for a general introduction to GPU programming, please see the respective material made available by the Khronos group or by hardware vendors like AMD, Intel, or Nvidia. Very roughly spoken, a modern GPU consists of several multi-processors which act independently of each other; each multi-processor itself processes a large number of "warps" of 32 threads in parallel; all threads of a warp execute the same instruction (or do nothing).

We give a very brief sketch of how we use the GPU: For storing the arena, we use separate arrays for storing attributes like owner, color, etc. The successors are stored similar to the Yale format used for sparse matrices. At the heart of all three algorithms lies the problem of computing the least or greatest solution of min-max systems (over different algebraic structures) which are directly derived from the graph structure underlying the arena (variables correspond to nodes, equations to edges). For instance, computing the usual attractor means to solve a min-max system where every variable takes only values in $\{0, 1\}$. In all three cases the min-max systems can be solved using standard fixed-point iteration. The basic idea common to all three implementations is to implement the fixed-point iteration on the GPU by assigning to each node a thread which re-evaluates its defining equation in each iteration. This approach is advantageous when a lot of variables need to be updated in every iteration, but unprofitable if only a few updates are required. For this reason, we have also experimented with a worklist implementation on the GPU based on the stream compaction methods of the thrust library; but in out experiments the added cost for handling the worklist outranges the benefit of processing less nodes.

## 3 Evaluation

We have benchmarked our current implementation on several instances of parity games and compared the results to PGSolver (Version 3.3, released January, 19th, 2013). All tests have been run on an Intel Core i7-3820 Processor, currently 280 €, with 16 GB of RAM and a Nvidia GTX660, currently 180 €, with 2 GB of RAM running Windows 7 64bit. To exclude device startup times from our benchmarks, we ran all GPU benchmarks four times, discarded the first and took the average of the remaining three runs.

We apply the following preprocessing steps to the arena before solving them or handing them to PGSolver: We order the nodes in a topological ordering using Tarjan's SCC algorithm as a heuristic to optimize memory access on the GPU. For each of the two players we remove all nodes which the player can win by visiting only nodes controlled by him. For each SCC we further "compact" colors in the obvious way, e.g. if no node uses the color 5, but the colors 4 and 6 are used, we reduce all colors greater than 5 by 2.

We implemented the SI algorithm both in the Nvidia specific CUDA framework and in the vendor independent OpenCL framework. The code is the same up to those changes necessiated by the frameworks. As the CUDA version outperformed the OpenCL version in all benchmarks, we implemented the SPM and the recursive algorithm using only the CUDA framework. For comparison, we ran PGSolver using the solvers corresponding to the SI, the SPM and the recursive algorithm.[1] PGSolver includes lots of (polynomial-time) optimizations and preprocessing steps that already solve parts or in some cases all of the parity game (in these cases all three solvers have nearly identical solving times) before the actual solver is applied. For comparison we also ran the recursive algorithm with disabled preprocessing/optimizations, labelled as "PG Rec (pure)".

To get a rough estimate of the behaviour of the implementation in general we used 100 randomly generated arenas of each of the following types: *Steady random arenas* have 500,000 nodes, 16 colors and in- and outdegree between 2 and 32. *Clustered random arenas* also have 500,000 nodes and 16 colors.[2] Using a timeout of one minute, every solver either solved all arenas (Figure 1 lists the average solving times) or none (denoted by a $*$). For more practical benchmarks, we used the reduction by Etessami et al. [3] of the language inclusion problem of nondeterministic Büchi automata (NBA) – which is at the heart of automata theoretical approach for LTL model checking – to parity games and used the NBAs found on http://languageinclusion.org/CONCUR2011 for benchmarking. These arenas use three colors and their number of nodes ranges between 40,000 and 1,100,000. The benchmark results are summarized in Figure 1. Also included in this table are two instances of PGSolver's elevator (LTL) verification game.

The speedup obtained by our implementations is in most cases quite noticeable: The SI algorithm is faster by a factor 1.5-4 when compared to PGSolver's recursive algorithm (note that PGSolver's SI and SPM had multiple timeouts

[1] The parameters for PGSolver are -global {optstratimprov, smallprog, recursive}.
[2] Additional parameters: 2 32 4 4 4 10 20. PGSolver manual offers more information.

| | SI | | | | SPM | | recursive | | |
|---|---|---|---|---|---|---|---|---|---|
| | cuda | ocl (CPU) | ocl (GPU) | PG | cuda | PG | cuda | PG (pure) | PG |
| clustered | 5.84 | 5.81 | 5.87 | * | * | * | 2.31 | 18.62 | 18.54 |
| steady | 5.40 | 8.11 | 5.58 | * | * | * | 4.12 | 21.69 | 32.93 |
| ele_6 | 0.73 | 1.85 | 1.61 | 0.95 | 12.03 | 0.95 | 0.10 | 2.00 | 0.95 |
| ele_7 | 7.60 | 20.29 | 8.63 | 10.83 | 559.09 | 10.81 | 0.85 | 16.38 | 10.81 |
| bakery | 0.43 | 1.11 | 0.62 | 2.54 | 26.55 | 1.40 | 0.24 | 2.95 | 0.84 |
| bakeryV2 | 0.22 | 0.81 | 0.37 | 0.69 | 14.61 | 0.70 | 0.13 | 1.53 | 0.47 |
| fischer | 0.90 | 1.73 | 0.96 | > 30 min | > 30 min | 10.12 | 0.97 | 8.24 | 7.41 |
| fischerV3 | 0.80 | 1.70 | 0.89 | 2.28 | 89.19 | 2.28 | 0.61 | 9.22 | 2.28 |
| fischerV4 | 0.07 | 0.62 | 0.19 | 0.09 | 1.77 | 0.09 | 0.04 | 0.44 | 0.09 |
| mcs | 1.02 | 1.78 | 1.15 | 2.84 | 134.173 | 2.87 | 0.62 | 13.73 | 2.84 |
| fischerV5 | 2.59 | 6.41 | 2.94 | 3.56 | > 30 min | 3.56 | 0.96 | 6.63 | 3.56 |
| philsV4 | 0.02 | 0.56 | 0.06 | 0.03 | 2.22 | 0.03 | 0.02 | 0.11 | 0.03 |

**Fig. 1.** Benchmark results. All times in seconds if not stated otherwise.

on arenas which our implementation did solve), the recursive algorithm in some cases reaches a speedup factor of 10. Although the SPM algorithm has the best worst-case upper bound, it performed worst in all of our experiments.
Regarding the question of the advantage of the GPU, in most of our benchmarks the OpenCL version of the SI algorithm performed perceivably better on the GPU than on the quad-core CPU (all cores were used). Future optimizations are certainly possible; an experimental version of our SPM solver containing a better preprocessing including SCC-decomposition on the GPU yielded drastically improved times: for instance, the language inclusion problem "mcs" can now be solved in 7 seconds instead of 134 seconds.

## References


1. Barnat, J., Bauch, P., Brim, L., Ceska, M.: Designing fast ltl model checking algorithms for many-core gpus. J. Parallel Distrib. Comput. 72(9), 1083–1097 (2012)
2. Björklund, H., Sandberg, S., Vorobyov, S.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In: MFCS'04. pp. 673–685. LNCS 3153, Springer (2004)
3. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for büchi automata. In: ICALP. pp. 694–707 (2001)
4. Friedmann, O., Lange, M.: The PGSolver collection of parity game solvers. University of Munich (2009), available from `http://www2.tcs.ifi.lmu.de/pgsolver/`
5. Jurdzinski, M.: Small progress measures for solving parity games. In: STACS. pp. 290–301 (2000)
6. Luttenberger, M.: Strategy iteration using non-deterministic strategies for solving parity games. Tech. rep., Technische Universität München, Institut für Informatik (April 2008)
7. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: CSL. pp. 369–384 (2008)
8. Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games (Extended abstract). In: CAV'00. LNCS, vol. 1855 (2000)
9. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. 200(1-2), 135–183 (1998)