Solving Mean-Payoff Games on the GPU

Philipp J. Meyer(✉) and Michael Luttenberger

Institut für Informatik, Technische Universität München, Munich, Germany
meyerphi@in.tum.de, luttenbe@model.in.tum.de

Abstract. General purpose computation on graphics processing units (GPGPU) is a recent trend in areas which heavily depend on linear algebra, in particular solving large systems of linear equations. Many games, both qualitative (e.g. parity games) and quantitative (e.g. mean-payoff games) can be seen as systems of linear equations, too, albeit on more general algebraic structures. Building up on our GPU-based implementation of several solvers for parity games [8], we present in this paper a solver for mean-payoff games. Our implementation uses OpenCL which allows us to execute it without any changes on both the CPU and on the GPU allowing for direct comparison.

We evaluate our implementation on several benchmarks (obtained via reduction from parity games and optimization of controllers for hybrid systems [10]) where we obtain a speedup of up to 10 on the GPU in cases of MPGs with $20 \cdot 10^6$ nodes and $60 \cdot 10^6$ edges.

1 Introduction

In a mean-payoff game (MPG) [5] two players, $\mathsf{P}_{\max}$ und $\mathsf{P}_{\min}$, move a pebble through a directed graph (V, E), called the arena, where every node $v \in V$ is assigned an owner $o(v) \in \{\mathsf{P}_{\max}, \mathsf{P}_{\min}\}$ and every edge $(u, v) \in E$ is assigned a (w.l.o.g.) integer weight $w(u, v) \in \mathbb{Z}$. It is assumed that every node has at least one successor, therefore the players usually play forever, yielding an infinite path $(v_i)_{i\in\mathbb{N}}$ in (V, E) where it is assumed that the owner of v_i has chosen to move to v_{i+1}. $\mathsf{P}_{\max}$ has the goal to maximize the average of the weight accumulated in the limit, i.e. $\liminf_{T\to\infty} \frac{1}{1+T} \sum_{i=0}^{T} w(v_i, v_{i+1})$, and $\mathsf{P}_{\min}$ has the opposite goal.

It is well-known that for every MPGs there exist memoryless strategies $\sigma_{\max}$, $\sigma_{\min} \colon V \ni v \mapsto w \in vE$ and a valuation $\nu \colon V \to \mathbb{Q}$ s.t. when $\mathsf{P}_{\max}$ uses $\sigma_{\max}$ to determine where to move the pebble to — no matter how $\mathsf{P}_{\min}$ chooses to move — the resulting average reward will be at least $\nu(v)$ for v the node the pebble has been placed initially, and symmetrically for $\mathsf{P}_{\min}$ using $\sigma_{\min}$. Determining ν and optimal strategies for both players is known to be in NP∩coNP [13]. Computing optimal strategies and the optimal valuation ν can be reduced e.g. via binary search to the p-mean partition problem [3]: given $p \in \mathbb{Q}$, partition the set

This work was partially funded and supported by the Deutsche Forschungsgemeinschaft (DFG) through the project “Game-based Synthesis for Industrial Automation” and the TUM International Graduate School of Science and Engineering (IGSSE).

V into the subsets $V_{\leq p} = \{v \in V \mid \nu(v) \leq p\}$ and $V_{>p} = V \setminus V_{\leq p}$ (resp. $V_{\geq p}$ and $V_{<p}$).

Our interest in MPGs comes from optimizing permissive controllers which have been synthesized for hybrid systems [10]: the controller is obtained by discretizing the hybrid system into a game, thus its size directly grows with the resolution at which the system is discretized leading to controllers with several million states and transitions. The controller is only synthesized w.r.t. qualitative objectives like reachability or safety, but the controller is non-deterministic in the sense that it still might be allowed to choose from several actions. Essentially, the goal is to refine the controller such that it also minimizes the number of times at which it switches from one action to another action. One simple way to obtain such kind of a posteriori optimization is by formulating this problem as an MPG played essentially on the controller itself.[1] To this end, we require a solver that can handle also MPGs with several millions of states.

Motivated by the existing success in using graphic processing units (GPU) for formal verification [1,8] we present here our GPU-based implementation of a solver for MPGs. The main motivation for using GPUs is that in many cases they offer a higher computational power while consuming less energy at the same time than most CPUs. Because of this, GPUs have become a central component of super computers. Modern GPUs excel in particular in problems which can be solved by a large number of very homogeneously behaving threads e.g. like solving systems of linear equations. MPGs can be seen as a linear optimization problem, albeit w.r.t. the tropical semiring. The optimization problem associated with mean-payoff games can be solved using an approach called *strategy iteration* [3]: while strategy iteration is not known to yield a polynomial-time algorithm for solving MPGs, similar to the simplex method, worst-case behavior is observed very seldom. As no polynomial time algorithms for MPGs are known so far, strategy iteration is a reasonable approach for solving MPGs.

We benchmark our implementation on several problems coming from applications like the sketched optimization of a controller and from model checking and equivalence problems obtained via the reduction of parity games to MPGs. In order to assess the speedup that the GPU offers compared to standard CPUs, our implementation uses OpenCL, thus the same code can be run both on the CPU and the GPU. Our current implementation already achieves a speedup of approx. 8 to 10 on the GPU. We are currently optimizing the code further, and expect even higher speedups (see the section on future work for more details).

Closely related to MPGs are *parity games* which can be directly reduced to the 0-mean partition problem. The standard reduction works by encoding the node coloring explicitly as edge weights. Alternatively, the mapping of colors to edge weights can only be implicitly represented (similar to the path profiles of [12]) which requires only a slight change to our implementation but already leads to a significant speed-up for parity games.

[1] While mean-payoff parity games allow to directly combine qualitative and quantitative objects [4], our approach does not require any changes to synthesis of the controller, neither to synthesis itself nor to numerical aspects of the synthesis.

The current implementation, benchmarks and detailed results are available at www.model.in.tum.de/tools/gpumpg.

2 GPU-Specific Implementation

OpenCL is a framework for heterogeneous platforms consisting of several CPUs and GPUs. It is maintained by the Khronos Group. An OpenCL device consists of one or more *compute units (CU)* which themselves consist of one or more *processing units (PU)*. A *kernel* is a program which is to be executed in parallel: kernel instances are grouped into *work-groups* which are further subdivided into *work-items*. A work-group consists of instances of the same kernel, and the whole work-group is executed by a single CU. A work-item of a work-group executes one kernel instance using one or more PUs of the CU running its work-group. At any given point of time, all active work-items of a work-group execute the same instruction (or a NOP) which considerably influences the way algorithms have to be implemented on the GPU.

Due to the page limit, we can only give a very brief sketch of our implementation leaving aside all details on how to efficiently use the GPU: The edge relation of the arena is stored similar to the Yale format used for sparse matrices. The main problem consists of computing the least or greatest solution of min-max systems which are directly derived from the graph structure underlying the arena. To solve these systems, we implemented a variant of the Bellman-Ford algorithm directly for GPUs: roughly spoken, with every node v of the arena a work-item is associated which checks the successors of v for changes, and, if a change is found, accordingly updates the value for v.

The Bellman-Ford algorithm lies at the heart of the strategy iteration used for solving the p-mean partition problem. Given an initial memoryless strategy for $\mathsf{P}_{\max}$, the Bellman-Ford algorithm is used to compute an optimal counter strategy for $\mathsf{P}_{\min}$; this counter strategy is used in turn to improve $\mathsf{P}_{\max}$'s strategy by checking all nodes controlled by $\mathsf{P}_{\max}$ if there are any successors which promise a higher limit-average than the successor currently proposed by the strategy — again this can be done in parallel by one work-item per node. As soon as $\mathsf{P}_{\max}$'s strategy cannot be improved anymore, we are guaranteed to have found an optimal strategy, and this solved the p-mean partition problem.

The p-mean partition problem is then used to recursively partition the MPG into smaller MPGs similar to a binary search: Initially, we solve the 0-mean partition problems yielding $V_{\leq 0}$ and $V_{\geq 0}$. The set of nodes $V_{=0} = V_{\leq 0} \cap V_{\geq 0}$ is obtained, and then the MPG is partitioned accordingly into smaller MPGs consisting of the nodes $V_{>0}$ resp. $V_{<0}$. Recursively and in parallel p-mean partition problems on these smaller MPGs are solved where for each of the smaller MPGs the new value for p is chosen by traversing the Stern-Brocot tree (see e.g. [7]) in combination with exponential search. This ensures that each value in the range of ν is reached within a logarithmic number of steps w.r.t. the size of the MPG. In this way, we compute ν and optimal strategies for $\mathsf{P}_{\max}$ and $\mathsf{P}_{\min}$.

One requirement for the strategy iteration algorithm is that there are no nonpositive cycles controlled by $\mathsf{P}_{\min}$. To eliminate those cycles, we introduce additional implicit nodes on either all forward or all backward edges between nodes controlled by the same player. As an alternative, we also offer to directly remove nonpositive cycles before solving each p-mean partition problem.

3 Evaluation

All experiments were performed on a machine equipped with an Intel Core i7-6700K Processor at 4.0 GHz with 32 GB of RAM and an AMD Radeon R9 390 with 8 GB of RAM, running Windows 10 64 bit. We only present the time spent to solve the games while the time spent on disk I/O and set-up is excluded. Our benchmarks come from two sources: (1) optimization of synthesized controllers for hybrid system, (2) parity games related to model checking problems and equivalence testing of processes. The time limit was 12 h for the controller benchmark suite and 30 min for the parity games benchmark suite.

In case of the optimization of the hybrid controller, the controller is transformed into an MPG where switching from one action to another leads to a transition with zero payoff, while using the same action subsequently yields a transition with positive payoff. To this end, the states of the MPG are essentially the states of the controller extended by the action last used by the controller. The size of the synthesized controller depends on the resolution η used for discretizing the hybrid system. The results are summarized in Table 1. Note that larger games may have a simpler structure and smaller values, leading to a smaller solving time, e.g. when increasing η^{-1} from 1000 to 2000 or from 5000 to 6000.

We used the standard reduction of parity games to the 0-mean partition problem of MPGs to reformulate the model checking and equivalence problems coming from the benchmark suite[2] of [9] as MPGs. We selected only games with at least 500 000 nodes to give a useful comparison between GPU and CPU, as we could solve all smaller games in less than half a second. The game sizes range up to 40 million nodes and 167 million transitions. We also ran the tool PGSolver [6] on this suite with the solver `recursive`, which proved to be its most efficient solver. Table 2 and Fig. 1 give the obtained results.

We can solve all of the benchmarks, taking at most 2 h for the largest controller and 70 s for a single instance of the parity games. On the GPU, we achieve a speedup over the CPU ranging from 2 to 11, and on average about 5. PGSolver can only solve about half of the instances, as it runs out of memory in a lot of cases. We outperform PGSolver significantly on all but two benchmarks, where we are only half as fast. Even on the CPU we outperform PGSolver on all but eight benchmarks. This is even though PGSolver itself solves the original, smaller parity game and further uses several heuristics to recognize and solve trivial instances without actually using the selected solver.

For comparison, we also solved the parity games by implicitly mapping the colors to edge weights as color profiles. Solving the parity games directly this way gives a further speedup of 4 on average. Still, this shows our approach also works well directly on mean-payoff games, which are potentially harder to solve.

[2] Available at https://github.com/jkeiren/paritygame-generator.

Table 1. Timings in seconds for the controller synthesis benchmark suite, including speedup. The optimal value ν is the same for all nodes.

η^{-1}	$\frac{\lvert V\rvert}{10^6}$	$\frac{\lvert E\rvert}{10^6}$	ν	GPU	CPU	$\frac{CPU}{GPU}$
1000	0.4	1.2	$^{32}/_{77}$	54	288	5.3
2000	1.7	4.9	$^{5}/_{12}$	16	144	8.8
3000	4.0	11.1	$^{41}/_{98}$	339	3234	9.6
4000	7.1	19.8	$^{31}/_{74}$	407	3686	9.0
5000	11.2	30.9	$^{31}/_{74}$	484	4760	9.8
6000	16.0	44.5	$^{13}/_{31}$	404	4525	11.2
7000	21.9	60.6	$^{73}/_{174}$	1983	21700	10.9
8000	28.6	79.2	$^{199}/_{474}$	6313	>12 h	n/a

Table 2. Games in the parity games benchmark suites (#) and instances successfully solved on the GPU, on the CPU and by PGSolver (PG).

Suite	#	GPU	CPU	PG
Equiv	45	45	45	15
Model	58	58	58	44
Total	103	103	103	59

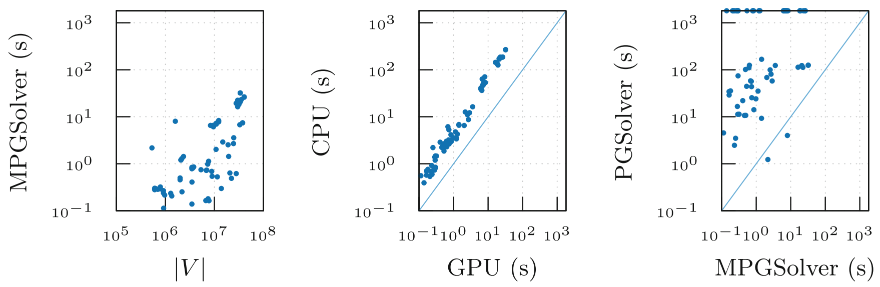

Fig. 1. Timings for the parity games benchmark suite, comparing GPU vs. CPU and our solver on the GPU vs. PGSolver. Negative results are set to 30 min.

In the experiments, the cycle elimination with implicit auxiliary nodes proved to be much more efficient than removing cycles directly, which often lead to timeouts on graphs with nonpositive cycles. Therefore we only used the implicit nodes, which caused minimal overhead even on graphs without cycles.

4 Conclusion and Future Work

Currently, the code is not optimized for the GPU, particularly the memory access pattern depends directly on the graph structure of the MPG. Still, the benchmarks indicate that GPUs offer a significant speedup of ten and more compared to CPUs. Also, we currently require an explicit representation of the system. However, as shown by the Divine model checker [2], explicit representation of the state space can be very successful in practice. To overcome possible memory limitations, a future goal is to incorporate the use of multiple GPUs in a single host system and the use of distributed system of PCs with multiple GPUs. This is also motivated by the fact that in recent years the price of GPUs dropped significantly faster than that of CPUs resp. the computational power available at a given price point increased much faster for GPUs. We thus believe that GPU-enabled solvers are relevant for model checking and synthesis in practice.

Further, we want to extend the solver by using symmetric strategy iteration [11], and we want to improve memory access by taking the graph structure of the MPG into account when arranging the nodes in memory.

Acknowledgments. We thank Majid Zamani and Matthias Rungger for kindly providing the example for the hybrid controller for our experimental evaluation.

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. Barnat, J., et al.: DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. LNCS, vol. 8044, pp. 863–868. Springer, Heidelberg (2013)
3. Björklund, H., Sandberg, S., Vorobyov, S.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 673–685. Springer, Heidelberg (2004). doi:10.1007/978-3-540-28629-5_52
4. Chatterjee, K., Majumdar, R.: Minimum attention controller synthesis for omega-regular objectives. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 145–159. Springer, Heidelberg (2011). doi:10.1007/978-3-642-24310-3_11
5. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. Int. J. Game Theory **8**(2), 109–113 (1979)
6. Friedmann, O., Lange, M.: The PGSolver collection of parity game solvers. University of Munich (2009). http://www2.tcs.ifi.lmu.de/pgsolver/
7. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1994)
8. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 455–459. Springer, Heidelberg (2013). doi:10.1007/978-3-319-02444-8_34
9. Keiren, J.J.A.: Benchmarks for parity games. In: Dastani, M., Sirjani, M. (eds.) FSEN 2015. LNCS, vol. 9392, pp. 127–142. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24644-4_9
10. Rungger, M., Zamani, M.: SCOTS: a tool for the synthesis of symbolic controllers. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, pp. 99–104. ACM, New York (2016)
11. Schewe, S., Trivedi, A., Varghese, T.: Symmetric strategy improvement. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 388–400. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47666-6_31
12. Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000). doi:10.1007/10722167_18
13. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theor. Comput. Sci. **158**(1&2), 343–359 (1996)