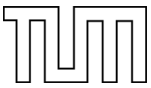

Technische Universität München

Department of Mathematics

Bachelor's Thesis

Strategy Iteration on the Graphics Card

Philipp Hoffmann

Supervisor: Univ.-Prof. Dr. Dr. h.c. Javier Esparza

Advisor: Dr. Michael Luttenberger

Submission Date: 11.9.2012

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Garching,

Zusammenfassung

Diese Bachelorarbeit behandelt die Implementierung eines Lösers für Paritätsspiele mittels NVIDIA CUDA [4]. Ziel ist durch Umsetzung eines bestehenden Algorithmus die Eignung des Problems für massiv parallele GPU-Programmierung zu evaluieren.

Paritätsspiele werden von zwei Personen auf einem Graphen gespielt, der in unserem Fall endlich ist. Das Spielziel besteht darin, einen unendlichen Pfad mit bestimmten Eigenschaften in diesem Graphen zu bilden. Diese Spiele dienen zum einen als alternative Beschreibung von μ-Kalkül Formeln und können zum Beispiel eingesetzt werden, um zu prüfen ob für ein gegebenes System ein eine Formel wahr ist ("model checking") oder um für eine Formel ein System zu finden, das sie erfüllt ("synthese") [7]. Auch die Komplexitätstheorie hat aufgrund der bisherigen Ergebnisse, die vermuten lassen, dass Paritätsspiele in polynomieller Zeit lösbar sind, reges Interesse an Paritätsspielen. Diese Spiele werden in Kapitel 2 vorgestellt.

Als Algorithmus zur Lösung dieser Spiele wird in dieser Arbeit die Strategieiteration präsentiert. Die hier benutzte Variante besteht zum größten Teil aus einem adaptierten Bellman-Ford Algorithmus, der gut parallelisierbar ist. Diese Eigenschaft soll durch den Einsatz von GPU-Programmierung, also unter Einbeziehung der Grafikkarte, genutzt werden, um einen Geschwindigkeitsvorteil gegenüber herkömmlichen CPU-basierten Lösern zu erreichen.

Kapitel 3 befasst sich mit dem PGSolver [6] von Oliver Friedmann, einem Tool zur Lösung von Paritätsspielen. Ich stelle einige von diesem Programm benutzte Optimierungen sowie Beispiele von Paritätsspielen vor. PGSolver dient als Grundlage zur Erzeugung von Paritätsspielen, zum Lösungsvergleich und wird in Benchmarks verwendet, um die Verbesserungen des GPU-basierten Algorithmus im Vergleich zu bestehenden CPU-Implementierungen zu zeigen.

In Kapitel 4 gehe ich näher auf CUDA und dessen Besonderheiten ein. Die Architektur der GPU-Programmierung wird genauer erläutert und einige wichtige Probleme, die sich dadurch ergeben werden diskutiert.

Kapitel 5 beschreibt die Besonderheiten meiner Implementierung im Zusammenhang mit CUDA und befasst sich genauer mit den Problemstellungen die in Verbindung mit GPU-Programmierung auftreten.

Kapitel 6 liefert eine abschließende Zusammenfassung der Ergebnisse und diskutiert offene Probleme und Verbesserungen.

Contents

Chapter 1

Introduction

In this work we cover the implementation of a parity game solver using NVIDIA CUDA[4]. By using an existing algorithm, we evaluate whether the problem of solving such games is suitable for the massively parallel GPU computing architecture.

Parity games are two player games played on finite graphs. During the game, an infinite path in the graph is formed by the players; certain properties of this path are then used to determine the winner. Parity games are subject of interest for theoretical informatics as they offer another representation for μ-calculus formulae. For a given formula, deciding whether a specific system satisfies it ("model checking") or finding a system that satisfies it ("synthesis") can be done using parity games [7]. They are also studied in complexity theory for their placement in UP $\cap$ co-UP [11]. In Chapter 2 these games will be presented.

For our implementation we choose strategy iteration as a solving algorithm. The variant used mainly consists of an altered Bellman-Ford algorithm that can be easily parallelized. This property shall be exploited by using GPU-programming, that is, using the graphics card for computations. Because of their immense computational power, this can offer a huge speedup over CPU computation for suitable problems.

In Chapter 3 we present an existing tool for parity game solving, PGSolver [6] by Oliver Friedmann. This application was used for multiple reasons in our work. First, it provides us an already existing solver to check our solutions. Second, it offers the possibility to create parity games of any size that were used for testing and benchmarking. Finally, it is used in benchmarks where its performance is compared to our GPU-version.

In Chapter 4 we discuss NVIDIA CUDA and the architecture behind GPU-programming. We show problems that arise when using the graphics card and also give an idea of how to optimize code for CUDA.

Chapter 5 covers out implementation, especially the CUDA related parts. We see more specific examples for optimizations as well as benchmarks that show how well our application performs.

In chapter 6 we give results and future adaptions and improvements of our implementation.

Chapter 2

Parity games

2.1 Introduction

This section introduces parity games, first with an informal description, then formally. We present definitions for determinacy and recapitulate that parity games are memoryless determined. In Section 2.2 basic ideas of strategy iteration are explained. We present two algorithms based on strategy iteration, one of which is the algorithm implemented for this thesis. Section 2.3 gives an overview on other approaches for solving parity games.

2.1.1 Intuitive description

Parity games are two player games played on finite graphs. The directed graph $G = (V, E)$ is partitioned into two sets of nodes belonging to the respective player. Additionally a coloring of the graph is given by a function c assigning natural numbers to the nodes. These colors will sometimes be called priorities of the nodes. A game can be imagined to proceed as follows:

A token is placed on one node of the graph. A move consists of choosing an outgoing edge from the node the token is placed on and moving the token along this edge. The player choosing the edge is the owner of the node the token is currently placed on. Proceeding in that manner an infinite path on G is formed. The winner of the game is defined by the parity of the maximal color appearing infinitely often along this path.

Of course it is not possible to play an infinite number of moves, so imagine both players fix some (finitely described) strategy and hand it over to a judge that decides the outcome. It can be shown that such finitely described strategies are sufficient to achieve the best outcome. In fact, so called memoryless or positional strategies, that is, the move depends only on the current node and not on the past moves, yield the same outcome of the game. This is called memoryless determinacy of parity games.

2.1.2 Formal description

A parity game is given as (V, E, o, c) where (V, E) is a finite, directed graph, $o : V \to \{0, 1\}$ assigns an owner to each node and $c : V \to \{0, \ldots, d-1\}, d \in \mathbb{N}$ is a coloring of V. We denote by $V_i = o^{-1}(i)$ the set of nodes belonging to player i ($i \in \{0, 1\}$) and by $E_i = E \cap (V_i \times V)$ the edges leaving player i nodes.

We assume that the graph has no dead ends (nodes without outgoing edges) and thus every game is infinite. This is not a real restriction as we will see later.

A play $\pi = (v_0, v_1, v_2, \dots) \in V^\omega$ is an infinite sequence of nodes in G where $(\pi(i), \pi(i+1)) \in E \ \forall i \in \mathbb{N}$. By $c(\pi) = (c(v_0), c(v_1), c(v_2), \dots)$ we denote the sequence of colors occurring in π. For any infinite sequence $a = (a_1, a_2, \dots)$, $\mathrm{Inf}(a) = \{k : a_i = k \text{ for infinitely many } i\}$ denotes the set of elements occurring infinitely often in a. A play π is won by player 0, if $\max(\mathrm{Inf}(c(\pi)))$ is even, else player 1 wins.

For a node $s \in V$, we denote by $sE = \{v \in V : (s, v) \in E\}$ the set of successors of s. Let $w \in V^*, w = v_0 v_1 \cdots v_k$ be a finite path in G and let $\sigma : V^* V_i \to \mathcal{P}(V)$ be a partial function. We call w conform with σ if for every $0 < j < k$ where $v_j \in V_i$, σ is defined at $v_0 v_1 \cdots v_j$ and we have $v_{j+1} \in \sigma(v_0 v_1 \cdots v_j)$. We call a function σ a strategy of player i if it is defined on all paths w conform with it that end in a player i node v_k and we have $\sigma(w) \subset v_k E$ for these paths. A play π is called conform with σ if every prefix of it is conform with σ.

A strategy σ is called deterministic if for all paths w on which it is defined, $|\sigma(w)| = 1$. If a strategy σ only depends on the current node, that is, for every two paths $w_1 v$, $w_2 v$ conform with σ and $v \in V_i$, $\sigma(w_1 v) = \sigma(w_2 v)$, it is called memoryless or positional. We can describe such memoryless strategies as $\sigma \subseteq E_i$ such that $\forall s \in V_i : s\sigma \neq \emptyset$. By $E_\sigma = E_{(1-i)} \cup \sigma$ we denote the restriction of E to σ.

A strategy σ of player i is called winning for a given node $s \in V$ if he wins every play starting from s that is conform with σ. Player i wins a node, if he has a winning strategy for it. The set of all nodes that player i wins is denoted by W_i and called his winning set. A set on which player i has a memoryless winning strategy for every node is called a i-paradise. It is well known that the game graph is partitioned into a 0-paradise and a 1-paradise. Therefore it suffices to consider memoryless strategies and we drop the "memoryless" from now one. In section 2.1.3 we present a proof for this fact.

The following two theorems show that there is a deterministic strategy of player i that wins all nodes in the winning set of player i. The first theorem reduces non-deterministic strategies to deterministic ones:

Theorem 2.1.1. *For every winning strategy σ of player i, there is a deterministic strategy σ' winning the same nodes.*

Proof. Let S be the set of nodes won by player i when he plays according to σ. Let $\sigma' \subseteq \sigma$ be any deterministic strategy consisting of edges of σ. Since every game starting from S and following σ is winning for player i, the particular game resulting when following σ' will be winning for player i. □

The next theorem shows how to combine strategies for different but not necessarily distinct sets.

Theorem 2.1.2. *Let S be the set of nodes won by player i. Then there is a winning strategy for player i that wins from all nodes in S.*

Proof. For every $s \in S$ there is a deterministic winning strategy σ_s that wins s (and possibly more). Let W_s be the set of nodes won by player i when playing according to σ_s. It is clear that

$$S = \bigcup W_s$$

Let $\prec$ be a total ordering of the strategies $\sigma_s, s \in S$. We now fix the strategy σ that wins every node in S as follows:

$$\forall v \in S : \sigma(v) = \sigma^*(v) : \sigma^* = \min_{\prec}\{\sigma_s : v \in W_s\}$$

Informally, we have fixed σ to always follow the smallest strategy according to $\prec$ that is winning for the current node.

The sequence of strategies we follow this way will be decreasing (a strategy winning for a node must also be winning for the successor node under that strategy) and thus eventually constant. As a finite prefix does not change the outcome of the game, the resulting game is winning for player i. □

Note that this theorem implies that the union if i-paradises is again an i-paradise.

Example

Figure 2.1 shows an example of a parity game. Player 0 nodes are drawn as circles, player 1 nodes as squares. The numbers inside the nodes represent their priorities.

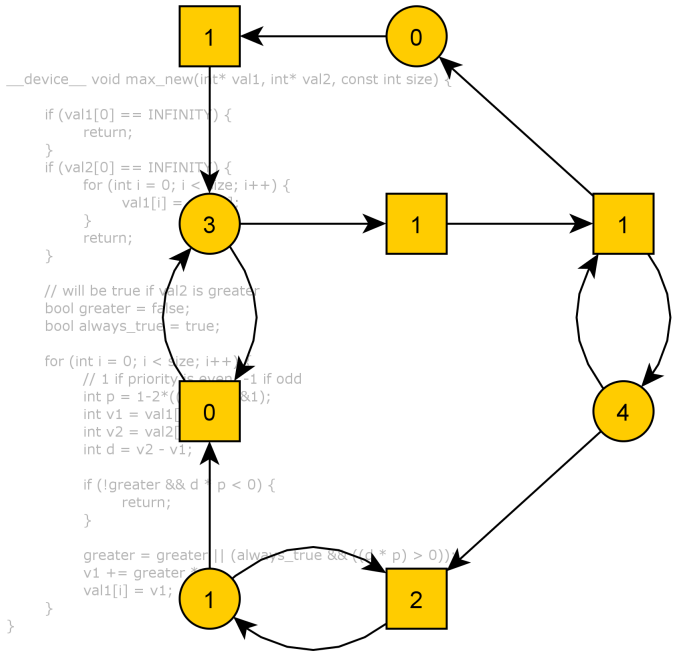

Figure 2.1: Example of a parity game

Finding a strategy for the players is easy in that case: Imagine the nodes are numbered from top to bottom row-wise, so the first row contains nodes 0 and 1, the second row contains nodes 2, 3, 4 and so on.

The circle of the nodes 0-1-2-3-4 is winning for player 1 as the highest priority is 3, thus player 0 will try to escape it. The only way he can do so is choosing the edge from node 2 to node 5. But then the circle 2-5 will be closed which is also winning for player 1.

On the other hand the circle 7-8 on the bottom is winning for player 0 and he can choose to close it by choosing the edge from 7 to 8. Moreover he can enter this circle from node 6.

By this observations, the winning set of player 1 consists of the nodes $\{0, 1, 2, 3, 4, 5\}$, player 2 wins the nodes $\{6, 7, 8\}$, the strategy choices of player 0 are $7 \to 8$ and $6 \to 8$, player 1 chooses $4 \to 1$.

2.1.3 Determinacy

Attractors and Traps

Given a set $S \subseteq G$, the i-attractor of S is the set of nodes from which player i can enforce a visit of S. Formally we define it inductively as follows:

$$pre(Y) = \{v \in V_i : vE \cap Y \neq \emptyset\} \cup \{v \in V_{(1-i)} : vE \subset Y\}$$

$pre(Y)$ is the set from which player i can enforce a visit to Y in the next step. Now to compute the i-attractor $\mathrm{Attr}_i(X)$, set $X^0 = X$ and for all $k \in \mathbb{N}$:

$$X^{k+1} = X^k \cup pre(X^k)$$

Definition 2.1.1. *Let $n \in N$ be the smallest number such that $X^n = X^{n+1}$. Then $Attr_i(X) = X^n$.*

Note that $\mathrm{Attr}_i(X)$ can be computed in polynomial time. If we want to emphasize that the attractor is computed in G we write $\mathrm{Attr}_i(G, X)$.

When computing the attractor of X, we can directly compute an attractor strategy that does enforce the visit of X. For a node $s \in \mathrm{Attr}_i(X)$ let the attractor rank be the smallest n such that $s \in X^n$ and $s \notin X^k \ \forall k < n$. Then for every node in $s \in \mathrm{Attr}_i(X) \backslash X$ choose as strategy an edge leading to some $s' \in \mathrm{Attr}(X)$ with lower attractor rank.

Definition 2.1.2. *A set $S \subseteq G$ is called an i-trap if*

$$\forall s \in S \cap V_i : sE \subset S \text{ and } \forall s \in S \cap V_{(1-i)} : sE \cap S \neq \emptyset$$

Intuitively, the other player $(1 - i)$ can force the token to stay in S thus "trapping" player i. Trivially, the complement of an i-attractor is an i-trap.

Now we can see why we can restrict ourselves to graphs without dead ends. Usually when finite plays are allowed, the player that cannot move any more loses. To compute those regions where one of the players loses in finite time, just find the 1-attractor to all dead ends for player 0 as well as the 0-attractor for all dead ends of player 1. Inside those attractors the respective player has the trivial attractor strategy which is winning for him. Since the complement of an i-attractor is an i-trap, we can simple remove the attractors without changing the rest of the game.

Memoryless determinacy

We say that a game is determined if for every node there is a (not necessarily memoryless) winning strategy for one of the players. Parity games with a given start vertex are determined. In fact, Martin showed the much more general result for Borel games. Borel games are games for which the winning condition is a Borel set, where the class of Borel sets is the smallest class of sets that contains the open sets and is closed under countable union and complementation. For a more in-depth definition of open sets and the Borel hierarchy see [13].

Theorem 2.1.3 (Martin)**.** *Every Borel game is determined [13]*

It can be shown that parity games are Borel games. Parity games are special in the sense that there is no memory needed while executing the winning strategy. This is called memoryless determinacy:

Theorem 2.1.4 (Memoryless determinacy)**.** *Parity games are memoryless determined*

The following proof is taken from [13]. We will show that in every parity game the vertices are partitioned into winning regions of player 0 and player 1 and each player has a memoryless strategy for his winning region.

The proof will be an induction over the maximum priority n in the parity game.

Lemma 2.1.1. *If the maximum priority occurring in G is 0, then G is partitioned into a 0-paradise and a 1-paradise.*

Proof. Since we assumed there are no dead ends, every game will be infinite. Player 0 will win every infinite game as the only priority occurring is 0. □

From now on we will assume the maximum priority to be greater than 0. By induction and Lemma 2.1.1 we will assume that Theorem 2.1.4 holds for all parity games with maximum priority less than n.

Induction step. Let $i = n \mod 2$ be the parity of the maximum priority, let W the union of all $(1-i)$-paradises (and thus the unique largest $(1-i)$-paradise). We will show that player i has a memoryless winning strategy for all nodes in $L = G \setminus W$. Observe that W is its own $(1-i)$-attractor (nodes in the attractor of W are also won by $(1-i)$ and therefore lie in W), thus L is a $(1-i)$-trap. We interpret the graph induced by L as a new game G'.

Let $N = \{v \in L : c(v) = n\}$ be the set of vertices with maximum priority in G' and Y its i-attractor in G':

$$Y = \mathrm{Attr}_i(G', N) \tag{2.1}$$

Because L is a $(1-i)$-trap, any attractor strategy of player i in G' is also an attractor strategy in G: Edges from L to W always start in player i nodes and thus adding W to G' to obtain G cannot change the attractor.

Let furthermore $Z = L \setminus Y$ be the complement of Y in L. Again we interpret the graph induced by Z as a new parity game H. The situation so far is depicted in Figure 2.2, the grey area is L.

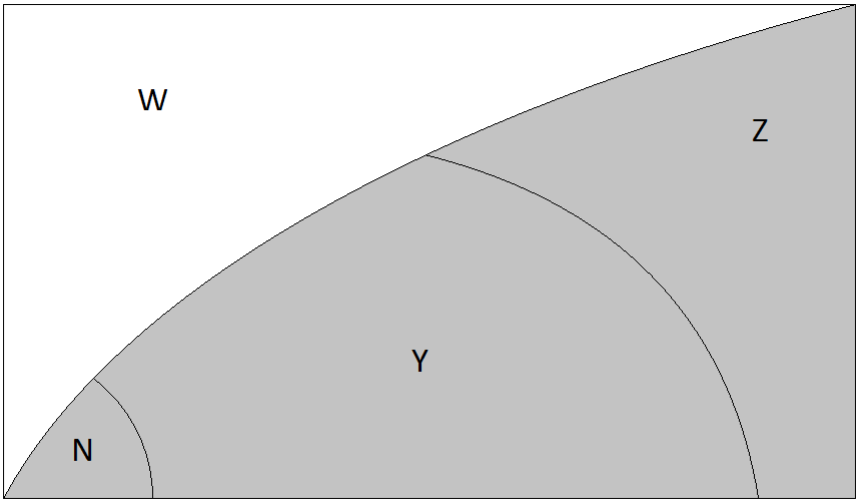

Figure 2.2: The memoryless strategies from [13]. The grey area is L.

Since $Z \cap N = \emptyset$, H has fewer than n priorities, thus we can apply Theorem 2.1.4 and will find a partition into paradises Z_0, Z_1 (the corresponding winning strategies are winning in H). We now show that $Z_{(1-i)} = \emptyset$:

Since Z is the complement of an i-attractor in G', it is a i-trap in G'. Thus if $Z_{(1-i)}$ is a non-empty winning region of player $(1-i)$ in H, player $(1-i)$ also wins $Z_{(1-i)}$ in G' with the same strategy. But then he also wins $Z_{(1-i)}$ in G as the play will either stay in $Z_{(1-i)}$ or at some point enter W where he wins all nodes. This contradicts the assumption that W is the winning region of player $(1-i)$.

We now claim that player i has the following winning strategy on L: In $Z = Z_i$ follow his winning strategy, in $Y \setminus N$ the attractor strategy, in N choose any successor that lies in L. Indeed the game will either enter Y and thus N infinitely often or from some point on stay in Z_i. Both possibilities are winning for player i. □

2.1.4 Complexity setting

Parity games are very interesting from the complexity theoretic point of view. There are some natural problems connected to parity games: Deciding which player wins from a given node, finding the winning sets of the players and finding winning strategies (for a node or the winning set). The first two problems are of course interreducible: Given the winning sets it is easy to decide which player wins a given node. And given an algorithm that decides which player wins a given node, one can easily construct the winning sets.

It has been shown that deciding the winner of a given node is in NP ∩ co-NP. There are even tighter bounds:

Theorem 2.1.5. *Deciding the winner of an initialized parity game is in UP ∩ co-UP [11]*

UP is the class of problems accepted by non-deterministic unambiguous turing machines in polynomial time. Unambiguous means that if the turing machine accepts, there is exactly one accepting path. So far, no deterministic polynomial time algorithm for

deciding the winner is known. Furthermore, parity games are the easiest (in a complexity theoretic sense) of four games all known to be in UP ∩ co-UP: It has been shown that parity games reduce to mean payoff games, these reduce to discounted payoff games and these finally reduce to simple stochastic games [3]. Therefore, if some or all of these games are solvable in polynomial time, it is likely that parity games will be the first of them to be solved.

2.1.5 Modal μ-calculus

The main application for parity games is in decision procedures for μ-calculus. Deciding acceptance and non-emptiness for alternating tree automata(ATA) can be reduced to deciding the winner of a parity game [7]. These automata have a strong connection to μ-calculus formulas: Model-checking of a μ-calculus formula can be polynomial-time reduced to deciding acceptance, synthesis reduces (also in polynomial time) to deciding non-emptiness of an ATA.

2.2 Strategy iteration

Strategy iteration is an approach of finding winning strategies for the players on their respective winning regions by iteratively improving a strategy until it is optimal. Strategy iteration was first used to solve stochastic games and discounted payoff games. Because of the structure of these games, real numbers were used. Jurdziński and Vöge came up with a much more intuitive discrete strategy improvement algorithm [19]. Since then, many adaptions of that algorithm appeared, all with the goal of better worst-case time bounds. The most notable idea, introducing the possibility to "give up" for one or both players, is used in [1], [12], and [15] and will be discussed below. This section gives a brief overview over a generic approach to strategy iteration as well as two specific implementations, the one by Jurdziński and Vöge and an adaption of the algorithm due to Björklund, Sandberg and Vorobyov [1] (see [12],[14]) .

2.2.1 Generic strategy iteration

Jurdziński and Vöge [19] give a very generic characterization of strategy improvement algorithms that will be presented here. It is basically a fixed-point iteration that starts with a strategy for one player and improves it in every iteration step until finally the best possible strategy is obtained. To achieve that goal we need an ordering of the strategies, given as a pre-order, and some way to enhance the current strategy.

The pre-order $\sqsubseteq$ on the set $Strategies_0$ of strategies for player 0 has to satisfy:

P1 There is a maximum element in the pre-order, i.e. there is a strategy σ such that $\kappa \sqsubseteq \sigma$ for all strategies κ

P2 If σ is a maximum element in the pre-order, then σ is a winning strategy for player 0 (from every vertex of the winning set)

This already simplifies the search for the optimal strategy as we only have to find a maximum element with respect to $\sqsubseteq$. Finding that element is done using an *Improve* operator of the following form:

$$Improve : Strategies_0 \to Strategies_0$$

satisfying the following postulates:

I1 If σ is not a maximum element in the preorder, then $\sigma \sqsubseteq Improve(\sigma)$ and $Improve(\sigma) \not\sqsubseteq \sigma$

I2 If σ is a maximum element in the pre-order, then we have $Improve(\sigma) = \sigma$

Now the algorithm for finding the winning strategy on the winning region is quite simple:

```
pick a strategy σ for player 0
while σ ≠ Improve(σ)
        σ = Improve(σ)
```

The difficulties arise when trying to find such a pre-order and *Improve* operator and proving that they indeed satisfy the constraints. Both algorithms presented here use the concept of valuations. These work as follows: For a fixed strategy a value is assigned to each node that intuitively holds information about how good this node is for player 0. The pre-order is then defined by point-wise comparison, that is, a strategy is better than another iff for every node the value is at least as good as for the other strategy and better for at least one node. Improving a strategy usually is realized by switching to a successor improving the current valuation (see [2]) or more general by choosing a subset of improving successors(see [12]). Luttenberger showed in [12] that selecting all improving successors yields Schewe's locally optimal improvement [14]. It is known that there always exists improvements such that after $|V|$ steps the optimum is reached [19], but no algorithm to find those improvements is known.

In some sense strategy improvement and the search for better *Improve* operators can be compared to the simplex algorithm and the choice of pivot elements when optimizing over a polytope. Both algorithms have worst case exponential running time in all known variants [5], but they perform really well in practice (see [17] and our results). For linear programming however, a polynomial time algorithm is known (that works in a completely different way than simplex) whereas for parity games this is not the case.

2.2.2 Jurdziński and Vöge

Jurdziński and Vöge [19] also give an implementation of this generic approach that will be briefly presented here. They use the notion of finite cycle domination games, finite games equivalent to parity games. Since memoryless strategies suffice for parity games, once a cycle is closed the game can be ended because exactly the nodes in the cycle will be visited infinitely often. Thus one only has to evaluate the cycle to determine the winner.

The valuation is computed using the following fact: Fixing deterministic strategies for both players one can immediately tell who wins a given node. Given two strategies, the

value of a node v is defined as a triple $(\lambda, \pi, \#)$ where λ is the maximum priority in the cycle of the play starting in v, π is the set of priorities greater than λ encountered on the way to the cycle and $\#$ is the length of the path from v until λ is reached.

A valuation of a strategy σ of player 0 is then created by solving the one player game of player 1 when strategy σ is fixed for player 0, thus finding the best counter-strategy for player 1, and then determining the value for the nodes. Finding the counter strategy and computing the values of the nodes can be done in one step using an adapted shortest path algorithm.

2.2.3 Generalization/Refinement of Jurdzińsky and Vöge

As mentioned earlier, one important change that leads to a much more compressed representation of values is the idea to allow players to end a game they would lose by "giving up" which is usually represented by a special sink node. This idea is used in [1] and [14] and refined for parity games in [12]. Because of the compact valuation and the good complexity bound in [12] this algorithm is the one implemented in this bachelor's thesis. There is also another reason why we choose this algorithm: Schewe's algorithm uses a bipartite graph, the algorithm of Jurdzińsky and Vöge assumes that every color is used at most once. The algorithm we use does not suffer from such a restriction. Formally the algorithm looks as follows:

For a parity game $A = (V, E, o, c)$ we introduce the sink game $A_\perp = (V_\perp, E_\perp, o_\perp, c_\perp)$ where $V_\perp = V \cup \{\perp\}$, $E_\perp = E \cup (V_1 \times \{\perp\})$, $o_\perp = o \cup \{\perp\}$, $c_\perp = c \cup (\perp, 0)$ although color and ownership of $\perp$ are of no importance.

We call a cycle 1-dominated if the highest priority occurring in this cycle is odd. Our algorithm will start with a strategy $\sigma_\perp$ that contains no 1-dominated cycle and every *Improve* step will maintain this fact as an invariant. Because we only fix strategies for player 0 we cannot prevent player 1 from playing winning self cycles (cycles consisting only of his own nodes), thus we remove them in a polynomial time preprocessing step.

We will again compute a valuation for the current strategy that assigns values to each of the nodes and then improve the strategy using the computed values. The value of a node will be given as a color profile, that is, an integer vector of size d, where $d - 1$ is the greatest priority used, or one of $\{-\infty, \infty\}$. There is a total ordering on those values given as follows:

A value p is smaller than p' (we write $p \prec p'$) if the highest index i where they differ is even and $p_i < p'_i$ or the highest index i where the differ is odd and $p'_i < p_i$. We write $p \preceq p'$ if p is smaller or equal than p'. This ordering tries to capture the intuition that even priorities are good for player 0, odd ones are not, and greater priorities are more important than lesser ones. Additionally we set ∞ as the maximum element of $\prec$, $-\infty$ as the minimum element. The set of all color profiles will be denoted by $\mathcal{P}$.

We will sometimes add a single priority to a color profile. This is defined as increasing the corresponding coordinate of the vector by one, or, if the profile is one of $\{-\infty, \infty\}$, not changing it at all. Such an addition will be written as $s + p, s \in V, p$ a color profile, where we add the priority of s to p.

The value of a node for a fixed strategy is the best possible outcome player 0 can guarantee when starting from that node. Because of our preprocessing step and the possibility to give up, player 0 will never run into a losing cycle, thus values will either be

∞ if player 0 wins from this node, or some vector that represents a final play that ends in the sink. These values can be computed as follows:

For a strategy σ (for which the graph contains no 1-dominated cycles) define $\mathcal{V}_\perp : V \cup \{\perp\} \to \mathcal{P}$ as $\mathcal{V}_\perp(\perp) = (0, \ldots, 0)$, $\mathcal{V}_\perp(v) = \infty \; \forall v \in V$.

Then the valuation is given as the fixed point of the following operator when starting from $\mathcal{V}_\perp$

$$
\begin{aligned}
&F_\sigma[\mathcal{V}_\perp](\perp) = (0, \ldots, 0) \\
&F_\sigma[\mathcal{V}_\perp](s) = s + \min_{\prec}\{(V)(t) : (s,t) \in E_1\} \text{ if } s \in V_1, \\
&F_\sigma[\mathcal{V}_\perp](s) = s + \max_{\prec}\{(V)(t) : (s,t) \in \sigma\} \text{ if } s \in V_0
\end{aligned}
$$

Improving the strategy is realized by using "all profitable switches":

$$Improve(\sigma) := \{(s,t) \in E_0 : \mathcal{V}_\sigma(s) \preceq s + \mathcal{V}(t)\}$$

It can be shown that by improving the strategy this way, no 1-dominated cycles will be introduced. As starting strategy without 1-dominated cycles we choose $\sigma_\perp$ as the set $V_0 \times \{\perp\}$, that is, player 0 gives up immediately.

Using this strategy improvement algorithm the optimal strategy will be found after at most $|V| \cdot (\frac{|V|}{d} + 1)^d$ iterations where one improvement step needs $O(|V| \cdot |E|)$ time.

2.3 Other Algorithms

There are numerous other algorithms for solving parity games that so far all exhibit super-polynomial running time. Some of them will be presented below. We will always use n as number of nodes, d as number of priorities and e as number of edges to describe the running time.

The recursive algorithm by Zielonka [20] decomposes the parity game into multiple smaller games in a manner closely related to the proof of Theorem 2.1.4. The winning strategy is always composed of winning strategies of some smaller games as well as attractor strategies. It has a worst case time complexity of $\mathcal{O}(e \cdot n^d)$.

The strategy improvement algorithm by Jurdziński and Vöge discussed earlier exhibits a running time of $\mathcal{O}(2^e \cdot n \cdot e)$.

The optimal strategy improvement method by Schewe [14] improves the process of choosing a better strategy and thus improves the worst case time bound to $\mathcal{O}(e \cdot (\frac{n+d}{d})^d \cdot \log(\frac{n+d}{d}))$.

The small progress measure algorithm by Jurdziński [10] uses the notion of progress measures. Those assign tuples of values to each node dependent on the node's priority, owner and successors in a specific manner. Because of the way values are assigned, progress measures can only exist on the winning set of player 0. It can be shown that a finite domain of these values suffices to express such a measure. To find the winning regions, the algorithm starts with tuples of 0 assigned to every node and iteratively increases those values violating the progress measure relation. The part of the game where the progress measure is successfully created is then won by player 0, on the other part the values will exceed the precomputed bounds and consequently those nodes will be won by player 1. Worst case time complexity: $\mathcal{O}(d \cdot e \cdot (\frac{n}{d})^{d/2})$.

There also exist multiple randomized algorithms for solving parity games. As computing winners of nodes given a strategy for one player is possible in polynomial time, a guess-and-check algorithm can just randomly select a strategy and compute the set it is winning on. This is repeated until the game is partitioned into winning regions.

The randomized strategy iteration of Björklund, Sandberg and Vorobyov [1] is an adaption of strategy iteration working on mean payoff games. Parity games can be reduced to mean payoff games in polynomial time[11]. This algorithm achieves a time bound of $\min(\mathcal{O}(n^4 \cdot e \cdot d \cdot (n/d+1)^d), 2^{\mathcal{O}(\sqrt{n \log n})}$

Chapter 3

PGSolver

3.1 Overview

PGSolver [6] is a tool developed by Friedmann and Lange that serves as a benchmarking tool for different algorithms to solve Parity Games. It contains implementations of about 10 algorithms including Zielonka's recursive algorithm [20], the strategy-improvement algorithms due to Jurdziński and Vöge[19] and due to Schewe [14], as well as algorithms solving parity games via reduction to discounted payoff games. There is also a reduction of parity games to SAT formulas due to Friedmann. Moreover, PGSolver offers the possibility to integrate user-written solvers (written in OCaml) into PGSolver.
For benchmarking purposes, PGSolver also brings tools to generate several different parity game instances of arbitrary size, especially worst-case examples for the solving algorithms provided. This makes it a very useful tool for comparing not only Landau-Notation but also real running times on multiple examples. In our work, PGSolver is used to provide example games for benchmarking as well as verifying our solutions. Furthermore, computation times of our implementation and PGSolver are compared to each other in benchmarks.

Section 3.2 will cover optimizations used in PGSolver and is intended to give ideas for possible improvements that can generally be applied when solving parity games. Section 3.3 will show some of the games provided by PGSolver that were used for benchmarking our implementation.

3.2 Optimizations

PGSolver offers many optimizations to provide better running times. Those optimizations are not part of our implementation. Some of them will be described below.

3.2.1 SCC Decomposition

Using for example Tarjan's algorithm, one can decompose the game graph into strongly connected components. When collapsed to a single node each, the resulting graph forms a directed acyclic graph. A SCC S is called proper if it actually contains an edge (that is, there are not necessarily distinct nodes $u, v \in S$ such that $(u, v) \in E$). It should be clear

that those SCCs which have no exiting edges can be solved without knowledge of the rest of the game.

Once the winning sets for those SCCs are computed, one can compute the attractors for the respective player and remove them from the game. In this process, some SCCs might be "damaged" and can be further decomposed. Applying the same steps repeatedly will lead to an algorithm solving the whole game by breaking it down into multiple smaller games.

3.2.2 Special case solving

Some types of games can be solved very efficiently by applying polynomial-time preprocessing algorithms. We can assume the game to be a single proper SCC. Improper SCCs are only single nodes for which the winner can be computed by solving the successor SCCs. There are several special cases with easy solutions:

One-parity games

If all nodes in a proper SCC have the same parity, trivially the corresponding player wins no matter what. Any strategy suffices here.

Self-cycle games

Suppose there is a cycle of player i nodes where the highest priority is winning for player i. Then clearly he wins on all nodes of this cycle and its attractor. Such cycles can be found in the following way:

1) restrict the graph to player i nodes: $G' = (G \cap V_1, E \cap V_1 \times V_1)$

2) for each node $s \in G'$ check if s is reachable from s in G' using only nodes with priorities not greater than $c(s)$

Step 2) can be done with an altered DFS and thus the algorithm runs in polynomial time. In the same way, one-player games can be solved. (A game is called a one-player game if all nodes of one of the players have out-degree 1)

Priority compression

The number of priorities in a parity game (or, more precisely, the maximum priority used) has, for most current solvers, a direct effect on the running time of those solvers. Trying to reduce this number is therefore a natural approach to speed up the solving process. One important idea to do so is the following:

Suppose there is no node with priority c. Then the outcome of the game will not change if we replace every occurence of $c+1$ by $c-1$. Thus we can lower all priorities greater than c by 2, reducing the maximum priority by 2.

This approach can be repeated until every priority between 1 and the maximum priority is used. (Note that we cannot guarantee that 0 is used.)

Priority propagation

Since our plays are infinite, after visiting a node s, one of its successors will always be visited. This fact can be used to reduce the priorities:

If all successors of s have greater priorities than s itself, the outcome of the game does not change if we change the priority of s to the lowest priority of its successors. In the same way, if all predecessors of s have greater priorities, we can change the priority of s to the lowest priority of its predecessors.

3.2.3 Special parity games

PGSolver not only provides tools for solving parity games, but also delivers multiple possibilities to generate parity games. Two of those that were used in our Benchmarks shall be presented here.

Jurdziński games

This family of games was introduced by Jurdziński [10] in order to show the exponential worst case behavior of his small progress measure algorithm. It is defined using two parameters, $l, b \in \mathbb{N}$. The game $J_{l,b}$ consists of l "levels" that contain b "blocks" each. One of those levels is called "odd", the other $l - i$ are called "even". The block building the odd level looks as follows:

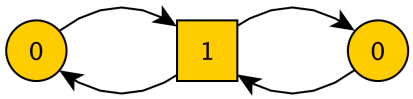

Figure 3.1: Odd building block

The blocks for the k-th even level are of the following structure:

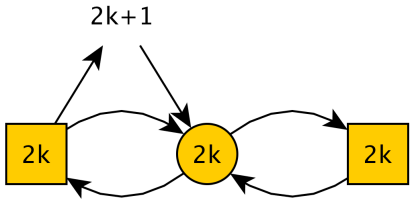

Figure 3.2: Even building block

Those levels then are connected by adding edges from every even level to the odd level, connecting the player 0 nodes from the even level to the player 1 vertex of the odd level and vice versa. See Figure 3.3 for an example.

Elevator verification games

These games are created by encoding an elevator system of n floors and a fairness constraint into a μ-calculus formula and then converting this formula into a parity game.

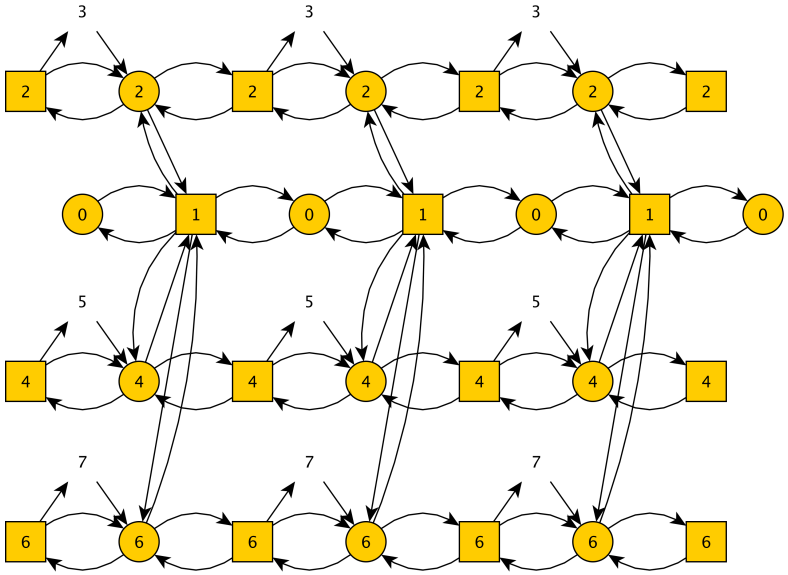

Figure 3.3: The game $J_{4,3}$

They grow in size very fast but have constant maximum parity of 3. The elevator state is modeled as a combination of the floor it is in, its door state and a list of requests. The door can be either open or closed. The elevator will always move towards the next request, open its door if a floor that was requested is reached (which removes this request), and handle new requests in a FIFO (first in, first out) principle. The fairness constraint states that a requested floor will eventually be visited.

Random games

Two types of random games were used in our benchmarks, denoted by random games and clustered random games.

Random games are dependent on 4 parameters: Number of nodes, highest priority, minimal and maximal outdegree. The game is then generated by randomly adding edges as long as there is a node violating the lower bound. This creation scheme leads almost always to a graph consisting of one big SCC.

To avoid this behavior, clustered random games are created in a much more sophisticated fashion: They depend on nine parameters including node count, highest priority, recursion depth r and breadth b as well as an interconnection rate x. If recursion depth is zero, the game created equals a random game. Otherwise, for recursion depth r, b many clustered random games of recursion depth $r-1$ are created such that the combined node count of them is equal to the nodes desired for the final game. Those games are then connected by x many edges.

Some of the explanations above are simplified, for example the recursion breadth of clustered random games does not have to be a constant but can be given as a range from which it is then chosen uniformly at random.

Chapter 4

NVIDIA CUDA

4.1 Introduction

Graphics cards are powerful computation devices used in many applications like physics simulation, medical imaging and stock market analysis. By design, the GPU has immense parallel computation power. In this chapter we have a look at the architecture behind GPU programming. Section 4.2 introduces the basics about threads and how they are processed. Section 4.3 consists of a simple example CUDA code. Section 4.4 shows effects of branching inside warps. In Section 4.5 the specifics of memory usage in CUDA are discussed. Section 4.6 points out important things to think about when optimizing CUDA code. A more detailed description of CUDA can be found in [4].

4.2 Kernels, threads and multiprocessors

Latest developments in CPUs move away from more and more powerful single-core CPUs and instead focus on multi-core setups. GPUs were designed for parallelism right from the beginning and for that reason contain much more cores. The basic unit in GPUs are the so called "Streaming Multiprocessors". A multiprocessor can hold up to 196 CUDA cores and manages several hundreds of parallel threads. To efficiently manage the threads, they are split up into warps of 32 threads each and then scheduled for execution by the warp scheduler. The execution can only happen concurrently if all threads in one warp execute the same instructions; if that is not the case, the threads that diverge are executed serially until they converge again. This architecture is called Single-Instruction, Multiple Thread architecture. An example for this can be found in section 4.3

Every piece of code that is to be executed on the GPU must be structured into kernels and is then run in parallel by multiple threads. A typical application can launch tens of thousands of threads. For easier handling of such enormous numbers CUDA structures them into grids and blocks as follows:

- Threads are grouped together into blocks that can be 1-, 2- or 3-dimensional. Up to 1024 Threads can be grouped into one block. All blocks have the same size specified by the programmer
- The blocks are then arranged in a 1-, 2- or 3-dimensional grid.

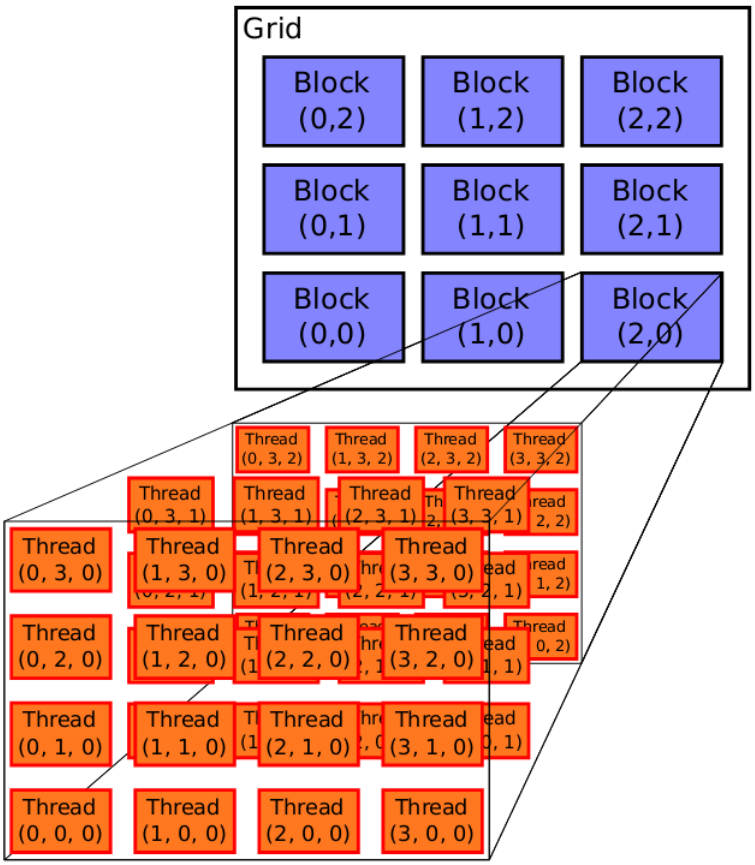

Figure 4.1: Example grid with 3x3 blocks of dimension 4x4x3 taken from [18]

All threads then execute the same kernel. Different data can be processed via the threadIdx and blockIdx which are individual for every thread. (See example below)

Each block is then passed to a Multiprocessor and processed there. Up to 16 Blocks can reside in one Multiprocessor concurrently. It is important to note that a kernel call will immediately return the control to the CPU thus making it possible to use the CPU otherwise until the kernel's computation is finished.

4.3 CUDA Example

This example shows the important aspects of a GPU implementation to multiply a vector with a given scalar. The complete example can be found in the appendix, Listing 7.1. Let us first look at an implementation for the CPU:

```
void mult_cpu(int* vec, int mult, int size) {
        for (int i = 0; i < size; i++) {
                vec[i] *= mult;
        }
}
```

In CUDA, instead of a loop we can assign a single thread to each of the elements of the vector and do all the work in parallel. The kernel then looks as follows:

```
// __global__ marks this as a kernel
__global__ void mult(int* vec, int mult) {
        // every thread processes the element according to his ID
        vec[threadIdx.x] *= mult;
}
```

Here we assume that our block has only one dimension (x). Therefore we only need to take threadIdx.x into account. Inside our main function, we need to specify how many threads we need. For simplicity we just choose *size* many threads in a grid of size 1. Here we want to multiply by 3.

```
// call the kernel with 1 block of size many threads
mult<<<1, size>>>(vec_D, 3);
```

The vector passed is named vec_D and points to some location in the device memory. Before calling, we need to copy our vector that is stored in the host memory(e.g. RAM) to the graphics card:

```
// declare the device pointer
int* vec_D;
// allocate device global memory
cudaMalloc(&vec_D, size*sizeof(int));
// copy the array to the device
cudaMemcpy(vec_D, vec, size*sizeof(int), cudaMemcpyHostToDevice);
```

After the kernel is finished, we use cudaMemcpy again to get the result back to the CPU.

Because a thread block can only contain 1024 threads, this code will not work for bigger vectors. Instead we need to split the work into multiple blocks and then address the vector inside the kernel not only by threadIdx, but also by blockIdx.

4.4 Divergence

We now have a look at divergence and how it is handelled internally. Consider the following kernel:

```
__global__ void even(int* vec) {
        if (threadIdx.x % 2 == 0) {
                vec[threadIdx.x] += 1;
        } else {
                vec[threadIdx.x] += 2;
        }
}
```

and assume for simplicity that every line can be executed in one clock cycle. The execution is depicted on the left of Figure 4.2. Now let us change the code as follows:

```
__global__ void even(int* vec) {
        vec[threadIdx.x] += 1 + (threadIdx.x % 2)
}
```

This kernel does exactly the same as our first kernel, but the execution, that is depicted on the right of Figure 4.2, is faster. It is of course not always possible to completely eliminate branching but one should try to keep branching inside warps to a minimum. As a worst case example imagine one thread per warp doing some extremely expensive operation. This leads to a slowdown factor of 32 compared to grouping those expensive operations together and execute them in parallel (provided the expensive operation is always the same).

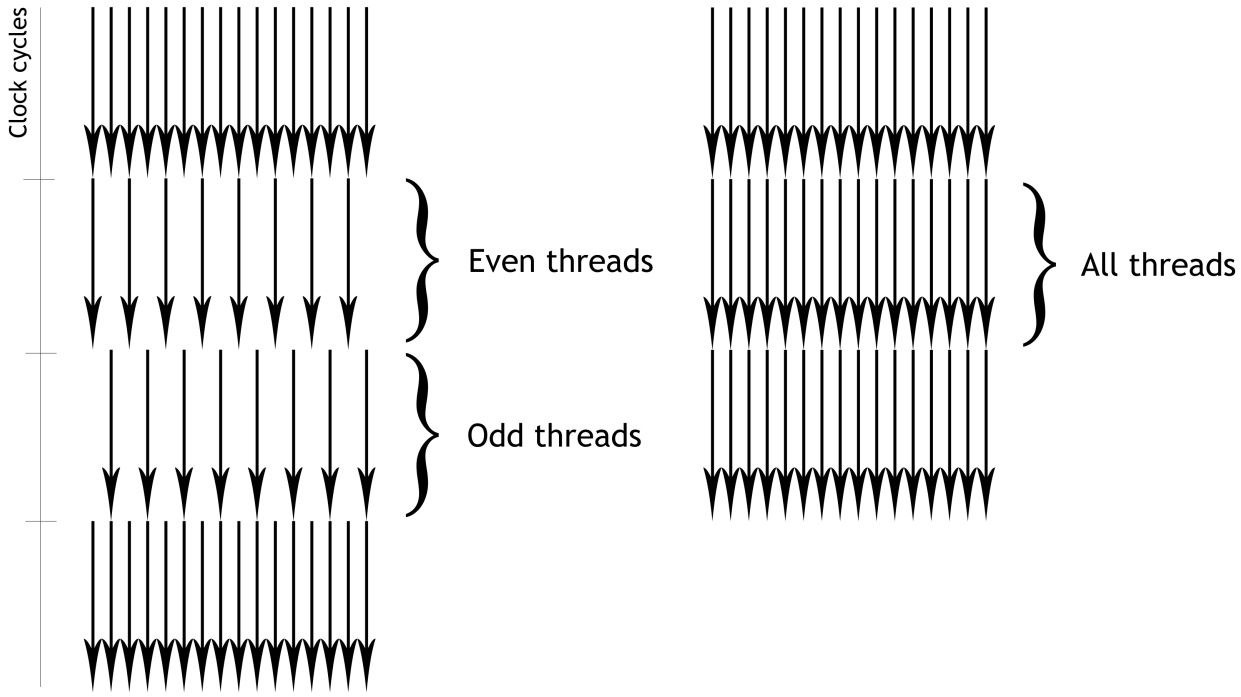

Figure 4.2: Execution using divergent (left) and non-divergent(right) kernel

4.5 Memory layout

While good memory layout can positively influence any application, the GPU suffers even more from bad layout than for CPU. Because of high latency and other restrictions, it is extremely important to choose data structures and program code around memory usage. There are four types of memory on a graphics card and the choice which one to use has a big influence on running time. The GPU cannot access data that is not stored on the graphics card. Therefore the CPU has to transfer the data needed for the computation using CudaMemcpy. The four types of memory are:

Global memory

The RAM-like memory of the graphics card with size of up to 4 GB. A good bandwidth ($>$ 100GB/s) can be achieved on this memory using specific access patterns, but the

latency is quite high. A small part of global memory is dedicated to texture memory and constant memory. Both serve as a read-only memory that is automatically cached. (Read-only means the data has to be written by the CPU via CudaMemcpy.) All threads can access global memory.

Data access is extremely fast in global memory when the access is coalesced, that means, the adresses required by the different threads lie in the same 128-bit fragment of the global memory. Such 128-bit fragments can be loaded into registers in one step rather than copying the data for each thread individually. According to NVIDIA, latency can be as high as 800 clock cycles. [4]

Local memory

Local memory is the private memory of each thread where specific data like large structures that would consume too much register space are stored. It is also used in case of register spilling, that is, if a thread uses more registers than available. Local memory is located off-chip in the device memory and thus underlies the same restrictions as global memory (high latency, coalesced access)

Shared memory

Shared memory is a very fast on-chip memory that can be accessed by all threads of one block. The latency is much lower than for global memory, but the size available is at most 48KB per multiprocessor. It can be seen as a "user managed cache". A restriction for shared memory performance are the bank conflicts: Shared memory is divided into 32 so called "banks". When two threads access different data in the same bank, their access is serialized. More information about bank conflicts can be found in [4]

Registers

Registers work the same way as for CPUs as an on-chip memory for the current computations and local variables of each thread. They are very fast, but also very limited in size. At most 64000 32-bit registers per multiprocessor are available, thus at 2000 threads residing on one multiprocessor (which is the maximum) each thread has access to about 32 registers.

4.6 Optimizing CUDA code

There are several possible bottlenecks that can slow down CUDA applications to the point where using the GPU is actually slower than doing the computation on the CPU. Some of them will be discussed below.

Branching

As mentioned before, threads will be grouped into warps of size 32 that will be executed concurrently. It is extremely important to avoid branching inside such warps as this will reduce performance drastically as seen in section 4.4. A single divergent thread can

already affect the performance achieved for that warp drastically. Thus the code should be structured in a way to avoid branches inside warps.

Memory usage

The second big factor is memory usage. While the programmer cannot decide to store data on registers or local memory, he has direct access to texture memory, constant memory, global memory and shared memory.

Texture memory offers different addressing and caching modes as well as interpolation between data points. This can be very helpful and speed up the computation, but if the interpolation is not needed it may actually lead to worse performance than global memory.

Constant memory also offers caching for data that remains the same during the kernel launch. Total amount of constant memory is 64KB.

Shared memory is the fastest type of memory the programmer has direct access to. As for constant memory, shared memory is restricted in size: 64KB of shared memory can be used per multiprocessor. It is advisable to copy data that will be used multiple times from global to shared memory.

Register usage

The programmer can not directly influence which data is stored on registers, but he should try to avoid cluttering them with unnecessarily many local variables. Since every multiprocessor has a limited register count, the amount of registers used per thread will directly influence how many threads can reside on one multiprocessor and thus the computation speed. It is often not trivial how to reduce the number of registers used as even seemingly insignificant changes like the point of declaration of variables can change the amount of registers used.

Coalesced access

Another important point is the access pattern on global memory. As mentioned before, data from global memory is accessed in blocks of size 32, 64 or 128 bit. Reducing the number of such transfer operations by accessing data blockwise will enhance performance greatly. Data structures should be chosen such that access always happens in a coalesced fashion if possible.

Chapter 5

Implementation

5.1 Introduction

This chapter covers our implementation of strategy iteration in CUDA. In Section 5.2 we discuss some CUDA specific details of our implementation. In Section 5.3 we evaluate the performance of our implementation by means of benchmarks. In Section 5.4 we present further improvements.

5.2 Implementation Details

5.2.1 Data structures

Below you find a more detailed description of the data structures used for representing the game. We use two representations of the parity game: The first one is based on the Boost Graph Library[16] which allows us to make use of the already existing implementations of Tarjan's strongly connected components algorithm, the other one is tailored towards the GPU.

Boost Graph

The struct used to represent a parity game with the Boost Graph Library (BGL)[16] looks as follows:

```
typedef boost::adjacency_list<boost::vecS, boost::vecS,
                boost::bidirectionalS> boost_graph;

typedef struct bg{
        boost_graph* g;
        list<int>* v0;
        list<int>* v1;
        vector<int>* priorities;
        vector<bool>* partition;
        bg();
        bg(int n);
        void add_vertex(int index, bool owner, int priority);
```

```
} boost_graph_S;
```

The upper typedef defines which representation inside BGL is used, namely a bidirectional adjacency list with vectors as containers for both edges and nodes.

The struct then consists of the boost graph as well as additional information: The vertex owners (in form of the boolean vector partition), the priorities and two lists v0 and v1. These lists are mainly used for the conversion to the array graph discussed in the next section. During this conversion nodes are removed. To prevent BGL from re-labelling the vertices they are not actually removed from the graph but only from the list v0 or v1.

Since CUDA can handle only the C fragment of C++, boost graph is not suitable for use inside of kernels.

Array Graph

The second struct used consists of data structures that can be directly passed to any kernel. It looks as follows:

```
typedef struct{
        int node_count;
        int priority_count;
        int* edges;
        int* edge_borders;
        int* priorities;
        bool* partition;
        unordered_map<int, int>* old_to_new;
        unordered_map<int, int>* new_to_old;
        int edge_count();
        void print();
        void toFile(string filename);
} array_graph;
```

Here the information that was previously stored only implicitly is now directly accessible as node_count, priority_count, edge_count(). The graph is represented as an array of edges and the corresponding edge_borders. The interpretation is the following:

To represent the graph we use a very compact adaption of an adjacency list: We store all the edges in a single array, edges. We use a second array, edge_border, to mark the which edges belong to which node. The edges with indices edge_borders[n] to edge_borders[n+1]−1 belong to the node n. This representation is very suitable for CUDA because its compactness reduces memory copies and it is traversable efficiently.

Additionally the priorities and partition are stored. Since the conversion may re-label some nodes, a mapping between the labels before and after is stored (and vice versa).

Parsing a game

The process of parsing a game given by PGSolver is the following: First the game is converted into a Boost Graph. In this Boost Graph some preprocessing is applied to guarantee the properties needed by our algorithm (no winning 1-cycles). Then the graph is converted to an Array Graph in a fashion to optimize it for CUDA code. This is achieved through rearranging the nodes to make coalesced access more likely. (see Section 5.3.3)

5.2.2 Kernels

Now we present the kernels implemented.

Initialize and init_valuation

These two kernels just handle basic initialization such as setting all values to infinity. Because we use integer arrays that can only store a finite domain of numbers, we had to represent infinity by some fixed number. Thus we used the fact that color profiles are always non-negative and represent an infinite value by setting the first entry of that value to -1.

f_sigma

This kernel is the implementation of the F_σ-operator discussed in chapter 2. It exists in multiple versions for different reasons:

f_sigma and f_sigma_big: CUDA does not allow dynamic memory allocation inside the kernel. Therefore there are two versions implemented that are basically equal except for the array used to temporarily store valuations. f_sigma can handle valuations with up to 128 priorities, f_sigma_big makes it possible to use up to 1024 priorities.

f_sigma_shared uses shared memory. It shows the basic idea of repeatedly copying parts of the valuation into shared memory because there is not enough space to copy the complete valuation.

update_strategy

Here the strategy improvements are computed. It works according to the "all possible improvements" idea.

Device functions

For comparing and max/min of valuations, the functions in dev_functions.cu were used. These functions are called very often and thus making them as efficient as possible is an important part of optimizing the code.

5.3 CUDA related design

This section is intended to show in more detail the design choices and experiments we made and whether they did work out or not.

5.3.1 Kernel launch

In our first implementation a single kernel would handle the complete procedure of evaluating and updating the strategy. This kernel was launched with one block of 1024 threads. Because of the architecture of CUDA GPUs, only a small proportion of the computational power (only one multiprocessor) was used and thus the program ran very slowly. When trying to launch the kernel with multiple blocks so that it can be distributed across all

multiprocessors, we encountered one of CUDA's restrictions: Threads cannot be synchronized across blocks. The only way to synchronize all threads in the different blocks is to split the task into multiple calls to (not necessarily different) kernels. Therefore now the code is split up in parts between which a synchronization is necessary and the kernels are launched with multiple blocks and at least one thread per node. The execution now proceeds as follows:

```
initialize
repeat {
        init_valuation
        repeat{
                f_sigma()
                cudaThreadSynchronize()
        } until (valuation does not change)
        update_strategy()
        cudaThreadSynchronize()
} until (strategy does not change)
```

5.3.2 Code development

We now turn to how the CUDA code developed over time and what optimizations we found.

Kernels

Of course all the kernels changed when we moved from one block to multiple blocks with one thread per node. Before the threads would loop through all the nodes, after the change every thread processes exactly one node. There were also other changes we made over time:

f_sigma did change mainly to remove branching whenever possible. This includes combining min and max into one function which is discussed below. Another change to optimize memory access was to use the intermediate array val as long as possible and copy it to global memory only after everything else is finished.

One of our ideas was to enhance f_sigma using shared memory to store the valuation. It did not work out as expected and is discussed in "other ideas" below.

update_strategy did not change at all except for the function compare that is part of the device functions discussed below.

Device functions

Two versions of the function that maximizes valuations, max_old and max_new, as well as the final version, now named update, can be found in Listings 7.2, 7.3, 7.4[1] in the appendix. max_old and max_new result in exactly the same outcome but max_new is

[1]You might notice the seemingly unnecessary variable always_true which has, as its name states, always the value true. Despite we cannot think of any logical reason why removing this variable should change anything, the code produces different (false) results when it is removed, replaced by true or marked as const. Therefore we assume there is some compiler error here.

faster. This is because it uses less computations, stores results like the parity instead of computing them twice and returns when val1 is found to be greater than val2 while max_old finishes its loop no matter what. Another change is that we store val1[i] and val2[i] into local variables. This is done to make storing on registers for the full duration of the loop more likely.

The function update is a bit more sophisticated because it can maximize or minimize depending on the player given. First it computes the variable cs, the comments in the code contain a table of the values cs can take. This computation allows us to distinguish the cases where we need to do nothing (cs > 1), we simply need to copy from val2 to val1 (cs $== 1$) or both values are finite and we need to compare them.

Comparison is done similar to max_new now also taking into account player and opponent.

5.3.3 Preprocessing

The integral part of the preprocessing necessary is guaranteeing that no winning self-cycles for player 1 exist. We present two possible approaches here: Removing the cycles or breaking them by adding player 0 nodes.

Cycle removal

Given a parity game with graph G we want to remove all winning 1-cycles. To find such circles we use the following algorithm:

1) $G' = (V_1, E \cap (V_1 \times V_1))$
2) Launch Tarjan's algorithm on G',
   let c be the vector of SCCs for the nodes
3) For each v in G' with odd priority p
4) if there is a non−empty path from v to v
   using only priorities not greater than p
5) mark SCC c(v) for removal
6) Remove all marked SCCs and their 1−Attractor in G.

Step 2 is done using Tarjan's algorithm provided by the Boost Graph Library. Step 4 is a simple BFS/DFS that can be additionally limited to stay in the already computed SCC. Computing the attractor in step 6 works according to the definition of the attractor by computing *pre* repeatedly until a fixed-point is reached.

Cycle breaking

A much simpler approach to guarantee that there are no winning self-cycles for player 1 is to ensure that there are no cycles for player 1 at all. This is done by inserting a player 0 node before each player 1 node (see Figure 5.1).

Cycle breaking is of course a much faster algorithm but it comes at a cost: The resulting graph has more nodes then when applying the cycle removal algorithm. The benchmarking section delives a comparison between the two algorithms.

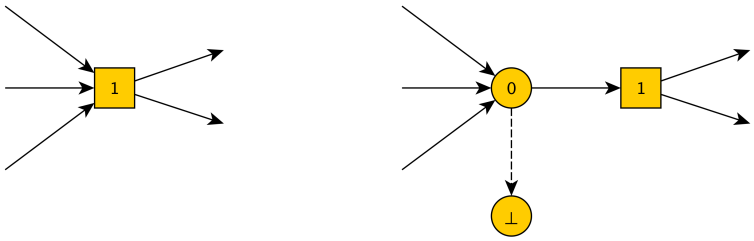

Figure 5.1: Cycle breaking, before(left) and after(right)

Sink node

The second important part of preprocessing is the introduction of a sink node. In our implementation, this node only exists virtually: As every player 0 node is connected to the sink node via an edge, the value of a player 0 node can never be worse than the value of the sink node plus its own priority. Thus instead of actually adding the sink node, we start maximizing player 0 nodes from the all zeros vector (the priority of the node is added after maximizing over all successors).

5.3.4 Node ordering

When arranging the graph as an array, the question arises whether node order matters and if so, how to order the nodes. The first thing to notice is that player 0 nodes are treated differently than player 1 nodes: The first are subject to maximizing, the second to minimizing the values. Thus one of our approaches was to order the nodes according to ownership. This approach turned out not to be fruitful. In the current version we combined min/max into one function that uses the player in its computation.

Another thing to look for is coalesced memory access. We made the assumption that most edges in a graph are between nodes that lie in the same strongly connected component (SCC). Therefore we tried ordering nodes according to their SCCs and found a speedup in many cases (see Section 5.4).

5.3.5 Other ideas

Below you can find some additional ideas we tried that did not quite work out as expected.

One observation was that valuations are always addressed in a "backward" manner, that is, when comparing two values the comparison proceeds from highest to lowest priorities. Therefore we tried to store valuations ordered from highest to lowest priority to improve caching, but it had nearly no effect

Choosing the amount of threads per block was also a question to which we cannot present a conclusive answer: According to multiple tutorials, 128 or 256 threads per block is optimal for most problems, but we observed nearly identical times on sizes 64, 512 and 1024 threads per block.

Another experiment included shared memory to store the valuations. The resulting kernel can still be found as f_sigma_shared. We tried to store two different things in

valuation: The intermediate array val that each thread uses as well as parts of the valuation. Our reason for storing val in shared memory is that locally declared arrays are often moved from the registers to local memory which we want to avoid. The valuations resides in global memory and is accessed multiple times, therefore we hoped to improve performance by using shared memory as some sort of cache. The main idea is to split the valuation into smaller parts that fit into shared memory, copy one part there, process all edges ending in that part, then proceed with the next part. The current implementation of this idea had nearly no effect on the running time at all.

Below is a code fragment that shows how we copy the valuation. SHARED_SIZE is a constant that holds how many bytes of shared memory we assign to each block.

```
// compute how many valuations will fit
const int space = SHARED_SIZE / priority_count - blockDim.x;

// the first blockDim.x many valuations are used for the val array
const int offset = blockDim.x * priority_count;

int currentEnd = 0;
int currentStart = 0;

// the edge currently processed
int currentEdge = edge_borders[node];

while (currentEnd < node_count) {
        // copy a bunch of valuations to shared
        currentStart = currentEnd;
        currentEnd += space;
        if (currentEnd > node_count) {
                currentEnd = node_count;
        }

        int pos = currentStart * priority_count + threadIdx.x;
        while (pos < currentEnd * priority_count) {
                shared_val[offset + pos - currentStart * priority_count] =
                        valuation[pos];
                pos += blockDim.x;
        }
        // here happens the f_sigma computation for those edges
        // that end inside the current valuation block
        ...
}
```

When choosing SHARED_SIZE one has to take into account that one multiprocessor has 48KB of shared memory, but this amount is split up between all blocks currently residing on this multiprocessor. That means if every block uses 48KB, only one block can reside on one multiprocessor at a given time. To use the computational power of the GPU, more than one block is needed per multiprocessor, that means less shared memory should be used.

	Nodes	Edges	CPU time	GPU time
cycle removal	67799	180146	44.87s	38.33s
cycle break	90450	202949	80.32s	55.4s

Table 5.1: Nodes resulting from different preprocessing algorithms

5.4 Benchmarks

In this section some of our design choices are benchmarked. It is divided into two parts: In the first part we compare different preprocessing implementations of ourselves against each other, in the second part a comparison to PGSolver takes place. For PGSolver, the solver chosen is always the a strategy iteration, either due to Jurdzińsky and Vöge or due to Schewe. By CPU and GPU we denote our implementations, running on the corresponding hardware.

5.4.1 Preprocessing

As mentioned earlier, we implemented two different algorithms that guarantee that no winning 1-cycle exist: Cycle break and cycle removal. The effect of those algorithms on the running time shall be benchmarked here. There are also two improvements on how to order the nodes, namely SCC-ordering and BFS-ordering inside SCCs, which are both evaluated here. The game type used for the Benchmarks of this section was the Jurdziński game

In Figure 5.2 you can see the effect of SCC- and BFS-ordering when using cycle removal. SCC ordering has a effect not clearly positive or negative, but when adding the BFS-ordering inside the SCCs, there is a speedup of about 20%. As you can see in Figure 5.3, the CPU implementation is nearly independent of the vertex ordering because the CPU suffers a lot less from random access.

In Figure 5.4 again SCC- and BFS-ordering are compared, but this time cycle break is used instead of cycle removal. The combination of SCC-ordering and BFS-ordering also achieves a speedup of close to 20%. Note that cycle break generates (for this specific examples) a game with about 30% more vertices than cycle removal and thus leads to longer computations. In Table 5.1 you can find exact numbers for the Jurdziński game with parameters 150, 150 which has 67800 nodes. Times are taken using SCC- and BFS-order.

Figure 5.6 shows that our preparation has nearly no effect for elevator verification games. These games consist of one very big SCC and multiple 1-node SCCs for which ordering has no effect.

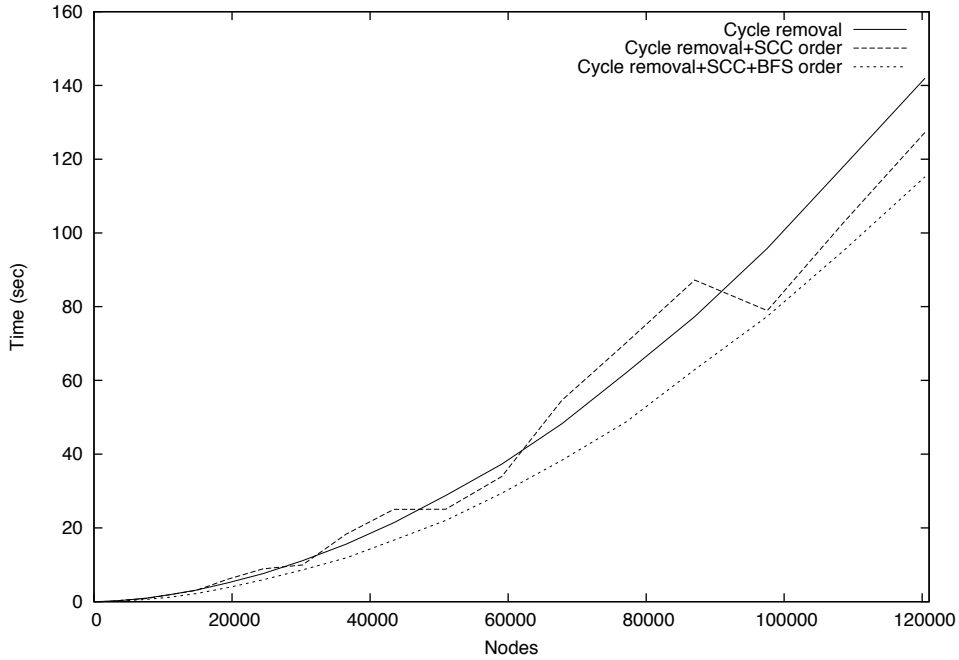

Figure 5.2: Jurdziński games on GPU using cycle removal

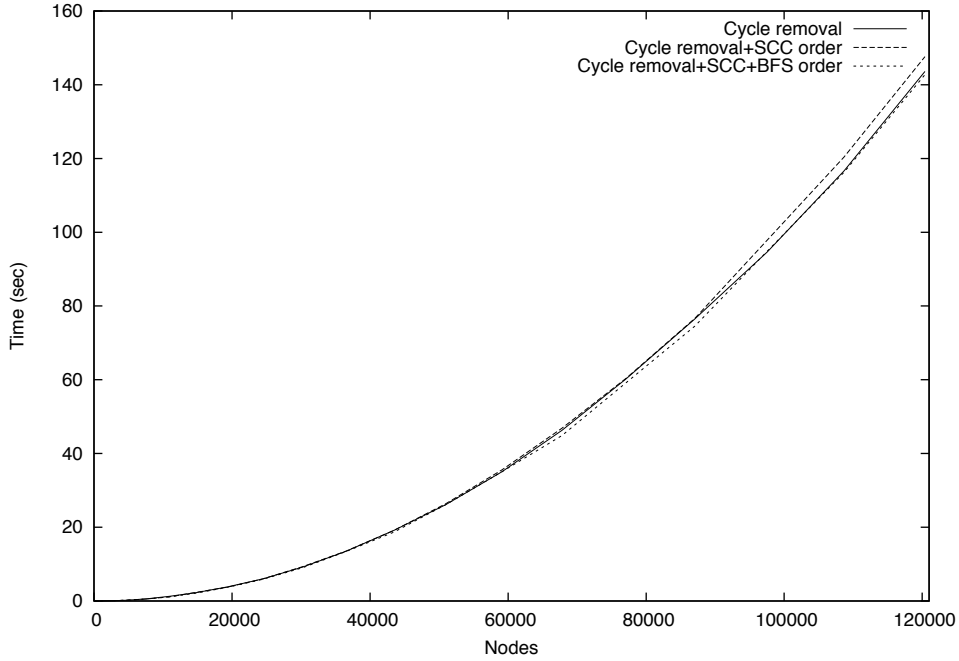

Figure 5.3: Jurdziński games on CPU using cycle removal

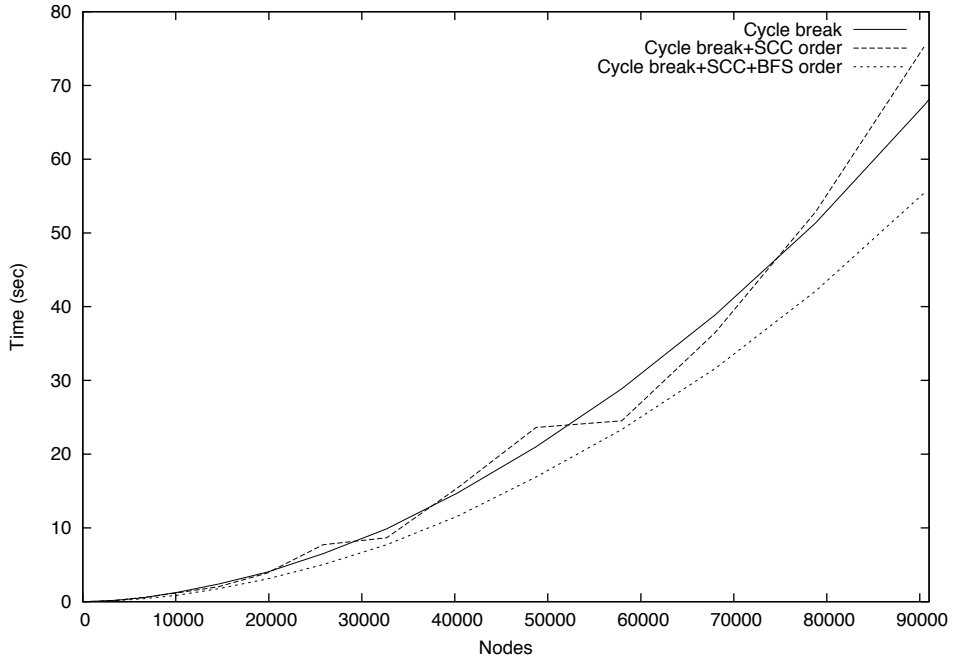

Figure 5.4: Jurdziński games on GPU using cycle break

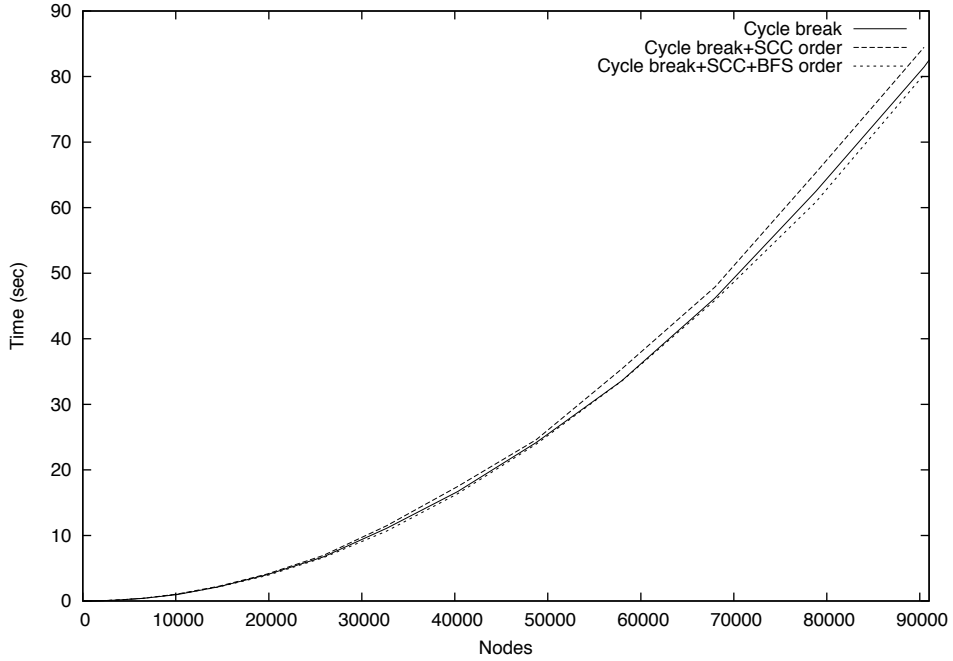

Figure 5.5: Jurdziński games on CPU using cycle break

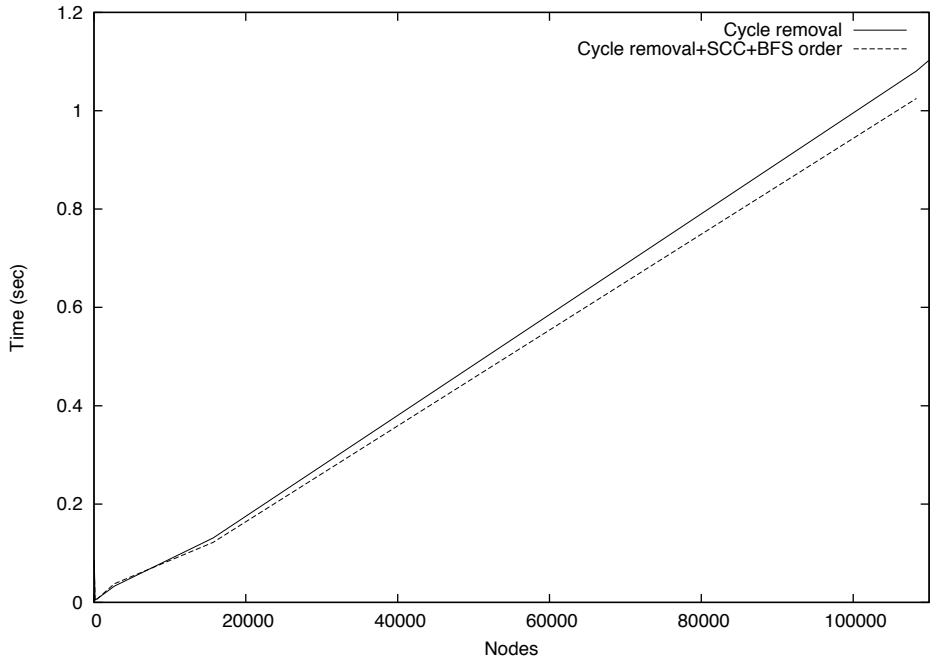

Figure 5.6: Elevator verification games on GPU using cycle removal

5.4.2 Performance

Now we turn to the performance of our implementation in comparison to PGSolver using the strategy iteration due to Jurdziński and Vöge and due to Schewe. Unless otherwise noted, all benchmarks use SCC and BFS-order and the cycle removal algorithm, PGSolver has its optimizations enabled.

Games used in this comparison are random games, clustered random games and elevator verification games. The results can be seen in Figures 5.7 to 5.13.

Two versions of random games and clustered random games have been used: A very sparse graph with out-degree of 2 to 5 per node, and a slightly less sparse one with out-degree 30 to 50. The simple structure of random games combined with 30 to 50 edges usually leads to cycle removal completely removing player 1 nodes as it is very likely that he has a winning cycle and that all his nodes form a single SCC. This is not the case for clustered random games.

For random and clustered random games, PGSolver uses the algorithm of Jurdziński and Vöge. As you can see in Figure 5.9, PGSolver times vary a lot in sparse clustered random graphs. This effect also occurs in random games, which is why there is no benchmark for PGSolver for the sparse version of random games: The times spiked between seconds and hours.

The performance improvement factor for random games compared to PGSolver varied between 10 and 20 on average. When disabling PGSolver's optimizations, it performed sometimes equal, sometimes considerably worse than it did before, depending on the games.

Comparing clustered random games with few edges against many edges, one can notice that games with fewer edges take longer to solve, which is counter-intuitive at first. Our theory to explain this is the following: More edges close more cycles, thus making the way to a winning cycle shorter. Therefore F_σ needs to be called less often to obtain a valuation. Although the individual call of F_σ takes more time, the total running time decreases.

Elevator verification games can be seen in Figures 5.11, 5.12 and 5.13. These examples shows the power of PGSolver's optimizations: When they are enabled, PGSolver can compete with our GPU implementation, without them its running time is extremely bad. Only Jurdziński and Vöge is depicted in Figure 5.11 because Schewe performed nearly identical. In Figures 5.12, 5.13 one can see that Jurdziński and Vöge is much more influenced when disabling optimizations. We believe that this is due to the fact that when no optimizations are applied, for Jurdziński and Vöge the graph has to be altered to contain every color at most once which is a huge impact because of the graph size. The benchmark graphs look very straight because of the lack of data points: Elevator verification games grow so quickly that we only tested 6 examples. The two biggest have about 16000 and 108000 nodes (the next one has about 862000 nodes).

As for random games, we see a speedup from CPU to GPU, but here the factor is much better: CPU needs about eight times as much time as GPU.

5.5 Possible Improvements

We now present some further improvements that could be implemented for a possible speedup.

Shared memory

As mentioned some of our experiments used shared memory but none of them could deliver a speedup over the current implementation. Yet we believe that used efficiently, there could be some benefits over global memory.

Texture memory

Since the graph is static, texture memory could be used to improve caching. It is not clear whether this yields any speedup as the cache structure of texture memory will not be used to its fullest when no interpolation is needed.

Node ordering

While the BFS-ordering has positive influence on the running time in some cases, there may be better orderings that optimize coalesced access. As seen on elevator verification games, there are cases in which this ordering shows only very little effect. Studying these cases could help finding new ideas on how to order nodes.

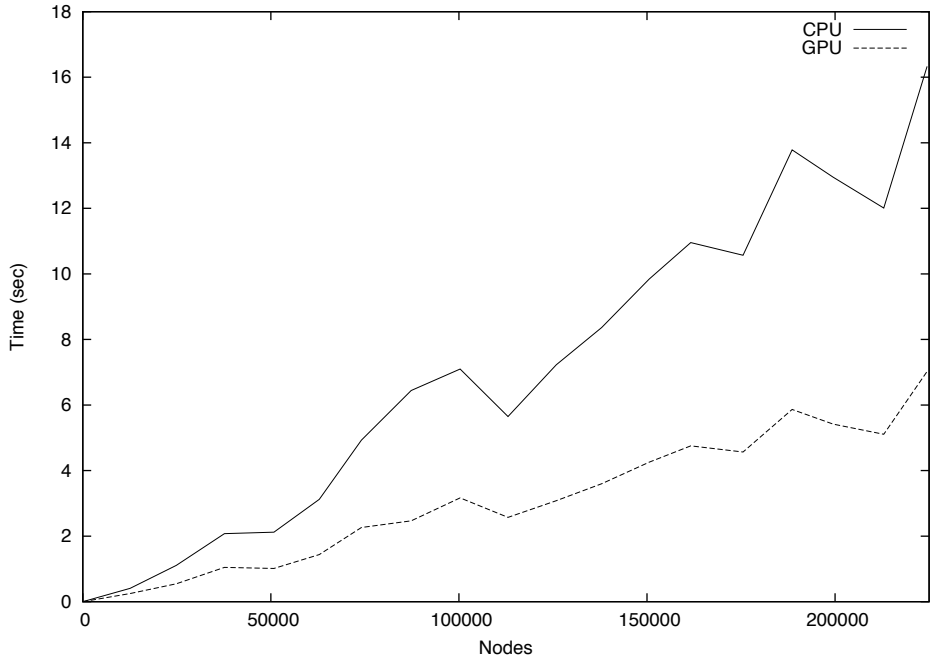

Figure 5.7: Benchmark for random games with out-degree 2-5

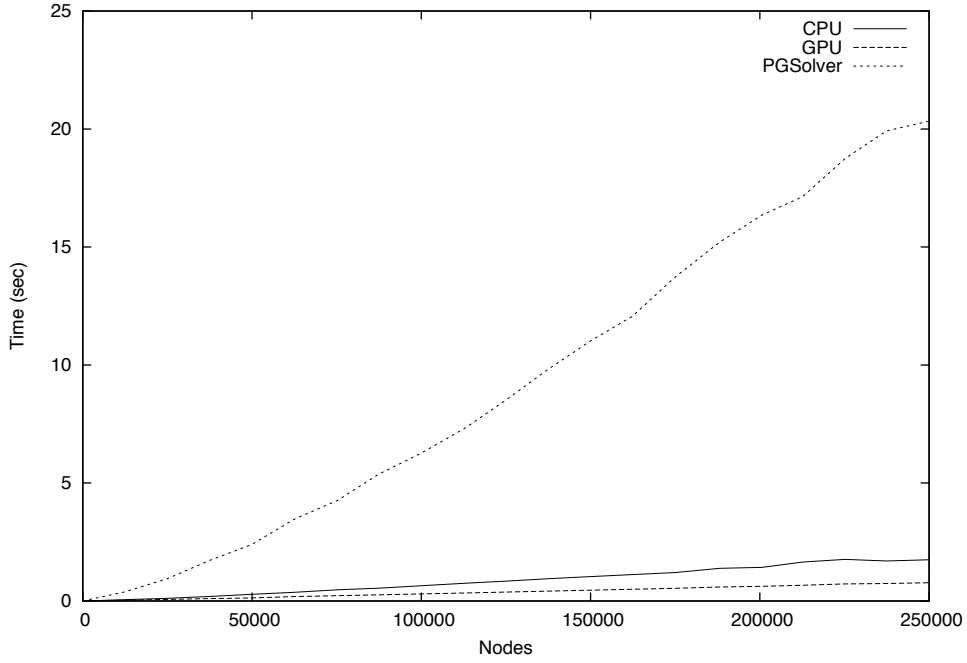

Figure 5.8: Benchmark for random games with out-degree 30-50

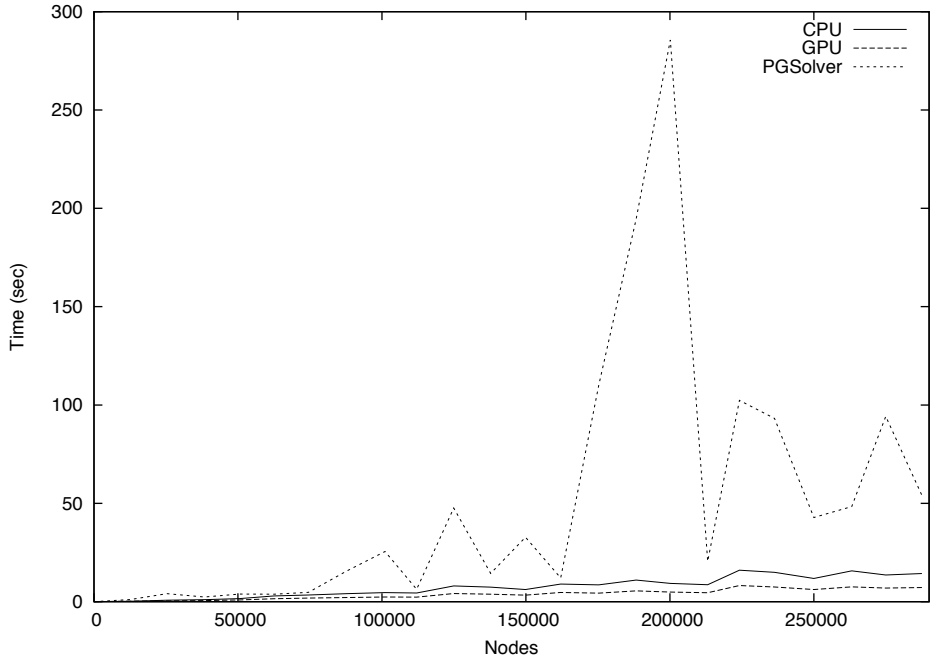

Figure 5.9: Benchmark for clustered random games with out-degree 2-5

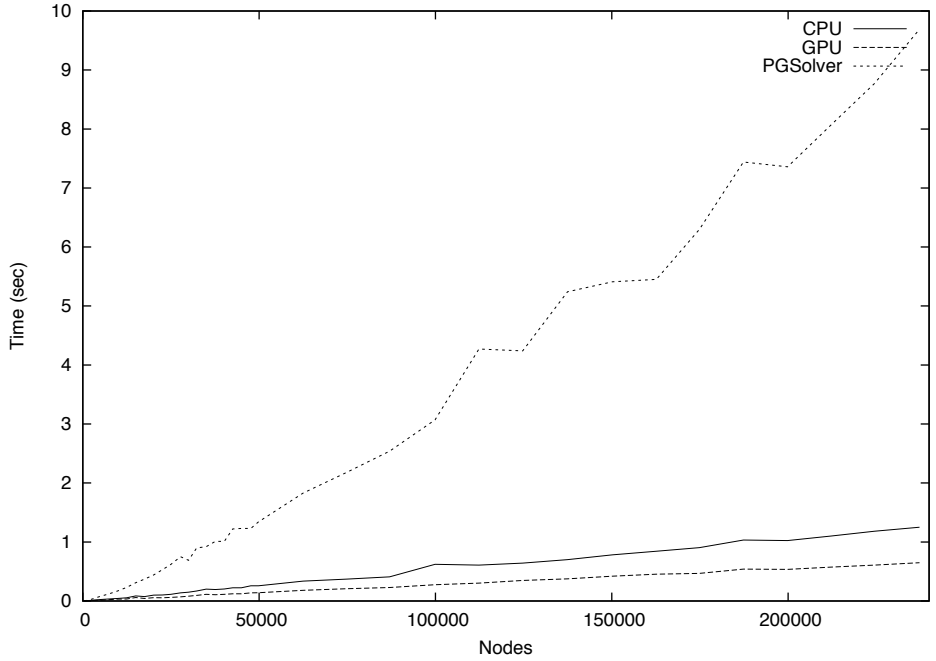

Figure 5.10: Benchmark for clustered random games with out-degree 30-50

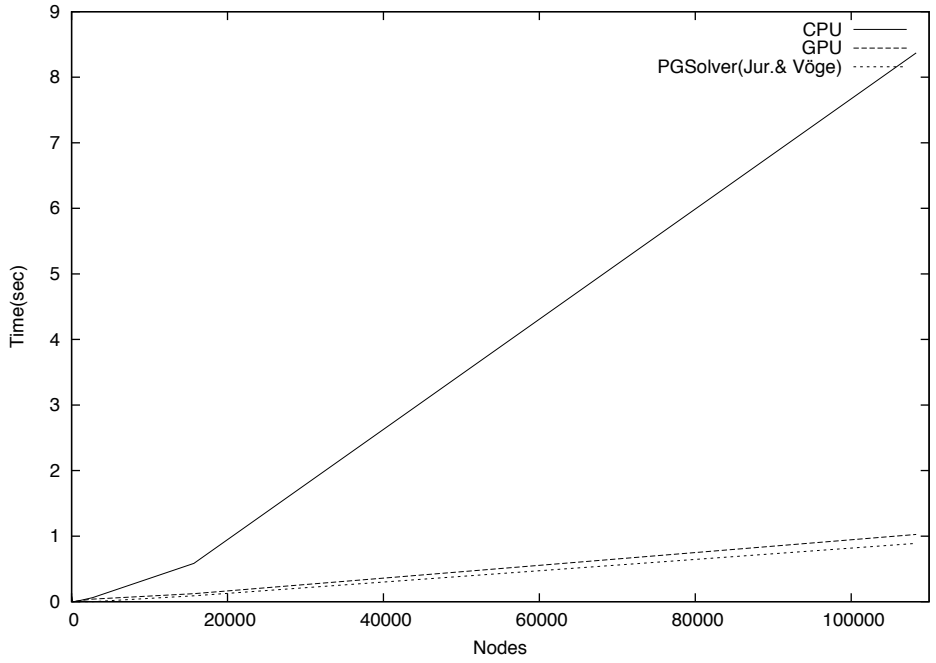

Figure 5.11: Benchmark for elevator verification games, PGSolver (Jurdzińsky and Vöge) uses optimizations

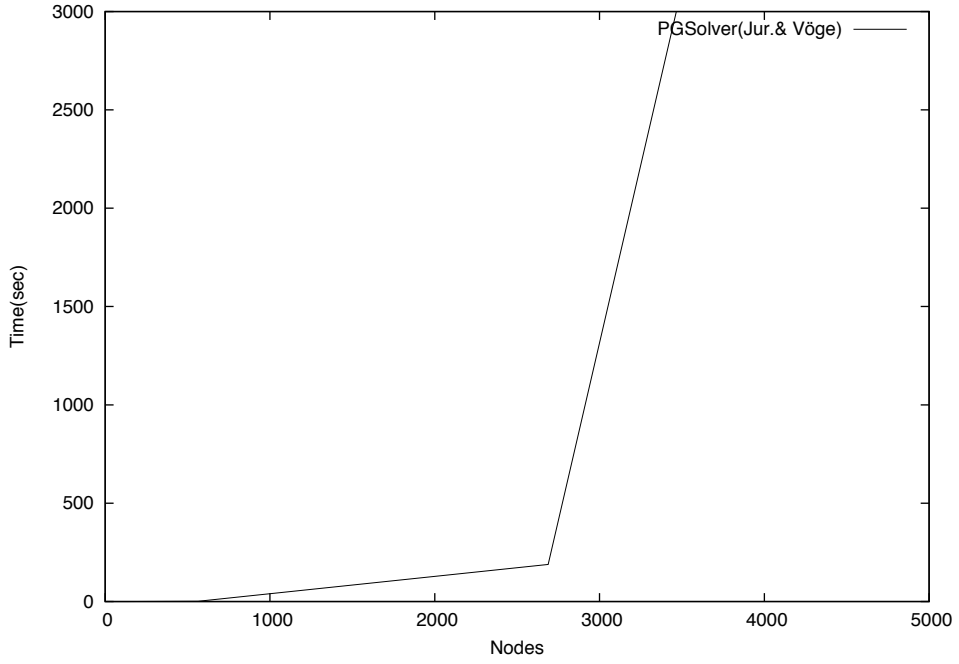

Figure 5.12: PGSolver (Jurdzińsky and Vöge) without optimization on elevator verification games

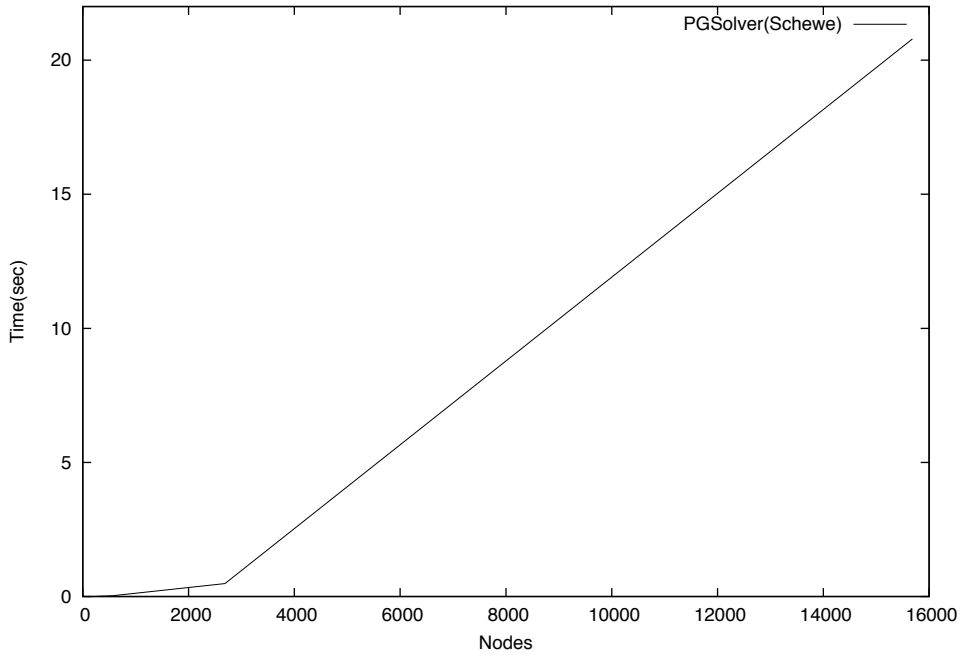

Figure 5.13: PGSolver (Schewe) without optimization on elevator verification games

Chapter 6

Results and future work

As we have seen, the computational power of GPUs can be used to speed up parity game solving in many cases, sometimes by a large margin. This leads us to the conclusion that strategy iteration, in particular the variant used in this work, is well-suited for the highly parallel nature of GPU computing. There are still many improvements to be done, some of which have already been noted in Chapter 5. Some more will be discussed below:

Multiple very advanced tools have been developed to simplify GPU programming and often offer better performance, most notably, the THRUST library [9]. This library could prove as a very valuable resource to speed up many operations, e.g. memory copies, and at the same time offers a high level interface which improves productivity and programming speed.

There are some cases in which there is only very little difference when comparing GPU and CPU implementation. These cases need to be studied in order to improve our implementation or the algorithm in general. Ideally the GPU implementation should be faster in all cases except for very small graphs where memory copy and device initialization are significant factors.

We only implemented a single heuristic for finding the improved strategy: We take all possible improvements. Other heuristics should be implemented and evaluated to see their performance in practice. Random choice or best valued switch could be such heuristics.

Our implementation computes valuations using a bellman ford adaptation. Schewe [14] mentions an adapted Dijkstra algorithm to accomplish the same in case of the all profitable switches heuristic. Depending on how good this algorithm is parallelizable we have high hopes that this would yield some speedup. [8] has already done research on the subject of large graph algorithms including single source shortest paths.

As solving parity games is very interesting in practice, another important topic is benchmarking this implementation on real-life examples. Studies about the typical structure of such games including amount and size of SCCs, priority count, average out-degree etc. could help a lot when optimizing the code.

Chapter 7

Appendix

Mathematical notion

This section contains mathematical notion that will be used without further explanation.
$\mathbb{N} = \{0, 1, 2, \dots\}$
For a set S, $\mathcal{P}(S)$ is its power set
The parity of a number $n \in \mathbb{N}$ is its residue when dividing by 2.

CUDA Example

Here is the complete and working CUDA example that was partially covered in section 4.

```
#include <stdlib.h>
#include <cuda_runtime.h>
#include <iostream>

// __global__ marks this as a kernel
__global__ void mult(int* vec, int mult) {
        // every thread processes the element according to his ID
        vec[threadIdx.x] *= mult;
}

void checkError() {
        cudaError_t error = cudaGetLastError();
        if(error != cudaSuccess)
        {
                // print the CUDA error message and exit
                printf("CUDA error: %s\n", cudaGetErrorString(error));
                exit(-1);
        }
}

int main(int argc, char** args){
        int size = 32;
        int* vec = (int*) malloc(size*sizeof(int));
        for(int i = 0; i < size; i++) {
```

```
            vec[i] = i;
        }

        // declare the device pointer
        int* vec_D;
        // allocate device global memory
        cudaMalloc(&vec_D, size*sizeof(int));
        // copy the array to the device
        cudaMemcpy(vec_D, vec, size*sizeof(int),
            cudaMemcpyHostToDevice);

        // call the kernel with 1 block of size threads
        mult<<<1, size>>>(vec_D, 3);

        // copy the result back to the host
        cudaMemcpy(vec, vec_D, size*sizeof(int),
            cudaMemcpyDeviceToHost);
        // check if there was an error
        checkError();

        for(int i = 0; i < size; i++) {
            std::cout << vec[i] << std::endl;
        }
        std::cin.get();
}
```

Listing 7.1: "CUDA example for scalar vector multiplication"

Code structure

Below the code structure of our implementation is listed.

cpu/
    cpu_kernel.cpp
cuda/
    cuda_stuff.cu
    dev_functions.cu
    kernel_gpu.cu
graph/
    boost_graph.cpp
    graph.cpp
stuff/
    filehandling.cpp
    stringfunctions.cpp
    tools.cpp
parity.cpp

cpu_kernel.cpp is the CPU equivalent of the kernel calls for the GPU. It exists for debugging and benchmarking purposes.

cuda_stuff.cu contains all CUDA preprocessings (e.g. allocating device memory) and the kernel calls. There are two versions:
cuda_stuff.cu is the basic version and can handle all games.
cuda_stuff_int4.cu can only handle games with at most 4 priorities.

def_function.cu contains min/max and compare functions for values.

kernel_gpu contains the kernels called by cuda_stuff. Again there are two versions, one for at most 4 priorities, the other one without that limitation.

boost_graph.cpp implements one of the two graph representations used. It is based on the Boost Graph Library [16] which is mainly used for their fast implementation of Tarjan's Algorithm as well as the ability to remove and add nodes easily. Some utility functions like parsing a parity game created by PGSolver are also included here.

graph.cpp implements the second graph representation which uses only basic C datastructures and thus can be passed to the CUDA kernels without further processing.

filehandling.cpp contains utility functions to save computed winning regions as well as compare them to PGSolver solutions.

stringfunctions.cpp and tools.cpp contain utility functions used for string manipulation and time measurement.

parity.cpp contains the main function and handles calls to PGSolver as well as to the CPU and GPU kernels.

Max and update

Below you will find our first version as well as an optimized version of the function max that computes the maximum of two values. It finally was combined with the function min to form the function update which is also listed below.

We begin with the first version:

```
//stores the max of val1 and val2 in val1
__device__ void max_old(int* val1, int* val2, int size) {

        if (val1[0] == INFINITY) {
                return;
        }
        if (val2[0] == INFINITY) {
                for (int i = 0; i < size; i++) {
                        val1[i] = val2[i];
                }
                return;
        }
        // will be true if any difference is found
        bool diff_found = false;
        // will be true if val2 is found to be greater than val1
        bool greater = false;

        for (int i = size - 1; i >= 0; i--) {
                greater = greater || (!diff_found && (i%2==0)
                        && ((val1[i]-val2[i])>>(sizeof(int)*8-1))!=0);
                greater = greater || (!diff_found && (i%2!=0)
```

```
                && ((val2[i]-val1[i])>>(sizeof(int)*8-1))!=0);
            diff_found = diff_found || !(val1[i] == val2[i]);
            val1[i] += (diff_found)*(greater)*(val2[i]-val1[i]);
        }
}
```

Listing 7.2: "First version of max"

This is how the function looks after some optimizations:

```
//stores the max of val1 and val2 in val1
__device__ void max_new(int* val1, int* val2, const int size) {

        if (val1[0] == INFINITY) {
            return;
        }
        if (val2[0] == INFINITY) {
            for (int i = 0; i < size; i++) {
                val1[i] = val2[i];
            }
            return;
        }

        // will be true if val2 is greater
        bool greater = false;
        bool always_true = true;

        for (int i = 0; i < size; i++) {
            // 1 if priority is even, -1 if odd
            int p = 1-2*((size-1-i)&1);
            int v1 = val1[i];
            int v2 = val2[i];
            int d = v2 - v1;

            if (!greater && d * p < 0) {
                return;
            }

            greater = greater || (always_true && ((d * p) > 0));
            v1 += greater * d;
            val1[i] = v1;
        }
}
```

Listing 7.3: "Optimized version of max"

Here is the combined min/max function:

```
__device__ void update( int* val1, int* val2, const int size,
            const int player, const int opponent) {

        int v1 = val1[0];
```

```
int v2 = val2[0];

// player | v1 == INF | v2 == INF | case
//   0    |     0     |     0     |  0
//   0    |     0     |     1     |  1
//   0    |     1     |     0     |  2
//   0    |     1     |     1     |  3
//   1    |     0     |     0     |  0
//   1    |     0     |     1     |  2
//   1    |     1     |     0     |  1
//   1    |     1     |     1     |  3
int cs = ((v1 == INFINITY) << opponent)
       | ((v2 == INFINITY) << player);

if( cs > 1 ){
        return;
}

if( cs == 1 ) {
        if( player == 0 ) {
                val1[0] = INFINITY;
        } else {
                for(int i = 0; i < size; i++)
                {
                        val1[i] = val2[i];
                }
        }
        return;
}
bool always_true = true;
bool replace = false;

for( int i = 0; i < size; i++) {
v1 = val1[i];
v2 = val2[i];
int d = v2 - v1;
// 1 if priority is even, -1 if odd
int p = 1-2*((size-1-i)&1);

// player | parity |  d  | return? |   formula
//    0   |  even  | <0  |   yes   |   1 *  1 * d < 0
//    0   |  even  | >0  |   no    |   1 *  1 * d > 0
//    0   |  odd   | <0  |   no    |   1 * -1 * d > 0
//    0   |  odd   | >0  |   yes   |   1 * -1 * d < 0
//    1   |  even  | <0  |   no    |  -1 *  1 * d > 0
//    1   |  even  | >0  |   yes   |  -1 *  1 * d < 0
//    1   |  odd   | <0  |   yes   |  -1 * -1 * d < 0
//    1   |  odd   | >0  |   no    |  -1 * -1 * d > 0
if (!replace && (opponent - player) * p * d < 0) {
```

```
                return;
            }
            replace = replace || (always_true
                    && (((opponent - player) * p * d) > 0));
            val1[i] = v2;
        }
    }
```

Listing 7.4: "Update function"

Bibliography

[1] H. Björklund, S. Sandberg, and S. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. *Mathematical Foundations of Computer Science 2004*, pages 673–685, 2004.

[2] Henrik Björklund, Sven Sandberg, and Sergei G. Vorobyov. Complexity of model checking by iterative improvement: The pseudo-boolean framework. In Manfred Broy and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 381–394. Springer, 2003.

[3] K. Chatterjee and N. Fijalkow. A reduction from parity games to simple stochastic games. *Arxiv preprint arXiv:1106.1232*, 2011.

[4] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, November 2011.

[5] O. Friedmann. An exponential lower bound for the parity game strategy improvement algorithm as we know it. In *Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on*, pages 145–156. IEEE, 2009.

[6] O. Friedmann and M. Lange. *The PGSolver collection of parity game solvers*, April 2010.

[7] E. Grädel, W. Thomas, and T. Wilke. *Automata, logics, and infinite games: a guide to current research*, volume 2500. Springer-Verlag New York Inc, 2002.

[8] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *HiPC*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2007.

[9] J. Hoberock and N. Bell. Thrust - parallel algorithms library, thrust.github.com.

[10] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000*, pages 290–301. Springer, 2000.

[11] Marcin Jurdziński. Deciding the winner in parity games is in UP [intersection] co-UP. *Inf. Process. Lett.*, 68(3):119–124, 1998.

[12] M. Luttenberger. Strategy iteration using non-deterministic strategies for solving parity games. *Arxiv preprint arXiv:0806.2923*, 2008.

[13] D. Perrin and J.E. Pin. *Infinite words: automata, semigroups, logic and games*, volume 141. Academic Press, 2004.

[14] S. Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Computer Science Logic*, pages 369–384. Springer, 2008.

[15] Sven Schewe. An optimal strategy improvement algorithm for solving parity games. Technical Report 28, AVACS - Automatic verification and analysis of complex systems, July 2007.

[16] Jeremy Siek, Lee Lie-Quen, and Andrew Lumsdaine. Boost graph library, www.boost.org/libs/graph/.

[17] D.A. Spielman and S.H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

[18] Severin Strobl. Efficient implementation of finite element operators on GPUs. Studienarbeit, 2008.

[19] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *Computer Aided Verification*, pages 202–215. Springer, 2000.

[20] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.