Eindhoven University of Technology

Master's thesis

Speeding up the small progress measures algorithm for parity games using the GPU

Author:
P.J.A. Bootsma

Supervisor:
dr.ir. T.A.C. Willemse

November 18, 2013

Abstract

Solving parity games is interesting because it is equivalent to model checking for μ-calculus. The small progress measures (SPM) algorithm by Jurdzinski is originally a sequential algorithm for solving parity games. The nature of this algorithm allows easy parallelization, and previous research has already adapted it to work on multi-core machines. Here, SPM is adapted to work on the many-core architecture used by modern graphics processing units (GPUs). Additionally, some existing techniques to speed up SPM or parity game solving in general are discussed, such as alternating SPM and simple graph preprocessing. A new technique is introduced that can help to reduces parity game solving times by assigning priorities to edges instead of vertices, after which shortcut edges can be added to the graph. Experimental validation shows that using the GPU for small progress measures can result in faster solving for some games, but the best solution is to let the CPU and GPU calculate in parallel and stop when either is done. Adding shortcuts when priorities are assigned to edges can also improve solving times for some games.

Contents

1 Introduction

Software is everywhere nowadays. For example, modern cars have software consisting of millions of lines of code, and modern medical devices are operated almost entirely via software. It is crucial that cars and medical devices operate safely to avoid accidents, but bugs in their software can impede their safe operation. The influence of bugs can be disastrous in a large number of software systems, ranging from financial transaction processing applications to industrial control systems. It is therefore important to be able to detect errors in these systems, so they can be fixed before any damage is done.

The difficulty in finding bugs in large systems is that their complexity makes it very hard to reason about their behavior. For example, two seemingly correct subsystems can interact in unforeseen ways, with possible results ranging from incorrect behavior to a complete deadlock of the system. Since the 1980s, research has focused on how to validate the correctness of complex systems. One of the methods developed for this is model checking, where all possible behaviors of the system are captured in a model, on which various properties can be checked. Validating whether these properties hold for the model is called the *model checking problem*. This problem can be translated to the problem of determining the winner of a *parity game*.

Algorithms for solving the model checking problem have almost exclusively focused on implementations for classical CPUs. Recently, graphical processing units, GPUs, are being used more and more for general-purpose computations. Due to the large difference in architecture in CPUs and GPUs, it is possible to solve some problems considerably faster on a GPU, especially if those problems involve applying simple operations often to a large amount of data. The *small progress measures* algorithm is an algorithm for solving parity games, and therefore also the model checking problem, which fits that description, so adapting this algorithm to be suitable for GPUs could provide a significant speed boost to solving the model checking problem.

Section 1.1 gives an introduction to the model checking problem, and describes how parity games can be used to solve this problem. Section 1.2 provides a description of parity games and their rules. Section 1.3 introduces the basic concepts of general-purpose GPU programming using the CUDA framework developed by NVIDIA.

1.1 The model checking problem

The model checking problem is the problem of, given a model of a system, validating whether a certain property holds on that model.

A model of a system is usually created as a *labeled transition system* or some variation on this formalism, such as a (mixed) Kripke structure. Such models can be used to describe the behavior of the, usually simple, subsystems that make up an entire system. These small models can be combined to form larger models which describe the complex behavior of the entire system when the subsystems work together.

The properties to validate on the model are described using some form of logic, such as *linear temporal logic* (LTL) [14], *computational tree logic* (CTL) [3] or modal μ-calculus [11]. Of these forms of logic, μ-calculus is the most expressive one and formulas written in most other forms of logic can be translated to formulas in μ-calculus.

A large number of different approaches to validating whether a property holds on a model exist. Some approaches work directly on the specified model and formula, while other approaches first translate the model and formula to a different formalism. One of the formalisms often used is that of *boolean equation systems* (BESs) [12]. A BES is a system of boolean fixpoint equations, where every equation assigns the result of a boolean formula to a boolean variable. Solving a BES consists of finding the values of every boolean variable.

A number of different techniques exist to solve a BES. Some techniques, such as Gauß elimination, work directly on the BES, while another way is to convert the BES to a *parity game*, which is a graph-based representation. This conversion is a simple linear-time process. Every vertex in the parity game corresponds to one of the boolean variables in the BES, and by solving the parity game the values of these variables can be obtained.

1.2 Parity games

A parity game is a game involving infinite paths played by two players on a parity graph [18]. A parity graph is a directed, total graph with an integer priority assigned to each vertex. The two players of a parity game are called $\Diamond$ (even) and $\Box$ (odd) and every vertex is owned by one of these two players. The game starts by putting a token on one of the vertices. Every turn, the player who owns the vertex which currently holds the token can decide where

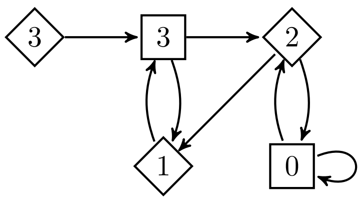


Figure 1.1: A parity game containing 5 vertices, of which 3 are owned by player ◇ and 2 are owned by player □. The priorities are noted inside of each vertex.

the token goes next, as long as the token moves only over edges. There is no end to this: every game is infinite. The path taken by a token is called a play and each play is infinite. The winner of a play is determined by the lowest priority which occurs infinitely often in the play: if this priority is even, player ◇ wins; otherwise, player □ wins. This game is called a min-parity game because we look at the lowest priority occurring infinitely often; there is also a max-parity variant where the highest priority occurring infinitely often wins. In this report, we only consider min-parity games. An example of a parity game is included in Figure 1.1.

Formally, a parity graph $G = (V, E, p)$ is a directed graph with a priority function $p : V \to \mathbb{N}$, which assigns a non-negative priority to each vertex in the graph. A parity game $\Gamma = (V, E, p, (V_\Diamond, V_\Box))$ consists of a parity graph and a division of the vertices in a set owned by ◇ ($V_\Diamond$) and a set owned by □ ($V_\Box$). These sets are disjoint ($V_\Diamond \cap V_\Box = \emptyset$) and together contain all vertices ($V_\Diamond \cup V_\Box = V$). Whenever player ○ is mentioned it can be either player ◇ or player □; player $\overline{\circ}$ is then the opposite player.

The choices both players make are represented by a strategy $\sigma_\bigcirc : V_\bigcirc \to V$ for each player. This strategy is a function which returns the next vertex in the play for every vertex the player owns. Moves are valid if and only if they are along an edge in the graph, so if and only if $(v, \sigma_\bigcirc(v)) \in E$ for all $v \in V_\bigcirc$. A play $\pi = \langle v_1, v_2, v_3, \dots \rangle$ is consistent with a strategy $\sigma_\bigcirc$ if and only if for all vertices v_l in the play owned by player ○, the next vertex is the outcome of the strategy function for v_l, so if and only if $v_{l+1} = \sigma_\bigcirc(v_l)$ for all $v_l \in V_\bigcirc$.

If we have a set of vertices $W \subseteq V$ and a strategy $\sigma_\bigcirc$ for player ○, then $\sigma_\bigcirc$ is a winning strategy for ○ if every play started from some vertex in W and consistent with $\sigma_\bigcirc$, is won by ○ regardless of the strategy of $\overline{\circ}$. So ○ wins all vertices in W if he has a winning strategy from W. Now let $W_\bigcirc \subseteq V$ be the largest set that ○ can always win and likewise for $W_{\overline{\bigcirc}}$. These sets always exist, are disjoint ($W_\bigcirc \cap W_{\overline{\bigcirc}} = \emptyset$) and together cover all vertices ($W_\bigcirc \cup W_{\overline{\bigcirc}} = V$) [4]. This means that the vertices can be divided uniquely into two sets: one containing all vertices won by ◇ and one containing all vertices won by □. The problem of solving a parity game is to find these sets: given a parity game $\Gamma = (V, E, p, (V_\Diamond, V_\Box))$, find the winning sets $W_\Diamond$

and $W_\Box$ for player $\Diamond$ and $\Box$ respectively.

Before we discuss how to solve this problem using the small progress measures algorithm in Chapter 2, we first present some notation used for reducing a parity game. A parity game can be reduced by removing some vertices from the game. If we have a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$ and a set of vertices $W \subseteq V$ we can remove all vertices in W from G to obtain the parity game $G \setminus W$. This game is defined as $G \setminus W = (V', E', p, (V'_\Diamond, V'_\Box))$ with $V' = V \setminus W$, $E' = \{(u, v) \in E \ : \ (u \notin W) \wedge (v \notin W)\}$, $V'_\Diamond = V_\Diamond \setminus W$ and $V'_\Box = V_\Box \setminus W$. Instead of removing all vertices in W, the game can also be restricted to all vertices in W giving the game $G \cap W$. This is defined similar to $G \setminus W$. Note that these reductions can affect the totality of the resulting graph.

1.3 Using CUDA for GPGPU programming

A modern GPU[1] consists of hundreds or even thousands of cores which each can process separate data. This facilitates the processing and rendering of (3D) graphics, which requires applying a small amount of code to a large number of items in the rendering pipeline. Other computations can also benefit from this computation model. This led to the introduction of GPGPU[2] programming.

One of the leading platforms in GPGPU programming is CUDA[3], developed by GPU-manufacturer NVIDIA. CUDA introduces a programming model tailored for running computations on the highly-parallel GPU architecture. Additionally, CUDA supplies a platform based on C++ to efficiently work with this programming model.

In the CUDA programming model, the CPU is called the *host* and the GPU is called the *device*. A device in CUDA is more general than just a GPU: it can be any co-processor which can execute CUDA programs and which has its own memory. Every CUDA application consists of code running on the host, which can copy data to and from the device memory and can start calculations on the device.

This section outlines the most important aspects of CUDA. For a more detailed description of the possibilities of the CUDA programming model and platform, and for exact device limitations, refer to the CUDA C Programming Guide[13].

[1]Graphics Processing Unit
[2]General-Purpose computing on Graphics Processing Units
[3]Compute Unified Device Architecture

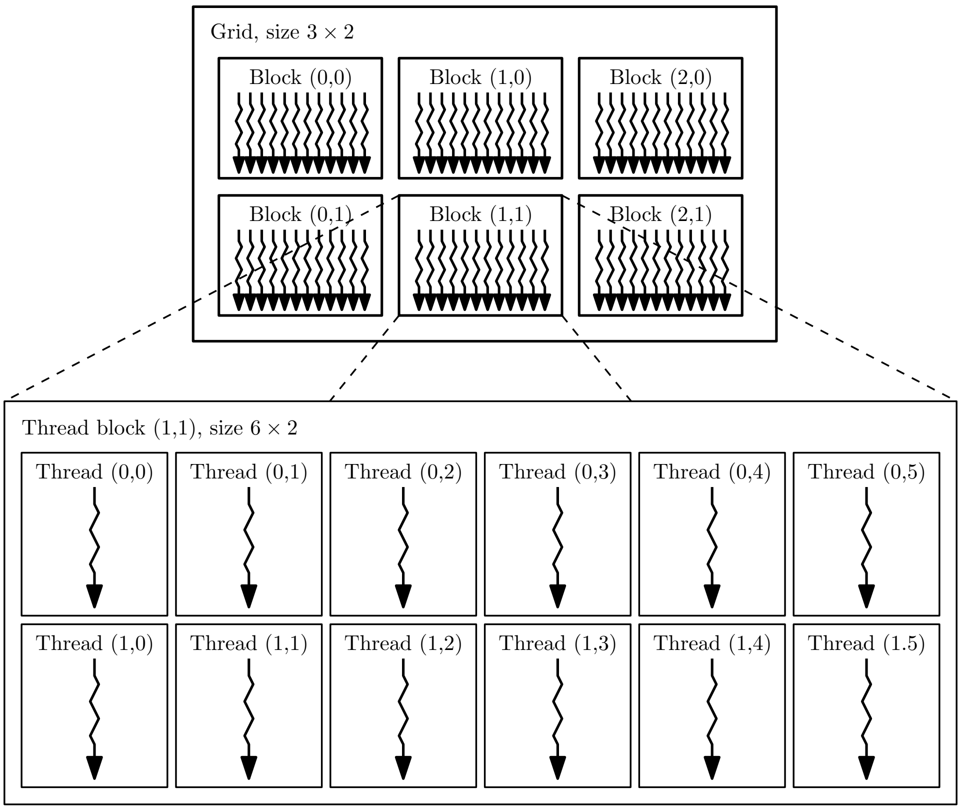


Figure 1.2: A 3×2 *grid of thread blocks, with each thread block having* 6×2 *threads.*

1.3.1 Managing threads

An important aspect of programming with CUDA is the ability to manage a large number of threads. A single CUDA computation might use millions or possibly even billions of threads. In general, a single thread performs a computation for a single data element, so there should be a thread for every data element.

The basic building block of a CUDA computation is a single thread. These threads are organized in thread blocks, which are a 1-, 2- or 3-dimensional block of threads. These thread blocks are then combined into grids, which can again be 1-, 2 or 3-dimensional. All thread blocks within a grid have the same dimensions. See Figure 1.2 for an example of a grid with thread blocks. Every thread knows the coordinates of itself inside of its thread block and the coordinates of its thread block inside the entire grid. This can be used to compute a unique index, e.g. for use when addressing data.

Since a single CUDA computation can use millions of threads, and the hardware resources on a GPU are limited, not all threads are active on the

GPU at the same time. The hardware of a GPU is divided into independent multiprocessors, and each multiprocessor has a limit on the number of simultaneously active threads. The exact limit of active threads depends on the hardware. For example, a current high-end GPU is the GeForce GTX 680, which can have at most 2048 simultaneously active threads per multiprocessor. The number of physical cores in a multiprocessor is lower; for the GTX 680, there are 192 cores in each multiprocessor and the entire GPU contains 8 multiprocessors for a total of 1536 cores.

To launch a computation on the GPU, a CPU-program has to specify the dimensions of the grid and thread blocks, as well as a *kernel* used for the computation. A kernel is a (short) piece of code which is run by every thread. Since every thread knows its thread index, a unique index can be calculated for each thread which is usually used to identify a single data-element for the thread to work on. For example, if a single calculation must be done for every vertex in some graph, a kernel is written which calculates a vertex index depending on the thread index, and then performs the required calculation on the vertex with the calculated index. This kernel is then launched with a total number of threads equal to the total number of vertices.

All threads running the same kernel must execute the same code on a different piece of data. The CUDA scheduler operates on *warps* of 32 threads each. The threads in a warp are run in lock-step: every thread executes the same instruction on different data. This works even when conditional statements and loops require different code paths: all relevant code paths are executed for all threads, but threads only actually store their results if the instruction is part of the thread's code path. If multiple threads in the same warp require different code paths it is called *divergence*. High divergence is bad for the performance of the algorithm, since calculations and/or memory transfers are performed for threads which ignore the results.

1.3.2 Memory hierarchy

CUDA exposes a memory hierarchy consisting of local, shared and global memory. Each of these memories has different characteristics and different ways to get the best performance out of them. Since CUDA algorithms often require a lot of data, it is important to design the algorithms to use the available memory optimally.

Local memory: Every thread has its own per-thread local memory. This is generally used for temporary variables such as loop counters, storing intermediate values, etc. If possible, all this memory is stored in on-chip registers which are very fast to read from and write to. If the required amount of memory is too big, part of the local memory can also be stored in global memory, which is substantially slower. It is therefore useful to keep the amount of local memory used low enough to fit within the available registers. The number of available registers depends on the number of thread blocks running on a multiprocessor simultaneously, and on the exact hardware used.

Shared memory: Every multiprocessor has shared memory, which can be used by all threads within the same thread block. It is also fast to access and can therefore not only be used to share data within a thread block, but also as a self-controlled cache. There is not much shared memory, usually 48 KiB per multiprocessor, and it has to be shared by all thread blocks currently active on the multiprocessor. The shared memory is divided into 32 banks and data can be read from or written to all banks simultaneously.

Global memory: The entire GPU has a large amount of global memory – modern high-end GPUs usually have 2 GiB of global memory. Reading from and writing to this memory is slow, but if all threads in a warp request a consecutive piece of memory, then these request can be coalesced into fewer, longer requests. This increases memory throughput. Basically, memory is read and written in 128-byte chunks, and if all memory requests from a warp fall into the same 128-byte chunk, only a single memory transfer is needed.

2 Small progress measures

To describe the small progress measures algorithm by Jurdziński [8], we will only use the case where we want to solve the game for player $\diamond$. This means that we will look at the game from the viewpoint of player $\diamond$ and that we can calculate a winning set $W_\diamond$ and strategy $\sigma_\diamond$ for player $\diamond$. Because the winning sets are disjoint and together contain all vertices, this also gives us the winning set $W_\square$ for player $\square$, but we will not have a strategy $\sigma_\square$. Solving the game for player $\square$ is similar, but can also be implemented by a preprocessing step which converts the game to one to be solved for player $\diamond$: all priorities have to be increased by 1 (to swap evenness of the priorities) and the owners of all vertices have to be swapped.

The following description of the small progress measures algorithm is based on both the original work by Jurdziński [8] and a different description of the same algorithm by Klauck [10].

The basic idea behind small progress measures is to characterize the cycles reachable from each vertex. A vertex can be won by player $\diamond$ if it can steer the game to a cycle where the lowest priority of that cycle is even, regardless of the strategy of player $\square$. To characterize the cycles reachable from a vertex, a measure is assigned to each vertex which counts the maximal number of times each odd priority can be seen until a vertex with a smaller priority is seen. If this measure is sufficiently large then player $\square$ can steer the game to a cycle where the lowest priority is odd.

A cycle in a parity graph is a path $\langle v_1, v_2, \ldots, v_n \rangle$ such that $v_1 = v_n$, and $v_1, v_2, \ldots, v_{n-1}$ are all different. Each cycle has an index i which is equal to the lowest priority occurring in the cycle. A cycle with index i is referred to as an i-cycle. A cycle is an even cycle if it is an i-cycle for even i.

To describe the cycles in a parity game, a progress measure will be attached to each vertex in the parity game. The progress measure used here is a d-tuple $m \in \mathbb{N}^d$, a tuple with d elements, where d is the highest priority in the game plus 1. For example, if the highest priority in a parity game is 4, then $d = 5$. Each tuple will then have 5 elements: one for each priority occurring in the game. The first element of such a tuple has index 0 and characterizes reachable 0-cycles; the second element has index 1 and characterizes reachable 1-cycles, and so on. The index of the last element is equal to the highest priority in the game.

Comparing d-tuples is done using lexicographical ordering. Additionally, it is possible to compare only the first few elements of the tuple by using operators such as $>_i$ and $\geq_i$. The expression $a >_i b$ means that tuple a must be strictly larger than tuple b when considering only the elements up to index i. This can be generalized to other comparison operators. For example, $(0,2,0,1) >_1 (0,1,0,2)$, $(0,1,0,0) \geq_2 (0,1,0,2)$ and $(0,2,0,1) =_0 (0,0,0,0)$, but $(0,1,0) \not\geq_1 (0,2,0)$.

2.1 Progress measures in solitaire games

A solitaire game is a parity game where one of the players cannot make any choices. This can for example be a game where all vertices are owned by the same player, or a game which is obtained by removing all edges inconsistent with the strategy of one player. For the latter case, we use the notation $G_{\sigma_{\bigcirc}}$ to indicate the parity game G where all edges inconsistent with $\sigma_{\bigcirc}$ have been removed. In these solitaire games, vertices in $V_{\bigcirc}$ have an out-degree of exactly 1, since the graph is total and there is no choice for $\circ$. Before explaining the small progress measures algorithm on game arenas, where two players can make choices, this section will explain the algorithm for solitaire games since that is conceptually simpler.

Definition 2.1. A strategy $\sigma_{\bigcirc}$ is closed on $W \subseteq V$ if every play starting in W and consistent with $\sigma_{\bigcirc}$ stays within W. Formally, $\sigma_{\bigcirc}$ is closed on W if

- $(\forall\ v \in W \cap V_{\bigcirc}\ :\ \sigma_{\bigcirc}(v) \in W)$, and
- $\Big(\forall\ (v,w) \in E\ :\ v \in W \cap V_{\overline{\bigcirc}} \implies w \in W\Big)$.

Let G be a parity game, let $W \subseteq V$ be a subset of vertices in the game, and let $\sigma_{\Diamond}$ be a strategy for $\Diamond$ closed on W. Now consider the solitaire game $G_{\sigma_{\Diamond}} \cap W$, which is the solitaire game (for $\Box$) obtained by removing all edges inconsistent with $\sigma_{\Diamond}$ from G and then reducing this game to only include the vertices in W. The strategy $\sigma_{\Diamond}$ is winning for player $\Diamond$ for all vertices $v \in W$ if and only if all cycles in $G_{\sigma_{\Diamond}} \cap W$ are even [10].

Given this observation, we can attempt to characterize vertices that can only reach even cycles. To do this, a parity progress measure is introduced, which is a function that assigns a measure to every vertex in the game. These measure are d-tuples as defined earlier, which are denoted by the type $\mathbb{M}_{\Diamond}$.

Definition 2.2. Given a solitaire game G, the function $\rho : V \to \mathbb{M}_{\Diamond}$ is a parity progress measure for G if for all edges $(v,w) \in E$ it holds that:

1. $\rho(v) \geq_{p(v)} \rho(w)$ if $p(v)$ is even
2. $\rho(v) >_{p(v)} \rho(w)$ if $p(v)$ is odd

There can only be a parity progress measure if all cycles are even. If there is an odd cycle, then the lowest priority in that cycle is odd; let this priority be p' and let it be the priority of vertex v_1. Then we have a cycle $\langle v_1, v_2, \ldots, v_n, v_1 \rangle$ and according to the definition of a parity progress measure we have $\rho(v_1) >_{p'} \rho(v_2) \geq_{p'} \cdots \geq_{p'} \rho(v_n) \geq_{p'} \rho(v_1)$, which means that $\rho(v_1) >_{p'} \rho(v_1)$, which is impossible, so no parity progress measure can exist if there is an odd cycle.

For the measures of type $\mathbb{M}_\diamond$, the d-tuples are restricted to a certain maximal value for each component of the tuple. The maximal value of a component with odd index i depends on the amount of vertices with priority i in the graph. Let V_i be the set of all vertices with priority i, then $|V_i|$ is the number of vertices with priority i. The maximal value for components in a restricted d-tuple with odd index i is $|V_i|$. For even index i, the value is always 0, regardless of $|V_i|$.

The rationale for these maximal values is that they describe how often a vertex with a certain priority can be seen before a vertex with a lower priority is seen. For even priorities this number does not matter: we want to look for even cycles. This also corresponds with the greater-equal comparison in the definition of parity progress measures as given above. For odd priorities the number does matter: if we see some vertex with odd priority more than once without seeing a vertex with lower priority, this means that we have reached a cycle with odd priority and a parity progress measure cannot exist. All valid values for odd priority i are thus at most $|V_i|$: for any value higher than this, some vertex with priority i is seen more than once and we have reached an i-cycle.

The above idea can be extended to not only check whether all cycles are even, but to check if some cycles are even when odd cycles may also exist. As seen above, when some odd cycle exists no parity progress measure can be calculated. The value $\top$ is introduced to indicate this case. This value represents infinity for restricted d-tuples. A value of $\top$ is strictly greater than any value $m \neq \top$. Comparisons between $\top$ and $\top$ are always true, most notably $\top > \top$ and $\top \geq \top$.

Using this extension to restricted d-tuples, an extended parity progress measure $\rho : V \to \mathbb{M}_\diamond^\top$ can be defined, where the vertices with label $\top$ constitute the winning region of player $\square$. Using this extended parity progress measure, there is no need to only look at a subset $W \subseteq V$ because we can find subsets of vertices where each player wins in the entire game. If any vertex has a value of $\top$, then player $\diamond$ has no choice at that vertex but to let the game advance to some vertex where player $\square$ can create an odd cycle. Similarly, if a vertex has a value different than $\top$, then the game will end in an even cycle regardless of any choice made by player $\square$. The winning regions $W_\Diamond$ and $W_\Box$ can thus be defined as follows:

$$\begin{array}{lcl} W_\Diamond & = & \{v \in V : \rho(v) \neq \top\} \\ W_\Box & = & \{v \in V : \rho(v) = \top\} \end{array}$$

Note that this only holds if we look at the *smallest* parity progress measure. An obvious solution to obtain a valid progress measure would be to set all values to $\top$, but this does not say anything meaningful about the game. To be able to talk about the smallest parity progress measure, an ordering on these functions is defined.

Definition 2.3. Let ρ and ϕ be two extended parity progress measures, then $\rho \sqsubseteq \phi$ if and only if it holds that $\rho(v) \leq \phi(v)$ for all $v \in V$. If it also holds that $\rho \neq \phi$, then $\rho \sqsubset \phi$.

The definition of the winning regions $W_\Diamond$ and $W_\Box$ is valid if ρ is the smallest parity progress measure under $\sqsubseteq$.

2.2 Progress measures in game arenas

The previous discussion only considered solitaire games where the strategy of player $\Diamond$ was fixed. This will now be extended to game arenas, where both players can make choices. For vertices owned by player $\Diamond$ there must be at least one neighbor which satisfies a certain progress relation; this would then be the move that player $\Diamond$ makes at this vertex. For vertices owned by player $\Box$ all possible moves must satisfy this progress relation, since any possible choice for player $\Box$ must be considered.

The progress relation used is $\text{prog}(\rho, v, w)$, where ρ is some progress measure and v and w are two neighboring vertices ($(v, w) \in E$). The value of $\text{prog}(\rho, v, w)$ is defined as follows:

Definition 2.4. $\text{prog}(\rho, v, w)$ is the least $m \in \mathbb{M}_\Diamond^\top$ such that

$$\begin{array}{ll} m \geq_{p(v)} \rho(w) & \text{if } p(v) \text{ is even} \\ m >_{p(v)} \rho(w) & \text{if } p(v) \text{ is odd} \end{array}$$

Note that the latter case can result in $\top$ if there is no $m \neq \top$ such that $m >_{p(v)} \rho(w)$.

Using this progress relation, a game parity progress measure is defined. If a vertex is owned by player $\Diamond$, then some neighbor must satisfy the progress relation and if a vertex is owned by player $\Box$, then all neighbors must satisfy the progress relation. This is because we solve the game for player $\Diamond$: we can decide the choice for player $\Diamond$, but must consider all possible choices by player $\Box$.

Definition 2.5. A function $\rho : V \to \mathbb{M}_{\Diamond}^{\top}$ is a game parity progress measure if for all $v \in V$ it holds that

$$\begin{array}{ll} \left(\exists\ (v,w) \in E\ :\ \rho(v) \geq_{p(v)} \mathrm{prog}(\rho, v, w)\right) & \text{if } v \in V_{\Diamond} \\ \left(\forall\ (v,w) \in E\ :\ \rho(v) \geq_{p(v)} \mathrm{prog}(\rho, v, w)\right) & \text{if } v \in V_{\Box} \end{array}$$

If ρ is a least game parity progress measure, where least is defined using the ordering $\sqsubseteq$ as defined in Definition 2.3, then we can say that $\rho(v) \neq \top$ if and only if all cycles reachable from vertex v are even. This means that using a least game parity progress measure, we can calculate the winning regions $W_{\Diamond}$ and $W_{\Box}$. They are defined identical to the winning regions in solitaire games, where $W_{\Diamond}$ contains all vertices from which all cycles reachable are even ($\rho(v) \neq \top$) and $W_{\Box}$ contains all other vertices.

What remains is that given a parity game, a least game parity progress measure must be calculated. The ordering $\sqsubseteq$ is monotonic and gives a complete lattice on the set of functions $V \to \mathbb{M}_{\Diamond}^{\top}$, so according to the Knaster-Tarski theorem [15] a least game parity progress measure exists and it is computable by fixed point iteration. To structure this fixed point iterations the function $\mathrm{lift}_v(\rho)$ is defined as follows for all $v \in V$:

$$\mathrm{lift}_v(\rho) = \begin{cases} \rho[v := \min\left((v,w) \in E\ :\ \mathrm{prog}(\rho, v, w)\right)] & \text{if } v \in V_{\Diamond} \\ \rho[v := \max\left((v,w) \in E\ :\ \mathrm{prog}(\rho, v, w)\right)] & \text{if } v \in V_{\Box} \end{cases}$$

This lift function is derived from the definition of game parity progress measures as given above. For vertices owned by player $\Diamond$, the resulting measure must be greater than or equal to the measure of at least one neighboring vertex, so we can pick the lowest measure of all neighbors. For vertices owned by player $\Box$, the resulting measure must be greater or equal to the measures of all neighboring vertices, so we must pick the highest measure of all neighbors.

Using this function the algorithm can now be described easily. The algorithm uses fixed point iteration to calculate the least game parity progress measure, where $\mathrm{lift}_v(\rho)$ is used to find the next value if the least fixed point is not yet found. The pseudocode for the algorithm is as follows:

```
Jurdzińksi$_{\Diamond}$($G$)
**Input:** a parity game $G = (V, E, p, (V_{\Diamond}, V_{\Box}))$
**Output:** the least game parity progress measure $\rho$
(1)    $\rho : V \to \mathbb{M}_{\Diamond}^{\top} \leftarrow \lambda v \in V.(0, \ldots, 0)$
(2)    **while** $\rho \sqsubset \mathrm{lift}_v(\rho)$ for some $v \in V$
(3)        $\rho \leftarrow \mathrm{lift}_v(\rho)$
(4)    **return** $\rho$
```

Note that this algorithm keeps finding some vertex v which needs to be lifted and then lifts it; there is no determined order in which the vertices are lifted. Choosing a good lifting strategy is one of the ways to get this algorithm to run fast. For the correctness of the algorithm, any lifting strategy will do; the lifting strategy only influences the practical speed of the algorithm.

The calculated game parity progress measure can not only be used to determine the winner of each vertex in a parity game, but can also be used to determine the strategy $\sigma_\diamond$ for player $\diamond$ for all $v \in V_\diamond$.

Definition 2.6. Given a least game parity progress measure ρ for player $\diamond$, the strategy for a vertex $v \in V_\diamond$ can be determined by picking the successor with the least measure assigned to it, i.e. $\sigma_\diamond(v) = v'$ such that $\rho(v') = \min\left(\rho(w) \ : \ (v, w) \in E\right)$ for all $v \in V_\diamond$.

3 GPU algorithm for Small Progress Measures

This chapter contains a description of the small progress measures algorithm that is adapted to be suitable for the GPU. Section 3.1 contains additional notations used in the algorithms. Section 3.2 describes the necessary global state. Section 3.3 contains the kernels for the basic small progress measures algorithm, and the corresponding host algorithm. Section 3.4 contains two non-trivial lifting strategies suitable for GPUs.

3.1 Notation for CUDA-algorithms

An introduction to the CUDA programming model was already given in Section 1.3. This section provides some generic implementation details, as well as pseudocode notations for CUDA constructs.

Both thread blocks and grids can be 1-, 2- or 3-dimensional, but here we only require 1-dimensional thread blocks and grids. Furthermore, since the algorithm does not depend on the number of threads in a thread block, we will only refer to the total number of threads started and leave the size of each thread block as an implementation detail.

The following statements will be used in the pseudocode of the algorithms to describe the special properties of CUDA algorithms:

- **launch** SOME_KERNEL(v) **for each** $v \in V$

  This is used to indicate that the kernel SOME_KERNEL(v) will be called with argument v, where every thread has a different v, such that $v \in V$ and all $v \in V$ are assigned to some kernel. The number of kernels launched is equal to $|V|$.

- Synchronize all threads in current thread block

  It is possible to synchronize all threads in one thread block. When this line is encountered, every thread in the thread block will wait until all threads reach that point, and only then will execution continue. This is useful when working with shared memory.

- **shared** *var*

  This indicates that a certain variable is placed in shared memory, meaning that every thread in a thread block has access to this memory. All further occurrences of this variable refer to the same shared variable. This can be used to synchronize various properties inside of a single thread block. Shared memory in one thread block is not accessible from any other thread block, and will be released when all threads in the thread block have finished execution.

Some more notation is used which is specific to the small progress measures algorithm. These are the following constructions:

- $x \leftarrow_p y$

  This can be used when assigning, to x, a progress measure stored in y. In some cases the progress measures must only be copied partially. The subscripted assignment indicates that the elements up to and including the element with index p are copied from y to x, and the remaining elements are filled with the value 0.

The kernels can be theoretically analyzed on both running time and global memory efficiency. The running time is analyzed in roughly the same way as for normal algorithms, with the exception that divergence is taken into account: when multiple different code paths can be executed for different threads in a warp, then the time for running all code paths is counted. For example, if half of the threads execute a code path with a running time of $\mathcal{O}(n)$, and the other half execute a code path with a running time of $\mathcal{O}(m)$, then the running time of both code paths combined is $\mathcal{O}(n + m)$, even if no single thread actually executes both code paths.

The global memory efficiency of a kernel is analyzed by looking at the number of values transfered from and to global memory, and determining how many of those values are actually used by the kernel. If a single coalesced memory transaction reads 32 consecutive values from memory, and every value is used by at least 1 thread, then 100% of the memory bandwidth for this transaction is used for useful data.

A non-coalesced memory access also reads 32 consecutive values from memory, but in the worst case only 1 of those values is used by any thread. The memory efficiency of all 32 memory transactions is only $1/32 = 3.125\%$. This is also the efficiency when a single global value is read from memory to be used by all threads, with the difference that it requires only 1 memory transaction to read the value for all threads: the block of 32 consecutive values to which the value belongs is read from memory, but only 1 value is actually used.

This memory efficiency analysis is based on the assumption that all values can be stored in 4 bytes, which is a reasonable assumption for the implementation of small progress measures.

3.2 Global state

Kernels cannot return any value, so all they can do is change the global state of the algorithm. This global state is stored in memory residing on the GPU. Data can be uploaded to or downloaded from the GPU by issuing a memory copy between host memory and device memory.

The global state used in this algorithm consists of the variables and arrays described below, which are always available to all threads. Arrays are described using a notation similar to functions. For example, $T : V \to \mathbb{B}$ denotes an array T which contains a boolean for every vertex in the parity game. All vertices and all edges have a sequentially numbered index which is used to lookup the correct element in such an array.

$G = (V, E, p, (V_\Diamond, V_\Box))$ — The parity game to solve, including its vertices V, edges E, priority function p and distribution of the vertices to the players using $V_\Diamond$ and $V_\Box$.

$\rho : V \to \mathbb{M}_\Diamond^\top$ — An array containing the current measure assigned to each vertex. This corresponds to the function ρ as introduced in Section 2.1.

$\rho_E : E \to \mathbb{M}_\Diamond^\top$ — An array containing the current measure assigned to each edge. This array is used to store the values of the progress relation $\mathrm{prog}(\rho, v, w)$ for every edge $(v, w) \in E$.

$M^\uparrow : \mathbb{M}_\Diamond^\top$ — The maximal value a measure can have before reaching $\top$: the only $m \in \mathbb{M}_\Diamond^\top$ such that $m > M^\uparrow$ is $\top$.

3.3 Kernels and host algorithm

The most important operation of the small progress measures algorithm is the lifting operation. The algorithm continuously lifts some vertex until no more vertices need to be lifted. GPU algorithms must exploit some form of parallelism, and this is done by lifting multiple vertices in parallel. For now we will assume a lifting strategy that lifts all vertices at once until all of their measures stabilized; further strategies are discussed in Section 3.4.

When a vertex $v \in V$ is lifted, all of its outgoing edges $(v, w) \in E$ must be iterated to find either the minimum or maximum value of $\mathrm{prog}(\rho, v, w)$, depending on the owner of the vertex. For the GPU algorithm, this process is split into two separate stages. First, PROG_KERNEL is launched for every edge $(v, w) \in E$ to calculate the value of $\text{prog}(\rho, v, w)$ and store it in $\rho_E[(v, w)]$. Then, LIFT_KERNEL is launched for every vertex $v \in V$ to find either the minimum or the maximum measure – depending on the owner of the vertex – stored in $\rho_E[(v, w)]$ for all $(v, w) \in E$, and to store this in $\rho[v]$.

Splitting up the process of lifting into these separate stages introduces memory safety, because both kernels write to different memory than they read from: PROG_KERNEL reads measures from ρ and writes them to ρ_E, while LIFT_KERNEL reads measures from ρ_E and writes them to ρ. This way, a thread will never overwrite a value that another thread is currently reading.

It is possible to create two stages in a different way by using distributivity of min over prog when v is fixed: $\min((v, w) \in E \ : \ \text{prog}(\rho, v, w)) = \text{prog}(\rho, v, w)$ with $(v, w) = \min((v, w) \in E \ : \ \rho(w))$. The first stage would then calculate for every vertex v its successor w with minimal $\rho(w)$, and store the corresponding measure in a separate array ρ'. The second stage then applies $\text{prog}(\rho, v, w)$ to the measure of this successor, stored in $\rho'(v)$, and stores the result in $\rho(v)$. This is more efficient, since the number of extra measures stored and the number of calls to prog are equal to $|V|$ instead of $|E|$, but it is incompatible with edge priorities introduced in Chapter 6. Because of this, the previously described stages, PROG_KERNEL and LIFT_KERNEL, are used.

prog_kernel

The first kernel, PROG_KERNEL, effectively assigns the value of $\text{prog}(\rho, v, w)$ to $\rho_E[e]$ when called with argument $e = (v, w)$. For this, it assigns $\rho[w]$ (the measure of w) to $\rho_E[e]$, but only up to priority $p(v)$ (the priority of v). After this assignment we have that $\rho_E(e)$ is the least value such that $\rho_E[e] \geq_{p(v)} \rho[w]$. However, if $p[v]$ is odd we need to have the least value for which $\rho_E[e] >_{p(v)} \rho[w]$. In this case the value needs to be increased to the smallest valid value, which is handled by the function INCREASE.

```
PROG_KERNEL($e$)
**Input:** an edge $e = (v, w)$ with $e \in E$
**Output:** nothing – updates $\rho_E[e]$
(1)     $\rho_E[e] \leftarrow_{p(v)} \rho[w]$
(2)     if $p(v)$ is odd
(3)         $\rho_E[e] \leftarrow$ INCREASE($\rho_E[e], p(v)$)
```

The function INCREASE returns the smallest measure that is strictly greater than the input measure m when compared up to priority p; that is, it returns the smallest $m' \in \mathbb{M}_{\diamond}^{\top}$ such that $m' >_p m$. This is done under the assumption that all fields in the measure for priorities greater than p are set to zero. The function loops over all priorities from p down to 0, and increments the value stored at that position i of the given measure; that is, it increments $m[i]$. It then checks whether the measure is still valid by comparing $m[i]$ with $M^{\uparrow}[i]$ to check if the value is still within bounds. If $m[i]$ is not greater than $M^{\uparrow}[i]$ the measure is still valid and it is returned. Otherwise, $m[i]$ is set to 0 and i is decremented to try the next lower priority. If there are no further priorities to increment, then no value greater than m exists in $\mathbb{M}_{\diamond}^{\top}$ and $\top$ is returned instead.

```
INCREASE($m, p$)
Input: a measure $m \in \mathbb{M}_{\diamond}^{\top}$
Output: a measure $m' \in \mathbb{M}_{\diamond}^{\top}$ such that $m'$ is the least $m' >_p m$
(1)   if $m \neq \top$
(2)     for $i \leftarrow p$ down to 0
(3)       if $m[i] < M^{\uparrow}[i]$
(4)         $m[i] \leftarrow m[i] + 1$
(5)         return $m$
(6)       else
(7)         $m[i] \leftarrow 0$
(8)   return $\top$
```

The running time of INCREASE depends on the maximum priority to loop over in a warp. The number of priorities to loop over can be at most d, the maximum priority in the game plus one, so the loop on line 2 will be executed at most $\mathcal{O}(d)$ times. In every loop, a constant number of operations is performed, resulting in a running time of $\mathcal{O}(d)$.

Measures can be stored coalesced, so we can assume that every global memory transaction involving measures has an efficiency of 100%: 1 transaction is required to read a single element of a measure for every thread. When updating the measure m in-place, there are 2 global memory transactions for every $m[i]$: one to read $m[i]$ and one to write its new value. All operations on $m[i]$ can be performed in registers, after which the new value is copied back. All threads read the same value for $M^{\uparrow}[i]$, which means that a single memory transaction with 32 values is performed, of which only 1 value is used. There is one additional memory access at the end of the function, where $\top$ is set if required. This means that there is a total of $3d + 1$ memory transactions, used for $2d + d/32 + 1$ separate values in every thread.

The kernel PROG_KERNEL first copies a measure and then optionally calls INCREASE. Copying a measure requires $\mathcal{O}(d)$ time, as does the call to INCREASE. The running time of PROG_KERNEL is therefore $\mathcal{O}(d)$.

In line 1 of PROG_KERNEL a measure is copied from a vertex to the edge for which PROG_KERNEL is called. The source measure can be anywhere in memory, giving d random memory accesses to read all d elements of that measure. In total, $32d$ read transactions are performed to read d values per thread. Writing the measures can be coalesced, so this results in only d write transactions to write d values per thread. Checking if $p(v)$ is odd in line 2 requires one memory access which cannot be coalesced: $p(v)$ is stored for the vertices at one of the endpoints of the vertex. This is another random memory access per thread, so reading this value for every thread in a warp requires 32 memory transactions. The final call to INCREASE requires $3d + 1$ memory transactions for $2d + d/32 + 1$ values per thread. In total, PROG_KERNEL requires $36d + 33$ memory transactions to read or write $4d + d/32 + 2$ values per thread. Most parity games encountered in practice have $d = 3$ or $d = 4$. For $d = 4$, the memory efficiency is 10.24%.

lift_kernel

The second kernel, LIFT_KERNEL, is launched for each vertex and assigns either the smallest or greatest measure calculated for its outgoing edges, depending on the owner of the vertex. If this new measure is strictly greater than the previous known measure, we update the value stored in the global state and mark the vertex as dirty to indicate that its measure has changed. Marking the vertex as dirty is required for detecting termination and for the strategies defined in Section 3.4

```
LIFT_KERNEL($v$)
**Input:** a vertex $v$
**Output:** nothing – updates $\rho[v]$
(1)    if $v \in V_\Diamond$
(2)        $m \leftarrow \min\left((v,w) \in E \ : \ \rho_E[(v,w)]\right)$
(3)    else
(4)        $m \leftarrow \max\left((v,w) \in E \ : \ \rho_E[(v,w)]\right)$
(5)    if $m > \rho[v]$
(6)        $\rho[v] \leftarrow m$
(7)        Mark vertex $v$ as dirty
```

The running time and number of memory transactions for LIFT_KERNEL depend on the maximum outdegree of any vertex in the parity game. Since parity games are usually sparse, $|V|$ is an overestimation of this number. Instead, O is used as the maximum outdegree of any vertex in the parity game.

Finding the minimum or maximum measure stored for any outgoing edge requires a loop over all outgoing edges and reading their measures. The number of edges per vertex is at most O, and every measure has d values.

Comparing whether the new candidate measure is actually larger than the currently stored measure, and storing it if so, requires d comparisons and optional writes. The running time of LIFT_KERNEL is therefore $\mathcal{O}(dO)$.

It is important to note for the number of memory transactions that finding the minimum or maximum measure stored for any outgoing edge can be implemented using a single code path, and therefore does not result in divergence. The actual comparison being done needs to differ, but reading values from memory can be done coalesced if the measures read by all threads are stored consecutively in memory.

The measures compared to find the minimum or maximum are stored per edge, but LIFT_KERNEL is launched per vertex. This means that reading the measures of all outgoing edges cannot be done coalesced, and requires $32dO$ memory transactions to read dO values per thread. If the number of elements is small, the value of m can be stored in shared memory and accessing it requires no global memory transactions. Comparing m with $\rho[v]$ can be done coalesced, so this results in d memory transactions for d values, as does writing m to $\rho[v]$ if it needs to be updated. Finally, marking v as dirty requires a single write to a boolean array, which is a single coalesced memory transaction. The total number of memory transactions is $(32O + 2)d + 1$ to read or write $(O + 2)d + 1$ values. For $d = 4$ and a maximum outdegree of $O = 20$, this gives a memory efficiency of 3.46%.

Host algorithm

To let the GPU algorithms do something useful, they need to be launched at the correct moments and in the correct order. This requires an algorithm for the host. The responsibilities of this algorithm are to initialize the required data structures on the GPU and to launch the kernels defined above. This algorithm is called SPM_GPU

Measures are initialized to $(0, 0, \ldots, 0)$ for every vertex by default. However, some measures can already be initialized to $\top$. If the vertex has an odd priority and a self-loop, then the self-loop is used if either player odd owns the vertex, or if player even owns the vertex and has no other choice. If this is the case, then player odd will always win that vertex and thus the measure can be initialized to $\top$.

```
SPM_GPU($G$)
Input: a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$
Output: the measures for each vertex
(1)  foreach $v \in V$
(2)      if $p(v)$ is odd $\wedge (v, v) \in E \wedge (v \in V_\Box \vee \neg(\exists\ (v, w) \in E\ :\ v \neq w))$
(3)          $\rho[v] \leftarrow \top$
(4)      else
(5)          $\rho[v] \leftarrow (0, 0, \ldots, 0)$
(6)  repeat
(7)      Mark all vertices $v \in V$ as not dirty
(8)      launch PROG_KERNEL($e$) for each $e \in E$
(9)      launch LIFT_KERNEL($v$) for each $v \in V$
(10) until no vertex $v \in V$ is marked as dirty
(11) return $\rho$
```

3.4 Lifting strategies

As mentioned in Section 2.2, an important aspect for the speed of the small progress measures algorithm is the lifting strategy used. On a CPU, a lot of lifting strategies can be employed since vertices are lifted in sequential order due to the sequential nature of a CPU. Often-used strategies include adding the predecessors of a lifted vertex to a working queue or stack, or more advanced strategies based on the structure of the graph, e.g. lifting strongly connected components as sub-graphs.

The choice of lifting strategies on a GPU is more limited due to its parallel nature. As explained in Section 1.3, GPU algorithms should launch a very large number of threads to be able to hide latencies in thread execution effectively. This means that the number of threads launched at once should lie in the order of the number of vertices or edges in a large graph.

Given the parallel nature of a GPU, and that the number of threads should be roughly the same as the number of vertices, an obvious strategy is to lift all vertices in parallel until no more measures change. This is efficient if all vertices need constant lifting, but in general most graphs only have a relatively small number of vertices which require a lot of lifting. This means that most calculations will be wasted on re-calculating already known values, which can be avoided by employing a better strategy.

This section provides two possible strategies: implicit and explicit queuing. In the case of implicit queuing, every thread checks whether the vertex assigned to it could be lifted. Explicit queuing works by creating a list of all vertices and edges that need processing, and only launching threads for vertices and edges in those queues.

3.4.1 Implicit queuing

In the case of implicit queuing, a thread will be launched for every edge (for PROG_KERNEL) and for every vertex (for LIFT_KERNEL), but the kernel will first read a boolean value to check whether it is useful to do any processing in the thread. The effect is that if there is an entire warp of threads that does not need to do any work, they only do a lookup of a boolean in global memory before terminating.

To detect whether any work has to be done, the algorithm as described in the previous section marks vertices as dirty if their measure changed in the last computation step. This information can be used to determine whether any work needs to be done for an edge or vertex. The value of $\text{prog}(\rho, v, w)$ as calculated by PROG_KERNEL depends on $\rho[w]$ (the measure of w) and $p(v)$ (the priority of v). Since $p(v)$ is fixed, PROG_KERNEL can detect whether to do any work by checking if vertex w was marked as dirty in the previous step. If vertex w was not marked as dirty, then the measure of w has not changed, and the value of $\text{prog}(\rho, v, w)$ will not change from the previously calculated value, which is still available in $\rho_E[v]$.

```
IMPLICIT_PROG_KERNEL($e$)
Input: an edge $e = (v, w)$ with $e \in E$
Output: nothing – updates $\rho_E[e]$
(1)    if vertex $w$ is marked as dirty
(2)        PROG_KERNEL($e$)
(3)        Mark vertex $v$ for recalculation
```

Since IMPLICIT_PROG_KERNEL consists of only 2 constant-time operations and a call to PROG_KERNEL, the running time is equal to that of PROG_KERNEL: $\mathcal{O}(d)$.

The memory efficiency is also similar: 2 non-coalesced memory access are added, resulting in $36d+97$ memory transactions to read or write $4d+d/32+4$ values per thread. For $d = 4$, the memory efficiency is 8.35%.

If the value of $\rho_E[(v, w)]$ was updated, then $\rho[v]$ needs to be recalculated, because its value might change. This is indicated by marking vertex v for recalculation in IMPLICIT_PROG_KERNEL. This information is then used by IMPLICIT_LIFT_KERNEL to let all threads easily check whether they need to do any work. Note that multiple threads might all mark a single vertex for recalculation. Every vertex can safely unmark itself for recalculation in IMPLICIT_LIFT_KERNEL, because the value for each vertex will not be read by any thread other than the thread setting the value.

```
IMPLICIT__LIFT__KERNEL($v$)
Input: a vertex $v$
Output: nothing – updates $\rho[v]$
(1)  if vertex $v$ is marked for recalculation
(2)      LIFT__KERNEL($v$)
(3)      Unmark vertex $v$ for recalculation
```

Since IMPLICIT__LIFT__KERNEL consists of only 2 constant-time operations and a call to LIFT__KERNEL, the running time is equal to that of LIFT__KERNEL: $\mathcal{O}(dO)$.

The memory efficiency is also very similar: only 2 coalesced memory access are added, resulting in $(32O + 2)d + 3$ memory transactions to read or write $(O+2)d+3$ values per thread. For $d = 4$ and maximum outdegree $O = 20$, the memory efficiency is 3.54%.

The host algorithm needs minimal changes to support implicit queuing. The only differences are in the marking of vertices as dirty or not dirty. Initially, all vertices need to be marked as dirty to ensure that all values of $\rho_E[e]$ will be calculated at least once. Subsequently, all vertices need to be marked as not dirty only after IMPLICIT__PROG__KERNEL was called; otherwise, that marking cannot be used to determine whether to do work for each edge.

```
IMPLICIT__SPM__GPU($G$)
Input: a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$
Output: the measures for each vertex
(1)  Initialize $\rho$
(2)  Mark all vertices $v \in V$ as dirty
(3)  repeat
(4)      launch IMPLICIT__PROG__KERNEL($e$) for each $e \in E$
(5)      Mark all vertices $v \in V$ as not dirty
(6)      launch IMPLICIT__LIFT__KERNEL($v$) for each $v \in V$
(7)  until no vertex $v \in V$ is marked as dirty
(8)  return $\rho$
```

A possible performance issue with implicit queuing is that there is a thread being launched for every edge or vertex, even if there is no work to do. To avoid this, kernels would need to be launched for only a subset of all edges and vertices. This is what explicit queuing does.

3.4.2 Explicit queuing

The explicit queuing strategy creates a work queue for both edges and vertices, and only launches kernels for edges and vertices when it is useful to do so. This does not mean that vertices and edges are added to these queues based on purely individual conditions. That would in most cases lead to inefficient memory usage, since memory coalescing would no longer be possible. This inefficiency is avoided by placing entire warps into the queue at once if at least one of the threads in that warp needs to be scheduled. Every warp contains threads working on consecutive edges or vertices.

Calculating the threads in a work queue is a relatively costly operation, which requires some additional kernels and global state. Because calculating a work queue is costly, it is not done after every step of the algorithm, but only after a certain number of steps. The following additional global state is required for explicit queuing:

$T : V \to \mathbb{B}$ — An array containing a boolean for each vertex, indicating whether that vertex must be added to the work queue.

$T_E : E \to \mathbb{B}$ — An array containing a boolean for each edge, indicating whether that edge must be added to the work queue.

$Q : [V]$ — A list containing all vertices in the work queue.

$Q_E : [E]$ — A list containing all edges in the work queue.

The ways to determine whether to add an edge or vertex to the work queue are the same as for implicit queuing. An edge (v, w) must be added to the work queue if $\rho[w]$ (the measure of endpoint w) has changed. A vertex v must be added to the work queue if $\rho_E[(v, w)]$ will be updated for any $(v, w) \in E$; in other words, a vertex v must be added to the work queue if vertex w was marked as dirty for any edge $(v, w) \in E$.

The process of creating the work queue for vertices can be split into two stages. The first stage calculates a boolean $T[v]$ for every vertex v which is true if and only if the vertex must be added to the work queue. The second stage creates the actual work queue Q by concatenating all vertices v for which $T[v]$ is true. The process of creating the work queue for edges can be split similarly.

The first stage is implemented by two kernels, DIRTY_EDGES_KERNEL and DIRTY_KERNEL. First, DIRTY_EDGES_KERNEL calculates the value of $T_E[e]$ (whether edge e must be added to the work queue) for every edge $e = (v, w)$ by checking whether vertex w is marked as dirty. The values of $T_E[e]$ are synchronized over all threads in a warp by using a shared boolean d per warp. This boolean is initialized to false, and set to true by every thread which needs to have its edge added to the work queue. At the end, all threads use the value of d to set their value of $T_E[e]$.

```
DIRTY_EDGES_KERNEL($e$)
Input: an edge $e = (v, w)$
Output: nothing – updates $T_E[e]$
(1)     shared $d \leftarrow$ false
(2)     Synchronize all threads in current thread block
(3)     if vertex $w$ is marked dirty
(4)         $d \leftarrow$ true
(5)         Mark vertex $v$ for recalculation
(6)     Synchronize all threads in current thread block
(7)     $T_E[e] \leftarrow d$
```

The running time of DIRTY_EDGES_KERNEL is $\mathcal{O}(1)$, since it only contains constant-time operations. There are 3 operations that require global memory access: checking if vertex w is marked dirty, marking vertex v for recalculation, and finally setting $T_E[e]$ to the correct value. Only the latter can be done coalesced, and the other two are essentially random memory accesses. This results in 65 memory transactions to read or write 3 values per thread, giving an efficiency of 4.62%.

Once DIRTY_EDGES_KERNEL is done, DIRTY_KERNEL is launched for every vertex v to calculate the value of $T[v]$; that is, whether to add vertex v to the work queue. This is again done using the same shared memory technique as in DIRTY_EDGES_KERNEL: a shared boolean is initialized to false, then set to true by any thread which needs its assigned vertex' measure recalculated, and is finally used to set the values of $T[v]$ for every thread in the warp. Note that this shared variable is, again, shared per warp. If a block size is chosen such that each block contains multiple warps, then there must be multiple shared variables in the block, such that there is one per warp.

```
DIRTY_KERNEL($v$)
Input: a vertex $v$
Output: nothing – updates $T[v]$
(1)     shared $d \leftarrow$ false
(2)     Synchronize all threads in current thread block
(3)     if vertex $v$ is marked for recalculation
(4)         $d \leftarrow$ true
(5)     Synchronize all threads in current thread block
(6)     $T[v] \leftarrow d$
```

The running time of DIRTY_KERNEL is $\mathcal{O}(1)$, since it only contains constant-time operations. There are 2 operations that require global memory access: checking if vertex v is marked for recalculation, and setting $T[v]$ to the current value. Both can be done using coalesced memory access, since both read from or write to an array of booleans per vertex. This results in 2 memory transactions to read or write 2 values per thread, giving an efficiency of 100%.

The second stage in calculating the work queue is to create the work queues Q and Q_E, containing the vertices v and edges e for which $T[v]$ and $T_E[e]$ are true, respectively. To create the working queue for vertices Q, the idea is to create an array containing all vertices $v \in V$. This array is coupled to the matching elements in T. The work queue Q then must contain exactly all values in the vertex-array where $T[v]$ is true. The technique for doing this is implemented in a number of support libraries for CUDA, such as Thrust[1].

The host algorithm must be modified to calculate the work queue at appropriate intervals, and to ensure that PROG__KERNEL and LIFT__KERNEL are only called for vertices and edges in the work queue.

```
EXPLICIT__SPM__GPU($G$)
Input: a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$
Output: the measures for each vertex
(1)  Initialize $\rho$
(2)  $Q, Q_E \leftarrow V, E$
(3)  while $Q \neq []$
(4)      Mark all vertices $v \in V$ as not dirty
(5)      for $i \leftarrow 1$ to $N$
(6)          launch PROG__KERNEL($e$) for each $e \in Q_E$
(7)          launch LIFT__KERNEL($v$) for each $v \in Q$
(8)      Unmark all vertices $v \in V$ for recalculation
(9)      launch DIRTY__EDGES__KERNEL($e$) for each $e \in E$
(10)     launch DIRTY__KERNEL($v$) for each $v \in V$
(11)     $Q \leftarrow [v \in V \ : \ T[v]]$
(12)     $Q_E \leftarrow [e \in E \ : \ T_E[e]]$
(13) return $\rho$
```

[1]http://thrust.github.io/

4 Implementing Small Progress Measures

An implementation of a CPU and GPU version of the small progress measures algorithm is created under the name `spm`. This implementation is used to do performance tests of the GPU algorithm, to compare it fairly to a mostly identical CPU implementation, and to test various other methods of speeding up the small progress measures algorithm. This chapter describes the implementation of `spm`.

Section 4.1 describes the implementation of the GPU algorithm. Section 4.2 describes the implementation for a CPU. Section 4.3 describes how these implementations can be combined to run in parallel.

4.1 GPU implementation

The GPU version of the small progress measures algorithm was described in Chapter 3, including two possible strategies. To obtain an implementation which is fast on a GPU, it is important to ensure that memory is accessed coalesced as often as possible, and that the impact of divergence is minimized.

4.1.1 Memory layout

The parity game is stored in memory using a number of arrays. For every property of a vertex or edge an array is created containing that property for every vertex or edge. For every vertex there are 4 properties: its priority, its owner, the index of its first outgoing edge, and the number of outgoing edges it has. For every edge there are 2 or 3 properties: its source, its target, and optionally its priority if edge priorities from Chapter 6 are used. By using separate arrays for every property, memory accesses to read these values can be coalesced: every block of successive threads reads successive values from memory.

Most of the memory used by the small progress measures algorithm is used to store measures assigned to vertices or edges in, respectively, ρ and

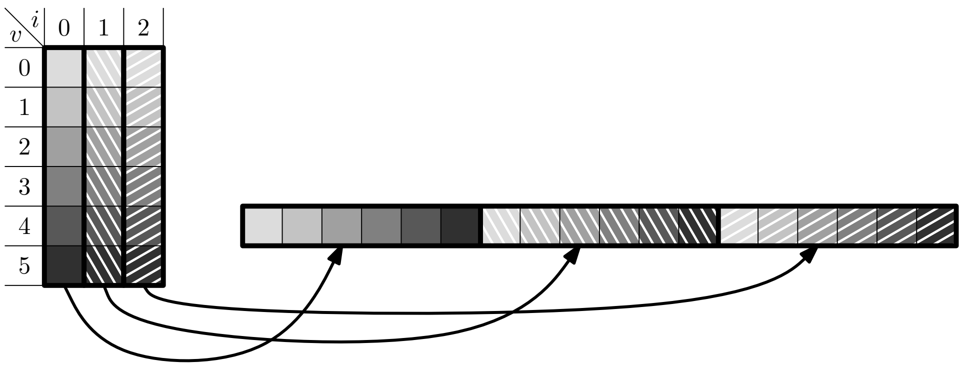


Figure 4.1: Storing the measures of 6 vertices, with each measure having 3 elements. In the table on the left, vertices are rows and measure element indices are columns. In the 1-dimsional array in memory, shown on the right, first all elements with index 0 *are stored, then all elements with index* 1*, and so on.*

ρ_E. To ensure that these measures can be read efficiently, it is important to choose a memory layout which supports coalesced memory access. To allow coalesced memory access, all threads in a warp must read from a consecutive 32-bits integer in global memory.

The arrays of measures, ρ and ρ_E, are stored in global memory as 1-dimensional arrays. These arrays represent 2-dimensional data: every element represents the value at one index of a measure for one vertex or edge. If the indices within the measure are columns and the vertices are rows, then the 2-dimensional data is packed into the 1-dimensional array in column-major order. This means that first all first elements of all measures are stored, followed by all second elements, etc. See Figure 4.1 for a visual explanation.

The values stored in each measure element are 32-bit unsigned integers. There is no extra space allocated for explicitly storing a value of $\top$ for a measure. This is encoded in the first bit of the first element of a measure: if this bit is 1, then the value of the measure is $\top$; otherwise, the value of the measure consists of the stored elements.

Finally, there is an array of booleans, one per vertex, to indicate whether each vertex is marked as dirty.

4.1.2 Strategies

Section 3.4 introduces 2 strategies for the GPU algorithm: implicit queuing and explicit queuing.

Implicit queuing

Implementing the implicit queuing strategy is straightforward. The only extra state information required is an array where vertices are marked for recalculation. This is stored as an array of booleans, similar to the array used for marking vertices as dirty.

Explicit queuing

The implementation of the explicit queuing strategy is more involving. The extra state information required consists of the arrays T and T_E, to indicate whether each vertex and each edge must be added to the work queue, and of the work queues Q and Q_E. The arrays T and T_E are again arrays of booleans, while the work queues are arrays of vertex and edge indices, plus a count to indicate how many elements in the array are actually used.

The difficult part of the explicit queuing strategy is creating the working queue by filling it with all vertices v for which $T[v]$ is true – or all edges e for which $T[e]$ is true; the process is the same for both. Filling the work queue is handled by the COPY_IF function of the Thrust library[1]. This library contains a large number of routines for GPU-accelerated data processing. The COPY_IF function requires an array of values and a so-called *stencil array* of booleans, and will copy all values for which the boolean in the stencil array is true to a target list. The array of values is an implicit array containing the indices of all vertices as its elements and T (or T_E) is used as the stencil array. The target array is the work queue Q (or Q_E).

4.1.3 Reducing global memory accesses

The slowest instructions on a GPU are those accessing the global memory, and especially instructions reading from the global memory: program execution cannot continue until the requested value is read. This can take hundreds of cycles, while other instructions typically finish in only 1 or possibly a few cycles. If the operation of a kernel is memory-bound, a significant speed-up can be obtained by reducing the number of values read from global memory. Since the small progress measures algorithm mainly moves data around and increments some values, it is memory-bound in practice, and reducing global memory accesses is a valid way to lower the actual execution times.

One of the places where values are read more often than is necessary is in PROG_KERNEL. First a measure is copied, and then it is possibly increased. It is possible to read the value from memory only once, optionally increase

[1]http://thrust.github.io/

it, and then write it to memory only once. The intermediate value is stored in a register and can thus be accessed quickly.

Another way to reduce global memory access is by copying data to shared memory during processing. This technique can be used to speed up LIFT_KERNEL. When selecting either the minimum or maximum measure of a vertex' outgoing edges, the current minimum or maximum measure can be stored in shared memory. This means that every comparison between the current minimum measure and a new candidate happens between 1 value in global memory and 1 value in shared memory, instead of between 2 values in global memory. Once the minimum measure has been found, it can be copied to the correct place in global memory using only reads from shared memory.

A limitation of this approach is that the amount of available shared memory is relatively low: on devices with CUDA compute capability 2.1, there is only 48 KiB of shared memory per multiprocessor. These devices can have 1536 threads resident on each multiprocessor, and since every measure value is a 32-bit integer, there is only enough shared memory to store 8 values per vertex in shared memory. Therefore, this technique can only be used when measures have at most 8 values.

Note that using shared memory can introduce bank conflicts, which can negate the performance gain obtained by using shared memory. To avoid bank conflicts, the measures need to be stored in the same order as used in global memory to enable coalesced memory access.

4.1.4 Reducing divergence

Another source of slow kernels is divergence. If different threads in a warp require different execution paths, they both need to execute both paths but can only use the result of one path.

The most important cause of divergence in the small progress measures algorithm is comparisons. It is easy to create a set of functions which do less-than, less-equal, greater-equal and greater-then comparison, but by creating 4 different functions a source of divergence is introduced. A single comparison function is created instead, which has 4 arguments: the 2 measures to compare, whether to return true or false if the first measure is smaller than the other, and the value to return when both are equal. The calls to the separate comparison functions can then be replaced by a call to the single comparison function with proper arguments.

A place where this is very useful is when determining either the minimum or the maximum measure assigned to the outgoing edges of a vertex in LIFT_KERNEL. Whether the minimum or maximum measure must be selected depends on the owner of the vertex, which can easily be translated into the correct argument for the function call. If both calculations are done independently instead, the amount of code executed is, usually, doubled.

4.2 CPU implementation

To able to objectively compare the influence of running small progress measures on a GPU, a nearly identical CPU implementation was made. Both implementations share a large part of the codebase, including basic routines for working with measures and graph preprocessing. Both implementations also use the same memory layout. The differences are in the code doing the lifts, and the employed strategies. The code for the lifts consists of kernels being launched for the GPU algorithm, but these concepts do not apply for the CPU algorithm. The strategies differ because the CPU can employ different strategies due to its serial instead of parallel nature.

The reason to use a nearly identical CPU implementation is to be able to objectively compare the the influence of running the algorithm on the GPU. The biggest difference between an implementation specifically tailored for the CPU and the implementation inspired by the GPU implementation is the memory layout. The effects of this on the running time would be the result of caching effects on the CPU, but these effects are negligible.

4.2.1 Strategies

There are 3 strategies implemented for the CPU: a queue, a stack and a random strategy. These are simple strategies that seem to work well in practice. The queue and stack strategies are completely deterministic, and it is possible to construct parity games in such a way that these strategies will always use the worst-case lifting order when solving the games. The random strategy does not suffer from this drawback, since it selects the next vertex to lift in a non-deterministic way.

All three strategies are based on the idea that the measure assigned to a vertex can only change if the measure of one of their successors changed. The predecessors of all vertices are stored in memory, and when the measure of a vertex is updated, all of its predecessors are added to the work queue. The order in which the vertices are extracted from the work queue differs per strategy. Every strategy has a *single-occurrence* version where vertices can be present in the working queue only once at a time.

Queue

The queue strategy employs a basic first-in-first-out queue. To implement the single-occurrence variant, a boolean is stored for every vertex which is true if and only if the vertex is present in the queue. This results in a queue with $\mathcal{O}(1)$ push and pop operations, and $\mathcal{O}(n)$ memory usage.

Stack

The stack strategy is based on a last-in-first-out queue. The implementation of the single-occurrence variant of a stack is a bit more involving, since a vertex added to the stack must be moved to the top of the stack if it is already present. The stack is implemented as a doubly-linked list, and for every vertex a pointer to its element in the list is stored, if present. If a vertex is then pushed onto the stack, it is checked whether its pointer is set. If it is set, it is removed from the list. Finally, that vertex is added to the top of the stack and its pointer is saved. Both push and pop operations are $\mathcal{O}(1)$, and memory usage is $\mathcal{O}(n)$.

Random

The random strategy assigns a random priority to every vertex when it is pushed into the working queue, and puts the vertex in a priority queue using the random priority as its weight. Vertices are extracted from the priority queue in increasing order of their priority. The single-occurrence variant is implemented similar to the queue strategy: a boolean is stored for every vertex which is true if and only if the vertex is present in the queue. The resulting push and pop operations are $\mathcal{O}(\log n)$, and memory usage is $\mathcal{O}(n)$.

4.3 Combining the CPU and GPU

The GPU and CPU algorithms have different characteristics: the CPU is versatile since lifts are handled one by one, while the GPU can handle multiple lifts in parallel but requires large amounts of lifts to be launched at the same time. Different parity games might require one or the other, or some might even fare well by combining both implementations. To facilitate this, both implementations can be run in parallel and their intermediate results can be shared between both games.

Both implementations are initialized and executed independently, with the exception of sharing their results between each other periodically. There is also a boolean flag which is read by both implementations, which indicates when one of the implementations has finished their calculations. When one implementation has finished, that implementation can provide the winners of every vertex in the game, and the other implementation can be terminated.

The synchronization is not done after a certain number of lifts, but is scheduled with intervals of at least one second. The exact interval depends on when an implementation is ready for synchronization: for the CPU, this can be done after every lift, but the GPU implementation requires all launched kernels to finish before synchronization can be performed.

A synchronization buffer is used to synchronize the measures; the buffer contains a measure for every vertex. Both implementations independently synchronize their measure to and from this synchronization buffer. Mutual exclusion is ensured: if the synchronization buffer is in use by one implementation and the other implementation attempts to synchronize, synchronization is skipped and attempted again at the next scheduled time.

Synchronization starts by locking the synchronization buffer. The synchronization buffer and the implementations array of measures are then compared: for every vertex, the maximum measure of both the buffer and the implementations array is selected and stored in both the buffer and the array. If the measure stored for a vertex in the implementations array of measure is updated, this vertex needs to be marked as dirty (or added to the working queue on the CPU) to ensure that the updated measure is used for further calculation. Once this is done, both the buffer and the array are updated, the synchronization buffer can be unlocked, and synchronization is complete. For the GPU implementation, the synchronization buffer is copied to the global memory of the GPU and comparing the measures is done on the GPU. The new synchronization buffer is copied back from the GPU to the host memory.

It is allowed to select the maximum value of the measures in the buffer and the implementations measure array because the measures can only increase during computation. When a bigger measure is stored in the synchronization buffer for a vertex, then the other implementation has advanced that measure more than the current implementation. Using this bigger measure allows the current implementation to skip a number of lifts, which it would have to calculate otherwise.

5 Alternating SPM

Until now we were only concerned with solving a parity game for one specific player. Some parity games, however, might prove difficult to solve for one player, but can be easy for the other player. Unfortunately, no known way exists to find out the easiest player to solve the game for beforehand. An alternative way to solve this problem is to solve the game for both players in parallel, and then stop whenever the game is solved for one of the players. The winning regions for both players can then be determined. When this technique is used in a sequential algorithm, calculations for both players are alternated instead of run in parallel, giving rise to the name *alternating SPM*.

When both games are being solved in parallel, it can be beneficial to share intermediate results between both calculations. Determining the certainly losing vertices for player $\circ$ is easy by simply looking at the measures which are $\top$, but it is difficult to do something useful in the game for player $\overline{\circ}$ with this information. It is more difficult to determine the certainly winning vertices for player $\circ$ in a partially solved game, but it is trivial to use this information in the game for player $\overline{\circ}$: all vertices certainly winning for player $\circ$ are certainly losing for player $\overline{\circ}$, and their measures can thus be set to $\top$ in the game for player $\overline{\circ}$.

This technique has already been implemented in both PGSolver and mCRL2's pbespgsolve, but neither tool provides documentation on how and why this technique works.

Section 5.1 describes how the winning set for a certain player is calculated. Section 5.2 describes how alternating SPM is implemented on both the CPU and the GPU.

5.1 Determining winning vertices

A parity progress measure was defined in Definition 2.2 to indicate when vertices are winning for player $\diamond$. The small progress measures algorithm iteratively updates an intermediate measure until it reaches a final valid value that adheres to the definition. Intermediate values in general say nothing useful about the game.

In Section 1.2 it was shown that a valid parity game G' can be obtained by only using a subset of the vertices W of a game G. An intermediate progress measure might be useful to say something about a smaller game $G' = G \cap W$. The measures for all vertices in W can together form a valid parity progress measure for only G'. It is important to ensure that the strategy for player $\Diamond$ on all vertices in $W \cap V_\Diamond$ is closed on W; otherwise, there is not enough information to be certain about the winning vertices, and additionally the totality of the graph can not be guaranteed.

Using the definition for parity progress measures, a definition for the set $\mathcal{W}_\Diamond$ can be given, such that all vertices $v \in \mathcal{W}_\Diamond$ will certainly be won by player $\Diamond$. This definition is valid for a, possibly partially-computed, game parity progress measure ρ that is calculated using the small progress measures algorithm. This means that ρ is at most the least game parity progress measure, but possibly smaller than that.

$$
\begin{aligned}
&v \in \mathcal{W}_\Diamond \implies \\
&\qquad \rho(v) \neq \top \\
&\quad \wedge \ \Big(\exists\ (v,w) \in E \ : \ w \in \mathcal{W}_\Diamond \wedge \rho(v) \geq_{p(v)} \mathrm{prog}(\rho, v, w)\Big) \quad \text{if } v \in V_\Diamond \\
&\quad \wedge \ \Big(\forall\ (v,w) \in E \ : \ w \in \mathcal{W}_\Diamond \wedge \rho(v) \geq_{p(v)} \mathrm{prog}(\rho, v, w)\Big) \quad \text{if } v \in V_\Box
\end{aligned}
$$

This definition is similar to the definition of a game parity progress measure. The only differences are the inclusion of the '$w \in \mathcal{W}_\Diamond$' clauses, to ensure that the strategy is closed on $\mathcal{W}_\Diamond$, and the additional restriction that $\rho(v) \neq \top$, since those vertices are won by player $\Box$. The game $G \cap \mathcal{W}_\Diamond$ is still total: all $v \in V_\Diamond \cap \mathcal{W}_\Diamond$ must have at least one outgoing edge, and all $v \in V_\Box \cap \mathcal{W}_\Diamond$ can only be in $\mathcal{W}_\Diamond$ if all outgoing edges of v in G are included.

To aid in proving that all vertices in $\mathcal{W}_\Diamond$ are indeed won by player $\Diamond$, some additional notation is introduced. The function $\rho_{\mathcal{W}_\Diamond} : \mathcal{W}_\Diamond \to \mathbb{M}_\Diamond^\top$ is the function ρ limited to all vertices in $\mathcal{W}_\Diamond$. Note that only the domain is changed and not the range; the type $\mathbb{M}_\Diamond^\top$ is still based on the entire game G.

The strategy $\sigma_{\mathcal{W}_\Diamond}$ is the strategy for player $\Diamond$ on all their vertices in $\mathcal{W}_\Diamond$, as obtained from $\rho_{\mathcal{W}_\Diamond}$. This strategy is obtained similar to the strategy $\sigma_\Diamond$ on the entire game as given in Definition 2.6, with the exception that only successors in $\mathcal{W}_\Diamond$ are considered. So $\sigma_{\mathcal{W}_\Diamond}(v) = v'$ s.t. $v' \in \mathcal{W}_\Diamond \wedge \rho(v') = \min\left(\rho(w) \ : \ (v,w) \in E \wedge w \in \mathcal{W}_\Diamond\right)$ for all $v \in V_\Diamond \cap \mathcal{W}_\Diamond$.

Theorem 5.1. *All vertices in $\mathcal{W}_\Diamond$ will be won by player $\Diamond$: $\mathcal{W}_\Diamond \subseteq W_\Diamond$.*

Proof. As noted in Section 2.1, it is only possible to form a parity progress measure for player $\Diamond$ if all vertices for the game are won by player $\Diamond$. To prove that player $\diamond$ wins from all vertices in $\mathcal{W}_\diamond$, it must be proven that the strategy $\sigma_{\mathcal{W}_\diamond}$ is closed on $\mathcal{W}_\diamond$, and that $\rho_{\mathcal{W}_\diamond}$ is a valid parity progress measure on $G' = G \cap \mathcal{W}_\diamond$.

1. *The strategy $\sigma_{\mathcal{W}_\diamond}$ is closed on $\mathcal{W}_\diamond$*

   The strategy $\sigma_{\mathcal{W}_\diamond}$ is defined for all vertices owned by player $\diamond$ in $\mathcal{W}_\diamond$. For all these vertices, the successor w with $w \in \mathcal{W}_\diamond$ and minimal value for $\text{prog}(\rho, v, w)$ is chosen. Additionally, if $v \in \mathcal{W}_\diamond$ is owned by player $\square$, then no possible move may end up outside of $\mathcal{W}_\diamond$.

   If $v \in \mathcal{W}_\diamond \cap V_\diamond$, then the definition of $\mathcal{W}_\diamond$ tells that there is a successor w of v, such that $w \in \mathcal{W}_\diamond \wedge \rho(v) \geq_{p(v)} \text{prog}(\rho, v, w)$. Because ρ is being constructed using the small progress measures algorithm, we know that $\rho(v) \leq \min(v, w \in E \ : \ \text{prog}(\rho, v, w))$. This means that there is an edge from v to some w, such that $w \in \mathcal{W}_\diamond$ and w is a possible value for $\sigma_{\mathcal{W}_\diamond}(v)$. Since $\sigma_{\mathcal{W}_\diamond}$ only considers vertices that are in $\mathcal{W}_\diamond$, and we know that there is such a vertex, we can conclude that $\sigma_{\mathcal{W}_\diamond}(v) \in \mathcal{W}_\diamond$ for all $v \in \mathcal{W}_\diamond \cap V_\diamond$.

   If $v \in \mathcal{W}_\diamond \cap V_\square$, then by definition of $\mathcal{W}_\diamond$, all successors of v are included in $\mathcal{W}_\diamond$.

2. *$\rho_{\mathcal{W}_\diamond}$ is a valid parity progress measure on $G' = G \cap \mathcal{W}_\diamond$*

   Recall that $\rho_{\mathcal{W}_\diamond}$ is a valid parity progress measure if, for all $v \in \mathcal{W}_\diamond$, $\rho_{\mathcal{W}_\diamond}(v) \in \mathbb{M}_\diamond$, and:

$$\begin{array}{rll} & \Big(\exists\ (v, w) \in E \ : \ \rho_{\mathcal{W}_\diamond}(v) \geq_{p(v)} \text{prog}(\rho_{\mathcal{W}_\diamond}, v, w)\Big) & \text{if } v \in V_\diamond \\ \wedge & \Big(\forall\ (v, w) \in E \ : \ \rho_{\mathcal{W}_\diamond}(v) \geq_{p(v)} \text{prog}(\rho_{\mathcal{W}_\diamond}, v, w)\Big) & \text{if } v \in V_\square \end{array}$$

   The function $\rho_{\mathcal{W}_\diamond} : \mathcal{W}_\diamond \to \mathbb{M}_\diamond^\top$ assigns a game parity progress measure (of type $\mathbb{M}_\diamond^\top$) to every vertex in $\mathcal{W}_\diamond$, while it must be a valid parity progress measures (of type $\mathbb{M}_\diamond$). The difference between $\mathbb{M}_\diamond^\top$ and $\mathbb{M}_\diamond$, is that $\mathbb{M}_\diamond^\top$ adds a possible value of $\top$. Since $v \in \mathcal{W}_\diamond$ implies that $\rho_{\mathcal{W}_\diamond}(v) \neq \top$, we have that $\rho_{\mathcal{W}_\diamond}(v) \in \mathbb{M}_\diamond$ for all $v \in \mathcal{W}_\diamond$.

   The remainder of the proof is a simple rewriting step: the only remaining difference between the definition of $\mathcal{W}_\diamond$ and the definition of parity progress measures is the inclusion of the $w \in \mathcal{W}_\diamond$ clauses. These clauses are known to be true, so by removing these clauses the predicate is weakened, resulting in the definition of parity progress measures.

Since both are proven to be true, it is shown that all vertices in $\mathcal{W}_\diamond$ will be won by player $\diamond$. $\square$

5.2 Implementing alternating SPM

The implementation of alternating SPM consists of two parts: the game must be solved for both players at the same time, and the winning sets of both players need to be copied to the other game.

CPU implementation

In a sequential implementation, solving the game for both players at the same time can be done in two different ways. One is to have a single game which contains measures for both players, but only has a single working queue for its strategy. This method is employed by `pgsolver` and by my implementation, `spm`. The other method, employed by mCRL2's `pbespgsolve`, is to create 2 independent games, each with their own working queue, and let them each run a number of steps in alternation.

After a certain amount of work, the results of both games need to be analyzed to determine the winning set for each player, which can then be used to set some measures to $\top$ in the complementing game. The amount of work to do between subsequent updates is measured in the number of lifts attempted. A number of $|V|$ lifts gives a good performance in practice.

Calculating the winning set is done by an iterative calculation. Initially, all vertices whose measure is not already $\top$ are marked as part of the winning set and added to a working queue. While the working queue is not empty, a vertex is popped from the queue and analyzed. If the vertex does not satisfy the criteria for being included in the winning set, then it is marked as not part of the winning set and all of its predecessors that are still part of the winning set are added to the working queue. This process continues until the working queue is empty.

```
WINNING_SET($G, \rho_\Diamond$)
Input: a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$ and corresponding (partial) game parity progress measure $\rho_\Diamond$
Output: the set of vertices certainly won by player $\Diamond$
(1)   $\mathcal{W}_\Diamond \leftarrow \{v \in V \ : \ \rho(v) \neq \top\}$
(2)   Initialize queue $Q$ with all $v \in \mathcal{W}_\Diamond$
(3)   while $\neg$EMPTY($Q$)
(4)       $v \leftarrow$ POP($Q$)
(5)       if $\Big(v \in V_\Diamond \wedge \neg\Big(\exists\ (v,w) \in E \ : \ w \in \mathcal{W}_\Diamond \wedge \rho(v) \geq_{p(v)} \text{prog}(\rho, v, w)\Big)\Big) \vee$
          $\Big(v \in V_\Box \wedge \neg\Big(\forall\ (v,w) \in E \ : \ w \in \mathcal{W}_\Diamond \wedge \rho(v) \geq_{p(v)} \text{prog}(\rho, v, w)\Big)\Big)$
(6)           $\mathcal{W}_\Diamond \leftarrow \mathcal{W}_\Diamond \setminus \{v\}$
(7)           foreach $w \in \{w \in V \ : \ (w,v) \in E \wedge w \in W_\Diamond\}$
(8)               PUSH($Q, w$)
(9)   return $\mathcal{W}_\Diamond$
```

If the queue is implemented as a single-occurrence queue, as used in the single-occurrence queue strategy for the CPU implementation of SPM, then a vertex can be in the queue at most once at any given time. An additional benefit of this is that all vertices in the queue are guaranteed to be in $\mathcal{W}_{\Diamond}$: they can only be added to the queue if they are in $\mathcal{W}_{\Diamond}$, and they can be only removed from $\mathcal{W}_{\Diamond}$ if they are first removed from the queue.

A vertex v can only be added to the queue $\text{outdegree}(v) + 1$ times: once at the beginning, and once for each of its outgoing edges. Summed over all vertices this means $\mathcal{O}(|E|)$ push, and therefore also pop, operations. Handling a single vertex v when it is popped from the queue has a running time of $\mathcal{O}(d|V|)$: popping is done in constant time, checking the properties of all successors, of which there are at most $|V|$, is done in $\mathcal{O}(d)$ time per successor because a measure is compared, and finally all predecessors, of which there are again at most $|V|$, can potentially be added to the queue. Summed over all $\mathcal{O}(|E|)$ pops, this gives a total running time of $\mathcal{O}(d|E||V|)$ to determine the winning set of one player.

After calculating the winning set for one player, using this in the game for the other player is straightforward: for all vertices $v \in \mathcal{W}_{\Diamond}$, $\rho_{\Box}(v)$ is set to $\top$. This is done in a simple loop, and it is obvious that the running time of the entire operation is dominated by actually finding the winning set.

GPU implementation

Implementing alternating SPM for the GPU has mostly the same considerations. It is also possible to either solve a single instance for both players, or to solve a single instance for each player. In the latter case, synchronizing the winning set is slightly more difficult: both instances need to synchronize at the same time to avoid writing to measures which are being read by a different kernel. As with the CPU implementation, my GPU implementation solves a single game for both players at the same time.

The most important difference is in the process of calculating the winning set. The CPU version contains a queue to which vertices are added if their state might change, but using such a queue is difficult on a GPU. Instead, a mechanism similar to the implicit queuing strategy for small progress measures is used. The contents of the loop of WINNING_SET are wrapped in a kernel, WINNING_SET_KERNEL, which is called for every vertex of the graph as long as some vertex changes. As an optimization, since a vertex can only be removed from $\mathcal{W}_{\Diamond}$ if it is in there, it is first checked whether the current vertex is still in $\mathcal{W}_{\Diamond}$ and if not, no further work is done in the kernel.

For the GPU, the set $\mathcal{W}_{\Diamond}$ is implemented as an array of booleans such that every vertex has a single boolean indicating whether it is in the set, i.e.

$\mathcal{W}_\Diamond : V \to \mathbb{B}$. Marking vertices as changed can be handled by a single global boolean, since it does not matter which vertex has changed, just whether any vertex has changed.

```
WINNING_SET_KERNEL($G, \rho_\Diamond, \mathcal{W}_\Diamond, v$)
Input: a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$, a corresponding (partial) game parity progress measure $\rho_\Diamond$, the current state of $\mathcal{W}_\Diamond$, and a vertex $v$ to process
Output: nothing – updates $\mathcal{W}_\Diamond$
(1)    if $\mathcal{W}_\Diamond[v] = $ true
(2)        if $\Big(v \in V_\Diamond \wedge \neg\Big(\exists\ (v, w) \in E\ :\ w \in \mathcal{W}_\Diamond \wedge \rho(v) \geq_{p(v)} \mathrm{prog}(\rho, v, w)\Big)\Big) \vee$
           $\Big(v \in V_\Box \wedge \neg\Big(\forall\ (v, w) \in E\ :\ w \in \mathcal{W}_\Diamond \wedge \rho(v) \geq_{p(v)} \mathrm{prog}(\rho, v, w)\Big)\Big)$
(3)            $\mathcal{W}_\Diamond[v] \leftarrow$ false
(4)            Mark $v$ as changed
```

The running time of this kernel is dominated by the checks on line 2: for every successor, a measure is compared. The maximum outdegree of any vertex in the parity game is O, and the number of elements in every measure is d. Therefore, the running time of WINNING_SET_KERNEL is $\mathcal{O}(dO)$.

The checks on line 2 also require the most memory transactions. A measure for every edge is compared to a measure for the current vertex. Reading the edge measure cannot be done coalesced, while reading the measure current vertex can be done coalesced. This results in $33dO$ memory transactions to read $2dO$ values per thread. One additional value is read per successor, to check if $w \in \mathcal{W}_\Diamond$. This cannot be done coalesced either, so this adds $32O$ memory transactions to read O values per thread. There are 3 additional memory accesses: reading and writing $\mathcal{W}_\Diamond[v]$, and marking v as changed. These can be done coalesced. In total this gives $(33d+32)O+3$ memory transactions to read or write $(2d+1)O+3$ values per thread. For $d = 4$ and $O = 20$, this gives an efficiency of 5.57%.

```
WINNING_SET_GPU($G, \rho_\Diamond$)
Input: a parity game $G = (V, E, p, (V_\Diamond, V_\Box))$ and corresponding (partial) game parity progress measure $\rho_\Diamond$
Output: the set of vertices certainly won by player $\Diamond$
(1)    foreach $v \in V$
(2)        $\mathcal{W}_\Diamond[v] \leftarrow (\rho_\Diamond(v) \neq \top)$
(3)    repeat
(4)        Mark all vertices $v \in V$ as not changed
(5)        launch WINNING_SET_KERNEL($G, \rho_\Diamond, \mathcal{W}_\Diamond, v$) for each $v \in V$
(6)    until no vertex $v \in V$ is marked as changed
(7)    return $\{v \in V\ :\ \mathcal{W}_\Diamond[v] = $ true $\}$
```

Initializing the array $\mathcal{W}_\diamond$ and constructing the resulting winning set to return requires $\mathcal{O}(|V|)$ operations. If checking whether any vertex has changed is done using a single global boolean, these checks require only constant time. Since WINNING_SET_KERNEL is launched for every $v \in V$, a single launch has a running time of $\mathcal{O}(d|E|)$. The kernel is launched at most $|V| + 1$ times: it is launched once at the beginning, and then it is only launched if at least one vertex has been removed from $\mathcal{W}_\diamond$. Every vertex can be removed from $\mathcal{W}_\diamond$ only once. The total running time of WINNING_SET_GPU is $O(d|V||E|)$.

6 Edge priorities

A play in a parity game is an infinite path, and, since it is sufficient to consider memoryless strategies only, must end in a cycle which is traversed an infinite number of times. Many algorithms for solving parity games, including small progress measures, attempt to find these cycles and their attractor sets, the paths leading to these cycles. When these cycles are made shorter by adding shortcuts, there are less steps to investige and solving parity games could get a speed boost. To aid in creating these shortcuts, while still taking the priorities of all vertices in account, priorities are assigned to edges instead of vertices. These edge priorities then allow shortcuts to be created in the parity graph.

Section 6.1 introduces the concept of edges with priorities. Section 6.2 describes various shortcut edges which can be added to the graph. Section 6.3 shows the modifications required to use edge priorities with small progress measures.

6.1 Adding priorities to edges

A parity game with edge priorities $G_E = (V, E, p, p_E, (V_\Diamond, V_\Box))$ is a parity game with an additional edge priority function $p_E : E \to \mathbb{N}$, which assigns a priority to every edge in the game. An edge priority function for a parity game without edge priorities can be created using the existing vertex priority function p. The priority of an edge $e = (v, w)$ is set to the priority of the vertex v:

$$p_E((v, w)) = p(v) \qquad \text{for all } (v, w) \in E$$

The winner of a play in a parity game with edge priorities is determined by the lowest edge priority occurring infinitely often in that play. Compare this to regular parity games, where the winner of a play is determined by the lowest vertex priority occurring infinitely often. In parity games with edge priorities, the winning condition has moved from vertex priorities to edge priorities.

Observe that the transformation of a normal parity game to one with edge measures, using the initialization of p_E as given above, does not change the winner of any vertex in that game. The set of possible plays is exactly the same, and the winner of those plays is also the same: all outgoing edges of a vertex have the same priority as that vertex, and leaving that vertex is only possible by traversing one of the outgoing edges.

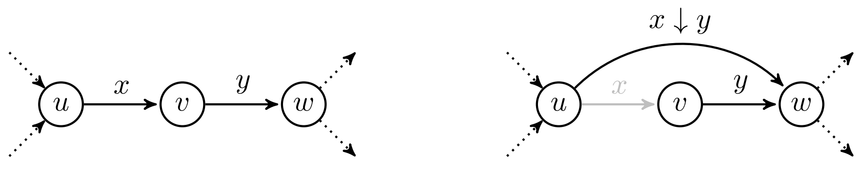
6.2 Creating shortcuts

Just adding priorities to edges does not help in solving parity games. These edge priorities, however, can be used to create shortcuts in the graph. This section describes when shortcuts can be added to a parity graph, without changing the winning sets for each player. Adding shortcuts might make some of the original edges superfluous.

The most simple case when a shortcut can be created is when 3 vertices occur on a line, with no possible choice from the first two vertices but to advance to the third vertex; see Figure 6.1 for an example. In this case, a shortcut edge can be created between the first and the last vertex, vertices u and w in Figure 6.1. This shortcut edge is used to skip over two edges, and the priority of this new edge is set to the minimum of the priorities of the two edges skipped.

Theorem 6.1. *In a parity game with vertices u, v and w, such that the only outgoing edge from u is to v with priority x, and the only outgoing edge from v is to w with priority y, adding an edge between vertices u and w with priority $\min(x, y)$ does not change the winning set of any player. Adding this edge makes the edge (u, v) superfluous.*

Proof. For every $v' \in W_{\bigcirc}$, player $\circ$ has a winning strategy from v'. If adding



(a) Original situation *(b) With shortcut*

Figure 6.1: Adding a shortcut in a part of a parity game. Vertex u has only one outgoing edge, to vertex v. Vertex v also has only one outgoing edge, to vertex w. Edge (u, v) has priority x and edge (v, w) has priority y. The owners of each vertex are irrelevant. An edge can be added between vertices u and w, with as priority the minimum of x and y. The edge between u and v is now superfluous and can be removed.

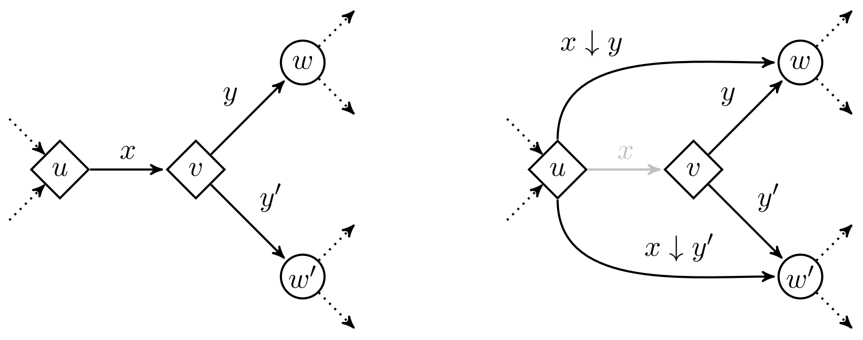
the shortcut edge does not influence the existence of a winning strategy, then the winning set of neither player will change. The winning strategy for player $\bigcirc$ from all $v' \in W_{\bigcirc}$ is $\sigma_{\bigcirc}$.

The strategies in u and v are fixed: $\sigma_{\bigcirc}(u) = v$ and $\sigma_{\bigcirc}(v) = w$. Any play visiting vertex u will therefore also visit vertex v and w. If these vertices are not visited infinitely often in any play, then the priorities of the edges between u and w, possibly via v, do not matter and can thus be anything. If these vertices are visited infinitely often in a play, then the winner of the play is the same if the original edges are used and if the shortcut edge is used; there are 3 possible situations to prove this for:

- x is the minimum priority occurring infinitely often in the play – The lowest priority occurring infinitely often is x because the edge (u, v), with priority x, is traversed infinitely often, and no edge with a lower priority is traversed infinitely often. Since the edge (u, v) can be skipped by using the shortcut edge (u, w), the shortcut edge must have priority x to not alter the winner of the play. Since x is the minimum priority occurring infinitely often, and both x and y occur infinitely often, $x \leq y$ and therefore $x \downarrow y = x$.
- y is the minimum priority occurring infinitely often in the play – This is similar to the previous case, but the shortcut edge must have priority y. Since y is the minimum priority occurring infinitely often, and both x and y occur infinitely often, $y \leq x$ and therefore $x \downarrow y = y$.
- Neither x nor y is the minimum priority occurring infinitely often in the play – The minimum priority occurring infinitely often is some priority z, and both $z < x$ and $z < y$. To ensure that the priority of the new edge does not become smaller than z, it can be set to either x or y. The result of $x \downarrow y$ is either x or y, so this is a valid priority for the new edge.

In all cases, changing $\sigma_{\bigcirc}(u)$ from v to w does not influence the minimum priority occurring infinitely often in any play. Therefore, the existence of a winning strategy from any vertex for either player is not influenced, and adding the edge (u, w) does not change the winning set of any player. Because $\sigma_{\bigcirc}(u)$ can be changed from v to w without any influence on the winner of any play, the edge (u, v) can be avoided in all plays and can thus be removed from the graph without consequence. $\square$

Note that the edge (v, w) cannot be removed: it is the only outgoing edge from vertex v and removing it would therefore result in a graph that is not total. Any play starting in v or arriving in v using a different edge than (u, v) would not have any way to leave v.



(a) Original situation (b) With shortcuts

Figure 6.2: Adding shortcuts when there is a choice in v. This is allowed if u and v have the same owner.

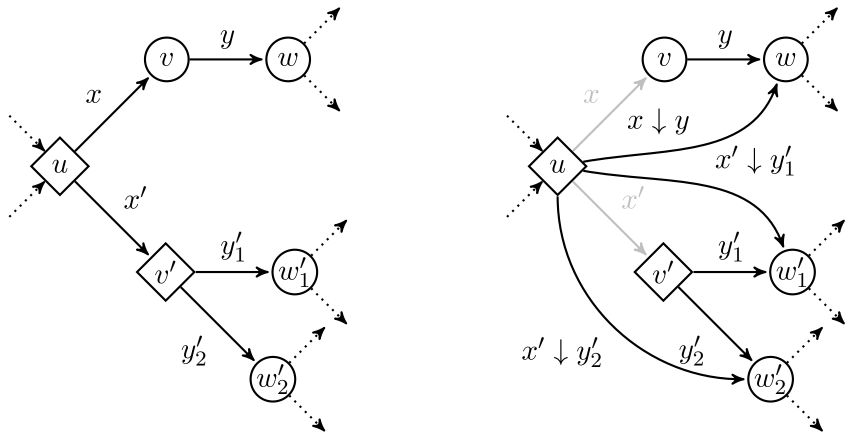
Adding shortcuts only when two consecutive vertices have only trivial choices is rather limiting. It is also possible to add a shortcut if the second of these vertices has multiple outgoing edges, but only if the owners of the first and second vertex match; see Figure 6.2 for an example. Intuitively, the choice made in the second vertex is moved to the first vertex. This is only allowed if the player making the choice does not change.

Theorem 6.2. *Consider a parity game with vertices u and v, and a set of vertices W, such that the only outgoing edge from u is to v, both vertices have the same owner, and the set W is the set of all direct successors of v. Adding an edge between u and w with priority $\min(p_E((u,v)), p_E((v,w)))$ for all $w \in W$ does not change the winning set of any player, and makes the edge (u,v) superfluous.*

Proof. For every $v' \in W_{\bigcirc}$, player $\circ$ has a winning strategy from v'. If adding the shortcut edge does not influence the existence of a winning strategy, then the winning set of neither player will change. Since strategies are memoryless, if any play visiting vertex u is won by player $\circ$, then all plays visiting vertex u will be won by player $\circ$.

The strategy in u is fixed and the strategy in v is limited: $\sigma_{\bigcirc}(u) = v$ and $\sigma_{\bigcirc}(v) \in W$. Any play visiting vertex u will therefore also visit vertex v and at least one of the vertices in W.

If player $\circ$ owns vertex u and v, and has a strategy $\sigma_{\bigcirc}$ such that he wins any play visiting vertex u in a game without shortcuts, then player $\circ$ will have a strategy $\sigma'_{\bigcirc}$ not using edge (u,v) such that he wins all plays visiting vertex u in a game with shortcuts. Any play visiting vertex u and consistent with $\sigma_{\bigcirc}$ will encounter the priorities $p_E((u,v))$ and $p_E((v,\sigma_{\bigcirc}(v)))$, and will then be in vertex $\sigma_{\bigcirc}(v)$. By adding the shortcut edge $(u,\sigma_{\bigcirc}(v))$, a valid strategy is $\sigma'_{\bigcirc} = \sigma_{\bigcirc}[u := \sigma_{\bigcirc}(v)]$. Any play visiting vertex u and consistent



(a) Original situation *(b) With shortcuts*

Figure 6.3: Adding shortcuts when there is a choice in u. Both choices can have shortcuts added independently.

with $\sigma'_\bigcirc$ will encounter priority $p_E((u, \sigma_\bigcirc(v)))$, and will then be in vertex $\sigma_\bigcirc(v)$. Since $p_E((u, \sigma_\bigcirc(v))) = \min(p_E((u, v)), p_E((v, \sigma_\bigcirc)))$, the minimum priority occurring infinitely often in a play consistent with $\sigma'_\bigcirc$ will be equal to the minimum priority occurring infinitely often in a play consistent with $\sigma_\bigcirc$, and therefore the winners of both plays are the same.

If player $\circ$ owns vertex u and v, but has no strategy $\sigma_\bigcirc$ such that he wins any play visiting vertex u in a game without shortcuts, then no strategy $\sigma'_\bigcirc$ exists such that player $\circ$ wins any play visiting vertex u in a game with shortcuts. If such a strategy exists, then player $\circ$ wins some play visiting vertex u because either it can win any play visiting vertex $\sigma'_\bigcirc(u)$, or because $p_E((u, \sigma'_\bigcirc(u)))$ is the minimum priority occurring infinitely often, and this priority is of the same parity as player $\circ$. The former cannot be true, because then a strategy $\sigma_\bigcirc$ would exist in the game without shortcuts by using $\sigma_\bigcirc(v) = \sigma'_\bigcirc(u)$. The latter cannot be true either, because then either $p_E((u, v))$ or $p_E((v, \sigma'_\bigcirc(u))$ would be the minimum priority occurring infinitely often, and the strategy $\sigma_\bigcirc = \sigma'_\bigcirc[u := v][v := \sigma'_\bigcirc(w)]$ results in traversing those edges infinitely often and would therefore result in player $\circ$ winning the play, but no such $\sigma_\bigcirc$ exists. Therefore, such a strategy $\sigma'_\bigcirc$ cannot exist. $\square$

Both cases for adding shortcuts handled until now assumed that there was only on outgoing edge from the first vertex, but this restriction is not required. When there are multiple outgoing edges from u, every outgoing edge can be treated separately, and can have shortcuts added according to the cases above. See Figure 6.3 for an example.

Theorem 6.3. *Given a parity game with vertices u, v and w, and edges (u,v) and (v,w), a shortcut edge can be created between u and w with priority $\min(p_E((u,v)), p_E((v,w)))$ if v has only one outgoing edge, or if u and v have the same owner. Doing so does not change the winning set of any player.*

Proof. For every $v' \in W_{\bigcirc}$, player $\circ$ has a winning strategy from v'. If adding the shortcut edge does not influence the existence of a winning strategy, then the winning set of neither player will change. Since strategies are memoryless, if any play visiting vertex u is won by player $\circ$, then all plays visiting vertex u will be won by player $\circ$.

If player $\circ$ owns u and has a winning strategy $\sigma_{\bigcirc}$ from vertex u in a game without shortcuts, then player $\circ$ has a winning strategy $\sigma'_{\bigcirc}$ from u in a game with shortcuts. If $\sigma_{\bigcirc}(u)$ has only one outgoing edge, then $\sigma'_{\bigcirc} = \sigma_{\bigcirc}[u := \sigma_{\bigcirc}(\sigma_{\bigcirc}(u))]$, and using the same reasoning as for Theorem 6.1 shows that player $\circ$ still wins the play. If $\sigma_{\bigcirc}(u)$ has multiple outgoing edges, and u and $\sigma_{\bigcirc}(u)$ have the same owner, then the strategy $\sigma'_{\bigcirc}$ as defined in Theorem 6.2 still results in a winning play for player $\circ$ from vertex u. Otherwise, there is no shortcut to bypass the winning moves, and $\sigma_{\bigcirc}(u)$ is also a winning move in the game with shortcuts.

If player $\circ$ owns u but has no winning strategy $\sigma_{\bigcirc}$ from vertex u in a game without shortcuts, then no strategy $\sigma'_{\bigcirc}$ exists such that player $\circ$ wins from vertex u in a game with shortcuts. If such a strategy exists, then $\sigma'_{\bigcirc}(u)$ must have multiple outgoing edges, since Theorem 6.1 states that the winner cannot change if $\sigma'_{\bigcirc}(u)$ has only one outgoing edge. Additionally, u and $\sigma'_{\bigcirc}(u)$ must be owned by different players, because Theorem 6.2 states that no such strategy can exist if they are owned by the same player. But if u and $\sigma'_{\bigcirc}(u)$ are owned by different players, and $\sigma'_{\bigcirc}(u)$ has multiple outgoing edges, then no shortcuts are added over the edge $(u, \sigma'_{\bigcirc}(u))$ and there would be a strategy $\sigma_{\bigcirc}$ with $\sigma_{\bigcirc}(u) = \sigma'_{\bigcirc}(u)$ in the original game such that player $\circ$ wins. Therefore, such a strategy $\sigma'_{\bigcirc}$ cannot exist. □

6.3 Edge priorities in small progress measures

To be able to use edge priorities in the small progress measures algorithm, it requires some modifications. Fortunately, these modifications are very limited.

At the heart of the algorithm is a progress relation, that ensures that the values of measures assigned to vertices become larger than their successors when needed. This progress relation, $\text{prog}(\rho, v, w)$ is already defined for edges, and can be easily adapted to use edge priorities instead of vertex priorities. The progress relation $\text{prog}_E(\rho, v, w)$ uses edge priorities and is defined as follows.

Definition 6.4. $\text{prog}_E(\rho, v, w)$ is the least $m \in \mathbb{M}^{\top}_{\Diamond}$ such that

$$\begin{array}{ll} m \geq_{p_E((v,w))} \rho(w) & \text{if } p_E((v,w)) \text{ is even} \\ m >_{p_E((v,w))} \rho(w) & \text{if } p_E((v,w)) \text{ is odd} \end{array}$$

By using $\text{prog}_E(\rho, v, w)$ as the progress relation instead of $\text{prog}(\rho, v, w)$, the entire algorithm will correctly use edge priorities. This requires a small change to the function $\text{lift}_v(\rho)$, resulting in $\text{lift}'_v(\rho)$.

$$\text{lift}'_v(\rho) = \begin{cases} \rho[v := \min\left((v,w) \in E \ : \ \text{prog}_E(\rho, v, w)\right)] & \text{if } v \in V_\Diamond \\ \rho[v := \max\left((v,w) \in E \ : \ \text{prog}_E(\rho, v, w)\right)] & \text{if } v \in V_\Box \end{cases}$$

GPU implementation

Section 3.3 mentioned that the process of lifting was split up in two stages, PROG_KERNEL and LIFT_KERNEL, in such a way that it is compatible with edge priorities. The result is that the GPU version of small progress measures requires only a small change to make use of $\text{prog}_E(\rho, v, w)$ instead of $\text{prog}(\rho, v, w)$. The value of $\text{prog}(\rho, v, w)$ is stored in $\rho_E[(v, w)]$ by PROG_KERNEL for every edge (v, w), and it is sufficient to change this kernel to use the value of $\text{prog}_E(\rho, v, w)$ instead. This results in the changed kernel PROG_KERNEL_EDGE.

```
PROG__KERNEL__EDGE($e$)
Input: an edge $e = (v, w)$ with $e \in E$
Output: nothing – updates $\rho_E[e]$
(1)    $\rho_E[e] \leftarrow_{p_E(e)} \rho[w]$
(2)    if $p_E(e)$ is odd
(3)        $\rho_E[e] \leftarrow$ INCREASE($\rho_E[e], p_E(e)$)
```

By using this kernel instead of the original PROG_KERNEL, the GPU implementation will use the edge priorities stored in p_E.

The running time of PROG_KERNEL_EDGE is equal to that of the original PROG_KERNEL, but the memory efficiency has slightly increased: previously, $p(v)$ was used to check if INCREASE must be called, but now $p_E(e)$ is used instead. The value of $p(v)$ could not be read coalesced because PROG_KERNEL is launched per edge, but the value of $p_E(e)$ can be read coalesced. This reduces the number of memory transactions required to read or write $4d + d/32 + 2$ values per thread from $36d + 33$ to $36d + 2$. The memory efficiency for $d = 4$ increases from 10.24% to 12.41%.

7 Graph preprocessing

The previous chapters discussed alternating SPM and shortcuts added in parity games with edge priorities, which are techniques that should help in speeding up small progress measures. The addition of edge priorities already modified the graph in order to attain a possible speedup, but requires modified algorithms to be used. There is also a number of ways to change the parity game graph before it is being handed to the solver, and which do not require any modification to the solver.

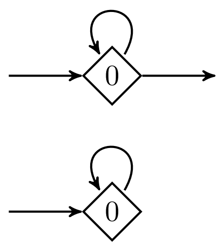
This chapter describes two of these techniques: self-loop elimination and sorting the vertices. There are more possible techniques, such as priority propagation and priority compression [5], that are not implemented in the implementation described in this report. The implemented techniques are only implemented on the CPU. When the parity game has been loaded from disk, these preprocessing steps are performed on the graph in memory. When preprocessing has finished, the graph is given to the actual CPU or GPU solver implementation to solve.

Section 7.1 describes self-loop elimination. Section 7.2 describes the various possible orderings that can be used to sort the vertices of a parity game.

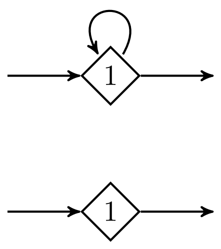
7.1 Self-loop elimination

One way of speeding up parity game solving is to reduce the size of the input graph without changing its structure. When a vertex has a self-loop, then either the self-loop or the other outgoing edges can be removed from that vertex without changing the global structure of the graph, or removing any valid choice for an optimal strategy. Figure 7.1 shows the edges that can be removed when a self-loop is present.

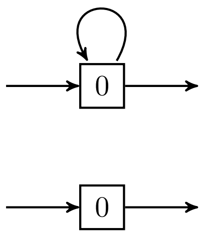
If vertex v has a self-loop, then the player owning that vertex can choose whether to always use that self-loop, or to not use it all. Using it only sometimes is useless, so either the self-loop or all the other outgoing edges can be removed.



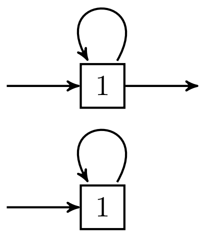
(a) Vertex with a self-loop and even priority owned by player ◇: all other outgoing edges can be removed.



(b) Vertex with a self-loop and odd priority owned by player ◇: the self-loop can be removed.



(c) Vertex with a self-loop and even priority owned by player □: the self-loop can be removed.



(d) Vertex with a self-loop and odd priority owned by player □: all other outgoing edges can be removed.

Figure 7.1: Removing outgoing edges from vertices with a self-loop.

Player ◇ wins a play if the lowest priority occurring infinitely often in the play is even, so if there is a vertex v with a self-loop and even priority owned by player ◇, he will always choose to use that self-loop. The remainder of the game will exist of only visiting vertex v, which has an even priority, so player ◇ then wins any game passing through v. On the other hand, if v has an odd priority, then using the self-loop would result in a game won by player □, and player ◇ can only win by using one of the other outgoing edges. The self-loop can therefore be removed.

A similar reasoning is valid for player □: vertices with a self-loop and owned by player □ can have their self-loop removed if their priority is even, or can have all their other outgoing edges removed if their priority is odd.

Note that a self-loop can only be removed if there are other outgoing edges. Otherwise, the self-loop is the only outgoing edge and removing it would result in a non-total graph.

7.2 Sorting vertices

In the definition of a graph, the vertices are considered an unordered set. When a graph is implemented, they are stored in an array in memory, which does have an ordering. Because the graph does not depend on the ordering of vertices, changing that ordering does not change the solution of the parity game.

Sorting the vertices might benefit in solving a parity game. In the GPU algorithm, the `lift_kernel` has a loop which is run once for every outgoing edge of a vertex. If vertices are sorted by the number of outgoing edges, then fewer threads will run useless iterations of this loop and the impact of this divergence is reduced. For the CPU algorithm, sorting the vertices will change the order in which vertices are initially added to the work queue. This could change the order in which vertices are lifted, possibly resulting in a different solving speed.

There are 5 sorting orders defined, next to the unsorted case, which loads vertices in the order that they appear in the input graph.

Reversed – The vertices are loaded in the reverse order of the input graph. Vertices loaded first in the unsorted case will be loaded last when reversed, and vice versa.

Outdegree – The vertices are sorted on the number of outgoing edges they have. Vertices with a high number of outgoing edges will be loaded first, and vertices with a low number of outgoing edges will be loaded last.

Priority – The vertices are ordered by their priority. Vertices with a high priority will be loaded before vertices with a low priority. The potential benefit of this is a reduction of divergence on the GPU: loops which copy, compare, or increase measures depend on the priority of the vertex to which that measure is assigned. If this priority differs per thread, then there can be a large number of wasted iterations for these loops.

Depth-first search – A depth-first search is performed on the input graph, and vertices are loaded in the order that they are visited by this depth-first search. This means that endpoints of edges are stored close together in memory, resulting in possible positive caching effects when retrieving information on the vertices from memory.

Breadth-first search – A breadth-first search is performed on the input graph, and vertices are loaded in the order that they are visited by this breadth-first search. This is a variation on the depth-first search ordering, that could have the effect that vertices that are lifted after each other are close to each other in memory.

8 Experiments

The GPU algorithm and other improvements discussed in this report should help in speeding up the small progress measures algorithm in practice. To validate whether the discussed techniques help, a number of experiments are conducted. The various techniques implemented in `spm` are compared to select a set of options which allows it to solve games fast. Using these options, it is compared to the existing solvers `pgsolver` and `pbespgsolve`.

Every set of options is tested on multiple parity games. A single test involves testing a single set of options on a single parity game. Every test consists of 5 runs of the program with the same options, of which the fastest and slowest are discarded and the remaining 3 times are averaged. For the random CPU strategy, a test consists of 10 runs of which the fastest and slowest are discarded and the remaining 8 times are averaged. The times are measured using the difference in wall clock time of just before the solver process is started and just after this process has terminated. Every run not finished after 15 minutes was terminated prematurely.

All experiments are conducted on a laptop running Kubuntu Linux 12.04, equipped with an Intel Core i7 2630QM CPU running at 2.00 GHz with hyperthreading enabled, and 6 GB of main memory. The GPU is an NVIDIA GeForce GT 555M, capable of running code for CUDA compute capability 2.1, with 144 CUDA cores running at 1.35 GHz and 2GB of global memory accessible via a 128-bit memory bus running at 900 Mhz.

The existing parity game solvers to which `spm` is compared are PGSolver[1] version 3.3 and the `pbespgsolve` tool of mCRL2[2] version 201310.0.

Section 8.1 describes the dataset used for the experiments. Section 8.2 compares the implemented CPU strategies. Section 8.3 compares the implemented GPU strategies. Section 8.4 shows the effects of combining the CPU and GPU implementations to run in parallel. Section 8.5 shows the influence of adding shortcuts in parity games with edge priorities. Section 8.6 compares various sorting orders for the vertices in the parity game. Section

[1]http://www.tcs.ifi.lmu.de/pgsolver
[2]http://mcrl2.org

8.7 compares spm to existing tools that solve parity games using the small progress measures algorithm. Section 8.8 compares spm on different GPUs to show the scalability. Section 8.9 contains a discussion of the results of the experiments.

8.1 Dataset

The test data used for the experiments is a subset of the dataset introduced by Keiren in chapter 5 of his PhD thesis [9]. The entire dataset is very large, and most parity games are either very simple to solve and therefore not interesting, or very hard to generate within reasonable time and memory limits – in my case, 1 hour and 4 GB per parity game. A selection of 12 games was made from the dataset, of which some properties are described in Table 8.1.

In addition to these 12 games, an instance of the Jurdzinski game is included, and a new class of games, called propagation games, is constructed in the hopes of finding a class of games that can be solved faster on a GPU then on a CPU. The Jurdzinski game[8] is a class of games for which the small progress measures algorithm always requires an exponential amount of lifts. The propagation games come in 2 variants: standard propagation

game	nr. of vertices	nr. of edges	outdegree			priorities	
			min	avg	max	max	nr.
cabp8	82 434	231 945	1	2.81	27	2	3
ctlsbc8	82 968	107 716	1	1.30	2	3	4
elev7f	861 780	1 431 610	1	1.66	8	2	3
elev7u	876 780	2 484 252	1	2.83	8	2	3
flctl7	49 830	122 701	1	2.46	194	1	2
hanoi12	531 443	1 594 321	1	3.00	3	1	2
lifti4l	134 162	372 852	1	2.78	7	4	4
nester5	62 707	82 920	1	1.32	2	981	16
pdlbc8	228 787	457 571	1	2.00	1 871	1	2
swp8_1i	147 458	406 665	1	2.76	25	2	3
swp4_2r	869 569	3 200 129	1	3.68	13	2	2
abpswp4_1	827 137	2 550 529	1	3.08	18	2	2
jurdz5_10	170	429	1	2.52	7	11	12
prop50_1k	50 003	100 002	1	2.00	50	4	5
tprop21	4 194 305	8 388 607	1	2.00	3	2	3

Table 8.1: Vertex count, edge count, minimum/average/maximum outdegree, maximum priority, and number of distinct priorities for all parity games in the dataset used for experiments.

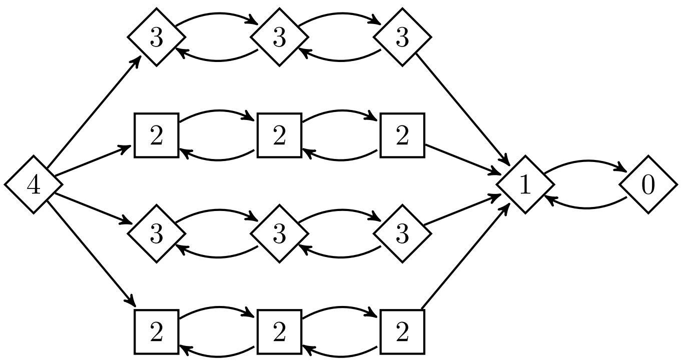


Figure 8.1: A standard propagation game with 4 paths of 3 nodes each.

games and propagation trees. The main idea behind them is that most vertices need to be lifted often, regardless of the employed strategy. While a GPU can can perform lifts faster than a CPU can, it cannot employ some of the more efficient strategies that a CPU can use. Because most vertices need to be lifted often, the advantage of an efficient strategy should disappear and the GPU should benefit.

A standard propagation game contains a number of bi-directionally connected paths between a source and a pair of target nodes. See Figure 8.1 for a standard propagation game with 4 paths of 3 nodes each. The owners of the paths are alternated, such that both players have a non-trivial game to solve. The priority of a path owned by player $\diamond$ is odd and vice versa, such that no winning loop is possible inside of a path for the player owning that path. The target nodes form an infinite 2-step loop that is won by player even. The propagation game in the dataset, `prop50_1k`, contains 50 paths of 1 000 nodes each.

A propagation tree consists of a tree with a certain number of levels and a fixed number of children for every tree node. See Figure 8.2 for a propagation tree with 3 levels and a fanout of 2. All connections in the tree are bidirectional, apart from an edge from one of the leafs of the tree to a pair of target nodes. These target nodes again together form an infinite loop won by player even. The priority of the root is high and even, and the priorities of all tree nodes are odd, such that player $\diamond$ cannot win unless it uses the edge to the target nodes. The propagation tree in the dataset, `tprop21`, contains 22 levels and has a fanout of 2.

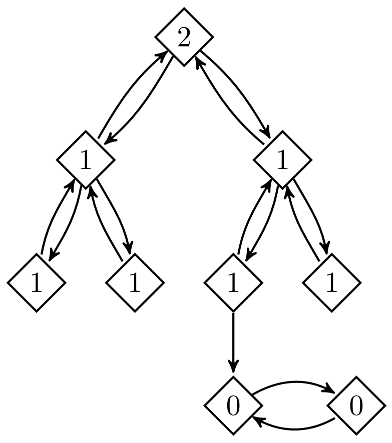


Figure 8.2: A propagation tree with 3 levels and a fanout of 2.

8.2 CPU Strategies

The single-occurrence variants of the queue, stack and random CPU strategies are compared. See Figure 8.3 for the running times of all test cases using these strategies.

The random strategy performs surprisingly well: in most test cases, its running times are only slightly slower than for the queue or random strategy. The most notable exception is `flctl7`, where the random strategy is 5 times faster than the other strategies. This is the only test case where the random strategy is the fastest strategy by far; in all other cases, the deterministic queue and stack strategies are usually faster, or are at least not much slower than the random strategy.

Overall, the stack strategy appears to perform slightly better than the queue strategy. Both strategies are faster in roughly half of the input files, but the difference in running times when the stack is slower are smaller on average. No clear best strategy can be given just yet, but enabling alternating SPM might result in a more clear difference.

The queue and stack strategies are compared with alternating SPM enabled and disabled. See Figure 8.4 for the running times of all test cases using these strategies.

When looking at both strategies with alternating SPM enabled, it is now possible to point to a better strategy: the queue strategy is almost always faster than the stack strategy, and in the test cases where it is slower the difference is not very big. There are a few cases, such as `nester5` and `abpswp4_1`, where the stack strategy is slower than the queue strategy by a large margin.

Enabling alternating SPM with the queue strategy provides a very no-

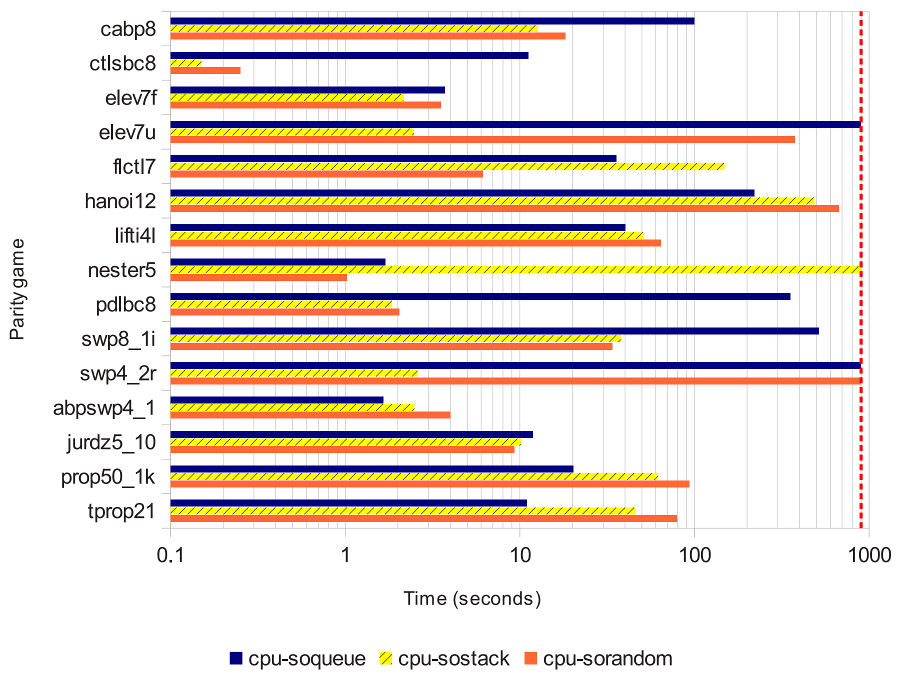


Figure 8.3: Running times for the single-occurrence variants of the queue (`cpu-soqueue`*), stack (*`cpu-sostack`*) and random (*`cpu-sorandom`*) strategies on the CPU, using a logarithmic time scale. The dashed line at 900 seconds is used to indicate that tests were terminated after 15 minutes.*

ticeable speedup. The biggest differences are present for the test cases `flctl7` and `jurdz5_10`, where alternating SPM performs more than 100 times better than either strategy with alternating SPM disabled. Even when enabling alternating SPM is slower than disabling it, the difference is almost always small. This is not unexpected: if alternating SPM does not reduce the number of lifts required, it requires more time because it performs the lifts for both players and because it periodically attempts to synchronize the winning set of both players, but the impact of this is not very large.

Figure 8.5 provides a different view on the same data as already included in Figure 8.4, but by using scatter plots the trends are more clear. Both scatter plots contain a diagonal line: points on this line represent test cases that are solved equally fast by both the queue and the stack strategy; points above this line are solved faster using the queue strategy, while points below the line are solved faster using the stack strategy.

In Figure 8.5a, alternating SPM is disabled. The points are evenly distributed on both sides of the line, but those below the line are on average farther away from the line. This coincides with the earlier observation that

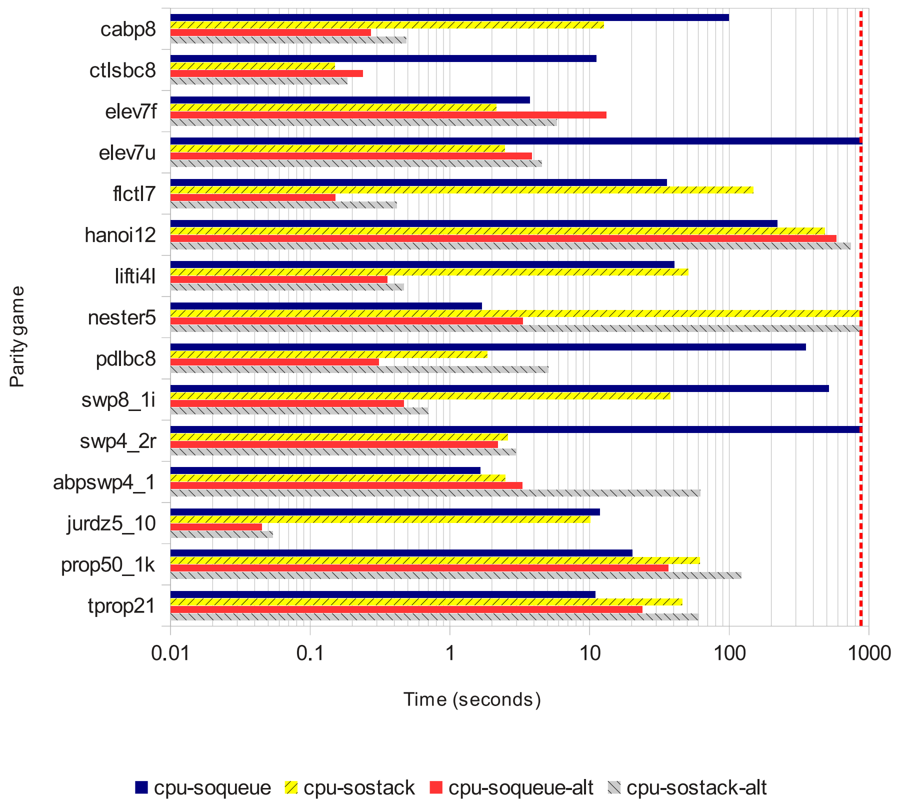
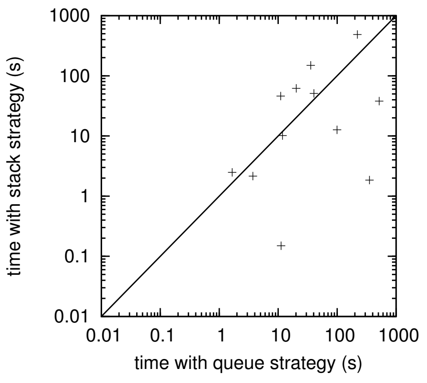


Figure 8.4: Running times for the single-occurrence variants of the queue (`cpu-soqueue`*) and stack (*`cpu-sostack`*) strategies on the CPU with alternating SPM disabled, and the queue (*`cpu-soqueue-alt`*) and stack (*`cpu-sostack-alt`*) CPU strategiews with alternating SPM enabled, using a logarithmic time scale.*

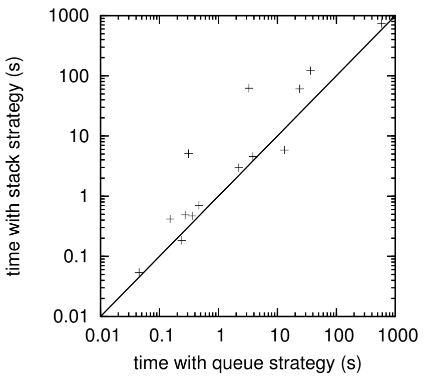
the stack strategy works slightly better when alternating SPM is disabled.

Figure 8.5b shows the effect of enabling alternating SPM: it is immediately obvious that, on average, the points have moved to the lower left corner, and are thus solved faster in general. It is also clear that most points are now above the line, again showing that the queue strategy works better when alternating SPM is enabled.

Considering these results, the best CPU strategy is the queue strategy with alternating SPM enabled.



(a) Alternating SPM disabled



(b) Alternating SPM enabled

Figure 8.5: A scatter plot comparing the running times for every test case when using the queue strategy and when using the stack strategy, with alternating SPM disabled and enabled, using a logarithmic time scale.

8.3 GPU Strategies

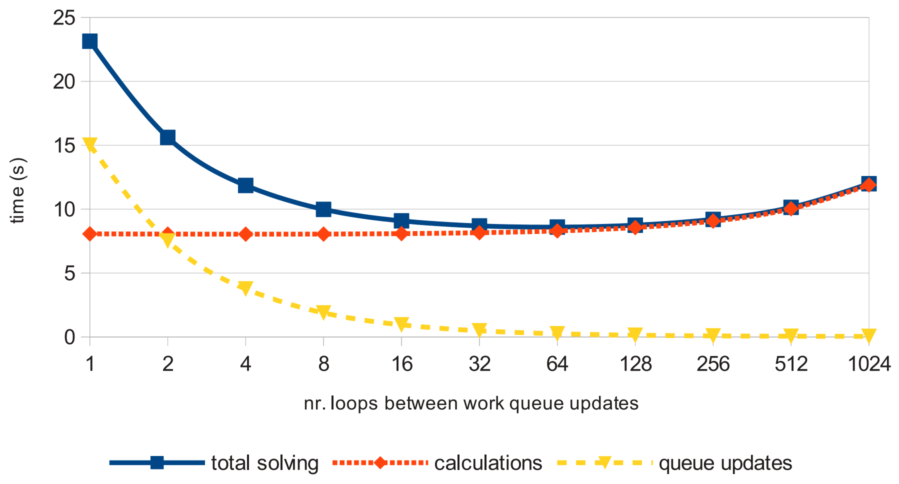
Before comparing the GPU strategies, there is one parameter of the explicit queuing strategy to give a value: the number of lifting steps to perform between two subsequent updates of the work queue. Figure 8.6 shows the impact on solving time when varying the number of loop iterations between work queue updates, as well as the time spent performing actual lifting calculations and time spent updating the work queue.

Figure 8.6a shows the expected shapes: a low number of loops between work queue updates results in a large amount of time spent on updating the work queue, while a high number of loops between work queue updates results in increased time spent on the actual lifting calculations. The optimal value is between 32 and 128 loops, since the total solving time is at the lowest point between those numbers.

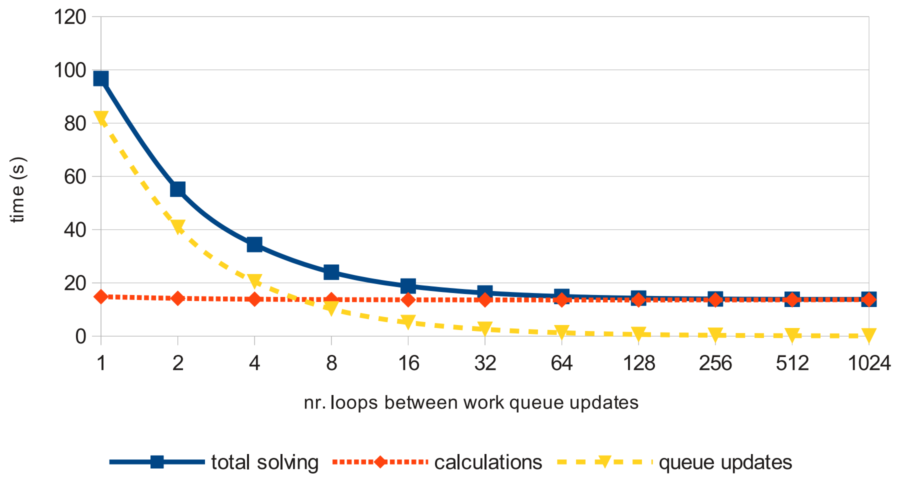
Figure 8.6b shows the graph for a different test case, which shows a pattern that occurs for multiple test cases: the value of the parameter does not make a big difference for the calculation times, so choosing a higher value results in lower overall running times. The time used to update the work queue is marginal when the number of loops is set to at least 32. Given both graphs, the default number of loops should be between 32 and 128. A default of 32 is chosen since that coincides with the number of threads in a warp.

The implicit queuing and explicit queuing GPU strategies are compared with alternating SPM disabled and enabled. See Figure 8.7 for the running times of all test cases using these strategies.

The first impression is that the implicit queuing strategy with alternating



(a) Time spent of actual lifting calculations and work queue updates, and resulting total solving time for the `hanoi12` *test case.*



(b) Time spent of actual lifting calculations and work queue updates, and resulting total solving time for the `ctlsbc8` *test case.*

Figure 8.6: Time spent on actual lifting calculations and work queue updates for two inputs using the explicit queuing GPU strategy, with varying number of lifting steps between two work queue updates.

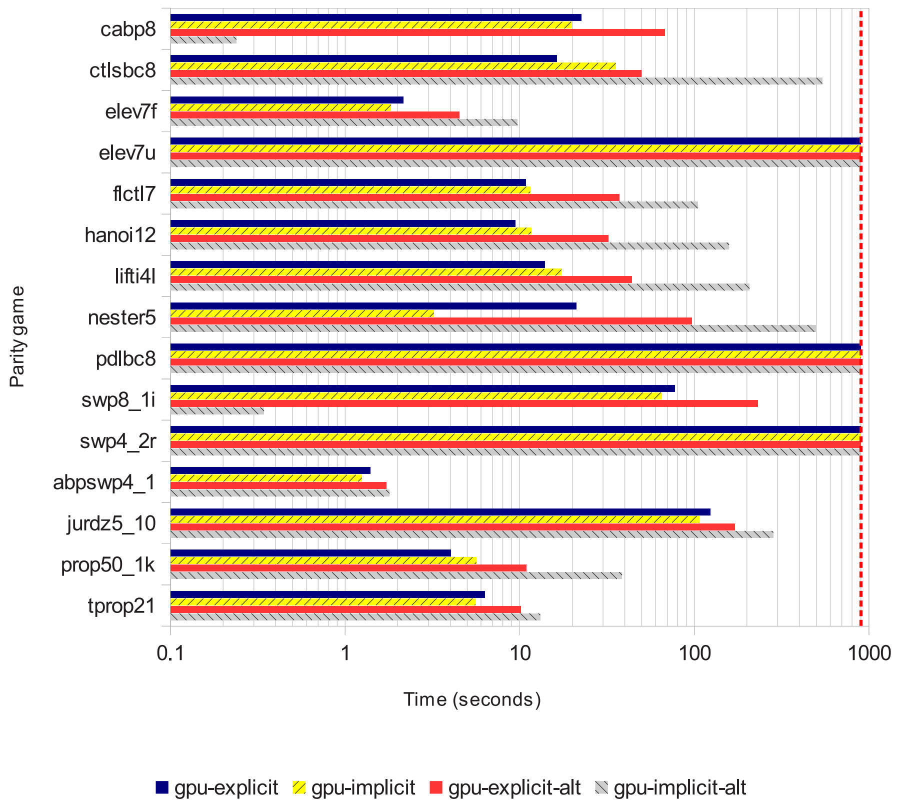
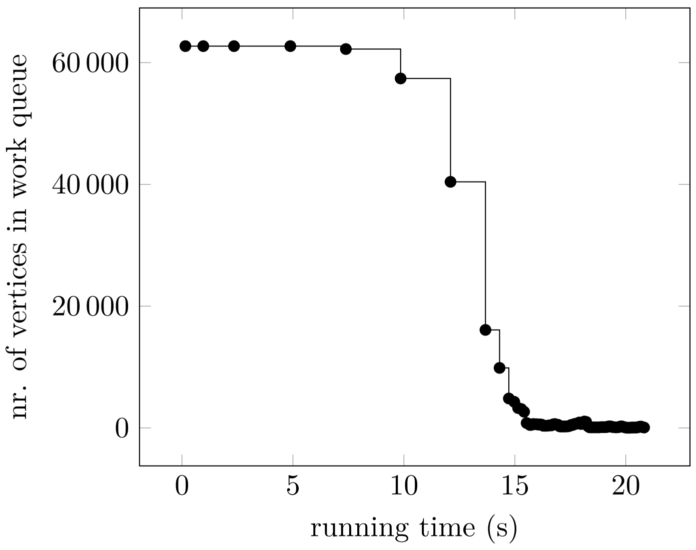


Figure 8.7: Running times for the for the explicit queuing (`gpu-explicit`*) and implicit queuing (*`gpu-implicit`*) GPU strategies with alternating SPM disabled, and the explicit queuing (*`gpu-explicit-alt`*) and implicit queuing (*`gpu-implicit-alt`*) GPU strategies with alternating SPM enabled, using a logarithmic time scale.*

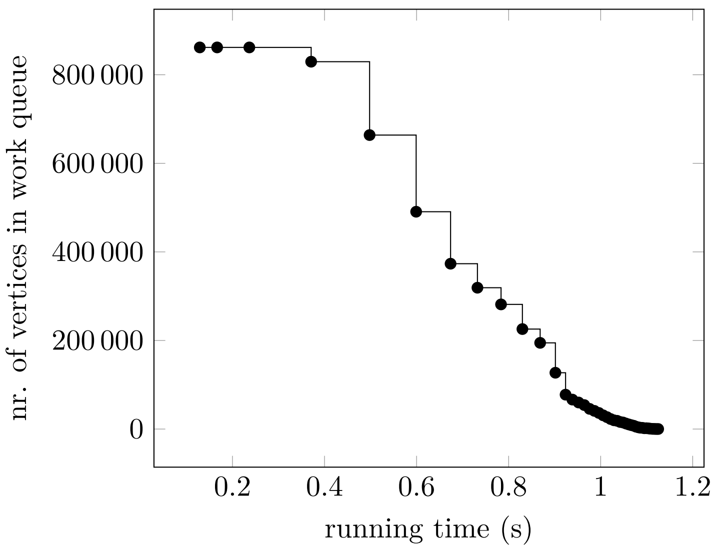
SPM enabled is the worst option of the four GPU configurations tested. There are however two test cases where this configuration is much faster than the other three: `cabp8` and `swp8_1i`. When comparing this to the CPU implementation using the queue strategy and alternating SPM, these times are not as spectacular: the CPU implementation solves these test cases only slightly slower than the GPU implementation does.

The explicit queuing strategy with alternating SPM enabled is slower than disabling alternating SPM for all test cases. While alternating SPM resulted in much better running times for the CPU algorithm, this does not appear to be the case for the GPU algorithm. The best results using the GPU algorithm are obtained when alternating SPM is disabled.

When alternating SPM is disabled, the difference between the explicit queuing and implicit queuing strategy is not very big. For most test cases, the running times are within a factor 2 of each other, with `nester5` as a notable exception: the implicit queuing strategy is about 7 times faster for this test case than explicit queuing. For this test case, the explicit queuing strategy could be slower due to the added overhead of work queue calculations.



(a) Work queue length over time for the `nester5` *test case.*



(b) Work queue length over time for the `elev7f` *test case.*

Figure 8.8: The length of the work queue used in the explicit queuing GPU strategy over time for two test cases. The marks indicate the moments when the work queue is updated.

Figure 8.8 shows the work queue length over time for the `nester5` and `elev7f` test cases. The overall shapes are similar: a few iterations at the start with nearly all vertices in the work queue, and then the size of the work queue shrinks until the last iterations work with a nearly empty work queue. The number of work queue updates is similar in both cases, but `nester5` has less than 10% of the number of vertices that `elev7f` has, and the running time is nearly 20 times higher. This shows that a single iteration takes much longer with `nester5`, which is caused by the high maximum priority in this case, 981, which results in measures with a large number of elements. The overhead of work queue calculations is very small for `nester5`, so the difference between the explicit and implicit GPU strategy is caused by other effects. It could be that the number of queued vertices per warp is low for implicit queuing, which would result in less possible divergence in execution. The impact of this is large because the number of elements per measure is high.

In general there is not much difference between the explicit and implicit queuing strategies, but if any strategy has to be named as the more efficient strategy, then the implicit queuing strategy wins because of its superior performance on `nester5`.

The kernels presented in Chapter 3 were theoretically analyzed for their memory efficiency. The NVIDIA Visual Profiler (`nvvp`), part of the CUDA SDK, can analyze the memory efficiency attained in practice per single kernel invocation. The memory efficiency is measured on the `nester5` input using the implicit queuing strategy, by looking at the memory efficiency reported by `nvvp` for a single kernel invocation at approximately halfway the entire calculation.

With $d = 982$ and maximum outdegree $O = 2$, the theoretical memory efficiency of IMPLICIT_PROG_KERNEL is $11,18\%$. The achieved memory efficiency as measured by `nvvp` is $14,74\%$. This is slightly higher than theoretically calculated, but not by much. The small difference can occur because some memory accesses counted as completely non-coalesced might be slightly coalesced, because a few threads could use multiple values from a single memory transaction where it was assumed that all threads required a separate memory transaction.

With $d = 982$ and maximum outdegree $O = 2$, the theoretical memory efficiency of IMPLICIT_LIFT_KERNEL is 6.06%. The achieved memory efficiency as measure by `nvvp` is 9.08%. As for IMPLCIT_PROG_KERNEL, this is slightly higher than the theoretical efficiency.

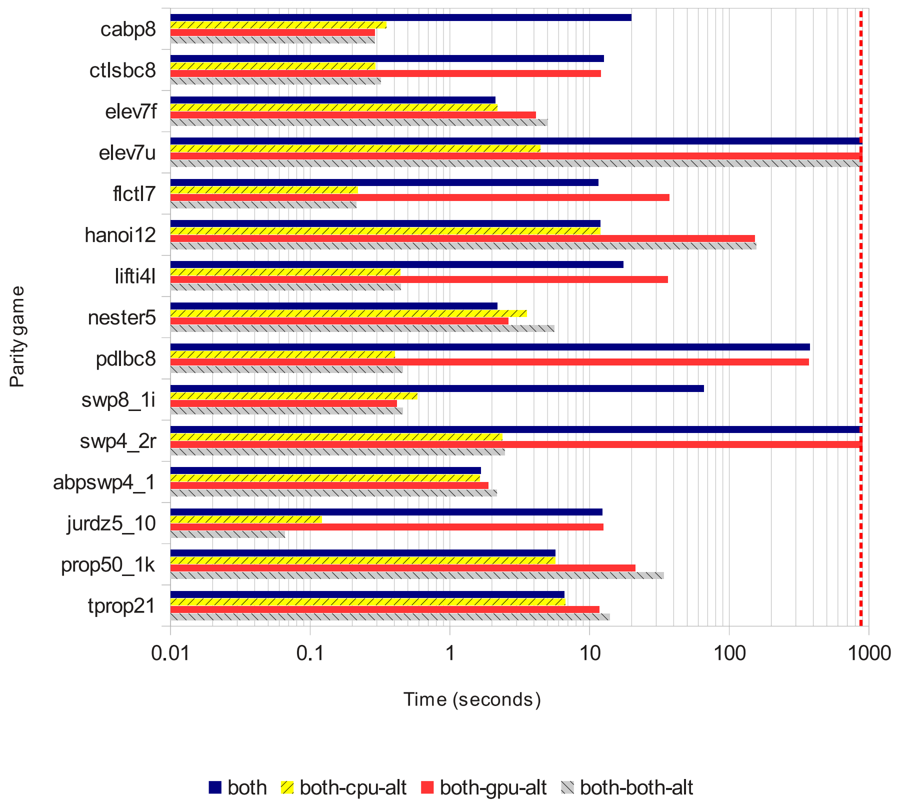


Figure 8.9: Running times for both the CPU and GPU implementation running in parallel, with alternating SPM disabled for both (`both`*), enabled for only the CPU (*`both-cpu-alt`*), enabled for only the GPU (*`both-gpu-alt`*), and enabled for both CPU and GPU (*`both-both-alt`*), using a logarithmic time scale.*

8.4 Combining CPU and GPU

When running both implementations in parallel as described in Section 4.3, the CPU implementation uses the queue strategy and the GPU implementation uses the implicit queuing strategy. Alternating SPM can be enabled independently for both implementations. See Figure 8.9 for the running times of both implementations running at once, with all possible configurations of enabling alternating SPM.

It is evident that alternating SPM must be enabled for the CPU implementation. The running times for the two configurations where it is disabled for the CPU implementation are nearly always the slowest of the running times.

The remaining question is whether alternating SPM must be enabled for

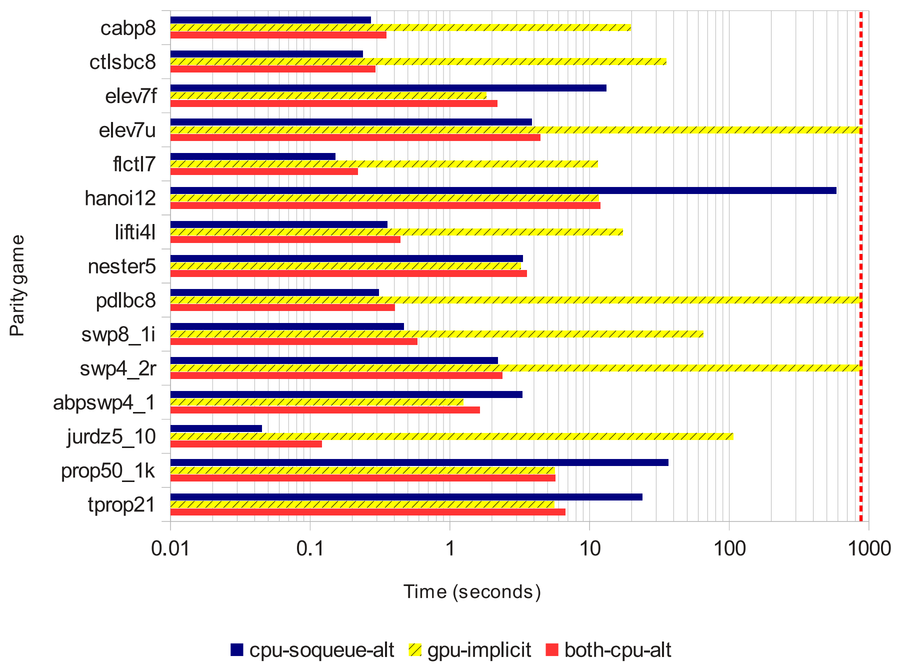


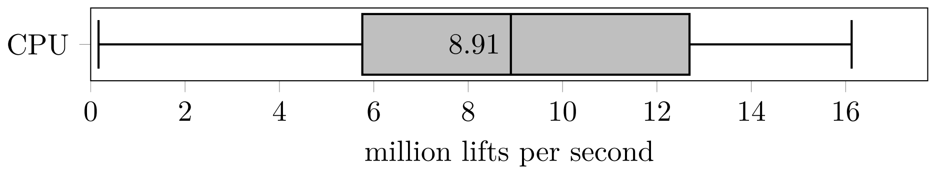
Figure 8.10: Running times for the CPU implementation using the queue strategy with alternating SPM enabled (`cpu-soqueue-alt`*), the GPU implementation using the implicit queuing strategy with alternating SPM disabled (*`gpu-implicit`*), and both implementations with those strategies running in parallel (*`both-cpu-alt`*), using a logarithmic time scale.*

the GPU as well. When it is disabled, the running times are never much slower than when it is enabled. Enabling it, however, might result in a performance loss; see for example the test cases `elev7u` and `hanoi12`. The best configuration is to enable alternating SPM only for the CPU, and not for the GPU.

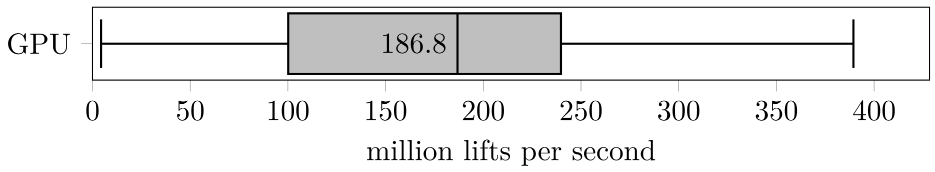
Combining both implementations to run at once should provide the best of both worlds: both implementations have only a small amount of extra overhead due to synchronizing their measures, and there might even be a benefit due to this synchronization. See Figure 8.10 for the running times of the combined implementation compared to the running times of the separate implementations.

For every test case, combining both implementations is slightly slower than the fastest of both implementations, which shows that combining both implementations indeed gives the best of both worlds.

This is also a good point to compare the speed of the CPU and GPU implementations. The CPU implementation is still the fastest implemen-



(a) Number of lifts performed per second by the CPU implementation using the queue strategy with alternating SPM disabled, in millions of lift per second.



(b) Number of lifts performed per second by the GPU implementation using the implicit queuing strategy with alternating SPM disabled, in millions of lift per second.

Figure 8.11: Lifting speeds of CPU and GPU in millions of lifts per second

tation for most of the test cases, but there are a few test cases where the GPU implementation is significantly faster than the CPU implementation. For example, `hanoi12` is solved approximately 50 times faster using the GPU implementation. The GPU implementation complements the CPU implementation, and combining both of them to run in parallel results in an implementation which provides decent solving times for all test cases.

The GPU has a large advantage in the number of lifts it can perform per second. See Figure 8.11 for a comparison of the number of lifts performed per second by the CPU implementation and the GPU implementation. This is measured by calculating the number of lifts performed for each test case, and dividing that by the time spent on solving the game. On average, the GPU performs over 20 times more lifts per second than the CPU.

Both propagation games can also be solved faster using the GPU implementation: the GPU is over 5 times faster for `prop50_1k` and roughly 4 times faster for `tprop21`. This class of games can indeed be solved faster on a GPU than when using a CPU, as was intended.

8.5 Edge priority shortcuts

The use of shortcuts added in parity games with edge priorities is examined. See Figure 8.12 for a comparison of the combined implementation which adds no shortcuts to an implementation which adds 1 or 2 levels of shortcuts.

It is obvious that adding 2 steps of shortcuts is inefficient: it causes timeouts to occur for 3 test cases, and it is never faster than adding only

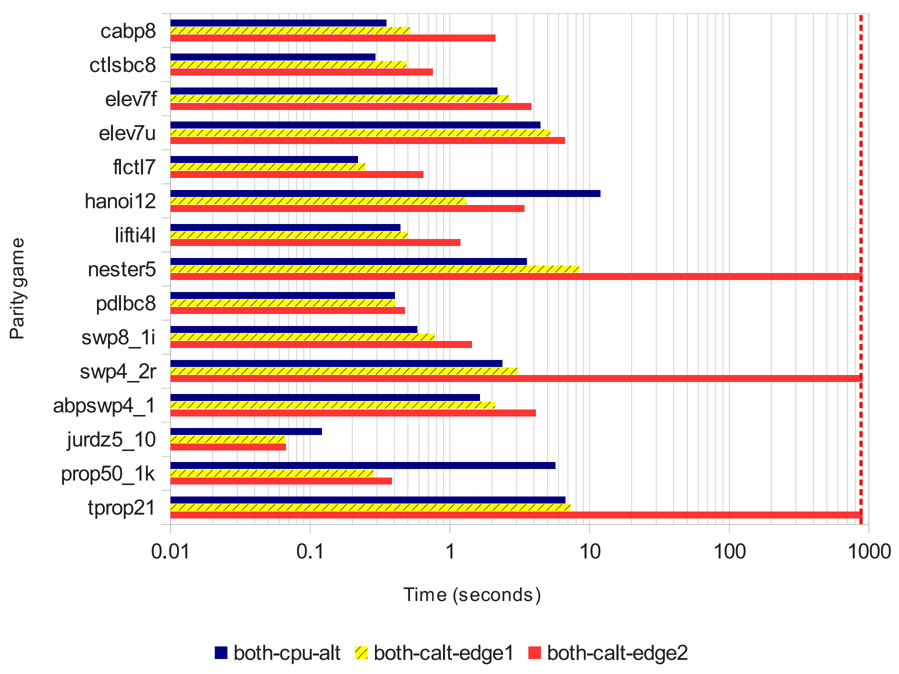


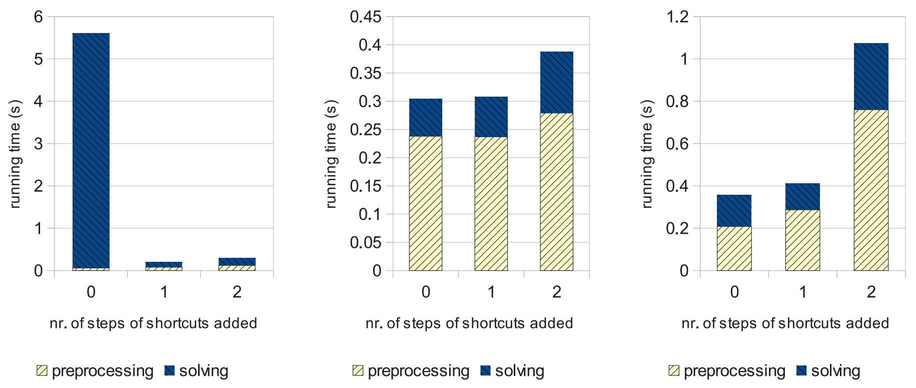
Figure 8.12: Running times for both CPU and GPU implementations running in parallel, with no added shortcuts (`both-cpu-alt`*), one step of shortcuts added (*`both-calt-edge1`*), and two steps of shortcuts added (*`both-calt-edge2`*), using a logarithmic time scale.*

1 step of shortcuts. Adding 1 step of shortcuts does help in some cases, especially for `hanoi12` and `prop50_1k`. It is almost never a big burden either: the worst runtime increase is a factor 3 for `nester5`. Enabling one step of shortcuts therefore seems to be a good choice overall.

Figure 8.13 shows the influence of adding shortcuts on the preprocessing and solving times of `prop50_1k`, `pdlbc8`, and `lifti4l`. The preprocessing time includes reading the graph from disk, and all graph preprocessing such as removing self-loops and adding shortcuts. Note that the running times given throughout this chapter involve both the preprocessing and the solving time.

The solving times of `prop50_1k` greatly benefit from adding a single step of shortcuts. This is related to the bi-directional edges used in most of the input graph: `hanoi12` also features a large number of bi-directional edges, and shows a similar increase in running time when adding shortcuts. Adding a second step increases the preprocessing time without speeding up the solving times.

Neither `pdlbc8` nor `lifti4l` benefits from adding shortcuts. For `pdlbc8`,



(a) `prop50_1k` *(b)* `pdlbc8` *(c)* `lifti4l`

Figure 8.13: The preprocessing and solving times for 3 test cases, when adding no, 1, and 2 steps of shortcuts.

the preprocessing time increases without any benefit to the solving time. For `lifti4l`, adding a single step of shortcuts results in a decrease in solving time, but the increase in preprocessing time is greater.

8.6 Sorting input graph

A number of different orderings for the vertices in the graph were presented in Section 7.2. Figure 8.14 shows the running time of the combined implementation when vertices are sorted according to each of these orderings.

What is immediately obvious, is that sorting using the BFS-order or DFS-order is a very bad idea: the running time increases by more than a factor 10 in most cases.

The other vertex orderings behave roughly the same: in all cases except for `jurd5_10`, they do not give any noticeable improvement. The original vertex order is the fastest ordering in the majority of the tests. The difference for `jurdz5_10` is the biggest difference, but still it is only a factor 2 and in absolute terms the difference in 60 ms.

Overall, sorting the vertices does not help in solving parity games faster. The possible effects on cache efficiency do not materialize, and when using the BFS or DFS orderings the results get a lot worse than when using the original ordering.

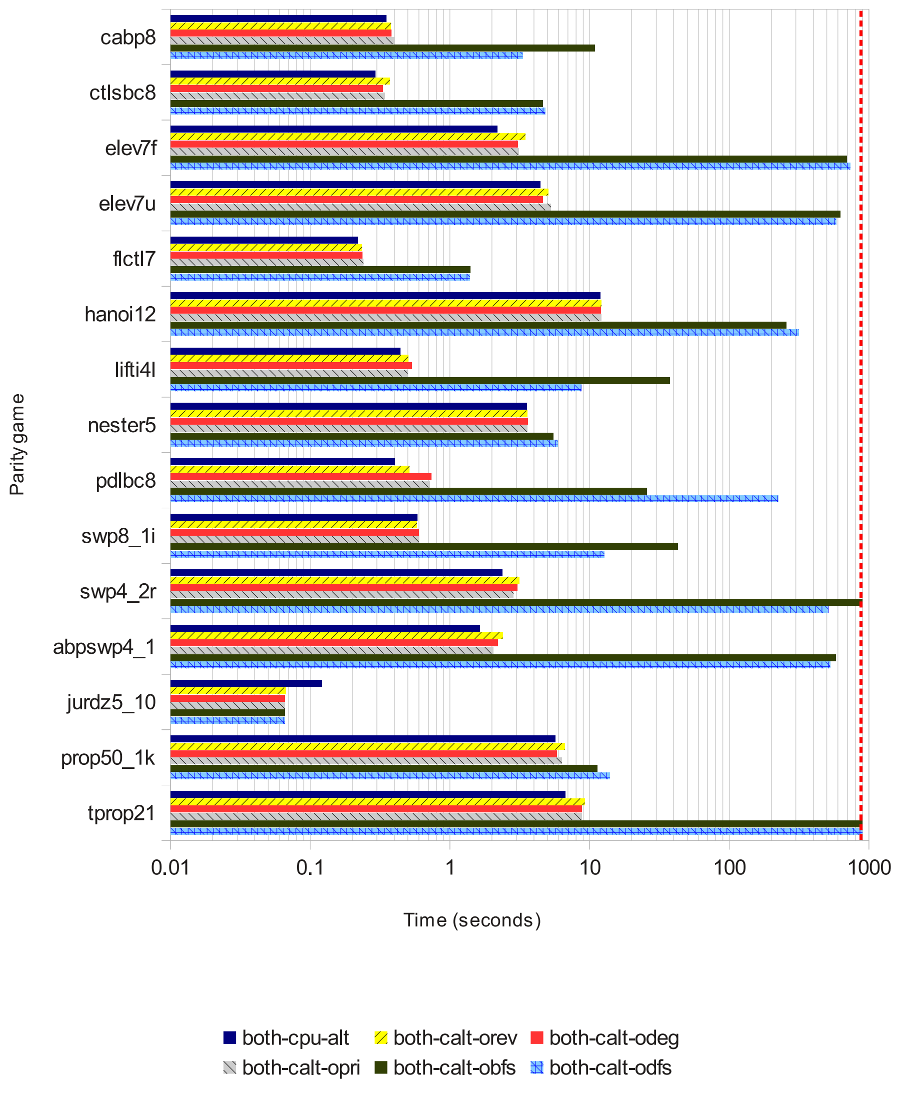


Figure 8.14: Running times for the combined implementation when the vertices are sorted in original order (`both-cpu-alt`*), in reverse order (*`both-calt-orev`*), by degree (*`both-calt-odeg`*), by priority (*`both-calt-opri`*), by BFS-order (*`both-calt-obfs`*) and by DFS-order (*`both-calt-odfs`*), using a logarithmic time scale.*

8.7 Existing tools

The best configuration for `spm` – combining the queue CPU strategy with the implicit queuing GPU strategy, enabling alternating SPM for the CPU only, and adding 1 step of shortcuts – is compared to the best times achievable using `pgsolver`, using the small progress measures algorithm, and `pbespgsolve`, for which alternating SPM is enabled, and SCC detection and cycle detection are disabled. See Figure 8.15 for the running times of these three tools.

For most cases, the three tools show comparable performance. It is obvious that, overall, `pgsolver` is the slowest of the three: its running times are usually at least twice as high as for the other solvers, but not much more than that. The other two tools are very close to each other in terms of performance, but `pbespgsolve` is slightly faster than `spm` for most cases. There are, however, a few cases where `spm` is clearly faster than the competition.

The cases where `spm` is clearly the fastest implementation are `elev7f`, `hanoi12` and both propagation games. In Figure 8.10, where the CPU and

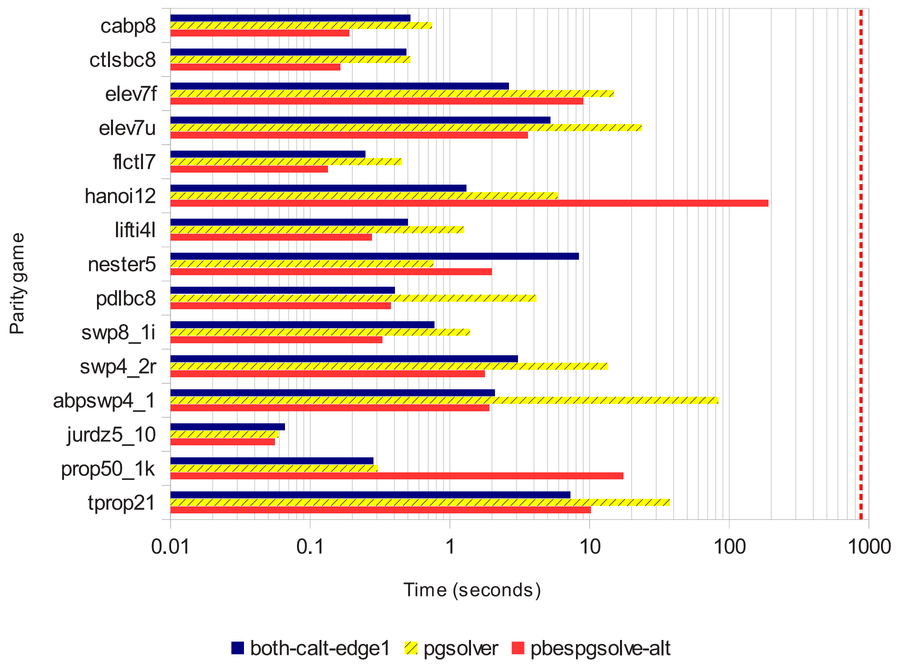


Figure 8.15: Running times for `spm` *(*`both-calt-edge1`*),* `pgsolver` *and* `pbespgsolve`*, using a logarithmic time scale.*

GPU implementations are compared, the GPU implementation shows a clear improvement over the CPU implementation on these same games. Adding the GPU implementation to the solver therefore helps to solve a number of games significantly faster than what was possible with previously existing solvers.

8.8 Scalability

The GPU implementation should show a speedup when faster hardware is used. Some tests are run on a system having a faster GPU: a NVIDIA GeForce GTX 660 Ti, capable of running code for CUDA compute capability 3.0, with 1344 CUDA cores running at 1.14 GHz and 2 GB of global memory accessible via 192-bit memory bus running at 3.00 GHz. When not considering the possible effects of the different architecture, the computational power of the GTX 660 Ti is over 7.5 times that of the GT 555M, while the available memory bandwidth is increased fivefold. The system containing the faster GPU is also equipped with a faster CPU. Figure 8.16 shows the running times for the GPU implementation and combined implementation on both GPUs.

When considering the running times of the GPU implementation, the running times are 2 to nearly 5 times lower on the faster GPU than on the slower GPU; on average, the running times are nearly 3 times lower. The average speedup is only half that of the maximal attainable speedup, but some test cases – `cabp8`, `ctlsbc8` and `lifti4l` – come quite close to achieving that.

There are no test cases which were clearly solved faster by the CPU implementation on the slower GPU, that are now clearly solved faster by the GPU. There is one case that was solved equally fast by both, `nester5`, that is now solved clearly faster by the GPU. The differences between the GPU implementation and the CPU implementation have become smaller though.

In general, it is clear that the GPU implementation scales reasonably well to faster hardware.

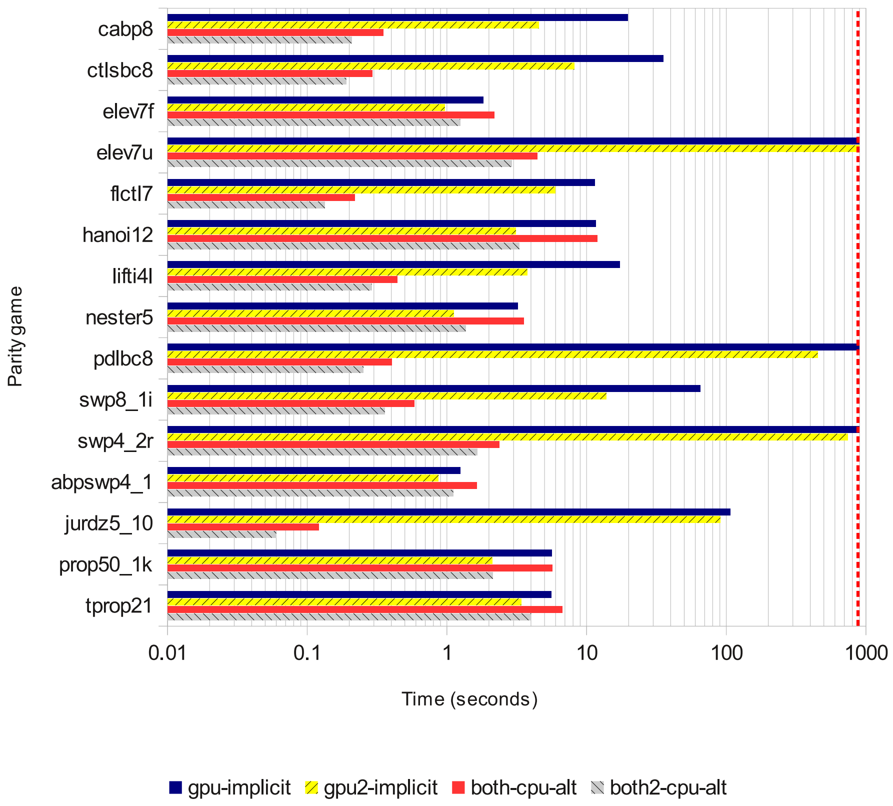


Figure 8.16: Running times for the implicit queuing GPU strategy on a GT 555M (`gpu-implicit`*) and a GTX 660 Ti GPU (*`gpu2-implicit`*), and the combined implementation on a GT 555M (*`both-cpu-alt`*) and a GTX 660 Ti GPU (*`both2-cpu-alt`*), using a logarithmic time scale.*

8.9 Discussion

Of the implemented CPU strategies, the single-occurrence queue strategy with alternating SPM enabled performs best. Both GPU strategies show similar performance, but enabling alternating SPM for them resulted in worse running times for most test cases. On first sight, the single-occurrence queue on the CPU should behave roughly the same as the queuing strategies on the GPU, since both implementations lift all relevant vertices before lifting a vertex again. There are a few difference in the process of lifting that can explain the observed differences in behavior. The most important difference is that the GPU effectively lifts of all vertices at once, while the CPU lifts them one-by-one. If the measure of the first vertex influences the measures of later vertices, then this can be used in a single iteration over all vertices on the CPU, while it requires multiple iterations over all vertices on the GPU. Another difference is that the explicit queuing strategy does not rebuild its work queue after every iteration, which means that some vertices might be in the queue for a few useless iterations, while others are added to the queue a number of iterations after they could have been.

Overall, the GPU implementation can perform a much larger number of lifts per second than the CPU implementation can, but the CPU can easily use better strategies to reduce the number of required lifts. The GPU implementation therefore has an advantage in games where a large number of lifts are required to solve the game, regardless of the employed strategy. This is also the reason why the propagation games are solved faster by the GPU: nearly all paths require a large number of lifting steps to stabilize the measures, regardless of the strategy employed.

Combining both implementations to run in parallel gives the best of both worlds: the running time of the combined implementation is only slightly slower than the fastest running time achieved by the separate implementations. No cross-over effects were seen where combining both implementations resulted in a faster solving time than both could achieve independently. This suggests that sharing the results between both implementations does not help in solving the parity games faster.

Enabling edge priorities and adding a single step of shortcuts can significantly reduce the solving time for some parity games, while it has no effect or a slightly negative effect on the solving time for most games. There is often a reduction in solving time, but the increase in preprocessing time negates small wins in solving time. Adding shortcuts results in a larger number of edges in the game, while only a few edges can be removed. This is a possible cause of the increase in solving time when a second step of shortcuts is added.

Comparing the implementation `spm` to the existing solvers `pgsolver` and `pbespgsolve` shows that `spm` has comparable performance to those solvers, and can even solve a number of games significantly faster because of the use of the GPU. Using the GPU for parity game solving can have a significant positive impact on the speed of solving some parity games.

9 Related work

There is currently a very active research field for model checking, part of which is concerned with solving the model checking problem of validating whether a given model satisfies a certain property. This chapter gives a short overview of work related to the research presented in this report.

The related work can be broadly categorized in 2 categories: faster solving of parity games using multi-core architectures, such as quadcore processors, and solving some form of the model checking problem using many-core architectures, such as a GPU.

Van de Pol and Weber [16] presented an adaptation of the small progress measures algorithm to run on multi-core systems. The workload is spread over multiple cores by dividing the parity game into a number of components equal to the number of available cores, and assigning every component to a separate core. Every core then performs all lifts on vertices in its component. This method provides faster parity game solver when multiple cores are employed, but the resulting speedup is not linear, and the effects are small when more than 4 cores are used.

Huth et al. [7] improved on the work by van de Pol and Weber. They reduced the critical sections of the original implementation, resulting in better scaling to 8 cores with a more linear speedup. They also redesigned the way work is distributed between cores, by using a parallel queue containing all vertices that may need to have their measure updated, and letting every core take the next vertex from the queue when they need to. This spreads out the load more evenly over the cores, and allows the algorithm to scale better up to 16 cores, at the expense of worse performance when employing only few cores.

Van der Berg [17] adapted the small progress measures algorithm to run on the multi-core Cell architecture of the Playstation 3 using a similar approach as Van de Pol and Weber. The graph is divided into a number of clusters of which all lifts are handled by the same core. The resulting implementation shows comparable results to that of Van de Pol and Weber: using more cores results in faster solving, but the speedup is not linear.

The general theme in scaling small progress measures to multi-core system is to divide the graph in clusters, but Huth et al. have shown that using a parallel queue provides better scaling when more cores are present. The GPU implementation of `spm` can use 2 different strategies based on queuing.

As for model checking using a GPU, Bonsacki et al. [2] attained a significant speedup in solving probabilistic model checking. The central part of probabilistic model checking consists of sparse matrix-vector multiplications, which they calculated on the GPU instead of the CPU. The result is that solving these games is up to 18 times faster when employing a GPU then when only using a CPU.

Barnat et al. [1] redesigned the maximal accepting predecessors algorithm for LTL model checking in terms of matrix-vector multiplication, and used the GPU to calculate these multiplications. There was still some CPU code involved to steer the GPU calculations, and those dominate the resulting running time. Nevertheless, using the GPU resulted in noticeably lower computation times.

Hoffmann and Luttenberger [6] recently investigated implementing three parity game solving algorithms on the GPU: the recursive algorithm by Zielonka, strategy iteration, and the small progress measures algorithm. The paper is coarse on details of how the algorithms were implemented, but hints that values assigned to vertices are continuously recalculated until they stabilize. Their results show that strategy iterations and, even more so, the recursive algorithm benefit greatly from being run on the GPU: the recursive algorithm is up to 20 times faster on a GPU then on a CPU. The small progress measures algorithm, however, is slower on the GPU in all of their test cases. The results obtained with `spm` show that there are some cases where employing the GPU is beneficial, but these cases might not be included in Hoffmann and Luttenbergers dataset. Another possible explanation for the difference is that `spm` can use slightly more elaborate strategies than were implemented by Hoffmann and Luttenberger.

10 Conclusion

The small progress measures algorithm is one of the algorithms used to solve parity games. It is actually a class of algorithms, of which the practical speed is determined by the lifting strategy used in the algorithm. Existing implementations of small progress measures are mostly tailored for the CPU.

The small progress measures algorithm has been adapted to be suitable for use on GPUs, and two basic strategies suitable for use on a GPU have been introduced, implicit queuing and explicit queuing. An implementation of this algorithm, `spm`, was created to be able to test the performance impact of using a GPU. This implementation also contains an implementation of the CPU algorithm, and a combination of both implementations running in parallel in order to get the best results of both worlds.

Two techniques are introduced which could speed up the implementation: alternating SPM and shortcuts in games with edge priorities. When alternating SPM is enabled, the parity game is solved for both players at the same time, while periodically synchronizing the partial winning set of both players. It has already been used in the `pgsolver` and `pbespgsolve` tools, but no formal description of how and why this works was available. When priorities are assigned to edges instead of vertices, it is possible to create shortcut edges which skip one or more vertices, without ignoring their priorities in the computation. This preprocessing step is done on parity games, but requires modified algorithms to solve the parity games. The required modifications to the small progress measures algorithm have been described.

Two generic parity game preprocessing techniques have been described: self-loop elimination and sorting vertices. By applying self-loop elimination, some outgoing edges of vertices containing self-loops can be removed, resulting in a smaller graph without changing the winner of any vertex. By sorting vertices, indirect effects such as slightly more efficient propagation of changes in measures or better use of caching could help in speeding up the computation time needed by the implementation.

Experiments show that using the GPU can help in solving a number of parity games significantly faster, but the CPU implementation is still faster for the majority of the parity games tested. Combining both to run in parallel resulted in an implementation which is always competitive with existing solvers, and is even significantly faster on games that can be solved faster by the GPU than by the CPU.

10.1 Future work

This report is only an exploration of the possibilities of various techniques to speed up the small progress measures algorithm, or solving parity games in general. There are a number of techniques that appear promising and are worth further investigation.

The strategies described and implemented for both CPU and GPU are simple strategies, while more elaborate strategies exist for use on the CPU. While the GPU implementation can perform much more lifts per second than the CPU implementation, it is less efficient in general partly because of its parallel nature, but largely because it is difficult to find a good lifting strategy. The large amount of possible strategies available for the CPU suggests that there are more possible strategies to use for small progress measures on the GPU than the two described in this report, and investigating which strategies are possible and work well could result in a much more competitive GPU implementation in general.

Another option that can be of benefit to the speed of the GPU implementation is the advancement of architectures, such as AMDs Heterogeneous Computing, where classical CPU-cores are combined with GPU-cores on a single die, where both share the same memory and where the cost of moving computations between CPU and GPU will be negligible. Some parts of the GPU algorithm, such as determining the work queue for the explicit queuing strategy, or calculating the winning set in alternating SPM, are relatively slow to execute on the GPU. When the current overhead of doing those calculations on the CPU, consisting mostly of memory copying overhead, disappears, these bottlenecks can be overcome by using the right processor for the right job.

Assigning priorities to edges instead of vertices resulted in a slightly different way of solving parity games. Doing so allows shortcuts to be created in parity games, which can reduce the time required to solve these games. Section 6.3 described how the small progress measures algorithm must be adapted to be able to use edge measures, but other algorithms can potentially benefit from this technique as well. It is worthwhile to adapt algorithms such as the recursive algorithm for solving parity games to use edge priorities, and to investigate the hopefully positive influence of adding shortcuts to parity games on the running time of these algorithms.

Bibliography

[1] Jiri Barnat, Lubos Brim, Milan Ceska, and Tomas Lamr. Cuda accelerated ltl model checking. In *ICPADS*, pages 34–41. IEEE, 2009.

[2] Dragan Bosnacki, Stefan Edelkamp, and Damian Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In Corina S. Pasareanu, editor, *SPIN*, volume 5578 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2009.

[3] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[4] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 368–377. IEEE Computer Society, 1991.

[5] Oliver Friedmann and Martin Lange. Solving parity games in practice. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2009.

[6] Philipp Hoffmann and Michael Luttenberger. Solving parity games on the gpu. In Dang Van Hung and Mizuhito Ogawa, editors, *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 455–459. Springer, 2013.

[7] Michael Huth, Jim Huan-Pu Kuo, and Nir Piterman. Concurrent small progress measures. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Haifa Verification Conference*, volume 7261 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2011.

[8] Marcin Jurdzinski. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.

[9] J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking.* PhD thesis, Eindhoven University of Technology, 2013.

[10] Hartmut Klauck. Algorithms for parity games. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata, Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 107–129. Springer, 2001.

[11] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[12] Angelika Mader. *Verification of modal properties using boolean equation systems.* PhD thesis, Technische Universität München, 1997.

[13] NVIDIA Corporation. *CUDA C Programming Guide 5.0.* 2012.

[14] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[15] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[16] Jaco van de Pol and Michael Weber. A multi-core solver for parity games. *Electr. Notes Theor. Comput. Sci.*, 220(2):19–34, 2008.

[17] Freark van der Berg. Solving parity games on the playstation 3. In *13th Twente Student Conference on IT*, June 2010.

[18] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.