REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

GPU-Based Markov Decision Process Solver

Ársæll Þór Jóhannsson

Master of Science
June 2009

Reykjavík University - School of Computer Science

M.Sc. Thesis

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

GPU-Based Markov Decision Process Solver

by

Ársæll Þór Jóhannsson

Thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science

June 2009

Thesis Committee:

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University

Dr. Björn Þór Jónsson
Associate Professor, Reykjavík University

Dr. Hjálmtýr Hafsteinsson
Associate Professor, University of Iceland

GPU-Based
Markov Decision Process Solver

by

Ársæll Þór Jóhannsson

June 2009

Abstract

Markov Decision Processes provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the decision maker. MDPs are used in a variety of areas including robotics, automated control, planning, economics and manufacturing. For the solving of MDPs various different approaches exists. Value Iteration is an algorithm which falls under the class of Dynamic Programming methods and can be used to solve MDPs. In recent years Graphics Processing Units have been evolving rapidly from being very limited processing devices with the sole purpose of accelerating certain parts of the graphics pipeline into fully programmable and powerful parallel processing units. In the autumn of 2007 NVIDIA introduced CUDA, a hardware and software architecture for utilizing the GPU for general purpose calculations. In this thesis we introduce two parallel CUDA based implementations of the Value Iteration algorithm: Block Divided Iteration and Result Divided Iteration. We discuss the different approaches each algorithm takes for utilization of the parallel processing power of the CUDA device. We also present a framework we implemented which enables researchers to easily apply the parallel algorithms to MDPs within C or C++ applications. Empirical results are also presented which show a substantial performance improvement achieved by the parallel algorithms compared to a sequential implementation running on a CPU.

Lausn Markov ákvörðunarferla á grafíkkortum

eftir

Ársæll Þór Jóhannsson

Júní 2009

Útdráttur

Markov ákvörðunarferlar (e. Markov Decision Processes) skilgreina stærðfræðilegan ramma fyrir ákvörðunartöku við aðstæður þar sem niðurstaðan er að hluta til slembikennd og að hluta til undir þeim komin sem tekur ákvörðunina. Notast er við Markov ávörðunarferla á mörgum mismunandi sviðum til dæmis við stjórnun vélmenna, í sjálfvirkri ákvörðunartöku og skipulagningu, hagfræði og framleiðslustjórnun. Til eru margar mismunandi aðferðir til þess að leysa Markov ákvörðunarferla. Gildisítrun (e. Value Iteration) er reiknirit sem fellur undir flokk kvikra bestunar aðferða (e. Dynamic Programming) og hægt er að nota til að leysa Markov ákvörðunar ferla. Á undanförnum árum hafa skjákort verið að þróast frá því að hafa mjög takmarkaða reiknigetu og þann eina tilgang að auka hraða ákveðins hluta grafík-pípunar yfir í að vera gríðarlega öflug og að fullu forritanleg kort, sem sérhæfð eru í samhliða vinnslu. Haustið 2007 kynnti NVIDIA CUDA, nýjung í bæði vél- og hugbúnaði sem gerir kleift að nýta skjákort á auðveldan hátt fyrir almenna útreikninga. Í þessari meistararitgerð kynnum við tvö reiknirit sem byggð eru á gildisítrun en nýta sér samhliða vinnslu og keyra á CUDA grafíkkortum. Við ræðum á hvaða mismunandi hátt reikniritin nýta sér möguleika CUDA kortsins til samhliða vinnslu og kynnum einnig umgjörð (e. Framework), sem við útfærðum, sem gerir aðilum kleift að beita reikniritunum á auðveldan hátt á Markov ákvörðunarferla innan C eða C++ forrita. Niðurstöður eru kynntar þar sem sýnt er fram á umtalsverða yfirburði reikniritanna sem að nýta sér samhliða vinnslu í samanburði við hefðbundna útfærslu sem keyrir eingöngu á örgjörva tölvunnar.

To my parents, for all the support and motivation they have given me throughout my life.

Acknowledgements

I would like to thank my supervisor Yngvi Björnsson for his support and insightful feedback during the duration of this project. I also want to thank my wonderful girlfriend Belinda for all her support and patience. Last but not least I want to thank my family and friends for all of their support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Markov Decision Processes provide a mathematical framework for decision making. They are widely used for the solving of real-world problems of both planning and control as they are surprisingly capable of capturing the essence of purposeful activity in a variety of situations. For those reasons they have formed the basis on which many important studies in the field of learning, planning and optimization have been built upon. As a result several different techniques have been developed for their solution.

Two algorithms which fall under the category of Dynamic Programming and have been successfully applied to solving MDPs are *Policy Iteration* and *Value Iteration*. These Dynamic Programming methods perform sweeps through the state space and do a full backup operation on each state. Each backup updates the value of a single state based on the values of all possible successor states and the probability of ending up in that state. Because of the requirement of these dynamic programming methods for doing complete sweeps of the whole state space they are often considered impractical for very large problems. But, comparing them to other methods for solving MDPs we notice that they are actually quite efficient, and that they are guaranteed to find an optimal policy in polynomial time. They are also actually better suited for handling of large state spaces than competing methods such as direct search and linear programming (Sutton & Barto, 1998). Another attractive quality of the Dynamic Programming algorithms is that they do not require the states to be backed-up in any particular order or equally often in order to converge. Thereby they offer the opportunity for using different approaches to sweeping the state space.

In recent years *graphic processing units* have been evolving at a rapid pace from being very limited processing devices with the sole purpose of accelerating certain parts of the graphics pipeline into fully programmable and powerful parallel processing units. This

has made the GPU attractive for problems that can be expressed as *data-parallel computations* and has caused a recent trend in computer science referred to as *GPGPU* (General Purpose computing on the Graphics Processing Unit) where the graphics processing unit is used to perform the computations rather than the central processing unit. Until recently, however it remained difficult to utilize the power of the GPU for non-graphic problems since the GPU could only be programmed through a graphics APIs which provided a very limiting access pattern to the GPU's DRAM. NVIDIA noticed this increasing trend and introduced *CUDA* in the autumn of 2007. CUDA (Compute Unified Device Architecture) is a new hardware and software architecture for performing computations on the GPU as a data-parallel computing device without the need of going through the graphics API ("NVIDIA CUDA Programming Guide", 2008).

The main focus of this thesis is the design and implementation of a *GPU-based MDP Solver framework* that utilizes CUDA. The framework allows users to harness the power of the GPU for more efficiently solving problems which can be formed as MDPs. We describe a way to formulate problems to fit the framework and algorithms for solving them on CUDA enabled devices. As the GPU offers parallelism of a much higher degree than previous devices that have commonly been utilized for parallelizing Dynamic Programming methods, this offers both new challenges and new opportunities.

The thesis is structured as follows: In Chapter 2 we review background work and its relation to this thesis. Chapter 3 describes how the Value Iteration algorithm can be adjusted to the CUDA framework and presents our two different proposals for parallel value iteration. Chapter 4 gives an in-depth description of the implementation of the algorithms in relation to the CUDA programming model and necessary performance considerations. In Chapter 5 we describe the solver framework which can be used to easily apply these algorithms to any MDP within C or C++ applications. In Chapter 6 we present our experimental results and end by giving our conclusion in Chapter 7.

Chapter 2

Background

In this chapter we give a general overview of Markov Decision Processes and the most common methods used to solve them. We then introduce the fundamentals of parallel computing and give a brief overview of how to use GPUs for general purpose computation by utilizing the CUDA programming model.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the decision maker. They originated in the study of stochastic optimal control in the 1950s and have remained of key importance in that area ever since. Their theory has continued to develop over the last decades to fit a broader spectrum of problems and has lead to a wealth of common algorithmic ideas and theoretical analysis. Today MDPs are used in a variety of areas, including robotics, automated control, planning, economics and manufacturing.

An MDP consists of an agent and an environment that the agent interacts with. These interactions happen over a sequence of discrete time steps t; at each time step t the agent perceives the state of the environment s_t and selects an action a_t to perform. The environment reacts to the action by making a transition to a new state s_{t+1} and returns a scalar reward $r_{t+1} \in \mathfrak{R}$. The agent's goal is to maximize the total amount of reward it receives from its interactions with the environment. The dynamics of the environment are stationary and the state signal must contain all relevant information but is otherwise unconstrained.

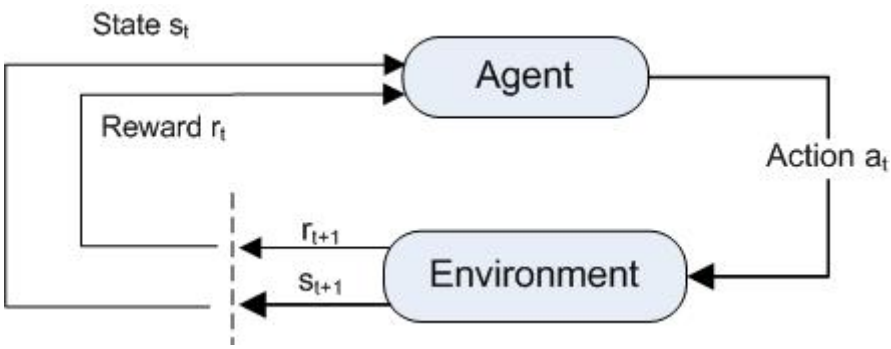

Figure 2.1: Markov Decision Process

An MDP is generally denoted by a four-tuple $(S, A, P_a(.,.), R_a(.,.))$. Where S is the state space, A is the action space, $P_a(s, s')$ is a function that determines the probability that taking action a in state s at time t will lead to state *s'* at time *t+1*, and $R_a(s, s')$ is a function that returns the expected immediate reward received after transition to state *s'* from state s. An MDP is called finite if the state and action sets are finite. The scope of this project includes finite MDPs.

For the agent to be able to maximize the reward from its interaction with the environment it must be able to evaluate the value of a state and implement a mapping from states to probabilities of selecting each possible action at each time step. This mapping is referred to as the agent's policy and is denoted by π, where $\pi_t(s, a)$ is the probability that action a will be selected at time t if we are in state s at time t. The estimated value of a state is defined in terms of future rewards that can be expected. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly its value function is defined with respect to a particular policy. Therefore the value of state s under policy π, denoted $V^\pi(S)$, is the expected return when starting in s and following π thereafter and can be defined formally as:

$$V^\pi = E_\pi\{R_t|s_t = s\} = E_\pi\{\sum_{k=0} \gamma^k r_{t+k+a}||s_t = s\} \tag{2.1}$$

where $E_\pi\{\}$ denotes the expected value given that the agent follows policy π and γ is the discount rate which determines how much we value future reward compared to immediate reward.

When we solve an MDP we are looking to obtain the optimal policy, which is defined as the policy with expected return greater than or equal to all other policies for all states. The optimal policy is denoted as π^* and there can be more than one optimal policy. The MDP framework is abstract, flexible, and provides the tools needed for the solution

of many important real-life problems. The flexibility of the framework allows it not only to be applied to many different problems but also in many different ways. For example, the time steps can refer to arbitrary successive stages of decision making and acting. The actions can be any decisions we want to learn how to make and the state can contain anything that might be useful in making them.

2.2 Solving Markov Decision Processes

The optimal policy can be computed by applying dynamic programming methods to the MDP. A key idea of the application of dynamic programming to MDPs is the use of value functions to organize and structure the search for good policies.

2.2.1 Policy Iteration

Policy Iteration is a dynamic programming algorithm that manipulates the policy directly when used to compute the optimal policy. It starts by evaluating an arbitrary policy, and then uses the value function of that policy in order to find better policies. This is done by considering a deviation from our current policy in state s where we want to know whether or not we should change the policy to deterministically choose an action a different from the one according to $\pi(s)$. We can determine if this change in the policy will lead us to a better policy by selecting a in s and thereafter follow the existing policy π. If we find out that the value of this new policy is greater then of our existing policy, then we have successfully improved our policy. This line of thought is natural to extend not only to consider a change for a single action in a single state but for all actions in all states. We would then evaluate each action in each state and select the actions that yield the highest return. This process of making a new policy that improves on an original policy, by making it greedy or nearly greedy with respect to the value function is called policy improvement. Once a policy π has been improved using V^{π} to yield a better policy π' we can compute its value function $V^{\pi'}$ and improve it again to yield an even better policy π'' where each policy is guaranteed to be a strict improvement over the previous one. Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. This way of interleaving policy evaluation with policy improvement is called policy iteration and is a fundamental algorithm in the study of MDPs.

2.2.2 Value Iteration

Value iteration is another dynamic programming algorithm that takes a different approach to obtain the optimal policy. Rather then manipulating the policy directly it obtains the optimal policy by computing the optimal value function. It does this by going through the state space and assigning each state the maximum estimated value based on the discounted value of its neighboring states. This iterative computation is continued until the maximum change in value for all states in each sweep is smaller then some predefined small positive number denoted as θ. The smaller the value of θ the higher the precision of the algorithm is. Value iteration requires each state to be processed only once in each sweep through the state space and thereby eliminates one of the drawbacks of policy iteration, which is policy evaluation which may require multiple sweeps through the state space. A formal description of the algorithm follows:

Algorithm 1 Value Iteration

$\mathcal{P}^a_{ss'}$ probability function {Returns probability of transistioning to state s' when action a is taken in state s}
$\mathcal{R}^a_{ss'}$ result function. {Returns the immediate reward recieved after transistioning to state s' from s with action a}
Initialize V arbitrarily e.g., $V(s) = 0$, for all $s \in S^+$
$\theta \leftarrow$ a small positive number
repeat
  $\Delta \leftarrow 0$
  for all $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$
Output a deterministic policy, π, such that
$\pi(s) = argmax_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$

The Value Iteration algorithm is flexible and does not require the value of the states to be computed in any strict order nor equally often in order to converge as long as all states are processed during a sweep (Moore & Atkeson, 1993). This gives the flexibility that the values of states can be computed in any order, using whatever values of other states that happen to be available; the value of some state can therefore be processed several times during a single sweep. This flexibility, along with its slow convergence rate, has been the catalyst for some efforts to speed up its computations (Bertsekas, 1982). Most of these efforts have been focused on one of two things, either parallelization or prioritizing of computation in an effort to reduce unnecessary computation (Wingate & Seppi, 2003,

2004, 2005). In this thesis our research efforts will be focused on the Value Iteration algorithm. The main reasons are that it is simpler than Policy Iteration, and studies have shown that in practice it is more robust and converges faster (Sutton & Barto, 1998).

2.2.3 Other Methods

There exists a variety of other agent based methods which can be used for the solving of MDPs. These methods mostly fall under two classes, Monte Carlo methods and Temporal Difference Learning methods. MC methods are based on averaging sample returns for the solving of problems. They differ from Dynamic Programming as they do not require a complete model of the environment and they do not bootstrap but rather the estimate for each state is independent. TD Methods, however, combine qualities from both of these method classes as they do not require a complete model of the environment and also bootstrap. TD methods are naturally implemented in an on-line and fully incremental fashion. Two well known TD methods worth mentioning are Q-Learning and Sarsa (Sutton & Barto, 1998).

2.3 Parallel Algorithms

Parallel computing is defined as a form of computing where multiple calculations are carried out simultaneously. It is derived from the principle that large problems can often be divided into smaller ones which can then be solved in parallel. Traditionally, to solve a problem an algorithm is constructed and implemented as a serial stream of instructions. These instructions are then executed one at a time on a central processing unit on a single computer. With parallel computing, on the other hand, multiple processing elements are used concurrently to solve a problem. This is accomplished by breaking the problem into independent parts and delegating them to the processing elements so that each of them can execute its part of the algorithm simultaneously with the others. The processing elements can range from a single computer with multiple processors, to a network of computers or specialized hardware (Wikipedia, 2009).

Understanding data dependencies is fundamental for implementing a parallel algorithm. No program can run faster than the time it takes to process the longest chain of dependent calculations, since calculations that depend upon prior calculations in the chain must be executed in order. Most algorithms, however, do not consist of only long chains of

dependent calculations but have segments of independent calculations which can be parallelized. Another fundamental property which parallel algorithms must preserve is sequential consistency, which guarantees that the results of the algorithm are same as if its operations were executed sequentially.

There are several different types of parallelism. The parallelism models that algorithm designers normally use when designing parallel algorithms are the data-parallel model and the task-parallel model. Data parallelism is defined as parallelism which is inherent in program loops. Data parallel programs normally unroll these loops if possible and distribute the data across computing nodes to be processed in parallel. This model fits well for systems capable of SIMD execution. SIMD stands for Single Instruction Multiple Data and means that the system is capable of executing the same instructions on multiple data elements concurrently. Task parallelism, on the other hand, is characterized by the ability to execute completely different sets of instructions on either the same or different sets of data in parallel. The task parallelism model fits MIMD (Multiple Instruction Multiple Data) capable systems well, such as computing clusters.

If we look at how parallelization should theoretically affect performance it is not illogical to assume that we would get a linear speed-up, that doubling the number of processors will reduce our runtime by half. However, it is not quite that simple since the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Therefore, no matter how many processors we devote to a parallelized executing of a program, the minimal execution time can not be less than the time it takes to execute its sequential part. Amdahl's Law can be used to predict the maximum possible speedup achievable when a problem is parallelized using parallel processors in comparison to using only a single serial processor, given that the problem size remains the same when parallelized (Wikipedia, 2008). It states that if P is the proportion of a program that can be made parallel, then the serial portion can be defined as (1-P). If we then define the total time for the serial computation as 1 for any time unit, we can compute the speedup by dividing the old computation time with the new computation time that consist of the serial portion plus the parallel portion divided by the number of parallel processors denoted as N. This gives us the following equation:

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{2.2}$$

Then we can see that as the number of processors tends to infinity the maximum speedup tends to

$$\frac{1}{(1-P)} \tag{2.3}$$

and the serial portion becomes dominant.

2.4 GPGPU and CUDA

The idea of using computer graphics hardware for general-purpose computation has been around now for over two decades. However, GPGPU did not really take off until ATI and NVIDIA introduced programmable shading in their commodity GPUs in 2002. This enabled programmers to write short programs that were executed for each vertex and pixel that passed through the rendering pipeline. Researchers were quick to realize that this was not only useful for graphics programming but that this could also be used for general purpose calculations. They now had an extra processor to speed up the computation of their problems and to make it even more attractive the processing power of the GPU has been increasing at a much faster pace than of the CPU. This caused a swift increase in research that utilized GPGPU computation.

Graphics processing can be parallelized as each vertex or pixel can most often be processed independently of other vertexes or pixels in each step of the graphics pipeline. As GPU development has mainly been driven by computer games and the quest for better and faster graphics it has caused the GPU to become specialized for intensive, highly parallel SIMD computation and is therefore designed such that more transistors are devoted to data processing rather than data caching and flow control unlike the CPU. Today GPUs have multiple cores driven by a high memory bandwidth, offer massive processing resources, and are especially well-suited to address problems that can be expressed as data-parallel computations.

The utilization of the processing power of the GPU, however, did not come for free. It required researchers to pose their problems as graphics rendering tasks and go through the graphics API, which is very restrictive when it comes to programming. The APIs where used in such a way that textures were used for input and output and fragment shaders (program stubs which run for each pixel projected to the screen) were used for processing. This meant a high learning curve for programmers not already familiar with the graphics APIs, and the environment was very limiting when it came to debugging. This also greatly narrowed the range of potential problems that could be solved by using the GPU. GPU manufacturers, however, noticed these efforts and have now introduced both software and hardware to greatly simplify the use of GPUs for general purpose computation.

In late 2007 NVIDIA introduced CUDA, a parallel-programming model and software environment designed to enable developers to overcome the challenge of developing ap-

plication software that scales transparently over parallel devices of different capabilities. With CUDA they also introduced a new line of graphics processing units that supported this environment and with its application these devices were no longer standard GPUs but became massively parallel stream processors which were now programmable through standard C.

2.4.1 CUDA Enabled Devices

The *GeForce 8 Series* were the first graphics cards to supported this architecture, in addition to NVIDIA's Tesla line which is a family of products for high performance computing. The GPUs of the GeForce 8 series have a large number of stream processors ranging from 16 to 128 depending on make and type. All of the processors of each device are of the same type with similar memory access speeds, which makes these devices a massive parallel processor. On the hardware level these Geforce 8 series devices consist of a collection of multi processors with 8 processors each. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture and constant memory caches. At any given cycle, each processor in the multiprocessor executes the same instruction on different data, which makes each multiprocessor a SIMD processor. Communication between multiprocessors is through the device memory, which is available to all the processors in the multiprocessor. One can access all the available device memory by using CUDA with no restriction on its representation, though the access times vary for different types of memory. The CUDA hardware employs a new hardware architecture called SIMT (Single Instruction Multiple Thread). It is similar to SIMD, but does not require all the code to follow the same execution path; instead it enables programmers to write code which specifies the execution and branching behavior of a single thread. If, during execution some of the threads which are being executed in parallel diverge, the hardware automatically serializes the branch and executes each branch path independently. This hardware architecture enables programmers to write thread-level parallel code for independent threads, as well as data-parallel code for coordinated threads. In our research we use a NVIDIA 8800 GTX device which has 768 MB of RAM and 8 multiprocessors which combine to a total of 128 stream-processors.

2.4.2 CUDA Programming Model

Through the CUDA model the programmer sees the CUDA device as a co-processor to the main CPU capable of executing a collection of threads in parallel. The CUDA

model abstracts these threads into a two-level hierarchy. First we have *blocks* which are a collection of threads that are all assigned to the same multiprocessor and are therefore able to share access to the processors' on-chip memory and their execution can also be synchronized by defining synchronization points in the code. Multiple blocks can be assigned to a single multi-processor and their execution is then time-shared. All threads of all the blocks that are assigned to a single multiprocessor divide its resources equally amongst themselves. The CUDA model also defines a *warp* as a collection of threads that can run simultaneously on a multiprocessor. The warp size is fixed and depends on the device; on the GeForce 8 series it is 32. If the number of threads is greater than the warp size, they are time-shared internally on the multiprocessor. A *grid* is the collection of all the thread blocks that are executed on the device at once.

The programmer is able to control the number of threads executed on the device by defining the number of threads within a block and the number of blocks within a grid. Each thread within the grid executes the same set of instructions referred to as a kernel. Each thread and block within the grid is assigned a unique identifier that can be used to assign different data to each thread or block.

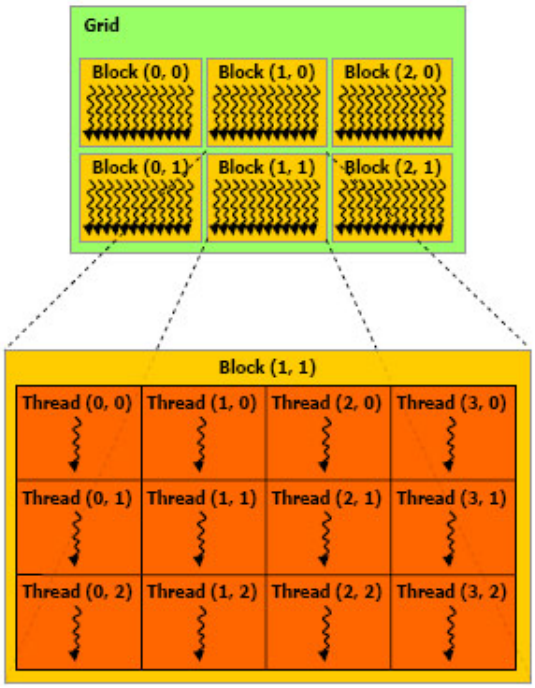

Figure 2.2: Division of a grid into blocks and threads. *(Image from NVIDIA's CUDA Programming Guide)*

When a kernel is executed on the device it has only access to the memory space of the device which is separated into on-chip memory and DRAM. The DRAM is divided into three different types of memory: global memory, constant memory and texture memory.

Each grid has read and write access to the global memory but only read access to the constant and texture memory. The global, constant and texture memory spaces can be read from or written to by the host and they are persistent across kernel launches by the same application. The on-chip memory space is reserved for registers and shared memory. The shared memory can be accessed from threads within the same block and is much faster than global memory access. Efficient use of shared memory is a key element to accelerating applications. These additions are exposed to the programmer as a set of extensions to the C language, and therefore should impose a relatively low learning curve for programmers familiar with standard programming languages such as C.

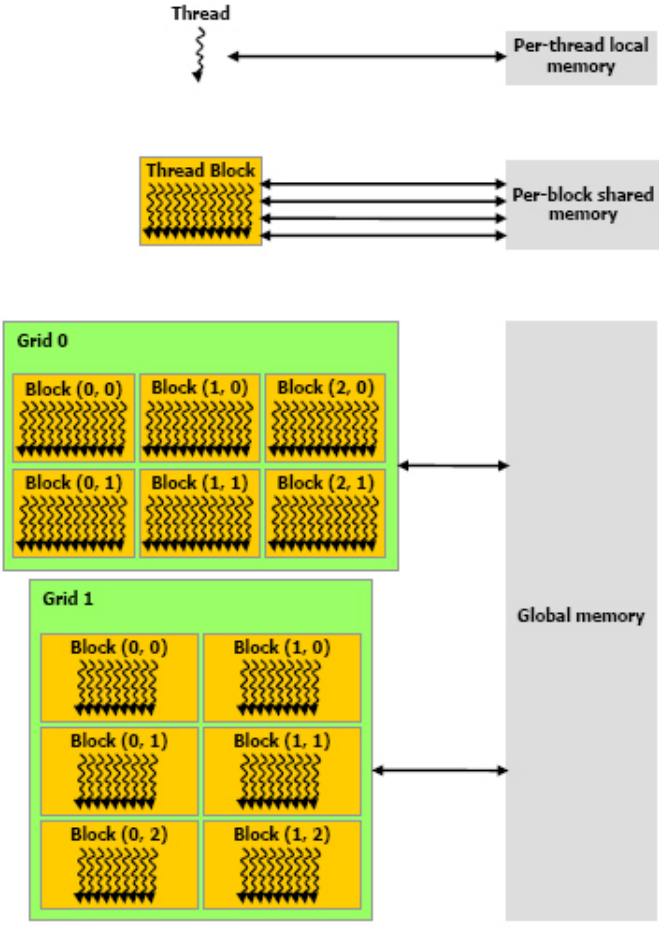

Figure 2.3: CUDA different types of device memory. *(Image from NVIDIA's CUDA Programming Guide)*

Through this abstraction the framework provides fine-grained data parallelism and thread parallelism nested within coarse-grained data parallelism and task parallelism. This guides the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel ("NVIDIA CUDA Programming Guide", 2008).

2.5 Summary

In this chapter we introduced the mathematical framework provided by MDPs for the modeling of decision problems. We then gave a brief discussion on how dynamic programming methods can be used for solving MDPs, and introduced two such algorithms: Policy Iteration and Value Iteration. Next we introduced the principles of parallel computing and how to estimate the potential speedup gained by parallelization. Finally we introduced the concept of using GPUs for General Purpose Computation, gave a description of the various resources of CUDA enabled devices and finally introduced the CUDA programming model. In the next chapter we present different ways of parallelizing the Value Iteration algorithm for execution on CUDA devices.

Chapter 3

Methods

In this chapter we analyze how well the Value Iteration algorithm is suited to the computational platform provided by CUDA, and what our potential gain in speed is if we successfully parallelize it. We then propose two different approaches to parallelizing the Value Iteration algorithm for the CUDA framework and suggest two performance enhancements inspired by prior efforts.

3.1 Solving MDPs With the GPU

The solving of MDPs by computing the value function maps well to the CUDA architecture. Many attempts to enhance the performance of the basic Value Iteration algorithm take advantage of the fact that it does not require the states to be processed in any particular order nor equally often to converge. This has most often resulted in algorithms that utilize some kind of partitioning of the state space and/or parallelizing of the computation to gain speedups. This can be achieved in two ways. By partitioning the state space, it can be divided between different computational resources, and secondly those computational resources can then be focused on certain parts of the state space where more passes are required for convergence. Research on asynchronous dynamic programming shows that it does not affect the convergence property of the algorithm though the backup of a state uses values of states that are not necessarily up to date as long as in the end the value of all states are backed up. This creates a certain data independence feature in the algorithm which makes it an ideal candidate for parallelization. These two features, partitioning and parallelization, fit nicely with the computational model supplied by CUDA.

The issue of allocating states to partitions is not treated in this work since it is outside the scope of this research. For domains with obvious and symmetric state relations, such as GridWorld, adequate partitioning can be constructed by simply grouping together connected states. However, for more general and complex MDPs, where the state relation is not as obvious, partitions could be generated by some existing k-way minimum-cut graph partitioning algorithms.

There has already been a fair amount of research focused on parallelizing calculations of the value iteration algorithm. These efforts have mostly focused on formulating the theory and if implemented they have been done on either specialized shared memory multi-CPU systems or cluster systems. Both of these types systems are limited in the parallelization they can offer since shared memory multi-CPU systems are expensive and additions to increase parallelism can cost thousands of dollars, while cluster systems incur a high communication cost since they do not have shared memory. This communication cost increases considerably with each computational resource addition so the threshold where the communication cost outweighs the benefit in performance is low. Both of these systems offer parallelism from four to a couple of dozen processes working on a problem at once. The parallelism offered by today's modern GPUs is quite different from these standard parallelization platforms, with the number of processors ranging from 112-240, with each capable of running several light-weight threads at once, these devices offer the opportunity of having thousands of threads working on a single problem in parallel. This higher degree of parallelism offers both new challenges and opportunities.

3.1.1 Analyzing Value Iteration for Parallelization

Though the Value Iteration algorithm is not complex, it offers at least two different possibilities for parallelization. If we go step-wise through the definition of the Value Iteration algorithm (see Algorithm 1, on page 6) with parallelization in mind the first thing that we notice is that the algorithm is defined per state. This gives us the possibility of parallelizing the entire algorithm on a state-wise level, excluding the convergence check. We also notice that the most essential part of the algorithm, the state value calculations, are a maximization over the action values of a state and that an action value is composed from the summarization of all possible successor states for that action. Both of these offer further possibilities for parallelization. The only part of the algorithm which is not inherently independent is the convergence check, which is dependent on the overall change in state value of each state. According to this analysis we can see that a large part of the algorithm can be altered and implemented in a parallel manner.

3.2 Proposed Methods

We propose two parallel algorithms that take into consideration the CUDA computational model and the parallelization possibilities of Value Iteration. Subsequently we suggest two additions which possibly could enhance performance.

3.2.1 Block Divided Iteration

Block Divided Iteration (BDI) is our proposal for a parallel version of the Value Iteration algorithm with a state-wise parallelization. The principal components of the parallel algorithm are the same as of the sequential one except that it has been adjusted to fit the computational model supplied by CUDA. It works by partitioning the state space into blocks which are equal in size and count to the execution blocks launched on the CUDA device. Within each block a thread is assigned a single state and is made responsible for the calculations of that state. A prerequisite of the algorithm is that before it is executed it requires calculations to be made that tell whether the successor states of the states actions reside within the same block.

Algorithm 2 shows a formal description of the algorithm. If we go stepwise through that description then the first two lines describe the two types of memory we use to store our state values. Line 3 describes our memory access function, it uses values from shared memory if they reside within the block, else values from global memory are used. Line 4 contains the convergence check which is executed on the host and the following lines are executed until the algorithm converges. From line 5 the execution has moved on from the host to the device and the remainder of the algorithm through line 19 is executed by each thread in parallel. In line 6 we have each thread loading its corresponding state value from global memory to shared memory, since good usage of the shared memory is critical for efficient execution. Next we do a thread synchronization to ensure that each thread in the block has loaded his state value into memory before we start the actual computation. Next we store the old value of the state and start our iterative computation of the state's value. In line 9 we start to execute a predetermined number of iterations on the state's value. Then we assign 0 to the state's Δ_s value which indicates how much its value changed during each iteration. This is different from the original algorithm as we now have a Δ value for each state in the state space. In lines 11 to 13 we iterate through each state's all possible actions and successor states and calculate the state's value accordingly. For each successor state s' however a check is made to determine if the value of s' resides within the block or whether it is necessary to go to global memory since communication between

Algorithm 2 Block Divided Iteration

1: $V_g(s)$ {State space residing in global memory}
2: $V_s(s)$ {State space residing in shared memory}
3: $V_{InBlock}(s) = \begin{cases} V_s(s) & \text{if } s \text{ is within the block} \\ V_g(s) & \text{if } s \text{ is not within the block} \end{cases}$
4: **while** $\Delta > \theta$ **do** {Converge check is done on host}
5: **for all** $s \in S$ in parallel **do** {Executed on device in parallel}
6: $V_s(s) \leftarrow V_g(s)$ {Copy state values from global memory to shared memory}
7: Synchronize threads
8: $v \leftarrow V_s(s)$
9: **for** iterations < number of episodes per convergence check **do**
10: $\Delta_s \leftarrow 0$
11: **for all** $a \in A \Rightarrow ss'$ **do**
12: $V_s(s) \leftarrow max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_{InBlock}(s')]$
13: **end for**
14: $\Delta_s \leftarrow max(\Delta_s, |v - V_s(s)|)$
15: $V_s(s) \leftarrow v$
16: Synchronize threads
17: **end for**
18: $V_g(s) = V_s(s)$
19: **end for**
20: **for all** Δ_s of $s \in S$ in parallel **do**
21: $\Delta \leftarrow max(\Delta_s)$
22: **end for**
23: Copy Δ to host
24: **end while**

blocks is not possible. Since it is more expensive to fetch values from the global memory some thought must be put into minimizing inter block dependencies when partitioning the state space. Next, in line 14 and 15, we assign the state its new Δ and state value and then do a synchronization to ensure that each thread in the block has completed a single sweep over its values before we continue. After all the sweeps have completed we copy the state's value back from shared memory to global memory since it is not persistent between kernel launches. In lines 20 to 22 we do a parallel max over the Δ values to determine the maximum Δ value. We then finally copy the Δ value to host memory for the converge check.

3.2.2 Result Divided Iteration

The Result Divided Iteration (RDI) algorithm parallelizes the value iteration algorithm differently from the BDI algorithm. It focuses on efficient parallelization of all subtasks of the value iteration process. It incorporates all the steps of the sequential algorithm

except that each step is now distributed between as many threads as possible. This kind of parallelization of all subtasks is only possible because of the massive parallelization ability of CUDA devices.

Algorithm 3 Result Divided Iteration

1: $V_g(s)$ {State space residing in global memory}
2: **while** $\Delta > \theta$ **do** {Converge check is done on host}
3: **for all** $s, a \Rightarrow s'$ where s and $s' \in S$ and $a \in A$ in parallel **do**
4: $Q_{s'}(s, a) \leftarrow \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_(s')]$ { Calculate value for all possible results }
5: **end for**
6: **for all** $Q_{s'}(s, a)$ where s and $s' \in S$ and $a \in A$ in parallel **do**
7: $Q(s, a) = \sum Q_{s'}(s, a)$
8: **end for**
9: **for all** $s \in S$ in parallel **do**
10: $v \leftarrow V(s)$
11: $V(s) \leftarrow max(Q(s, a))$
12: $\Delta_s \leftarrow (|v - V(s)|)$
13: **end for**
14: **for all** Δ_s of $s \in S$ in parallel **do**
15: $\Delta \leftarrow max(\Delta_s)$
16: **end for**
17: **end while**

Instead of having a thread for each state in the state space like in the Block Value Iteration algorithm we now have a thread for each possible successor state for each action in every state. That thread batch calculates the result value for each possible successor state, this can been seen in lines 3 to 5. In lines 6 to 8 another batch of threads is launched which is equal to the total number of actions for every state and that batch summarizes all of the result values into state action values for each state action combination. Then the third batch of threads is launched in lines 9 to 13. This batch is equal to the state space in size and calculates the maximum of each action value for each state and assigns that maximum to the state value. Finally in lines 14 to 16 a parallel max is performed to determine the maximum delta value for the convergence check.

The RDI algorithm offers a much more fine grained parallelism than the BDI algorithm. The dynamics of the MDP which is to be solved also has no influence on how the MDP is divided for parallelization. For the BDI algorithm, on the other hand, domains with a low number of states and very high number of actions would not fit very well as it would result in very few threads being launched and would require each thread to do a lot of computation which would not map well to the fine grained parallelism of the CUDA model. The fine grained parallelism of the RDI algorithm, however, limits considerably

the amount of computation performed by each thread and is therefor unable to gain much from shared memory use.

3.3 Performance Enhancements

In this section we introduce two simple changes to the BDI algorithm in an effort to improve its performance. These enhancements are of a different nature as one is based on better usage of the various kinds of memory available to us on CUDA device's while the other is based on prior research efforts for focusing computations where they are most needed.

3.3.1 Buffering Enhancement

Since the Value Iteration algorithm does allow the value of states to be computed from values which are available at each time and still converge correctly it allows us to introduce a small addition to the BDI algorithm in the effort to reduce global memory access. The change involves that now each thread caches in shared memory the values that reside outside its block, but are needed for the computation of its value. For the computation of each value it will no longer need to go to global memory but will rather use the value of this shared memory cache which is faster.

3.3.2 Prioritizing Enhancement

The prioritizing enhancement is another small addition to the standard BDI algorithm that is focused on reducing unnecessary computation. With this enhancement each thread is not assigned a single state but rather a predetermined number of states, somewhere in the range of two to five states. In each pass the state with the highest need for computation is selected within each thread and its value computed. For controlling the state selection we use a simple heuristic that factors in the number of times a state has been selected and its overall change in value the last time it was updated.

$$\Delta_s + U_c * \frac{1}{UpdateCount} \tag{3.1}$$

Equation 3.1 shows how the eligibility of state for update is computed. U_c is an update constant. UpdateCount is the number of times the state has been updated during this sweep. This enhancement can be used whether buffering is used or not.

3.4 Summary

In this chapter we have shown how the CUDA framework can be applied to the solving of MDPs through value iteration. We presented an analysis of the Value Iteration algorithm with respect to parallelization. From that analysis we derived and presented two fundamentally different parallel versions of the value iteration algorithm, Block Divided Iteration and Result Divided Iteration. Additionally, two possible performance improvements were introduced for the BDI algorithm. In the next chapter we look in more detail into the implementation of these algorithms for CUDA.

Chapter 4

Implementation

Although programming of the GPU for general purpose computation has been simplified by the introduction of CUDA it is far from trivial to implement algorithms that efficiently utilize its computing power. Many algorithms can be converted without much effort to run on CUDA, but if we strive to utilize the full potential of the CUDA device for achieving high performance, several things must be considered. In this chapter we give an overview of the topics which are necessary to consider when implementing code for execution on CUDA devices. We then discuss how each topic influenced the implementation of our algorithms.

4.1 Arithmetic Intensity and Shared Memory

The first thing to consider is arithmetic intensity, which is the ratio of computation to bandwidth. The greater arithmetic intensity our algorithm has the more likely it is to gain from being executed on a CUDA device. CUDA devices also have very fast on-chip memory called shared memory which can be can be accessed from threads within the same block, usage of this memory can make a big different in the speedup achieved from moving algorithms to CUDA devices.

4.1.1 Considerations

To get the most from our CUDA device we want to optimize our algorithm so that we do as much computation per memory access as possible. Some instructions are also more costly than others, such as integer divisions and modulo operations. The use of these low

throughput instructions should be minimized and replaced with bitwise operations where possible. Any flow control instructions can significantly impact the effective instruction throughput by causing threads of the same warp to diverge. If this happens, the different executions paths must be serialized, increasing the total number of instructions executed for this warp. Unrolling loops, where possible, reduces conditional statements in the code and can greatly increase the performance of the code.

Efficient usage of the device's different types of memory is crucial. Issuing a memory operation (read/write) takes 4 clock cycles, but when accessing global or local memory there are additional 400-600 clock cycles of memory latency. On the other hand, much of the global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. Since global memory is of a much higher latency and lower bandwidth than shared memory, global memory accesses should be minimized and shared memory utilized where possible.

The shared memory is divided into equally sized memory modules that can be accessed simultaneously to achieve its high memory bandwidth. That enables any memory read or write requests for n addresses that fall into equally many distinct memory banks to be serviced simultaneously. If, however, two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access must be serialized. To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts. Figure 4.1 compares different types of access patterns to shared memory and shows whether they cause banking conflicts or not.

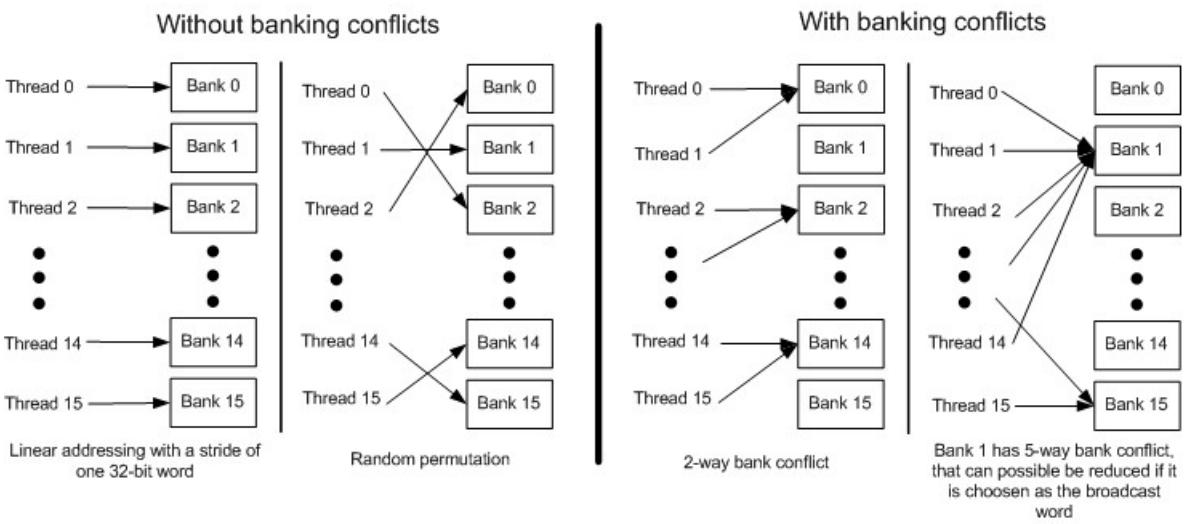

Figure 4.1: Shared memory access patterns

4.1.2 Effect on our Implementation

In our implementation of the Block Value Iteration algorithm we considered having each thread load all the state transition information of its actions into shared memory. Each thread would then be able to calculate its value without going to global memory, if the value of all its preceding states remained within its block. This comes at the cost of added shared memory usage per thread, which further limits the number of thread blocks available to run on each processor. The most limiting factor of this method, however, is that it makes the shared memory usage of each block dependent on the dynamics of the MDP. As the number of actions and possible outcomes per action impacts the shared memory required by each thread, and if the number of actions or possible outcomes per actions is increased the shared memory requirement can quickly becomes greater than the capabilities of the device. The same dilemma is encountered with the implementation of the buffering enhancement for the BDI algorithm. We therefore found it more important that the amount of shared memory allocated per thread would not be dependent on the dynamics of the MDP to be solved. The state and delta values for each state are stored in shared memory since the dynamics of the MDP do not affect their memory requirements.

4.2 Global Memory Access Pattern

Since accessing the global memory is expensive and not cached, it is important to follow the right access pattern to get the maximum memory bandwidth.

4.2.1 Considerations

To efficiently access the global memory it firstly requires data-structures to be aligned to 8 or 16 bytes, since the device is capable of reading 32-bit, 64-bit, or 128-bit words from global memory into a register in a single instruction. Properly aligning data-structures can make a big difference when loading large amounts of data from global memory. Secondly, access to global memory by all 16 threads of a half-warp is coalesced into one or two memory transactions only if it satisfies three conditions.

- All threads are accessing a 32-bit, 64-bit, or a 128-bit word.
- All the 16 words lie in the same segment which is of size equal to the memory transaction size.
- The threads must be accessing the words in a sequence.

If a half-warp does not fulfill these requirements, a separate memory transaction is issued for each thread and throughput is significantly reduced.

An intuitive way to define data-structures for the definition of an MDP would be to define a struct for the definition of a state, and another struct that resides within the state struct that represented each possible transition for each action. The complete state space could then by contained in an array of state structs. An example of this kind of struct definition can be seen in Listing 4.1.

```
1  typedef struct State
2  {
3          int stateId;
4          float delta;
5          float value;
6          StateTransistion Transistions[];
7  } State;
8
9  typedef struct StateTransistion
10 {
11         float probability;
12         float reward;
13         int resultStateID;
14 }StateTransistion;
```

Listing 4.1: Naive struct definitions

Although this representation is easy to understand and work with, it will result in a highly inefficient memory access pattern. The size of the StateTransistion struct is 12 bytes and so is the State struct, including additional 12 bytes for each StateTransistion it contains. This does not conform yo the 8 or 16 bytes alignment requirement of data-structures for coalesced reading and will result in a so-called misaligned read (see Figure 4.2). In addition the data for each state will become aligned together in memory which makes it impossible for us to make the kth thread in a half-warp access the kth element in the block being read. Therefore all global memory access with these data-structures will be uncoalesced and a separate read instruction issued for each variable.

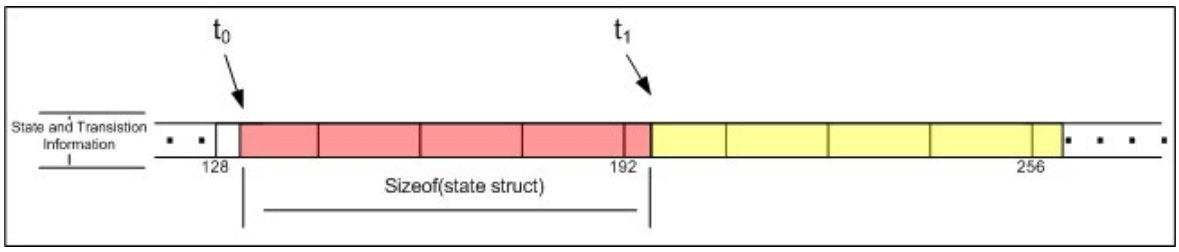

Figure 4.2: Access pattern for array of structs data structures. As the datastructures are only 12 bytes and not aligned to 16 bytes it will result in a mis-aligned read

In order to achieve coalesced memory transactions we must transform this representation to satisfy the three conditions stated earlier. If we will be having a single thread reading each state it is straight forward to coalesce the reading of state information by changing the state space representation from an array of structs representation to a struct of arrays representation. That is done by defining an array for each variable that a state contains, such a representation can be seen in Listing 4.2. The same can be done for the state transitions, but to achieve coalescing there we need to load the state transition data into the array in such a way that we do not get misaligned memory reads, that however can be very challenging as the alignment of data into memory is dependent on the number of actions and successor states of the MDP.

```
1  typedef struct StateSpace
2  {
3          float stateValues[NUM_STATES];
4          float deltaValues[NUM_STATES];
5  }
6
7  typedef struct Transistions
8  {
9          float2 probabilityAndReward[NUM_STATES * NUM_ACTIONS * NUM_DIVERGENCE];
10         int resultState[NUM_STATES * NUM_ACTIONS * NUM_DIVERGENCE];
11 }
```

Listing 4.2: Optimized Structs

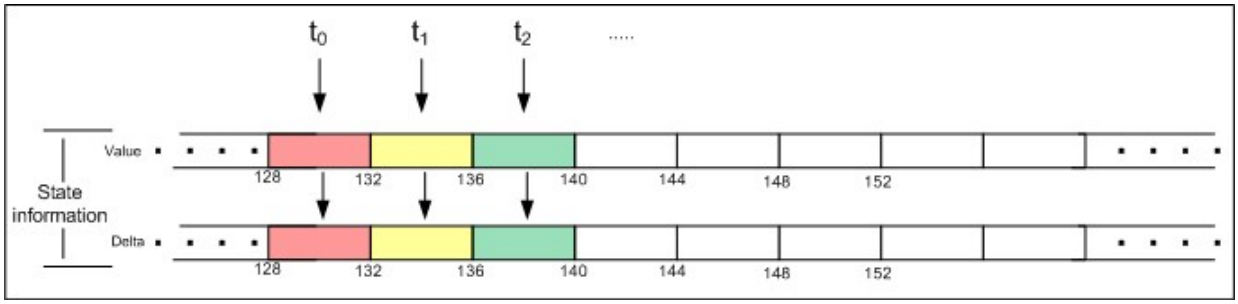

Figure 4.3: Access pattern for optimized state data

By defining the state data this way we should be able to get coalesced reads/writes for the state and delta values. Each thread now accesses a single float of 4-bytes which are all aligned in a sequence, this can be seen in Figure 4.3.

This definition of the transition data causes the probability value to be read in a single 8-byte read by each thread, and a 4-byte read for the result state id. All of these reads can be coalesced if the transition data is loaded so the memory reads of the threads can be in sequence and the dynamics of the MDP does not cause the alignment to break any of the requirements for coalescing, this can be seen in Figure 4.4.

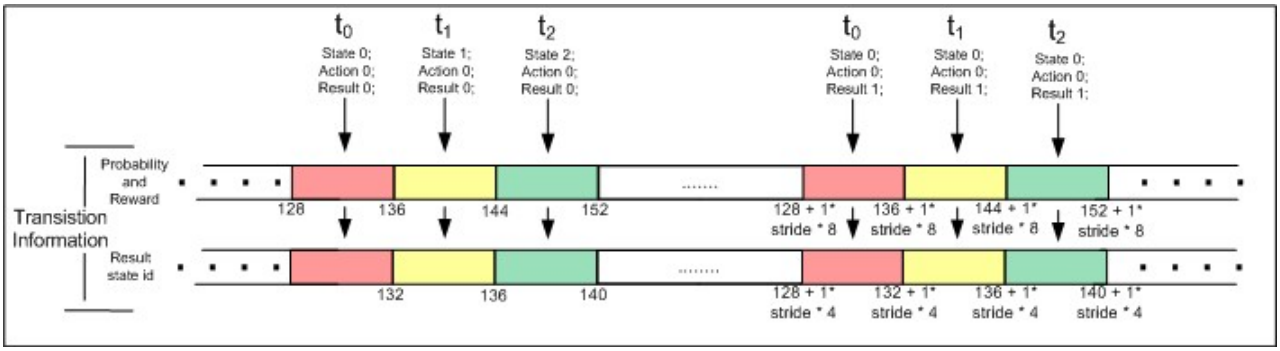

Figure 4.4: Access pattern for optimized transition data

4.2.2 Effects on our Implementation

In our algorithms we are dependent on values which must reside in global memory for the state value calculations since we store the dynamics of the MDP in global memory. The problem with that is that each memory read from global memory has very high latency and since it has no cache and our algorithm does not have high enough arithmetic intensity it is not be able to hide all the delay. Our solution is to utilize texture memory for storing the dynamics of the MDP instead of global memory. Texture memory resides in the same memory space as global memory but differs in a few important ways. Texture memory can only be written to by the host, which is not a problem for us since the dynamics of the MDP are unchanged during the computation. Texture memory has a cache which is optimized for two dimensional locality. In our implementation we use a 3D texture where the X and Y coordinates are used to identify the state, while the Z coordinate indexes each possible successor state. Since each thread will be accessing the same values in the texture memory when iteratively computing the values of the state we can expect a high hit rate in the cache and a noticeable increase in performance. The state and delta values, which are loaded into shared memory at each kernel launch and back to global memory at the end of its execution, are aligned in memory in such a way that coalesced reading and writing from and to global memory is ensured.

4.3 Grid and Block Dimensions for Efficient Execution

When determining the number of threads per block and the number of blocks per grid several things must be considered and the dimensions should be chosen to maximize the utilization of the available computing resources.

4.3.1 Considerations

When determining the size of blocks and threads to be launched, it is important to keep in mind that there should be at least as many blocks as there are multiprocessors on the device. Running only a single block per multiprocessor, however, will force the multiprocessor to idle during thread synchronization and also during device memory reads if there are not enough threads per block to cover the load latency. Ideally there should be at least two or more blocks per multiprocessor but the amount of shared memory available to a single multiprocessor limits the number of possible blocks per processor. The number of threads per block should be chosen as a multiple of the warp size to avoid wasting computing resources with under-populated warps. Allocating more threads per block is better for efficient time slicing, but the more threads there are per block the fewer registers are available for each thread. Since this balance can be hard to achieve, NVIDIA supplies the CUDA Occupancy Calculator with the CUDA Standard development kit. The CUDA Occupancy Calculator is a Microsoft Excel spreadsheet which allows you to input your kernel's register count, shared memory usage, and number of threads per block, it then outputs the total occupancy of each multiprocessor and enables you to determine what are the limiting factors of your kernel.

4.3.2 Effects on our implementation

We used the CUDA Occupancy Calculator to determine the optimal block size for the execution of our algorithms. The BDI algorithm was assigned a block size of 124 threads which resulted in a maximum of 67% occupancy per multiprocessor while the RSI algorithm was assigned a block size of 256 threads which resulted in a 100% occupancy per multiprocessor.

4.4 Summary

In this chapter we presented the most important issues to keep in mind when implementing algorithms for execution on CUDA devices, and discussed how this affected the implementation choices of our algorithms. Next we introduce a framework which enables us to easily apply these algorithms on MDPs within C or C++ applications.

Chapter 5

GPU Based MDP Solver Framework

Value Iteration is an efficient and robust algorithm which can be applied to a wide range of decision-making problems. However, the curse of dimensionality can cause its computational costs to quickly become overbearing for many learning systems. One of the main benefits of using the GPU for doing general purpose calculations is not only that it can potentially speed up your algorithms but also that now you have an extra independent processor which allows the processing power of the CPU to be utilized for something else. This can be of great benefit for techniques which rely on accurate value function estimates to make other decisions. In this chapter we describe our implemented framework which is a general framework for utilizing CUDA devices for solving of finite MDPs.

5.1 Framework Structure

The framework is implemented in C, is simple to use, and can be easily extended. The framework can be divided into three parts relative to their distance from the CUDA device. First we have the MDP Solver Interface which lies the furthest from the CUDA device and is a collection of function declarations which enables users to utilize the GPU solver from any C or C++ program without writing any CUDA specific code. The second part lies between the interface exposed to the user and the code executing on the device. This part contains the implementation of the functions exposed via the interface, handling of memory allocation for both the host and the device, launching of the device kernels and other service tasks. This part also includes some utility functions which are not exposed to the user but can be handy when debugging or extending the framework. The third part of the framework includes the collection of the various algorithms that have been implemented to run on the CUDA device.

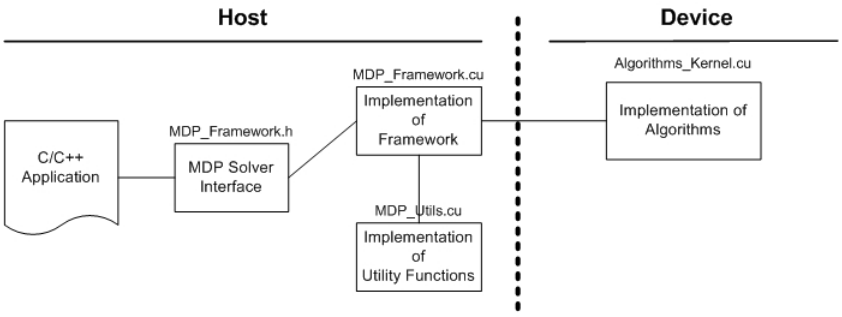

Figure 5.1: Frameworks structure

5.2 The MDP Solver Interface

The interface provided by the solver will be the main point of interest for most users. It enables them to easily solve MDPs within C or C++ application with very few lines of code. The functions of the framework can be categorized as follows:

- Initialization of device: The framework contains two functions for initializing and uninitializing the CUDA device.

```
//CUDA Initialization function
extern "C" bool InitializeCUDA();

//CUDA De-Initialize function
extern "C" void DeInitializeCUDA();
```

Listing 5.1: Initialization of device

- Initialization of domain and constants: The framework contains functions for initializing the domain and constants required for the algorithms both on the host and device.

```
//Initialize the dimensions of the MDP and the necessary memory
//Parameters:
// stateCount - number of states in the MDP
// actionCount - maximum number of actions per state in the MDP
// divergenceCount - maximum number of successor states per action in the MDP
extern "C" void InitalizeDomainOnHost(int stateCount,
                                      int actionCount,
                                      int divergenceCount);

//Sets the constants of the algorithm
//Parameters:
// gamma - algorithm discount rate
// convThreashold - the convergence threashold
extern "C" void SetValueIterationConstants(float gamma,
                                           float convThreashold);

//CUDA Initialize Domain on the GPU
//Parameters:
```

```
19 // useTextures - if true MDP dynamics are loaded in texture memory
20 extern "C" void InitializeDomainOnGPU(bool useTextures);
```

Listing 5.2: Initialization of domain and constants

- Application of algorithm: The framework contains functions for applying any of the implemented algorithms on the domain for either a predefined number of iterations or until it is solved.

```
1  //Do a predefined number of iterations
2  //Parameters:
3  // numberOfIterations - number of iterations to be performed
4  // algorithmType - constant which determines the algorithm to be used
5  extern "C" void IterateOnDomain(int numberOfIterations, int algorithmType);
6
7  //Does computations on the domain until it is solved
8  //Parameters:
9  // algorithmType - constant which determines the algorithm to be used
10 extern "C" void SolveDomain(int algorithmType);
```

Listing 5.3: Applying algorithms

- Data Access: The framework contains functions which enable access to all of the arrays of the MDP representation.

```
1 //Getter functions for pointers
2 extern "C" float* getStateValuesPointer();
3 extern "C" float* getStateDeltasPointer();
4 extern "C" float2* getProbabilityAndRewardPointer();
5 extern "C" int2* getResultStatesPointer();
```

Listing 5.4: Data Access

The application of the framework is rather straight-forward and is probably best explained with a code example. Listing 5.5 shows a simple code example that demonstrates the usage of the framework. In lines 8 to 12 an MDPReader is declared and initialized with constants that describe the dimensions of the MDP that is to be read from file. The MDPReader is a utility class that is supplied with the framework and can read files containing MDP descriptions which have the syntax which is described in Appendix A. In line 14 the CUDA device is initialized. Line 16 through 18 contains the initialization of the domain on the host and then the MDPReader is used to load the domain from file into host memory in lines 20 to 23. To load the domain from file the MDPReader is supplied the file name, a pointer to the arrays which the MDP is to be loaded into and a constant which describes the file format that is to be read. In line 25 the constants for the value iteration algorithm are set and in line 27 the domain is initialized on the CUDA device and takes in a parameter that determines whether the dynamics of the domain are loaded into texture memory rather than global memory. In line 29 the MDP is then solved with the algorithm

determined by the parameter. In line 31 the state values of the solved domain are then set as a parameter into a print function which prints the results on screen. Finally the CUDA device is then uninitialized in line 33.

```
1
2  #include "MDP_Framework.h"
3  #include "MDPReader.h"
4
5  /** Example of the usage of the GPU Based MDP Solver Framework **/
6  //The following constants are used in the example
7  //ENV_STATE_COUNT = number of states in the MDP
8  //ENV_MAX_NUM_ACTIONS = maximum number of actions per state
9  //ENV_MAX_DIVERGENCE = maximum number of successor states per action
10 //DOMAIN_FILE = path to the file containing the MDP description
11 //mdpReader.TEXT = the file that is read is in text format, mdpReader.XML
12 //                                  is used for XML files
13 //BLOCKVALUE_ITERATION = The BDI algorithm is to be used for solving the domain,
14 //                        RESULT_ITERATION can be used for the RDI algorithm.
15
16 int main(int argc, char** argv)
17 {
18   MDPReader mdpReader;
19   mdpReader.InitializeReader(ENV_STATE_COUNT,
20                                                          ENV_MAX_NUM_ACTIONS,
21                                  ENV_MAX_DIVERGENCE);
22
23   InitializeCUDA();
24
25   InitalizeDomainOnHost(ENV_STATE_COUNT, ENV_MAX_NUM_ACTIONS,
26                                                ENV_MAX_DIVERGENCE);
27
28   mdpReader.LoadDomain(DOMAIN_FILE, getStateValuesPointer(),
29       getProbabilityAndRewardPointer(),
30           getResultStatesPointer(), mdpReader.TEXT);
31
32   SetValueIterationConstants(GAMMA, CONVERGENCE_THRESHOLD);
33
34   InitializeDomainOnGPU(false);
35
36   SolveDomain(BLOCKVALUE_ITERATION);
37
38   printMDPValues(getStateValuesPointer());
39
40   DeInitializeCUDA();
41
42   return 0;
43 }
```

Listing 5.5: Example usage of the CUDA enabled MDP Solver framework

5.2.1 Representation of an MDP within the Framework

Within the framework the representation of the MDP is kept as compact as possible in an effort to efficiently utilize the memory of the CUDA device. The four-tuple representation of the MDP, see Section 2.1, is broken into four different arrays:

- StateValues:
  Array of floating point numbers containing the value of each state. The size of the array is equal to the number of states.
- DeltaValues:
  Array of floating point numbers containing the delta value of each state. The size of the array is equal to the number of states.
- ProbabilityAndRewards:
  Array of two floating point numbers (float2) which contains the probability of each successor state and the resulting reward. The size of the array is equal to the number of all possible successor states of all states.
- ResultStates:
  Array of two integer number (int2) which contains the identifier of a corresponding successor state and whether or not it is within the computational block. The size of the array is equal to the number of all possible successor states of all states.

MDPs must be stored this way in memory to be solved with the framework. The MDPReader that is supplied with the framework can be used to load MDP descriptions from file into memory so they can be solved directly by the framework.

5.2.2 Additional Tools

To accommodate the framework a file reader was implemented for the loading of files containing MDP descriptions. The file reader can process both descriptions of MDPs in XML format and text format. A custom syntax for both of these formats was developed and it can be seen in Appendix A. An MDP generator was also implemented for the testing of the framework and algorithms.

5.3 Summary

In this chapter we presented the GPU Based MDP Solver Framework. We gave an overview of how it is structured, how it requires the data to be structured, and gave an brief code example of its usage. Next we present the results of our empirical evaluation.

Chapter 6

Empirical Evaluation

In this chapter we present the results of our empirical evaluation. The first section describes the Experimental setup used for the evaluation of the algorithms. The second section gives the results of a performance comparison of the parallel GPU-based methods described in Chapter 3 and a sequential CPU-based implementation of the Value Iteration algorithm. The third section describes the effect of localization on the performance of the parallel algorithms. Then we analyze the result in an effort to better understand what are the main factors influencing the performance of the algorithms.

6.1 Experimental Setup

To measure the performance of the algorithms, several MDPs of different dimensions and complexities were generated. All of the MDPs have the characteristics of a Grid-world based domain, such as a high locality and a low number of actions, which is a feature common to many reinforcement learning domains. High locality is also an inherent feature of MDPs created by discretization of domains with real value features such as Mountain car (Sutton & Barto, 1998). The reward states of the MDPs were generated randomly and contain a randomly generated reward within the predetermined range of 2 to 20. There are three types of domain dynamics for each differently sized MDP:

- Deterministic: each action has a single possible successor state.
- Two possible outcomes: each action has two possible successor states, one with 90% possibility of occuring and the other one 10%.
- Four possible outcomes: four possible successor states per action where one state has a 70% chance of occuring while the others have a 10% possibility each.

Table 6.1: Characteristics of the MDPs used for performance evaluation

Domain Name	**Number of States**	**Actions per State**	**Successors per Action**	**Reward States**	**Computational Cost**
GW-64x64x1	4,096	4	1	5	16,384
GW-256x256x1	65,536	4	1	65	262,144
GW-512x512x1	262,144	4	1	260	1,048,576
GW-1024x1024x1	1,048,576	4	1	1,024	4,194,304
GW-64x64x2	4,096	4	2	5	32,768
GW-256x256x2	65,536	4	2	65	524,288
GW-512x512x2	262,144	4	2	260	2,097,152
GW-1024x1024x2	1,048,576	4	2	1,024	8,388,608
GW-64x64x4	4,096	4	4	5	65,536
GW-256x256x4	65,536	4	4	65	1,048,576
GW-512x512x4	262,144	4	4	260	4,194,304
GW-1024x1024x4	1,048,576	4	4	1,024	16,777,216

The dynamics of the non-deterministic versions of the MDPs are inspired by the Wetfloor domain (Bonet & Geffner, 2006). For the evaluation of the results we also define for the MDPs the concept of *computational cost* which is the number of states multiplied by the number of actions per state multiplied by the number of possible successor states per action. Our largest MDPs have state dimensions of 1024x1024 states. We were unable to use a larger dimension of 2048x2048 as the memory requirements of an MDP of those dimensions with four possible successor states per action exceeded the available memory on the device. Table 6.1 summarizes the properties of the test domains.

Each algorithm was executed until it converged and timed twelve times for each MDP listed in Table 6.1. The best and the worst times were then eliminated from the results and the remaining results averaged to obtain the execution time. As a baseline for the performance measurements, a sequential Value Iteration algorithm was implemented and executed on a CPU.

All the algorithms were executed with the same parameter values. All the algorithms had a γ value of 0.9 and a θ value of 0.0001. The GPU algorithms also had their sweep count parameter set to 8. The BDI algorithm uses a block size of 16x8 for a total of 124 threads per block, while the RDI algorithm uses a block size of 256x1 for a total of 256 threads per block.

The timing and execution of the CUDA code was conducted on a PC running Windows XP with 2 GBs of RAM, a 2.4 GHZ Pentium IV Processor and a single NVIDIA GeForce 8800 GTX GPU with 768 MB of RAM, and 8 multiprocessors which combine to a total

of 128 stream processors. For the CPU implementation a PC was used with 3.5 GBs of RAM, a 2.4 GHz Intel Core 2 Duo Pentium processor and running Windows XP. The CUDA applications were implemented using CUDA 2.1, C and C++, while the CPU implementation was written in C++.

6.2 Speedup

Performance is an important issue when it comes to solving MDPs, as their size can quickly grow very large. In this section we present the results of our algorithms' performance in comparison to a sequential implementation executed on a CPU.

For the performance measurements the algorithms were evaluated using the twelve MDPs shown in Table 6.1. For each algorithm the execution time and the number of iterations required for convergence was measured. Figure 6.1 shows the execution time, in milliseconds, of the three algorithms. The test domains are ordered by computational cost on the x-axis. The Figure clearly shows that the parallel algorithms outperform the sequential implementation and that the performance gap between the algorithms quickly increases as the computational cost of the domains grows.

In Figure 6.2 the execution time is shown on a logarithmic scale to give us a clearer picture of how it develops in relation to the computational cost. The gap between the sequential method and the BDI algorithm is considerable from the beginning and slowly widens as the computational cost increases. For the RDI algorithm the execution time is slightly worse in the beginning than that of the sequential method. As the computational cost grows, however, we see a significant improvement in execution time for the RDI algorithm and the gap between it and the sequential method quickly widens. For the more costly MDPs the RDI algorithm has almost closed the gap between it and the BDI algorithm.

Figure 6.3 has the domains ordered by computational cost on the x-axis and number of iterations required for convergence on the y-axis. There we see that the parallel methods require a greater number of iterations to converge. For all algorithms there seems to be a relationship between the dimensions of the domain and the number of iterations needed. For the sequential and the RDI algorithm there is a slight increase in the number of iterations needed as the dimensions grow, but the number of iterations needed reduces as the number of successor states grows. The change in number of iterations is more subtle in the RDI algorithm than the sequential one. Contrary to the other two algorithms the number of iterations needed for the BDI algorithm to converge grows as the number of

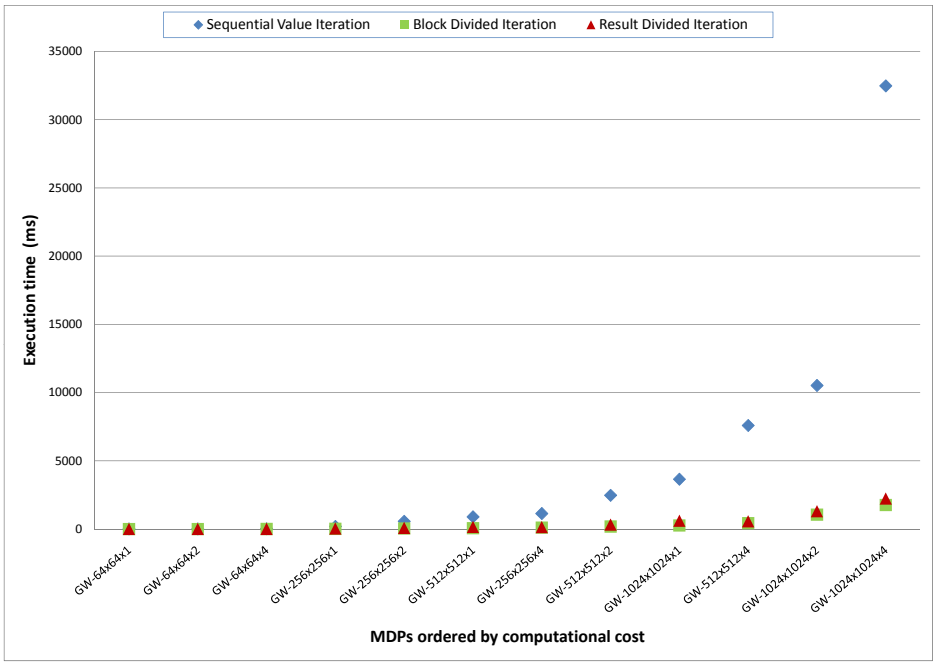

Figure 6.1: Comparison of solving time of MDPs

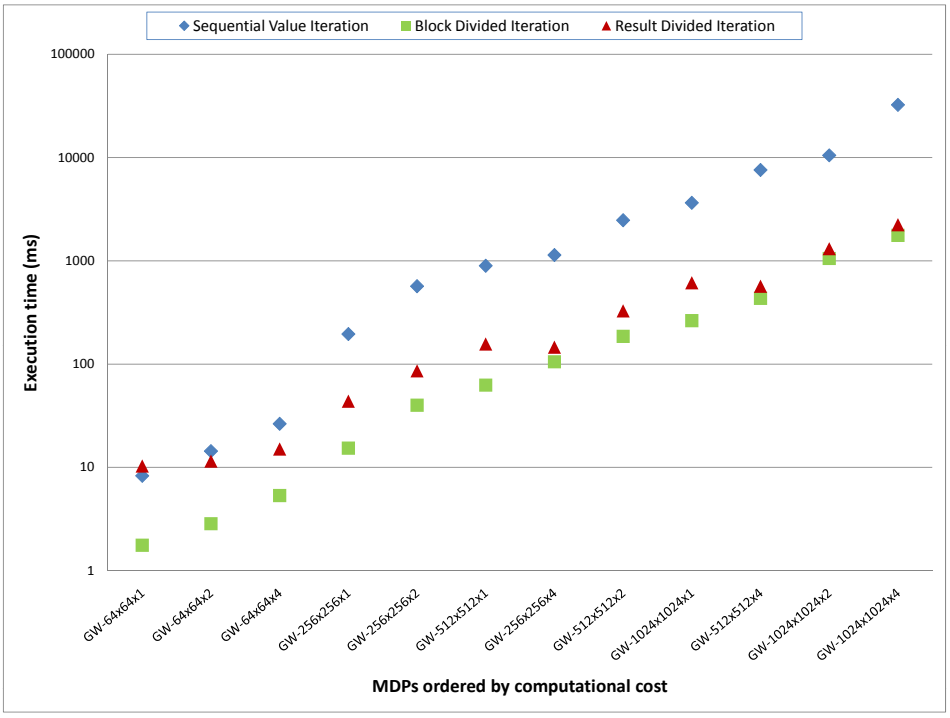

Figure 6.2: Comparison of solving time of MDPs with logarithmic time scale

successor states increases and its change in number of iterations between MDPs is much larger. It is, however, important to keep in mind that the location and the reward of the

reward states are random between domains of different dimensions and their dynamics could also have some influence on the number of iterations needed.

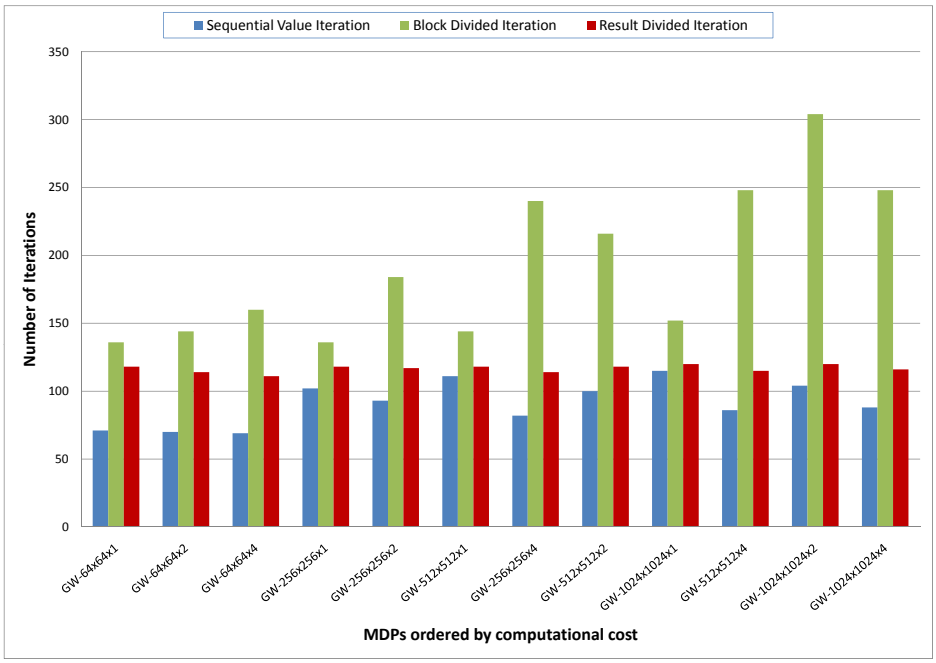

Figure 6.3: Comparison of convergence rate

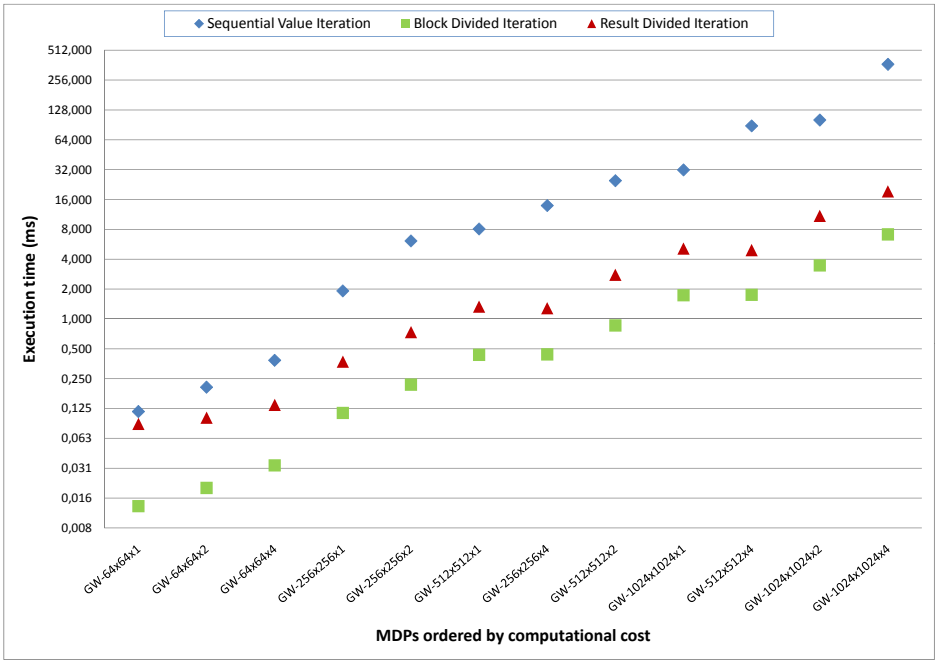

Figure 6.4: Comparison of time per state space sweep with logarithmic time scale

Table 6.2: Speedup of parallel algorithms in relation to sequential algorithm

MDP		B.D.I		R.D.I	
Name	Comp. Cost	Conv.	Sweep	Conv.	Sweep
GW-64x64x1	16,384	4.7x	9.0x	0.8x	1.3x
GW-64x64x2	32,768	5.0x	10.4x	1.2x	2.0x
GW-64x64x4	65,536	5.0x	11.5x	1.8x	2.8x
GW-256x256x1	262,144	12.7x	17.0x	4.5x	5.2x
GW-256x256x2	524,288	14.2x	28.1x	6.6x	8.3x
GW-512x512x1	1,048,576	14.3x	18.6x	5.7x	6.1x
GW-256x256x4	1,048,576	10.8x	31.6x	7.8x	10.8x
GW-512x512x2	2,097,152	13.3x	28.8x	7.5x	8.9x
GW-1024x1024x1	4,194,304	13.9x	18.3x	6.0x	6.2x
GW-512x512x4	4,194,304	17.5x	50.5x	13.4x	17.9x
GW-1024x1024x2	8,388,608	10.0x	29.2x	8.0x	9.3x
GW-1024x1024x4	16,777,216	18.4	51.8x	14.5x	19.1x

If we take a look at the time it takes the algorithms to do a single sweep through the state space it is no surprise that the parallel algorithms also outperform the sequential one in this area. The computational cost of the MDPs grows by a factor of 512 from the simplest to the most costly MDP. The sweep time of the sequential method, however, grows by a factor of approximately 868 from the MDP with the least computational cost to the MDP with the greatest computational cost. If we look at the parallel methods, on the other hand, the sweep time of the BDI algorithm grows only by a factor of approximately 268 and the RDI algorithm by a factor of approximately 125. This can be clearly seen in Figure 6.4, which has the domains ordered by computational cost on the x-axis and execution time in milliseconds with logarithmic scaling on the y-axis. As the graph has logarithmic scaling we can see how the execution time gap widens between the sequential and the parallel methods while it narrows between the BDI and RDI algorithms.

Table 6.2 and Figure 6.5 shows the speed up for both convergence and sweep time for both parallel algorithms in relation to the sequential algorithm. In the figure we have the domains ordered by computational cost on the x-axis and the speedup relative to the sequential implementation on the y-axis. The BDI algorithm shows the better performance by reducing the execution time on average by a factor of 11.7 while the RDI algorithm only manages an average speedup of 6.5. It is also noticeable that the time it takes the algorithms to do a single sweep through the state space greatly reduces in relation to the sequential algorithm as the computational cost of the MDPs increases. For the RDI algorithm, the speedup of the convergence and sweep time is relatively consistent. For the

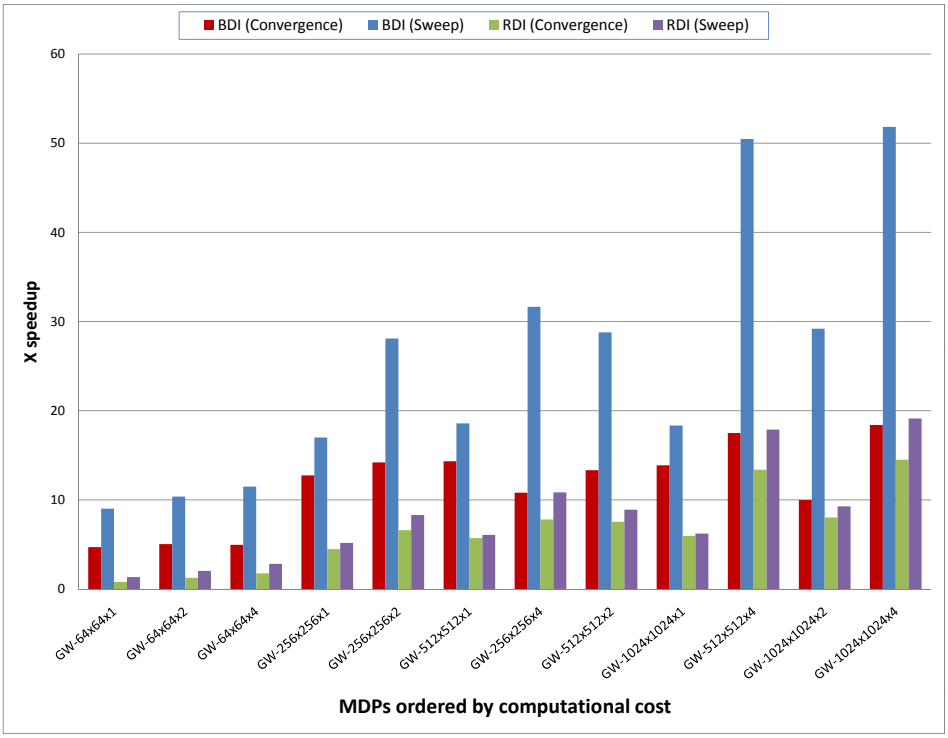

Figure 6.5: Achieved speedup in comparison to the sequential implementation

BDI algorithm, however, the convergence speedup does not increase nearly as rapidly as the sweep time.

6.3 Effects of Localization on Performance

If the algorithms are to be applicable to a wide range of MDPs of different characteristics it is important that their performance is consistent and that they are not dependent on some special characteristics of MDPs for acceptable performance. To determine how locality of the domains affects the performance of the algorithms we used a deterministic MDP with 65,536 states and four actions. We then modified the domain in such a way that a predetermined percentage of the successor states were changed to a random state within the state space. By introducing these actions the locality of the MDP is affected. We used the same domain with five different percentages of random successor states: 5%, 10%, 15%, 30%, and 50%.

In Figure 6.6 the results from the locality experiment is shown, is has the percentage of random actions within the MDP on the x-axis and the number of iterations required for convergence on the y-axis. For all of the algorithms, the locality seems to have an insignificant effect on the number of iterations required for the algorithms to converge.

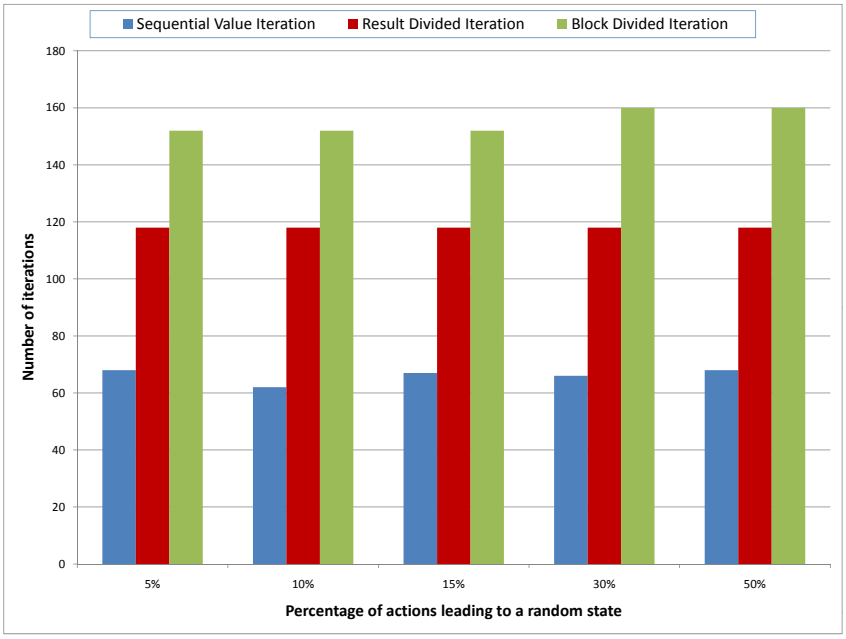

Figure 6.6: Comparison of number of iterations required for MDPs of different locality

The number of iterations vary slightly for the sequential and the BDI algorithm, while the RDI algorithm converges with the same number of iterations for each domain.

It is not surprising that the locality of the domains has little effect on the sequential and the RDI algorithm as they compute the value of each state from values of successor states obtained from the current or the previous sweep. It is interesting, however, to see that the locality seems to have little effect on the number of iterations the BDI algorithms requires to converge, though a large part of the states will be backed up with values which can be several sweeps old, since values of successor states which reside outside each block are loaded from global memory and the global memory is only synchronized with the shared memory at a predetermined interval defined by the sweep count parameter. We can therefore assume that the slow propagation of values because of the block division is having a more profound effects on the convergence rate of BDI than domain locality.

6.4 Analysis of Results

As can be seen from the results, both parallel algorithms perform considerably better than the sequential one when it comes to the time it takes for them to converge. Their speedup, however, is not as significant as one might expect considering how well this approach of solving MDPs maps to CUDA and the computational power of modern GPUs. In the case of our parallel algorithms there are several factors influencing the overall speedup.

If we look at the speedup gain for a single sweep through the state space we notice that for the BDI algorithm the sweep time speedup grows rapidly as the computational cost increases and achieves an average speedup of 25.5; the same goes for the RDI algorithm which achieves an average speedup of 8.2. The main cause of this can be seen if we divide the sweep time by the computational cost as shown in Figure 6.7. For the sequential algorithm the time per computational cost unit increases with additional computing cost, while for the BDI and the RDI algorithm it decreases and almost levels out as the computational cost grows. The difference in time between the two parallel algorithms can be traced back to their different nature and usage of computational resources. The BDI algorithm makes use of shared memory and only performs the convergence check on a predetermined interval. The RDI algorithm, on the other hand, is unable to make use of shared memory and does a convergence check on each sweep.

If we input the register and shared memory usage and the dimensions of the block size into the occupancy calculator supplied by NVIDIA we get that for our particular configurations we should be getting a 67% occupancy for the BDI algorithm and 100% occupancy for the RDI algorithm. As this is contrary to what we are seeing in the sweep speedup, where the RDI algorithm is only achieving less than half of what the BDI algorithm is achieving, it strongly suggests that our RDI algorithm is bound by memory latency and that the kernel does not have enough arithmetic instructions to hide the latency.

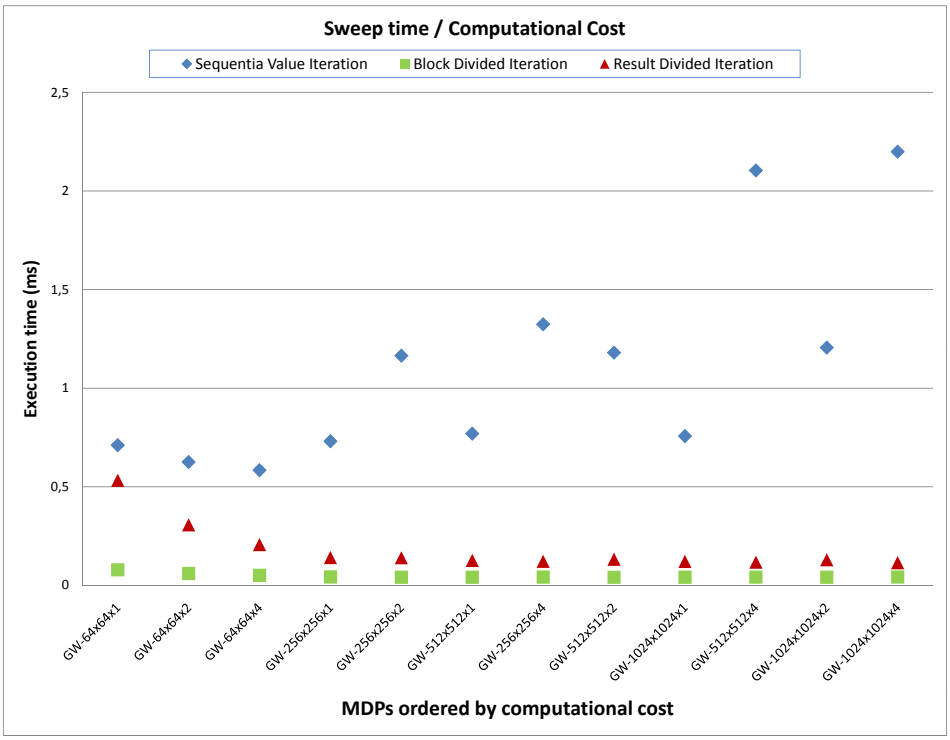

Figure 6.7: Comparison between methods of time per computational cost unit

The speedup of the sweep time is, however, not fully passed on to the convergence time speedup as the parallel algorithms require a greater number of sweeps through the state space in order to converge. In case of the RDI algorithm the additional sweeps required for it to converge are caused by a disadvantage of the parallelization, since if we compute the value of a collection of states in parallel which values are dependent on each other, those computations will all be based on values from the previous sweeps, while in the case of the sequential method the new value of those states will be based on values from this sweep as well as the previous one. This causes the RDI algorithm to follow a similar pattern as the sequential algorithm in the number of iterations needed. The BDI algorithm, however, seems to follow a different path when in comes to the number of iterations required. As could been seen in Figure 6.4 the number of iterations needed seems to be related to the dimensions of the MDP, or probably, to be more accurate, related to the number of blocks; the number of blocks increases with growing dimensions since their size is constant. The more blocks we have the more iterations are needed for convergence. The cause of this can be clearly seen if we think about a large grid world domain with only two reward states with high rewards at opposite corners. For the BDI algorithm the value of the shared memory is only synchronized with the global memory at an interval defined by the sweep count parameter, which means that the values of those corner states are only propagated between blocks at an interval equal to the sweep count parameter. In the case of the parameter configuration of our experiment it will thus take 8 sweeps for the value to propagate between blocks. Consequently, this influences the convergence rate for MDPs which are divided into many blocks and have states with high reward values which influence the value of states within multiple blocks.

6.5 Summary

In this chapter we evaluated the performance of the algorithms in practice. Both of the parallel algorithms show considerable improvement in convergence time over the sequential method. Both of the parallel algorithms seem to be better fit for the solving of larger MDPs as their relative speedup increases with additional computational cost. If we only look at MDPs with greater computational cost than 2 million, the average speedup grows from 11.7 to 14.6 for the BDI algorithm and from 6.5 to 9.9 for the RDI algorithm. The parallelization does come at the cost of a slower convergence rate, which has a more profound effect on BDI than RDI and is the main cause for not achieving a more significant speedup. We also evaluated the effects of localization on the convergence rate of the algorithms, and found it to be insignificant.

Chapter 7

Conclusion

MDPs play a significant role in a wide variety of fields such as robotics, automated control, and planning. The ability to solve them quickly and efficiently is thus important. In this thesis we have presented a framework for solving MDPs on CUDA devices and introduced two parallel implementations of the Value Iteration algorithm: Block Divided Iteration and Result Divided Iteration. The framework enables researchers to easily utilize their CUDA device for the solving of MDPs from within in C or C++ applications.

The method of applying Value Iteration for the solving of MDPs lends it self well to the CUDA architecture as it consists mainly of independent calculations which can be parallelized. We take two different approaches to the mapping of the Value Iteration algorithm to the CUDA programming model. The first approaches results in the BDI algorithm which is focused on parallelization on a state level, while the latter results in the RSI algorithm and involves a much more fine grained parallelization of subtasks.

We used the framework to solve several different MDPs and evaluated the performance of the two parallel algorithms in comparison to a baseline implementation running on a single CPU. Our results showed that the parallel algorithms run significantly faster, although they required a greater number of iterations to converge. More specifically the RDI algorithm achieved an average speedup of 8.2 in comparison to the baseline to perform a single sweep over the state space while the BDI algorithm achieved a speedup of 25.5. This speedup, however, is not fully carried into the speedup of convergence time and is reduced to a factor of 6.5 for the RDI algorithm and 11.7 for the BDI, because of the slower convergence rate.

Two factors were identified which have an effect on the overall performance of the algorithms. The most limiting factor for the RDI algorithm is that the arithmetic intensity of each thread is too low and the device is therefore unable to hide all the memory latency

of accessing the global memory. The BDI algorithm, however, suffers from slow propagation of values through the state space as a result of the division of the state space into relatively small blocks compared to the overall size of the domain. This causes the BDI algorithm to require a greater number of iterations to converge, which adversely affects its convergence time.

The most limiting factor of our approach is the requirement of the MDPs to be explicitly represented, but this is necessary for the framework to be able to solve any MDP without the need for any additional code to be written. As a result of this limitation, the largest MDPs we used for our experiments had state dimensions of 1024x1024 for a total of 1,048,576 states, four actions, and four successor states per action. As we moved to a larger MDP with state dimensions of 2048x2048, for a total of 4,194,304 states with four actions and four successor states, we were unable to load the domain onto the device as its memory requirements exceeded the device's available memory. This is one of the drawbacks of using the GPU compared to the CPU, as all memory usage must be monitored and handled by the programmer. In our case this could be solved by implementing functionality into the framework which executes on the host and partitions MDPs, whose size is above a certain threshold, into smaller partitions. Those smaller partitions would then be assigned to the CUDA device for execution.

For future work there are several additional evaluations and improvements that are of interest. The MDPs we used for our evaluation all have similar characteristics. To be able to better evaluate the methods and understand their behavior we need to apply them to a wider range of MDPs and preferably from various problem domains. In relation to a wider variety of MDPs it would be interesting to analyze in more detail the effects of different block sizes and parameter settings on the performance of the algorithms. It could also be interesting to see how a hybrid between the BDI and the RDI algorithm would perform where each action would be assigned to a thread. It would offer the opportunity to use shared memory but be more fine grained than the BDI algorithm.

Within the field of AI there is an increasing demand for faster ways to solve a variety of problems, for some of which CUDA might just be the right answer. CUDA is still relatively new and is evolving at a rapid pace and with each new release the computational abilities of the devices grow and it becomes easier to harness their computational power. In this thesis we addressed only one of these challenges and were able to achieve good speedup. We therefore look forward to applying our knowledge of CUDA gained from this project to other problems within the field.

It is becoming increasingly more difficult to increase the performance of modern CPUs by placing more and more transistors per square inch, but that has been one of the main

driving forces for the increase in computational capability of CPUs over the last decades. Instead we are seeing CPUs with multiple cores capable of parallel processing. As a response to recent developments in GPU computing and advances of the CPU towards parallel processing, an open standard called OpenCL has been introduced. OpenCL is a framework for writing programs, that can take advantage of both task-based and data-based parallelism executing across heterogeneous platforms consisting of CPUs, GPUs, and other processors. With OpenCL a vendor independent standard has been introduce, which is an important step since it enables users to write code for parallel execution independent of execution device. In our opinion parallel processing will without a doubt play an important role in the future development of computing.

Bibliography

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*, 81-138.

Bertsekas, D. P. (1982). Distributed dynamic programming. *IEEE Transactions on Automatic Control*, *27*, 610-616.

Bleiwiess, A. (2008). GPU accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (p. 65-74).

Bonet, B., & Geffner, H. (2006). Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *ICAPS '06 : Proceedings of the 16th International Conference on Automated Planning and Scheduling* (p. 142-151).

Edelkamp, S., Jabbar, S., & Bonet, B. (2007). External memory value iteration. In *ICAPS '07 : Proceedings of the 17th International Conference on Automated Planning and Scheduling* (p. 128-135).

Feinberg, E. A., & Shwartz, A. (2002). *Handbook of Markov decision processes: methods and applications*. Springer.

Harish, P., & Narayanan, P. (2008). *Accelerating large graph algorithms on the GPU using CUDA* (Tech. Rep.). Center for Visual Information Technology.

Katz, G. J., & Jr, J. T. K. (2008). All-pairs shortest-paths for large graphs on the GPU. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics hardware* (p. 47-55).

Kushida, M., Takahashi, K., Ueda, H., & Miyahara, T. (2006). A comparative study of parallel reinforcement learning methods with a PC cluster system. In *IAT '06: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology* (p. 416-419).

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. In *Machine Learning* (p. 103-130).

NVIDIA CUDA programming guide (2.1 ed.) [Computer software manual]. (2008, 8).

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., et al. (2005). A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, *1*, 21-51.

Pashenkova, E., Rish, I., & Dechter, R. (1996, 4). Value iteration and policy iteration algorithm for Markov Decision Problem. In *AAAI '96: Workshop on Structural Issues in Planning and Temporal Reasoning*.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & W. Hwu, W. mei. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (p. 73-82).

Sutton, R. S. (1997). On the signifcance of Markov Decision Processes. In *ICANN '97: Proceedings of the 7th International Conference on Artificial Neural Networks* (p. 273-282).

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT Press.

Vineet, V., & Narayanan, P. (2008). *CUDA cuts: Fast graph cuts on the GPU* (Tech. Rep.). Centre for Visual Information Technology.

Wikipedia. (2008). *Amdahl's law — wikipedia, the free encyclopedia.* Available from `http://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=285081003` ([Online; accessed 10-May-2009])

Wikipedia. (2009). *Parallel computing — wikipedia, the free encyclopedia.* Available from `http://en.wikipedia.org/w/index.php?title=Parallel_computing&oldid=287909970` ([Online; accessed 10-May-2009])

Wingate, D., & Seppi, K. (2003). Efficient value iteration using partitioned models. In *ICMLA '03: Proceedings of the International Conference on Machine Learning and Applications* (p. 53-59).

Wingate, D., & Seppi, K. D. (2004). P3VI: a partitioned, prioritized, parallel value iterator. In *ICML '04: Proceedings of the twenty-first international conference on machine learning* (p. 109).

Wingate, D., & Seppi, K. D. (2005). Prioritization methods for acceleraing MDP solvers. *Journal of Machine Learning Research*, *6*, 851-881.

Appendix A

MDP Input File Formats

This appendix contains the format for the input files supported by the MDPReader, which is a reader for files containing MDP descriptions implemented for this project. The MDPReader supports two kinds of input formats: XML and plain text. Both of the formats have similar structure and differ mainly in syntax. The XML format however imposes a much larger overhead and is therefore less practical for large MDPs. The syntax for the XML representation is self-explanatory and is best described by an example. The syntax for the plain text format is as follows. Each state starts with the letter S, following the S are three colon-separated numbers which describe the state's id, initial value, and number of actions available in that state. Following in the lines below each S are the description for the state's actions. Each action begins with the letter A and is followed by two colon-separated numbers which describe the id of the action and the number of possible divergences for that action. In the lines below each action there is a D followed by three colon-separated numbers for the description of each possible divergence for that action. The colon separated number of the divergence describes the id of the successor state, the probability of the divergence and finally the resulting reward.

A.1 XML Format example

```
<?xml version="1.0" encoding="utf-8"?>
<MDP domain="Grid World" author="Arsaell Johannsson">
  <Description>
     A standard 64x64 Grid World Domain 4096 states
  </Description>
  <Actions>
    <Action id="0" description="Move up"/>
    <Action id="1" description="Move right"/>
    <Action id="2" description="Move down"/>
```

```
10      <Action id="3" description="Move left"/>
11    </Actions>
12    <States>
13       <State id="0">
14         <AvailableActions>
15            <Action id="0">
16             <ResultingState id="0"
17                             probability="1"
18                             actionReward="0"/>
19            </Action>
20            <Action id="1">
21             <ResultingState id="1"
22                             probability="1"
23                             actionReward="0"/>
24             </Action>
25             <Action id="2">
26              <ResultingState id="64"
27                              probability="1"
28                              actionReward="0"/>
29             </Action>
30             <Action id="3">
31              <ResultingState id="0"
32                              probability="1"
33                              actionReward="0"/>
34             </Action>
35         </AvailableActions>
36      </State>
37       ...
38       ...
39       ...
40      <State id="4095">
41         <AvailableActions>
42             <Action id="0">
43               <ResultingState id="4094"
44                              probability="1"
45                              actionReward="0"/>
46              </Action>
47              <Action id="1">
48               <ResultingState id="4095"
49                              probability="1"
50                              actionReward="0"/>
51              </Action>
52              <Action id="2">
53               <ResultingState id="4095"
54                              probability="1"
55                              actionReward="0"/>
56               </Action>
57               <Action id="3">
58                <ResultingState id="4095"
59                              probability="1"
60                              actionReward="0"/>
61               </Action>
62          </AvailableActions>
63          </State>
64    </States>
65  </MDP>
```

Listing A.1: Example of MDP representation in XML format

A.2 Plain Text Example

```
S:0:0:4
A:0:1
D:0:1:0
A:1:1
D:64:1:0
A:2:1
D:1:1:0
A:3:1
D:0:1:0
S:1:0:4
A:0:1
D:0:1:0
A:1:1
D:65:1:0
A:2:1
D:2:1:0
A:3:1
D:1:1:0
...
...
...
S:4095:0:4
A:0:1
D:4094:1:0
A:1:1
D:4095:1:0
A:2:1
D:4095:1:0
A:3:1
D:4031:1:0
```

Listing A.2: Example of MDP representation in plain text format

Appendix B

Sequential Implementation

This appendix shows the implementation of the baseline sequential Value Iteration algorithm. The implementation uses the same memory representation for the MDPs as the parallel algorithms, therefor, float2 and int2 structs are used for storing the transition dynamics of the MDPs.

```
while( delta >= CONVERGENCE_THRESHOLD)
{
  delta = 0.0f;
  for(int stateIndex = 0; stateIndex < ENV_STATECOUNT; stateIndex++)
  {
    maxValue = -99999.0f;
    oldValue = MDP_State_Values[stateIndex];
    for( int actionIndex = 0; actionIndex < actionCount; actionIndex++)
    {
      divValue = 0;
      newValue = 0;
      for( int divIndex = 0; divIndex < divCount;divIndex++)
      {
        int currentIndex = stateIndex +
            ((actionIndex * ENV_DIVERGENCE + divIndex) *
            ENV_STATECOUNT);

         divValue = MDP_ProbabilityAndReward[currentIndex].x *
         ( MDP_ProbabilityAndReward[currentIndex].y + GAMMA *
           MDP_State_Values[MDP_ResultStateIds[currentIndex].x]);

         newValue += divValue;
        }
        maxValue = max(newValue, maxValue);
    }
    delta = max(delta, abs(oldValue - maxValue));
    MDP_State_Values[stateIndex] = maxValue;
  }
}
```

Listing B.1: Implementation of the algorithm used as a baseline

REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is