Parallelization of the Value-Iteration algorithm for Partially Observable Markov Decision Processes

Dennis Noer

Kongens Lyngby 2013
DTU Compute-B.Sc.-2013-31

Technical University of Denmark
DTU Compute
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
compute@compute.dtu.dk
www.compute.dtu.dk DTU Compute-B.Sc.-2013-31

Summary (English)

Partially observable Markov decision processes (POMDP) is a strong framework for mission planning in a partially observable and stochastic environment. To determine the solution of a POMDP, algorithms with a running time in the PSPACE-Hard area are utilized. In this project, we will explore the potential for reducing the computation time using the massive parallel processing power of modern Graphic Processing Units (GPU). Our experiments show that the GPU platform can accelerate a PBVI based POMDP solver, while more advanced synchronization features are needed to exploit the full potential.

Summary (Danish)

Delvist observerbare Markov-beslutningsprocesser (POMDP) er et kraftfuld værktøj til missions planlægning, givet et delvist observerbar og stokastisk miljø. For at bestemme løsningen til et POMDP problem, benyttes algoritmer med køretid i det PSPACE-Hard område. Vi vil i dette projekt undersøge mulighederne for at reducere køretiden, ved hjælp af den massive parallelle beregnings kraft fundet på en Graphic Processing Unit (GPU). Vores exerimpenter viser at GPUen er i stand til at accelerere en PBVI baseret POMDP løser, dog er mere avancerede synkroniserings funktioner nødvendige, for at kunne udnytte GPUens fulde potentiale.

Preface

This thesis was prepared at the department of DTU Compute at the Technical University of Denmark in fulfilment of the requirements for acquiring an B.Sc. in Informatics.

The thesis represents a workload of 15 ETCS points during the period of April to August 2013. Associate professor Hans Henrik Løvengreen from DTU Compute has supervised the project.

The thesis deals with the parallization and implementation of a POMDP solver on a GPU platform.

The thesis consists of of a summery in English and Danish, 8 chapters on the subject, a bibliography, and an appendix.

Lyngby, 11-August-2013

Dennis Noer

Acknowledgements

I would like to thank Søren Bøg for help understaning the mathematical model of the POMDP solver.

I would like to thank Hans Henrik Løvengreen, Yousif Subhi, and Sara Næraa for feedback during the writing process.

Contents

Chapter 1

Introduction

An intelligent agent is needed for many problems where constant human interaction is inconvenient. This is the case in many areas ranging from autonomous robotics to artificial intelligence in entertainment products.

The classic intelligent agent operates in a cycle of sensing, planning, and executing [Stu09], where the agent uses sensors to make observations about its environment, and then update its internal state accordingly, which is used in the planning process. Finally it chooses how to interact with its environment. Most agents work in fully deterministic domain, where the given agent is certain of the resulting effect of the executed action, and together with a perfect perception, the agent is able to construct an optimal plan as a sequence of deterministic actions. Agents that operate in real world mission planning will often not have the perfect awareness, thus a more advanced model is needed.

Intelligent agents that operate in a state based environment and uphold the Markov property [Mar54], where the resulting state only depends on the current state and the executed action, can utilize the family of Markov models to determine an optimal policy. This depends on the ability to perceive the agent's current state and the consequences of its actions. Here, one of the following 4 models can be used:

	States is fully observable	States is paritally observable
Deterministic execution	Markov chains	Hidden Markov models
Stochastic execution	Markov decision process	POMDP

In this project we will work with agents that operate in a stochastic and partially observable environment because it is a strong model of the real world. We are not always able to determine the exact outcome of a given action e.g. when we switch on the lights we assume the room will be lit, which is almost always the case, but the outcome is not given, and thus we live in a stochastic environment. Furthermore, the ability to perceive our environment relies on our sensing abilities which can be severely limited or noisy, thus our world is partially observable. Given these parameters, we will utilize the POMDP framework to construct a plan.

To solve a POMDP for the resulting policy, a powerful dynamic programming algorithm is utilized, but the running time still resides in the PSPACE-Hard area.

To improve the running time of a problem, we can make improvements to the algorithm or the platform. Graphic Processing Units (GPUs) have been used to offload and accelerate vectorized calculations in high demanding computer games for decades. A new trend in High Performance Computing (HPC) exploits the massive parallel processing power of the GPU for general purpose problems. We are interested in utilizing the massive processing power of the GPU to improve the running time of a POMDP solver.

1.0.1 Problem Definition

The main problem of the POMDP model is that its running time is in the PSPACE-Hard area, where most algorithms are optimized for serial execution.

In this project, we will explore the possibilities for improving the running time of the POMDP model, by parallelization of a suited algorithm using a multicore architecture. The resulting implementation should be widely available and usable on "over the counter" hardware, thus a standard desktop computer will suffice for development and performance testing.

Chapter 2

Problem Analysis

In this section, we will look at the inner workings of the POMDP framework. Many different algorithms have been proposed to solve the POMDP problem. First, we will find one suited for parallel implementation. Next, we will look into the world of Graphic Processing Units, where we will explore the massive parallel architecture and the basis of the programming model.

2.1 Partially observable Markov decision processes

POMDP is a strong mathematical framework for agent planning. This section looks into how the model works in detail.

2.1.1 Agent Model

The POMDP framework can construct a plan for agents operating in a stochastic state machine. A simple example of an agent that operates a power supply is illustated in figure 2.1.

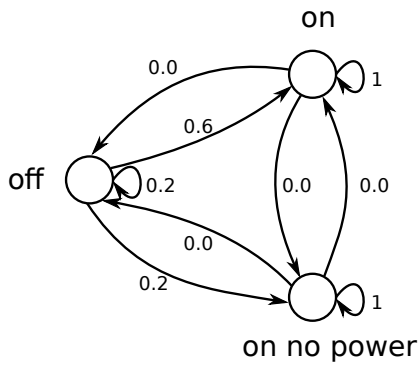

Figure 2.1: Stochastic State Machine

The agent can be in one of three states: on, off, and on_no_power. Figure 2.1 shows some of the actions that the agent can execute. Given the stochastic nature of the model, the result of executing an action is given by a probability distribution $p(s'|a, s)$. Furthermore the agent does not have perfect perception and can therefore not know its current pure state. However, using sensors to observe its environment, it can make a best guess on the probability of being in a certain state. This belief state is a vector where each element corresponds to the probability of being in a pure state, and it is given that a belief state must sum to 1.

The following sections will explore how this basic model can be used to determine an optimal plan for the agent to execute.

2.1.2 POMDP Framework

The POMDP framework models its environment as a tuple $< S, A, T, R, \Omega, O >$ where each element is defined as follows:

- S is the set of states the agent can be in.
- A is the set of actions the agent can execute.
- Ω is the set of observations the agent can receive.
- T is the transition function $T(s, a, s')$ and describes the probability of transition from state s to the resulting state s' when executing action a, which is declared as $p(s'|a, s)$. The uncertainty of an action's consequences is reflected in the probability distribution.

- R is the reward function $R(s, a)$ which describes the agent's reward for executing the action a in a given state s.
- O is the observation function $O(s', a, o)$ which describes the probability of a certain observation after executing an action a and reaching state s'. The uncertainty of the agent's current state is reflected in the probability distribution of the receivable observations in a given state, e.g given a very simple proximity sensor on a robot, an observation can be the number of walls to the sides of the robot. This information will probably not reveal the true position of the robot, but using the observation function a probability can be assigned to it.

2.1.3 Agent Operation

When the agent is executing it will run a cycle of sense-think-act which consists of: observing its environment, updating its state, and executing what it believes to be the best action.

Given that the agent operates in a partial observable environment, perfect perception of the current state is not possible. Therefore the inner state of the agent is defined as a probability distribution throughout the dimension S, and the space of this distribution is defined as the belief space and denoted Δ. Because all belief states must sum to 1, they will be bound to a $(|S| - 1) -$ dimensionalspace.

The initial state of the agent is represented by the belief state b_0 which can have any distribution e.g. a specific start state or the uniform distribution which represents the agent having no knowledge about its initial state.

When the agent has executed an action a and then received an observation o, it will update its internal belief state using the Bayes' rule given by the following formula:

$$b_a^o(s') = \frac{p(o|s', a)}{p(o|a, b)} \sum_{s \in S} p(s'|s, a) b(s) \tag{2.1}$$

- $\sum_{s \in S} p(s'|s, a) b(s)$ is the probability of ending in s' given the belief state b and action a.
- $p(o|a, b)$ is a normalizing factor that ensures that $b_a^o(s')$ sums to 1, and is given as: $p(o|a, b) = \sum_{s \in S} p(o|s', a) \sum_{s \in S} p(s'|s, a) b(s)$.
- $p(o|s', a)$ is simply the probability of making the observation o when executing action a in state s.

Through the agents lifespan the belief state b will contain all information about the agents past.

2.1.4 Policy

The goal of the agent is to execute the best possible action given a state, and this relation between state and action is contained in the agent's policy, which is denoted π.

Because the belief state is continuous through out the belief space, the policy π can not be described by a simple lookup table that maps a pure state to an action. Therefor we have to describe an action in the continuous belief space, using a function which is denoted the value function $V^{\pi} : \Delta \rightarrow \Re$. The value function defines the expected future reward attainable by the agent using the policy π from belief state b:

$$V^{\pi}(b) = E_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} \sum_{s \in S} r(s, \pi(b_t)) \cdot b(s) \mid b_0 = b\right] \tag{2.2}$$

The γ coefficient is the discount rate, which will ensure that the value function is bound to a finite sum. The discount rate must be between 0 and 1, and it is normally set close to 1.

The value function represents a number of functions, each given by a vector through the belief space. Figure 2.2 shows a simple example with two actions with the belief space in between, and the vectors in the value function are shown as the lines through the surface. Given a belief state, the best action is bound to the vector with the highest value.

The vectors in the value function resides in the $(|S| - 1) -$ dimensionalspace, where the surface defined by each vector will divide the n-dimensional space, which is the definition of a hyperplane. The combined set of hyperplanes $\left\{\alpha_n^i\right\}, i = 1...|V_n|$ will be piecewise linear and convex (PWLC) over the belief space. Each vector defines a region in the belief space where it maximizes V_n, which represents the best action $a(\alpha_n^i) \in A$ to execute given the belief state and its value of the value function $V(b)$ is the attainable future reward from that belief state.

Given a belief state b and a value function with the set of vectors $\left\{\alpha_n^i\right\}_{i=1}^{|V_n|}$ the

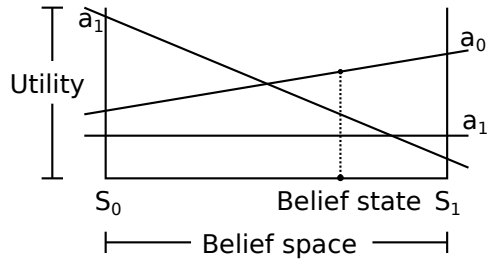

Figure 2.2: Value function example

attainable future reward is found by:

$$V_n(b) = \max_{\{\alpha_n^i\}_i} \left[b \cdot \alpha_n^i \right] \tag{2.3}$$

The best hyperplane is found by:

$$\alpha_n^b = arg \max_{\{\alpha_n^i\}_i} \left[b \cdot \alpha_n^i \right] \tag{2.4}$$

Our goal is to find a policy which maximizes the value function V^π. In the following section, we will go through the different types of algorithms used to solve a POMDP and determine the optimal policy π^*.

2.1.5 POMDP Algorithms

To determine the optimal policy π^*, we need to determine the corresponding optimal value function V^* by solving the POMDP. A classic approach is the value iteration algorithm [Bel57], in which the value iteration algorithm will approximate V^* using dynamic programming on the initial value function V_0 to generate the series $V_0, V_1, V_2 ... V^*$ through a number of iterations.

The value iteration algorithm first determines the value function for a horizon length of one step, which will simply be the immediate reward, and no future rewards are considered. In the next step, the value iteration algorithm will consider a solution with the horizon of two steps, which is convenient as the best solution leading to step two was recently calculated. Thus the value function is simply the reward so far plus the immediate reward.

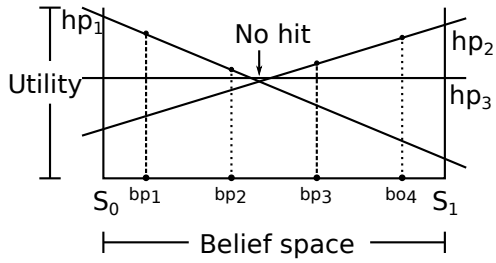

Figure 2.3: PBVI Pruning Example

The basic idea of the value iteration algorithm is to take a step into the future in each iteration and to use the accumulated utility as a base to determine the next action and its reward.

The value iteration algorithms used to generate the series of value functions generally fall into two categories:

1. Value iteration algorithms that makes all combinations of actions and observations to generate all possible hyperplanes. Most of the generated hyperplanes will have same solution or be dominated by others, which is the case for the lowest hyperplane in figure 2.2. Different methods are used to remove the useless hyperplanes, e.g. the simplex algorithm, which is used to prune the set of hyperplanes between each iteration, this way the minimal set of vectors will be maintained while still maintaining the exact solution. The simplex algorithm has a very high running time which is the cost for an exact solution.

2. Value iteration algorithms that uses a set of belief points from the belief space, all combinations is still generated, but the pruning is now only done in the belief points. Figure2.3 illustrate a PBVI pruning example, where all combinations have generated three hyperplanes, and the set of belief points is selected uniformly over the belief space. The pruning saves all hyperplanes that contribute to the maximum solution in the belief points, thus although hyperplane 3 is the best solution in a small area, with no belief point to sample it, it will be removed from the value function.

In this project the ultimate goal is a high performance implementation, thus algorithms with a very high running time are to be avoided, and furthermore given a base model where the agent is not even certain about its current state or the consequences of its actions, we argue that an approximative solution will suffice, which is why we will look further into POMDP solvers based on the Point Based Value Iteration (PBVI) algorithm described in [PGT03].

2.1.6 PBVI Solver

The PBVI algorithm uses a set B of points in the belief space which can be generated in three general ways:

1. Random initialization of B where the belief points are selected at random from the belief space.
2. Uniform distribution over the belief space, in witch the uniform generation starts by adding the pure states to the set. Then the distances between all belief points are found and the longest distance is divided by a new belief point. This will result in a uniform belief point net spanned over the belief space.
3. Simulated initialization, although the belief space is continuous allowing the agent state any distribution, the reachable belief states are given by the possible combinations of the state transition function and the observation function. By constructing sequences of actions and observations, the POMDP can be simulated using the belief state update function seen in equation 2.1. The points found by the simulation must be reachable by the POMDP and define a good estimator of the best belief points in which to update the value function. It is unlikely that all belief points will be visited.

The PBVI algorithm will start with a single hyperplane in the value function V_0. This vector will be initialized to $\frac{1}{1-\gamma}\min[R(s,a)]$, and this value is the minimum attainable reward of the POMDP and is guaranteed to be lower than V^* as shown in lemma 3 in [ZZ01].

The PBVI algorithm utilizes the set of belief points to update the value functions in the value iteration step. In each iteration, each belief point has its horizon incremented and its value improved and updated. The value in the belief point is the sum of the immediate reward attainable in the current belief state and the discounted future reward:

The immediate reward is the reward gained in a state weighted with the probability of being in the state.

$$\sum_{s \in S} b(s) r_a(s) \implies b \cdot r_a$$

The future discounted reward is defined by:

$$\gamma \sum_{o \in O} p(o|a,b) \cdot V_n(b_a^o)$$

b_a^o is the updated belief state given by equation 2.1, and the value in the selected belief point is weighted with the probability $p(o|a,b)$ of receiving the observation which was used to update the belief state e.g. future rewards with a zero chance of getting observed will not reward any utility. By summerizing all observations, we can generate all combinations of possible futures. Finally, the future reward is discounted with the γ coefficient, which will bound the future gain in utility and ensure a finite sum in the value function.

By combining the terms for the immediate and future reward, we can iterate the value function to V_{n+1} by finding the action that maximizes the sum of both immediate and future reward:

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} p(o|a,b) \cdot V_n(b_a^o) \right] \tag{2.5}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} p(o|a,b) \cdot \max_{\{\alpha_n^i\}_i} \left[\sum_{s' \in S} b_a^o(s') \cdot \alpha_n^i(s') \right] \right] \tag{2.6}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} p(o|a,b) \cdot \max_{\{\alpha_n^i\}_i} \left[\sum_{s' \in S} \frac{p(o|s',a)}{p(o|a,b)} \sum_{s \in S} p(s'|s,a) \cdot b(s) \cdot \alpha_n^i(s') \right] \right] \tag{2.7}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} \max_{\{\alpha_n^i\}_i} \left[\sum_{s' \in S} p(o|s',a) \sum_{s \in S} p(s'|s,a) \cdot b(s) \cdot \alpha_n^i(s') \right] \right] \tag{2.8}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} \max_{\{\alpha_n^i\}_i} \left[\sum_{s \in S} b(s) \sum_{s' \in S} p(o|s',a) p(s'|s,a) \cdot \alpha_n^i(s') \right] \right] \tag{2.9}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} \max_{\{g_{a,o}^i\}_i} \left[\sum_{s \in S} b(s) \cdot g_{a,o}^i(s) \right] \right] \tag{2.10}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} \max_{\{g_{a,o}^i\}_i} \left[b \cdot g_{a,o}^i \right] \right] \tag{2.11}$$

$$V_{n+1}(b) = \max_a \left[b \cdot r_a + \gamma \sum_{o \in O} b \cdot \arg \max_{\{g_{a,o}^i\}_i} \left[b \cdot g_{a,o}^i \right] \right] \tag{2.12}$$

$$V_{n+1}(b) = \max_a \left[b \cdot \left(r_a + \gamma \sum_{o \in O} \arg \max_{\{g_{a,o}^i\}_i} \left[b \cdot g_{a,o}^i \right] \right) \right] \tag{2.13}$$

$$V_{n+1}(b) = \max_{\{g_a^b\}_{a \in A}} b \cdot g_a^b \tag{2.14}$$

$$V_{n+1}(b) = b \cdot \arg \max_{\{g_a^b\}_{a \in A}} b \cdot g_a^b = b \cdot \alpha_{n+1} \tag{2.15}$$

Where $g_{a,o}^i$ and g_a^b is defined as:

$$g_{a,o}^i(s) = \sum_{s' \in S} p(o|s',a) p(s'|s,a) \cdot \alpha_n^i(s') \tag{2.16}$$

$$g_a^b = r_a + \gamma \sum_{o \in O} \arg \max_{\{g_{a,o}^i\}_i} \left[b \cdot g_{a,o}^i \right] \tag{2.17}$$

$$backup(b) = \arg \max_{\{g_a^b\}_{a \in A}} \left[b \cdot g_a^b \right] \tag{2.18}$$

Equation 2.5 is the combination of the immediate and future reward, where the value of $V_n(b_a^o)$ is the dot product of the best hyperplane in V_n and the belief point b_a^o, thus equation 2.6.

Using formula 2.1 we exchange b_a^o in equation 2.7 and elimination $p(o|a,b)$ in the expression 2.8.

Equation 2.9 is a simple rewriting of the terms which lets us extract $g_{a,o}^i$ seen in equation 2.16.

In equation 2.10 the sum over a vector product is simply the dot product as seen in equation 2.11.

Using the identity $\max_j b \cdot \alpha_j = b \cdot \arg\max_j b \cdot \alpha_j$ in equation 2.11, we are able extract g_a^b seen in equation 2.17.

The value iteration is then defined as the *backup*(b) operation, where the action which maximizes equation 2.17 in the given belief point b, will be the resulting hyperplane for the next iteration.

During the value iteration all belief point are independent of the other points in B, thus the PBVI value iteration algorithm is well suited for parallelization on a massive parallel architecture such as a GPU.

2.2 Graphic Processing Unit (GPU)

Using GPUs in high-performance computing is a relative new trend in High Performance Computing (HPC), where the GPU is used to accelerate computationally intensive tasks. This approach has enabled significant performance improvements due to the massive parallel calculation power. The GPU executes collections of threads based on the Single Instruction Multiple Data (SIMD), which is why the GPU is well-suited for algorithms that can exploit a high degree of data parallelism.

The programming model of GPUs is vendor specific standard based on the C language, of which the most notable are the CUDA toolkit [SK10, NVI13a] and the platform independent OpenCL standard [Khr13].

The CUDA programming model is the oldest and perhaps the most advanced, which is why we chose it for our development and implementation.

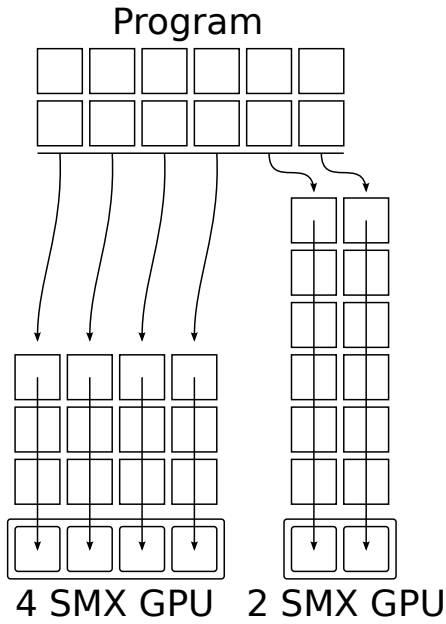

Figure 2.4: GPU Auto Scaling from [Noe11]

2.2.1 GPU Architecture

The GPU is an interesting platform for parallel processing on a massive scale because the architecture has thousands of computation cores and the bandwidth is close to an order of magnitude better than on CPUs. The many cores are divided in local processors called Streaming Multiprocessors (SMX).

CUDA programs are encapsulated in a kernel that is divided in a number of blocks. Each SMX is able to execute a block which has a number of assigned threads. The block size and threads per block is defined when calling the CUDA kernel. The diversion in blocks allow the platform to auto-scale a given problem as shown in figure 2.4, where the CUDA kernel is divided in 12 blocks. A device with four SMXs will use three iterations to execute the kernel, and a device with half the computing capacity will still be able to execute the same CUDA program, but will require twice the number of iterations.

2.2.2 Memory Hierarchy

The GPU platform offers a rich memory hierarchy, ranging from the biggest "global memory" to the smallest and fastest "shared memory". From CUDA version 4, the host and device shares a unified virtual address space, but transferring data will still use the CUDA programming interface, such as cudaMalloc() and cudaMemcopy() which will be explained in section 2.2.3.

Following memory types are available on the device:

- Global memory, which is the biggest memory ranging from hundreds of megabytes to several gigabytes. The global memory is of the GDDR type and therefore in the same performance rank as DDR RAM. GDDR RAM is placed off-chip which makes it accessible from both the host and all SMXs on the device.
- Constant memory, which is a small read only file of memory (48KB) found off-chip. The constant memory is cached and optimized for many threads accessing the same data at a given point in time.
- Shared memory, which is a small file of on-chip memory (48KB) found in each SMX. This type of memory is shared among all threads executing on a SMX. Shared memory and the SMX cache is taken from the same memory file, and the size of the division can be controlled when compiling the code. Given that shared memory has the same speed and latency as the SMX cache file, only algorithms with a simple memory access pattern will gain from using this memory.
- Register file. Each SMX has a small file of register memory (64KB) found on-chip in each SMX, and this type of memory is divided among all threads executing in a SMX and has the highest speed. If a program uses more registers than available, the data will spill to global memory, thus slowing down the execution.

Figure 2.5 shows the relation of the memories on the GPU:

NVIDIAs newest architecture is called Kepler and is explained in detail in [NVI13b].

2.2.3 Programming Model

NVIDIAs programming model is based on the C language. It supplies a programming interface used to control the device, some of which are explored in the following section.

```
#define N (4)
#define B (2)
// GPU code
__global__ void VecAdd(float* A, float* B, float* C) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  C[tid] = A[tid] + B[tid];
}

// Host code
int main(){
  float *devA,*devB,*devC;
  float *hostA,*hostB,*hostC;

  initializeData(hostA, hostB, hostC);

  cudaMalloc((void**) &devA, B*N*sizeof(float) );
  cudaMemcpy(devA, hostA, B*N*sizeof(float),
   cudaMemcpyHostToDevice);

  cudaMalloc((void**) &devB, B*N*sizeof(float) );
  cudaMemcpy(devb, hostB, B*N*sizeof(float),
   cudaMemcpyHostToDevice);

  cudaMalloc((void**) &devC, B*N*sizeof(float) );

  // Kernel call with B blocks of N threads.
  VecAdd<<<B, N>>>(A, B, C);

  cudaMemcpy(hostC, devC, B*N*sizeof(float),
   cudaMemcpyDeviceToHost);

  doStuff(hostC);
}
```

The example code shows the execution of a CUDA program, where each element

in vector A and B is summed in parallel, one thread per element. The program uses a number of CUDA API calls:

- Device memory allocation is seen in line (15, 18, 21), where cudaMalloc() will allocate a given amount of global memory on the device.
- Data transfer to the device is seen in line (16, 19), where cudaMemcpy() will transfer data between global memory and host memory. The direction is given by the last flag in the function. The transfer speed is limited by the PIC Express bus speed, which relative to the on-chip memory is an order of magnitude slower.
- The kernel execution is seen in line (25). The syntax describes the number of blocks and threads per block that the kernel will execute. The defined number of threads per block will all execute the CUDA kernel in parallel, and in this case 4 threads will execute the VecAdd() function at the same time, instruction by instruction, but possible on different data, thus SIMD. Given that execution of a CUDA program is SIMD, branching will cause the execution to be serialized and thus degrade performance.
- The CUDA kernel which will execute on the device is defined via the identifier __global__. Each thread has an index in its block and each block has an index on the grid. Using these indexes, the global index of any given thread can be found as shown in line (5). The grid of this specific program is seen in figure 2.6, which illustates two blocks each with 4 threads, of which each has a local index in its block given by threadIdx.x, and each block has an index on the grid given by blockIdx.x.

2.2.4 Speed-up Potential

Modern GPUs has a massive parallel architecture with raw calculation power in range of a few teraFLOPs, and with a memory speed in the range of 150-300 GB/s.

Assuming a modern CPU has raw calculation power in range of 60 GFLOPs and a memory speed of 30 GB/s, we should be able to improve the execution time by up to 100 times compared to a CPU implementation.

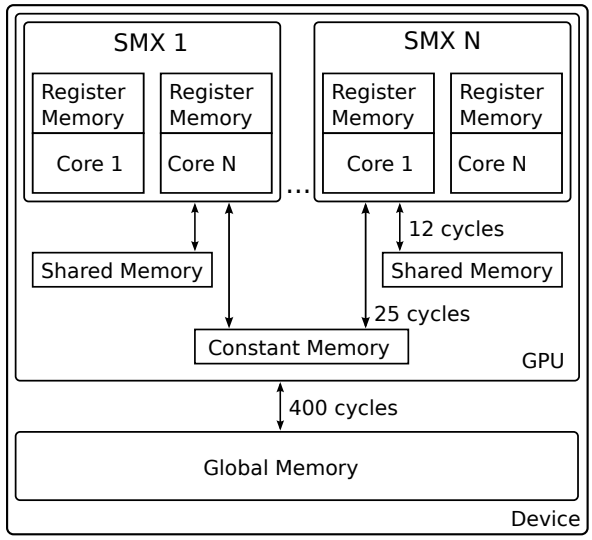

Figure 2.5: GPU Architecture from [Noe11]

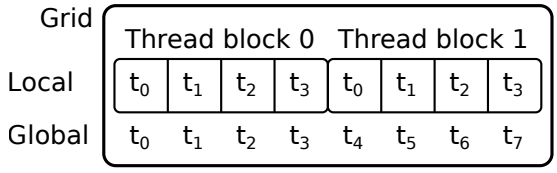

Figure 2.6: CUDA Grid

Chapter 3

Design

In section 2.1.6 we reviewed that the PBVI algorithm are well-suited for parallel implementation, and furthermore we found that the GPUs provide a powerful platform for parallel execution. In this section, we will go through the process of designing the PBVI algorithm.

3.1 Pseudo Code

Given the mathematical model deduced in section 2.1.6, we can outline the pseudo code needed to describe equation 2.18, 2.17 and 2.16.

The backup operation 2.18 will loop over all actions $a \in A$ which will be used to maximize the function g_a^b .

```
Backup(b)->R^n
best : float vector
  for a in A
    best = max g(b,a)
return best
```

The g_b^a operation will take the belief state and action and use them to sum the best hyperplanes for each observation:

```
g(b,a) -> R^n
sum : float vector
best : float vector
  for o in O
    for i in V
      best = max g(i,a,o)
    sum += best
  sum = r + gamma*sum
return sum
```

The $g_{a,o}^i$ operation is the sum over each resulting state:

```
g(i,a,o) -> R^n
sum : float vector
  for s' in S
    sum += O[o,a,s']T[s',a,:]V[s']
return sum
```

By looking at the pseudo code we see that the backup operation will be executed for each belief point in B, and that each execution will result in a new and improved hyperplane. The generation of the new hyperplanes must only be dependent on the value function of its current iteration, thus the new hyperplanes can not be inserted into the current value function V, but must be saved in a buffer V_{tmp} until the entire iteration is complete.

The generation of new hyperplanes in each iteration could lead to reallocation of data many times in each iteration which would degrade performance. With a basic implementation, the data flow should be more clear and measures can be taken to reuse resources.

The general implementation should consist of 5 loopsm of which 4 is used to generate all possible futures, thus the best hyperplane. Using the pseudo code, the loops will have the following relations:

```
for i in 0:ITERATIONS
  for b in Beliefpoints
    for a in Actions
      for o in Observations
        for hp in V
          find combination value
      sum best combinations
    discount best sum
    save best result in V(b)
```

3.2 Vector Prune

Each belief point will have a corresponding hyperplane and if two belief points are in close proximity, it is given that they might share the same hyperplane. This will cause the inner most $g^i_{a,o}$ operation to iterate over identical combinations which will waste calculation. We will therefore implement a pruning operation that ensures the minimal set of hyperplanes between each iteration.

3.3 GPU Design

When implementing the GPU program we need to choose the parameter to parallelize; the backup(b) operation is the outermost loop of each iteration and each belief point is independent, thus well suited for parallelization.

The vector pruning will need to compare the hyperplanes of all belief points, thus is not data independent and have to be implemented for serial execution. This should not impair performance much, given the relative small nature of sorting a few thousands values.

To realize the full potential of the GPU, we will make multiple implementations using both the slow global memory and the fastest shared memory. The only difference is that shared memory will be manually loaded from global memory in the beginning of the kernel execution.

Chapter 4

Implementation

In this section, we will look into the more interesting parts in each implementation. The resulting implementation consist of a few thousand lines of code, all of which can be found in the appendix.

4.1 Naive CPU Implementation

Given that the naive CPU implementation will be the base for all future generations, we have tried to make a general implementation. By using vectorization of data it should be easy to port the CPU version to the GPU.

4.1.1 Data Swapping

Using the pseudo code a very basic implementation was implemented. The implementation was based on allocating new hyperplane vectors in each backup operation and moving the data around via memcpy(), thus performance was horrible. Using the very naive implementation, the flow of data could be analyzed closer and it was clear that besides the buffer V_{tmp}, local buffers are needed in each function implementing the g_b^a and $g_{a,o}^i$ operation.

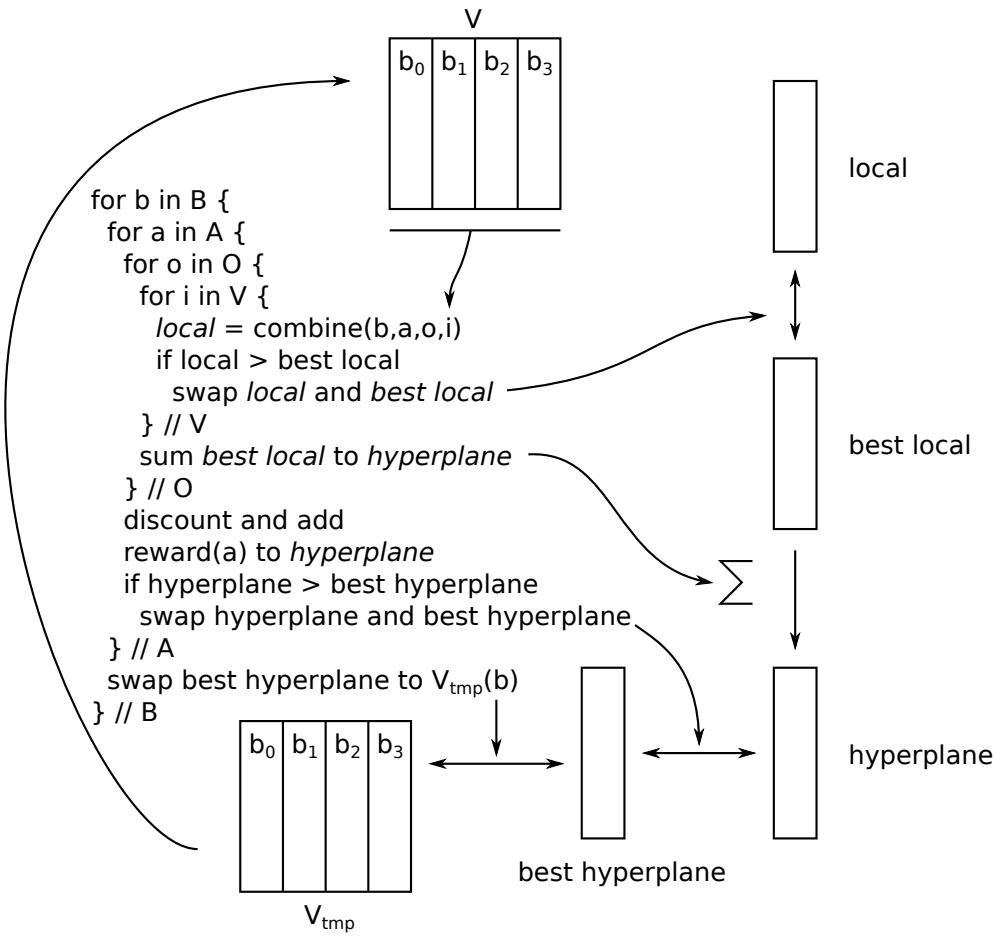

Figure 4.1: Data flow and Hyperplane Swapping

Figure 4.1 shows a data flow graph with swapping of the per-allocated resources. In addition to the V_{tmp} buffer, four buffers were needed, these denoted *local*, *best_local*, *hyperplane*, and *best_hyperplane*. If a better combination is found, it is swapped with a buffer containing outdated values. This way, we do not need to reallocate memory at each iteration, for example:

```
if (local_value > local_max) {
  local_max = local_value;
  swap(local_hyperplane, best_local_hyperplane);
}
```

In the code fragment, the local hyperplane contains the newest hyperplane and local value contains the dot product of the hyperplane and the given belief point. If the new hyperplane is a better solution, i.e. the local value is higher that the currently best value local max, then we need to save the hyperplane. We do this by swapping the local hyperplane and the best local hyperplane. In this way, the important information is saved in the best local hyperplane and the local hyperplane data is now obsolete and can be used for new calculations.

4.1.2 Vectorization

To ensure a good memory access pattern, the data matrices holding the probability distributions of the transition and observation functions are vectorized. This way when reading a value, the next will load with the cache line and be a cache hit.

4.2 Naive GPU Implementation

The naive GPU implementation is a direct port from the CPU version, the only major differences is that the host code must allocate and transfer data to the device, the vectorization of the data matrices makes it easy to transfer data to the device using cudaMalloc() and cudaMemcpy().

4.2.1 Branch Divergence

When the implementation finds a hyperplane with a better value, it will save the new max and swap the hyperplane with its buffer, as seen in the following code fragment:

```
if (local value > local max){
  local max = local value
  swap(local hyperplane, best local hyperplane)
}
```

This control statement will cause a small branch divergence, because a better hyperplane is probably not found by the same combination of a,o and$i$, this will cause a small serialization of execution, but because the divergence is limited to a load and a swap, the performance should not suffer much.

4.3 CPU Vector Prune

To reduce the unnecessary calculations on the same hyperplanes, we will improve the naive implementation to only maintain the unique set of hyperplanes in each iteration. We achieve this by assuming the first hyperplane in V_{tmp} is unique and then comparing the following hyperplanes to it, if a unique hyperplane is found it will be swapped to the unique counter index and the unique counter will be incremented.

4.3.1 Vector Comparison

Given the hyperplane vectors is of the flooding point type, it would not make sense to compare the values directly. Instead we look at the difference of two elements, when this is over a given threshold the vector is distinct.

4.4 GPU Prune

The implementation of the pruning operation on the GPU can not be implemented the same way as on the CPU, given the concurrent execution of the kernels, we can not compare a newly generated hyperplane to the incomplete set. We need to wait until all blocks has completed the iteration before the unique set can be found. The task of finding the unique set among a few thousand vectors is very small compared to finding and calculating all combinations in many dimensions, thus a naive approach was implemented; we assume the first vector to be unique and then compare the following vectors in V_{tmp} to the unique set, if a vector is not in the set it will be added.

4.4.1 Block Synchronization

When the iteration is done generating the new set of hyperplanes we need to synchronize all blocks before we can prune the new set of hyperplanes, the CUDA __syncthreads() function will only synchronize the threads in a block and to our surprise CUDA do not offer any block synchronization features. To ensure that all blocks are done generating the next set of hyperplanes, we need to let the kernel return to host level and let the host synchronize via cudaDeviceSynchronize(). Furthermore a block will complete its execution once started, the platform does not support switching of blocks during execution, thus if we were to implement our own barrier using atomic instructions and spin loops, the belief point size can at most be equal to the number of cores on the device.

4.5 GPU Shared Memory Prune

The base implementation is almost the same as the GPU prune implementation, the only difference is that the data is moved to shared memory in the beginning of each iteration. To ensure that the movement of data from global to shared memory happens as fast as possible, we divide the load between all threads in the block, furthermore we make sure to read along the length of the vectors.

Because of the missing block synchronization features at device level, the kernel needs to reload all data to shared memory on each iteration, which will degrade performance.

4.6 Auxiliary Implementations

4.6.1 POMDP Parser

The POMDP file format described in [Cas09] needs to be parsed in order to work with our implementations. We have chosen to use a MATLAB parser [Spa03], the notation of the reward matrix is $r(s, s', a)$ while our model uses the reward matrix described as $r(s, a)$, to translate the notation we need to free the function of the resulting state dimension, we do this by multiplying the transition probability with the reward e.g. *reward .* transition*, this gives the expected weighted utility, then by summing along the rows of the resulting state, we will get the weighted reward attainable in each state s by executing a: *sum(reward .* transition, 2)*, the translation is wrapped in a MATLAB script which outputs the data in a format which can be directly read into the data matrices of the algorithm, the resulting MATLAB wrapper is found in the appendix.

4.6.2 Build System

When building a CUDA program we have to set a number of flags to be able to use the features of the given architecture and to get the best performance.

Newer versions of CUDA has more advanced compute-capabilities, to enable the full set of features the compiler needs to know the version of the architecture on which the kernel will execute, this is set with the compiler flag "-gencode=arch=compute_30,code=sm_30", where the two digit number represent the architecture version. If the GPU do not have the necessary compute-capabilities the code will not be able to run on the device.

Chapter 5

Testing

Testing the code is essential for software development, both while implementing to ensure correct and working code and for subsequent performance analysis. In this section we will look into the test bench used to ensure the correctness of the implementations and the test bench used to test the performance of the resulting implementation.

5.1 Correctness

To test if our implementations is correct we will compare the resulting solution with a solution of a known implementation that we assume to be correct, to this extend we will use the POMDP solver implemented by [Cas05].

5.1.1 POMDP Comparison Base

To determine a base solution that we will assume to be correct, we have to solve a given POMDP using the POMDP-solver [Cas05], experimentation with this solver have shown that an exact solution will take a very long time to generate

and is impractical to use. Given the approximative nature of our solution, we argue that comparing it to another approximated solution of good quality should suffice in determining if our solution is correct.

We have decided to create the base solution using the POMDP-solve's PBVI implementation, which is able to solve a problem within a day.

5.1.2 Solution Comparison

To ensure that our solution is as correct as the POMDP-solve, we need to compare the found solutions. But comparing two value functions represented by a belief simplex is not trivial, thus instead of comparing the raw data of the found solution, we will compare the found solution by simulating the agent using each policy. We will do this by generating 100.000 random belief states and make both the base solution and our solution find an optimal action to execute. By noting the degree to which the two models agree on the same action, we will decide the correctness of our solution.

Given the approximative nature of both solutions we will expect minor discrepancies between the solutions. We will accept our solution when the error is under 5%.

5.2 Performance

To test the performance of the results we would like to test the implementations against many different problems, but experimentation has shown that the running time of bigger problems will be so high, that they are impractical to use in performance testing, where we need to determine the average of many runs. Thus the performance testing will be based on our relative small homemade fps.pomdp problem that models the power supply of a satellite, the problem is found in the appendix.

5.2.1 Hardware

The hardware used in all performance tests is standard over the counter desktop equipment with the following specification:

- Intel® Core™ i7-920 Processor.
- 6 GB RAM 1600MHz CL9-9-9-24.
- GTX 660 TI - 7 SMXs each with 192 cores.

The system is running a Linux distribution with kernel version 3.5.0-17-generic, at the time of the testing, the newest drivers and compilers was used. All code has been compiled with the highest optimization level.

5.2.2 Testing Parameters

The goal of the performance testing is to compare the speedup of the corresponding implementations, that is naive vs naive and purge vs purge, the easiest way is to compare the computation time using a fixed number of iterations in each implementation. If we were to compare the performance of the time it takes for a problem to converge close to the optimal solution, we would risk measuring other factors such as the quality of the chosen belief points.

To ensure an undisturbed performance picture all measurements is the average of 30 trials, the tests will run on a rebooted system where all non-essential processes have been closed.

The belief points set will be randomly generated, with the random seed hard coded in all implementation, this way all implementation will execute on the same data set.

5.2.3 GPU Testing

The performance of GPU programs can be very sensitive to the block count and threads pr block parameter, we will try different combinations of the parameters. When running the performance testing on a GTX 660 TI with 7 SMX processors, it would not make sense to test with a block count under 7, furthermore to ensure that all cores on the GPU is utilized the total number of threads can not be less that the total core count (1344), therefor we have chosen to run all performance comparison using 4096 belief points, thus 4096 threads in the GPU.

Experimentation has shown that execution CUDA programs in a system with one GPU, while running a desktop environment on 3 monitors with 4 desktops on each, will degrade performance an significant amount. Therefore all performance

testing will be executed with two GPUs installed, one to drive the desktop environment and one for performance testing. By installing two GPUs in the system the PCI Express bus will have its bandwidth cut in two, given we only have to move the problem data to the device once, this change will not influence the performance of the implementation.

Chapter 6

Results

In this section we will go through the results of the testing. It would not make sense to test the performance of a potentially wrong solution, thus each implementation has been tested for correctness before performance testing.

6.1 Correctness

Using the base solution from a number of classic POMDO problems, we will test our solution for correctness by comparing the resulting agents.

Our solutions is based on 4096 belief points that are iterated until the difference between the value function in each iteration is below 1e-7 or 4 hours of calculation time.

The comparison test consist of 100.000 randomly generated belief states which have been given to both our policy and the testing base policy, the following table lists the error between the two policies.

problem	fps.pomdp	maze.pomdp	tiger-grid.pomdp
Naive CPU	0%	0.17%	0.39%
Prune CPU	0%	0.17%	0.39%
Naive GPU	0%	0.14%	0.33%
Prune GPU	0%	0.14%	0.33%
SM GPU	0%	0.14%	0.33%

The fps.pomdp problem is a very small homemade testing problem used through out development, given its small size and the relatively big number of belief points, two almost identical solutions were found with a minuscule error.

The error in all tests are well below 1%, thus we assume our solutions are as correct as the POMDP-solver.

The fps.pomdp problem is found in the appendix and the remaining problems is found at [Cas13].

6.2 Performance

The results from the performance testing is found in the following tables, all times is the average of 30 runs with 30 iterations on 4096 belief points.

- CPU:

	time
Naive CPU	691.1 s
Purge CPU	8.8 s

- GPU

	block size	avg time	speed up
Naive GPU	64	64.1 s	x 11.3
	32	59,2 s	x 11.7
	16	57.2 s	x 12.1
	8	63.8 s	x 10.8
Purge GPU	64	2.39 s	x 3.68
	32	2.36 s	x 3.72
	16	2.33 s	x 3.78
	8	2.63 s	x 3.34
Shared memory GPU	64	1.01 s	x 8.71
	32	0.98 s	x 8.98
	16	0.97 s	x 9.07
	8	0.92 s	x 9.57

The speed up is in relation to the corresponding CPU implementation, thus the naive GPU compared to the naive CPU and the purged CPU to the purged and shared memory GPU.

6.3 Performance Analysis

6.3.1 Naive Speedup

The speedup of the naive implementation is the expected result, given a model that is mostly data bound and the global memory that is 10 times faster than normal RAM.

The combination of block size shows that 16 blocks is the best solution. With 16 blocks there will be 256 threads per block given 4096 belief points, this way the 192 cores in each SMX is fully utilized with the threads of one block, thus each SMX can give all resources to one block.

The worst combination is 8 blocks, this combination might have most threads in one SMX, but given a GPU with 7 SMX processors the 7 first blocks will execute with all SMXs doing work, but given that a block can only execute on one SMX, the last remaining block will not utilize more than 1/7 of the GPUs powers.

6.3.2 Purge Speedup

The CPU purge implementation has given a significant performance boost in the range of 80 times over the naive CPU implementation, this speedup was not expected to be so high. The very high performance boost of the purge speed up, can be due to the nature of the problem, given the limited time available to run tests, a very small problem size with only three states was choosen, when covering the very small belief space with thousands of vectors, it is given that most belief points will share the same hyperplanes, thus the purge will reduce the problem to a fraction of the original problem, this way the code will spend less time generating the many combinations, which the GPU is able to speed up using the massive parallel power.

It is our belief that given a bigger problem the purge algorithm will not be able to reduce the problem as much, which will make the relative improvement of the GPU implementation more significant. Testing the implementation against a big enough problem would take weeks to be statistical significant, which is not possible given the time frame of this project.

6.3.3 Shared Memory Speedup

The found speed up of the shared memory implementation, tends to favor a small block size, this way each block has a maximal number of threads per block which will prolong the lifetime of the block, thus letting more threads exploit the fast shared memory.

Because there is no way to synchronize across blocks while running a kernel, we have to return from the kernel on each iteration and the synchronize on the

host side, this way the memory moved to shared memory will be deleted and will have to be reloaded in the next iteration.

Chapter 7

Discussion

Our implementations did not reach the hypothesized level of improvement as the raw power of the GPU platform has to offer.

The reason to why we were unable to unleash the full potential of the GPU platform is that the pruning operation improved the execution time so much, that it has become essential for the algorithm. As this is data dependent across belief points and must execute in serial. We could choose not the prune the value function and then keep the kernel running on the GPU for the entire lifetime of the program, and this would probably improve the performance of the implementation close to the order of magnitude, but because pruning have shown improvements in the running time over the naive implementation of close to 80 times, it is a given part of the algorithm.

The performance of this type of problem will probably not be able to exploit the full potential of the GPU before CUDA will support block synchronization from the kernels. We assume that the reason to why this essential feature is not yet implemented is because the architecture simply does not support this feature. Otherwise, it would probably have been implemented in the first version.

7.1 GPU Computing Future

Future generations of GPUs will melt together with the CPU using the Heterogeneous System Architecture (HSA) [GK13]. In HSA, the CPU and GPU will be on the same chip with a fully coherent memory and cache system, and furthermore, the CPU and GPU will share a real unified address space in which data can be freely passed between the CPU and GPU. On this type of architecture, we contemplate that a PBVI based POMDP solver could utilize the maximal potential of the architecture, given the GPU could accelerate the embarrassing parallel parts of the code, while reaping the benefits of the advanced synchronization features of the CPU. Furthermore the problem size can be increased given the limit of the device memory no longer applies.

Chapter 8

Conclusion

In this project, we have reviewed the development and implementation of a PBVI POMDP solver. The implementations was able to determine a correct solution, and was faster compared to coresponding CPU implementations in performance tests.

While the potential power of the platform is closer to 100 times that of the CPU, in terms of faster speed, our implementation reached only a 10-fold improvement. The main reason for this is the algorithm's dependency on serial execution after each iteration. Upcoming improvements to a combination of the CPU and GPU platform promise a powerful platform, which may better address the type of problem reviewed in this project. This may enable our implementation to reach the full potential.

Appendix A

Code

A.1 CPU Implementation

Listing A.1: CPU Implementation

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <vector>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <limits>

using namespace std;

#define BELIEF_POINT_COUNT  (4096)
#define ITERATIONS    (30)

/* Tolerance for sum = 1 */
#define EPSILON      (0.0000001)

//#define PURGE

int hyperplane_action[BELIEF_POINT_COUNT] ;

typedef struct {

 int actions;
```

```
28   int states;
29   int observations;
30   int belief_point_count;
31
32   int *hyperplane_action;
33
34   float discount;
35
36   float **B;
37   float *R;
38   float *O;
39   float *T;
40
41   float **V;
42   float **Vtmp;
43  } POMDP;
44
45  vector<string> split(string s, char delim){
46
47   vector<string> result;
48   int pos = s.find(delim);
49
50   if (-1 == pos) {
51    result.push_back(s);
52    return result;
53   }
54
55   while (-1 != pos) {
56    pos = s.find(delim);
57    result.push_back(s.substr(0, pos));
58    s = s.substr(pos+1);
59   }
60
61   return result;
62  }
63
64  POMDP load_pomdp(char *path){
65
66   ifstream input( path );
67
68   POMDP pomdp;
69   pomdp.actions = -1;
70   pomdp.states = -1;
71   pomdp.observations = -1;
72
73   if (!input) {
74    cerr << "Error - POP file not loaded" << endl;
75    return pomdp;
76   }
77
78   string line;
79   while( getline(input, line) ){
80    cout << line << endl;
81
82    int pos = 0;
83    if ( string::npos != line.find("discount:")) {
84     pos = line.find(": ") + 2;
85     string s = line.substr(pos);
86     istringstream(s) >> pomdp.discount;
87    } else if ( string::npos != line.find("states:") ) {
88     pos = line.find(": ") + 2;
89     string s = line.substr(pos);
90     istringstream(s) >> pomdp.states;
91    } else if ( string::npos != line.find("actions:") ) {
92     pos = line.find(": ") + 2;
```

```
93      string s = line.substr(pos);
94      istringstream(s) >> pomdp.actions;
95    } else if ( string::npos != line.find("observations:") ) {
96      pos = line.find(": ") + 2;
97      string s = line.substr(pos);
98      istringstream(s) >> pomdp.observations;
99    } else if( string::npos != line.find("T") ){
100     if (-1 == pomdp.states || -1 == pomdp.observations || -1 == pomdp.actions)
            {
101       printf("Load error - missing information");
102       return pomdp;
103     }
104
105     pomdp.T = new float[pomdp.actions * pomdp.states * pomdp.states];
106     int index = 0;
107     // For the number of lines
108     for (int i = 0; i < pomdp.actions * pomdp.states; ++i) {
109       getline(input, line);
110       cout << line << endl;
111       vector<string> tmp = split(line, ',');
112
113       for (int j = 0; j < pomdp.states; ++j) {
114         istringstream(tmp[j]) >> pomdp.T[index++];
115       }
116     }
117   } else if( string::npos != line.find("O") ){
118
119     if (-1 == pomdp.states || -1 == pomdp.observations || -1 == pomdp.actions)
            {
120       printf("Load error - missing information");
121       return pomdp;
122     }
123
124     pomdp.O = new float[pomdp.actions * pomdp.states * pomdp.observations];
125
126     int index = 0;
127     // For the number of lines
128     for (int i = 0; i < pomdp.actions * pomdp.states; ++i) {
129       getline(input, line);
130       cout << line << endl;
131       vector<string> tmp = split(line, ',');
132
133       for (int j = 0; j < pomdp.observations; ++j) {
134         istringstream(tmp[j]) >> pomdp.O[index++];
135       }
136     }
137   } else if( string::npos != line.find("R") ){
138
139     if (-1 == pomdp.states || -1 == pomdp.observations || -1 == pomdp.actions)
            {
140       printf("Load error - missing information");
141       return pomdp;
142     }
143
144     pomdp.R = new float[pomdp.actions * pomdp.states];
145
146     int index = 0;
147     // For the number of lines
148     for (int i = 0; i < pomdp.actions; ++i) {
149       getline(input, line);
150       cout << line << endl;
151       vector<string> tmp = split(line, ',');
152
153       for (int j = 0; j < pomdp.states; ++j) {
154         istringstream(tmp[j]) >> pomdp.R[index++];
```

```
155       }
156      }
157     }
158    }
159
160
161    /*
162     * Setup initial hyperplane value.
163     */
164    double min = pomdp.R[0];
165
166    for (int action = 0; action < pomdp.actions; ++action) {
167     for (int state = 0; state < pomdp.states; ++state) {
168      if ( pomdp.R[ pomdp.states * action + state ] < min ) {
169       min = pomdp.R[ pomdp.states * action + state ];
170      }
171     }
172    }
173
174    float initial_value = 0;
175    if (1.0 != pomdp.discount) {
176     initial_value = (1/(1-pomdp.discount))*min;
177    }
178    printf("initial value: %f\n", initial_value);
179
180
181    /* Allocate a vector of pointers */
182    pomdp.V = new float*[BELIEF_POINT_COUNT];
183    pomdp.Vtmp = new float*[BELIEF_POINT_COUNT];
184
185    for (int b = 0; b < BELIEF_POINT_COUNT; ++b) {
186
187     /* Allocate a hyperplane vector for each belief point. */
188     pomdp.V[b] = new float[pomdp.states];
189     pomdp.Vtmp[b] = new float[pomdp.states];
190
191     for (int state = 0; state < pomdp.states; ++state) {
192      pomdp.V[b][state] = initial_value;
193      pomdp.Vtmp[b][state] = initial_value;
194     }
195    }
196
197   #ifndef PURGE
198    pomdp.belief_point_count = BELIEF_POINT_COUNT;
199   #endif
200
201   #ifdef PURGE
202    pomdp.belief_point_count = 1;
203   #endif
204    return pomdp;
205   }
206
207   double dist(float *a, float *b, int length){
208
209    double result = 0;
210    for (int i = 0; i < length; ++i) {
211     result += (a[i]-b[i])*(a[i]-b[i]);
212    }
213
214    return sqrt(result);
215   }
216
217   void vector_print(float *p, int length){
218
219    for (int i = 0; i < length; ++i) {
```

```
220   //cout << p[i] << " ";
221   printf("%1.20f", p[i]);
222  }
223  printf("\n");
224 }
225
226 int sum_to_one(float *p, int length){
227  double sum = 0;
228  for (int i = 0; i < length; ++i) {
229   sum += p[i];
230  }
231  if ( fabs(1.0 - sum) > EPSILON ) {
232   return 0;
233  }
234  return 1;
235 }
236
237 /*
238  * count:  number of belief points to generate.
239  * states:  number of states, thus number of elements in each belief point
         vector.
240  */
241 float ** init_B_uniform( int count, int states ){
242
243  float **B = new float*[count];
244
245  // Number of generated belief points.
246  int bp;
247  // Setup the extreme belief points.
248  for (bp = 0; bp < states; ++bp) {
249   B[bp] = new float[states];
250   memset(B[bp], 0, states*sizeof(float));
251   B[bp][bp] = 1;
252  }
253
254  double** dist_matrix = new double*[count];
255  for(int i = 0; i < count; ++i){
256   dist_matrix[i] = new double[count];
257  }
258
259  for (int i = 0; i < count; ++i) {
260   for (int j = 0; j < count; ++j) {
261    dist_matrix[i][j] = 0;
262   }
263  }
264
265  double max_dist = 0;
266  int index1 = -1, index2 = -1;
267
268  // Calculate the initial distances and find the longest distance between two
         points.
269  for (int i = 0; i < states; ++i) {
270   for (int j = i+1 ; j < states; ++j) {
271    dist_matrix[i][j] = dist(B[i], B[j], states);
272
273    if ( dist_matrix[i][j] > max_dist ) {
274     max_dist = dist_matrix[i][j];
275     index1 = i; index2 = j;
276    }
277   }
278  }
279
280  for (; bp < count; ++bp) {
281
282   // Make a new belief point.
```

```
283   B[bp] = new float[states];
284
285   // Set the belief point value between the two points with the longest
          distance.
286   for (int k = 0; k < states; ++k) {
287    B[bp][k] = (B[index1][k] + B[index2][k]) / 2.0;
288   }
289   // Sanity test on the vectors sum, which always must be 1.
290   if (!sum_to_one(B[bp], states)) {
291    printf("Error - Belief point between points at %d x %d, don't sum to 1!\n"
          , index1, index2);
292    vector_print(B[bp], states);
293   }
294
295   // Calaulate the distances from the new belief point to all other points.
296   for (int i = 0; i < bp; ++i) {
297    dist_matrix[i][bp] = dist(B[bp], B[i], states);
298   }
299
300   // Null the old longest distance in the distance matrix.
301   dist_matrix[index1][index2] = 0;
302
303   // Find the new longest distance.
304   max_dist = 0;
305   for (int i = 0; i <= bp; ++i) {
306    for (int j = i+1 ; j <= bp; ++j) {
307     if ( dist_matrix[i][j] > max_dist ) {
308      max_dist = dist_matrix[i][j];
309      index1 = i; index2 = j;
310     }
311    }
312   }
313  }
314
315  return B;
316 }
317
318 /*
319  * count:  number of belief points to generate.
320  * states:  number of states, thus number of elements in each belief point
          vector.
321  */
322 float ** init_B_random( int count, int states ){
323
324  float **B = new float*[count];
325
326  // Number of generated belief points.
327  int bp;
328  // Setup the extreme belief points.
329  for (bp = 0; bp < states; ++bp) {
330   B[bp] = new float[states];
331
332   memset(B[bp], 0, states*sizeof(float));
333
334   B[bp][bp] = 1;
335  }
336
337  // Initialize random seed
338  srand(42);
339
340  for (; bp < count; ++bp) {
341
342   // Make a new belief point
343   B[bp] = new float[states];
344
```

```
345   float sum = 0;
346   // Assign random numbers to each element in a belief point.
347   for (int j = 0; j < states; ++j) {
348    // Make a random number.
349    B[bp][j] = rand() % 1000000 + 1;
350    sum += B[bp][j];
351   }
352
353   // Normalize the vector to sum to 1.0.
354   for (int j = 0; j < states; ++j) {
355    B[bp][j] /= sum;
356   }
357   // Sanity test on the vectors sum, which always must be 1.0
358   if (!sum_to_one(B[bp], states)) {
359    // Redo the random vector.
360    bp--;
361   }
362  }
363  return B;
364 }
365
366 float dot(float *a, float *b, int length){
367  float sum = 0;
368  for (int i = 0; i < length; ++i) {
369   sum += a[i] * b[i];
370  }
371  return sum;
372 }
373
374 int vector_equal(float *a, float *b, int length){
375  for (int i = 0; i < length; ++i) {
376   // if just one element is different, go on.
377   if (fabs( a[i] - b[i] ) > 0.001) {
378    return 0;
379   }
380  }
381  return 1;
382 }
383
384 int main() {
385  POMDP pomdp = load_pomdp( "fps.pop" );
386
387  timeval t1, t2;
388  double elapsedTime;
389  // start timer
390  gettimeofday(&t1, NULL);
391
392  cout << "Generating grid:" << endl;
393  float **B = init_B_random(BELIEF_POINT_COUNT, pomdp.states);
394  cout << "Done generating grid:" << endl;
395
396  float *hyperplane = new float[pomdp.states];
397  float *local_hyperplane = new float[pomdp.states];
398  float *best_local_hyperplane = new float[pomdp.states];
399  float *best_hyperplane = new float[pomdp.states];
400
401  int best_action;
402  int uniq = 0;
403  for (int i = 0; i < ITERATIONS; ++i) {
404   for (int b = 0; b < BELIEF_POINT_COUNT; ++b) {
405
406    float max = 0;
407    for (int action = 0; action < pomdp.actions; ++action) {
408
409     memset(hyperplane, 0, pomdp.states*sizeof(float));
```

```
410     for (int observation = 0; observation < pomdp.observations; ++observation
            ) {
411
412       float local_max = 0;
413       memset(best_local_hyperplane, 0, pomdp.states*sizeof(float));
414
415       /* For each hyperplane */
416       for (int i = 0; i < pomdp.belief_point_count; ++i) {
417
418         for (int s = 0; s < pomdp.states; ++s) {
419          local_hyperplane[s] =
420            pomdp.O[ pomdp.states*pomdp.observations * action + observation ]*
421            pomdp.T[ pomdp.states*pomdp.states * action + pomdp.states * s ]*
422            pomdp.V[i][0];
423         }
424         for (int sp = 1; sp < pomdp.states; ++sp) {
425          for (int s = 0; s < pomdp.states; ++s) {
426           local_hyperplane[s] +=
427             pomdp.O[ pomdp.states*pomdp.observations * action + pomdp.
                  observations * sp + observation ]*
428             pomdp.T[ pomdp.states*pomdp.states * action + pomdp.states * s + sp
                   ]*
429             pomdp.V[i][sp];
430          }
431         }
432
433         float local_value = dot(local_hyperplane, B[b], pomdp.states);
434
435         if (local_value > local_max) {
436          local_max = local_value;
437          /* Swap the best local and newest hyperplane - making the old best the
                 new buffer*/
438          swap(local_hyperplane, best_local_hyperplane);
439         }
440       } // Hyperplane
441
442       for (int j = 0; j < pomdp.states; ++j) {
443        hyperplane[j] += best_local_hyperplane[j];
444       }
445     } // Observations
446
447     /* Discount the vector and add immediate reward */
448     for (int j = 0; j < pomdp.states; ++j) {
449      hyperplane[j] =  pomdp.R[ pomdp.states * action + j ] + pomdp.discount*
            hyperplane[j];
450     }
451
452     float value = dot(hyperplane, B[b], pomdp.states);
453
454
455     if ( value > max ) {
456      max = value;
457      /* Swap the best local and newest hyperplane - making the old best the
             new buffer*/
458 #ifndef PURGE
459      swap(hyperplane, pomdp.Vtmp[b]);
460      hyperplane_action[b] = action;
461 #endif
462
463 #ifdef PURGE
464      swap(hyperplane, best_hyperplane);
465      best_action = action;
466 #endif
467     }
468    } // Actions
```

```
469
470 #ifdef PURGE
471    int distinct = 1;
472    for (int u = 0; u < uniq; ++u) {
473     // If the HP already exists, just go on.
474     if ( vector_equal(pomdp.Vtmp[u], best_hyperplane, pomdp.states) ) {
475      distinct = 0;
476      break;
477     }
478    }
479    if (distinct) {
480     swap(best_hyperplane, pomdp.Vtmp[uniq]);
481     hyperplane_action[uniq] = best_action;
482     uniq++;
483    }
484 #endif
485   } // Belief points
486
487   swap(pomdp.V, pomdp.Vtmp);
488 #ifdef PURGE
489   pomdp.belief_point_count = uniq;
490   uniq = 0;
491 #endif
492   cout << ">>>> " << i << endl;
493  } // Iterations
494
495  // stop timer
496  gettimeofday(&t2, NULL);
497  // compute and print the elapsed time in millisec
498  elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0; // sec to ms
499  elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0; // us to ms
500
501  printf("\n\n%f ms.\n\n", elapsedTime );
502  cout << "!!!Job done!!!" << endl;
503  return 0;
504 }
```

A.2 GPU Implementation

Listing A.2: GPU Implementation

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <iostream>
4  #include <sys/time.h>
5  #include <time.h>
6  #include <fstream>
7  #include <sstream>
8  #include <vector>
9  #include <math.h>
10 #include <limits>
11 #include <string.h>
12 #include <cuda.h>
13
14 #define STATE_COUNT  (3)
15 #define ACTION_COUNT  (4)
16 #define OBSERVATION_COUNT  (3)
17
18 #define BELIEF_POINT_COUNT  (4096)
19 #define BLOCKS  (8)
20 #define ITERATIONS  (30)
21 #define DISCOUNT  (0.95)
22
23 #define THREADS_PR_BLOCK  (BELIEF_POINT_COUNT / BLOCKS)
24 /* Tolerance for sum = 1 */
25 #define EPSILON  (0.0001)
26
27 using namespace std;
28
29 __global__ void pbvi(
30   float **V, float **Vtmp,
31   float *T, float *O, float *R, float *B,
32   int *best_action,
33   int states,
34   int observations,
35   int actions,
36   float discount );
37
38 __global__ void pbvi_arc(
39   float **V, float **Vtmp,
40   int *best_action,
41   int states,
42   int observations,
43   int actions,
44   float discount);
45
46 __global__ void pbvi_shared_memory(
47   float **V, float **Vtmp,
48   float *T, float *O, float *R, float *B,
49   int *best_action,
50   int states,
51   int observations,
52   int actions,
53   float discount);
54
55 __global__ void init_pbvi(float **dev_V, float **dev_Vtmp, float
       initial_value, int states);
56
57 __global__ void pbvi_get(float **dev_V, float *result, int states);
58
59 __global__ void pbvi_purge(float **dev_V, int states);
```

```
60
61 __device__ int unique_belief_points = BELIEF_POINT_COUNT;
62
63
64 __global__ void init_pbvi( float **V, float **Vtmp, float initial_value, int
       states ){
65   int b = threadIdx.x + blockIdx.x * blockDim.x;
66   V[b] = new float[states];
67   Vtmp[b] = new float[states];
68
69   for (int i = 0; i < states; ++i){
70     V[b][i] = initial_value;
71     Vtmp[b][i] = initial_value;
72   }
73 }
74
75 __global__ void pbvi(
76   float **V, float **Vtmp,
77   float *dev_T, float *dev_O, float *dev_R, float *dev_B,
78   int *best_action,
79   int states,
80   int observations,
81   int actions,
82   float discount){
83
84   int b = threadIdx.x + blockIdx.x * blockDim.x;
85   best_action[b] = -1;
86
87   float *hyperplane = new float[states];
88   float *local_hyperplane = new float[states];
89   float *best_local_hyperplane = new float[states];
90
91   float max = 0;
92   for (int action = 0; action < actions; ++action) {
93
94     memset(hyperplane, 0, states*sizeof(float));
95     for (int observation = 0; observation < OBSERVATION_COUNT; ++observation)
         {
96
97       float local_max = 0;
98       memset(best_local_hyperplane, 0, states*sizeof(float));
99       /* For each hyperplane */
100      for (int i = 0; i < unique_belief_points; ++i) {
101        for (int s = 0; s < states; ++s) {
102          local_hyperplane[s] =
103          dev_O[ states*observations * action + observation ]*
104            dev_T[ states*states * action + states * s ]*
105            V[i][0];
106        }
107        for (int sp = 1; sp < states; ++sp) {
108          for (int s = 0; s < states; ++s) {
109            local_hyperplane[s] +=
110            dev_O[ states*observations * action + observations * sp +
                 observation ]*
111            dev_T[ states*states * action + states * s + sp ]*
112            V[i][sp];
113          }
114        }
115
116        float local_value = 0;
117        for (int i = 0; i < states; ++i) {
118          local_value += local_hyperplane[i] * dev_B[states * b + i];
119        }
120
121        if ( local_value > local_max ) {
```

```
122           local_max = local_value;
123           /* Swap the best local and newest hyperplane - making the old best
                  the new buffer*/
124           float *tmp = local_hyperplane;
125           local_hyperplane = best_local_hyperplane;
126           best_local_hyperplane = tmp;
127         }
128       } // Hyperplane
129
130       for (int j = 0; j < states; ++j) {
131         hyperplane[j] += best_local_hyperplane[j];
132       }
133     } // Observations
134
135     /* Discount the vector and add immediate reward */
136     for (int j = 0; j < states; ++j) {
137       hyperplane[j] =  dev_R[states * action + j] + discount*hyperplane[j];
138     }
139     float value = 0;
140     for (int i = 0; i < states; ++i) {
141       value += hyperplane[i] * dev_B[states * b + i];
142     }
143
144     if ( value > max ) {
145       best_action[b] = action;
146
147       max = value;
148       /* Swap the best local and newest hyperplane - making the old best the
              new buffer*/
149       float *tmp = hyperplane;
150       hyperplane = Vtmp[b];
151       Vtmp[b] = tmp;
152     }
153   } // Actions
154
155   delete hyperplane;
156   delete local_hyperplane;
157   delete best_local_hyperplane;
158 }
159
160 __global__ void pbvi_shared_memory(
161   float **V, float **Vtmp,
162   float *T, float *O, float *R, float *B,
163   int *best_action,
164   int states,
165   int observations,
166   int actions,
167   float discount){
168
169   int b = threadIdx.x + blockIdx.x * blockDim.x;
170
171   __shared__ float s_T[ STATE_COUNT * STATE_COUNT * ACTION_COUNT ];
172   __shared__ float s_O[ STATE_COUNT * ACTION_COUNT * OBSERVATION_COUNT];
173   __shared__ float s_R[ STATE_COUNT * ACTION_COUNT ];
174   __shared__ float s_B[ STATE_COUNT * THREADS_PR_BLOCK ];
175
176   for (int i = 0; i < STATE_COUNT; ++i) {
177     s_B[states * threadIdx.x + i] = B[states * b + i];
178   }
179
180   // Transfer T to shared memory.
181   if ( STATE_COUNT * STATE_COUNT * ACTION_COUNT <= THREADS_PR_BLOCK ) {
182     // prevent the index from overflowing.
183     if ( threadIdx.x < STATE_COUNT * STATE_COUNT * ACTION_COUNT ) {
184       s_T[ threadIdx.x ] = T[ threadIdx.x ];
```

```
185       }
186     } else {
187       // homemade ceil: ceil(a/b) = (int)((a+b-1)/b)
188       int iterations = (int)((STATE_COUNT * STATE_COUNT * ACTION_COUNT +
              THREADS_PR_BLOCK - 1)/ THREADS_PR_BLOCK);
189       int index;
190       for (int i = 0; i < iterations; ++i) {
191         index = threadIdx.x * iterations + i;
192         // prevent the index from overflowing.
193         if ( index < STATE_COUNT * STATE_COUNT * ACTION_COUNT ) {
194           s_T[ index ] = T[ index ];
195         }
196       }
197     }
198
199     // Transfer O to shared memory.
200     if ( STATE_COUNT * ACTION_COUNT * OBSERVATION_COUNT <= THREADS_PR_BLOCK) {
201       // prevent the index from overflowing.
202       if ( threadIdx.x < STATE_COUNT * ACTION_COUNT * OBSERVATION_COUNT ) {
203         s_O[ threadIdx.x ] = O[ threadIdx.x ];
204       }
205     } else {
206       // homemade ceil: ceil(a/b) = (int)((a+b-1)/b)
207       int iterations = (int)((STATE_COUNT * ACTION_COUNT * OBSERVATION_COUNT +
              THREADS_PR_BLOCK - 1)/ THREADS_PR_BLOCK);
208       int index;
209       for (int i = 0; i < iterations; ++i) {
210         index = threadIdx.x * iterations + i;
211         // prevent the index from overflowing.
212         if ( index < STATE_COUNT * ACTION_COUNT * OBSERVATION_COUNT ) {
213           s_O[ index ] = O[ index ];
214         }
215       }
216     }
217
218     // Transfer R to shared memory.
219     if ( STATE_COUNT * ACTION_COUNT <= THREADS_PR_BLOCK) {
220       // prevent the index from overflowing.
221       if ( threadIdx.x < STATE_COUNT * ACTION_COUNT ) {
222         s_R[ threadIdx.x ] = R[ threadIdx.x ];
223       }
224     } else {
225       // homemade ceil: ceil(a/b) = (int)((a+b-1)/b)
226       int iterations = (int)((STATE_COUNT * ACTION_COUNT + THREADS_PR_BLOCK -
              1)/ THREADS_PR_BLOCK);
227
228       int index;
229       for (int i = 0; i < iterations; ++i) {
230         index = threadIdx.x * iterations + i;
231         // prevent the index from overflowing.
232         if ( index < STATE_COUNT * ACTION_COUNT ) {
233           s_R[ index ] = R[ index ];
234         }
235       }
236     }
237
238     best_action[b] = -1;
239     float *hyperplane = new float[states];
240     float *local_hyperplane = new float[states];
241     float *best_local_hyperplane = new float[states];
242
243     float max = 0;
244     for (int action = 0; action < actions; ++action) {
245
246       memset(hyperplane, 0, states*sizeof(float));
```

```
247     for (int observation = 0; observation < OBSERVATION_COUNT; ++observation)
            {
248
249       float local_max = 0;
250       memset(best_local_hyperplane, 0, states*sizeof(float));
251       /* For each hyperplane */
252       for (int i = 0; i < unique_belief_points; ++i) {
253
254         for (int s = 0; s < states; ++s) {
255           local_hyperplane[s] =
256             s_O[ states*observations * action + observation ]*
257             s_T[ states*states * action + states * s ]*
258             V[i][0];
259         }
260         for (int sp = 1; sp < states; ++sp) {
261           for (int s = 0; s < states; ++s) {
262             local_hyperplane[s] +=
263             s_O[ states*observations * action + observations * sp +
                    observation ]*
264             s_T[ states*states * action + states * s + sp ]*
265             V[i][sp];
266           }
267         }
268
269         float local_value = 0;
270         for (int i = 0; i < states; ++i) {
271           local_value += local_hyperplane[i] * s_B[states * threadIdx.x + i];
272         }
273         //float value = dot(local_hyperplane, B[b], states);
274
275         if ( local_value > local_max ) {
276           local_max = local_value;
277           /* Swap the best local and newest hyperplane - making the old best
                the new buffer*/
278           float *tmp = local_hyperplane;
279           local_hyperplane = best_local_hyperplane;
280           best_local_hyperplane = tmp;
281         }
282       } // Hyperplane
283
284       for (int j = 0; j < states; ++j) {
285         hyperplane[j] += best_local_hyperplane[j];
286       }
287     } // Observations
288
289     /* Discount the vector and add immediate reward */
290     for (int j = 0; j < states; ++j) {
291       hyperplane[j] =  s_R[states * action + j] + discount*hyperplane[j];
292     }
293
294     float value = 0;
295     for (int i = 0; i < states; ++i) {
296       value += hyperplane[i] * s_B[states * threadIdx.x + i];
297     }
298     if ( value > max ) {
299       best_action[b] = action;
300
301       max = value;
302       /* Swap the best local and newest hyperplane - making the old best the
            new buffer*/
303       float *tmp = hyperplane;
304       hyperplane = Vtmp[b];
305       Vtmp[b] = tmp;
306     }
307
```

```
308   } // Actions
309
310   delete hyperplane;
311   delete local_hyperplane;
312   delete best_local_hyperplane;
313 }
314
315 __global__ void pbvi_get(float **dev_V, float *dev_result, int states){
316   int b = threadIdx.x + blockIdx.x * blockDim.x;
317   for (int i = 0; i < states; ++i){
318     dev_result[states * b + i] = dev_V[b][i];
319   }
320 }
321
322 __device__ int cuda_vector_equal(float *a, float *b, int length){
323   int tid = threadIdx.x + blockIdx.x * blockDim.x;
324   for (int i = 0; i < length; ++i) {
325     // if just one element is different, go on.
326     if ( fabs( a[i] - b[i] ) > 0.001) {
327       return 0;
328     }
329   }
330   return 1;
331 }
332
333 __global__ void pbvi_purge( float **Vtmp, int states ){
334
335   int tid = threadIdx.x + blockIdx.x * blockDim.x;
336   int unique_count = 1;
337   int is_uniq = 1;
338   /*
339    * If multiple threads and blocks is allocated, make sure that only one
          thread
340    * makes changes to the hyperplanes.
341    */
342   if (tid == 0)  {
343     /* The first belifepoint will allways be unique, thus we start from 1 */
344     for (int i = 1; i < BELIEF_POINT_COUNT; ++i) {
345       is_uniq = 1;
346       /* Compare all unique to the next hyperplane */
347       for (int j = 0; j < unique_count; ++j) {
348         if ( cuda_vector_equal(Vtmp[i], Vtmp[j], states) ){
349           is_uniq = 0;
350           break;
351         }
352       }
353       /* If the hyperplane is unique */
354       if ( is_uniq ) {
355         /* Swap the new unique hyperplane to the unique count index
356          * this way the unique_count first vectors will hold the unique */
357         if ( i != unique_count){
358           float *tmp = Vtmp[unique_count];
359           Vtmp[unique_count] = Vtmp[i];
360           Vtmp[i] = tmp;
361         }
362         unique_count++;
363       }
364     }
365   }
366   unique_belief_points = unique_count;
367 }
368
369 __constant__ float c_O[ OBSERVATION_COUNT * STATE_COUNT * ACTION_COUNT ];
370 __constant__ float c_R[ STATE_COUNT * ACTION_COUNT ];
371 __constant__ float c_B[ BELIEF_POINT_COUNT * STATE_COUNT ];
```

```
372 __constant__ float c_T[ STATE_COUNT * STATE_COUNT * ACTION_COUNT ];
373
374 __global__ void pbvi_constant_memory(
375   float **V, float **Vtmp,
376   int *best_action,
377   int states,
378   int observations,
379   int actions,
380   float discount){
381
382   int b = threadIdx.x + blockIdx.x * blockDim.x;
383   best_action[b] = -1;
384
385   float *hyperplane = new float[states];
386   float *local_hyperplane = new float[states];
387   float *best_local_hyperplane = new float[states];
388
389   float max = 0;
390   for (int action = 0; action < actions; ++action) {
391
392     memset(hyperplane, 0, states*sizeof(float));
393     for (int observation = 0; observation < OBSERVATION_COUNT; ++observation)
          {
394
395       float local_max = 0;
396       memset(best_local_hyperplane, 0, states*sizeof(float));
397       /* For each hyperplane */
398       for (int i = 0; i < BELIEF_POINT_COUNT; ++i) {
399
400         for (int s = 0; s < states; ++s) {
401           local_hyperplane[s] =
402           c_O[ states*observations * action + observation ]*
403           c_T[ states*states * action + states * s ]*
404           V[i][0];
405         }
406         for (int sp = 1; sp < states; ++sp) {
407           for (int s = 0; s < states; ++s) {
408             local_hyperplane[s] +=
409             c_O[ states*observations * action + observations * sp +
                    observation ]*
410             c_T[ states*states * action + states * s + sp ]*
411             V[i][sp];
412           }
413         }
414
415         float local_value = 0;
416         for (int i = 0; i < states; ++i) {
417           local_value += local_hyperplane[i] * c_B[states * b + i];
418         }
419
420         if ( local_value > local_max ) {
421           local_max = local_value;
422           /* Swap the best local and newest hyperplane - making the old best
                  the new buffer*/
423           float *tmp = local_hyperplane;
424           local_hyperplane = best_local_hyperplane;
425           best_local_hyperplane = tmp;
426         }
427       } // Hyperplane
428
429       for (int j = 0; j < states; ++j) {
430         hyperplane[j] += best_local_hyperplane[j];
431       }
432     } // Observations
433
```

```
434       /* Discount the vector and add immediate reward */
435       for (int j = 0; j < states; ++j) {
436         hyperplane[j] =  c_R[states * action + j] + discount*hyperplane[j];
437       }
438
439       float value = 0;
440       for (int i = 0; i < states; ++i) {
441         value += hyperplane[i] * c_B[states * b + i];
442       }
443
444       if ( value > max ) {
445         best_action[b] = action;
446
447         max = value;
448         /* Swap the best local and newest hyperplane - making the old best the
                new buffer */
449         float *tmp = hyperplane;
450         hyperplane = Vtmp[b];
451         Vtmp[b] = tmp;
452       }
453     } // Actions
454
455     delete hyperplane;
456     delete local_hyperplane;
457     delete best_local_hyperplane;
458   }
459
460   typedef struct {
461
462     int actions;
463     int states;
464     int observations;
465     int belief_point_count;
466
467     int *hyperplane_action;
468
469     float discount;
470     float initial_value;
471
472     float **B;
473     float *R;
474     float *O;
475     float *T;
476
477     float **V;
478     float **Vtmp;
479   } POMDP;
480
481   /*
482    * Returns a string vector of the split tokens
483    */
484   vector<string> split(string s, char delim){
485
486     vector<string> result;
487     int pos = s.find(delim);
488
489     if (-1 == pos) {
490       result.push_back(s);
491       return result;
492     }
493     while (-1 != pos) {
494       pos = s.find(delim);
495       result.push_back(s.substr(0, pos));
496       s = s.substr(pos+1);
497     }
```

```
498   return result;
499 }
500
501 POMDP load_pomdp(char *path){
502
503   ifstream input( path );
504
505   POMDP pomdp;
506   pomdp.actions = -1;
507   pomdp.states = -1;
508   pomdp.observations = -1;
509
510   if (!input) {
511     cerr << "Error - POP file not loaded" << endl;
512     return pomdp;
513   }
514
515   string line;
516   while( getline(input, line) ){
517     int pos = 0;
518     if ( string::npos != line.find("discount:")) {
519       pos = line.find(": ") + 2;
520       string s = line.substr(pos);
521       istringstream(s) >> pomdp.discount;
522     } else if ( string::npos != line.find("states:") ) {
523       pos = line.find(": ") + 2;
524       string s = line.substr(pos);
525       istringstream(s) >> pomdp.states;
526     } else if ( string::npos != line.find("actions:") ) {
527       pos = line.find(": ") + 2;
528       string s = line.substr(pos);
529       istringstream(s) >> pomdp.actions;
530     } else if ( string::npos != line.find("observations:") ) {
531       pos = line.find(": ") + 2;
532       string s = line.substr(pos);
533       istringstream(s) >> pomdp.observations;
534     } else if( string::npos != line.find("T") ){
535
536       if (-1 == pomdp.states || -1 == pomdp.observations || -1 == pomdp.
            actions) {
537         printf("Load error - missing information");
538         return pomdp;
539       }
540
541       pomdp.T = new float[pomdp.actions * pomdp.states * pomdp.states];
542
543       int index = 0;
544       // For the number of lines
545       for (int i = 0; i < pomdp.actions * pomdp.states; ++i) {
546         getline(input, line);
547         vector<string> tmp = split(line, ',');
548
549         for (int j = 0; j < pomdp.states; ++j) {
550           istringstream(tmp[j]) >> pomdp.T[index++];
551         }
552       }
553     } else if( string::npos != line.find("O") ){
554
555       if (-1 == pomdp.states || -1 == pomdp.observations || -1 == pomdp.
            actions) {
556         printf("Load error - missing information");
557         return pomdp;
558       }
559
560       pomdp.O = new float[pomdp.actions * pomdp.states * pomdp.observations];
```

```
561
562       int index = 0;
563       // For the number of lines
564       for (int i = 0; i < pomdp.actions * pomdp.states; ++i) {
565         getline(input, line);
566         vector<string> tmp = split(line, ',');
567
568         for (int j = 0; j < pomdp.observations; ++j) {
569           istringstream(tmp[j]) >> pomdp.O[index++];
570         }
571       }
572     } else if( string::npos != line.find("R") ){
573
574       if (-1 == pomdp.states || -1 == pomdp.observations || -1 == pomdp.
            actions) {
575         printf("Load error - missing information");
576         return pomdp;
577       }
578
579       pomdp.R = new float[pomdp.actions * pomdp.states];
580
581       int index = 0;
582       // For the number of lines
583       for (int i = 0; i < pomdp.actions; ++i) {
584         getline(input, line);
585         vector<string> tmp = split(line, ',');
586
587         for (int j = 0; j < pomdp.states; ++j) {
588           istringstream(tmp[j]) >> pomdp.R[index++];
589         }
590       }
591     }
592   }
593
594   /*
595    * Setup initial hyperplane value.
596    */
597   double min = pomdp.R[0];
598
599   for (int action = 0; action < pomdp.actions; ++action) {
600     for (int state = 0; state < pomdp.states; ++state) {
601       if ( pomdp.R[ pomdp.states * action + state ] < min ) {
602         min = pomdp.R[ pomdp.states * action + state ];
603       }
604     }
605   }
606
607   pomdp.initial_value = 0;
608   if (1.0 != pomdp.discount) {
609     pomdp.initial_value = (1/(1-pomdp.discount))*min;
610   }
611   printf("initial value: %f\n", pomdp.initial_value);
612
613   /* Allocate a vector of pointers */
614   pomdp.V = new float*[BELIEF_POINT_COUNT];
615   pomdp.Vtmp = new float*[BELIEF_POINT_COUNT];
616
617   for (int b = 0; b < BELIEF_POINT_COUNT; ++b) {
618
619     /* Allocate a hyperplane vector for each belief point. */
620     pomdp.V[b] = new float[pomdp.states];
621     pomdp.Vtmp[b] = new float[pomdp.states];
622
623     for (int state = 0; state < pomdp.states; ++state) {
624       pomdp.V[0][state] = pomdp.initial_value;
```

```
625       pomdp.Vtmp[0][state] = pomdp.initial_value;
626     }
627   }
628   // In the begnining only one uniq hyperplane excist
629   pomdp.belief_point_count = BELIEF_POINT_COUNT;
630
631   return pomdp;
632 }
633
634 double dist(float *a, float *b, int length){
635   double result = 0;
636   for (int i = 0; i < length; ++i) {
637     result += (a[i]-b[i])*(a[i]-b[i]);
638   }
639   return sqrt(result);
640 }
641
642 void vector_print(float *p, int length){
643
644   for (int i = 0; i < length; ++i) {
645     printf("%1.20f", p[i]);
646   }
647   printf("\n");
648 }
649
650 int sum_to_one(float *p, int length){
651   double sum = 0;
652   for (int i = 0; i < length; ++i) {
653     sum += p[i];
654   }
655   if ( fabs(1.0 - sum) > EPSILON ) {
656     return 0;
657   }
658   return 1;
659 }
660
661 /*
662  * count:   number of belief points to generate.
663  * states:  number of states, thus number of elements in each belief point
       vector.
664  */
665 float ** init_B_uniform( int count, int states ){
666
667   float **B = new float*[count];
668   // Number of generated belief points.
669   int bp;
670   // Setup the extreme belief points.
671   for (bp = 0; bp < states; ++bp) {
672     B[bp] = new float[states];
673     memset(B[bp], 0, states*sizeof(float));
674     B[bp][bp] = 1.0;
675   }
676
677   // The distance matrix contains the distances between all belief points.
678   double** dist_matrix = new double*[count];
679   for(int i = 0; i < count; ++i){
680     dist_matrix[i] = new double[count];
681   }
682
683   for (int i = 0; i < count; ++i) {
684     for (int j = 0; j < count; ++j) {
685       dist_matrix[i][j] = 0;
686     }
687   }
688
```

```
689   double max_dist = 0;
690   int index1 = -1, index2 = -1;
691
692   // Calculate the initial distances and find the longest distance between
          two points.
693   for (int i = 0; i < states; ++i) {
694     for (int j = i+1 ; j < states; ++j) {
695       dist_matrix[i][j] = dist(B[i], B[j], states);
696
697       if ( dist_matrix[i][j] > max_dist ) {
698         max_dist = dist_matrix[i][j];
699         index1 = i;
700         index2 = j;
701       }
702     }
703   }
704
705   for (; bp < count; ++bp) {
706
707     // Make a new belief point.
708     B[bp] = new float[states];
709     // Set the belief point value between the two points with the longest
            distance.
710     for (int k = 0; k < states; ++k) {
711       B[bp][k] = (B[index1][k] + B[index2][k]) / 2.0;
712     }
713
714     // Sanity test on the vectors sum, which always must be 1.
715     if (!sum_to_one(B[bp], states)) {
716       printf("Error - Belief point between points at %d x %d, don't sum to
              1!\n", index1, index2);
717     }
718
719     // Calaulate the distances from the new belief point to all other points.
720     for (int i = 0; i < bp; ++i) {
721       dist_matrix[i][bp] = dist(B[bp], B[i], states);
722     }
723
724     // Null the old longest distance in the distance matrix.
725     dist_matrix[index1][index2] = 0;
726     // Find the new longest distance.
727     max_dist = 0;
728     for (int i = 0; i <= bp; ++i) {
729       for (int j = i+1 ; j <= bp; ++j) {
730         if ( dist_matrix[i][j] > max_dist ) {
731           max_dist = dist_matrix[i][j];
732           index1 = i; index2 = j;
733         }
734       }
735     }
736   }
737   return B;
738 }
739
740 /*
741  * count:   number of belief points to generate.
742  * states:  number of states, thus number of elements in each belief point
             vector.
743  */
744 float ** init_B_random( int count, int states ){
745
746   float **B = new float*[count];
747
748   // Number of generated belief points.
749   int bp;
```

```
750   // Setup the extreme belief points.
751   for (bp = 0; bp < states; ++bp) {
752     B[bp] = new float[states];
753     memset(B[bp], 0, states*sizeof(float));
754     B[bp][bp] = 1;
755   }
756
757   // Initialize random seed
758   srand(42);
759   for (; bp < count; ++bp) {
760     // Make a new belief point
761     B[bp] = new float[states];
762     float sum = 0;
763     // Assign random numbers to each element in a belief point.
764     for (int j = 0; j < states; ++j) {
765       // Make a random number.
766       B[bp][j] = rand() % 1000000 + 1;
767       sum += B[bp][j];
768     }
769
770     // Normalize the vector to sum to 1.0.
771     for (int j = 0; j < states; ++j) {
772       B[bp][j] /= sum;
773     }
774     // Sanity test on the vectors sum, which always must be 1.0
775     if (!sum_to_one(B[bp], states)) {
776       // Redo the random vector.
777       bp--;
778     }
779   }
780   return B;
781 }
782
783 int vector_equal(float *a, float *b, int length){
784   int hit = 1;
785   for (int i = 0; i < length; ++i) {
786     // if just one element is different, go on.
787     if ( fabs(a[i] - b[i]= > 0.0001) {
788       hit = 0;
789       break;
790     }
791   }
792   return hit;
793 }
794
795 float dot(float *a, float *b, int length){
796   float sum = 0;
797   for (int i = 0; i < length; ++i) {
798     sum += a[i] * b[i];
799   }
800   return sum;
801 }
802
803
804 float *init_B_uniform_vector(int count, int states){
805   float **BB = init_B_uniform(count, states);
806   float *B = new float[count * states];
807   for (int i = 0; i < count; ++i){
808     for (int j = 0; j < states; ++j){
809       B[states * i + j] = BB[i][j];
810     }
811   }
812   return B;
813 }
814
```

```
815 float *init_B_random_vector(int count, int states){
816   printf("Generating␣random␣vectors\n");
817   float **BB = init_B_random(count, states);
818   float *B = new float[count * states];
819   for (int i = 0; i < count; ++i){
820     for (int j = 0; j < states; ++j){
821       B[states * i + j] = BB[i][j];
822     }
823   }
824   printf("Done␣generating␣vectors\n");
825   return B;
826 }
827
828 void pbvi_base(char *path){
829
830   POMDP pomdp = load_pomdp( path );
831
832   //float *B = init_B_uniform_vector(BELIEF_POINT_COUNT, pomdp.states);
833   float *B = init_B_random_vector(BELIEF_POINT_COUNT, pomdp.states);
834
835   float **dev_V;
836   cudaMalloc((void**) &dev_V, BELIEF_POINT_COUNT * sizeof(float*));
837
838   float **dev_Vtmp;
839   cudaMalloc((void**) &dev_Vtmp, BELIEF_POINT_COUNT * sizeof(float*));
840
841
842   /*
843    *  Allocate and transfer R, O, T, B and the action data to global memory.
844    */
845   float *dev_R;
846   cudaMalloc((void**) &dev_R, pomdp.actions * pomdp.states * sizeof(float) );
847   cudaMemcpy(dev_R, pomdp.R, pomdp.actions * pomdp.states * sizeof(float),
          cudaMemcpyHostToDevice);
848
849   float *dev_O;
850   cudaMalloc((void**) &dev_O, pomdp.actions * pomdp.states * pomdp.
          observations * sizeof(float) );
851   cudaMemcpy(dev_O, pomdp.O, pomdp.actions * pomdp.states * pomdp.
          observations * sizeof(float), cudaMemcpyHostToDevice);
852
853   float *dev_T;
854   cudaMalloc((void**) &dev_T, pomdp.actions * pomdp.states * pomdp.states *
          sizeof(float) );
855   cudaMemcpy(dev_T, pomdp.T, pomdp.actions * pomdp.states * pomdp.states *
          sizeof(float), cudaMemcpyHostToDevice);
856
857   float *dev_B;
858   cudaMalloc((void**) &dev_B, pomdp.belief_point_count * pomdp.states *
          sizeof(float) );
859   cudaMemcpy(dev_B, B, pomdp.belief_point_count * pomdp.states * sizeof(float
          ), cudaMemcpyHostToDevice);
860
861   int *dev_best_action;
862   cudaMalloc((void**) &dev_best_action, BELIEF_POINT_COUNT * sizeof(int));
863
864   init_pbvi<<<BLOCKS, THREADS_PR_BLOCK>>>(dev_V, dev_Vtmp, pomdp.
          initial_value, pomdp.states);
865   cudaDeviceSynchronize();
866
867   for (int i = 0; i < ITERATIONS; ++i) {
868
869     pbvi<<<BLOCKS, THREADS_PR_BLOCK>>>(
870       dev_V, dev_Vtmp,
871       dev_T, dev_O, dev_R, dev_B,
```

```
872         dev_best_action,
873         pomdp.states,
874         pomdp.observations,
875         pomdp.actions,
876         pomdp.discount);
877       cudaDeviceSynchronize();
878       swap(dev_V, dev_Vtmp);
879     }
880
881     float *dev_result;
882     cudaMalloc((void**) &dev_result,  STATE_COUNT * BELIEF_POINT_COUNT * sizeof
          (float) );
883
884     pbvi_get<<<BLOCKS, THREADS_PR_BLOCK>>>(dev_V, dev_result, pomdp.states);
885
886     float *result = new float[STATE_COUNT * BELIEF_POINT_COUNT];
887     int *best_action = new int[BELIEF_POINT_COUNT];
888
889     cudaMemcpy(result, dev_result, STATE_COUNT * BELIEF_POINT_COUNT * sizeof(
          float), cudaMemcpyDeviceToHost);
890     cudaMemcpy(best_action, dev_best_action, BELIEF_POINT_COUNT * sizeof(int),
          cudaMemcpyDeviceToHost);
891
892     cudaFree(dev_best_action);
893     cudaFree(dev_result);
894
895     cudaFree(dev_V);
896     cudaFree(dev_Vtmp);
897     cudaFree(dev_R);
898     cudaFree(dev_O);
899     cudaFree(dev_T);
900     cudaFree(dev_B);
901 }
902
903 void pbvi_purge(char *path){
904
905     POMDP pomdp = load_pomdp( path );
906
907     //float *B = init_B_uniform_vector(BELIEF_POINT_COUNT, pomdp.states);
908     float *B = init_B_random_vector(BELIEF_POINT_COUNT, pomdp.states);
909
910     float **dev_V;
911     cudaMalloc((void**) &dev_V, BELIEF_POINT_COUNT * sizeof(float*));
912
913     float **dev_Vtmp;
914     cudaMalloc((void**) &dev_Vtmp, BELIEF_POINT_COUNT * sizeof(float*));
915
916     /*
917      *  Allocate and transfer R, O, T, B and the action data to global memory.
918      */
919     float *dev_R;
920     cudaMalloc((void**) &dev_R, pomdp.actions * pomdp.states * sizeof(float) );
921     cudaMemcpy(dev_R, pomdp.R, pomdp.actions * pomdp.states * sizeof(float),
          cudaMemcpyHostToDevice);
922
923     float *dev_O;
924     cudaMalloc((void**) &dev_O, pomdp.actions * pomdp.states * pomdp.
          observations * sizeof(float) );
925     cudaMemcpy(dev_O, pomdp.O, pomdp.actions * pomdp.states * pomdp.
          observations * sizeof(float), cudaMemcpyHostToDevice);
926
927     float *dev_T;
928     cudaMalloc((void**) &dev_T, pomdp.actions * pomdp.states * pomdp.states *
          sizeof(float) );
```

```
929   cudaMemcpy(dev_T, pomdp.T, pomdp.actions * pomdp.states * pomdp.states *
          sizeof(float), cudaMemcpyHostToDevice);
930
931   float *dev_B;
932   cudaMalloc((void**) &dev_B, pomdp.belief_point_count * pomdp.states *
          sizeof(float) );
933   cudaMemcpy(dev_B, B, pomdp.belief_point_count * pomdp.states * sizeof(float
          ), cudaMemcpyHostToDevice);
934
935   int *dev_best_action;
936   cudaMalloc((void**) &dev_best_action, BELIEF_POINT_COUNT * sizeof(int));
937
938   init_pbvi<<<BLOCKS, BELIEF_POINT_COUNT / BLOCKS>>>(dev_V, dev_Vtmp, pomdp.
          initial_value, pomdp.states);
939   cudaDeviceSynchronize();
940
941   /* Initial purge setting the unique_belief_points to 1. */
942   pbvi_purge<<<1,1>>>(dev_V, pomdp.states);
943
944   cudaDeviceSynchronize();
945
946   for (int i = 0; i < ITERATIONS; ++i) {
947
948     pbvi<<<BLOCKS, BELIEF_POINT_COUNT / BLOCKS>>>(
949       dev_V, dev_Vtmp,
950       dev_T, dev_O, dev_R, dev_B,
951       dev_best_action,
952       pomdp.states,
953       pomdp.observations,
954       pomdp.actions,
955       pomdp.discount);
956     cudaDeviceSynchronize();
957
958     pbvi_purge<<<1,1>>>(dev_Vtmp, pomdp.states);
959
960     cudaDeviceSynchronize();
961     swap(dev_V, dev_Vtmp);
962   }
963
964   float *dev_result;
965   cudaMalloc((void**) &dev_result,  STATE_COUNT * BELIEF_POINT_COUNT * sizeof
          (float) );
966
967   pbvi_get<<<BLOCKS, BELIEF_POINT_COUNT / BLOCKS>>>(dev_V, dev_result, pomdp.
          states);
968
969   float *result = new float[STATE_COUNT * BELIEF_POINT_COUNT];
970   int *best_action = new int[BELIEF_POINT_COUNT];
971
972   cudaMemcpy(result, dev_result, STATE_COUNT * BELIEF_POINT_COUNT * sizeof(
          float), cudaMemcpyDeviceToHost);
973   cudaMemcpy(best_action, dev_best_action, BELIEF_POINT_COUNT * sizeof(int),
          cudaMemcpyDeviceToHost);
974
975   cudaFree(dev_best_action);
976   cudaFree(dev_result);
977
978   cudaFree(dev_V);
979   cudaFree(dev_Vtmp);
980   cudaFree(dev_R);
981   cudaFree(dev_O);
982   cudaFree(dev_T);
983   cudaFree(dev_B);
984 }
985
```

```
986 void pbvi_shared_memory(char *path){
987
988   POMDP pomdp = load_pomdp( path );
989   float *B = init_B_random_vector(BELIEF_POINT_COUNT, pomdp.states);
990
991   float **dev_V;
992   cudaMalloc((void**) &dev_V, BELIEF_POINT_COUNT * sizeof(float*));
993
994   float **dev_Vtmp;
995   cudaMalloc((void**) &dev_Vtmp, BELIEF_POINT_COUNT * sizeof(float*));
996
997   /*
998    *  Allocate and transfer R, O, T, B and the action data to global memory.
999    */
1000  float *dev_R;
1001  cudaMalloc((void**) &dev_R, pomdp.actions * pomdp.states * sizeof(float) );
1002  cudaMemcpy(dev_R, pomdp.R, pomdp.actions * pomdp.states * sizeof(float),
        cudaMemcpyHostToDevice);
1003
1004  float *dev_O;
1005  cudaMalloc((void**) &dev_O, pomdp.actions * pomdp.states * pomdp.
        observations * sizeof(float) );
1006  cudaMemcpy(dev_O, pomdp.O, pomdp.actions * pomdp.states * pomdp.
        observations * sizeof(float), cudaMemcpyHostToDevice);
1007
1008  float *dev_T;
1009  cudaMalloc((void**) &dev_T, pomdp.actions * pomdp.states * pomdp.states *
        sizeof(float) );
1010  cudaMemcpy(dev_T, pomdp.T, pomdp.actions * pomdp.states * pomdp.states *
        sizeof(float), cudaMemcpyHostToDevice);
1011
1012  float *dev_B;
1013  cudaMalloc((void**) &dev_B, pomdp.belief_point_count * pomdp.states *
        sizeof(float) );
1014  cudaMemcpy(dev_B, B, pomdp.belief_point_count * pomdp.states * sizeof(float
        ), cudaMemcpyHostToDevice);
1015
1016  int *dev_best_action;
1017  cudaMalloc((void**) &dev_best_action, BELIEF_POINT_COUNT * sizeof(int));
1018
1019  init_pbvi<<<BLOCKS, BELIEF_POINT_COUNT / BLOCKS>>>(dev_V, dev_Vtmp, pomdp.
        initial_value, pomdp.states);
1020
1021  cudaDeviceSynchronize();
1022
1023  /* Initial purge setting the unique_belief_points to 1. */
1024  pbvi_purge<<<1,1>>>(dev_V, pomdp.states);
1025  cudaDeviceSynchronize();
1026
1027  timeval start_time, end_time;
1028  double elapsedTime;
1029  // Start timer
1030  gettimeofday(&start_time, NULL);
1031
1032  for (int i = 0; i < ITERATIONS; ++i) {
1033
1034    pbvi_shared_memory<<<BLOCKS, THREADS_PR_BLOCK>>>(
1035      dev_V, dev_Vtmp,
1036      dev_T, dev_O, dev_R, dev_B,
1037      dev_best_action,
1038      pomdp.states,
1039      pomdp.observations,
1040      pomdp.actions,
1041      pomdp.discount);
1042    cudaDeviceSynchronize();
```

```
1043
1044      pbvi_purge<<<1,1>>>(dev_Vtmp, pomdp.states);
1045
1046      cudaDeviceSynchronize();
1047      swap(dev_V, dev_Vtmp);
1048    }
1049
1050    float *dev_result;
1051    cudaMalloc((void**) &dev_result,  STATE_COUNT * BELIEF_POINT_COUNT * sizeof
          (float) );
1052
1053    pbvi_get<<<BLOCKS, BELIEF_POINT_COUNT / BLOCKS>>>(dev_V, dev_result, pomdp.
          states);
1054
1055    float *result = new float[STATE_COUNT * BELIEF_POINT_COUNT];
1056    int *best_action = new int[BELIEF_POINT_COUNT];
1057
1058    cudaMemcpy(result, dev_result, STATE_COUNT * BELIEF_POINT_COUNT * sizeof(
          float), cudaMemcpyDeviceToHost);
1059    cudaMemcpy(best_action, dev_best_action, BELIEF_POINT_COUNT * sizeof(int),
          cudaMemcpyDeviceToHost);
1060
1061    cudaFree(dev_best_action);
1062    cudaFree(dev_result);
1063
1064    cudaFree(dev_V);
1065    cudaFree(dev_Vtmp);
1066    cudaFree(dev_R);
1067    cudaFree(dev_O);
1068    cudaFree(dev_T);
1069    cudaFree(dev_B);
1070  }
1071
1072
1073  int main(int argc, char **argv) {
1074    printf("Welcome␣to␣CUDA␣PBVI:\n");
1075
1076    timeval start_time, end_time;
1077    float elapsedTime;
1078    // Start timer
1079    gettimeofday(&start_time, NULL);
1080
1081  //  pbvi_base( "fps.pop" );
1082  //  pbvi_purge( "fps.pop" );
1083    pbvi_shared_memory( "fps.pop" );
1084
1085
1086    // Stop timer
1087    gettimeofday(&end_time, NULL);
1088    elapsedTime = (end_time.tv_sec - start_time.tv_sec) * 1000.0;
1089    elapsedTime += (end_time.tv_usec - start_time.tv_usec) / 1000.0;
1090
1091    printf("\n\n%f␣ms.\n\n", elapsedTime );
1092    printf("The␣end\n");
1093    return 0;
1094  }
```

A.3 MATLAB POMDP Parser

Listing A.3: MATLAB POMDP Parser

```
function pomdp = parser(filename)


% Read and parse the POMDP without use of sparse matrices.
pomdp = readPOMDP(strcat(filename, '.pomdp'), 0);

% Transpose the observation matrix, to be indexed by:
% action x state x observation, which correspond to the way PERSUS notation
% reads the matrix.
pomdp.observation = permute(pomdp.observation, [1 3 2])

% Transpose the transition matrix to action x state x state
pomdp.transition = permute(pomdp.transition, [2 1 3])


% Transform the reward matrix to the be indexed by: action x state
% Which is the weighted max reward given the execution in a given state.
%
% Multiplying the transition probability with the reward, gives the
% expected weighted utility:
% reward .* transition
%
% By summing over each row, gives the expected utility
% for executing an action in a given state:
% sum(reward .* transition, 2)
%
% The reshape will eliminate the empty corrdinates.
pomdp.reward = reshape( sum(pomdp.reward3.*pomdp.transition,2), pomdp.
    nrStates, pomdp.nrActions)'

% Clear old files
delete(strcat(filename,'.pop'));

fid = fopen(strcat(filename,'.pop'),'at');

fprintf(fid,'states: %d\n',pomdp.nrStates);
fprintf(fid,'actions: %d\n',pomdp.nrActions);
fprintf(fid,'observations: %d\n',pomdp.nrObservations);
fprintf(fid,'discount: %f\n',pomdp.gamma);

fprintf(fid,'T\n');

% Save the observations
for j=1:pomdp.nrActions
   dlmwrite(strcat(filename,'.pop'),pomdp.transition(:,:,j),'precision','
    %10.5f','-append')
end


fprintf(fid,'O\n');
% Save the observations
for j=1:pomdp.nrActions
   dlmwrite(strcat(filename,'.pop'),pomdp.observation(:,:,j),'precision','
    %10.5f','-append')
end

fprintf(fid,'R\n');
% Save the observations
dlmwrite(strcat(filename,'.pop'),pomdp.reward3,'precision','%10.5f','-
    append')
```

```
57
58 fclose(fid);
```

A.4 POMDP Problems

Listing A.4: fps.pomdp

```
discount: 0.95
values: reward
states: off on_pwr on_nopwr
actions: on off sense wait
observations: none on off

start:
uniform

T: on
0.2 0.6 0.2
0 1 0
0 0 1

T: off
1 0 0
1 0 0
1 0 0

T: sense
1 0 0
0 1 0
0 0 1

T: wait
0.8 0.1 0.1
0.1 0.8 0.1
0.1 0.1 0.8

O: on
1 0 0
1 0 0
1 0 0

O: off
1 0 0
1 0 0
1 0 0

O: sense
0 0 1
0 1 0
0 0 1

O: wait
1 0 0
1 0 0
1 0 0

R: * : * : * : * -1
R: wait : on_pwr : on_pwr : * 50
```

Listing A.5: maze.pomdp

```
################################################################################
# FILENAME: 4x3.POMDP

# Stuart Russell's 4x3 maze
#
# The maze looks like this:
#
#
#    ######
#    #    +#
#    # # -#
#    #     #
#    ######
#
# The + indicates a reward of 1.0, the - a penalty of -1.0.
# The # in the middle of the maze is an obstruction.
# Rewards and penalties are associated with states, not actions.
# The default reward/penalty is -0.04.
# There is no discounting, but a there is an absorbing state that
# + and - transition to automatically.  The absorbing state cannot be exited.
#
# States are numbered from left to right:
#
# 0   1   2   3
# 4       5   6
# 7   8   9   10
#
# I removed the absorbing state
#
# The actions, NSEW, have the expected result 80% of the time, and
# transition in a direction perpendicular to the intended on with a 10%
# probability for each direction.  Movement into a wall returns the agent
# to its original state.
#
# Observation is limited to two wall detectors that can detect when a
# a wall is to the left or right.  This gives the following possible
# observations:
#
# left, right, neither, both, good, bad, and absorb
#
# good = +1 reward, bad = -1 penalty,

discount: 0.95
values: reward
states: 11
actions: n s e w
observations: left right neither both good bad

start:
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
    0.111111 0.111111

T: n
0.9 0.1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.1 0.8 0.1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.1 0.8 0.1 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
    0.111111 0.111111
0.8 0.0 0.0 0.0 0.2 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.8 0.0 0.0 0.1 0.1 0.0 0.0 0.0 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
    0.111111 0.111111
0.0 0.0 0.0 0.0 0.8 0.0 0.0 0.1 0.1 0.0 0.0
```

```
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.8 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.8 0.0 0.0 0.1 0.0 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.8 0.0 0.0 0.1 0.1

T: s
0.1 0.1 0.0 0.0 0.8 0.0 0.0 0.0 0.0 0.0 0.0
0.1 0.8 0.1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.1 0.0 0.1 0.0 0.8 0.0 0.0 0.0 0.0 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
     0.111111 0.111111
0.0 0.0 0.0 0.0 0.2 0.0 0.0 0.8 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.1 0.1 0.0 0.0 0.8 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
     0.111111 0.111111
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.9 0.1 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.8 0.1 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.8 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.9

T: e
0.1 0.8 0.0 0.0 0.1 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.2 0.8 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.1 0.8 0.0 0.1 0.0 0.0 0.0 0.0 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
     0.111111 0.111111
0.1 0.0 0.0 0.0 0.8 0.0 0.0 0.1 0.0 0.0 0.0
0.0 0.0 0.1 0.0 0.0 0.0 0.8 0.0 0.0 0.1 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
     0.111111 0.111111
0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.1 0.8 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2 0.8 0.0
0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.1 0.0 0.8
0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.0 0.9

T: w
0.9 0.0 0.0 0.0 0.1 0.0 0.0 0.0 0.0 0.0 0.0
0.8 0.2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.8 0.1 0.0 0.0 0.1 0.0 0.0 0.0 0.0 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
     0.111111 0.111111
0.1 0.0 0.0 0.0 0.8 0.0 0.0 0.1 0.0 0.0 0.0
0.0 0.0 0.1 0.0 0.0 0.8 0.0 0.0 0.0 0.1 0.0
0.111111 0.111111 0.111111 0.0 0.111111 0.111111 0.0 0.111112 0.111111
     0.111111 0.111111
0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.9 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.8 0.2 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.8 0.1
0.0 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.8 0.1

O: *
1.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 1.0 0.0
0.0 0.0 0.0 1.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 1.0
1.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0

R: * : 0 : * : * -0.04
R: * : 1 : * : * -0.04
R: * : 2 : * : * -0.04
```

```
R: * : 3 : * : * 1.0
R: * : 4 : * : * -0.04
R: * : 5 : * : * -0.04
R: * : 6 : * : * -1.0
R: * : 7 : * : * -0.04
R: * : 8 : * : * -0.04
R: * : 9 : * : * -0.04
R: * : 10 : * : * -0.04
```

Bibliography

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[Cas05] A. R. Cassandra. POMDP-solver v5.3. http://www.pomdp.org/pomdp/code/index.shtml, 2005. [Online; accessed July-2013].

[Cas09] A. R. Cassandra. POMDP File Format. http://www.pomdp.org/pomdp/code/pomdp-file-spec.shtml, 2009. [Online; accessed July-2013].

[Cas13] A. R. Cassandra. POMDP problems. http://www.pomdp.org/pomdp/examples/index.shtml, 2013. [Online; accessed July-2013].

[GK13] AMD George Kyriazis. Heterogeneous System Architecture: A Technical Review. developer.amd.com/wordpress/media/2012/10/hsa10.pdf, 2013. [Online; accessed July-2013].

[Khr13] Khronos. OpenCL Toolkit Documentation. http://www.khronos.org/opencl, 2013. [Online; accessed July-2013].

[Mar54] A.A. Markov. *Theory of Algorithms*. Works of the mathematical institute Imeni V. A. Steklov. Academy of Sciences of the USSR, 1954.

[Noe11] Dennis Noer. Improved Software Implementation of DES Using CUDA and OpenCL, 2011.

[NVI13a] NVIDIA. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/index.html, 2013. [Online; accessed July-2013].

[NVI13b] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture. http://goo.gl/vmRQN8, 2013. [Online; accessed July-2013].

[PGT03] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for pomdps. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1025 – 1032, August 2003.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[Spa03] Matthijs Spaan. POMDP parser. http://staff.science.uva.nl/~mtjspaan/software/pomdp/, 2003. [Online; accessed July-2013].

[Stu09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Dec 1, 2009.

[ZZ01] Nevin L. Zhang and Weihong Zhang. Speeding up the convergence of value iteration in partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 14:2001, 2001.