Solving Large MDPs with Value-Iteration on Multi-GPU Systems

Yantao Luo YL5929@NYU.EDU
New York University

Paul Koettering PMK2054@NYU.EDU
New York University

Abstract

Markov Decision Processes (MDPs) are a powerful decision making framework however modern MDP problems can be extremely large with respect to the size of the state and action spaces. Graphical Processing Units (GPUs) can improve algorithms efficiency by introducing parallel computation capabilities. Furthermore, there has been significant recent progress in developing methods for systems with multiple GPUs which enhance the memory size and computational performance. We develop a novel method and investigate the performance of using multi-GPU systems to solve MDPs through Value Iteration. We compare the performance to both single GPU systems and CPU-only systems.

1. Introduction

General Purpose Graphical Processing Units (GPGPUs) have become widely used in scientific and research applications recently as the ability to perform massively parallel computations offers significant performance enhancement for a range of tasks (Nickolls & Dally, 2010). Applications in artificial intelligence (Wang et al., 2019), graph algorithms (Hong et al., 2011) and big data analytics (Singh & Reddy, 2014) have been found for GPUs.

Markov Decision Processes (MDPs) (Puterman, 1994) are fundamental to many modern artificial intelligence applications such as reinforcement learning (Buşoniu et al., 2008) (Szepesvari, 2010) and have long been using for controlling complex systems in a number of ways (Benini et al., 1998). The computational requirements of solving MDPs however can be prohibitively large depending on the size of the action and state space (Littman et al., 1995). Developing techniques to make efficient, compact and optimized MDPs has been the focus of many studies. MDPs have become the standard formalism of sequential decision making as a multitude of problems such as game-playing and robot control can be modelled in terms of MDPs.

Applications benefit most from being run on a GPU tend to be those which are computationally intensive, require many independent similar computations and have a very large problem size. MDPs satisfy all of these requirements and therefore we think they are well suited to be parallelized. Solving MDPs requires large memory requirements in order to store the reward and transition function arrays, we therefore believe that using multi-GPU systems which have enhanced memory scales are particularly suitable.

The main contributions of this project are therefore:

- Proposed a novel method for solving large MDPs with both single and multi GPU systems.
- Analysed the performance of the algorithm through extensive experimentation.
- Identified classes of MDPs which are suited to our multi-GPU algorithm.
- We provide our full code with instructions on how to run experiments `https://github.com/luoyantao99/The-Path-to-Multi-GPU-Systems`

2. Problem Definition and Challenges

In this section we give some technical preliminaries of our problem as well as outline some of the challenges current approaches face when trying to solve MDPs.

2.1. MDPs

A MDP is a discrete-time stochastic control process which is used to model decision making where the outcome of actions is partly random. Due to the interventions of the

Final Project for Graphical Processing Units course at NYU.

decision maker and the dynamics of the system, the system evolves through a sequence of state transitions. We can describe a MDP by a 5-tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ where $\mathcal{S}$ is the set of all possible states of the system, $\mathcal{A}$ are the set of all actions which the decision maker can choose, $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function for a given state-action pair which describes the probability of transitioning to any other state, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbf{R}$ is the reward function for a given state, action and successive state which indicates the reward that the agents receives for a given transition, and finally γ is the discount factor which determines the rate of decrease of the important between present and future rewards. In this way we see that $T(s, a, s') = P(s_{t+1}|s_t = s, a_t = a)$ and the system is dynamic. Our goal is to find a policy which maximizes the expected sum of rewards:

$$\mathbf{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s'_t)]$$

Here the expectation is over the transition probability function. The deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ determines which action to take in a given state of the environment. The optimal policy π^* is such that it maximizes the expected sum of rewards.

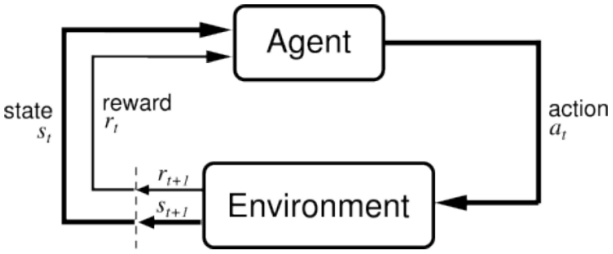

Figure 1. Diagram of a Markov Decision Process

2.2. Value Iteration

A common and simple approach to solving MDPs is a dynamic programming approach called Value Iteration (Bellman, 1957a) (Bellman, 1957b) (often also called Backward Induction or Bellman Update). In this approach we do not use the policy itself but directly learn a value function for each state, which represents the expected sum of rewards accumulated when starting in the given state and acting optimally. The Value Iteration algorithm begins with an initial value for each state and at each iteration a new value function is generated according to the following update rule:

$$V^*_{i+1} \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*_i(s')]$$

These updates are performed until the values convergence and the right hand side equals the left. At this point we can easily find the optimal policy by following:

$$\pi^*(s) = \text{argmax}_{a \in A} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')]$$

In this way the infinite horizon optimal policy is stationary, such that the optimal action at each state is the same at all times. This allows for efficient storage of the value array. The challenge for algorithms solving extremely large MDPs is that even just storing the transition probability and reward function arrays can be difficult. Furthermore, solving MDPs using the value iteration algorithm has a $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|N)$ time complexity, where N is the number of iterations required until convergence. When the size of the state and action space are extremely large this time complexity is prohibitive. The transition and reward function are known beforehand (this separates this problem from reinforcement learning setup where the transition and reward functions are unknown). Dealing with an array of this size when the action and state spaces are very large introduces difficulties for GPU algorithms which have limited memory capacity.

2.3. Tree Reduction

One of the most common parallel computation pattern is tree reduction which we can use to find the maximum of an array in $\mathcal{O}(\log K)$ time, where K is the number of elements in the array. At each step, every active thread performs the maximum operation with another element value and the thread array value. In this way the number of active threads halves each step. We can increase the performance of the kernel by keeping active threads consecutive and therefore avoiding kernel thread divergence until the number of threads falls below 32 (which is the size of the GPU warp).

2.4. GPU Memory Systems

Modern GPUs can achieve superior performance on many demanding applications when data and algorithms are carefully adapted to the problem at hand and the programming model. In a GPU every thread in a warp, a block of 32, executes the same instruction according to the single-instruction-multiple-data (SIMD) paradigm. Device memory on a GPU is hierarchically structures and data transfers between CPU and GPU memory is possible is low latency. The GPU has device global memory which can be accesses by each thread however this is inefficient, and so often complex and intricate schemes are devised in order to facilitate efficient algorithms. Systems with multiple GPUs have increased parallel computation capability however memory

operations can become more complicated and reduce performance if not managed correctly. One strategy developed by NVIDIA to migrate data on demand efficiently is called Unified Memory, something we take advantage of in our multi-GPU algorithm. Unified Memory creates a pool of shared memory between the CPU and GPU which is accessible by both with single pointers. This allows for significant ease of programming since the data is automatically migrated under the hood and without the programmers control to the different devices and the CPU.

3. Related Work

3.1. Solving Large MDPs

Solving large MDPs has a long history and many approaches have been suggested by researchers over the years. The authors of (Meuleau et al., 1998) aimed to avoid the large action and state space by breaking up the full MDP into separable tasks. Furthermore, (Dean & Lin, 1995) investigated methods to decompose MDPs into multiple local problems and then combing the local solutions once they have been solved separately. Partially observable MDPs provide a suitable framework for planning of autonomous robots and much work has been done into solving these problems when the action and state space is large (Kurniawati et al., 2008) (Smith & Simmons, 2005). Many of the algorithms proposed exploit some particular structure of the partially observable MDP, such as the Bounded Policy Iteration algorithm (Poupart, 2005). Value Iteration algorithms have also been suggested to solve large MDPs on CPUs (Jain & Sahni, 2020). Alternatively, the authors of (Hoey et al., 1999) developed a novel value iteration algorithm that uses algebraic decision diagrams to represent value functions and policies.(Kim & Dean, 2002) and (Dean et al., 1998) focused on solving MDPs which have particularly large action spaces, and presented a set of problems types appropriate for these algorithms which are also appropriate for our project.

3.2. MDPs on GPU Systems

A new class of MDP solvers which use GPUs for parallel computation acceleration have emerged recently which offer an order of magnitude improvement over methods which only use CPUs. (Sapio et al., 2018) presents a novel algorithm called the Sparse Parallel Value Iteration which combines parallel computing with sparse linear algebra techniques. The authors of (Ruiz & Hernández, 2015) solved MDPs on GPU's through a matrix multiplication value-iteration strategy. Using the ability of the GPU to produce interactively obstacle-free paths the authors shows convergence to an optimal policy with a 90x speedup compared with CPU approaches. (Noer, 2013) used a GPU platform to reduce the computational time required for solving partially observable MDPs. All of the three above algorithms use a combination of C++ and CUDA as programming language, something we follow in this project. (Sapio et al., 2019) presented the GPU accelerated Embedded Mdp testBench (GEMBench) in order to facilitate experimental research into the use of GPUs to solve Markov Decision Proceses. This benchmark is particularly targeted towards the use of embedded GPU platforms.

4. Proposed Idea

In all of the algorithms below, we can sample the transition and reward function by accessing an $|S||A||S|$ sized array for each function. Each element in the reward function array represents the reward for a particular state-action-state input and each element in the transition function array represents the probability of transitioning to a particular state from a given state-action pair chosen by the policy.

Looking at the Sequential Value-Iteration Algorithm 1 we see that the 4 for loops result in an extremely large time complexity when the state and action spaces are large. This is the base case against which we will be testing our other algorithms against.

Algorithm 1 Sequential Value-Iteration Algorithm

$V_0^* = 0$
for $i \in$ MaxIterations **do**
  for $s \in \mathcal{S}$ **do**
    MaxA $\leftarrow 0$
    for $a \in \mathcal{A}$ **do**
      SumS $\leftarrow 0$
      for $s' \in \mathcal{S}$ **do**
        $K = T(s, a, s')[R(s, a, s') + \gamma V_i^*(s')]$
        SumS $\leftarrow$ SumS $+ K$
      end for
      if SumS $>$ MaxA **then**
        MaxA $\leftarrow$ SumS
      end if
    end for
    $V_{i+1}^*(s) \leftarrow$ MaxA
  end for
end for

In order to parallelize the algorithm we have to decide what system will be responsible for each for loop, we must take memory capabilities into account when making this decision. The algorithm we propose for the single GPU method 2 involves each thread performing the innermost summation loop and then a subsequent tree reduction step to find the maximum summation across the action space. Since it is more efficient to perform a tree reduction across a block we have two GPU kernels, the first of which finds the maximum summation for each GPU thread block and

then another which finds the maximum across all the block maximums. In this way we have a two level tree reduction algorithm. Only a subset of the full transition and reward arrays are moved to the GPU at each iteration, each of size $|\mathcal{S}||\mathcal{A}|$. The necessary memory allocating, copying and freeing CUDA calls in order to ensure efficient memory operations are included in the code but not shown in the pseudocode.

Algorithm 2 Single-GPU Value-Iteration Algorithm

```
V_0^* = 0
for i ∈ MaxIterations do
    for s ∈ S do
        Move T(s,:,:) and R(s,:,:) to GPU
        TreeReductionBlock
        Device Synchronize
        TreeReductionState
        Device Synchronize
        V_i^*(s) = Max
    end for
end for
```

For the multi-GPU algorithm 3 we make use of the unified memory capability for CPU-GPU memory operations. In this algorithm we move the full transition and reward arrays into the unified memory and we are able to directly call them in our GPU kernel function. Similar to the single GPU approach we loop over the states in order to update the value function array, except since we have multiple GPUs we can assign each GPU a subset of the states for each to run. We then perform a tree reduction similar to the single GPU case, however now we are saving the results directly into the unified memory and reading directly from the unified memory. Furthermore, since the maximum summation term of each thread block is in the unified memory for all the states, we are able to call the block level reduction together in one kernel. For this second tree reduction kernel call, the number of blocks is the size of the state space and the number threads is the number of blocks of the initial tree reduction. Notice that in this method we must call the device synchronization run time call in a loop for all the devices in the multi-GPU system after the first and second tree reduction state. This is to ensure that the GPU devices do not perform the value iteration algorithm with a value function array from a previous iteration.

Algorithm 3 Multi-GPU Value-Iteration Algorithm

```
V_0^* = 0
for i ∈ MaxIterations do
    for k ∈ |S|/NumberofDevices do
        for d ∈ NumberofDevices do
            SetDevice
            MultiTreeReductionBlock
        end for
    end for
    for d ∈ NumberofDevices do
        SetDevice
        Device Synchronize
    end for
    MultiTreeReductionState
    for d ∈ NumberofDevices do
        SetDevice
        Device Synchronize
    end for
end for
```

5. Experimental Setup

Three versions of the MDP solver were prepared for testing: CPU sequential version, single-GPU version, and multi-GPU version. The C library function clock(void) was used in those programs to measure the time it took to solve the generated MDP problem. Action space size, state space size, and the number of iterations were the variables used to generate different experimental configurations. Then, each experimental configuration was tested using the three versions of the MDP solver, and time measurements were recorded.

For the testing of the CPU sequential version, crunchy1.cims.nyu.edu Compute Server which had Four AMD Opteron 6272 (2.1 GHz) (64 cores) and 256 GB of memory was used. For the testing of both GPU versions, cuda2.cims.nyu.edu Compute Server which had two Intel Xeon E5-2660 (2.60 GHz) (40 cores), two GeForce RTX 2080 Ti (11 GB memory each), and 256 GB of memory was used.

Since we are only interested in the performance of the algorithm with respect to time, we can fill the transition and reward function arrays arbitrarily. We fill every element with a unique number so we are simulating solving a dense problem setup. Since we are also not interested in the number of iterations that each problem takes to convergence, since this depends upon the problem formulation, we investigate the time to perform a fixed number of iterations and compare across the different algorithms. The convergence of the algorithm with respect to the number of iterations is constant for the different algorithms since they are all performing the same fundamental operation. We perform an accuracy check to compare the different methods and to ensure they all are producing the same array values. The GPU speedup for both the single and multi GPU approach is calculated relative to the CPU time.

6. Experiments and Analysis

All three experiments showed a similar story. Although both GPU programs had positive speedup compared to the CPU program, the multi-GPU version performed better than the single-GPU version with all problem configurations. The CPU time scales linearly with the problem complexity $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|N)$ as we would have expected but this is not the case for the multi and single GPU algorithms.

As the value of the independent variable increased, the performance difference between the two versions only widened. When the array size reached 1.07E+9 bytes and the number of iterations reached 100, the speedup stabilized at around 2.5 for the single-GPU version and 110 for the multi-GPU version. Based on this finding, multi-GPU version will always give better performance while solving a reasonably large MDP problem. The only time single-GPU version should be used is when the action space size is less than or equal to 128, the state space size is less than or equal to 64, and the number of iteration is less than 40.

This huge difference in performance between the single-GPU and the multi-GPU versions can be explained by the cost of memory copy. For every iteration, the CPU had to move an enormous amount of array data to the GPU memory in the single-GPU version, while in the multi-GPU version it didn't have to due to the use of unified memory. As the number of iterations increased, the total cost of memory copy increased with it, which slowed the single-GPU version down. On the other hand, since the multi-GPU version didn't have to suffer this performance issue, its performance advantage over the single-GPU version just became progressively more significant as the number of iteration increased.

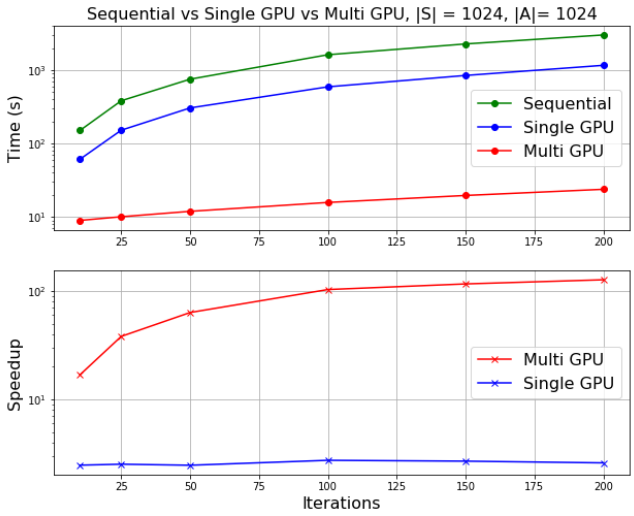

Figure 2. Graph showing how the time and speedup change with respect to the increasing number of iteratins.

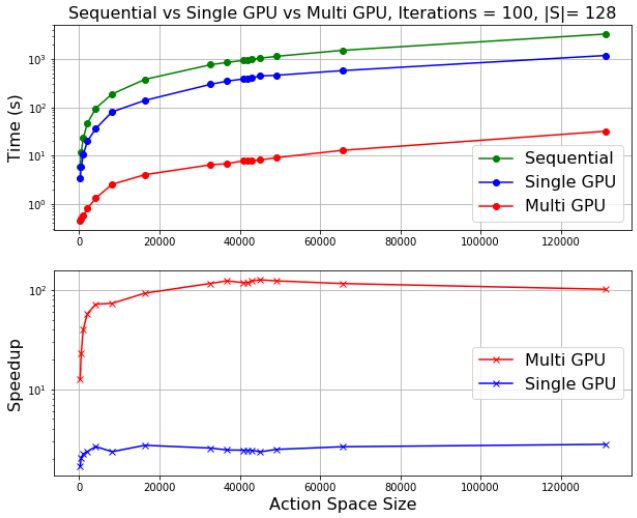

Figure 3. Graph showing how the time and speedup change with respect to the increasing action space size.

7. Conclusions

Clearly our results provide some positive indication that the multi-GPU approach performs favorably to the single-GPU approach. One of the strengths of our multi-GPU algorithm is that it does not rely upon any particular structure of the MDP we are trying to solve. Many of the other state of the art solvers rely upon properties such as sparsity or if the problem can be factorized into smaller parts in order to do efficient solving. We do not need to know anything about the problem beforehand apart from the size of the state and action space in order to show that the multi-GPU approach offers an advantage.

Since the computing power has only scaled by a factor of two for our multi GPU, the primary speedup must be due to the efficiency of the unified memory system over manually moving the data from the CPU to GPU. In order to do further experimentation we would have liked to have been able to try running on more than two GPUs however this was not possible on the Courant computing servers. We would expect the performance increases to scale linearly with the number of GPUs.

In the future we would like to use the GEMBench (Sapio et al., 2019) to compare our method with other GPU accelerated MDP solvers and other CPU solver algorithms, this would provide us with some understanding of whether the algorithm can outperform the state of the art.

References

Bellman, Richard. A markovian decision process. *Indiana University Mathematics Journal*, 6:679–684, 1957a.

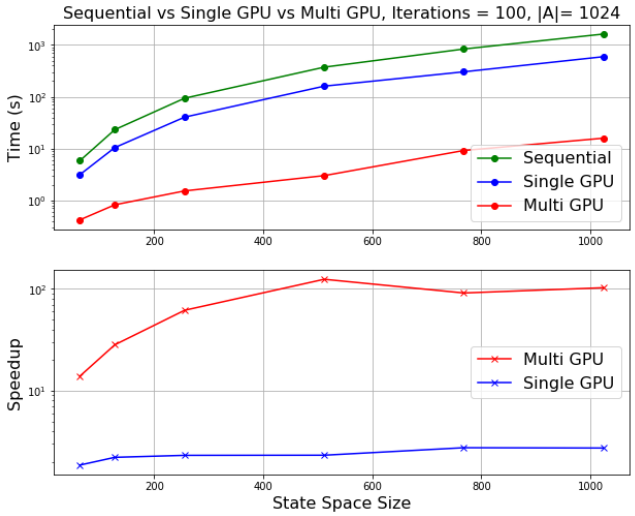

Figure 4. Graph showing how the time and speedup change with respect to the increasing state space size.

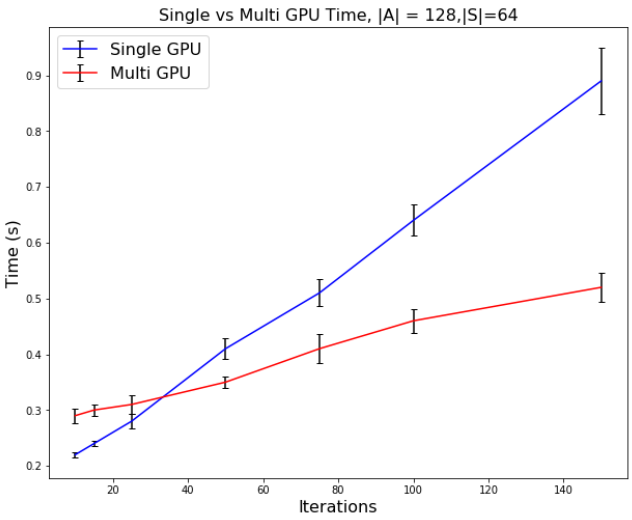

Figure 5. Graph showing that only for a small state and action space size and a small number of iterations does the single GPU algorithm outperform the multi-GPU algorithm.

Bellman, Richard. Dynamic programming. *Science*, 153: 34 – 37, 1957b.

Benini, Luca, Bogliolo, Alessandro, Paleologo, Giuseppe A., and Micheli, Giovanni De. Policy optimization for dynamic power management. *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, pp. 182–187, 1998.

Buşoniu, Lucian, Babuka, Robert, and Schutter, Bart De. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38:156–172, 2008.

Dean, Thomas L. and Lin, Shieu-Hong. Decomposition techniques for planning in stochastic domains. In *International Joint Conference on Artificial Intelligence*, 1995.

Dean, Thomas L., Givan, Robert, and Kim, Kee-Eung. Solving stochastic planning problems with large state and action spaces. In *AIPS*, 1998.

Hoey, Jesse, St-Aubin, Robert, Hu, Alan J., and Boutilier, Craig. Spudd: Stochastic planning using decision diagrams. *ArXiv*, abs/1301.6704, 1999.

Hong, Sungpack, Kim, Sang Kyun, Oguntebi, Tayo, and Olukotun, Kunle. Accelerating cuda graph algorithms at maximum warp. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2011.

Jain, Anuj K. and Sahni, Sartaj. Value iteration on multicore processors. *2020 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pp. 1–7, 2020.

Kim, Kee-Eung and Dean, Thomas L. Solving factored mdps with large action space using algebraic decision diagrams. In *PRICAI*, 2002.

Kurniawati, Hanna, Hsu, David, and Lee, Wee Sun. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems*, 2008.

Littman, Michael L., Dean, Thomas L., and Kaelbling, Leslie Pack. On the complexity of solving markov decision problems. *ArXiv*, abs/1302.4971, 1995.

Meuleau, Nicolas, Hauskrecht, Milos, Kim, Kee-Eung, Peshkin, Leonid, Kaelbling, Leslie Pack, Dean, Thomas L., and Boutilier, Craig. Solving very large weakly coupled markov decision processes. In *AAAI/IAAI*, 1998.

Nickolls, John R. and Dally, William J. The gpu computing era. *IEEE Micro*, 30, 2010.

Noer, D. Parallelization of the value-iteration algorithm for partially observable markov decision processes, 2013. URL `http://www.compute.dtu.dk/English.aspx`. DTU supervisor: Hans Henrik Løvengreen, hhlo@dtu.dk, DTU Compute.

Poupart, Pascal. Exploiting structure to efficiently solve large scale partially observable markov decision processes. 2005.

Puterman, Martin L. Markov decision processes: Discrete stochastic dynamic programming. In *Wiley Series in Probability and Statistics*, 1994.

Ruiz, Sergio and Hernández, Benjamín. A parallel solver for markov decision process in crowd simulations. *2015 Fourteenth Mexican International Conference on Artificial Intelligence (MICAI)*, pp. 107–116, 2015.

Sapio, Adrian E., Bhattacharyya, Shuvra S., and Wolf, Marilyn Claire. Efficient solving of markov decision processes on gpus using parallelized sparse matrices. *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 13–18, 2018.

Sapio, Adrian E., Tatiefo, Rocky L., Bhattacharyya, Shuvra S., and Wolf, Marilyn Claire. Gembench: A platform for collaborative development of gpu accelerated embedded markov decision systems. In *SAMOS*, 2019.

Singh, Dilpreet and Reddy, Chandan K. A survey on platforms for big data analytics. *Journal of Big Data*, 2, 2014.

Smith, Trey and Simmons, Reid G. Point-based pomdp algorithms: Improved analysis and implementation. In *Conference on Uncertainty in Artificial Intelligence*, 2005.

Szepesvari, Csaba. Algorithms for reinforcement learning. In *Algorithms for Reinforcement Learning*, 2010.

Wang, Yu Emma, Wei, Gu-Yeon, and Brooks, David M. Benchmarking tpu, gpu, and cpu platforms for deep learning. *ArXiv*, abs/1907.10701, 2019.