EFFICIENT AND HIGHLY SCALABLE TECHNIQUES TO ACCELERATE BAYESIAN MARKOV DECISION PROCESS COMPUTATION

A Dissertation

by

HE ZHOU

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jiang Hu
Co-Chair of Committee,	Sunil P Khatri
Committee Members,	Laszlo B. Kish
	Duncan Henry M. Walker
Head of Department,	Miroslav M. Begovic

December 2019

Major Subject: Computer Engineering

Copyright 2019 He Zhou

ABSTRACT

Markov Decision Process (MDP) is a kernel model for solving sequential decision making problems, when the system behaviour is stochastic. Bayesian Markov Decision Process (BMDP) can be applied in the special case in which the system model is uncertain. Both MDP and BMDP must repeatedly conduct exhaustive search for a non-stationary policy, and thus entail exponential computational complexity. This has hindered their practical application to date.

In this thesis, we develop several computation techniques to overcome the obstacle of the exponential runtime complexity as well as the exponential memory requirement for both the MDP and BMDP problems. To reduce the runtime complexity, we investigate acceleration techniques, using the Graphic Processing Unit (GPU) platform, which allows massive parallelism. Our GPU-based acceleration techniques are applied with two different MDP approaches: the Optimal Bayesian Robust (OBR) policy and the Forward Search Sparse Sampling (FSSS) method. However, since the GPU utilizes a Single Instruction Multiple Data (SMID) computation paradigm and (B)MDP has an inherent issue of “curse of dimensionality”, the GPU-based solution has an exponential memory complexity issue. To overcome the memory storage impediment, we first exploit the fact that in many practical problems the system model is likely to be sparse. Exploiting this, we develop a novel Duplex Sparse Storage (DSS) scheme in this thesis. Another approach we develop to reduce the memory is a highly memory-efficient representation for the (B)MDP system model using Binary Decision Diagram (BDD) based sampling. For both DSS and BDD-based sampling approaches, we develop corresponding (B)MDP solvers on a heterogeneous CPU-GPU platform.

To further improve the efficiency of BMDP, we develop a new Scaled Population (SP) based arithmetic computation approach that achieves considerable improvements over existing Stochastic Computing (SC) techniques. Note that the SP arithmetic approach can be used in applications other than BMDP as well. SP arithmetic introduces scaling operations that significantly reduce the numerical errors as compared to SC. Besides, the SP arithmetic erases the inherent serialization associated with stochastic computing, thereby significantly improving the computational delays.

Our experiments show that the GPU-based parallel computation techniques reduce the runtime of (BMDP) by two orders of magnitude over sequential implementations. The DSS and BDD-based sampling approaches reduce the memory utilization by $4.1\times$ and two orders of magnitude compared with the use of floating point numbers, respectively. The SP arithmetic achieves a 31.89% improvement over SC in terms of the accuracy for BMDP, and also improves the accuracy and the utilization of hardware resources for other applications such as single multiplication/addition, matrix inner production and MNIST image classification.

DEDICATION

To my dearest father and mother, who have been supporting my decisions and choices for all these years. Special thank to Professors Jiang Hu, Sunil P Khatri and Dr. Frank Liu, for supporting me during my PhD career. Thank you to all the committee members, who have offered valuable advice for this thesis.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Jiang Hu (co-chair) and Professor Sunil P Khatri (co-chair), Frank Liu of IBM Research, Professor Laszlo B. Kish of the Department of Electrical & Computer Engineering, and Professor Duncan Henry M. Walker of the Department of Computer Science and Engineering.

Part of the OBR algorithm implementation in Chapter 4 uses the code from Mohammadmahdi Rezaei Yousefi of the Ohio State University.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by a fellowship from Texas A&M University.

NOMENCLATURE

MDP	Markov Decision Process
BMDP	Bayesian Markov Decision Process
t	Time step in the MDP/BMDP system
n	Number of states
m	Number of actions
z_t	The state of the MDP/BMDP system at time t
d_t	Action applied on the MDP/BMDP system at t
$Z = \{z_1, d_1, z_2, d_2 ..., d_{t-1}, z_t\}$	State transition sequence
$S = \{s_1, s_2, ..., s_n\}$	The state set of the MDP/BMDP system
$A = \{a_1, a_2, ..., a_m\}$	The action set of the MDP/BMDP system
$\mathcal{P}$	State transition probability matrix (TPM)
$\overline{\mathcal{P}}$	Expected state transition probability matrix
$\overline{\mathcal{P}}'$	Posterior expected state transition probability matrix
$r_{a_k}(s_i, s_j)$	Reward function
$g_{ij}(a)$	Immediate reward
λ	Discount factor
$\pi : S \to A$	Policy of the MDP/BMDP system
V_{s_i}	Value function for state s_i
J	Long-term accumulated reward
T	Time horizon of the system
DSS	Duplex sparse storage
SP	Scaled population

GRN	Gene regulatory network
OBR	Optimal bayesian robust
FSSS	Forward search sparse sampling

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS viii

LIST OF FIGURES xi

LIST OF TABLES xiii

1. INTRODUCTION AND MOTIVATION 1

   1.1 Parallel Computing for MDP/BMDP 2
   1.2 Memory Use Reduction Techniques for MDP/BMDP 3
   1.3 MDP/BMDP Computation Techniques on New Arithmetic 4
   1.4 Goal and Contribution 5
   1.5 Organization of the Dissertation 6

2. BACKGROUND 7

   2.1 The Classical MDP Problem 7
   2.2 The Bayesian MDP Problem 8

3. PREVIOUS WORK 11

   3.1 Reinforcement Learning and MDP 11
   3.2 Previous Work on Computational Efficiency 14
      3.2.1 Previous Work on Improving the Scalability 14

4. PART 1: PARALLEL BMDP ON A GPU PLATFORM 17

   4.1 Background and Introduction 17
   4.2 Bayesian Control Of GRNs 19
      4.2.1 Problem Formulation 19
      4.2.2 OBR Algorithm 20
      4.2.3 FSSS Algorithm 20
   4.3 GPU-based Acceleration 21

4.3.1 Parallelization of OBR 21
4.3.2 GPU-Based Parallel Computing for FSSS 22
4.3.3 GPU Memory Utilization and Thread Organization 24
4.4 Result 25
4.5 Conclusion 27

5. PART 2: A DUPLEX SPARSE STORAGE (DSS) SCHEME FOR BMDP ON GPU PLATFORM 29

5.1 Introduction and Background 29
5.1.1 Preliminaries - CSR Storage Format 31
5.2 Our Approach - Duplex Sparse Storage (DSS) 33
5.2.1 Problems of Using Conventional Full Matrix 33
5.2.2 DSS Format in BMDP 34
5.2.3 Parallel DSS-based OBR 37
5.2.3.1 Look Up $\overline{\mathcal{P}}_{ij}$ 37
5.2.3.2 Bayesian Update 39
5.2.4 Comparison Between DSS and CSR 42
5.3 Experimental Results 43
5.4 Conclusion 48

6. PART 2: A NOVEL BINARY DECISION DIAGRAM (BDD) BASED SAMPLING FOR BMDP 52

6.1 Introduction and Background 52
6.1.1 Preliminaries - Binary Decision Diagrams 54
6.2 BDD-based Sampling Representation and Operations 56
6.2.1 Overview 56
6.2.2 BDD-based Sampling Representation of TPM 56
6.2.2.1 Generating BDD_M 58
6.2.2.2 Looking up $\overline{\mathcal{P}}_{ij}$ from BDD_M 59
6.2.2.3 Bayesian Update on BDD_M 60
6.2.3 G-BDD: A BMDP Solver Using BDD-based Sampling Representation on CPU-GPU 61
6.3 Experiment Results 63
6.4 Conclusion 68

7. PART 2: ANALYSIS OF THE UPPER ERROR BOUND OF DSS AND BDD-BASED APPROACHES 70

7.1 Upper Error Bound of DSS 70
7.2 Upper Error Bound of BDD-based Sampling Approach 74

8. PART 3: SCALED POPULATION ARITHMETIC FOR EFFICIENT BMDP COMPUTATION 78

8.1 Background and Introduction 78

8.2 Stochastic Computing and Previous Works ... 80
8.3 Scaled Population Arithmetic ... 81
8.3.1 Number Representation ... 81
8.3.2 Arithmetic and Supporting Operations ... 82
8.3.2.1 Multiplication ... 83
8.3.2.2 Addition ... 84
8.3.2.3 Generator ... 86
8.3.2.4 Shuffle Unit ... 87
8.3.2.5 Density Checker Unit ... 88
8.3.2.6 Scaling Unit ... 88
8.4 Experiment Results ... 89
8.4.1 Single Arithmetic Operation ... 89
8.4.2 Application 1: BMDP Problem ... 92
8.4.3 Application 2: Matrix Inner Product ... 93
8.4.4 Application 3: MNIST Digit Classification ... 94
8.5 Conclusion ... 95

9. CONCLUSION AND FUTURE WORK ... 97

REFERENCES ... 99

LIST OF FIGURES

FIGURE Page

3.1 An Example of Factored Model ... 15

4.1 Computational Structure of OBR ... 21

4.2 Runtime of different input size, $P_{th} = 0.01^L$ ... 26

4.3 Run time of GPU-FSSS with different K, $N = 5$ ... 27

4.4 Memory and error performance of GPU-FSSS and GPU-OBR ... 28

5.1 CSR Example ... 31

5.2 Histogram of P_{ij} after 5 Bayesian Updates ... 34

5.3 DSS Example ... 36

5.4 Example of Looking Up $\overline{\mathcal{P}}_{44}$ on GPU ... 38

5.5 Bayesian Update Example ... 39

5.6 Performance of G-DSS and G-CSR ... 46

5.6 Performance of G-DSS and G-Full when $K = 3$ ... 50

5.7 Performance of G-DSS when $R = 20\%$ and number of states $= 32$ ... 50

5.8 Trade-off between K and number of states with a 2GB memory limitation when $R = 20\%$ ... 51

5.9 Look-up and computation time of G-DSS ... 51

6.1 An example of using BDD to represent 2 Boolean functions f and g. The left (right) child of each vertex is its 0-cofactor (1-cofactor) with respect to vertex variable. ... 55

6.2 The BMDP Framework on Heterogeneous Computing Platform ... 62

6.3 An Example of the GPU Computation ... 63

6.4 Memory Utilization in CMDP-BDD ... 64

6.5 Raw Memory of $\overline{\mathcal{P}}$ Matrices in BMDP 64

6.6 Performance of Memory Utilization and Runtime 66

6.7 Result Accuracy 67

7.1 DSS Upper Error Bound: $\overline{\mathcal{P}}_{i*}(a)$ and W 71

7.2 DSS Upper Error Bound Curve: Linear 72

7.3 DSS Upper Error Bound Curve: $\overline{\mathcal{P}}_{i*}(a)$ and W 73

7.4 DSS Upper Error Bound Curve: Step 74

7.5 BDD Upper Error Bound: $\overline{\mathcal{P}}_{i*}(a)$ and W 75

7.6 BDD Upper Error Bound Curve: Linear 76

7.7 BDD Upper Error Bound: $\overline{\mathcal{P}}_{i*}(a)$ and W 77

7.8 BDD Upper Error Bound Curve: Step 77

8.1 Multiplication: $\frac{4}{8} \times \frac{4}{8} = \frac{2}{8}$ 80

8.2 Addition: $\frac{2}{8} + \frac{3}{8} = \frac{5}{8}$ 80

8.3 The top level view of SP scheme. 82

8.4 The SP-based Multiplication 84

8.5 SP-based *Skewed* Addition 85

8.6 An Example of Generate Operation, with $X = 0.25$, and $\Pi = 4$ 86

8.7 Shuffle Example ($W = w = 4$) 87

8.8 Relative errors proportional to darkness. 90

LIST OF TABLES

TABLE Page

5.1 Differences Between CSR and DSS 42

5.2 Summary of Notations Used From Fig. 5.6 to Fig. 5.9 45

8.1 Error Contribution of SP-based Multiplication (%) 89

8.2 Error Contribution of SP-based Addition (%) 91

8.3 Efficiency Comparison for Multiplication ($\Pi = 32$) 91

8.4 Efficiency Comparison for Addition ($\Pi = 32$) 92

8.5 Error of BMDP Problem (%) ($\Pi = 64$, $M = 64$) 93

8.6 Error of Matrix Inner Production (%) ($\Pi = 32$) 94

8.7 Error of Matrix Inner Production (%) ($\Pi = 64$) 94

8.8 Error of Matrix Inner Production (%) ($\Pi = 128$) 94

8.9 MNIST Classification Success Rate (%) 95

1. INTRODUCTION AND MOTIVATION

The Markov Decision Process (MDP) has been applied to many fundamental optimization problems, such as gene regulatory network control [1], robotics [2, 3], inventory management [4], finance [5, 6], etc. The theory of MDP is a natural framework for constructing optimal intervention policies for a system in which the state transition behaviour of the system is stochastic. Due to the dynamic characteristics of the system and the intervention of the decision maker, the state of the system changes through a sequence of t state transitions. Systems that can be described as an MDP usually obey the Markov property, which indicates that the probability distribution of the future states of the system only depends on the present state and not on the state transition history. The MDP is widely used to model the sequential decision making process in a dynamic environment.

Although the conventional MDP is able to model the decision making process in a dynamic setting, it requires perfect knowledge of the system, which is rarely available due to the inherent complexity and variability of the processes that govern the underlying system behaviour. The system model is usually represented by a state transition probability matrix (TPM). In order to handle the uncertainty of the system, a class of TPMs with a prior probability over this class is assumed, in which the prior probability indicates the confidence level of how likely each TPM might represent the underlying true TPM. As more observations are made, the prior probability will be updated to the posterior probability by applying Bayes' rule [7]. The MDP with such an uncertainty framework is called a Bayesian MDP (BMDP), which implements model-based reinforcement learning.

In the BMDP problem, an expected TPM derived from the probability over the class of TPMs is used to represent the system model. To search for the optimal non-stationary policy, dynamic programming is performed on the expected TPM across the uncertainty class. Throughout the process of policy search, Bayes' rule is applied to update the prior probability distribution of the TPM class and generate the posterior distribution. Since the BMDP framework is able to deal with the uncertainty of a system, it has a wide range of applications such as the Gene Regulatory

Network control [1], robot navigation [8], dynamic pricing [9], etc.

Although the BMDP model is effective to overcome the difficulty arising from partial knowledge of the system model, its computational complexity is exponential. The BMDP framework also suffers from the "curse of dimensionality" [10] inherited from the MDP framework and dynamic programming. As a result, when the dimension of the BMDP problem increases, the CPU runtime and memory required to compute a solution increases exponentially as a function of the size of state space, the size of action space and the depth of horizon. This is because the prior distribution of the TPMs is updated when there is a new observation, hence the corresponding expected TPM also changes. Therefore, we need to track the expected TPM throughout the entire dynamic programming process. Also, in many real-world applications, the system environment may change over time and therefore transition probabilities are dynamic. When the problem size is large, the TPMs of the BMDP system would demand a large memory space, which is not always available. For example, one application of BMDP is to model the autonomous systems in the real world, where the computational platform is usually a microcontroller with limited memory. Many autonomous navigation systems contain millions of states [11] and an autonomous planning system can have more than 22 million states [12]. It is very difficult, if not impossible, for typical microcontrollers to handle such large BMDP problems.

1.1 Parallel Computing for MDP/BMDP

Hardware acceleration techniques have been studied to improve the efficiency of BMDP computing. One of our approaches is to use parallel computing based on Graphic Processing Units (GPUs). The GPU is a high performance computing platform which employs massive multi-threading, fast hardware-based context switching as well as high memory bandwidth. A GPU has thousands of small processor cores, and when properly utilized, it can achieve impressive performance gains for several applications compared to traditional multi-core CPUs. Compared to parallel computing on a CPU, the speedup from GPU is often much greater, since a GPU has thousands of simple processor cores while a CPU typically contains only a few cores. The GPU-based approach is often much more cost-effective than parallel computing on a cloud or server farm.

1.2 Memory Use Reduction Techniques for MDP/BMDP

Although a GPU-based implementation can greatly accelerate the BMDP computation, it faces a key bottleneck on its scalability. That is, a single GPU has limited memory storage and operates in a single instruction multiple data (SIMD) manner. Also in a modern GPU platform, the communication cost of transferring data between the GPU and the main memory is non-negligible. Hence memory utilization often results in a performance bottleneck for GPU-based parallel computing. Alleviating the memory bottleneck for large BMDP problems is a critical problem, and is one of the key goals in this thesis.

In order to solve the memory problem, we develop three approaches. First, we explore the applicability of applying Forward Search Sparse Sampling (FSSS) [13] on the GPU platform. FSSS is a heuristic for solving the MDP problem. Instead of using a value to evaluate the state-action pair as the dynamic programming algorithms in MDP/BMDP does, FSSS calculates the upper and lower bound values for each state-action pair. Also, FSSS samples only critical transitions, instead of examining every state-action pair in the dynamic programming process, which becomes the key reason for the speedup of FSSS. By implementing FSSS on a GPU, we are able to alleviate the memory problem, while obtaining significant speedup.

For the second approach, we exploit the observation that the expected TPMs tend to be sparse in many real life problems [14, 3]. This sparsity of the TPM motivates us to investigate sparse data storage techniques to address the memory issue. Since both the prior and posterior distributions of the TPM in BMDP are known, we can exploit the sparsity of the TPM. The Compressed Sparse Row (CSR) format is an efficient approach to store a sparse matrix [15]. Compared to full matrix storage, in which every single entry is stored in memory regardless of its value, CSR only stores the non-zero entries of the matrix. The CSR format has some overhead to store the row/column locations of the non-zero entries. We propose the Duplex Sparse Storage (DSS) scheme which improves over CSR in both overhead and accuracy.

We also propose a compact model which represents the transition probabilities by sampling a set of Boolean functions instead of using floating point numbers to represent the probability. If

there are n states, a probability matrix in this model uses a constant number of Boolean functions, each of which has $\mathcal{O}(\log n)$ inputs, as opposed to $\mathcal{O}(n^2)$ floating numbers in the conventional representation. In our approach, the Boolean functions are stored as Binary Decision Diagrams (BDDs), which facilitate sharing with in a Boolean function as well as among multiple Boolean functions. We also developed a technique for performing Bayesian update on the BDD sampling-based TPM representation.

1.3 MDP/BMDP Computation Techniques on New Arithmetic

Another approach proposed in this thesis to improve the efficiency of BMDP is based on approximation computing. Approximate computing is a non-conventional approach with an emphasis on area and power efficiency, while sacrificing accuracy. For certain classes of applications which has a tolerance to computational errors, approximate computing can achieve better area and power characteristics compared with exact arithmetic. One popular technique for approximate computing is stochastic computing (SC) [16]. SC is an arithmetic scheme for area-efficient implementation of error-tolerant applications. Stochastic computing has received renewed interest due to, among other reasons, the degrading reliability of recent VLSI fabrication processes, its purported decrease in power, and its robustness to bit-flip errors. In stochastic computing, values are represented by binary bit streams, and the arithmetic operations can be processed by simple logic circuits, such as OR/AND gates for addition and multiplication, respectively. However, classical SC has its own limitations. First, its accuracy depends heavily on the density and the randomness of the 1's in the binary bit-stream [17]. Second, since SC uses a population-based representation alone for all numbers, it can only represent numbers in $[0, 1]$. The limitation can be problematic when overflow occurs in the operations, especially in addition. The third limitation of SC is the runtime complexity. Although the arithmetic operation units consist of only OR/AND logic gates, the supporting units, e.g., the random number generator (RNG) and the shuffler, have a runtime complexity of $\mathcal{O}(k)$, where k is the number of bits in the SC representation. These weaknesses limit the applicability of SC.

In order to alleviate the above limitations of SC, we propose a new Scaled Population (SP)

arithmetic based computing which achieves fast, approximate computing with a low area/power overhead and improved accuracy. SP arithmetic uses some of the basic ideas of SC, but with three key enhancements: a) the inherent serialization in SC is avoided; b) the errors of SC are significantly reduced by providing a scaling (exponent) term in SP arithmetic; and c) the range of numbers that can be represented by SP is much larger than what is possible in SC. The key design goal of SP arithmetic is that each operation be computed using $\mathcal{O}(1)$ gate delays (as opposed to clock cycles). Unlike SC, SP never allows any operation which requires a serial traversal of the bits of the operand. The SP arithmetic achieves a dramatic speedup over SC on single addition and multiplication operations, and it achieves better accuracy as well.

1.4 Goal and Contribution

In our research, we aim to develop both hardware acceleration techniques and model oriented BMDP computing techniques such that the computation complexity and memory storage challenge can be addressed. The novelty of this research lies in its emphasis on the interplay between hardware platforms, and BMDP algorithms and scalable BMDP system model representation, which have been largely neglected in the past. A variety of hardware-based approaches for efficiency improvement and memory reduction are investigated in this research. Note that our approaches can also be applied to many other applications other than MDP/BMDP, which will be also discussed in the dissertation.

1.5 Organization of the Dissertation

The rest of the dissertation is organized as follows:

In Chapter 2, we discuss the background of the classical and Bayesian MDP problems

In Chapter 3, we present the previous work of solving the MDP problems as well as improving the efficiency.

This thesis has 3 parts, a) approaches to parallelize the MDP/BMDP, b) develop techniques to reduce memory ultilization for MDP/BMDP, and c) approximate arithmetic to improve the efficiency of the MDP/BMDP computation.

In Chapter 4, we present a parallel paradigm on a GPU platform to reduce the overcome the runtime issue of the MDP/BMDP problem. We also discuss the parallelization on FSSS to further improve the efficiency.

In Chapter 5, we present a Duplex Sparse Storage (DSS) representation to reduce the memory utilization of the MDP/BMDP problem.

In Chapter 6, we present a BDD-based sampling representation to reduce the memory utilization of the MDP/BMDP problem.

In Chapter 7, we analyze the upper error bound of the DSS and BDD-based representation approaches.

In Chapter 8, we propose a scaled population arithmetic for efficient MDP/BMDP computation.

Finally, in Chapter 9, we conclude our work and discuss the possible future work.

2. BACKGROUND

2.1 The Classical MDP Problem

The Markov Decision Process (MDP) is a framework for making a sequence of decisions over a stochastic dynamic system. Due to the dynamic characteristics of the system and the intervention of the decision maker, the state of the system changes through a sequence of t state transitions. Formally, a sequence is denoted as $Z = \{z_1, d_1, z_2, d_2..., d_{t-1}, z_t\}$, where z_t is the state of the system at time t and d_t is the action applied to the system at time t.

An MDP system [18] is described by the 5-tuple (S, A, P, r, λ), where $S = \{s_1, s_2, ..., s_n\}$ describes the set of states. $A = \{a_1, a_2, ..., a_m\}$ is a set of actions which can be applied to the system. The system model in the MDP is usually represented by a state transition probability matrix (TPM) P. Each entry of P is denoted as $P_{ij}(a_k)$. $P_{ij}(a_k) = Prob.(z_{t+1} = s_j \mid z_t = s_i, d_t = a_k)$ describes the state transition probability of the system at time t, when the state of the system at time t is s_i, the state of the system at time $t+1$ is s_j, and the action applied to the system at time t is a_k. The reward function $r_{a_k}(s_i, s_j)$ describes the reward that the system receives by transitioning from state s_i to s_j under action a_k. The discount factor $\lambda \in (0, 1]$ expresses the rate of decrease of the importance between present rewards and future rewards. A rule representing which action the agent selects at each possible state is defined as a mapping from past states observed to the set of actions. Since the system is dynamic, after applying the action a_k, the next state s_j obeys a certain probability distribution rather than being deterministically decided. A process is Markovian if this distribution depends only on the last state of the observation sequence and the action. A policy is called stationary if it does not change over time.

The fundamental MDP problem is to find a policy $\pi : S \to A$ to inform the decision maker which action to take when the system is in a certain state, so that the cumulative reward of the

system over horizon T, as defined in Eq. (2.1) is maximized.

$$\sum_{t=1}^{T-1}(\lambda^t r_{d_t}(z_t, z_{t+1})) \tag{2.1}$$

Each state s_i is associated with a value function $V_(s_i)$, which defines the long term return and defined as:

$$V_{s_i} = \max_{a_k \in A}\{\sum_{s_j \in S} P_{ij}(a_k)(r_{a_k}(s_i, s_j) + \lambda V_{s_j})\} \tag{2.2}$$

Many approaches for solving an MDP problem are based on two dynamic programming algorithms: value iteration [19] and policy iteration [20]. Value iteration begins with a value function $V_{s_i}^0$ for each state s_i. At each iteration t, a new value function is generated by applying Eq. (2.2) to the current value function, until V_{s_i} converges. Policy iteration starts with an initial policy. At each iteration, it evaluates the value function of the current policy, this process is referred to as policy evaluation, and then a policy improvement step is performed, in which a new policy is generated as a greedy policy with respect to the value of the current policy. Policy iteration is known to produce a strictly monotonically improving sequence of policies.

2.2 The Bayesian MDP Problem

The classical MDP framework requires perfect knowledge of the system, particularly the TPM P and the immediate reward r. In reality, the knowledge, especially of P, is usually not completely available due to the complicated underlying dynamic characteristics of the system. Since only partial prior knowledge of the TPM is known, it is assumed that there is a family of TPMs following a certain distribution $\Omega(P)$, which describes the degree of confidence we place on each TPM [21]. When a new observation is made in which the system transits from state s_i to s_j under a certain action a, the posterior probability distribution of the TPMs can be computed by following Bayes' rule.

In the special case where both the prior and posterior distributions belong to the same distribution family, it can be assumed that the TPM family obeys the Dirichlet distribution. In order to account for the uncertainty of the TPM in the classical MDP formulation, the state s_i of the MDP is extended to a hyperstate $(s_i, \overline{\mathcal{P}})$, where $\overline{\mathcal{P}}$ is the expected TPM over $\Omega(P)$.

The policy of an MDP model with hyperstate $(s_i, \overline{\mathcal{P}})$ defines the action executed at a certain state s_i, so that the long-term accumulated reward $J^*(s_i, \overline{\mathcal{P}})$ of the system starting from an initial state with a prior expected TPM $\overline{\mathcal{P}}$ is maximized. $J^*(s_i, \overline{\mathcal{P}})$ is defined in Eq. (2.3), where $g_{ij}(a)$ is the immediate reward obtained when the system transitions from state s_i to s_j under action a, $\overline{\mathcal{P}}$ is the prior expected TPM, and $\overline{\mathcal{P}}'$ is the posterior expected TPM.

$$J^*(s_i, \overline{\mathcal{P}}) = \max_{a \in A}\{\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda J^*(s_j, \overline{\mathcal{P}}')]\} \tag{2.3}$$

In Eq. (2.3), $\overline{\mathcal{P}}_{ij}(a)$ is the expected state transition probability from s_i to s_j under action a. If the TPM family follows the Dirichlet distribution with a parameter matrix α, $\overline{\mathcal{P}}_{ij}(a)$ is mathematically defined as follows:

$$\overline{\mathcal{P}}_{ij}(a) = \frac{\alpha_{ij}}{\sum_{\forall m \in S} \alpha_{im}} \tag{2.4}$$

When a new state transition from s_i to s_j is observed, the prior α is updated to the posterior α' as follows [22, 1]:

$$\alpha' = \alpha + \gamma \tag{2.5}$$

where γ is a matrix of zeros, except $\gamma_{ij} = 1$. Then the corresponding $\overline{\mathcal{P}}'$ matrix can be obtained via Eq. (2.4) with the posterior α'.

A key issue of a sequential decision making problem is exploration, which determines how the agent should choose actions while learning about the task. Another key issue is exploitation, in which actions are selected so as to maximize expected reward with respect to the current value

function estimate. In the BMDP, exploration and exploitation are naturally balanced in a coherent mathematical framework[23]. Policies are expressed over the full information state (or belief), including model uncertainty. In this framework, the optimal Bayesian policy will be to select actions not only based on how much reward they give, but also based on how much information they provide about the system model. In order to solve Eq. (2.3), an Optimal Bayesian Robust (OBR) approach is proposed in by applying successive approximation [22]. In the OBR scheme, $J^*(s_i, \alpha)$ is approximated recursively by Eq. (2.6), using K iterations with a set of predefined $J_0(s, \overline{\mathcal{P}})$.

$$J_{k+1}(s_i, \overline{\mathcal{P}}) = \max_{a \in A}\{\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda J_k(s_j, \overline{\mathcal{P}}')]\} \tag{2.6}$$

In [22], it is proved that $J_{k+1}(s_i, \overline{\mathcal{P}})$ will converge monotonically to the optimal solution $J^*(s_i, \overline{\mathcal{P}})$ for a valid $\overline{\mathcal{P}}$ under certain moderate conditions.

3. PREVIOUS WORK

As discussed in Chapter 1, although the BMDP is a framework formulated to describe the uncertainty of the system, and BMDP algorithms have been extensively studied, it still faces runtime and memory challenges. There are many research efforts attempting to address these challenges. In this chapter, we will give a general overview of these previous work. More detailed discussion of the previous work corresponding to our approaches will be provided in later chapters.

3.1 Reinforcement Learning and MDP

Reinforcement learning is a class of learning problems in which an agent (or controller) interacts with a dynamic, stochastic, and uncertain environment, with the goal of finding an optimal action-selection policy, to maximize its rewards in long term. MDP is one of the key model for reinforcement learning. If the environment state is not always completely observable, then partially observable Markov decision process (POMDP) [24] is usually applied.

As discussed in Chapter 2, in some cases, MDPs can be solved analytically, and in many cases it can be solved iteratively by dynamic or linear programming [25]. However, in many real life applications, these methods are not applicable, since either the state space is too large, or we don't have a prefect knowledge for the system model. In these cases, other techniques and algorithms may be helpful. Sampling-based methods are widely used for solving MDPs. In many cases, an explicit transition-probability model (TPM) is not available. In such cases, the goal would be to find a reasonably good approximation for the policy, i.e., a policy leading to large enough value function.

Generally speaking, reinforcement learning solutions for MDP problems can be categorized in 2 different ways [23].

- Model-based vs. Model-free Methods:

  In model-based methods, reinforcement learning algorithms explicitly learn a system model and use it to solve an MDP problem. In model-free methods, reinforcement learning al-

gorithms do not explicitly learn a system model and only use sample trajectories obtained by direct interactions with the system. These methods include popular algorithms such as Q-learning [26], SARSA [27], etc.

- Value Function vs. Policy Search Methods [23]:

  In value function methods, we first find the optimal value function, and then compute the optimal policy from the value function. Some examples include value iteration [25], policy iteration [25], SARSA [27] etc.

  Another approach for solving an MDP is to directly search in the space of policies. Meta-heuristic search [28] or local greedy methods such as policy gradient algorithms are usually used, since the number of policies is exponential with respect to the size of the state space. The policy is taken to be an arbitrary differentiable function of a parameter vector in these algorithms, and the search in the policy space is directed by the gradient of a performance function with respect to the policy parameters [28].

  There is a third class which combines the previous two. It uses policy gradient algorithm to search in the policy space, while estimating a value function. These algorithms are called actor-critic [29]. Actor-critic methods are based on the simultaneous online estimation of the actor and the critic. The actor corresponds to conventional action-selection policy and the critic corresponds to value function. These problems are separable, but are solved simultaneously to find an optimal policy.

Bayesian reinforcement learning leverages methods from Bayesian inference to incorporate information into the learning process. It assumes the prior information about the problem in a probabilistic distribution for the system is provided, and that new information can be incorporated using standard rules of Bayesian inference. Usually, the information can be encoded by a parametric representation which describes the system dynamics. BMDP is a key model for Bayesian reinforcement learning.

There are various classes of approximate algorithms for estimating the value function in the

BMDP. The goal is to compute an optimal policy that maximizes the expected return over the hyper-state of the BMDP. Since the state space in the BMDP is usually large, it is intractable to compute the policy within a practical time or scale. One solution is to devise approximate algorithms that leverage structural constraints to achieve computationally feasible solutions.

There are a couple of approaches to solve BMDP based on offline value approximation. Finite-State Controllers (FSC), proposed in [30], compactly represents the optimal policy of a BMDP problem, which reduces the size of the policy domain. Then the optimal FSC policy is computed in the reduced domain. In general, this method is computationally practical only for small problems with limited number of states. For many real-world domains, the number of states needed to achieve good performance is far too large. Another approximate offline algorithm to solve the BMDP problem is BEETLE [31]. It was originally designed for partially observable Markov decision process (POMDP) planning. In BEETLE, hyper-states are sampled from random interactions with the BMDP system, which will essentially convert a BMDP problem to a POMDP problem. However, BEETLE requires that the parameter space of the system is small.

There are also a couple of approaches to solve BMDP based on online value approximation, which interleave planning and execution on a step-by-step basis. One approach is Bayesian dynamic programming [32]. The idea is to sample a model from the posterior distribution over the potential possible models, then solve a selected model using dynamic programming techniques. Models are sampled periodically. However, this approach can be very slow.

Another class of approaches to solve the BMDP problem is based on online tree search approximation. Generally speaking, instead of selecting actions using a complete characterization of the uncertainty model, these methods perform a forward search in the space of hyper-states. In [33], an online planning algorithm that estimates the optimal value function of a BMDP problem using Monte-Carlo sampling is proposed. For a given current state, a forward-search tree is grown to a fixed depth d. At each internal action node, a fixed number C of next states are sampled using a certain distribution. The values at the internal states nodes are estimated using the Bellman equation based on the value of their child nodes. The sampling-based approaches achieve low error.

However, this approach has an exponential computation complexity due to the tree structure, and so far the approach has only been applied to small BMDPs.

3.2 Previous Work on Computational Efficiency

A set of techniques to improve the MDP computation speed is covered in Approximate Dynamic Programming (ADP) [34]. A key idea in ADP is to approximate the value functions by sampling, e.g., Monte Carlo simulation. However, such sampling based approximation assumes a specific probability distribution of the transitions, which may not always be valid. State aggregation is one approximation technique, whose speedup is proportional to solution quality loss [18, 34].

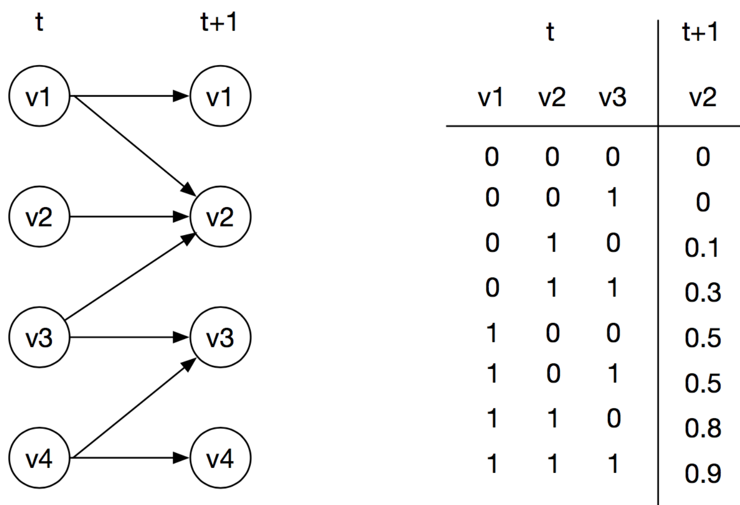
Another approach is to approximate the value function by neural networks, using so-called neuro-dynamic programming [35]. However, it suffers from the inflexibility to goal changes or generalizations, since the value functions need to be retrained if the goal changes. In [13, 36, 37], the sampling technique is further applied in the BMDP. These techniques are based on the Monte-Carlo tree search and make use of imperfect knowledge on transition probabilities and do not rely on assumptions of the probability distributions.

In order to solve the runtime issue, hardware acceleration techniques have not received much attention. A GPU-based parallel MDP technique is introduced in [38] and [39], but it is for the conventional MDP without considering partial knowledge of transition probabilities.

3.2.1 Previous Work on Improving the Scalability

In order to improve the scalability, memory reduction is the key due to the exponential memory utilization of the MDP/BMDP problem and the limited memory size on modern computing platform. Factored model [2] reduces the storage size of transition probabilities by exploiting the independence among certain state variables. The state space can be encoded by the the combinations of state variables. Hence the number of state variables is always smaller than the size of the state space. An example of factored MDP is shown in Fig. 3.1. In Fig. 3.1, suppose there are 4 state variables, and each variable has binary values. Hence there are $2^4 = 16$ states in total. The

left side indicates the dependency among the variable. For example, the value of $v2$ at time $t+1$ depends on the values of $v1, v2$ and $v3$ at time t. The table on the right side shows the transition probability of v2 being 1 at time $t+1$. If a TPM is used to represent the MDP model, then the size of the TPM is 16×16. However, by using the factored model, the size of the table on the right side for all the 4 variables is $2^3 \times 4$, which is smaller than TPM. However, the factored



t			t+1
v1	v2	v3	v2
0	0	0	0
0	0	1	0
0	1	0	0.1
0	1	1	0.3
1	0	0	0.5
1	0	1	0.5
1	1	0	0.8
1	1	1	0.9

Figure 3.1: An Example of Factored Model

model is restricted to cases where the dependence/independences among the states is known in advance, which is quite restrictive for wide real-world applications. Moreover, it is not clear how to perform Bayesian update for the factored model. Hence, it is very difficult, if not impossible, to apply the factored model in BMDP. Algebraic Decision Diagram (ADD) [40] [41] and Multi-valued Decision Diagram (MDD) [42] based approaches can also reduce the memory utilization of $\overline{\mathcal{P}}$ matrices in classical MDP, where ADDs and MDDs are used to directly store the unique entries of the $\overline{\mathcal{P}}$ matrices in the leaf nodes (without sampling, which our work does). The ADD-based approaches also require a conditional probability table for the ADDs to represent the $\overline{\mathcal{P}}$ matrix, which is usually unavailable in the BMDP problem formulation.

In our approach, we also explore the sparsity of the TPM in the MDP/BMDP problems. Practi-

cal MDP problems often have sparse TPMs [43], in which most of the elements are zero or close to zero. The most widely used sparse matrix storage format is Compressed Sparse Row (CSR) [44]. Compared to full matrix storage, in which every single entry is stored in memory regardless of its value, CSR only stores the non-zero entries of the matrix. The CSR format has some overhead to store the row/column locations of the non-zero entries. Obviously the overall efficiency of the CSR format depends heavily on the sparsity ratio SR (or the percentage of zero entries compared to the matrix size) of the matrix. Another commonly used format is the Jagged Diagonal Storage (JDS) format which is particularly suitable for iterative methods and sparse matrix vector multiplications [45]. The main advantage of JDS is that it has a lower overhead when representing long vectors. However since JDS uses the explicit positional information, its overhead still can't be ignored and, like CSR it suffers from the problem of indexed memory accesses.

4. PART 1: PARALLEL BMDP ON A GPU PLATFORM

In this chapter, we present a parallel BMDP computing approach on a GPU platform. We use Markovian Genetic Regulatory Network (GRN) regulation as an application platform to describe our approach. A recently developed approach to precision medicine is the use of Bayesian Markov Decision Processes (BMDPs) on Gene Regulatory Networks (GRNs). Due to very limited information on the system dynamics of GRNs, the BMDP must repeatedly conduct exhaustive search for a non-stationary policy, and thus entails exponential computational complexity. This has hindered its practical application to date. With the goal of overcoming this obstacle, we investigate acceleration techniques, using the Graphic Processing Unit (GPU) platform, which allows massive parallelism. Our GPU-based acceleration techniques are applied with two different MDP approaches: the Optimal Bayesian Robust (OBR) policy and the Forward Search Sparse Sampling (FSSS) method. Simulation results demonstrate that our techniques achieve a speedup of two orders of magnitude over sequential implementations. In addition, we present a study on the memory utilization and error trends of these techniques.

4.1 Background and Introduction

Developing effective therapeutics is a fundamental issue in translational genomics and precision medicine, especially when faced with complex and individualized genetic diseases such as cancer. The variability of genomic makeups between patients and even the cells within a patient (caused by the randomness of genetic mutations) demands highly personalized treatment strategies. Gene Regulatory Network (GRN) models play a crucial role in this context, as one can search for optimal intervention within a model-based framework, and use these predictions to devise intervention strategies in real scenarios. However, due to the inherent complexity and variability of the biological processes involved in regulatory systems, in most cases, little precise knowledge regard-

©2016 IEEE. Reprinted, with permission, from He Zhou, Jiang Hu, Sunil P. Khatri, Frank Liu, Cliff Sze, Mohammadmahdi R. Yousefi, GPU Acceleration for Bayesian Control of Markovian Genetic Regulatory Networks, IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI), 2016

ing the true underlying GRN is available. When such knowledge is available, it is in the form of regulatory pathways that, for simplicity, ignore the dynamics and stochasticity of regulations. This motivates us to introduce uncertainty into GRNs at different levels. By assuming a parameterized reference model for gene regulation dynamics, we start at the highest level, by constructing an uncertainty class on the parameters.

We use probabilistic Boolean networks (PBNs) as our reference model to represent GRNs, where network state transitions are assumed to be stochastic [46]. Since the PBN state evolution can be modeled by a Markov chain, the Markov Decision Process (MDP) theory is the natural framework of choice for constructing optimal intervention policies for PBNs. The optimal intervention policy dictates what action is optimal at every state of the network model. However, conventional MDP requires exact knowledge of the state transition probabilities, represented by a transition probability matrix (TPM), to find the optimal intervention. This is problematic in real world scenarios, since this knowledge generally does not exist.

One approach to solving this problem is to assume that there is a class of TPMs, as well as a prior probability distribution over this class, indicating the degree of belief in how much each member might represent the true, but unknown, TPM [47]. Then, dynamic programming is repeatedly applied to search for the non-stationary policy that performs well on average across this uncertainty class. Along with the policy search, the Bayes rule is applied to update the probability distribution of TPMs. Such an approach has been applied in GRN control and is called the Optimal Bayesian Robust (OBR) policy [1]. Although the OBR approach effectively addresses the challenge of partial knowledge of the TPM, its computational complexity is exponential. Several sampling-based techniques [33, 48, 36, 49, 37] are introduced to solve the similar problem of Bayesian model learning, and can help reduce the complexity.

In this work, we strive to accelerate the GRN control through parallel computing, an approach, to our best knowledge, that has not been explored before. In particular, we investigate acceleration techniques using Graphic Processing Units (GPUs) [50]. GPU cores are relatively simple and the GPU uses a single instruction multiple data (SIMD) architecture that requires careful paralleliza-

tion. These issues are addressed in our study.

In this chapter, we describe parallel acceleration techniques for the OBR algorithm [1] and the Forward Search Sparse Sampling (FSSS) method [13]. Simulations on one thousand test-cases show that our GPU-OBR and GPU-FSSS achieve an average speedup of $498\times$ and $378\times$, respectively, compared to the sequential variants of these algorithms. Our simulation results also exhibit significant improvement in scalability with respect to problem size. Additionally, we present a comparison of memory utilization of these techniques and errors arising from the FSSS method.

4.2 Bayesian Control Of GRNs

4.2.1 Problem Formulation

A GRN is modeled as a PBN with a set of nodes $V = \{v_1, v_2, ..., v_N\}$. Each $v_k \in V$ represents a gene in the GRN, and it only has two expression values: on (1) or off (0). Any combination of expression values of these N nodes is a gene state s_i. A PBN with N nodes will have 2^N gene states: $S = \{s_1, s_2, ..., s_{2^N}\}$. Due to the interaction among genes, the gene states of the PBN will change stochastically over time as a Markov process. More specifically, the state z_k at time step k depends on only the state z_{k-1}. If $z_{k-1} = s_i \in S$ and $z_k = s_j \in S$, this dependence can be described by the probability of transition from s_i to s_j. We use a Transition Probability Matrix (TPM) $\mathcal{P}$ whose size is $2^N \times 2^N$, to model these stochastic transitions among gene states.

The action a_i applied on the GRN flips the expression value of a specific gene, at a specific step i. It is modeled as a control action to the Markov process. Without loss of generality, we use a simple action space $A = \{0, 1\}$, where action $a = 1$ means that the intervention is applied and $a = 0$ implies no intervention. As such, the state transition also depends on action a. Assuming a sequence of t state transitions $Z = \{z_1, a_1, z_2, a_2..., a_{t-1}, z_t\}$, where z_t is state at time t and a_t is the action taken at time t, then state transition probability $\mathcal{P}_{ij}(a) = Prob.(z_{k+1} = s_j \mid z_k = s_i, a_k = a)$, where $1 \leqslant k \leqslant t$.

With only a partial prior knowledge of the TPM, it is assumed that there is a family of TPMs following probability distribution $\pi(\mathcal{P})$, which is parameterized by a matrix $[\alpha_{ij}]$ with exactly the

same size as the $\mathcal{P}$ matrix. To account for the uncertainty of the TPM, the state s_i is extended to a hyperstate $(s_i, \pi(\mathcal{P}))$, and can be also represented as $(s_i, [\alpha_{ij}])$. In the Markov process with control actions, the $\pi(\mathcal{P})$ is updated to obtain the posterior distribution according to the Bayes rule. More details about this model can be found in [32, 51]. This model can fit into the framework of the BMDP as described in Section 2.2.

4.2.2 OBR Algorithm

To solve Eq. (2.3) described in Section 2.2, a robust method using successive approximation is proposed in [22], which is referred as OBR. The main idea is to recursively approximate $J^*(s_i, \alpha)$, and after finite K iterations, the approximation of $J^*(s_i, \alpha)$ is expected to converge. Suppose we have some initially defined $J_0(s_j, \alpha)$ according to biological knowledge, then the recursive relationship is defined as follows:

$$J_{k+1}(s_i, \alpha) = \max_{a \in A}\{\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda J_k(s_j, \alpha')]\} \tag{4.1}$$

where k is the iteration index. Eq. (4.1) induces a decision tree structure. Suppose $N = 2$ and $K = 2$ and we want to know which action to take when the current state of PBN is s_1, then the corresponding tree is shown in Fig. 4.1. Every circle node is a hyperstate node and every square node is an action node. Our goal is to decide which action to take at the root node so that Eq. (2.3) will be satisfied for the initial state at the root node. Since every node will have a unique transition path from the initial state, the $[\alpha]$ matrix will also be different. This induces every node in this tree to have a specific hyperstate. This also means our computational flow needs to traverse every node inside the tree and calculate the J value for each node in order to get the J value of the root. Such a computation is intractable since the size of the tree is $(|S| \times |A|)^K$, which is exponential with respect to the problem size.

4.2.3 FSSS Algorithm

Forward Search Sparse Sampling (FSSS) [13] is a heuristic for solving the BMDP problem defined in Eq. (2.3). Instead of using an approximate $J(s_i, \alpha)$ value as the OBR algorithm does,

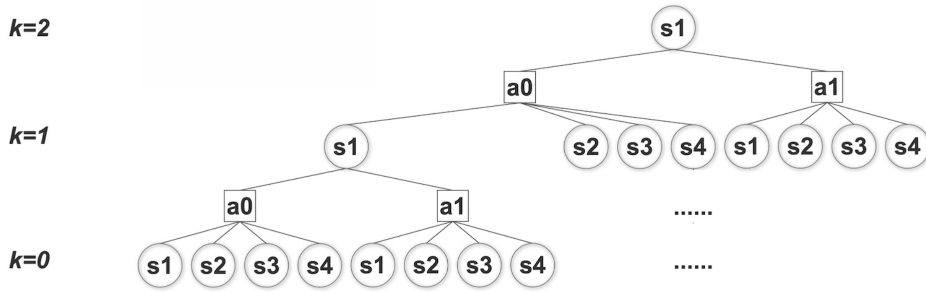


Figure 4.1: Computational Structure of OBR

FSSS calculates the upper and lower bounds of $J(s_i, \alpha)$ values for each node. FSSS samples only critical transitions, which have relatively large $\overline{\mathcal{P}}_{ij}(a)$, instead of examining every node in the tree. This is the key reason for the speedup of FSSS. Eventually, one action in the action space A will have a lower bound larger than the upper bound of other actions. This action will be the optimal action that we look for. Further details and theoretical proofs are described in [13].

4.3 GPU-based Acceleration

4.3.1 Parallelization of OBR

In order to accelerate the OBR method, we parallelize it on a GPU. From Fig. 4.1, one can see that J value estimations of circled nodes at the same tree level are independent of each other, and can be executed in parallel. One GPU thread with an identifier tid is assigned to one circled node. We traverse the tree in Fig. 4.1 level by level from the level just above the bottom level to the root. A tree index k with a smaller value is closer to the bottom level of the tree. Our parallelization scheme for OBR implemented on the GPU is referred as GPU-OBR and is described in Algorithm 1, where steps 7-10 are executed in parallel. On the GPU, it is easy to assign an identifier to each thread, which means GPU-OBR can be mapped cleanly on a GPU. Since the tree in Fig. 4.1 has a exponential size with respect to N, it requires a large amount of memory, especially when updating $\pi(\mathcal{P})$. $\pi(\mathcal{P})$ can be parameterized as an $[\alpha]$ matrix with a size of $|S|^2$ as described in Section 4.2.1. We need to deal with the largest number of nodes when $k = 1$ (in particular, $|S| \times |A|^{K-1}$

Algorithm 1 GPU-OBR

1: Initialize a PBN with N nodes, state space $S = \{s_1, s_2, ..., s_{2^N}\}$, action space $A = \{a_0, a_1\}$, initial node n_{init} with hyperstate $(s_{init}, \pi(\mathcal{P}))$, reward function $g_{ij}(a)$, J_0 for each state in S space and discount factor λ
2: Define K as the maximal horizon for the tree, $k = 1$
3: **while** $k \leqslant K$ **do**
4:     Get the number of state nodes $numS$ on level k
5:     Assign each node n with an ID tid
6:     **parfor** $tid \leftarrow 1, numS$ **do**
7:         Generate a transition path Z_{tid} from n_{init} to N_{tid}
8:         Based on Z_{tid}, update $\pi(\mathcal{P})$ [22]
9:         Use Eq. (2.4) to calculate $\overline{\mathcal{P}}$ matrix
10:         Use Eq. (4.1) to calculate $J_k(n_{tid})$
11:     **end parfor**
12:     k=k+1
13: **end while**

nodes), and each node will have its own $[\alpha]$ matrix. As a result, we need memory for at least $|S| \times |A|^{K-1} \times |S|^2 = 2^{(N+1)\times(K-1)+2N}$ floating numbers if we only consider 2 actions. GPUs usually don't have as much memory as CPUs do, which causes the algorithm to hit a memory bottleneck even with a relatively small N. To address the memory issue of GPU-OBR, we discard some rows of the $[\alpha]$ matrix. Although each node will have its own $[\alpha]$ matrix, when calculating J for a fixed node, only one row in the $[\alpha]$ matrix is used.

4.3.2 GPU-Based Parallel Computing for FSSS

In GPU-FSSS, the memory requirement is further reduced by exploiting the relationship between probability values and the size of S. When $|S|$ gets large, some transitions will have an extremely small $\overline{\mathcal{P}}_{ij}(a)$ since $\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a) = 1$. Such transitions will contribute little to $J_k(i)$, because $J_k(i)$ is the sum of $J_k(j)$ and g_{ij} weighted by $\overline{\mathcal{P}}_{ij}(a)$. The key idea of FSSS is to explore this fact.

Since the computation of FSSS has a similar tree structure as OBR, we borrow some ideas from GPU-OBR to parallelize FSSS. First, a top-down process is added to select critical transition paths starting from root to some hyperstate nodes with relatively large transition probability, denoted as $\overline{\mathcal{P}}_{Z_j}$. Suppose a transition path is $Z_j = \{z_1, a1, z_2, ..., a_{j-1}, z_j\}$ where z_1 is the root node, then the

$\overline{\mathcal{P}}_{Z_j}$ is defined as:

$$\overline{\mathcal{P}}_{Z_j} = \prod_{k=1}^{j-1} \overline{\mathcal{P}}_{z_k z_{k+1}}(a_k). \tag{4.2}$$

Another modification is to calculate the upper and lower bound of J value for each node instead of a specific J value. Given upper bound U_0 and lower bound L_0 for the leaf nodes in the tree, we can adapt Eq. (4.1) to calculate the upper and lower bounds for all the hyperstate nodes which are on the critical paths:

$$U_{k+1}(s_i, \alpha) = \max_{a \in A}\{\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda U_k(s_j, \alpha')]\} \tag{4.3}$$

$$L_{k+1}(s_i, \alpha) = \max_{a \in A}\{\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda L_k(s_j, \alpha')]\} \tag{4.4}$$

The modified algorithm is referred as GPU-FSSS and the top-down phase is described in Algorithm 2.

Algorithm 2 Top-down phase of GPU-FSSS

1: Same as Line 1 in GPU-OBR
2: Define a probability threshold P_{th}
3: Label all state nodes as expandable
4: Define K as the maximal horizon for the tree, $k = K - 1$
5: **while** $k \geqslant 0$ **do**
6:     Get the number of expandable state nodes $numS$ on level k
7:     Assign each node n with an ID tid
8:     **parfor** $tid \leftarrow 1, numS$ **do**
9:         Generate a transition path Z_{tid} from root to N_{tid}
10:         Based on Z_{tid}, update $\pi(\mathcal{P})$ [22]
11:         Use Eq. (4.2) to calculate $\overline{\mathcal{P}_{Z_{tid}}}$
12:         **if** $\overline{\mathcal{P}_{Z_{tid}}} < P_{th}$ **then**
13:             Label child nodes of N_{tid} as not expandable
14:         **end if**
15:     **end parfor**
16:     k=k-1
17: **end while**

Then a bottom-up phase is needed to update the bounds of the nodes on the paths we selected

in the top-down phase. This phase is very similar to GPU-OBR, but Eq. (4.1) is replaced by Eq. (4.3) and Eq. (4.4).

4.3.3 GPU Memory Utilization and Thread Organization

The GPU utilizes a single instruction multiple data (SIMD) paradigm. In this paradigm, the same instruction is run on multiple datastreams, in different *threads* that contain the same code (*kernel*). Threads run on one of several *streaming multiprocessor* (SM) on the GPU, each of which has several SIMD processors in it. Each SM has a pool of R hardware registers for its threads. The Nvidia GeForce GTX 760 GPU that we used has 6 SMs and 192 processors per multiprocessor, with $R = 256KB$ memory.

The GPU has a hierarchical memory structure. The registers have smallest latency and the global memory has the largest latency since it is not cached. In practice, one should avoid to use the global memory and use preferentially registers in order to speed up the algorithm. However, the GPU has limited number of registers R. Generally, the usage of registers is proportional to the number of operations. As a result, the computation needs to be decomposed into smaller parts. In our algorithm, we decomposed Eq. (4.1) into small parts, such that for one small part, only one binary operation is executed for each GPU kernel. Each kernel requires a certain number of registers D. The total number of threads T that can be issued simultaneously is limited, and must within the constraint $T \times D \leq R$. Thus, if large kernels are used, then potential parallelism is reduced. Making the kernels too small, on the other hand, incurs a larger thread issue overhead. Therefore, we partition the code into smaller kernels, experimenting with T until the speedup was maximized. The efficient utilization of register files also affects the block size. The threads in the GPU are grouped in multiple blocks, and each block will be assigned to a streaming multiprocessor (SM) in the GPU. Threads in the same SM share resources such as registers and shared memory. As a result, a very large block size will result in less resources for each thread. However, a very small block size is also inefficient, since there will be some resources unused.

Although we try to avoid using global memory, it's inevitable to use global memory when a large amount of data is involved. Global memory access latencies can be hidden across accesses,

which reduces the cost of global memory access. In our algorithm, the $\overline{\mathcal{P}}$ matrices are stored in global memory. Memory alignment will influence the performance of algorithms on GPU. Threads in the GPU are further grouped as "warp" when accessing the global memory. A warp has 16 or 32 threads. Global memory is accessed via 32-, 64-, or 128-byte memory transactions. When a warp access the global memory, the optimal case is that the threads of this warp are accessing memory addresses within one segment of 32, 64 or 128 bytes. Suppose now each thread in a warp needs to access a float (4 bytes) in global memory, then the size of these 32 floats will be 128 bytes. If these 32 floats are allocated in one segment, it only takes one transition to read these 32 floats from the global memory. However, if one of these 32 floats is allocated in another segment, then it will take two transitions to read these 32 floats from the global memory: it will first read the 31 floats in one segment, second read the misaligned float from the other segment. This will introduce a high latency. In our algorithm, the allocation of $\overline{\mathcal{P}}$ matrices needs to be carefully designed so that when an instruction is executed, there is no misalignment.

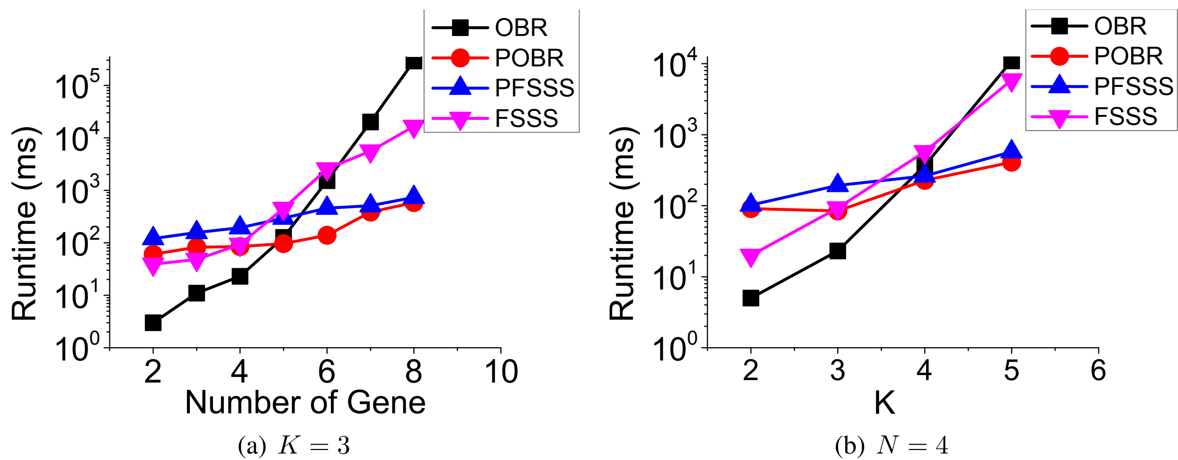
4.4 Result

We implemented the algorithms using Nvidia CUDA Toolkit 7.5, and compared them with the OBR and FSSS algorithms on a single CPU core. We ran simulations on 1000 randomly synthesized networks, generated by the process described in [1]. The main hardware features of the machine are as follows:

- CPU: Intel Core i5-4670, 4 Cores, 3.4GHz.
- RAM: 8GB DDR2 Memory.
- GPU: Nvidia GTX 760, 980MHz Clock Rate, 2GB GDDR5 Memory, 6.0 Gbps Memory Speed. [52]

In Fig. 4.2(a) and Fig. 4.2(b), we show how runtime is affected by the number of genes and the horizon depth of the tree. For GPU-FSSS, all the nodes with a probability less than P_{th} will not be expanded any more. $P_{th} = PB^L$, where $L \in [1, K]$ is distance from the current processing

level to the root, and $PB = 0.01$. The term PB is used as the *probability base*, which is used for constructing P_{th}. As shown in Fig. 4.2(a), our parallel implementations have a better scalability with respect to N compared with OBR. Due to the computational overheads, when the problem size N is small, the speedup of GPU-OBR and GPU-FSSS with respect to OBR is limited. However, the speedup is better for larger problem sizes. The best speedup occurs for $N = 8$ in which GPU-OBR has a $498\times$ speedup and GPU-FSSS is $378\times$ faster over OBR. GPU-FSSS needs to traverse the tree twice, and hence GPU-FSSS is generally slower than GPU-OBR. For GPU-FSSS, both the input size and the probability threshold influence the runtime, as shown in Fig. 4.3. The missing data points in Fig. 4.3 could not be computed due to memory limitation.



(a) $K = 3$

(b) $N = 4$

Figure 4.2: Runtime of different input size, $P_{th} = 0.01^L$

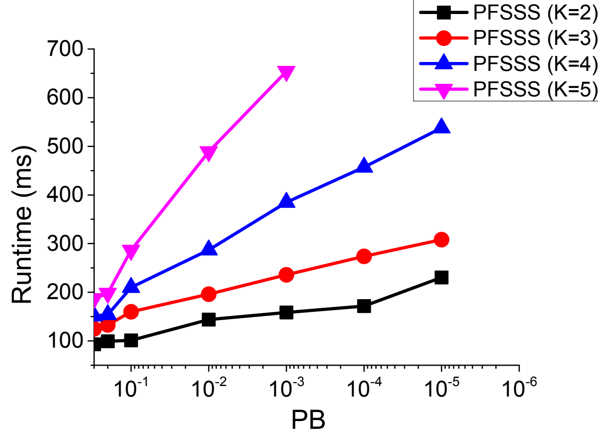


Figure 4.3: Run time of GPU-FSSS with different K, $N = 5$

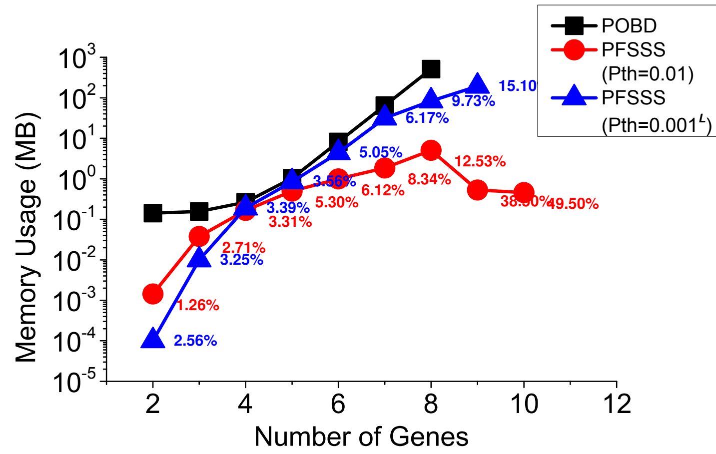
In Fig. 4.4(a), we report the memory usage and corresponding error. Since OBR has been proved to yield a suboptimal J value in [1], here we will focus on the deviation (from OBR) of the bounds at root node obtained by GPU-FSSS. The error of the bound is defined as

$$error = \frac{1}{2}(\frac{|UB - J_{OBR}|}{J_{OBR}} + \frac{|J_{OBR} - LB|}{J_{OBR}}) \times 100\%. \tag{4.5}$$

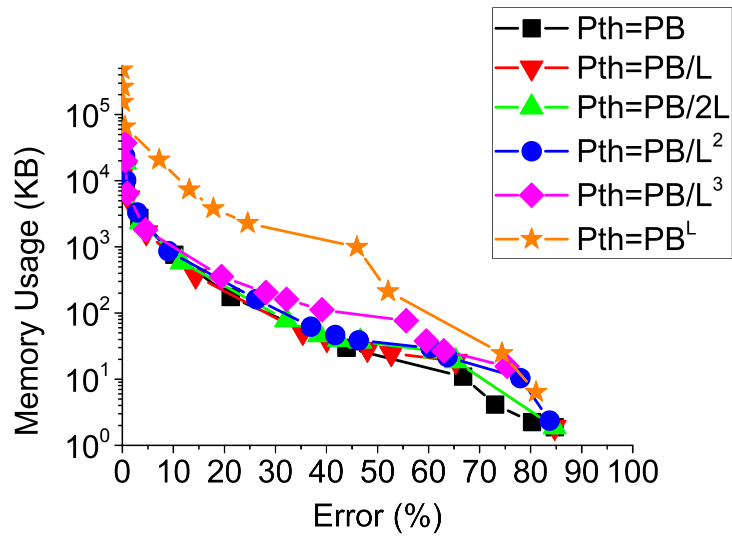
where UB is upper bound and LB is lower bound. The performance of GPU-FSSS is naturally sensitive to the probability threshold P_{th}, so we present several P_{th} schemes. These schemes vary P_{th} for different levels of the tree. Fig. 4.4(b) presents the relationship between memory usage and error under different type of P_{th} schemes, with varying PB. We observe that the error doesn't decrease as much as the increase of the memory usage for larger N. To fully utilize the limited memory, a careful choice needs to be made on how to balance result quality and resource cost.

4.5 Conclusion

In this chapter, we alleviate the intractable computation complexity of the control problem on GRNs with uncertainty of transition probability by parallelizing two MDP-based methods: OBR and FSSS on the GPU. Our experimental results show a significant speedup. We also explore the trade-off between result quality and resource cost. In the future, one could optimize memory utilization by applying more efficient data structures to allow the approach to solve larger problem



(a) Memory w/ error label, $K = 3$



(b) Memory vs. error

Figure 4.4: Memory and error performance of GPU-FSSS and GPU-OBR

in a reasonable time.

5. PART 2: A DUPLEX SPARSE STORAGE (DSS) SCHEME FOR BMDP ON GPU PLATFORM

In this chapter, we introduce a novel Duplex Sparse Storage (DSS) scheme to represent the expected TPM in the BMDP framework, and develop a BMDP solver with the DSS scheme on a heterogeneous GPU-based platform. By employing the Optimal Bayesian Robust (OBR) policy, Bayesian Markov Decision Process (BMDP) can be used to solve many practical problems. However, due to the "curse of dimensionality", the data storage limitation hinders the practical applicability of the BMDP. To overcome this impediment, we propose a novel Duplex Sparse Storage (DSS) scheme in this chapter. We applied DSS with OBR on the Graphic Processing Unit (GPU) platform. The simulation results demonstrate that our approach achieves a $5\times$ reduction in memory utilization with a 2.4% "decision difference" and an average speedup of $4.1\times$ compared to the full matrix based storage scheme. Additionally, we present the tradeoff between the runtime and result accuracy for our DSS techniques versus the full matrix approach. We also compare our results with the well known Compressed Sparse Row (CSR) approach for reducing memory utilization, and discuss the benefits of DSS over CSR.

5.1 Introduction and Background

In order to solve the memory problem, we exploit the observation that the expected TPMs tend to be sparse. For example, in the GRN regulation problem, the expression of one gene usually depends on only a few other genes [14]. This sparsity feature of the TPM motivates us to investigate sparse data storage techniques to address the memory issue. Since both the prior and posterior distributions of the TPM in BMDP are known, we can exploit the sparsity of the TPM. The Compressed Sparse Row (CSR) format is an efficient approach to store a sparse matrix [15]. Obviously the overall efficiency of the CSR format heavily depends on the sparsity ratio SR (or the percent-

©2017 ACM. Reprinted, with permission, from He Zhou, Sunil P. Khatri, Jiang Hu, Frank Liu, Cliff Sze, Fast and Highly Scalable Bayesian MDP on a GPU Platform, Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics, 2017

age of zero entries compared to the matrix size) of the matrix. CSR is a general purpose model to store elements with an arbitary data range. However, since the TPM has special properties, we are motived to propose an enhanced storage scheme based on CSR to store the TPM in the BMDP problem. When the sparsity ratio SR of the matrix is high, CSR can be expensive when the size of the sparse matrix is large, since the lower bound of the CSR format is the number of rows of the original sparse matrix. Our DSS scheme offers two improvements over CSR. To address potential accuracy issues, we lump all trivial values (the smallest $(1-R) \times 100\%$ of the entries, where R is the percentage of entries the DSS stores) of the TPM into a single entry whose value is the average of all the $(1-R)\times 100\%$ trivial values. Furthermore, we enhance the CSR to gain better efficiency, by duplex use of the same words of storage.

We use "GPU-based OBR" (or GPU-OBR) to generically refer to the parallel OBR approach for solving the BMDP problem on GPU. The GPU-based OBR using a full matrix to represent the expected TPM in [53] is denoted as "G-Full". We implemented the GPU-based OBR with CSR and our proposed DSS method to represent the expected TPM (called "G-CSR" and "G-DSS" respectively). We have simulated G-DSS on the BMDP-based gene regulatory network (GRNs) control problem defined in [1, 54]. Simulations on 500 synthetic GRNs demonstrated that G-DSS achieves a $5\times$ reduction in memory utilization while introducing only 2.4% "decision difference" in the largest GRN compared with G-Full. The corresponding memory reduction and "decision difference" for G-CSR are $2.1\times$ and 6.3% respectively. Our simulation results of DSS on the GPU exhibit an average speedup of $4.1\times$ compared with using full sparse matrix on the GPU.

The key contributions of this chapter is that we developed DSS which is used to alleviate the memory issue in G-Full. Compared to CSR, the DSS scheme has a smaller memory utilization, a better accuracy and a better L^{∞} norm. Compared to the full matrix storage scheme, DSS also has a smaller memory utilization, and a better speedup. Moreover, the DSS is designed to enable efficient Bayesian updates in GPU-based OBR, by eliminating the need to reallocate memory during Bayesian updates.

5.1.1 Preliminaries - CSR Storage Format

Compressed Sparse Row (CSR) is a storage format for sparse matrices and is widely used for scientific and engineering computations [44]. Fig. 5.1 shows an example of CSR. The 4×4 matrix M in Fig. 5.1 is sparse since most of the elements are zero. The bold italic elements are non-zero elements, and are elements of interest. Clearly, there is no need to store the zero-valued entries.

M

	0	1	2	3
0	0	***8.1***	0	0
1	0	0	***10.4***	***8.6***
2	0	0	0	0
3	0	***9.8***	0	0

data	***8.1***	***10.4***	***8.6***	***9.8***	
colIndex	1	2	3	1	
rowPtr	0	1	3	3	4

Figure 5.1: CSR Example

The CSR format stores the sparse matrix using three vectors: the *data* vector, the *colIndex* vector and the *rowPtr* vector. The *data* vector stores the non-zero elements in a row-major fashion. In the *colIndex* vector, $colIndex[i]$ stores the column position of $data[i]$ in M. The *rowPtr vector* is defined as follows:

$$rowPtr[0] = 0$$
$$rowPtr[i] = rowPtr[i-1] + NZ(i-1), \qquad i > 0$$

where $NZ(i)$ is the number of non-zero elements in the ith row of the original M matrix. Algorithm 3 shows how to look up M_{ij} in the CSR storage format.

In most scientific and engineering computations, the information of interest in M is the value, the row position index and the column position index of the non-zero elements. Although the storage cost per element in CSR is higher, the total storage cost of CSR is significantly smaller

Algorithm 3 Look Up M_{ij} in the CSR Format

```
1:  StartIndex = rowPtr[i];
2:  EndIndex = rowPtr[i+1];
3:  Found = 0;
4:  for k = StartIndex; k < EndIndex; k++ do
5:      if colIndex[k] == j then
6:          Found = 1;
7:          return data[k]
8:      end if
9:  end for
10: if Found == 0 then
11:     return 0
12: end if
```

than the full matrix when M is sparse. If the size of the original matrix is $N \times N$ and the total number of the non-zero elements is N_{nz}, then the memory utilization of the full matrix is $O(N^2)$. However, in the CSR format, the memory utilization is $O(2N_{nz} + N)$, which is less than that of the full matrix if M is large and sufficiently sparse.

Although the CSR format alleviates the memory utilization compared to the full matrix, the memory utilization can still be large when the original sparse matrix has many non-zero entries, since $rowPtr$ depends on the size of the matrix. Moreover, in BMDP problem settings, the matrix may have many elements close to zero, but not exactly zero. We note that the $\overline{\mathcal{P}}$ matrix in the BMDP problem has the following properties:

$$\begin{cases} 1 > \overline{\mathcal{P}}_{ij} > 0 \\ \sum_{\forall j \in S} \overline{\mathcal{P}}_{ij} = 1 \end{cases} \tag{5.1}$$

In such a scenario, there could be a significant number of small values in each row. Storing such small values using the CSR format can often increase the total memory requirement, and hence negate the benefit of CSR. A naive approach is to discard elements that are smaller than a threshold value and store them as zero. By doing so, we can artificially boost the sparsity of the matrix. However such an approach can have a negative impact on the accuracy of the subsequent MDP

computation steps. The approach we take in this paper is to use the Duplex Sparse Storage (DSS) format, which is described next.

5.2 Our Approach - Duplex Sparse Storage (DSS)

5.2.1 Problems of Using Conventional Full Matrix

In general, the expected TPM (i.e., the $\overline{\mathcal{P}}$ matrix) with each element defined in Eq. (2.4) can be represented by a full floating-point matrix with the size of $|S| \times |S|$. However, in the GPU-based acceleration approach, each action node (the square nodes in Fig. 4.1) requires a row of the $\overline{\mathcal{P}}$ matrix to perform the computation of Eq. (2.6). Therefore, using an $|S| \times |S|$ matrix leads to a $(|S| \times |A|)^K$ memory utilization complexity. Such a prohibitively expensive memory complexity limits the applicability of the GPU-based BMDP approach on large problems. In this chapter we focus on how to improve the storage cost of the $\overline{\mathcal{P}}$ matrix.

In many practical problems, the $\overline{\mathcal{P}}$ matrix is likely to be sparse due to the natural sparsity of the problem and its large state space. In the context of gene regulatory networks (GRNs) or more general biochemical networks, the structure of the problem tends to be sparse [55, 56, 57, 58]. Hence the corresponding $\overline{\mathcal{P}}$ matrix is also sparse. Generally speaking, for such problems, most elements of each row of the $\overline{\mathcal{P}}$ matrix are close to zero. However, OBR is oblivious to this natural sparsity, and performs the computation in a brute force manner. Before we can employ sparse storage techniques to this problem, we need to ensure that the sparsity is conserved after K Bayesian updates during OBR. Fig. 5.2 shows the histogram of a randomly selected row in a 16×16 $\overline{\mathcal{P}}$ matrix after 5 Bayesian updates by following 100 different random histories. In this figure, each curve is associated with one Bayesian update history. Note that in Fig. 5.2, many of these 100 curves overlap. Fig. 5.2 indicates that if the prior $\overline{\mathcal{P}}$ matrix is sparse, then after K Bayesian updates (usually $K \leq 5$ in practical cases), the posterior $\overline{\mathcal{P}}'$ matrix tends to remain sparse as well. Due to this feature of property sparsity conservation, sparse matrix storage techniques can be effective in alleviating the high memory complexity throughout the entire G-Full approach.

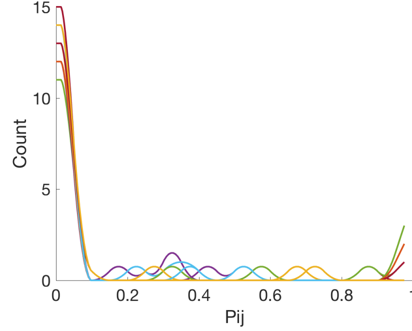


Figure 5.2: Histogram of P_{ij} after 5 Bayesian Updates

5.2.2 DSS Format in BMDP

In DSS, instead of using three vectors, we only use one vector $data_col$ to store all the information needed (in a row-major fashion). A DSS for an $N \times N$ matrix is a vector composed by N components, where each component corresponds to one row in the matrix. Each component contains three fields:

1. X words to save the top X significant entries of its corresponding row. Since we would like to store the matrix elements with relatively large values, we store the top X largest values from every row (such that $R = X/N \times 100\%$, where N is the size of one row of $\overline{\mathcal{P}}$).

2. $K-1$ words reserved for matrix entries that would change from a "trivial" value to significant value in a future Bayesian update.

3. One word for storing the average value of the $N - X$ "trivial" entries.

4. Each significant entry is stored in a duplex fashion, using a single word to store the value of the entry (in 12-bit binary) and its column position (in 20-bit binary).

Suppose a sequence of state transitions $Z_T = (z_1, d_1, z_2, d_2..., z_m, ..., z_{M-1}, d_{M-1}, z_M)$ is observed. Assume $z_m = s_i$ and $z_{m+1} = s_j$, and from time step 0 to time step $(m + 1)$, there are C state transitions from s_i to other states in Z_T. Then the ith row of the $\overline{\mathcal{P}}'$ matrix at time step $(m+1)$

can be obtained from the $\overline{\mathcal{P}}$ matrix at time step m. The resulting $\overline{\mathcal{P}}'$ can be derived from Eq. (2.4) and Eq. (2.5), and is shown in Eq. (5.2). All the other rows of the $\overline{\mathcal{P}}'$ matrix are unchanged by this Bayesian update.

$$\begin{cases} \overline{\mathcal{P}}'_{ik} = \frac{C \times \overline{\mathcal{P}}_{ik}}{C+1}, \quad k \neq j \\ \\ \overline{\mathcal{P}}'_{ij} = \frac{C \times \overline{\mathcal{P}}_{ij} + 1}{C+1} \end{cases} \tag{5.2}$$

Eq. (5.2) indicates that after the Bayesian update $\overline{\mathcal{P}}'_{ij}$ becomes larger, which means that $\overline{\mathcal{P}}'_{ij}$ may qualify to be stored in the DSS format, even though $\overline{\mathcal{P}}_{ij}$ is close to zero. This increase in $\overline{\mathcal{P}}'_{ij}$ caused by the Bayesian update requires DSS to reserve "holes" in order to store the elements which become larger due to Eq. (5.2). A BMDP problem with a horizon length of K needs to preform $K-1$ Bayesian updates in all. Hence the DSS approach needs to leave exactly $K-1$ "holes" for each row of a $N \times N$ $\overline{\mathcal{P}}$ matrix, and $N \times (K-1)$ "holes" in total for the entire $\overline{\mathcal{P}}$ matrix. This guarantees that no array re-allocations are needed in the DSS approach, for the entire BMDP computation.

Since the $\overline{\mathcal{P}}$ matrix is sparse, with a reasonable choice of the storage ratio R, we are able to represent all the "non-trivial" elements in the $\overline{\mathcal{P}}$ matrix while maintaining a small memory footprint. However, since the size of the $\overline{\mathcal{P}}$ matrix is usually large in practical BMDP problems, the number of trivial elements in the $\overline{\mathcal{P}}$ matrix can still be quite large. Even though we can neglect the contribution of a single "trivial" element to the subsequent computation, we may not be able to neglect the contribution of a large number of trivial elements. Therefore, in the DSS, we use a special cell to store the average value of the trivial elements for each row in the $data_col$ vector. From Eq. (5.1), we can easily compute the sum for the trivial elements by subtracting the sum of the non-trivial elements from 1. Hence the average of the trivial elements is also easy to compute. An intuitive way to understand this approach is that each row represents the "mass" of the probability distribution in each state. Instead of throwing away the entries with small values (which is equivalent to losing mass), we conserve the total "mass" by accumulating the small entries. Even though we keep only an approximated value of each individual small element, we improve the

memory efficiency of the BMDP method while not impacting error appreciably. The benefit is clearly demonstrated in the experimental results.

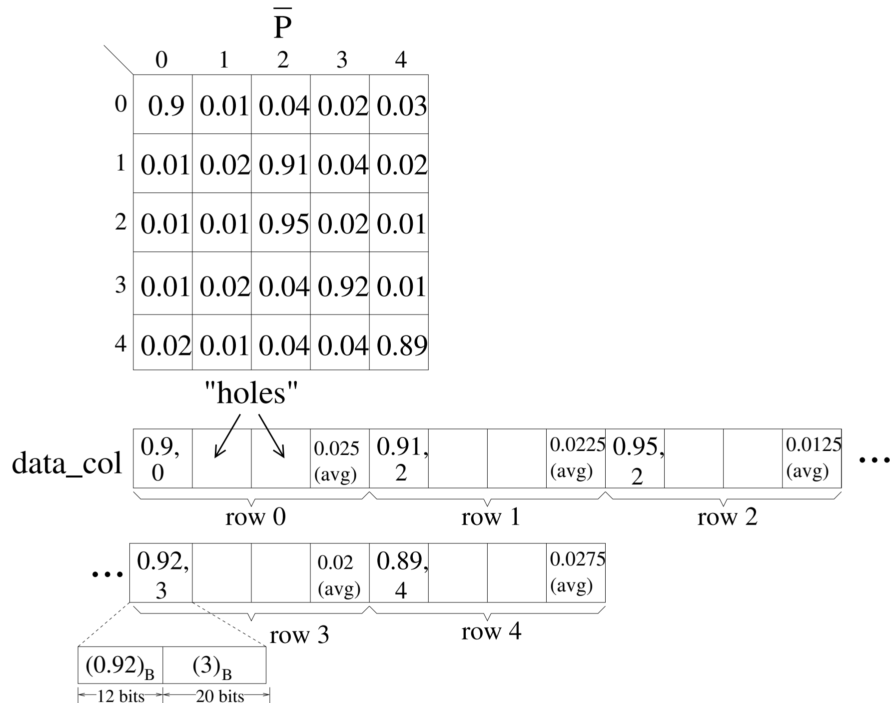


Figure 5.3: DSS Example

An example of DSS for a 5×5 $\overline{\mathcal{P}}$ matrix is shown in Fig. 5.3 with $R = 20\%$. Suppose the DSS representation in Fig. 5.3 is used for a BMDP problem with $K = 3$. Then in the $data_col$ vector, there are 2 “holes” reserved for each row (for the Bayesian update). The average of the trivial elements in each row is stored in the cell labeled “avg”. In the DSS, $(1 - R) \times 100\%$ of the entries are deemed as “trivial”. Since the number of elements to be stored in each row of the $\overline{\mathcal{P}}$ matrix is fixed, the starting position of the ith row of the $\overline{\mathcal{P}}$ matrix in the $data_col$ vector is exactly $(i \times (X + K))$. Hence the $rowPtr$ vector in the CSR format is no longer needed. The number of bits used to store the entry value and its column position is 12 and 20 respectively, which don’t match any standard data type in popular programming languages. 12-bits provides a good resolution for P_{ij} in the TPM, and 20-bits can handle a large matrix (the largest matrix size

that DSS can store is $(2^{20})^2 = 2^{40}$). The total number of bits used for every entry of $data_col$ is 32, since the native word size of the architecture we used is 32. For other applications, the number of bits used to store the value of the entries and their column position may be different.

We next discuss how a Bayesian update is performed on DSS. Once a state transition from $z_m = s_i$ to $z_{m+1} = s_j$ is observed, we need to go through a component of the $data_col$ vector in which the ith row of the $\overline{\mathcal{P}}$ matrix is stored. If $\overline{\mathcal{P}}_{ij}$ doesn't exist in the non-trivial cells of this component (i.e. $\overline{\mathcal{P}}_{ij}$ is not among the $R \times 100\%$ largest entries of row i), we need to use the average value of the trivial elements as $\overline{\mathcal{P}}_{ij}$. Then we compute $\overline{\mathcal{P}}'_{ij}$ using Eq. (5.2), and store the $\overline{\mathcal{P}}'_{ij}$ in the first empty "hole" in this component. The average value of the trivial elements in the ith row of $\overline{\mathcal{P}}'$ also needs to be updated.

5.2.3 Parallel DSS-based OBR

As discussed earlier, the G-Full approach presented in [53] is fast but has memory problems. To alleviate the memory utilization issues, we use DSS to store the expected TPM data in GPU-based OBR. To accomplish this, the following extra algorithms were implemented for DSS on the GPU:

- Look up $\overline{\mathcal{P}}_{ij}$
- Bayesian Update

5.2.3.1 Look Up $\overline{\mathcal{P}}_{ij}$

Algorithm 4 shows how to look up $\overline{\mathcal{P}}_{ij}$ in the DSS format on the GPU. In step 8, a bitwise *AND* operation is used to mask the contents of $data_col[i]$ so that the value of the element and the column position information can be separated. Since the value of the data is located in the most significant 12 bits in $data_col[i]$, a bitwise shift operation is needed to return the correct $\overline{\mathcal{P}}_{ij}$. An example of how look-up works in DSS on GPU is shown in Fig. 5.4.

The $\overline{\mathcal{P}}$ matrix and the corresponding DSS are the same as Fig. 5.3, where $X = 1$ and $K = 3$. Since some "holes" in DSS might be filled due to the Bayesian update, the $(K - 1)$ "holes" also need to be checked in the look-up. Hence a total of $(X + K - 1)$ cells in the $data_col$ vector

Algorithm 4 Look Up $\overline{\mathcal{P}}_{ij}$ in the DSS Format

```
1:  Found = 0
2:  StartIndex = (X + K) × i;
3:  EndIndex = (X + K) × (i + 1) − 1
4:  Assign (X + K − 1) threads;
5:  for ALL the threads do
6:      tid ← getThreadID;
7:      loc = StartIndex+tid;
8:      if (data_col[loc] AND 0x000FFFFF) == j then
9:          Found = 1
10:         return (data_col[loc] AND 0xFFF00000)>> 20
11:     end if
12: end for
13: if Found == 0 then return data_col[EndIndex]
14: end if
```

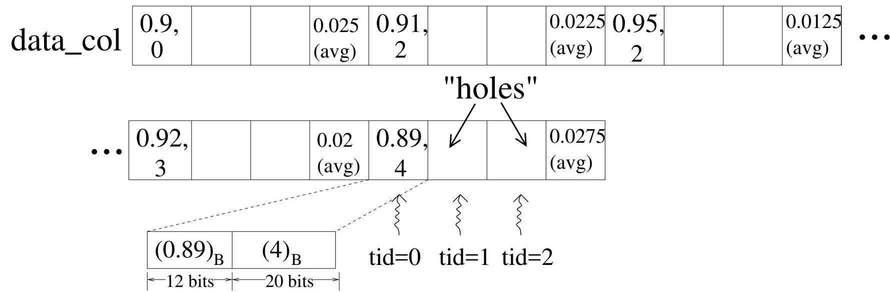


Figure 5.4: Example of Looking Up $\overline{\mathcal{P}}_{44}$ on GPU

need to be checked in parallel to find any $\overline{\mathcal{P}}_{ij}$. Suppose we look up $\overline{\mathcal{P}}_{44}$. In this case, 3 threads are launched by the GPU in Fig. 5.4 with a tid starting from 0. Each thread checks a different cell in the $data_col$ vector, and the location of its cell is defined in step 7 in Algorithm 4. In step 8, a bitwise *AND* operation is used to mask the contents of $data_col[loc]$ so that the value of the element and its column position can be separated. Since the value of the data is located in the most significant 12 bits of $data_col[loc]$, a bitwise shift operation is needed to return the correct $\overline{\mathcal{P}}_{ij}$. If we neglect the overhead incurred by the thread launching on the GPU, the time complexity of Algorithm 4 is $O(1)$.

5.2.3.2 Bayesian Update

The GPU-based OBR algorithm proposed in [53] yields a tree computation structure. Fig. 5.5 shows an example of how the Bayesian update works. For the BMDP in Fig. 5.5, assume that $S = \{s_0, s_1, s_2, s_3\}$ $A = \{a_0, a_1\}$, and $K = 3$. If we want to compute the J value of the black circled node in Fig. 5.5 by following Eq. (2.6), we need to know the 0th row in $\overline{\mathcal{P}}'$ after following the transition $Z_3 = \{s_0, a_0, s_3, a_0, s_0\}$. By following Eq. (2.5) and Eq. (2.4), $\overline{\mathcal{P}}_{03}$ needs to be updated. Since for the GPU-based OBR algorithm, the problem is the memory issue instead of runtime, we compute $\overline{\mathcal{P}}'$ from $\overline{\mathcal{P}}$ without storing the internal result of $\overline{\mathcal{P}}$ at the circled node labeled $S3$ in Fig. 5.5. Algorithm 5 describes how to do Bayesian update on the ith row of the $\overline{\mathcal{P}}$ matrix in

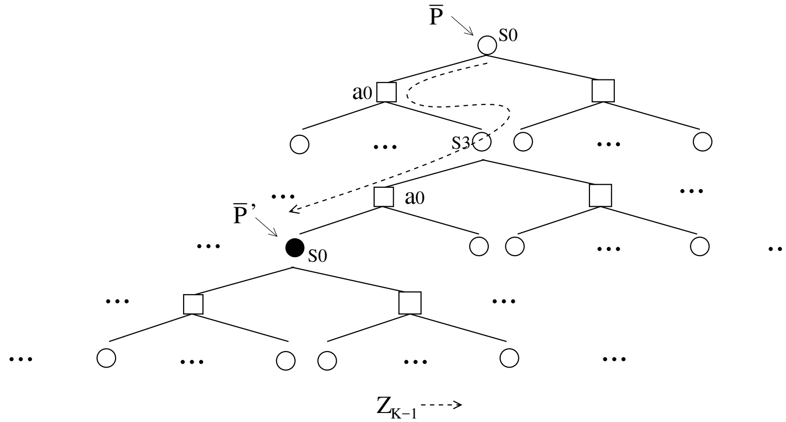


Figure 5.5: Bayesian Update Example

DSS when a sequence of state transition Z_M is observed, where $Z_M = \{z_1, d_1, z_2, d_2, ...d_{M-1}, z_M\}$, and $s_i = z_M$. Similar to Algorithm 4, in Algorithm 5, $(X + K)$ threads are launched to update the ith row of the $\overline{\mathcal{P}}$ matrix, and each thread processes one entry whose location in the $data_col$ vector is defined in step 7. Each thread goes through the state transition sequence and decides what to do with its own $data_col[loc]$. There are 4 possible cases:

- The state transition from z_m to z_{m+1} doesn't contribute to the Bayesian update on the ith row

Algorithm 5 Bayesian update on the ith row of $\overline{\mathcal{P}}$ matrix when a sequence of state transition Z_M is observed.

```
1:  StartIndex = (X + K) × z_M;
2:  AvgIndex = StartIndex + (X + K − 1)
3:  set flag = 0 in global memory.
4:  Assign (X + K) threads;
5:  for ALL the (X + K) threads do
6:      tid ← getThreadID;
7:      loc = StartIndex+tid;
8:      C = 0; j = (data_col[loc] AND 0x000FFFFF);
9:      for m=1; m<M; m++ do
10:         if z_m == z_M then
11:             C = C+1;
12:             if z_{m+1} == j then
13:                 increase "P̄_ij" in data_col[loc] using Eq. (5.2);
14:                 flag =1;
15:             else
16:                 decrease "P̄_ik" in data_col[loc] using Eq. (5.2);
17:             end if
18:             wait for all the threads to reach to here;
19:             if (flag==0 && tid == (X + K)) then
20:                 Insert a new DSS element into the first empty hole in the data_col vector;
21:                 update data_col[AvgIndex];
22:             end if
23:         end if
24:     end for
25: end for
```

of the $\overline{\mathcal{P}}$ matrix. In such case, Algorithm 5 does nothing.

- The state transition from $z_m = s_i$ to $z_{m+1} = s_j$ contributes to the Bayesian update on the ith row of the $\overline{\mathcal{P}}$ matrix. The current thread processes the DSS entry including $\overline{\mathcal{P}}_{ij}$. In this case, this thread executes step 13 and step 14 in Algorithm 5 to increase its $\overline{\mathcal{P}}_{ij}$ by following Eq. (5.2).

- The state transition from $z_m = s_i$ to $z_{m+1} = s_j$ contributes to the Bayesian update on the ith row of the $\overline{\mathcal{P}}$ matrix. The current thread doesn't process the DSS cell including $\overline{\mathcal{P}}_{ij}$. In this case, this thread executes step 16 in Algorithm 5 to decrease its $\overline{\mathcal{P}}_{ik}$ by following Eq. (5.2). Note that the "average" of trivial elements also just needs to decrease as $\overline{\mathcal{P}}_{ik}$ in Eq. (5.2).

- The state transition from $z_m = s_i$ to $z_{m+1} = s_j$ contributes to the Bayesian update on the ith row of the $\overline{\mathcal{P}}$ matrix. However, $\overline{\mathcal{P}}_{ij}$ was a trivial element in time step m. In this case, the $(X + K)^{th}$ thread executes from step 19 to step 21 in Algorithm 5 to insert a new DSS element to the first available "hole" in $data_col$ vector. The most significant 12-bits in this new DSS element notated as $\overline{\mathcal{P}}'_{ij}$ is defined as follows:

$$\overline{\mathcal{P}}'_{ij} = \frac{C \times data_col[AvgIndex] + 1}{C + 1} \tag{5.3}$$

  And the least significant 20-bits in this new DSS is z_{m+1}. The data_col[AvgIndex] also needs to be updated in order to keep the sum of the ith row to be 1.

The time complexity of Algorithm 5 is $O(M)$. Since M (which is bounded above by K) is usually small in practical BMDP problems, the practical time complexity of Algorithm 5 is nearly $O(1)$. Since the G-Full method in [53] stores the full $\overline{\mathcal{P}}$ matrix, the Bayesian update in G-Full needs to traverse N elements for each black circled node in Fig. 5.5 (where N is the row length of the matrix). This means the GPU has to launch N threads for each black circled node. However, in the DSS method, only $(X + K)$ elements need to be touched, which means the GPU has to launch only $(X + K)$ threads for each black circled node. This gives the G-DSS algorithm the potential

to handle more black circled nodes (i.e. more hyperstates in the BMDP) simultaneously, which increases the speedup of the G-DSS algorithm.

5.2.4 Comparison Between DSS and CSR

Table 5.1 shows a summary of the differences between DSS and CSR in the context of storing $\overline{\mathcal{P}}$ for the GPU-based OBR. Here we assume that the CSR treats all the trivial elements (smaller

Table 5.1: Differences Between CSR and DSS

	CSR	DSS
Elements Stored	elements $>$ the threshold	X largest elements in each row (where $X/N \times 100\% = R$)
How to deal with trivial elements	Abandon (treat as 0)	Use the average value
Data Structure	3 vectors	1 vector combines of $\overline{\mathcal{P}}_{ij}$ and j
Memory Complexity	$O(2N_{th} + N)$	$O(N(X + K))$
L^{∞} Norm	≤ 1	1

than a threshold) as 0 in order to store the $\overline{\mathcal{P}}$ matrix with a small memory utilization. Let N_{th} be the number of such elements.

Let the J value computed in Eq. (2.6) using the full matrix be called J_m, the J value computed using CSR be called J_{CSR}, and the J value computed using DSS be denoted as J_{DSS}. Then $J_m > J_{DSS} > J_{CSR}$ with an appropriate choice of X. This is because in Eq. (2.6), every variable is positive and CSR abandons the trivial elements while DSS uses an average value for them. In other words, DSS contains more accurate information about the $\overline{\mathcal{P}}$ matrix compared to CSR.

In Table 5.1 we also show the L^{∞} norm of both CSR and DSS. The L^{∞} norm of the original full $\overline{\mathcal{P}}$ matrix is 1 due to Eq. (5.1). Suppose we reconstruct the $\overline{\mathcal{P}}$ matrix from the DSS format and the CSR format, and denote them as $\overline{\mathcal{P}}_{DSS}$ and $\overline{\mathcal{P}}_{CSR}$ respectively. Since DSS uses an average value to represent the trivial elements, the sum of each row in $\overline{\mathcal{P}}_{DSS}$ is still 1. However, since $\overline{\mathcal{P}}_{CSR}$ abandons some entries of the original full $\overline{\mathcal{P}}$ matrix in order to maintain a small memory utilization, the sum of each row in $\overline{\mathcal{P}}_{CSR}$ is often less than 1. If the CSR doesn't abandon any

entry in the full $\overline{\mathcal{P}}$ matrix in extreme cases, then the sum of each row in $\overline{\mathcal{P}}_{CSR}$ will be equal to 1. Therefore, we have the following relationship among the L^{∞} norms of $\overline{\mathcal{P}}$, $\overline{\mathcal{P}}_{DSS}$ and $\overline{\mathcal{P}}_{CSR}$:

$$||\overline{\mathcal{P}}_{CSR}||_{\infty} \leq ||\overline{\mathcal{P}}_{DSS}||_{\infty} = ||\overline{\mathcal{P}}||_{\infty} = 1 \tag{5.4}$$

In other words, the L^{∞} norm for P_{CSR} is inaccurate, while that of P_{DSS} matches $||\overline{\mathcal{P}}||_{\infty}$. We claim that this could be one of the reasons why the J and A values of DSS are more accurate than those of CSR, as shown in our experiments. In addition, in DSS, since the upper-bound of the number of new non-trivial elements introduced by Bayesian update is predefined and DSS reserves "holes" for these new non-trivial elements, we don't have to reallocate memory for the DSS while performing Bayesian updates. In the special case where all the elements in the original $\overline{\mathcal{P}}$ matrix are identical, the DSS method can still handle such matrices without introducing any error for the future computations.

Since the value of each element and its column position are combined in one 32-bit word in DSS, we need to perform bit-wise operations to separate the value of the element from the column position when needed. Although any contemporary computer architecture can perform this type of operation very efficiently through bitwise manipulation instructions, these operations do take up computation cycles. In some cases, when the row length N of the $\overline{\mathcal{P}}$ matrix is small and K is large, DSS may consume more memory than CSR. However, such cases are not of our interest when we are considering large BMDP problem instances.

5.3 Experimental Results

The DSS approach is implemented to store the expected TPM data in the GPU-based OBR algorithm by using Nvidia CUDA Toolkit 8.0 [59]. The element values in DSS are represented using 12-bit fractions, while the full matrix and CSR methods use 32-bit floating point values. We ran the simulations on 500 synthetic GRNs generated following the descriptions in [1]. The main hardware specifications are listed as follows:

- CPU: Intel Core i5-4670, 3.4GHz

- RAM: 8GB DDR2 Memory
- GPU: Nvidia GTX 780, 863MHz Clock Rate, 3GB GDDR5 Memory, 6.0 Gbps Memory Speed [52]

In our experiments, we compare the performance among using three different storage methods to store the $\overline{\mathcal{P}}$ matrix in the GPU-based OBR algorithm: G-DSS, G-CSR and G-Full. Runtime and memory utilization are used to quantify the effectiveness of these three methods. We also include the sequential OBR algorithm using full matrix in the results for reference, which is denoted as S-Full.

We used the differences of J values and final action decisions (denoted as J_{Full}^{DSS} and A_{Full}^{DSS} respectively) between G-DSS and G-Full to assess the accuracy of DSS. J_{Full}^{DSS} and A_{Full}^{DSS} are defined as follows:

$$\begin{aligned} J_{Full}^{DSS} &= \frac{|J_{DSS} - J_{Full}|}{J_{Full}} \times 100\% \\ A_{Full}^{DSS} &= \begin{cases} 100\%, & A_{DSS} \neq A_{Full} \\ 0\%, & A_{DSS} = A_{Full} \end{cases} \end{aligned} \tag{5.5}$$

where J_{DSS} and A_{DSS} are respectively the J value and action decision computed by G-DSS, while J_{Full} and A_{Full} are respectively the J value and action decision computed by G-Full(or S-Full). In some experiments, the G-Full algorithm ran out of memory and didn't complete. In such a case, we use the J value and action decision computed by S-Full as J_{Full} and A_{Full} respectively. J_{Full}^{CSR} and A_{Full}^{CSR} are defined similarly. Table 5.2 summarizes the notations used from Fig. 5.6 to Fig. 5.9.

Fig. 5.6 compares the differences of the performance between G-CSR and G-DSS. Fig. 5.6(a) shows that G-DSS uses less memory than G-CSR. When the number of states is larger than 32, G-DSS uses about half the memory compared to G-CSR. This is mainly because DSS stores both $\overline{\mathcal{P}}_{ij}$ and j in one word.

Table 5.2: Summary of Notations Used From Fig. 5.6 to Fig. 5.9

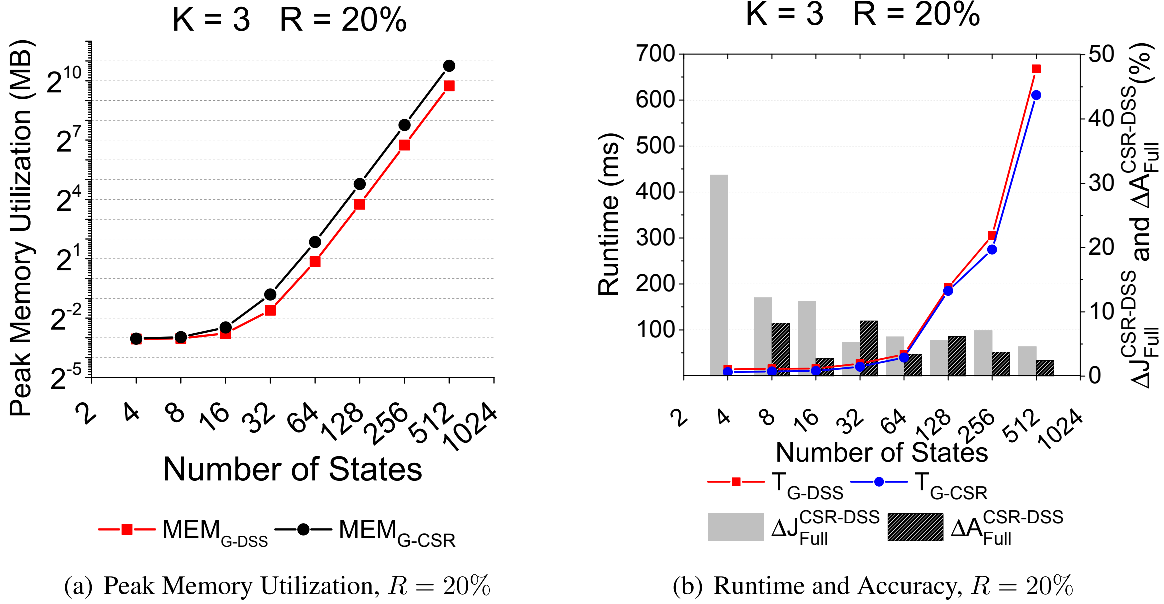
Symbol	Meaning
R	the percentage of elements the DSS stores (i.e. X/N)
K	Number of horizons in the OBR framework (i.e. number of the tree levels in Fig. 4.1)
J_{Full}^{CSR}	The difference of J values between using CSR and using full matrix
A_{Full}^{CSR}	The difference of action decisions between using CSR and using full matrix
T_{G-DSS}	Runtime of GPU-based OBR using DSS
T_{G-CSR}	Runtime of GPU-based OBR using CSR
T_{G-Full}	Runtime of GPU-based OBR using full matrix
T_{S-Full}	Runtime of sequential OBR using full matrix
MEM_{G-DSS}	Peak memory utilization of GPU-based OBR using DSS
MEM_{G-CSR}	Peak memory utilization of GPU-based OBR using CSR
MEM_{G-Full}	Peak memory utilization of GPU-based OBR using full matrix

In Fig. 5.6(b), $\Delta J_{Full}^{CSR-DSS}$ and $\Delta A_{Full}^{CSR-DSS}$ are defined as follows:

$$\Delta J_{Full}^{CSR-DSS} = J_{Full}^{CSR} - J_{Full}^{DSS} \tag{5.6}$$

$$\Delta A_{Full}^{CSR-DSS} = A_{Full}^{CSR} - A_{Full}^{DSS} \tag{5.7}$$

When $\Delta J_{Full}^{CSR-DSS}$ (or $\Delta A_{Full}^{CSR-DSS}$) is larger than zero, it means that the J value (or the final action decision result) computed by G-DSS is closer to that computed by G-Full, compared to G-CSR. Fig. 5.6(b) shows that G-DSS has a better accuracy on J value and final action decision compared to G-CSR for all the cases. DSS uses 12-bit fixed-point fractions to represent $\overline{\mathcal{P}}_{ij}$ (as opposed to 32-bit floats for CSR). This might seem to sacrifice computation accuracy. However, unlike CSR, DSS doesn't abandon the trivial values but instead uses their average value to approximately represent them. Hence its results are closer to the results of G-Full, for both the J value and final action decision. In Fig. 5.6(b), the runtime of G-DSS is found to be 25.6% higher than G-CSR on average. This is mainly because DSS needs to update the average value of all the trivial entries during Bayesian update, which requires more computations than CSR. Based on these results, we note that DSS is superior to CSR in terms of memory utilization and accuracy with a



(a) Peak Memory Utilization, $R = 20\%$ (b) Runtime and Accuracy, $R = 20\%$

Figure 5.6: Performance of G-DSS and G-CSR

modest performance penalty.

In Fig. 5.6, we present the runtime, accuracy and the peak memory utilization of G-DSS and compare these quantities with G-Full, while varying the R value. In some experiments, certain data points of G-Full are missing, because the G-Full algorithm ran out of memory and didn't terminate. In Fig. 5.6, we focus on how the performance changes with respect to the number of states (the size of the $\overline{\mathcal{P}}$ matrix). Note that the runtime, peak memory utilization and the number of states are represented on a logarithmic scale, with base value of 10, 2 and 2 respectively. From these figures, we notice that the G-DSS is faster than G-Full. In particular, for $R = 10\%$, the G-DSS is faster by $5.7\times$ on average, and this improvement reduces slightly as R increases. As discussed in Section 5.2.3, the reason why G-DSS is faster is that DSS only stores $(X + K)$ elements for each row instead of G-Full which stores N elements. Hence, in G-DSS, the GPU only needs to launch $(X + K)$ threads for each node in the computation tree shown in Fig. 4.1 (versus N threads for G-Full). Generally speaking, launching more threads on the GPU will require more hardware resources. Since G-DSS requires less threads than G-Full for each node in the tree, it can process

more nodes simultaneously. As R increases from 10% to 40%, the average runtime of G-DSS increases by 47.2%. In general, J^{DSS}_{Full} and A^{DSS}_{Full} improve as the number of states increases and R increases. Note that A^{DSS}_{Full} is not monotonic as the number of states increases. This is mainly because the final decision A_{DSS} depends on the accumulated rewards over multiple actions as described in Eq. (2.6). The DSS scheme introduces an error to the rewards under different actions in varying amounts, depending on the number and sparsity of the entries in the $\overline{\mathcal{P}}$ matrix. Hence, there is some noise in the trend of A^{DSS}_{Full} as the number of the states increases. The peak memory utilization reduces by $6.3\times$ on average when $R = 10\%$. This improvement reduces as R increases as expected. For $R = 30\%$, the peak memory utilization is $2.4\times$ better for G-DSS on average.

Considering both runtime and peak memory utilization, a practical choice of R could be 20% or 30%. When $R = 20\%$, the average speedup of DSS over full matrix from 4 states to 256 states is about $4.1\times$. In the largest case (with 512 states), G-Full can't handle such case because it runs out of memory. Hence we use the J value and action decision computed by S-Full as J_{Full} and A_{Full} respectively to calculate the J^{DSS}_{Full} and A^{DSS}_{Full} following Eq. (5.5). When the number of states is 512, $J^{DSS}_{Full} = 10.3\%$ and $A^{DSS}_{Full} = 2.4\%$.

Fig. 5.7 shows the runtime, accuracy and the peak memory utilization of G-DSS compared with G-Full as K is varied. When K increases, the J^{DSS}_{Full} and A^{DSS}_{Full} also increase. This is mainly because the error in the J value at level $(k-1)$ propagates to level k according to Eq. 2.6. Also, even though $T_{G-DSS} < T_{G-Full}$ for all the cases, the runtime of G-DSS increases at a faster rate than G-Full as K increases in Fig. 5.7(i). A reason for this is that DSS needs to do more computation for looking up a single $\overline{\mathcal{P}}_{ij}$. Although look-ups are parallelized, the number of extra look-up computations depends heavily on K. Hence when K increases, the total amount of extra look-up work also increases, which means that the GPU needs more time to process these look-ups with its fixed hardware resources. In Fig. 5.7(j), the peak memory utilization of G-DSS is $1.1\times$ lower than G-Full when $K = 2$. For $K = 4$, G-DSS uses $3.1\times$ less memory than G-Full. When $K = 5$, G-Full runs out of memory.

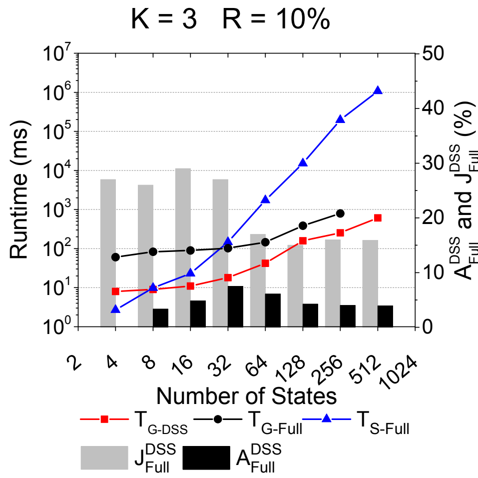
We also present the trade-off between K and the number of states for G-DSS with a fixed

memory size of 2GB and $R = 20\%$, as shown in Fig. 5.8. Fig. 5.8 shows the maximal number of states that G-DSS and G-Full can handle when K varies from 2 to 9. When K is fixed, the number of states that G-DSS can handle is more than or equal to that of G-Full, since DSS uses less memory than full matrix. On average, G-DSS can handle $1.63\times$ as many states as G-Full. From Fig. 5.8, it's indicated that DSS can handle larger problem size than full matrix.

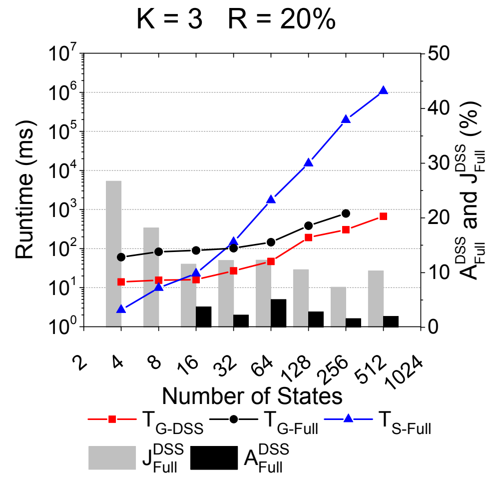
Fig. 5.9 shows the split between the lookup and computation time of G-DSS. Note that both horizontal and vertical axes are in logarithmic scale. We also include the total runtime of the full matrix approach for reference. Since it's quite difficult to separate the time consumed by accessing global memory in GPU, we aren't able to separate the runtime of look-up from the total runtime for GPU-based OBR with full matrix. The purpose of look-up is to find $\overline{\mathcal{P}}_{ij}$ in DSS data structure by following Algorithm 4. Since DSS requires more computation for lookup compared with full matrix, it's important to quantify the trend in the runtime for lookup operations as the number of states increases. Fig. 5.9 shows that compared with G-Full, the runtime of G-DSS lookup isn't the dominant component of the total G-DSS runtime for 32 or fewer states. This is mainly because the lookup time of DSS benefits from the parallel processing of GPU. As the number of states increases beyond 32, the lookup becomes a noticeable fraction of the total G-DSS runtime.

5.4 Conclusion

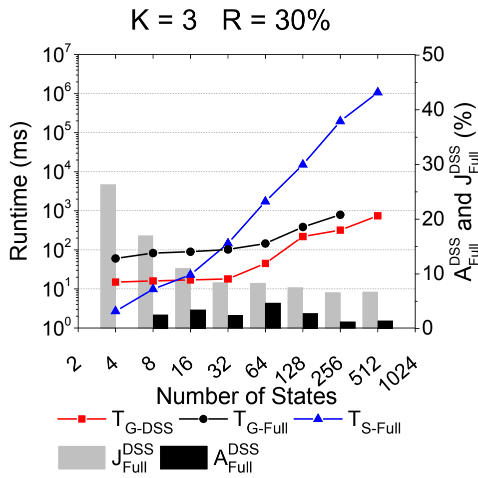
In this chapter, we propose a Duplex Sparse Storage (DSS) format to alleviate the memory growth of the GPU-based OBR algorithm in the context of the GRN control problem. By combining the column position index and the entry value in the same word and using an average value to represent all trivial elements (instead of abandoning them as in CSR), our method significantly reduces the memory utilization compared with both full matrix and CSR. DSS also improves the result accuracy of GPU-based OBR compared with CSR. Our DSS is faster than the full matrix by $4.1\times$ in the context of GPU-based OBR on the GRN control problem when $R = 20\%$. With a fixed memory limitation, our DSS can handle a larger state space and deeper horizons for the GPU-based OBR algorithm.



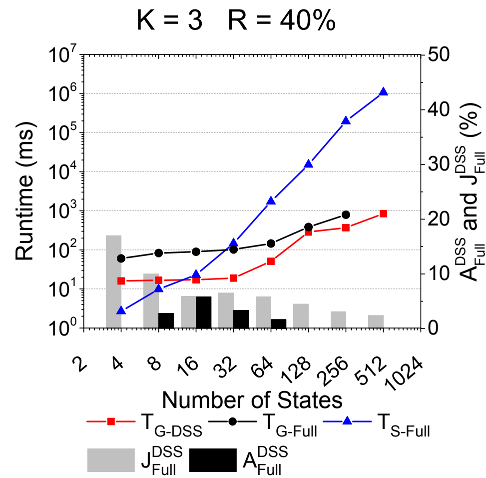
(a) Runtime vs. Accuracy, $R = 10\%$



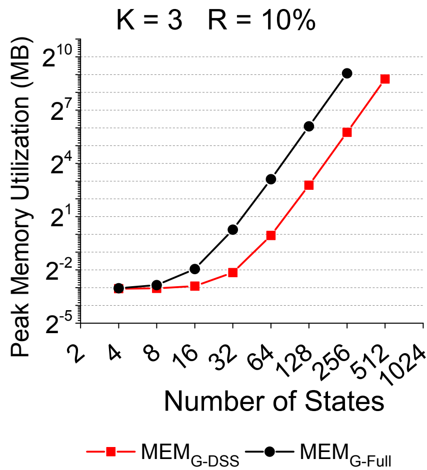
(b) Runtime vs. Accuracy, $R = 20\%$



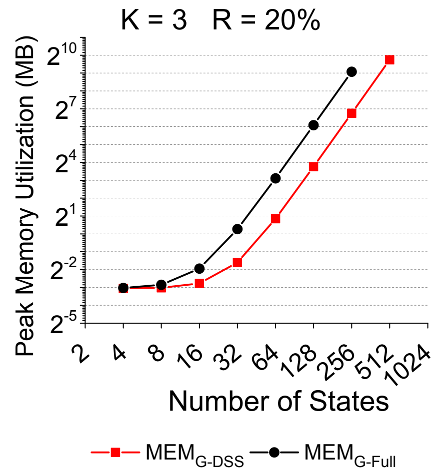
(c) Runtime vs. Accuracy, $R = 30\%$



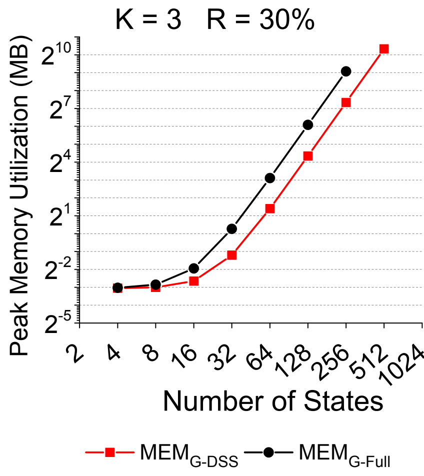
(d) Runtime vs. Accuracy, $R = 40\%$



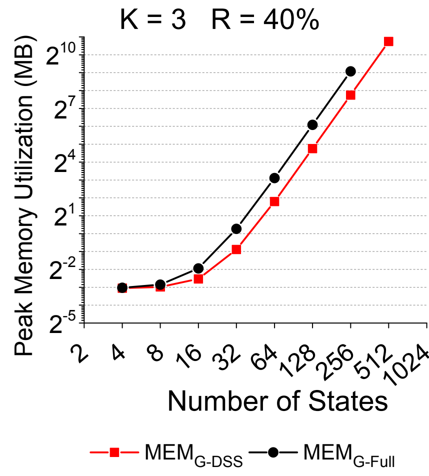
(e) Peak Memory Utilization, $R = 10\%$



(f) Peak Memory Utilization, $R = 20\%$

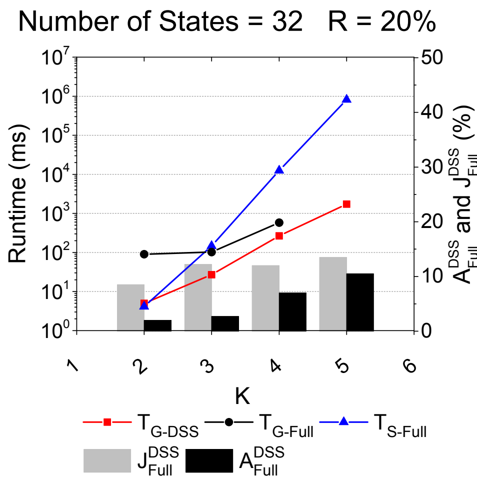


(g) Peak Memory Utilization, $R = 30\%$

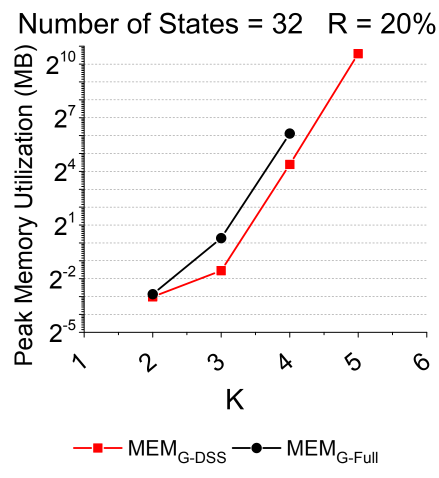


(h) Peak Memory Utilization, $R = 40\%$

Figure 5.6: Performance of G-DSS and G-Full when $K = 3$



(i) Runtime vs. Accuracy



(j) Peak Memory Utilization

Figure 5.7: Performance of G-DSS when $R = 20\%$ and number of states $= 32$

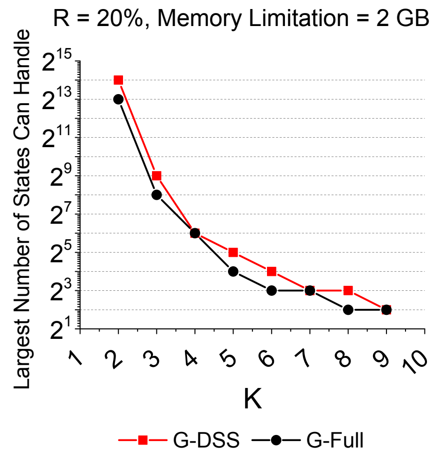


Figure 5.8: Trade-off between K and number of states with a 2GB memory limitation when $R = 20\%$

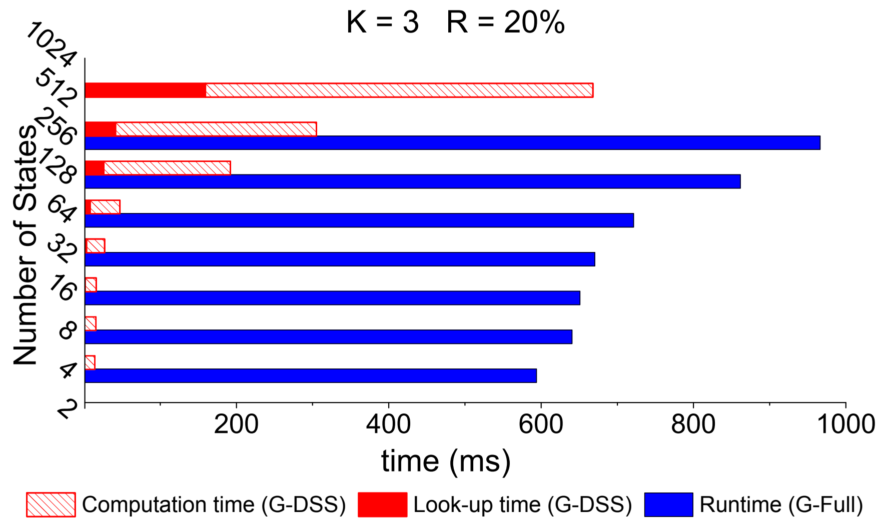


Figure 5.9: Look-up and computation time of G-DSS

6. PART 2: A NOVEL BINARY DECISION DIAGRAM (BDD) BASED SAMPLING FOR BMDP

In this chapter we will describe a novel BDD-based approach to store data in the BMDP computation. Due to the inherent "curse of dimensionality", the practical applicability of the BMDP is limited by data storage constraints. This is particularly true when a single instruction multiple data (SIMD) platforms such as Graphic Processing Units (GPUs) are used to accelerate the computation. To overcome this obstacle, in this chapter we propose a highly memory-efficient Binary Decision Diagram (BDD) based sampling representation for the BMDP model, and develop a corresponding BMDP solver on a heterogeneous CPU-GPU platform. Our simulation results demonstrate that our approach achieves significantly better memory scalability, reducing the memory utilization of the overall BMDP algorithm by $13.2\times$ with an "error" of 3.4% compared with the conventional floating point representation for a BMDP problem with $262,144$ transitions. For the same BMDP problem, our BDD-based sampling approach reduces the memory required to store the transition probability matrix (TPM) by three orders of magnitude compared to a conventional floating point representation. Our scheme can handle BMDP problems with up to 4.19×10^6 state transitions on a desktop computer, which no other technique has been able to achieve.

6.1 Introduction and Background

As discussed in Chapter 4.3.1, the GPU-BMDP computing generally entails exponential memory complexity. To mitigate the memory bottleneck of both classical and Bayesian MDP computations, we propose a compact model that represents a transition probabilities by sampling a set of Boolean functions instead of using floating point numbers to represent the probability. If there are N states, a probability matrix of this model uses a constant number of Boolean functions, each of which has $O(\log N)$ inputs, as opposed to $O(N^2)$ floating numbers in the conventional

©2019 IEEE. Reprinted, with permission, from He Zhou, Sunil P. Khatri, Jiang Hu, Frank Liu, A Memory-Efficient Markov Decision Process Computation Framework Using BDD-based Sampling Representation, 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019

representation. Instead of storing the expected TPM explicitly as floating point numbers, we store it implicitly, and generate an entry by sampling a fixed number (M) of Boolean functions. If an entry in the expected TPM is p_{ij}, then exactly $q = \lfloor M \cdot p_{ij} \rfloor$ of the M Boolean functions evaluate to "true" when the input to these functions is the concatenated binary code of i and j. Therefore, by evaluating the M Boolean functions, any entry in the expected TPM can be generated. The M Boolean functions can be represented by a Reduced Ordered Binary Decision Diagram (ROBDD, henceforth abbreviated as BDD). A BDD is a compact Directed Acyclic Graph (DAG) representation for Boolean functions [60, 61]. Since the BDD eliminates isomorphic subgraphs, it is frequently more compact compared with other representations for a Boolean function [62]. We have also developed an implicit technique for performing Bayesian update on the BDD sampling-based probability representation.

Sample-based approaches in [13, 37, 36] can also reduce the memory utilization of $\overline{\mathcal{P}}$ matrices on the GPU platform if executed in parallel. One key idea of these sample-based approaches is that instead of expanding the entire decision tree in Fig. 4.1, only parts of the tree are expanded by using different sampling strategies. Therefore, the overall memory footprint of $\overline{\mathcal{P}}$ matrices can be reduced by decreasing the number of $\overline{\mathcal{P}}$ matrices instead of decreasing the size of one single $\overline{\mathcal{P}}$ matrix as our BDD-based approach does. However, the sample-based approaches don't fundamentally change the memory scalability of a single $\overline{\mathcal{P}}$ matrix. Also, if executed on GPU, these approaches require a runtime overhead to determine which part of the decision tree should be expanded. In summary, [13, 37, 36] use sampling to reduce the number of $\overline{\mathcal{P}}$ matrices, while our approach uses sampling to reduce the storage required for a single $\overline{\mathcal{P}}$ matrix.

There are also some previous work using Decision Diagrams (DDs) to reduce the memory footprint for MDP problems, such as FODD [63], SPUDD [41] and APRICODD [40]. However, one fundamental difference of these methods from ours is that these methods are not based on sampling. Instead, these methods utilize the Decision Diagram data structure to approximate the model of MDP problems. Compared with our method, FODD doesn't represent the $\overline{\mathcal{P}}$ matrix and there is no arithmetic function in the decision diagrams that FODD uses. SPUDD and APRICODD

both use Algebraic Decision Diagrams (ADDs, which are DDs that represent numerical values) to directly represent the reward function, value function, policy and the $\overline{\mathcal{P}}$ in the MDP problems, which in general will not have too much sharing among the internal edges of the diagrams. Both SPUDD and APRICODD require a Conditional Probability Table (CPT) for the ADD variables to represent the $\overline{\mathcal{P}}$ matrix, while our sampling-based scheme does not require such information. FODD, SPUDD and ARICODD focus on solving classical MDP, in which the model is static. However, in our sampling-based approach, we support classical and Bayesian MDP, in which the $\overline{\mathcal{P}}$ matrix can be static or dynamic, since we present an efficient algorithm to perform the Bayesian update.

The proposed model greatly improves the memory efficiency of MDP computations, but incurs an increase in computing errors and runtime. The computing errors can be easily controlled by using a sufficiently large number of BDDs for the representation. The runtime increase is not critical for classical MDP, since its policy computation is performed offline. For Bayesian MDP, the runtime overhead can be overcome by utilizing GPU computing. Simulation results show that our approach can reduce memory utilization by $13\times$ and $137\times$ for Bayesian MDP with 512 states and classical MDP with $32K$ states, respectively. In both scenarios, the computing errors are within a few percent. Overall, our approach allows a single CPU/GPU to handle MDP problems several times larger than conventional methods.

6.1.1 Preliminaries - Binary Decision Diagrams

The Reduced Ordered Binary Decision Diagram (ROBDD) [60, 64] is a canonical representation for a Boolean function. An ROBDD (often also referred to as a BDD) is essentially a Directed Acyclic Graph (DAG) with one root vertex and two terminal vertices 0 and 1. Each vertex v is associated with a variable x_i, and has two children vertices representing f_{x_i} and $f_{\bar{x}_i}$, where f is the function corresponding to v. Semantically, the vertex function is expressed by Shannon's Expansion Theorem: $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$. Here, $f_{x_i} = f(x_1, x_2, \cdots, x_i = 1, \cdots, x_{n-1}, x_n)$ and $f_{\bar{x}_i} = f(x_1, x_2, \cdots, x_i = 0, \cdots, x_{n-1}, x_n)$ are referred to as the cofactors of f with respect to x_i and $\bar{x}_i$ respectively. The functions f_{x_i} and $f_{\bar{x}_i}$ are independent of the variable x_i. In each path from

the root to any terminal vertex of the ROBDD, the same variable ordering is encountered. Hence the ROBDD is referred to as *ordered*. Also, isomorphic subgraphs are merged. As a result, the ROBDD is referred to as *reduced*.

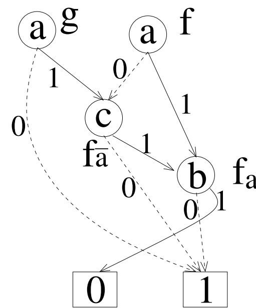


Figure 6.1: An example of using BDD to represent 2 Boolean functions f and g. The left (right) child of each vertex is its 0-cofactor (1-cofactor) with respect to vertex variable.

Fig. 6.1 illustrates the ROBDD of a function $f(a,b,c) = \bar{a}\bar{c} + \bar{a}c\bar{b} + a\bar{b}$ and $g(a,b,c) = \bar{a} + \bar{b} + \bar{c}$. The input variables of the Boolean function are also the variables of the corresponding BDD. The Boolean function is obtained by tracing all paths in the BDD to the "1" vertex. The variable ordering used in the BDD of Fig. 6.1 is $a \to c \to b$. Every vertex in a BDD is a logic function given by the Shannon Expansion Theorem. For example, the root vertex of the BDD in Fig. 6.1 can be expressed as:

$$f(a,b,c) \equiv f = \bar{a}f_{\bar{a}} + af_a \tag{6.1}$$

where $f_a = f(a=1,b,c) = \bar{b}$, and $f_{\bar{a}} = f(a=0,b,c) = \bar{c} + c\bar{b}$. Hence f can be written as $f = \bar{a}(\bar{c} + c\bar{b}) + a(\bar{b})$.

The process of getting f_a or $f_{\bar{a}}$ is called *cofactor operation* on a or $\bar{a}$ respectively. Note that the cofactor operation can be applied not only upon a single literal (a Boolean input variable or the complement of a Boolean input variable), but also upon a $minterm$ (which includes literals of all

the Boolean input variables).

As mentioned previously, BDDs are a canonical representation, which means that the isomorphism of two BDDs implies that their corresponding Boolean functions are equivalent.

Note that sharing of vertices can happen within one BDD or across several different BDDs. The BDD vertex labeled f_a in Fig. 6.1 is shared by two paths in the BDD vertex of f, hence resulting in a compact representation. Also, the vertex labeled $f_{\bar{a}}$ is shared between the BDDs of f and g, illustrating compactness due to vertex sharing *between* different functions. In software implementations of an ROBDD package [65], all the vertices of the BDDs are stored in a special structure called the *unique table*, which guarantees that every vertex is represented exactly once in the BDD package. The unique table and auxiliary data structures are included in a *manager*, which stores all the BDDs that have been created in the BDD package. The *manager* is also responsible for garbage collection, which periodically deallocates unused BDD vertices, which in turn enhances the storage efficiency.

In our approach, we use the sampled values of M Boolean functions (each represented as BDDs) to represent the individual entries of the TPM. This results in a compact representation of the TPM entries, since a) BDD vertices can be shared by multiple paths in the BDD representation of any one of the M BDDs, and b) BDD vertices can be shared between different BDDs as well.

6.2 BDD-based Sampling Representation and Operations

6.2.1 Overview

In this section, we describe our Binary Decision Diagram (BDD) based sampling representation for the $\overline{\mathcal{P}}$ matrix in an MDP. We first describe the BDD-based sampling representation, along with essential algorithms in the context of the classical MDP and BMDP problems. We next show how we implement our BDD-based MDP on a heterogeneous platform consisting of a CPU and a GPU.

6.2.2 BDD-based Sampling Representation of TPM

As mentioned in Section 2.3, the memory required to store the expected TPM $\overline{\mathcal{P}}$ using floating point numbers grows quadratically in the size of the state space of the BMDP problem. We propose

to represent every entry of the $\overline{\mathcal{P}}$ matrix implicitly by sampling M Boolean functions. If the entry corresponding to $(s_i, s_j)^{th}$ in the expected TPM is p_{ij}, then exactly $q = \lfloor M \cdot p_{ij} \rfloor$ of the M Boolean functions evaluate to "true". All the M functions share the same input variable values indicating a transition from s_i to s_j. The N states of a system can be encoded by $log_2(N)$ binary variables. Then, a transition can be represented by a pair of $log_2(N)$ bit vectors, one for the current state and the other for the next state. For transition from s_i to s_j, if s_i and s_i are encoded by $X_i = \{x_1, x_2, \ldots, x_{log_2(N)}\}$ and $Y_j = \{y_1, y_2, \ldots, y_{log_2(N)}\}$, respectively, then the transition is represented by $\{x_1, x_2, \ldots, x_{log_2(N)}, y_1, y_2, \ldots, y_{log_2(N)}\}$. One can see that each Boolean function depends on $O(\log(N))$ input variables. Note that M is a constant parameter that is chosen based on the desired numerical precision. Since $p_{ij} = q/M$, the precision of p_{ij} by sampling M Boolean functions is limited to $1/M$. In practice, M is usually much smaller than N and thus its impact on storage complexity is not critical.

In theory, like any other representation, the BDD size can be exponential with respect to the number of variables. However, in practice its size is usually far less than exponential [61] for many functions of interest. If the M Boolean functions are represented by a set of M BDDs $BDD_M = \{bdd_1, bdd_2, ..., bdd_M\}$, the theoretical storage complexity for a TPM is at most $O(N)$, which is much better than the $O(N^2)$ complexity of the floating point number representation. Moreover, the multiple BDDs used in a BMDP can share many nodes and edges. To illustrate this, consider Fig. 6.1, in which the BDDs f and g share the subgraph rooted at vertex $f_{\bar{a}}$, resulting in a compact representation.

In order to use BDD_M to represent the $\overline{\mathcal{P}}$ matrix for BMDP, algorithms for the following three operations should be defined:

- generating BDD_M from the $\overline{\mathcal{P}}$ matrix
- looking up p_{ij} from BDD_M
- performing the Bayesian update on BDD_M when a state transition from s_i to s_j is observed

The first two apply both to classical MDP as well as Bayesian MDP, while the last operation applies

to Bayesian MDP alone. We will discuss these algorithms in the next three sections.

6.2.2.1 Generating BDD_M

The procedure of generating BDD_M based on $\overline{\mathcal{P}}$ is shown in Algorithm 6. Since the sum of each row in matrix $\overline{\mathcal{P}}$ is 1, the entries in the row are represented by disjoint subsets of BDD_M. Step 5 in Algorithm 6 specifies the range of the subset in BDD_M that represents an entry $\overline{\mathcal{P}}_{ij}$. In order to "store" the value of $\overline{\mathcal{P}}_{ij}$, we include the minterm s_{ij}, which is the concatenation of X_i and Y_j, from bdd_{start} to bdd_{end}. Since each entry in matrix $\overline{\mathcal{P}}$ has a unique combination of X_i and Y_j, the corresponding minterm s_{ij} for each entry is also unique, and hence the ambiguity of different $\overline{\mathcal{P}}_{ij}$ entries is avoided.

Algorithm 6 Generate BDD_M

Require: TPM: $\overline{\mathcal{P}}$; the number of states: N; the total number of bdds: M
Ensure: $BDD_M = \{bdd_1, bdd_2, \ldots, bdd_M\}$

```
1:  Set all the bdds to logic 0
2:  start = 1
3:  for i = 1 to N do
4:      for j = 1 to N do
5:          end=start+P̄_ij × M-1
6:          X_i = s_i expressed in binary format
7:          Y_j = s_j expressed in binary format
8:          s_ij = X_i · Y_j /*'·' indicates concatenation*/
9:          for k = start to end do
10:             bdd_k = bdd_k + s_ij /*s_ij is a minterm*/
11:         end for
12:         start = end+1
13:     end for
14:     start = 1
15: end for
16: return BDD_M = {bdd_1, bdd_2, ..., bdd_M}
```

For example, suppose there are 4 states in the BMDP and $M = 10$, and the first row of $\overline{\mathcal{P}}$ is: $\overline{\mathcal{P}}_{00} = 0.1$, $\overline{\mathcal{P}}_{01} = 0.2$, $\overline{\mathcal{P}}_{02} = 0.3$, and $\overline{\mathcal{P}}_{03} = 0.4$. After processing the first row, the functions for BDD_M are:

$bdd_1 = \overline{x_1}\overline{x_2}\overline{y_1}\overline{y_2}$,

$bdd_2 = bdd_3 = \overline{x_1}\overline{x_2}\overline{y_1}y_2$,

$bdd_4 = bdd_5 = bdd_6 = \overline{x_1}\overline{x_2}y_1\overline{y_2}$

$bdd_7 = bdd_8 = bdd_9 = bdd_{10} = \overline{x_1}\overline{x_2}y_1y_2$.

If the $\overline{\mathcal{P}}$ matrix of N states is stored by a floating point matrix, the storage complexity is $\mathcal{O}(N^2)$, and there is no information sharing among different entries. Theoretically, if a BDD has $n = 2log_2(N)$ variables, then size of the BDD in the worst case will be $\mathcal{O}(2^n) = \mathcal{O}(N^2)$. However, as discussed in Section 6.1.1, the size of BDDs is reduced because it combines isomorphic nodes and eliminates redundant nodes. Therefore, many BDD nodes are shared across the minterms in BDD_M, which indicates information sharing across different entries in the $\overline{\mathcal{P}}$ matrix. As a result, the memory utilization of BDD_M scales very well in practice, which will be shown in our simulation results in Section 6.3.

6.2.2.2 *Looking up $\overline{\mathcal{P}}_{ij}$ from BDD_M*

Once BDD_M is generated to represent the expected TPM $\overline{\mathcal{P}}$, Algorithm 7 shows how to look up $\overline{\mathcal{P}}_{ij}$ from BDD_M. Similar to Algorithm 6, step 4 creates a minterm indicating the transition from s_i to s_j. In step 6, we sample each bdd_k with input s_{ij} using the cofactor operation. The result of the cofactor operation is either 1 or 0, depending on whether bdd_k includes the minterm s_{ij} or not. If the $(s_i, s_j)^{th}$ entry of $\overline{\mathcal{P}}$ is $\overline{\mathcal{P}}_{ij}$, then according to Algorithm 6, there will be $q = \lfloor \overline{\mathcal{P}}_{ij} \times M \rfloor$ bdds in BDD_M including the minterm s_{ij}. In Algorithm 7, these q bdds will yield a "1" after cofactoring with s_{ij}. After step 9 in Algorithm 7, $\overline{\mathcal{P}}_{ij} = q/M$, which is consistent with the original $\overline{\mathcal{P}}_{ij}$.

If Algorithm 7 is executed on a serial platform, the computation complexity of step 5 is $\mathcal{O}(M)$. In the worst case, the computation complexity of step 6 is $\mathcal{O}(log_2(N))$, because there are $2log_2(N)$ input variables in total. Since the number of lookups to compute Eq. (2.6) once is N, the overall computation complexity for one J computation of the classical MDP/BMDP in the worst case is $\mathcal{O}(MNlog_2(N))$. Considering the number of J computations (i.e. Eq. (2.6)) in the entire BDP process, the lookup operations become the runtime bottleneck of the BDD-based sampling approach

in the classical MDP/BMDP problem. We alleviate this bottleneck in Section 6.2.3 by exploiting the parallelism of the GPU to implement Algorithm 7.

Algorithm 7 Look up $\overline{\mathcal{P}}_{ij}$

Require: $BDD_M = \{bdd_1, bdd_2, \ldots, bdd_M\}$; the total number of bdds: M; "from" state: s_i; "to" state: s_j
Ensure: $\overline{\mathcal{P}}_{ij}$
1: $\overline{\mathcal{P}}_{ij} = 0$
2: $X_i = s_i$ expressed in binary format
3: $Y_j = s_j$ expressed in binary format
4: $s_{ij} = X_i \cdot Y_j$
5: **for** $k = 1$ to M **do**
6: $result_k$ = Cofactor(bdd_k, s_{ij})
7: $\overline{\mathcal{P}}_{ij} = \overline{\mathcal{P}}_{ij} + result_k$
8: **end for**
9: $\overline{\mathcal{P}}_{ij} = \overline{\mathcal{P}}_{ij}/M$
10: return P_{ij}

6.2.2.3 *Bayesian Update on BDD_M*

To use BDD_M in the BMDP problem, BDD_M should be able to support the Bayesian update. Consider timestep t, when the system state is s_i and the expected TPM is $\overline{\mathcal{P}}$. At time $t+1$, the state transitions to s_j and the expected TPM needs to be updated to $\overline{\mathcal{P}}'$ according to Bayes' rule. Among the t transitions from time step 1 to $t+1$, if C of them are from s_i to another state, the ith row of the $\overline{\mathcal{P}}'$ matrix can be computed as

$$\begin{cases} \overline{\mathcal{P}}'_{ik} = \frac{C \times \overline{\mathcal{P}}_{ik}}{C+1}, & k \neq j \\ \\ \overline{\mathcal{P}}'_{ij} = \frac{C \times \overline{\mathcal{P}}_{ij} + 1}{C+1}, & k = j \end{cases} \tag{6.2}$$

Eq. (6.2) is derived from Eq. (2.4) and Bayes' rule. The other rows of matrix $\overline{\mathcal{P}}'$ remain unchanged. Algorithm 8 describes how to carry out the computation of Eq. (6.2) on BDD_M.

Algorithm 8 Bayesian Update

Require: $BDD_M = \{bdd_1, bdd_2, \ldots, bdd_M\}$; the total number of bdds: M; state transition from $z_t = s_i$ to $z_{t+1} = s_j$; the number of state transitions from s_i from time step 1 to $t+1$: C

Ensure: updated BDD_M

1: $P_{ij} = 0$

2: $X_i = s_i$ expressed in binary format

3: $Y_j = s_j$ expressed in binary format

4: s_{ij}=$X_i \cdot Y_j$

5: **for** m=1 to M **do**

6: $result_m$ = Cofactor(bdd_m, s_{ij})

7: **if** $result_m == 0$ **then**

8: generate a random number $g \in [0, 1]$

9: **if** $g < \frac{1}{C+1}$ **then**

10: $bdd_k = (bdd_k \wedge \overline{X_i}) \vee s_{ij}$

11: **end if**

12: **end if**

13: **end for**

14: return $BDD_M = \{bdd_1, bdd_2, \ldots, bdd_M\}$

According to Eq. (6.2), for each state s_k where $k \neq j$, we reduce p_{ik} by $1/(C+1)$ of its original value, and add what we subtracted from p_{ik} into p_{ij}. Hence, whenever the "if" statement condition in step 7 of Algorithm 8 is true, it means that bdd_m is contributing to some p_{ik} $(k \neq j)$. In steps 8 and 9, the minterm s_{ik} is eliminated from bdd_m with probability $1/(C+1)$, hence we can achieve the goal of reducing p_{ik} by $1/(C+1)$ of its original value in one for loop without a need to explicitly know the value of p_{ik}. However, it is not known which p_{ik} $(k \neq j)$ that bdd_m contributes to, but we do know the value of i. In step 10, we AND bdd_m with $\overline{X_i}$, where $\overline{X_i}$ is the complement of the binary representation of s_i. Therefore, it is guaranteed that no matter what the value of k is, the minterm s_{ik} is removed in one step. Since we need to add what we subtracted from $\overline{\mathcal{P}}_{ik}$ into $\overline{\mathcal{P}}_{ij}$, in step 10, after the minterm s_{ik} is removed, we need to perform OR operation on the bdd with the minterm s_{ij}.

6.2.3 G-BDD: A BMDP Solver Using BDD-based Sampling Representation on CPU-GPU

We have developed a BMDP solver on a heterogeneous CPU-GPU platform by using the BDD-based sampling representation for the $\overline{\mathcal{P}}$ matrix. As discussed in Section 6.2.2.2, the runtime

bottleneck of using the BDD-based sampling representation for the BMDP problem arises from looking up $\overline{\mathcal{P}}_{ij}$ from BDD_M, due to the number of the lookup operations and the computation complexity of Algorithm 7. Since the cofactor operations of all the M BDDs in BDD_M are independent, step 5 in Algorithm 7 can be performed in parallel for all the M BDDs.

There is yet another avenue of parallelism in Eq. (2.6) which we exploit. The independence of the $J(s_i, \overline{\mathcal{P}})$ values of different state nodes at the same level in the computational tree of the BDP approach (as shown in Fig. 6.3) can be exploited as well. Since the GPU works in a SIMD fashion, before performing Eq. (2.6), all the $\overline{\mathcal{P}}$ matrices of different state nodes at the same level need to be pre-computed based on the state transition path from the root of the computation tree and the Bayes' rule. In other words, different BDD_Ms corresponding to different $\overline{\mathcal{P}}$ matrices need to be ready on the GPU before performing Eq. (2.6). As mentioned in Section 6.1.1, all the BDD nodes are stored in the $manager$. Different BDD nodes can be distinguished by different pointers. We denote all the BDD nodes data at level k in the computation tree as $DdChunk_k$. The data in $DdChunk_k$ includes multiple BDD_Ms representing all the posterior $\overline{\mathcal{P}}$ matrices after Bayesian update following Algorithm 8. Fig. 6.2 shows the top-level framework on a heterogeneous computing platform of the BMDP solver. In Fig. 6.2, the CPU is responsible for generating BDD_M from the prior $\overline{\mathcal{P}}$ at the root of the computation tree, and generating $DdChunk_k$ to represent all the posterior $\overline{\mathcal{P}}$ matrices at each tree level. Then, $DdChunk_k$ is transferred from CPU to GPU.

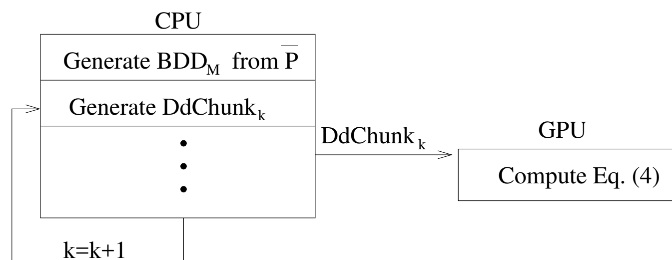


Figure 6.2: The BMDP Framework on Heterogeneous Computing Platform

Eq. (2.6) is computed on the GPU. An example of the GPU computation is shown in Fig. 6.3,

where $N = 4$ and $K = 3$. The J value computations of all state nodes at the same level are performed by master threads in parallel. While each master thread performs the J computation, it needs the values of $\overline{\mathcal{P}}_{ij}$ with a fixed i. The lookup operation is performed by the slave threads following Algorithm 7. To look up one $\overline{\mathcal{P}}_{ij}$, M slave threads are required per master thread, to cofactor M BDDs in parallel. To avoid memory explosion, each $\overline{\mathcal{P}}_{ij}$ for a fixed i is immediately consumed (and not stored) after being computed by Algorithm 7. Therefore, the lookup operations of each $\overline{\mathcal{P}}_{ij}$ for a fixed i are executed serially for the same parent node at level $k+1$. For a fixed j, the lookup operations of each $\overline{\mathcal{P}}_{ij}$ are executed in parallel. The master and slave threads need to be synchronized to avoid data race conditions.

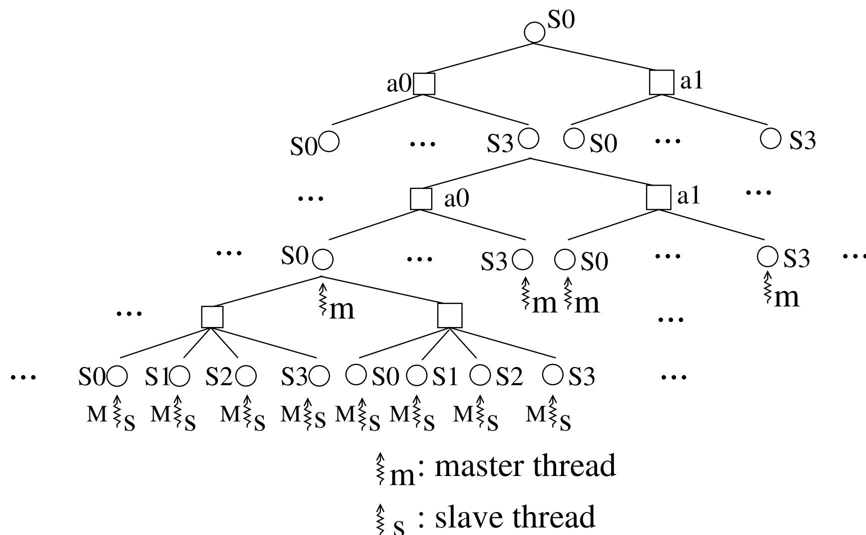


Figure 6.3: An Example of the GPU Computation

6.3 Experiment Results

The BMDP solver using our BDD-based sampling representation (denoted as G-BDD) is implemented using Nvidia CUDA Toolkit 8.0. The BMDP solver using floating point on GPU is denoted as G-FP. The classical MDP solver using BDD on CPU is denoted as CMDP-BDD. We implemented the BDDs using the CUDD package [65], and migrated the cofactor operation of

CUDD from CPU to GPU. We ran our simulations on 1000 synthetic prior $\overline{\mathcal{P}}$ matrices. The hardware platforms are:

- CPU: Intel Xeon CPU E5-2680 v4 @ 2.40GHz, 128GB DDR4 Memory
- GPU: NVIDIA K40, 745MHz Clock Rate, 12 GB GDDR5 Memory, 288 GB/sec Memory Bandwidth

We compare the peak memory utilization, the runtime as well as the result quality to evaluate the proposed G-BDD engine. In our experiment setup, K is the horizon depth (the number of tree levels explored) in the BMDP, and M is the number of BDDs used to represent one $\overline{\mathcal{P}}$ matrix. We assume that there are only 2 actions in the action space (i.e. $|A| = 2$).

We compare the BDD representation and the floating point (FP) representation in both classical MDP (CMDP) and BMDP framework. The floating point representation stores the $\overline{\mathcal{P}}$ matrix using N^2 floating point numbers, where N is the size of the state space. We first show the overall memory performance of CMDP-BDD, shown in Fig. 6.4. We employ value iteration method to solve classical MDP. Compared to floating point representation, the BDD approach achieves a $137\times$ memory reduction for 32768 states.

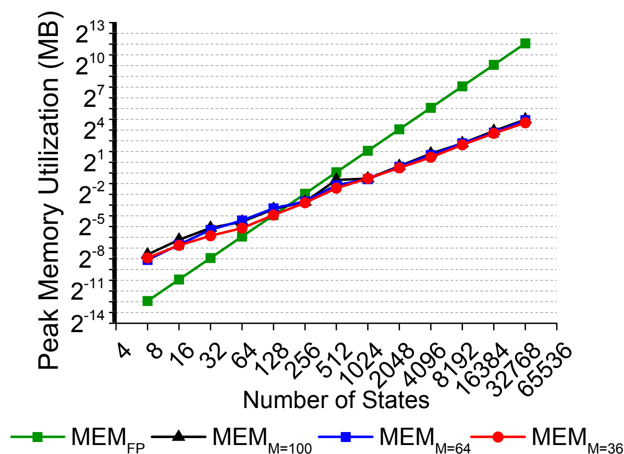


Figure 6.4: Memory Utilization in CMDP-BDD

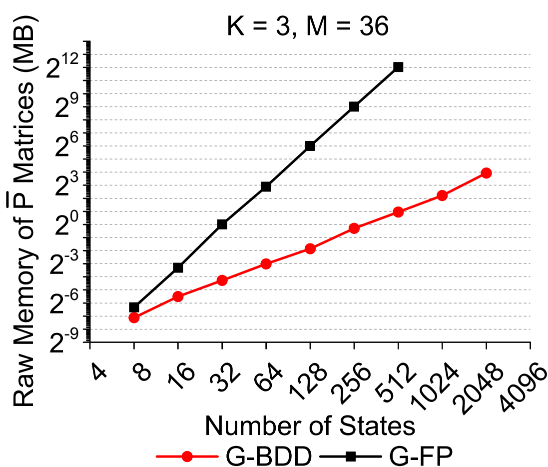
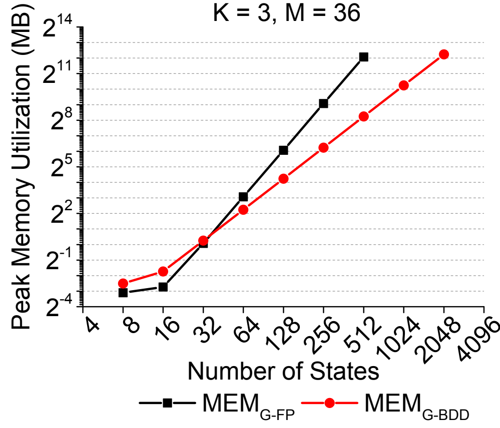


Figure 6.5: Raw Memory of $\overline{\mathcal{P}}$ Matrices in BMDP

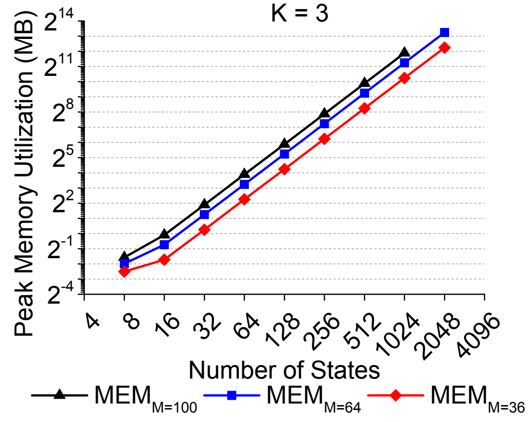
We then evaluate the memory usage of G-BDD and G-FP in the BMDP framework. If the number of states is N, then there are $(2^N \times |A|)^{(K-1)} = 2^{(N+1)(K-1)}$ nodes at level $K-1$ in the computation tree. Each node stores one row in one of the $\overline{\mathcal{P}}$ matrices. In Fig. 6.5, we present the raw memory utilization of storing only different $\overline{\mathcal{P}}$ matrices. G-FP ran out of memory when the number of states reaches 512, while G-BDD can process up to 2048 states with $M = 36$ on one GPU. Compared to G-FP, G-BDD reduces the raw memory utilization by 460$\times$ on average, and 2225$\times$ for 512 states.

Next we compare the complete BMDP implementation by using G-BDD and G-FP. In Fig. 6.6(a), the peak memory utilizations of G-FP and G-BDD are depicted over the entire BMDP computation. The peak memory utilization of G-FP is about $3.1\times$ more than G-BDD on average from 8 states to 512 states. For $N = 512$ states (i.e. 262144 state transitions), G-BDD reduces the memory utilization by $13.2\times$ compared to G-FP. G-BDD is able to handle $N = 2048$ states (i.e. 4.19×10^6 state transitions), while G-FP runs out of memory. The memory bottleneck of G-FP is to store *a row* of $\overline{\mathcal{P}}$ matrix for each state node, which takes $\mathcal{O}(N)$ space. At each state node, the memory bottleneck of G-BDD is to store M Boolean values of cofactor results and M pointers to BDDs, which are shared among all TPMs. Thus, the storage complexity of G-BDD at each node is $\mathcal{O}(M)$. This dependence on M is confirmed in Fig. 6.6(b).

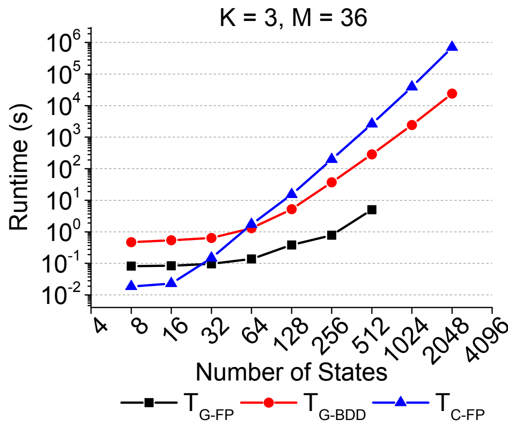
Fig. 6.6(c) shows the comparison of runtime T among G-BDD, G-FP and C-FP, which is sequential implementation of G-FP on a CPU. We observe that G-BDD is faster than C-FP by $3.9\times$ on average and $28.9\times$ for $N = 2048$ states. For G-BDD, the price paid for the large memory reduction is an average of $19\times$ runtime compared to G-FP. The reason for this is as follows: to query an entry in the $\overline{\mathcal{P}}$ matrix, G-FP only needs one data access, while G-BDD needs to perform the cofactor operations on M BDDs. Although we exploit the parallelism across M BDDs, the cofactor operation on one BDD cannot be parallelized. Also, in G-BDD, the lookup operations across different $\overline{\mathcal{P}}_{ij}$ for a fixed i are executed serially to avoid memory explosion as discussed at the end of Section 5.2. Therefore, we are not able to divide the GPU computation of G-BDD into fine-grained pieces as in G-FP, and as a consequence the GPU kernel function has a large size,



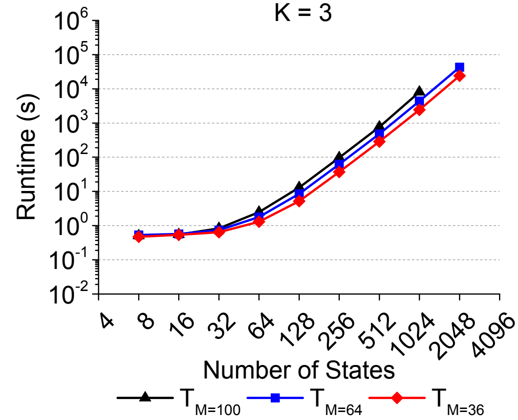
(a) MEM of Different Methods



(b) MEM of Different M



(c) T of Different Methods



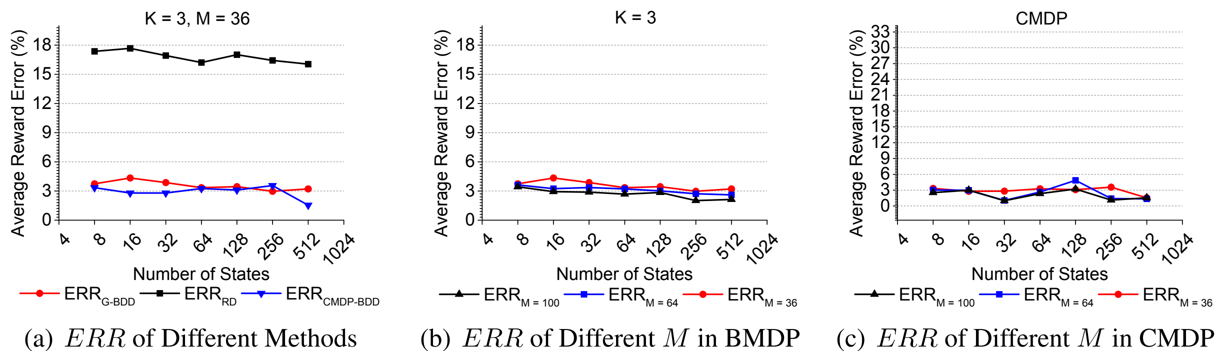
(d) T of Different M

Figure 6.6: Performance of Memory Utilization and Runtime

which hinders parallelism. Fig. 6.6(d) shows that the runtime of G-BDD is generally proportional to M, but the runtime difference for different M values is not significant.

We next evaluate the result quality of different methods. Since the BDD representation uses M BDDs to sample the $\overline{\mathcal{P}}$ matrix, the precision of $\overline{\mathcal{P}}_{ij}$ is limited by $1/M$ and the results thus are different compared to floating point representation. We evaluate the result quality by simulating the actual reward received over a finite series of state transitions. We assume that there is a ground truth TPM $\mathcal{P}$, and generate the expected TPM $\overline{\mathcal{P}}$ by adding a uniformly distributed noise δ to $\mathcal{P}$: $\overline{\mathcal{P}} = (1-\epsilon)\mathcal{P} + \epsilon\delta$, where ϵ is the noise ranging from 0 to 1 indicating the deviation of the prior $\overline{\mathcal{P}}$ from the $\mathcal{P}$. In our tests, we set $\epsilon = 0.1$. In the simulation, we first solve the BMDP and classical

MDP problem at state s_t to decide the action. Then, this action is applied to the simulated ground truth system, and the system transitions to the next state s_{t+1} with a reward. According to the transition, Bayesian update is applied to compute the posterior $\overline{\mathcal{P}}$ matrices, which will serve as the prior for the next timestep. The procedure is repeated for 20 steps, and the relative average reward error of using the BDD representation compared to using the floating point representation over the 20 steps is used to evaluate the result quality of G-BDD. Since G-FP is proved to converge to an optimal solution in [22], its result is presumed to be the correct solution here. A smaller average reward error ERR indicates the method being tested is more accurate. In the ideal case, if G-BDD and G-FP give the same action decision for every time step, then the relative average reward error should be 0. To obtain a better intuition, we also include a random decision-making strategy RD as a reference. RD makes random decisions at every timestep.



(a) ERR of Different Methods (b) ERR of Different M in BMDP (c) ERR of Different M in CMDP

Figure 6.7: Result Accuracy

In Fig. 6.7(a), we compare the result quality (error) of G-BDD, CMDP-BDD and RD. Since G-FP serves as the golden result and it can't terminate after 512 states because of the memory limitation,we only show the results from 8 to 512 states. For all the test cases, both G-BDD and CMDP-BDD yield a much smaller average reward error compared with RD. The average reward error of RD is 3.8× larger than that of G-BDD, and 6.1× larger than that of CMDP-BDD. Since RD makes random decision for each time step, it neither exploits any knowledge of the system,

nor explores the underlying system model. Therefore its decisions are very different from G-FP, and in turn result in the largest error. DSS stores the non-trivial elements in $\overline{\mathcal{P}}$ with a fixed bit resolution, and lumps all the trivial entries to an average value as approximation. Although DSS is able to capture the structure of the $\overline{\mathcal{P}}$ matrix, especially for the non-trivial entries, it can't capture the structure of the trivial entries, since these entries get smoothed by the average value. On the other hand, G-BDD is able to capture both the trivial and non-trivial values, but with a resolution bounded by M. Therefore, G-BDD yields a smaller error than DSS. Since the precision of the BDD representation is bounded by $1/M$, a larger M leads to less error, as shown in Fig. 6.7(b) and Fig. 6.7(c). A practical choice of M could be 36 since it achieves the smallest peak memory use without sacrificing the result quality much.

We also compared our work with the the well-known Compressed Sparse Row (CSR) representation and DSS representation [66]. DSS achieves better performance than CSR. However, DSS doesn't fundamentally improve the memory scalability. G-BDD is clearly superior to DSS in term of memory efficiency. For a system with 1024 states, the memory utilization of DSS was $5.7\times$ higher than that of G-BDD, while for the same system, CSR uses $11.4\times$ more memory than G-BDD. The maximal problem size that can be handled by G-BDD is twice as large as that of DSS, and four times as large as that of CSR. Moreover, the result quality of G-BDD was usually better than DSS. The price paid for these benefits of G-BDD is increased runtime (by about two orders) compared to DSS.

6.4 Conclusion

We propose a novel Binary Decision Diagram (BDD) based sampling representation for transition probability matrices to overcome the memory bottleneck in both classical and Bayesian Markov Decision Process computations. By sampling a fixed number of BDDs, each entry in the expected TPM can be expressed implicitly. We develop efficient Bayesian update and lookup algorithms to make the BDD-based sampling representation applicable for the Bayesian/classical MDP problem. We also design a heterogeneous CPU-GPU computation framework to achieve improved performance. Our BDD-based approach on GPU reduces memory use by $137\times$ for classical MDP

with $32K$ states and $13\times$ for Bayesian MDP with 512 states compared to conventional representation. Our BDD-based sampling approach is $2225\times$ more compact in storing the TPMs, compared to the full matrix. Our approach can process a classical MDP (or BMDP) problem with $1G$ transitions (4.19×10^6 transitions) on a desktop computer, which to our best knowledge no existing technique has been able to accomplish.

7. PART 2: ANALYSIS OF THE UPPER ERROR BOUND OF DSS AND BDD-BASED APPROACHES

In this chapter, we will analysis the upper error bound of DSS and BDD-based sampling approach, which indicates the most error can be introduced by these two approaches. As discussed in Chapter 5 and 6, both DSS and BDD-based sampling approaches sacrifice the result accuracy to achieve a reduced memory utilization in the MDP/BMDP problems. Since the final policy of a MDP/BMDP problem depends on Eq. 7.1 as described in Chapter 2, and in Eq. 7.1, only $\overline{\mathcal{P}_{ij}}(a)$ is stored in the DSS or BDD-based format, therefore the error introduced by the approach comes from the multiplication term "$\overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda J^*(s_j, \overline{\mathcal{P}}')]$".

$$J^*(s_i, \overline{\mathcal{P}}) = \max_{a \in A}\{\sum_{\forall j \in S} \overline{\mathcal{P}}_{ij}(a)[g_{ij}(a) + \lambda J^*(s_j, \overline{\mathcal{P}}')]\} \tag{7.1}$$

In order to simplify the analysis, instead of analyzing Eq. 7.1, we analyze the error bound of "$\sum_{\forall j \in S}(\overline{\mathcal{P}}_{ij}(a) w_j)$" instead, where w_j is the element in a $1 \times N$ vector W, and N is the number of states. "$\sum_{\forall j \in S}(\overline{\mathcal{P}}_{ij}(a) w_j)$" is essentially the inner product of two vectors, and each vector is $1 \times N$. We first need to make some assumption on W.

Assumption 1. *The N elements in vector W are all between 0 and 1, and the average value of the N elements in W is $\frac{1}{2}$.*

7.1 Upper Error Bound of DSS

In order to find the upper error bound of the DSS approach, we first make an assumption about each row in the $\overline{\mathcal{P}}$ matrix.

Assumption 2. *If we sort the elements in the ith row of the $\overline{\mathcal{P}}$ matrix (denoted as $\overline{\mathcal{P}}_{i*}(a)$) in descending order, the elements in $\overline{\mathcal{P}}_{i*}(a)$ can be expressed as a linear function on the element index.*

Under Assumption 1 and Assumption 2, the elements in $\overline{\mathcal{P}}_{i*}(a)$ and the W vector is shown in Fig. 7.1. In the top figure of Fig. 7.1, $\overline{\mathcal{P}}_{ij}(a)$ are sorted in descending order. The blue solid line is the original true $\overline{\mathcal{P}}_{ij}(a)$, and the red dash line is the value of the approximated $\overline{\mathcal{P}}_{ij}(a)$ using the DSS representation. Note that since the sum of all the elements in one row of the $\overline{\mathcal{P}}$ matrix should be 1. Therefore, $max(\overline{\mathcal{P}}_{ij}(a))$ under Assumption 2 should be $\frac{2}{N}$. The bottom figure shows the corresponding w_j value in order to maximize the error, in black solid line. R is the ratio of how many elements in $\overline{\mathcal{P}}_{i*}(a)$ are kept to be the original values.

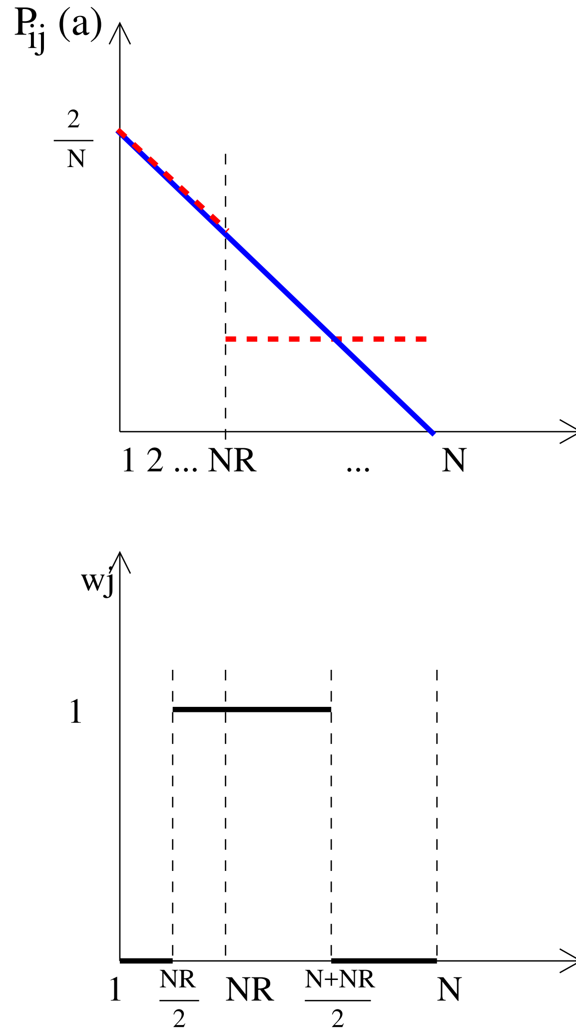


Figure 7.1: DSS Upper Error Bound: $\overline{\mathcal{P}}_{i*}(a)$ and W

Under these assumptions, the upper bound of the error E of $\sum_{\forall j \in S}(\overline{\mathcal{P}}_{ij}(a)w_j)$ is shown in Eq. 7.2. The corresponding curve of E v.s. R is shown in Fig. 7.2. When R increases, E will

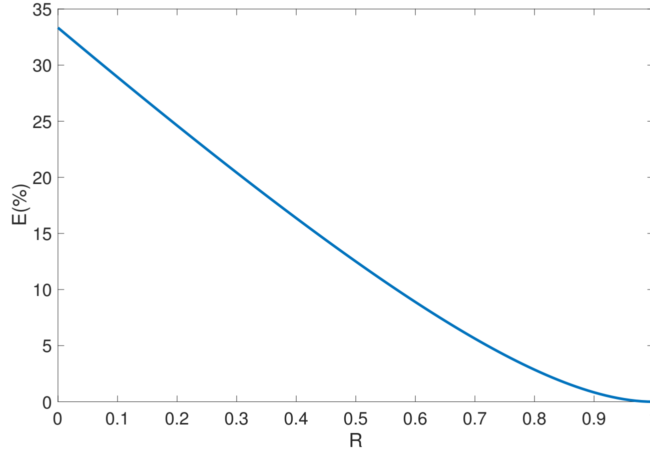


Figure 7.2: DSS Upper Error Bound Curve: Linear

decrease, since more entries are represented accurately.

$$E(\%) = \frac{(1-R)^2}{3-2R} \times 100\% \tag{7.2}$$

Another possible assumption about each row in the $\overline{\mathcal{P}}$ matrix is described in Assumption 3.

Assumption 3. *If we sort the elements in the ith row of the $\overline{\mathcal{P}}$ matrix (denoted as $\overline{\mathcal{P}}_{i*}(a)$) in descending order, the elements in $\overline{\mathcal{P}}_{i*}(a)$ form a step function upon the element index.*

Note that compared with Assumption 2, Assumption 3 is more likely to be applicable to the problem studies in this thesis, where the elements in $\overline{\mathcal{P}}$ matrix obey the Dirichlet distribution. Similarly, under Assumption 3, the elements in $\overline{\mathcal{P}}_{i*}(a)$ and the W vector are shown in Fig. 7.3, for the upper error bound calculation. Assume that there are $NR + Q$ non-trivial elements in $\overline{\mathcal{P}}_{i*}(a)$.

Note that since all the elements in one row of the $\overline{\mathcal{P}}$ matrix should be 1. Therefore, $max(\overline{\mathcal{P}}_{ij}(a))$ under Assumption 3 should be $\frac{1}{NR+Q}$. In Fig. 7.3, it doesn't matter what the values of w_j are

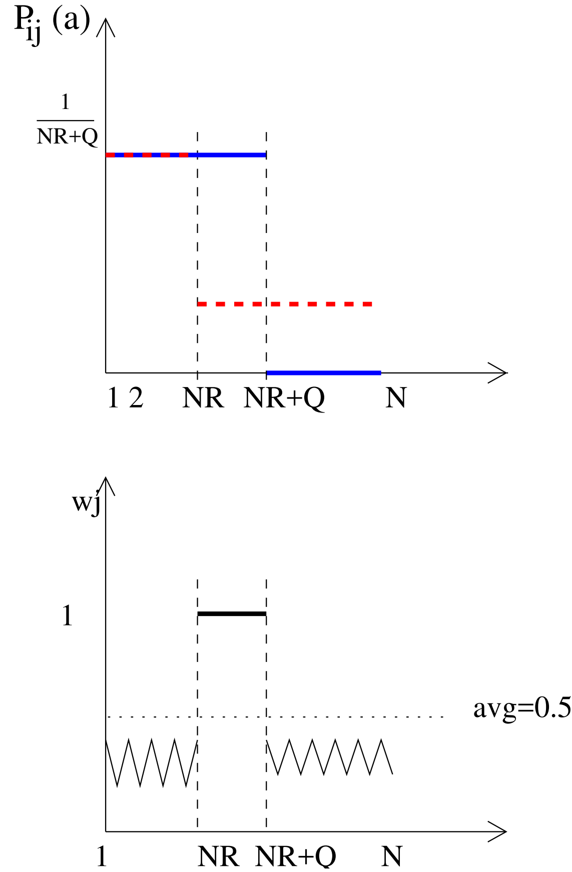


Figure 7.3: DSS Upper Error Bound Curve: $\overline{\mathcal{P}}_{i*}(a)$ and W

when the indices are out of $[NR, NR+Q]$, as long as the average value of w_j is 0.5. Under such assumption, the upper error bound E of $\sum_{\forall j \in S}(\overline{\mathcal{P}}_{ij}(a)w_j)$ is shown in Eq. 7.3, where $Q = k \times N$, $k \in [0, 1]$ and $0 < k + R \leq 1$

$$E(\%) = \frac{(1-R-k)k}{(1-R)(R+k)} \times 100\% \tag{7.3}$$

The corresponding curve of E v.s. R and k is shown in Fig. 7.4. When R increases, E will decrease, since the ratio of $\overline{\mathcal{P}}_{i*}(a)$ entries being represented accurately increases. However, when

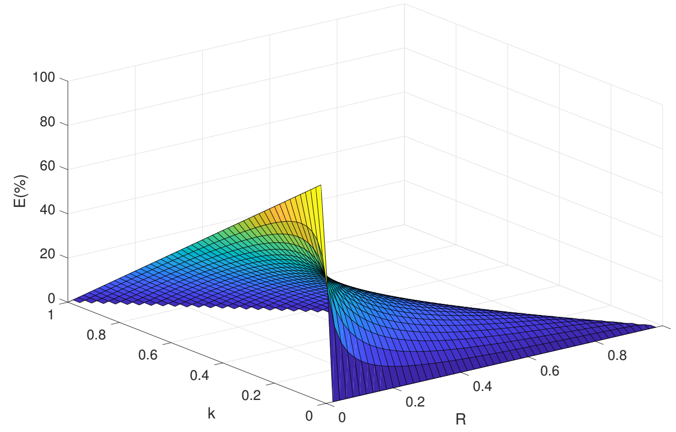


Figure 7.4: DSS Upper Error Bound Curve: Step

k increases, E can both increase or decrease, which actually follows the theoretical analysis from Eq. 7.3. The error of the DSS approach come from using an average value to represent all the "trivial" values in the $\overline{\mathcal{P}}$ matrix.

7.2 Upper Error Bound of BDD-based Sampling Approach

The BDD-based sampling approach sacrifices result accuracy in order to reduce the memory storage requirement. Unlike DSS, the error of the BDD-based sampling approach is caused by the limited resolution. As with the DSS approach, in the BDD-based sampling approach, we also have the same two different assumptions on the distribution of the elements in the $\overline{\mathcal{P}}$ matrix.

When we follow Assumption 2, the elements in $\overline{\mathcal{P}}_{i*}(a)$ and the W vector are shown in Fig. 7.5 for the error upper bound. In the top figure of Fig. 7.5, $\overline{\mathcal{P}}_{ij}(a)$ are sorted in descending order. The blue solid line is the original true $\overline{\mathcal{P}}_{ij}(a)$, and the red dashed line is the value of the approximated $\overline{\mathcal{P}}_{ij}(a)$ using BDD-based sampling representation. In the bottom figure of Fig. 7.5, for every $\frac{N}{M}$ elements in W, half of them are $1's$, and the other half are $0's$. By following Fig. 7.5, the upper error bound E is:

$$E(\%) = \frac{3}{(2M - 1 + \frac{7}{M})} \times 100\% \tag{7.4}$$

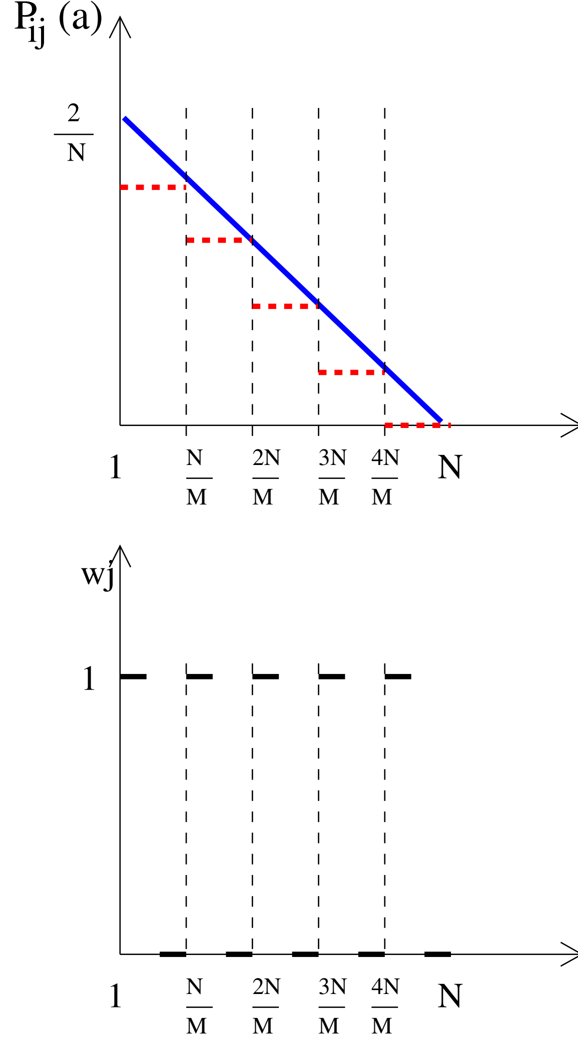


Figure 7.5: BDD Upper Error Bound: $\overline{\mathcal{P}}_{i*}(a)$ and W

Usually M is greater than 100. From Eq. 7.5, when M increases, E decreases. As discussed in Chapter 6, when M increased, the resolution increases. Hence, the upper error bound decreases. The corresponding curve of E v.s. M is shown in Fig. 7.6.

When we follow Assumption 3, for the upper error bound analysis, the $\overline{\mathcal{P}}_{i*}(a)$ and W vectors are shown in Fig. 7.7. We assume that there are $(\frac{N}{2}+Q)$ non-trivial values in $\overline{\mathcal{P}}_{i*}(a)$, and $-\frac{N}{2} \leq M \leq \frac{N}{2}$. Note that M can be negative. Again, since the sum of all the elements in $\overline{\mathcal{P}}_{i*}(a)$ should be 1, $P_{max} = \frac{2}{N+2Q}$ P_{bdd} is the numerical value of the approximated P_{max}, using BDD-based sampling representation. As discussed in Algorithm 6, $P_{bdd} = \frac{\lfloor P_{max} \times M \rfloor}{M}$. Therefore, the error

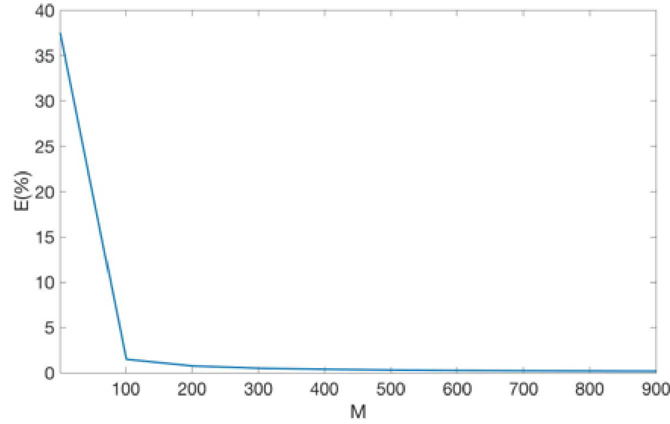
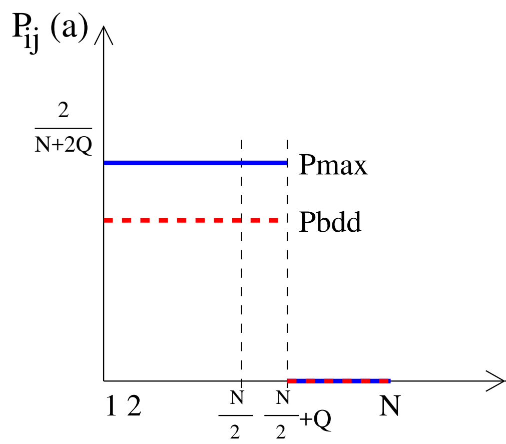


Figure 7.6: BDD Upper Error Bound Curve: Linear

upper bound E is as follows:

$$E(\%) = \frac{P_{max} - P_{bdd}}{P_{max}} \times 100\% \tag{7.5}$$

Generally speaking, if Q increases, E will increase as well. This is because more elements are expressed approximately. When M increases, P_{bdd} will decrease. Hence E will increase accordingly. Note that in this case, under the same Q, E is ralated to both N and M. This is because P_{max} is determined by N and Q, and P_{bdd} is the floor value of P_{max}. Error comes from the difference between P_{max} and P_{bdd}. Therefore, N will determine E as well. In Fig. 7.8, we show the curve of E v.s. N and M under different Q values.



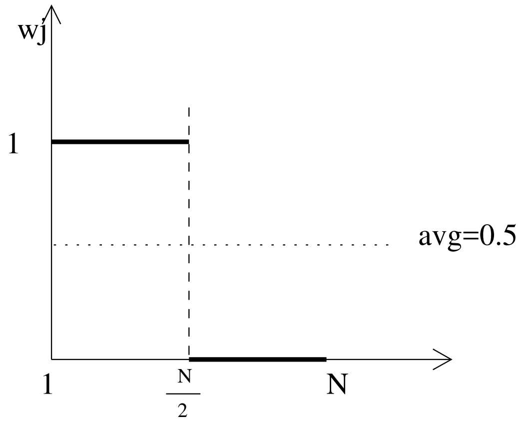
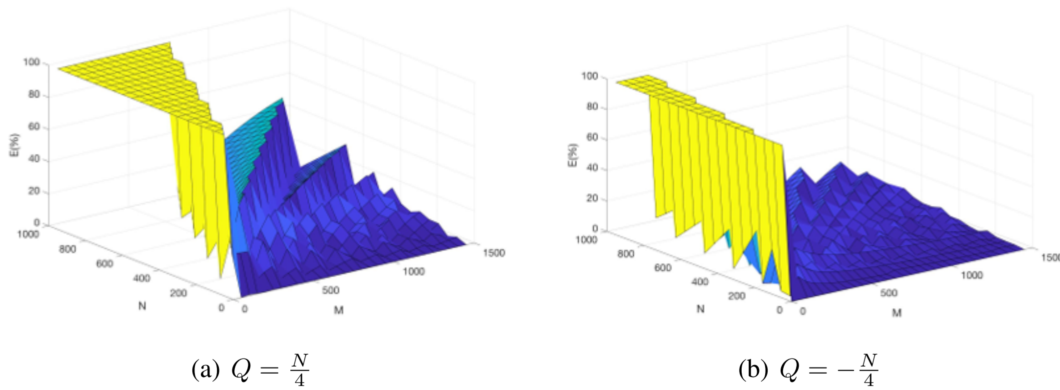


Figure 7.7: BDD Upper Error Bound: $\overline{\mathcal{P}}_{i*}(a)$ and W



(a) $Q = \frac{N}{4}$ (b) $Q = -\frac{N}{4}$

Figure 7.8: BDD Upper Error Bound Curve: Step

8. PART 3: SCALED POPULATION ARITHMETIC FOR EFFICIENT BMDP COMPUTATION

In this chapter, we will discuss a novel scaled population arithmetic scheme for stochastic computing, which is illustrated using BMDP, matrix inner production and MNIST image classification as application examples.

8.1 Background and Introduction

Approximate computing is an approach with an emphasis on area and power efficiency, while sacrificing accuracy. For certain classes of applications that are tolerant to computational errors, approximate computing can achieve better area and power characteristics compared with exact arithmetic. Hence, it has shown promising application in scientific computing [67], machine learning [68], signal processing [69], and real-time systems [70].

Popular techniques for approximation computing include the following: precision scaling [71], inexact or faulty hardware [72], voltage over-scaling [73], and skipping tasks and memory accesses [74]. Among these techniques, stochastic computing [16] is a non-conventional arithmetic scheme for area-efficient implementation of error-tolerant applications. Stochastic computing has received renewed interest due to, among other reasons, the degrading reliability of recent VLSI fabrication processes, its purported decrease in power and energy, and its robustness to bit-flip errors. In stochastic computation, values are represented by binary bit streams, and the arithmetic operations can be processed by simple logic circuits, such as OR/AND gates for addition and multiplication, respectively.

However, classical stochastic computing, which is abbreviated as SC in the sequal, has its limitations. First, although it was claimed that SC has a high error tolerance to bit flips [75], its accuracy depends heavily on the *density* and the randomness of the 1's in the binary bit-stream [17]. To the best of our knowledge, the error of SC have not been quantified to date. This section presents an error analysis for the proposed scaled population arithmetic as well as SC. Second, since SC

uses a population-based representation alone for all numbers, it can only represent numbers in $[0, 1]$. The limitation can be problematic when overflow occurs in the operations, especially in addition. The third limitation of SC is the runtime complexity. Although the arithmetic operation units consist of only OR/AND logic gates, the supporting units, e.g., the random number generator (RNG) and the shuffler, have a runtime complexity of $\mathcal{O}(k)$, where k is the number of bits in SC representation. These weaknesses limit the applicability of SC in real world applications.

In order to alleviate the above limitations, we propose a new Scaled Population (SP) arithmetic based computation which achieves fast, approximate computing with a low area/power overhead and improved accuracy. SP arithmetic uses some of the basic ideas of SC, but with three key enhancements: a) the inherent serialization in SC is avoided; b) the errors of SC are significantly reduced by providing a scaling (exponent) term in SP arithmetic; and c) the range of numbers that can be represented by SP is much larger than what is possible in SC. The key design goal of SP arithmetic is that each operation be computed using $\mathcal{O}(1)$ gate delays (as opposed to clock cycles). Unlike SC, SP never allows any operation to perform a serial traversal of the bits of the operand. The SP arithmetic achieves a dramatic speedup over SC.

Our proposed SP computation greatly improves the accuracy of a single multiplication and addition operation by $6.3\times$ and $4.0\times$, compared with SC. Our experimental results show that for addition and multiplication, our SP approach uses $7.13\times$ and $3.75\times$ fewer LUTs than conventional floating point number based arithmetic circuit, respectively. We also test our approach in the scenarios of BMDP problem, matrix inner product and image classification using MNIST dataset. Our approach achieved a 31.89%, 2072.32% and 32.79% improvement over SC in terms of the accuracy for the BMDP, matrix inner production and image classification problem, respectively.

The key contributions of the SP approach are:

- Introduction of SP, with larger range, better error and reduced delay than SC.
- Achieving $\mathcal{O}(1)$ delay for all operations, and design for speed and accuracy over SC.
- Quantifying the errors of SP arithmetic and SC.

- Applying the SP approach on simple addition/multiplication, BMDP problem, matrix inner product and MNIST classification.

8.2 Stochastic Computing and Previous Works

Stochastic computing is an approximate arithmetic approach that allows area-efficient circuit implementation for some operations on fractional numbers. Consider a fractional number $P_x \in [0, 1)$. In conventional binary number representation, it is represented as $X = x_1x_2...x_k$ such that $P_x = \sum_{i=1}^{k} 2^{-i}x$. In stochastic computing, by contrast, it is represented by a $\Pi-$bit vector π, where $|\pi| \leq \Pi$ bits are randomly chosen to be 1, so that $P_x = \frac{|\pi|}{\Pi} \in [0, 1]$.

In [16, 76], the key elements of stochastic computing, including circuit implementations and a comparison with analog computing, are introduced. One prominent benefit of stochastic computing is the very low area cost in implementing certain arithmetic operations. Fig. 8.1 and Fig. 8.2 show examples of multiplication and addition operations, respectively. The area advantage is clearly evident. The computing results depend on the number and the locations of 1's in the bit-streams, and therefore are usually inaccurate. Moreover, the 1's are required to be randomly located in each bit-stream.

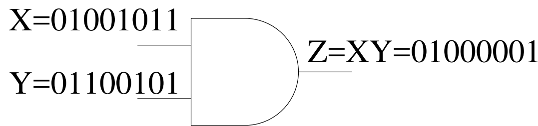


Figure 8.1: Multiplication: $\frac{4}{8} \times \frac{4}{8} = \frac{2}{8}$

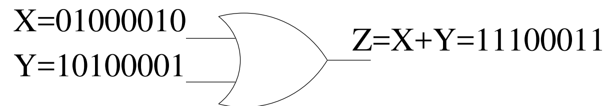


Figure 8.2: Addition: $\frac{2}{8} + \frac{3}{8} = \frac{5}{8}$

The work of [77] shows how to realize subtraction in stochastic computing by using a multiplexer (MUX) and a NOT gate. In [76], addition and subtraction approaches are introduced to solve the overflow problem. Although this approach solved the problem of overflow, it is a serial process, i.e., only one bit in π is processed at a time. Hence it can easily form a performance bottleneck. A stochastic division circuit design, called CORDIV, is proposed in [78].

In the basic form of stochastic computing, only numbers in $[0, 1]$ are allowed. In [17], multiple representation schemes are reported to overcome this limitation. The bipolar format increases the operands' range to [-1,1] [79]. In [80], the numerical value of a bit-stream representation is no longer population-based, but interpreted as the ratio of 1's to 0's, which increases the range to $[0, +\infty]$. Although these approaches increase the range of the operands, they require a more complicated design for the arithmetic operation circuits, while our SP arithmetic achieves a large range for number representation while ensuring that all operations incur only $\mathcal{O}(1)$ gate delays.

Since the accuracy of stochastic computation relies highly on the randomness of the 1's in the bit-stream, data shuffling has been used in SC [81], through random number generators. However, these approaches either introduce more overhead with respect to runtime, area and power, or are not able to introduce enough randomness. By contrast, our SP approach makes use of several multi-level Linear-Feedback Shift Register (LFSR) based shuffler, which improves both efficiency and randomness.

In spite of the considerable studies on stochastic computing, the research attention on quantifying its accuracy and error characteristics has been surprisingly light. In particular, there is a lack of a systematic investigation on the errors due to the densities of 1's in the bit-stream of the SC number representation. A key contribution of our SP arithmetic is to fill this void and remarkably improve the accuracy over SC.

8.3 Scaled Population Arithmetic

8.3.1 Number Representation

The number representation in Scaled Population (SP) arithmetic is an enhancement of that in SC, with a scaling (exponent) term which allows SP to cover a range beyond $[0, 1]$. Specifically, the SP representation of a number x is an M-bit tuple $x = \{\sigma, \pi\}$, where σ is a Σ-bit scaling term and π is a Π-bit population vector such that $M = \Sigma + \Pi$. The numerical value of x is $\frac{|\pi|}{\Pi} \times 2^{(\sigma - \Sigma_0)} \simeq x$, where $|\pi|$ is the number of 1's in the population vector π, and Σ_0 is a constant, typically chosen to be $2^{(\Sigma-1)}$. The reason that we include the Σ_0 constant is to allow the value of the scaling term in

the SP representation (i.e. $2^{(\sigma-\Sigma_0)}$) to be smaller than 1, so that we can have increased resolution, allowing us to increase (or decrease) the density of the population vector without changing the numerical value of x. We note that the 1's in the population vector π are uniformly distributed, which has the similar characteristic with SC [76]. The SP representation described above only handles positive numbers. However, augmenting SP to handle signed computation can be easily accomplished by adding a sign bit.

For example, if $\{\sigma, \pi\} = \{110, 1011010101\}$ then $|\pi| = 6$, $\Pi = 10$, $\Sigma = 3$ and $\sigma = 6$. Hence the numerical value of the SP number x is $\frac{6}{10} \times 2^2$, which equals 2.4. Note that the inclusion of the scaling term is something that SC does not have. The SP number representation not only covers a much larger range of numbers, but also, and more importantly, facilitates arithmetic operations that have improved computing accuracy, as will be elaborated in the sequal.

8.3.2 Arithmetic and Supporting Operations

In this section, we will describe two most commonly used arithmetic operations, multiplication and addition, followed by a description to supporting operations such as shuffling, density checking and scaling.

Fig. 8.3 is a top level block diagram of the proposed SP arithmetic system. The input operands are represented as conventional binary numbers X and Y. The generators convert X and Y to the SP format, e.g., $x = \{\sigma, \pi\}$. Then, the operands x and y in SP format are fed into the arithmetic processing units, such as adder and multiplier, for computation.

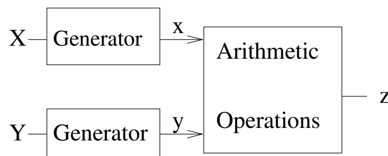


Figure 8.3: The top level view of SP scheme.

In designing SP-based arithmetic circuits, we ensure that each operation incurs $\mathcal{O}(1)$ gate de-

lays. In particular, any computation that requires us to iterate over the M bits of an SP vector (which requires serialization) is avoided. Note that all the operations described below are approximate in nature. Also, the computations on the scaling term are efficient, since Σ is a very small value.

8.3.2.1 Multiplication

Multiplication uses an AND gate as the underlying function, as in SC. In [76], it was proved that an AND gate is able to achieve multiplication when the operands are presented in population-based vectors. In SP arithmetic, we propose a scaling operation, to be performed prior to each multiplication in order to improve the computation accuracy. This improvement is based on the observation that the multiplication accuracy is higher when there are more 1's in the population vectors of the operands. Consider multiplication between x and y, with the result being z. The computational error ε from the AND gate-based multiplication decreases when $\frac{|\pi_x|}{\Pi}$ or $\frac{|\pi_y|}{\Pi}$ increase.

Proof. Let $p_x = \frac{|\pi_x|}{\Pi}$, $p_y = \frac{|\pi_y|}{\Pi}$ and $p_z = \frac{|\pi_z|}{\Pi}$. Then each bit in π_x, π_y and π_z is 1 with a probably of p_x, p_y and p_z, respectively. For the ith bit, ideally $p_z = p_x \times p_y$. Hence, the error of the ith bit in π_z occurs when the probability of it being 1 is not $p_x \times p_y$, i.e., the error at the ith bit in π_z is $\varepsilon_i = (1 - p_x \times p_y)$. When p_x or p_y increases, ε_i decreases. Therefore, considering the entire population vector, when p_x or p_y increases, ε will decrease as well. □

Based on Lemma 8.3.2.1, the average error of multiplication with p_x and p_y varying over the interval $[0, 1]$ is $\int_0^1 \int_0^1 (1 - p_x p_y) dp_x dp_y = 0.75$.

The scaling term in the SP number representation allows us to control the density of population vectors by scaling. A *density checker unit* and a *scaling unit* are needed together to perform the density control. Also, the randomness of the distribution of 1's in the population vectors affects the accuracy as well. The more uniformly randomly the 1's are distributed, the more accurate the result is. Therefore, a *shuffle unit* is additionally needed in our design for achieving randomness. The key elements for the SP multiplication are shown in Fig. 8.4.

Consider multiplication between x and y. We first check if the population vectors of the two

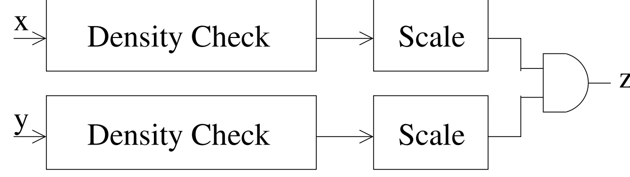


Figure 8.4: The SP-based Multiplication

numbers are dense enough, i.e. $|\pi_x| \geq T_1$ and $|\pi_y| \geq T_1$, where T_1 is a sufficiently high fraction. Typically, $T_1 \sim 0.7 \times \Pi$ is a good value according to our experimental results. Such density checking is performed by the density checker unit. If the numbers are not dense enough, the population vectors are scaled to make them dense enough, and corresponding changes are made to the scaling terms. Now we compute $\mathsf{z} = \mathsf{x} \times \mathsf{y}$ as $\pi_z = \pi_x \& \pi_y$, and $\sigma_z = \sigma_x + \sigma_y - \Sigma_0$.

8.3.2.2 Addition

In our approach, addition is approximately achieved by using an OR gate. Suppose we want to add x and y. A high accuracy requires that the population vectors of both numbers have a density lower than a threshold T_2. Suppose we want to add x and y, with the result being z. The error ε of the OR-based addition decreases when $\frac{|\pi_x|}{\Pi}$ or $\frac{|\pi_y|}{\Pi}$ decreases.

Proof. Let $p_x = \frac{|\pi_x|}{\Pi}$, $p_y = \frac{|\pi_y|}{\Pi}$ and $p_z = \frac{|\pi_z|}{\Pi}$. Then, each bit in π_x, π_y and π_z is 1 with a probably of p_x, p_y and p_z, respectively. The OR operation performed on π_x and π_y leads to $p_z = p_x + p_y - (p_x \times p_y)$. For the ith bit in π_x and π_y, the error at the ith bit in π_z is $\varepsilon_i = p_x \times p_y$. In other words, the error of the ith bit in π_z occurs when the ith bits in π_x and π_y are both 1's. From the above equation, when p_x or p_y decrease, ε_i decreases. Thus, considering the entire population vector, when p_x or p_y decreases, ε will decrease. □

According to Lemma 8.3.2.2, the average error of addition with p_x and p_y varying over the interval $[0, 1]$ is $\int_0^1 \int_0^1 (p_x p_y) dp_x dp_y = 0.25$.

However, unlike multiplication, as long as the 1's in the population are uniformly distributed, we don't have to use a density check or scaling unit for addition. Instead, we perform *skewed*

addition, where each operand occupies different halves of the π bits. Therefore, in *skewed* addition, it is guaranteed that no matter what the density of the operands' population vectors is, the 1's in the two operands will never be aligned at the same bit position. To perform the skewing, we use 2 Π-bit masks m_x and m_y, where m_x has the left $\frac{\Pi}{2}$ bits set to 1, and m_y has the right $\frac{\Pi}{2}$ bits set to 1. The result will be $\pi_z = (\pi_x \& m_x)|(\pi_y \& m_y)$, with $\sigma_z = \sigma_x + 1 = \sigma_y + 1$. If $\sigma_x \neq \sigma_y$, then the scaling unit will be used to adjust the density of the population vectors of x and y until $\sigma_x = \sigma_y$. Fig. 8.5 shows an example of how *skewed* addition is processed. In Fig. 8.5, $x = \{01, 00111001\}$ and $y = \{01, 10000010\}$. The numerical values of x and y are $2^{(1-2^{(2-1)})} \times \frac{4}{8} = 0.25$ and $2^{(1-2^{(2-1)})} \times \frac{2}{8} = 0.125$, respectively, assuming $\Sigma_0 = 2$. After the $skewed$ addition, the numerical value of the result $z = \{10, 00110010\}$ is $2^{(2-2^{(2-1)})} \times \frac{3}{8} = 0.375$, which matches the theoretical addition.

Note that $skewed$ addition relies on the randomness of the distribution of 1's in the population vector. If the 1's in the population vector are not uniformly distributed, $\pi_x \& m_x$ or $\pi_y \& m_y$ will not have half of the 1's in π_x or π_y approximately, which will introduce an error into the final addition result.

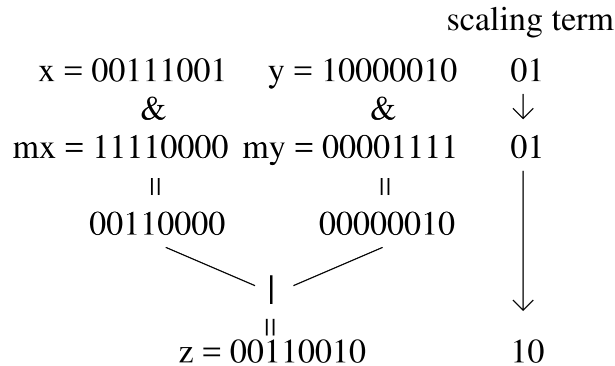


Figure 8.5: SP-based *Skewed* Addition

8.3.2.3 *Generator*

The generate operation converts a conventional binary number to the SP format. The generators used in SC have a computational complexity that is proportional to the bit stream length [82]. Our approach generates a π population vector with $\mathcal{O}(1)$ gate delays, by replicating the bits of the original number based on their bit position. Assuming the original binary number to be $X = \{x_{n-1}, x_{n-2}, ...x_1, x_0\}$, where n is the number of bits. We convert it to the SP format by producing 2^i copies of bit x_i. The resulting bits are then fed to a shuffle unit, which randomizes the bits of π and yields a SP representation of X. Note that in case this would result in a π vector with $|\pi| > \Pi$, then we appropriately adjust the population vector by decimating or dropping the additional bits. Since we assume the 1's are uniformly distributed after shuffling, dropping the additional bits won't change the numerical value of the population vector.

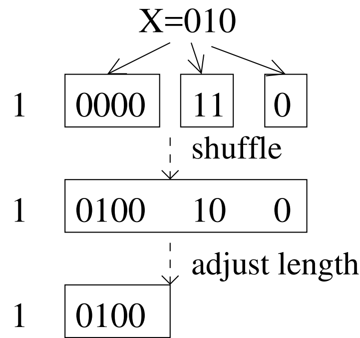


Figure 8.6: An Example of Generate Operation, with $X = 0.25$, and $\Pi = 4$

An example of the generate operation is shown in Fig. 8.6. The numerical value of the binary weighted number $X = 0, 1, 0$ is 0.25. Assume that the length of the population vector Π is 4, $\Sigma = 2$ and $\Sigma_0 = 1$. The initial scaling term is 1, since there is no scaling, initially.

Generating 2^i copies of bit x_i is accomplished by wires. Since the shuffle unit and the length adjust unit are both done with $\mathcal{O}(1)$ gate delays (as will be discussed in the following sections), the generator incurs $\mathcal{O}(1)$ gate delays as well.

8.3.2.4 *Shuffle Unit*

In order to let the 1's in the population vector π be uniformly randomly distributed, a 2-level shuffle unit is designed in our SP arithmetic system. The bits in π are grouped into W chunks. The *first level* of the shuffle unit generates W permutations of the chunks, and a particular permutation is selected by an LFSR that randomly cycles through a count between 1 and W. Next, within all the chunks, the *second level* generates w permutations of the bits within every chunk, and a particular permutation is selected by an LFSR that randomly cycles through a count between 1 and w. Note that $wW = \Pi$. In order to reduce the number of LFSRs, we select the same bit-level permutation for all the chunks. Additionally, a third LFSR randomly selects a logical shift of the resulting permuted number, performed by a barrel shifter. We choose between left and right shifts randomly. Further, the number of positions V to shift is also chosen randomly.

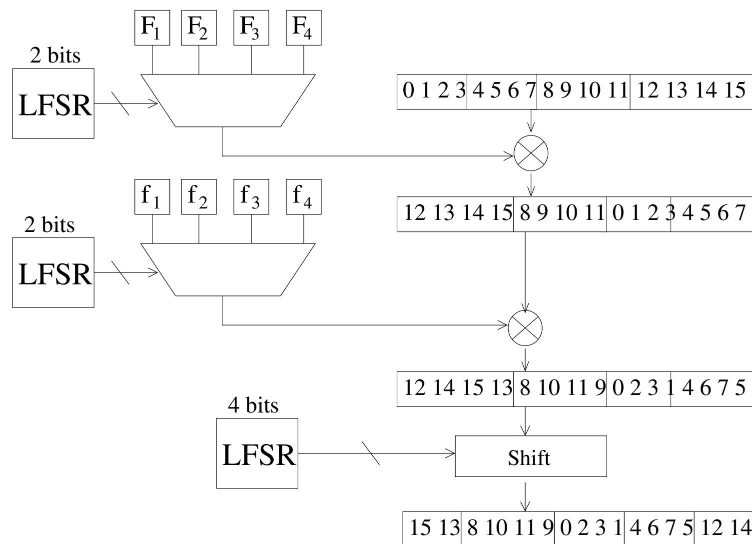


Figure 8.7: Shuffle Example ($W = w = 4$)

Fig. 8.7 shows an example of the shuffle unit operation. Since $\Pi = 16$, the position indices as shown on the top position vector range from 0 to 15. F_1, F_2, F_3, F_4 are different permutations for

4 chunks in π (i.e. $W = 4$), while f_1, f_2, f_3, f_4 are the bit-level permutation within the chunks (i.e. $w = 4$). The $\otimes$ symbol means applying the permutation on π, which essentially swaps bits around.

8.3.2.5 Density Checker Unit

In SP multiplication, we need to test if the population density $\frac{|\pi|}{\Pi}$ is greater than T_1, which is a threshold greater than 0.5. We convert the problem to check if $\frac{|\pi|}{\Pi} \times \frac{0.5}{T_1} > 0.5$. Note that $\frac{0.5}{T_1}$ is smaller than 1, and so we use a Π-bit mask with $\frac{0.5}{T_1} \times \Pi$ 1's, and bit-wise AND this mask with π in order to get $\frac{|\pi|}{\Pi} \times \frac{0.5}{T_1}$, which we denote as π_2. Next, we check if π_2 has a density greater than 0.5. Note that we need to do this approximately, with $\mathcal{O}(1)$ gate delays. Since the 1's in π_2 are randomly distributed, we can do this approximately by checking $\pi' = (\pi_2)|(\pi_2 << 1)|(\pi_2 << 2)$. In other words, we perform the logical OR of π_2 with its left-shifted counterparts (by 1 and 2 bit positions respectively). If the result π' is all 1's, we conclude that the density of $|\pi|$ is greater than T_1. The test of whether the number is all 1's is done by using a hash function (HF) of Class 3 [83], in order to ensure a $\mathcal{O}(1)$ gate delay. We hash π', and perform a bit-wise comparison of the result with the pre-computed hash of a population vector with Π 1's.

8.3.2.6 Scaling Unit

According to Lemma 8.3.2.1, the density of the population vector needs to be adjusted by the scaling unit to achieve an improved accuracy for SP multiplication. Along with a scaling operation, the scaling term σ for each operand needs to be updated accordingly.

Suppose we would like to adjust the density of the population vector π by β, to yield the result $\pi \cdot \beta$. We will discuss how to process such an adjustment for different values of β.

1. When $0 < \beta < 1$, we use a mask M_β with Π bits, where there are $\beta\Pi$ 1's in the mask at arbitrary locations. Then we perform a bit-wise AND operation of the mask M_β with π, shuffle the result, and increase σ by $\frac{1}{\beta}$, in order to keep the numerical value of the SP representation to be the same.

2. When $1 < \beta < 2$, we solve the problem of computing $\beta \times |\pi|$ as follows. We first scale down π by $\frac{\beta}{2}$ by using the mask $M_{\frac{\beta}{2}}$ with density of 1's being $\beta/2$, and resulting population

vector is called π'.

$$\pi' = \pi \& M_{\frac{\beta}{2}} \tag{8.1}$$

In Eq. 8.1, $M_{\frac{\beta}{2}}$ is a mask with Π bits, where $\frac{\beta}{2}\Pi$ bits are 1's. Since, $\beta \times |\pi| = 2 \times (|\pi| \times \frac{\beta}{2}) = 2 \times |\pi'|$, we next double the density of π' by using the following equation:

$$\pi^d = (\pi' | \pi'^s) | (\pi' \& \pi'^s) \tag{8.2}$$

In Eq. 8.2, π^d is the population vector after doubling, and π'^s is the population vector after shuffling π'. If the 1's are uniformly randomly distributed in π', $\pi' | \pi'^s$ will result in $2\pi' - \pi'^2$. Therefore, another term of $\pi' \& \pi'^s$ is added in Eq. 8.2 as a compensation for the numerical loss of π'^2. Due to the error of using bit-wise OR operation to perform addition, π^d is only an approximate version of $2\pi'$. However, it is computed with $\mathcal{O}(1)$ gate delays.

3. When $\beta > 2$, we repeatedly double $|\pi|$ until the remaining adjustment ratio is less than 2. Then we can use the methods mentioned above to compute $\pi \cdot \beta$.

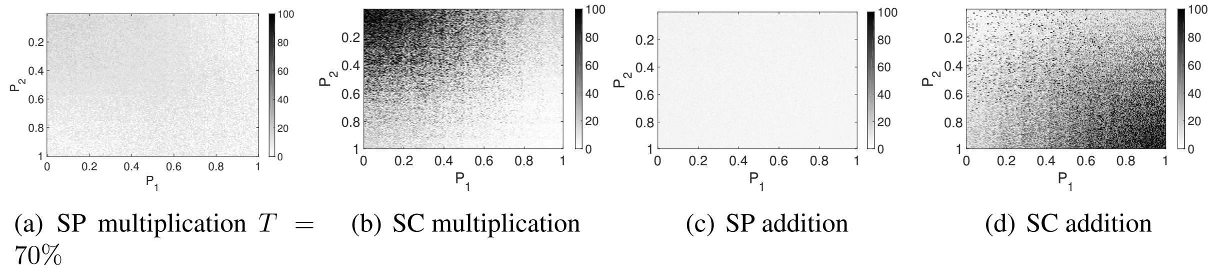
8.4 Experiment Results

The proposed SP arithmetic scheme is evaluated for single multiplication/addition operations, matrix inner product computation, and MNIST image classification, in terms of accuracy, power, delay and area. It is implemented on a Zybo Zynq-7000 development board which uses a Xilinx XC7Z010-1CLG400C FPGA device with 17,600 look-up tables (LUTs) and 35,200 flip-flops.

8.4.1 Single Arithmetic Operation

Table 8.1: Error Contribution of SP-based Multiplication (%)

T	Perfect	Generator	Density Check	Scale	Shuffle	Imperfect
50%	32.76(10.49)	32.91(10.87)	33.42(9.81)	35.02(12.66)	34.96(11.23)	35.12(13.98)
60%	16.39(4.33)	16.99(4.39)	17.73(5.91)	19.94(12.23)	18.30(7.92)	18.95(9.18)
70%	5.76(3.49)	5.91(3.50)	7.82(4.78)	9.94(11.28)	9.84(9.73)	10.26(11.02)



(a) SP multiplication $T = 70\%$ (b) SC multiplication (c) SP addition (d) SC addition

Figure 8.8: Relative errors proportional to darkness.

We first show the results of evaluating accuracy. Our goal of SP arithmetic is to improve accuracy over SC. The accuracy is evaluated by comparing average relative errors (over a large number of operations) with respect to the conventional SC using IEEE 32-bit floating point representation as a baseline for comparison. The errors are depicted as heat maps in Fig. 8.8, where error is proportional to darkness. The two axes indicate $P_1 = \frac{|\pi_1|}{\Pi}$ and $P_2 = \frac{|\pi_2|}{\Pi}$, which are the densities of 1's in the population vectors of the two operands. For multiplication, the density threshold which we use to decide whether to scale or not is $T = 70\%$ for both operands. As expected, SC multiplication causes large errors for low densities, while SC addition results in large errors for high densities. By contrast, the errors from the SP arithmetic are much smaller.

Since an operand of multiplication is scaled if the density of 1's in its population vector is lower than T, we analyze the errors for two different situations. Errors e_{lo} are for the case where either operand has density less than T. Errors e_{hi} are for the case where both operands have densities greater than T. We report the separated error results in the format of $e_{lo}(e_{hi})$. In this format, the average errors from SP multiplications are $10.26\%(11.02\%)$ while those from SC multiplication are $71.04\%(3.19\%)$. Please note that we categorize SC multiplication errors into the $e_{lo}(e_{hi})$ format for the ease of comparison. For addition, the average SP error is 5.83% while the error of SC is 23.43%.

We further show the errors from individual components of each SP arithmetic operation. In Table 8.1, columns 3-6 display the errors from generator, density check, scaling unit and shuffle unit, respectively, for SP multiplication. When estimate the error of one component, we compute the

other components using software, which is regarded as perfect and contributes no error. Column 2 summarizes the errors when all components are perfect and the errors are from the SP number representation alone, while Column 7 is for the cases where all components are realized in circuits. Such decomposed error analysis is performed for different threshold levels. Note that the shuffle operation has the highest error contribution. Note that Column 2 is the error we would like to have for the circuit realization of Column 7.

Table 8.2: Error Contribution of SP-based Addition (%)

Perfect	Generator	Shuffle	Imperfect
0.84	1.21	3.57	5.83

The errors from different components in SP addition are provided in Table 8.2. Please note that SP addition does not need the density checker or scaling unit. One can see that most of the errors are from the shuffle unit. The reason behind is that the shuffling results are not uniformly random enough.

Table 8.3: Efficiency Comparison for Multiplication ($\Pi = 32$)

	delay (ns)	#LUTs
SP	1.92	20
SC	4.28	16
Floating point	14.53	956
Fixed point	6.12	84

Next, we show the FPGA resource utilization and performance in comparison with SC, Floating point (conventional arithmetic using IEEE 32-bit floating point representation) and fixed point

(conventional arithmetic using 32-bit fixed point representation). The results for multiplication are summarized in Table 8.3. Note that for SP and SC, since the supporting units such as generator, shuffle unit, scaling unit, and density checker are shared among multiple operations, the area and LUTs used by these operations are not counted in Table 8.3. For reference, if these units are counted, SP-based addition and multiplication use $7.13\times$ and $3.75\times$ fewer LUTs than the conventional floating point number based arithmetic circuit, respectively. One can see that SP arithmetic is $2.2\times$ faster than SC. On the other hand, SP arithmetic uses fewer LUTs as conventional floating point arithmetic, and much fewer when not considering the supporting units. Thus, SP reaches a compromise between SC and conventional arithmetic on performance and resource utilization. Meanwhile, earlier results indicates its great accuracy improvement over SC. A similar trend can be observed for addition, whose results are in Table 8.4.

Table 8.4: Efficiency Comparison for Addition ($\Pi = 32$)

	delay (ns)	#LUTs
SP	1.45	22
SC	3.24	18
Floating point	12.81	535
Fixed point	4.96	48

8.4.2 Application 1: BMDP Problem

The accuracy of SP arithmetic is evaluated for the BMDP problem and the results are shown in Table 8.5. Note that in this experiment, the multiplication and addition in Eq. (2.6) are calculated using our population arithmetic. The max operation is performed with software. The table includes the results of BDD-based sampling representation described in Chapter 6 with a total number of BDDs $M = 64$, and a random decision maker which makes random decisions for every state. The way we evaluate the error is the same as the way we described in Section 6.3. The SP ap-

proach improves the accuracy by 31.89% on average compared to SC. Compared with BDD-based sampling approach, SP introduces error not only from the limited resolution in the population part, but also from the stochastic computation. Therefore, BDD-based sampling representation is more accurate.

Table 8.5: Error of BMDP Problem ($\%$) ($\Pi = 64$, $M = 64$)

Number of states	SC	SP	BDD	Random decision maker
4	9.25	5.26	4.67	17.01
8	8.22	6.37	5.12	17.62
16	7.23	5.77	5.01	17.90
32	9.98	5.69	4.11	16.45
64	8.87	6.81	4.39	15.57
128	7.05	4.56	3.67	17.51
Avg	8.43	5.74	4.50	17.01

8.4.3 Application 2: Matrix Inner Product

The accuracy of SP arithmetic is also evaluated for the matrix inner product computation and the results are shown in Table 8.6. The table includes the results of Perfect SP, where all its components are implemented with software and the only approximation is from the number representation. Errors from individual operations accumulate in an application that contains many arithmetic operations. The errors increase with the vector size. The SP approach improves the accuracy by $20.72\times$ on average compared to SC. The errors from perfect SP and fixed point only comes from the limited resolution of the number representation. Therefore, they are more accurate.

Table 8.7 and Table 8.8 show the relative error of the matrix inner production when the bitwidth is 64 and 128 correspondingly. When the bitwidth increases, the error of all the population-based representations (SC, SP and perfect SP) and fixed point representation decrease. However, the error of the population-based representations decreases more rapidly than fixed point. This is because all the bits are weighted the same in the population-based approaches, while in fixed point representation, the bits are binary weighted.

Table 8.6: Error of Matrix Inner Production (%) ($\Pi = 32$)

Vector size	SC	SP	Perfect SP	Fixed point
32	172.50	16.55	1.00	0.21
64	283.52	18.92	1.11	0.38
128	420.15	21.03	1.19	0.46
256	589.39	24.59	1.29	0.49
512	797.42	28.02	1.38	0.52
Avg	452.21	21.82	1.19	0.41

Table 8.7: Error of Matrix Inner Production (%) ($\Pi = 64$)

Vector size	SC	SP	Perfect SP	Fixed point
32	143.71	9.38	0.68	0.20
64	251.42	11.92	0.74	0.37
128	391.05	13.63	0.78	0.44
256	459.19	14.88	0.83	0.48
512	582.21	17.14	0.87	0.52
Avg	365.52	13.39	0.78	0.40

Table 8.8: Error of Matrix Inner Production (%) ($\Pi = 128$)

Vector size	SC	SP	Perfect SP	Fixed point
32	115.97	5.73	0.41	0.20
64	213.72	6.29	0.48	0.37
128	342.81	7.30	0.54	0.43
256	399.31	10.17	0.62	0.47
512	439.07	12.42	0.69	0.50
Avg	302.18	8.38	0.55	0.39

8.4.4 Application 3: MNIST Digit Classification

The effect of SP approximation is also evaluated in a neural network application on MNIST digit classification [84]. The neural network has two hidden layers, 784 input nodes and 200 hidden layer nodes. The training set and test set have 60,000 samples and 10,000 samples, respectively. The size of each image is 28×28 pixels. The multiplications and additions in the network are implemented with SP, SC, conventional floating point and fixed point arithmetic. The classification success rates from these arithmetic methods of different bitwidth are listed in Table 8.9. In SP and

perfect SP, the desired density for multiplication is $T = 70\%$.

On average, SP can reach accuracy of about 81%, which is a significant improvement over the 60% accuracy by SC. When the length of the bits increases, the accuracy of IEEE-FP doesn't change appreciably. However the length of the bits influences the accuracy of perfect SP, SP and SC, since these 3 approaches are population based. Generally speaking, SC has the worst accuracy since first, it is not able to handle values greater than 1, and second, it is not able to adjust the density of the population vector, so that it suffers from the inherent error of SC discussed in Lemma. 8.3.2.1 and Lemma. 8.3.2.2. The accuracy of the perfect SP is the upper bound of the accuracy of SP, since every component in perfect SP has no error. Binary-weighted representation has a slightly better accuracy than perfect SP. This is because although the perfect SP eliminates the error of the hardware implementation, it still suffers from the limited resolution. Compared with classical SC, SP improves the accuracy by about 30%. Considering the hardware implementation efficiency reported in Table. 8.3, for real-time image classification, SP can be a solution if the application has a high error tolerance.

Table 8.9: MNIST Classification Success Rate ($\%$)

	SP	Perfect SP	SC	Fixed point	Floating point
32-bit	69.06	75.63	49.42	91.78	93.12
64-bit	78.41	84.72	60.07	92.62	93.68
128-bit	86.26	90.11	67.15	93.34	93.96
256-bit	90.86	92.74	70.26	93.99	94.11
Avg	81.15	85.81	61.72	92.93	93.71

8.5 Conclusion

A scaled population arithmetic is proposed to improve accuracy compared to classical stochastic computation, by introducing a scaling (exponent) term, along with a population vector. Delays of the scheme are kept low by ensuring each operation uses $\mathcal{O}(1)$ gate delays. The SP arithmetic scheme improves the accuracy of multiplication and addition by $6.5\times$ and $4.0\times$, respectively com-

pared to classical stochastic computing. Its computation is also much faster than classical stochastic computing. We also apply the SP arithmetic scheme on a BMDP problem and improve the accuracy by 31.89% compared to SC. Note that the SP arithmetic scheme can be used as a general purpose computation scheme and is not limited to the BMDP problem. We also apply it on matrix inner production and the MNIST image classification application. In the future, more arithmetic operations will be designed, including subtraction, division and logarithm. A quantified error analysis is provided to further support the theoretical foundation of the SP scheme. Our design incurs $\mathcal{O}(1)$ gate delays, to ensure efficiency.

9. CONCLUSION AND FUTURE WORK

Bayesian Markov Decision Process (BMDP) can be used to solve many practical sequential decision making problems since it captures the uncertainty of the environment. However, due to its exponential runtime and memory storage requirements, its practicability is limited. In this thesis, we aim to solve both the runtime and memory issues of BMDP by developing hardware-oriented acceleration algorithms and scalable compact BMDP model representations.

In our work, we present a GPU-based BMDP computation, a parallel Forward Search Sparse Sampling (FSSS) BMDP algorithm in order to solve the runtime issue. Note that the parallel FSSS-based BMDP can also alleviate the memory utilization. For the memory issue, we present a Duplex Sparse Storage (DSS) scheme and a Binary Decision Diagram (BDD) scheme. We also describe a new Scaled Population (SP) arithmetic to further improve the efficiency and accuracy of the computation in BMDP algorithm. The SP arithmetic can also be used in many other applications in addition to BMDP. Our results show that these techniques can alleviate the runtime and memory problems, which makes the potential of BMDP for solving practical problems more readily realized.

In the future, we plan to expend our work by exploring the followings:

- **A graph BMDP model.** We also plan to develop a neuronal network like graph BMDP model in order to further alleviate the memory requirement. In this graph BMDP model, the data of the expected TPM will be represented underlying the weights of the edges.

- **A ring-based MDP engine.** We plan to design a ring-based MDP circuit engine. The expected TPM data is transferred on a ring-based data bus, which feeds to the computation module. The BMDP result computation will be preformed on levelized computation module. Such hardware-based engine will significantly reduce the runtime.

- **DSS/BDD with FSSS-BMDP.** The DSS and BDD scheme can also be applied on the FSSS-BMDP algorithms. In our current work, we only implement the DSS and BDD with the

GPU-BMDP. However, embedding DSS or BDD with the FSSS-BMDP will help further reduce the runtime and memory storage.

- **Expand the SP operation set.** More arithmetic operations can be supported by the SP arithmetic. For example, division, logarithmic function, etc, need to be explored.

REFERENCES

[1] M. R. Yousefi and E. R. Dougherty, “A comparison study of optimal and suboptimal intervention policies for gene regulatory networks in the presence of uncertainty,” *EURASIP Journal on Bioinformatics and Systems Biology*, vol. 2014, no. 1, p. 6, 2014.

[2] O. Sigaud and O. Buffet, *Markov decision processes in artificial intelligence*. Hoboken, NJ: John Wiley & Sons, 2013.

[3] A. Nilim and L. El Ghaoui, “Robust control of markov decision processes with uncertain transition matrices,” *Operations Research*, vol. 53, no. 5, pp. 780–798, 2005.

[4] O. Berman and E. Kim, “Stochastic models for inventory management at service facilities,” *Stochastic Models*, vol. 15, no. 4, pp. 695–718, 1999.

[5] N. Bäuerle and U. Rieder, *Markov decision processes with applications to finance*. New York, NY: Springer Science & Business Media, 2011.

[6] N. Bäuerle and U. Rieder, “MDP algorithms for portfolio optimization problems in pure jump markets,” *Finance and Stochastics*, vol. 13, no. 4, pp. 591–611, 2009.

[7] N. Friedman and Y. Singer, “Efficient Bayesian parameter estimation in large discrete domains,” in *Proceedings of the 11th International Conference on Neural Information Processing Systems*, pp. 417–423, MIT Press, 1998.

[8] S. Koenig and R. Simmons, “Xavier: A robot navigation architecture based on partially observable markov decision process models,” *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems*, pp. 91–122, 1998.

[9] Y. Aviv and A. Pazgal, “A partially observed markov decision process for dynamic pricing,” *Management Science*, vol. 51, no. 9, pp. 1400–1416, 2005.

[10] R. E. Bellman and S. E. Dreyfus, *Applied dynamic programming*. Princeton, New Jersey: Princeton university press, 2015.

[11] N. Roy and S. Thrun, “Coastal navigation with mobile robots,” in *Proceedings of Advances in Neural Information Processing Systems*, pp. 1043–1049, 2000.

[12] J. Boger, J. Hoey, P. Poupart, C. Boutilier, G. Fernie, and A. Mihailidis, “A planning system based on markov decision processes to guide people with dementia through activities of daily living,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 10, no. 2, pp. 323–333, 2006.

[13] T. J. Walsh, S. Goschin, and M. L. Littman, “Integrating sample-based planning and model-based reinforcement learning.,” in *Association for the Advancement of Artificial Intelligence*, 2010.

[14] M. Hecker, S. Lambeck, S. Toepfer, E. Van Someren, and R. Guthke, “Gene regulatory network inference: data integration in dynamic models - a review,” *Biosystems*, vol. 96, no. 1, pp. 86–103, 2009.

[15] Y. Saad, *Iterative methods for sparse linear systems*. Philadelphia, PA: SIAM, 2003.

[16] B. R. Gaines, “Stochastic computing,” in *Proceedings of the Joint Computer Conference*, pp. 149–156, ACM, 1967.

[17] A. Alaghi, W. Qian, and J. P. Hayes, “The promise and challenge of stochastic computing,” *The IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 37, no. 8, pp. 1515–1531, 2017.

[18] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. Hoboken, NJ: John Wiley & Sons, 2014.

[19] P. J. Schweitzer and A. Seidmann, “Generalized polynomial approximations in markovian decision processes,” *Journal of mathematical analysis and applications*, vol. 110, no. 2, pp. 568–582, 1985.

[20] S. P. Meyn, “The policy iteration algorithm for average reward markov decision processes with general state space,” *IEEE Transactions on Automatic Control*, vol. 42, no. 12, pp. 1663–1680, 1997.

[21] N. Friedman and Y. Singer, "Efficient Bayesian parameter estimation in large discrete domains," in *Advances in neural information processing systems*, pp. 417–423, 1999.

[22] J. J. Martin, *Bayesian decision problems and Markov chains*. Hoboken, NJ: John Wiley & Sons, 1967.

[23] M. Ghavamzadeh, S. Mannor, J. Pineau, A. Tamar, *et al.*, "Bayesian reinforcement learning: A survey," *Foundations and Trends in Machine Learning*, vol. 8, no. 5-6, pp. 359–483, 2015.

[24] C. H. Papadimitriou and J. N. Tsitsiklis, "The complexity of Markov Decision Processes," *Mathematics of operations research*, vol. 12, no. 3, pp. 441–450, 1987.

[25] R. Bellman *et al.*, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[26] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[27] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in neural information processing systems*, pp. 1038–1044, 1996.

[28] H. S. Chang, W. J. Gutjahr, J. Yang, and S. Park, "An ant system approach to Markov Decision Processes," in *Proceedings of the 2004 American Control Conference*, vol. 4, pp. 3820–3825, IEEE, 2004.

[29] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, pp. 1008–1014, 2000.

[30] M. O. Duff, "Monte-carlo algorithms for the improvement of finite-state stochastic controllers: Application to Bayes-adaptive markov decision processes.," in *The International Conference on Artificial Intelligence and Statistics*, 2001.

[31] P. Poupart, N. Vlassis, J. Hoey, and K. Regan, "An analytic solution to discrete bayesian reinforcement learning," in *Proceedings of the 23rd international conference on Machine learning*, pp. 697–704, ACM, 2006.

[32] M. Strens, "A Bayesian framework for reinforcement learning," in *The International Conference on Machine Learning*, pp. 943–950, 2000.

[33] T. Wang, D. Lizotte, M. Bowling, and D. Schuurmans, "Bayesian sparse sampling for on-line reward optimization," in *The International Conference on Machine Learning*, pp. 956–963, 2005.

[34] W. B. Powell, *Approximate Dynamic Programming: Solving the curses of dimensionality*, vol. 703. John Wiley & Sons, 2007.

[35] D. P. Bertsekas and J. N. Tsitsiklis, "Neuro-dynamic programming: an overview," in *Proceedings of the 34th IEEE Conference on Decision and Control*, vol. 1, pp. 560–564, IEEE, 1995.

[36] A. Guez, D. Silver, and P. Dayan, "Efficient Bayes-adaptive reinforcement learning using sample-based search," in *Proceedings of Advances in Neural Information Processing Systems*, pp. 1025–1033, 2012.

[37] R. Fonteneau, L. Busoniu, and R. Munos, "Optimistic planning for belief-augmented markov decision processes," in *2013 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning*, pp. 77–84, IEEE, 2013.

[38] A. P. Jóhannsson, "GPU-based markov decision process solver," *Master's thesis, School of Computer Science, Reykjavík University*, 2009.

[39] P. Chen and L. Lu, "Markov decision process parallel value iteration algorithm on GPU," in *2013 International Conference on Information Science and Computer Applications*, Atlantis Press, 2013.

[40] R. St-Aubin, J. Hoey, and C. Boutilier, "Apricodd: Approximate policy construction using decision diagrams," in *Proceedings of Advances in Neural Information Processing Systems*, pp. 1089–1095, 2001.

[41] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, "SPUDD: Stochastic planning using decision diagrams," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 279–288, Morgan Kaufmann Publishers Inc., 1999.

[42] A. Miner and D. Parker, "Symbolic representations and analysis of large probabilistic systems," in *Validation of Stochastic Systems*, pp. 296–338, Springer, 2004.

[43] M. L. Puterman and M. C. Shin, "Modified policy iteration algorithms for discounted markov decision problems," *Management Science*, vol. 24, no. 11, pp. 1127–1137, 1978.

[44] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. Van Der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Philadelphia, PA: SIAM, 2000.

[45] Y. Saad, "Krylov subspace methods on supercomputers," *Scientific and Statistical Computing*, vol. 10, no. 6, pp. 1200–1232, 1989.

[46] I. Shmulevich, E. R. Dougherty, S. Kim, and W. Zhang, "Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks," *Bioinformatics*, vol. 18, no. 2, pp. 261–274, 2002.

[47] N. F. Y. Singer, "Efficient Bayesian parameter estimation in large discrete domains," in *NIPS*, vol. 11, p. 417, MIT Press, 1999.

[48] J. Asmuth and M. L. Littman, "Approaching Bayes-optimalilty using Monte-carlo tree search," in *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 2011.

[49] A. Guez, N. Heess, D. Silver, and P. Dayan, "Bayes-adaptive simulation-based search with value function approximation," in *NIPS*, pp. 451–459, 2014.

[50] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer graphics forum*, vol. 26, pp. 80–113, Wiley Online Library, 2007.

[51] S. Ross, B. Chaib-draa, and J. Pineau, “Bayes-adaptive POMDPs,” in *NIPS*, pp. 1225–1232, 2007.

[52] “The Nvidia GeForce website,” 2015.

[53] H. Zhou, J. Hu, S. P. Khatri, F. Liu, C. Sze, and M. R. Yousefi, “GPU acceleration for Bayesian control of Mrkovian genetic regulatory networks,” in *Proceedings of the 3rd International Conference on Biomedical and Health Informatic*, pp. 304–307, IEEE, 2016.

[54] I. Shmulevich and E. R. Dougherty, *Probabilistic Boolean networks: the modeling and control of gene regulatory networks*. Philadelphia, PA: SIAM, 2010.

[55] X. Cai, J. A. Bazerque, and G. B. Giannakis, “Inference of gene regulatory networks with sparse structural equation models exploiting genetic perturbations,” *PLOS Computational Biology*, vol. 9, no. 5, p. e1003068, 2013.

[56] J. Tegner, M. K. S. Yeung, J. Hasty, and J. J. Collins, “Reverse engineering gene networks: integrating genetic perturbations with dynamical modeling,” *the National Academy of Sciences*, vol. 100, no. 10, pp. 5944–5949, 2003.

[57] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai, “Lethality and centrality in protein networks,” *Nature*, vol. 411, no. 6833, pp. 41–42, 2001.

[58] D. Thieffry, A. M. Huerta, E. Pérez-Rueda, and J. Collado-Vides, “From specific gene regulation to genomic networks: a global analysis of transcriptional regulation in escherichia coli,” *Bioessays*, vol. 20, no. 5, pp. 433–440, 1998.

[59] “CUDA toolkit documentation v8.0,” 2016.

[60] S. B. Akers, “Binary decision diagrams,” *IEEE Transactions on computers*, no. 6, pp. 509–516, 1978.

[61] R. E. Bryant, “Symbolic Boolean manipulation with ordered Binary Decision Diagrams,” *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[62] V. Lagoon and P. J. Stuckey, "Set domain propagation using ROBDDs," in *International Conference on Principles and Practice of Constraint Programming*, pp. 347–361, Springer, 2004.

[63] S. Joshi, K. Kersting, and R. Khardon, "Decision-theoretic planning with generalized first-order decision diagrams," *Artificial Intelligence*, vol. 175, no. 18, pp. 2198–2222, 2011.

[64] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[65] F. Somenzi, "CUDD: CU decision diagram package release 3.0. 0," *University of Colorado at Boulder*, 2015.

[66] H. Zhou, S. P. Khatri, J. Hu, F. Liu, and C. Sze, "Fast and highly scalable Bayesian MDP on a GPU platform," in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pp. 158–167, 2017.

[67] B. Grigorian, N. Farahpour, and G. Reinman, "Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing," in *HPCA*, pp. 615–626, 2015.

[68] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *ISCA*, pp. 554–566, 2015.

[69] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 505–516, 2014.

[70] Y. Wang, H. Li, and X. Li, "Real-time meets approximate computing: An elastic CNN inference accelerator with adaptive trade-off between QOS and QOR," in *DAC*, pp. 1–6, 2017.

[71] G. Keramidas, C. Kokkala, and I. Stamoulis, "Clumsy value cache: An approximate memoization technique for mobile GPU fragment shaders," in *Workshop on Approximate Computing*, 2015.

[72] A. B. Kahng and S. Kang, “Accuracy-configurable adder for approximate arithmetic designs,” in *DAC*, pp. 820–825, 2012.

[73] J. S. Vetter and S. Mittal, “Opportunities for nonvolatile memory systems in extreme-scale high-performance computing,” *Computing in Science & Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[74] M. Samadi and S. Mahlke, “CPU-GPU collaboration for output quality monitoring,” in *1st Workshop on Approximate Computing Across the System Stack*, pp. 1–3, 2014.

[75] A. Alaghi and J. P. Hayes, “Survey of stochastic computing,” *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 2s, p. 92, 2013.

[76] B. R. Gaines, “Stochastic computing systems,” in *Advances in Information Systems Science*, pp. 37–172, Springer, 1969.

[77] A. Alaghi and J. P. Hayes, “Exploiting correlation in stochastic circuit design,” in *ICCD*, pp. 39–46, 2013.

[78] T.-H. Chen and J. P. Hayes, “Design of division circuits for stochastic computing,” in *ISVLSI*, pp. 116–121, 2016.

[79] A. Alaghi and J. P. Hayes, “A spectral transform approach to stochastic circuits,” in *ICCD*, pp. 315–321, 2012.

[80] S.-J. Min, E.-W. Lee, and S.-I. Chae, “A study on the stochastic computation using the ratio of one pulses and zero pulses,” in *ISCAS*, vol. 6, pp. 471–474, 1994.

[81] Z. Wang, S. Mohajer, and K. Bazargan, “Low latency parallel implementation of traditionally-called stochastic circuits using deterministic shuffling networks,” in *ASPDAC*, pp. 337–342, 2018.

[82] M. Van Daalen, P. Jeavons, J. Shawe-Taylor, and D. Cohen, “Device for generating binary sequences for stochastic computing,” *IEE Electronics Letters*, vol. 29, pp. 80–80, 1993.

[83] M. Ramakrishna, E. Fu, and E. Bahcekapili, “Efficient hardware hashing functions for high performance computers,” *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[84] Y. LeCun, C. Cortes, and C. Burges, “Mnist handwritten digit database,” *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, p. 18, 2010.