International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 2, pages 321–335, July 2014

Optimal Periods for Probing Convergence of Infinite-stage Dynamic Programmings on GPUs

Tsutomu Inamoto
Graduate School of Science and Engineering, Ehime University
3 Bunkyo-cho, Matsuyama, Ehime, Japan 790–8577

Yoshinobu Higami
Graduate School of Science and Engineering, Ehime University
3 Bunkyo-cho, Matsuyama, Ehime, Japan 790–8577

Shin-ya Kobayashi
Graduate School of Science and Engineering, Ehime University
3 Bunkyo-cho, Matsuyama, Ehime, Japan 790–8577

Received: February 15, 2014
Revised: May 3, 2014
Accepted: June 3, 2014
Communicated by Akihiro Fujiwara

Abstract

In this paper, we propose a basic technique to minimize the computational time in executing the infinite-stage dynamic programming (DP) on a GPU. The infinite-stage DP involves computations to probe whether a value function gets sufficiently close to the optimal one. Such computations for probing convergence become obvious when an infinite-stage DP is executed on a GPU, since those computations are not necessary for finite-stage DPs, and hide behind loops for updating state values when a DP is executed on a CPU. The heart of the proposed technique is to suppress those computations for probing by thinning out them. By the proposed technique, differences between state values before and after being updated are periodically transferred to the main memory, then are checked to probe convergence. This intermittent probing makes contrast to ordinary methods in which computations for probing are processed every time. The technique also proposes a formulation to determine optimal periods for probing based on simple statistics given by preliminary experiments. The effectiveness of the proposed technique is examined on two problems; the one is a kind of the animat problem in which an agent moves around in a maze to collect foods, and the other is the mountain-car problem in which a powerless car on a slope struggles to pass over a higher peak. Computational results display that a method with the proposed technique decreases computational times for both problems compared to methods in which computations for probing convergence are processed every time, and the degree of decreasing seems remarkable when the state space is larger.

Keywords: dynamic programming, value iteration, GPU

1 Introduction

The advent of the computer has brought the numerical approach to complicated optimization problems. The non-linear control problem is one of such problems. Its optimal policy can be obtained by

solving the Hamilton-Jacobi-Bellman equation [1]. This equation is formally solvable by applying the dynamic programming (DP as acronym) [2]. The DP is said to be comprised of a sequence of sweeps. Here, a sweep represents a chunk of computations to update state values, each of which represents an estimation of the accumulated cost of the corresponding state. Computations which comprise a sweep are usually simple, so we can expect that a GPU (Graphics Processing Unit), which suits for simple computations, can effectively execute DPs. This expectation is reasonable as far as each sweep, but the DP is a sequence of sweeps. That sequence is finished when state values are probed to converge. Costs of computations to probe convergence are obscure when DPs are executed on CPUs, since computations to calculate the maximum difference between state values before and after a sweep are processed in the same loop to computations for sweeps. In order to decrease the computational time in executing an infinite-stage DP on a GPU, it is necessary to consider not only sweeps as usual but also computations to probe convergence. It is a straightforward technique to not always but periodically process computations to probe convergence.

In this paper, we propose a technique to decrease computational times in executing infinite-stage DPs on GPUs. The technique is to thin out computations to probe convergence; they are periodically processed, whereas sweeps are always processed. A formulation is also proposed that is based on simple statistics and gives optimal periods for probing convergence. The objective of this paper is to display the effectiveness of the proposed technique as to two problems; the one is a kind of the animat problem in which an agent moves around in a maze to collect foods, and the other is the mountain-car problem in which a powerless car on a slope struggles to pass over a higher peak [3].

The paper is comprised as follows. In Sections 2 and 3, the Markov decision process and the DP are respectively outlined. Some general issues in implementing DPs for GPUs by using CUDA (Compute Unified Device Architecture) [4] are stated in Section 4. The proposed technique is described in Section 5. Problem-specific issues in implementing DPs on GPUs and evaluation results as to the aforementioned two problems are displayed in Section 6. Finally, Section 7 summarizes the paper and lists future works.

2 Markov decision process

2.1 Basic formalization

Many systems in real world exhibit stochastic behaviors. In analyzing such behaviors of discrete time systems, it is natural to regard them to take some states for each moment and states' developments to be statistically determined. Such developments are attributed to, if defined stationary, state transition probabilities which of each represents the probability that the system transits from a state to a state. The state at time $t+1$ of a system which is governed by state transition probabilities depends only on information at time t. Such a system is said to have Markov property. The problem of optimally controlling or scheduling a system having Markov property is called Markov decision process (MDP as acronym). The objective of an MDP is to minimize costs (or maximizing rewards) which are accumulated during states' developments.

In order to formalize the MDP concerning to a system having Markov property, let T, $\mathcal{S}$, and $\mathcal{U}$ denote the planning time, the state space, and the decision space, respectively. Here, all spaces are assumed countable. Additionally, let us assume that the system incurs cost $C(s)$ $\left(0 \leq \cdot \leq \bar{C}\right)$ in state s at discrete time $t \in \{1, \ldots, T\}$, here $\bar{C}$ denotes the upper bound of the cost. The sequence $\pi := \langle \mu_1(\cdot), \ldots, \mu_{T-1}(\cdot) \rangle \in \Pi$, which is comprised of functions which of each designates a decision at a certain time, is called policy. The set Π, which is composed of all possible policies, is called policy space. Moreover, let us assume that the probability of transiting from state s to state s' by making decision u is given as $p^{(u)}_{s,s'}$. By using these symbols, the value of policy π starting from state s can be considered as the expectation of the summation of discounted costs, and is represented as follows:

$$F^{(\pi)}_T(s) := \sum_{t=0}^{T-1} \alpha^t \sum_{s' \in \mathcal{S}} \hat{p}^{(\pi)}_{s,s'}(t+1)\, C(s'). \tag{1}$$

In this equation, α $(0 < \cdot \leq 1)$ denotes the discount factor, and $\hat{p}^{(\pi)}_{s,s'}(t)$ denotes the probability of

transiting from state s to state s' within t times by policy π. $\hat{p}^{(\pi)}_{s,s'}(t)$ is recursively represented as follows:

$$\begin{aligned} \hat{p}^{(\pi)}_{s,s'}(1) &:= p^{(\mu_1(s))}_{s,s'}, \\ \hat{p}^{(\pi)}_{s,s'}(t+1) &:= \sum_{s''\in\mathcal{S}} p^{(\mu_{t+1}(s))}_{s,s''} \hat{p}^{(\pi)}_{s'',s'}(t). \end{aligned} \tag{2}$$

The optimal policy which minimizes the objective value displayed in Eq. (1) is given as follows:

$$\pi^* = \arg\min_{\pi\in\Pi} F^{(\pi)}_T(s). \tag{3}$$

When the optimal policy is denoted $\pi^* = \left\langle \mu^*_1(\cdot), \ldots, \mu^*_{T-1}(\cdot) \right\rangle$, the optimal decision in state s at time t is represented as $\mu^*_t(s)$.

2.2 Approximating finite MDPs to infinite MDPs

In general, the optimal decision in the same state may differ if times differ. Thus the spatial cost to hold all decisions over all states is basically proportional to $|\mathcal{S}| \cdot T$. If the state space is large and/or the planning time is long, such spatial cost is so high that the applicable range of the DP diminishes. This analysis motivates to approximate the finite MDP to the infinite MDP by assuming that the system has the ergodic property and $\alpha < 1$, $T \to \infty$. This approximation is adopted in this paper. By this approximation, the policy becomes not to depend on time, and can be represented as a single function $\mu(s)$. Thus policy π and decision rule μ can be interchangeably used for ease of writing.

In the case of finite MDP, the function $J_0(\cdot) \equiv 0$ and the relation (2) make it possible to recursively represent the function of Eq. (1) as follows:

$$\begin{aligned} F^{(\pi)}_0(s) &:= J_0(s), \\ F^{(\pi)}_{t+1}(s) &:= \sum_{s'\in\mathcal{S}} p^{(\mu_{t+1}(s))}_{s,s'} \left(C(s') + \alpha F^{(\pi)}_t(s') \right). \end{aligned} \tag{4}$$

In the case of the infinite MDP, based on Eq. (4), the objective function is represented as follows:

$$F^{(\mu)}(s) := \lim_{T\to\infty} F^{(\mu)}_T(s). \tag{5}$$

2.3 Formalizing state transition probabilities

In previous sections, state transition probabilities of a system are assumed to be given in beforehand. This assumption seems rationale if the behavior of that system can be formalized by rigid equations such as state equations. However, such formalization is quite difficult for many real systems, since their states and state transitions are frequently discrete and/or discontinuous. The state transition probabilities of such system are often numerically approximated by using Monte-Carlo methods which deploy simulation programs [5]. This strategy is supreme in terms of applicability, but not so in terms of computational burden.

Let us assume that the behavior of a considered system is reproducible (i.e. can be simulated). The state and decision of such a system are discrete and finite even if they are coded as floating-point real variables, since those variables are finite binary vectors in nowadays computers. Furthermore, we believe that the elemental fountain which disturbs that system can be evinced with its occurring probability. Such fountain must be discrete and finite in computers, and is called situational input [6]. A situational input is denoted $w \in \mathcal{W}$, where $\mathcal{W}$ is the situational input space. By representing the occurring probability of w as $P(w)$, the state transition probability is formalized as follows [5]:

$$p^{(u)}_{s,s'} = \sum_{w\in\mathcal{W}} P(w)\, \delta\left(f(s,u,w), s'\right). \tag{6}$$

In this equation, $f(s,u,w) \in \mathcal{S}$ denotes the state transition function which represents the state resulted by making decision u under situational input w at state s, and $\delta(\cdot,\cdot) \in \{0,1\}$ denotes

Kronecker's delta function. The state transition function can be represented as a binary vector-valued function by representing state, decision, and situational input as binary vectors, and formalizing the progress of each state variable as a binary function [6]. Thus, it is basically possible, and may be promising if $|\mathcal{W}|$ is small, to formalize state transition probabilities of a reproducible system. So we think that the applicable range of the MDP and the DP is not so narrow.

3 Dynamic programming

3.1 Solutions for DPs

In this paper, a DP for an infinite MDP is called infinite-stage DP. There are three representative solutions for DPs. They are the linear programming, policy iteration [2], and value iteration (VI as acronym). The linear programming is hardly applicable, since such applications require constraint equations which numbers are proportional to $|\mathcal{S}| \cdot |\mathcal{U}|$, where this number is usually enormous. The policy iteration is more applicable than the linear programming, but it is rather complicated. This complication is essentially attributed to the procedure of the policy iteration that it updates both the value and tentatively-optimal decision for each state. On the other hand, the VI is rather simpler than the policy iteration. It only repeats sweeps until a certain terminal condition is satisfied. These observations lead us to focus on the VI, since its simpleness may assure its suitability for being processed in parallel. Thus, in this paper, we select only the VI. Furthermore, the word "DP" and "VI" are sometimes used interchangeably, and a VI for an infinite MDP is called infinite-stage DP for ease of writing.

3.2 Infinite-stage DP

In this section, the infinite-stage DP (VI in concrete) is formalized. At first, let us define two mappings, $\mathcal{M}$ and $\mathcal{M}_\mu$, as follows [5]:

$$\mathcal{M}(J)(s) := \min_{u \in \mathcal{U}_s} \sum_{s' \in \mathcal{S}} p_{s,s'}^{(u)} \left(C(s') + \alpha J(s') \right), \tag{7}$$

$$\mathcal{M}_\mu(J)(s) := \sum_{s' \in \mathcal{S}} p_{s,s'}^{(\mu(s))} \left(C(s') + \alpha J(s') \right). \tag{8}$$

Here, the function $J(\cdot)$ denotes the value function which designates values of states, and $\mathcal{U}_s \subseteq \mathcal{U}$ denotes the set of decisions which can be taken in state s. By using the latter mapping, the value function $J^{(\mu)}(\cdot)$ which gives the objective value of decision rule μ in the infinite MDP can be represented as follows:

$$J^{(\mu)}(s) := F^{(\mu)}(s) = \lim_{T \to \infty} \mathcal{M}_\mu^T (J_0)(s). \tag{9}$$

If $\mu^*(\cdot)$ denotes the optimal decision rule, then the optimal value function $J^*(\cdot)$ is defined as follows:

$$J^*(s) := F^{(\mu^*)}(s). \tag{10}$$

The optimal value function satisfies the Bellman equation displayed in Eq. (11), and is obtained by Eq. (12).

$$J^*(s) = \mathcal{M}(J^*)(s). \tag{11}$$

$$J^*(s) = \lim_{k \to \infty} \mathcal{M}^k (J_0)(s). \tag{12}$$

Equation (12) is realized by the repetitive calculation displayed in Eq. (13) [2].

$$\begin{aligned} \underline{J}_0(s) &:= 0, \\ \underline{J}_{k+1}(s) &:= \mathcal{M}(\underline{J}_k)(s) \ \ (k \geq 0). \end{aligned} \tag{13}$$

Finally, the procedure displayed in Eq. (13) is called value iteration.

3.3 Terminal conditions of infinite-stage DPs

In theoretical, the infinite-stage DP is never finished and the procedure of Eq. (13) is iterated forever. In practical, the allowed error between a tentative value and optimal one of a state is specified by a decision maker, and the iteration is terminated when the maximum difference between state values before and after a sweep becomes less or equal to a value which is calculated from that error.

Let us assume that all state values were updated $k+1$ times and yielded value function $\underline{J}_{k+1}(\cdot)$. If the relation (14) is satisfied, then the difference between the objective value obtained by obeying function $\underline{J}_{k+1}(\cdot)$ and the optimal value becomes less or equal to ϵ; that is, relation (15) is satisfied [5].

$$\max_{s\in\mathcal{S}} \left|\underline{J}_{k+1}(s) - \underline{J}_k(s)\right| \leq R(\epsilon) := \frac{(1-\alpha)\,e}{2\alpha}. \tag{14}$$

$$\max_{s\in\mathcal{S}} \left|F^{(\mu)}(s) - J^*(s)\right| \leq \epsilon. \tag{15}$$

Since $|\underline{J}_{k+1}(s) - \underline{J}_k(s)| \leq \alpha^k \bar{C}$, an iteration count $I(\alpha,\epsilon)$ sufficient to satisfy Eq. (15) is:

$$I(\alpha,\epsilon) := \left\lceil \frac{\log((1-\alpha)\,\epsilon) - \log\left(2\alpha\bar{C}\right)}{\log\alpha} \right\rceil, \tag{16}$$

here, $\lceil r \rceil$ is the minimum integer greater or equal to r.

In summary, there are two conditions to prove that the infinite-stage DP has converged within an allowed error ϵ.

- To check whether the maximum differences between state values before and after a sweep are less or equal to $R(\epsilon)$.
- To iterate sweeps for $I(\alpha,\epsilon)$ times.

$I(\alpha,\epsilon)$ is quite large if α is close to 1, so the former condition is usually adopted when an infinite-stage DP is executed on a CPU.

4 Implementing DPs for GPUs

4.1 Applicability of GPUs to DPs

In this paper, we concentrate on the infinite-stage DP. The primary computations of that method is to update state values, or sweeps. One of those computations for state s is conducted along the following steps.

(1) To calculate states succeeding to s.

(2) To fetch values of those succeeding states.

(3) To calculate a new value according to values of succeeding states.

(4) To store that value as the new value of state s.

Although steps (2) and (4) can collide with accesses by computations for other states, those steps are basically independent from each other, thus the infinite-stage DP seems to appropriate for being parallelized [7]. In addition to this parallelism, the number of states is numerous and the computational burden in those steps are so light that a simple core in a GPU can easily conduct them. These characteristics make us expect that the infinite-stage DP is effectively solvable by deploying GPUs. This expectation has been confirmed as far as a simple infinite-stage MDP in [8, 9].

4.2 How to parallelize DPs on GPUs

Let us call the chunk of instructions assigned to a GPU, which has many cores, kernel as like the CUDA [4] which is explained lately at Section 4.4. A kernel has some index parameters and reads and/or writes contents of the memory equipped with the corresponding GPU according to those parameters. Let us call a kernel with specific values of those parameters kernel instance. A kernel instance is given to and executed in a core in a GPU. Values of those parameters are controlled by the scheduler in a GPU for different cores to take different values. Thus, it is possible to parallelize a whole kernel by dividing it as kernel instances then executing them on cores.

In parallelizing DPs on GPUs, the kernel and the index parameters correspond to Eq. (13) and state s in that equation, respectively [8, 9]. That is, a sweep is divided into computations each of which is responsible to update the value of the corresponding state, and those computations are executed in cores in a GPU in parallel. Here, it is possible to define a mapping between the state and the index parameters, since the state space is assumed countable and the index parameters are discrete. For an example based on a source chunk in [4], if the state is two dimensional and defined by two variables of $\langle \mathrm{x1}, \mathrm{x2} \rangle$, the state and those index parameters correspond as follows:

$$\begin{aligned} \mathrm{x1} &= \mathrm{blockIdx.x} * \mathrm{blockDim.x} + \mathrm{threadIdx.x}, \\ \mathrm{x2} &= \mathrm{blockIdx.y} * \mathrm{blockDim.y} + \mathrm{threadIdx.y}, \end{aligned} \tag{17}$$

where `blockIdx`, `blockDim` and `threadIdx` are structured variables embedded in CUDA as mentioned lately at Section 4.4. Divided computations are too enormous for human to design their assignments to cores, thus are automatically assigned by schedulers in a GPU.

4.3 Relationship to existing researches

A research which aims to accelerate a DP by a GPU can be characterized according whether the considered problem is finite or infinite. There are plenty of researches dedicated to finite MDPs. Most of them aim to decrease the computational time in executing Smith-Waterman algorithm [10, 11]. On the other hand, there may be quite a few researches dedicated to infinite MDPs. A research which handles a problem that an agent moves around in a plane to collect foods [8] is one of such researches.

Regardless of the type of considered MDPs, most of existing researches agree to their aim. It is to decrease the computational time required in processing a sweep, and the effectiveness of those researches depends on how to employ machineries equipped with GPUs. The technique of this paper differs from those researches in the objective; the technique aims to decrease the computational time for processing a sequence of sweeps, and does not make an effort to employ GPU machineries. The technique can be combined with other techniques of existing researches, thus the proposed technique is not a competitor with but a complementary to them.

4.4 CUDA

In this paper, we utilize NVIDIA Corporation's CUDA [4] to implement infinite-stage DPs for GPUs. The detailed specification of CUDA is controlled by a kind of version number called *compute capability*. The contents of this section concern to compute capability 3.0 due to the used GPU mentioned at Section 6.4 in later.

The features of CUDA can be viewed from a hardware-related and a computation-managing standpoints. It is said from the former standpoint that the minimum processing unit is *CUDA core*, and the minimum unit in assigning jobs is *Streaming Multiprocessor* (SM as acronym), which is comprised of many CUDA cores. A CUDA core can access the device memories called *register*, *shared memory*, *constant memory*, *texture memory*, and *global memory* [4]. These memories have a trade-off between the accessing speed and the size. For example, the register is lowest in latency but smallest in size, whereas the global memory is highest in latency but largest in size. This trade-off can be detoured by using techniques such as the vector-access to shared memories in SMs, the cache of constant and texture memories, and so on.

From the computation-managing standpoint, the minimum job, the minimum group of jobs, and the minimum unit in assigning jobs are *thread*, *warp*, and *thread block*. Here, 32 threads comprise one warp, and some warps comprise one thread block.

In implementing programs for GPUs by CUDA, the following two points are to be designed: what is the computational unit which comprises the whole computation, and how threads compose a thread block. The computational unit is desired to be sufficiently small like single addition so as to be executed on a CUDA core. The latter design dominates the way by which the whole computation is divided into thread blocks. The dimension of the thread block is accessible from a thread via the built-in variable `blockDim`. Each thread block is assigned to an idle SM. A thread block for an SM is divided into some warps, and threads in each warp are executed on CUDA cores of the corresponding SM. In this phase, threads in a same warp are executed in a concurrent style (Single-Instruction, Multiple-Threads; SIMT) [4], and may emit some *coalesced* accesses to device memories when accessed addresses are properly aligned. When a thread is executed in a CUDA core, the index of the thread block which contains that thread is accessible via the built-in variable `blockIdx`. As like, the index of a thread in a corresponding thread block is accessible via the built-in variable `threadIdx`. The absolute index of a thread becomes available by using those three built-in variables. An image of a case of two dimensionality is displayed in Fig. 6 of [4].

By employing those aforementioned machineries, an infinite-stage DP can be highly accelerated as far as a simple problem [9]. However, such simple problem is not usual in real world, thus a methodology applicable to more complicated problems is necessary. In this sense, the technique proposed in this paper seems to have a certain degree of significance, as it does not highly depend on aforementioned machineries and is applicable to any problem.

5 Proposed technique

5.1 Key idea and algorithm

When an infinite-stage DP is executed on a CPU in serial, there is only one loop over state space. In that loop, computations of updating state values and those of calculating the maximum difference between old and new state values are both involved as displayed in Fig. 1. Computations in that loop are divided into two loops in a straightforward implementation of infinite-stage DPs [8] as displayed in Fig. 2: the one is to update state values and executed on a GPU, and the other is to calculate the maximum difference and executed on a CPU.

In Fig. 2, the loop on a GPU has to be conducted for every time, whereas the loop on a CPU is not necessary. Let us assume that the latter loop is conducted for each m iterations of the former loop as displayed in Fig. 3. This figure displays also the algorithm of the infinite-stage DP which employs the proposed technique.

If we know that the convergence is attained after iterating the former loop for n times, then letting $m := n$ is obviously optimal. However, it is undoubtedly impossible to know n before an optimal result is obtained. If m is too short than n, then iterations of the latter loop are too many than optimal. On the other hand, if m is too long than n, then $m - n$ iterations of the former loop are fruitless. The images of these situations are displayed in Fig. 4. This figure implies that there may be optimal values of m.

The proposed technique is to conduct two types of loops as displayed in Fig. 3 by using proper periods for probing convergence. That is, a probing loop on a CPU is conducted when $i \pmod{m} = 0$, where i denotes the iteration count of sweeps on a GPU. The technique is quite simple, but there is no DP which employs the same technique as far as we know. The reason of this scarcity seems to lay on the weak interest on executing infinite-stage DPs on GPUs, and such weakness may be attributed to the difficulty called "curse of dimensionality," which is crucial to DPs. We believe that such interest grows in some day by technological developments like which enable to create super computers deploying GPUs.

There can be another implementation in which the loop for calculating maximum differences and that for proving convergence are both conducted on a GPU. Investigation on that implementation

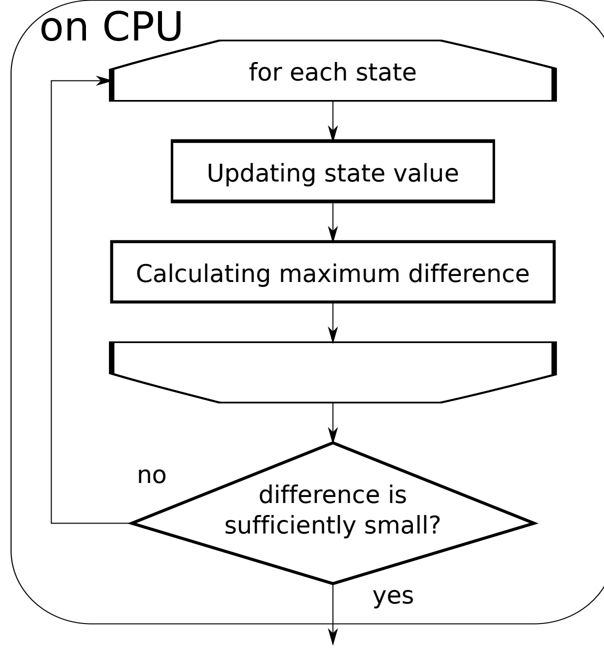

Figure 1: A sketch of an algorithm of executing one loop on a CPU.

is treated as one of future works, since its existence does not harm the applicability of the proposed technique.

5.2 Optimal periods for probing convergence

In this section, the optimal value of m mentioned in Section 5.1 is considered. Let us denote a chunk of computations consist of m sweeps and computations of those for probing convergence "turn." If the minimum iteration of sweeps is known as n, the number of turns is roughly $\lceil n/m \rceil$. When the computational time of executing one sweep on a GPU, the communication time of transferring difference values from a GPU to a CPU, and the computational time of calculating a maximum difference over all states on a CPU are respectively given as t^{V}, t^{D}, and t^{M}, the computational time for each turn is represented as $t^{\mathrm{C}} + mt^{\mathrm{V}}$. Here, the computational time for probing convergence is denoted as t^{C} and defined as $t^{\mathrm{C}} := t^{\mathrm{D}} + t^{\mathrm{M}}$.

By multiplying the former count by the latter times, the whole computational time by the proposed technique is roughly estimated as follows:

$$g\left(m\right) := \left\lceil \frac{n}{m} \right\rceil \left(t^{\mathrm{C}} + m\ t^{\mathrm{V}}\right) \simeq \left(\frac{n+m}{m}\right) \left(t^{\mathrm{C}} + m\ t^{\mathrm{V}}\right). \tag{18}$$

By letting $a := nt^{\mathrm{C}}/t^{\mathrm{V}}$ and arranging Eq. (18), it is revealed that the right hand side of that equation achieves the minimum with:

$$m = \sqrt{a} = \sqrt{n \frac{t^{\mathrm{C}}}{t^{\mathrm{V}}}}. \tag{19}$$

This value in Eq. (19) is not available, since n is unknown until a corresponding problem is solved. An alternative of m, which is denoted m', is possible by substituting $I\left(\alpha, \epsilon\right)$ for n in Eq. (19) as follows:

$$m' = \sqrt{I\left(\alpha, \epsilon\right) \frac{t^{\mathrm{C}}}{t^{\mathrm{V}}}}. \tag{20}$$

m' given by Eq. (20) is no more than an approximation of m, but the value is expected to have some significance, since n approaches to $I\left(\alpha, \epsilon\right)$ when α is close to 1.

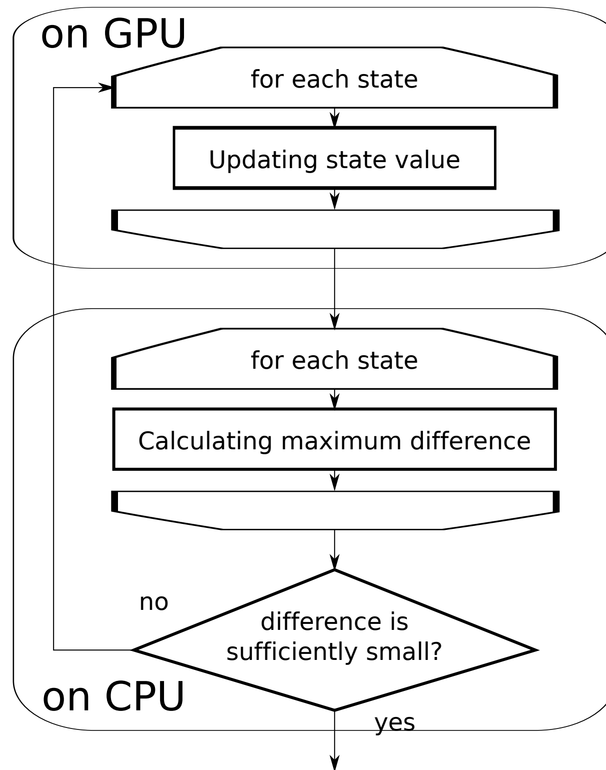

Figure 2: A sketch of an algorithm of executing two loops on a GPU and CPU.

Table 1: Compared methods.

Method	Deploying GPU	Employing the proposed technique
VI	no	no
VI-D	yes	no
VI-P	yes	yes

6 Evaluation

6.1 Compared methods

In this section, the effectiveness of the proposed technique is displayed by comparing some infinite-stage DPs. Those methods are the ordinary VI executed on a CPU, the one executed by deploying a GPU, and another one which deploying a GPU and the proposed technique. They correspond to algorithms in Figs 1, 2, and 3, and called VI, VI-D, and VI-P as Table 1, respectively.

6.2 Target problems

The aforementioned three methods are applied to two MDPs; the one is a kind of animat problem, and the other is the mountain-car problem which is a typical in the discipline of the reinforcement learning [12].

6.2.1 Animat problem

This problem has appeared in [8] at first. The objective of that problem is to control an agent so as to maximally collect foods in a square plane which is surrounded by walls. That plane is slippery as the agent moves to its intended direction (east, south, west, or north) with probability 0.7, or moves

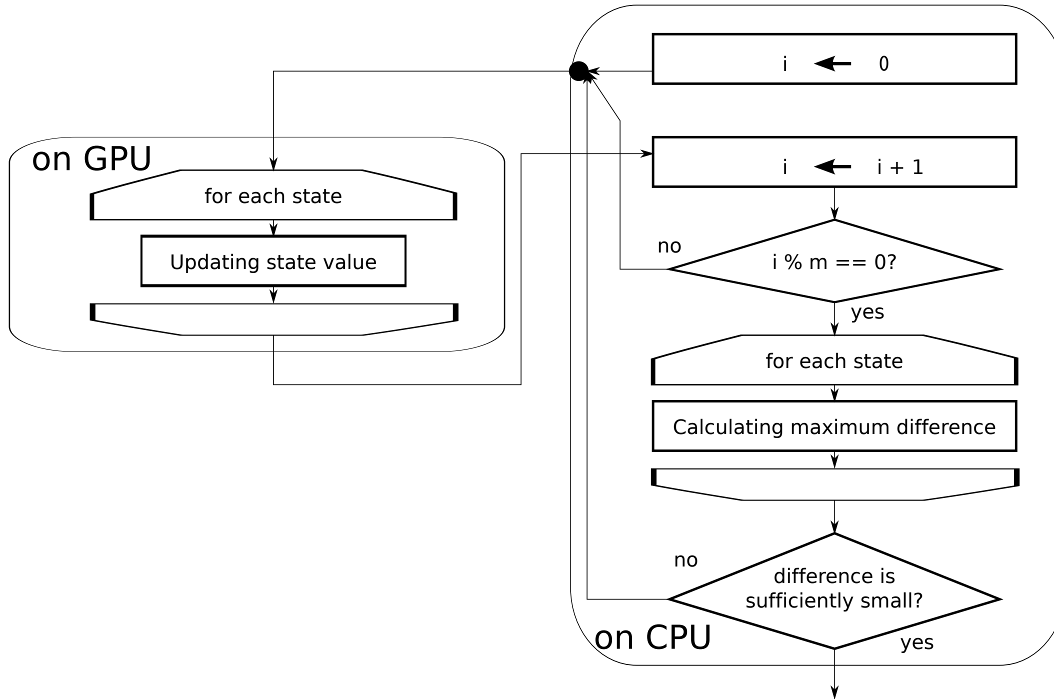

Figure 3: The image of periodically probing convergence.

to one of other directions with probability 0.3. This image is illustrated in Fig. 5. The parameters of this problem are M, M^{f}, $\underline{r}$, and $\bar{r}$. They denote the width (and height) of the plane, the number of foods, the minimum and maximum value of foods, respectively.

This problem is so elementary that the application of CUDA has achieved about 180x speedups by deploying the shared memory [9].

6.2.2 Mountain-car problem

This problem is more realistic than the animat problem. The objective of this problem is to make a car on a slope pass over the higher peak. The sketch of the problem is displayed in Fig. 6. The car is too powerless to directly climb the steeper slope, so has to swing between the lower and higher hills [3]. This problem is a typical of researches which aim to improve the applicability of the reinforcement learning [12] to continuous state space, since the position and velocity of the car are intrinsically real.

In this paper, that problem is simply handled as a discrete problem by scaling up the position and velocity then discretizing them. The scaling factor Z is the only parameter of that problem.

6.3 Detailed issues in implementation

As mentioned in Section 5.1, computations of the infinite-stage DP are divided into two kinds: those to update state values, and those to calculate maximum differences. In short, the proposed technique aims to omit the latter computations, and does not care the effectiveness of computations of both kinds. It is expected that a contrivance on the implementation of the former computations such as employing the shared memory and/or adjusting the design of the thread block will make computational times for those computations shorter and make the effectiveness of the proposed technique more remarkable. Such contrivance is not adopted in this paper, since it depends on applied problems and may harm the claim that the proposed technique is applicable to any problem.

In concrete, any of the constant memory, texture memory, and shared memory is not used, and only the global memory is used. The global memory holds three 1-D array of single floating-point

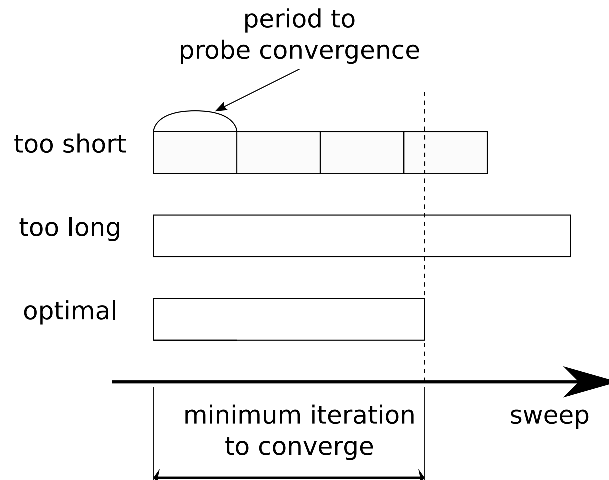

Figure 4: The images of values of m which are too short, too long, and optimal.

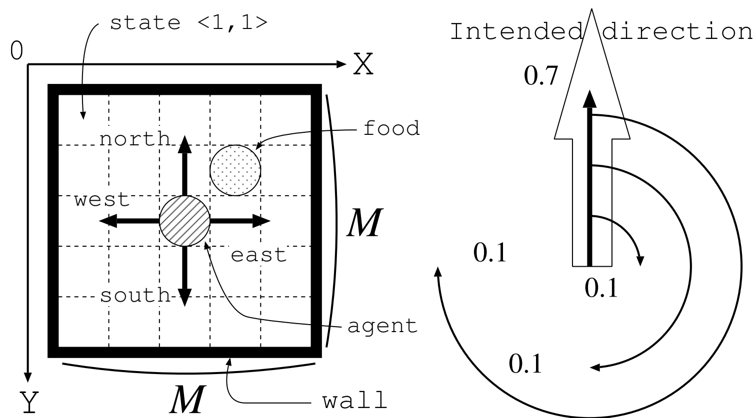

Figure 5: The slippery plane and the stochastic behavior of the agent.

real variables for both problems. Those arrays are (1) old state values before updated, (2) new state values after updated, and (3) their differences, each of which is computed by subtracting new value of a state from its old value. In the animat problem, the occurring probabilities of situational inputs are also held as 1-D array. The design of the thread block is not examined and set to a moderate one for both problems according to previous results [9].

The state defined by dimensions x_1 and x_2 is converted to the index in 1-D array by the following equation:

$$(x_2 - \underline{x_2}) \cdot \left(\bar{x_1} - \underline{x_1}\right) + x_1 - \underline{x_1}, \tag{21}$$

where $\underline{x}$ and $\bar{x}$ denote the lower and upper bound of dimension x, respectively. x_1 and x_2 in Eq. (21) correspond to the x- and y-coordinates of the agent in the animat problem, and the position and the velocity of the car in the mountain-car problem, respectively.

6.4 Parameter setting

The setting of parameters which determine the scales of the problems etc. are displayed in Table 2. They are basically same to those in [8] and [3]. If a real value is held as a single floating-point real variable (which consumes 4 bytes), the parameter setting results that, at least, 285 MBytes and 12 MBytes of device memories are consumed in the mountain-car and animat problem, respectively.

A computer with a GPU of GeForce GTX680 and a CPU of Intel Core i7 3.4 GHz was used

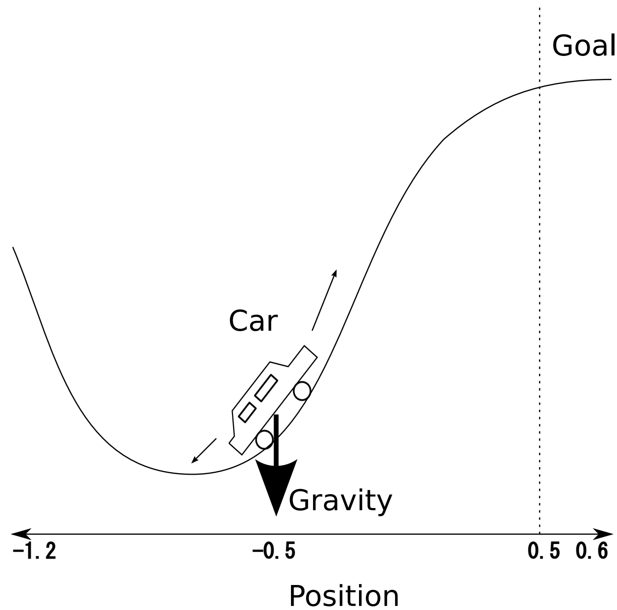

Figure 6: The sketch of the mountain-car problem.

Table 2: Parameter setting.

Problem	Parameter	Value
—	ϵ	0.0001
	Block dimension	64x × 1y
animat	α	0.9
	M	1,024
	M^{f}	1,024
	$\underline{r}$, $\bar{r}$	2, 20
mountain-car	α	0.99
	Z	10,000

to run programs which had implemented the three methods. That GPU has 8 SMs, each of which contains 192 CUDA cores [13], and a device memory of 4 GBytes. The size of the main memory of the computer was 32 GBytes. CUDA Toolkit version 5 was used in implementing programs, here it is noted that this version is like the version of the SDK tool and independent of the compute capability.

6.5 Computational results

A preliminary experiment was conducted to obtain proper m and/or m' for each problem. That experiment was comprised of 100 runs of programs. For each run, only one sweep was conducted and computations for probing convergence were processed only once, thus both t^{V} and t^{C} were measured. Basic statistics on t^{V} and t^{C} obtained by 100 runs are displayed in Table 3. t^{C} is comprised of t^{D} and t^{M}, where t^{D} denotes a communication time to transfer difference values from a global memory to a CPU memory, and t^{M} denotes a computation time for calculating maximum differences. Statistics on those two metrics are displayed in Table 4.

Values m' calculated from statistics in Table 3 are displayed in Table 5. This table also displays values of n, which become known after the primary experiment is conducted, and values of m, which become available by using values of n. The primary experiment is comprised of running programs 10 times for each period setting, where a period setting is defined as a composition of the target problem, the used method, and the value of m or m'. The reason why multiple period settings which

Table 3: Minimum, average, and maximum values of t^{V} and t^{C} [msec].

Problem	t^{V}			t^{C}		
	Min.	Avg.	Max.	Min.	Avg.	Max.
animat	12.101	12.1845	12.287	4.461	4.5764	4.959
mountain-car	39.961	40.1026	40.809	1,216.66	1,232.44	1,252.82

Table 4: Minimum, average, and maximum values of t^{D} and t^{M} [msec].

Problem	t^{D}			t^{M}		
	Min.	Avg.	Max.	Min.	Avg.	Max.
animat	0.659	0.6869	0.72	3.787	3.8895	4.292
mountain-car	34.784	39.8168	44.991	1,178.03	1,192.62	1,210.37

are same except for m or m' are considered is to claim that the proposed technique is valid even when the preliminary experiment brings inappropriate t^{C} and/or t^{V}.

Minimum, average, and maximum values of m and m' in Table 5 were calculated by using Eqs. (19) and (20), where values t^{C} and t^{V} were selected from 100 results for each problem as displayed in Table 6. In this table, the left-most column represents types of m or m', and $t_i^{\mathrm{C}}, t_i^{\mathrm{V}}$ $(i = 1, \dots, 100)$ represent the computational times of one sweep and that of calculating the maximum difference of i-th run in the preliminary experiment. For example, the minimum m for the mountain-car problem is 58 as displayed in Table 5. This value results by substituting 112, $\min_{i=1,\dots,100} t_i^{\mathrm{C}} =$ 1,216.66, and $\max_{i=1,\dots,100} t_i^{\mathrm{V}} = 40.809$ for n, t^{C} and t^{V} in Eq. (19), respectively.

The computational times of three programs which implement three methods in Section 6.1 are displayed in Table 7. Here, those times are averaged over 10 runs for each parameter setting, and the column "Period" represents the period for probing convergence, thus contains values of m or m' displayed in Table 5.

We can see the following points from Tables 3 and 7:

1. By comparing VI and VI-D, it is observed that computational times of executing the infinite-stage DP can be decreased by using a GPU.

2. By the maximum computational times (i.e. $\max_{i=1,\dots,100}(t_i^{\mathrm{C}} + t_i^{\mathrm{V}})$) for the preliminary experiment are 0.017089 and 1.29363 [sec] for the animat and mountain-car problems, respectively.

3. By comparing VI-P and VI-D, it is said that the proposed technique can decrease computational times for both problems. If computational times of the preliminary experiment are disregarded, the speedup ratios brought by VI-P against VI-D are about 125.487% and 1,402.42% on the animat and mountain-car problems, respectively. If computational times of the preliminary experiment are regarded, those ratios become 124.527% and 1,243.87%. Here, computational times of the preliminary experiment are, as mentioned before, 0.017089 and 1.29363 [sec] for the animat and mountain-car problems.

4. As to the mountain-car problem, the proposed technique is more effective when the probing period is 58 or 59 than when it is 207, 211, or 213.

Table 5: Numbers of minimum sweeps and values of m and m'.

Problem	n	m			$I(\alpha, \epsilon)$	m'		
		Min.	Avg.	Max.		Min.	Avg.	Max.
animat	114.3	6	7	7	144	7	7	8
mountain-car	112	58	59	59	1,443	207	211	213

Table 6: Combinations of t^{C} and t^{V} values used in calculating minimum, average, and maximum m and m'.

Type of m or m'	Statistics on t^{C} and t^{V}	
	t^{C}	t^{V}
Min.	$\min_{i=1,\ldots,100} t_i^{\mathrm{C}}$	$\max_{i=1,\ldots,100} t_i^{\mathrm{V}}$
Avg.	$\sum_{i=1}^{100} t_i^{\mathrm{C}}/100$	$\sum_{i=1}^{100} t_i^{\mathrm{V}}/100$
Max.	$\max_{i=1,\ldots,100} t_i^{\mathrm{C}}$	$\min_{i=1,\ldots,100} t_i^{\mathrm{V}}$

Table 7: Computational times.

Problem	Method	Period	Time [sec]
animat	VI	-	39.7937
	VI-D		2.78103
	VI-P	6	2.52461
		7	2.2218
		8	2.21622
mountain-car	VI	-	936.367
	VI-D		142.325
	VI-P	58	7.82148
		59	7.91743
		207	9.91896
		211	10.078
		213	10.1489

The third point seems to indicate robustness of the proposed technique against fluctuations of t^{C} and t^{V} measured in the preliminary experiment, since computational times by VI-P are shorter than those by VI-D even if values of m' are far from average. The fourth point can be interpreted that in the case when the probing period is given according to n, the minimum iteration count of sweeps, the proposed technique is more effective than the case when that period is given according to $I(\alpha, \epsilon)$, an approximation of n. However, this interpretation is not valid as to the animat problem. This reason may lay on such difference found in Table 3 that the computational time of t^{V} is longer than that of t^{C} on the animat problem, whereas they are vice versa on the mountain-car problem. Further researches are necessary to investigate on effects brought by that difference.

7 Conclusion

In this paper, we proposed a technique to decrease computational times in executing infinite-stage DPs on GPUs. The technique was inspired by such notice that calculating maximum differences between state values before and after a sweep is costly when a DP is executed on a GPU, since that computation can not be processed in the same loop to a sweep. The proposed technique is to process sweeps in a GPU every time, whereas to process computations for probing convergence in a CPU periodically. In order to specify proper periods for probing, a simple formulation was deduced. This formulation can bring optimal periods if computational times of a sweep, those of computations for probing, and the minimum sweep count are available. In computer illustrations, those two times were measured by a preliminary experiment, and minimum sweep counts were replaced by maximum sweep counts. Three methods: an infinite-stage DP on a CPU, that on a GPU, and that employing the proposed technique, were compared as to the animat problem and the mountain-car problem. In executing programs which had implemented the proposed technique, some periods were considered so as to claim the insensitivity of that technique on results by the

preliminary experiment. Computational results indicated that the proposed technique can decrease computational times of an infinite-stage DP if it is executed on a GPU, and this tendency holds even if results by a preliminary experiment have been inaccurate and resultant periods were far from average. The speedup ratios brought by the proposed technique against the infinite-stage DP on a GPU were about 125% and 1,244% on the animat problem and the mountain-car problem, respectively.

Future works include to consider implementations in which maximum differences are calculated on GPUs, to think out more proper substitution for the minimum sweep count n than $I(\alpha, \epsilon)$, to verify the proposed method for other programs including those with higher dimensionality, and to employ machinery such as the shared memory.

References

[1] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, 2000.

[2] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. PRENTICE-HALL, Inc., Englewood Cliffs, N.J. 07632, 1987.

[3] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

[4] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, 6.0 edition, 2014.

[5] Warren B. Powell. *Approximate Dynamic Programming*. John Wiley & Sons, Inc., 2007.

[6] Tsutomu Inamoto, Hisashi Tamaki, and Hajime Murao. Dynamic programming on reduced models and its evaluation through its application to elevator operation problems. *SICE JCMSI*, 2(4):213–221, 2009.

[7] Dimitri P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, AC-27(3):610–616, 1982.

[8] Àrsæll Pòr Jòhannsson. GPU-based Markov decision process solver. Master's thesis, School of Computer Science, Reykjavìk University, 2009.

[9] Tsutomu Inamoto, Takuya Matsumoto, Chikara Ohta, Hisashi Tamaki, and Hajime Murao. An implementation of dynamic programming for many-core computers. In *Proceedings of SICE Annual Conference 2011 (DVD-Paper)*, pages 961–966, Waseda University, Japan, September 2011.

[10] Ali Khajeh-Saeed, Stephen Poole, and J. Blair Perot. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*, 229:4247–4258, 2010.

[11] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 73(2), 2009.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[13] *GeForce GTX 680 Whitepaper*. `http://www.nvidia.es/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf`.