無限期間動的計画法のGPU実装における収束判定の処理時間削減に向けた検討

稲元 勉[1,a)]　樋上 喜信[1]　小林 真也[1]

概要：本稿では，無限期間マルコフ決定過程のための動的計画法を GPU 上で実行する際の計算時間を削減するため，動的計画法が収束したか否かを判定する処理を効率化する技法を提案し，小規模な問題に対する評価結果を示すことを目的とする．提案技法は，GPGPU の枠組みとして NVIDIA 社の CUDA を用いることを前提とし，状態価値の変化量の最大値が閾値以下であるか否かとして収束判定を行う際に，GPU 側で部分状態空間に対応するスレッドブロック単位で変化量の最大値を算出しておき，この値の全部分状態空間にわたる最大値をホスト側で算出するという素朴なものである．計算結果では，animat 問題，mountain-car 問題を対象とし，提案技法を用いた場合の計算時間を示す．

キーワード：動的計画法，価値反復法，GPGPU, CUDA

Consideration to Decrease Computational Times in Verifying Convergence in GPU Implementation of Infinite-stage Dynamic Programming

Inamoto Tsutomu[1,a)]　Higami Yoshinobu[1]　Shin-ya Kobayashi[1]

Abstract: In this paper, we propose a technique to decrease the computational time of the value iteration program which is implemented on a GPU to solve infinite-stage Markov decision process. In order to judge whether the value iteration has been converged or not, the difference between state values before and after a sweep, which represents the computation to update all state values, is calculated for each state, then the the maximum of such differences over the whole state space is calculated. Here, it is obvious that taking such maximum over whole state space is equal to taking the maximum over values each of which is the maximum difference of state values over a sub state space. The proposed technique premises to use the CUDA to implement the value iteration for NVIDIA's GPU, and considers a thread block in the CUDA as a sub state space, then conducts computations to take the maximum difference of a sub state space in a corresponding thread block on a GPU. The effectiveness of the proposed technique is examined in terms of computational times through its applications to the animat problem and the mountain-car problem.

Keywords: dynamic programming, value iteration, GPGPU, CUDA

1. はじめに

現実に制御／計画の対象となるシステムの多くは，連続時間上で連続的状態をとり，連続的決定を受けて振舞う連続的システムである．本来は連続的であっても，制御／計画がディジタルプロセッサによって行われるならば，制御／計画の対象となるシステムは離散時刻上で離散的状態をとり，離散的決定を受けて振舞う離散的システムとして扱われることになる．このような離散的システムは，複数の時間にわたるシステムの属性を含めるなど十分細かく状態を定義すれば，マルコフ性を有するようモデル化できる．

[1] 愛媛大学大学院理工学研究科
Graduate School of Science and Engineering, Ehime University, Bunkyo-cho 3, Matsuyama 790–8577, Japan
[a)] inamoto@ehime-u.ac.jp

ⓒ 2016 Information Processing Society of Japan

マルコフ性を有するシステムの最適な制御／計画を求める問題は，決定を下す回数が有限ならば有限期間マルコフ決定過程，無限あるいは不定ならば無限期間マルコフ決定過程と呼ばれる [1].

マルコフ決定過程 (Markov Decision Process; 以降 MDP) は，状態空間がある程度小さく，かつ状態遷移を計算機上で再現できる (reproducible) ならば，動的計画法 (Dynamic Programming; 以降 DP) により解くことができる [1]．状態を，状態にて選択する決定を与える関数へと写す関数は，方策と呼ばれる [1]．MDP における目的がコストの最小化であると前提したとき，ある状態を，その状態以降所定の方策にしたがって決定を選択し続けた場合の累積コストの期待値へと移す関数は，状態価値関数と呼ばれる [1], [2]．DP は，最適な状態価値関数を求める手法である．そのための具体的手法として，何らかの暫定方策を仮定し，それを用いた場合の状態価値関数の更新，および更新された状態価値関数に基づく暫定方策の更新を繰り返す方策反復法と，暫定方策をおかずに状態価値の更新を繰り返す価値反復法 (Value Iteration; 以降 VI) の 2 つが挙げられる．これらのうち，とくに後者の VI を構成する計算式は単純であり，かつ状態間で独立に行える [3]．この特徴から，GPU の活用により VI を効率化できると期待できる [4], [5].

以上の期待から，著者らは，GPU の活用により VI の計算時間を削減する取り組みを進めている．その一環として，扱いやすい MDP として，餌の置かれた仮想平面内でエージェントが摂取する餌の量を最大化する方策を求める animat 問題 [5]，低い丘と高い丘のあいだの谷に置かれた車が勢いをつけて高い丘を乗り越える方策を求める mountain-car 問題 [6] をとりあげ，それらの問題のための VI を GPU 実装し，求解時間を削減できることを示した [7]．それらの MDP は無限期間であり，そのための無限期間 VI は，状態価値の更新が収束したか否かを随時判定し，収束したと判定されたときに終了する．この収束判定は，CPU 上で VI を実行する場合，状態価値の更新処理に付随させることができるため，計算コストは低い．しかし，GPU 上で VI を実行する場合，そのように付随させることはできず一定の計算コストがかかる．この点に着目し，収束判定の計算コストを，収束判定回数を間引くことで削減する手法を提案した [7]．しかし，1 回の収束判定処理自体の計算コストを削減することは試みていなかった．

以上の背景のもと，本稿では，GPU 実装した VI における 1 回の収束判定処理の計算コストを削減する技法を提案・評価することを目的とする．提案技法は，GPU 実装の枠組みとして NVIDIA 社の CUDA[8] を前提とし，状態価値の変化量をすべての状態について CPU 側で調べるのではなく，スレッドブロックごとに GPU 側で算出した各最大値について CPU 側で調べるという素朴なものである．計算結果では，animat 問題，mountain-car 問題を対象として，提案技法を用いた場合の計算時間を示す．

2. マルコフ決定過程

2.1 状態遷移確率

$1,...,T$ という離散時刻をとり，状態，決定，外乱のすべてが離散的であるシステムを考える．システムがとりうる状態から構成される状態空間，システムへ与えうる決定から構成される決定空間，システムが被りうる外乱から構成される外乱空間を，それぞれ $\mathcal{S}$，$\mathcal{U}$, $\mathcal{W}$ とする．外乱 $w \in \mathcal{W}$ の生起確率が $P(w)$ として，また状態 $s \in \mathcal{S}$ において決定 $u \in \mathcal{U}$ を与えられたシステムが外乱 w を被って遷移する先の状態が $f(s,u,w)$ として，それぞれ利用可能であるとする．このとき，状態 s と s' が一致するか否かをそれぞれ 1, 0 として表す関数 $\delta(s,s') \in \{0,1\}$ を用いれば，状態 s において決定 u を与えた場合にシステムが状態 s' へ遷移する確率 $p^{(u)}_{s,s'}$ をつぎのように算出できる．

$$p^{(u)}_{s,s'} := \sum_{w \in \mathcal{W}} P(w) \cdot \delta(f(s,u,w), s'). \tag{1}$$

離散的システムの状態は，2 値をとる N^{s} 個の状態変数 $s_j \in \{0,1\}$ $(j \in \mathcal{J} := \{1,\ldots,N^{\mathrm{s}}\})$ で表すことができる．また，システムの状態遷移を計算機上で再現できるならば，状態 s において決定 u を与えたシステムが外乱 w を被ったとき，状態変数 j の値がどのように変化するかを，2 値関数 $f_j(s,u,w) \in \{0,1\}$ として表すことができる．状態 s の状態変数 j の値を表す関数 $S_j(s) \in \{0,1\}$ を用いれば，$\delta(f(s,u,w),s')$ はつぎのように表現できる [9].

$$\prod_{j \in \mathcal{J}} \big(f_j(s,u,w) S_j(s') + (1 - f_j(s,u,w))(1 - S_j(s'))\big). \tag{2}$$

方策 π が時刻 t の状態を写す関数を $\mu(t)$ $(t = 1,\ldots,T)$ とする．時刻 t にシステムが状態 s をとっているとき，方策 π を用いて決定を与えた場合に，システムが状態 s' へ遷移する確率を $\hat{p}^{(\pi)}_{s,s'}(t)$ と表す．この確率は $p^{(u)}_{s,s'}$ を用いてつぎのように算出できる．

$$\begin{aligned} \hat{p}^{(\pi)}_{s,s'}(1) &:= p^{(\mu_1(s))}_{s,s'}, \\ \hat{p}^{(\pi)}_{s,s'}(t+1) &:= \sum_{s'' \in \mathcal{S}} p^{(\mu_{t+1}(s))}_{s,s''} \hat{p}^{(\pi)}_{s'',s'}(t). \end{aligned} \tag{3}$$

2.2 状態価値関数

システムは状態 s において $C(s)$ $(0 \le \cdot \le \bar{C})$ というコストを被るとする．ここで $\bar{C}$ はコストの上界である．1 時刻将来のコストを α $(0 < \cdot \le 1)$ だけ割り引くとすれば，方策 π を用いた場合の有限期間 MDP における状態 s の価値は，つぎの関数として表すことができる．

$$F^{(\pi)}_T(s) := \sum_{t=0}^{T-1} \alpha^t \sum_{s' \in \mathcal{S}} \hat{p}^{(\pi)}_{s,s'}(t+1)\, C(s'). \tag{4}$$

この関数の値は，任意の状態 s に対して 0 を返す関数 $J_0(s)=0$ を用いれば，つぎのように算出できる．

$$
\begin{aligned}
F_0^{(\pi)}(s) &:= J_0(s), \\
F_{t+1}^{(\pi)}(s) &:= \sum_{s' \in \mathcal{S}} p_{s,s'}^{(\mu_{t+1}(s))} \left(C(s') + \alpha F_t^{(\pi)}(s') \right).
\end{aligned} \quad (5)
$$

無限期間 MDP では，方策が返す関数は時刻に依存しなくなり，状態をその状態の価値に写す関数 μ へと，方策は縮退する [1]．方策 μ を用いた場合の無限期間 MDP における状態 s の価値は，つぎのように表すことができる．

$$F^{(\mu)}(s) := \lim_{T \to \infty} F_T^{(\mu)}(s). \quad (6)$$

3. 動的計画法

3.1 価値反復法

VI は，時刻 $t+1$ の状態価値を，時刻 t の状態価値を用いて更新することを，所定の収束条件が成立するまで反復する手法である [1]．無限期間 MDP のための VI の記述にあたり，つぎのように定義される写像 $\mathcal{M}, \mathcal{M}_\mu$ を考える．

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

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

ここで，$J(\cdot)$ は状態価値関数を，$\mathcal{U}_s \subseteq \mathcal{U}$ は状態 s において選択可能な決定の集合を表す．$\mathcal{M}_\mu(J)(s)$ を用いて，方策 μ による状態価値を与える関数 $J^{(\mu)}(\cdot)$ をつぎのように表すことができる．

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

最適な決定を与える方策を $\mu^*(\cdot)$ とすれば，最適な状態価値関数 $J^*(\cdot)$ はつぎのように定義できる．

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

$J^*(\cdot)$ は (11) 式のベルマン方程式を満たし，(12) 式により求めることができる．

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

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

(12) 式の具体はつぎのような再帰式になる [1]．

$$\underline{J}_0(s) := 0, \quad (13)$$

$$\underline{J}_{k+1}(s) := \mathcal{M}(\underline{J}_k)(s) \;\; (k \geq 0). \quad (14)$$

VI は (14) 式を繰り返す手法である．

3.2 価値反復法の終了条件

理論的には，無限期間 MDP のための状態価値関数を求めるためには (14) 式を無限回数反復することになる．実際には，(14) 式の処理を行う前の状態価値から実行後の状態価値を減じた変化量がある閾値以下であれば収束したと判定し，反復を終了する．この閾値は意思決定者が定めるパラメータである．状態価値の許容誤差を $\epsilon\ (>0)$ とし，反復が $k+1$ 回実行済み，すなわち価値関数 $\underline{J}_{k+1}(\cdot)$ が算出済みであるとする．このとき，つぎの関係：

$$\max_{s \in \mathcal{S}} \left| \underline{J}_{k+1}(s) - \underline{J}_k(s) \right| \leq R(\epsilon) := \frac{(1-\alpha)\epsilon}{2\alpha}. \quad (15)$$

が成立していれば，関数 $\underline{J}_{k+1}(\cdot)$ が与える状態価値と，最適な状態価値関数が与える状態価値との差異に関して，以下の関係の成立が保証される [10]．

$$\max_{s \in \mathcal{S}} \left| F^{(\mu)}(s) - J^*(s) \right| \leq \epsilon. \quad (16)$$

割引率，許容誤差，コストの上界をそれぞれ $\alpha, \epsilon, \bar{C}$ とする．このとき，(15) 式（(16) 式）を満たすために十分な反復回数を $I\left(\alpha, \epsilon, \bar{C}\right)$ は，$|\underline{J}_{k+1}(s) - \underline{J}_k(s)| \leq \alpha^k \bar{C}$ であることから，つぎのように与えることができる．

$$I\left(\alpha, \epsilon, \bar{C}\right) := \left\lceil \frac{\log\left((1-\alpha)\epsilon\right) - \log\left(2\alpha\bar{C}\right)}{\log \alpha} \right\rceil. \quad (17)$$

ここで，$\lceil r \rceil$ は，r 以上で最小の整数を表す．

4. 価値反復法の GPU 実装

4.1 前提

本稿では，対象システムが 2 節で示した離散的システムであること，および状態を構成する状態変数の取りうる最小値・最大値が既知であると前提する．また，VI の GPU 実装にあたり，CUDA[8] の利用を前提とする．状態とスレッドとの対応付けを容易とするために，状態変数が 2 であるシステムのみを考慮し，状態とスレッドとを全単射として対応づけられることを前提とする．加えて，システムの状態空間は，その価値が 6.3 節に後述する評価のために用いる単一の GPU のデバイスメモリ収まらないほどには大きくはないことを前提とする．

4.2 状態価値の更新処理

本稿の実装では，1 つの状態の状態価値を更新する処理を 1 つのスレッドに担当させる [5]．また，GPU のデバイスメモリ上で状態価値関数を 1 次元の配列として取り扱う．これに関して，システムの状態をスレッド ID へ写す関数を用意する．この関数は，状態を規定する状態変数が $X_1 \in \{\underline{X}_1, \ldots, \overline{X}_1\}$, $X_2 \in \{\underline{X}_2, \ldots, \overline{X}_2\}$ の 2 つであるならば，つぎのように定められる．

$$H\left(Z\left((X_1 - \underline{X}_1)\left(\overline{X}_2 - \underline{X}_2\right) + X_2 - \underline{X}_2\right)\right). \quad (18)$$

ここで，$\underline{X}_j, \overline{X}_j$ はそれぞれ状態変数 X_j のとりうる最小値，最大値であり，$Z\ (\geq 1)$ は状態変数の拡大率を表し，$H(\cdot)$ は実数を最寄の整数へ写す関数を表す．(18) 式と逆の写像を行う関数は，あるスレッドに対応する状態からの遷移先状態を算出する際や，求め終えた状態価値を利用する際に用いられる．

CUDA を用いる際，(18) 式に示した写像における状態変数 $X_j\ (j = 1, 2)$ と，もとの問題における状態変数とをどのように対応づけるかは，システムの状態遷移の性質によっては重要となる．CUDA では，すべてのスレッドを同時に並列実行するわけではなく [*1]，全スレッドをスレッドブロックという単位へ分割し，あるスレッドブロックを構成する一群のスレッドを，複数の CUDA コアから構成される 1 つのマルチプロセッサ（以降，MP）へ割り付ける．各 MP では，スレッドブロックはウォープ (warp) と呼ばれる単位へ分割され，ウォープを構成する一群のスレッドが並列実行される．あるウォープがメモリアクセスを待っている間，そのウォープを一時的に退避し，同じ MP に割り付けられたべつのウォープを実行することで，処理が高速化されている．ウォープを構成するスレッドによるメモリアクセスは，アクセスするアドレスが連続していれば，スレッドごとにアクセスするのではなく 1 回のアクセスで複数のデータを読み書きする合同 (coalesced) が適用されることで，高速化される．合同を活用するためには，同じウォープに属する各スレッドについて，それらに対応する状態からの遷移先状態のスレッド ID が近しい必要がある．前述の状態変数の対応は，この性質が満たされるよう設計することが肝要である．

4.3 価値反復法の収束判定処理

すべての状態の状態価値を更新する処理をスイープと呼ぶ．$k+1$ 回目のスイープを実行するためには，k 回目のスイープを終えた状態価値関数が必要である．本稿では，状態価値を Gauss-Seidel 的には更新しないため，状態集合のサイズに比例する配列が少なくとも 2 つ必要となる．

有限期間 MDP のための VI ならば，必要なスイープ回数は計画期間数として自明である．一方，本稿で対象とする無限期間 MDP のための VI では，状態価値のスイープ前後での変化量が閾値以下となるための反復回数は，(17) 式のように許容誤差，割引率，コストの上界から算出できる．一般にこの繰返し回数は十分であるが必要ではなく多すぎるため，スイープ実行前後での状態価値の変化量が (15) 式を満たしているかチェックするほうが一般的である．これら 2 つの折衷的処理として，状態価値の変化量を間欠的にチェックする手法が提案されている [7].

状態価値の変化量の最大値は，ホストメモリ上に古い状態価値関数を保管しておき，新しい状態価値関数を GPU デバイスメモリからホストへ転送したのち，ホスト上で新旧の状態関数値の差をとることで算出できる．より計算コストが小さいと考えられる処理として，変化量を GPU の各コア上で算出しておき，状態価値関数ではなく変化量をホストへ転送し，この変化量が閾値以下であるか否かをホストで判定することが考えられる．本稿でもこの処理を行う．この処理では，更新前後の状態価値関数に加えて変化量も保管しておく必要から，状態空間のサイズに比例する配列を 3 つ GPU メモリ上に確保する必要がある．よって，本稿の GPU 実装で解くことのできる MDP の規模は，利用する GPU のデバイスメモリサイズを M，状態価値を表現するデータ構造のバイト数を b とすれば，$3b|\mathcal{S}| << M$ である必要がある．

*1 そのためには膨大なコアが必要である．

5. 提案技法

提案技法は，(15) 式に示した処理の効率化を目的とする．CUDA では，あるスレッドブロックを構成するスレッドは同一の MP 内で実行される．それらのスレッドは共有メモリを利用することで，データを高速にやり取りできる．提案技法は，共有メモリを利用し，GPU ではスレッドブロック内で状態価値の変化量の最大値を算出し，ホストではスレッドブロック間で変化量の最大値を算出することで，状態空間全体にわたる変化量の最大値を算出するという素朴なものである．形式的記述のため，全スレッドがスレッドブロックへ分割されるように，状態空間を部分状態空間 $\mathcal{S}_1, \ldots, \mathcal{S}_L$ へと分割する．ここで L は部分状態空間の数を表し，以下の関係が成立する．

$$\mathcal{S} = \bigcup_{l=1,\ldots,L} \mathcal{S}_l, \quad \mathcal{S}_l \cap \mathcal{S}_{l'} = \emptyset \ \ (\text{if } l \neq l'). \tag{19}$$

状態価値関数を $k+1$ 回目に更新した際の部分状態空間 l における状態価値の変化量の最大値を $d_l^{(k)}$ とする．この値はつぎのように算出される．

$$d_l^{(k)} := \max_{s \in \mathcal{S}_l} \left(J^{(k+1)}(s) - J^{(k)}(s) \right). \tag{20}$$

この値を用いて (15) 式の収束判定はつぎのように行える．

$$\max_{l=1,\ldots,L} d_l^{(k)} \leq R(\epsilon). \tag{21}$$

提案技法は，状態価値の変化量から VI の収束判定を行うために，スレッドブロックと部分状態空間を対応付け，スレッドブロック $l\ (l = 1, \ldots, L)$ ごとに $d_l^{(\cdot)}$ を GPU で求めておき，$d_l^{(\cdot)}\ (l = 1, \ldots, L)$ を GPU のデバイスメモリからホストメモリへ転送し，ホスト上で (21) 式に示した処理を行うというものである．加えて，スレッドブロック l における $d_l^{(\cdot)}$ の算出を，つぎのように並列して行う．

1° スレッドブロックのサイズを n とし，長さ n 以上の配列 v を共有メモリに用意する．ここで，n は 2 のべき

乗であると前提する．

2° スレッド i $(i = 1, \ldots, n)$ の状態価値の変化量を，それぞれ $v[i]$ へ設定する．ここで，i は当該スレッドブロックにおける相対値であり，スレッド ID とは異なる．

3° スレッドごとに，べつのスレッドと状態価値の変化量を比較する．比較対象のスレッドの当該スレッドからの相対位置を表す変数 a を，$a \leftarrow 1$ と設定する．

4° スレッド i $(i = 1, \ldots, n)$ ごとに，$a \leq n$ かつ比較対象スレッドのスレッド ID が不正でない（対応する状態が状態空間に含まれる）あいだ，以下を実行する．

(a) $v[i] \leftarrow \max(v[i], v[i + a])$.

(b) $a \leftarrow 2a$.

5° $v[1]$ の値を，当該スレッドブロックの $d_l^{(\cdot)}$ として，ホストへ送信する．

6. 評価

6.1 対象問題

本稿で提案する技法を，animat 問題 [5]，mountain-car 問題 [6] を対象として評価する．

animat 問題は，セルから構成される 2 次元平面とその中を動き回るエージェントを考え，平面内にランダムに設置された餌をできるだけ摂取することを目的として，当該エージェントが各セルでどちらへ動くかを定める問題である．この問題には，2 次元平面の大きさ，餌の数，餌の価値，エージェントの移動方向が意図とは逸れてしまう確率といったパラメータが存在する [5]．本稿では，2 次元平面を一辺 $1,024$ の正方形とすることを含めて，animat 問題を対象とした既存研究 [5] と同じパラメータ値を用いる．この設定では，状態空間のサイズは約 100 万である．animat 問題におけるシステム状態は，エージェントの 2 次元平面内における位置である．この位置と，4.2 節で用いた 2 つの状態変数との対応は，animat 問題では重要ではない．

mountain-car 問題は，低い丘と高い丘の中央の谷に置かれた車が，高い丘を最短時間で超えるよう動かすことを目的とする．車は各丘に向けて加速できるが，高い丘を直接上れるほどの加速はできないため，低い丘を登って降りて勢いをつける必要がある．この問題における決定は，低い丘方向へ加速，高い丘方向へ加速，加速しないの 3 つであり，状態は車の位置，速度という 2 つの状態変数をもつ．これらの状態変数は本来連続値をとり，状態空間が連続である問題のテストベッドとして用いられることが多い [6], [11]．本稿では，位置，速度をそれぞれ $10,000$ 倍ののち離散化して，離散状態空間問題として取り扱う．この設定では，状態空間のサイズは約 2,400 万である．ここで対象とするシステムでは，ある状態とその遷移先状態を比較したとき，位置は異なることが多く，速度は異なることが少ないという性質をもつ．このため，4.2 節に示した状態変数 X_1 と速度を，状態変数 X_2 と位置を，それぞれ対応させる．この逆に対応させた場合，位置が 1 つでも異なればスレッド ID は $Z\left(\overline{X}_2 - \underline{X}_2\right)$ だけ変化してしまい，遷移先状態集合のスレッド ID の不連続性は高くなり，メモリアクセスが合同されることはまれになる．

表 1 評価対象の技法

Table 1 Evaluated techniques.

技法	転送するデータ	収束判定間隔
技法 1	状態価値の変化量	毎回
技法 2		間欠的
技法 3	部分状態空間における状態価値の変化量の最大値	毎回
技法 4		間欠的

6.2 評価対象の技法

本稿で評価する 4 つの技法を表 1 に示す．技法 2, 4 における収束判定間隔は，既存研究 [7] を用いて定める．

6.3 計算時間

表 1 に示した 4 つの技法について，スレッドブロックのサイズ（5 節における n）を 16, 32, 64, 256, 512 としたプログラムを作製した．animat 問題の GPU 実装においては，状態遷移確率，セルでの報酬の大きさ，遷移先状態の扱い方について選択の余地がある．本稿では，これまでのところ計算時間が最短である [4] 設定として，状態遷移確率は定数メモリを，報酬データは 1D キャッシュを介して渡し，遷移先状態はデータを渡さずに GPU 上で毎回算出するという実装を行った．2 つの問題ごとに各プログラムを，CPU として Xeon E5-2640 v2，GPU として GeForce GTX TITAN Black を備える計算機上で 10 回実行し，計算時間の平均をとった．animat 問題を対象とした平均計算時間を対数軸の縦軸に，スレッドブロックのサイズを横軸にとったグラフを図 1 に示す．ここで，見易さのため，スレッドブロックのサイズの表示間隔を均等としている．

mountain-car 問題に対しては，図 2 における技法 1 と技法 3，また技法 2 と技法 4 の数値をそれぞれ比較することにより，提案技法は計算時間を削減するという意味で有効であることを確認できる．提案技法を用いた場合の計算時間は，用いない場合に対して，収束判定を毎回行う技法では 2.907%，間欠的に行う技法では 28.55%であった．この比率はすべてのブロックサイズにわたる平均である．

animat 問題に対しては，図 1 における技法 1 と技法 3 の数値を比較すると，収束判定を毎回行う場合，提案技法は有効であることを確認できる．提案技法を用いることで計算時間は，平均 83.08%に減少した．一方，収束判定を間欠的に行う技法 2, 4 の数値を比較すると，ブロックサイズが 4, 16, 32 の場合，提案技法を用いない技法 2 のほうが技法 4 よりも計算時間が短くなっている．このおもな原因は，本稿の animat 問題では状態空間が小さい上，技法 2,

ⓒ 2016 Information Processing Society of Japan

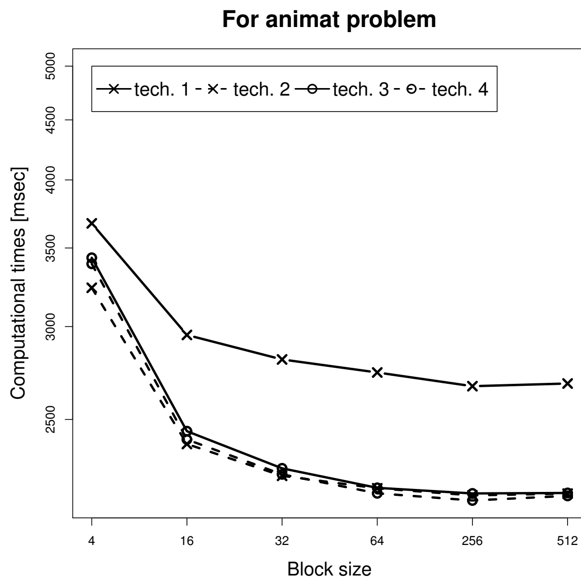

図 1　animat 問題を対象とした計算時間

Fig. 1　Computational times required for the animat problem.

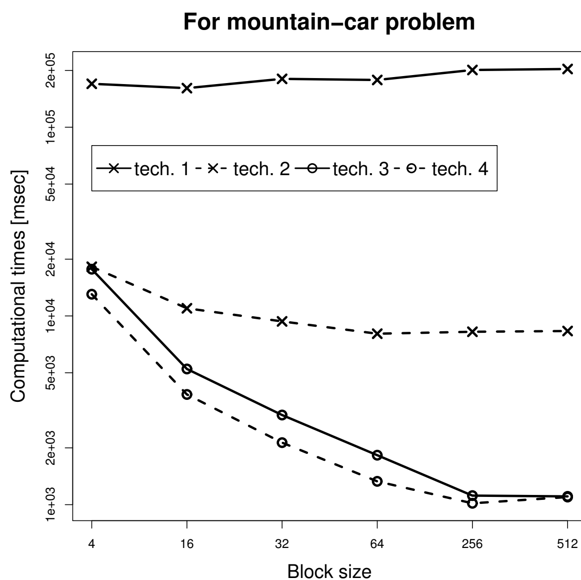

図 2　mountain-car 問題を対象とした計算時間

Fig. 2　Computational times required for the mountain-car problem.

4 では収束判定回数が少なく，収束判定に要した計算時間があまり長くないことにあると考える．加えて，スレッドブロックのサイズが小さい場合には，MP 上で同時に実行されるウォープ数が少なく，提案技法がスレッドブロックごとに引き起こす 5 節に示した処理が並列化により隠されなかったことから，サイズが 4, 16, 32 の場合には計算時間が長くなったと推測する．

7. おわりに

本稿では，無限期間マルコフ決定過程のための価値反復法の GPU 実装における収束判定処理を高速化するための技法を提案した．提案技法は，状態価値の変化量の状態空間にわたる最大値をホスト側で算出する代わりに，部分状態空間にわたる最大値を GPU デバイス側で算出・転送し，ホスト側では各部分状態空間にわたる最大値を算出するという素朴なものである．状態空間が約 100 万，2,400 万である animat 問題，mountain-car 問題を対象とした評価の結果，状態空間が大きければ，あるいはスレッドブロックのサイズが大きければ，提案技法により計算時間を短縮できることを確認できた．

今後の課題として，状態とスレッド ID との対応が自明でない複雑なシステムを対象とした価値反復法の GPU 実装や，可到達状態集合を数え上げる処理の GPU 実装，単一のデバイスのメモリに収まらない状態空間を有するシステムを対象とした動的計画法の GPU 実装などがあげられる．

参考文献

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

[2] Sutton, R. S. and Barto, A. G.: 強化学習, 森北出版, 第 1 版 (2000), 三上貞芳，皆川 雅章 共訳.

[3] Bertsekas, D. P.: Distributed Dynamic Programming, *IEEE Trans. Autom. Control*, Vol. AC-27, No. 3, pp. 610–616 (1982).

[4] Inamoto, T., Matsumoto, T., Ohta, C., Tamaki, H. and Murao, H.: An Implementation of Dynamic Programming for Many-Core Computers, in *Proceedings of SICE Annual Conference 2011 (DVD-Paper)*, pp. 961–966, Waseda University, Japan (2011).

[5] Jòhannsson, À. P.: GPU-Based Markov Decision Process Solver, Master's thesis, School of Computer Science, Reykjavìk University (2009).

[6] Singh, S. P. and Sutton, R. S.: Reinforcement Learning with Replacing Eligibility Traces, *Machine Learning*, Vol. 22, pp. 123–158 (1996).

[7] Inamoto, T., Higami, Y. and Kobayashi, S.: Optimal Periods for Probing Convergence of Infinite-stage Dynamic Programmings on GPUs, *International Journal of Networking and Computing (Special Issue on the First International Symposium on Computing and Networking)*, Vol. 4, No. 2, pp. 321–335 (2014).

[8] NVIDIA corporation: *NVIDIA CUDA Programming Guide*, 3.0 edition (2010).

[9] 稲元勉, 玉置久, 村尾元：エレベータ運行計画問題に対する動的計画法の一構成と状態遷移モデルの縮約による効率化, 計測自動制御学会論文集, Vol. 44, No. 2, pp. 174–182 (2008).

[10] Powell, W. B.: *Approximate Dynamic Programming*, John Wiley & Sons, Inc. (2007).

[11] 釜谷博行, 北山数行, 藤村敦子, 阿部健：連続状態空間における強化学習, 計測自動制御学会東北支部第 229 回研究集会資料集 (2006).