General-Purpose Computation on GPUsを用いた再現性のあるAgent-Based Simulationの高速化

○原田拓弥　村田忠彦　（関西大学）

Accelerating Reproducible Agent-Based Simulations Using the General-Purpose Computation on GPUs

∗T. Harada and T. Murata (Kansai University)

概要– In this study, we try to accelerate reproducible Agent-Based Simulation (ABS) using the General-Purpose Computation on GPUs (GPGPU). GPGPU is a technology that uses the calculation resource of GPU to calculate other than the image processing. CPU and GPU architecture and their programming technique are diffrent. Therefore it is difficult to run reproducible ABS on CPU and GPU. In this study, we propose two models to run reproducible ABS in CPU and GPU. The first is a stand-alone model that to parallelizes the independent ABS. The other is a distributed model to run large-scale ABS parallelizing decisions agents. These models can be ensured reproducibility even in the experiment of changing the number of parallelization. The results of the experiment show that the stand-alone model was able to obtain a 40-fold acceleration rate than CPU. The distributed model was able to obtain a 120-fold acceleration rate than CPU.

キーワード: Agent-Based Simulation, Large-Scale Simulation, GPGPU, CUDA, Random Number Generator

1　はじめに

本研究では，Agent-Based Simulation（以下 ABS）の高速化を General-Purpose computation on Graphic Processing Units（以下 GPGPU）を用いて行う．GPGPU とは，画像処理を行う GPU を数値計算に利用する技術である．ABS においてエージェントの意思決定は，エージェント同士の相互作用や環境からのフィードバックにより行う．そのため，ABS を並列化する際には同期やメモリアクセス，実験の再現性等の問題が発生する．また，GPGPU を用いることで CPU より高速化が実現できたとしても，同じシミュレーション設定で非並列化の CPU と並列化した GPU とで計算結果が異なる場合，GPU で得られた計算結果の信頼性はない．これは，CPU で並列化を行った場合も同様である．本研究では，エージェントが環境からのフィードバックのみで意思決定を行うモデルを使用し，CPU と GPU，また並列数を変更した場合においても再現性のあるシミュレーションの高速化手法を提案する．

GPGPU を含めた並列分散処理などで複数の乱数生成器を用いて計算を行う場合，乱数生成器の初期化に注意すべきである．乱数生成器の初期化手法によっては，複数の乱数生成器から同じ乱数列が出力される場合がある．シミュレーションを行う上で同じ乱数列が用いられることは好ましくない．本研究では，重複のない乱数列を生成する複数の乱数生成器の初期化手法及び，GPGPU に適した乱数生成器を示す．

先行研究で我々は ABS のモデルの 1 つである Minority Game[1)]（以下 MG）を用いてシミュレーションを行い，MG における勝者の数の周期的変化の原因を報告した [2)]．このような MG の振る舞いを調べるためには大規模化や試行回数を増やす必要があることがわかった．しかしながら，大規模化や試行回数を増やすと計算時間が非常に長くなる．そこで本研究では GPGPU を用いて MG の高速化を行う．

関連研究として先山ら [3)] は GPU を用いたマルチエージェント・シミュレーション用ライブラリである MasCL を OpenCL を用いて開発を行った．OpenCL は GPGPU の環境の 1 つである．OpenCL の他にも CUDA や DirectCompute が GPGPU の環境として有名である．先山らはシミュレーションの描画に重きを置いており，シミュレーションの再現性については言及されていない．本研究ではシミュレーションの高速化及び再現性に焦点を当て，再現性のあるシミュレーションを行うモデルを提案する．

OpenCL は CUDA に比べ GPU のパフォーマンスを引き出すことは難しく，同じアルゴリズムであっても CUDA のほうが高速化に期待できる [4)]．荒井らは CUDA のコンパイラが生成する，GPU アセンブリ言語の PTX コードと OpenCL が生成する PTX コードの比較を行い，CUDA が生成する PTX コードが優れていることを示した [5)]．また，CUDA で行われている基本的な最適化が OpenCL では行われていない．OpenCL が CUDA と同等のパフォーマンスを得るためには，PTX を直接記述する必要がある．

2　Minority Game

MG[1)] とは奇数人のエージェントが二者択一し，少数派に属すると利得を得るモデルである．エージェントは勝者の履歴（以下履歴）と戦略表を用いて意思決定を行う．履歴は全エージェントで共有されており，過去履歴長 m 期の勝者の選択を保持している．初期の履歴はランダムに生成する．戦略表は全ての履歴に対応したエージェントの選択と得点が記されている．意思決定を行う際，エージェントは自身が保持する戦略表の数 s 個のうち，最も得点の高い戦略表を使用し，現在の履歴に対応した選択をする．同じ得点の戦略表が複数ある場合，その中から使用する戦略表をランダムに 1 つ選択する．ゲームに勝てば使用した戦略表に 1 点加点，負ければ 1 点減点する．戦略表はシミュレーション開始時にランダムに生成し，1 回のシミュレーション中は同一の戦略表を用いる．

MG を大規模化や試行回数を増加させると勝者の数

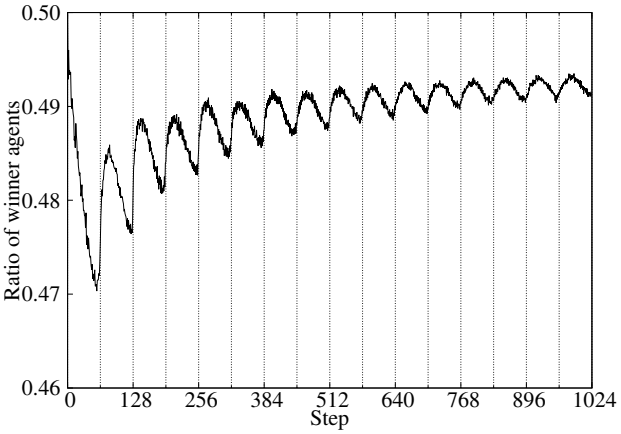

Fig. 1: 勝者の数の周期的変化

が周期的に増減する．Fig. 1 にシミュレーション結果を示す．シミュレーションの設定は，1,000 万エージェント，履歴の長さ $m = 5$，戦略表の数 $s = 2$，シミュレーション期間 $p = 1{,}024$ とし，1,000 回シミュレーションを行った．縦軸は勝者の割合の 1,000 回の平均を，横軸はシミュレーション期間を示している．勝者の割合が 64 Step にかけ減少し，64 Step 付近で大きく増加している．これはある履歴 h を初めて使用した時，戦略表はランダムに生成されるため，2 つのグループを選択するエージェントの比率は 1:1 となる．負けたエージェントの一部が戦略表を切り替える．戦略表を切り替えたエージェントの約半分が次に履歴 h を使用するとき，前回履歴 h を使用したゲームの少数派グループを選択する．2 つのグループを選択する比率が 1:1 から大きく離れ，勝者の数が減る．ゲームに負けた一部のエージェントが戦略表を切り替えるため，次回履歴 h を使用したゲームでは，2 つのグループを選択するエージェントの比率は再度 1:1 となる．以降 2^{m+1}Step 毎に，これらの操作が繰り返し実行されるときに周期が発生する．

大規模化や試行回数の増加させたり，MG の振る舞いを観察するためにシミュレーション期間を長くしたりすると，非常に時間がかかる．本研究では，シミュレーションにかかる時間を短縮するために GPGPU を用いた高速化を行う．また，時間短縮を目的としているため，シミュレーション結果が CPU と GPGPU で同一である必要がある．本研究の目的は，シミュレーションの高速化及び CPU と GPGPU とで同一の結果を得ることとする．

3 GPGPU の環境

3.1 概要

GPGPU の環境として NVIDIA 社の CUDA[6]（Compute Uniffied Device Architecture）や Khronos Group の OpenCL[7]（Open Computing Language），Microsoft の DirectCompute が有名である．OpenCL は GPU 以外にも CPU や Cell プロセッサ等，様々な計算環境で動作するクロスプラットフォームなフレームワークである．一方，CUDA は NVIDIA 製の GPU でしか動作せず，OpenCL と比べるとハードウェアの汎用性がない．しかし，CUDA はハードウェアの性能を最大限引き出せるように設計されており，プログラマがハードウェアの機能を使用した最適化が可能である．DirectCompute は Microsoft Windows 上でのみ動作する．OpenCL や CUDA は Windows 以外にも Linux や Mac OS 等で動作するため，DirectCompute はソフトウェアの汎用性がない．また，DirectCompute はグラフィックス連携を想定して設計されている．本研究ではシミュレーションの描画は行わないため，DirectCompute は不適切である．

OpenCL は実行時に使用するデバイス向けのコードを生成する，Just-In-Time 方式のコンパイルを採用している．一方 CUDA は，プログラムのビルドと同時に GPU 向けのコードを生成するため，OpenCL よりコンパイラが最適化が行われたコードを生成できる．また，NVIDIA は自社の CUDA があるため，NVIDIA 製の GPU の OpenCL の対応状況があまりよくない．特に，NVIDIA 製の GPU を用いた OpenCL では，GPU メモリが 3 GB 以上使用できないため，大規模なシミュレーションを行えない．本研究では ABS の高速化を目的としているため，GPGPU の環境として CUDA を使用する．

3.2 CUDA のプログラミング

CUDA ではプログラミング言語として CUDA C が用いられる．CUDA C は C 言語をベースに C++の一部の機能を組み込んだ言語である．プログラムを GPU で動作させるためには，GPU で動作させたい関数の戻り値の型の前に__global__（以下カーネル関数）を指定する．カーネル関数から GPU で動作させる関数には__device__を指定する．CPU でのみ動作させる関数には__host__を指定する．何も指定しない場合は__host__と同等である．__device__と__host__は同時に指定でき，本研究のように CPU と GPU で同一の計算を行う際には同時に指定すると便利である．カーネル関数を CPU で実行することはできないため，カーネル関数を CPU で動作させる場合は同等の関数を作成する必要がある．

また，CUDA にはカーネル，グリッド，ブロック，スレッドの 4 つの概念があり，Fig. 2 を用いて説明する．なお，Fig. 2 は NVIDIA の CUDA C Programming Guide[8] を参考に作成した．カーネルとは GPU で動作させるプログラムのことである．カーネル関数を実行する際，関数の引数とは別にグリッドとブロックの大きさを指定する．グリッドはブロックをまとめたもので，カーネルごとに 1 つである．ブロックはスレッドをまとめたものである。グリッドサイズ，ブロックサイズの上限は共に GPU の世代 [1] によって異なる．スレッドは処理を行う最小単位である．グリッドサイズ，ブロックサイズはともに 1 次元から 3 次元単位で指定できる点が CPU の並列化と大きく異なる．

ブロックは GPU に複数個搭載されているプロセッサで処理を行う．スレッドはプロセッサ内の各コアが処理を行う．GPU のプロセッサの名称を NVIDIA は Streaming Multiprocessor（以下 SM）や Streaming Multiprocessor eXtream（SMX）としている [2]．本研究では CPU と区別する必要がある場合，GPU のプロセッサを SM とする．また，CUDA は複数のブロックそれぞれが複数のスレッドを持つ構造である．本研究

[1] NVIDIA は GPU の世代を Compute Capability という数値で定めている．Compute Capability は CUDA Sample の DeviceQuery により取得できる．

[2] GPU の世代ごとに名称が異なっている．

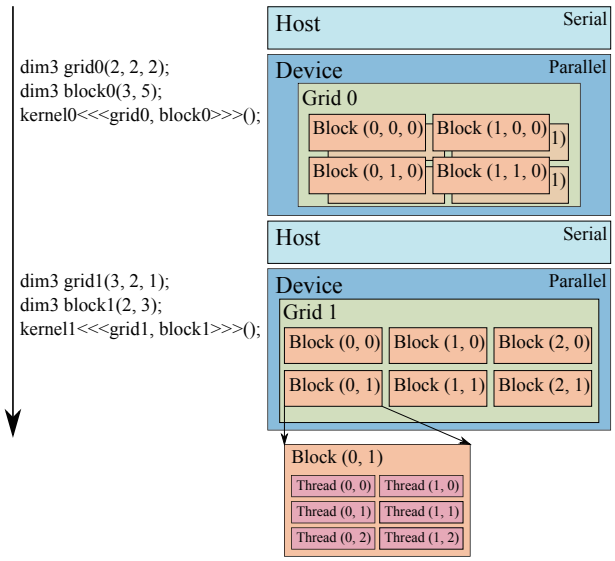

Fig. 2: CUDA のプログラミング概要

Table 1: 実験環境

CPU	Intel Core i7-3930K
メモリ	DDR3-1600 8GB×8
GPU	NVIDIA GeForce 670 GTX
OS	Microsot Windows 8.1 Pro 64bit
ドライバ	Geforce 347.25
API	CUDA 7.0 Release Candidate

ではスレッド数とはプロセッサあたりのスレッド数とし，並列数は全プロセッサのスレッド数とする．例えばブロック数が 100，スレッド数が 50 の場合，スレッド数は 50，並列数は 5,000 となる．

3.3 実験環境

本研究の実験環境を Table 1 に，CPU，GPU のプロセッサの概要を Table 2 に示す．GPU は CPU に比べ動作クロックは低くメモリサイズも小さいものの，コア数は多く，メモリ速度は速い．GPGPU は GPU を汎用的な計算に用いる技術だが，GPGPU で高速化するためには GPU が処理を行いやすいプログラムを書く必要がある．

GPU は動画像処理を行う目的で作られている．動画像処理では一般的に同じプログラムを複数のデータに対して行う．GPU では動画像を分割し，分割した領域を SM が処理する．複数個の SM が並列に処理を行うことで，GPU は CPU に比べて非常に高速に動画像処理を行うことができる．このように動画像の領域を分割するなど，異なるデータを並列に処理をするアーキテクチャを Multiple Instruction Multiple Data streams[9]（以下 MIMD）という．MIMD の中で同じプログラムを並列実行するものを，Single Program Multiple Data streams（以下 SPMD）といい，GPGPU では SPMD 型でプログラムを作成する．本研究で我々が用いる ABS では，エージェントはエージェント自身が保有するデータと複数のエージェントが共有しているデータを基に，データは異なるが同じルールの意思決定を行う．そのため，ABS は SPMD 型でのプログラムの作成が容易である．

Table 2: プロセッサの概要

	CPU	GPU
クロック	3.2 GHz	0.98 GHz
プロセッサ数	1	7
プロセッサあたりのコア数	6	192
総コア数	6	1,344
メモリ速度	51 GB/s	192 GB/s
メモリサイズ	64 GB	4 GB

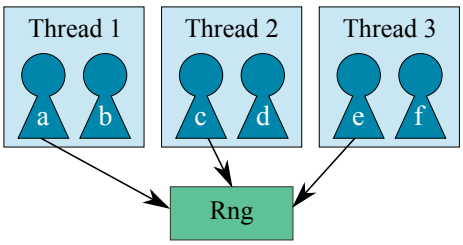

Fig. 3: 並列計算における乱数生成器の参照

4 モデル

4.1 概要

GPGPU に限らず並列分散処理を行う場合には，処理する量を並列数だけ分割する必要がある．ABS では複数のエージェントが意思決定を行うモデルであるため，分割する単位としてエージェントが思いつく．Fig. 3 にエージェントを複数のスレッドに分割した例を示す．Fig. 3 の Rng とは乱数生成器である．ABS ではエージェントの意思決定に乱数がよく用いられる．Fig. 3 では 2 つのスレッドに対して乱数生成器は 1 つである．2 つのスレッドは同時に処理を行うため，エージェント a とエージェント c，エージェント e のどのエージェントが先に乱数を生成するかわからず，シミュレーションの再現性を取ることができない．

Fig. 4 はスレッドごとに乱数生成器を持たせた例である．各スレッドが処理をする順番は変わらないため，Fig. 4 ではシミュレーションの再現性を取ることができる．しかし，Fig. 4 では並列数を変更した場合では乱数生成器の数が異なるため，シミュレーションの再現性を取ることができない．特に CPU と GPU ではスレッド数が大きく異なる．CPU に合わせて GPU のスレッド数を少なくすると GPU の性能を活かしきれず，また GPU は CPU に比べて周波数が低いため，GPU による高速化は期待できない．GPU に合わせて CPU のスレッド数を非常に多くすることは難しい．なぜなら，スレッドを生成すると各スレッドにスタック空間として，環境によって異なるが 1 MByte 程度割り当てられる．スレッド数を増やすとスタックサイズの合計が非常に多くなり，シミュレーションに使用するメモリ領域を確保できないという問題が発生する．他にも，シミュレーション結果が並列数，すなわち，乱数生成器の数によって変化することも問題である．非並列化や CPU による並列化，GPU による並列化全てにおいて同一のシミュレーション結果を得ることが理想的である．

そこで本研究では，Fig. 5 のモデルを提案する．Fig. 5 では，複数のエージェントとそのエージェントが参照する乱数生成器をグループ化し，グループを並列数に応じてに均等に分配する．グループを処理の単位とす

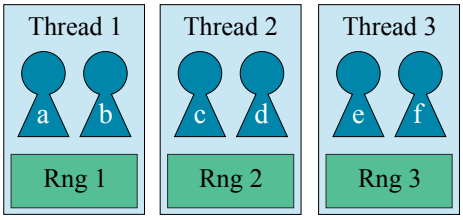

Fig. 4: 並列数依存な乱数生成器の参照

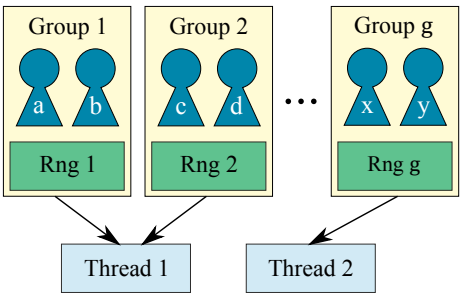

Fig. 5: 並列数に依存しない乱数生成器の参照

ることで計算機や並列数に依存しないシミュレーションが可能である．しかし，グループ数が並列数未満の場合，処理を行うことができないスレッドが発生し効率が悪くなる．そのため，グループ数は並列数の上限あたりがよい．

Fig. 5 ではエージェントをまとめたグループをスレッドに割り当て処理を行う．CPU の場合は問題ないが，GPU の場合スレッドをまとめたブロックがある．グループを複数のブロック内のスレッドに割り当てると大規模なシミュレーションを行えるが，小規模なシミュレーションは行えない．

本研究では，Fig. 5 を組み込んだ 2 つのモデルを作成する．1 つはグループを複数のブロック内のスレッドに割り当て大規模なシミュレーションを行うモデルである．もう 1 つは，グループの割り当てを 1 つのブロック内にとどめ，複数のブロックで独立した MG を実行するモデルである．本研究では前者を分散型，後者を独立型とする．

4.1.1 分散型

GPU で分散型のモデルを実行するためには，全ブロック内の全スレッドに対して同期する必要がある．MG の場合，全エージェントが 2 つのグループを選択した後，少数派グループを決める際に同期する必要がある．CUDA の同期命令として，GPU の全ての処理が終わるまで CPU が待つ命令と，GPU が行う命令の中でブロック内の全スレッドが同期する命令が用意されているが，GPU が行う命令の中で全ブロックが同期する命令は用意されていない．本研究では，ブロック間の同期を行う命令を自作し CPU で同期するモデルと比較検討を行う．CPU で同期するモデルを分散カーネル同期型，ブロック間の同期するモデルを分散グリッド同期型とする．分散カーネル同期型は CPU で同期を行うため，複数の GPU を使用しより大規模なシミュレーションを行える．一方分散グリッド同期型の同期は GPU 内部で行うため，複数の GPU を使用した大規模化は行えない．

NVIDIA が全ブロックを同期する命令を用意しない理由として以下を挙げている [10]．

- ハードウェアの実装にコストがかかる
- デッドロックを避けるためにブロック数を少なくする必要があり，全体的な効率が落ちる

分散グリッド同期型の場合，デッドロックが起きる可能性がある．本研究では，分散グリッド同期型の処理時間とともに，デッドロックが起きるか検証し，デッドロックが起きる場合は，ブロック数及びスレッド数の上限を示す．

4.1.2 独立型

独立型のモデルの場合，各ブロックは独立したシミュレーションが動作しているため，全ブロックで同期する必要はない．グループ数は GPU の最大スレッド数と同数とすると，GPU のスレッド数を変更しても再現性のあるシミュレーションを行うことができる．また，グループ内のエージェント数はシミュレーションに合わせて変化させる．独立型のモデルの場合，エージェント数はグループ数以上となる．効率よく処理を行うためには，エージェント数はグループ数の倍数にする必要がある．

4.2 乱数

4.2.1 乱数生成器

CPU や GPU などの異なる環境で再現性のあるシミュレーションを行うためには，計算機依存でない乱数生成器を使用する必要がある．乱数生成器といえば Mersenne Twister[11] が有名である．Mersenne Twister は周期が $2^{19937}-1$ と長く，また周期に対するメモリ効率が良いため，様々なソフトウェアで使用されている．しかし，Mersenne Twister はメモリ効率は良いものの，使用するメモリ量は 19,968 Bytes と多い．Mersenne Twister を CPU で使用する場合，メモリ量は多く乱数生成器の数は少数で済むため，乱数生成器が使用するメモリ量は気にする必要はない．しかし，GPU のメモリ量は少なく，本研究で提案するモデルには大量の乱数生成器が必要であるため，Mersenne Twister は適切でない．

GPU を使用する乱数生成器として，Mersenne Twister for Graphic Processors[12]（以下 MTGP）がある．MTGP は Mersenne Twister を GPU で動作させ高速に乱数を生成できる．MTGP は乱数生成の高速化に焦点を当てた乱数生成器であるため，本研究のモデルに適切でない．

使用するメモリ量が少ない乱数生成器として Xorshift[13] や TinyMT[14] がある．Xorshift の周期は $2^{128}-1$ と Mersenne Twister と比べて極めて短いものの，使用するメモリ量は 16 Bytes と非常に小さい．TinyMT の周期は $2^{127}-1$，メモリ量は 28 Bytes と Xorshift と比べメモリ使用量が多く周期は短い．しかし TinyMT は疑似乱数テストの 1 つである TestU01[15] の BigCrush テストをパスしているため，TinyMT の方が高品質な乱数を生成できる．

Xorshift を改良し開発された乱数生成器として Xorshift-Add[16]（以下 XSadd）がある．XSadd は周期や使用メモリ量は Xorshift と変わりなく，BigCrush テストをパスした乱数生成器である．XSadd の 64 bit 版である Xorshift+[17] も BigCrush テストをパスしている．GPU のレジスタ長は 32 bit であるため，GPU で

はXorshift+よりもXSaddの方が高速な動作に期待できる．Xorshift+は複数のバージョンがあり，それぞれ周期長が異なる．Xorshift+の周期は $2^{64}-1$，$2^{128}-1$，$2^{1024}-1$，$2^{4096}-1$ の4種類ある．本研究ではGPU上で高速な動作が期待できるXSaddを乱数生成器として用いる．

シミュレーションでXSaddを使用するためには，MGの乱数使用回数がXSaddの周期である $2^{128}-1$ 以下であることを確認する必要がある．1回のMGの乱数使用回数 r の算出方法を式（1）に示す．

$$r = n \times (s \times 2^m + p) + 1 \quad (1)$$

式（1）の n はエージェント数，s は戦略表の数，m は履歴の長さ，p はシミュレーション期間である．$s \times 2^m$ は戦略表の生成に必要な乱数の数である．また p はシミュレーション中に必要な乱数の数である．MGではゲーム開始時に履歴を生成する．最後の1は履歴の生成回数である．

仮にMGの設定を $n = 1{,}000{,}000{,}000$，$m = 10$，$s = 10$ とし，シミュレーション期間 $p = 100{,}000$，試行回数1,000,000としても，$r \times 1{,}000{,}000 = 1.10 \times 10^{20}$ となりXSaddの周期である $2^{128} - 1 = 3.40 \times 10^{38}$ 以下であるため，MGのシミュレーションではXSaddの周期でも十分である．XSaddの周期長で不十分の場合，Xorshift+を用いることで周期に関しては解決できるが，周期が長くなれば使用するメモリ量も増加する．シミュレーションに使用するメモリ量やシミュレーションに使用する乱数の数を注意し，Xorshift+のバージョンを選択する必要がある．

4.2.2 並列計算における乱数生成器の初期化

複数の乱数生成器を使用する場合は乱数生成器の初期化に注意する必要がある．Fig. 6に並列計算における理想的な乱数生成器の初期化を示す．Fig. 6の円は乱数列，Rngは乱数生成器，灰色の楕円は乱数生成器から生成される乱数列である．乱数生成器を初期化すると初期化に使用した情報から乱数列の特定の位置を算出し，その位置から順に乱数を生成する．複数の乱数生成器を使用する場合，乱数生成器を適当に初期化するとFig. 7のように複数の乱数生成器から生成される乱数列が重複することがある．特に非常に多くの乱数生成器を使用する本研究のモデルでは，乱数生成器が生成する乱数列の重複が起きやすくなる．このような状況はシミュレーションを行う上で好ましくない．

そこで本研究では乱数生成器の初期化する情報は全乱数生成器で同一とし，初期化後，各乱数生成が生成する乱数列をずらすことで乱数生成器が生成する乱数列の重複を避けている．乱数列をずらす最も簡単な方法はずらす数だけ乱数を生成する．しかし非常に時間がかかる．効率よく乱数列をずらすためには，使用する乱数生成器のアルゴリズムをよく理解し，乱数生成器を初期化する必要がある．幸いにも本研究で使用するXSaddはMITライセンスで公開されており，乱数列を指定の数だけずらす関数が用意されている．本研究ではこの関数を使用し，重複のない乱数列を生成する複数の乱数生成器を初期化している[3]．

[3]Xorshiftにおける並列計算用の乱数生成器の初期化は，大矢らの手法[18]を用いることで高速化が可能である．

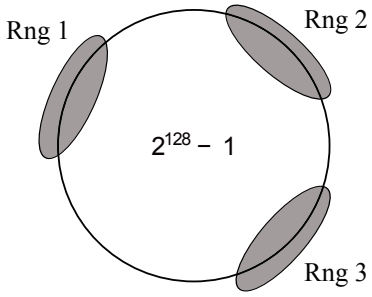

Fig. 6: 理想的な乱数生成器の初期化

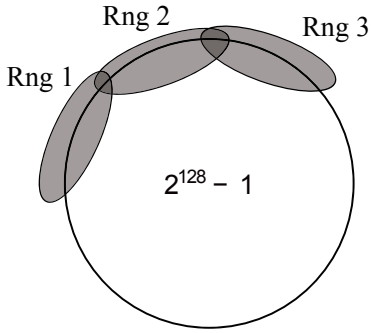

Fig. 7: 問題のある乱数生成器の初期化

乱数生成器の初期化時に各乱数生成器がずらす量 r_{init} を式（2）に示す．

$$r_{init} = r \times tr + \frac{r}{div} \times id \quad (2)$$

式（2）の tr は現在の試行回数で，div は処理の分割数である．独立型の場合，分割数 div はスレッド数と同等である．分散型の場合，分割数 div は並列数と同等である．id は $id \in \{0, 1, 2, ..., div-1\}$ であり，スレッドの識別番号である．

5 処理時間

5.1 再現性の確認

本研究ではCPUやGPU，並列数を変更したシミュレーションにおいても，同一のシミュレーションを行えたか確認するために，計算に使用した様々なデータを出力し比較を行った．比較を行ったデータを以下に示す．

- 全シミュレーション期間における勝者の数
- 履歴の推移
- 各エージェントが保有する全ての戦略表
- 全シミュレーション期間において各エージェントが使用した戦略表
- 各乱数生成器が出力した乱数列

CPUやGPU，並列数を変更したシミュレーションにおいても上記全てのデータが一致していることが確認でき，本研究で提案するモデルはグループ数を変更しない限り様々な環境で再現可能であるといえる．MGに使用するデータとして上記以外に各戦略表の得点や勝者の選択，全シミュレーション期間における各エージェントの取った行動がある．これらのデータは上記のデータを基に算出できるため，比較を行っていない．

5.2 処理時間計測方法

本研究では処理時間の計測にC++標準ライブラリのchronoを用いる．chronoにはsystem_clock,

Table 3: steady_clock の実装と精度

	MSVC	gcc（libstdc++）
実装	GetSystemTimeAsFileTime()	clock_gettime() gettimeofday() time()
精度	1,000 マイクロ秒	1 マイクロ秒

steady_clock，high_resolution_clock の 3 種類の時計型がある．system_clock はシステム時間を取得する時計型である．steady_clock はシステム時間が変更されても時間が逆行しない時計型である．high_resolution_clock は高精度の時計型であるが，MicroSoft Visual C++（以下 MSVC）や gcc（libstdc++）等，現在主要なコンパイラでは system_clock と同等である．

本研究では長時間時間計測を行うため，時間計測に steady_clock を使用する．chrono の実装は実装依存である．そのため steady_clock の精度も実装依存である．Table 3 に steady_clock の実装と本研究の環境での精度を示す．gcc（libstdc++）の実装は 3 種類あり，libstdc++のビルド時に使用される関数が決定する．本研究の環境では gettimeofday 関数が使用されていた．本研究では MSVC を用いてビルドを行う．そのため本研究の処理時間の精度は 1 ミリ秒である．

5.3 処理時間

本研究が提案する独立型，分散カーネル同期型，分散グリッド同期型の処理時間と CPU の処理時間の比較を行う．CPU の処理時間には非並列化と OpenMP による並列化の 2 つを用意する．GPU の各モデルと CPU の条件を同一とするため，CPU の並列化は GPU の各モデルと同等な並列処理が行えるようにプログラムを作成した．独立型は独立したシミュレーションを並列化するモデルである．CPU では独立したシミュレーションを並列化する．分散型は大規模なシミュレーションを行うため，エージェントの意思決定を並列化するモデルである．CPU ではエージェントの意思決定を並列化する．分散型のモデルは分散カーネル同期型，分散グリッド同期型の 2 つあるが，同期を行う場所が異なるだけである．そのため，CPU による並列化を行うプログラムは独立型で 1 つ，分散型で 1 つの合計 2 種類作成する．

本研究では，関数を呼び出す前後の時間を取得し，その差分を処理時間とする．CPU と GPU 基本的に同一のプログラムを使用している．例外として，GPU で処理を行う場合，GPU は CPU のメモリにアクセスできない．CPU と GPU 間でデータの転送を行うときには，CPU からデータの転送を行う命令をする必要がある．そのため GPU の処理時間に，シミュレーションの初期化やシミュレーション結果など，データを転送する時間も含めている．また，本研究では Table 1 の GPU を演算専用とする．演算専用とすることで，Operating System（OS）などの描画命令を行わず，GPU の全ての計算資源をシミュレーションに充てる．

5.3.1 独立型

独立型の処理時間を Table 4 に示す．シミュレーション設定は，履歴の長さ $m = 5$，戦略表の数 $s = 2$ とし，シミュレーション期間 $p = 4{,}096$，試行回数 $tr = 224$ とし，エージェント数は $n = 32{,}767$ と $n = 1{,}048{,}575$ の 2 種類シミュレーションを行った．また，GPU のブロック数は 112，スレッド数を 512，CPU の並列数を 6，グループ数は 1,024 とした．

Table 4: 独立型の処理時間（秒）

	エージェント数	
	$n = 32{,}767$	$n = 1{,}048{,}575$
GPU	3.3	75.5
CPU（非並列化）	238.5	7,152.0
CPU（並列化）	51.7	3,254.8
加速率	15.67	43.11

エージェント数は $n = 32{,}767$ と $n = 1{,}048{,}575$ で約 32 倍増加させた．処理時間は 32 倍になると予想できる．CPU（非並列化）では $n = 1{,}048{,}575$ の処理時間は $n = 32{,}767$ の処理時間の約 30 倍と予想通りである．CPU（並列化）の $n = 1{,}048{,}575$ の処理時間は $n = 32{,}767$ の約 63 倍と遅い．GPU の $n = 1{,}048{,}575$ の処理時間は $n = 32{,}767$ の約 23 倍と，$n = 32{,}767$ の処理時間が遅い．これは $n = 32{,}767$ の処理量が少なく，GPU の計算資源を活かしきれていないと考えられる．

Table 4 の加速率は CPU（並列化）と GPU の処理時間を基に算出した．$n = 32{,}767$ の加速率は 15 倍，$n = 1{,}048{,}575$ の加速率は 43 倍と CPU に比べ GPU は非常に速くシミュレーションを行えた．

5.3.2 分散型

分散型の処理時間を Table 5 に示す．エージェント数 $n = 100{,}007{,}935$ と $n = 500{,}039{,}679$ の 2 種類シミュレーションを行った．シミュレーション設定は，履歴の長さ $m = 5$，戦略表の数 $s = 2$ とし，シミュレーション期間 $p = 1{,}000$，試行回数 $tr = 10$ とし，$n = 100{,}007{,}935$ はグループ数を 28,672，$n = 500{,}039{,}679$ のグループ数を 143,360 とした．また，両シミュレーション共に，GPU のブロック数は 56，スレッド数を 128，CPU の並列数を 6 とした．

$n = 500{,}039{,}679$ のシミュレーションでは履歴の長さを $m = 3$ に減らした．分散グリッド同期型の場合は，$m > 3$ ではメモリ不足のため実行できないからである．分散カーネル同期型の場合は，GPU を複数使用することにより $m > 3$ であってもシミュレーションを行えるが，本研究では設定を統一するために $m = 3$ でシミュレーションを行った．

Table 5 から，分散カーネル同期型は分散グリッド同期型に比べ約 15 倍に遅い結果となった．分散カーネル同期型はエージェントの意思決定後，CPU で同期を行う．その後，各グループを選択したエージェントの数を CPU に転送し，履歴を更新する．分散カーネル同期型は毎回 GPU から CPU にデータの転送を行う．CPU と GPU 間のデータ転送速度が高速であれば問題ない．

CPU と GPU 間のデータ転送速度を Fig. 8 に示す．CPU と CPU メモリ（RAM）間や GPU と GPU メモリ（以下 VRAM）間の転送速度に比べ，CPU と GPU 間は非常に遅い．GPU はよく PCI Express で接続される．PCI Express の転送速度は PCI Express の規格とレーン数によって決まる．PCI Express の最新の規格は PCI Express 3.0 であるが，本研究では PCI Express 2.0 である．PCI Express 2.0 の 1 レーンあたりの転送

Table 5: 分散型の処理時間（秒）

	エージェント数	
	$n = 100{,}007{,}935$	$n = 500{,}039{,}679$
分散グリッド同期型	78.2	391.2
分散カーネル同期型	1,223.6	6,076.7
CPU（非並列化）	11,880.7	164,180.0
CPU（並列化）	3,217.7	48,668.5
加速率	41.15	124.4

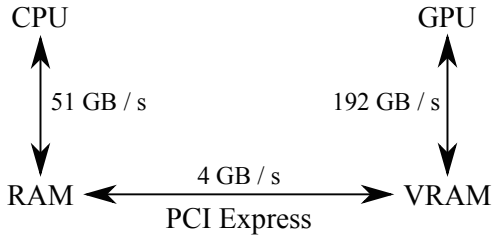

Fig. 8: CPU・GPU 間の転送速度

速度は，1 方向あたり 0.5GB / s である．本研究の接続レーン数は 8 であるため，CPU と GPU 間の転送速度は 4GB / s である．

CPU と GPU 間の転送速度は非常に遅く，毎回各グループを選択したエージェントの数を CPU に転送する分散カーネル同期型はあまり高速化はできなく，加速率は 2.6 倍であった．一方 GPU 内で全ての処理を行える分散グリッド同期型の加速率は 41 倍と独立型（Table 4）と遜色ない結果となった．

独立型のエージェント数 $n = 1{,}048{,}575$ や分散型のエージェント数 $n = 500{,}039{,}679$ において，CPU（並列化）の処理速度があまり高速化できなかった．本研究では，グループ数を GPU の並列数に合わせて設定した．そのため，グループ数が CPU の並列数で割り切れず，グループがスレッドに均等に割り当てることができなかった．また，大規模化したことにより，グループあたりのエージェント数が増加し，各スレッドが処理をするエージェント数に偏りが生じたため，効率が下がったと考えられる．しかし，CPU の並列数を 4 や 8 など，グループ数を均等に割り当てることができる並列数に設定し実験を行ったが，処理時間は並列数 6 と変わらず，グループ数が原因ではないと考えられる．また，分散型の CPU（非並列化）及び CPU（並列化）では，エージェント数を 5 倍に増加させたシミュレーションにおいて，15 倍ほど処理時間が増加しており，本研究のプログラムが CPU に適していないと考えられる．

6 最適化手法

本研究では GPU で高速に処理を行うために様々なプログラムの最適化を行った．これらの最適化は一部を除き CPU で動作するプログラムにおいても有効である．5.3 節では本節の最適化を全て取り入れた処理時間である．本研究で行う最適化手法は CUDA C で行える範囲とし，アセンブリや GPU の世代に合わせた最適化は行わない．GPU の世代毎の最適化を行うときは NVIDIA の Tuning Guide[20, 21] が参考になる．比較を行うモデルは独立型で，シミュレーション設定はエージェント数 $n = 1{,}048{,}575$，履歴の長さ $m = 5$，戦略表の数 $s = 2$ とし，シミュレーション期間 $p = 4{,}096$，試行回数 $tr = 224$ とし，GPU で処理を行う．

Table 6: 分割数を変更した処理時間（秒）

ブロック数	スレッド数					
	1024	512	256	128	64	32
224	76.1	75.6	76.8	81.9	84.9	128.2
112	76.5	75.5	76.6	79.6	85.3	128.0
56	76.4	75.5	76.0	76.4	127.9	232.8
28	76.2	75.4	76.0	128.4	231.3	446.5
14	76.0	75.5	128.4	232.7	444.9	884.6
7	75.9	126.7	232.7	451.1	886.6	1,776.5

Table 7: 分割数を変更した処理時間（秒）

ブロック数	スレッド数				
	16	8	4	2	1
224	250.8	502.7	1,000.3	2,002.7	3,968.3
112	250.6	499.8	997.9	2,003.0	3,973.5
56	457.3	914.4	1,815.3	3,626.9	7,169.2
28	891.3	1,781.8	3,519.4	7,033.9	13,930.4
14	1,764.1	3,531.1	7,000.4	13,996.2	27,705.3
7	3,546.9	7,093.0	14,059.1	28,109.0	55,543.3

6.1 分割数

CUDA ではグリッド及びブロックの大きさを適切に指定することで高速化が可能である．グリッドやブロックの大きさはカーネル関数を実行する際に，関数の引数及びグリッドとブロックの大きさを指定する．グリッドとブロックの大きさ，すなわちブロック数とスレッド数を変化させた場合，処理時間がどのように変化するか観察する．Table 6 及び Table 7 にブロック数とスレッド数を変化させた処理時間を示す．紙面の都合上，Table 6 及び Table 7 の 2 つに分離した．

Table 6 からスレッド数が 128 以上かつ並列数が 7,168 以上（Table 6 の灰色部分）のときに，最も速い処理時間を得られた．Table 6 及び Table 7 からスレッド数が 32 以下のときに，スレッド数を半分にすると処理時間が 2 倍となっている．NVIDIA の GPU では，スケジューラは同じ SM 内の 32 コアに対して同じ命令を発行する．すなわち，32 コアが異なるデータを処理するものの，同時に同じプログラムを実行する．これを NVIDIA では Warp といい，CUDA では Warp 単位で動作するようにプログラムを書く必要がある．

1 スレッドで 1 コアの演算能力を 100%活かすことができれば，同期によるオーバーヘッドを削減できる．しかし，現実的な計算においてコアの性能を 100%引き出すことは困難である．なぜなら，各種演算に比べメモリへのアクセスは非常に遅く，メモリへのアクセス中は演算が中断される．CUDA では Warp がメモリにアクセスするとき，その Warp はメモリアクセスが完了するまで待機状態となる．その間 SM は他の Warp の演算を行い処理速度の低下を防いでいる．このことは，Table 6 のスレッド数が 32 より 64 の方が速いことからもいえる．

6.2 データ構造の最適化

先行研究で我々は MG のデータ構造を最適化し，シミュレーションに必要なメモリを削減することで大規

Table 8: データ構造の最適化による処理時間の変化

	処理時間（秒）
本研究	75.5
先行研究	9,173.7
加速率	121.51

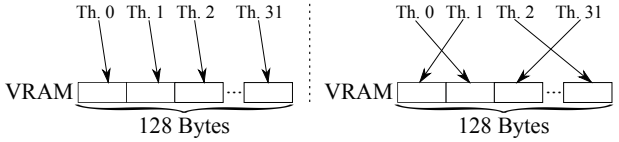

Fig. 9: コアレッシングなアクセス

模なシミュレーションを行った[19]．この最適化は戦略表に着目し，戦略表に必要なメモリ量の削減を行った．エージェントは 2 つのグループを二者択一する．すなわち 1 bit でよい．戦略表は全ての履歴のパターンに対応した戦略が記されている．戦略表 1 つに使用するメモリ量は 2^m bit で済む．

先行研究のデータ構造の最適化はメモリ使用量削減に限ったものである．戦略表のメモリ配置を 3 次元配列にし，履歴を 1 次元目に，戦略表の数 s を 2 次元目に，エージェント数 n を 3 次元目としていた．MG のある 1 期に使用するデータは，その時の履歴 h に対応する s 個の戦略（行動）である．そのため，実際にプロセッサがエージェントの意思決定の処理を行う際には，配列の 2 次元目及び 3 次元目をアクセスするため，メモリアクセスはランダムになり処理速度が低下する．本研究では，メモリアクセスがシーケンシャルになるようにデータ構造の最適化を行った．戦略表のメモリ配置を 3 次元配列にしエージェント n を 1 次元目に，戦略表の数 s を 2 次元目に，履歴を 3 次元目とした．配列の 1 次元目及び 2 次元目をアクセスすることで，ランダムアクセスにならず高速化が期待できる．

メモリ配置の変更による処理速度の変化を Table 8 に示す．データ構造の最適化を行わない場合，最適化を行った CPU（非並列化）の処理時間（Table 4）より遅い結果となった．この原因として，先行研究のデータ構造は VRAM のランダムアクセスが起こることはもちろん，コアレッシングなアクセスができていないからである．コアレッシングなアクセスを Fig. 9 を用いて説明する．Fig. 9 の Th. 0 はスレッド 0，Th. 1 はスレッド 1 である．なお，Fig. 9 は NVIDIA CUDA C Best Practices Guide[22] を参考に作成した．コアレッシングなアクセスとは，Warp のスレッドが一定のメモリ領域にアクセスするとき，そのメモリアクセスをまとめ 1 命令でアクセスすることができる．このとき，Fig. 9 の右のように，連続したスレッドが連続したメモリ領域にアクセスする必要はなく，Warp が一定の領域のメモリにアクセスできればよい．VRAM へのアクセスは低速であるため，VRAM へのアクセスを削減することで，プログラムの高速化が可能である．

6.3 条件分岐の置換

NVIDIA の GPU は Warp 単位で処理の命令が発行されるため，条件分岐などのジャンプ命令は苦手である．Fig. 10 を用いて条件分岐による処理速度の低下を説明する．プログラムは簡単な条件分岐である．変数

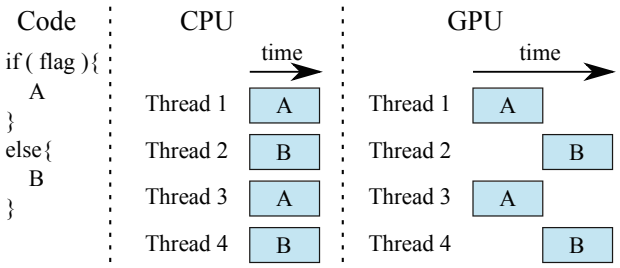

Fig. 10: 条件分岐による処理速度の低下

Table 9: 条件分岐の削減による処理時間の変化

	処理時間（秒）
最適化後	75.5
最適化前	92.7
加速率	1.23

flag が真のときに処理 A を行い，偽のときは処理 B を行う．CPU の各スレッドは命令を発行するスケジューラを持っているため，各スレッドが処理 A や処理 B を行うときでも並列に処理できる．一方，GPU は Warp 単位，すなわち 32 スレッド単位で命令が発行される．そのため条件分岐では，まず処理 A を行うスレッドが動作し，処理 B を行うスレッドは何も行わない．処理 A が終了後，同様に処理 B を行うスレッドが動作し，処理 A を行うスレッドは何も行わない．条件分岐では CPU と GPU のコアが同一の処理速度を持っていたとしても，GPU が 2 倍以上遅くなる．

条件分岐を削減することができれば，GPU でより高速に処理を行える．分岐後の処理が異なる値の演算など 1 命令で済む場合，条件演算子を用いることで条件分岐を削減できる．条件演算子は演算でありジャンプ命令は行わない．そのため GPU でも Fig. 10 のような処理速度の低下は起こらない[4]．また，Warp の全スレッドが条件分岐の後に同一の処理を行う場合は処理時間の低下は起こらない．条件分岐を条件演算子に置き換えるか，Warp の全スレッドが同一の処理を行えるような処理に変更することで，速度の低下が起こらない．

条件分岐の削減による処理速度の変化を Table 9 に示す．条件分岐の削減することで 23%の高速化が行えた．本研究では条件分岐を 2 か所条件演算子に置き換えた．置き換えた場所はループにより多く実行される部分であるため，効果が大きかったと考えられる．

6.4 ループの展開

条件分岐の削減と同様にループを展開することで処理速度の高速化に期待できる．本研究ではビルド時にループ数が決定しており，かつループ数が少ないものをビルド時に展開する．ビルド時のループ展開として C++の template を使う方法もあるが，本研究では C++ライブラリの boost[23] のプリプロセスを用いる．template を用いたループの展開はプログラムが非常に複雑になるため，プリプロセスによるループの展開を行った．

ループの展開による処理速度の変化を Table 10 に示す．ループの展開の有無による処理速度の変化はなかった．処理速度に変化がなかった原因として，コンパイラ

[4]条件演算子の中で関数を呼ぶなど，1 命令で処理を行えない場合はこの限りでない．

Table 10: ループの展開による処理時間の変化

	処理時間（秒）
最適化後	75.5
最適化前	76.1
加速率	1.01

Table 11: コンパイラの最適化の有無による処理時間の変化（ループの展開）

	処理時間（秒）
最適化有効	76.1
最適化無効	117.7
加速率	1.55

の最適化によりループが展開されたと考えられる．そこで，ループのカウント変数の最適化を無効（volatile）にし，処理速度の比較を行う．Table 11 にコンパイラの最適化の有無による処理時間の変化を示す．Table 11 からループの展開により 55%の高速化ができたことがわかる．しかし，Table 10 からコンパイラの最適化によりループの展開が行われている．繰り返し数や繰り返す処理が少ない場合，プログラマがループの展開を行う必要がないといえる．

7 考察

7.1 GPGPU による処理速度の向上

CPU と GPU の各種演算能力を Table 12 に示す．GPU の性能は NVIDIA の Programming Guide[8] より作成した．CPU の演算能力について Intel は公開していないため，CPU の拡張命令である Streaming SIMD Extensions（以下 SSE）命令や，Intel Advanced Vector eXtensions（以下 AVX）命令の演算能力から算出した．SSE は 128 bit のレジスタを用い，32 bit の演算を同時に 4 回行える．AVX は SSE の 2 倍の 256 bit のレジスタを用い，32 bit の演算を同時に 8 回行える．本研究の CPU は，AVX 命令では整数演算を行えないため，Table 12 の浮動小数点演算は AVX を用いた演算能力，整数演算は SSE 命令を用いた演算能力を示した．

MG に用いる演算は浮動小数点演算は一切なく，整数演算のみである．MG では整数演算の中でシフト演算や論理演算が大半である．Table 12 から本研究の GPU の整数演算，特に MG に用いる論理演算やシフト演算は CPU の 2.67 倍である．しかし，CPU と比較し独立型（Table 4）は 40 倍強，分散グリッド同期型（Table 5）は 120 倍強の加速率を得た．

GPU が CPU の演算能力比以上に高速化が実現できた理由として，メモリ転送速度の違いが考えられる．Table 2 より，CPU のメモリ転送速度は 51 GB / s，GPU のメモリ転送速度は 192 GB / s と 4 倍弱の差がある．本研究が提案する全モデルの GPU のメモリコントローラ使用率は，5.3 節の設定では，約 70%であった．メモリ転送速度は 130 GB / s 強となり，CPU のメモリ転送速度[5] は 51 GB / s と，CPU 以上に高速なメモリアクセスが可能である．

[5]GPU のメモリ転送速度は各種モニタリングソフトウェアで取得できるが，CPU の実際のメモリ転送速度は取得できない．本研究では，CPU のメモリ転送速度は理論値である 51GB / s を用いる．

Table 12: CPU と GPU の 1 秒間の演算能力（10 億回）

	GPU	CPU	性能比
単精度浮動小数点（加算・乗算）	2,459.52	307.2	8.01
倍精度浮動小数点（加算・乗算）	102.48	153.6	0.67
32 Bit 整数（加算・減算）	2,049.60	153.6	13.34
33 Bit 整数（乗算）	409.92	153.6	2.67
32 Bit 整数（シフト演算）	409.92	153.6	2.67
32 Bit 整数（bit 反転）	2,049.60	153.6	13.34
32 Bit 整数（論理演算）	409.92	153.6	2.67
比較演算	2,049.60	153.6	13.34

7.2 ブロック間同期によるデッドロック

4.1.1 節では分散型のモデルの説明を行い，ブロック間の同期はデッドロックが発生する可能性があることを示した．分散グリッド同期型はブロック間の同期を行うため，ブロック数やスレッド数を増加させると，デッドロックが起きる可能性がある．5.3.2 節では，ブロック数を 56，スレッド数を 128 でシミュレーションを行った．この条件ではデッドロックは起こらなかった．しかし，ブロック数を 56，スレッド数を 128 以上に増加させたシミュレーションでは，デッドロックが発生した．本研究では GPU 使用率が 0%以外かつ，メモリコントローラ使用率が 0%である時に，デッドロックが発生したと判断した．正常にシミュレーションが行われているときでは，ブロック数やスレッド数を極端に少なくしない限り，GPU はある程度のメモリにアクセスする．メモリに全くアクセスせず正常にシミュレーションを行えるとは考えにくいため，デッドロックの判断にメモリコントローラ使用率を用いた．

NVIDIA はブロック間同期を提供しない理由として，プログラムが使用できるブロック数が減り，効率が下がるとしている．本研究では，使用できるブロック数は減少したものの，効率は上がった．CPU で同期を行う分散カーネル同期型は，MG の 1 エージェントの処理量が少なく，また，全エージェントの意思決定後に各グループに所属するエージェント数を CPU に転送する必要がある．CPU と GPU 間のメモリ転送は非常に遅いため，分散グリッド同期型より効率が下がったと考えられる．

7.3 GPGPU により高速化が期待できるモデル

本研究では ABS のモデルの 1 つである MG を GPGPU を用いて高速化を行った．MG のエージェントの意思決定は環境からのフィードバックのみで，エージェント同士の相互作用はなく，並列化が容易なモデルである．エージェントの意思決定にエージェント同士の相互作用がある場合，並列化する際にはプログラムを工夫する必要がある．エージェントの意思決定に他のエージェントの情報が必要な場合，前期のエージェントの情報を保持したり，同期を行ったりする必要がある．前者はメモリ使用量が増える．GPU のメモリ量は少ないため，シミュレーションに必要なメモリ量が増えることは好ましくない．後者は，同期にはコストがかかる．また，CUDA はブロック間の同期を提供しておらず，自作する場合はデッドロックが起こらないようにブロック数及びスレッド数を設定する必要がある．

また，GPU は 6 章で示した通り，条件分岐やループ

に弱い．条件分岐に関しては条件演算子に置き換えたり，Warp 内のスレッドが同じ処理を実行したりするようにプログラム変更することで高速化が可能である．ループは本研究のようにコンパイラの最適化により展開されることもある．ループが展開されない場合，比較演算とジャンプ命令を繰り返し数だけ行う必要があり，パフォーマンスが低下する．プログラムの可読性は低下するものの，高速化するためにはプログラマがループの展開を行うと確実である．

上記を踏まえ GPGPU により高速化が期待できる処理内容を以下に示す．

- シミュレーションに必要なメモリ量が VRAM 以下である
- メモリアクセスの遅延を防ぐために，グリッドサイズ及びブロックサイズを大きくする
- グリッドサイズを SM の倍数に，ブロックサイズを 32 の倍数にする
- Warp 内の全スレッドがなるべく同一の処理を実行する
- Warp が一定のメモリ領域にアクセスする
- 各ブロックがある程度独立した処理を行う

これらの内容が多く含まれているモデルほど GPGPU による高速化が期待できる．

8 おわりに

本研究では，再現性のある ABS を実行するモデルとして，独立型，分散型の 2 つを提案した．分散型は同期の場所が異なる，分散カーネル同期型と分散グリッド同期型の 2 種類作成した．実験の結果，並列化した CPU と比較し，独立型は 40 倍強，分散グリッド同期型は 120 倍強の加速率を得た．分散カーネル同期型はあまり高速化が行えず，並列化した CPU との比較で 8 倍程度の加速率となり，分散グリッド同期型より低速な結果となった．

本研究ではデッドロックが起こらない設定として，ブロック数を 56，スレッド数を 128 とし，5.3.2 節でシミュレーションを行った．この設定は本研究が使用する GPU に依存する設定である可能性がある．異なる GPU であっても，デッドロックが起こらない設定については今後の課題とする．また，独立型，分散型共に GPU ではシミュレーションの高速化が行えたが，CPU の並列化ではあまり高速化ができておらず，CPU であっても高速化できるようにモデルを改良する必要がある．

近年，Intel 社の Many Integrated Core（以下 MIC）が注目されている．MIC は並列コンピューティング用の演算装置である．2014 年 11 月の TOP 500[24] では，第 1 位のスーパーコンピュータに MIC が搭載されている．今後，再現性のある ABS の高速化について GPU と MIC の比較検討を行いたい．

参考文献

1) Damien Challet，Yi-Cheng Zhang：Emergence of cooperation and organization in an evolutionary game，*Physica A*，**246**，407/418 (1997)
2) 原田拓弥，村田忠彦：マイノリティ・ゲームにおける効率性の周期の分析，計測自動制御学会第 7 回社会システム部会研究会，15/20 (2014)
3) 先山賢一，芳賀博英：GPU によるマルチエージェント・シミュレーション用ライブラリ MasCL の設計と実装，計測自動制御学会第 7 回社会システム部会研究会，39/46 (2014).
4) Kamran Karimi，Neil G. Dickson，Firas Hamze：A Performance Comparison of CUDA and OpenCL，arXiv preprint arXiv:1005.2581 （2010）
5) 荒井勇亮，佐藤功人，滝沢寛之，小林広明：OpenCL による GPU コンピューティングの性能評価，研究報告ハイパフォーマンスコンピューティング，**2010-HPC-124**-11，1/7（2010）.
6) NVIDIA CUDA ZONE，`https://developer.nvidia.com/cuda-zone`.
7) OpenCL - Khronos Group，`https://www.khronos.org/opencl/`.
8) NVIDIA CUDA C Programming Guide，`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.
9) Michael J. Flynn：Some Computer Organizations and Their Effectiveness，IEEE Transactions on Computers，Vol. C-21，948（1972）
10) NVIDIA Optimizing Parallel Reduction in CUDA，`http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`.
11) Makoto Matsumoto，Takuji Nishimura：Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator，*ACM Transactions on Modeling and Computer Simulation*，**8**-1，3/30（1998）
12) Mutsuo Saito，Makoto Matsumoto：Variants of Mersenne Twister Suitable for Graphic Processors，*Transactions on Mathematical Software*，**39**，1/23（2013）
13) George Marsaglia：Xorshift RNGs，*Journal of Statis-tical Software*，**8**-14，1/6（2003）
14) 斎藤睦夫，松本眞：高速並列計算用の状態空間の小さな高品質疑似乱数生成器，研究報告ハイパフォーマンスコンピューティング，**2011-HPC-131**-3，1/6（2011）
15) Pierre L'Ecuyer，Richard Simard：TestU01: A C library for empirical testing of random number generators，*ACM Transactions on Mathematical Software*，**15**-14，346/361（2007）
16) XORSHIFT-ADD (XSADD): A variant of XORSHIFT，`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/XSADD/index.html`.
17) Sebastiano Vigna：Further scramblings of Marsaglia's xorshift generators，arXiv preprint arXiv:1404.0390，1/11（2014）
18) 大矢賢太郎，北田孝典，田中慎一：モンテカルロコードにおける乱数発生方法の研究，日本原子力学会和文論文誌，**10**-4，301/309（2011）
19) 原田拓弥，村田忠彦：マイノリティゲームの大規模化による効率性への影響，第 29 回ファジィシステムシンポジウム，441/446 (2013)
20) NVIDIA Kepler Tuning Guide，`http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html`.
21) NVIDIA Maxwell Tuning Guide，`http://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html`.
22) NVIDIA CUDA C Best Practices Guide，`http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.
23) boost C++ LIBRARIES，`http://www.boost.org/`.
24) TOP500 Supercomputer Sites，`http://www.top500.org/`.