GPGPU を用いるエージェントシミュレーション開発用フレームワークに関する実用化のための拡張

○橋場悠人　佐々木晃（法政大学）

Improving Practicality through Enhancement of a Framework for Developing Agent Simulation using GPGPU

* Y. Hashiba and A. Sasaki (University of Hosei)

概要ー　マルチエージェントシミュレーション（MAS）はエージェント同士の相互作用によって構成されるシミュレーションであるが，大規模化の際は相互作用により実行速度が大きく低下する．GPU を利用し並列実行することで MAS を高速化する言語や環境はいくつか存在するが，これらは利用するプログラム言語と並列技術の学習を要する．先行研究において，我々はこれらの知識を不要としながら MAS の並列実行を可能とするためのフレームワークの開発を行った．これは，Python 言語を利用し，プログラムを並列実行可能な形式に自動変換することを中支とする．本研究ではこのフレームワークの実用性を向上するための再現性，最適化の手法について提案を行い，機能の検証を行う．

キーワード:　マルチエージェントシミュレーション，GPGPU，再現性，最適化

1　まえがき

マルチエージェントシミュレーションはエージェントと呼ばれる個々の存在が他のエージェントに与える影響や，その結果生み出される全体としての現象を観察することができるシミュレーションである．このシミュレーションはエージェント同士の相互作用によって構成される．しかしエージェントの総数が増えると相互作用の発生数は急激に上昇し，実行速度が遅くなる．

既存技術として，ドメイン固有言語やC++言語，Java言語で記述することで，シミュレーションを分散並列化し高速化することが可能なプラットフォームや言語が存在する．これらは言語の習得や並列化の概念の理解を必要とし，容易に利用できるとは言えない．

そこで我々は，Python言語で書かれたプログラムをCUDA C++言語に自動変換することで、高速化と必要な学習量を極力減らしたフレームワークの開発を行った．これにより並列処理やCUDA言語についての知識を必要とせず，学習の容易なPython言語の知識のみでGPGPUシミュレーションの開発が可能となった．しかしGPUを使用したときの結果が逐次実行時と異なる場合がある他，シミュレーション内に記述された処理によっては高速化率が大きく下がる結果となっている．

本研究では前述のフレームワークの実用化に関する機能の強化を目指した．並列化するシミュレーションに対して自動的に再現性を確保する機能や，転送時，実行時の実行時間のさらなる低下のための機能の検討を行った．これら機能の実装に伴い，本フレームワークの構造を見直し，開発効率の強化とCPU実行時の最適化を行った．

2　関連技術

2.1 マルチエージェントシミュレーション

マルチエージェントシミュレーション（MAS）はエージェントと呼ばれる個々の存在が他のエージェントに与える影響や，その結果生み出される全体としての現象を観察することができるシミュレーションである．使用例として，交通システムにおける渋滞の発生や経済現象におけるブームの発生，鳥の群れでの飛行の仕組みを調査するために使用されている．MASでは上記のような現象を引き起こすために，エージェントには簡略化された現実の動作が定義されている．そしてエージェントは定義された動作に従って他のエージェントに影響を与える．これは相互作用と呼ばれ，MASはこの相互作用によって実現される．MASはこの特性のため，エージェント同士の相互作用が処理の多くを占める．しかしこの相互作用の発生回数はエージェントの総数が増えるに従って飛躍的に向上する．そのため，エージェントの個体数を大規模にしただけのMASでも，同等の処理を持つ小規模なMASと比較して膨大な実行時間を有することになる．そこで，相互作用を高速化するための技術として多くの分散システムや並列システムが考案，作成されている．例として，Repast HPCやD-MASON，OpenABL，XAXISが存在する．これらの言語，環境ではMASを分散，並列化することで実行速度を高速化することができる一方で，C++言語，Java言語，X10言語，もしくは独自言語で開発する必要がある．

2.2 CUDA C++言語

CUDA C言語とはNvidia製のGPUでGPGPUを記述するための言語であり，C/C++言語をベースとして拡張した言語である．ソースコードにはCPUで実行する処理とGPUで実行する処理の両方を記述し，コンパイラを通すことでGPGPUプログラムが作成される．このコンパイラはnvccと呼ばれ，CPUで実行される処理はgccを使用してコンパイルし，GPUで実行される処理はGPU用の中間言語に変換する．GPUは通常のメモリアクセスで用いるグローバルメモリのほかにシェアードメモリ，コンスタントメモリ等を持ち，特殊な修飾子を使用することでこれらをプログラムから利用できる．

2.3 PyCUDA[1)]

PyCUDAはPythonのライブラリであり，PythonからCUDAを呼び出すことができるラッパーツールである．これを利用することで，GPU上のメモリの動的な確保やデータの転送，Python上で定義された文字列のnvccでのコンパイルや実行が可能になる．Pythonによる呼び出しを行うため，Pythonの多彩なライブラリを使用することや，スクリプト言語の記述から実行までが比較的容易に行える利点を生かしつつ，画像処理やビッ

PG0001/21/0000-0068 © 2021 SICE

グデータ解析などの時間のかかる処理をGPUの高い並列計算能力を用いて高速化することができる.

2.4 NetLogo[2)]

NetLogoはMAS用のプログラマブルモデリング環境である．Logo言語の派生言語を使用し，少ないコード数でシミュレーションを作成することができる．実行前や実行中にシミュレーションのパラメータをGUIから変更することができ，ユーザは視覚的にシミュレーションの条件を変更することができる．多数のサンプルプログラムが用意されており，ユーザはこのサンプルを用いて独自言語の使い方やシミュレーション，GUIの作り方を学ぶことができる．しかしNetLogoには分散，並列実行用の機能はない.

2.5 OpenABL[3)]

OpenABLはドメイン固有言語（DSL）およびコンパイラであり，様々な分野のシミュレーションで使用することができる．DSLにより，少ない行数のコードでシミュレーションを記述でき，エージェントの並列処理，同期および局所性の記述も可能である.

本研究ではGPUを用いた並列処理を行った上でシミュレーションの再現性を確保するために，いくつかの手法を利用している．その手法の中には，並列実行の順序にかかわらずに書き込みの結果を一定にする手法がある．これはOpenABLではダブルバッファリングと呼ばれ，エージェントのある種類のデータに書き込みと読み込みを同時に行う場合，別に用意されたデータ領域へ書き込むことで全てのエージェントの読み込まれる値への上書きを阻止する手法である.

2.6 ブランチダイバージェンス

本研究ではGPGPUを扱う上で発生するブランチダイバージェンス（BD）の対策についての機能の実験を行った. BDとはGPUカーネルにおいて条件分岐の両方を実行することで起きる問題を指し, BDによって実行時間は増大する．そのため，BDの発生を抑えることでGPGPUによる高速化の効率を向上することができる.

BDは，GPUが実行時にWarpと呼ばれる単位で並列実行することに起因し発生する．Warpには32個のコアが存在し，1コアは1スレッドとして処理を行うため，同時に32コアがSIMD形式で動作する．このとき，条件分岐の際に32コアのすべてが同じ分岐先に分岐しない場合，各分岐先に分岐するコアの処理を全て行うために，各コアは別のコアの分岐先の処理を待機することになる.

既存の対策手法として，Eddy Z.Zhangらの研究[4)]で挙げられている同一の分岐を行うデータを整列される方法や，加藤らの研究[5)]で挙げられているforループの1ループ毎にデータを整列させるかを確認する方法などがある．本フレームワークの先行研究ではEddy Z.Zhangの手法を用いてif文に対してデータを整列する最適化機能を実装している.

3 エージェントシミュレーションのためのフレームワーク

この項では今回の研究によって実装された機能とその目的について記載する．今回作成した機能や実験は先行研究で開発されたフレームワークを対象としており，以前の仕様からの変更点についても記載する.

3.1 GPGPUを用いたエージェントシミュレーションの高速化支援[6)]

この研究ではGPGPUプログラムを容易に記述可能なフレームワークを目標に開発を行った．従来のGPGPUプログラムの開発はCUDA C言語やOpenACC言語，OpenCL等のC言語を基底とした言語を使用することで行われていた．そこで，この研究ではPythonを用いてプログラムの記述を容易にしつつ，GPGPUプログラムへの自動変換機能を用意することで実行速度を向上できるフレームワークを開発した．このフレームワークではPython言語で書かれた逐次処理用のプログラムを並列処理用プログラムへ変換でき，メインメモリとGPUメモリ間の転送処理はフレームワークが自動で行うため，ユーザは知識の習得や記述を行う必要がない．結果として，このフレームワークによって，シミュレーションの種類によって約20~148倍高速化することが可能となっている．また，単にプログラムをGPGPUプログラムに変換して実行するだけでなく，各メモリ間のデータ転送の最適化機能やBDの削減機能が存在し，これにより転送時間を大きく削減することでさらなる処理の高速化を可能としている.

3.2 GPGPUを用いるエージェントシミュレーション開発用フレームワークのデータ転送の高速化と記述性の拡張の研究[7)]

この研究では，3.1. にて作成されたフレームワークに対し，NumbaによるGPGPUを実行する機能，記述性を向上する変数群の実装とその変換機能，PyCUDAを使用したプログラムの最適化, 新たにGPGPU化可能な処理の追加を行った．これにより，シミュレーションが行う処理の多くまたは全てをGPGPU化することができるようになり，シミュレーションのテストを簡単な記述で実行できるようになった．また，定められた記法でシミュレーションを記述する必要があるなかで，直感的に扱える変数やエージェントのリストを使うことで，よりPythonらしい記述が可能になっている.

このフレームワークの欠点としては以下の4つが挙げられる.

(1) 追加されたGPGPU化可能な処理は，Numbaを使用する場合のみ使用可能
(2) 並列化したシミュレーションに再現性がない
(3) シミュレーション中のデータは操作不可能
(4) 実行時はコマンドラインを使用する必要がある

3.3 本研究の手法

本研究では，これまでの研究で開発されたフレームワークに実用化のための機能を追加し，もともとの利点である使用難度の低さについての問題点の改善を行う．今回改善を行った内容は以下の3つである.

(1) シミュレーションの並列化における拡張
    1. 逐次，並列の両実行時の再現性の確保
    2. 高速化，最適化
(2) ユーザの使用性の向上
    1. フレームワークの拡張容易性
    2. サンプルプログラムやエージェントシミュレーション用ライブラリの用意

3. ドキュメンテーションの作成

このうち，(2)の項目2および項目3が使用難度の低減のための実装である．

4 並列シミュレーションの強化

本フレームワークの特徴はPythonプログラムで記されたシミュレーションのGPGPU化であるが，この機能には再現性を喪失する点やフレームワークによる最適化がなされない点があった．今回はそれらの点を補強することで，本フレームワークの実用性や優位性を向上することを目指す．

4.1 並列化シミュレーションの再現性の確保

以前のフレームワークでは CPU と GPU での実行結果が異なることがあるが，その理由は乱数の生成と実行順にある．GPU ではスレッドの実行順が不定であり，CPU で実行した際のエージェントの処理順と同じにならないことがある．この問題によって，乱数生成時のシード値の消費順や，読み書きの両方に使用するエージェントのデータ上書きの順番が実行時に変動する．上記二つの順番を CPU と GPU で同一になるようにした．

これらの再現性の問題は，ユーザが意識してプログラムを記述することで回避可能である．ただしこの問題は GPU のハードウェア的要因に基づくため，本フレームワークの目的である，並列知識の学習を必要としない GPGPU 化機能に合致しない．そこで，上記二つの対策機能を，シミュレーション実行時に使用するコマンドライン引数から指定可能とした．これによりユーザは CPU と GPU 間の再現性に意識することなくプログラムを作成することができる．

4.1.1 乱数生成器とシード値

乱数の生成において，各エージェントが固有の乱数のシード値を持つようにし，乱数生成器についても Python と CUDA C++で同一のアルゴリズムを使用するようにした．乱数生成器のアルゴリズムには線形合同法を採用し，乱数生成器の検証の論文[8]から次のシード値を設定するアルゴリズムを引用している（Fig. 1）．これにより，生成される乱数はエージェントに基づくことになり，実行順や時間による影響を受けない乱数の生成が可能となる．このときの各エージェントが持つ初期シード値はユーザによって指定されたシード値を元に作成するため，ユーザが指定することで固定することができる．また，ユーザが記述するエージェントのプログラム内で random ライブラリを使用する場合でも，本フレームワークがそれらプログラムの読み込み時にシード値を設定するため，シミュレーション開始前での乱数生成においても固定されるようになっている．また，今回の作成した乱数生成器はユーザがエージェントのプログラム内で呼び出せるようにしており，後述の自動変換機能を使用しない場合でも使うことができる．乱数の周期は現在 2^{24}-1 となっており，エージェントがシミュレーション内で周期を一周することはほぼ無いと考えている．しかしシード値を各エージェントがそれぞれ持つことに起因し，エージェントの個体数が増えるにつれて，あるエージェントの乱数列が別のエージェントの乱数列と重なることが考えられる．例として，エージェント A が初期シード値として n を持つとき，別のエージェント B が乱数生成を繰り返し現在のシード値が n となると，これ以降エージェント B は初期のエージェント A とまったく同じ乱数を生成し続けることになる．この対策として生成する際に乱数列をずらす方法をとった研究[9]があるが，本研究ではずらしによる実行時間増加の問題を解決できていないため，現在対策していない．また，今回は線形合同法を採用したため，シード値が 0 に設定されると乱数生成器として機能しなくなるという問題もある．

4.1.2 並列時のエージェントのデータ上書き操作

GPGPU 化したプログラムの一つの関数内において，各エージェントがある 1 つの属性の値に対して，他エージェントのその値の読みこみと自身のその値に対する書き込みを行うとき，Fig. 2 のようにエージェントの実行順によって差が生じることがある．これは逐次実行時は常に起きていることであるが，エージェントの実行順が変わらなければ結果は一定になり問題とならない．ただしこの処理を並列化する場合は実行するエージェントの順が一定とならず，スレッドごとの処理速度等により結果が変わることになる．そこで，並列実行時の結果を同一とし，かつ，Python のみでの実行時の結果とも同一とするために OpenABL にてダブルバッファリングと呼ばれている処理を適用する機能を作成した．このダブルバッファリングとは，(1)上記の他エージェントの属性の読み込み先のデータと自身の属性への書き込み先のデータを別のデータとして 2 つに分ける，(2)対象となる処理を全エージェントが終了した後，全エージェントで同時に読み込み用のデータに書き込み用のデータを上書きする，という 2 プロセスからなる方法である．これにより並列実行時に，実行順と関係なく書き込まれる前の値を常に取得することが可能となる．その後の上書き処理によりエージェントは他エージェントの処理に影響を与えずに値の更新を行うことができる．しかし，前述のとおり逐次実行時は通常，自身より前に実行されたエージェントのデータは更新後の値，実行前のエージェントのデー

```
def get_unif_rand(data):
    now_seed  = data["__seed"]
    next_seed = (now_seed * 48271) % 16777215
    data["__seed"] = next_seed
    return (next_seed % 1000) / 1000
```

Fig. 1: ユーザによる属性の値の範囲の指定方法.

タは更新前の値となる前提で処理を行うため，このダブルバッファリングを適用した GPGPU プログラムと逐次実行時の結果を同一のものとするには，逐次実行時もこの処理方法を適用する必要がある．

このダブルバッファリング手法では，上記の書き込みと読み込みが行われる属性の分だけ新しく書き込み用の属性を追加する必要がある．そのため，GPGPU 化する際にメインメモリと GPU メモリでやり取りするデータ量が増えてしまい，転送にかかる時間が増えることとなる．また，書き込み用の値を読み込み用の値に更新する処理は，他エージェントのデータを読み込む処理が全てのエージェントで終了した後でなければならず，同期処理が発生する．そのためこの一連の処理を適用すると，適用しない場合と比較し全体の実行時間にいくらかの遅延が生じることになる. ただし，同期後の上書き処理に関しては基本的に全て GPGPU 化可能であるため，この処理を GPGPU 化することで遅延を削減することが可能である．

4.2 並列化シミュレーションの高速化，最適化

本研究では並列実行を最適化する手法として，GPU 上のデータを配列として持つ機能を作成した．また，データ転送処理に関する2つの最適化機能の追加を行った．さらに実行時と転送時の両方の最適化を行うデータのパック機能について実験を行った．

まずデータの配列への変換による高速化について記述する．以前のフレームワークではPythonプログラムの変換時にPythonの辞書をCUDAで実装し使用していた．そのため，文字列のマッチング処理が発生していた．そこで今回は最適化オプションとして，エージェントのデータをPythonでは辞書として扱いCUDAでは配列として扱うことで，ユーザのプログラムを変えずにGPU上での処理を高速化する機能を実装した．

データ転送に関しては2つの最適化機能を実装した．データを一括で転送することによる最適化について，以前はPyCUDAでは使用できなかった一括転送の最適化機能を使用できるよう変更した．これにより，以前のフレームワークに存在する，GPUとメインメモリ間の転送の自動省略機能と併用することが可能になり，さらなる転送時間の軽減が可能となった．

データのパック機能について記述する．この機能ではGPU上のエージェントのデータを，いくつかの属性の値を一変数としてまとめた状態で保持する．この圧縮方法は，エージェントのそれぞれのデータが必要とするビット長を計算し，32ビットの変数上に配置しなおすことで実現する．このとき，圧縮されたデータを

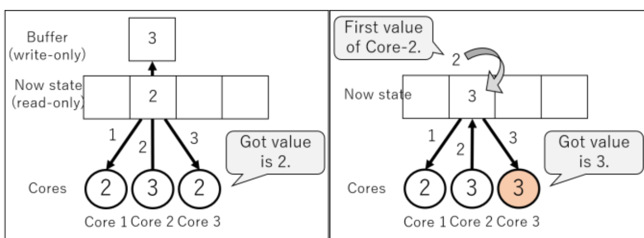

Fig. 2: ダブルバッファリングによる値の同一性.

復元できるように，ユーザはエージェントの各データの値が取りうる範囲を指定する必要がある（Fig. 3）.

この手法の利点として，エージェントの全属性を少ない変数量で表現することで全体のデータ転送量を削減することが挙げられる．さらに，この機能ではforループを使ってエージェントのデータを取得する処理を自動変換時に最適化する．これにより実行時間も削減することも利点として挙げられる．

このforループの最適化は以下2つの手法を適用することで得る．一つはグローバルメモリからのデータ取得時にコアレスアクセスとすることでプログラム全体でのグローバルメモリへのアクセス回数を削減する手法(1)，もう一つはグローバルメモリから得られたデータをシェアードメモリへ格納することで次回以降のデータアクセスを高速化する手法(2)である．

(1)について解説する．GPUはグローバルメモリへのアクセスの際，コアレスアクセスによりデータを最大128ビット分まとめて取得することができる. 本フレームワークではGPU上のエージェントのデータはグローバルメモリに置かれる．通常はデータを一エージェント分ずつアクセスするが，本手法によって複数のエージェントのデータをまとめて取得することで，レイテンシの高いグローバルメモリへのアクセス回数を抑制し，全体の実行速度を向上する．さらに，エージェ

```
def get_packing_info(agent_max_size, loop_max_size):
    return {
        "is_infected": "0~1",  # 0 ~  1 (1 bit)
        "count": "0~31",       # 0 ~ 31 (5 bit)
        "company_id": "0~9",   # 0 ~  9 (4 bit)
        "school_id": "0~7",    # 0 ~  7 (3 bit)
    }
```

Fig. 3: ユーザによる属性の値の範囲の指定方法.

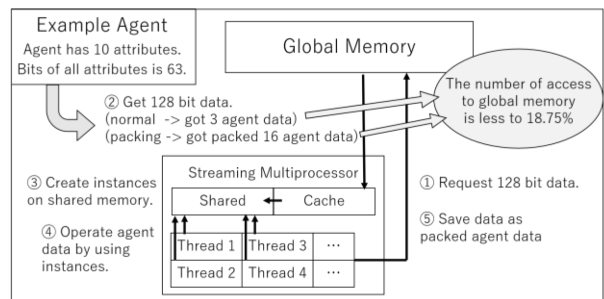

Fig. 4: for ループにおけるデータ取得の効率.

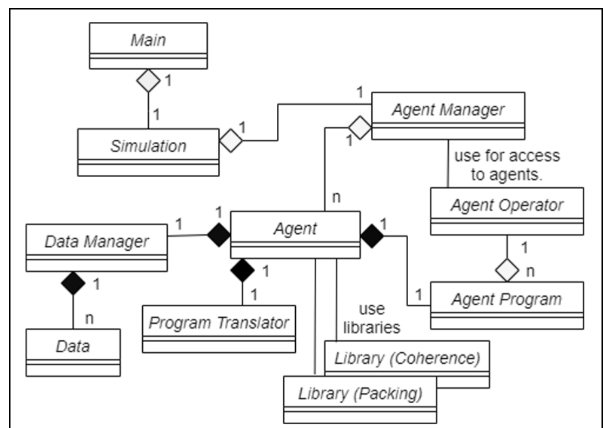

Fig. 5: 本研究での再構成後の設計.

ントのデータをパックする都合，一度のアクセスで得られるエージェントの数が向上することが見込める．この理由は，例としてFig. 4のようにエージェントが属性を10個持ち，これらの属性が2変数（8バイト）分に圧縮可能である場合，同時に128ビットのデータを取得すると，圧縮しない場合は3エージェント分のデータを，圧縮する場合は16エージェント分のデータをまとめて取得できることによる．これらのコアレスアクセス化とパックによるデータの圧縮の2つによってグローバルメモリへのアクセス時間を削減することができる．

(2)について解説する．グローバルメモリから取得したエージェントのデータは，通常の変数操作が可能なデータが格納された配列の状態に復元し，シェアードメモリに格納する．その後，同一ブロック上のスレッドはこのシェアードメモリを用いて他エージェントのデータにアクセスする．これにより次回以降のアクセスではグローバルメモリへのアクセスが必要なくなる．

エージェントのデータに書き込む際はカーネル終了時のデータをパックしグローバルメモリへ書き込むことで完了する．このとき保存するのは各スレッドが処理に使用する自身のデータのみで，他エージェントのデータの保存は行わない．これは，本研究での並列化を行ったうえで他エージェントのデータへの書き込みを行うと基本的にデータハザードが発生するためである．

この機能では，for文を使用したエージェントのデータへのアクセス処理からグローバルメモリへのアクセス回数が削減される代わりに，シェアードメモリ上でのエージェントのデータ配列の作成とシェアードメモリへのアクセス，エージェントのデータの圧縮と保存処理が新たに発生する．しかし各メモリのアクセス速度の違いにより，グローバルメモリへのアクセス回数を削減できるほど高速化可能と考える．

5　フレームワークの実装とその改良

本フレームワークの実用性の向上に先立ち，フレームワーク内部の構造について再構成を行った（Fig. 5）．これにより，各種機能の実装の簡易化，影響範囲の縮小化の効果が得られた．以下では，先に本フレームワークの動作の流れや主な機能について解説し，その後再構成に伴う新たに作成された機能について解説する．

このフレームワークはユーザによって記述されたエージェントの定義が書かれたプログラムを読み込み，シミュレーションを進行する．ユーザは基本的にエージェントの振る舞いと描画の2つを定義することができる．振る舞いの定義では，処理内容によって関数を分割することでGPGPU化する箇所を選択できる．描画の定義は必ずしもする必要はないが，座標と色を定義するだけで作成することができる．本フレームワークはこれらのプログラムを読み込み，各エージェント種の決められた順番に沿って振る舞いの処理を実行し，指定された数の分のループ処理を行う．ユーザはシミュレーション開始時にループ回数のほか，フィールド（座標）の大きさ，乱数の初期シード値，描画の有無，描画用の待機時間等を設定できる．また，プログラムのGPGPU化を行うときは，本フレームワークはPythonのプログラムの構造を解析し，同一の構造のCUDA C++言語のプログラムを生成する．このプログラムをGPU上で実行するには，エージェントのデータをGPUへ転送する必要があるが，この転送もフレームワークが必要な場合に判断し自動で転送を行う．そのため，ユーザはCUDA C++の言語やCPU，GPU間のデータ転送処理に触れることなくシミュレーションのGPGPU化が可能となる．

新たに作成された機能は大きく分けて二つ存在する．
(1) シミュレーションのテスト用ファイル，コマンド
(2) フレームワーク内の処理で利用されるデータの設定ファイル

(1) について，新たにシミュレーションのテスト用のファイルを作成した．Pythonでの実行となるため，このファイルから実行することでシミュレーションのテストを開始でき，ここで2つの新たな機能が利用可能となる．一つはテストの実行回数の指定機能であり，もう一つはシミュレーションの各ループ内でかかった時間をJsonファイル形式で出力する機能である．

また，平均実行時間の計測を自動で行う機能，実行時間の最大最小値を任意に除外する機能を追加した．これらを利用することで，指定回数分の平均実行時間の取得，複数のシード値でのシミュレーションの正当性の検査が容易になる．

(2) について，新たにフレームワーク内で利用されるデータの設定ファイルを追加した．本フレームワークでは描画処理をユーザが上書きすることができ，tkinterライブラリを使用する元から用意された処理から変更することができる．ただし，このメソッドがループ内で必ず実行するプロセスとして認識される問題があったため，この設定ファイルからユーザがメソッドの除外設定を行えるようにした．この他に，フレームワークによって定義されるエージェントの内部データの名前を定義することができる．現在はエージェントのIDとシード値がある．またユーザが内部データを追加する場合も使用することで簡潔に記述できる．

今回の再構成に伴うモジュール分割により，Pythonでのシミュレーション実行速度が大きく低下した．そこで，以前まではエージェントのデータ操作に必要だったラッピングを外すことで高速化した．結果として，現在のPythonでの実行速度は，以前のフレームワークの約2倍の処理速度となっている．

6　ユーザ利便性に関する実装

本フレームワークの利点の一つであるユーザによるフレームワークの改変の難易度の低下や，初めて使う利用者によるこのフレームワークの学習の難易度の低下を目指し，設計の改変や機能追加を行った．

6.1 サンプルプログラムの実装

本研究ではフレームワークに使用可能なサンプルシミュレーションプログラムを作成した．今回作成したシミュレーションはスポット型の感染症シミュレーションである．スポット型はエージェントが座標を持つシミュレーションとは違ったプログラム構造を持つ（Fig. 6）．スポット型ではエージェントが所属するスポットごとに固有の処理が定義されることが多く，処理の開始直後からスポットごとに処理内容が分かれることも多い．このような処理の別れ方をするシミュレーションはGPGPUではBDを発生させやすく，スポットの個数が多いほどワープ内の各スレッドが別の分岐をしやすく，実行時間が増えやすくなる．そのため

GPGPU化の恩恵を受けにくいシミュレーションの一つと考える．本研究ではこのシミュレーションを新たに実行速度計測にて使用し，機能の検証を行うとともにスポット型シミュレーションのGPGPU化の適正について考察する．

6.2 ドキュメンテーションの用意

本研究では、フレームワークを利用するにあたってのドキュメンテーションを用意する。なお、現段階では、そのうちの一部のみの情報が利用可能となっている。以下の二つの役割を目指す．一つは，ユーザが初めて本フレームワークを扱う際のプログラムの生成，記述，実行について解説することである．本フレームワークではPythonの標準ライブラリではないライブラリを使用するためそれらのインストール手順について記載する．MASでは基本的にループ処理があるが本フレームワークが要求するエージェントのプログラムではループ処理は書く必要がない．こういった記述の必要な処理，フレームワークが行う処理を明確に分類し解説する．シミュレーションの実行時にはコマンドラインから呼び出す必要がある．そこで実行時のコマンドライン引数，また本研究にて追加されたテスト用コマンドについて解説を行う．

もう一つの役割はGPGPUプログラムに変換する際に発生するエラーや，シミュレーションの挙動の差異の原因を探索するための知識を提供することである．本フレームワークはPythonのプログラム構造をそのままCUDA C++へ変換するために，変換後処理内容が食い違うことがある．これはPythonとCUDA C++の言語仕様による．例としてはfor文のインクリメント変数への操作や型のバイト長による上限値，演算子が予測する値の型等がある．これらの差異は，ユーザがエージェントのプログラムをPythonのプログラムとして記述すると正しくシミュレーションをGPGPU化できない可能性を示す．これらの差異に対するデバッグにはC言語やCUDA C++の挙動を学習する必要があり，本フレームワークの指針と異なる．そこでこれらの差異と回避方法を，ドキュメンテーションやサンプルプログラムを利用する形でユーザに対して提供することで，デバッグのための学習時間を削減する．

7 評価

この項ではこのフレームワークを使用し，作成した機能による速度効率についての計測と並列化時の再現性の検証を行い評価する．

Table 1: 実験環境

OS	Ubuntu 18.04
CPU	Intel Core i7-10700
GPU	GeForce RTX 3070
メインメモリ容量	8 GB
Python version	3.7.8
PyCUDA version	2020.1
CUDA Version	11.2

7.1 実行速度の評価・考察

実行速度の評価では，Table 1に示す環境にて計測を行う．計測では本フレームワークを使用してGPGPU化したシミュレーションに対し，インデックスアクセスに変換する機能，再現性を確保する機能，データをパックする機能を使用した際の実行速度を比較する．この計測に使うシミュレーションとして、(1)感染症，(2)食物連鎖，(3)スポット型感染症の三つを使用する．シミュレーションではユーザが定義した描画を行うことができ，実験で使用するサンプルプログラムにも描画用の関数を用意してあるが，本計測では描画処理は行わず，実行時間に含めない．

7.1.1 各シミュレーションの特徴と使用方法

感染症シミュレーションについて解説する．本研究で使用する感染症シミュレーションは，NetLogoにてサンプルプログラムとして作成されているプログラムの挙動を再現したものである．このシミュレーションの主な挙動は，同一座標に存在する他のエージェントが感染している場合に自身を感染させるというものである．計測ではエージェントの個体数を1000から10000まで1000ずつ増やし実行する．

食物連鎖シミュレーションについて解説する．このシミュレーションも感染症シミュレーションと同じくNetLogoを参考に作成したプログラムである．主な挙動は狼と羊が同一座標に存在する場合に，狼は羊を捕食し，羊は捕食され削除されるというものである．このシミュレーションで発生するエージェント数の増減処理はGPGPU化できないために，シミュレーションの規模が大きいとき，全体の処理速度はPythonの処理速度に大きく影響を受け低速化する．そのため本研究では各機能の効率を検証するためにGPGPU化可能な処理の実行時間のみを抜き出し計測する．計測では50x50の座標内に羊が200体，狼が100体いるとして，人口密度を保ちつつ座標を50から250まで50ずつ増やす．

スポット型感染症シミュレーションについて解説する．このシミュレーションは感染症シミュレーションをスポット型に適用したもので，エージェントは別の感染者のエージェントと同じスポットにいるときに低確率で感染する．計測では感染症シミュレーションと同様に個体数を変化させて実行する．

7.1.2 再現性とインデックスアクセスによる速度効率

再現性を保つ機能は，プログラム内で乱数生成を行う場合はエージェントにシード値を持たせ，ダブルバファリング機能によって，上書きが発生する属性がある場合は書き込み専用の属性をエージェントに持たせる．そのためエージェントは多くの場合，再現性のために属性を多く持つことになる．また，ダブルバッファリングでは読み込み専用の値を書き込まれた値に更新する必要があり，このための処理が新たに生成される．こういった取り組みにより，再現性を保つ場合の実行速度は再現性を保たない場合に比べて低下する．

インデックスアクセスではデータアクセスを配列でのランダムアクセスとすることで文字列操作を撤廃し，ポインタ経由回数を削減している．これにより他エージェントのデータへのアクセス回数が多いほどこの機能による効果が表れやすいと考える．本項ではこの二つの機能を使用した場合に各シミュレーションの実行速度がどう変換するかの実験結果を提示し，考察する．

それぞれの結果はFig. 6, Fig. 7, Fig. 8のとおりである．すべてのシミュレーションで再現性を保つことで保た

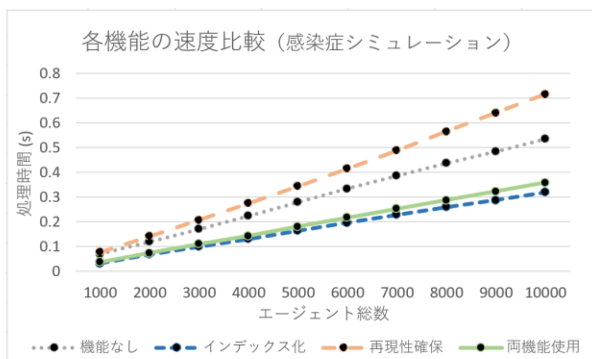

Fig. 6: 感染症シミュレーションと 2 機能.

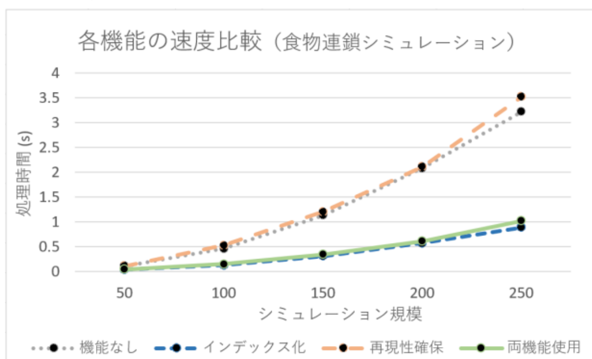

Fig. 7: 食物連鎖シミュレーションと 2 機能.

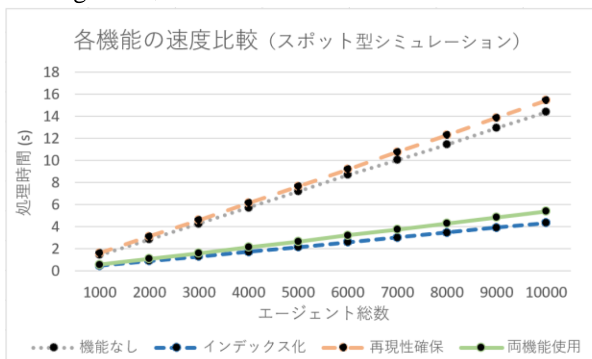

Fig. 8: スポット型シミュレーションと 2 機能.

ない場合に比べ若干の速度低下が見られ，インデックスアクセスを利用することで最大約3倍の高速化が可能となっている．インデックスアクセスによる効果は食物連鎖が一番高く，次いでスポット型となっているが，これはプログラム内部の他エージェントの属性にアクセスする処理の割合の高さに比例していると考えられる．感染症シミュレーションのプログラムでは分岐処理でのショートサーキットが効く構造になっており，プログラムの内容はスポット型と変わらないものの，他エージェントの属性へのアクセス機会が低くなっていたことが原因と考える．

7.1.3 パック機能による最適化

パック機能の検証では3つのテストを実施した.一つ目では後述の分岐処理部分を調整した感染症のシミュレーションを使用した．二つ目はグローバルメモリへのアクセス回数が正しく減っているかを確認するだけのテストプログラムを使用し，三つ目は自動変換後のプログラムを手動で最適化したスポット型シミュレーションを使用した．最適化では，(1)エージェントの属性の圧縮と解凍の際に発生していた関数呼び出しとif分岐の削除，(2)エージェントの属性の数だけ行っていたグローバルメモリへのアクセス処理を1回のみに削減，(3)乱数生成処理とデータ解凍処理内の%記号を用いた余剰演算をビット演算で代用，の3つを行った．

シェアードメモリにデータを格納する処理の都合上，スポット型シミュレーション等のforループによる他エージェントの探索よりも先に分岐があるプログラムでは正しく動作しない．そのためforループ処理からbreak文を削除している．このテストのスポット型シミュレーションではスポットによる分岐ができないため，この分岐がない一部の処理を対象に変換しており，計測の実行時間効率には変換できない箇所は含んでいない．実行時間のほか，GPUへのデータ送信処理，GPUからのデータ受信処理についても時間効率を検証した．

結果はFig. 9, Fig. 10, Fig. 11の通りである．感染症シミュレーションでは実行速度が約0.5倍となっており，データ送信処理の速度に関してもデータをパックしない場合と比べてほぼ変わらない結果となっている．グローバルメモリへのアクセスだけのテストプログラム（Fig. 10）では，実行時間効率が約2割ではあるが向上している．そのためグローバルメモリへのアクセス量自体は最適化によって低減できていると考える. また，データの受信処理については約2倍の効率で転送ができている．Fig. 11では実行速度が約7.6倍向上，転送速度はほぼ等倍, 受信速度は約1.4倍から2.9倍へ高速化していることが分かる. 上記の最適化を施した際はFig. 9での結果とは異なり実行時間の効率が上がることが分かった．受信時間の効率はエージェントが増えるにつれ上がっており，Fig. 9と似た結果となった．送信時間の効率もFig. 9と同様となった．

7.2 再現性の評価・考察

本研究では再現性を保つ機能が正しく動作するかについて3つのシミュレーションを使用して検証した. シミュレーションの規模は今回計測に使用した中でそれぞれ最小のものを使用した．30ループ目にIDの若い順から100体分のエージェントのデータを出力し，CPUとGPUで同一か確かめることを，乱数の初期シード値を5回変更して確かめた. また，この評価では今回検証に使用したPCとは別のPCも使用し，2環境で検証を行い，CPUとGPUの結果が同一となったため正しく機能していると考える．もう一つのPCではGPUとしてGeForce GTX 950を使用しており，別のGPUを使用した際も再現性が保たれていることが分かった．

8 結論

本研究では，シミュレーションの並列化機能の強化とユーザの使用性を向上するための取り組みを行った. 並列化機能の強化では，再現性を保つ記述に自動で変換する機能や，並列化時の処理を最適化する機能を作成した．再現性を保つ機能では同一の結果を出力することに成功した．この機能を使用すると実行速度が下がってしまうが，最適化機能であるインデックスアクセス化機能と同時に使用することで，元の実行時間よりも短くすることができた．パック機能については，データの扱い方の構想は良く，効果があることが分かったが，本研究で用意した変換機能を使用すると冗長な処理によって実行速度が低下するため，変換機能の

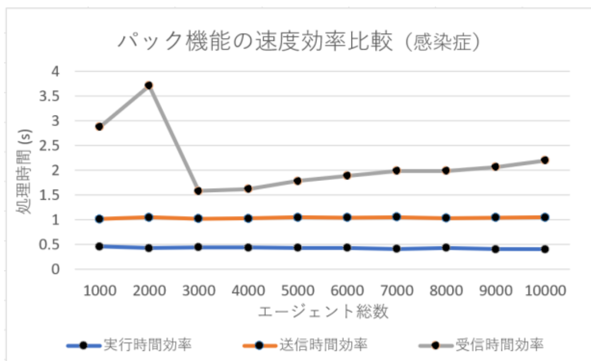

Fig. 9: パック機能使用時の感染症の効率.

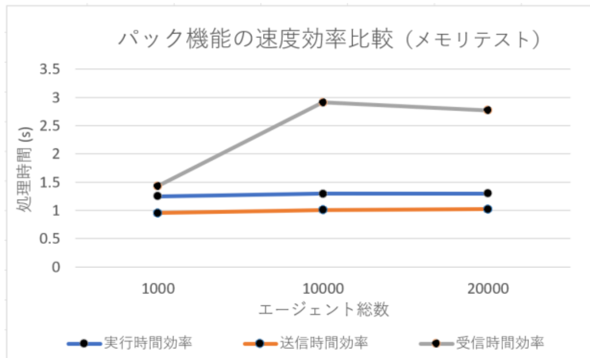

Fig. 10: パック機能使用時のメモリテストの効率.

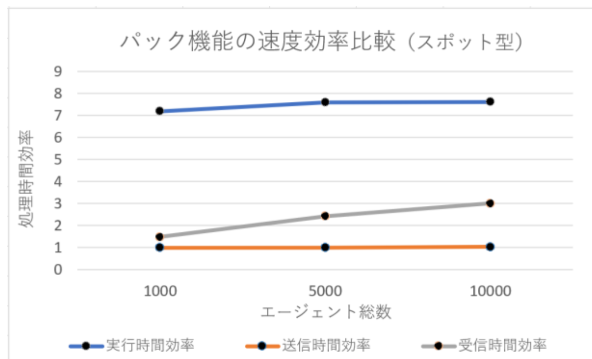

Fig. 11: スポット型シミュレーションを使用した最適化後のパック機能の速度効率.

改善は必要非可決である．

スポット型シミュレーションはスポット数が多いほどBDを発生しやすく，これを解決しない限りGPGPUには向いていないが，インデックスアクセスによる最適化は効果があることが分かった．ハードウェア資源に関する最適化はその資源を多用するプログラムほど効果があるため，最適化手法と適合するようなプログラムで定義されたシミュレーションであれば，今回使用したスポット型のシミュレーションであっても最適化が十分効果を発揮する．

また，本研究にて開発したパック機能を使用することで，長さ制限があるがエージェントの属性として配列や文字列を利用できるようになると考える．現在はエージェントのデータの転送時に，float型の値としてすべての属性をまとめた配列を作成し転送している．そのため型が違う値はGPUへは一度に転送できないが，すべての値をビット列で表現し実行時に元の型の値に変換する方法であれば値を元の型に復元する処理を経ることで可能になると考える．文字列を表現する場合は，辞書のように文字を数値で表現すること[10]によって，単純に文字コードの通りのビット列とした場合と比べて小さいビット列で表現することができ，データの圧縮率を向上させることができると考える．

本フレームワークはエージェントが保持する属性のデータの分だけVRAM容量を占有する．例として，各エージェントが属性を10個持ち, 合計100万体存在する場合，エージェントのデータだけで40MB使用することになる．さらに，再現性を確保する機能を使用するとシード値を持つため，全エージェントが一つ分多く属性を持つ．また，エージェントが属性として配列を持つ場合，現在は配列の要素の数だけ属性の確保を必要とする．本研究の実験で使用したGPUは8GBの容量を持ち，40MBであれば使用率は全体の0.5%であるが，より大規模のシミュレーションや別条件のシミュレーションを同時実行する場合は，より多くの容量が必要となることが考えられる．本研究のパック機能では各属性のとりうる範囲によるが, 2値しかとらないデータであれば32倍に圧縮可能であり，この機能を適用することでGPUメモリを効率良く使用することができると考えている．

謝辞
本研究はJSPS科研費18K11247の助成を受けたものです．

参考文献

1) A. Klockner, et al.: PyCUDA: GPU Run-Time Code Generation for High-Performance Computing, Parallel Computing, **38**-3, 157/174,（2012)

2) S. Tisue, U. Wilensky: NetLogo: Design and Implementa-tion of a Multi-Agent Modeling Environment, SwarmFest (2004)

3) Biagio Cosenza, et, al.: OpenABL: A domain-specific language for parallel and distributed agent-based simulations, Euro-Par 2018, 505/518 (2018)

4) E. Z. Zhang: Streamlining GPU Applications on the Fly - Thread Divergence Elimination through Runtime Thread-Data Remapping, the 24th ACM International Conference on Supercomputing, 115/126 (2010)

5) 加藤 誠也 : GPU におけるダイバージェンス削減による高速化手法, IPSJ SIG Technical Report, (2012)

6) 鎌田 知也 : GPGPU を用いたエージェントシミュレーションの高速化支援， 法政大学 2017 年度修士論文.

7) 橋場 悠人，佐々木 晃，鎌田 知也 : GPGPU を用いたマルチエージェントシミュレーションの開発向けフレームワークの研究，第 18 回社会システム部会研究会, L2-2, (2019)

8) Stephen K. Park, Keith Willam Miller: Random Number Generators: Good Ones Are Hard to Find, Communications of the ACM, **31**-10, 1192/1201 (1988)

9) 原田 拓弥，村田 忠彦 : General-Purpose Computation on GPUs を用いた再現性のある Agent-Based Simulation の高速化，第 8 回社会システム部会研究会，研究発表 6, (2015)

10) 若月 駿尭，欅 惇志，宮崎 純: 効率的なテキスト処理を目指した簡潔データ構造を用いるトライ木の GPU 上での実装，DEIM Forum, (2016)