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

○橋場悠人　佐々木晃（法政大学）鎌田知也（株式会社フィックスターズ）

A Study on Framework for Developing Multi Agent Simulation Using GPGPU

* Y. Hashiba and A. Sasaki (Hosei University), T. Kamata (Fixstars Corporation)

概要ー　本研究はマルチエージェントシミュレーション（MAS）の実行速度向上が目的である. GPU を処理に用いると処理を高速化することができるが, アーキテクチャや言語等の専門的な知識が必要になる．そこで，本研究では GPGPU 特有の命令を使わずに GPGPU をエージェントシミュレーションに適用できるフレームワークを開発した．これは GPGPU の記述に自動で変換することで実現する．また，実行時の最適化支援として，ブランチダイバージェンスの削減を行う.

キーワード: Python，GPGPU，エージェントシミュレーション，自動変換

1　まえがき

マルチエージェントシミュレーション（MAS）は複数のエージェントを用いて行い，発生するエージェント同士の挙動や相互作用を実験するものである．これにより交通渋滞や株価の動向，自然現象などの複雑なふるまいを観察することができる. しかし MAS では，その性質上エージェントの数が多くなると計算量が増え，実行速度が低下してしまう．そのため実行時間を現実的な時間内に収められるモデルの規模や種類は限定されてしまう.

この問題を解決する方法として，GPGPU を用いて実行速度の低下を緩和することが考えられる. GPGPU とは画像処理用のデバイスである GPU を汎用計算に利用する技術であるが，しかし，これを扱うには GPU の動作や並列処理の知識が必要となり，実装は容易ではない.

本研究では，Python 言語による MAS 開発用のライブラリを実現し，このライブラリによって作成したシミュレーションプログラムを GPGPU に自動変換する方法をとる. これにより, GPGPU についての深い知識を必要とせず，シミュレーションの高速化が図れる. GPGPU は通常 C/C++の低レベルな処理として記述されるが，そのような記述が不要になり，記述量の減少と可読性の向上を得られる．また，Python 言語での開発であるためシミュレーションのデバッグを Python にて行うことができる．本研究で実現した方法は，Python で記述した MAS プログラムを，CUDA-C 言語によるプログラムに変換する方法，また，Python 向け GPGPU ライブラリ Numba を使った処理に変換するものである. 本稿では, GPGPU に変換するための記述方法の記述性，GPGPU 化による高速化の議論や考察について述べる．さらに，GPGPU では，分岐処理を持つ並列処理では，処理能力を低下させるブランチダイバージェンスが発生するが，本研究ではエージェントの並べ替えを行うことでこの問題に対する対策を試みる.

2　研究背景・目的

2.1 研究背景

商業施設の顧客の流れや災害時の住民の避難経路を模索するために MAS を利用することがある．近年では施設の増築や駅の再開発により規模の大きいシミュレーションでの実験が求められている. ところで MAS は規模が大きい，エージェント数の多いモデルではエージェント同士の相互作用が多く発生するため，そこにかかる時間が多くなり，実行時間が飛躍的に上昇してしまう．MAS の開発環境の中には，スーパーコンピュータを扱うものがあり，これにより実行時間の短縮を図るものもあるが，エージェント数が数万を超える場合は CPU での実行コストが非常に高くなる. 本研究ではこの対策として GPGPU を利用し，規模の大きいシミュレーションの高速化を図る．また，シミュレーションのプログラミングの経験がないエンジニアでは効率のいいプログラムを記述できないことがあり，これによって実行時間が増える可能性もある．そこで，今回は近年学習用としても使われ，使用する現場や開発者が増えてきた Python 言語を使用して記述できるようにする．本研究では GPGPU の記述方法を隠蔽し Python の構文を利用してシミュレーションの開発を行うことができるフレームワークを作成する．これによって記述性と高速化を両立できると考えた.

2.2 本研究の目的

本研究の目標は，GPGPU を使って高速化できる MAS のプログラムを, フレームワークの利用者が容易に開発できるようにすることである．そのため本研究では以下の二点を重視して開発を行う.

- 処理の GPGPU 化による高速化
- GPGPU を扱う処理の記述の容易化

また, 本研究では最適化支援として, GPGPU の開発で発生するブランチダイバージェンスの対策を行う. GPGPU の記述を利用者に隠蔽するためにフレームワークの機能として実装し，この機能の使用を利用者が実行時に選択できるようにしている.

本研究は 3 節で説明する Repast HPC[3] などのエージ

ェントシミュレーションの並列化フレームワークと比べ, 記述言語に Python を用いることで学習しやすくする，GPGPU の処理は自動変換によって使用者が記述する必要を無くす，という２点により高速化のための負担を軽減する点が特色である．

3 準備と関連研究

3.1 マルチエージェントシミュレーション

エージェントシミュレーションとは再現するシミュレーションにおける人や動物などそれぞれが独立した意思に従って行動するものをエージェントとし，自身の行動ルールと相互作用を定義したモデルを用いるシミュレーションである．一般的なエージェントシミュレーションでは複数のエージェントモデルを用いてシミュレーションを行うためマルチエージェントシミュレーションやエージェントベースシミュレーションなどと呼ばれる．エージェントが相互に作用する局所的な現象を再現することが目的であり，感染症の伝染，災害時の避難，交通渋滞，SNS などによる情報拡散など，社会的な現象を研究対象とする一つのツールとして用いられる．

エージェントシミュレーションは，その原理は単純であるが，設計や実現をする場合には，プログラミングなどの知識を要する．そこで，そのような負担を軽減し，効率的にシミュレーションを開発可能にする開発環境が存在する．

NetLogo[1] は複合的な自然現象や社会現象をモデリングしシミュレーションを行うことができるマルチエージェントプログラミング環境である．汎用的に用いられるエージェントの振るまいや，相互作用の手続きがライブラリとして提供され，効率的なシミュレーション開発を実現している．また，シミュレーションに用いるパラメータ等を容易に変更することが可能で，効率的にシミュレーションの検証ができる．

Repast Simphony[2] は複雑適応系(Complex Adaptive Systems : CAS)におけるエージェントシミュレーションを開発することができるオープンソースソフトウェアであり，高度なモジュール化されたプラグインによってネットワークやログ，スケジューラといった機能を変更することができる．開発には ReLogo と呼ばれる Logo 言語の方言の一つを用いるか, Java を用いてシミュレーションを作成することができ，プログラムの記述に統合開発環境の一つである Eclipse による開発が行えるようプラグインが提供されている．これを用いることでエージェントの定義やシミュレーションの流れ，実行やデバッグなどを行える．またボイラープレートと 呼ばれる実行時の設定を記述する支援が行われており，基本的な英語と同等な記述を行うことができる．

これらのシミュレーション環境構築には研究者が容易に記述できるよう設計されているが，エージェント数が大量になる場合，シミュレーションの実行速度が低下する．並列実行環境を扱う Repast HPC[3] などのフレームワークも存在するが，並列処理等に関する専門の知識が必要であり，手軽に扱えるとは言えない．

3.2 CUDA と GPGPU

CUDA (Compute Unified Device Architecture) は，NVIDIA が提供している GPU 向けの言語統合開発環境であり，コンパイラ，ライブラリなどから構成されている．開発には C++言語を主に用いる．汎用的な数値演算に利用される数値型に対応し, 開発者は C++言語と同等にプログラミングを行う. CUDA は無償で提供されており，NVIDIA が提供している CUDA に対応した GPU を搭載したコンピュータを保持していれば開発を行うことができ，設備による導入コストは低い．CUDA は，1 つの命令を複数のデータに対して適応し，並列に処理を行う SIMD(Single Instruction Multiple Data)での並列処理を行えることが特徴である. 実際には，1 つの命令をスレッドの集合である warp という単位ごとに処理するため SIMT(Single Instruction Multiple Threads)と呼ばれることもある．

本研究では，各エージェントの処理をスレッドに割り当てるという素直な方法で, GPU 上でシミュレーションを実現し並列化を図る．しかし，C++言語での GPGPU によるプログラミング開発は，シミュレーションを作成する研究者等にとってはハードルとなる．本研究では, より修得が簡単である Python 言語をベースとした環境を提供する．本研究では，PyCUDA[4] および Numba[5] という GPGPU ツールを利用する．

PyCUDA は 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．GPU のメモリ確保や転送，プログラムのコンパイル，実行などを行う関数が提供されている. Python による呼び出しを行うため，Python の多彩なライブラリを利用することや，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活かしつつ，画像処理やビッグデータ解析など時間のかかる処理を GPU の高い並列処理能力を使用することで短縮することができる．さまざまな GPU による線形代数学や高速フーリエ変換，数値解析のライブラリが提供されており，開発者はこれらを用いることで容易に処理時間を短縮することができる.

Numba は Python プログラムに対して，デコレータによって指定した部分をネイティブコードにコンパイルをすることで, プログラムを高速化する技術である. デコレータでは C 言語への変換と GPGPU への変換のどちらかを指定できる．Numba による GPGPU は Python プログラムを GPGPU にコンパイルすることで実現するため，Python の構文を使うことができる．さらに Python の math ライブラリや乱数を生成する関数を使用することができ，PyCUDA と比べると数学的な処理を使うプログラムの実装が容易になっている．

3.3 GPU におけるブランチダイバージェンス

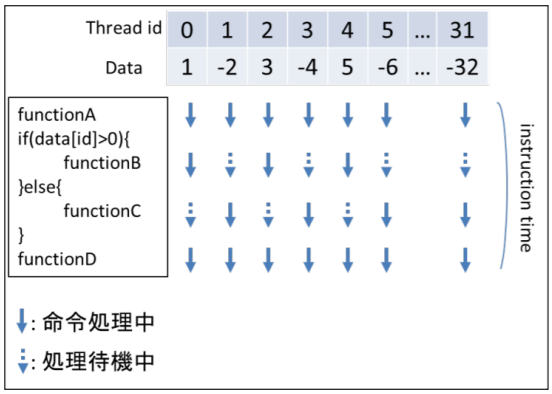

Fig. 1: ブランチダイバージェンス.

CUDA では一般的に 32 個のスレッドを束ねた warp と呼ばれる単位ごとに命令を割り当てる．この warp は命令呼び出しごとではなく関数単位で割り当てられる．ところが，条件分岐文によって warp 内のスレッドが異なる命令を処理する必要がある場合，一部のスレッドが処理を行い，残りのスレッドは処理を行うことなく待機しているだけとなる．また，warp の中で 1 つのスレッドのみが実行する場合でも，のこりの実行されないスレッドに対しても同等のコストがかかる．そのためブランチダイバージェンス（分岐によって制御の流れが異なったものになること）が増加した場合，GPU で処理する命令は増加し，また処理を行っていないスレッドがあるため使用率が低下し処理にかかる時間が増加する．

Fig. 1 にブランチダイバージェンスが発生する例を示す. CUDA では 32 個のスレッドに対して同一の命令を出すため，functionA は同時に処理を行うことができる．しかし，2 行目の条件分岐によって functionB と functionC のどちらかを処理するため, functionB を処理するときは functionC を処理するスレッドは待機している．functionC を処理するときも同様に functionB を処理するスレッドは待機する必要がある. この例では各スレッドは 3 処理で処理が終わるが warp で見ると 4 処理必要になり，GPU の使用率が下がると同時に処理時間が悪化することがわかる．

エージェントシミュレーションでは，個性の異なるエージェントを GPU のスレッドに割り当てるため，ダイバージェンスの問題が発生する．本研究では，この対策について，既存の対策手法を適用した．

既存の対策では， Eddy Z.Zhang らの研究 [6] で挙げられている同一の分岐を行うデータを整列させる方法や，加藤らの研究 [7] で挙げられている for ループの 1 ループごとにデータを整列させるかを確認する方法などがある．本研究では if 文によるブランチダイバージェンスに対してデータを整列させて対策を行う．

4 本研究で開発したフレームワーク

```
def mydata(who):
    return {“x”: 10, “y”: 20, “id”: who}
```

Fig. 4: エージェント固有データの定義.

本研究では, Python で定義したエージェントのメイン処理を CUDA を用いて並列に処理することによってシミュレーションの高速化を図る．本ツールは，オブジェクト指向フレームワークとして構成され，シミュレーションを行うにあたり必要となるいくつかの機能を提供している．主な機能としてエージェントの個性を定義するエージェント定義，シミュレーション全体の流れを定義するシミュレータ定義の 2 つに分けられる．また，さらなる高速化のための機能を使用してカスタマイズすることが可能である．

4.1 シミュレーションの概要

本フレームワークでは，人や動物などを表す「エージェント」および，それら以外の「オブジェクト」の 2 種類によって状態を表し，「シミュレータ」によって状態の管理やエージェントの処理の呼び出しが行われる．開発者は，上記の 3 種類のそれぞれについて，初期化，前処理，後処理を定義し，エージェントについてはさらにメイン処理を定義する．これらの処理は以下の順序で実行される．

1. シミュレータの初期化，前処理を行う．
2. オブジェクトの初期化，前処理を行う．
3. エージェントの初期化を行う．
4. エージェントの前処理，メイン処理，後処理をそれぞれ行う．
5. 画面表示を定義している場合，表示を行う．
6. 定義したステップ数だけ 4，5 を繰り返す．
7. オブジェクトの後処理を行う．
8. シミュレータの後処理を行う．

それぞれの処理における例としては，初期化では定数や変数の定義，またフィールドクラスの定義を行う．前処理ではパラメータや属性の設定を行う．メイン処理ではエージェントの意思決定を伴う処理を行う．後処理ではデータの更新やファイル出力などを行う．ここで述べた例は一例であり，シミュレーションの種類や目的に応じて適宜変更して記述することができる．

エージェントの定義の一例を Fig. 2 に示す．これは感染症の拡散のシミュレーションのエージェントの処理である．この例では前処理（pre 関数）と後処理（post 関数）は無く，メイン処理（mainloop 関数）のみを行うモデルになっている．mydata 関数で自身を初期化し，view 関数で描画の際のエージェント単位の処理を定義している．先頭で定義されている変数 agentManager と field には実行時に後述の各クラスのインスタンスが代入される．

4.2 シミュレータの定義

シミュレータの定義ではシミュレーションを行う際に必要となるパラメータや関数の定義を行うことができる．設定項目は，シミュレーションのステップ数，実行するシミュレーション名または使用するエージェントの種類と個体数，画面表示によるフィードバック

```python
#person program
import random
agentManager=None
field=None

def mydata(who):
    return {"x":random.randint(1,fW), "y":random.randint(1,fH),
            "mx":random.randint(1,5), "my":random.randint(1,5),
            "infect":False if( who != 0 ) else True, "count":0}
def pre(data):
    pass
def mainloop(data):
    if not data["infect"]:
        for d in personList:
            if d["x"]==data["x"] and d["y"]==data["y"] and d["infect"]:
                data["infect"]=True
                break
    else:
        data["count"]+=1
        if data["count"]>30:
            data["count"]=0
            data["infect"]=False
    data["x"]+=data["mx"]
    data["y"]+=data["my"]
    if not ((0<data["x"]) and (data["x"]<fW)):
        data["mx"]*=-1
    if not ((0<data["y"]) and (data["y"]<fH)):
        data["my"]*=-1
def post(data):
    pass
def view(data):
    if data["infect"]:
        return [[data["x"],data["y"],data["x"]+10,data["y"]+10],
                {"fill":"black"}]
    return [[data["x"],data["y"],data["x"]+10,data["y"]+10],
            {"fill":"blue"}]
```

Fig. 2: エージェントのプログラム例.

の有無である．シミュレーションのステップ数は整数値による指定と永久実行の 2 つに対応しておりユーザの目的に合わせた設定を行うことができる．エージェントの種類と個体数の設定では，ユーザが作成した様々なエージェントを使用するか，しないかの指定，また，個体数の設定を行うことが可能で，シミュレーションの規模やエージェント種間のバランスを変更した様々なシミュレーションを行うことができる．

これらの設定はファイルに記述することもできる．この場合はシミュレーション名と同名のディレクトリ内にある，実行時に使用するエージェントの種類と個体数が記述された CSV ファイルから設定を読み込んで自動で設定する．さらにこのシミュレーション名には略称を使うことができ，実行するシミュレーションを変更するときに容易に書き換えることができる．この機能は用意された専用の CSV ファイルに略称を記述することで使用できる．

画面表示によるフィードバックでは，シミュレーションの情報を表示するための命令を記述する．ユーザがフィードバックのプログラムの経験がない場合を考慮し画面の表示等をあらかじめ定義したクラスを提供しており，これによりユーザは描画を行うための開発に時間をかけずに済む．また，開発者が独自の表示系プログラムを保持している場合は本システムから描画関数を呼び出すよう記述することで表示することができる．

```
class Field:
    def getWidth():
        フィールドの横幅を返す
    def getHeight():
        フィールドの縦幅を返す
```

Fig. 3: フィールド情報管理クラス.

```
class AgentManager:
    def getSize( ):
        全エージェント数を返す
    def getAgent( id ):
        id 番目のエージェント情報を返す
```

Fig. 5: エージェント情報管理クラス.

シミュレータで定義されたフィールドの情報を取得することができるクラスの一部を Fig. 3 に示す．このクラスを用いることでフィールドの縦横の幅やフィールドの情報を取得することができ，エージェントの移動や処理に用いることができる．また，この関数を実行して得られる値を最初から格納している定義済みの変数を試験的に実装している．これを使うことで先述の pre, mainloop, post の各関数内でこの関数を実行して値を取得する必要が無くなる．

4.3 エージェントの定義

本環境では基底クラスが用意される．この基底クラス群はプログラマには隠蔽される．シミュレーションの用途に合わせたメソッドを複数定義しており，これらを組み合わせることで様々な個性を持つエージェントを定義することができる．提供されている関数として，

(1) エージェント固有のデータを定義する関数
(2) エージェントの前処理
(3) エージェントのメイン処理
(4) エージェントの後処理

がある．

エージェント固有のデータは Fig. 4 のように Python の Dictionary と同等の処理を行うクラスを提供しておりこれを用いて定義，参照できるため，特殊なクラスを利用しておらず Python を使用することができる開発者であれば新たに学習する必要はない．

シミュレーションに利用されるエージェントの情報を柔軟に取得するためのクラスの一部を Fig. 5 に示す．これによりユーザ自身でエージェントの情報を保持した変数の定義や管理をし，引数の増加を防ぐことができる．またこのクラスではエージェントの生成，削除の機能を設けており，個体数が変化するシミュレーションにおいても対応することができる．

4.4 更なる高速化のためのカスタマイズ

本フレームワークでは，シミュレーションを高速化するための細かいチューニングが加えられるようになっている．まず画面表示のタイミングを制御することができる．一般的なプログラムの処理の中で画面表示は時間のかかる処理の一つである．そして大規模なシミュレーションの場合では表示するエージェントやオブジェクトが非常に多くなりコストが高くなることが多いため，画面出力の頻度を減らすことでシミュレーションの結果を変えることなく処理の高速化を行うことができる．さらにブランチダイバージェンスを削減するための機能を使用するかどうかを指定できる．

また，本環境では主にエージェントのメイン処理を

```
def post(data):
    data[“x”] = data[“nx”] #x 座標更新
    data[“y”] = data[“ny”] #y 座標更新
```

Fig. 6: 後処理の並列可能例.

GPU による実行によって高速化しているが，この他にも通常は CPU で実行するエージェントの前処理や後処理を GPU により処理することができる．Fig. 6 のプログラムのようにエージェントの座標更新といったエージェントが互いに参照しあうデータは全エージェントで同期をとる必要がある．しかし他のエージェントからデータを参照されることがない関数では並列に処理することができる．ユーザはこのようなプログラムを記述した場合に GPU で処理するように設定することで高速化することができる．

5 実装

5.1 変換

本ツールではユーザが記述した Python コードを GPU で実行できるソースコードに変換する．この方法としては PyCUDA で行う方法と Numba で行う方法の 2 通りから選択できる．本節では各変換方法にて GPGPU 化できる処理の種類と変換方法について説明する．

5.1.1 PyCUDA を用いるプログラムへの変換

PyCUDA を使用する GPGPU への変換では，本フレームワーク内に Python 言語で定義されたクラスを CUDA C 言語でも実装することで，同じ流れで処理ができるようにしている．この変換は以下の処理に対して行うことができる．

A) エージェントのデータへの代入・参照処理
B) Field クラスのインスタンスのメンバ変数
C) エージェントのデータを順次取得するループ処理

各処理をエージェントのプログラムの一例の Fig. 7 を参考に説明する．この例は，感染症の拡散のシミュレーションにおけるエージェントの意思決定部分である．ここで，変数 data はエージェント自身を表す．全体のエージェントの状態を参照し，その結果に応じて，自分自身のデータを変更する．A)の処理は Fig. 7 の 2 行目の参照処理（data[“infect”]）や 5 行目の代入処理である．このようにエージェントの表現は，Python の辞書（dictionary）を用いた記述を採用しているが，これを CUDA C に変換した場合には，属性名によってアクセスができるようにクラスを定義することで実装している．B)と C)については，「シミュレータ」に対応する計算を CUDA C 側でも定義することで変換を可能としている．Fig. 5 の AgentManager クラスはエージェント全体の集合を管理するクラスである．しかし，AgentManager クラスのメンバ関数である，エージェントの追加と削除を行う関数については，GPU 上での実行を現実装ではサポートしていない．

```
def mainloop( data ):
    if not data[“infect”]: # 感染していない場合
        for d in personList:
            if d[“x”] == data[“x”] and d[“y”] == data[“y”] and d[“infect”]:
                data[“infect”] = True
                break
    else: # 感染している場合
        data[“count”] += 1
```

Fig. 7: 最適化に適したプログラムの一例.

5.1.2 Numba を用いるプログラムへの変換

Numba を使用する GPGPU への変換では，Numba では CUDA C 言語でクラスを定義できないこともあり，PyCUDA とは違った変換方法を採用している．しかしながら，Numba の GPGPU 関数の記述が容易であることもあり，5.1.1 節の A)と B)，C)の他に，以下の処理に対応が可能となり，GPU で実行できる範囲が増えている．

D) Python の math ライブラリを使用した処理
E) Python の random ライブラリの random 関数, randint 関数を使用した処理
F) エージェントの削除処理
G) エージェントの追加処理

各処理の変換について説明する．D)については，Numba にて既にサポートされているため，その機能をそのまま使用している．E)については，Numba にて各スレッドでの乱数生成を行うクラスが用意されており，そのクラスを使った処理に変換している．F)と G)は GPGPU と CPU による逐次実行を組み合わせて実装している．F)では処理の中で削除することが決定したエージェントをマーキングし，並列での処理が終わった時に逐次実行でまとめて削除している．G)では処理をエージェントの追加処理部分とその前後の 3 つに分断し，エージェントを並列でまとめて追加できるようにしている．

5.2 ブランチダイバージェンス

本提案ツールでは CUDA におけるブランチダイバージェンスを削減することによる最適化を行う．エージェントシミュレーションにおいて，異なるふるまいを行うエージェントのデータが隣接していることが一般的であり，ブランチダイバージェンスを引き起こす要因となる．本提案環境では，各スレッドが参照するデータを変更することで同一の分岐を行わせる手法を用いて最適化を行う．

本最適化支援の有効になるプログラムの一例としては先に挙げた Fig. 7 がある．このプログラムでは，処理の大部分が条件分岐によって分岐したのちに行われている．またエージェントの過去の状態を用いて warp の割り当てを変更するため比較的変化の少ないパラメータを用いた条件に特に有効である．

6 性能評価

本節では，エージェントシミュレーション実行時の処理時間を計測し性能を評価するとともに，最適化支援としてブランチダイバージェンスを削減した場合の実行時間について計測する．また，開発者がエージェントの個性を定義する容易性について評価を行う．

6.1 サンプルプログラム

本研究では，2 つのシミュレーションについて実験

を行った．これらは NetLogo が提供しているサンプルモデルをそれぞれ同等の処理を行うよう移植したものである．以下に示す．

1. 食物連鎖のシミュレーション
2. 感染症の拡散のシミュレーション
3. 蜂の巣の作成のシミュレーション

これらの特徴について説明する．まず食物連鎖のシミュレーションだが，このモデルは生態系が安定する状況を検証することができ，捕食や餓死等によってエージェントの個体数が増減する特徴を持つ．感染症の拡散のシミュレーションは他のエージェントへの感染拡大をシミュレーションすることができ，本研究においては全ての処理を PyCUDA で GPGPU 化することができる．蜂の巣の作成のシミュレーションは蜂の巣の 6 角形の巣穴を作成する過程を観察することができる．本研究では，これらのプログラムをサンプルプログラムとして提供している．フレームワークの利用性を向上するためには，単純なサンプルを用意することが必要であると考える．使用者はサンプルをテンプレートとして改造する形で開発を行うことができる．今後サンプルは増やす必要がある．

本サンプルを用いた評価では 1 と 2 のプログラムの実行時間を計測することで GPGPU 変換の効果を確認する．今回の計測では 3 は使用しない．

6.2 実行速度の評価

この評価では Python と本フレームワークを用いた時のシミュレーション実行の処理時間を計測する．計測に用いた実験環境は Table. 1 の通りである．計測時間はデータの転送命令を含むエージェントの前処理，メイン処理，後処理を毎ステップ計測し合計した値を用い，各 5 回計測した中で最大値と最小値を除いた平均値を示す．また，並列化時のスレッド数は毎回 512 で，ブロック数はエージェント数を 512 で割った値に 1 を足した値であり，6.2.2 節の場合は 2~16，6.2.3 節の場合は個体数が増減するため 1~24 となる．

Table 1: 実験環境

OS	Windows 10 Home
CPU	Corei7-8700 3.20GHz
GPU	GeForce GTX 1060
Python Version	3.6
CUDA Version	9.2
Numba Version	0.42.0

6.2.1 AIDS

感染症の拡散のシミュレーションの実行時間を Fig. 8 に示す．今回の実行では初期感染者は 1 体，フィールドの大きさは 100×100 で上下左右が接続し，ステップ数は 100，エージェントの個体数を 1000 体から 1000 体ずつ増加させて行う．縦軸が計測時間で単位は秒，横軸がエージェント数を示している．

計測の結果を見ると，PyCUDA では最大 106 倍，Numba では最大 754 倍の処理効率を示した．エージェントの個体数が増加するにしたがって処理にかかる時間が増加しているが，CPU による処理に比べ GPU による処理ではエージェントの個体数の変化による影響が小さいことがわかる．そのためさらに処理が増加した場合であっても処理時間が大きく変化することがないと考えられる．

6.2.2 Wolf Sheep predation

続いて，食物連鎖シミュレーションの実行時間を Fig. 9 に示す．今回の実行では，ステップ数 100，羊は 200 体，オオカミは 100 体，初期座標はフィールド内でランダム，フィールドの大きさは 50×50 を初期値とし，フィールドは上下左右で接続している．その他捕食時のパラメータ等は NetLogo があらかじめ定義している数値を用いている．羊とオオカミは密度を保ったままフィールドの大きさを 1 辺 50 ずつ変更しシミュレーションを行った．縦軸が計測時間で単位は秒，横軸がフィールドの大きさを示している．また，どちらかの種が絶滅した場合のシミュレーションは計測に含んでいない．

結果としては，PyCUDA では最大 63 倍，Numba では最大 222 倍の高速化を行えた．こちらでも AIDS と同様に，CPU による処理に比べ GPU による処理ではエージェントの個体数の変化による影響が小さいことがわかる．このシミュレーションではエージェント生成処理によって完全な GPGPU 化ができず，逐次処理を挟むため AIDS 程高速化しないと考えられる．

6.3 ブランチダイバージェンスの削減

本環境におけるブランチダイバージェンスの削減率について検証する．使用するプログラムは 6.2.1 節で使用した AIDS シミュレーションのプログラムを使用し，パラメータも同様のものを使用した．処理時間を計測し，その結果を Fig. 10 に示す．また，参考として別の環境で行ったブランチダイバージェンスの削減率のグ

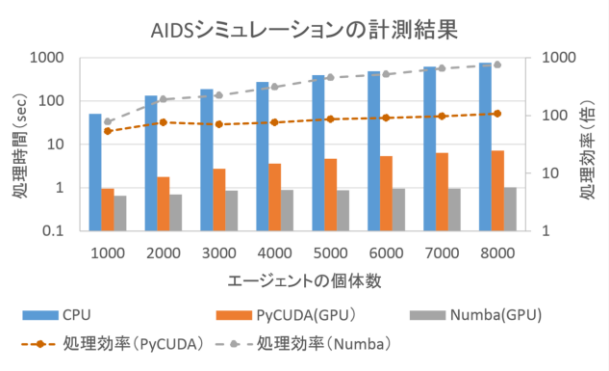

Fig. 8: AIDS 計測結果.

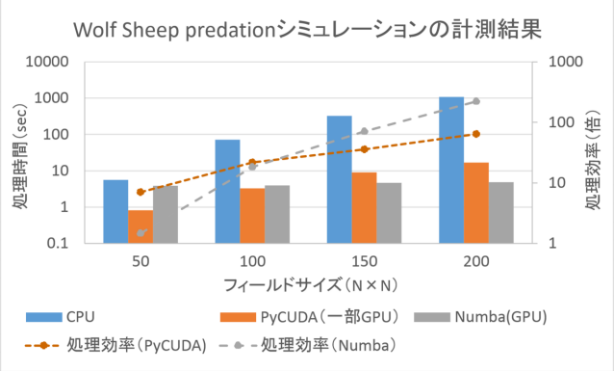

Fig. 9: Wolf Sheep predation 計測結果.

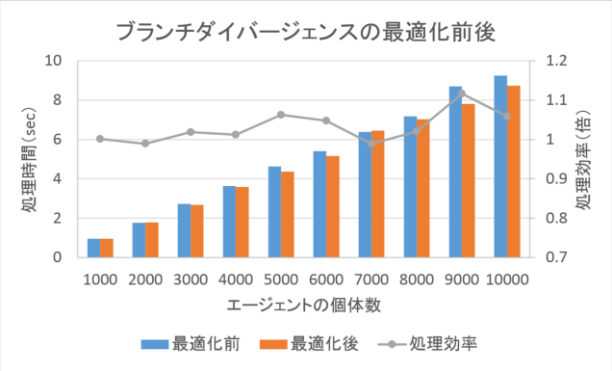

Fig. 10: 最適化時処理時間.

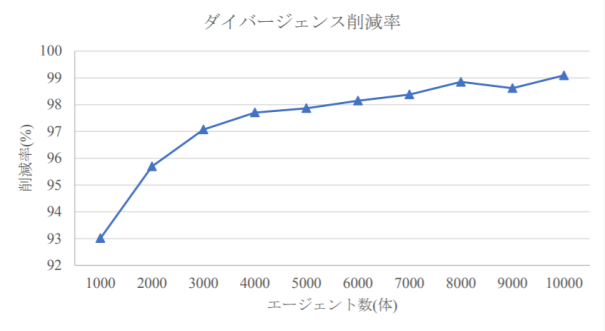

Fig. 11: （参考）ブランチダイバージェンス削減率.

ラフを Fig. 11 に示す．これは NVIDIA が提供するプロファイラを用いて計測を行っている．

Fig.11 では最適化支援によってブランチダイバージェンスの発生そのものを抑えることができていることが分かる．一方で，処理時間の向上は数パーセントであり，さらに，効率が低下しているケースもあることがわかる．これはブランチダイバージェンス削減のためのデータ列の整理に多くの処理時間が必要になるからと考えられる．また，データ列の整理に実行時の参照するデータを変更する手法を用いており，データアクセスがランダムに近くなることも要因の一つだと考える．

本研究によってエージェントの動作を並列化することでシミュレーションを高速化することができる．しかし効果が発揮されにくいシミュレーションも考えられ，その特徴としては，(1)上記に挙げたブランチダイバージェンスが多いもの，(2)エージェントが持つデータの量が多いもの，(3)並列化する処理の中で何回も同期をとる必要があるものが挙げられる．(1)に関しては，多数のエージェント種が存在し，それぞれの種に対して性質の異なる処理・手続きが適用されるようなモデルであり，(2)は，エージェントが細かい多数の特徴量によって定義されるようなモデルであり，(3)は，ある属性の値に対する参照と代入がエージェント種間で頻繁に行われるようなモデルである．

6.4 記述性の評価

本環境を用いた際にユーザが記述するプログラムについて考察する．性能評価に用いた 6.2.1 節のプログラムの GPU で処理を行う関数の一部を Fig. 12 に示す．これは典型的なエージェントシミュレーションで行われる自身の情報と他のエージェントの情報を用いた処理を行っており，自身を除くすべてのエージェントに対して自身と同じ座標でかつ感染状態であるエージェントがいた場合，自身を感染状態に変化させる処理を記述している．

```
for d in agentList:
    if d["x"]==data["x"] and d["y"]==data["y"]
                         and d["infect"]:
        data["infect"]=True
```

Fig. 12: 他エージェント参照例.

7 考察

7.1 実行速度

評価実験より本提案環境では Python と比較した場合，PyCUDA では AIDS において最大 106 倍，Wolf Sheep predation において最大 63 倍，Numba ではそれぞれ 754 倍，222 倍の高速化を得ることができ，GPGPU による性能向上を見ることができる．特にエージェントの個体数の増加に応じて Python による実行と比べて本提案環境を用いた場合では処理時間の増加が緩やかであり，エージェントの個体数が多い場合において削減率が大きくなることが見られた．また PyCUDA ではコード変換や GPU 上のメモリ確保等を最初の呼び出しのみにすることでオーバーヘッドの削減したことも速度向上の一因と考えられる．

また今回の Numba を用いた変換では PyCUDA を用いた時よりも高速化していたが，Numba では中身が空の pre, post 関数の実行時間を大幅に削減できることが理由の一つと考える．これらの関数は例えば AIDS シミュレーションでは PyCUDA での実行時間の 67%を占めている．また，Numba ではエージェントのデータが引数で与えられるのに対し，今回の PyCUDA を用いた変換では，エージェントの値を得るためにクラスの関数を経由する必要があり，エージェントの個体数を増やすことでそのアクセス時間が増えたと考える．しかしこのクラスは CUDA C 言語で Python の dictionary のようなアクセス方法を使用するために必要であり，将来の機能として考えている CUDA C コードの出力機能のためには欠かせないものとなっている．

7.2 ブランチダイバージェンス

本フレームワークでは最適化支援としてブランチダイバージェンスの削減を試みた．ブランチダイバージェンスの計測結果では本フレームワークの機能を用いることで 95%以上の削減を達成できることがわかる．今回の計測で用いたプログラムに対して有効であると考える．しかしブランチダイバージェンスを削減した場合であっても処理時間を短縮されていないことが見られた．これは本機能の処理がブランチダイバージェンスの削減による短縮される時間より大きくなってしまうことが考えられる．また本フレームワークにおけるブランチダイバージェンスの削減はデータの参照先を変更する手法を用いているため，メモリアクセスがランダムに近くなり処理に時間がかかっていることが考えられる．さらに本機能におけるデータの整列においてプログラムの解析と整列に用いる手法についての検証を行っていない．プログラムの解析を行い，条件

分岐の命令を検出できた場合であっても数値や真偽値によって重みや処理を変更する必要があると考える．それらに加えて,データ整列のコストがかかっている．そのためシミュレーション全体の処理時間は低下しており最適化の手法が不十分であると考えられる．しかし，CUDA を用いたプロトタイプ実装での事前調査において本手法での最適化を行った場合では速度の向上が見られた．これらのことから，エージェント個別のデータにアクセスするコストやフレームワークによるオーバーヘッドが原因と考えられ，検証し原因を明確にするとともに改善することが必要である．

7.3 記述性

記述性においては GPGPU で必要となるデータの流れや呼び出しを開発者に見せることなく処理を行っており, Python と同等の記法によってプログラムを作成でき，一般的な制御構文に加えてクラスによるオブジェクトの定義が可能で単純な四則演算から複雑な構造を持つオブジェクトを扱ったプログラムまで作成が可能である．さらにエージェントやフィールドのデータを取得するクラスを提供しており，他エージェントのデータやフィールドの情報を参照することが容易になったと考える．また，データを受け渡すための引数の増大を防ぐことができ，記述量の削減に加えて可読性も向上したと考える．

また，本環境で記述したソースコードは Python によって処理することができ, CPU による実行でデバッグを行うことも可能であるため，GPU による処理を行う前に正しい動作をするかを確認したのちに処理を行うことが可能である. また CPU による処理を行う関数では Python のライブラリを用いることができる. そのためエージェントを一つのデータの集まりとしてとらえることで，暗号の解読や画像処理といったエージェントシミュレーション以外の目的や，Python のライブラリを用いてシミュレーションにネットワークを用いることができると考えられる．

定義済みの変数を用いることで本フレームワークを利用する際の記述方法の記述量を削減し，可読性を上げることができる．エージェントをループするプログラムでは Fig. 12 のとおり Python らしく記述することができている．

8 あとがき

本研究では GPU を用いたエージェントシミュレーションのフレームワーク開発の研究を行った．このフレームワークにより GPGPU に必要となる GPU 上のメモリへのデータ転送命令やスレッドの設定などの命令をユーザから隠蔽しつつ実行時間の削減が図れた．また，ユーザによっては，さらなる高速化のための機能を使うことで実行時間の短縮を行うことができる．

今後の課題としては，サンプルプログラムの増強を行い利用者の記述方法の学習をより支援しやすくすることの他に，対応構文の拡充と型推論があげられる．本提案手法では Python の構文すべてに対応しておらず，あらかじめ対応している構文に合わせてプログラムを作成する必要がある．また，言語変換器によって作成されたプログラムも同等の処理を行うが制約があり，意図した処理を行うために記述量が増加することがある．

型推論は, Python は動的型付け言語であるため不要であるが，CUDA C で必要となり変数や関数に型情報を持たせる必要があるが，フレームワークが適切な型を付与しているとは限らず，暗黙の型変換で対応できない箇所に関してはエラーになってしまう．

さらに分散処理に対応することも必要だと考える．本論文にて用いたプログラムはエージェントの個体数が 10000 体程度であり商業施設や駅であれば対応することができている．しかし，市町村規模でのシミュレーションでは数十万から数百万規模のエージェントが必要となり，また人以外のエージェントも存在するため処理が増大することが考えられる．そこで複数のコンピュータを用いた分散処理や複数の GPU を用いたマルチ GPU に対応することでこれらに対応することができると考える．この問題には GPU のシェアードメモリやコンスタントメモリ等の高速なメモリを利用することも有効であると考えられる．シェアードメモリはグローバルメモリよりも高速にデータアクセスを行える領域で，グローバルメモリと比較して 100 倍以上の速度でアクセスができる [8]．分散処理による煩雑さを防ぐためにも，これらの高速なメモリを利用できるようにすることが考えられる．

参考文献

1) S. Tisue, U. Wilensky: NetLogo: Design and Implementation of a Multi-Agent Modeling Environment, SwarmFest (2004)
2) N. J. Michael, C. T. Nicholson, O. Jonathan, T. R. Eric, M. M. Charles and Mark Bragen and Pam Sydelko: Complex adaptive systems modeling with Repast Simphony, Complex Adaptive Systems Modeling, (2013)
3) Nicholson Collier, Michael North: Parallel agent-based simulation with Repast for High Performance Computing. SIMULATION, **89**-10, 1215/1235 (2013)
4) A. Klockner, *et al*.: PyCUDA: GPU Run-Time Code Generation for High-Performance Computing, *Parallel Computing*, **38**-3, 157/174 (2012)
5) Siu Kwan Lam, *et al*.: Numba: a LLVM-based Python JIT compiler, *The Second Workshop on the LLVM Compiler Infrastructure in HPC*, (2015)
6) 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)
7) 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)
8) Hisa Ando：GPU を支える技術 超並列ハードウェアの快進撃[技術基礎], 技術評論社, (2017)