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

Kamata, Tomoya / 鎌田, 知也

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
13
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2017-03-31
(URL)
https://doi.org/10.15002/00021520

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
Support for acceleration of agent simulation using GPGPU

鎌田　知也
Tomoya Kamata
法政大学大学院情報科学研究科情報科学専攻
E-mail: tomoya.kamata.2b@cis.k.hosei.ac.jp

Abstract

Recently, the number of station buildings and commercial facilities has increased due to redevelopment of cities, emergence of new business etc. In designing these facilities simulations are performed for predicting phenomenon caused by a very number of people aiming for example, to selecting escape routes and promote sales promotion. However, processing by the CPU requires much time for processing. Also, although processing can be performed at high speed by processing using the GPU, it is necessary to acquire knowledge of GPU architecture and special language. In this research, we propose a framework for reducing execution cost in agent simulation. In this framework, the development cost is reduced by processing the simulation with the GPU while concealing the GPGPU-specific commands to the developer. The framework also provides an optimization support for reducing divergence and shorten processing time at runtime. As a confirmation of the usefulness of the framework, we compare the execution time of simulation by CPU and GPU, discuss the description when using this framework, and evaluate this research.

1 はじめに

近年駅の再開発等により様々な駅ビルや百貨店などの商業施設が増えている．商業施設を設計する際に災害時の避難経路の選定のためや，販売促進のために人の流れを予測，コントロールするために利用するなど，シミュレーションの重要性が増している．しかし商業施設等のシミュレーションでは一定のルールに基づいて自律的に行動するエージェントをモデル化し，利用者と同数程度の数を用いる必要がある．また様々な人が集まる商業施設を再現するためには，異なる個性を持つエージェントを定義する必要がありプログラムが複雑になるとともにこれを管理することも必要になる．さらに，一度に数万のエージェントを用いるようなシミュレーションの場合，CPU による並列化のみでは実行時コストが非常に高く様々なパラメータでの実験を行うことが難しくなる．

高い並列処理を行えるプログラミングとして General-purpose computing on GPU (GPGPU)がある．GPGPU は単純な処理ができるプロセッサを多量に搭載しており CPU と比べ高い並列能力を持つ．GPGPU に用いる言語に NVIDIA が提供している CUDA があり，これは C++を拡張した言語だが，処理に用いるデータを GPU 上のメモリへ転送するための命令やスレッドの同期など GPGPU 特有のものがあり習得まで時間を必要とする．さらにプログラムの高速化を行う手法の一つにブランチダイバージェンスの削減がある．ブランチダイバージェンスは CUDA のスレッドが warp と呼ばれる単位で命令を実行するが，条件分岐によって異なる命令を実行する場合，他のスレッドを休止し命令を実行するため，GPU の使用率が低下し処理に時間がかかってしまう．これを削減するためには条件分岐を減らす手法や同じ命令を実行するスレッドを同一の warp に割り当てる手法などがある．しかし，動的に変化するデータを用いた分岐の場合，ユーザがデータを監視し適切に割り当てる必要があり導入するにはコストが高くなっている．

既存のシミュレーション環境として NetLogo があるが独自の言語を用いた記述が必要であり Java や C のような汎用言語を用いることが出来ない．また，実行時に並列に処理を行うが，計算機のプロセッサに応じた並列処理を行うためエージェントが多量になった際に処理に多くの時間を必要とする．他の環境として Repast Simphony があり，Java を用いたモデル定義が可能となるが，データ構造を Repast Simphony が提供しているものに合わせる必要がある．

本研究ではエージェントシミュレーションにおける実行時コストを削減するためのフレームワークを提案する．本研究は GPU を用いたシミュレーションのフレームワークを構築し，GPGPU 特有の命令を隠蔽しつつシミュレーションの実行時コストを削減することを目的とする．これにより記述量の削減や可読性，小規模でのデバッグによる確認を行うことができる．また実行時の最適化としてブランチダイバージェンスの削減を行い最適化の支援機能を備えることで高速化を図った．本研究の性能評価として NetLogo が提供しているサンプルモデルと同等の処理を行うプログラムを作成し処理時間を比較する．

2 関連技術，関連研究

2.1 関連技術

2.1.1 NetLogo

NetLogo は自然現象や社会現象をシミュレーションすることができるマルチエージェントモデリング環境である．モデル定義には Logo 言語と呼ばれる JVM で実行す

Supervisor: Prof. Akira Sasaki

る言語を用いて定義する．これは自然現象や社会現象をシミュレーションすることに特化したモデリング環境である．開発者は独立して動作するエージェントを定義することができ，ミクロレベルの行動とマクロレベルの行動パターンを検証することが出来る．

2.1.2 Repast Simphony

Repast Simphony はエージェントのモデリングとシミュレーションを連続して行えるオープンソースプラットフォームである．開発者は Logo の方言の一つである ReLogo や Java を用いた開発を行う．またプログラムの記述には IBM によって開発されオープンソースの Eclipse を用いる．

2.1.3 PyCUDA

PyCUDA は Python から CUDA を呼び出すことが出来るラッパーツールである．これを用いることで，CUDA で一般的に用いられる C++言語以外を用いて GPGPU を行うことが出来る．また，ホスト言語の利点を活かしつつ GPU の高い並列処理能力を使用することが出来る．GPU で処理する命令は LLVM IR へ JIT コンパイルし CUDA で実行する．

2.2 関連研究

本研究の関連研究として 2 つの研究を取り上げる．1 つ目は，Smith らによる並列シミュレーション時におけるプログラミングモデルに関する研究，もう 1 つは Eddy Z. Zhang らによる GPU におけるダイバージェンスを削減することによるプログラムの高速化の研究である．

2.2.1 Indexing Programming Model [2] [4]

Indexing Model は並列処理を行う際に起きるデータの整合性やプロセッサ間での協調性を意識する必要がなくなりデバッグが容易であるプログラミングモデルである．様々な言語による実装が可能で複雑で洗練されたプログラムが容易に記述することができる．

2.2.2 ダイバージェンス削減による高速化 [1]

Eddy Z.Zhang らはブランチダイバージェンスの削減手法を提案している．この手法では，CPU を利用してデータ列を整列することで条件分岐命令による同一のパスを選択するデータ列を同一 warp に集約することでブランチダイバージェンスを削減している．

3 提案ツール

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 によって処理することによって実行時コストを削減することを主な目的としている．本提案フレームワークでは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，熟練者向けの機能選択によってカスタマイズすることが可能である．

```
class AgentManager:
  def getSize():
    return “エージェント数”
  def getAgent(id):
    return “id 番目のエージェント情報”
```

図 1 エージェント情報管理クラス

3.1 エージェント定義

エージェントの定義にはオブジェクト指向を用いて行う．本環境では基底クラスを隠蔽し，シミュレーションの流れに合わせた関数を複数定義しており，これを組み合わせることで様々な個性に合わせたエージェントを定義することが出来る．また，エージェント固有のデータは Python の Dictionary を用いて定義，参照でき，ユーザは特殊なクラスを利用する必要はない．

GPU を用いた並列処理を行う場合，一般的には既存の Python のソースコードを CUDA や OpenCL で実行できるソースコードへ書き換えを行う必要があるが，本提案環境では並列処理を行うために特別な命令の記述なしで並列処理を行うことができる．これによりユーザは小規模のシミュレーションを作成し，実行テストを行った後に目的のエージェント数などのパラメータへ変更し GPU による並列処理を行うことが可能である．また，CPU 用のプログラムと GPU 用のプログラムが同一であるため，低い移行コストで実行時コストを大幅に削減することが可能である．

また，シミュレーションに利用される他エージェントの情報を取得するための関数の一部を図 1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，引数の増加を防ぐことができる．

3.2 シミュレータ定義

シミュレータ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．設定することができる内容としてシミュレーションのステップ数，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の有無を設定することができる．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・不使用，個体数の設定を行うことができ，シミュレーションの規模や他エージェントと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．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，ユーザはエージェントを定義するためのコーディングの時間を使用することができる．また，開発者が独自の表示系

```
class Field:
  def getWidth():
    return “フィールドの横幅”
  def getHeight():
    return “フィールドの縦幅”
```

図 2 フィールド情報管理クラス

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

図 3 後処理の並列可能例

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．また，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を提供しておりその一部を図 2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る．

3.3 熟練者向けカスタマイズ

熟練した開発者にはシミュレーションを高速化するためのいくつかの機能を変更できるようにしている．まず画面表示のタイミングを制御し大規模なシミュレーションを行う上でコストが高くなる画面出力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 GPU 上へのメモリへデータ転送を管理することがでる．データの転送は CPU によってデータを変更することが見込まれる場合や画面表示などを行う際には必要となるが，これらを行わないステップでは転送によるコストを削減することができ，シミュレーションでの高速化が見込める．また，本環境では主にエージェントのメイン処理を GPU による実行によって高速化を行っているが，前処理や後処理を GPU を用いて処理することができる．処理の内容が多いが図 3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．しかし他エージェントからデータを参照されることがない関数の場合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．

3.4 最適化支援

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ではエージェントの状態に応じた処理を行うが，本提案手法ではエージェントの過去の状態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を図 4に示す．このように本最適化では，処理の大部分を条件分岐によって分岐したのちに行われるものにおいて有効であり，またエージェントの過去の状態を

```
def kernel():
 if (エージェントの状態に合わせた条件):
  functionA()
 else:
  functionB()
```

図 4 最適化に適したプログラム

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．

本提案手法におけるダイバージェンスの削減手法を感染症シミュレーションを例に説明する．感染症シミュレーションでは自身が感染していた場合は治癒のための処理を行い，感染していない場合はフィールドの情報や他エージェントの状態に応じて次ステップに感染状態へ遷移するといった異なる処理を行う必要がある．この時ユーザはエージェントの感染状態に応じた条件式を設けそれぞれの処理を各条件節に記述するのが典型である．

4 実装

本フレームワークは Python のクラスライブラリとして提供する．また処理系も Python によって実現している．シミュレーションの流れの定義群，ユーザが定義したエージェントを GPU 実行するための言語変換ツール，エージェントのデータの転送管理ツール，実行時の最適化ツールによって構成される．

4.1 シミュレーション定義

本フレームワークではシミュレータ，オブジェクト，エージェントの 3 つに初期化，前処理，後処理，エージェントのみ前処理，後処理に加えてメイン処理を定義する．それぞれ以下の順序で処理している．

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 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．メイン処理ではシミュレーションの主となる処理を行う．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．

4.2 言語変換

本ツールではユーザが記述した Python コードを CUDA C へ変換を行い GPU で実行できるソースコードを生成する．本節では環境での言語仕様，ユーザが記述したプログラムへの処理について述べる．

```
mod = Module(stmt* body)

stmt = FunctionDef(identifier name,
                   arguments args, stmt* body)
    | ClassDef(identifier name, expr* bases,
                                    stmt* body)
    | Return(expr? value)
    | Assign(expr targets, expr value)
    | AugAssign(expr target, operator op, expr value)
    | For(expr target, expr iter, stmt* body)
    | While(expr test, stmt* body)
    | If(expr test, stmt* body, stmt* orelse)
    | Expr(expr value)
    | Pass | Break | Continue

expr = BoolOp(boolop op, expr* values)
    | BinOp(expr left, operator op, expr right)
    | IfExp(expr test, expr body, expr orelse)
    | Compare(expr left, cmpop* ops,
                            expr* comparators)
    | Call(expr func, expr* args)
    | Num(object n)
    | Str(string s)
    | Attribute(expr value, identifier attr)
    | Subscript(expr value, slice slice)
    | Name(identifier id)

slice = Index(expr value)
boolop = And | Or
operator = Add | Sub | Mult | Div | Mod | LShift
         | RShift | BitOr | BitXor | BitAnd
cmpop = Eq | NotEq | Lt | LtE | Gt | GtE
arguments = expr* args
```

図 5 提案環境の言語仕様

4.2.1 言語仕様

GPU で実行するプログラムの記述は Python の構文のサブセットとなる言語を記述することが出来る．主な式としてビット演算，数値演算，関数呼び出しがあり，文としては条件分岐や繰り返し，関数定義がある．本言語仕様は他言語に同等の命令があるものに注視し設計した．図 5にコード変換に対応している言語の抽象構文を示す．

4.2.2 コード生成

本システムは Python の AST を用いてソースコードから得られた情報をもと GPU で実行するソースコードを生成する．生成する際，同一構文であっても記法が異なるものも存在し，その場合には同等の処理を行うソースコードを生成する．また，コード生成では変数の代入やメソッドの呼び出し時の引数の型情報を確認しない．そのためソースコードを生成するが代入等ができない場合は実行時エラーとして出力される．

表 1 実験環境

	Machine1	Machine2
OS	CentOS 6.7	CentOS 7.4
CPU	Core i7 860 2.8GHz	Core i7 4790 3.6GHz
GPU	Quadro K2200	GeForce GTX 1080Ti
Python Version	3.5	3.5
CUDA Version	6.5	9.0

4.3 データ転送

データ転送ではユーザが定義したマップ型のデータを GPU 上のメモリへの送受信を行う．GPU 上のメモリは C++の配列のように明示的に確保する必要があるが，フレームワークが提供しているデータ転送を管理するクラスによってメモリの確保，データの送受信を行う．また，毎ステップメモリを確保せず再利用しており，さらにエージェント個別にデータを転送するのではなく，前処理として 1 つのデータとして転送することで API や関数の呼び出し回数を減らし効率化を図っている．

4.4 最適化

本提案環境では CUDA のブランチダイバージェンスを削減し実行時間の短縮を図る最適化を行う．エージェントのプログラムで条件分岐を検索し，条件式で参照されているエージェントのパラメータを抽出しこれを用いて同一のブロックを処理するエージェントを集約することでブランチダイバージェンスを削減する．

5 評価実験

本節では，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，最適化支援としてダイバージェンスの削減率について計測する．また，ユーザ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．

5.1 性能評価

本提案環境の実行時間計測では Python，本環境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表 1の通りであり，シミュレーションの内容は NetLogo が提供しているサンプルモデルのうち感染症シミュレーションの AIDS，食物連鎖シミュレーションの Wolf Sheep Predation をそれぞれ同等の処理を行うよう言語移植したものを用いる．また計測時間はデータの転送命令を含むエージェントのメイン処理を毎ステップ計測し，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．

5.1.1 AIDS

AIDS シミュレーションはヒト免疫不全ウイルスが人の接触による感染拡大をシミュレーションすることができる．このシミュレーションでは他エージェントの情報を用いた処理が必要となり，座標が同一であり相手が感

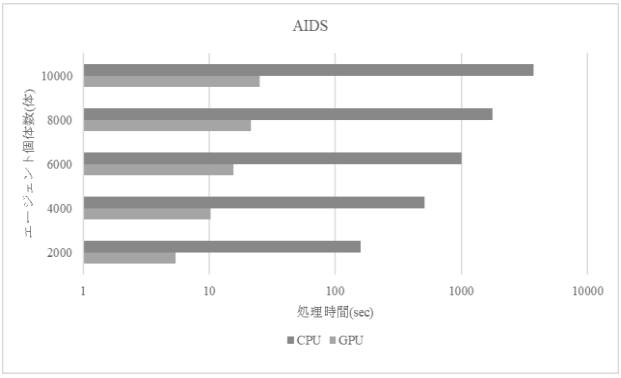

図 7 AIDS 計測結果(Machine1)

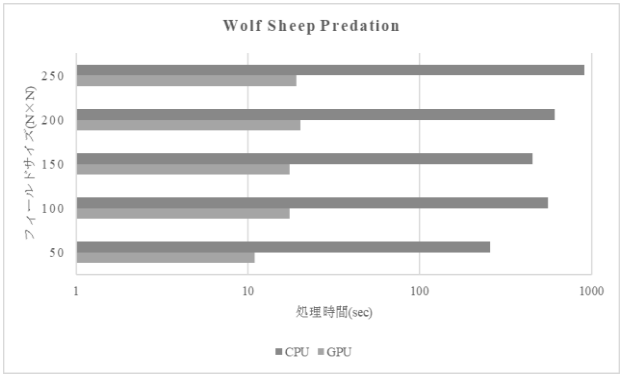

図 8 Wolf Sheep Predation 計測結果(Machine1)

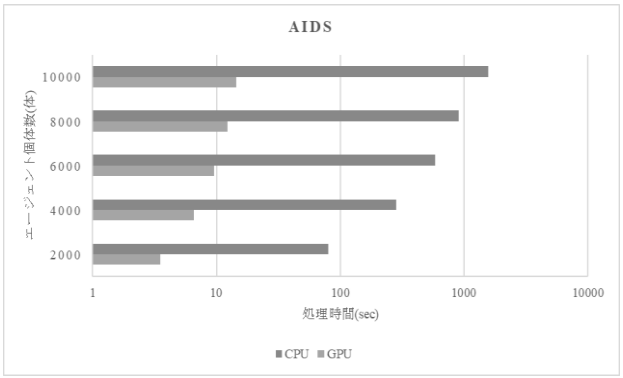

図 6 AIDS 計測結果(Machine2)

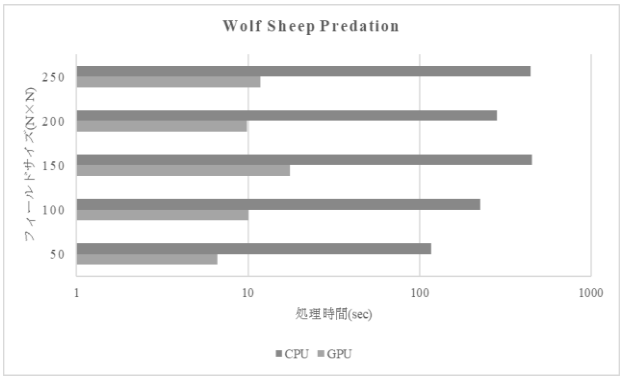

図 9 Wolf Sheep Predation 計測結果(Machine2)

染している場合に自身を感染状態へ変化させる．また感染状態が 30 ターン経過したのちに回復し，非感染状態へ変化させている．

本評価では初期感染者は 1 体，フィールドの大きさは 100x100，ステップ数は 100，初期座標はフィールド内でランダム，エージェントの個体数を 2000 体から 2000 体ずつ増加させシミュレーションを行いその計測結果を図 7，図 6に示す．横軸が計測時間で単位は秒，縦軸がエージェント数を示している．

5.1.2 Wolf Sheep Predation

Wolf Sheep Predation は捕食者と被食者の生態系が安定する状況を検証することができる．このモデルでは複数の個性を持つエージェントを定義する必要があり，互いの状態から自身の状態を更新することが必要である．

本評価ではステップ数 100，羊は 100 体，オオカミは 50 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 50x50 を初期値としている．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いその計測結果を図 8，図 9に示す．横軸が計測時間で単位は秒，縦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．

表 2 ダイバージェンス計測結果

エージェント数	最適化なし	最適化あり	削減率(%)
2000	5716	246	95.70
4000	9812	225	97.71
6000	13956	258	98.15
8000	17781	204	98.85
10000	2246	201	99.10

5.2 ダイバージェンス削減

本環境における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 5.1.1 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．また，ダイバージェンスの計測には NVIDIA が提供するプロファイラを用いて計測を行う．計測の結果を表 2に示す．各ステップでのダイバージェンスの総和と最適化機能によって削減された割合を示す．

5.3 記述性

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 AIDS のプログラムの一部を図 10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

```
for agentid in range(agentManager.getSize()):
  d=agentManager.getAgent(agentid)
  if d["x"]==data["x"] and d["y"]==data["y"] and d["infect"]:
        data["infect"]=True
```

図 10 他エージェント参照例

情報を用いた処理を行っており，様々な記述ができることを確認する．

6 考察

評価実験より本提案環境では Python と比較した場合，AIDS において最大 85 倍，Wolf Sheep Predation において最大 47 倍から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が増加に応じて CPU を用いた場合に比べて GPU を用いることで処理時間の増加が緩やかであり，削減率が大きくなることが見られた．また，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．しかし，計測において個体数を増加させた場合であっても処理時間が短くなる場合が見られた．これはフィールドの大きさに対してエージェント数が多くなり毎ステップ接触する状況が増えることで処理内容が変化し処理時間に影響したと考えられる．

実行時間の削減では GPU による並列処理によるものに加えて，転送命令の呼び出しの削減を行っている．エージェントごとにデータを送受信する場合，エージェント 1 体あたり約 500ms かかるが，転送する際に 1 つの大きなデータに集約することで API の呼び出しや関数呼び出しの回数を減らすことができ，700ms 前後に抑え実行時間を削減している．開発者にはデータの転送を隠蔽しつつ，実行時コストを削減しており開発者の負担を軽減することができていると考える．しかし，GPU 上のメモリからの受信にかかる時間を削減することができたが，送信する際の時間を削減することができず処理時間の大部分を占めている．これを削減することでさらに処理時間の短縮を行うことができると考えている．

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%以上の削減を達成していることがわかる．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，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ダイバージェンスの削減はデータの参照箇所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のだと考える．

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．

また，本環境で利用したソースコードは Python によって処理することができ，デバッグを行うことも可能であり 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．

7 おわりに

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，ユーザの熟練度に合わせた機能の選択や実行環境の変更の設定を設けることで様々なユーザに対応することができる．

今後の課題として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出来ている．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出来ると考える．

8 引用文献

[1] E. Z. Zhang, “Streamling GPU Applications on the Fly - Thread Divergence Elimination through Runtime Thread-Data Remaping,” ICS, 2010.

[2] B. Smith, “A New Parallel Programming Model for Computer Simulation,” Argonne Report ANL, 2014.

[3] 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.

[4] A. M. Geoffrion, “ INDEXING IN MODELING LANGUAGES FOR MATHEMATICAL PROGRAMMING,” MANAGEMENT SCIENCE, 1992.

[5] 佐野 義仁, “マルチエージェントシミュレーションのマルチコア実行環境でのエージェントコード最適化フレームワークの提案,” The 28th Annual Conference of the Japanese Society for Artificial Intelligence, 2014.