F-017

VBO を用いた rbcuda 用 VRAM ライブラリの設計と実装
Design and implementation of VRAM library for rbcuda using Vertex Buffer Object (VBO)

高谷　聡†　Satoshi Takatani
清水　幸紘†　Yukihiro Shimizu
大岩　朗†　Akira Oiwa
芳賀　博英†　Hirohide Haga

1. はじめに

マルチエージェントシミュレーション(Multi Agent Simulation, 以下 MAS)における人工社会は実社会を縮小化したものであるため, MAS で避難経路シミュレーションのような，実社会に近いシミュレーションを行う場合，膨大な数のエージェントが必要となる．したがって，CPU 単一の実行ではシミュレーションの実行時間が膨大になるという問題点がある．そこで本報告では，GPU(Graphics Processing Unit)の高い並列処理能力に注目し，GPGPU(General Purpose computing on GPUs)環境である CUDA をスクリプト言語 Ruby から利用可能にした DSL(Domain Specific Language)である rbcuda を，MAS に利用するための基礎的検討として，描画処理の高速化のためのデータ転送量の削減方法について述べる．

2. GPGPU の概要

近年，GPGPU という GPU を画像処理以外の用途として汎用計算に用いる動きが加速している．GPU は多くのコアが積み込まれており，メモリ帯域幅が高いため CPU よりも高い並列性を持つ. GPGPU を利用する開発環境は NVIDIA 社が提供する CUDA がある．言語は C, C++言語をベースに作られていることから CUDA C と呼ばれる．

図 1 は CPU と GPU の関係を示している．CUDA において CPU 側の環境をホスト，GPU 側の環境をデバイスと呼び区別する．それぞれは独立しており，ホスト側のプログラムをホストプログラム，デバイス側のプログラムをカーネルプログラムと呼ぶ．また，ホストとデバイスはそれぞれメインメモリ(DRAM)，VRAM という特有のメモリ空間を持つ．GPGPU はプログラマがメモリ割り当てや，データ転送，メモリ解放などを手動で行わなければならない．

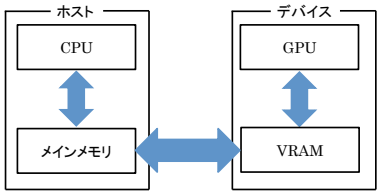

図 1　CPU と GPU の関係

3. rbcuda の概要

従来の CUDA C では，2.で述べたようにメモリ管理をプログラマ自身が手動で行わなければならないという問題点がある. 従来研究ではホストとデバイス間のメモリ管理を自動化する rbcuda が設計された. rbcuda とは, CUDA を Ruby から利用可能にした DSL である. rbcuda はホストとデバイス間のメモリ管理を自動化でき，カーネルプログラムを Ruby で記述可能であるため, プログラマが記述するコード量を削減することもできる. rbcuda は以下の図 2 に示すように，メモリ管理，CUDA 関連処理の自動化を行うシステム，CUDA のホストから呼び出される関数(カーネルプログラム)を GPU 用の実行ファイルに変換する(PTX ファイル)システムにより構成される．

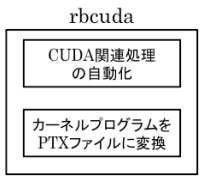

図 2　rbcuda の構成

4. rbcuda の課題

3.で述べた rbcuda は，GPGPU を用いた汎用計算を簡潔に行うことができるが，現状の rbcuda では MAS のような描画処理が必要とする機能が実装されておらず, ホストで描画処理を呼び出さなければならない設計となっている．また，ホストが命令実行中でデバイスを呼び出し, デバイス側ではホストから転送されたデータ群を元に計算を行う. そしてデバイスで実行されたデータ群はホストに転送されて実行結果表示等に利用されるといったデータの流れで実行される. したがって, デバイスで実行するデータ量が増加するとホストに転送するデータ量も増加するため実行時間が大幅に増加するという問題点がある．そこで研究目的として，rbcuda における描画処理を高速化するためにデバイスから転送されるデータ転送量の削減を行うことで実行時間の削減を行う．

5. VRAM ライブラリの実装手法

rbcuda から直接描画を行うために OpenGL という汎用描画ライブラリによる描画処理を組み込む必要がある．rbcuda におけるホストとデバイス間のデータ転送量の削減のために OpenGL により提供される VBO というバッファを用いる．VBO (Vertex Buffer Object) とは OpenGL によって描画される座標や色の情報を保持するためにメモリ上に生成されるバッファである．VBO を用いることで現状の rbcuda においてデバイスで計算したデータ群をホストに転送し，転送されたデータ群を元に描画を行うという工程を省くことができる．VBO を利用するためのライブラリの流れを表すフローチャートを図 3 に示す．

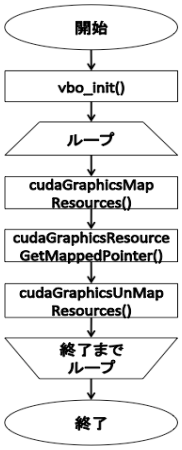

図 3　ライブラリのフローチャート

まず色や座標を表すデータを VBO として利用するために，`vbo_init()`によって VRAM にそれらを表す VBO を作成する．このライブラリの呼び出しはプログラムを実行した初めの一度のみ実行される．これによって VBO のメモリ確保や VBO のポインタを生成され，VBO を CUDA から利用するために VBO を CUDA から利用できるリソースとして登録を行う．これで VBO を CUDA から利用できる準備が完了する. そして `cudaGraphicsMapResources()` によって登録されたリソースを VRAM へのポインタにマップし，`cudaGraphicsResourceGetMappedPointer()`によってそのポインタを取得する. このポインタを参照することで登録されたリソースにアクセスすることができるので，VBO へのアクセスも可能となる. ただリソースがマップされた状態では描画を行うことができないので, 描画を行う際は `cudaGraphicsUnMapResources()`によって

† 同志社大学大学院理工学研究科
Graduate School of Engineering, Doshisha University

アンマップを行う必要がある．上記の手順を繰り返し，VBO を更新することで rbcuda におけるホストとデバイス間のデータ転送量の削減を行うことができる．図 4 に VBO を利用するライブラリの流れを CPU と GPU 間のデータの流れに沿ってまとめた概要図を示す．

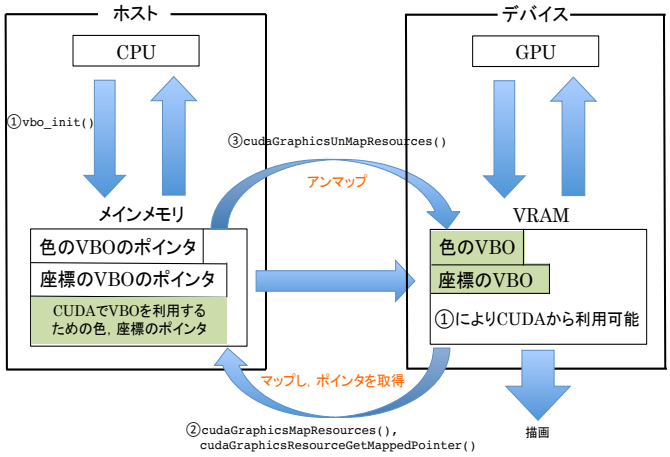

図 4　VBO を利用する手順

6. 実行時間の比較による評価

MAS の代表的なモデルを Ruby, Ruby+CUDA C, Ruby+rbcuda, Ruby+Erbcuda(今回作成した rbcuda)で記述し，実行時間とソースコードの行数の比較を行う．実行時間の比較結果を表 1，表 2，表 3，ソースコードの行数の比較結果を表 4 に示す．また，実行時間を比較する際，エージェント数は分居モデルとライフゲームは 100 万個，200 万個，400 万個，boid モデルはアルゴリズムが他のモデルに比べて複雑であり計算量も膨大であるため実行時間が膨大になるといった問題から 1 万個，2 万個，4 万個に設定し，シミュレーションを 1000 回計測し，シミュレーション 1 回当たりの平均実行時間を計測した．比較対象となる 3 つのモデルの概要を以下に示す．いずれのホストプログラムも Ruby で記述されている．

・分居モデル
　–白人と黒人の 2 種のエージェントがランダムに人種ごとに居住する地域が分かれるモデル
・ライフゲーム
　–生命の誕生や淘汰を単純なルールでシミュレーションするモデル
・boid モデル
　–鳥が群れを成している様子を表すモデル

表 1　分居モデルの実行時間の比較

	Ruby	CUDA C	rbcuda	Erbcuda
100 万体	15.51 s	0.18 s	1.21 s	0.39 s
200 万体	34.09 s	0.37 s	2.07 s	0.75 s
400 万体	75.44 s	0.89 s	15.46 s	1.54 s

表 2　ライフゲームの実行時間の比較

	Ruby	CUDA C	rbcuda	Erbcuda
100 万体	9.78 s	0.02 s	0.78 s	0.03 s
200 万体	10.35 s	0.04 s	1.42 s	0.05 s
400 万体	41.08 s	0.06 s	3.11 s	0.07 s

表 3　boid モデルの実行時間の比較

	Ruby	CUDA C	rbcuda	Erbcuda
1 万体	30.19 s	0.09 s	0.13 s	0.11 s
2 万体	120.48 s	0.22 s	0.49 s	0.41 s
4 万体	480.32 s	0.78 s	1.61 s	1.51 s

分居モデル，ライフゲームでは Erbcuda の方が rbcuda より実行時間が短い．これは VBO を用いることによって CPU と GPU 間のデータ転送が削減できたためだと考えられる．CUDA C と Erbcuda では CUDA C の方が僅かだが実行時が短い．どちらも VBO を用いているがカーネルプログラムを記述する言語が Ruby と C 言語であり，スクリプト言語とコンパイル言語の実行時間の差が生じていると考えられる．

boid モデルでは Erbcuda，rbcuda のエージェントの数を変更し，実行しても実行時間に大差がない．boid モデルでは一つのエージェントが他のすべてのエージェントとの距離を測るアルゴリズムを行っているため，エージェント数が増加するとその実行時間は大幅に増加する．したがって，実行時間に差がないのはライフゲームなどに比べて少数のデータ量をデバイスからホストへ転送する時間が大きいのではなく，同じアルゴリズムを膨大なエージェント数で実行しているためだと考えられる．

7. 各モデルにおける計算量

Erbcuda で今回作成した各モデルはエージェント数を決定し実行したが，大規模シミュレーションのエージェント数は主観的であるため，Big-O 記法による各プログラムのアルゴリズムから計算量を計算した．表 4 に Erbcuda で今回作成した各プログラムの Big-O 記法による計算量の結果を示す．

表 4　Big-O 記法による計算量

	分居モデル	ライフゲーム	boid モデル
計算量	O(1)	O(1)	$O(n^2)$

表 4 と表 1，2，3 から、ライフゲームと分居モデルにおいてはエージェント数 100 万体における実行時間を基準としたとき，エージェント数が 2 倍に増えると実行時間も約 2 倍となっている．boid モデルにおいてはエージェント数 1 万体を基準としたとき，エージェント数が 2 倍に増えると実行時間は 2^2=4 倍になっている．これらから今回測定した実行時間は Big-O 記法の計算量に従っていると考えられる．従って，アルゴリズムが複雑であり計算量が膨大である boid モデルにおいては今回のアルゴリズムでは実行できなかったエージェント数が 100 万体になると約 1100 秒の実行時間がかかると考えられる．

8. おわりに

本報告では従来の rbcuda を MAS に利用するための基礎的検討として，描画処理の高速化のためのデータ転送量の削減を行うことで、従来の rbcuda より高速に描画を行うことができた．また，CUDA のメモリ管理やデータ転送などの GPU の内部構造を理解せずコードを記述することができる．

だが実際に MAS を行うためのライブラリなどが Erbcuda には実装されていないため，今後の課題としては実際にプログラマが MAS を行うために必要となる周囲のエージェントの数や情報を得る等といった機能の実装を行っていく必要がある．

参考文献

[1] 乾正知，“GPU 並列図形処理入門”，技術評論社，2014
[2] 山影進，“人工社会構築指南”，書籍工房早山，2007
[3] 中塚智之，“GPGPU による並列処理のため Ruby フロントエンドの設計と実装”，電子情報通信学会研究報告，114(66)，31-36，2014