GPUを用いたLGA法の高速化

高　橋　直　也*・竹　下　大　樹

Acceleration of LGA method using GPU

Naoya TAKAHASHI* and Daiki TAKESHITA

（平成23年11月25日受理）

LGA method supposes a fluid to be the set of the particle, and it expresses a flow of fluid from interaction between particles. The interaction is calculated in the whole system all at once. Therefore, LGA method is theoretically suitable for parallel computation. Then, it attempted by using GPU that the parallel processing ability is high for the speed-up of the fluid simulation program by the LGA method. CUDA was used for the programming of GPU. As a result, as compared to programs using only CPU, the processing speed was improved 6-7 times.

1. 緒論

格子ガスオートマトン法（以下LGA：lattice gas cellular automaton）法は流体をシミュレーションする手法の一つである。この手法では平面を格子状に分割し，粒子の流れを格子線上に制限する。そして，粒子間で質量と運動量を保存する衝突を繰り返し，流体の速度場を解析する。この相互作用は系全体で一斉に計算されるため，原理的に並列処理に適している。また，近年，画像処理を専門とする補助演算装置であるGPUを画像処理以外の目的に応用するGPGPUという技術が注目されている。GPUは一度に大量のデータを処理することに特化しており，CPUよりも多くの演算ユニットを搭載している。本研究ではこのGPUを用いてLGA法による流体シミュレーションの高速化を図り，CPUのみの場合と比較，検討をすることを目的とする。なお，GPUを用いたプログラムの作成には，NVIDIA社の提供するGPGPU専用統合開発環境であるCUDA（Compute Unified Device Architecture）を用いた。

2. LGA法

LGA法では，平面は図１に示すように正三角形を敷き詰めた格子に分割され，単位質量の粒子は格子点上に存在しており，その移動は格子線上に限られる。また，各粒子には時間と境界について初期条件が与えられており，各粒子は時間ステップごとに格子線上を通り，最近接の格子点まで単位速度で移動する。格子点上では，別の粒子と衝突・散乱して速度方向が変わる。そして，この移動・衝突・散乱のプロセスをあらかじめ決めた衝突側に従ってすべての格子点で一斉に行い，平衡状態になるまで繰り返す。

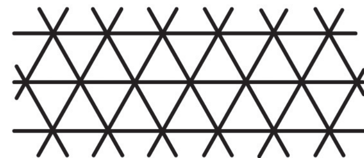


図１　三角格子状に分割された平面

粒子の移動方向のパターンは図２のように格子点上の粒子を中心として計７種類で表され，LGA法によるシミュレーションはこの７種類の状態の粒子の有無により行われる。

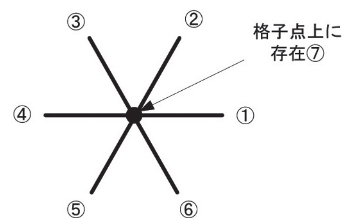


図２　一つの格子点上での粒子の移動方向

*　秋田高専専攻科学生

本研究では衝突側にFHPモデルを用いた。FHPモデルにおいて，考慮される衝突パターンを表1に示す。なお，白丸は格子点上に停止している粒子である。

表1　FHPモデルの衝突例

粒子数	衝突の分類	衝突の種類数			衝突のパターン
		FHP-Ⅰ	FHP-Ⅱ	FHP-Ⅲ	
2体	2体正面	3	3	3	
	2体		6	6	
	1体と停止		6	6	
3体	3体対称	2	2	2	
	3体非対称			12	
	2体正面と停止		3	3	
	2体停止			6	
4体	4体4角			3	
	4体			6	
	3体対称と停止		2	2	
	3体と停止			12	
5体	5体			6	
	4体4角と停止			3	
	4体と停止			6	
衝突パターンの総数		5	22	76	

3.　CUDA

3.1　プログラムの構成

CUDAによるプログラムは大きくホストプログラムとカーネル関数に分類される。ホストプログラムは，CPU上で実行されるプログラムであり，C言語によるプログラムを基本として，GPUを用いるにあたってデータ転送用のメモリ領域の確保やカーネル関数の定義，呼び出しも行う。一方，カーネル関数とはGPU上で実行されるプログラムである。このプログラムはホストプログラムから呼び出されることで実行される。CUDAでのプログラムの流れを図3に示す。

3.2　スレッドによる管理

CUDAでは，プログラムをスレッド単位で管理し，大量のスレッドを同時に実行させる形でGPUによる並列処理を行う。さらに，このスレッドを集めたものをブロックと呼び，同じサイズのブロック

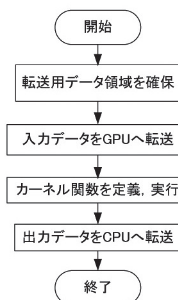


図3　CUDAでのプログラムの流れ

を集めたものをグリッドと呼ぶ。プログラム中ではこのブロックとグリッドという単位も加え，スレッドを管理していく。なお，ブロックは最大3次元の配列として，グリッドは最大2次元の配列として定義できる。スレッド，ブロック，グリッドの関係を図4に示す。

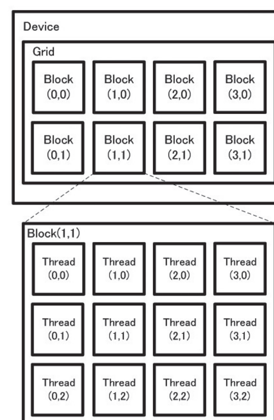


図4　スレッド，ブロック，グリッドの関係

具体的なスレッドの管理はスレッド固有のIDを用いて行う。カーネル関数を定義するとき，ホストプログラムではグリッドとブロックのサイズを決定する。そして，CUDAではこのサイズに従って，以下の型の変数が自動的に用意され，そのスレッドに対応した値が格納されている。

（1）threadIdx型
「threadIdx.x」や「threadIdx.y」と記述することで，ブロック内でのスレッドのIDを取得することができる。
（2）blockIdx型
threadIdx型と記述の仕方は同じで，グリッド内でのブロックのIDを取得することができる。
（3）blockDim型

ブロックの各次元の大きさを取得することができる。
(4) gridDim型
グリッドの各次元の大きさを取得することができる。

これらの変数を用いることでIDを計算により求めることができる。例として図5のグリッドを考えると，各スレッドのIDはblockIdx.x×blockDim.x＋threadIdx.xにより求めることができる。図中の灰色のスレッドの場合，IDを計算するために必要な各変数の値は

blockIdx.x　：1
blockDim.x　：4
threadIdx.x　：2

となり，スレッドのIDは

blockIdx.x ＊ blockDim.x + threadIdx.x
＝1＊4＋2
＝6

となる。

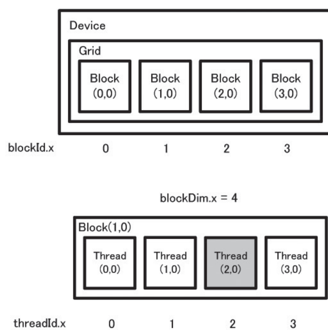


図5　一次元スレッドのブロックと一次元グリッド

3.3 CUDAでのメモリ

CUDAにおいて，GPU側で使用できるメモリは以下の種類がある。

(1) レジスタ，ローカルメモリ
スレッド固有のメモリで，レジスタはGPUチップ内に実装されているオンチップメモリ，ローカルメモリはGPUチップ外のデバイスメモリに配置されているオフチップメモリである。よって，レジスタの方がローカルメモリよりもアクセス速度は高速である。カーネル関数内のローカル変数を保持のために用いられ，レジスタのみではメモリが不足する場合，ローカルメモリをローカル変数の退避領域として用いる。

(2) グローバルメモリ
全スレッドから利用可能な大容量のメモリ領域であるが，アクセス速度は低速である。
(3) シェアードメモリ
オンチップメモリであり，レジスタと同様に高速なメモリである。ただし，アクセス可能なのは同一ブロック内のスレッドのみである。
(4) コンスタントメモリ，テクスチャメモリ
デバイス上のキャッシュ機能を持つメモリだが，GPU側では読み込み専用のメモリとなる。

これらのメモリモデルを図6に示す。

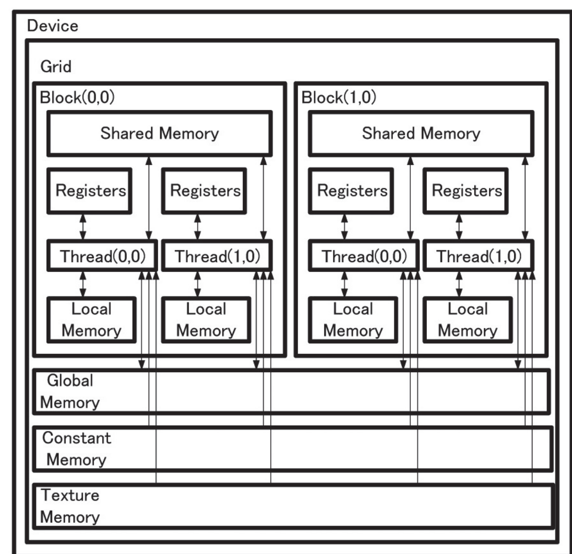


図6　CUDAでのメモリモデル

3.4 メモリへのアクセス

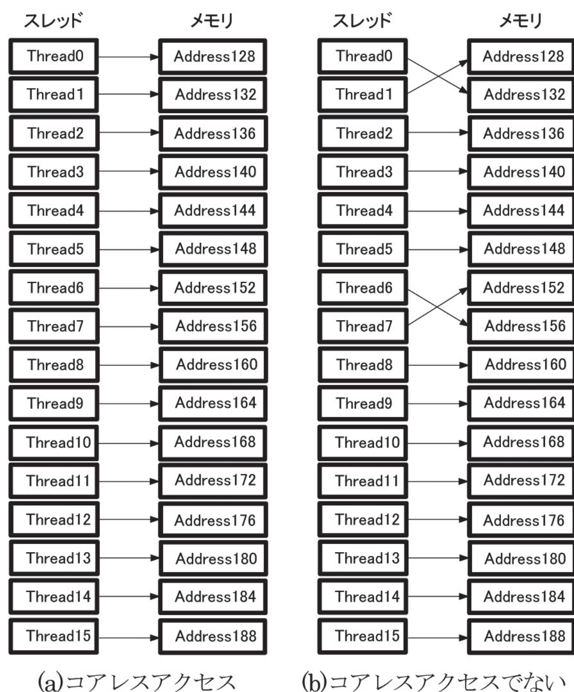
プログラミングを行うときメモリへのアクセス方法によって，その処理速度に違いができる。特徴的なのものは以下の二つである。

(1) コアレスアクセス
グローバルメモリへのアクセス方法に関するものである。CUDAでは一度に32スレッドが実行され，これをワープと呼ぶ。グローバルメモリにアクセスするときスレッドが連続したメモリに同時にアクセスするようにすると，ハーフワープに相当する16スレッド分のデータを一度の処理で転送することができる。これがコアレスアクセスである。そのためには

① データ型のサイズが32，64または128bit

② ハーフワープの先頭スレッドが64byteまたは128byte境界にアライメントされたアドレスにアクセスする。

といった条件があるが，コアレスアクセスにしない場合，アクセス命令の分だけ転送処理が必要となる上に通常の演算命令が４〜数十サイクルかかるのに対し，グローバルメモリからの転送は400〜600サイクルもかかるため，処理速度は大きく低下する。そのため，グローバルメモリを使用するときは，コアレスアクセスにすることが必要となる。グローバルメモリへのアクセスの例を図７に示す。

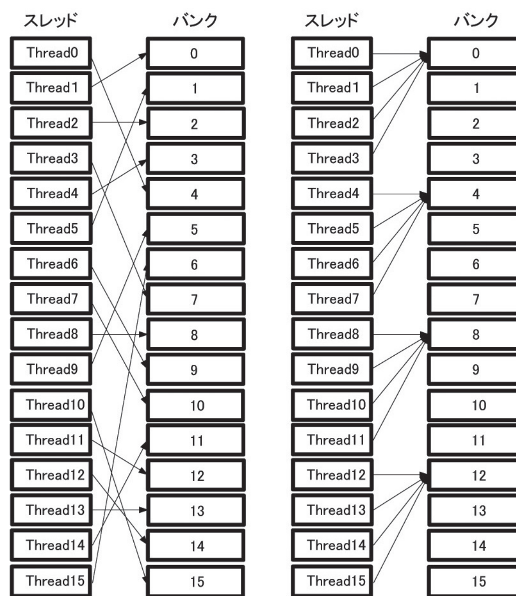


(a)コアレスアクセス　(b)コアレスアクセスでない

図７ グローバルメモリ（float型）へのアクセスの例

⑵ バンクコンフリクト

シェアードメモリへのアクセス方法に関するものである。CUDAではシェアードメモリをバンクという単位に16分割する。そして，ハーフワープの16スレッドが別個のバンクにアクセスすることで全てのバンクを使い，並列処理を行う。しかし，複数のスレッドが同じアドレスにアクセスする場合はアクセス先バンクの衝突が起こる。これがバンクコンフリクトである。バンクコンフリクトが起こると，一つのバンクを複数のスレッドから読み込むことはできないため，そのスレッドの数だけシェアードメモリへのアクセスを行うことになり，効率は低下する。よって，シェアードメモリを用いる場合は，バンクコンフリクトを回避するようにメモリアクセスすることが必要となる。シェアードメモリへのアクセスの例を図８に示す。



(a)バンクコンフリクト無し　(b)バンクコンフリクト有り

図８ シェアードメモリへのアクセスの例

4. GPU上でのLGA法の実装

本研究ではNVIDIA社のGeForce9800GTX＋を用いた。デバイスの情報を表２に示す。表におけるコアの数とはGPUの演算ユニットの数であり，ストリーミングプロセッサ（SP）の数を表す。SPをグループ化したものがマルチプロセッサ（SM）である。よって，SM一つあたり８個のSPを持つことになる。このSPがCUDAでのスレッドに相当し，同じSM内のSPは４サイクルにわたって同じ命令を繰り返す。このためCUDAでは一度に32スレッドを実行することになっている。

表２のデバイスの情報を踏まえて，本研究では22464（縦128×横208）個の格子点をシミュレーションの対象とし，一つの格子点に一つのスレッドを対応させた。この格子点の数はCPUのみで行っていたプログラムのサイズが416×256であり，その比率を残したままグリッドの最大サイズを超えない格子点数を確保した。また，１ブロックのスレッド数は64（縦８×横８）とした。ここで，CPUのみを用いたLGA法のフローチャートを図９に示す。

GPUには図９の灰色で示された処理を担当させ

表２　GeForce9800GTX＋のデバイス情報

マルチプロセッサの数	16
コアの数	128
ワープ	32
グローバルメモリの合計	511MB
ブロック当たりの最大スレッド数	512
ブロックの各次元での最大サイズ	512×512×64
グリッドの各次元での最大サイズ	65535×65535×1
ブロックあたりのシェアードメモリの合計量	16KB
クロック周波数	1.84GHz

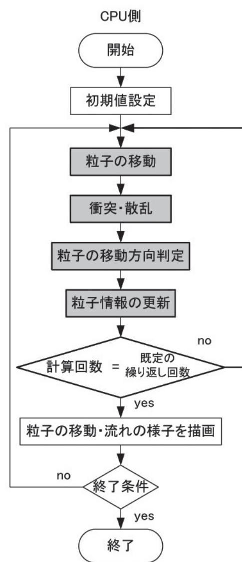


図９　CPUのみを用いたLGA法のフローチャート

る。描画処理を行うまではCPUには結果を転送せず，GPUとグローバルメモリ間のみで粒子情報の更新を行う。GPUを用いたLGA法のフローチャートを図10に示す。GPUを用いた場合，CPU側で初期値を設定した後，転送用のメモリ領域を用いてそのデータをGPUへ転送する。その後，カーネルを実行することで各格子点に対応するスレッドがそれぞれ粒子の移動から粒子情報の更新までを行い，結果をCPUへと転送する。CPU側では描画処理を行うまでに必要な繰り返し回数だけカーネルを実行したかを判断し，再びカーネルを実行するか，グローバルメモリから結果を受け取って粒子の流れを描画する。終了条件は，シミュレーション全体で十分な回数だけ計算を繰り返したかである。

さて，グローバルメモリ上で粒子の情報の更新を行うとしたが，これはGPU側の粒子の移動処理のためである。説明のため，図11に示す格子上に分割

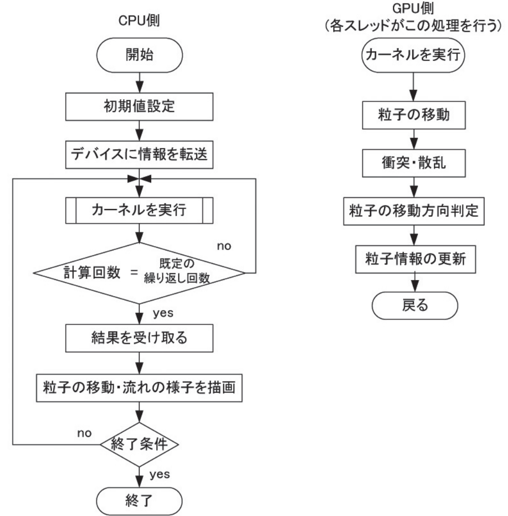


図10　CUDAを用いたLGA法のフローチャート

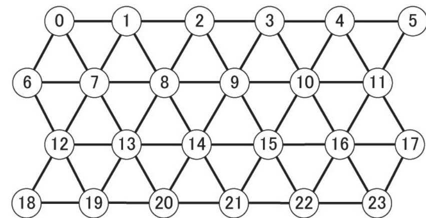


図11　分割された平面と格子点

cell[0]

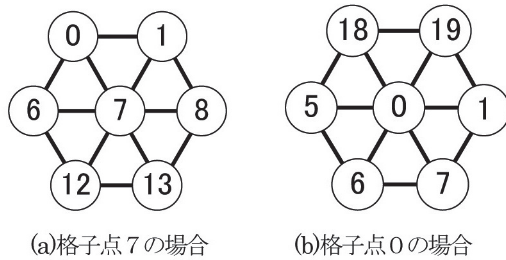
0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

図12　図11の格子点の並びに対応した配列

された平面を考える。番号はその格子点を表すものとする。また，図11の格子点に対応した配列cellを図12に示す。

粒子の移動処理では，各格子点の周囲６点の粒子情報からその格子点に向かってくる粒子の数を判断する。例えば，図10の格子点７では図13(a）のように格子点0, 1, 6, 8, 12, 13の情報を参照する。また，境界に位置する格子点では平面の格子点の並びが連続するものとして考える。つまり，格子点０の場合は図13(b）のようになる。

今，2×2のブロック６つからなるグリッドを考え，各ブロックのシェアードメモリに対応する配列



(a)格子点７の場合　　(b)格子点０の場合

図13　参照する格子点

ブロック0

0	1
6	7

ブロック1

2	3
8	9

ブロック2

4	5
10	11

ブロック3

12	13
18	19

ブロック4

14	15
20	21

ブロック5

16	17
22	23

図14　各ブロックのシェアードメモリのデータ

の一部をコピーする。すると，各ブロックのシェアードメモリには図14のようにデータが格納される。

図14のシェアードメモリのデータを用いて粒子の移動処理を行うとする。しかし，シェアードメモリは同一ブロック内のスレッド以外はアクセスできないという特徴がある。図13の場合，参照するデータが異なるブロックのシェアードメモリに存在するためアクセスすることが出来ず，処理を行うことができない。ブロックのサイズを大きくしたとしても，いずれブロックを越えてアクセスする必要が出てくる。よって，シェアードメモリを用いる方法は粒子の移動処理には適さないことがわかる。

次にグローバルメモリを用いて処理を行うとする。図12のデータ構造のままグローバルメモリに格納し，各スレッドからアクセスする場合，スレッドのIDと配列の添え字が対応しているとすれば，格子点７～10，13～16のアクセスする配列の位置関係は図15となる。なお，tidは中央の格子点に対応するスレッドのIDである。

しかし，境界上に位置する格子点０～6，12，13および17～23の場合は図15のように一律に表すことができない。例えば格子点０の場合は図16(a)，格子点23の場合は図16(b) となる。

格子点ごとにアクセスの記述を変えれば処理は行えるが，その場合は分岐処理となる。GPUでは単一ワープ内のスレッドが同じ命令を実行するときに

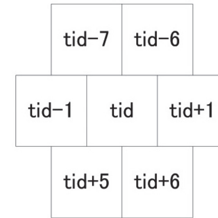
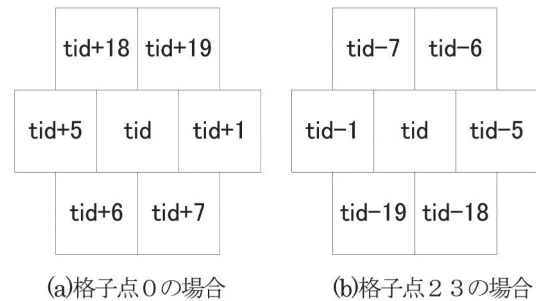


図15　対象格子点とアクセスする配列の位置関係



(a)格子点０の場合　　(b)格子点２３の場合

図16　一律に表せない位置関係

最高効率が得られる。単一ワープ内のスレッドで分岐処理が起きた場合は各分岐を順に実行し，分岐先の異なるスレッドはその間処理を行わない。LGA法では衝突判定のときに分岐処理が必要である。シミュレーションのプロセス上この処理は取り除くことができないので，この処理以外での分岐処理はできるだけ無いこと望ましい。そこで，図17のようにあらかじめ図12のデータの周囲に境界上の格子点が必要とするデータを並べたものを用意し，グローバルメモリに転送することにする。

このデータを用いて図17の周囲を除く配列に粒子の移動処理を適用する。すると，格子点のアクセスする位置関係は図18のように一律に表すことができる。

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47

(a)配列の番号

	18	19	20	21	22	23	
5	0	1	2	3	4	5	0
11	6	7	8	9	10	11	6
17	12	13	14	15	16	17	12
23	18	19	20	21	22	23	18
	0	1	2	3	4	5	

(b)データの対応する格子点番号

図17　新たにデータを配置した配列

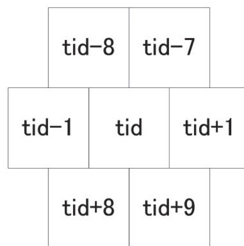


図18　一律に表した新たな配列の位置関係

位置関係を一律に表せたことにより，各スレッドは同じ命令で処理を行う。そのため，分岐処理無しに粒子の移動処理のためのデータへのアクセスを行えるようになる。よって，本研究では粒子情報の更新はグローバルメモリ上で行うことにした。

5.　結果と考察

CPUのみを用いた場合とGPUを用いた場合のシミュレーションの所要時間についての結果を表３に示す。

表３　CPUとGPUでの実効結果

実行モデル	処理時間(秒)		実行速度(倍)
	CPUのみ	GPU使用	
FHP-Ⅰ	200	32	6.3
FHP-Ⅱ	238	35	6.8
FHP-Ⅲ	261	37	7.1

いずれの衝突パターンでも実行速度が6〜7倍になっており，シミュレーションの高速化を実現することが出来たといえる。しかし，CPUがスレッド１つで計算しているのに対しGPU側では22464個のスレッドを用いており，理論的には計算ユニット数の多さだけ計算速度は速くなるはずである。この原因としては，まずグローバルメモリへのアクセスが考えられる。粒子の情報はグローバルメモリ上で更新され，シミュレーションはそのデータを基に繰り返している。通常の演算命令よりも多くのサイクルを必要とするグローバルメモリへのアクセスが多いことで，スレッドでのデータ取得までの待ち時間が発生し，速度の低下に繋がっていると考えられる。データへのアクセスの関係で本研究ではグローバルメモリを用いたが，この結果から高速化のためにはグローバルメモリよりもシェアードメモリを使うことが必要であるといえる。ただし，容量は小さく，アクセスに関する制限もあるためプログラムの並列処理化やデータ構造の改善も必要になるといえる。データの構造に関してはAoS（Array of Structures）ではなくSoA（Structure of Arrays）を用いることが挙げられる。AoSとは構造体を要素とする配列のことであり，例を図19に示す。一方，SoAは本来関連性がある構造体のメンバを個々に配列にまとめ，それらを再び配列とするものである。例を図20に示す。なお，図19，20のx，y，zは各座標のデータを表すとする。

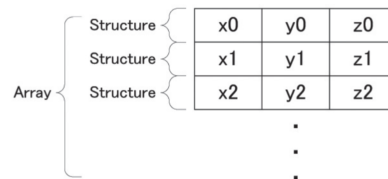


図19　AoSの例

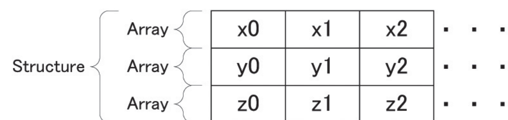


図20　SoAの例

SoAを用いると同じ属性を持つ要素の配列ごとに処理を行えばよく，その処理の内容は同じであるから結果として並列処理を記述しやすくなる。

他の原因としては描画処理時のデータの転送が挙げられる。図10のフローチャートに示すように描画処理を行う場合，一度GPU側からCPU側へと結果を転送する。これがオーバーヘッドとなり，処理速度の低下に繋がったと考えられる。

次に，衝突側で比較すると衝突パターンの多いFHP-Ⅲが最も速度が速くなっている。これは衝突パターンが多いほど分岐処理の負荷が大きくなり，GPUの方がCPUよりも処理能力において優位にな

るためと考えられる。

6. 結言

(1) LGA法にCUDAを適用することで6～7倍のシミュレーションの高速化を実現することができ，GPUの処理能力の高さがわかった。
(2) CUDAで扱うメモリにはコアレスアクセスやバンクコンフリクトといったアクセスに関する制限や容量と速度のトレードオフな関係があり，それぞれの特徴を考えてプログラミングする必要がある。また，扱うデータのデータ型にも注意が必要である。
(3) CUDAではワープに基づいて処理が行われるため，ワープを意識してブロック，グリットのサイズの決定する必要がある。
(4) GPUの性能を引き出すにはプログラムの並列処理化や並列処理に適したデータ構造などが必要となる。

参考文献

1) 加藤恭義，光成友孝，築山　洋：「セルオートマトン法―複雑系の自己組織化と超並列処理―」，森北出版株式会社，47P
2) 内田智史：「C言語によるプログラミング―応用編―　第2版」，オーム社