Vol.2016-HPC-155 No.46
2016/8/10

データレイアウト最適化による GPU用粒子法プログラムの改良

高田 貴正[1,a)]　大野 和彦[1,b)]

概要：流体シミュレーション手法の粒子法は，流体解析に限らず構造解析や衝突解析など幅広い分野で利用されている．一方で問題規模の拡大や高精度化などに伴い計算コストが大きくなっている．そのため，近年性能向上が目覚ましい GPU を用いた並列計算による高速化の研究が行われてきた．粒子法の一つである SPH 法では，密度や圧力の計算など複数のフェーズに分けて段階的に行う．その際，密度や位置座標などのメンバを持った粒子データの構造体配列へのアクセスは，各フェーズにおいて特定のメンバに対して連続的に行われる．構造体の配列に対し特定のメンバのみに連続アクセスすると，メモリ上では不連続なメモリアドレスへのアクセスとなる．このような場合 CPU と比べて GPU ではアクセス効率が大きく低下する．本研究では，各フェーズにおけるメモリアクセス効率が最大になるように粒子データ構造体のメモリレイアウトを最適化することで粒子法プログラムの性能向上を図った．

キーワード：SPH 法，データレイアウト最適化，アライメント，CUDA，GPU

1. はじめに

連続体に関するシミュレーション手法の一つである粒子法は，連続体を粒子の集まりとして粒子同士の相互作用を計算することにより，流体などをシミュレートする手法である [1]．他の手法に対する利点として，形状データの生成が容易であること，大きな変形，ひずみを伴うシミュレーションを高精度に行えることなどが挙げられる．代表的な粒子法に SPH 法がある [2]．SPH 法は流体解析以外にも構造解析や衝突解析などに用いる研究が進み，幅広い分野で利用されている．一方で問題規模の拡大や高精度化などに伴いシミュレーションにかかる計算コストが大きくなっている．

大量のプロセッサで並列に処理できる GPU は近年 CPU に比べて性能向上がめざましく，GPU に汎用的な計算を行わせる GPGPU では標準的な CPU 以上の処理の高速化を実現している [3]．これらのことから，GPU を用いた粒子法の高速化の研究が行われてきた [4], [5], [6].

GPU ではメモリアドレスの連続したデータへ高速にアクセスできる．しかし，各スレッドが構造体配列の特定のメンバへ連続アクセスする場合，メモリアドレスが不連続になるためアクセス速度が低下する．そこで，構造体の配列を配列の構造体に分割することで，このようなアクセスを高速化できる [7]．しかし．SPH 法による運動計算では上記のようなアクセス以外に，構造体の複数メンバへアクセスする場合もある．配列の構造体に分割すると構造体のメンバはメモリアドレスが不連続になるため，かえってアクセス効率が低下する場合がある．

そこで本研究では，複数のアクセスパターンが存在する場合のメモリアクセスを効率化するため，構造体のすべてのメンバを分割するのではなく，いくつかのメンバをまとめた構造体に分割した．さらに構造体配列のアライメントにより構造体データへのアクセスを効率化した．その結果，最適化を行わない場合に比べて Kepler 世代の GPU Tesla K20c では 55%程度，Maxwell 世代の GPU Geforce GTX 980 では 22%程度の高速化を実現できた．

2. 背景

2.1 SPH 法

粒子法の一つである SPH 法は，元々は宇宙の銀河形成のシミュレーション手法として考案された．この計算法を応用し，圧縮性流体や非圧縮性流体，構造解析など幅広い分野で利用されている．

粒子の相互作用の計算は，個々の粒子にかかる密度・圧力の計算，外力の計算，位置の更新と大きく 3 つのフェーズに分けられる．各フェーズにおいて粒子間でのデータ依

[1] 三重大学 Mie University
[a)] takada@cs.info.mie-u.ac.jp
[b)] ohno@cs.info.mie-u.ac.jp

存はなく，個々の粒子の相互作用計算を並列に行える．

2.1.1 SPH 法で用いる構造体

SPH 法の一般的な実装として，粒子 1 個の属性を図 2 のような構造体で表し，扱う粒子の集合をこの構造体の配列に全粒子のデータを格納する．1 粒子の相互作用計算時にその粒子と周辺のすべての粒子のデータが必要なため，上記の構造体配列内のそれぞれに対応した要素にアクセスする．

```
struct Particle{
  float pos_x,pos_y,pos_z;        /* 位置座標 */
  float vel_x,vel_y,vel_z;        /* 速度 */
  float accel_x,accel_y,accel_z; /* 加速度 */

  float density;                  /* 密度 */
  float pressure;                 /* 圧力 */
};
```

図 1 1 粒子の属性を定義した構造体

2.1.2 SPH 法の高速化手法

SPH 法では，全ての粒子間ではなく互いの距離が一定以下の粒子間のみ相互作用する．粒子は空間内を自由に移動できるため，タイムステップ毎に，各粒子に対して相互作用する近傍粒子を探索する必要がある．この探索を高速化する手法の一つとして空間分割法がある．空間分割法ではシミュレーション空間を分割しておき，各粒子がどの分割空間内に存在するかあらかじめ登録して，探索範囲を限定することで探索にかかるコストを削減する．しかし，これらの手法を用いても粒子数の増加に伴い計算やデータアクセスコストは増加するため，さらなる最適化が必要になる．

シミュレーション空間上で近い距離に存在する粒子に対応する構造体配列内の要素を，メモリ上で連続して並ぶようにソートしておくことで，各粒子の相互作用計算時のデータアクセスを高速に行える [8].

2.2 GPU と CUDA

2.2.1 GPU

GPU は演算を行うコアを大量に搭載し多数の処理を並列に実行できる．GPU ではコア数を超えるスレッドを生成でき，これらの大量のスレッドは 32 スレッド単位で分割され管理・実行される．この 32 スレッドのグループをワープという．ワープ内の 32 スレッドは同じ命令を実行する SIMD 型の並列処理を行う．

GPU はキャッシュを搭載した階層型のメモリアーキテクチャを採用している．デバイスメモリへのアクセスはキャッシュのラインサイズである 128byte 単位で行われる．ワープ内のスレッドが同時に同一キャッシュライン上のデータにアクセスすれば，複数のデータ転送を一度のデバイスメモリへのアクセスで行える．このようなアクセスをコアレッシングアクセスという．また，同一ライン内のデータに対して，時間的局所性のあるアクセスを行えば，キャッシュメモリ上にデータが存在するので高速にアクセスできる．

2.2.2 CUDA

CUDA は NVIDIA 社より提供されている GPGPU 用の SDK であり，C 言語を拡張した文法とライブラリ関数を用いて GPU プログラムを開発できる [9]．CUDA では低レベルなコーディングがサポートされており，データアクセスやスレッドマッピングの最適化など，GPU アーキテクチャを意識したプログラミングによるチューニングが可能である．

2.3 GPU 上でのデータアクセス最適化手法

2.3.1 データレイアウト変更による最適化

構造体のメンバは定義順にメモリ上に連続して並び，構造体の配列はこのメンバの並びが連続して繰り返される．たとえば，int 型のメンバ x,y,z を持つ構造体の配列のメモリ上での各データの配置は図 2 のようになる．一般に GPU のスレッドは ID に対応した配列要素を処理するため、連続した領域へ同時にアクセスしコアレッシングアクセスの効果が大きくなる。しかし構造体配列の特定メンバを一斉にアクセスすると、不連続領域へのアクセスとなる．たとえば，図 2 の構造体配列の 0 から 31 番目までの要素のメンバ x に対してワープ内の各スレッドがアクセスした場合，メモリアドレスが不連続なアクセスとなるため，コアレッシングアクセスの効果が低下する．そこで，構造体の配列を配列の構造体に変換することで，このようなアクセスを高速化できる．以降，構造体の配列と配列の構造体を，それぞれ AoS(Array Of Structure)，SoA(Structure Of Array) と記述する．図 2 の AoS を SoA に変換すると，各メンバのメモリ上の配置は図 3 のようになる．各メンバ毎に連続して並んでいるため，元の構造体配列の特定のメンバへのアクセスは，コアレッシングアクセスにより効率化できる．

しかし，特定メンバへの連続アクセスではなく，一つの構造体データの複数のメンバへ時間的局所性のあるアクセスが発生した場合，各メンバはメモリ上に分散配置されているため，アクセス効率が低下する．たとえば，図 3 で 0 番目の要素の各メンバ x,y,z に対して，時間的局所性のあるアクセスした場合，各メンバのメモリアドレスの間隔が 128byte を超えると，デバイスメモリへ 3 回アクセスする必要がある．

AoS の場合，図 2 の並びとなり構造体の全てのメンバがメモリ上で連続しており，まとめてキャッシュメモリ上に格納される．そのため，複数のメンバへの時間的局所性のあるアクセスをした場合，キャッシュヒット率が上昇し高

速にアクセスできる．

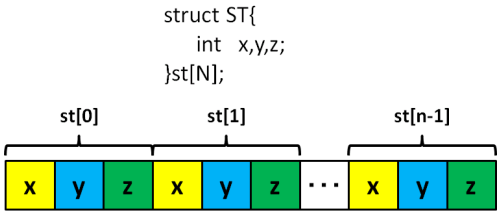

図 2 構造体配列のメモリ上の配置

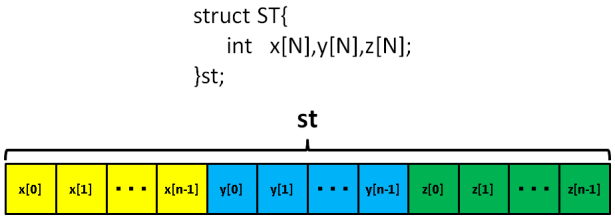

図 3 配列の構造体のメモリ上の配置

2.3.2 構造体のアライメントによる最適化

GPU の各コアによるデバイスメモリへの書き込み，または読み出しは，1,2,4,8,16byte 単位でのアクセス命令のいずれかにより実行される．デバイスメモリへの 4byte 変数の書き込みは，4byte 単位の書き込み命令によって実行される．4byte メンバを 2 個以上持つような構造体の値のデバイスメモリへの書き込みも，4byte 単位の書き込み命令を 2 回実行する．このとき，8byte や 16byte 単位の書き込み命令によって複数のメンバの書き込みや読み出しを 1 度の命令で実行するためには，構造体をアライメントする必要がある．CUDA プログラミングでは構造体のアライメントをサポートしており，`__align__`キーワードによって適用できる．アライメント後の構造体の配列を図 4 に示す．

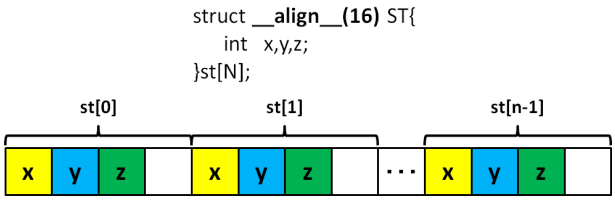

図 4 アライメントを適用した構造体のメモリ上の配置

構造体のアライメントにより，構造体変数への書き込み・読み出しが効率よく行われる．たとえば，4byte のメンバを 4 個持つ構造体を 16byte でアライメントした場合，デバイスメモリへの書き込みは 16byte 書き込み命令 1 回で実行される．4byte のメンバを 7 つ持つ構造体を 16byte でアライメントした場合，16byte 単位の書き込み 1 回，8byte 単位の書き込み 1 回，4byte の書き込み 1 回によって実行される．このようにアライメントにより複数ワードの書き込みまたは読み出しを 1 命令で実行することにより，アクセスを効率化できる．

また，複数ワードアクセスを用いたアクセス効率化は変数への代入・参照単位でしか行われない。たとえば構造体型同士の代入を記述した場合、全メンバのコピーは複数ワードアクセスを組み合わせて最適化したコードが生成されるが、個々のメンバ同士の代入を複数記述した場合、それらのメンバがメモリ上で連続配置されていても、このような最適化は適用されない．

3. 提案手法

SPH 法での相互作用計算時のデータアクセスを高速化するために，データレイアウト変更および構造体のアライメントを用いた最適化を行う．

GPU の SPH 法では，スレッドは各々一個の粒子の運動計算を担当する．運動計算時にスレッドが担当する粒子データはレジスタに格納されるので，デバイスメモリへのアクセスは読み出しと計算結果の書き込みの 2 回である．つぎに，シミュレーション空間上で担当粒子と近い距離にある周辺粒子データへのアクセスは，空間分割法により限定された空間内に存在する全ての粒子データへアクセスする．つまり，限定空間内の粒子の数だけデバイスメモリへのアクセスが必要になる．ただし，キャッシュメモリやコアレッシングアクセスにより高速にアクセスできる場合がある．

これらのことから，周辺粒子データへのアクセスコストが膨大であることがわかる．そのためデータレイアウトは周辺粒子データへのアクセスを対象にした最適化を考える．

各計算フェーズでスレッドが行う担当粒子および周辺粒子のデータへの書き込み・読み出しを表 1 に示す．表内の文字 R，W，-はそれぞれメンバに対して読み出しを行う，書き込みを行う，読み出しも書き込みも行わないことを表す．

表 1 各計算フェーズでの粒子データへのアクセス

フェーズ		位置座標	速度	加速度	密度・圧力
密度・圧力計算	担当粒子	R	-	-	W
	近傍粒子	R	-	-	-
外力計算	担当粒子	R	R	W	R
	近傍粒子	R	R	-	R
位置更新	担当粒子	R/W	R/W	R	-
	近傍粒子	-	-	-	-

3.1 SPH 法でのデータレイアウト最適化

表 1 に示した各フェーズで周辺の粒子データへのアクセスの最適化を図り，3 通りのデータレイアウトを提案する．

粒子データへのアクセス時にメモリ上に連続して並ぶ他の粒子のデータも，同一キャッシュラインに収まれば，まとめてキャッシュメモリに格納される．ソートにより，メモリ上に同空間内の粒子のデータがメモリ上に連続して並んでいる場合，同空間のデータはまとめてキャッシュされるため，近傍粒子探索時のアクセス効率がさらに向上する．

それぞれのデータレイアウトについて詳細を以下に記述する．

3-3-3-2 分割

3-3-3-2 分割後の各構造体を図 5 に示す．位置座標，速度，加速度，密度と圧力それぞれのメンバをまとめた構造体を定義した．密度・圧力計算フェーズでの位置座標のみへのアクセスは，コアレッシングアクセスとキャッシュヒット率の向上により，アクセスが効率化される．外力計算フェーズにおいて，周辺粒子の位置座標，速度，密度と圧力のそれぞれのデータが 128byte 単位でキャッシュされる．もし，相互作用計算後にこれらのデータがキャッシュ上に存在すれば，次のアクセス効率は非常によくなるが，必要なキャッシュメモリの容量が増えたことによりキャッシュミス率が上昇した場合，アクセス効率が低下する．

8-3 分割

8-3 分割分割後の各構造体を図 6 に示す．密度・圧力計算時の必要となる周辺粒子のデータは位置座標のみとなるが，図 6 の Particle1 構造体の配列にアクセスした場合，アクセス時 128byte でキャッシュメモリに格納されるデータは，位置座標と速度，密度，圧力が格納され必要のないデータがキャッシュを占有するため，キャッシュヒット率が低下する．しかし，外力計算フェーズではまとめてキャッシュに格納されるデータが全て計算時に必要になるため，アクセス効率が向上する．

4-4-3 分割

4-4-3 分割分割後の各構造体を図 7 に示す．密度・圧力計算では 8-3 分割に比べて，キャッシュメモリに格納される無駄なデータが減少し，外力計算フェーズでは，3-3-3-2 分割に比べて，必要なキャッシュメモリの容量が減少しているため，キャッシュヒット率を向上できる．

```
struct Particle1{
  float pos_x,pos_y,pos_z;        /* 位置座標 */
};
struct Particle2{
  float vel_x,vel_y,vel_z;        /* 速度 */
};
struct Particle3{
  float accel_x,accel_y,accel_z; /* 加速度 */
};
struct Particle4{
  float density;                  /* 密度 */
  float pressure;                 /* 圧力 */
};
```

図 5 3-3-3-2 分割

```
struct Particle1{
  float pos_x,pos_y,pos_z;        /* 位置座標 */
  float vel_x,vel_y,vel_z;        /* 速度 */
  float density;                  /* 密度 */
  float pressure;                 /* 圧力 */
};
struct Particle2{
  float accel_x,accel_y,accel_z; /* 加速度 */
};
```

図 6 8-3 分割

```
struct Particle1{
  float pos_x,pos_y,pos_z;        /* 位置座標 */
  float density;                  /* 密度 */
};
struct Particle2{
  float vel_x,vel_y,vel_z;        /* 速度 */
  float pressure;                 /* 圧力 */
};
struct Particle3{
  float accel_x,accel_y,accel_z; /* 加速度 */
};
```

図 7 4-4-3 分割

3.2 データレイアウト最適化とアライメント

前述した各データレイアウトの構造体を__align__によりアライメントした．それぞれの構造体のメンバへのアクセスは，複数データのアクセス命令で実行されるためアクセス効率が向上する．各データレイアウトでのアライメントの詳細を以下に示す．

3-3-3-2 分割

図 5 の各構造体に対して，位置座標，速度，加速度のそれぞれの構造体を 16byte でアライメントし，密度・圧力の構造体を 8byte でアライメントした．位置座標，速度，加速度の構造体は 12byte の構造体を 16 バイトでアライメントしているため，各構造体はメモリ上に 4byte の空きが発生し，構造体配列の 1 要素あたり 4byte の無駄が生じ，3 つの構造体の合計で 12byte のメモリ上の無駄が生じる．

8-3 分割

図 6 の各構造体に対して，位置座標，速度，密度・圧力をまとめた構造体と加速度の構造体をそれぞれ 16byte でアライメントした．加速度の構造体は 12byte なので，16byte でアライメントした場合に，構造体 1 変数あたり 4byte の無駄が生じる．

4-4-3 分割

図 7 の各構造体に対して，位置座標，密度をまとめた構造

体，速度，密度をまとめた構造体，そして加速度の構造体をそれぞれ 16byte でアライメントした．加速度の構造体は 12byte なので，16byte でアライメントした場合に，構造体 1 変数あたり 4byte の無駄が生じる．

4. 評価

4.1 評価プログラムと実行環境

粒子を 800,000 個用いた SPH 法によるダム崩壊シミュレーションを評価プログラムとして各手法の実行速度向上率を計測した．本プログラムでは粒子のデータを持つ構造体配列を 1 ステップ毎にソートすることにより，探索時のアクセス最適化を行う手法を用いた [12].

評価環境は IntelCore i7-930, メモリ 6GB, Tesla K20c と Intel Xeon CPU E5-1620, メモリ 16GB, GeForce GTX980 を搭載したそれぞれの計算機で行った．Tesla K20c は Kepler 世代アーキテクチャ，GeForce GTX980 は Kpler の次世代となる Maxwell 世代アーキテクチャを採用している [10], [11].

4.2 性能評価

AoS，SoA および 3.1 節で述べたデータレイアウトとそれらのアライメントの有無の組み合わせによる各手法の性能評価を行った．AoS は 16byte でアライメントし，その他のデータレイアウトは 3.2 節で述べたアライメントを適用した．ただし，SoA はアライメントできない．以降，それぞれの手法を表 2 に示したように，データレイアウトとアライメントの有無を-で繋いで表記する．

それぞれの評価プログラムの実行時間を各環境で計測し，AoS-noalign を用いた場合に対する各最適化手法の速度向上率を図 8，9 に示す．

Tesla K20c を用いた場合，4-4-3 分割とアライメントを組み合わせた手法が速度向上率が 55%程度と最も大きかった．GeForce GTX980 を用いた場合では，8-3 分割とアライメントを組み合わせた手法が速度向上率が 21%程度と最も大きかった．

各環境で最も向上率が大きかった手法を用いて，粒子数を 100,000 と 500,000 と 800,000 と 3 通りの粒子数で実行時間を計測し、データサイズによる速度向上率の変化を評価した。結果を図 10，11 に示す．いずれの場合も速度向上率はほぼ一定であり，提案手法がデータサイズによらず安定した効果が得られることを示している．

5. 考察

5.1 データレイアウト最適化による速度向上

図 8，9 より，アライメントせずデータレイアウト変更のみによる最適化では，速度向上率の変化が小さく-10%〜5%程度であった．これは本評価プログラムでは粒子データのソートにより，アクセス効率がある程度向上しているため，本手法による効率化の影響が小さくなったと考えられる．

Tesla K20c を用いた場合，SoA の適用によるデータレイアウト最適化が最も実行速度が向上しており，5%程度の速度向上を実現している．これは，各計算フェーズで構造体

表 2 各最適化手法の組み合わせ

データレイアウト	アライメント	
	有	無
AoS	AoS-align	AoS-noalign
SoA	–	SoA-noalign
4-4-3 分割	4-4-3-align	4-4-3-noalign
3-3-3-2 分割	3-3-3-2-align	3-3-3-2-noalign
8-3 分割	8-3-align	8-3-align

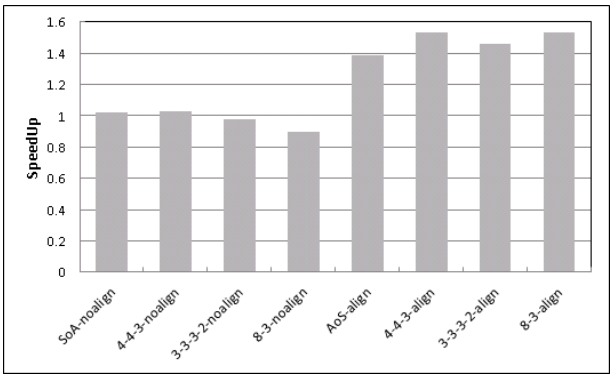

図 8 最適化による速度向上率 (Tesla K20c)

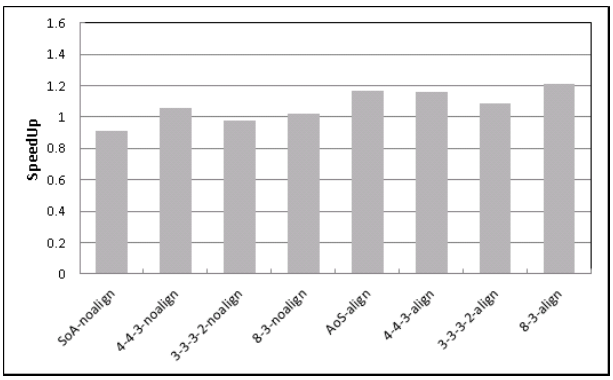

図 9 最適化による速度向上率 (GeForce GTX980)

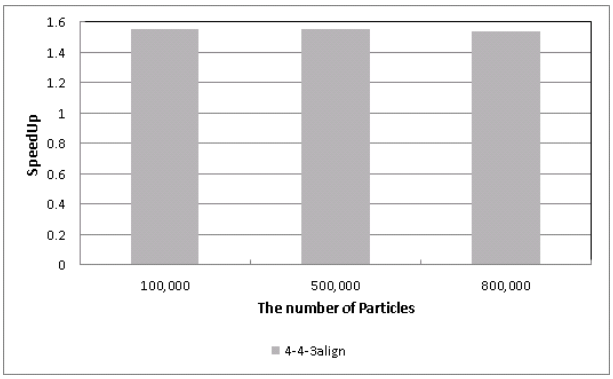

図 10 データサイズ変更による速度向上率の変化 (Tesla K20c)

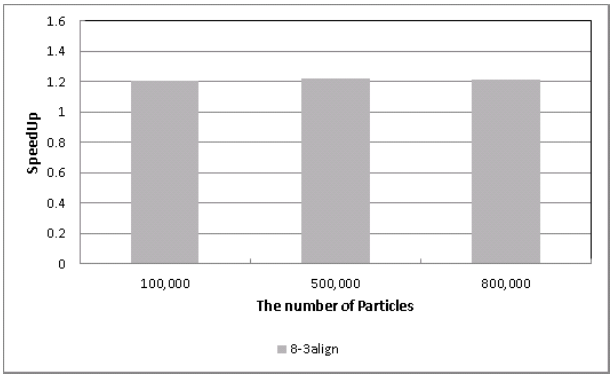

図 11 データサイズ変更による速度向上率の変化 (GeForce GTX980)

配列内で連続した要素の特定のメンバへのアクセスが頻繁に行われたため，SoA の適用によりこれらのアクセス効率を大きく向上できたと考えられる．

また，8-3 分割では実行速度が 10%程度低下している．外力計算フェーズでの計算結果を加速度に書き込む際に，AoS では事前の粒子データへのアクセスで加速度もまとめてキャッシュに格納されるため，一部結果の書き込み時にキャッシュヒットしていた．しかし，8-3 分割では事前の粒子データアクセスでは，加速度以外のデータがキャッシュに格納されるため，加速度データへのアクセス時にキャッシュヒットしなくなり，アクセス効率が低下したからだと考えられる．

GeForce GTX980 を用いた場合，構造体の 4-4-3 分割によるデータレイアウト最適化がもっとも実行速度が向上しており，5%程度の速度向上を実現している．これは 3.1 節で述べたように他のデータレイアウトに比べて，各計算フェーズでキャッシュを効率よく利用できたからだと考えられる．また，SoA を適用した場合に実行速度が低下している．これは Maxwell 世代ではメモリアドレスが不連続なアクセスが改良され，コアレッシングアクセスによる実行速度向上率が低下したため，構造体配列の特定の要素の複数メンバへのアクセス効率の低下が大きく影響し，実行速度が低下したと考えられる．

5.2 データレイアウト変更とアライメントによる速度向上

図 8，9 より，データレイアウト変更のみの最適化時に遅くなったレイアウトを，アライメントと組み合わせた場合，他の手法以上の速度を向上できている．アライメントにより各データアクセスにかかる時間が短くなり，時間あたりのアクセス命令回数が増えキャッシュメモリ上のデータの入れ替え頻度が増加する．データレイアウト最適化により必要になるデータがキャッシュに格納されやすくなるので，入れ替えが頻繁に発生しても，キャッシュ上にデータが存在する確率が高くなり，キャッシュヒット率が向上したからだと考えられる．

Tesla K20c での計測結果では，AoS-align を適用した場合でも，40%程度の速度を向上できた．データレイアウト最適化とアライメントを組み合わせることで，最大 55% 程度の速度向上を達成できた．

3-3-3-2 分割では他のデータレイアウトより速度向上率が低い．4byte 変数を 4 個持つ構造体のデータアクセスは，16byte のアクセス命令 1 回で実行される．対して，4byte 変数を 3 個持つ構造体のデータアクセスは，8byte のアクセス命令 1 回と 4byte のアクセス命令 1 回の合計 2 回に分けて実行される．3-3-3-2 分割では，このような構造体を 3 つ用いる必要があり，他の手法よりもデータアクセス時の命令回数が増加するため実行速度が低下したと考えられる．

GeForce GTX980 の計測結果では AoS-align を適用した場合でも，15%程度の速度を向上できた．データレイアウト最適化とアライメントを組み合わせることで，最大 22%程度の速度向上が達成できた．3-3-3-2 分割では Tesla K20 での場合と同様の理由から，実行速度が低下したと考えられる．

8-3 分割が 4-4-3 分割よりも速くなっているのは，外力計算フェーズで必要になる周辺粒子データが，8_3 分割ではメモリ上に連続しているため，効率よくキャッシュを利用できたからだと考えられる．

6. まとめと今後の課題

本研究では構造体配列の連続した要素の特定のメンバへのアクセスと，構造体配列の特定の要素の複数メンバへのアクセスという，2 種類のアクセスパターンが存在する粒子法プログラムに対して，効率よくアクセスできるデータレイアウト最適化および構造体配列のアライメントを提案・実装し，実行時間の比較を行った．

その結果，データレイアウト最適化によるアクセス時のコアレッシング率およびキャッシュヒット率の向上と，アライメントによる 1 命令あたりのアクセス効率化を組み合わせることで Kepler 世代の GPU Tesla K20c では最大 55%程度，Maxwell 世代の GPU GeForce 980 では最大 22%程度の速度向上が実現できた．また，アライメントによるアクセス効率化では時間あたりのアクセス命令数が増加するため，キャッシュメモリの書き換えが頻繁に発生する．このような場合，データレイアウト最適化によりキャッシュメモリ上に必要になるデータを効率よく格納することで，キャッシュヒット率を向上しアクセスの最適化が期待できる．

今後の課題として，各計算フェーズで異なるデータレイアウトに変更することで，各フェーズに合わせたアクセス効率化によりさらなる高速化が期待できる．また，データアクセス以外にも計算フェーズ内での分岐命令による，アイドルスレッドの削減による高速化などがあげられる．

参考文献

[1] W, T, Reeves. *Particle Systems - a Technique for Modeling a Class of Fuzzy Objects.*, ACM Transactions on Graphics,pp.359-375, 1983.
[2] J. J. Monaghan. Smoothed particle hydrodynamics. *Annu.Rev.Astrophys.*,Vol. 30, pp. 543-574, 1992.
[3] *GPGPU.org: General-Purpose computation on Graphics Processing Units*,
入手先 ⟨http://www.gpgpu.org/⟩ (参照 2016-07-10).
[4] 原田隆宏, 田中正幸, 越塚誠一, 河口洋一郎. グラフィックスハードウェアを用いた個別要素法の高速化, 日本計算工学会論文集, Vol.2007, No.20070011(2007).
[5] S, Green. *Particle Simulation using Cuda*,NVIDIA Whitepaper,(2012).
[6] 平林久義, 佐藤雅弘. 線形リストを用いた粒子法の近傍粒子探索. 日本計算工学会論文集,Vol. 2010,No.20100001,2010.
[7] J. A. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. D. Liu, and W. Hwu. *Otimization and architecture effects on GPU computing workload performance.* In Proc.Innovative Parallel Computing 2012, InPar 2012, pp. 1-10, 2012.
[8] 渡辺勢也, 青木尊之, 都築怜理, 下川辺隆史. 接触による粒子相互作用の *GPU* 計算での近傍探索手法. 情報処理学会論文誌. コンピューティングシステム. Vol.8, No.4, pp.50-60,2015.
[9] *NVIDIA Developer CUDA Zone*,
入手先 ⟨http://developer.nvidia.com/category/zone/cuda-zone⟩, (参照 2016-07-10).
[10] NVIDIA.*NVIDIA Kepler GK110 Architecture Whitepaper.*
入手先 ⟨https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf⟩, (参照 2016-07-10).
[11] NVIDIA.*NVIDIA GeForce GTX 980*
入手先 ⟨http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF⟩, (参照 2016-07-10).
[12] 高田貴正, 大野和彦. 等間隔格子を用いた *SPH* 粒子法プログラムの *CUDA* による高速化. 情報処理学会研究報告ハイパフォーマンスコンピューティング, Vol.2015-HPC-150, No.31,pp.1-6(2015).