Vol.2017-HPC-158 No.29
2017/3/10

メモリアクセス命令の実行回数とアドレスを利用したGPUプログラムのデータレイアウト自動最適化

中井 裕登[1,a)]　大野 和彦[1,b)]

概要：GPU は多数のスレッドを並列に実行できるが，高速化にはハードウェアの特性を意識したコーディングが求められる．特にメモリ上のデータレイアウトは，プログラムの性能への影響が大きく最適化が必要であるため，本研究ではプログラムに最適なデータレイアウトへの自動変換手法を提案する．本手法では，動的解析によってカーネル関数におけるメモリアクセス命令のアクセス先のアドレスと実行回数を収集する．このアドレスを用いて各命令におけるコアレシングアクセスの有無を判定し，レイアウト変換のための重みを決定した．また，実行回数を用いてその重みの補正と構造体メンバのクラスタリングを行った．実アプリケーション 1 本とベンチマーク 3 本を用いた評価の結果，一般的な高速化手法である SoA への変換と比較して最大 1.60 倍の高速化を達成した．

キーワード：CUDA，GPU，データレイアウト

1. はじめに

グラフィックス処理用プロセッサである GPU に汎用的な演算を行わせる GPGPU[1] は，CPU 以上の計算性能を発揮することもあり，近年，期待が高まっている．GPU は多数のスレッドを並列に実行できるが，高速化にはハードウェアの特性を意識したコーディングが求められる [2].

メモリ上のデータレイアウトは，プログラムの性能への影響が大きいため最適化が必要である．さらに，一般にデータ構造はプログラマにとってプログラムの理解が容易となるように記述されるため，メモリアクセスの時間的局所性が考慮されていない場合がある．しかし，プログラマが最適なレイアウトを見つけることは困難であり，自動最適化が望まれている．

メモリ上のデータレイアウトの最適化に必要な情報を収集する方法として静的解析と動的解析がある．静的解析を用いる場合，特定のプログラムに対して最適なレイアウトを決定できない可能性がある．例えば，間接参照を用いるものや，実行時の条件分岐によって配列の添え字式の値が変わるプログラムは実行するまでメモリアクセスパターンが不明である．

そのようなプログラムへの解析の対応と高速化を目的として，本研究では，GPU 上の処理を記述したカーネル関数におけるメモリアクセス命令の実行回数とアクセス先のアドレスを利用したデータレイアウトの自動最適化手法を提案する．我々の手法では動的解析によって実行回数とアドレスを取得し，これらを用いて各命令におけるコアレシングアクセスの有無を判定し，レイアウト変換のための重みを決定する．ただし，提案手法は構造体の配列を用いた GPU プログラムを対象としている．実アプリケーション 1 本とベンチマーク 3 本を用いた評価の結果，一般的な高速化手法である SoA への変換と比較して最大 1.60 倍の高速化を達成した．

以降，まず 2 章で研究の背景を示し，続く 3 章で GPU プログラムにおけるデータレイアウト及びメモリアクセスの自動最適化に関する関連研究を紹介する．そして，4 章で提案手法の概要を示す．その後，5 章と 6 章で手法を示し，7 章で評価の結果を示す．最後に，8 章で本論文をまとめる．

2. 背景

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

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

```
struct AOS{          struct SOA{
   int x, y, z;          int x[N], y[N], z[N];
}aos[N];             }soa;
```

図 1 AoS と SoA の定義

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 L2 キャッシュのラインサイズである 128byte 単位で行われる．以後，本論文におけるキャッシュとは L2 キャッシュを指すものとする．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 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．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 AoS と SoA

構造体の配列 (Array Of Structure) と配列の構造体 SoA(Structure Of Array) の定義例を**図 1** に示す．また，このときメモリ上では**図 2** のように配置される．以下では，構造体の配列を AoS，配列の構造体を SoA と表記する．各スレッドが配列の各要素を処理対象としており各メンバを参照したときのアクセス先は，図 2 の網掛部分になる．この性質により，連続した領域へ同一ワープ内のスレッドがアクセスするとコアレシングアクセスの効果が大きくなる．しかし図 2 のように AoS 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．そこで，AoS を SoA に変換することで連続した領域へのアクセスとなり，このようなメモリアクセスを高速化できる．だが，メモリアクセスパターンによっては AoS の方が高速となることもあるため，プログラムに適したレイアウトを選択する必要がある．

2.4 Array-of-Structure-of-TiledArrays(ASTA)

Sung ら [3] は，一般に高速とされる SoA に代わるレイアウトとしてタイル化 AoS(ASTA) への変換手法を提案した．定義の例を**図 3** に示す．また，このときメモリ上では

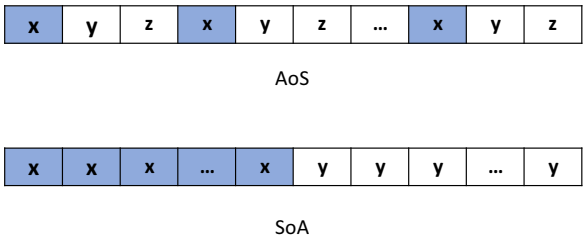

図 2 AoS と SoA のメモリ上の配置

```
struct ASTA{
   int x[4], y[4], z[4];
}asta[N/4];
```

図 3 ASTA の定義

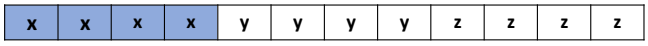

図 4 ASTA のメモリ上の配置

図 4 のように配置される．このレイアウトは各構造体メンバがタイル数ずつ配置されることが特徴である．図 4 はタイル数を 4 にしたときの例である．このタイル数によって性能が変わるため，プログラムに適したタイル数を設定する必要がある．これにより，AoS が持つ空間的局所性への優位性と SoA が持つ連続アクセスによる優位性を両立している．しかし，Sung ら [3]，Kofler ら [4] の NVIDIA GPU を用いた評価では SoA と同程度の性能となっており，AoS，SoA より優れるとはいえない．

2.5 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，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 5** に示す．

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

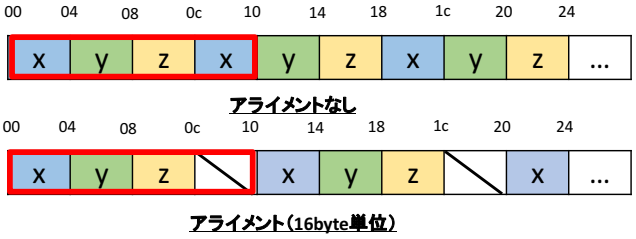

図 5 アライメントを適用した構造体のメモリ上の配置

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. 関連研究

GPU プログラムを対象として，メモリ上のデータレイアウトを自動最適化する研究がある．Kofler ら [4] は，OpenCL(Open Computing Lan-guage) で記述された GPU コードのデータレイアウトを自動最適化するために Kernel Data Layout Graph (KDLG) を定義し，それを用いた手法を提案している．この既存手法ではメモリアクセスの局所性を表す KDLG を生成するために，静的解析により必要な情報を取得する．そして，デバイスの L1 キャッシュサイズを基に KDLG を用いた構造体メンバのクラスタリングとレイアウトの決定を行い，GPU コードの自動変換を行う．

Weber ら [5] は静的解析と経験的解析のいずれかを使用して GPU コードを最適化する MATOG フレームワークを開発した．この既存手法では AoS，SoA，AoSoA をサポートし，最適なレイアウトを選択する決定木を構築する．

これらの静的解析を用いた手法に対して，Fauzia ら [6] は動的解析を用いたメモリアクセス最適化フレームワークを開発した．解析によって各メモリアクセス命令のアクセス先のアドレスを取得し，アドレスが連続していればコアレシング，そうでなければ非コアレシングという特徴付けを行った．そして，非コアレシング命令がアクセスする配列の添え字式を書き換えることで，コアレシングアクセスの効果を向上させた．しかし，データレイアウトの変更は実装していない．

上記のデータレイアウト最適化の研究では，AoS，SoA，AoSoA の中からレイアウト選択している．しかし，これら以外にも有用なレイアウトは存在する．Mei ら [7] は，IDW 補間の高速化として，AoS，SoA，AoSoA，アライメントされた AoS(AoS-align) などでの評価を行った．この中で AoS-align は AoS，SoA の両方と比べて高い性能を発揮した．本研究では SoA と AoS-align の中から最適化レイアウトを決定する．AoS-align については，アライメントの効果を高めるために構造体メンバのクラスタリングを行う．

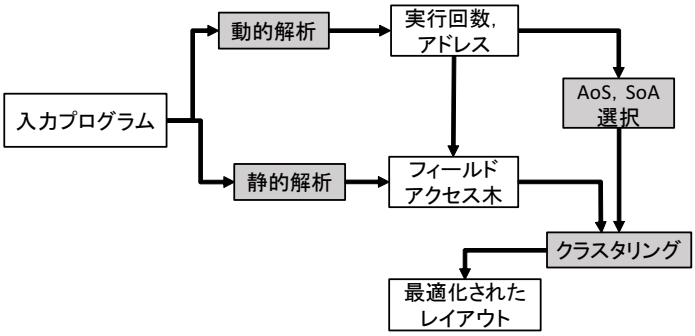

図 6 提案手法の流れ

表 1 動的解析によって収集する情報

instID	各命令を識別する
warpID	各命令を実行したワープを識別する
threadID	各命令を実行したスレッドを識別する
addr	各命令のアクセス先のアドレスを示す
kernelID	各命令が実行されたカーネルを識別する
loopID	各命令が実行されたループを識別する

4. 提案手法の概要

図 6 に提案手法における処理の流れを示す．本手法は動的解析，静的解析，レイアウト選択，クラスタリングの 4 フェーズを持つ．AoS で書かれた CUDA コードを入力とし，その入力に対して動的解析と静的解析を行う．動的解析によって取得する各カーネル関数でのメモリアクセス命令の実行回数とアクセス先のアドレスを用いて AoS と SoA のどちらがプログラムに適しているか判定する．この判定結果が AoS の際は静的解析と動的解析で取得したデータを用いてフィールドアクセス木を生成する．そして，それを基に構造体メンバのクラスタリングを行う．

5. 動的解析結果を用いたレイアウト選択

データレイアウト最適化のためにカーネル関数での各メモリアクセス命令のアクセスパターンを判定する必要がある．そこで，Fauzia ら [6] が用いた動的解析手法を拡張し，実行時に**表 1** に示す情報を出力する．この `addr` を参照することでアクセス先のアドレスを，同じ `instID` を持つ `uniqueID` をカウントすることで実行回数を取得する．これらを用いることで AoS と SoA のどちらが適しているかを判定する．`kernelID`，`loopID` はフィールドアクセス木の `Count` を決定する際に用いる (6.1 節)．**図 7** に動的解析結果例の一部を示す．

uniqueID	instName	instID	warpID	threadID	addr	kernelID	loopID
0	st[tid].x	0	0	0	0x70041e000	0	0
1	st[tid].x	0	0	1	0x70041e00c	0	0
2	st[tid].x	0	0	2	0x70041e018	0	0
3	st[tid].x	0	0	3	0x70041e024	0	0
4	st[tid].x	0	0	4	0x70041e030	0	0
5	st[tid].x	0	0	5	0x70041e03c	0	0
6	st[tid].x	0	0	6	0x70041e048	0	0

図 7 動的解析結果

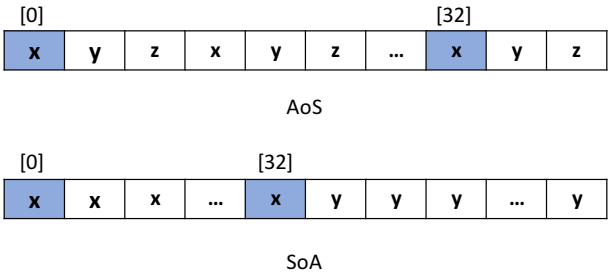

図 8 UnCoaless での AoS と SoA

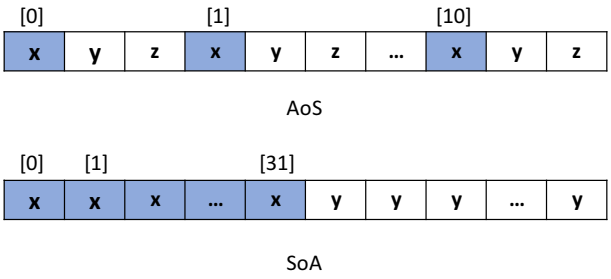

図 9 Coaless での AoS と SoA

5.1 アクセス先のアドレスを用いた重み付け

各命令について, アドレスを参照し, UnCoaless, Coaless, ShareCoaless の 3 種類へ分類した．これらの分類はそれぞれ以下のように定義する．

- UnCoaless
  同一ワープ内の 1 スレッドのみが同一キャッシュライン上のデータにアクセスしている状態を指す．このとき，AoS は SoA よりメモリアクセス性能が高くなる．これは AoS と SoA は共にコアレシングにならないが，AoS は SoA よりキャッシュ効率が良いためである．例を**図 8** に示す．
- Coaless
  同一ワープの複数スレッドが同一キャッシュライン上のデータにアクセスしている状態を指す．このとき，SoA は AoS よりメモリアクセス性能が高くなる．これは SoA は連続アクセスになるが AoS は連続アクセスにならないためである．例を**図 9** に示す．
- ShareCoaless
  同一ワープ内の複数スレッドが同じアドレスにアクセスしている状態を指す．このとき，AoS は SoA よりメモリアクセス性能が高くなる．これはレイアウトに関係なくコアレシングになり，AoS は SoA よりキャッシュ効率が良いためである．例を**図 10** に示す．

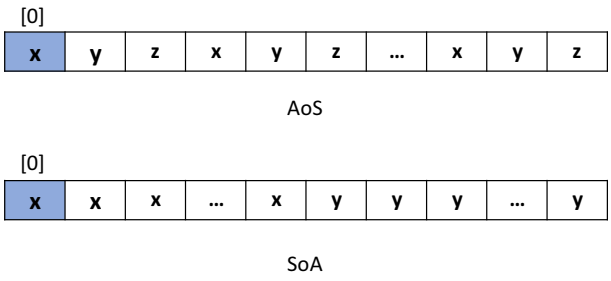

図 10 ShareCoaless での AoS と SoA

図 11 に分類アルゴリズムを示す．2 行目から 24 行目では全 uniqueID を走査し，各命令における同一キャッシュライン上のデータにアクセスする同一ワープ内のスレッド数であるコアレシングスレッド数をカウントする．4 行目と 5 行目でアクセス先が同一キャッシュライン上となるアドレスの範囲を決定する．キャッシュラインサイズは 128byte であるため，128 を足し合わせている．6 行目から 17 行目では同じ命令を実行する同一ワープ内のスレッドのアクセス先のアドレスについて処理する．7 行目で対象となるアドレスが同一キャッシュラインサイズ上にあるか判定し，真となる場合は 8 行目でコアレシングスレッド数に 1 を加算する．9 行目で対象となる 2 つのアドレスが同じであるか判定し，真となる場合は ShareCoaless であるため，8 行目で判定用変数に 1 を代入する．13 行目の offset は while ループ内で処理した uniqueID の数をカウントしている．また，15 行目では各命令の実行回数をカウントしている．一度 while ループで処理した uniqueID を以降の for ループで処理しないようにするため，16 行目で uniqueID に offset を足し合わせ，for ループでの処理対象を調節している．プログラムではループなどで同じ命令が複数回実行される場合がある．そこで，18 行目から 19 行目で平均を求め，それを 5.2 節以降の重み補正で用いるコアレシングスレッド数としている．そして，21 行目以降で，平均コアレシングスレッド数が 0 ならば UnCoaless，1 以上なら Coaless，同じアドレスに複数のスレッドがアクセスしている（share[instID]=1）場合は ShareCoaless に分類する．

図 7 に分類したものが**図 12** である．各命令について AoS のアクセス性能が SoA より優れる UnCoaless と ShareCoaless には重み w_1 を，SoA に劣る Coaless には重み w_2 を設定する．以後，本論文においては $w_1 = 1$，$w_2 = -1$ としている．さらに，2 種類の補正を重みに適用することで命令の実行回数やコアレシングアクセスの効果を加味したレイアウト選択を行う．

```
1: 入力 : 動的解析結果
2: for 全 uniqueID do
3:    offset ← 1
4:    min_range ← addr[uniqueID]
5:    max_range ← addr[uniqueID] + 128
6:    while instID[uniqueID] = instID[uniqueID+offset]
      &   warpID[uniqueID] = warpID[uniqueID + offset]
      do
7:       if addr[uniqueID+offset] ≥ min_range &
         addr[uniqueID + offset] ≤ max_range then
8:          threadNUM[instID] ← threadNUM[instID] + 1
9:          if  addr[uniqueID] = addr[uniqueID + offset]
            then
10:            share[instID] ← 1
11:         end if
12:      end if
13:      offset ← offset + 1
14:   end while
15:   exeNUM[instID] ← exeNUM[instID] + 1
16:   uniqueID ← uniqueID + offset
17: end for
18: for 全 instID do
19:   threadNUM[instID]
      ←threadNUM[instID] / exeNUM[instID]
20: end for
21: for 全 instID do
22:   if threadNUM[instID] = 0 then
23:      class[instID] ← UnCoaless
24:   else if threadNUM[instID] > 0 then
25:      class[instID] ← Coaless
26:   else if share[instID] > 0 then
27:      class[instID] ← ShareCoaless
28:   end if
29: end for
```

図 11 分類アルゴリズム

5.2 コアレシングスレッド数による補正

Coaless に関しては，コアレシングスレッド数によって AoS と SoA の性能差が変わるため，式 (1) によって補正する．n はコアレシングスレッド数を指す． 例えば，図 11 の `instID 0` はコアレシングスレッド数が 11 なので式 (1) より，重みは $w_2 \times 11 \ / \ 32 = -0.34$ に補正される．

$$\text{重み} = \begin{cases} w_1 & (UncCoaless, ShareCoaless) \\ w_2 \times n \ / \ 32 & (Coaless) \end{cases} \tag{1}$$

5.3 実行回数を用いた重みの補正

5.2 節で重みの補正を行った各命令に対して，実行回数を掛けることで実行回数に応じた補正を行う．例えば，図 11 の `instID 0` は実行回数が 800 なので 5.2 節で補正された重みはさらに $-0.34 \times 800 = -272$ に補正される．

instID	class	exeNUM	CoalessThreadNUM
0	Coaless	800	11
1	ShareCoaless	800	32
2	Coaless	800	11
3	ShareCoaless	832	32
4	ShareCoaless	832	32
5	Coaless	32	11

図 12 分類結果

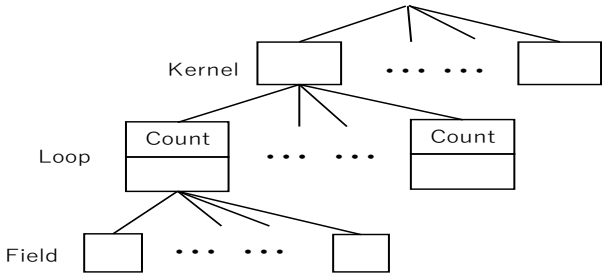

図 13 フィールドアクセス木

補正を行った各命令の重みの総和を取り，それが正であれば AoS を，負であれば SoA が適したレイアウトとして選択される．例では総和が正となるため AoS が選択される．

6. クラスタリングによるアライメント効果の向上

レイアウト選択フェーズで AoS が選択された際は構造体メンバに対してクラスタリングを行う．プログラム実行時の時間的局所性はコード上の局所性と相関がある．そこで，本研究では動的解析で取得した実行時間と，静的解析で取得したコード上の局所性を基に構造体メンバをクラスタリングする．構造体のサイズやメンバの組み合わせによってアライメントの効果は変化する．そこで，アライメントの効果が高くなるように，これらの点に注目する．

6.1 フィールドアクセス木

本手法で用いるフィールドアクセス木の構造を**図 13** に示す．これは各カーネル関数において構造体メンバへのアクセスが発生するループとその回数を表し、これを参照することでアクセス頻度を基にしたクラスタリングを行う．

フィールドアクセス木は 3 階層で構成される．`Kernel` 階層のノードはコード上の各カーネル関数を表す．`Loop` 階層のノードは親ノードのカーネル内のループを表し，`Count` を持つ．この `Count` の値として動的解析で取得した実行回数を用いる．このとき，どのカーネル，ループにおける実行回数であるかを判別するために動的解析で取得した `kernelID`，`loopID` を使用する．`Field` 階層は親ノードのループ内でアクセスされる構造体メンバを示す．同じ親ノードを持つ `Field` はコード上での局所性が高いことから，同じクラスタにすることでキャッシュヒット率の向上が期待できる．

```
struct St{
   float vx, vy, vz;
   float px, py, pz;
   float ax, ay, az;
   float prs, dns;
};
```

図 14 クラスタリング前の構造体

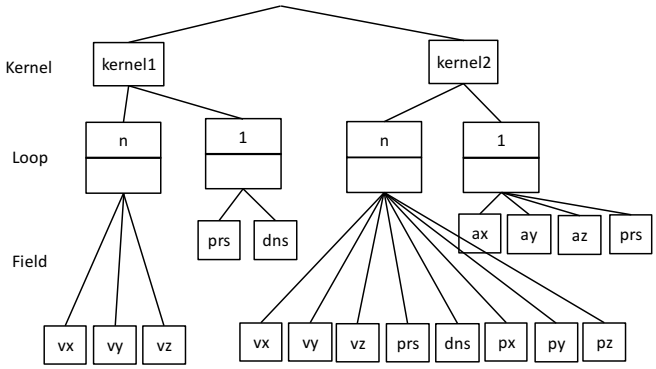

図 15 クラスタリング前のフィールドアクセス木

6.2 アクセス木を用いたクラスタリング

以下の手順でクラスタリングを行う．

(1) 木の生成
全カーネルを静的解析し，その結果と実行回数を基にフィールドアクセス木を作成する．

(2) カーネル内統合
木を走査し，同一カーネル内で Count が同じ Loop ノード以下を統合する．

(3) カーネル間統合
異なるカーネル間で共通する Field ノードを持ち，且つ Count が同じ Loop ノード以下を統合する．このとき，アライメントの効果が高くなるように，Loop ノードが持つ Field ノードの合計サイズが 16byte の倍数になるように調整する．

(4) 変換
変形された木を基に，構造体を書き換える．統合が完了した時点で同じ親ノードを持つ Field ノードは同一構造体のメンバとする．

図 14 の構造体を持つプログラムに対して手順 1 を適用すると**図 15** のアクセス木を得る．続いて手順 2 を適用すると**図 16**，手順 3 を適用すると**図 17** のアクセス木になる．最後に手順 4 を適用すると，**図 18** の構造体定義が得られる．

7. 評価

7.1 評価プログラムと実行環境

実アプリケーションである SPH 法によるダム崩壊シミュレーション [8] と，ベンチマークである IDW[7]，Rodinia benchmark[9][10] の cfd と particle filter を評価プログラムとして，提案手法で最適化されたレイアウトと既存のレイアウトを用いたときの実行速度を計測した． SPH は構造体配列の添え字式に間接参照が用いられているため，実行前にアクセス先を予測することは不可能である．cfd は添え字式が多項式であるため，実行前にアクセス先を予測することは困難である．particle filter の構造体配列の添え字式は単項式だが，変数の値が条件分岐によって変化する．これらのことから，SPH，cfd，particle filter については静的解析によるメモリアクセスパターンの判定は困難であると考えられる．

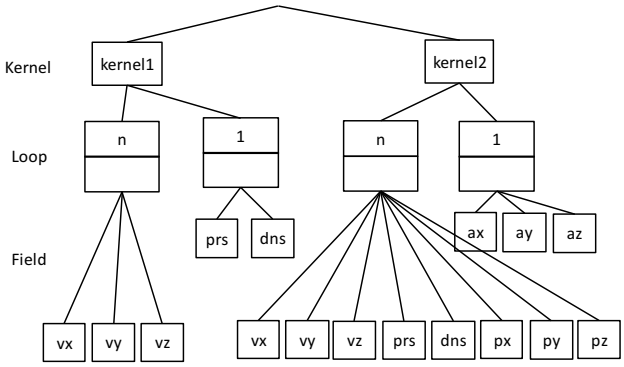

図 16 カーネル内統合後のフィールドアクセス木

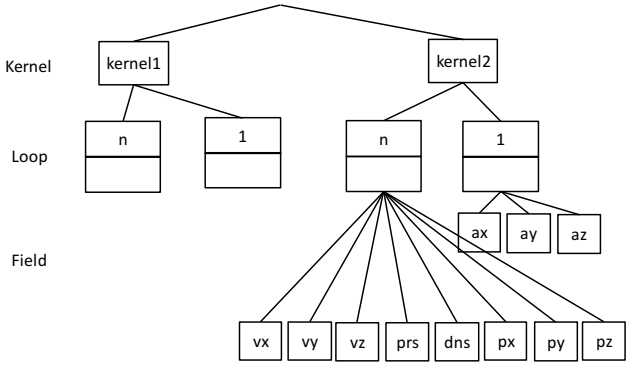

図 17 カーネル間統合後のフィールドアクセス木

```
struct __align__(16) St1{
   float vx, vy, vz;
   float px, py, pz;
   float prs, dns;
};
```

```
struct __align__(16) St2{
    float ax, ay, az;
};
```

図 18 クラスタリング後の構造体

評価は Intel Core 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 Kepler の次世代となる Maxwell 世代アーキテクチャを採用している [11][12].

7.2 性能評価

AoS と SoA，提案手法で最適化されたレイアウトによる性能評価を行った．それぞれの評価プログラムの実行時

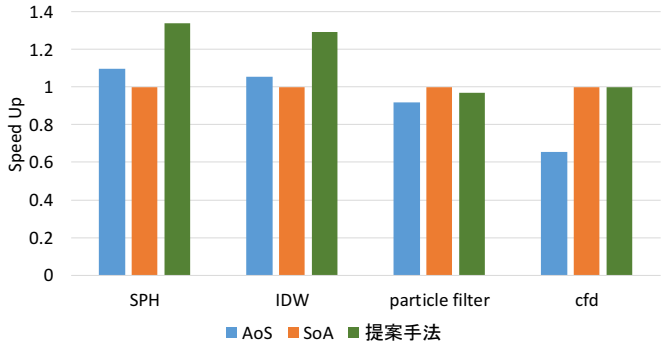

図 19 最適化による速度向上率 (GeForce GTX980)

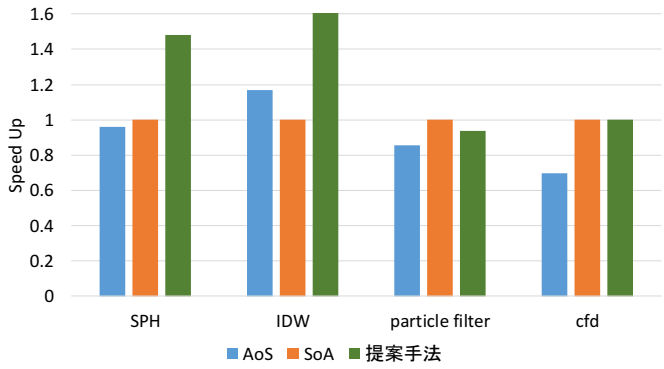

図 20 最適化による速度向上率 (Tesla K20c)

間を各環境で計測し，SoA に対する各レイアウトの速度向上率を**図 19**，**図 20** に示す．GeForce GTX980 を用いた場合，提案手法によって最大 1.33 倍の高速化を達成した．Tesla K20c を用いた場合は，最大 1.60 倍の高速化となった．

レイアウト選択フェーズでは SPH，IDW，particle filter は AoS が，cfd は SoA が選択された．評価結果から提案手法により SPH，IDW，cfd を最適なレイアウトへ変換できたことがわかる．Particle filter については最適なレイアウトに変換できなかった．これは，レイアウトのメモリアクセス性能はメモリアクセス命令だけでなく計算命令による影響も受けるからであると考えられる．負荷の大きな計算命令がある場合は，その直前のメモリアクセス時間は隠蔽されてしまう．しかし，最適なレイアウトに対する性能差はわずかであり，本手法では安定した効果が得られることを示している．

また，間接参照や多項式によるアクセスを行う SPH と cfd に対して適切なレイアウトを選択できていることから，動的解析は複雑なメモリアクセスを行うプログラムに対して有効であるといえる．

8. まとめ

本研究では AoS で書かれた CUDA コードのデータレイアウトを自動で最適化することを目的として，動的解析によるレイアウト選択とクラスタリングによるアライメントの効果向上を提案し，実装ならびに性能評価を行った．本手法では，コアレシングアクセスとなる場合でも AoS が SoA より高速となるパターンがあることに注目し，それを動的解析によって検出した．これによってメモリアクセス命令の分類を増やすことで，コアレシングアクセスとなるパターンを AoS が高速となるものと SoA が高速になるものへ分類することが可能になった．そして，それを基に重み付け，重み補正をすることで適切なレイアウト選択が可能となった．

その結果，実アプリ 1 本とベンチマーク 3 本の計 4 本の内 3 本に対して最適なレイアウトへの変換ができた．また，一般的な高速化手法である SoA への変換と比較して Kepler 世代の GPU Tesla K20c では最大 1.60 倍，Maxwell 世代の GPU GeForce 980 では最大 1.33 倍の高速化を達成した．

今後の課題として，レイアウト選択精度の向上と SoA が選択された際の更なる高速化が挙げられる．配列の添え字を書き換えることで SoA の性能を向上させることが可能であるため，本手法による効果の増大が期待できる．

参考文献

[1] *GPGPU.org: General-Purpose computation on Graphics Processing Units*, 入手先 ⟨http://www.gpgpu.org/⟩ (参照 2017-02-05)
[2] *NVIDIA Developer CUDA Zone*, 入手先 ⟨http://developer.nvidia.com/category/zone/cuda-zone⟩, (参照 2017-02-05)
[3] Sung, I-Jui, Geng Daniel Liu, Wen-Mei W. Hwu.: *DL: A data layout transformation system for heterogeneous computing*, Innovative Parallel Computing (InPar), 2012
[4] Kofler, Klaus, Biagio Cosenza, Thomas Fahringer.: *Automatic data layout optimizations for gpus*, European Conference on Parallel Processing, 2015.
[5] Weber, Nicolas, Sandra C. Amend, Michael Goesele.: *Guided profiling for auto-tuning array layouts on GPUs*, Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, 2015.
[6] Fauzia, Naznin, Louis-Nol Pouchet, P. Sadayappan.: *Characterizing and enhancing global memory data coalescing on GPUs*, Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2015
[7] Mei, Gang, and Hong Tian.: *Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation*, SpringerPlus 5.1, 2016
[8] 高田貴正, 大野和彦.: 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, 研究報告ハイパフォーマンスコンピューティング (HPC) 2016.46,pp1-7,2016
[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, K. Skadron.: *Rodinia: A benchmark suite for heterogeneous computing*, In IISWC, pages 4454. IEEE, 2009
[10] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, K. Skadron.: *A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads*,

In IISWC,pages 111. IEEE, 2010
[11] NVIDIA.: *NVIDIA Kepler GK110 Architecture Whitepaper.*
入手先 ⟨http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf⟩,
(参照 2017-02-05)
[12] NVIDIA.: *NVIDIA GeForce GTX980.*
入手先 ⟨http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF⟩,
(参照 2017-02-05)