

修士論文

題目

ネスト構造を含む
GPU プログラムの
データ構造最適化

指導教員

大野 和彦

2024 年

三重大学大学院 工学研究科 情報工学専攻
コンピュータアーキテクチャ研究室

伊藤 亘輝 (422M504)

ネスト構造を含む GPU プログラムの データ構造最適化

伊藤 亘輝

内容梗概

グラフィック処理に開発されたプロセッサである GPU は、多数のコアが内蔵されているため高並列処理に適しており、粒子シミュレーションや AI など多くの分野で需要が高い。しかし、メモリアクセスがボトルネックになる場合が多いため、メモリアクセスの最適化がプログラムの性能改善に有効である。

一般に実プログラムでは、単なる配列や構造体だけでなく、これらを組み合わせた複雑なデータ構造が使用される。このとき、同一の内容であってもデータ構造が異なる場合、メモリアクセス効率が変化し、実行速度にも影響する。比較的単純なデータ構造である構造体の配列 (SoA) や配列の構造体 (AoS) については、相互変換を行ったり、静的解析により実行効率の高い方を判断したりする研究が行われてきた。しかし、構造体のメンバに構造体を含むなど複数段にネストした多段ネスト構造に対応できていない。

本研究では、従来手法を一般化して多段ネスト構造を含むプログラムの実行効率化を目指している。多段ネスト構造では、SoA と AoS のように内容は等価だがメモリ上のレイアウトが異なるデータ構造が多数存在するため、手動ですべて導出するのは困難である。また、データ構造を変更したとき、その構造にアクセスするコードもすべて変更する必要があり、これも手動で行うのは煩雑である。本稿では多段ネスト構造を含むプログラムを対象として、等価なプログラム群を自動で生成する手法を提案する。具体的には、等価なデータ構造をすべて生成し、さらに各データ構造に対応するようアクセスコードの変更も自動で行う。

本手法を実装し、多段ネスト構造を含むプログラムを対象にして評価を行ったところ、等価なデータ構造をすべてを生成し、それぞれのアクセスコードも正しく変更できた。また、生成されたプログラム群をすべて実行しオリジナル版に対する速度向上率を求めたところ、オリジナル版のコードに対して最大で 1.42 倍のコードが発見でき、本手法による等価なプログラムの生成がプログラムの高速化に有効であることが示された。

Data structure optimization of GPU programs including nested structures

Koki ITO

Abstract

GPUs, processors developed for graphics processing, are suitable for highly parallel processing due to their many built-in cores and are in high demand in many fields, such as particle simulation and AI. However, since memory access is often a bottleneck, optimization of memory access is effective in improving program performance.

In general, real programs use not only simple arrays and structures but also complex data structures that combine them. At this time, even if the content is the same, if the data structure differs, the memory access efficiency changes, which also affects the execution speed. For relatively simple data structures, such as arrays of structures (SoA) and structures of arrays (AoS), research has been conducted to perform mutual conversion or to determine the one with higher execution efficiency through static analysis. However, they have not been able to deal with multi-stage nested structures that nest multiple levels, such as including structures as members of a structure. This study aims to generalize conventional methods to improve the execution efficiency of programs containing multistage nested structures. In a multistage nested structure, there are many data structures such as SoA and AoS that are equivalent in content but have different layouts in memory, and it is difficult to manually derive all of them. When a data structure is changed, the code that accesses the structure must also be changed, which is also complicated to do manually. In this paper, we propose a method for automatically generating equivalent programs for programs containing multi-stage nested structures. Specifically, all equivalent data structures are generated, and the access codes are automatically changed to correspond to each data structure.

We implemented the proposed method and evaluated it on a program containing a multi-stage nested structure, and found that it generated all equivalent data structures and correctly changed the access codes for each of them. The speedup ratio of the generated programs compared to the original version was calculated, and it was found to be at most

1.42 times faster than the original version, indicating that the generation of equivalent programs by this method is effective in speeding up the programs.

目次

1	はじめに	1
2	研究背景	3
2.1	GPU	3
2.2	データ構造のレイアウト	4
3	提案手法	6
3.1	提案手法の概要	6
3.2	前提条件	6
3.3	用語説明	7
3.3.1	構造体ノード	7
3.3.2	コードモデル	8
3.3.3	レイアウト変換命令	8
3.3.4	データレイアウト	9
3.3.5	親構造体, 子構造体	9
3.3.6	コンフリクト	9
3.4	手法の流れ	10
3.5	データレイアウトの変換	10
3.5.1	レイアウト変換命令の定義	11
3.6	コードモデルの生成	17
3.7	データレイアウトの探索	20
3.8	コード変換	22
3.9	構造体のコピー	23
3.10	問題点と対応方法	25
4	性能評価	30
5	関連研究	33
6	おわりに	35
	謝辞	36
	参考文献	37

目 次

2.1 GPU アーキテクチャ	5
2.2 AoS, SoA のデータレイアウト	5
3.3 コードモデル	9
3.4 提案手法の流れ	10
3.5 ExpandMemberToVariable	13
3.6 ExpandMemberToMember	14
3.7 ConvergenceVariableToMember	14
3.8 ConvergenceMemberToMember	15
3.9 TransposeVariable	16
3.10 TransposeMember	16
3.11 入れ替えの例	23
3.12 SoA から AoS 変換の例	24
3.13 SoA から AoS 変換の相互変換コード	24
4.14 Boid モデルの実行時間グラフ	31
4.15 交通シミュレーションの実行時間グラフ	32
4.16 粒子シミュレーションの実行時間グラフ	32

表 目 次

3.1 レイアウト変換命令	11
-------------------------	----

1 はじめに

GPU用のプログラミングフレームワークであるCUDA[4]は主にGPGPUを目的として活用されている。GPGPUとはGPUを画像処理ではなく、科学技術計算やシミュレーションのような計算目的で活用する技術である。GPGPUを目的としたプログラムでは実行時間短縮のために並列計算を導入している為、最適化によるさらなる時間短縮の必要がある。実際にコード変換、プログラミング技法、アプローチによって性能を最適化しようとする研究が行われている[6]。GPUではメモリアクセスの効率が実行速度に大きくに影響する。そのため、プログラマはアクセス効率の良いアクセスパターンのコードを書くことが求められる。特に多段ネストデータ構造によっては等価なデータ構造のパターンが多く、メモリアクセス効率が最適となるデータ構造をプログラマが求めることが困難である。また、従来手法では実プログラムで使われる多段ネストデータ構造に対応できないため、従来手法より一般的なデータ構造最適化手法が必要である。そこで本研究では、CUDAのソースコードを対象として、任意の多段ネストデータ構造に対して等価なデータ構造を総当たりで生成する手法と、ソースコードの変換手法を提案する。以降、2章では研究背景としてCUDAとコアレスアクセスの仕組みについての解説を行う。3章で提案手法の説明

をする. 4 章では, 多段ネストデータ構造の含まれるソースコードを対象として, 手法に従って作成した等価なデータ構造のパターンすべてで実行時間を比較した. 5 章では等価なデータ構造によってメモリアクセス効率を最適化しようとする関連研究を紹介する. 最後に, 6 章で本論文をまとめる.

2 研究背景

2.1 GPU

CUDA は Nvidia 製 GPU 用の開発プログラミングフレームワークである。CUDA で最適化されたコードを書くにはハードウェアへの理解が必要である為、まず GPU アーキテクチャについて説明する。図 2.1 のように、GPU は多数のストリーミングマルチプロセッサ (SM) を持っており、容量の大きいデバイスメモリと L2 キャッシュがある。SM 内には複数の CUDACore が存在し、SM につき一つの L1 キャッシュとシェアドメモリーを持っている。そして CUDACore それぞれが 1 スレッドを担当するため非常に多くの並列計算ができるといえる。

GPU はワープといわれる単位で動作する。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。32 スレッド上で同じ命令が実行されるため、通常ならばワープ中でグローバルメモリから値をロード、ストアする場合は 32 回のロード、ストアが行われることになる。実際はデバイスメモリへのフェッチ時に、近傍のライン上 (32word) に要求がまとまっていればそのトランザクションを合体させることができる。これをコアレスアクセスと呼ぶ。コアレスアクセスによってデバイスメモリから 32 トランザクション未満でデータをフェッチできると言える。そのた

めコアレスアクセスを生かすようにコードを書くとフェッチ回数が少なくなり、性能向上につながる。実際に [3] の研究ではスレッドごとのストライドアクセスの間隔によって、デバイスメモリへのレイテンシに影響があることが示されている。

2.2 データ構造のレイアウト

次にデータ構造のレイアウトについて説明する。配列の構造体 (AoS) や構造体の配列 (SoA) など、同じ内容のデータに対して複数の表現形式が可能である。説明のため、図 2.2 では三次元ベクトルを表す Vector3 構造体を例にしている。AoS の場合は構造体メンバがスカラーで宣言されているのに対して、変数は配列で宣言される。SoA の場合は構造体メンバが配列で宣言されているのに対して、変数がスカラーで宣言されている。GPU ではコアレスアクセスの仕組みがあるため、メモリへのアクセスパターンが実行時間に与える影響が大きい。プログラマが複合データ構造の含まれるソースコードから最適なデータ構造を見つけるには、取りうるパターンが多く困難である。そのため、本研究ではソースコードを静的解析し変換可能なデータレイアウトのパターンを列挙する手法と、ソースコードの変換手法を提案する。

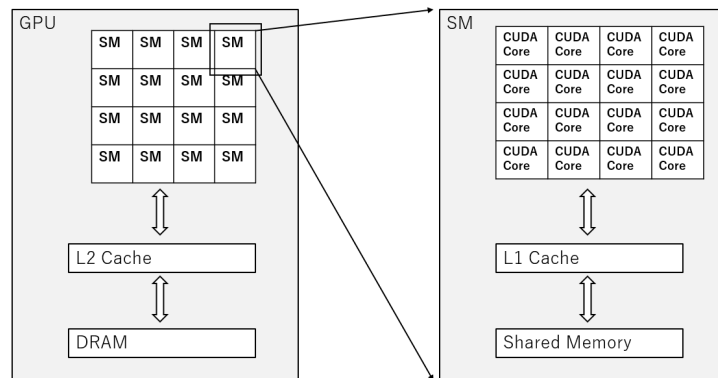


図 2.1: GPU アーキテクチャ

• **Array of Struct**

```
struct Vector3{
    float x,y,z;
};
Vector3 v[N];
```



• **Struct of Array**

```
struct Vector3{
    float
    x[N],y[N],z[N];
};
Vector3 v;
```

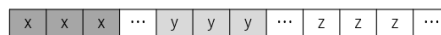


図 2.2: AoS,SoA のデータレイアウト

3 提案手法

3.1 提案手法の概要

提案手法では,CUDA コードを入力とする. 入力されたコードを静的解析し, コード中にデータ構造のレイアウトから新しく等価なデータ構造を生成する.

3.2 前提条件

静的解析を行う都合上, 本研究が対象とするソースコードに以下の制約を設けるものとする.

- プログラム中でデータ構造として使用される連続メモリ領域は, すべて大きさが静的に固定であること.
- 上記のデータ構造はフィールド構成がコード中で明示的に示されていること.
- 構造体のメンバについて, メモリ上の並び順に依存したコードを含まないこと.
- データ構造中にポインタを含まないこと.

- 共用体や cast により文脈によって同じフィールドを異なる方として扱うコードは対象外である.

3.3 用語説明

3.3.1 構造体ノード

静的解析によってソースコード中のデータ構造と変数を抽出する. メンバまたは変数は (Listing1) のように表す. 1 つの構造体について, 2 つの構造体ノードを生成する. 1 つ目は構造体のメンバをリストにしたもの, 2 つ目は構造体の型で宣言された変数をリストにしたものである. 構造体ノードは (Listing2) のように表す. 変数の場合はスコープ, 仮引数に関係なくすべての変数をリストにするものとする. 変数とメンバの情報には, 型の名称, 変数名またはメンバ名, N 次元配列それぞれの長さを持つ. 同じデータ構造で表現できるのでアルゴリズム上同一に扱う. アルゴリズム上, 基本データ型も構造体ノードに含める. その場合, メンバが存在しない構造体であるとして扱う. また, 外部から変数であるかメンバであるかが isMember によって判別できる.

Listing 1: 変数またはメンバを表すクラス

```
1 class Variable{
2     public int ID; // 比較用 ID
3     public string typeName; // 型の名称
4     public string name; // 変数またはメンバの名称
5     public List<int> nestArray; // 配列の長さのリスト
6 }
```

Listing 2: 構造体ノードを表すクラス

```
1 class StructNode{
2     public string typeName; // 型の名称
3     public List<Variable> nodes; // 変数またはメンバのリス
      スト
4     public bool isMember; // メンバのリストであるか変数の
      リストであるか,
5 }
```

3.3.2 コードモデル

コードモデルはソースコードから作成された構造体ノードをすべてリストにまとめたものである。図 3.3 のように構造体ノードを組み合わせたデータの集合として表される。

3.3.3 レイアウト変換命令

レイアウト変換命令はコードモデルに対して適応することで別のコードモデルを生成できる。また、コードモデルの状態から適用可能なレイアウト変換命令を導出することができる。

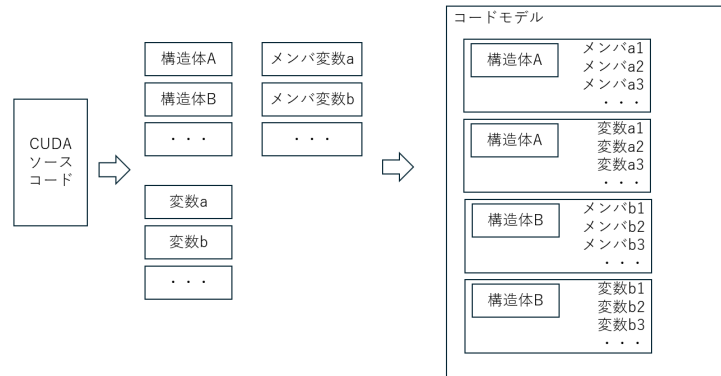


図 3.3: コードモデル

3.3.4 データレイアウト

データレイアウトはコードモデルで表されるソースコード中のデータ構造の状態を表す。コードモデル同士での配列の有無や位置が異なるならば、異なるデータレイアウトとして扱う。

3.3.5 親構造体, 子構造体

構造体 A のメンバに構造体 B がある場合を考える。この時、構造体 A は構造体 B の親構造体と表現し、構造体 B は構造体 A の子構造体と表現する。

3.3.6 コンフリクト

手法によってソースコードの書き換えが不可能であると判断された場合、コンフリクトが起こったと表現する。コンフリクトが起こったときの

コードモデルへの書き換えはできないと判断して, 候補から取り除く.

3.4 手法の流れ

まずは提案手法の流れを (図 3.4) に示す. CUDA ソースコードを入力とし, 最初に構文解析を行うパーサーによって構造体とメンバと変数の情報を抽出し, コードモデルを生成する. コードモデルから命令生成期によって複数のレイアウト変換命令を生成し, 変換命令に従って CUDA ソースコードを変換することで異なるデータレイアウトを持った等価なソースコードを複数生成する.

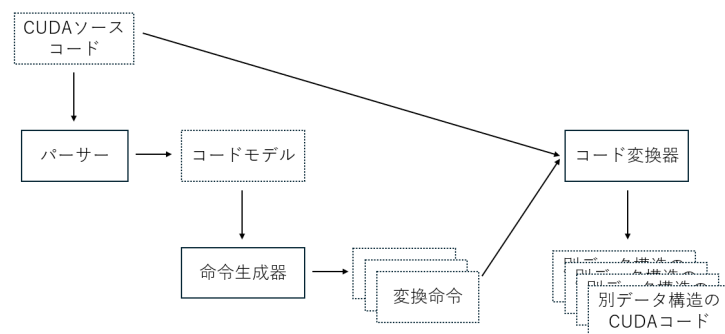


図 3.4: 提案手法の流れ

3.5 データレイアウトの変換

データレイアウトを変換するための操作を, レイアウト変換命令と呼ぶことにする. 表 3.1 のように, 6 つの命令を定義する.

表 3.1: レイアウト変換命令

1	ExpandMemberToVariable	構造体メンバの配列を 変数側に移動させる
2	ExpandMemberToMember	構造体メンバの配列を 子構造体のメンバに移動させる
3	ConvergenceVariableToMember	構造体変数の配列を メンバに移動させる
4	ConvergenceMemberToMember	構造体メンバの配列を 親構造体のメンバに移動させる
5	TransposeVariable	構造体メンバの二次元以上の 配列を転置する
6	TransposeMember	変数の二次元以上の 配列を転置する

3.5.1 レイアウト変換命令の定義

レイアウト変換命令の情報には6パターンある変換命令と, 対象となる構造体と, メンバまたは変数の集合が含まれる. Listing3 にレイアウト変換命令のデータ構造を表す.

Listing 3: レイアウト変換命令を表すクラス

```
1 enum OrderType{
2     ExpandMemberToVariable = 1,
3     ExpandMemberToMember = 2,
4     ConvergenceVariableToMember = 3,
5     ConvergenceMemberToMember = 4,
6     TransposeVariable = 5,
7     TransposeMember = 6
8 }
9 class Order{
10     public OrderType order; // 型の名称
11     public string from; // 対象の構造体名
12     public string to; // 対象の構造体名
13     List<Variable> fromElements; // 対象の変数またはメンバ
14     List<Variable> toElements; // 対象の変数またはメンバ
15     int N; // 静的配列の長さ
16     int M; // 転置する静的配列の長さ 変換命令5,6 で利用す
        る.
17     int index; // 転置する静的配列の位置 変換命令5,6 で利
        用する.
18 }
```

次に、それぞれの命令について説明する。図には、コードモデルに対してレイアウト変換命令を適用した場合の適用前と適用後が示されている。また、ソースコードに適用した場合の変化も示してある。

1. ExpandMemberToVariable この操作における対象となる構造体は1つである。それを構造体 A とする。ExpandMemberToVariable は構造体 A のメンバの配列を、構造体 A の変数に移動させることを意味する。これは AoS から SoA への変換と同義である。図 3.5 に例を示す。この操作を行える条件は、対象となる構造体 A のメンバそれぞれが

同じ長さの配列を持っている場合に適用可能である。

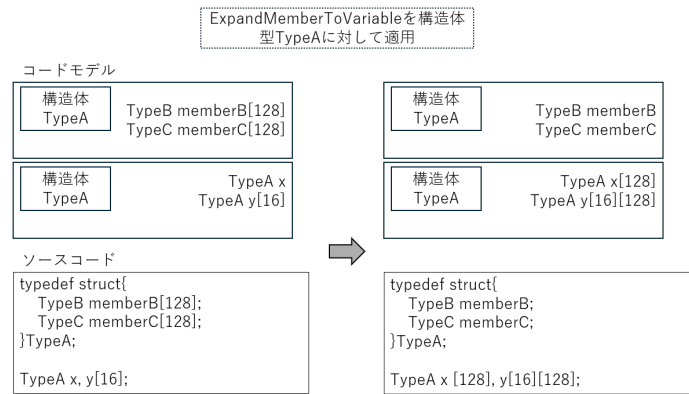


図 3.5: ExpandMemberToVariable

- ExpandMemberToMember この操作における対象となる構造体は2つである。それらを構造体 A,B とする。ExpandMemberToMember は構造体 A のメンバの配列を構造体 B のメンバに移動させることを意味する。図 3.6 に例を示す。この操作を行える条件は、構造体 A は構造体 B の親構造体であることと、構造体 A に含まれる構造体 B のメンバそれぞれが同じ長さの配列を持っている場合に適用可能である。
- ConvergenceVariableToMember この操作における対象となる構造体は1つである。それを構造体 A とする。ConvergenceVariableToMember は構造体 A の変数の配列を構造体 A メンバに移動させることを意味する。これは SoA から AoS への変換と同義である。図 3.7 に例を

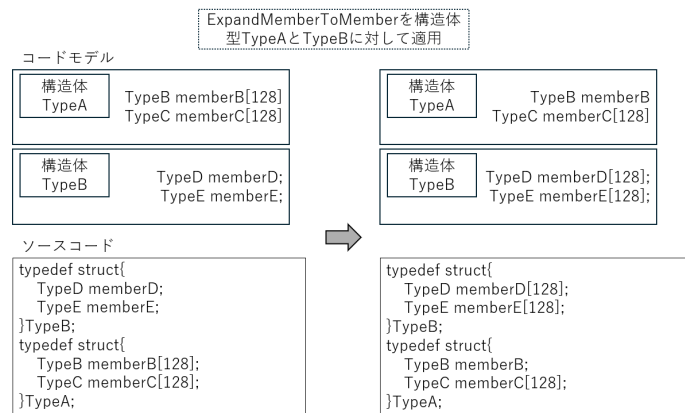


図 3.6: ExpandMemberToMember

示す. この操作を行える条件は, 構造体 A のメンバ 1 つ以上が同じ
長さの配列を持っている場合に適用可能である.

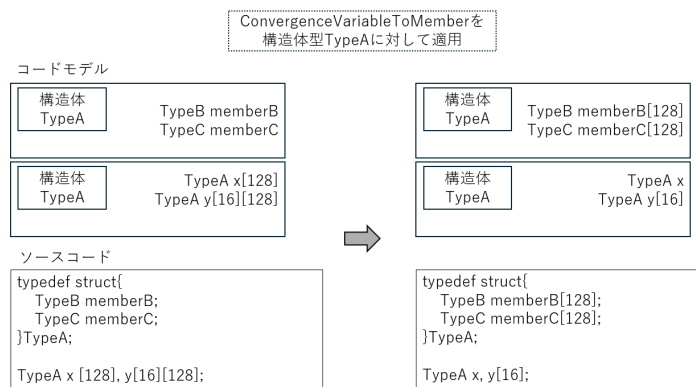


図 3.7: ConvergenceVariableToMember

4. ConvergenceMemberToMember この操作における対象となる構造体
は 2 つである. それらを構造体 A,B とする.ExpandMemberToMem-

ber は構造体 A のメンバの配列を構造体 B のメンバに移動させることを意味する. 図 3.8 に例を示す. この操作を行える条件は, 構造体 B は構造体 A の親構造体であることと, 構造体 A のメンバそれぞれが同じ長さの配列を持っている場合に適用可能である.

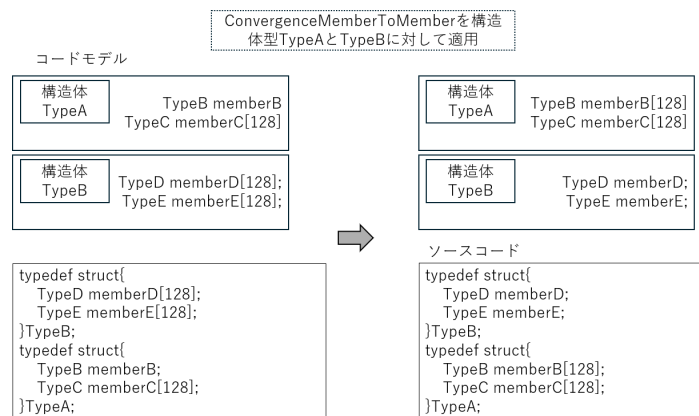


図 3.8: ConvergenceMemberToMember

5. TransposeVariable この操作における対象となる構造体は 1 つである. それを構造体 A とする. TransposeVariable は構造体 A に含まれる 2 次元以上の変数のインデックス位置を交換することを意味する. 図 3.9 に例を示す. この操作における条件は 2 次元以上の配列である限り無い.
6. TransposeMember この操作における対象となる構造体は 1 つである. それを構造体 A とする. TransposeVariable は構造体 A に含まれる

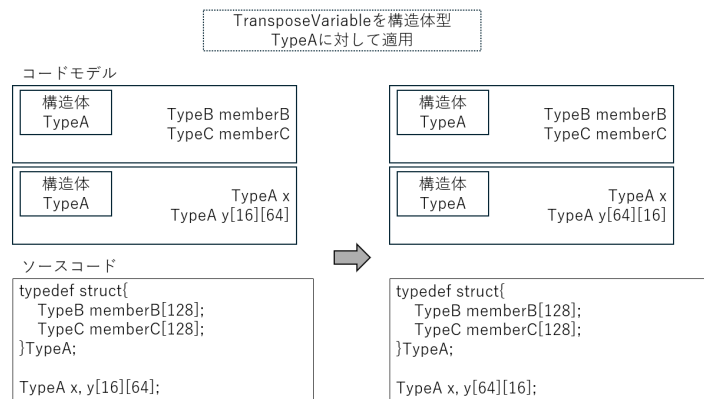


図 3.9: TransposeVariable

る 2 次元以上のメンバのインデックス位置を交換することを意味する。図 3.10 に例を示す。この操作における条件は 2 次元以上の配列である限り無い。

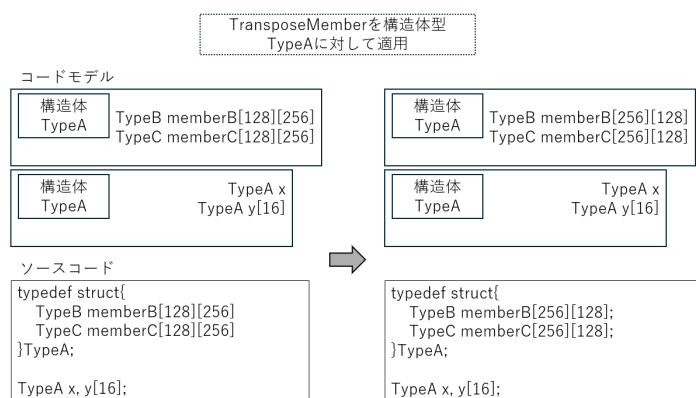


図 3.10: TransposeMember

これらの操作を組み合わせたことによって複数の等価なデータ構造を生成することができる。

3.6 コードモデルの生成

次にコードモデルからレイアウト変換命令を求める方法を疑似コードで説明する。まず、構造体命令の集合それぞれから命令の候補を導出する。構造体ノードに含まれるメンバまたは変数リストのすべてが長さが N の配列を持っている場合、`GetRepresentative()` 関数で N を求められる。構造体ノードに含まれるメンバまたは変数リストの一部が配列を持っている場合、`GetCandidates()` で重複のない配列の長さのイテレータを求められる。 `GetPair` 関数では、1つの構造体について2つの構造体ノードがあるため、そのもう片方を検索する関数である。 `bool` 変数 `isMember` によってメンバのリストか変数のリストかを判別できる。なお、文法はC#に従っており、反復子によってレイアウト変換命令の候補を列挙している。これらを疑似コードで表したものを (Listing 4) に示す。実際のコード上ではレイアウト変換命令を表すクラスを返していて、対象の構造体や変更する配列の長さなどの情報を付与しているが、疑似コード上では省略していてレイアウト変換命令の名称のみ記述してある。

Listing 4: コードモデルからレイアウト変換命令の生成

```
1 IEnumerable<Order> G(Codemodel m){ // 変換命令のイテレー  
    タを返す  
2  
3 foreach(StructNode node in m){ // コードモデルから構造体  
    ノードをそれぞれ処理する m  
4 if(node.isMember){  
5     int length;  
6     if ((length = node.GetRepresentative()) != 0) // メン  
        バがすべて長さlength の配列を持っている  
7     {  
8         if (GetPair(node).nodes.Count > 0) // 変数が個以  
            上宣言されている 0  
9         {  
10            yield return ExpandMemberToVariable;  
11        }  
12        yield return TransformOrder.  
            ConbergenceMemberToMember;  
13    }  
14    foreach (int n in node.GetCandidates())  
15    {  
16        yield return ExpandMemberToMember; // メンバの一部  
            に長さn の配列を持っている  
17    }  
18 }else{  
19     int length;  
20     if ((length = node.GetRepresentative()) != 0) // 変数  
        がすべて長さlength の配列を持っている  
21     {  
22        yield return ConvergenceVariableToMember;  
23     }else{  
24        foreach (int n in node.GetCandidates()) // 変数の  
            一部に長さn の配列を持っている  
25        {  
26            yield return ConvergenceVariableToMember;  
27        }  
28    }  
29 }  
30 foreach (var t in node.GetTranspose()) // 変数またはメン  
    バの一部が転置できる  
31 {  
32     if(node.isMember){  
33        yield return TransposeMember;
```

```
34     }else
35     {
36         yield return TransposeVariable;
37     }
38 }
39 }
```

探索で生成した命令をコードモデルに適応し新たなデータレイアウトを持ったコードモデル生成する関数 $F(m,o)$ について, 疑似コードで説明する. FindFromNode と FindToNode は命令から対象の構造体を検索する. TransposeIndexer 関数では対象の変数またはメンバの N,M の配列インデックス位置を交換する. PullIndexder 関数では対象の変数またはメンバから右端に長さ N の配列インデックスを追加する. PushIndexder 関数では対象の変数またはメンバから右端の配列インデックスを削除する. これらを疑似コードで表したものを (Listing 5) に示す.

Listing 5: コードモデルの変更

```
1 public void F(Order t)
2 {
3     StructNode from = FindFromNode(t); // 対象の構造体を検
        索する
4     StructNode to = FindToNode(t); // 対象の構造体を検索
        する
5
6     if (t==TransposeVariable||t==TransposeMember)
7     {
8         from.TransposeIndexer(t.fromVariables, t.N, t.M, t
            .index); // 配列[N][M] を入れ替
                える
9     }
10    else
11    {
12        from.PullIndexer(t.fromVariables, t.N) ;// 配列
            [N] を削除する
13        to.PushIndexer(t.toVariables, t.N) ;// 配列
            [N] を追加する
14    }
15 }
```

これらの操作によって, コードモデル上での異なるデータレイアウトを
列挙することができる.

3.7 データレイアウトの探索

列挙したコードモデルには, すでに発見したものもあるため, 探索アル
ゴリズムを利用する.

ここで, ソースコードの状態 M_n , レイアウト変換命令 O_n , 命令生成器
 $G(m)$, 命令適用関数 $F(m,o)$ とする. なお, M_n はソースコードの状態を表
し, M_0 が初期状態とする. 命令生成器 $G(m)$ と命令適用関数 $F(m,o)$ のア

ルゴリズムは実装方法の章で説明する. オープンキュー, クローズリスト
を利用して幅優先探索を探索を行うとアルゴリズムを (Listing 6) に疑似
コードとして示す.

Listing 6: 探索アルゴリズム

```
1 Queue<CodeModel> open=new Queue<CodeModel>();
2 List<CodeModel> close=new List<CodeModel>();
3 open.Enqueue(M0); // 初期状態をキューに追加 M0
4
5 while(open.Count>0){ // キューがなくなるまで続ける 0
6
7     CodeModel target=open.Dequeue();
8     if(close.Contains(target))continue; // に同じものが
        入っているなら close もどる,
9     close.Add(target); // に入れる close
10
11     foreach(Order o in G(target)){ // 命令を生成する
12         CodeModel newPattarn=F(target,o); // 生成した命令
            を適用する
13         open.Enqueue(newPattarn); // 変更後のコードモデル
            をキューに追加
14     }
15 }
```

3.8 コード変換

データ構造のレイアウト変換命令を導出した後は、ソースコードをレイアウト変換命令に従って変更する必要がある為、ここではその方法について議論する。探索した際に使用した変換命令のリストを保持しておくことで、それを使ってソースコードの変換を行う。変更がある構造体について、その変数を列挙し、それらの変数についてメンバや配列のアクセスを書き換えることを主にする。変換一度につき、メンバや配列のアクセス構文が入れ替わる。(図 3.11) に例を示す。

このようにアクセス構文の入れ替えによってコードを書き換えること

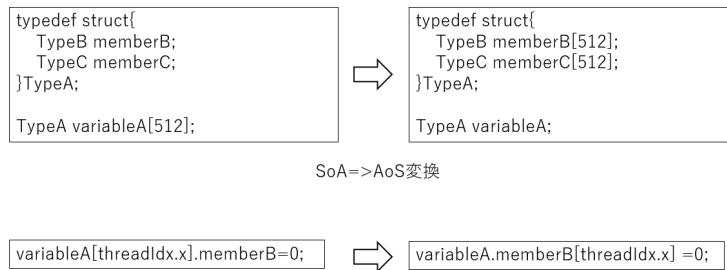


図 3.11: 入れ替えの例

ができる。また、構造体の定義部分は `[]` を追加または削除することで書き換えを行う。

3.9 構造体のコピー

ただ、問題となるのがコピーが生成されたときの場合である。このとき、オリジナルの構造体はそのままにしておいてコピーした構造体は名称を変更する。ここでは名称の前に `_auto_` を付けるものとする。さらにコピー前の構造体とコピー後の構造体での相互変換コードが必要になる。データ構造の変換によってメモリ上のデータレイアウトが変更されるため、任意のデータ構造に再配置するようなコードが要求される。

SoA から AoS 変換で構造体のコピーが発生したときの例を (図 3.12) に示す。この例では `TypeA` の変数の配列が `TypeA` のメンバに移動している。そして `TypeA` と `_auto_TypeA` の相互変換コードは (図 3.13) である。デー

タレイアウトの変更に対応するために複数行使ってデータのディープコピーを行うか, 内に記述して構造体を再構成している.

- $B = \{A.member_1[i], A.member_2[i], A.member_3[i], \dots\};$

構造体要素を $\{ \}$ 内に記述する文法は宣言時にしか書けないため, それ以外の場合は以下のように書く.

- $B.member_1[i] = A.member_1;$
- $B.member_2[i] = A.member_2;$
- $B.member_3[i] = A.member_3; \dots$

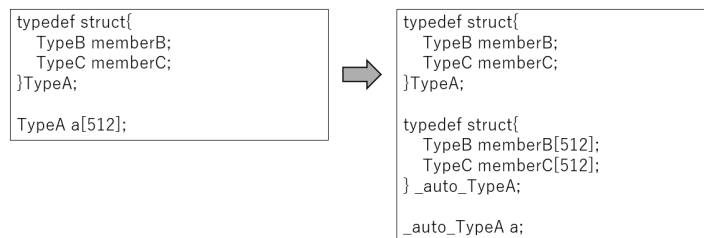


図 3.12: SoA から AoS 変換の例

3.10 問題点と対応方法

これらの操作によってソースコードにコンフリクトが生じる場合が2つある.1つ目は親, 子構造体への影響によるもの,2つ目はデータ構造に

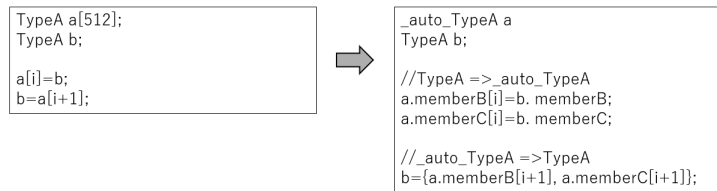


図 3.13: SoA から AoS 変換の相互変換コード

依存したアルゴリズムである。1つ目が起こるパターンは子に変更されることによって親のデータ構造が変化する場合である。この問題の対処法について、変更後の構造体を新しく作るということが考えられる。これは、もともとある構造体に影響がないという利点があるなお、構造体のコピーが発生する場合、後に説明する構造体の相互変換コードが必要になる。そのため、相互変換のコストや手法が複雑になることが考えられるためこの提案手法では1つの構造体につきコピーは1つであるという制約を設けることにする。Listing7, Listing8 には、構造体のコピーが発生したときのソースコードの変化を示してある。これは実験で利用したソースコードをもとにしている。もともと変数 `boidsDevice` にあった配列が、データレイアウトの変更によって `Vector3` 構造体に移動したときの例である。引数に `Vector2` が含まれる `getDistance` 関数は `_auto_Vector2` 構造体を新しく作成したことによって変更する必要がなくなる。そして呼び出し側で

は `_auto_Vector2` 構造体を `Vector2` 構造体に変換する必要がある. 同様に `Vector2` 構造体を `_auto_Vector2` 構造体に変換するコードも必要になる場合がある. 構造体要素を内に記述する

2つ目が起こる場合は, 二次元配列を一次元の配列の配列として配列をむき出して利用する場合などがある. ヒューリスティックに解決できる場合もあるが, 本手法ではその対応ができていないため, このパターンが出現した場合自動変換ができないと判断することにする. 次はどのような場合に変換できないパターンや対処方法を見ていく.

1. `ExpandMemberToVariable` 対象の構造体で宣言されている変数がすべて同じ長さの配列を持っている場合であれば問題ないが, 一部の変数のみ配列を持っている場合に問題が起こる. その一部の変数との SoA 変換を行うと残りの変数にコンフリクトが生じる. 例えば `A.member[i]` のようなコードであれば問題ないが `A.member` までのアクセスによって配列を取り出そうとした場合にコンフリクトが生じる.
2. `ExpandMemberToMember` 対象の構造体 A,B について,A のメンバ配列を B に移動させる場合 B の構造体の内容が変化してしまうため, 構造体 B の変数との間にコンフリクトが生じる. その場合は構造

体 B を複製して対処する. A.member までのアクセスによって構造体 B に依存するコードの場合にコンフリクトが生じる.

3. ConvergenceVariableToMember 対象の構造体 A について, A[i].member までのアクセスによって構造体 B に依存するコードの場合にコンフリクトが生じる.
4. ConvergenceMemberToMember 対象の構造体 A, B について, B のメンバ配列を A に移動させる場合 A のメンバである B の構造体に直接アクセスする場合にコンフリクトが生じる.
5. TransposeVariable 変形を行う変数の一次元のみの部分にアクセスするとコンフリクトが生じる.
6. TransposeMember 変形を行うメンバの一次元のみの部分にアクセスするとコンフリクトが生じる.

Listing 7: 構造体のコピー前

```
1 typedef struct {
2     float x, y;
3 }Vector2;
4
5 typedef struct {
6     Vector2 pos;
7     Vector2 vel;
8     int id;
9 }Boid;
10
11 // 引数にVector2 が含まれる関数
12 __device__ float getDistance(Vector2 b1, Vector2 b2) {
13     float dx = b1.x - b2.x;
14     float dy = b1.y - b2.y;
15     return (float)(sqrt(dx * dx + dy * dy));
16 }
17 __device__ Boid boidsDevice[1024];
18
19 void main(){
20     // getDistance の呼び出し
21     getDistance(boidsDevice[i].pos, boidsDevice[j].pos);
22
23     // Boid のpos メンバへの代入
24     Vector2 temp;
25     boidDevice[i].pos=temp;
26 }
```

Listing 8: 構造体のコピー後

```
1 // 新しく生成された構造体
2 typedef struct {
3     float x[1024], y[1024];
4 }_auto_Vector2;
5
6 typedef struct {
7     float x, y;
8 }Vector2;
9
10 typedef struct {
11     _auto_Vector2 pos;
12     _auto_Vector2 vel;
13     int id;
14 }Boid;
15
16 // 引数にVector2 が含まれる関数
17 __device__ float getDistance(Vector2 b1, Vector2 b2) {
18     float dx = b1.x - b2.x;
19     float dy = b1.y - b2.y;
20     return (float)(sqrt(dx * dx + dy * dy));
21 }
22 __device__ Boid boidsDevice;
23
24 int main(){
25     // getDistance の呼び出し
26     getDistance({boidsDevice.pos.x[i], boidsDevice.pos.y[i]
27                 }, {boidsDevice.pos.x[j], boidsDevice.pos.y[j]
28                 });
29
30     // Boid のpos メンバへの代入
31     Vector2 temp;
32     boidDevice.pos.x[i]=temp.x;
33     boidDevice.pos.y[i]=temp.y;
34 }
```

4 性能評価

ネストされたデータ構造を含む CUDA ソースコードを対象に手法によって異なるデータ構造レイアウトを作成し,それぞれ実行時間を計測した. CPU は Intel Core i7-6700,GPU は NVIDIA GeForce RTX 3060 を利用して実験を行った. 対象のソースコードは,Boid モデル,粒子シミュレーション,交通シミュレーションの3つを使用した. また,パターン0はオリジナルのソースコードである.

(図 4.14) には Boid モデルでのそれぞれのパターンでの実行時間を示している. Boid モデルとは,鳥の群れをシミュレートするためのアルゴリズムである. 衝突回避, 整列, 接近の三つのルールによって行われ, 各個体の計算をそれぞれ並列化することが可能である. 提案手法によって2種類のデータ構造レイアウトを生成することができ, そのうちパターン1はオリジナルのものよりも約 1.60 倍高速であった. 生成したデータが少ないパターンであったが, パターンごとにそれぞれ実行時間が異なることを示せた.

(図 4.15) には交通シミュレーションでのそれぞれのパターンでの実行時間を示している. 盤目上の道路で車がランダムに進行するアルゴリズムを実装した. 交差点上では信号がそれぞれ一定周期で切り替わり, 単位

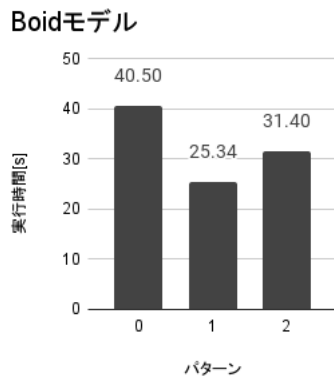


図 4.14: Boid モデルの実行時間グラフ

時間ごとに次状態を計算するアルゴリズムである。提案手法によって 11 種類のパターンを生成することができ、そのうちのパターン 6 はオリジナルのものよりも約 1.42 倍高速であった。この実験では、メインループの部分でアクセスパターンによって実行時間が高速、低速の 2 つに分かれた。

(図 4.16) には粒子シミュレーションでのそれぞれのパターンでの実行時間を示している。空間をいくつかのセルに分け、そのセル内にある粒子のリストを作成することで近傍粒子の候補を減らすセルリンクリスト (CLL) を用いての粒子シミュレーションアルゴリズムを実装した。提案手法によって 8 種類のパターンを生成することができ、そのうちのパターン 8 はオリジナルのものよりも約 1.21 倍高速であった。

交通シミュレーション

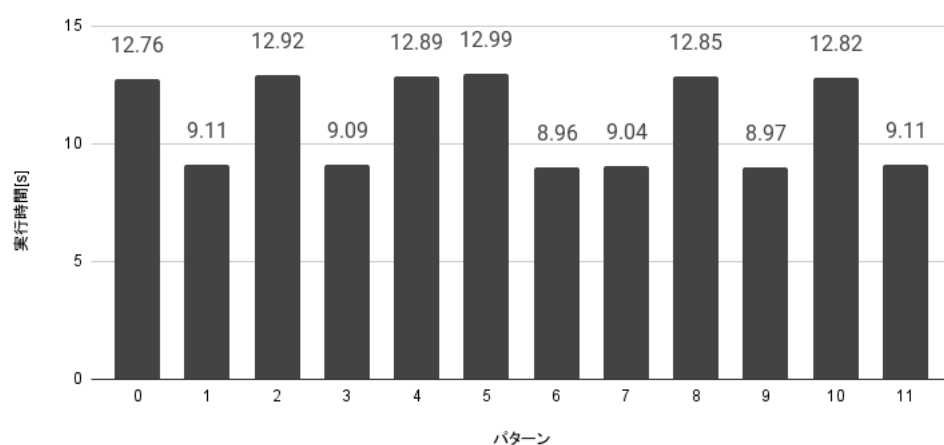


図 4.15: 交通シミュレーションの実行時間グラフ

粒子シミュレーション

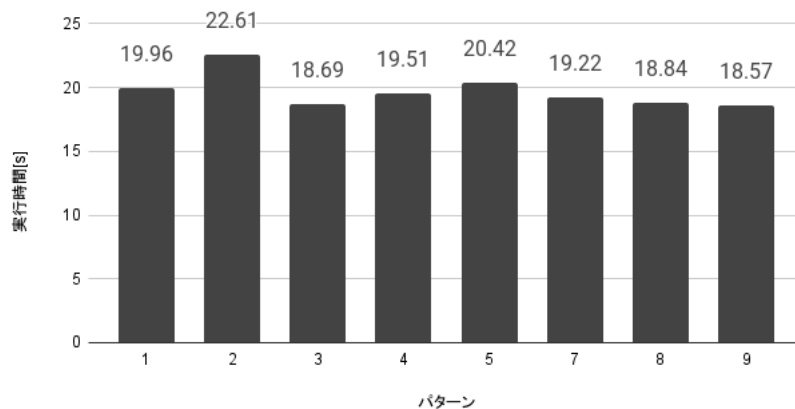


図 4.16: 粒子シミュレーションの実行時間グラフ

5 関連研究

従来, アクセスパターンによるソースコードの最適化のために AoS, SoA の相互変換に関する研究が行われてきた. AoS, SoA 変換により, メモリ上のデータレイアウトが変更されるため GPU グローバルメモリに対してのアクセスパターンが変化する. よって, アクセス効率に影響が生じ, AoS, SoA それぞれで実行時間が変化する.

[1] の研究では, 構造体に対してテンプレートパラメータを与えることで AoS, SoA 変換を自動的に行うことができる, 抽象化レイヤーを実装している.

[2] の研究では, AoS 構造体をいくつかのクラスタに分割し, 決定木を用いてクラスタごとの優れたデータレイアウトを選択する手法を提案している. AoS, SoA だけではなく, Array of Struct of Array(AoSoA) と言われる中間レイアウトにも対応していることが特徴である. このように SOA, AOS を相互変換する試みはすでに行われている.

[5] の研究では, CUDA アプリケーションの配列メモリアクセスを抽象化しハードウェアに合わせて最適なデータ構造レイアウトを決定できる. これは AoS, SoA, AoSoA, 二次元配列の転置に対応している.

しかし, これらの研究では構造体のメンバに構造体が入っている, ネス

トされた構造体に対応できていないという問題がある．実際のソースコードには, ネストされた構造体が含まれていることは十分にあり得るため, 本研究ではそれに対応した．

6 おわりに

本研究では提案手法によってデータ構造パターンを複数生成し、ソースコードに反映させ、それぞれ実行時間を計測した。その結果、オリジナルのソースコードよりも実行時間の短いデータ構造を見つけることができた。また、従来手法では対応できないデータ構造にネスト構造を含むソースコードへの対応が可能となった。

配列の移動を行うに際して、配列の一部を移動させるタイリングによって生まれるパターンにより最適なものが含まれる可能性もある.[9] のようにタイリングによって高速化を達成している研究や,[5] のようにタイリングを含めたデータ構造パターンの最適化を行う研究がある。そのため、データ構造にタイリングについては今後の課題である。現在の手法に加えてタイリングのパターンも考慮することでより多くのデータ構造パターンを生成し、さらにメモリアクセスパターンを改善したソースコードを得られる可能性がある。今後は、生成したデータ構造パターンに対して実行時間を推定する評価関数を実装することで、ソースコードを実行することなく性能を評価し最適だと思われるデータ構造を決定することが将来的に可能であると考えられる。

謝辞

本研究を行うにあたり，多数のご指導をいただきました大野和彦講師，並びに，多数の助言を頂きました高木一義教授，深澤祐樹技術員に深く感謝いたします。また，お世話になりましたコンピュータアーキテクチャ研究室の皆様に深く感謝いたします。

参考文献

- [1] PStrzodka, Robert. (2012). Abstraction for AoS and SoA layout in C++.10.1016/B978-0-12-385963-1.00031-9.
- [2] Kofler, K., Cosenza, B., Fahringer, T. (2015).Automatic Data Layout Optimizations for GPUs. In: Tr ¨aff, J., Hunold, S., Versaci, F. (eds) Euro-Par 2015: Parallel Processing. Euro-Par 2015. Lecture Notes in Computer Science(), vol 9233. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-48096-0_21
- [3] FM. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “ A compiler framework for optimization of affine loop nests for GPGPUs,” ACM International Conference on Supercomputing (ICS), 2008
- [4] Nvidia developer CUDA Toolkit. <https://developer.nvidia.com/cuda-downloads>
- [5] Nicolas Weber and Michael Goesele. 2017. MATOG: array layout auto-tuning for CUDA. ACM Transactions on Architecture and Code Optimization (TACO) 14, 3 (2017)

- [6] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. *ACM Comput. Surv.* 55, 11, Article 239 (November 2023), 81 pages. <https://doi.org/10.1145/3570638>
- [7] Y. Hu, H. Liu, and H. H. Huang. 2018. TriCore: Parallel triangle counting on GPUs. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018).
- [8] T. Dong, A. Haidar, P. Luszczek, et al. 2014. LU factorization of small matrices: Accelerating batched DGETRF on the GPU. In *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst* (2014).
- [9] R. Nath, S. Tomov, T. “Tim” Dong, et al. 2011. Optimizing symmetric dense matrix-vector multiplication on GPUs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

- [10] N. Delbosc, J. L. Summers, A. I. Khan, et al. 2014. Optimized implementation of the lattice boltzmann method on a graphics processing unit towards real-time fluid simulation. *Computers Mathematics with Applications* 67, 2 (2014)