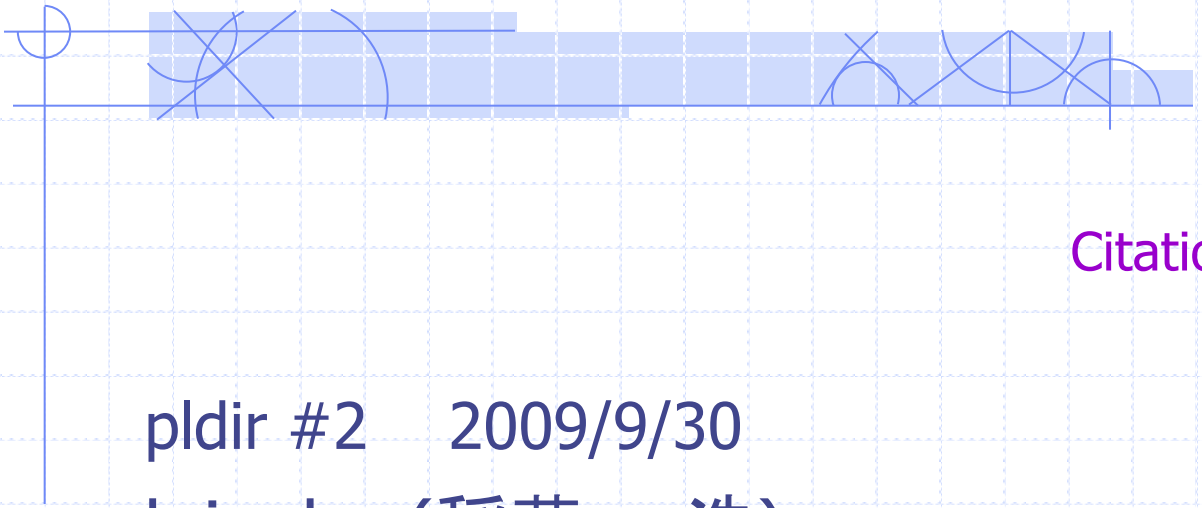
- This slide was
  - a material for the "Reading PLDI Papers (PLDIr)" study group
  - written by Kazuhiro Inaba (www.kmonos.net), under my own understanding of the papers published at PLDI
    - So, it may include many mistakes etc
- For your correct understanding, please consult the original paper and/or the authors' presentation slide!

Reading

"The Implementation and Evaluation of Fusion and Contraction in Array Languages"

[E. C. Lewis, C. Lin, and L. Snyder]



Citation Count (ACM) 17

pldir #2 2009/9/30

k.inaba (稲葉 一浩)

http://www.kmonos.net/

Background

- 配列演算できる言語がある
  - Fortran 90
  - High Performance Fortran
  - ZPL
- 例

```
A(1:n, 1:m) =
    B(1:n, 1:m) + C(1:n, 2:m+1) * D(0:n-1,1:m)

// for( i=1; i≦n; ++i )
//   for( j=1; j≦m; ++j )
//     A[i][j] = B[i][j] + C[i][j+1] * D[i-1][j];
```

Background

- ナイーブに実装すると効率が悪い
- 既存のコンパイラは
  - ナイーブに実装して
  - 配列演算をスカラー化（ループに直す）した後
  - 普通のスカラーコンパイラのループ最適化
- という方式で、効率を取り戻そうとしている

```
A(1:n, 1:m) =
    B(1:n, 1:m) + C(1:n, 2:m+1) * D(0:n-1,1:m)
// for( i=1; i≦n; ++i )
//   for( j=1; j≦m; ++j )
//     A[i][j] = B[i][j] + C[i][j+1] * D[i-1][j];
```

単純な式に直す

```
tmp(1:n,1:m) = C(1:n, 2:m+1) * D(0:n-1,1:m)
A(1:n,1:m) = B(1:n, 1:m) + tmp(1:n, 1:m)
```

スカラー化

```
for( i=1; i≦n; ++i )
  for( j=1; j≦n; ++j )
    tmp[i][j] = C[i][j+1] * D[i-1][j];
for( i=1; i≦n; ++i )
  for( j=1; j≦n; ++j )
    A[i][j] = B[i][j] + tmp[i][j];
```

ループ融合

```
for( i=1; i≦n; ++i )
  for( j=1; j≦n; ++j )
    tmp[i][j] = C[i][j+1] * D[i-1][j];
    A[i][j] = B[i][j] + tmp[i][j];
```

Background

- その他の、最適化したい例
  - 依存のない、同時に回せるループ

```
B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)
```

  - プログラマが入れたテンポラリ

```
B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)
C(1:n,1:m) = B(1:n,1:m)
```

  - 再代入

```
A(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)
A(1:n,1:m) = A(0:n-1,1:m)+A(0:n-1,1:m)
```

この研究の内容

- このような最適化
  - Fusion：一緒に回せるループは一緒に回す
  - Contraction：不要なテンポラリを消す
- を、スカラレベルの
  ループ最適化に任せるのではなく、
  配列レベルでおこなうべき / おこなった
  - というお話
  - Typically 20%～ Max 400% の高速化

ながれ

- ZPL
- Normalized Array Statement
  - Fusion 最適化しやすい形式の配列演算文
- Array Statement Dependence Graph
  - 配列演算文どうしの依存関係をグラフにしたもの
- Fusion Partition
  - 同じループで回して問題ない配列演算をまとめる
- Scalarization
  - 適切なループ方向や添え字順を選ぶ

Normalized Array Statement

- この論文で最適化する対象言語

```
// こんな形式の文の列のみ扱える
[1..n,1..m] A@(0,0) := B@(0,-1)
[1..n,1..m] A@(0,0) := B@(1,2) + C@(-1,0)
```

  - “[添字の動く範囲] 演算文”
    - 配列に @(整数, 整数) の形式でオフセット指定
      - 書けない: for(i,j) A[i][j] = B[j][i]
      - 書けない: for(i)　A[i] = B[2*i]
      - 書けない: for(i)　A[i] = B[n-i]
    - 一つの文で入出力両方にあらわれる配列はない
      - 書けない:　for(i)　A[i] = A[i+1]

テンポラリ変数を入れれば、この形は消せる。あとで最適化でテンポラリも消す

ASDG
(Array Statement Dependence Graph)

- 配列演算文どうしの依存関係
  - Flow Dependence

```
[1..n] A@(0) := B@(0) // こっちが先！
[1..n] C@(0) := A@(0)
```

  - Anti Dependence

```
[1..n] B@(0) := A@(0)
[1..n] A@(0) := C@(0) // こっちが後！
```

  - Output Dependence

```
[1..n] A@(0) := B@(0)
[1..n] A@(0) := C@(0) // こっちが後！
```

ASDG
(Array Statement Depende

これが
ASDGの持つ
依存関係情報

- Flow/Anti/Output 依存があっても
  まだ同じループに入らないとは限らない
- Iteration Space 間の依存関係

```
[1..n,1..m] A@(0,0) := B@(0,0)
[1..n,1..m] C@(0,0) := A@(1,0)

for(i=n; i≧1; --i)    // こっち向きで回れば行ける
  for(j=1; j≦m; ++j)
     A[i][j] = B[i][j]
     C[i][j] = A[i+1][j]
```

[A, (-1,0), flow]

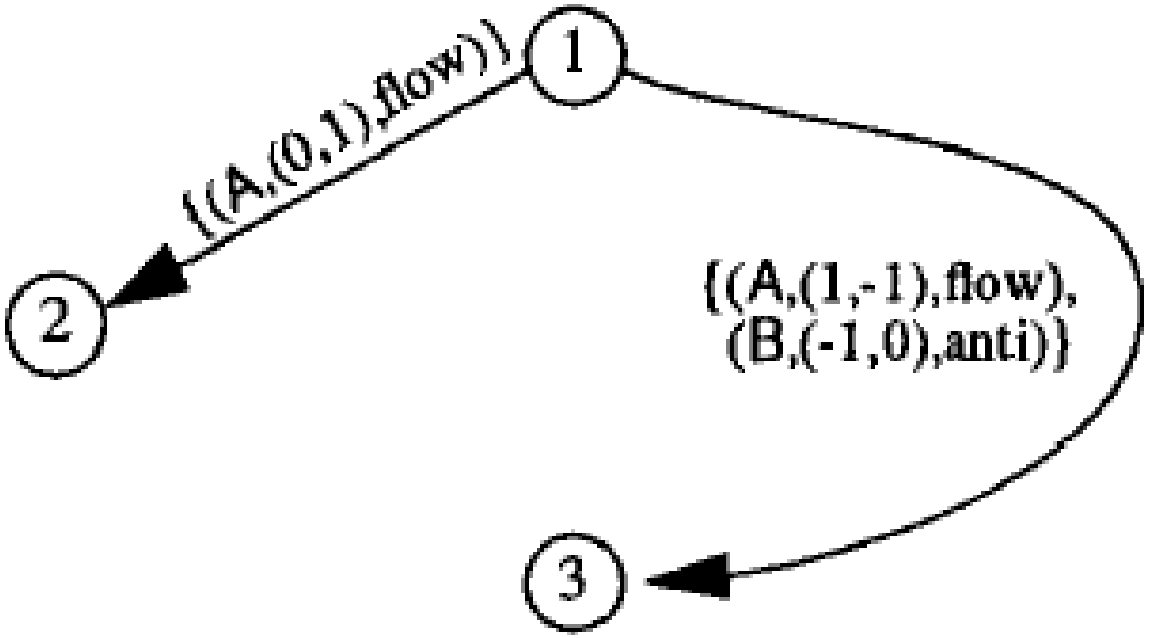
ASDG
(Array Statement Dependence Graph)

- ノード = 配列演算文1個
- エッジ = 依存関係情報
  - [依存原因の変数,
    ループ添字オフセットを引き算したベクトル,
    flow|anti|output]
- からなるグラフ

ASDG
(Array Statement Dependence Graph)

- 例

```
1   [1..m,1..n] A := B@(-1,0);
2   [1..m,1..n] C := A@(0,-1);
3   [1..m,1..n] B := A@(-1,1);
```



Fusion Partition

- いっしょに回せる配列演算文を
  できるだけクラスタリング
  - Fusion for Contraction
    - 消せるテンポラリ変数は消すためのFusion
  - Fusion for Locality
    - 同じ配列に対するループをできるだけまとめて
      メモリ局所性を得るためのFusion
  - Fusion for 適当にGreedyにできそうならやってみる

stering

- に回せる配列演算文を
  できるだけクラスタリング

```
FUSION-FOR CONTRACTION(G)
1   P ← trivial partition of G
2   l ← |V|
3   x ← array vars in G sorted by decreasing weight
4   for i ← 1 to |x|   { consider var x_i for contraction }
5       c ← {P_j | P_j contains a reference to variable x_i}
6       c ← c ∪ GROW(c, G)
7       if CONTRACTIBLE?(x_i, c, G) ∧
               FUSION-PARTITION?(c, G)
8           k ← smallest j for P_j ∈ c
9           P_k ← ∪_{x∈c} x
10          l ← l − (|c| − 1)
11  return P
```

1: とりあえず
全部
別のクラスタ

2: (大雑把に
言うと)出現回数
の多い変数を
先に考える

3: 変数xiを
含む
クラスタ
全部

4: それに
挟まれてる
クラスタも
全部入れる

5: ひとつに
潰せそうなら

6: ひとつの
クラスタへと
Fusion!

ste

- に回せる配列演
  できるだけクラスタリング

```
FUSION-P      CONTRACTION(G)
1    P ←      vial partition of G
2    l ← |
3    x ← a    y vars in G sorted b    ecreasing weight w
4    for i    - 1 to |x|   { consi    var x_i for contraction }
5         c ←  {P_j | P_j contains a    eference to variable x_i }
6         c ←  c ∪ GROW(c, G)
7         if   CONTRACTIBLE?(x_i, c, G) and
                   FUSION-PARTITION?(c, G)
8              k ← smallest j for P_j ∈ c
9              P_k ← ∪_{x∈c} x
10             l ← l − (|c| − 1)
11   return P
```

「ひとつに潰せそうなら」

- 同じループ範囲を回っている
- Contract対象の変数のdep vecが(0,..,0)
- …
- 等々

- 詳しくは論文にて熟知すべし

スカラー化（ループに落とす）

- 考えなきゃいけないポイント
  - N次元配列のどの次元から順にループするか
    - for(i) for(k) ..., or for(k)for(i) ... ?
  - それぞれ、昇順ループにするか降順ループにするか
- Dependence Vectorを見ればわかる

全ての制約で
「昇順がいいか降順がいいか」が
揃ってる次元から順にループ

```
FIND-LOOP-STRUCTURE(C)
1  for j ← 1 to n                    { initialize unassigned mask }
2      b_j ← true                    { b_j = true ⇒ array dimension j has
                                     not yet been assigned to a loop}
3  for i ← 1 to n                    { iterate over loops }
4      for j ← 1 to n                    { iterate over array dimensions }
5          if b_j
                          ⎧ +1   if ∀u ∈ C, u_j ≥ 0
               d ← ⎨ −1   if ∀u ∈ C, u_j ≤ 0 and ∃u ∈ C, u_j < 0
                          ⎩ 0    otherwise
               if d ≠ 0    { can loop i iterate over dimension j? }
                   b_j ← false
                   p_i ← jd
                   C ← C − {u ∈ C | u_j ≠ 0}
                   break out of j loop
12     return NOSOLUTION     { no dimension found for loop i }
13 return p
```

もう考え
なくていい
制約

評価・まとめ

- 論文の図など見ながら

(軽く読んだ方のメモ)

“Type-Based Alias Analysis” 1/5

[A. Diwan, K.S. McKinley, and J.E.B. Moss]　Citation Count (ACM) 35

- エイリアス解析したい
  - エイリアス解析＝「二つのメモリ参照式が同じメモリを指してる可能性があるやなしや？」
- この論文の主張：既存の方法は
  - マジメにやりすぎてて重い。使えない
  - 評価方法が「どれだけaliasの可能性を正しく判定できたかの割合」なのが気にくわない
    - エイリアス解析結果によって「得られる最適化の機会の割合」でも評価すべき

(軽く読んだ方のメモ)

“Type-Based Alias Analysis” 2/5

[A. Diwan, K.S. McKinley, and J.E.B. Moss]　Citation Count (ACM) 35

- この論文の出す対案：型でやる
  - ややこしい新規な型システムとかではなく、ホントに今そこらの言語にあるような型でやる
  - 主として対象は Modula-3 (Cと違って型安全)
- LV1 [TypeDecl]
  - 型が違うメモリ参照は同じ所を決して指さない
    - mayAlias(e1,e2) = subtypes(type(e1))∩subtypes(type(e2))≠Φ

(軽く読んだ方のメモ)

“Type-Based Alias Analysis” 3/5

[A. Diwan, K.S. McKinley, and J.E.B. Moss]　Citation Count (ACM) 35

- LV2 [FieldTypeDecl]
  - 違うフィールドへのアクセスは違うメモリ参照
    - mayAlias(e1.field1,e2.field2) = [TypeDecl] & field1==field2
- LV3 [SMTypeRefs]
  - LV1は変数の宣言型を使ってたが一般的すぎ
    - e1 := e2 という代入がプログラムにあったら
    - type(e1)とtype(e2)を同値類に突っ込む
    - mayAlias = (subtypes∩eqclass)(type(e1))∩同(e2)≠Φ

(軽く読んだ方のメモ)

“Type-Based Alias Analysis” 4/5

[A. Diwan, K.S. McKinley, and J.E.B. Moss]　Citation Count (ACM) 35

- LV1 だと
  - ArrayList 型の変数と List 型の変数は、常に、エイリアスの可能性ありと判定される
- LV3 だと
  - プログラム中に ArrayList 型と List 型の間の代入がなければ（i.e., プログラマがArrayListは常にArrayList型で使い、Listには入れないでいれば） エイリアスの可能性なしと判定
  - いずれにせよ、ぴったり同じ型の変数は依然として常にエイリアスの可能性ありと判定

(軽く読んだ方のメモ)

“Type-Based Alias Analysis” 5/5

[A. Diwan, K.S. McKinley, and J.E.B. Moss]　Citation Count (ACM) 35

- すっごいシンプル。速い
  - プログラムサイズの線形時間に近い(と思う)
  - でもこんなんで精度出るの？
- RLE (Redundant Load Elimination)という最適化に対する有効度で評価
  - わるくない
  - これだけでも、8個中6個のベンチマークでRLの95%以上を検出できた