引言

如果两个内存访问会触碰一段相同的内存，且其中至少有一个是写操作，则认为它们之间存在依赖(即不是独立的)。

本文假定数组总是 row-major 的，考察如下形式的循环：

```
for (i1 = d1; i1 <= u1; i1 += s1) {
    for (i2 = d2; i2 <= u2; i2 += s2) {
        // 第 n 层循环
        for (in = dn; in <= un; in += sn) {
      // 循环体
     }
    }
}
```

- 第 k 层循环的迭代变量 i_k 的下界 d_k、上界 u_k 均是外层迭代变量 $i_1, i_2, \cdots, i_{k-1}$ 的仿射函数 (形如 $f(x_1, x_2, \cdots) = c_0 + c_1x_1 + c_2x_2 + \cdots$，其中 $c_0, c_1, c_2, \cdots$ 均为常量)
- 步长 s_k 为常量
- $u_k - d_k$ 可整除 s_k
- 循环体内的数组访问 (形如 $\mathrm{a}[j_1, j_2, \cdots]$) 的索引 $j_1, j_2, \cdots$ 均是外层迭代变量 $i_1, i_2, \cdots$ 的仿射函数
- 若 a 和 b 表示两个不同的数组，则 $\mathrm{a}[\cdots]$ 与 $\mathrm{b}[\cdots]$ 之间是独立的

数组访问依赖分析要解决的核心问题：

(1)数组访问间是否独立

(2)能否通过等效变换，将有 依赖关系 的数组访问安排在一起

编译器 作各种变换都需要保持内存访问间的依赖。数组访问依赖分析可用于判断 Loop Interchange、Loop Vectorization 等许多变换的合法性；通过将有依赖关系的数组访问安排在一起，可以提升程序的时间/空间局部性，相互独立的部分也能并行。

下面看一个例子：

```
for (i = 1; i < 100; i += 1)
    for (j = 0; j < i; j += 1)
```

```
        a[j, i+1] = a[j, i] + 1
```

$a[j, i+1]$ 与 $a[j, i]$ 之间存在依赖。

作如下 Loop Interchange 变换以提升程序的空间局部性：

```
for (j = 0; j < 99; j += 1)
    for (i = j + 1; i < 100; i += 1)
        a[j, i+1] = a[j, i] + 1
```

> 注意：变换前后保持了依赖关系，但内层循环的不同执行之间已不存在依赖，从而可以并行处理（按 j 将任务分割，交给一个核执行）。

数组访问的数学表述

若一个迭代变量 i 的下界为 d，步长为 s，可用一个新的归一化（即下界和步长均为 1）迭代变量 $j = \frac{i-d+s}{s}$ 代替。如下所示：

```
for (i = d; i <= U; i += S)
```

```
for (j = 1; j <= (U - L + S) / S; j += 1)
    i = j * S - S + L
```

不失一般性，令迭代变量均为 归一化 的。

一个 n 层循环，i_k 为第 k 层循环的迭代变量，i_k 上界为 u_k，称 $\mathrm{I}=[i_1,i_2,\cdots,i_n]^T$ 为迭代向量，I 所有取值的集合为迭代空间。记 $\mathrm{U}=[u_1,u_2,\cdots,u_n]^T$，有 $\mathrm{U}-\mathrm{I}\geq 0$ 。因为 u_k 为 $i_1,i_2,\cdots,i_{k-1}$ 的仿射函数，u_k-i_k 也为 I 的仿射函数，从而 $\mathrm{U}-\mathrm{I}$ 可表示为 $\mathrm{BI}+\mathrm{b}$。

一个静态数组访问 $a[j_1,j_2,\cdots,j_m]$，称 $[j_1,j_2,\cdots,j_m]^T$ 为访问向量。因为 j_k 为 I 的仿射函数，访问向量可表示为 $\mathrm{FI}+\mathrm{f}$。

一个静态数组访问对应一个四元组 $\mathcal{F}=\langle\mathrm{F},\mathrm{f},\mathrm{B},\mathrm{b}\rangle$，其表示一个映射 $\mathrm{I}\mapsto\mathrm{FI}+\mathrm{f},\ \mathrm{BI}+\mathrm{b}\geq 0$。$\mathrm{I}$ 的每一个取值对应一个动态数组访问。

看一个例子：

```
for (i = 1; i <= 100; i += 1)
    for (j = 1; j <= 3 * i; j += 1)
        a[i, i + j * 2 + 1] += 1
```

静态数组访问 a[i, i + j * 2 + 1] 对应：

$$\mathrm{F}=\begin{bmatrix}1&0\\1&2\end{bmatrix}\quad \mathrm{f}=\begin{bmatrix}0\\1\end{bmatrix}\quad \mathrm{B}=\begin{bmatrix}-1&0\\3&-1\end{bmatrix}\quad \mathrm{b}=\begin{bmatrix}100\\0\end{bmatrix}$$

判断数组访问间是否独立

两个静态数组访问 $\mathcal{F}_1=\langle\mathrm{F}_1,\mathrm{f}_1,\mathrm{B}_1,\mathrm{b}_1\rangle$ 、$\mathcal{F}_2=\langle\mathrm{F}_2,\mathrm{f}_2,\mathrm{B}_2,\mathrm{b}_2\rangle$，它们访问同一个数组，可能位于不同的循环，且其中至少有一个是写操作。

若以下方程有解，则认为 $\mathcal{F}_1$、$\mathcal{F}_2$ 之间存在依赖：

$$\begin{aligned}\mathrm{F}_1\mathrm{I}_1+\mathrm{f}_1&=\mathrm{F}_2\mathrm{I}_2+\mathrm{f}_2\\ \mathrm{B}_1\mathrm{I}_1+\mathrm{b}_1&\geq 0\\ \mathrm{B}_2\mathrm{I}_2+\mathrm{b}_2&\geq 0\end{aligned}$$

上述方程的求解是典型的整数规划问题，可参阅相关资料，不再赘述。此类问题是 NP 完全问题，无法精确求解。实践中通常根据一系列启发式的依赖关系测试作出判断。

仿射划分

为了将有依赖关系的动态数组访问安排在一起，我们需要找到一个划分：若两个动态数组访问之间存在依赖，则它们应当属于同一个集合。

正式地，令 $j = CI + c$ 给出 I 对应的动态数组访问集合的一个划分。我们希望找到 C_1、c_1、C_2、c_2，使得

$$\begin{array}{l} F_1 I_1 + f_1 = F_2 I_2 + f_2 \\ \quad B_1 I_1 + b_1 \ge 0 \\ \quad B_2 I_2 + b_2 \ge 0 \end{array} \implies C_1 I_1 + c_1 = C_2 I_2 + c_2$$

我们希望划分包含的元素尽量少，亦即 C 的秩应当尽量大。

上述问题的求解与整数规划类似，不再赘述。

可按照划分转换 I 对应的循环，使得 i_a、i_b 位于同一个循环，当且仅当 $Ci_a = Ci_b$，其中 i_a、i_b 属于 I 对应的迭代空间。

在分别转换 I_1、I_2 对应的循环后，再进行重排（即将具有相同 j 值的部分安排在一起）。

每一个仿射划分都可以分解为以下几种基本转换之叠加，每种基本转换对应一类对源码的简单修改。

1. Fusion

转换前:

```
for (i = 1; i <= N; i++)
    Y[i] = Z[i]
for (i = 1; i <= N; i++)
    X[i] = Y[i]
```

划分：

$$C_1 = [1] \quad c_1 = [0] \quad C_2 = [1] \quad c_2 = [0]$$

转换后:

```
for (j = 1; j <= N; j++)
    X[j] = Y[j]
```

```
    Y[j] = Z[j]
```

2. Fission

转换前:

```
for (i = 1; i <= N; i++)
    Y[i] = Z[i]
    X[i] = Y[i]
```

划分:

$$C_1 = [1] \quad c_1 = [0] \quad C_2 = [1] \quad c_2 = [0]$$

转换后:

```
for (j = 1; j <= N; j++)
    Y[j] = Z[j]
for (j = 1; j <= N; j++)
    X[j] = Y[j]
```

3. Re-indexing

转换前:

```
for (i = 1; i <= N; i++)
    Y[i] = Z[i]
    X[i] = Y[i-1]
```

划分:

$$C_1 = [1] \quad c_1 = [0] \quad C_2 = [1] \quad c_2 = [-1]$$

转换后:

```
if (N >= 1)
 X[1] = Y[0];
for (j = 1; j <= N - 1; j++)
 Y[j] = Z[j]
 X[j+1] = Y[j]
if (N >= 1)
 Y[N] = Z[N]
```

4. Scaling

转换前:

```
for (i = 1; i <= N; i++)
    Y[2*i] = Z[2*i];
for (i = 1; i <= 2*N; i++)
    X[i] = Y[i];
```

划分:

$$C_1 = [2] \quad c_1 = [0] \quad C_2 = [1] \quad c_2 = [0]$$

转换后:

```
for (j = 1; j <= 2*N; i++)
 if (j % 2 == 0)
  Y[j] = Z[j]
    X[j] = Y[j];
```

5. Reversal

转换前:

```
for (i = 1; i <= N; i++)
    Y[N-i] = Z[i];
for (i = 1; i <= N; i++)
```

```
    X[i] = Y[i];
```

划分：

$$C_1 = [-1] \quad c_1 = [N] \quad C_2 = [1] \quad c_2 = [0]$$

转换后:

```
for (j = 1; j <= N; j++)
    Y[j] = Z[N-j];
    X[j] = Y[j];
```

6. Permutation

转换前:

```
for (i1 = 1; i1 <= N; i1++)
 for (i2 = 0; i2 <= M; i2++)
  Z[i1, i2] = Z[i1-1, i2]
```

划分：

$$C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad c = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

转换后:

```
for (j1 = 0; j1 <= M; j1++)
 for (j2 = 1; j2 <= N; j2++)
  Z[j2, j1] = Z[j2-1, j1]
```

7. Skewing

转换前:

```
for (i1 = 0; i1 <= N+M-1; i1++)
 for (i2 = max(1, i1+N); i2 <= min(i1, M); i2++)
```

```
  Z[i1, i2] = Z[i1-1, i2-1]
```

划分：

$$C = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \quad c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

转换后:

```
for (j1 = 1; j1 <= N; j1++)
 for (j2 = 1; j2 <= M; j2++)
  Z[j1, j2-j1] = Z[j1-1, j2-j1-1]
```

在 LLVM 中的应用

DependenceAnalysis pass 实现了基本的数组访问依赖分析，可用于依赖关系判断，也能进行简单的仿射划分。LoopFuse、LoopInterchange、LoopTiling 等 transformation pass 均需要 DependenceAnalysis 提供的信息才能工作。

在 MLIR 'affine' dialect 中，数组访问依赖分析的实现更加完备(因为在 MLIR 中数组访问的相关信息被完全保留了，而非像在 IR 中那样需要从 getelementptr 等指令反向推导出来)，不仅支持依赖关系判断，更支持较复杂的仿射划分。MLIR 'affine' dialect 尤其适用于存在大量复杂的高维数组操作的场景，通过仿射划分自动分割出可以并行的部分，应用常常能因此得到成倍性能提升。

参考

1. Compilers: Principles, Techniques, & Tools, Second Edition. Jeffrey D. Ullman

2. Practical Dependence Testing. Goff, Kennedy, Tseng

3. https://mlir.llvm.org/docs/Dialects/Affine