并行编译与优化
Parallel Compiler and Optimization

计算机研究所编译系统室 方建滨

Lecture 16 Vectorization

第十六课：向量化

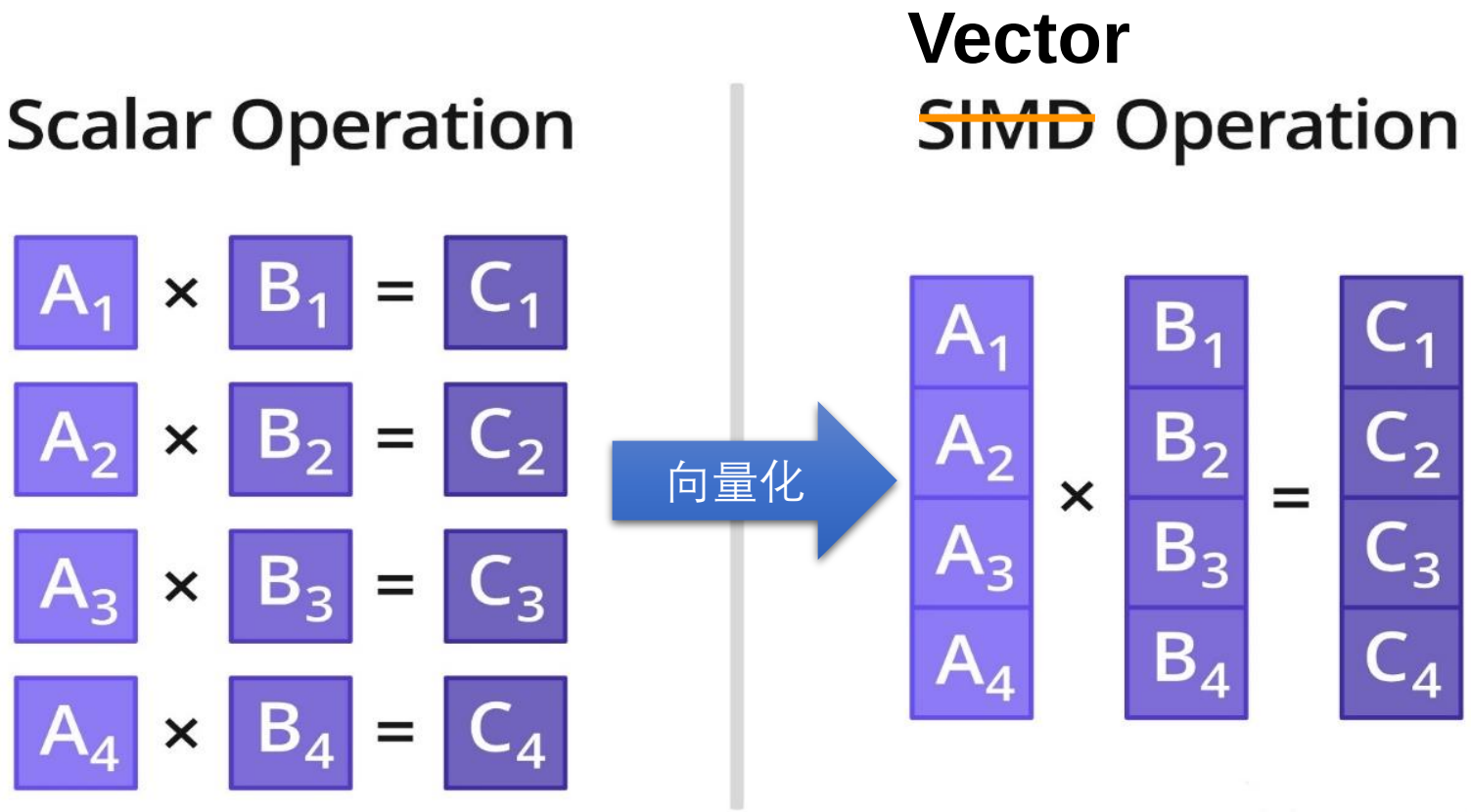
2024-05-28

学习内容

- 1. 向量化简介
- 2. 向量化判定条件
- 3. 向量化与循环变换
- 4. 向量化编译实现

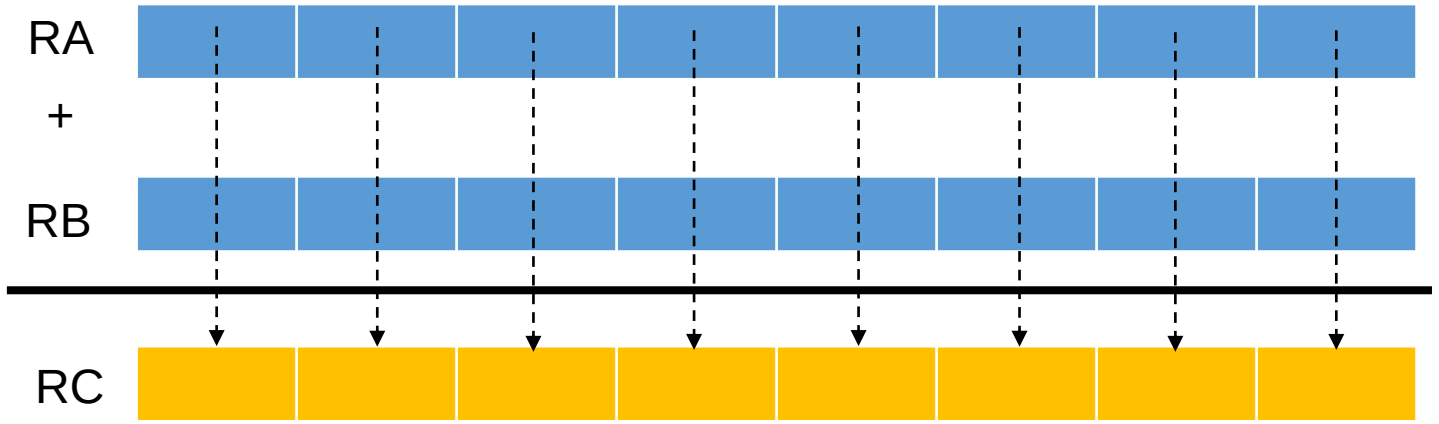
什么是向量化?

- 一个向量(Vector)是由一组元素组成的有序集
- 元素的个数称为向量长度



向量运算

- 向量运算在Intel Xeon Gold 6130上的执行过程
  - 假设有位于外存的A、B、C三个double类型向量，求C=A+B
  - 将A、B加载到向量寄存器(RA, RB)，再用向量加法部件将它们相加暂存在向量寄存器(RC)，最后将结果送到C
  - 当向量长度大于向量寄存器长度时，运算分段(或分块)完成



什么是向量化?

诸葛连弩中的向量化思想

向量计算机

- 除标量寄存器和标量功能部件外，向量计算机还专门设有向量寄存器、向量长度寄存器、向量屏蔽寄存器、向量流水功能部件和向量指令系统，能对向量运算进行高速处理
- 典型代表：YH-I, Cray-I, Earth Simulator

银河-I

Cray I

向量计算机

Cray-1巨型机

- 1976年由Cray Research设计
- 计算峰值性能：136Mflops
- 使用单个主处理器+向量单元
- 第一次使用了晶体管存储器
- 主要用于核爆实验模拟和密码破译

Cray I

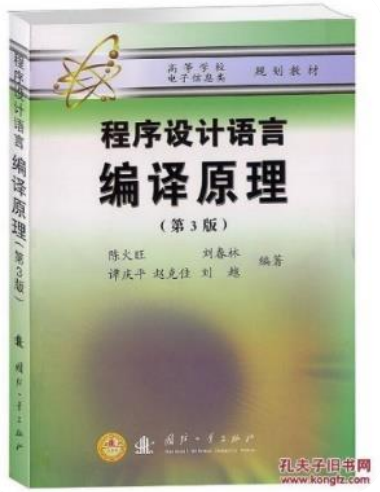
向量计算机

YH-1巨型机

- 1983年银河 I 巨型计算机研制成功，采用向量机体系结构
- 运算速度达每秒1亿次，是我国首台亿次计算机
- 研制了我国首个向量FORTRAN77语言编译系统

银河-I

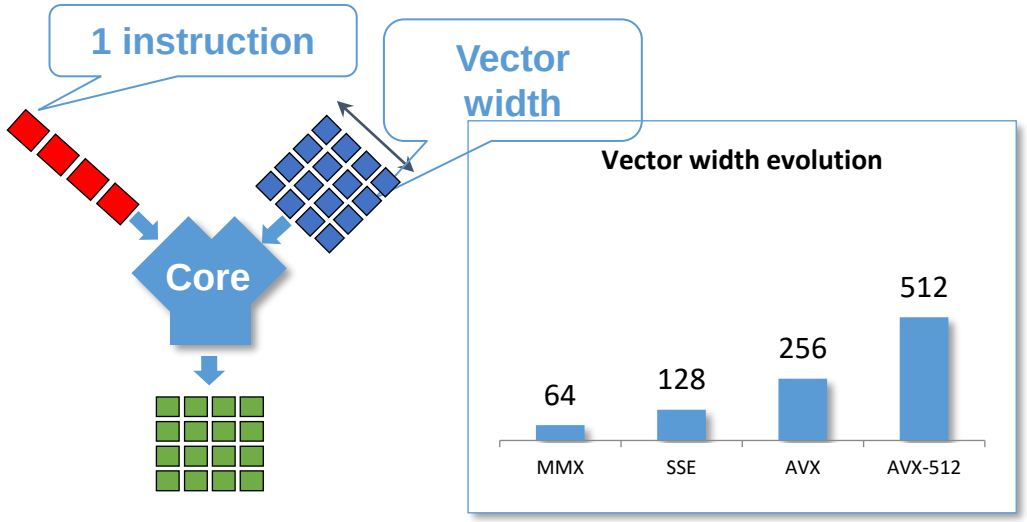
陈火旺院士

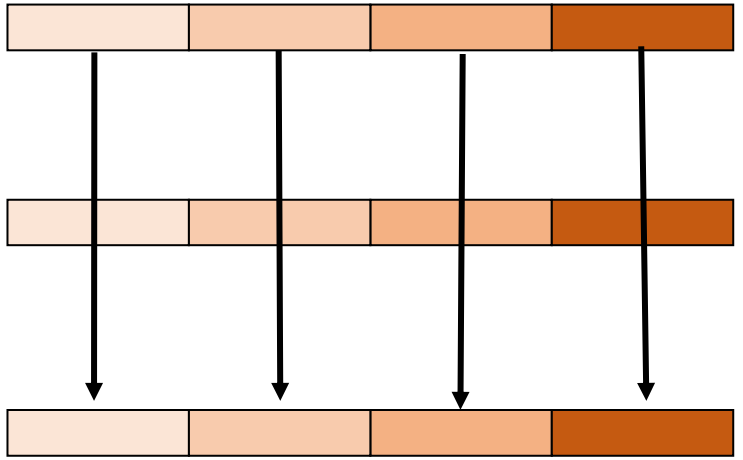


编译原理教材

SIMD: 向量化的进化版

- 现代CPU SIMD ISA
  - X86 SSE/AVX/AVX2/AVX512, ARM NEON/SVE (256-512)
  - NUDT Matrix-3000 (1024)
- 最近的CPU有AMX, SME, MMA, ...





如何利用向量?

- 移植性好
- 可扩展性好
- 零开发开销
- 未知性能

自动向量化

```
…
float *A, *B, *C;
…
for(int i…)
  C[i] = A[i] + B[i];
$> clang –O2 …
```

- 显式地表示向量化
- 可移植性好
- 可扩展性好
- 低代码编写开销
- 可预测性能

显式向量化

```
…
float *A, *B, *C;
…
#pragma omp simd
for(int i…)
  C[i] = A[i] + B[i];
$> clang –O2 –Xclang – fopenmp=…
```

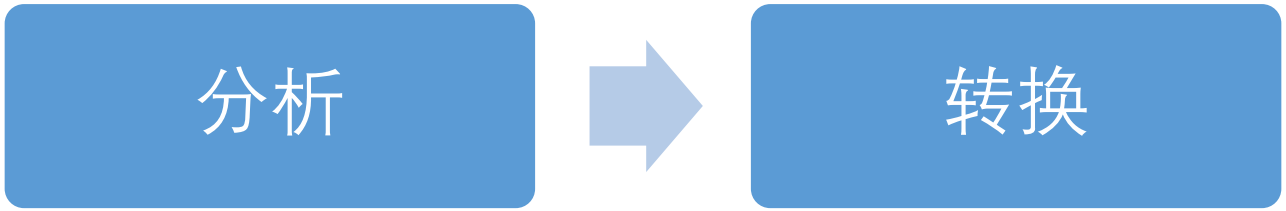
- 性能最好
- 可移植性差
- 可扩展性差
- 开发开销大

手工向量化

```
#include "xmmintrin.h“
…
float *A, *B, *C;
__m128 a, b, c;
…
for(int i…)
{
 a = _mm_load_ps(A + i*4);
 b = _mm_load_ps(B + i*4);
 c = _mm_add_ps(a, b);
  _mm_store_ps(C+i * 4, c);
}
$> clang …
```

自动向量化

- 自动向量化指的是在依赖关系分析的基础上，通过一系列的程序变换技术，消除妨碍向量化的依赖关系，使程序中尽可能多的循环向量化
- 一个可向量化的循环在源语言一级可用向量语言或串行语言加向量化编译指导命令来表示



自动向量化

- 向量化对象
  - 数组赋值语句（Fortran）
  - FORALL（Fortran）
  - 循环
- 如何向量化嵌套循环$L=(L_1, L_2, \cdots, L_m)$?
  - 判定可向量化循环
  - 变换循环形式

学习重点

学习内容

- 1. 向量化简介
- 2. 向量化判定条件
- 3. 向量化与循环变换
- 4. 向量化编译实现

可向量化循环的定义

- 如果循环中**仅含赋值语句**，且每个语句S_i都可以在循环中它之后的语句执行之前，执行完对应于**循环区间**的每一个实例，而结果与串行执行时相同，则称该循环是**可向量化的**（vectorizable）

```
L:   do  I = 1,N
S1:      A(I) = D(I) * 2
S2:      C(I) = B(I) +A(I)
     enddo
```

串行执行时，语句实例的执行顺序：

$S_1(1), S_2(1), S_1(2), S_2(2), \ldots, S_1(N), S_2(N)$

判定向量执行时，语句实例的执行顺序：

$S_1(1), S_1(2), \ldots, S_1(N), S_2(1), S_2(2), \ldots, S_2(N)$

循环*L*可向量化

可向量化循环的定义

- 如果循环中**仅含赋值语句**，且每个语句S_i都可以在循环中它之后的语句执行之前，执行完对应于**循环区间**的每一个实例，而结果与串行执行时相同，则称该循环是**可向量化的**（vectorizable）

```
   do  I = 2,N
S1:    A(I) = C(I-5) * 2
S2:    C(I) = B(I)
   enddo
```

串行执行时，语句实例的执行顺序：

$S_1(1), S_2(1), S_1(2), S_2(2), \ldots, S_1(N), S_2(N)$

判定向量执行时，语句实例的执行顺序：

$S_1(1), S_1(2), \ldots, S_1(N), S_2(1), S_2(2), \ldots, S_2(N)$

循环*L*不可向量化

向量化判定条件

- 定理：一个循环是可向量化的，当且仅当

  对于循环体中的任意两个语句S和T，

  ① 当S<T时，不存在S依赖于T的依赖关系(即TδS)；

  ② 当S=T时，不存在S依赖于S的流依赖关系(即Sδ^fS)

- 证明：
  - 必要性
  - 充分性

必要性证明（1/2）

- 假设当S<T时，存在S依赖于T的依赖关系TδS，则由依赖关系的定义，存在实例S(i)和T(j)，且j<i，因此实例T(j)必须先于S(i)而执行。但向量化则要先执行完S的所有迭代，再执行T的所有迭代，即导致S(i)先于T(j)而执行，故不能向量化

```
   do  I = 2,N
S1:      A(I) = C(I-1) * 2
S2:      C(I) = B(I)
   enddo
```

$S_1 < S_2$ and $S_2 \delta^f S_1$ ✗

必要性证明（2/2）

- 假设当S=T时，存在S依赖于S的流依赖关系$S\delta^f S$，则意味着存在实例S(i)和S(j)，且j<i，使得S(j)先对某个变量定值，而S(i)后使用该变量的值。但向量化导致先使用该变量的老值，再对变量进行定值，因此也不能向量化。

```
   do  I = 2,N
S1:      A(I) = A(I-1) * 2
   enddo
```

$S_1 \delta^f S_1$ ✗

充分性证明（1/2）

- 设当S<T时，不存在S依赖于T的依赖关系TδS，则只可能存在SδT的依赖关系，其相关迭代为{(i, j): j=i+x, x≥0, p≤i≤q}，p、q分别时循环的下、上界。因此有i<j，即语句S的所有相关实例S(i)都在语句T的所有相关实例T(j)之前执行。故，由定义可知，这个循环是可向量化的。

充分性证明（2/2）

- 又设当S=T时，不存在S依赖于S的流依赖关系$S\delta^f S$，则只能可能存在S依赖于S的反依赖关系或输出依赖关系。
  - 对于$S\delta^a S$，根据依赖关系的定义，存在实例S(i)和S(j)，且j<i，使得S(j)先使用某个变量的值，而S(i)后对该变量定值，因此它不妨碍向量化；
  - ~~对于$S\delta^o S$，向量化总能保证对表达式左部量的定值顺序与串行执行时一致，因此它也不妨碍向量化。~~

向量化判定条件

- 例子

```
   do  I = 1,N
S1:    A(I) = D(I) * 2
S2:    C(I) = B(I) +A(I)
   enddo
```

$S_1 < S_2$ and $S_1 \delta^f S_2$ √

```
   do I = 1,N
S1:    A(I) = B(I) +C(I+1)
S2:    C(I) = A(I)+D(I)
   enddo
```

$S_1 < S_2$

$S_1 \delta^f S_2$ $S_1 \delta^a S_2$ √

向量化判定条件

例子

```
   do  I = 2,N
S1:      A(I) = C(I-5) * 2
S2:      C(I) = B(I)
   enddo
```

$S_2\ \delta^{f}_{<5>}\ S_1$ √

向量化判定条件

- 例子

```
   do  I = 2,N
S1 :     A(I) = A(I-1) + 1
   enddo
```

$S_1 \delta^f S_1$ ×

```
   do  I = 2,N
S1 :     A(I) = A(I+1) + 1
   enddo
```

$S_1 \delta^a S_1$ √

嵌套循环的向量化判定

- 定理：嵌套循环$L=(L_1, L_2, \cdots, L_m)$的**最内层循环$L_m$**可向量化当且仅当

  对于循环体中的任意两个语句S和T，

  - 当S＜T时，不存在方向向量为$\sigma_1=\sigma_2=\cdots\sigma_{m-1}=0, \sigma_m=1$的S依赖于T的依赖关系(即$T\delta S$)；
  - 当S＝T时，不存在方向向量为$\sigma_1=\sigma_2=\cdots\sigma_{m-1}=0, \sigma_m=1$的S依赖于S的流依赖关系(即$S\delta^f S$)

必要性证明（1/2）

- 假设当S<T时，存在具有方向向量σ=(0, …, 0, 1)的S依赖于T的依赖关系TδS，则由依赖关系与方向向量的定义，存在实例S(i)和T(j)且j<$_m$i。因此在第m层上，实例T(j)必须先于S(i)而执行，故不可以被向量化。

```
   do  I = 2,N
S1:    A(I) = C(I-1) * 2
S2:    C(I) = B(I)
   enddo
```

$S_1 < S_2$ and $S_2 \delta^f S_1$ σ=(1) ✗

必要性证明（2/2）

- 假设当S=T时，存在具有同样方向向量的S依赖于S的流依赖关系$S\delta^f S$，则意味着在第m层上存在实例S(i)和S(j)，且$j <_m i$，使得S(j)先对某个变量定值，而S(i)后使用该变量的值。但对于一个语句，向量执行时将先取等号右边的向量，经运算后再赋给等号左边的向量，这就导致先使用老值后定义新值，破坏了依赖关系，故不能向量化。

```
   do  I = 2,N
S1:      A(I) = A(I-1) * 2
   enddo
```

$S_1 \delta^f S_1$ σ=(1) ✗

充分性证明（1/2）

- 设当S<T时，不存在具有方向向量σ的S依赖于T的依赖关系TδS。要证明的是具有其它形式方向向量的依赖关系不影响向量化。
  - 对于SδT的依赖关系，因依赖顺序与语句顺序一致，故均不影响向量化；
  - 对于形如(*, …, *, 0)的TδS的依赖关系，其方向向量的依赖关系与循环L_m无关，故不妨碍的向量化；
  - 对于形如(*, …, *, ±1)的TδS的依赖关系，其方向向量总是非负的，且它不是(0, …, 0, 1)形式，因此一定存在前导元素$\sigma_l>0$且$1\le l<m$，即TδS是在第l层的依赖关系。由于向量化仅针对循环L_m，外层循环的L_l仍保持串行执行，因而不会破坏L_l层的这个依赖关系，故L_m可以向量化。

充分性证明（2/2）

- 又设当S=T时，不存在具有方向向量σ=(0, …, 0, 1)的S依赖于S的流依赖关系，则只可能存在具有其它形式方向向量的S依赖于S的流依赖关系，或任意方向向量的S依赖于S的反依赖关系或输出依赖关系
  - 对于具有其它形式方向向量的S依赖于S的流依赖关系，可类似于S<T的形况证明；
  - 对于具有任意方向向量的Sδ^{a}S，根据依赖关系定义，存在实例S(i)和S(j)，且j<$_{m}$i，使得S(j)先使用某个变量的值，而S(i)后对该变量定值，因此不妨碍向量化；
  - 对于具有任意方向向量的Sδ^{o}S，向量化总能保证对表达式左部量的定值顺序与串行执行时一致，因此它也不妨碍向量化

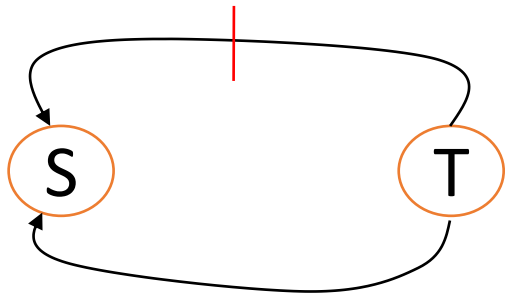
嵌套循环的向量化判定

- 例子

```
   do  I = 2,N-1
      do  J = 2,N-1
S:       A(I,J) = B(I-1,J) +C
T:       B(I,J) = A(I,J+1)/2
      enddo
   enddo
```

$for\ A,\ T\ \delta^{a}_{(0,1)}S$

$for\ B,\ T\ \delta^{f}_{(1,0)}S$



内层循环不能被向量化

循环变换与向量化

- 当循环中依赖距离不小于向量长度时，可通过循环分段使新的内层循环可向量化

```
   do  I = 2,N
S1:      A(I) = C(I-5) * 2
S2:      C(I) = B(I)
   enddo
```

$S_1 < S_2$ and $S_2\ \delta^f_{<5>}\ S_1$

Vector length = 4

```
   do  I = 2, N, 4
      do II = I, II+4, 1
S1:      A(II) = C(II-5) * 2
S2:      C(II) = B(II)
      endo
   enddo
```

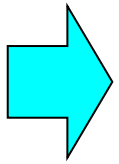
√

学习内容

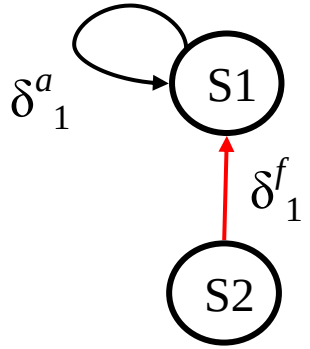
- 1. 向量化简介
- 2. 向量化判定条件
- 3. 向量化与循环变换
- 4. 向量化编译实现

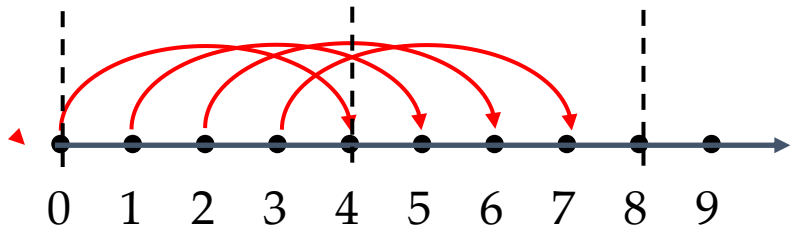
循环分段（1/2）

```
      do I=0, N
S1      C(I)=A(I)+C(I+1)
S2      A(I+4)=D(I)
      enddo
```



```
      do Is = 0, N, 4
        do I=Is, min(N, Is+3)
S1        C(I)=A(I)+C(I+1)
S2        A(I+4)=D(I)
        enddo
      enddo
```





δ, d=(4), σ=(1)

循环分段（2/2）

- **确定分段大小S**
  - **缓存大小**
  - **向量化单元长度**

```
L:  do  Is = 1, N, S
        do  I = Is, min(N, Is+S-1)
            H(I)
        enddo
    enddo
```

- 循环剥除
  - 避免调用min函数

```
    N1 = ⌊N/S⌋
    N2 = N1*S
L:  do  Is = 1, N2, S
        do  I = Is, Is+S-1
            H(I)
        enddo
    enddo
    do I = N2+1, N
       H(I)
    enddo
```

规约变量替换

- 若一个变量的一组值是一个算数或几何级数，则称该变量是规约变量
  - 这些变量常常是循环控制（索引）变量的函数
  - 当数组下标是规约变量的函数时，循环不能向量化
- 规约变量替换可以消除规约变量
  - 不但能减少循环体中的运算量，而且还能使循环向量化

```
L:  do  I = 1, N
      J = 2 * I + 1
      A(I) = A(I) + B(J)/2
    enddo
```

```
L:  do  I = 1, N
      J = 2 * I + 1
      A(I) = A(I) + B(2 * I + 1)/2
    enddo
```

递归下标消除

- 有时循环中数组元素的下标是循环控制变量的递归函数，称为递归下标
- 使用递归下标消除技术来使循环可向量化

```
L：  J = 0
     do  I = 1, N
        J = J + I
        A(I) = A(I) + B(J)
     enddo
```

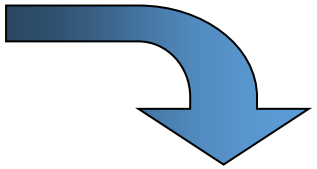
```
L：  ~~J = 0~~
     do  I = 1, N
        ~~J = J + I~~
        A(I) = A(I) + B(I * (I + 1)/2)
     enddo
```

循环剥除（1/2）

COMP CCRG Open MP

删除某些数据依赖

```
   do i = 1, n
S:     y(i, n) = y(1, n) + y(n,n)
    enddo
```



i = 1 and i = n 有依赖

```
      y(1, n) = y(1, n) + y(n,n)
      do i = 2, n-1
S:       y(i, n) = y(1, n) + y(n,n)
      enddo
      y(n, n) = y(1, n) + y(n,n)
```

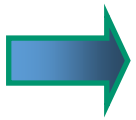
经过剥除，循环可以被向量化

循环剥除（2/2）

COMP CCRG Open MP

消除循环中的分支判断

```
for (i =1; i < n; i++) {
   if (i == 1)
      a[i] = 0;
   else
      a[i] = b[i-1];
 }
```



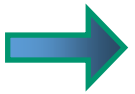
```
a[1] = 0;
for (i=2; i < n; i++) {
     a[i] = b[i-1]
 }
```

循环分裂（1/2）

- 循环分裂是将一个循环的索引集合分成两部分，并对两部分的循环体作必要的修改
- 删除与循环索引变量相关的条件判断

```
do i = 1, 100
   A(i) = B(i) + C(i)
   if i > 10 then
      D(i) = A(i) + A(i-10)
   endif
enddo
```

分裂前



```
do i = 1, 10
   A(i) = B(i) + C(i)
enddo
do i = 11, 100
   A(i) = B(i) + C(i)
   D(i) = A(i) + A(i-10)
enddo
```

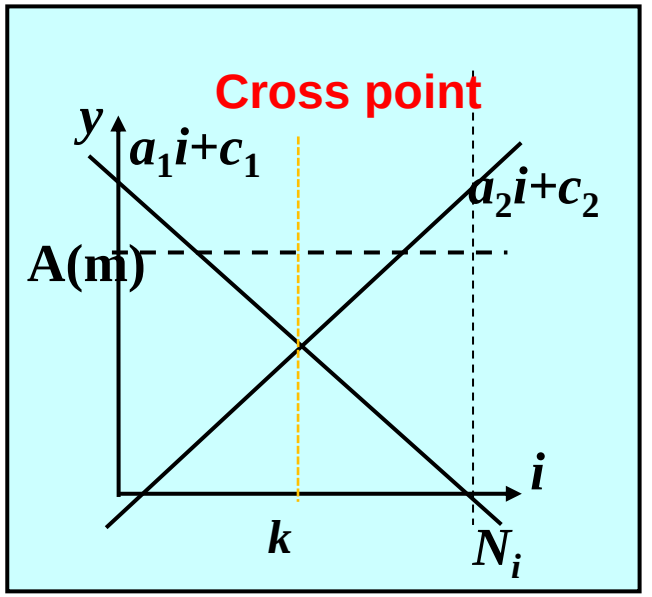
分裂后

循环分裂（2/2）

消除数据依赖

```
      do i = 1, n
S:       y(i) = y(n-i+1)
      enddo
```

```
    do i = 1, (n+1)/2
       y(i) = y(n-i+1)
    enddo
    do i = (n+1)/2+1, n
       y(i) = y(n-i+1)
    enddo
```



k=(n+1)/2

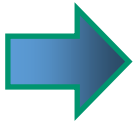
经过分裂，循环可以被向量化

循环去开关化

- 循环去开关化是指将循环不变的条件分支移至循环体之外

```
do i=1, n
   do j=2, n
      if (t(i) .gt. 0) then
         a(i,j) = a(i, j-1)*t(i)
      else
         a(i,j) = t(i)
      endif
   enddo
enddo
```

变换前



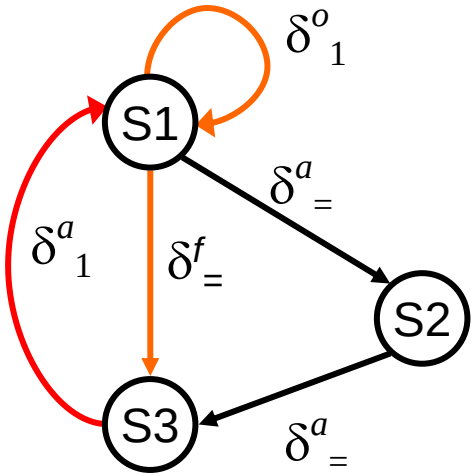
```
do i=1, n
   if (t(i) .gt. 0) then
      do j=2, n
         a(i,j) = a(i, j-1)*t(i)
      enddo
   else
      do j=2, n
         a(i,j) = t(i)
      endo
   endif
enddo
```

变换后

标量扩张（1/2）

- 标量扩张使得原本不能向量化的循环变得可以被向量化

```
L:   do i=1, N
S1:    T = A[i]
S2:    A[i]=B[i]
S3:    B[i] = T
     enddo
```

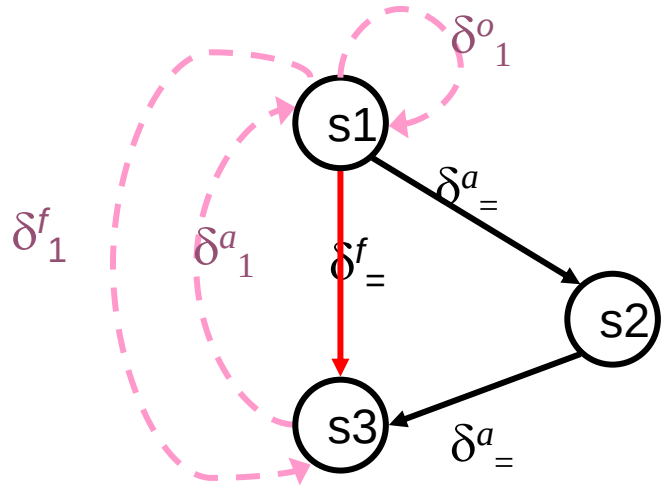


循环L不能被向量化

标量扩张（2/2）

- 标量扩张使得原本不能向量化的循环变得可以被向量化

```
L':   do i=1, N
S1:     $T[i] = A[i]
S2:     A[i]=B[i]
S3:     B[i] = $T[i]
      enddo
      T=$T[N]
```



经过标量扩展，循环L'可以被向量化

循环联合

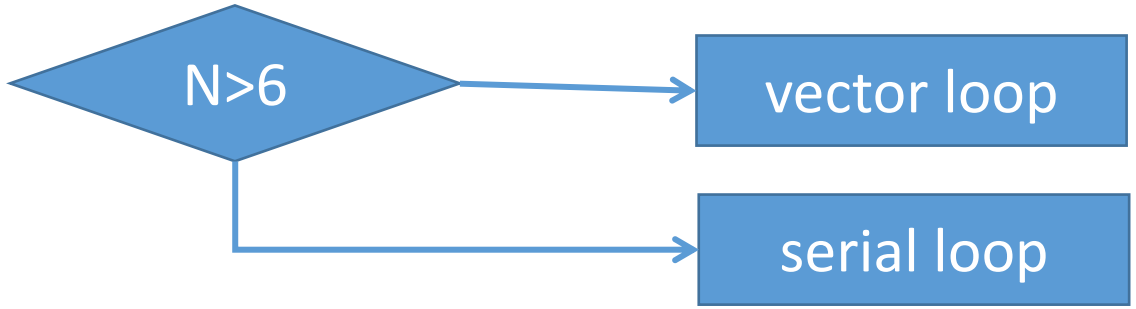
- 循环联合将两个嵌套循环变为一个单层循环
  - 能够增加向量化循环的向量长度
  - 定义一个与原二维数组等价的新的一维数组，代替原数组参加运算，并将原两层循环改写成一层循环

```
L:    do I=1, 8
          do J=1, 8
S1:         A(I, I) = B(I, J) + 2
          enddo
      enddo
```

```
P: equivalence(A(1, 1), AA(1)),
(B(1, 1), BB(1))
      do I=1, 64
S1:       AA(I) = BB(I) + 2
      enddo
```

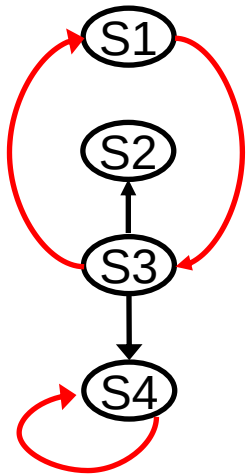
循环多版本

- 循环多版本技术用来对循环迭代次数不是常量的循环生成一个标量执行版本和一个向量执行版本
- 从硬件角度看，向量运算的启动开销比变量运算大
  - 例如，在Cray机器上，当向量长度小于6时，向量运算速度可能低于对应的标量运算
- 一个可向量化循环的迭代次数在每次被执行时可能不同
  - 当迭代次数小时，用标量计算该循环；反之用向量计算

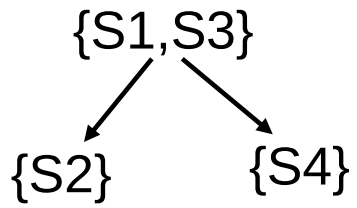


循环分布

```
      do I=4, 100
S1:     A(I)=B(I-2)+1
S2:     C(I)=B(I-1)+F(I)
S3:     B(I)=A(I-1)+2
S4:     D(I)=D(I-1)+B(I-1)
      enddo
```



strong connected components




```
L1:   do I=4, 100
S1:     A(I)=B(I-2)+1
S3:     B(I)=A(I-1)+2
      enddo
```

```
L2:   do I=4, 100
S2:     C(I)=B(I-1)+F(I)
      enddo
```

```
L3:   do I=4, 100
S4:     D(I)=D(I-1)+B(I-1)
      enddo
```

L1, L2, L3　or　L1, L3, L2

语句重排（1/2）

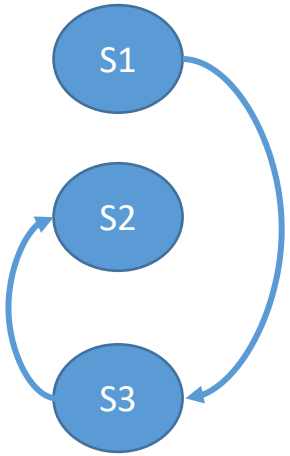
- 语句重排是基于语句依赖图的一种程序变换
  - 改变循环中语句的词法顺序但不改变语句的依赖关系
  - 语句重排常用于循环向量化
- 向量操作能保护与语句执行顺序一致的依赖关系
  - 当一循环有两个语句S和T，S < T，且存在与语句执行顺序相反的依赖关系TδS，则该循环不能被向量化
  - 若循环中依赖关系不构成圈，即不同时存在SδT，则可通过语句重排交换S和T的语序，从而把与语句执行顺序相反的依赖关系（即向上的依赖关系）改为与语句执行顺序一致的依赖关系（即向下的依赖关系），使得该循环可向量化

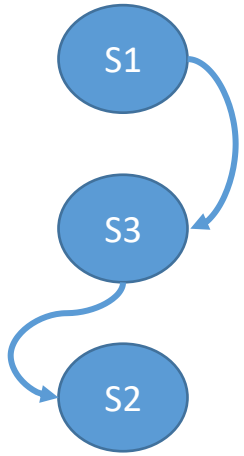
语句重排（2/2）

例子

```
L:   do i=1, N
S1:    A[I] = D[I] * T
S2:    B[I] = (C[I] + E[I])/2
S3:    C[I+1] = A[I] + 1
     enddo
```

```
L:   do i=1, N
S1:    A[I] = D[I] * T
S3:    C[I+1] = A[I] + 1
S2:    B[I] = (C[I] + E[I])/2
     enddo
```

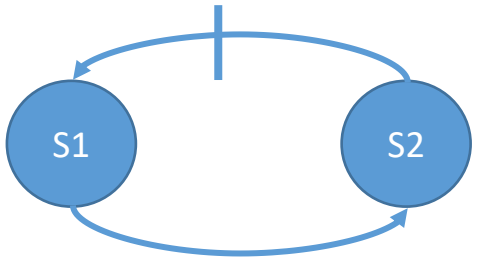




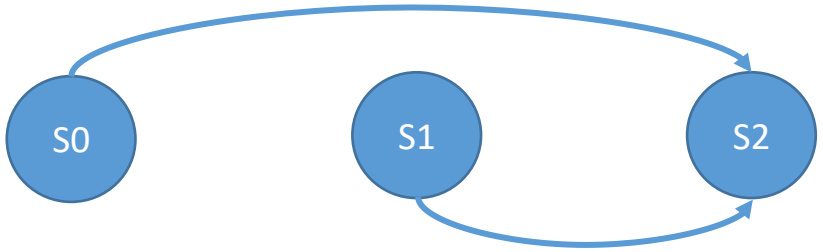
语句分裂

- 当一个循环中的依赖关系构成圈时，语句重排不可将循环向量化；但若构成圈的依赖关系中有一个是反向依赖或输出依赖，则可用语句分裂技术将圈消除，从而使该循环可向量化

```
L:    do i=1, N
S1:     A[I] = B[I] * C[I]
S2:     D[I] = (A[I] + A[I+1])/2
enddo
```



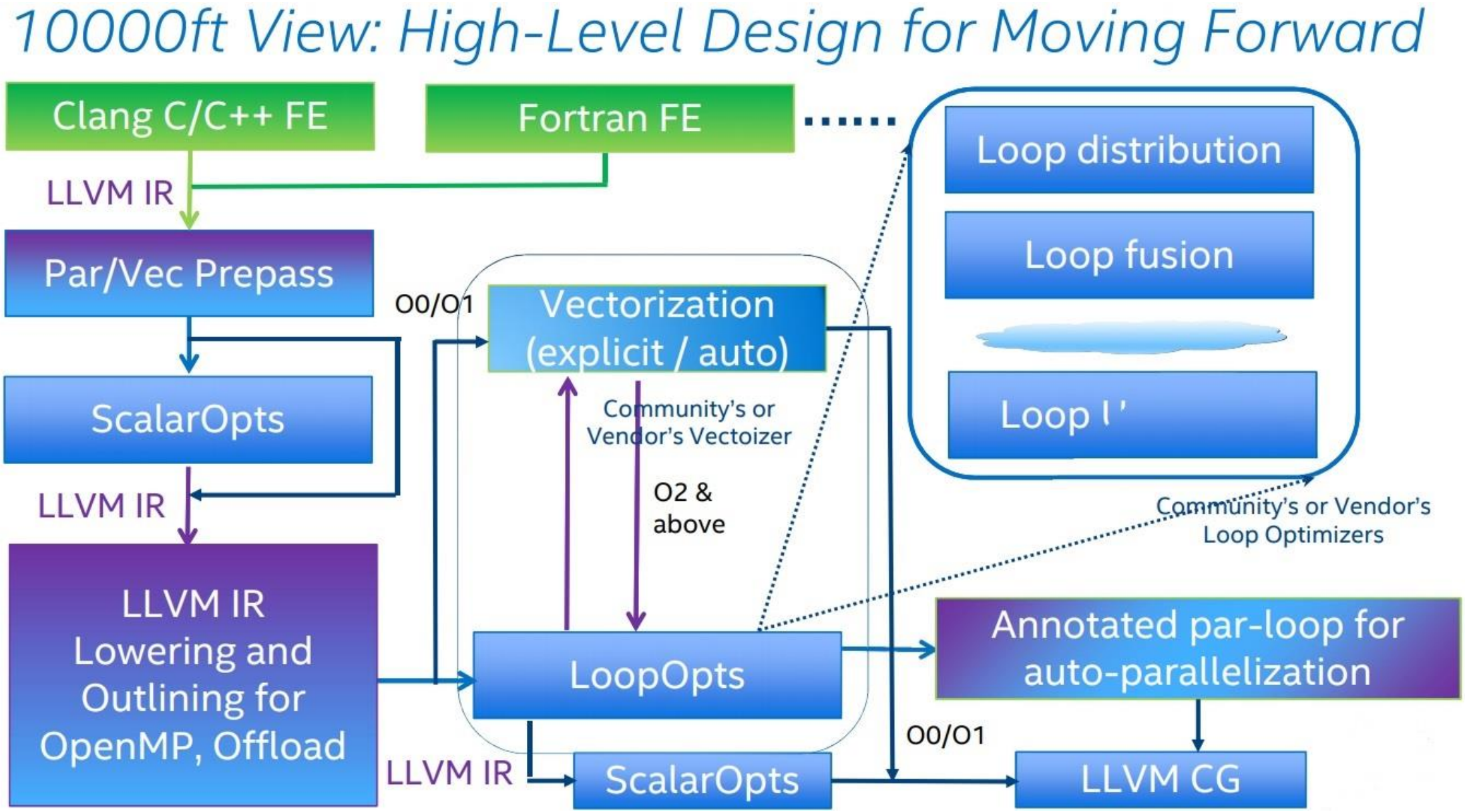

```
L:    do i=1, N
S0:     TEMP[I] = A[I+1]
S1:     A[I] = B[I] * C[I]
S2:     D[I] = (A[I] + TEMP[I])/2
enddo
```



学习内容

- 1. 向量化简介
- 2. 向量化判定条件
- 3. 向量化与循环变换
- 4. 向量化编译实现

自动向量化实现


10000ft View: High-Level Design for Moving Forward
Clang C/C++ FE
Fortran FE
LLVM IR
Par/Vec Prepass
O0/O1
Vectorization (explicit / auto)
ScalarOpts
Community's or Vendor's Vectoizer
O2 & above
LLVM IR
LLVM IR Lowering and Outlining for OpenMP, Offload
LoopOpts
LLVM IR
ScalarOpts
Loop distribution
Loop fusion
Loop I '
Community's or Vendor's Loop Optimizers
Annotated par-loop for auto-parallelization
O0/O1
LLVM CG

LLVM的向量化实现

- 自动向量化将串行代码自动转换为使用向量化部件的代码
- LLVM有两个向量化模块
  - Loop vectorizer：以循环作为向量化对象
  - SLP (a.k.a. superword-level parallelism) vectorizer
    - 合并多个标量运算

```
void foo(int a1, int a2, \
  int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
}
```

循环向量化器使用

- 循环向量化器使能情况
  - -O0没有任何优化
  - -O1可以向量化带编译指导的循环
  - -O2可以向量化所有循环

```
$ clang -O1 -emit-llvm -S file.c
```

- 使用代价模型最优的向量化因子
  - 用户可通过命令行选项进行显式指定

```
$ clang  -mllvm -force-vector-width=8 ...
```

循环向量化器的使用

COMP CCRG Open MP

- 循环编译指导命令可作用到特定的循环上
  - 下例显式向量化while循环

```
#pragma clang loop vectorize(enable)
while(...) {
   ...
}
```

  - 下例显式指定向量化长度

```
#pragma clang loop vectorize_width(2)
for(...) {
   ...
}
```

循环向量化器的使用

COMP CCRG Open MP

- 诊断信息提供关于向量化成功与否及原因
  - -Rpass=loop-vectorize: 识别被成功向量化的循环
  - -Rpass-missed=loop-vectorize: 识别不能被向量化的循环
  - -Rpass-analysis=loop-vectorize: 输出导致向量化失败的语句

```
#pragma clang loop vectorize(enable)
for (int i = 0; i < Length; i++) {
  switch(A[i]) {
  case 0: A[i] = i*2; break;
  case 1: A[i] = i;   break;
  default: A[i] = 0;
  }
}
```

LLVM编译器实现代码


```
$ls -l ./llvm-project/llvm/lib/Transforms/Vectorize
```

```
LoadStoreVectorizer.cpp
LoopVectorizationLegality.cpp
LoopVectorizationPlanner.h
LoopVectorize.cpp
SLPVectorizer.cpp
VectorCombine.cpp
Vectorize.cpp
VPlan.cpp
VPlanDominatorTree.h
VPlan.h
VPlanHCFGBuilder.cpp
VPlanHCFGBuilder.h
VPlanLoopInfo.h
VPlanPredicator.cpp
VPlanPredicator.h
VPlanSLP.cpp
VPlanTransforms.cpp
VPlanTransforms.h
VPlanValue.h
VPlanVerifier.cpp
VPlanVerifier.h
VPRecipeBuilder.h
```

LLVM编译器实现代码

COMP CCRG Open MP

```
$ls -l ./llvm-project/llvm/lib/Analysis
```

```
AliasAnalysis.cpp                   Delinearization.cpp                 LazyValueInfo.cpp                   ProfileSummaryInfo.cpp
AliasAnalysisEvaluator.cpp          DemandedBits.cpp                    LegacyDivergenceAnalysis.cpp        PtrUseVisitor.cpp
AliasAnalysisSummary.cpp            DependenceAnalysis.cpp              Lint.cpp                            README.txt
AliasAnalysisSummary.h              DependenceGraphBuilder.cpp          Loads.cpp                           RegionInfo.cpp
AliasSetTracker.cpp                 DevelopmentModeInlineAdvisor.cpp    LoopAccessAnalysis.cpp              RegionPass.cpp
Analysis.cpp                        DivergenceAnalysis.cpp              LoopAnalysisManager.cpp             RegionPrinter.cpp
AssumeBundleQueries.cpp             DominanceFrontier.cpp               LoopCacheAnalysis.cpp               ReleaseModeModelRunner.cpp
AssumptionCache.cpp                 DomPrinter.cpp                      LoopInfo.cpp                        ReplayInlineAdvisor.cpp
BasicAliasAnalysis.cpp              DomTreeUpdater.cpp                  LoopNestAnalysis.cpp                ScalarEvolutionAliasAnalysis.cpp
BlockFrequencyInfo.cpp              EHPersonalities.cpp                 LoopPass.cpp                        ScalarEvolution.cpp
BlockFrequencyInfoImpl.cpp          FunctionPropertiesAnalysis.cpp      LoopUnrollAnalyzer.cpp              ScalarEvolutionDivision.cpp
BranchProbabilityInfo.cpp           GlobalsModRef.cpp                   MemDepPrinter.cpp                   ScalarEvolutionNormalization.cpp
CallGraph.cpp                       GuardUtils.cpp                      MemDerefPrinter.cpp                 ScopedNoAliasAA.cpp
CallGraphSCCPass.cpp                HeatUtils.cpp                       MemoryBuiltins.cpp                  StackLifetime.cpp
CallPrinter.cpp                     ImportedFunctionsInliningStatistics.cpp MemoryDependenceAnalysis.cpp    StackSafetyAnalysis.cpp
CaptureTracking.cpp                 IndirectCallPromotionAnalysis.cpp   MemoryLocation.cpp                  StratifiedSets.h
CFG.cpp                             InlineAdvisor.cpp                   MemorySSA.cpp                       SyncDependenceAnalysis.cpp
CFGPrinter.cpp                      InlineCost.cpp                      MemorySSAUpdater.cpp                SyntheticCountsUtils.cpp
CFLAndersAliasAnalysis.cpp          InlineSizeEstimatorAnalysis.cpp     MLInlineAdvisor.cpp                 TargetLibraryInfo.cpp
CFLGraph.h                          InstCount.cpp                       models                              TargetTransformInfo.cpp
CFLSteensAliasAnalysis.cpp          InstructionPrecedenceTracking.cpp   ModuleDebugInfoPrinter.cpp          TFUtils.cpp
CGSCCPassManager.cpp                InstructionSimplify.cpp             ModuleSummaryAnalysis.cpp           Trace.cpp
CMakeLists.txt                      Interval.cpp                        MustExecute.cpp                     TypeBasedAliasAnalysis.cpp
CmpInstAnalysis.cpp                 IntervalPartition.cpp               ObjCARCAliasAnalysis.cpp            TypeMetadataUtils.cpp
CodeMetrics.cpp                     IRSimilarityIdentifier.cpp          ObjCARCAnalysisUtils.cpp            ValueLattice.cpp
ConstantFolding.cpp                 IVDescriptors.cpp                   ObjCARCInstKind.cpp                 ValueLatticeUtils.cpp
ConstraintSystem.cpp                IVUsers.cpp                         OptimizationRemarkEmitter.cpp       ValueTracking.cpp
CostModel.cpp                       LazyBlockFrequencyInfo.cpp          PHITransAddr.cpp                    VectorUtils.cpp
DDG.cpp                             LazyBranchProbabilityInfo.cpp       PhiValues.cpp                       VFABIDemangling.cpp
DDGPrinter.cpp                      LazyCallGraph.cpp                   PostDominators.cpp
```

往年竞赛实现 (1/4)

COMP CCRG Open MP

- 实现了两种指令的循环向量化
  - 循环体中的dst[i]=c (c是常数,i是迭代变量)
  - 循环体中的dst[i]=src[i]

```
for(int i=0; i<n; i++)
{
    dst[i]=c;
    //or dst[i]=src[i];
}
```

外卡二等奖

快码加编队	北京航空航天大学	王士举 潘卓然 边一宸 宁然
药枣杞组	北京航空航天大学	高揄扬 鹿煜恒 张杨 张君楷
北关大学第 83 号代表队	南开大学	朱璟钰 孟笑朵 贾宇航 王卓然

https://gitlab.eduxiji.net/educg-group-18973-1895971/20231005520175l-2501/-/blob/master/src/Vectorization.cpp

往年竞赛实现 (2/4)

- 向量化的实现流程
  - 1. 检查循环体中的指令，确保符合向量化的基本条件
    - 循环迭代步为1，迭代变量递增，比较指令为小于（less than）等
  - 2. 识别循环体中可以向量化的语句

    dst[i]=c

    dst[i]=src[i]
  - 3.更新循环的迭代步

    i=i+1 => i=i+4
  - 4.尾循环处理

往年竞赛实现 (3/4)

COMP CCRG Open MP

类型判断条件　dst[i]=c

```
if(ins->isGep() && !ins->getNext()->isGep())
   {
      if(!ins->getNext()->isStore())
      {
         break;
      }
      StoreInstruction* store =
                  dynamic_cast<StoreInstruction*>(ins->getNext());
      if(!store->getOp()[1]->isConst())
      {
         break;
      }
   }
```

往年竞赛实现 (4/4)

- 类型判断条件　dst[i]=src[i]
  - 1. store指令的op1不是常数
  - 2. store指令的op0(即dst)是一条GEP指令的dst
  - 3. store指令的op1(即src)是一条LOAD指令的dst
  - 4. 这条LOAD指令的op1(即src)是一条GEP指令的dst (保证双方都是数组)
  - 5. 这两条GEP指令的index都等于循环变量i

课堂小结

- 什么是向量化、向量机？(熟悉)
- 循环可向量化的判定条件(掌握)
- 向量化与循环变换(掌握)
- 自动向量化的编译实现(了解)
- 向量化仍是未解难题、研究热点

课后作业

- 分析给定循环代码的内层循环能否被向量化
  - 写出依赖关系，并进行判断
  - 在头歌系统提交