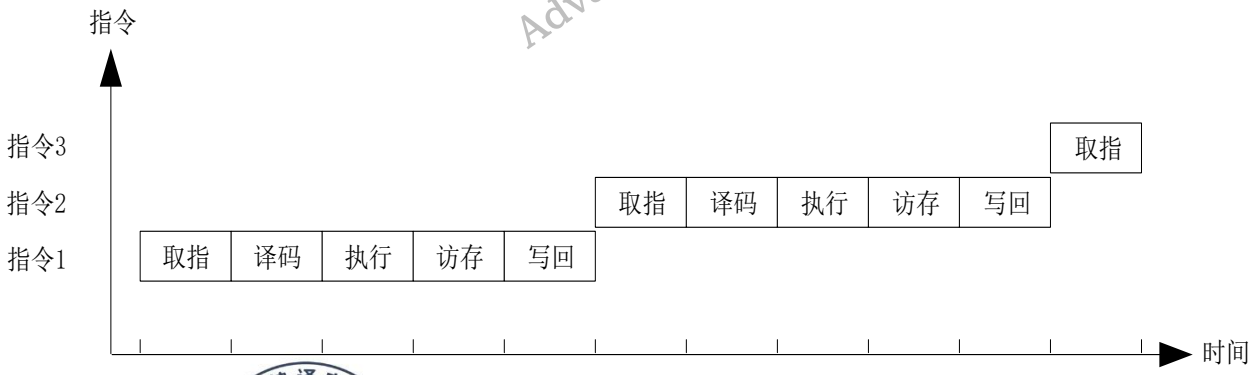
编译论坛

指令级并行I

嘉宾：柴晓楠

现代处理器大都采用了流水线的设计思想，将指令操作划分为更多的阶段，例如一条指令的执行过程可以划分为取指阶段、译码阶段、执行阶段、访存阶段、写回阶段这五个阶段，每个阶段分别在对应的功能部件中完成，利用指令的重叠执行来加速处理速度。



处理器内部的流水线超过5至6级以上就可以称为超级流水线，又叫做深度流水线。超级流水线对提升处理器的主频有帮助，但流水线级数越多，同一时刻重叠执行的指令就越多，可能会导致存在相关性的指令间发生冲突，造成处理器的高频低能。指令间相关性会导致流水线停顿，如下：

```
mul  R1，  R2，  R3   #指令1
add  R3，  R0，  R4   #指令2
sub  R6，  R5，  R7   #指令3
sub  R9，  R8，  R10  #指令4
```

修改后

```
mul  R1，  R2，  R3   #指令1
sub  R6，  R5，  R7   #指令3
sub  R9，  R8，  R10  #指令4
add  R3，  R0，  R4   #指令2
```

结构相关性包括反相关性和输出相关性，反相关性是指指令A在程序中的位置位于指令B之前，指令B中操作数写入的寄存器或存储单元是指令A操作数读的寄存器或存储单元，如果将两个指令调整执行顺序将影响结果的正确性，例如以下指令片段：

```
mul  R1，  R2，  R3   #指令A
sub  R4，  R5，  R1   #指令B
```

反相关性

```
mul  R1，  R2，  R3   #指令A
sub  R4，  R5，  R3   #指令B
```

输出相关性

除数据相关性以及结构相关性之外，还存在某些指令的执行受控于其它指令的情况，即指令间的控制相关，以下面的指令段为例。

```
bne  R1，  R2，  Label     #指令1
add  R3，  R4，  R5     #指令2
mul  R5，  R0，  R6    #指令3
Label: sub R1，  R6，  R6    #指令4
```

指令2和指令3的执行情况受控于指令1的执行结果，当指令1中R1和R2相等时不跳转到Label，此时指令2和指令3会执行；而当指令1中R1和R2不相等时则直接跳转到Label，此时指令2和指令3不会执行。

主流的编译器中会尝试破除指令的控制相关，但是一般仅对最内层循环中的简单控制流结构，对程序中形式复杂的控制相关语句则无能为力，因此优化人员对代码的优化依然是提升程序性能的有效手段。

右图代码段存在控制相关，可以采用控制语句外提的方法优化改写。将循环不变量的判断条件控制语句外提到循环外，从而减少或消除循环内的控制相关。

```
for(int i=0; i<N; i++){
    if(an>10){
        a[i]=c[i];
    }
    else{
        a[i]=d[i];
    }
    m[i]=n[i];
}
```

```
for(int i=0; i<N; i++){
    if(an>10){
        a[i]=c[i];
    }
    else{
        a[i]=d[i];
    }
    m[i]=n[i];
}
```

if外提后

```
if (an > 10){
    for (int i = 0; i < N; i++){
        a[i] = c[i];
        m[i] = n[i];
    }
}
else{
    for (int i = 0; i < N; i++){
        a[i] = d[i];
        m[i] = n[i];
    }
}
```

另一种处理控制相关的方法是控制转换，即将控制相关转为数据相关，通常将这种转换方法称为if转换。

if转换是指将程序中的条件分支语句及相关语句变换为顺序执行的条件赋值语句，从而把控制依赖转换成数据依赖。

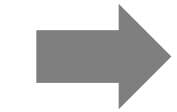
```
for (int i = 0; i < N; i++){
    if (i * 2 > 2){
        c[i] = C0;
    }
    else if (i * 2 < 2){
        c[i] = C1;
    }
    else{
        d[i] = D0;
    }
}
```

if转换后

```
for (int i = 0; i < N; i++){
    c[i] = (i * 2 > 2) ? C0 : c[i];
    c[i] = ((!(i * 2 > 2)) && (i * 2 < 2)) ? C1 : c[i];
    d[i] = ((!(i * 2 > 2)) && (!(i * 2 < 2))) ? D0 :
d[i];
}
```

if转换后，可以对条件语句进行合并，以下面代码为例说明转换后对条件语句进行合并的过程。

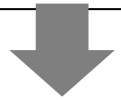
```
for (int i = 0; i < N;
i++){
        if (i * 2 > 2){
        a[i] = C0;
        }
        else{
        a[i] = C1;
        }
}
```



if转换后

```
for (int i = 0; i < N; i++){
        a[i] = (i * 2 > 2) ? C0 : a[i]; //S4
        a[i] = !(i * 2 > 2) ? C1 : a[i]; //S5
}
```

合并后



```
for (int i = 0; i < N; i++){
        a[i] = (i * 2 > 2) ? C0 : C1; //S6
}
```

AdvancedCompiler
Tel: 13839830713

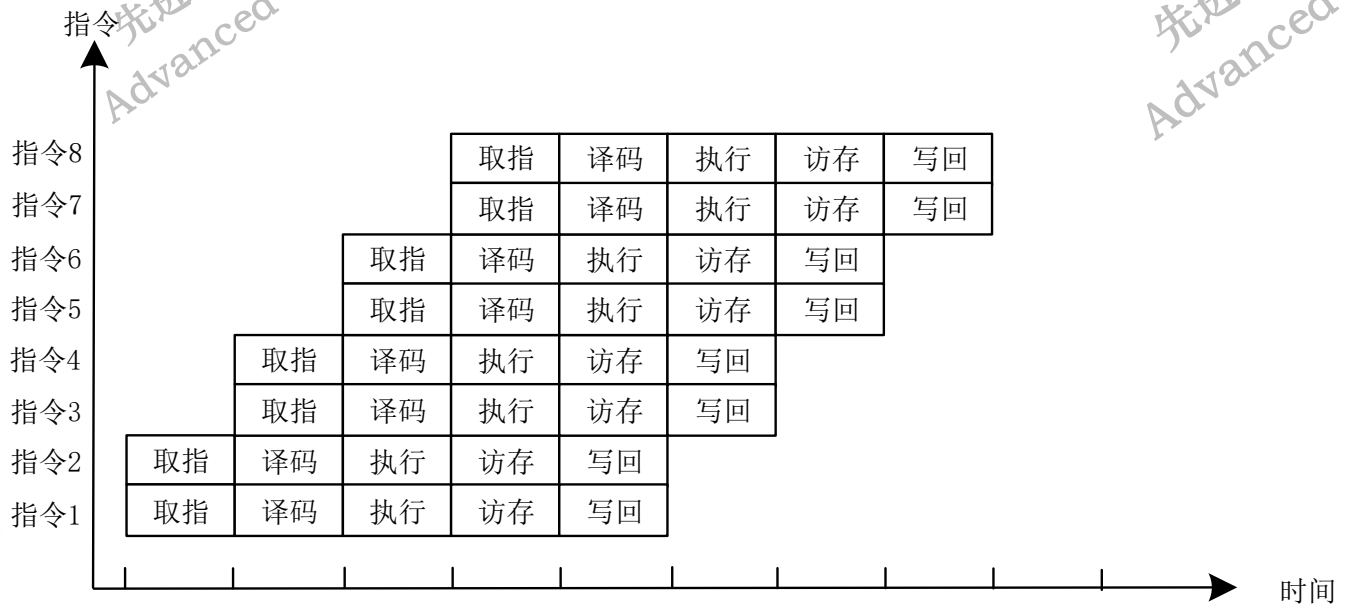
编译论坛

指令级并行II

嘉宾：柴晓楠

指令级并行

程序的执行由一系列指令操作组成，为了节省程序的执行时间，发射单元可以一次发射多条指令，这就是指令的多发射并行。多发射处理器支持指令级并行，每个周期可以发射多条指令，一般为2-4条，这样可以使处理器每个时钟周期的指令数倍增，从而提高处理器的执行速度。多发射处理器的每个发射部件上同样可以进行指令流水，如下图。



指令级并行

指令多发射的方法有超标量（Superscalar）和超长指令字（Very Long Instruction Word,VLIW）两种，它们的不同之处在于并行发射指令的指定时间不一样。

超长指令字在编译阶段由编译器指定并行发射的指令，而超标量在执行阶段由处理器指定并行发射的指令，因此超标量的硬件复杂性更高，而超长指令字硬件复杂性较低。超标量通常会配合乱序执行来提高并行性，下面介绍乱序执行是如何提升超标量处理器性能的。

如果程序中的所有指令都按照既定的顺序执行，那么一旦相邻多条指令不能并行执行，处理器的多发射部件就处于闲置状态，造成硬件资源的浪费。采用乱序执行可以很好的解决该问题，乱序执行就是程序不按照既定的指令顺序执行，在指令间不存在相关性的前提下通过调整指令的执行顺序以提升程序指令的并行性，是提高指令级并行的一种重要方式。

以下面指令段为例说明乱序发射：

```
add  R3，  R2，  R1   #指令1
mul  R1，  R0，  R4   #指令2
mul  R6，  R5，  R7   #指令3
```

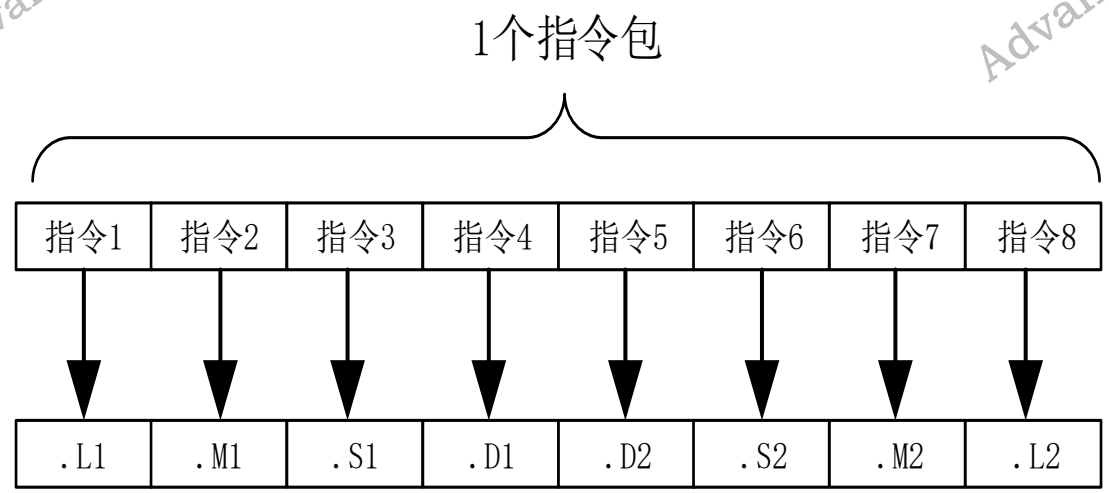
```
add  R3,  R2,  R1  #指令1
mul  R1,  R0,  R4  #指令2
mul  R6,  R5,  R7  #指令3
```

乱序执行调整执行顺序

```
add  R3,  R2,  R1  #指令1
mul  R6,  R5,  R7  #指令3
mul  R1,  R0,  R4  #指令2
```

调整指令2和指令3的顺序后，指令1和指令3之间因为不存在依赖关系，并且使用不同的功能发射部件，可以并行执行。处理器的乱序执行需要在重排序缓存区中分析指令间的相关性，先通过寄存器重命名去除其相关性，然后利用指令调度器调整指令的执行顺序，让更多的指令并行处理。

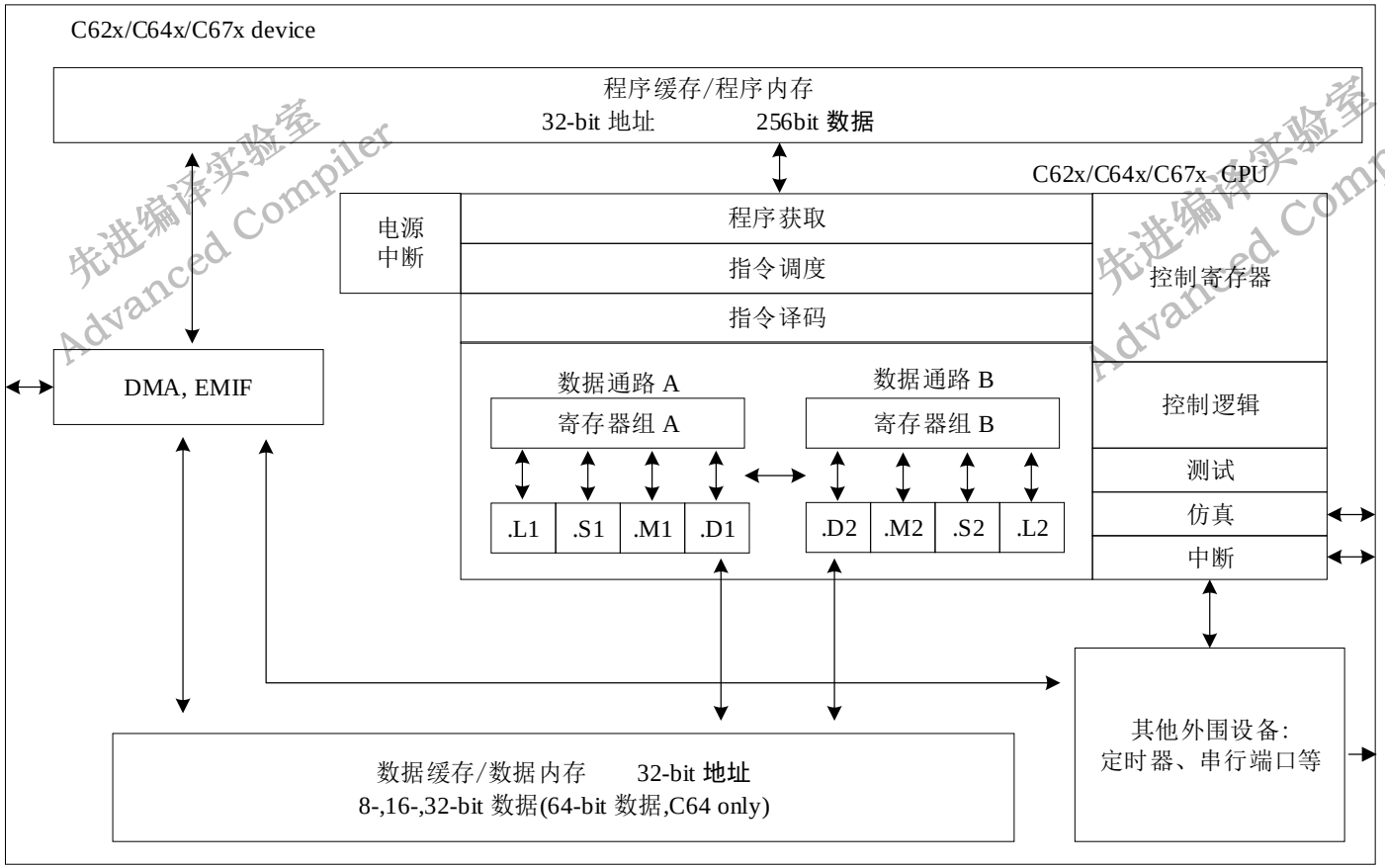
超长指令字处理器的每一条超长指令装有多条常规的指令，并于同一时刻被发射出去。超长指令字结构广泛应用于（Reduced Instruction Set Computer,RISC）精简指令集的处理器上，典型代表为（Digital Signal Process,DSP）数字信号处理器。以德州仪器TI的C6000系列数字信号处理器为例。



超长指令字处理器通常利用编译器指定并行的指令，也可以由优化人员在汇编中指定并行的指令，如下面的汇编代码所示。指令2、指令3、指令4前面的“||”，表示这条指令和上条指令在同一个周期执行，如果没有“||”，则表示这条指令在下一个周期执行。

```
add R1，  R2，  R3   #指令1
|| sub R1，  R2，  R6  #指令2
|| mul R3，  R4，  R5  #指令3
|| and R8，  R6，  R7  #指令4
```

以德州仪器TI的C6000处理器为例进行说明，C6000处理器包含如图7.7所示的四种类型功能部件，分别标记为S、L、D、M。



如下面的指令片段，充分利用了上述八个功能部件，可以有效的提升指令执行效率。在一个时钟周期内将硬件提供的各种功能部件使用起来，就可以充分发挥处理器的指令级并行优势。

```
ADD .S1 A1, A2, A1
‖ADD .S2 A4, A5, A4
‖ MUL .M1 B1, B2, B1
‖ MUL .M2 B4, B5, B4
‖ SUB .L1 A3, A6, A3
‖ SUB .L2 B3, B6, A3
‖ LDDW .D1T1 *A7++, A9:A8
‖ LDDW .D2T2 *B7++, B9:B8
```


实际程序中受限于指令间的相关性以及指令的数量不足，往往不能充分发挥处理器提供的指令级并行优势。此时可以利用循环展开进行优化，发掘程序指令级并行性。以下面这段矩阵乘为例，

```
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++){
        c[i][j] = 0;
        for (k = 0; k < N; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
}
```


循环展开优化后，循环体内指令的条数足够多，编译器就可以在这些指令间进行调度，选择相对较好的并行指令发射组合，因此能够提升程序的性能。以德州仪器TI6678计算平台上矩阵乘汇编为例。

```
.dwpsn      file "../main.c",line 33,column 7,is_stmt,isa 0
        ZERO    .L2     B4              ; |33|
        STW     .D2T2   B4,*+SP(24)     ; |33|
            .dwpsn      file "../main.c",line 33,column
14,is_stmt,isa 0
        CMPLT   .L2     B4,10,B0        ; |33|
  [!B0] BNOP    .S1     $C$L14,5        ; |33|
        ; BRANCHCC OCCURS {$C$L14}      ; |33|
```

指令调度优化后

```
        MVKL    .S2     c,B4
||      ZERO    .L1     A22
||      MVK     .S1     320,A3
||      MV      .L2X    A13,B22         ; |33|
||      MV      .D2     B11,B6
||      MVK     .D1     0xa,A16         ; |33|

        MVKH    .S2     c,B4
||      ADD     .L1X    A3,B11,A18
||      LDDW    .D2T2   *B22,B25:B24
||      MVK     .S1     40,A5

        ADD     .L2X    B4,A22,B5
||      LDDW    .D2T1   *+B22(24),A7:A6

        LDDW    .D2T2   *+B22(8),B9:B8
        LDDW    .D2T1   *+B22(32),A9:A8
        LDDW    .D2T1   *+B22(16),A21:A20
```

AdvancedCompiler
Tel: 13839830713

编译论坛

数据级并行I

嘉宾：柴晓楠

数据级并行

数据级并行是指处理器能够同时处理多条数据的并行方式，大部分处理器采用SIMD向量扩展作为计算加速部件，SIMD扩展部件可以将原来需要多次装载的标量数据一次性装载到向量寄存器中，通过一条向量指令实现对向量寄存器中数据元素的并行处理。使用SIMD方法执行的代码称为向量代码，将标量代码转换成向量代码的过程即为向量化。通常使用两种方式获得向量程序：

（1）由程序设计人员编写向量代码；

（2）借助于编译器的向量化编译自动生成向量代码；

为了让程序获得更好的向量化性能，优化人员需了解程序在向量化过程中面临的问题，这样不仅有利于辅助编译器生成更有效的向量程序，也有利于自行编写出高效的向量程序，将从向量程序的编写和优化两个方面展开具体介绍。示例多以intel SIMD扩展指令改写，为了便于说明，本文使用128位向量寄存器操作指令进行演示。常用的指令按照释义划分如下表所示。

向量操作名称	向量操作指令
向量对齐读内存向量不对齐读内存	_mm_load_ps _mm_loadu_ps
向量对齐写内存向量不对齐写内存	_mm_store_ps _mm_storeu_ps
向量减	_mm_sub_ps
向量乘	_mm_mul_ps
向量混洗	_mm_shuffle_ps
向量加	_mm_add_ps
向量设置	_mm_set_ps
向量收集	_mm_i32gather_ps

向量化的本质是重写程序，以便其同时对多个数据进行相同的操作。编写向量化程序时可以从循环、基本块以及函数等层次发掘数据的并行性，本节将分为以下六个部分展开讨论。

(1) 循环的向量化
(2) 基本块的向量化
(3) 函数的向量化
(4) 分支的向量化
(5) 归约的向量化
(6) 合适的向量长度

当需要计算的数据较多时，直接进行计算需要多个for循环，代码冗长且不好理解。而将循环向量化后可以将多次for循环变成一次计算，较为方便且代价小。

```
for (int i = 0; i < N; i++){
    a[i] = b[i];
}
```

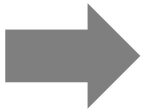
128位向量
指令改写后

```
for (int i = 0; i < (N / 4); i++){
    __m128 second = _mm_load_ps(b + i * 4);
    _mm_store_ps(a + i * 4, second);
}
```

有些循环不适合直接进行向量化，此时可以使用循环变换技术对循环进行变换，例如循环分布可以将可向量化和不可向量化的语句分块，循环剥离可以使得循环内的数据引用变得对齐，循环交换可以使得内层循环的访存变得连续，同时还可以通过将某个外层循环交换至最内层进行向量化。

```
for(i=0；i<N；i++)
        for(j=0；j<N；j++){
                c[i][j]=0；
                for(k=0；k<N；k++)
                c[i][j]=c[i][j]+a[i][k]*b[k][j]；
}
```

面对多层循环时，最内层循环的依赖关系更容易计算清楚，因此一般都选择最内层循环作为向量化的目标。以矩阵乘代码为例。



128位向量
指令改写后

```
for (i = 0; i < N; i++){
                for ( j = 0; j < N; j++){
                        __m128 vc=_mm_set_ps(0,0,0,0);
                        for ( k = 0; k < N; k+=4){
                        __m128 va=_mm_load_ps(&a[i][k]);
                        __m128 vb=_mm_set_ps(b[k+3][j],b[k+2][j],b[k+1][j],b[k][j]);
                        __m128   vab=_mm_mul_ps(va,vb);
                        vc=_mm_add_ps(vab,vc);
                        }
                _mm_store_ps(sum,vc);
                sum[0]=sum[0]+sum[1];
                sum[2]=sum[2]+sum[3];
                c[i][j]=sum[0]+sum[2];
                }
        }
```

当最内层循环存在依赖环向量化后不再满足正确性要求或优化效果不佳时，且无法使用循环交换的前提下，可以考虑对外层循环进行向量化。

外层循环向量化后

```
for (i = 0; i < N; i++){
        for (j = 0; j < N; j += 4){
            __m128 sum = _mm_set_ps(0, 0, 0, 0);
            for (k = 0; k < N; k++) {
                __m128 vb = _mm_load_ps(&b[k][j]);
                __m128 va = _mm_set_ps(a[i][k], a[i][k], a[i][k], a[i][k]);
                __m128 vab = _mm_mul_ps(va, vb);
                sum = _mm_add_ps(vab, sum);
            }
            _mm_store_ps(&c[i][j], sum);
        }
```

面向基本块的向量化又叫直线型向量化，其要求基本块内有足够的并行性，否则会因为有大量的向量和标量之间的转换而影响向量化效果，基本块级向量化方法与循环级向量化方法不同，是指从指令级并行中挖掘数据级并行。

面向基本块的向量化方法中常常提到打包、解包的概念，包是一个同构语句的集合，即语句参数可能不同但可编译的部分是相同的，将多条同构语句组成包的过程称为打包，反之则称为拆包。

同构语句

```
while (i < vlast) {
        sum0 += a[0][0] * v[0][i] + a[0][1] * v[0][i] + a[0][2] * v[0][i];
        sum1 += a[1][0] * v[1][i] + a[1][1] * v[1][i] + a[1][2] * v[1][i];
        sum2 += a[2][0] * v[2][i] + a[2][1] * v[2][i] + a[2][2] * v[2][i];
        i++;
}
```

考虑面向基本块的向量化时，代码中非同构语句更为常见，但非同构语句会影响后续向量化，此时可以将代码中的非同构语句转为同构语句。

```
//非同构
for(i=0;i<N;i+=2){
        C[i]=B[i]*0.5+2;//S1
        C[i+1]=B[i+1] + 1;//S2
}
```

转换为同构语句

```
//同构
for(i=0;i<N;i+=2){
        C[i]=B[i]*0.5+2;
        C[i+1]=B[i+1]*1+1;
}
```

```
//非同构
for(i=0;i<N;i++){
        C[i].real = (A[i].real - B[i].real)*0.5; //S1
        C[i].imag = (A[i].imag + B[i].imag)*0.5;
//S2
}
```

转换为同构语句

```
//同构
for(i=0;i<N;i++){
        C[i].real = (A[i].real +(-1)* B[i].real)*0.5;//S3
        C[i].imag = (A[i].imag + 1*B[i].imag)*0.5;//S4
}
```

```
//非同构
for(i=0;i<N;i+=2){
        C[i]=B[i]*0.5+ a0 + A[i]-D[i];//S1
        C[i+1]=B[i+1]*0.5 + a1;//S2
}
```

转换为同构语句

```
//同构
int a2=a0;
for(i=0;i<N;i+=2){
        a0=a2+A[i]-D[i];//S3
        C[i]=B[i]*0.5+ a0; //S4
        C[i+1]=B[i+1]*0.5 + a1;//S5
}
```

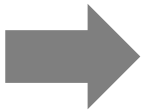
循环展开将迭代间并行转为迭代内并行，可以增加循环内语句的数量，因此优化人员可以在展开后的循环块内实施面向基本块的向量化。

```
for (int i = 0; i < N; i += 6)
{
        c[i + 0] = 0;
        c[i + 1] = 1;
        c[i + 2] = 2;
        c[i + 3] = 3;
        c[i + 4] = 4;
        c[i + 5] = 5;
}
```

基本块向量化之后

```
for(int i=0; i< N; i+=6){
        _mm_storeu_ps(&c[i],z);
        c[i+4]=4;
        c[i+5]=5;
   }
```

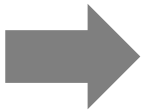
```
for (int i = 0; i < N; i += 24){
      c[i] = 0; c[i + 6] = 0; c[i + 12] = 0; c[i + 18] = 0;
      c[i + 1] = 1; c[i + 1 + 6] = 1; c[i + 1 + 12] = 1; c[i + 1 + 18] = 1;
      c[i + 2] = 2; c[i + 2 + 6] = 2; c[i + 2 + 12] = 2; c[i + 2 + 18] = 2;
      c[i + 3] = 3; c[i + 3 + 6] = 3; c[i + 3 + 12] = 3; c[i + 3 + 18] = 3;
      c[i + 4] = 4; c[i + 4 + 6] = 4; c[i + 4 + 12] = 4; c[i + 4 + 18] = 4;
      c[i + 5] = 5; c[i + 5 + 6] = 5; c[i + 5 + 12] = 5; c[i + 5 + 18] = 5;
  }
```



循环展开4次后
基本块向量化

```
__m128 z = _mm_set_ps(3,2,1,0);
__m128 z1 = _mm_set_ps(1,0,5,4);
__m128 z2 = _mm_set_ps(5,4,3,2);

   for(int i=0; i< N; i+=24){
      _mm_storeu_ps(&c[i],z);
      _mm_storeu_ps(&c[i+4],z1);
      _mm_storeu_ps(&c[i+8],z2);
      _mm_storeu_ps(&c[i+12],z);
      _mm_storeu_ps(&c[i+16],z1);
      _mm_storeu_ps(&c[i+20],z2);
   }
```

```
for (int i = 0; i < N; i += 24){
      c[i] = 0; c[i + 6] = 0; c[i + 12] = 0; c[i + 18] = 0;
      c[i + 1] = 1; c[i + 1 + 6] = 1; c[i + 1 + 12] = 1; c[i + 1 + 18] = 1;
      c[i + 2] = 2; c[i + 2 + 6] = 2; c[i + 2 + 12] = 2; c[i + 2 + 18] = 2;
      c[i + 3] = 3; c[i + 3 + 6] = 3; c[i + 3 + 12] = 3; c[i + 3 + 18] = 3;
      c[i + 4] = 4; c[i + 4 + 6] = 4; c[i + 4 + 12] = 4; c[i + 4 + 18] = 4;
      c[i + 5] = 5; c[i + 5 + 6] = 5; c[i + 5 + 12] = 5; c[i + 5 + 18] = 5;
  }
```



循环展开两次后
基本块向量化

```
__m128 z = _mm_set_ps(3,2,1,0);
__m128 z1 = _mm_set_ps(1,0,5,4);
__m128 z2 = _mm_set_ps(5,4,3,2);
  for(int i=0; i< N; i+=12){
      _mm_storeu_ps(&c[i],z);
      _mm_storeu_ps(&c[i+4],z1);
      _mm_storeu_ps(&c[i+8],z2);
  }
```

不论是面向循环还是面向基本块的向量化方法都是在标量函数内的，即该函数的参数为标量。而函数级向量化是将几个相邻的计算实例合并为一个向量实例，是一种单程序多数据的程序，即函数的参数为向量，返回值也为向量。

```
float fun1(float x, float y) {
    float z = x * y;
    return z;
}
 for (int i = 0; i < N; i++){
        a[i] = fun1(b[i], c[i]);
    }
```

函数向量化

```
__m128 vecfun1(__m128 x, __m128 y) {
    __m128 vz = _mm_mul_ps(x, y);
    return vz;
}
for (int i = 0; i < N; i += 4){
        __m128 vb = _mm_loadu_ps(&b[i]);
        __m128 vc = _mm_loadu_ps(&c[i]);
        __m128 va = vecfun1(vb, vc);
        _mm_storeu_ps(&a[i], va);
    }
```

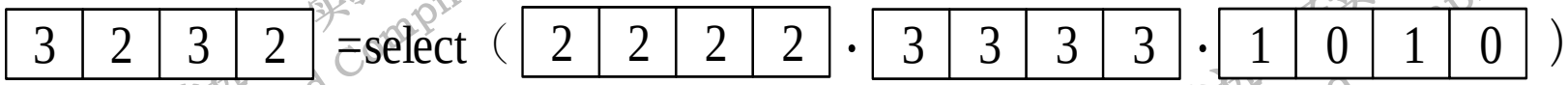
AdvancedCompiler
Tel: 13839830713

编译论坛

数据级并行II

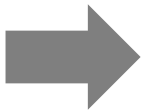
嘉宾：柴晓楠

If转换是向量化控制依赖最常用的方法，可以将控制依赖转换为数据依赖，其需要借助于向量条件选择指令完成向量指令生成。向量选择指令select的格式如图所示：



dst = select (src1，src2， mask)，指令有三个参数，其中mask为掩码，src1和src2是两个源操作数。当掩码位置的值为1时，取src2的值赋给dst，否则将src1的值赋给dst。

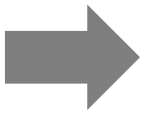
```
for(int i=0; i<N; i++){
        if (a[i] > 0 && a[i] > b[i]) {
                v[i] = w[i];
                if (b[i] > 10)
                        p[i] = m[i];
                else {
                        p[i] = n[i];
                        if (c[i] < 100)
                                x[i] = y[i];
                }
        }

}
```



if转换

```
for (int i = 0; i < N; i++){
        v[i] = (a[i] > 0 && a[i] > b[i]) ? w[i] : v[i];
        p[i] = (a[i] > 0 && a[i] > b[i] && b[i] > 10) ? m[i] : p[i];
        p[i] = (a[i] > 0 && a[i] > b[i] && b[i] <= 10) ? n[i] : p[i];
        x[i] = (a[i] > 0 && a[i] > b[i] && b[i] <= 10 && c[i] < 100) ?
y[i] : x[i];
    }
```

```
for (int i = 0; i < N; i++){
        v[i] = (a[i] > 0 && a[i] > b[i]) ? w[i] : v[i];
        p[i] = (a[i] > 0 && a[i] > b[i] && b[i] > 10) ? m[i] : p[i];
        p[i] = (a[i] > 0 && a[i] > b[i] && b[i] <= 10) ? n[i] : p[i];
        x[i] = (a[i] > 0 && a[i] > b[i] && b[i] <= 10 && c[i] < 100) ?
y[i] : x[i];
    }
```

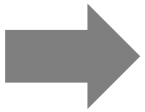


向量化后

```
for (int i = 0; i < N; i += 4){
                    __m128 va = _mm_loadu_ps(&a[i]);
                    __m128 vb = _mm_loadu_ps(&b[i]);
                    __m128 vw = _mm_loadu_ps(&w[i]);
                    __m128 vv = _mm_loadu_ps(&v[i]);
                    __m128 v0 = _mm_set_ps(0, 0, 0, 0);
                    __m128 com1 = _mm_cmpgt_ps(va, v0);
                    __m128 com2 = _mm_cmpgt_ps(va, vb);
                    com1 = _mm_and_ps(com1, com2);
                    vw = _mm_and_ps(vw, com1);
                    vv = _mm_andnot_ps(com1, vv);
                    vv = _mm_add_ps(vw, vv);
                    _mm_storeu_ps(&v[i], vv);
            }
```

当基本块内同构语句条数足够多时，基于if转换的控制流向量化生成的代码并不高效，因为这些语句对应的控制条件相同，不需要再生成条件语句指令，可以直接进行向量化，此种向量化方法称为直接SIMD向量化控制流方法。

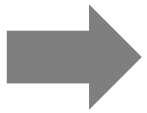
```
for (int i = 0; i < N; i += 2) {
    a[i] = 2 * b[i];
    a[i + 1] = 2 * b[i + 1];
    if (i < N / 2) {
        a[i] += b[i];
        a[i + 1] += b[i + 1];
    }
    else {
        a[i] -= b[i];
        a[i + 1] -= b[i + 1];
    }
    c[i] = a[i] + 2;
    c[i + 1] = a[i + 1] + 2;
}
```



向量化后

```
for (int i = 0; i < N; i += 4) {
    __m128 va = _mm_loadu_ps(&a[i]);
    __m128 vb = _mm_loadu_ps(&b[i]);
    __m128 v2 = _mm_set_ps(2, 2, 2, 2);
    va = _mm_mul_ps(vb, v2);
    _mm_storeu_ps(&a[i], va);
    if (i < N / 2) {
        va = _mm_loadu_ps(&a[i]);
        vb = _mm_loadu_ps(&b[i]);
        va = _mm_add_ps(va, vb);
        _mm_storeu_ps(&a[i], va);
    }else {
        va = _mm_loadu_ps(&a[i]);
        vb = _mm_loadu_ps(&b[i]);
        va = _mm_sub_ps(va, vb);
        _mm_storeu_ps(&a[i], va);
    }
    va = _mm_loadu_ps(&a[i]);
    __m128 vc = _mm_add_ps(va, v2);
    _mm_storeu_ps(&c[i], vc);
}
```

```
for (int i = 0; i < N; i += 4) {
        __m128 va = _mm_loadu_ps(&a[i]);
        __m128 vb = _mm_loadu_ps(&b[i]);
        __m128 v2 = _mm_set_ps(2, 2, 2, 2);
        va = _mm_mul_ps(vb, v2);
        _mm_storeu_ps(&a[i], va);
        if (i < N / 2) {
            va = _mm_loadu_ps(&a[i]);
            vb = _mm_loadu_ps(&b[i]);
            va = _mm_add_ps(va, vb);
            _mm_storeu_ps(&a[i], va);
        }
        else {
            va = _mm_loadu_ps(&a[i]);
            vb = _mm_loadu_ps(&b[i]);
            va = _mm_sub_ps(va, vb);
            _mm_storeu_ps(&a[i], va);
        }
        va = _mm_loadu_ps(&a[i]);
        __m128 vc = _mm_add_ps(va, v2);
        _mm_storeu_ps(&c[i], vc);
    }
```



进一步精简

```
for (int i = 0; i < N; i += 4) {
        __m128 vb =
_mm_loadu_ps(&b[i]);
        __m128 v2 = _mm_set_ps(2, 2, 2,
2);
        __m128 va = _mm_mul_ps(vb, v2);
        if (i < N / 2) {
            va = _mm_add_ps(va, vb);
        }
        else {
            va = _mm_sub_ps(va, vb);
        }
        __m128 vc = _mm_add_ps(va, v2);
        _mm_storeu_ps(&a[i], va);
        _mm_storeu_ps(&c[i], vc);
    }
```

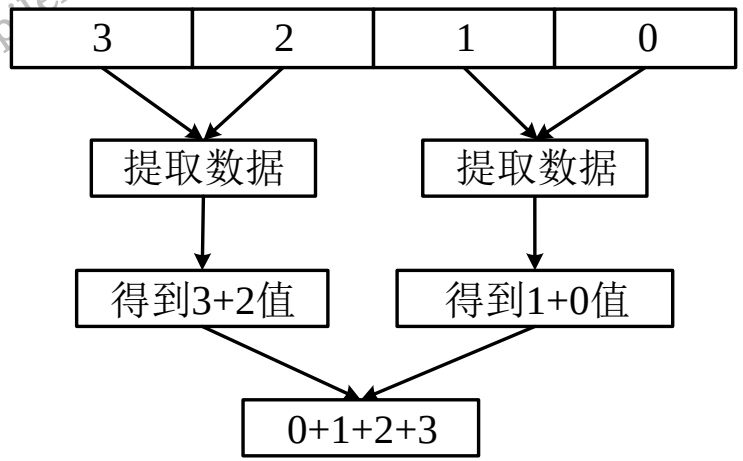
归约操作是指将多个元素归并为单个元素的过程，该操作把向量中的多个元素归约为一个元素，常见的归约操作包括归约加、归约乘等。

```
for (j = 0; j < N; j++) {
    sum = sum + a[j];
}
```

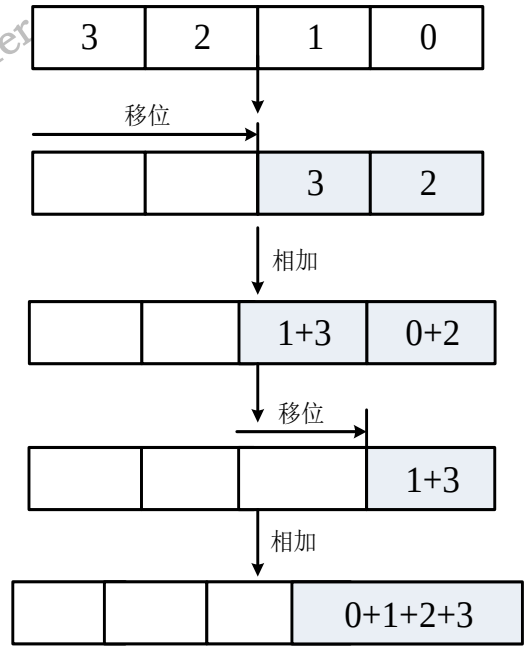
向量化后

```
for (i = 0; i < N / 4; i++) {
    ymm0 = _mm_load_ps(a + 4 * i);
    ymm1 = _mm_set_ps(0, 0, 0, 0);
    ymm2 = _mm_hadd_ps(ymm0, ymm1);
    ymm3 = _mm_hadd_ps(ymm2, ymm1);
    _mm_storeu_ps(s, ymm3);
    sum = s[0] + sum;
}
```

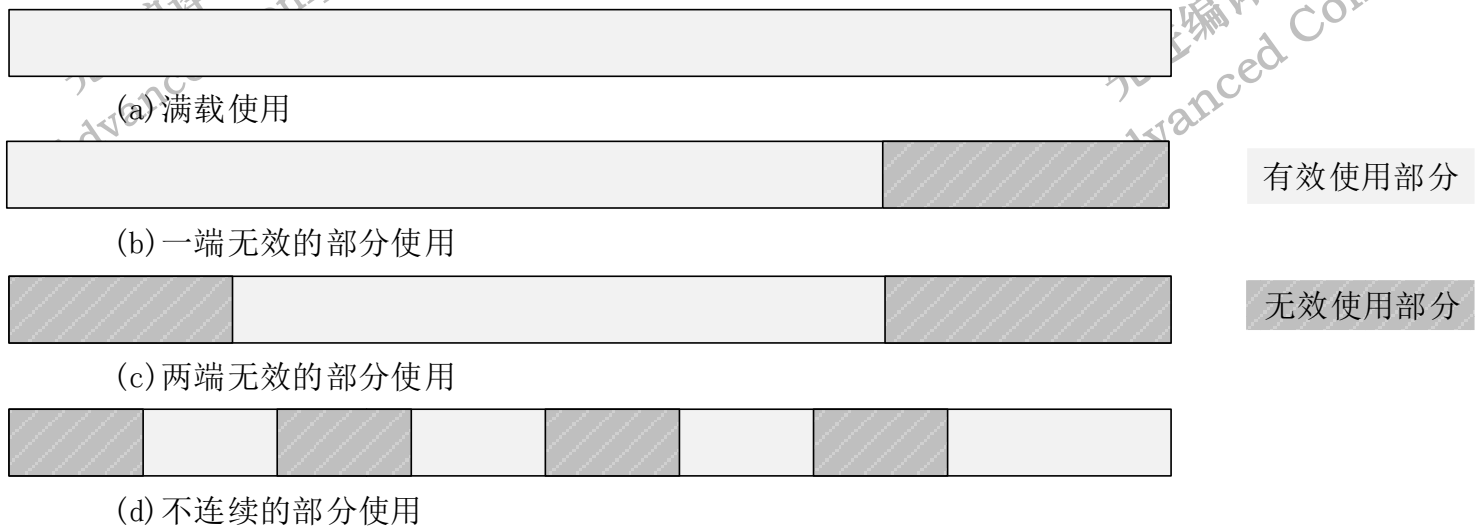
以上面的代码段为例，将向量寄存器槽位中的4个数据相加，利用提取指令实现的上述归约需要进行三次提取，然后进行三次加法。



此外还可以利用向量移位指令实现归约加法，实现原理如图所示。此种方法通过两次移位、两次相加获得最终结果。



当前大多数向量寄存器在使用时为一个不可拆分的整体，即向量寄存器中的每个数据都是有效的。但语句中的数据并行性不足时，需要向量寄存器的部分使用，即向量寄存器中的某些槽位为有效数据，其它槽位为无效数据。向量寄存器有四种使用方式，分别为满载使用、一端无效的部分使用、两端无效的部分使用、不连续的部分使用。



其中寄存器满载使用的情况也可称为程序充分向量化，而部分使用的情况可称为程序不充分向量化。

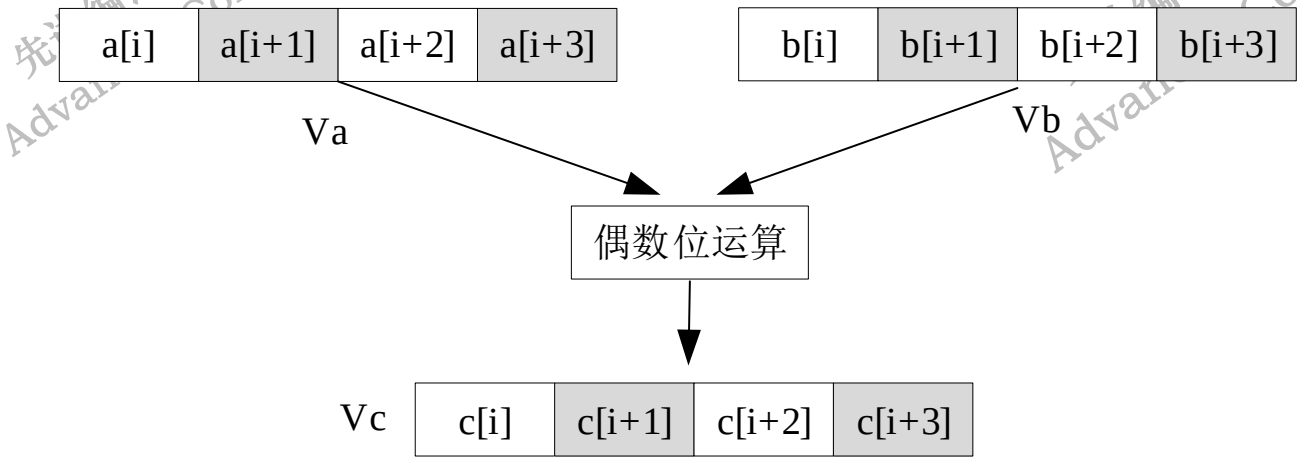
1

一是当平台没有向量重组指令或者向量重组指令的功能较弱时，如果强制将不连续的访存数据组成向量可能导致向量化没有收益，而不充分向量化不需考虑平台是否支持向量重组指令，同样可以生成向量程序。

2

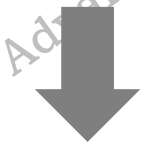
二是向量重组指令的代价过大而导致向量化没有效果，即使用充分向量化效果不如使用不充分向量化效果。

常用的不充分向量化方法分为三类，分别为掩码内存读写方法、插入/提取方法以及加宽向量访存方法。首先介绍掩码内存读写方法，以下图所示的语句S1:c[2i]=a[2i] + b[2i]为例，load指令从内存中连续地加载数据到向量寄存器Va和Vb中，其中的偶数位是有效槽位，奇数位是无效的槽位 。



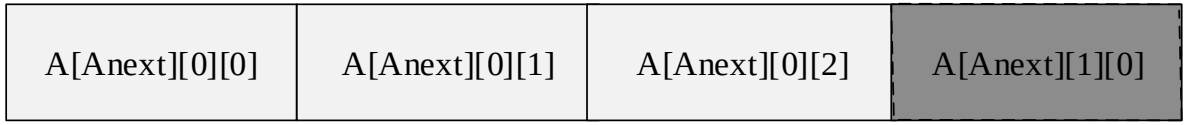
不充分向量化方法的代码生成需要从三个方面进行考虑，首先在读内存时需要标记出有效槽位和无效槽位，然后在运算时需要将参与运算的向量寄存器槽位相对应，最后再将结果写入内存时需要避免将无效槽位的值写入内存。在申威平台以下面的基本块代码为例说明如何生成不充分向量化代码。

```
w[col][0] =
A[Anext][0][0]*v[i][0] ;
w[col][1] =
A[Anext][0][1]*v[i][0] ;
w[col][2] =
A[Anext][0][2]*v[i][0] ;
```

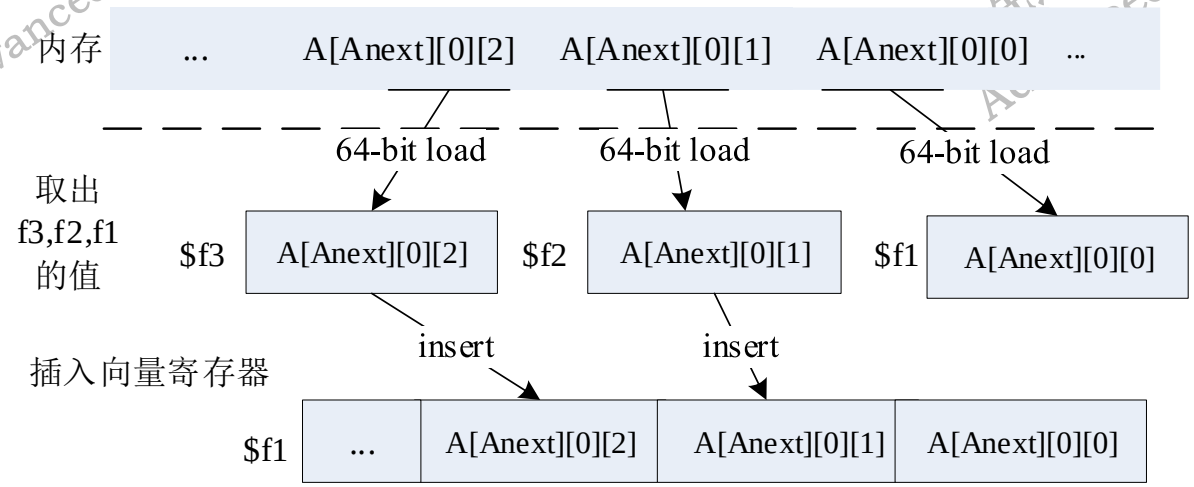


```
Loade  $f1,  v[i][0]       //取值和赋值
Vload  $f2,  A[Anext][0][0] //向量取
Vmul   $f1,  $f2,  $f3     //向量乘
Vstore $f3,  w[col][0]     //向量存
```

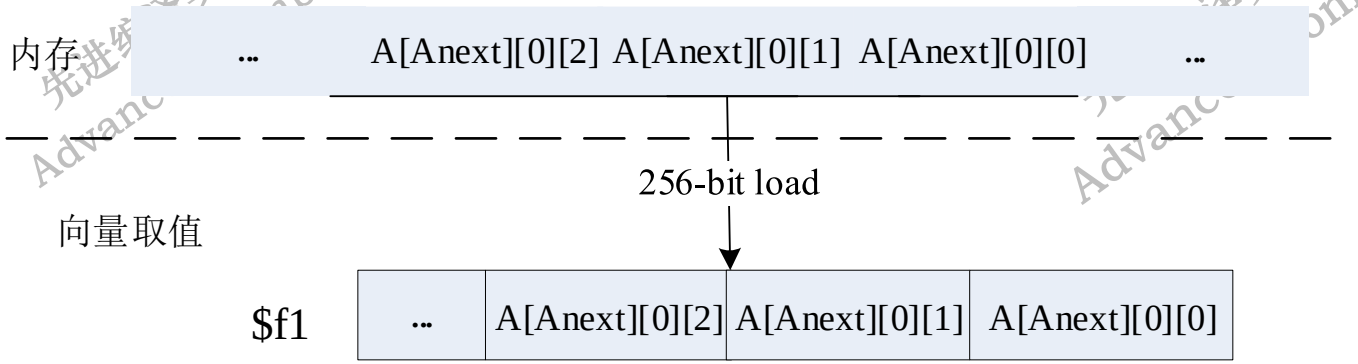
因为读写内存操作会访问到原程序没有涉及到的值w[col+1][0]和A[Anext][1][0]，如下图所示，这可能导致程序结果不正确和内存溢出，并且在对无效槽位进行乘法运算时可能引发异常，此时可以采用插入/提取的方法对程序进行向量化。



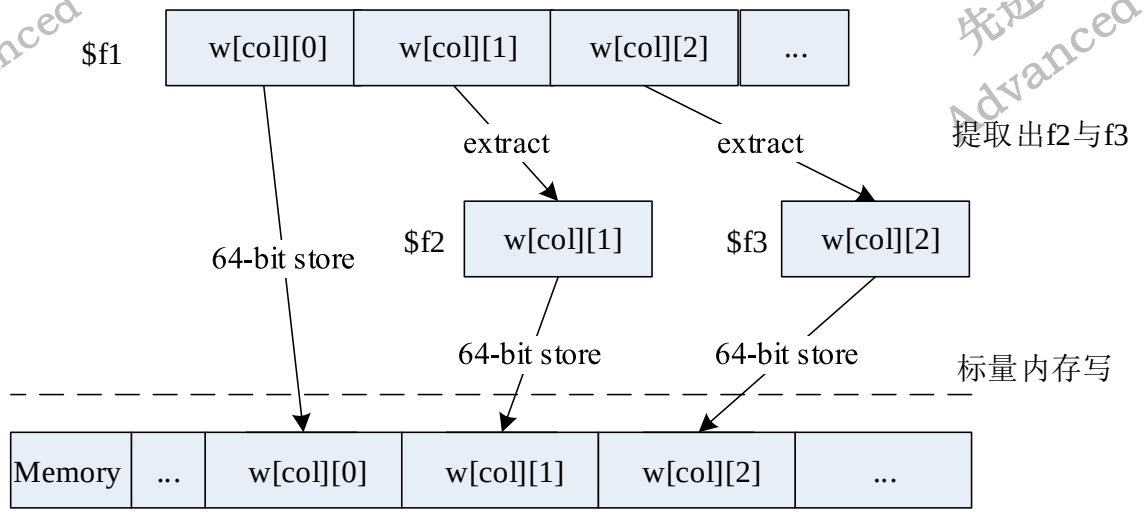
利用插入指令实现时首先通过多条标量内存读操作将数据存放到标量寄存器中，然后将数据分别插入到向量寄存器中，如下图所示。一些平台支持向量插入指令，如Intel的SSE指令集可利用一条插入读指令“__mm_set_ps(f[k][0],f[k][1], f[k][2],0)”，就可以将f[k][0]、f[k][1] 和f[k][2]的值加载到向量寄存器中。



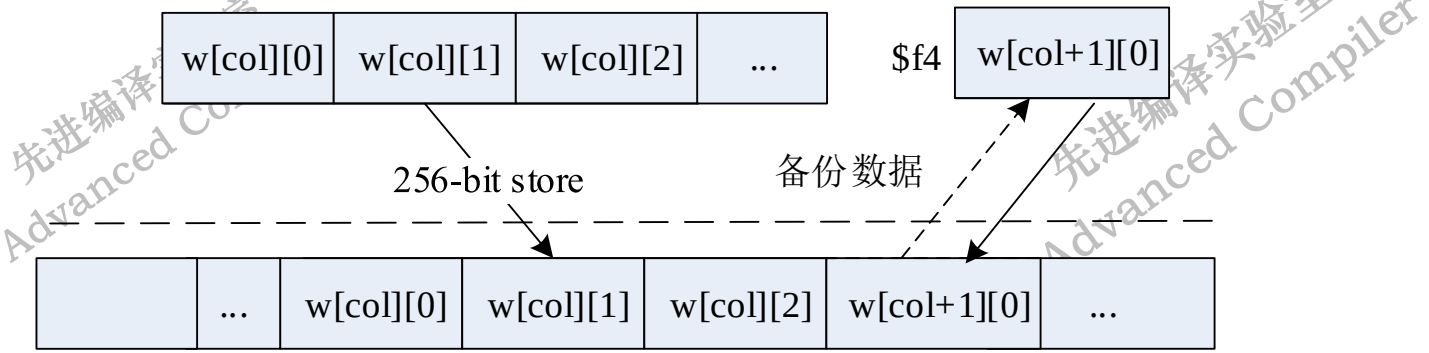
同样可以利用加宽向量访存方法实现向量化，如图所示，如果需要的值在内存中是连续的，那么一条加宽的向量读指令就可以将其加载到向量寄存器中实现部分向量读操作。



代码中的向量写操作可以使用提取指令实现，如图所示。利用提取指令将数据从向量寄存器中提取出来，然后利用标量内存写指令实现。AVX2指令集中提供有掩码内存写指令，一条掩码内存写指令“vmaskmovpd ymm3，mask，x[k][0]”就可以将x[k][0]，x[k][1]和x[k][2]的值写入内存。

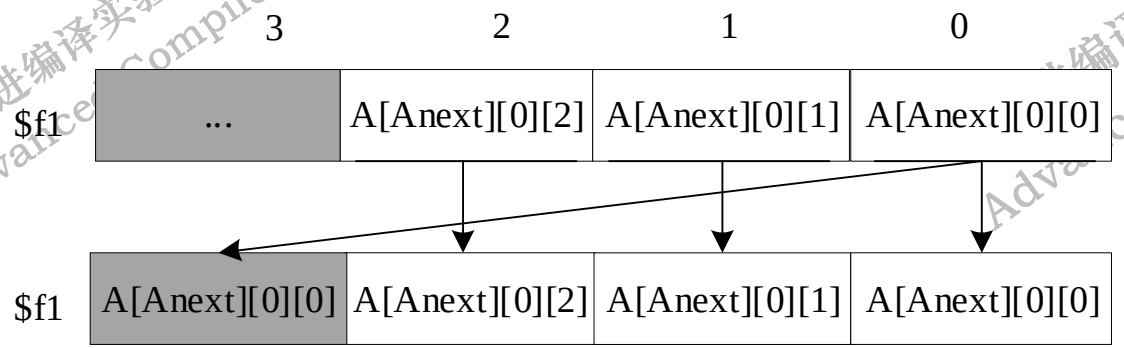


与加宽向量读指令相似，加宽向量写指令可以用于部分向量写内存操作，然而该操作不仅需要避免访存溢出，同时还要避免对内存造成误写，可以在尾部添加一些内存空间避免内存访问溢出。



示例中需要避免w[col+1][0]的值在部分内存写时被改变，使用备份和恢复机制可以更正被改变的值，首先将w[col+1][0]的值读到标量寄存器$f4中，然后利用一个加宽的向量存操作将结果写入内存，最后再利用一个标量存操作恢复w[col+1][0]的值。

在原程序中，无效槽位的数据不需要任何计算。因此需要将无效槽位填充一些数据以避免无效槽位引入算术异常。利用插入指令或者混洗指令，将任意一个有效槽位中的数据填充到无效槽位，来避免无效槽位引入的算术异常。



使用槽位　　未用槽位

Vinsert3 $f1,$f1,$f1　// Inserttion

利用加宽向量访存生成的不充分向量化汇编代码如下所示：

```
Loade  $f1，  v[i][0]          //取值和赋值
Vload  $f2，  A[Anext][0][0]   // 向量取
Vinsert3 $f2，  $f2，  $f2          //插入值
Vmul   $f1，  $f2，  $f3            // 向量乘
Load   $f4，  w[col+1][0]      // 标量取
Vstore  $f3，  w[col][0]       // 向量存
Store   $f4，  w[col+1][0]     // 标量存
```


与前面直接向量化的错误代码相比，添加了一条插入指令以保证部分向量运算时不会引起算术异常，然后添加了一条标量读和标量写指令实现部分向量操作。

AdvancedCompiler
Tel: 13839830713

编译论坛

数据级并行III

嘉宾：柴晓楠

本节介绍在已经改写的向量化程序上如何更进一步的提升性能，主要包含以下六点内容：

（1）不对齐访存；
（2）不连续访存；
（3）向量重用；
（4）向量运算融合；
（5）循环完全展开；
（6）全局不变量合并；

访存对齐性是影响向量程序性能的重要因素，内存对齐访问是指内存地址A 对n求余等于0，其中n为访存数据的字节数。如果向量访存是不对齐的，与对齐的向量访存相比，需要额外的开销才能实现数据的存储操作。在编写向量程序时，应尽量使用对齐的访存指令。然而现实程序中更多的是不对齐访存，优化人员可以借助程序变换的方法，将不对齐访存调整为对齐以提升向量程序的性能。


```
for (int i = 0; i < 100; i++)
        B[i + 1] = A[i + 1] + C;
```

循环剥离后

```
for (int i = 0; i < 3; i++)
        B[i + 1] = A[i + 1] + C;
for (int i = 3; i < 99; i++)
        B[i + 1] = A[i + 1] + C;
for (int i = 99; i < 100; i++)
        B[i + 1] = A[i + 1] + C;
```

先进编译实验室
Advanced Compiler

当多维数组的最低维长度不是向量长度的整数倍时，难以判断访存的对齐性，此时一般会使用非对齐访问指令保证程序的正确性。这个问题可以使用数组填充来解决，即当数组最低维长度不是向量化因子的整数倍时，通过增加数组最低维的长度，使得向量化的时候能够统一按照对齐的方法进行向量化装载或者存储。

```
for (int i = 0; i < 1335; i++)
        for (int j = 0; j < 1335; j++)
                A[i][j] = A[i][j] * 2;
```

数组填充后

```
for (int i = 0; i < 1335; i++)
        for (int j = 0; j < 1336; j++)
                A[i][j] = A[i][j] * 2;
```

现实程序中数据访存不对齐的情况更多，如果程序实际是不对齐访存，而写为对齐访存，那么就会造成程序运行错误。因此，如果优化人员不能确定访存的对齐性时，需要使用不对齐指令进行访存，以保证程序的正确运行。


```
for (int i = 0; i < N; i++)
        C[i] = A[i + 2];
```

不对齐指令改写

```
for (int i = 0; i < N; i += 4) {
        __m128 va = _mm_loadu_ps(&A[i + 2]);
        _mm_storeu_ps(&C[i], va);
}
```

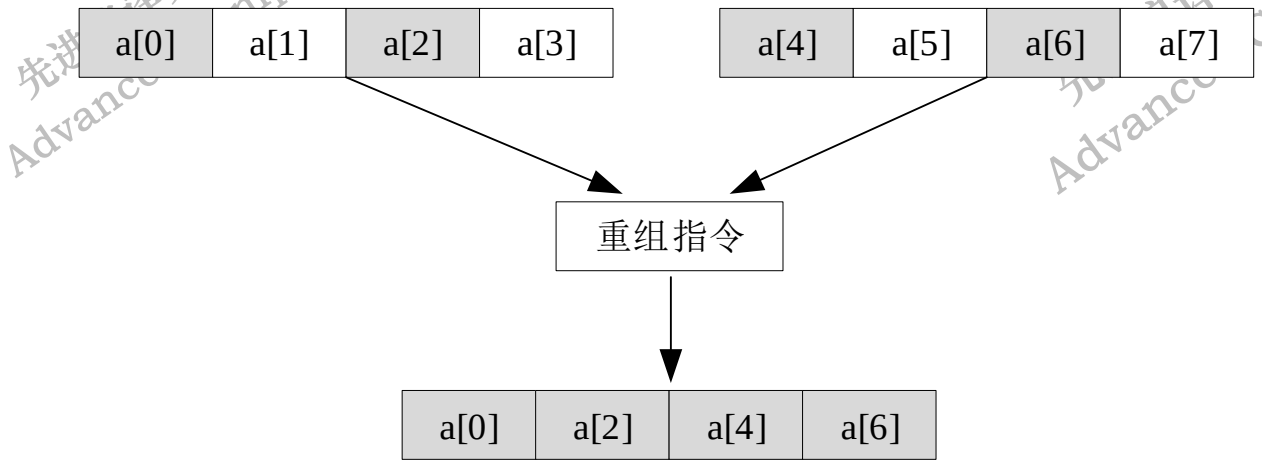
连续的向量访存不仅可以提高向量访存指令的效率，还可以提高向量寄存器中有效数据的比率。但多数计算都不是理想的连续访存情况，本节将介绍如何在不改变引用顺序和数据布局的情况下，利用处理器提供的向量指令实现不连续访存程序的向量化。

```
for (i = 0; i < N; i += 4) {
    a[i] = i + 2;
}
for (i = 0; i < N; i++) {
    sum += a[i] + sum;
}
```

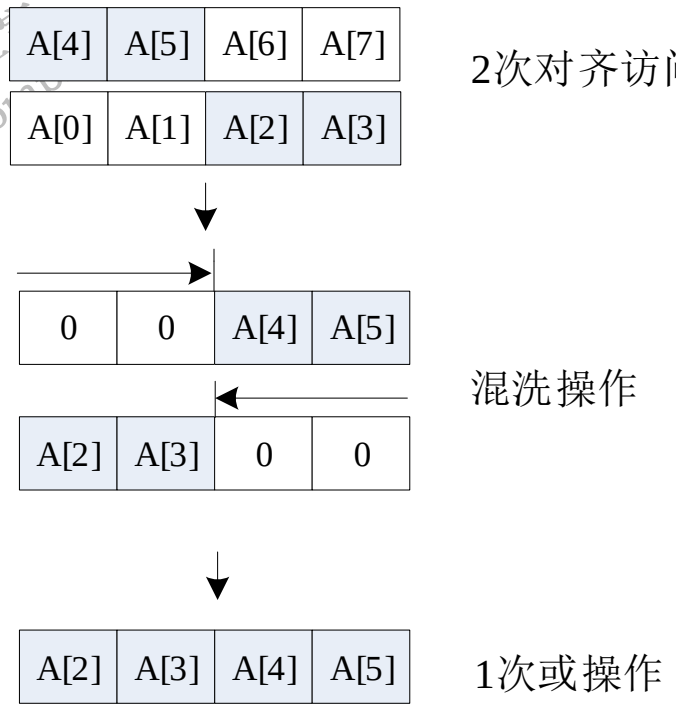
不连续访存改写

```
__m128 gather = _mm_set_ps(0, 0, 0, 0);
__m128 ymm1, ymm2, ymm3, ymm4;
  for (i = 0; i < N / 4; i++) {
    ymm1 = _mm_i32gather_ps(a + 4 * i, gather, sizeof(float));
    //set the first data in new ymm1
    _mm_storeu_ps(b + i, ymm1);
  }
for (i = 0; i < N / 4 / 4; i++) {
    ymm1 = _mm_load_ps(b + 4 * i);
    ymm2 = _mm_set_ps(0, 0, 0, 0);
    ymm3 = _mm_hadd_ps(ymm1, ymm2);
    ymm4 = _mm_hadd_ps(ymm3, ymm2);
    _mm_storeu_ps(s, ymm4);
    sum = s[0] + sum;
  }
```

然而并不是所有的平台都支持聚集和分散指令，对于一些不连续访存、但是访问内存有规律的程序可以利用向量重组实现不连续访存，向量重组是指当目标向量中的所有元素不在同一个向量中时，通过多个向量之间的重新组合得到目标向量。

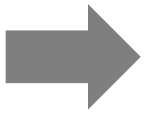


对于上一节不对齐访问的优化示例，并不是所有的平台都支持不对齐的访存指令，当平台不支持不对齐访存指令时，可以借助于数据重组实现不对齐访存。如通过两次对齐访存，然后将数据移位或者重组。



使用Intel处理器的SIMD扩展指令集，进一步如何说明利用向量重组实现不连续访存程序的向量化。可以通过混洗指令simd_vshff将数据重组。

```
for (int i = 0; i < N - 4; i += 2) {
        x11 = y[idx1 + i];
        x12 = y[idx1 + i + 1];
        x21 = y[idx2 + i];
        x22 = y[idx2 + i + 1];
        x[idx3 + i] = x11 * x21 - x12 *
x22;
        x[idx3 + i + 1] = x12 * x21 + x11
* x22;
    }
```



混洗指令改写

```
for(int i=0; i< N-idx2; i+=8){
        vx1 = _mm_loadu_ps(y+i+idx1);
        vx2 = _mm_loadu_ps(y+i+idx1+4);
        vxj1 = _mm_shuffle_ps(vx1,vx2,_MM_SHUFFLE(2,0,2,0));
        vxo1 = _mm_shuffle_ps(vx1,vx2,_MM_SHUFFLE(3,1,3,1));
        vx3 = _mm_loadu_ps(y+i+idx2);
        vx4 = _mm_loadu_ps(y+i+idx2+4);
        vxj2 = _mm_shuffle_ps(vx3,vx4,_MM_SHUFFLE(2,0,2,0));
        vxo2 = _mm_shuffle_ps(vx3,vx4,_MM_SHUFFLE(3,1,3,1));
        vy1 = _mm_mul_ps(vxj1,vxj2);
        vy2 = _mm_mul_ps(vxo1,vxo2);
        vy3 = _mm_mul_ps(vxj1,vxo2);
        vy4 = _mm_mul_ps(vxj2,vxo1);
        tp1 = _mm_sub_ps(vy1,vy2);
        tp2 = _mm_add_ps(vy3,vy4);
        tp3 = _mm_shuffle_ps(tp1,tp2,_MM_SHUFFLE(1,0,1,0));
        tp4 = _mm_shuffle_ps(tp1,tp2,_MM_SHUFFLE(3,2,3,2));
        tp3 = _mm_shuffle_ps(tp3,tp3,_MM_SHUFFLE(3,1,2,0));
        tp4 = _mm_shuffle_ps(tp4,tp4,_MM_SHUFFLE(3,1,2,0));
        _mm_storeu_ps(x+i+idx3,tp3);
        _mm_storeu_ps(x+i+idx3+4,tp4);
    }
```

混洗指令不仅可以用作数据整理，当掩码为零的时候也可以用作数据广播填充。如下：

```
for(int i=0;i<N;i+=4){
        __m128 vy=_mm_loadu_ps(y+i);
        __m128 vx=_mm_shuffle_ps(vy,vy,0);
        _mm_storeu_ps(&x[i],vx);
        }
```

通过以上使用混洗指令，实现了float类型的数据转换成四个槽位的相同数据，作用类似于向量取值指令。在编写程序时，可以根据实际情况使用不同的方式进行数据填充转换。

不对齐访存代码可以利用向量重用进一步优化。以下面代码为例，在使用对齐指令对C赋值时，循环体内对于不对齐数组的向量访存需要两条对齐的向量访存和一条拼接指令，可以将其中一次访存指令重用。

```
for (int i = 0; i < N; i += 4) {
    __m128 va = _mm_loadu_ps(&A[i + 2]);
    _mm_storeu_ps(&C[i], va);                }
```

使用对齐
指令改写

```
__m128 va1 = _mm_load_ps(a);
  for (int i = 0; i < N; i += 4){
     __m128 va2 = _mm_load_ps(&a[i + 4]);
     __m128 V1 = _mm_shuffle_ps(va1, va2, _MM_SHUFFLE(1, 0, 3, 2));
     _mm_store_ps(&c[i], V1);
     va1 = va2;
  }
```

向量寄存器的重用可以减少或者去除访存的需求。向量寄存器含有多个数据项，因此向量寄存器的重用包括全部数据项的重用和部分数据项的重用，向量寄存器的完全重用是最理想的情况，将避免后续所有的向量访存，如下：

```
for (int i = 0; i < N;
i++){
      a[i] = b[i] + c[i];
      d[i] = b[i] - c[i];
   }
```

使用向量指令改写

```
for (int i = 0; i < N; i += 4){
      __m128 vb = _mm_loadu_ps(b + i);
      __m128 vc = _mm_loadu_ps(c + i);
      __m128 va = _mm_add_ps(vb, vc);
      _mm_storeu_ps(a + i, va);
      vb = _mm_loadu_ps(b + i);
      vc = _mm_loadu_ps(c + i);
      __m128 vd = _mm_sub_ps(vb, vc);
      _mm_storeu_ps(d + i, vd);
   }
```

向量重用后

```
for (int i = 0; i < N; i += 4){
      __m128 vb = _mm_loadu_ps(b + i);
      __m128 vc = _mm_loadu_ps(c + i);
      __m128 va = _mm_add_ps(vb, vc);
      _mm_storeu_ps(a + i, va);
      __m128 vd = _mm_sub_ps(vb, vc);
      _mm_storeu_ps(d + i, vd);
   }
```

```
for (int i = 0; i < N; i++){
    a[i] = d[i] + b[i] * c[i];
}
```

使用向量指令改写

```
for (int i = 0; i < N; i += 4){
    __m128 v1 = _mm_loadu_ps(&b[i]);
    __m128 v2 = _mm_loadu_ps(&c[i]);
    __m128 v3 = _mm_mul_ps(v1, v2);
    __m128 v4 = _mm_loadu_ps(&d[i]);
    __m128 v5 = _mm_add_ps(v4, v3);
    _mm_storeu_ps(&a[i], v5);
}
```

向量运算融合就是将多条向量运算指令合并为一条向量运算指令，以提高向量程序的执行性能。

乘加指令改写后

```
for (int i = 0; i < N; i += 4){
    __m128 v1 = _mm_load_ps(&b[i]);
    __m128 v2 = _mm_load_ps(&c[i]);
    __m128 v3 = _mm_load_ps(&d[i]);
    v3 = _mm_fmadd_ps(v1, v2, v3);
    _mm_storeu_ps(&a[i], v3);
}
```

循环展开不仅可以提高程序的指令级并行还可以提高寄存器重用，优化人员可以在循环被向量化后继续对循环进行展开，相当于在发掘完程序数据级并行的基础上，进一步发掘程序的指令级并行，同时提升向量寄存器的重用。

假设使用的向量寄存器长度为128位，一次能够处理4个float数据，向量化后原来的循环仅需要两次迭代，因此优化人员在展开时可以将循环完全展开，去掉循环控制结构，如下：

```
for (int i = 0; i < 8; i++)
        B[i] = A[i];
```

循环完全展开

```
__m128 v1 = _mm_load_ps(&A[0]);
_mm_store_ps(&B[0], v1);
v1 = _mm_load_ps(&A[4]);
_mm_store_ps(&B[4], v1);
```

循环不变量是指该变量的值在循环内不发生变化。向量化的过程中会引入很多向量类型的循环不变量，如果未将向量类型的循环不变量移到循环外，将影响程序的向量化性能。


```
for (int i = 0; i < N; i++)
        a[i] = C * b[i];
```

不变量外提后

```
V1 = _mm_set_ps(C, C, C, C);
  for (int i = 0; i < N; i += 4){
     V2 = _mm_load_ps(&b[i]);
     V3 = _mm_mul_ps(V1, V2);
     _mm_store_ps(&a[i], V3);
  }
```

如果其它循环在向量化过程中也产生了同样的向量常数，可以在过程内甚至程序内进行更大范围的常数合并。

```
for (int i = 0; i < N; i++)
        a[i] = C * b[i];
for (int i = 0; i < N; i++)
        d[i] = C + x[i];
```

向量改写

```
V1 = _mm_set_ps(C, C, C, C);
  for (int i = 0; i < N; i += 4){
     V2 = _mm_load_ps(&b[i]);
     V3 = _mm_mul_ps(V1, V2);
     _mm_store_ps(&a[i], V3);
  }
  for (int i = 0; i < N; i += 4){
     V1 = _mm_set_ps(C, C, C, C);
     V2 = _mm_load_ps(&x[i]);
     V3 = _mm_add_ps(V1, V2);
     _mm_store_ps(&d[i], V3);
  }
```

不变量外提

```
V1 = _mm_set_ps(C, C, C, C);
  for (int i = 0; i < N; i += 4){
     V2 = _mm_load_ps(&b[i]);
     V3 = _mm_mul_ps(V1, V2);
     _mm_store_ps(&a[i], V3);
  }
  for (int i = 0; i < N; i += 4){
     V1 = _mm_set_ps(C, C, C, C);
     V2 = _mm_load_ps(&x[i]);
     V3 = _mm_add_ps(V1, V2);
     _mm_store_ps(&d[i], V3);
  }
```

AdvancedCompiler
Tel: 13839830713