四、部分写规则

$$\exists i \quad (V\_store[i].B\_saclars \cap simd\_store.B\_saclars'') \neq simd\_store.B\_saclars''$$
$$\Rightarrow V\_store[i].B\_saclars = simd\_store(V\_store[i].S\_vector)$$
$$\Rightarrow S\_vector'' = simd\_store(B\_saclars'')$$
$$\Rightarrow V\_store[i].delete(\cdot)$$
$$\Rightarrow V\_load.add((S\_vector'', B\_saclars''))$$
$$\Rightarrow V\_store.add((S\_vector'', B\_saclars'')).$$

其中,*simd_load*(*S_vector*′,*B_scalars*′)表示需要生成数据打包指令的二元组,*simd_store*(*S_vector*″,*B_scalars*″)表示需要生成数据拆包指令二元组,*V_load*[·]表示读缓存队列,*V_store*[·]表示写缓存队列.

在处理 IF 基本块 B_{IF} 之前,假定 B_{IF} 中所用的变量都在它的前驱节点中有定义并且与具有相同的打包方式,那么 B_{IF} 中只存在向量复用的情况,只使用向量复用规则即可完成对它的处理,图 7 是块间向量复用优化算法的伪代码.

```
01 功能:块间向量复用优化
02 输入:控制流图G=CFG(B,E)
03 func Between_Blocks_Vector_Reuse(G)
04     Pack_queue[]=Null;              //需要打包的数据队列
05     V_store=V_load=NULL;            //初始化 V_load,V_store
06       while (n<-breadth-first search  of G)
07         Pack_queue[]<-SLP(n.B);
08         reuse_vectors[]=Null;
09         for i=0 to Pack_queue do
10             reuse_vectors[]<-gen simd_load or simd_store of Pack_queue[i] with reuse rules;
11             if (reuse_vectors[])  then
12                 gen other simd_* with reuse_vectors;
13             end if
14         end for
15     end while
16     if (V_store) then
17         for all (S_vector,B_vector) in V_store do
18             generate V_scalsr=simd_store(B_scalars);
19         end for
20     end if
```

Fig.7 A vector reuse algorithm of across basic block

图 7 块间向量复用算法

块间向量复用优化算法的基本流程是:使用宽度优先遍历对控制流图进行遍历,对控制流图中对应的每个节点使用 SLP 算法进行向量化,获得需要打包的 *Pack_queue* 队列,按照块间向量复用规则指导 *Pack_queue* 队列中 *simd_load* 和 *simd_store* 的生成,并返回向量复用队列 *reuse_vectors*[·]={(*R_vector*,*B_scalars*)},该队列保存着 *B_scalars* 所能复用向量 *R_vector* 的变量符号信息,基于该队列生成基本块内其他 SIMD 指令.最后,当遍历完成后,将非空 *V_store* 中二元组(*S_vector*,*B_scalars*)生成相应的 *simd_store* 指令.

对图 6(a)而言,*BB*0 经过 SLP 后,写缓存 *V_store*=(*A*_0,(*A*[*i*],*A*[*i*+1])),读缓存 *V_load*=((*A*_0,(*A*[*i*],*A*[*i*+1])),(*B*_0,(*B*[*i*],*B*[*i*+1])),(*T*_0,(2,2))).当处理 *BB*1 和 *BB*2 时,发现 *reuse_vectors*=((*A*_0,(*A*[*i*],*A*[*i*+1])),(*B*_0,(*B*[*i*],*B*[*i*+1]))),从而使用 *A*_0 和 *B*_0 生成其他指令.在 *BB*3 中,*reuse_vectors*=((*A*_0,(*A*[*i*],*A*[*i*+1])),(*T*_0,(2,2))).当遍历完成时,将 *V_store*=((*C*_0,(*C*[*i*],*C*[*i*+1])),(*A*_0,(*A*[*i*],*A*[*i*+1])))生成 *simd_store* 指令,最终生成的优化 SIMD 代码,如图 6(c)所示.

4 收益指导的向量代码生成

当前的向量化代价模型都是仅仅考虑了连续性、对齐性等访存代价以及向量运算的代价,而没有将控制依赖的特征在代价模型中体现出来.本节首先提出含有控制依赖的收益评估模型.超字条件分支指令能够根据其超字条件的值进行跳转,极大地提高程序的执行效率.然而,当超字条件分支的值大部分为真时,不仅没有绕过

select 指令的执行,反而加入了 BOSCC 指令的开销,降低了控制流向量化的效果,因此需要利用收益模型指导 BOSCC 指令的生成.

4.1 含有控制依赖的收益评估模型

当前的向量化代价评估模型基本上都是从访存和运算两个角度考虑,而没有考虑循环体内含有控制依赖的这个因素,一个典型的代价评估模型[11]如下:

$$Cost_S=\frac{|D^S|}{VF}\left(\sum c_{scalar\to simd}\right)+\sum_{m\in ref_s}\left(\frac{|D^S|}{VF}Cost^m_{\text{extra}}\right) \tag{1}$$

其中,$scalar\to simd$ 表示标量所对应 SIMD 指令,$Cost^m_{extra}$ 表示非连续对齐数组引用 m 的额外开销,$|D^S|$表示语句 s 执行次数:

$$Cost^m_{\text{extra}}=\begin{cases}Cost_{\text{simdset}}, & m\text{为常数}\\ Cost_{\text{load}}+Cost_{\text{shuffle}}, & m\text{连续且不对齐}\\ (coeff-1)(Cost_{\text{load}}+Cost_{\text{shuffle}}), & m\text{不连续}(Coeff<VF)\\ VF\cdot Cost_{\text{load}}+(VF-1)Cost_{\text{shuffle}}, & m\text{不连续}(Coeff\geq VF)\end{cases} \tag{2}$$

当语句不受控制依赖约束时,运行时语句肯定会被执行到,因此,不受控制依赖约束的语句的标量代价直接按照公式计算即可.而控制流内的语句只有当条件为真时才会被执行到,因此在计算标量代价时,要考虑控制条件为真的概率.如果不考虑程序运行时控制条件为真的概率,则会导致标量的代价计算不准确.假设控制条件为真的概率为 p,那么控制条件为假的概率为$(1-p)$.在没有考虑控制条件的概率时,标量的代价可以表示为

$$Cost\_before=Cost\_true+Cost\_false \tag{3}$$

其中,*Cost_true* 表示分支为真内的标量总代价,*Cost_false* 表示分支为假内的标量总代价.而在考虑控制条件的概率时,标量的代价可以表示为

$$Cost\_after=p\times Cost\_true+(1-p)\times Cost\_false \tag{4}$$

由于 $0\leq p\leq 1$,因此 *Cost_after* 小于 *Cost_before*.也就是说,如果没有考虑控制条件分支的概率,则直接利用公式(1)计算得到的标量代价是偏大的.由于向量代码是对 if 转换后条件语句向量化的结果,此时循环体内已经没有分支,因此在计算向量代码的代价时,不需要考虑控制条件执行的概率.因此,是否考虑控制依赖分支的真假,对计算向量代码的代价没有影响.

由于之前计算的标量代价偏大,可能导致一些循环本来向量化没有收益,反而被认为是有收益的.通过反馈,我们在程序运行时获得控制条件为真的概率 p,计算出精确的标量代价去指导生成的向量代码是否有收益.

4.2 收益指导的**BOSCC**指令的生成

超字条件分支指令能够根据其超字条件的值进行跳转.图 8(a)所示的串行程序,if 转换后利用 select 指令生成的向量程序如图 8(b)所示,插入 BOSCC 指令后的程序如图 8(c)所示.

```
for (i=0; i<1024; i++)
  if (fore[i]!=255)
    back[i]=fore[i];
```

(a) 串行

```
for (i=0; i<1024; i+=4){
  v255=(255,255,255,255);
  v_pT=fore[i:i+3]!=v255;
  back[i:i+3]=select(back[i:i+3],
        fore[i:i+3],v_pT)}
```

(b) 利用select指令

```
for (i=0; i<1024; i+=4){
  v255=(255,255,255,255);
  v_pT=fore[i:i+3]!=v255;
 branch-on-none(v_pT) L1;
 back[i:i+3]=select(back[i:i+3],
        fore[i:i+3],v_pT);
   L1;          }
```

(c) 利用select指令和BOSCC指令

Fig.8 Generation flow of BOSCC instruction

图 8 BOSCC 指令的生成过程

当超字条件 *v_pT* 内所有的值为假时,直接跳转到 *L*1,从而避免了 select 指令的执行.但是如果 *v_pT* 的值有一个为真,则不能跳过 select 指令的执行.因此,如果不考虑 *v_pT* 的真值而盲目地生成 BOSCC 指令,则不仅不能

提高生成的向量代码效率,反而会使向量代码的执行效率降低.超字条件分支指令可绕过具有相同预测谓词的语句.当条件分支的真值大部分为真时,不仅没有绕过 select 指令的执行,反而引入了判断 *vpT* 的指令,降低了控制流向量化的效果.因此,需要利用收益模型指导 BOSCC 指令的生成,保证 BOSCC 的生成是有收益的.

构建指导 BOSCC 指令生成的模型需要考虑两个因素:一是超字条件内所有值都为假的比重(percentage of all false superword,简称 PAFS),这个值可以利用反馈编译得到;二是当插入 BOSCC 指令后不需要执行指令的数量,受 BOSCC 条件控制的指令数量(number of bypassed instructions,简称 NBI),NBI 的值可在编译阶段确定.BOSCC 指令分为两类:一类是 BOA,是指所有分支都为真时跳转;一类是 BON,是指所有分支都为假时跳转.由于 BOA 指令发掘和 BON 指令发掘过程类似,因此以 BON 的生成过程为例,说明生成 BOSCC 指令的过程.

4.2.1 PAFS 的计算

由于超字条件内所有值都为假的比重与超字条件值都为假的次数和超字条件的总次数有关,因此我们通过插桩的方法获得程序运行时超字条件执行的次数和超字条件值都为假的次数,进而得到超字条件内所有值都为假的比重:

$$PAFS = \frac{\text{超字条件值都为假的次数}}{\text{超字条件执行的次数}} \tag{5}$$

4.2.2 BOSCC 的识别

在代码生成前,编译器确定和谓词相关的 select 以及被这个谓词控制的指令集.Select 指令的第 3 个操作数代表谓词.BOSCC 的识别算法分为 3 个部分,如算法 3 所示.

- 首先计算符合条件的 select 指令 *dst*=*select*(*src*1,*src*2,*pred*),对于那些第 1 个操作数和目标操作数相同,即 *dst*=*src*1 时,其为 BON 指令所能跳转的 select,其他形式的 select 指令不为 BON 的处理范围.
- 第 2 步是识别出当谓词为真时的指令集合.当谓词的所有值都为假时,select 的结果为 *src*1,因此我们能够绕过以 *src*2 为定义的所有指令,这个指令集可被认为是原来程序的分支.虽然它可以包含一个巨大的指令,递归地跟踪 *src*2 定义的变量,那些有单一定义到达单一使用的指令在谓词控制范围内,因此可以被 BOSCC 指令绕过.
- 第 3 步是用来消除当 *pred* 所有条件为假时的没必要的内存访问.如果在代码中有 *load* 到 *src*1 然后 *store*,并且在 *load* 和 *store* 之间没有修改,并且没有其他指令依赖于 *load* 和 *store*,这两个访存指令可被 *pred* 控制.

算法 **3**. BOSCC 的识别算法.

输入:向量化后的循环.

输出:超字条件控制的语句集合.

1. 对于循环中每个 select 指令 *dst*=*select*(*src*1,*src*2,*pred*),如果 *dst*=*src*1,则转到步骤 2;否则,转到步骤 4.
2. 递归地跟踪 *src*2 定义的变量.那些有单一定义到达单一使用可以被谓词控制范围内,因此可以被 BOSCC 指令绕过.
3. 消除当 *pred* 所有条件为假时的没必要的内存访问.
4. 结束.

4.2.3 BOSCC 指令的生成

假设未插入 BOSCC 指令时需要执行 *NBI* 条指令,那么插入一条 BOSCC 指令后,需要执行的指令数量为 1+*NBI*×(1−*PAFS*) BOSCC 指令.插入 BOSCC 指令是有收益的当且仅当插入 BOSCC 指令后指令执行的代价小于插入前指令执行的代价.假设未插入 BOSCC 指令前 *NBI* 条指令的总代价为 *NBIC*,那么插入 BOSCC 指令后的代价为 *NBIC*×(1−*PAFS*)+*NB*,其中,*NB* 为 BOSCC 指令的代价.插入 BOSCC 指令是有收益的当且仅当 *NBIC*>*NBIC*×(1−*PAFS*)+*NB*,即当 *PAFS*>*NB*/*NBIC* 时,BOSCC 指令的生成如算法 4 所示.

算法 **4**. BOSCC 指令的生成算法.

输入:向量化后的循环.

输出:对有收益的 BOSCC 语句,生成 BOSCC 指令.

1. 对于循环中的每个 select 指令,利用算法 3 识别出其谓词控制的语句.
2. 在保证依赖关系的情况下,通过语句重排序组成谓词控制的语句最大组.
3. 计算该语句区域的指令的总代价为 *NBIC*.
4. 如果谓词的 *PAFS* 大于 *NB*/*NBIC*,表明有收益,则在 BOSCC 指令区域前添加 BOSCC 指令,在 BOSCC 区域结束后添加分支结束语句;否则,不插入 BOSCC 指令.

BOSCC 指令的生成分为以下几个部分:

- 首先,对于循环中的每个 select 指令,利用算法 3 识别出其谓词控制的语句.
- 然后,在保证依赖关系不变的情况下,通过语句重排组成谓词控制的语句最大组,目标是形成最大的 BOSCC 区域.
- 接下来计算该语句区域的指令的总代价 *NBIC*,同时,根据收益公式计算插入 BOSCC 指令是否有收益:如果有收益,则在 BOSCC 指令区域前添加 BOSCC 指令,在 BOSCC 区域结束后添加分支结束语句;否则,不插入 BOSCC 指令.

5 实验和分析

将本文的工作在开源编译器 Open64 中实现,并将改进后的工具命名为 SW-VEC.编译环境为 Linux 操作系统,版本为 Readhat Enterprise 5.实验时,首先用 SW-VEC 将源程序转化为向量程序;然后,再用基础编译器编译成二进制代码,并在国产 CPU 申威-1600 上运行;最后,用串行程序的运行时间除以向量化程序的运行时间就得到向量化的加速比.实验平台 CPU 主频为 2.0GHz,内存为 2GB,*L*1 数据 cache 为 32KB,*L*2 cache 为 256KB,基本页面为 8KB,向量寄存器的宽度为 256 位,可以同时处理 4 个浮点型数据或者 8 个整型数据.SW-VEC 是支持交互式向量化的工具,包括静态分析信息和动态反馈得到的信息,能够通过插桩语句收集程序每个条件表达式的真值信息,指导是否生成 BOSCC 指令.在经过 if 转换后,SW-VEC 中实现了基于 loop-based 和 SLP 的控制流向量化方法.后续实验中提到的改进前方法是指基于 loop-based 和 SLP 的控制流向量化方法,从核心测试和整个测试两个方面对本文提出的改进算法进行测试.

5.1 核心测试

5.1.1 含控制依赖的循环分布

测试比较利用本文提出的循环分布前后核心函数的向量化效果,测试结果如后文图 10 所示,其中,scalar 一列表示串行程序的加速比.由于以串行程序为基准,因此,在图 10 中所有程序 scalar 一列的值为 1.经过 if 转换后,SW-VEC 已经实现了基于 loop-based 的控制流向量化方法,本节以此作为改进前的向量化方法.在图 10 中,simd 一栏表示改进前的向量化加速比,no_reuse+distribution 一栏表示利用本文提出的循环分布算法但是不考虑数据重用获得的向量化加速比,reuse+distribution 一栏表示利用本文提出的循环分布算法同时考虑数据重用获得的向量化加速比.我们选择 SPEC2006 中 456.hmmer 中的 P7Viterbi 和 Berkeley 测试集中 lame 的核心函数 quantize 作为测试用例.

对于核心函数 P7Viterbi 来说,分布前循环的形式如图 9(a)所示,因为其有一条阻碍向量化的真依赖语句,导致获得的向量化加速比为 1.图 9(b)为不考虑局部性的情况下得到的循环分布结果,将其分为 3 部分,分别为{*S*1,*S*2},{*S*3},{*S*4,*S*5,*S*6}.此时,循环向量化后获得的加速比为 1.78.图 9(c)为在考虑局部性的情况下获得的循环分布结果,为{*S*1,*S*2},{*S*3},{*S*4,*S*5},{*S*6},获得了 1.89 的向量化加速比.此时不仅将不能向量化语句分布到了循环外,而且考虑了其他可向量化语句间的数据重用,因此获得的加速效果最高.

对于核心函数 quantize 来说,其循环内同样存在阻碍向量化的真依赖,因此在未利用本文提出的循环分布算法前不能将循环向量化,因此加速比为 1.0;在未考虑局部性的循环分布时,向量化后的加速比 1.12;在考虑局部性时,增加了循环内的数据重用的机会,因此当得到的向量化加速比大于未考虑分布时,加速比为 1.25.

图 10 中最后一列 avg 表示核心函数 P7Viterbi 和 quantize 的在各个优化选项下的平均值.在未利用循环分

别时,向量化加速比的平均值是 1.0;在利用循环分布但是不靠数据重用时,向量化加速比的平均值为 1.45;考虑数据重用后的向量加速比为 1.57.因此,考虑数据重用的控制依赖分布算法比改进前提高了 57%.

```
    for (k=1; k<=M; k++) {
S1      if ((sc1=ip[k-1]+tpim[k-1])>mc[k]) mc[k]=sc1;
S2      if ((sc2=dpp[k-1]+tpdm[k-1])>mc[k]) mc[k]=sc2;
S3      dc[k]=dc[k-1]+tpdd[k-1];
S4      if ((sc3=mc[k-1]+tpmd[k-1])>dc[k]) dc[k]=sc3;
S5      if (dc[k]<-INFTY) dc[k]=-INFTY;
        if (k<M) {
            ic[k]=mpp[k]+tpmi[k];
            if ((sc4=ip[k]+tpii[k])>ic[k]) ic[k]=sc4;
S6          ic[k]+=is[k];
            if (ic[k]<-INFTY) ic[k]=-INFTY;
          }
    }
```

(a) 分布前

```
    for (k=1; k<=M; k++) {
S1    if ((sc=ip[k-1]+tpim[k-1])>mc[k]) mc[k]=sc;
S2    if ((sc=dpp[k-1]+tpdm[k-1])>mc[k]) mc[k]=sc;}
    for (k=1; k<=M; k++) {
S3    dc[k]=dc[k-1]+tpdd[k-1];}
    for (k=1; k<=M; k++) {
S4    if ((sc=mc[k-1]+tpmd[k-1])>dc[k]) dc[k]=sc;
S5    if (dc[k]<-INFTY) dc[k]=-INFTY;}
          if (k<M) {
          ic[k]=mpp[k]+tpmi[k];
S6        if ((sc=ip[k]+tpii[k])>ic[k]) ic[k]=sc;
          ic[k]+=is[k];
          if (ic[k]<-INFTY) ic[k]=-INFTY;} }
```

(b) 不考虑数据布局的分布

```
    for (k=1; k<=M; k++) {
S1    if ((sc=ip[k-1]+tpim[k-1])>mc[k]) mc[k]=sc;
S2    if ((sc=dpp[k-1]+tpdm[k-1])>mc[k]) mc[k]=sc;}
    for (k=1; k<=M; k++) {
S3    dc[k]=dc[k-1]+tpdd[k-1];}
    for (k=1; k<=M; k++) {
S4    if ((sc=mc[k-1]+tpmd[k-1])>dc[k]) dc[k]=sc;
S5    if (dc[k]<-INFTY) dc[k]=-INFTY;}
      for (k=1; k<=M; k++) {
        if (k<M) {
          ic[k]=mpp[k]+tpmi[k];
S6        if ((sc=ip[k]+tpii[k])>ic[k]) ic[k]=sc;
          ic[k]+=is[k];
          if (ic[k]<-INFTY) ic[k]=-INFTY;} }
```

(c) 考虑数据布局的分布

Fig.9 Distribution results of P7Viterbi kernel

图 9 函数 P7Viterbi 核心循环的分布结果

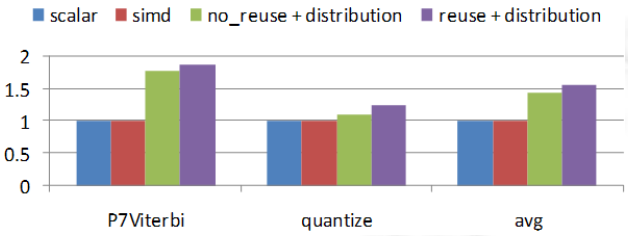

Fig.10 Results of loop distribution with control dependence

图 10 含控制依赖的循环分布测试结果

5.1.2 面向向量重用的直接 SIMD 向量化控制流

面向向量重用的直接控制流向量化的测试结果如图 11 所示,其中,scalar 一列表示串行程序的加速比.由于以串行程序为基准,因此在图 11 中,所有程序 scalar 一列的值为 1.经过 if 转换后,SW-VEC 已经实现了参考文献[8]中基于 SLP 的控制流向量化方法,本节以此作为改进前的向量化方法.

在图 11 中,simd 一栏表示改进前的向量化加速比;direct_simd 一栏表示利用本文提出的直接控制流向量化方法,但是没有考虑基本块间的向量复用;direct_simd+reuse 一栏表示利用直接控制流向量化方法,并且考虑基本块的向量复用.测试用例分别是 SPEC2000 的 191.fma3d 的核心函数 solve、183.equake 的核心函数 smvp、NPB 中 MG 的函数 zran3 以及多媒体程序中的 H.264.

核心函数 solve 基于 if 转换的向量化加速比为 1.17,不考虑数据重用的直接控制流向量化加速比为 1.08,考虑重用后的直接控制流向量化的加速比为 1.29.核心函数 smvp 基于 if 转换的向量化加速比为 1.09,不考虑数据重用的直接控制流向量化加速比为 1.20,考虑重用后的直接控制流向量化的加速比为 1.25.

核心函数 zran3 基于 if 转换的向量化加速比为 1.35,不考虑数据重用的直接控制流向量化加速比为 1.28,考虑重用后的直接控制流向量化的加速比为 1.41.核心函数 H.264 基于 if 转换的向量化加速比为 1.32,不考虑数据重用的直接控制流向量化加速比为 1.45,考虑重用后的直接控制流向量化的加速比为 1.51.

从 solve 和 zran3 来看,在没有考虑基本块间的数据重用时,直接控制流向量化加速比是小于基于 if 转换的向量化加速比的;考虑基本块间的数据重用后,加速比大于基于 if 转换的向量化加速比.由此可以看出基本块间的数据重用对于直接控制流向量化的重要性.

图 11 中最后一列 avg 表示前面程序的各个选项下的平均值.基于 if 转换的向量化加速比的平均值为 1.23,不考虑数据重用的直接向量化加速比的平均值为 1.25,考虑数据重用后向量化加速比的平均值为 1.37.因此,考虑数据重用的直接向量化加速比比基于 if 转换向量化加速比提高了 11%.

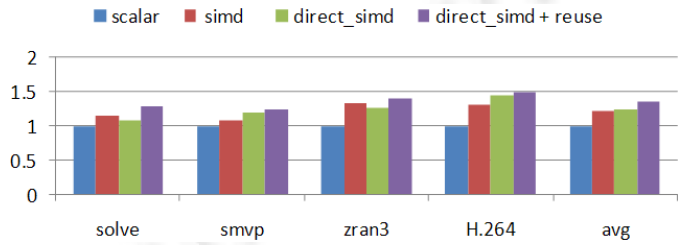

Fig.11 Result of direct control flow vectorization method

图 11 直接控制流向量化测试结果

5.1.3 收益指导的向量代码生成

收益模型指导的向量代码生成测试结果如图 12 所示,图中 scalar 一栏表示标量执行的加速比,simd 一栏表示没有利用代价模型获得的加速比,simd+cost_model 一栏表示利用本文提出的代价模型后获得的加速比.

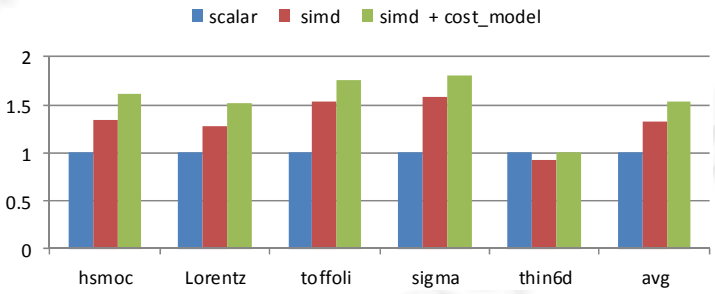

Fig.12 Results of vectorization method guided by benefit

图 12 收益指导的向量代码测试结果

hsmoc 和 lorentz 都来自 434.zeusmp.hsmoc 在没有利用代价模型时,向量化后的加速比为 1.34;利用收益模型后,向量化的加速比为 1.61.加速比提高的原因分为两个部分:首先,在没有利用收益模型时,核心函数内一些循环的向量执行效率低于标量执行效率,而利用收益模型指导后没有对这部分循环向量化.此外,通过生成 BOSCC 指令,也进一步提升了核心函数的向量化加速比.核心函数 lorentz 和 hsmoc 的情况相同,没有利用收益模型指导时向量化的加速比为 1.28,利用收益模型指导后向量化的加速比为 1.52.

toffoli 和 sigma 来自 462 的 3 个核心循环.toffoli 的直接向量化的加速效果为 1.53,由于核心循环是一个两层的嵌套 if 语句而且循环体内无其他语句,因此通过生成 BOSCC 指令大大提高了程序的向量执行效果,加速比

为 1.75.核心函数 sigma 的情况和 toffoli 相似,循环内为一层的嵌套 if 语句,因此通过生成 BOSCC 指令同样可以提高程序的向量执行效率.在没有利用收益模型时向量化的加速比为 1.58,利用收益模型后向量化的加速比为 1.80.

thin6d 是 SPEC2000 中的 200.sixtrack 核心函数,在未利用收益模型指导时,向量化的加速比为 0.92,说明该核心函数向量执行的代价大于标量代价.因此,在经过收益分析后,该核心函数不能转为向量执行,而采用标量执行.因此,采用收益模型指导向量代码生成后的向量加速比为 1.0.

图 12 中最后一列 avg 表示前面的程序在各个选项下的平均值.在未利用收益模型指导代码生成时,向量化加速比的平均值是 1.33;利用代价模型后,向量化加速比的平均值为 1.54.利用收益模型后,向量化加速比提高了 16%.

5.2 整体测试

我们选定 SPEC2006 和 SPEC2000 中核心函数含有控制流语句的程序作为整体测试用例,选定测试程序的核心函数、核心函数比重和程序功能等基本情况见表 1.

Table 1 Overview of the kernels for full program test

表 1 整体测试用例的基本情况

程序名称	测试集	核心函数	核心函数比重(%)	程序功能
456.hmmer	Spec2006	P7Viterbi	79.76	基因序列搜索
434.zeusmp	Spec2006	hsmoc	38.74	计算流体力学
434.zeusmp	Spec2006	Lorentz	22.35	计算流体力学
462.libquantum	Spec2006	toffoli	72.68	量子计算
462.libquantum	Spec2006	sigma	21.35	量子计算
183.equake	Spec2000	smvp	72.87	地震波传导模拟

这些核心函数都排在所有函数的第一位或者第二位,若能提高这些核心函数的向量执行效率,则必定能够大幅度加快整个程序的运行速度,在前面的核心测试部分已经对这些程序的核心函数进行了测试.整体测试程序包括 456.hmmer,434.zeusmp,462.libquantum 和 183.equake,测试这 4 个程序在利用本文改进算法前后的向量化加速比.测试结果如图 13 所示,其中,scalar 一列表示串行程序的加速比.由于以串行程序为基准,因此在图 13 中,所有程序 scalar 一列的值为 1.经过 if 转换后,在 SW-VEC 中实现了基于 loop-based 和 SLP 的控制流向量化方法,即 loop-aware 控制流向量化方法,本节以此作为改进前的向量化方法.图 13 中,simd 表示改进前程序的向量执行的加速比,simd+improved 表示利用本文提出的改进方法后程序的向量执行的加速比.

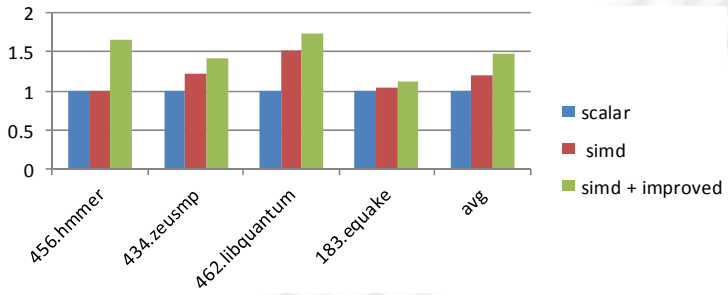

Fig.13 Test results of full programs

图 13 整体测试结果

456.hmmer 的核心函数为 P7Viterbi,所占比重为 79.76%.由于 P7Viterbi 内存在一条阻碍向量化的真依赖,没有采用循环分布时不能对其向量化,因此在图 13 中,simd 一栏对应的值为 1;采用本文提出的含控制依赖循环分布后,可以成功地将其转为向量执行,同时增加了数据重用的机会,因此在改进后获得了 1.66 的加速比.

434.zeusmp 的两个核心函数 hsmoc,lorentz 所占比重为 38.74%和 22.35%,它们在程序中都存在最大值 max

操作,我们将其转换为控制语句然后向量化.没有改进前获得的向量化加速比为 1.23,改进后获得的向量化加速比为 1.41.这是由于改进后代价模型指导是否向量化后有收益,并通过插入 BOSCC 指令进一步提高加速比.

462.libquantum 的 2 个核心函数 toffoli,sigmax 所占的比重分别为 72.68%和 21.35%.这两个核心都含有控制流语句,没有改进前获得的向量加速比为 1.52;改进后由于有代价模型指导是否对控制流进行向量化,并且通过插入 BOSCC 所得加速比为 1.74.

183.equake 的核心函数为 smvp,所占的比重为 72.87%,改进前的加速比为 1.04,利用本文提出的改进方法后,加速比为 1.13,这是直接向量化控制流方法在起作用.

图 13 中最后一列 avg 表示前面程序的各个优化方案的平均值.在未改进前,向量化加速比的平均值是 1.19,而利用本文提出的改进控制流向量化方法后的平均向量加速比为 1.48.因此,利用本文提出的改进控制流向量化方法后向量化加速比提高了 24%.

6 相关研究

目前,自动 SIMD 向量化研究的内容主要包括以下两个方面:一是访存的对齐性和连续性,二是发掘方法.由于对齐访存要比不对齐访存快得多,而应用程序中很多程序都不是对齐的,因此处理好不对齐访存是提高向量化代码执行效率的重要途径[12].解决不对齐访存的方法可分为 3 类:一是通过多次对齐访存,然后移位[13]或者重组[14];二是通过循环变换[7];三是硬件支持不对齐访存[15].由于访存指令只能从内存中连续地加载数据到向量寄存器中,因此,不连续访存的程序就需要额外的操作使其在向量寄存器内连续.解决不连续访存程序的方法可分为 3 类:一是向量重组[16,17],二是数据重组[18],三是硬件支持[19,20].

在发掘方法方面,loop-based 向量化方法与面向向量机的传统向量化方法相同[7],都是在迭代间并行.随后提出了面向 SIMD 的外层循环向量化[21]、多面体指导的多重循环向量化[11,22].SLP 是第 1 种发掘迭代内并行的算法,后面又对其进行了改进[23,24].此外还有函数级向量化[25],但由于函数级向量化涉及到过程间分析和别名分析,使得发掘难度比较大.文献[26]提出了一种面向 SIMD 扩展部件的出口分支向量化方法.文献[27]提出了如何对嵌套的控制流语句生成 BOSCC 指令,但并没有考虑代价的问题.基于投机并行的 SIMD 向量化是未来研究的重要方向[28,29].

7 结束语

针对当前控制流向量化方法生成的代码效率较低的问题,本文给出了改进的控制流 SIMD 向量化方法.首先,提出了含有控制依赖的循环分布算法,分离循环的可向量化部分和不可向量化部分,进而将可向量化部分转为向量执行,同时考虑分布后循环的数据局部性;其次,提出了一种考虑基本块间向量重用的直接 SIMD 向量化控制流的方法,以发掘循环内同构语句条数较多的循环,避免了超字选择指令的生成;最后,利用代价模型指导超字选择指令和超字条件分支指令的生成,提高生成向量代码的效率.经过标准测试集的测试结果表明,与现有的控制流向量化方法相比,本文提出的改进方法生成的向量代码性能提高了 24%.利用投机并行解决编译时依赖信息不能确定的控制流向量化问题,是未来研究的重要方面.

致谢 在此,向对本文研究工作提供基金支持的单位和评阅本文的审稿专家表示衷心的感谢,向为本文研究工作提供基础和研究平台的前辈致敬.

References:

[1] Gao W, Zhao RC, Han L, Pang JM, Ding R. Research on SIMD auto-vectorization compiling optimization. Ruan Jian Xue Bao/ Journal of Software, 2015,26(6):1265−1284 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4811.htm [doi: 10.13328/j.cnki.jos.004811]

[2] Peng F, Gu NJ. SIMD compiler optimization and analysis based on Godson-3B processor. Journal of Chinese Computer Systems, 2012,33(12):2733−2737 (in Chinese with English abstract). [doi: 10.3969/j.issn.1000-1220.2012.12.032]

[3] Xin NJ, Chen XC. Extending the vector instruction set for high-performance DSP matrix based on GCC. Computer Engineering & Science, 2012,34(1):57−63 (in Chinese with English abstract). [doi: 10.3969/j.issn.1007-130X.2012.01.010]

[4] Xu HY, Zheng QL. Vectorization algorithm for multi-cluster and VLIW DSP. Computer Application & System, 2013,22(12): 140−143 (in Chinese with English abstract). [doi: 10.3969/j.issn.1003-3254.2013.12.027]

[5] Allen J, Kennedy K, Porterfield C, Warren J. Conversion of control dependence to data dependence. In: Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). New York: ACM Press, 1983. 177−189. [doi: 10.1145/567067.567085]

[6] Smith JE, Faanes G, Sugumar R. Vector instruction set support for conditional operations. In: Proc. of the 27th Annual Int'l Symp. on Computer Architecture. Vancouver, 2000. 260−269. [doi: 10.1145/339647.339693]

[7] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, 2001.

[8] Shin J, Hall M, Chame J. Superword-Level parallelism in the presence of control flow. In: Proc. of the 3rd Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). New York: ACM Press, 2005. 165−175. [doi: 10.1109/CGO.2005.33]

[9] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2000. 145−156. [doi: 10.1145/349299.349320]

[10] Wolf ME, Lam MS. A data locality optimizing algorithm. In: Proc. of the SIGPLAN'91 Conf. on Programming Language Design and Implementation. Toronto, 1991. 30−44. [doi: 10.1145/113445.113449]

[11] Trifunovic K, Nuzman D, Cohen A, Zaks A, Rosen I. Polyhedral-Model guided loop-nest auto-vectorization. In: Proc. of the 2009 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2009. [doi: 10.1109/PACT.2009.18]

[12] Shahbahrami A, Juurlink B, Vassiliadis S. Performance impact of misaligned accesses in SIMD extensions. In: Proc. of the 17th Annual Workshop on Circuits, Systems and Signal Processing. 2006. 334−342.

[13] Eichenberger AE, Wu P, O'Brien K. Vectorization for SIMD architectures with alignment constraints. In: Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2004. 82−93. [doi: 10.1145/996841.996853]

[14] Larsen S, Witchel E, Amarasin SP. Increasing and detecting memory address congruence. In: Proc. of the 2002 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2002. 18−29. [doi: 10.1109/PACT.2002.1105970]

[15] Chang H, Sung W. Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware. In: Proc. of the 2008 Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES). 2008. 167−176. [doi: 10.1145/1450095.1450121]

[16] Bik AJC, Girkar M, Grey PM, Tian X. Automatic intra-register vectorization for the intel architecture. Int'l Journal of Parallel Programming, 2002,30(2):65−98. [doi: 10.1023/A:1014230429447]

[17] Ren G, Wu P, Padua DA. Optimizing data permutations for SIMD devices. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2006. 118−131. [doi: 10.1145/1133981.1133996]

[18] Li YX, Shi H, Chen L. Vectorization-Oriented local data regrouping. Computer System, 2009,30(8):1529−1534 (in Chinese with English abstract).

[19] Nuzman D, Rosen I, Zaks A. Auto-Vectorization of interleaved data for SIMD. In: Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation (PLDI). 2006. 132−143. [doi: 10.1145/1133981.1133997]

[20] Huang LB, Shen L, Wang ZY, Shi W, Xiao N, Ma S. SIF: Overcoming the limitations of SIMD devices via implicit permutation. In: Proc. of the 16th Int'l Symp. on High-Performance Computer Architecture (HPCA). 2010. 355−366. [doi: 10.1109/HPCA.2010.5416631]

[21] Nuzman D, Zaks A. Outer-Loop vectorization—Revisited for short SIMD architectures. In: Proc. of the 2008 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2008. 2−11. [doi: 10.1145/1454115.1454119]

[22] Kong M, Veras R, Stock K. When polyhedral transformations meet SIMD code generation. In: Proc. of the 2013 Conf. on Programming Language Design and Implementation (PLDI). 2013. 127−138. [doi: 10.1145/2491956.2462187]

[23] Barik R, Zhao JS, Sarkar V. Efficient selection of vector instructions using dynamic programming. In: Proc. of the 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO). 2010. 5−12. [doi: 10.1109/MICRO.2010.38]

[24] Liu J, Zhang YR, Kandemir M. A compiler framework for extracting superword level parallelism. In: Proc. of the 2012 Conf. on Programming Language Design and Implementation (PLDI). 2012. 347−358. [doi: 10.1145/2254064.2254106]

[25] Karrenberg R, Hack S. Whole-Function vectorization. In: Proc. of the 9th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). 2011. 12−21. [doi: 10.1109/CGO.2011.5764682]

[26] Zhu JF, Zhao RC. A vectorization method of export branch for SIMD extension. In: Proc. of the 10th Conf. IEEE/ACIS Int'l Conf. on Computer and Information Science (ICIS). 2011. 265−269. [doi: 10.1109/ICIS.2011.49]

[27] Shin J. Introducing control flow into vectorized code. In: Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT). Washington: IEEE Computer Society, 2007. 280−291. [doi: 10.1109/PACT.2007.41]

[28] Kumar1 R, Martínez A. Speculative dynamic vectorization for HW/SW codesigned processors. In: Proc. of the 2012 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT). 2012. 459−460. [doi: 10.1145/2370816.2370895]

[29] Haque M, Yi Q. Past dependent branches through speculation. In: Proc. of the 22nd Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT). Washington: IEEE Computer Society, 2013. [doi: 10.1109/PACT.2013.6618831]

附中文参考文献:

[1] 高伟,赵荣彩,韩林,庞建明,丁锐.SIMD 自动向量化编译优化概述.软件学报,2015,26(6):1265−1284. http://www.jos.org.cn/1000-9825/4811.htm [doi: 10.13328/j.cnki.jos.004811]

[2] 彭飞,顾乃杰.龙芯 3B 的 SIMD 编译优化及分析.小型微型计算机系统,2012,33(12):2733−2737. [doi: 10.3969/j.issn.1000-1220.2012.12.032]

[3] 辛乃军,陈旭灿.基于 GCC 的高性能 DSP Matrix 向量指令集扩展.计算机工程与科学,2012,34(1):57−63. [doi: 10.3969/j.issn.1007-130X.2012.01.010]

[4] 徐华叶,郑启龙.面向多簇超长指令字 DSP 的向量化优化算法.计算机应用系统,2013,22(12):140−143. [doi: 10.3969/j.issn.1003-3254.2013.12.027]

[18] 李玉祥,施慧,陈莉.面向向量化的局部数据重组.小型微型计算机系统,2009,30(8):1528−1534.

高伟(1988－),男,黑龙江齐齐哈尔人,博士生,主要研究领域为先进编译技术.

李颖颖(1984－),女,讲师,主要研究领域为高性能计算.

孙回回(1991－),女,博士生,主要研究领域为先进编译技术.

李雁冰(1989－),男,博士生,CCF 专业会员,主要研究领域为先进编译技术.

赵荣彩(1957－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为高性能计算,先进编译技术.