Journal of Computer Applications
计算机应用,2013,33(11):3149-3154
ISSN 1001-9081
CODEN JYIIDU
2013-11-01
http://www.joca.cn

文章编号:1001-9081(2013)11-3149-06
doi:10.11772/j.issn.1001-9081.2013.11.3149

非正规化循环的单指令多数据向量化

侯永生*,赵荣彩,高 伟,高 伟
(数学工程与先进计算国家重点实验室(信息工程大学),郑州 450002)
(*通信作者电子邮箱 magichys@126.com)

摘 要:针对非正规化循环的上下界、步长等循环信息不确定的问题,解决了循环条件为逻辑表达式、增量减量语句和 do-while 循环的正规化问题。对不能正规化的循环提出了一种展开压紧算法,并用超字并行向量化方法发掘展开压紧的结果。实验结果表明,与现有的非正规化循环的单指令多数据(SIMD)向量化方法相比,所提出的转换方法和展开压紧方法能够更好地发掘非正规化循环的向量化特性,生成代码的性能加速比提高了6%以上。

关键词:非正规化循环;单指令多数据向量化;展开压紧;依赖关系分析
中图分类号:TP314 **文献标志码**:A

Single instruction multiple data vectorization of non-normalized loops

HOU Yongsheng*, ZHAO Rongcai, GAO Wei, GAO Wei
(*State Key Laboratory of Mathematical Engineering and Advanced Computing (Information Engineering University), Zhengzhou Henan 450002, China*)

Abstract: Concerning that the upper, lower bounds and stride of the non-normalized loop are uncertain, some issues were normalized based on a transform method such as that loop conditions were logical expression, increment-reduction statement and do-while. An unroll-jam method was proposed to deal with the loops that cannot be normalized, which mined the unroll-jam results by Superword Level Parellelism (SLP) vectorization. Compared with the existing Single Instruction Multiple Data (SIMD) vectorization method for non-normalized loops, the experimental results show that the transform method and unroll-jam method are better to explore the parallelism of the non-normalized loops, which can improve the performance by more than 6%.

Key words: non-noramlized loop; Single Instruction Multiple Data (SIMD) vectorization; unroll and jam; data dependence analysis

0 引言

循环正规化是指将循环的下界变为1、上界变为某个特定的值,并且跨距变为1的过程[1]。循环正规化成功得到的是正规化循环,失败得到的是非正规化循环。循环正规化失败是循环索引或者索引的步长和上下界等信息不确定造成的。传统向量化方法、超字并行(Superword Level Parallelism, SLP)向量化方法以及循环变换技术都是基于正规化循环。而非正规化循环在预分析和优化阶段没有进入到向量化循环候选集合中,因此失去了单指令多数据(Single Instruction Multiple Data, SIMD)向量化的机会。然而非正规化循环广泛应用于程序中,如何发掘其向量化特性是亟待解决的问题。

开源编译器 Open64 在高级中间表示阶段有三种循环结构,分别是 DO_LOOP、DO_WHILE、WHILE_DO,其中 Fortran 程序的 DO 循环生成为 DO_LOOP,C 程序中的 for 循环和 while 循环以及 Fortran 中的 do-while 循环都生成为 WHILE_DO 结构。C 程序中的 do-while 循环生成为 DO_WHILE 结构。DO_LOOP 是正规化循环,WHILE_DO 和 DO_WHILE 是非正规化循环。编译时如加选项-O3 则在循环嵌套优化阶段将满足循环正规化条件的 WHILE_DO 转换为 DO_LOOP。本文中将正规化循环标记为 DO_LOOP 循环,而把非正规化循环标记为 WHILE_DO 循环。

文献[2]针对流水体系结构机器提出了对 while 循环的并行方法,取得了很好的加速比;文献[3]利用多面体模型解决 while 循环的并行发掘问题;文献[4]利用时空转换的方法处理 while 循环的并行问题;文献[5]研究了含有 while 的循环利用循环变换技术发掘并行性的问题;文献[6]用不变量关系分析 while 循环的结束条件。OpenMP version2.5 之前的版本要求循环必须是可正规化循环,但是 OpenMP version3.0 引入了 section 并行,因此利用 section 发掘不规则内存访问的 while 循环并行是目前一个研究热点;Larsen 等[7]提出的超字并行向量化算法首先进行循环展开使基本块内有足够的并行性,而目前没有关于非正规化循环展开的方法,因此发掘 while 循环的 SIMD 向量化受到限制;魏帅等[8]提出了面向多重循环的向量化方法;Liu 等[9]提出一种面向全局的发掘超字并行向量化的方法;文献[10]综合考虑了 SIMD 向量化代码生成的对齐、连续等问题。

1 非正规化循环的分类

WHILE_DO 循环又叫“当型”循环结构,图1给出了 WHILE_DO 循环的流程,其特点是:首先计算循环条件的值,若非0,执行循环体;若为0则退出 WHILE_DO 循环。本文对标准测试集中所有的 WHILE_DO 循环进行了研究并按循环条件可分为常量、函数和表达式这三类。

收稿日期:2013-05-14;**修回日期**:2013-07-22。

作者简介:侯永生(1978-),男,河南郑州人,博士研究生,主要研究方向:先进编译技术; 赵荣彩(1957-),男,河南洛阳人,教授,博士生导师,博士,主要研究方向:高性能计算、先进编译技术、反编译技术; 高伟(1988-),男,黑龙江齐齐哈尔人,博士研究生,主要研究方向:先进编译技术; 高伟(1989-),男,吉林桦甸人,硕士研究生,主要研究方向:先进编译技术。

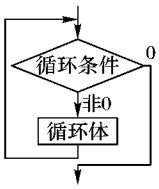


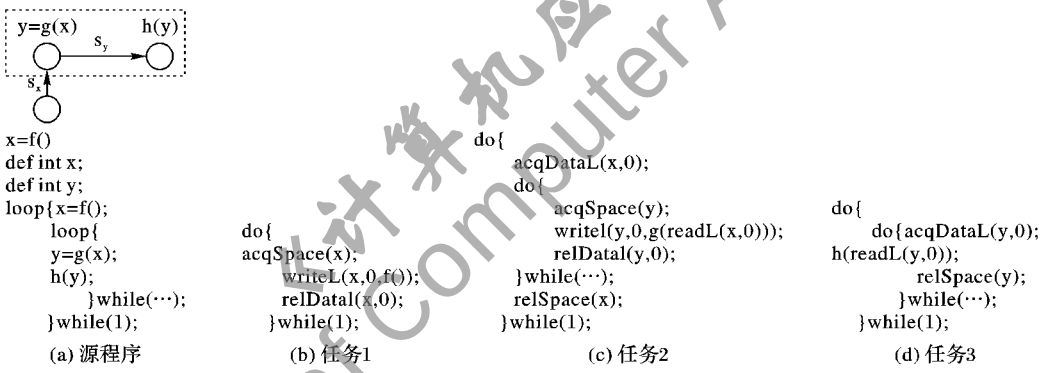
图 1 WHILE_DO 循环的流程

循环条件为常量的 WHILE_DO 循环大约占被研究测试集中 WHILE_DO 循环总数的 2% 左右。WHILE_DO 循环条件可为任意常量,但是常用 WHILE_DO(1)形式,在 WHILE_DO 循环中有时用户编写成 WHILE_DO(0)的形式。此类循环的循环体中一般都存在 goto,break 等跳转语句。跳转语句将循环体分割成了多个基本块,增加了并行发掘的难度。Geuns 等[11]针对一类常量循环提出了其在共享内存多处理器上并行的方法。如图 2(a)所示的循环可将其分成三个任务:任务 1 计算 x 的值并将其放入缓冲区中,如图 2(b)所示;任务 2 从缓冲区中去 x,根据 x 的值计算 y 的值,并将计算出的 y 值放入缓冲区,如图 2(c)所示。任务 3 从缓冲区取出任务 2 计算出的 y 值,计算 h(y),如图 2(d)所示。缓冲区采用先进先出策略,这样任务 1、任务 2 和任务 3 便可并行。

循环条件为函数的 WHILE_DO 循环大约占被研究测试集中 WHILE_DO 循环总数的 3% 左右。如函数返回为真则执行循环体,否则退出循环。循环条件是函数的WHILE_DO循环并不多,但是循环体内含有函数调用语句的 WHILE_DO 循环却比较多。目前几乎所有的并行发掘算法都不发掘带有函数调用的循环,原因是函数调用涉及到过程间分析,这给依赖关系分析带来了困难。所以编译器保守的认为存在依赖而不并行。如何并行含有函数调用的循环是编译的难点。

循环条件为表达式的 WHILE_DO 循环大约占被研究测试集中 WHILE_DO 循环总数的 95% 左右,具有重大并行发掘的潜能。表达式是指用操作符、操作数以及括号连接起来的符合语法规范的式子。C 语言中表达式包括算术表达式、逻辑表达式、关系表达式、赋值表达式和逗号表达式。在研究的测试集中循环条件没有是逗号表达式的情况,程序员一般不会将逗号表达式作为循环条件。赋值表达式作为循环条件也比较少见。关系表达式、逻辑表达式和算术表达式是常见的 WHILE_DO 循环的循环条件。因此如何发掘此类循环的并行性是本文要解决的问题。

循环条件为表达式的 WHILE_DO 循环 SIMD 向量化发掘流程如图 3 所示。和 WHILE_DO 循环相比,DO_LOOP 循环的并行发掘方法和循环变换技术更成熟,因此优先将 WHILE_DO 循环转换为 DO_LOOP 循环(标号①),并对转换后的 DO_LOOP 循环发掘 SIMD 并行(标号③)。对于不能够转换的 WHILE_DO 循环,将其进行循环展开(标号②),并发掘 SIMD 并行(标号③)。



(a) 源程序 (b) 任务1 (c) 任务2 (d) 任务3

图 2 一种并行 WHILE_DO 常量循环方法

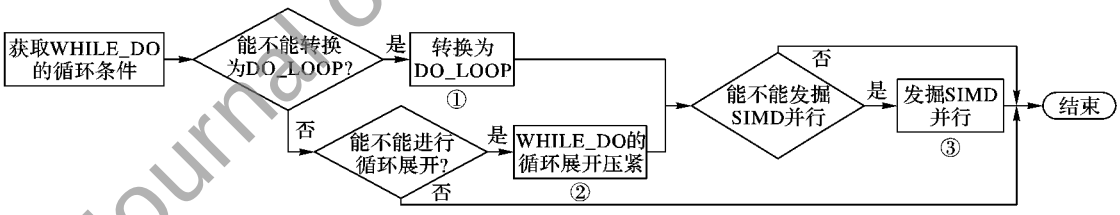


图 3 非正规化循环的 SIMD 向量化发掘流程

2 非正规化循环的转换

2.1 循环可正规化的条件

WHILE_DO 转换为 DO_LOOP 的条件与最新 OpenMP 规范对循环条件要求基本相同。转换条件如下:

1)循环条件的操作符必须为小于或大于等比较操作;

2)一端的操作数必须为循环不变量 invariant,另一端的操作数必须为变量 variant;

3)WHILE_DO 循环前必须有对 variant 的定义语句 INIT;

4)WHILE_DO 循环体内必须有对 variant 的定义语句 INC;

5)INC 语句须为 variant = incr + / - variant 形式,incr 为循环不变量;

6)INIT 和 INC 语句必须为直接内存访问操作,而不能是间接内存访问。

下面解决 3 种由于转换条件限制过于严格而正规化失败的情况:1)循环条件为逻辑表达式的循环正规化;2)循环条件为增量、减量语句的循环正规化;3)中间表示为DO_WHILE 的循环正规化。

2.2 循环条件为逻辑表达式的循环正规化

循环条件含双目逻辑运算符的 WHILE_DO 不能转换为 DO_LOOP。双目逻辑运算符包括逻辑与(&&)和逻辑或(‖)。这类循环其实是包含多个循环条件,其中或表示多个循环条件中有一个满足便执行循环体,而与表示多个循环条件都要满足才执行循环体。

循环条件含有逻辑运算符或的 WHILE_DO 由于满足其

中任意一个循环条件都执行循环体,因此在编译阶段无法确定其执行的顺序,虽可通过动态插桩的方法收集循环运行时信息简化循环条件,但是难度太大且可能无收益,因此放弃将此类循环转换为 DO_LOOP。而循环条件含有逻辑运算符与的 WHILE_DO 由于必须满足所有循环条件才执行循环体,因此可在编译阶段将其转为 DO_LOOP。对于形如图 4(a)的循环,其中 B1 表示其中一个满足 WHILE_DO 转 DO_LOOP 的循环条件,B2 表示其他的循环条件,INC1 表示 B1 的增量,INC2 表示 B2 的增量,S 表示循环体内除了 INC1 和 INC2 其他的语句。为了保证变换的正确性,限定循环条件 B1 的变量仅在 INC1 内定义,B2 的变量仅在 INC2 内定义,它们在且仅在 S 内使用,即 INC1 与 INC2 无依赖关系。

```
while(B1&&B2){                    for(INIT1;B1;INC1){
    S;                                if(B2){
    INC1;                                 S;
    INC2;                                 INC2;}
  }                                   else
                                      break;
                                    }
```

(a) 非正规化循环　　(b) 等价的正规化循环

图 4　逻辑表达式的转换

定理 1　形如图 4(a)中的 WHILE_DO 可转换为形如图 4(b)中语义等价的 DO_LOOP。

证明　采用数学归纳法,设循环执行的次数为 N(N≥0),循环条件 B1 执行的次数为 N1(N1≥N),B2 执行的次数为 N2(N2≥N)。

当 N = 0 时,WHILE_DO 语句执行序列为 B1B2,DO_LOOP 语句执行序列为 B1B2,WHILE_DO 和 DO_LOOP 语句执行序列相同,语义等价。

当 N = 1 时,WHILE_DO 语句执行序列为 B1B2 S INC1 INC2,DO_LOOP 语句执行序列为 B1 B2 S INC2 INC1,由于 INC1 与 INC2 无依赖关系,因此可交换其执行顺序,WHILE_DO 和 DO_LOOP 语句执行序列相同,语义等价。

当循环体执行第 n((N≥n≥1)次时,此时还循环还未结束。WHILE_DO 语句执行序列为 B1 B2 S INC1 INC2(n 个),DO_LOOP 语句执行序列为 B1 B2 S INC2 INC1(n 个),WHILE_DO 和 DO_LOOP 语句执行序列相同,语义等价。

当执行第 n + 1 次时。如未退出循环,此时 n < N。WHILE_DO 语句执行序列为 B1 B2 S INC1 INC2(n + 1 个),DO_LOOP 语句执行序列为 B1 B2 S INC2 INC1(n + 1 个),WHILE_DO 和 DO_LOOP 语句执行序列相同,语义等价。如退出循环,此时 n = N。如因 B1 条件不满足退出循环,WHILE_DO 语句执行序列为 B1 B2 S INC1 INC2(N 个)B1,DO_LOOP 语句执行序列为 B1 B2 S INC2 INC1(N 个)B1,语义等价。如因 B2 条件不满足退出循环,WHILE_DO 语句执行序列为 B1 B2 S INC1 INC2(N 个)B1B2,DO_LOOP 语句执行序列为 B1 B2 S INC2 INC1(N 个)B1B2,WHILE_DO 和 DO_LOOP 语句执行序列相同,语义等价。

综上,图 4(a)中 WHILE_DO 与图 4(b)DO_LOOP 等价。证毕。

可通过反馈编译[12]的方式确定循环条件不满足 B1 还是 B2 而退出循环,如能确定因 B1 不满足而退出循环可删除循环体内 else 分支。图 5(a)是标准性能测评公司(Standard Performance Evaluation Company,SPEC)的测试集 177. mesa 中 span. c 代码,其中 B1 为 x < 0,INC1 为 x ++ ,x 的初值为 INIT,B2 为 n > 0,INC2 为 n - - 。利用逻辑表达式转换的方法将其转换为等价的 for 形式如图 5(b)所示。

```
while(x<0 && n>0) {                for(x=INIT;x<0; x++){
  red[i] = green[i] = 0;             if(n>0) {
  blue[i] = alpha[i] =0 ;              red[i] = green[i] = 0;
  x++;                                 blue[i] = alpha[i] = 0;
  n--;                                 n--;
  i++;                                 i++;
    }                                    }
                                     else
                                       Break;}
```

(a) 非正规化循环　　(b) 等价的正规化循环

图 5　逻辑表达式转换的实例

2.3　循环条件含有增量和减量运算符的循环正规化

C 语言中定义了自增、自减运算符,作用是使变量的值增 1 或者减 1。自增运算有两种形式:i ++ 和 ++ i。不同之处在于:++ i 是先执行 i = i + 1 后,再使用 i 的值,这是真依赖的情况;而 i ++ 是先使用 i 的值后,再执行 i = i + 1,这是反依赖的情况。当它们出现在 WHILE_DO 的循环条件时,却有很大的不同。形如图 6(a)所示的 WHILE_DO 因其中间表示存在反依赖而不满足转换为 DO_LOOP 的条件,而形如 6(b)的中间表示无反依赖却可将其转换为 DO_LOOP。可以将循环条件含有 i ++ 循环转换为等价循环条件含有 ++ i 循环,而后将其转换为 DO_LOOP 进行并行发掘。

```
While (i++< N)        While (++i< N)        While (++i<N)
{S}                   {S}                     {S; }
                                              if((i-1)<N)
                                                S;
```

(a) i++形式　　(b) ++i形式　　(c) 等价的++i形式

图 6　增量语句的转换

定理 2　形如图 6(a)中的 WHILE_DO 循环可以转换为图 6(c)中语义等价的 WHILE_DO 循环。

证明　采用数学归纳法。设循环(a)执行的次数为 N(N≥0)(比较操作排除语句执行序列外),记 i = i + 1 为语句 INC。

当 N = 0 时,循环(a)语句执行序列为 INC,循环(b)语句执行序列为 INC,两循环语句执行序列相同,语义等价。

当 N = 1 时,循环(a)执行一次,语句执行序列为 INC S,循环(b)中 WHILE_DO 循环条件不满足,但是 if 条件满足,语句执行序列 INC S,两循环语句执行序列相同,语义等价。

当执行次数为 n(N > n)时,此时循环未结束。循环(a)的语句执行序列为 INC S(n 个),循环(b)的语句执行序列为 INC S(n 个),两循环语句执行序列相同,语义等价。

当执行次数 n + 1 时,分为两种情况:

第 1 种情况　n < N - 1 时,循环(a)的语句执行序列为 INC S(n + 1 个),循环(b)的语句执行序列为 INC S(n + 1 个),两循环语句执行序列相同,语义等价;

第 2 种情况　n = N - 1 时,由于前 N - 1 次执行序列相同,因此只要第 N 次序列相同即可。循环(a)第 N 次执行的序列为 INC S,而循环(b)中 WHILE_DO 循环执行 INC 后退出 WHILE_DO 循环,而 if 的条件满足因此还要执行语句 S,所以循环(b)的执行序列也为 INC S,两循环语句执行序列相同语义等价。

综上循环(a)与循环(b)等价。　　证毕。

SPEC2000 测试集 253. perlbmk 中 pp. c 中一个 WHILE_DO 循环如图 7(a)所示,将其转换后的形式如图 7(b)所示。

2.4　中间表示为 DO_WHILE 的循环正规化

由于 DO_WHILE 循环可以转换为语义等价 WHILE_DO

循环。为了发掘 DO_WHILE 的并行性,将 DO_WHILE 转换为 WHILE_DO,然后再将 WHILE_DO 转换为 DO_LOOP。

```
while(--len > 0) {                    if (bits & 1) culong++;
                                        bits >>= 1;}
while(len--> 0) {                     if((len+1)>0){
    if (bits & 1) culong++;               if (bits & 1) culong++;
      bits >>= 1;}                          bits >>= 1;}
```

(a) i--形式　　(b) 等价的--i形式

图 7　增量语句转换的实例

定理 3　形式为 do{S1}while(S0)的 do-while 循环,其语义等价的 while 循环形式为 S1while(S0){S1},其中 S1 是循环体内语句集合,而 S0 为循环条件。

证明　采用数学归纳法,假设循环体执行的次数为 N(N≥1)。

当 N=1 时 do-while 循环语句序列为 S1,而 while 循环语句序列为 S1,语义等价。(n-1 个)

当循环体执行为 n(N>n>1)时,do-while 循环语句序列为 S1(S0S1)(S0S1)…(S0S1),而 while 循环语句序列为 S1(S0S1)(S0S1)…(S0S1),语句序列相同。

当循环体执行为 n+1 时,do-while 循环语句序列为 S1(S0S1)(S0S1)…(S0S1),而 while 循环语句序列为 S1(S0S1)(S0S1)…(S0S1),语义等价。　证毕。

SPEC2006 测试集 435. gromacs 中 random. c 中一个 do-while 循环如图 8(a)所示,将其转换后的形式如图 8(b)所示。

```
do{                                   v1 = 2.0*rando(ig)-1.0;
  v1 = 2.0*rando(ig)-1.0;             v2 = 2.0*rando(ig)-1.0;
  v2 = 2.0*rando(ig)-1.0;             r = v1*v1 + v2*v2;
  r = v1*v1 + v2*v2;                while (r >= 1.0){
} while (r >= 1.0);                   v1 = 2.0*rando(ig)-1.0;
                                      v2 = 2.0*rando(ig)-1.0;
                                      r = v1*v1 + v2*v2; }
```

(a) do-while循环　　(b) 等价的while-do循环

图 8　do-while 转换为 while 的实例

本章叙述了目前 Open64 编译中 WHILE_DO 循环转换为 for 循环的条件,并解决了目前 Open64 中不能转换的几种情况。算法实现过程中仅涉及到拷贝等简单的中间 whirl 节点操作,因此未详细描述。下章处理未能转换为 DO_LOOP 循环的 WHILE_DO 循环,主要通过循环展开压紧发掘其 SIMD 向量化特性。

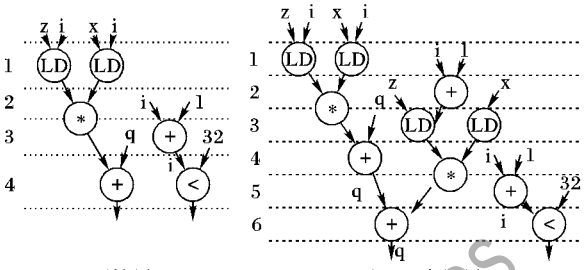
3　非正规化循环的展开压紧

循环展开压紧是一种很有效的循环优化技术[13]。它不仅可以提高高速缓存 cache 的局部性和指令级并行,同时还可以带来如寄存器重用等其他优化。如图 9(a)所示的循环将其循环展开一次得到图 9(b)所示形式。

假设计算部件有一个加法器,一个乘法器,一个比较器,两个内存取单元。除了乘法部件需要两个周期,其余都需要一个周期。取单元包括计算地址和读操作,并且可以同时取。那么如图 9(a)所示未循环展开大约需要 32×4=128 个周期,而如图 9(b)所示循环展开两个需要 16×6=96 个周期。所以当展开因子 VF=2 时性能大约提升 25%。

但是以往研究循环展开都是基于 DO_LOOP,如:Ferrer 等[14]提出了面向 OpenMP 任务级并行的循环展开;文献[15]解决了自动选择最优的循环展开因子。循环体内是数组操作的 DO_LOOP 很容易展开的原因是:1)维数、上下界等数组信息编译时是确定的;2)索引变量在循环体内值不变;3)循环只有一个出口。和 DO_LOOP 不同,WHILE_DO 循环由于上述三个因素不确定而难以进行循环展开,但是其主要原因还是不能确定循环结束条件。

```
i=0;                                  i=0;
while(i<32){                          while(i<32;){
    q+=x[i]*z[i];                         q+=x[i]*z[i];
    i+=1; }                               q+=x[i+1]*z[i+1];
                                          i+=2;}
```



(a) 原始循环　　(b) 展开一次的循环

图 9　循环展开前后执行周期

本文提出一种非正规化循环的展开压紧算法。假设 WHILE_DO 循环结构为 while B do S 形式,其中称 B 为循环条件,称 S 为循环体。初始的循环条件称为 B_1,初始循环条件 B_1 对应的第一次语句执行序列称为 S_1。那么 B_1 经过 WHILE_DO 循环体 S_1 得到第二次循环条件 B_2,记作 $B_2=W(S_1,B_1)$。而第二次循环条件 B_2 对应的第二次语句执行序列 $S_2=W(S_1,B_2)$。以此类推能够得到第 u 次的循环条件 $B_u=W(S_{u-1},B_{u-1})$,以及第 u 次语句执行序列 $S_u=W(S_{u-1},B_{u-1})$。因此能够对形如 while B do S 的不规则循环进行循环展开。对其进行循环展开 u 次的形式如图 10 所示,其中 A 表示逻辑与。因此从理论上对于给定的 WHILE_DO 循环可以通过循环展开压紧取得加速效果。但实际上这种循环变化能力可能有限,原因是受到迭代间的数据依赖关系或者循环体内语句的影响,此外有些循环条件条件可能不能简单地用逻辑等价来替换,可能导致运行时错误甚至结果错误不正确。

```
while(B1 A B2 A…A Bu) do
    S1
    S2
    …
    Su
while B do S; //尾循环处理
```

图 10　WHILE_DO 循环的展开

为了保证循环展开的正确性,需要对 WHILE_DO 循环加以限定:1)循环体为单个基本块;2)循环体内不存在方向向量为(<,>)的依赖;3)单循环条件且为 variant OP invariant 或者 invariant OP variant 类型。算法如图 11 所示,可以分为如下几个步骤:

1)获得待展开循环的初始循环条件 B_1 及其对应的语句执行序列 S_1,然后递归地计算第 i($1\leqslant i\leqslant u$,u 为展开因子)次循环的循环条件 B_i 及其语句执行序列 S_i。

2)简化展开后形如 B_1A B_2A…A B_u 的循环条件。假设展开一次,$B_1=(i<N)$,$B_2=(2*i<N)$,因其为直接内存访问,在编译阶段可将 B_1A B_2 简化为 B_2。而对间接内存访问如 $B_1=(a[i]<N)$,$B_2=(a[2*i]<N)$则不能简化循环条件。

3)压紧展开后循环体内的规约语句。假设展开前循环体内含有一条语句 i=i+1,展开一次含有两条语句 stmt1;i=i+1,stmt2;i=i+1。将 stmt1 和 stmt2 压紧后变为 i=i+2。

4)将展开后的循环条件和循环体合并到一起得到 WHILE_DO 循环展开结果。

5)尾循环处理。

为了更清晰地说明 WHILE_DO 展开压紧过程,选用 NPB 测试集中程序 FT 中的一个 WHILE_DO 循环进行说明。按照图 11 步骤循环展开过程如图 12 所示。

```
1)   procedure UnrollAndJam
2)    Input:需要进行展开压紧的非规则循环L和展开次数u
3)    Output:展开u次的非规则循环L'
4)    获取循环L的初始循环条件B1及其对应的语句执行序列S1
5)    获取循环条件的访存类型AccessMem_kind及其操作数类型
      type_kind
6)    存放所有循环条件的数组B_Array及其对应语句执行序列
      数组S_Array
7)    B_Array[1]=B1, S_Array[1]=S1;
8)    for each i∈(2,u) do      //step 1 计算第i次循环条件 Bi及
9)     | Bi=ComputeBi(Bi-1,Si-1); //其对应的语句执行序列 Si
10)    | Si=ComputeSi(Bi,Si-1);
11)    └B_Array[i]=Bi, S_Array[i]=Si;
12)   B_Array_Simplify= SimplifyLoopCondition(B_Array)
                                        //step 2 简化循环条件
13)   S_Array_Red=ReducteLoopBody(S_Array)
                                        //step 3 规约循环体内语句
14)   L1=GenerateUnrollBody(B_Array_Simplify , S_Array_Red)
                                        //step 4 生成展开后的循环
15)   L'=L Merge L1;                    //step 5 尾循环处理
16)     return L';
17)  end UnrollAndJam
18)  function ComputeBi(Bi-1,Si-1)
19)   获取Bi-1的操作数variant0 和variant1;
20)   if variant0 是一个变量then
21)    └ iv = variant0;
22)   else
23)    └ iv = variant1;
24)   获取Si-1中对iv的定义记为incr_stmt;
25)   将循环条件Bi-1中iv的替换为incr_stmt并记为Bi;
26)  return Bi;
27)  function ComputeSi(Bi,Si-1);
28)   获取Bi中的变量记作iv;
29)   获取Si-1中iv的定义-使用关系记为Def-Uselist;
30)   根据Def-Uselist将Si-1中所有的使用替换为iv并记为Si;
31)  return Si;
31 function GenerateUnrollBody(B_Array3,S_Array3)
32)   B=B_array3[1];
33)   S=S_Array3[1];
34)   for each i ∈(2,u) do
35)    | B=BMergeB_array3[i];    //生成循环展开后的循环条件
36)    └ S=S Merge S_Array3[i];  //生成循环展开后的循环体
37)   loop=B Merge S;                  //生成展开后的循环
38)  return loop;
39)  function SimplifyLoopCondition(B_Array2)
40)   if访存类型为直接内存访问then
41)    └ B_Array1[1] = B_Array2[u];
                                  //简化循环条件为第u次循环条件
42)   else
43)    | for each i∈(1,u) do
44)    └ └B_Array1[i] = B_Array2[i];
                                  //其他情况循环条件无法简化
45)  return B_Array1;
46)  function ReducteLoopBody(S_Array2)
47)   获取循环体的第一条语句记为stmt;
48)   while stmt!=NULL do
49)    | if stmt是规约语句then
50)    | | 获取stmt的规约变量记为Red_var;
51)    | | 获取Red_var在S_Array2内的定义-使用关系记为
       | |  Def_Uselist;
52)    | | 用Hash表存放规约变量及其对应的定义-使用关系;
53)    | stmt=stmt ->next;
54)   根据Def_Uselist将定义语句带入到使用语句中并记
      替换后的循环体为S_Array1;
55)  return S_Array1;
```

图 11　WHILE_DO 循环展开算法

DO_LOOP 循环利用此方法同样可以产生正确的循环展开结果。展开后的 WHILE_DO 循环循环体内含有足够的并行性。2000 年 Samuel Larsen 提出了面向 SIMD 体系结构的超字并行(SLP)向量化算法,目前 SLP 算法已成为发掘 SIMD 并行性的主要算法。SLP 一般需要经过以下几个步骤:

1)循环展开;

2)对齐分析及死代码消除等预优化操作;

3)按照地址相邻的原则生成初始的 pack;

4)根据 DU 和 UD 链对 pack 进行扩展;

5)按照依赖关系对 pack 进行调度。

为了让基本块内含有足够并行可能,SLP 算法首先进行循环展开。由于不支持 WHILE_DO 循环展开,因此 SLP 算法不能发掘 WHILE_DO 循环的 SIMD 向量化。现在提出了 WHILE_DO 循环的展开方法,因此可以按照 SLP 算法对 WHILE_DO 循环进行 SIMD 向量化。

```
步骤1) 循环展开前          步骤2) 计算B2、S2          步骤3) 简化B1 A B2
do while(nn .lt. n)        B1: nn .lt. n              简化nn .lt. n && nn*2 .lt. n
nn = nn*2                  S1:nn = nn*2               为nn*2 .lt. n
lg = lg+1                    lg = lg+1
end do                     B2: nn*2 .lt. n
                           S2: nn = nn*2
                             lg = lg+1

步骤4) 压紧S1, S2          步骤5) 生成展开后循环      步骤6) 循环展开后
压紧前S1, S2:              do while (nn*2 .lt. n)     do while (nn*2 .lt. n)
  nn = nn*2                  nn = nn*4                  nn = nn*4
  lg = lg+1                  lg = lg+2                  lg = lg+2
  nn = nn*2                end do                     end do
  lg = lg+1                步骤5: 尾循环处理          do while (nn .lt. n)
压紧后S1, S2:              do while (nn .lt. n)         nn = nn*2
  nn = nn*4                  nn = nn*2                  lg = lg+1
  lg = lg+2                  lg = lg+1                end do
                           end do
```

图 12　用例说明循环展开过程

4　实验结果和分析

基于 Open64 实现了本文提出的非正规化循环的 SIMD 向量化算法,在 IBM x3650 上进行实验,实验平台采用 Intel 至强处理器 5500,内存 4 GB。处理器的向量化寄存器长度为 128 b,可以同时处理 4 个整型数据或者 2 个浮点型数据。为便于向量化发掘,整型数据展开因子为 4,浮点类型展开因子为 2。SPEC 是一个致力于发布管理计算机性能标准化测试的组织,本文选用 SPEC CPU2000、SPEC CPU2006 和 SPEC MPI2007 中部分程序进行测试,选用测试用例如表 1 所示。

表 1　选定的测试集

程序名	来源	功能
gap	SPEC CPU2000	群论
perlbmk	SPEC CPU2000	Perl 语言解释器
hmmer	SPEC CPU2006	基因序列搜索
sphinx3	SPEC CPU2006	语音识别
RAxML	SPEC MPI2007	XML 处理

采用两遍编译方式,即:第一遍将源程序进行源源翻译,生成高级语言程序;第二遍再将翻译结果提交基础编译器,然后在测试平台上,运行获得向量化运行时间。直接利用基础编译器生成可执行文件在测试平台上运行获得串行时间。验证其运行结果是否与直接编译源程序的结果一致(认为基础编译器是可靠的)。用串行时间除以并行化时间得到程序并行化的加速比。比较实现非规则循环的 SIMD 向量化发掘方法前后的识别率以及加速比。表 2 给出了改进前后 SIMD 向量化的识别率变换情况。表 2 中没有统计程序中已有 DO_LOOP 循环,仅统计其中 WHILE_DO 循环。按照图 3 叙述的发掘流程首先将 WHILE_DO 循环转化为 DO_LOOP 循环(表中步骤 1),此过程对应图 3 中编号①。不能转化为 DO_LOOP 循环的 WHILE_DO 循环如满足条件进行循环展开(表中步骤 2),此过程对应图 3 中编号②。最后利用超字并行 SLP 算法生成 SIMD 向量化代码(标记为步骤 3),此过程对应

图3中编号③。以程序gap为例:改进前有7个WHILE_DO循环转化为DO_LOOP循环,程序中剩下WHILE_DO循环中展开的个数为0,最后生成SIMD向量化代码的WHILE_DO循环有1个;而改进后有10个WHILE_DO循环转化为DO_LOOP循环,程序中剩下WHILE_DO循环中展开的个数为6,最后生成SIMD向量化代码的WHILE_DO循环有3个。

表2 选定测试集中非正规化循环的SIMD向量化

程序名	改进前			改进后		
	步骤1	步骤2	步骤3	步骤1	步骤2	步骤3
gap	7	0	1	10	6	3
Perlbmk	8	0	1	12	5	1
Hmmer	3	0	1	5	2	1
sphinx3	2	0	0	4	4	4
RAxML	4	0	1	6	8	3

选定的测试用例都通过了正确性测试。图13列出了现有方法和本文算法的加速比,从实验结果可以看出改进后的方法取得了一定的作用。改进前gap的加速比1.03,而改进后加速比为1.12,改进后获得更好性能是因为被向量化循环的个数由改进前的1变为了3,且被向量化的循环所占的运行时间比重较大。虽然Perlbmk改进前后被SIMD向量化的循环个数都为1,但改进后WHILE_DO循环展开的个数由0变为了5,因此加速比由1.03提升到1.05。程序Hmmer改进前加速比为1.08,改进后加速比仍为1.08,因为改进后被向量化循环的个数与改进前相同,虽然有2个WHILE_DO循环被展开但这2个循环不是核心循环,因此改进后没有获得加速效果。sphinx3和RAxML改进前的加速比分别是1.05和1.02,而改进后的加速比分布为1.15和1.10,一是因为改进前SW-VEC生成的SIMD向量化代码很少,而采用本文提出的WHILE_DO发掘方法后生成了较多的SIMD向量化代码;二是因为改进后展开压紧了一些WHILE_DO循环。选定测试程序改进前的平均加速比为1.04,而改进后平均加速比为1.10。自动生成的SIMD向量化代码性能加速比提高了6%以上。所以,本文提出的WHILE_DO循环并行识别方法是可行并且有效的。

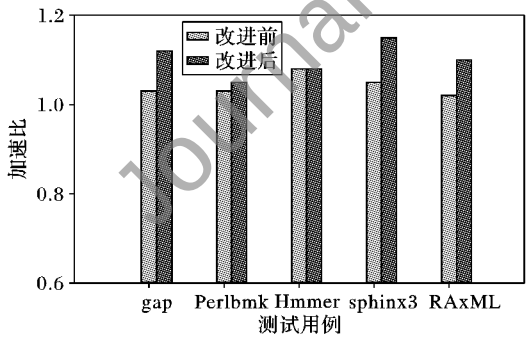


图13 非正规化循环的SIMD向量化实验结果

5 结语

非正规化循环广泛存在于应用程序中,目前对其并行发掘的研究很少。本文首先根据循环条件将非正规化循环进行分类,分析了各类循环的并行发掘方法。目前只能将一些简单的非正规化循环正规化,这大大降低了非正规化循环的并行性。本文首先放宽了正规化循环的条件,对不能正规化的循环提出了循环展开方法。实验结果表明这些方法取得了很好的加速效果。OpenMP规范3.0中给出了任务并行的规范,如何在非正规化循环中发掘任务级并行是将下一步工作的主要内容。函数调用是制约发掘程序向量化的主要因素,如何发掘带有函数调用循环的SIMD向量化特性也是下一步的工作。

参考文献:

[1] ALLEN R, KENNEDY K. 现代体系结构的优化编译器[M]. 张兆庆,乔如良,冯晓兵,等译. 北京:机械工业出版社,2004.

[2] AYGUADE E, COPTY N, DURAN A, *et al.* The design of OpenMP tasks [J]. IEEE Transactions on Parallel and Distributed Systems, 2009, 20(3), 404 - 418.

[3] LENGAUER C, GRIEBL M. On the parallelization of loop nests containing while loops [C]// Proceedings of the First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis. Washington, DC: IEEE Computer Society, 1995: 10 - 18.

[4] COLLARD J F. Space-time transformation of while-loops using speculative execution [C]// Proceedings of the 1999 Scalable High-Performance Computing Conference. Piscataway: IEEE Press, 1994: 37 - 42.

[5] GRIEBL M. The mechanical parallelization of loop nests containing while loops [D]. Passau, Germany: University of Passau, 1996.

[6] LARSEN S, RABBAH R, AMARASINGHE S. Exploiting vector parallelism in software pipelined loops [C]// Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC: IEEE Computer Society, 2005: 50 - 58.

[7] LARSEN S, AMARASINGHE S. Exploiting superword level parallelism with multimedia instruction sets [C]// Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. New York: ACM Press, 2000: 145 - 156.

[8] 魏帅,赵荣彩,姚远. 面向SLP的多重循环向量化[J]. 软件学报, 2012, 23(7): 1717 - 1728.

[9] LIU J, ZHANG Y R, JANG O, *et al.* A compiler framework for extracting superword level parallelism [C]// Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2012: 59 - 64.

[10] KONG M, VERAS R, STOCK K, *et al.* When polyhedral transformations meet SIMD code generation [C]// Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2013: 45 - 55.

[11] GEUNS S J, BEKOOIJ M J G, BIJLSMA T, *et al.* Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems [C]// Proceedings of the 2011 Design, Automation & Test in Europe Conference & Exhibition. Piscataway: IEEE Press, 2011: 1 - 6.

[12] 郝云龙,赵荣彩,侯永生,等. 反馈式编译技术在循环级性能分析中的应用[J]. 计算机工程, 2011, 37(9): 32 - 36.

[13] GIRBAL S, VASILACHE N, BASTOUL C, *et al.* Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies [J]. International Journal of Parallel Programming, 2006, 34(3): 261 - 317.

[14] FERRER R, DURAN A, MARTORELL X, *et al.* Unrolling loops containing task parallelism [C]// Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing. Berlin: Springer-Verlag, 2009: 13 - 21.

[15] SARKAR V. Optimized unrolling of nested loops [C]// Proceedings of 14th International Conference on Supercomputing. New York: ACM Press, 2000: 153 - 166.