循环展开技术在向量程序中的应用

高 伟 赵荣彩 于海宁 张庆花
(信息工程大学 郑州 450001) (数学工程与先进计算国家重点实验室 无锡 214125)

摘 要 循环展开是一项常用的循环优化技术。当前针对串行程序的循环展开技术已经比较成熟,但是在实际应用中没有针对向量程序进行有效的循环展开。为了解决这个问题,提出了一种面向向量程序的循环展开技术。首先,针对向量寄存器压力和代码膨胀等限制因素,提出了一种自动计算展开因子的CUFVL算法;其次,根据向量循环展开的特点,制定了完全展开策略;最后结合CUFVL算法和完全展开策略,设计了向量循环展开的总体流程。实验结果表明,该方案能够计算出合适的展开因子,进而对向量程序进行适当的循环展开或完全展开,从而有效提升应用程序的性能。

关键词 向量程序,循环展开,展开因子,完全展开
中图法分类号 TP312 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.1.049

Loop Unrolling in Vectorized Programs

GAO Wei ZHAO Rong-cai YU Hai-ning ZHANG Qing-hua
(PLA Information Engineering University, Zhengzhou 450001, China)
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China)

Abstract Loop unrolling is a loop transformation that has been developed well and widely used in serial programs. However, it usually loses its efficiency in practical vectorized applications. To solve this problem, this paper proposed a loop unrolling technique for vectorized loops. First, we presented the CUFVL algorithm to automatically determine the unroll factors via taking the register pressure and code explosion problems into account. Second, we designed a complete unrolling strategy according to the specific characters of vectorized loops. Finally, we showed the flow chart of the proposed technique based on the CUFVL algorithm and the complete unrolling strategy. Experimental results show that the proposed technique can find out the most suitable unroll factor for the programs used in the experiments, thereby enhancing their performance efficiently.

Keywords Vectorized programs, Loop unrolling, Unroll factor, Complete unrolling

1 引言

在加速和优化程序运行时间的努力中,计算机科学家一直在尝试各种各样的技术来提高计算机每个时钟周期中的操作个数。循环展开[1]在浮点算术和线性代数中的应用是其中最成功的技术之一。循环展开通常用来降低循环开销[2],为具有多个功能单元的处理器提供指令级并行[3],也有利于指令流水线的调度和更好地实现数据预取技术[2,4]。

由于多媒体应用的代码往往具有较高的并行性[5]且21世纪多媒体产业迅猛发展,目前几乎所有主流的处理器厂商都为其微处理器增加了SIMD(Single Instruction Multiple Data)扩展部件,以充分发掘微处理器的计算能力。SIMD扩展的出现为提升程序性能提供了硬件支持,但要充分发挥SIMD扩展的性能,在程序中还需要有适于SIMD功能部件执行的向量化代码。串行程序的向量化主要有两种方法,一种是基于编译器的自动向量化[6],另一种是手工向量化[7]。虽然手工向量化能够充分挖掘程序的并行性,并且能够从算法级对程序结构进行优化,从而得到更优的性能加速效果,但是手工向量化要求程序员熟练掌握SIMD扩展的体系结构和指令集特点并深入了解程序结构,而且手工改写的向量化代码可重用程度不高,可读性较差,编写大规模程序时实现难度大且极易出错。因此,利用并行化编译器自动生成向量化程序是解决串行程序向量化的一种有效途径。目前很多编译器都集成了自动向量化模块,如Intel编译器[8]、GCC编译器[9]、Pathscale EKOPath编译器[10]等。

为了充分利用国产高性能处理器的向量部件,课题组基于开源编译器Open64-5.0[11]设计并研发了一款面向国产高性能处理器的自动向量化工具SW-VEC。Open64-5.0是当

到稿日期:2014-11-13 返修日期:2015-03-30 本文受"核高基"国家科技重大专项(2009ZX01036-001-001-2),数学工程与先进计算国家重点实验室开放课题(2013A11)资助。

高 伟(1989—),男,硕士生,主要研究领域为先进编译技术,E-mail:huadiangao@163.com(通信作者);赵荣彩(1957—),男,博士,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;于海宁(1989—),男,硕士生,主要研究领域为先进编译技术;张庆花(1991—),女,硕士生,主要研究领域为先进编译技术。

前最先进的产品级"源-源"编译器，能够兼容 gcc/g++并能与之交互工作，易扩展和移植。使用 SW-VEC 编译器对程序实施自动向量化，能够生成高效的向量化代码。然而，Open64-5.0 设计的重点是用来发掘程序中并行性的循环交换、循环合并、循环分布和幺模变换等循环变换以及其他优化技术，这些优化都位于向量化模块之前，并没有对向量化后的代码进行循环优化。因此，为了能够进一步提升 SW-VEC 编译器自动向量化的优化效果，本文设计并实现了一种面向向量程序的循环展开算法，通过考虑向量寄存器压力和代码膨胀等限制因素来自动确定展开因子，从而对向量化后的代码进行适当的循环展开。

本文第 2 节介绍了相关工作；第 3 节分析了向量循环展开的特点以及需要注意的一些问题；第 4 节介绍了计算展开因子的 CUFVL（Compute Unroll Factor for Vectorized Loop）算法和向量循环展开的具体流程；第 5 节是实验结果及分析；最后总结全文。

2 相关研究

循环展开是一项常用的程序优化技术，其初始动机是减少循环开销。对于当前的处理器，循环展开的主要收益包括提高指令级并行、寄存器局部性和层次化存储局部性[12]。同时，循环展开也是高效开发某些硬件特性的必要手段，比如发掘生成双重指令的机会[13]，或者通过多条装载/存储指令抵消单个预取指令的开销[14]。然而，不恰当的展开可能会给程序性能带来负面收益[15]，比如过度展开会导致额外的寄存器溢出，从而使程序的运行性能降低。另一方面，过激的循环展开会引起指令缓存区溢出，因为循环体经过展开后在源代码级别就已经变得很大，当它们被翻译成目标代码时代码规模会非常庞大。

循环展开的核心问题是如何确定展开因子。普遍采用的方法是构建代价模型自动地计算展开因子[1,16-18]。Sarkar[1]针对多重完美嵌套循环，通过代价函数评估循环展开后程序性能的提升来确定展开因子，其算法对 SPEC95pf 测试集中的 7 个例子有平均 1.08 倍的加速效果。Yin Ma 等[16]提出了一种低代价(Low Cost)方法，即通过考虑标量替换、常规标量优化、软件流水和寄存器分配共同作用的影响，在实施展开和压紧之前预测循环的寄存器压力，然后利用预测结果来自动确定展开因子，从而获得最好的运行性能。

另一种方法是利用监督学习技术(Supervised Learning Techniques)来确定展开因子[19,20]。Monsifrot 等使用一种基于 BDT(Boosted Decision Tree)学习技术的映射机制来确定对哪些循环进行展开[20]。Mark Stephenson 等收集了与循环展开相关的 38 种程序特征，利用 NN(Near Neighbor)和 SVM(Support Vector Machines)这两种机器学习技术构建程序特征与展开因子间的映射关系[19]。

3 向量循环展开的相关问题

3.1 依赖分析

对于 SW-VEC 编译器而言，向量化分为传统向量化模块和 SLP 向量化模块。SLP 向量化模块由于是用于发掘基本块内的并行性的，因此不适合进行循环展开。而在面向循环的传统向量化模块中，为了保证向量化的正确性，已经剔除了包含数据依赖以及函数调用等不适合向量化的循环，因此向量循环展开要比标量循环展开更为简单。

循环展开是一种保持语句执行顺序的循环变换，所以一般不需要像其他循环变换一样进行依赖分析。但是在向量化的依赖分析中有一类特殊情况，当依赖距离 d 不小于向量因子 VF 时，循环是可以被向量化的，因此要考虑这类依赖关系是否会影响到循环展开。为了使分析过程更为直观，不妨设 $d=2, VF=2$。如图 1 所示，虽然图 1(a)中的循环存在依赖，但是依赖距离等于向量因子，因此不影响向量化，向量化结果如图 1(b)所示。对向量化后的循环进行两次展开的结果如图 1(c)所示。展开后循环内存在语句 S1 到语句 S2 的真依赖，或者分别截取串行程序的前 4 次迭代运算与循环展开后循环的第一次迭代运算进行比较，可以发现二者的运算次序完全一致。因此，只要在循环展开的后续优化中不再对循环体复制块间的语句进行重排序，就能够保持原有的依赖关系。

```
for(i=2;i<34;i++)
  a[i]=a[i-2]+b[i];
```
(a)串行程序

```
for(i=2;i<34;i+=2){
  V1= simd_load(a[i-2]);
  V2= simd_load(b[i]);
  V3= V1+V2;
  simd_store(a[i],V3);
}
```
(b)向量化

```
for(i=2;i<34;i+=4){
  V1= simd_load(a[i-2]);
  V2= simd_load(b[i]);
  V3= V1+V2;
S1  simd_store(a[i],V3);
S2  V4= simd_load(a[i]);
  V5= simd_load(b[i+2]);
  V6= V4+V5;
  simd_store(a[i+2],V6);
}
```
(c)循环展开

```
a[2]=a[0]+b[2];
a[3]=a[1]+b[3];
a[4]=a[2]+b[4];
a[5]=a[3]+b[5];
```
(d)串行程序的前 4 次迭代运算

```
a[2]=a[0]+b[2];
a[3]=a[1]+b[3];
a[4]=a[2]+b[4];
a[5]=a[3]+b[5];
```
(e)展开后的 1 次迭代运算

图 1 数据依赖距离不大于展开因子的循环的向量化及循环展开

3.2 特殊循环结构

循环展开常用于归约变量的优化，在向量化的过程中，归约、归纳等循环是作为特殊的循环结构进行处理的。归约向量化结果如图 2(c)所示，将数组 b 中的变量每两个作为一个向量装载到 $V2$ 中进行累加运算，最后将累加向量 $V1$ 内的操作数进行累加即可得到目标结果，这与串行程序的循环结构相同。因此，这类特殊的循环结构同样可以循环展开。

```
for(i=0;i<32;i++)
  a+= b[i];
```
(a)串行程序

```
for(i=0;i<32;i+=2)
  a+= b[i]+b[i+1];
```
(b)规约变量优化

```
V1=0;
for(i=0;i<32;i+=2){
  V2=simd_load(b[i]);
  V1 += V2;
}
a=reduce_add(V1);
```

(c)向量化

图 2　归约循环的归约变量优化及向量化

对于例 1 这种包含分支的循环，考虑到软件流水中旋转寄存器的特点，需要对分支上的定值操作添加额外的拷贝操作，因此在展开时需要适当减少展开次数，以避免循环对拷贝操作的过度依赖[2]。使用 SW-VEC 编译器对该循环进行向量化，得到例 2 所示的代码。select 指令的引入使得条件分支语句转换为普通的向量运算指令，因此在对向量化循环进行展开时，不需要考虑分支语句的影响。

例 1
```
for (i=0;i<N;i++){
  if (c[i]<0)
    a[i]=b[i];
  else
    a[i]=c[i];
}
```

例 2
```
V_0= type_scp(0.0);
for (i=0;i<N;i+=4){
  V_c=simd_load(c[i]);
  V_b=simd_load(b[i]);
  V_cmp=simd_vfcmple(V_c,V_0)
  V_a=simd_vseleq(V_cmp,V_c,V_b)
  simd_store(a[i],V_a);
}
```

3.3　寄存器分配

进行展开时，需要考虑寄存器压力对展开因子的限制。对于标量循环，标量寄存器不仅要参与循环体内的运算和访存操作，还要用来存储循环索引变量；同时，在大多数计算环境中还要保留一些标量寄存器，用作基址寄存器、栈顶指针寄存器或其他类似的用途。

在向量循环中，除了循环索引用到的标量寄存器外，循环体内的运算和访存操作使用的都是向量寄存器，而向量寄存器通常不会有其他用途，因此在分析寄存器压力时只需要考虑当前循环对向量寄存器的需求。

4　面向向量程序的循环展开

4.1　计算向量展开因子

向量化后的循环各次迭代之间通常是相互完全独立的，具有很大的并行性。例 3 是一个典型的循环结构。

例 3
```
for(i=0;i<32;i++)
  a[i]=b[i]*c[i]+r;
```

向量化结果如下：

例 4
```
V1=type_scp(r);
for(i=0;i<32;i+=2){
  V2=simd_load(b[i]);
  V3=simd_load(c[i]);
  V4=V2*V3;
  V5=V4+V1
  simd_store(a[i],V5);
}
```

将目标机器用一个简单模型代替，在这个模型中，可以在同一个时钟周期内发出一个向量加载指令、一个向量存储指令、一个向量算术运算；算术运算是完全流水线化的。每个时钟周期可以启动一个算术运算，但是运算结果要到 2 个时钟周期后才可用。其他指令的执行延时为一个时钟周期。

应用上述模型，例 4 中的循环每次迭代需要 7 个时钟周期。如果将例 4 中的循环展开 4 次，然后使用一个简单的列表调度算法来对这些运算进行调度，那么就可以得到图 3 显示的调度方案，展开后总共需要 13 个时钟周期，平均每次迭代只需要 3.25 个时钟周期。如果按 k 次展开，循环体则至少需要 $2k+5$ 个时钟周期，平均每次迭代需要 $2+5/k$ 个时钟周期，因此展开的次数越多，循环就运行得越快。

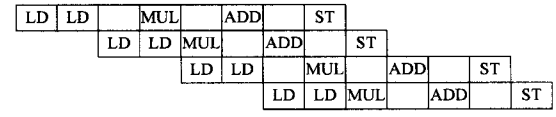

图 3　例 4 的指令流水调度

从降低循环开销和提高指令级并行这两方面考虑，展开因子当然是越大越好，但是过度展开也会引起寄存器溢出和指令缓存区溢出等负面影响，因此首先要考虑限制展开因子的因素。对于特定的体系结构和编译技术，对不同特性的循环结构，在展开时所选择的展开因子是不同的。在对串行程序的展开中，限制展开因子的因素主要有以下 3 方面[2]。

因素 1：循环本身是否包含分支；

因素 2：循环自身的寄存器需求；

因素 3：循环体中的操作个数。

对于因素 1，由 3.2 节分析可知，对向量循环进行展开时不会遇到包含分支语句的循环，因此不需要考虑因素 1 的限制。

对于因素 2，考虑图 3 中的调度情况，按照不影响指令流水并行性且使用最少寄存器的分配原则，可以得到图 4 显示的向量寄存器分配方案，加上循环外存储不变量的 1 个寄存器，只需要 7 个向量寄存器就可以完成该调度；而且随着展开次数继续增加，7 个向量寄存器仍然可以满足调度需求。因此，对于有 32 个向量寄存器的平台来说，几乎不会出现向量寄存器溢出的情况，因此也不需要考虑因素 2 的限制。

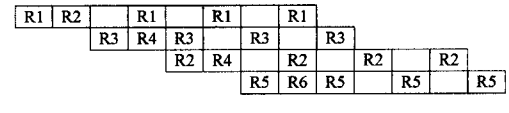

图 4　图 3 中调度的一种寄存器分配方案

对于因素 3，虽然理论上展开的次数越多，原有循环迭代的平均周期就越短，但是我们不得不考虑代码膨胀的代价，因此在展开时需要考虑对展开后语句条数的限制。

结合以上分析，设计 CUFVL 算法流程如下。

输入：需要展开的向量化后的循环 innerloop

输出：自动计算的展开因子 UF

算法：

```
Compute_Unrollfactor (innerloop){
  SL← sl_init;//设定的语句条数限制
  N_Stmt ← 0;
```

```
//统计循环内语句条数
for(stmt=first_stmt(innerloop);;stmt=next_stmt(stmt)){
    N_Stmt++;
    if (stmt == last_stmt(innerloop))
    break;
}
//计算展开因子上限
UFU ← SL/N_Stmt;
loop_end ← Loop_End(innerloop);
if (Is_constant(loop_end));{
  simd_ite ←Iterations(innerloop);//向量循环迭代次数
  UFU ← min(UFU, simd_ite);
}
//调整展开因子为 2 的整数次幂
unroll_factor ← Regulate_Unrollfactor (UFU);
return unroll_factor;
}
```

算法首先根据循环内语句条数以及设定的语句条数上限 SL 计算出展开因子的上限 UFU，其中 SL 通过实验获得。然后，对于迭代次数已知的循环，调整展开次数不大于迭代次数。最后，为了尽量减少冗余预取的个数，需要把展开因子调整为 2 的整数次幂。

4.2 完全展开

对于迭代次数比较少的循环，可以通过完全展开来消除循环开销。尤其是在循环级向量化中，循环的迭代次数会变为原来的 $1/VF$，迭代次数将大幅度减少，而且尾循环的迭代次数必然要少于 VF，因此在向量化中有很多适合完全展开的循环。

针对不同的循环要采用不同的完全展开策略，主要分为以下两种情况。

a)向量化产生的尾循环：按串行程序的展开标准进行展开；

b)向量化循环：为了避免代码过度膨胀，只对满足条件 $simd\_ite \leqslant UFU+1$ 的循环进行完全展开，其中 $simd\_ite$ 为向量化循环的迭代次数。当 $simd\_ite=UFU+1$ 时，循环会以 UFU 作为展开因子进行展开，且会生成一个迭代次数为 1 的向量化尾循环，与完全展开的代码膨胀程度相同；$simd\_ite=UFU$ 时，展开后的循环中实际上只有一次迭代；而当 $simd\_ite<UFU$ 时，循环适合展开，但是迭代次数不够。因此，对这几类循环进行完全展开，能够有效消除不必要的循环开销。

4.3 向量循环展开的总体流程

为了便于程序优化调试，添加了编译选项 $simd\_unroll$ 来控制循环展开，具体设定为

$$simd\_unroll=\begin{cases}0, & \text{默认值，关闭循环展开}\\ 1, & \text{自动计算展开因子}\\ 2\sim16, & \text{指定展开因子}\end{cases}$$

向量循环展开的具体流程如图 5 所示。当选项 $simd\_unroll=0$ 时，不进入展开流程；$simd\_unroll=1$ 时，采用自动计算的展开因子进行展开，如果迭代次数已知，还要判断是否适合完全展开；$simd\_unroll>1$ 时，采用选项指定的展开因子进行展开。

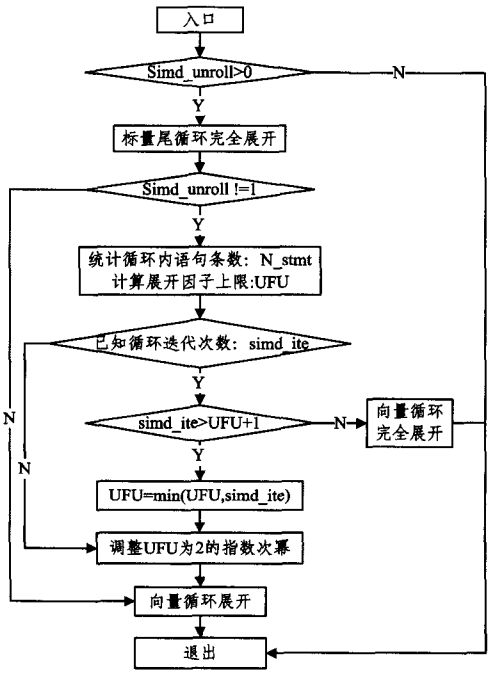

图 5 向量循环展开的总体流程

5 实验评估

本节对 CUFVL 算法及向量循环展开的有效性进行了实验验证。测试平台为 SunWay 系统，编译操作系统环境为 Redhat Enterprise 5，CPU 主频为 2.0GHz，内存为 2GB，L1 数据 cache 为 32kB，L2 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。

5.1 CUFVL 算法的有效性验证

由于测试集中的循环间没有规律性和相关性，不能体现出限制因素对展开因子影响的渐变过程。因此在测试中使用自己构造的例子来验证 CUFVL 算法的有效性。

测试采用的循环如下：

```
for (j=0;j<128;j++)
  for (i=0;i<128;i++){
S1 a[j][i] =b[j][i] * c[j][i];
S2 a[j][i] =b[j][i] * c[j][i] * d[j][i] * e[j][i];
S3 a[j][i] =b[j][i] * c[j][i] * d[j][i] * e[j][i] * f[j][i] * g[j][i];
S4 S3;S3_rename;
  }
```

循环内 4 条语句的操作个数呈递增趋势，基本涵盖了实际程序中大部分语句规模。分别对只包含其中一条语句的循环进行向量化，然后按照 2～16 次进行展开，得到的对应加速比如图 6 所示。其中展开次数为 1 表示向量化后不进行展开的结果。

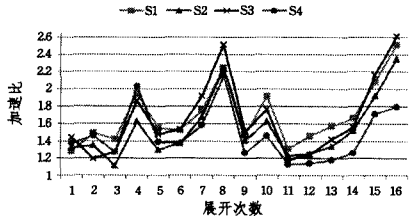

图 6 不同展开次数的加速比

从图 6 可以看出，展开次数为 4、8、16 时为加速比曲线的峰值。对于 S1 和 S2，展开 16 次的加速比要明显高于展开 8 次的加速比。对于 S3，展开 16 次会获得最高加速比，但只是略高于展开 8 次的加速比，而其代码膨胀程度却是后者的 2 倍。对于 S4，展开 16 次要比展开 8 次时的加速效果差很多，说明代码过度膨胀的负面效果抵消了展开带来的收益。

表 1 第 3 列为采用 CUFVL 算法计算出的展开因子，对语句 S1、S2 计算出的展开因子与获得最优加速比的展开因子相同，对语句 S3、S4 计算出的展开因子为最优展开因子的一半，当然代码膨胀也是后者的一半，但获得的加速比只是略低于最优加速比。与不做展开的向量化结果相比，按照由 CUFVL 算法计算出的展开因子对向量化后的循环进行展开，程序性能得到了极大提升。由此可见，CUFVL 算法可以根据所设定的循环内语句限制，自动地算出比较合适的展开因子，由此进行展开不仅避免了代码过度膨胀，也可以有效提升程序性能。

表 1　应用算法前后展开因子及加速比

	不展开	CUFVL 算法		最优加速比	
	加速比	展开次数	加速比	展开次数	加速比
S1	1.28	16	2.51	16	2.51
S2	1.32	16	2.35	16	2.35
S3	1.44	8	2.51	16	2.61
S4	1.37	4	2.03	8	2.15

5.2　向量循环展开的正确性验证

上节内容验证了向量循环展开因子算法的有效性，在此前提下以 456.hmmer 程序为例来验证 4.3 节中提出的循环展开算法的正确性。图 7(a)所示是 456.hmmer 程序中 fast_algorithm.c 文件核心函数的代码，该循环是 456.hmmer 程序的热点循环，所在函数占整个程序总运行时间的 92.15%。

首先，由于循环中部分语句不能被向量化，因此要将其分布出去，结果如图 7(b)所示。在此基础上进行向量化，得到图 7(c)所示的代码。向量循环中共有 27 个操作，计算最多可展开 3 次，取 2 的整数次幂结果为 2，即该向量循环展开两次。按照 4.1 节中的寄存器分配原则进行分配，向量循环展开后共需要 10 个向量寄存器，远小于向量寄存器上限，从而验证了向量循环展开算法的正确性。

```
for (i=1; i <= L; i++) {
  for (k=1; k <= M; k++) {
    mc[k]=mpp[k-1] + tpmm[k-1];
    if ((sc=ip[k-1] + tpim[k-1]) > mc[k]) mc[k]=sc;
    if ((sc=dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k]=sc;
    if ((sc=xmb + bp[k]) > mc[k]) mc[k]=sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k]=-INFTY;
    dc[k]=dc[k-1] + tpdd[k-1];
    if ((sc=mc[k-1] + tpmd[k-1]) > dc[k]) dc[k]=sc;
    if (dc[k] < -INFTY) dc[k]=-INFTY;
  if (k < M) {
    ic[k]=mpp[k] + tpmi[k];
    if ((sc=ip[k] + tpii[k]) > ic[k]) ic[k]=sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k]=-INFTY;
  }
}
```

(a)源代码

```
for (i=0; i < L; i++) {
  for (k=0; k < M; k++) {
    mc[k+1]=mpp[k]+tpmm[k];
    se4[k]=ip[k] + tpim[k];
    se3[k]=dpp[k] + tpdm[k];
    se4[k]=bp[k] +xmb;
    if (k +1< M) {
        ic[k+1]=mpp[k+1] + tpmi[k+1];
        sc=ip[k+1] + tpii[k+1];
        ic[k+1] += MAX(sc, ic[k+1]);
        ic[k+1] += MAX(ic[k+1], -INFTY);
      }
      mc[k+1]=MAX(mc[k+1], se4[k]);
      mc[k+1]=MAX(mc[k+1], se3[k]);
      mc[k+1]=MAX(mc[k+1], se2[k]);
      mc[k+1]=mc[k+1]+ms[k+1];
      mc[k+1]=MAX(mc[k+1], -INFTY);
      se1[k]=mc[k] + tpmd[k];
      }
    for (k=0; k < M; k++) {
      dc[k+1]=dc[k] + tpdd[k];
      dc[k+1]=MAX(se1[k], dc[k+1]);
      dc[k+1]=MAX(dc[k+1], -INFTY);
    }
}
```

(b)循环分布后代码

```
for (i=0; i < L; i++) {
  V_12=simd_loade(xmb);
  V_16=simd_loade(M);
  V_25=simd_loade(-INFTY);
  for (k=0; k < M/4 * 4; k+=4) {
    V_0=simd_load(mpp[k]);
    V_1=simd_load(tpmm[k]);
    V_2=simd_add(V_0, V_1);
    V_3=simd_load(ip[k]);
    V_4=simd_load(tpim[k]);
    V_5=simd_add(V_3, V_4);
    V_6=simd_max(V_2, V_5);
    V_7=simd_load(dpp[k]);
    V_8=simd_load(tpdm[k]);
    V_9= simd_add(V_7, V_8);
    V_10=simd_max(V_6, V_9);
    V_11=simd_load(bp[k]);
    V_13= simd_add(V_11, V_12);
    V_14=simd_max(V_10, V_13);
    simd_storeu(mc[k+1], V_14);
    V_15=simd_set(k+1, k+2, k+3, k+4);
    V_17=simd_vfcmp(V_15, V_16);
    V_18=simd_loadu(mpp[k+1]);
    V_19=simd_loadu(tpmi[k+1]);
    V_20=simd_add(V_18, V_19);
    V_21=simd_loadu(ip[k+1]);
    V_22=simd_loadu(tpii[k+1]);
    V_23=simd_add(V_21, V_22);
    V_24=simd_max(V_20, V_23);
    V_26=simd_max(V_24, V_25);
    V_27=simd_loadu(ic[k+1]);
    V_28=simd_vseleq(V_17, V_27, V_26);
```

```
  }
    //remainderloop
    for(k=M/4*4; k < M; k++){
    ...
  }
for (k=0; k < M; k++) {
  dc[k+1]=dc[k] + tpdd[k];
  dc[k+1]=MAX(sel[k], dc[k+1]);
  dc[k+1]=MAX(dc[k+1], -INFTY);
  }
}
```

(c)向量化后代码

图 7 fast_algorithm.c 函数核心循环代码及其向量化代码

5.3 向量循环展开的性能测试

选择 SPEC2006 测试集中具有不同特点的 5 个程序：410.bwaves、434. zeusmp、437. leslie3d、454. calculix 和 456.hmmer。分别用 icc14.0 编译器和 SW-VEC 编译器对以上 5 个程序进行编译，在 ref 规模下运行，测试结果如图 8 所示。其中系列 1 为 icc14.0 自动向量化的加速比，系列 2 为 SW-VEC 自动向量化的加速比，系列 3 为 SW-VEC 自动向量化结合向量循环展开 SW-VEC+unroll 的加速比。

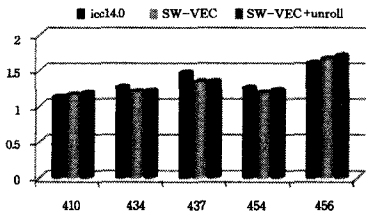

图 8 应用向量循环展开前后与 icc14.0 向量化加速比的对比

对于 410.bwaves 程序，其核心循环内的数组引用相对于次内层迭代具有更好的并行性，因此 icc14.0 和 SW-VEC 都对其进行了循环分布及循环交换来提升向量化性能，二者的加速比分别为 1.12 和 1.16。循环交换后最内层循环只有 5 次迭代，SW-VEC 会将其转换为一次向量循环迭代和一次标量循环迭代，进行完全展开后加速比进一步提升为 1.18。

对于 434. zeusmp 程序，icc14.0 和 SW-VEC 的加速比分别为 1.26 和 1.20。SW-VEC 加速比较低是因为 434. seusmp 中存在大量非对齐访存，AVX 指令集中硬件支持非对齐访存，其效率接近对其访存，而 SunWay 平台只能通过移位重组实现非对齐访存。434. seusmp 核心函数中的循环都包含大量语句，只有小部分循环适合展开，因此结合向量循环展开只能达到 1.21 的加速比。

437. leslie3d 程序中大部分都是上下界不定循环，SW-VEC 还无法对这类循环进行向量化，因此其性能要明显低于 icc14.0。其次，程序中还有很多适合 SLP 向量化的语句，考虑到向量化后循环内可能会包含标量语句，没有对其进行循环展开，加速比与打开向量循环展开前相同。

对于 454. calculix 程序，icc14.0 和 SW-VEC 的加速比分别为 1.25 和 1.19。454. calculix 中部分循环只有 3 次迭代，icc14.0 可以用 128 位的 SIMD 指令进行向量化，而 SW-VEC 不能对其进行向量化，因此加速比略低于 icc14.0。应用向量循环展开后，加速比提升到 1.22，已经比较接近 icc14.0 的加速效果。

456. hmmer 核心循环内主要是 if 分支语句。SW-VEC 通过循环分布把核心循环内含有依赖的一条语句分布出去后，可以进行例 2 所示的控制流向量化。向量循环展开后加速比由 1.66 提升为 1.70，而 icc14.0 向量化加速比为1.60，相对于 icc14.0 有 10%的性能提升。

结束语 循环展开是一项常用的循环优化技术，当前标量循环展开技术已经比较成熟，但是大部分高性能编译器中还没有给出有效的向量循环展开方案。本文首先分别从依赖分析、特殊循环结构和寄存器分配等 3 个方面探讨了向量循环展开与标量循环展开的相同点和不同之处；然后根据限制向量展开因子的几个因素，设计了 CUFVL 算法，并给出了向量循环展开的具体流程。实验测试与分析验证了向量循环展开算法的正确性和有效性。然而，当前的向量循环展开只能处理向量循环，对 SLP 向量化产生的同时含有向量语句和标量的循环还不能进行优化，因此，本文的下一步工作计划是实现面向混合语句的循环展开，从而进一步提升 SW-VEC 的优化能力。

参 考 文 献

[1] Sarkar V. Optimized unrolling of nested loops[C]// Proc. of the 14th Int'l Conf. on Supercomputing. New Mexico: ACM Press, 2000

[2] Li W L, Liu L, Tang Z Z. Loop unrolling optimization for software pipelining[J]. Journal of Beijing University of Aeronautics and Astronautics, 2004, 30(11): 1111-1115(in Chinese)
李文龙，刘利，汤志忠. 软件流水中的循环展开优化[J]. 北京航空航天大学学报，2004，30(11)：1111-1115

[3] Lam Y M, Coutinho J G F, Luk W, et al. Unrolling-based loop mapping and scheduling[C]// 2008 International Conference on Field-Programmable Technologuy (ICFPT'2008). Taipei, Taiwan, 2008: 321-324

[4] Callahan D, Kennedy K, Porterfield A. Software prefetching[C]// Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM Press, 1991

[5] Zhu J H. Research on SIMD Compiling Optimization Techniques [D]. Shanghai: Fudan University, 2005(in Chinese)
朱嘉华. SIMD 编译优化技术研究[D]. 上海：复旦大学，2005

[6] Jiang W H, Mei C, Guo Y. Vectorization for Real-life Multimedia Applications on Processors' Multimedia Extensions [J]. Chinese Journal of Computers, 2005, 28(8): 1254-1266(in Chinese)
姜伟华，梅超，郭一. 一种针对多媒体扩展指令集和实际多媒体程序的自动向量化方法[J]. 计算机学报，2005，28(8)：1254-1266

[7] Zhang W H, Zang B Y. Research on SIMD Compiling Optimization Techniques [J]. Communications of the China Computer Federation(CCCF), 2007, 3(2): 27-36(in Chinese)
张为华，臧斌宇. SIMD 编译优化技术研究概述[J]. 中国计算机学会通讯，2007，3(2)：27-36

[8] Intel Corp. Intel C/C++ and Intel Fortran Compilers for Linux [EB/OL]. http://www.intel.com/software/products/compilers

[9] The GNU Compiler Collection[EB/OL]. http://gcc.gnu.org

[10] Pathscale Compiler User's Guide[EB/OL]. http://www.pathscale.com

[11] http://sourceforge.net/projects/open64/files/open64/Open64-5.0

(下转第 245 页)

$(x,y)\in C_B\}$，$T_B(x)=\{y\in U;(x,y)\in T_B\}$，可得 $C_B(x)\subseteq T_B(x)$。若 $x\in \underline{T}_B(X)$，则 $T_B(x)\subseteq X$，从而 $C_B(x)\subseteq X$，故有 $x\in \underline{C}_B(X)$。于是 $\underline{T}_B(X)\subseteq \underline{C}_B(X)$。

又因 $S_B\subseteq C_B$，$S_B(x)=\{y\in U;(x,y)\in S_B\}$，故 $S_B(x)\subseteq C_B(x)$。若 $x\in \underline{C}_B(X)$，则 $C_B(x)\subseteq X$，从而 $S_B(x)\subseteq X$，故有 $x\in \underline{S}_B(X)$。于是 $\underline{C}_B(X)\subseteq \underline{S}_B(X)$。

综上，$\underline{T}_B(X)\subseteq \underline{C}_B(X)\subseteq \underline{S}_B(X)$。

又因为 $\overline{T}_B(X)=\bigcup\limits_{x\in X}T_B(x)$，$\overline{C}_B(X)=\bigcup\limits_{x\in X}C_B^{-1}(x)$，$\overline{S}_B(X)=\bigcup\limits_{x\in X}S_B^{-1}(x)$，$\overline{L}_B(X)=\bigcup\limits_{x\in X}L_B(x)$，$S_B^{-1}(x)\subseteq C_B^{-1}(x)\subseteq T_B(x)$，所以 $\overline{S}_B(X)\subseteq \overline{C}_B(X)\subseteq \overline{T}_B(X)$ 成立。

(2)同理可证。

推论 1 若 $\forall x\in U, P_B(x)\neq\varnothing$，则 $L_B=C_B=M_B$。

例 3 考虑例 1 中的不完备决策表，根据本文提出的限制相似关系，相应的限制相似类为：

$M_B(c_1)=\{c_1,c_3,c_4,c_8,c_9\}$，$M_B(c_2)=\{c_2,c_3,c_4\}$

$M_B(c_3)=\{c_1,c_2,c_3,c_4\}$，$M_B(c_4)=\{c_1,c_2,c_3,c_4\}$

$M_B(c_5)=\{c_5,c_6\}$，$M_B(c_6)=\{c_5,c_6\}$，$M_B(c_7)=\{c_7,c_8,c_9\}$

$M_B(c_8)=M_B(c_9)=\{c_1,c_7,c_8,c_9\}$，$M_B(c_{10})=\{c_{10}\}$

$M_B(c_{11})=\{c_{11}\}$

$\overline{M}_B(\Phi)=\{c_1,c_2,c_3,c_4,c_7,c_8,c_9,c_{10}\}$

$\underline{M}_B(\Phi)=\{c_2,c_3,c_4,c_{10}\}$

$\overline{M}_B(\Psi)=\{c_1,c_5,c_6,c_7,c_8,c_9,c_{11}\}$

$\underline{M}_B(\Psi)=\{c_5,c_6,c_{11}\}$

结束语 信息系统知识约简与规则获取是粗糙集理论的重要研究方向。针对不完备信息系统，人们提出了多种刻画对象相似性的不可区分关系，如容差关系、非对称相似关系、限制容差关系等。本文对以上不可区分关系进行了对比分析，在此基础上提出了限制相似关系，讨论了限制相似关系以及相应的粗糙集模型中近似算子的性质。借助本文的结果可以进一步研究基于限制相似关系的信息系统知识约简方法。

参 考 文 献

[1] Pawlak Z. Rough set [J]. International Journal of Computer and Information Science, 1982, 11: 341-356

[2] Kryszkiewicz M. Rough set approach to incomplete information systems [J]. Information Sciences, 1998, 112: 39-49

[3] Slowinski R, Stefanowski J. Rough classification in incomplete information systems [J]. Computing Modeling, 1989, 12 (10/11): 1347-1357

[4] Wang G Y. Extension of rough set under incomplete information systems[J]. Journal of Computer Research and Development, 2002, 39(10): 1238-1243(in Chinese)
王国胤. Rough 集理论在不完备信息系统中的扩充[J]. 计算机研究与发展, 2002, 39(10): 1238-1243

[5] Yin X R, Shang L. Expansion of Rough set model under incomplete information system [J]. Journal of Nanjing University, 2006, 42(4): 337-341(in Chinese)
尹旭日，商琳. 不完备信息系统中 Rough 集的扩充模型[J]. 南京大学学报, 2006, 42(4): 337-341

[6] Wang G Y. Rough Set Theory and Knowledge Discovery[M]. Xi'an: Xi'an Jiaotong University Press, 2001(in Chinese)
王国胤. Rough 集理论与知识获取[M]. 西安：西安交通大学出版社, 2001

[7] Stefanowski J, Tsoukias A. On the extension of rough sets under incomplete information[C]// Zhong N, Skowron A, Ohsuga S, eds. Proc of the 7th Int'l Workshop on New Directionsin Rough Sets, Data Mining, and Granular-Soft Computing. Berlin: Springer-Verlag, 1999: 73-81

[8] Guan L H. Incomplete information processing method based on Rough sets [D]. Chengdu: Southwest Jiaotong University, 2012 (in Chinese)
官礼和. 基于 Rough 集的不完备信息处理方法研究[D]. 成都：西南交通大学, 2012

[9] Yao Y Y. Relational interpretations of neighborhood operators and rough set approximation operators [J]. Information Sciences, 1998, 101: 239-259

[10] Qin K Y, Zhao H, Pei Z. The reduction of decision table based on generalized indiscernibility relation[J]. Journal of Xihua University, 2013, 32(4): 1-4(in Chinese)
秦克云，赵华，裴峥. 基于广义不可区分关系的决策表约简[J]. 西华大学学报, 2013, 32(4): 1-4

(上接第 231 页)

[12] Bacon D F, Graham S L, Shap O J. Compiler Transformations for High-Performance Computing [J]. ACM Computing Surveys, 1994, 26(4): 345-420

[13] Alexander M J, Bailey M W, Childers B R, et al. Memory bandwidth optimizations for wide-bus machines[C]// Proceedings of the 26th Hawaii International Conference on System Sciences. Wailea, Hawaii, 1993: 466-475

[14] Mowry T C. Tolerating Latency Through Software-Controlled Data Prefetching[D]. Stanford University, March 1994

[15] Fog A. Optimizing subroutines in assembly language [D]. Copenhagen University College of Engineering, September 2012

[16] Ma Y, Carr S. Register Pressure Guided Unroll-and-Jam[M]// The 2008 Open64 Workshop. Boston, MA, USA, April 6, 2008

[17] Carr S, Guan Y. Unroll and Jam using Uniformly Generated Sets [C]// Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30). 1997: 349-357

[18] Carr S, Kennedy K. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops[J]. ACM Transactions on Programming Languages and Systems (ToPLaS), 1994, 16(6): 1768-1810

[19] Mark S, Saman A. Predicting Unroll Factors Using Supervised Classification[C]// Proceedings of the International Symposium on Code Generation and Optimization(CGO). 2003: 204-215

[20] Monsifrot A, Bodin F, Quiniou R. A Machine Learning Approach to Automatic Production of Compiler Heuristics[M]// Artificial Intelligence: Methodology, Systems, Applications. 2002: 41-50

[21] Mowry T C, Lam M S, Gupta A. Design and evaluation of a compiler algorithm for prefetching[C]// Proceeding of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. Massachusetts: ACM Press, 1992: 62-73