软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(8):2046–2063 [doi: 10.13328/j.cnki.jos.005121]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种改进的控制流 SIMD 向量化方法*

高 伟, 李颖颖, 孙回回, 李雁冰, 赵荣彩

(数学工程与先进计算国家重点实验室(解放军信息工程大学),河南 郑州 450001)

通讯作者: 高伟, E-mail: yongwu22@126.com

摘 要: SIMD 扩展部件是近年来集成到通用处理器中的加速部件,旨在发掘多媒体和科学计算等程序的数据级并行.控制依赖给发掘程序中的数据级并行带来了阻碍,当前,无论基于 loop-based 还是 SLP 的控制流向量化方法都需要 if 转换,而没有考虑循环内蕴含的向量并行度,导致生成的向量代码效率较低.此外,不精确的代价模型指导控制流向量化,同样导致生成的向量代码效率较低.为此,提出了改进的控制流 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,同时考虑分布时数据的局部性;其次,提出了一种直接向量化控制流的方法,该方法考虑了基本块间的向量重用;最后,利用精确的代价模型指导超字选择指令和超字条件分支指令的生成.实验结果表明:与现有的控制流向量化方法相比,改进方法生成的向量代码性能提高了 24%.

关键词: SIMD 扩展部件;控制依赖;数据依赖;循环分布
中图法分类号: TP314

中文引用格式: 高伟,李颖颖,孙回回,李雁冰,赵荣彩.一种改进的控制流 SIMD 向量化方法.软件学报,2017,28(8):2046–2063. http://www.jos.org.cn/1000-9825/5121.htm

英文引用格式: Gao W, Li YY, Sun HH, Li YB, Zhao RC. Improved SIMD vectorization method in the presence of control flow. Ruan Jian Xue Bao/Journal of Software, 2017,28(8):2046–2063 (in Chinese). http://www.jos.org.cn/1000-9825/5121.htm

Improved SIMD Vectorization Method in the Presence of Control Flow

GAO Wei, LI Ying-Ying, SUN Hui-Hui, LI Yan-Bing, ZHAO Rong-Cai

(State Key Laboratory of Mathematical Engineering and Advanced Computing (PLA Information Engineering University), Zhengzhou 450001, China)

Abstract: SIMD extension is an acceleration component integrated into the general processor for developing data level parallelism in multimedia and scientific computing applications. Control dependence hinders exploiting data level parallelism in the programs. Current vectorization methods in the presence of control flow, loop-based or SLP, all need if-conversion. They do not consider the SIMD parallelism in programs, and thus result in a lower SIMD executing efficiency. Another factor that leads to low efficiency is the lack of cost model to direct SIMD code generation in the presence of control flow. To address these problems, an improved control flow SIMD vectorization method is proposed. First, loop distribution with control dependence is put up to separate the vectorizable parts and unvectorizable parts, taking data locality into account simultaneously. Second, a direct vectorization method of control flow is presented considering the reuse of data between basic blocks. Finally, cost model is used to guide the generation of select and BOSCCs to improve the efficiency of SIMD code. Experimental results show that the code performance generated by the improved methods increase by 24% compared with those of the existing control flow vectorization methods.

Key words: SIMD extension; control dependence; data dependence; loop distribution

* 基金项目: “核高基”国家科技重大专项(2009ZX01036)
Foundation item: CHB National Major Science and Technology Project Foundation of China (2009ZX01036)
收稿时间: 2015-06-24; 修改时间: 2016-03-18; 采用时间: 2016-05-23; jos 在线出版时间: 2016-10-11
CNKI 网络优先出版: 2016-10-12 16:26:33, http://www.cnki.net/kcms/detail/11.2560.TP.20161012.1626.001.html

伴随着多媒体产业发展迅猛,20 世纪 90 年代中期,各大厂商在处理器中集成了一套专用的多媒体扩展部件.该部件的指令集采用单指令多数据(single instruction multiple data,简称 SIMD)扩展技术,可同时对多个数据进行相同的操作,称为 SIMD 扩展部件.1996 年,Intel 在其奔腾处理器上集成了 SIMD 扩展部件 MMX,后来又相继推出了 SSE,AVX,IMCI 和 AVX-512.此外存在的 SIMD 扩展部件还有摩托罗拉公司 PowerPC 处理器中的 AltiVec、Sun 公司 SPARC 处理器中的 VIS、HP 公司 PA-RISC 处理器中的 MAX、DEC 公司 Alpha 处理器中的 MVI-2、MIPS 公司 V 处理器中的 MDMX[1].SIMD 扩展部件最初仅用于多媒体领域和数字信号处理器中,后来,人们将 SIMD 扩展部件应用于高性能计算机中,如 IBM 的超级计算机 Blue Gene/L 和国产的神威蓝光超级计算机中都有短向量扩展.国产处理器中龙芯[2]、飞腾[3]以及魂芯一号[4]都含有 SIMD 扩展部件.

SIMD 扩展部件的长度在不断增大,如 Intel 由最初 MMX 的 64 位到 AVX 的 256 位,再到 IMCI 的 512 位,支持的数据类型和指令也在不断丰富,这些都说明,基于 SIMD 扩展部件的向量化已成为程序并行的重要手段之一.手工向量化程序不仅容易出错,而且需要程序员了解 SIMD 指令集,所以自动向量化成为发掘生成 SIMD 指令的首要选择.如今,诸如 ICC,GCC 和 Open64 等主流编译器都集成了自动向量化的功能.

SIMD 扩展指令能够将原来需要多次装载的内存中地址连续的数据一次性装载到向量寄存器中,通过一条 SIMD 扩展指令,实现对 SIMD 向量寄存器中所有数据元素的并行处理.SIMD 扩展部件适合于处理计算密集、数据相关性少的程序.SIMD 编译优化是在发掘程序中的数据级并行,而控制流为数据流分析带来了很多的不确定性和复杂性.和数据依赖一样,控制依赖是 SIMD 向量化时需要考虑的另一种依赖形式,控制依赖是指一条语句的执行受控于另一条语句.

将控制依赖转换为数据依赖可以消除控制依赖,这个过程称为 if 转换,转换后的语句称为控制语句[5],if 转换是向量化控制依赖最常用的方法.传统向量机中,由于硬件支持可直接将控制语句转换为向量语句,但是这种控制流向量化的方法并不适合于 SIMD 扩展部件,因为最初各平台的 SIMD 指令集中没有条件选择指令.向量条件选择 select 指令的引入后,含有控制流的程序才被在 SIMD 扩展部件上向量化[6].select 指令的格式如图 1 所示,*dst*=*select*(*src*1,*src*2,*mask*),指令有 3 个参数,其中,*mask* 为掩码,*src*1 和 *src*2 是两个源操作数.当掩码位置的值为 1 时,取 *src*2 的值赋给 *dst*;否则,将 *src*1 的值赋给 *dst*.

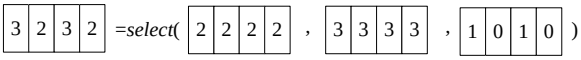

Fig.1 Format of select instruction

图 1 select 指令格式

面向 SIMD 部件的 loop-based 向量化方法是由传统向量化方法引入的,它可以将满足依赖的语句由标量类型直接转为向量类型,因此在 select 的支持下,可直接将控制语句由标量类型转为向量类型[7].文献[8]提出了一种基于超字并行(superword level parallelism,简称 SLP)[9]的控制流向量化方法,其流程也是首先通过 if 转换生成控制语句,然后构建控制语句的谓词层次图(predicate hierarchy graph,简称 PHG),并通过 PHG 构建控制语句的定义-使用(define-use,简称 DU)链和使用-定义(use-define,简称 UD)链,最后利用 SLP 向量化方法生成 select 指令.与采用 loop-based 的向量化方法相比,该方法存在两个缺点:一是发掘方法复杂,该方法不仅需要构建控制语句间的谓词层次图,而且需要在谓词层次图上建立控制语句的 DU 链和 UD 链,加大了向量发掘的难度;二是适用范围小,这是因为 SLP 算法更合适发掘迭代内并行,而实际上存在迭代内并行的程序很少,大部分程序都是存在迭代间并行,并且 SLP 算法需以地址相邻的数组为打包种子,循环体内没有地址相邻的数组时则不能生成向量化代码.

当前,基于 if 转换的向量化方法存在以下两个问题,导致生成的向量代码效率较低:一是对所有的程序首先进行 if 转换,而没有根据其循环体内蕴含的向量并行度进行选择,导致生成的向量代码效率较低;二是由于向量选择指令 select 的开销较大,在没有代价模型指导的情况下,所有情况都生成 select 指令,同样导致生成的向量代码执行效率较低.

无论是基于 loop-based 还是 SLP 的控制流向量化方法,生成的向量代码效率都较低,因此,本文提出了一种

改进的控制流 SIMD 向量化方法.具体的改进包括以下 3 个方面:一是提出了面向控制流的循环分布算法,分布循环内可向量化部分和不可向量化部分,同时考虑循环的数据局部性,提高生成向量程序的效率;二是提出了直接向量化控制流的方法,主要针对当基本块内向量并行性充足时,直接向量化各个基本块,然后考虑各个基本块间的向量重用,避免 select 指令的生成;三是提出了收益模型指导控制流向量化代码的生成,以保证生成向量代码的效率.

本文第 1 节介绍编译框架.第 2 节介绍含控制依赖的循环分布.第 3 节介绍面向向量重用的直接 SIMD 向量化控制流.第 4 节为收益指导的向量代码生成.第 5 节是实验和分析.第 6 节为相关研究.最后总结全文.

1 编译框架

含有控制依赖的向量化编译框架如图 2 所示,可分为 4 个阶段,分别是预分析与优化、向量代码发掘、向量代码生成和向量代码优化,粗黑虚线框里的阶段为本文工作的部分.预分析就是排除不能向量化的循环形式,if 外提能够将控制语句提到循环外,使得循环体内没有控制依赖,便于向量化发掘.本文在预分析与优化阶段加入面向控制依赖的循环分布算法,分布循环内可向量化部分和不可向量化部分,提高向量识别率.向量代码发掘就是将循环内的语句由标量形式识别为向量形式,当前存在的发掘方法都是先利用 if 转换,然后利用 SLP 方法或者 loop-based 方法发掘.本文提出了面向向量重用的直接 SIMD 向量化控制流方法,并且根据循环的向量并行度指导选择是否经过 if 转换.向量代码生成就是将中间表示转为向量选择指令 select 的过程.向量代码优化阶段能够将未向量化的控制语句通过 if 重构转换为控制流形式,本文提出了收益指导的向量代码生成,保证生成向量代码的效率.本节后续部分介绍编译框架内已有的工作,第 2 节~第 4 节详细介绍本文的工作.

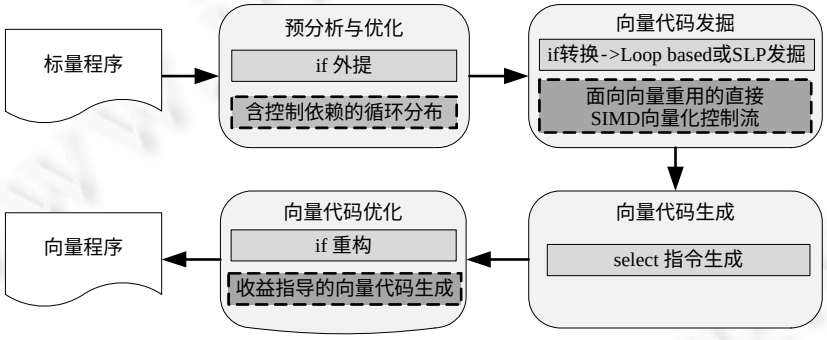

Fig.2 A vectorization framework with control flow dependence

图 2 含控制依赖的向量化框架

1.1 if外提

If 外提能够将控制语句提到循环外,使得循环体内没有控制依赖,便于向量发掘.此外,如果不外提,则每次迭代都会计算表达式的值,降低了程序的执行效率.但并不是所有的 if 语句都能外提:当 if 的判断条件为循环不变表达式时,外提才是合法的;当 if 的判断条件为循环变表达式时,将其提到循环外面,程序语义将出错.

1.2 if转换

If 转换能够将控制依赖转换为数据依赖,if 转换是一个删除程序中所有分支的过程.为保证程序的正确执行,分支不能简单地删除,而是需要用其他成分来代替.if 转换引入了控制执行的概念,就是说,每一个语句都隐式地包含一个逻辑表达式来控制其执行.只有当它的控制表达式的值为真时,语句才会被执行.编译器中引入三元操作符 *opr_select*,形式为 *dst*=*opr_select*(*cond*,*src*1,*src*2):当 *cond* 值为真时,*dst*=*src*1;当 *cond* 值为假时,*dst*=*src*2.操作符 *opr_select* 不仅可以表示单层 if-else 的结构,也可以表示嵌套 if 和含有 else-if 的结构.以下面的代码段为例,说明利用 *opr_select* 实现 if 转换的算法过程.

```
        if   (exp1)   then
S1          b(i)=B0
                if   (exp2)   then
S2              c(i)=C0
                else if (exp3)
S3              c(i)=C1
                else
S4              d(i)=D0
                endif
            else
S5          b(i)=B1
S6          x(i)=y(i)
            endif
```

利用操作符 *opr_select* 将循环体内的每条语句转换为控制语句,计算每条语句的控制条件作为操作符 *opr_select* 的操作数 1,将控制条件为真的值作为 *opr_select* 的操作数 2,将控制条件为假的值作为 *opr_select* 的操作数 3.然后,利用合并规则合并控制语句,语句 *Si* 和 *Sj* 是可合并的需要满足两个条件:一是 *Si* 和 *Sj* 都对同一变量赋值;二是 $Si.cond \vee Sj.cond=1$,即,语句 *Si* 和 *Sj* 条件的析取值为真.如果语句 *Si* 和 *Sj* 是可合并的,则需要利用依赖关系判断语句 *Si* 和语句 *Sj* 移动的合法性.首先考虑语句 *Sj* 移动到语句 *Si* 的合法性,如果不合法,再考虑语句 *Si* 移动到语句 *Sj* 的合法性.实例中,*S*1 和 *S*5 是可合并的,并且语句移动合法,所以将其合并为 *S*1.*S*2 和 *S*3 虽然都是对 *c*(*i*)赋值,但是两条语句控制条件的析取值不为真,所以不可合并.合并后语句集合如下所示:

```
S1   b(i)=opr_select(exp1,B0,B1)
S2   c(i)=opr_select(exp1 .AND.   exp2,C0,c(i))
S3   c(i)=opr_select(exp1 .AND.   .NOT. Exp2 .AND. exp3,C1,c(i))
S4   d(i)=opr_select(exp1 .AND.   .NOT. Exp2 .AND. .NOT.exp3,D0,d(i))
S6   x(i)=opr_select(.NOT.exp1,y(i),x(i))
```

1.3 向量代码发掘与生成

在满足依赖关系的情况下,loop-based 向量化方法直接将一条标量语句转换为向量语句,由于中间操作 *opr_select* 和向量选择指令 select 具有相同功能和操作数,代码生成时可直接将中间表示映射到 select 指令.

利用 SLP 方法发掘向量代码时,需要对 if 转换生成控制语句构建谓词层次图,并通过谓词层次图构建控制语句的 DU 链和 UD 链,最后根据 DU 链和 UD 链进行打包,并最终生成 select 指令.

1.4 if重构

当循环内的控制语句由于存在依赖等原因而向量化失败时,需要通过 if 重构减少判断条件执行的次数.为了避免if转换后这些没有向量化的循环性能变差,采用if重构的目标将这个部分被控制执行的代码用一个最小的分支集合来代替.if 重构就是将具有相同控制条件的语句合并到一个条件分支语句中.

2 含控制依赖的循环分布

当循环体内存在不可向量化语句时,可以通过循环分布将可向量化语句与不可向量化语句分开,进而将可向量化语句转为向量执行.然而当循环体内含有控制依赖时,循环分布不仅要考虑数据依赖,而且要考虑控制依赖.如何将可向量化语句与不可向量化语句分离开,是一个值得研究的问题.本节的创新点是在对含有控制依赖的循环,将其面向 SIMD 扩展部件分布为可向量化部分和不可向量化部分的同时,进一步考虑分布后循环的数据局部性.

我们在控制依赖图中加入数据依赖,将控制依赖作为数据依赖的扩展进行处理.由于为图中每个节点最多只有一个控制依赖前驱的代码段分布是容易的,只要在控制条件的真分支或假分支内生成每条语句即可.因此,首先将控制依赖图变换成一个含有一组控制依赖树的规范形式,该形式具有以下性质:一是每条语句最多控制依赖于一条其他语句;二是依赖树可以按这样的顺序排序,使得所有树之间的数据依赖都从该序列中前面的树中流出,并流入到该顺序后面的树中去,本文利用文献[7]中的控制依赖的分裂算法.

接下来,根据节点间的依赖关系确定哪些节点是可以向量化的,哪些节点是不能向量化的.在面向向量机的传统向量化方法中,其假设硬件支持的向量长度无限长,循环的所有迭代可以一次执行完毕.而这样的假设面向的是整个迭代空间的重排序,可以称为全重排序.面向 SIMD 扩展部件其向量长度是一定的,将迭代空间分组,组内的几次迭代可并行,而组间保持原执行顺序,这种重排序方法称为分组重排序.假设循环迭代 8 次,串行的执行顺序如图 3(a)所示,全重排序的如图 3(b)所示,分组重排序如图 3(c)所示.

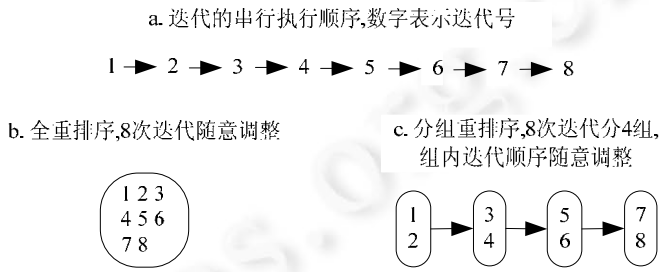

Fig.3 Illustration graph for full reordering and packet reordering

图 3 全重排序和分组重排序

针对 SIMD 扩展部件的部分重排序具有“整体串行,局部并行”的特征,当依赖距离大于组内的迭代数时,此依赖的源点和汇点将不会在组内同时出现,因此,这种依赖不影响组内的重排序,组内的几次迭代可以打包为向量执行.因此,面向 SIMD 扩展部件的依赖关系分析需要将依赖距离纳入考虑范围.

当依赖距离为 0,即为循环无关依赖时,节点可以被向量化;当依赖距离大于 0 小于 vf 时,节点不满足重排序的要求,因此不能被向量化;当依赖距离大于 vf 时,节点满足重排序的要求,因此可以向量化.其中,vf 为向量因子,即向量操作可容纳的数据个数.根据上述原则,判断图中各个节点的向量化情况,并将节点标记为可向量化节点和不可向量化节点.

当一个循环有多种分布策略时,可在循环分布时进一步考虑数据的局部性,以获得更优的分布结果.本文根据文献[10]中 wolf 提出的 cache 代价模型,计算分布后加入一条语句时的 cache 命中率,以提高程序的数据局部性.含有控制依赖的循环分布算法如算法 1 所示,代码生成从依赖图中没有控制依赖或者数据依赖前驱的节点开始,对其经过 gencode 算法处理后,将其删除并更新图中没有数据依赖或者控制依赖节点的集合,核心过程 gencode 如算法 2 所示.

算法 **1**. 含控制依赖的 SIMD 向量化循环分布算法.

输入:循环的控制依赖图和数据依赖图.

输出:可 SIMD 向量化和不可 SIMD 向量化的子图.

1. 计算 P 中没有控制依赖前驱或数据依赖前驱的语句的集合 S:当 S 不为空时,转到步骤 2;否则,转到步骤 4.
2. 令 n 是 S 中的任一元素,从 S 中删除 n,从 n 调用 gencode 过程开始生成代码,转到步骤 3.
3. 将没有控制依赖前驱且数据依赖前驱已经由 gencode 处理的节点加入 S,转到步骤 1.
4. 算法结束.

算法 **2**. 循环分布的代码生成过程 gencode.

输入:没有控制依赖或者数据依赖前驱的节点待分布的节点 n.

输出:从节点 n 开始生成的可 SIMD 向量化和不可 SIMD 向量化的子图.

1. 由于当前节点可能自身存在阻碍向量化的真依赖,因此首先判断当前节点是否可向量化:如果可向量化,则转到步骤 2;否则,转到步骤 5.
2. 继续判断其加入当前可向量化区域后是否可以被向量化,因为有可能因为新语句的加入带来依赖环,导致当前可向量化区域不可向量化.同时判断这样的代码移动是否满足数据依赖的要求:如果加入当前可向量化区域后,整个向量化区域仍然可以向量化,则转到步骤 3;否则,转到步骤 4.
3. 考虑加入当前的可向量化区域后的数据重用问题:如果加入后有重用,则加入当前可向量化区域;如没有重用,则转到步骤 4.
4. 重新生成一个可向量化区域,并将当前节点加入,转到步骤 6.
5. 加入当前不可向量化区域后,判断其是否有重用:如果有重用,则加入该区域;否则,重新生成一个不可向量化区域,转到步骤 6.
6. 算法结束.

如图 4 所示的循环体内含有 9 条语句,分别标记为 1~9,图中实线代表控制依赖,虚线代表数据依赖.true 表示该数据依赖为真依赖,anti 表示该依赖为反依赖,vectoriable 表示该节点为可向量化节点,unvectoriable 表示该节点为不可向量化节点.其中,节点 5 为不可向量化节点,它的存在导致当前循环不能直接被向量化.

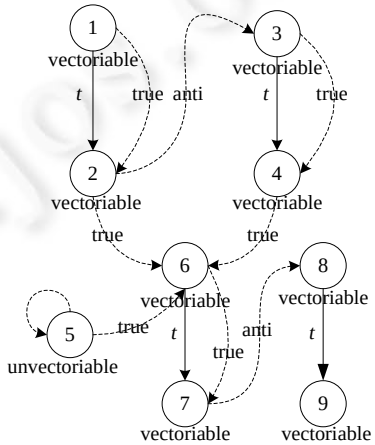

Fig.4 Dependence graph of a loop to be distributed

图 4 待分布循环的依赖图

由于初始每条语句最多控制依赖于 1 条其他语句,因此不需要利用控制依赖分裂算法.初始没有数据依赖或者控制依赖前驱的为节点 1,所以代码生成从节点 1 开始.节点 1 和节点 2 可向量化,因此创建一个可向量化区域将节点 1 和节点 2 加入,然后代码生成从节点 3 开始,节点 3 和节点 4 也可向量化,将其加入当前节点 1 和节点 2 的区域同样可向量化,并且不会影响 cache 的局部性,因此将其加入当前的可向量化区域.

接下来代码生成从节点 5 开始,由于节点 5 本身是一个不可向量化的节点,因此创建一个不可向量化区域,将其加入.

然后,代码生成从节点 6 开始,节点 6 是可向量化节点,但是由于节点 5 到节点 6 存在依赖边,因此将其加入当前节点 1 到节点 4 组成的可向量化区域是不合法的.节点 6 需要重新创建一个可向量化区域,同时将节点 7 加入.下一步代码生成从节点 8 开始,将节点 8 和节点 9 加入当前节点 6 和节点 7 组成的可向量化区域是合法的,但是加入节点 6 和节点 7 组成的可向量化区域后,cache 的局部性降低,因此节点 8 和节点 9 重新创建一个可向量化区域.

利用本文提出的含有控制依赖的循环分布算法最终的循环分布结果如图 5 所示.

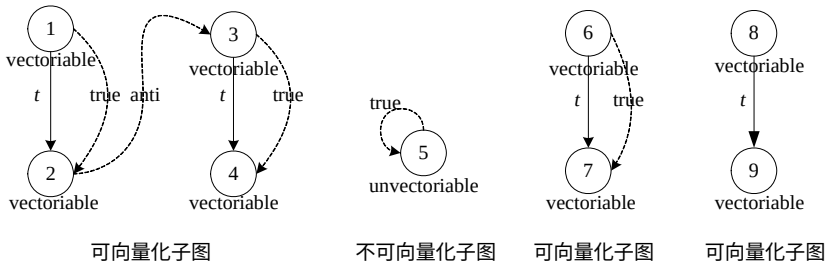

Fig.5 Vectorizaton graph after loop distribution

图 5 循环分布后的可向量化和不可向量化子图

3 面向向量重用的直接 SIMD 向量化控制流

当前,无论是基于 loop-based 还是 SLP 的控制流向量化方法,都需要先利用 if 转换将控制依赖转为数据依赖,最终利用 select 指令生成向量代码.这种向量化方法能够将所有可向量化的语句转为向量执行,然而当循环体的迭代内向量并行度较高时,即循环体内同构语句条数足够多时,基于 if 转换的控制流向量化方法生成的代码并不高效.因为这些语句对应的控制条件相同,不需要生成 select 指令,可以直接利用 SLP 方法打包,我们将这种向量化方法称为直接 SIMD 向量化控制流.在有直接 SIMD 向量化控制流方法的基础上,我们就可以根据循环体内蕴含的向量并行度,去选择直接 SIMD 向量化控制流,还是经过 if 转换后再对控制语句进行向量化发掘.

直接向量化控制流方法适合于 if 语句形成的各个基本块内有足够的向量并行性时,这种情况在程序中也经常出现,如 SPEC 2000 的 191.fma3d 的核心函数 solve 中,183.equake 的核心函数 smvp 中以及 NPB 中 MG 的核心函数 zran3 中.如图 6(a)所示,if 语句形成的各个基本块内都含有地址相邻的同构语句,因此可以直接利用 SLP 方法对各个基本块进行打包,在每个基本块内生成向量化代码.此时相当于发掘迭代内向量并行性,基本块内每个语句的执行条件相同,因此不需要生成 select 指令.

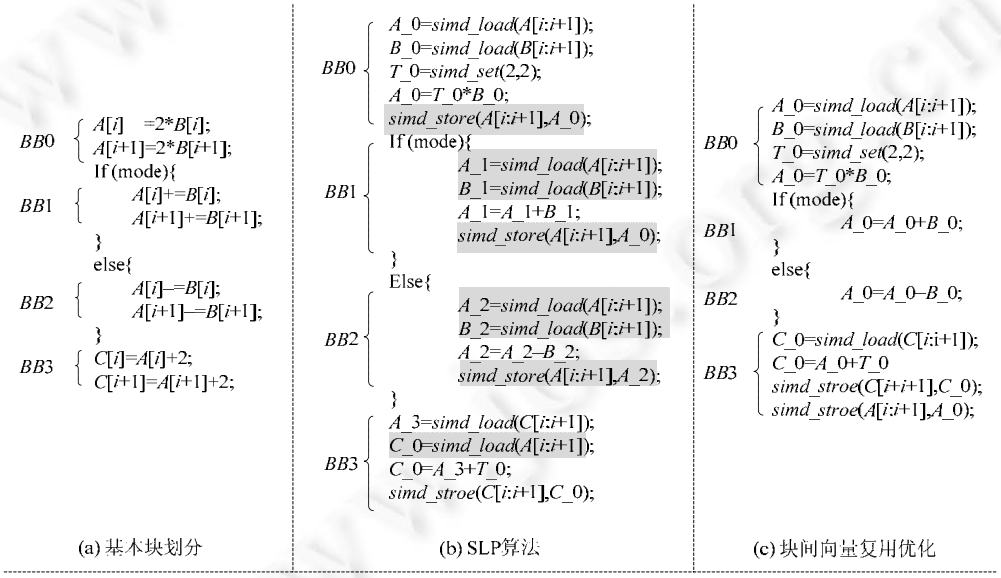

Fig.6 An example of using across basic block vectorization

图 6 跨基本块向量化实例

直接向量化控制流是首先向量化各个基本块.本文采用 SLP 方法对基本块进行向量化.SLP 算法需要在每

个基本块内生成数据打包指令和数据拆包指令,分别用来将数据从内存载入到SIMD寄存器内,以及将SIMD寄存器中的计算结果回写到内存中.SLP 算法的分析范围仅限于基本块内,无法获取块外的数据打包信息,当基本块之间使用相同向量时,会产生冗余的 *simd_load* 和 *simd_store* 指令,如图 6(b)阴影中的指令所示.因此,我们提出块间向量复用算法删除冗余向量语句,得到图 6(c)中的优化结果.

块间向量复用优化算法如下.

首先引入两个全局缓冲区:*V_store* 表示向量写缓存,*V_load* 表示向量读缓存.

- 向量写缓存 *V_store* 主要保存生成 *simd_store* 指令所需的信息,其保存格式表示为二元组(*S_vector*, *B_scalars*),该二元组等价于一条 *simd_store* 指令:*B_scalars*=*simd_store*(*S_vector*).
- 读缓存 *V_load* 主要保存生成 *simd_load* 指令所需的信息,其保存格式与写缓存 *V_load* 相一致,唯一的区别是它等价于一条 *simd_load* 指令:*S_vector* =*simd_load*(*B_scalars*).

SLP 算法一般会在基本块的入口将块内引用到的标量进行打包,并在基本块的结尾处将向量进行拆包,将向量计算结果更新到标量中.块间向量复用优化需要对上述数据打包、拆包时机进行适当的提前和延后.

1) 提前读

当 SLP 算法在基本块 *B* 入口处需要装载 *B_scalars* 时,先检查读缓存 *V_load* 中是否存在 *B_scalars* 对应的二元组:如果不存在,则使用 *simd_load* 指令将 *B_scalars* 载入到 *S_vector* 中,并新生成一个二元组(*S_vector*′, *B_scalars*′)放入到 *V_load* 中;如果存在,就直接使用 *V_load* 中(*S_vector*′,*B_scalars*′)的 *S_vector* 进行后续的 SIMD 指令生成.将这种处理策略称为提前读.

2) 缓迟写

当 SLP 算法在基本块的结尾需要用向量 *S_vector* 中的计算结果更新对应标量 *B_scalars* 的值时,先不生成 *simd_store* 指令,而是将 *simd_store* 中二元组信息(*S_vector*,*B_scalars*)保存到写缓存 *V_store* 和 *V_load* 中.直到后续基本块中引用或定义了 *B_scalars* 中的部分值时,再使用二元组(*S_vector*,*B_scalars*)生成 *simd_store* 指令.将这种处理方式称之为缓迟写.

通过缓迟写可以实现减少不必要的 *simd_store* 指令,“提前读”在“缓迟写”的基础上实现块间向量复用.在静态分析时,由于无法确定 IF 语句执行轨迹,需要将基本块分为两类进行分别处理:一类是普通基本块 *Bs*,另一类是 IF 基本块 B_{IF}.其中,B_{IF} 包含 IF 所有分支的基本块.

以下是基本块 *Bs* 的块间向量复用规则.

一、向量复用规则

$$\exists i \quad V\_load[i].B\_saclars == simd\_load.B\_saclars'$$
$$\Rightarrow simd\_load.S\_vector' = V\_load[i].S\_vector$$
$$\forall i \quad V\_load[i].B\_saclars \neq simd\_load.B\_saclars'$$
$$\Rightarrow \text{生成} S\_vector' = simd\_load(B\_saclars')$$
$$\Rightarrow V\_load.add((S\_vector', B\_saclars')).$$

二、缓存写规则

$$\forall i \quad V\_store[i].B\_saclars \neq simd\_store.B\_saclar''$$
$$\Rightarrow V\_store.add(S\_vector'', B\_saclar'')$$
$$\forall i \quad V\_load[i].B\_saclars \neq simd\_store.B\_saclar''$$
$$\Rightarrow V\_load.add(S\_vector'', B\_saclar'').$$

三、部分读规则

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

四、部分写规则

$$\begin{aligned}
&\exists i \quad (V_store[i].B_saclars \cap simd_store.B_saclars'') \neq simd_store.B_saclars'' \\
&\qquad \Rightarrow V_store[i].B_saclars = simd_store(V_store[i].S_vector) \\
&\qquad \Rightarrow S_vector'' = simd_store(B_saclars'') \\
&\qquad \Rightarrow V_store[i].delete(\cdot) \\
&\qquad \Rightarrow V_load.add((S_vector'', B_saclars'')) \\
&\qquad \Rightarrow V_store.add((S_vector'', B_saclars'')).
\end{aligned}$$

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,$V_load[\cdot]$表示读缓存队列,$V_store[\cdot]$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 $reuse\_vectors[\cdot]=\{(R\_vector,B\_scalars)\}$,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 $L1$,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 $1+NBI\times(1-PAFS)$ 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 $NBIC\times(1-PAFS)+NB$,其中,*NB* 为 BOSCC 指令的代价.插入 BOSCC 指令是有收益的当且仅当 $NBIC>NBIC\times(1-PAFS)+NB$,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

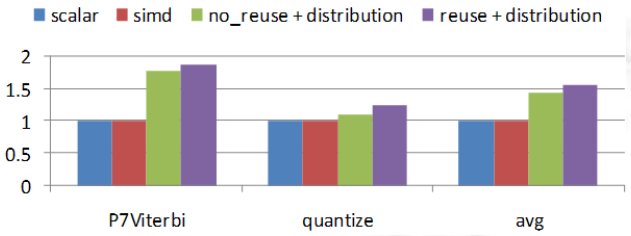

Fig.10 Results of loop distribution with control dependence

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%.

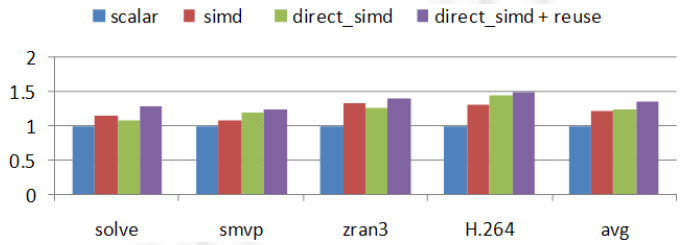

Fig.11 Result of direct control flow vectorization method

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.

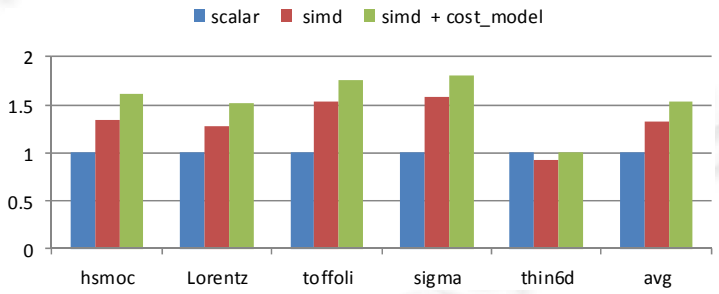

Fig.12 Results of vectorization method guided by benefit

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.

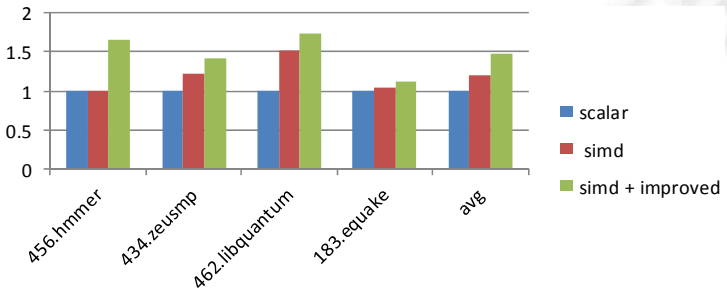

Fig.13 Test results of full programs

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.

李颖颖(1984－),女,讲师,主要研究领域为高性能计算.

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.

孙回回(1991－),女,博士生,主要研究领域为先进编译技术.