一种基于剪切的 SLP 向量化方法*

李颖颖[1,2]，奚慧兴[3]，高 伟[1,2]，李 伟[4]，翟胜伟[4]
(1. 信息工程大学，郑州 450002；2. 数学工程与先进计算国家重点实验室，郑州 450002；3. 鞍山师范学院，辽宁 鞍山 114007；4. 中国电子科技集团公司第二十七研究所，郑州 450047)

摘 要：作为多媒体和科学计算等领域重要的程序加速器件之一，SIMD 扩展部件现已广泛集成于各类处理器中。自动向量化方法是目前生成 SIMD 向量化程序的重要手段。超字并行 SLP (superword level parallelism) 方法现已广泛应用于编译器中，并成为实现基本块级代码向量化的主要手段。SLP 在进行收益评估时仅考虑代码段整体向量化的收益，并没有考虑到向量化收益为负的片段会降低最终整体的向量化收益，从而导致 SLP 方法无法达到最好的向量化效果。基于此，提出了一种基于剪切的 SLP 向量化方法(throttling SLP，TSLP)。通过寻找最优的向量化子图，去除了向量化收益为负的代码段，从而可以获得更好的向量化效果。通过标准测试程序的实验结果表明，与原来的 SLP 方法相比，TSLP 方法平均能够获得 9% 的性能提升。

关键词：单指令多数据扩展部件；自动向量化；超字并行；代价模型
中图分类号：TP301.6 **文献标志码**：A **文章编号**：1001-3695(2018)09-2578-05
doi:10.3969/j.issn.1001-3695.2018.09.004

SLP vectorization method based on throttling

Li Yingying[1,2], Xi Huixing[3], Gao Wei[1,2], Li Wei[4], Zhai Shengwei[4]
(1. *Information Engineering University, Zhengzhou* 450002, *China*; 2. *State Key Laboratory of Mathematical Engineering & Advanced Computing, Zhengzhou* 450002, *China*; 3. *Anshan Normal University, Anshan Liaoning* 114007, *China*; 4. *The 27th Research Institute, China Electronics Technology Group Corporation, Zhengzhou* 450047, *China*)

Abstract: SIMD vectors are widely adopted in modern general purpose processors as they can boost performance and energy efficiency for media and scientific applications. Compiler-based automatic vectorization is one approach for generating code that makes efficient use of the SIMD units. The SLP vectorization algorithm is the most well-known implementation of automatic vectorization. Choosing whether to vectorize is a one-off decision for the whole graph that has been generated. However, this is suboptimal because the graph may contain code that is harmful to vectorization due to the need to move data from scalar registers into vectors. Therefore, this paper proposed a solution to overcome this limitation by introducing throttling SLP (TSLP), a novel vectorization algorithm that finds the optimal graph to vectorize. The decision did not consider the potential benefits of throttling the graph by removing this harmful code. The experiments show that TSLP can decrease execution time by 9% compared to SLP on average.

Key words: SIMD extension; auto-vectorization; superword level parallelism(SLP); cost model

0 引言

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指令集，该指令集采用单指令多数据(single instruction multiple data，SIMD)扩展技术，可同时对多个数据进行相同的操作，称之为 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[1]。其他 SIMD 扩展部件还包括摩托罗拉 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 等[2]。SIMD 扩展部件现已广泛集成于各类处理器中，已成为多媒体和科学计算等领域程序重要的加速器件之一[3]。

生成 SIMD 向量化程序的两种主要方法分别是手工向量化和自动向量化。虽然手工向量化理论上能够实现最高程度的向量化，但由于程序员不仅要掌握程序的结构特征和数据布局，还要熟悉目标处理器底层结构和指令集的特点，这增加了程序员的编程难度和工作量[4]。而自动向量化不需要程序员考虑过多细节，通过自动向量化编译器自动地生成针对目标处理器 SIMD 扩展部件的向量程序，减轻了程序员的工作负担。因此自动向量化逐步成为当前 SIMD 向量化程序的主要方式。循环级向量化和基本块级向量化是自动向量化编译器中两种主要的向量化方法[5]。面向 SIMD 扩展部件的循环级向量化方法是通过借鉴传统向量化方法引入的，该方法通过求解语句的依赖关系来判定是否可进行向量化变换，通过依赖关系图中强连通分量的拓扑排序来确定语句的执行顺序。基本块向量化方法从另一个粒度对程序的向量并行性进行发掘，基本块向量化方法从指令级并行中挖掘数据级并行。面向基本块的向量化方法中常常提到打包、解包的概念。包是一个同构语句的集合，将多条同构语句组成包的过程称为打包；反之则称为拆包[6,7]。

超字并行(superword level parallelism，SLP)方法现已广泛

收稿日期：2017-04-18；**修回日期**：2017-06-12 **基金项目**：国家自然科学基金资助项目(61472447)；国家“863”计划资助项目(2014AA01A300)；国家“核高基”重大专项资助项目(2013ZX0102-8001-001-001)

作者简介：李颖颖(1984-)，女，河南郑州人，讲师，博士研究生，主要研究方向为高性能计算、先进编译技术(liyingying1005@163.com)；奚慧兴(1981-)，男，实验师，硕士，主要研究方向为先进编译技术、软件工程；高伟(1988-)，男，博士研究生，主要研究方向为高性能计算、先进编译技术；李伟(1979-)，男，工程师，学士，主要研究方向为高性能计算；翟胜伟(1982-)，男，工程师，硕士，主要研究方向为先进计算.

应用于编译器中,它首先通过扫描编译器的中间表示,识别出重复执行的标量指令序列,然后将重复执行的标量指令序列转换为向量指令[8]。SLP 不同于循环级向量化方法的主要之处在于 SLP 算法的处理对象是基本块,这使得 SLP 方法更加灵活。当前 SLP 方法在进行收益评估时仅考虑代码段整体向量化的收益,并没有考虑到向量化收益为负的片段会降低最终整体的向量化收益,从而导致 SLP 方法达不到最好的向量化效果。基于此,本文提出了一种基于剪切的向量化方法(TSLP)。TSLP 方法通过改进 SLP 方法的代价模型,去除了向量化收益为负部分的代码段,从而可以获得更好的向量化效果。

1　SLP 向量化方法的流程

SLP 算法的流程如图 1 所示。其中未突出强调的部分属于 SLP 算法,而突出强调的部分属于本文提出的 TSLP 算法。本章首先对 SLP 算法的整个框架进行描述,然后在第 2 章对 TSLP 算法进行详细说明。SLP 算法以互相没有依赖并且访存连续的语句作为种子,然后沿着数据依赖图中自底向上进行搜索生成可以向量化的包,一旦遇到不能向量化的指令则停止搜索,生成的每个包内包含有向量化代价等必要的数据。

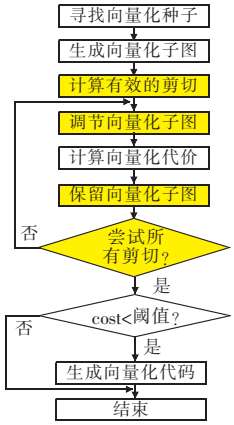


图 1　SLP 算法的流程

SLP 方法在对向量化代码进行性能评估时,需要考虑目标平台每条指令的代价,包括标量代价(scalarCost)和向量代价(vectorCost)。对每个包计算出标量代价和向量代价的差值 costDiff(costDiff = vectorCost - scalarCost)。当 costDiff 结果为正时表示这个包向量化是有收益的,而当 costDiff 结果为负时表示这个包向量化是无收益的。在对整个代码区域的向量化性能进行评估时,还需要考虑额外的指令开销,如标量与向量之间的数据转换开销(scatterGatherCost),因此整个区域向量化的总代价为所有包代价的总和加上数据转换开销,公式如下:

$$\text{totalCost} = \sum_{g=1}^{\text{groups}} \text{costDiff}(g) + \text{gatherScatterCost}$$

SLP 方法最后会将整个向量化代价 totalCost 与一个阈值相比较,然后决定是否进行向量化。阈值一般设置为 0,即当 totalCost < 0 时向量化是有收益的,否则认为向量化无收益。向量化是有收益的当且仅当程序向量执行的代价小于标量执行的代价。编译器最后根据收益模型判断是否将程序转为向量形式。如果判断应该向量化,则编译器将标量中间表示转为对应的向量中间表示,否则继续保持标量执行。

SLP 方法是到最后对整个向量化区域的收益进行评估。它并没有考虑到当部分代码片段向量化后收益为负时的情况,这个负收益代码段向量化后会抵消整体向量化的收益。特别是当整个代码段向量化收益为负时,导致整个代码段向量化失败,进而失去了向量化的机会,或者向量化获得的是次优解。为解决这个问题,本文对 SLP 向量化流程进行改进,提出了基于剪切的 SLP 向量化方法。以图 2(a)中的代码段说明本文提出的基于剪切的 SLP 向量化方法。

图 2(c)所示为图 2(a)代码的数据依赖关系图,代码段中数组 B、C、D 和 E 的运算结果存入数组 A 中。其中两条待向量化语句对数组 A 和 B 访存是连续的,而其他数组访存并不是连续的,因为它们的索引为 $2*i$ 和 $3*i$,这些不连续的数组在图中用蓝色标记(见电子版),其并不能向量化。如果继续按照 SLP 算法进行向量化是没有收益的,如图 2(d)所示。

下面利用代价模型对向量化进行评估。为便于叙述,本文简单地将每条指令的开销设为 1。图 2(d)中的节点标记出了标量和向量的代价差值。例如,$G1$ 节点两个节点向量执行的代价为 1,而标量执行的代价为 2,因此差值 costDiff 为 1 - 2 = -1,程序如果继续保持标量执行的 costDiff 为 2 - 2 = 0。向量化过程中额外生成如插入节点等,它们的 costDiff 值为负,因为如果保持标量执行并不需要这些节点,利用 SLP 方法向量化后 totalCost 为 0,即将代码转为向量执行后也没有收益。

SLP 方法向量化后没有收益是因为 $G4$ 以上节点子图的向量化收益为负,而 SLP 方法将整个图当做一个区域看待不能识别出这样的问题。为此本文提出了 TSLP 方法,将向量化收益为负的代码与收益为正的代码通过剪切隔离开来,本文将这种剪切操作称为调节。调节使得 SLP 向量化时不是仅仅只向量化整个代码区域,而是通过剪切将整个代码区分为向量化区域和不向量化区域以获得更多的收益。按照本文提出的 TSLP 向量化方法最后生成的向量化结果如图 2(e)所示。其中在剪切以上部分所有的指令均为标量指令,而在剪切以下部分所有的指令均为向量指令,在剪切点处需要生成插入指令以将数据从标量寄存器转移到向量寄存器中。TSLP 方法的 totalCost 为 -1,这意味着利用 TSLP 方法对该代码段进行向量化是有收益的。在 Xeon 平台上测得,TSLP 方法生成的向量化代码性能是 SLP 方法生成的向量化代码性能的 17%。

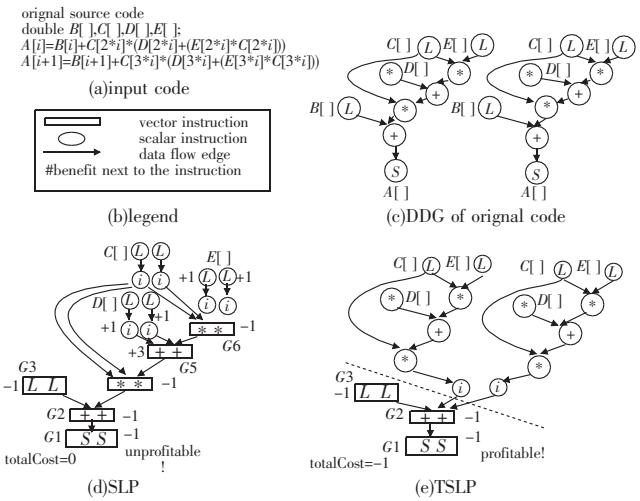


图 2　基于剪切的 SLP 向量化方法示例

2　基于剪切的 SLP 向量化方法

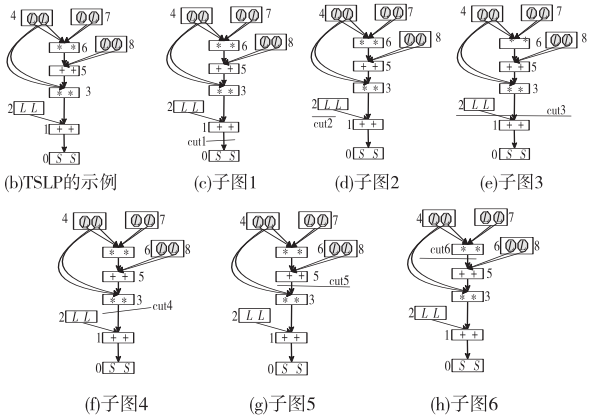
2.1　算法框架

TSLP 算法能够通过调节向量化代码的数量以改进向量化的性能。它丢弃了向量化收益为负部分的代码,即使那些代码实际上可以被向量化。TSLP 算法是 SLP 算法的改进,图 1 中白色框图部分属于原始 SLP 部分的步骤,而黄色强调部分的框图表示 TSLP 的部分(见电子版)。TSLP 算法首先也是构建数据依赖图,图中的每个节点表示可能会被向量化的标量节点。SLP 和 TSLP 向量化图都是自底向上地构建,图中的边表示数据依赖。其从种子指令开始,可以被向量化的指令形成一

个包,并且占有图中的一个节点。一旦遇到读内存指令或者不可向量化指令图构建结束。每个节点包含的信息包括含有的标量指令、依赖关系、是否需要数据拼凑和包的代价。节点格式如图 3(a)所示。

node format	0	1	2	3	4	5	6	7	8
Scalars	S,S	+,+	L,L	*,*	L,L	*,*	+,+	L,L	L,L
Dependencies	1,	2,3	2,3	4,5	-	6,8	4,7	-	-
NeedToGather	No	No	No	No	Yes	No	No	Yes	Yes
Cost(V,S,Total)	1,2,-1	1,2,-1	1,2,-1	1,2,-1	2,0,+2	1,2,-1	1,2,-1	2,0,+2	2,0,+2

(a)TSLP的向量表示



(b)TSLP的示例 (c)子图1 (d)子图2 (e)子图3

(f)子图4 (g)子图5 (h)子图6

图3 TSLP数据结构和bitmap子图的表达

与 SLP 算法不同,TSLP 算法需要计算图中不同部分的代价,然后判断其向量化后是否会抵消当前的向量化收益。这种尝试是通过在原始图中生成剪切操作,然后评估剪切后子图的向量化收益得以实现的。也就使得剪切内的节点保持向量执行,而剪切外的节点保持标量执行。有效的剪切是指创建了一个连通子图,每个子图的代价被评估,然后最小代价的被标记。重复此过程,直到尝试原始图中所有的有效剪切。而 SLP 仅对整个图的向量化代价评估一次,不会尝试以任何方式修改图。SLP 方法不好的方面是即使向量化后代码的收益为负,它也不会将收益为负的部分移除。TSLP 算法的关键是每生成一个向量节点后,都要检查向量化的收益性。加入当前的向量化节点后,如果是有收益的,那么算法将该节点子图中的标量指令转为向量指令,余下的部分剪掉,否则代码保持不变。

2.2 调节

TSLP 算法的目的是对于给定向量化原图找到收益最大的向量化子图。当 totalCost 最小时停止向量化,在收益为负的代码关联到这个图前,这有可能是由于过多的 gather 或者 scatter 指令的原因。在原始图中计算所有可能是 TSLP 的工作。剪切将原来的向量化图划分为两个部分,分别是可以向量化节点和标量节点。一个有效的剪切是指包含根节点的连通可向量化子图。将数据由标量引入向量数据跨越剪切时,一个剪切将会引入新的插入指令。

图 4 中的例子表明了整个对图调节的过程。图右侧的表显示了对于每一步剪切(cut0 ~ cut6)代价的计算过程,以及每一步调节的过程。在 cut6 处剪切表示现有的 SLP 算法,因为所有输入的节点均为不可向量化节点,并且进一步处理不会有任何收益。执行向量化的总代价 totalCost 是向量代价与标量代价之间的差值。向量代价是所有向量、标量以及数据拼凑指令的总和。例如在 cut1 处生成了单个向量节点,包括两条存节点"[S S]",所以 $V=1$。代码剩余部分为标量代价即 16($S=16$)。此处需要两个插入指令,用于由标量数据生成向量数据,因此 $G=2$。这使得向量代价为 1 + 16 + 2。标量代价等于所有标量指令的代价之和为 18。此处的剪切不影响标量。

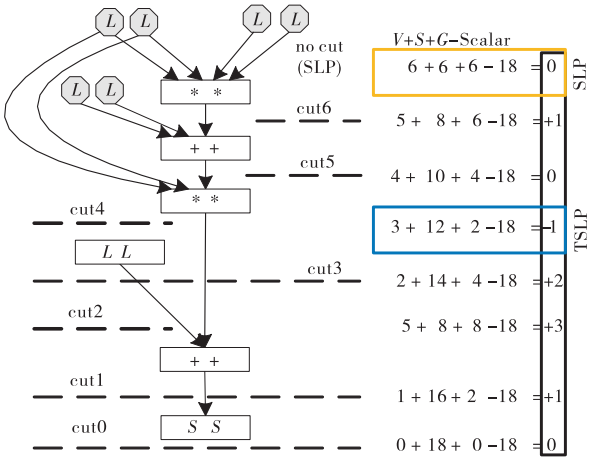


图4 不同剪切方案的向量化收益

一般情况下向量化节点越多,聚集或者分散的代价越大。因为为了生成更多的向量化节点,可能会引入越多的聚集或者分散操作。同时在离根节点越远处剪切,生成的向量节点越多。如果向量化是有收益的,那么总体代价为负;如果向量化没有收益,那么整体代价为正。在这个例子中,向量化在 cut4 处是有收益的,因为代价为 -1。一般情况下 TSLP 算法会尝试所有可能的剪切,并且选择最小代价的一种。在这个例子中,TSLP 决定在 cut4 处进行调节向量化,这样不仅节省了代价 1 而且使得向量化有收益。

算法 1 列出了整体所有有效的子图集合。算法的输入是 SLP 图,输出为所有可以调节的子图集合,每个子图均为一个包含根节点的连通子图。调节子图的出边与调节引入的剪切相对应。算法包含两个函数,分别是前端函数和后端函数。前端函数创建一个图 init_g,它仅仅包含初始节点。然后调用递归函数用新图作为一个参数。

算法 1 剪切子图生成算法

```
功能:生成剪切子图。
输入:SLP 向量化子图。
输出:剪切的 SLP 子图集合。
//Front-end function
Gen_throttled_subgraphs(SLPGraph){
        root = SLPGraph. get_root( )
        Graph init_g
        Init_g. add_node(root)
        Gen_throttled_subgraph_re(init_g)
}
Gen_throttled_subgraph_rec(subg){
        //如果子图 subg 已在 gset 中,则退出
        if(subg in gset)
            return
        //将子图 subg 插入到剪切子图中
        gset. insert(subg)
        for(neighbor in subg neighbors){
        //跳过已经添加的节点
            if(neighbor in subg)
              continue
        Graph sub_cp = copy of subg
        Subg_cp. add_node(neighbor)
        //递归地调用剪切子图函数
        gen_throttled_subgraphs_rec(subg_cp)
        }
}
```

递归函数是计算的核心。简言之,这个函数保留输入子图到输出集,然后每次递归地添加一个邻接节点。为了避免重复计算,如果子图已经在集合中,那么计算开始于第一个出口。随后这个子图插入到输出集合中。接下来函数遍历所有子图的邻居为了将它们添加到子图中。如果邻居已经在子图中,那么跳过;否则复制一个子图的副本 subg_cp(包含邻节点),结

果子图用于下一步递归的参数。

实现细节:利用 LLVM 的 SLP 框架,重用 SLP 的向量表示。向量表示中的每个入口在图中表示一个节点。SLP 中间表示并不能保持图中的数据依赖,SLP 利用数据依赖是为了构建数据依赖图。扩展每个节点的框架为了加入数据依赖边。将一个简单的整型向量表示用在边的表达,向量表示中每个数字 N 表示组中的 groups_vector[N]节点。例如,包 1 依赖于包 2 和 3,所以 groups_vector[1]中的依赖预包括向量 2 和 3。

为了获得一个快速的实现需要一个有效的子图表达,所以本文采用 bitmaps 进行尝试。位置 i 表明 bitgroups_vector[i]是否为子图的一部分。查看节点 i 的数据是常数时间。这种表示同样需要常数时间的比较,以及常数时间的检查看子图是否已经包含在 gset 集合中。为了避免对于多节点的复杂性爆炸,在 gset 中节点数量达到一个阈值节点后限制尝试。在达到这个阈值后,每个新的子图添加所有的邻节点。对于本实验,经验性地将阈值设置为 50。尝试了阈值从 10 ~ 1 000,并针对每一个阈值对 TSLP 算法进行评估,发现阈值设置为 30 左右时,TSLP 算法性能最好。实验显示当设置为这个阈值后,收益开始增加。原因是发现挨着根节点剪切比贴近叶子节点剪切更重要。因为在根节点剪切,可能潜在地剪掉图中大量向量化负收益的节点。

2.3　代价模型

在已获得调节图的基础上,下一步工作就是决定是否进行向量化,或者继续保持标量执行。为了不降低生成代码的性能,TSLP 算法需要一个精确的代价模型用于评估每种情况的代价。由于本文提出的 TSLP 算法最后在编译器 LLVM 中实现,所以复用编译器 LLVM 中提供的 SLP 向量化方法的代价模型。SLP 方法的代价模型是借助编译器 LLVM 提供的指令开销模型,计算所有指令代价的和,包括将标量数据转为向量数据或者将向量数据转为标量数据等额外开销。

3　实验分析

将本文提出的 TSLP 算法在 LLVM(version 3.6)中实现,并利用一些典型应用程序的核心段评估 TSLP 算法,这些核心段来自 SPEC 2006 和 NPB 3.3。表 1 对这些核心进行了描述。以编译选项-O3-allow-partial-unroll-march = core-avx2 -mtune = core-i7 作为编译这些核心的基础选项,并用-O3 代替。对这些核心从以下三个方面进行测试:a)仅打开 O3 和循环级向量化,关闭 SLP 和 TSLP 向量化方法,并将这种情况标记为 O3;b)在 O3 基础上仅打开 SLP 向量化,将这种情况标记为 SLP;c)在 O3 基础上仅打开 TSLP 方法,并标记为 TSLP。目标系统是 Intel 的 Core i5-4570,主频 3.2 GHz,16 GB 的 RAM,SSD 硬盘。操作内核为 Linux 3.10.7,glibc 的版本为 2.17。

表 1　测试用例描述

核心	描述
compute_rhs	Xi-direction fluxes(NPB 2.3,BT)
mult_su3_mat_vec_sum_4dir	Su3 matrix by vector mult.(433.milc)
ewald_LRcorrection	Kernel from CPU 2006(435.gromace)
compute_triangle_bbox	Triangle bounding box(453.povray)
lbm_handleInOutFlow	Kernel from CPU 2006(470.lbm)
shift_LRcorrection	Kernel from CPU 2006(435.gromacs)
motivation	Section Ⅱ-B code

3.1　性能测试分析

首先对选定测试用例的性能进行评测,性能评测结果如图 5 所示。执行时间以 O3 为基准。从图 5 可以看出,与 SLP 方法相比,TSLP 方法对于大部分核心都能起到提升性能的作用。前三个核心(motivation、compute_rhs 和 mult_su3_mat_vec_sum_4dir)利用 TSLP 方法进行向量化时有加速效果,而利用 SLP 方法进行向量化时并没有加速效果。这是因为 SLP 方法生成的向量化图中包含向量化收益为负的代码,而这些代码标量执行比向量执行的节拍数更少;TSLP 方法去除了这些收益为负部分代码的向量执行,使得代码最终向量化后可以获得收益。

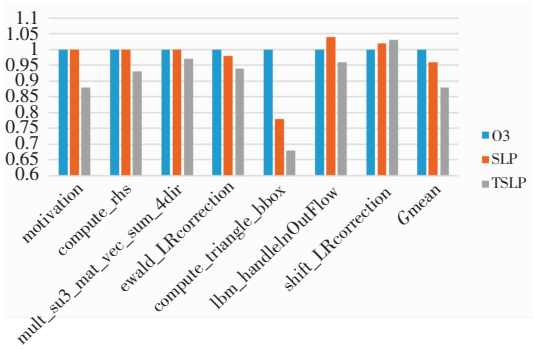


图 5　性能测试结果

对于 ewald_LRcorrection 和 compute_triangle_bbox 而言,利用 SLP 方法向量化后比 O3 选项获得的效果好。在这两个核心中向量化整个区域的代价小于标量的代价,所以 SLP 向量化能够获得性能提升。但是 TSLP 方法能够去除向量化收益为负部分的代码,因此获得了更好的向量化性能。

最后的两个核心由于代价模型不精确,导致实际性能的下降。在 lbm_handleInOutFlow 中,代价模型不精确导致 SLP 向量化后代码的执行时间比 O3 的时间长,即使代价模型认为向量化是有收益的,而 TSLP 已经移除了向量化收益为负的代码,执行效果好于 O3 和 SLP。在 shift_LRcorrection 中,由于代价模型不精确,直接导致 SLP 和 TSLP 的向量化效果都弱于 O3,而 TSLP 比 SLP 方法还弱一些。图 5 中最后一列表示选定测试核心执行时间的几何平均值,与 O3 相比 TSLP 对于选定的核心平均提供 12% 的性能提升,与 SLP 方法相比 TSLP 对于选定的核心平均提供 9% 的性能提升。

3.2　静态代价统计分析

虽然性能结果显示 TSLP 方法获得的效果好一些,但还是需要进一步通过静态统计的方法来说明 TSLP 方法的优越性,因为静态结果排除了不精确代价模型的影响。向量化总代价的静态结果对 SLP 和 TSLP 两种方法在一个特定的代价模型下进行了比较。TSLP 方法会获得代价最小的向量化子图,而 SLP 方法需要用整个图的代价去决定是否进行向量化。提升代价模型的精度能够让这两种算法都受益,但是 TSLP 方法通常情况下好于 SLP 方法,至少也会与 SLP 方法一致。

图 6 显示了根据 LLVM 的代价模型静态统计的标量、SLP 和 TSLP 的代价。为了获得一个更有意义的评测,每个测试用例的代价已经相对于标量代价进行了正规化。图 6 中的结果显示不论真实性能如何,TSLP 总能获得最好的向量化性能。TSLP 平均的代价比标量小 19%,比 SLP 方法小 9%。

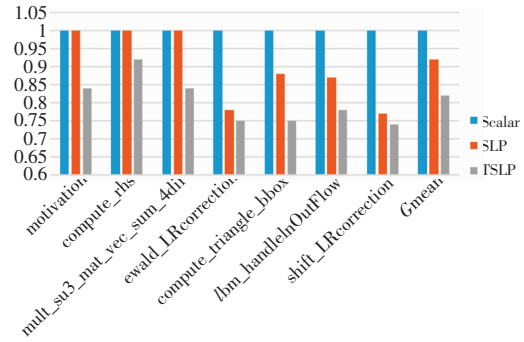


图 6　静态代价统计结果

图 6 中的静态统计与图 5 中显示的性能测试结果并不是完全一致,这主要由于两个方面的原因:a)代价模型的精确性,也就是即使在静态统计代价时认为向量化是有收益的,但是在向量执行时也有可能比标量代码慢;b)向量化代码占整个程序运行时间的比重,也就是向量化区域仅占程序执行时间的一小部分。

核心 lbm_handleInOutFlow 和 shift_LRcorrection 属于第一种情况,而 ewald_LRcorrection 和 mult_su3_mat_vec_sum_4dir 属于第二种情况。在 ewald_LRcorrection 中,仅有两个小函数完全被向量化,它们仅占整个执行时间的一部分;而 mult_su3_mat_vec_sum_4dir 的主循环保持标量执行,仅有一小部分区域转为了向量执行。然而也有一些可以预测的行为。SLP 向量化没有收益的程序不意味着任何运行时间的改变,同样当 SLP 和 TSLP 在静态时认为向量化有收益的程序,也能够看到性能的提升。

4 相关研究

循环是自动向量化处理的主要对象,早期 Allen 等人[4]在 Parallel Fortran Converter 编译器上解决了很多自动向量化领域的基础问题。后续许多工作都是对这个循环级自动向量化方法进行的改进。文献[9]提出了如何进行有效的运行时对齐问题,而文献[10]提出了有效的静态对齐概念。文献[11]提出了如何利用最少的向量数据重组指令实现程序的向量化。文献[12]提出了不连续访存程序的向量化问题。尽管对循环级向量化方法进行了很多改进工作,但是主流的自动向量化编译器如 GCC、ICC 等仍然仅能向量化标准测试集中很小的一部分[13]。编译器向量化能力较弱的原因包括三个方面:a)缺乏精确的依赖关系分析;b)较弱的程序变换能力阻碍了程序向量并行性的发掘;c)缺乏有效的代价模型。

为了能够对基本块内的直线型代码进行向量化,Larsen 等人首先提出了 SLP 方法。目前包括 GCC、LLVM 等一些编译器中已经实现了 SLP 方法。Shin 等人[14]通过将控制流转为数据流,进而利用 SLP 方法通过生成 select 指令实现了控制流的向量化。Porpodas 等人[15]通过将非同构语句转为同构语句进而可以生成更多的向量化代码。Barik 等人[16]提出了利用动态规划方法在编译器后端实现基本块级向量化代码的生成。Holewinski 等人[17]提出了一种在运行时动态的进行依赖关系分析的方法,进而利用依赖关系分析结果指导其向量化。

5 结束语

本文提出了一种基于剪切的 SLP 向量化方法,称之为 TSLP 算法。TSLP 算法通过调节代码中向量化的范围改进性能,剪切掉向量化收益为负的部分,即使这部分代码已经成功被向量化。算法的核心是根据 SLP 的原始图生成一些子图,然后根据代价模型评估不同子图的向量化收益,仅对子图内的代码进行向量化,而对子图外其他的节点保持标量执行。将本文提出的 TSLP 算法在开源编译器 LLVM 中实现。对标准测试程序中的核心进行测试结果表明,与 SLP 算法相比,本文提出的 TSLP 算法平均能够获得 9% 的性能提升。

参考文献:

[1] 高伟,赵荣彩,韩林,等. SIMD 自动向量化编译优化概述[J]. 软件学报,2015,26(6):1265-1284.

[2] Huo Xin, Ren Bin, Agrawal G. A programming system for Xeon Phis with runtime SIMD parallelization[C]//Proc of the 28th ACM International Conference on Supercomputing. New York: ACM Press, 2014: 283-292.

[3] Ramachandran A, Vienne J, Van Der Wijngaart R. Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi[C]//Proc of the 42nd International Conference on Parallel Processing. Washington DC: IEEE Computer Society, 2014: 736-743.

[4] Allen R, Kennedy K. Optimizing compilers for modern architectures [M]. San Francisco: Morgan Kaufmann Publishers, 2001.

[5] Nuzman D, Zaks A. Outer-loop vectorization-revisited for short SIMD architectures[C]//Proc of the 17th International Conference on Parallel Architectures and Compilation Techniques. 2008.

[6] Trifunovic K, Nuzman D, Cohen A, *et al*. Polyhedral-model guided loop-nest auto-vectorization[C]//Proc of the 18th International Conference on Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE Press, 2009.

[7] Kong M, Veras R, Stock K. When polyhedral transformations meet SIMD code generation[C]//Proc of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2013.

[8] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[C]//Proc of ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2000: 145-156.

[9] Chang H, Sung W. Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware[C]//Proc of International Conference on Compilers, Architectures and Synthesis for Embedded Systems. New York: ACM Press, 2008: 167-176.

[10] Eichenberger A E, Wu Peng, O'Brien K. Vectorization for SIMD architectures with alignment constraints[C]//Proc of ACM SIGPLAN Conference on Programming Language design and Implementation. New York: ACM Press, 2004: 82-93.

[11] Ren Gang, Wu Peng, Padua D. Optimizing data permutations for SIMD devices[C]//Proc of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2006: 118-131.

[12] Nuzman D, Rosen I, Zaks A. Auto-vectorization of interleaved data for SIMD[C]//Proc of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2006: 132-143.

[13] Maleki S, Gao Yaoqing, Garzaran M J, *et al*. An evaluation of vectorizing compilers[C]//Proc of International Conference on Parallel Architectures and Compilation Techniques. Washington DC: IEEE Computer Society, 2011: 372-382.

[14] Shin J, Hall M, Chame J. Superword-level parallelism in the presence of control flow[C]//Proc of International Symposium on Code Generation and Optimization. Piscataway, NJ: IEEE Press, 2005.

[15] Porpodas V, Magni A, Jones T M. PSLP: padded SLP automatic vectorization[C]//Proc of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Washington DC: IEEE Computer Society, 2015: 190-201.

[16] Barik R, ZhaoJisheng, Sarkar V. Efficient selection of vector instructions using dynamic programming[C]//Proc of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington DC: IEEE Computer Society, 2010: 201-212.

[17] Holewinski J, Ramamurthi R, Ravishankar M, *et al*. Dynamic trace-based analysis of vectorization potential of applications[C]//Proc of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2012: 371-382.