面向部分向量化的循环分布及聚合优化

韩 林[1,2] 徐金龙[1,2] 李颖颖[1] 王 阳[1]

(信息工程大学 郑州 450001)[1] (数学工程与先进计算国家重点实验室 无锡 214125)[2]

摘 要 大量循环中都存在着少数无法向量化的语句以及许多可向量化语句,循环分布通常可以将这些语句分离到不同的循环中,进而实现循环的部分向量化。目前主流的优化编译器仅支持简单激进的循环分布方法,因而导致向量化后的循环开销过大,且不利于寄存器和 cache 的重用。针对上述问题,提出了面向部分向量化的循环分布及聚合方法。首先,分析了一般循环分布的两个关键问题:语句集的划分和循环执行顺序的确定;其次,提出了面向最大聚合的凝聚图结点排序方法来指导循环合并,在不影响并行性的前提下减小了循环开销;最后,通过实验对提出的方法进行了验证。实验结果表明,对于测试用例,提出的方法能够生成正确的向量化代码,并且能够显著提高向量化程序的执行效率。

关键词 部分向量化,循环分布,循环聚合,凝聚图

中图法分类号 TP312 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.02.008

Method of Loop Distribution and Aggregation for Partial Vectorization

HAN Lin[1,2] XU Jin-long[1,2] LI Ying-ying[1] WANG Yang[1]

(Information Engineering University, Zhengzhou 450001, China)[1]

(State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China)[2]

Abstract There are a large number of loops which contain few unvectorizable statements and many vectorizable statements. Loop distribution separates these specific statements into different loops, and then partial vectorization can be achieved. Currently, the mainstream optimizing compiler just support loop distribution which is simple and aggressive, resulting in large loop overhead and bad reuse of register and cache. To solve these problems, a method of loop distribution and aggregation for partial vectorization was proposed. Firstly, two key issues were analyzed in loop distribution, which are grouping of statements and execution order of distributed loops. Secondly, a modified topological sorting method was presented to achieve better loop aggregation, which reduces the loop overhead. Finally, we evaluated the proposed method in the experimental section. The experimental results show that the proposed method can produce correct SIMD code, and can significantly improve the efficiency of implementation program.

Keywords Prtial vectorization, Loop distribution, Loop aggregation, Aggregation graph

1 引言

随着社会发展对计算能力需求的攀升及计算机硬件并行性能的不断提高,怎样获得高效的并行程序成为必须解决的问题。一个程序在运行时,相当大一部分时间都消耗在循环的执行上,因此循环优化是提高程序运行效率的主要途径之一,也成为并行编译器设计人员开发程序并行性的研究热点。在编译优化领域中,循环分布是一种循环优化,它将源循环分裂为多个目标循环,每个目标循环只对应于源循环体的一部分[1-2],因此它能将可向量化的无环语句与不可向量化的有环语句分离开来,进而实现循环的部分向量化。目前的激进循环分布方法在不影响程序正确性的前提下将循环中的语句尽可能分离出去并形成新的独立循环。虽然循环分布是实现部分向量化的有效方法,但由于循环分布后循环数量激增,导致循环开销迅速增加。

循环分布的目的是通过循环变换获得更多可并行的循环,并未考虑程序并行后的进一步优化空间,而且并行带来的收益不一定多于激进循环分布带来的开销,因此循环分布简单应用于部分向量化难以产生理想的效果。为了使循环分布能够更有效地应用于向量化,提高向量程序的执行效率,提出了面向部分向量化的循环分布及聚合优化方法,具体地,本文的主要贡献有:

到稿日期:2015-11-03 返修日期:2016-02-26 本文受郑州市科学技术局,前沿技术研究开发计划(141PQYJS558),数学工程与先进计算国家重点实验室开放课题(2013A11)资助。

韩 林(1978-),男,博士,副教授,CCF 会员,主要研究领域为高性能计算、先进编译技术,E-mail:strollerlin@163.com;徐金龙(1986-),男,博士,讲师,主要研究领域为高性能计算、编译技术;李颖颖(1984-),女,硕士,讲师,主要研究领域为编译技术;王 阳(1986-),女,硕士,讲师,主要研究领域为编译技术。

(1)分析了一般优化编译器(如open64)中采用的激进式循环分布方法。将其归结为“语句集划分”和“循环排序”两个关键步骤。具体过程见第3节。

(2)提出了面向最大聚合的凝聚图排序方法，有效地保证了排序后可聚合结点的相邻性，有利于进一步实施循环合并。具体过程见第4节。

2　相关研究

循环中通常包含少量无法向量化的语句以及许多可以向量化的语句，挖掘此类循环的向量并行性，提高程序的效率是本文研究的重点。从两个方面来描述此类适合于部分向量化的情形：1)在循环体基本块的内部存在多条可并行执行的同构语句，同时存在无法并行的单语句，这属于迭代内的部分向量化；2)由于依赖环的存在，使得循环的某些语句在多次迭代中的实例是不可以向量执行的，但其他语句是可以向量执行的，属于迭代间部分向量化。

最近的研究[3-6]论证了基本块内的部分向量化是一种有效的向量化代码生成方法，它采用向量形式执行基本块中多个同构操作，根据SIMD数据长度，同构的操作被组合到一条或多条向量指令中。文献[3]提出了SLP(超字并行)向量化方法，用来发掘基本块内的向量机会(对循环而言，即迭代内的向量化)。它可将循环体基本块内可并行执行的语句打包后采用向量指令执行，而不能并行执行的语句则保持标量串行执行，由于需要遍历所有的向量方案以得到最优或较优解，导致算法的复杂度很高，因此采用了启发式的向量化算法，先依据相邻地址引用进行初始打包，然后再通过定义-使用关系进行pack包扩展。它本身就是一种面向循环迭代内的部分向量化方法。Liu等[7]采用了一种修改的类似SLP的方法来向量化SPEC2006中数个测试用例。Park等人[4]引入了一种子图级并行化方法(Subgraph Level Parallelism, SGLP)。子图是指基本块内含有相同操作类型和数据流的同构语句组，SGLP代表这些同构语句组之间的并行性。

Barik等[5]提出了基于动态规划的向量指令选择方法来提高向量代码的质量。它去除了SLP中按照定义-使用链和使用-定义链扩展的限制，采用动态规划来寻求基本块内最优的向量化方案。该方法用代价模型来指导SIMD指令的选择，用打包冲突图提供更多包重用的机会，在包调度时决定包的执行顺序和包内语句的顺序，以减少打包、拆包的次数。文献[8]基于Open64实现了基本块内的部分向量化方法，它借鉴文献[4]的方法，采用同构树模式匹配的方法来发掘并行性，同时借鉴并改进了文献[5]的方法，实现了面向可变向量长度的并行发掘，同时在编译消耗时间上有了较大进步。

相对于迭代内的向量并行性，迭代间的向量并行性更具普遍性。文献[9]提出面向向量机的向量代码生成算法，它是发掘迭代间向量并行性的有效方法；此方法不能直接应用于当前主流的SIMD扩展部件。当前的主流编译器(GCC[10]，ICC[11]，Open64[12]和LLVM[13]等)针对文献[9]的向量化算法进行改进，提供了对SIMD自动向量化的支持，一般仅支持最内层循环的向量化。当循环具有迭代间部分向量化特性时，一般需要将可向量化的语句与不可向量化的语句区分开来，而这种功能需要循环分布来实现。

循环分布是挖掘循环细粒度并行性的一种方法，当粒度小至一条指令时，这条指令在不同迭代中的实例将由向量指令来实现，即循环的向量化。Open64[12]中提供了一种激进的循环分布功能，循环分布后产生大量循环，虽然达到了分离可向量化与不可向量化语句的目的，但分布后循环的开销过大。LLVM[13]为了实现循环的部分向量化也加入了一个循环分布的遍，它并未采用激进的循环分布算法，而是简单地将循环体中不可向量化语句组分裂到新的循环中，并未考虑寄存器与缓存的重用优化空间。虽然循环分布是实现部分向量化的有效方法，目前有关循环分布的研究仅限于发掘出更多的并行性，并未针对SIMD向量部件做优化，因此将之简单应用于部分向量化难以产生理想的效果。

为了使循环分布更有效地应用于部分向量化，提高向量程序执行效率，提出了面向部分向量化的循环分布及合并方法。循环合并是循环分布的逆操作，也是提高程序性能的有效方法，本文涉及的循环合并都是在循环分布后，并基于循环的依赖凝聚图的聚合，因此也称之为循环聚合。本文在激进循环分布的基础上，通过依赖凝聚图结点的重排序来实现循环的最大聚合。

3　激进式循环分布

循环分布属于循环变换的一种，它在不影响程序正确性的前提下，将循环中的语句尽可能分离出去并形成新的独立循环，将这种循环变换称为激进的循环分布。循环分布要解决两个关键问题：1)语句集合的划分，即哪些语句可以被单独分布出去，哪些语句必须被分布到同一循环中；2)确定分布并获得循环的执行顺序。

3.1　激进的语句集划分

编译器对程序实施的很多优化都是通过对语句顺序进行重新排序来实现的，称之为重排序变换。任何一个重排序变换只有在满足依赖的基本定理[9]时才被称为是有效的。循环分布也是一种重排序变换，因此其算法的依据也是程序中待分布循环的依赖图，根据循环内语句之间的依赖关系，判定哪些语句可以被分布出去，哪些语句被分布出去之后必须存在于同一个循环内。

在编译器中实现循环分布功能主要涉及4种依赖关系图，分别是数组依赖图adg、语句依赖图sdg和强连通分量依赖图dep_g_p和强连通分量凝聚图ac_g。数组依赖图adg是其他依赖图的基础，图中的每个顶点代表一个数组引用，每条边表示该边所连接的两个顶点之间存在一个依赖。在进入循环优化阶段后，编译系统首先构建循环的数组依赖图，将每个数组引用结点抽象成数组依赖图中的一个顶点，并根据数组引用的地址判断任意两顶点之间是否存在依赖，在对应顶点之间添加依赖边。在数组依赖图的基础上，编译器可以进一步构建循环的语句依赖图，该图中的每个顶点代表一条语句，每条边表示对应两条语句之间存在的依赖。

由于依赖环阻碍循环分布的正确性[9]，因此当语句依赖图中一个顶点存在于依赖环之中，那么此顶点对应的语句是不可以被分布的。可在语句依赖图中识别出所有的强连通分

量，如果强连通分量的顶点个数大于1，那么该强连通分量内的所有顶点都存在于依赖环中。将识别出的每个强连通分量凝聚成单个结的结点（scc结点），同时，若依赖边对应的两顶点分别位于不同的强连通分量，那么在对应的scc结点之间添加相应的依赖边，这样就得到了强连通分量凝聚依赖图，简称凝聚图。相对于原有的语句依赖图，凝聚图的顶点个数一般会减少。以1个例程来描述各种依赖图的形成情况。

例程：

```
for(i=1;i<N;i++){
S1 a[i-1]=b[i];
S2 c[i]=d[i-1]+a[i];
S3 d[i]=c[i-1];
}
```

该循环的数组依赖图、语句依赖图、强连通分量依赖图以及强连通凝聚图分别如图1(a)－图1(d)所示。对于强连通分量凝聚图而言，每个强连通分量的所有语句必须被分布到同一个循环内，才能保证源程序的依赖关系维持不变。

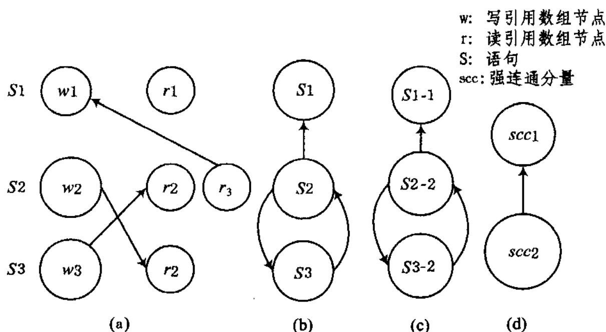


图1 例程所示循环的4种依赖图

激进的循环分布算法：根据凝聚图将所有结点全部分布出去，针对每个结点生成新的循环，循环中的语句就是强连通分量结点中的语句，不难看出，数组依赖图是基础，而凝聚图是编译器实现循环分布的最终依据。

3.2 确定循环执行顺序

上文已经介绍了编译器中实现循环分布的基本过程，确定了哪些语句可以被单独分布出去，哪些语句必须被分布到同一循环中。仍然存在的问题是循环分布得到的多个循环应该以怎样的顺序执行，针对凝聚图结点进行拓扑排序是一种有效的方法。

凝聚图的每个结点对应一个循环，分布后的多个循环必然构成一个线性执行序列，结点间的依赖要求这些循环必须按照依赖方向先后执行。如果生成的代码违背了执行顺序，那么程序就是错误的。优化编译器Open64提供的循环分布功能就没有考虑此执行序，对于例程所示的循环，在实施循环分布之后会得到如下的代码。

```
Loop1:for(i=1;i<N;i++)
        S1 a[i-1]=b[i];
Loop2:for(i=1;i<N;i++){
        S2 c[i]=d[i-1]+a[i];
        S3 d[i]=c[i-1];
        }
```

然而，对例程所示的循环进行分布后，程序运行结果与原程序结果不一致。参照该循环的凝聚图（见图1(d)），此分布结果对应的执行顺序为(scc1→scc2)，显然违背了原有的依赖。

依赖凝聚图中，所有依赖环都已被凝聚为单个结点，因此它一定是无环的。结点间的依赖是有向的，因此依赖凝聚图一定是有向无环图DAG。每个DAG都可以通过拓扑排序将图中的顶点排成一个线性序列，使得对于图中任意一对顶点u和v，若存在一条从u到v的边，那么顶点u在线性序列中将出现在顶点v之前。因此，对依赖凝聚图中的顶点进行拓扑排序在实质上保证了分布后的循环执行顺序总是沿着依赖的方向，从而维持了程序中原有的依赖。

可按下述步骤对给定的有向无环图进行拓扑排序：

1)任选一个入度为0的顶点输出。

2)删除此顶点和以它为出发点的所有边。

3)重复执行上述两步，直至图中没有入度为0的结点。

4)若输出顶点数小于图中的顶点数，则说明图中存在环，无法产生其拓扑排序，否则输出的顶点序列即为此图的一个拓扑排序。

在Open64中，基于上述步骤实现了对依赖凝聚图的拓扑排序算法，Open64编译器自带的拓扑排序算法如下。

输入：无环的凝聚图ac_g

输出：经过拓扑排序的队列queue[]

算法：

```
Topological_Sort(queue[],ac_g){
  g←ac_g;
  index ← 0;
  i←1;
  while(i<g->vertex_count_num+1){
    if(!vertex[i]->Get_In_Edge())
      queue[index++] ← i;
      else
        return;
  }
  for (i=0;i<head;i++) {
      j←queue[i];
      e←g->Get_Out_Edge(j);
      while(e){
        e1←e;
        e←g->Get_Next_Out_Edge(e);
        sink← g-> Get_Sink(e1);
        g->Delete_Edge(e1);
        if(!g->Get_In_Edge(sink))
          queue[index++]←sink;
      }
    }
  return queue[];
}
```

其中，ac_g是循环的依赖凝聚图，调用该函数之后队列queue中保存的是经过拓扑排序的结点顺序。对于例程所示的循环，经过拓扑排序后得到正确的循环分布结果如下所示。

```
Loop2:for(i=1;i<N;i++){
        S2 c[i]=d[i-1]+a[i];
        S3 d[i]=c[i-1];}
```

```
Loop1:for(i=1;i<N;i++)
    S1 a[i-1]=b[i];
```

4　循环聚合优化

用传统的方法来发掘细粒度的向量并行性时，尽可能地使用循环分布，将每条语句放到由它自己构成的循环中，即采用上述的激进式循环分布方法。而面向 SIMD 短向量并行部件，这种激进的方法是不合适的。粒度的减小，循环数量的增加，导致循环开销迅速增大，同时不利于寄存器和 cache 的重用。因此，需要在不影响并行性的条件下进行循环合并，生成尽可能少的循环。

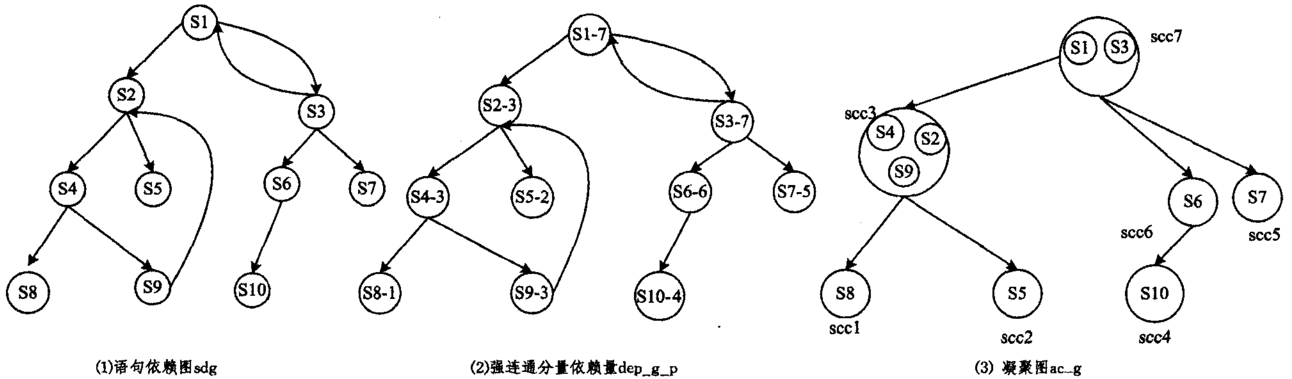
由于循环分布是基于凝聚图的，凝聚图的每个结点对应一个循环，因此循环合并可直接对应于凝聚图的结点聚合，本文也称之为循环聚合。

4.1　简单循环聚合

激进的循环分布将生成多个循环，每个循环对应依赖凝聚图中的一个结点。为了减小循环开销，可采用循环合并将多个循环合成为单个循环，对应于依赖凝聚图就是将图中的多个结点聚合为单个结点。本节在激进式循环分布的基础上实施循环合并循环，同时保证不影响循环的向量并行性，即若某循环中不存在依赖环，那么循环合并后也不存在依赖环。

第 3 节分析了循环分布的一般思路：即建立数组依赖图 adg、语句依赖图 sdg 和强连通分量依赖图 dep_g_p，并最终生成强连通分量凝聚图 ac_g；然后对 ac_g 进行拓扑排序，生成强连通分量序列 sequence。激进的循环分布方法将 sequence 中的每个结点都按顺序输出为一个循环，导致输出大量的循环。而本文的聚合方法将 sequence 中相邻的无环结点组合输出到一个循环中，相邻的环结点也组合输出到同一循环中，进而减少输出循环的个数。聚合需要遵循的原则为：1)不存在依赖环的相邻结点聚合在一起；2)存在依赖环的相邻结点聚合在一起；3)聚合为尽可能少的结点。

通过一个稍复杂的例子来展示循环聚合的效果，如图 2 所示。图 2(1)是一个循环的语句依赖图，图 2(2)为对应的强联通分量依赖图，图 2(3)为最终得到的凝聚图。



注：S7-5，代表7号语句，5号强连通分量

图 2　适用于聚合优化的依赖关系图

对图 2(3)的凝聚图进行拓扑排序后可得到图 3(1)所示的强连通分量序列，若直接对其进行激进的循环分布，将得到图 3(3)所示的结果；将凝聚图中的无环结点聚合后再进行拓扑排序可得到如图 3(2)所示的序列，对其进行循环输出得到如图 3(4)所示的结果。显然，聚合优化后产生的循环个数少于前者。

(1) scc7→ scc6→ scc5→scc4→ scc3→ scc2→ scc1
(2) (scc7)→ (scc6-scc5-scc4) →(scc3) → (scc2-scc1)

(3) loop1: S1;S3;
loop2: S6;
loop3: S7;
loop4: S10;
loop5: S2; S4; S9;
loop6: S5;
loop7: S8;

(4) loop1: S1; S3;
loop2: S6; S7; S10;
loop3: S2; S4; S9;
loop4: S5; S8;

图 3　拓扑排序指导循环聚合(1)

4.2　面向最大聚合的排序

对有向无环图进行拓扑排序的结果并不是唯一的，结点的聚合是在“排序结果”的基础上实施的，拓扑排序结果将影响聚合的结果。本文给出了另外一种拓扑排序结果如图 4(1)所示，在这种序列下进行结点聚合得到了图 4(2)所示的聚合序列，它们对应的循环分布结果分别如图 4(3)和图 4(4)所示。显然，基于本次拓扑排序，聚合优化后的结果的循环个数减少为 2，优于前一次拓扑排序。

(1) scc7→ scc3→ scc5→scc6→ scc1→ scc2→ scc4
(2) (scc7-scc3)→ (scc5-scc6-scc1-scc2-scc4)

(3) loop1: S1;S3;
loop2: S2; S4; S9;
loop3: S7;
loop4: S6;
loop5: S8;
loop6: S5;
loop7: S10;

(4) loop1: S1; S3; S2; S4; S9;
loop2: S7; S6; S8; S5; S10;

图 4　拓扑排序指导循环聚合(2)

综上所述，拓扑排序结果将影响序列中结点的聚合，进而影响最终的循环分布结果。寻找一种拓扑排序序列，使得结点序列可最大程度地聚合是本文要解决的问题。本文提出一种面向最大聚合的拓扑排序方法，使得无环结点尽可能相邻，依赖环结点也尽可能相邻。

首先将图中结点分为环结点和非环结点两种类型，对给定的凝聚图实施面向最大聚合的拓扑排序，可按下述步骤进行。

Step1　创建一个链表 list_no_inedge，用来存放当前入度为 0 的结点。

Step2　将图中入度为 0 的结点加入链表 list_no_inedge。

Step3　获得 list_no_inedge 中第一个结点的结点类型(环结点还是非环结点)。

Step4 从链表 list_no_inedge 表头开始遍历队列中的结点,如果当前结点类型与 Step3 获得的结点类型一致,那么输出该结点,删除此顶点和以它为出发点的所有边,并将由此产生的新的入度为 0 的结点插入到链表 list_no_inedge 的尾部。如果当前结点类型与 Step3 输出的结点类型不一致,那么跳过该结点。

Step5 重复 Step3 和 Step4,直至链表 list_no_inedge 为空。

上述步骤执行完后,结点的输出顺序就是最终的拓扑排序顺序。

基于上述思想,本文在 Open64 激进循环分布的基础上实现了面向最大聚合的拓扑排序方法 New_topological_Sort(),其伪代码如下。

输入:无环的凝聚图 ac_g

输出:经过拓扑排序的队列 queue[]

算法:

```
New_topological_Sort(queue[],ac_g){
  g←ac_g;
  index←0;
  i←1;
  while(i<g—>vertex_count_num+1){
    if(!vertex[i]—>Get_In_Edge())
        list_no_inedge—>insert_last(i);
    else
        return;
  }
  while(!(list_no_inedge—>isempty())){
    orgnode←list_no_inedge—>first();
    orgattr←orgnode—>attr();
    for (node←list_no_inedge—>first();
        node!=list_no_inedge—>last();
          node ←list_no_inedge—>next()){
    if(node—>attr()==orgattr){
      i←node—>value();
      queue[index++]←i;
        list_no_inedge—>delete(node);
        e←g—>Get_Out_Edge(i);
        while(e){
          e1←e;
          e←g—>Get_Next_Out_Edge(e);
          sink ←g—> Get_Sink(e1);
          g—>Delete_Edge(e1);
          if(!g—>Get_In_Edge(sink))
            list_no_inedge—>insert_last(sink);
      }
    }else{
        Continue;
      }//end if(node—>attr==orgattr)
    }//end for ( node=list_no_inedge—>first();…
  }//end while(list_no_inedge—>count())
  return queue[];
}
```

以图 2(3)所示的凝聚图为例来说明本算法的作用,首先找出图中入度为 0 的结点集合,即{scc7},将之放入链表 list_no_inedge 中,第一次输出为 scc7,如图 5(1)所示。删除表中 scc7 和以 scc7 为出发点的边,新产生的入度为 0 的结点集合为{scc3,scc6,scc5},更新 list_no_inedge,如图 5(2)所示。输出其中与 scc7 同类型的结点 scc3,新产生的入度为 0 的结点集合为{scc1,scc2},再更新 list_no_inedge,如图 5(3)所示,链表中的结点为 scc6,scc5,scc1 和 scc2,至此已无与 scc7 同类型的结点。再次执行 Step3 和 Step4,可输出 scc6,scc5,scc1,scc2 和 scc4,如图 5(3)和图 5(4)所示。链表已空,最后的输出顺序如图 5 所示,此顺序与图 4 所示的利于循环合并的拓扑排序一致。

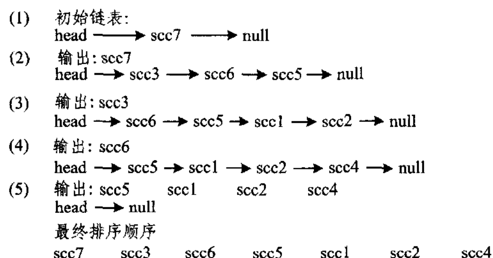


图 5 图 2(3)所示的凝聚图的排序过程

5 实验与分析

本文的方法在 SW-VEC 编译平台上得到了初步实现。可执行程序的运行平台为国产申威服务器,CPU 主频 2.0GHz,内存 2GB,L1 数据 cache 为 32kB,向量寄存器宽度为 256 位,可容纳 4 个浮点类型或 8 个整型数据。本节对提出的循环聚合优化方法分别进行正确性测试和性能测试,并对测试结果进行分析。

5.1 正确性测试

为了验证所提方法的正确性,选取了 spec2006 测试集的部分测试用例,该测试集中包含 14 个 C 程序,6 个 fortran 程序,7 个 C++程序,4 个 C 与 Fortran 混合程序。之所以只测了部分测试集,是由于聚合优化模块在 SW-VEC 平台的具体编码实现并不完善。表 1 列出了所选测试用例及其测试结果。结果表明,本文采用的聚合优化方法未对程序正确性带来不良影响,运行结果前后一致,即本文提出的循环聚合优化方法是正确的。

表 1 聚合优化 spec2006 部分用例的正确性测试

程序	401. bzip2	410. bwaves	429. mcf	433. milc	435. gromacs	437. leslie3d	456. hmmer	458. sjeng	462. libquantum	464. h264ref	470. lbm	998. specrand	999. specrand
关闭聚合优化	√	√	√	√	√	√	√						
打开聚合优化	√	√	√	√	√	√	√	√	√	√	√	√	√

(下转第 81 页)

[34] JIMENEZ Y, CERVELLO-PASTOR C, Garcia A J. On the controller placement for designing a distributed SDN control layer [C]// Networking Conference, 2014 IFIP. Trondheim, Norway, IEEE, 2014: 1-9.

[35] GUO Z, SU M, XU Y, et al. Improving the performance of load balancing in software-defined networks through load variance-based synchronization [J]. Computer Networks, 2014, 68: 95-109.

[36] CUI H, ZHU Y, YAO Y, et al. Design of intelligent capabilities in SDN [C] // 2014 4th International Conference on Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE). Shenzhen, Guangdong, China, IEEE, 2014: 1-5.

[37] CIVANLAR S, PARLAKISIK M, TEKALP A M, et al. A qos-enabled openflow environment for scalable video streaming[C]// GLOBECOM Workshops (GC Wkshps), 2010 IEEE. Miami, Florida, USA, IEEE, 2010: 351-356.

[38] CAO B, LIU S, SUN Q. Dynamically adaptive load balancing strategy under the software defined network structure[J]. Journal of Chongqing University of Posts & Telecommunications, 2015, 27(4): 460-465. (in Chinese)
曹傧,刘鹏,孙奇. 软件定义网络架构下的动态自适应负载均衡策略研究[J]. 重庆邮电大学学报(自然科学版), 2015, 27(4): 460-465.

(上接第 74 页)

5.2 性能测试

为了测试本文优化方法对自动向量化程序在性能提升方面的作用,选择 SPEC2006 测试集中的 3 个程序作为测试用例,包括 433. milc, 456. hmmer 和 462. libquantum。针对这 3 个测试用例分别生成多种优化版本(各版本对应的优化组合为:A、"串行版本",B、"激进循环分布",C、"激进循环分布+向量化",D、"循环聚合优化+向量化"),并测试各版本相对于串行源程序的加速比,加速比越高说明性能提升越明显,测试结果如图 6 所示。结果表明:

1)仅仅对程序进行激进式循环分布(B 类优化)将降低程序的性能,说明激进式循环分布确实引入了过多开销。

2)如果对程序施加 C 类优化,程序的性能较 A 类优化有明显提升,说明向量化提高了程序的性能,但性能仍存在提升空间。

3)对程序施加 D 类优化,即本文提出的优化方法,其效果是上述优化结果中最好的,456. hmmer 的加速比超过了 1.5。

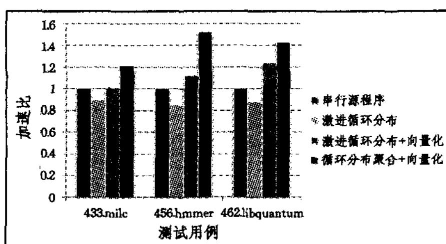


图 6　测试用例多种优化版本的加速比对比

结束语　循环分布是实现部分向量化的有效手段,但目前主流编译器中循环分布过于简单激进,尤其针对 SIMD 短向量部件,难以得到高效的向量化代码。本文提出了面向部分向量化的循环分布及聚合方法,其目的是在保证向量并行性的条件下,通过循环聚合来减小循环开销并提高向量代码的执行效率。实验结果表明,此方法是有效的。然而,本方法目前并不完善,循环聚合显然为数据重用带来了更大的优化空间,怎样在减小循环开销的同时,最大限度地提高寄存器和 cache 的重用也是值得进一步研究的问题。因此下一步工作中将针对此问题展开研究,在循环聚合的基础上实施更有效的数据重用优化。

参 考 文 献

[1] KENNEDY K, MCKINLEY K S. Loop distribution with arbitrary control flow [C] // Proceedings of Supercomputing '90. IEEE, 1990: 407-416.

[2] MCKINLEY, KEN K, KATHRYN S. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution[M]// Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg, 1997: 301-320.

[3] LARSEN S, AMARASINGHE S. Exploiting superword level parallelism with multimedia instruction sets[C]// Proceedings of the SIGPLAN'00 Conference on Programming Language Design and Implementation, 2000: 145-156.

[4] PARK Y, SEO S, PRAK H, et al. Simd defragmenter: efficient ilp realization on data-parallel architectures[J]. ACM SIGARCH Computer Architecture News. ACM, 2012, 40(1): 363-374.

[5] BARIK R, ZHAO J, SARKAR V. Efficient selection of vector instructions using dynamic programming [C] // 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2010: 201-212.

[6] KIM S, HAN H. Efficient SIMD code generation for irregular kernels[J]. ACM Sigplan Notices, 2012, 47(8): 55-64.

[7] LIU J, ZHANG Y, JANG O, et al. A compiler framework for extracting superword level parallelism [J]. ACM Sigplan Notices, 2012, 47(6): 347-357.

[8] RAMANARAYANAN R, GUPTA M, CHAKRABORTY S S, et al. Harnessing partial vectorization in Open64 compiler[C]// 2014 IEEE International Advance Computing Conference (IACC). IEEE, 2014: 813-824.

[9] ALLEN R, KENNEDY K. Optimizing compilers for modern architectures: a dependence-based approach [M]. San Francisco: Morgan Kaufmann, 2002.

[10] GCC Team. Gcc, the gnu compiler collection[OL]. http://gcc.gnu.org.

[11] Intel Corporation. Intel® C and C++ Compilers[OL]. https://software.intel.com/en-us/intel-compilers.

[12] Open64. Overview of the open 6 4 Compiler Infrastructure [EB/OL]. http://open64.sourceforge.net.

[13] CHEN K H, Shen B Y, Yang W. An automatic superword vectorization in LLVM [C] // 16th Workshop on Compiler Techniques for High-Performance and Embedded Computing. 2010: 19-27.