面向多面体模型的静态控制块识别扩展方法

夏文博[1] 胡伟方[1] 郭浩然[2]

[1](郑州大学信息工程学院 河南 郑州 450000)

[2](北京空间信息中继传输技术研究中心 北京 100094)

摘 要 在编译优化中,多面体模型可以对计算密集型程序中的耗时较多的循环代码进行并行性和数据局部性优化。但是,多面体建模过程中存在诸多限制,程序中只有少量代码可以被识别进而转换为多面体表示进行优化。基于 LLVM 编译框架提出一种分析方法,对多面体建模中的非规则因素进行了规范化处理,对非仿射因素提出一种定值扩展方法,消除了多面体建模的部分限制。实验表明,该方法在 SPEC 2006 测试集中静态控制块的识别数目增加了 44%,同时提升了多面体优化效果。

关键词 程序并行化 循环优化 多面体模型 LLVM 静态控制块

中图分类号 TP312 **文献标志码** A DOI:10.3969/j.issn.1000-386x.2022.03.004

AN EXTENSION METHOD OF STATIC CONTROL PART RECOGNITION FOR POLYHEDRAL MODEL

Xia Wenbo[1] Hu Weifang[1] Guo Haoran[2]

[1](*School of Information Engineering, Zhengzhou University, Zhengzhou 450000, Henan, China*)

[2](*Beijing Space Information Relay Transmission Technology Research Center, Beijing 100094, China*)

Abstract In compilation optimization, the polyhedral model can optimize the parallelism and data locality of the time-consuming loop code in computation-intensive programs. However, there are many limitations in the process of polyhedron modeling, and only a few codes in the program can be identified and transformed into polyhedron representation for optimization. This paper proposes an analysis method based on the LLVM compilation framework, which normalizes the irregular factors in polyhedron modeling, and proposes a fixed value expansion method for non-affine factors, which eliminates some limitations of polyhedron modeling. The experiments show that the method increases the recognition number of static control parts in the SPEC 2006 test set by 44%, and improves the polyhedron optimization effect.

Keywords Program parallelization Loop optimization Polyhedron model LLVM SCoP

0 引 言

近年来,随着以深度学习算法为代表的人工智能技术不断应用于各个领域,大量计算密集型程序对计算性能的要求逐步增高。为了利用程序中存在的并行性与数据局部性,确保程序快速执行,降低设备功耗,通用 CPU 提供了多级缓存、多核、SIMD 部件,甚至 GPU 等专用的计算加速器也用于提升程序的执行效率[1]。更加复杂的并行层次及存储器层次结构给编译优化带来了新的挑战。

计算密集型程序中往往存在着大量的循环,例如在图像处理程序中,计算核心耗时一般在图像矩阵的运算,对这些运算进行处理时,需要多层循环来表示运算操作。将循环进行适当的变换,并且映射到不同目标平台以充分发掘程序中的并行性和数据局部性则需要编译优化的支持[2]。目前,大多数工业级编译器虽然支持基本循环变换、向量化、自动并行代码生成等功

收稿日期:2020-01-13。郑州大学 2018 年科研启动基金项目(38879938)。**夏文博**,硕士生,主研领域:高性能计算。**胡伟方**,硕士生。**郭浩然**,高工。

能,但是一旦涉及需要多种复杂转换才能确保优化所需的数据局部性及并行性的情况,常规编译优化过程是难以分析并进行处理的,这严重限制了编译器对程序性能的优化能力。

多面体模型[3]是一种集成了静态控制块分析、依赖分析、循环调度变换于一体的数学表示方法。多面体模型比传统抽象语法树(Abstract Syntax Tree,AST)表示更准确地描述了程序静态信息与动态特性。作为一种数学模型,多面体模型可以更抽象地描述循环嵌套,更加精确地计算语句实例之间的依赖关系。同时,多面体模型表示可以自然地表示多种循环变换方法,例如循环交换、循环倾斜等。利用多种循环变换方法,有利于发掘程序中的向量化机会,生成 SIMD[4]代码,也可以利用循环分块等变换,帮助生成面向 GPU 体系结构的 SIMT(单指令多线程)代码[5]。随着 GPU、AI 芯片等多种全新体系结构的出现,多面体编译优化已成为一种越来越流行的技术。

对程序应用多面体优化时,首先需要找到可以被转换成多面体表示的相关代码段,称为静态控制块(Static Control Parts,SCoP),并为它们创建一个多面体模型描述。然而,SCoP 的构成必须非常规范,实际程序中往往具有许多限制循环嵌套被识别为 SCoP 的非规则因素。在通过对 SPEC2006[6]测试基准的测试中发现许多测试用例程序中的相关循环代码段存在诸多限制 SCoP 识别的问题。例如图 1 中的代码,在进行 SCoP 检测的过程中,由于大多数编译器的别名分析相对保守,编译器无法证明循环嵌套中所有数组访问不会造成重叠,会将其识别为存在别名问题,多面体识别算法会判定该循环为非 SCoP。

```
void function (int * A, int * C){
    int k,d = 2;
    for(k = 0; k < M; k++){
        C[k] = d * A[i];
    }
}
```

图 1 带有可能别名的循环结构

为了扩展多面体编译的实用性,目前已经有一些学者对限制静态控制块识别的若干非规则因素进行了研究。Benabderrahmane 等[7]利用任意边界建模循环结构,解决了 while 循环的多面体建模处理。Strout 等[8]研究了稀疏矩阵的多面体编译优化技术,并基于运行时重排序转换技术,扩展了非仿射数组下标在多面体模型中的应用。另外也有开发人员选择使用一些工具来缓解多面体建模不充分的问题,例如:SCoP 的解析器 Clan[9],要求开发人员手工标记每个 SCoP 来手动帮助编译器识别更多的 SCoP,但是这种做法也给程序开发人员带来了许多额外的负担,需要进行大量的人工分析来确定 SCoP 的位置及范围。

本文首先使用 SPEC2006 测试集进行多面体建模测试,分析并总结出实际应用程序中阻碍 SCoP 识别的主要因素。针对多面体建模的多种主要限制因素,基于 LLVM 编译框架提出了两种提升多面体建模能力的解决方案:(1)针对多面体模型的 SCoP 识别限制中的复杂循环格式问题,在编译流程中添加循环规范化遍历,对待分析的循环进行规范化处理,消除部分不规范因素的影响。(2)针对造成多面体建模失败最多的非仿射数组访问模式与循环边界的问题,利用 LLVM 的静态程序分析信息,提出了一种定值替换的方法来解决该问题。

1 SCoP 识别限制分析

为了分析程序中限制 SCoP 识别的因素,对 SPEC2006 基准测试中的 C 语言程序进行了 SCoP 检测测试与评估。在 LLVM 的多面体建模模块中的 SCoP 识别功能函数中加入诊断信息输出语句,统计导致 SCoP 识别失败的原因,分析得出的主要原因如下:

(1)非仿射的数组访问表达式。在多面体模型中,要求所有的数组访问表达式或条件表达式都必须是输入参数或归纳变量的仿射表达式,否则 SCoP 的识别算法不会将该代码所在的区域视为 SCoP,从而阻碍了多面体模型对其进行优化。举例如图 2 所示。

```
a.  仿射表达式:
      x = A[3 * i]
b.  非仿射表达式:
      x = A[3 * i * Param_X]
(Param_X 未知, x 为二元表达式)
```

图 2 非仿射表达式

此外,SCoP 中所有循环迭代的上下限也必须是输入参数或归纳变量的仿射表达式。

(2)别名。当编译器对循环的分析结果无法证明两个内存访问的基地址是否不相交时,特别是当指针与参数值相关而不是全局变量或堆栈分配的内存空间地址时,编译器的别名分析就会保守地将其识别为指针别名,导致对该内存访问所在的程序段的多面体建模失败。

(3)函数调用。在多面体模型中,调度变换需要首先分析并尝试消除程序中存在的依赖关系。然而,

在具有外部函数调用或间接函数调用的循环中,依赖关系将变得非常复杂。基于多面体模型的依赖关系分析方法也不能给出程序中精确的数据依赖关系。所以,在 SCoP 的检测过程中,往往会排除包含函数调用的相关代码段。

(4) 非规范化的归纳变量。在多面体模型对循环嵌套的抽象表示中,循环嵌套中某个语句的迭代域会被表示成多维仿射空间中的有界凸多面体,该多面体中每个整数点都代表该语句的一个执行实例。这使得在多面体建模过程中,对循环中的归纳变量具有一定的限制:在循环每次迭代中,迭代变量递增或递减幅度要求为 1。编译器在多面体模型建模之前,应尽量将循环中的归纳变量等进行规范化处理。

(5) 复杂的控制流。在多面体模型中,优化需要得到代码段中的循环的各个信息,如果程序代码段中出现了比较复杂的终止符(例如:switch、goto 等)或者具有复杂形式的控制流(通过 while 和 for 或者 if 结构无法表示),多面体模型将难以对程序的静态信息与动态特性进行建模,不能识别为 SCoP。

2　基于 LLVM 静态分析的识别优化

LLVM[10] 是一组用于构建编译器的工具和库。它围绕一种和语言以及平台无关的中间表示(LLVM-IR)构建,提供了先进的程序分析、优化和目标代码生成等模块功能。除了支持 x86-32、x86-64 和 ARM 等后端,还有针对大多数 GPU 加速器甚至硬件描述的目标代码生成。LLVM 支持目前大部分常用的编程语言的静态和即时编译。LLVM 模块化的结构、有良好可读性的 IR 和一系列有用的工具,便于开发人员进行基于 LLVM 的编译模块开发。基于 LLVM 的多面体工具的静态控制块检测实现流程如图 3 所示。

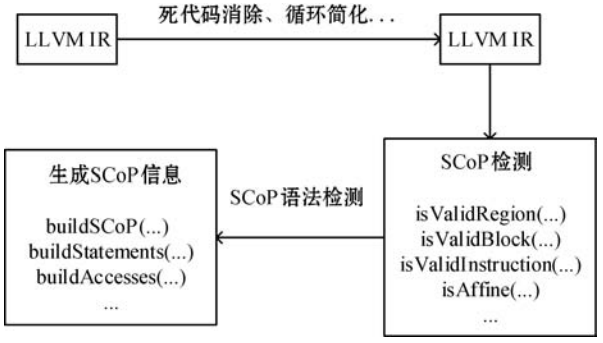

图 3　SCoP 识别流程

2.1　复杂循环格式的优化

LLVM-IR 是 LLVM 编译器的中间表示语言[11]。将源程序降级到 LLVM-IR 之后,源程序的复杂性就会显著降低,并且生成的 LLVM-IR 没有高级语言中的循环表示,只有跳转语句和 goto 语句。源程序中的数组或仿射表达式也转换为指针运算和三地址格式的操作。在这种表示中,源程序中的所有必要信息都会通过 LLVM 提供的内部分析 Pass 重新计算。所以,在 LLVM-IR 中,可以使用循环检测或支配树信息等简单分析来验证待分析程序段是否只包含结构化控制流,也可以使用更复杂的分析来检查别名或提供关于函数调用副作用的信息。这里本文选择运行一组 LLVM 规范化 Pass,进一步规范化代码,解决程序中不规范的循环格式。具体实现如图 4 所示。

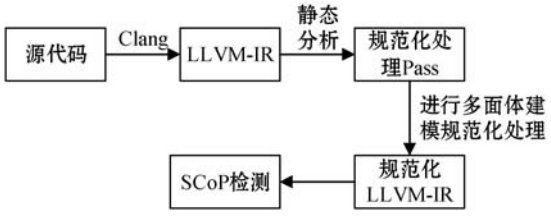

图 4　不规范循环格式处理流程

LLVM 编译框架具有非常完备的程序变换和规范化 Pass 集。其中包括基本别名分析、内存到寄存器的提升、库调用的简化、指令简化、尾部调用消除、循环简化、闭环 SSA 表单计算和归纳变量规范化。表 1 中给出了一些限制 SCoP 识别的非规范因素相应的解决方案,将这些方案加入到规范化 Pass 中,用以解决多面体的建模的一些限制。

表 1　不同限制因素解决方案

限制因素	解决方案
别名	在循环之前,引入别名检查,确保循环中不变基指针的访问不存在冲突
函数调用	对调用函数进行内联; 将 const 属性添加到函数声明中; 禁止函数读写除函数参数之外的任何值。 函数的返回值只能依赖于给定的参数
复杂的控制流	将 switch 语句转换为 if 语句; 在 LLVM-IR 层面,重新对其进行语义分析

2.2　非仿射问题的优化

如图 5 所示,多面体模型不支持维数是未知的数组。一种常用的做法是对数组进行降维。使用一维数组并“手动”执行索引算法来实现 n 维数组。这样就可以在一维数组上创建仿射下标,使用静态分析可以推断出该一维数组是二维数组的映射访问。但是,目前 LLVM 编译器中并没有相关的转换实现。另外,这样的转换也会导致代码的复杂度增加,可能会导致静态分析失败,对后续的分析造成影响。所以,非仿射表达式的主要问题是:虽然表达式的值是固定不变的,但是其表达式是非仿射的,并将其判定为不满足多面体

模型建模条件。

```
void case(int * A, int N, int image){
    int m,n;
    for(m = 0; m < N; m++){
        for(n = 0; n < N; n++){
            A[n*image + m] += A[m*n];
        }
    }
}
```

图 5 非仿射表达式

本文基于 LLVM 分析 Pass 得到的循环分析结果，对非仿射的表达式进行结果判断，通过保持恒定的参数值来消除上述形式的非仿射表达式对多面体建模的影响。具体所用的分析如下：

（1）静态单一赋值（SSA）：SSA 表示的抽象视图是一种声明性语言[12]，通过插入额外的标量变量，使每个定义都是唯一的。标量变量的赋值可以是表达式的结果，也可以是位于目标基本块中控制流节点的 phi 节点的结果。例如，图 6 中所示的程序代码通过添加新的变量名和在控制流节点处合并值的 phi 节点，将其转换为 SSA。

```
a = 0;
b = 0;
loop:
  a = a + 1;
  b = a * 2;
  if (a > 10)
    goto end;
  goto loop;
end:
c = b;
```

```
a_0 = 0;
b_1 = 0;
loop:
  a_2 = phi(a_0,a_4);
  b_3 = phi(b_1,b_5);
  a_4 = a_2 + 1;
  b_5 = a_4 * 2;
  if (a_4 > 10)
    goto end;
  goto loop;
end:
c_6 = phi(b_5);
```

图 6 SSA 转换

循环 phi 节点定义递归表达式：循环 phi 节点的第一个参数定义初始值，第二个参数使用自引用定义递归。例如：

$a = loop\_1\text{-}phi(0, a+1)$

$b = loop\_1\text{-}phi(0, 2\times a)$

a 在第一次迭代时定义为 0，并且对所有其他迭代使用自引用表达式 $a+1$；b 在第一次迭代中定义初始值 0，接着是对于所有后续迭代引用另一个 loop-phi 节点 $2\times a$ 的表达式。

循环 close-phi 节点用来计算 loop-phi 节点在循环中定义的最后一个值：它们对应于部分递归函数的 min 运算符。例如：

$c = loop\_1\text{-}cloase\text{-}phi(b, a>10)$

c 被定义为当 a 变为大于 10 时第一次迭代的 b 的值：在示例中，c 的值可以被静态地计算为 22。

（2）标量演化分析（SCEV）：从上述 SSA 表示的抽象开始，通过识别循环 phi 节点、循环闭合 phi 节点和派生的标量声明来分析标量变量演化函数。循环 phi 节点由初始值和包含自引用的表达式来声明，当自引用与当前循环中的标量值或不变表达式一起出现在加法表达式中时，SCEV 表示具有线性或仿射演化的递归函数。循环 close-phi 节点被声明为一个由表达式计算的最后一个值，该表达式可能在循环中发生变化，然后 SCEV 表示一个部分递归函数。所有其他标量声明都可以表示为从其他递归和部分递归函数的声明派生的 SCEV。SCEV 分析将为上述运行示例提供以下表达式：

$a = \{0, +, 1\}\_1$

$b = \{0, +, 2\}\_1$

$c = 22$

a 的 SCEV 分析其初值为 0，每次迭代值线性递增加 1。类似地，b 的初值为 0，每次迭代递增加 2。由于 c 的计算是在循环外，它的值并不随循环的迭代而变化，所以，在这里可以通过 SCEV 的静态分析得到表达式 c 的值为常量 22。

（3）迭代次数的分析：提供一个表示循环执行的次数的 SCEV 分析。迭代的次数被计算为闭合 phi 节点（close-phi）的 SCEV，或者 SCEV 的最后一个值，从 0 开始，在循环的每次迭代中增加 1。在运行的示例中，迭代数可以在退出循环时计算为 a 的值，并可以静态地计算为 11。

通过上述的分析可知，可以通过 LLVM 编译器中的循环分析 Pass 得到程序中循环嵌套的中表达式的静态分析值，再从中筛选出循环中重复出现的，可被定值为常量的参数值。然后，将其插入到非仿射参数或者表达式中，通过替换定值让表达式保持恒定，并创建一个新的定值循环版本。最后将调度代码插入到相应的循环检查基本块中，以在运行时确定实际参数值是否存在符合要求的版本。如果不符合，则使用原始版本。这样可以将一些原本不能看成仿射的，但是循环上下界都是常数值的非仿射表达式表示成为仿射的，从而对更多的循环进行多面体优化。其具体实现如图 7 所示。

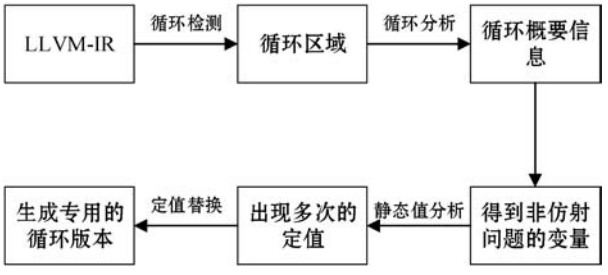

图 7 定值版本循环生成流程

多面体模型代码检测阶段,对循环代码进行分析时,如果发现循环中存在非仿射表达式,调整其判断策略:将其进行标识为待分析的仿射表达式;再判断是否存在专用的循环版本,如果存在则识别为SCoP,进行多面体优化,如果不存在,则判定为非SCoP,让其按照原来的方案进行处理。实现流程如图 8 所示。

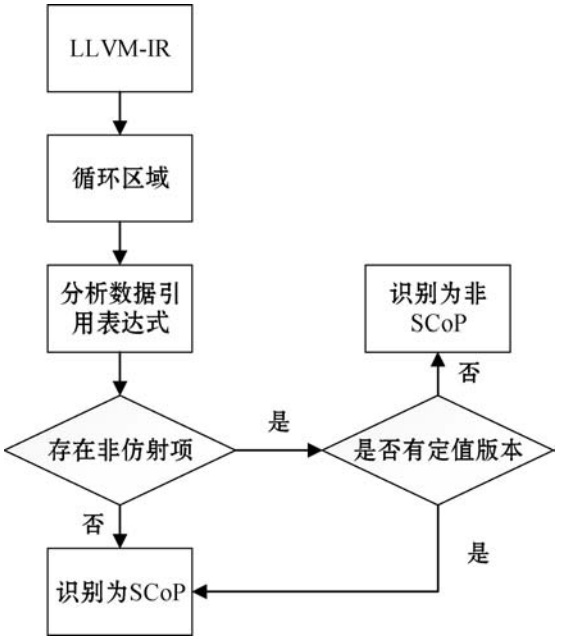

图 8　基于静态分析的非仿射表达式处理流程

3　实验与分析

3.1　测试指标与实验平台

在多面体优化过程中,SCoP 的识别是优化的基础。本文进行了如下两个方面的测试:SCoP 识别数目的优化测试和优化后的程序运行加速比测试。

(1) SCoP 识别数量测试:测试优化前和优化后,识别出来的符合多面体建模条件的静态控制块的平均数目,通过对比得出优化的效果,验证本文提出的非规范因素消除方法的有效性。

(2) 程序运行加速比测试:首先进行不引入多面体优化的基准测试;之后分别测试初始多面体优化工具和经过本文优化后的多面体优化工具测试结果。通过对比两次测试相对于基准测试的加速比,来分析SCoP 识别数目对多面体优化的影响。

测试平台采用处理器为 Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50 GHz,128 GB 内存,操作系统为CentOS7;编译器版本为 LLVM-8.0.0。

3.2　测试内容

为了对比测试优化方案在实际应用中对 SCoP 检测能力的优化效果,实验采用 SPEC2006 测试集来进行测试。分别在 LLVM 中的多面体工具和经过本文中设计的优化后的 LLVM 多面体工具上进行测试,得出识别到的 SCoP 数目的平均值。

LLVM 编译器可以使用 OPT 工具来手动实施对 LLVM-IR 的单独优化,使用-polly-detect 选项来直接测试识别出的 SCoP,其中-polly 选项代表引入多面体优化。针对优化后的版本,本文添加编译选项-polly-canonicalize,利用 LLVM 标准化 Pass 来规范多面体检测代码,非仿射表达式的处理则使用-polly-opt-nonaff 选项进行控制。

3.3　实验结果及分析

3.3.1　识别率分析

为了更确切地表示基于 LLVM 的静态分析对消除多面体建模中 SCoP 识别的优化能力,实验依然采用 SPEC2006 的 C 语言测试用例。实验结果如图 9 所示,其中黑色部分表示优化前识别出的 SCoP 数量,8 个测试用例共识别出 296 个 SCoP。白色表示经过本文方法优化后识别出的 SCoP 数目,8 个测试用例共 426 个 SCoP,平均识别数目增加了 1.4 倍。除 mcf 的识别出的循环个数没有发生改变外,其余测试用例符合 SCoP 条件的循环个数都得到了提高。并且 sjeng 测试用例静态控制块数目提升最为明显,提升了约4.3 倍。实验结果表明,基于 LLVM 的静态分析优化以及规范化方法能够有效提升 SCoP 的识别率。

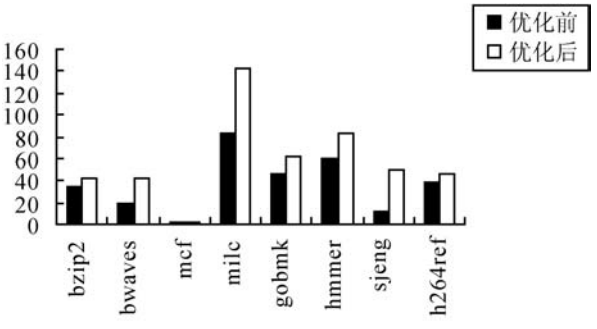

图 9　动态优化前后静态控制块数量对比图

3.3.2　加速比分析

本文采用多面体模型编译测试集 PolyBench 作为加速比测试的测试用例来测试 SCoP 识别数目对多面体优化效果的提升效果。PolyBench[13] 是一个具有 30 个数值计算的基准测试套件,包含了线性代数计算、模

板计算、图像处理和数据挖掘的计算内核。在计算加速比时,先使用经过 LLVM 编译器最优选项-O3 优化后,获取的程序运行时间作为基准时间,通过使用和不使用本文中的 SCoP 识别优化方法,分别测试程序运行的时间,通过与基准时间相比较计算出各自的加速比。

测试结果如图 10 所示,其中黑色部分表示在没有进行 SCoP 识别优化情况下取得的加速比,白色部分表示在对 SCoP 识别方法进行改进后的情况下所取得的加速比。经过优化后,测试用例中的 3 mm 运行速度最高相对于优化前提升了 2.3 倍。大部分情况下,加速比有了相应的提升,说明基于循环格式规范化和定值替换的方法可以避开多面体建模的一些限制,提升 SCoP 的识别率,发掘更多的多面体优化机会以获得性能提升。然而,也有一些测试用例出现了加速效果降低的现象,例如:mvt、symm。这种情况的出现,主要是因为新增加的一些循环代码区域本身存在一些依赖或者其他原因,造成并行化优化失败。这样就会出现优化带来的收益,不能抵消多面体建模或者 SCoP 识别带来的时间消耗的情况,从而导致程序运行速度变慢。也有另外一种原因是 SCoP 识别数目增加后,并行处理这些静态控制单元,需要更加频繁地进行线程开启和同步,造成了线程开销增加的问题,抵消了程序并行运行带来的收益。要解决这个问题,需要后续对 SCoP 的处理算法进行改进。

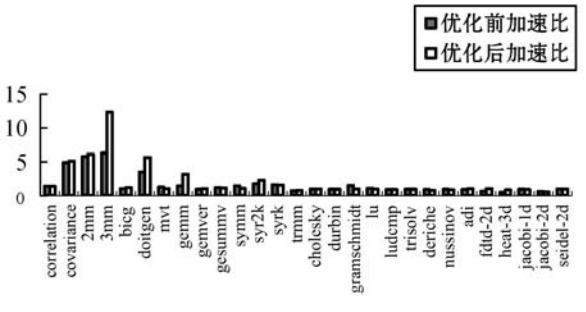

图 10 多面体优化加速对比

4 结 语

本文针对多面体建模有诸多限制的问题,对阻碍程序代码中实现多面体模型建立的因素进行了全面分析。对不规则的循环代码,在规范化处理 Pass 中整合了解决方案,对进行多面体建模识别的区域进行了规范化处理,解决了复杂的循环结构的简化和处理。针对限制多面体建模占比最多的非仿射表达式的问题,基于 LLVM 的静态分析中的标量演化,进行了预分析优化,将循环中不确定的变量参数,进行了定值化处理,生成了定值版本的循环,从而对非仿射问题进行了扩展。最后经过实验验证,本文所做的方法优化,可以有效提升 SCoP 的识别数量,增强了多面体模型的优化效果。

参 考 文 献

[1] 赵捷,李颖颖,赵荣彩. 基于多面体模型的编译“黑魔法”[J]. 软件学报,2018,29(8):2371 - 2396.

[2] 王舒心,贺细平. C 语言程序中循环结构的性能优化[J]. 电脑与信息技术,2019,27(5):67 - 69.

[3] Feautrier P, Lengauer C. Polyhedron model [EB/OL]. [2020 - 01 - 13]. https://www. researchgate. net/publication/233401207_Polyhedron_Model.

[4] 王琦,韩林等. 不充分 SIMD 向量化技术研究[J]. 计算机应用与软件,2018,35(9):108 - 112.

[5] Shirako J, Hayashi A, Sarkar V. Optimized two-level parallelization for GPU accelerators using the polyhedral model [C]//Proceedings of the 26th International Conference on Compiler Construction. ACM, 2017: 22 - 33.

[6] Integer component of SPEC CPU2006[CP/OL]. (2006 - 08 - 24)[2020 - 01 - 13]. http://www. spec. org/cpu2006/CINT2006/.

[7] Benabderrahmane MW, Pouchet L, Cohen A, et al. The polyhedral model is more widely applicable than you think [C]//Proceedings of the International Conference on Compiler Construction. Springer, 2010: 283 - 303.

[8] Strout M, LaMielle A, Carter L, et al. An approach for code generation in the sparse polyhedral framework [J]. Parallel Computing, 2016, 53: 32 - 57.

[9] Bastoul C. Clan a polyhedral representation extractor for high level programs[EB/OL]. [2020 - 01 - 13]. http://citeseerx. ist. psu. edu/viewdoc/download? doi = 10. 1. 1. 162. 283&rep = rep1&type = pdf.

[10] 莫培弘,衷璐洁. LLVM 中静态程序信息的过程间分析方法[J]. 计算机工程与设计,2018,39(6):1610 - 1618.

[11] 汪雷. 基于 LLVM 中间表示的缺陷静态分析工具实现[D]. 北京:北京邮电大学,2016.

[12] 殷文建. 面向 ARM 体系结构的代码逆向分析关键技术研究[D]. 郑州:解放军信息工程大学,2010.

[13] PolyBench[CP/OL]. (2018 - 02 - 08)[2020 - 01 - 13]. https://sourceforge. net/projects/polybench/.