

基于数据重用分析的多面体循环合并策略

胡伟方^{1,2} 陈云^{1,2} 李颖颖³ 商建东¹

1 河南省超级计算中心 郑州 450000

2 郑州大学信息工程学院 郑州 450000

3 战略支援部队信息工程大学 郑州 450000

(iewfhu@foxmail.com)

摘要 现有多面体编译工具往往使用一些简单的启发式策略来寻找最优的语句合并,对于不同的待优化程序,需要手工调整循环合并策略以获得最佳性能。针对这一问题,面向多核 CPU 目标平台,文中提出了一种基于数据重用分析的循环合并策略。该策略避免了不必要的且会影响数据局部性利用的合并限制;针对调度的不同阶段,提出了面向不同并行层次的并行性合并限制;对于数组访问关系较为复杂的语句,提出了面向 CPU 高速缓存优化的分块性合并限制。相较于以往的合并策略,该策略在计算合并收益时考虑到了空间局部性的变化。文中基于 LLVM 编译框架中的多面体编译模块 Polly 实现了这一策略,并选用 Polybench 等测试套件中的部分测试用例进行测试。实验结果表明,相较于现有的多种合并策略,在单核执行情况下,测试用例平均获得了 14.9%~62.5% 的性能提升;在多核执行情况下,多个测试用例平均获得了 19.7%~94.9% 的性能提升,在单个测试用例中最高获得了 1.49x~3.07x 的加速效果。

关键词: 编译优化;多面体模型;LLVM 编译框架;循环合并;数据局部性

中图法分类号 TP314

Loop Fusion Strategy Based on Data Reuse Analysis in Polyhedral Compilation

HU Wei-fang^{1,2}, CHEN Yun^{1,2}, LI Ying-ying³ and SHANG Jian-dong¹

1 Supercomputing Center of Henan Province, Zhengzhou 450000, China

2 School of Information Engineering, Zhengzhou University, Zhengzhou 450000, China

3 PLA Strategic Support Force Information Engineering University, Zhengzhou 450000, China

Abstract Existing polyhedral compilation tools often use some simple heuristic strategies to find the optimal loop fusion decisions. It is necessary to manually adjust the loop fusion strategy to get the best performance for different programs. To solve this problem, a fusion strategy based on data reuse analysis is proposed for multi-core CPU platform. This strategy avoids unnecessary fusion constraints that affecting the mining of data locality. For different stages of scheduling, the parallelism constraint for different parallel levels is proposed. And a tiling constraint for CPU cache optimization is proposed for statements with complex array accesses. Compared with the previous loop fusion strategies, this strategy takes into account the changes in spatial locality when calculating the fusion profits. This strategy is implemented based on the polyhedral compilation module Polly in the LLVM compilation framework, and some test cases in test suites such as Polybench are selected for testing. In the case of single-core testing, compared with the existing fusion strategies, the average performance is improved by 14.9%~62.5%. In the case of multi-core testing, compared with the existing fusion strategies, the average performance is improved by 19.7%~94.9%, and the speedup is up to 1.49x~3.07x.

Keywords Compiler optimization, Polyhedral model, LLVM, Loop fusion, Data locality

1 引言

随着计算机体系结构的发展,计算机的并行层次以及存储器层次结构越来越复杂,为编译器的自动优化带来了巨大的挑战。作为编译优化方法的一种,循环合并是提升程序数据局部性、减少访存时延的重要手段。对循环进行恰当的循环合并可以在不影响并行性的前提下,提高寄存器或高速缓存

的利用率。对于具有较多循环嵌套与数据访问的应用,如图像处理、深度学习等领域的相关应用,使用循环合并方法往往能够有效地提升程序性能^[1-2]。

在早期的一些研究中,对循环合并的考虑往往脱离了其他的循环优化方法,忽略了循环变换与循环合并之间的交互作用。同时,这些循环合并方法的研究对象以循环层为最小的研究单位,而不是语句执行实例。因此在一些情况下可能

难以得到最优的循环合并结果。

基于多面体模型的编译优化是近年来发展较为迅速的编译优化方法。与传统的编译优化模型相比,其最重要的特征之一是对程序中的每个语句执行实例建模。通过对程序中的静态控制块(Static Control Part, SCoP)进行分析,得到语句的迭代域、依赖关系及调度。基于多面体模型的编译优化方法具有应用范围广、表示能力强、优化空间大等优点,极大地提升了编译器的优化能力^[3]。目前,基于多面体模型的编译优化已应用于多个编译工具,如 PLuTo^[4], PoCC 以及 PPCG^[5]等。多面体编译方法也被应用于典型开源编译器中,例如 GCC 中的 GRAPHITE^[6], LLVM 编译框架中的 Polly 模块^[7]。另外,在面向深度学习应用的编译优化中,多面体编译也发挥了相当重要的作用,例如在 Tensor Comprehensions^[8], Tiramisu^[9], MLIR^[10]等 AI 编译框架中,使用多面体模型对深度学习应用的底层 IR 进行自动调度。

多面体编译模型中,循环合并与分布对应于是否对依赖图中的成分进行合并或分离调度的计算。在对语句较多且较复杂的静态控制块进行调度计算时,循环合并通常具有较大的搜索空间,目前的多面体编译工具往往采用一些启发式方法来决策语句之间的合并。例如,在 PLuTo 算法中,依据一定的分割策略对图中的强连通分量进行分离调度。整数线性规划库(Integer Set Library, ISL)^[11]中的增量调度方法首先计算依赖图所有强连通分量的调度,接着依照一定的策略来进行多个强连通分量的合并调度计算。

这些合并策略在许多复杂的测试用例中都取得了较好的结果。然而,在多面体调度计算过程中,若在不同调度阶段保持固定不变的限制条件,则可能会失去一些本来可以利用数据局部性的机会。另一方面,虽然用户可以使用编译选项来调整部分合并限制以指定不同的优化目标,但很难使用同一选项在面向不同程序情形时都取得不错的合并结果。此外,现有的合并策略对合并收益的评估只考虑了合并带来的时间局部性收益,没有准确地评估合并带来的空间局部性变化。

本文提出了一种面向多核 CPU 的循环合并策略。该策略基于多面体编译模型中的增量调度方法,通过对依赖图中强连通分量构成的集合内部不同的数据重用项进行分析,并将其用于上述集合的合并限制条件以及合并收益评估,可以在不影响必要的后续优化的前提下,尽量合并语句以提升数据局部性。尤其是,通过分析待合并集合中的数组访问模式数目来判断是否需要保留调度带(schedule band)长度相关限制,并通过最高阶数据重用项数目对合并的局部性收益进行评估。

本文在 ISL 库中的增量调度方法的基础上实现了这一合并策略,并基于 LLVM 中的多面体编译模块 Polly 进行了实验。实验使用的测试用例来自 Polybench, Spec2000 等测试套件以及 PLuTo 编译器中的测试集。在单核执行情况下,相较于性能表现最优的传统合并策略,本文提出的合并策略使测试用例平均获得了约 14.9% 的性能提升。在多核执行情况下,相较于性能表现最优的合并策略,所提策略使测试用例平均获得了约 19.7% 的性能提升,在 advect3d 等测试用例中最高获得了约 1.5x 的加速效果。实验结果表明,本文提出的合并策略在不同的测试用例中均取得了较好的合并效果。同时,该策略在部分测试用例中获得了比其余合并策略更优的

性能。通过与当前常用的增量调度方法中的循环合并策略进行比较,所提合并策略能够在调度计算的不同阶段动态调整合并限制以得出更优的合并决策,具有更强的灵活性。

2 相关工作

循环合并是编译优化中的重要组成部分,初期的一些工作研究了以保留并行性、减少并行同步开销并提升数据局部性为目标的循环合并方法^[12-13]。其中, Kennedy 等证明了寻找最大化局部性的循环合并方式是 NP 难问题。另一部分研究工作将循环合并与其余循环变换相结合,如循环移位、分块等,但并没有将循环合并放在一种统一的循环变换模型中进行研究。在很多情况下,循环合并只有在进行了循环置换、循环倾斜等变换后才是合法的。另一方面,这些研究使用的依赖关系分析模型较为简单,且合并的最小粒度为循环层,会导致错过一些较为复杂的循环合并方式。

在基于多面体模型的循环合并方法中, Feautrier 调度算法与划分平面方法^[14-16]虽然可以表示程序中的循环合并与循环分布状况,但并没有提供有效的循环合并方法。Bondhugula 等在以往的划分平面方法的基础上,通过定义一种有效的代价模型来求解满足程序依赖关系的语句调度^[17]。在这一模型中,依赖图中强连通分量之间依赖边的切割状况对应于多面体语句之间的合并状况,当两个强连通分量不具有相同的调度维度时,可考虑对它们之间的依赖边进行切割,从而完成循环分布。之后,在线性规划问题中, Bondhugula 等在保留并行性限制的同时加入了硬件预取特性限制,以寻找较优的循环合并^[18]。Louis-Noël 等构建了多面体语句全序关系的凸空间,并使用依赖限制进行简化,得到的调度空间可以表示所有可行的循环合并情形^[19];同时,他们还利用迭代编译的方法对该空间进行搜索,在确定了选用的语句合并状态后将对应的调度限制加入划分平面的求解限制中^[20]。尽管该方法在较小的计算内核中能获得较优的结果,但其构建的循环合并可行解空间会随着多面体语句数目的增加而呈指数级增长,导致时间复杂度极高。

Mehta 等在原有的多面体循环合并方法的基础上增加了挑选预合并调度的过程,依照特定的优化原则对依赖图中强连通分量的先序关系进行重排^[21],例如,具有相同维度的强连通分量在依赖图中应尽量紧邻。Verdoolaege 等^[5]提出了增量调度方法,该方法不对依赖图中强连通分量进行分割,而是首先对依赖图中每个强连通分量计算调度,并依次挑选源点和汇点之间可对齐的维度最高且在图中最相邻的集合进行合并调度的计算;同时,合并过程必须满足若干限制条件。Zinenko 等提出了一种算法模板^[22],对多面体模型中的空间局部性进行了建模,并在求解调度时考虑了空间局部性;另外,当外层已有足够并行性时,不再保留合并时的并行性限制。然而,该方法并没有将数组访问模式及空间局部性分析引入合并限制中,虽然在求解计算调度时能够得到具有较优空间局部性的调度,但在进行强连通分量集合的合并后,调度可能会发生变化,反而导致局部性变差。

本文提出的合并策略将原有增量调度方法中的并行性限制划分为面向外层并行性及内层向量并行性两种,并提出了可根据强连通分量集合中数组访问模式数目进行调整的分块

性限制。同时,对合并前后集合内的数据重用项数进行评估,将合并后产生的空间局部性变化考虑在内,从而更加准确地评估合并收益。

3 多面体编译模型与循环合并

本节主要介绍多面体编译模型对程序中存在的静态控制块进行优化调度的过程,以及多面体编译中的循环合并方法。

3.1 多面体编译模型

多面体模型对程序中存在的静态控制块进行建模。一个静态控制块指一组连续的语句组成的集合,且这些语句所在的循环以及条件表达式都是循环迭代变量以及参数的仿射函数。对于符合约束条件的静态控制块,可以分析得到每个多面体语句的迭代域以及语句执行实例间的依赖关系。通过将迭代域中的每个元素映射到一个时间戳,可以得到语句执行实例之间的先序关系。这一映射关系称为语句实例的调度,依据该时间戳指定的先序关系生成代码的过程实质上等价于进行一系列的程序变换,如循环置换、循环倾斜等。

ISL 库是目前大多数多面体编译工具使用的调度求解工具。ISL 中的默认调度求解算法为基于 Bondguhuala 等提出的 P_{Lu}To 算法。求解调度时,为每个多面体语句计算一个调度函数序列。调度函数形如 $\phi_S = i \cdot c + p \cdot d + C$ 。其中 i, p 分别为迭代变量及静态控制块中参数组成的向量。 c, d 为未知整数向量, C 为未知整数值。若语句 S, R 之间存在依赖关系,且 S 为源点, R 为汇点,当调度函数满足 $\phi_R(i, p) - \phi_S(i, p) \leq 0$ 时,该调度符合这一依赖关系。此时可使用法卡斯引理将依赖关系转换为对调度系数的约束,并构建整数线性规划问题对其进行求解,以得到保留程序语义的调度函数。另一方面, $\phi_R(i, p) - \phi_S(i, p)$ 代表依赖距离的值,当依赖距离足

够小时,就有可能从该依赖关系中获得数据重用。通过应用法卡斯引理,可以获得依赖距离的上界,即 $\phi_R(i, p) - \phi_S(i, p) \leq u \cdot p + w$, 其中 u 为行向量。此时可将调度的求解转换为一个字典序的最小化问题,可获得符合程序语义且最小化依赖距离的调度。

对于上述给定的约束条件,调度算法迭代地求解出调度函数,并且保证在同样的约束条件下得到的调度应是线性无关的。当对同样的约束条件求解出多个调度函数时,这些调度函数组成的序列构成了一个交换带(permutable band)。即由于交换带中每一维的调度函数都符合同样的依赖关系,它们之间可以任意互换位置。若求解调度函数的过程中为线性规划问题添加了并行性约束,则计算出的对应维度的调度是具有并行性(coincidence)的。当求解算法无法获得有效的解时,将目前已有调度中已满足的依赖关系去除,继续计算下一个交换带。

在 ISL 中使用调度树(schedule tree)^[23] 表示调度。调度树包含多种节点,其中最基本的节点类型包括:调度带(band)节点,用于表示不同语句实例之间的执行次序,通过使用多维分段拟仿射表达式元组来指定语句实例的部分调度;顺序(sequence)节点,用于表示其子节点所代表的语句执行实例按照给定的顺序执行。每一个顺序节点的子节点都是一个 Presburger 集合,该集合表示一组语句实例,称为筛选(filter)节点。例如,对于 Polybench 测试集中的 gemver 程序,图 1(a)展示了其核心运算部分代码段,其对应的部分调度树如图 1(b)所示。其中, $S1, S2$ 的调度带中均有两个成员,即包围语句 $S1, S2$ 的两层循环均可进行循环交换。对于 $S2$,最外层循环是可并行化的,而对于 $S1$,外层循环和内层循环均可进行并行化。

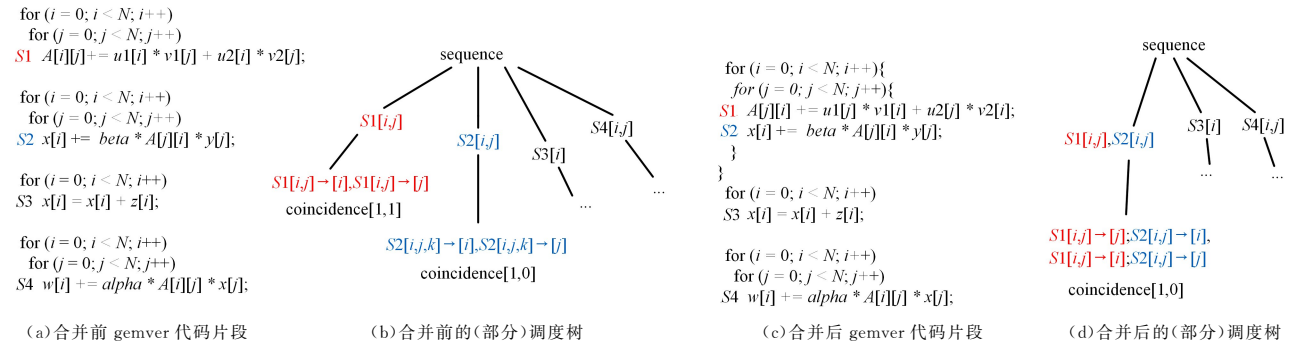


图 1 合并前后的 gemver 程序及其调度树

Fig. 1 gemver program fragments and schedule trees before and after loop fusion

3.2 多面体编译中的循环合并

多面体模型为给定的静态控制块构建依赖图,依赖图中的节点为静态控制块中的语句,节点之间的边表示依赖关系。图中强连通分量是否共享某一部分调度代表循环合并与循环分布的情况。在原始的 P_{Lu}To 算法中,首先提取依赖图中的弱连通分量,对每一个弱连通分量按照上述调度计算方法进行调度求解。当无法得到有效解时,根据不同的策略对弱连通分量中的强连通分量进行分割,接着继续对分离开的强连通分量进行调度计算,并通过在分离层插入标量维调度来实现循环分布。

增量调度方法从依赖图中的每一个强连通分量出发,首

先为每一个强连通分量构建由其自身组成的集合,并分别计算调度。接着,依据强连通分量之间的依赖边,迭代地挑选调度维度对齐数目最大的集合对尝试进行合并,当存在多个这样的集合对时,挑选图中最邻近的一对。确定待合并的依赖边后,分析两集合中强连通分量之间的依赖关系,若两集合之间存在间接的依赖关系,那么还需要将中间集合加入到待合并的强连通分量列表中,并为待合并集合中包含的节点以及相关的边构建新的依赖图,然后进行调度求解。尝试进行集合合并的过程中,根据优化目标的不同,合并需要满足不同的限制。

(1) 调度长度限制:调度长度限制要求合并后的调度带中

调度函数的数目不会减少,以避免产生不完备的合并调度。

(2)并行性限制:并行性限制要求合并后的调度带中具有并行性的调度函数的数目不会减少,用于保证合并不会损失并行性。

(3)有效性限制:合并后应保证至少有一条依赖边被优化,即在合并后的调度中该依赖边的依赖距离足够小。

依据上述合并限制判断该合并是否可以被接受,如果这一合并不符合某一限制条件,在之后的合并过程中,涉及到该依赖边的所有合并都会被拒绝。例如,对于图 1(a)~图 1(c)中描述的循环合并,依赖图中共有 4 个强连通分量集合,分别由语句 S1~S4 单独构成,且集合 1 有到集合 2 的依赖边。对集合 1、集合 2 单独计算调度,合并计算调度后得到的调度结果分别如图 1(b)、图 1(d)所示。合并后计算得到的调度中,语句 S1 的最外层调度带并行成员数目变少,但调度带长度不变。当合并策略只保留调度带长度限制时,这一合并是可以接受的。

当依赖图中仍存在多个强连通分量集合或未尝试进行合并的边时,重复上述过程直到原依赖图中只有一个集合,或所有连接不同集合的依赖边都被拒绝合并。当依赖图中仍有多个集合时,插入顺序节点,将多个集合按照依赖边定义的拓扑顺序进行排序。

4 依赖图强连通分量集合内的数据重用

现有增量调度方法中的合并策略相较于以往的循环合并方法性能表现更好。然而,这一合并策略并没有将语句的数组访问情况考虑在内,并且忽略了语句合并带来的空间局部性影响。受 Zinkenko 等的研究中对空间局部性建模的启发,本文首先计算强连通分量集合内部存在的两类数据重用,并分析合并集合中存在的数组访问模式数量以及数据重用的变化。数据重用包括迭代间的数据重用与迭代内的数据重用。

给定维度为 m 的数组 A ,对其具有数据访问的语句 S_1 , S_2 的迭代变量分别为: $\mathbf{i}=(i_1, i_2, \dots, i_{n_1})$, $\mathbf{j}=(j_1, j_2, \dots, j_{n_2})$ 。语句 $S(i)$ 对 A 各维度的访问函数分别为 $f_1(i)$, $f_2(i)$, $\dots$, $f_m(i)$ 。

定义两种数据重用条件:迭代间数据重用和迭代内数据重用。其公式分别如下:

$$\mathbf{i} \rightarrow \mathbf{i}_p: f_k(\mathbf{i}) = f_k(\mathbf{i}_p) \wedge \left\lfloor \frac{f_m(\mathbf{i})}{C} \right\rfloor = \left\lfloor \frac{f_m(\mathbf{i}_p)}{C} \right\rfloor \quad (1)$$

$$\mathbf{i} \rightarrow \mathbf{j}: f_k^1(\mathbf{i}) = f_k^2(\mathbf{j}) \wedge \left\lfloor \frac{f_m^1(\mathbf{i})}{C} \right\rfloor = \left\lfloor \frac{f_m^2(\mathbf{j})}{C} \right\rfloor \quad (2)$$

其中, C 为缓存粒度; $k=1, 2, \dots, m-1$; $\mathbf{i}, \mathbf{i}_p$ 为语句 S 在指定调度下邻近的两个语句实例变量组成的向量。 $\mathbf{i}, \mathbf{j}$ 分别为语句 S_1, S_2 在同一迭代中执行的语句实例变量所组成的向量。

迭代间数据重用表明一个语句在某一种调度下,相邻的两次迭代所访问的数组数据是否在同一缓存粒度内。当循环合并前后调度发生变化时,可使用该条件判断合并是否会造成语句本身空间局部性的损失。迭代内数据重用表明同一迭代内执行两个分属于不同语句的语句执行实例之间对数组元素的重用关系。特别地,当 $f_m^1(\mathbf{i}) = f_m^2(\mathbf{j})$ 或 $f_m(\mathbf{i}) = f_m(\mathbf{i}_p)$ 时,这种数据重用关系代表了时间(temporary)重用关系。

为了完成语句合并,不同集合内强连通分量中语句的调度可能会发生变化,问题在于如何根据给定调度得到 $\mathbf{i}, \mathbf{i}_p$ 以

及 $\mathbf{i}, \mathbf{j}$ 之间的对应关系,即需要计算出在给定一种调度时在某一或相邻迭代中执行的两对数据访问。

对于迭代间数据重用,若语句调度为 $S \subseteq (\mathbf{i} \rightarrow \mathbf{o})$,构建 $S_p \subseteq (\mathbf{i}_p \rightarrow \mathbf{o}_p)$ 以及二元关系:

$$\mathbf{I}: \{\mathbf{i} \rightarrow \mathbf{i}_p \mid f_k(S) = f_k(S_p) \wedge f_n(S_p) = f_n(S) + 1\} \quad (3)$$

其中, $k=1, 2, \dots, n-1$; $f_k(S)$ 为语句 S 调度中第 k 维关于迭代变量的仿射表达式。当 $\mathbf{I}$ 中存在符合式(1)的实例时,该数组访问具有迭代间的数据重用关系。

对于迭代内数据重用,若语句 S_1, S_2 的调度分别为 $S_1 \subseteq (\mathbf{i} \rightarrow \mathbf{o}_1)$ 和 $S_2 \subseteq (\mathbf{i} \rightarrow \mathbf{o}_2)$,其中 $\mathbf{i}, \mathbf{j}$ 分别为 n_1, n_2 维向量,且 $\mathbf{o}_{1i} = \mathbf{o}_{2i}$, $i=1, 2, \dots, \min(n_1, n_2)$ 。若 $n_1 \neq n_2$,则将维度较小的调度在尾部补零至维度相等。由于处于同一迭代内的两语句执行实例具有相同的调度值,构建同一迭代内语句实例之间的二元关系:

$$\mathbf{R}: \{\mathbf{i} \rightarrow \mathbf{j} \mid f_k(S_1) = f_k(S_2)\} \quad (4)$$

其中, $k=1, 2, \dots, \max(n_1, n_2)$ 。当 $\mathbf{R}$ 中存在符合式(2)的实例时,两数组访问具有迭代内的数据重用。

当式(1)、式(2)中 $f_k(\mathbf{i}) - f_k(\mathbf{i}_p)$ 或 $f_k^1(\mathbf{i}) - f_k^2(\mathbf{j})$ 包含迭代变量,或者当 $\mathbf{i}, \mathbf{j}$ 具有不同的维度时,意味着对应的数据重用只会在某些特定迭代下得到利用。为了简化问题并降低分析过程的复杂度,当将上述分析用于循环合并策略时,本文只关注对于任意迭代变量取值,相应数组访问维度差值均为常数的情况,即只有最高阶的数据重用机会才会被考虑。当满足这样的条件时,除了访问数据在缓存行开头的语句实例,语句迭代域内的每一个语句执行实例在执行时都会获得数据重用。

例如,若语句 S_1, S_2 在合并后的调度为:

$$S_1(i_1, i_2, i_3) \rightarrow (i_2, i_3, i_1 + 1)$$

$$S_2(j_1, j_2, j_3) \rightarrow (j_1 + 1, j_2 - 1, j_3)$$

对同一数组 A 的访问关系为:

$$S_1(i_1, i_2, i_3) \rightarrow A(i_2, i_3 + i_2, i_1 - 1)$$

$$S_2(j_1, j_2, j_3) \rightarrow A(j_1 + 1, j_2 + j_1, j_3 + 3)$$

构建 $\mathbf{R}: \{\mathbf{i}' \rightarrow \mathbf{j}' \mid i_1' = j_3' - 1, i_2' = j_1' + 1, i_3' = j_2' - 1\}$,则在同一迭代内,两语句访问数组下标的差值为 $(i_2' - j_1', i_3' + i_2' - j_2' - j_1', i_1' = j_3' - 1, i_1' - 1 - j_3' - 3) = (0, 0, 5)$ 。当 $C > 5$ 时, S_1, S_2 两个数据访问符合条件(2),具有迭代内的数据重用关系。

5 基于数据重用分析的合并限制

循环合并与分布的情况取决于依赖图中存在的每个依赖边所连接的两个强连通分量集合是否能被合并。增量调度算法通过引入若干限制来保证合并不会影响到重要的循环优化,并且尽量保证程序的局部性能得到提升。在考虑进行循环合并时,将待合并的集合合并在一起,并重新构建 ILP 问题来求解合并后的调度。通过判断合并后调度是否减少了调度带中的成员以及并行成员的数目,来决策这一合并是否可以被接受。然而,在面向多核处理器的优化中,并行性限制可能会产生对“冗余”并行性的限制,从而阻碍数据局部性的利用。即合并后虽然并行成员数目减少,但仍保留了足够进行硬件映射的并行性,在这种情况下进行合并是可以接受的。调度带长度限制能够保证循环合并后的分块性,例如,对于 GPU

平台,外层交换带长度足够进行分块并映射到 Block,Thread 等并行层次。而对于多核 CPU 平台,应保证内层调度带具有足够的成员以进行缓存分块。实际应用中,这种固定不变的调度长度限制在一些情况下同样会造成局部性损失。

针对上述问题,本文面向多核 CPU 平台提出两种合并决策限制条件,包括并行性限制以及基于数据重用分析的分块性限制,并提出了基于数据重用分析的合并收益评估方法。

5.1 并行性限制

为了在保证必要的并行性的同时避免冗余的并行性限制,通过判断合并后集合中是否存在足够的并行性以映射到目标平台来对合并进行决策。即,当需要具备 n 层并行性以完成硬件映射时,若待合并的集合中存在满足该条件的调度而在合并后不满足该条件,则这一合并会被拒绝。针对多核 CPU 平台,在以下两个调度计算情况中对并行性进行限制。

(1)当前计算的调度没有外层调度带,或者外层调度带没有具有并行性的调度函数。

(2)当前计算的调度为最后一层调度带。

在第一种情况中,该限制保证了在外层发生的循环合并不会影响到对原本可以进行并行化的循环进行并行化。另一方面,当某一个最内层调度带具有并行成员时,其所代表的内层循环可以通过循环交换以发掘向量化机会。这一过程在部分多面体编译工具中被称为预向量化(pre-vectorize)。基于这一考虑,当计算的调度已经是最后一层调度带,即调度对于每个语句都完备时,循环合并必须考虑到并行性限制。

5.2 分块性限制

为了避免循环合并造成缓存分块机会的丢失,并尽量避免不必要的调度长度限制,在本文提出的合并限制条件中,只引入特定的强连通分量集合以保留内层语句分块性的调度长度限制。

分块性限制:假设集合中最高维语句 S 的维度为 m , S 在合并后集合内的调度带长度为 n 。当语句 S 在原集合的调度中满足缓存分块条件,即该调度中的最后一个调度带长度大于 2 时,要求 $m-n \neq 1$ 。

当待合并集合中数组访问模式较多时,应优先保证该集合中最内层语句的可分块性,此时需要在合并过程中引入上述分块性限制。使用该分块性限制代替原有基于调度带成员数目的调度长度限制,可以避免出现外层可进行分块但内层循环无法分块的情形。

为了评估是否应对合并进行分块性限制,在确定待合并的依赖边并找到所有待合并集合后,首先挑选出具有最高维语句的待合并集合,依次计算集合中的最高阶数组访问模式数目,如算法 1 所示。

算法 1 计算强连通分量集合在特定调度下的数组访问模式数目

输入:强连通分量集合 C 、调度 S 、对应的数组访问关系列表 $AccessMapList$

输出:在调度 S 下 cluster C 的最高阶数组访问模式数目

- 1. $P_C^S = 0$
- 2. for each A_i in $AccessMapList$ /* $AccessMapList$ 中的数组种类集合 $\{A_i\}$ */
- 3. $mapList \leftarrow$ search maps related to A_i in $AccessMapList$
- 4. $reusePatternList = []$

- 5. for each $mapinmapList$:
- 6. if $\dim(\text{domain}(\text{map})) < C_{\text{maxvar}} / * C_{\text{maxvar}}$ 为集合内最高维语句变量数 */
- 7. continue
- 8. for each $pattern$ in $reusePatternList$
- 9. for each $patternMap$ in $pattern$
- 10. extract schedules for map and $patternMap$
- 11. if satisfy(2)
- 12. $pattern \leftarrow map, goto 4.$
- 13. endfor
- 14. endfor
- 15. $newPattern = []$
- 16. $newPattern \leftarrow map$
- 17. $reusePatternList \leftarrow newPattern$
- 18. end
- 19. $P_C^S = P_C^S + \text{len}(\text{reusePatternList})$
- 20. end

其中, $AccessMapList$ 是独特化的,即如果 $list$ 中存在某一语句对一数组的多个访问,则这多个访问的访问函数一定是不同的。

由于合并可能导致新计算出的调度带并不完备,在这一情况下难以对合并后的集合进行数据重用的分析,因此,该算法应用于每一个具有最高维语句的待合并集合。对于一个待合并集合,算法 1 计算其在单独调度的情况下,集合内部语句之间的最高阶迭代内重用,并将其聚集成为数组访问模式 $pattern$ 。当一个数组访问与某个访问模式内任意一个数组访问之间具有最高阶的迭代内重用关系时,将其加入到该 $pattern$ 中。当某个数组访问不和任意已存在的模式具有重用关系时,则为该数组访问新增一个数组访问模式。对每一个数组类别进行这样的分析,统计最终得到的数组访问模式数目。该访问模式数目近似待合并集合最内层语句执行时需要访问的不同缓存行数。

最后,当该数组访问模式数目大于某个阈值时,在合并过程中对其施加分块性限制。对于不同平台、不同的存储器层次结构,这一阈值可能是不同的。在本文中,该阈值设置为 L2 缓存相联度的 1/2。

5.3 基于数据重用分析的收益评估

在增量调度方法中,使用依赖边源点和汇点对合并后强连通分量集合内的依赖距离进行收益评估。这一评估只考虑了合并带来的时间局部性变化,但忽略了合并前后可能发生的空间局部性变化。

为了更好地评估合并前后数据局部性的变化,分别计算合并前后集合中存在的两种最高阶数据重用项数。对单个集合计算最高阶数据重用项数的算法如算法 2 所示。

算法 2 计算强连通分量集合在特定调度下的最高阶数据重用项数

输入:cluster C 、 C 中各语句的调度集合 S 、对应的数组访问关系列表 $AccessMapList$ 。/* 要求 C 中所有语句的调度都是完备的 */

输出:在调度 S 下 cluster C 的最高阶数据重用项数

- 1. $R_C = 0$
- 2. for each A_i in $AccessMapList$
- 3. build $mapList$ and $reusePatternList$ as algorithm 1.
- 4. $R_C += \text{len}(\text{mapList}) - \text{len}(\text{reusePatternList})$

```
5.   for each map in map_list
6.       extract schedule for map
7.       if satisfy(1) Rc=Rc+1
8.       end
9.   end
10. Rc=Rc * Cmaxvar
```

算法 2 对每一个数组计算最高阶的数据重用项数,收集所有对该数组的访问关系,并且要求该访问关系属于集合中的最高维语句。首先,该算法通过计算被聚集到现有的访问模式中的访问关系数目,得到符合条件的迭代内重用项数。接着,对于集合中最高维语句中的每一个数组访问,该算法判断其在当前集合的调度下是否具有迭代间重用,得到集合内的最高阶迭代间重用项数。最后,为了处理不同集合内的最高维语句维度不同的情况,在对集合进行最高阶重用项数计算时,需将其乘以一个权重,该权重为当前集合的最大调度维度。

为了保证合并能够获得比较有效的数据重用,并且当合并造成了数据重用损失时,新获得的数据重用能够弥补这些损失。当尝试对多个强连通分量集合进行合并时,如果该合并并没有影响到必要的并行性与分块性,并且已经计算出完备的调度,则分别计算合并之前若干个待合并集合在原有调度下的最高阶数据重用项数 R_i ,并在合并后计算新的集合在新的调度下的最高阶数据重用项数 R_{merge} 。当合并满足 $R_{merge} > \sum R_i$ 时,则判定这一合并能够带来局部性提升。

如图 2 所示的代码段,语句 S1,S2 合并后的调度为 $S_1(i,j) \rightarrow (j,i)$ 和 $S_2(i,j) \rightarrow (i,j)$,最外层并行性得到保留,不会对 S1 和 S2 进行分块,并且获得了对 $A[j][i]$ 的数据重用。然而,为了得到保留程序语义的合并结果,需要对 S1 进行循环交换。应用算法 2 可以得到 $R_{merge}^{1,2}=4,R_1+R_2=6$ 。在这一过程中,尽管获得了迭代内的数据重用,但失去了 S1 内对数组 B 访问的迭代间数据重用,造成最高阶重用项数减少。虽然引起合并的依赖边在合并后的依赖距离为 0,但这一合并并不一定会带来局部性收益。

```
for(i=0;i<N;++i){
  for(j=0;j<N;++j)
    S1 A[j][i]=A[j][i]+
      B[j][i]+B[j][i-1];
}
for(i=0;i<N;++i){
  for(j=0;j<N;++j)
    S2 C[i]=A[i][j];
}
```

➡

```
for(i=0;i<N;++i){
  for(j=0;j<N;++j){
    S1 A[j][i]=A[j][i]+
      B[j][i]+B[j][i-1];
    S2 C[i]=C[i]+A[j][i];
  }
}
```

图 2 造成数据重用项数减少的循环合并

Fig. 2 Fusion that results in a reduction in reused items

6 实验结果与分析

6.1 实验环境

本文基于 LLVM 编译框架的多面体编译模块 Polly 实现了这一合并策略,相关工具链版本为:Clang/LLVM10,Flang/LLVM9,ICC 19.1,ISL-0.20。使用的测试平台为 Intel Xeon 处理器(E5-2682 v4 2.50GHz),16 核 32 线程。每个核心具有 32 kB 的 L1d Cache,32 kB 的 L1i Cache,256 kB 的 L2 Cache。16 个核心共享 40 MB 的 L3 Cache,内存大小为 128GB。以 ICC 编译器在 O2 以及 O3 两个优化等级中性能最优的测试结果作为性能对比基准。多核性能测试时使用 16 个线程。实验对比的循环合并策略如下。

min:为每个强连通分量单独计算调度,不进行任何合并。

max:对每个弱连通分量统一计算调度,调度计算无法得到有效解时尝试找到影响求解的依赖边并进行分割,类似于 P_{LuTo} 中的 maxfuse。

inc:增量调度方法,单独计算每一个强连通分量的调度,并对依赖图进行增量的合并调度。其中,根据合并限制的选取,包括以下 4 种合并策略:1) band,强连通分量集合合并时对调度带内成员数量进行限制;2) coincidence,只对调度带内并行成员数目进行限制;3) band_coincidence,对成员数目以及并行成员数目均进行限制;4) none,不对合并前后的调度带中调度带成员数目与并行成员数目进行限制。

reuse_analyze_based:本文提出的合并策略,基于数据重用分析的增量调度方法。

测试中,LLVM 使用的编译优化选项为-O3-polly,并打开了循环分块选项。多核执行时打开了 Polly 模块中的自动 OpenMP 并行化选项以及 ICC 中的-parallel 选项。为了保证实验结果的准确性,关闭了 Polly 模块中针对矩阵乘相关内核的模式识别优化。对于 fortran 编写的测试用例,为了能够充分发挥多面体模型的分析能力,需要对编译得到的 LLVM IR 进行数组的逆线性化处理^[24]。

6.2 测试用例

受限于多面体模型中静态控制块的检测与分析能力,测试用例选用了部分具有较多循环嵌套且控制流较为简单的大型测试用例,如 spec2000 中的 swim 等,以及 Polybench 中的部分小型计算内核。同时,测试用例在多种合并策略下具有较明显的差异。选用的测试用例如表 1 所列。

表 1 实验选用测试用例及相关描述

Table 1 Test benchmarks and related descriptions

基准测试	所属测试套件	类别	问题规模
swimm	Spec2000	浅水模拟	Reference
applu	Spec2000	计算流体力学	Reference
hmmer	Spec2006	基因序列搜索	Reference
advect3d	PLuTo	气象模型	$nx=ny=nz=300$
dct	PLuTo	离散余弦变换	$M=1\ 024$
3mm	Polybench	三矩阵乘	$NI=800,NJ=900,NK=1\ 000,NL=1\ 100,NM=1\ 200$
adi	Polybench	交替方向隐式求解器	$N=1\ 000$
covariance	Polybench	协方差计算	$M=1\ 200,N=1\ 400$
doitgen	Polybench	多分辨率内核分析	$NQ=140,NR=150,NP=150$
gemver	Polybench	向量乘法与矩阵加法	$N=1\ 500$
lu	Polybench	LU 分解	$N=1\ 500$

6.3 结果分析

通过使用不同的合并限制,本文提出的合并策略在不破坏程序中依赖关系的情况下对语句执行实例重排序。即在调度过程中只对合法变换空间中的合并情形进行决策。实验结果验证了本文方法在上述测试用例中的正确性,且没有精度损失。

6.3.1 单核测试

单核执行情况下,使用各种策略的测试用例性能测试结果如图 3 所示。其中 GM 表示各合并策略所测性能的几何平均值。

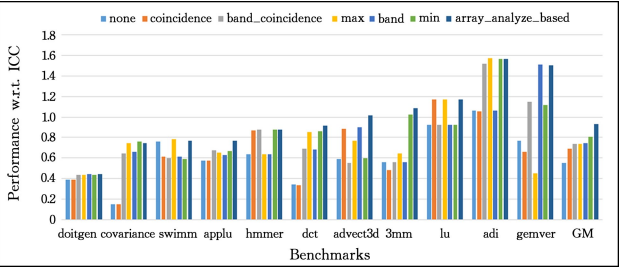


图 3 单核执行下多种合并策略在测试集上的测试结果

Fig. 3 Single core results on test sets of different fusion strategies

在多数测试用例中,ICC 获得的性能较好。由 ICC 输出的编译优化报告可知,对于 gemver,虽然 ICC 能够分析出语句间的依赖关系并给出优化指导意见,但没有对其进行优化。对于 adi,由于静态控制块内依赖较复杂,ICC 没有进行合适的循环变换以发掘向量化机会,并且没有对最内层语句进行分块。而基于多面体模型的编译方法能够准确分析该依赖关系,内核中的每个语句都可以进行向量化,并且对最内层语句进行了分块。

串行执行时,较优的循环合并策略应能保证内层语句的可分块性,并且保留向量化机会。max 策略首先计算每一个弱连通分量的调度,当无法继续求解调度时直接对弱连通分量中的每个强连通分量单独计算调度。这种方式忽略了调度长度限制以及并行性限制,导致合并后无法对关键的语句进行分块及向量化。例如,gemver 内核为包含 4 个语句的循环嵌套,循环最高维度为 2。当使用 max 策略时,直接尝试对

S1—S4 组成的弱连通分量进行调度计算。由于依赖关系的限制,此时最多只能求解出一维调度,且该维度没有并行性,即实质上将 S1—S4 在最外层进行合并,在最内层分离。同时 S2 失去了原本具有的向量化机会,S1,S2,S4 失去了缓存分块的机会。而在 hmmer 的核心计算函数 P7Viterbi 中,当使用 max 策略或不带有并行性限制的合并策略时,合并结果会影响到关键运算语句的向量化,使得性能差异十分明显。

由于没有任何调度长度的限制,none 以及 coincidence 这两种合并策略的性能较差。其中,coincidence 保留了并行性限制,在 hmmer,advect3d 等测试用例的内层循环中保留了部分向量化机会,获得了相对较好的性能表现。例如,对于 swimm 等程序中的部分循环嵌套,虽然 max,none 等策略的合并造成并行调度成员减少,但并不会损失分块性与向量化机会,在串行执行时取得了相对更好的性能。

min 策略对每一个强连通分量单独计算调度而不进行任何合并,在大多数测试用例中都取得了较优的性能表现。然而,对于 swimm,applu,advect3d 等程序,min 策略在所有循环层都不进行任何循环合并,使循环嵌套失去了较多数据重用的机会,导致在这些测试用例中,其没有获得最优的性能。

本文提出的合并策略在保证分块性与向量化机会的前提下,尽量对语句进行合并以提高数据局部性。相较于 min 策略,在 swimm,advect3d,adi 中本文的合并策略进行了合适的循环合并且没有影响到向量化与缓存分块。例如,图 4 所示的 advect3d 代码段中,计算内核语句中包含大量的数组访问时,min 合并策略虽然保证了语句的分块性与向量化机会,但损失了部分合并带来的局部性收益;band 合并策略保留了调度长度限制,但合并时造成了最内层语句向量化机会消失;coincidence 合并策略受并行成员数目限制,没有进行合适的循环合并。

表 2 列出了 advect3d 程序在不同合并策略下的循环分块语句、向量化语句、依赖边合并数目以及使用 PAPI 工具分析得到的各级缓存 cache miss 数目。

表 2 advect3d 程序在不同合并策略下的依赖边合并、循环分块、向量化与 cache miss 数目

Table 2 Merge edges,tiling,vectorization numbers and cache misses of advect3d under different fusion strategies						
合并策略	分块语句	向量化语句	合并依赖边	L1D cache miss ($\times 10^7$)	L2 cache miss ($\times 10^7$)	L3 cache miss ($\times 10^6$)
min	5	0	0	10.76	6.25	21.18
max	3	0	—	11.01	6.09	8.29
band	1	0	6	8.20	5.42	9.78
none	1	0	7	9.19	7.25	22.84
band_coincidence	0	1	4	10.92	6.30	19.83
coincidence	2	0	4	10.84	6.25	23.07
array_analyze_based	4	1	6	11.13	6.01	8.82

本文提出的合并策略没有对 advect3d 内核中内部的两层循环进行合并,以保证内层语句的分块性与并行性;同时将 advect3d 中的部分语句在最外层合并,以尽量提升数据局部

性。由表 2 可知,本文方法在保证最内层循环向量化的基础上对 6 条依赖边进行了连通分量的合并,同时 4 个多面体语句均可进行多层的循环分块,取得了较好的缓存利用率。

```
for (j = 4; j <= my+9-2; j++)
  for (i = 4; i <= nx+9-3; i++)
    for (k = 4; k <= nz+9-3; k++)
S1   ab[j][i][k] = (0.2 * (a[j-1][i][k] + a[j][i-1][k]) + 0.5 *
      (a[j-2][i][k] + a[j+1][i-1][k]) + 0.3 * (a[j-3][i][k] +
      a[j+2][i][k])) * 0.3 * xyb[j][i][k];

for (j = 4; j <= my+9-3; j++)
  for (i = 4; i <= nx+9-2; i++)
    for (k = 4; k <= nz+9-3; k++)
S2   a[j][i][k] = (0.2 * (a[j][i-1][k] + a[j][i][k]) + 0.5 *
      (a[j][i-2][k] + a[j][i+1][k]) + 0.3 * (a[j][i-3][k] +
      a[j][i+2][k])) * 0.3 * ux[j][i][k];

for (j = 4; j <= my+9-3; j++)
  for (i = 4; i <= nx+9-3; i++)
    for (k = 4; k <= nz+9-2; k++)
S3   af[j][i][k] = (0.2 * (a[j][i][k-1] + a[j][i][k]) + 0.5 *
      (a[j][i][k-2] + a[j][i][k+1]) + 0.3 * (a[j][i][k-3] +
      a[j][i][k+2])) * 0.3 * uz[j][i][k];

for (j = 4; j <= my+9-3; j++)
  for (i = 4; i <= nx+9-3; i++)
    for (k = 4; k <= nz+9-3; k++)
S4   athirda[j][i][k] = a[j][i][k] + (a[j][i+1][k] -
      a[j][i][k]) + (ab[j+1][i][k] - ab[j][i][k]) +
      (af[j][i][k+1] - af[j][i][k]);
```

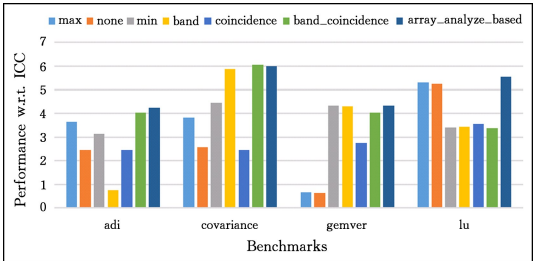
图 4 advect3d 中核心计算部分代码

Fig. 4 Core computing code fragment in advect3d

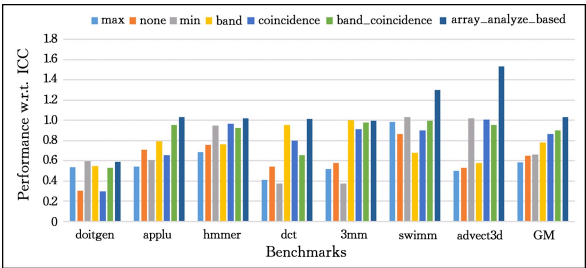
整体上,本文提出的合并策略在单核执行的情况下,使用最内层语句的并行性限制以及面向分块性的调度长度限制,在多数测试用例中都选取了较优的语句合并。相比其他不同的合并策略平均获得了 14.9%~62.5%的性能提升。

6.3.2 多核测试

多核情况下,使用 LLVM/Polly 中的自动 OpenMP 代码生成选项进行多线程并行化。不同循环合并策略的多核性能测试结果如图 5 所示。



(a) 相比 ICC 性能加速较大的测试用例实验结果



(b) 相比 ICC 性能加速较小的测试用例实验结果

图 5 多核执行下多种合并策略在测试集上的测试结果

Fig. 5 Multicore results on test sets for different fusion strategies

对于 adi, covariance, gemver 等 Polybench 中的测试用例,ICC 没有进行合适的循环变换以发掘外层并行性。在这类需要进行适当变换后进行局部性及并行性挖掘的测试用例中,多面体编译优化方法取得了较好的并行化效果与语句合并。例如,在 covariance 中,对于若干最内层语句,由于语句自身存在的输出依赖,ICC 没有成功对其进行外层的并行化。

对于 gemver 中的 S1, S2, 直接进行合并会破坏必要的依赖关系,需要对 S1 进行循环交换,接着进行合并。ICC 没有尝试将两语句进行合并,而是对 S1 进行并行化后,在编译优化报告中给出了对 S2 进行循环交换的优化意见,而没有进行并行化。

基于多面体的编译优化方法中,max 以及 none 策略得到的性能较差,在 applu, advect3d, covariance 等测试用例中,没有并行性限制的合并策略会造成核心语句最外层并行性损失,这在并行化能够带来较大收益的测试用例中较为明显。以 covariance 为例,表 3 列出了其在不同合并策略下得到的内外层并行循环数目。

表 3 covariance 程序在不同合并策略下的并行循环数目

Table 3 Number of parallel loops for covariance under different

fusion strategies			
合并策略	外层并行	内层并行	循环总数
min	9	7	42
max	1	3	9
band	3	2	21
none	2	1	6
band_coincidence	5	3	28
coincidence	5	3	28
array_analyze_based	5	3	28

由表 3 可知,尽管 min 策略取得了最多的并行循环数目,但这是其为每一个语句均生成多层循环嵌套而导致的。在性能测试中,多个一维循环的单独并行化造成了较大的线程启动开销,并且无法充分利用程序中存在的数据局部性,导致性能表现较差。同时,尽管 band 以及 none 策略对语句进行了适当的合并,但部分核心语句的外层并行性有所损失。

在目前已有的合并策略中,band_coincidence 取得了最优的性能表现。该策略包含的两种限制在一定程度上保证了外层并行性与内层语句的缓存分块性不会因合并而消失。对于 adi,当打开调度长度限制时,合并保证了最内层语句的分块性。表 4 对比了 adi 程序在不同合并方法下的外层并行化数目、分块情况、cache miss 等指标。本文提出的基于数据重用分析模型的合并策略根据强连通分量集合内的最高阶数组访问模式来决策是否保证最高维语句的分块性,并将阈值设置为 4。即当待合并集合内的最高阶数组访问模式小于或等于 4 时,不会强制要求保证其最高维语句的分块性。由于 adi 语句中数组访问模式较多,本文策略在进行合并时保证了其中对应语句的分块性。同时,对于具有较复杂数组访问的语句,本文提出的合并策略避免了对其进行合并时会造成缓存压力过大的问题。因此,本文策略在 adi 的若干核心语句中取得了与 band_coin 一致的合并结果,得到了较优的缓存利用率,如表 4 所列。

然而,band_coin 策略在 lu 中没有得到最优的语句合并。表 5 对比了 lu 程序在不同合并方法下的外层并行化数目、分块情况、cache miss 等指标。在 lu 中,当使用 band 以及 band_coin 策略时,任意合并都需要满足调度带长度限制,为了保证各语句可以进行分块,lu 中的语句均未进行合并。这两种策略下的访存开销比其余方法大,如表 5 所列。

表 4 adi 程序在不同合并策略下的依赖边合并、循环分块、外层并行循环与 cache miss 数目

Table 4 Merge edges,tiling,outer parallel loop numbers and cache misses of adi under different fusion strategies

合并策略	分块语句	合并依赖边	外层并行循环	L1D cache miss ($\times 10^7$)	L2 cache miss ($\times 10^7$)	L3 cache miss ($\times 10^7$)
min	12	0	14	5.06	4.33	1.91
max	4	—	14	5.20	4.45	2.00
band	2	10	2	101.90	40.63	15.28
none	0	17	4	8.58	9.56	2.90
band_coincidence	12	8	6	4.99	4.23	1.84
coincidence	4	10	2	8.26	9.26	2.79
array_analyze	12	8	7	4.72	3.99	1.88

表 5 lu 程序在不同合并策略下的依赖边合并、循环分块、外层并行循环与 cache miss 数目

Table 5 Merge edges,tiling,outer parallel loop numbers and cache misses of lu under different fusion strategies

合并策略	分块语句	合并依赖边	外层并行循环	L1D cache miss ($\times 10^7$)	L2 cache miss ($\times 10^7$)	L3 cache miss ($\times 10^6$)
min	2	0	3	2.13	2.11	9.36
max	0	—	1	1.93	1.98	8.28
band	2	0	3	2.12	2.11	9.34
none	0	3	1	1.72	1.78	8.00
band_coincidence	2	0	3	2.04	2.02	9.94
coincidence	1	2	3	2.12	2.11	8.41
array_analyze	1	2	3	1.82	1.89	8.12

由于 lu 中待合并的连通分量集群进行数据重用分析后得到的数组访问模式数目较小,本文提出的合并策略优先保证了依赖边造成的数据重用,不对该合并过程进行分块性限制。相比不加任何合并限制并取得最优访存开销的 none 策略,本文策略在保证了一定的缓存利用率的同时也保留了外层循环的并行性。如图 5 所示,在对 lu 中语句进行合并后,该测试用例获得了约 63% 的性能提升。

对于图 4 所示的 advect3d 程序,在多核执行情况下,max,band,none 3 种合并策略将 S1—S4 在最外层进行合并,导致最外层的并行性消失。min,band_coincidence 等策略没有对任何语句进行合并,放弃了部分数据重用机会。本文提出的合并策略将语句 S1 和 S4 在最外层进行合并,合并后仍保证了两语句内层的分块性,且所在最外层循环能进行并行化。此时语句 S1 与 S4 中对数组 a,ab 的访问获得了迭代内的数据重用,同时没有造成迭代间重用的消失。如图 5 所示,相比不进行任何合并,语句 S1 与 S4 的最外层循环合并使得该测试用例获得了约 1.5 倍的性能提升。

由于大多循环合并并不会改变语句调度,合并往往能带来最高阶数据重用项数的提升。对于部分测试用例,如 gemver,adi,applu 等,部分语句之间合并前后的最高阶数据重用项数保持不变。例如,对于图 1 所示的 gemver 程序,由于 S1,S2 合并后并没有导致最高阶数据重用项的增加,本文提出的合并策略拒绝了这一合并,获得了与 min 策略一致的合并结果。gemver 程序的实验结果表明,在多核执行情况下,不进行合并获得了约 7% 的性能提升。另一方面,在 adi,applu 等测试用例中,这一策略也避免了非最高阶数据依赖间造成的合并,减少了最内层循环内的控制流开销。

如图 5 所示,在多核执行条件下,本文提出的合并策略相比其余合并策略平均获得了 19.7%~94.9% 的性能提升。在大多数测试用例中,该策略取得了与最优合并策略一致的合并结果,而在 adi,advect3d,applu 等测试用例中相较于原

有最优的合并策略获得了 4.5%~49.6% 的性能提升。

结束语 针对不同程序,现有多面体编译工具的循环合并策略往往需要手工进行编译选项调整以获得最优的循环合并结果。本文提出了一种基于数据重用分析的合并策略,通过使用面向不同层次并行性以及缓存分块的循环合并限制,在不影响必要的并行性与分块性的前提下,尽量合并语句以提升数据局部性。同时,本文提出的合并策略对强连通分量集合内部存在的数据重用进行了建模,并将其用于合并限制以及合并收益评估,在多种大型测试用例以及计算内核中均取得了较优的合并效果。

相较于现有多种不同的合并策略,本文提出的策略在单核性能方面获得了 14.9%~62.5% 的性能提升,在多核性能方面获得了 19.7%~94.9% 的性能提升。实验结果证明,本文提出的策略能够在不影响必要优化的前提下有效地进行循环合并;在数组访问关系较为复杂的静态控制块中能够更准确地分析合并收益,相较于以往多面体模型中的循环合并策略具有更高的灵活性。

循环合并是进行程序性能优化的重要手段,尤其是对于目前应用较为广泛的图像处理、深度学习等应用领域,不同合并结果产生的性能差异较大。虽然多面体编译模型将循环合并与其他循环变换方法结合在一起进行考虑,但仍存在一些不足。首先,部分多面体编译工具依赖于手工使用编译制导语句对程序中的静态控制块进行标记。面向大型复杂应用程序,自动静态控制块检测仍然是影响循环合并等相关优化的重点问题。其次,对于语句较多的大型静态控制块,循环合并的可行解空间较大,如何减小算法复杂度并避免启发式方法获得局部最优解还需进一步的研究。最后,面向不同目标平台与体系结构,最佳的循环合并策略不尽相同,需要在合并过程中考虑到目标平台的特征信息以对合并决策进行指导。

致谢 感谢数学工程与先进计算国家重点实验室(郑州)的赵捷老师对本文研究内容与写作方面的指导与建议。

参 考 文 献

[1] RAGAN-KELLEY J, ADAMS A, SHARLET D, et al. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing [J]. Communications of the ACM, 2018, 61(1):106-115.

[2] CHEN T Q, MOREAU T. TVM: an automated end-to-end optimizing compiler for deep learning [C] // Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, 2018:579-594.

[3] ZHAO J, LI Y Y, ZHAO R C. Black Magic of Polyhedral Compilation [J]. Journal of Software, 2018, 29(8):2371-2396.

[4] BONDHUGULA U, HARTONO A, RAMANUJAM J, et al. A practical automatic polyhedral parallelizer and locality optimizer [J]. ACM Sigplan Notices, 2008, 43(6):101-113.

[5] VERDOOLAEGE S, CARLOS JUEGA J, COHEN A, et al. Polyhedral parallel code generation for CUDA [J]. ACM Transactions on Architecture & Code Optimization, 2013, 9(4):1-23.

[6] POP S, COHEN A, CÉDRIC B, et al. GRAPHITE: Polyhedral analyses and optimizations for GCC [C] // Proceedings of the 2006 GCC Developers Summit, 2006:179-197.

[7] POUCHET L N, GRLINGER A, ANDREAS S, et al. Polly-pol: polyhedral optimization in LLVM [C] // International Workshop on Polyhedral Compilation Techniques (IMPACT), 2011.

[8] VASILACHE N, ZINENKO O, THEODORIDIS T, et al. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically [J]. ACM Transactions on Architecture and Code Optimization, 2019, 16(4):1-26.

[9] BAGHDADI R, RAY J, ROMDHANE M B, et al. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code [C] // Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Press, 2019:193-205.

[10] LATNER C, AMINI M, BONDHUGULA U, et al. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation [C] // 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). ACM, 2021.

[11] VERDOOLAEGE S. isl: An Integer Set Library for the Polyhedral Model [C] // Proceedings of the Third International Congress Conference on Mathematical Software. Springer-Verlag, 2010:299-302.

[12] DARTE A. Huard. Loop shifting for loop parallelization [R]. Technical Report RR2000-22; ENS Lyon, 2000.

[13] QASEM A, KENNEDY K. Profitable loop fusion and tiling using model-driven empirical search [C] // Proceedings of the 20th Annual International Conference on Supercomputing. Association for Computing Machinery, 2006:249-258.

[14] FEAUTRIER P. Some efficient solutions to the affine scheduling problem Part II Multidimensional time [J]. International Journal of Parallel Programming, 1992, 21(6):389-420.

[15] LIM A W, LAM M S. Maximizing parallelism and minimizing synchronization with affine transforms [C] // Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for Computing Machinery, 1997:201-214.

[16] LIM A W, CHEONG G I, LAM M S. An affine partitioning algorithm to maximize parallelism and minimize communication [C] // Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing. ACM Press, 1999:228-237.

[17] BONDHUGULA U, BASKARAN M, KRISHNAMOORTHY S, et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model [C] // Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction. Springer-Verlag, 2008:132-146.

[18] BONDHUGULA U, GÜNLÜK O, DASH S, et al. A model for fusion and code motion in an automatic parallelizing compiler [C] // Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. Association for Computing Machinery, 2010:343-352.

[19] LOUIS-NOËL P, BONDHUGULA U, CÉDRIC B, et al. Loop Transformations: Convexity, Pruning and Optimization [J]. ACM Sigplan Notices, 2011, 46(1):549-562.

[20] LOUIS-NOËL P, BONDHUGULA U, CÉDRIC B, et al. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework [C] // Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010:1-11.

[21] MEHTA S, LIN P H, YEW P C. Revisiting loop fusion in the polyhedral framework [C] // Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Association for Computing Machinery, 2014:233-246.

[22] ZINENKO O, VERDOOLAEGE S, REDDY C, et al. Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling [C] // Proceedings of the 27th International Conference on Compiler Construction. Association for Computing Machinery, 2018:3-13.

[23] VERDOOLAEGE S, GUELTON S, GROSSER T, et al. Schedule trees [J]. Geodimica Acta, 2013, 25(1/2):86-95.

[24] TOBIAS G, RAMANUJAM J, LOUIS N, et al. Optimistic de-linearization of parametrically sized arrays [C] // Proceedings of the 29th ACM International Conference on Supercomputing. Association for Computing Machinery, 2015:351-360.



HU Wei-fang, born in 1995, master, is a student member of China Computer Federation. His main research interests include advanced compiler technology and high performance computing.



SHANG Jian-dong, born in 1968, Ph.D., professor, Ph.D supervisor, is a member of China Computer Federation. His main research interests include data mining and high performance computing.