软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(12):5704−5723 [doi: 10.13328/j.cnki.jos.006757]

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

面向 Stencil 计算的自动混合精度优化*

宋广辉[1,2], 郭绍忠[1,2], 赵 捷[1,2], 陶小涵[1,2], 李 飞[1,2], 许瑾晨[1,2]

[1](信息工程大学, 河南 郑州 450001)

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

通信作者: 许瑾晨, E-mail: atao728208@126.com

摘 要: 混合精度在深度学习和精度调整与优化方面取得了许多进展, 广泛研究表明, 面向 Stencil 计算的混合精度优化也是一个很有挑战性的方向. 同时, 多面体模型在自动并行化领域取得的一系列研究成果表明, 该模型为循环嵌套提供很好的数学抽象, 可以在其基础上进行一系列的循环变换. 基于多面体编译技术设计并实现了一个面向 Stencil 计算的自动混合精度优化器, 通过在中间表示层进行迭代空间划分、数据流分析和调度树转换, 首次实现了源到源的面向 Stencil 计算的混合精度优化代码自动生成. 实验表明, 经过自动混合精度优化之后的代码, 在减少精度冗余的基础上能够充分发挥其并行潜力, 提升程序性能. 以高精度计算为基准, 在 x86 平台上最大加速比是 1.76, 几何平均加速比是 1.15; 在新一代国产申威平台上最大加速比是 1.64, 几何平均加速比是 1.20.

关键词: 自动混合精度; Stencil 计算; 多面体模型; 循环嵌套; 调度树

中图法分类号: TP18

中文引用格式: 宋广辉, 郭绍忠, 赵捷, 陶小涵, 李飞, 许瑾晨. 面向Stencil计算的自动混合精度优化. 软件学报, 2023, 34(12): 5704–5723. http://www.jos.org.cn/1000-9825/6757.htm

英文引用格式: Song GH, Guo SZ, Zhao J, Tao XH, Li F, Xu JC. Automatic Mixed Precision Optimization for Stencil Computation. Ruan Jian Xue Bao/Journal of Software, 2023, 34(12): 5704–5723 (in Chinese). http://www.jos.org.cn/1000-9825/6757.htm

Automatic Mixed Precision Optimization for Stencil Computation

SONG Guang-Hui[1,2], GUO Shao-Zhong[1,2], ZHAO Jie[1,2], TAO Xiao-Han[1,2], LI Fei[1,2], XU Jin-Chen[1,2]

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

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

Abstract: Mixed precision has made many advances in deep learning and precision tuning and optimization. Extensive research shows that mixed precision optimization for stencil computation is challenging. Moreover, the research achievements secured by the polyhedral model in the field of automatic parallelization indicate that the model provides a good mathematical abstraction for loop nesting, on the basis of which loop transformations can be performed. This study designs and implements an automatic mixed precision optimizer for Stencil computation on the basis of polyhedral compilation technology. By performing iterative domain partitioning, data flow analysis, and scheduling tree transformation on the intermediate representation layers, this study implements the source-to-source automatic generation of mixed precision codes for Stencil computation for the first time. The experiments demonstrate that the code after automatic mixed precision optimization can give full play to its parallelism potential and improve the performance of the program by reducing precision redundancy. With high-precision computing as the benchmark, the maximum speedup is 1.76, and the geometric average speedup is 1.15 on the x86 architecture; on the new-generation Sunway architecture, the maximum speedup is 1.64, and the geometric average speedup is 1.20.

Key words: automatic mixed precision; Stencil computation; polyhedral model; loop nesting; scheduling tree

---

* 基金项目: 国家自然科学基金 (U20A20226)
收稿时间: 2022-04-01; 修改时间: 2022-06-11, 2022-07-20; 采用时间: 2022-08-01; jos 在线出版时间: 2023-02-22
CNKI 网络首发时间: 2023-02-24

Stencil 计算是浮点计算应用中一种常见的循环嵌套运算模式, 也是加州大学伯克利分校并行计算实验室提出的 7 个计算主题之一[1], 广泛应用于从物理模拟[2]到机器学习的各个领域, 其计算特点是在时间步迭代下遍历计算网格并根据网格中的每个元素及其相邻元素来更新网格. 一般而言, 可以根据其维度的不同将其分为一维 (1D) Stencil、二维 (2D) Stencil 和三维 (3D) Stencil . 此外, 还可以根据其网格形状对其进行分类, 最广泛使用的是星形 (star) 和盒形 (box), 星形 Stencil 一次仅在一个维度的方向上访问与当前元素相邻的元素 (即没有对角线访问), 而盒形 Stencil 则形成完整的正方形 (对于 2D) 或立方体 (对于 3D).

Stencil 计算中使用的浮点数一般遵循 IEEE 754 标准, 其中最经常使用的浮点类型是 32 位的单精度 (float) 和 64 位的双精度 (double)[3]. 在程序中只使用 double 比只使用 float 能够得到更精确的结果, 但是只使用 double 所需的计算资源、内存存储以及消耗的功率都比 float 要多很多[3]. 然而并非每个应用都需要特别精确的结果, 因此对于这些应用可以同时使用不同的精度进行运算, 即“混合精度”. 其优势在于能够使精度保持在给定范围内的同时有效提升计算效率, 使得程序占用更少的资源, 运行速度更快[4,5]. 东京大学和橡树岭实验室利用混合精度模型计算地震波对于东京建筑系统的影响, 该模型在 Summit 上的运行速度比更高精度的模型快了 25 倍[4], 也表明混合精度相较于高精度有着更好的速度优势. 2018 年百度与 NVIDIA 联合发表论文并提出了混合精度训练[5]的方法, 该方法一经提出便在学术界和工业界引起众多的专家和学者的关注并开展了相关研究. 目前, 越来越多的研究人员开始研究混合精度这一课题, 这也是未来高性能计算的一个发展方向.

现有的混合精度的研究核心都集中在有明显精度差别的变量声明、算子等层次上, 对于程序中的循环嵌套关注度很少, 但是 Stencil 计算的性能和误差热点一般是循环嵌套中的高精度计算, 其严重拖慢程序执行时间, 而如果是将其直接转换成低精度计算, 又会使误差大幅度提高. 因此, 针对 Stencil 计算中的循环嵌套, 在满足误差阈值的前提下利用混合精度技术提升程序性能成为一个难题.

为了解决这一难题, 本文基于多面体模型设计并实现了一个面向 Stencil 计算的自动混合精度优化器 (automatic mixed precision optimizer for Stencil computations, AMPO-SC), 该优化器可以将 C 语言源程序中指定的循环嵌套区的 Stencil 计算代码自动转换成混合精度优化之后的代码. 本文的主要贡献如下.

(1) 提出了基于仿射约束表达式的迭代空间划分算法, 在多面体模型的指导下能够对任意 Stencil 计算中的循环嵌套进行划分, 给出了混合精度计算所需的相互分离的迭代空间.

(2) 将 (1) 与数据流分析和调度树转换结合在一起, 实现了一个面向 Stencil 计算的自动混合精度优化器, 即首次实现了源到源的面向 Stencil 计算的混合精度优化代码自动生成.

(3) 选取了 Stencil 计算 10 个典型的测试用例进行实验, 其结果表明, 本文所实现的面向 Stencil 计算的自动混合精度优化器能够自动生成混合精度优化之后的代码, 且可以充分挖掘 Stencil 计算中潜在的精度冗余和性能增长.

本文第 1 节介绍混合精度的研究现状. 第 2 节介绍本文研究的基础知识, 包括多面体模型和调度树. 第 3 节介绍面向 Stencil 计算的自动混合精度优化器的框架. 第 4 节详细介绍面向 Stencil 计算的自动混合精度优化器的核心算法和实现细节. 第 5 节分析本文所开展的实验及其结果. 第 6 节总结全文.

1 研究现状

混合精度是一种在存在精度冗余的程序中同时使用高精度计算和低精度计算的程序优化手段. 目前混合精度的研究主要分为两种, 一种是算子级的混合精度, 典型的就是人工智能领域的混合精度训练, 另一种是变量级的混合精度, 典型的就是各种精度调整和分析工具, 根据其特点又可以分为静态分析工具和动态分析工具. 他们的核心都是通过一定的策略利用不同精度的变量和算子来实现混合精度优化, 在一定的误差阈值下尽可能地提升应用性能.

1.1 混合精度训练

2018 年, 百度与 NVIDIA 联合发表的论文[5]中提出了混合精度训练的方法. 混合精度训练是指在训练过程中, 同时使用单精度 (FP32) 和半精度 (FP16)[5], 其目的是相较于只使用 FP32 训练模型, 在保持精度持平的条件下, 能够加速训练. 混合精度训练的内存占用更少, 计算速度更快的优势很快就引起了学术界和工业界的极大重视, 目前

主流的人工智能开发框架 TensorFlow、PyTorch、OneFlow、PaddlePaddle 和 MindSpore 等都陆续支持了自动混合精度训练.

人工智能应用的计算过程可以分为建模、训练和推理这 3 个过程, 其中建模是根据应用的特征建立合适的模型, 如果是语音识别、手写识别和文字识别之类的需要处理和预测时间序列中间隔和延迟非常长的重要事件的应用, 一般需要建立长短期记忆人工神经网络 (long short-term memory, LSTM), 而如果是图像识别和人脸识别之类的需要具有表征学习 (representation learning) 能力并能够按其阶层结构对输入信息进行平移不变分类的应用, 一般需要建立卷积神经网络 (convolutional neural network, CNN); 其次, 是用海量的数据信息来训练模型, 通过大量的训练来提升模型捕获输入数据中特征的能力, 具体而言就是在每次训练之后, 获取模拟的结果与真实结果之间的损失 (loss), 然后通过一定的手段和策略调整模型中的参数再次训练, 如此循环往复, 直到损失满足实际应用需求为止; 最后, 便于用训练好的模型去求解现实中的特定问题, 以推理得到较好的计算结果, 这一过程与人类的思维过程十分类似, 因此称为推理.

因为人工神经网络的参数和中间结果绝大部分都是采用 FP32 进行存储和计算的, 当网络变得超级大时, 这部分的计算和访存将会成为程序的性能瓶颈, 大大降低程序的执行速度. 因此在对精度不太敏感的训练过程中降低浮点数精度, 比如使用半精度浮点数, 显然是提高计算速度, 降低存储开销的一个很直接的办法, 这便是混合精度训练的核心思想. 然而副作用也很显然, 如果直接降低浮点数的精度直观上必然导致模型训练精度的损失, 因此为了解决精度损失的问题, 混合精度训练还提出了权重备份和 loss 缩放等方法来规避这一问题. 同时为了提升混合精度训练在人工智能应用中的广度和深度, 目前主流的人工智能开发框架都实现了对自动混合精度训练 (也被称为自动混合精度, automatic mixed precision, AMP) 的支持.

1.2 精度分析和调整工具

浮点研究人员已经开发了很多精度分析和调整工具, 以便来测试、分析和改进他们的代码[6]. 根据精度调整方式的不同可将其分为手动调整精度和自动调整精度 (也称为自动精度调整) 两种[6], 程序开发人员手动调整精度的局限性较大, 基本只能应用于程序较小的时候, 而自动精度调整则不会受程序大小的限制, 但是其实现难度较大. 自动精度调整方法是目前精度调整研究的热门领域, 一般分为动态分析方法 (技术) 和静态分析方法 (技术) 两种. 米兰理工大学的 Cherubin 等人发表的一项调查对 29 个精度调整工具进行了详细的对比和分析, 其具体内容可以参考文献 [6].

静态分析技术直接从源代码中提取相关信息并通过相关的理论计算来预测误差的最坏情况[6], 而无需使用输入数据对程序进行测试. 这种方法虽然可以保证计算的精确度, 但是这种过于保守的方法可能会导致极大的误差范围[6], 因此其通常会混合使用几种技术来实现更高的精度. Damouche 等人提出的基于抽象表示的静态分析方法的 Salsa[7]是一个能够提升浮点计算精度的源到源的自动化工具, 其通过对原始程序应用一组变换来自动生成比初始程序计算更精确的优化程序. 其核心是基于一定规则的重写方法来对源代码进行转换以提高浮点计算的准确性, 比如它为 C 语言提供了过程内和过程间的重写优化. Cherubin 等人提出的基于 LLVM 编译器框架的 TAFFO[8]是一个用于浮点到定点优化的精度调整工具, 它提供了一种权衡计算精度以提高运行速度的解决方案, 并作为一个支持 C 和 C++程序的编译器扩展插件以供用户使用. 但是 Salsa 和 TAFFO 都要求用户对源代码中的数值变量的范围进行注释, 更准确地说, 他们本质上都是基于区间算术进行值域分析[6], 这也正是它们无法实现循环嵌套内的精度和性能的权衡的根本原因. 其他的基于静态分析方法的工具也都未能解决循环嵌套内的精度和性能的权衡, 比如: Darulova 等人提出的利用遗传编程根据重写规则能够生成等价表达式的 Xfp 工具[9]、Kotipalli 等人提出的通过分析依赖图中具有强关联关系的浮点变量组合以减少精度转换开销的 AMPT-GA 工具[10]以及 Chiang 等人提出的基于泰勒展开在给定的误差范围内能够针对表达式生成混合精度配置方案的 FPTuner[11]工具等的静态分析工具.

由于静态分析的局限性, 往往得出的精度分配方案比较保守, 不能有效地找到最佳方案, 故动态分析方法是更为常用的技术. 动态分析技术一般是在具有代表性的输入集上运行降低精度的版本和原始的较为精确的版本, 并比较两者的结果进而得到精度和性能较为权衡的精度分配方案. Kum 等人提出的 Autoscaler For C[12]是一个源到

源的用于将数字信号处理的 C 语言程序中的浮点数转换成定点整数的转换器. 它不仅可以转换数据类型并支持自动缩放, 还可以进行移位优化以提高执行速度[12]. 同时, 它还制定了表示缩放开销的成本函数, 并使用整数线性规划或模拟退火算法来最小化这个成本函数[12], 从而达到进一步优化程序的目的. Menon 等人提出的 ADAPT[13]是一个使用自动微分算法来分析浮点变量和计算操作的精度敏感度, 并利用贪心算法来降低变量和操作的精度用以指导程序员开发混合精度应用程序的工具, 其核心是通过微分来估计每个浮点变量的精度敏感度, 并将所有敏感度组合成一个误差模型[13], 用于估计精度降低对给定变量的影响, 然后从对结果影响最小的变量开始, 迭代地将变量的精度降低, 直到误差达到阈值时停止. 此外, 还有 Nathan 等人提出的以低精度程序为输入利用启发式方法逐渐提高精度以获得给定误差阈值下的混合精度分配方案的 AMP[14]工具、Rubio-González 等人提出的利用 LLVM 框架来调整变量声明以构建混合精度配置方案的 Precimonious[15]工具、在 Precimonious 基础上通过影子执行找到可以忽略的浮点变量进而减少搜索空间的 Blame Analysis[16]工具、在 Precimonious 基础上通过对程序进行依赖分析进而减少搜索空间的 HiFPTuner[17]工具等的动态分析工具. 但是, 这些工具无论是直接修改变量声明还是在中间表示层修改变量类型, 亦或是将浮点数转换成定点整数, 都不能解决 Stencil 计算的精度和性能相权衡的难题.

综上所述, 混合精度计算的相关工作都取得了很好的进展, 但是都不能解决具有循环嵌套运算模式的 Stencil 计算的精度和性能的权衡难题, 一方面是因为循环本身是没有明确的精度类型—循环的精度取决于循环中语句的计算精度和循环的迭代次数, 并且由于编译器可能会对精度进行自动提升, 语句实际的计算精度往往是隐含的. 另一方面, 在一个循环结构中, 往往又会包含另一个完整的循环结构 (即循环嵌套), 因此, 循环嵌套中的计算顺序具有一定的复杂性, 外层循环体每执行一次, 内层循环都要整体循环一次. 再加上循环的嵌套方式和层数十分丰富多样, 这些都大大增加了对 Stencil 计算中的循环嵌套进行混合精度的复杂性. 因此本文拟基于混合精度的基本思想, 对用户程序中的 Stencil 计算开展自动混合精度优化工作.

2 基础知识

本文的工作主要基于多面体模型和调度树, 下面就相关概念和基本知识予以介绍.

2.1 多面体模型

多面体模型 (polyhedral model)[18]是一种用于分析和转换程序的高度抽象化的编译优化模型, 与其他模型相比, 它最大的特点是通过紧凑表示法单独处理每个语句实例 (即循环嵌套内语句的每次动态执行) 和每个数组元素[19], 具有应用范围广、表示能力强、优化范围大等优点[20]. 多面体模型主要由迭代空间 (iteration domains)、访存映射 (write and read)、依赖关系 (dependences) 和调度 (schedule) 这 4 个部分组成[20].

多面体编译工具的一般编译流程如图 1 所示, 其一般由抽象分析、调度变换和代码生成 3 个阶段组成, 且线性整数规划始终贯穿其中[20]. 抽象分析是多面体编译工具的前端, 其作用是将输入语言中给定的程序段表示成多面体形式, 并计算其迭代空间、访存映射和依赖关系; 调度变换是多面体编译工具的中间优化部分, 其作用是通过一系列的分析和转换在不破坏程序依赖关系的前提下计算一个新的调度顺序, 从空间几何角度来看, 调度变换的过程本质上就是将程序段对应的多面体进行多维空间几何变基的过程, 因此该阶段也是最复杂的一个阶段; 代码生成是多面体编译工具的后端, 其作用是将多面体模型的中间表示依据目标体系结构的编程模型生成对应语言规范的程序代码.

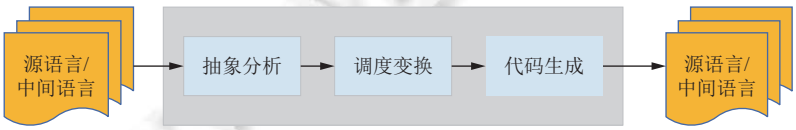

图 1 基于多面体模型的一般编译流程

在程序自动并行化领域, 多面体模型不仅取得了很多突破性的成就, 还在 Pluto 编译工具[21]、PPCG 编译工具[22]、GCC 中的 Graphite 框架[23]和 LLVM 中的 Polly 模块[24]等开源工具和商业应用中被广泛使用. 故本文拟基

于多面体模型开展面向 Stencil 计算的自动混合精度优化研究.

2.2 调度树

多面体模型中的调度是一种用于表示程序执行顺序的中间表示 (intermediate representation, IR) 形式, 结合数据结构中“树”的一些特征, 我们不难发现, 多面体模型中使用的调度天然具有树的形式, 因为它特别适合处理循环和复合语句. 前人提出的通用调度表示形式, 例如 Kelly 等人提出的抽象表示形式[25]、Girbal 等人提出的“2d+1”调度表示形式[26]和 Verdoolaege 提出的 Presburger 关系表示形式[27], 本质上都是对这种调度树的编码. 尽管这些编码原则上足以作为抽象语法树 (abstract syntax tree, AST) 生成器的输入, 但它们会使中间操作 (如对调度树中的特定结点进行分块) 变得更加麻烦. 因此本文采用调度树作为多面体模型中调度的中间表示形式, 以便在树上执行各种操作, 然后将结果发送到 AST 生成器, 而不必再将调度树转换为另一种表示形式.

一般而言, 调度树中可以包含 *domain*、*sequence*、*filter*、*band*、*mark* 和 *extension* 等类型的结点. 其中 *domain* 结点是调度树的根结点; *sequence* 结点的子结点必须按先后顺序执行; *filter* 结点一般是 *sequence* 等结点的子结点, 表示当前子树中所有语句实例的集合; *band* 结点与循环嵌套相对应; *mark* 结点表示当前子树中的附加信息; *extension* 结点是一个可由程序员操作的扩展结点. 有关调度树的结点信息以及由其生成 AST 的算法和实现原理可以参考文献 [19].

基于调度树, 可以很方便地对循环嵌套进行各种优化, 一方面是因为调度树这一中间表示形式可以通过一定的手段将其描绘成“树”的形状, 十分直观且易于理解, 程序开发人员可以很方便地对其进行修改和插入操作, 从而实现特定的功能. 另一方面是因为 PPCG 等多面体编译器中代码生成模块可以根据调度树生成其对应的 AST, 进而自动生成面向特定体系结构的并行代码, 程序开发人员只需要确保调度树等信息符合特定的规范即可. 因此本文采用调度树作为多面体模型中调度的中间表示形式, 以实现面向 Stencil 计算的自动混合精度优化.

3 面向 Stencil 计算的自动混合精度优化器

为了解决混合精度对 Stencil 计算的普遍适应性难题, 针对程序中给定的 Stencil 计算自动生成混合精度程序, 本文设计了如图 2 所示的面向 Stencil 计算的自动混合精度优化器. 它以 C 语言源程序为输入, 用户指定的混合比例为参数, 通过预处理、混合精度优化和代码生成 3 个模块, 自动生成该比例下的混合精度程序. 然后可以通过测试混合精度程序的误差和性能表现情况, 进一步调整混合精度比例, 直至误差阈值和性能需求满足用户需求为止. 该优化器采用“源到源”的编译方式, 将 C 语言源程序中指定的循环嵌套区的 Stencil 计算代码自动转换成混合精度优化之后的代码, 最终可以得到在 x86 通用计算平台和新一代国产申威平台上执行的混合精度程序.

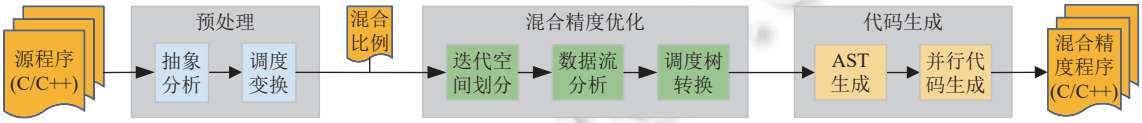

图 2 面向 Stencil 计算的自动混合精度优化器示意图

面向 Stencil 计算的自动混合精度优化器的编译优化流程如图 2 所示, 该工具由预处理、混合精度优化以及代码生成 3 个模块组成, 其中混合精度优化模块 (图 2 中绿色步骤) 是本文工作的核心.

(1) 预处理模块将从 C 语言源程序中识别用户指定的循环嵌套程序段, 将该程序段表示成多面体形式, 通过调用线性整数规划过程计算依赖关系, 然后利用 isl (integer set library) 调度算法将原始调度转换成另一种新的调度, 并生成调度树中间表示.

(2) 混合精度优化模块将根据混合比例的参数值对循环嵌套的迭代空间进行划分, 然后对划分出来的两个子空间进行数据流分析, 并通过计算得出混合精度计算中需要进行强制类型转换的相关数据, 最后依据这些信息对

调度树进行转换, 即利用树中各种结点通过插入和删除等操作将原始调度树转换成混合精度计算的调度树.

(3) 代码生成模块将调度树等数据作为输入, 借助线性整数规划过程生成 AST, 然后根据 C 语言规范将 AST 转变成最终代码, 即得到经过自动混合精度优化的目标代码.

预处理模块其实是基于多面体模型的一般编译流程中的抽象分析和调度变换两个阶段的组合, 称之为预处理是针对第 2 部分的混合精度优化而言的; 混合精度优化模块本质上也是在满足依赖关系的前提下, 将一种调度转换成另外一种调度的过程, 即其本质上也是一种调度变换, 与自动并行化领域的调度变换不同的是, 其不是以挖掘程序并行性和数据局部性为目的, 而是以实现混合精度计算优化为目的; 代码生成模块本质上则是对原有编译工具的代码生成阶段进行适当的改造和扩展以使其支持混合精度优化之后的调度树.

从理论上讲, 混合精度优化的目标代码中包含的高精度计算占比越高, 那么误差就会越小, 精度也就越高; 程序运行耗时就会越长, 性能也就越差. 然而, 实际上其精度和性能情况还会受到其他多种因素的影响, 且不同因素之间相互关联, 例如: 不同混合比例下的混合精度计算程序在运行时其 Cache 命中率、累积误差的大小、强制类型转换引起的舍入误差等都会不同, 且其均会对精度和性能产生一定的影响, 因此目前尚没有一种成熟的方法可以直接量化或者分析出来这种相互关系, 但是这个趋势是普遍成立的, 本文的实验也充分验证了这一趋势. 因此用户可以通过一定的策略生成几个特定混合比例下的混合精度代码, 并根据其误差和性能表现来进一步确定最合适的混合比例或其区间, 比如: 利用二分搜索的思路通过判断误差是否符合应用的精度需求来确定混合比例所在的区间, 再结合区间内的性能表现来确定最合适的混合比例.

4 优化器实现细节

本文的面向 Stencil 计算的自动混合精度优化器是在 PPCG[22]的基础上开发的 (开源代码仓库地址: https://gitee.com/sheenisme/ppcg.git), 其核心功能均是利用 pet 抽象分析库和 isl 线性整数规划工具来实现的. 本文以 Stencil 计算中常见的一维 Jacobi 迭代计算为例来介绍其核心算法及功能实现, 而 pet 和 isl 库的实现原理及算法可分别参考文献 [28,29].

4.1 预处理

预处理模块分为抽象分析和调度变换两个阶段, 抽象分析阶段利用 pet 库从用户输入的 C 语言源代码中识别用#pragma scop 和#pragma endscop 编译指示指定的循环嵌套程序段, 然后构建该程序段的多面体模型, 其中包括迭代空间、访存关系以及原始调度, 并以仿射约束的形式储存起来, 以便对该程序段进一步分析和优化.

迭代空间是对应的程序段中所有语句实例的集合, 且语句实例与迭代向量一一对应. 示例一维 Jacobi 迭代计算的核心代码如图 3 所示, 其迭代空间可以用公式 (1) 来表示, 其中每一个迭代向量 (t, i) 都与语句 S_1 和 S_2 的一次执行实例一一对应. 例如 $S_1(2,3)$ 表示当循环索引 t=2 且 i=3 时语句 S_1 的执行实例.

```
#pragma scop
for(t=0; t<20; t++){
  for(i=1; i<29; i++)
S1:   B[i] =alpha*(A[i−1]+A[i]+A[i+1]);
  for(i=1; i<29; i++)
S2:     A[i]=beta*(B[i−1]+B[i]+B[i+1]);
}
#pragma endscop
```

图 3 一维 Jacobi 迭代计算代码

$$domain = \{S_1(t,i) : 0 \leqslant t \leqslant 19 \wedge 0 < i \leqslant 28; S_2(t,i) : 0 \leqslant t \leqslant 19 \wedge 0 < i \leqslant 28\} \tag{1}$$

访存映射是语句实例与其访问的数组元素之间映射关系的集合, 其中包括了读访存映射关系与写访存映射关系. 对于一维 Jacobi 迭代计算的示例, 两者分别可以用公式 (2) 和公式 (3) 来表示, 其中“→”表示源点的语句实例读或者重写了汇点元素的数据, 例如 *Write* 集合中的 $S_1(t,i) \rightarrow B(i)$ 表示语句 S_1 执行到迭代向量为 (t, i) 时会向 $B(i)$ 元素写入新的数据.

$$Read=\left\{\begin{array}{l} S_1(t,i)\rightarrow A(j):\\ (j=i\wedge t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28)\\ \vee(j=i-1\wedge t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28)\\ \vee(j=i+1\wedge t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28);\\ S_2(t,i)\rightarrow B(j):\\ (j=i\wedge t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28)\\ \vee(j=i-1\wedge t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28)\\ \vee(j=i+1\wedge t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28);\\ S_1(t,i)\rightarrow alpha: t\geqslant 0\wedge t\leqslant 19\wedge \mathrm{i}>0\wedge i\leqslant 28\\ S_2(t,i)\rightarrow beta: t\geqslant 0\wedge t\leqslant 19\wedge \mathrm{i}>0\wedge i\leqslant 28 \end{array}\right\} \tag{2}$$

$$Write=\left\{\begin{array}{l} S_1(t,i)\rightarrow B(i): t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28;\\ S_2(t,i)\rightarrow A(i): t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28 \end{array}\right\} \tag{3}$$

调度主要是为语句实例分配执行顺序, 在多面体模型中一般采用通过映射关系将每一个语句实例与一个无名整数元组一一对应的形式来表示. 示例一维 Jacobi 迭代计算的原始调度可以由公式 (4) 来表示, 其中变量 t 和 i 均须满足公式 (1) 中迭代空间的约束条件. 这里之所以称其为原始调度是为了和调度变换之后的调度区分开来, 其调度树表示形式如图 4(a) 所示.

$$schedule=\{S_1(t,i)\rightarrow(t,i);S_2(t,i)\rightarrow(t,i)\} \tag{4}$$

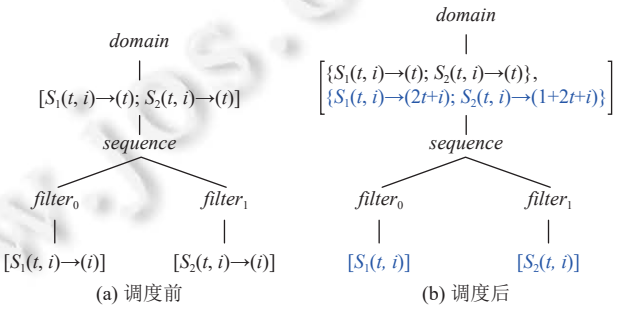

图 4 调度变换前后的调度树示意图

依赖关系是相互依赖和关联的语句实例的集合, 同时它也是判断程序是否正确执行的关键, 因为一旦程序的依赖关系发生改变, 那么程序的语义和结果也会随之而改变. 因此, 在抽象分析阶段一般还会以迭代空间和访存映射为输入, 通过调用线性整数规划过程来计算依赖关系[20]. 示例一维 Jacobi 迭代计算的依赖关系可以由公式 (5) 来表示:

$$dependence=\left\{\begin{array}{l} S_1(t,i)\rightarrow S_2(t,j): t\geqslant 1\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28\wedge j\geqslant i-1\wedge j\leqslant i+1;\\ S_2(t,i)\rightarrow S_1(t,j): t\geqslant 0\wedge t\leqslant 19\wedge i>0\wedge i\leqslant 28\wedge j\geqslant i-1\wedge j\leqslant i+1 \end{array}\right\} \tag{5}$$

在程序自动并行化领域, 多面体编译工具在将循环嵌套内的语句表示成空间多面体形式之后, 会进行一系列的以提升程序并行性和数据局部性为目的分析和变换, 即调度变换. 为了使程序具有较好的并行性和局部性, 面向 Stencil 计算的自动混合精度优化器在抽象分析阶段结束之后, 采用了 isl 库中改进的 isl 调度算法[29]来对循环嵌套进行变换. 一维 Jacobi 迭代计算示例进行调度变换之后的调度: $\{S_1(t,i)\rightarrow(t,2t+i);S_2(t,i)\rightarrow(t,2t+i+1)\}$, 其中变量 t 和 i 也均须满足公式 (1) 中迭代空间的约束条件, 其对应的调度树表示形式如图 4(b) 所示, 图中蓝色部分即为调度变换所修改的部分.

4.2 混合精度优化

经过预处理模块完成对程序段的抽象分析以及调度变换后, 优化器将得到一个包含多个 *band* 结点的调度树, 然后混合精度优化模块将针对调度树中 Stencil 计算的最外层 *band* 结点及其子结点进行迭代空间划分、数据流分析和调度树转换等操作, 从而得到一个经过混合精度优化之后的调度树. 需要说明的是, 下文中的 *band* 结点默认是调度树中 Stencil 计算的最外层 *band* 结点. 对于一维 Jacobi 迭代计算的示例而言, 从图 4 所示的调度变换后的调度树中不难看出, 树的第 2 层即为 Stencil 计算的最外层 *band* 结点.

4.2.1 迭代空间划分

迭代空间划分是指利用迭代空间划分算法将 *band* 结点对应的循环嵌套的迭代空间进行划分, 将原本的一个迭代空间划分成两个相互分离的两个迭代空间, 以便在不同的迭代空间进行不同精度的计算的过程. 由于 *band* 结点有可能包含有多层循环嵌套, 所以我们在对迭代空间划分时只对最外层的循环进行切分, 而不是每层都切分或者对内层切分, 以达到充分发挥程序并行性和数据局部性, 尽可能地提升程序性能的目的. 由于我们是以仿射约束的形式来储存迭代空间的, 所以这里采用的是基于仿射约束表达式的迭代空间划分算法, 其算法描述如算法 1.

算法 1. 基于仿射约束表达式的迭代空间划分算法.

输入: *rate*, *band* node;

输出: band_node_list.

```
initialize two new band nodes
foreach domain of band node do
    initialize two new iteration domain
    foreach statement of domain do
        initialize a new empty statement in each new iteration domain
        foreach affine constraint pair of statement do
            if index = 0 then //如果是最外层循环的仿射约束对
                get the boundary of two affine constraint expressions, x and y
                val = floor((y–x) × rate/100)
                compute the affine constraint pair: val – t ⩾ 0 and –x + t ⩾ 0
                compute the affine constraint pair: −1 − val + t ⩾ 0 and y − t ⩾ 0
                add the pair of affine constraint to the corresponding statement respectively
            else
                add the pair of affine constraint to both statement
            end
        end
    end
    add the two new iteration domain to the corresponding band node respectively
end
combine two band nodes into band_node_list
return band_node_list
```

示例一维 Jacobi 迭代计算中 *band* 结点的调度可以用公式 (6) 来表示:

$$[\{S_1(t,i)\to(t);S_2(t,i)\to(t)\},\{S_1(t,i)\to(2t+i);S_2(t,i)\to(1+2t+i)\}] \tag{6}$$

可以发现, 由于调度变换阶段对循环嵌套实现了循环合并优化, 所以优化后的两条语句在同一个循环体中, 即语句 S_1 和 S_2 在同一个迭代空间中, 且只有这一个迭代空间.

由于程序中一切合法的循环必须要同时有上下界, 以便程序能够在适当的时候跳出循环, 所以一般一层循环会有两个仿射约束表达式, 我们将其称为一对仿射约束表达式. 以语句 S_1 为例, 其实际存储的仿射约束可以用公式 (7) 来表示:

$$(\{S_1(t,i):t\geqslant 0\},\{S_1(t,i):19-t\geqslant 0\},\{S_1(t,i):-1+i\geqslant 0\},\{S_1(t,i):28-i\geqslant 0\}) \tag{7}$$

那么, 语句 S_1 最外层循环的仿射约束对可以用公式 (8) 来表示:

$$(\{S_1(t,i):t\geqslant 0\},\{S_1(t,i):19-t\geqslant 0\}) \tag{8}$$

首先从其中提取循环的上下界, 即仿射约束表达式的边界值 x 和 y, 然后根据设定的混合比例参数 *rate* 的数

值计算切分的中间值 *val*, *val* 的计算过程可以用公式 (9) 来表示:

$$val = \left\lfloor (y-x) \times \frac{rate}{100} \right\rfloor \tag{9}$$

算法 1 在设计时便充分考虑到了如何尽可能帮助用户能够快速准确地找到最合适的混合比例 *rate* 的值, 为此在算法中采用了等分的策略将最外层循环进行一百等分, 这样 *rate* 的搜索空间便被大幅度缩小, 同时 *rate* 代表了第 1 个迭代空间大小占原始迭代空间大小的百分比, 即 *rate* 的值描述了切分后的两个迭代空间大小比例是 *rate* : (100–*rate*), 其示意图如图 5 所示.

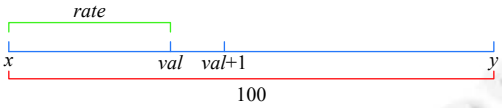

图 5 最外层循环切分示意图

然后依次计算切分后的第 1 个迭代空间循环边界对应的仿射约束表达式和切分后的第 2 个迭代空间循环边界对应的仿射约束表达式, 并将这两对仿射约束分别添加到两个迭代空间的语句中. 这两个语句最外层循环的仿射约束对的可以分别用公式 (10) 和公式 (11) 来表示:

$$(\{S_1(t,i): t \geqslant 0\}, \{S_1(t,i): val - t \geqslant 0\}) \tag{10}$$

$$(\{S_1(t,i): -1 - val + t \geqslant 0\}, \{S_1(t,i): 19 - t \geqslant 0\}) \tag{11}$$

对于一维 Jacobi 迭代计算的示例, 其 x=0、y=19, 当 *rate*=50 时, *val*=9, 划分之后的两个迭代空间分别是 $\{S_1(t,i): 0 \leqslant t \leqslant 9 \wedge 0 < i \leqslant 28; S_2(t,i): 0 \leqslant t \leqslant 9 \wedge 0 < i \leqslant 28\}$ 和 $\{S_1(t,i): 10 \leqslant t \leqslant 19 \wedge 0 < i \leqslant 28; S_2(t,i): 10 \leqslant t \leqslant 19 \wedge 0 < i \leqslant 28\}$, 经过迭代空间划分之后的调度树如图 6 所示, 图中绿色部分即为划分后的两个迭代空间.

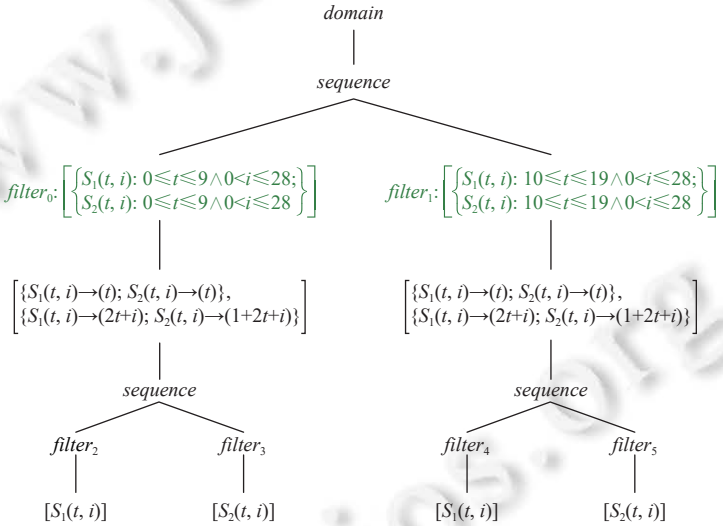

图 6 迭代空间划分之后的调度树

算法 1 在实现时由于仿射约束对需要自行提前判断, 且还需要考虑到各种非法情况以及对应的异常处理, 所以其具体的代码会比上面描述的过程更加复杂, 本文算法的实现代码可以参考论文开源代码中 *amp_get_filters*() 函数, *val* 的计算过程可以参考 *amp_get_partition_aff*() 函数, 如有需要, 可以对根据实际应用需求对该函数进行个性化定制和修改. 比如: 如果实际应用时需要修改公式 (9) 的分母为 10 000, 那么直接通过将该函数中的常数 100 修改为 10 000 即可.

需要说明的是, 该迭代空间划分算法等价于循环分段和循环剥离的结合, 并不会破坏程序内原有的依赖关系, 由依赖基本定理可知, 该程序变换可以保证其合法性[26]. 此外, 由 Stencil 计算最外层循环是时间步迭代, 内层循环是遍历并更新网格, 且内层和最外层没有依赖的计算特性可知, 对最外层循环进行划分既不会改变其时间步的先

后顺序, 还可以保证其引入的精度转换开销不会发生重复, 即确保了精度转换开销最小.

4.2.2 数据流分析

数据流分析在本文中是指通过对程序段中所有数组的引用以及语句实例的访存映射进行分析, 并对其中的数组引用进行分组、计算和分块, 从而得到混合精度计算所需要的强制类型转换的流映射关系和低精度数组的信息, 以便实施混合精度计算的过程. 由于循环嵌套中计算误差的特性, 一般而言混合精度计算都是先用高精度计算, 然后再用低精度计算, 因为只有这样整个过程中累积的误差才会最小, 计算结果才会最精确. 因此, 一般而言本过程计算的结果都只是低精度计算所在的第 2 个迭代空间中的相关数据, 换言之就是高精度计算所在的迭代空间的计算结果一般为空, 也就是其不需要进行任何的强制类型转换, 这显然与我们的常识相一致.

我们首先将所有需要的调度信息提取出来, 然后依次考虑每个数组, 对程序段中的数组引用进行分组, 检测过程的算法描述如算法 2.

算法 2. 数组引用分组检测算法.

输入: arrays, band_node_list;

输出: local_array.

```
get scheduling and other information of the band_node_list
foreach array of arrays do
    initialize the groups of array references
    foreach any two groups of array do
        foreach any access relation of groups do
            if the access relations is overlapping then
                if one of access is write then
                    combine these two groups into one group
                end
            end
        end
    end
    tiling the array reference groups according to the divided iteration domain, and copy array to local_array
    foreach any two groups of local_array do
        foreach any access relation of groups do
            if the access relations is overlapping then
                if one of access is write then
                    combine these two groups into one group
                end
            end
        end
    end
    calculate the index offset and other information, perform a series of checks at the same time
    update the information of the local_array
end
return local_array
```

首先对代码中数组引用进行初始化分组, 即一个数组的引用及其相关信息初始化为一个组, 但是, 如果两个组

中的引用访问了相同的数组元素, 并且如果两个引用中的至少一个是写入 (即存在依赖), 那么需要将这两个引用放在一起考虑 (合并这两个组), 因为仅考虑其中一个可能会导致数据不一致. 因此还需要通过遍历经过初始化的分组结果来判断是否存在两个分组有交叉访问关系, 然后再判断访问关系中是否包含写入, 如果同时满足这两个条件则需要将这两个引用组合并成一个组. 然后根据划分的迭代空间对合并后的数组引用组进行分块, 得到本地数组副本 local_array, 之后针对 local_array 再次进行数组引用组的遍历、判断和合并, 根据合并结果计算索引偏移量等信息, 同时进行一系列的检查, 最后更新 local_array 中的信息.

在得到本地数组 local_array 之后, 虽然其中的计算索引偏移量等信息已经更新, 但是其中的大小、边界等信息还是原始数组的信息副本, 以原始数组的大小为例, 其可能取决于参数, 但是利用提取出来的上下文信息可以获得对这些参数的限制, 进而可以简化这些表达式, 与此同时, 还可以利用数组引用分组的信息计算本地数组被引用部分的大小, 并将其与预处理阶段提取的上下文等信息进行结合进而更新和简化本地数组 local_array 中的大小和索引偏移量等信息. 标量数据也被当作数组处理, 并将其大小视为 1. 如果更新后的某一个数组大小是一个零函数或者 NULL, 则表示该数组在这里没有被引用, 由于访问函数实际上无法访问任何内容, 因此将数组大小打印为 0 并没有什么影响, 同时还可以提高算法的鲁棒性和健壮性.

以一维 Jacobi 迭代计算为例, A 和 B 两个数组都是被一个由 3 次读和 1 次写组成的引用组所访问, 但一般来说, 数组的不同部分可以通过不同的引用组访问, 我们可能只想将其中一些数据转换成低精度, 而不是整个数组, 因此对数组引用组分块和再次重新检测以及检查是必要的. 除此之外, 还可以根据数组引用组计算本地数组被引用部分的大小, 其大小即为混合精度优化阶段需要申请的该数组对应的低精度数组大小, 从而能够尽可能地降低自动混合精度优化所引入的内存开销.

最后, 标记所有数组引用组非空的 local_array 数组, 并为其生成强制类型转换的流映射关系, 并存储在 local_array 的数组引用组中, 同时在低精度所在的子树中插入一个包含流映射关系的 *extension* 结点. 对于一维 Jacobi 迭代计算示例, 其强制类型转换的流映射关系可以用公式 (12) 来表示:

$$\left\{\begin{array}{l}[[]\rightarrow A(i)]\rightarrow amp\_lower\_A(i);\\ [[]\rightarrow B(j)]\rightarrow amp\_lower\_B(j);\\ [[]\rightarrow alpha()]\rightarrow amp\_lower\_alpha();\\ [[]\rightarrow beta()]\rightarrow amp\_lower\_beta()\end{array}\right\} \tag{12}$$

对于输入流而言, 其含义是读取 A、B、$alpha$、$beta$ 的数据并存储在对应的 amp_lower_A、amp_lower_B、amp_lower_alpha、amp_lower_beta 中, 而对于输出流来说, 其含义则与之相反. 经过数据流分析之后的调度树如图 7 所示, 图中绿色部分便是强制类型转换的流映射关系.

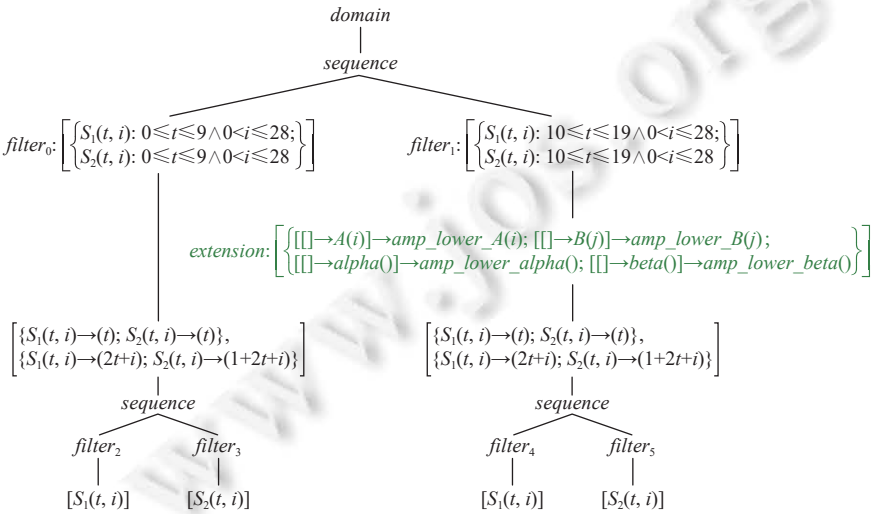

图 7 数据流分析之后的调度树

4.2.3 调度树转换

调度树转换过程是指依据划分后的迭代空间和数据流分析的结果, 通过对原始调度树中的结点施以各种合法的操作, 将其自动转换成符合混合精度计算的新的调度树的过程. 其具体的步骤如下.

(1) 通过算法 3 利用 local_array 中的信息计算低精度计算时所需的输入和输出流, 再将他们与流映射关系结合, 从而计算出强制类型转换对应的调度 *conv_in* 和 *conv_out*, 并生成对应的 *band* 结点.

(2) 在 *extension* 结点处创建一个 *sequence* 结点作为子结点, 然后依次为 *conv_in* 对应的 *band* 结点、低精度计算对应的 *band* 结点和 *conv_out* 对应的 *band* 结点创建父系的 *filter* 结点, 并按照计算的先后顺序依次插入 *sequence* 结点下.

调度转换算法本质上就是通过对树进行一系列操作以达到对其改造的目的, 但是其具体实现时由于要充分考虑各种情况的树形, 因此会设置很多 *mark* 结点, 并进行一系列的检查和异常处理, 并在最后将这些 *mark* 结点去掉, 以保持调度树的简洁. 值得一提的是, 本文在得到流映射关系后, 不是直接利用其将数组的全部数据都进行了强制类型转换和赋值, 而是采用了算法 3 来计算低精度计算时的输入和输出流, 并以此来生成强制类型转换的 *band* 结点, 尽可能地减少了不必要的强制类型转换, 从而能够充分发挥出混合精度程序的性能优势.

算法 3. 输入和输出流计算算法.

输入: local_array;
输出: in, out.

```
init in and out
foreach array of local_array do
    if reference groups of array not empty then
        foreach group in reference groups do
            get all read and write access relationships of the group
            delete the access relationships that are only needed to communicate data within current domain
            add the results into in and out
        end
    end
end
return in and out
```

对于一维 Jacobi 迭代计算示例而言, 其低精度计算的输入流可以用公式 (13) 来表示, 其输出流可以用公式 (14) 来表示, 其转换之后的调度树如图 8 所示, 图中绿色部分依次是输入和输出流.

$$\left\{\begin{array}{l} read[[] \to B(j)] : j = 0 \vee 29; \\ read[[] \to A(i)] : 0 \leqslant i \leqslant 29; \\ read[[] \to alpha()]; \\ read[[] \to beta()] \end{array}\right\} \tag{13}$$

$$\{write[[] \to A(i)] : 0 < i \leqslant 28\} \tag{14}$$

由于 PPCG 内嵌了对领域专用语言 (domain specified language, DSL) Pencil 语言的支持, 使其能够采用“制导+多面体编译优化”的模式对程序进行分析, 因此我们的优化器也天然地支持用户将__pencil_kill() 等编译指示添加到通用编程语言中, 为多面体模型进行特定优化提供领域相关信息. 同时, 我们这里的输出流和调度树都是经过编译指示优化之后的结果, 以便尽可能地还原 Stencil 计算在实际应用场景中的使用情景, 更准确地反映混合精度优化在实际应用场景中所能发挥的作用.

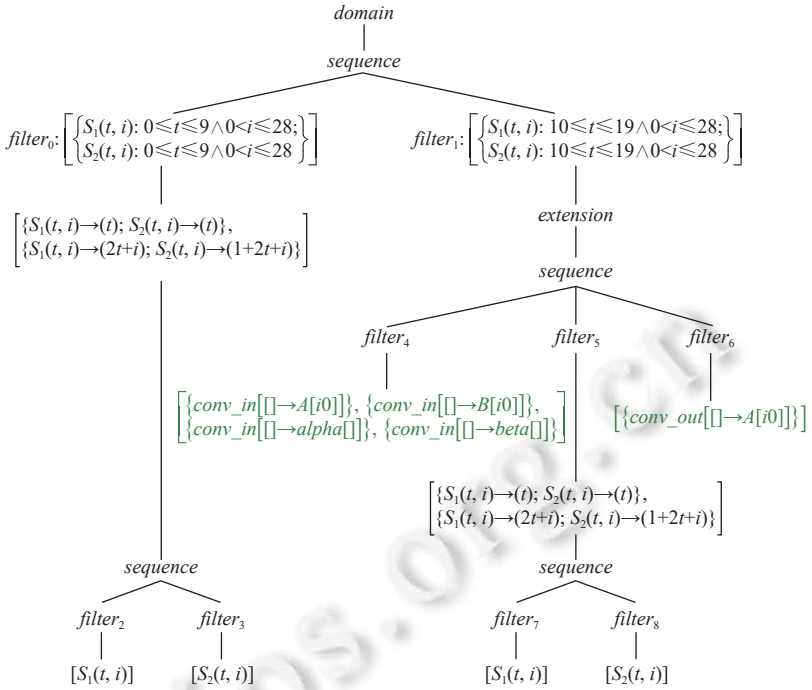

图 8 转换之后的调度树

4.3 代码生成

代码生成模块将经过混合精度优化后的调度树作为输入, 利用多面体扫描技术[19]生成与调度树相对应的 AST, 最后, 再以 AST 为基础参照 C 语言规范生成对应的代码. 多面体扫描技术其任务就是通过对调度树进行深度优先遍历来扫描调度树中的每一个结点, 并按照调度树所表示的顺序和规则生成对应的 AST. 利用 isl 库实现以调度树为输入的 AST 生成过程可以参考 PPCG 作者关于扫描多面体技术的文献 [19]. 从 AST 到最后目标程序语言的代码生成只需要根据 AST 向目标文件中输出相关语句即可, 最后再由基础编译器进行编译链接从而生成可执行文件, 便可得到最终的目标程序.

由于 PPCG 的代码生成模块做得十分细致且完善, 因此本文在实现时只是对其进行了一些扩展和改进, 以支持混合精度优化的强制类型转换等特殊代码的生成, 故不再赘述其生成细节.

一维 Jacobi 迭代计算示例最终生成的代码如后文图 9 所示, 可以看到代码中首先根据计算出来的 local_array 数组大小声明了低精度的数组和变量, 然后进行了原始的高精度计算, 之后便对 double 精度的数组进行了强制类型转换并赋值给了 float 精度的数组, 然后又进行了低精度的计算, 最后又将 float 的结果进行了强制类型转换并赋值给了 double 精度的数组, 保证了程序执行结果的正确性. 同时不难发现, 强制类型转换的所有数据都是经过细致分析和计算过的, 并没有任何多余的强制类型转换, 且还自动添加了 OpenMP 并行编程模型的编译指示, 具备多线程并行执行的能力. 此外, 一系列的实验结果表明混合精度优化后的代码与原始的代码相比, 具有更快的执行速度, 更好的程序性能, 能在一定程度上充分发掘程序中 double 计算和 float 计算的双流水并行执行的潜能, 实现指令流水级别的并行.

5 实验分析

为验证面向 Stencil 计算的自动混合精度优化器的有效性, 我们利用 PAPI (performance application programming interface) 性能测试工具[30]以及 Stencil 测试用例对自动生成的混合精度优化代码的正确性、误差和性能进行

评估. 我们首先利用 PAPI 性能测试工具读取硬件计数器[30], 即对程序运行时不同精度的操作进行计数, 从而验证混合精度的正确性; 其次将混合精度优化后代码与原始程序的高精度代码和低精度代码的运行结果进行对比, 以准确地评估其性能和误差; 最后对系统的编译时长进行了测试, 以评估优化器的性能.

```
float amp_lower_A[30], amp_lower_B[30], amp_lower_alpha, amp_lower_beta;
{
  for (int c0 = 0; c0 <= 9; c0 += 1)
    for (int c1 = 2 * c0 + 1; c1 <= 2 * c0 + 29; c1 += 1)
    {
      if (2 * c0 + 28 >= c1)
        B[−2 * c0 + c1] = (alpha * ((A[−2 * c0 + c1 −1] + A[−2 * c0 + c1]) + A[−2 * c0 + c1 + 1]));
      if (c1 >= 2 * c0 + 2)
        A[−2 * c0 + c1 −1] = (beta * ((B[−2 * c0 + c1 −2] + B[−2 * c0 + c1 −1]) + B[−2 * c0 + c1]));
    }
  // amp_lower
  {
    #pragma omp parallel for
    for (int c0 = 0; c0 <= 29; c0 += 1)
      amp_lower_A[c0] = (float)A[c0];
    amp_lower_B[0] = (float)B[0], amp_lower_B[29] = (float)B[29], amp_lower_alpha = (float)alpha, amp_lower_beta = (float)beta;
    for (int c0 = 10; c0 <= 19; c0 += 1)
      for (int c1 = 2 * c0 + 1; c1 <= 2 * c0 + 29; c1 += 1)
      {
        if (2 * c0 + 28 >= c1)
          amp_lower_B[−2 * c0 + c1] = (amp_lower_alpha * ((amp_lower_A[−2 * c0 + c1 −1] + amp_lower_A[−2 * c0 + c1]) + amp_lower_A[−2 * c0 + c1 + 1]));
        if (c1 >= 2 * c0 + 2)
          amp_lower_A[−2 * c0 + c1 −1] = (amp_lower_beta * ((amp_lower_B[−2 * c0 + c1 −2] + amp_lower_B[−2 * c0 + c1 −1]) + amp_lower_B[−2 * c0 + c1]));
      }
    #pragma omp parallel for
    for (int c0 = 1; c0 <= 28; c0 += 1)
      A[c0] = (double)amp_lower_A[c0];
  }
}
```

图 9 一维 Jacobi 迭代计算示例最终生成的代码

5.1 环境配置和测试用例

我们在 PPCG 原有功能的基础上提供了一个--automatic-mixed-precision 选项来选择是否开启自动混合精度优化 (默认开启), 一个--automatic-mixed-precision-rate (或-R) 选项用于接收用户自定义的混合比例. 本文实验所采用的测试平台的相关信息如表 1 所示.

表 1 测试平台信息

类型	x86平台	国产申威平台
Architecture	x86_64	Sunway
CPU	Intel(R) Xeon(R) Gold 6230 CPU 2-socket NUMA	SW26010P CPU
Clock	2.10 GHz	2.10 GHz
Cores	20 cores/socket, 2 socket (total: 40 cores)	41 932 800 cores
Hyperthreading	enabled	unknown
Cache sizes	64 KB L1, 1 024 KB L2, 28 160 KB L3	unknown
Compiler	gcc 9.4.0	SWGCC
Common flags	-O2 -DPOLYBENCH_USE_C99_PROTO -lm	-O2 -DPOLYBENCH_USE_C99_PROTO -lm
PAPI test flags	-DPOLYBENCH_PAPI -lpapi	-DPOLYBENCH_PAPI -lpapi
Error test flags	-DPOLYBENCH_DUMP_ARRAYS	-DPOLYBENCH_DUMP_ARRAYS
Performance test flags	-DPOLYBENCH_TIME	-DPOLYBENCH_TIME
PPCG	0.08.5-33	0.08.5-33
ppcg flags	--target=c [--no-automatic-mixed-precision/-R $rate]	--target=c [--no-automatic-mixed-precision/-R $rate]
Tiling options	--tile --tile-size=…	none
PAPI	6.0.0.1	not support
OS	Ubuntu 20.04.4 LTS (GNU/Linux 5.14.0-1027-oem)	unknown
runtime state	single core single thread	single MPE single thread

我们采用了 Polybench 测试集中 Stencil 计算的测试用例[31]、Pluto 编译器开源测试代码中 Stencil 计算的测试用例[21]等来验证本文的算法. 测试集中包含 10 个测试用例, 涵盖了 Stencil 计算的各种情况, 测试用例的信息如表 2 所示, 其中也给出了在国产申威平台和 x86 平台下各测试用例原始程序 (未进行循环分块优化) 的执行时间.

表 2 测试集信息

测试用例	类型	问题规模	国产申威平台执行时间 (s)		X86平台执行时间 (s)	
			double	float	double	float
3d7pt	星形	1200×128×128×128	54.84	36.06	3.75	6.98
3d27pt	盒型	800×128×128×128	108.51	90.48	14.56	14.45
fdtd-1d	星形	100000×10240	6.63	6.62	1.62	1.59
fdtd-2d	星形	3072×2047×2557	471.84	313.43	79.33	51.44
jacobi-1d	星形	819320×20480	16.32	15.77	3.09	2.12
jacobi-2d	星型	2048×2048×2048	360.68	312.04	23.53	18.60
heat-1d	星形	100000×10240	7.86	7.86	1.32	0.82
heat-2d	星形	8000×1024×1024	157.46	93.06	11.67	11.19
heat-3d	星形	1000×200×200×200	373.19	322.43	34.31	30.53
seidel-2d	盒型	2048×1024×1024	85.07	69.81	23.72	22.03

在该测试集中, 3d7pt 和 3d27pt 用例代表了大数据规模的 Stencil 计算, 对这些用例的测试可以验证混合精度优化在大型应用中的有效性; fdtd-1d/2d 用例中包含多条语句, 分别用于测试混合精度优化在特殊情况下的适用性; jacobi-1d/2d、seidel-2d 用例分别代表一维、二维的 Jacobi 计算以及 Gauss-Seidel 计算, 可测试其迭代计算在不同收敛情况下的适用性; heat-1d/2d/3d 用例分别代表一维、二维、三维的 heat 计算, 对这 3 个用例的分析可以说明混合精度优化在不同维度上的可扩展性.

5.2 PAPI 指令计数测试

从第 4.3 节生成的代码可以看到, 代码里面确实包括两个不同精度计算的循环, 但是在编译时编译器有可能会自动进行精度提升的操作, 因此不能从代码就直接得出已经实现了混合精度计算的结论, 还需要进一步的测试来证明. 于是, 我们通过调用 PAPI 测试工具[30]的接口实现了对硬件性能计数器进行访问, 从而得到循环嵌套内不同精度的操作在运行时计数的结果, 可以理解为不同精度类型的指令执行次数. 实验结果表明, 随着 *rate* 值的不断增大, double 操作逐渐增多, float 操作逐渐减少, 且两者总和保持不变, 这充分地证明了程序在运行时真正地实现了混合精度计算.

5.3 误差测试结果

在对混合精度进行误差测试时, 我们采用绝对误差来刻画不同结果的可靠程度, 并以 PPCG 的高精度计算的结果为基准, 其低精度计算的结果为对比进行测试, 即 *rate*=100 时高精度计算的误差为 0, *rate*=0 的误差表现即为低精度的误差表现. 其中 heat-1d/2d、jacobi-1d/2d、fdtd-1d/2d 计算的结果数组中的最大绝对误差如图 10 所示, 而图中没有展示的 seidel-2d 的最大绝对误差在任何比例下均为 0, 换言之, 就是其低精度的计算并不会带来精度损失, 而 3d7pt、3d27pt、heat-3d 的最大绝对误差在任何比例下均与低精度计算的误差保持相同, 分别为 0.000 029 45、0.000 054 58、0.000 000 76, 也就是说其对数据的精度是高度敏感的, 通过进一步分析可以发现, 其中的误差主要是由精度转换引起, 也就是其计算所需的数据对精度十分敏感.

从图 10 可以发现, 随着 *rate* 值的不断增大, 最大误差呈现线性下降的趋势, fdtd-1d、fdtd-2d 因为计算过程中额外产生了一些误差的累积和相消从而导致其下降的趋势不是严格的线性关系, 但总体趋势仍是在不断地下降. 这些都充分证明了混合精度计算中误差与其高精度计算的占比存在着一定的相关关系, 即从整体上看, 高精度计算占比越高, 误差就越小, 高精度计算占比越低, 误差也相应的越大. 而混合精度就是利用这个特性来尽可能减少不必要的精度冗余, 在一定的误差阈值下, 尽可能地减少高精度计算, 取而代之的是相对低精度的计算, 从而达到尽可能提高程序性能的目的.

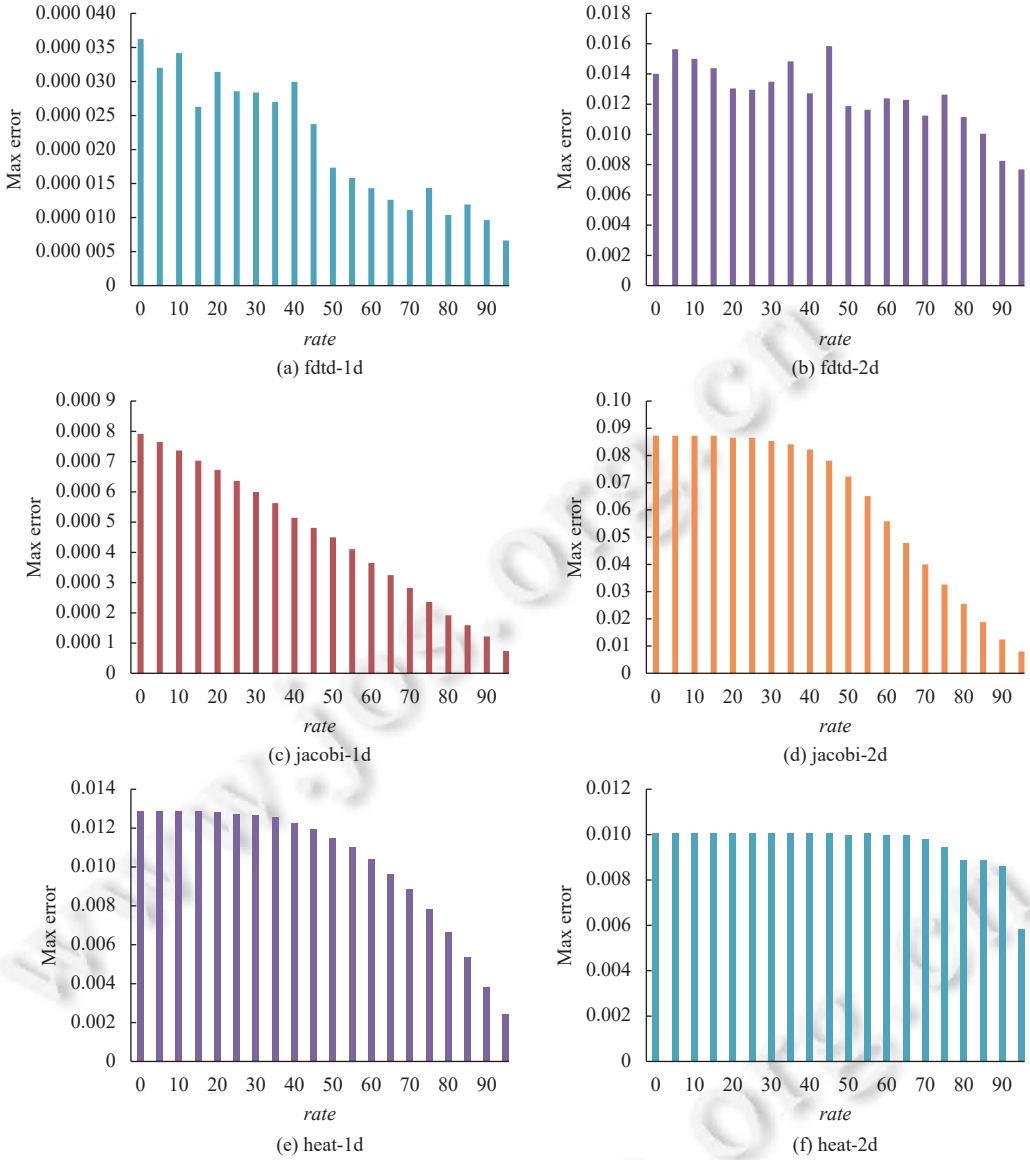

图 10 不同比例下的误差表现情况

5.4 性能测试表现

我们在性能测试时, 为了准确地测量混合精度指令级并行的加速效果, 同时测试了 PPCG 的 double 精度计算、float 精度计算、不同混合比例的混合精度计算代码的执行耗时, 对它们还进行了多种大小的循环分块, 找到各自的最小的执行耗时, 并以 double 精度计算为基准计算 float 精度计算的加速比以及混合精度计算的加速比, 从而来量化混合精度指令级并行的加速效果. 加速比的计算方法如公式 (15) 所示, 其中 T_{before} 代表加速前的执行耗时, T_{after} 代表加速后的执行耗时.

$$Speedup = \frac{T_{\text{before}}}{T_{\text{after}}} \tag{15}$$

此外, 为了避免执行耗时的测量会受到软件和硬件等环境因素的影响, 本文采取了执行程序 5 次, 并分别去掉一次最长和最短的执行耗时, 然后计算剩余 3 次执行的平均耗时来表示其执行耗时, 同时还计算了 3 次执行耗时

与平均执行耗时的最大绝对偏差, 来帮助评估执行耗时测量的精确度. 本文中展示的性能测试结果的最大绝对偏差均小于 5%.

由于混合比例为 100 时, 优化器不会对原始的高精度计算程序做混合精度优化; 混合比例为 0 时, 优化器生成的目标代码虽然只有低精度计算, 但是还会包含精度转换, 一方面从严格意义上来说并不属于混合精度计算, 另一方面实际应用中与用户手工修改原始程序得到的纯低精度版本相比, 性能较差, 无实际应用价值. 故下文中我们测试性能表现选取的混合比例是 $\{rate|rate\%5=0, rate\in[5,95]\}$, 且单精度 (float) 计算均指没有精度转换开销的纯 float 计算.

在新一代国产申威平台上, 由于其主核 (MPE) 上的指令更为全面和完善, 因此我们选择了其主核进行测试. 同时受其 Cache 大小的影响, 我们没有对其进行循环分块来提升数据局部性. 另外, 由于两个平台指令集和架构的不同, 其性能表现也有较为明显的差异, 一方面表现在同一测试用例的加速效果并不相同, 其中最典型的就是 3d7pt 用例, 在 X86 平台上受其计算的类型、计算的宽度、数据量、Cache 大小、自动向量化和 SSE 指令架构等特性的影响, 其 double 计算快于 float 计算, 而申威平台自动向量化和指令集等较为简单, 导致其并没有出现这样的加速效果; 另一方面还表现在性能最优时混合比例的不同. 不同测试用例在不同平台上性能最优时的混合比例大小如表 3 所示, 其中 Tile size 代表 x86 平台上性能最优时循环分块的分块大小, *rate* 代表性能最优时的混合比例大小.

表 3 性能最优时的混合比例

测试用例	国产申威平台	x86平台	
	rate	Tile size	*rate*
3d7pt	5	0	95
3d27pt	5	512	5
fdtd-1d	5	8 192	5
fdtd-2d	5	0	5
jacobi-1d	95	1 024	95
jacobi-2d	5	4 096	95
heat-1d	55	8 192	95
heat-2d	5	0	5
heat-3d	90	0	90
seidel-2d	5	4	5

x86 平台上所有测试用例的性能表现如图 11 所示, 从图中可以看到, 单精度计算的最大加速比是 1.86, 几何平均加速比是 1.12; 混合精度计算的最大加速比是 1.76, 几何平均加速比是 1.15, 高于单精度.

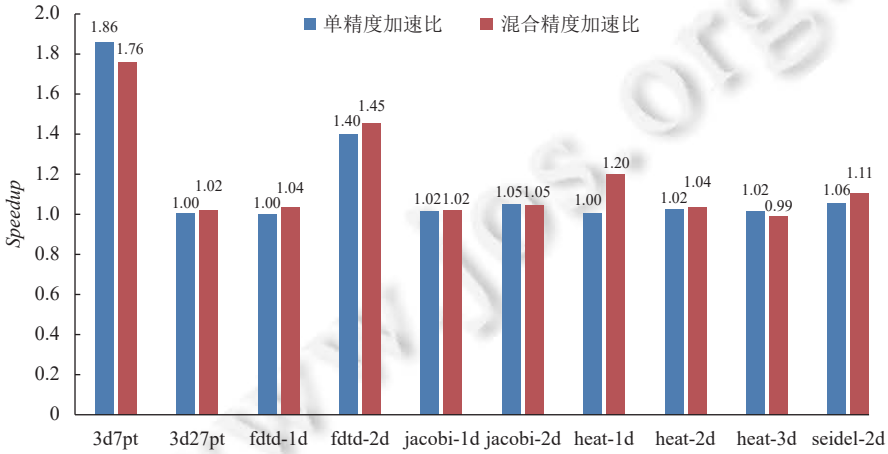

图 11 x86 平台上不同测试用例的加速比

申威平台上所有测试用例的性能表现如图 12 所示, 从图中可以看到, 单精度计算的最大加速比是 1.69, 几何平均加速比是 1.23; 混合精度计算的最大加速比是 1.64, 几何平均加速比是 1.20.

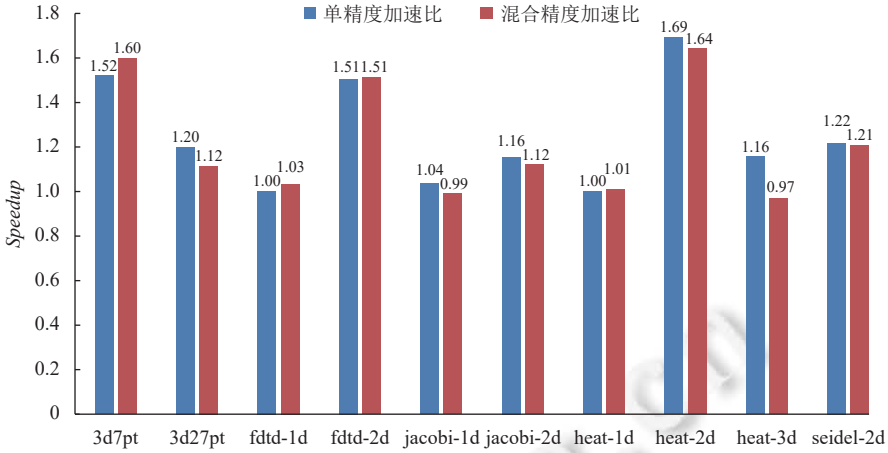

图 12 申威平台上不同测试用例的加速比

由于 heat-3d 本身的计算和访存特性的影响, 混合精度优化后的运行速度不升反降, 故其加速比略小于 1, 但是从整体上看, 混合精度的加速效果还是值得肯定的. 另外, 从表 2 可知 3d7pt 的 float 精度的计算比 double 精度的计算要慢, 因此在 x86 平台上该用例的数据是以 float 精度为基准, 以 double 精度为对比对象计算的.

5.5 编译时长测试

本文除了对自动生成的混合精度程序代码开展性能测试, 还对代码生成过程中的编译时长也进行了测试, 从而达到以量化的方式准确评估优化器性能的目的, 编译时长测试结果如表 4 所示.

表 4 编译时长测试结果

方法	3d7pt	3d27pt	fdtd-1d	fdtd-2d	jacobi-1d	jacobi-2d	heat-1d	heat-2d	heat-3d	seidel-2d	Average
PPCG	505	3 188	135	420	148	347	146	250	1 250	208	659.5
Our	642	3 748	142	505	163	413	159	321	1 549	252	789.5

整个测试中 10 个测试用例的平均编译时长为 789.5 ms, 其中耗时最长的 3d27pt 用例编译时长为 3 748 ms, 与 PPCG 相比编译时长平均多 130 ms, 而手工编写 Stencil 计算的混合精度程序至少需要以小时为单位的时间, 甚至面对复杂应用时需要以天为单位计算. 相比之下, 利用本文提出的面向 Stencil 计算的自动混合精度优化器能够大大降低混合程序开发的成本, 从而解决 Stencil 计算的混合精度程序的编程难的问题.

6 总 结

由于混合精度可以减少计算中不必要的精度冗余, 并且其内存占用更少、计算速度更快, 因此受到了极大的关注, 很多研究人员都对其开展了研究, 目前虽然已经在精度调整和混合精度训练方面取得了相当不错的进展, 但是循环嵌套级的混合精度仍然是一个很具有挑战性的问题. 另一方面, 基于多面体模型的各种编译工具层出不穷, 其将循环嵌套抽象成空间多面体, 并通过多面体上的几何操作来优化程序, 在程序自动并行化领域取得了许多突破. 本文将多面体模型和混合精度结合在了一起, 设计并实现了第 1 个面向 Stencil 计算的自动混合精度优化器. 该优化器在基于多面体模型的一般编译流程的中间表示层, 扩展并实现了迭代空间划分、数据流分析和调度树转换, 使得能够在调度树上实施混合精度优化, 从而实现面向 Stencil 计算的自动混合精度优化. 对 Stencil 计算用例测试的实验结果证明了所提出方案的有效性和实用性.

必须承认的是, 面向 Stencil 计算的自动混合精度优化器仍有很多优化的潜力, 一方面我们目前正在支持除 Stencil 计算外的其他常用的计算模式, 另一方面我们目前也正在尝试在提取多面体模型之后, 进行迭代空间划分之前, 根据循环嵌套内的计算语句及其边界, 对其误差和性能的提升以及损耗进行建模, 并通过对该模型计算直接得到误差阈值下性能最好的混合比例, 而不再需要用户指定比例参数, 并用该模型来指导后续过程, 从而直接自动

生成最优化的混合精度优化代码. 另外, 我们还在考虑对面向 Stencil 计算的混合精度优化器进行优化和扩展, 使其支持更多的循环计算类型, 从而实现循环嵌套级的自动混合精度优化, 并将其扩展到 GPU 等异构架构上, 以提高其通用性和对主流异构架构的适用性. 不仅如此, 我们还在尝试对其编译后端进行扩展, 使其能够自动生成指定平台下二进制文件, 而不再只是生成目标代码, 并在其进行指令选择和重排序时, 采用一定的策略和算法实现高精度和低精度计算的多流水并行, 最大幅度地挖掘混合精度计算的性能优势.

References:

[1] Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz JD, Lee EA, Morgan N, Necula G, Patterson DA, Sen K, Wawrzynek J, Wessel D, Yelick KA. The parallel computing laboratory at U. C. Berkeley: A research agenda based on the Berkeley view. Technical Report, Berkeley: University of California at Berkeley, 2008.

[2] Taflove A, Hagness SC, Piket-May M. Computational electromagnetics: The finite-difference time-domain method. In: Chen WK, ed. The Electrical Engineering Handbook. Boston: Elsevier, 2005. 629–670. [doi: 10.1016/B978-012170960-0/50046-3]

[3] Xu JC, Huang YZ, Guo SZ, Zhou B, Zhao J. Testing platform for floating mathematical function libraries. Ruan Jian Xue Bao/Journal of Software, 2015, 26(6): 1306–1321 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4589.htm [doi: 10.13328/j.cnki.jos.004589]

[4] DeVries PMR, Viégas F, Wattenberg M, Meade BJ. Deep learning of aftershock patterns following large earthquakes. Nature, 2018, 560(7720): 632–634. [doi: 10.1038/s41586-018-0438-y]

[5] Micikevicius P, Narang S, Alben J, Diamos GF, Elsen E, Garcia D, Ginsburg B, Houston M, Kuchaiev O, Venkatesh G, Wu H. Mixed precision training. In: Proc. of the 6th Int'l Conf. on Learning Representations. Vancouver: OpenReview.net, 2018.

[6] Cherubin S, Agosta G. Tools for reduced precision computation: A survey. ACM Computing Surveys, 2021, 53(2): 33. [doi: 10.1145/3381039]

[7] Damouche N, Martel M. Salsa: An automatic tool to improve the numerical accuracy of programs. Automated Formal Methods, 2018, 5: 63–76.

[8] Cherubin S, Cattaneo D, Chiari M, Di Bello A, Agosta G. TAFFO: Tuning assistant for floating to fixed point optimization. IEEE Embedded Systems Letters, 2020, 12(1): 5–8. [doi: 10.1109/LES.2019.2913774]

[9] Darulova E, Kuncak V, Majumdar R, Saha I. Synthesis of fixed-point programs. In: Proc. of the 2013 Int'l Conf. on Embedded Software (EMSOFT). Montreal: IEEE, 2013. 1–10. [doi: 10.1109/EMSOFT.2013.6658600]

[10] Kotipalli PV, Singh R, Wood P, Laguna I, Bagchi S. AMPT-GA: Automatic mixed precision floating point tuning for GPU applications. In: Proc. of the 2019 ACM Int'l Conf. on Supercomputing. Phoenix: ACM, 2019. 160–170. [doi: 10.1145/3330345.3330360]

[11] Chiang WF, Baranowski M, Briggs I, Solovyev A, Gopalakrishnan G, Rakamarić Z. Rigorous floating-point mixed-precision tuning. In: Proc. of the 44th ACM SIGPLAN Symp. on Principles of Programming Languages. Paris: ACM, 2017. 300–315. [doi: 10.1145/3009837.3009846]

[12] Kum KI, Kang JY, Sung W. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing, 2000, 47(9): 840–848. [doi: 10.1109/82.868453]

[13] Menon H, Lam MO, Osei-Kuffuor D, Schordan M, Lloyd S, Mohror K, Hittinger J. ADAPT: Algorithmic differentiation applied to floating-point precision tuning. In: Proc. of the 2018 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Dallas: IEEE, 2018. 614–626. [doi: 10.1109/SC.2018.00051]

[14] Nathan R, Naeimi H, Sorin DJ, Sun XB. Profile-driven automated mixed precision. arXiv:1606.00251, 2016.

[15] Rubio-González C, Nguyen C, Nguyen HD, Demmel J, Kahan W, Sen K, Bailey DH, Iancu C, Hough D. Precimonious: Tuning assistant for floating-point precision. In: Proc. of the 2013 Int'l Conf. on High Performance Computing, Networking, Storage and Analysis. Denver: ACM, 2013. 27. [doi: 10.1145/2503210.2503296]

[16] Rubio-González C, Nguyen C, Mehne B, Sen K, Demmel J, Kahan W, Iancu C, Lavrijsen W, Bailey DH, Hough D. Floating-point precision tuning using blame analysis. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: ACM, 2016. 1074–1085. [doi: 10.1145/2884781.2884850]

[17] Guo H, Rubio-González C. Exploiting community structure for floating-point precision tuning. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 333–343. [doi: 10.1145/3213846.3213862]

[18] Feautrier P, Lengauer C. Polyhedron model. In: Padua D, ed. Encyclopedia of Parallel Computing. Boston: Springer, 2011. 1581–1592. [doi: 10.1007/978-0-387-09766-4_502]

[19] Grosser T, Verdoolaege S, Cohen A. Polyhedral AST generation is more than scanning polyhedra. ACM Trans. on Programming Languages and Systems, 2015, 37(4): 12. [doi: 10.1145/2743016]

[20] Zhao J, Li YY, Zhao RC. “Black magic” of polyhedral compilation. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2371–2396 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5563.htm [doi: 10.13328/j.cnki.jos.005563]

[21] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. In: Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Tucson: ACM Press, 2008. 101–113. [doi: 10.1145/1375581.1375595]

[22] Verdoolaege S, Carlos Juega J, Cohen A, Gómez JI, Tenllado C, Catthoor F. Polyhedral parallel code generation for CUDA. ACM Trans. on Architecture and Code Optimization, 2013, 9(4): 54. [doi: 10.1145/2400682.2400713]

[23] Trifunovic K, Cohen A, Edelsohn D, Li F, Grosser T, Jagasia H, Ladelsky R, Pop S, Sjödin J, Upadrasta R. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. 2010. https://hal.inria.fr/inria-00551516/

[24] Grosser T, Groesslinger A, Lengauer C. Polly—Performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters, 2012, 22(4): 1250010. [doi: 10.1142/S0129626412500107]

[25] Kelly W, Pugh W. A unifying framework for iteration reordering transformations. In: Proc. of the 1st Int’l Conf. on Algorithms and Architectures for Parallel Processing. Brisbane: IEEE, 1995. 153–162. [doi: 10.1109/ICAPP.1995.472180]

[26] Girbal S, Vasilache N, Bastoul C, Cohen A, Parello D, Sigler M, Temam O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int’l Journal of Parallel Programming, 2006, 34(3): 261–317. [doi: 10.1007/s10766-006-0012-3]

[27] Verdoolaege S. Counting affine calculator and applications. 2011. http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-05-slides.pdf

[28] Verdoolaege S, Grosser T. Polyhedral extraction tool. In: Proc. of the 2nd Int’l Workshop on Polyhedral Compilation Techniques. Paris: IMPACT, 2012. 1–8.

[29] Verdoolaege S. isl: An integer set library for the polyhedral model. In: Proc. of the 3rd Int’l Congress on Mathematical Software. Kobe: Springer, 2010. 299–302. [doi: 10.1007/978-3-642-15582-6_49]

[30] Terpstra D, Jagode H, You HH, Dongarra J. Collecting performance data with PAPI-C. In: Müller MS, Resch MM, Schulz A, Nagel WE, eds. Tools for High Performance Computing 2009. Berlin: Springer, 2010. 157–173. [doi: 10.1007/978-3-642-11261-4]

[31] PolyBench/C 4.2. 2018. https://sourceforge.net/projects/polybench/

附中文参考文献:

[3] 许瑾晨, 黄永忠, 郭绍忠, 周蓓, 赵捷. 一个浮点数学函数库测试平台. 软件学报, 2015, 26(6): 1306–1321. http://www.jos.org.cn/1000-9825/4589.htm [doi: 10.13328/j.cnki.jos.004589]

[20] 赵捷, 李颖颖, 赵荣彩. 基于多面体模型的编译“黑魔法”. 软件学报, 2018, 29(8): 2371–2396. http://www.jos.org.cn/1000-9825/5563.htm [doi: 10.13328/j.cnki.jos.005563]

宋广辉(1997－), 男, 硕士生, 主要研究领域为高性能计算, 先进编译技术.

陶小涵(1996－), 男, 博士生, 主要研究领域为先进编译技术.

郭绍忠(1964－), 女, 教授, CCF 高级会员, 主要研究领域为高性能计算, 分布式处理.

李飞(1996－), 男, 硕士生, 主要研究领域为高性能计算.

赵捷(1987－), 男, 讲师, CCF 专业会员, 主要研究领域为先进编译技术.

许瑾晨(1987－), 男, 讲师, 主要研究领域为高性能计算.