计算机科学
COMPUTER SCIENCE

面向矩阵乘计算的自动混合精度优化

何昊天, 周蓓, 郭绍忠, 张作言, 郝江伟, 冀立光, 许瑾晨

引用本文

何昊天, 周蓓, 郭绍忠, 张作言, 郝江伟, 冀立光, 许瑾晨. 面向矩阵乘计算的自动混合精度优化[J]. 计算机科学, 2024, 51(11A): 240300057-10.

HE Haotian, ZHOU Bei, GUO Shaozhong, ZHANG Zuoyan, HAO Jiangwei, JI Liguang, XU Jinchen. Automatic Mixing Precision Optimization for Matrix Multiplication Calculation[J]. Computer Science, 2024, 51(11A): 240300057-10.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240300057

面向矩阵乘计算的自动混合精度优化

何昊天　周　蓓　郭绍忠　张作言　郝江伟　冀立光　许瑾晨
信息工程大学网络空间安全学院　郑州 450001
(m18503880251@163.com)

摘　要　针对矩阵乘计算的混合精度优化的实现,极大地提升了矩阵乘计算的性能,但与高精度矩阵乘计算相比,混合精度矩阵乘计算时引入了误差。为有效降低混合精度计算中引入的误差,实现了一个面向矩阵乘计算的自动混合精度工具 AMAO。该工具在低精度乘高精度加基础混合精度计算的基础上,通过迭代空间进行划分的精度优化算法将原本的基础混合精度计算按照一定比例划分成两部分计算,一部分用高精度计算,另一部分用基础混合精度计算,并根据该算法实现了混合精度代码自动生成工具。实验表明,与混合精度工具 AGMMMPC 相比,AMAO 生成的混合精度代码性能平均降低 5.90%,精度平均提升了 49.31%。

关键词:混合精度;矩阵乘法;多面体模型;调度变换;代码生成
中图分类号　TP314

Automatic Mixing Precision Optimization for Matrix Multiplication Calculation

HE Haotian,ZHOU Bei,GUO Shaozhong,ZHANG Zuoyan,HAO Jiangwei,JI Liguang and XU Jinchen
School of Cyberspace Security,University of Information Engineering,Zhengzhou 450001,China

Abstract　The implementation of mixed-precision optimization for matrix multiplication computation greatly improves the performance of matrix multiplication computation,but at the same time,compared with high-precision matrix multiplication computation,mixed-precision matrix multiplication computation introduces errors. In order to effectively reduce the errors introduced in the mixed-precision computation,this paper implements an automatic mixed-precision tool AMAO for matrix multiplication computation. On the basis of low precision times high precision plus basic mixing accuracy calculation,the tool divides the original basic mixing accuracy calculation into two parts according to a certain proportion through the precision optimization algorithm of iterative space division,one part uses high precision calculation method and the other part uses the basic mixing accuracy calculation method,and realizes the automatic generation tool of mixed accuracy code according to the algorithm. Experiments show that compared with the mixed-precision tool AGMMMPC,the performance of mixed-precision codes generated by AMAO is reduced by 5.90% on average,and the accuracy is improved by 49.31% on average.

Keywords　Mixed precision,Matrix multiplication,Polyhedral model,Scheduling transformation,Code generation

1　引言

矩阵乘在计算机科学中有着广泛的应用,例如在机器学习中用于卷积神经网络和神经网络计算[1],在人工智能领域,计算神经网络的权重矩阵,以及执行矩阵分解操作都需要用到矩阵乘法,矩阵乘代码的执行效率将直接影响整个程序的性能。因此,如何更好地提升矩阵乘的执行效率变得特别关键。用混合精度方法对矩阵乘计算进行优化已经成为近年来的研究热点之一,主要是因为其可以有效地提高计算性能。

作者在之前的工作中面向矩阵乘计算设计实现了自动混合精度工具 AGMMMPC。该工具以高精度矩阵乘计算源程序为输入,经过初始化、基础混合两个步骤自动生成低精度乘高精度加基础混合精度矩阵乘计算代码,提高了矩阵乘运算的性能。然而与高精度矩阵乘计算相比,低精度乘高精度加基础混合精度矩阵乘计算引入了误差。为了降低误差,该工具通过精度调优算法找到基础混合精度矩阵乘计算中误差大的点,将这些点用高精度计算,从而降低基础混合精度矩阵乘计算中的误差,并自动生成高级混合精度矩阵乘代码。然而根据这种精度调优算法生成的高级混合精度矩阵乘代码并不能有效降低基础混合精度矩阵乘计算中引入的误差。为了进一步有效降低基础混合精度矩阵乘计算中的误差,本文在基础混合精度矩阵乘计算的基础上提出了另一种迭代空间划分的精度优化算法,并根据算法设计实现了矩阵乘混合精度优化工具 AMAO(Automatic Mixing Accuracy Optimization)。

本文的主要贡献如下:

1)在基础混合精度矩阵乘计算的基础上,提出了迭代空间划分算法。

2)在基础混合精度矩阵乘计算的基础上,根据迭代空间划分算法,设计实现自动混合精度优化的工具。

3)选取多个矩阵规模的矩阵乘法测试用例进行性能和精度测试。相比高级混合精度矩阵乘计算,生成的矩阵乘混合精度程序性能平均降低 5.90%,精度平均提升 49.31%。

本文第 2 章介绍了混合精度相关知识、多面体模型技术和调度树相关知识;第 3 章简要介绍 AMAO 工具自动生成框

通信作者:许瑾晨(atao728208@126.com)

架并详细介绍了 AMAO 工具实现细节；第 4 章对生成的混合精度代码进行测试和分析；最后总结全文并展望未来。

2 相关工作

2.1 混合精度

混合精度技术是一种在计算过程中同时使用不同精度数据类型的方法，适用于存在精度冗余的程序。它可以同时运用高精度和低精度计算，以提高计算效率。目前，混合精度技术主要分为两类：算子级混合精度和变量级混合精度。算子级混合精度主要应用于深度学习训练等人工智能领域，而变量级混合精度则用于各种精度调整和分析工具中。不论是哪种混合精度技术，它们的共同目标都是在保持适当计算精度的基础上，提高计算效率并减少资源消耗。混合精度技术的研究和应用对于深度学习训练、高性能计算以及其他相关领域的性能优化至关重要。通过合理选择和应用混合精度技术，能够获得更好的计算结果，加速解决各种复杂问题的过程。

2.1.1 变量级混合精度

变量级混合精度，主要用于各种精度调优和分析工具。混合精度工具一般要求用户在使用时给定应用程序可接受的误差阈值。分析精度调整后的混合精度程序是否可以满足要求有两种方法，一种是静态分析方法，一种是动态分析方法。米兰理工大学的 Stefano 教授发表的一项调查对 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[2]。

静态分析方法不依赖于特定的输入，不需要实际的执行程序，但经常会给出过于保守的精度分配方案，从而无法找到最优解。使用静态分析的混合精度工具有 AMPT-GA[3]，FPTuner[4]，Daisy[5]，Rosa[6]，TAFFO[7]等。这些工具旨在找到最佳的精度分配方案，从而实现更高效的程序运行。

动态分析，相比静态分析能够更准确地给出较优的混合精度方案，同时耗时也大大减少。因此，在精度调整工具的范畴内，动态分析方法更为常见。使用动态分析的混合精度工具包括 Autoscaler For C[8]，Blame Analysis[9]，ADAPT[10]，fpPrecisionTuning[11]，GPUMixer[12]等。

2.1.2 算子级混合精度

算子级混合精度(Mixed-precision Arithmetic)是一种优化计算性能和减少存储器使用的技术，它通过使用不同精度的数据类型来计算复杂的数学运算。2018 年，百度与 NVIDIA 联合发表论文：MIXED PRECISION TRAINING[13]。其中提出了混合精度训练的方法，该方法在深度学习模型训练时使用 16 位浮点数(FP16)，减少了内存需求并提高了硬件效率。混合精度训练的引入，对于模型训练效率的提高和成本的降低具有重要意义。目前，混合精度训练已被广泛应用于图像分类、语音识别、自然语言处理等领域。

在实际应用中，算子级混合精度已经被广泛地应用于深度学习等领域中，可以大幅提高计算性能，降低功耗，减少存储器占用等。但是，由于精度差异可能会影响模型的收敛速度和精度，因此需要根据具体情况来设计和优化混合精度计算方案。

2.2 多面体模型

多面体模型[14]是一种高度抽象化的编译优化模型，主要用于分析和改进程序性能。它采用紧凑的表示方法单独处理每个语句实例和每个数组元素[15]。这种模型具有广泛的应用范围、强大的表示能力和广泛的优化范围[16]。通过多面体模型，我们可以利用迭代空间、访存关系、依赖关系和调度等元素来描述程序及其语义，从而更好地理解程序的结构和行为，进而进行有效的性能优化和转换。

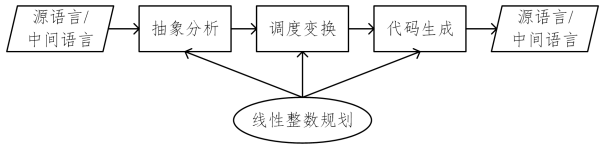

图 1 多面体模型的一般编译流程

Fig. 1 General compilation flow for polyhedral models

多面体编译工具通常包括抽象分析、调度变换和代码生成 3 个阶段，而线性整数规划一直贯穿于整个流程中[16]。在抽象分析阶段，工具首先识别需要处理的程序段，然后将其转换为多面体形式，并计算迭代空间和访存映射。接着，根据迭代空间和访存映射，通过线性整数规划计算依赖关系。调度变换阶段为中间优化部分，在满足依赖关系的前提下，利用线性整数规划计算出适合目标程序语言和体系结构特点的调度方案，也包括循环变换的结果。最后，代码生成阶段根据目标代码程序的语言规范将线性整数规划生成的抽象语法树生成最终代码。

在程序自动并行化领域，多面体模型已经取得了许多引人注目的成果。除了 Pluto[17]和 PPCG[18]等代表性工具以外，多面体模型在开发和商业应用中的影响不断扩大，如 GCC 的 Graphite[19]框架、LLVM 的 Polly 模块[20]，以及多面体模型在 Open64 和 IBM XL 编译器中的运用。在实现源到源的程序转换过程中，多面体模型不仅作为通用编译器的编译遍或模块进行嵌入，还作为独立的编译工具序，体现出了多面体编译技术的独立性和兼容性。

2.3 调度树

调度树(Schedule Tree)是一种数据结构，用于表示程序的循环嵌套结构，主要用于在并行计算中对程序进行并行化和优化。调度主要描述了程序中语句实例的执行顺序，因此调度本质上可以被当成树形结构。在多面体模型中，除了常见的集合和映射之外，还存在多种其他中间表示形式(IR)来表示执行过程，如 Presburger 算数表达式[23]、Kelly 等[21]提出的中间表示、“$2d+1$”形式的中间表示[22]等。

调度树是由多个结点构成的数据结构，每个结点代表一个循环结构，包括循环体、循环范围和迭代次数等信息。一般情况下，调度树中包含的结点类型主要有 domain，sequence，filter，band、mark 和 extension 等。其中，domain 结点是调度树的根结点；sequence 结点的子结点 filter 必须按照先后顺序执行；filter 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 结点用于表示当前子树中的备注信息；程序员进行可扩展的操作可以用 extension 结点实现。关于调度树结点的详细信息和结点生成抽象语法树的算法和实现原理，可以参考文献[15]。

通过使用调度树，对基础混合精度矩阵乘循环程序的优化更加方便，因为调度树的形状类似于树形结构，易于理解和修改，可以轻松地进行修改和插入操作。在程序员确定调度树等相关信息符合规范的前提下，多面体编译器(如 PPCG)

可以根据调度树生成相应的抽象语法树，从而自动生成针对不同体系结构的并行代码。在本文中，多面体模型中调度的中间表示形式选用调度树，从而实现针对矩阵乘计算的自动混合精度优化。

3 AMAO 的框架及实现

本文在低精度乘高精度加基础混合精度基础上，为了降低基础混合精度矩阵乘计算引入的误差，设计实现了自动混合精度工具 AMAO，工具的示意图如图 2 所示。该工具以基础混合精度 C 语言程序为输入，根据用户指定高精度计算在基础混合精度计算中的比例，通过预处理、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。其中，精度优化模块是本文工作的核心。

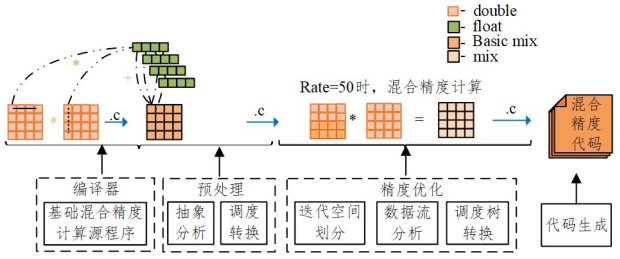

图 2 AMAO 工具示意图

Fig. 2 AMAO tool diagram

3.1 预处理

预处理主要进行抽象分析和调度变换两个阶段操作。在抽象分析阶段主要使用 PET[24] 库，从用户输入的 C 语言矩阵乘源代码中识别出由用户编译指示指定的目标程序段。然后获取该程序段的迭代空间、访存关系以及调度，均以仿射约束形式表示等信息。

迭代空间信息存放程序段中所有语句实例的集合，每个语句实例与一个唯一的迭代向量相对应。以图 3 所示的基础混合精度矩阵乘计算的核心代码为例，其迭代空间可以用式(1)表示。

```
float low_A[1024][1024];
float low_B[1024][1024];
// low
// amp_kernel
{
  for (int c0 = 0; c0 <= 1023; c0 += 1)
    for (int c1 = 0; c1 <= 1023; c1 += 1)
      low_A[c0][c1] = (float)A[c0][c1];
  for (int c0 = 0; c0 <= 1023; c0 += 1)
    for (int c1 = 0; c1 <= 1023; c1 += 1)
      low_B[c0][c1] = (float)B[c0][c1];
  for (int c0 = 0; c0 <= 1023; c0 += 1)
    for (int c1 = 0; c1 <= 1023; c1 += 1)
      for (int c2 = 0; c2 <= 1023; c2 += 1)
        C[c0][c1] = (C[c0][c1] + (low_A[c0][c2] * low_B[c2][c1]));
}
```

图 3 基础混合精度矩阵乘计算代码

Fig. 3 Calculation codes of basic mixed-precision matrix multiplication

$$\{S_0(i,j): 0\leqslant i<M \wedge 0\leqslant j<N;$$
$$S_1(i,j): 0\leqslant i<M \wedge 0\leqslant j<N;$$
$$S_2(i,j,k): 0\leqslant i<M \wedge 0\leqslant j<N \wedge 0\leqslant k<K\} \quad (1)$$

其中，每一个迭代向量 i,j,k 都与语句 S_0 和 S_1 的一次执行实例一一对应。

访存映射是将语句实例与语句实例访问的数组元素之间建立起映射关系的集合。它包括了读取和写入访存映射关系。图 3 所示基础混合精度矩阵乘计算代码的写访存关系为：

$$\{S_0(i,j)\rightarrow low\_A(i,j): 0\leqslant i<M \wedge 0\leqslant j<N;$$
$$S_1(i,j)\rightarrow low\_B(i,j): 0\leqslant i<M \wedge 0\leqslant j<N;$$
$$S_2(i,j,k)\rightarrow C\_mix(i,j): 0\leqslant i<M \wedge 0\leqslant j<N\} \quad (2)$$

读访存关系：

$$\{S_0(i,j,k)\rightarrow A(i,k): 0\leqslant i<M \wedge 0\leqslant k<K;$$
$$S_1(i,j,k)\rightarrow B(k,j): 0\leqslant k<K \wedge 0\leqslant j<N;$$
$$S_2(i,j,k)\rightarrow low\_A(i,k): 0\leqslant i<M \wedge 0\leqslant k<K;$$
$$S_2(i,j,k)\rightarrow low\_B(k,j): 0\leqslant k<K \wedge 0\leqslant j<N;$$
$$S_2(i,j,k)\rightarrow C\_mix(i,j): 0\leqslant i<M \wedge 0\leqslant j<N\} \quad (3)$$

调度用于表示程序中语句的执行顺序，并将每个语句映射到一个整数元组上。这些整数元组按照字典序排序，以确定程序中语句实例的执行顺序。例如，在矩阵乘法计算中，原始的调度可以表示为式(4)。

$$Schedule=\{S_0(i,j)\rightarrow(i,j); S_1(i,j)\rightarrow(i,j);$$
$$S_2(i,j,k)\rightarrow(i,j,k)\} \quad (4)$$

判断程序是否正确执行和程序变换是否合法的重要因素是依赖关系。在使用调度变换来发掘程序并行性之前，必须对程序的依赖关系进行仔细分析处理，以确保程序的正确性和可靠性。以基础混合精度矩阵乘计算为例，根据访存关系，其语句实例间的依赖关系为：

$$\{S_0(i,j)\rightarrow S_1(i,j,0); S_1(i,j)\rightarrow S_2(i,j,0); S_2(i,j,k)\rightarrow S_2(i,j,k+1)\} \quad (5)$$

多面体编译工具通过将循环嵌套内的语句表示成空间多面体形式，在程序自动并行化领域进行调度变换，以提升程序并行性和数据局部性。本文采用了改进后的 ISL[25] 调度算法来对循环嵌套进行变换，以确保程序具有良好的并行性和局部性。调度树变换前后的示意图如图 4 所示。

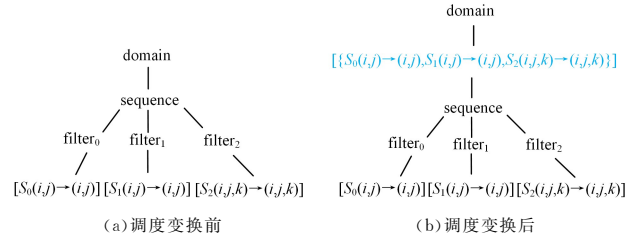

图 4 调度树变换前后的示意图

Fig. 4 Schematic of the scheduling tree before and after scheduling transformation

3.2 精度优化

精度优化子模块主要对预处理子模块的调度树经过迭代空间划分、数据流分析和调度树转操作，使其从一个调度树变成一个混合精度优化之后的目标调度树。

3.2.1 迭代空间划分

迭代空间划分是通过迭代空间划分算法对最外层 band 结点对应的循环层的迭代空间进行划分，将原本的一个迭代空间分割成两个相互独立的迭代空间，然后在这两个不同的迭代空间进行不同精度的计算。本文在基础混合精度计算的基础上，对最外层的循环进行切分，从而充分发挥程序的并行性和数据局部性，尽可能地提升程序的性能。因为迭代空间是由仿射约束的形式存储，所以本文采用的迭代空间划分算法是基于仿射约束表达式的。该算法的过程如算法 1 所示。

算法 1 迭代空间划分算法

输入：rate(percentage of partitioning)，band node

输出：band_node_list

1. Initialize two new band nodes，band_node1 and band_node2.
2. For each domain D of the band node：
3. Create two new iteration domains，iter _ domain1 and iter_domain2.
4. For each statement S in domain D：
5. Create a new empty statement S_1 in iter_domain1 and a new empty statement S_2 in iter_domain2.
6. For each pair of affine constraints(C_1，C_2) in statement S：
7. If the pair is from the outermost loop(index=0)：
8. Calculate the rangeval=floor((C_2-C_1) * rate / 100).
9. Compute the new affine constraint pairs for iter_domain1 and iter_domain2 based onval and the original constraints C_1，C_2.
10. Add the new constraint pairs to statements S_1 and S_2 respectively.
11. Else：
12. Add the original pair of affine constraints(C_1，C_2) to both S_1 and S_2.
13. Add iter_domain1 to band_node1 and iter_domain2 to band_node2.
14. Combine band_node1 and band_node2 intoband_node_list.
15. Return band_node_list.

首先，从最外层循环中提取循环的上下界 C_1 和 C_2。接着，根据设定的混合比例参数 *rate* 的数值，用式(6)计算切分的中间值 *val*。

$$val=\left\lfloor (C_2-C_1)*\frac{rate}{100}\right\rfloor \tag{6}$$

该算法为帮助用户快速准确地找到最合适的混合比例 rate 的值，采用了将最外层循环进行一百等分的策略。有效缩小 rate 的搜索空间，使用户快速确定最佳的划分比例。rate 代表了高精度计算的迭代空间占基础混合精度计算迭代空间大小的百分比。

以基础混合精度矩阵乘计算为例，经过迭代空间划分后的调度树如图 5 所示，当 $C_1=0$，$C_2=99$，$rate=50$ 时，$val=49$，划分后的两个迭代空间为图 5 中蓝色部分。

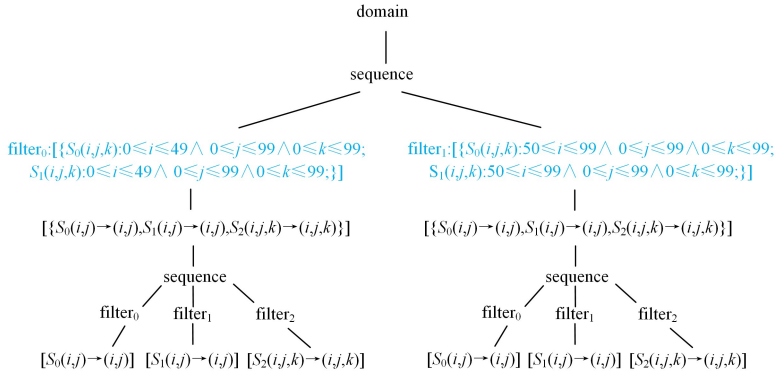

图 5 迭代空间划分之后的调度树

Fig. 5 Scheduling tree after iteration space partitioning

需要说明的是，对于矩阵乘计算来说，每层循环之间都没有依赖，因此该迭代空间划分算法对最外层循环进行划分不会破坏程序内原有的依赖关系，不会改变计算结果，并且可以保证精度转换开销的最小化。

3.2.2 数据流分析

数据流分析是一种编译优化技术，主要针对程序中的数组引用和语句进行了分析。经过分析访存映射、数组引用等情况，将对访存映射和数组引用等进行合适的分组和计算。这个过程的目的是获取关于低精度数组以及混合精度计算所需的强制类型转换流映射关系的信息，以便更好地支持混合精度计算的实施。本文中在第一个迭代空间中进行高精度计算，在第二迭代空间中进行基础混合精度计算，这样计算可以使混合精度矩阵乘计算中误差最小。

采用下面算法 2 对所有数组进行处理并对程序段中的数组引用进行分组。

算法 2 数组引用分组算法

输入：array，band_node_list

输出：local_array

1. retrieve the scheduling and additional information from the band_node_list.
2. For each array of arrays do：
3. initialize the groups of array references.
4. For each any two groups of array do：
5. For each any access relation of groups do：
6. if the access ralations is overlapping then：
7. if one of access is write then：
8. combine these two groups into one group.
9. end
10. end
11. end
12. end
13. Partition the array reference groups based on the segmented iteration domain，and then replicate the array intolocal_array.
14. For each any two groups of array do：
15. For each any access relation of groups do：
16. if the access ralations is no overlapping then：
17. If both access operations are reads：
18. combine these two groups into one group.
19. end
20. If one access operation is a write and the other is a read：
21. combine these two groups into one group.
22. end
23. end
24. end
25. end
26. calculates index offsets and other information while perfor-

ming a series of checks.
27. update the information of thelocal_array.
28. end
29. Return local_array.

首先，对程序中的数组引用和语句实例进行分析，根据访存映射、数组引用等情况对它们进行分组。首先，需要将一个数组的引用以及与之相关的信息初始化为一个组。然而，若两个组的引用访问了相同的数组元素，并且其中至少一个引用是写入操作（即存在依赖），那么个引用需要被放在同一组内共同考虑。这样做是为了确保数据的一致性，防止由于只考虑其中一个引用而导致数据不一致的情况发生。因此，在遇到这种情况时，我们会合并这两个组，以确保数据的正确性。如果两个组之间不存在依赖关系，即它们的访问关系都只是读取操作，那么我们将这两个组合并为一组。但如果两个组的访问关系中一个组是写入操作，另一个组是读取操作，那么也会将这两个组合并为一组。接着，对引用数组进行遍历、判断和合并操作，并根据合并结果计算索引偏移量等信息。最后，在更新本地数组副本中的信息时，本地数组被引用部分的大小可以通过数组引用分组信息得到，并将本地数组被引用大小与预处理阶段提取的上下文等信息结合，以更新和简化本地数组中的大小和索引偏移量等信息。

以基础混合精度矩阵乘计算为例，经过数据流分析之后的调度树如图 6 所示。

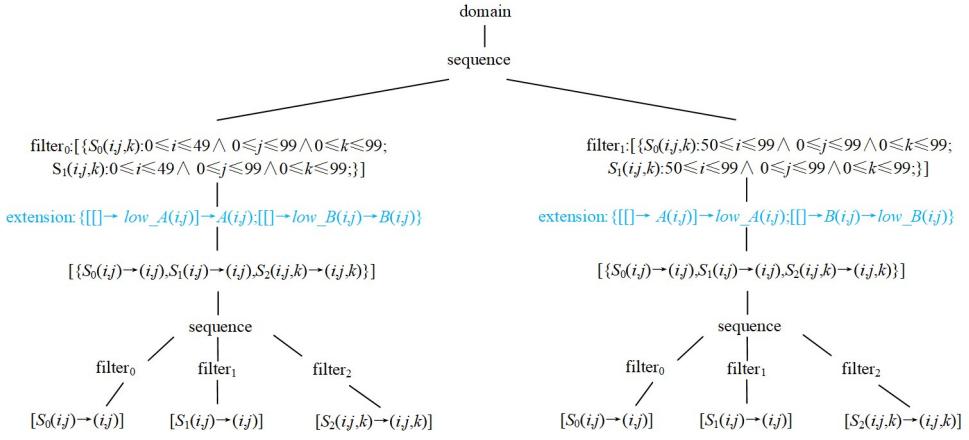

图 6　数据流分析之后的调度树

Fig. 6　Scheduling tree after data flow analysis

3.2.3　调度树转换

调度树转换主要是在数据流分析之后，根据数据流分析的结果，对原始调度树中的结点进行各种操作，将其转换成符合混合精度计算的新的调度树。

以基础混合精度矩阵乘计算为例，其转换之后的调度树如图 7 所示。

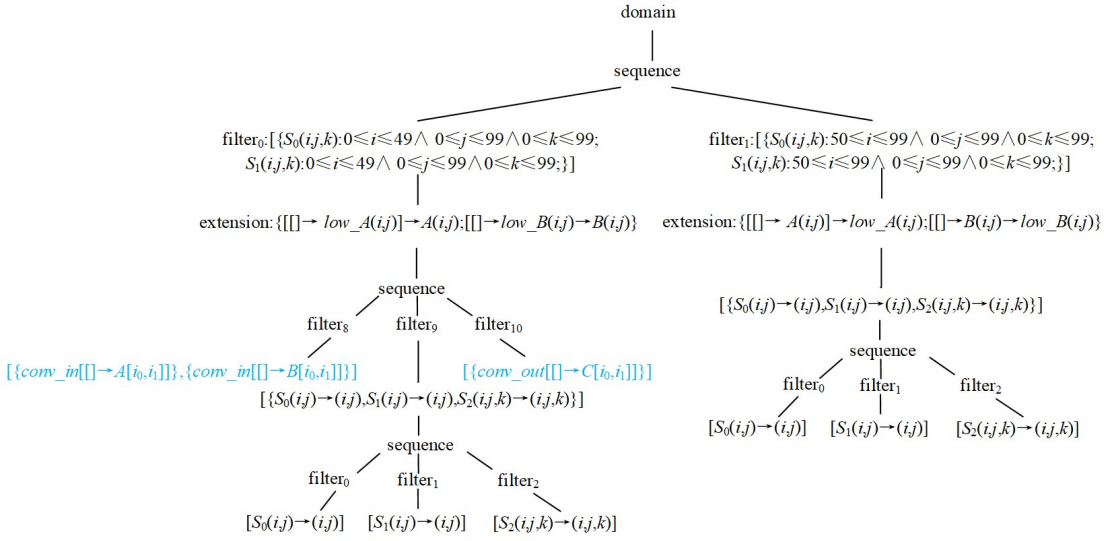

图 7　转换之后的调度树

Fig. 7　Scheduling tree after conversion

3.2.4　代码生成

代码生成的流程是“调度树→AST→C 语言代码”，在调度树转换后会得到混合精度优化后的调度树，将经过精度优化后的调度树作为输入，利用多面体扫描技术[15]生成与调度树相对应的抽象语法树（AST），然后按照 C 语言规范使用 isl 库实现从 AST 到最终代码的生成代码。最后，通过基础编译器进行编译链接，生成可执行文件。

本文为了生成混合精度代码，需要生成混合精度优化等特殊情况下的强制类型转换语句，因此对 PPCG 的代码生成模块进行了扩展和改进。虽然 PPCG 的代码生成模块已经非常细致和完善，但这些改进仍然是对其进行了一些补充。在实现过程中，我们并未赘述代码生成具体实现的细节。生成

的混合精度矩阵乘计算的代码如图 8 所示。

```
float low_A[1024][1024];
float low_B[1024][1024];
// low
// amp_kernel
{
    for (int c0 = 0; c0 <= 511; c0 += 1)
      for (int c1 = 0; c1 <= 1023; c1 += 1)
        for (int c2 = 0; c2 <= 1023; c2 += 1)
          C[c0][c1] = C[c0][c1] + A[c0][c2] * B[c2][c1];
  for (int c0 = 512; c0 <= 1023; c0 += 1)
    for (int c1 = 0; c1 <= 1023; c1 += 1)
      low_A[c0][c1] = (float)A[c0][c1];
  for (int c0 = 0; c0 <= 1023; c0 += 1)
    for (int c1 = 0; c1 <= 1023; c1 += 1)
      low_B[c0][c1] = (float)B[c0][c1];
  for (int c0 = 512; c0 <= 1023; c0 += 1)
    for (int c1 = 0; c1 <= 1023; c1 += 1)
      for (int c2 = 0; c2 <= 1023; c2 += 1)
        C[c0][c1] = (C[c0][c1] + (low_A[c0][c2] * low_B[c2][c1]));
}
```

图 8 矩阵乘混合精度计算代码核心段

Fig. 8 Code core segment of matrix multiplication mixed precision calculation

代码中首先根据计算出来的本地数组大小声明了低精度的数组，然后进行了高精度计算，之后将一定比例的高精度的输入矩阵 ***A*** 和 ***B*** 强制类型转换为低精度的数组，最后将剩下的比例用低精度乘高精度加基础混合精度计算。

4 实验及分析

为了验证 AMAO 工具的有效性，在多个矩阵规模下将工具生成的混合精度代码与矩阵乘优化工具 AGMMMPC、国际主流的精度优化工具 TAFFO[7] 和国内主流的混合精度工具 AMPO-SC[26] 进行对比测试，以准确地评估其性能和误差。

4.1 实验平台及测试用例

在 128 * 128 * 128 到 1024 * 1024 * 1024 之间选取了 13 个规模矩阵乘法程序作为测试用例。实验平台信息如表 1 所列。

表 1 测试平台信息

Table 1 Test platform information

type	X86 platform
CPU	Intel(R) Core(TM) i7-9700K CPU
Compiler	GCC 9.4.0
ICC	-O0
PPCG	0.08.1
PPCG	--target=c
OS	Ubuntu 20.04.4 LTS
TAFFO	TAFFO V0.3.0

4.2 性能测试

为准确测试生成的矩阵乘混合精度程序的性能，以高精度矩阵乘计算为基准，计算每个矩阵规模下不同比例的矩阵乘混合精度程序的加速比。加速比计算方法如式(7)所示：

$$speedup = T_{before} / T_{after} \tag{7}$$

其中，T_{before} 代表加速前的执行时间，T_{after} 代表加速后的执行时间。同时，为了更准确地测试生成的矩阵乘混合精度程序的性能，将自动混合精度计算工具(下文简称为 AMAO)与矩阵乘优化工具 AGMMMPC、国际主流的精度优化工具 TAFFO[7] 和国内主流的混合精度工具 AMPO-SC[26] 进行对比测试。

为了保证性能测试的稳定性，采用以下性能测试方法：获取 10 次程序运行时间，对 10 次程序执行时间求平均值。只要有一次执行时间超过平均值的 5%，就重新获取 10 次程序运行时间，获取新的平均值，重复判断，判断 5 次后，如果结果仍不符合条件，则将去掉这组数据中高于平均值 5%的值，对其余执行时间求取平均值。

所有测试规模的性能表现如图 9 所示。从图 9 中可以看到，与高精度相比，AMAO 性能加速比平均提升了 13.83%；与 AGMMMPC 工具相比，AMAO 的性能加速比平均降低了 0.55%；与 AMPO-SC 相比，性能加速比平均提升了 1.14%；与 TAFFO 相比，性能加速比平均提升了 4.63%。

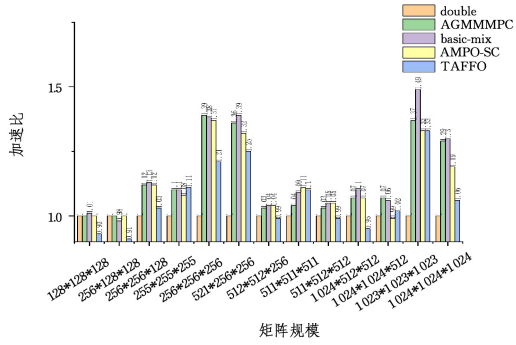

图 9 不同测试规模的加速比

Fig. 9 Acceleration ratios for different test sizes

为了准确评估 AMAO 工具的加速效果，观察各个矩阵规模下不同比例代码的性能变化趋势，在每个矩阵规模下设置 10 个混合比例生成代码，并与 AGMMMPC 工具进行对比测试。其中，mix 是 AMAO 工具生成程序的加速比，change 是 AGMMMPC 工具生成程序的加速比。128 * 128 * 128 规模的矩阵性能表现如图 10 所示，256 * 128 * 128 规模的矩阵性能表现如图 11 所示。相较于 AGMMMPC，AMAO 在 128 * 128 * 128 规模下加速比提升了 0.5%，在 256 * 128 * 128 规模下性能下降了 1.1%。

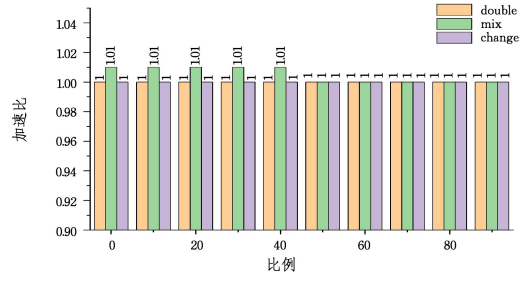

图 10 128 * 128 * 128 规模的矩阵性能表现

Fig. 10 Matrix performance at the scale of 128 * 128 * 128

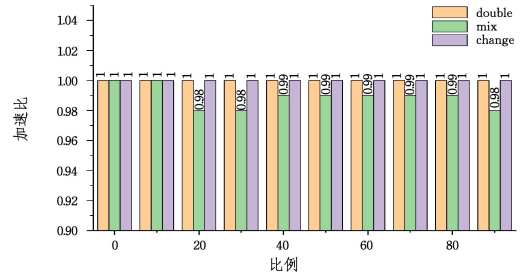

图 11 256 * 128 * 128 规模的矩阵性能表现

Fig. 11 Matrix performance at the scale of 256 * 128 * 128 scale

256 * 256 * 128 规模的矩阵性能表现如图 12 所示，256 *

256 * 256 规模的矩阵性能表现如图 13 所示。相较于 AGMMMPC,AMAO 在 256 * 256 * 128 规模下加速比下降了 2.75%,在 256 * 256 * 256 规模下性能下降了 15.2%。

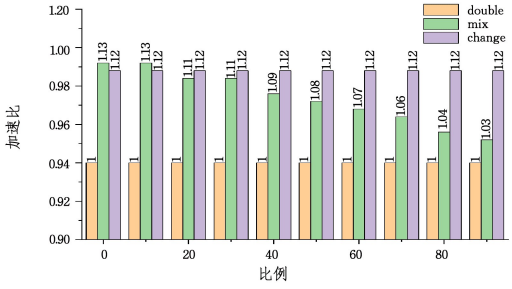

图 12 256 * 256 * 128 规模的矩阵性能表现

Fig. 12 Matrix performance at the scale of 256 * 256 * 128

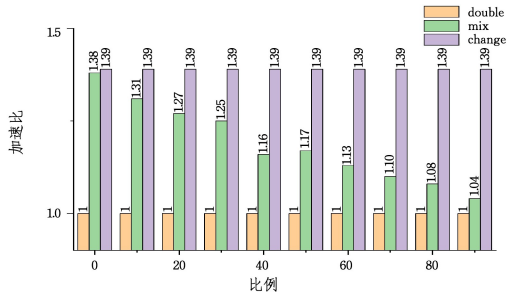

图 13 256 * 256 * 256 规模的矩阵性能表现

Fig. 13 Matrix performance at the scale of 256 * 256 * 256

512 * 256 * 256 规模的矩阵性能表现如图 14 所示,512 * 512 * 256 规模的矩阵性能表现如图 15 所示。相较于 AGMMMPC,AMAO 在 512 * 256 * 256 规模下加速比下降了 14.7%,在 512 * 512 * 256 规模下性能下降了 2%。

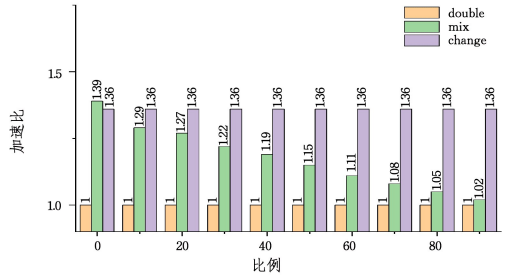

图 14 512 * 256 * 256 规模的矩阵性能表现

Fig. 14 Matrix performance at the scale of 512 * 256 * 256

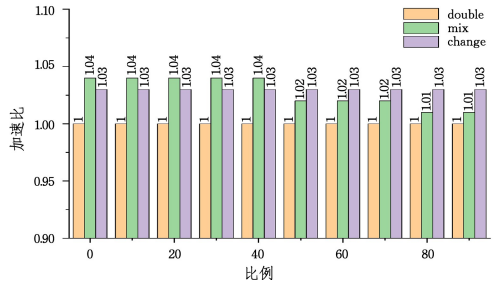

图 15 512 * 512 * 256 规模的矩阵性能表现

Fig. 15 Matrix performance at the scale of 512 * 512 * 256

512 * 512 * 512 规模的矩阵性能表现如图 16 所示,1024 * 512 * 512 规模的矩阵性能表现如图 17 所示。相较于 AGMMMPC,AMAO 在 512 * 512 * 512 规模下加速比下降了 2.08%,在 1024 * 512 * 512 规模下性能下降了5.6%。

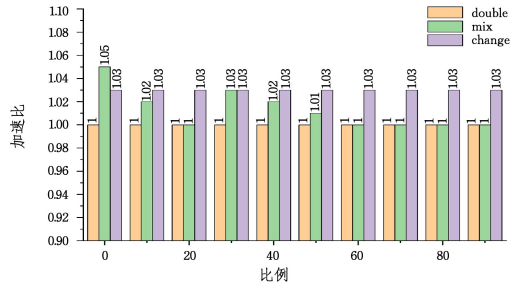

图 16 512 * 512 * 512 规模的矩阵性能表现

Fig. 16 Matrix performance the scale of 512 * 512 * 512

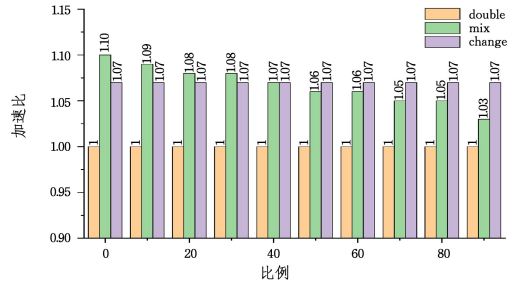

图 17 1024 * 512 * 512 规模的矩阵性能表现

Fig. 17 Matrix performance at the scale of 1024 * 512 * 512

1024 * 1024 * 512 规模的矩阵性能表现如图 18 所示,1024 * 1024 * 1024 规模的矩阵性能表现如图 19 所示。相较于 AGMMMPC,AMAO 在 1024 * 1024 * 512 规模下加速比下降了 5.87%,在 1024 * 1024 * 1024 规模下性能下降了 10.5%。

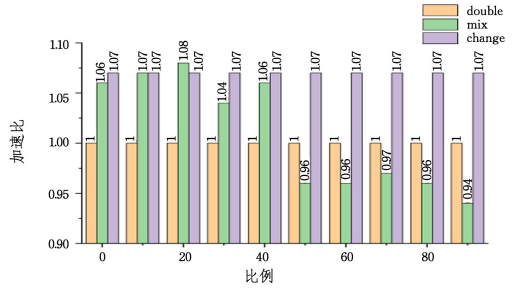

图 18 1024 * 1024 * 512 规模的矩阵性能表现

Fig. 18 Matrix performance at the scale of 1024 * 1024 * 512

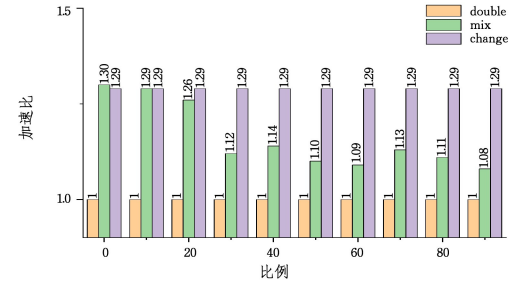

图 19 1024 * 1024 * 1024 规模的矩阵性能表现

Fig. 19 Matrix performance at the scale of 1024 * 1024 * 1024

255 * 255 * 255 规模的矩阵性能表现如图 20 所示,511 * 511 * 511 规模的矩阵性能表现如图 21 所示,1023 * 1023 * 1023 规模的矩阵性能表现如图 22 所示,相较于 AGMMMPC,AMAO 在 255 * 255 * 255 规模下加速比下降了 5.88%,在 511 * 511 * 511 规模下性能下降了 0.55%,在 1023 * 1023 * 1023 规模下性能下降了 11%。

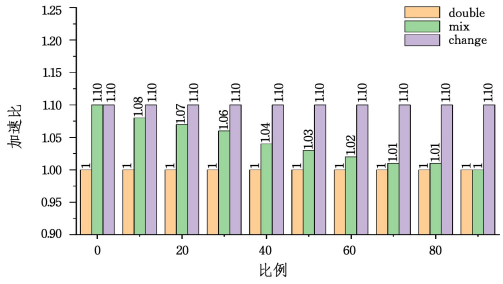

图 20 255 * 255 * 255 规模的矩阵性能表现

Fig. 20 Matrix performance at the scale of 255 * 255 * 255

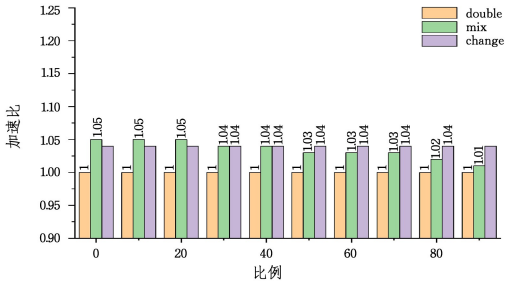

图 21 511 * 511 * 511 规模的矩阵性能表现

Fig. 21 Matrix performance the scale of 511 * 511 * 511

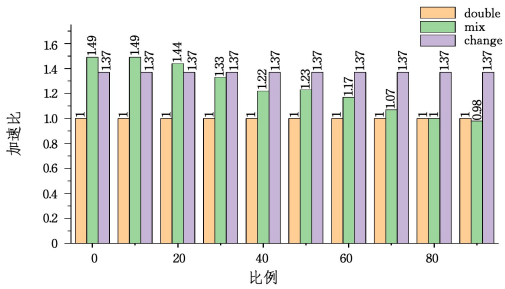

图 22 1 023 * 1 023 * 1 023 规模的矩阵性能表现

Fig. 22 Matrix performance at the scale of 1 023 * 1 023 * 1 023

对所有性能测试结果进行分析可得到，随着高精度计算比例的增加，性能呈现下降的趋势。与 AGMMMPC 工具相比，在测试的 13 个矩阵规模的 10 个比例下，AMAO 工具生成的代码性能加速比平均降低了 5.90%，与基础混合精度计算相比，性能加速比平均降低了 6.45%。

4.3 精度测试

为了评估 AMAO 在基础混合精度计算的基础上能否提升精度，以及其相对于 AGMMMPC 的精度表现，在多个矩阵规模下进行了精度对比测试。同时，为了评估 AMAO 的精度表现，还与 AMPO-SC 和 TAFFO 进行对比测试。在对生成的混合精度代码进行误差测试时，采用平均误差来刻画不同结果的可靠程度，并在 128 * 128 * 128 到 1024 * 1024 * 1024 之间选取了 4 个规模的矩阵乘法程序作为测试用例，以高精度矩阵乘计算为基准。每次测试得到的误差可能会因为计算过程中的舍入误差和数值不稳定性而有所不同。为了选择最大的误差进行评估，记录每次测试的误差，并选择其中的最大值作为评估指标。平均误差的计算方法是矩阵中所有绝对误差总和除以矩阵规模。测试每个矩阵规模下，10 个不同比例生成的程序的平均误差，同时以与矩阵乘优化工具 AGMMMPC 生成的程序为对比对象。在 128 * 128 * 128 矩阵规模下的精度测试结果如图 23 所示。

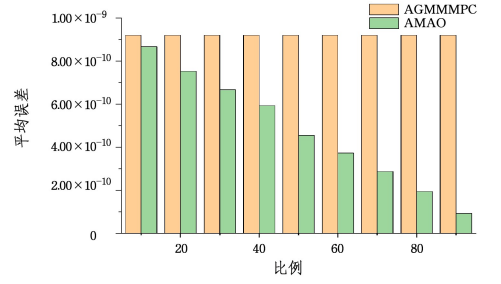

图 23 128 * 128 * 128 精度测试结果

Fig. 23 Accuracy test results at the scale of 128 * 128 * 128

在 256 * 256 * 256 矩阵规模下的精度测试结果如图 24 所示。

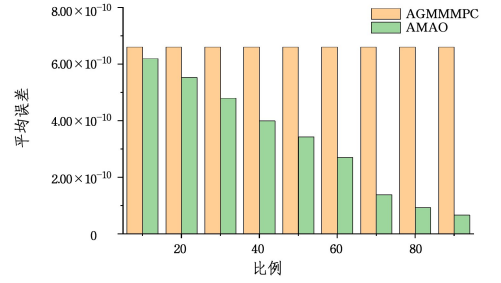

图 24 256 * 256 * 256 精度测试结果

Fig. 24 Accuracy test results at the scale of 256 * 256 * 256

在 512 * 512 * 512 矩阵规模下的精度测试结果如图 25 所示。

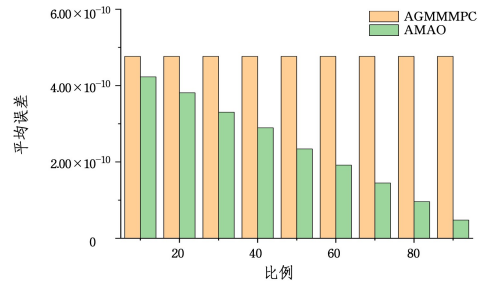

图 25 512 * 512 * 512 精度测试结果

Fig. 25 Accuracy test results at the scale of 512 * 512 * 512

在 1 024 * 1 024 * 1 024 矩阵规模下的精度测试结果如图 26 所示。

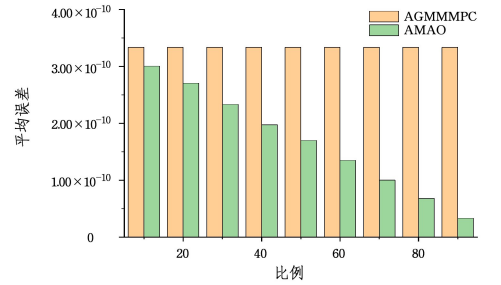

图 26 1 024 * 1 024 * 1 024 精度测试结果

Fig. 26 Accuracy test results at the scale of 1 024 * 1 024 * 1 024

根据以上精度测试结果分析可得到，随着高精度计算比例的增加，AMAO 工具生成代码的平均误差不断减小。经验证，其他规模下也呈现出同样的趋势。与 AGMMMPC 工具相比，在 128 * 128 * 128 到 1 024 * 1 024 * 1 024 之间的 13 个

矩阵规模下，AMAO 工具生成的代码精度平均提升了 49.31%；与基础混合精度相比，AMAO 工具生成的代码精度平均提升了 50.78%。为了更准确地评估 AMAO 工具的精度，以 AGMMMPC，AMPO-SC 和 TAFFO 这 3 个工具为对比对象，选取 13 个矩阵规模下性能最好时的代码进行精度测试，测试每个矩阵规模下引入的平均误差。对比测试结果如表 2 所列。

从表 2 可以看出，AMAO 工具的平均误差小于 AGMMMPC，且远小于 AMPO-SC 和 TAFFO 的平均误差误差，也即 AMAO 工具在矩阵乘计算时具有更高的精确度。

表 2　精度对比测试

Table 2　Accuracy comparison test

Matrix size	Basic-mix	AMAO	AGMMMPC	AMPO-SC	TAFFO
128 * 128 * 128	9.35×10^{-10}	6.668×10^{-10}	9.201×10^{-10}	2.36×10^{-1}	2.41×10^{-4}
256 * 128 * 128	9.61×10^{-10}	8.776×10^{-10}	9.201×10^{-10}	2.36×10^{-1}	2.42×10^{-4}
256 * 256 * 128	9.74×10^{-10}	8.411×10^{-10}	9.581×10^{-10}	2.23×10^{-1}	2.43×10^{-4}
255 * 255 * 255	6.77×10^{-10}	5.606×10^{-10}	6.628×10^{-10}	2.25×10^{-1}	4.87×10^{-4}
256 * 256 * 256	6.72×10^{-10}	6.191×10^{-10}	6.60×10^{-10}	2.23×10^{-1}	4.87×10^{-4}
512 * 256 * 256	6.82×10^{-10}	5.936×10^{-10}	6.661×10^{-10}	2.24×10^{-1}	4.88×10^{-4}
512 * 512 * 256	4.80×10^{-10}	5.961×10^{-10}	6.770×10^{-10}	2.12×10^{-1}	4.88×10^{-4}
511 * 511 * 511	4.83×10^{-10}	4.228×10^{-10}	4.759×10^{-10}	2.37×10^{-1}	9.76×10^{-4}
512 * 512 * 512	4.80×10^{-10}	4.228×10^{-10}	4.767×10^{-10}	2.24×10^{-1}	9.76×10^{-4}
1024 * 512 * 512	4.83×10^{-10}	4.217×10^{-10}	4.781×10^{-10}	2.37×10^{-1}	9.76×10^{-4}
1024 * 1024 * 512	4.79×10^{-10}	4.290×10^{-10}	4.755×10^{-10}	1.88×10^{-1}	9.76×10^{-4}
1023 * 1023 * 1023	4.85×10^{-10}	3.360×10^{-10}	4.805×10^{-10}	2.13×10^{-1}	9.56×10^{-4}
1024 * 1024 * 1024	3.36×10^{-10}	3.006×10^{-10}	3.336×10^{-10}	2.37×10^{-1}	1.95×10^{-3}

综合性能测试和精度测试可得出，本文实现的工具 AMAO 相较于工具 AGMMMPC，性能平均下降了 5.90%，但是其精度平均提升了 49.31%。相较于 AMPO-SC 工具和 TAFFO 工具，AMAO 工具性能和精度均具有优势。

结束语　本文在前期工作生成的基础混合精度矩阵乘代码的基础上，为了进一步降低基础混合精度矩阵乘计算引入的误差，提出了混合精度矩阵乘代码自动生成工具 AMAO。该工具在基础混合精度计算的基础上，经过迭代空间划分算法，将一个迭代空间划分成两个迭代空间，将原本的基础混合精度计算根据比例拆分成两种计算，一部分用高精度计算，一部分用基础混合精度计算，并根据迭代空间划分算法自动实现混合精度优化工具。通过对矩阵乘法计算用例进行测试，并与其他工具进行比较实验，验证了该工具的有效性。本文虽然实现了混合精度矩阵乘代码的自动生成，但是仍有很多优化的地方。下一步需要进行的工作有：1）根据不同比例生成对应的混合精度代码之后，对其性能、误差提升和损耗进行建模，通过对该模型的计算得出在误差阈值下性能最佳的混合比例；2）可以对其进行优化，以在 GPU 等异构架构上进行扩展，从而更好地适配主流异构架构并提升其通用性。

参 考 文 献

[1] MICIKEVICIUS，PAULIUS，et al. Mixed precision training for deep neural networks[J]. arXiv：1710.03740，2017.

[2] CHERUBIN S，AGOSTA G. Tools for reduced precision computation：a survey[J]. ACM Computing Surveys，2021，53(2)：1-35.

[3] KOTIPALLI PV，SINGH R，WOOD P，et al. AMPT-GA：Automatic mixed precision floating point tuning for GPU applications[C]// Proceedings of the ACM International Conference on Supercomputing(ICS'19). Phoen ix Arizona：ACM，2019：160-170.

[4] SOLOVYEV A，JACOBSEN C，RAKAMARIĆZ，et al. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions[C]// FM 2015：Formal Methods. Cham：Springer International Publishing，2015，9109：532-550.

[5] DARULOVA E，HORN E，SHARMA S. Sound mixed-precision optimization with rewriting[C]// 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems(ICCPS). IEEE，2018：208-219.

[6] DARULOVA E，KUNCAK V. Towards a compiler for reals[J]. ACM Transactions on Programming Languages and Systems(TOPLAS)，2017，39(2)：1-28.

[7] CHERUBIN S，CATTANEO D，CHIARI M，et al. TAFFO：Tuning assistant for floating to fixed point optimization[J]. IEEE Embedded Systems Letters，2019，12(1)：5-8.

[8] KUM K I，KANG J Y，SUNG W Y. Autoscaler for C：an Optimizing floating-point to integer C program converter for fixed-point digital signal processors[J]. IEEE Transactions on Circuits and Systems II：Analog and Digital Signa l Processing，2000，47(9)：840-848.

[9] RUBIO-GONZÀLEZ C，NGUYEN C，MEHNE B，et al. Floating-point precision tuning using blame analysis[C]// 2016 IEEE/ACM 38th International Conference on Software Engineering(ICSE). IEEE，2016：1074-1085.

[10] MENON H，LAM M O，OSEI-KUFFUOR D，et al. ADAPT：Algorithmic differentiation applied to floatingpoint precision tuning[C]// SC18：International Conference for High Performance Computing，Networking，Storage and Analysis. IEEE，2018：614-626.

[11] HO N M，MANOGARAN E，WONG W F，et al. Efficient floating point precision tuning for approximate computing[C]// 2017 22nd Asia and South Pacific Design Automation Conference(ASP-DAC). IEEE，2017：63-68.

[12] LAGUNA I，WOOD P C，SINGH R，et al. Gpumixer：Performance-driven floating-point tuning for gpu scientific applications[C]// International Conference on High Performance Computing. Cham：Springer，2019：227-246.

[13] MICIKEVICIUS P，NARANG S，ALBEN J，et al. Mixed Preci-

sion Training[J]. arXiv:1710.03740,2018.
[14] FEAUTRIER P,LENGAUER C. Polyhedron model[C]// Proc. of the Encyclopedia of Parallel Computing. 2011:1581-1592.
[15] GROSSER T,VERDOOLAEGE S,COHEN A. Polyhedral AST generation is more than scanning polyhedra[J]. ACM Trans. on Programming Languages and Systems(TOPLAS),2015,37(4):12:1-12:50.
[16] ZHAO J,LI Y Y,ZHAO R C. "Black magic" of polyhedral compilation[J]. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8):2371-2396.
[17] BONDHUGULA U,HARTONO A,RAMANUJAM J,et al. A practical automatic polyhedral parallelizer and locality optimizer [C]// Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM Press, 2008:101-113.
[18] VERDOOLAEGE S,CARLOS JUEGA J,COHEN A,et al. Polyhedral parallel code generation for CUDA[J]. ACM Trans. on Architecture and Code Optimization(TACO),2013,9(4):54:1-54:24.
[19] TRIFUNOVIĆ K,COHEN A,EDELSOHN D,et al. Graphite two years a fter:First lessons learned from real-world polyhedral compilation[C]// Proc. of the 2nd GCC Research Opportunities Workshop(GROW). 2010.
[20] GROSSER T,GROESSLINGER A,LENGAUER C. Polly-Performing polyhedral optimizations on a low-level intermediate representation[J]. Parallel Processing Letters, 2012, 22(4):1250010.
[21] KELLY W,PUGH W. A unifying framework for iteration reordering transformations[C]// Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing. Brisbane,Qld,Australia:IEEE,1995,1:153-162.
[22] GIRBAL S,VASILACHE N,BASTOUL C,et al. Semi-Automatic composition of loop transformations for deep parallelism and memory hierarchies[J]. Int'l Journal of Parallel Programming(IJPP),2006,34(3):261-317.
[23] VERDOOLAEGE S. Counting affine calculator and applications [C]// Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques(IMPACT'11). 2011.
[24] VERDOOLAEGE S,GROSSER T. Polyhedral extraction tool [C]// Proc. of the 2nd Int'l Workshop on Polyhedra l Compilation Techniques(IMPACT). 2012.
[25] VERDOOLAEGE S. isl:An integer set library for the polyhedral model[C]// Proc. of the ICMS 2010. Berlin, Heidelberg: Springer-Verlag,2010:299-302.
[26] SONG G H,GUO S Z,ZHAO J,et al. Automatic hybrid accuracy optimization for Stencil computing[J]. Journal of Software, 2023,34(12):5704-5723.

HE Haotian, born in 1997, postgraduate. His main research interest is high-performance computing.

XU Jinchen, born in 1987, Ph.D, associate professor. His main research interest is high-performance computing.