计算机科学
COMPUTER SCIENCE

基于多面体模型的矩阵乘法向量代码生成

王博溙, 庞建民, 徐金龙, 赵捷, 陶小涵, 朱雨

引用本文

王博溙, 庞建民, 徐金龙, 赵捷, 陶小涵, 朱雨. 基于多面体模型的矩阵乘法向量代码生成[J]. 计算机科学, 2022, 49(10): 44-51.

WANG Bo-yang, PANG Jian-min, XU Jin-long, ZHAO Jie, TAO Xiao-han, ZHU Yu. Matrix Multiplication Vector Code Generation Based on Polyhedron Model[J]. Computer Science, 2022, 49(10): 44-51.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于 GPU 加速的并行 WMD 算法

Parallel WMD Algorithm Based on GPU Acceleration

计算机科学, 2021, 48(12): 24-28. https://doi.org/10.11896/jsjkx.210600213

基于数据重用分析的多面体循环合并策略

Loop Fusion Strategy Based on Data Reuse Analysis in Polyhedral Compilation

计算机科学, 2021, 48(12): 49-58. https://doi.org/10.11896/jsjkx.210200071

适用于线性网络编码关键路径的实时性算法

Novel Real-time Algorithm for Critical Path of Linear Network Coding

计算机科学, 2020, 47(9): 232-237. https://doi.org/10.11896/jsjkx.190800023

CompCert 编译器目标代码生成机制分析

Analysis of Target Code Generation Mechanism of CompCert Compiler

计算机科学, 2020, 47(9): 17-23. https://doi.org/10.11896/jsjkx.200400018

基于 AADL 的自主无人系统可成长框架

Growth Framework of Autonomous Unmanned Systems Based on AADL

计算机科学, 2020, 47(12): 87-92. https://doi.org/10.11896/jsjkx.201100173

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210800247

基于多面体模型的矩阵乘法向量代码生成

王博漾[1]　庞建民[1,2]　徐金龙[2]　赵　捷[2]　陶小涵[2]　朱　雨[2]
1 郑州大学网络空间安全学院　郑州 450000
2 数学工程与先进计算国家重点实验室(信息工程大学)　郑州 450000
(w_boyang1997@163.com)

摘　要　矩阵乘法是众多科学计算的核心,而向量化编程是提升其性能的主要手段之一。针对现有的向量化优化往往存在需要手工进行调优以及与硬件结构映射的问题,基于多面体编译器PPCG,在多面体模型中引入向量代码生成框架,提出了基于多面体模型的矩阵乘法向量代码生成框架。通过对矩阵乘法的向量化方案进行收益分析来确定向量化方案,指导应用框架的代码生成,基于该代码生成框架,有利于矩阵乘法的向量化快速优化。选取13个规模在64×64×64到1024×1024×1024之间的矩阵乘法用例进行实验,结果表明,该框架能够正确生成向量化代码,与基础编译器ICC的自动向量化功能相比,应用该框架生成的向量化代码最高获得了5.09倍的加速和3.39倍的平均加速。

关键词:矩阵乘法;多面体模型;向量化;调度变换;代码生成
中图法分类号　TP312

Matrix Multiplication Vector Code Generation Based on Polyhedron Model

WANG Bo-yang[1],PANG Jian-min[1,2],XU Jin-long[2],ZHAO Jie[2],TAO Xiao-han[2] and ZHU Yu[2]
1 School of Cyber Science and Engineering,Zhengzhou University,Zhengzhou 450000,China
2 State Key Laboratory of Mathematical Engineering and Advanced Computing,PLA Information Engineering University,Zhengzhou 450000,China

Abstract　Matrix multiplication is the core of many scientific calculations,and vectorized programming is one of the main means to improve its performance. In view of the existing vectorization optimization problems that often require manual tuning and need to be mapped to the hardware structure,based on the polyhedron compiler PPCG,a vector code generation framework is introduced into the polyhedron model,and a matrix multiplication vector code generation framework based on the polyhedron model is proposed. Through the profit analysis of the matrix multiplication vectorization program,the vectorization program is determined,and the code generation of the application framework is guided. Based on this framework,it is conducive to the rapid optimization of vectorization of matrix multiplication. Selecting 13 matrix multiplication cases with a scale between 64×64×64 and 1024×1024×1024 for experiments. The results show that the framework can generate vectorized code correctly. Compared with the automatic vectorization of the basic compiler ICC,the vectorized code generated by the framework has a speedup of 5.09 times and an average speedup of 3.39 times.

Keywords　Matrix multiplication,Polyhedron model,Vectorized,Scheduling transformation,Code generation

1　引言

矩阵乘法作为众多科学计算程序的核心,在图像处理以及深度学习领域占据着重要的地位,其代码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法的执行效率变得尤为关键。针对矩阵乘法对计算资源消耗较大的特点,在软件优化方面有了许多研究工作[1-2]。其中,向量化技术作为主要的优化手段被广泛应用。当前主流的基础编译器都具有较完善的向量化功能[3],并且进行了不断改进。然而,随着硬件体系结构的不断发展,向量寄存器的长度也随之发生了改变。为了最大限度地发挥处理器的性能,通常需要程序员手工进行调优。这不仅要求程序员对硬件体系结构和程序数据结构有较深的了解,同时也会产生较大的时间成本。因此,如何利用编译器自动生成高效的向量化代码成为了待解决的重要问题。

矩阵乘法是典型的循环程序,循环优化是提升其程序性能的主要途径之一。多面体模型[4]的出现为矩阵乘法优化提供了全新的研究思路。多面体模型最关注的就是循环优化,

到稿日期:2021-08-27　返修日期:2022-06-02
基金项目:国家自然科学基金区域创新发展联合基金(U20A20226)
This work was supported by the National Natural Science Foundation Regional Innovation and Development Joint Fund(U20A20226).
通信作者:庞建民(jianmin_pang@126.com)

它改变了传统的程序优化手段，基于数学线性规划的思想，通过对输入的源程序进行提取与分析，挖掘程序的并行性和数据局部性，从而提升代码性能。当前许多主流的商用编译器以及研究性编译器皆引入了多面体编译技术，以进行更广泛的应用，包括自动并行优化、数据布局优化、内存管理优化、通信优化等。其中，PPCG[5]作为一个"源-源"的多面体编译器，支持面向特定体系结构的映射，被应用于众多领域。

目前，基于多面体模型的"源-源"优化编译器中缺少对向量代码生成的支持。例如，Pluto[6]仅可以实现向量化分析，仍需通过添加向量化标志的形式，利用基础编译器进行进一步的向量化开发；而 PPCG 还没有支持向量化的相关操作。结合多面体模型的优势，直接在多面体模型中实现自动向量化，可以更加充分地利用多面体强大的依赖分析与循环变换能力，完成程序的向量化变换。与基础编译器直接在中间表示上进行向量化相比，其可以利用基础编译器对自动生成的向量化代码进行进一步优化，程序优化空间较大。此外，与多面体模型的分块技术相结合，可以提高程序的数据局部性。

综上，本文提出了基于多面体模型的向量代码生成框架(Polyhedron Model Vectorization，PMV)，将向量代码生成功能添加到"源-源"的 PPCG 编译器中，从而生成更高效的矩阵乘法向量代码。

本文的主要贡献如下：

(1)结合多面体模型和向量化技术，设计并实现了基于多面体模型的矩阵乘法向量代码生成框架。

(2)对矩阵乘法程序进行向量化分析，从中选取较优的向量化方案，指导代码生成框架进行高性能代码生成。

(3)选取不同规模下的矩阵乘法测试用例，对生成的向量代码进行正确性验证。同时，与 ICC 等编译器进行对比，平均获得了 3.39 倍的性能提升。

本文第 2 节介绍了向量化技术以及多面体模型；第 3 节简要介绍了基于多面体模型的向量代码生成框架；第 4 节给出了针对矩阵乘法的向量化方案以及收益分析；第 5 节和第 6 节分别介绍了向量化调度变换与向量化代码生成的过程；第 7 节对应用 PMV 框架生成的程序执行效率进行了测试和分析；最后总结全文并展望未来。

2 背景及相关工作

本节介绍了向量化技术与多面体模型的相关研究与现状，同时对开发平台 PPCG 编译器进行了说明。

2.1 向量化 SIMD 技术

向量化[7]是提升程序性能的关键手段，被广泛应用于众多编译器中，它是实现数值计算可扩展性能的有效方法之一。随着体系结构的不断改变，向量化技术也在不断发展。

Intel 于 1996 年第一次在处理器中集成了 SIMD 扩展部件，此后推出了 SSE，AVX，FMA[8]等向量指令集。除此之外，ARM 最新研究出了适合其体系结构的 SVE[9]，提供了较好的可移植性。基于 SIMD 扩展部件的向量化技术成为了主要的优化技术手段。

如今，SIMD 向量化技术被广泛应用于科学计算程序中。通过编译器的自动向量化技术或调用编译器支持的 SIMD 指令集接口进行向量化优化，可大大提升程序的性能。

2.2 多面体编译技术

多面体模型[10]最关注的就是循环的优化，是发掘多重循环向量化的有效方法，它应用范围广、表现能力强、优化空间大的优势为研究人员提供了全新的思路。在多面体模型中，通常使用调度树[11]的形式对语句实例进行表示。基于调度树可以快速实现各种循环变换，同时可以根据调度树生成抽象语法树[12]，进行代码生成。基于多面体模型的优点，有学者开始考虑将向量化技术与多面体模型相结合，如 Kong 等提出了面向 ISA 的 SIMD 代码生成[13]，但未开源。除此之外，也有许多主流的基础编译器引入了多面体模块，如 GCC 的 Graphite 框架以及 LLVM 的 Polly 模块[14]。多面体模型的编译流程如图 1 所示。

本文选取基于多面体模型的 PPCG 编译器作为开发平台，它通过调度变换[15]进行程序的优化，在代码生成模块中进行特定目标平台的映射。

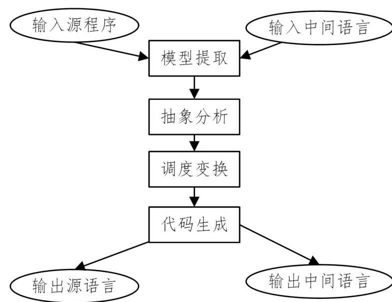

图 1 多面体模型的编译流程

Fig. 1 Polyhedron model compilation process

2.3 矩阵乘法向量化的相关研究

当前，矩阵乘法被广泛应用于计算机技术的各个领域，如何提升其程序性能变得尤为重要。向量化技术作为主要的程序优化手段，被应用于矩阵乘法优化工作中。

目前，矩阵乘法的自动向量化优化主要通过基础编译器，运用对应体系架构所支持的向量指令集进行向量指令替换，如向量化效果较好的 Intel 平台 ICC 编译器。通常情况下，编译器会对其进行向量化分析，结合循环交换技术，对矩阵乘法循环变换后的内层循环进行自动向量化。通过测试可知，其自动向量化加速比平均达到了 1.65。然而，由于向量部件的不断改变，已有的矩阵乘法自动向量化优化工作无法完全满足性能优化的需求，往往需要手工调优。

因此，关于矩阵乘法向量化的研究有很多。Liu 等[16]提出了一种高效的面向多核向量处理器的矩阵乘法向量化方法，针对体系结构的特点，设计了基于 SRAM 的优化方法。Liu 等[17]针对龙芯的体系结构特点，结合龙芯向量访存指令和乘加指令，实现了矩阵乘法的向量化运算。

对于向量化优化来说，其方案的选择与向量代码的自动生成对矩阵乘法的性能有较大的影响，因此需要利用编译器自动生成更高效的向量化代码，使其达到更好的执行效率。

3 向量代码生成框架

本文设计并实现了一个向量代码生成框架。该框架选取

前文介绍的多面体编译器 PPCG 作为开发平台，可以自动生成对应目标平台上高效的矩阵乘法 SIMD 向量代码。本文的主要工作是在 PPCG 已有的依赖分析和调度变换功能的基础上，结合矩阵乘法程序的特点，对其进行向量化分析、向量化调度变换优化以及代码生成工作。图 2 给出了向量代码生成框架，主要分为 3 个部分：

(1)PPCG 编译器从输入的 C 程序中通过静态控制块，提取其中的循环层部分用于多面体分析。同时，通过该模型对矩阵乘法进行向量化分析，选取较优的向量化方案。

(2)通过多面体模型对获取的源程序迭代空间、访存映射等信息进行分析，生成对应的原始调度树，通过 PMV 模型采用循环分段等优化手段对其循环层进行优化，完成调度变换。然后，根据调度变换后的调度树生成抽象语法树。

(3)代码生成阶段。该阶段可以细分为两个部分：1)抽象语法树表示部分，在这一部分向量代码生成模型主要完成对向量化循环层节点信息的收集，同时针对向量化循环层添加对应的向量化标志位；2)根据第一部分生成的抽象语法树，生成对应目标平台的向量代码，该模型通过结合目标平台的向量化操作指令，最终生成目标平台对应的向量化代码。

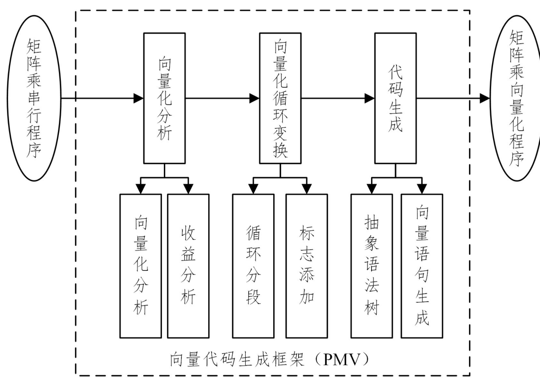

图 2 向量代码生成框架

Fig. 2 Framework of vector code generation

4 矩阵乘法向量化分析

本节主要对矩阵乘法进行向量化分析，选取较优的向量化方案指导 PMV 框架进行向量代码生成。其中，依赖分析是保证程序变换合法性的前提，对于向量化操作来说，主要与数据依赖分析有关。在多面体模型中，通过 ISL 库对输入的程序进行依赖分析，这里不再过多赘述。输入程序如图 3 所示。

```
for (i = 0; i < M; i ++)
    for (j = 0; j < N; j ++){
        c[i][j] = 0;
        for (k = 0; k < K; k ++)
            c[i][j] += a[i][k] *  b[k][j];
    }
```

图 3 矩阵乘法

Fig. 3 Matrix multiplication

4.1 向量化方案选择

从图 3 可以看出，矩阵乘法是一个典型的嵌套循环程序，有 3 层循环。其中，循环最内层包含一条计算语句，用于进行多次的乘加操作。一般情况下，针对 N 层的嵌套循环程序，一般会有多种不同的向量化方案。我们需要通过向量化分析，从中选取收益较高的向量化方案进行优化。对于矩阵乘法来说，一般会有 3 种向量化方案，分别是从最内层、次内层以及循环交换后的最内层进行向量化操作。这里以 Intel AVX 指令集为例，手工编写 3 种向量化代码进行分析，如图 4 所示。

(1)针对最内层的向量化方案

默认情况下，传统的编译器一般会针对循环的最内层进行向量化处理。一般来说，在大多数循环程序中，数组在最内层的循环索引是连续的，因此选择最内层向量化容易实现，同时还可以获得较高的收益。但是，当最内层循环的迭代次数不足或者最内层存在归约等操作时，若仍然针对最内层进行向量化操作，则会导致代价较大，甚至造成负收益。由于矩阵乘法最内层存在归约操作，在进行向量化时会把归约操作认定为循环存在潜在的依赖而不进行向量化，因此需要对最内层的归约操作进行一定处理，通过标量扩展进行消除，在此过程中会把标量替换成临时数组，但这样会导致在进行优化时增加冗余操作，造成程序效率的降低。同时，标量扩展等手段的采用，会造成不必要的内存开销。

(2)针对循环变换后最内层的向量化方案

在进行程序优化时，为了使内存访问满足局部性的原则，一般会通过循环交换来提升程序的局部性。在矩阵乘法中，通用的循环交换一般选择把循环 j 层和 k 层进行交换，把向量并行性移到最内层，再对其进行向量化处理。从图中可以看到，由于按行优先存储，在每一次迭代中，a，b，c 这 3 个数组可以满足存取连续性的要求，发生缓存未命中的可能性很小。然而，在对变换后的矩阵乘法进行向量化处理时，需要对 c 数组进行多次存取操作，这样会带来额外的内存开销，降低向量化优化的加速。

(3)针对次内层的向量化方案

通常在进行向量化时，需要考虑数据布局影响向量化性能的问题，主要包括数据局部性和数据重用两个方面。首先是数据局部性。在进行向量化时，我们通常需要尽可能避免从缓存而非内存中获取数据，这样速度会大大提升。如果访存不是连续的，那就会增加 CPU 的访存次数，大幅度降低程序的执行效率。在矩阵乘法中，通常以行存储优先。因此，在进行向量化时，考虑到访存的问题，通常在矩阵规模较小时，会把矩阵的行和列存入 Cache 缓存中，在 CPU 对其进行操作时，可以从 Cache 中直接读取所需的数据，而不用频繁地访问主存。其次是数据重用。如果程序需要多次从缓存中获取数据，那么访问就会连续发生，有助于提高访问效率，从而缩短程序的执行时间。从图 4 中可以看出，在针对次内层进行向量化时，我们通常利用广播向量指令，对 $a[i][k]$ 数组进行广播，这样就可以实现通过一次存储完成 4 次计算操作，从而提升数据的重用性，减少了很多 CPU 的直接访存，同时提升了寄存器的重用，达到了提升性能的目的。此外，我们需要在循环外对归约操作进行初始化操作，同时最内层的计算操作采用向量 FMA 乘加操作指令，降低计算指令的操作数，更易于

发挥处理器的峰值计算能力。

```
for (i = 0; i < M; i ++) {
    for (j = 0; j < N; j++) {
        vc = _mm256_setzero_pd ();
            for (k = 0; k < K; k += 4) {
                va = _mm256_load_pd ( &a[i][k] );
                vb = _mm256_set_pd ( b[k+3][j],
                        b[k+2][j], b[k+1][j], b[k][j] );
                vc = _mm256_fmadd_pd ( va, vb, vc );
            }
            _mm256_store_pd ( tmp, vc );
            c[i][j] += tmp[0] + tmp[1] + tmp[2] + tmp[3];
    }
}
```

（a）最内层向量化

```
for (i = 0; i < M; i ++)
    for (k = 0; k < K; k ++){
        va = _mm256_broadcast_sd ( &a[i][k] );
        for (j = 0; j < N; j += 4){
            vc = _mm256_load_pd ( &c[i][j] );
            vb = _mm256_load_pd ( &b[k][j] );
            vc = _mm256_fmadd_pd ( va, vb, vc );
            _mm256_store_pd ( &c[i][j], vc );
        }
    }
```

（b）循环变换后最内层向量化

```
for (i = 0; i < M; i ++)
    for (j = 0; j < N; j += 4){
        vc = _mm256_setzero_pd ();
        for (k = 0; k < K; k ++){
            vb = _mm256_load_pd ( &b[k][j] );
            va = _mm256_broadcast_sd ( &a[i][k] );
            vc = _mm256_fmadd_pd ( va, vb, vc );
        }
        _mm256_store_pd ( &c[i][j], vc );
    }
```

（c）次内层向量化

图4　矩阵乘向量化方案

Fig. 4　Matrix multiplication vectorization scheme

4.2　收益分析

收益分析是进行向量化优化的前提。本节主要通过对上文提到的矩阵乘法的3种向量化方案分别进行收益分析，确定指导PMV框架进行代码生成的方案。

4.2.1　分析计算说明

向量化收益计算通过标量开销与向量化开销进行对比。

（1）标量开销

对于矩阵乘法来说，标量计算开销主要包括计算开销和访存开销，其中计算开销主要指 $M*N*(K-1)$ 次加法计算以及 $M*N*K$ 次乘法计算。这里以其操作的指令延迟为计算标准：

$$Ccost = num_{add} * Cycl_{add} + num_{mul} * Cycl_{mul} \tag{1}$$

访存开销主要包括存取操作数等，由访存次数以及平均访存的指令延迟进行表示，暂不考虑缓存不命中问题：

$$Mcost = num_M * Cycl_{aver} \tag{2}$$

由式(1)、式(2)可以得到标量版本矩阵乘法的开销：

$$Scost = Ccost + Mcost \tag{3}$$

（2）向量化开销

向量化开销主要由向量指令开销和访存开销组成。除此之外，由于矩阵乘法还具有归约操作，因此在进行向量化时还会产生冗余操作的开销。其中，向量指令开销主要包括计算操作的指令开销 $VCOcost = num_{Vcompute} * Cycl_{Vcomputer}$，以及数据拷贝插入移动(Vo)和数据广播(Vbroadcast)产生的指令开销 $DMcost = Cycl_{V_o} + Cycl_{Vbroadcast}$。因此，对矩阵乘法向量化后的向量指令开销为：

$$Vcost = VCOcost + DMcost \tag{4}$$

进行向量化操作后会产生一定的访存开销，主要包含访存向量指令开销以及进行访存操作时产生的额外开销 $Ecost$（对齐、数据重组等）。

$$VMcost = num_M * Cycl_{aver} + Ecost \tag{5}$$

由于需要对归约操作进行处理，因此会造成冗余操作的开销，这里以冗余语句增加的数量(RSnum)及其平均产生的指令延迟(Cycl)来表示。

$$Rcost = RSnum * Cycl \tag{6}$$

从式(4)—式(6)可以得出矩阵乘法的向量化开销为：

$$VECcost = VCOcost + DMcost + num_M * Cycl_{aver} + Ecost + RSnum * Cycl \tag{7}$$

4.2.2　向量化方案收益计算

本节通过前文提出的公式，结合3种向量化方案生成的汇编代码，分别对3种向量化方案的一次迭代开销进行简单计算，选取最优的向量化方案。以规模大小为64×64×64的矩阵乘法为例，汇编代码不再进行展示。

（1）针对最内层循环进行向量化。在对最内层进行向量化时，一次迭代内计算操作指令开销主要包含乘加操作以及最后对临时变量的加法操作。访存开销方面主要包括存取操作以及产生的额外开销组成，主要是数据移动插入等。其向量化开销的计算式如下：

$$VECcost = 4 * Cycl_{vadd} + 15 * Cycl_{vfmaa} + Cycl_{vmul} + 16 * Cycl_{Vinsert} + num_m * Cycl_{vmov} + 6 * Cycl \tag{8}$$

（2）针对循环变换后的最内层循环进行向量化。在该方案下，对程序进行变换会产生额外的开销，其向量化开销的计算式为：

$$VECcost = 2 * Cycl_{vfmaa} + Cycl_{Vbroadcast} + num_m * Cycl_{vmov} + Ecost \tag{9}$$

（3）针对次内层循环进行向量化。开销的计算式如下：

$$VECcost = Cycl_{vfmaa} + Cycl_{Vbroadcast} + num_m * Cycl_{Vmov} \tag{10}$$

通过式(8)—式(10)的对比可知，第三种针对次内层循环进行向量化的方案的向量操作指令开销最小，因此选取了第三种向量化方案指导PMV框架进行向量化代码生成。

5　向量化循环变换

调度变换，是多面体模型编译技术的核心阶段。基于前期PPCG编译器对输入程序的分析，生成对应的原始调度树。在该阶段，多面体编译器可以通过不同的循环变换方案，去挖掘不同层次、不同粒度的并行性，提高程序的数据局部性和程序的运行效率。

在向量化调度变换阶段，PMV框架的主要工作在于：通过

对 PPCG 编译器自动生成的原始调度树进行循环分段、添加标志位等操作，对矩阵乘法进行调度变换，为向量化操作创造条件。在中间表示上进行面向向量化的循环变换，与目标平台无关，具有较好的可移植性。图 5 给出了基于 PMV 框架生成的矩阵乘法的原始调度树。矩阵乘法对应的原始调度树中存在多种不同类型的节点，其中，band 节点用于确定迭代顺序，简单来说就是程序的循环嵌套，filter 节点表示迭代器与语句实例，sequence 和 set 节点用于指定其子节点 filter 的迭代执行类型。

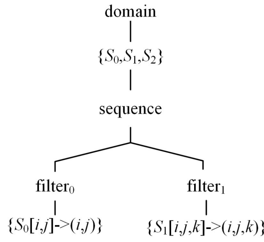

图 5 原始调度树

Fig. 5 Original scheduling tree

基于前文的分析可知，在进行矩阵乘法向量化优化时，针对次内层做向量化处理可以获得更好的加速效果。因此，在该阶段主要针对次内层进行向量化循环变换。由图 5 可知，原始调度树中包含一个 band 节点$\{S_0, S_1, S_2\}$，该节点分别对应矩阵乘法中 i,j,k 这 3 层循环的调度表示。由于 PMV 框架需要对次内层循环 j 进行向量化处理，因此首先需要对 band 节点进行拆分，通过 split_band 函数从 band 节点中将 j 层循环分割出来，将原本表示为一个 band 节点的三层循环调度通过拆分变为 3 个新的子节点。其次，PMV 框架需要通过对拆分出来的 j 层循环进行循环分段，将原本的 j 层循环分成两层循环。其中，循环分段参数 size，由目标平台的向量寄存器长度以及数据类型确定。

$$size=\frac{Vector\ register\ length}{data\ type} \tag{11}$$

通过创建的 tile_band1 函数，对原始的 j 层调度进行循环分段变换，tile_size 对应循环分段参数。循环分段后的调度树如图 6 所示。

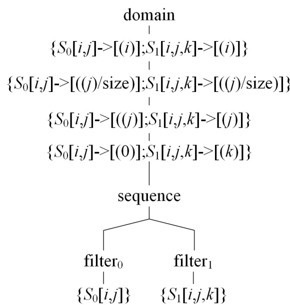

图 6 循环分段调度树

Fig. 6 Loop segment scheduling tree

进行循环分段后，原来的 j 层循环被分为两层嵌套循环，新划分出来的循环满足向量化的需求。最后，PMV 框架在进行分段后划分出来的循环前添加一个 mark 标记节点“vec”，并将其作为向量化标志。向量化调度变换阶段最终生成的调度树如图 7 所示。

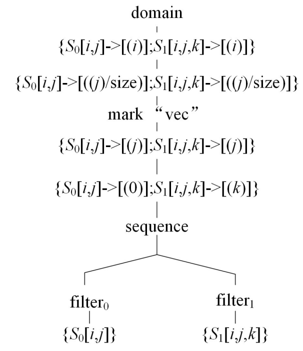

图 7 最终生成的调度树

Fig. 7 Final scheduling tree

6 代码生成

在代码生成部分，PMV 框架主要完成两部分的工作：1)在 PPCG 编译器抽象语法树的生成过程中，根据最终生成调度树上的 mark 节点，在抽象语法树上添加相应的向量操作相关标志位等；2)根据抽象语法树，进行特定平台的向量代码生成。

6.1 抽象语法树

在抽象语法树阶段，PMV 框架需要解决的主要问题是进行向量化标志位的添加及赋值。在调度变换阶段，通过循环分段将矩阵乘法从三层嵌套循环变为了四层循环。在抽象语法树部分，多面体模型会通过遍历生成语法树中的 mark 节点，收集语法树节点类型以及其对应的内容。通过对抽象语法树进行修改，可以支持多平台的代码生成，同时方便进行各种向量指令的变换。因此，在这一部分 PMV 框架主要从以下两个部分进行处理。

(1)通过多面体模型匹配向量化调度变换阶段新添加的 mark 标记节点“vec”，该标记节点把 band 节点标志为 AST 中对应可向量化的 for 节点。

(2)针对需要向量化的 for 循环节点，对其添加新的循环属性变量，即向量标志位 vectorize，并进行赋值，为后面的向量语句生成提供判断条件。

6.2 向量语句生成

在向量语句生成阶段，PMV 框架主要的工作是根据前面添加的向量化标志位对循环层的打印形式进行判断。针对进行向量化的循环层，将其退化为一条赋值语句，其余循环层仍需按照循环的表示形式进行代码生成。矩阵乘法的计算操作根据采用的向量指令集进行目标平台的向量代码生成。向量代码生成的主要流程如图 8 所示。

(1)通过对 for 节点前一阶段添加的标志位 vectorize 进行判断，来决定是否将循环层打印为一条赋值语句。如果标志位为 1，则需要去除该循环头表示形式，为该循环层生成一条赋值语句，同时通过调用创建的函数 print_vec 进行向量初始化赋值，以及归约操作的初始化；否则不改变原始表示，仍生成为循环层形式。

(2)获取前期判断向量标志位为 1 的 for 节点对应的子节点列表，对其包含的节点类型进行逐一判断，若该节点类型

为 for 节点,则添加标志位 simd 并赋值为 1。同时,调用 PPCG 已有的函数 print_node 将该节点打印为循环层表示形式。若该节点不是 for 节点,则不进行赋值代码生成,继续进行下一个节点类型的判断。

(3)针对生成循环层的 for 节点,判断其包含的属性 vectorize 值,若为 1,则判断 simd 标志位是否为 1,如果是则调用新创建的 print_simd 函数完成最终向量语句的生成,否则生成循环体形式。

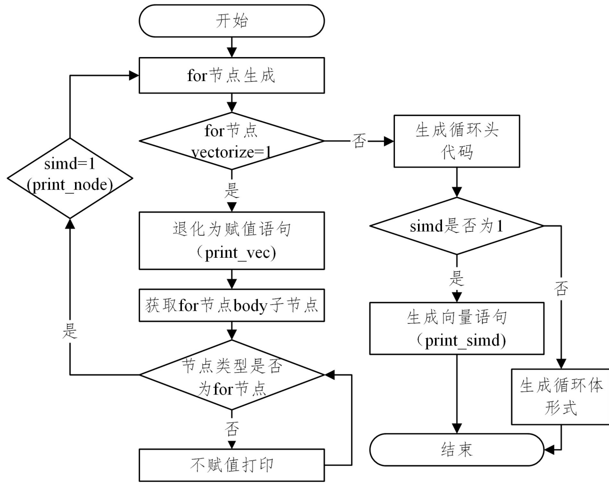

图 8 向量代码生成的主要流程

Fig. 8 Main process of vector code generation

最终生成的向量代码的核心段如图 9 所示。

```
__m256d va,vb,vc,vd;
for (int c0 = 0; c0 <= 511; c0 += 1)
    for (int c1 = 0; c1 <= 511; c1 += 4) {
        // vec
        int c2 = 0;{
            vc = _mm256_setzero_pd();
            for (int c3 = 0; c3 <= 511; c3 += 1){
                vb = _mm256_load_pd(&b[c3][c1+c2]);
                va = _mm256_broadcast_sd(&a[c0][c3]);
                vc = _mm256_fmadd_pd(va, vb, vc);
            }
             _mm256_store_pd(&c[c0][c1+c2], vc);
        }
    }
```

图 9 向量代码核心段

Fig. 9 Vector code core section

7 实验结果及分析

本节对应用 PMV 框架生成的向量代码中添加结果比对功能进行正确性验证。同时,为了测试自动生成向量代码的性能,选取了 ICC 和 PLUTO 编译器进行性能测试。

7.1 实验平台与测试用例

实验选取 13 个规模在 64×64×64 到 1 024×1 024×1 024 之间的矩阵乘法程序作为测试用例,测试平台为 Intel 至强处理器 T640。PMV 框架采用 AVX 指令集,可以同时处理 4 个 double 类型的数据。由于 PPCG 是一个“源-源”的编译器,因此需要借助基础编译器 ICC 在相应的目标平台上对其生成的代码进行运行。实验平台信息如表 1 所列。

表 1 实验平台信息

Table 1 Experiment platform information

Intel © Xeon © Silver 4110	
主频/GHZ	2.10
核心	32
L1 数据缓存/kB	32
L2 缓存/kB	1024
ICC 版本	19.1.1.217
PPCG 版本	0.08.3
PLUTO 版本	0.11.4

7.2 CPU 平台向量化性能测试

本节主要对 PMV 框架生成的向量化代码进行性能测试。首先通过 ICC 对矩阵乘法串行程序开启自动向量化优化的执行时间进行记录。其次,通过 PPCG 和 PLUTO 编译器对矩阵乘法串行程序进行多面体分析与变换,生成的代码通过 ICC 编译器开启向量化优化进行运行,分别记录运行时间。此外,对应用 PMV 框架生成的向量化代码应用 ICC 编译器运行的时间进行记录。最后,通过与 ICC 未开启向量化优化的执行时间进行对比,以获得加速比。其中,所有的程序运行结果对比正确。相关测试用例以及编译选项如表 2 所列。相应的测试结果如图 10 所示。

表 2 测试基准及编译选项

Table 2 Compilation options

编译平台	编译选项
ICC(串行)	-O3 -no-vec
ICC	-O3
PLUTO	--parallel
PPCG	--target=c
PMV	--target=c --vectorize
PMV+unroll	--target=c --vectorize --unroll

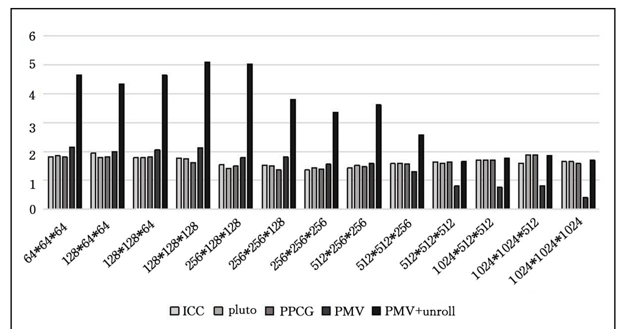

图 10 CPU 平台上向量化的加速比

Fig. 10 Vectorize acceleration ratio on CPU platform

从图 10 可以看出,在 CPU 平台上,小规模矩阵乘法可以获得更好的性能。采用基于 PMV 框架生成的代码性能优于基础编译器 ICC 的自动向量化性能,同时也高于利用 ICC 的自动向量化功能运行 PPCG 和 PLUTO 生成的代码性能。但是,到达 512×512×512 规模后采用该框架生成的向量化代码性能大幅度下降,相比其余 3 种测试程序的向量化加速比更低,甚至产生了负加速。分析产生该实验结果的原因,主要有以下两点。

(1)当矩阵乘法进行计算时,通常按行优先的顺序进行。由于 Cache 读取数据是以块为单位的,对于矩阵乘法来说,当规模较小时,CPU 对其进行操作可以直接从相邻 Cache 的

一块数据中读取需要的数据,访问连续的地址,具有较高的命中率。由于 CPU 访问缓存的速度远快于访问主存的速度,因此此时的程序具有较好的局部性以及较优的执行效率。而随着矩阵规模的增大,每次读取的一块数据无法满足一次操作所需要的全部数据。因此,此时 CPU 会频繁地对 Cache 进行存取操作,这样会导致大规模矩阵在计算时,Cache 的命中率大大降低,造成加速逐渐下降。

(2)对于大规模矩阵计算产生负加速的结果,通过对比 ICC 的自动向量化生成报告可知,ICC 在进行自动向量化操作的同时,还对循环层进行了循环展开操作。而本文工作仅仅对其进行了向量化操作。为了更好地验证采用 PMV 框架生成的程序性能,在向量化的基础上,对其进行了展开操作,展开策略与 ICC 保持一致,进一步进行了测试。

在进行循环展开优化之后可以看到,与未进行展开优化相比,大规模的矩阵乘法向量化加速比产生了正加速,且稍高于其他 3 种测试程序的加速比。进行循环展开之后,所有规模下的加速比相比只做向量化优化时增加了一倍,尤其是在小规模矩阵中,加速尤为明显。通过分析可知,循环展开操作可以有效地提升寄存器的重用,同时可以极好地提升指令流水,从而提高程序的运行速度。但是,仍然存在随着矩阵规模的增大,加速比逐渐下降的现象,并且加速比远低于理论加速比。通过前文的分析可以得知,在进行大规模矩阵乘法计算时,由于 Cache 命中率的降低,导致程序的性能下降。因此,我们采用了循环分块的优化手法,进一步对向量化加速进行测试。

7.3 基于循环分块的向量化性能测试

本节对采用 PMV 框架生成的向量化代码进行循环分块,利用 ICC 编译器对其进行编译;同时,记录 ICC 编译自动向量化加分块优化的串行程序的运行时间。将二者与 ICC 未加自动向量化的分块优化程序执行时间进行对比,其结果验证正确。分块后的编译选项如表 3 所列。

表 3 分块后的测试程序及编译选项

Table 3 Compilation options under after blocking

编译平台	编译选项
ICC(tile)	-O3 -no-vec
ICC(SIMD+tile)	-O3
PMV+tile	--target=c --vectorize --tile

在当前通用的矩阵乘法优化方法中,通常会对大规模矩阵进行分块处理。因此,为了更好地验证应用框架生成的向量化代码性能,结合上文的分析,我们进一步对矩阵进行了循环分块处理。简单来说,其原理就是在进行计算时,把大矩阵划分为一个个子矩阵,从而可以以分块大小为单位进行存取,大幅度提升了程序的访问速度。通过记录不同分块大小的程序运行时间,从中选取最优效果的分块大小。对基准程序同样进行分块操作,分块大小与 PMV 框架生成的向量化代码分块大小保持一致。分块后的编译选项如表 3 所列。分块后的向量化加速比如图 11 所示。

通过图 11 可以看到,在进行了矩阵分块后,向量化加速比更为明显,特别是在大规模矩阵的情况下,加速比呈上升趋势。但是,对于小规模矩阵来说,加速比相比未分块前变化不大,出现该实验结果的原因是:对矩阵进行分块,可以有效地避免 CPU 频繁地对 Cache 进行存取操作,可以充分利用高速缓冲区,使程序具有较好的局部性,从而提升了大矩阵的向量化代码性能。但是,对于小规模矩阵乘法来说,Cache 的命中率低并不是其性能瓶颈,因此是否分块对其性能影响不大。在进行循环分块的基础上,本文工作实现的加速比明显高于 ICC 分块后的自动向量化加速比,平均达到了 2.64。

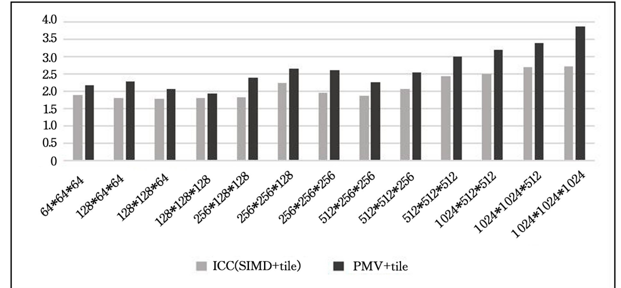

图 11 分块后的向量化加速比

Fig. 11 Ratio of vector quantitative acceleration after blocking

7.4 基于多线程的向量化性能测试

在循环分块的基础上,我们借助通用的多线程 OpenMP 优化手段,希望通过划分子任务来提升程序的计算效率,同时结合硬件体系结构,发挥处理器的性能,这里以 16 个线程进行测试。利用 ICC 编译器对多线程加分块优化的串行程序和基于 PMV 框架生成的向量化加分块以及多线程的程序进行编译并记录二者的运行时间,与 ICC 编译多线程加分块但未做向量化的程序执行时间进行对比,计算加速比,其程序结果运行正确。多线程下的相关编译选项如表 4 所列,基于循环分块以及多线程下的向量化加速比如图 12 所示。

表 4 16 个线程下的测试基准及编译选项

Table 4 Options and benchmarks under sixteen threads

编译平台	编译选项
ICC(16 线程+tile)	-O3 -no-vec -qopenmp
ICC(SIMD+16 线程+tile)	-O3 -qopenmp
PMV+(tile+16 线程)	--target=c --vectorize --tile --openmp

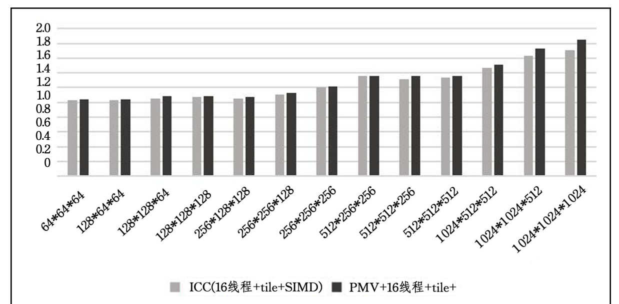

图 12 16 线程后的向量化加速比

Fig. 12 Ratio of vector quantitative acceleration after 16 threads

可以看出,在小规模矩阵下,ICC 和 PMV 的加速效果都不明显,其原因是,采用了 OpenMP 优化后,会产生创建和合并线程的开销,同时具有负载不均衡的问题,大大降低了程序的执行效率。而随着矩阵规模的增加,产生的调度开销可以忽略不计,因此可以发现,在进行多线程优化后,大规模矩阵的向量化加速比更加明显。

7.5 小结

本节分别对 PMV 框架自动生成的向量化代码、分块后的向量化代码以及多线程下的向量化代码的性能进行了测试。总体来说，应用本文设计的 PMV 框架自动生成的向量化代码的性能高于 ICC 编译器的自动向量化性能。其原因主要有以下几点：1）PMV 始终从最优的向量化方案出发，进行代码生成；2）其可以充分利用多面体模型的分块以及多线程能力，生成局部性较高的代码；3）PMV 在进行向量代码生成的过程中没有标量扩展、循环分布等操作，减少了不必要的内存开销；4）基础编译器在进行自动向量化操作时，使用了数据重组指令，带来了额外的开销。

虽然采用该框架生成的向量化代码的性能较高，但仍未达到理论加速比，其主要原因是对次内层进行向量化时数组 a 对 j 的循环索引不连续，使得存在跨幅加载和存储的问题，造成了向量化性能的降低。

基于多面体模型的向量化代码生成框架（PMV）可以自动生成分块、多线程的高效代码，缩短了手工编程的时间，同时具有较低的编译成本，在各种规模下时间不超过 0.1 s。

结束语 为了利用编译器生成高效的矩阵乘法向量代码，本文提出了基于多面体模型的向量代码生成框架（PMV），并针对矩阵乘法进行了向量化分析，指导框架进行代码生成。通过测试，对应用 PMV 框架生成的向量代码进行了正确性和性能测试，未来可以将该框架应用到矩阵乘法在多核处理器上的优化工作中。

虽然本文实现了基于多面体模型的矩阵乘法向量代码生成，但是该框架针对矩阵乘法程序的分块优化手段较为单一。同时，该框架只针对矩阵乘法程序，应用范围不够广泛。因此，下一步需要进行的工作有：1）完善 PMV 框架，结合循环分块手段，进一步进行向量化优化，提升矩阵乘法的性能；2）对于批量矩阵乘法，应用 PMV 框架对其进行细粒度并行开发；3）对于其他的典型科学计算核心，如模板计算等进行分析，针对不同的程序特性进行向量化分析，应用基于多面体模型的向量代码生成框架进行代码生成。

参 考 文 献

[1] KANG H，KWON H C，KIM D. HPMaX：heterogeneous parallel matrix multiplication using CPUs and GPUs[J]. Computing，2020，102(12)：2607-2631.

[2] HEMEIDA A M，HASSAN S A，ALKHALAF S，et al. Optimizing matrix-matrix multiplication on intel'sadvanced vector extensions multicore processor[J]. Ain Shams Engineering Journal，2020，11(4)：1179-1190.

[3] HOSSEIN A，ASADOLLAH S. SIMD programming using Intel vector extensions - ScienceDirect[J]. Journal of Parallel and Distributed Computing，2020，135：83-100.

[4] FEAUTRIER P，LENGAUER C. Polyhedron model[J/OL]. https://link. springer. com/referenceworkentry/10. 1007/978-0-387-09766-4_502.

[5] VERDOOLAEGE S，CARLOS JUEGA J，COHEN A，et al. Polyhedral parallel code generation for CUDA[J]. ACM Transactions on Architecture and Code Optimization（TACO），2013，9(4)：54：1-54：24.

[6] BONDHUGULA U，HARTONO A，RAMANUJAM J，et al. A practical automatic polyhedral parallelizer and locality optimizer [C]// Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'08). ACM，2008.

[7] GAO W，ZHAO R C，HAN L，et al. Overview of SIMD automatic vectorization compilation optimization [J]. Journal of Software，2015，26(6)：1265-1284.

[8] EDAMATSU T，TAKAHASHI D. Accelerating Large Integer Multiplication Using Intel AVX-512IFMA[M]. Algorithms and Architectures for Parallel Processing，2020：60-74.

[9] STEPHENS N，BILES S，BOETTCHER M，et al. The ARM Scalable Vector Extension[J]. IEEE Micro，2017，37(2)：26-39.

[10] ZHAO J，LI Y Y，ZHAO R C. "Black magic" of polyhedral compilation[J]. Journal of Software，2018，29(8)：2371-2396.

[11] VERDOOLAEGE S，GUELTON S，GROSSER T. Schedule Trees[J]. Geodinamica Acta，2013，25(1/2)：86-95.

[12] GROSSER T，VERDOOLAEGE S，COHEN A. Polyhedral AST generation is more than scanning polyhedra[J]. Acm Transactions on Programming Languages & Systems，2015，37(4)：1-50.

[13] KONG M，VERAS R，STOCK K，et al. When polyhedral transformations meet SIMD code generation[C]// Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM，2013.

[14] LENGAUER G C. Polly－Performing Polyhedral Optimizations on a Low-Level Intermediate Representation[J]. Parallel Processing Letters，2012，22(4)：14-53.

[15] VERDOOLAEGE S，JANSSENS G. Scheduling for PPCG [EB/OL]. https://www. researchgate. net/publication/317826152_Scheduling_for_PPCG.

[16] LIU Z，TIAN X. Matrix multiplication vectorization method for multi-core vector processors[J]. Chinese Journal of Computers，2018，41(10)：2251-2264.

[17] LIU G，ZHANG H，MAO R，et al. Optimization of DGEMM Function for Loongson3B1500 Architecture[J]. Small Microcomputer System，2014，35(7)：1523-1527.

WANG Bo-yang，born in 1997，postgraduate. Her main research interests include high-performance computing and so on.

PANG Jian-min，born in 1964，professor，Ph.D，is a member of China Computer Federation. His main research interests include high-performance computing and information security.

（责任编辑：喻藜）