基于指导语句的函数向量化技术研究

刘丽丽，单征，李颖颖，武文浩，刘文博

引用本文

刘丽丽，单征，李颖颖，武文浩，刘文博. 基于指导语句的函数向量化技术研究[J]. 计算机科学，2025，52(5)：76-82.

LIU Lili, SHAN Zheng, LI Yingying, WU Wenhao, LIU Wenbo. Research on Function Vectorization Technology Based on Directive Statements [J]. Computer Science, 2025, 52(5): 76-82.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.231200174

基于指导语句的函数向量化技术研究

刘丽丽[1] 单 征[1] 李颖颖[1] 武文浩[2] 刘文博[1]
1 解放军战略支援部队信息工程大学 郑州 450001
2 国家并行计算机工程技术研究中心 江苏 无锡 100190
(liull_lili@163.com)

摘 要 随着处理器技术的不断发展,SIMD(Single Instruction Multiple Data)向量化已经在各个领域得到广泛的应用。然而,过去的研究主要集中在循环和基本块上,而全函数向量化可以更好地利用SIMD指令的优势,从而提高应用程序的性能。文中提出了一种基于指导语句的函数向量化方法。首先,在涉及函数调用的循环上加上一种较为简单的指导语句,即可对循环中涉及函数调用的指令进行向量化。其次,对于被调函数的向量化采用全函数向量化的方式,生成向量化的全函数而不是对其内联。最后,处理循环中的函数调用点,生成向量化的函数调用指令。这种方法可以充分利用SIMD指令的优势,提高应用程序的性能。从ISPC基准测试和SIMD库基准测试中选取了10个基准测试来评估所提方法,实验结果表明该方法与标量相比,平均加速比达到了6.949倍。

关键词:函数向量化;SIMD;自动向量化
中图分类号 TP312

Research on Function Vectorization Technology Based on Directive Statements

LIU Lili[1],SHAN Zheng[1],LI Yingying[1],WU Wenhao[2] and LIU Wenbo[1]
1 PLA Strategic Support Force Information Engineering University,Zhengzhou 450001,China
2 National Research Center of Parallel Computer Engineering Technology,Wuxi,Jiangsu 100190,China

Abstract With the continuous development of processor technology,SIMD(Single Instruction Multiple Data) vectorization has been widely applied in various fields. However,previous research has mainly focused on loops and basic blocks,while full-function vectorization can better exploit the advantages of SIMD instructions,thereby improving application performance. This paper proposes a guided statement-based function vectorization method. Firstly,a relatively simple guided statement is added to the loop involving function calls,which can vectorize the instructions involving function calls in the loop. Secondly,the vectorization of the called function is achieved by using full function vectorization to generate a vectorized full function instead of inlining it. Finally,the function call instructions in the loop are processed to generate vectorized function call instructions. We selected 10 benchmarks from ISPC benchmark tests and SIMD library benchmark tests to evaluate our method,and the experimental results show that compared to scalar,the average speedup achieved is 6.949 times.

Keywords Function vectorization,SIMD,Automatic vectorization

1 引言

随着计算机技术的发展和处理器性能的不断提升,利用现代处理器的并行性能进行高效计算已经成为计算机科学研究的重要方向。其中,SIMD技术是一种广泛应用于现代处理器中的并行计算技术,它可以在一个指令中同时处理多个数据元素,从而显著提高计算性能。自动向量化是一种编译器优化技术,它可以将SIMD指令应用于连续的数据序列,从而实现高效的并行计算。

然而,自动向量化的研究大多在循环和基本块层面上,很少有对全函数向量化的研究。全函数向量化是一种将整个函数转换为可同时处理多个数据元素的SIMD指令的技术,可用于提高计算效率。不同于传统的循环向量化,全函数向量化的优点在于它可以将整个函数的计算效率最大化,包括控制流和变量使用等方面,可以更好地利用SIMD指令的优势,其代码具有较高的可读性和易用性。除此之外,函数向量化比传统的基于循环的操作更快,特别是在处理大型数据集时效果明显。随着多核处理器和GPU等并行计算平台的出现,SIMD向量化技术在许多领域都得到了广泛的应用,如图像处理、数据处理、机器学习、深度学习等。

Kandiah等[1]提出了一种名为Parsimony的SPMD编程方法,旨在通过高效地利用CPU的SIMD/vector单元来实现

到稿日期:2023-12-25 返修日期:2024-06-29
基金项目:2024年先进计算与智能工程实验室(ACE)项目
This work was supported by the 2024 Laboratory for Advanced Computing and Intelligence Engineering(ACIE) Project.
通信作者:李颖颖(ieulyy@163.com)

高计算性能，并与标准编程模型、语言和编译器工具链兼容。该方法可以对含有函数调用的循环进行很好的向量化。然而，这种编译方式采取内联的方式对函数调用进行向量化，并且不能处理带有 break，return 等发散控制流。Moll 等[2]提出了一种 partial linearization 的 if-conversion 算法，该算法可以有效地处理 SIMD 程序中的分支，通过将非发散分支进行 if-conversion，从而避免了在 SIMD 程序中执行多个目标的问题，提高了 SIMD 的利用率。同时，该算法可以处理 break，return 等发散控制流，对含有函数调用等复杂情况也能够很好地支持，并在外层循环向量化进行实现。然而，它对于函数调用依然采用内联的方式进行向量化，没有对调用点进行处理。若强制不内联，则会因没有对被调函数进行提前预处理而不能向量化。Rapaport 等[3]提出了一种 CHORσuS（C Higher-Order Vector Semantics）的 C 语言扩展，它是一种轻量级的静态扩展，允许程序员将计算表示为应用于标量内核的可组合向量操作。在 C 语言中实现全函数向量化，通过使用 map 和 fold 函数来表示向量操作。虽然它并没有内联处理函数调用，但是要求程序员在 C 语言中编写含有 map 和 fold 函数表示的向量化代码，给程序员增加了负担，有违自动向量化。Tian 等[4]提出了一种在多核 SIMD 处理器上编译的 C/C++ SIMD 扩展，以实现函数和循环向量化。他们还介绍了一组新的 C/C++高级向量扩展和扩展的 Intel C++产品编译器，用于将这些向量扩展转换为优化的 SIMD 指令序列，以实现向量化的函数和循环，然而其扩展子句过多且有些繁琐，对程序员来说并不友好。Masten 等[5]通过在循环向量化 pass 之前添加一个名为 VecCLONE 的 pass 来实现函数级的向量化，通过 OpenMP 编译指示实现函数向量化。VecClone pass 的函数向量化方式是将函数体中的代码放在一个 vf（Vectorization Factor）次循环中，这种实现方式改变了函数体的内部结构，使其可读性降低。Karrenberg[6]提出了一种基于控制流图（CFG）的 SSA 形式的低级中间代码的全函数向量化代码转换方法，该方法利用 OpenMP 指导语句，需要同时对调用点所在的循环和被调函数都加上指导语句来对其进行向量化，并且 OpenMP 指导语句的子句较多，使用起来较为繁琐。

基于上述问题，本文提出了一种基于指导语句的函数向量化。首先，只需要在函数调用语句所在的循环中加上一种较为简单的指导语句，而无需对被调函数进行任何标记，即可对循环中涉及函数调用的指令进行向量化。其次，对于被调函数的向量化采用全函数向量化的方式，而不是将其内联，为被调函数提供一个全函数向量化的版本。最后，对循环中的函数调用点进行处理，生成向量化的函数调用指令，提高代码可读性和易用性。在 LLVM 编译器基础设施中对其进行实现，选取了 10 个基准测试来评估所提方法，实验结果表明该方法与标量相比，平均加速比提升达到了 6.949 倍。

2 背景知识

SIMD 扩展部件可以在不同粒度下进行向量化，主要包括基本块级向量化、循环级向量化和函数级向量化[7-8]。

2.1 基本块级向量化

基本块级向量化是一种针对基本块的向量化技术，旨在将相似的独立指令组合成向量指令。这种技术可以应用于内存访问、算术运算、比较运算和 PHI 节点。基本块向量化与循环向量化不同，它更关注迭代内基本块中的向量化机会。基本块向量化的过程大致分为识别相邻内存引用、扩充打包列表、合并打包列表和生成向量代码 4 个步骤。首先，识别相邻内存引用是 SLP 向量化过程中的关键步骤。算法会遍历基本块中的任意语句对，检查它们是否访问了相邻的内存地址。如果满足条件，并且这两个语句可以打包在一起，那么它们将被添加到同一个向量中。接着，扩充打包列表，利用已经识别出的相邻内存引用，进一步扩充打包列表。这意味着算法会查找与已打包语句相关的其他语句，并检查它们是否也可以打包在一起。然后，合并打包列表，算法会尝试将一些打包列表合并起来，以生成更长的向量。这可以通过识别两个打包列表中共有的元素来实现。最后，生成向量代码，根据数据依赖关系，将打包列表中的语句整理成向量指令。

Larsen 等[9]首次提出基本块向量化技术，即超字并行（Superword Level Parallelism，SLP），利用基本块内数据的连续内存访问和复用信息，将多条相似且可并行执行的语句组合成一个向量指令，从而提高代码的执行效率。之后，学术界对 SLP 向量化的研究从未停止。Porpodas 等[10-11]提出了基于表达式等价变换的 SLP 向量化方法——LSLP（Left-to-Right Superword Level Parallelism）和 SN-SLP（Superword-Level Parallelism with Non-Isomorphic Statements），通过调整非同构语句的操作顺序或利用等价关系和等价扩展关系，将操作数顺序不同的非同构语句转换为操作数顺序相同的同构语句，从而实现向量化。随后，Feng 等[12]针对操作数数目不同的非同构语句的向量化问题，提出了一种名为 SLP-E 的方法，其将操作数数目不同的非同构语句转换为操作数数目相同的同构语句，从而实现向量化。

2.2 循环级向量化

循环向量化主要包括循环结构分析、依赖分析、向量指令的生成和尾部循环处理 4 个部分。首先进行循环结构分析，编译器会分析循环的结构，以确定是否适合向量化。循环的结构必须能够被有效地分解成一系列可以并行执行的操作，排除无法向量化的循环，如含有函数调用、跳转语句等。接着，依赖分析识别循环中的数据相关关系，包括读取和写入变量的依赖关系，通过构建语句依赖图，求解强连通分量，确定能够向量化的语句。只有当语句依赖图中所有真依赖环上的依赖距离之和小于或等于向量化因子时，才能进行向量化。其他依赖形式可通过循环分布、节点分裂、标量扩展等技术解决[13]。然后进行代码生成，将标量操作转换为向量操作，以便同时处理多个数据元素，利用硬件上的 SIMD 功能。最后，处理尾循环，如果循环的迭代次数不能被向量宽度整除，编译器可能需要处理尾循环，这可能包括使用标量指令处理剩余的迭代。

Allen 等[14]首次提出了基于循环的自动向量化方法。这种方法通过对内层循环的迭代空间进行操作，将整个数组视为一个向量单元。通过进行依赖关系分析，能够将在不同迭代之间并且彼此之间不会形成依赖环的多条语句转换为向量形式。随着循环向量化的发展，学术界开始寻找外层循环的向量化机会，Bik[15]和 Hampton 等[16]针对最内层循环存在依

赖环、归约或数组引用与循环索引不连续等情况时，向量化代价较大或无法实现的问题，提出循环交换可将某外层循环交换至最内层以实现向量化。为充分利用跨外层循环迭代的数据并行性或直线代码中的数据并行性，Nuzman[17]提出了一种新的 SLP(Loop-Aware SLP)方法，它将循环向量化与 SLP 向量化结合使用，以寻找迭代间的向量化机会。

2.3 函数级向量化

函数级向量化从函数粒度识别程序中的数据级并行发展到过程间分析。函数的参数为向量，返回值也为向量，因此需要将多次标量函数调用转换为一次向量函数调用。一般情况下，连续多次函数调用一般仅出现在循环体内，因此函数向量化一般与循环向量化配合使用。如图 1 所示，图 1(a)为标量函数，图 1(b)是图 1(a)的向量化函数，其传入参数 a，b 和返回值 c 都是向量。

```
float foo (float a, float b)
{
    float c;
    if(a < b)
      c = a + 1;
    else
      c = a - 1;
    return c;
}
```

(a)

```
__m128 foo_sse (__m128 a, __m128 b)
{
        __m128 mask = _mm_cmpgt_ps (a, b);
__m128 s = _mm_add_ps (a, _mm_one);
        __m128 t = _mm_sub_ps (a, _mm_one);
        __m128 c = _mm_blendv_ps (mask, s, t );
        return c;
}
```

(b)

图 1 函数向量化图

Fig. 1 Function vectorization diagram

目前，在循环中存在函数调用，一般采取不向量化或将其内联的方式进行向量化，然而这并不是真正的函数向量化。Karrenberg[6]在静态单赋值表示形式(SSA)下，提出了一种基于 SSA 形式的全函数向量化变换，通过数据流分析和变换利用掩码和选择指令解决函数向量化中运行操作不一致的问题。Li 等[18]通过分析程序中操作和基本块的 SIMD 特性，提出了一种基于 SIMD 特性实现全函数向量化的代码优化，包括实例多版本、实例重组和向量化指令优化。

3 基于指导语句的函数向量化流程

基于指导语句的函数向量化主要包括解析指导语句确定向量化循环、预转换、向量化分析、掩码生成、选择生成、控制流图(CFG)线性化和向量化代码生成 7 个阶段，具体如图 2 所示。

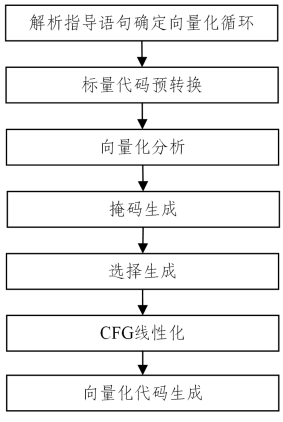

图 2 基于指导语句的函数向量化流程图

Fig. 2 Function vectorization process based on directive statements

3.1 解析指导语句确定向量化循环

编译器首先能够识别需要向量化的循环，才能进行后续的向量化工作。在 C/C++语言中，通过使用指导语句来告诉编译器需要进行向量化优化。目前基于指导语句的函数向量化需要在循环和被调函数上同时加上相应的指导语句。本文采用 LLVM 编译器的 clang 指导语句，只需要在函数调用语句所在的循环中加上一种较为简单的指导语句，而无需对被调函数进行任何标记，即可对循环中涉及函数调用的指令进行向量化。如图 3 所示，vectorize(enable)表示启用向量化，vectorize_width(W)表示设置向量寄存器的宽度为 W。这些指令会被编译器前端解析，并告诉编译器进行向量化优化。

```
#pragram clang loop vectorize(enable) vectorize_width(W)
for (int i = 0; i < N; i++)
{
    c = foo(a, b);
}
```

图 3 添加指导语句的循环

Fig. 3 Loop with instruction statements added

在编译过程中，编译器前端会识别循环体中的 # pragma clang loop vectorize(enable)指令，并将其转换为对应的中间表达式形式，将循环元数据中的信息进行更新，生成一个名为 llvm. loop. vectorize. enable 的元数据节点和一个名为 llvm. loop. vectorize. width 的元数据节点，其值将被设置为 true 和 W，表示该循环已经被指示进行向量化优化，并且指定向量寄存器宽度为 W。需要注意的是，vectorize_width 的具体取值应根据目标硬件架构进行选择。常见的向量寄存器宽度包括 128 位(如 SSE 指令集中的 xmm 寄存器)和 256 位(如 AVX 指令集中的 ymm 寄存器)。在选择向量寄存器宽度时，应根据目标处理器的支持程度和性能需求进行综合考虑。

接着，编译器中端通过循环标识符(LoopID)来获取循环的元数据节点 llvm. loop. vectorize. enable 的值，以确定该循环是否需要进行向量化。如果该循环需要进行向量化，则将其加入向量化任务列表中，以便后续对其进行向量化处理。在向量化过程中，编译器会针对循环体内的操作进行优化，生成相应的向量指令，从而实现并行计算。

通过采用指导语句进行函数向量化的方式，编译器能够根据开发者的指导和代码结构来判断哪些循环适合进行向量化优化，并生成相应的优化指令，从而提高程序的执行效率。

3.2 预转换

在向量化之前，要进行一些准备性的转换，其中最重要的是对循环进行简化，以确保每个循环都只有一个入边和一个回边，这样可以确保存在唯一的循环 header、唯一的循环 pre-header(循环进入的块)和唯一的循环 latch(一个从循环后端返回到循环头部的块)。除此之外，运行时函数通常涉及一些对程序状态的修改或系统调用等操作，无法直接进行向量化处理。因此，对这些函数进行运行时函数降低处理，降低为基本指令可以使编译器能够更好地进行代码优化。

3.3 向量化分析

为充分利用指令集架构的并行性，加速程序的执行速度，在向量化之前需要对代码进行向量化分析，通过对代码进行静态分析来跟踪确定变量的向量形状，对不同的向量化形状进行不同的向量化处理或标量处理。

3.3.1 向量形状介绍

形状分析试图跟踪单个变量在 SIMD 寄存器上的特性。如表 1 所列，如果一个变量在每个实例中都包含相同的值，那么它的向量化形状就是统一的(Uniform)。统一值可以存储在标量寄存器中，并通过标量指令进行操作，这可以改善许多 CPU 架构中的延迟、吞吐量和寄存器压力。如果所有实例共同的基值加上每个实例的偏移量，那么它的向量化形状就是跨步的(Strided)。如果跨步为 1，那么它的向量化形状就是连续的(Contiguous)。对于连续的和跨步的变量，我们只需要存储其基值，偏移量可以在后续使用时根据跨步大小生成相应的向量。如果 SIMD 实例的结果是自然数，并且第一个实例的结果是向量宽度的倍数，那么它的向量化形状就是对齐的(Alignment)。SIMD 硬件通过更有效的向量内存操作来访问对齐的内存位置。对于还没有进行计算的变量形状，将其设为未定的(Undef)，后续进行分析计算。对于在不同的实例持有不同的值且这些值没有任何规律的变量，将其向量化形状设为变化的(Varying)。Varying 的变量就需要变成相应的向量变量，进行向量的运算操作。

表 1 向量形状描述

Table 1 Vector shape description

Symbol	Name	Example	Description
U	Uniform	〈3,3,3,3〉	所有实例具有相同的值
V	Varying	〈3,9,8,2〉	不同实例具有不同值且无规律
C	Contiguous	〈3,4,5,6〉	实例具有连续的值
S	Strided	〈3,5,7,9〉	实例具有跨步的值
A	Alignment	〈0,1,2,3〉	实例 1 与向量宽度的整数倍对齐
UD	Undef	向量形状还未定义	向量形状还未定义

3.3.2 形状传播

该过程首先初始化迭代变量形状，迭代变量的形状一般是跨步的或者变化的，跨步为 1 即为连续。其次，向量形状从不依赖于其他的值开始传播，如果没有参数的调用，则使用常数作为输入值的 phi 节点、Alloca 指令等。最后，根据指令的操作数和操作符等信息，计算指令的向量形状。例如对于一个加法指令，先获取其两个加法操作数的向量化形状，检查形状是否已定义，如果其中一个未定义(Undef)，则返回一个未定义(Undef)的形状。然后检查形状是否具有常量步长(即每个元素之间的固定偏移量)，如果其中一个操作数形状不具有常量步长，则指令的形状为变化的(Varying)形状，且对齐为两个操作数对齐及其跨步的最大公约数。如果两个操作数形状都具有常量步长，则此加法指令的向量形状是跨步的(跨长为两个操作数形状的跨步之和)，且对齐为两个操作数对齐的最大公约数。关于函数调用指令的形状计算，通过计算其函数体内各个指令及变量的形状，最后计算出来的返回值的形状即为函数调用指令的向量化形状。

3.4 掩码生成

在控制流中，条件分支的存在可能会导致控制流发散，即这些分支可能会产生不同的执行路径，因为分支的条件可能对某些实例为真，而对其他实例为假。因此，所有代码都会执行。为避免副作用的产生，需在控制流边上使用掩码(也经常称为谓词)来显式地传递控制。一个块的入口掩码是所有入边的掩码的析取。对于循环头部而言，入口掩码是一个 phi 函数，其输入值来自循环的预头部和尾部。离开一个块的控制流边的掩码由块的入口掩码和该块的进入掩码以及潜在的出口掩码决定。图 4 给出了图 3 所示实例掩码生成的一些表示方式。基本块 A 为预头部，基本块 B 是循环体，基本块 C 是循环退出块。循环的入口掩码 m_B 是一个 *phi* 函数，其输入值一个来自循环头部($m_{A\to B}$)，一个来自循环尾部($m_{B\to B}$)。$m_{A\to B}$ 为边 $A\to B$ 的退出掩码，$m_{B\to B}$ 为边 $B\to B$ 的退出掩码。掩码操作通过将当前迭代中离开循环的实例元素设置为 true 来更新出口掩码，phi 函数用来保存当前的出口掩码。故在此过程之后，边 $B\to C$ 的退出掩码变为 m_{exit} 而不是 $m_{B\to C}$。如果出口分支是条件性的，则真边的出口掩码是其入口掩码和分支条件的合取。假边的出口掩码是其入口掩码和否定的分支条件的合取。

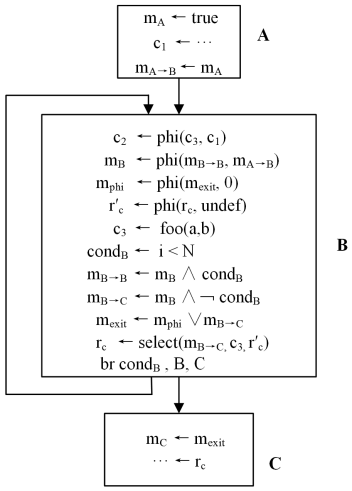

图 4 掩码生成和选择生成图

Fig. 4 Mask generation and select generation

3.5 选择生成

只有当不活动实例的结果被丢弃时，控制流的线性化才是可能的。在掩码生成中使用掩码变量可以表示哪些分支是有效的，哪些分支是无效的。为了将无效分支的结果与有效分支的结果进行合并，从而产生一个线性指令序列，还需要在控制流连接点和循环 latch 处插入选择操作。原始 CFG 中的 phi 函数表示在控制流图中，多个分支可能会合并为一个分支。为了将 phi 函数转换为线性指令序列，可以使用选择指令来替换 phi 函数。具有 n 个输入值的 phi 函数可以被转换为 $n-1$ 个连接的选择指令序列。此外，每个循环都需要结果向量，以保留那些提前离开循环的实例的循环活跃值。循环活跃值指那些在循环迭代之间保持活动状态的值，这些值可以在循环的后续迭代中使用，或者在循环之外使用。如图 4 所示，变量 c 即为循环活跃值，其在基本块 A，B，C 中都使用了。r_c' 是循环活跃值 c 的结果向量，每当一个实例离开循环时，对应的 c 元素会混合到结果向量中。

3.6 CFG 线性化

在插入所有掩码和选择操作后，可以通过数据流表示所有的控制流，因此可以删除这些控制流。为了实现这个目标，需要按照原始 CFG 中的执行顺序对基本块进行排序。在 G 的每个可能执行的过程中，如果 A 在 B 之前执行，那么在扁平化的 CFG G0 中，A 必须在 B 之前。如果 CFG 分为两条路

径，则首先执行一条路径，然后执行另一条路径。这个顺序通过递归地对 G 的循环树进行拓扑排序来确定。

3.7 向量化代码生成

在线性化之后，便可以开始进行实际的向量化代码转换。单个指令向量化基本上是将标量指令一对一地转换为它们的 SIMD 版本，例如一个标量的 add 指令转换为一个向量的 add 指令。特别地，对于向量形状为 Uniform 的变量，保持其标量形式，必要时可将其广播为向量。但是对于函数调用指令，需要做另外的处理，具体步骤如图 5 所示。

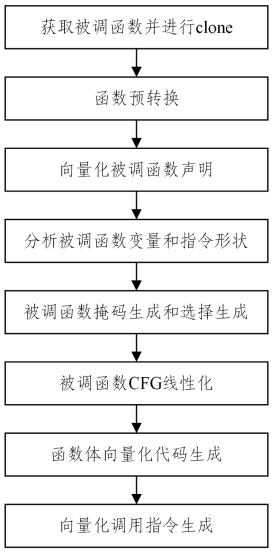

图 5 函数调用点处理过程图

Fig. 5 Process of function call point processing

因循环向量化时并没有将循环中被调函数进行内联，所以对循环已经进行的预转换，形状分析和线性化都没有包含被调函数。因此，对于被调函数，需要重新进行向量化的分析和转化等步骤。

首先，获取被调函数，对其进行 clone，为向量化被调函数做准备。在向量化之前，需要对函数体内的代码进行一些准备性的转换。例如，被调函数中如果还含有循环，则需要对循环进行简化，以确保每个循环都只有一个入边和一个回边。

接着，对被调函数声明进行向量化，包括创建一个向量的函数名，以及对参数和返回值（如果有）进行向量化。然后，根据指令的操作数和操作符等信息，计算函数体内各个指令和变量的向量形状。

随后进行掩码生成，确定每个基本块的入口掩码和出口掩码。选择生成阶段将原始控制流图（CFG）中的 phi 函数替换为选择指令，将所有的掩码和选择操作插入到代码中，并按照特定的顺序对基本块进行拓扑排序。这样可以对控制流进行线性化，从而实现数据流有效编码。

函数体向量化代码的生成，即为标量到向量的一对一映射。对于不能向量化的标量代码，也要将其结果打包成向量。若不能向量化的标量存在发散分支，则还要加入级联块（Cascade blocks）来实现条件分支和掩码操作。

最后，对调用指令进行向量化，获取向量的参数值和向量化函数名，生成向量化调用指令。因调用指令或许并非被所有实例执行，故还要考虑到向量化函数调用的保护机制，以确保向量化函数的调用不会造成副作用。如果对于向量化函数调用指令，每个实例都执行或都不执行，那么可以生成更高效的代码，即在调用指令之前对其谓词进行规约。若规约结果为真，说明含有需要执行此基本块的实例，需要再检查其谓词是否统一为真（即全部需要执行）。若统一为真，则所有实例全部执行不需要进行掩码判断；若不统一为真，则需要根据掩码判断实例是否执行。若规约结果为假，则说明没有实例需要执行此基本块。如图 6 所示，图 6(a)给出了添加指导语句的 C 代码；图 6(b)给出了循环体的控制流图；图 6(c)给出了对图 6(b)中的 CFG 图进行 if 转换，将条件 cond 生成的掩码 m1 应用于 BB1 基本块的过程；图 6(d)中 Reduce 表示对掩码规约，通过 All 检测统一真谓词，BB1(nomask)基本块没有谓词；图 6(e)即为图 6(a)加入掩码规约和掩码统一真检测的向量化伪代码，其中 foo_simd 函数为 foo 函数的向量化函数，对于规约后掩码是统一真的，直接调用向量化函数 foo_simd，而不再需要根据掩码选择调用函数。

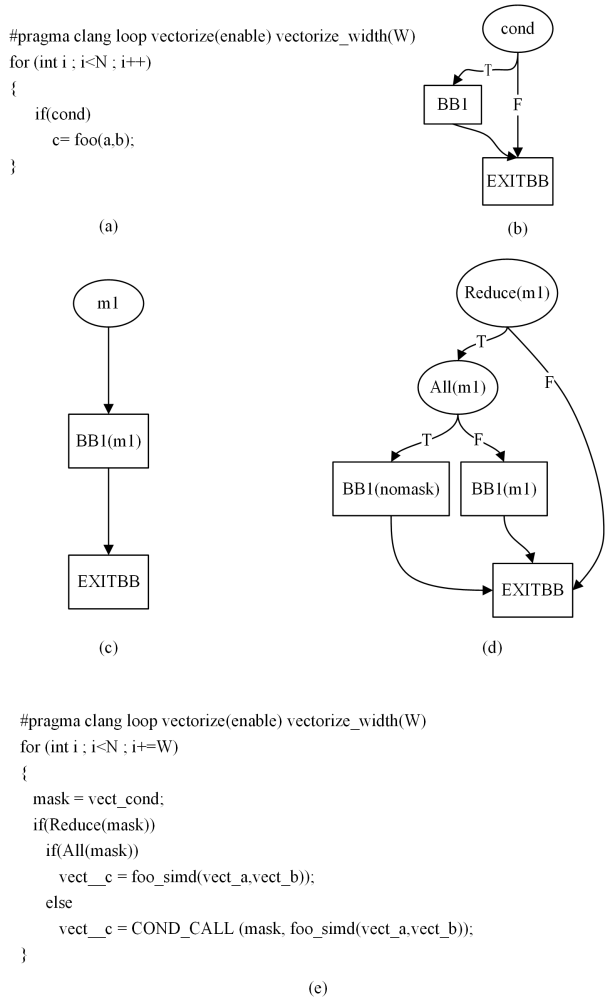

图 6 掩码规约和掩码统一真检测

Fig. 6 Mask reduction and mask uniform true detection

4 评估实验

本文在 LLVM16 编译器基础设施中对基于指导语句的函数向量化进行实现，从 ISPC 基准测试和 SIMD 库基准测试中选取了 10 个基准测试对其进行评估，如表 2 所列。ISPC（Intel SPMD Program Compiler）基准测试是[1]由 Intel 公司开发的一种针对 SIMD 指令集的基准测试套件。其主要针对图

像处理算法进行优化，包括图像滤波、缩放、旋转等操作。ISPC 基准测试中的程序都是基于真实世界的应用场景设计的，这样可以更好地评估不同向量编程技术在实际应用中的性能表现。Simd 库[2]（C++图像处理库）基准测试是由 Yermalayeu Ihar 开发的一种基准测试套件，它主要针对一些基本的数学运算进行优化。SIMD 基准测试包括了各种基本的数学运算，如加法、乘法、除法、浮点数运算、卷积、矩阵乘法、排序等。其优点在于涵盖了各种基本的数学运算，因此可以很好地评估自动向量化的性能。

实验在 Intel(R) Xeon(R) E5-2630 处理器上进行，CPU 主频为 2.20GHz，运行内存为 128GB，运行的操作系统为 Ubuntu Linux 22.04 64 位。在测试过程中，使用 clang 选项“-O3 -fno-vectorize -fno-slp-vectorize”关闭自动向量化，采用本文提出的基于指导语句的函数向量化方法进行向量化，而对比实验采用 clang 选项“-O3”进行编译，默认打开自动向量化。

实验结果如图 7 所示，在 10 个基准测试上与标量代码相比，本文提出的基于指导语句的函数向量化实现了 6.949 倍的平均加速比，而 clang -O3 选项打开向量化的平均加速比只有 2.928。AbsDifference 和 AbsGradientSaturatedSum 基准测试加速比高达 26.259 倍和 12.125 倍，通过对其源码进行分析，不难发现这两个基准测试中都含有嵌套循环，对于标量的顺序执行来说，需要花费很长时间来执行。而基于指导语句的函数向量化将控制流由数据流表示，节省了大部分时间。AbsGradientSaturatedSum 基准测试中还存在函数调函数再调函数的情况，虽然会产生一些调用开销，但函数之间传递的参数也是向量，返回值也是向量，中间不再需要向量转标量等转换，因此性能很好。而 Float32 基准测试含有多个同级循环，且循环中含有较多同级函数调用，产生了较多的开销，从而导致最终性能并没有显著提升。

表 2 选择的测试程序用例介绍

Table 2 Introduction of selected test program cases

测试用例	描述	来源
Lpb	计算图像的二进制编码	SIMD 库基准测试
Float32	8 位整数和单精度浮点数的转换	SIMD 库基准测试
Background	图像背景处理	SIMD 库基准测试
Interference	处理图像中的干扰	SIMD 库基准测试
AbsDifference	计算两个图像之间每个像素点的差异	SIMD 库基准测试
AbsGradientSaturatedSum	计算像素中梯度值进行饱和相加	SIMD 库基准测试
Mandelbrot	计算复杂二次递归方程的抽象数学	ISPC 基准测试
Options	股票期权估计	ISPC 基准测试
Aobench	使用着色器来计算遮挡衰减	ISPC 基准测试
Noise	Perlin 噪声算法	ISPC 基准测试

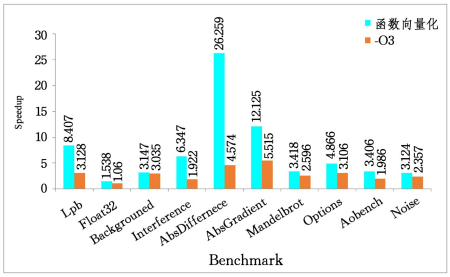

图 7 Benckmark 加速比图

Fig. 7 Benchmark speedup diagram

结束语 全函数向量化可以将整个函数的计算效率最大化，以更好地利用 SIMD 指令的优势，从而提高应用程序的性能。本文在部分控制流线性化实现的工具[2]基础上进行改进，提出了一种基于指导语句的函数向量化方法——基于指导语句的函数向量化。该方法主要包括解析指导语句确定向量化循环、预转换、向量化分析、掩码生成、选择生成、控制流图(CFG)线性化和向量化代码生成 7 个阶段。由于循环向量化时未将函数内联，因此需要单独对被调函数进行另外的处理。本文在 LLVM 编译器基础设施中实现了基于指导语句的函数向量化方法，并选取了 10 个基准测试来评估本文方法。实验结果表明，与标量代码相比，本文方法平均加速比达到了 6.949 倍。需要指出的是，本文方法已经在主流的 x86 平台上实现，如 Intel 和 AMD 处理器平台，但还存在一些局限性，其对于 ARM 架构与国产异构架构平台的兼容性尚不完善，如申威处理器和鲲鹏处理器。因此，接下来的工作重点将是移植此方法，以实现对国产异构架构和 ARM 架构的 CPU 的兼容性。

参 考 文 献

[1] KANDIAH V, LUSTIG D, VILLA O, et al. Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows [C]// Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization. 2023: 186-198.

[2] MOLL S, HACK S. Partial control-flow linearization[J]. ACM SIGPLAN Notices, 2018, 53(4): 543-556.

[3] RAPAPORT G, ZAKS A, BEN-ASHER Y. Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics[C]// Parallel & Distributed Processing Symposium Workshop. IEEE, 2015.

[4] TIAN X, SAITO H, GIRKAR M, et al. Compiling C/C++ SIMD extensions for function and loop vectorizaion on multicore-SIMD processors[C]// 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. IEEE, 2012: 2349-2358.

[5] MASTEN M, TYURIN E, MITROPOULOU K, et al. Func-

[1] https://github.com/ispc/ispc/releases/tag/v1.18.0https://github.com/ispc/ispc/releases/tag/v1.18.0

[2] https://github.com/ermig1979/Simd

tion/Kernel Vectorization via Loop Vectorizer[C]// Workshop on the LLVM Compiler Infrastructure in HPC. 2018.
[6] KARRENBERG R. Whole-function vectorization[C]// Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. 2015:141-150.
[7] GAO W, ZHAO R, HAN L, et al. SIMD automatic vectorization summary of compiler optimization [J]. Journal of Software, 2015, 26(6):1265-1284.
[8] FENG J, HE Y, TAO Q. Automatic vectorization, the recent progress and future [J]. Journal of communication, 2022(3):43.
[9] LARSEN S, AMARASINGHE S. Exploiting superword level parallelism with multimedia instruction sets[C]// Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. New York: ACM Press, 2000:145-156.
[10] PORPODAS V, ROCHA R C O, GÓES L F W. Look-ahead SLP: Auto-vectorization in the presence of commutative operations[C]// Proceedings of the 2018 International Symposium on Code Generation and Optimization. 2018:163-174.
[11] PORPODAS V, ROCHA R C O, BREVNOV E, et al. Super-Node SLP: Optimized vectorization for code sequences containing operators and their inverse elements[C]// 2019 IEEE/ACM International Symposium on Code Generation and Optimization(CGO). IEEE, 2019:206-216.
[12] FENG J, HE Y, TAO Q, et al. An SLP Vectorization Method Based on Equivalent Extended Transformation[J/OL]. https://onlinelibrary. wiley. com/doi/10. 1155/2022/1832522.
[13] ALLEN R, KENNEDY K. Automatic translation of Fortran programs to vector form[J]. ACM Transactions on Programming Languages and Systems, 1987, 9(4):491-542.
[14] ALLEN R, KENNEDY K, PORTERFIELD C, et al. Conversion of control dependence to data dependence[C]// Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New York: ACM Press, 1983: 177-189.
[15] BIK A J. The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance[M]. Intel Press, 2004.
[16] HAMPTON M, ASANOVIC K. Compiling for vector-thread architectures[C]// Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2008:205-215.
[17] NUZMAN D. loop aware SLP in GCC[C]// GCC Developers Summit. 2007.
[18] LI Y, GAO Y, WANG D, et al. Optimizations of the Whole Function Vectorization Based on SIMD Characteristics[C]// Parallel Architecture, Algorithm and Programming: 8th International Symposium(PAAP 2017). Haikou, China, Springer Singapore, 2017:152-171.

LIU Lili, born in 1999, postgraduate. Her main research interests include automatic vectorization of compilers and so on.

LI Yingying, born in 1984, Ph.D, associate professor, master supervisor. Her main research interests include high performance computing and advanced compilation techniques.

(责任编辑:何杨)