计算机科学
COMPUTER SCIENCE

一种基于指令MKS的自动向量化代价模型

王震，聂凯，韩林

引用本文

王震，聂凯，韩林. 一种基于指令MKS的自动向量化代价模型[J]. 计算机科学，2024，51(4)：78-85.

WANG Zhen, NIE Kai, HAN Lin. Auto-vectorization Cost Model Based on Instruction MKS[J]. Computer Science, 2024, 51(4): 78-85.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230200024

一种基于指令 MKS 的自动向量化代价模型

王 震[1] 聂 凯[2] 韩 林[2]
1 郑州大学计算机与人工智能学院 郑州 450000
2 郑州大学国家超级计算郑州中心 郑州 450000
(wangzcs@163.com)

摘 要 自动向量化代价模型是编译器进行自动向量化优化时的重要组成部分,其作用是评估代码在应用向量化转换后能否获得性能提升。当代价模型不准确时,编译器会应用负收益的向量化转换,从而降低程序的执行效率。针对 GCC 编译器默认代价模型的不精确问题,以 Intel Xeon Silver 4214R CPU 为平台,提出了一种基于指令 MKS 的自动向量化代价模型。该模型充分考虑了指令的机器模式、运算类型以及运算强度等,并使用梯度下降算法自动搜索不同指令类型的近似代价。在 SPEC2006 以及 SPEC2017 上进行了单线程测试,实验结果表明,该模型能够减少收益评估错误的情况。与默认代价模型生成的向量程序相比,GCC 编译器添加 MKS 代价模型后,在 SPEC2006 课题上最高获得了 4.72%的提速,在 SPEC2017 课题上最高获得了 7.08%的提速。

关键词:GCC 编译器;自动向量化;代价模型;收益评估;梯度下降
中图分类号 TP314

Auto-vectorization Cost Model Based on Instruction MKS

WANG Zhen[1], NIE Kai[2] and HAN Lin[2]
1 School of Computer and Artificial Intelligence, Zhengzhou University, Zhengzhou 450000, China
2 National Supercomputing Center in Zhengzhou, Zhengzhou University, Zhengzhou 450000, China

Abstract The auto-vectorization cost model is an important component of compiler's auto-vectorization optimization. Its role is to evaluate whether the code can achieve performance improvement after applying vectorization transformation. When the cost model is inaccurate, the compiler will apply vectorization transformation with negative benefit, thus reducing the execution efficiency of the program. Aiming at the inaccuracy of the default cost model of GCC compiler, based on Intel Xeon Silver 4214R CPU, an auto-vectorization cost model based on instruction MKS is proposed. The model fully considers the machine mode, operation type and operation intensity of instructions, and uses gradient descent algorithm to automatically search the approximate cost of different instruction types. Single-thread tests are carried out on SPEC2006 and SPEC2017. Experimental results show that the model can reduce the error of benefit estimation. Compared with the vector program generated by the default cost model, the GCC compiler, after adding the MKS cost model, achieves a maximum speedup of 4.72% on the SPEC2006 benchmark and 7.08% on the SPEC2017 benchmark.

Keywords GCC compiler, Auto-vectorization, Cost model, Profit evaluation, Gradient descent

1 引言

随着计算科学领域的快速发展,生物、化学以及物理等多个学科对高性能计算的需求日益旺盛[1],如何提升程序的运行效率变得越来越重要。在计算资源有限的情况下,充分应用以 MPI(Message-Passing Interface)为基础的进程级并行、以 OpenMP(Open Multi-Processing)为基础的线程级并行以及以 SIMD(Single Instruction Multiple Data)扩展部件为基础的数据级并行是行之有效的解决方法[2-3]。

数据级并行作为提升程序性能的重要手段,具有提高计算速度和效率、降低程序的内存访问开销和指令数量、提高处理器的吞吐量和资源利用率等优点,在图像处理[4]、生物医药[5]、人工智能[6]等领域有着广泛的应用。数据级并行的实现方式主要有两种:一种是通过程序员依据程序特征手工编写 SIMD 程序,另一种是通过编译器识别程序中可并行部分自动生成 SIMD 程序。工业界通常将前者称为手工向量化,

到稿日期:2023-02-04 返修日期:2023-06-13
基金项目:2022 年河南省重大科技专项(221100210600);22 求是科研启动(自)(32213247)
This work was supported by the Major Science and Technology Special Projects in Henan Province for 2022(221100210600) and 22 Qiushi Research Initiation(Natural Science)(32213247).
通信作者:韩林(hanlin@zzu.edu.cn)

将后者称为自动向量化。手工向量化通过内嵌手写向量汇编代码或编译器提供的向量接口函数来使用 SIMD 指令[7]，自动向量化通过编译器中的数据依赖关系分析、控制流转换等方法，将标量代码自动转换为向量代码[8]。随着计算机技术的飞速发展，研发人员编写了大量的科学计算程序。要让程序员通过手工向量化的方式将这些优秀的软件移植到新型处理器上，是一件难度很大且既费时又费力的事情。自动向量化相比手工向量化而言，避免了程序员手工改写或者编写高效向量代码带来的挑战，可以极大地减轻程序员的负担。作为 Linux 系统的标准编译器，GCC 编译器[9]从 4.1 版本开始，就实现了基于静态单赋值（Static Single Assignment，SSA）的自动向量化功能模块。

由于标量代码转换为向量代码的过程中可能需要引入较为耗时的额外操作，如向量拼接与拆分[10]、跨步访存优化[11]等，因此并非所有标量代码转换为向量代码后都能获得性能提升。因此，包括 GCC 编译器在内的大多数编译器在进行自动向量优化时都会进行收益评估。收益评估在编译器内部的实现思想是构造向量指令转换的代价模型，编译器依赖代价模型的输出结果来决定是否进行向量化转换，因此一个好的代价模型对编译器的自动向量化优化至关重要。当代价模型不准确时，编译器会进行负收益的向量化转换操作，实际上会降低应用程序的运行效率。

随着处理器行业的快速发展，与其配套的新的编译器不断涌现，传统的通过手工调整编译器参数对特定处理器进行移植优化的方法已不再适用[12-13]。在此背景下，基于机器学习的编译器调优技术逐渐流行。与手工调整参数相比，机器学习技术在减少开发人员工作量的同时，能够取得更好的运行时效果。

本文首先从多个角度分析并确定了 GCC8.3.0 编译器默认代价模型的不足之处，之后提出了一种基于指令 MKS 的代价模型，该模型充分考虑了指令的机器模式、运算类型以及运算强度等情况，并使用机器学习中的梯度下降算法自动搜索不同指令类型的近似代价。与默认代价模型相比，本文模型在 SPEC CPU2006 和 SPEC CPU2017 的课题上取得了有效的性能提升。

2　背景及相关工作

2.1　自动向量化代价模型

标量到向量转换是程序优化的重要手段之一，通过如图 1 所示的方式一次性同时操作一组数据，可以极大地提高程序的执行效率。

```
for(i=0; i<n; i++){
    a[i]=b[i]+c[i];
}
```

(a)标量执行

```
for(i=0; i<n; i+=8){
    a[i:i+8]=b[i:i+8]+c[i:i+8];
}
```

(b)向量执行

图 1　自动向量化执行示例

Fig. 1　Example of auto-vectorization execution

GCC 编译器支持两种主要类型的向量化操作[14]，分别是超字并行（Superword Level Parallelism，SLP）向量化以及循环级向量化（Loop Level Vectorization，LLV）。本文仅针对循环级向量化的代价模型进行研究，将 SLP 向量化的代码模型留待后续研究。在 GCC 编译器的循环级向量化中，一段标量代码进行向量转换需要满足 3 个必要的条件[15]：1）转换前后语义具有可观察等价性；2）生成处理器支持的 SIMD 指令；3）代价模型判断有收益。其执行流程如图 2 所示。

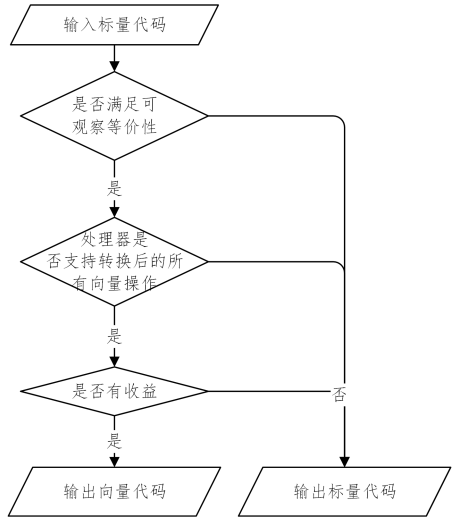


图 2　循环级向量化执行流程

Fig. 2　Workflow of loop-level vectorized execution

其中，代价模型的收益判断是进行自动向量化优化时的重要组成部分。当代价模型认为向量代码的执行时间比标量代码的执行时间长时，停止自动向量化优化。GCC 将循环级向量化分析中代码转换前后的代价划分为 4 个部分[16]，其具体含义如下：

1）标量迭代代价（Scalar Iteration Cost，SIC），代表转换前基本块（Basic Block，BB）内标量代码一次迭代内的代价总和。

2）向量迭代代价（Vector Iteration Cost，VIC），转换后 BB 内向量代码一次迭代内的代价总和。

3）标量外部代价（Scalar Outside Cost，SOC），标量循环需要版本化或为了使迭代数对齐时使用循环剥离而产生的额外代价。

4）向量外部代价（Vector Outside Cost，VOC），向量化后用于检查迭代次数、版本控制、程序间跳转以及使用掩码等操作而产生的额外代价，由向量头代价（Vector Prologue Cost，VPC）与向量尾代价（Vector Epilogue Cost，VEC）两部分组成。

$$SIC * vf > VIC \tag{1}$$

$$n * SIC + SOC > \frac{(n - npeel)}{vf} * VIC + VOC \tag{2}$$

计算完上述 4 部分代价之后，GCC 首先通过不等式（1）是否为真来判断一个循环是否值得进行向量化。如果为真，则进而计算能使不等式（2）为真的最小迭代数 n，之后通过插入运行时检查来决定程序在运行时执行标量代码还是向量代码。其中，vf 指向量化因子的大小，$npeel$ 指产生 VPC 与 VEC 时剥离的循环次数。

2.2　梯度下降

梯度下降算法[17]是机器学习中常用的解搜索算法之一。使用梯度下降算法求解函数的最小值需要以下步骤：

1）对函数中所有变量随机赋初值。

2)计算所有变量的梯度。

3)沿着梯度下降的方向移动一段距离。

4)重复步骤 2)和步骤 3),直到移动的距离小于设定的阈值或达到了最大迭代次数。

5)输出函数的最小值。

图 3 给出了使用梯度下降算法求解函数 $y=x^2$ 最小值的执行示例。图 3 中,灰色的点为初始点,黑色的点为迭代搜索中经过的点,红色的点为最终的结果。

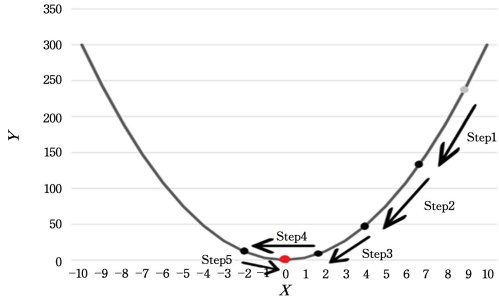


图 3 梯度下降算法执行示例(电子版为彩图)

Fig. 3 Example of gradient descent algorithm execution

目标函数、损失函数、梯度以及步长是梯度下降算法中最重要的 4 个概念。目标函数指模型所追求目标的计算方式,例如在线性回归模型中,目标函数等于将所有特征的特征值乘以权重之后再相加。损失函数是评估模型好坏的标准,反映模型预测值与实际值之间的误差。损失函数越小,代表模型参数越准确。对损失函数各个特征分别求偏导数后,能使偏导数降低的特征值修改方向即为梯度下降的方向。步长决定了每一次梯度下降的幅度,当步幅过小时,可能会陷入局部最优解;当步幅过大时,可能会跳过最优解。

2.3 TSVC 测试集

TSVC 测试集[18]是用于评估编译器自动向量化能力的一组测试套件,包括了数据依赖测试、语句重排测试、控制流测试以及归纳变量识别测试等。TSVC 测试集中共含有 151 个函数,每个函数都包含一个或多个嵌套循环,且不同函数以及同一函数内的每次循环迭代都是相互独立的。这使得可以直接通过对比同一函数的标量版本与向量版本的执行时间,来获得向量化后的实际加速比$S_{实}$。

现有的 TSVC 测试集仅能对 float 类型进行测试,本文为了检验编译器对不同数据类型的自动向量化能力,对 TSVC 测试集做出了如下修改:

1)删除了用于正确性验证的 checksum 函数。

2)修改了数据的初始化方法,使之可以分别对 int(32 bit),long long(64 bit),float(32 bit),double(64 bit)这 4 种数据类型进行动态初始化。

2.4 自动向量化代价模型的相关研究

自动向量化在现代计算机系统中扮演着不可或缺的重要角色,它具有提高计算效率、降低芯片功耗,以及提高代码的可读性、可移植性和兼容性等优点。自动向量化代价模型是编译器自动向量化优化的重要组成部分,通常有两种实现方法,分别是基于指令计数的评估方法以及基于机器学习的评估方法。

目前,基于指令计数的评估方法广泛地应用于开源编译器中代价模型的实现。例如,GCC 编译器为 Aarch64 架构处理器以及 Alpha 架构处理器实现的特定的代价模型[9]等。该方法具有计算简单、实现容易以及适用范围广等优点。

随着计算科学领域的快速发展,对自动向量化优化的要求也越来越高。在此背景下,基于指令计数的评估方法的不足逐渐显露出来,如精度有限、依赖特定指令集等。

为了解决上述问题,基于机器学习的评估方法逐渐流行。例如,Stock 等[19]为解决默认代价模型不准确的问题,使用多种机器学习技术分析汇编代码的相关特征,并指导向量代码的生成。但是该方法主要针对的是模板计算领域,在基准测试 SPEC 课题上的优化效果并未给出。Pohl 等[20]使用线性回归技术分析程序的中间表示特征,以此实现了一种具有改进的内存访问模式的自动向量化代价模型。但该模型仅仅适用于特定指令集,并不具有通用性。本文充分考虑了自动向量化优化过程中指令的机器模式、运算类型以及运算强度等,提出了一种基于指令 MKS 的自动向量化代价模型,经 SPEC 测试其具有更为广泛的适用范围。

3 默认代价模型存在的问题

本文使用 Intel Xeon Silver 4214R CPU 在修改后的 TSVC 测试集上对 GCC 编译器默认的自动向量化代价模型进行测试。

3.1 测试方法

在 TSVC 基准测试集上,分别对 int, long long, float, double 这 4 种数据类型运行以下 3 种测试。

1)标量测试。关闭自动向量化的-O3 优化,编译选项为:-O3 -mavx2 -fno-tree-vectorize。目的是获取所有函数的关闭向量化的运行时间。

2)开启默认代价模型测试。打开自动循环级向量化的-O3 优化代码,即编译选项为:-O3 -mavx2 -fno-slp-vectorize(去掉 SLP 向量化,本文仅研究循环级向量化)。目的是获取通过默认代价模型检查的可向量化函数的运行时间。

3)关闭默认代价模型测试。运行没有通过式(1)检查的函数,编译选项为:-O3 -mavx2 -fno-slp-vectorize -fvect-cost-model=unlimited。目的是获取关闭默认代价模型检查而直接进行循环级向量化的函数运行时间。

其中,-mavx2 指定编译器生成 AVX2 指令集的代码。之后通过 GCC 打印的中间优化信息,分析实际加速比与预测加速比之间的差异。

由于 AVX2 指令集所支持的向量宽度为 128bit 或 256bit,并且 TSVC 测试集中每个循环的最内层迭代次数已知,且全部能被支持的向量宽度整除,TSVC 测试集的向量化分析过程中不产生或产生很小的 SOC 与 VOC。因此,可以将预测加速比$S_{预}$简化为:

$$S_{预}=\frac{SIC*vf}{VIC} \tag{3}$$

TSVC 测试集中大部分函数的运行时间只有几秒钟,因此运行时间易受干扰。参考同行测试结果,多次调用同一函数,其运行时间会相差 5%左右[20]。为了排除平台系统的

波动性误差，约定当$S_{实}>1.05$时认为向量化有收益，当$S_{实}<0.95$时认为向量化带来了负加速。将代价模型判定为有收益，但实际执行时为负加速的情况称为假阳性（$S_{预}>1, S_{实}<0.95$）。将未通过代价模型检查，但实际向量化后有收益的情况称为假阴性（$S_{预}<1, S_{实}>1.05$）。

3.2　测试结果

GCC 共支持 80 余种 X86 下的 CPU 架构，如 AMD k8 架构、Intel Core i7 架构等，每一种架构都拥有独立的代价生成规则。其中 GCC 编译器在 Intel Xeon Silver 4214R CPU 上默认使用的是 GENERIC 架构规则。

图 4 给出了在修改后的 TSVC 测试集上分别对 int，long long，float，double 这 4 种数据类型进行基准测试的运行结果。通过分析中间优化信息，GCC 对于不同的数据类型分别识别出了 72，71，57，57 个可向量化函数。

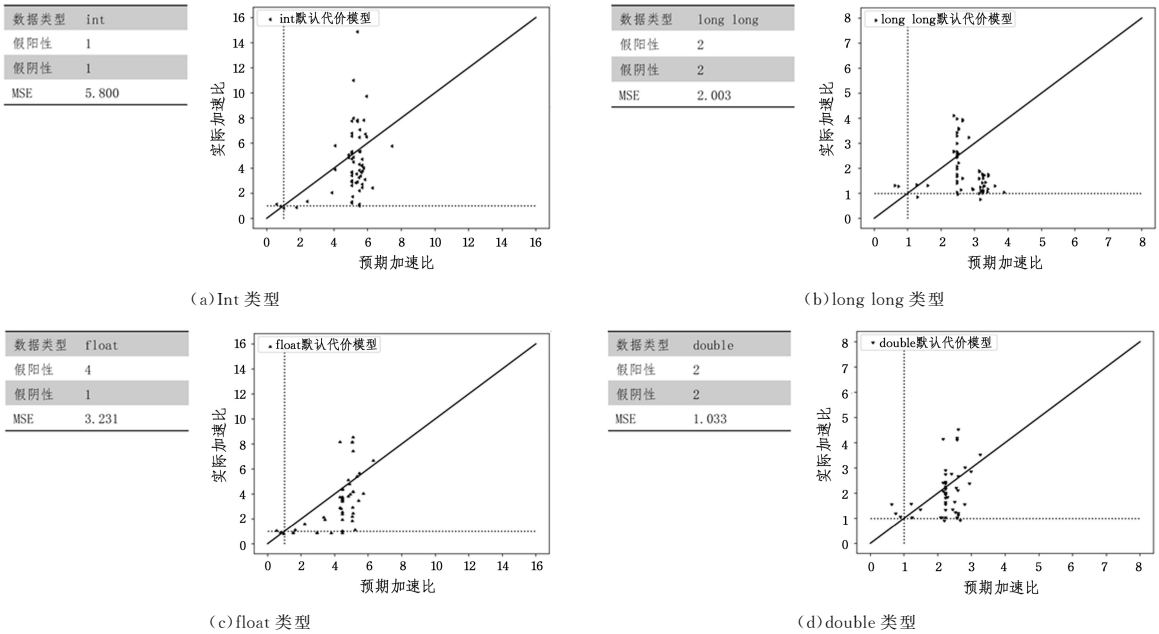


图 4　使用 GCC 默认代价模型时预测加速比与实际加速比对比

Fig. 4　Comparison of predicted and actual speedup using default cost model in GCC

由于向量宽度的限制，int 类型以及 float 类型的最大向量化因子大小 VF_{max} 为 8，long long 以及 double 的最大向量化因子大小 VF_{max} 为 4。当 $S_{实}>2*VF_{max}$ 时，认为出现了超线性加速，显然如此高的加速比已经超出向量化所能带来的加速，编译器可能在执行自动向量化优化后做了其他特定的优化，认为当 $S_{预}>2*VF_{max}$ 时属于预测错误。在排除超线性加速以及预测错误的情况后，4 种数据类型对应的 TSVC 测试集分别剩余了 69，68，54，54 个可向量化的函数。

图 4 中斜率为 1 的实线代表 $S_{实}==S_{预}$ 的完美预测，图中的点越接近该实线，代表代价模型的预测越准确。点落入虚线左上方虚线框住的区域代表假阴性，落入虚线右下方框住的区域代表假阳性。

使用预测加速比与实际加速比之间的均方值误差（Mean Square Error，MSE）作为评估代价模型好坏的标准，具体的计算式如式（4）所示：

$$MSE=\frac{\sum_{i=1}^{i=n}(S_{实}-S_{预})^2}{n} \tag{4}$$

其中，n 为排除超线性加速以及预测错误的情况后，不同数据类型剩余的可向量化函数的总数。

从图 4 可以看出，GCC 在当前 CPU 上使用的代价模型并不完善，在 TSVC 测试集上共出现了 15 次假阳性或假阴性预测以及较高的 MSE。出现这种情况的主要原因是 GENERIC 规则为了追求一定的普适性，并没有充分考虑不同 CPU 之间细微的硬件差异。

4　MKS 模型

为了降低假阳性与假阴性的出现概率以及预测与实际加速比之间的 MSE，本文提出了一种新的成本建模方法，称作“MKS 模型”。

4.1　建模流程

机器学习领域通常将问题划分为分类问题或回归问题，其中分类问题属于定性问题，回归问题属于定量问题。从是否将一个 BB 进行自动向量化转换的角度分析，MKS 模型属于分类问题；从 MKS 模型获取到 BB 的预测加速比，之后由程序中的其他逻辑判定是否进行向量化转换的角度看，MKS 模型属于回归问题。

当前工作的预期目标为辅助现有的代价模型进行收益判断，并不是开发一套完备的代价模型。因此，将 MKS 模型归类为回归问题，将应用向量化转换之后的加速比作为回归预测的输出值，将转换前与转换后指令的相关信息作为回归预测的输入值。

具体建模流程如图 5 所示，使用前文中的 int，long long，float，double 这 4 种数据类型对应的 TSVC 测试集的中间优化信息作为数据来源，相关的数据收集工作已于前文中完成。

图 5 MKS 建模流程

Fig. 5 MKS modeling process

4.2 数据集生成

MKS(mode, kind, subcode)三元组所代表的具体含义如下：

1)mode，指令的机器模式。当前版本的 GCC 共支持 VOIDmode，SImode，V4SImode 等 107 种机器模式。

2)kind，指令的语句类型。GCC 中内置的专门用于成本建模时使用的语句类型，包括 scalar_stmt，scalar_load，scalar_store，vector_stmt 等 14 种。

3)subcode，指令的操作子码。规定只有当指令类型为 scalar_stmt 或 vector_stmt，且为一个赋值操作时才有子码。根据 X86 架构的设计规则，部分子码应归为一类，如除法操作与取余操作、左移与右移操作等。归类之后的子码约有 8 种。

使用 MKS 三元组唯一确定一种指令类型，将不同的指令类型作为输入数据的不同特征，每种特征拥有唯一的指令开销 W_i。使用当前语句对应的指令重复次数 $count$、向量化因子大小 vf 唯一确定数据集中的一条记录。为了计算的简便性，将数据集中前 m 个特征归类为 SIC，将后 n 个特征归类为 VIC。

对于每一种数据类型对应的 TSVC 测试集，分别进行特征提取，并将 TSVC 测试集中的每一个可向量化函数对应为数据集中一条独立的记录。

具体的数据集生成方法如算法 1 所示，主要思想是通过遍历所有函数的标量版本与向量版本中所有指令的方式获取对应特征的特征值，并最终返回数据集中一条完整的记录。

算法 1 数据集生成

输入：func_BB/ * 某一函数在进行向量化分析时的各种信息 * /

输出：features，$S_{实}$ / * 数据集中的一条记录 * /

```
1. func dataset_generation(func_BB)
2.   初始化长度为 m+n 的数组 features
3.   for instruction in func_BB
4.     获取 instruction 的参数信息
5.     获取该指令类型在 MKS 集合中的下标 index
6.     if index 属于 SIC then
7.       features[index] += vf * count
8.     else
9.       features[index] += count
10.    end if
11.  end for
12.  计算该函数的实际加速比 S实
13.  return features, S实
14. end func
```

4.3 模型构建

式(5)用来计算 MKS 模型输出的预测加速比 S_{MKS}，其中 m 和 n 分别代表 BB 中属于 SIC 的个数以及属于 VIC 的个数。使用 S_{MKS} 与 $S_{实}$ 的 MSE 作为梯度下降的损失函数，即评判模型好坏的标准。

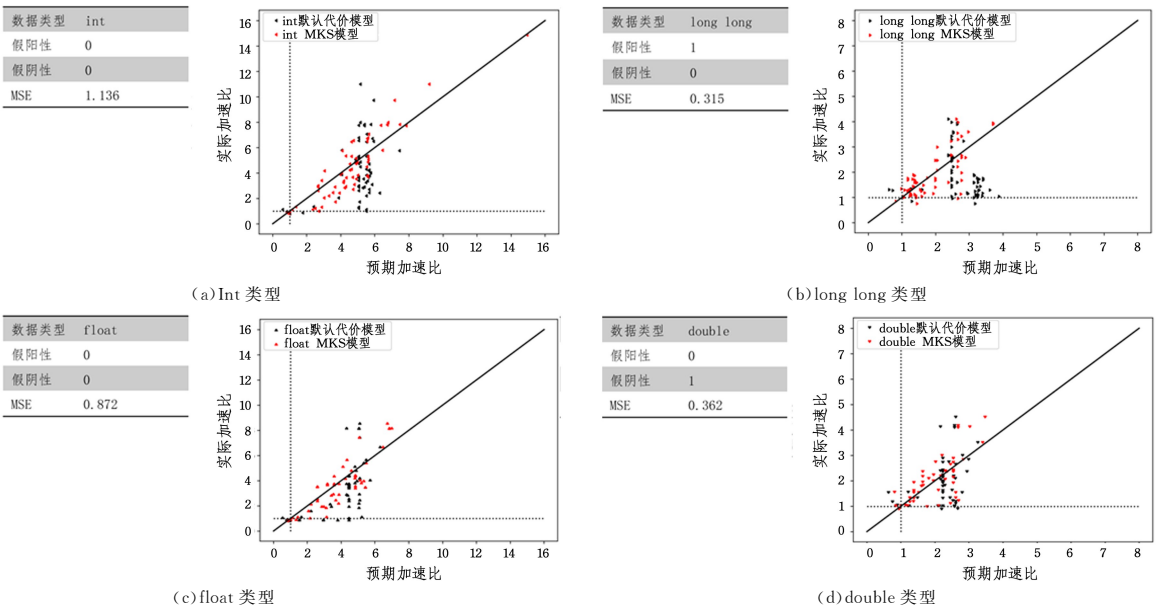
$$S_{MKS}=\frac{\sum_{i=1}^{i=m}W_i * vf * count}{\sum_{i=m}^{i=m+n}W_i * count} \tag{5}$$

由于 MKS 模型无须考虑泛化性，因此使用数据集中的所有数据进行训练与验证。多次训练后，将能使数据集 MSE 最小的指令开销与对应的指令类型保存。

为了防止模型出现不符合实际的过拟合现象，如指令开销为负等，规定不同指令类型的开销 $W_i \geqslant 1$。为了防止使用梯度下降算法时陷入局部最优解，将初始步长设置为 0.1，之后每 100 次迭代将步长降低至原先的 1/10。

4.4 模型评估

图 6 给出了 MKS 代价模型下 int，long long，float，double 类型对应的 TSVC 测试集预测与实际加速比的对比结果。其中，黑色的点属于默认代价模型，红色的点属于 MKS 模型。



(a)Int 类型 (b)long long 类型

(c)float 类型 (d)double 类型

图 6 使用 MKS 代价模型时预测加速比与实际加速比的对比(电子版为彩图)

Fig. 6 Comparison of predicted and actual speedup using MKS cost model

对比图 4 和图 6 可知，与默认代价模型下生成的结果相比，应用 MKS 模型后每种数据类型对应的预测与实际加速比之间的 MSE 都大幅降低。其中 int 类型由原来的 5.800 降低至 1.136，long long 类型由原来的 2.003 降低至 0.315，float 类型由原来的 3.231 降低至 0.872，double 类型由原来的 1.033 降低至 0.362。同时，对于不同的数据类型，MKS 模型将假阳性与假阴性总的出现次数从原先的 15 次降低至 2 次。表 1 列出了建模前后不同数据类型指令开销变化最大的几种指令类型，其中 subcode 只列出了部分操作。

表 1　各数据类型建模前后指令开销变化的对比

Table 1　Comparison of instruction cost changes before and after modeling for each data type

数据类型	mode	kind	subcode	默认代价	MKS 模型代价
int	V8SI	vector_stmt	+ −	4	37.9
int	V8SI	vec_perm	null	4	33.8
int	V8SI	scalar_stmt	+ −	4	25.8
int	V8SI	vec_to_scalar	null	4	25.7
long long	TI	scalar_stmt	null	4	44.2
long long	V4DI	vector_stmt	&. |	4	16.9
long long	DI	scalar_stmt	+ −	4	1.1
long long	V4DI	vector_load	null	20	4.9
float	SF	scalar_stmt	*	16	1.0
float	SF	scalar_store	null	12	1.0
float	V8SF	vector_stmt	*	16	1.8
float	V8SI	vector_stmt	+ −	4	1.1
double	DF	scalar_stmt	*	20	1.1
double	V4DF	vector_load	null	20	1.1
double	V4DF	unaligned_store	null	20	140.9
double	V4DF	vector_stmt	null	12	61.4

4.5　模型实现

算法 2 是 MKS 模型下的 BB 代价计算方法，通过遍历给定 BB 中的所有指令的方式，获取不同指令类型在 MKS 模型中对应的代价。算法约定 BB 中出现未知的指令类型时，该 BB 的代价为 0。

算法 2　MKS 模型的代价计算方法

```
输入：向量化前的 BB 或向量化后的 BB
输出：BB 的代价
1. func compute_costs(BB)
2.     costs=0
3.     for instruction in BB
4.         获取 instruction 的参数信息
5.         获取该指令类型在 MKS 集合中的下标 index
6.         if index==-1 then
7.             return 0
8.         end if
9.         根据 index 获取 mks 的权重 weight
10.            获取 BB 的类型 BB_type
11.            if BB_type==SCALAR then
12.                costs+=count * vf * weight
13.            else
14.                costs+=count * weight
15.            end if
16.    end for
17. end func
```

MKS 模型的具体实现方法为，通过算法 2 分别计算标量 BB 的代价与对应的向量 BB 的代价。当标量 BB 的代价与向量 BB 的代价均不为 0 时，返回两者相除的值，否则返回 0。约定只有出现以下两种情况时，MKS 模型才会生效。

1）默认代价模型判定无收益，但 MKS 模型返回值大于 1，此时强制进行向量化。

2）默认代价模型判定有收益，但 MKS 模型返回值小于 1 且不等于 0，此时阻止向量代码的生成。

4.6　复杂性分析

算法 1 和算法 2 操作的数据结构都是编译优化过程中产生的 BB。在 GCC 中，函数的 BB 都是一种线性有限的语句序列。因此，算法 1 和算法 2 在最坏的情况下将执行 $O(M)$ 步，其中 M 为 BB 中语句的数量。

因此，算法 1 和算法 2 的执行时间与 BB 的大小成正比。依据编译过程中 BB 的划分算法，通常 BB 内包含的语句序列较少。因此，这两种算法通常可以在较短时间内结束执行。

5　实验评估

本文使用 SPEC2006 和 SPEC2017 测试集对 MKS 模型进行测试，使用的编译器为 GCC 8.3.0，操作系统为 CentOS 8.5，CPU 型号为 Intel Xeon Silver 4214R。

5.1　测试集与测试内容

SPEC2006 和 SPEC2017 是世界公认的 CPU 基准测试套件之一，其中 SPEC2006 包含 12 个整数（Integer）基准、17 个浮点（Floating Point）基准以及 2 个用于验证 CPU 随机数生成功能是否正确的基准。SPEC2017 包含 10 个 Integer rate 基准、10 个 Integer speed 基准、13 个 Floating Point rate 基准以及 10 个 Floating Point speed 基准。

在测试规模上，SPEC2006 与 SPEC2017 均支持 3 种规模的测试，分别是 test，train 以及 ref，分别为用于测试结果正确性的数据集、用于结果反馈优化的测试集以及默认使用的标准数据集。

针对同一循环形式，不同的代价模型可能会做出不同的收益评估，进而决定是否进行向量化转换。因此，在编译以及运行条件相同的情况下，可以直接通过对比同一程序在不同代价模型下执行时间的方式，来判断代价模型的优劣。

本文使用 SPEC2006 中 ref 规模的 Integer 基准和 Floating Point 基准以及 SPEC2017 中 ref 规模的 Integer speed 基准和 Floating Point speed 基准作为测试集验证 MKS 模型的有效性。使用相同的编译选项（-O3 -mavx2）以及运行条件（单线程运行），对测试集分别进行默认代价模型以及 MKS 模型的测试。

5.2　结果与分析

表 2 列出了采用 MKS 模型前后，在 SPEC 课题上的运行时间对比。

采用 MKS 模型后在 SPEC2006 和 SPEC2017 的部分课题获得了速度提升，其中 SPEC2006 中 482.sphinx3 课题的提速达 4.72%，SPEC2017 中 620.omnetpp_s 课题的提速达 7.08%。所有 SPEC 课题中没有出现向量加速提升为负的情况。

表 2 MKS 模型部署前后在 SPEC 测试集上的运行时间对比

Table 2 Comparison of runtime on SPEC test suite before and after deployment of MKS model

程序名称	MKS 模型移植前/s	MKS 模型移植后/s	速度提升
433. milc	393.0	381.0	+3.05%
450. soplex	172.0	166.0	+3.49%
453. povray	100.0	96.8	+3.20%
482. sphinx3	424.0	404.0	+4.72%
403. gcc	209.0	202.0	+3.35%
456. hmmer	142.0	139.0	+2.11%
483. xalancbmk	145.0	142.0	+2.07%
603. bwaves_s	3176.0	3021.0	+4.88%
654. roms_s	4024.0	3796.0	+5.67%
620. omnetpp_s	424.0	394.0	+7.08%
631. deepsjeng_s	400.0	374.0	+6.50%

通过打印编译优化中间表示信息，选取了如图 7 所示的两种具有代表性的循环形式，其中变量 a 是一个结构体数组，并且该结构体内至少存在 3 个及以上的 int 类型的成员变量。在 SPEC 测试集上获得速度提升的原因有以下两点：

1)阻止部分向量代码的生成。如图 7(a)所示，GCC 默认代价模型认为该代码计算强度较高，在向量化后有一定的速度提升，因此决定进行向量化转换。但 MKS 模型认为该循环存在较高的访存开销，无法产生正收益，因此阻止向量代码的生成。实验结果证明，不使用向量指令的汇编代码具有更高的运行效率。

2)强制对部分代码进行向量化。如图 7(b)所示，GCC 默认代价模型认为该代码计算强度较低，并且向量拼接和拆分过程需要产生大量的访存开销，导致向量化后无法获得收益，因此阻止进行向量化转换。但 MKS 模型认为该循环以固定跨步对结构体元素赋值，向量拼接和拆分产生的开销较小，因此强制进行向量化。实验发现，使用向量指令的代码运行效率更高，因此提升了运算速度。

```
for(i=0; i<n; i++){
    temp=a[i].para2+a[i].para3;
    a[i].para1+ = temp;
}
```

(a)阻止该类型的向量化

```
for(i=0; i<n; i++){
    a[i].para1=1;
}
```

(b)对该类型进行强制向量化

图 7 MKS 模型生效的代码形式

Fig. 7 Code form with MKS model in effect

图 7(a)类似形式的循环在 SPEC2006 中的 433. milc，450. soplex 等程序以及 SPEC2017 中的 602. gcc_s，631. deepsjeng_s 等程序中均有出现。图 7(b)类似形式的循环在 SPEC2006 中的 403. gcc，450. soplex 等程序以及 SPEC2017 中的 600. perlbench_s，620. omnetpp_s 等程序中均有出现。

分别对图 7(a)和图 7(b)两种形式的代码进行多种结构体大小以及多种迭代长度下的手工测试。与标量执行时间相比，图 7(a)向量化后的执行时间平均增加了 21.5%，图 7(b)向量化后的执行时间平均减少了 24.9%。该结果进一步验证了 MKS 模型的有效性。

结束语 本文针对 GCC 编译器自动向量化代价模型不精确的问题，提出了一种基于指令 MKS 的成本建模方法，用于辅助 GCC 默认代价模型，并进行了有效性验证。该模型充分考虑了指令在执行时的机器模式、访存方式以及运算强度等，并使用梯度下降算法自动搜索不同指令类型的近似代价。实验结果表明，基于指令 MKS 的代价模型能够有效提高编译器在自动向量化阶段的收益分析能力。

MKS 模型依赖指令的类型计算代价，只有当 BB 内所有指令的指令类型均出现在现有 MKS 集合中时，才能够准确进行辅助判断。当前 4 种数据类型对应的 TSVC 测试集仅覆盖了 82 种不同的指令类型，下一步将继续扩充并完善现有的 MKS 集合，以获取适用性更广泛 MKS 模型。

参 考 文 献

[1] JIN Z, LU Z H, LI H Y, et al. Origin of High Performance Computing—Current Status and Developments of Scientific Computing Applications[J]. Bulletin of Chinese Academy of Sciences, 2019, 34(6): 625-639.

[2] RABENSEIFNER R, HAGER G, JOST G. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes[C]// 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing. IEEE, 2009: 427-436.

[3] WENDE F, MARSMAN M, ZHAO Z, et al. Porting VASP from MPI to MPI+ OpenMP [SIMD][C]// International Workshop on OpenMP. Cham: Springer, 2017: 107-122.

[4] HUA Z, ZHANG K, LI Y, et al. Visually secure image encryption using adaptive-thresholding sparsification and parallel compressive sensing[J]. Signal Processing, 2021, 183: 107998.

[5] HAUTANIEMI S, LAAKSO M. High-performance computing in biomedicine[C]// 2013 International Conference on High Performance Computing & Simulation (HPCS). IEEE, 2013: 233-233.

[6] TANG Y, WANG C. Performance modeling on DaVinci AI core [J]. Journal of Parallel and Distributed Computing, 2023, 175: 134-149.

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

[8] NUZMAN D, HENDERSON R. Multi-platform auto-vectorization[C]// International Symposium on Code Generation & Optimization. IEEE, 2006.

[9] Free Software Foundation, Inc. GCC, the GNU compiler collection [EB/OL]. (2022-12-23). https://gcc.gnu.org/.

[10] TAN H, CHEN H, SHENG L, et al. Modeling and evaluation for gather/scatter operations in Vector-SIMD architectures [C]// 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2017.

[11] HARPER III D T,LINEBARGER D A. Conflict-free vector access using a dynamic storage scheme[J]. IEEE Transactions on Computers,1991,40(3):276-283.

[12] LEATHER H,CUMMINS C. Machine learning in compilers: Past,present and future[C]// 2020 Forum for Specification and Design Languages(FDL). IEEE,2020:1-8.

[13] ASHOURI A H,KILLIAN W,CAVAZOS J,et al. A survey on compiler autotuning using machine learning[J]. ACM Computing Surveys(CSUR),2018,51(5):1-42.

[14] SUI Y,FAN X,ZHOU H,et al. Loop-oriented pointer analysis for automatic simd vectorization[J]. ACM Transactions on Embedded Computing Systems(TECS),2018,17(2):1-31.

[15] FENG J G,HE Y P,TAO Q M. Auto-vectorization: recent development and prospect[J]. Journal on Communications,2022,43(3):180-195.

[16] NAISHLOS D. Auto vectorization in GCC[C]// Proceedings of the 2004 GCC Developers Summit. 2004:105-118.

[17] RUDER S. An overview of gradient descent optimization algorithms[J]. arXiv:1609. 04747,2016.

[18] MALEKI S,GAO Y,MJ GARZARÁN,et al. An Evaluation of Vectorizing Compilers[C]// International Conference on Parallel Architectures & Compilation Techniques. IEEE,2015.

[19] STOCK K,POUCHET L N,SADAYAPPAN P. Using machine learning to improve automatic vectorization[J]. ACM Transactions on Architecture and Code Optimization(TACO),2012,8(4):1-23.

[20] POHL A,COSENZA B,JUURLINK B. Vectorization cost modeling for NEON,AVX and SVE[J]. Performance Evaluation,2020,140:102106.

WANG Zhen, born in 1999, postgraduate. His main research interests include compiler optimization and high-performance computing.

HAN Lin, born in 1978, Ph.D, associate professor, is a senior member of CCF (No. 16416M). His main research interests include compiler optimization and high-performance computing.

(责任编辑:喻藜)