计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications [doi: 10.15888/j.cnki.csa.008686]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

GCC 非满载 SLP 向量化①

刘浩浩, 韩　林, 崔平非

(中原工学院 前沿信息技术研究院, 郑州 450007)
通信作者: 韩　林, E-mail: strollerlin@163.com

摘　要: 随着向量长度的不断增长, SIMD 扩展部件得以处理更为庞大的数据级并行, 但程序的并行阈值也随之提高. 对于现有的自动向量化编译器, 如果在分析阶段不能从串行代码中发掘出足够的数据级并行以完全填充向量寄存器, 则不会进入相应的向量代码变换阶段, 从而无法向量化. 较长的向量长度使得某些并行性不足的程序失去了向量化的机会, 造成了性能下降. 为了更加充分的利用 SIMD 部件, 介绍了一种面向基本块的非满载向量化方法 ISLP. 基于开源 GCC 编译器, 从并行性检测、代码生成和代价模型 3 个方面详细阐述了 ISLP 的设计与实现. 在标准测试集上的实验结果表明, 该方法可以有效地对超字级并行性不足的程序进行向量化处理, 提高程序执行效率. 选取的测试用例在向量化后的平均加速比达到 1.14, 性能较常规 SLP 方法提升 11.8%.

关键词: GCC; SIMD 扩展; 非满载向量化; 超字级并行性; 代码生成; SLP

引用格式: 刘浩浩,韩林,崔平非.GCC 非满载 SLP 向量化.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/8686.html

Insufficient SLP in GCC

LIU Hao-Hao, HAN Lin, CUI Ping-Fei

(Research Institute of Frontier Information Technology, Zhongyuan University of Technology, Zhengzhou 450007, China)

Abstract: With the increase in vector length, SIMD extension can deal with more huge data level parallelism, but the parallelism threshold of the program also increases. For the current auto-vectorization compiler, if enough data level parallelism can not be found from the scalar code to completely fill the vector register in the analysis stage, it will not enter the vector code transformation stage, and vectorization cannot be achieved. The improvement of vector length makes some programs with insufficient parallelism lose the opportunity of vectorization, resulting in performance degradation. To make full use of SIMD components, this study introduces a basic block oriented insufficient vectorization method ISLP. Based on the GCC compiler, the design and implementation of ISLP are described in detail from three aspects: parallelism detection, code generation and cost model. Experiments on the standard test set show that this method can effectively vectorize the program with insufficient super-word level parallelism and improve the program execution efficiency. The average speedup ratio of the selected test cases after vectorization reaches 1.14, and the performance is 11.8% higher than that of the conventional SLP method.

Key words: GNU compiler collection (GCC); SIMD extension; insufficient vectorization; superword level parallelism; code generation; SLP

SIMD 扩展[1] 作为一种重要的并行加速部件已经被包括数字信号处理器, 游戏机, 通用处理器以及超级计算机在内的大多数现代计算设备所采用. SIMD 扩展指令集允许将多个数据元素上的同构标量操作转换为

① 收稿时间: 2021-12-09; 修改时间: 2022-01-07; 采用时间: 2022-01-27; csa 在线出版时间: 2022-06-16

对应数据元素合集上的向量操作. 这种并行计算方式极大提高了数据吞吐量, 被广泛的应用于数字信号处理、多媒体应用、科学计算等多个领域. 随着 SIMD 部件的发展, SIMD 扩展的向量长度不断增加. 以 intel x86 处理器 SIMD 扩展为例, 从 MMX 开始, 历经 SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX512, 向量长度由最初的 64 位增加至 512 位[2–8]. 国产处理器的 Matrix 向量指令集达到了 1 024 位[9]. ARM 自 ARMv6-A 开始支持 SIMD, 最初的向量宽度仅为 32 位, 可同时计算 2 个 16 位或 4 个 8 位操作数. ARMv7-A 和 ARMv8-A 的向量长度分别是 64 位和 128 位. 现在可伸缩向量扩展 (scalable vector extension, SVE) 最长可支持 2 048 位向量长度[10]. 向量长度的增加提高了 SIMD 扩展部件的并行处理能力, 但同时也增加了组成向量发掘数据级并行的难度. 对于现有的自动向量化编译器, 如果在分析阶段不能从串行代码中发掘出足够的数据级并行性以完全填充向量寄存器, 则不会进入相应的向量代码变换阶段, 从而无法向量化. 因此, 在向量部件长度较大时, 如何实现并行性不足程序的向量化是亟待解决的问题.

目前已经出现了一些针对上述问题的研究: 结合向量长度不可知 (vector length agnostic, VLA) 编程模型[11], ARM SVE 允许向量程序在任一向量长度的 SVE 实现上运行, 文献 [12] 研究了面向 ARM SVE 的快速傅里叶变换 (FFT) 算法的向量化实现. 通过增强向量指令集的功能充分发挥较长向量长度的优势, 文献 [13] 提出了通道级并行性 (lane level parallelism, LLP) 的概念, 研究了点积运算向量化在 AVX-512 VNNI 指令集上的高效实现.

通过构造冗余数据模拟向量寄存器的满载使用可以对并行性不足的程序进行向量化. 文献 [14] 介绍了一种对 3 路标量操作的 4 通道向量化方法 PAVER (partial vectorizer), 并在 LLVM 中予以实现. 文献 [15] 对向量寄存器的非满载使用方式进行了研究, 指出可以使用常规向量存储与数据重组相结合的方式实现非满载的向量存储操作. 提出了一种面向对齐剥离的循环向量化框架, 使用非满载的向量化方法对剥离产生的头、尾循环进行向量化. 文献 [16] 给出向量并行度的概念及其计算方法. 使用向量并行度作为指导, 为具有不同并行特征的循环选择合适的向量化方法. 文献 [17] 指出在使用掩码指令对并行性不足的程序进行向量化时, 可以考虑对掩码加载操作进行外提以减少向量化开销.

GCC (GNU compiler collection) 是 GUN 下的编译器合集. GCC 实现了两种形式的向量化器, 即循环向量化器 (loop vectorizer) 和基本块向量化器 (basic block vectorizer), 它们分别具有独立的入口[18]. 基本块向量化器使用 SLP 方法发掘直线型代码中的超字级并行性[19]. 本文介绍了一种面向基本块的非满载向量化方法 ISLP (insufficient SLP), 旨在完成超字级并行性不足程序的向量化. 在深入分析了 GCC 7.1.0 的 SLP 向量化框架后, 详细探讨了 ISLP 在 GCC 中的实现, 包括非满载 SLP 计算树的构建, 基于数据重组策略的代码生成和用于收益评估的代价模型. 最后在标准测试集上对扩展后的 SLP 框架进行了评估. 本文的主要贡献包括:

(1) 介绍了一种面向基本块的非满载向量化方法 ISLP.

(2) 阐述了 ISLP 在 GCC 中的设计与实现.

本文的章节安排如下. 第 1 节介绍非满载 SLP 向量化的概念; 第 2 节对 GCC 7.1.0 的基本块向量化器进行分析; 第 3 节从并行性检测, 代码生成, 代价模型 3 个方面描述对非满载 SLP 向量化的支持.

1 非满载 SLP 向量化

向量化将多个数据元素上相互独立的同构标量操作转换为对应数据元素合集上的向量操作, 从而实现对多个数据元素的并行处理. 典型编译器如 GCC、LLVM、ICC, 都实现了自动向量化功能以完成从标量程序到向量程序的自动变换. 目前的自动向量化方法仅支持向量寄存器的满载使用. 满载使用是指在向量操作中向量寄存器中的每个数据都是有效的, 从向量装载到向量计算, 再到向量存储, 一系列的操作都要保证寄存器中每个数据的有效性. 向量寄存器的满载使用方式要求自动向量化方法能够从程序中发掘出足够的数据级并行, 以完全填充向量寄存器. 当程序中的数据级并行性不足时, 由于无法完全填充向量寄存器, 现有的自动向量化方法无能为力, 为此提出了基于向量寄存器非满载使用的向量化方法. 在非满载的向量化框架下, 向量寄存器处于一种非满载的状态, 即在向量寄存器中只有部分槽位存放着原标量程序所定义的有效数据, 其余槽位是无效的冗余数据.

SLP 方法用于发掘直线型代码中的超字级并行性, 超字级并行性的检测通过检索基本块中的同构语句来

完成. SLP 方法将同一基本块内的 *VF* 条可并行执行的同构语句进行打包并替换为相应的向量语句. *VF* 称之为向量化因子 (vector factor), 表示处理器的向量部件在一次向量操作中所能并行处理的数据元素的个数. 当同构语句数目小于 *VF* 时, 由于可并行处理的数据不足以完全填充向量寄存器, SLP 方法拒绝执行向量化. 此外, 由于循环展开可以将向量并行性转换为超字级并行性[19], 一些经过展开的循环结构也可以使用 SLP 方法进行向量化.

非满载 SLP 方法用于完成超字级并行性不足程序的向量化. 在超字级并行性不足的情况下, 基本块内的可并行同构语句数目小于 *VF*, 待处理的数据元素无法完全填充向量寄存器, 需要使用某种方法构造冗余数据对向量寄存器中的剩余槽位进行填充. 在向量加载阶段, 有效数据和无效数据一起从内存被加载至向量寄存器. 无效数据和有效数据一起参与计算. 在写回内存时, 有效数据被存储到相应的内存位置, 而无效数据则不会被存储. 非满载 SLP 向量化维持程序在向量执行后的计算状态的正确性.

2 GCC 中的 SLP 向量化

非满载 SLP 向量化是基于 GCC 开源编译器实现的, 本节概述 GCC 中的 SLP 向量化框架. 清单 1 展示了 SLP 向量化的流程, vect_slp_analyze_bb_1 完成对基本块内超字级并行性的分析, vect_schedule_slp 负责向量代码的生成.

向量化器首先完成基本块内数据引用的收集. 借助于标量演化分析器, vect_analyze_data_refs 为每一个数据引用构建访问函数. 访问函数将被用于依赖分析、访存模式和对齐分析.

清单 1. SLP 框架

```
vect_slp_analyze_bb_1
  vect_analyze_data_refs
  vect_analyze_data_ref_accesses
  vect_analyze_slp
  vect_slp_analyze_and_verify_instance_alignment
  vect_slp_analyze_instance_dependence
  vect_slp_analyze_operations
  vect_bb_vectorization_profitable_p
vect_schedule_slp
  vect_schedule_slp_instance
    vect_schedule_slp_instance
    vect_transform_stmt
```

vect_analyze_data_ref_accesses 负责数据引用的访存模式分析, 交错链 (interleaving chains) 和组访问 (group access) 的识别与建立. 交错链由基址具有线性关系、跨幅相同、数据类型相同的数据引用所组成, 根据数据引用的读写类型又可分为交错加载 (interleaving load) 和交错存储 (interleaving store). 特别的, 当数据引用的访存模式为非连续访问且基址相邻时, 这些互相独立的内存访问构成了一种特殊的访存模式, 称之为组访问, 对应的交错链称为组加载 (group load) 或组存储 (group store). 组访问实际上是对一块连续内存地址空间的访问. 每一个 group store 都将作为一个独立的 SLP 种子参与后续超字级并行性的检测.

vect_analyze_slp 检测基本块内的超字级并行性并建立 SLP 实例 (SLP instance). SLP 实例被定义为一颗 SLP 计算树 (SLP computing tree), 树的每一个节点都包含一组同构的标量语句, 根节点为 group store, 叶子节点为加载类型的同构语句, 父子节点通过操作数之间的定义使用链进行关联. 所谓同构性指的是节点内的其他语句同构于第一条语句, 即它们具有相同的操作类型和操作数, 并且保持相同的操作顺序.

SLP 计算树的构建主要包括同构语句的检索和打包. 从 group store 开始, 根据定义使用链递归地检索同构语句. 每获得一组标量语句, 首先检查数据引用类型和标量语句的同构性, 更新最小数据类型信息 *max_nunits*, *max_nunits* 表示 SLP 计算树中最小数据类型对应的向量化因子. 不同构的标量语句无法进行打包, 当遇到一组不同构的标量语句时停止搜索并返回. 数据类型的检测包括: (1) 检测三地址码中的最小数据类型, 最小数据类型将用于计算展开因子; (2) 检查最小数据类型对应的向量类型是否被目标机支持, 不被支持的数据类型无法向量化; (3) 检查最小数据类型下的向量寄存器是否满载, 若同构语句的数目小于向量化因子, 无法并行执行.

同构性和数据类型检测通过后, 尝试对标量语句进行打包. 首先考虑同构语句是否为加载类型, 加载类型的同构语句被视为 SLP 计算树的叶子节点, 打包后立即返回. 对非加载类型的同构语句进行打包前, 要首先遍历它的孩子节点, 对它的孩子节点完成打包. 这是因为父节点的操作数依赖于孩子节点. 当所有的孩子节点打包完成后, 返回至父节点进行打包. 根节点的 group store 打包后, 一棵完整的 SLP 计算树就被建立起来.

递归过程结束后对 SLP 计算树进行初步的验证, 主要包括: (1) 是否需要循环展开, 在循环级和基本块级相结合的向量化框架下, 可以通过循环展开同时发掘循环迭代间的向量并行性和迭代内的超字级并行性; (2) 加载节点数据重组 (load permutaion) 的合法性检查, 地址连续的加载操作可以天然的组成向量加载, 而不连续的加载操作, 通过额外的数据重组也有机会得以向量化. 展开因子的计算公式为:

$$UF = \frac{LCM(max\_nunits, group\_size)}{group\_size}$$

其中, *max_nunits* 表示 SLP 计算树中最小数据类型对应的向量化因子, *group_size* 为 SLP 计算树节点的同构语句数目. 对于基本块向量化, 这表明 SLP 方法希望并行执行的同构语句数目 *group_size* 是最大向量化因子 *max_nunits* 的整数倍, 这样 SLP 向量化总是合法的. 此外这也表明, 对于循环向量化, 向量化器以最小数据类型为基准讨论可能存在的循环展开, 以完全填充向量寄存器. 在基本块向量化中, 若 *UF* 不为 1, 即 *group_size* 不是 *max_nunits* 的整数倍, 并且存在着 *max_nunits* 大于 *group_size* 的节点时, SLP 向量化是无论如何不能被完成的, 因为不可能展开直线型代码来获得更多的数据级并行. 而当 *max_nunits* 小于 *group_size* 时, 向量化器从原始同构语句组中分裂出一个大小为 *group_size-group_size%max_nunits* 的同构语句组, 和一个数目不足 *max_nunits* 的同构语句组, 前者可以直接将其向量化, 后者则无法向量化. 这是非满载 SLP 向量化的一个重要检测点, 将在下一节中讨论. 展开因子分析完成后, 向量化器对需要进行数据重组的加载节点进行分析、验证, 以确认数据重组操作可以被完成.

vect_slp_analyze_instance_dependence 对基本块内的数据引用进行依赖分析. 通过判断 SLP 计算树执行路径上的数据依赖, 决定是否真的可以将分布在基本块内不同位置的标量语句进行重排序以完成打包. vect_slp_analyze_and_verify_instance_alignment 对访存节点中数据引用的对齐情况进行分析. 不支持的对齐类型无法进行向量化. 不对齐访存较对齐访存的代价更高, 在之后的代价模型中对齐信息将用于收益分析. vect_slp_analyze_operations 分析基本块中所有 SLP 计算树的操作类型. 对于每一个 SLP 计算树, 沿着定义使用关系从叶子节点到根节点, 递归的分析节点的向量操作是否被目标机所支持, 对于目标机所不支持的树节点进行移除.

vect_bb_vectorization_profitable_p 完成对基本块向量化的收益分析. 代价模型对基本块中所有 SLP 计算树的收益进行累加, 作为基本块向量化的收益. 当收益大于零时, 所有的 SLP 计算树都将被向量化; 否则任何 SLP 计算树都不会被向量化. SLP 计算树的收益被定义为树中每个节点的收益之和, 每个节点的收益等于对应的标量语句代价和向量语句代价的差值.

分析和决策阶段通过后, 进入向量代码生成阶段. 向量化器依次对基本块内的 SLP 计算树进行调度. 每调度一个 SLP 计算树, 从叶子节点到根节点递归地调用 vect_transform_stmt 为每个节点生成向量代码.

3 ISLP 实现

3.1 分析

分析阶段的目标是构建可以安全应用非满载 SLP 向量化的 SLP 计算树. 在当前的实现中, 非满载 SLP 向量化对应一棵不存在类型转换操作的非满载 SLP 计算树. 为了对非满载 SLP 向量化有更严格的描述, 首先介绍一些相关概念, 随后扩充 SLP 分析框架以支持非满载 SLP 向量化.

定义 1. 一个 SLP 计算树中的节点称为是非满载的, 如果有:

$$group\_size < VF$$

其中, *group_size* 是节点中同构语句的数目, *VF* 表示向量化因子, 即可并行执行的同构语句数.

定义 2. 一个 SLP 计算树称为是非满载的当且仅当树中的全部节点都是非满载的.

适用于非满载 SLP 向量化的 SLP 计算树的构建主要包括 3 个部分: (1) 构建非满载 SLP 计算树; (2) 检查类型转换操作; (3) 检查根节点的数据重组是否被目标机支持.

非满载 SLP 计算树的构建过程与与常规 SLP 计算树基本相同. 由定义 2 可知, 非满载 SLP 计算树要求所有的树节点都是非满载的, 因此只要在同构语句检测阶段认为 *VF* 大于 *group_size* 的同构语句组是合法的, 那么在一个非满载 SLP 种子的基础上, 根据定义使用链递归的进行扩展, 一个非满载 SLP 计算树就可以被建立起来.

在当前的实现中, 非满载的类型转换向量化是不被支持的. 类型提升需要对向量进行拆包扩展, 而类型

下降则需要压缩合并, 这些在非满载 SLP 向量化中是过于复杂的. 类型转换操作可能发生某一个节点, 也可能发生在多个节点. 可能是单一类型转换, 也可能多种类型转换同时存在. 但无论发生何种类型转操作, SLP 计算树中的数据类型一定不唯一. 由于在当前的 SLP 框架中有关于最小数据类型信息 *max_nunits* 的检测, 因此再添加对最大数据类型信息 *mini_nunits* 的检测即可, *mini_munits* 表示 SLP 计算树中最大数据类型对应的向量化因子. 在同构语句检测阶段, 检查并记录三地址码中操作数的最大数据类型信息 *mini_nunits*, 该功能由 vect_get_biggist_scalar_type 完成. SLP 计算树建立后, 如果 *UF* 不为 1 且 *max_nunits*大于 *group_size*, 这表明 SLP 计算树中存在着非满载节点, 如果同时有 *max_nunits* 等于 *mini_nunits*, 即所有节点的数据类型都对齐到 *max_nunits* 且是一致非满载的, 则说明该 SLP 计算树是非满载.

下面对非满载 SLP 计算树的存储节点进行验证. 非满载的存储操作会破坏程序的正确性, 在当前的实现中, 通过添加额外的向量重组指令对结果中的无效数据进行替换从而实现非满载存储的正确性. 为了保证向量重组指令的合法性, 需要对存储节点的数据类型进行检查, 以确保可以生成向量重组指令. 现在一个可以安全向量化的非满载 SLP 计算树已经建立起来, 接下来是正常的对齐分析、依赖分析等.

3.2 变换

ISLP 按照常规的调度顺序从叶子节点到根节点为非满载 SLP 计算树生成向量代码. ISLP 代码生成的核心是通过添加额外的数据重组操作来维持程序计算状态的正确性和避免可能的算术异常. 我们以清单 2 所示的标量 C 代码为例, 通过展示 4 通道下 3 路加法操作的向量化方法, 对 ISLP 的代码生成策略进行说明. 向量代码如清单 3 所示, 在向量写回阶段, 额外的数据重组操作被引入以完成对无效数据的替换.

清单 2. 标量 *C* 代码

```
float a[5], b[3], c[3];
a[i+0] = b[i+0] + c[i+0];
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
```

清单 3. 4 通道向量代码

```
vload v1 b
vload v2 c
vadd v3 v1 v2
vload v4 a
permuate v5, v3, v4, (0, 1, 2, 7)
vstore a v5
```

详细的 ISLP 代码生成如图 1 所示. 对于非满载的加载节点, 根据对齐信息从起始地址处生成常规向量加载. 为了避免可能存在的访存越界, 在发射指令之前考虑对待处理的数组进行尾部填充. 对于固定分配数组, 直接增加数组的大小; 对于动态数组, 调整参数申请更大的空间. 以 b 数组为例, 其内存分配方式为固定分配类型, 大小为 3. 4 通道向量加载操作将内存中以 b[0] 为起始地址的 4 个连续存放的数据元素加载至向量寄存器 v1, 向量加载可能触发地址越界, 造成程序错误. 为此, 通过增加数组大小对 b 数组进行尾部填充, 在有效数据之后填充若干个 dummy, 即未初始化的冗余数据元素. 这样 vload 操作不会访问到未知的地址空间, 越界异常得以避免.

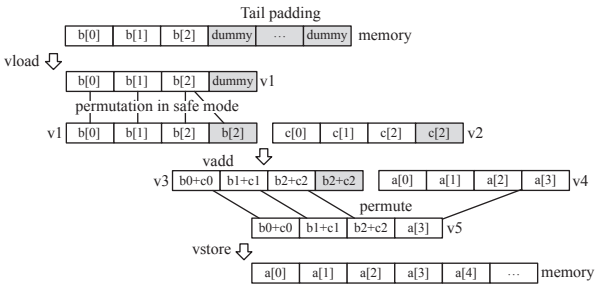

图 1 ISLP 代码生成

常规向量加载在向量寄存器中引入了无效的冗余数据, 无效数据和有效数据一同参与计算, 算术操作可能触发不可预测的异常. 为此通过向量重组指令使得向量寄存器中的有效数据覆盖无效数据. 有效数据的操作总是可以信任的, 因此算术异常得以避免. 异常的发生并不是必然的, 换言之也可以忽略异常发生的风险, 以换取更高的性能. 我们将算术异常的避免设置为一种可选的安全模式供用户选择是否启用. b 数组中的 dummy 和 b[0], b[1], b[2] 一起被加载至向量寄存器 v1, 用于参加后续算术操作. 由于 dummy 为未知数据, 该通道可能触发浮点异常. 大多数 SIMD 扩展都支持 permute 操作对向量寄存器中的数据进行重组. 在安全模式启用的情况下, 使用已知数据, 例如 b[2], 替换 dummy. 同理, 根据数据的对应关系, 使用 c[2] 替换 v2 中的 dummy. 这样可以避免 dummy 所在通道发生

不可预测的算术异常.

面向 ISLP 的向量存储维持程序计算状态的正确性. 当递归至根节点时, 可以根据对齐信息从起始地址处生成常规向量存储, 在此之前需要对向量寄存器中的无效数据进行替换, 否则它会覆盖有效数据并导致程序错误. 首先从写回地址处加载原始数据到向量寄存器, 接着按照有效数据的对应位置对两个向量寄存器中的数据进行重组, 最后使用常规向量存储将结果写回内存. 若直接使用 vstore 指令将向量寄存器 v3 中的数据写回以 a[0] 为起始地址的连续地址空间, 将导致 a[3] 被冗余数据覆盖, 造成程序计算状态的错误. 为此首先使用 a[3] 处的数据替换原 dummy 通道的冗余数据 b[2]+c[2], 然后写回内存. permute 操作可以帮助完成这一功能, 首先从 a[0] 处进行向量加载, a[3] 被加载至向量寄存器 v4, 接着和 v3 进行数据重组, 结果存放至 v5, 最后将 v5 中的内容写回内存. 这就保证了向量化前后程序计算状态的一致性.

此外, 在发射数据重组指令或者使用插入指令构造向量操作数时, 需要对掩码数组或者向量操作数进行尾部填充, 通常重复有效数据到填充位置即可.

3.3 代价模型

非满载 SLP 向量化作为常规 SLP 向量化的扩展重用现有的基本块向量化代价模型. 基本块向量化的收益等于基本块内所有 SLP 计算树的收益之和. SLP 计算树的收益等于每个节点的收益之和. 每个节点的收益由对应的标量语句代价和向量语句代价的差值刻画. 非满载 SLP 向量化在完成某些向量操作时存在额外的语句代价, 在进行代价计算时需要对该类型节点的向量代价进行修正.

根据当前的代码生成方式, 在计算加载节点向量代价时, 如果安全模式被启用, 需要添加相应的数据重组代价. ISLP 加载节点的向量代价:

$$stmt\_cost_{\text{vload}} + bool \times stmt\_cost_{\text{perm}}$$

其中, $stmt_cost_{\text{vload}}$为常规向量加载代价, $stmt_cost_{\text{perm}}$表示向量重组语句代价. 当安全模式启用时, $bool$为 1, 否则为 0.

在向量写回阶段要求额外的数据加载和重组操作, 加载语句的对齐方式与存储语句要保持一致. ISLP 存储节点的向量代价:

$$stmt\_cost_{\text{vstore}} + stmt\_cost_{\text{vload}} + stmt\_cost_{\text{perm}}$$

其中, $stmt_cost_{\text{vstore}}$为常规向量存储代价.

4 实验与分析

为了验证非满载 SLP 向量化的有效性, 从标准测试集中选取适合做非满载 SLP 向量化的程序作为测试用例, 对向量化的收益进行评估. 测试用例包括: SPEC CPU2006 测试集中的 435.gromacs, SPEC CPU2000 测试集中的 183.equake、191.fma3d, NPB 测试集中的 BT. 这些程序的核心函数中都存在着超字级并行性不足的情况, 可以使用非满载 SLP 向量化进行发掘. 实验在 intel xeon E5-2 620 上进行, 操作系统为 Linux, 编译器为 GCC 7.1.0, 开启 AVX 指令集扩展, 向量长度 256 bit, 可同时处理 8 个单精度浮点数据或 4 个双精度浮点数据.

进行 3 组测试: 标量版本, 使用-ftree-no-vectorize 选项关闭向量化; 非满载 SLP 向量化优化前, 使用-ftree-vectorize 选项开启常规向量化; 非满载 SLP 向量化优化后, -insufficient-slp 选项开启非满载 SLP 向量化. 记录程序的执行时间, 以标量版本为基准, 使用标量执行时间除以向量执行时间得到加速比.

实验结果如图 2 所示. 所选测试用例在非满载 SLP 向量化后获得了不同程度的性能提升. 435.gromacs 的核心循环存在间接数组访问, 183.quake 的核心为 do-while 循环, 191.fma3d 的核心为结构体访问, 它们都无法发掘向量并行性, 而迭代内同构语句数目为 3, 小于 AVX 平台的向量化因子, SLP 向量化无法发掘核心中的超字级并行性, 因此常规向量化方法的加速比较低, 分别为 1.03, 1.0, 1.02. 开启 ISLP 后, 核心循环均被向量化. 435.gromacs 核心循环中可并行部分占比较少, 加速效果不明显, 加速比为 1.08. 183.quake 和 191.fma3d 分别为 1.25 和 1.13. BT 的核心为非循环结构且存在着语句数目小于向量化因子的同构语句块, SLP 方法不能识别, 常规向量化加速比为 1.02, 开启 ISLP 后核心函数被向量化, 加速比 1.10. average 一栏展示了对应向量化版本在 4 个测试用例上加速比的算术平均值, 常规向量化方法为 1.02, 打开 ISLP 后为 1.14, ISLP 扩展后的基本块向量化器比常规版本在性能上提升 11.8%. 实验表明, 非满载 SLP 向量化可以有效地增强 GCC 的向量化能力, 提高程序的执行效率.

5 总结与展望

本文针对较长向量长度下的程序并行性相对不足这一问题, 介绍了一种面向基本块的非满载向量化方法 ISLP. 在对 GCC 的 SLP 向量化框架进行了深入分

析后, 从并行性检测、代价模型、代码生成 3 个方面阐述了 ISLP 在 GCC 中的设计与实现. AVX 平台上的实验结果表明, 扩展后的 SLP 框架可以有效地对超字级并行性不足的程序进行向量化处理, 提高程序执行效率. 值得注意的是, ISLP 在多线程环境下是不安全的, 而且代价模型的准确性仍有待提高, 这些都是进一步需要研究的问题[20].

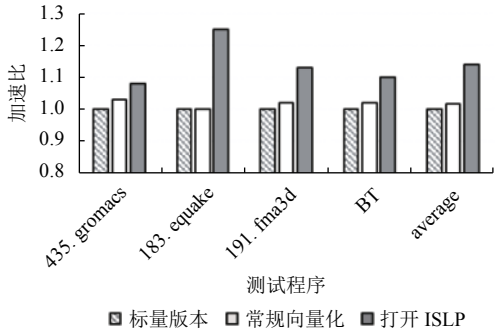

图 2 实验结果

参考文献

1 高伟, 赵荣彩, 韩林, 等. SIMD 自动向量化编译优化概述. 软件学报, 2015, 26(6): 1265–1284. [doi: 10.13328/j.cnki.jos.004811]

2 Bhargava R, John LK, Evans BL, *et al*. Evaluating MMX technology using DSP and multimedia applications. Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture (MICRO 31). Dallas: IEEE, 1998. 37–46.

3 Diefendorff K. Pentium III = Pentium II + SSE: Internet SSE architecture boosts multimedia performance. Microprocessor Report. https://docencia.ac.upc.edu/ETSETB/SEGPAR/microprocessors/pentium3%20(mpr).pdf. (1999-03-08).

4 Xiang YX, Zhang HM, Xiang XX, *et al*. Optimization of H.264 encoder based on SSE2. 2010 IEEE International Conference on Progress in Informatics and Computing. Shanghai: IEEE, 2010. 752–755. [doi: 10.1109/PIC.2010.5688015]

5 Takahashi D. An implementation of parallel 1-D FFT using SSE3 instructions on dual-core processors. 8th International Workshop on Applied Parallel Computing. State of the Art in Scientific Computing. Umeå: Springer, 2007. 1178–1187.

6 Francés J, Bleda S, Márquez A, *et al*. Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computing on FDTD scheme for solid and fluid vibration problems. The Journal of Supercomputing, 2014, 70(2): 514–526. [doi: 10.1007/s11227-013-1065-x]

7 Muła W, Lemire D. Faster Base64 encoding and decoding using AVX2 instructions. ACM Transactions on the Web, 2018, 12(3): 20. [doi: 10.1145/3132709]

8 Anderson CS, Zhang JW, Cornea M. Enhanced vector math support on the Intel® AVX-512 architecture. 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH). Amherst: IEEE, 2018. 120–124. [doi: 10.1109/ARITH.2018.8464794]

9 Xin NJ, Chen XC, Sun HY, *et al*. Extending the vector instruction set for high-performance DSP matrixes based on GCC. Computer Engineering & Science, 2012, 34(1): 58–63.

10 Stephens N, Biles S, Boettcher M, *et al*. The ARM scalable vector extension. IEEE Micro, 2017, 37(2): 26–39. [doi: 10.1109/MM.2017.35]

11 Petrogalli F. A sneak peek into SVE and VLA programming. ARM White Paper, 2016.

12 Takahashi D, Franchetti F. FFTE on SVE: SPIRAL-generated kernels. Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020). Fukuoka: Association for Computing Machinery, 2020. 114–122. [doi: 10.1145/3368474.3368488]

13 Chen YS, Mendis C, Carbin M, *et al*. VeGen: A vectorizer generator for SIMD and beyond. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Virtual: Association for Computing Machinery, 2021. 902–914. [doi: 10.1145/3445814.3446692]

14 Zhou H, Xue JL. A compiler approach for exploiting partial SIMD parallelism. ACM Transactions on Architecture and Code Optimization, 2016, 13(1): 11. [doi: 10.1145/2886101]

15 徐金龙, 赵荣彩, 赵博. SIMD 向量指令的非满载使用方法研究. 计算机科学, 2015, 42(7): 229–233. [doi: 10.11896/j.issn.1002-137X.2015.07.049]

16 高伟, 韩林, 赵荣彩, 等. 向量并行度指导的循环 SIMD 向量化方法. 软件学报, 2017, 28(4): 925–939. [doi: 10.13328/j.cnki.jos.005029]

17 王琦, 韩林, 姚金阳, 等. 不充分 SIMD 向量化技术研究. 计算机应用与软件, 2018, 35(9): 108–112. [doi: 10.3969/j.issn.1000-386x.2018.09.019]

18 Nuzman D, Zaks A. Autovectorization in GCC—two years later. Proceedings of the 2006 GCC Developers Summit. Ottawa, 2006. 145–158.

19 Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets. ACM SIGPLAN Notices, 35(5): 145–156. [doi: 10.1145/358438.349320]

20 Mendis C, Renda A, Amarasinghe S, *et al*. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. Proceedings of the 36th International Conference on Machine Learning. Long Beach: PMLR, 2019. 4505–4515.

(校对责编: 孙君艳)