中国科学: 信息科学 2017 年 第 47 卷 第 3 期: 310–325
SCIENTIA SINICA Informationis
论文

基于有向图可达性的 SLP 向量化识别方法

赵捷*, 赵荣彩

解放军信息工程大学数学工程与先进计算国家重点实验室, 郑州 450001
* 通信作者. E-mail: zjbc2005@163.com

收稿日期: 2016–06–09; 接受日期: 2016–07–19; 网络出版日期: 2017–01–13
"核高基" 国家科技重大专项 (批准号: 2009ZX01036-001-001-2) 和数学工程与先进计算国家重点实验室开放课题 (批准号: 2013A11) 资助项目

摘要 SLP (superword level parallelism) 是一种实现 SIMD (single instruction multiple data) 向量化的方法, 当前的主流向量化编译器都实现了这种向量化方法. 然而, 当前算法在进行 SLP 向量化时, 对应用程序中可向量化循环的分析过程过于保守, 导致其识别 SLP 向量化的能力不足. 为了提升该能力, 本文提出了一种基于有向图可达性的 SLP 向量化识别方法. 首先, 基于数组依赖图构建包含数组和语句依赖信息的有向图, 使同一条语句内的所有数组节点都在一个强连通分量内, 并对强连通分量之间的依赖边进行剪枝; 其次, 分析不同强连通分量节点之间的可达性, 根据节点的可达性获得识别 SLP 向量化所需的所有依赖信息, 从而确定语句中的循环是否可以进行 SLP 向量化. 将该方法在 Open64-5.0 编译器中实现后, SLP 向量化效果得到大幅提升. 对 gcc-vect 测试集中程序的实测结果表明, 优化后的 Open64-5.0 编译器识别 SLP 向量化循环的能力优于 GCC4.9, 与 Intel ICC14.0 相当, 生成的向量化代码性能优于当前最优算法.

关键词 向量化编译器, 超字并行, 依赖, 有向图, 可达性

1 引言

随着多媒体应用的普及, 主流处理器厂商都为其微处理器增加了多媒体扩展, 如 Intel 的 MMX、AMD 的 3DNow! 以及 IBM 的 VMX/Altivec 等. 然而, 实现这些多媒体扩展的体系结构不尽相同, 因此各处理器又集成了 SIMD (single instruction multiple data) 指令集来提升不同应用程序的性能. SIMD 技术不仅能够用于提高通用处理器的多媒体处理性能, 而且在游戏机芯片 [1]、数字信号处理器 (DSP) 以及大规模并行处理机 [2] 中也得到了广泛应用. 然而, 由于用户对这些多媒体扩展部件并不熟悉, 无法有效利用这些加速部件来提升程序性能, 因此需要使用自动 SIMD 向量化来帮助用户解决困难.

实现 SIMD 向量化的方法主要有两种. 一种是针对传统向量机实现的传统向量化方法 [3], 这种方法主要针对内层循环的迭代空间, 以整个数组作为一个向量单元进行操作, 没有向量长度的限制, 其

引用格式: 赵捷, 赵荣彩. 基于有向图可达性的 SLP 向量化识别方法. 中国科学: 信息科学, 2017, 47: 310–325, doi: 10.1360/N112016-00146
Zhao J, Zhao R C. Identifying superword level parallelism with directed graph reachability (in Chinese). Sci Sin Inform, 2017, 47: 310–325, doi: 10.1360/N112016-00146

主要思想是将程序划分成不同的 π 块, 即不同的强连通区域, 并根据依赖关系对强连通区域做拓扑排序, 由此来识别循环中的可向量化区域并实施向量化.

另一种方法是超字并行 (superword level parallelism, SLP) 向量化 [4]. 这种方法通过将数据进行组合打包组成超字, 并利用 SIMD 指令对这些超字进行操作, 其主要思想是先识别程序中的同构语句, 这些同构语句中对应位置上的操作相同, 并且对应位置上的操作数具有相同的数据类型, 然后将这些同构语句进行打包组成超字指令一起执行. 与传统向量化方法相比, 超字并行的一个最重要的优点就是能够在应用程序中的并行性较小或适中的情况下仍能够有效利用多媒体扩展实施并行, 而传统向量化方法只有在程序中存在大量并行性时才能进行向量化.

几乎所有的主流向量化编译器如 Intel ICC, GCC, IBM XL 以及 Open64 等, 都实现了 SLP 向量化算法. 然而, 在实际应用中对程序实施 SLP 向量化时, 其效果却不是十分明显. 经过分析发现, 这是由于向量化编译器中对可向量化循环的分析过程过于保守, 导致其识别 SLP 向量化的能力不足. 因此, 为了提升该能力, 本文提出一种基于有向图可达性的 SLP 向量化识别方法. SLP 向量化循环的识别过程, 主要是对待测循环内依赖关系的分析过程, 传统的方法首先根据依赖测试结果构建数组依赖图, 在 SLP 向量化识别阶段, 再根据数组依赖图构建语句依赖图, 通过分析语句之间的依赖关系判定循环是否可以向量化, 这种方法需要交替遍历数组依赖图和语句依赖图, 并且需要在遍历的过程中找到产生依赖的数组和语句节点, 不仅容易出错, 而且要求编译器开发人员需要对依赖关系以及数组和语句依赖图的使用十分了解.

为了克服这个缺点, 首先将数组依赖图扩展成同时包含数组依赖关系和语句依赖关系的有向依赖关系图, 在识别 SLP 向量化循环时只需遍历该有向依赖图, 并且只需对语句中的某个读引用数组节点进行分析就能够得到识别 SLP 向量化时所必需的依赖信息, 不仅简化了代码实现过程, 而且不容易出错. 具体来讲, 本文的主要贡献如下所述.

(1) 通过扩展语句依赖图构建了一种新的有向依赖关系图, 该依赖关系图同时包含数组和语句之间的依赖关系信息, 便于简化 SLP 向量化识别的过程;

(2) 提出了一种基于该有向依赖图内节点之间可达性的 SLP 向量化识别方法, 根据节点之间的可达性关系能够获得识别 SLP 向量化所需的所有依赖信息;

(3) 在 Open64-5.0 编译器中实现了采用上述识别方法的 SLP 向量化算法, 大幅提升了 Open64-5.0 编译器识别 SLP 向量化的能力;

(4) 利用优化后的 Open64-5.0 编译器对实际应用程序进行了测试, 结果表明优化后的 Open64-5.0 编译器识别 SLP 向量化的能力优于 GCC4.9 编译器, 与 Intel14.0 编译器相当; 对部分应用程序, 利用优化后的 Open64-5.0 编译器实现 SLP 向量化, 其性能优于商业编译器和当前最优算法.

本文结构安排如下: 第 2 节对 Open64-5.0 编译器中 SLP 向量化识别的过程进行概述, 并阐述研究动机; 第 3 节介绍如何根据数组依赖图构建同时包含数组和语句依赖信息的有向依赖关系图; 第 4 节说明如何利用该有向依赖关系图识别 SLP 向量化; 第 5 节给出实验结果, 并对实验结果进行讨论; 第 6 节介绍相关工作; 第 7 节总结全文.

2 研究背景与动机

2.1 Open64 中的 SLP 向量化识别过程

为了验证编译器识别 SLP 向量化的能力, 从 gcc-vect 测试集中随机抽取了 100 个程序进行了测

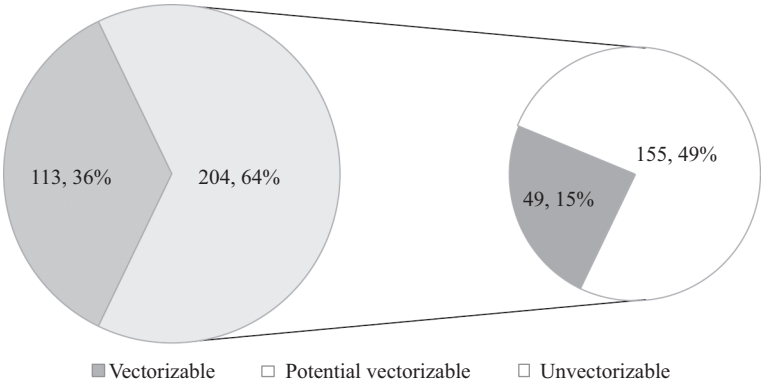

图 1　移植 SLP 方法之后的 Open64-5.0 向量识别比例

Figure 1　Observation on the SLP vectorization of Open64-5.0

试, 这些程序中共包含 317 个循环, 测试结果如图 1 所示. 从图 1 中可以看出, 只有 113 个循环能够被 Open64-5.0 编译器正确识别并生成向量化语句.

对于其他 204 个未能够被向量化的循环进行分析之后发现, 24% 的循环本可以被向量化但没有被当前 SLP 算法正确识别, 76% 是无法被向量化的串行循环. 由此可以看出, 当前 SLP 算法对可向量化循环的分析过程过于保守, 导致其识别 SLP 向量化的能力不足.

对于一个循环, 首先判定其基本块是否能够被识别为 SLP 向量化, 而该判定则是通过分析基本块内的语句所携带的依赖决定的. 如果某个写引用数组携带循环携带真依赖, 或在两个相邻迭代之间没有访存连续地址, 那么产生依赖的两个数组都会被加入到 novect_ref_set 集合中, 该集合用于保存阻碍向量化的数组. 当 novect_ref_set 集合中数组元素的个数与基本块内所有数组个数之比达到一个阈值时, 该基本块就无法进行 SLP 向量化.

当前 SLP 向量化识别存在一定的问题. 首先, 把数组加入 novect_ref_set 集合时, 只分析写引用数组的出边, 也就是说该过程不考虑反依赖. 事实上, 反依赖的确不影响 SLP 向量化的识别, 但是, 一个先进的优化编译器应该能够在识别的过程中进行适当的优化, 为后面实施语句打包提供方便. 反依赖不影响识别, 但会影响到优化的实施, 我们将在后面说明这个问题. 其次, 只要存在循环携带依赖, 就会被加入到 novect_ref_set 集合中, 而向量寄存器都有固定的长度, 现有的过程既没有考虑依赖距离, 也没有考虑依赖的层. 再次, 在识别 SLP 向量化的过程中, 只利用由数组依赖图传递过来的信息, 其实现过程并没有构建语句依赖图, 然而, 在识别 SLP 向量化时编译器不仅需要考虑数组是否携带依赖, 而且还需要考虑构成依赖的语句之间是否存在依赖环、依赖是前向/后向依赖等, 这些信息很难从数组依赖图获得.

2.2　研究动机

数组依赖图是根据依赖测试结果构建出来的数组之间的依赖关系图. 依赖测试返回数组之间是否存在依赖, 在判定有依赖时还能够确定该依赖是循环携带依赖或是循环无关依赖. 如图 2(b) 是图 2(a) 所示代码的数组依赖图, 从中可以获取依赖距离和依赖层次等信息, 该依赖是真依赖、反依赖或是输出依赖也可以通过简单判定依赖源点和汇点的存储类型得出. 然而, 在识别 SLP 向量化时, 除上述信息外, 还需要考虑产生依赖的两个语句之间是否存在依赖环、依赖是否为自身依赖、语句之间的真依赖是前向/后向依赖, 这只能通过分析语句依赖图来判定.

语句依赖图是根据数组依赖图构建出来的语句之间的依赖关系图. 如图 2(c) 是图 2(a) 所示代码的语句依赖图. 识别 SLP 向量化的一种有效方法是在该阶段构建语句依赖图, 在原有信息的基础上, 从语句依赖图中获取上述信息从而识别向量化. 然而, 这种方法需要交替遍历数组依赖图和语句依赖图, 并且需要在遍历的过程中找到产生依赖的数组和语句节点, 不仅容易出错, 而且要求编译器开发人员需要对依赖关系以及数组和语句依赖图的使用十分了解. 因此, 本文通过构建一种便于实现且不易出错的有向图来收集识别向量化所必需的依赖信息, 并基于此识别向量化.

3 一种包含数组和语句依赖信息的依赖图

3.1 构建扩展依赖关系图

在图 2(b) 中, w 表示写引用节点, r 表示读引用节点. 显然, 语句 $S1$ 和 $S2$ 之间存在两个依赖, 一个是前向的真依赖 e_1, 另一个是后向的反依赖 e_2. 按照当前算法识别 SLP 向量化的过程, 首先分析 w_1 节点, 由于存在一条以 w_1 节点为源点的依赖边 e_1, 且该依赖为循环携带真依赖, 因此, SLP 算法会将依赖边 e_1 的源点和汇点都加入到 novect_ref_set 集合中; 接下来继续分析 w_2 节点, 由于不存在以 w_2 节点为源点的依赖边, 因此 novect_ref_set 集合没有更新. 至此已分析完所有的写引用节点, 分析过程结束. 由于 novect_ref_set 集合中元素个数所占比例超过了阈值, 该循环被识别为无法 SLP 向量化.

显然该过程没有考虑依赖距离, 由于 e_1 边引起的依赖距离为 4, 当向量化因子小于等于 4 时, 不应该将 w_1 和 r_2 加入 novect_ref_set 集合中. 此外, 依赖边 e_2 没有被分析, 即使此时 novect_ref_set 集合中元素个数与所有节点个数的比例没有超过设定的阈值, 而将该循环识别为可进行 SLP 向量化, 其生成的向量化代码也是错误的, 因为这两个语句之间存在依赖环, 需要进行优化. 因此, 虽然反依赖不影响 SLP 向量化的识别, 但会对向量化代码产生影响. 此时, 需要分析语句依赖图如图 2(c) 所示. 根据现有的主流向量化编译器思想, 分析过程应该如下: 首先分析 w_1 节点, 过程与刚才所述相同, 由于依赖距离为 4, novect_ref_set 集合没有被更新; e_1 边连接的源点和汇点分别为 w_1 和 r_2 节点, 而 w_1 节点的父节点为语句 $S1$, r_2 节点的父节点为语句 $S2$, 在语句依赖图中存在从 $S2$ 到 $S1$ 的依赖, 因此两条语句之间存在依赖环; 接下来依次遍历 $S2$ 中的数组节点, 寻找 e_2 边的源点 r_3, 从而确定 $S2$ 到 $S1$ 之间为反依赖; 然后再分析 w_2, 该过程中 novect_ref_set 集合一直都没有被更新. 至此已分析完所有的写引用节点, 分析过程结束.

虽然此时 novect_ref_set 集合中元素个数为 0, 一定小于设定的阈值, 但是由于存在依赖环, 此循环需要进行优化. 由于从 $S2$ 到 $S1$ 的边为反依赖, 依赖环中的反依赖可通过节点分裂 [5] 消除, 因此对代码进行优化后得到程序如图 2(d) 所示, 此时依赖环已被消除, 可生成正确的向量化代码.

为了简化交替遍历数组依赖图和语句依赖图中节点的复杂过程, 我们构建了该程序的扩展依赖关系图如 2(e) 所示, 构建过程如下: 首先, 创建一个空图, 并将数组依赖图复制到该图中; 然后, 向该依赖图中为属于同一个语句内的不同节点添加有向边, 使同一条语句内的各节点相互可达. 该过程将原来的数组节点划分到不同的强连通分量, 而每个强连通分量与程序中的一条语句相对应; 原数组依赖图中的依赖边则将各强连通分量连接起来. 需要注意的是依赖边可能会使该扩展依赖关系图中的强连通分量数量变少, 而我们在后面分析过程中所指的强连通分量是指去掉依赖边之前与语句一一对应的强连通分量.

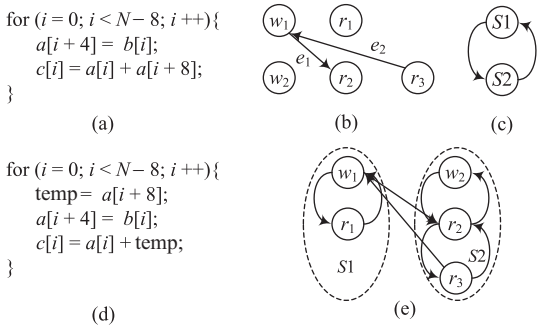

图 2　示例代码及其相应的依赖关系图

Figure 2　An example with its dependence graphs

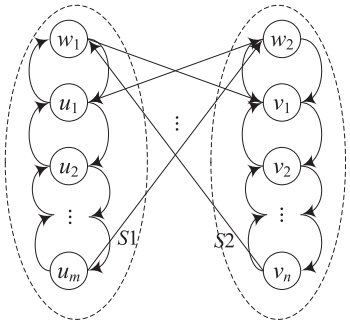

图 3　任意两个语句之间的扩展依赖关系图

Figure 3　An extended dependence graph of arbitrary statements $S1$ and $S2$

3.2　可达性与向量化识别

基于该扩展依赖关系图的 SLP 向量化分析过程如下. 对于读引用节点 r_1 和 r_2, 从 r_1 到 r_2 存在唯一可达路径 (r_1, w_1, r_2), 相应地从 r_2 到 r_1 的唯一路径为 (r_2, r_3, w_1, r_1). 依赖的层、依赖距离可从 e_1 和 e_2 中直接获得; 相互可达表明两个语句之间存在依赖环; 两个节点之间的可达路径必通过且仅通过写引用节点 w_1, 因此从 $S1$ 到 $S2$ 为真依赖, 从 $S2$ 到 $S1$ 为反依赖; 而前向和后向关系则可以根据语句执行顺序与依赖方向之间的关系进行判定. 因此从该扩展依赖关系图中可获得所有需要的依赖信息. 此外, 与传统的先构建语句依赖图再进行判定的方法相比, 该方法只需在每个语句中遍历一个读引用节点, 大大减少了遍历节点的次数, 从而简化识别过程.

4　基于扩展依赖关系图的 SLP 向量化识别

在程序中产生依赖的源点和汇点可能属于相同的语句, 也可能属于不同的语句, 前者属于自身依赖, 后者属于语句间的依赖.

4.1　语句间依赖的 SLP 向量化识别

为了便于说明, 我们以图 3 为例, 对任意两个语句之间的读引用节点进行编号 ($u_1, \ldots, u_m, v_1, \ldots, v_n$). 分析过程中可任意选取读引用节点, 这是因为对于任意 u_i ($1 \leqslant i \leqslant m$) 和 v_j ($1 \leqslant j \leqslant n$), u_i 必可达 w_1, v_j 必可达 w_2, 根据依赖关系的定义, 两个语句之间存在依赖时至少有一条语句是向存储单元中写入数据, 因此, 强连通分量 $S1$ 和 $S2$ 之间的依赖边必定经过 w_1 或 w_2 两者之一, 那么 u_i 可达 v_j 当且仅当 u_k ($k \neq i$) 可达 v_j, 同理可知 v_j 可达 u_i 当且仅当 v_l ($l \neq j$) 可达 u_i. 由此可知, 在分析 SLP 向量化时只要找到两条语句的第一个读引用节点, 即 u_1 和 v_1 就可以确定依赖关系, 而不用遍历全部节点.

在分析该有向依赖关系图之前, 需要对强连通分量之间的依赖边进行剪枝. 对于 SIMD 向量化而言, 如果依赖距离大于或等于向量化因子, 那么该依赖是无效的, 这个事实已经在文献 [6] 中得到了证明. 而依赖距离可以由依赖测试确定, 在构建该扩展依赖图的过程当中不同强连通分量之间的依赖边就已经携带依赖距离的信息, 因此, 我们只需判定其是否大于向量化因子, 并在满足条件时从扩展依赖图中剪去该依赖边即可. 接下来, 就是根据不同强连通分量内节点之间的可达性来判定依赖关系的其他信息. 因此, 有如下的定理.

定理1 对于任意节点 u_i ($1\leqslant i\leqslant m$) 和 v_j ($1\leqslant j\leqslant n$), 如果 u_i 可达 v_j 且 v_j 不可达 u_i, 那么 $S1$ 和 $S2$ 可以向量化, 但 $S1$ 的执行必须在 $S2$ 之前.

证明 首先, v_j 不可达 u_i, 那么不存在从 $S2$ 到 $S1$ 的依赖, 而 u_i 可达 v_j, 即 $S2$ 依赖于 $S1$, 此时需要讨论两种情况.

(1) 原程序中 $S1$ 先执行而 $S2$ 后执行, 那么所有依赖要么为循环无关依赖, 要么为前向的循环携带依赖, 因此 $S1$ 和 $S2$ 可以向量化;

(2) 原程序中 $S2$ 先执行而 $S1$ 后执行, 那么 $S1$ 与 $S2$ 之间的依赖必定是循环携带依赖, 否则存在从 $S2$ 到 $S1$ 的依赖, 与 v_j 不可达 u_i 矛盾, 根据依赖的基本定理 [4], 交换 $S1$ 与 $S2$ 之间的顺序仍将保持原程序的语义, 而交换后所有的依赖将变成前向依赖, 因此 $S1$ 和 $S2$ 可以向量化, 而此时 $S1$ 在 $S2$ 之前执行.

综上所述, 当 u_i 可达 v_j 且 v_j 不可达 u_i 时, $S1$ 和 $S2$ 可以向量化, 但 $S1$ 的执行在 $S2$ 之前.

定理2 对于任意节点 u_i ($1\leqslant i\leqslant m$) 和 v_j ($1\leqslant j\leqslant n$), 如果 u_i 和 v_j 相互可达, 且 u_i 和 v_j 之间不存在经过 w_2 的路径, 那么 $S1$ 和 $S2$ 可以向量化, 但 $S2$ 须进行节点分裂.

证明 u_i 和 v_j 相互可达, 说明语句 $S1$ 和 $S2$ 之间存在依赖环, 而所有路径中不存在经过 w_2 的路径, 说明既不存在从 $S1$ 到 $S2$ 的反依赖, 也不存在从 $S2$ 到 $S1$ 的真依赖, 那么 $S1$ 和 $S2$ 之间依赖环的类型只能是: 从 $S1$ 到 $S2$ 的依赖为真依赖, 而从 $S2$ 到 $S1$ 之间的依赖为反依赖. 考虑真依赖, 此时需要讨论 4 种情况.

(1) 当从 $S1$ 到 $S2$ 的真依赖为循环无关依赖时, $S1$ 必定在 $S2$ 之前执行, 那么 $S2$ 到 $S1$ 的反依赖必定为循环携带依赖, 并且该反依赖为后向反依赖, 将 $S2$ 进行节点分裂, 并将分裂之后的反依赖节点与 $S1$ 进行语句交换, 则所有的依赖均变成前向依赖, 因此 $S1$ 和 $S2$ 可以向量化;

(2) 当从 $S1$ 到 $S2$ 的真依赖为循环携带依赖时, 如果 $S1$ 在 $S2$ 之前执行, 那么真依赖是前向的, 而且 $S2$ 到 $S1$ 的反依赖是后向的循环携带依赖, 那么, 先对 $S2$ 进行节点分裂, 并将分裂之后的反依赖节点与 $S1$ 进行语句交换就可以向量化 $S1$ 和 $S2$;

(3) 如果 $S1$ 在 $S2$ 之后执行, 当 $S2$ 到 $S1$ 的反依赖是循环携带依赖时, 交换 $S1$ 和 $S2$ 符合依赖的基本定理, 可以通过语句交换将后向真依赖转换成前向依赖;

(4) 当 $S2$ 到 $S1$ 的反依赖是循环无关依赖时, 将 $S2$ 进行节点分裂, 并将分裂之后的真依赖节点与 $S1$ 进行语句交换, 则所有的依赖均变成前向依赖, 因此 $S1$ 和 $S2$ 可以向量化.

综上所述, 当 u_i 和 v_j 相互可达, 且不存在经过 w_2 的路径时, $S1$ 和 $S2$ 可以向量化, 但 $S2$ 须进行节点分裂.

需要注意的是, 在判定 u_i 和 v_j 之间的路径是否经过 w_2 时, 无需遍历所有的依赖边, 因为从数组依赖图中可直接分析得出 w_2 是否有出边和入边, 而该信息同样被传递到扩展依赖关系图中, 只需判定 w_2 的出边和入边个数是否都为 0, 即可说明不存在经过 w_2 的路径.

定理3 对于任意节点 u_i ($1\leqslant i\leqslant m$) 和 v_j ($1\leqslant j\leqslant n$), 如果 u_i 和 v_j 相互可达, 且所有从 u_i 出发经 v_j 再返回 u_i 的回路都是先经过 w_1 再经过 w_2, 那么 $S1$ 和 $S2$ 不可以向量化.

证明 与定理 2 相同, 语句 $S1$ 和 $S2$ 之间存在依赖环, 由于从 u_i 出发经 v_j 再返回 u_i 的回路都是先经过 w_1 再经过 w_2, 即 u_i 只能经过 w_1 到达 $S2$ 语句, 而 v_j 只能经过 w_2 到达 $S1$ 语句, 说明 $S1$ 和 $S2$ 之间均为真依赖, 仍需考虑两种情况.

(1) 如果有循环无关依赖, 那么无法交换语句顺序, 由于同时存在后向的循环携带真依赖, 因此此时 $S1$ 和 $S2$ 无法向量化.

```
for (i = 0; i < N−4; i ++){
    a[i + 4] = a[i];
}
```
(a)

```
for (i = 0; i < N−4; i ++){
    a[i] = a[i + 4];
}
```
(b)

```
for (i = 0; i < N−4; i ++){
    temp= a[i];
    a[i + 4] = temp;
}
```
(c)

```
for (i = 0; i < N−4; i ++){
    temp = a[i + 4];
    a[i] =temp;
}
```
(d)

图 4　含有自身依赖的代码示例

Figure 4　Self-dependence cases

(2) 如果只有循环携带依赖, 那么两条语句之间的所有依赖都是循环携带依赖, 并且可以交换顺序, 然而即使交换语句顺序也无法消除后向的真依赖, 因此 $S1$ 和 $S2$ 无法向量化.

综上所述, 当 u_i 和 v_j 相互可达, 且所有从 u_i 出发经 v_j 再返回 u_i 的回路都是先经过 w_1 再经过 w_2 时, $S1$ 和 $S2$ 不可以向量化.

定理4　对于任意节点 u_i ($1\leqslant i\leqslant m$) 和 v_j ($1\leqslant j\leqslant n$), 如果 u_i 和 v_j 相互可达, 且所有从 u_i 出发经 v_j 再返回 u_i 的回路都是先经过 w_2 再经过 w_1, 那么 $S1$ 和 $S2$ 可以向量化, 但 $S2$ 须进行节点分裂.

证明　基于同样的原因, 语句 $S1$ 和 $S2$ 之间存在依赖环, 由于从 u_i 出发经 v_j 再返回 u_i 的回路都是先经过 w_2 再经过 w_1, 即 u_i 只能经过 w_2 到达 $S2$ 语句, 而 v_j 只能经过 w_1 到达 $S1$ 语句, 说明 $S1$ 和 $S2$ 之间均为反依赖, 仍需考虑两种情况.

(1) 当存在循环无关依赖时, 不妨假设 $S1$ 到 $S2$ 的依赖是循环无关的, 那么将 $S2$ 进行节点分裂, 并将分裂之后的反依赖节点与 $S1$ 进行语句交换, 则所有的依赖均为前向依赖, 此时 $S1$ 和 $S2$ 可以向量化.

(2) 当只存在循环携带依赖时, 两条语句都是循环携带依赖, 由于此时语句交换是合法的, 不妨假设 $S1$ 到 $S2$ 的依赖是前向的, 将 $S2$ 进行节点分裂, 并将分裂之后的反依赖节点与 $S1$ 进行语句交换, 则所有的依赖均为前向依赖, 因此 $S1$ 和 $S2$ 可以向量化.

综上所述, 当 u_i 和 v_j 相互可达, 且所有从 u_i 出发经 v_j 再返回 u_i 的回路都是先经过 w_2 再经过 w_1 时, $S1$ 和 $S2$ 可以向量化, 但 $S2$ 须进行节点分裂.

定理5　对于任意节点 u_i ($1\leqslant i\leqslant m$) 和 v_j ($1\leqslant j\leqslant n$), 如果 u_i 和 v_j 相互不可达, 那么 $S1$ 和 $S2$ 可以向量化.

证明　u_i 和 v_j 相互不可达, 也就是说 $S1$ 和 $S2$ 之间不存在依赖, 那么以哪种顺序执行 $S1$ 和 $S2$ 都将维持原来的依赖, 因此 $S1$ 和 $S2$ 可以向量化.

4.2　自身依赖的 SLP 向量化识别

识别语句间的 SLP 向量化时, 我们都基于一种假设, 即产生依赖的源点和汇点属于两个不同的语句. 然而, 在实际应用程序中, 可能存在依赖源点和汇点都属于同一条语句的情况, 因此, 本节对这种情况进行处理.

考虑如图 4(a) 和 (b) 所示代码是自身依赖的两种情况, 如果对这两种情况实施节点分裂, 那么这两种情况分别转换成如图 4(c) 和 (d) 的情况. 由于转换后的程序满足依赖的基本定理, 因此, 转换前后的代码是等价的. 这样就将自身依赖转换为语句间依赖的情况. 因此, 有如下的定理.

定理6 对于任意节点 u_i ($1 \leqslant i \leqslant m$) 和 w_1, 如果存在从 w_1 到 u_i 的依赖边, 那么语句 $S1$ 不可以向量化.

证明 如果存在从 w_1 到 u_i 的依赖边, 那么语句 $S1$ 存在到自身的真依赖, 如图 4(a) 所示, 经过节点分裂之后, 代码如图 4(c) 所示. 此时两条语句存在依赖环, 且两个依赖均为真依赖, 根据定理 3, 经过节点分裂之后的两条语句不可向量化, 即 $S1$ 不可向量化.

定理7 对于任意节点 u_i ($1 \leqslant i \leqslant m$) 和 w_1, 如果不存在从 w_1 到 u_i 的依赖边, 那么语句 $S1$ 可以向量化.

证明 此时需要分两种情况讨论.

(1) 如果不存在从 u_i 到 w_1 的依赖边, 那么语句 $S1$ 不存在自身依赖, 那么 $S1$ 可以向量化.

(2) 如果存在从 u_i 到 w_1 的依赖边, 那么语句 $S1$ 存在到自身的反依赖, 如图 4(b) 所示, 经过节点分裂之后, 代码如图 4(d) 所示. 此时两条语句之间不存在依赖环, 并且所有的依赖均为前向依赖, 因此, 经过节点分裂之后的两条语句可以向量化, 即 $S1$ 可以向量化.

5 实验评估

为了验证本文提出方法的有效性, 我们将算法在 Open64-5.0 编译器上进行了实现. 首先, 对比分析优化前后 Open64-5.0 编译器识别 SLP 向量化的能力, 然后再比较优化后的 Open64-5.0 编译器与 GCC 和 Intel ICC 之间识别 SLP 向量化循环的能力, 最后通过对几个基准测试集的向量化来说明我们的识别方法在哪些地方优于现有的向量化编译器和最优算法.

5.1 优化前后 Open64 编译器的识别能力对比

我们编写了 15 个小程序来对比优化前后 Open64 的识别能力. 这 15 个测试程序覆盖了前向/后向依赖、真/反依赖[1)]、循环携带/无关依赖、自身/语句间依赖以及是否存在依赖环、依赖距离是否大于向量化因子等各种情况, 能够反应一个编译器在含有复杂依赖关系情况下识别向量化的能力. 测试结果如表 1 所示, 其中, 每个测试程序的核心代码也列入表中, original-Open64 表示未使用本文提出方法的 Open64 编译器, 而 optimized-Open64 表示实现了本文方法的 Open64 编译器. 编译选项为 -O3 -LNO:slp=1, 实验中的数据类型为 64 位整数, 并假设向量寄存器长度为 256 位, 即向量因子为 4.

对表 1 中列出的 15 个程序, 未优化的 Open64 都无法将其识别为并行的循环, 而本文方法只有对第 1 个程序和第 12 个程序无法识别为向量化, 这与定理 3 和 6 所描述的情况一致. 对于其他 13 个程序, 优化后的 Open64 都可以正确识别并生成正确的向量化代码.

其中, 第 15 个程序的向量识别容易引起读者混淆. 从第 4 节介绍的几种情况来看, 没有与之相关的定理. 事实上, 该程序是一个多维数组的例子, 含有依赖环, 并且两者都是真依赖. 由于存在循环携带依赖, 未优化前的 Open64 不能将其识别为向量化; 虽然该程序两条依赖边都是真依赖, 但是数组 a 的依赖是由 i 循环携带, 而数组 b 的依赖是由 j 循环迭代, 所在依赖层不同, 因此, 该程序不存在依赖环, 优化后的 Open64 编译器考虑了依赖的层对向量化的影响, 在代码实现时对这种情况进行了分析, 因此可以正确识别该程序. 表 1 测试结果说明, 使用本文提出方法的 Open64 编译器识别 SLP 向量化的能力得到大幅度提升.

1) 这里未考虑输出依赖, 因为输出依赖对向量化不会产生影响.

表 1　手工编写程序的 SLP 向量化识别对比

Table 1　SLP vectorzation detection of the hand-coded programs

Item	Kernel codes	Dependence types	Original-Open64	GCC4.9	ICC14.0	Optimized-Open64
1	$a[i+4]=b[i-4];$ $b[i]=a[i];$	Cyclic, true dependences	×	×	×	×
2	$a[i+4]=c[i];$ $b[i]=a[i];$	Acyclic, forward true dependences	×	√	√	√
3*	$a[i]=b[i];$ $b[i+1]=c[i];$	Acyclic, backward true dependences	×	×	×	√
4	$a[i-4]=b[i+4];$ $b[i]=a[i];$	Cyclic,anti-dependences	×	√	√	√
5	$a[i]=c[i+4];$ $c[i]=b[i];$	Acyclic, forward anti-dependences	×	√	√	√
6*	$a[i]=c[i];$ $b[i]=a[i+1];$	Acyclic, backward anti-dependences	×	×	×	√
7	$a[i+4]=b[i];$ $b[i-4]=a[i];$	Forward true & forward anti	×	√	√	√
8	$a[i+4]=b[i];$ $c[i]=a[i]+a[i+8];$	Forward true & backward anti	×	√	√	√
9*	$c[i]=a[i]+a[i+2];$ $a[i+1]=b[i];$	Backward true & forward anti	×	×	×	√
10*	$a[i-1]=b[i];$ $b[i+1]=a[i];$	Backward true & backward anti	×	×	×	√
11	$a[i+4]=a[i];$	True self-dependence	×	√	√	√
12	$a[i+1]=a[i];$	True self-dependence	×	×	×	×
13	$a[i]=a[i+4];$	Anti-self-dependence	×	√	√	√
14*	$a[i]=a[i+1];$	Anti-self-dependence	×	×	×	√
15*	$a[i][j]=b[i][j-1];$ $b[i][j]=a[i-1][j-1];$	Multi-dimensional	×	×	×	√

5.2　与 GCC 和 ICC 的识别能力对比

为了进一步验证本文提出方法能够有效识别 SLP 向量化, 本节将优化后的 Open64 编译器与当前主流向量化编译器 GCC 和 Intel ICC 的向量化识别能力进行比较. 为了充分说明本文提出方法与当前主流向量化编译器识别 SLP 能力的不同, 本节实验从两个方面进行验证.

首先, 我们仍使用这 3 个编译器对 5.1 小节中手工编写的小程序进行测试. 采用的编译器版本分别为 GCC4.9[2)]和 ICC14.0[3)], 都是这两个编译器目前最新的版本, 测试结果如表 1 所示. GCC4.9 编译选项为 -O3, ICC14.0 编译选项为 -O3 -vec-report3. 其中带有 “*” 标记的程序是这两个编译器都没有识别为 SLP 向量化, 但本文方法识别为 SLP 向量化的程序, 对于其他程序, 我们的编译器与这两个编

2) GCC 4.9 release series changes, new features, and fixes http://gcc.gnu.or g/gcc-4.9/changes.html.

3) Intel ®C & C++ compilers. http://software.intel.com/en-us/c-compilers.

译器识别情况相同, 因此, 重点讨论这些带有 "*" 标记的程序[4].

事实上这些带有 "*" 标的程序都有与之相对应的语句顺序相反, 但依赖距离不同的例子. 例如, 程序 3 和 2, 事实上这两个程序除了语句顺序不同之外, 只有依赖距离不同. 我们知道程序 3 可以通过交换语句顺序变成程序 2 的形式, 但只有我们的编译器识别了程序 3, 而 GCC 和 ICC 都将其识别为不可向量化. 当我们将程序 2 中的数据依赖距离改为小于 4 时 GCC 和 ICC 也都识别为不可向量化, 在大于等于 4 时情况相反, 这说明这两个编译器假设的向量寄存器长度均为 256 位, 依赖距离不小于向量因子时该依赖边被消除, 如 4.1 小节中开头所述, 而本文方法是将向量寄存器长度设置为变量, 可以是 256 位也可以根据目标体系结构自动扩展为 512 位的计算方法, 这样, 即使该依赖距离小于向量化因子, 我们同样将其识别为向量化, 在生成向量化代码时, 可以先将写引用进行打包, 而后面有偏移的读引用只需按照依赖距离进行向量寄存器内数据的移位即可省去解包再打包的冗余过程, 从而仍可以生成有效的向量化代码.

考虑依赖环的例子, 如程序 9, 与其对应的例子是程序 8. 显然, 由于依赖距离小于 4, 程序 9 没有被 GCC 和 ICC 识别为向量化, 而程序 8 则同时被 3 个编译器识别为可向量化. 我们的编译器分别对程序 8 和 9 进行了节点分裂、语句交换、向量寄存器内数据移位等优化而将该循环识别为向量化. 也就是说, GCC 和 ICC 编译器只是通过判定依赖距离是否不小于向量化因子来消除依赖边, 而没有考虑如节点分裂、语句交换等优化, 而我们的识别方法充分利用了这些优化, 从而对如表 1 中复杂的依赖时, 其识别能力更强.

接下来, 分别使用这 3 个编译器对 gcc-vect 测试集中的随机抽取的 100 个例子中的 317 个循环进行测试, 这 100 个例子与 2.1 小节中所选取的例子相同. gcc-vect 测试集是 GCC 编译器开发人员提供的基准测试集, 其功能是用于测试向量化编译器识别程序向量化的能力. 该测试集中包括数百个应用程序, 程序中包括了含有复杂数据依赖关系、数据类型转换以及函数调用等各种实际应用中所涉及的情况, 能够很好地反映一个向量化编译器的识别能力. 我们选取 100 个具有代表性的程序测试结果, 对这 100 个程序中 317 个循环识别的结果如图 5 所示. 从图中可以看出, 本文方法能够将 46.37% 的循环识别为 SLP 向量化, 优于 GCC4.9 的 41.64%, 但低于 ICC14.0 的 55.52%. 值得注意的是, 当识别多层嵌套循环时, GCC4.9 更倾向于向量化外层循环, 而 Open64-5.0 和 ICC14.0 更倾向于识别内层的循环.

因此, 对 ICC 能够识别为向量化, 而本文方法无法识别为向量化的循环进行了分析, 发现本文方法不能够识别为 SLP 向量化的循环有以下几类: (a) while 循环; (b) 含有结构体数组引用的循环; (c) 调用内函数的循环; (d) 向量化语句中含有归纳变量的循环. 对于第 1 种情况, Open64 在分析过程中试图将所有循环转换为标准 Fortran DO 循环的形式, 而有些 while 循环无法被转换成这种情况, 而能够被成功转换的循环则可以被正确识别; 对于第 2 种情况, 涉及到指向分析的问题, 我们只是沿用 Open64-5.0 编译器自身的分析过程, 没有再深入扩展, 因此存在部分结构体和指针无法识别的情况; 对于第 3 种情况, 循环内包含对系统自带的数学函数 pow 的调用, 对于这类函数调用过程, 我们通过将其转换为 sqrt 和 square 函数进行识别. 下面重点讨论最后一种情况.

如前文所述, GCC 和 ICC 中同时实现了传统向量化和 SLP 向量化, 而 Open64 本身实现了传统向量化, 对于语句中含有归纳变量的循环, 我们曾试图将其识别为 SLP 向量化, 但此时对这些语句进行打包时开销较大, 而 Open64 本身的传统向量化功能可以将其识别为向量化, 并且不需要进行打包, 因此在 SLP 阶段向量化识别阶段, 取消了对这种语句的 SLP 向量化识别, 而将其交给传统向量化处理, 这与 GCC 和 ICC 中的做法一致. 我们将 Open64 中传统向量化的开关打开并再次对这 100 个程

4) ICC14.0 能够对程序 3 实施部分向量化, 前提是对该程序进行循环分布和语句重排序.

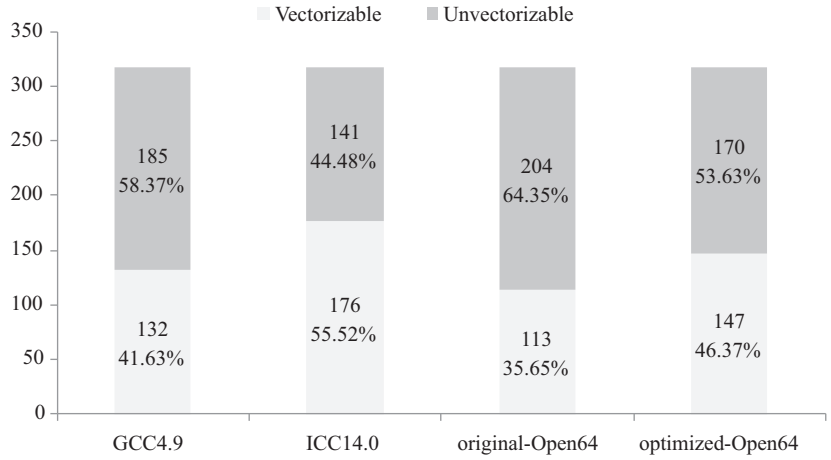

图 5　识别归约前编译器识别 SLP 向量化的能力对比

Figure 5　Comparison of SLP vectorization detection by optimized-Open64 without the reduction operation

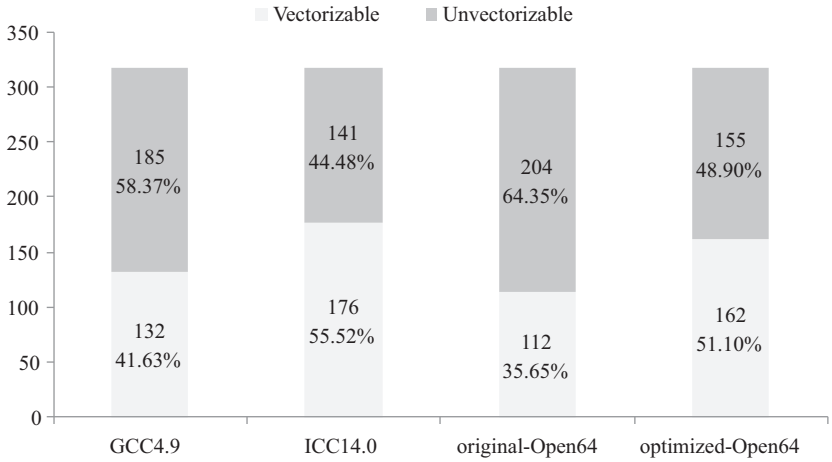

图 6　识别归约后编译器识别 SLP 向量化的能力对比

Figure 6　Comparison of SLP vectorization detection by optimized-Open64 with the reduction operation

序进行了识别, 此时, 本文方法能够识别的个数已达到 51.10%, 测试结果如图 6 所示.

从中可以看到, 此时本文方法在识别数量上已接近 ICC14.0 识别的个数. 对于这 3 个编译器都无法识别为向量化的循环, 我们也进行了分析, 结果如图 7 所示. 结果表明, 我们的编译器受限于 while 循环、内函数和结构体数组引用, 而 GCC 和 ICC 充分考虑了这些情况; 而在处理伪依赖的能力上, 由于我们充分考虑节点分裂、语句交换以及向量寄存器内数据移位等各种优化, 能够在一些有循环携带真依赖的情况仍能够进行向量化, 而 GCC 编译器在这方面实现得还不够完善. 从图中可以看出, 本文提出的方法在不考虑上述 4 种原因的情况下无法识别为向量化的循环个数, 即因无法有效判定依赖而导致无法将循环识别为 SLP 的个数, 已经与 ICC14.0 相同.

5.3　应用程序向量化性能测试

本节利用本文提出的方法对实际应用程序进行变换, 并与 ICC14.0 和 PathScale5.0[5)]编译器生成

5) PathScale EKOPath compiler suite. http://www.pathscale.com/ekopath-compiler-suite.

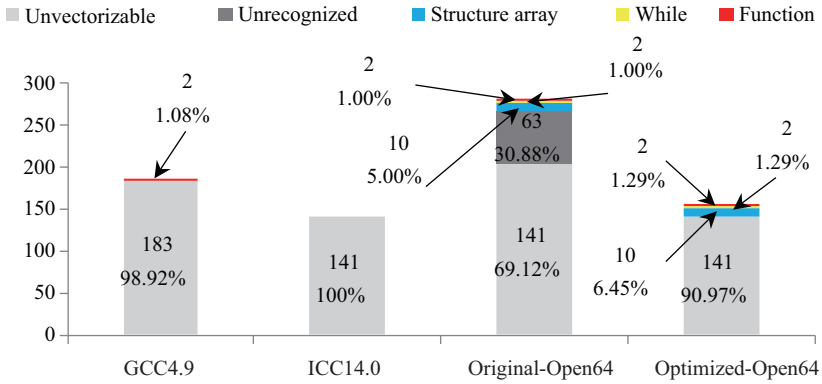

图 7 (网络版彩图) 无法识别为向量化的情况

Figure 7 (Color online) Details of unvectorizable loops

的代码性能进行比较. 除此之外, 还实现了 Liu 等 [7] 提出的向量化识别方法. 实验对比方法是首选运行串行程序, 并记录执行时间, 然后利用我们的方法和上述几种编译器和优化算法进行代码生成, 执行向量程序并计算性能提升.

采用的实验方法如下: 首先, 在关闭向量化开关的条件下, 用 Intel ICC14.0 和 PathScale5.0 编译串行程序并记录运行时间, 然后打开向量化开关, 记录向量化程序的运行时间; 其次, 用本地编译器编译[6)]串行程序并记录运行时间, 然后再用本文提出的方法和 Liu 等提出的方法将串行程序变换为向量化程序, 将生成的向量化代码交给本地编译器编译并运行, 记录向量化程序运行时间. 分别计算向量化加速比, 并进行比较. 实验中使用的应用程序包括一个小型的 FFT 程序, 一个大型 OpenCFD 程序和从 SPEC2006 和 Nas Parallel Benchmarks 测试集抽取的 16 个基准程序, 这些应用程序的具体信息如表 2 所示.

GCC4.9 和未优化的 SLP 向量化算法对这些程序基本无性能提升, 因此本节未给出这两种方法的实验结果.

对于 FFT 程序, 所有的编译器和算法在分析依赖关系时, 都判定出其核心循环携带真依赖, 由于依赖距离小于向量化因子, 本文方法能够将该循环识别为向量化, 而其他方法却无法对其进行向量化. 因此, 本文方法对 FFT 程序有 46.4% 的性能提升, 而其他方法无法进行向量化.

对于 OpenCFD 程序, 共有 31 个循环无法被其他方法向量化, 这些循环中都具有携带依赖并且依赖距离小于 4, 而使用本文提出的方法, 能够正确识别并变换其中的 12 个循环, 最终, 本文提出的方法识别出的可向量化循环个数多于其他方法, 并得到 80.4% 的性能提升. 对其他应用程序, 基于同样的原因, 我们比现有的向量化编译器和 Liu 等提出的优化算法得到了更好的性能. 由于本文方法需要先根据数组依赖图和语句依赖图构建扩展依赖图, 因此在编译时间上有所延迟, 在整个 SLP 向量化识别过程中, 添加了本文算法之后, 该部分的编译时间增加了约 23%.

与 ICC14.0 相比, 本文方法在 hmmer 和 libquantum 这两个程序上分别有 7.6% 和 12.1% 的性能下降. 虽然本文方法能够识别出含有携带依赖并且依赖距离小于向量化因子的循环, 但是, 由于目前 Open64-5.0 编译器的向量化识别能力受限于结构体数组的识别, 即使采用了本文提出的方法之后, 其识别出的向量化循环个数仍少于 ICC14.0 编译器, 因此, 这些程序的向量化效果较 ICC14.0 稍逊. Liu

6) 利用本地编译器编译时关闭所有优化开关, 以此消除由此可能带来的影响.

表 2 测试用例

Table 2 Test suite details

No.	Program	Description
1	FFT	Fast Fourier transformation
2	OpenCFD	Numerical simulation on compressible turbulent boundary flow
3	BT	Block tri-diagonal solver for computational and data movement
4	FT	3D partial differential equation solution using fast Fourier transformations
5	LU	Lower-upper Gauss-seidel solver for computational and data movement
6	SP	Scalar penta-diagonal solver for computational and data movement
7	433.milc	Simulations of four dimensional lattice gauge theory
8	434.zeusmp	Solver for equations of hydrodynamics and magnetohydrodynamics
9	435.gromacs	Simulation of the Newtonian equations of motion
10	436.cactusADM	Solver for the Einstein evolution equations
11	450.soplex	Solving a linear program using the Simplex algorithm
12	453.povray	Calculating an image of a scene
13	454.calculi	Solver for linear and nonlinear three- dimensional structural applications
14	456.hmmer	Searching for patterns in DNA sequences
15	458.sjeng	Playing chess and several chess variants
16	462.libquantum	Simulation of a quantum computer
17	464.h264ref	A reference implementation of H.264/AVC
18	470.lm	Using lattice Boltzmann method to simulate incompressible fluids

等提出的优化算法对于程序 poverty 和 calculix 得到了最优的效果, 这是因为他们的算法借助指令调度策略来得到最佳的数据布局, 从而降低了数据访存延迟. 对于这两个程序, 本文提出的方法提升的性能分别落后 2.4% 和 1.4%. 测试结果如图 8 所示.

6 相关工作

SLP 向量化方法首先要识别基本块内的同构语句, 然后将这些同构语句进行打包形成超字指令, 通过执行超字指令来实现并行执行的目的. Larsen 和 Amarasinghe [4] 在基本块内识别同构语句, 因此得到的是一种局部最优的打包方案. 研究 [4, 8] 表明, 超字打包和解包开销是降低 SLP 向量化性能的一个关键因素, 如何降低打包和解包带来的开销是提升 SLP 向量化性能的一种有效手段. 因此, Liu 等 [7] 提出了一种在整个程序内全局最优的超字指令识别方法, 以此来提高超字指令的重用性, 并减少指令打包和解包的次数. 为了进一步提高 SLP 向量化的效率, 他们还提出了一种数据布局优化手段, 以此来降低实施 SLP 向量化时内存访存的开销. 与本文工作相比, 他们并未对 SLP 向量化的识别过程本身进行更深入的分析, 而是借助数据局部性来提高程序执行效率.

针对降低内存访存开销的问题, Shin 等 [9, 10] 面向 Intel SSE 以及 Motorola AltiVec 等支持超字指令的硬件结构提出了一种数据布局优化手段, 通过将向量寄存器视为一种编译器控制的 cache, 利用向量寄存器重用来减少内存访存开销. 他们不仅考虑了时间开销, 而且也考虑了空间开销. 此外, 他们还研究了如何在含有控制流语句的条件下开发 SLP 向量化 [8, 11]. 考虑到 SLP 是面向基本块的向量化方法, 他们通过推测分析来实现 if 指令转换并得到更大的基本块, 从而获得更大的 SLP 并行性. 由于生

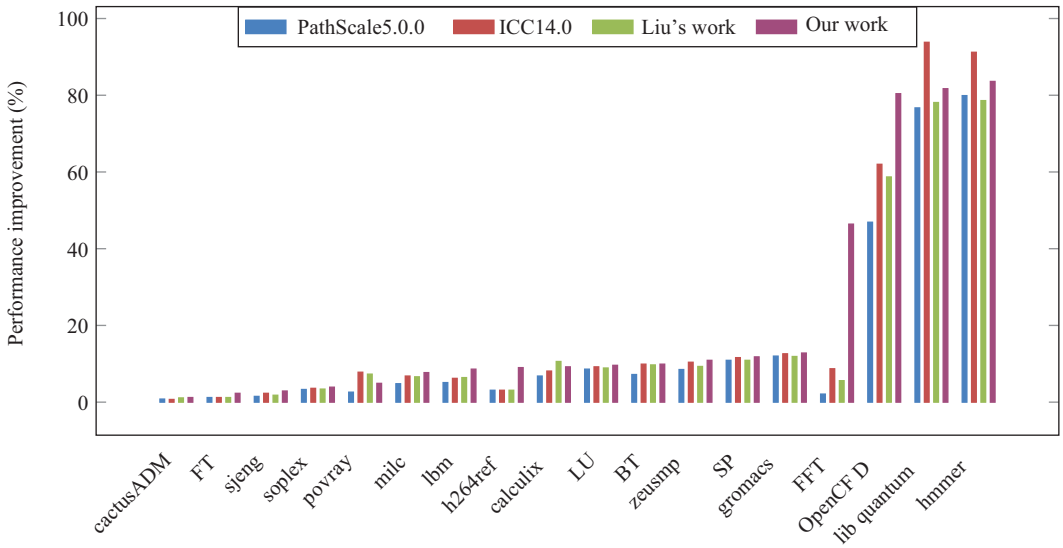

图 8 (网络版彩图) 与当前最优先进算法的比较

Figure 8 (Color online) Performance comparison among different methods

成的指令中通过 select 语句判定原程序中的条件, 他们还提出了一种如何减少 select 指令数目的算法. 与他们的方法相比, 我们的工作不依赖于目标硬件平台, 而且能够在将控制流转换为数据依赖时得到更精确的分析结果.

Karrenberg 等 [12] 提出了一种使用 SSA 形式标记和选择指令的方法, 实现任意控制流图的向量化. Bik 等 [13] 提出了一种比特标记的方式分析不同控制流分支的计算结果, 从而进行控制流的向量化. 此外, Tenllado 等 [14,15] 研究了如何在内层循环携带依赖的情况下识别向量化, 使 SLP 能够适应数字信号处理和多媒体技术应用程序. 然而, 这些方法都局限于数组依赖图和语句依赖图的分析, 在识别向量化的时间复杂度和精度上都不如依赖于扩展依赖图的结果.

非连续和跨幅访存数据的计算在实际应用程序中常常出现, 向量寄存器不支持跨幅访存数据的操作, 但是如果数据布局合理就能够实现跨幅访存数据的向量化. 因此, Nuzman 等 [16] 实现了一种面向跨幅访存数据的自动 SIMD 向量化方法. 通过扩展传统的向量化方法, Nuzman 等的工作能够处理步长是 2 的整数次幂的循环, 与之前研究人员所认识到的不同的是, 只要向量化因子足够大, 跨幅访存 (如步长是 16 或 32 的情况) 能够被很好地向量化. 我们的方法在进行分析之前, 已经将大于向量化因子的依赖边剪去, 因此很容易实现这些满足特定条件的非连续访存数据的向量化识别.

Nuzman 和 Zaks [17] 还研究了如何实现嵌套循环中外层循环的向量化. 对于某些科学计算而言, 外层循环可能更适于向量化, 而之前的方法则往往只关注最内层循环的向量化, 而忽视了外层循环向量化. 外层循环向量化的方法有多种, 如 Nuzman 和 Zaks 提出的方法是在外层循环所在位置直接进行向量化, 通过按向量因子对待向量化的循环进行展开, 并对循环体内的语句, 包括内层循环, 进行压紧, 这种方法适于短向量 SIMD 部件体系结构. Allen 和 Kennedy [3] 提出的将外层循环和内层循环同时进行向量化的方法, 为外层循环向量化的工作奠定了基础, 该方法要求被外层循环向量化的语句必须能被内层循环向量化; 文献 [1,18] 提出的基于循环交换的外层循环向量化, 即通过将待向量化的外层循环与内层循环位置进行交换, 将待向量化的循环交换到最内层再按照内层循环向量化的方法实现, 这种方法适于长向量部件的向量化; Wu 等 [19] 通过消除内层循环来实现外层循环向量化的方法,

该方法通过将内层循环进行完全展开, 使外层循环转换为内层循环从而实施向量化, 但这种方法带来的结果是代码膨胀率较大. 我们对多重循环的处理方式在于对依赖层的精确利用, 借助依赖层的概念消除伪依赖环, 为内外层循环实现向量化提供了方便.

7 结论

SLP 是一种实现 SIMD 向量化的方法, 但目前的 SLP 算法识别 SLP 向量化的能力却无法满足应用需求. 因此, 本文提出了一种基于有向依赖图可达性的 SLP 向量化识别方法, 与传统交替遍历数组依赖图和语句依赖图来判定语句之间依赖关系的方法不同, 本文提出的方法可以大大降低对依赖图中节点的遍历次数, 并且能够根据构建的有向依赖图获得识别 SLP 向量化所需要的各种依赖信息. 实验结果表明, 优化后 Open64 编译器识别 SLP 向量化的能力得到明显提升. 与当前主流的向量化编译器 GCC 和 ICC 相比, 本文提出的方法不仅消去了依赖距离大于向量化因子的依赖边, 而且在识别出依赖关系的情况下采用节点分裂、语句交换等优化技术来消除或转换阻碍向量化的依赖, 因此, 本文提出的方法在识别含有复杂依赖关系的循环时能力更强. 对 gcc-vect 测试集程序的测试结果表明, 本文方法能够识别 SLP 向量化循环的能力优于 GCC4.9 编译器, 与 ICC14.0 相当. 由于受到 Open64 编译器中语言特点的限制, 目前优化后的 Open64-5.0 识别 while 循环和结构体数组等特殊问题的能力还不足, 如何从这两个方面进一步提升 Open64 编译器识别 SLP 向量化的能力将是本文的下一步研究工作.

致谢 在此, 向为本文研究工作提供基础和研究平台的前辈致敬.

参考文献

1 Kahle J A, Day M N, Hofstee H P, et al. Introduction to the cell multiprocessor. IBM J Res Dev, 2005, 49: 589–604
2 Bachega L, Chatterjee S, Dockserz K A, et al. A high-performance SIMD floating point unit for blueGene/L: architecture, compilation and algorithm design. In: Proceedings of the 13rd International Conference on Parallel Architecture and Compilation Techniques. Washington: IEEE Computer Society, 2004. 85–96
3 Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-based Approach. San Francisco: Morgan Kaufmann Publishers Inc, 2001
4 Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. New York: ACM, 2000. 145–156
5 Padua D A, Wolfe M J. Advanced compiler optimizations for supercomputers. Commun ACM, 1986, 29: 1184–1201
6 Bulic P, Gustin V. D-test: an extension to banerjee test for a fast dependence analysis in a multimedia vectorizing compiler. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium. Washington: IEEE Computer Society, 2004: 535–546
7 Liu J, Zhang Y, Jang O, et al. A compiler framework for extracting superword level parallelism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2012. 347–358
8 Shin J. Compiler optimizations for architectures supporting superword-level parallelism. Dissertation for Ph.D. Degree. California: University of Southern California Los Angeles, 2005
9 Shin J, Chame J, Hall M. Compiler-controlled caching in superword register files for multimedia extension architectures. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. Washington: IEEE Computer Society, 2002. 45–55
10 Shin J, Chame J, Hall M. Exploiting superword-level locality in multimedia extension architectures. J Instruction Level Parall, 2003, 5: 1–28
11 Shin J, Hall M, Chame J. Superword-level parallelism in the presence of control flow. In: Proceedings of the International Symposium on Code Generation and Optimization, 2005. 165–175
12 Karrenberg R, Hack S. Whole-function vectorization. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO). Washington: IEEE Computer Society, 2011. 141–150
13 Bik A, Girkar M, Grey P, et al. Automatic intra-register vectorization for the Intel architecture. Int J Parall Prog, 2002, 30: 65–98

14 Tenllado C, Pinuel L, Prieto M, et al. Pack transposition: enhancing superword level parallelism exploitation. In: Proceedings of the International Conference Parallel Computing: Current & Future Issues of High-End Computing, Malaga, 2005. 33: 573–580
15 Tenllado C, Prieto L P M, Tirado F, et al. Improving superword level parallelism support in modern compilers. In: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. New York: ACM, 2005. 303–308
16 Nuzman D, Rosen I, Zaks A. Auto-vectorization of interleaved data for SIMD. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2006. 132–143
17 Nuzman D, Zaks A. Outer-loop vectorization-revisited for short SIMD architectures. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. Washington: IEEE Computer Society, 2008. 2–11
18 Scarborough R G, Kolsky H G. A vectorizing Fortran compiler. IBM J Res Dev, 1986, 30: 163–171
19 Wu P, Eichenberger A E, Wang A, et al. An integrated SIMDization framework using virtual vectors. In: Proceedings of the 19th Annual International Conference on Supercomputing. New York: ACM, 2005. 169–178

Identifying superword level parallelism with directed graph reachability

Jie ZHAO* & Rongcai ZHAO

State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Information Engineering University, Zhengzhou 450001*, China*
* Corresponding author. E-mail: zjbc2005@163.com

Abstract SLP (superword level parallelism) is an efficient solution to exploit the parallelism between statements in the basic blocks of SIMD (single instruction multiple data). It has been implemented in almost all the mainstream vectorizing compilers. However, its vectorization ability is limited due to the conservativeness of the parallelism identification process. To solve this problem, this paper proposes an edg (extended dependence graph) approach to identify SLP. First, we extend the adg (array dependence graph) and sdg (statement dependence graph) to construct the edg, which includes both the dependences between each array pair and those between each statement pair. When a statement is represented as an SCC (strong connected component), all of its array references are also constructed in this SCC. We then eliminate the redundant dependence edges between the SCCs from the edg. The dependence information of each statement pair and its SLP vectorization are thus determined by analyzing the reachability between each node pair from the corresponding SCC. We implement this approach to optimize the Open64-5.0 compiler, which improves the compiler's identification ability. The evaluation tests on the gcc-vect benchmarks show that the optimized Open64-5.0 compiler can identify more SLP vectorizable loops than the GCC4.9, and that the number of vectorizable loops is comparable to that of ICC14.0. The performance of our generated codes is better than the state-of-the-art for most practical applications.

Keywords vectorizing compiler, superword level parallelism, data dependence, directed dependence graph, reachability

Jie ZHAO was born in 1987. He received the Master's degree in computer software and theory from the PLA Information Engineering University, Zhengzhou, in 2002. He is pursuing his Ph.D. degree from the PLA Information Engineering University. His research interests include high-performance computing and parallel compilation.

Rongcai ZHAO was born in 1957. He received his Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2002. Currently, he is a professor at the PLA Information Engineering University. His research interests include high-performance computing, software engineering, and advanced compilation techniques.