一种基于 SLP 的新型编译框架*

张素平，王 冬，丁丽丽，王鹏翔，宫 一，于海宁
（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）

摘 要： 对于 SLP（superword level parallel）算法不能高效处理并行代码占有率较小的大型应用程序的问题，提出并评估了一种新型的基于改进的 SLP 算法的编译框架。它主要包括三个阶段：将代码中结构相似的异构语句通过改进的 SLP 算法尽可能地改为同构语句；用全局的观点，在优化目标代码之前获取其数据模型重用；联合数据布局优化进行进一步的性能提升。针对框架做了大量实验，实验结果表明该框架比 SLP 算法性能更佳，性能提高约 15.3%。

关键词： 超字并行；同构；超字重用；数据布局
中图分类号： TP314 **文献标志码：** A **文章编号：** 1001-3695(2017)01-0021-06
doi:10.3969/j.issn.1001-3695.2017.01.004

New framework based on SLP

Zhang Suping, Wang Dong, Ding Lili, Wang Pengxiang, Gong Yi, Yu Haining
(*State Key Laboratory of Mathematical Engineering & Advanced Computing, PLA Information Engineering University, Zhengzhou* 450001, *China*)

Abstract: Since the SLP(superword level parallel) algorithm could not efficiently deal with the large-scale applications which covered few parallel codes. This paper proposed and evaluated a new compile framework based on the improved SLP algorithm. It contained three phases. First, it tried to transform the non-isomorphic but similar instruction sequences to isomorphic instruction sequences by the improved algorithm as far as possible. Second, it took a global point of view of the target application when capturing the superwords reuse patterns before making the optimization decisions. Eventually, it combined data layout optimization for further performance improvement. This paper did much experiment on the framework. The experimental results indicates that the optimization of the compile framework is better than SLP algorithm, the performance increases about 15.3%.

Key words: superword parallel; isomorphic; superword reuse; data layout

多媒体应用计算大多是计算密集型、适合并行处理的计算，需要先进且高效的并行处理技术。为满足应用需求，越来越多的处理器集成了 SIMD 扩展部件，然而先前的多媒体扩展只能支持较短长度的数据类型，现在新的扩展可以支持 256 bit 甚至 Intel 的 AVX2 可以支持 512 bit 的超字操作。因此，出现了一种新型的可以处理较宽数据类型的数据并行类型超字并行(SLP)[1]。SLP 就是将数据打包成较大的超字，然后用 SIMD 向量指令进行计算操作，它与向量级并行不同，向量级并行适用于含有大量并行代码的应用，而 SLP 则适合并行代码量适中的应用。

本文做了如下工作：首先对 SLP 算法作了一些改进，然后提出了一种高效的编译框架，包含三个阶段：异构语句同构化、组建超字语句和数据布局优化。其中异构语句同构化主要包括寻找 SLP 算法遗漏的向量化机会和构建 SLP 补充图[2]，即先寻找 SLP 算法优化代码时可能遗漏掉的向量化机会，再通过补充一些冗余节点，使得相似但不完全相同的依赖图同构化，构造可以被 SLP 向量化处理的 SLP 补充图。组建超字语句阶段主要包括语句分组和语句调度两部分。语句分组是在尽可能地增加 SIMD 并行性和获得组之间超字重用的前提下为 SIMD 操作分组；语句调度是在尽可能地减少向量寄存器的排序开销的前提下给各个组以及每组中语句排序。数据布局优化阶段则分析输入数据经过前两阶段之后的数据访问模式，并根据其数据访问模式重组内存中的数据，使得 SLP 的内存操作代价最小。本文运行了新型编译框架并做了相关实验，实验结果证明新型框架性能优于 SLP 算法。

本文对 SLP 算法作了简单介绍并叙述了相关研究，给出了编译框架的概述；简要说明了语句同构化的处理过程，详细介绍了如何组建超字语句，并给出了数据布局优化的方法，通过一个示例讨论了实验结果。

1 SLP 及相关研究

1.1 SLP

SIMD 编译优化[3]主要解决如何有效地将各种高级语言代码转换成多媒体扩展对应的高效的 SIMD 指令。向量机[4~9]自动并行的传统编译技术不仅依赖于复杂的循环转换，还只对内含大量并行代码的应用程序起作用，对那些不能并行处理的应用效果甚微。

收稿日期：2016-01-06；修回日期：2016-05-19 基金项目："核高基"国家科技重大专项资助项目(2009ZX01036-001-001-2)

作者简介：张素平(1991-)，女，河南禹州人，硕士，主要研究方向为高性能计算、先进编译技术(892841546@qq.com)；王冬(1991-)，男，硕士，主要研究方向为高性能计算、先进编译技术；丁丽丽(1992-)，女，硕士，主要研究方向为高性能计算、先进编译技术；王鹏翔(1988-)，男，硕士，主要研究方向为高性能计算、先进编译技术；宫一(1987-)，男，硕士，主要研究方向为高性能计算、先进编译技术；于海宁(1989-)，男，硕士，主要研究方向为高性能计算、先进编译技术.

为了解决以上问题，Larsen 等人[9] 提出了一种新型的超字并行方式 SLP，其目标是解决多媒体扩展应用问题。SLP 算法主要包括五个步骤：寻找可以作为向量化种子的指令代码；根据种子指令的数据依赖图，为可向量化指令进行分组；评估标量和向量的代码性能；比较两种形式的代价；如果向量化是有收益的，用等价的向量代码代替标量代码。SLP 的一个很重要的优势为即便代码的并行比例适中，它依然可以高效地使用多媒体扩展指令来处理。

1.2 相关研究

Shin 等人[10,11] 开发了一种策略来管理向量寄存器文件，让它作为一个编译控制的缓存，使之在利用 SLP 时提高对本地数据的管理效力；而且，他们[12,13] 在使用当前控制流的指令断言时派生出一种大的基本块，用以识别更多的超字级并行。Tenllado 等人[14,15] 提供了一种技术可以在内层循环存在循环携带依赖时高效地利用 SLP。Nuzman 等人[16] 研究了一种可以支持插入数据的高效向量化的编译技术。Nuzman 等人[17] 还研究了短 SIMD 框架的外层循环向量化技术。Barik 等人[8] 提出在接近机器层的底层 IR 上进行自动向量化，实现在动态编译时获得更高的编译效率。本文在原始的 SLP 算法的基础上提出了一种更加全面且不同的提取 SLP 的策略。

2 整体框架概述

首先输入一个基本块集，然后经过以下几个优化过程处理该基本块集：a) 将相似却异构的依赖图同构化；b) 通过生成更多的超字语句增加并行；c) 通过得到更多的超字重用来减少超字打包/解包的数目；d) 通过数据布局优化减少强制打包/解包及内存中重排序指令的开销。

图 1 给出了本文的编译框架的概述，主要有四个模块：预处理、同构化处理、全局 SLP 优化[18]、后端处理。输入程序源代码，预处理模块对输入进行循环展开和对齐分析，暴露出更多 SLP 利用的机会；同构化异构相似依赖图，应用全局 SLP 优化程序并生成向量化代码；后端处理模块实现寄存器分配和其他的底层优化并输出。

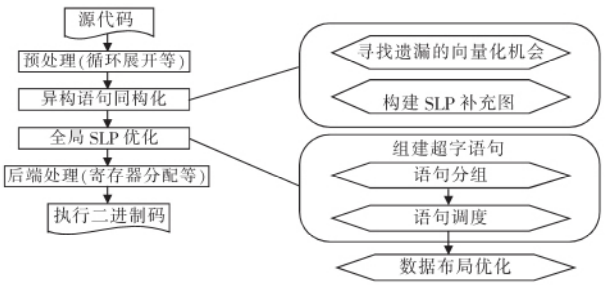

图 1 新型框架概述

3 同构化处理

3.1 寻找 SLP 遗失的向量化机会

SLP 算法可以同时处理多条同构指令，但它仍然可能遗漏掉很多优化的可能，因为在它之前程序已经经历过很多优化，这些优化很可能将本阶段的向量化机会给消除了。例如，早期的冗余节点删除很可能使得本来在此阶段同构能被 SLP 处理的依赖图变成异构的不能被 SLP 处理的依赖图，从而遗失了本来可以向量化的机会。所以，首先要将语句构成依赖图，然后尝试拟组建超字组，尽可能地找出 SLP 算法遗漏的向量化机会。

3.2 构建 SLP 补充图

当找到 SLP 算法遗失的向量化机会时，就添加冗余节点将异构语句同构化，构建 SLP 补充图。构建 SLP 补充图需要解决以下几个问题：

a) 构建最大通用子图。寻找两个图的最大通用子图被认为是一个 NP 的同构问题。可以用 McGregor 的算法[19] 来提供解决方案。实际情况中图都非常小，所以本文提出了一个接近于最优算法的快速 backtracking 算法。算法思路如下：

首先输入两个图 g_1 和 g_2；然后给图节点分类，对其进行自底向上的遍历；最后调用算法的主递归函数，从图 g_1 的一个节点开始，在另一个图 g_2 中寻找与之匹配的节点，在搜索过程中，跳过以下节点：已经匹配的节点、不匹配的类型、会产生环的节点、不同基本块内的节点、不能被并行调度的节点。

一旦找到两个匹配节点，就将它们作为候选并插入 g_1 和 g_2 的映射中进行反转，然后在给定的范围内找到其他可能的匹配对并开始新的搜索。为降低复杂性，忽略那些流动性高于临界值很多的节点，只考虑最有可能有收益的节点匹配对。

当算法跳出循环，节点映射结束，算法会再创建两个边映射(映射 g_1 和 g_2 中通用的子图边)。在边和节点映射的基础上，过滤掉映射图中不存在的节点和边生成最大通用子图，然后比较目前最好的子图并找出其中最大的那个。

b) 构建最小通用超图。用最大通用子图来计算最小通用超图。将最大通用子图与它和原图之间的差异相结合得到最小通用超图。例如，如果 g_1 和 g_2 是原图，maxCS 是 g_1 和 g_2 的最大通用子图，那么 $\text{minCS1} = \text{maxCS} + \text{diff1} + \text{diff2}$，其中 $\text{diff1} = g_1 - \text{maxCS}$，$\text{diff2} = g_2 - \text{maxCS}$。

c) 选择插入节点。SLP 补充图发射选择指令在边参数流进节点之前为其选择需要流进的边参数。为保证原图语义不变，选择那些属于原节点的 diff。

d) 移除冗余。补充图的最后阶段是删除那些可以被优化掉的选择指令。这个过程可以实现四种选择指令优化，优化的结果是删除代码中的冗余选择指令以提高性能。a) 用选择的常量替换常量；b) 选择有相同处理器节点的边；c) 选择读常量识别操作的指令和可以在选择指令与本指令通用的指令节点；d) 降级选择节点；e) 构建多图的最小超图。SLP 补充图的构建是用一种快速但并非最理想的方法计算多个图的最小超图，计算一组图的最小超图并将它作为下一组图的左图。重复以上步骤，得到多图的补充图。

4 组建超字语句

4.1 语句分组

用一个迭代过程来实现语句分组。首先使用一个块分组算法找到一个大小为 2 的组，然后将已经存在的组作为一个原子语句，继续应用块分组算法得到更大块的组，直到 SIMD 数据宽度完全被利用为止。分组阶段主要分为基本分组和迭代分组两部分。

4.1.1 基本分组算法

以下是使用基本分组算法得到大小为 2 的基本块的一个示例。

$$S_1: V_1 = V_3; S_2: V_2 = V_5; S_3: V_5 = V_7$$
$$S_4: V_4 = V_1 + V_1; S_5: V_5 = V_2 + V_5$$

1) **识别候选组**　一个候选组是指包含两个同构语句$\{S_i, S_j\}$，且$S_i, S_j \in S$的 SIMD 指令组。候选组的超字大小不应该超过目标 SIMD 数据宽度，且候选组的同构语句之间没有顺序。例如，基本块示例中代码的候选组集是$C = \{\{S_1, S_2\}, \{S_1, S_3\}, \{S_4, S_5\}\}$。

2) **构建变量打包冲突图**　一个变量打包是指一个候选组中不同语句相同位置的一组变量集合。基本块示例中，来自$\{S_1, S_2\}$的变量集$\{V_1, V_2\}$和$\{V_3, V_5\}$是超字的候选集。变量打包冲突图$VP = (V, T)$构建过程如下：依次遍历C中的候选组；对于每一个候选组$\{S_p, S_q\}$，创建一个由语句生成的所有变量包的节点集，并且在新建节点与原节点之间插入边。图 2 显示了基本块示例代码中的候选组集所生成的变量打包冲突图。

3) **使用早期识别的候选组和先前得到的变量打包冲突图来建立语句分组图**$SG = (V', T')$　图中的每个节点$(\in V')$表示基本块中的一个语句S_p，每条边$(\in T')$表示两条语句之间的候选组。语句分组图是一个加权图，每条边的权值代表它可能带来的收益，如图 3 所示。为了计算候选组$\{S_p, S_q\}$中的两条语句S_p和S_q之间边的权值，提取变量打包图VP中的所有包(节点)来构建一个辅助图。假定要计算图 2 中语句S_4和S_5之间的边的权值，其辅助图如图 4 所示。它包括图 2 中所有的节点，这些节点与$\{S_4, S_5\}$中的变量包相同但却并不冲突。通过联合辅助图中剩余的节点$\{V_1, V_2\} - \{S_1, S_2\}$和$\{V_3, V_5\} - \{S_1, S_2\}$，以及$VP$图中的候选组的变量包$\{V_3, V_5\} - \{S_4, S_5\}$，$\{V_1, V_2\} - \{S_4, S_5\}$和$\{V_1, V_5\} - \{S_4, S_5\}$，可以得到整个基本块中来自候选组$\{S_p, S_q\}$的变量包(超字)的平均重用值。例如，候选组$\{S_4, S_5\}$的值是 2/3，重用值是图$SG$中语句$S_p$和$S_q$之间边的权值。因此，语句在基本块(全局影响)中的潜在收益(超字重用)是由最终代码将它们分组为超字时所得的。

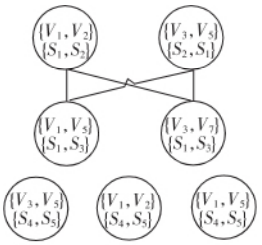

图 2　基于块示例代码的变量打包冲突图

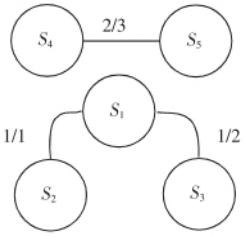

图 3　基于块示例代码的语句分组图

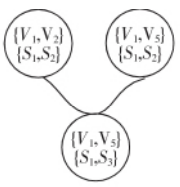

图 4　计算候选组$\{S_4, S_5\}$的权值的辅助图

4) **基于之前所构建的分组图**SG**建立分组决策**　根据递减权值筛选图SG中所有的边，选择其中权值最大的边，如果有两条边权值相同，就随机选择一个。一旦作了决策，就更新SG图和VP图，删除SG图中 SIMD 组刚找到的语句节点以及所有与之相关联的节点(如冲突语句)，如图 5 所示；删除VP图中由最新决策 SIMD 组生成的变量包的节点以及与之相关联的节点(如冲突变量包)，如图 6 所示；然后重新计算SG图中所有边的权值。重复这个过程直到SG图中没有剩余边为止。

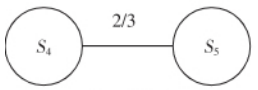

图 5　第一个分组决策$\{S_1, S_2\}$的语句更新图

图 6　第一个分组决策$\{S_1, S_2\}$之后的变量打包冲突图

算法 1 是基本分组算法的伪代码。算法的第一行识别候选语句组，第 2 ~ 10 行构建变量打包冲突图，第 11 ~ 17 行初始化语句分组图；算法的关键部分是第 20 ~ 41 行进行分组决策，其中第 21 ~ 29 行基于当前候选组与既定组构建权值计算辅助图；辅助图中的冲突在 30 行被解决，31 ~ 37 行用来计算权值(平均超字重用)；第 39 ~ 40 行选择最大权值的边作为当前分组决策，然后更新变量打包冲突图和语句分组图；第 20 ~ 41 行重复以上工作直到所有的组都确定下来。基本分组算法的复杂度是$O(E_{SG}^2 \times N_{VP})$，其中$E_{SG}$是语句分组图中边的数目，$N_{VP}$是变量打包冲突图中节点的数目。

算法 1　基本块分组阶段伪代码

输入：一个基本块语句序列，$S = \langle S_1, S_2, S_3, \cdots, S_n \rangle$。
输出：一组语句分组D'。

```
1:  识别候选语句组 g;
2:  初始化 VP; //VP 为变量打包冲突图
3:  for {Si,Sj} ∈g do
4:    for{Vi,Vj},Vi∈Si.v 且 Vj∈Sj.v do
5:      VP.v←VP.v∪{{Vi,Vj}-{Si,Sj}};
6:      for{Vq,Vr}-{Sq,Sr} 且{Sq,Sr}∩{Si,Sj}≠0 do
7:        VP.ε←VP.ε∪{{Vi,Vj}-{Si,Sj},{Vq,Vr}-{Sq,Sr}};
8:      end for
9:    end for
10: end for
11: 初始化 SG; //SG 为语句分组图
12: for {Si,Sj}∈g do
13:   for S∈{Si,Sj} 且 S∉SG.v do
14:   SG.v ← SG.v∪{S};
15:     end for
16:     SG.ε←SG.ε∪{Si,Sj};
17: end for
18: D'←0;
19: while SG.ε≠0 do
20: for {Si,Sj}∈SG.ε do
21:   初始化 AG; //AG 为辅助图
22:   for{Vi,Vj}-{Sq,Sr} ∈VP.v do
23:   if {Vi,Vj}-{Si,Sj} ∈VP.v ,{Sq,Sr}≠{Si,Sj} 且{{Vi,Vj}-{Si,Sj},{Vi,Vj}-{Sq,Sr}}∉VP.v then
24:       AG.v←AG.v∪{Vi,Vj}-{Sq,Sr};
25:     end if
26:   end for
27:   for Pm,Pn∈AG.v 且{Pm,Pn} ∈VP.ε do
28:   AG.ε←AG.ε∪{Pm,Pn};
29:  end for
30:  解决 AG 中的冲突;
31:  //计算{Si,Sj}的重用超字.
32:  r←0;
33:   for{Vi,Vj} ∈AG.v or D'∪{Si,Sj} do
34:     r←r+(N-{Vi,Vj}-1); //N-{Vi,Vj}:{Vi,Vj}出现的次数。
35:  end for
36:  W-{Si,Sj} = r/Nt; //Nt 为 D'∪{Si,Sj}中的打包数
37:  SG.w ←SG.w∪{W-{Si,Sj}};
38: end for
39: D'←D'∪{Si,Sj}; //W-{Si,Sj} = max(SG.w).
40: 更新 VP 和 SG;
41: SG.w←0;
42: end while
```

4.1.2　迭代分组

基本分组算法生成的 SIMD(超字语句)的大小为 2。为了解决其不能充分利用特定框架下 SIMD 数据宽度的问题，本文用一个迭代过程来获得更大的 SIMD 组以扩充基本分组算法。其基本的观点是，第一遍之后，把每一个 SIMD 组$\{S_p, S_q\}$作为

一个单独的语句,每一个变量包作为一个新的单一变量;增加新语句更新输入基本块中原始语句集,使用更新过后的语句集再次对输入基本块使用基本分组算法。迭代地使用这个策略直到最终生成的超字语句可以最大可能地使用 SIMD 数据宽度。所以,即便在未来 SIMD 扩展的宽度会增加,本文的框架也能够高效地利用它。

4.2 语句调度

分组将语句分组为超字语句,通过获得更多的超字重用来减少昂贵的内存访问操作的使用。但如果相同的数据在两个超字中的顺序不同,将用到额外的向量寄存器重排序指令。调度阶段主要解决两个问题:a)为基本块中所有的语句(包含单字语句和超字语句)安排一个有效的执行顺序并且使得超字语句中的超字重用尽可能地近;b)固定超字语句中的语句顺序尽可能地减少重排操作的数目。

基于输入基本块原始语句的执行顺序,本文在生成的超字语句之间构造一个依赖图。在这个图中,每个节点表示一个超字语句,每个方向边代表两个相关联的超字语句之间的依赖。依赖图可以提供很多不同的调度,可以充分利用它的灵活性来获得更多的收益。基于这个依赖图,调度超字语句并且决定它们的执行顺序。定义一个当前超字集作为与当前向量寄存器中最为接近的超字集。在当前超字集中,每个超字中的语句顺序是既定的。起初,当前超字集为空,从依赖图中提取依赖关系已经解决的超字语句作为目标执行序列的候选语句;然后计算每个候选超字语句中超字重用的数目以及当前超字集中超字重用的数目。拥有最大超字重用数目的候选语句作为下一个要执行的语句,找到向量寄存器中近距离且高可能性的超字重用,从而决定超字语句的执行顺序。通过插入新的有序超字和访问相同数据的现存超字来更新当前超字集,然后通过提取新的已存在的超字继续重复以上步骤直到依赖图中所有节点都被处理完为止。

通过将每个超字语句中语句的排序决策从分组阶段隔离出来并将它延迟到调度阶段,使得本文的框架可以充分利用直接和间接的超字重用。间接的超字重用是关键的,因为它通过引入寄存器重排指令而避免访存操作。在实现基本块分组和调度之后,考虑所有要素,应用一个与文献[3]中相似的代价模型来评估转换后的代码的加速。如果该转换会降低程序性能,则不使用它。算法2是调度阶段伪代码,其中第1~9行用检查组来形成依赖图,第10~13行初始化就绪超字语句集;第15~18行联合当前超字语句集,选择高超字重用数目的超字语句集作为执行语句序列中的下一个执行语句;第19~27行基于尽可能少的重排序操作为所选的超字语句排序;第28~35行更新当前超字集、依赖图以及就绪超字语句集。然后算法继续向前决定下一个将要被放进语句执行序列的超字语句。

算法2 调度阶段伪代码

输入:一组语句分组 D'。

输出:D'的语句分组顺序以及每个分组 D 中的语句顺序 Q,其中 $D \in D'$。

```
1:  初始化 DG; //DG 为分组依赖图
2:  for D ∈ D' do
3:      GDG.v←GDG.v∪{D};
4:  end for
5:  for D_i,D_j ∈ GDG.v do
6:      if D_i 依赖于 D_j then
7:          GDG.ε←GDG.ε ∪ ⟨D_i,D_j⟩;
8:      end if
9:  end for
10: 初始化 LP 和 RD; //LP 为现存的打包,RD 为准备就绪的分组
11: for D_i ∈ GDG.v 且不存在 ⟨D_j,D_i⟩ ∈ GDG.ε do
12:     RD←RD∪{D_i};
13: end for
14: while RD = 0 do
15: for D ∈ RD do
16:     N_r = (D,LP) 中重用超字的数目;
17: end for
18: 选择有最大重用超字 N_r 的 D;
19: //决定 D_i 中语句的顺序;
20: Q'←0; //Q'为 D 的候选序列集
21: for D 的每一个序列 Q,当 D 可以直接重用 LP 中的一个包时 do
22:     Q'←Q'∪{Q};
23: end for
24: for Q ∈ Q' do
25:     N_r = (D,Q,LP) 的重排序的数目;
26: end for
27: 选择 D 中具有最小 N_p 的 Q;
28: for D 中的每一个包 ⟨V_i,V_j⟩,按照 Q 的顺序 do
29:     LP←LP∪⟨V_i,V_j⟩;
30:     if ⟨V_q,V_r⟩ ∈ LP 且 ⟨V_q,V_r⟩ = ⟨V_i,V_j⟩ then
31: LP←LP_{-⟨V_q,V_r⟩};
32:     end if
33: end for
34: 更新 GDG;
35: 更新 RD;
36: end while
```

5 数据布局优化

前面章节中的优化可以减少打包/解包操作的数目,但是并不能完全地删除它们。SLP 算法[1]对本文所述的超字重用来说有很弱的解决效力,而且它高度依赖于当前现存的数据布局。由此引入数据布局优化,通过减少强制打包/解包所带来的内存操作和寄存器指令的数目来增加收益。

数据布局优化的基本观点是用一种特定的方式来组织内存中的超字,使得在向量寄存器加载它们(或者将它们从向量寄存器写回内存)时所花费的开销最小。如图7所示,假定 SIMD 的宽度可以装载两个基本数据类型,本文在第一阶段之后将语句 S_1 和 S_2 组成一个可以生成两个超字的组 $\langle a,b\rangle$ 和 $\langle A[4i],A[4i+3]\rangle$。假定这两个超字在当前执行时不在向量寄存器中,在应用 SIMD 操作之前/后,它们需要被打包/解包装入向量寄存器/从向量寄存器中取出。此时,如果变量 a 和 b 在内存中相邻且对齐,则仅需要一个内存操作来写回 $\langle a,b\rangle$ 即可。本文主要优化两类数据的数据布局,即标量超字和数组引用超字,用地址对齐来处理标量超字,用数据转换和复制来处理数组引用超字。

```
S1: a = A[4i];
S2: b = A[4i+3];
```

图7 数据布局优化示例

6 实验评估

图8给出了新型编译框架对一个基本块的优化过程以及与原始的 SLP 算法的对比,假定一个超字可以装载两个变量,图8(a)是源代码。

图8中第一步转换由 SLP 算法实现。将内存中相邻的语

句作为种子指令识别同构语句{〈S_1,S_4〉},根据定义使用链和使用定义链获得额外的超字语句,例如〈S_2,S_5〉,同时获得语句〈a,b〉的超字重用。最终生成的超字语句是{〈S_1,S_4〉,〈S_2,S_5〉,〈S_3,S_6〉,〈S_7,S_8〉},如图 8(b)所示。在优化的代码中,有一组超字重用可以节省一个打包操作,即超字〈S_2,S_5〉可以通过使用超字〈S_1,S_4〉中的〈a,b〉而节省一个打包操作。

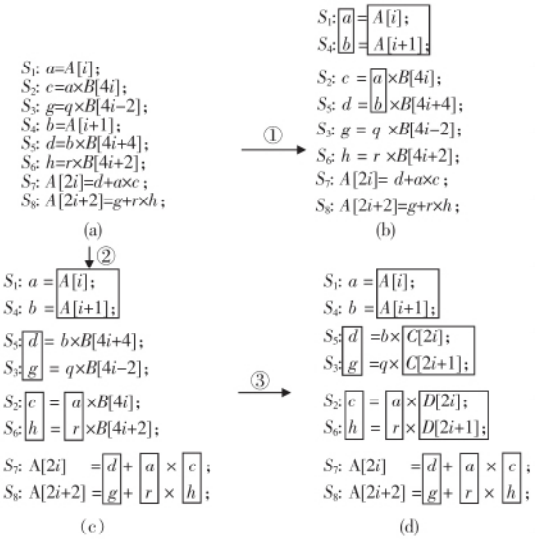

图 8　整体优化示例

图 8 中的第二步转换是应用新型框架优化源代码,经过框架的异构语句同构化和超字组建过程,可得如图 8(c)的超字,即{〈S_1,S_4〉,〈S_5,S_3〉,〈S_2,S_6〉,〈S_7,S_8〉}。图 8(b)与(c)的不同主要是对语句{S_2,S_3,S_5,S_6}的分组决策不同。本文选择将它们分为{〈S_5,S_3〉,〈S_2,S_6〉}而不是{〈S_2,S_5〉,〈S_3,S_6〉},这么分组可以使整个基本块得到更多的超字重用。用一种全局的观点对语句进行分组,使得在代码转换阶段(图 8(c)所示)可以得到三组超字重用{〈d,g〉,〈c,h〉,〈a,r〉},与图 8(b)中只有一组超字重用相比,此方法可以节省更多的打包/解包操作。

第三个阶段显示了代码的数据布局优化阶段,如图 8(c)所示。通过对标量超字(〈a,b〉,〈d,g〉,〈c,h〉)以及数组引用超字(〈$B[4i+4]$,$B[4i-2]$〉,〈$B[4i]$,$B[4i+2]$〉)的优化,可以节省更多的打包/解包操作。生成的代码如图 8(d)所示,其中数组 C 和 D 应用数据复制和数据重命名来构建。在图 8(d)中出现但(c)中没有的组就是由数据布局优化得来的。例如,当对 S_1 和 S_4 应用 SIMD 向量化时,图 8(c)引入了标量引用 a 和 b 的解包开销,但是图 8(d)则不会,因为 a 和 b 经过数据布局优化在内存中已经相邻了并且可以一次性地从向量寄存器写回内存。

6.1　实验环境

本文用 16 个标准测试集在两个系统上对新型编译框架进行了测试。并在 SW-VEC 编译结构上执行了此框架。与文献[1]中提出的算法进行了对比,在两次执行中,为了在编译时有更多的超字级并行,在预处理部分实现了对齐分析和循环展开等优化。两次执行中,本文将寄存器混洗/重排操作与由底层框架支持的指令集而非物理内存支持的加载/存储操作相映射。在下面的讨论中,将新型框架的同构化异构语句与从全局观念进行优化称为全局优化,将 SLP 算法的优化称为 SLP,将所有优化联合称做整体优化,将不做任何优化的代码执行叫做本地执行。最后,在实验中,本文在两个系统中将这三组优化与本地执行作了对比,考虑到编译开销问题,与 SLP 优化相比,本文的方法平均增加了 26% 的编译时间。

实验中使用英特尔 Dunnington 和 AMD 的 Phenom Ⅱ 两个系统,它们都支持 SSE/SSE2 指令集,且都包含一组 128 bit 的向量寄存器,可以一次运行两个 64 bit 操作(如双精度浮点操作),四个 32 bit 操作(如单精度浮点操作和整型操作),八个 16 bit 的操作操作(如半字整型操作)等,表 1 给出了这两个系统的基本配置。

表 1　实验环境配置

	Intel	AMD Phenom Ⅱ
核数	6	4
核类型	Xeon CPU E5-2620 v2 @2.10 GHz	AMD Phenom Ⅱ ×4 (clocked at 3.00 GHz)

本研究中使用了表 2 和 3 中的测试集,即 SPEC2006 中 10 个浮点测试集[20]和六个 NPB 测试集[21]。这些测试集可以展示本文工具对多种类型的应用程序的并行效果,对于每一个测试集,本文使用允许范围内的最大输入块。下面的报告显示了在相同的配置下,几种不同的优化方式的优化结果对比。

表 2　SPEC2006 测试集

名称	说明
433. milc	Simulations of 3D SU(3) lattic gauge theory
435. gromacs	performing molecular dynamics
436. cactusADM	solving the Einstein evolution equations
444. namd	simulation of large biomolecular systems
447. dealII	object oriented finite element software library
450. soplex	liner programming solver using simplex algorithm
453. povray	ray-tracing: a rendering technique
454. calculix	setting up finite element equations and solves them
470. lbm	lattice Boltzmann method
481. wrf	weather research and forecasting

表 3　NPB 测试集

名称	说明
ua	unstructured adaptive
ft	3D fast Fourier transform(FFT)
bt	block tridiagonal
sp	scalar pentadiagonal
mg	multigrid to solve the 3D Possion PDE
cg	conjugate gradient

6.2　实验结果

第一个结果集展示了每个测试集在 Intel 系统上全局优化、SLP 优化以及与本地执行结果的对比。在图 9 中,x 轴上的测试集是按照全局优化从性能最差到性能最优的结果。本文的全局优化与 SLP 优化对测试集中的三个应用产生了相同的结果,但对于其他应用,全局优化的性能都高于 SLP 优化。而且,当基本块中有更加有效的调度候选和更多潜在超字重用时,本文的全局优化更加优于 SLP 算法优化。由图 9 可知,SLP 优化组与本地执行组对四个应用的优化结果相同。

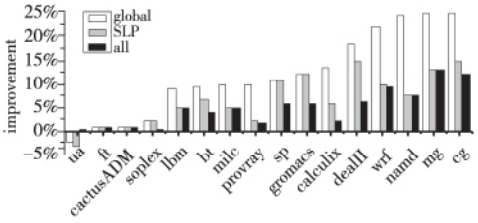

图9　Intel上几种优化的执行时间性能对比

为了解释本文的全局优化与 SLP 优化的不同之处,图 10 中展示了全局优化与 SLP 优化执行时的相关的动态指令开销与打包/解包开销。可以看出,本文的方法可以减少动态指令

和打包/解包操作的比率分别为 14.6% 和 43.7%，可以观察到本文的全局优化对大多数测试集有效。

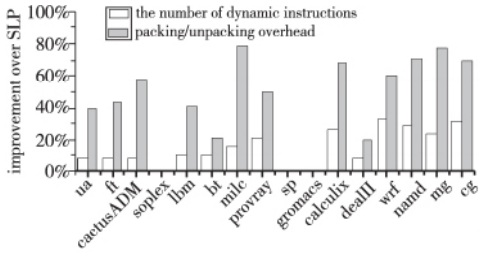

图10 全局优化比SLP优化节省的动态指令与打包/解包开销

图 11 给出了整体优化与标量执行的结果对比。可以观察到数据布局优化对给出测试集中的七个应用(ua、ft、soplex、bt、milc、gromacs、dealII)有额外的性能提升，剩余的测试集受限于前面所提到的特定的约束所以效果不佳。与图 10 中 SLP 算法的结果相比，整体优化的性能超过 SLP 算法的性能大约 15.3%。

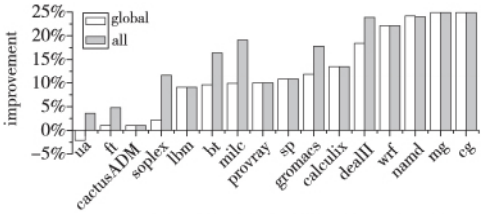

图11 Intel上全局优化与整体优化性能对比

图 12 展现了全局优化、整体优化在 AMD 系统的执行效果。可以看到，全局优化和整体优化分别带来平均 10.8% 和 14.1% 的性能提升，与 Intel 上(全局优化 12.2% 和整体优化 14.9%)结果相似。本文认为，AMD 上的效果之所以低于 Intel 的效果，是因为打包/解包操作的开销的原因。

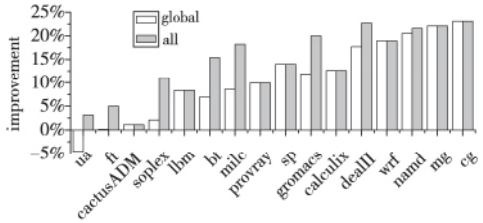

图12 AMD上全局优化与整体优化性能对比

7 结束语

本文的主要贡献是实现自动检测和利用应用程序中的超字级并行的编译框架。本文有三个策略：异构语句同构化、组建超字语句以及数据布局优化，通过提取超字重用来减少超字打包/解包操作，减少内存中数据重组时的强制打包/解包操作的开销以及内存中重排指令的使用，增加 SIMD 级并行。在两个商用系统上的实验结果表明本文的新型框架比单独的 SLP 算法性能更优，与之相比有 15.3% 的性能提升。

参考文献：

[1] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets [J]. ACM SIGPLAN Notices, 2000, 35(5): 145-156.

[2] Porpodas V, Magni A, Jones M T. PSLP: padded SLP automatic vectorization [C]//Proc of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. Washington DC: IEEE Computer Society, 2015: 190-201.

[3] 高伟，赵荣彩，韩林，等. SIMD 自动向量化编译优化概述 [J]. 软件学报，2015，26(6)：1265-1284.

[4] Leißa R, Hack S, Wald I. Extending a C-like language for portable SIMD programming [J]. ACM SIGPLAN Notices, 2012, 47(8): 65-74.

[5] Vasilache N, Meister B, Baskaran M, *et al*. Joint scheduling and layout optimization to enable multi-level vectorization [C]//Proc of International Workshop on Polyhedral Compilation Techniques. 2012.

[6] Kong M, Pouchet N, Sadayappan P. Abstract vector SIMD code generation using the polyhedral model, 4/13-TR08 [R]. Columbus: Ohio State University, 2013.

[7] Ren Bin, Agrawal G, Larus J R, *et al*. Fine-grained parallel traversals of irregular data structures [C]//Proc of the 21st International Conference on Parallel Architectures and Compilation Techniques. New York: ACM Press, 2012: 461-462.

[8] Barik R, Zhao Jisheng, Sarkar V. Efficient selection of vector instructions using dynamic programming [C]//Proc of IEEE/ACM International Symposium on Microarchitecture. 2010: 201-212.

[9] Park Y, Seo S, Park H, *et al*. SIMD defragmenter: efficient ILP realization on data-parallel architectures [J]. ACM SIGARCH Computer Architecture News, 2012, 40(1): 363-374.

[10] Shin J, Chame J, Hall M W. Compiler-controlled caching in superword register files for multimedia extension architectures [C]//Proc of International Conference on Parallel Architectures & Compilation Techniques. Washington DC: IEEE Computer Society, 2002: 45-55.

[11] Shin J, Chame J, Hall M W. Exploiting superword-level locality in multimedia extension architectures [J]. Journal of Instruction Level Parallelism, 2003, 5: 1-28.

[12] Shin J. Compiler optimizations for architectures supporting superword-level parallelism [D]. Los Angeles: University of California, 2005.

[13] Shin J, Hall M, Chame J. Superword-level parallelism in the presence of control flow [C]//Proc of International Symposium on Code Generation and Optimization. 2005: 165-175.

[14] Tenllado C, Piñuel L, Prieto M, *et al*. Pack transposition: enhancing superword level parallelism exploitation [C]//Proc of International Conference on Parallel Computing: Current & Future Issues of High-End Computing. 2008: 573-580.

[15] Tenllado C, Prieto M, Tirado F, *et al*. Improving superword level parallelism support in modern compilers [C]//Proc of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. New York: ACM Press, 2005: 303-308.

[16] Nuzman D, Rosen I, Zaks A. Auto-vectorization of interleaved data for SIMD [J]. ACM SIGPLAN Notices, 2006, 41(6): 132-143.

[17] Nuzman D, Zaks A. Outer-loop vectorization: revisited for short SIMD architectures [C]//Proc of the 17th International Conference on Parallel Architectures & Compilation Techniques. New York: ACM Press, 2008: 2-11.

[18] Liu Jun, Zhang Yuanrui, Jang O, *et al*. A compiler framework for extracting superword level parallelism [C]//Proc of the 33rd Conference on Programming Language Design and Implementation. New York: ACM Press, 2012: 347-357.

[19] Wolfe M. High performance compilers for parallel computing [M]. [S. l.]: Pearson, 1995.

[20] SPEC® 2006 [EB/OL]. [2011-09-07]. http://www.spec.org/cpu2006/.

[21] NAS parallel benchmark [EB/OL]. http://www.nas.nasa.gov/Resources/Software/npb.html.