计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220354
60(12): 2907−2927, 2023

基于多种同构化变换的 SLP 向量化方法

冯竞舸[1] 贺也平[1,2] 陶秋铭[1] 马恒太[1]
[1](基础软件国家工程研究中心(中国科学院软件研究所) 北京 100190)
[2](计算机科学国家重点实验室(中国科学院软件研究所) 北京 100190)
(jingge@iscas.ac.cn)

SLP Vectorization Method Based on Multiple Isomorphic Transformations

Feng Jingge[1], He Yeping[1,2], Tao Qiuming[1], and Ma Hengtai[1]
[1](*National Engineering Research Center for Fundamental Software* (*Institute of Software*, *Chinese Academy of Sciences*), *Beijing* 100190)
[2]((*State Key Laboratory of Computer Science* (*Institute of Software*, *Chinese Academy of Sciences*), *Beijing* 100190))

Abstract SLP (superword level parallelism) is an efficient auto-vectorization method to exploit the data level parallelism for basic block, oriented to SIMD (single instruction multiple data), and SLP has been widely used in the mainstream compilers. SLP performs vectorization by finding multiple sequences of isomorphic instructions in the same basic block. Recently there is a research trend that the compilers translate the sequences of non-isomorphisic instructions into the sequences of isomorphisic instructions to extend application scope of the SLP vectorization method. In this paper, we introduce SLP-M, a novel auto-vectorization method that can effectively vectorize the code containing sequences of non-isomorphic instructions in the same basic block, translatting the code into isomorphic form by selection and conduction of multiple transformation methods based on condition judgment and benefit evaluation. A new transformation method for binary expression replacement is also proposed. SLP-M improves application scope and performance benefit for SLP. We implement SLP-M in LLVM. A set of applications are taken from some benchmarks such as SPEC CPU 2017 to compare our approach and prior techniques. The experiments show that, compared with the existing methods, the performance of SLP-M improves by 21.8% on kernel functions, and improves by 4.1% in the overall tests of the benchmarks.

Key words SIMD extension; auto-vectorization; superword level parallelism (SLP); sequence of non-isomorphism instructions; isomorphic transformation

摘 要 超字级并行(superword level parallelism, SLP)是一种面向处理器单指令多数据(single instruction multiple data, SIMD)扩展部件实现程序自动向量化的方法,这种方法被广泛应用于主流编译器中.SLP 方法有赖于先找到同构指令序列再对之进行自动向量化.将非同构指令序列等价转为同构指令序列以扩展 SLP 方法的适用范围是当前研究趋势之一.提出 SLP 的一种扩展方法——SLP-M 向量化方法,引入二元表达式替换同构转换方式,基于条件判断和收益计算的选择,利用多种指令序列同构化转换,将满足特定条件的非同构指令序列转换为同构指令序列,再进一步实施自动向量化,从而提升 SLP 的适用范围和收益.在 LLVM 中实现了 SLP-M 方法,并利用 SPEC CPU 2017 等标准测试集进行了测试评估.实验结果表明,SLP-M 方法相比于已有方法在核心函数测试中性能提升了 21.8%,在基准测试程序整体测试中性能提升了 4.1%.

收稿日期: 2022-05-23; **修回日期**: 2023-02-09
基金项目: 中国科学院战略性先导科技专项(XDA-Y01-01, XDC02010600)
This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (XDA-Y01-01, XDC02010600).
通信作者: 冯竞舸(jingge815@gmail.com)

关键词 SIMD 扩展; 自动向量化; 超字级并行; 非同构指令序列; 同构化变换

中图法分类号 TP312

自动向量化[1]是利用处理器单指令多数据(single instruction multiple data, SIMD)扩展部件实现程序数据级并行的编译优化方法,它与手工编写 SIMD 向量程序相比,不需要程序员深入理解 SIMD 扩展部件的功能特性,减少了程序员的负担. 随着 SIMD 硬件扩展指令集的不断发展,向量指令应用的领域和需求也越来越多,然而如何生成高效的向量指令至今依然是个挑战[2-3].

目前自动向量化主要有 2 大类方法: 基于循环的自动向量化方法[4]和超字级并行(superword level parallelism, SLP)自动向量化方法[5],其分别旨在实现循环和基本块的自动向量化.SLP 一直是向量化领域关注的重要方法,本文重点对其展开研究.

SLP 方法通过寻找同构指令序列[5],将数据合并到同一向量寄存器中并行处理. 由于同构指令序列在实际程序中占比不是很高以及方法自身的原因[6], SLP 的适用范围受限. 近年来,有研究者开始关注于通过等价变换将满足特定条件的非同构指令序列转换为同构指令序列,从而为实施 SLP 创造条件. 如 PSLP 方法[7]采用基于程序差异特征的图模式匹配,通过添加选择(select)指令进行扩充,从而将非同构指令序列转换成同构指令序列.LSLP 方法[8]利用交换律等价关系式对非同构序列中的运算操作和操作数进行重排,从而获得同构指令序列. 类似地, SN-SLP 方法[9]和 SLP-E 方法[10]分别采用减法/除法的等价关系式和扩展等价关系式对非同构序列中的运算操作和操作数进行变换,从而获得同构指令序列.

除了采用的同构化转换方法外,我们发现实际还存在其他方法可应用于指令序列同构化转换中. 例如利用二元表达式等价替换关系(如 $X\times4=X<<2$)将非同构指令序列转换为同构指令序列.

在对非同构指令序列进行 SLP 向量化时,充分利用多种同构化转换方法比只考虑单一转换方法更有优势. 例如在 SPEC CPU 基准测试程序中存在一些包含多条二元操作且存在常量操作数的语句,需要经过多种方法的同构化转换后才可对其有效实施自动向量化. 然而,目前还无文献专门针对同时利用多种同构化转换的向量化方法进行研究.

与只考虑单一同构化转换不同,当考虑多种同构化转换方法时,需解决同构化转换方法的选择问题. 对于任一非同构指令序列,首先需要分析确认哪些同构化转换方法适用; 其次,对于同一指令序列不同同构化转换方法产生的代价是不同的,需要选择合适的同构化转换方法,提升程序的自动向量化性能收益. 因此指令序列同构化转换方法的选择涉及 2 个问题: 问题 1: 一些指令序列存在不同的同构化转换方法,需进行适用同构化转换方法的识别和搜索. 问题 2: 不同同构化转换方法产生的收益不同,需要针对各种同构化转换方法给出收益评估方法.

本文提出一种 SLP 扩展方法——SLP-M 向量化方法,将多种方法应用于非同构指令序列的同构化转换中,并评估每种同构化转换方法的性能收益,根据收益进行同构化转换方法的选择. 为了解决同构化转换方法选择的第 1 个问题,本文利用处理程序的特性和同构化转换方法的特点进行分析,采用启发式方法,优先搜索那些需转换指令数量较少以及引入自动向量化的性能收益相对较高的同构化转换方法. 为了解决同构化转换方法选择的第 2 个问题,本文发现指令的操作类型相似程度越大,那么这些语句越容易被实施自动向量化,越有机会带来自动向量化的性能收益,因此利用程序操作及操作数的类型相似程度并结合同构化转换方法的特征进行收益评估.

SLP-M 方法与先前工作的主要区别在于:

1)SLP-M 综合利用多种方法(包括二元表达式等价替换、扩展变换、基于 shuffle 指令的变换等)进行同构化转换,而先前的 PSLP 和 LSLP 等方法都只采用了单一类型的同构化转换方法.SLP-M 与先前工作比较,提高了将非同构指令序列转换为同构指令序列的能力. 其中,基于二元表达式替换的指令序列同构化转换方法与 PSLP 比较,对于特定类型程序,不需要生成引入额外运行代价的指令就可进行同构化转换. 基于二元表达式的替换变换与 LSLP 和 SN-SLP 进行比较,能够同构化转换一些先前方法不能同构化转换的指令序列.

2)由于 SLP-M 综合采用多种同构化转换方法,为了选择合适的同构化转换方式,SLP-M 方法解决了采用多种方法进行同构化转换过程中的一些特殊问题,如多种同构化转换方法的识别、搜索和选择等,这些是 PSLP 和 LSLP 等工作未涉及的.

本文基于 LLVMv10.0 实现了 SLP-M 方法,并基于 SPEC CPU 2017 等测试集进行了测试和评估. 实验

结果表明，本方法相比已有方法在核心函数测试中性能提升了 21.8%，在基准测试程序整体性能测试中性能提升了 4.1%.

本文的主要贡献包括 3 个方面：

1）提出基于多种指令序列同构化转换方法的 SLP-M 向量化方法，扩展了对非同构指令序列的自动向量化适用范围；

2）除了已有指令序列同构化转换方法，本文还引入和利用了基于二元表达式替换的方法，该方法不仅不会引入额外运行代价，而且减少了指令的操作类型和数量；

3）提出指令序列同构化转换的选择方法，发挥了多种同构化转换方法的优势，提升了对非同构指令序列自动向量化的性能收益.

1 背景与研究动机

目前大部分关于自动向量化的研究主要是针对某类特殊类型的程序向量化，将某个具体的程序变换方法与自动向量化结合，或利用某个特殊 SIMD 扩展指令优化自动向量化等方法，这些研究往往对特定类型程序是有效的，但是利用单一自动向量化方法变换后得到的结果并不总是有效. 例如循环，既可包含循环内的并行语句又可包含循环间的并行语句，如果单独利用基于循环或基本块内的自动向量化方法，得到的向量化方案不总是最优的.

超字级并行[5]是对基本块内同构指令序列进行自动向量化的方法，可有效提升程序的性能，得到许多研究者的关注和进一步研究，相关成果集中于基于同构指令序列构建同构链[7]的研究，具体包含 2 方面内容：1）扩展 SLP 向量化的适用范围，如旨在对循环[11-15]和包含分支程序[16]的向量化，以及多级融合（跨循环和跨基本块融合）[17]的向量化.2）提高向量化的性能收益，利用全局策略或局部贪心策略构建同构链，提高生成向量指令收益，减少重组指令的代价. 全局策略构建同构链方法是基于全局搜索的方式构建同构链[18-21]. 局部贪心策略构建同构链方法是基于局部贪心策略逐层构建同构链[22-24]. 此外，除了应用于自动向量化，SLP 方法也被扩展应用于动态二进制翻译[25]、内嵌汇编形式向量代码优化[26]和非 SIMD 向量指令优化[27]等领域.

SLP 方法有赖于先找到程序中的同构指令序列，这对 SLP 方法的实际应用造成一定局限. 近年来，有学者[7]开始关注如何将 SLP 方法扩展应用到非同构指令序列如 PSLP 和 LSLP 等，PSLP 和 LSLP 等方法将特定的非同构指令序列转换为同构指令序列，然后再进一步实施向量化. 然而，PSLP 和 LSLP 等方法都存在其特定的适用范围，如果单一实施，向量化能力有限.

图 1 描绘了 SPEC CPU 2017 625case 中的一段程序代码采用几种不同的自动向量化方法进行处理的过程. 其中图 1(a)(b)分别为一段包含非同构指令序列的输入程序及其对应的数据依赖图.图 1(c)(e)(g)(i)是将多条标量指令转换为向量形式的分组图，其中，阴影长方形表示可自动向量化的操作或操作数；虚线框的长方形表示自动向量化所需生成的存在额外运行代价的指令. 向量分组图中的 *benefit* 表示自动向量化的收益，即标量指令代价与向量指令代价的差值，若 *benefit*>0，则自动向量化有收益，编译器实施自动向量化；否则自动向量化无收益，编译器不进行程序转换.图 1(d)(f)(h)表示不同自动向量化方法处理输入程序的转换过程.

SLP 方法的向量分组如图 1(c)所示.SLP 方法先以 $A[0]$, $A[1]$, $A[2]$, $A[3]$ 连续内存访问为种子，并基于种子扩展同构指令序列，扩展中发现访存、移位、乘法操作，其类型不同，因而停止同构指令序列的进一步扩展，编译器评估自动向量化无收益，因此不对输入程序进行转换.

PSLP 方法对输入程序的处理流程如图 1(d)(e)所示，代码中 4 条语句对应数据依赖图的差异较大，PSLP 方法在自动向量化中生成较多额外运行代价的选择指令，编译器评估自动向量化无收益，因此不对输入程序进行程序转换.

LSLP 方法只能对可交换运算操作的操作数进行重排序，由于处理中的访存、移位、乘法操作不都是可交换运算操作，因此无法实施同构化转换.SN-SLP 方法与 LSLP 方法类似，无法对输入程序实施自动向量化.

对于图 1(a)中的原始输入程序，其实存在的多种适用的同构化转换方法至少有 2 种：

1）通过扩展转换关系将非同构指令序列转换为同构指令序列. 如图 1(f)(g)所示，先将第 1 行语句的 $B[0]$ 扩展变换为 $B[0]<<0$，然后将第 3 行的 $B[2]\times3$ 扩展变换为 $(B[2]\times3)<<0$，这样程序转为 $A[0]=B[0]<<0$；$A[1]=B[1]<<1$；$A[2]=(B[2]\times3)<<0$；$A[3]=B[3]<<2$，最后对第 1，2，4 行语句扩展乘法操作，将程序转为同构形式. 从性能收益上考虑，由于将输入程序中的 2 个标量移位和 1 个标量乘法替换为 1 个向量移位和 1

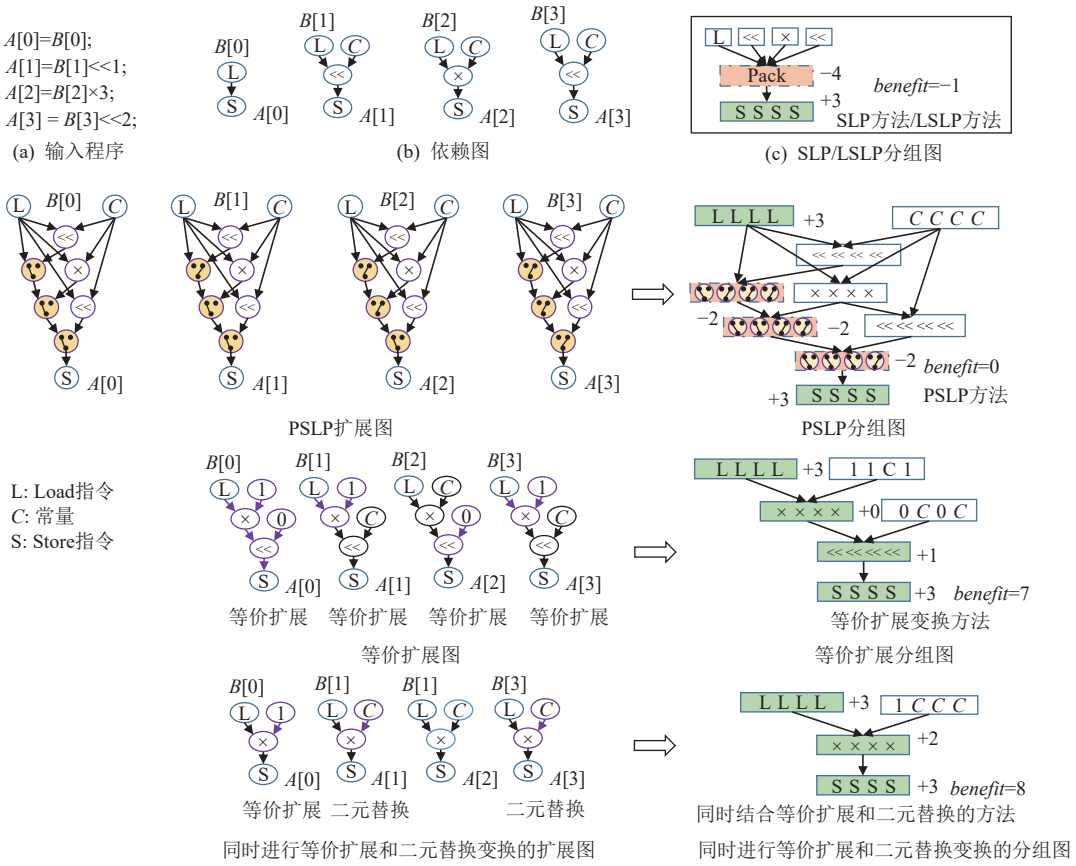


Fig. 1 Process of various auto-vectorization methods adopted by an example program

图 1 一个示例程序采用不同自动向量化方法进行处理的过程

个向量乘法，对于移位和乘法而言自动向量化的收益为 1，再加上将输入程序的 8 个标量访存转换为 2 个向量访存，自动向量化总体收益是 7，可有效实施自动向量化.

2)同时采用扩展转换关系和二元表达式等价替换关系将非同构指令序列转换为同构指令序列. 如图 1(h)(i)所示，首先将第 1 行语句的 $B[0]$ 扩展变换为 $B[0]\times1$，然后将第 2 行语句的 $B[1]<<1$ 替换为 $B[1]\times2$，最后同理对第 4 行语句进行二元表达式替换，使非同构指令序列转换为同构指令序列，与上文单独采用扩展变换比较，减少了向量移位操作，此时自动向量化可带来收益($benefit=8$)，因此，进一步提升了自动向量化的性能收益.

从上面 2 种转换示例可以看出：1)当引入多种同构化转换方法的时候，确实能够获得更多的向量化机会和性能收益；2)选择不同同构化转换方法会影响向量化的性能收益. 有鉴于此，本文希望研究和实现能够有效利用多种指令序列同构化转换方式的向量化方法.

2 指令序列同构化转换方法

目前已有的指令序列同构化转换方法不是很多. PSLP 方法可以被认为是第 1 个对非同构指令序列进行转换研究的工作，它利用硬件选择指令来实施同构化转换. 此后，有研究者注意到可以基于表达式等价关系实现同构化转换. 例如 LSLP 方法利用交换律进行同构化转换，SN-SLP 方法利用减法以及除法等价关系式进行同构化转换，SLP-M 利用等价扩展变换进行同构化转换. 除了上述涉及操作符/操作数顺序和个数变化的表达式等价变换外，我们发现实际还存在多种其他涉及操作符类型变化的等价转换关系可应用于同构化转换中，如 $X<<2=X\times4$，$X\times2=X+X$ 等，也就是可将“形式不同但本质功能相同”的操作符等价转为类型相同的操作符.

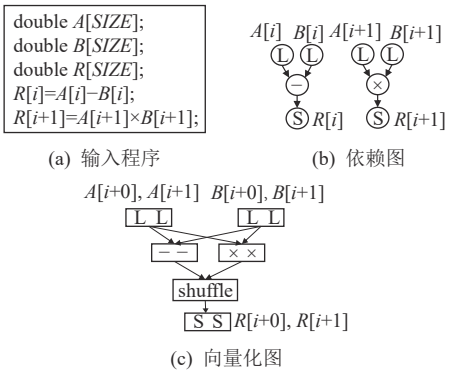
结合目前已有方法以及我们自己的分析归纳，本文采用了 4 种指令序列同构化转换方法：基于 shuffle 指令的同构化转换、重排序变换、扩展变换、二元表达式替换. 下面逐一介绍.

2.1 基于 shuffle 指令的同构化转换

shuffle 是将多个数据合并、重组和排序的指令，通常包含 2 个输入操作数和 1 个输出操作数，输出操作数是输入操作数重组后的数据. 绝大部分处理器都支持类似 shuffle 功能的指令，如 x86 和 ARM 系列处理器等.

基于 shuffle 指令的同构化转换方法是指利用 shuffle 指令将不同的操作合并到同一向量寄存器，并基于它们相同位置的操作数，向“定义—使用”链和“使用—定义”链的方向继续建立扩展同构指令序列的程序变换方式. 基于 shuffle 指令的同构化转换与 PSLP 方法中采用选择指令的转换方法本质相同，都是将不同类型的操作通过特殊指令合并到同一向量寄存器中. 选择指令的向量形式是基于 shuffle 指令实现的.

具体示例如图 2 所示，其中图 2(a)(b)分别是输入程序以及对应的依赖图，图 2(c)表示利用 shuffle 指令将减法和乘法操作合并到同一向量寄存器中.



(a) 输入程序 (b) 依赖图

(c) 向量化图

Fig. 2 An example of program transformation based on shuffle instruction

图 2 基于 shuffle 指令的程序转换示例

2.2 重排序变换

重排序变换指基于交换律、减法性质[9]或者除法性质[9]的等价变换，通过重排序操作符或操作数获得同构指令序列，涉及操作符或操作数顺序变化，不涉及其个数和类型的改变.

基于交换律的重排序等价变换式如：$A⊕B==B⊕A$，其中⊕是可交换运算，A 和 B 是⊕的操作数. 满足交换律的运算普遍存在于实际程序中，包括加法、乘法、与运算、或运算等. 重排序可交换运算的操作数，如 LSLP 方法，使得多个可交换运算的操作数所对应的指令组成同构指令序列，如将 $c[0]=a[0]+b[0]$ 和 $c[1]=b[1]+a[1]$ 的第 2 条语句变换为 $c[1]=a[1]+b[1]$. 则这 2 条语句加法的第 1 个和第 2 个操作数分别是 $a[0]$, $a[1]$, $b[0]$, $b[1]$, 它们是连续的内存访问，可实施向量化.

SN-SLP 所采用的减法性质是只由减法或加法所组成的表达式，交换其减法以及对应操作数的顺序，输出值不变如 $a-b+c=a+c-b$，所采用的除法性质类似.SN-SLP 方法采用运算符性质的重排序变换，使得多个减法或者除法运算的操作数组成同构指令序列，如 $a[0]=b[0]-c[0]-d[0]$ 和 $a[1]=b[1]-d[1]-c[1]$ 可通过减法性质将 $a[1]=b[1]-d[1]-c[1]$ 变换为 $a[1]=b[1]-c[1]-d[1]$，则这 2 条语句减法对应的操作数分别为 $a[0]$, $a[1]$; $b[0]$, $b[1]$; $c[0]$, $c[1]$, 它们是连续的内存访问，可实施向量化.

2.3 扩展变换

扩展变换是通过对表达式进行等价的扩展变换[10]，将程序中非同构指令序列转换为同构指令序列，涉及操作符和操作数的个数改变.

首先介绍扩展等价表达式. 如果一个二元运算⊕、常数 C(C 为 0 或 1)，以及 X(X 可为变量、常量或者表达式)满足 $X⊕C==X$ 或 $C⊕X==X$，那么本文把 $X⊕C$ 或 $C⊕X$ 称为 X 的扩展等价表达式. 扩展等价表达式存在多种类型，表 1 给出了本文已实现的扩展等价表达式.

Table 1 Equivalent Extended Expression of X

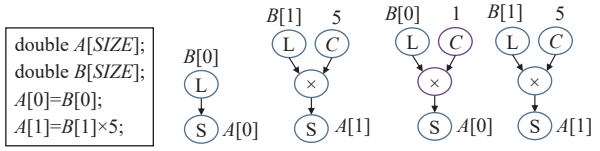
表 1 X 的扩展等价表达式

序号	X 的扩展等价表达式	依据的等价关系
1	$X+0$	$X+0=X$
2	$0+X$	$0+X=X$
3	$X-0$	$X-0=X$
4	$X\times1$	$X\times1=X$
5	$1\times X$	$1\times X=X$
6	$X/1$	$X/1=X$
7	$X<<0$	$X<<0=X$
8	$X>>0$	$X>>0=X$

扩展变换就是将表达式 X 转换为 X 的扩展等价表达式，使其扩展表达式的操作在特定条件可与其他的操作组成同构指令序列. 扩展变换示例如图 3 所示，其中图 3(a)(b)分别是输入程序以及对应的依赖图.图 3(c)表示通过扩展变换方法将第 1 行语句的 $B[0]$ 扩展为 $B[0]\times1$，扩展变换后获得同构语句.

2.4 二元表达式替换

二元表达式替换是本文新引入的一种等价变换. 在介绍该变换前，先引入二元替换等价表达式的概



(a) 输入程序 (b) 依赖图 (c) 扩展变换图

Fig. 3 An example of extended transform

图 3 扩展变换示例

念. 对于表达式 1：XⓉY, 表达式 2：ZⓂK, 若同时满足 3 个条件则表达式 2 被称为表达式 1 的二元替换等价表达式：

1)表达式 1 和表达式 2 中的Ⓣ和Ⓜ都是二元操作符, 且操作符类型不同；

2)表达式 1 和表达式 2 至少存在 1 个相同的操作数；

3)表达式 1 中Ⓣ可被直接替换为Ⓜ(不涉及操作符个数的变化), 且表达式 1 和表达式 2 输出值相同.

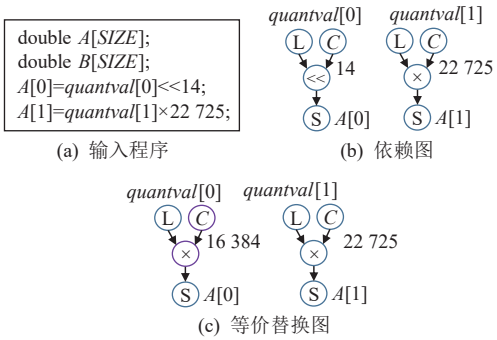
二元表达式替换是指将程序中特定的二元表达式转换为其二元替换等价表达式, 进而将程序中的非同构指令序列转换为同构指令序列. 二元替换等价表达式在程序中存在多种类型, 本文已实现的表达式类型如表 2 所示. 二元表达式替换不同于重排序变换和扩展变换, 不涉及操作符顺序和个数的改变, 而涉及操作符类型的改变.

Table 2 List of Patterns of Replacement for Binary Expressions

表 2 二元表达式替换模式列表

原表达式	二元替换等价表达式	二元表达式替换的条件
$X+C_1$	$X-C_2$	C_1 和 C_2 是常数，且 $C_1+C_2=0$
$X+X$	$X\times 2$	
$X-C_1$	$X+C_2$	C_1 和 C_2 是常数，且 $C_1+C_2=0$
$X\times C_1$	$X<<C_2$	C_1 和 C_2 是常数，C_1 是 2 的整数次幂，且 $C_1=2^{C_2}$
$X\times C_1$	$X+X$	C_1 是常数 2
$X\times C_1$	X/C_2	C_1 和 C_2 是常数，且 $C_1\times C_2=1$
X/C_1	$X\times C_2$	C_1 和 C_2 是常数，$C_1\times C_2=1$
$X>>C_1$	$X\times C_2$	C_1 和 C_2 是常数，且 $C_2=2^{C_1}$

二元表达式替换示例如图 4 所示, 其中图 4(a)(b)分别是输入程序以及对应的依赖图.图 4(c)表示通过二元表达式替换的方法将第 1 行语句的移位操作替换为乘法操作, 进而将其转换为同构的形式.



(a) 输入程序 (b) 依赖图 (c) 等价替换图

Fig. 4 An example of replacement of binary expression

图 4 二元表达式替换示例

3 SLP-M 向量化方法

3.1 方法概述

SLP-M 是一种改进的 SLP 向量化方法.SLP 的基本处理过程是基于“种子”并根据“定义-使用(def-use)”链和“使用-定义(use-def)”链扩展同构图, 然后结合同构图分析向量及标量形式代码的代价, 若向量代码代价相对较低, 则实施自动向量化.SLP 方法不考虑对非同构指令序列的向量化, 而 SLP-M 对 SLP 的改进就在于对非同构指令序列的分析和处理, SLP-M 会尝试对非同构指令序列进行同构化转换, 并在条件满足时对非同构指令序列实施自动向量化.

SLP-M 引入和运用了多种同构化转换方法, 包括基于 shuffle 指令的转换、重排序变换、扩展变换、二元表达式替换. 如引言所述, 当考虑多种同构化转换方法时候, 对于同一指令序列有时存在多种适用的同构化转换方法, 需解决同构化转换方法的选择问题. 同时, 对于同一指令序列不同的同构化转换方法产生的代价不同, 需要选择合适的同构化转换方法, 提升程序的自动向量化性能收益. 该问题主要涉及 2 个问题.

问题 1. 同构化转换方法的识别和搜索.

有些指令序列存在大量适用的同构化转换方法, 例如 shuffle 指令理论上可把任意的操作存放到同一向量寄存器中, 这就引入了很多转换方法. 若采用穷举法遍历每种适用的同构化转换方法, 那么方法数量成指数级增长, 计算复杂度太高. 因此需要寻找和设计高效的同构化转换方法的识别和搜索方法.

问题 2. 针对特定同构化转换方法的收益评估.

不同同构化转换方法不仅影响目标处理指令序列, 而且影响其操作数, 进而影响自动向量化的整体性能收益, 需针对每种方法进行整体性能收益评估.

对于问题 1，本文结合处理程序及同构化转换方法的特点，给出了一种启发式方法用以识别和搜索更有机会带来性能收益的转换方式. 同构化转换的目的是以选择 1 个指令作为基准参照指令，通过等价转换使指令序列中其他所有指令与其操作类型相同. 原则上指令序列中的每个指令都可选作基准参照指令. 但如果启发式将指令序列中的每个指令都尝试作为基准参照指令，遍历所有适用的同构化转换方式，计算复杂度太大. 其实，许多非同构指令序列是可以被自动向量化的，这些指令序列中大部分指令的操作类型是相似的. 若将指令序列中的与其他指令相似程度最高的指令作为基准参照指令（本文中被称为基准指令，参见 3.2 节），那么以基准指令作为参照的同构化转换的候选集合大概率会包含适用于这类指令序列的同构化转换方式. 鉴于此，SLP-M 以基准指令作为参照，然后结合不同同构化转换方法的具体特点，如转换方式对操作的处理以及引入自动向量化的额外运行代价等，启发式识别和搜索更有机会带来自动向量化性能收益的具体变换方式.

对于问题 2，需要评估指令序列在不同方法同构化转换下自动向量化的整体性能收益，SLP-M 利用程序对应依赖图中的操作类型相似信息评估自动向量化的性能收益，从目标处理指令对应依赖图中的节点开始，并递归沿着其子孙节点的方向，分析指令操作及其操作数的同构化转换方法，评估并统计整体的性能收益.

本文利用多种方法来进行同构化转换，利用基准指令的选择方法来启发式识别和选择合理的同构化转换方法，降低选择的代价，通过同构化转换方法的搜索并利用评估模型来评估适合的方法，由此得到自动向量化性能收益较高的具体变换方式. 而 PSLP 和 LSLP 等采用单一类型的同构化转换方法，不存在多种同构化转换方法的搜索和选择问题，因此未用到基准指令的选择和同构化转换的搜索方法.

SLP-M 的方法处理流程图如图 5 所示，其中同构化分析及转换包括 4 个具体步骤：1）基准指令的选择. 根据程序对应依赖图中操作节点及子孙节点的信息选择基准指令. 2）同构化转换方法的搜索. 针对特定的指令序列搜索适用的同构化转换方法. 3）同构化转换方法的选择. 评估收益并选择其中评估性能收益最高的同构化转换方法. 4）同构化转换. 下面对这 4 个步骤分别进行介绍.

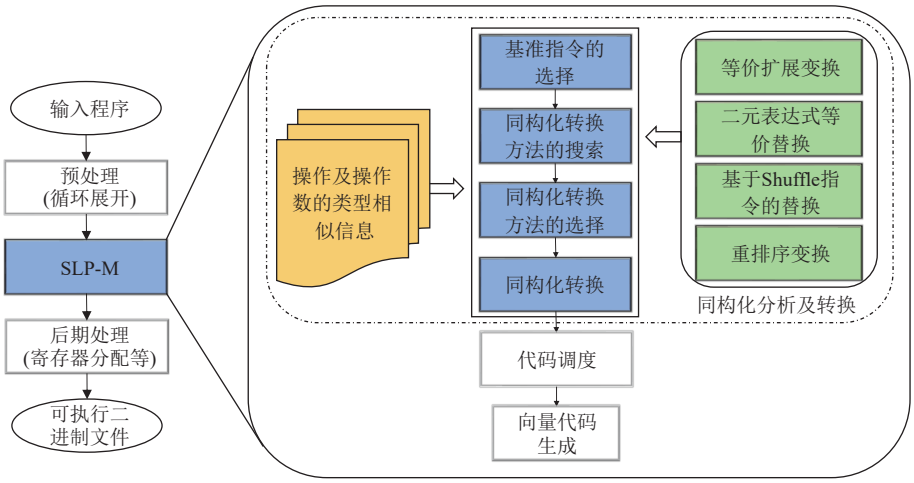


Fig. 5 Process flow of SLP-M

图 5 SLP-M 的处理流程

3.2 基准指令的选择

为了选择基准指令，SLP-M 遍历指令序列中的每个指令，计算该指令与其他指令对应依赖图中的操作类型相同节点（下文称为“匹配节点”）的个数，或者可通过二元表达式替换转换为匹配节点的个数.SLP-M 将与指令序列中其他指令的匹配节点个数最多的指令作为基准指令.SLP-M 选择基准指令的步骤有 2 个.

1）遍历处理指令序列 *insts* 中的每个指令（记为指令 *x_inst*），计算指令 *x_inst* 与指令序列中其他指令（记为指令 *y_inst*）的对应依赖图中的匹配节点或者可通过二元表达式替换转换为匹配节点总数. 计算匹配节点个数的方法是：从指令 *x_inst* 和指令 *y_inst* 对应依赖图的节点开始自底向上统计匹配节点个数.

①若指令 *x_inst* 和指令 *y_inst* 的操作类型相同，统计计数加 1，继续向它们的子节点方向统计计算；

②若指令 *x_inst* 和指令 *y_inst* 的操作类型不同，但可通过二元等价替换的方式将指令 *y_inst* 转换为与指令 *x_inst* 相同类型的操作，实施二元等价替换，并统计计数加 1，继续向它们的子节点方向统计计算；

③若指令 *x_inst* 和指令 *y_inst* 的操作类型不同，且无法通过二元表达式替换转换为与其相同的操作，或者遇到叶子节点，则停止向它们的子节点方向统计计算.

2)根据步骤 1 计算的匹配节点个数，选择指令序列 *insts* 中与其他指令对应匹配个数最多的指令为基准指令. 当多个指令与其他指令的匹配节点个数相同时，可启发式选择排在更前面的指令.

3.3 同构化转换方法的搜索

在求得基准指令后，SLP-M 以基准指令作为参照，搜索将指令序列中的指令转换为与基准指令类型相同操作的转换方式. 本文根据特定方法的转换特征，如转换方式对操作的处理以及引入自动向量化的额外运行代价，搜索每种同构化转换方法具体的转换方式.

基于 shuffle 指令的程序转换会生成 shuffle 指令，引入额外运行代价.SLP-M 排除引入较多额外运行代价转换的具体变换方式，仅考虑对于指令序列包含 2 类二元操作的条件下将基于 shuffle 指令的同构化转换纳入候选同构化的转换方式.

重排序变换通过重排序指令的操作以及操作数使得在特定条件下增多自动向量化语句的数量[8-9]不会引入额外代价. 因此，若在指令序列中存在与基准指令操作类型相同的可交换运算操作，SLP-M 会将对其的重排序变换纳入候选同构化转换方式.

扩展变换将指令转换为其的扩展等价表达式，使得扩展后的指令与原始程序的指令组成同构指令序列，尽管在程序扩展转换中增加了运算指令，但是这些指令可与原程序中标量代码一同转为向量形式，不会引入自动向量化的额外运行代价. 因此，若指令存在与基准指令操作类型相同的等价扩展式，SLP-M 会将对其的扩展变换纳入候选同构化转换方式.

二元表达式替换变换是在特定的条件下将一个表达式替换为另一个表达式，使替换后指令与原始程序的指令组成同构指令序列，尽管存在会将代价较低的标量指令转为代价较高的标量指令的情形，但是转换生成的指令可与原始程序一同转为向量形式，因此向量化不会引入额外运行代价，且由于替换变换减少了源程序标量操作的种类，因此向量化后进一步减少生成指令的种类和数量. 若指令存在与基准指令操作类型相同的二元表达式替换式，SLP-M 会将对其的二元表达式替换变换纳入候选同构化转换方式.

为了表示指令序列的同构化转换方式，本文沿用文献 [18] 中指令对和转换模式的术语，并结合本文要解决的同构化转换问题进一步明确其含义：1)指令对. 指由 2 个指令组成的集合.2)转换模式. 指对于特定的指令对，描述其操作、操作数、具体的同构化转换方式的集合，包括操作指令集合 *v*、同构化转换方式 *action* 和同构化转换后的操作集合 *v′* 这 3 个部分，可表示为(*v*, *action*, *v′*)三元组形式，转换模式示例如图 6 所示. 1 个指令序列可由 1 个或者多个指令对组成，其同构转换方式可描述为其中多个指令对的转换模式集合.

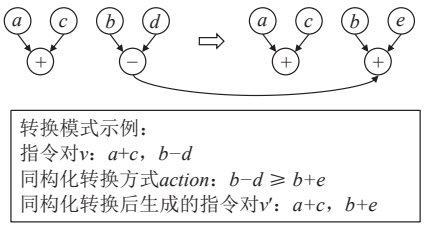


Fig. 6 An example of conversion pattern

图 6 转换模式示例

对于指令序列 *insts*，SLP-M 首先搜索满足 2 个条件的指令对集合 *inst_pairs*：1)指令对中的指令都属于指令序列 *insts*；2)指令对中必须包含 *insts* 的基准指令. 然后，针对 *inst_pairs* 中的每个指令对，在第 2.1~2.3 节描述的同构化转换的可转换条件所述满足特定条件的候选同构转换方式中遍历所有可将其中的非基准指令转换为与基准指令相同类型操作的转换方式，将其记录于转换模式集合 *patterns* 中.

3.4 同构化转换方法的选择及实施

在搜索完所有具体适用的同构化转换方法后，需评估每种方法自动向量化的性能收益，根据收益进行同构化转换方法的选择. 考虑到评估方法的效率和准确度，同构化转换方法的选择以及实施向量化的评估分步处理，本文设计相对高效的基于操作符和操作数类型的相似信息评估模型进行同构化转换方法的选择，待确定具体的同构化转换方式后，利用准确度相对较高的 LLVM 自带的评估模型判定是否实施向量化，若评估有收益则实施向量化，否则程序以原标量形式执行. 为了评估自动向量化的性能收益，本文介绍 2 个基本概念.

1)单层指令对的自动向量化收益. 指对于1个指令对, 对其进行自动向量化获得性能收益. 这里给出单层指令对的自动向量化收益评估方法, 具体描述为:

①若指令对的2条指令的操作类型相同, 则单层指令对的收益计为1;

②若指令对的2条指令经过二元表达式替换或者重排序变换后, 其操作的类型相同, 则单层指令对的收益计为1;

③若指令对的2条指令需经过扩展变换后, 其操作的类型相同, 则单层指令对的收益计为0;

④若指令对2条指令的操作类型不同, 且无法通过同构化转换方法将其转为操作类型相同的形式, 则单层指令对的收益计为0.

2)整体指令对的自动向量化收益. 指对于1个指令对, 若对其进行自动向量化, 其本身指令对以及子操作数所依赖所有指令对自动向量化性能收益的总和. 这里给出评估整体指令对自动向量化收益的方法, 即整体指令对的自动向量化收益是单层指令对的自动向量化收益与其操作数所组成指令对的整体指令对自动向量化收益之和. 整体指令对的自动向量化收益计算如式(1)所示, 其中 x 和 y 都是指令, $x.operands(k)$ 和 $y.operands(k)$ 分别是指令 x 和指令 y 的第 k 个操作数, $score_{\text{whole}}$ 是整体指令对 $\langle x, y\rangle$ 的自动向量化收益, $score_{\text{single}}$ 是单层指令对 $\langle x, y\rangle$ 的自动向量化收益.

$$score_{\text{whole}}\langle x, y\rangle = score_{\text{single}}\langle x, y\rangle + \sum_{k=0}^{n} score_{\text{whole}}\langle x.operands(k), y.operands(k)\rangle . \quad (1)$$

SLP-M通过评估每个指令对在特定转换模式下的整体指令, 对自动向量化收益选择同构化转换方法, 同构化转换选择函数 *pattern_selection* 伪代码如函数1所示.

函数 1. *pattern_selection*.

输入: 指令序列 *insts*;

输出: 匹配分数 *score*, 相对最优的同构化转换方法 *pattern*.

① *base_inst* = *get_base_inst*(*insts*); /*求得基准指令*/
/*遍历指令序列中每条指令*/
② for(each *inst* in *insts*)
③ *patterns* = *search_pattern*(*inst*, *base_inst*); /*遍历搜索指令适用的转换模式, 将其存储到集合 *patterns* 中*/
④ if(*patterns* == NULL)then
⑤ return;
⑥ end if
⑦ if (*inst* == *base_inst*)then
⑧ continue;
⑨ end if
⑩ *best_score* = 0; /**best_score* 用于记录最优转换模式的分数*/
⑪ *best_pattern* = NULL; /**best_pattern* 用于记录最优的转换模式*/
/*遍历所有的转换模式*/
⑫ for each *pattern* in *patterns*
⑬ *out_inst* = *patterns.out_inst*(); /**out_inst* 用于存储特定转换模式转换后输出的指令*/
⑭ *score* = *init_score*(*patterns*); /*初始化特定转换模式的分数*/
⑮ *score* = *score* + *cal_match_score*(*base_inst*, *inst*); /*计算特定转换模式的分数*/
⑯ if(*score* > *best_score*)then
⑰ *best_score* = *score*;
⑱ *best_pattern* = *pattern*;
⑲ end if
⑳ end for
㉑ *insts_score* += *best_score*;
㉒ *insts_pattern*[*inst*] = *best_pattern*; /**insts_pattern* 用于存储指令序列的最优转换模式*/
㉓ end for
㉔ return(*insts_score*, *insts_pattern*).

为了计算整体指令对自动向量化的收益, 需要计算指令对的单层自动向量化收益以及其子操作数所组成指令对的整体指令对的自动向量化收益.SLP-M的目标是找到使得整体指令对自动向量化收益相对最大的同构化转换方式.SLP-M利用每种方法转换后的指令序列的依赖图, 沿着对应的节点自底向上递归统计和评估, 先评估子操作数对应指令对在每种转换方法下的整体收益, 并选择收益最大的同构化转换方式, 再进行指令对的整体自动向量化的收益评估和同构化转换方法的选择. 具体整体指令对自动向量化收益评估函数的伪代码详见函数2和函数3.

函数 2. *cal_match_score*.
输入：指令 *base_inst*，指令 *inst*；
输出：匹配分数 *match_score*.
① *best_score* = 0;
/*若可进行扩展变换，则进行分数评估*/
② if(*can_extend_trans*(*get_opcode*(*base_inst*), *get_opcode*(*inst*)))then
③ *new_inst* = *get_extended_expression*(*base_inst*, *inst*); /*尝试进行扩展变换分析*/
④ *extend_score* = *cal_match_score*_1(*base_inst*, *new_inst*); /*递归计算指令的操作数分数*/
⑤ if(*extended_score* > *best_score*)then
⑥ *best_score* = *extended_score*;
⑦ end if
⑧ end if
/*若可进行二元表达式替换或重排序变换，则进行分数评估*/
⑨ if(*can_binary_trans*(*get_opcode*(*base_inst*), *get_opcode*(*inst*)))then
⑩ *new_inst* = *get_bin_trans_expression*(*base_inst*, *inst*); /*尝试进行二元表达式替换或重排序变换分析*/
⑪ *bin_trans_score* = 1 + *cal_match_score*_1(*base_inst*, *new_inst*); /*递归计算指令的操作数分数*/
⑫ if(*bin_trans_score* > *best_score*)then
⑬ *best_score* = *bin_trans_score*;
⑭ end if
⑮ end if
/*若原始指令可进行向量化*/
⑯ if(*can_vectorize*(*base_inst*, *inst*))then
⑰ *original_score* = 1 + *cal_match_score*_1(*base_inst*, *inst*);
⑱ if(*original_score* > *best_score*)then
⑲ *best_score* = *original_score*;
⑳ end if
㉑ end if
㉒ return *best_score*.

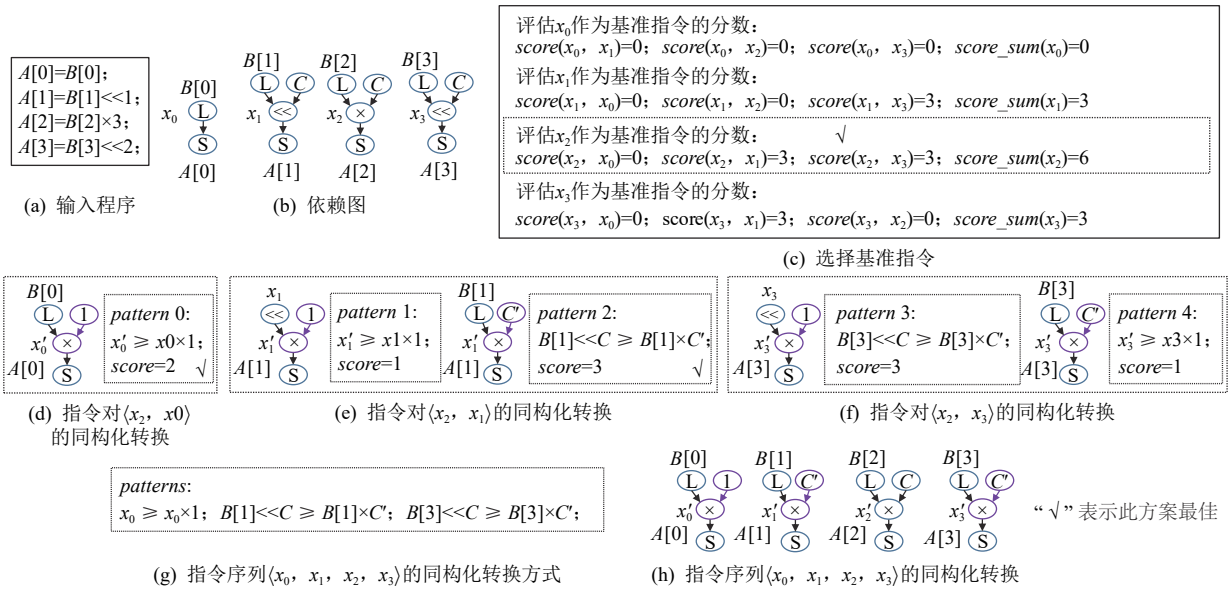
函数 3. *cal_match_score*_1.
输入：指令 *base_inst*，指令 *inst*；
输出：匹配分数 *score*.
① *score* = 0;
② for each *operand_idx* of *base_inst*
③ *base_operand* = *get_operand*(*base_inst*, *operand_idx*); /*求得基准指令的操作数*/
④ *new_operand* = *get_operand*(*new_inst*, *operand_idx*); /*求得转换得到新指令的操作数*/
/*调用函数 *cal_match_score* 求得最优转换模式和分数*/
⑤ *operand_score* = *cal_match_score*(*base_operand*, *new_operand*);
⑥ *score* += *operand_score*;
⑦ end for
⑧ return *score*.

值得注意的是，由于基于 shuffle 指令的变换方式会引入额外运行代价，SLP-M 评估其收益需特殊考虑，在上述类似评估方法的基础上减去生成 shuffle 指令的代价.SLP-M 在评估完基于 shuffle 指令的收益后，将其与其他同构化转换方法的收益比较，选择收益较高的同构化转换方法. 待完成处理程序的整体同构化转换方式的选择后，SLP-M 利用 LLVM 中 SLP 实现的评估模型判定该程序是否实施向量化，若评估有收益则实施向量化.

处理程序对应依赖图的高度对 SLP-M 的性能收益和编译时间存在较大的影响.SLP-M 利用操作符和操作数的类型信息进行同构化转换方法的选择，分析程序对应依赖图高度越高，采用的类型相似信息越多，越有助于同构化转换方法的选择. 然而，采用优化程序对应依赖图高度过高并不总能有效提升程序的性能收益，并且由于需要分析更多的同构化转换选择方式和评估计算，增加了较多的编译优化时间. 其实，为了有效提升程序的性能，考虑整体优化程序对应依赖图不是必要的. 由于 SLP-M 方法与 SLP 都是类似自底向上构建同构图，与根节点距离较近的节点对于向量化的收益评估更重要[7]，而节点的高度越高，其对应向量化的收益评估的重要性越低. 由于依赖图高度较高的节点依赖于高度较低节点，因同构化转换需要生成更多数量的额外代价指令的概率增高，因此高度较高的节点可向量化的难度更大. 我们在实验中发现绝大部分可有效向量化的层数不超过 20.

同构化转换方法选择的具体实施过程示例如图 7 所示.图 7(a)(b)分别为一段包含 4 条语句的输入程序及其对应的数据依赖图. 假设 SLP-M 正在处

理由指令 *x*0, *x*1, *x*2, *x*3 组成的指令序列〈*x*0, *x*1, *x*2, *x*3〉, 其操作分别为访存、移位、乘法、移位. 为了选择基准指令, SLP-M 遍历指令序列中的每条指令, 计算与其他指令在对应依赖图中匹配的节点个数或者可通过二元表达式替换可转换为匹配的节点个数, 将其作为基准指令的评估分数.



(a) 输入程序 (b) 依赖图 (c) 选择基准指令

(d) 指令对〈x_2, $x0$〉的同构化转换 (e) 指令对〈x_2, x_1〉的同构化转换 (f) 指令对〈x_2, x_3〉的同构化转换

(g) 指令序列〈x_0, x_1, x_2, x_3〉的同构化转换方式 (h) 指令序列〈x_0, x_1, x_2, x_3〉的同构化转换

Fig. 7 An example of isomorphism transform selection method

图 7 同构化转换选择方法示例

首先, 评估 *x*0 作为基准指令的分数, 其他指令及其二元表达式替换后的指令与 *x*0 都没有操作类型相同的形式, 因此指令 *x*0 作为基准指令的评估分数是 0.

其次, 评估 *x*1 作为基准指令的分数, *x*0 和 *x*2 及其二元表达式替换后的指令与 *x*1 都没有操作类型相同的形式, 因此 *x*1 分别与 *x*0 和 *x*2 的匹配分数均是 0, *x*3 与 *x*1 的操作类型相同, 继续沿着对应依赖图, 向其子孙节点递归计算匹配分数, 共计包含 3 个匹配节点, 分别是移位操作、内存访问操作、常量操作. 因此 *x*1 与 *x*3 的匹配分数是 3. 综上, *x*1 作为基准指令的评估分数是与上述其他指令匹配分数的总和, 即 *score_sum*(*x*1)=0+0+3=3.

然后, 评估 *x*2 作为基准指令的分数, *x*0 与 *x*2 没有操作类型相同的形式, 因此 *x*2 与 *x*0 的匹配分数是 0, *x*1 和 *x*3 都可通过二元表达式替换转换为与 *x*2 操作类型相同的形式, 继续沿着对应依赖图, 向其子孙节点递归计算匹配分数得到 *x*2 分别与 *x*1 和 *x*3 的匹配分数都是 3. 综上, *x*2 作为基准指令的评估分数 *score_sum*(*x*2)= 0+3+3=6.

最后, 评估 *x*3 作为基准指令的分数, 与计算 *x*1 作为基准指令的评估过程类似, 计算得到 *x*3 作为基准指令的评估分数是 3.根据上文计算得到的匹配分数, 指令 *x*2 的分数最多, 因此 SLP-M 将其作为基准指令.

SLP-M 在确定基准指令后, 搜索在指令序列〈*x*0, *x*1, *x*2, *x*3〉中包含基准指令 *x*2 的所有指令对, 如指令对〈*x*2, *x*0〉, 〈*x*2, *x*1〉, 〈*x*2, *x*3〉, 然后针对每个指令对, 搜索所有适用的转换模式 *pattern*0, *pattern*1, *pattern*2 等. 接下来计算每个同构化转换模式下的指令对的整体自动向量化的收益.

指令对〈*x*2, *x*0〉包含同构化转换模式 *pattern*0. *pattern*0 将 *x*0 扩展变换为 *x*0′, 根据 3.4 节所示的单层指令对收益评估方法, *x*2 与 *x*0′的单层指令对的收益为 0, 继续沿着对应依赖图, 向其子孙节点递归计算单层指令对的收益, *x*2 和 *x*0′的第 1 个操作数节点对应的指令分别是 *B*[2] 和 *B*[0], 其操作类型相同, 因此 *B*[2] 和 *B*[0] 的单层指令对的收益为 1, 同理 *x*2 和 *x*0′的第 2 个操作数都是常量操作, 它们对应的单层指令对的收益为 1, 进而得出指令对〈*x*2, *x*0〉在转换模式 *pattern*0 下的整体指令对的自动向量化收益为 *score*(*pattern*0)=0+1+1=2.

指令对〈*x*2, *x*1〉包含同构化转换模式 *pattern*1 和 *pattern*2.*pattern*1 将 *x*1 扩展变换为 *x*1′, 根据 3.4 节所示的单层指令对收益评估方法, *x*2 与 *x*1′的单层指令对收益为 0, 继续沿着对应依赖图, 向其子孙节点递

归计算单层指令对的收益, $x2$ 和 $x1'$ 的第 1 个操作数节点对应的指令分别是 $B[2]$ 和移位操作, 其操作类型不同, 并且没有适用的同构化方法可将移位操作转为与 $B[2]$ 操作类型相同的形式, 因此 $B[2]$ 和移位操作的单层指令对的收益为 0.$x2$ 和 $x1'$ 的第 2 个操作数都是常量操作, 它们对应的单层指令对的收益为 1.进而得出指令对〈$x2$, $x1$〉在转换模式 *pattern*1 下的整体指令对的自动向量化收益为 *score*(*pattern*1)=1. *pattern*2 是将 $B[1]<<C$ 等价替换为 $B[1]\times C'$, 其中 $x1'$ 是由 $x1$ 二元表达式替换生成的, $x2$ 与 $x1'$ 的单层指令对的收益为 1, 继续沿着对应依赖图, 向其子孙节点递归计算单层指令对的收益, $x2$ 和 $x1'$ 的第 1 个操作数节点对应的指令分别是 $B[2]$ 和 $B[1]$, 其操作类型相同, $B[2]$ 和 $B[1]$ 的单层指令对的收益为 1. 同理 $x2$ 和 $x1'$ 的第 2 个操作数都是常量操作, 它们对应的单层指令对的收益为 1, 进而得出指令对〈$x2$, $x1$〉在转换模式 *pattern*2 下的整体指令对的自动向量化收益为 *score*(*pattern*2)=1+1+1=3.

指令对〈$x2$, $x3$〉包含同构化转换模式 *pattern*3 和 *pattern*4, 其计算过程与 *pattern*1 和 *pattern*2 的计算过程类似, 它们整体指令对的自动向量化收益分别为 3 和 1.

在计算完所有转换模式下的整体指令对的自动向量化收益, 根据收益选择每个指令对的同构化转换方式. 指令对〈$x2$, $x0$〉只包含一种同构化转换模式 *pattern*0, 将其作为指令对〈$x2$, $x0$〉同构化转换方式. 指令对〈$x2$, $x1$〉包含 2 种同构化转换方式 *pattern*1 和 *pattern*2, 由于 *pattern*2 下的整体指令对的自动向量化收益相对较高, 因此 *pattern*2 作为指令对〈$x2$, $x1$〉的同构化转换方式, 同理, *pattern*3 作为指令对〈$x2$, $x3$〉的同构化转换方式.

在分析完成指令对〈$x2$, $x0$〉, 〈$x2$, $x1$〉, 〈$x2$, $x3$〉的同构化转换方式后, SLP-M 也就确定了指令序列〈$x0$, $x1$, $x2$, $x3$〉中每条指令的同构化转换方式, 如图 7(g)(h)所示.

在选出同构化转换方法后, SLP-M 方法实施相应转换, 并继续向其子节点方向扩展同构链, 直到遇到叶子节点或者无适用的同构化转换为止.

对于可同时处理的指令数量, SLP-M 方法与已有方法如 PSLP 和 LSLP 等类似, 根据实际处理器所支持的 SIMD 扩展部件的寄存器长度和程序特征选择向量化因子[14]. 若处理指令存在差异如指令类型或排布等, 本文将尝试利用扩展变换和二元表达式替换等同构化转换方法并进行选择. 若可利用上述方式进行同构化转换且有收益则进行向量化, 否则采用与 PSLP 和 LSLP 类似的方法, 将向量化因子减半, 继续进行同构化分析及转换, 直到存在有收益的向量化转换方式或者向量化因子等于 1 时为止.

4 实验与结果分析

本文基于 LLVMv10.0 实现了 SLP-M 方法, 从构造函数、基于 SPEC CPU 2017/2006/2000 和 MediaBench[28] 测试集提取的核心函数、整体测试 3 个方面对 SLP-M 方法进行测试, 并与 SLP 方法和 PSLP 方法进行对比. 目前 LLVM 中实现的 SLP 方法如 LLVM-SLP 已经集成了 LSLP 和 SN-SLP 方法, 本文不再另外单独将 LSLP 和 SN-SLP 与 SLP-M 比较. 实验所用计算机的处理器型号为 Intel i7-4 790, 其主频为 3.2 GHz, 支持 AVX2, AVX1, SSE 向量指令集, 其中 AVX2 的向量寄存器长度为 256 b, 它能同时处理 4 个 double 数据或者 8 个 float 数据.AVX1 和 SSE 的向量寄存器长度为 128 b, 它能同时处理 2 个 double 数据或者 4 个 float 数据. 测试处理器包含 4 个物理核, 每个物理核可支持 2 个逻辑核, L1 缓存大小为 32 KB(8way, 64B/line), L2 为 256 KB(8way, 64B/line), L3 为 8 MB(共享内存), 内存为 20 GB, 操作系统为 Ubuntu20.04. 实数域的运算性质在计算机的浮点实现存在偏差[29-30], 为了保证基于等价表达式的程序变换, 维持原始程序语义不变, 本文采用原始 LLVM 编译器中相关变换对操作及操作数类型的限制规则, 具体详见 LLVM 的常量折叠以及强度削弱优化的具体实现[31]. 如 3.4 节所述随着彼此依赖指令数量的增多, 本文方法所需的编译运行时间逐步增多, 绝大部分可有效向量化程序对应依赖图高度较低, 为了确保 SLP-M 编译优化在合理时间范围内, 本文实验限定优化程序对应依赖图的高度不超过 20.后续程序员可综合考虑程序特征和优化时间设定该数值.

4.1 构造函数测试

为了考察各种 SLP 向量化方法的分析处理能力, 本文构造了多种非同构指令序列的程序, 覆盖了各种典型的语句间差异, 包括操作类型不同、操作个数不同、数据类型不同及语句数量不同. 具体如表 3 所示.

本文测试了多种自动向量化方法在构造测试程序上运行时间, 计算了各种自动向量化方法相对于在关闭 SLP 向量化优化条件下的 O3 优化加速比, 也就是 LLVM 编译器关闭 SLP 向量化功能的优化, 并

打开其他所有 LLVM O3 编译优化功能，包括开启循环自动向量化和循环展开优化等，其编译选项为–O3 –march=haswell –mtune=haswell –fno-slp-vectorize，后续为描述方便，本文利用 O3 表示上述的编译优化. 其他多种自动向量化方法都是在 O3 优化条件的基础上打开特定的向量化优化方法. 除了测试 SLP-M 外，本文还测试了在 SLP-M 优化基础上关闭二元表达式替换功能优化的性能，用于评估二元表达式替换优化对性能的影响，为了描述方便，用 SLP-M-RelaceOff 表示. 性能结果如图 8 所示，其中，横轴表示测试程序，纵轴表示不同方法在测试程序上相对于 O3 的性能加速比 .LLVM-SLP、PSLP、SLP-M-RelaceOff、SLP-M 的平均加速比分别为 1.08, 1.29, 1.34, 3.17.对于大部分程序，SLP-M 显著提升了程序的性能，而 LLVM-SLP 和 PSLP 对程序的性能提升幅度相对较小.

Table 3 Constructed Functions

表 3 构造函数

测试函数	核心代码	程序特征	测试函数	核心代码	程序特征
*s*1	*a*[0] = (*b*[0]+*c*[0]) ×5.0; *a*[1] = *b*[1]+*c*[1].	操作个数不同，双精度浮点类型，2 条语句	*s*9	*a*[0] = *b*[0]/4.0; *a*[1] = *b*[1]×5.0; *a*[2] = *b*[2]×6.0; *a*[3] = *b*[3]×7.0.	操作类型不同，单精度浮点类型，4 条语句

*s*2	*a*[0] = *b*[0]/11.0;
	a[1] = *b*[1]×13.0.
		操作类型不同，双精度浮点类型，2 条语句	*s*10	*a*[0] = *b*[0]+*c*[0];
				a[1] = (*b*[1]+*c*[1]) ×5.0;
				a[2] = (*b*[2]+*c*[2]) /4.0;
				a[3] = (*b*[3]+*c*[3]) ×7.0.
					操作个数和类型不同，单精度浮点类型，4 条语句

*s*3	*a*[0] = *b*[0]+11.0; *a*[1] = *b*[1]×13.0.	操作类型不同，双精度浮点类型，2 条语句	*s*11	*a*[0] = *b*[0]−4.0; *a*[1] = *b*[1]×5; *a*[2] = *b*[2]/4.0; *a*[3] = *b*[3]×7.	操作类型不同，单精度浮点类型，4 条语句
*s*4	*a*[0] = *b*[0]; *a*[1] = *b*[1]×11; *a*[2] = *b*[2]×11; *a*[3] = *b*[3]×11.	操作个数不同，整数类型，4 条语句	*s*12	*a*[0] = *b*[0]+*c*[0]; *a*[1] = (*b*[1]+*c*[1]) ×5.0; *a*[2] = (*b*[2]+*c*[2]) ×6.0; *a*[3] = (*b*[3]+*c*[3]) ×7.0.	操作个数不同，双精度浮点类型，4 条语句
*s*5	*a*[0] = *b*[0]<<2; *a*[1] = *b*[1]×5; *a*[2] = *b*[2]×6; *a*[3] = *b*[3]×7.	操作类型不同，整数类型，4 条语句	*s*13	*a*[0] = *b*[0]/4.0; *a*[1] = *b*[1]×5.0; *a*[2] = *b*[2]×6.0; *a*[3] = *b*[3]×7.0.	操作类型不同，双精度浮点类型，4 条语句

*s*6	*a*[0] = *b*[0];
	a[1] = *b*[1]×5;
	a[2] = *b*[2]<<2;
	a[3] = *b*[3]×7.
		操作个数和类型不同，整数类型，4 条语句	*s*14	*a*[0] = *b*[0]+*c*[0];
				a[1] = (*b*[1]+*c*[1]) ×5.0;
				a[2] = (*b*[2]+*c*[2]) /4.0;
				a[3] = (*b*[3]+*c*[3]) ×7.0.
					操作个数和类型不同，双精度浮点类型，4 条语句

*s*7	*a*[0] = *b*[0]−4; *a*[1] = *b*[1]×5; *a*[2] = *b*[2]<<2; *a*[3] = *b*[3]×7.	操作类型不同，整数类型，4 条语句	*s*15	*a*[0] = *b*[0]−4.0; *a*[1] = *b*[1]×5; *a*[2] = *b*[2]/4.0; *a*[3] = *b*[3]×7.	操作类型不同，双精度浮点类型，4 条语句
*s*8	*a*[0] = *b*[0]+*c*[0]; *a*[1] = (*b*[1]+*c*[1]) ×5.0; *a*[2] = (*b*[2]+*c*[2]) ×6.0; *a*[3] = (*b*[3]+*c*[3]) ×7.0.	操作个数不同，单精度浮点类型，4 条语句	*s*16	*a*[0] = *b*[0]−4.0; *a*[1] = 5×*b*[1]; *a*[2] = *b*[2]/4.0; *a*[3] = *b*[3]×7.	操作类型不同，双精度浮点类型，4 条语句

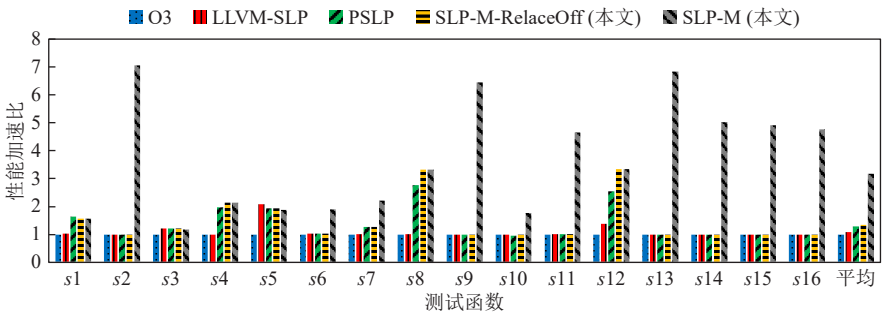


Fig. 8 Speedup ratios of various methods on constructing kernel functions

图 8 不同方法在构造核心函数上的加速比

下面分析多种优化方法对构造函数编译优化的测试结果,分别从操作个数不同、操作类型不同、操作个数及操作类型都不同的3类构造函数进行分析.

1)操作个数不同的程序包括测试程序 *s*1, *s*4, *s*8 等.LLVM-SLP 对其中大部分程序,如 *s*1 和 *s*4,未实施自动向量化,也未有效提升该类程序的性能.PSLP, SLP-M-RelaceOff, SLP-M 对其中所有程序实施了自动向量化,进一步提升了程序的性能.

2)操作类型不同的程序包括测试程序 *s*2, *s*5 和 *s*9 等.LLVM-SLP, PSLP, SLP-M-RelaceOff 通过生成 shuffle 指令,对其中少数程序实施了自动向量化(如 *s*2 和 *s*5),提升了该类程序的性能.SLP-M 对该类程序都实施了自动向量化,且通过二元表达式替换和减少了其中的操作种类数量,进一步提升了该类程序的性能.例如 LLVM-SLP, PSLP, SLP-M-RelaceOff, SLP-M 都对 *s*9 实施了自动向量化,LLVM-SLP, PSLP, SLP-M-RelaceOff 都利用了向量乘法、向量除法、shuffle 指令组合实施了自动向量化.我们发现尽管 LLVM-SLP 方法采用了向量的代码形式,而与 O3 采用标量的代码形式比较,并未带来实际性能提升.SLP-M 将 *s*9 中的除法转为乘法,并实施了自动向量化,由于乘法操作相比于除法操作在 Intel 机器上实际执行快数倍,因此显著提升了程序的性能,*s*2 也类似.

3)操作数量及类型都不同的程序包括测试程序 *s*6, *s*10, *s*15 等,LLVM-SLP, PSLP, SLP-M-RelaceOff 对于其中大部分程序要么未实施向量化,要么只对其中部分语句实施向量化,对该类程序的程序的性能提升幅度较少.SLP-M 对该类程序都实施了自动向量化,显著地提升了该类程序的性能.例如 *s*6 包含访存、乘法、移位操作.LLVM-SLP 未对该类程序实施自动向量化,PSLP 方法和 SLP-M-RelaceOff 方法尽管可将该类程序转换为同构的形式,但是向量化无收益,因此未实施向量化.SLP-M 方法采用了多种同构化转换方法,将 *s*6 第 1 条语句的 *b*[0] 转换为 *b*[0]×1,将第 3 条语句中的移位操作转为乘法操作,进而实施了向量化,有效提升了程序的性能.对于 *s*15, SLP-M 可显著提升其性能的原因与 *s*9 类似,都是减少了向量除法操作以及重组指令操作带来的性能提升.

除了测试和计算了性能加速比外,本文还通过 LLVM 自带的性能评估模型,评估了各种向量化方法对于构造核心函数的性能收益.图 9 是 LLVM 统计多种优化方法对测试程序相对于 O3 编译优化减少指令代价的统计图,其中横轴表示核心函数,纵轴表示程序经过自动向量化方法的优化后,利用 LLVM 评估模型得出的减少的指令代价.总体而言,SLP-M 方法自动向量化减少的指令代价多于其他方法,这反映出 SLP-M 方法较强的自动向量化优化能力,能够有效提升向量化的性能收益.

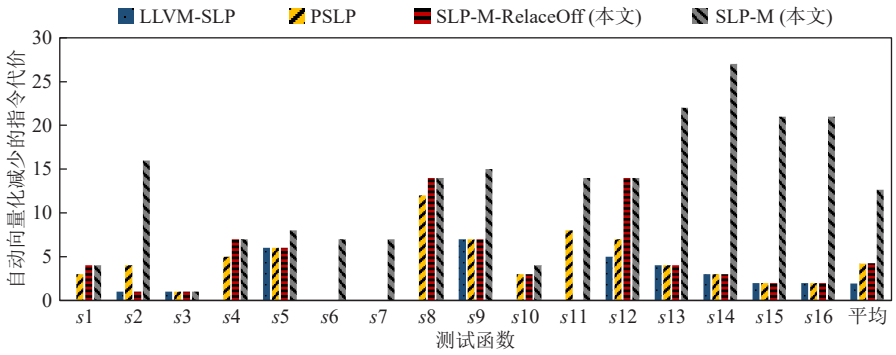


Fig. 9 Reduced instruction costs of auto-vectorization methods on constructing kernel functions

图 9 构造核心函数自动向量化方法减少的指令代价

4.2 实际应用核心函数测试

本文从实际应用如 SPEC CPU 2017/2006/2000 和 MediaBench2 基准测试集中提取代表性的核心函数代码片段,如表 4 所示.提取的核心函数代码片段主要来自在科学计算领域和多媒体领域计算密集型的代码(如生物系统模拟和影像关系追踪等),其热点包含较多的非同构指令序列,代码覆盖了程序中语句间的典型差异,包括操作类型不同、操作个数不同,同时包含类型及个数上的差异.本文通过在循环中多次执行同一个核心函数,使得总体运行时间超过 20 min,以减小环境因素影响.

本文测试了多种自动向量化方法在核心函数上的运行时间,计算了各种自动向量化方法相对于 O3 的加速比,结果如图 10 所示,其中,横轴表示测试程序,纵轴表示不同方法在测试程序上相对于 O3 的性能加速比,LLVM-SLP, PSLP, SLP-M-RelaceOff, SLP-M 的平均加速比分别为 1.0, 1.25, 1.1, 1.6.SLP-M 相对于 LLVM-SLP(包含 LSLP 和 SN-SLP)的平均性能提

Table 4 Description of the Kernel Functions

表 4 核心函数描述

序号	函数	描述	类型	语言
1	*gl_render_vb*	OpenGL 函数库（SPEC CPU 2000 177）	操作个数不同	C
2	*u2s*	PERL 编程语言（SPEC CPU 2006 400）	操作个数不同	C
3	*calc_pair_energy*	生物系统模拟（SPEC CPU 2006 444）	操作个数不同	C++
4	*start_pass_fdctmgr*	图像压缩（MediaBench2 cjpeg）	操作类型不同	C
5	*ssim_end*1	x264 编解码（SPEC CPU 2017 625）	操作类型不同	C
6	*box_UVCoord*	影像光线追踪（SPEC CPU 2006 453）	操作类型不同	C++
7	*intra*16*x*16*_plane_pred_mbaff*	x264 编解码（SPEC CPU 2017 625）	操作个数和类型不同	C
8	*intra*16*x*16*_plane_pred*	x264 编解码（SPEC CPU 2017 625）	操作个数和类型不同	C
9	*start_pass*	图像压缩（MediaBench2 cjpeg）	操作个数和类型不同	C

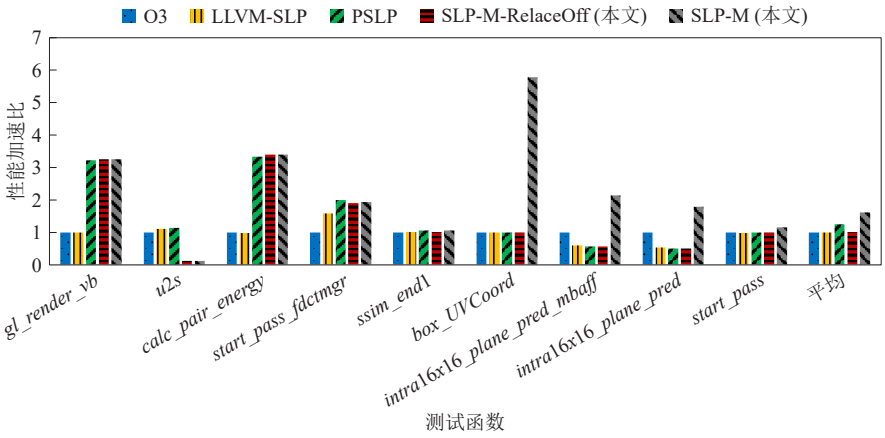


Fig. 10 Speedup ratios of various methods on kernel functions

图 10 不同方法在核心函数上的加速比

升了 40.4%, 相对于 PSLP 的平均性能提升了 21.8%.

下面分析多种优化方法对实际应用核心函数编译优化的测试结果, 也是分别对不同操作个数、不同操作类型、不同操作个数及操作类型的 3 类程序进行分析.

对于第 1 类操作个数不同的程序, SLP 未能有效提升该类程序的性能, PSLP, SLP-M-RelaceOff, SLP-M 可显著提升其中部分程序的性能. 对于函数 *gl_render_vb*, 其核心代码片段如图 11(a)所示, 尽管 4 条语句都是同构的, 但是第 4 条语句在自动向量化前编译器将其中的 $i-0$ 变换为 i, 使得转换后的语句不是同构的, LLVM-SLP 未对该程序实施自动向量化, PSLP, SLP-M-RelaceOff, SLP-M 对其实施了自动向量化, 具体的不同自动向量化生成的汇编代码如图 12 所示. 对于函数 *u2s*, 其核心代码片段如图 11(b)所示, LLVM-SLP 和 PSLP 同构化转换能力较弱, 无法对其进行同构化转换, 因此未对其实施自动向量化, 实际以标量形式执行. 而 SLP-M-RelaceOff 和 SLP-M 尽管

```
vlist[0]=i-3;
vlist[1]=i-2;
vlist[2]=i-1;
vlist[3]=i-0;
```

(a) *gl_render_vb*

```
*s++=(U8)( u       & 0xFF);
*s++=(U8)(( u >> 8) & 0xFF);
*s++=(U8)(( u >> 16) & 0xFF);
*s   =(U8)(( u >> 24) & 0xFF);
```

(b) *u2s*

```
pli[0]=j;
pli[1]=j+1;
pli[2]=j+2;
pli[3]=j+3;
```

(c) *calc_pair_energy*

Fig. 11 Code fragments of the kernel function with different numbers with opcodes

图 11 不同操作数量的核心函数代码片段

```
⋮
vmovdqa %xmm1, mem1[]
lea     -0x3(%rcx), %edx
mov     %edx,mem2[]
lea     -0x2(%rcx), %edx
⋮
```

(a) O3/LLVM-SLP

```
⋮
vpbroadcastd mem1[], %xmm1
vpsubd %xmm0, %xmm1, %xmm1
vmovdqa %xmm1, mem2[]
⋮
```

(b) PSLP/SLP-M-RelaceOff/SLP-M

Fig. 12 Assembly instructions generated by the code fragments of the function *gl_render_vb*

图 12 函数 *gl_render_vb* 代码片断生成的汇编指令

可对其实施自动向量化, 但是性能反而不如标量程序的性能, 通过查看汇编程序发现, SLP-M 将 32 b 转

换为 8 b 类型数据中生成了代价较高的非对齐访存以及重组指令，在这种情况下自动向量化不能带来性能收益，然而在实施向量化评估中本文采用了 LLVM 的评估代价模型，其分析是有收益的，因此对该程序实施了向量化，其性能不如标量程序. 函数 *calc_pair_energy* 与函数 *gl_render_vb* 类似，如图 11(c)，SLP 未对其实施自动向量化，PSLP, SLP-M-RelaceOff, SLP-M 对其实施了自动向量化.

对于第 2 类操作类型不同的程序，SLP-M 与 LLVM-SLP, PSLP, SLP-M-RelaceOff 比较，对该类程序性能的提升幅度更大. 对于函数 *start_pass_fdctmgr*，其核心代码片段如图 13(a)所示，LLVM-SLP, PSLP, SLP-M-RelaceOff, SLP-M 都对其实施了向量化. 函数 *box_UVCoord* 多处包含如图 13(b)所示的代码片段，编译器在自动向量化前将其中的第 1 个除法转为乘法，使得 2 条语句转换为非同构的形式.LLVM-SLP 未对图 13(b)中程序实施了自动向量化，PSLP 和 SLP-M-RelaceOff 通过生成向量除法指令，对图 13(b)中的程序实施了自动向量化.SLP-M 方法将除法指令转为乘法指令，并利用向量乘法指令对其实施向量化，与 PSLP 比较，未生成延时较大的向量除法，由于乘法操作相比于除法操作在 Intel 机器上实际执行将近数倍，因此显著提升了程序的性能，具体不同自动向量化生成的汇编代码如图 14所示. 函数 *ssim_end*1 核心代码以及优化方法生成的汇编代码如图 13(c)和图 14 所示，LLVM-SLP, PSLP, SLP-M-RelaceOff, SLP-M 都未对其实施自动向量化，它们优化后程序的性能差异不大.

```
tmp1 = quantval[0]<<14
tmp2 = quantval[1]×22 725
tmp3 = quantval[2]×21 407
tmp4 = quantval[3]×19 266
```

(a) *start_pass_fdctmgr*

```
result[U] = (1.0 / 4.0) + (P[X] / 4.0);
result[V] = (1.0 / 3.0) + (P[Y] / 3.0);
```

(b) *box_UVCoord*

```
static float ssim_end1( int s1, int s2, int ss, int s12 ){
        static constintssim_c1 = (int)(0.01×0.01×255×255×64+0.5);
        static constintssim_c2 = (int)(0.03×0.03×255×255×64×63+0.5);
        int vars = ss×64−s1×s1−s2×s2;
        int covar = s12×64−s1×s2;
        return (float)(2×s1×s2 + ssim_c1) × (float)(2×covar+ ssim_c2) /
          ((float)(s1×s1 +s2×s2 + ssim_c1) × (float)(vars+ ssim_c2));
}
static float ssim_end4( int sum0[5][4], int sum1[5][4], int width ){
        float ssim = 0.0;
        for( int i = 0; i < width; i++ )
          ssim += ssim_end1( sum0[i][0] + sum0[i+1][0] + sum1[i][0] + sum1[i+1][0],
                sum0[i][1] + sum0[i+1][1] + sum1[i][1] + sum1[i+1][1],
                sum0[i][2] + sum0[i+1][2] + sum1[i][2] + sum1[i+1][2],
                sum0[i][3] + sum0[i+1][3] + sum1[i][3] + sum1[i+1][3] );
        return ssim;
}
```

(c) *ssim_end*1

Fig. 13 Code fragments of the kernel function of different types of opcodes

图 13 不同操作类型的核心函数代码片段

```
⋮
vmulsd mem1[],%xmm0,%xmm3
vaddsd %xmm1,%xmm3,%xmm3
vmovsd mem2[],%xmm4
vdivsd %xmm2,%xmm4,%xmm4
⋮
```

(a) O3

```
⋮
vmovapd mem1[],%xmm2
vdivpd %xmm0,%xmm2,%xmm2
vaddsd %xmm1,%xmm2,%xmm2
vmovapd %xmm2,mem2[]
⋮
```

(b) LLVM-SLP/PSLP/SLP-M-RelaceOff

```
⋮
vmulpd mem1[],%xmm0,%xmm2
vaddpd %xmm1,%xmm2,%xmm2
vmovapd %xmm2,mem2[]
⋮
```

(c) SLP-M

Fig. 14 Assembly instructions for the code fragments of the function *box_UVCoord* generated by various auto-vectorization methods

图 14 多种自动向量化方法对于函数 *box_UVCoord* 代码片断生成的汇编指令

对于第 3 类操作数量及类型都不同的程序，这类程序需利用多种同构化转换方法并进行合理选择，才能对其实施自动向量化，目前在被测试的优化方法中只有 SLP-M 显著提升了该类程序的性能. 如函数 *intra*16*x*16*_plane_pred_mbaff*，其单层循环的核心代码片段如图 15(a)所示，其中 *i* 是归纳变量，编译器在自

```
×prd++ = (imgpel) iClip1(max_imgpel_value, ((ibb + (i−7) × ib) >> 5));
×prd++ = (imgpel) iClip1(max_imgpel_value, ((ibb + (i−6) × ib) >> 5));
×prd++ = (imgpel) iClip1(max_imgpel_value, ((ibb + (i−5) × ib) >> 5));
×prd++ = (imgpel) iClip1(max_imgpel_value, ((ibb + (i−4) × ib) >> 5));
```

(a) *intra*16*x*16_*plane_pred_mbaff* 函数代码片段

```
⋮
mov    0x2ee9(%rip),%ecx
mov    0x34bd3(%rip),%edx
shr    $0x6,%edx
lea    (%rdx,%rcx,1),%esi
mov    %esi,0x16767(%rip)
lea    (%rcx,%rdx,2),%esi
⋮
```

(b) O3

```
⋮
vmovd  %r8d,%xmm1
vpinsrd $0x1,%r10d,%xmm1,%xmm1
vpinsrd $0x2,%esi,%xmm1,%xmm1
vpinsrd $0x3,%edi,%xmm1,%xmm1
vinserti128 $0x1,%xmm1,%ymm0,%ymm0
vpbroadcastd 0x2e88(%rip),%ymm1
vpaddd %ymm1,%ymm0,%ymm0
vmovdqu %ymm0,0x1670c(%rip)
⋮
```

(c) LLVM-SLP/PSLP/SLP-M-RelaceOff

```
⋮
vpermd %ymm3,%ymm1,%ymm3
vpmulld %ymm2,%ymm3,%ymm3
vpbroadcastd 0x2eb5(%rip),%ymm4
vpaddd %ymm4,%ymm3,%ymm3
vmovdqu %ymm3,0x16739(%rip)
⋮
```

(d) SLP-M

Fig. 15 Code fragments of the function *intra*16*x*16_*plane_pred_mbaff*

图 15 函数 *intra*16*x*16_*plane_pred_mbaff* 代码片段

动向量化优化前，对于不同的 i 值将代码转为不同的形式如 i = 8，那么第 1 行语句，(*i*−7)×*ib* 会被转换为 *ib*，第 4 行语句的(*i*−4)×*ib* 转为 *ib*<<2.LLVM-SLP 未对原始程序实施自动向量化，PSLP 将函数实施了同构化转换，但是需添加较多数量的重组指令，使得向量化无性能收益，因而未实施向量化. SLP-M 同时选择运用了扩展变换和二元表达式替换变换，将函数转为同构的形式，有效实施了自动向量化，提升了性能，具体不同方法生成的汇编代码如图 15(b)(c)(d)所示.

函数 *intra*16*x*16_*plane_pred* 核心代码片段如图 16 所示，与函数 *intra*16*x*16_*plane_pred_mbaff* 类似，只有 SLP-M 方法能够对其实施自动向量化.

函数 *start_pass* 核心代码片段如图 17 所示，尽管原始代码是同构的，但是编译器在自动向量化前通过常量折叠优化将语句 1 中((*b*[0]×16 384)+1 024)>>11 转为 *b*[0]<<3，变换后的语句并不是同构的，LLVM-SLP, PSLP, SLP-M-RelaceOff 都未对其实施自动向量化，SLP-M 利用多种同构化转换方法，对左移和右移操作时通过添加 shuffle 指令进行扩展，然后通过 2 步扩展变换将 *b*[0] 转换为 *b*[0]×1+0 的形式，进而将

```
for (i = 0;i < MB_BLOCK_SIZE; ++i)
{
    mb_pred[j][i] = (imgpel) iClip1(max_imgpel_value,
      ((iaa + (i −7) × ib + (j −7) × ic + 16) >> 5))
}
```

Fig. 16 Code fragments of the function *intra*16*x*16_*plane_pred*

图 16 函数 *intra*16*x*16_*plane_pred* 代码片段

原始程序转为同构的形式，并实施了自动向量化，有效提升了程序的性能，具体汇编代码如图 17 所示.

通过上述核心函数测试结果可以看出，对于第 1 类程序，除了 *u2s* 外，SLP-M 与目前业界测试最优方法的向量化优化能力持平，对于第 2, 3 类程序，SLP-M 优于其他方法. 尤其对于第 3 类程序，其他测试方法无法对该类程序进行有效向量化，而 SLP-M 可显著提升了该类程序的性能. 从上述测试看出，对于操作个数或操作类型不同的程序，采用单一的同构化转换方法带来的自动向量化性能收益并不总是最优的，SLP-M 利用了多种同构化转换方法，并对其进行选择，可有效提升这类程序的性能.

与 4.1 节类似，本文也通过 LLVM 自带的性能评估模型评估了各种自动向量化方法对于实际应用核

```
a[0] = ((b[0] × 16 384) + 1 024) >> 11;
a[1] = ((b[1] × 22 725) + 1 024) >> 11;
a[2] = ((b[2] × 21 407) + 1 024) >> 11;
a[3] = ((b[3] × 19 266) + 1 024) >> 11;
```

(a) *start_pass*函数代码片段

```
⋮
mov  mem1[],%ecx
shl  $0x3,%ecx
imul $0x58c5,mem2[],%edx
mov  %ecx,mem3[]
add  $0x400,%edx
sar  $0xb,%edx
mov  %edx,mem4[]
imul $0x539f,mem5[],%ecx
⋮
```

(b) O3/LLVM-SLP/PSLP/SLP-M-RelaceOff

```
⋮
vpmulld mem1[],%xmm0,%xmm3
vpaddd %xmm1,%xmm3,%xmm3
vpsravd %xmm2,%xmm3,%xmm3
vmovdqa %xmm3,mem2[]
⋮
```

(c) SLP-M

Fig. 17 Assembly instructions for the code fragments of the function *start_pass* generated by various auto-vectorization methods

图 17 多种自动向量化方法对于函数 *start_pass* 代码片断生成的汇编指令

心函数的性能收益.具体结果如图 18 所示.总体而言，SLP-M 自动化向量化减少的指令代价多于 LLVM-SLP 和 PSLP，这反映出 SLP-M 具备较强的向量化优化能力，能够有效提升向量化的性能收益.

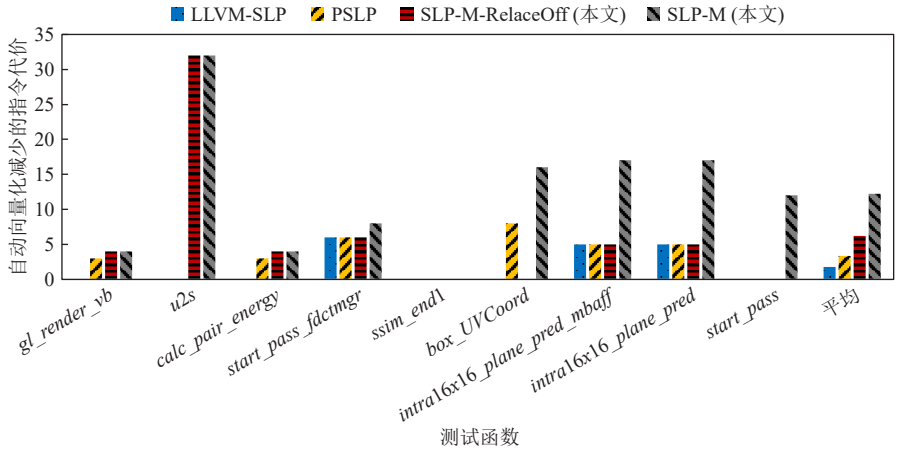


Fig. 18 Reduced instruction costs of auto-vectorization methods on kernel functions

图 18 核心函数自动向量化方法减少的指令代价

4.3 整体测试

本文利用SPECCPU2017/2006/2000 和MediaBench2 基准测试集测试 SLP-M 的整体加速比，排除了实验的自动向量化方法都未有效触发实施转换的程序，执行基准测试集 12 次，分别剔除性能最优以及最差的测试数据，然后对剩余的测试数据取几何平均值.整体加速比如图 19 所示，其中横轴表示测试程序，纵轴表示不同方法在测试程序上相对于 O3 的性能加速比. LLVM-SLP, PSLP, SLP-M-RelaceOff, SLP-M 的平均加速比分别为 1.06, 1.07, 1.09, 1.11.SLP-M 方法对于整体性能的提升优于实验的其他自动向量化方法.

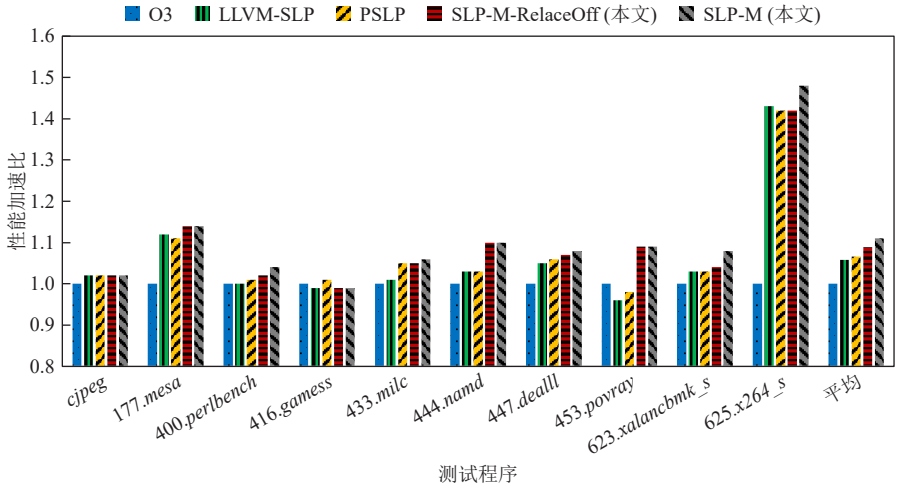


Fig. 19 Speedup ratios of various auto-vectorization methods on whole benchmarks

图 19 不同自动向量化方法在整体基准测试程序集上的加速比

SLP-M 相比于其他向量化方法对不同类型程序性能提升幅度不同，分别对 3 类情况进行论述：1）SLP-M 与已有方法相比程序的性能进一步提升；2）SLP-M 与已有方法相比程序的性能下降；3）SLP-M 与已有方法相比程序性能提升幅度趋同.

对于第 1 种类型如 *453.povray*, *623.xalancbmk_s*, *625.x264_s* 测试程序，SLP-M 与已有方法比较进一步提升了程序的性能.由于 SLP-M 应用了多种同构化转换方法并进行选择，提升了同构化转换能力，进而增加了向量化的机会. *453.povray*, *623.xalancbmk_s*, *625.x264_s* 包含较多加法、减法、乘法组合运算操作，SLP-M 对这些程序利用二元表达式等价替换及其他同构化转换方法对该类程序转换同构指令序列，并实施了自动向量化，有效提升了这些程序的性能.

对于第 2 种类型如 *416.gamess*，SLP-M 相比于 PSLP 使得 *416.gamess* 性能下降. 我们查看程序编译后生成的汇编代码，分析 LLVM 的评估模型，发现程序性能下降并不是 SLP-M 程序变换引起的，而是由于 LLVM 的评估模型不准确造成的.LLVM 未充分考虑超标量处理器中标量指令级并行和访存因素，对于本质上无收益的程序变换，却认为有收益，进而实施无效的程序变换，导致程序性能下降.

对于第 3 种类型如 *cjpeg*，*433.milc*，*447.dealII* 测试程序，SLP-M 与 PSLP 对于上述程序性能提升幅度趋同，这是由于利用这 2 种方法后实施相同优化转换方式程序的占比较高，对于这些程序 SLP-M 应用了与 PSLP 类似的同构化转换方法（基于 shuffle 指令的变换方式）进行向量化，如 *433.milc* 和 *447.dealII* 热点区域，或都不对其向量化.

SLP-M 对整体的性能提升没有像对核心函数那样提升显著，主要有 3 方面的原因：1）SLP-M 能够有效触发的程序并不是都是应用程序的热点区域；2）SLP-M 进行同构化转换方法的选择属于启发式方法，对有些程序的同构化方法的选择并不是最优的，这就使得程序优化引入的性能提升是有限的；3）LLVM 的自动向量化代价模型准确性能有待提高，本文采用 LLVM 自带的自动向量化评估模型，发现有些程序经过 SLP-M 的同构化分析评估是有收益的，可实施自动向量化，然而，实际上对于有些程序并未带来性能提升，甚至使其性能略有下降.

SLP-M 与现有方法比较可有效提升程序的性能. 由于 SPEC CPU Benchmark 是编译优化领域标杆式的测试集，许多研究者针对该测试集进行研究及优化[20]，且 SLP 本身是自动向量化领域重要的优化方法，也可用于其他多个领域如二进制翻译等，SLP-M 对于 SPEC CPU Benchmark 的性能提升依然是难能可贵的.

5 总　　结

基本的 SLP 方法在处理非同构指令序列时存在局限性. 本文提出了一种 SLP 扩展方法——SLP-M，它与现有方法相比，将多种方法应用于非同构指令序列的同构化转换中，发挥了多种同构化转换方法的优势，进一步扩大了“同构化转换”处理的对象范围，整体提升了自动向量化的性能收益. 该方法并不局限于本文提到的几种同构化转换方法，还可在此基础上扩展应用更多的同构化转换方法，如三角函数等价变换、指数等价变换等. 随着更多方法的应用，有助于进一步发挥多种方法的优势，提升 SLP 向量化的适用范围和收益.

本文基于 LLVM 编译器实现了 SLP-M 方法，并基于 SPEC CPU 2017 等测试集进行了测试实验. 实验表明，对于核心函数代码片段，SLP-M 方法相对于 LLVM-SLP 方法（包含 LSLP 和 SN-SLP 方法）的平均性能提升了 40.4%，相对于 PSLP 方法的平均性能提升了 21.8%. 对于整体性能测试，SLP-M 方法相对于 LLVM-SLP 方法（包含 LSLP 和 SN-SLP 方法）的平均性能提升了 5%，相对于 PSLP 方法的平均性能提升了 4.1%.

指令序列同构化转换方法的选择直接影响 SLP-M 方法的适用范围及性能收益，本文采用的单层指令对和整体指令的向量化收益结合性能评估和选择方法有效提升了程序的性能收益，然而该方法属于启发式方法，实验发现对于少数程序未带来性能提升. 下一步深入分析各种同构化转换方法的特点，从适用范围、性能收益和程序特征等角度考虑，研究更好的选择策略和方法如机器学习方法[32].

作者贡献声明：冯竞舸提出了方法思路和实验方案，完成实验并撰写论文；贺也平、陶秋铭、马恒太提出指导性意见.

参　考　文　献

[1] Gao Wei, Zhao Rongcai, Han Lin, et al. Research on SIMD auto-vectorization compiling optimization[J]. Journal of Software, 2015, 26(6): 1265−1284 (in Chinese)
(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)

[2] Maleki S, Gao Yaoqing, Garzar M J, et al. An evaluation of vectorizing compilers[C] //Proc of the 20th IEEE Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE, 2011: 372−382

[3] Feng Jingge, He Yeping, Tao Qiuming. Auto-vectorization: Recent development and prospect[J]. Journal on Communications, 2022, 43(3): 180−195 (in Chinese)
(冯竞舸, 贺也平, 陶秋铭. 自动向量化: 近期进展与展望[J]. 通信学报, 2022, 43(3): 180−195)

[4] Kennedy K, Allen J R. Optimizing Compilers for Modern Architectures: A Dependence-based Approach [M]. San Francisco, CA: Morgan Kaufmann, 2001

[5] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[J]. Programming Language Design and Implementation, 2000, 35(5): 145−156

[6] Li Yuxiang, Shi Hui, Chen Li. Vectorization-oriented local data regrouping[J]. Computer System, 2009, 30(8): 1528−1534 (in

Feng Jingge, born in 1988. PhD, senior engineer. His main research interests include compiler optimization and performance optimization.

冯竞舸, 1988 年生. 博士，高级工程师. 主要研究方向为编译优化和性能优化.

He Yeping, born in 1962. PhD, professor. His main research interests include system security and privacy protection.

贺也平, 1962 年生. 博士，研究员. 主要研究方向为系统安全、隐私保护.

Tao Qiuming, born in 1979. PhD, associate professor. His main research interests include operating system, compiler, and software engineering.

陶秋铭, 1979 年生. 博士，副研究员. 主要研究方向为操作系统、编译技术、软件工程.

Ma Hengtai, born in 1970. PhD, associate professor. His main research interests include software security analysis and operating system security.

马恒太, 1970 年生. 博士，副研究员. 主要研究方向为软件安全分析、操作系统安全.