学 校 代 码 10459
学号或申请号 201822362014289
密 级

郑州大学

专业硕士学位论文

面向申威平台的 LLVM 自动向量化移植与优化

作 者 姓 名：李嘉楠
导 师 姓 名：韩 林
专业学位名称：工程硕士
培 养 院 系：信息工程学院
完 成 时 间：2021 年 6 月

A thesis submitted to

Zhengzhou University

for the degree of Master

Transplantation and optimization of auto-vectorization of LLVM compiler based on Sunway processor

By Jianan Li

Supervisor: Lin Han

Electronics and Communication Engineering

School of Information Engineering

June, 2021

摘 要

通过自动向量化编译技术自动生成向量程序，以有效地利用微处理器底层SIMD硬件提升程序的执行性能，已成为编译器研究的一个重要方向。然而，由于技术路线不同，以及各种微处理器的SIMD扩展指令集的巨大差异，自动向量化要针对具体的SIMD扩展部件改变算法、调整参数才能适配目标SIMD硬件特性。此外，在自动向量化编译的实现过程中，循环中的控制流结构、基本块中的同构语句数、打包方式的随机性等因素，也需要面向具体处理器体系结构与SIMD扩展指令集的特点开展针对性的优化研究，以充分发掘出程序中潜在的向量并行性。

本文的主要工作和创新点有：

（1）LLVM的自动向量化技术研究。LLVM开源编译器中已初步实现了循环级与基本块级两种方式的自动向量化方法，本文从合法性分析、向量发掘、向量代码生成三个方面分别梳理了LLVM编译器自动向量化模块中循环级与基本块级的代码实现，构建了适用于申威平台的LLVM自动向量化流程，为开展自动向量化功能模块的移植与优化奠定了基础。

（2）循环级向量化的移植与优化。由于向量长度及指令集功能的差异，面向申威平台的自动向量化过程与开源LLVM存在着较多差异，本文从向量寄存器长度以及向量化信息两方面，进行面向申威平台的循环级向量化的移植工作。针对申威平台不支持掩码指令的问题，提出了一种基于控制流分析的掩码指令转换算法，TSVC标准测试集测试表明算法改进后控制流向量化识别率提升48%，平均加速比提升60%；针对控制流向量化方法的单一性问题，提出了一种利用select向量指令增强控制流向量化的phi节点优化算法，优化后TSVC测试集中的s441测试用例加速比达2.4。

（3）基本块级向量化的移植与优化。LLVM中现有的基本块级向量化方法的实现不适用于申威平台，为此本文从同构语句数约束以及向量指令代价评估两方面，进行面向申威平台的基本块级向量化的移植工作。针对基本块级向量化打包同构语句的随机性导致向量代码收益不佳的问题，提出了一种基于向量指令代价评估的打包算法，算法改进后SPEC标准测试集中的453.povray测试用例加速比提升17.1%；针对LLVM中向量化发掘能力不足的问题，提出了一种

结合迭代内与迭代间向量化发掘的混合优化方法，优化后典型示例程序的平均加速比达 2.04。

本文工作基于 LLVM-7.0.0 版本，已在申威 1621 处理器上予以实现。通过 SPEC CPU2006 与 TSVC 标准测试集的整体测试，验证了移植与优化工作的正确性。SPEC CPU2006 标准测试集的定点程序平均性能提升 2.7%，浮点程序平均性能提升 18.2%，整体平均性能提升 11.3%，矩阵乘测试用例平均加速比达 7.2，验证了移植与优化工作的有效性。本文相关成果已应用于申威平台，有效地利用处理器内的向量指令实现了 SIMD 扩展部件的性能提升。

关键词：LLVM；申威平台；自动向量化；循环级；基本块级

Abstract

Automatic vectorization technology can effectively use the underlying SIMD hardware and become an important means to improve the execution performance of scientific computing programs. At the same time, automatic vectorization optimization has also become a research hotspot of compilers. However, due to different technical routes, the SIMD extended instruction sets of various microprocessors vary greatly. The automatic vectorization needs to change the algorithm and adjust the parameters for the specific SIMD extension components to give full play to the hardware characteristics. In addition, the structure of the control flow in the loop, the insufficient number of isomorphic sentences in the basic block, and the randomness of the packaging method will hinder the ability of vectorization to explore. Proper optimization is needed to explore the potential parallelism.

The main contributions of this thesis include:

Research on LLVM's automatic vectorization technology. The LLVM open source compiler has initially implemented two automatic vectorization methods at the loop level and the basic block level. The above two methods are the current research hotspots of automatic vectorization and compilation. This article sorts out the implementation code in the loop-level and basic block-level automatic vectorization modules of the LLVM compiler from three aspects: legality analysis, vector mining, and vector code generation, and constructs LLVM automatic vectorization suitable for the Sunway platform. The process lays the foundation for the transplantation of automatic vectorization function modules.

Transplantation and optimization of loop vectorization. Due to the difference in vector length and instruction set functions, there are many differences between the automatic vectorization process for the Sunway platform and the open source LLVM. This article carries out the loop-level vectorization for the Sunway platform from the

two aspects of vector register length and vectorization information. Transplantation work. Aiming at the problem that the Sunway platform does not support mask instructions, a mask instruction conversion algorithm based on control flow analysis is proposed. TSVC standard test set test shows that after the algorithm is improved, the recognition rate of control flow vectorization has increased by 48%, and the average speedup has increased by 60 %; Aiming at the singularity of the control flow vectorization method, a phi node optimization algorithm using the select vector instruction to enhance the control flow vectorization is proposed. After optimization, the speedup ratio of the s441 test case in the TSVC test set is 2.4..

Transplantation and optimization of basic block vectorization. The implementation of the existing basic block-level vectorization method in LLVM is not suitable for the Sunway platform. For this reason, this article carries out the transplantation of the basic block-level vectorization for the Sunway platform from the two aspects of the constraint of the number of isomorphic sentences and the evaluation of the vector instruction cost. jobs. Aiming at the problem that the randomness of basic block-level vectorized packing isomorphic sentences leads to poor returns of vector codes, a packing algorithm based on vector instruction cost evaluation is proposed. After the algorithm is improved, the speedup ratio of the 453.povray test case in the SPEC standard test set is Improved by 17.1%; in response to the problem of insufficient vectorization mining capabilities in LLVM, a hybrid optimization method combining intra-iteration and inter-iteration vectorization mining was proposed. After optimization, the average speedup of typical sample programs reached 2.04.

The work of this paper is based on LLVM-7.0.0 version, which has been implemented on Sunway 1621 processor. The overall test of SPEC CPU2006 and TSVC standard test set verifies the correctness of transplantation and optimization. The average performance of fixed-point programs in the SPEC CPU2006 standard test set has increased by 2.7%, the average performance of floating-point programs has increased by 18.2, and the overall average performance has increased by 11.3%. The average speedup of matrix multiplication test cases has reached 7.2, which verifies the effectiveness of porting and optimization. The relevant results of this article have been

applied to the Sunway platform, effectively using the vector instructions in the processor to achieve the performance improvement of the SIMD expansion components.

Key words：LLVM；Auto-vectorization；Sunway；Loop-level；Basic block-level

目录

图和附表清单

1 绪论

1.1 课题研究背景

1.1.1 高性能计算机的发展

高性能计算的开发可用于解决社会复杂的全球性问题的科学应用程序，例如工业仿真、气候气象、海洋模拟、生物医药、疫苗发现、改善食品生产和发展绿色能源等。随着社会的发展对计算能力的需求越来越大，高性能计算技术得到了迅猛发展[1]。

高效能的超级计算系统，是衡量一个国家科技核心竞争力和综合国力的重要标志。在 2019 年前，全国共建成天津、济南、长沙、深圳、广州、无锡六家国家超级计算中心。第七家国家超级计算中心国家超级计算郑州中心，已于 2020 年 11 月通过验收并正式纳入序列管理，是河南省首个国家级重大科研基础设施。随着国家专项投入的持续推动，我国在高性能计算机研制方面取得了丰硕成果，如表 1.1 所示为 2020 年中国高性能计算排名。

表 1.1 2020 年中国高性能计算排名

序号	系统/型号	研制单位	安装地点	Linpack（Tflops）	峰值（Tflops）
1	神威太湖之光 40960*Sumway SW26010 260C 1.45GHz 自主网络	国家并行计算机工程技术研究中心	国家超级计算无锡中心	93015	125436
2	天河二号升级系统，TH-IVB-MTX Cluster + 35584*Intel Xeon E5-2692v2 12C 2.2GHZ + 35584-Matrix-2000, TH Express-2	国防科大	国家超级计算广州中心	61445	100679
3	北京超级云计算中心 A 分区，6000*AMD EPYC 7452 32C 2.350GHz,FDR	DELL	北京超级云计算中心	3743	7035
4	内蒙古高性能计算公共服务平台，3200*Intel Xeon Gold 6254 18C 3.1GHz,EDR	方同	内蒙古自治区和林格尔新区	3185	5345

神威•太湖之光超级计算机于 2016 年世界高性能计算机排名中位列首位，我国成为继美、日之后第三个使用自主 CPU 构建千万亿次超级计算机的国家，标志着我国高性能计算机已处于世界领先水平。申威系列处理器是我国自主研发、面向高性能计算的通用微处理器，其计算速度不断提升，成为推动许多学科领域飞速发展的重要力量。

随着半导体制造工艺的不断提高，并行计算处理的规模在持续扩大，但计算能力依然保持“摩尔定律”朝前发展[2]。在电子器件的更新已不能满足日益增长的计算需求下，体系结构开始引领高性能计算机的发展。此情况下微型的向量并行部件 SIMD（Single Instruction Multiple Data）扩展[3]迅速发展起来，被广泛地应用到各种科学计算类程序的加速优化中。

1.1.2 SIMD 向量化方法

SIMD 扩展部件是一种单指令多数据扩展技术，通过一条指令对多个数据同时进行处理[4]。当前，通过有效地利用底层 SIMD 硬件提升程序性能，是程序员以及处理器厂商研究和开发的重点。

SIMD 扩展部件非常适合处理数据量大、数据之间依赖少且计算步骤比较多的应用程序[5]，通过使用一条指令实现对多个数据地处理，将内存中连续的数据一次性地加载到向量寄存器中进行向量操作，如图 1.1 所示。SIMD 扩展对具有访存模式重复、局部数据上重复操作特征的程序性能提升显著，例如多媒体技术中的离散小波变换、快速傅里叶变换、几何结构处理等多媒体应用都有明显的优化效果。

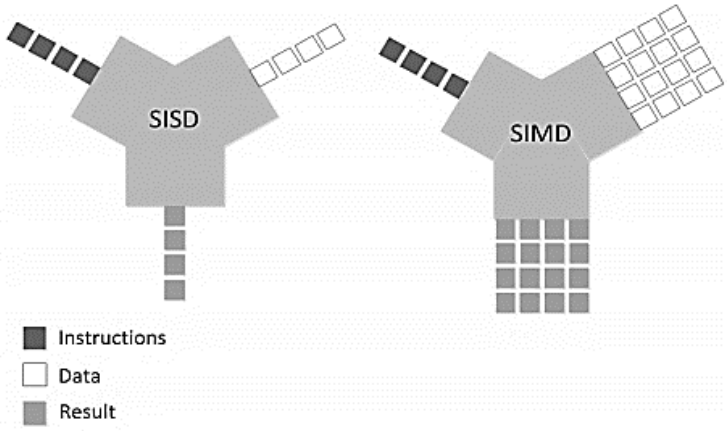


图 1.1 SISD 与 SIMD 比较图

随着 SIMD 扩展部件在高性能计算机中的广泛应用，向量寄存器的长度变得越来越长，Intel 的 SIMD 扩展指令集从最初的 MMX，经历了 SSE、AVX，到现在的 AVX512，数据宽度从最初的 64 位扩展到了 512 位[6]，江南计算所的申威系列处理器由最初支持的 128 位向量长度扩展到了 256 位，下一步将继续增加到 512 位[7]。与此同时 SIMD 扩展部件支持逻辑运算、算数运算以及数学函数等向量计算[8]，支持的向量指令集也变得越来越丰富。向量寄存器硬件耗能相对低，合理地利用向量寄存器对计算密集型程序进行向量化可以获得较好的性能提升，成为程序加速的重要手段之一。

SIMD 扩展指令发掘方法有两种，手工向量化与自动向量化。手工向量化对于开发人员来说操作较为繁琐，程序员需要通过在程序中嵌入 Intrinsic 函数、C++类库扩展、机器汇编语言、SIMD 库等相关指令来使用 SIMD 扩展部件[2]。Leißa 在 C 语言基础上设计了一种 SIMD 扩展语言[9]，解决了一些移植性问题；Jonathan Parri 描述了可重定位的本地 SIMD 库，通过使用 API 中规定的 Java 方法直接访问 SIMD 内联函数[10]；Pierre Esterie 设计了一个简化的 SIMD 编程 C++模块库[11]。

自动向量化可以使程序员摆脱复杂繁琐的底层细节的困扰，更加便捷地提升程序的运行性能，已逐渐发展成为 SIMD 向量化的主要途径。目前，面向 SIMD 扩展部件的自动向量化编译已成为程序向量化变换的主要方式[5]。

1.1.3 自动向量化方法

目前，实现自动向量化的方式有主要有两种：面向循环的传统向量化、面向基本块的超字并行（Superword Level Parallelism，SLP）向量化。

面向循环的传统向量化技术的基本思想是利用强连通分量 Tarjan 算法[12]在循环语句数据依赖图中搜索依赖环，把不存在真依赖的语句作为向量执行的候选语句，使用与标量指令功能相同的向量指令进行替换，完成串行程序自动向量化过程。

传统意义中外层循环向量化是通过将外层循环与最内层循环互换，然后在最内层位置进行向量化来执行的，D.Nuzman[13]在 GCC 编译器中实现了外层循环向量化，与最内层循环矢量化相比，可以提供显着的性能改进。针对向量化过程中的控制流向量化发掘，Lokuciejewski 等人[14]提出了一种新型的 IF 外提方法，是基于 WCET 的优化驱动解决了之前由于 IF 外提产生的代码膨胀问题，但缺点

在于 WCET 信息不能得到及时的更新。Trifunovic 等人[15]研究了多面体框架下循环转换与向量化之间的相互作用，对不同的循环转换与向量化策略对性能的影响进行建模，这种预测模型考虑了不同向量化方案的后续影响，同时更好地应用了多种循环变换技术。

面向基本块的超字并行技术来源于 Larsen 等人[16]提出的基本块内发掘向量化的超字并行算法，通过对基本块中同构语句打包进行向量化。超字并行向量化方法的性能提高很大程度依赖于向量指令的选择策略，基本块内指令级别的操作增加了选择策略的复杂度[2]。

V. Porpodas[17]提出了 Super Graph-SLP（SG-SLP）方法，是一种改进的 SLP 向量化算法，克服了现有算法的局限性。SG-SLP 可以在较大的区域上运行，使它可以到达并成功向量化之前无法访问的代码。此外，因为该算法可以更加全面地查看程序代码，有助于消除代价计算的不准确性。Rocha 等人[18]基于 SLP 算法进行了改进优化，针对包含可交换运算符（例如加法）及其对应的逆元素（减法）的表达式进行了优化，该算法利用可交换运算符及其逆元素的代数性质来实现其他转换，从而提升 SLP 向量化性能。Jaewook Shin 等人[19]实现了含有控制流的 SLP 向量化，通过在指令中添加谓词来完成转换。

目前自动向量化面临的主要问题：

（1）可移植性差。由于技术路线以及专利知识产权等现实问题，处理器之间的 SIMD 扩展指令集差异很大，自动向量化需要针对具体的 SIMD 扩展部件进行算法的改进、参数的调整，才能充分发挥硬件特性[20]，造成了 SIMD 自动向量化的可移植性较差的局面。

（2）控制流向量化能力差。SIMD 自动向量化通过增强向量发掘的分析能力 [21]，尽可能多的利用数据流中的并行性来提升程序运行的性能。程序中控制流的出现造成了数据流分析的困难[22][23]，是 SIMD 自动向量化研究的一个难点。一些处理器针对控制流向量化提供了非连续访存的向量指令，如 Intel AVX2 指令集支持掩码访存操作，但大多数微处理器并不支持此类向量指令。

（3）缺少有效的代价评估模型。现代体系结构的编译器大多支持循环级与基本块级的自动向量化功能，标量程序转换成向量代码的能力逐步提升，但是由于缺少精准的代价评估模型，向量化阶段无法进行有效的向量决策。

1.2 LLVM 编译系统

1.2.1 LLVM 简介

LLVM 是对任意编程语言提供的一种基于 SSA（Static Single Assignment）的静态与动态编译的现代编译技术。与其他主流编译器相比，LLVM 的主要优势在于它的多功能性、可重用性和灵活性[24]。编译器的高度模块化特点，使源代码结构清晰易于开发者阅读理解与分析调试，组件易于以库的形式抽取并用于其它领域。高级语言被 LLVM 前端解析为平台无关的中间表示（Intermediate Representation，IR）[25]，使编译器能够在编译、链接以及代码生成的各个阶段忽略语言特性，进行全面有效的优化和分析[26]。

LLVM 创始人 Chris Lattner 及他的团队于 2005 年加入 Apple 公司，为其电脑系列开发应用程序系统，现今 LLVM 已发展成为 Mac OS X 及 iOS 开发工具的一部分。背靠大树好乘凉，在 Apple 的赞助下 LLVM 得到了迅速的发展，Android 系统使用了新的运行时 ART（Android Runtime）后，LLVM 在 Android 系统中的使用比重得到了巨大的提升。LLVM 从学术界的懵懂，迅速成长为具有工业级稳定性和高效性的编译器[27]。

将研究重点放在编译器算法的研究上，提高软硬件协同工作的性能，是今后未来芯片厂商所要关注的一个重点，这也是高通、英特尔、ARM、华为等芯片公司长期占据着 LLVM 基金会赞助商榜首的原因。

1.2.2 LLVM 编译框架

LLVM 支持多后端的交叉编译器，基于传统的三段式设计：编译器前端、优化器和编译器后端，如图 1.2 所示。

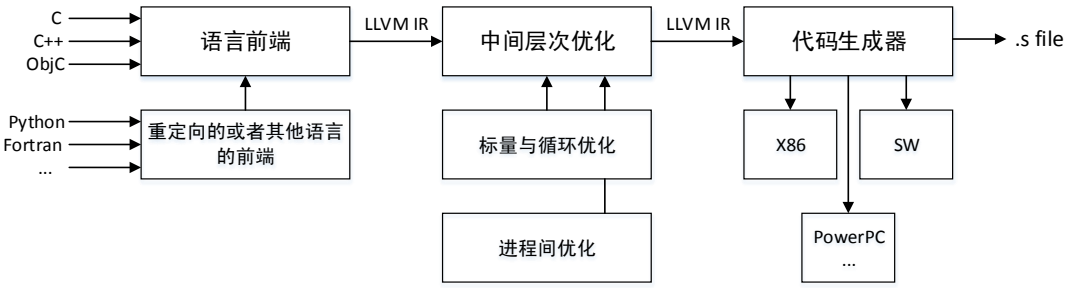


图 1.2 LLVM 编译系统

编译器前端把源代码转化为抽象语法树（Abstract Syntax Tree，AST），将高级语言解析为统一的中间表示。通用优化器针对中间表示进行分析、优化、转

换处理，常用的循环优化发生在此阶段。编译器的后端也叫代码产生器，负责把优化后的代码转换为处理器支持的汇编指令集，最后进行机器指令（Machine Code，MC）的生成[24]。

LLVM 为编译器开发者提供了一系列转换库，包括运行库、代码生成库和分析库等[28]，实现了开发过程所需的基本功能。针对各阶段不同的优化，提供了一套完备的转换工具，利用丰富的工具集可以有效的提高开发编译器的效率。编译流程中各个工具的使用方法如图 1.3 所示。

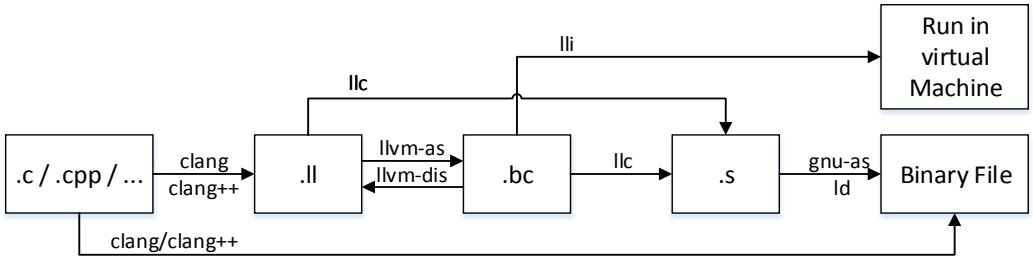


图 1.3　LLVM 编译流程

LLVM 基于统一的中间表示来实现优化遍，中间表示采用静态单赋值形式，该形式的虚拟指令集能够高效的表示高级语言，具有灵活性好、类型安全、底层操作等特点[29][30]。例如图 1.4 所示，当同一变量出现多次赋值时，通过 SSA 变量重命名的方式加以区分，可以避免出现多次定义的情况。

```
1.  x = 9;                1.  x1 = 9;
2.  y = x;                2.  y1 = x1;
3.  x = x + 1;            3.  x2 = x1 + 1;
4.  y = x + 7;            4.  y2 = x2 + 7;
      (a)                       (b)
```

图 1.4　普通代码与 SSA 形式的代码对比

LLVM 已初步实现了循环级与基本块级的自动向量化方法，循环级向量化方法是当前获得程序自动向量化加速的主要手段，基本块向量化算法实现较为复杂但向量发掘能力强，两者成为当前编译开发团队研究的热点。申威系列处理器包含了丰富的向量运算指令和完善的向量重组指令，为程序的自动向量化优化提供了良好的硬件基础。

1.3 课题来源

江南计算技术研究所委托开发项目“面向某国产平台 LLVM 编译器的程序变换和优化系统”。

1.4 课题研究内容

申威处理器是我国自主研发、面向高性能计算的通用微处理器，该处理器针对计算密集性程序可以进行准确高效的计算。本课题研究内容基于支持申威后端的 LLVM 编译器，该基础编译器是针对 C、C++和 Fortran 开发的优化编译器。LLVM 编译器中自动向量化部分的算法以及向量指令的应用方面与国产申威处理器存在差异，所以本课题面向申威处理器进行 LLVM 编译器的自动向量化功能的移植与优化，本文研究内容分为以下几个部分：

（1）自动向量化的移植研究。自动向量化包含循环级向量化与基本块级向量化方法，通过向量化可以使程序运行得到有效的加速效果。各大主流编译器为了使自己能够更加广泛的应用到各种处理器平台，优化部分往往采用了更加保守的方法。制作工艺以及使用途径的不同造成了处理器之间指令集的差异，申威处理器由我国自主研发，自动向量化实现方法与开源 LLVM 存在差异。本文考虑到指令集特征问题，从向量寄存器描述、完善向量化信息、向量化因子约束等方面进行自动向量化的研究与移植工作。

（2）控制流向量化方法的优化。循环向量化阶段通过一条 SIMD 向量指令完成内存连续的多条数据的操作，控制流的出现导致了不连续访存，阻碍了向量化的发掘能力。通过指令转换，将控制依赖转换为数据依赖是当前处理控制流向量化的最佳方式。目前 LLVM 只针对某些简单格式的控制流完成了指令转换，大部分情况下需要调用后端相关的额掩码访存指令完成向量转换，但是申威指令集并不支持掩码访存指令，导致失去向量化机会。针对以上问题，本文第二个研究内容是基于申威指令集对包含控制流结构的循环进行向量化优化。

（3）代价模型的完善。向量化收益来源于减少的指令周期数，代价模型可以评估自动向量化的加速效果，循环级与基本块级向量化皆有自己专属的代价评估系统。循环级向量化在完善控制流向量化算法后，需要更新代价模型的计算方式。基本块级向量化算法中最为复杂的是同构语句的打包，打包的指令不同带来的收益也大不相同。本文利用代价模型验证自动向量化移植的有效性，在循环

级向量化中完善代价计算，在基本块级向量化的同构语句打包过程中引入收益分析，完善精准的代价模型指导，提升向量化加速效果。

（4）混合向量化方法的优化。LLVM 中基本块级向量化针对迭代内的基本块进行向量发掘，根据一条向量指令处理的标量数据个数进行同构语句的打包。当同构语句数大于或者小于向量化因子时，会生成不够简洁的向量指令或失去向量化机会。针对此类问题，本文将循环展开与向量化相结合，使基本块级向量化不仅仅局限于迭代内的向量发掘，而是结合迭代间与迭代内，增加向量化机会的同时生成的向量代码也更加高效简洁。

1.5 论文组织结构

本文以支持申威处理器后端的 LLVM 编译器为基础，进行自动向量化的移植与优化工作。针对循环级与基本块级向量化进行后端 SIMD 向量寄存器以及向量化信息的描述，结合向量化特征进行适配性的修改以完成基本的移植工作。在此基础上针对两种向量化方法分别提出了一些优化算法，提升自动向量化性能。全文共分为六章：

第一章绪论，对课题研究背景、LLVM 编译系统、课题来源以及研究内容等方面进行了简单的阐述。课题研究背景中简述了我国高性能计算机的发展与应用，对 SIMD 扩展部件以及 SIMD 自动向量化方法进行简单描述，提出了目前自动向量化研究中面临的问题。另外基于课题研究，对 LLVM 编译系统进行了简单描述，介绍了编译器的整体框架。

第二章研究了 LLVM 编译器的自动向量化技术。分析了向量化之前标量优化中的预分析，分析结果可在自动向量化阶段直接获取以减少循环分析的工作量。最后，从循环级与基本块级向量化两方面介绍了当前 LLVM 编译器的自动向量化流程。

第三章研究了循环级向量化的移植与优化。完善基础的向量寄存器内容的描述，并结合指令延迟对向量化信息进行补充，完成循环级向量化的移植工作。在此基础上针对控制流向量化进行算法优化，利用掩码指令转换算法将控制依赖转换为数据依赖，并在简化控制流图阶段控制指令下沉，对 phi 指令生成算法进行改进，提升控制流向量化能力。

第四章研究了基本块级向量化的移植与优化。除去与循环级向量化共用的向

量寄存器信息，完善基本块级向量化特有的同构语句数约束以及数据重组的指令代价计算。针对打包策略的复杂性与不确定性，在打包过程中引入收益分析，完善代价模型的同时优化了打包策略。最后将循环展开与基本块级向量化相结合，设计了一种混合向量化方法，增强向量发掘能力提升向量代码生成质量。

第五章对本文的研究内容进行了实验测试与结果分析。首先介绍了实验使用的测试集与环境，进行正确性测试验证移植的有效性。使用具有不同特征的程序进行性能测试，最后利用测试集对本文工作进行整体的性能测试，验证优化的有效性。

第六章总结了本文的工作，对下一步研究方向进行计划与展望。

2 基于 LLVM 的自动向量化框架

2.1 研究动机

LLVM 支持循环级与基本块级的自动向量化功能，自动向量化编译可概括为三个阶段：向量发掘、向量优化和向量代码生成。

向量发掘过程可以从面向循环以及面向基本块两方面进行。循环级自动向量化方法可以有效的发掘 SIMD 扩展部件特性，是提升程序性能的最常用、最直观的自动向量化方法[31]。与循环级的向量并行发掘方法不同，基本块级向量化方法主要用于发掘基本块内的并行性。

向量优化是指对编译器构造的标量代码进行向量代码转化的过程。标量代码中连续的向量访存不仅可以提高向量访存指令的效率，也可以提高向量寄存器中有效数据的比率，但控制流情况的出现引发了不连续的访存操作[32]，带来额外的开销降低导致向量化收益降低。因此，如何在向量化过程中以更灵活的访存模式解决不连续访存问题值得深入研究。

向量代码生成是指向量化优化后生成的中间表示以及降级到目标平台向量指令的过程。当处理器支持新指令时，向量代码生成过程将面临调整难度大、可移植性差等问题。随着扩展部件的体系结构和指令集的不断改进，向量代码生成需面向特定体系结构和指令集进行指令优化。

LLVM 支持循环级向量化与基本块级向量化，本章主要介绍了 LLVM 中两种向量化方法实现的过程，对优化方式进行了概括介绍，为面向申威的自动向量化的移植与优化工作奠定理论基础。

2.2 向量化预分析

LLVM 在向量化优化前会经过很多分析遍的处理，这样转换遍才能使用分析遍的结果来完成相应的转换。别名分析与标量演化分析过程势必会增加程序的编译时间，所以在向量化之前会对循环进行处理，舍弃那些不具备向量化潜质的循环以提升编译效率。

别名分析技术用于确定两个指针是否可以指向内存中的同一个对象[33]，当多于两个的表达式在同一程序引用相同的存储位置时将存在指针别名问题。别名

分析的结果往往是其它程序分析与变换的基础，指针别名的精度影响了自动向量化、安全性分析以及多线程分析等多个优化应用的性能[34] [35]。

目前有许多不同的别名分析算法和分类方法，指针别名分析方法技术主要围绕提高别名分析的精度而各显其独特优势析[36]。指针别名分为三种：指示两个指针始终指向同一个对象；可能指向同一个对象；已知永远不会指向相同的对象。前两种情况无需处理，对于不确定是不是别名的指针引用添加运行检测代码，即在运行时检测两个指针对应的内存块是否有重叠。

在实际程序中可能存在大量的指针引用，每个指针对之间都可能存在别名，插入的运行时检测代码较多，可能会抵消掉向量化的性能提升，如图 2.1 所示示例表示别针别名的运行时检测。

```
//源程序：存在别名程序
Func(int *A, int *B){        //A和B可能存在别名
 for (int i = 0; i < 16; i++) {
   a[i] = b[i];
}

//实施运行时检查以及循环多版本的程序，可以进行向量化
Func(int *A, int *B){
      if(|A-B|>=X)          //运行时检测
        for (int i = 0; i < 16; i++)       //A、B非别名
          a[i] = b[i] ;
      else
        for (int i = 0; i < 16; i++)       //A、B是别名
          a[i] = b[i] ;
}
```

图 2.1　指针别名运行时检查代码示例

标量演化（Scalar Evolution，SCEV）可用于分析、重写和识别循环中的标量表达式。在循环的迭代过程中利用抽象的符号表达标量变量值的变化，用于识别一般的归纳变量和归约变量[37]。标量演化分析应用于归约变量简化、循环向量化、基本块级向量化、依赖分析等优化过程。

标量演化的理论基础是将标量使用递推链代数进行表示，然后使用一些符号重写和折叠该递推表达式[38]。图 2.2 所示程序为循环中标量示例代码，表现了循环中的标量在每次迭代过程中的变化。

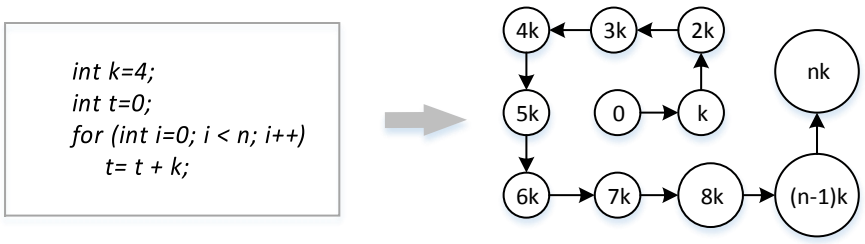


图 2.2 标量在每次迭代过程中的变化

2.3 循环级向量化框架

循环级向量化在迭代间发掘向量化机会进行向量代码生成[39]，LLVM 循环向量化主要分为三个步骤：向量化的合法性分析；循环的收益分析；标量语句结构到向量语句结构的转换。

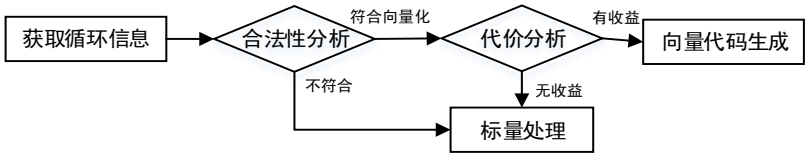


图 2.3 循环级向量化流程

2.3.1 合法性分析

合法性分析首先检查循环是否为嵌套循环，排除不能向量化的循环例如多出口多回边结构。对于包含控制流的循环，检查循环结构后收集分支掩码以用于后续向量代码生成。对于 phi 指令以及调用指令，分析其是否符合向量化格式要求。最后调用循环访存信息，分析访存指令是否具有阻止向量化的依赖关系。

（1）检查循环格式。循环向量化认为所有的循环都需要进行格式简化，以保证循环具有一个 pre-header 块、一个 latch 块、出口有唯一的后继块、退出块必须以条件转移结束等条件约束。

循环向量化中针对循环结构的名词说明：

basic block：基本块由一系列必须按顺序执行的指令组成。任何可以影响下一个执行的指令的操作，即任何分支指令都只能发生在块的末尾。

header：循环中不受任何其他循环块支配的基本块称为 header 块。

pre-header：循环的 pre-header 是一个在循环单位之外的基本块，具有到循环

header 的无条件分支。

back edge ：循环中的基本块到循环 header 块的回边。

exiting block ：循环的退出块位于循环内部，但是它有一个延伸到循环外部的边。

latch：循环中基本块具有唯一的回边，此基本块称为 latch 块。

在合法性分析的格式要求下，控制流结构向量化时循环也必须拥有一个规范的格式：具有合法的 pre-header、单回边（back edge）、单退出块（exiting block），且需要满足 exiting 块与 latch 块为同一个基本块。针对控制流结构进行合法性分析的格式检查要求，以图 2.4 控制流示例为例进行分析说明。

```
//控制流示例1：
for (int i = 0; i < 1024; i++) {
  if (x[i] > 0)
    a[i] = b[i] + 1;
  else
    a[i] = c[i] - 1;
}
```

```
//控制流示例2：
for (int i = 0; i < 1024; i++) {
  if (x[i] > 0) {
    a[i] = b[i] + 1;
    break;
  }
  else
    a[i] = c[i] - 1;
}
```

图 2.4　控制流程序示例

由于循环向量化是基于中间表示层进行的优化，这里取控制流示例 1 的中间代码的控制流图进行说明分析，如图 2.5 所示。

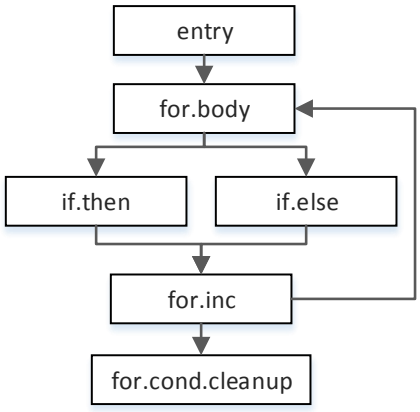


图 2.5　控制流示例 1 的控制流图结构

控制流图结构中的 entry 基本块为循环的 pre-header 块，循环体 for.body 为

循环的 header 块，同时 for.inc 满足了既是循环的 exiting 块又是 latch 块的条件，符合包含控制流结构的循环进行向量化的规范格式。示例 2 循环结构中具有两个 exiting 块，循环为多出口情况不符合向量化的循环格式，合法性分析无法通过阻止了一步的向量化发掘。

（2）控制流向量化检查。当循环中包含控制流结构时，合法性检查变得更为复杂。首先，需要识别循环中的指针指令，收集每次执行循环迭代时都可以在循环体范围内进行解引用的安全指针。安全指针所指向的内存可以在循环 header 块中无条件地解引用，而不会引入新的错误，例如非分支基本块中的访存操作中使用的指针指令。然后，收集需要预测执行的基本块，目的是检查循环内是否包含控制流结构。循环中不能主导 latch 块的基本块即为需要预测执行的基本块，例如图 2.5 控制流示例 1 中的 if/else 基本块。最后，当识别出循环中包含控制流结构时，判断分支基本块是否可以进行预测执行，并检查是否有可以作为操作数捕获的常量表达式。同时，根据分支基本块执行的顺序，收集掩码访存指令所需的掩码。

（3）指令加宽转换分析。对于一些特殊的指令，例如 phi 指令与调用指令是否能够符合向量化加宽要求需要进行额外检查。目前，循环向量化中控制流向量代码的生成皆是依靠特殊结构的 phi 指令转换而来。合法性检查时需要首先确认 phi 指令的类型是否合法，只有位于 header 块之外的 phi 指令才有机会转换成控制流向量化中所需的 select 指令。由于控制流向量化的特殊性，还需额外检查 phi 指令的输入值个数。

（4）访存指令依赖分析。调用循环访存分析来检查依赖关系，依赖关系是确定指令执行顺序的一种偏序关系，根据约束不同可分为数据依赖关系（Data Dependence）和控制依赖关系（Control Dependence），是程序变换前后程序语义一致性的判断标准，是串行程序并行化技术的理论基础[40]。

依赖约束了程序变换，依赖关系分析技术为数组索引的下标表达式建立依赖方程组，通过求该方程组的解来确定是否存在数据依赖关系[12]。按照存-取顺序的表示，在程序中存在三种类型的依赖：

1) 真依赖

第一条语对单元存入数据而后由第二条语句读出，描述为 S1 δ S2。需保证第二条语句接收到正确的第一条语句计算的值，此类型的依赖也称为流依赖。

2) 反依赖

第一条语句对单元读出数据而后由第二条语句存入，描述为 S1 δ^{-1} S2。本质上此种依赖的出现阻碍程序变换，例如进行语句交换后产生了新的真依赖。

3) 输出依赖

第一条语句 S1 与第二条语句 S2 对同一单元写入数据，描述为 S1 δ^{0} S2，此类依赖同样会阻碍程序变换。

将依赖概念扩展到循环，循环依赖分为循环无关依赖和循环携带依赖，循环无关依赖决定了循环迭代内的语句执行顺序，循环迭代的执行顺序则由循环携带依赖决定[41]。所有的依赖关系中只有真依赖影响向量化执行的正确性，所以在面向 SIMD 扩展部件的向量化中，只有真依赖环上的依赖距离大于向量化因子时才可进行向量化[12]。

2.3.2 向量代码生成

为生成高效的向量代码，循环向量化调用代价模型进行不同向量化因子下的收益分析，选出最优向量化因子后与标量收益进行对比，选出最优向量化决策进行向量代码生成，具体收益评估以及改进方法见本文 3.4 章节。

结合循环体 a[i]=b[i]+1 且循环上界为 1023 向量化因子为 4 的程序为例，分析向量代码生成过程，原始标量循环的控制流图如图 2.6 所示，for.body 为标量循环体。

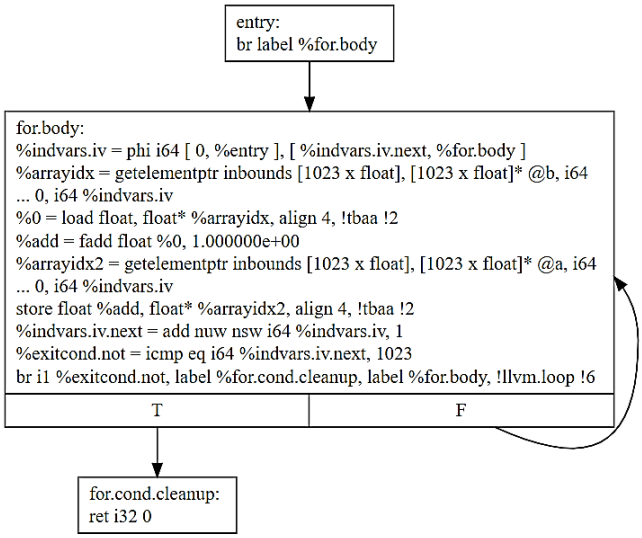


图 2.6 标量循环的控制流图

循环向量化中向量代码生成主要分为以下三步：

（1）在标量循环体前创建一个新的空循环，使其成为向量指令的载体。添加的新循环必须遵循循环体的规范格式，也就是除了向量循环体外还需相应的 pre-header 块与 latch/exiting 块。

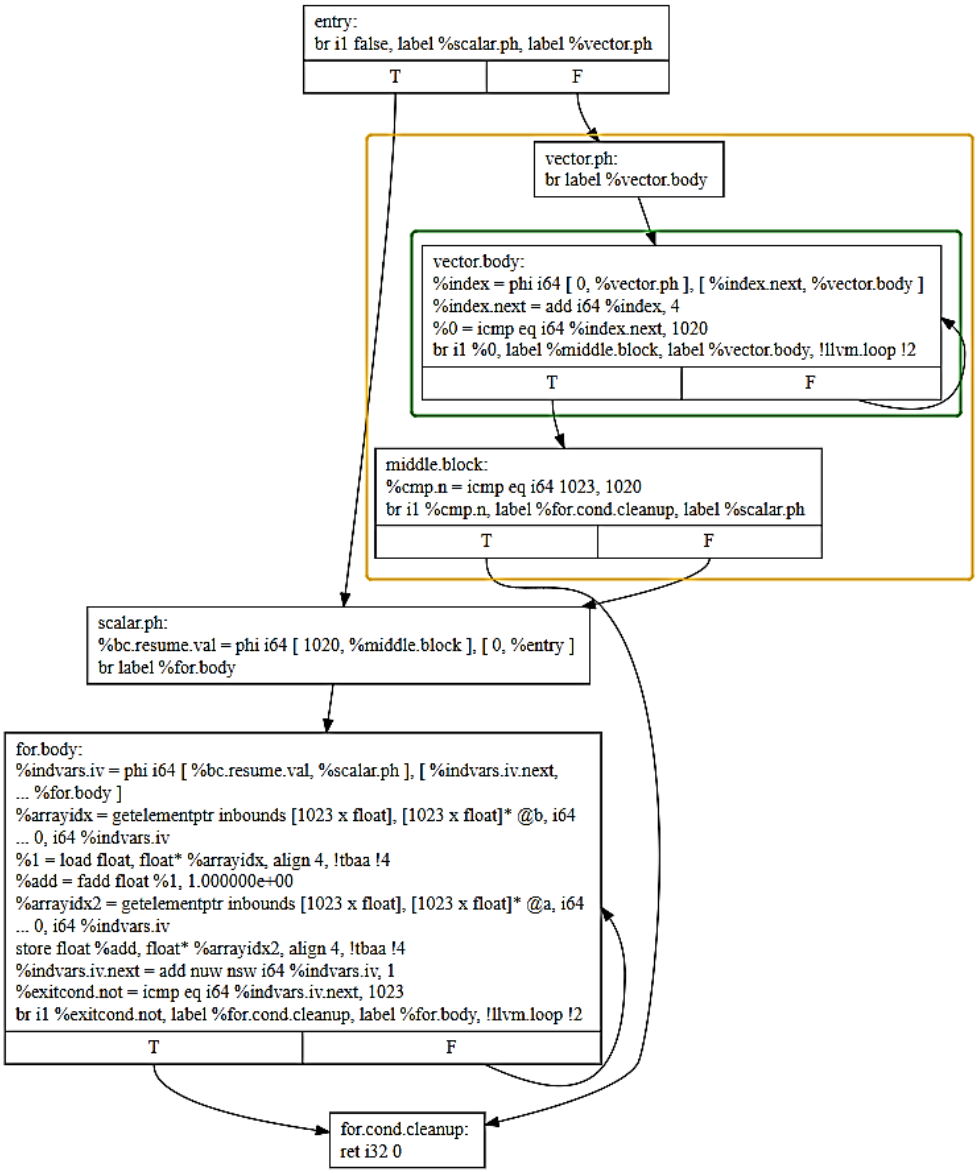


图 2.7　创建新循环的控制流图表示

构建新的循环时，需要注意归纳变量的处理。向量循环体循环上界的计算：循环上界=循环上界-循环上界%（VF*UF），其中 VF 为向量化因子 UF 为循环展开数。当循环上界无法整除向量化因子时，剩余“循环上界减去向量循环体的

循环上界”次迭代由旧循环作为标量尾循环处理。新循环结构如图 2.7 黄色方框所示，绿色方框为向量指令的实际载体，剩余 3 条标量指令由旧循环处理。

（2）标量循环中不同类型的指令分别调用专有的转换函数进行向量指令生成，生成的向量指令逐一添加到新的循环体中。标量指令转换为含有 SIMD 指令的向量化代码，需要考虑目标平台支持哪些 SIMD 向量扩展指令，否则会出现编译错问题。

基本逻辑运算与内存连续的访存指令的加宽方式较为简单，保持指令操作码不变，将标量数据类型替换为向量数据类型，同时保持操作数的统一即可。如图 2.8 所示为向量语句填充后的指令形式：

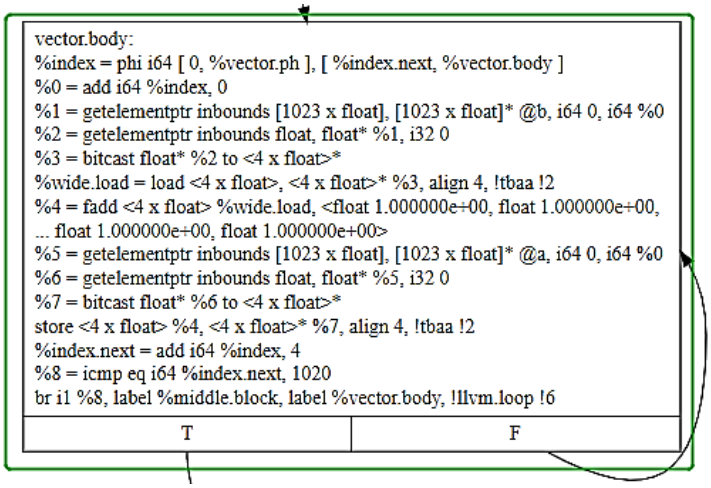


图 2.8　向量指令填充的新循环

不连续访存指令的加宽方式更为复杂。分析向量化过程中是否需要使用聚合指令、内存顺序是否为反序、后端是否支持掩码访存指令；获取合法性分析过程中获取的分支执行的掩码；进行掩码内建函数的调用以生成向量访存指令。但是申威处理器并不支持掩码访存指令，这导致自动向量化时失去控制流向量化机会。

（3）修复向量代码，更新分析信息。此步骤主要是修复归纳变量，将新循环体的相关信息取代旧循环体，设置以及更新向量循环与标量尾循环的分配关系。

2.4　基本块级向量化框架

与传统循环级的向量并行发掘方法不同，基本块级向量化发掘方法主要是在

基本块内寻找同构语句，发掘基本块内指令的并行机会，基本块向量化流程如图 2.9 所示。

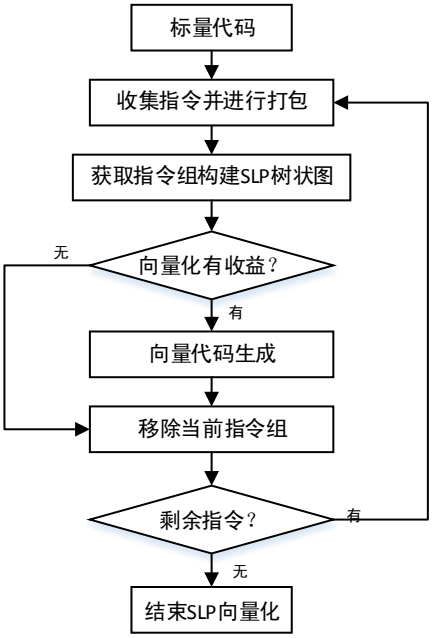


图 2.9　基本块级向量化流程

2.4.1　向量化发掘算法

超字并行的向量化方法用于发掘基本块内的并行性[16]。以存储指令或者归约格式的 phi 指令为根节点遍历语法树结构，利用相邻地址的存储指令作为候选语句，通过使用-定义链自下而上进行打包[42][43]，将同构语句转换成向量指令，其中每个节点都包含可并行处理的相同数目的同构语句。包（pack）指的是同构语句的集合，将多个同构语句组成包的过程叫做打包[1]。

循环中的每一个操作都可以被目标平台以向量形式支持时，进行语法树的代价分析。在有收益情况下构建向量化树，从上到下扫描基本块的所有语句，在需要向量转化的标量指令前插入向量指令。最后，移至下一组指令重复以上分析，直至完成基本块内所有指令的向量发掘。

基本块级向量化算法以抽象语法树为基础，满足同构语句数大于向量因子的条件后对语法树按照自底向上的顺序遍历，以图 2.10 程序为例。

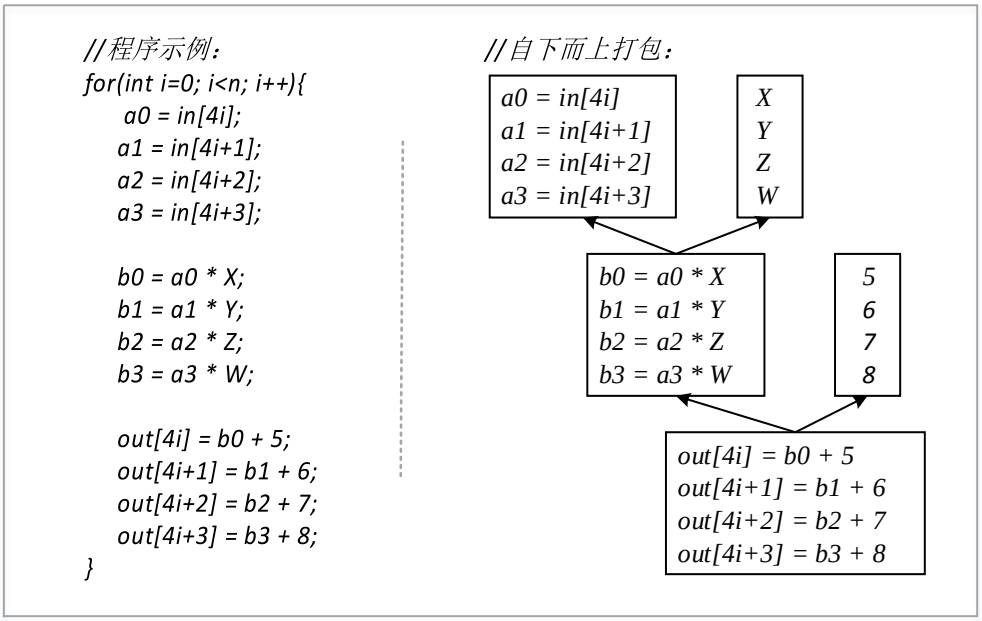


图 2.10　基本块级向量化打包方式

（1）以存储指令为根节点，按照地址相邻的原则生成初始包。地址临界的内存引用具有天然的 pack 属性，一次访存便可实现对多个内存的操作，因此启发式算法首先寻找地址连续的访存指令来生成初始 pack[16]。以内存首地址为基准采取不同的偏移分析同构语句包，内存连续的语句进行向量化打包可以有效提升性能。利用相邻地址的数据引用算法，生成{out[4i] = b0 + 5;out[4i+1] = b1 + 6;out[4i+2] = b2 + 7;out[4i+3] = b3 + 8;}初始 pack。

（2）按照定义-使用链扩展包。生成以存储指令为根节点的初始化同构语句列表后，根据语法树自下而上搜寻[44]，向列表中添加更多的同构语句。通过定义-使用链找到候选语句，创建一个新组并将其添加到 pack 列表中[45]。同样的方法扩展出来两个新 pack 为{ b0 = a0 * X;b1 = a1 * Y;b2 = a2 * Z;b3 = a3 * W;}和{5; 6; 7; 8;}。重复此步骤自下而上进行扩展搜寻，当节点内所有语句为循环不变量或 load 语句时该节点为叶子节点，当节点内所有语句为循环变量或无依赖关系的非 load 指令时此节点为子节点，其他情况下将终止并删除树。

（3）按照依赖关系调度包。调用 calculateDependencies 函数检查数据依赖关系后对同构语句包进行调度，依赖关系分析要考虑依赖距离与向量并行宽度的关系，若循环中“依赖距离>=向量化因子”，则依赖不影响循环的向量并行性。最后生成的打包结果如图 2.9 右侧结构图所示。

2.4.2 向量代码生成

与循环向量化相同，基本块级向量化在向量代码生成前调用代价模型判断向量化是否有收益。不同的处理器具有不同指令延迟节拍，但不是所有向量代码都比标量代码整体收益好，具体基本块级代价模型以及优化见本文 4.3 章节。

基本块向量化是并不是以循环为单位进行向量代码生成，而是以语法树为单位进行向量化代码生成。主要分为以下两步骤：

（1）在同构语句包前插入向量指令。

```
…%arrayidx27 = getelementptr inbounds [32000 x float], [32000 x float]* @a, i64 0, i64 %2
store float %add24, float* %arrayidx27, align 4, !tbaa !2
%10 = bitcast float* %arrayidx6 to <4 x float>*
store <4 x float> %9, <4 x float>* %10, align 16, !tbaa !2
store float %add17, float* %arrayidx20, align 8, !tbaa !2
store float %add10, float* %arrayidx13, align 4, !tbaa !2
store float %add, float* %arrayidx6, align 16, !tbaa !2 …
```

（2）擦除标量指令并更新前后语句关系。

```
…%arrayidx27 = getelementptr inbounds [32000 x float], [32000 x float]* @a, i64 0, i64 %2
%6 = bitcast float* %arrayidx6 to <4 x float>*
store <4 x float> %5, <4 x float>* %6, align 16, !tbaa !2 …
```

相较于循环向量化，基本块级向量代码生成方式较为简单。它以同构语句包中第一条指令的位置为基准插入向量指令，不需要额外创建循环或者函数作为向量指令的载体，完成后将原有标量指令进行死代码删除，转换示例如图 2.11 所示。

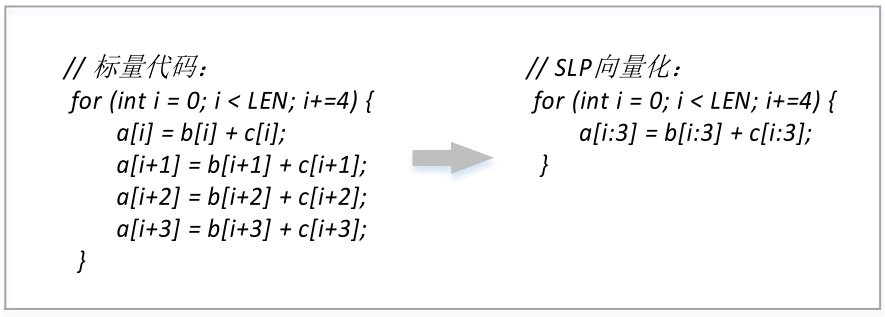


图 2.11 基本块级向量化转换示例

2.5 本章小结

本章节主要从向量化预分析、向量化发掘、向量代码生成等方面，研究了 LLVM 支持的循环级与基本块级的自动向量化方法。在循环级向量化过程中针对控制流结构，引用示例说明其向量化过程中合法性的要求，通过中间表示层的控制流图变换分析了向量代码生成的步骤，并针对循环中的数据依赖与控制依赖进行分析研究，提出了当前控制流向量化的缺点，为第三章面向循环级向量化的移植与优化奠定基础。基本块级向量化过程中同构语句的打包方式较为复杂且存在随机性，本章节主要分析了向量发掘算法，为第四章面向基本块级向量化的移植与优化奠定基础。

3 面向循环级向量化的移植与优化

3.1 研究动机

处理器之间指令集、SIMD 扩展部件的差异导致自动向量化实现方法有所不同，LLVM 编译器中自动向量化实现算法并不完全适用于国产申威处理器。例如 LLVM 支持 X86 的 AVX 指令集兼容 256 位与 128 位的向量长度，而申威处理器只支持单一的向量长度，若强行使用可变向量长度的自动向量化方法，申威处理器会在自动向量化后生成不支持的向量指令。处理不支持的向量指令需要程序员利用原有的指令进行拼接，完成一系列指令的降级，工作量大且容易导致程序产生段错误以及结果错等问题。针对以上问题，进行面向申威处理器的循环级向量化功能的移植工作。

SIMD 自动向量化目的是发掘并利用数据流中的并行性，对于连续的数据引用，编译器可直接进行向量化处理生成向量指令完成加速效果。程序中的非连续的数据引用，例如包含条件选择控制语句（如 if-else 语句）的循环，为数据流分析带来了很多不确定性和复杂性[46]，严重阻碍了向量化的发掘。

LLVM 在循环向量化中利用掩码访存指令处理控制流向量化，后端指令集对掩码访存指令的支持与否，决定了包含控制流结构的循环是否能够进行向量化。但并不是所有的处理器都支持掩码访存指令，例如申威处理器，这就导致自动向量化时失去控制流向量化机会。本章节针对循环级向量化中的控制流向量化进行算法改进，解决后端不支持掩码访存指令的问题，以及控制流向量化方法的单一性问题。

3.2 循环级向量化的移植

作为编译优化领域的研究热点，SIMD 扩展部件在不断发展。申威系列处理器的 SIMD 长度在不断增长的同时，指令集功能也越来越丰富，这需要基于每一种处理器的向量指令特征实现其自动向量化功能的移植。移植主要包含两大方面：后端 SIMD 相关信息、自动向量化优化遍。

3.2.1 向量寄存器特征

自动向量化移植基于处理器对 SIMD 指令的支持，主要体现在向量代码生成阶段[47]。根据目标机器的指令及系统，完成向量化后中间表示的虚拟指令集到目标机器指令集的指令降级过程。LLVM 采用经典的三段式编译器框架，并且有独立于前后端的中间表示，所以只需要对编译器后端进行扩充以支持申威架构即可，主要步骤如下：

（1）描述目标平台向量寄存器集合：在目标平台后端的 SWRegisterInfo.td 文件中按照规定的语法格式描述目标平台寄存器，使用 TableGen 将 SWRegisterInfo.td 文件生成寄存器相关的目标平台指令代码。

（2）描述目标平台的 SIMD 指令集：在目标后端的 SWInstrFormats.td 和 SWInstrInfo.td 文件中描述目标平台的指令集。

（3）实现 SIMD 中间表示语句到目标平台汇编指令的降级。完成从 IR 语句到有向无环图（Directed Acyclic Graph，DAG），从 DAG 到机器指令（Machine Instruction，MI），从 MI 到机器代码指令（Machine Code Instruction，MCInst），再从 MCInst 到汇编指令的降级过程。

传统向量处理器利用流水线技术完成向量的并行处理，将数据被从存储器或向量寄存器中提取，运算后再写回寄存器中[48][49]。现代通用处理器的 SIMD 扩展部件的制造工艺约束了功能单元的数量，因此 SIMD 向量寄存器通常为 128 或 256 位，并且一般不支持可变长度向量操作。

循环级向量化与基本块级向量化的寄存器信息是通用的，所以需要在以上基础需要添加第四个步骤，完善 SIMD 向量寄存器特征信息：即完善 SWTargetTransformInfo.cpp 文件中寄存器宽度的描述。必要时数据类型长度也需要根据指令集信息进行修改，否则会生成后端不支持的向量长度或向量类型，增加后端指令降级工作导致向量代码生成效率的降低。

3.2.2 向量化信息

后端 SWTargetTransformInfo.cpp 文件包含了针对申威处理器自动向量化转换所需的基础信息，包含指令延迟数即指令代价、寄存器相关信息、跨幅因子信息等。当不具备后端相关的向量化转换信息时将读取 Analysis 文件中的通用转换，与实际后端信息的不匹配会导致向量化后产生倒加速。

具有目标平台相关的向量化转换信息以及指令延迟信息，自动向量化阶段才

能做出精准的向量化决策。具有与目标平台相匹配的自动向量化优化，才能最大限度的发挥 SIMD 优势。其中跨幅因子、基本指令代价的精确描述是十分必要的。

（1）基本指令延迟信息：基本指令包含逻辑运算指令、类型转换指令、比较指令、内建函数指令、访存指令等。根据申威处理器硬件方面提供的指令延迟信息，在后端 SWTargetTransformInfo.cpp 文件中对指令代价进行精确描述，针对数据类型、操作码进行识别，对指令延迟数进行补充。手动将后端不支持的向量指令代价调高，防止向量化产生倒加速。对于复杂指令例如混洗指令，结合后端指令降级中自定义处理的指令的拆分组合情况进行精确描述。

这里以逻辑运算指令为例，说明完善指令代价的主要步骤：

1) 首先评估类型合法化以及合法化类型的代价：记录合法化类型的代价，默认为 1；当数据类型需要扩展时，例如<8 x double>向量类型，需要将向量分裂成两个后端支持的<4 x double>类型才能进行下一步分析以及运算，此时合法化类型的代价为 2；然后，获取数据类型 MVT（Machine Value Type），目标处理器支持的任何合法的值类型都可以由 MVT 表示。

2) 获取目标指令操作符：LLVM 使用 ISD 表示 SelectionDAG 节点类型，枚举了各种运算指令的操作符，用于后续指令代价的精确识别。

3) 处理不支持的向量类型：例如本课题使用的申威处理器不支持 128 位的向量长度，当识别出来<2 x double>类型后返回一个极大值，这样可以在自动向量化收益分析时，避免选择此类型的向量决策，从而避免发生程序运行错误等问题。

4) 匹配指令代价信息：依靠操作符与数据类型读取指令的代价信息。MVT 可以指定数据类型，但是无法识别后端不支持的类型，所以步骤三的处理是十分必要的。

（2）跨幅因子设置：跨幅因子即循环向量化中的“Interleave Count”，是基本块中单条语句的展开数，默认为 1。循环向量化阶段读取处理器寄存器信息，从可用寄存器的数目中减去循环不变量的数目，剩余的寄存器将被所有局部展开的指令使用，除以循环所需的寄存器数以 2 的幂次方四舍五入后计算出跨幅因子。向量指令局部展开可以提升性能，根据处理器特性在后端转换信息中约束最大跨幅因子，保证局部展开后不造成溢出。

结合申威处理器向量指令特征，将 SWTargetTransformInfo.cpp 中最大跨幅因

子描述为 4，循环向量化阶段调用代价模型从 1 到 4 范围内分析出最佳跨幅因子，增加了性能提升的机会。为验证此步骤的实际意义，以 TSVC（Test Suit for Vectorizing Compilers）测试集中 vpvts 函数为例进行测试分析，修改后通过收益分析选择出最佳跨幅因子 4，进行向量代码的局部展开，较原始向量化性能提升了 70%。

3.3 控制流向量化优化

3.3.1 控制流向量化流程

实现自动向量化编译的基础是精确的数据流分析，程序控制流的存在带来了许多不确定性和复杂性，如何对包含控制流的程序进行向量化是当前自动向量化编译面临的一个技术难点[50]。

控制流的出现伴随了控制依赖，控制依赖关系是由于程序分支语句改变了程序执行顺序而产生的一种依赖关系[51]。自动向量化在处理控制依赖关系时一般采用依赖转换方法，通过引入 select 指令将控制依赖转换为数据依赖后再进行向量化[50]。

目前 LLVM 编译器中控制流通过循环级向量化生成 select 指令只有一种方法，由通过合法性分析的 phi 节点向量化转换而来，生成方法比较单一且约束性较高。例如当前 LLVM 向量化仅对图 3.1 中程序 1 完成由 phi 节点到 select 向量指令的生成，程序 2、3 因无法在标量优化中生成可以进行控制流向量化转换的 phi 节点，导致在循环级向量化时无法进行有效的控制流向量化。

```
//程序1：
for (int i = 0; i < 1024; i++) {
    if (x[i] > 0)
      a[i] = b[i] + 1;
    else
      a[i] = c[i] - 1;
    }
```

```
//程序2：
for (int i = 0; i < 1024; i++) {
    if (x[i] > 0)
      a[i] += b[i] + 1;
    else
      a[i] += c[i] - 1;
}
```

```
//程序3：
for (int i = 0; i < 1024; i++) {
    if (x[i] > 0 )
      a[i] = c[i] +1;
    else
      b[i] = c[i] +1;
}
```

图 3.1 控制流程序示例

LLVM 提供了静态单赋值格式的控制流变换，在自动向量化前对标量结构进行简化 CFG（Control flow graph）优化生成 phi 指令。phi 指令可根据前面执行

的基本块来选择一个值进行后续向量化操作，向量化阶段将符合格式要求的 phi 指令转换成 select 向量指令。控制流的存在产生了复杂的数据重组[52]，导致指令代价大量消耗在存储指令上，在此情况下循环向量化利用掩码访存指令解决控制流造成的访存收益低的问题。

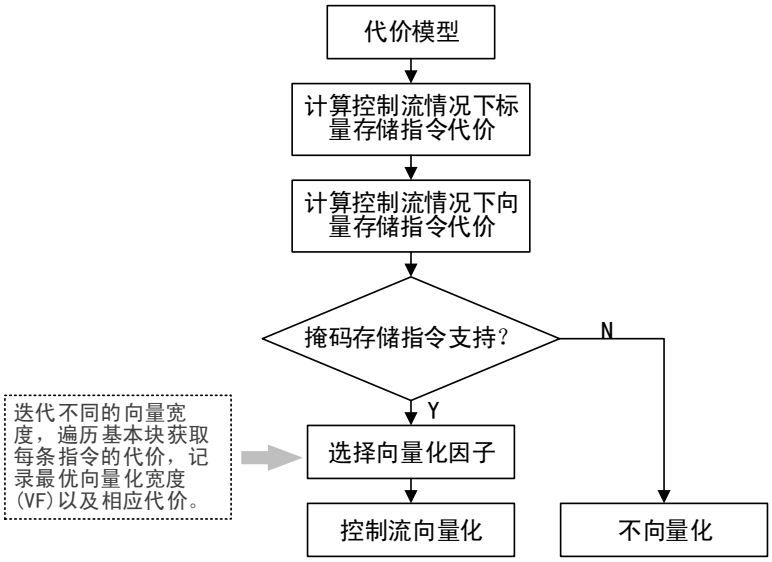


图 3.2　控制流情况下的收益分析流程

对于包含控制流的循环，收益分析流程如图 3.2 所示。循环级向量化调用代价模型，以 2 的 n 次幂迭代不同的向量化因子进行收益分析。由辅助函数查询此向量化因子下有无支持的掩码内建函数，有则计算循环进行向量化的整体收益分析，比较不同向量化因子下指令的平均代价，选出最优向量化因子传递给主函数进行向量代码生成，否则将因为向量化代价极大而放弃向量化机会。以上分析过程可以看出，后端指令集对掩码访存指令的支持与否，直接决定了包含控制流语句的循环是否能够进行向量化。

3.3.2　掩码指令转换算法与实现

掩码内建函数是对 LLVM 基本指令集的补充，由特定目标平台的特殊指令组合而成。但是并不是所有目标架构指令集都具备全面的掩码指令，例如申威处理器，这就导致自动向量化时产生负收益，失去控制流向量化机会。

本文提出了一种优化算法，在循环级向量化时将掩码存储指令进行 select 向量指令转换，解决控制流向量化的问题。在 LLVM 进行循环级向量化决策时，遍历最内层循环查找 IF 结构，实现向量代码的生成，算法如图 3.3 所示。

```
1   功能：通过指令转换使包含IF结构的循环进行向量化
2   输入：最内层循环体IF结构
3   输出：将IF结构中的不连续存储转换成select系列指令
4   profitable = VPlan(CostNewSI) ;//调用代价模型获取指令代价
5 //读取更新后的CostNewSI进行代价分析
6 if (profitable)
7     NewLoop = CreateNewBody(IF)//创建包含IF结构新的循环体
8     //进行加宽指令决策
9     if (phi_Sel节点){
10        VPBlend::execute;
11    }else{
12        for ( Instr : NewLoop)
13            if (!MemoreyInstr)
14                //调用对应的指令向量化方法
15                VPWiden::execute;
16            else
17                TranslateIftoSelect();
18    }
19 endif
20 procedure TranslateIftoSelect(){
21    get (store ->getHeader())//获取指令所属基本块的分支条件块;
22    get (cmp ->getCmp)//获取header块分支条件cmp;;
23    if mask存在 //获取合法性分析中mask
24        // 将原有调用掩码存储指令替换成以下Select指令
25        // 获取参数进行指令构建
26        LI = CreateLoad(DataTy, VecPtr);
27        SelectI = CreateSelect(mask, StoredVal, LI);
28        NewSI = CreateStore(SelectI, VecPtr);
29    end if
30      //更新指令信息;
31      NewSI替换store;
32    return;
33    //更新NewSI指令代价计算;
34    CostNewSI += CostLoad + CostSel;
35 }
```

图 3.3　掩码指令转换算法

指令转换后改变了原有的语句结构，更新存储指令代价的计算以获取最佳向量化决策。算法优化后在向量化因子大于 1 的情况下不再调用掩码存储指令，而是在循环中获取控制流的 if 条件来创建 select 语句。通过以上转换生成 select 指令，将控制依赖转化成了数据依赖。

在 IF 结构中将掩码存储指令进行 select 指令转换，增加了向量代码生成的机会。图 3.4 原程序中数组 a 与数组 b 由于控制流的存在皆为不连续访存，经掩码指令转换算法改进后能有效生成 select 指令如图 3.4（b）中代码段所示，最终生成的向量指令如（c）所示代码段，能够生成有效的向量指令。

```
for (int i = 0; i < 1024; i++) {
    if (x[i] > 0 )
      a[i] = c[i] +1;
    else
      b[i] = c[i] +1;
 }
```
(a) 原程序

```
for (int i = 0; i < 1024; i++) {
   a[i]=select(x[i] > 0, c[i]+1 ,a[i]) ;
   b[i]=select(x[i] > 0, c[i]+1,b[i]) ;
 }
```
(b) IF转换select后代码

```
for (int i = 0; i < 1024; i+=4) {
     simd_load( Vx, x[i,i+3] );
     simd_load( Vc, c[i,i+3] );
     V0 = (floatv4)(0);
     Vc' = simd_add(Vc, 1);
     Vcmp = simd_vcmple( Vx, V0);
     simd_load( Va, a[i,i+3] );
     Va = simd_vselect(Vcmp, Vc', Va );
     simd_store(Va, a[i,i+3]);
     simd_load( Vb, b[i,i+3] );
     Vb = simd_vselect(Vcmp, Vc', Vb );
     simd_store(Vb, b[i,i+3]);
}
```
(c) 向量化代码

图 3.4　select 指令转换示例

3.3.3　phi 节点优化算法与实现

掩码指令优化算法为生成 select 向量指令提供了新的转换方法，解决了控制流向量化中掩码指令不支持的约束性问题。针对具有控制流结构的循环，通过在标量优化中控制指令下沉生成可转换为 select 向量指令的 phi 节点，增加控制流向量化机会。

LLVM 编译器中的优化和分析被组织成 Pass（遍）结构，通过 Pass 完成不同的优化算法。简化 CFG 遍负责完成循环控制流图的清理，以帮助循环顺利完成其它优化过程，例如一个非规范的 CFG 结构导致另一个循环优化执行得不太理想，那么就可以在这个遍中修复循环的 CFG 结构。

在循环级向量化之前，针对 IF 结构中对同一数据进行写操作的问题，对简

化控制流图优化遍中的指令下沉算法进行改进，具体算法如图 3.5 所示。

```
功能：控制指令下沉生成phi节点
输入：if.then与if.else基本块
输出：包含phi节点的NewBB基本块
 if (isunconditional(BB)) //识别出if.then/if.else基本块
   for (Inst : BB){
      T = B->getTerminatorPreInst();//除分支外倒数第一条指令
      if (Store) //根据定义使用链找到数据写的操作
         def = (Store->getOperand(1));
   }
endif
if (!def.then->isisIdenticalTo(def.else))
   return false;//若两基本块不是对同一数据进行写操作不进行下沉
canSinkInst(Insts);//指令下沉分析
NewBB = InstToSink(T,def);//将范围内指令进行下沉
PHINode = ExecuteBB(NewBB);//调用函数生成phi节点
Update(NewBB);
```

图 3.5　phi 节点优化算法

图 3.1 中程序 2 不能生成 phi 节点的原因是对同一数据分别进行了读和写操作，导致其定义使用链复杂化，在简化 CFG 时阻止了指令的下沉。图 3.5 算法的改进解决了此类问题，在保证正确性的同时增加了 phi 节点生成机会，phi 节点在循环级向量化时可直接转换为 select 向量语句，增强了控制流向量化能力。

3.4　向量化收益评估

LLVM 的循环向量化阶段采用代价模型（Cost Model）来确定最佳的向量化策略[53]，确定向量化策略之后驱动 LLVM 向量化执行该决策，完成中间表示的转变。代价模型可以评估自动向量化移植优化后的程序执行效率[54]，通过构建精准的代价模型编译器可以对向量化能力进行评估，循环级向量化收益分析流程如图 3.6 所示。

代价分析时首先判断是否有用户强制的向量化信息，有无最大向量化因子限制。然后，计算可行的最大向量化因子 MaxVF，获取后端向量寄存器长度 WidthRegister 以及数据宽度 WidthType。

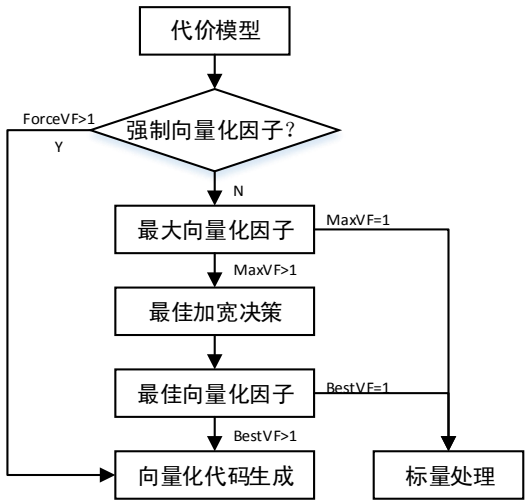


图 3.6　循环级向量化收益分析流程

对于只支持单一向量长度的申威后端：

$$MaxVF = WidthRegister / WidthType \quad (3.1)$$

之后进入收益分析最重要的两部分：针对访存指令的最佳加宽决策选择，针对循环体的最佳向量化因子选择。访存指令对程序的向量化或者性能提升起决定性作用[55]，循环级向量化中针对访存指令具有专属的决策分析以求最大收益。对于连续访存的指令代价，直接从后端 SWTargetTransformInfo.cpp 文件中的指令代价表获取。对于控制流导致的不连续访存，需要对比跨幅访存、聚合访存、标量化访存三种方案下的收益，选取最佳决策进行向量代码生成。

（1）跨幅访存收益分析：跨幅访存代价计算较为复杂，首先需要获取连续访存指令的代价$Cost_{\mathrm{MemoryOp}}$，保证向量类型合法化后删除无用数据的指令代价，例如：

%vec = load <16 x i64>, <16 x i64>* %ptr

%v0 = shufflevector %vec, undef, <0, 4>

向量类型<16 x i64>的数据通过类型合法化后分为 4 个<4 x i64>类型的 load 指令，那么只有 4 条标量 load 指令会被用到，其他皆为无用数据。由于跨幅访存向量化中存储指令必须连续，所以这里我们将加载指令代价进行缩放计算。$NumLegalInsts$为类型合法的 load 指令数，$UsedInsts$为真正使用到的指令数，此时代价：

$$Cost = Cost_{\mathrm{MemoryOp}} * (UsedInsts / NumLegalInsts) \quad (3.2)$$

之后需要累加跨幅访存中使用到的额外指令的代价$Cost_{ExtraInsert}$，包含插入与抽取指令。加载与存储指令的额外代价计算方式稍有区别，跨幅加载指令从源向量中提取子向量的元素，然后将它们插入子向量中，例如中间表示：

%vec = load <8 x i32>, <8 x i32>* %ptr

%v0 = shuffle %vec, undef, <0, 2, 4, 6>

额外代价来源于从<8 x i32>向量中的 0、2、4、6 位抽取数据并插入到<4 x i32>向量过程中所需的插入抽取指令，设$GroupNums$为跨幅访存组指令数量，其额外代价：

$$Cost_{ExtraInsert} \mathrel{+}= Cost_{Extra} + GroupNums * Cost_{Insert} \tag{3.3}$$

跨幅存储指令从子向量中提取所需元素，并将它们插入到目的向量中，例如中间表示：

%v3 = shuffle %v0, %v1, <0, 4, 1, 5, 2, 6, 3, 7>

store <8 x i32> %v3, <8 x i32>* %ptr

额外指令代价来源于从两个<4 x i32>向量中抽取八个元素插入到<8 x i32>向量过程中所需的插入抽取指令，设跨步为 Steps，其额外代价：

$$Cost_{ExtraInsert} \mathrel{+}= Cost_{Extra} * Steps + Cost_{Insert} \tag{3.4}$$

最后，跨幅访存的总代价为：

$$Cost \mathrel{+}= Cost_{ExtraInsert} \tag{3.5}$$

（2）聚合访存收益分析：聚合访存代价的计算方式较为简单，设$Cost_{\text{address}}$为地址计算的代价，$Cost_{\text{gatherscatter}}$为聚合指令代价：

$$Cost = \left(Cost_{\text{address}} + Cost_{\text{gatherscatter}}\right) * GroupNums \tag{3.6}$$

对于不支持聚合指令的后端，可将$Cost_{\text{gatherscatter}}$手动调整到较大的代价，防止产生不必要的中间表示指令。

（3）标量化收益分析：标量化访存代价需要获取标量存储指令和地址计算的代价后进行计算：

$$Cost = VF * Cost_{\text{address}} + VF * Cost_{\text{MemoryOp}} \tag{3.7}$$

包含控制流的循环属于不连续访存，掩码指令转换算法将包含控制流的循环进行 select 指令转换，改变了包含控制流访存收益分析的计算方式，优化后收益分析计算：

$$Cost = Cost_{Load} + Cost_{\text{Select}} + Cost_{Store} \tag{3.8}$$

更新包含控制流的代价计算后，根据基本块执行的概率来扩展代价，循环

向量化认为每个基本块执行的概率为 50%，所以分支基本块的代价：$Cost\ /= 2$，最后将循环中所有指令进行累加。

对比同一向量化因子下三种不连续访存的收益，选择最优收益方案执行。首先，以 2 的幂次方迭代不同的向量化因子，从[2,MaxVF]范围选择最优。对于只支持单一向量长度的处理器，只需要考虑 MaxVF 情况下的收益，对比标量与向量形式下两者的收益，选择进行向量代码生成或者保持原有的标量执行。

3.5 本章小结

本章节针对申威处理器平台进行循环级向量化移植与优化工作，提高循环级向量化适配能力。完善了向量化所需的指令代价信息，并在精准代价模型的指导下生成后端支持的向量指令。同时提出了一种掩码指令转换算法，对 IF 结构进行 select 指令转换，使控制流自动向量化时不再依靠内建函数，完成了控制依赖到数据依赖的转换。在此基础上对原有的 phi 节点生成方法进行算法优化，增加控制流向量化机会。

4 面向基本块级向量化的移植与优化

4.1 研究动机

Larsen 提出的超字并行向量化方法利用基本块内同构语句生成向量指令，将地址连续的数据作为根节点进行打包操作，再根据定义-使用链进行包的扩展，最后对不包含数据依赖环的包进行向量指令转换。LLVM 采用此算法完善自动向量化功能，针对申威处理器特性，从同构语句数约束以及代价信息完善两方面开展面向申威处理器的基本块级向量化功能的移植工作。

理论上基本块级向量化算法的复杂度高于循环级向量化，具有更强的向量发掘能力，但基本块内同构语句打包过程存在较大的随机性，往往难以得到满意的结果。通过在打包过程引入收益分析，可以得到适合申威后端的同构语句包，向量代码转化后提升程序向量化性能。

并行应用程序的特征复杂多样，发掘基本块内的并行性的向量化方法需要不断完善。绝大多数的并行机会都存在于循环的多次迭代间，单一的向量化算法无法全面的进行向量发掘。将基本块级向量化与循环展开相结合，可以弥补以上问题，既可以发掘迭代内的并行性，也可以发掘迭代间的并行性。

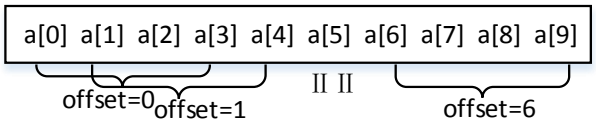
4.2 基本块级向量化的移植

基本块级向量化的移植工作同样分为后端 SIMD 相关信息与自动向量化优化两大部分，其中后端 SIMD 相关信息与循环级向量化共用。循环级与基本块级向量化算法以及实现方式不同，基本块级向量化移植过程中的工作重点在于数据打包长度与向量化因子的设置，以及一些专属的后端向量化信息的完善。

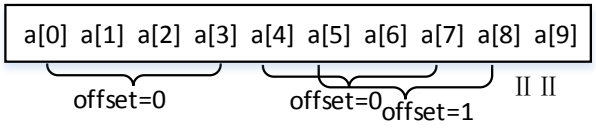
4.2.1 同构语句数约束

基本块级向量化要求指令是同构的，即语句具有相同的数据类型、操作码和操作数。以存储指令为根指令进行打包时，连续的地址内存引用具有天然的打包优势，当算法识别出来内存连续会以内存中数据起始位置的地址为参考，依次迭代不同的偏移进行打包分析。如图 4.1 所示 a[0]~a[9]数据内存连续，假设向量化因子为 4，SLP 算法以 a[0]地址为基准，从第 0 位偏移开始以向量长

度 4 进行打包分析。对语法树进行调度以及代价分析，若不符合打包要求便从下一位偏移位 1 开始重复以上步骤，直至整个连续数据分析结束即第 6 位偏移处，如图 4.1 示例 1 所示。若第 0 位偏移符合向量化的打包要求，将 a[0]~a[3]进行打包后以 a[4]地址为参考继续进行打包分析，如图 4.1 示例 2 所示。



（a）示例1



（b）示例2

图 4.1　同构语句打包内存偏移示例

LLVM 默认向量寄存器长度为 128bit，在自动向量化过程中最小同构语句数限制为 2。这在申威平台是不适用的，不匹配的向量长度会造成数据处理过程中读取错误信息，使程序运行时引发段错误。

基本块级向量化过程中需要对一次性打包的同构语句数进行限制，使其生成 256bit 长度的向量指令，例如源码中针对向量化因子的约束，由原始同构语句数不小于 2 修改为不小于 4，同时针对调用与被调用函数中有关同构语句数限制的地方进行同步修改。当面向申威处理器的向量化过程识别到向量寄存器为 256bit 时，对于同构语句数小于 4 的向量类型不再进行向量化分析与代码生成。

4.2.2　向量指令代价信息

基本块级向量化的基本指令代价、向量寄存器信息与循环向量化共用，除去公共部分，基本块级针对数据重组使用到的混洗指令与插入抽取指令具有专属的分析函数，以保证收益分析的准确性。

基本块级向量化利用混洗指令对向量寄存器中的数据进行重组，各种混洗指令类型的代价计算方式不同，需要根据特定处理器指令集特征以及降级处理进行代价的补充与计算。向量化过程中混洗指令主要分为以下几种类型：

1) Broadcast 类型：将元素“0”广播到向量指令中的每一个子向量元素中，只需要引用寄存器中输入的第一个元素，代价计算较为简单一般为 1。
2) Reverse 类型：将向量寄存器中数据位置取反，若处理器支持此类型指令，可以直接写成枚举进行代价匹配。
3) Select 类型：从任一源操作相应的位置抽取子向量元素，组合成向量指令，相当于带有常量条件操作数的 select 向量指令。
4) InsertSubvector 类型：按照偏移在相应位置插入子向量元素，索引表示起始偏移量。
5) ExtractSubvector 类型：从向量指令相应位置抽取子向量元素，索引表示起始偏移量。如果子向量从向量的开头开始抽取，指令代价为零，如果子向量是对齐的，指令代价也会变得很低。
6) SK_PermuteSingleSrc 类型：根据掩码将向量指令中的子向量元素的顺序进行重新排序。

对于插入抽取指令，首先进行类型合法化检查，当数据类型可以拆分时将索引规范化为新类型，例如将<8 x double>拆分成<4 x double>后，更新索引位置 Index = Index % VF，最后根据特定后端的特征进行代价补充。

4.3 向量化收益分析与优化

4.3.1 向量化收益分析

根据语句的并行信息分析各种可能的向量化方案所带来的收益，选定方案后进行向量代码生成，收益有无决定了是否进行向量化。收益分析可以反映出向量化移植工作的完整性与有效性，基本块级向量化代价模型以抽象语法树为基础，根据收益分析选择生成向量代码或者保持标量执行，具体过程如图 4.2 所示。

代价模型获取申威处理器后端的向量寄存器信息，计算填充一条 SIMD 向量指令所需的同构语句数，根据指令代价选择收益最大的策略进行向量化。同构语句长度（即同构语句数）与向量化因子：设同构语句数为 E，向量化因子为 VF（例如向量寄存器长度为 256 位，标量元素 double 为 64bit，则向量化因子 VF 等于 4），满足 E≥VF 时进行向量化的收益分析。

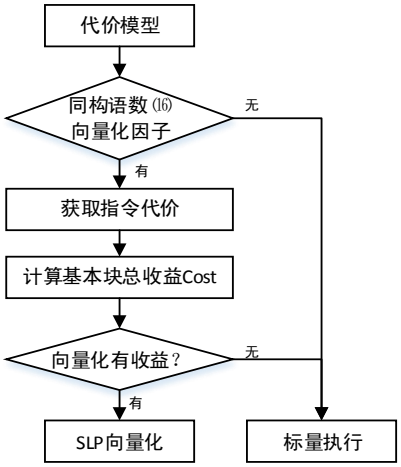


图 4.2　基本块级向量化收益分析流程

代价模型以存储指令为根节点自下而上遍历语法树，当同构语法树的最后一条指令的数据来源于不同的操作时，将使用到混洗指令进行数据拼接，以保证可进行向量化的语法树的完整性。假设一条标量指令的代价为S_{Cost}，相应的向量指令代价为V_{Cost}，$ShuffleNumbers$为需要拼接的指令数，此标量语句进行向量化时包含的混洗指令代价：

$$Shuffle_{Cost} = (ShuffleNumbers - VF) * S_{Cost} \tag{4.1}$$

基本块向量化后以向量指令和标量指令的指令延迟数作为计算依据，向量化收益来源于减少的指令周期数，则此标量指令进行向量化时的收益为：

$$Cost = Shuffle_{Cost} + V_{Cost} - VF * S_{Cost} \tag{4.2}$$

基本块的整体收益为所有向量指令的收益的累加，该基本块向量化的收益总和为：

$$Cost = \sum_{i=1}^{N} Cost_{InstVect[i]} \tag{4.3}$$

存在一些特殊的程序在向量化时要考虑寄存器溢出与指令重组对性能的影响，此时需要遍历语法树，计算额外开销$Cost_{extra}$（理想状态下为 0），在此情况下基本块的收益为：

$$Cost_1 = Cost + Cost_{extra} \tag{4.4}$$

另外，一些不包含存储指令的计算特征，将不会被基于同构 store 组的自底向上的遍历过程捕获。例如标量的规约计算以及向量化指针指令的索引计算，此类收益分析除去基本的计算指令，还需要考虑插入/抽取指令以及索引信息指令的代价$Cost_{user}$（从后端相关指令延迟表中获取，默认为 0），此类情况下收

益为：

$$Cost_2 = Cost + Cost_{user} \quad (4.5)$$

最后将基本块总收益$Cost_1$或$Cost_2$与阈值（默认为 0）进行对比，小于零表示向量化后有收益，调用函数完成向量代码生成，否则放弃向量化。

4.3.2 代价评估优化算法

LLVM 中性能提升主要依靠循环级向量化，原因在于基本块级向量化时要遍历所有的语句以查询同构语句，功能实现的算法复杂度较高，即使采用了启发式的向量化算法，在同构语句打包过程中仍存在较大的随机性，难以得到符合特定后端的满意的结果。

如图 4.3 中程序 a，假设向量化因子为 2，向量化算法自下而上遍历语法树搜寻同构语句，首先将 s0 与 s1 同构部分“a = b + c”进行打包生成向量指令，过程中 s1 将使用插入抽取指令进行数组 e 的填充，以维持向量化过程中数据访存的逻辑顺序，之后以同样的方法对 s2 与 s3 进行打包与向量化。此时向量化是有收益的，现有的代价模型完成收益分析后生成向量代码，但是显而易见 s1 与 s2 打包、s3 与 s4 打包才能生成更加简洁高效的向量指令。所以需要一个合适的算法，在打包过程中选取最佳同构语句包进行向量代码的生成。

```
//示例程序 a.c
 for (int i = 0; i < LEN; i+=6) {
s0:  a[i] = b[i] + c[i];
s1:  a[i+1] = b[i+1] + c[i+1] + e[i+1];
s2:  a[i+2] = b[i+2] + c[i+2] + e[i+2];
s3:  a[i+3] = b[i+3] + e[i+3];
s4:  a[i+4] = b[i+4] + e[i+4];
s5：  a[i+4] = b[i+4] + c[i+4];
  }
```

```
//示例程序 b.c
 for (int i = 0; i < LEN; i+=6) {
     a[i] = b[i] + c[i];
     a[i+1] = b[i+1] + c[i+1];
     a[i+2] = b[i+2] + c[i+2] + e[i+2];
     a[i+3] = b[i+3] + c[i+3] + e[i+3];
     a[i+4] = b[i+4] + c[i+4] + e[i+4];
    a[i+5] = b[i+5] + c[i+5] + e[i+5];
 }
```

图 4.3 同构语句打包的示例程序

针对以上问题，本文设计了一种优化算法，在打包过程中引入收益分析，完善代价模型的同时选取打包过程中的最佳同构语句包，具体算法实现如图 4.4 所示。

```
1   功能：实现基本块级向量化中收益最好的打包方法
2   输入：以存储指令为根指令的同构语句语法树
3   输出：最佳同构语句的向量指令
4   procedure codegenInBestOffset(){
5   int BestCost = 0;//以0为基准判断正负收益
6   unsigned BestOffset = 0;
7   OriginalCost = SLPFunc(Offset);//记录原始SLP算法下StoreChain向量化的收益
8   for (Idx=0, E=GS;Cnt + VF <= E){//迭代连续数据的不同偏移
9       int Cost = calculateCost(Chain, R, Cnt);//计算相应偏移下的收益
10      if (BestCost > Cost){
11          BestCost = Cost;//选取StoreChain上最优收益
12          BestOffset = Cnt;//记录最优收益的偏移位置
13      }
14    ++Cnt;
15    }
16
17  if (BestCost < 0 && BestCost < OriginalCost){//对比原始SLP算法下的起始偏移
位与最佳起始偏移位下的向量化收益
18      CodeGen(BestOffset);//以最佳起始偏移位开始进行打包，生成向量指令
19  }else{
20      CodeGen(Offset);//保持原有SLP算法的向量化方法
21  }
22 }
```

图 4.4　基本块级向量化中代价评估优化算法

经过算法优化后示例程序 a.c 在向量化过程中自动选取了收益更佳的打包方式，如图 4.5 蓝色部分所示，分别从偏移位 1 与 3 开始向上遍历语法树，进行同构语句的打包并生成向量指令。

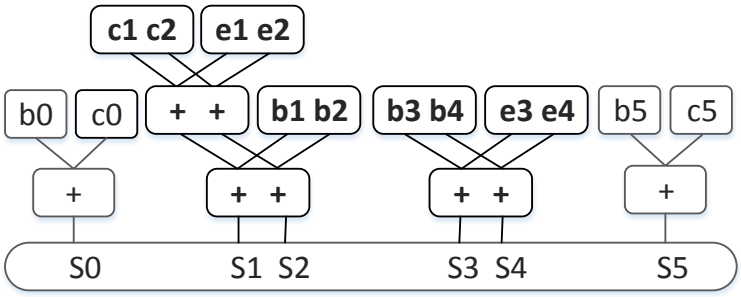


图 4.5　同构语句包发掘过程

基本块级向量化过程中，性能的提升与向量指令一次处理的数据个数息息相关。向量化因子越大存在的收益也就越明显，如示例程序 b.c，当向量化因子为 4 时，经算法优化后加速比可达 1.7。

4.4 混合基本块级向量化方法

向量化阶段面向的应用程序多种多样，需要针对程序特征设计向量化方法。单一的向量化方法在向量发掘过程中受到约束，例如向量化因子为 4 同构语句数为 2 时，由于并行数据量不足以充满一个向量寄存器而使程序失去向量化机会。针对此类问题，本文提出了一种混合向量化方法，将基本块级向量化与循环展开相结合，使循环迭代间的并行机会转移到循环迭代内，增强 SLP 算法的发掘能力。

4.4.1 循环展开技术

循环展开技术通过将循环体内的代码复制多次，增加每次迭代计算的元素的数量从而减少循环迭代次数，是一项常用的程序优化技术。循环展开技术不仅可以提高流水线功能部件的效能[1]，还能增大处理器指令调度的空间，使程序获得更多的指令级并行的机会。

循环展开保持了语句的执行顺序，所以一般不需要依赖分析[7]。但是不恰当的展开可能会导致代码膨胀，也会因为调度循环时寄存器需求变大造成寄存器溢出[56]等问题，给程序性能的提升带来负面影响。以图 4.6 示例对循环进行展开。

```
// 原始循环
for (int i = 0; i < 1024; i++) {
    a[i] = b[i] + 1;
}
```

```
//按因子为2进行展开
for (int i = 0; i < 1024; i+=2) {
    a[i] = b[i] + 1;
    a[i+1] = b[i+1] + 1;
}
```

图 4.6 循环展开示例

LLVM 中循环展开主要分为以下四个步骤：

（1）遍历 CFG，定位需要展开的循环。循环展开要求循环格式正规，正规化可能会增加新的内部循环，所以必须在合法性和收益分析之前进行检查。这

意味着循环展开优化将对所有循环进行结构正规化操作，即使最后循环没有展开。

（2）确定循环展开的循环上界以及循环展开数。如果有多个出口块选取其中的 latch 块，使用 latch 块中的归纳变量信息来估计循环上界，否则使用单个退出块信息进行估计。当外界没有传入展开因子时，调用函数进行收益分析，确定展开因子以及合适的展开方案。

（3）对循环进行展开。展开循环算法的核心在于复制循环体后进行重组，并替换原始循环。首先对循环体基本块进行整体复制，为保证展开后循环体支配关系不变，需要删除复制后的循环体基本块中首条 phi 指令，另外循环体复制次数为循环展开数减 1，例如图 4.6 中原始循环的 IR 表达：

```
//原始循环体基本块：
    for.body:                                 ; preds = %for.body.1, %entry, %for.body
      %indvars.iv = phi i64 [ 0, %entry ], [ %indvars.iv.next, %for.body ]
      %arrayidx = getelementptr inbounds [1024 x float], [1024 x float]* @b,
i64 0, i64 %indvars.iv, !dbg !10
      ……
      %exitcond = icmp ne i64 %indvars.iv.next, 1024, !dbg !19
     br          i1          %exitcond,          label          %for.body,
label %for.cond.cleanup, !dbg !8, !llvm.loop !20
```

进行复制的基本块删除了首条 phi 指令，其他指令与原始循环保持相同：

```
//复制的基本块：
    for.body.1 :                                          ; No predecessors!
    %arrayidx.1 = getelementptr inbounds [1024 x float], [1024 x float]* @b,
i64 0, i64 %indvars.iv, !dbg !10
    ……
    %exitcond.1 = icmp ne i64 %indvars.iv.next, 1024, !dbg !19
    br          i1          %exitcond,          label          %for.body,
label %for.cond.cleanup, !dbg !8, !llvm.loop !20
```

基本块复制完成后，设置分支语句将基本块连接起来。调用辅助函数将复制后的基本块与原始基本块进行合并，合并后的循环体即为按照相应展开因子展开后的指令结构。

（4）更新支配关系，进行常量传播和无用代码删除来简化展开的循环。

```
for.body:                                         ; preds = %for.body, %entry
  %indvars.iv = phi i64 [ 0, %entry ], [ %indvars.iv.next.1, %for.body ]
  %arrayidx = getelementptr inbounds [1024 x float], [1024 x float]* @b,
i64 0, i64 %indvars.iv, !dbg !10
  ……
  %arrayidx.1 = getelementptr inbounds [1024 x float], [1024 x float]*
@b, i64 0, i64 %indvars.iv.next, !dbg !10
  ……
  %exitcond.1 = icmp ne i64 %indvars.iv.next.1, 1024, !dbg !19
  br i1 %exitcond.1, label %for.body,
label %for.cond.cleanup, !dbg !8, !llvm.loop !20
```

基于基本块级混合向量化算法的一个重要问题是循环展开因子的确定[57]，最直接的方案是按可并行指令的宽度来确定展开因子，例如若 SIMD 单元能够同时操作四个单精度浮点数或者八个整数，那么展开因子可为四和八，但此类方法会导致冗余指令的生成使循环体指令结构不够简洁。当循环中含有多种数据类型的运算时展开因子的计算会变得复杂，不合适的展开因子会影响 SLP 算法的效率。

4.4.2 混合向量化算法

基本块级向量化向量发掘算法的特殊性，要求同构语句打包数必须与向量化因子数相同，条件约束较为严苛。这就造成当同构语句数小于向量化因子时失去了向量化机会，而同构语句数大于向量化因子时将产生局部标量化的情况。基于以上问题，本文结合基本块级向量化算法与循环展开的特点，实现两种算法相结合的混合向量化方法。

针对应用程序的不同特征，基本块级向量化算法分配如表 4.1，其中 GS（Group Size）代表程序同构语句数，VF 代表向量因子即一个 SIMD 向量寄存器所存放的数据个数。

表 4.1 基本块级向量化算法分配

向量化对象特征	特征	向量化方法
循环迭代内并行	GS = VF	SLP 基本块级向量化
循环“迭代间+迭代内”并行	GS < VF；GS > VF	Unroll+SLP 混合向量化
非循环结构	GS ≥ VF	SLP 基本块级向量化

对于非循环结构，采用基本块级算法进行基本块内语句并行的向量化发

掘，不符合则不进行向量化。对于循环结构，依靠同构语句数与向量化因子的关系，选择基本块级算法或混合向量化方法。在完成候选发掘算法的分析后，可以得出如何向量化循环的决策。

混合向量化算法中最重要的是循环展开因子的计算。循环因子过大会导致额外的寄存器溢出[58]，同时因为代码量庞大也会引起指令缓存区溢出 [59]，降低程序的运行性能。结合基本块级向量化算法特征，混合向量化产生的循环展开因子取同构语句数与向量因子的最小公倍数，具体算法如图 4.7 所示。

```
1 功能：实现基本块级向量化与循环展开结合的混合向量化
2 输入：同构语句数与向量化因子数不同的循环体
3 输出：包含简洁向量指令的循环体
4 procedure UnrollLoopInSLP(){
5    GS->collectSeedInstructions(BB);//收集存储指令组GS
6    for (Idx = GS-1; Idx >= 0; --Idx)
7       FindConsecutiveAccess();//保证GS组数据内存连续
8
9    for (Loop *L : *LI)//查询最内层循环
10      Worklist.push_back(L);
11
12   for (Loop *CurL : Worklist)
13      //识别GS与VF关系,MaxElts=SIMD向量长度/数据长度
14      if (GS < MaxElts || GS > MaxElts){
15         for (auto BB : CurL)
16            if (!isomorphic(BB))//识别是否同构
17               break;
18
19         //通过以上验证，开始进行循环展开
20         UCount = (lcm(GS,MaxElts)/GS);//计算循环展开数
21         SLPUnrollResult = UnrollLoop(UCount);//调用循环展开函数进行展开
22         for(Instruction &I : *BB)
23            NewStores .push_back(I);//  更新新的Store组
24
25         Changed |= vectorizeStores(NewStores);//调用SLP进行向量化
26      }
27 }
```

图 4.7　混合向量化算法

该算法以最内层循环作为优化的基础，确认同构语句数后计算展开因子数，将展开因子传递给循环展开函数进行基本块的语句展开，完成后调用基本块级向量化完成同构语句的打包与向量代码生成。

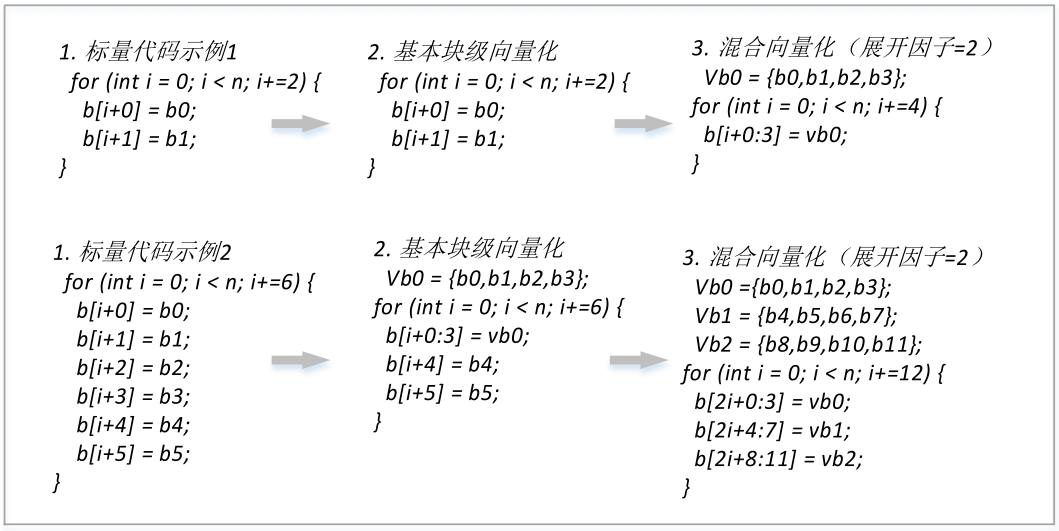


图 4.8　混合向量化代码生成示例

以向量化因子为 4 为例，当同构语句数为 2 时不满足向量化发掘要求放弃了向量化机会，如图 4.8 示例 1 的步骤 2 所示标量执行，采用混合向量化进行循环展开后符合基本块级向量化要求并进行了向量代码的生成，如示例 1 的步骤 3 所示进行了向量执行；当同构语句数为 6 时，原始向量化只对前四条同构语句进行了打包合并生成了向量代码而后面两条保持标量执行，如示例 2 的步骤 2 所示部分代码进行了向量化，进行循环展开后可以有效利用所有标量指令，生成了简洁高效的向量指令如示例 2 的步骤 3 所示。

4.5　本章小结

本章节针对申威处理器平台进行循环级向量化移植与优化工作，完善了基本块级向量化中数据重组所需的指令代价，为符合后端处理器特征修改了一次性打包的语句数量。基于现有的代价模型，设计了一种代价模型的优化算法，在打包过程中引入收益分析，选取打包过程中的最佳同构语句包，生成最符合申威后端的简洁高效的向量指令。另外结合循环展开与基本块级向量化，设计了一种混合向量化的方法，实现了迭代间与迭代内的向量发掘。

5 测试与结果分析

本文采用 SW1621 处理器为测试平台，进行 LLVM 编译器自动向量化移植与优化的功能测试，处理器主频 1.60GHz，32KB L1 数据/指令 Cache，512KB L2 Cache，32MB L3 Cache，编译器版本为 7.0.0。分别采用 SPEC CPU2006 与 TSVC 标准测试集以及向量化应用类测试，从正确性以及性能两方面进行测试与分析。

5.1 自动向量化移植正确性测试

SPEC CPU2006 标准测试集是 SPEC 组织推出的 CPU 子系统评估软件，包含定点 CINT2006 与浮点 CFP2006 两类测试题。CINT2006 用于测量和对比整数性能，包含压缩程序、量子计算机仿真、象棋程序等。CFP2006 则用于测量和对比浮点性能，包括分子动力学质点法、有限元模型结构化网格法、流体动力学稀疏线性代数法等。

采用 SPEC2006 标准测试集的 29 道测试题验证自动向量化移植与优化的正确性，测试结果如表 5.1 所示。

表 5.1 SPEC 测试正确性分析

程序名称	测试项目	编程语言	自动向量化移植前	自动向量化移植后
400.perlbench	Perl 语言解释器	C	通过	通过
401.bzip2	压缩程序	C	通过	通过
403.gcc	AMD 机器码生成	C	通过	通过
410.bwaves	流体力学	F77	结果错	通过
416.games	量子化学	Fortran	结果错	通过
429.mcf	组合优化	C	通过	通过
433.milc	量子力学	C	通过	通过
434.zeusmp	磁体力学	F77	通过	通过
435.gromacs	生物化学	C、Fortran	通过	通过
436.cactusADM	广义相对论	F90、C	通过	通过
437.leslie3d	流体力学	F90	通过	通过
444.namd	生物分析系统	C++	通过	通过

445.gobmk	围棋	C	结果错	通过
447.dealII	有限元分析	C++	通过	通过
450.soplex	线性编程优化	C++	通过	通过
453.povray	影像光线追踪	C++	通过	通过
454.calculix	结构力学	F90、C	结果错	通过
456.hmmer	基因序列搜索	C	通过	通过
458.sjeng	国际象棋	C	通过	通过
459.GemsFDTD	电磁学计算	F90	结果错	通过
462.libquantum	量子计算	C	通过	通过
464.h264ref	视频压缩	C	段错误	通过
465.tonto	量子化学	F95	编译错	通过
470.lbm	流体动力学	C++	通过	通过
471.omnetpp	离散事件仿真	C++	通过	通过
473.astar	寻路算法	C++	通过	通过
481.wrf	天气预报	F90、C	段错误	通过
482.sphinx3	语音识别	C	结果错	通过
483.xalancbmk	XML 处理	C++	通过	通过

基于申威处理器进行 LLVM 的自动向量化的移植，移植前的向量化正确性测试如表一第四列所示。中间代码在向量化中生成了后端不支持的向量类型，在后端指令降级过程中找不到匹配的降级方法与指令模板，导致测试出现段错误以及结果错等问题。

以测试题 464.h264ref 为例，段错误产生于基本块级向量化，原因在于向量化 phi 节点时（以 phi 指令为根节点的规约向量化，根据定义-使用链自底向上分析最终向量化 load 指令），生成了 load <2 x double>的向量类型。而申威处理器支持的向量长度为 256bit，针对双浮点数据类型只支持向量化因子为 4 的情况，通过对向量化因子以及同构语句数进行约束，解决了此测试题段错误的问题。类似的问题也发生在循环向量化中，例如测试题 410.bwaves 的结果错，LLVM 中默认的向量寄存器长度为 128bit，导致生成后端不支持的向量类型，通过向量寄存器特征的准确描述解决了以上问题。另外可以在向量指令代价表中，针对不支持的向量指令进行手动的代价提升，经过收益分析后自然不会选择包含不支持指令的向量策略。

综上，通过完善循环级向量化阶段的向量寄存器特征与向量化信息，约束基本块级向量化同构语句数，解决了测试题编译与运行错等问题，移植后正确性测试如表 5.1 最后一列所示，全部正确。

5.2 循环级向量化测试

循环级向量化性能测试主要分为两部分，一部分是控制流向量化能力测试，验证掩码指令转换算法与 phi 节点优化算法的有效性；另一部分则是在完善指令代价表并更新循环向量化代价模型后的性能测试。

本模块采用 TSVC 标准测试集，TSVC 主要针对编译器的自动向量化能力进行性能测试。测试以优化前的向量化能力作为基准，进行循环级向量化相关优化的性能分析。

5.2.1 控制流向量化策略性能测试

针对循环中因包含控制流结构而影响向量发掘的情况，改进向量化算法进行掩码指令转换优化，实现了原本不支持的控制流向量化方法，充分利用了申威指令集中的向量指令，实验结果如图 5.1 所示。

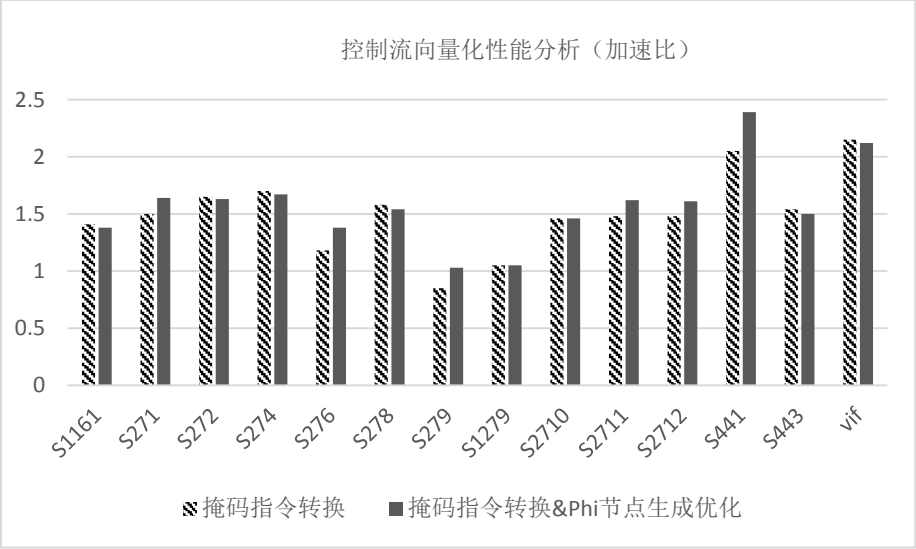


图 5.1 控制流向量化示例

掩码指令转换策略后的性能测试如蓝色柱状图所示，整体平均加速比提升 50%。其中 s271、s1279、s2711、s2712、vif 测试题在算法改进前会生成大量的向量抽取指令，不能充分利用申威支持的特殊向量指令，改进后生成了简洁高效的 select 向量指令，性能得到了有效提升。在此基础上进行 phi 节点优化算法进行测试，其中 s441 测试用例效果显著加速比达 2.4。

控制流向量化优化整体性能如黄色柱状图所示，TSVC 标准测试集中针对

控制流结构的循环识别率提升 48%，平均加速比提升 51%。

5.2.2 精准代价指导下性能测试

自动向量化的移植工作完善了向量指令的代价信息，基于基本运算指令代价的精准描述，自动向量化做出了符合申威平台的向量化决策，生成了简洁高效的向量汇编指令。采用 TSVC 测试集中典型例题，对比移植优化前后的加速性能，实验结果如图 5.2 所示。

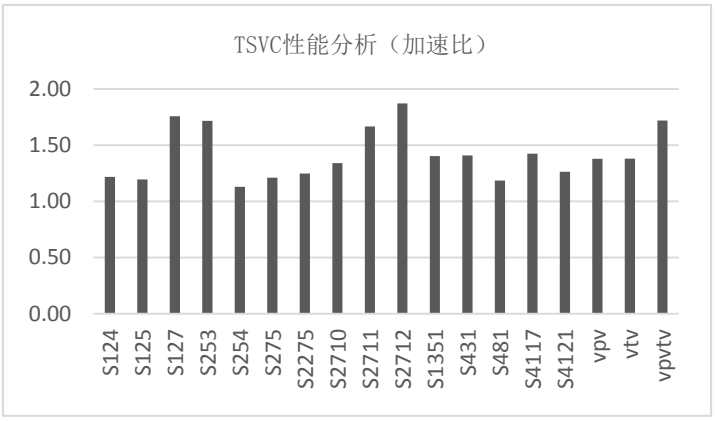


图 5.2 精准代价指导下向量化示例

循环级向量化利用代价模型选取符合后端的向量化决策与向量化因子，精准代价指导下的代价模型将自动向量化与申威后端特征相结合，有效地利用了申威处理器的 SIMD 向量部件，平均加速比提升 42%。

移植优化前编译器对不支持的向量指令进行指令降级处理，会生成冗余的标量指令导致程序产生倒加速。精准代价模型指导下的自动向量化排除了后端不支持的向量类型，使倒加速得到了有效的改善，测试结果如图 5.3 所示。

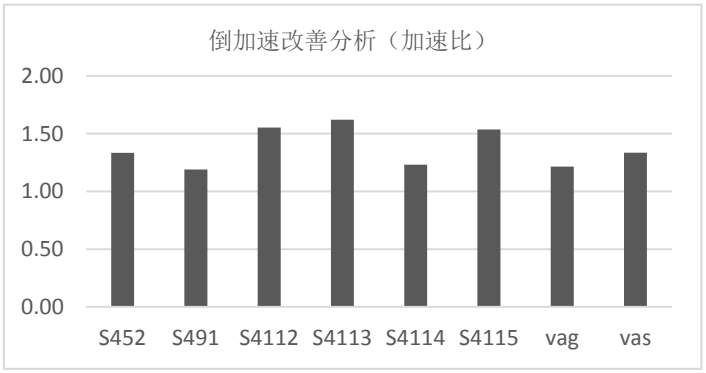


图 5.3 倒加速改善示例

精准代价指导的自动向量化选择了与后端匹配的向量化因子以及向量化方法，在原有移植基础上平均加速比提升 28%，倒加速情况得到明显改善。

5.3 基本块级向量化测试

基本块级向量化性能测试分为两部分，一是代价评估优化算法后的性能分析，另一部分是混合向量化策略下的性能分析，测试分别采用 SPEC 标准测试集中的 453.povray 测试用例以及特征明显的典型示例。

5.3.1 代价评估优化策略性能测试

通过在同构语句的打包过程中引入收益分析，使向量发掘过程能有效的选择出最佳同构语句，打包后生成更加简洁高效的向量代码。选取 SPEC 标准测试集中 453.povray 测试用例，使用 gprof 获取程序中各函数在 ref 规模下的运行时间，优化后的性能提升效果如图 5.4 所示。

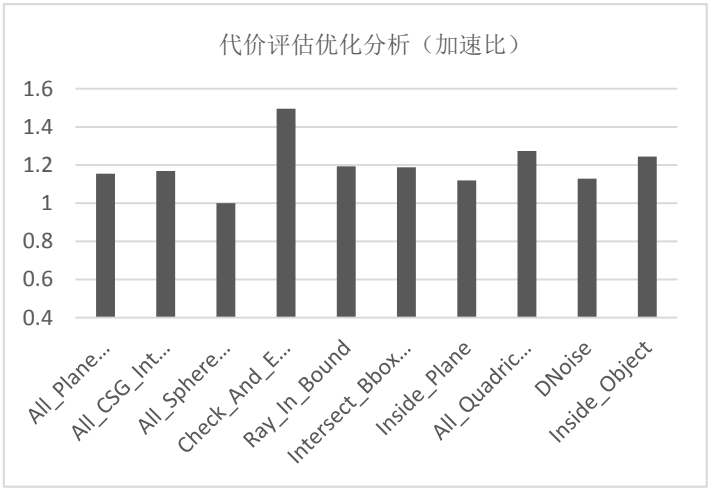


图 5.4 代价评估优化策略下加速比

测试选取前十个耗时函数进行分析，优化后皆有显著的性能提升，平均加速比提升 19.6%。测试示例中的影像光线追踪是基于文本的场景描述来生成图像，支持大量的几何图元和构造实体几何图形，在 453.povray 测试题的 vector.h 头文件中，包含了大量的空间几何的向量计算。而 frame.h 头文件定义了全局可访问的类型与常量，并应用于 POV-Ray 中的所有 C 模块。函数的性能提升主要来源于以上两个头文件中函数的最优向量化，在代价模型指导下选取了收

益最佳的同构语句包。Check_And_Enqueue 函数加速效果尤为明显，也是因为其调用的子函数摒弃了不合适的打包方式。基于代价评估优化策略，453.povray 测试题的整体加速比提升 17.1%。

5.3.2 混合向量化策略有效性测试

混合向量化优化将循环展开与基本块级向量化相结合，循环展开因子决定了向量化后能否生成简洁高效的指令。为了覆盖更多并行特征，本文选用了 4 个核心循环用以测试有效性，此类循环常见于多媒体应用，通过与优化前的功能进行对比验证算法的有效性。

测试所用的 SIMD 向量寄存器宽度为 256 位，可同时处理 4 个双精度浮点数据或 8 个整型数据。针对向量化因子为 4 的情况，选取同构语句数分别为 2、3、4、6 的循环体进行混合向量化策略的有效性测试，向量化情况与循环展开因子如表 5.2 所示。

表 5.2 混合向量化分析

测试序号	同构语句数	展开因子	优化前是否向量化	优化后是否向量化
Test1	2	2	否	是
Test2	3	4	否	是
Test3	4	1	是	是
Test4	6	2	是	是

循环展开后程序能够有效的进行基本块级向量化，生成了符合申威处理器的向量指令，增加了向量化的识别率，测试典例的加速效果如图 5.5 所示。

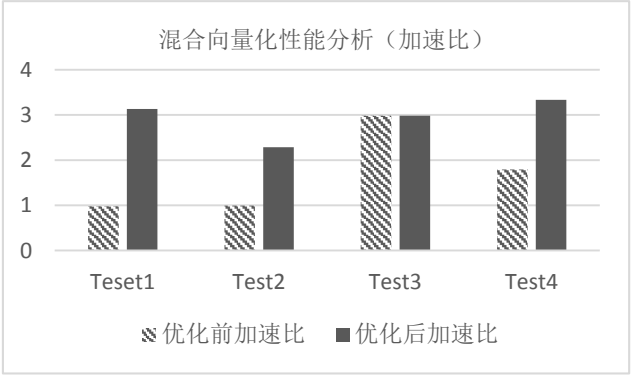


图 5.5 混合向量化策略性能分析

算法优化后使程序运行性能得到提升，平均加速比达 2.04。其中 Test3 同构语句为 4 与向量化因子相同，所以优化前亦可进行基本块级向量化。而 Test1 与 Test2 的同构语句数量无法填充满一条 SIMD 向量指令，所以优化前保持标量执行，优化后同构语句数量整除向量化因子符合向量化要求，性能提升较为显著平均加速比达 2.65。Test4 测试题只有部分指令进行了向量化，优化后所有标量指令都转换成为简洁的向量指令，加速比达 1.8。

5.4 整体测试

5.4.1 SPEC 性能测试

整体性能测试选取 SPEC 标准测试集的 29 道测试示例进行分析，编译优化选项-O3 -static（-O3 选项下默认打开自动向量化），采用 ref 规模单进程运行，取 3 次运行结果的平均值，测试结果如图 5.6 所示。

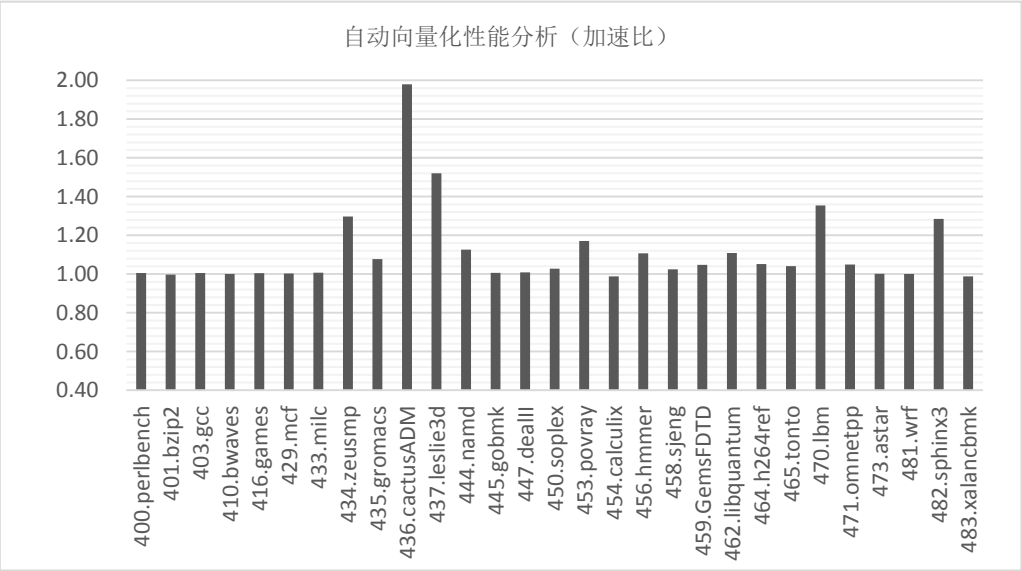


图 5.6 SPEC 整体性能测试分析

一些程序由于存在依赖环或函数调用等特征，自身不具备向量化的潜质，例如 416.games 核心函数为非连续数组且存在向量依赖，459.GemsFDTD 核心函数存在子函数调用，所以选取与向量化相关的代表性的测试题进行分析。对比 X86 平台与申威平台的向量化能力，验证移植的正确性与优化策略的有效性。测试对比结果如图 5.7 所示，其中 X86 选用与申威 SIMD 长度相同的 AVX

指令集。

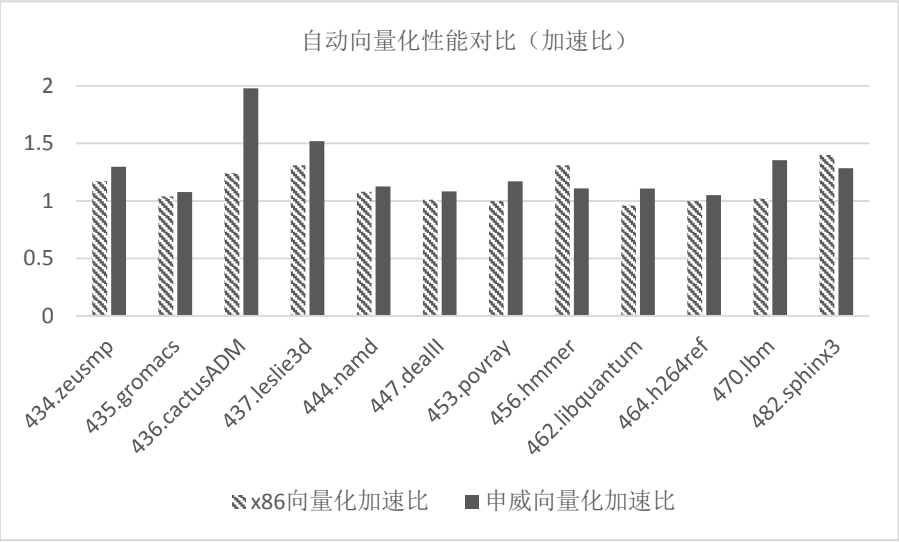


图 5.7 SPEC 性能测试对比分析

前端将测试程序解析为中间表示代码，自动向量化在中间表示层进行优化，生成向量表示的中间代码后由后端进行降级处理，最后生成特定指令集的汇编码。移植优化后生成的向量代码较标量更为高效，符合申威后端指令集特征。

自动向量化移植优化后 SPEC 标准测试集的定点程序平均性能提升 2.7%，浮点程序平均性能提升 18.2%，整体平均性能提升 11.3%。其中 436.cactusADM 加速效果最为明显提升了 97%，437.leslie3d 加速比提升 52%，434.zeusmp、470.lbm、482.sphinx3 加速比平均提升 21%，另外 453.povray 加速比提升 17%。处理器制作工艺的差异导致指令集存在不同，一些特殊的向量指令申威处理器本身就不支持，导致个别测试程序例如 456.hmmer 与 482.sphinx3 损失提升性能的机会，但自动向量化移植优化后申威平台的向量化能力整体优于 X86 平台。

5.4.2 应用程序测试

快速傅立叶变换是一种离散傅立叶变换的高效算法，该算法被广泛应用于信号分析，其性能的提升离不开矩阵运算的优化。另外，图形处理、游戏开发、科学计算中也包含了大量的矩阵运算，实验以矩阵乘为代表，进行自动向量化相关的应用程序的性能测试分析。

程序采用三层循环，以 3 种 N*N 规模进行测试，每个规模运行 3 次取平均值。采用移植优化前后的性能进行对比分析，实验数据如图 5.8 所示。

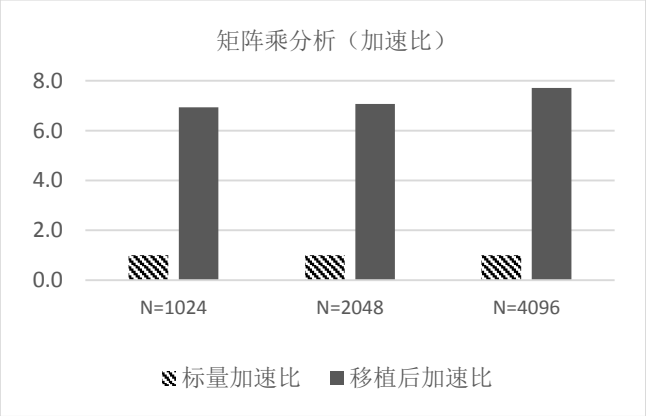


图 5.8　矩阵乘运算分析

自动向量化的移植与优化工作完成后，矩阵乘运算的平均加速比提升达 7.2，矩阵规模为 1024*1024 时加速比达 6.9，在 2048*2048 规模下加速比达 7.1，4096*4096 规模下加速比达 7.7。矩阵乘性能收益主要来源于向量化信息选取了匹配的跨幅因子，收益分析后选择了符合申威处理器的向量化因子，在自动向量化后最大程度地发挥了 SIMD 向量指令优势。

6 结论

6.1 工作总结

随着国产高性能计算芯片的日益成熟，如何构建其计算生态，充分发挥硬件的执行效能已成为当前工作的重点。通过自动向量化有效地利用微处理器底层 SIMD 硬件，是编译器性能提升的重点。本文基于 LLVM 编译器，进行面向国产申威处理器的自动向量化的移植与优化研究，致力于提升申威处理器的向量化能力。

本文的主要工作和贡献有：

（1）介绍了 LLVM 编译器的自动向量化技术。LLVM 编译器中已初步实现了自动向量化，本文从合法性分析、向量发掘、向量代码生成三个方面分别介绍了当前 LLVM 编译器中循环级与基本块级的向量化方法，构建了适用于申威平台的 LLVM 自动向量化框架。

（2）完善了申威平台 SIMD 向量寄存器内容的描述与向量化信息，完成了循环级向量化的移植工作。在此基础上实现了针对控制流向量化改进的两种算法：针对申威平台不支持掩码指令而导致失去控制流向量化的问题，实现了一种掩码指令转换算，解决了因指令集不支持掩码指令而限制了控制流向量化问题；针对控制流向量化方法的单一性问题，实现了一种的 phi 节点优化算法，解决了因无法生成 phi 指令失去控制流向量化机会的问题。

（3）完善了同构语句数约束以及数据重组的指令代价计算，完成了基本块级向量化的移植工作。在此基础上实现两种优化算法：针对基本块级向量化打包同构语句的随机性导致向量代码收益不佳的问题，实现了一种代价评估的打包算法，解决了同构语句打包的随机性问题；针对 LLVM 中向量化发掘能力不足的问题，实现了一种混合优化方法，解决了因发掘能力不足失去向量化机会的。

6.2 展望与计划

通过以上几个方面的研究，自动向量化技术在申威平台得到了较好的实现，针对大部分计算密集的浮点程序具有良好的加速效果。但是应用程序特征

多种多样，现有的向量化发掘算法仍有较大的改进空间，因此在本文工作的基础上，可以从以下几个方面展开后续工作：

（1）本文在基本块级向量化的优化中提出的混合向量化优化算法是基于以存储指令为根节点的向量化方法，但是基本块级向量化中也存在以 phi 指令为根节点的规约向量化。针对此类方法首先需要考虑如何识别，识别之后在何处进行展开。另外，由于规约向量化方法涉及多个基本块，盲目展开后函数结构可能会被破坏，适不适合进行展开也需要考虑到算法中。

（2）本文研究的循环向量化技术目前只针对最内层循环进行了分析，多层循环向量化中由于跨幅访存的出现，导致一些程序放弃了向量化，如果可以将循环交换技术与向量化相结合，便可使得原本在外层不连续的循环交换后在最内层连续。因此可以从循环交换入手，开展外层循环向量化的研究。

（3）通过研究自动向量化的移植工作，发现存在一些相对重要的向量指令在申威后端中是不支持的，这会影响到向量化的发掘能力。因此，考虑对于那些不支持的指令，是否可以利用申威指令集中的基本向量指令进行拼接组合，重组的向量指令是否在原有的基础上具有性能提升，这就需要综合考虑申威指令集与自动向量化后代码生成的关系。

参考文献

[1] 高伟,赵荣彩,韩林,庞建民,丁锐.SIMD 自动向量化编译优化概述[J].软件学报,2015,26(06):1265-1284.

[2] 侯永生. 多重循环SIMD向量化方法及性能优化技术研究[D].[博士学位论文].解放军信息工程大学,2014.

[3] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization[C].In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11). USA: IEEE Press, 2011:141-150.

[4] C. Böhm and C. Plant, Mining Massive Vector Data on Single Instruction Multiple Data Microarchitectures[C].In 2015 IEEE International Conference on Data Mining Workshop (ICDMW), USA:IEEE, 2015: 597-606.

[5] 王琦. 面向非规则应用的SIMD自动向量化编译技术研究[D]. [硕士学位论文].郑州:战略支援部队信息工程大学,2018.

[6] Leaf Petersen, Dominic Orchard, and Neal Glew. Automatic SIMD vectorization for Haskell[C].In Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. USA:Association for Computing Machinery,2013.25–36.

[7] 徐金龙. 面向SIMD的自动并行化关键技术研究[D]. [博士学位论文].郑州:解放军信息工程大学,2015.

[8] Hao Zhou and Jingling Xue. A Compiler Approach for Exploiting Partial SIMD Parallelism[J]. ACM Trans. 2016, Article 11 (April 2016):26-35.

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

[10] J. Parri, J. Desmarais, D. Shapiro, M. Bolic and V. Groza, Design of a custom vector operation API exploiting SIMD intrinsics within Java[C].In CCECE 2010, Calgary:AB, 2010.1-4.

[11] Estérie P, Gaunard M, Falcou J, et al. Boost. simd: generic programming for portable simdization[C].In Proceedings of the 21st international conference on Parallel architectures and compilation techniques. USA:ACM, 2012. 431-432.

[12] Tarjan R. Depth-first search and linear graph algorithms[J]. SIAM journal on computing, 1972, 1(2): 146-160.

[13] D. Nuzman and A. Zaks, Outer-loop vectorization - revisited for short SIMD architectures[C].In 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT), Toronto:ON, 2008. 2-11.

[14] Lokuciejewski P, Stolpe M, Morik K, et al. Automatic Selection of Machine Learning Models for WCET-aware Compiler Heuristic Generation[J]. 2013.

[15] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks and I. Rosen, Polyhedral-Model Guided Loop-

Nest Auto-Vectorization[C].In 2009 18th International Conference on Parallel Architectures and Compilation Techniques, USA:ACM, 2009:327-337.

[16] Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets[M]. UAS:ACM, 2000.6.

[17] VasileiosPorpodas.SuperGraph-SLP Auto-Vectorization [C].In 2017 International Conference on Parallel Architecture and Compilation (PACT). USA:IEEE, 2017. 330-342.

[18] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. Look-ahead SLP: auto-vectorization in the presence of commutative operations[C].In Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018), USA: ACM, 2018.163-174.

[19] Jaewook Shin, Mary Hall, Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow[C]. CGO 2005.

[20] O. V. Moldovanova, M. G. Kurnosov and A. Mel'nikov, Energy Efficiency and Performance of Auto-Vectorized Loops on Intel Xeon Processors[C].In 2018 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC), USA:IEEE, 2018.1-6.

[21] 姚远. SIMD自动向量识别及代码调优技术研究[D]. [博士学位论文].郑州:解放军信息工程大学,2012.

[22] 孙回回,赵荣彩,高伟,李雁冰.基于条件分类的控制流向量化[J].计算机科学,2015,42(11):240-247.

[23] Shin J. Introducing control flow into vectorized code[C].In Parallel Architecture and Compilation Techniques, 2007. 16th International Conference on. USA:IEEE, 2007: 280-291.

[24] PANDEY M,SARDA S.LLVM cookbook [M].Packt Publishing Ltd.,2015.

[25] Xinmin Tian, Hideki Saito, Ernesto Su, Abhinav Gaba, Matt Masten, Eric Garcia, and Ayal Zaks. LLVM framework and IR extensions for parallelization, SIMD vectorization and offloading[C]. In Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC (LLVM-HPC '16).USA:IEEE, 2016. 21-31.

[26] 赵晔. 基于LLVM的交叉编译器的设计与实现[D].[硕士学位论文].西安:西安电子科技大学,2015.

[27] Carlo Bertolli, Samuel F. Antao, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. 2014. Coordinating GPU threads for OpenMP 4.0 in LLVM[C]. In Proceedings of the 2014 LLVM Compiler Infrastructure in HPC. USA:IEEE,2014. 12–21.

[28] F. Petrogalli and P. Walker, LLVM and the Automatic Vectorization of Loops Invoking Math Routines: -FSIMDMATH[C].In 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), USA:ACM, 2018. 30-38.

[29] Harini Srinivasan, James Hook, and Michael Wolfe.Static single assignment for explicitly parallel programs[c]. In Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. USA:Association for Computing Machinery, 1993.260–272.

[30] Allen Leung and Lal George. Static single assignment form for machine code[C]. In Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. Association for Computing Machinery, USA:ACM, 1999.204–214.

[31] Hao Zhou and Jingling Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead[C]. In Proceedings of the 2016 International Symposium on Code Generation and Optimization. USA:ACM, 2016.59-69.

[32] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD[J]. Association for Computing Machinery,2006,6(June 2006):132-143.

[33] De A, D'Souza D. Scalable flow-sensitive pointer analysis for java with strong updates[M].ECOOP 2012–Object-Oriented Programming. Springer Berlin Heidelberg, 2012: 665-687.

[34] Yu H, Xue J, Huo W, et al. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code[C].In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. USA:ACM, 2010. 218-229.

[35] Hardekopf B, Lin C. Semi-sparse flow-sensitive pointer analysis[C].In ACM SIGPLAN Notices. USA:ACM, 2009. 226-238.

[36] Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code[C].In Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on. USA:IEEE, 2011. 289-298.

[37] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences—a method to expedite the evaluation of closed-form functions. In Proceedings of the international symposium on Symbolic and algebraic computation. USA:Association for Computing Machinery, 1994. 242–249.

[38] Eugene V. Zima. On computational properties of chains of recurrences[C]. In Proceedings of the 2001 international symposium on Symbolic and algebraic computation. USA:Association for Computing Machinery, 2001. 345-355.

[39] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization[C]. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. USA:IEEE Computer Society, 2011. 141–150.

[40] Allen J R, Kennedy K, Porterfield C, et al. Conversion of control dependence to data dependence[C].In Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. USA:ACM, 1983. 177-189.

[41] Frances E. Allen. Control flow analysis[C]. In Proceedings of a symposium on Compiler optimization. USA:Association for Computing Machinery, 1970. 1–19.

[42] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A compiler framework for extracting superword level parallelism[C].In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. USA:Association for Computing Machinery, 2012. 347–358.

[43] 高雨辰,赵荣彩,韩林,李雁冰.循环自动并行化技术研究[J].信息工程大学学

报,2019,20(01):82-89.

[44] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. PSLP: padded SLP automatic vectorization[C]. In Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. USA:IEEE Computer Society, 2015. 190–201.

[45] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. VW-SLP: auto-vectorization with adaptive vector width[C]. In Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. USA:Association for Computing Machinery,2018. 1–15.

[46] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity[C]. In Proceedings of the 12th ACM conference on Computer and communications security. USA:Association for Computing Machinery,2005. 340–353.

[47] D. Nuzman and A. Zaks. Autovectorization in GCC – two years later[C]. in the GCC Developer's summit, June 2006:145-158.

[48] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form[C]. Program. Lang. USA:ACM. 1987. 491–542.

[49] r. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution[C]. In Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. Association for Computing Machinery, USA:ACM, 1987. 63–76.

[50] Allen J R, Kennedy K, Porterfield C, et al. Conversion of control dependence to data dependence[C].In Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. USA:ACM, 1983. 177-189.

[51] Shin J, Hall M W, Chame J. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures[J]. Microprocessors and Microsystems, 2009, 33(4): 235-243.

[52] Allen R. Optimizing Compilers For Modern Architectures: A Dependence-based Approach Author: Randy Allen, Ken Kennedy, Publisher: Mo[J]. 2001.

[53] Yao Jinyang, Zhao Rongcai, Wang Qi, and Tao Xiaohan. Loop-nest Auto-vectorization Method Based on Benefit Analysis[C].In Proceedings of the 2nd International Conference on Advances in Image Processing (ICAIP '18). USA:ACM, 2018. 240–244.

[54] 张媛媛. 自动向量化中的收益评估技术研究[D]. [硕士学位论文].郑州:解放军信息工程大学,2011.

[55] Andrew Anderson, Avinash Malik, and David Gregg. Automatic Vectorization of Interleaved Data Revisited[J]. ACM Trans,2016,Article 50 (January 2016): 1-25.

[56] Srikanth Kurra, Neeraj Kumar Singh, and Preeti Ranjan Panda. The impact of loop unrolling on controller delay in high level synthesis[C]. In Proceedings of the conference on Design, automation and test in Europe. EDA Consortium, San Jose, CA, USA:ACM, 2007. 391–396.

[57] Mark Stephenson and Saman Amarasinghe. Predicting Unroll Factors Using Supervised Classification[C]. In Proceedings of the international symposium on Code generation and

optimization. USA:IEEE, 2005. 123–134.

[58] 徐金龙,赵荣彩,韩林.分段约束的超字并行向量发掘路径优化算法[J].计算机应用,2015,35(04):950-955.

[59] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. Vectorization-aware loop unrolling with seed forwarding[C]. In Proceedings of the 29th International Conference on Compiler Construction. USA:Association for Computing Machinery. 2020. 1–13.

致谢

岁月不居，时节如流。三年的研究生时光转瞬即逝，在这三年的学习和生活中，我有过迷茫有过挫折，但也有攻克难题的喜悦和刻苦钻研的收获。值此论文完成之际，谨向在我攻读硕士学位期间指导、帮助、鼓励过我的导师、朋友、亲人们表达最衷心的感激之情。

首先，诚挚的感谢我的导师韩林老师，是他带我走入编译领域的知识海洋。学习工作期间与老师的每次讨论都使我深受启发、获益匪浅，虽然我之前并未接触过课题相关的内容，但老师耐心地教导与鼓励，使我沉下心来找到了适合自己的学习方法，逐渐的掌握了做研究的思路，领略到了编译的魅力。生活上他更是我的人生导师，他的谆谆教诲无时无刻不在激励着我，提醒自己还有许多方面需要努力，他诲人不倦的精神、精益求精的工作作风更是我学习的榜样。在撰写论文期间，老师悉心地指导使论文内容得到了充实和提升。

同时，真诚的感谢高伟老师，是他让我明白了科研的乐趣。在学习上，他不仅引导我学会如何思考问题，如何掌握解决问题的方法，还对我的研究工作提出了很多宝贵意见。高老师待人真诚，幽默风趣，使我在科研学习过程中树立了良好的心态。在他的指导和帮助下，我的课题实践和论文撰写得以顺利完成。

感谢沈莉、尉红梅、何王全、李雁冰、刘璐老师们在出差联调联测期间对我工作以及生活上的帮助，与沈莉老师一起工作更是开阔了我的视野，使我学到了许多编译器相关的底层知识。

感谢夏文博师兄，帮助我迅速融入研究生生活。感谢胡浩师兄、巩令钦师姐，耐心地给我讲解基础知识、梳理思路，手把手地带我学习实践。感谢王洪生师兄在学习期间的照顾，他扎实的基础知识使每次讨论都让我受益匪浅。感谢师兄师姐，在生活以及工作上给予我的帮助。

感谢课题组的黄驻峰，陈梦尧，我们一起奋力改 bug 也一起挨过骂，一起看过故宫傍晚的夕阳也一起走过青砖白瓦的古镇，在那段艰苦学习的时光里感谢你们的陪伴。感谢柴赟达、陈云、王荣勤、朱梦瑶、以及九楼西的小伙伴们，正是大家对我生活以及学习中的关心和帮助，使我度过了愉快难忘的硕士生生涯。

感谢父母、亲人、朋友和最关心我的人，在精神上给予我极大的支持，是他们无私的爱让我有不断进取的动力，感谢他们为我做的一切！

最后，感谢在百忙之中参与论文评阅、评议和参加答辩的各位专家、教授！