计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330058
60(6): 1232−1245, 2023

面向飞腾迈创数字处理器的内核代码自动生成框架

赵宵磊　陈照云　时　洋　文　梅　张春元
（国防科技大学计算机学院　长沙　410073）
（zhaoxiaolei14@nudt.edu.cn）

Kernel Code Automatic Generation Framework on FT-Matrix

Zhao Xiaolei, Chen Zhaoyun, Shi Yang, Wen Mei, and Zhang Chunyuan
（*College of Computer Science and Technology*, *National University of Defense Technology*, *Changsha* 410073）

Abstract　Digital signal processors (DSPs) commonly adopt VLIW-SIMD architecture and facilitate cooperation between scalar and vector units. As a typical VLIW-SIMD DSP architecture, the extreme performance of FT-Matrix DSP relies on highly optimized kernels. However, hand-crafted methods for kernel operator development always suffer from heavy time and labor overhead to unleash the potential of DSP hardware. General-purpose compilers are suffering from poor portability or performance, struggling to explore optimization space aggressively. We propose a high-performance automatic kernel code-generation framework, which introduces the characteristics of FT-Matrix into hierarchical kernel optimizations. The framework has three optimization component layers: loop tiling, vectorization and instruction-level optimization, and can automatically search for optimal tile size according to memory hierarchy and data layout, and further introduce the vectorization with scalar-vector unit cooperation to improve data reuse and parallelism, while some optimization space on collaborating scalar and vector units for specific design in architectures by different vendors is overlooked. The performance of VLIW architecture is determined by instruction-level parallelism (ILP) to a great extent. Moreover, Pitaya provides the assembly intrinsic representation on FT-Matrix DSP to apply diverse instruction-level optimizations to explore more ILPs. Experiments show that kernels generated by Pitaya outperform these from target DSP libraries by 3.25 times and C vector intrinsic kernels by 20.62 times on average.

Key words　kernel code generation; VLIW-SIMD; loop tiling; scalar-vector cooperation; digital signal processor(DSP)

摘　要　数字信号处理器（digital signal processor, DSP）通常采用超长指令字（very long instruction word, VLIW）和单指令多数据（single instruction multiple data, SIMD）的架构来提升处理器整体计算性能，从而适用于高性能计算、图像处理、嵌入式系统等各个领域. 飞腾迈创数字处理器（FT-Matrix）作为国防科技大学自主研制的高性能通用数字信号处理器，其极致计算性能的体现依赖于对 VLIW 与 SIMD 架构特点的充分挖掘. 不止是飞腾迈创系列，绝大多数处理器上高度优化的内核代码或核心库函数都依赖于底层汇编级工具或手工开发. 然而，手工编写内核算子的开发方法总是需要大量的时间和人力开销来充分释放硬件的性能潜力. 尤其是 VLIW+SIMD 的处理器，专家级汇编开发的难度更为突出. 针对这些问题，提出一种面向飞腾迈创数字处理器的高性能的内核代码自动生成框架（automatic kernel code-generation framework on FT-Matrix），将飞腾迈创处理器的架构特性引入到多层次的内核代码优化方法中. 该框架包

收稿日期：2023-01-10；**修回日期**：2023-04-10
基金项目：国家自然科学基金项目（62002366）
This work was supported by the National Natural Science Foundation of China (62002366).
通信作者：陈照云（chenzhaoyun@nudt.edu.cn）

括 3 层优化组件：自适应循环分块、标向量协同的自动向量化和细粒度的指令级优化. 该框架可以根据硬件的内存层次结构和内核的数据布局自动搜索最优循环分块参数，并进一步引入标量-向量单元协同的自动向量化指令选择与数据排布，以提高内核代码执行时的数据复用和并行性. 此外，该框架提供了类汇编的中间表示，以应用各种指令级优化来探索更多指令级并行性（ILP）的优化空间，同时也为其他硬件平台提供了后端快速接入和自适应代码生成的模块，以实现高效内核代码开发的敏捷设计. 实验表明，该框架生成的内核基准测试代码的平均性能是目标—数字信号处理器（DSP）——的手工函数库的 3.25 倍，是使用普通向量 C 语言编写的内核代码的 20.62 倍.

关键词 内核代码生成；超长指令字-单指令多数据；循环分块；标量-向量协同；数字信号处理器

中图法分类号 TP314

数字信号处理器（digital signal processor, DSP）通常采用超长指令字（very long instruction word, VLIW）和单指令多数据（single instruction multiple data, SIMD）的体系结构设计[1-3]. VLIW 和 SIMD 提高了在处理器上执行各种应用程序时的计算性能、实时处理能力和能效比. 作为具有该典型架构的通用高性能的 DSP 之一，飞腾迈创数字处理器（FT-Matrix）利用了灵活的标向量硬件单元协同与多层次并行性挖掘来提高处理器的整体性能与效能[1].

然而，绝大多数处理器上极致性能的释放依然依赖于高度优化的内核代码[2]，例如核心算法库等. 由于通用编译器对于特定的体系结构指令或硬件优势的支持有限，不能很好地平衡可编程性和能效比，因此极致优化的内核代码通常采用手工汇编级优化或基于底层汇编级的工具开发. 然而，手工编写优化内核级代码需要大量的时间和劳动成本，尤其是对于 VLIW+SIMD 架构的处理器，如何充分挖掘数据重用与潜在的并行性是一个重大的挑战.

一般地，一名资深工程师可能会耗费几个月的时间来对一个新的内核函数进行针对硬件的深度优化. 因此，面向以飞腾迈创为代表的 VLIW+SIMD 处理器架构，设计一个高性能内核代码自动生成工具是必要的. 这可以将开发人员从繁重的内核代码优化和函数库开发的负担中解放出来.

目前，针对 CPU 和 GPU 等通用处理器和加速器上的代码自动优化和生成已是领域的研究热点. 传统的代码优化手段（如重复访存和内联等）已不是新兴的优化手段中的重点，这些手段在现有的 DSP 编译中均已有应用，但不是本文关注的重点. 循环优化如多面体[3]，自动向量化如 VW-SLP[4] 等工作均对代码优化和生成提供了新思路，但此类优化更加注重代码的上层优化，如算子级和循环级的代码优化工作. 如果不考虑软件兼容问题，循环优化如文献 [3−4] 中的方法也可直接用于 DSP 的代码优化，且文献 [3−4] 中的工作考虑更多的是 SIMD 并行性，并没有完全发掘出 VLIW 架构的特性，除此之外 VLIW+SIMD 的体系结构的特异性带来的底层优化问题需要更加繁杂的移植和适配工作，文献 [3−4] 中的方法未有针对硬件后端的指令级优化，优化完成后仍需要工程师对代码进行指令静态调度、软流水等一系列手段以发挥 VLIW 架构的最佳性能，这些底层优化手段对于 DSP 来说同样重要.

与用于特定 AI 加速器的张量编译器[3] 和用于具有全新 SIMT 编程模型 GPU 的 CUDA/OpenCL 工具包[5] 不同，DSP 缺少高级计算调度语言的支持，DSP 应用编译时的核心算子映射逻辑与其硬件参数的匹配对于性能释放具有很大影响. 设计用于通用 DSP 的内核代码自动生成工具时需要同时考虑可编程性和能效比，以释放 VLIW+SIMD 架构的潜力. 然而，到目前为止，关于 DSP 方向的有关研究大都集中在体系结构设计和极致提高能效比上，而未能详细考虑其软件兼容性和面向 DSP 的代码优化[1]. 文献 [6] 提出了一种通过平等性饱和方法[7] 来进行自动向量化，从而面向 DSP 的内核代码进行优化的编译器 Diospyros，但是该方法对内核代码规模有限制，且无法实现端到端的代码生成支持. 因此，实现一个面向 VLIW+SIMD 架构的完整的内核代码生成框架仍然面临 3 点挑战：

1）考虑到内存层次结构和容量的限制，设计人员通常使用乒乓缓冲机制来拆分和处理大规模的输入数据. 然而，为不同内核找到一组最佳的循环分块参数以最大限度地提高其在飞腾迈创或其他目标平台上的数据重用是具有挑战性的. 这需要开发人员深入了解内核代码执行时硬件的细节和特征. 自动循环分块可以将程序员从手动调节循环分块参数和重复的迭代优化工作的负担中解放出来.

2)为了利用处理器执行 SIMD 指令时闲置的标量单元，实现标量单元和向量单元的工作负载均衡是必要的. 目前大多数研究只关注充分利用硬件上的向量单元的自动向量化技术，而忽略了不同供应商在体系结构中设计的标量单元和向量单元之间进行协作的优化空间.

3) VLIW 体系结构的性能很大程度上受到指令级并行性(instruction-level parallelism, ILP)的影响. 现有的通用编译器虽然提供了部分 VLIW 的支持，但对于以飞腾迈创处理器为代表的 VLIW+SIMD 架构来说，其后端支持比较有限，尤其是指令级优化和快速代码生成，这些优化的支持可以提高内核代码的指令级并行度，增强代码的可移植性并减轻手工为内核编写汇编代码的工作负担.

为了解决这 3 点挑战，本文提出了一种面向飞腾迈创数字处理器的高性能内核代码自动生成框架(automatic kernel code-generation framework on FT-Matrix). 该框架由 3 个不同粒度的优化组件层构成，通过分层编译优化来提高内核代码在飞腾迈创上的数据重用和并行性. 这些分层优化包括带反馈的自动循环分块、标量-向量协同的自动向量化和细粒度指令级优化，其结构如图 1 所示. 通过利用自动循环分块，该框架可以根据硬件的内存层次结构确定最佳循环分块参数和内核数据布局. 对于每个分块，利用标量-向量协同可以提高自动向量化效率并减少其产生的数据移动. 此外，该框架还提供了细粒度的指令级优化，以提高内核代码执行时的指令级的并行性.

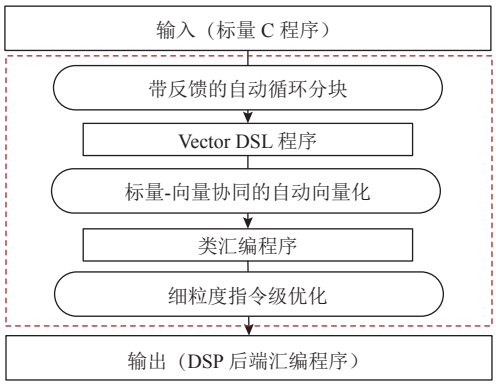


Fig. 1 DSP kernel code-generation framework with three layers of optimization structure

图 1 具有三层优化结构的 DSP 内核代码生成框架

本文的主要工作是为 DSP 平台提供了一个将内核代码由高级语言到类汇编的底层指令优化生成过程的框架，框架中实现的循环分块算法和自动向量化模块也可以更换为其他算法，用户完全可以自行进行配置. 此框架实现了将多层优化过程进行组合和自动调优，减少了开发人员的额外工作，生成的代码可以直接在硬件上高效执行.

实验表明，面向飞腾迈创的内核代码自动生成框架的编译性能是人工开发的优化库函数的 3.25 倍，是支持向量 Intrinsic 指令的 C 代码编写的内核函数的 20.62 倍.

本文的主要贡献包括 4 个方面：

1)提出了一种面向飞腾迈创数字处理器的内核代码自动生成框架. 该框架具有 3 个优化组件层：带反馈的自动循环分块、标量-向量协同的自动向量化和细粒度指令级优化. 框架通过分层优化生成高度优化的内核代码，提高了其在飞腾迈创上的数据重用和并行性.

2) 本文设计的框架提供了带自反馈的自动循环分块算法，可以根据输入内核程序自动搜索最优的循环分块参数，能够减轻人工手动进行代码调优的负担，提高内核数据的复用性.

3) 为了提高向量化效率，本文提出了一种新的标量单元和向量单元之间协同的自动向量化技术，减少由激进的向量化策略导致的 SIMD 体系结构上的数据移动开销.

4) 本文设计的框架自动生成类汇编代码，能够支持细粒度的指令级优化. 类汇编能够为生成高效的内核代码提供诸多便利，扩大了指令级并行优化的探索空间，并可以通过外部指令集描述文件适配其他硬件平台，使开发人员减少额外的移植工作，为应用开发提供敏捷设计.

1 背　　景

1.1 飞腾迈创处理器体系结构

以飞腾迈创处理器为代表的 DSP 处理器大多采用 VLIW 和 SIMD 体系结构. 以飞腾迈创架构为例，如图 2 所示，指令派发组件同时向标量单元(SU)和单指令多数据单元(SIMDU)发射指令[1]. SU 中的标量存储单元(SM)支持标量内存访问. 向量处理单元(VPU)由 16 个同构的向量处理引擎(VPE)组成[8]，每个 VPE 包含 3 个浮点乘加器(MAC)和 2 个向量数据访存单元. 内核中的阵列存储器(AM)可实现 16 路的向量数据访问. 同时，VLIW 体系结构需要在静态编译过程中进行超长指令字的指令并行打包，才能发挥其架构特性.

1.2 向量化

由于通常不会激进地探索极其复杂的数据移动策略，向量化技术(从循环依赖性分析到自动向量化)

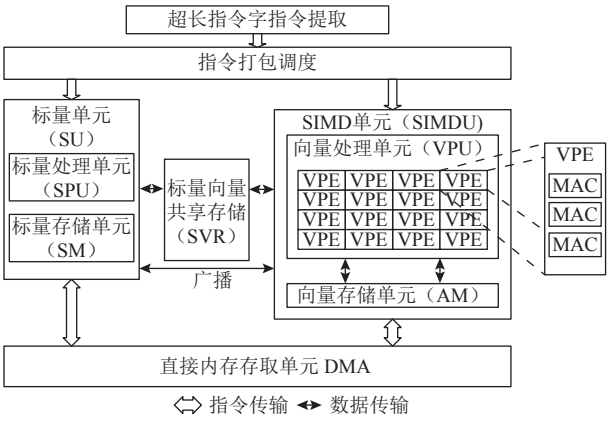


Fig. 2 Architecture of FT-Matrix

图 2 FT-Matrix 架构

会错过一些向量化优化的可能性. 平等性饱和利用等价图(e-graph)来实现先进的由重写规则驱动的编译优化[9]. 文献 [6] 利用平等性饱和来避免手工寻找特定的向量化模式, 并提供了比自动调优(auto tuning)更大的搜索空间覆盖范围. 虽然 e-graph 是一个强大的自动向量化工具, 但它同时也需要高昂的时间和存储开销来处理中、大规模的循环输入代码, 对于通用内核代码优化不具有普适性. 本文也借鉴了 e-graph 技术手段, 但补充了额外的分块支持与标向量协同技术, 弥补了其方法的弱点, 进一步提高了并行性挖掘.

2 框架设计

本文提出的内核代码自动生成框架探索并提高了内核代码在飞腾迈创上的数据重用和并行性, 以充分发挥硬件的利用效率. 本节首先介绍了整体框架的结构. 然后, 分别介绍了框架中的各层结构及其优化方法, 包括带反馈的自动循环分块、标量-向量协同的自动向量化和细粒度指令级优化.

2.1 框架概述

该内核代码自动生成框架具有 3 层不同粒度的优化层级, 其结构如图 3 所示.

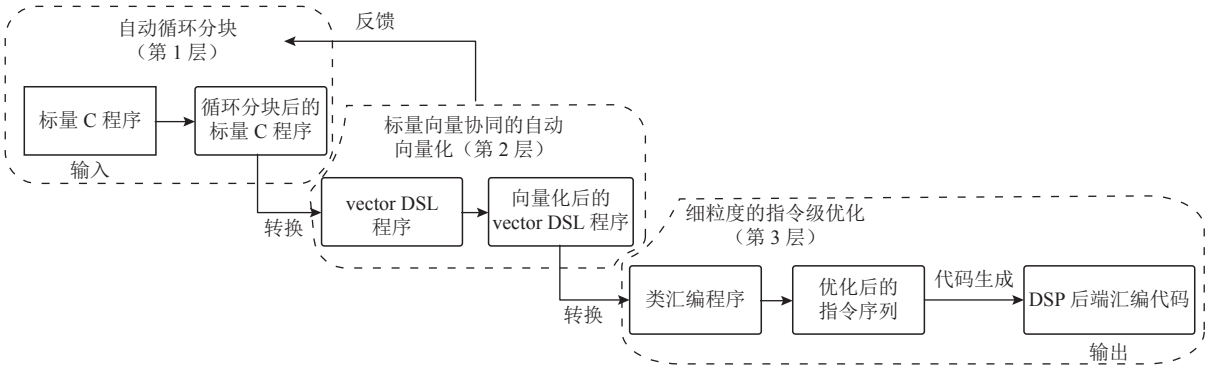


Fig. 3 Overview of our framework structure and compilation process

图 3 本文框架结构和编译过程总览

整个框架的输入是未经任何优化的标量 C 程序, 这避免了其他代码生成工具需要用户将代码转换为特定领域语言(domain-specific language, DSL)的过程. 基于输入的标量程序, 自动循环分块算法会分析输入的循环维度参数的比例, 计算出相应的循环分块参数值, 经自动向量化的优化和反馈, 进行多次迭代和更新以搜索到循环分块的最优解. 该层也支持其他循环分块算法的部署与扩展. 此优化完成后该框架会将经循环分块后标量 C 程序自动转换为一种 DSL 语言编写的中间表示(intermediate representation, IR), 称为向量领域特定语言(vector DSL).

标量-向量协同的自动向量化能够探索每个分块的向量化, 可优化空间并生成优化后的 vector DSL 程序, 利用闲置的标量单元以减少内核中的数据移动开销. 该层设置了代价函数来判断自动向量化的优化效果, 该代价函数值也将返回至上一个优化层, 以指导自动循环分块对循环参数进行调整, 直到总代价达到收敛状态.

在循环分块和向量化之后, 该框架将代码转换为目标平台后端特定的 IR, 如飞腾迈创后端的类汇编语言. 基于此 IR, 该框架可以部署细粒度的指令级优化, 如指令调度、软件流水线等, 可以在飞腾迈创上探索更多的指令级并行优化空间.

2.2 带反馈的自动循环分块

循环分块对内核代码在硬件平台上的数据重用具有显著影响. 内核的最佳循环分块参数(通常由专家手动调整)可以提高内核在硬件上执行时的计算效率和内存读写效率.

多面体[3]技术是一种统一化的程序变换表示技术,它通过迭代域、仿射调度和访存函数来表示代码,有利于表示程序变换的组合,有助于解决包含循环程序的向量化问题.但该技术不适合应用在带反馈的自动调优框架中,原因有3点:1)调整的灵活性.本文框架设计要求反馈机制能够迅速发挥作用,建模的循环分块可以根据反馈结果快速计算出下一次迭代过程的分块规模,能够根据自动向量化的结果进行灵活调整.多面体则需要根据迭代域和仿射关系的调整来进行循环级优化.2)适配所需的工作繁杂.由于DSP后端支持薄弱,通用的高层次的编译优化手段需要根据硬件架构特性进行细粒度的适配工作,以保障该手段能够有效发挥作用.3)架构特性带来的优化空间.多面体面向通用处理器有较好的优化效果,但对于标向量协同架构特性带来的并行性发掘有限,如飞腾架构的标向量协同指令带来的并行性优化.分块算法和指令选择是面向DSP代码生成的重要一环,通过框架设计,利用朴素的循环分块算法和平等性饱和工具进行标向量协同的向量化的搜索,可以更好地利用架构中的一些特殊指令,如飞腾的广播指令来利用标量和向量协同工作,提高硬件的利用效率.

此外,一些关于内核代码生成的研究,如文献[6],由于其优化过程的时间和存储成本高昂,只能处理小规模的输入.为了充分利用硬件特性并减轻手动优化的负担,设计了一种带有代价反馈的自动循环分块算法来实现此过程.

本文根据循环中各维度的数据重用率和内存限制条件建立了一个分块规模选择(tile size selection, TSS)模型[10].根据TSS模型,一个维度为i, k, j的二维矩阵乘法在未经任何优化时,初始化R_i: R_k: R_j = 1: 2: 1(每次计算中,矩阵$\boldsymbol{A}$和矩阵$\boldsymbol{B}$的维度K的数据重用了2次).循环分块的大小$T=(i,k,j)$在满足内存限制的条件下根据此数据重用比例的反比进行初始化设置.

经过循环分块后的内核程序段将被转换为vector DSL代码段后进行自动向量化的优化过程.然而,探索SIMD并行性可能会导致各维度数据重用率发生变化.那么,T需要在向量化后进行更新.如图4所示,最初,矩阵$\boldsymbol{A}$第1行中的一个元素应乘以矩阵$\boldsymbol{B}$第1列中的每个元素,数据重用的情况在图4中用竖纹块标记.通过自动向量化优化后,该优化结果将矩阵$\boldsymbol{A}$中的元素利用标量单元进行广播操作,以提高数据的复用率.如图4中的斜纹块所示,此时,矩阵$\boldsymbol{A}$中每一行的每个元素(维度K)被重用4次.然后,将数据重用比例相应地更新为$R_i:R_k:R_j=1:4:1$,并重新进行循环分块.自动向量化提供一个代价值函数$cost$进行反馈,该函数值表示当前循环分块大小下vector DSL代码段在硬件上的执行周期值,反映了此循环分块代码段优化的效果.同时,更新后的数据重用比例指导算法进行循环分块参数T的调整.

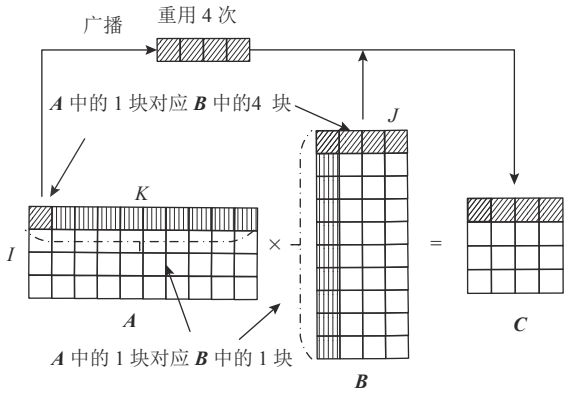


Fig. 4 Data reuse before and after applying broadcast instruction

图4 应用广播指令前后的数据重用

自动循环分块的过程如算法1所示.算法1根据初始比例选择循环各维度的分块大小,并根据此大小切分循环(行⑤~⑥).通过标量-向量协同优化,算法得到优化后的vector DSL代码的代价值$cost$和数据复用情况的变化(行⑧).算法1根据比例调整分块大小,并自动返回到分块阶段(行⑥).自动调优过程将进行迭代,直到总代价值$Cost_{\text{total}}$收敛(行④).算法1的最终结束状态为向量化的反馈无法改变数据重用情况且向量化后循环分块的总代价值收敛至最小值.

算法1. 带反馈的自动循环分块算法.

输入:输入程序P,总内存容量M;

输出:循环维度切分参数的集合T.

① 分析输入程序P;

② 初始化循环参数维度集合$\{I,K,J,\cdots\}$和数据重用集合$R=\{R_i,R_k,R_j,\cdots\}$;

③ 初始化代价函数$cost$和总代价$Cost_{\text{total}}$;

④ while $Cost_{\text{total}}$不收敛

⑤ 根据循环维度、数据重用和片上存储容量M计算数据重用比例R_i: R_k: R_j: …和分块规模大小$T=(i,k,j,\cdots)$;

⑥ 根据分块规模T将输入程序P切分成N块;

⑦ 将分块后的程序转换为vector DSL并进行向量化;

⑧ 根据优化后的 vector DSL 程序获得代价函数值 *cost* 和数据重用 *R*;

⑨ 更新 $Cost_{total} = cost \times N$;

⑩ end while

⑪ return *T*.

2.3 标量-向量协同的自动向量化

为了提高内核代码的数据复用和并行性，采用向量化技术能够最大限度地提高计算效率和硬件利用率. 在内核代码生成框架中，本文采用了一种 vector DSL 作为自动向量化的中间描述[6].

内核代码生成通常采用的向量化技术，包括循环依赖分析等，仅关注向量单元. 类似地，文献 [6] 旨在向量单元之间实现灵活地数据移动，却忽略了闲置的标量单元的作用. 另外，其数据移动的实现依赖于硬件提供的混洗(shuffle)指令，然而混洗指令虽然通用灵活，但是通常其硬件执行开销也较大，因此依赖混洗指令的方法生成的内核代码性能在该硬件上大打折扣，甚至出现负优化的情况[11].

如何释放标量单元与向量单元协同的潜力来增强内核代码中的数据重用和并行性是本节设计基于飞腾迈创处理器的自动向量化策略的主要挑战.

如图 5 所示，带有标向量硬件单元的 DSP 供应商通常提供的交互指令(包括 broadcast, reduce, scatter, gather, allreduce 等)可以显著提高标量和向量单元之间的数据移动效率[1]. 为了释放这部分潜力，在代码生成过程中支持这些交互指令和重写规则显得尤为重要.

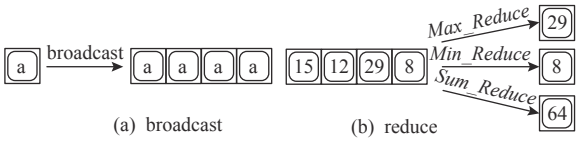


Fig. 5 Typical ISA-specific scalar-vector cooperation instructions

图 5 典型的特定指令集架构中的标量和向量单元协作指令

本文将这些特定指令的重写规则扩充至平等性饱和[7]工具中进行，以探索标量—向量协同指令为导向的自动向量化空间的搜索.

以图 6(a)中的标量 C 程序为例，它表示一个 1×4 的矩阵 ***A*** 乘以 4×4 的矩阵 ***B***. 该程序可以转换为图 6(b)中的未经优化的 vector DSL 代码段，并以此作为自动向量化的输入对象.

平等性饱和工具可以依据扩充的等价规则来重写 vector DSL 代码，通过替换相同操作的不同指令选择来生成最优的内核代码. 例如，*VecMac*( )节点的功能与 *VecAdd*(*VecMul*( ))节点的功能相同，均可以表示向量的乘加操作. 因此，平等性饱和工具可以在图 6(c)中找到相应的等效表示并生成指令替换后的优化代码.

```
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    c[i] += a[j] × b[4 × j + i]
```

(a) 标量 C 程序

```
(Concat VecMac (Vec (a0, a1, a2, a3)
                    Vec (b0, b4, b8, b12))
               VecMac (Vec (a0, a1, a2, a3)
                    Vec (b1, b5, b9, b13))
               VecMac (Vec (a0, a1, a2, a3)
                    Vec (b2, b6, b10, b14))
               VecMac (Vec(a0, a1, a2, a3)
                    Vec (b3, b7, b11, b15)))
```

(b) vector DSL 对 (a) 的转换

```
(Trans
(VecAdd VecMul (Vec (a0, a0, a0, a0)
                    Vec (b0, b1, b2, b3))
               VecMul (Vec (a1, a1, a1, a1)
                    Vec (b4, b5, b6, b7))
               VecMul (Vec (a2, a2, a2, a2)
                    Vec (b8, b9, b10, b11))
               VecMul (Vec (a3, a3, a3, a3)
                    Vec (b12, b13, b14, b15))))
```

(c) 经转置规则的 vector DSL 程序

```
(Trans
(VecAdd VecMul (Broadcast(a0)
                    Vec (b0, b1, b2, b3))
               VecMul (Broadcast(a1)
                    Vec (b4, b5, b6, b7))
               VecMul (Broadcast(a2)
                    Vec (b8, b9, b10, b11))
               VecMul (Broadcast(a3)
                    Vec (b12, b13, b14, b15))))
```

(d) 经广播规则的 vector DSL 程序

Fig. 6 An example of rewrite rule application with the broadcast instruction

图 6 广播指令重写规则应用的示例

但是，*Vec*( )节点可能包含大量无序数据，这些数据会导致生成的代码中具有大量的 shuffle 指令来进行数据移动，从而导致目标硬件产生高昂的数据移动开销. 这一现象存在可优化的空间，可以利用闲置的标量单位来减少此开销.

我们补充了特定内置指令的重写规则，以利用标量—向量的协作来减少数据移动. 以广播指令为例：

$$Vec(a, a, a, a) \sim> Broadcast(a)$$

广播只需要一次数据加载，就可以减少冗余的内存访问并减少访存开销. 我们根据硬件不同指令的周期在代价函数中调整每个可替换指令的代价函数值. 此外，还为用户提供了一个接口，以根据目标硬件上的指令周期来调整代价函数，为其他硬件实现敏捷开发提供便利.

该过程的搜索效率体现在可通过有限时间得到向量化搜索的结果，具体数据如表 1 所示. 平等性饱和搜索是指令选择优化技术中的一种，其根据用户提供的重写规则来对等价的指令进行替换而得到新

的指令序列，并根据代价函数来评估经指令选择后代码序列的性能. 我们定义的等价规则包括乘加指令、移位指令、转置指令、数据拼接指令和体系结构支持的特殊指令等，等价规则包含了结合律和分配律. 搜索空间即由这些等价指令及其等价规则组合形成. 此外，我们未在等价规则中配置交换律，交换操作数在指令级别没有优化意义，以此裁剪了平等性饱和搜索的搜索空间. 同时，带性能反馈的自动向量过程也提供了有选择性的分块规模参数，此过程也裁剪了大部分搜索迭代过程.

Table 1 Benchmark Programs Adopted in Experimental Evaluation

表 1 实验评估中采用的基准程序

基准程序	数据规模/浮点数	迭代次数	编译时间/ min	未经分块模块处理的编译时间/ min
自相关（AutoCor）	10	1	0.13	0.13
	16	1	0.17	0.17
	256	107	89	
	512	131	113	
快速傅里叶变换（FFT）	4	1	0.08	0.08
	8	1	1.35	1.35
	16	1	7	7
	32	1	12	12
	128	4	22	30
	256	5	35	
	512	5	35	
	1 024	5	35	
二维卷积（2D Conv）	4×4, 2×2	1	0.07	0.07
	16×16, 2×2	1	0.23	0.23
	256×256, 4×4	25	121	
	512×512, 4×4	25	121	
矩阵乘法（Mat Mul）	4×4×4	1	0.07	0.07
	8×8×8	1	0.17	0.17
	16×16×16	1	1.72	1.72
	256×256×256	63	189	
	512×512×512	67	202	

代价函数能够准确反馈平等性饱和工具搜索的指令选择序列在硬件上的执行周期，从而能够帮助其确定最佳的代码重写和优化方式. 代价函数的返回值为指令序列中所有指令的执行周期的总和. 我们为各类指令分别预设了周期值，并且提供了接口，支持用户对特殊指令的周期值进行设置(例如飞腾支持较好的广播指令等). 对于 VLIW 架构的 DSP 硬件后端来说，静态排布后指令序列的执行周期为确定值，用户可直接对代价函数进行计算. 代价函数的返回值也会发回循环分块算法以指导自动循环分块过程. 在该例中，vector DSL 中的表达式与模式 *Vec*(*a*, *a*, *a*, *a*)匹配并用 *Broadcast*(*a*)重写它. 由于广播指令在飞腾迈创处理器上的执行代价很低，最终的内核代码通过平等性饱和搜索和标量-向量协同优化，优先选择广播指令进行替换，能够在飞腾迈创上实现最短的执行周期，如图 6(d)所示.

2.4 细粒度指令级优化

由于 VLIW 架构在很大程度上依赖于静态指令级优化，该优化层提供了一个将 vector DSL 代码转换为类汇编代码的组件来为飞腾迈创或其他供应商提供敏捷开发的支持. 如图 7 所示，类汇编指令参照基本的 VLIW 汇编指令格式[12−13]，但省略了需要手工排布的一些繁琐信息，如并行符号和功能单元信息，并且条件域和操作数列表也可由变量来代替物理寄存器. 这样在编写代码时可以不考虑指令延迟时隙、VLIW 打包、寄存器分配等细节优化问题，适合作为细粒度代码的优化和生成的输入代码格式，这些细节优化问题将交由本组件自动完成适配与生成.

该组件的优化方式主要包括指令调度和自适应代码生成. 类汇编的中间描述对于各供应商提供的细粒度指令优化方式具有一定的通用性，用户可以根据自身需求来对其中具体的优化算法进行替换或优化. 本节以飞腾迈创后端支持较好的指令调度和软件流水优化算法为例来简要说明该组件的基本结构与优化过程.

1)指令调度

面向 VLIW 后端的现代编译器一般采用列表调度（LS）算法或其变体进行指令调度，但是指令调度是一个典型的 NP 难题[14]. LS 算法的局限性在于寻找调度解决方案的效率不佳，得到的是局部最优解决

Fig. 7 Formats of assembly instructions and assembly intrinsic instructions

图 7 汇编指令与类汇编指令的格式

方案而不是全局最优解. 本文的指令调度采用了一种基于动态规划的指令调度策略(DPS)[15]来获得VLIW指令级调度的近似最优解，同时兼顾指令排布的并行、功能单元与读写端口的冲突等，以探索更多用于目标硬件体系结构的指令级并行优化空间. 除了指令调度外，针对VLIW架构处理器，本文框架还引入了一些循环优化手段，可以对指令级并行进行更加深入的挖掘，例如循环展开、软件流水等. 尤其是以软件流水为代表的优化手段可以大幅度提升内核代码的执行效率. 图8展示了一个经过软件流水优化后的内核代码的指令排布示意图.

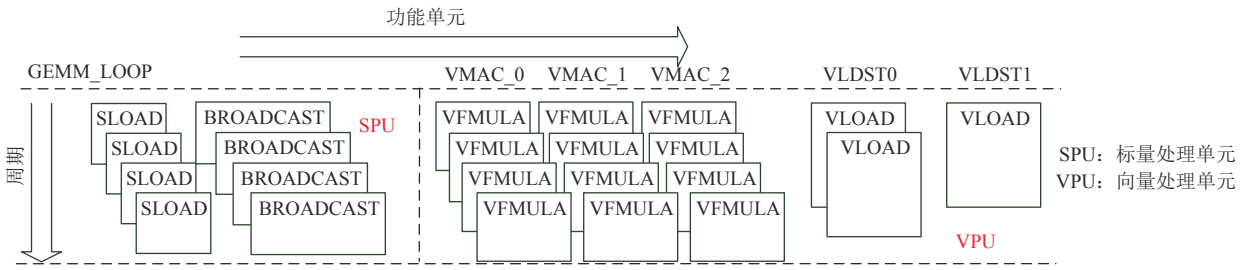


Fig. 8 An example of instruction arrangement in kernel code after software pipeline optimization

图8 经过软件流水优化后的内核代码的指令排布示例

2)快速和自适应的代码生成

在完成一系列优化之后，本文的代码生成框架将内核代码从类汇编代码转换为最终硬件支持的汇编代码. 飞腾迈创指令集内部的类汇编指令的一个简单的代码段如图9所示.

```
BROADCAST    F32 %2, VF32 %5
LOAD         VADDR %0，VF32 %6
LOAD         VADDR %1，VF32 %7
MULA         VF %6，VF %7，VF %5，VF %8
 ⋮
```

Fig. 9 A code segment example of assembly intrinsic instruction

图9 类汇编指令的一个代码段示例

一般地，代码生成与供应商各自的指令集架构(ISA)和目标硬件后端紧密耦合. 为了打破枷锁，实现面向不同硬件后端的自适应代码生成过程，本文提出了一个利用外部ISA描述文件进行代码快速生成的方式，为VLIW后端(包括飞腾迈创)提出了一种敏捷设计. 通过分析发现，本文提出的架构和VLIW架构的指令集具有一定程度的相似性，将其中与最终指令并行排布相关的指令信息进行了抽取，并将之整合到一个外部描述文件(类似于编译器后端的TD文件，但是其格式内容相对更加简洁友好)，其格式如图10所示. 不同的VLIW处理器后端只需要提供此指令集描述文件，就可以直接嵌入到最终代码生成的框架中. 基于外部ISA描述文件，代码生成框架可以获取到所有必需的ISA信息，并能够将类汇编代码自动降低到飞腾迈创或其他VLIW硬件上的后端固有的指令集，以及能够支持VLIW架构的硬件平台，受限于可获得的硬件平台，该框架目前在飞腾迈创系列的两代DSP芯片产品上得到性能验证，两代产品之间的功能单元数量和指令节拍均有所区别，但本文提出的框架可以快速兼容和适配. 同时，本文在4.5节对硬件参数进行了模拟和性能评估以保证性能的可移植性.

指令名称	功能单元	指令周期	读操作数	写操作数	读端口占用周期	写端口占用周期

Fig. 10 Format of instruction information in external ISA description file

图10 外部ISA描述文件的指令信息格式

3 本文框架的实现

本文代码自动生成框架的实现基于以飞腾迈创处理器为代表的VLIW和SIMD体系结构. 该框架的实现包含500行以上的Python代码，用于自动循环分块算法和其反馈迭代过程. 同时，扩展了文献[6]中的自动向量化工具，使用了大约1 000行的Racket和Rust语言的代码，用于标量、向量协同的自动向量化模块的实现. 最后，本文实现了一个包含6 000多行基于C++语言的类汇编优化器，用于细粒度的指令级优化. 本文框架可以实现飞腾迈创处理器系列芯片上内核级代码的自动优化与生成.

4 实验评估

本节从3个方面评估本文代码生成框架的性能：1)对于小规模数据(<256浮点数)，本文框架能

否实现更好的内核代码执行性能？

2)对于中等规模的数据(≥256浮点数)，本文框架能否发掘最佳循环分块大小并生成与硬件厂商提供的DSP函数库相媲美的高性能代码？

3)通过单独应用该框架中的每个组件，内核代码能够分别实现多少性能提升？

4)对于不同的硬件平台和参数，本文框架能否生成匹配且高效的内核代码？

4.1 实验设置

本文在运行Ubuntu 16.04的带有Intel i7-8700CPU的机器上运行基于飞腾迈创的代码生成框架，并将该框架生成的内核程序与文献[6]生成的程序、朴素标量(Naive scalar)程序、朴素向量(Naive vector)程序和DSP函数库(DSPLIB)程序分别部署在飞腾迈创处理器(具有16个向量处理单元(VPE)，每个VPE中有3个乘加器(MAC)单元且同时支持2个向量的访存操作，寄存器宽度为32b)上进行比较. 本文框架也可以支持其他VLIW+SIMD架构的硬件处理器，但是受限于实验条件，目前可获得的真实硬件环境主要以迈创系列处理器为主，但引入额外的模拟环境来验证本文工作的可扩展性.

因Diospyros[6]处理中等规模的数据的时间过长、所需内存过大，只在小规模内核中对其进行评估. 朴素标量程序是具有朴素标量C语言循环的标量内核程序，其在执行时只具备编译器的默认优化. 朴素向量程序是DSP厂商提供的，由C语言+向量Intrinsic指令编写而成的程序版本，该朴素向量程序适用于飞腾系列的DSP芯片，其编写了内核函数的计算过程，考虑并进行了面向架构的特定优化，在适当节点利用了broadcast等飞腾支持较好的指令. DSP函数库是由供应商提供的高性能函数库，它以内核程序在硬件上的实际执行周期作为评估标准来反映各个程序的真实性能，此指标由飞腾迈创后端工具链提供的性能统计工具来进行反馈.

4.2 内核函数基准测试程序

表1列出了本文采用的内核基准测试程序，这些基准测试来自于机器学习和信号处理中广泛应用的内核程序.

自相关(AutoCor)是DSP中广泛使用的信号处理函数. 快速傅里叶变换(FFT)被大量用于各种DSP应用. 二维卷积(2D Conv)和矩阵乘法(Mat Mul)均是机器学习领域最常用的算子.

对于每种内核函数，我们均采用2种数据维度. 采用小规模数据来评估标量-向量协作的向量化模块和细粒度指令级优化模块的优化效果；中等规模的数据用于评估自动循环分块和整个代码生成框架对于内核代码在飞腾迈创上的优化效率.

表1还列出了每个基准测试程序的总编译时间. 本文提出的代码生成框架对于小规模的内核编译只需要进行1次迭代，时间在2min以内.

从测试基准程序来看，小规模程序的框架与Diospyros中向量化所需的时间基本持平，由于循环分块模块的处理，小规模框架可以搜索到Diospyros中向量化无法直接处理的数据规模的优化结果，且搜索时间为迭代次数与分块后小规模程序的搜索时间的乘积. 我们也限制了单次搜索的最长时间，超时后此过程所产生的结果是一种次优解，其性能峰值与理论最优解相近，以此策略来保证搜索效率与搜索结果的平衡. 实验结果也证明了在实际应用中，限定时间内的搜索结果可以获得较好的性能提升.

对于中等规模的AutoCor, FFT和2D Conv内核函数，本文框架只需要约120 min来搜索最佳的循环分块参数；此外，由于Mat Mul的循环具有三维搜索空间，框架在生成代码时具有最大的开销，需要大约200 min. 另外，将每次迭代的时间限制为10 min，以避免每次迭代耗时过长的情况. 在该限制条件下，本文框架生成的代码(一种次优解)与最优情况下的代码在性能差距和耗时差距上的表现处于可接受的范围[6].

4.3 内核性能结果和分析

中、小规模下4种内核程序的执行加速比如图11~15所示. 本文所提到的加速比由真实值以2为底的对数函数计算得出. 本文将朴素标量程序的执行周期设置为对比基准. 本文框架实现的平均加速比为14.01，DSPLIB的平均加速比为4.77，Diospyros实现的平均加速比为3.57(且仅限小规模内核).

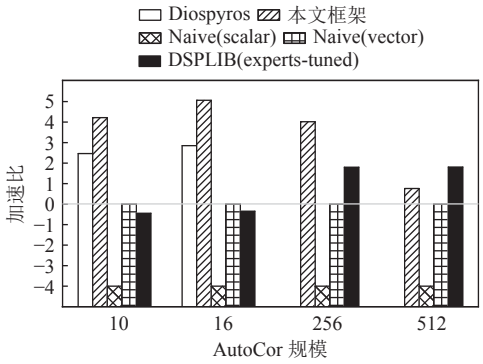


Fig. 11 Speedup ratios of AutoCor kernels

图11 AutoCor内核的加速比

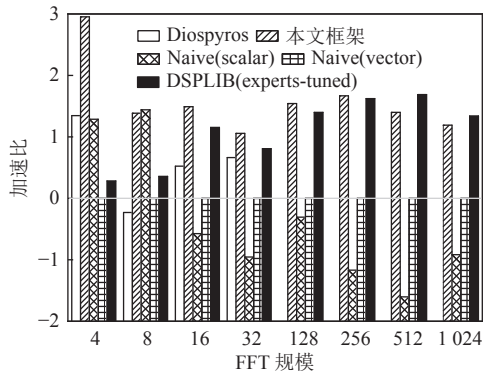


Fig. 12 Speedup ratios of FFT kernels

图 12 FFT 内核的加速比

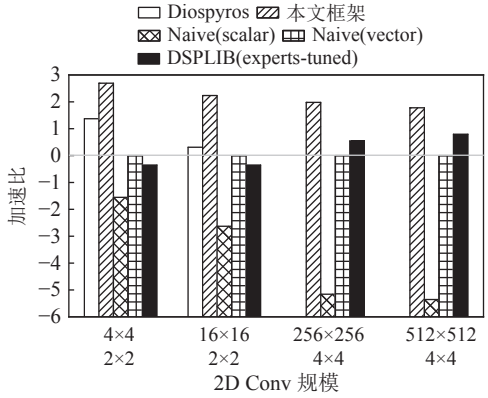


Fig. 13 Speedup ratios of 2D Conv kernels

图 13 2D Conv 内核的加速比

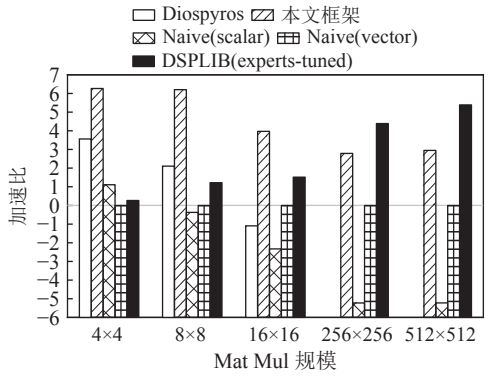


Fig. 14 Speedup ratios of Mat Mul kernels

图 14 Mat Mul 内核的加速比

对于小规模内核程序，Diospyros 严重依赖 shuffle 指令，而飞腾迈创对于该指令需要额外的执行周期来进行数据的重排，这反映在 Mat Mul 的内核程序测试结果中，由 Diospyros 生成的代码包含许多 shuffle 指令用以实现向量化，而本文的框架使用飞腾迈创支持更好的广播指令，并实现了最优的结果. 对于小规模内核程序，本文框架的平均性能是 Diospyros 的 7.33 倍，具体从 FFT 中的 1.31 倍到 Mat Mul 中的

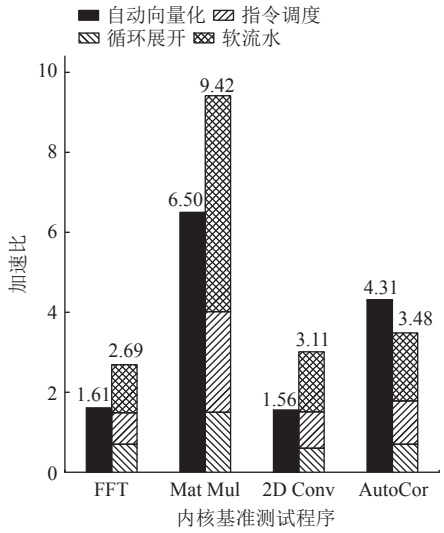


Fig. 15 Average speedup ratios provided by vectorization and instruction-level optimization in breakdown experiment

图 15 分解实验中向量化和指令级优化分别提供的平均加速比

33.39 倍不等. 这些增益来自于标量-向量协作的向量化和细粒度的指令级优化. 当内核函数的数据规模很小时，朴素向量程序（Naive vector）仍需进行完整的向量指令调用和实现，且无法完全发挥指令级优化的作用（如指令调度和软流水的空间被缩减），导致其内核函数的性能没有朴素的标量实现的性能优异. 此类现象也体现在 4×4 的 Mat Mul 内核函数的实验结果中（4×4 Mat Mul）. DSP 函数库是基于最佳数据规模（通常是大规模）进行设计，以发挥硬件最优性能. 对于小规模的内核函数，也需要调用完整的 DSP 库函数来执行，导致其对小规模内核的支持并不友好. 本文框架对于小规模内核的平均性能是 DSP 函数库的 17.56 倍. 本文框架由于充分利用了标量单元和向量单元的协作，并同时进行了细粒度指令级优化，对于小规模内核函数的优化性能在 4 者中最佳.

对于中等规模的内核程序，由于繁重的时间和存储开销，Diospyros 无法处理这些内核. 由于全局优化和专家在某些情况下手动调整部署的数据分块，DSP 函数库表现突出，硬件资源的占用率最高. 但 Diospyros 移植性差，在硬件参数更改时整个函数库均需要重写，这需要额外的人工和时间成本.

受搜索空间和时间的限制，本文的框架在某些情况下未能找到最佳循环分块大小，并且忽略了循环分块之间的优化空间. 但是，该框架生成的内核代码标量单元和向量单元协作的指令（如 broadcast 和 reduce）进行数据整齐的加载来减少存储和数据移动

开销. 该框架仍然可以为中等规模的内核函数实现大约 5.06 的平均加速比，这与专家手工调优的 DSP 函数库代码相比十分具有竞争力. 本文框架生成的内核代码在中等规模下与 DSP 库函数代码相比实现了平均 2.34 的加速比，其中 AutoCor 实现了 2.55 的加速比，并且对 FFT 保持了接近 1 的加速比.

从深层次的角度来看，本文框架可以克服 Diospyros 和 DSP 函数库的弱点. 但是它也有其自身的局限性，在某些情况下，为了增强可编程性和可移植性，它减少了寻找最优解的时间代价，但也牺牲了潜在的加速性能. 本文框架能够充分利用 DSP 硬件的架构特征，对于 DSP 中的典型应用场景，如信号和图像处理应用，以及卷积和矩阵运算的应用都可通过该框架获得加速，并在发挥硬件极端性能和提高开发效率之间得到了一种平衡.

4.4 分解评估

为了分析本文框架中每个组件的性能提升效果，本文设置了分解实验来探索每个组件对于内核基准测试函数的加速比的影响.

表 1 最后 2 列的编译时间对比说明了自动循环分块的效率. 如果没有自动循环分块算法，本文框架便无法生成中等规模以上的内核函数代码(规模大于 256).

自动向量化和细粒度的指令级优化的效果如图 15 所示. 在本文的基准测试中，标量—向量协作的向量化与这些内核函数的原始向量化策略相比可实现 3.49 的平均加速比. 此外，该框架部署的细粒度的指令级优化策略与未采用指令级优化的后端相比，可以实现 4.67 的平均加速比. 这充分说明了本文框架中每层优化均对生成的内核代码起到了加速作用. 同时，3 层优化的组合框架，特别是带反馈的优化过程要优于 3 种优化手段的简单叠加.

4.5 扩展性评估

为了证明本文框架对于其他硬件平台的支持性和扩展性，我们在不同的硬件参数下生成内核代码，模拟其性能来检验其是否可以生成正确的内核代码和能够取得优良的执行性能.

表 2 列出了该框架在 3 组不同硬件参数下生成的内核代码推算的 2 组加速比，向量加速比的基准为向量宽度为 1，VPE 中有 1 个 MAC 单元的抽象硬件环境；标量加速比的基准为原始的标量程序.

与向量和标量基准代码相比，随着向量宽度和 MAC 单元数量的增加，本文框架生成的内核代码的加速比逐步提高，呈现出加速比随硬件资源的增加呈正比上升，验证了本文框架的可拓展性. 在 FFT 中，由于计算中的取数过程并不完全与边界对齐并且存在更多标量操作，硬件参数的增长对其加速效果远没有其余 3 种计算对齐的应用明显. 另外，向量程序采用向量接口编程，可以直接调用高效的向量指令. 相对于标量程序编译时需要指令选择的优化，向量宽度和 MAC 数量均为 1 的硬件环境下向量程序仍可具有一定的加速收益，例如向量乘加指令、各类访存指令等.

Table 2 Hardware Parameters and Speedup Ratios of Generated Kernel Code of Our Framework

表 2 本文框架的硬件参数以及生成的内核代码的加速比

基准程序	规模/浮点数	向量宽度/b	MAC数量	向量加速比	标量加速比
AutoCor	512	1	1	1	2.24
		16	2	10.67	23.90
		32	4	56.70	126.72
		64	4	146.54	328.26
FFT	512	1	1	1	1.58
		16	2	19.32	30.53
		32	4	28.98	45.8
		64	4	56.74	89.65
2D Conv	512×512	1	1	1	11.70
	4×4	16	2	17.94	209.91
		32	4	69.9	817.93
		64	4	84.47	988.33
Mat Mul	512×512	1	1	1	24.01
	512×512	16	2	28.67	688.36
		32	4	116.82	2 804.74
		64	4	216.12	5 189.13

以表 2 中的 64 b 向量宽度的硬件模拟平台为例(向量宽度为 64 b，每个 VPE 中有 4 个 MAC 单元，寄存器宽度为 32 b)，对于一个 8×512×256 的矩阵乘法应用，该框架生成的内核代码指令排布如图 16 所示.

此推算说明本文框架完全可以根据硬件参数的变化生成在不同硬件平台上的高效内核代码，具有拓展性，可以为其他自主芯片后端设计提供自动化代码生成的敏捷设计.

5 相关工作

文献 [16] 设计了一种可移植的编译器，用于以独立于体系结构的方式自动生成内核代码，这种方法更侧重于单独的内核计算方式分析，而未能考虑

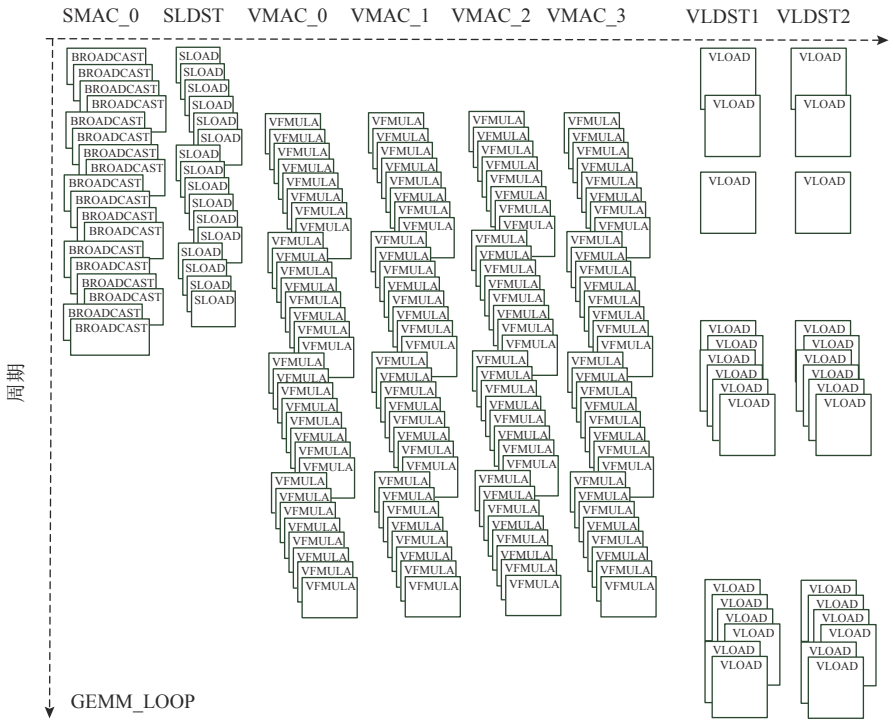


Fig. 16 An example of instruction arrangement of kernel code after software pipeline optimization

图 16 经过软件流水优化后的内核代码的指令排布示例

一般应用在通用体系结构下的适配性. 文献 [17] 为目标 DSP 应用提供了一种 DSL, 但未能探索标量-向量单元协作的优化空间, 使 VLIW 和 SIMD 架构的性能潜力无法完全释放. 文献 [18] 专注于优化稀疏算子, 并从高级语言出发生成代码.

文献 [19–20] 采用了不规则的数据移动策略进行向量化. 它们利用了 SIMD 的特性, 但不支持针对 VLIW 架构的细粒度指令级优化. 文献 [3, 21] 着眼于在通用硬件(如 CPU 和 GPU)上部署深度神经网络, 采用的方法是将张量表达式语言降低到特殊的 DSL 以采用激进的优化策略.

文献 [22] 可以在异构硬件上进行算子编译的自动调优, 具有良好的可移植性, 并且其为各种算子开发了高性能的实现方式. 但是其仍然需要程序员手动编写计算调度模板.

文献 [23] 提出了一种使用修剪和机器学习技术为 CPU, GPU 和 FPGA 上的张量计算生成高性能调度方案的自动优化框架, 无需考虑硬件平台细节. 但是, 对于自主设计的 DSP, 由于缺少后端的支持, 此技术的移植也将是一个繁重的工程.

6 结 论

本文提出了一种具有 3 层优化组件层的基于飞腾迈创处理器的内核代码自动生成框架. 通过带反馈的自动循环分块、标量-向量协同的自动向量化和细粒度指令级优化, 显著提高了内核代码在飞腾迈创上的数据复用和并行性. 实验表明, 本文框架能够自动为飞腾迈创处理器生成深度优化、执行高效的内核代码. 同时, 本文框架也为其他目标硬件平台提供了一种代码生成框架的敏捷设计, 通过替换外部 ISA 描述文件, 能够生成相应平台的深度优化的内核汇编代码.

作者贡献声明: 赵宵磊提出主要研究思路、完成实验并撰写论文; 陈照云指导研究进程、修改和审核论文; 时洋协助修改论文; 文梅和张春元提供研究方向和论文意见.

参 考 文 献

[1] Chen Shuming, Wang Yaohua, Liu Sheng, et al. FT-matrix: A coordination-aware architecture for signal processing[J]. IEEE International Symp on Microarchitecture, 2013, 34(6): 64−73

[2] VanHattum A, Nigam R, Lee V T, et al. Vectorization for digital signal processors via equality saturation[C] //Proc of the 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. Nem York: ACM, 2021: 874−886

[3] Zhao Jie, Li Bojie, Nie Wang, et al. AKG: Automatic kernel

generation for neural processing units using polyhedral transformations[C] //Proc of the 42nd ACM SIGPLAN Int Conf on Programming Language Design and Implementation. New York: ACM, 2021: 1233−1248

[4] Porpodas V, Rocha R C O, Góes L F W. VW-SLP: Auto-vectorization with adaptive vector width[C] //Proc of the 27th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2018: 1−15.

[5] Zhao Xiaolei, Wen Men, Chen Zhaoyun, et al. Automatic mapping and code optimization for OpenCL kernels on FT-matrix architecture (WIP paper)[C] //Proc of the 22nd ACM SIGPLAN/SIGBED Int Conf on Languages, Compilers, and Tools for Embedded Systems (LCTES'21). ACM, 2021: 37−41

[6] Vanhattum A , Nigam R , Lee V T , et al. A synthesis-aided compiler for DSP Architectures (WiP Paper)[C] //Proc of the 21st ACM SIGPLAN/SIGBED Conf on Languages, Compilers, and Tools for Embedded Systems (LCTES '20). New York: ACM, 2020: 131−135

[7] Tate R, Stepp M, Tatlock Z, et al. Equality saturation: A new approach to optimization[C] //Proc of the 36th annual ACM SIGPLAN-SIGACT Symp on Principles of programming languages. New York: ACM, 2009: 264−276

[8] Liu Sheng, Lu Kai, Guo Yang, et al. A self-designed heterogeneous accelerator for exa-scale high performance computing[J]. Journal of Computer Research and Development, 2021, 58(6): 1234−1237 (in Chinese)
(刘胜, 卢凯, 郭阳, 等. 一种自主设计的面向E级高性能计算的异构融合加速器[J]. 计算机研究与发展, 2021, 58(6): 1234−1237)

[9] Willsey M, Nandi C, Wang Y R, et al. Egg: Fast and extensible equality saturation[J]. Proceedings of the ACM on Programming Languages, 2021, 5(POPL): 1−29

[10] Narasimhan K, Acharya A, Baid A, et al. A practical tile size selection model for affine loop nests[C] //Proc of the ACM Int Conf on Supercomputing. New York: ACM, 2021: 27−39

[11] Chen Shuming, Sheng Liu, Wan Jianghua, et al. Coordinate multi-core DSP YHFT-QMBase: Architecture and implementation[J]. SCIENTIA SINICA Informationis, 2015, 45(4): 560−573 (in Chinese)
(陈书明, 刘胜, 万江华, 等. 协同多核DSP YHFT-QMBase: 体系结构及实现[J]. 中国科学: 信息科学, 2015, 45(4): 560−573)

[12] Texas Instruments. C66x CPU and instruction set reference guide [EB/OL]. 2010 [2023-01-12]. https://www.ti.com/lit/pdf/sprugh7

[13] Cadence Design Systems. Cadence tensilica fusion G6 [EB/OL]. 2022 [2023-01-12]. https://www.cadence.com/zh_CN/home/tools/ip/tensilica-ip/fusion-dsps/fusion-g3-g6.html

[14] Hartmanis J. Computers and intractability: A guide to the theory of NP-Completeness[J]. SIAM Review, 1982, 24(1): 90−91

[15] Deng Can, Chen Zhaoyun, Shi Yang, et al. Exploring ILP for VLIW architecture by quantified modeling and dynamic programming-based instruction scheduling[C] //Proc of the 27th Asia and South Pacific Design Automation Conf (ASP-DAC). Piscataway, NJ: IEEE, 2022: 256−261

[16] Su Xing, Liao Xiangke, Xue Jingling. Automatic generation of fast BLAS3-GEMM: A portable compiler approach[C] //Proc of the 2017 IEEE/ACM Int Symp on Code Generation and Optimization (CGO). Piscataway, NJ: IEEE, 2017: 122−133

[17] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[C]// Proc of the 34th ACM SIGPLAN Conf on Programming language design and implementation. New York: ACM, 2013: 519–530

[18] Kjolstad F, Kamil S, Chou S, et al. The tensor algebra compiler[J]. Proceedings of the ACM on Programming Languages, 2017, 1(OOPSLA): 1−29

[19] McFarlin D S, Arbatov V, Franchetti F, et al. Automatic SIMD vectorization of fast Fourier transforms for the Larrabee and AVX instruction sets [C] //Proc of the Int Conf on Supercomputing. New York: ACM, 2011: 265−274

[20] Franchetti F, Püschel M. Generating SIMD vectorized permutations[C] //Proc of the Int Conf on Compiler Construction. Springer, Berlin: Springer, 2008: 116−131

[21] Chen Tianqi, Moreau T, Jiang ziheng, et al. TVM: An automated end-to-end optimizing compiler for deep learning[C] //Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI 18). Berkeley, CA: USENIX Association, 2018: 578−594

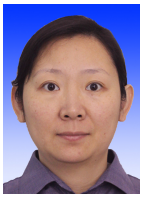
[22] Chen Tianqi, Zheng Lianmin, Yan E, et al. Learning to optimize tensor programs [C] //Proc of the 32nd Int Conf on Neural Information Processing Systems. New York: ACM, 2018: 3393−3404

[23] Zheng S, Liang Y, Wang S, et al. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system[C] //Proc of the 25th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 859−873

Zhao Xiaolei, born in 1996. PhD. His main research interests include systematic compiler and computer architecture.
赵宵磊, 1996 年生. 博士. 主要研究方向为系统编译和计算机体系结构.

Chen Zhaoyun, born in 1990. PhD, assistant professor. His main research interests include systematic compiler and computer architecture.
陈照云, 1990 年生. 博士, 助理研究员. 研究方向为系统编译与计算机体系结构.

Shi Yang, born in 1992. PhD. His main research interests include parallel and distributed computing, resource management, and network scheduling
时　洋, 1992 年生. 博士. 主要研究方向为并行与分布式计算、资源管理和网络调度.



Wen Mei, born in 1975. PhD, professor. Member of CCF. Her main research interests include computer architecture, parallel programming, and scientific computing

文　梅, 1975 年生. 博士, 教授. CCF 会员. 主要研究方向为计算机体系结构、并行编程以及科学计算.

Zhang Chunyuan, born in 1964. PhD, professor. Member of CCF. His main research interests include computer architecture, parallel programming, embedded systems, and scientific computing.

张春元, 1964 年生. 博士, 教授. CCF 会员, 主要研究方向为计算机体系结构、并行编程、嵌入式系统和科学计算.