本 科 毕 业 论 文

课题名称：基于 **RISC-V** 存算一体芯片的编译器关键技术研究

学 员 姓 名：简泽鑫　　学　　号：202102001019

首次任职专业：无　　学历教育专业：计算机科学与技术（计算机系统）

命 题 学 院：计算机学院　　年　　级：2021 级

指 导 教 员：曾坤　　职　　称：副研究员

所 属 单 位：计算机学院微电子与微处理器研究所

国防科技大学教育训练部制

目　录

摘 要

SRAM 存算一体架构可以有效减少数据的无效搬运，具备突破冯诺依曼架构瓶颈的潜力。由于 RISC-V 指令集具有开源开放、扩展性强等优势，已经逐渐成为构建存算一体芯片的首选。然而，基于 RISC-V 构建的存算芯片具有异构、碎片化的特点，这就要求开发者面向不同存算架构开发多个版本的应用，开发效率低、部署难。因此，如何实现将软件操作（包括计算、数据通信等）和硬件配置（如异构计算单元、存储层次等）解耦，以便 AI 应用开发不再依赖存算 IP 设计，是解决“编程墙”的关键问题。

不仅如此，AI 应用的不规则的发展趋势和存算芯片的异构化、碎片化的现状，使得我们需要探索新的动态编译优化方法，这种优化方法既需要能够充分的考虑到 AI 应用的动态变化的特质，又需要能够充分的挖掘未来存算芯片的架构特征。通过动态编译优化，可以实时调整编译策略，使得生成的代码能够更好地适应硬件的运行环境，提高计算效率和资源利用率。

因此本研究致力于解决上述难题，构建一个可以面向 RISC-V SRAM 存算一体芯片进行自动后端优化的编译器系统。编译器对应用程序进行应用特征分析，识别出可以加速的计算部分，并转为特定的 RISC-V 加速指令，充分利用 RISC-V 已有指令集实现在执行 AI 任务运行时对各类计算资源的灵活调度，充分发挥 SRAM 存算一体阵列高能效、高算力密度的硬件优势。

通过测试表明，本文实现的编译器能够对预训练神经网络模型进行优化、将应用算子自动映射到具有不同 IP 设计的加速部件，根据不同芯片架构特征生成正确的指令流来协调各个计算部件，挖掘芯片内部的计算并行性以及基于目标体系结构的代码生成。

关键词：深度学习编译器；LLVM 编译器；调度器；存算一体

ABSTRACT

The SRAM *Computing-in-Memory* (CIM) architecture can effectively reduce the ineffective transfer of data and has the potential to break through the bottleneck of the von Neumann architecture. Due to the advantages of open source and strong scalability of the RISC-V instruction set, it has gradually become the first choice for building computing-in-memory chips. However, the computing-in-memory chips built on RISC-V are heterogeneous and fragmented, which requires developers to develop multiple versions of applications for different storage-computing architectures, which is inefficient and difficult to deploy. Therefore, how to decouple software operations (including computing, data communication, etc.) and hardware configurations (such as heterogeneous computing units, storage levels, etc.) so that AI application development no longer relies on computing-in-memory IP design is the key issue in solving the “programming wall”.

Not only that, the irregular development trend of AI applications and the heterogeneous and fragmented status of computing-in-memory chips require us to explore new dynamic compilation optimization methods, which need to fully consider the dynamic characteristics of AI applications and fully explore the architectural characteristics of future computing-in-memory chips. Through dynamic compilation optimization, the compilation strategy can be adjusted in real time, so that the generated code can better adapt to the hardware operating environment and improve computing efficiency and resource utilization.

Therefore, this study is committed to solving the above problems and building a compiler system that can automatically optimize the backend for RISC-V SRAM CIM chips. The compiler analyzes the application characteristics of the application, identifies the computing parts that can be accelerated, and converts them into specific RISC-V acceleration instructions, making full use of the existing RISC-V instruction set to realize the flexible scheduling of various computing resources when executing AI tasks, and giving full play to the hardware advantages of high energy efficiency and high computing power density of SRAM storage and computing integrated arrays.

Experimental results shows that the compiler implemented in this paper can optimize the pre-trained neural network model, automatically map the application operator to the acceleration component with different IP designs, generate the correct instruction stream according to the characteristics of different chip architectures to coordinate the various

computing components, explore the computing parallelism inside the chip, and generate code based on the target architecture.

KEY WORDS: Deep Learning Compiler, LLVM Compiler, Scheduler, Computing in Memory

第 1 章　绪论

1.1　研究背景与意义

深度学习作为机器学习领域中一个非常重要的分支，近几年来发展迅速，在计算机视觉[1]、自然语言处理[2]、电子商务[3]和药物研发[4]等领域都取得了显著的成果。随着卷积神经网络（CNN）[5]、循环神经网络（RNN）[6]、长短期记忆网络（LSTM）[7]和生成对抗网络（GAN）[8]等多种深度学习模型的不断涌现，简化各种深度学习模型的编程对于实现其更广泛的应用至关重要。

与此同时，随着人工智能算法复杂度呈指数级增长，以及物联网终端设备产生的数据量也突破 ZB 量级，传统的计算架构正面临前所未有的“双重困境”：一方面，受限于冯诺依曼架构中存储单元与计算单元的物理分离特性，在数据处理过程中，处理器与存储器之间需要不断地通过数据总线交换数据。处理器性能以每 2 年 3.1 倍的速度增长，而内存性能以每 2 年 1.4 倍的速度提升，导致存储器的数据访问速度越来越跟不上处理器的数据处理速度。处理器的性能与效率因此受到严重制约，从而出现了“存储墙”；另一方面，在冯诺依曼架构中，数据在处理过程中需要不断地从存储器单元“读”数据到处理器单元中，处理完之后再将结果“写”回存储器单元。数据在片外存储与运算核心之间的频繁迁移带来了严重的传输功耗问题。根据英特尔的研究显示，在半导体工艺进入 7nm 时代时，数据搬运功耗高达 35pJ/bit，占比达到 63.7%。数据传输所导致的功耗损失越来越成为芯片发展的制约因素。在 AI 计算平台上，面对海量的数据，“存储墙”和“功耗墙”的问题愈发用凸显，成为整个计算平台的掣肘。

在此严峻形势之下，存算一体（Compute-In-Memory，CIM）架构应运而生，为突破传统架构限制带来了全新的希望与解决方案。该架构的核心创新之处在于将计算功能巧妙地集成于存储单元之中，从根源上显著减少了数据传输的庞大体量，从而成功突破了长期以来困扰业界的冯诺依曼瓶颈，实现了系统性能以及能效的大幅提升与飞跃，为计算架构领域开辟了全新的发展方向与路径。

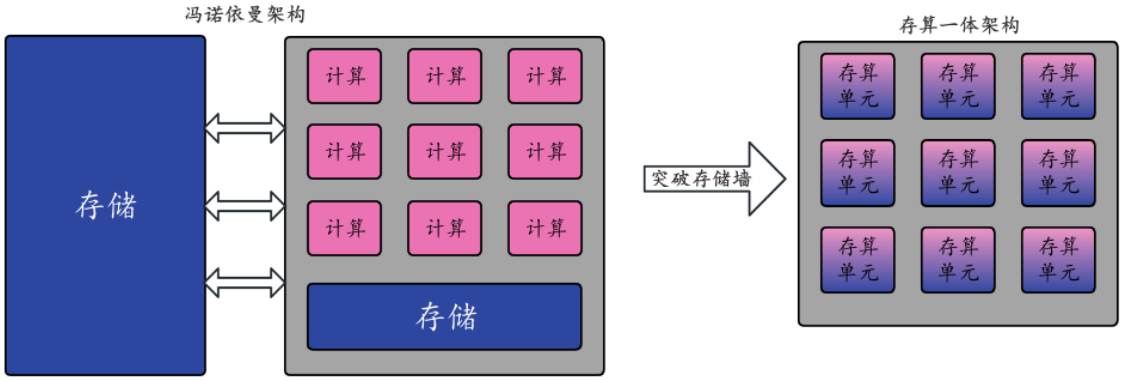

图 1.1　冯诺依曼 v.s. 存算一体架构

同时，基于 RISC-V 指令集构建存算一体芯片逐渐成为 AI 加速器主流。一方面，RISC-V 指令集具有高度开放、标准化等优势，搭配上模块化设计以及强大且卓越的可定制性优势，适合用领域定制的芯片开发和设计。另一方面，可以借助 RISC-V 国际社区的力量，通过 RISC-V 扩展标准化的推动和发展促进统一、高效的 AI 编程模型和系统软件支撑框架的形成。因此，谷歌、脸书、微软等巨头，都基于 RISC-V 指令集搭建自有 AI 芯片。2023 年 RISC-V SoC 的市场渗透率达到 2.6%，市场规模 61 亿美元，并正在持续上升。

然而，人工智能深度领域定制的趋势导致了 RISC-V 存算一体芯片异构化、碎片化的特征。一方面，存算一体芯片本身具有异构性。芯片包含加速矩阵运算的张量核心以及加速向量计算的向量核心等加速部件。另一方面，虽然不同机构均基于 RISC-V 指令集进行芯片设计，但是不同机构的存算一体芯片具有迥异的架构特征，导致内部互联、存储器的存取方式等设计各不相同，造成了碎片化，给用户编程、程序优化带来了显著挑战，开发者在面向不同存算架构进行应用开发时，往往不得不针对每一种特定架构开发多个不同版本的应用程序，不仅极大地降低了开发效率，还使得应用部署过程变得异常艰难与繁琐。鉴于此，如何巧妙地实现软件操作（涵盖了计算过程、数据通信等关键环节）与硬件配置（诸如异构计算单元、存储层次结构等复杂要素）之间的深度解耦，从而使得 AI 应用开发能够摆脱对存算 IP 设计的高度依赖，已然成为破解当前“编程墙”困局的关键所在与核心突破口。

基于上述现状与需求，本课题将聚焦于面向 RISC-V 存算一体芯片的编译支持工作，构建一个可以面向 RISC-V SRAM 存算一体芯片进行自动后端优化的编译器系统，使其能够有效支持存算指令的执行与处理。具体而言，编译器对应用程序进行应用特征分析，识别出可以加速的计算部分，并转为特定的 RISC-V 加速指令，以自动将它们卸载到 CIM 加速器的 NPU 核心进行计算计算，同时运行时在 CPU 和 NPU 异构计算单元之间进行指令的动态调度，充分利用 RISC-V 通用核心、NPU 加

速核心的高效能特征，为 RISC-V 存算一体芯片构建起坚实可靠的编译支持体系，助力其在实际应用中发挥出卓越的性能表现。

1.2 国内外研究现状

1.2.1 深度学习编译器

随着深度学习技术的日臻成熟，深度学习编译器领域也随之迎来了快速发展的浪潮。在这一阶段，出现了许多具有代表性的深度学习编译器：

TensorFlow XLA（Accelerated Linear Algebra/加速线性代数）[9]：Google 于 2017 年开发的一个专门针对特定领域的线性代数编译器，旨在加速 AI 框架下 TensorFlow 中的计算过程，核心思想是通过对计算图进行优化和编译，以实现更高效的计算。其接收来自 PyTorch、TensorFlow 和 JAX 等 ML 框架的模型，在中间优化层级，XLA 包括整体模型优化，如简化代数表达式、优化内存数据布局和改进调度等等。但是 XLA 主要针对 TensorFlow 优化，对其他框架的支持可能需要额外的工作；同时，其主要面向 GPU 和谷歌的 TPU，其中间表示为深度学习算子级别的抽象，使其难以拓展到 RISC-V 存算一体加速器。

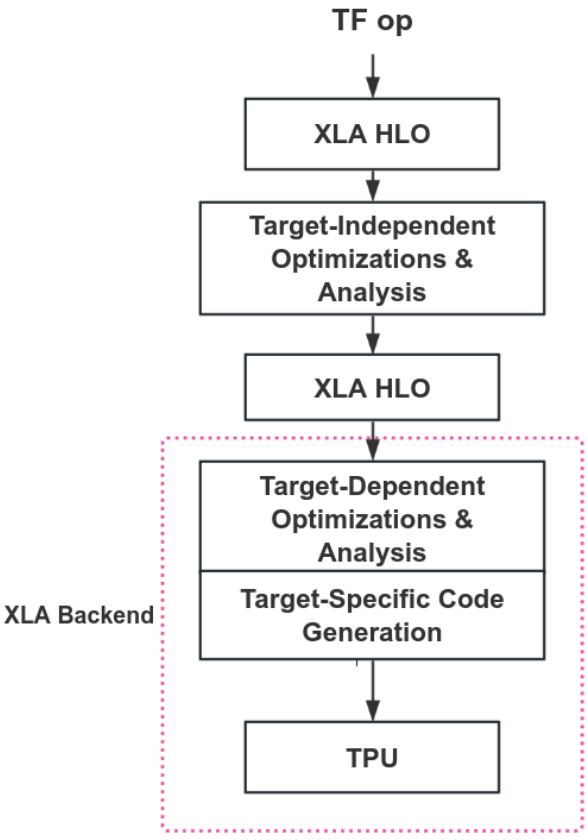

图 1.2 XLA 架构图

TVM（Tensor Virtual Machine）[10]：华盛顿大学陈天奇团队于 2018 年提出的开源的深度学习编译器堆栈，旨在通过对神经网络模型的端到端优化，使其在各种硬件平台上高效执行。其以 TensorFlow、PyTorch 或 ONNX 等 ML 框架导入模型，将模型编译为可链接对象模块，然后轻量级 TVM Runtime 可以用 C 语言的 API 来动态加载模型，也可以为 Python 和 Rust 等其他语言提供入口点。在中间优化层级，其提出了 Relay IR[11] 和 Tensor IR[12] 两层中间表示来进行硬件无关（如常数折叠、算符融合等）、硬件相关（如计算模式识别与加速指令生成等）的优化。TVM 可以自动为多种硬件（包括 CPU、服务器 GPU、移动端 GPU 以及基于 FPGA 的加速器）来生成优化代码，支持端到端的学习优化，并且具备灵活的编译流程，但是 TVM 在面对新型加速器时，不但需要开发者根据芯片指令去扩展 TVM 中的 IR，还需要根据芯片的体系结构设计去添加定向优化策略，导致其扩展性较为有限。

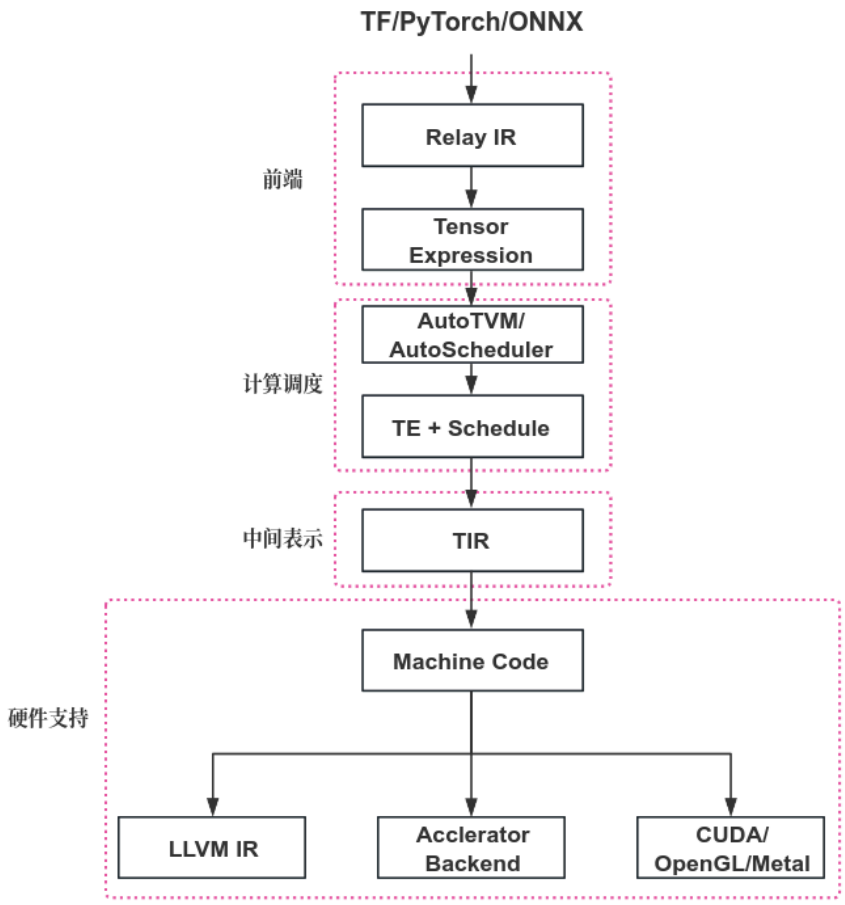

图 1.3　TVM 架构图

MindSpore AKG[13]：作为由华为主导开发并集成于其开源深度学习框架 MindSpore 中的深度学习编译器框架。AKG 可以接收来自 ML 框架的模型，生成针对特定硬件优化的内核。在中间优化层级，AKG 通过自动性能调优工具，自动生

成优化的内核。同时 AKG 提供了自动化的调优过程，可以显著提高性能。然而，目前 AKG 主要针对华为的昇腾系列 AI 加速器和英伟达的 GPU 进行了优化支持，对于 RISC-V 存算一体异构芯片，其支持程度相对有限，适配性欠佳。

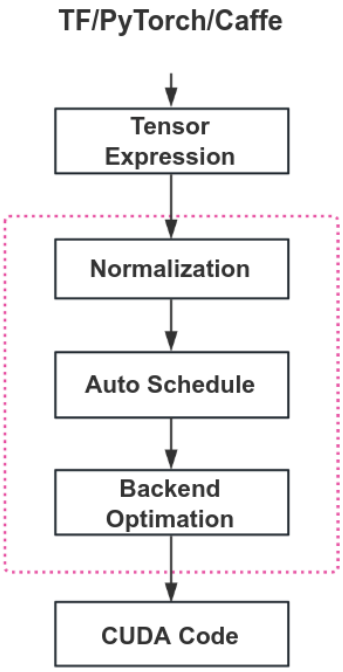

图 1.4　AKG 架构图

Triton[14]：OpenAI 于 2021 年推出的编译器，主要用于加速深度学习应用在 GPU 上执行效率。Triton 是一种 Python DSL，专门用于编写机器学习内核，支持 CPU、GPU 和 ASIC 等多种硬件平台，具备生成针对特定硬件优化内核的能力。在中间优化层级，Triton 编译器通过块级数据流分析技术，自动优化深度学习模型的执行过程。不过，Triton 主要针对英伟达和 AMD 的 GPU 加速器进行优化，对于 RISC-V 存算一体异构芯片支持相对有限。

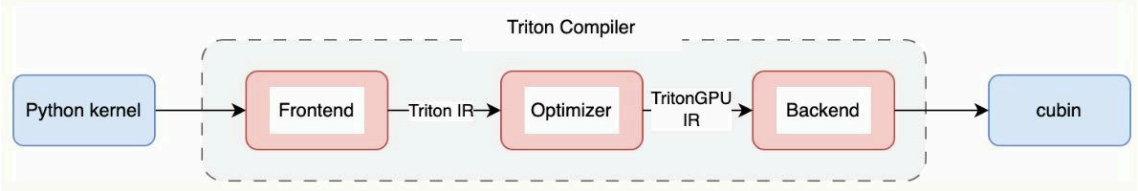

图 1.5　Triton 架构图

IREE[15]：Google 于 2019 年发布的一个开源的通用编译和运行时框架。通过输入高层次的机器学习模型，IREE 为各种硬件生成优化的可执行代码。在中间优化层级，IREE 利用 MLIR 进行多阶段优化，确保模型在目标平台上高效运行。IREE 提

供了高性能的编译器后端，硬件抽象层允许轻松添加对新硬件的支持，但是 IREE 框架主要针对深度学习模型进行端到端的优化，而缺少统一编程模型。

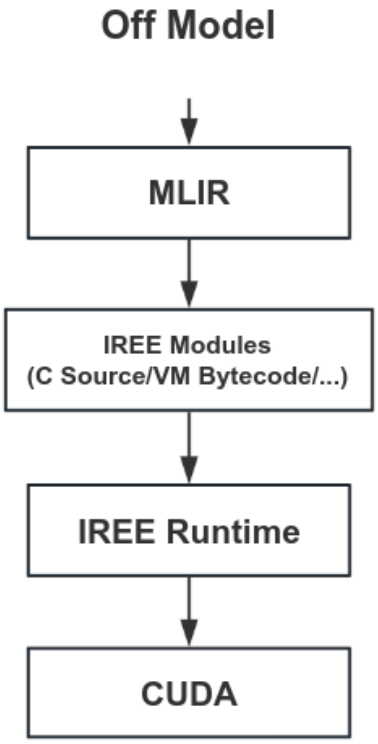

图 1.6　IREE 架构图

当下，众多深度学习编译器虽已面世，但依旧存在诸多局限性。例如，TVM 将算法定义与调度策略分离，而其调度策略需要手动编写或依赖现有模板调用，这对于缺乏编译优化专业知识的深度学习研究人员而言，使用难度较大。MindSpore AKG 能够自动生成优化的计算图，然而其优化重点在于为算子生成高性能 kernel，算子间的并行性尚未得到充分发掘等等。下表对上述深度学习编译器进行了对比总结。

表 1.1　国内外代表性工作总结

工作分类研究	核心思想	关键特征
TVM	将机器学习模型自动编译成可供不同硬件执行的机器语言	算子融合与图优化、量化技术、优化调度、Relay IR、代码生成和后端部署等
Triton	简化 GPU 上执行的复杂操作的开发，提供比 CUDA 更高的生产力	基于分块的编程范式、灵活的 DSL 以及自动性能调优。它允许用户编写高效的内核，同时不必关心底层硬件细节
XLA	将 TensorFlow 图编译成一系列专门为给定模型生成的计算内核，从而利用模型专属信息进行优化	操作融合、内存优化和专用内核生成
IREE	提供模块化和可扩展的编译器流水线，支持从高级中间表示到硬件特定执行的全流程	对不同硬件的兼容性、高效的内存管理以及对实时应用的支持
AKG	通过自动化的方式来探索不同的算法实现和调度策略，找到最优的执行方案	自动调优、多硬件支持和高性能内核生成

1.2.2　深度学习加速器

随着深度学习技术的广泛应用，为其算法定制硬件加速器已成为了学术界与工业界的研究热点，目前，深度学习加速器主要沿着两大方向发展：

一方面，沿用传统的计算架构来提高硬件的加速性能，如 GPU、AISC、FPGA 等。寒武纪在 2014 年到 2016 年间陆续发表了 DIANNAO 系列论文[16]，提出了一系列全定制 AI 加速器的设计方案，通过优化数据流和存储结构，显著提升了深度学习算法的执行速度，为后续研究奠定了重要的理论基础和设计思路；Google 于 2016 年推出以脉动阵列为计算核心加速矩阵运算的 AI 加速器 TPU[17]；同时 Yu-Hsin Chen 等人针对缓存与内存之间大量数据搬移问题设计了一种具有可重配置功能的深度学习加速器 Eyeriss[18]，主要通过行固定（Row Stationary，RS）等方法来降低数据搬运带来的延迟和能耗开销，之后又提出了一种用于紧凑神经网络模型的加速器 Eyeriss v2[19]；清华的 thinker 团队则提出了一种基于 CGRA 的可重构加速器[20]，该加速器可以通过对计算引擎单元阵列进行动态配置，实现以相同的硬件支持包括卷积在内的大多数神经网络运算。

加速器的另外一个发展方向是颠覆传统的冯诺依曼架构。2010 年惠普实验室的 Williams 教授团队用忆阻器实现简单布尔逻辑功能[21]；2016 年，美国加州大学

圣塔芭芭拉分校的谢源教授团队提出使用 RRAM 构建存算一体架构的深度学习神经网络（PRIME），首次验证了基于浮栅晶体管的存内计算在深度学习应用中的效用，相较于传统冯诺伊曼架构的传统方案，PRIME 可以实现功耗降低约 20 倍、速度提升约 50 倍，引起产业界广泛关注[22] 。

1.3 论文的主要研究工作

本文的主要工作是构建一个可以面向 RISC-V SRAM 存算一体芯片进行自动后端优化的编译器系统。编译器对应用程序进行应用特征分析，识别出可以加速的计算部分，并转为特定的 RISC-V 加速指令，充分利用 RISC-V 已有的指令集实现在执行 AI 任务时对各类计算资源的灵活调度，充分发挥 SRAM 存算一体阵列高能效、高算力密度的硬件优势。本文的主要研究工作包括以下几个方面：

1. 对本文基于的 RISC-V 存算一体加速器进行了编译器架构的总体设计，具体设计中包括通过智能识别 NPU 指令以及在 CPU 和 NPU 异构计算单元之间进行指令的调度，实现高效的数据复用和全局数据流分析，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征，提升整体 AI 应用的性能；
2. 设计并实现基于 RISC-V 存算一体芯片的编译器后端，提出对应的计算管理、内存管理以及代码生成方案，同时利用存算一体芯片的硬件特性，降低算子内的数据搬运，减少对 SRAM 频繁写入，提升整体的计算效率；
3. 实现所设计的编译器，通过一个实例对编译器的整体功能进行了验证和分析。

1.4 论文的章节安排

全文共包含 6 个章节，主要研究结构安排如图 7 所示：

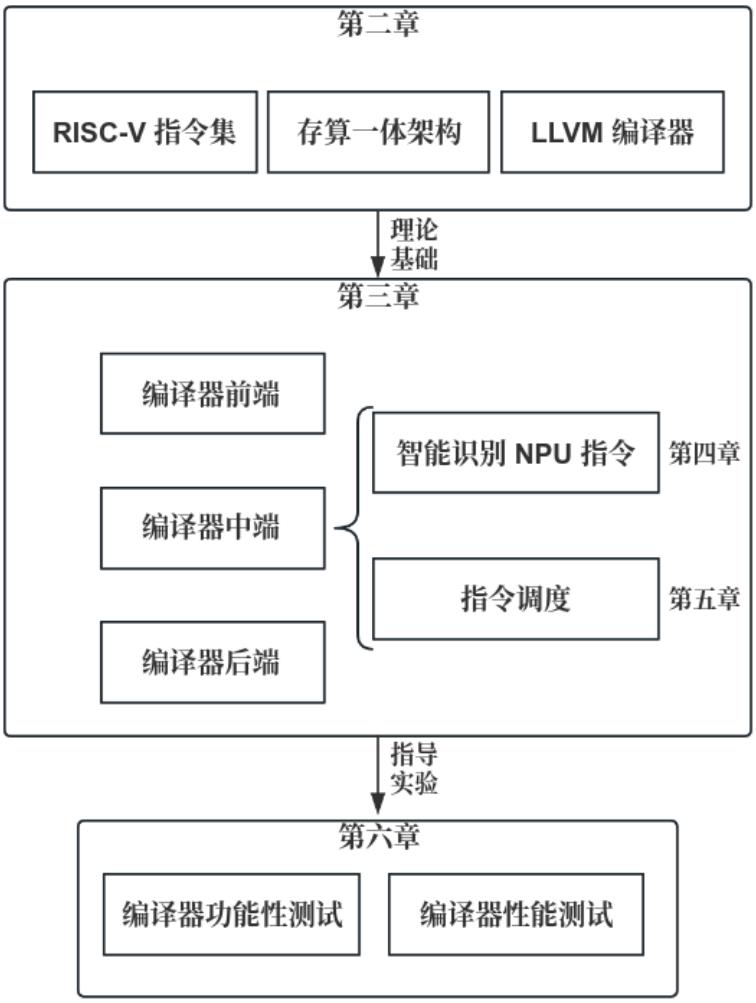

图 1.7　研究内容框架图

第一章为绪论。首先对存算一体加速器的发展背景进行了介绍，接着阐述了深度学习模型在落地部署时面临的困难以及目前专用深度学习编译器所面临的一些挑战；然后又介绍了深度学习编译器、深度学习加速器的研究和发展现状；最后为本文的研究工作和章节安排。

第二章为主要技术基础。介绍了本文研究所涉及的相关基础概念，首先介绍了 RISC-V 基础指令集和扩展指令集；接着对存算一体加速器的总体架构进行了介绍；最后还阐述了 LLVM 编译器的结构和其后端的编译流程。

第三章为基于 RISC-V 存算一体加速器的编译器总体设计。给出了本文编译器的总体结构，对本文编译器从编译器前端、中端以及后端各个模块的设计进行了具体介绍。

充分利用 RISC-V 已有指令集实现在执行 AI 任务运行时对各类计算资源的灵活调度，充分发挥 SRAM 存算一体阵列高能效、高算力密度的硬件优势。

第四章为智能识别 NPU 加速指令。本章着重介绍了编译器如何在 LLVM IR 中间表示上进行应用特征分析，识别出可加速的 LLVM IR 范式。着重分析了三种典型的可加速范式，通过对这几类加速范式的循环结构深入分析，为编译器智能识别出 NPU 加速指令提供关键依据。

第五章为指令调度。本章围绕如何在 CPU 和 NPU 异构计算单元之间进行高效指令调度展开了详细的介绍。

第六章为编译器测试与分析。本章通过实验对所实现编译器的整体功能进行了测试，通过部署神经网络模型对编译器的总体功能以及编译器生成后端代码的正确性进行了验证，同时选取了神经网络中比较常见的 20 种算子在 RISC-V 存算一体模拟器中评估性能表现。

第七章为工作总结与展望。总结了本文的研究工作，并指出了本文设计的编译器的不足之处和今后需要改进完善的方向。

第 2 章　主要技术基础

2.1　RISC-V

RISC-V 是一种基于精简指令集计算（Reduced Instruction Set Computing，RISC）原则的开源指令集架构，2010 年始于加州大学伯克利分校。它的出现意图解决现有的指令集结构（如 X86、ARM、MIPS 等）的不合理设计。相较而言，其开源特性和模块化的架构保证了设计的灵活性和高效性，以满足各种不同应用场景。架构指令集方面，RISC-V 除标准功能设计指令外，包含实现多个不同功能的可选扩展指令。设计人员可以根据实际设计要求选择基础指令集和多个扩展指令集组合，并结合硬件平台组件扩展处理器的功能范围。

RISC-V 共有 5 种基础指令集[23]，指令空间涵盖不同位宽的指令格式，分别是弱内存次序指令集（RVWMO）、32 位整数指令集（RV32I）、32 位嵌入式整数指令集（RV32E）、64 位整数指令集（RV64I）、128 位整数指令集（RV128I）。在基础指令集的基础上 RISC-V 通过对指令集的架构设计的冗余指令进行分类，以提供扩展非标准架构指令的能力，为更专业的硬件提供设计余量。它为处理器设计中的特殊领域结构预留了指令编码空间，用户可以方便地扩展指令子集。如 图 8 所示，RISC-V 体系结构在 32/64 位指令中保留 4 组自定义指令类型，分别是 Custom-0、Custom-1、Custom-2/rv128、Custom-3/rv128。

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (>32*b*)
00	LOAD	LOAD-FP	*Custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*Custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	⩾80*b*

图 2.8　RISC-V 指令集格式

根据 RISC-V 体系结构说明，用户自定义指令空间 Custom-0 和 Custom-1 被保留，不会用做标准扩展指令。而标记为 custom-2/rv128 和 custom-3/rv128 的操作码保留供未来 rv128 使用，标准扩展也会回避使用，以供用户进行指令扩展。

2.2　LLVM 编译器

LLVM[24] 是一个开源的编译器基础设施项目，它以“Low-Level Virtual Machine”的缩写命名，尽管名称中包含了“虚拟机”一词，但 LLVM 不仅仅是一个虚拟机，而是一个综合的编译器工具链。LLVM 提供了一套通用的工具和库，用于开发编译器、优化器、代码生成器等。LLVM 的核心思想是基于中间表示（Intermediate Representation，IR），它定义了一种与机器和语言无关的中间代码表示形式。LLVM IR 是一种低级别的静态单赋值（Static Single Assignment，SSA）形式，它使用基本块和指令的层次结构来表示程序的结构和行为。

2.2.1 LLVM 框架

LLVM 框架主要由前端、中端、后端三大部分组成：

前端（Front End）阶段负责将高级编程语言（如 C、C++、Objective-C、Swift 等）的源代码转换为 LLVM 中间表示（LLVM IR）。这一过程涉及词法分析、语法分析、语义分析等操作，把高级语言的代码解析成编译器能够理解和处理的形式。

中端（Middle End）阶段主要对 LLVM IR 进行优化处理，目的是提高代码的质量和执行效率。优化操作包括但不限于消除无用代码、常量折叠、公共子表达式消除、循环优化等等。中端的优化是与目标硬件平台无关的，它只关注 LLVM IR 本身的优化，不涉及具体的机器指令生成。

后端（Back End）阶段将经过优化的 LLVM IR 转换为目标硬件平台能够执行的机器码。后端需要了解目标硬件的指令集架构、寄存器分配、内存布局等细节，根据这些信息将 LLVM IR 映射为相应的机器指令。同时，后端也会进行一些与硬件相关的优化，如指令调度、寄存器分配优化等，以充分发挥目标硬件的性能。LLVM 后端支持多种不同的硬件平台，包括 x86 架构的处理器、ARM 架构的处理器、PowerPC、MIPS、RISC-V 等，还包括一些新兴的专用硬件加速器。

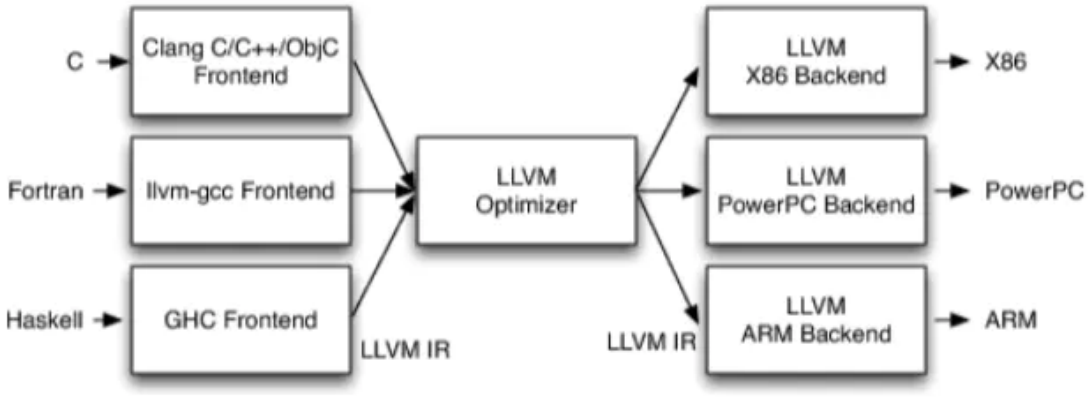

图 2.9 LLVM 编译器的结构

可以看到，若需引入新的编程语言，仅需开发相应的前端，让前端能够生成LLVM IR 结构，就可以利用 LLVM 框架的相关优化。若要使编译器支持新型硬件设备，只需针对该硬件架构实现一个 LLVM 后端，将 LLVM 的中间表示（IR）转换为目标设备的机器码即可。

2.2.2 LLVM IR

LLVM IR 是 LLVM 编译器框架中的一种中间语言，它提供了一个抽象层次，使得编译器能够在多个阶段进行优化和代码生成。LLVM IR 具有类精简指令集、使用三地址指令格式的特征，使其在编译器设计中非常强大和灵活。

LLVM IR 的设计理念类似于精简指令集（RISC），这意味着它倾向于使用简单且数量有限的指令来完成各种操作。其指令集支持简单指令的线性序列，比如加法、减法、比较和条件分支等。这使得编译器可以很容易地对代码进行线性扫描和优化。

2.3 存算一体架构

2.4 本章小结

本章主要描述了论文所涉及的相关技术基础，首先介绍了深度学习相关的基本概念，如

第 3 章　基于 RISC-V 存算一体加速器的编译器设计

本课题旨在解决异构 RISC-V 处理器的“编程墙”难题，以高能效的边缘场景为切入点，设计了一个可以面向基于 RISC-V SRAM 存算一体芯片进行自动后端优化的编译器系统。本章首先介绍了本文编译器的总体结构，然后对编译器前端、中端以及后端各个模块的设计进行具体介绍。关于如何智能识别出 NPU 加速指令以及如何在 CPU 和 NPU 异构计算单元之间进行指令的动态调度将会在接下来几章具体讨论。

3.1　编译器总体架构

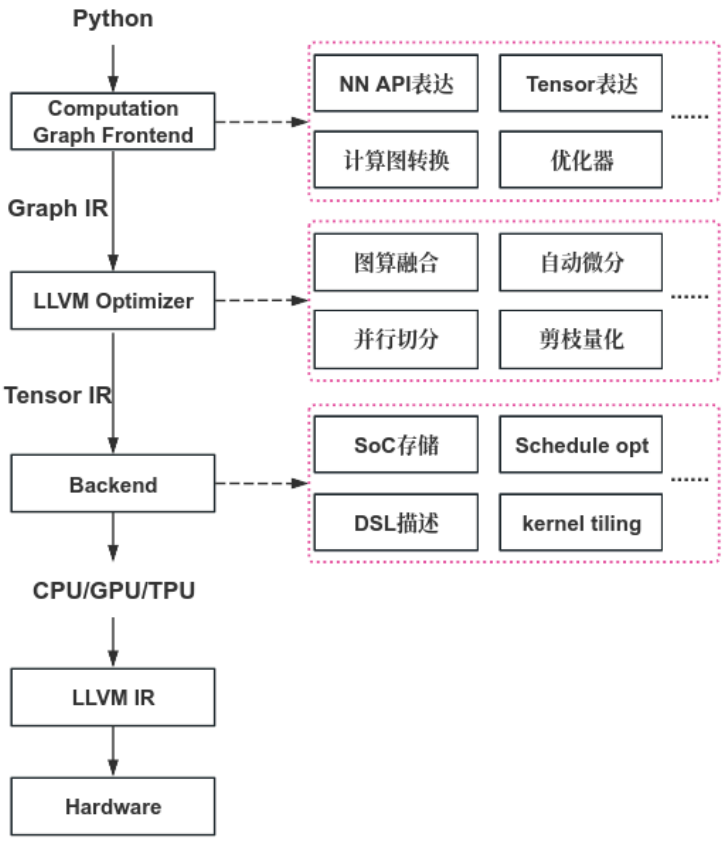

图 3.10　深度学习编译器架构

深度学习编译器的工作是将上层深度学习模型通过各种优化技术生成硬件平台执行所需要的指令，确保神经网络模型在硬件上高效执行，其处于深度学习框架和底层硬件之间，它提供了一种中间层，可以覆盖不同的加速器硬件。其大致的框架图如图 10 所示。

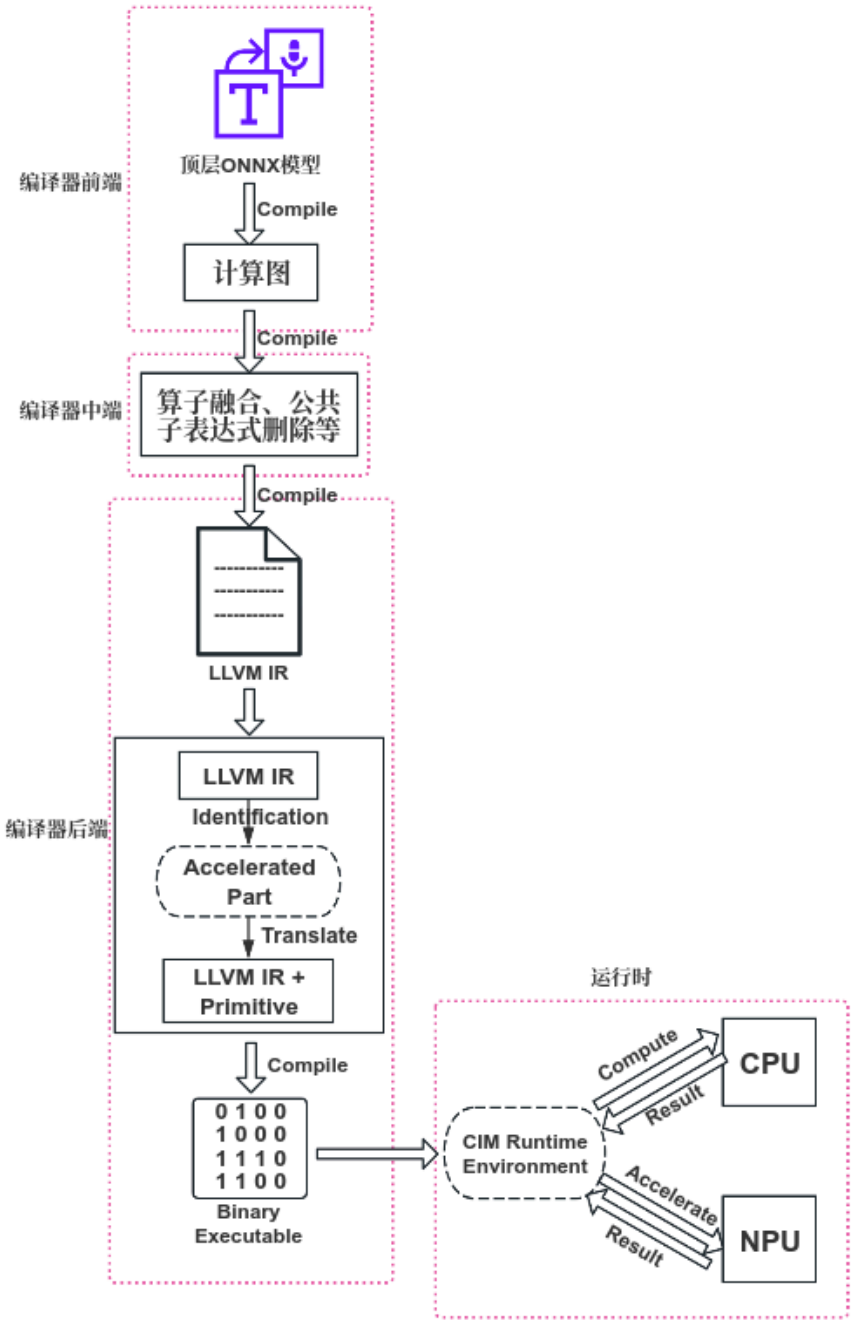

图 3.11　编译器整体架构

本文所设计的基于 RISC-V 存算一体模拟器的深度学习编译器的结构，具体如图 11 所示。鉴于 LLVM 优秀的模块化与可扩展性设计，以及其具有足够表现力的中间表示 —— LLVM IR，本文编译器采用 LLVM IR 作为中间表示形式，并将 LLVM 作为后端，来复用其强大的优化器与代码生成器，同时通过扩展 RISC-V 加速指令来实现对 RISC-V 存算一体模拟器的支持。下面基于图 11 对本文编译器各层次的主要功能及设计思想进行简要概述。

（1）编译器前端。前端的主要作用为负责接收和处理来自不同 AI 框架的模型，并将其解析为计算图，进行初步优化。目前包括 PyTorch[25] 和 TensorFlow[26] 在内的大多数主流深度学习框架都支持将模型导出或转为 ONNX 格式，故本文中选定 ONNX 作为本文编译器的输入格式，且 ONNX 的设计定位就是作为不同框架间或框架与工具间模型转换的中间格式，所以使用 ONNX 作为输入便能够直接或间接的支持多种框架模型。

（2）编译器中间优化器。中间优化器的主要作用为接收编译器前端构建的计算图，对其进行体系结构无关的变换与优化。本文编译器提供的计算图优化策略包括算子融合、死代码删除以及公共子表达式删除等等。通过对计算图进行等价变换，可以降低计算图的计算复杂度和空间复杂度，减少模型推理的运算时间，提升整体的计算效率。最后本文编译器将优化之后的计算图转换为 LLVM IR 输入到后端进行 NPU 加速指令的识别、指令的动态调度以及目标硬件代码生成。

（3）编译器后端。后端是编译器体系中与目标体系结构关联最为密切的部分，主要功能为做与硬件结构相关的优化以及生成针对目标体系结构的优化代码。本文以本课题组已有的 RISC-V 存算一体模拟器为目标体系结构，设计并实现了一套编译器后端架构。该后端以 LLVM IR 为输入，执行一系列与体系结构相关的变换及优化操作，包括通过对 LLVM IR 进行应用特征分析，识别出可加速的 LLVM IR 范式，以自动将它们卸载到 CIM 加速器的 NPU 核心进行加速计算以及运行时在 CPU 和 NPU 异构计算单元之间进行指令的动态调度等等，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征。

3.2 编译器前端

编译器前端（Compiler Frontend）主要负责接收和处理来自不同 AI 框架的模型，并将其转换为通用的中间表示（IR），进行初步优化。本文编译器选定 ONNX 模型文件格式作为本文编译器的输入格式，通过解析 ONNX 模型文件来构建计算图进行图级别优化。目前包括 PyTorch[25] 和 TensorFlow[26] 在内的大多数主流深度学习框架都对 ONNX 有着不同程度的支持，这也便于算法模型在不同框架之间的迁移以及对编译器前端接口的设计。

图 3.12　编译器前端工作流程图

3.2.1 ONNX

ONNX[27]（Open Neural Network Exchange），开放神经网络交换，是一套表示深度神经网模型的开放格式，由微软和 Facebook 于 2017 年推出。ONNX 定义了一组与环境和平台均无关的标准格式，用于在各种深度学习训练和推理框架转换的一个中间表示。它定义了一种可扩展的计算图模式、运算符和标准数据类型，为不同框架提供了通用的 IR。如图 13 使用 Netron[28] 对 yolov3-tiny[29] 模型进行可视化。

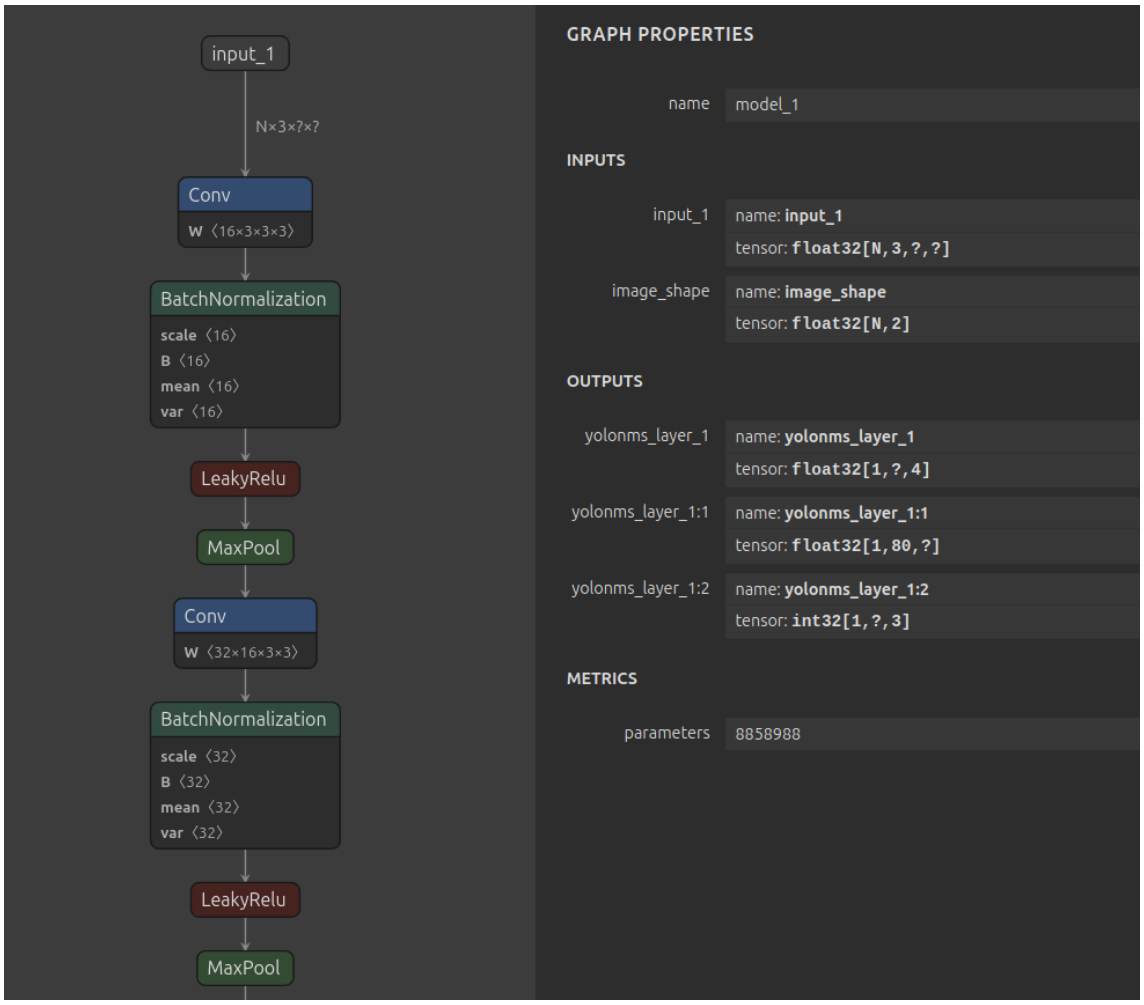

图 3.13 yolov3-tiny ONNX 可视化

ONNX 使用 Protobuf 序列化数据结构来存储神经网络的权重信息。Protobuf 是一种轻便高效的结构化数据存储格式，可以用于数据的结构化和序列化，适合做数据存储或数据交换格式（与语言无关、平台无关）。下表展示了 Add 算子的简化版 ONNX 模型实例。

```
ir_version: 10
graph {
```

```
  node {
    input: "input1"
    input: "input2"
    output: "output"
    name: "add_node"
    op_type: "Add"
  }
......
```

表 3.2　ONNX 结构

类型	用途
ModelProto	定义了整个网络的模型结构
GraphProto	定义了模型的计算逻辑，包含了构成图的节点，这些节点组成了一个有向图结构
NodeProto	定义了每个 OP 具体操作
ValueInfoProto	序列化的张量，用来保存 weight 和 bias
TensorProto	定义了输入输出形状信息和张量的唯独信息
AttributeProto	定义了 OP 中的具体参数，比如 Conv 中的 stride 和 kernel_size 等

ONNX 结构如表 2 所示，当我们将 ONNX 模型加载进来之后，得到的是一个 `ModelProto`，它包含了一些版本信息，生产者信息和一个非常重要的 `GraphProto`；在 `GraphProto` 中包含了四个关键的 repeated 数组，分别是 node（`NodeProto` 类型），input（`ValueInfoProto` 类型），output（`ValueInfoProto` 类型）和 initializer（`TensorProto` 类型），其中 node 中存放着模型中的所有计算节点，input 中存放着模型所有的输入节点，output 存放着模型所有的输出节点，initializer 存放着模型所有的权重；每个计算节点都同样会有 input 和 output 这样的两个数组，通过 input 和 output 的指向关系，我们就能够利用上述信息快速构建出一个深度学习模型的拓扑图。最后每个计算节点当中还包含了一个 `AttributeProto` 数组，用于描述该节点的属性，例如 Conv 层的属性包含 `group`，`pads` 和 `strides` 等等。

3.2.2　构建计算图

编译器首先需要对读取的模型文件进行反序列化才可以得到对应模型在内存中的表示形式。使用 `onnx.load()` 函数加载 ONNX 模型文件，得到一个 `ModelProto` 对象，从 `ModelProto` 对象中获取 `GraphProto` 对象，然后可以遍历图中的节点（`NodeProto`）、输入（`ValueInfoProto`）、输出（`ValueInfoProto`）等。对于每个节点，可以获取其名称、类型、输入和输出名称、属性等信息，属性包含操作

的具体参数。同时也可以获取模型中的权重和偏置张量，这些张量存储在图的初始器（Initializer）中。

根据解析到的节点和张量信息，构建一个计算图。根据节点的输入和输出关系，在计算图中添加相应的节点和边。边表示数据的流动方向，通常从一个节点的输出指向另一个节点的输入。最后将解析到的张量与计算图中的节点联系起来。

3.3 编译器中间优化器

编译器中间优化器是对前端构建的计算图进行与硬件结构无关的优化，在不影响计算结果的情况下降低模型的空间复杂度以及计算复杂度从而减少模型在加速器上的推理时间。中间层图优化策略中根据粒度主要分为层级优化、张量级优化和元素级优化。本文编译器接收前端构建的计算图，从层级优化出发，使用算子融合、公共子表达式删除、死代码删除等优化来对构建的计算图进行处理，来提高神经网络模型执行效率。

3.3.1 算子融合

在编译器前端解析 ONNX 模型后可以得到对应的计算图，计算图是对算子执行过程形象的表示，假设$C = \{N, E, I, O\}$为一个计算的计算表达，计算图是一个有向连通无环图，由节点N（Node）、边集E（Edge）、输入边I（Input）以及输出边E（Output）组成的四元组，这样的抽象使我们能够更加专心于逻辑上的处理而不用在意具体的细节。而算子融合主要通过对计算图上存在数据依赖的“生产者-消费者”算子进行融合，从而提升中间 Tensor 数据的访存局部性，以此来解决内存墙问题。

算子的融合方式非常多，不同的算子融合有着不同的算子开销，也有着不同的内存访问效率的提升。现举出几个例子进行说明。

假设我们有如图 14 左侧所示的计算图，其有 4 个算子 A，B，C，D，此时我们将 C 和 D 做算子融合（可行的话），此时可以减少一次的 kernel 开销，也减小了一次的中间数据缓存。

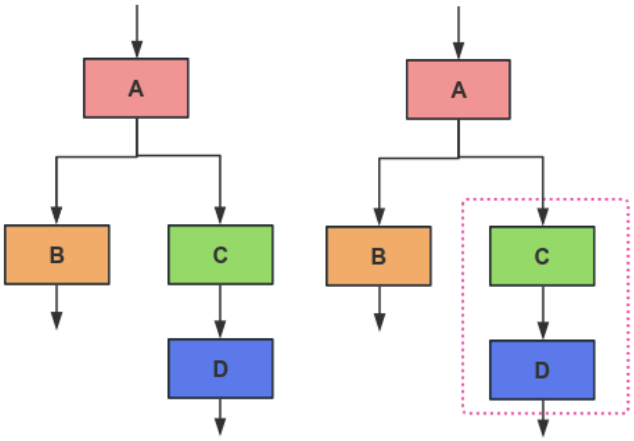

图 3.14　算子融合案例 1

如果是图 15 左侧所示的计算图，B 和 C 算子是并行执行的，但此时有两次访存，可以将 A“复制”一份分别与 B，C 做融合，如图 15 右侧所示，此时我们 A，B 与 A，C 可以并发执行且只需要一次访存。

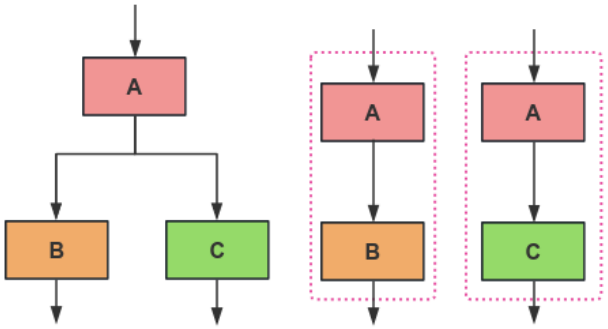

图 3.15　算子融合案例 2

依然还是图 15 左侧所示的计算图，此时我们可以变换一下融合的方向，即横向融合，将 B 和 C 融合后减少了一次 Kernel 调度，同时结果放在内存中，缓存效率更高。

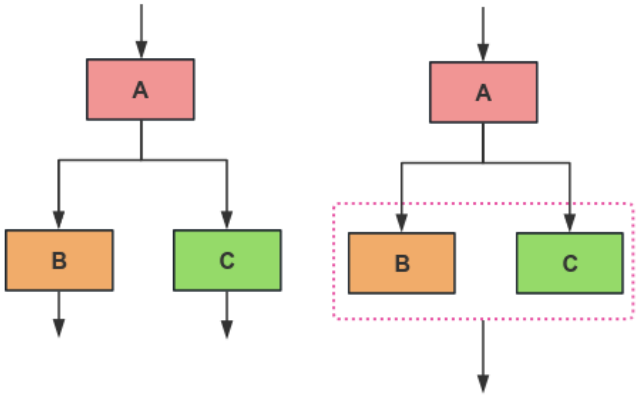

图 3.16　算子融合案例 3

还是图 15 左侧所示的计算图，我们也可以将 A 和 B 进行融合，此时运算结果放在内存中，再给 C 进行运算，此时可以提高内存运算效率。

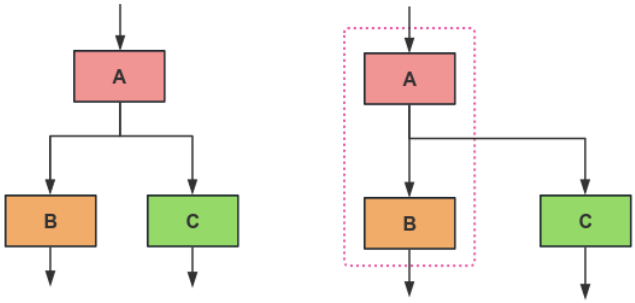

图 3.17　算子融合案例 4

3.3.2　公共子表达式消除

公共子表达式消除（Common Subexpression Elimination，CSE）也称为冗余表达式消除，是普遍应用于各种编译器的经典优化技术。旨在消除程序中重复计算的公共表达式，从而减少计算量和提高执行效率。

在程序中，有时会出现多个地方使用相同的表达式进行计算，并且这些表达式的计算结果相同。重复计算这些表达式，会增加不必要的计算开销。公共表达式消除的目标就是识别出这些重复的计算，并将其提取出来，只计算一次，然后将结果保存起来供后续使用。

由于 AI 编译器中子表达式是基于计算图或图层 IR，通过在计算图中搜索相同结构的子图，简化计算图的结构，从而减少计算开销，如图 18 中 Op3 和 Op4 都经过了相同的图结构 $\{\{\mathrm{op}_1, \mathrm{op}_2\}, \mathrm{op}_1 \rightarrow \mathrm{op}_2\}$，编译器会将相同子图的所有不同输出都连接到同一个子图，然后会在后续的死代码消除中删除其他相同的子图，从而达到简化计算图的目的，减少计算开销。

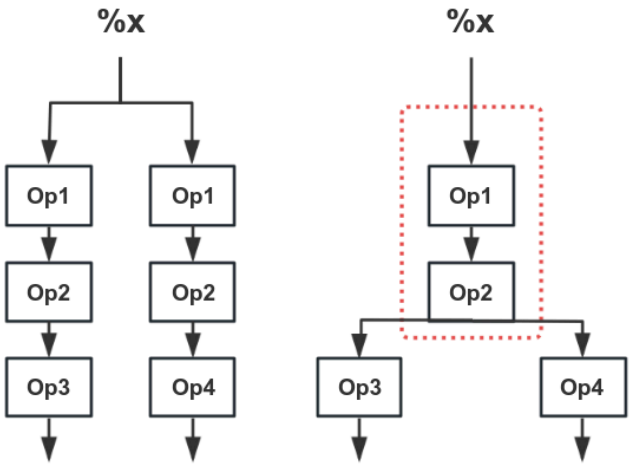

图 3.18　公共子表达式消除案例

3.3.3　死代码删除

死代码消除（Dead Code Elimination，DCE）旨在删除程序中不会被执行的代码，从而提高程序的执行效率和资源利用率。死代码是指在程序的当前执行路径下不会被访问或执行的代码片段。AI 编译器通常是通过分析计算图，找到无用的计算节点或不可达的计算节点，然后消除这些节点。

在计算图中，不可达节点是指从输入节点通过图中的有向边无法到达的节点。如图 19 所示，计算图中有 A，B，C 三个算子，假设三个算子都不是输入节点。不存在一条路径从输入节点到 B 节点，所以 B 节点是不可达节点，AI 编译器会删除该节点，并删除其到可达节点的边，即边 $B \rightarrow C$。

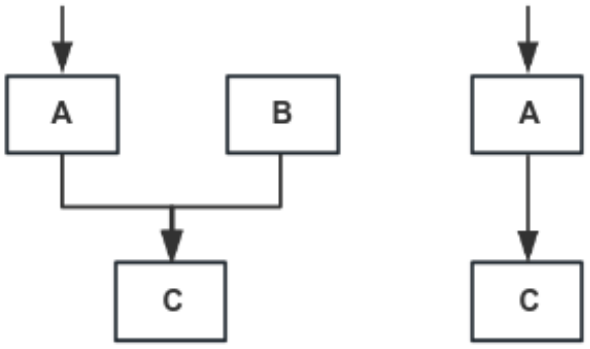

图 3.19　死代码消除案例 1

在计算图中，无用节点是指某个计算节点的结果或副作用不会对输出节点产生影响。如图 20 所示，计算图中有 A，B，C 三个算子，B 节点输出没有后续节

点，不会对后续的计算图产生影响，所以 B 节点是无用节点，AI 编译器会将该节点删除。

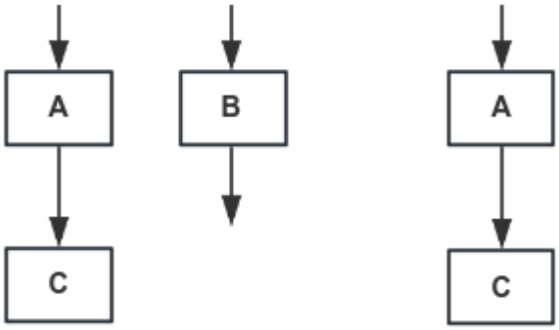

图 3.20　死代码消除案例 2

3.4　编译器后端

编译器后端（Compiler Backend）负责将优化后的 IR 转换为特定硬件平台的低层次表示，并进行硬件特定优化和代码生成。由于本文基于的 RISC-V 存算一体加速器，编译器后端的主要工作是识别出优化后的 LLVM IR 中的可加速范式，以自动将它们卸载到 CIM 加速器的 NPU 核心进行加速计算，最终输出在 CIM 加速器执行所需要的指令序列。下面详细介绍本文设计的编译器中基于加速器硬件结构在后端所做的工作，关于如何智能识别 NPU 指令以及指令动态调度会在本文第 4 章和第 5 章详细介绍。

3.4.1　内存分配管理

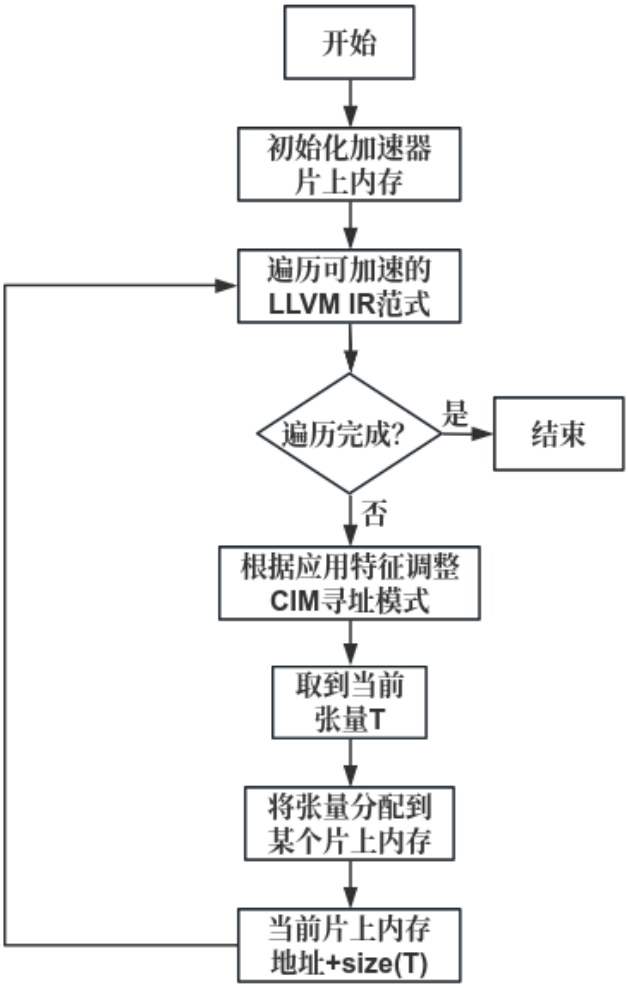

图 3.21　内存分配流程图

在智能识别出可加速的 LLVM IR 指令后，编译器需要为对应的输入张量在片上内存分配内存空间以便于加速器对计算数据的读写和操作，同时还要进行内存管理防止各张量之间地址冲突从而产生的数据覆盖。内存分配的流程如图 21 所示，在识别出优化后的 LLVM IR 中的可加速范式之后，首先初始化 NPU 核心上的片上内存，然后遍历识别出的 LLVM IR 中的可加速范式，根据输入张量的不同形状，切换 CIM 加速器的寻址方式来选择最佳的加速阵列来加速对应的计算范式，同时选择一块片上内存来为其输入张量分配实际内存大小，每次分配时只需要把当前片上内存空间的地址指针作为该输入张量的首地址，同时利用底层 RISC-V 扩展指令将数据从片外内存传输到当前片上内存，然后根据张量大小对该片上内存地址指针叠加并更新，而分配的内存空间大小由张量自身的尺寸来维护，最后重复此操作以完成对可加速 LLVM IR 范式中的张量的内存分配工作。当该计算完成时，释放对应的片上内存，同时将对应的输出结果写回到片外内存，避免造成内存资源浪费。

3.4.2 计算逻辑管理

计算逻辑管理就是基于 RISC-V 存算一体加速器的指令集架构将计算操作转为特定的 RISC-V 加速指令以及 RISC-V 通用指令，充分利用 RISC-V 通用核心、NPU 加速核心。RISC-V 存算一体加速器可以加速的基本计算范式为矩阵相关操作、向量相关操作等等，本文编译器后端根据这些范式识别 LLVM 中间表示（IR）中是否存在与之匹配的代码片段，并进一步检查对应的计算任务的规模是否可以被加速，实现自动将原 IR 代码转换成可以在这种异构架构下执行的目标 IR 代码。后续会在第 4 章中详细介绍如何智能识别 LLVM IR 中的可加速范式。

在本小节中以最简单的向量向量操作为例子来说明如何智能识别 NPU 加速指令。

3.5 本章小结

本章节介绍了面向 RISC-V 存算一体加速器的编译器总体架构。在 3.1 节中提出了本文编译器的总体架构，并对编译器的组成以及各个模块的功能进行了介绍。从编译器前端、中间优化器到编译器后端进行了系统的阐述。后续会着重介绍如何智能识别 NPU 加速指令以及指令动态调度。

第 4 章　智能识别 NPU 指令

上一章主要对本文编译器的整体架构，从编译器前端、中端到后端进行了系统的阐述，本章着重介绍编译器如何在 LLVM IR 中间表示上进行应用特征分析，识别出可加速的 LLVM IR 范式，以自动将它们卸载到 CIM 加速器的 NPU 核心进行加速计算，转为特定的 RISC-V 加速指令，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征。

4.1　LLVM IR 与可加速范式

当前基于 RISC-V 存算一体模拟器仅通过支持 RISC-V 向量、矩阵扩展指令集来进行加速计算，为了利用 RISC-V 存算一体加速器中的 NPU 核心来加速计算，编译器应该识别出 LLVM IR 中可以通过 NPU 核心加速的典型计算模式。尽管 CIM 加速器的应用场景越来越多，但 CIM 加速器可以加速的基本操作却非常有限。根据其底层硬件设计，我们识别出三种典型的可加速计算范式，即向量 - 向量操作（Vector-Vector Multiplications，VVM）、矩阵 - 向量操作（Matrix-Vector Multiplications，MVM）、矩阵 - 矩阵操作（Matrix-Matrix Multiplications，MMM），这几种计算范式都是通过比较复杂的循环策略来实现的。因此，为了实现这些可加速范式，本章对 LLVM IR 的循环结构进行深入分析，旨在剖析不同的加速范式所对应的循环结构的特征，从而为编译器智能识别出 NPU 加速指令提供关键依据，进而充分发挥 RISC-V 存算一体加速器中 NPU 核心的加速效能。

以矩阵 - 向量操作（MVM）为例，MVM 通常是由两层嵌套的循环来实现的，并且内循环体中的迭代语句与向量和矩阵相关。因此，为了识别出 MVM 计算范式，我们必须首先识别出两层嵌套循环的循环结构，然后识别出内层循环中两个向量之间的操作。

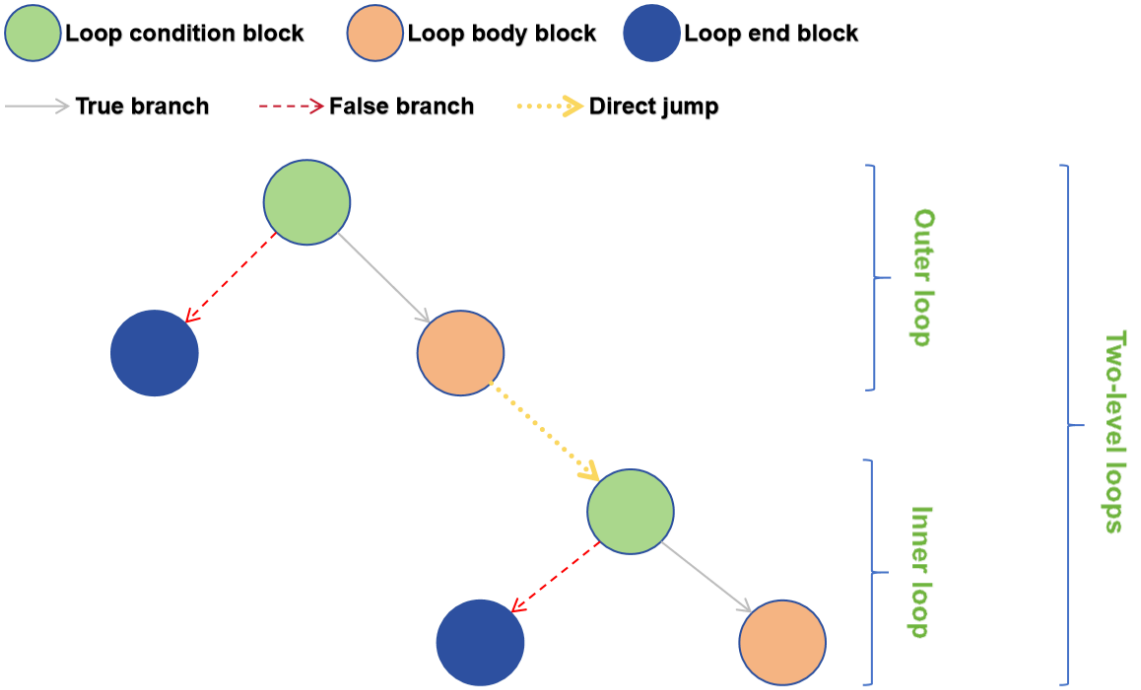

图 4.22　MVM 循环结构分析

而 LLVM IR 是 LLVM 编译器框架中的一种中间语言，它提供了一个抽象层次。LLVM IR 具有类精简指令集、使用三地址指令格式的特征，使其在编译器设计中非常强大和灵活，因此，可以利用它来识别可以通过 CIM 加速器进行加速的计算模式。它们的结构和关键信息都可以在 LLVM IR 的不同块中来找到。一个 IR 文件通常包含许多块，每个块由多条语句组成。图 22 展示了 LLVM IR 中两层嵌套循环的逻辑块，不同颜色的圆圈代表不同类型的块，其中绿色、橙色和蓝色的圆圈分别代表了循环条件、循环体以及循环结束的逻辑块，这些块通过跳转语句连接起来，跳转语句包括无条件跳转和条件跳转。因此，我们在 LLVM IR 中分析这些块，以识别出可加速的计算范式，并将其转为特定的 RISC-V 加速指令，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征。

通常，表示循环条件、循环体以及循环结束的逻辑块很容易在 LLVM IR 中识别，因为它在编译器中维护了一些有用的信息。例如，我们可以直接从名称推断出块的类型（`for.cond, for.boby, for.end`）。

4.2　向量 - 向量操作（Vector-Vector Multiplications，VVM）

在深度学习算法中，向量 - 向量操作（Vector-Vector Multiplication，VVM）虽不如矩阵 - 矩阵操作、矩阵 - 向量操作那般频繁出现，但也颇具常见性。例如点积（内积），它将两个向量映射为一个标量，常用于计算向量之间的相似度，如在注意

力机制（Attention Mechanism）里，通过查询向量（Query）和键向量（Key）的点积来衡量它们的相关性，进而确定值向量（Value）的权重，对信息进行筛选整合，这对像机器翻译、文本生成这类序列建模任务的性能发挥着关键作用，影响着结果的准确度和合理性。

对于一个大小为 $1 \times m$ 的向量 Q 和一个大小为 $m \times 1$ 的向量 K, 它们的乘积是一个大小为标量 $value$。数学表达式为 $value = Q \cdot K$，具体计算逻辑代码如下所示。

```
void vec_vec_processing(int *Q, int *K, int c, int m){
  for (int i = 1; i <= m; i++) {
    c += (a[i] * b[i]);
  }
}
```

接下来，我们分析由上述代码生成的 LLVM IR，Pattern 1 显示了 LLVM IR 中的 VVM 可加速范式。其中，`loop.condA` 是循环的循环条件块，`loop.bodyB` 是循环的循环体，字母 `A - B` 表示各模块的标识，`loop` 可以表示由 `for` 或 `while` 语句产生的循环结构。`loop.bodyB` 主要用于实现应用程序的计算逻辑。此外，一些关键指令和指令之间的依赖关系在 `loop.bodyB` 中受到约束。

Pattern 1: A typical VVM pattern in LLVM IR

```
1  loop.condA: ; preds = %28, %4
2      %11 = load i32, ptr %9, align 4
3      %12 = load i32, ptr %8, align 4
4      %13 = icmp sle i32 %11, %12
5      br i1 %13, label %loop.bodyB, label %loop.endX
6  loop.bodyB: ; preds = %loop.condA
7      %15 = load ptr, ptr %5, align 8
8      %16 = load i32, ptr %9, align 4
9      %17 = sext i32 %16 to i64
10     %arrayidx1 = getelementptr inbounds i32, ptr %15, i64 %17
11     %arrayvalue1 = load i32, ptr %arrayidx1, align 4
12     %20 = load ptr, ptr %6, align 8
13     %21 = load i32, ptr %9, align 4
14     %22 = sext i32 %21 to i64
15     %arrayidx2 = getelementptr inbounds i32, ptr %20, i64 %22
16     %arrayvalue2 = load i32, ptr %arrayidx2, align 4
17     %value = mul nsw i32 %arrayvalue1, %arrayvalue2
18     %26 = load i32, ptr %7, align 4
19     %ans = add nsw i32 %26, %value
20     store i32 %ans, ptr %7, align 4
21     br label %28
22 28:
23     %29 = load i32, ptr %9, align 4
24     %30 = add nsw i32 %29, 1
25     store i32 %30, ptr %9, align 4
26     br label %loop.condA, !llvm.loop !6
```

4.3 矩阵 - 向量操作（Matrix-Vector Multiplications，MVM）

矩阵 - 向量操作（Matrix-Vector Multiplications，MVM）是一种典型的计算模式，可以通过 CIM 加速器进行加速。对于大多数神经网络应用和图像处理应用，其中包含大量的 MVM 操作，并且它们往往会对性能和能耗产生很大的影响。

对于一个大小为 $m \times n$ 的矩阵 A 和一个大小为 $n \times 1$ 的向量 x, 它们的乘积是一个大小为 $m \times 1$ 的向量 y。数学表达式为 $y = Ax$，具体计算逻辑代码如下所示。

```
void mat_vec_processing(int **a, int *b, int *c, int m, int n){
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      c[j] += (a[i][j] * b[j]);
    }
  }
}
```

接下来，我们分析由上述代码生成的 LLVM IR，Pattern 2 显示了 LLVM IR 中的 MVM 可加速范式，我们可以在图 22 中相对应的循环逻辑结构。其中，`loop.condA` 和 `loop.condC` 分别是外层循环和内层循环的循环条件块，`loop.bodyB` 和 `loop.bodyD` 分别是外层循环和内层循环的循环体，字母 `A - D` 表示各模块的标识，`loop` 可以表示由 `for` 或 `while` 语句产生的循环结构。`loop.bodyB` 主要用于初始化内层循环的相关循环变量，`loop.bodyD` 主要用于实现应用程序的计算逻辑。此外，一些关键指令和指令之间的依赖关系在 `loop.bodyD` 中受到约束。

Pattern 2: A typical MVM pattern in LLVM IR

```
loop.condA:
    %<rang_i> = load i32, ptr %<i>, align 4
    %cmp1 = icmp sle i32 %<rang_i>, <m>
    br i1 %cmp1, label %loop.bodyB, label %loop.endX
loop.bodyB:
    br label %loop.condC
loop.condC:
    %<range_j> = load i32, ptr %<j>, align 4
    %cmp2 = icmp sle i32 %<range_j>, <n>
    br i1 %cmp2, label %loop.bodyD, label %loop.endY
loop.bodyD:
    %27 = load ptr, ptr %26, align 8
    %28 = load i32, ptr %12, align 4
    %29 = sext i32 %28 to i64
    %arrayidx1 = getelementptr inbounds i32, ptr %27, i64 %29
    %arrayvalue1 = load i32, ptr %arrayidx1, align 4
    %32 = load ptr, ptr %7, align 8
    %33 = load i32, ptr %12, align 4
    %34 = sext i32 %33 to i64
    %arrayidx2 = getelementptr inbounds i32, ptr %32, i64 %34
    %arrayvalue2 = load i32, ptr %arrayidx2, align 4
    %value = mul nsw i32 %arrayvalue1, %arrayvalue2
    %38 = load ptr, ptr %8, align 8
    %39 = load i32, ptr %11, align 4
    %40 = sext i32 %39 to i64
    %arrayidx3 = getelementptr inbounds i32, ptr %38, i64 %40
    %arrayvalue3 = load i32, ptr %arrayidx3, align 4
    %ans = add nsw i32 %arrayvalue3, %value
    store i32 %ans, ptr %arrayidx3, align 4
    br label %block
block:
    %45 = load i32, ptr %12, align 4
    %46 = add nsw i32 %45, 1
    store i32 %46, ptr %12, align 4
    br label %loop.condC, !llvm.loop !6
```

接下来分析上述 Pattern 1 中的 LLVM IR，`arrayidx#` 表示 MVM 操作中矩阵或向量对应位置的数据的地址，`<i>`，`<j>` 表示循环条件变量，`%range_#` 表示对应的值。`icmp` 指令用来比较两个变量，并将比较后的结果存储在布尔变量 `%cmp#` 中，这条关键指令用于判断循环终止的条件。其中 `i32` 表示整数类型。上述 Pattern 中首先将向量乘法的结果赋值给输出变量 `%value`，然后进行累加将结果赋值给 `%ans`。因此，应该循环内部约束 `mul` 和 `add` 之间的依赖关系。总的来说，为了识别 MVM 加速模式，我们必须首先识别两层的嵌套循环结构，然后识别内部循环中的乘法和加法指令。

4.4 矩阵 - 矩阵操作（Matrix-Matrix Multiplications，MMM）

在传统得到计算机体系结构中，矩阵 - 矩阵操作（Matrix-Matrix Multiplications，MMM）是一个在时间和能耗方面花费都很昂贵的计算任务，但它是神经网络等应用中的一个常见操作。其识别过程与 MVM 类似，只需要识别 IR 中的三层嵌套循环结构和最内层的循环中有关两个矩阵间的相关操作。

对于一个大小为 $m \times n$ 的矩阵 A 和一个大小为 $n \times p$ 的矩阵 B, 它们的乘积是一个大小为 $m \times p$ 的矩阵 C。数学表达式为 $C = AB$，具体计算逻辑代码如下所示。

```
void mat_mat_processing(int **a, int **b, int **c, int m, int n, int p){
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= p; j++) {
      for (int k = 1; k <= n; k++) {
        c[i][j] += (a[i][k] * b[k][j]);
      }
    }
  }
}
```

Pattern 3: A typical MMM pattern in LLVM IR

```
loop.condA:……
loop.bodyB:……
loop.condC:……
loop.bodyD:……
loop.condE:……
loop.bodyF:
    %35 = load ptr, ptr %34, align 8
    %36 = load i32, ptr %15, align 4
    %37 = sext i32 %36 to i64
    %arrayidx1 = getelementptr inbounds i32, ptr %35, i64 %37
    %arrayvalue1 = load i32, ptr %arrayidx1, align 4
    %44 = load ptr, ptr %43, align 8
    %45 = load i32, ptr %14, align 4
    %46 = sext i32 %45 to i64
    %arrayidx2 = getelementptr inbounds i32, ptr %44, i64 %46
    %arrayvalue2 = load i32, ptr %arrayidx2, align 4
    %value = mul nsw i32 %arrayvalue1, %arrayvalue2
    %54 = load ptr, ptr %53, align 8
    %55 = load i32, ptr %14, align 4
    %56 = sext i32 %55 to i64
    %arrayidx3 = getelementptr inbounds i32, ptr %54, i64 %56
    %arrayvalue3 = load i32, ptr %arrayidx3, align 4
    %ans = add nsw i32 %arrayvalue3, %value
    store i32 %ans, ptr %arrayidx3, align 4
    br label %60
60:
    %61 = load i32, ptr %15, align 4
    %62 = add nsw i32 %61, 1
    store i32 %62, ptr %15, align 4
    br label %loop.condE, !llvm.loop !6
loop.endZ:……
loop.endY:……
```

接下来分析上述 Pattern 3 中的 LLVM IR，MMM 的逻辑结构类似于 MVM，其循环体结构如图 23 所示。MMM 的操作应该在三层嵌套循环中执行，其中关键指令在最内层的循环体中执行。同理，`arrayidx#` 表示 MVM 操作中矩阵或向量对应

位置的数据的地址，<i>，<j>，<k>表示循环条件变量，%range_#表示对应的值。icmp 指令用来比较两个变量，并将比较后的结果存储在布尔变量%cmp#中，这条关键指令用于判断循环终止的条件。其中 i32 表示整数类型。上述 Pattern 中首先将向量乘法的结果赋值给输出变量%value，然后进行累加将结果赋值给%ans。因此，应该循环内部约束 mul 和 add 之间的依赖关系。总的来说，为了识别 MVM 加速模式，我们必须首先识别三层的嵌套循环结构，然后识别最内部循环中的乘法和加法指令。

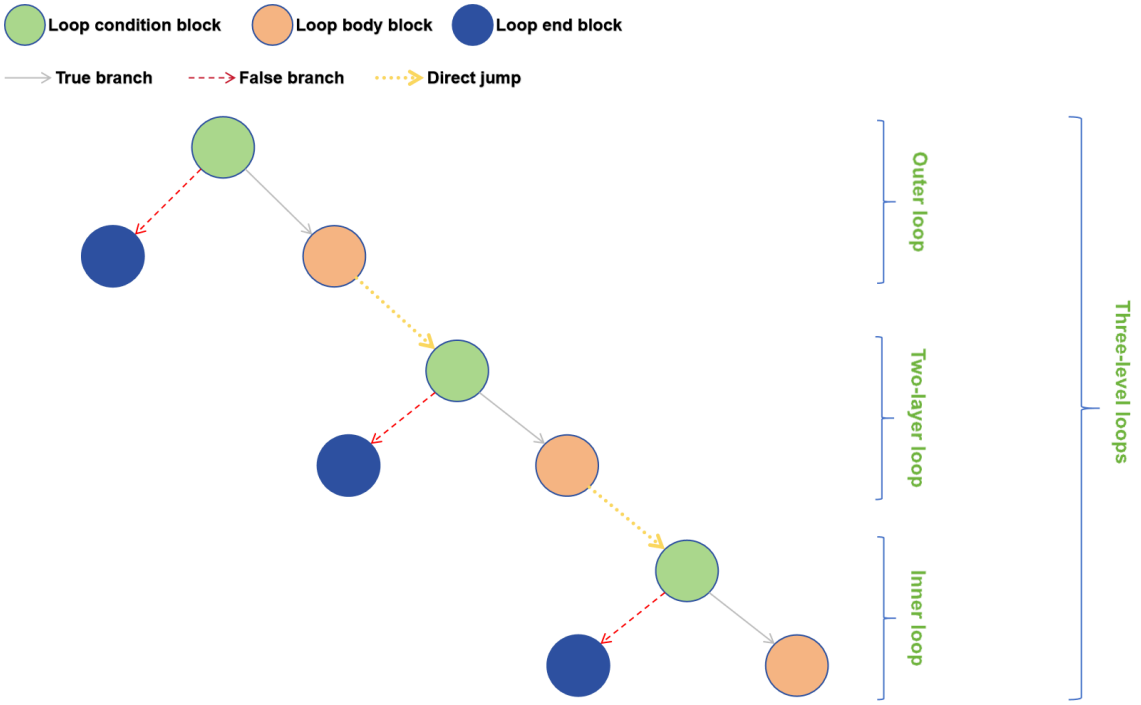

图 4.23　MMM 循环结构分析

4.5　本章小结

本章主要描述了如何在 LLVM IR 中间表示上智能识别出可加速范式，并对几种典型的可加速计算范式分析了其所对应的循环结构的特征，进而编译器可以依据此来智能识别出 NPU 加速指令，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征。

第 5 章　指令调度

本章着重对如何在 CPU 和 NPU 异构计算单元之间进行指令的静态和动态调度进行了阐述，当编译器后端智能识别出 NPU 加速指令后，如何在异构架构下调度生成的 RISC-V 指令来充分发挥 SRAM 存算一体阵列高能效、高算力密度的硬件优势，是一个重要的问题。本章对此进行了系统的阐述。

5.1　指令调度简介

指令调度是指对程序块或过程中的操作进行排序以有效利用处理器资源的任务。指令调度的目的就是通过重排指令，提高指令级并行性，使得程序在拥有指令流水线的 CPU 上更高效的运行。指令调度优化的一个必要前提就是 CPU 硬件支持指令并行，否则，指令调度是毫无意义的。

根据指令调度发生的阶段，可以把其分为静态调度和动态调度。

（1）静态调度：发生在程序编译时期。静态调度由编译器完成，在生成可执行文件之前通过指令调度相关优化，完成指令重排。

（2）动态调度：发生在程序运行时期。需要提供相应的硬件支持，比如乱序执行（OoOE: Out-of-Order Execution），此时指令的发射顺序和执行顺序可能是不一致，但是 CPU 会保证程序执行的正确性。

无论是静态调度还是动态调度，都是通过指令重排以提高指令流水，进而提高程序执行性能。静态调度和动态调度二者相辅相成，可以弥补对方的一些天然不足，协同完成指令流水优化，提高程序性能。

5.2　指令调度问题与约束

指令调度受到多方面的约束，如数据依赖约束、功能部件约束、寄存器约束等，在这些约束下，寻找到最优解，降低指令流水间的 stall，就是指令调度的终极目标。

指令流水间的 stall 主要由数据型冒险、结构性冒险、控制型冒险导致。

（1）数据型冒险：当前指令的执行依赖与上一条指令执行的结果。数据型冒险共有三种：写后读（RAW）、读后写（WAR）、写后写（WAW）。数据冒险可能产生数据流依赖。

（2）结构型冒险：多条指令同时访问一个硬件单元的时候，由于缺少相应的资源，导致结构型冒险。

（3）控制型冒险：存在分支跳转，无法预测下一条要执行的指令，导致控制型冒险。

编译器解决上述冒险的方法就是通过插入 `NOP` 指令，增加流水间的 stall 来化解冒险。

下面简单介绍一下三种数据型冒险（即数据依赖）：

（1）写后读（RAW）：一条指令读取前一条指令的写入结果。写后读是最常见的一种数据依赖类型，这种依赖被称为真数据依赖（true dependence）。

```
x = 1;
y = x;
```

（2）读后写（WAR）：一条指令写入数据到前一条指令的操作数。这种依赖被称为反依赖或反相关（anti dependence）。

```
y = x;
x = 1;
```

（3）写后写（WAW）：两条指令写入同一个目标。这种依赖被称为输出依赖（output dependence）。

```
x = 1;
x = 2;
```

5.3 静态指令调度

静态指令调度是编译器后端优化的一个非常重要的阶段。现代的处理器当中绝大部分都使用流水线结构，即指令是流水执行的，我们知道理想的流水线可能使得作业效率有成倍的提高，但是对应到计算机处理器当中的流水线，要想达到效率最优是非常困难的，这就是因为指令之间的数据依赖和结构相关所造成的，静态指令调度就是在还不知道程序某些动态信息和行为的情况下，根据所分析的指令之间依赖关系以及目标硬件架构的资源状况，对指令序列进行重排，从而减少流水线停顿，以期缩短程序的执行时间。

5.3.1 表调度算法（List Scheduling）

本文编译器中采用表调度（List Scheduling）算法，表调度是一种贪心 + 启发式方法，用以调度基本块中的各个指令操作，是基本块中指令调度的最常见方法。基于基本块的指令调度不需要考虑程序的控制流，主要考虑数据依赖、目标体系结构指令延迟、硬件资源、流水线情况等信息。

表调度的基本思想：维护一个用来存储已经准备执行的指令的 ready 列表和一个正在执行指令的 active 列表，ready 列表的构建主要基于数据依赖约束和硬件资源信息；根据调度算法以周期为单位来执行具体的指令调度，包括从列表中选择及调度指令，更新列表信息。

表调度算法大致分为以下三步：

（1）根据指令间依赖，建立依赖关系图。

（2）根据当前指令节点到根节点的长度以及指令的 latency，计算每个指令的优先级。

（3）不断选择一个指令，并调度它。使用两个队列维护 ready 的指令和正在执行的 active 的指令；在每个周期，选择一个满足条件的 ready 的指令并调度它，更新 ready 队列；检查 active 的指令是否执行完毕，更新 active 列表。

5.3.2 举个例子

假设当前 CPU 有两个计算单元，即每个周期可以执行两条指令。加法指令的 latency 为 2 cycles，其他指令为 1 cycle。以下面代码为例。

```
r0: a = 1;
r1: f = a + x;
r2: b = 7;
r3: c = 9;
r4: g = f + d;
r5: d = 13;
r6: e = 19;
r7: h = f + c;
r8: j = d + y;
r9: z = -1;
r10: JMP L1;
```

（1）根据数据依赖关系构建出依赖关系图。如图 24 所示。

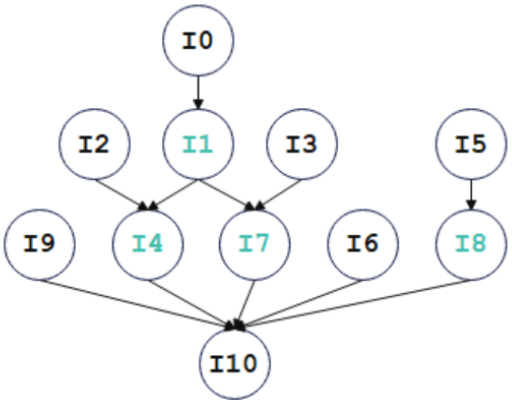

图 5.24　依赖关系图

（2）计算指令节点的优先级。优先级计算公式为：

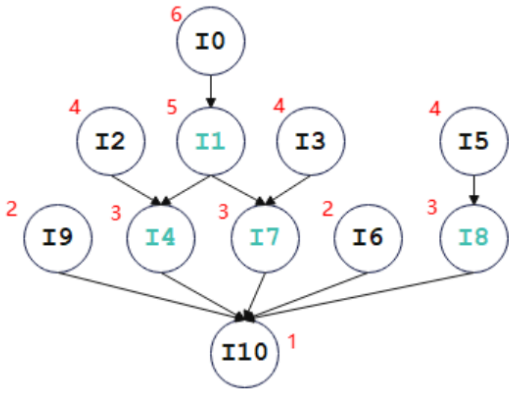

图 5.25　计算指令节点的优先级

5.4　动态指令调度

所谓静态指令调度就是在编译阶段由编译器实现的指令调度，目的是通过调度尽量地减少程序执行时由于数据相关而导致的流水线暂停即处理器空转。所以静态指令调度方法也叫做编译器调度法。由于在编译阶段程序并没有真正执行，有一些相关可能未被发现，这是静态指令调度根本无法解决的问题。

修改后的 LLVM IR 可以进一步编译为二进制可执行文件，然后在基于 RISC-V 存算一体模拟器中执行，初始阶段，所有的指令和数据均统一存储在主存储器里。当程序启动并开始运行时，CPU 便承担起指令调度的重任。

对于 CIM 加速指令，它们会被识别并从常规指令流中分离出来，随后卸载到专门的 NPU 上进行高效执行。而除 CIM 加速指令之外的其他普通指令，则依然按照传统的执行流程，由 CPU 本身负责一步步完成。

在 CIM 加速指令于 NPU 上执行完毕之后，所产生的计算结果会及时地返回给 CPU。CPU 收到这些结果后，会进一步将其发送回主存储器中，以便后续程序阶段的调用和使用。

为了确保 CPU 与 NPU 之间的协同工作能够高效且准确地进行，我们在系统中引入了同步指令（Synchronous Instruction）。当应用程序调用 CIM 加速指令时，就如同调用本地的一个普通函数一般，程序的执行流程会在该加速指令处等待。只有当 NPU 上的 CIM 加速指令执行完成，并且相关的结果数据安全地回传至 CPU 之后，程序才会从等待状态中恢复，继续向下执行后续的指令。

通过这种设计，有效地避免了 CPU 和 CIM 加速器之间可能出现的数据竞争问题。一方面，保证了整个程序执行过程的正确性和数据的一致性；另一方面，充分发挥了 CIM 加速器在特定计算任务上的性能优势，提升了整个系统的运行效率。

系统架构图如图 26 所示，更直观地呈现了指令动态调度过程中各组成部分之间的关系以及数据流向。

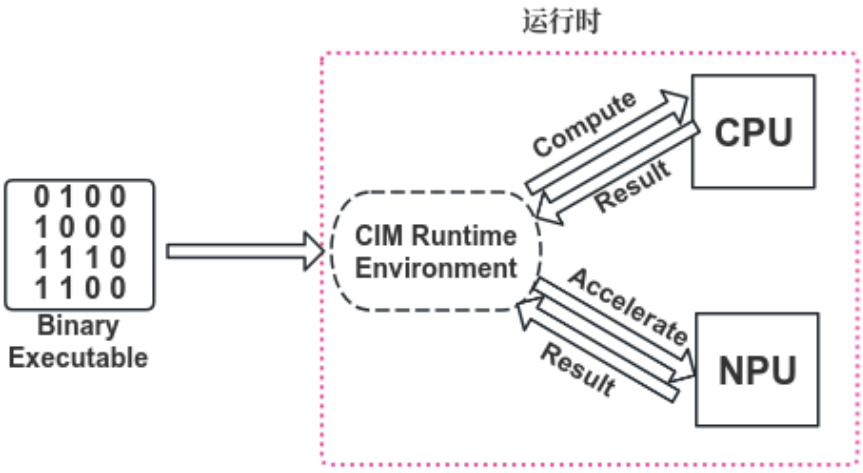

图 5.26　指令调度

5.5　本章小结

本章主要介绍了如何在 CPU 和 NPU 异构计算单元之间进行指令的静态和动态调度。当编译器后端智能识别出 NPU 加速指令后，如何在异构架构下调度生成的 RISC-V 指令来充分发挥 SRAM 存算一体阵列高能效、高算力密度的硬件优势。

本章深入探讨了在基于 RISC-V 存算一体芯片的异构架构下，CPU 和 NPU 之间的指令调度策略。通过对指令的静态和动态调度机制的详细阐述，展示了如何合

理安排 RISC-V 指令的执行顺序，使得 NPU 加速指令能够充分调用 SRAM 存算一体阵列的强大计算资源，发挥其高能效和高算力密度的优势。同时，协调 CPU 与 NPU 之间的工作，保证整个计算过程的高效性和正确性，为提高系统的整体性能提供了关键支撑。

第 6 章　编译器测试与分析

本文在第 3 章从编译器前端、中端以及后端介绍了基于 RISC-V 存算一体加速器设计的编译器的总体架构，分别在第 4 章和第 5 章着重介绍了如何智能识别 NPU 加速指令以及指令调度，来充分发挥存算一体的优势。本章主要在 RISC-V 存算一体模拟器上对深度学习网络中常见的算子开展功能性验证和性能测试，并选取自定义的 FASHION MNIST 网络模型作为实例，同时呈现最终的测试结果。

6.1　编译器功能测试

FASHION-MNIST 是一个替代 MNIST 手写数字集的图像数据集。它是由 Zalando 旗下的研究部门提供的，其涵盖了来自 10 种类别的共 70000 个不同商品的正面图片。FASHION-MNIST 的大小、格式和训练集/测试集划分与原始的 MNIST 完全一致。60000/10000 的训练测试数据划分，$28 * 28$ 的灰度图片。

本次实验使用的网络模型为我们自定义的 FASHION MNIST 网络模型，该网络模型使用 FASHION MNIST 数据集进行训练。实验使用的自定义的 FASHION MNIST 网络模型结构如表 3 所示，ONNX 可视化部分结果如图 27 所示。该网络模型由 3 个卷积层和 2 个全连接层构成，除最后一个全连接层外每个层的输出都使用 ReLU 函数进行激活。

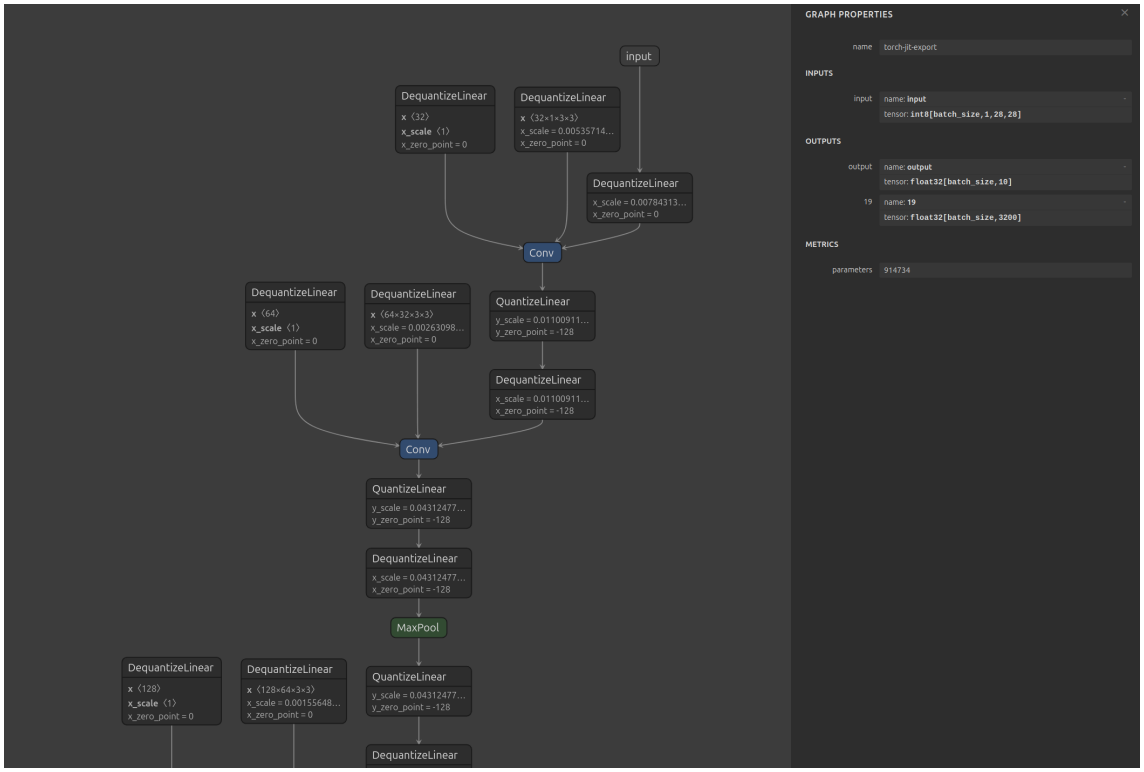

图 6.27　网络架构图

```
x1 = conv2d(input_image, weight_1, bias_1)
x2 = x1 * scale_1
x2 = clip(x2)

x3 = conv2d(x2, weight_2, bias_2)
x3 = x3 * scale_2
x4 = clip(x3)

x5 = maxpool2d(x4, stride_1)

x6 = conv2d(x5, weight_3, bias_3)
x6 = x6 * scale_3
x7 = clip(x6)

x8 = maxpool2d(x7, stride_2)

x9 = flatten(x8)

xa = linear(x9)
xa = xa * scale_4
xb = clip(xa)
```

```
xc = linear(xb)
xc = xc * scale_5

xd = log_softmax(xc)
```

表 6.3　自定义 FASHION MNIST 网络模型结构

层类型	核尺寸/步长
Conv	$3 * 3/1, 1$
Conv	$3 * 3/1, 1$
MaxPool	$2 * 2/2, 2$
Conv	$3 * 3/1, 1$
MaxPool	$2 * 2/2, 2$
Flatten	-
Full_Connect	-
Full_Connect	-
LogSoftmax	-

本次实验基于 FASHION MNIST 数据集，采用自定义的 FASHION MNIST 网络模型，通过所设计的针对 RISC-V 存算一体加速器的编译器进行编译并执行推理任务，来评估编译器在该特定场景下的性能表现以及能效情况。同时，使用 Python 来模拟该网络模型，记录模型的每一层的输出以及最终输出的计算结果，对编译器的功能性进行对比验证。

经过验证对比，我们发现通过本文编译器编译 ONNX 模型和使用 Python 模拟该网络模型，其每一层的输出以及最终输出的计算结果一致，故，本文编译器的功能性得到了验证。

```
* * * * Performance Analysis * * * *
NPU work ratio: 95%
  Off-chip Transfer ratio: 65%
  Tensor Manipulate ratio: 3%
  Matrix Processing ratio: 15%
  Vector Processing ratio: 16%
CIM Analysis:
  CIM Compute ratio: 6.0%
  CIM Space Utilization: 69.9%
  CIM Utilization: 4.2%
  Effective Performance: 42.804GOPS @INint8-Wint8
```

从性能分析结果来看，我们发现在此次推理过程中 NPU 的工作比例达到了 95%，矩阵处理占比 15%，向量处理占比 16%，二者合计占比 31%，Tensor Manipulate 占比 3%，这表明编译器能够有效地识别、映射中间表示到指定的张量、向量等 RISC-V 扩展指令，以通过 NPU 核心加速，并根据程序依赖生成必要的同步指令在保证 CPU 和 NPU 协同计算的正确性的同时挖掘计算的并发性，体现了编译器在指令调度和资源分配方面的良好性能。关于 Tensor Manipulate 占比之所以这么低，是因为在底层硬件设计中 Tensor Manipulate 表示的操作是片上内存之间的数据搬运，而在推理过程中片上存储之间的数据搬运很少，所以 Tensor Manipulate 的占比很低。然而，片上片外数据传输（Off-Chip Transfer）比例却占据了 65%，成为 NPU 内部最大耗时环节，凸显“内存墙”问题。此现象与 RISC-V 存算一体芯片的层次化存储架构相关。当我们将某些计算卸载到 CIM 加速器的 NPU 核心进行加速计算时，我们需要把对应的参数的权重、偏置等等从片外内存传输到片上内存，成为影响系统整体性能的关键瓶颈。特别是该 FASHION MNIST 网络模型最后两层都是全连接层，性能受限于该全连接层的权重搬运。

在存算一体（CIM）相关分析中，CIM 计算比例仅为 6.0%，利用率仅为 4.2%，这主要是因为片上片外数据传输（Off-Chip Transfer）比例占据了 65%，特别是该 FASHION MNIST 网络模型最后两层都是全连接层，性能受限于该全连接层的权重搬运，所以 CIM 的计算比例和利用率很低。CIM 空间利用率达到了 69.9%，这一数值表明 CIM 内部存储空间的利用率处于一个相对合理的水平，编译器后端内存分配管理充分利用了 CIM 的存储资源，减少了不必要的存储冗余。CIM 有效性能为 42.804 GOPS @INint8 - Wint8，在 INT8 量化下，NPU 实现了 42.8 GOPS 有效算力。

```
* * * * Power Analysis * * * *
- - - - NPU Level - - - -
  Spad R/W energy cost: 3491064.32pJ, ratio:44%
  PE vector energy cost: 0.0pJ, ratio:0%
  CIM R/W energy cost: 1295134.72pJ, ratio:16%
  CIM compute energy cost: 3120578.56pJ, ratio:39%
  NPU energy cost: 7906777.6pJ
  CIM Compute Energy Efficiency: 12.5TOPS/W
  NPU Energy Efficiency: 4.93TOPS/W @INint8-Wint8
- - - - System Level - - - -
  NPU ENERGY
    NPU energy cost: 7906777.6pJ, ratio:9%
  CPU ENERGY
    CPU energy cost: 0pJ, ratio:0%
```

```
Off-chip ENERGY
  pSRAM energy cost: 75476480pJ, ratio:91%
Energy Efficiency
  Total energy cost: 83383257.6pJ
  Total Energy Efficiency: 0.47TOPS/W @INint8-Wint8
```

在能效分析方面，于 NPU 级别，Spad R/W 能量成本占比最高，达到了 44%，Scratchpad 频繁读写与高片上片外数据传输占比（Off-Chip Transfer）形成了因果关系，这凸显了芯片内部存储单元的读写操作对能量消耗的显著影响，这提示我们在未来的编译器优化过程中，需要重点关注如何降低 Spad 的读写频率或优化其读写策略，以减少这部分的能量开销。CIM R/W 能量成本占比为 16%，CIM 计算能量成本占比为 39%，这表明 CIM 的能量消耗主要集中在计算过程以及数据的读写操作上，这也与 CIM 的工作原理和特点相符合。CIM 计算能效为 12.5 TOPS/W，而 NPU 能效为 4.93 TOPS/W @INint8 - Wint8，相比之下，CIM 在能效方面表现出了一定的优势，这也进一步证明了存算一体架构在能效提升方面的潜力，但同时也需要注意到整个 NPU 的能效还有较大的提升空间，需要综合考虑编译器的优化策略以及硬件架构的改进来进一步提高能效。

在系统级别，NPU 的能量成本占比为 9%，而片上片外内存传输（Off-Chip）的能量成本占比高达 91%，这再次凸显了片上片外数据传输对系统整体能量消耗的主导地位，这一结果与性能分析中的片上片外数据传输占比（Off-Chip Transfer）比例较高的现象相呼应，进一步强调了减少片上片外数据传输对于提升系统能效的重要性。整个系统的总能量成本为 83383257.6 pJ，总能效仅为 0.47 TOPS/W @INint8 - Wint8，这一较低的系统能效值凸显片上片外数据传输对能效的毁灭性影响，需硬件-编译器协同设计：如采用 3D 堆叠内存等等。

6.2 编译器性能测试

本次实验不仅基于 FASHION MNIST 数据集完成了自定义网络模型的推理任务，还对一些常见的算子在利用 NPU 和不利用 NPU 两种情况下进行了详细测试与对比，性能测试结果如表 5 所示，深度学习常见的算子的分类如表 4 所示。

表 6.4　测试算子类别表

算子类别	举例
逐元素操作算子	Add, Multiply, Equal, And, Quantization
乘累加算子	Conv, GEMM, Full_Connect
激活函数算子	Exp, Sigmoid, Tanh, ReLu, Leaky_ReLu, Softmax
归一化算子	Layer_Normalization
数据排布算子	ReduceMax, ArgueMax, Transpose, Clip, Max_Pooling

表 6.5　算子测试结果对比（单位：Cycle）

算子名称	**CPU**	**CPU + NPU**	加速比
Add	818412	28709	28.51
Multiply	769260	28709	26.8
Equal	1457989	28709	50.79
And	1351485	28709	47.08
Quantization	1150040	98922	11.63
Conv	6353602	8434	753.34
GEMM	694459	3019	230.02
Full_Connect	1047333	3339	313.67
Exp	33307840	1878977	17.73
Sigmoid	7314928	1878977	3.89
Tanh	36234072	1878977	19.28
ReLu	962011	18080	53.21
Leaky_ReLu	986634	76925	12.83
Softmax	30612059	2008261	15.24
Layer_Normalization	1391198	89989	15.46
ReduceMax	813208	15937	51.03
ArgueMax	791174	326694	2.42
Transpose	794094	1782	445.62
Clip	987623	34087	28.97
Max_Pooling	826146	20145	41.01

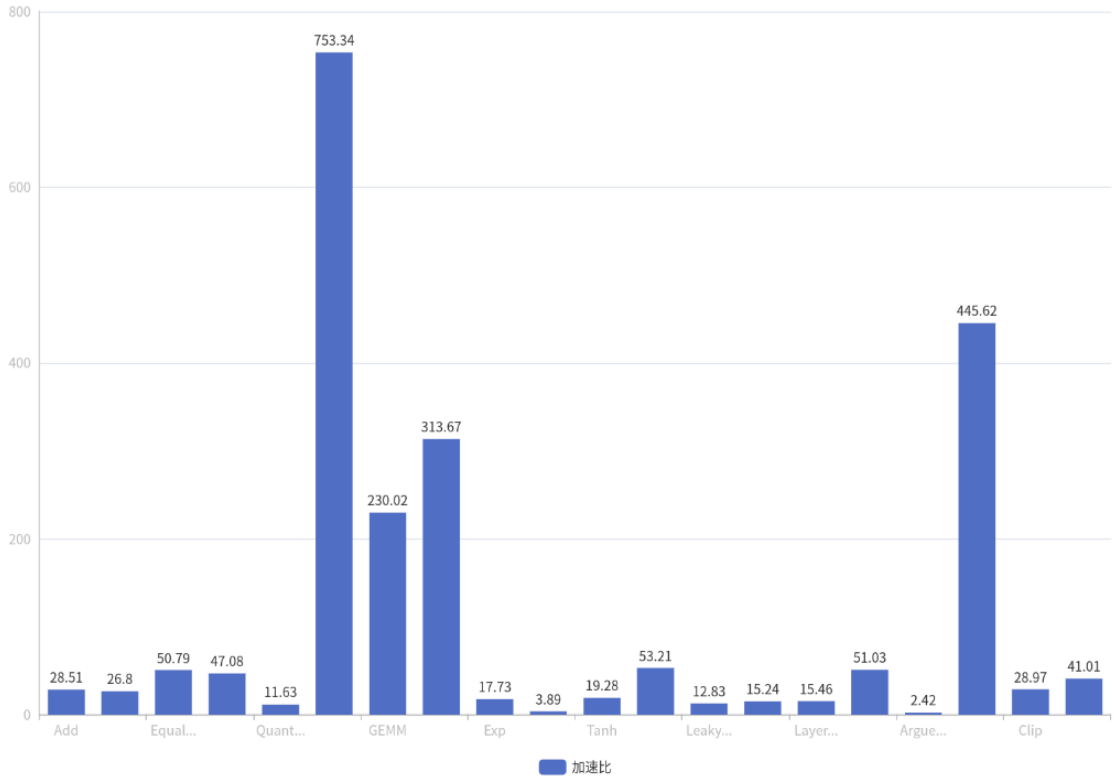

图 6.28　算子测试结果

从测试结果可以看出，NPU 的引入为大多数算子带来了显著的加速效果。其中，Conv 算子的加速比高达 753.34，GEMM 算子的加速比也达到了 230.02，这表明编译器能够有效利用 NPU 的加速计算的能力，优化卷积和矩阵运算等计算密集型操作的执行效率。此外，像 Add、Multiply、Equal 等基础算术和逻辑运算算子，以及 Leaky_ReLu、Clip 等激活函数相关的算子，其加速比也都在 20 以上，充分体现了 NPU 在处理这些常见算子时的优势。

然而，对于某些特定算子，如 ArgueMax，其加速比仅为 2.42，相对较低。这是因为该算子本身的计算特性与 NPU 的架构优势不完全匹配。此外，像 Exp、Sigmoid、Tanh 等算子，虽然也取得了一定的加速效果，但加速比相较于其他算子并不算突出，这提示我们在后续的编译器优化工作中，可以重点关注这类算子，深入分析其计算模式和数据访问特点，进一步挖掘 NPU 的潜在性能。

综合来看，本次实验结果表明，本文所设计的编译器能够有效利用 NPU 的硬件资源，为大多数算子带来显著的性能提升，但针对个别特殊算子的优化仍有改进空间，以实现更广泛的算子加速效果，进而提升整个神经网络模型在 RISC-V 存算一体加速器上的执行效率。

6.3 本章小结

本章对编译器的整体功能进行了功能性测试和性能测试，并进行了结果分析，验证了其基本功能的完整性。实验表明本文的编译器可以成功的将深度学习模型编译为等价的 RISC-V 通用指令和 RISC-V 加速指令，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征。

第 7 章　总结与展望

7.1　工作总结

面向 RISC-V 芯片构建编译器方面，现有人工智能编译社区提出了统一的中间表示 MLIR，已经在 IREE、Triton 等多个 AI 编译器中得到应用。然而，这些编译器并不能感知存算一体芯片架构，以及其使用的 RISC-V 扩展指令集，也无法表征存算芯片的计算、并发、协同、通信、数据重用等特征。因此本文提出了面向 RISC-V 存算一体加速器的编译器设计与实现，经过总结，本文完成的主要工作如下：

（1）对本文所基于的 RISC-V 存算一体加速器进行了深度学习编译器的基本设计，在设计实现中可以解析 ONNX 模型为计算图，通过算子融合等优化方法减少了加速器对内存的访问以及存储空间的浪费，同时还使用内存分配地址叠加的方式避免了各张量之间的数据覆盖。

（2）智能识别 NPU 加速指令。为了在 RISC-V 存算一体加速器中利用到 NPU 核心进行加速，我们从 LLVM IR 中识别出几种典型的计算模式，这些模式不受高级编程范例的影响。我们在 LLVM IR 中间表示上进行应用特征分析，识别出可加速的 LLVM IR 范式，以自动将它们卸载到 CIM 加速器的 NPU 核心进行加速计算，转为特定的 RISC-V 加速指令，充分利用 RISC-V 通用核心、NPU 加速核心的高效能特征。

（3）指令的动态调度。在 CPU 和 NPU 异构计算单元之间进行指令的动态调度。在异构架构下，充分利用 RISC-V 已有指令集实现在执行 AI 任务运行时对各类计算资源的灵活调度，充分发挥 SRAM 存算一体阵列高能效、高算力密度的硬件优势。

7.2　工作展望

深度学习与 AI 芯片都是目前的热门研究领域，作为连接深度学习算法及其实现的深度学习编译器也得到了越来越多的关注。本文基于 RISC-V 存算一体加速器设计的深度学习编译器实现了智能识别 NPU 加速指令以及运行时在 CPU 和 NPU 异构计算单元之间进行指令的动态调度，达到了本文的设计目的。但本文编译器未来仍有改进空间：

1. 加强编译器对不同深度学习模型和任务的适配性。随着深度学习领域的不断发展，新的模型架构和任务类型不断涌现，需持续完善编译器的模型解析和优化模块，使其能够更好地支持更多的模型和任务场景，满足多样化的应用需求。
2. 进一步优化编译器的性能。目前虽然在算子融合、指令识别与卸载等方面取得了一定成果，但仍可探索更高效的算法和策略，以获取更高的计算效率和更低的内存占用，例如引入机器学习算法智能来地决策算子融合的顺序和程度等。

致　谢

时光荏苒，转眼间我的大学本科生活即将画上句号。回首这四年的点点滴滴，心中充满了无尽的感慨与思绪。在毕业论文完成之际，我愿将这四年的经历与感悟凝聚成文字，向求学路上给予我帮助的师长和亲友表达我最真挚的谢意。

师恩如海，深不可测。首先，我要特别感谢我的导师菩提教授。从初入大学时的懵懂无知，到如今能够独立完成毕业设计，菩老师始终是我前行路上的明灯。他不仅在学术上给予我悉心的指导，帮助我拓宽视野，提升能力，还在生活中给予我无微不至的关怀，让我感受到如家人般的温暖。在这次毕业设计的过程中，从选题到实验，从撰文到定稿，菩老师的全程指导让我受益匪浅。每一次对实验结果的精益求精，每一次对论文的反复修改，都让我深刻体会到菩老师在科研工作中的严谨态度和对学生的严格要求。在师门的四年时光里，菩老师不仅传授给我学术知识，更教会了我踏实、认真、负责、勤勉的品质，这些品质将伴随我一生，无论是在科研还是其他工作中，甚至在日常生活中。在此论文完成之际，我衷心感谢菩老师一路以来的教导、呵护与关怀。

参考文献

[1] REAL E, MOORE S, SELLE A, 等. Large-Scale Evolution of Image Classifiers[EB/OL](2017). https://arxiv.org/abs/1703.01041

[2] LEE L. Book Reviews: Foundations of Statistical Natural Language Processing[J/OL]Computational Linguistics, 2000, 26(2). https://aclanthology.org/J00-2011/

[3] HA J W, PYO H, KIM J. Large-Scale Item Categorization in e-Commerce Using Multiple Recurrent Neural Networks[C/OL]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data MiningSan Francisco, California, USAAssociation for Computing Machinery, 2016: 107-115. https://doi.org/10.1145/2939672.2939678. DOI:10.1145/2939672.2939678

[4] CHEN H, ENGKVIST O, WANG Y, 等. The rise of deep learning in drug discovery[J/OL]Drug Discovery Today, 2018, 23(6): 1241-1250. https://www.sciencedirect.com/science/article/pii/S1359644617303598. DOI:https://doi.org/10.1016/j.drudis.2018.01.039

[5] O'SHEA K, NASH R. An Introduction to Convolutional Neural Networks[EB/OL](2015). https://arxiv.org/abs/1511.08458

[6] SHERSTINSKY A. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network[J/OL]Physica D: Nonlinear Phenomena, 2020, 404: 132306. http://dx.doi.org/10.1016/j.physd.2019.132306. DOI:10.1016/j.physd.2019.132306

[7] HOCHREITER S, SCHMIDHUBER J. Long Short-Term Memory[J/OL]Neural Comput., 1997, 9(8): 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735. DOI:10.1162/neco.1997.9.8.1735

[8] GOODFELLOW I J, POUGET-ABADIE J, MIRZA M, 等. Generative Adversarial Networks[EB/OL](2014). https://arxiv.org/abs/1406.2661

[9] SABNE A. XLA : Compiling Machine Learning for Peak Performance[Z]2020

[10] CHEN T, MOREAU T, JIANG Z, 等. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning[C/OL]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)Carlsbad, CAUSENIX Association, 2018: 578-594. https://www.usenix.org/conference/osdi18/presentation/chen

[11] ROESCH J, LYUBOMIRSKY S, WEBER L, 等. Relay: a new IR for machine learning frameworks[C/OL]//Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming LanguagesPhiladelphia, PA, USAAssociation for Computing Machinery, 2018: 58-68. https://doi.org/10.1145/3211346.3211348. DOI:10.1145/3211346.3211348

[12] FENG S, HOU B, JIN H, 等. TensorIR: An Abstraction for Automatic Tensorized Program Optimization[C/OL]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2Vancouver, BC, CanadaAssociation for Computing Machinery, 2023: 804-817. https://doi.org/10.1145/3575693.3576933. DOI:10.1145/3575693.3576933

[13] ZHAO J, LI B, NIE W, 等. AKG: automatic kernel generation for neural processing units using polyhedral transformations[C]//PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation2021

[14] TILLET P, KUNG H T, COX D. Triton: an intermediate language and compiler for tiled neural network computations[C/OL]//Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming LanguagesPhoenix, AZ, USAAssociation for Computing Machinery, 2019: 10-19. https://doi.org/10.1145/3315508.3329973. DOI:10.1145/3315508.3329973

[15] iree[Z]https://github.com/iree-org/iree

[16] CHEN Y, CHEN T, XU Z, 等. DianNao family: energy-efficient hardware accelerators for machine learning[J/OL]Commun. ACM, 2016, 59(11): 105-112. https://doi.org/10.1145/2996864. DOI:10.1145/2996864

[17] JOUPPI N P, YOUNG C, PATIL N, 等. In-Datacenter Performance Analysis of a Tensor Processing Unit[C/OL]//Proceedings of the 44th Annual International Symposium on Computer ArchitectureToronto, ON, CanadaAssociation for Computing Machinery, 2017: 1-12. https://doi.org/10.1145/3079856.3080246. DOI:10.1145/3079856.3080246

[18] CHEN Y H, KRISHNA T, EMER J S, 等. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks[J/OL]IEEE Journal of Solid-State Circuits, 2017, 52(1): 127-138. DOI:10.1109/JSSC.2016.2616357

[19] CHEN Y H, YANG T J, EMER J, 等. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices[EB/OL](2019). https://arxiv.org/abs/1807.07928

[20] YIN S, OUYANG P, TANG S, 等. A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications[J/OL]IEEE Journal of Solid-State Circuits, 2018, 53(4): 968-982. DOI:10.1109/JSSC.2017.2778281

[21] BORGHETTI J, SNIDER G S, KUEKES P J, 等. 'Memristive' switches enable 'stateful' logic operations via material implication[J]NATURE, 2010(7290): 873-876

[22] CHI P, LI S, XU C, 等. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory[C/OL]//2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA): 卷 02016: 27-39. DOI:10.1109/ISCA.2016.13

[23] CUI E, LI T, WEI Q. RISC-V Instruction Set Architecture Extensions: A Survey[J/OL]IEEE Access, 2023, 11: 24696-24711. DOI:10.1109/ACCESS.2023.3246491

[24] LATTNER C, ADVE V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]//Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime OptimizationPalo Alto, CaliforniaIEEE Computer Society, 2004: 75

[25] PASZKE A, GROSS S, MASSA F, 等. PyTorch: an imperative style, high-performance deep learning library[M]//Proceedings of the 33rd International Conference on Neural Information Processing SystemsRed Hook, NY, USACurran Associates Inc., 2019

[26] ABADI M, BARHAM P, CHEN J, 等. TensorFlow: a system for large-scale machine learning[C]//Proceedings of the 12th USENIX Conference on Operating Systems Design and ImplementationSavannah, GA, USAUSENIX Association, 2016: 265-283

[27] onnx[Z]https://github.com/onnx/onnx

[28] netron[Z]https://github.com/lutzroeder/netron

[29] ADARSH P, RATHI P, KUMAR M. YOLO v3-Tiny: Object Detection and Recognition using one stage improved model[C/OL]//2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS): 卷 02020: 687-694. DOI:10.1109/ICACCS48705.2020.9074315