编号：SP9100242022212726　　密级：公开

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 i
ABSTRACT ii
第 1 章　绪论 1
1.1　研究背景与意义 1
1.2　国内外研究现状 3
1.2.1　深度学习编译器 3
1.2.2　深度学习加速器 8
1.3　论文的主要研究工作 9
1.4　论文的章节安排 10
第 2 章　主要技术基础 12
2.1　RISC-V 12
2.2　存算一体架构 13
2.3　LLVM 编译器 15
2.3.1　LLVM 框架 15
2.3.2　LLVM IR 16
2.3.3　LLVM 后端编译 16
2.3.3.1　模块加载与解析 16
2.3.3.2　优化处理 16
2.3.3.3　寄存器分配 17
2.3.3.4　指令选择与调度 17
2.3.3.5　代码生成 17
2.4　本章小结 17
第 3 章　面向 RISC-V 存算一体加速器的编译器 18
3.1　编译器总体架构 18
3.2　编译器前端 20
3.2.1　ONNX 20
3.2.2　构建计算图 22
3.3　编译器中间优化器 23
3.3.1　算子融合 23
3.3.2　公共子表达式消除 25
3.3.3　死代码删除 26
3.4　编译器后端 27
3.4.1　内存分配管理 27

3.4.2 计算逻辑管理 28
3.5 本章小结 29
第 4 章 NPU 指令的智能识别 30
4.1 LLVM IR 与可加速范式 30
4.2 向量 - 向量操作范式 31
4.3 矩阵 - 向量操作范式 33
4.4 矩阵 - 矩阵操作范式 35
4.5 本章小结 37
第 5 章 指令调度 38
5.1 指令调度简介 38
5.2 指令调度问题与约束 38
5.3 静态指令调度 40
5.4 动态指令调度 42
5.5 本章小结 44
第 6 章 编译器测试与分析 45
6.1 编译器功能测试 45
6.2 编译器性能测试 51
6.3 本章小结 53
第 7 章 总结与展望 54
7.1 工作总结 54
7.2 工作展望 54
致 谢 56

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，将面向 RISC-V SRAM 存算一体芯片修改 LLVM 编译器，实现对存算指令的支持，这包括但不限于对指令集的扩展、内存模型的适配、NPU 指令的智能识别、指令调度以及优化策略的调整等。通过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 project is committed to solving the above problems. The LLVM compiler will be modified for RISC-V SRAM CIM chips to support CIM instructions, including but not limited to the expansion of the instruction set, the adaptation of the memory model, the intelligent identification of NPU instructions, instruction scheduling, and the adjustment of optimization strategies. By analyzing the application characteristics of the application, the computing parts that can be accelerated are identified and converted into specific RISC-V acceleration instructions, and the existing RISC-V instruction set is fully utilized to realize the flexible scheduling of various computing resources when executing AI tasks, giving full play to the hardware advantages of the high energy efficiency and high computing density of the SRAM storage and computing integrated array.

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

近几年来，深度学习在计算机视觉 CV[1] 、自然语言处理 NLP[2] 、语音识别 Audio[3] 等领域不断刷新着人们对技术可能性的认知边界。例如在自然语言处理方面，深度学习让机器能够理解人类语言的语义和情感，从而实现高质量的机器翻译[4] 等功能。这些成就的取得，很大程度上得益于卷积神经网络 CNN[5] 、循环神经网络 RNN[6] 、图神经网络 GNN[7] 和生成对抗网络 GAN[8] 等新型 AI 模型结构的相继涌现。然而，随着模型复杂度的不断增加，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，物联网终端设备产生的数据量突破 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，由于冯诺依曼架构中存储单元与计算单元物理分离，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来交换数据。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 倍的速度增长，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，在冯诺依曼架构下，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。如此以来，数据在片外存储与运算核心之间的频繁往返迁移，引发了严重的传输功耗问题。根据英特尔的研究显示[9]，当半导体工艺迈入 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已经成为芯片发展的关键制约因素之一。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。

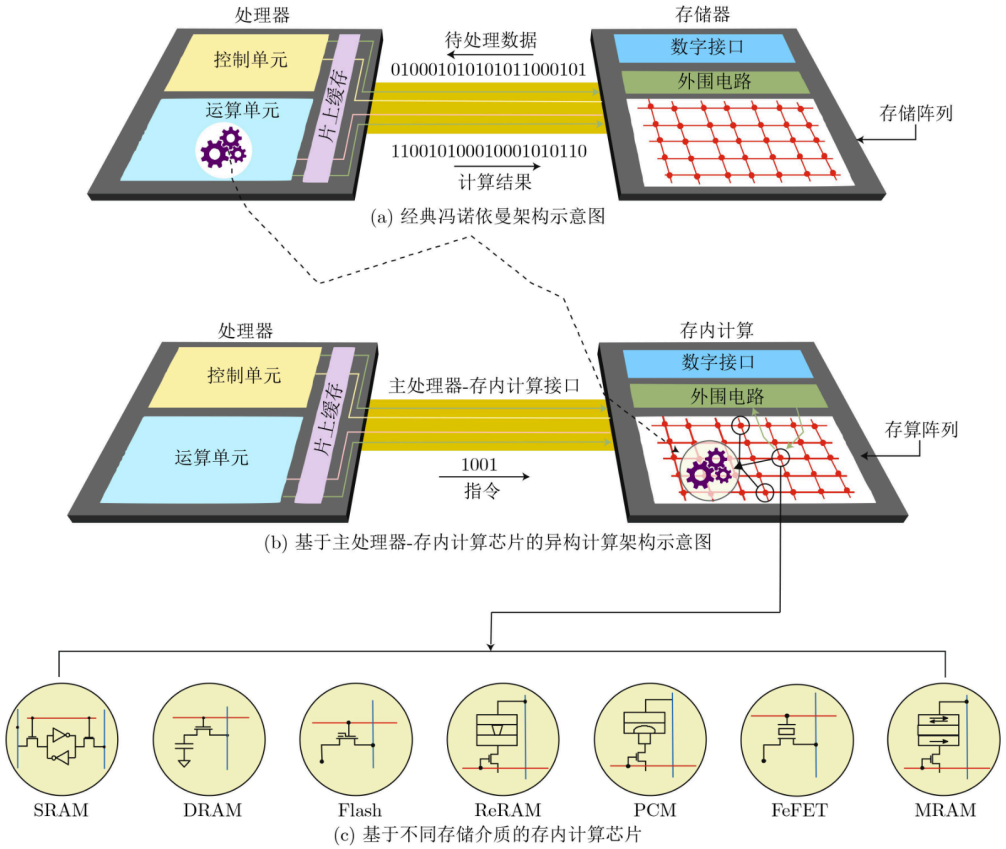

图 1.1　基于不同存储介质的计算架构演变图[10]

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[11]。

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。基于 RISC-V 扩展指令构建的加速核心在功能与指令支持上，与基于标准 RISC-V 指令集构建的通用 CPU 核心存在本质差异；另一方面，芯片内部通常集成了多种类型的计算加速单元，如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，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，本课题将面向 RISC-V SRAM 存算一体芯片修改 LLVM 编译器，构建针对 NPU/CPU 异构协同的编译支持体系。

一是数据搬运瓶颈的解决。 CIM 加速核的高吞吐来源于阵列内原位乘累，但频繁的片外/片上 SRAM 与通用 CPU 之间往返传输会大幅耗时耗能。为此，我们在 LLVM 编译器后端引入张量访问模式静态分析，并基于此自动生成数据分块（tiling）策略，将输入特征和权重局部化到 SRAM Scratchpad；同时采用双缓冲流水线，在前一块数据上计算时异步预取下一块，最后通过循环重叠将搬运指令与 CIM 计算指令并行调度，最大程度隐藏数据移动延迟。

二是针对 CIM 架构的指令级特化与调度优化。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 核心进行加速计算，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。

以上两方面改进，将实现从应用特征分析到 CIM 指令生成，再到 NPU/CPU 异构单元的动态调度全过程，为 RISC-V 存算一体芯片提供有力的编译支持。

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）[12]：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。

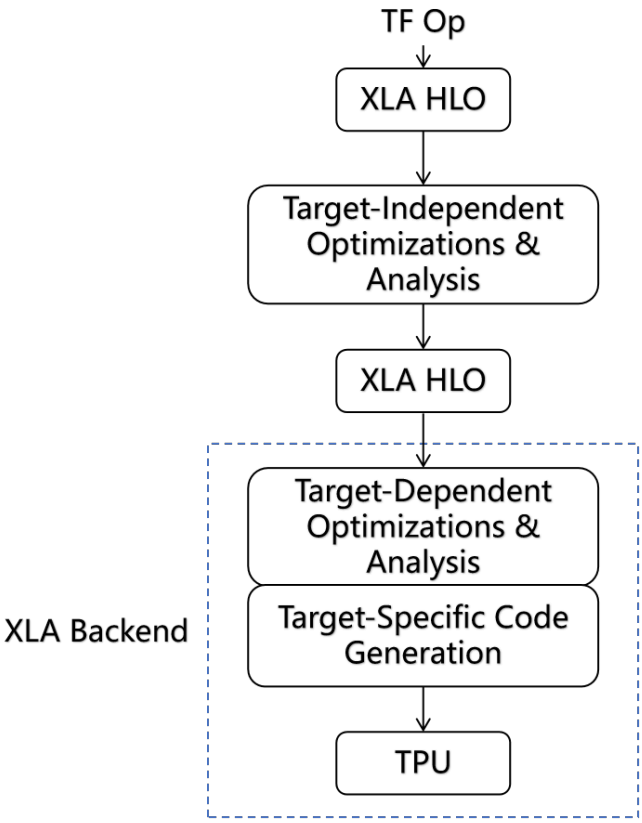

图 1.2　XLA 架构图[12]

TVM（Tensor Virtual Machine）[13]：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[14] 和 Tensor IR[15] 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。

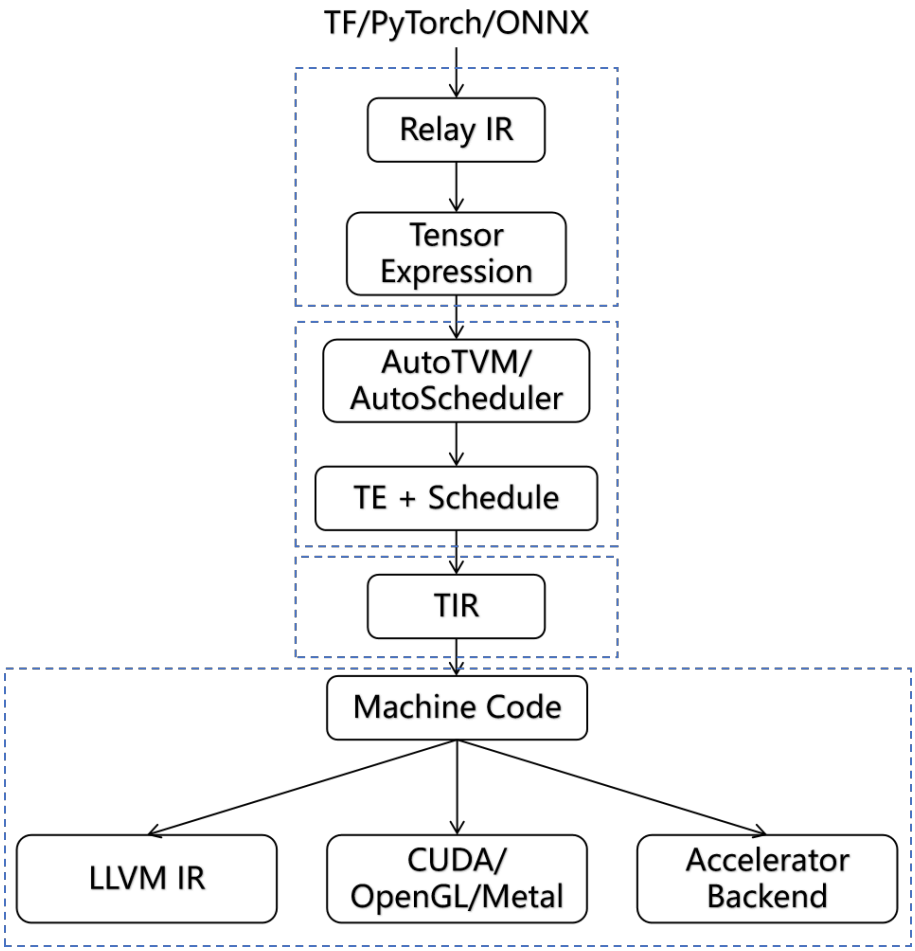

图 1.3　TVM 架构图[13]

MindSpore AKG[16]：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。

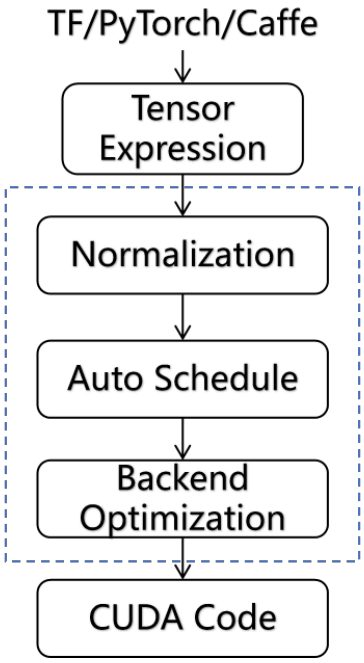

图 1.4　AKG 架构图[16]

Triton[17]：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。

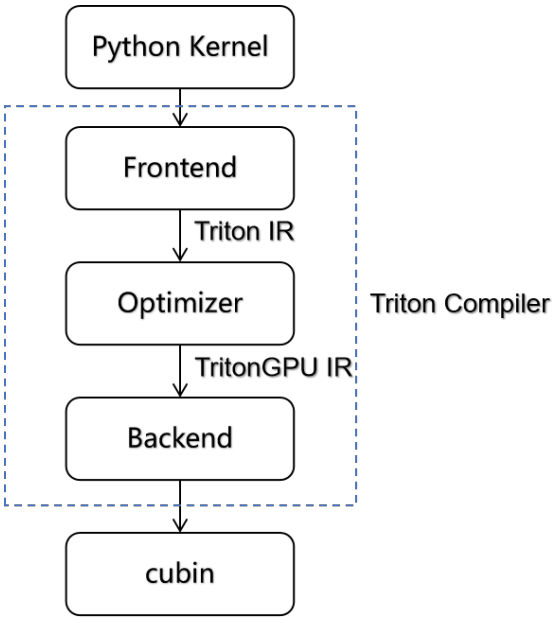

图 1.5　Triton 架构图[17]

IREE[18]：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。

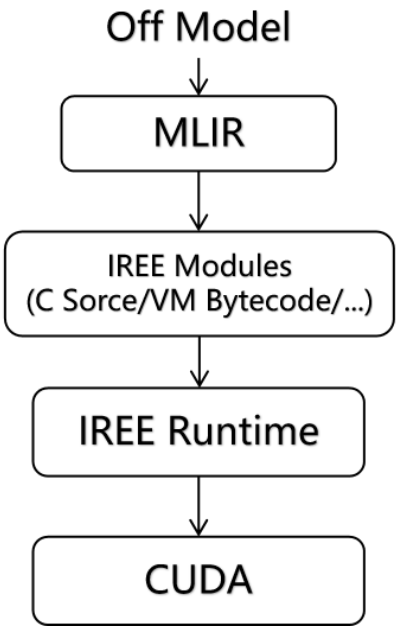

图 1.6　IREE 架构图[18]

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

随着深度学习技术在多领域的广泛应用，为其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三个层面展开：

首先，在基于传统计算架构的专用定制方面，学术界与工业界纷纷推出全定制 ASIC 和可重构计算平台以提升矩阵乘加性能。寒武纪 DIANNAO[19]（2014 - 2016）系列通过分层缓存和定制数据流最大化数据复用；Google TPU[20]（2016）采用大规模 systolic array 并行处理；Chen et al.[21,22] 的 Eyeriss（2017/2019）引入 Row-Stationary 策略显著降低带宽与能耗；同时，清华 Thinker 团队基于 CGRA 的可重构加速器可动态调度算子[23]。FPGA 领域亦涌现出多种基于 Xilinx/Intel 平台的 DNN IP 核，支持灵活网络拓扑与低延迟推理。

其次，近存（Near-Memory）计算通过将轻量化算术单元下推至高带宽存储体（如 HBM 或 DRAM bank）来减少数据搬运开销。Samsung PIM[24] 在 HBM2 外围集成向量 ALU，并利用 TSV 总线实现大带宽张量运算；DRAM-PIM 原型 AMBIt[25] 和 UPMEM[26] 则将位线/字线内置简单逻辑，可在 DRAM bank 级别完成点积或置位操作；3D-stacked DRAM[27]（如 HBM/DDR5）则通过逻辑层与存储层的紧耦合，为 Near-Memory 应用提供了更大容量与更低延迟。

最后，存内（Compute-in-Memory）计算则更进一步，将大规模乘累运算彻底挪至存储阵列内部。UCSB 提出的 PRIME[28]（2016）使用浮栅晶体管阵列完成存内乘加，实验证明功耗下降 20X、吞吐提升 50X；ISAAC、AEVis、PipeLayer 等方案利用 RRAM/PCM 交叉阵列和 ADC/DAC 完成模拟 MAC，配以数字后处理实现端到端 DNN 推理；RxNN[29] 在 ReRAM CIM 平台上实现了小于 1% 的推理精度损失。与上述基于新型存储材料的模拟加速不同，SRAM-Based CIM 则依托成熟的 SRAM 工艺，将数字或混合信号 MAC 单元嵌入片上高速 SRAM 区，实现低延迟、易集成的算子级加速。

1.3 论文的主要研究工作

本文的主要工作是面向 RISC-V SRAM 存算一体芯片修改 LLVM 编译器，实现对存算指令的支持。通过深入研究 LLVM 编译器架构和工作原理，分析 RISC-V SRAM 存算一体芯片的特性，探索如何在 LLVM 中添加对 RISC-V CIM 的支持。这包括但不限于对指令集的扩展、内存模型的适配、NPU 指令的智能识别、指令调度以及优化策略的调整等。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描述，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. 基于 RISC-V SRAM 存算一体芯片扩展 LLVM 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. 对实现的编译器进行功能测试和性能测试。功能测试方面，对预训练神经网络模型予以优化，并将其部署于 RISC-V 存算一体模拟器之上，以此验证编译器各个模块的功能；性能测试方面，聚焦于深度学习网络中常见算子，验证 NPU 在实际运行中的加速成效，确保其能在深度学习场景下有效提升运算效率，充分发挥加速优势。

1.4 论文的章节安排

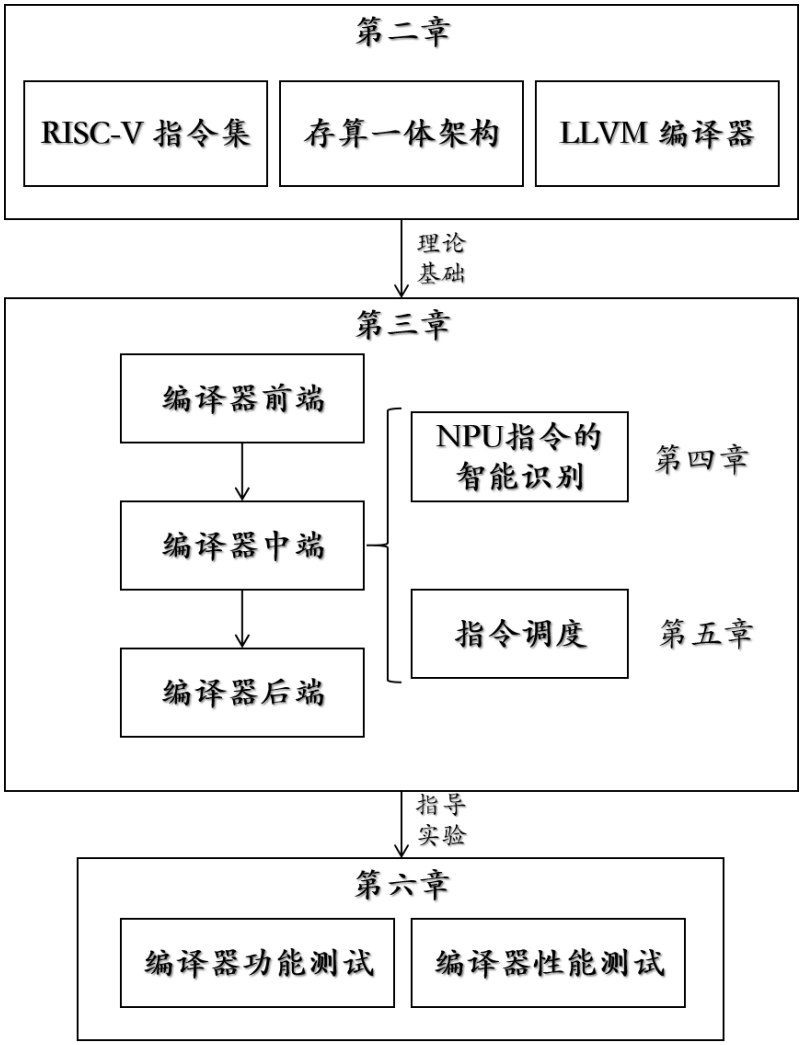

图 1.7 研究内容框架图

全文共包含 7 个章节，主要研究结构安排如图 1.7 所示：

第 1 章为绪论。概述存算一体加速器的发展背景，点明深度学习模型部署及专用深度学习编译器面临的挑战，系统梳理了深度学习编译器与加速器领域的最新进展，最后明确本研究的工作内容与章节布局。

第 2 章聚焦于核心技术基础。本章全面梳理了本研究所需的关键技术要点，依次深入剖析了 RISC-V 基础指令集及扩展指令集；系统阐述了存算一体加速器的技术原理；最后详实介绍了 LLVM 编译器的模块化结构及 LLVM IR 的设计，为后续章节的编译器优化设计提供了坚实的理论支撑。

第 3 章为基于 RISC-V 存算一体加速器的编译器总体描述。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。

第 4 章为 NPU 加速指令的智能识别。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 5 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。深入探讨如何突破异构壁垒，实现指令的精准、快速调度，优化计算资源调配，是提升编译器运行效率的核心驱动力。

第 6 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 NPU 核心的性能表现。

第 7 章为工作总结与展望。对本文的研究工作进行了全面的盘点，系统梳理了研究成果。在此基础上，我们对编译器存在的不足进行了细致的剖析，并深入洞察了后续改进与完善的必要性，为后续提升工作锚定关键着力点。

第 2 章　主要技术基础

2.1　RISC-V

RISC-V[30] 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 采用“Base ISA + Extensions”的模块化设计 —— 基础指令集（如 RV32I、RV64I 等）提供最小的通用计算能力，利用标准扩展（M、A、F、D、C、B、V 等）或自定义扩展（Custom-0/1/2/3）满足特定需求。其中，标准扩展由 RISC-V 国际基金会维护，保障跨平台兼容；而自定义扩展则面向专用应用，设计者可在保留的编码空间内自由定义指令，实现硬件加速功能。如图 2.1 所示。

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
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 2.1　RISC-V 指令集格式

面向深度学习和高性能矩阵计算的 NPU（Neural Processing Unit），通常需要大规模的矩阵乘加（MAC）和张量操作。针对这一需求，RISC-V 提出了 Attached Matrix Extension（AME），在硬件层面集成多路 MAC 单元，并新增 `LDMAT`、`STMAT`、`MADD.M` 等矩阵指令，支持高效的矩阵载入、存储与运算。与基于可编程向量寄存器的 Vector（V）扩展不同，AME 直接通过专用矩阵加速单元（MAU）实现低延迟、高吞吐的张量计算。

在实际 NPU 微架构中，Accelerator 内部集成了多级缓存（Cache/Scratchpad）和指令缓冲队列，以保证矩阵数据的局部性与指令流水线的高效调度。编译器后端

通过张量分块与指令重排序，将高层深度学习算子映射为 AME 指令序列，从而在 RISC-V 核心与矩阵加速硬件之间实现紧耦合的高性能计算平台。

2.2 存算一体架构

存算一体（Computing in Memory，CIM）加速器作为一种新型的计算架构，近年来在深度学习加速领域受到了广泛关注。其最早可以追溯到 20 世纪 70 年代，美国斯坦福研究所（Stanford Institute）的 Kautz 首次提出了存算一体的概念，其核心思想是直接利用内存完成计算功能，减少数据在存储器和处理器之间的搬运开销。这种架构尤其适合深度学习中的矩阵运算和卷积运算等数据密集型任务，因为这些任务通常涉及大量的数据访问和计算操作。通过减少数据传输，存算一体架构能够显著降低系统功耗，并提高系统的整体性能。

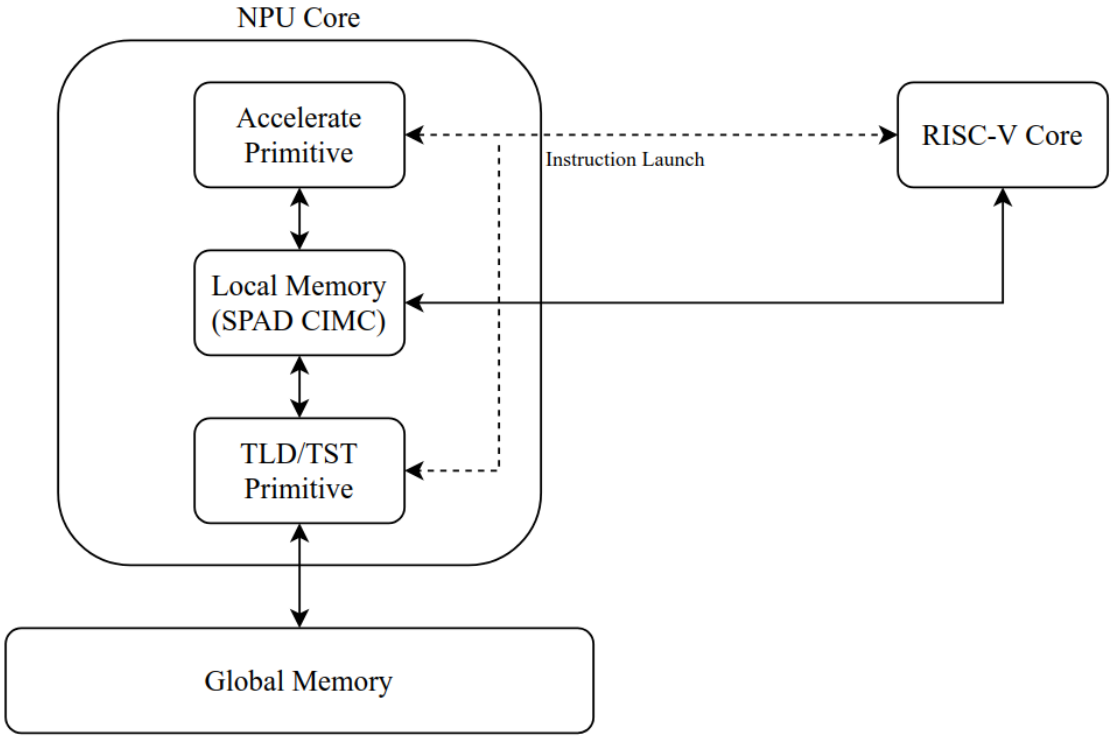

图 2.2　CIM 架构

如图 2.2，在本文使用的 RISC-V 存算一体模拟器中，NPU Core 作为 NPU 的一部分，与外部的 RISC-V Core 共享程序计数器（PC），其二者通过 NICE 接口来进行通信，RISC-V Core 与 NPU Core 通过扩展指令或内存映射的方式互连，共同组成 NPU。本文使用的 CIM 内核由四大模块构成：

（1）存算阵列（SRAM-Compute Array），基于多路 SRAM 单元构建的乘累加阵列（MAC Array），通过行/列选通与模拟加法网络完成向量/矩阵级别的并行乘加运算；

（2）指令译码与调度单元（Decoder & Scheduler），将 RISC-V 代码生成的自定义指令解码为阵列操作序列，负责张量分块、指令流水与读写冲突管理；

（3）本地缓冲区（Local Buffer），多级 SRAM Scratchpad，用于存放中间结果与权重子块，配合数据预取（Prefetch）和写回（Write-back）策略，降低外部存储带宽压力；

（4）AXI 接口模块（AXI Master），支持突发传输（Burst）、非阻塞读写（Out-of-order），保证阵列操作所需数据的连续高效获取。

在存算一体架构中，存储单元不再仅仅是数据的存储介质，同时也承担了部分计算任务。例如，在基于忆阻器的存算一体架构中，忆阻器不仅可以存储数据，还可以通过调整其电阻状态来实现基本的加法和乘法运算。这种架构的设计使得数据的读取和处理可以在同一位置完成，从而减少了数据传输的开销。

相比较于传统的冯诺依曼架构，存算一体架构的优势是：

1. 降低数据传输开销：在传统的冯诺依曼架构中，数据需要在存储单元和计算单元之间频繁传输，这不仅增加了系统的功耗，还限制了系统的性能。存算一体架构通过将计算和存储结合在一起，极大地减少了数据传输的需求，从而降低了功耗并提高了计算效率。
2. 提高能效：由于减少了数据传输，存算一体架构在能效方面具有显著的优势。这使得其在移动设备和嵌入式系统等对功耗敏感的应用场景中具有很大的潜力。
3. 提升计算速度：存算一体架构能够并行处理大量的数据，尤其是在深度学习中常见的矩阵运算中，这种架构可以实现高效的并行计算，从而显著提升计算速度。

存算一体架构在深度学习领域具有广泛的应用前景。深度学习模型通常包含大量的矩阵乘法和卷积操作，这些操作对计算资源和存储资源的需求都非常高。存算一体架构可以通过在存储单元内部直接进行这些计算，显著提高模型的推理和训练速度。

此外，存算一体架构还适用于其他数据密集型应用，如图像处理、信号处理和大数据分析等领域。随着技术的不断发展，存算一体架构有望在更多领域中发挥其优势，推动相关技术的进一步发展。

2.3 LLVM 编译器

LLVM（Low-Level Virtual Machine）[31] 是一个广泛使用的编译器基础设施，旨在为各种编程语言提供一个通用的编译和优化框架。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 的设计目标是提供一个模块化、可扩展的编译器框架，使得开发者可以轻松地实现针对不同硬件平台和编程语言的编译器。

2.3.1 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。

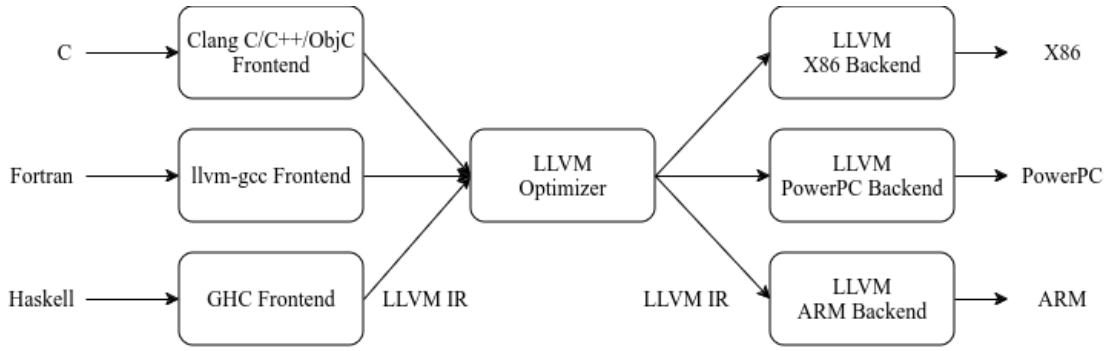

图 2.3 LLVM 编译器的结构

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。

2.3.2 LLVM IR

LLVM IR 是 LLVM 编译器基础设施中的核心组件，是一种低级、类型安全的中间表示语言。其设计旨在为编译器的前端和后端提供一个通用的、易于优化的中间表示形式，使得不同源语言的代码可以经过统一的优化流程并最终生成高效的机器代码。LLVM IR 采用基于寄存器的三地址代码形式，通过有限的指令集来表达程序的各种操作和控制流。这种简洁的指令集设计降低了编译器的复杂度，使得前端到中间表示的转换以及后端从中间表示到机器代码的生成都更为高效。

LLVM IR 具有严格的类型系统，涵盖标量类型、指针类型、结构体类型和数组类型等。这种类型系统不仅提高了代码的安全性和可靠性，还为深度学习编译器处理复杂数据类型（如张量）提供了坚实的基础。此外，LLVM IR 的指令集设计注重可读性和可操作性，支持数据操作、内存操作和控制流指令。这些特性使得 LLVM IR 能够准确地表示深度学习模型中的各种计算逻辑和数据结构，为后续的优化和代码生成提供支持。

2.3.3 LLVM 后端编译

2.3.3.1 模块加载与解析

整个流程始于模块加载阶段，此阶段主要负责将输入的 LLVM IR 模块文件读入内存，并对其进行语法和语义解析，将其转换为 LLVM 内部的数据结构表示形式。这一过程确保了后续编译阶段能够正确地理解和处理输入代码。

2.3.3.2 优化处理

优化是编译过程中的重要环节，旨在提高生成代码的性能和质量。LLVM 后端提供了丰富的优化策略，包括但不限于：

- 常量折叠与传播：在编译时尽可能地计算常量表达式的值，并将其替换掉，减少运行时的计算开销。
- 死代码消除：识别并去除程序中不会产生任何可见副作用的代码，降低目标代码的大小和复杂度。
- 指令组合与简化：合并多个简单的指令为更复杂的指令，或者将复杂的指令分解为更简单的形式，以提高指令的执行效率。

- 循环优化：对循环结构进行分析和变换，如循环展开、循环融合、循环分配等，以减少循环控制开销、提高数据局部性和指令级并行性。

这些优化策略相互配合，从不同层面挖掘代码的潜在性能提升空间，使生成的目标代码更加高效。

2.3.3.3 寄存器分配

寄存器分配是编译器后端的一个关键任务，它决定将程序中的虚拟寄存器映射到目标机器的实际寄存器上。LLVM 采用多种寄存器分配算法，如贪心算法、线性扫描算法等，以在有限的寄存器资源下实现最优的寄存器分配方案，减少寄存器溢出到内存的频率，从而提高程序的执行速度。

2.3.3.4 指令选择与调度

指令选择阶段将 LLVM IR 中的中间代码转换为目标机器指令集中的具体指令。LLVM 通过模式匹配等技术，将高级的 IR 操作映射到合适的机器指令上。同时，指令调度则关注指令的执行顺序，根据目标处理器的架构特性（如指令流水线结构、功能单元特性等），对指令序列进行合理安排，以减少数据相关性导致的等待时间、充分利用处理器的并行执行能力，进一步提升代码的执行性能。

2.3.3.5 代码生成

在完成上述阶段后，进入代码生成阶段。此阶段将经过优化和处理的指令序列转换为目标机器的二进制机器代码。LLVM 根据不同的目标架构（如 x86、ARM、RISC-V 等），生成相应的机器指令编码和相关的辅助信息（如重定位信息、调试信息等），最终输出可执行文件或目标文件，完成整个编译过程。

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 RISC-V 基础指令集和扩展指令集；接着对存算一体加速器的架构进行了基本介绍；最后本章对 LLVM 编译器结构和其后端编译流程进行了介绍。

第 3 章　面向 RISC-V 存算一体加速器的编译器

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 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

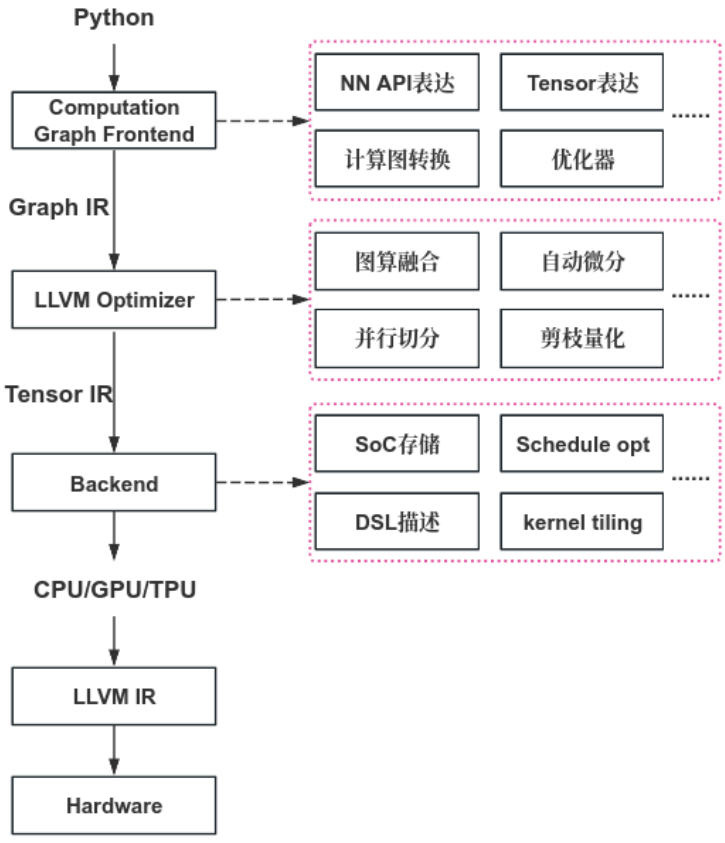

图 3.1　深度学习编译器架构

深度学习编译器的工作是接收上层深度学习模型作为输入，运用各种优化技术，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，从而为深度学习模型的部署与运行提供强大的支持。其大致的框架图如图 3.1 所示。

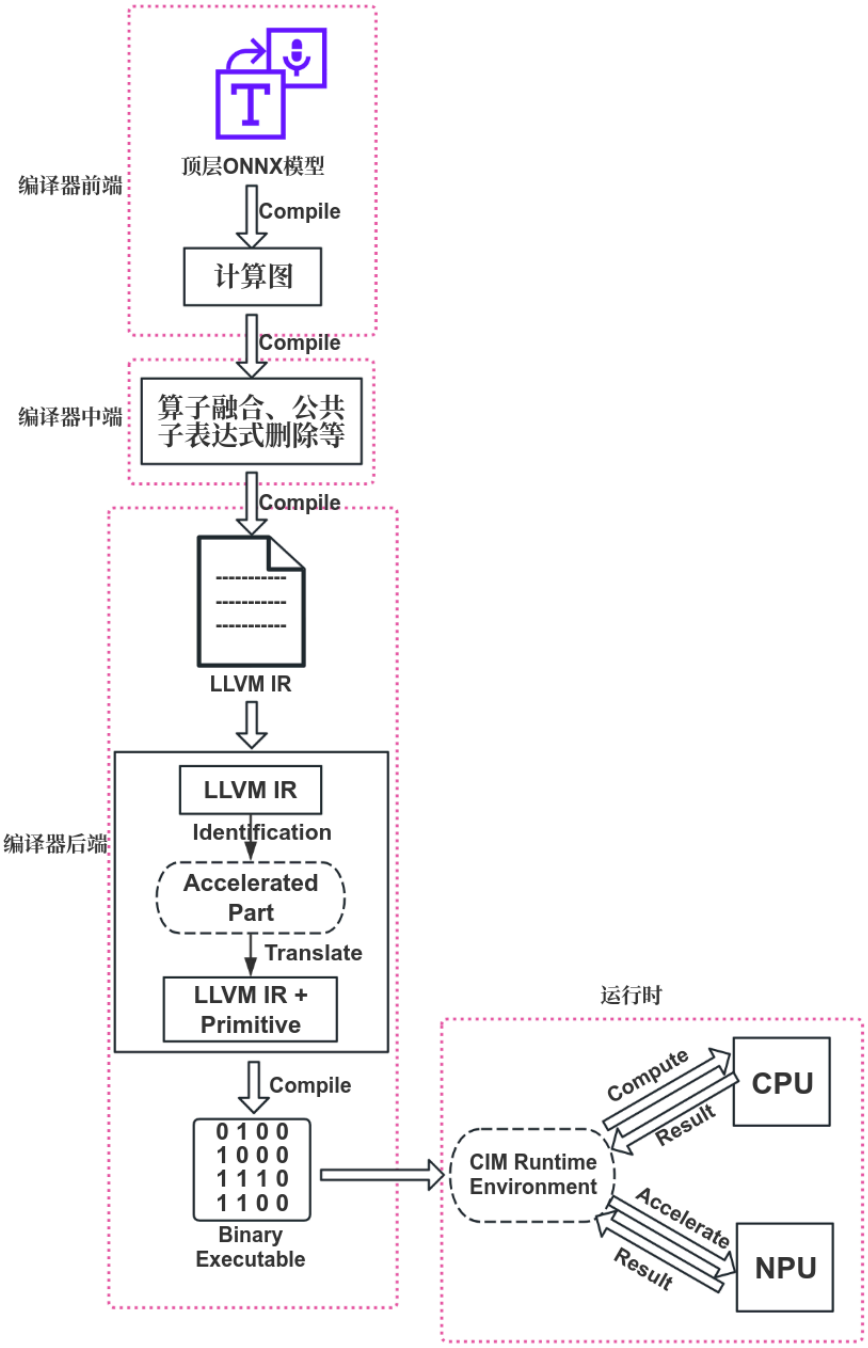

图 3.2　编译器整体架构

本文基于 RISC-V 存算一体模拟器的深度学习编译器数据流具体如图 3.2 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 3.2 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。鉴于目前大多数主流深度学习框架，如 PyTorch[32] 和 TensorFlow[33]，皆支持将模型导出或转换为 ONNX 格式，本文选用 ONNX 作为所设计编译器的输入格式。ONNX 本身的设计初衷便是作为不同框架之间以及框架与工具之间的模型转换中间格式。因此，基于其作为输入格式，本编译器能够直接或间接地兼容多种框架下的模型。

（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生成的计算图，对其进行与体系结构无关的优化。通过实施等价变换，能够有效降低计算图的计算复杂度以及空间复杂度，进而缩短模型推理过程中的运算时间，显著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、指令调度以及目标硬件代码生成。

（3）编译器后端：后端部分与目标体系结构关联紧密，其主要职能包括实施与硬件结构相关的优化举措，以及生成专为特定目标体系结构优化后的代码。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，分析 RISC-V 存算一体模拟器的特性，在 LLVM 中添加了对 RISC-V CIM 的支持。该后端以 LLVM IR 为输入，执行一系列与目标体系结构相关的优化操作，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。

图 3.3 编译器前端工作流程图

3.2.1 ONNX

ONNX[34]（Open Neural Network Exchange，开放神经网络交换），是一种用于表示深度神经网模型的开放格式，由微软和 Facebook 于 2017 年共同推出。作为一个标准化框架，ONNX 确立了一组与运行环境和硬件平台无关的标准格式，旨在实现不同深度学习训练和推理框架间的模型转换与互操作。它不仅定义了一种可扩展的计算图模式、各类运算符，还涵盖了标准化的数据类型，为不同框架提供了一种统一的 IR。如图 3.4 使用 Netron[35] 对 yolov3-tiny[36] 模型进行可视化。

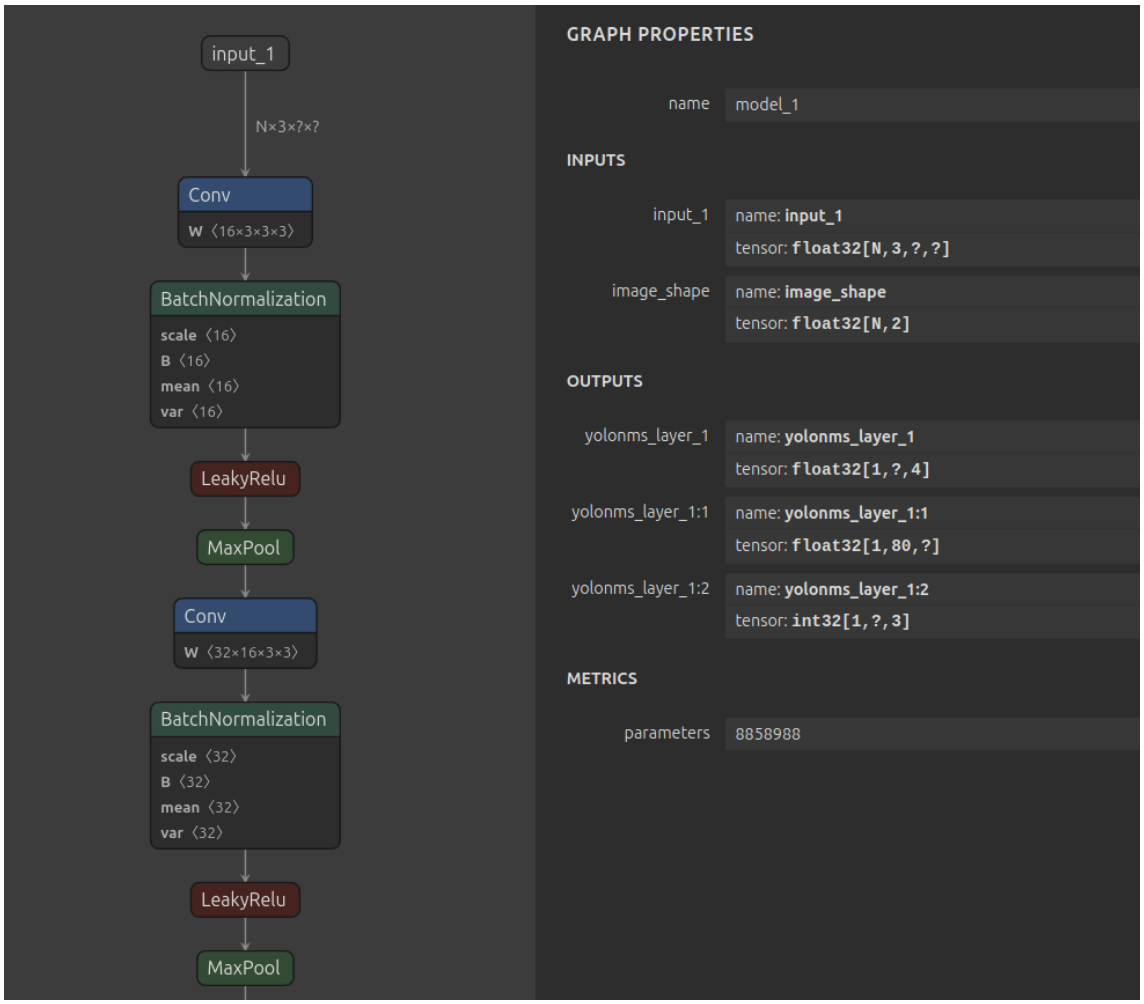

图 3.4　yolov3-tiny ONNX 可视化

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 3.1　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
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 3.1 所示，加载 ONNX 模型后，得到的是一个 `ModelProto` 对象，其中不仅包含了版本信息和生产者信息，还涵盖了至关重要的 `GraphProto` 组件。`GraphProto` 内嵌四个关键的可重复数组：node（`NodeProto` 类型）、input（`ValueInfoProto` 类型）、output（`ValueInfoProto` 类型）以及 initializer（`TensorProto` 类型）。其中，node 存储了模型的所有计算节点；input 记录了模型的所有输入节点；output 保存了模型的所有输出节点；initializer 则存放着模型的全部权重数据。每个计算节点内部同样配置了 input 和 output 两个数组，借助 input 与 output 的指向关系，我们能够依据上述信息高效地构建出深度学习模型的拓扑结构图。此外，每个计算节点还内含一个 `AttributeProto` 数组，用于详细描述该节点的属性特征，例如，卷积层（`Conv`）的属性便包括分组数（`group`）、填充（`pads`）和步幅（`strides`）等关键参数。

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（Intermediate Optimizer）位于编译器前端与后端之间，主要负责对前端生成的 IR 进行体系结构无关的优化。本文编译器接收前端构建的计算图，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 3.5 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。

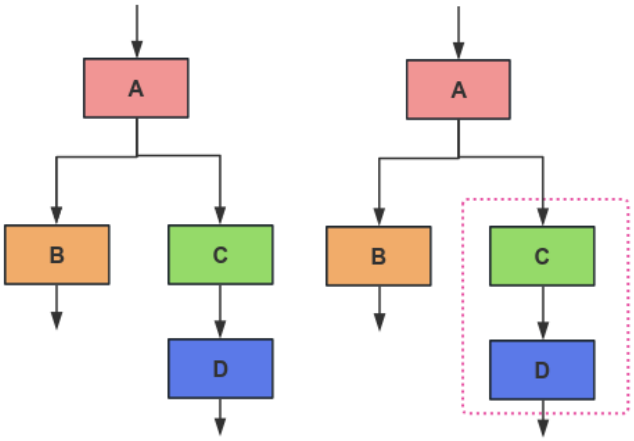

图 3.5　算子融合案例 1

如果是图 3.6 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 3.6 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。

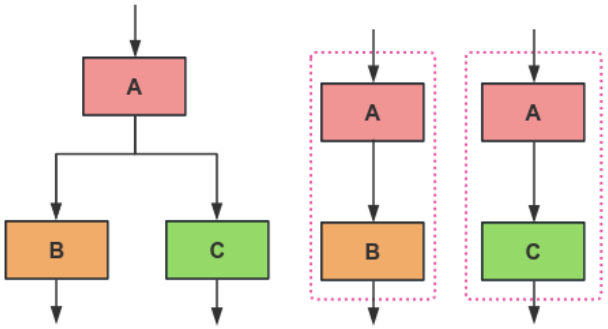

图 3.6　算子融合案例 2

依然还是图 3.6 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。

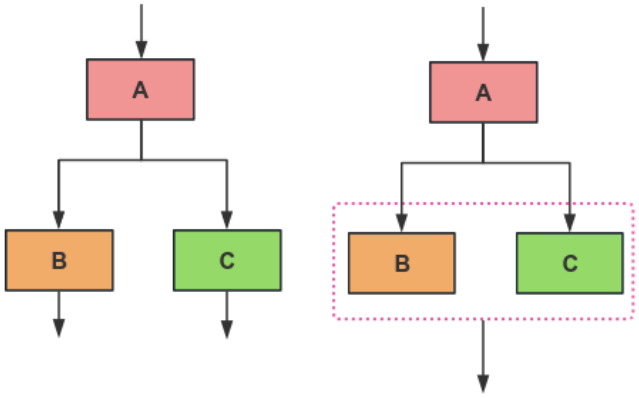

图 3.7　算子融合案例 3

还是图 3.6 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。

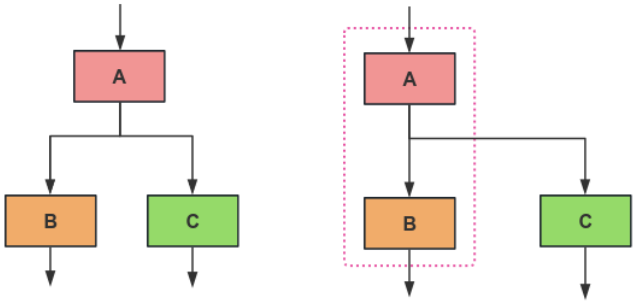

图 3.8　算子融合案例 4

3.3.2　公共子表达式消除

公共子表达式消除（Common Subexpression Elimination，CSE）是一种广泛应用于各种编译器的经典优化技术。其核心目标在于识别并消除程序中重复计算的公共表达式，借此降低计算量，进而提升程序的执行效率。

在实际编程场景中，常常会出现这样的情况：多个不同的代码位置使用了相同的表达式进行计算，而这些表达式的计算结果却完全一致。对这些相同的表达式进行重复计算，无疑会引入额外且不必要的计算开销。公共子表达式消除技术的要点就在于精准识别出这些重复的计算实例，将它们提取出来，确保每个公共子表达式仅计算一次，并将计算结果存储起来，供后续需要相同结果的部位直接调用，从而有效避免重复计算，优化整体计算流程。

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 3.9 中 Op3 和 Op4 都

经过了相同的图结构 $\{\{\mathtt{op}_1, \mathtt{op}_2\}, \mathtt{op}_1 \rightarrow \mathtt{op}_2\}$，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。

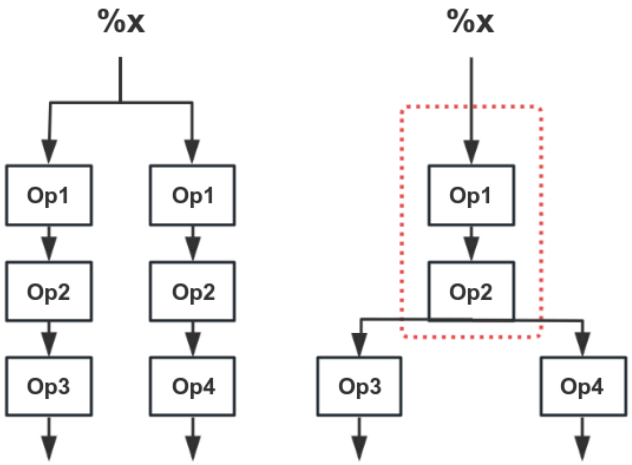

图 3.9　公共子表达式消除案例

3.3.3　死代码删除

死代码删除（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 3.10 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 $B \rightarrow C$。

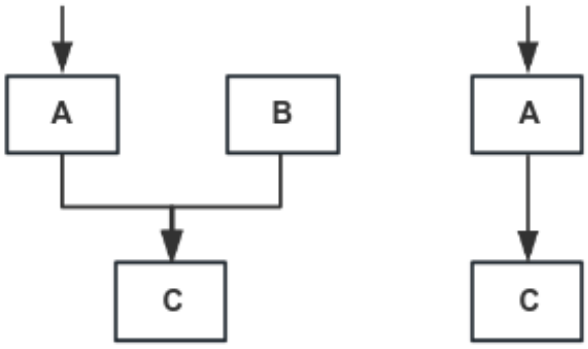

图 3.10　死代码消除案例 1

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 3.11 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。

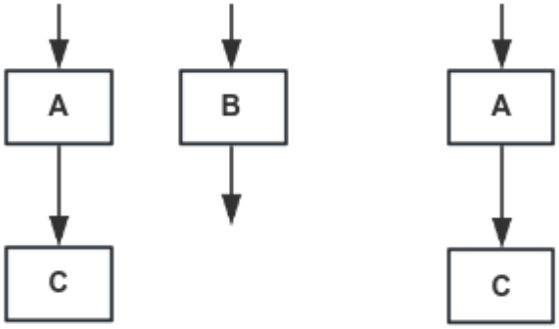

图 3.11　死代码消除案例 2

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

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 3.12 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。

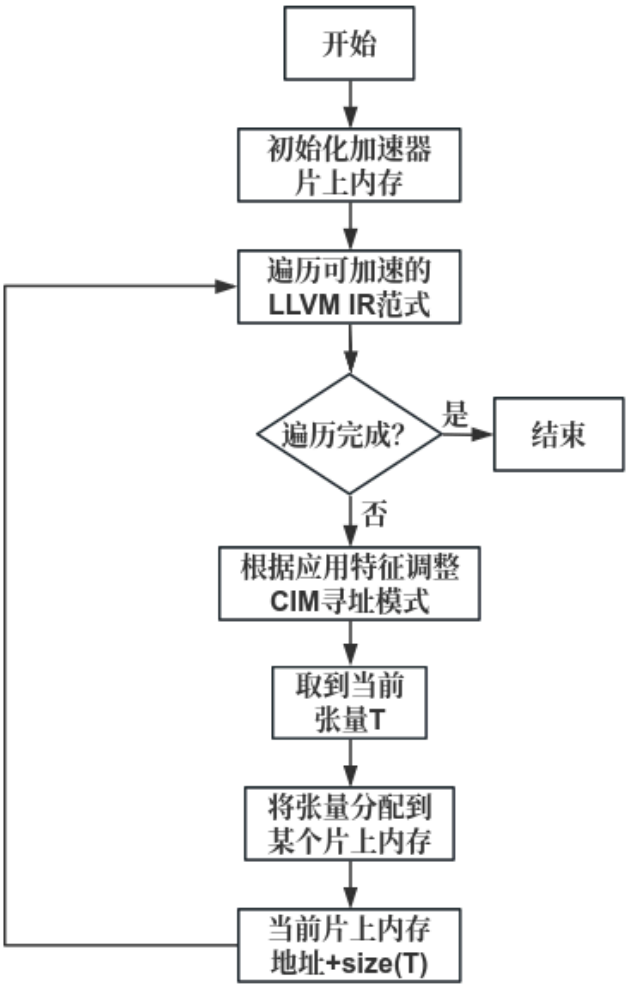

图 3.12　内存分配流程图

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。

3.5 本章小结

本章旨在全面介绍面向 RISC-V 存算一体加速器设计的编译器总体架构。在 3.1 节里，我们提出了本文编译器的整体架构框架，同时深入阐述了编译器的构成要素及其各模块的功能特性，系统地梳理了从编译器前端、中间优化器到后端的整个工作流程。后续章节将重点关注如何智能识别 NPU 加速指令以及实现指令调度的策略与方法。

第 4 章　NPU 指令的智能识别

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、矩阵扩展指令集以及自定义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。

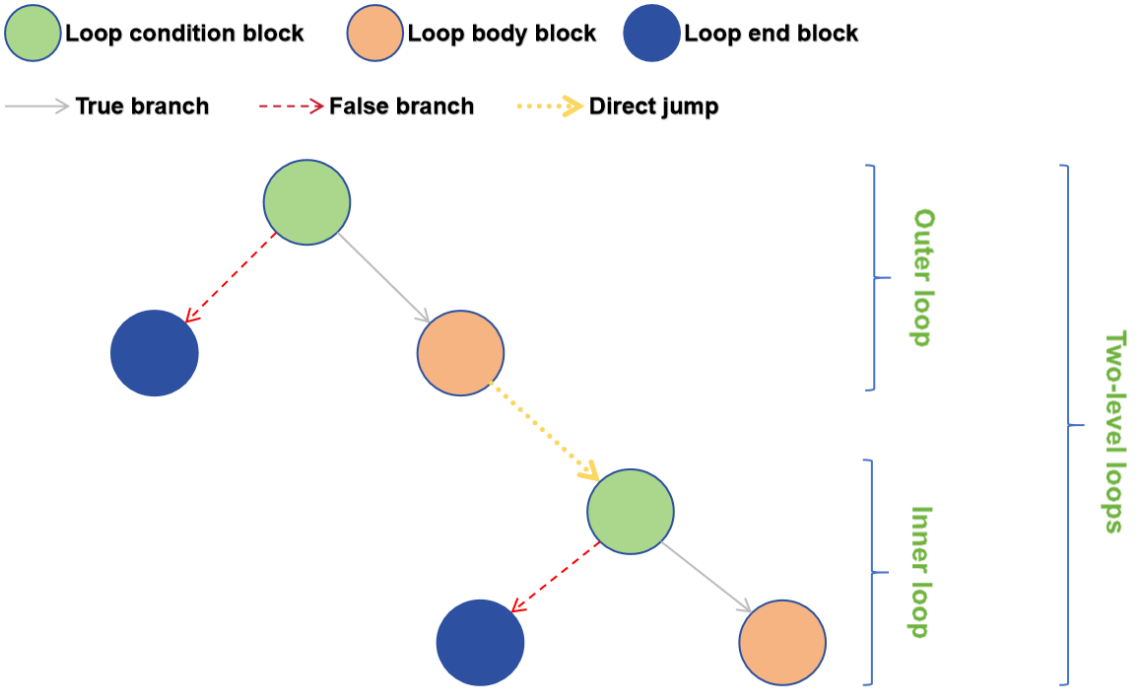

图 4.1　MVM 循环结构分析

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 4.1 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 - 向量操作范式

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。

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
```

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。

4.3 矩阵 - 向量操作范式

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 4.1 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 - 矩阵操作范式

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 4.2 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。

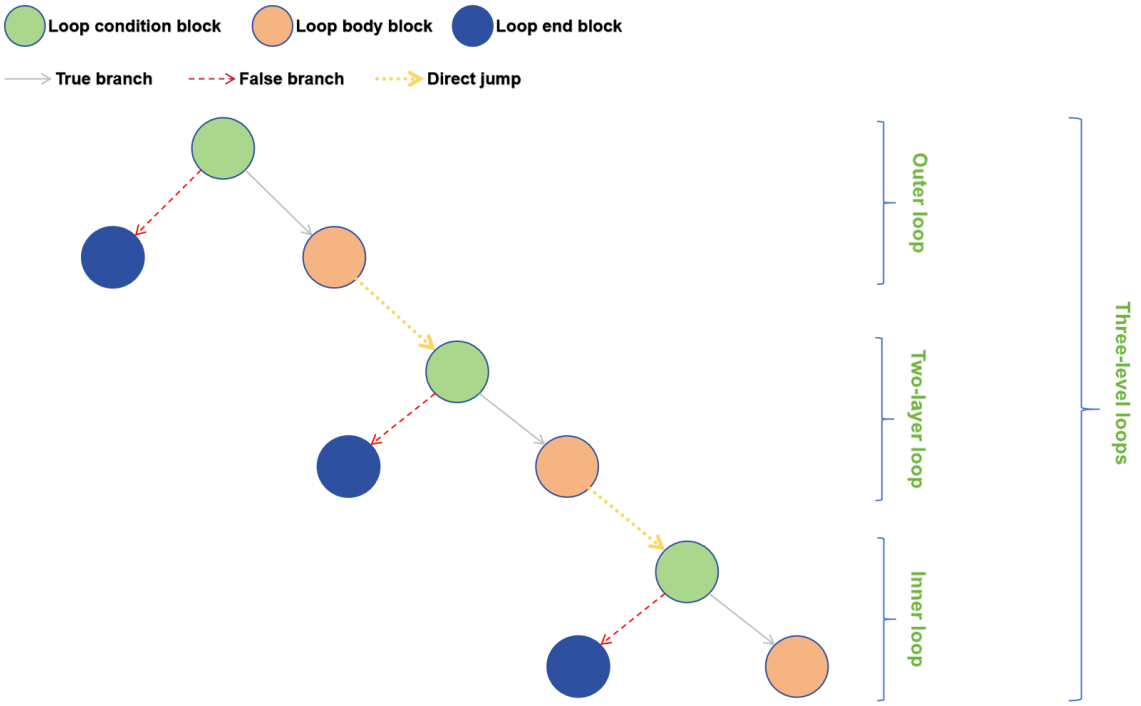

图 4.2　MMM 循环结构分析

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

在程序设计与编译优化领域，指令调度是一种关键的优化技术，它通过对程序块或过程中的操作进行重新排序，从而有效利用处理器资源，提高指令的执行效率。其核心目标在于通过重排指令来增强指令级并行性，确保程序能够在具备指令流水线结构的 CPU 上实现更高效的运行。然而，指令调度优化能否发挥作用的一个重要前提是 CPU 硬件必须支持指令并行，否则，指令调度将失去意义。

根据指令调度发生的阶段，可以将其分为静态调度和动态调度两大类：

（1）静态调度：此过程发生在程序的编译阶段，由编译器完成，在生成可执行文件之前，编译器通过一系列指令调度优化手段，对指令序列进行重新排列。以提升程序的执行效率。

（2）动态调度：与静态调度不同，动态调度发生在程序运行时期间。这一过程需要相应的硬件支持，例如乱序执行（OoOE: Out-of-Order Execution）。在这种情况下，指令的发射顺序和执行顺序可能并不一致，但 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，它们都是通过指令重排来提高指令流水线的效率，进而提高程序的整体执行性能。静态调度和动态调度相辅相成，能够互相补充，完成优化指令流水线，从而有效提高程序的执行效率。

5.2　指令调度问题与约束

在指令调度过程中，调度策略受到多种约束因素的影响，包括数据依赖约束、功能部件约束以及寄存器约束等，在这些约束条件下，指令调度的最终目标是寻找最优的指令排列序列，以此来降低指令流水线中的停顿（stall），从而提高整个程序的执行效率。

指令流水线中的停顿主要由以下三种类型的冒险引起：

（1）数据型冒险：当某条指令的执行依赖于前一条指令的执行结果时，就会产生数据型冒险。数据型冒险可以细分为三种情况：写后读（RAW）、读后写（WAR）以及写后写（WAW）。这些数据冒险可能会引发数据流依赖问题，从而影响指令的执行顺序。

（2）结构型冒险：当多条指令同时试图访问同一个硬件资源单元时，由于硬件资源单元不足，就会引发结构型冒险。这通常是由于硬件资源有限，无法同时满足多个指令的访问需求。

（3）控制型冒险：在程序中存在分支跳转的情况下，由于无法准确预测下一条要执行的指令，从而导致控制型冒险。这种冒险使得指令流水线中的下一条指令不确定，增加了调度的复杂性。

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。

为了应对上述冒险问题，编译器通常采用插入 NOP 指令的方法来增加流水线中的停顿，以此来化解潜在的冒险。通过这种方式，编译器能够在保证程序正确性的前提下，优化指令的执行顺序，提高指令流水线的效率。

下面简要介绍三种数据型冒险（即数据依赖）：

（1）写后读（RAW）：当一条指令读取前一条指令所写入的数据结果时，就形成了写后读依赖。这是最为常见的数据依赖类型，被称为真数据依赖（true dependence）。

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

（2）读后写（WAR）：当一条指令向数据存储位置写入数据时，若该位置的数据曾被前条指令读取，则形成读后写依赖。这种依赖关系被称为反依赖或反相关（anti dependence）。

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

（3）写后写（WAW）：当两条指令依次向同一目标位置写入数据时，就会产生了写后写依赖。这种情况被称为输出依赖（output dependence）。

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

5.3 静态指令调度

静态指令调度是编译器优化过程中的关键技术之一，它在编译期间对指令序列进行重新排列组合，以优化程序的执行效率。静态指令调度通过分析指令之间的相关性和依赖关系，合理安排指令的执行顺序，目的是尽可能地减少数据相关性导致的处理单元等待时间、充分发挥处理器的指令级并行性，从而提高硬件资源的利用率并减少程序的总体执行时间。常用的静态指令调度算法包括基于依赖图的调度算法和基于启发式规则的调度算法等。

本文基于表调度（List Scheduling）设计了一种的静态指令调度策略，以在异构处理系统中提升 CPU 与 NPU 之间的并行执行效率，目标是在编译阶段提前发射可独立执行的 NPU 指令，以减少 CPU 的空转等待时间。该策略在 LLVM 后端的指令调度阶段引入，核心思想是：尽可能将不依赖 CPU 结果的 NPU 指令提前调度至程序片段的前部，从而使得 NPU 能够尽早启动执行，减少 CPU 在等待 NPU 执行完成过程中的空转时间。为确保调度的正确性与通用性，我们基于经典的表调度算法作为基础框架，并在其上引入了 NPU 指令优先的调度规则。

表调度算法是一种融合了贪心策略与启发式规则的调度方法，主要用于对基本块内的各类指令操作进行合理调度，凭借其实用性及高效性，已成为基本块内指令调度领域的常见且关键方法之一。基于基本块的指令调度方式，其优势在于无需过多考量程序的控制流，而是将重点聚焦于数据依赖关系、目标体系结构的指令延迟特性、硬件资源的分配情况以及流水线的工作状态等多方面信息，从而实现对指令的精准调度。

表调度算法的核心思想主要体现在以下几个方面：构建并维护一个用于存放已具备执行条件指令的 `ready` 列表，以及一个用于跟踪正在执行指令的 `active` 列表。其中，`ready` 列表的生成主要依据数据依赖约束条件以及硬件资源的相关信息。在调度过程中，以周期为单位，按照既定的调度算法执行一系列操作，包括从列表中甄选合适指令进行调度，以及及时更新列表中的指令状态信息，以确保整个调度流程的连贯性与准确性。

具体而言，表调度算法的操作流程大致可分为以下三个关键步骤：

（1）依据指令间的依赖关系，构建一张详尽的依赖关系图，为后续的调度决策提供基础依据。

（2）结合当前指令节点到根节点的路径长度以及各指令自身的 latency（延迟）特性，对每个指令的优先级进行综合计算，以此来确定指令的执行顺序。

（3）持续从 ready 列表中选取指令并对其进行调度。在此过程中，利用两个队列分别对 ready 状态的指令和 active 状态的指令进行维护管理。在每个周期内，优先选择满足执行条件的 ready 指令予以调度，并同步更新 ready 队列；同时，检查 active 列表中的指令是否已完成执行，据此更新 active 列表，确保整个调度流程的动态性与实时性。

值得关注的是，在实际的指令调度执行过程中，当遇到具有相同优先级的多条指令时，由于不同的调度方案选择，可能会导致不同的调度结果。为应对这种情况，可以通过引入额外的度量标准，进一步优化优先级的计算方案，从而提高调度结果的合理性与有效性。尽管表调度方法在理论上无法绝对保证得到最优的调度结果，但凭借其独特的调度策略与启发式规则，往往能够获取到接近最优解的调度方案，展现出较高的实用价值与应用潜力。

基于上述经典的表调度算法作为基础框架，本文在其上引入了 NPU 指令优先的调度规则。该调度算法主要分为两部分：

1. 调度候选生成：
   - 构建 DAG（有向无环图）表示基本块中所有机器指令；
   - 计算每条指令的依赖关系（数据依赖和调度顺序约束）；
   - 对于所有入度为 0 的指令（无前驱），将其加入就绪队列（ready list）。
2. 调度选择逻辑（核心）：
   - 每次从 ready list 中选择一条指令调度；
   - 若存在无依赖的 NPU 指令，优先调度该 NPU 指令；
   - 否则，采用标准的表调度优先级策略进行调度（如 critical path、latency、资源利用等）。

本调度算法以传统的表调度（List Scheduling）为基础，引入了如下关键优化：

1. NPU 优先策略：在每个调度周期中，优先尝试选择无数据依赖的 NPU 指令进行调度，确保 NPU 核心尽早获得任务并启动；
2. 异构并发优化：通过将 NPU 指令提前，CPU 能够更早进入其独立执行路径，从而最大化利用两种处理器资源；
3. 可扩展性强：该策略保持了表调度的模块化结构，可与现有 LLVM 调度器兼容。

本调度算法伪代码如下：

```
SheduleInstructions(DAG):
 1  readyList = []                              // 当前准备好可以调度的指令
 2  sheduledList = []                           // 已调度的指令列表
 3  pendingNPU = []                             // 可提前调度的 NPU 指令候选
 4
 5  for SU in DAG.SUnits:                       // 初始化 readyList
 6      if SU.NumPredsLeft == 0:
 7          readyList.append(SU)
 8
 9  while readyList!=[] || pendingNPU !=[]:
10      nextSU = null
11      for SU in readyList:                    // 优先调度无依赖的 NPU 指令
12          if isNPUInst(SU.Instr):
13              if !hasCPUDependency(SU):
14                  nextSU = SU
15                  break
16
17      if nextSU == null:
18          nextSU = selectBest(readyList)
19
20      emitInstruction(nextSU.Instr)
21      sheduledList.append(nextSU)
22      readyList.remove(nextSU)
23
24      for succ in nextSU.Successors:          // 更新依赖关系
25          succ.NumPredsLeft -= 1
26          if succ.NumPredsLeft == 0:
27              readyList.append(succ)
```

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 可进一步被编译为适用于 RISC-V 架构的二进制可执行文件，并在基于存算一体结构的模拟器上执行。在程序初始阶段，所有指令和数据统

一被存储于主存储器中。当程序启动并开始运行时，CPU 便负责指令的获取、调度和执行管理，特别是在异构计算环境下，承担调度 CPU 与 NPU 指令的关键角色。

在指令执行过程中，系统会识别出特定的 CIM 加速指令，并将其从常规的 CPU 指令流中分离出来。这些指令被卸载至专用的 NPU 上执行，以实现更高的计算并行度和能效比。而其余常规的控制流、逻辑判断或轻量级算术操作等普通指令，仍由 CPU 按照顺序执行。

在 CIM 加速指令于 NPU 上执行完毕之后，其结果通过总线返回至 CPU。CPU 负责将这些结果同步写回主存储器，以供后续程序阶段访问与使用。该机制确保了计算结果的一致性和数据访问的可预测性。

为了确保 CPU 与 NPU 之间的高效协作与数据一致性，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。这种设计可有效避免 CPU 和 NPU 之间的数据竞争问题，保证程序语义正确性。

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 5.1 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。

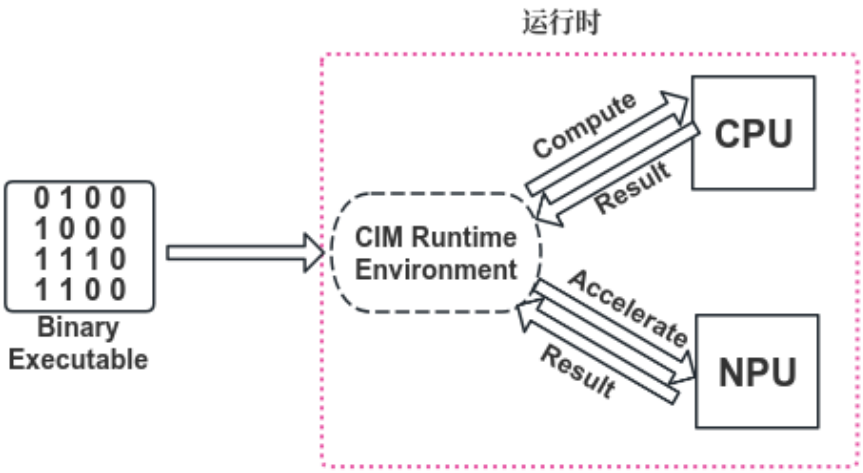

图 5.1　指令调度

5.5 本章小结

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架构的优势。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，来呈现最终的测试结果。

6.1　编译器功能测试

FASHION-MNIST[37] 是由 Zalando 研究团队提供的图像数据集，用于替代传统的 MNIST[38] 手写数字集。该数据集包含了来自 10 个不同类别的共 70000 张商品正面图片，涵盖了各种类型的服装和配饰。FASHION-MNIST 在数据规模、格式规范以及训练集与测试集的划分比例上，均与原始的 MNIST 数据集保持高度一致，其中训练集包含 60000 张图像，测试集包含 10000 张图像，所有图片均为 28 × 28 像素的灰度图像。

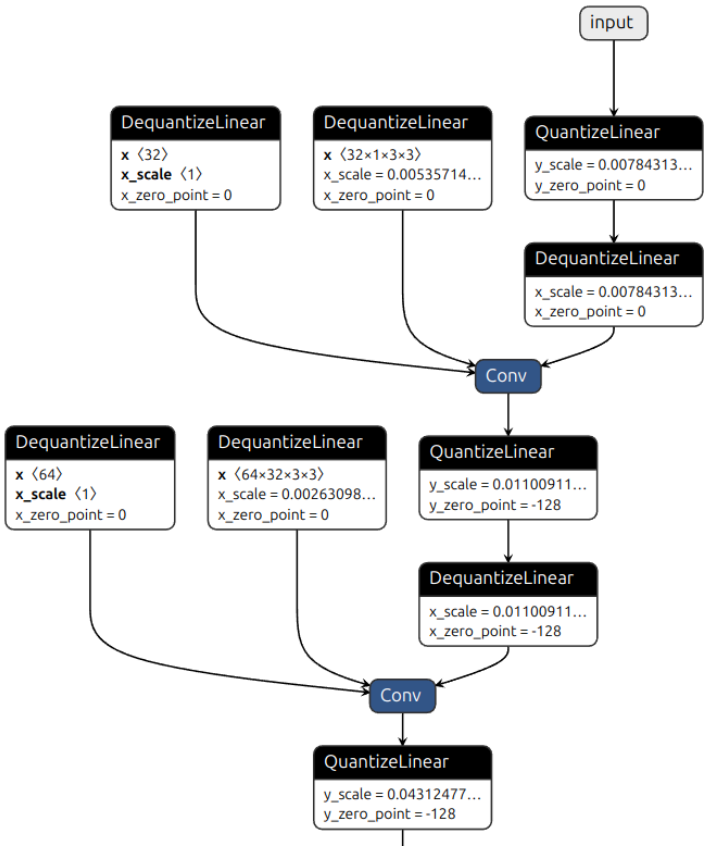

图 6.1　网络架构图

在本次实验中，我们采用了一个自定义设计的 FASHION-MNIST 网络模型进行研究，该模型专门针对 FASHION-MNIST 数据集进行了训练优化。网络模型的具体结构如表 6.1 所示，其对应的 ONNX 可视化结果则展示在图 6.1 中。该网络架构由 3 个卷积层和 2 个全连接层组成，在除最后一个全连接层之外的每个层的输出端，均采用了 ReLU 激活函数来进行非线性激活处理，从而增强模型对复杂特征的学习能力。

```
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 6.1　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: 61%
  Off-chip Transfer ratio: 41%
  Tensor Manipulate ratio: 2%
  Matrix Processing ratio: 9%
  Vector Processing ratio: 11%
CIM Analysis
  CIM Compute ratio: 3.7%
  CIM Space Utilization: 69.9%
  CIM Utilization: 2.6%
  Effective Performance: 26.739GOPS @INint8-Wint8
```

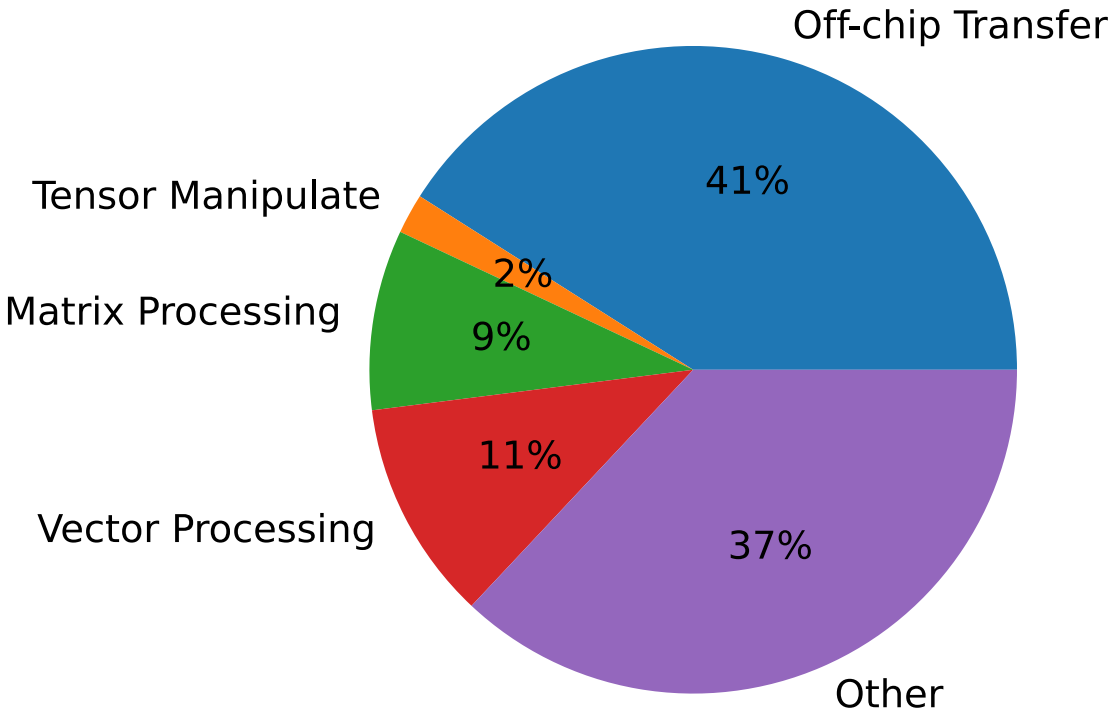

图 6.2　NPU 工作占比

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 61%，矩阵处理占比 9%，向量处理占比 11%，二者合计占比 20%，Tensor Manipulate 占比 2%，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 41%，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 3.70%，利用率仅为 2.60%，这主要是因为片上片外数据传输（Off-Chip Transfer）比例占据了 41%，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 69.90%，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 26.74 GOPS @INint8 - Wint8，在 INT8 量化下，NPU 实现了 26.74 GOPS 有效算力。

```
* * * * Power Analysis * * * *
- - - - NPU Level - - - -
  Spad R/W energy cost: 3491064.32pJ, ratio:44%
  PE vector energy cost: 1.11pJ, ratio:0%
  CIM R/W energy cost: 1295134.72pJ, ratio:16%
  CIM compute energy cost: 3120578.56pJ, ratio:39%
  NPU energy cost: 7906778.71pJ
  CIM Compute Energy Efficiency: 12.5TOPS/W
  NPU Energy Efficiency: 4.93TOPS/W @INint8-Wint8
- - - - System Level - - - -
  NPU ENERGY
    NPU energy cost: 7906778.71pJ, ratio:9%
  CPU ENERGY
    CPU energy cost: 0pJ, ratio:0%
  Off-chip ENERGY
    pSRAM energy cost: 75476480pJ, ratio:91%
  Energy Efficiency
    Total energy cost: 83383258.71pJ
    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 12.50 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 83383258.71 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。

```
* * * * Hardware Traces * * * *
Traces of Control:
  Syncs=0;SW_CIMCs=4;WR_LMEMs=0;RD_LMEMs=0;Working Time=4
Traces of Tensor Load:
  Configs=840;Loads=120;Working Time=295670
Traces of Tensor Store:
  Configs=0;Stores=0;Working Time=0
Traces of Tensor Manipulate:
  Configs=1211;Broadcasts=0;Moves=1524;Transposes=0
  Working Time=13199
Traces of Matrix Processing:
  Configs=1331;Prepares=121;Convs=121;Dwconvs=0
  Working Time=66516
Traces of Vector Processing:
  Configs=4334;Prepares=3493;VVVs=725;VSVs=4115;VSs=2;VVs=4061
  Working Time=79577
Traces of Spad0:
  Total Reads=7670;Total Writes=7212
Traces of Spad1:
  Total Reads=19620;Total Writes=21488
Traces of Spad2:
  Total Reads=12552;Total Writes=8872
Traces of Spad3:
  Total Reads= 7696;Total Writes=6148
Traces of CIM Cluster4:
  Total Reads=0;Total Writes=28745
  Total Page Reads=121;Total Computes=1248231424
Traces of PE Tensor:
  Total Computes=0
Traces of PE Vector:
  Total Computes=44148
Traces of MainMem:
  Total Reads=29483;Total Writes=0
```

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 存算一体模拟器中来评估 NPU 核心的性能表现，性能测试结果如表 6.3 所示，深度学习常见的算子的分类如表 6.2 所示。

表 6.2 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

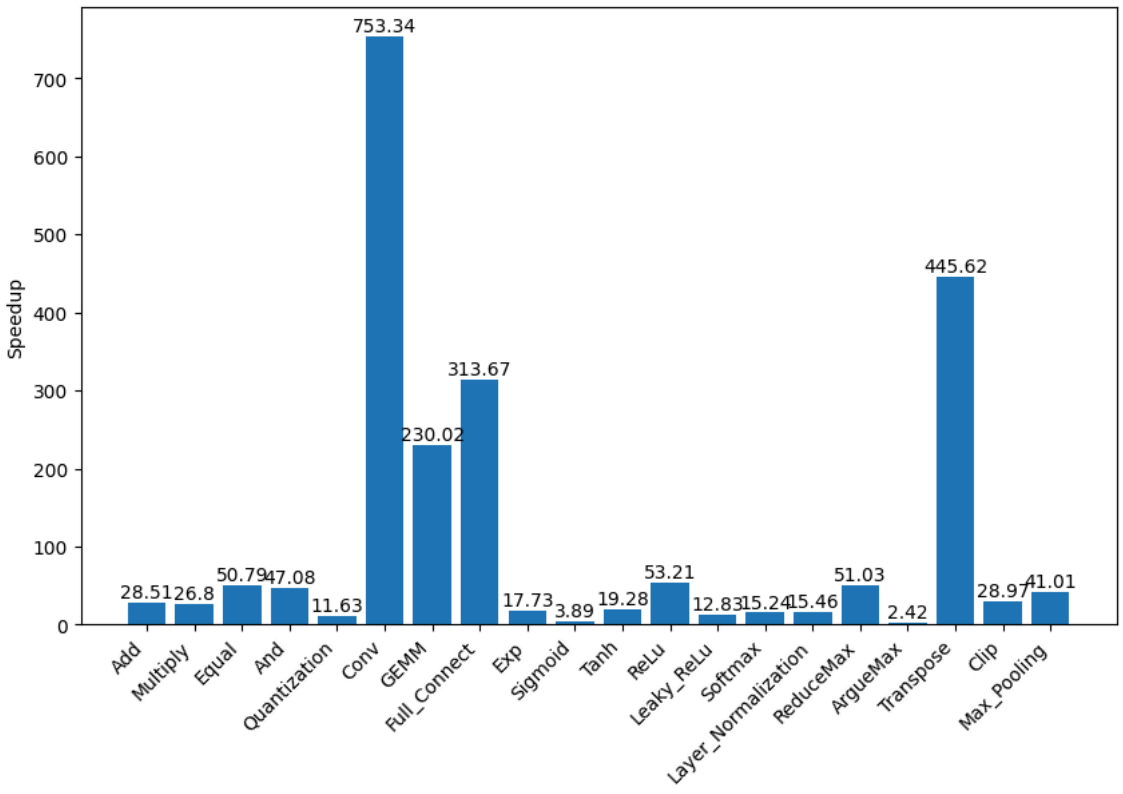

图 6.3 算子测试结果

表 6.3　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

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 NPU 的矩阵乘加（MAC）单元擅长做大规模、规则的乘累累加运算，能够在一条指令中并行完成数十甚至上百次乘加。而 ArgMax 只是在一维或多维张量上做最大值比较和记录下标，它主要是比较和条件写回操作，乘累运算极少，无法驱动大规模 MAC 单元并行工作，导致算力资源严重闲置，因而加速比偏低。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。因此本文面向 RISC-V SRAM 存算一体芯片修改 LLVM 编译器，实现对存算指令的支持，经过总结，本文完成的主要工作如下：

（1）面向 RISC-V SRAM 存算一体芯片修改 LLVM 编译器，实现了对存算指令的支持。通过深入研究 LLVM 编译器架构和工作原理，分析 RISC-V SRAM 存算一体芯片的特性，探索如何在 LLVM 中添加对 RISC-V CIM 的支持。这包括但不限于对指令集的扩展、内存模型的适配、NPU 指令的智能识别、指令调度以及优化策略的调整等，以实现应用算子的自动映射和正确指令流的生成，从而更好地协调计算部件，挖掘芯片内部的计算并行性，为 RISC-V SRAM 存算一体芯片提供有力的编译支持。

（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）指令调度。在 CPU 和 NPU 异构计算单元之间进行指令的静态和动态调度。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 芯片是当今备受瞩目的研究前沿领域，深度学习编译器作为深度学习算法及其实际部署之间的关键桥梁，也逐渐成为了学术界与工业界的研究热点。本文面向 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. 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。

除了具体的技术完善方向外，本文的研究也在更广泛的层面上为我国 RISC-V 芯片产业的发展提供了积极价值：

- 推动国产 RISC-V 架构芯片软件生态建设。当前我国在芯片领域正大力推进自主可控的发展战略，而 RISC-V 作为开源指令集架构，正成为推动我国芯片产业发展的重要突破口。本文基于 RISC-V 架构进行编译器定制优化，特别面向新型存算一体芯片的特性进行编译支持开发，有助于完善 RISC-V 软硬件协同设计生态，降低国产芯片落地部署的门槛，推动国产 AI 芯片在终端、边缘等应用场景中的广泛应用。
- 构建具有自主知识产权的 AI 编译器核心技术。面向 RISC-V 存算一体芯片的编译器关键技术研究，不仅提升了芯片的使用效率，也为我国积累了面向新型 AI 架构的自主编译器研发能力。在未来芯片架构不断创新的背景下，此类研究将为我国在智能计算编译技术方面建立竞争优势，助力打造自主、安全、可控的软硬件一体化系统。

致　谢

落笔于此，湘江畔的梧桐已悄然历经四度春秋。四年的时光倏然而过，仿佛昨日初入校园的悸动仍在心头，而今日，已执笔写下人生一个重要阶段的句点。

盛行千里，不忘师恩。首先，我要特别感谢我的导师曾坤老师和王晨曦老师。从初入大学时的懵懂无知，到如今能够独立完成毕业设计，曾坤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，曾坤老师和王晨曦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，都让我深刻体会到曾坤老师和王晨曦老师在科研工作中的严谨态度和对学生的严格要求。

我还要感谢我的大学：国防科技大学，这是一座将星辉与书卷同辉、理想与责任同行的学府。这里有晨光熹微下的军姿列列，也有灯火阑珊处的潜心钻研；有挥洒汗水的操场，有洒满月光的实验室。我始终铭记，那些与室友们共度的深夜，那些一同并肩攻坚的清晨，都已悄然镌刻在我的青春年轮之上。

感谢在我大学生活中同行的兄弟与朋友。与侯华玮、杨俯众一起走过的四年时光，我们携手完成了许多充满意义的项目，欢笑与不甘、失落与突破，都化作记忆中一颗颗熠熠生辉的星辰。感谢陈爽、陈诚、梁积新、朱夏辉，是你们在我彷徨时伸出温暖的手，是你们在我疲惫时说出笃定的话语，你们是我岁月中最真实的慰藉与同行。尤其感谢文智华师哥，是他牵我走进科研的世界，在浩瀚的知识之海中点亮第一盏灯，带我看见探索的美与深潜的力量。

最后的最后，我要感谢我的父母，生育之恩，终身难忘。感谢他们在我十多年的学习生涯中的支持与鼓励。无数个夜晚，当我因失败而踟蹰，是你们温柔的话语穿越千里，抚慰我焦灼不安的心。你们从不言苦，却为我默默担起所有风雨；你们从不索求，却用全部的爱托举我向前。在未来的科研道路上，我愿以银河为志、以家国为念，秉承“银河精神”，行走于星辰大海，不负韶华、不负期许。

我将继续寻找，就算这无尽的星辰令我的寻找希望渺茫，就算我将单枪匹马。愿未来的我们，都能带着这份温柔与坚定，走向更加辽阔的远方。

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]. Computational Linguistics, 2000, 26(2).

[3] Huang R, Huang J, Yang D, 等. Make-An-Audio: Text-To-Audio Generation with Prompt-Enhanced Diffusion Models[EB/OL]. (2023). https://arxiv.org/abs/2301.12661.

[4] Zhou C, Meng F, Zhou J, 等. Confidence Based Bidirectional Global Context Aware Training Framework for Neural Machine Translation[C]//Muresan S, Nakov P, Villavicencio A. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Dublin, Ireland: Association for Computational Linguistics, 2022: 2878-2889.

[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]. Physica D: Nonlinear Phenomena, 2020, 404: 132306.

[7] Scarselli F, Gori M, Tsoi A C, 等. The Graph Neural Network Model[J]. IEEE Transactions on Neural Networks, 2009, 20(1): 61-80.

[8] Goodfellow I J, Pouget-Abadie J, Mirza M, 等. Generative Adversarial Networks[EB/OL]. (2014). https://arxiv.org/abs/1406.2661.

[9] 7 nm lithography process [Z]. https://en.wikichip.org/wiki/7_nm_lithography_process.

[10] 郭昕婕, 王光燿, 王绍迪. 存内计算芯片研究进展及应用[J]. 电子与信息学报, 2023(5): 1888-1898.

[11] Comprehensive version of the RISC-V Market Analysis Report[Z]. https://www.newswire.com/news/the-shd-group-has-released-a-complimentary-version-of-the-2024-risc-v-22213417.

[12] Sabne A. XLA : Compiling Machine Learning for Peak Performance[Z]. 2020.

[13] Chen T, Moreau T, Jiang Z, 等. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, 2018: 578-594.

[14] Roesch J, Lyubomirsky S, Weber L, 等. Relay: a new IR for machine learning frameworks[C]//Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. Philadelphia, PA, USA: Association for Computing Machinery, 2018: 58-68.

[15] Feng S, Hou B, Jin H, 等. TensorIR: An Abstraction for Automatic Tensorized Program Optimization[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. Vancouver, BC, Canada: Association for Computing Machinery, 2023: 804-817.

[16] 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 Implementation. 2021.

[17] Tillet P, Kung H T, Cox D. Triton: an intermediate language and compiler for tiled neural network computations[C]//Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. Phoenix, AZ, USA: Association for Computing Machinery, 2019: 10-19.

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

[19] Chen Y, Chen T, Xu Z, 等. DianNao family: energy-efficient hardware accelerators for machine learning[J]. Commun. ACM, 2016, 59(11): 105-112.

[20] Jouppi N P, Young C, Patil N, 等. In-Datacenter Performance Analysis of a Tensor Processing Unit[C]//Proceedings of the 44th Annual International Symposium on Computer Architecture. Toronto, ON, Canada: Association for Computing Machinery, 2017: 1-12.

[21] Chen Y H, Krishna T, Emer J S, 等. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks[J]. IEEE Journal of Solid-State Circuits, 2017, 52(1): 127-138.

[22] 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.

[23] Yin S, Ouyang P, Tang S, 等. A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications[J]. IEEE Journal of Solid-State Circuits, 2018, 53(4): 968-982.

[24] Kim J H, Ro Y, So J, 等. Samsung PIM/PNM for Transfmer Based AI : Energy Efficiency on PIM/PNM Cluster[C]//2023 IEEE Hot Chips 35 Symposium (HCS): 卷 0. 2023: 1-31.

[25] Seshadri V, Lee D, Mullins T, 等. Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology[C]//Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. Cambridge, Massachusetts: Association for Computing Machinery, 2017: 273-287.

[26] Hyun B, Kim T, Lee D, 等. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology[C]//2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA): 卷 0. 2024: 263-279.

[27] Weis C, Wehn N, Igor L, 等. Design space exploration for 3D-stacked DRAMs[C]//2011 Design, Automation & Test in Europe: 卷 0. 2011: 1-6.

[28] Chi P, Li S, Xu C, 等. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory[C]//2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA): 卷 0. 2016: 27-39.

[29] Jain S, Sengupta A, Roy K, 等. RxNN: A Framework for Evaluating Deep Neural Networks on Resistive Crossbars[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021, 40(2): 326-338.

[30] Cui E, Li T, Wei Q. RISC-V Instruction Set Architecture Extensions: A Survey[J]. IEEE Access, 2023, 11: 24696-24711.

[31] 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 Optimization. Palo Alto, California: IEEE Computer Society, 2004: 75.

[32] 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 Systems. Red Hook, NY, USA: Curran Associates Inc., 2019.

[33] 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 Implementation. Savannah, GA, USA: USENIX Association, 2016: 265-283.

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

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

[36] Adarsh P, Rathi P, Kumar M. YOLO v3-Tiny: Object Detection and Recognition using one stage improved model[C]//2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS): 卷 0. 2020: 687-694.

[37] Xiao H, Rasul K, Vollgraf R. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms[EB/OL]. (2017). https://arxiv.org/abs/1708.07747.

[38] Deng L. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web][J]. IEEE Signal Processing Magazine, 2012, 29(6): 141-142.