# 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

随着 AI 技术逐渐渗透到各大应用场景，市场对算力的需求呈现爆发式增长。

因此本研究致力于解决上述难题，主要研究了

通过测试表明，本文实现的编译器能够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。

**关键词**：深度学习编译器；LLVM 编译器；调度器；存算一体

# ABSTRACT

Abstract.

**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

深度学习是机器学习算法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]等领域都取到了显著的成果。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出现，简化各种深度学习模型的编程对于实现其广泛应用至关重要。

同时，随着人工智能算法复杂度呈指数级跃迁以及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，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，数据在片外存储与运算核心之间的数据频繁迁移带来严重的传输功耗问题。根据英特尔的研究显示，半导体工艺到了 7nm 时代，数据搬运功耗达到 35pJ/bit，占比达 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。这种“功耗墙”现象严重制约了 AI 芯片在边缘计算场景的部署能力。

另一方面，随着半导体工艺逼近物理极限，单纯依靠工艺微缩带来的性能提升已显疲态。ITRS 路线图指出，传统架构下每代工艺节点的性能增益从 28nm 时代的 40% 骤降至 5nm 节点的 15% [2]，摩尔定律的失效迫使学界寻求架构层面的突破性创新。

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。

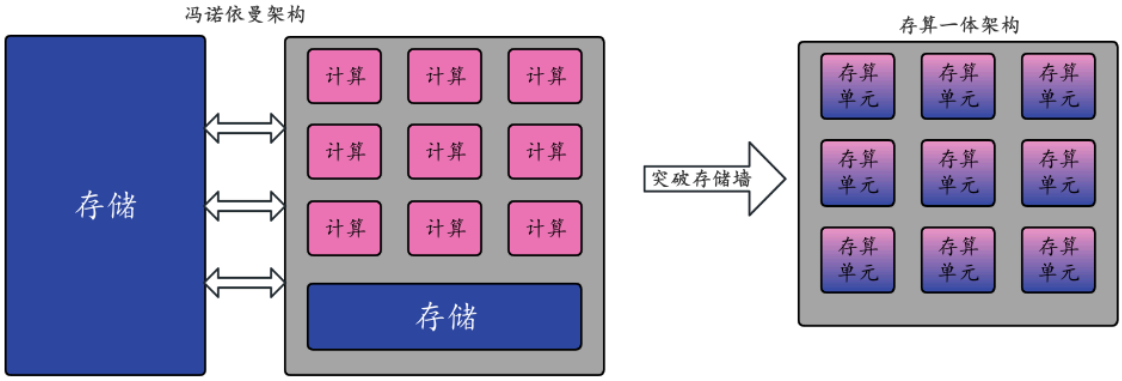


图 1.1　冯诺依曼 v.s. 存算一体架构

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，针对性地对 LLVM 编译器进行深入修改与定制优化，使其能够有效支持存算指令的执行与处理。具体而言，我们将深入钻研 LLVM 编译器的架构体系以及内在工作原理，全面细致地分析 RISC-V 存算一体芯片的特性与需求，积极探索如何在 LLVM 编译框架中成功添加对 RISC-V CIM 架构的全面支持。这一过程涵盖了对指令集的合理扩展、内存模型的精准适配以及优化策略的科学调整等多个关键环节与技术要点。通过这些努力，我们旨在实现应用算子的自动化精准映射以及正确指

令流的高效生成，进而更好地协调各计算部件之间的协作关系，深度挖掘芯片内部所蕴含的计算并行性优势，最终为 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。

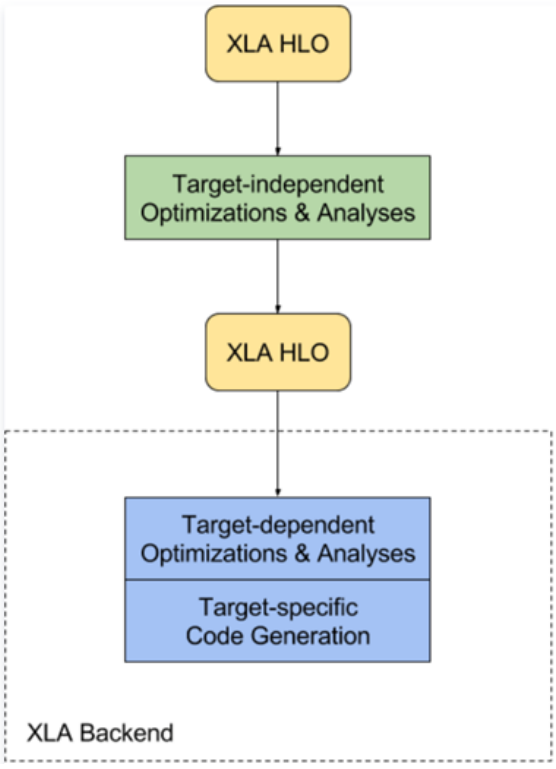


图 1.2 XLA 架构图

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。

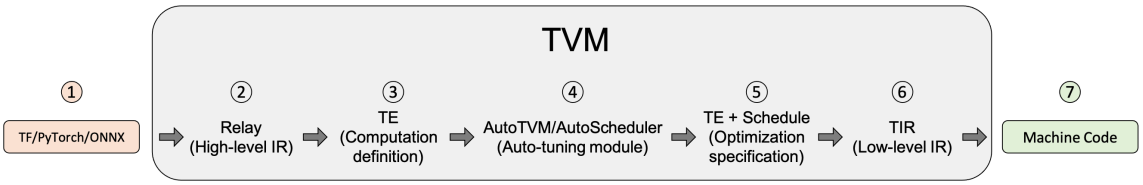


图 1.3　TVM 架构图

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。

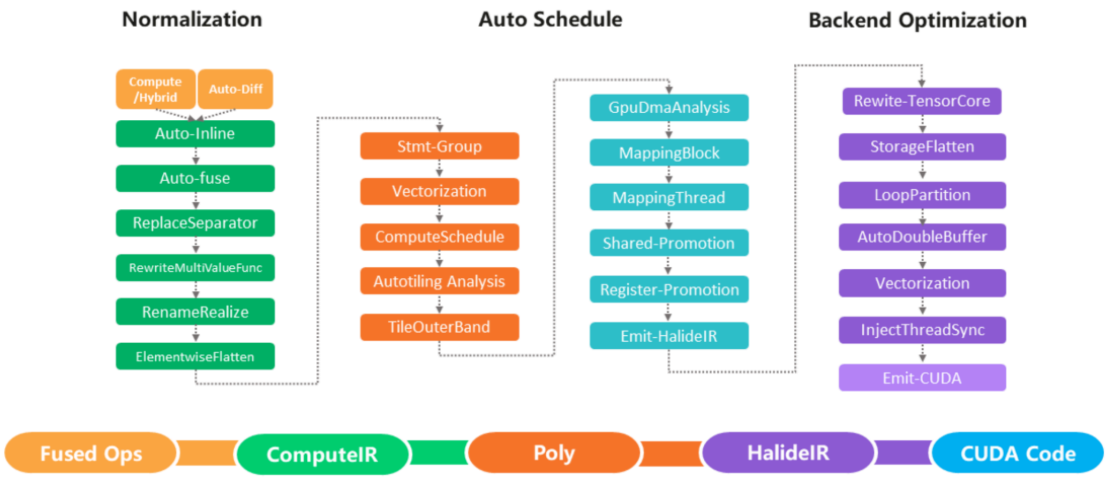


图 1.4　AKG 架构图

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。

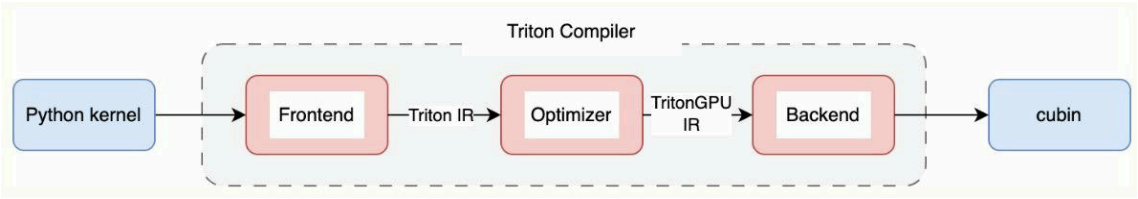


图 1.5　Triton 架构图

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。

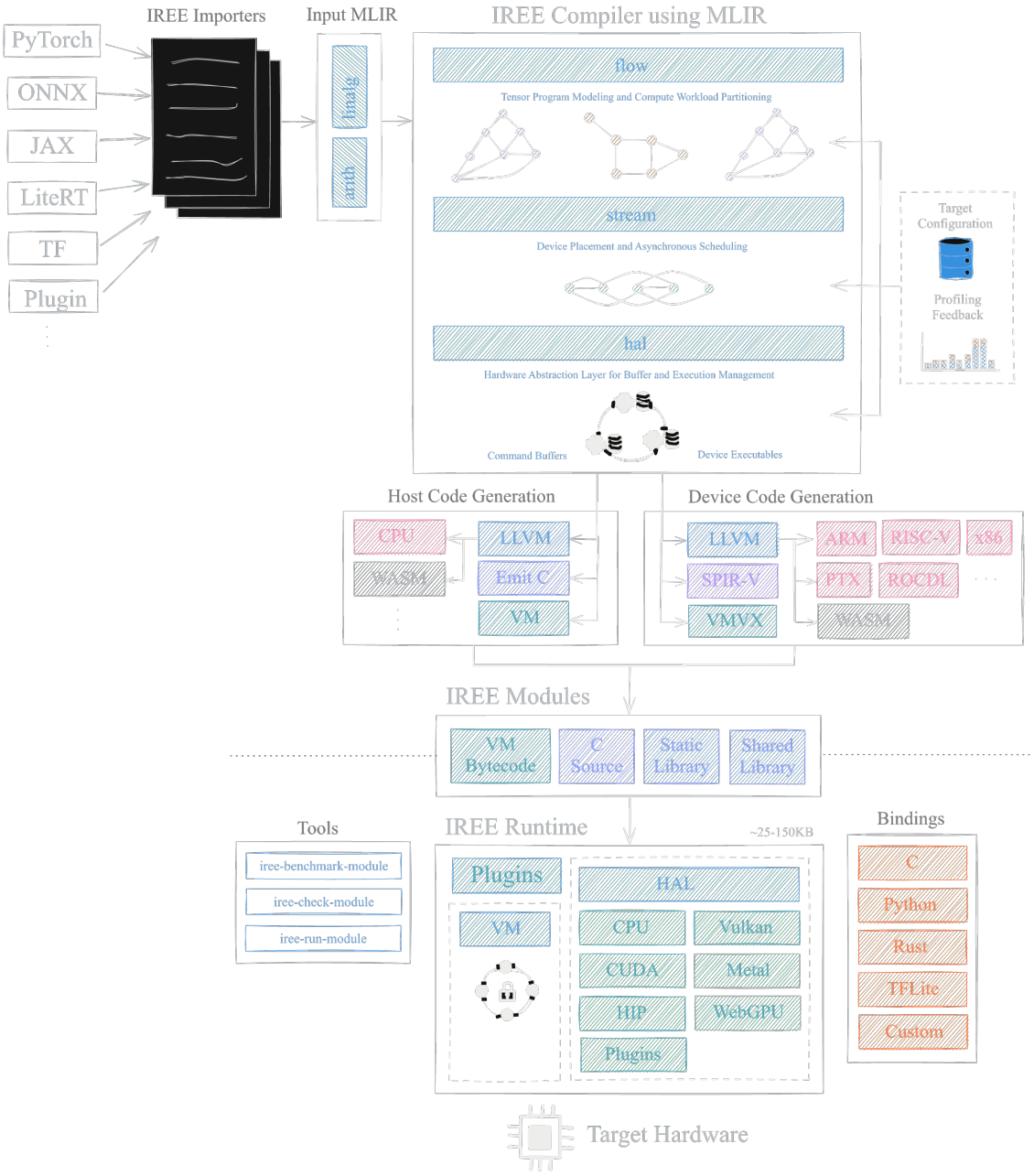


图 1.6　IREE 架构图

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，当前，深度学习加速器主要沿着以下两个方向逐步发展：

其中一个是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，为后续深度学习加速器的研究和开发提供了重要的理论基础和设计思路；Google 于 2016 年提出一种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 通用核心、NPU 加速核心的高效能特征。本文的主要研究工作包括以下几个方面：

1. 对本文基于的存算加速器进行了编译器架构的设计，具体设计中通过智能识别 NPU 指令和指令动态调度，实现高效的数据复用和全局数据流分析，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。

## 1.4 论文的章节安排

全文共包含 6 个章节，全部章节的安排内容如下：

第一章为绪论。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，然后介绍了存算一体加速器的总体架构，最后还阐述了 LLVM 的结构和其后端的大致编译流程。

第三章为指令动态调度。……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。

第四章为基于 RISC-V 存算一体芯片的编译器后端设计。……

第五章为编译器测试与分析。实验对所实现编译器的各个模块进行了正确性验证，同时选取了神经网络中比较常见的 20 种算子在 RISC-V 存算一体模拟器中性能表现。

第六章为工作总结与展望。总结了本文的研究工作，并指出了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 7 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]<br>inst[6:5] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111<br>(>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 2.7　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　存算一体架构

## 2.3 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。

### 2.3.1 LLVM 结构

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。

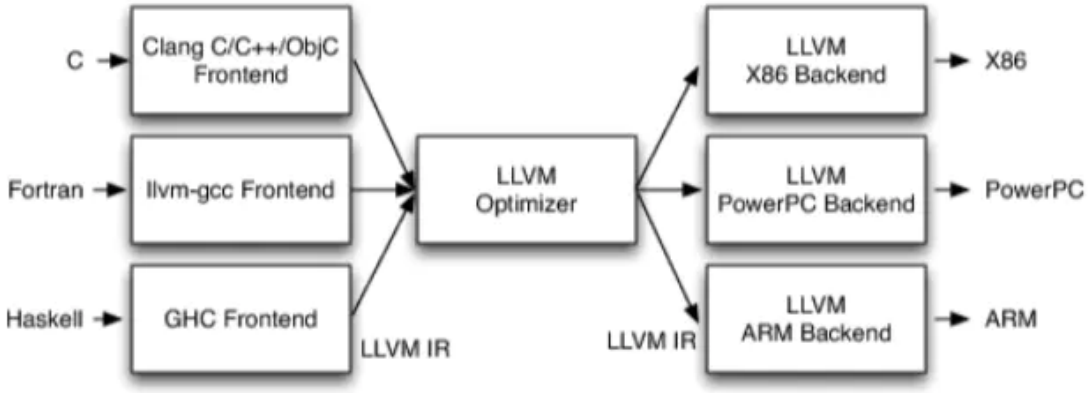


图 2.8　LLVM 编译器的结构

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

## 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

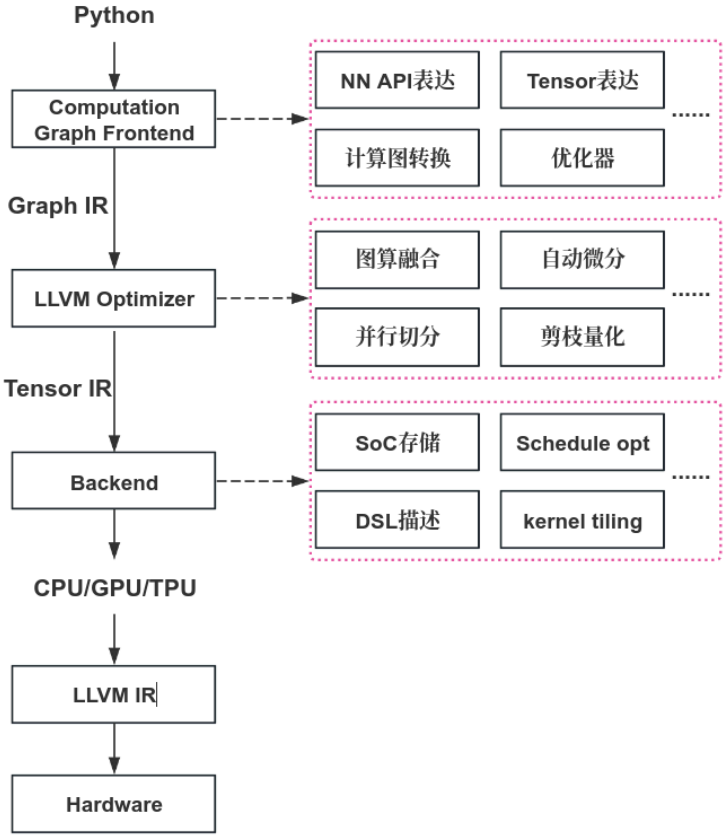


图 3.9　深度学习编译器架构

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。其大致的框架图如图 9 所示。

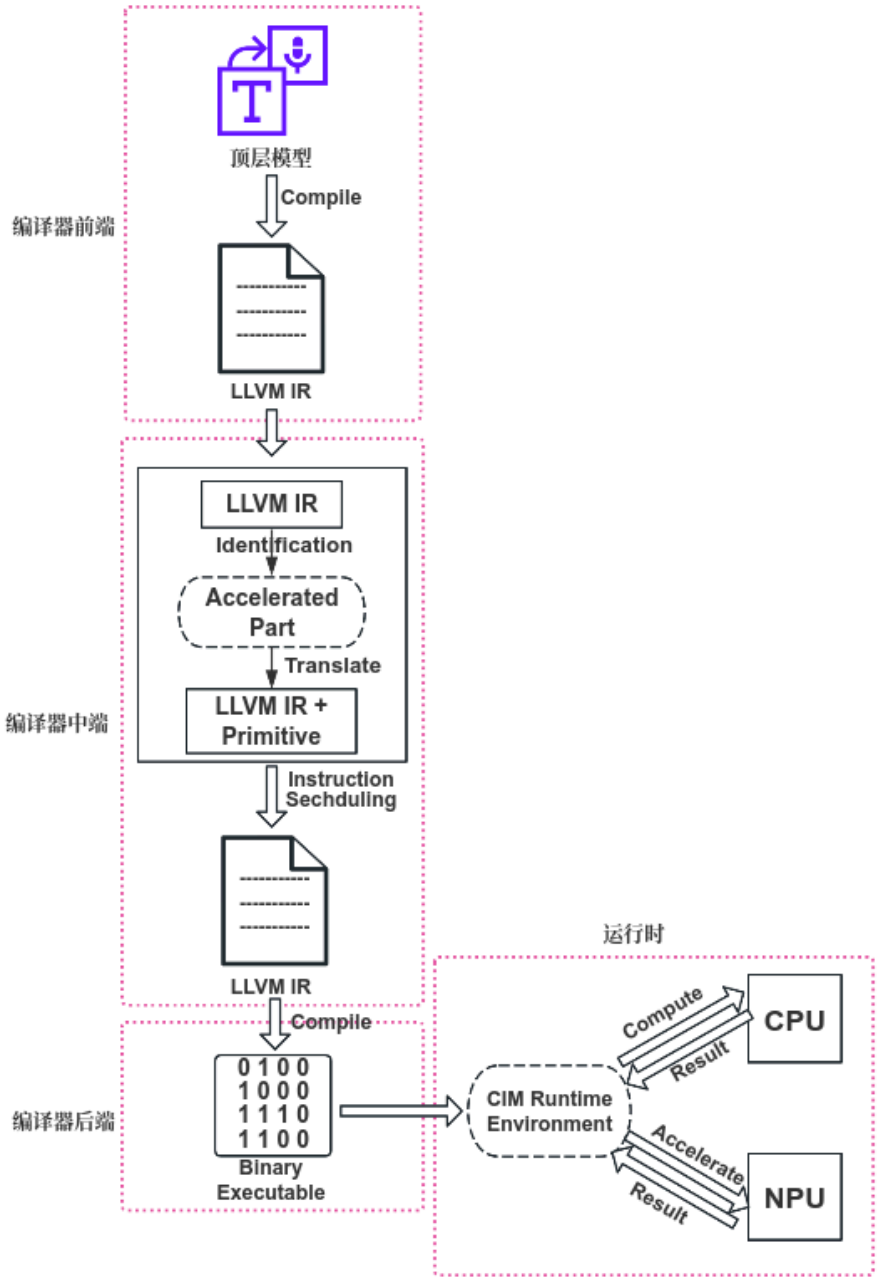


图 3.10　编译器整体架构

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 10 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 10 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，并将其转换为 LLVM IR，进行初步优化。目前本文编译器仅支持一种模型载入方

式，即借助解析器载入一个完整的模型文件，接收的模型文件格式为开放神经网络交换格式 ONNX。选定 ONNX 作为编译器输入格式的主要原因为，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，目前包括 PyTorch 和 TensorFlow 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，所以使用 ONNX 作为输入的优势为无需额外添加多个模型解析器，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 LLVM IR，对其进行体系结构无关的变换与优化以及 NPU 加速指令的识别。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，转为特定的 RISC-V 加速指令，能够有效降低算子的计算复杂度和空间复杂度，从而减少模型推理过程所需的运算时间，提升整体的计算效率。

（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 RISC-V 存算一体加速器运行所需的指令。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，包括指令的静态调度以及运行时在 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。本文编译器仅支持一种模型载入方式，通过解析 ONNX 模型文件来构建计算图进行优化。目前主流模型框架都对 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模型解析器接口的设计。


图 3.11　编译器前端工作流程图

### 3.2.1 ONNX

ONNX[25]（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 12 使用 Netron[26] 对 yolov3-tiny[27] 模型进行可视化。

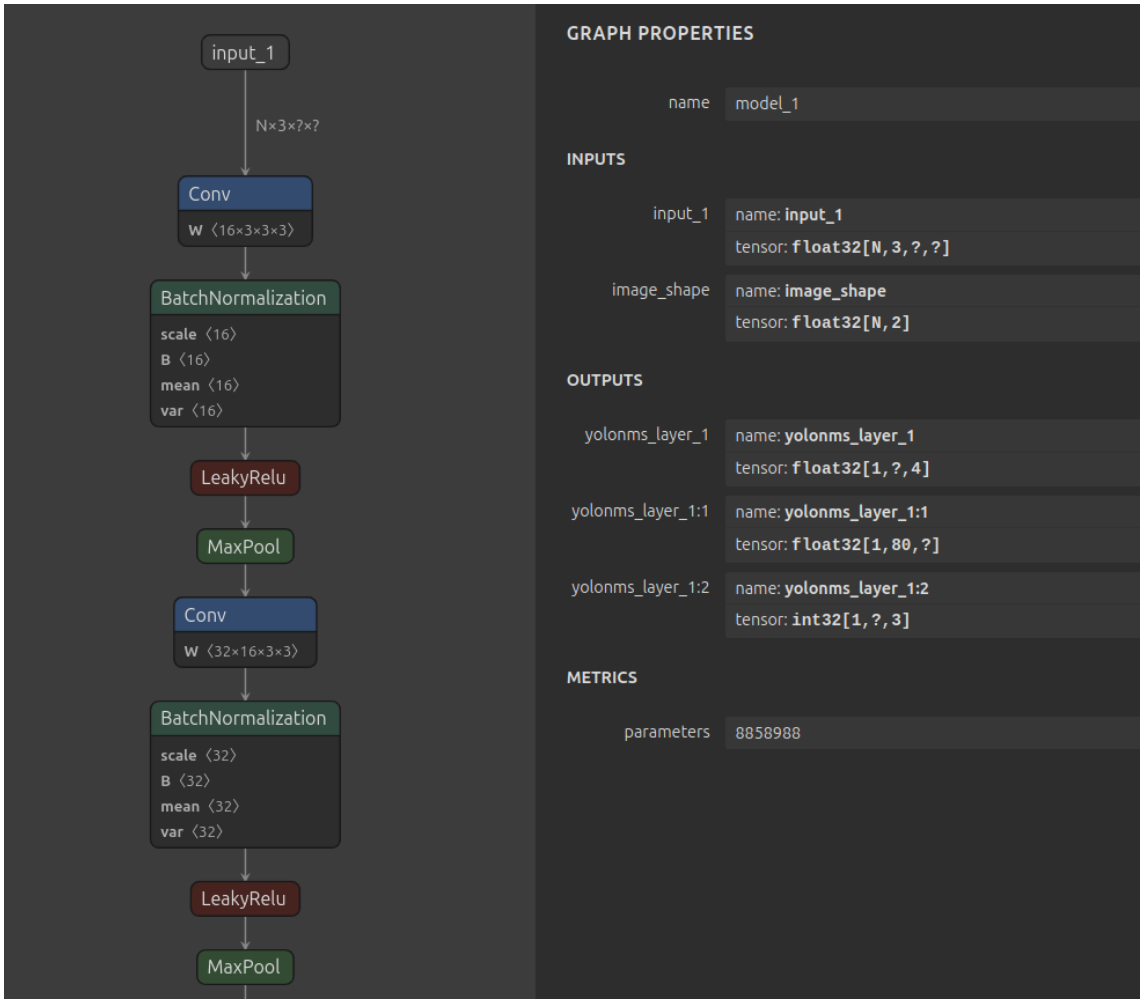


图 3.12　yolov3-tiny ONNX 可视化[26]

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"
  }
......
```

在 ONNX 中，顶层结构是一个“Model”，用于将 MeraData 与 Graph 关联起来。ONNX 中的运算符分为一组原语和函数，其中函数是可以通过其他运算符的子图来表达计算结果的运算符。Graph 用于描述函数。Graph 中包含 Node List、Input List、Output List 和初始化器（输入的常量值或默认值）的列表。无环数据流图构建为图形中节点列表的拓扑排序。Graph 中的每个 Node 都包含它调用的运算符的 Name、Input List、Output List 以及与运算符关联的 Attribute List。输入和输出可以标记为可变参数或可选参数。

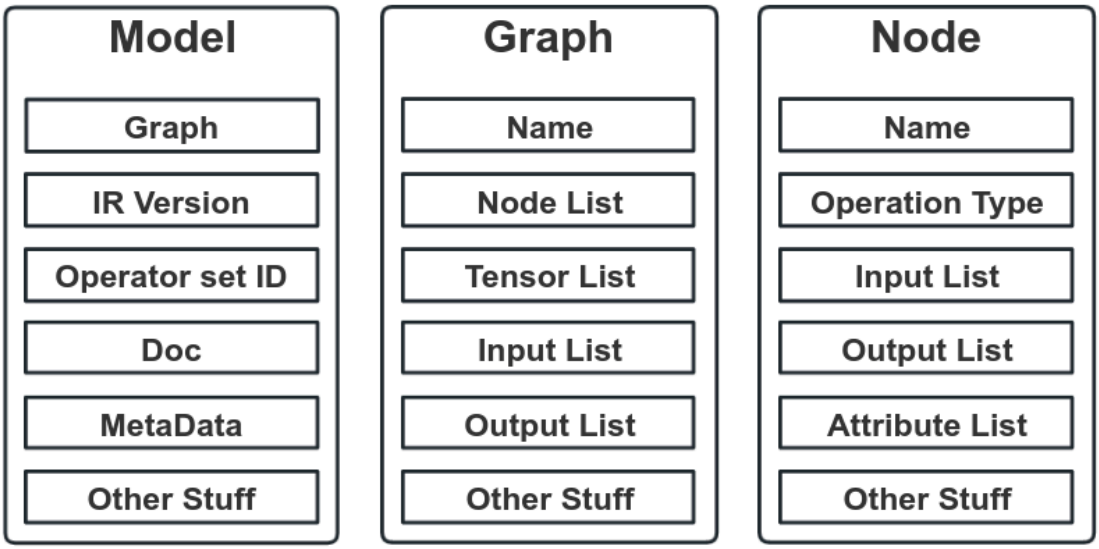


图 3.13　ONNX IR 结构

## 3.3　编译器中间优化器

编译器中间优化器是对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。通过遍历计算图节点并执行图转换来应用这些优化。优化 passes 可以捕获计算图的特定特征，并对其进行重写以实现优化。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。基于 RISC-V 存算一体加速器，本文编译器从层级优化出发，使用算子融合来提高神经网络模型执行效率，这可以减少

中间结果的内存访问，从而减少片上内存和片外内存之间的数据传输，有助于提高带宽利用率。

### 3.3.1 算子融合方式

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。

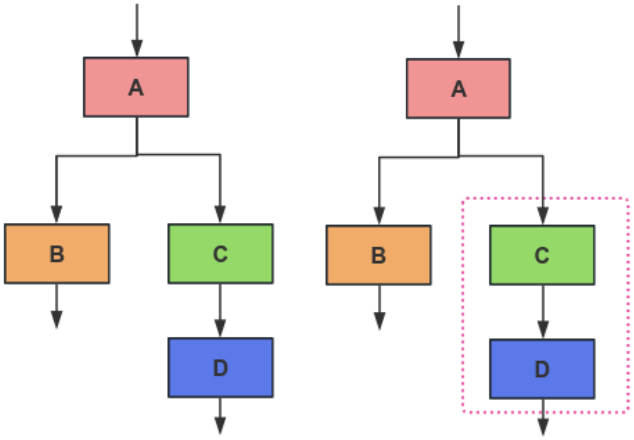


图 3.14　算子融合案例 1

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。

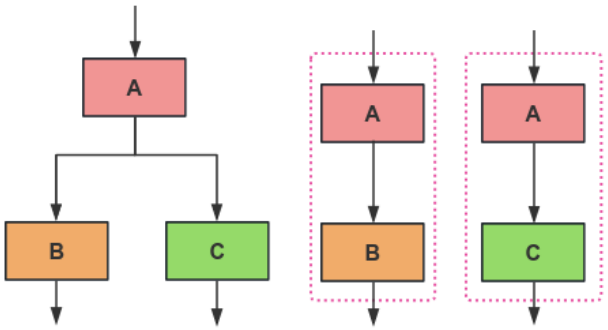


图 3.15　算子融合案例 2

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。

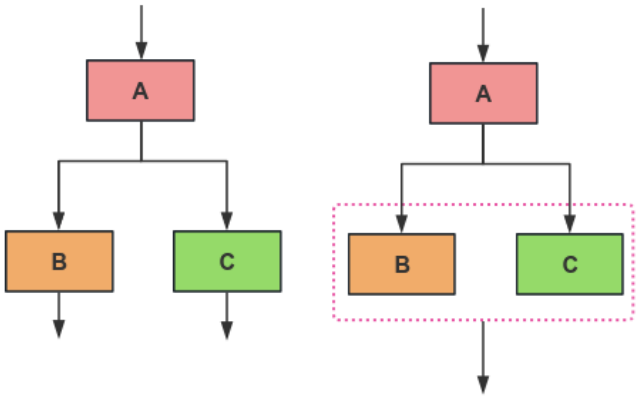


图 3.16　算子融合案例 3

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。

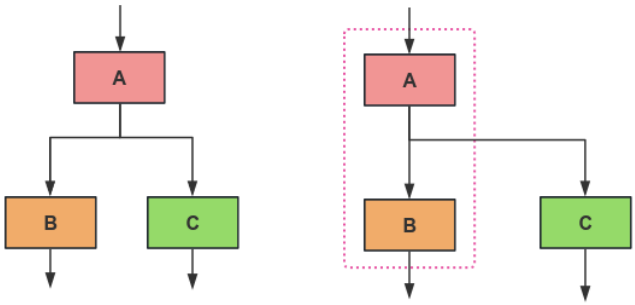


图 3.17　算子融合案例 4

### 3.3.2　算子融合案例

## 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

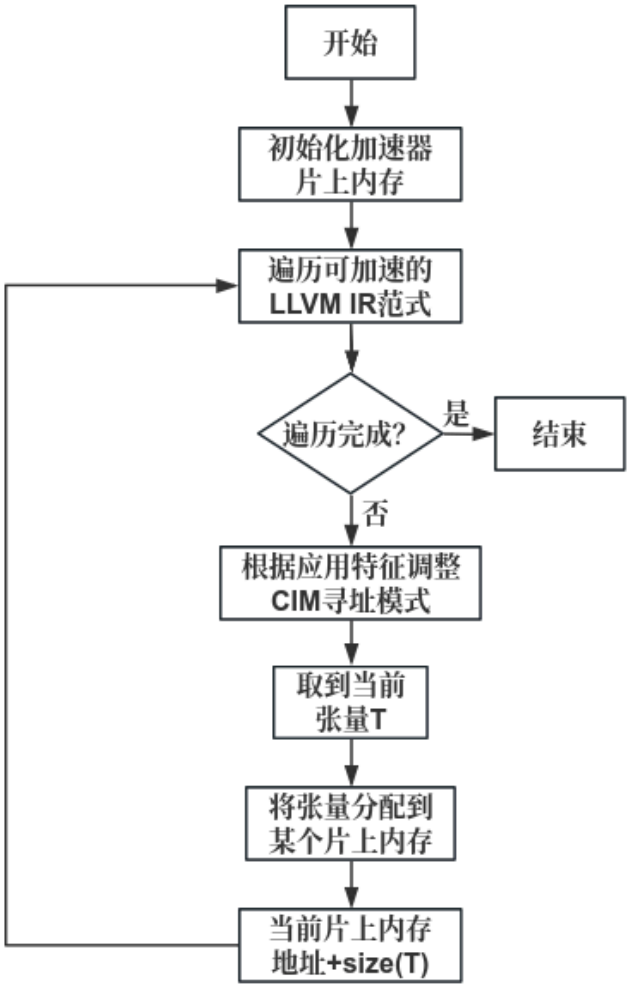


图 3.18　内存分配流程图

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 18 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。

## 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。

# 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

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。

## 4.2　可加速范式

## 4.3　本章小结

本章主要实现了以 RISC-V 存算一体模拟器为目标设备的编译器后端。

# 第 5 章　指令动态调度

如何在 CPU 和 NPU 异构计算单元之间进行指令的动态调度进行了阐述，本章则介绍编译器的后端设计。后端是编译器中与体系结构密切相关的部分，本章以课题组已有的 RISC-V 存算一体模拟器为目标设备实现了一个编译器后端，并提出了相应的计算管理方案、内存管理方案以及代码生成方案，后端设计的目的是智能识别 NPU 指令并将其映射到硬件的执行单元，实现高效的数据复用和全局数据流分析，提升整体 AI 应用的性能。

## 5.1　本章小结

# 第 6 章　编译器测试与分析

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 |

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 |

## 6.1　本章小结

# 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

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

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

[27] 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