本科毕业论文（设计）

基于 RISC-V 扩展指令的 AI 卷积加速设计

ACCELERATION OF CONVOLUTIONAL OPREATIONS BASED ON RISC-V CUSTOM INSTRUCTIONS

罗腾

哈尔滨工业大学

2024 年 5 月

密级：公开

本科毕业论文（设计）

基于 RISC-V 扩展指令的 AI 卷积加速设计

本　　科　　生：罗腾

学　　　　号：7203610626

指　导　教　师：翁睿

专　　　　业：计算机科学与技术

学　　　　院：未来技术学院

答　辩　日　期：2024 年 5 月 23 日

学　　　　校：哈尔滨工业大学

摘　要

卷积神经网络（CNN）是当今人工智能领域的重要技术之一，被广泛应用于图像分类、目标检测、自然语言处理等领域。然而，CNN 通常需要处理大规模图像数据，且需要大量的计算资源来执行复杂度较高的卷积运算。在这一背景下，针对卷积计算的硬件加速成为了一个重要的研究方向。

RISC-V 是一种开源的精简指令集架构，设计精简灵活，适用性和可扩展性强，且允许用户设计自定义指令针对特定领域计算场景进行优化。利用卷积操作的计算特性，针对性地设计 RISC-V 扩展指令和相应加速方案，提高其计算速度和效率，是非常有意义的研究方向。

Winograd 卷积算法是目前深度学习框架中最常用的卷积加速算法之一。首先，本文以 Gem5 中 RISC-V 乱序 CPU 为基准，分析了该算法的执行瓶颈；其次，在理论分析的基础上，本文设计了 5 条 RISC-V 自定义扩展指令，并完成了相应加速方案的设计与实现；最后，本文扩展了 RISC-V 交叉编译工具链，编写了内联汇编函数，并使用内联函数改写了 Winograd 卷积算法。

为了测试设计的加速效果，将改写的 Winograd 卷积算法应用于卷积神经网络 Darknet-tiny 的部分卷积层，并在 Gem5 上进行仿真模拟。实验结果表明，所设计的加速方案能够将整个 Darknet-tiny 的运行时间减少 23%，其中单个卷积层的加速比最高能达到 1.98。

关键词: RISC-V；卷积计算；自定义指令；神经网络加速；Winograd 卷积算法

Abstract

Convolutional Neural Network(CNN) is one of the pivotal technologies in today's field of artificial intelligence. It is widely used in areas such as image classification, object detection, and natural language processing. However, CNNs typically deal with large-scale image data and require substantial computational resources to perform complex convolution operations. In this context, hardware acceleration for convolutional computations has become an important research direction.

RISC-V is an open-source Reduced Instruction Set Computing (RISC) architecture, known for its streamlined and flexible design, strong applicability and scalability. Additionally, It allows users to design custom instructions tailored to optimize specific domain-specific computing scenarios. Leveraging the computational characteristics of convolutional operations, designing targeted RISC-V extension instructions and corresponding acceleration schemes to enhance their computational speed and efficiency is a highly meaningful research direction.

The Winograd convolution algorithm is one of the most commonly used convolution acceleration algorithms in current deep learning frameworks. In this paper, we first analyze the execution bottleneck of this algorithm using the RISC-V out-of-order CPU in Gem5 as a baseline. Subsequently, building upon theoretical analysis, we design five RISC-V custom extension instructions and complete the design and implementation of corresponding acceleration schemes. Finally, this article extends the RISC-V cross-compilation toolchain, writes inline assembly functions, and rewrites the Winograd convolution algorithm using inline functions.

To evaluate the effectiveness of the designed acceleration scheme, the rewritten Winograd convolution algorithm is applied to partial convolution layers of the convolutional neural network Darknet-tiny.The experimental results indicate that the designed acceleration scheme can reduce the overall runtime of Darknet-tiny by 23%, with the highest acceleration ratio of individual convolutional layers reaching 1.98.

Keywords: risc-v，convolutional computing，custom instructions，neural network acceleration，winograd convolution algorithm

目　录

第 1 章　绪　论

1.1　课题研究背景及意义

近年来，人工智能的快速发展对信息化产业产生了深远的影响。其中，卷积神经网络作为人工智能领域最为成功的算法之一，被广泛应用于图像分类[1-3]、目标检测[4,5]等领域，并取得了显著的效果。

卷积神经网络的使用一般包括训练和推理两部分，前者对算力和存储带宽要求较高，需要使用大量数据集进行多批次的前向和反向计算，一般部署在高端 AI 芯片中；后者需要的算力较低，可以部署在边缘端。近年来，越来越多基于该算法的人工智能应用被部署在边缘计算平台中，用于执行图像或视频的渲染和处理任务[6,7]。

在卷积神经网络中，高维度卷积计算承担了主要的计算负担，卷积操作的计算效率直接影响了整个神经网络的执行时间[8]。卷积层的计算通常涉及大量的矩阵乘法和加法操作，是一种数据密集型的计算任务，且在层内和各通道上具有较好的并行性。目前，GPU、TPU 等架构已经发展得比较成熟，可以很好地满足这类神经网络的计算需求，但这类计算架构的功耗高、价格昂贵，主要面向的是高端服务器领域，不适合部署在一些低端的嵌入式和边缘设备中；CPU 作为边缘计算平台中常用的核心处理器，其被设计用于执行各种不同类型的计算任务。虽然现代 CPU 通常具有超标量和乱序发射等技术，但是其指令级并行性（ILP）有限，难以充分利用硬件资源进行并行计算。此外，卷积计算涉及大量的数据读取和写入操作，而 CPU 与内存之间的数据传输速度相对较慢，访存带来的延迟会进一步降低 CPU 在执行卷积计算时的效率。如何提高 CPU 上执行卷积计算的效率，进而提升边缘设备上卷积神经网络推理的速度，也成为了亟需解决的问题。

RISC-V 作为一种适用性广泛、可扩展性强的精简指令集架构，在边缘计算平台和嵌入式处理器领域日趋成熟[9,10]。RISC-V 为用户预留了一些操作码，供用户设计自定义指令以针对特定领域进行优化。结合 RISC-V 架构，设计专用的扩展指令和相应的加速方案，可以针对卷积计算的特点进行优化，提高卷积运算的效率。与传统意义上的硬件加速器相比，基于 RISC-V 扩展指令的硬件加速方案具有更高的可编程性和灵活性，能够适应不同的网络结构和应用场景，具有重要意义。

1.2 国内外研究现状

无论是 RISC-V 指令集，还是卷积神经网络，都是近年来研究的热点。随着人工智能物联网(Artificial Intelligence of Things,AloT)的发展，不少研究者也开始着手研究基于 RISC-V 进行卷积加速的课题

1.2.1 RISC-V 指令集发展

RISC-V 指令集是由加州大学伯克利分校 David Patterson 带领的研究团队提出的一种开源的精简指令集架构[1]。得益于其开源性，RISC-V 具有良好且完整的生态系统，任何研究者都可以查看、使用和贡献到 RISC-V 的生态系统中，这为其发展提供了无限的可能性。RISC-V 的生态系统包括硬件、软件和工具链等多个方面：

在硬件方面，有许多不同的 RISC-V 处理器核可供选择，包括小型的嵌入式核、高性能的服务器级核，以及各种定制的专用核。其中大部分处理器核的设计和实现是开源的，研究者可以在开源社区中自行选择，并对处理器核进行修改扩展。经典的开源 CPU 核实现包括 BOOM(Core)、Rocket Core 和 PicoRV32 等；在软件方面，RISC-V 拥有丰富的操作系统支持，包括 Linux、FreeRTOS、Zephyr 等。许多常见的开源软件项目已经或正在逐步移植到 RISC-V 平台上，为开发者提供了丰富的软件资源；在工具方面，RISC-V 提供了完善的工具链，包括编译器、模拟器和调试器等。这些工具的开源性和广泛性为开发者提供了便利，使得他们可以在 RISC-V 平台上进行软件开发和调试。在模拟器方面，常用的 RISC-V 模拟器包括 Spike、QEMU 和 Gem5：Spike 是一个全系统模拟器，可以模拟整个计算机系统的运行，用于软件开发和系统调试。QEMU 是一款通用的虚拟机监视器，支持在宿主系统上运行 RISC-V 操作系统和应用程序。而 Gem5 则是一款指令级模拟器，用于性能分析、系统设计和验证等方面。

在指令集设计方面，RISC-V 采用的是模块化的设计方式，包含一个基本的指令子集和多个可选的标准扩展指令子集。其中，基础指令子集被标记为 I，包含了所有必须被实现的整数指令。可选的标准扩展指令子集包括 A（原子操作指令）、C（压缩指令）、D（双精度浮点指令）、F（单精度浮点指令）和 M（乘除法指令）等。研究者可以根据需要对标准扩展指令子集进行灵活选择，以满足不同场景的应用需求。

RISC-V 是真正意义上的的精简指令集架构。如图 1-1 所示，RISC-V 指令集 32 位指令共包含 6 种基本指令格式，各指令操作码均位于指令低 7 位。RISC-

V 指令包含三个寄存器索引：rs1（源操作数 1）、rs2（源操作数 2）和 rd（目的操作数），且所有寄存器字段被放置在了相同的位置。在 imm（立即数）字段的设计上，设计者也进行了精心考量，以确保适应各种运算和数据处理需求。这样的设计，使得 RISC-V 处理器的译码部分能够得到简化，避免其成为处理器时序上的关键路径。

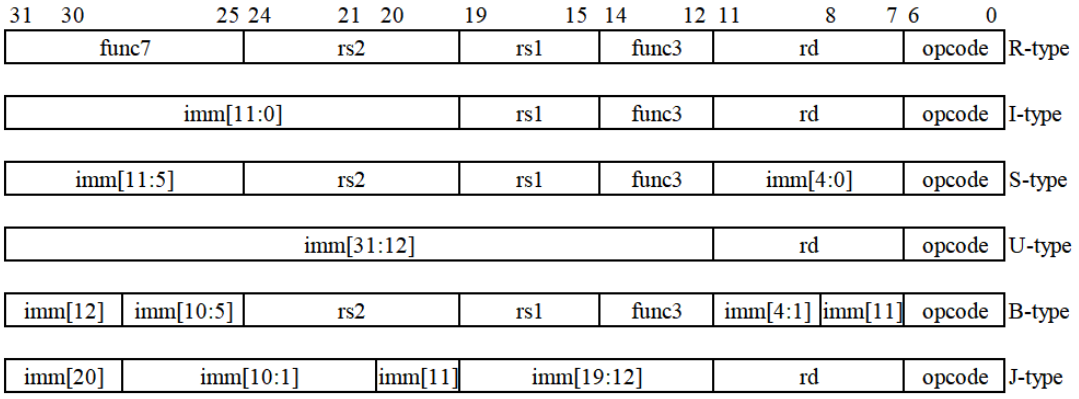

图 1-1　RISC-V 基本指令格式

除此之外，RISC-V 还明确支持特定领域的定制扩展：在指令集的设计方面，RISC-V 基本指令子集仅使用了部分编码空间，并为开发者预留了可供选择的操作码；在编译工具链的支持方面，指令定义相关部分的源码已经开源，用户能够方便地在工具链中添加自定义指令，重新编译工具链源码后，交叉编译工具即可完成对扩展指令的识别。RISC-V 指令集的这一特性为各领域的创新和性能优化提供了极大的便利，研究者们设计了各种定制扩展指令，在信息安全[12,13]、数字信号处理[14,15]、人工智能[16-18]等特定领域实现了计算加速。

1.2.2　卷积计算及其加速设计研究

卷积神经网络的概念最早由神经计算学者 Yann LeCun 等人在 1989 年提出，他们在发表的一篇论文中提出了卷积神经网络 Lenet[19]，用于识别手写数字。在当时，由于计算资源有限、数据集缺乏和梯度消失等原因，CNN 的发展相对缓慢。直到 2012 年，AlexNet 在 ImageNet 大规模视觉识别挑战赛中取得了较大成功[20]，这类算法才逐渐成为主流。AlexNet 通过增加网络深度和参数量、利用 GPU 进行加速训练提升了网络性能和实用性，为后续更深、更复杂的神经网络奠定了基础。

卷积神经网络通常包含卷积层、池化层、激活函数层等交替出现的网络层。近年来，卷积神经网络日趋复杂，随着网络深度和参数量不断增加，其对算力

的要求也不断增大。其中，卷积计算包含大量矩阵乘法、加法运算，占据了大部分计算负担，在许多情况下超过 90%[21]。因此，针对卷积任务的计算和访存特性，在软硬件方面设计合适的策略进行加速，从而提升卷积神经网络的执行效率，一直是深度学习领域研究的热点。

目前深度学习框架中常用的卷积加速算法包括 FFT（快速傅里叶变换）、im2col、Winograd 算法等，它们适用于不同的计算场景。FFT 算法利用快速傅里叶变换的性质，将卷积操作转换为频域上的元素乘法，适用于卷积核较大的情况，然而，该算法涉及到复数运算和频域转换，在一些场景下可能会引入一定的误差；im2col 算法首先将输入图像转换为一个合适的矩阵，然后使用矩阵乘法来执行卷积操作。这种算法需要利用高性能的矩阵乘法库，且在矩阵转换的过程中会增加内存开销，对访存带宽的需求较大，不适合于边缘计算平台；Winograd 算法通过使用加法运算来替代乘法，减少了计算复杂度，进而加速卷积，在卷积核尺寸较小的情况下计算效率较高。

在硬件层面，新的硬件架构和加速器如 GPU[22]、ASIC[23,24]和 TPU[25]等得到了广泛应用。这些架构的共同特点在于它们都采用了高度并行化的设计，计算能力强大，能够高效地加速深度学习任务中的训练和推理过程。然而，这类架构也有着各自的缺点：GPU 的功耗较高；ASIC 开发周期长且高度定制化，缺乏灵活性；TPU 依赖特定框架且价格昂贵。综合来看，这类架构在资源受限的环境下适用性不强，基本上只能面向高性能服务器。在物联网节点和低端嵌入式设备中，往往需要更加节能紧凑和更具通用性的加速方案，以适应其特定的应用场景和资源限制。

近年来，越来越多基于卷积神经网络的人工智能应用被部署在边缘计算平台。随着 RISC-V 指令集在物联网、嵌入式等领域的普及，不少研究者开始尝试使用自定义的 RISC-V 扩展指令实现卷积加速。作者在[26]中实现的紧密型 CNN 加速器的基础上，利用自定义指令控制加速器的启动、计算等流程，并通过优化加速链互联结构，在低硬件成本的条件下实现了卷积计算 6.27 倍的加速比。Yu 等人在[27]中设计了自定义的数据操作和存取指令，并在 RISC-V 开源处理器 Zero-riscy 上进行相应扩展，在流水线中实现了这些指令。作者在 CNN 和单层卷积计算任务上进行实验验证，加速比分别是 1.5 和 2.48-2.82。此外，该设计中针对卷积计算的数据操作指令采用了细粒度的设计方式，因此可以通过组合实现灵活的扩展，支持各类大小的卷积核运算。为了优化卷积神经网络推理计算中的卷积、激活和池化操作，Wang 等人提出了 7 条 SIMD 扩展指令[28]。其主要的贡献在于设计了 CONV23 指令，利用 Winograd 算法加快了$F(2\times 2,3\times 3)$

的计算速度。设计的自定义指令能够在加速器中重用数据，并以批处理的形式执行。他们将设计的加速器集成在 RI5CY 开源处理器上，并使用 FPGA 进行实现验证，在 LeNet、VGG16 和 ResNet18 上均取得了较高的性能提升。Li 等人[29]则是完整地设计了一个可高效计算卷积神经网络的 RISC-V 五级流水线处理器。除了设计并实现了四条向量扩展指令之外，作者考虑到卷积操作中不同通道的特征图具有计算独立性，采用 Ping-pong 操作优化数据流处理方式，提高了层内计算的并行性。MARCO 等人在[30]中重点关注存储对卷积计算的影响，设计了 8 或 16 位 posit 与 32 位 IEEE 浮点数之间进行转换的扩展指令，并在 CVA6 核心中集成了一个轻量级的 posit 处理单元。Posits 的精度与浮点数相当，但对存储空间的占用更少，因此能够通过压缩神经网络权重参数的方式提高计算效率，同时不损失计算精度。

从加速结构来看，上述相关研究中主要的方式有两种：一种是使用 CPU+加速器的结构，另一种则是直接对 CPU 流水线进行修改。第一种方式需要为加速器单独设计控制逻辑，且需要堆叠大量计算部件，对访存带宽要求较高，不适合于部署在边缘设备中；在第二种方式中，研究者选择的基准 CPU 都为简单的顺序流水线，具有较少的流水级数。这种 CPU 更易于扩展指令的实现和适配，但其性能远低于目前普遍使用、性能较高的乱序处理器，因此在某种程度上可能缺乏实际应用的意义。

而从设计的扩展指令来看，最简单普遍的是使用乘累加指令，这类指令针对的是卷积计算原始的窗口滑动法，将窗口内元素和对应权重相乘并累加的操作合并为一条指令，减少指令数量进而提高计算效率。这类指令电路延迟较大，若在一个时钟周期内完成计算可能会影响 CPU 的主频，对计算效率的提升带来一定的影响。此外，上述加速方案基本都采用了 SIMD（单指令多数据流）的方式来实现并行化处理。通过扩充向量寄存器组并设计相应的向量化指令，充分挖掘卷积计算中的数据并行性，达到向量化处理的效果，进而加速卷积计算。

1.3 研究思路与内容

本文以 Winograd 卷积算法在开源指令级模拟器 Gem5 乱序 CPU 模型上的执行与加速优化为研究对象，研究思路为：首先分析 Winograd 卷积算法在 CPU 上的执行瓶颈，总结出潜在优化点；然后，基于这些优化点，设计并在模拟器上实现基于 RISC-V 扩展指令的卷积加速方案；接着，通过仿真实验验证加速方案的正确性和有效性。

结合上述研究思路，本文的主要研究内容包含以下部分：

（1） RISC-V 指令集与定制指令设计：

a) 学习 RISC-V 指令集的基本结构，了解其模块化的设计。

b) 考量 RISC-V 灵活性、开放性以及社区支持等方面的优势，分析其作为定制 AI 卷积加速基准指令集的合理性。

c) 基于 Winograd 卷积算法的计算瓶颈分析，设计合适的自定义指令和相应的加速方案。

（2） 扩展指令实现与工具链支持：

a) 实现设计的扩展指令和加速方案，集成于 Gem5 乱序 CPU 模型中。

b) 修改 RISC-V 交叉编译工具链，使编译器能够识别自定义指令。

c) 使用内联汇编改写 Winograd 卷积算法。

（3） 功能验证与性能评估：

a) 进行功能级和系统级的验证，验证指令实现的正确性。

b) 使用合适的卷积神经网络，评估处理器执行卷积操作的加速效果，分析设计的优势与不足。

1.4 论文章节安排

论文分为五个章节，具体安排如下：

第 1 章介绍了本文的研究背景和意义，对课题相关研究现状进行了简要概述和分析，介绍了本文的研究思路与内容。

第 2 章介绍了本文的理论分析。首先，介绍了 Winograd 卷积算法的基本原理和优势，以及该算法在二维卷积上的应用；随后，介绍了本文使用的性能分析平台-开源指令级模拟器 Gem5，对其中使用的乱序 O3-CPU 模型进行了详细介绍；最后，在 Gem5 上进行仿真，分析 Winograd 卷积算法主要执行瓶颈，总结潜在优化点。

第 3 章介绍了本文的加速方案设计与实现。首先，介绍了硬件方面的工作，包括设计的 RISC-V 扩展指令和相应的加速方案，以及加速方案在 Gem5 乱序 CPU 上的实现；随后介绍了软件方面的支持，包括编译工具链的修改、内联汇编函数的编写，以及 Winograd 卷积算法的改写。

第 4 章介绍了本文的实验分析部分。首先，简要介绍了加速方案的功能验证，然后重点介绍了加速方案的性能测试，并对实验结果进行了分析。

最后的结论部分对全文进行了总结，归纳了本文主要完成的工作和创新之处，并指出了设计的不足，对后续研究工作做了展望和讨论。

第 2 章　Winograd 算法概述及性能瓶颈分析

本文研究的卷积算法为 Winograd 算法，本章将重点介绍该算法的原理和计算过程，并利用模拟器 Gem5 对其进行分析，通过查看输出信息和流水线可视化的方式分析其在 CPU 上执行的性能瓶颈。

2.1　Wingread 算法原理介绍

Winograd 卷积算法是当前深度学习框架中最常用的卷积计算方法之一，最早由学者 Lavin 提出[31]。其核心思想是通过减少卷积运算中的乘法操作来降低计算延迟，当卷积核尺寸较小时，这种算法的计算效率较高。下面首先使用一维卷积介绍该算法，然后介绍该算法在二维卷积上的应用。

将一维卷积表示为$F(m,r)$，其中 m 是卷积输出元素的尺寸，r 是过滤器的尺寸。一个 4 元素输入向量$d=(d_0,d_1,d_2,d_3)$，和一个 3 元素过滤器向量$g=(g_0,g_1,g_2)$之间的一维卷积，会生成一个 2 元素的输出，这一操作可以被表示为$F(2,3)$。将 4 元素输入向量转换为一个 2 维矩阵$\begin{bmatrix}d_0 & d_1 & d_2\\ d_1 & d_2 & d_3\end{bmatrix}$，则$F(2,3)$可以表示为两矩阵相乘的形式，见式(2-1)。

$$F(2,3)=\begin{bmatrix}d_0 & d_1 & d_2\\ d_1 & d_2 & d_3\end{bmatrix}\begin{bmatrix}g_0\\ g_1\\ g_2\end{bmatrix}=\begin{bmatrix}r_0\\ r_1\end{bmatrix} \tag{2-1}$$

式中 $\begin{bmatrix}r_0\\ r_1\end{bmatrix}$——输出的 2 元素向量；

由公式(2-1)可知，直接计算$F(2,3)$涉及到 6 个乘法操作和 4 个加法操作。若采用 Winograd 算法，可以先计算以下 4 个数：

$$\begin{aligned} m_0&=(d_0-d_2)g_0 \quad m_1=(d_1+d_2)\frac{g_0+g_1+g_2}{2}\\ m_3&=(d_1-d_3)g_2 \quad m_2=(d_2-d_1)\frac{g_0-g_1+g_2}{2}\end{aligned} \tag{2-2}$$

然后再由公式(2-3)即可计算得到$F(2,3)$的结果：

$$F(2,3)=\begin{bmatrix}d_0 & d_1 & d_2\\ d_1 & d_2 & d_3\end{bmatrix}\begin{bmatrix}g_0\\ g_1\\ g_2\end{bmatrix}=\begin{bmatrix}m_0+m_1+m_2\\ m_1-m_2-m_3\end{bmatrix} \tag{2-3}$$

g是过滤器，相关操作可以提前计算，因此总共的运算量包括公式(2-2)中与d相关的 4 次加减法、d和g之间的 4 次乘法，以及公式(2-3)中的 4 次加减法，一共是 4 次乘法和 8 次加减法。由于乘法运算比加法运算复杂，在硬件上会造成更大的延迟，因此 Winograd 算法是通过减少乘法次数实现了计算加速。

如图 2-1 所示，二维 Winograd 卷积$F(2\times 2,3\times 3)$完成的是尺寸为4×4的输入矩阵与尺寸为3×3的卷积核进行卷积，得到尺寸为2×2的输出矩阵的操作，嵌套使用一维 Winograd 卷积$F(2,3)$算法可推导得出其公式。

假设Y是尺寸为2×2的二维卷积输出矩阵、d是尺寸为4×4的输入矩阵、g 是尺寸为3×3的卷积核，并定义几个辅助矩阵：$A^T=\begin{bmatrix}1 & 1 & 1 & 0\\ 0 & 1 & -1 & -1\end{bmatrix}$、$B^T=\begin{bmatrix}1 & 0 & -1 & 0\\ 0 & 1 & 1 & 0\\ 0 & -1 & 1 & 0\\ 0 & 1 & 0 & -1\end{bmatrix}$、$G=\begin{bmatrix}1 & 0 & 0\\ 1/2 & 1/2 & 1/2\\ 1/2 & -1/2 & 1/2\\ 0 & 0 & 1\end{bmatrix}$，则二维 Winograd 卷积可由式(2-4)计算得到：

$$Y=A^T[(GgG^T)\odot(B^TdB)]A \tag{2-4}$$

式中GgG^T是对卷积核进行转换，可以提前完成计算。实际计算式(2-4)时，需要完成三个步骤：对输入矩阵d应用两次矩阵乘法进行转换、将转换后的卷积核与输入矩阵进行逐元素乘法、对点乘结果矩阵应用两次矩阵乘法。由于式中涉及的辅助矩阵A、B的各元素是固定的，且都为 0、1 或-1，因此实际上这些矩阵乘法可以转换为加法操作。例如，假设矩阵d由 0-3 行、0-3 列组成，$d_{x,y}$表示输入矩阵第 x 行、第 y 列元素，则矩阵乘法B^Td仅需计算四个加减运算：$d_{0,0}$-$d_{0,2}$、$d_{1,1}$+$d_{1,2}$、-$d_{2,1}$+$d_{2,2}$、$d_{3,1}$-$d_{3,3}$，其余矩阵乘法同理。因此，最终整个算法中仅需计算步骤二中的 16 个逐元素乘法，而使用原始滑动窗口法则需要计算 36 次乘法，由此可达到加速的效果。

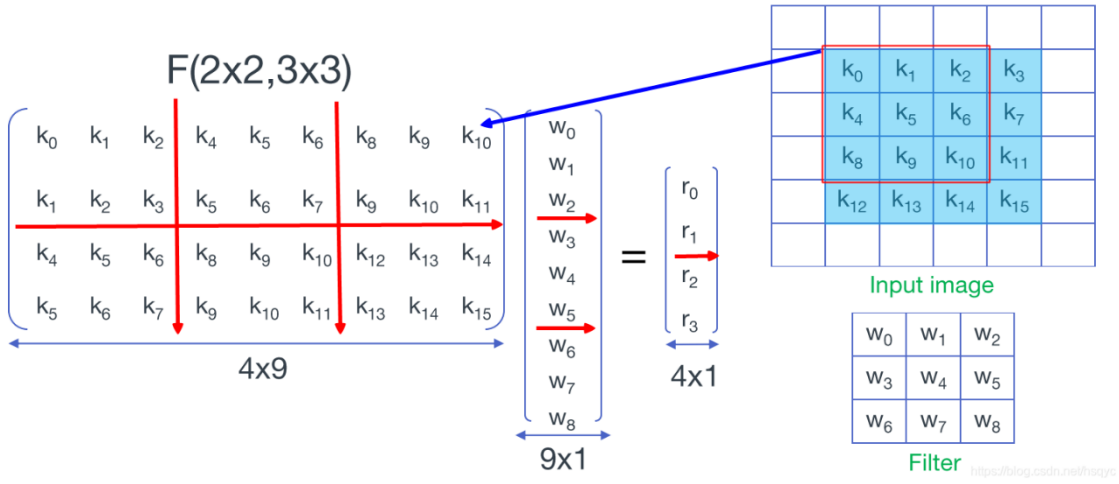

图 2-1　二维 Winograd 卷积算法

以上介绍的算法，要求输入矩阵的尺寸为4×4。在实际应用场景中，卷积层输入矩阵的尺寸往往远大于此。因此，需要对输入矩阵进行拆分，将其分为多个尺寸为4×4的小矩阵（各个小矩阵之间重叠的大小为 2），分别应用上面介绍

的算法，得到输出矩阵。

在使用 Winograd 二维卷积算法对卷积神经网络的某一卷积层进行计算时，涉及到的操作步骤包括：输入矩阵切块重排、卷积核转换、对重排矩阵各块分别应用二维 Winograd 卷积算法$F(2\times 2,3\times 3)$、输出矩阵重排。其中，计算复杂度最大、耗时最长的步骤即为$F(2\times 2,3\times 3)$的多次循环迭代，本文将对此进行重点研究和分析，并针对性地设计出加速方案。

2.2 性能分析平台介绍

本文使用的性能分析平台是 Gem5，这是一款开源的指令级模拟器，用于对计算机系统的性能进行评估和研究，下面对其进行简要介绍。

2.2.1 基本功能与使用方法

目前，Gem5 支持 SE（Syscall Emulation，系统调用模拟）和 FS（Full System，全系统）两种仿真模式。前者不考虑操作系统和其他硬件的影响，主要用于模拟处理器的执行过程，对处理器架构进行功能验证和性能评估；后者模拟了包括处理器、内存、硬盘和操作系统等组件在内的完整计算机系统，常用于研发操作系统或测试大型应用程序。本文主要研究的是 Winograd 算法在通用处理器上的执行瓶颈，因此统一使用 SE 仿真模式。

Gem5 支持多种指令集架构，包括 X86、Alpha、MIPS、ARM、RISC-V 等，并且也支持多种 CPU 架构，如 Simple-CPU（提供功能模拟）、Minor-CPU（四级顺序处理器）、O3-CPU（乱序处理器）、Trace-CPU（利用 trace 快速仿真）等。Gem5 中的指令集和 CPU 模型是解耦的，用户可以根据需要进行组合，本文使用的是 RISC-V 乱序 O3-CPU 模型。

Gem5 提供了一个高度可配置的模拟平台，用户可以编写仿真脚本，对仿真系统中的各种参数进行配置，包括 CPU 类型、内存配置、工作负载等。图 2-2 展示了一种典型的系统配置，该系统可分为 CPU 内外两部分：CPU 内部包含流水线、内存管理单元 MMU、核内的 1 级数据缓存和 1 级指令缓存。CPU 外部包含 2 条总线，其中 L2 总线对 CPU 内部的访存请求进行仲裁，并连接到核外 2 级缓存；而系统总线则是连接了 2 级缓存和主存。

使用配置好的系统即可运行相应程序进行仿真分析，完成仿真后，Gem5 会生成相应的输出信息，如仿真延迟、Cache 命中率、功能部件使用率等指标。此外，在需要对处理器微架构及程序运行细节进行分析时，还可通过指定 debug

标志的方式查看相应的 trace 文件。例如，可启用 O3CPUAll 标志，查看乱序 O3-CPU 在运行仿真过程中，各流水级在不同周期的 debug 输出；启用 O3PipeView 标志，还能够结合相应插件进行流水线可视化。

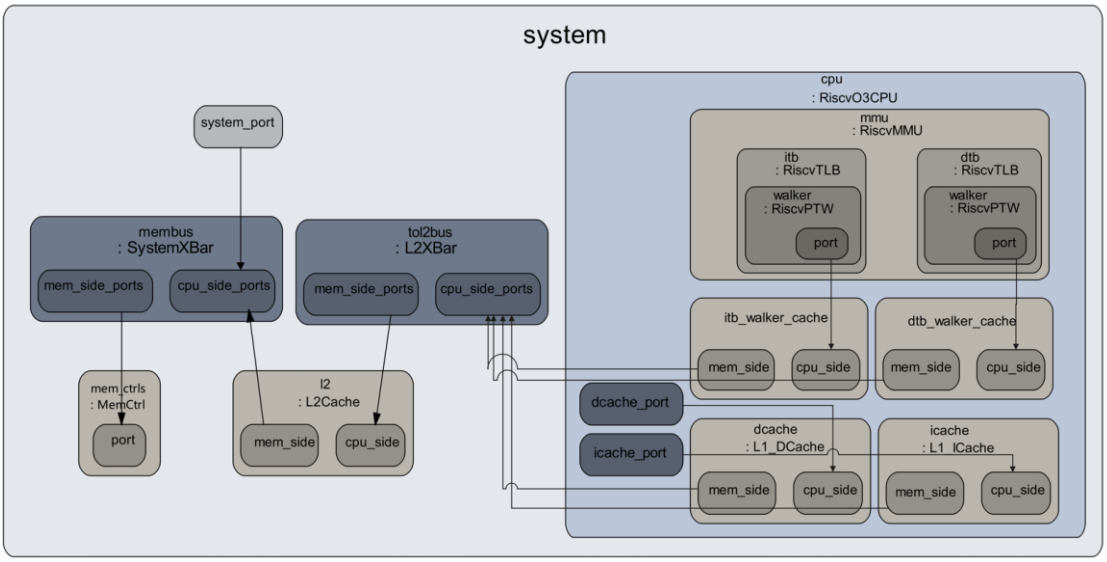

图 2-2　Gem5 系统配置

2.2.2　Gem5 乱序 CPU 模型介绍

本文需要分析 Winograd 卷积算法在 CPU 上的执行瓶颈，选用了 Gem5 中乱序 O3-CPU 架构作为基准 CPU 模型，下面对该架构进行简要介绍。

如图 2-3 所示，Gem5 乱序 O3-CPU 模型包含 Fetch（取指）、Decode（译码）、Rename（重命名）、Issue（发射）、Execute（执行）、Writeback（写回）、Commit（提交）7 个阶段，使用统一的物理寄存器进行寄存器重命名，集中式发射队列（Issue Queue，IQ）存放待执行指令，并使用 ROB（Reorder Buffer，再定序缓冲）实现按序提交。

流水线前端包含取指、译码、重命名三个阶段。取指阶段，处理器每周期会根据线程优先级策略，选择相应的线程进行取指令操作，将取出的指令放入取指队列中。译码阶段，会对指令进行解码，得到指令的基本信息，包括寄存器使用情况、指令种类、操作码等。在该阶段，还将检测分支预测器的预测情况，对分支预测失误进行处理。在重命名阶段，使用 RAT（Rename Alias Table，重命名映射表）对指令的源寄存器进行重命名操作，并使用 Freelist（空闲寄存器列表）管理处理器中可使用的物理寄存器，将目的物理寄存器号分配给需要写

寄存器的指令。在此阶段，指令会被写入 ROB 中。

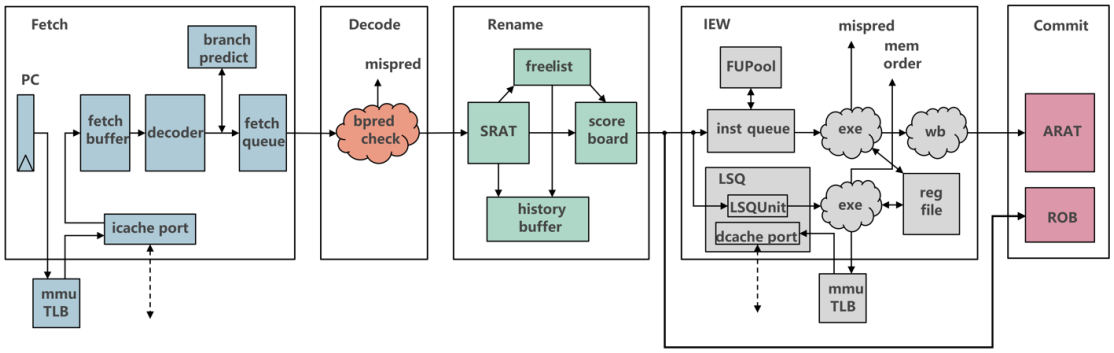

图 2-3　Gem5 乱序 O3-CPU 架构

流水线后端包括图中的 IEW 阶段和提交阶段，其中 IEW 代表发射（I）、执行（E）和写回（W）三个阶段。在发射阶段，指令会被统一分配到集中式发射队列中，其中操作数就绪的指令会被写入就绪队列中，并按照一定的调度策略被发射到相应的执行单元。在执行级，各指令在相应的执行单元中完成其指定操作，Gem5 会按照配置对指令的执行延迟进行模拟。完成执行后，指令会进入写回阶段，在该阶段唤醒后续与其有数据相关的指令。指令最终在提交阶段按序退休，完成对处理器状态的修改，该阶段还会对指令的异常、分支预测失误等特殊情况进行处理。

本文将对该 CPU 模型进行扩展，主要修改流水线后端 IEW 阶段的代码，以完成扩展指令在乱序流水线中的适配，详细内容将在第 3 章中进行介绍。

2.3　算法性能瓶颈分析

编写仿真脚本，指定 CPU 架构为 Gem5 中自带的 RISC-V 乱序 O3-CPU 模型，仿真负载为 Winograd 卷积算法前向推理任务，其余配置均采用 Gem5 中默认配置。运行仿真后，通过查看统计输出信息和流水线可视化的方式分析 Winograd 卷积算法最内层循环$F(2\times 2,3\times 3)$的执行瓶颈。

图 2-4 展示了执行 Winograd 卷积算法时，其中典型代码段对应的流水线情况，图中左侧指示了每一条具体指令对应的反汇编代码，包括指令名称、操作数、使用的寄存器等。右侧各种不同颜色的方框代表不同的流水级，方框上标记的数字为指令在该流水级累计的延迟周期，不同指令在不同流水级的延迟不同。此外，图中的黄线指示了具有数据依赖关系的指令组，操作数未就绪的指令需要等待写回其源寄存器的指令完成执行。下面针对该典型代码段的执行情况，进行具体分析。

图 2-4　Winograd 算法典型代码段执行情况

由图可知，该代码段中的主要延迟来源于 flw（浮点加载）指令，此处的浮点加载是在完成从内存中读取卷积核权重的操作。结合源程序可知，卷积核的权重在算法内循环中是不变量，仅需在外循环中进行加载。但由于架构性浮点寄存器不足，编译器在编译时仍然需要将这部分值在内循环多次迭代过程中重复加载。这样在实际运行时，就会造成不必要的浮点加载延迟，而浮点加载完成的是从主存中去操作数的操作，通常位于数据相关链的顶端，若其不能及时完成，后续需要用到相应数据的相关计算指令也将被堵塞。这一问题，可以通过增加硬件上的缓冲寄存器解决，该缓冲寄存器仅需在外循环中加载卷积核权重，可以减少因浮点加载延迟带来的性能损失。

此外，图中包含 11 个 flw 指令，这些指令实际上完成的是读取数组、即对内存中连续数据的读取操作，它们的操作类型相同并且各条指令之间并没有数据依赖关系，完全能够并行执行。实际上，Winograd 卷积算法的数据并行性较高，在执行过程中包含大量类似的程序段，除了浮点加载外，还存在浮点加法、浮点乘法、浮点存数等可并行化执行的指令。然而，在乱序 CPU 中，对并行度的挖掘受限于编译器、底层计算部件数量、流水线的发射宽度等因素，程序员难以灵活地通过编程进行调控。因此，有必要设计合适的向量化指令，以达到充分利用数据并行性的目的。

见表 2-1，在仿真过程中，经常出现因资源部件空间不足导致重命名和发射流水级堵塞的情况，而造成资源部件空间不足的一个重要原因在于指令无法正常从指令队列中发射到相应的执行单元。由图 2-3 中较为密集的黄线可知，该算法中存在大量具有数据依赖关系的浮点操作，虽然在乱序流水线中能够通过唤醒、旁路等方式尽可能减少因数据真相关带来的延迟，但过多的指令在等待操作数就绪，仍然会在 ROB、IQ、LSQ（Load Store Queue，访存队列）等部件中占据空间，进而造成资源不足的情况。若能将部分具有数据相关的指令进行合并，并以流水的形式在计算部件中执行，则能够在减少指令条数、资源部件

占用率的同时保证流水线的高效运行。

表 2-1　资源部件空间不足造成流水级堵塞次数

资源部件	重命名级	发射级
ROB	286	—
IQ	3177	603
LSQ	2214	549

综上，Winograd 卷积算法包含以下潜在优化点：

1. 包含大量重复的浮点加载，造成较大的执行延迟。
2. 未采用向量化处理方式，无法充分、灵活地挖掘数据并行性。
3. 大量数据依赖造成资源部件不足，导致流水级堵塞较多。

因此，本文考虑扩充一定数量的向量寄存器，存放 Winograd 卷积算法中使用到的操作数，并重点关注内循环中不变量的缓存策略，减少不必要的浮点访存请求和加载延迟。同时，基于扩充的向量寄存器设计相应的向量化指令，以充分挖掘数据并行性，并尝试将多条具有数据依赖的指令进行合并，减少因数据相关造成的资源不足和流水线堵塞。此外，本文还将为每条自定义指令编写相应的内联汇编函数，给程序员提供底层指令的接口，以便程序员灵活地调整指令顺序。

2.4　本章小结

本章主要介绍了 Winograd 卷积加速算法及其性能瓶颈分析。首先，从一维和二维两种场景入手，介绍了 Winograd 卷积算法的计算过程、加速原理以及在卷积神经网络中的应用。随后介绍了本文使用到的性能分析平台-开源指令级模拟器 Gem5，简要概述了其功能和使用方法，并详细介绍了其中乱序 O3-CPU 的整体架构。最后，使用模拟器 Gem5 对 Winograd 算法进行仿真，通过查看仿真输出和流水线可视化的方式，分析了该算法在 CPU 上的执行瓶颈，并总结得出三个潜在的优化点，提出了相应的优化思路。

第 3 章　卷积加速方案设计与实现

基于 2.3 节中的分析结果，本章首先将针对 Winograd 卷积算法在 CPU 上执行的三个性能瓶颈进行优化设计，提出基于 RISC-V 扩展指令的卷积加速方案。随后，通过软硬件协同设计的方法完成加速方案的实现，确保 Winograd 卷积算法在 CPU 上的高效执行。

3.1　硬件设计与实现

硬件设计包含三部分内容：硬件扩展、自定义指令设计，以及指令控制与执行方案设计。总体设计思路是：扩充向量寄存器，将 Winograd 算法最内层循环 $F(2\times 2,3\times 3)$中对数组的操作转换为硬件上的自定义向量化指令。同时，扩充计算部件和控制单元，并为扩展指令设计合适的控制方案，确保指令能够正确地在乱序流水线中执行。硬件实现具体需要修改 Gem5 乱序 O3-CPU 源码，在该架构上实现加速方案。

3.1.1　硬件扩展

在硬件层面，扩展了向量寄存器、指令控制单元和指令计算部件，下面分别进行具体介绍。

$F(2\times 2,3\times 3)$的输入包括经重排后 16 个输入矩阵中的元素、经转换后的 16 个卷积核权重、以及 4 个用于累加的输出矩阵中的元素，且源码实现上总是以 4 元素数组作为基本单位进行各种运算。为了尽可能减少硬件资源消耗，本文以一次循环迭代为基准，扩充了 11 个向量寄存器，每个向量寄存器的位宽均为 128bit，可存放 1 个由 4 个单精度浮点数（位宽为 32bit）组成的向量。其中 0 至 3 号寄存器用于存放 16 个卷积核权重，这部分寄存器的值在最内层循环中是不变量，无需重复加载。8 号寄存器用于存放输出矩阵的值，4 至 7 号寄存器用于存放输入矩阵的值，9 号和 10 号寄存器作为临时寄存器，存放计算过程中产生的中间变量。在计算过程中，4 号和 7 号寄存器也会存放一部分中间变量。

为了确保自定义指令在乱序流水线中正确执行，需要在处理器原有的发射逻辑上进行扩充，因此新增了相应的控制单元，其中包含两个关键的表：busyTable 和 ctrlTable。前者记录了每一种指令当前是否有对应的动态指令正在执行，用于处理不同循环迭代的动态指令之间的执行冲突问题；后者记录了每一个向量寄存器当前的状态值，这些状态值决定了使用到该寄存器的指令能否

在乱序流水线中发射，且会被执行完成的指令修改。

为了适应指令向量化、多层级的计算需求，扩充了相应的 3 层级计算单元，其中每一级分别是：4 个浮点加法部件、4 个浮点加法部件以及 4 个浮点乘法部件，各层级部件以及浮点加法、乘法计算部件内部均可流水执行。不同扩展指令的源和目的操作数都为 4，但需要完成的操作不同，因此在该计算单元中所需的执行延迟也不同。

3.1.2 指令编码与含义

见表 3-1，RISC-V 指令集为不同长度、不同指令子集的指令分配了各自的操作码，并用指令位域第 2 至 6 位进行指示。出于便利开发者自由扩展指令的目的，RISC-V 指令集设计者还在 32 位指令中为开发者预留了 4 个操作码，它们对应的指令是 cus0 至 cus3，即 custom0、custom1、custom2 和 custom3。在具体实现时，用户可结合特定处理器对指令集的实现情况和自身设计需求，选择合适的操作码进行扩展指令设计。

表 3-1 RISC-V 指令集指令操作码分配

inst[6:5] \ inst[4:2]	000	001	010	011	100	101	110	111
00	load	ldfp	cus0	mmem	opimm	auipc	opi32	48b
01	store	stfp	cus1	amo	op	lui	op32	64b
10	madd	msub	nmsub	nmad	op-fp	rsvd	cus2	48b
11	br	jalr	rsvd	jal	sys	rsvd	cus3	≥80b

在 Gem5 目前针对 RISC-V 指令集的实现中，4 个自定义操作码均未使用。本文选择 cus-1 对应的 0x0b（01010）作为扩展指令操作码，所有指令的编码见表 3-2，其中 rvd 位域包括 funct3[14:12]和 rd[11:7]。不同扩展指令之间主要通过比特位域 funct2 和 cvec 进行区分。比特位域 rs2、rs1 和 rd 分别代表使用到的源通用寄存器 1、源通用寄存器 2、目的通用寄存器，在本文中仅有访存指令使用到 rs1 作为访存地址，其余指令均未使用通用寄存器；cvec 位域除了用于隐含指示各条指令使用到的源和目的自定义向量寄存器号外，也用于区分 triadd 和 oacc 这两条指令；funtc3 通常情况下用于指示通用寄存器使用情况，在本文

中并未使用。在所设计的扩展指令中，ld_tilerow 和 wb_tile 是访存类指令，aamul、triadd 和 oacc 是计算类指令，下面结合指令的编码，详细介绍各条指令的具体含义和设计意义。

表 3-2　自定义指令编码

指令	cvec[31:27]	funct2[26:25]	rs2[24:20]	rs1[19:15]	rvd[14:7]
ld_tilerow	0-8	0	—	地址	—
aamul	0-3	1	—	—	—
triadd/oacc	0-2	2	—	—	—
wb_tile	8	3	—	地址	—

ld_tilerow 用于从内存中读取一个 4 浮点数向量，将其存放入相应的向量寄存器中。操作数 rs1 表示浮点数组在内存中的首地址，cvec 用于指示目的向量寄存器号，该指令的等价表达式为 mem[rs1]->vreg[cvec]（式中 mem 代表主存，vreg 代表扩充的向量寄存器，下同）。该指令的意义在于将后续计算所需的操作数存入自定义的寄存器中，且其中一部分操作数无需在内循环中重复加载，能够降低访存延迟。

wb_tile 指令用于将向量寄存器组中一个 4 浮点数向量存放入内存中，操作数 rs1 表示浮点数组在内存中的首地址，cvec 用于指示源向量寄存器号。在目前的设计中，8 号寄存器用于存放卷积计算输出结果，所以此处 cvec 固定为 8，该指令的等价表达式为 vreg[8]->mem[rs1]。

三种计算类指令 aamul、triadd 和 oacc 均为向量化指令，且合并了多个具有数据相关的指令，能够大大减少指令数量。向量的长度设计为 4，符合算法源程序特性，能够很好地挖掘数据并行性；多级计算的设计，则能够减少因数据依赖造成的资源不足、流水线堵塞的问题。

其中，指令 aamul 用于完成一个三层级浮点运算，源操作数为 3 个长度为 4 的浮点数向量：首先源操作数 1 和源操作数 2 进行向量浮点加减运算，得到一个中间向量，该中间向量内部 4 个浮点数相互之间进行浮点加减操作得到新的中间向量，最后该中间向量与源操作数 3 进行向量浮点乘操作，得到结果向量。cvec 的值与源和目的向量寄存器号的对应关系见表 3-3，该指令的等价表达式为:inner_cal(vreg[src1]+vreg[src2])*vreg[src3]->vreg[dest]（式中“+”、“*”分别

代表向量加法和向量乘法）。

表 3-3　aamul 指令中 cvec 的含义

cvec	源寄存器号 1	源寄存器号 2	源寄存器号 3	目的寄存器号
0	0	4	6	4
1	3	7	5	7
2	1	5	6	9
3	2	6	5	10

triadd 和 oacc 指令都完成一个两层级浮点运算，源操作数为 3 个 4 浮点数向量，它们共享了 funtc2=2 的位域，通过比特位域 cvec 进行区分。对于 triadd 指令，首先将源操作数 1 和源操作数 2 进行向量浮点加减运算，得到一个中间向量，该中间向量再与源操作数 3 进行向量浮点加减操作，得到结果向量。该指令的等价表达式为:vreg[src1]+vreg[src2]+vreg[src3]->vreg[dest]。oacc 指令完成的操作稍微复杂，需要将源操作数 3 分别与源操作数 2 和源操作数 1 进行带掩码的浮点加减操作，然后再对结果进行累加，该指令的等价表达式为:(vreg[src1]+vreg[src3])+(vreg[src2]+vreg[src3])->vreg[dest]（式中“+”代表带掩码的向量浮点加法）。这两种指令 cvec 的值与源和目的向量寄存器号的对应关系见表 3-4。

表 3-4　triadd 和 oacc 指令中 cvec 的含义

指令	cvec	源寄存器号 1	源寄存器号 2	源寄存器号 3	目的寄存器号
triadd	0	4	9	10	4
triadd	1	7	10	9	7
oacc	2	4	7	8	8

3.1.3　指令控制与执行

本文设计的扩展指令并不是在 CPU 外接的加速器中执行的，而是与指令集

中其他指令一样，都需要在乱序流水线中流经取指、译码、重命名、发射、执行、写回等流水级，并最终按序提交，完成对处理器状态的改变。为了将扩展指令适配在乱序流水线中，需要对流水线进行适当改动，增加相应的控制逻辑与执行单元，以提供对扩展指令的支持。

由于这些扩展指令使用到了自定义的向量寄存器，而这部分寄存器并不是通用寄存器，无法通过原有的重命名逻辑解决数据相关，因此需要使用 3.1.1 中提到的 ctrlTable 进行控制。出于精确异常回滚的需要，ctrlTable 需要设置两个，一个是“推测性”的，另一个是“架构性”的（与乱序流水线中的 RAT 类似）。前者用于控制指令的发射逻辑，后者用于在流水线冲刷时恢复前者。

具体来说，ctrlTable 为每个向量寄存器分配了一个值。自定义指令在发射前，需要查看其源寄存器和目的寄存器在推测性 ctrlTable 中对应的值是否满足指令发射条件；当指令流经写回级后，会设置推测性 ctrlTable 中相应向量寄存器对应的值，以完成对其他指令的唤醒操作；当指令提交后，会按照同样的设置策略更改架构性 ctrlTable 中相应的值。由于指令是按序提交，因此架构性 ctrlTable 中的值总是正确的，当乱序流水线中出现因分支预测错误和中断异常等因素导致的流水线冲刷的情况时，需要使用架构性 ctrlTable 对推测性 ctrlTable 进行恢复，保证后续指令的正确执行。具体每条指令对 ctrlTable 的使用见表 3-5。

表 3-5　扩展指令对 ctrlTable 的使用

指令	发射-检查	写回/提交-设置
ld_tilerow	ctrlTable[dest]=0	ctrlTable[dest]=1
aamul	ctrlTable[srcs]=1	ctrlTable[dest]=2
triadd	ctrlTable[srcs]=2	ctrlTable[dest]=4
oacc	ctrlTable[4/7]=4	ctrlTable[8]=2
wb_tile	ctrlTable[8]=2	ctrlTable[8]=0

此外，出于硬件资源消耗的考虑，目前设计中扩充的向量寄存器数目较少。在不同循环迭代中，同一种指令对应的不同动态指令使用的源和目的向量寄存器号是一致的。由于这些动态指令对应的发射条件一致，因此需要避免靠后迭代的指令错误地发射，影响到靠前迭代中后续指令的计算。此处需要使用 3.1.1 中提到的 busyTable 进行相应控制。

具体来说，该表为每一种指令分配 1bit 数，指示当前是否有该种指令对应

的动态指令在执行。指令在进入指令队列等待发射时是按序进行的，那么这些指令也会按序被检查是否满足发射条件。当某一条指令满足发射条件准备发射时，就将其在 busyTable 中对应的值置 1，指示后续同类指令不可发射；当指令流经写回级后，将其在 busyTable 中对应的值置 0，后续同类指令才有可能发射。在实现时，错误路径上的指令可能会将 busyTable 中相应的值置 1，若该指令在未流经写回阶段时就被流水线冲刷，busyTable 中相应的值将无法置 0，后续指令也就会被堵塞。因此，当一条扩展指令在流水线中被冲刷时，需要检查其是否已设置 busyTable，若是则需要将相应的值置 0。

在完成发射后，扩展指令会根据其指令类型来到对应的计算部件中。对于计算类指令，这些指令将以流水的形式在新扩充的计算部件中执行。Gem5 默认配置下，浮点加法和浮点乘法的执行延迟分别是 2 周期和 4 周期，由此即可计算得到计算类指令的执行延迟周期，见表 3-6。至于扩展的访存类指令，由于它们的执行过程与普通访存指令一致，区别仅在于访存的宽度，因此无需另外设置执行延迟，直接复用原有的访存功能部件即可。

表 3-6　计算类扩展指令延迟

指令	第一级	第二级	第三级	总延迟(cycles)
aamul	浮点加法	浮点加法	浮点乘法	8
triadd	浮点加法	浮点加法	—	4
oacc	浮点加法	浮点加法	—	4

3.1.4　加速方案实现

在完成扩展指令和相应加速方案的逻辑设计后，需要仔细阅读、修改 Gem5 源码以实现设计。Gem5 采用的是 CPU 与 ISA 解耦的代码架构，本文使用的 CPU 是乱序 O3-CPU 模型，ISA 是 RISC-V 32 位指令集，因此分别针对这两部分代码进行了修改。

ISA 相关部分包含指令译码、寄存器定义、特权指令等内容，代码主要由 DSL（Domain-Specific Language，领域特定语言）和 Python 编写。DSL 提供了高水平的抽象和简洁的语法，用于编写指令的译码、比特位域以及相应的执行定义等代码格式较为类似，容易使用模板进行编写的部分；Python 代码则是用

于实现 DSL 解析器，对 DSL 部分的代码进行解析。Gem5 源码经编译后，所有 DSL 代码都会被 Python 解析器解析为相应的 C 语言代码。在具体实现时，首先是扩充了 11 个自定义的向量寄存器，随后根据指令的具体编码和定义，按照 Gem5 中提供的模板修改了译码部分，使得译码器能够识别扩展指令，并明确指令在处理器中需要执行的操作。

CPU 部分的核心代码由 C++编写，包含各种 CPU 模型的实现，而各 CPU 模型的实现则包含不同流水级、不同功能部件等内容。本文使用的乱序 O3-CPU 模型涉及到的代码较为复杂，但实现时并没有对架构进行大的调整，主要修改了指令的发射、写回和唤醒这三个部分，其余部分复用原有架构即可。为了将扩展指令适配在原有的乱序流水线中，使用 C++语言实现了指令的控制方案，包括两个数据结构 ctrlTable 和 busyTable 的维护、指令发射条件的判断以及乱序流水线中边界情况的处理。对于扩展指令的执行部分，则是根据不同类型指令进行了不同处理：针对计算类指令，根据指令执行延迟对设计的计算单元进行了简单的配置；针对访存类指令，其地址计算、访存请求发送和接收部分与原有指令相同，仅对访存的宽度进行了修改。

3.2 软件设计

完成硬件层次的设计与实现后，需要在软件层次提供支持，以此提高本方案的可编程性和灵活性。该部分包含 3 个工作点：修改编译工具链使编译器能够识别自定义指令、为自定义指令编写相应内联汇编函数，以及使用内联函数改写 Winograd 卷积算法。

在硬件上完成扩展指令的设计与实现后，需要在代码中使用到扩展指令，目前常用的方法包括以下三种：第一种方式是在汇编程序中直接添加机器码；第二种方式是在源程序中利用.insn 模板进行编写；第三种方式需要借助 riscv-opcodes 工具对编译工具进行修改，修改完成后程序员可以在 C 程序中内嵌汇编，反汇编之后也可以看到相应的指令。综合来看，前两种方式实现过于繁琐，可编程性较差，因此本文选择第三种方式。

本文使用的编译工具是 RISC-V 交叉编译工具链 riscv-gnu-toolchain，其源码实现较为复杂，但本文不关注其编译器内部具体的实现，仅需完成扩展指令识别部分的工作：首先，根据扩展指令的编码，按照指定的格式编写相应模板，并利用 riscv-opcodes 工具生成扩展指令对应的宏；随后，将这些宏加入工具链 binutils 部分的头文件中，并在头文件对应的源程序中增加指令定义；最后，单独编译 binutils 部分的源码。

完成工具链的修改与编译后，编译器已经可以识别自定义的指令，此时程序员可以使用这些自定义指令进行编程。本文在此基础上，为每一条设计的自定义指令编写了相对应的内联汇编函数，将这些扩展指令封装到了 C 语言函数中，使得程序员可以直接在 C 语言源程序中像调用函数一样使用自定义的指令。一个扩展指令对应的完整内联汇编函数实例如下所示，下面对它的格式与含义进行简要介绍。

Code 1 The inline assembly function of the extended instruction ld_tile

```
static void ld_tile ( float* addr )
{
   __asm__    __volatile__ (
     " ldtileo    x0 , %0 , x0 "
      :
      : "r"    ( addr )
     ) ;
}
```

该函数对应的扩展指令是访存指令 ld_tile，这条指令使用到了一个通用寄存器作为访存地址，并使用向量寄存器作为目的地址。函数声明下方第一行的_asm_是 GCC 关键字 asm 的宏定义，violatile 指示编译器不可优化汇编指令。“ldtileo x0,%0,x0”前半部分是汇编指令名，后半部分用于指示通用寄存器使用情况，其中%0 表示分配寄存器给源操作数 rs1，前后两个 x0 表示不使用 rd 和 rs2。汇编指令下方是输出操作数，由于该指令直接写入扩充的寄存器组中，因此此处无需填写；最后一行指示输入操作数，“r”（addr）表示由编译器自动分配寄存器，存储操作数变量 addr。

完成编译工具链的修改以及所有设计的自定义指令相对应内联汇编的编写后，编写了一些简单的汇编程序，初步对单条指令和编译工具的正确性进行了验证。随后，使用这些内联汇编函数改写了 Winograd 卷积算法，改写后算法内循环部分见下方伪代码。

使用扩展后的 RISC-V 交叉编译工具对改写后的 Winograd 卷积算法进行编译，编译选项为-Ofast。改进后的算法经编译后，一共产生不到 40 条汇编指令，而原始算法经编译后一共产生了 170 条左右的汇编指令。向量化指令以及多层级计算指令的设计，使得指令条数大幅度减少。此外，内联汇编函数为程序员

提供了底层自定义指令的调用接口，为程序员调整指令顺序，进而调控计算与访存的平衡提供了便利。

Code 2 Pseudocode of the modified Winograd convolution algorithm

```
winograd_custom()
{
    ld_tile vec0,kernel_addr          // Load kernel into vector0-3
    ld_tile vec1,kernel_addr+4
    ...
    ld_tile vec4,ifmap_addr           // Load ifmap into vector4-7
    ld_tile vec5,ifmap_addr+4
    ld_tile vec8,ofmap_addr           // Load ofmap into vector8
    ...
    aamul    vec4,vec6,vec4           // Use custom calculate instruction
    aamul    vec7,vec5,vec7
    triadd   vec4,vec9,vec10,vec4
    oacc     vec8,vec4,vec7,vec8
    ...
    wb_tile vec8,ofmap_addr       // Writeback value in vector8 into ofmap
}
```

3.3 本章小结

本章主要介绍了基于扩展指令的加速方案的软硬件协同设计与实现。首先，在理论分析的基础上设计了相应的加速方案：扩充了 11 个向量寄存器，每个寄存器的位宽为 128bit，可存放 4 个单精度浮点数；设计了 5 条自定义扩展指令，包括指令编码和具体含义；扩充了指令的计算和控制单元，并设计了指令在乱序流水线中的控制与执行方案。完成设计后，在 Gem5 乱序 O3-CPU 上实现了加速方案。其次，修改了 RISC-V 交叉编译工具链，使得扩展指令能够被编译器识别。在此基础上，编写了扩展指令对应的内联汇编函数，并使用这些内联汇编函数改写了原始 Winograd 卷积算法。

第 4 章　仿真测试与分析

本章首先简要介绍加速方案的功能验证。随后借助开源深度学习框架 Darknet，在 Gem5 模拟器上运行仿真进行性能测试，以验证加速方案的有效性，并对性能测试结果进行分析。

4.1　功能验证

本文完成了扩展指令及加速方案的设计，在硬件层次对处理器的部分逻辑进行了调整，在软件层次使用内联函数改写了 Winograd 卷积算法内循环 $F(2\times 2,3\times 3)$的实现。因此需要进行功能验证，确保设计方案的正确性，功能验证示意如图 4-1 所示，具体验证过程如下：

首先，编写简单的汇编测试程序，验证单条扩展指令在 CPU 上执行的正确性。模拟器 Gem5 提供的仿真输出 trace 文件，详细记录了仿真过程中的事件序列，包括对内存、寄存器、缓存等系统组件的读写操作等。设计的扩展指令对 CPU 中内存和寄存器的读写操作见表 4-1，实验中通过查看各指令写回的值，与汇编程序中指定的值进行对比，验证了各条指令的正确性。通过查看输出的 trace 文件，还验证了指令对寄存器和内存的读取操作，确保了指令能够按照预定的设计去执行。

表 4-1　扩展指令读写情况

指令	内存	通用寄存器	扩展寄存器
ld_tilerow	读	读	写
wb_tile	写	读	读
计算类指令	—	—	写/读

其次，使用 3.2 节中改写后的 Winograd 算法，验证其中一次内循环 $F(2\times 2,3\times 3)$的正确性。该内循环的计算结果是一个尺寸为2×2的矩阵，实验中通过比对原始和改进算法的输出矩阵，验证了改写算法的正确性。这部分验证主要关注指令控制方案的正确性，即指令的发射、相互之间唤醒的逻辑与实现、以及对流水线边界情况的处理。在控制方案中，两个重要的控制表也是系统组件，因此同样可通过查看 trace 文件的方法，验证指令执行过程中对表的读

写操作是否符合控制方案的设计。

最后，运行实际的卷积层验证全套方案的正确性，包括不同循环迭代之间的控制，流水线冲刷等边界情况的处理等。本文改进的 Winograd 算法适用于卷积核大小为 3 的情况，实验中针对卷积层输入图像的大小和通道数、填充（padding），以及卷积核的个数这四个参数进行修改：填充分别设置为 0 和 1，输入图像的大小设置为 4 至 512 之间所有 4 的倍数，输入图像通道数、卷积核个数分别设置为 3、16、32、64、128、512，总计共有 9216 组参数组合。设置输入图像各像素和卷积核权重为随机值，在不同参数组合下进行仿真。经验证，原始和改进两种方案下卷积层输出矩阵的值在所有测试组中均相同，加速方案运行卷积层的正确性得到了进一步保证。

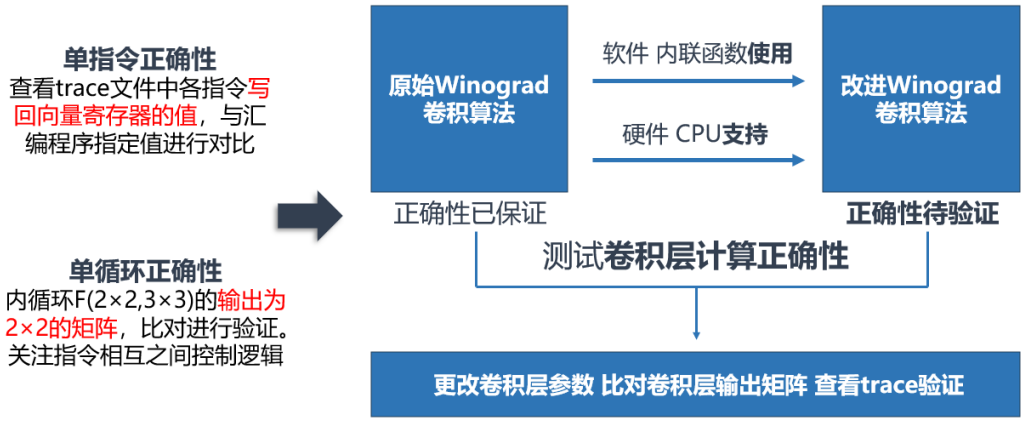

图 4-1　功能验证示意图

经以上功能验证后，加速方案的正确性基本能够得到保证。在 4.2 节性能测试中，还将介绍加速方案在整个卷积神经网络中的使用，卷积神经网络功能正确，是对加速方案正确性更充分的验证。

4.2　性能测试

选择开源轻量级深度学习框架 Darknet 进行性能测试，该框架提供了 C 语言版本的支持，并提供了多种深度学习模型的网络层配置、预训练权重，仅需简单配置即可运行各种神经网络的推理任务。

4.2.1 实验对象与步骤

Darknet-tiny 是 Darknet 框架中提供的一种用于图像分类的卷积神经网络，本节选择其中卷积核大小为 3 的卷积层作为实验对象。Darknet-tiny 由 22 层网

络组成，具有较少的网络层和参数。该网络包含 16 个卷积层，它们的参数数据见表 4-2，表中 IH、IW、IC 分别表示输入图像的高、宽和通道数，KH、KW、OC 分别表示卷积核的高、宽与个数，卷积核的通道数与输入图像的通道数保持一致。在所有卷积层中，一共有 8 个卷积核大小为 3 的卷积层，它们分别是 C1、C2、C4、C6、C8、C10、C12 和 C14，其中 C4 和 C6、C8 和 C10、C12 和 C14 这三组卷积层的参数数据相同，因此实际上一共有 5 种参数差异较大的卷积层。

表 4-2　Darknet-tiny 卷积层参数数据

卷积层	输入图像规模(IH,IW,IC)	卷积核规模(KH,KW,OC)
C1	224,224,3	3,3,16
C2	112,112,16	3,3,32
C3	56,56,32	1,1,16
C4	56,56,16	3,3,128
C5	56,56,128	1,1,16
C6	56,56,16	3,3,128
C7	28,28,128	1,1,32
C8	28,28,32	3,3,256
C9	28,28,256	1,1,32
C10	28,28,32	3,3,256
C11	14,14,256	1,1,64
C12	14,14,64	3,3,512
C13	14,14,512	1,1,64
C14	14,14,64	3,3,512
C15	14,14,512	1,1,128
C16	14,14,128	1,1,1000

性能测试实验中使用了两种计算方案，将原始方案和加速方案的仿真时间进行对比，实验的具体过程如下：

在框架原始 C 语言实现中，仅使用 im2col 算法进行卷积层前向推理计算。实验原始方案中，选取神经网络中 8 个卷积核大小为 3 的卷积层，将其前向推理算法更改为原始的 Winograd 算法，即采用 im2col 算法+原始 Winograd 算法的方式运行整个卷积神经网络。

在加速方案中，将原始 Winograd 算法替换为 2.4 节中经改进后的 Winograd 算法，即采用 im2col 算法+改进 Winograd 算法的方式运行整个卷积神经网络。改进后的算法使用到了扩展指令，由于在 CPU 上做出了相应支持，因此应起到加速的效果。

使用 3.2 节中修改后的 RISC-V 交叉编译工具对程序进行编译，编译选项设置为-Ofast，得到对应的二进制文件。随后编写 Gem5 仿真脚本，指定 CPU 为 Gem5 中经修改后可支持扩展指令的 RISC-V 乱序 O3-CPU 模型，各流水级的宽度均设置为 4，一级 cache 的大小设置为 16kB，二级 cache 的大小设置为 256kB，并将运行负载设置为编译好的二进制文件。

对修改源码后的 Gem5 进行编译，使用 Gem5.fast（快速仿真版本，无调试信息）运行仿真，统计两种方案下各卷积层前向推理和整个卷积神经网络的计算时间，进行对比，以验证加速方案的有效性。

4.2.2 实验结果分析

按照 4.2.1 中的实验部署，进行实验并整理数据。通过查看仿真时间可知，使用加速方案运行整个卷积神经网络的时间为使用原始方案的 77%，且程序输出表明结果无精度损失，图片分类效果与原始方案一致。其中经修改的卷积层运行时间如图 4-2 所示，下面进行分析。

图 4-2 中，底侧横轴表示 8 个经修改的卷积层，左侧纵轴指示在 Gem5 上运行仿真的周期数，用于衡量卷积计算的执行时间。可以发现，在使用加速方案后，各卷积层计算均取得了一定的加速效果。其中，C12 与 C14 两层的优化效果最佳，平均加速比约为 1.98；C2、C4、C6、C8 和 C10 这几层的优化效果较好，平均加速比约为 1.34；C1 层的加速比最低，约为 1.21。结合卷积层参数信息和各种仿真输出信息进行分析，可以发现以下几个关键点：

经计算，C1 层的内循环$F(2\times 2,3\times 3)$迭代次数为 602112，而其余 7 个卷积层的迭代次数均为 1605632。从仿真时间上看，内循环迭代次数较少的 C1 层，其加速效果比其他卷积层的要差。原因在于本文正是针对内循环$F(2\times 2,3\times 3)$

设计的加速方案，内循环迭代次数越多，使用扩展指令的加速收益越大，这也验证了加速方案的有效性。

除 C1 层外，其余卷积层内循环迭代次数相同。其中，C2、C4 和 C6、C8 和 C10 这几组卷积层的仿真时间和加速比十分接近，而 C12 和 C14 层的仿真时间加速比却远高于其他卷积层，这与分支预测有较大的关系：使用加速方案时，C12 和 C14 层的平均失误率为 1.77%，其他层的平均失误率为 5.85%；使用原始方案时，这两项数据分别是 0.66%和 0.25%。显然，在加速方案中分支预测的变化更大，对流水线的影响也更大。分支预测失误率大幅度降低，也使得程序的仿真时间减少，进而提高了加速比。

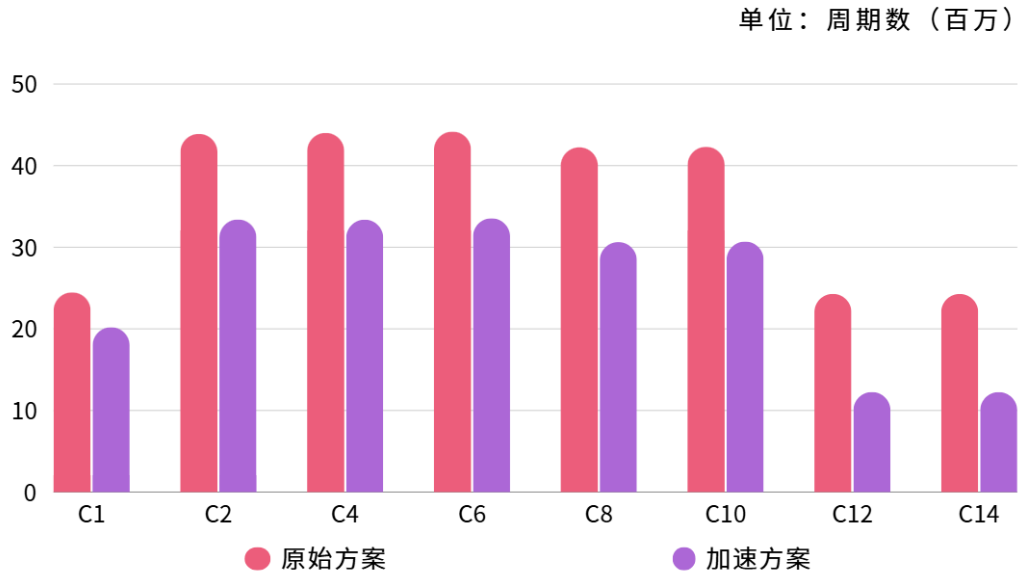

图 4-2　卷积层运行时间对比

表 4-3 展示了两种方案下仿真输出的执行指令数和访存信息，表中数据均为 8 个卷积层仿真的累加结果，且做了近似处理。整体来看，使用加速方案后，执行的动态指令数、访存请求数和访存延迟都大大降低。

执行指令数减少的主要原因在于使用了向量化扩展指令，即采用了 SIMD 的处理方式提升了操作数的宽度。此外，所设计的自定义指令中的所有计算指令都包含多层级操作，即对原始程序中多个具有数据依赖的操作进行了合并，有效地减少了对资源部件的占用。虽然与指令集原有的普通指令相比，这些计算指令的复杂度较高，单条指令的执行延迟较大，但这些指令均能够在设计的计算单元中以流水的形式执行，所以它们的执行延迟能够被隐藏，在绝大多数情况下，不会因此产生负面影响。

访存请求数和访存延迟的减少，得益于所扩充的缓冲寄存器和向量化访存

方式。在加速方案中，向量寄存器中用于存放卷积核权重的寄存器避免了重复的浮点加载操作。此外，访存的宽度增加为原来的 4 倍，这也进一步导致了访存请求数的减少。虽然在加速方案中，cache 命中率有所降低，但由于其访存请求数仅为原始方案中的 10%左右，其访存延迟周期仍然大大减少。由 2.3 节中的分析可知，访存延迟是 Winograd 卷积算法在乱序 CPU 上执行的瓶颈之一，因此访存延迟的减少也使得卷积层的仿真时间大幅度下降。

表 4-3　执行指令数和访存信息

方案	执行指令数（百万次）	访存请求数（百万次）	访存延迟（百万周期）
原始方案	2166.93	157.12	705161.84
加速方案	574.19	15.93	268634.58

综合来看，加速方案能够达到较好的优化效果，整体加速效果受到内循环迭代次数、动态执行指令数和访存延迟等因素的影响，实验分析结果基本与加速方案设计的思路相符。

4.3　本章小结

本章介绍了对加速方案进行功能验证和性能测试的过程与结果。

在功能验证中，通过执行汇编程序验证了扩展指令和内循环$F(2\times 2,3\times 3)$的正确性，通过执行不同组卷积计算验证了改写后 Winograd 卷积算法的正确性，确保软硬件协同设计方案能够正确执行。

在性能测试中，使用开源深度学习框架 Darknet 提供的卷积神经网络 Darknet-tiny 进行验证。实验结果表明，使用加速方案运行卷积神经网络 Darknet-tiny 的前向推理任务，能够在保证结果正确性的前提下，将运行时间缩短 23%。针对网络中单个卷积层的加速，其性能提升与具体卷积层的参数有关，最高能达到 1.98 的加速比。

综合来看，本文提出的基于 RISC-V 扩展指令的卷积加速方案有效地解决了 Winograd 卷积算法在 CPU 上的执行瓶颈，加速了卷积运算，取得了明显的整体性能效果提升。

结　论

本文主要介绍了基于 RISC-V 扩展指令的卷积加速设计。首先，通过在开源模拟器 Gem5 的 RISC-V 乱序 CPU 上运行 Winograd 卷积算法，对卷积计算在 CPU 上的执行瓶颈进行了分析；其次，基于理论分析设计了 RISC-V 扩展指令的及相应卷积加速方案，并在软硬件方面做出了相应支持；最后，通过仿真测试，对加速方案进行了功能验证和性能评估。

本论文的主要创新性工作包括：

1. 使用开源指令级模拟器 Gem5 对 Winograd 卷积算法在乱序处理器上的执行进行了仿真模拟，从底层指令、处理器流水线执行的层次分析其动态执行瓶颈，并提出了 3 个潜在优化点。

2. 基于理论分析，设计了相应的加速方案：扩充了 11 个位宽为 128bit 的向量寄存器，每个寄存器可存放 4 个单精度浮点数；设计了 5 条 RISC-V 向量化扩展指令和指令在乱序流水线中的控制方案。设计的扩展指令包含计算和访存类指令，其中计算类指令合并了具有数据相关的多条指令。

3. 在 Gem5 的 RISC-V 乱序 O3-CPU 架构中实现了扩展指令及相应加速方案。实现中，所有的扩展指令与其他指令一样在乱序流水线中流动，无需另设加速器运行扩展指令。

4. 扩展了 RISC-V 交叉编译工具链以识别设计的扩展指令，并为这些指令编写了对应的内联汇编函数，供程序员使用。使用内联汇编函数，改写了原始 Winograd 卷积算法。改写后的算法经编译后，产生的汇编指令数量大大减少，约为原始算法编译后的 23%。

5. 经实验验证，所设计的加速方案能够降低单个卷积层和整个卷积神经网络的执行延迟。针对卷积神经网络 Darknet-tiny，使用加速方案运行能够在保证结果正确性的前提下，将运行时间缩短约 23%，其中对网络中单个卷积层的加速，最高能达到约 1.98 的加速比。

今后还应在以下几个方面继续深入研究：

1. 本文针对自定义指令所扩充的向量寄存器的数量并非是最优的，且指令控制方案会随着向量寄存器数量变化而变化。在以后的研究中，可以探索通用的控制方案，将指令控制方案和向量寄存器个数解耦，并进行设计空间探索，以确定最合适的向量寄存器个数。

2. 本文仅针对 Winograd 卷积算法进行了分析，在以后的研究中，可以针对 im2col、FFT 等其他常用的卷积加速算法进行分析，设计出相应的扩展指令和加速方案，并探索如何合理搭配这些卷积算法，高效计算卷积神经网络。

参考文献

[1] Krizhevsky A, Sutskever I, Hinton G E. Imagenet Classification with Deep Convolutional Neural Networks[J]. Communications of the ACM，2017，60（6）：84-90.

[2] Wang Z, Zheng X, Li D, et al. A VGGNet-like Approach for Classifying and Segmenting Coal Dust Particles with Overlapping Regions[J]. Computers in Industry，2021，132：103506.

[3] Al-Qizwini M, Barjasteh I, Al-Qassab H, et al. Deep Learning Algorithm for Autonomous Driving Using Googlenet[C]//IEEE Intelligent Vehicles Symposium（IV），2017：89-96.

[4] Fu L, Feng Y, Majeed Y, et al. Kiwifruit Detection in Field Images Using Faster R-CNN with ZFNet[J]. IFAC-PapersOnLine，2018，51（17）：45-50.

[5] Redmon J, Divvala S, Girshick R, et al. You only Look Once: Unified, Real-Time Object Detection[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition，2016：779-788.

[6] 张港红. 农业物联网专用处理器芯片设计研究[D]. 北京: 中国农业大学，2018，12-25.

[7] 张雷. 基于 RISC-V 架构的物联网节点处理器研究与设计[D]. 南京: 南京航空航天大学，2020，6-12.

[8] Yu-Hsin Chen, Tushar Krishna, Joel S. Emeret, et al. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks[J]. IEEE Journal of Solid-State Circuits，2017，52（1）：127-138.

[9] Cui Enfang, Tianzheng Li and Qian Wei. RISC-V Instruction Set Architecture Extensions: A Survey[J]. IEEE Access，2023（11）：24696-24711.

[10] 雷思磊. RISC-V 架构的开源处理器及 SoC 研究综述[J]. 单片机与嵌入式系统应用，2017（2）：56-60.

[11] Waterman A, Lee Y, Avizienis R, et al. The RISC-V instruction set[C]//2013 IEEE Hot Chips 25 Symposium（HCS），2013：9-22.

[12] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, et al. Classification of Lightweight Block Ciphers for Specific Processor Accelerated Implementations[C]//IEEE International Conference on Electronics, Circuits and Systems（ICECS），2019：747-750.

[13] Bastian Koppelmann, Peer Adelt, Wolfgang Mueller, et al. RISC-V Extensions

for Bit Manipulation Instructions[C]//International Symposium on Power and Timing Modeling, Optimization and Simulation（PATMOS），2019：41-48.

[14] Hela Belhadj Amor, Carolynn Bernier, Zdenek Prikryl. A RISC-V ISA Extension for Ultra-Low Power IoT Wireless Signal Processing[J]. IEEE Transactions on Computers，2022，71（4）：766-778.

[15] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traberet al. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems，2017，25（10）：2700-2713.

[16] Wenkai Tang, Peiyong Zhang. GPGCN: A General-Purpose Graph Convolution Neural Network Accelerator Based on RISC-V ISA Extension[J]. Electronics，2022，11（22）：3833.

[17] Shuenn-Yuh Lee, Yi-Wen Hung, Yao-Tse Changet al. RISC-V CNN Coprocessor for Real-Time Epilepsy Detection in Wearable Application[J]. IEEE Transactions on Biomedical Circuits and Systems，2021，15（4）：679-691.

[18] Garofalo A, Tagliavini G, Conti F, et al. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors through ISA Extensions[C]//EDA Consortium, 2020：186-191.

[19] Lecun Y, Bottou L, Bengio Y, et al. Gradient-based Learning Applied to Document Recognition[J/OL]. Proceedings of the IEEE，1998：2278-2324.

[20] Krizhevsky A, Sutskever I, Hinton G E. ImageNet Classification with Deep Convolutional Neural Networks[J/OL]. Communications of the ACM，2017：84-90.

[21] J. Cong and B. Xiao. Minimizing Computation in Convolutional Neural Networks[C]//International Conference on Artificial Neural Networks. Springer，2014：281–290.

[22] Parashar A, Rhu M, Mukkara A, et al. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks[C]//ACM，2017：27-40.

[23] Chen Tianshi, Du Zidong, Sun Ninghui,et al. 2014. Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning[J]. ACM SIGARCH Computer Architecture News，2014：269–284.

[24] Chen Y, Luo T, Liu S, et al. DaDianNao: A Machine-Learning Supercomputer[C]//IEEE Computer Society，2014：609-622.

[25] N. P. Jouppi et al. In-datacenter Performance Analysis of a Tensor Processing

Unit[C]//ACM/IEEE 44th Annual International Symposium on Computer Architecture（ISCA），2017：1-12.

[26] Chang J, Kang S. Optimizing FPGA-based Convolutional Neural Networks Accelerator for Image Super-Resolution[C]//Asia and South Pacific Design Automation Conference（ASP-DAC），2018：343-348

[27] Yu X, Yang Z, Peng L, et al. CNN Specific ISA Extensions Based on RISC-V Processors[C]//International Conference on Circuits, Systems and Simulation（ICCSS），2022：116-120.

[28] S. Wang et al. Optimizing CNN Computation Using RISC-V Custom Instruction Sets for Edge Platforms[J]. IEEE Transactions on Computers，2024，73（5）： 1371-1384.

[29] Li Z, Hu W, Chen S. Design and Implementation of CNN Custom Processor Based on RISC-V Architecture[C]//IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems（HPCC/SmartCity/DSS），2019：1945-1950.

[30] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldiet al. A Lightweight Posit Processing Unit for RISC-V Processors in Deep Neural Network Applications[J]. IEEE transactions on emerging topics in computing，2022，10（4）：1898-1908.

[31] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition，2016：4013–4021.

哈尔滨工业大学本科毕业论文（设计）
原创性声明和使用权限

本科毕业论文（设计）原创性声明

本人郑重声明：此处所提交的本科毕业论文（设计）《基于 RISC-V 扩展指令的 AI 卷积加速设计》，是本人在导师指导下，在哈尔滨工业大学攻读学士学位期间独立进行研究工作所取得的成果，且毕业论文（设计）中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本毕业论文（设计）的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：罗腾　　日期：2024 年　5 月　24 日

本科毕业论文（设计）使用权限

本科毕业论文（设计）是本科生在哈尔滨工业大学攻读学士学位期间完成的成果，知识产权归属哈尔滨工业大学。本科毕业论文（设计）的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存本科生上交的毕业论文（设计），并向有关部门报送本科毕业论文（设计）；（2）根据需要，学校可以将本科毕业论文（设计）部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）本科生毕业后发表与此毕业论文（设计）研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉本科毕业论文（设计）的使用权限，并将遵守有关规定。

作者签名：罗腾　　日期：2024 年　5 月　24 日

导师签名：　　日期：2024 年　5 月　24 日

致　谢

本人于 2020 年 9 月进入哈尔滨工业大学，开启了近四年的大学本科生活。在这四年时光里，我学习了很多知识，找到了自己喜欢的研究方向。同时，我也结交了很多良师益友，他们在学习和生活中都给予了我很大的帮助，让我能够顺利地度过这四年时光。

本次毕业设计的顺利完成离不开各位老师、师兄和同门的帮助。首先我要感谢翁睿老师，他是我在大三学年计算机网络这门课的老师，也是我毕业设计的校内指导老师。翁老师对待教学和科研非常严谨，对同学们非常亲切友好，他在毕设期间一直关注着我的毕设进展，并给我的论文提出了宝贵的修改意见。

我还要感谢中国科学院计算技术研究所的李文明老师、范志华博士和夏腾飞同门。李老师在选题建议和设计思路方面给予了我很大帮助，范师兄在文献查阅、论文润色等方面给予了我很大支持。夏腾飞同学思路灵活，也给我的设计方案提出了一些重要的建议。没有他们，我的毕业设计工作将变得非常困难。

当然，我也要感谢舒燕君老师和胡光辉同学。没有他们，或许我也不会在本科期间找到自己喜欢的研究方向。舒老师是我在大三学年计算机组成原理、计算机组织与体系结构这两门课程的任课老师，她把这两门复杂的课讲得通俗易懂，并且鼓励、指导我们参加“龙芯杯”比赛。胡光辉同学是我在“龙芯杯”比赛中的队长，去年我们利用暑假的时间完成了一个 MIPS 乱序 CPU 的设计与实现，并在比赛中取得了好成绩。胡同学具有坚韧不拔的毅力，且工程能力极强，给予了我很大的帮助。

除了学习之外，我在大学期间最常参与的就是各类体育活动。首先，我要感谢史一帆、赵以诚、张元瑞同学和刘培源学长，与他们切磋乒乓球技术，让我收获了快乐。其次，我还要感谢未来技术学院足球队的全体成员，我们在场上共同拼搏，共同承担失利的痛苦、享受赢球的喜悦。其中，我与刘宇晟、李灏然、王新宇、吴梓昊和赵方旭等同学从大一学年的新生杯开始，一直并肩作战，这是无比珍贵的情谊和回忆。

此外，我要感谢我的父母和姐姐，他们在生活、精神方面给予了我很大的支持，家人永远是我最强大的精神支柱。

最后，要感谢自己没有浪费本科期间的四年时光。毕设和本科结束之后，学习生活将翻开新的篇章，新的挑战也即将来临，希望我还能保持本科期间的良好状态，健康、充实、快乐地度过研究生阶段。