

西安交通大学

硕士学位论文

基于 GEMM 的卷积神经网络加速器软硬件设计研究

学位申请人：秦海鹏

指导教师：梁峰 教授

学科名称：电子科学与技术

2024 年 05 月

Research on Software and Hardware Design of Convolutional Neural Network Accelerator Based on GEMM

A thesis submitted to
Xi'an Jiaotong University
in partial fulfillment of the requirements
for the degree of
Master of Engineering

By
Haipeng Qin
Supervisor: Prof. Feng Liang
Electronic Science and Technology
May 2024

硕士学位论文答辩委员会

基于 GEMM 的卷积神经网络加速器软硬件设计研究

答辩人：秦海鹏

答辩委员会委员：

西安交通大学 教授 梅魁志：梅魁志 (注：主席)

西安交通大学副教授 杨 晨：杨晨

西安交通大学副教授 程 军：程军

西安交通大学 教授 张 斌：张斌

西安交通大学 教授 梁 峰：梁峰

答辩时间：2024 年 05 月 18 日

答辩地点：创新港泓理楼 4-3202

摘 要

深度学习技术的飞速发展给人们的生活带来更多便捷的居住体验和更加智能的生活模式。数据的爆炸式增长成为常态，云计算、边缘计算、分布式计算等范式的融合和协同为人工智能技术的落地提供了强有力的支持。针对 CPU 计算密度低、GPU 运行功耗高等问题，领域特定架构（Domain Specific Architecture, DSA）芯片随着半导体工艺的成熟逐渐成为现阶段的主流解决方案。本文提出了一个针对 CNN 硬件加速的异构方案，主要贡献如下：

1) 采用 Chisel 敏捷开发思想，设计了以脉动阵列为核心的 CNN 硬件加速器。整体采用权重固定的数据流格式，增加权重预加载数据通路以提高 PE 的利用率。在数据流优化方面采用算子融合、重量化、大尺寸特征图分块、Layout 变换等方法有效降低了数据访存延时和硬件计算时间。

2) 开发硬件适配的 AI 推理引擎，实现了神经网络算法的端到端部署。在软件前端工具链中，采用 INT8 训练后量化（Post-training Quantization, PTQ），并根据算子融合及重量化等硬件需求进行计算图优化。同时，在边缘端裸核运行环境中针对驱动层、加速库和应用层代码进行层层解耦和抽象，有效降低了二次开发难度。

3) 基于 Verilator 开源仿真器，搭建了软硬件联合仿真框架。该框架支持硬件模块级、算子级、网络级的轻量化仿真，并提供故障定位、数据自动校验、批量测试、性能分析等功能。

最后，本文在 FPGA 平台中搭建了 SoC 原型系统和视频流实时处理系统以验证软硬件方案的正确性。在与 ARM Cortex-A53 CPU@1GHz 的对比实验中，NPU@200MHz 在典型 CNN 推理中实现了约 1000 倍的性能提升。采用 SMIC 28nm 工艺库并保留 20% 时序裕量进行 RTL 综合，NPU 最大频率为 666MHz，可提供 2.728TOPS@INT8 的峰值算力。

关 键 词：卷积神经网络；领域特定架构；GEMM；端到端部署

论文类型：应用研究

ABSTRACT

The rapid development of deep learning technology has led to more convenient living experiences and smarter lifestyles. Accompanied by the explosive growth of data, the integration of cloud computing, edge computing, distributed computing, and other paradigms has been strongly supported for the effective implementation of artificial intelligence technology. In response to issues such as the low computing density of CPUs and the high operating power consumption of GPUs, Domain Specific Architecture (DSA) chips have been increasingly recognized as the mainstream solution as semiconductor processes continue to mature. A heterogeneous solution, suitable for CNN hardware acceleration, is proposed in this thesis. The main contributions are outlined as follows:

- 1) Utilizing Chisel agile development, a CNN hardware accelerator is designed, featuring a systolic array. The overall data flow format with weight static is employed, and a weight preloading data path is added to enhance the utilization of PE. In terms of data flow optimization, methods such as operator fusion, re-quantization, large-size feature map blocking, and layout transformation is employed to effectively reduce data access delays and minimize hardware computing time.
- 2) Using hardware-adapted AI inference engine, the end-to-end deployment of neural network algorithms is realized. In the software front-end tool chain, INT8 post-training quantization (PTQ) is employed, and the calculation graph is optimized according to hardware requirements such as operator fusion and re-quantization. Simultaneously, the driver layer, acceleration library, and application layer code are decoupled and abstracted layer by layer in the edge-end bare-core operating environment, effectively reducing the difficulty of secondary development.
- 3) Based on the Verilator open source simulator, a software and hardware joint simulation framework is constructed. The framework is capable of supporting lightweight simulation at the hardware module level, operator level, and network level. Additionally, functions such as fault location, automatic data verification, batch testing, and performance analysis are provided.

Finally, a SoC prototype system and a video stream real-time processing system are constructed on the FPGA platform to verify the correctness of the software and hardware solutions. Compared with ARM Cortex-A53 CPU@1GHz, NPU@200MHz achieves approximately 1000 times performance improvement in typical CNN inference. Utilizing the SMIC 28nm target library and retaining 20% timing margin for RTL synthesis, a maximum frequency of 666MHz is attained, providing a peak computing power of 2.728TOPS@INT8.

KEY WORDS: CNN; DSA; GEMM; End-to-end deployment

TYPE OF THESIS: Application Research

目 录

| | |
|------------------------------------|-----|
| 摘 要 | I |
| ABSTRACT | III |
| 1 绪论 | 1 |
| 1.1 研究背景和意义 | 1 |
| 1.2 国内外研究现状 | 2 |
| 1.3 本文主要研究内容 | 6 |
| 1.4 章节安排 | 7 |
| 2 神经网络硬件加速设计优化 | 9 |
| 2.1 卷积神经网络基础 | 9 |
| 2.1.1 卷积神经网络的发展 | 9 |
| 2.1.2 卷积计算并行维度的探索 | 10 |
| 2.2 卷积加速算法 | 12 |
| 2.3 模型压缩 | 15 |
| 2.3.1 模型剪枝 | 16 |
| 2.3.2 模型量化 | 16 |
| 2.4 本章小结 | 18 |
| 3 硬件加速器架构设计 | 19 |
| 3.1 硬件加速器顶层设计 | 19 |
| 3.1.1 加速器接口 | 19 |
| 3.1.2 加速器数据流 | 21 |
| 3.2 核心计算单元设计 | 22 |
| 3.2.1 脉动阵列设计 | 22 |
| 3.2.2 向量 ALU 设计 | 25 |
| 3.3 im2col 单元设计 | 28 |
| 3.3.1 im2col 硬件实现方案 | 28 |
| 3.3.2 Layout 模块设计 | 29 |
| 3.3.3 Padding 和 Reorder 模块设计 | 30 |
| 3.4 大尺寸特征图分块策略 | 31 |
| 3.5 算子融合 | 33 |
| 3.6 重量化策略 | 34 |
| 3.6.1 重量化原理 | 34 |
| 3.6.2 重量化硬件设计 | 36 |
| 3.7 本章小结 | 38 |

| | |
|-----------------------------------|----|
| 4 模型部署工具链及验证平台设计 | 39 |
| 4.1 面向深度学习的技术路线 | 39 |
| 4.2 硬件适配的 AI 推理引擎 | 40 |
| 4.2.1 NCNN 推理引擎 | 40 |
| 4.2.2 Mat 数据结构 | 42 |
| 4.2.3 网络参数文件 | 43 |
| 4.3 神经网络端到端部署工具链 | 45 |
| 4.4 边缘端软件设计 | 47 |
| 4.5 基于 Verilator 的软硬件协同仿真框架 | 49 |
| 4.6 本章小结 | 51 |
| 5 实验测试与性能分析 | 52 |
| 5.1 Verilator 仿真性能分析 | 52 |
| 5.2 DC 综合结果 | 54 |
| 5.3 FPGA 原型验证 | 55 |
| 5.3.1 SoC 系统搭建 | 55 |
| 5.3.2 视频流实时处理系统搭建 | 59 |
| 5.3.3 NPU 推理精度分析 | 60 |
| 5.3.4 NPU 加速性能分析 | 62 |
| 5.4 本章小结 | 64 |
| 6 总结与展望 | 66 |
| 6.1 总结 | 66 |
| 6.2 展望 | 67 |
| 致 谢 | 68 |
| 参考文献 | 69 |
| 攻读学位期间取得的研究成果 | 72 |
| 答辩委员会会议决议 | 73 |
| 常规评阅人名单 | 74 |
| 声明 | |

CONTENTS

| | |
|--|-----|
| ABSTRACT (Chinese)..... | I |
| ABSTRACT (English) | III |
| 1 Introductions..... | 1 |
| 1.1 Background and Significance..... | 1 |
| 1.2 Related Researches..... | 2 |
| 1.3 Main Content..... | 6 |
| 1.4 Structure of the Thesis..... | 7 |
| 2 Neural Network Hardware Acceleration Optimization | 9 |
| 2.1 CNN Foundation | 9 |
| 2.1.1 CNN Development | 9 |
| 2.1.2 Exploration of Parallel Dimensions | 10 |
| 2.2 Convolution Acceleration Algorithm | 12 |
| 2.3 Model Compression | 15 |
| 2.3.1 Pruning | 16 |
| 2.3.2 Quantization | 16 |
| 2.4 Brief Summary | 18 |
| 3 Hardware Accelerator Architecture Design..... | 19 |
| 3.1 Top-level Design | 19 |
| 3.1.1 Interfaces | 19 |
| 3.1.2 Data Flow | 21 |
| 3.2 Computing Engine..... | 22 |
| 3.2.1 Systolic Array | 22 |
| 3.2.2 Vector ALU..... | 25 |
| 3.3 Im2col Architecture..... | 28 |
| 3.3.1 Hardware Implementation..... | 28 |
| 3.3.2 Layout..... | 29 |
| 3.3.3 Padding and Reorder | 30 |
| 3.4 Large-size Feature Map Blocking Strategy..... | 31 |
| 3.5 Operator Fusion..... | 33 |
| 3.6 Re-quantization | 34 |
| 3.6.1 Principle..... | 34 |
| 3.6.2 Hardware Implementation..... | 36 |
| 3.7 Brief Summary | 38 |
| 4 Model Deployment Tool Chain and Verification Platform Design | 39 |
| 4.1 Technical Route for Deep Learning | 39 |
| 4.2 Hardware-adapted AI Inference Engine | 40 |
| 4.2.1 NCNN Inference Engine | 40 |
| 4.2.2 Mat Data Structure | 42 |

| | |
|---|----|
| 4.2.3 Network Parameter File..... | 43 |
| 4.3 End-to-end Deployment Toolchain | 45 |
| 4.4 Edge Devices Software Design | 47 |
| 4.5 Co-simulation Framework Based on Verilator | 49 |
| 4.6 Brief Summary | 51 |
| 5 Experimental Testing and Performance Analysis | 52 |
| 5.1 Verilator Simulation Performance Analysis | 52 |
| 5.2 DC Synthesis Results | 54 |
| 5.3 FPGA-based Prototype System | 55 |
| 5.3.1 SoC System | 55 |
| 5.3.2 Video Stream Real-time Processing System | 59 |
| 5.3.3 NPU Inference Accuracy Analysis | 60 |
| 5.3.4 NPU Acceleration Performance Analysis..... | 62 |
| 5.4 Brief Summary | 64 |
| 6 Conclusions and Suggestions | 66 |
| 6.1 Conclusions | 66 |
| 6.2 Suggestions..... | 67 |
| Acknowledgements | 68 |
| References | 69 |
| Achievements | 72 |
| Decision of Defense Committee..... | 73 |
| General Reviewers List | 74 |
| Declarations | |

1 绪论

1.1 研究背景和意义

随着现代科学技术的迅速发展和人民精神需求的日益增长，以深度学习为代表的人工智能技术在各行各业中发挥着越来越大的作用。目标检测、图像分割、语音识别、智能驾驶等技术已经深入千家万户，其不仅带来了更智能、更便捷的生活体验，还推动了科学、医疗、制造、农业等领域的创新。与此同时，在人工智能的发展潮流中，也涌现了各种新型技术和应用，如 ChatGPT、Sora 以及数字人技术等。这些技术不仅在人机交互、娱乐、虚拟助手等方面取得了显著进展，还为社会创造了更多的可能性。二十一世纪是信息的社会，大数据和人工智能的双重加持给我们的生活带来更多的惊喜和便利。随着数据爆炸式增长成为常态，对数据的计算能力也提出了更高的要求。在日常生活中，无论是智能手机、安防监控，还是电动汽车、服务器集群，无处不体现着云计算、分布式计算和边缘计算的身影。这些计算模式的融合和协同使得数据的处理更加高效，为人工智能的发展提供了强有力的支持。

机器学习技术早在上世纪八十年代就初露头角，但是由于计算能力的限制，前期发展较为缓慢。直到本世纪 GPU（Graphics Processing Unit）的飞速发展，强大的算力对神经网络的训练提供了坚实基础，深度学习逐渐提高到了前所未有的高度。随着网络拓扑结构日益复杂、网络推理精度日益提高，模型参数量和计算量也极具增加。在神经网络计算平台中，CPU（Central Processing Unit）在通用性方面表现出色，但是其串行执行严重限制了数据的并行处理能力。为此科研人员做出了许多努力，从编译器优化出发，通过对神经网络模型进行图优化、指令调度等工作，以确保生成的指令序列最大限度的利用硬件资源。同时多线程技术和 SIMD（Single Instruction Multiple Data）技术的引入也在一定程度上提高了计算的并行度。GPU 在深度学习计算中扮演着主力角色，其中以 NVIDIA 为代表提出的 CUDA（Compute Unified Device Architecture）并行计算架构最为著名。CUDA 为 GPU 提供了一套高效、灵活的编程模型，广泛支持深度学习架构，帮助开发者充分利用硬件并行计算能力服务于计算密集型领域，但是 GPU 高功耗的特点使其在边缘设备应用严重受限。

随着半导体工艺和芯片设计水平的高速发展，领域特定架构（Domain-Specific Architecture, DSA）逐渐成未来计算架构的发展潮流。针对特定应用领域或者特定工作负载的计算需求，领域特定架构与传统通用计算架构不同，其旨在通过专用硬件加速特定类型的计算任务。领域特定架构芯片种类繁多，因其低成本、低功耗、高算力的优势备受关注。在深度学习领域，国内外厂商发布了众多专用于神经网络推理和训练的 NPU（Neural Processing Unit）芯片。早在 2016 年，Google 就推出其自研的 TPU 芯片用于云服务器加速；同年寒武纪发布了首个终端 AI 商用 NPU 芯片，Cambricon-1A；随后 2019 年华为在手机 SoC 中集成 DaVinci NPU。领域特定架构的灵活性和高效性使

其在满足特定需求的同时，对硬件资源的利用更加精准，为未来计算技术的创新提供了新的可能性。扩大 DSA 在各行各业的应用将推动更多创新性的产品和服务的涌现，引领计算架构的新潮流。

本文旨在设计一款面向深度学习领域的高性能硬件加速器，系统采用异构设计，结合 DSA 设计理念，实现软硬件协同加速。基于 Verilator 开发软硬件联合仿真器，分析系统性能瓶颈，优化控制流及数据流，并在 FPGA 平台上进行原型验证。

1.2 国内外研究现状

近些年来，随着应用市场的进一步扩大，智能汽车、智能终端等领域迎来了飞速的发展，形成了一个蓬勃发展的智能化生态。这一发展趋势受益于大数据时代的来临，丰富的信息和日益增长的数据流量为各个领域的智能化提供了强大支持，但也给云端和边缘端设备带来新的挑战。同时，领域特定架构的发展也推动着学术界和工业界在深度学习硬件加速上的创新。

Eyeriss^{[1][2]}是 Yu-Hsin Chen 等人在 2016 年提出的一种高能效、可重配置的深度卷积神经网络加速器，该架构旨在应对深度学习推断过程中由数据搬移引起的能耗问题。为了最大程度地进行数据复用，Eyeriss 采用了 Row Stationary 的数据流配置策略，并利用权重、输入特征图和累加和的不同特征，提出了三种不同的数据复用方法，如图 1-1 所示。针对 2D 卷积，权重沿 PE（Processing Element）水平方向复用，输入特征图沿 PE 对角线方向复用，而累加和在垂直方向复用。PE 间的数据复用和累加减少了对全局缓冲区和 DRAM 的访问。除此之外，借助稀疏优化进行数据压缩，在 AlexNet 网络测试中，以 278mW 的功耗实现了每秒 34.7 帧的推理速度。

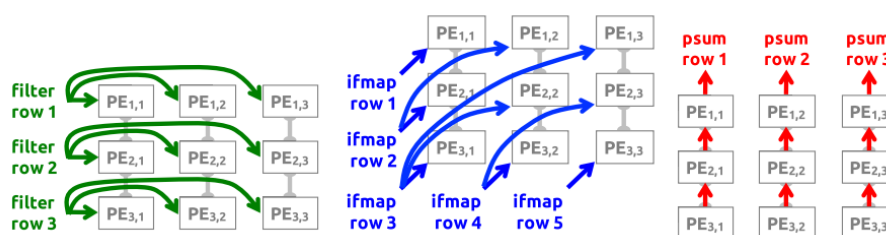


图 1-1 Eyeriss 数据复用策略图^[1]

Yu-Hsin Chen 等人在 2018 年提出 Eyeriss v2^[3]，该架构专为移动设备中稀疏网络的推理而设计。相比于 Eyeriss，Eyeriss v2 可以直接在压缩域中处理权重和激活的稀疏数据，进一步提高了稀疏模型的推理速度，同时有效降低了运行功耗。NVDLA 是英伟达开源的深度学习软硬件参考方案，具有可扩展性、模块化、高度可配置等特点。该硬件广泛支持物联网设备，通过整合 AI 解决方案，让更多行业融入 AI SoC 的设计中。英特尔提出 Nervana 用于处理神经网络中的矩阵乘法和卷积任务，该内存架构无需多层次缓存，即可实现数据的高效计算和移动，从而提升了深度学习的训练性能。

在人工智能的热潮中，涌现了众多优秀的硬件加速架构。自 2014 年开始，寒武纪依次推出了 DianNao^[4]、DaDianNao^[5]、ShiDianNao^[6]、PuDianNao^[7]和 Cambricon-X^[8]等一系列 AI 加速器。DianNao NPU 是寒武纪的开山之作，与 128 位 2GHz 的 SIMD 处理器相比，实现了 117.87 倍的性能提升，同时功耗降低了 21.08 倍。DianNao NPU 主要应用于嵌入式设备，而 DaDianNao NPU 承担着服务器中高性能计算的任务。尽管神经网络加速器在计算机视觉和自然语言处理中占有重要地位，但仍然受到内存带宽的限制。对于 CNN 而言，权重在许多神经元之间共享，从而可以大大减小内存占用。ShiDianNao 充分利用这一特征，将 CNN 完全映射到 SRAM 中，并采用 NPU 与 CMOS 紧耦合设计，极大降低了 DRAM 的访问。DianNa、DaDianNao 和 ShiDianNao 主要应用于深度学习领域，PuDianNao 则将 NPU 的应用推广到机器学习领域，以支持 SVM、线性回归、分类树等问题。Cambricon-X 利用神经网络的稀疏性进行优化，通过对每个 PE 分配一个单独的索引单元以实现权重和输入数据的匹配。

Chen Zhang 等人提出 RoofLine 设计方案，定量分析 CNN 在 FPGA 部署中计算吞吐量和内存带宽的匹配问题，追求具有最佳性能和最低 FPGA 资源需求的解决方案^[9]。神经网络在训练期间进行模型剪枝会产生零值权重，并且在推理期间 ReLU 激活之后也会产生零值^{[10][11]}，A Parashar 等人通过压缩编码保留权重和激活后数据的稀疏性，从而消除了不必要的数据传输，降低存储要求^[12]。Wenyan Lu 等人针对计算引擎和 CNN 工作负载并行结构不匹配的问题，优化数据流结构，利用特征图、神经元和突触并行性之间的互补效应来减轻不匹配现象^[13]。

在深度学习硬件加速领域，脉动阵列^{[14][15]}（Systolic Array）被广泛应用于加速矩阵乘法等线性代数运算中，该技术最早由 H.T.Kung 在 1978 年提出。脉动阵列采用流水化和并行化的设计思想，通过对数据的复用，能够在较小的内存带宽下实现较大的数据吞吐量。为了平衡存储带宽和计算速度，J Shen 等人将卷积层进行细粒度划分，采用三种不同的脉动阵列映射策略以屏蔽外部存储的访问时间^[16]。Liancheng Jia 等人使用 Chisel^{[17][18]}敏捷开发，设计了一款脉动阵列生成器，模块化的设计范式极大方便了张量算法自动生成脉动硬件架构^[19]。针对脉动阵列进行小规模卷积和深度卷积 PE 利用率急剧下降的问题，Rui Xu 等人在脉动阵列中添加数据路径，允许用户通过配置分割阵列，提高了系统整体的灵活性和计算效率^[20]。考虑到深度学习中权重和输入特征图的数据稀疏性，Bo Wang 等人提出 GSMC（Group Structure Maintained Compression）压缩策略，在脉动阵列中实现双边稀疏加速^[21]。

Google 第一代 TPU^[22]（Tensor Processing Unit）芯片于 2016 年首次亮相，作为一款专为深度学习工作负载设计的定制化硬件，其采用脉动阵列作为计算单元，以实现低成本高算力的目标。其核心采用 65536 个 8 位宽 MAC 单元，可以提供 92TOPS 的峰值算力以及 28MiB 的片上存储空间。TPU 采用 CISC 指令集，与传统 CPU 相比，

该设计更加简单粗暴，无需考虑分支预测、乱序执行等复杂逻辑；除此之外，其依托 Tensorflow 深度学习框架，可以在云服务器中实现深度学习算法的快速部署。

VTA^[23]是一款以通用矩阵乘法（General Matrix Multiplication, GEMM）核心为基础构建的深度学习加速器，专用于实现高吞吐量的密集矩阵乘法操作。设计理念上采用了解耦的访问执行机制，旨在有效隐藏内存访问延迟并最大化计算资源的利用率。VTA 不仅为广大开发者提供了一个完整的硬件加速设计范例，也为编译栈提供了清晰的张量计算抽象。该硬件架构由取指、加载、计算和存储四个模块组成，在硬件整体层次上组成任务流水线，模块间通过 FIFO 队列与 SRAM 内存块进行通信，实现了任务级的管道并行，如图 1-2 所示。VTA 核心由 GEMM Core 和张量 ALU 组成，GEMM 执行高度密集的线性代数计算，ALU 执行逐元素计算，以实现多种算子的硬件加速。同时，VTA 借助 TVM^[24]强大的空间优化能力，在硬件后端上实现了高效的调度。然而，需要注意的是，VTA 的开发采用了高层次综合（High Level Synthesis, HLS）实现，局限性较大。

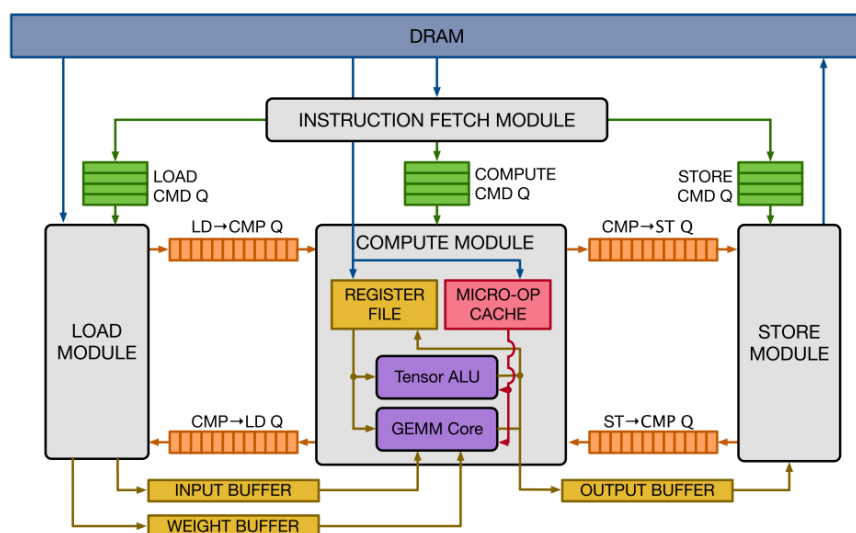
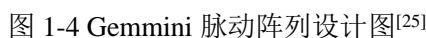
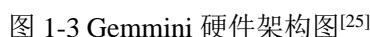


图 1-2 VTA 硬件架构图^[23]

Gemmini^{[25][26]}是 UCB 提出的面向深度学习应用的硬件生成器，参数化的设计使其具备高度的灵活性。该设计采用 Chisel 开发，提供了灵活的硬件模板、多层软件堆栈和集成的 SoC 环境。Gemmini 的加速器核心是一个用于实现矩阵乘的脉动阵列，采用 SRAM 作为脉动阵列的输入输出缓冲区，如图 1-3 所示。该架构中的脉动阵列可配置为输出固定和权重固定两种数据流格式，多层级 MAC 设计和缓存结构进一步增大了数据吞吐量。除此之外，设计中还包含激活单元、缩放单元、转置单元等，共同构成了一套完整的硬件加速器。Gemmini 采用 RoCC 接口进行指令集扩展，基于 Chipyard 项目，可以快速实现加速器与 Rocket^[27]和 BOOM^[28]核集成。



硬件架构的飞速发展给底层带来了无尽的可能，但是如何实现端到端的部署成为一个具有挑战性的难题，其中 AI 推理引擎和编译器发挥着重要作用。

TVM^[24]是一款开源的、端到端的深度学习模型编译栈，借助其强大的优化性能，可以实现多个硬件后端的快速部署。目前，TVM 已经对多种深度学习框架进行了适配，包括 Pytorch、Tensorflow、MXNet 等。输入的模型首先进行计算图优化，转换为高层次 IR；随后借助 AutoTVM 在搜索空间中确定最优策略。TVM 采用了计算和调度解耦的设计理念，以此来最大限制的利用硬件资源。MLIR^[33]是一个由 Google 推动的开源项目，通过多层次的中间表示，以提供一个高度灵活、可复用、可扩展的编译器框架。该架构旨在解决软件跨平台的碎片化问题，通过整合现有编译器资源，改善异构系统编程生态，统一领域特定架构编程模型。MLIR 通过 Dialect 来定义不同层次的 IR 来适用不同的应用场景，当然也可以开发者自行定义，通过 Dialect 之间的相互转换，得到目标平台的机器码。CIRCT^[34]是一个基于 MLIR 的开源 EDA 框架，试图统一硬件的中间表示，与 MLIR 共同打造软硬件新生态，缩短 IC 前端开发周期。

在边缘端和移动端的神经网络端到端部署过程中，相比于 AI 编译器，轻量级的 AI 推理引擎技术更加成熟，MNN^[35]是阿里巴巴开源的推理引擎，支持深度学习的训练和推理，适用于云服务器及各种嵌入式设备。TNN 和 NCNN 都是腾讯优图实验室开源的高性能、轻量化的神经网络推理框架，具备跨平台、高性能等突出优点。除此之外，TensorFlow Lite、ONNX Runtime、CoreML、TensorRT、OpenVINO、ArmNN 等 AI 推理引擎也在各硬件后端中广泛应用。

1.3 本文主要研究内容

本文的研究方向主要包括以下几个方面：

1) 采用 Chisel 敏捷开发理念，开发了高性能 CNN 硬件加速核心。

在本文的研究工作中，设计一款面向 CNN 硬件加速的高性能、高灵活性的 NPU，该硬件核心采用 Chisel 敏捷开发，极大提升了硬件开发速度。NPU 的核心计算单元由脉动阵列和向量 ALU 单元组成，其中，脉动阵列由 1024 个 PE 组成，每个 PE 可以在单周期内完成两组 INT8 的 MAC 计算。此外，在硬件架构设计中结合算子融合、重量化等设计思想有效地降低了硬件的访存时间和计算时间。采用 SMIC 28nm 工艺库综合电路且保留 20% 的时序裕量，NPU 主频可达 666MHz，提供 2.728TOPS@INT8 的峰值算力。

2) 开发硬件适配的 AI 推理引擎，实现了神经网络算法的端到端部署。

在通用硬件加速领域，软件的完整度和灵活度决定了整个系统的上层建筑。本文以 NCNN 推理引擎为基础，根据硬件算子融合情况在工具链中进行计算图优化；并添加自定义硬件后端，提供模型转换和模型量化脚本。此外，在边缘端裸核运行环境中，移植 FATFS 文件系统，通过驱动层、加速库以及应用层代码的解耦设计极大降低了二次开发难度。在模型量化角度，本文采用训练后对称量化方案，并实现了神经网络的端到端部署。

3) 基于 Verilator 开源仿真器, 实现了软硬件联合仿真。

针对传统 RTL 验证流程繁琐、耗时耗力等问题, 本文基于 Verilator 设计了一款轻量化的软硬件联合仿真器, 支持本文提出的 NPU 软硬件系统, 并在仿真环境中实现了神经网络的端到端模型部署和仿真验证。此外, 该框架支持硬件模块级、算子级、网络级的仿真粒度, 并支持数据自动校验、故障定位、随机测试、批量测试、AXI 时序诊断、性能分析和波形记录等功能。

1.4 章节安排

第一章为绪论部分。首先介绍了深度学习算法的发展趋势和应用前景, 并简述各种硬件通用计算平台的优缺点。其次, 介绍了领域特定架构神经网络硬件加速的国内外研究现状, 并针对典型的硬件加速设计进行详细剖析。此外, 阐述了现阶段工业界和学术界主流的 AI 编译器和推理引擎解决方案。最后, 简要概述了本文的主要研究内容和章节安排。

第二章为神经网络硬件加速设计优化部分。首先介绍了卷积神经网络的发展历程并总结了现阶段常见的 CNN 网络的计算量和参数量。其次从循环展开的角度分析了卷积算子的硬件加速方案并给出了适用于 FPGA 的深度可配置行缓冲机制。此外, 详细介绍了常见的卷积加速算法, 如 im2col+GEMM 等, 并对比了脉动阵列中常见的权重固定和输出固定两种数据流格式。最后, 简要概述了适用于硬件加速的模型压缩方案, 如模型剪枝、模型量化等。

第三章为硬件加速器架构设计部分。首先介绍了本文 NPU 设计的数据流架构以及顶层数据流和控制流接口。该硬件加速方案的核心计算模块由脉动阵列和向量 ALU 组成, 以实现卷积、深度卷积、全连接、以及各种逐元素算子层的映射。此外, 添加 im2col、软硬协同的大尺寸特征图分块策略、算子融合和重量化策略以优化数据访存, 进而显著提升 NPU 的推理性能。

第四章为模型部署工具链及验证平台设计部分。首先介绍了硬件加速在面向深度学习领域的技术路线发展, 并详细介绍了 NCNN 推理引擎的优缺点、硬件后端适配情况、内存模型以及网络参数文件格式等内容。其次, 针对硬件算子融合的支持情况进行计算图优化, 采用 INT8 量化方案, 支持模型的端到端部署。此外, 在边缘端裸核运行环境中, 实现了驱动层、加速库和应用层的软件解耦设计, 极大降低了应用二次开发的成本。最后, 本文提出了基于 Verilator 的软硬联合仿真器, 支持硬件模块级、算子级、网络级的仿真验证、性能分析、批量测试和故障定位等功能。

第五章为实验测试与性能分析部分。首先采用本文提出的 Verilator 软硬联合仿真方案对系统中存在的性能瓶颈进行分析。其次, 搭建 FPGA SoC 原型系统和视频流实时处理系统, 对比分析 NPU 的加速性能和推理精度。最后采用 Design Compiler 进行电路综合, 并分析了 NPU 的整体性能、关键时序路径、资源面积等内容。

第六章为总结与展望部分。对本文的整体研究工作进行系统性总结, 并针对目前

软硬件方案存在的缺点和性能瓶颈进行简要分析，指明下一步研究方向。

2 神经网络硬件加速设计优化

2.1 卷积神经网络基础

2.1.1 卷积神经网络的发展

卷积神经网络（Convolutional Neural Network, CNN）的发展历程可以追溯到 LeNet^[36]时期，由 Yann LeCun 等人在 1998 年提出，主要应用于手写数字识别领域。LeNet 网络作为早期 CNN 代表之一，采用了卷积、池化和全连接的基本结构，奠定了当前 CNN 网络的模型框架。然而，由于当时计算能力的限制以及大规模数据集的缺失，深度学习的发展受到了很大阻碍。2012 年，AlexNet^[37]横空出世，并在 ImageNet 大规模图像分类挑战赛（ImageNet Large Scale Visual Recognition Challenge, ILSVRC）中拔得头筹，在沉寂已久的深度学习领域激起一阵轰鸣。紧接着，GoogleNet^[38]和 VGGNet^[39]在 2014 年相继提出，斩获 ILSVRC 冠亚军。VGGNet 采用相同尺寸的小卷积核来构建深层网络，通过多层小尺寸卷积达到大尺寸卷积核类似的感受野，进而减小了参数量。GoogleNet 采用 Inception 模块化设计，该结构通过不同尺寸的卷积核捕捉图像中不同尺度的特征，进而增强了对细节的理解能力。其广泛使用 1×1 卷积，在增加网络表达能力的同时，降低了模型参数量。

随着网络深度的增加，梯度消失和梯度爆炸的问题越来越严重。2015 年，何凯明等人提出 ResNet^[40]网络，采用残差结构解决了超深网络在训练过程中的性能退化问题，使网络深度大幅增加，但是过多的冗余层也增加了计算负担。为了进一步提升残差网络的性能，Gao Huang 等人于 2016 年提出了 DenseNet^[41]网络结构，其独特之处在于通过密集连接的方式建立起每一层和后续所有层的桥梁，从而实现特征和细节的高效传递和共享。虽然 CNN 网络在不断的发展过程中，模型的准确度不断提升，但随之而来的参数量和计算量的问题也愈加严峻，对模型的边缘端部署提出了新的挑战。自 2016 年起，相继出现了一系列轻量化网络，如 SqueezeNet^[42]、MobileNet^[43]、ShuffleNet^[44]等，通过深度可分离卷积等手段，成功降低了模型参数量，实现了模型轻量化，在资源受限的嵌入式系统中极具竞争力。除此之外，在计算机视觉领域，YOLO 作为一类目标检测算法，因其优秀的性能获得了各领域的关注，典型代表有 YOLOv3^[45]、YOLOv5、YOLOv8 等。

表 2-1 中列举了常见 CNN 网络模型的参数量和计算量。AlexNet 和 VGG 网络较浅，由于结构中存在多层的全连接，网络参数量较大。GoogleNet 采用模块化设计，使用大尺寸平均池化代替全连接层，大幅度降低参数量。ResNet 引入残差块的设计使训练更深的网络成为可能，常见的 ResNet 结构包括 ResNet18、ResNet34、ResNet50、ResNet101 和 ResNet152 五种。YOLO 系列算法在边缘端和移动端目标检测中占据重要地位，YOLOv3_Tiny 凭借简单结构，有较多研究将其部署在 FPGA 中加速；而 YOLOv3 中包

含 75 层卷积层，参数量和计算量均较大，对于存储能力和计算能力有限的边缘端应用受到一定限制。YOLOv5 和 YOLOv8 由 Ultralytics 公司开源，两者设计理念类似，均采用骨干网络（Backbone）进行特征提取，使用颈部网络（Neck）和检测头（Head）进行特征融合。YOLOv5s 是 YOLOv5 的最小版本，其拥有 YOLOv5 系列中最少的层和最小的计算复杂度，除此之外 YOLOv5 系列中还包括 YOLOv5m、YOLOv5l、YOLOv5x 和 YOLOv5n。SqueezeNet、MobileNet、ShuffleNet 均是边缘端和移动端常用的轻量化模型，所需算力与 YOLOv3 相差数十倍甚至数百倍；SqueezeNet 和 ShuffleNet 常用于目标分类任务，而 MobileNet 常结合 YOLO、SSD 等骨干网络完成实时目标检测与识别任务。

表 2-1 常见 CNN 模型参数量及计算量

| 网络名称 | 输入分辨率 | 参数量/M | 计算量/GFLOPS |
|-------------------|---------|--------|------------|
| AlexNet | 416×416 | 61.10 | 2.44 |
| VGG16 | 416×416 | 138.36 | 53.05 |
| VGG19 | 416×416 | 143.67 | 67.41 |
| GoogleNet | 416×416 | 6.63 | 5.21 |
| DenseNet121 | 416×416 | 7.98 | 9.99 |
| ResNet18 | 416×416 | 11.69 | 6.29 |
| ResNet50 | 416×416 | 25.56 | 14.25 |
| ResNet152 | 416×416 | 60.19 | 40.02 |
| YOLOv3_Tiny | 416×416 | 8.85 | 5.56 |
| YOLOv3 | 416×416 | 61.89 | 65.86 |
| YOLOv5s | 640×640 | 7.2 | 16.5 |
| YOLOv8s | 640×640 | 11.2 | 28.6 |
| SqueezeNet | 416×416 | 1.25 | 2.93 |
| MobileNetv2 | 416×416 | 3.51 | 1.13 |
| MobileNetv3_large | 416×416 | 5.48 | 0.80 |
| ShuffleNetv2_0.5x | 416×416 | 1.37 | 0.15 |

2.1.2 卷积计算并行维度的探索

卷积神经网络一般采用卷积层和深度卷积层进行特征提取，使用池化层进行下采样，公式 2-1 和 2-2 给出了输出特征图尺寸（ W_{out} 、 H_{out} ）与输入特征图尺寸（ W_{in} 、 H_{in} ）的关系， p_{top} 、 p_{bottom} 、 p_{left} 和 p_{right} 表示输入特征图四个方向的 Padding 大小， s_w 和 s_h 表示水平和垂直方向的 stride， k_w 和 k_h 表示 kernel 的长宽。对于卷积层和池化层，输入特征图的填充方式一般采用 Same Padding 模式以保持输入和输出特征图长宽相同，Valid Padding 和 Full Padding 模式使用较少。卷积层中 1×1 和 3×3 尺寸的 kernel 最为

常见，当然也有例外，如 AlexNet 采用了 11×11 卷积、ResNet 和 GoogleNet 采用了 7×7 卷积等。除此之外，输出特征图的通道数等于卷积核的个数、即滤波器的个数，如公式 2-3 所示。公式 2-4 和 2-5 给出了卷积层和深度卷积层的权重参数量，其中深度卷积输入通道数等于输出通道数。

$$H_{out} = \frac{H_{in} + p_{top} + p_{bottom} - k_h}{s_h} + 1 \quad (2-1)$$

$$W_{out} = \frac{W_{in} + p_{left} + p_{right} - k_w}{s_w} + 1 \quad (2-2)$$

$$C_{out} = num_filters \quad (2-3)$$

$$wgt_conv_size = k_w \times k_h \times C_{in} \times C_{out} \quad (2-4)$$

$$wgt_convdw_size = k_w \times k_h \times C_{in} \quad (2-5)$$

在 CNN 网络，卷积运算消耗了整个网络推理过程中的绝大部分时间，因此，硬件加速器架构往往针对卷积运算做极致的访存优化和并行优化。源码 2-1 以 C 语言的形式展示了卷积运算的六重循环结构，其中 IC 表示输入特征图通道数，OC 表示输出特征图通道数，OW 和 OH 表示输出特征图的长宽，KW 和 KH 表示卷积核的长宽；ofm 为输出特征图的首地址，ifm 为输入特征图的首地址，wgt 为权重的首地址。

源码 2-1 卷积运算循环展开

```

1.  for(int oc=0; oc<OC; oc++)           // 遍历 ofm channel 方向
2.      for(int ic=0; ic<IC; ic++)         // 遍历 ifm channel 方向
3.          for(int ow=0; ow <OW; ow++)    // 遍历 ofm w 方向
4.              for(int oh=0; oh <OH; oh++) // 遍历 ofm h 方向
5.                  for(int kw=0; kw<KW; kw++) // 遍历 kernel w 方向
6.                      for(int kh=0; kh<KH; kh++) // 遍历 kernel h 方向
7.                          ofm[oc][ow][oh] += ifm[ic][ow+kw][oh+kh]*wgt[oc][ic][kw][kh];

```

不管是基于 SIMD 的软件加速方案，还是硬件底层架构的设计，并行计算的核心都围绕着六重循环中某些循环的展开，下面以 FPGA 加速 CNN 中常见的滑窗法为例，分析数据并行方案、访存优化和数据复用问题。由于 kernel 级尺寸往往较小，一般以 3×3 和 1×1 卷积为主，硬件设计中可以采用全并行方案。当 $s_w < k_w$ 时，可以在水平方向上对特征图进行数据复用，因此一般不对特征图水平方向上进行循环展开；由于输入特征图的每个位置均需要和 oc 个卷积核进行卷积运算，而片上存储空间有限，故常对 ih 和 oc 做展开处理。当沿 ih 方向做特征图分块时，仅相邻块边界存在数据重叠，特征图部分访存压力较小。当沿 oc 进行展开时，假设每次步进 oc_step，则需要连续读取输入特征图 oc/oc_step 次，增加了访存时间。但是权重数据较多，片内 RAM 无法满足存储需求，为此 oc 分块往往用于面积和速度的平衡。除此之外，卷积运算中需要沿 ic 方向

进行累加，如果对 ic 分块，则需要将中间结果写回外部存储器，浪费性能，因此，ic 方向可以做并行加速但不进行分块处理。

滑窗法常用于嵌入式 FPGA 或定制化 ASIC 加速专用神经网络，图 2-1 给出了滑窗法的一种硬件实现方案，核心理念在于深度可配置行缓冲机制的设计。行缓冲在 FPGA 加速数字图像处理领域非常常见，如高斯滤波、边缘检测、腐蚀膨胀等算法的硬件实现。深度行缓冲机制则是在行缓冲的基础上进行改进，增加 Mux 使得行缓冲深度可变。

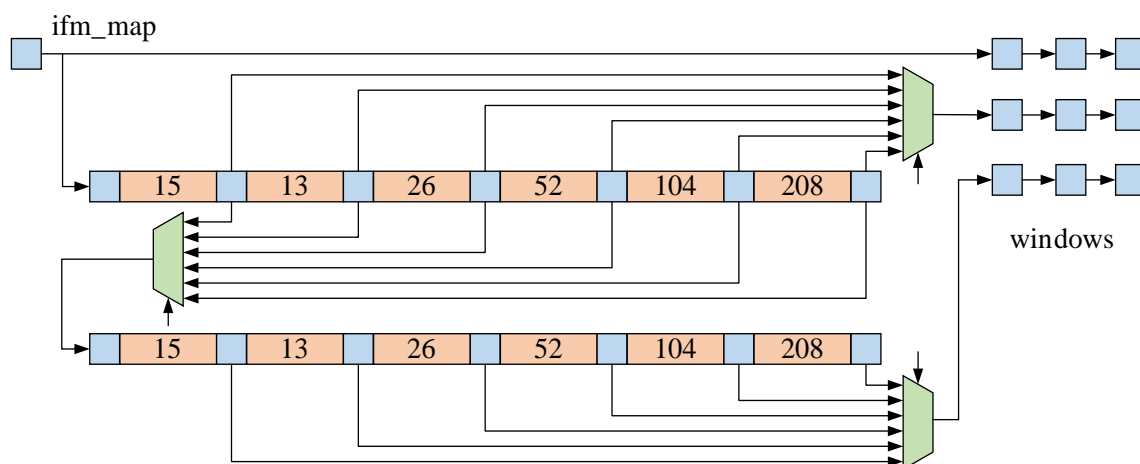


图 2-1 适用于硬件加速的深度可配置行缓冲设计

行缓冲由多行移位寄存器组成，每行的移位寄存器根据预先设置给出抽头。图 2-1 移位寄存器的深度和抽头专为 YOLOv3_Tiny 网络 3×3 卷积设计，输入图像为 416×416 的 RGB 图像，推理过程中临时特征图尺寸包括 13×13 、 26×26 、 52×52 、 104×104 和 208×208 。当特征图的输入尺寸为 13×13 时，由于 Padding 的需要，特征图水平方向需要填充两个元素，因此首个移位寄存器深度为 15，即行缓冲第一个抽头输出至下一行的行缓冲输入。同理，特征图尺寸为 26×26 时在第二个抽头处输出，通过缓冲两行的特征图数据，即可产生 3×3 卷积窗。该行缓冲单元可以对输入特征图的单个通道进行计算，可以例化 N 个上述单元实现 ic 方向的并行；同时， M 个卷积核均可同时与卷积窗计算，实现 oc 方向的并行。此时，每个周期最多可以完成 $M \times N \times 9$ 次乘法运算。

2.2 卷积加速算法

滑窗法在专用神经网络硬件加速器中被广泛采用，其结构简单但缺乏灵活性和通用性，逐渐退出硬件通用加速的舞台。目前，在领域特定架构的 NPU 设计中，常见的卷积加速算法主要包括 FFT、Winograd、im2col+GEMM 等。FFT 算法即快速傅里叶变换算法，在数据信号处理和数字图像处理领域广泛应用。基于 FFT 的卷积算法将空间域的卷积运算转换为频率域的矩阵点积运算，尤其在大尺寸卷积核的情况下，能够实现较好的加速效果。然而，在卷积网络发展过程中，小尺寸卷积逐渐取代大尺寸卷积并成为主流。由于特征图尺寸通常远大于卷积核尺寸，需要对卷积核进行零值填充以匹配特征图尺寸，从而增加存储开销，因此现阶段硬件加速器中较少采用 FFT 算法。

2016 年, A Lavin 等人首次将 Winograd 算法应用于卷积加速领域^[40], 目前该算法在 NNPACK、NCNN、cuDNN 等框架中广泛使用。该算法核心思想是通过矩阵变换减小乘法的计算次数, 从而实现加速, 但同时该方法会增加加法的次数以及需要额外的转换计算。在硬件部署过程中会消耗更多的资源存储转换矩阵, 同时较高的硬件复杂度也限制了其发展。GEMM 即通用矩阵乘法, 常常搭配 im2col 算法使用, 该方法结构简单、通用性强, 在 GPU、NPU 芯片中得到广泛应用。

im2col 算法常用于优化卷积运算, 它将传统的空间卷积转换为矩阵乘法运算, 从而提高了计算效率。这种转换对于硬件后端的实现尤为重要, 现代计算机处理器中有对矩阵乘法高度优化的实现, 例如基本线性代数算子库 (BLAS) 利用底层硬件的高并行性进行加速。im2col 算法的核心思想把卷积操作中输入特征图按照滑窗法的位置, 将其对应的局部区域数据提取出来并展开成行向量, 重复这一过程, 得到输入特征图矩阵, 矩阵的列数等于 $W_{out} \times H_{out}$, 矩阵的行数等于 $k_w \times k_h \times C_{in}$; 同时将每个卷积核均展开成列向量, 得到行数为 C_{out} 、列数为 $k_w \times k_h \times C_{in}$ 的权重矩阵。图 2-2 展示 im2col 算法进行卷积展开的原理, 其本质就是将三维的特征图根据卷积核的尺寸按原图对应位置展开成行, 卷积核也对应的展开成列, 实现降维。

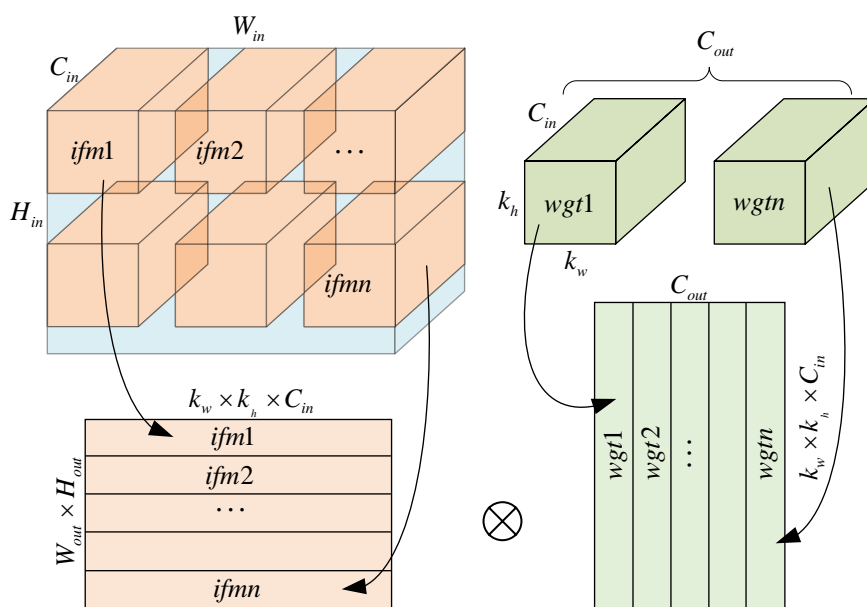


图 2-2 im2col 算法原理图

矩阵乘法常见加速方案包括内积、外积和脉动阵列三种实现方案, 内积法和外积法的并行张量计算主要针对向量的加速。内积法主要思想是将矩阵计算 lower 成向量计算, 循环多次以达到矩阵计算的目的。该方法将输入矩阵 A 的每一个行向量和输入矩阵 B 的每一个列向量进行内积运算, 得到输出矩阵 C 中的一个元素, 迭代这一过程, 得到矩阵 C 的全部数据, 例如英伟达开源的 NVDLA 采用的就是这一设计。采用内积法设计的硬件加速器可以很容易通过堆叠计算单元提升系统算力, 但是该方案计算单元的数据流采用广播输入, 需要在数据路径中插入大量 Buffer 以降低高扇出对性能和

时序的影响，进而产生很多无效逻辑，增大面积。外积法采用输入矩阵 **A** 的每一个列向量和输入矩阵 **B** 的每一个行向量进行外积计算，该方法在硬件加速策略中较少使用。脉动阵列是用于实现矩阵 Core 最常见的微架构，具有算法简单易于实现、硬件友好、灵活高效、高性能等特点。脉动阵列的硬件实现方案一般采用多层级设计，即堆叠 PE 组成 Tile，多块 Tile 构成 Mesh，以提高 PE 的利用率并增加对输入数据的复用能力。

脉动阵列的数据流格式包括权重固定和输出固定两种。输出固定的数据流如图 2-3 所示，采用 2×2 个 PE 组成的脉动阵列以方便理解数据流结构。PE 阵列同时串行输入两个矩阵的列向量，当矩阵数据在计算单元之间流动时，乘积结果累加在计算单元的暂存器中。经过若干个周期，计算单元中得到完整的乘累加计算结果。因为 PE 单元之间通过寄存器传输数据，存在一个周期的时钟延时，为了在正确的时钟周期抵达同一个计算单元，脉动阵列每一行的输入相比上一行多插入一个时钟周期延时，每一列的输入也同理延迟一个时钟周期。

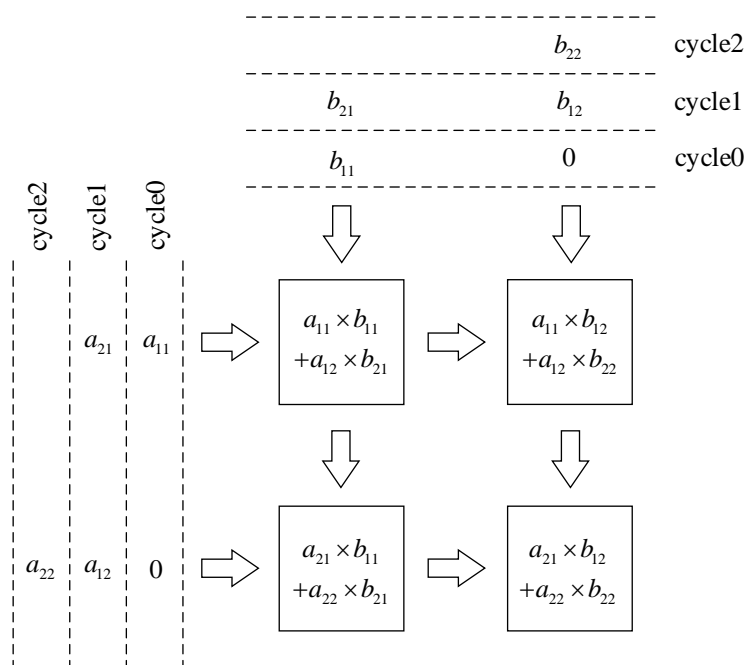


图 2-3 脉动阵列的 OS 数据流格式

权重固定的数据流如图 2-4 所示，预先将一个矩阵存入脉动阵列中，此时输入另一个矩阵，乘累加计算结果在计算单元之间流动，经过若干周期后，脉动阵列输出乘累加结果。与 OS 数据流类似，由于本设计采用时序电路，水平方向输入矩阵的列向量也在时间关系上延迟一个时钟周期。基于输出固定的数据流设计结构更加简单，但是每个 PE 单元中都需要一个额外的寄存器存储部分和。当数据精度为 INT8 的脉动阵列执行矩阵乘法任务时，极易出现部分和的上溢或下溢，需给定足够的加法器位宽，但也因此增加了额外的面积消耗。基于权重固定的数据流设计结构相对复杂，由于部分和在计算单元间流动，需要在脉动阵列底端添加额外的加法器进行部分和的累加；矩阵 **B** 预先存入阵列中，数据复用性更高，且计算单元中加法器的数据位宽要求更低。

采用 WS 固定的数据流方案，通过对 PE 进行改进，同样可以实现矩阵 B 的脉动，进而降低布线难度、提高系统频率，具体设计见第三章。

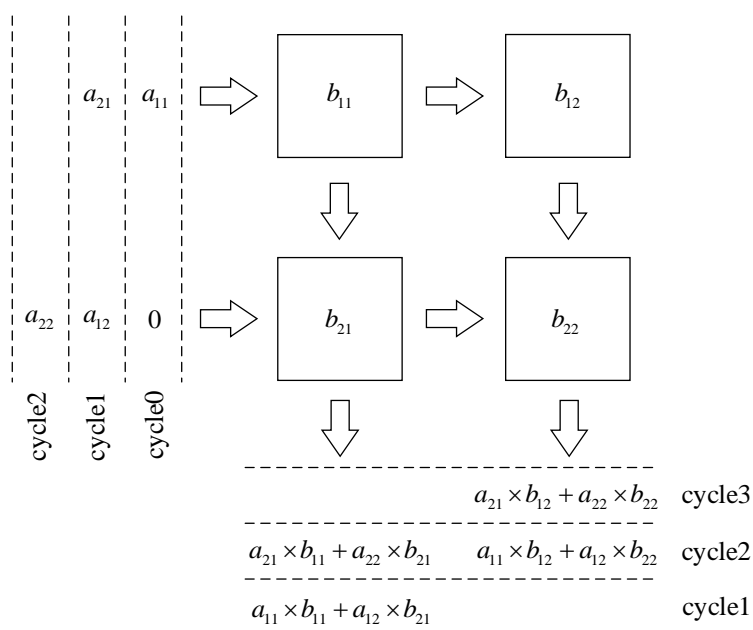


图 2-4 脉动阵列的 WS 数据流格式

以 OS 固定数据流为例，传统矩阵乘法和脉动阵列加速的矩阵乘法计算量对比如表 2-2 所示，显而易见，基于脉动阵列的矩阵乘法复杂度明显低于传统矩阵乘。随着输入尺寸的增大，脉动阵列的加速效果也更加明显。

表 2-2 传统矩阵乘和脉动阵列计算量对比

| 维度 | 传统矩阵乘运算 | | 脉动阵列运算 | |
|--------------|---------|------------|--------|-------|
| | mul | add | mul | add |
| 3×3 | 27 | 18 | 9 | 9 |
| 4×4 | 64 | 48 | 16 | 16 |
| 5×5 | 125 | 100 | 25 | 25 |
| $n \times n$ | n^3 | $n^2(n-1)$ | n^2 | n^2 |

2.3 模型压缩

针对 CNN 网络计算量和存储成本较大的问题，国内外学者提出了很多模型压缩算法，主要包括模型剪枝、量化、知识蒸馏、低秩分解以及紧凑卷积等方案。具体来讲，模型剪枝的核心思想是对权重裁剪得到稀疏模型，进而减小计算量；量化则采用低计算成本的数据格式代替高计算成本的数据格式，例如采用 FP16、BF16、INT8 等数据格式进行网络推理。知识蒸馏一般采用大型复杂网络（教师网络）辅助训练精简网络（学生网络），以实现相似的模型准确度。低秩分解采用奇异值分解（SVD）等算法将大尺寸的矩阵分解成多个小尺寸的矩阵，并使用低秩矩阵近似原本权重矩阵，极大

降低了模型参数量。紧凑卷积方案在轻量化网络中常常使用，比如分组卷积、深度可分离卷积等。

2.3.1 模型剪枝

模型剪枝通过消除网络中的冗余参数来减少模型的大小和计算量，同时保持模型的性能。根据剪枝的粒度，可以分为非结构化剪枝和结构化剪枝，如图 2-5 所示，非结构剪枝具有细粒度、不规则的特点，消除的权重呈现一定的随机性；结构化剪枝为满足硬件后端的需求，在牺牲一定精度的前提下，对模型的部分通道和滤波器等进行剪枝，粒度较粗、但零值权重排列较为规则。

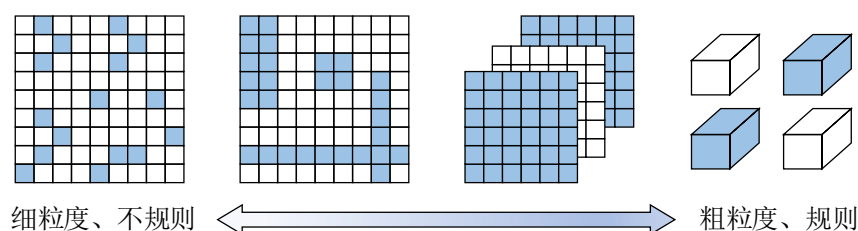


图 2-5 模型剪枝粒度划分

非结构化剪枝主要应用于支持稀疏计算的 NPU 中，对于以 BLAS 为基础的通用硬件后端，产生的加速性能有限，如 GPU 等。非结构化剪枝一般分为剪枝和微调两个步骤实现，剪枝过程常以权重参数的绝对值、以及训练过程中的参数梯度和二阶导数为度量，将满足约束的参数用零值代替；剪枝后的模型会出现性能和精度下降的现象，微调旨在通过重训练的方式恢复或提高模型精度。Han S^[46]等人结合非结构化剪枝、量化以及霍夫曼编码，成功将网络模型压缩 35 倍以上。X Ding^[47]等人提出基于动量 SGD 的优化方法，通过给定全局压缩比动态裁剪来降低网络复杂度，实现端到端训练。MIT J Frankle^[48]等人提出“彩票中奖”理论，研究证明剪枝的 CNN 网络能有效降低训练中的过拟合现象，且预先设计的初始化剪枝结构甚至可以达到优于原始模型的性能。

相比于非结构化剪枝，结构化剪枝视野不在局限于单个权重参数，而是针对于网络结构进行操作，例如层级剪枝、滤波器剪枝和通道剪枝等。在保证模型无损的前提下，结构化剪枝只能实现 50% 的模型压缩，而非结构化剪枝可以压缩 80% 以上的参数量。鉴于该方法在通用计算平台强大的加速和优化能力，国内外学者提出了众多实现方案，如以范数为度量的启发式设计方案^[49]、基于强化学习的剪枝搜索算法^[50]等。

2.3.2 模型量化

在神经网络模型的轻量化部署过程中，量化往往是最简单有效的压缩手段。该方法在数据传输和计算中采用高吞吐量低精度的数据类型来减小网络规模，进而缩短不同硬件后端的推理延迟。随着量化研究的深入和细化，涌现出了多种技术方案，图 2-6 从训练方法、量化策略和量化粒度三个角度进行分类。根据训练方法可以将量化技术分为训练后量化 (Post-training Quantization, PTQ) 和量化感知训练 (Quantization Aware

Training, QAT)。PTQ 方法是工业界常常使用的技术,具备快速、灵活、简便的特点,但是由于在模型校准的过程中需要一部分训练集数据,可能存在知识产权冲突以及隐私安全性等风险。训练后量化采用逐层的参数统计分析来确定最佳的量化步长和量化饱和阈值,由于此过程神经网络参数已经冻结,不会存在任何模型参数的更新,精度补偿受到很大限制,即使在训练集辅助校准的情况下,仍有可能带来较大的精度下降。量化感知训练的核心思想是在神经网络模型的训练过程中融合量化层的结构,在前向传播阶段即考虑量化误差,并在训练过程中直接进行量化补偿。相比于训练后量化方法,量化感知训练往往可以得到更高的量化性能。目前,对于主流的深度学习框架均有相应的模型量化 API 供开发者调用,如 Pytorch、Tensorflow、Paddle 等。

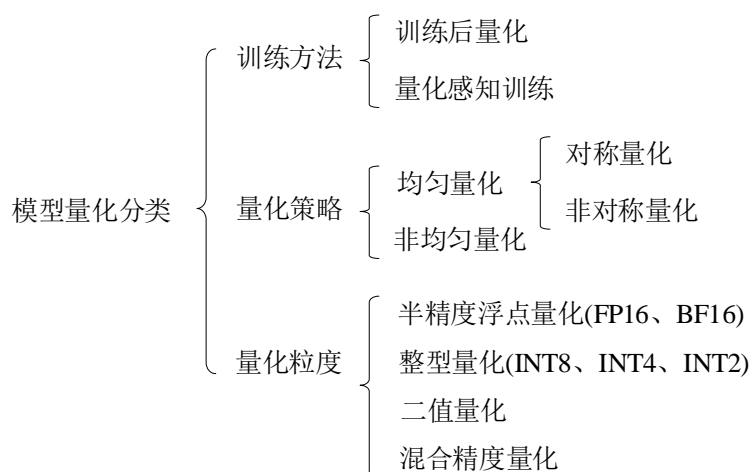


图 2-6 模型量化技术分类

从量化策略角度可以将模型量化分为非均匀量化和均匀量化两种,均匀量化又可以分为对称量化和非对称量化。在对网络参数和激活结果的直方图统计中可以发现,权重和激活值呈现非均匀分布的特点,因此可以利用非均匀量化进一步减小预测误差。均匀量化是模型端到端部署经常采用的方案,图 2-7 展示了其范围映射的原理。

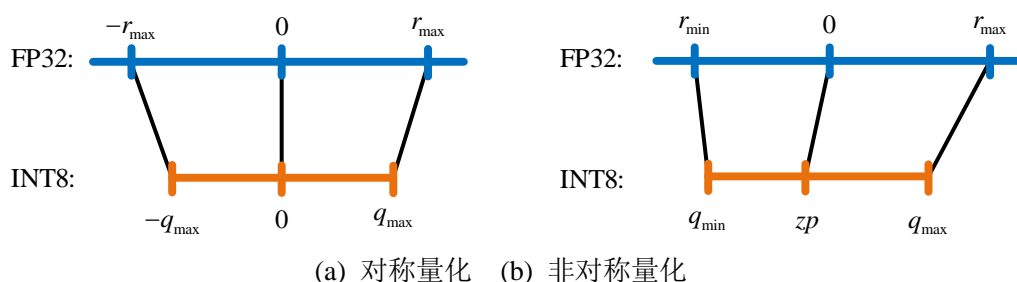


图 2-7 均匀量化示意图

模型训练中主要采用 FP32 的数据格式,而嵌入式硬件和移动设备常采用 INT8 的数据格式进行推理。图 2-7(a)对称量化中,原始参数的零点位置与量化后参数的零点位置对齐,因此只需要确定网络各层参数的量化区间 $[-r_{\max}, r_{\max}]$ 即可实现到 $[-q_{\max}, q_{\max}]$ 的映射,如公式 2-6 所示,其中 s 表示量化缩放因子, b 表示有符号整数的位宽。量化

函数和反量化函数如公式 2-7 和 2-8 所示, x 为原始数据, x_q 为量化后的数据, \hat{x} 为反量化后的数据, clip 和 round 分别为饱和截断和圆整函数。

$$s = \frac{2^{b-1} - 1}{r_{\max}} \quad (2-6)$$

$$x_q = \text{quantize}(x, b, s) = \text{clip}(\text{round}(s \cdot x, -2^{b-1} + 1, 2^{b-1} - 1)) \quad (2-7)$$

$$\hat{x} = \text{dequantize}(x_q, s) = \frac{1}{s} x_q \quad (2-8)$$

图 2-7(b)解释了非对称量化的基本原理, 相比于对称量化, 该方法在量化缩放因子的基础上引入了量化原点 z 的概念。需要注意的是, 对称量化后数据范围为 $[-2^{b-1} + 1, 2^{b-1} - 1]$, 而非对称量化后数据范围为 $[-2^{b-1}, 2^{b-1} - 1]$ 。量化缩放因子和量化原点的计算如公式 2-9 和 2-10 所示, 当采用 INT8 量化时, 实现原始参数区间 $[r_{\min}, r_{\max}]$ 到 $[-128, 127]$ 的线性映射, 公式 2-11 和 2-12 给出了非对称量化的量化函数和反量化函数。

$$s = \frac{2^b - 1}{r_{\max} - r_{\min}} \quad (2-9)$$

$$z = -\text{round}(r_{\max} \cdot s) - 2^{b-1} \quad (2-10)$$

$$x_q = \text{quantize}(x, b, s, z) = \text{clip}(\text{round}(s \cdot x + z), -2^{b-1}, 2^{b-1} - 1) \quad (2-11)$$

$$\hat{x} = \text{dequantize}(x_q, s, z) = \frac{1}{s} (x_q - z) \quad (2-12)$$

目前, 在神经网络模型边缘端和移动端的部署过程中, CNN 一般采用 INT8 量化, RNN 一般采用 FP16、BF16 量化, 以权衡推理精度和计算成本。当然, 某些特殊的应用场景也会采用低比特量化、二值量化以及混合精度量化, 例如存算一体芯片中由于 ADC 面积和精度的限制, 常采用压缩率更大的方案。

2.4 本章小结

本章首先介绍了卷积神经网络的发展, 并对常见 CNN 模型的参数量和计算量进行统计。其次从循环展开的角度剖析了卷积并行计算的理论可行性, 给出了适用于 FPGA 卷积加速的深度可配置行缓冲方案, 并详细阐述了 im2col+GEMM 的卷积加速算法。最后介绍了剪枝、量化等常见的模型压缩技术。

3 硬件加速器架构设计

3.1 硬件加速器顶层设计

3.1.1 加速器接口

本文设计的领域特定架构基本组成如图 3-1 所示，系统整体采用异构设计，以此支持更大的通用性和灵活性。CPU 可采用 ARM、RISC-V、MIPS 等指令集核心，适配能力和可移植能力较强，添加自定义的 NPU 单元，以提供 Scalar、Vector、Matrix 的全方位加速，并进一步增强系统的计算性能。此外，NPU 单元的设计采用 Chisel 开发，支持总线、缓冲区、浮点单元参数化，硬件模式和仿真模式切换，功能模块选配等功能，增强了加速器硬件平台的伸缩性以及软件仿真平台的轻量化。

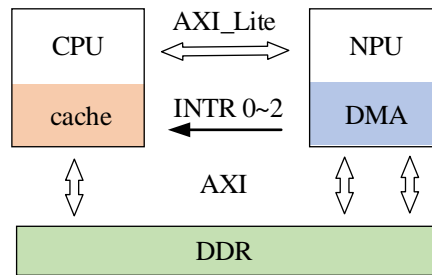


图 3-1 加速器 SoC 架构

NPU 为自行设计的加速器单元，在深度学习的加速工作中，数据传输量往往较大，在图中所示领域特定架构中，CPU 与 NPU 采用共享内存的方式进行数据交互。由于 CPU 中 Cache 引入的层次化存储结构，且 AXI 并非一致性总线，在数据的交互过程中要格外注意对 Cache 数据的 Flush 和 Invalidate。NPU 支持两条独立的 AXI 通道同时进行数据读写，每个通道的数据位宽为 128，地址位宽为 32，Outstanding 能力为 16；支持增量突发、支持非对齐传输、支持跨 4K 处理等 AXI 特性。

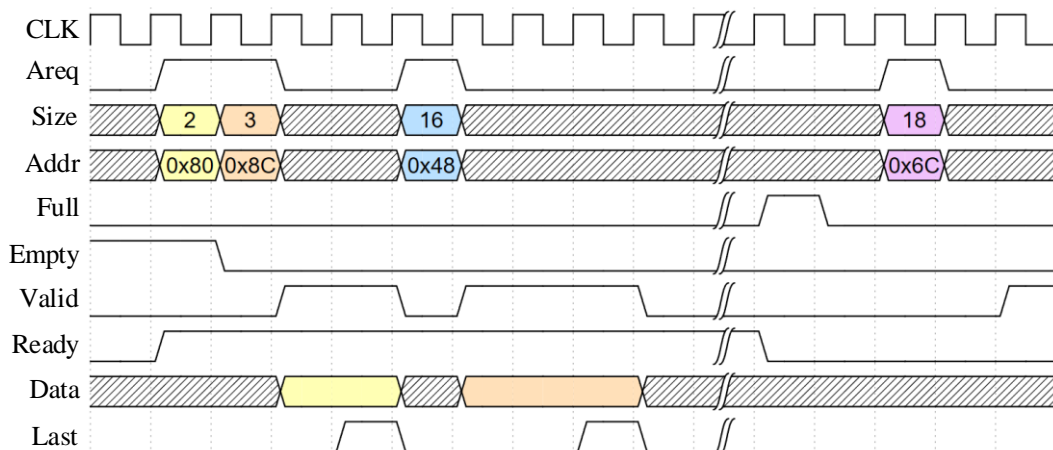




图 3-2 AXI_DMA 读事务接口时序

AXI 协议中规定了五个通道用于提升总线速率，其中两个通道用于读事务，三个通道用于写事务，信号繁多且控制复杂，为此，在 NPU 中添加 AXI_DMA 主控制器用于简化各模块的数据请求和发送。图 3-2 给出了 AXI_DMA 进行突发读事务的时序图，其中，Size 和 Addr 信号用于设置 burst 的长度和首地址，burst_len 没有限制且控制信号仅 Areq 上升沿有效；支持 Outstanding 传输，Full 和 Empty 用于指示 Outstanding FIFO 的空满状态，最大深度为 16；数据通道使用握手信号进行传输，并在每次突发传输的最后一个时钟周期拉高 Last 信号。

CPU 与 NPU 之间的控制流交互采用 AXI-Lite 总线实现，以配置寄存器的方式实现了 CPU 对 NPU 单元的控制和微量的数据传输。其中，AXI-Lite 数据位宽设置为 32，即每个寄存器大小为 4 字节；AXI-Lite 地址位宽设置为 9，以地址偏移量的形式表示，基地址在 AXI Interconnect 中设置。NPU 寄存器地址空间设置为 512Byte，允许最大寄存器数量为 128 个，实际寄存器使用 300 字节，其余地址空间保留。该地址段实现了 CPU 对 NPU 中 GEMM、ALU、POOL 等功能模块的参数配置，表 3-1 列出了 GEMM 控制寄存器的配置明细。该寄存器编号为 60，地址偏移量为 0x0F0，主要用于卷积任务和深度卷积任务的硬件初始化，具体包括 kernel 尺寸、stride 长度、padding 模式以及硬件数据通路选通等参数信息。

表 3-1 寄存器配置举例

| GEMM_CTRL_REG | |
|--|----------------------------|
| ID: 60 ADDR_OFFSET: 0x0F0 | |
| 高 16 位 [31:16] | |
|  | |
| 低 16 位 [15:0] | |
|  | |
| [0:0]: GEMM 单元使能，高电平有效 | [12:11]: padding_left 长度 |
| [2:1]: GEMM 加速算子类型 | [14:13]: padding_right 长度 |
| 0: 卷积 | [16:15]: padding_top 长度 |
| 1: 深度卷积 | [18:17]: padding_bottom 长度 |
| 其他: 保留 | [19:19]: bias 使能，高电平有效 |
| [5:3]: kernel_size | [20:20]: requant 使能，高电平有效 |
| [8:6]: stride | [21:21]: layout 使能，高电平有效 |
| [10:9]: padding_mode | [22:22]: oscale 使能，高电平有效 |
| 0: 零值填充 | [23:23]: div_ic 使能，高电平有效 |
| 其他: 保留 | [31:24]: 保留 |

软件端采用任务发放的方式进行调度，因此，引入 NPU 到 CPU 的中断信号来标识任务的完成情况。该设计中采用 3 条外部中断连接主处理器，对于 CPU 中断数量紧

张的应用场景，可以在寄存器组增加中断号标识寄存器，进而减少外部中断至一条。

3.1.2 加速器数据流

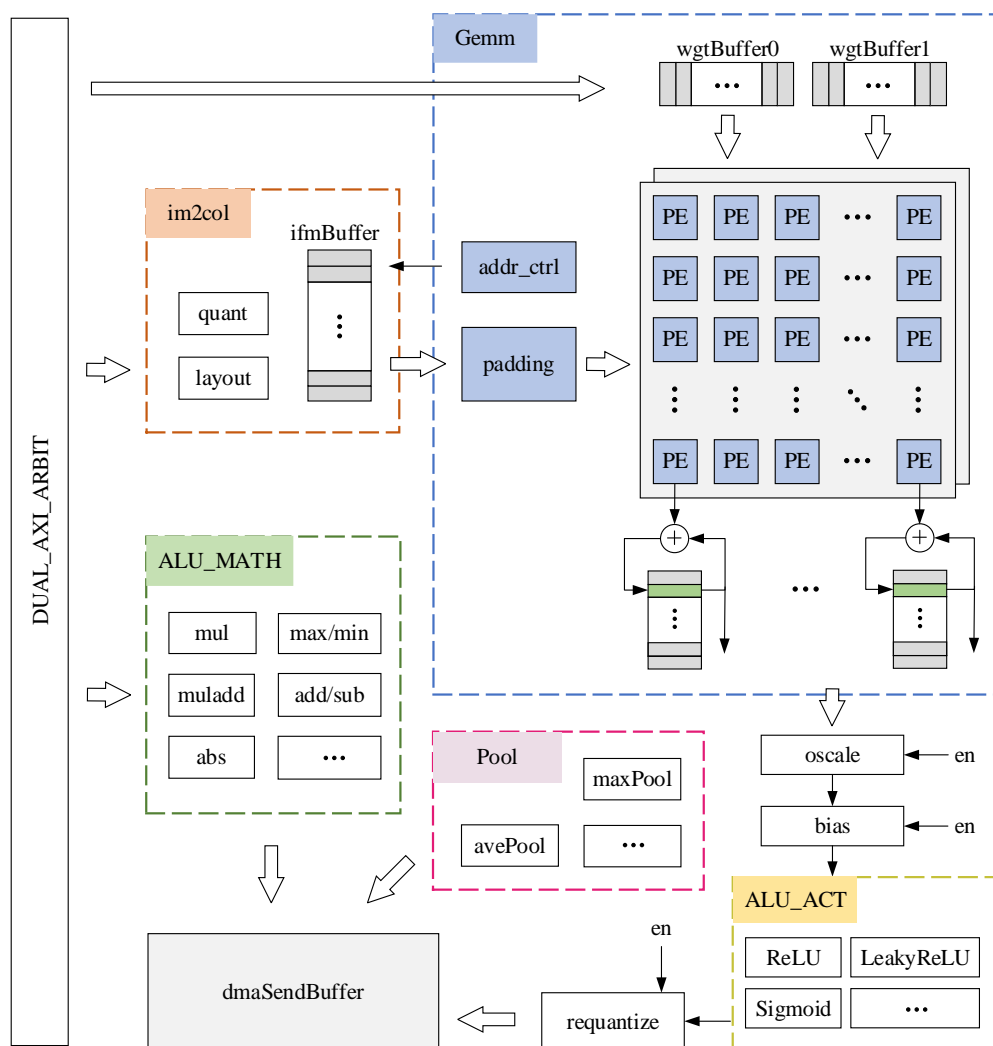


图 3-3 NPU 数据流图

NPU 整体架构数据流图如图 3-3 所示，核心计算单元由 GEMM 模块和 ALU 模块组成。GEMM 模块基于脉动阵列原理实现矩阵乘法的加速，该阵列由 32×32 个 PE 单元构成，每个 PE 单元可以在单周期内完成两组 INT8 的 MAC 计算。脉动阵列采用权重固定的数据流，最大限度实现对输入权重的复用，同时在其后端添加 ACC_MEM 存储器 and 加法器实现对部分和的累加。添加 im2col 模块以实现卷积运算和深度卷积运算到脉动阵列的映射，该模块可以同时完成输入特征图的量化、Layout 变换以及数据缓存功能。ifmBuffer 采用 DPRAM 实现，数据位宽设置为 256，总大小为 512KB。由于输入缓冲区面积较大，时序收敛困难，因此，本设计中类似的缓冲区均采用多个 Block 拼接实现。Padding 地址控制器根据初始化的参数信息和当前计算的特征图位置得到对应数据在 ifmBuffer 中的地址信息，由于脉动阵列的每个 PE 可以同时两组特征图输入进行计算，因此，同时例化两组 Padding 控制器从 DPRAM 中读出数据。权重缓冲

区采用乒乓设计，数据位宽设置为 256，单个 wgtBuffer 大小为 144KB。本设计中针对神经网络模型采用 INT8 量化，该缓冲区可以同时存储 144K 个权重参数，并在脉动阵列计算过程中，利用乒乓机制更新数据，以此来减小权重数据的传输延时。在模型参数生成的过程中，根据硬件架构和各缓冲区的特点对模型参数进行重新排列，有效避免了内存中数据不连续以及非对齐传输对整体性能的影响。

ALU 单元包括 ALU_MATH、ALU_ACT 等模块，主要负责单个特征图或多个特征图之间的逐元素运算。ALU 向量计算的数据格式为 FP32，ALU_MATH 模块支持向量加减法、乘法、混合乘加、比较、绝对值以及点积等运算，ALU_ACT 模块支持多种激活函数实现，包括 ReLU 系列、ELU 系列、Clip、Sigmoid、Swish 等。同时充分挖掘神经网络推理中的计算特点，在硬件加速器中实现了常见的算子融合，如卷积层、深度卷积层与批归一化层、激活函数层的融合处理等。为此，在硬件推理的过程中，可能出现 NPU 同时调用 ALU 和 GEMM 两大计算核心实现同一组任务的加速情景。为了进一步减小数据传输带宽压力，添加重量化模块，将 NCHW 格式排列的 FP32 特征图转换为 NHWC 格式排列的 INT8 特征图。

除此之外，NPU 硬件架构还支持最大值池化和均值池化等。为了节约硬件资源，GEMM 模块、ALU 模块和 POOL 模块共用输出缓冲区，各模块的缓冲区使用权通过优先级仲裁的方式获得。整个 NPU 与 DDR 的数据传输借助双通道 AXI_DMA 实现，但是系统中存在着众多的 DMA 请求，因此，每个 AXI_DMA 后均添加一个优先级仲裁器以处理各模块的事务请求。

3.2 核心计算单元设计

3.2.1 脉动阵列设计

在矩阵乘法的硬件实现过程中，脉动阵列凭借着其较高的数据复用率和计算速度受到广泛关注，目前，Nvidia、Google 等公司的产品中都包含着脉动阵列的微架构设计。基于脉动阵列的矩阵乘法加速方案，有两种不同的数据流格式，即权重固定和输出固定。输出固定的数据流及控制器较为简单，但数据复用匮乏使其在高吞吐量的场景应用受到限制。权重固定的数据流可以实现高效的高性能运算，对于深度学习任务来讲，可以更好的实现对权重的重利用，基于脉动阵列实现的主流 NPU 也均采用权重固定的设计。经典的脉动阵列采用层次化设计，包括 Mesh、Tile 和 PE 三层结构。PE 是整个脉动阵列计算的最小核心，Tile 层由多组 PE 堆叠而成，两者均由组合逻辑实现；Mesh 是脉动阵列的顶层连接，其由多组 Tile 和流水线寄存器组成，通过在不同 Tile 层间插入时序电路，减小布线难度，并实现数据的流水化。本设计方案中，对经典的脉动阵列结构进行改进，保留 Mesh 和 PE 的结构并对此进行数据流和控制流的优化，提高脉动阵列的 PE 利用率和数据复用率。

PE 的微架构设计如图 3-4 所示，每组 PE 中除了用于暂存权重数据的两组寄存器外，其余结构均由组合逻辑实现。该结构可以在每个时钟周期内完成两组 INT8 的乘加

运算，数学模型如公式 3-1 和 3-2 所示。

$$out_c0 = in_a0 \times reg_b + in_d0 \quad (3-1)$$

$$out_c1 = in_a1 \times reg_b + in_d1 \quad (3-2)$$

寄存器 reg_b1 和 reg_b2 用于权重数据的暂存，位宽 8bit，控制信号 p 决定了当前时刻输入权重 in_b 的暂存位置以及权重向下传播时的数据来源。当 $p=1$ 时， reg_b1 对输入的数据进行寄存，同时将原本 reg_b1 中的权重向下传播， reg_b2 的数据保持不变。控制信号 sel 决定了当前参与 MAC 计算的权重来源，当 $sel=1$ 时， reg_b1 中暂存的数据与其他输入一同进行乘加运算；反之使用 reg_b2 中暂存的数据。 reg_b1 和 reg_b2 成了一组乒乓寄存器，可以在特征图水平流动计算的过程中实现权重自上而下的更新；同时添加数据通路保证特征图可以在 PE 间水平流动，部分和在 PE 间垂直流动。

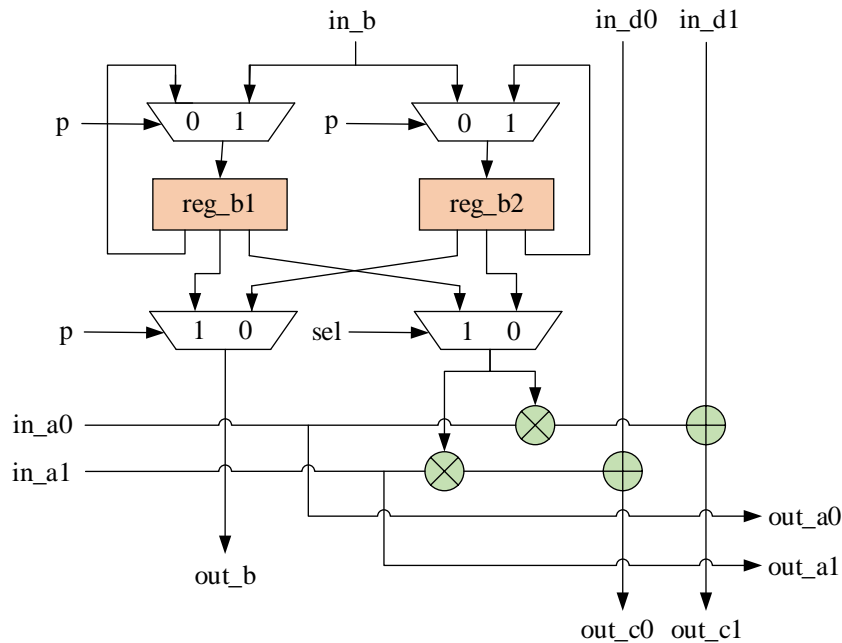


图 3-4 PE 微架构设计

在本文设计的脉动阵列实现方案中，数据流和控制流均实现了流水化。图 3-5 展示了 Mesh 微架构中数据通路的设计，整体来看，Mesh 结构由 32×32 个 PE 堆叠而成。PE 的设计包含时序逻辑构成的权重数据寄存器和组合逻辑构成的乘加运算核心两大部分，为了实现运算的流水化并改善硬件时序，在网格状的 PE 结构间添加流水线寄存器。每个周期 PE 可以完成两组乘加计算，因此，每个周期脉动阵列水平方向的每一行均需要同步输入两行特征图数据、垂直方向的每一列需支持两路部分和的流动。两路特征图的输入在时间维度上依次流入，前行数据领先后行数据一个时钟周期；在空间维度上以平行四边形排列同步流入。同理，两路部分和也在空间上沿垂直方向以平行四边形规则流出。

脉动阵列每一列输出的部分和仅为两个长度为 32 的向量点积的结果，为实现任意维度的矩阵乘法，需要对脉动阵列输出的部分和进行累加。ACC_MEM 由 32 个小尺寸

TPRAM 和 32 个累加器组成，TPRAM 的深度为 32，宽度为 32，可以暂存 32 个 INT32 数据格式的部分和。由于每个 PE 单周期内有两个部分和同时输出，此处需要两组 ACC_MEM 模块进行处理。ACC_MEM 的读写地址随时间递增且读地址位置超前于写地址位置，将当前读地址指针指向的数据与脉动阵列输出的部分和累加，同时读指针自增并将结果写回读指针之前指向的空间，即写指针位置。重复上述过程 32 次完成 ACC_MEM 中暂存数据的全部更新，假设输入矩阵的维度为 $M \times 64$ ，则完成一次矩阵乘法，需更新 ACC_MEM 全部数据 $M / 32$ 次，当完成全部累加，将结果输出至下一次计算单元。

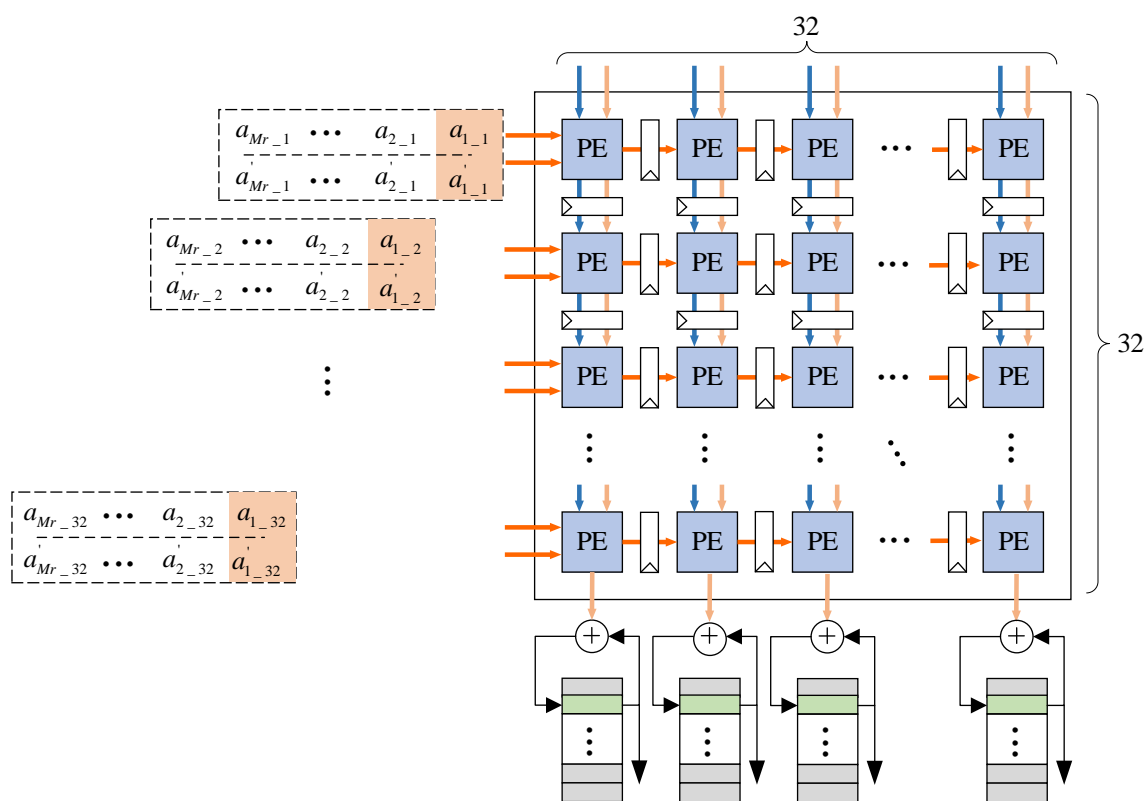


图 3-5 Mesh 微架构数据通路

为解决控制信号广播机制带来的高扇出和高延时问题，本文对 Mesh 微架构中控制信号同样进行了流水处理。控制信号 p 作为权重更新的控制信号，在空间维度自上而下按照平行四边形的规则流动，权重更新和乘加计算同时进行以掩盖权重固定模式下权重预加载的时间损耗。对于一个维度为 32×32 的基本权重块，其相邻两列的数据在脉动阵列中的水平相对位置存在一个周期的延迟，为实现不同权重块的顺利切换，需要控制 sel 信号在空间维度上沿水平方向自左向右流动。此时在时间角度， sel 信号在流动过程中呈现周期为 64，占空比为 50% 的方波信号，每 32 个时钟周期完成一次基本权重块的切换。

PE 的微架构设计中，两路 MAC 计算采用相同权重数据，不同的特征图输入。借助权重的复用和 MAC 单元的堆叠，实现了算力翻倍的目标，但这也给数据缓冲带来了更大的挑战。图 3-6 给出了矩阵乘法在脉动阵列映射中产生的分块策略，输入矩阵的

尺寸分别为 $M \times N$ 和 $P \times M$ ，输出矩阵的尺寸为 $P \times N$ 。为了简化 PE 的控制，矩阵 A 的列数和矩阵 B 的行数需保持为 32 的整数倍，矩阵 A 的行数和矩阵 B 的列数需保持为 64 的整数倍，因此，需要对不满足输入约束的矩阵进行零值填充。

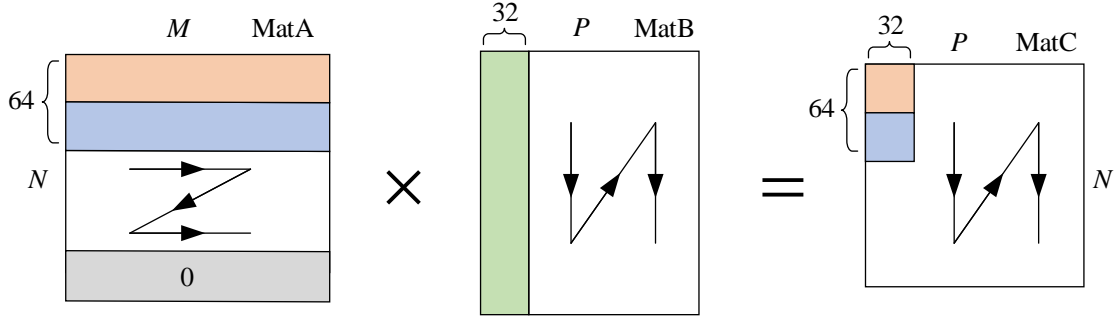


图 3-6 矩阵分块策略

在脉动阵列计算过程中，矩阵 A 同时连续输入 32×2 路行向量，矩阵 B 同时连续输入 32 路列向量，计算完成后得到 32×64 的数据块。完成两组 $(32, M) \times (M, 32)$ 矩阵乘法所需的时间如公式 3-3 所示，align 为向上对齐二元函数以实现自变量 x 对齐到 n ，具体行为如公式 3-4 所示。

$$periods = \text{align}(M, 32) + 32 \quad (3-3)$$

$$\text{align}(x, n) = (x + n - 1) \& -n \quad (3-4)$$

为了增加对矩阵 B 的数据复用，需首先完成对矩阵 A 的遍历；其次更新矩阵 B 的后续 32 路列向量，并继续对矩阵 A 进行遍历；迭代这一过程，自上而下、自左而右的得到输出矩阵 C。通过对脉动阵列底层逻辑的优化，实现了两个矩阵数据的连续不间断输入，完成一组 $(N, M) \times (M, P)$ 矩阵乘法所需的时间如公式 3-5 所示。

$$all_periods \approx \text{align}(M, 32) \times \text{align}(N, 64) \times \text{align}(P, 32) \quad (3-5)$$

3.2.2 向量 ALU 设计

硬件加速器中 GEMM 单元负责矩阵的加速，ALU 单元负责向量的加速，两大计算核心相互配合以贡献硬件平台大部分算力。ALU 单元由 ALU_MATH、ALU_ACT 和 ALU_INNERPROD 三部分组成。从具体实现的算子类型角度，ALU_MATH 支持神经网络推理过程中常用的逐元素计算和二进制操作，如 Scale、Bias、Threshold、Dropout、Abs 等算子层；ALU_ACT 支持 CNN 主流的激活函数硬件加速，如 ReLU、ELU、Sigmoid、Swish 等系列激活函数；ALU_INNERPROD 支持任意维度向量内积的计算，主要用于硬件全连接层的加速。

图 3-7 给出了 ALU 数据流的基本结构，其中所有数据通路的选择均可由软件寄存器配置。控制信号 op 决定了 ALU 当前的运算类型以及操作数的个数，激活函数等运算单元操作数一般设置为 1，而 Scale 和 Bias 等算子操作数设置为 2。ALU 可以从两条数据通路获取输入向量，第一条通道通过 AXI 总线读取内存数据并进行硬件加速，第

二条通道的数据流源于 GEMM 单元，以支持常见的算子融合。根据算子操作数的个数，决定是否启动 src_mem 缓冲区，该缓冲区深度和宽度均为 128，数据位宽与 AXI 总线的读写数据位宽保持一致。ALU 单元支持任意长度向量的逐元素运算，考虑到片上缓冲区和 AXI 突发长度的限制，添加 axi_ctrl FSM 以实现分批次计算。ALU 的结果写回也包含多条数据通路，对于 CPU 直接派发的任务其结果往往通过 axiSendFifo 直接写回主存，但对于向量点积的计算则直接通过寄存器组写回，并同时触发 ALU 中断。当 ALU 配合 GEMM 进行融合计算时，输出数据流返回至重量化等模块，且不会触发 ALU 中断。

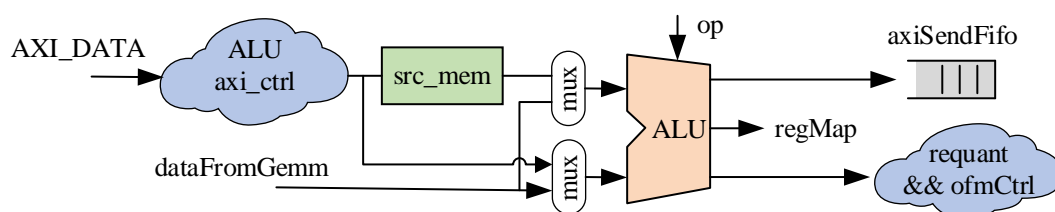


图 3-7 ALU 数据流基本结构

ALU 任意长度向量的分批次计算通过内存数据加载状态机实现，如图 3-8 所示，主要包含 6 个状态。AXI 数据传输包含产生突发请求和等待传输完成两个过程，对于操作数为 2 的计算事务，优先满足 src_mem 的数据需求。

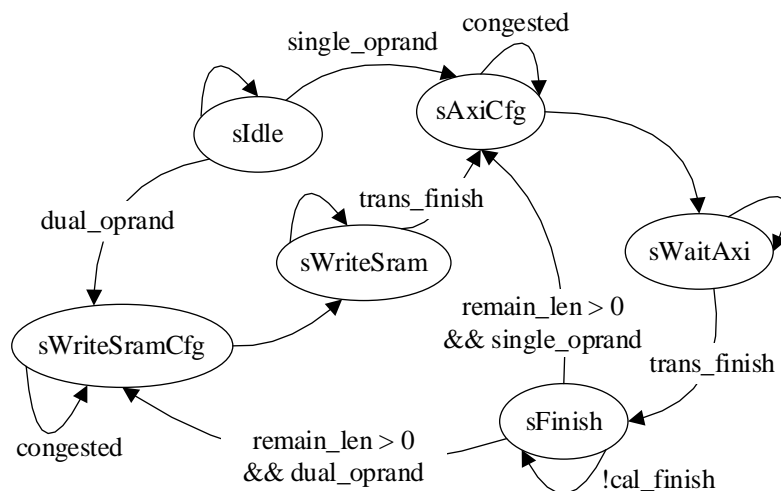


图 3-8 ALU 内存数据加载状态机

在内存数据加载过程中，ALU 支持数据传输和计算的同步处理以掩盖传输延时并自动更新参数完成下一批次的处理，整个加速过程仅需 CPU 发起任务请求并等待事务完成中断即可，有效降低了软件驱动移植和应用程序开发的难度和工作量。考虑到后期维护以及资源面积等实际因素的限制和影响，ALU_MATH 和 ALU_ACT 的微架构设计均深度耦合资源复用理念。图 3-9 展示了 ALU_MATH 的资源复用设计，Mux 的选通由软件控制实现，支持向量与向量、向量与标量的加减法、乘法、乘加和比较等运算。ALU_ACT 采用最小二乘法分段拟合，并根据激活函数奇偶性等特点进一步提高拟


```

    result
  }

  implicit def Float2Bits(x: Float): UInt = x.bits //隐式转换
}

```

3.3 im2col 单元设计

3.3.1 im2col 硬件实现方案

im2col 算法常常搭配 GEMM 以实现卷积等深度学习算子的硬件加速，该算法核心思想是**将卷积等运算转换为二维矩阵乘法**，进而实现卷积、深度卷积等算子的硬件加速。图 3-10 从数据结构角度给出了卷积的映射方案，其中该示例采用 3×3 卷积，stride 为 2，上下左右四个方向的 Padding 长度均为 1，并采用零值填充。

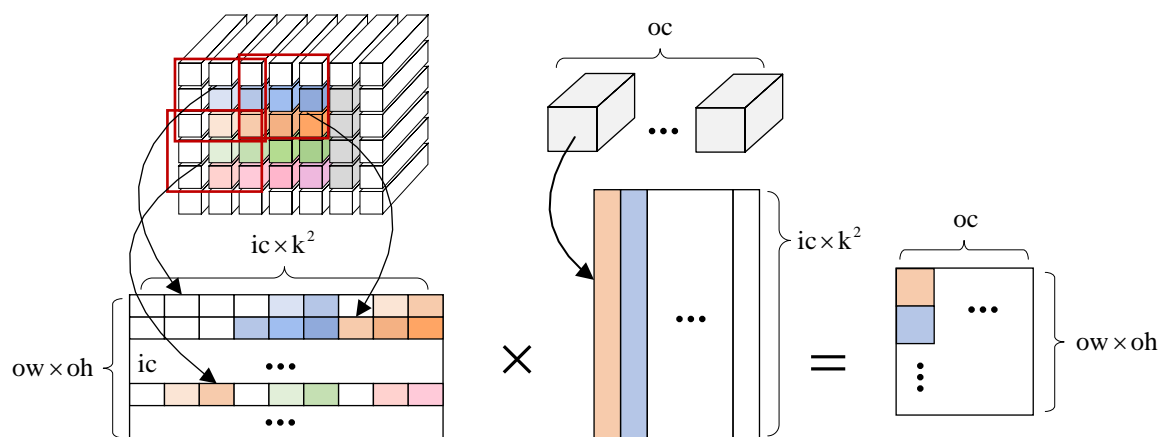


图 3-10 卷积算子的脉动阵列映射方案

在映射的具体实现中，输入特征图根据卷积窗的位置提取 $ic \times kw \times kh$ 个数据，依次按照 ic 、 kw 和 kh 方向排列成行向量。随着卷积窗位置的移动，实现对输入特征图的遍历，得到 $ow \times oh$ 个行向量。其次，单个卷积核也按照 ic 、 kw 和 kh 方向排列成列向量， oc 个卷积核拼接成权重矩阵。当卷积核的 stride 小于 kernel 长度时，会产生较大的数据重叠，重叠的数据块在图 3-10 中以相同颜色表示，白色的数据块表示零值。对于 GPU、CPU 等通用编程平台采用 SIMD 等数据级并行方案，借助 BLAS 库可以实现矩阵乘法的快速计算，无需考虑数据重叠的影响。本文的 Mesh 微架构设计中借助脉动阵列和矩阵分块策略实现了通用矩阵乘法，在面积资源昂贵的 ASIC 领域，无效的数据缓冲被严格限制，因此硬件化过程中必须对传统的转换算法进行改进。

在 im2col 的硬件方案具体实施中，权重量化离线进行。因此，神经网络的边缘端部署过程中可以预先对权重数据进行重排列，在软件栈进行模型加载的过程中直接进驻内存，并在模型推理中采用软拷贝的方式进行访问。特征图的量化在线进行且硬件片上缓冲区面积有限，因此，需设计特定的分块策略以及硬件数据排列方案以充分利

用片上资源。图 3-11 展示了输入特征图采用 im2col 算法展开的硬件实现原理，分为 Layout 变换和 Padding and Reorder 两个步骤。

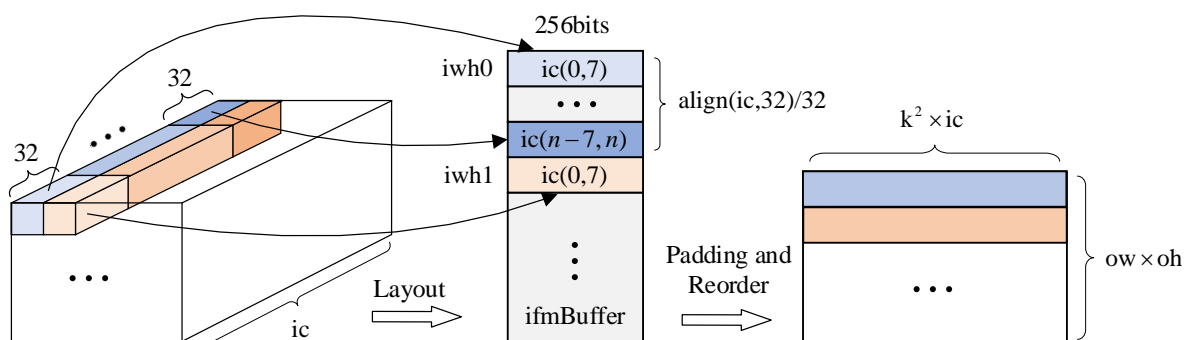


图 3-11 im2col 硬件实现方案

Layout 变换将常见的 NCHW 格式排列的特征图转换为 NHWC 格式排列，暂存在 ifmBuffer 中。ifmBuffer 的数据位宽为 256，输入特征图采用 INT8 量化，因此缓冲区每个地址对应 32 个数据，在具体的数据排列过程中，输入缓冲区优先排列 ic 维度，其次排列 iw 和 ih 维度。若输入通道数不为 32 的整数倍，则需要对特征图补零处理。除此之外，ifmBuffer 暂存的输入特征图在空间上没有数据重叠，且 NHWC 的数据排列极大简化了硬件 Padding 的流程。Padding 和 Reorder 模块根据 ifmBuffer 缓存数据结构特点以及脉动阵列输入数据流需求，设计双通道地址控制器，并按照 im2col 展开矩阵的排列位置同步读取 32×2 个行向量。迭代这一过程，完成输入特征图的动态 Padding 以及 Mesh 的数据流准备工作。该模块设计过程中，针对数据流进行多角度优化，采用双通道地址控制器实时进行数据读取，从而保证无需额外的缓冲区。

3.3.2 Layout 模块设计

NHWC 和 NCHW 两种数据格式的在深度学习领域广泛应用，其中 NCHW 格式 C 维度排列在前，相邻的两次卷积任务必须等待前一层计算全部完成才可以开始下一层的计算，而 NHWC 的排列方式局部性更好，节省临时内存使用和访存时间，提升计算性能。除此之外，不同 NPU 设计厂商也针对性的设计张量的存储方式以满足混合精度计算和高性能计算的需求，如晟腾 AI Core 采用 NC1HWC0 的五维 Layout 以充分适配硬件微架构。本设计的软件栈中采用 NCHW Layout，硬件加速以 NHWC Layout 为主，因此添加 Layout 硬件模块实现不同张量存储方式的转换，如图 3-12 所示。

参数生成状态机根据当前的特征图读取位置生成 AXI 的突发地址和长度以及坐标信息存入参数队列中。Layout 模块中包含两个量化和转换单元，与双通道 AXI 相配合使用，通过优先级仲裁器来获取 AXI 配置信息和特征图计算位置。量化电路负责将输入的单精度浮点数转化为 INT8 进行计算，浮点的输入特征图与量化因子直接相乘，得到量化后的特征图。转换电路由 32 个小 RAM 组成，每一个 RAM 的深度均为 16，位宽为 32。单个 RAM 内部存储单通道连续 64 个 wh 平面的数据，不同 RAM 间存储不同通道的数据，填充全部 RAM 至少需要产生 32 次 AXI 突发传输请求。输入数据分配

伪代码 3-1 硬件 Padding 和重排序实现方案

```

for(int oh=0; oh<OH; oh++) //遍历 OFM 全部位置
    for(int ow=0; ow<OW; ow++)
        for(int kw=0; kw<KW; kw++) //遍历卷积窗
            for(int kh=0; kh<KH; kh++){
                iw_pd_cnt = ow*stride+kw; //计算当前的特征图位置
                ih_pd_cnt = oh*stride+kh;
                for(int ic_idx=0; ic_idx< align(IC,32)/32, ic_idx++){
                    if(iw_pd_cnt < pd_left || ih_pd_cnt < pd_top ||
                       iw_pd_cnt > IW + pd_left || ih_pd_cnt > IH + pd_top)
                        gemmdata = 0; //边缘处零值填充
                    else{
                        iw_cnt = iw_pd_cnt - pd_left; //计算特征图实际位置
                        ih_cnt = ih_pd_cnt - pd_top;
                        baseaddr = (ih_cnt *IW + iw_cnt)*align(IC,32)/32; //计算缓冲区 iwh 位置
                        gemmdata = buffer[baseaddr+ ic_idx]; //输出 iwh 处全部 ic
                    }
                }
            }
    }
}

```

3.4 大尺寸特征图分块策略

在前文的分析中，依次对矩阵乘法的脉动阵列映射方案和卷积等算子的矩阵乘法映射方案进行讨论，兼具高效和灵活的特点。但是在实际的硬件开发设计过程中，算法的迁移往往受到资源、面积、时序、成本等多种因素的限制。为了实现硬件性能和资源面积的均衡，片上缓冲区的设计尤为重要，表 3-2 汇总了 ALU 和 GEMM 硬件实现中各缓冲区的类型、尺寸等参数。大尺寸缓冲在底层实现中采用了多 Bank 堆叠设计，缓冲区的位宽、深度和数量仅在宏观上进行统计。在大尺寸特征图的映射过程中，设计了多种数据分块方案，从 ih、ic、oc 等多角度进行任务分割，在有限的硬件资源中释放最大的性能和灵活性。

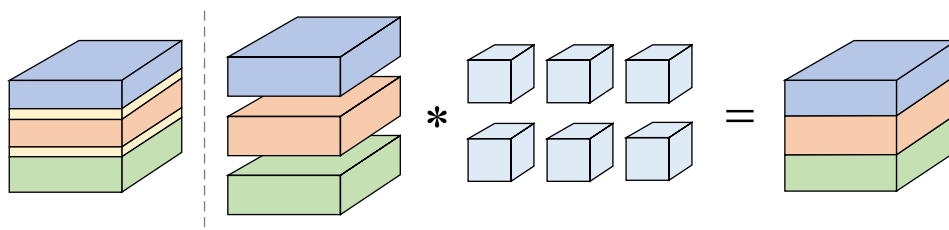
表 3-2 片上缓冲区汇总

| 名称 | 类型 | 位宽 | 深度 | 数量 | 尺寸 |
|-----------|----|-----|-------|----|-------|
| ifmBuffer | DP | 256 | 16384 | 1 | 512KB |
| layoutMem | SP | 32 | 16 | 64 | 4KB |
| wgtBuffer | SP | 256 | 4608 | 2 | 288KB |

表 3-2 片上缓冲区汇总（续）

| | | | | | |
|------------------|----|-----------|------|----|--------|
| oscaleAndBiasMem | SP | 128 | 256 | 2 | 8KB |
| accMem | TP | 29 | 32 | 64 | 7.25KB |
| ofmBuffer | TP | 128 | 256 | 64 | 256KB |
| aluBuffer | SP | 128 | 128 | 2 | 4KB |
| axiSendBuffer | TP | 128/32/32 | 1024 | 2 | 48KB |

本设计中输入缓冲区分配大小 512KB，当输入特征图尺寸过大时，需要进行分块处理，如图 3-13 所示。该分块策略由软件实现，根据 iw 、 ic 以及 $ifmBuffer$ 大小计算支持缓冲的最大深度 ih_{max} 并将输入特征图在 h 方向划分成多个数据块，进而实现将单个大尺寸卷积任务分割成多个小尺寸卷积任务。当卷积核垂直方向步进小于 $kernel_h$ 时，相邻的两次任务间存在输入数据重叠，增加访存时间；但重叠数据较少、对系统整体性能影响忽略不计。软件部分特征图存储采用 NCHW 张量格式，在进行任务分割过程中不改变其数据排列方式，因此仅需设置每个输入输出数据子块的地址偏移即可实现 NPU 和 DDR 的数据交互。除此之外，该分块策略具有普遍性，对卷积、深度卷积等算子均适用。

图 3-13 输入特征图 h 方向分块示意

权重缓冲区采用乒乓设计，单个缓冲区分配 144KB，最大存储 $k^2 \times ic \times 32$ 个 INT8 数据，其中 32 与 Mesh 的尺寸紧密相关。当输入特征图的卷积窗尺寸或者通道数过大时，权重缓冲区无法缓存全部数据，特征图和权重的分块势在必行。由于卷积窗在输入特征图 wh 平面内计算深度耦合且为了实现特征图的数据复用，本设计中采用了同时对输入特征图和权重在 ic 方向进行分块的策略，如图 3-14 所示。首先根据权重缓冲区的大小计算硬件支持的最大输入通道数，并将输入特征图和权重划分成 n 个不同的数据子块。其次，每个子块完成对应的卷积计算，得到 n 个维度为 $ow \times oh \times oc$ 的输出特征图。由于每个卷积窗的位置沿 ic 方向均进行累加操作，故对 n 个部分和结果进行累加得到最终的输出。从硬件部署角度，输入特征图和权重 ic 方向的分块策略需要 ALU 辅助 GEMM 实现。

以上两种分块策略足以支持大部分卷积和深度卷积计算的硬件加速场景，出于硬件功能完整性角度的考虑，添加图 3-15 所示的权重分块策略将加速范围推广到任意输入维度。在图 3-3 所示的 NPU 整体数据流图中，GEMM 的输出依次流入反量化模块和偏置模块，其对应的反量化系数和偏置系数的个数等于卷积核的个数，即输出特征图

的通道数。oscaleAndBiasMem 数据位宽为 128，深度为 256，因此，该缓冲区支持最大 1024 个通道的系数缓存；当卷积核的个数超过 1024 时，需对权重沿 oc 方向进行任务分割。该分块策略同样基于软件实现，在特征图的内存模型中，通道维度的排列位于 w 和 h 之后，因此，只需对每次任务输出特征图的地址偏移进行配置即可完成不同任务输出的拼接。

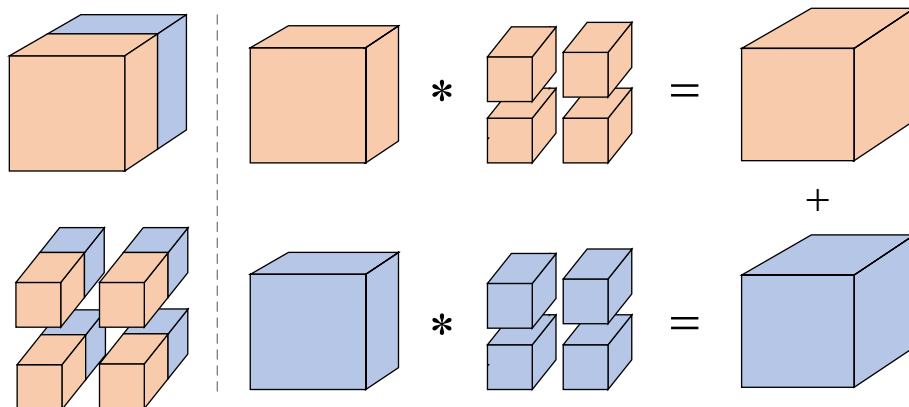


图 3-14 输入特征图和权重 ic 方向分块示意

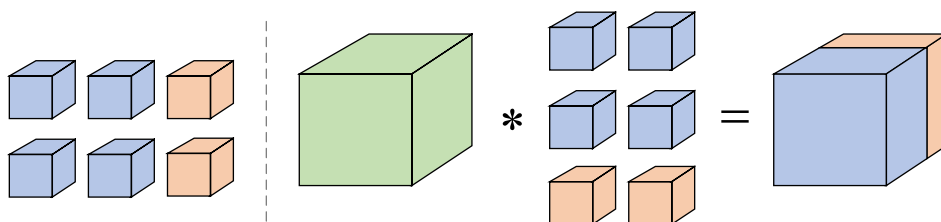


图 3-15 权重 oc 方向分块示意

3.5 算子融合

在神经网络硬件部署过程中，推理延时主要由访存时间和计算时间两部分组成。算子融合技术主要针对访存延时进行优化，常见做法是**将多个算子层进行级联，一次访存即可完成多个算子的计算**。在本文的设计方案中，实现了卷积、深度卷积与反量化、批归一化、激活函数之间的层融合，大幅度缩减层间的访存延时、提高推理性能。随着神经网络层数的加深，数据分布逐渐偏移标准的正态分布，在训练过程中极易出现梯度消失和梯度爆炸的现象，为此，常常在卷积层和深度卷积层后添加批归一化进行数据规范。公式 3-6 和公式 3-7 给出卷积层和批归一化层的数学模型，其中 ω 、 b 分别表示卷积层的权重和偏置， μ 和 σ^2 分别表示批归一化层输入的均值和方差， γ 和 β 通过训练得到以实现将规范到 $N(1,0)$ 的标准正态分布仿射变换到 $N(\gamma,\beta)$ ， ε 通常设置为一个较小的常量，避免发生除零异常，增强数据稳定性。

$$y = \omega x + b \quad (3-6)$$

$$y = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta \quad (3-7)$$

将公式 3-7 代入到公式 3-6 中，得到卷积和批归一化层融合后的数学表达，如公式 3-8 所示。其中， $\frac{\omega \times \gamma}{\sqrt{\sigma^2 + \varepsilon}}$ 表示融合后的权重， $\frac{b - \mu}{\sqrt{\sigma^2 + \varepsilon}} \gamma + \beta$ 表示融合后的偏置。而激活函数主要应用于线性运算之间，以去除特征信息中的冗余，增强网络的特征提取能力。由于非线性激活函数的计算采用逐元素的方式进行，在硬件映射的过程中可直接对输出特征图进行激活处理。

$$y = \frac{\omega \times \gamma}{\sqrt{\sigma^2 + \varepsilon}} x + \left(\frac{b - \mu}{\sqrt{\sigma^2 + \varepsilon}} \gamma + \beta \right) \quad (3-8)$$

算子融合部分的硬件微架构如图 3-16 所示，oscale 模块和 bias 模块用于反量化和偏置的计算，实现了批归一化层的融合，**相关系数在任务发起时优先从 DDR 加载到 oscaleAndBiasMem 缓冲区中**。激活函数模块与前级运算单元级联，借助 ALU 实现了卷积、深度卷积和激活函数的融合计算，并**根据激活函数的复杂度分为直接计算和最小二乘逼近两种形式**。此外，算子融合部分添加硬件重量化模块，若算子层间可以借助重量化优化，则写回数据格式为 INT8、NHWC Layout 的特征图，每个特征图仅有一个反量化系数，并通过软件寄存器进行设置；反之，写回数据格式为 FP32、NCHW Layout 的特征图。

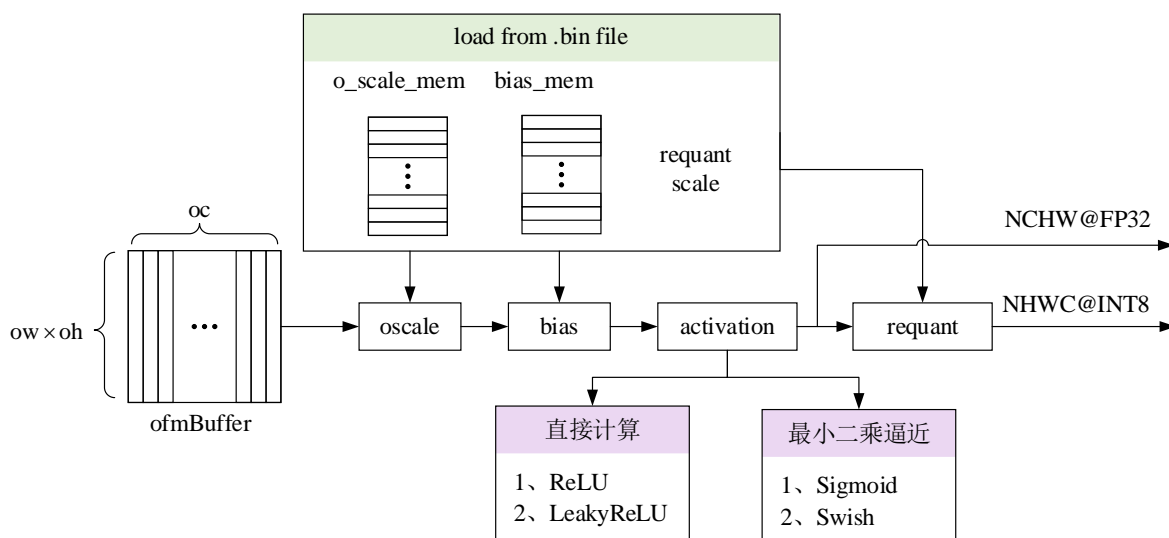


图 3-16 算子融合微架构

3.6 重量化策略

3.6.1 重量化原理

在深度学习领域，模型的推理效率和运行成本一直是工业应用的重点研究对象。在嵌入式设备和移动设备部署过程中，由于计算资源和存储空间的限制，模型量化作为一种有效的模型压缩和加速技术被广泛应用。量化技术可以在保持或者轻微牺牲模型精度的约束下，大幅度降低神经网络的推理成本。

从训练方法角度可以将量化技术概括为训练后量化和量化感知训练，其中训练后量化方法简单高效被工业界大量采用。从量化策略的角度来看，量化技术可以分为均匀量化和非均匀量化。均匀量化通过等间隔地映射浮点数到量化值，简化了量化过程，同时能够较好地匹配硬件加速器的计算模型，因此在硬件上的实现更加高效。相反，非均匀量化根据数据分布的特性进行量化，理论上可以保留更多的信息，从而可能获得更高的精度。但具体实现更为复杂，特别是在硬件上的支持不足，使得产品的落地难度增加。进一步地，均匀量化可以细分为对称量化和非对称量化。对称量化采用相同的量化区间对正负值进行量化，简化了量化过程并有助于在硬件上更高效地实现。非对称量化则允许正负数有不同的量化区间，这在一定程度上可以提高量化的灵活性和精度，但相应地也会增加计算和实现的复杂度。在本文的设计方案中，我们采用了训练后量化结合均匀对称量化的策略，旨在构建一套高效轻便的软硬件系统。

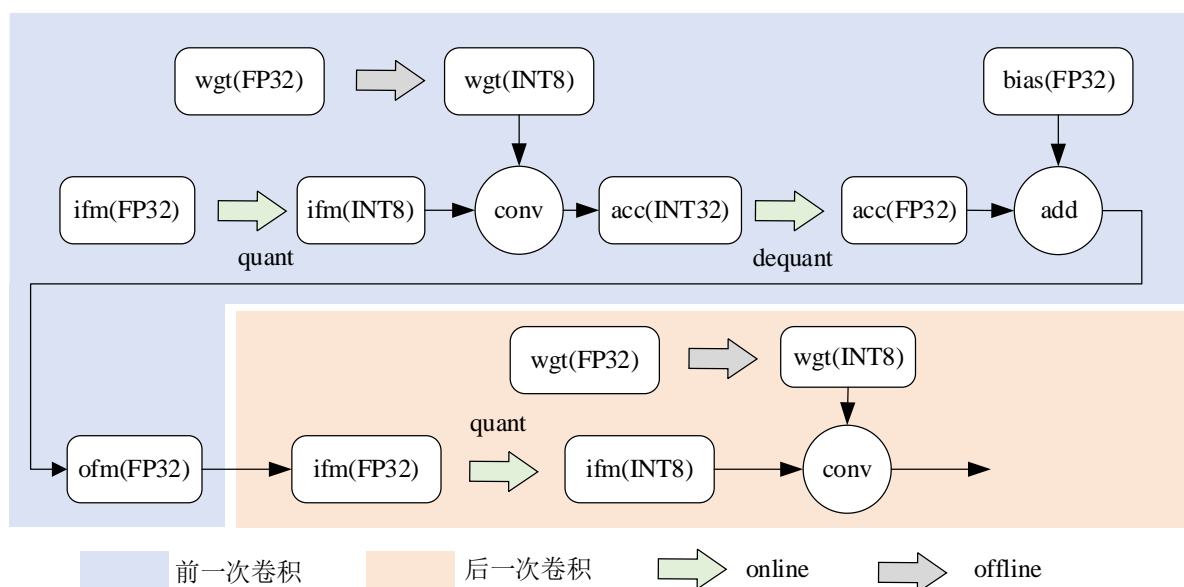


图 3-17 相邻卷积层未优化时数据流格式

在 PC 端或云端的神经网络训练过程中，常采用单精度浮点的数据存储格式以保证模型精度，但是在边缘端和移动端设备部署过程中，常采用 INT8 量化的方案，相邻卷积层的数据流格式如图 3-17 所示，其仅针对卷积层进行了量化处理。输入特征图的量化在线进行，该步骤将 FP32 格式的数据量化为 INT8 格式，一般来讲，单个特征图量化系数仅有一个且量化过程会有一定的精度损失。权重的量化可以离线进行，预先在 PC 端进行数据量化和重排以适应硬件架构。硬件卷积中累加器位宽设置为 32，以防止数据溢出，累加结果经过反量化转换回 FP32 格式，一般反量化系数的数量与卷积核个数相同且该过程没有精度损失。将反量化的输出与偏置相加，得到 FP32 格式的输出特征图并写回内存。

在两个相邻的卷积层结构中，前一层的输出和下一层的输入均为单精度浮点，访存延时较大。重量化策略将下一层的量化提前到前一层的末尾进行，如图 3-18 所示。此时，相邻的两个卷积层中，数据传输采用 INT8 格式进行，节省了四分之三的传输带

宽，大幅度提高推理速度。在卷积神经网络中，重量化策略可以应用在更多的算子层间，例如 YOLOv3_Tiny 骨干中卷积层和池化层相间叠加，若硬件支持 INT8 池化即可实现卷积和池化的重量化；以及包含大量残差块的残差网络，仅需硬件 ALU 支持 INT8 数据格式即可实现高效计算。在具体实施过程中，重量化主要重点和难点在于软件端神经网络计算图的解析和参数提取。由于本人的时间和精力有限，本文的设计方案中仅针对相邻卷积层、相邻深度卷积层以及卷积层和深度卷积层间应用重量化策略。

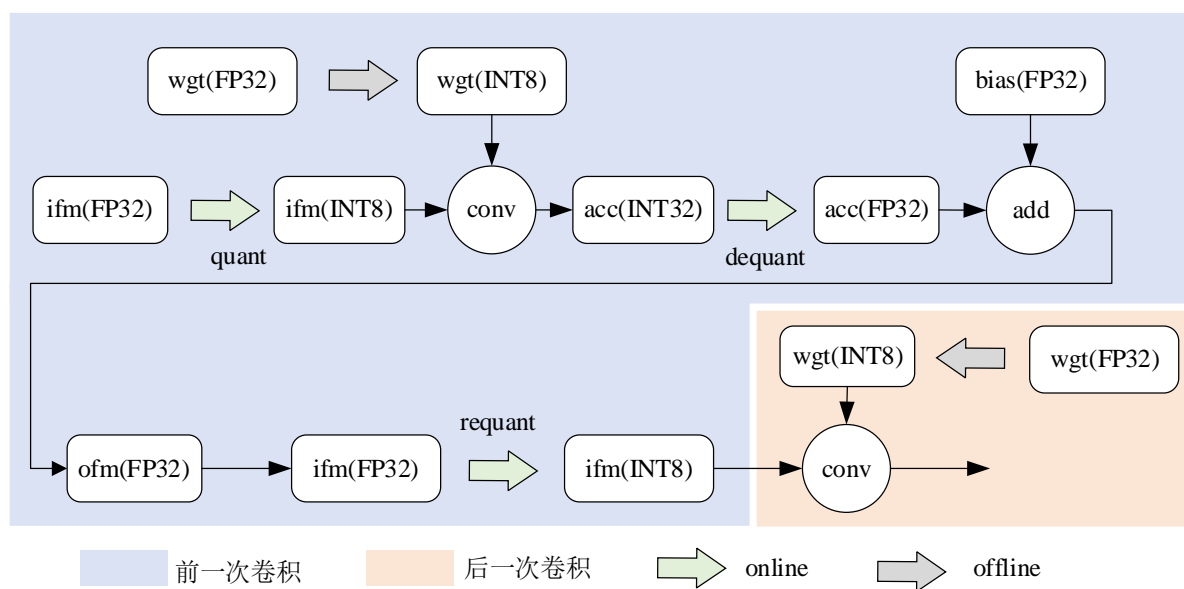


图 3-18 相邻卷积层重量化后数据流格式

3.6.2 重量化硬件设计

在本文的设计方案中为了降低系统开发的复杂度，在硬件架构中，特征图采用了 NHWC Layout 的数据存储格式，而软件栈为了保持高度的灵活性和移植性，临时特征图采用了 NCHW Layout 的数据存储格式。在图 3-16 所示算子融合架构中可见，重量化模块需要承担下一层算子的量化以及数据存储格式转换的任务，该模块的设计如图 3-19 所示。

反量化和量化过程相似，单个特征图仅有一个反量化系数并可通过软件寄存器配置，多个浮点乘法器和数据格式强制转换器并联即可实现多路数据并行计算。数据存储格式转换电路设计借鉴了脉动阵列的设计思想，其转换阵列由 32×32 组寄存器网格状级联，与脉动阵列相比，转换电路的最小单元仅具备数据暂存能力，而不具备计算能力。转换阵列支持 32 个任意长度向量的转置运算，通过水平和垂直输入方向的切换，保证转换过程中不存在空泡。当水平方向连续输入 32×32 个数据时，数据自上而下流动填满整个阵列；之后调整输入方向切换为垂直，继续输入 32×32 个数据，在上一轮填充的数据自左而右流出阵列的同时完成当前轮的阵列填充。最后通过 32 个 Mux 对水平方向和垂直方向的输出数据进行选择。重复这一过程，完成对 32 个向量的转置，需要注意的是，在执行到最后一个数据块时，需要继续填充 32×32 个数据的零值块使得转换阵列中的数据全部流出。继续迭代这一过程，即可实现矩阵的转置，进而将

NHWC Layout 的特征图转换为 NCHW Laout 的特征图。

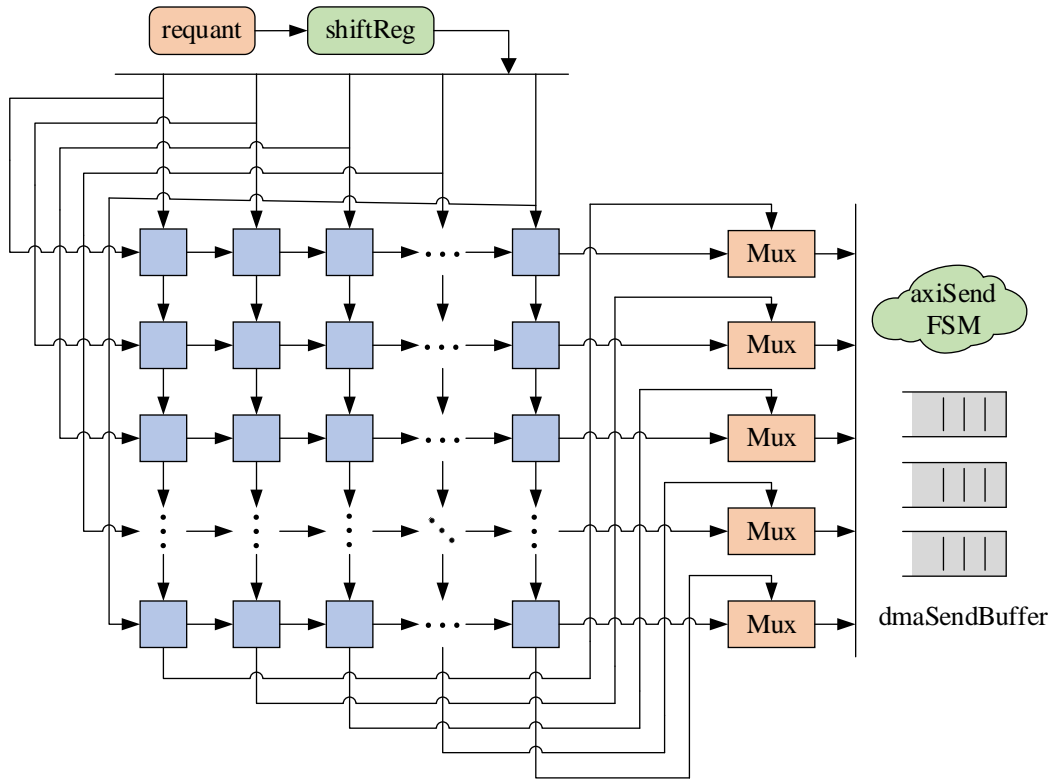


图 3-19 重量化硬件微架构

源码 3-2 展示了重量化微架构中转置阵列的 Chisel 实现，相比于传统的 Verilog 及 SyetemVerilog 开发，采用 Seq 序列结构的 fill 方法批量例化并结合 for 循环进行模块间连线，简洁高效，极大降低了模块例化和连线复杂度。此外，Array、List、Map、Tuple 等集合结构也在设计中经常使用。转置阵列模块实现了结构参数化设计，支持任意尺寸的转置阵列生成，灵活性和通用能力显著。在具体的电路设计中，每个 transposeCell 均包含两组数据输入输出端口以支持水平和垂直方向的数据流动。转置阵列内部呈网状逐级连接，左侧和顶部的 transposeCell 输入源于外部数据，右侧和底部的 transposeCell 输出至后续单元模块，并通过 Vec 方式将水平和垂直方向的线网输出进行打包处理。该转置阵列支持 N 行向量的连续输入，并根据 add_zero_valid 信号进行零值填充，以输出阵列存储的全部数据。

源码 3-2 转置阵列 Chisel 实现

```
val memArray = Seq.fill(N, N)(Module(new transposeCell)) //定义转置矩阵
for (j <- 1 until N) {
  for (i <- 1 until N) {
    memArray(i)(j).io.i_data0 := memArray(i - 1)(j).io.o_data0 //阵列内部垂直连线
    memArray(i)(j).io.i_data1 := memArray(i)(j - 1).io.o_data1 //阵列内部水平连线
  }
}
```

源码 3-2 转置阵列 Chisel 实现（续）

```

}
val h_out = Wire(Vec(N, UInt(8.W)))
for (i <- 0 until N) { //阵列左侧数据输入
  memArray(i).head.io.i_data1 := Mux(add_zero_valid, 0.U, datagp_t(N - i - 1))
  if (i > 0)
    memArray(i).head.io.i_data0 := memArray(i - 1).head.io.o_data0 //阵列左侧连线
  h_out(i) := memArray(i)(N - 1).io.o_data1 //水平数据输出
}
val v_out = Wire(Vec(N, UInt(8.W)))
for (j <- 0 until N) { //阵列顶部数据输入
  memArray.head(j).io.i_data0 := Mux(add_zero_valid, 0.U, datagp_t(N - i - 1))
  if (j > 0)
    memArray.head(j).io.i_data1 := memArray.head(j - 1).io.o_data1 //阵列顶部连线
  v_out(j) := memArray(N - 1)(j).io.o_data0 //垂直数据输出
}

```

3.7 本章小结

本章针对硬件 NPU 的架构设计进行详细阐述。首先介绍了 NPU 的整体数据流架构及相应接口定义，其次详细展示了 GEMM、向量 ALU 和 im2col 等模块的电路设计细节以及硬件实现方案。最后，针对 CNN 硬件加速提出了一系列算子映射和优化技术，如多角度大尺寸特征图分块、算子融合、重量化等。

4 模型部署工具链及验证平台设计

4.1 面向深度学习的技术路线

随着领域特定技术的飞速发展，各种面向深度学习应用的硬件加速芯片迅速迭代，包括各大科技巨头以及初创企业均希望在庞大的市场中占据一席之地。指令集作为软硬件交互的桥梁，硬件平台虽然提供了足够的计算能力支持，但是软件编译器的缺失和生态环境的不足严重限制了硬件的进步。

在 GPGPU 和 NPU 领域，底层编程框架的实现在不同厂商采用的方案中差异较大。软件人员常采用 CUDA 或者 OpenCL 等编程框架进行开发，其中 CUDA 作为一个并行的计算平台和编程模型，并提供一系列的优化工具和线性加速库，允许开发者应用 NVIDIA 设备进行高性能计算；OpenCL 提供了一个开放的标准，广泛应用于跨平台、跨厂商的并行计算场景。除此之外，工业界和学术界也采用了许多自定义的编程框架，呈现百鸟争鸣的态势。

为了方便应用层算法的快速部署，深度学习编译器和推理引擎逐渐成为研究热点。TVM 提供了一个统一、灵活、开源的机器学习编译器框架，基于端到端的优化策略和自动调优调度方案实现了多前端、多硬件的支持和适配。MLIR 旨在解决现有编译器和转换工具链中存在的互不兼容、不可扩展等问题，该框架可以大幅度简化和优化软硬件编译工作，有着深度学习编译器一统江山的潜力。而深度学习推理引擎由于其开发难度较低、硬件部署简单的特点，在边缘端、移动端甚至云端均大量应用，如 TensorFlow Lite、ONNX Runtime、CoreML、TensorRT、OpenVINO、ArmNN 等。

在面向人工智能领域的技术路线中，软件工具链主要包括量化、优化、算子分解和代码生成四大部分，如图 4-1 所示。目前，主流的量化方式主要包括两种，分别是训练后量化和量化感知训练。顾名思义，训练后量化就是在训练好的浮点模型的基础上进行量化微调，对于较深的网络结构，这种方案可能的表现不太理想，不过在实际模型边缘化部署的问题上，往往受限于算力，大都采用轻量级网络，层数也不会很深。量化感知训练，就是在训练的过程中加入量化误差，目前学术界出现了各种各样的算法，但是训练起来都比较慢。两相比较、本设计中采用了训练后量化的量化方案，这也是目前工业界普遍采用的方法。

在对模型量化之后，需要对模型进行优化，分为图优化和张量优化两大类，具体的优化方法包括算子融合、Layout 变换、调度算法等，其中以 TVM 最具代表性。当然，优化策略对于一个模型部署来讲，不是一个必选项。算子分解旨在将复杂计算分解成更加简单高效的计算模型进而提高计算效率、降低内存消耗，包括特征图分块、矩阵分解等算法。在代码生成阶段根据目标的颗粒度可以分为不同的实现方案，基于 AI 编译器的方案常借助强化学习进行细粒度的优化和编译，如 VTA、RK3588 等。而 AI 推理引擎的硬件实现更加多样化，如基于 ONNX Runtime 的 Gemmini、基于 ArmNN 的

ARM Ethos 系列 NPU 等。

深度学习算法在嵌入式和移动设备的部署过程中，由于硬件功耗、资源成本等因素的限制，硬件专用加速单元往往功能有限，为了实现高度的灵活性和通用性，降低后期维护难度，大多数设计中均采用了异构设计方案。根据 NPU 和 CPU 的耦合程度，可以将代码生成过程重新进行分类。若 CPU 支持指令集扩展，则 NPU 方案常采用紧耦合设计，如 Gemmini+Rocket 的设计方案中，Rocket Core 可以通过 RoCC 进行指令集扩展，自定义指令集部分则主要包含了矩阵乘，数据流加载和存储的一些指令；国内 RISC-V 厂商芯来的 N 系列、NX 系列 CPU Core 采用 NICE 进行指令扩展，软件编程实现采用 C 内嵌汇编的方式实现。除此之外，若 CPU 不支持指令集扩展，NPU 可以通过外设的方式与系统总线相连，此时代码生成过程实际上相当于生成众多子任务，以配置寄存器的形式对加速器单元进行控制。

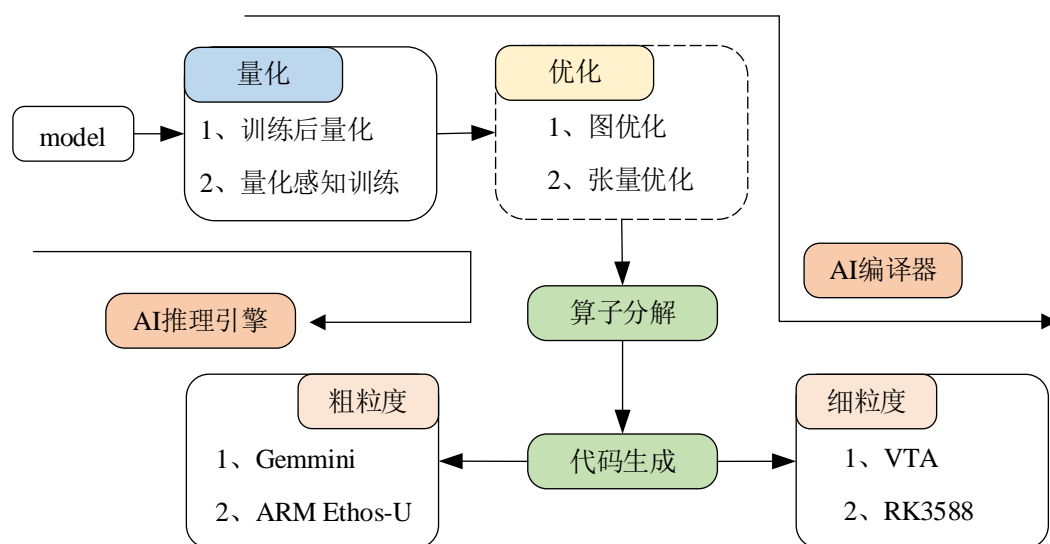


图 4-1 深度学习模型部署软件技术路线

本文的设计方案中，NPU 单元采用 Chisel 敏捷化设计并具备了 SoC 集成能力，硬件连接如图 3-1 所示。NPU 软件栈开发硬件适配的 AI 推理引擎，支持完整的模型转换、优化、量化工具链，通过在不同的算子层添加 NPU 推理函数，配合 AXI 总线和外部中断，实现了深度学习算法端到端的高效部署。

4.2 硬件适配的 AI 推理引擎

4.2.1 NCNN 推理引擎

NCNN 是腾讯优图实验室开源的面向移动和嵌入式领域的深度学习框架，该框架兼容多种前端输入、支持跨平台推理、凭借着轻量化、简洁化、高度灵活性和易移植性等突出特点受到了广大开发人员的关注。NCNN 推理引擎底层采用 C++ 编写，包含用于边缘端和移动端的简易 OpenCV 库、Math 库等源码，提供 Linux、Windows、macOS、Android、IOS 等平台部署示例。本文的设计方案中，以此为基础进行二次开发，实现

了嵌入式裸核环境 NCNN 推理引擎的移植。与此同时，原作者在设计之初充分考虑了多平台硬件后端优化的需求，针对不同指令集 SIMD 进行汇编级优化，无需添加 NNPACK、BLAS 等任何第三方库依赖。

表 4-1 列举了 NCNN 现阶段支持的部分软件平台和硬件后端，总体而言，该框架在 Android、IOS 等移动端中支持较为全面，支持众多型号的 CPU 和 GPU 设备。在 PC 端，开发人员可供选择的编译器和推理引擎较多，如 TensorRT、TVM 等；由于 NCNN 仅支持低功耗的 Vulkan API 实现 GPU 加速，不支持 CUDA 和 OpenCL 等编程模型，因此可能在开发过程中面临驱动兼容问题。除此之外，NCNN 针对不同指令集的 CPU 后端均进行了 SIMD 优化，在 Linux 环境下得到了验证，并提供了交叉编译工具链，可以快速实现嵌入式设备的 CPU 部署。

表 4-1 NCNN 硬件后端支持现状（部分）

| System | ISA | CPU(32bit) | CPU(64bit) | GPU |
|---------|-----------|------------|------------|-----|
| Linux | ARM | No | Yes | No |
| | MIPS | Yes | yes | No |
| | RISC-V | No | Yes | No |
| | LoongArch | No | Yes | No |
| Windows | x86 | Yes | Yes | Yes |
| macOS | ARM | Yes | Yes | No |
| | ARM | No | Yes | Yes |
| Android | ARM | Yes | Yes | Yes |
| IOS | ARM | Yes | Yes | Yes |

图 4-2 对 NCNN 推理引擎的软件框架进行了总结，其支持多种深度学习框架的模型输入，如 Tensorflow、ONNX、Pytorch、Caffe、MXNet、keras、MLIR、Darknet 等，提供模型转换器将前端输入转换为 NCNN 支持的后缀为 param 的网络结构配置文件和后缀为 bin 的模型参数文件。NCNN 针对全套转换工具链均进行了开源，支持图优化以进行算子融合，支持 INT8、FP16 数据格式的量化、存储，支持 OpenMP 和 SimpleMP 多核加速。除此之外，NCNN 在 Runtime 过程中从算法和硬件架构进行了多方位细粒度优化。在算法优化层次，首先采用了精细的内存管理和数据结构设计，运行过程中临时内存占用极低；并针对现代 CPU 架构中矩阵列方向处理速度快于行方向的问题，提出了矩阵内存管理的 Pack 策略，通过访存优化提高推理性能。其次，NCNN 支持 im2col+GEMM 和 Winograd 两种卷积加速算法，并针对卷积层、深度卷积层和全连接层提供 INT8、FP16、BF16 推理能力。在硬件优化层次，NCNN 原作者 nihui 展开了大量的工作，在汇编级对不同 ISA 进行 SIMD 优化并不断更新，如针对 x86 架构采用 SSE、AVX、AVX512 进行加速，针对 ARM 架构采用 NEON 进行加速，针对 RISC-V 架构采用 RVV 进行加速，针对 MIPS 架构采用 MSA 进行加速，针对 Loongarch 架构采用 LSX、

LASX 进行加速。

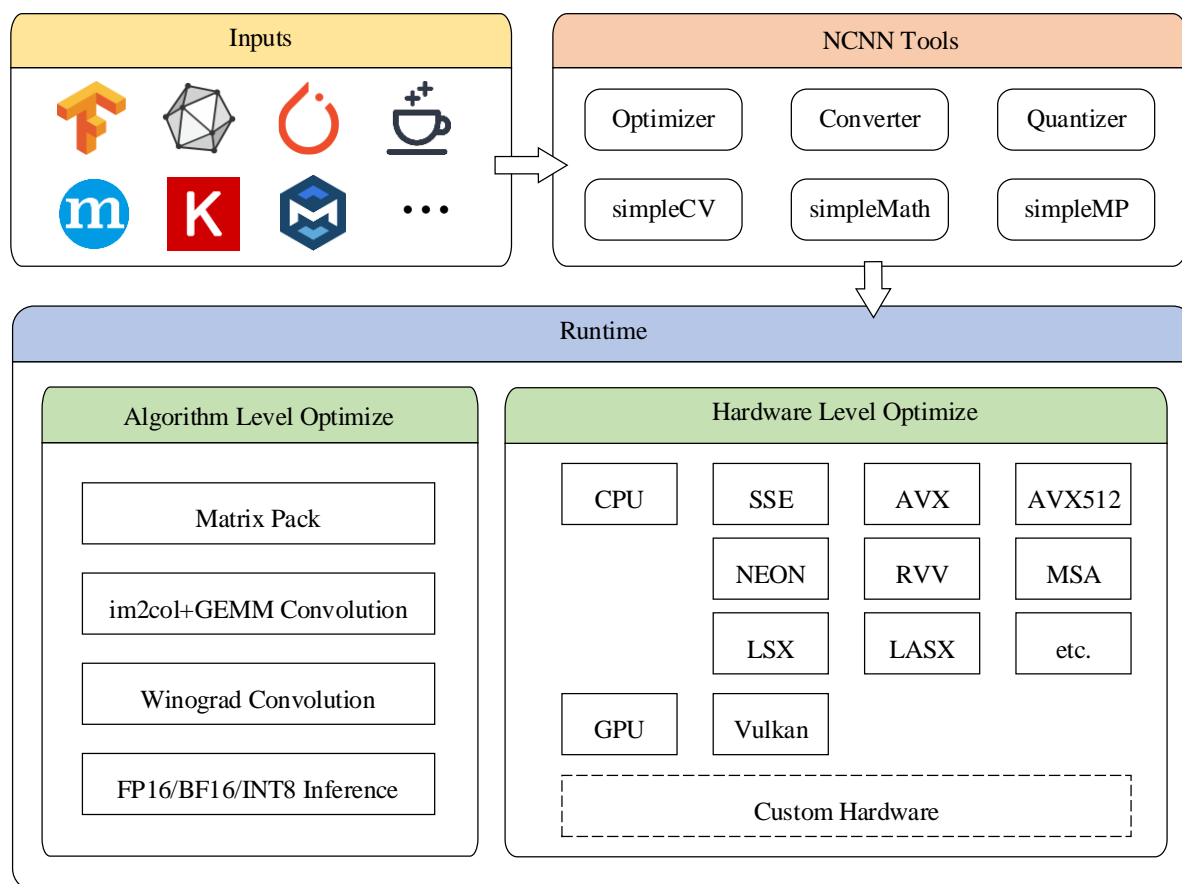


图 4-2 NCNN 推理引擎软件框架

目前，NCNN 推理引擎包含基本算子近百种，由于软件框架以算子层作为最小计算单元并采用 `cmake` 进行项目构建，方便了用户自定义硬件后端的添加。本文在 NPU 异构加速系统的 AI 推理引擎设计过程中充分耦合 NCNN 设计理念，并在此基础上进行修改和适配。在保证软硬件功能完整的前提下，有效降低了软件环境的开发工作量和难度。在推理引擎底层架构上，针对张量数据结构进行硬件总线适配的通道对齐处理以提高数据访存速度；同时在网络参数文件中进行文件内数据块对齐，结合软拷贝实现了模型参数的快速加载，并添加更多算子属性以满足硬件推理需求。其次，在神经网络模型的端到端部署工具链中进行了计算图优化，以适配硬件微架构的算子融合、重量化等优化策略。在边缘端推理引擎的移植过程中添加了自定义 NPU 后端的 C++ 驱动库并打通了 NCNN 底层算子调用 API。此外，在 Linux 环境下搭建基于 Verilator 的仿真框架，融合硬件适配的 AI 推理引擎，实现了软硬件的联合仿真。

4.2.2 Mat 数据结构

硬件适配的推理引擎底层代码中对内存管理和数据结构进行了细致的优化。在神经网络推理过程中会产生较多的临时特征图，其数据大小常常分布在数十 KB 到数百 MB，合理的数据对齐方案可以给访存带来极大的性能提升。推理引擎底层框架中定义

了一种名为 **Mat** 的数据结构用于张量的存储。**Mat** 数据结构包含多种常用方法，可以快速实现张量的构造、拷贝、内存分配和不同 **channel** 数据的索引等操作。除此之外，该数据结构通过 C++ Class 实现，并包含多种属性，如维度 (**dims**)、尺寸 (**w**, **h**, **c**, **d**)、数据指针 (***data**)、元素大小 (**elemsize**) 以及数据步长 (**cstep**) 等。

在硬件设计方案中，**AXI** 总线的数据位宽为 128，可以在同一周期内完成 16Bytes 数据的传输，尽管 **AXI** 支持非对齐传输，但对于 **SIMD** 加速和硬件加速方案而言，地址对齐可以极大的提升计算效率。通过对 **Mat** 数据结构的修改，目前采取宏定义的方式已经实现了任意通道首地址任意字节对齐，本文软件方案中采用 16 字节对齐以兼容硬件。数据块的首地址对齐从实现角度来讲包括编译条件指定和数据结构设计两种方法。在 C 语言中可以使用 `__align(n)`、`__attribute(aligned(n))`、`#pragma pack(n)` 等手段指定地址 **n** 字节对齐，并在编译过程中对齐数据。而数据结构方法则是通过软件设计实现地址对齐，下文给出一种常见的方案。首先在内存中分配一块略大于数据块大小的内存空间，记录首地址的位置以防止内存泄露，同时计算该地址对齐后的位置作为该数据块实际存放的首地址即可。为了实现任意通道的首地址对齐，可以采用如下两种方案。第一种方案较为简单，开发人员可以连续开辟与通道数相同的对齐数据块，但是具体执行过程效率不高。第二种方案则直接开辟一块对齐的内存空间用于存储全部数据，并保证不同通道间总字节数对齐即可，显然，该方法更加高效、并易于对数据进行管理。本文的 **AI** 推理引擎采用后者的实现，即 **Mat** 数据结构。

图 4-3 中例举了 **Mat** 尺寸为 (2,3,5) 时张量块的内存模型。在 **w** 和 **h** 方向，数据连续排列，在 **c** 方向进行了数据对齐处理，因此会造成一部分数据空间无效。当张量或特征图尺寸较大时，用于数据对齐的内存浪费相比于整体影响微乎其微。**cstep** 作为特征图 **Mat** 结构中非常重要的属性，通过硬件参数寄存器的配置，可以实现 **NPU** 单元对特征图任意通道的对齐访问，其计算公式如 4-1 所示。

$$cstep = \text{align}(w \times h \times elemsize, 16) / elemsize \quad (4-1)$$

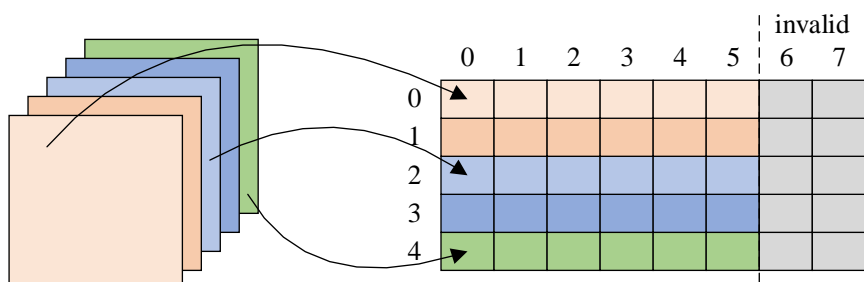


图 4-3 内存模型举例

4.2.3 网络参数文件

随着深度学习技术的飞跃式发展，其商业价值和军事价值逐渐明朗，国内外企业和科研机构纷纷推出各自的深度学习训练框架和推理平台，例如 **FaceBook** 提出的 **Pytorch** 采用 **pt** 或者 **pth** 后缀格式文件存储模型、**Google** 提出的 **Tensorflow** 采用 **pb** 后

缀格式文件等。为了兼容多种前端模型输入，推理引擎适配中采用了独立的网络参数文件定义并支持全套转换优化工具用于模型转换。其中，网络参数文件与 Caffe 的设计理念相似，param 文件用于网络参数的配置，bin 文件用于模型参数的存储。

本文设计的 NPU 主要针对密集计算进行加速，累计支持 20 个常用算子。图 4-4 例举了 YOLOv5s 网络参数文件的部分配置，主要包含算子层类型、名称、配置、算子层输入输出数据以及输入输出 Blob 的名称等。

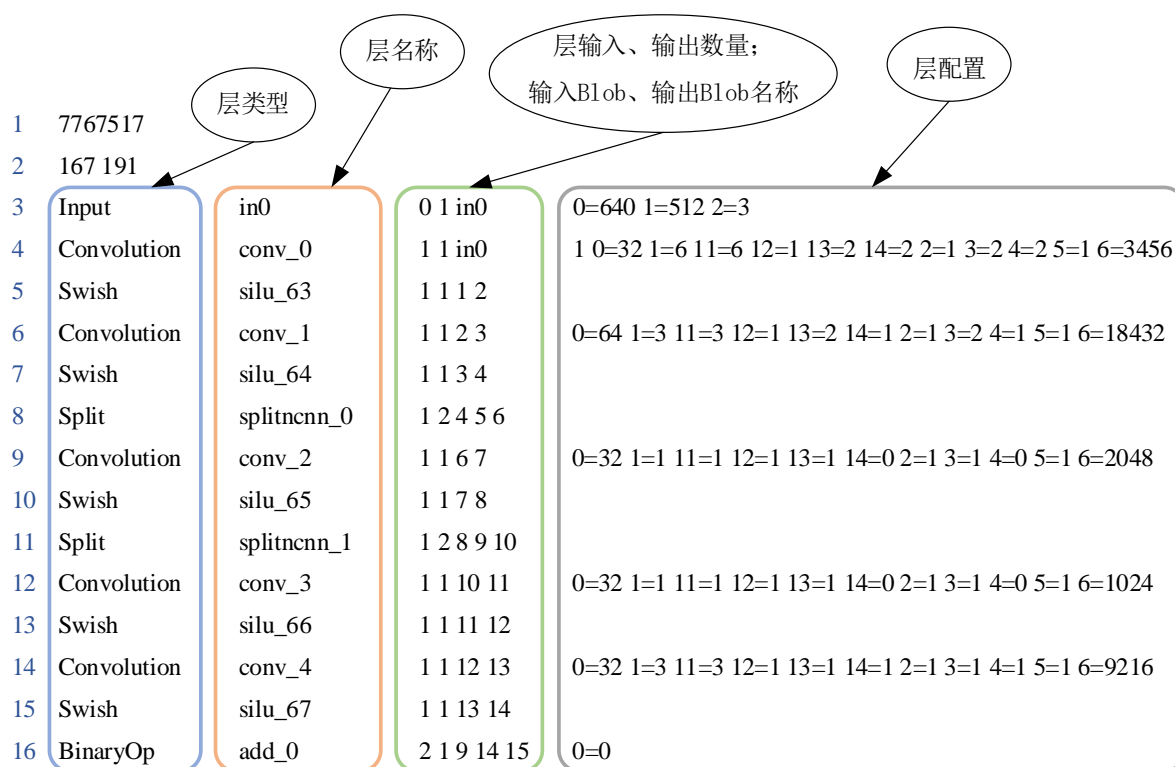


图 4-4 YOLOv5s 网络参数配置文件（部分）

表 4-2 网络参数配置文件手册（部分）

| 算子名称 | ID | 参数解释 | 默认值 | ID | 参数解释 | 默认值 |
|-------------|----|------------------|----------|----|-------------------|------------|
| BinaryOp | 0 | op_type | 0 | 1 | with_scalar | 0 |
| | 2 | b | 0.f | | | |
| | | | | | | |
| Convolution | 0 | num_output | 0 | 1 | kernel_w | 0 |
| | 2 | dilation_w | 1 | 3 | stride_w | 1 |
| | 4 | pad_left | 0 | 5 | bias_term | 0 |
| | 6 | weight_data_size | 0 | 8 | int8_scale_term | 0 |
| | 9 | activation_type | 0 | 10 | activation_params | [] |
| | 11 | kernel_h | kernel_w | 12 | dilation_h | dilation_w |
| | 13 | stride_h | stride_w | 18 | pad_value | 0.f |

其中，Convolution 算子层定义了卷积行为，层配置中不同的 ID 索引标识着该层的卷积核个数、卷积窗长度、步长、Padding 以及激活函数等信息，如表 4-2 所示。YOLOv5s

采用 Swish 作为激活函数，图 4-4 为其网络参数文件的部分内容。在本文的软硬件方案中，已经实现了常用激活函数的算子融合。此外，input 层定义了输入图像的分辨率和通道数，Split 层用于产生网络拓扑结构中的分支，BinaryOp 层用于残差块的计算。

bin 文件用于模型参数的存储，主要包含权重、偏置、量化系数、反量化系数、重量化系数等。在硬件加速过程中，仅针对卷积、深度卷积以及全连接层进行了 INT8 量化，其他算子层仍采用 FP32 的格式进行网络推理。因此，在 bin 文件中会包含多种数据格式的参数字，本文采用 Tag 的方式在每项参数的头部位置进行标识，数据格式与 Tag 的映射方式如表 4-3 所示。除此之外，为了简化文件的解析流程，提高模型加载速度，本文对 bin 文件中存储的数据和 Tag 均进行了 16Bytes 对齐处理，不足的位置使用零值填充，推理引擎执行过程中仅通过浅拷贝即可获得内存的数据块。

表 4-3 bin 文件数据格式与 Tag 映射

| 数据格式 | Tag |
|------|------------|
| FP32 | 0x00000000 |
| FP16 | 0x01306B47 |
| INT8 | 0x000D4B38 |

4.3 神经网络端到端部署工具链

本文提出的推理引擎支持完整的神经网络端到端部署工具链，如图 4-5 所示。该工具链支持多种深度学习框架的模型格式输入，如 Pytorch、ONNX、Tensorflow 等，具备极强的兼容性。在具体的部署过程中，首先利用模型转换器将前端输入统一为 NCNN 内建的原始网络配置文件和模型参数文件，即 origin.param 和 origin.bin。其次利用模型优化器进行图优化，本文的软硬件设计在 NCNN 后端添加定制的 NPU 硬件加速器并设计多条数据通路以支持计算和访存延时的优化。硬件加速的计算核心由脉动阵列和向量 ALU 组成，在优化器中实现了卷积层、深度卷积层、全连接层与各种激活函数的算子融合，以及卷积层、深度卷积层与批归一化层的算子融合。除此之外，在图优化的过程中添加重量化策略，有效降低硬件访存压力。

模型量化采用训练后量化的对称量化方案，也称非饱和量化。神经网络模型的量化包括输入特征图和权重的量化，其中特征图量化一般采用在线方案，权重量化一般采用离线方案。考虑到单独卷积核的数据分布往往比较密集，因此可以对每个卷积核进行数据统计，利用权重参数的最大值直接计算权重的缩放因子，并完成 FP32 的原始权重到 INT8 的量化权重的直接映射，如公式 4-2 所示。其中， ω 为原始单精度浮点权重， ω_{\max} 为原始权重的最大值， ω_q 为量化后的 INT8 权重数据。该权重量化方案与 NVIDIA 的 TensorRT 实现方案相同，对于卷积和深度卷积层，权重量化因子的个数等于卷积核的个数。

$$\omega_Q = \frac{\omega \times 127}{\omega_{\max}} \quad (4-2)$$

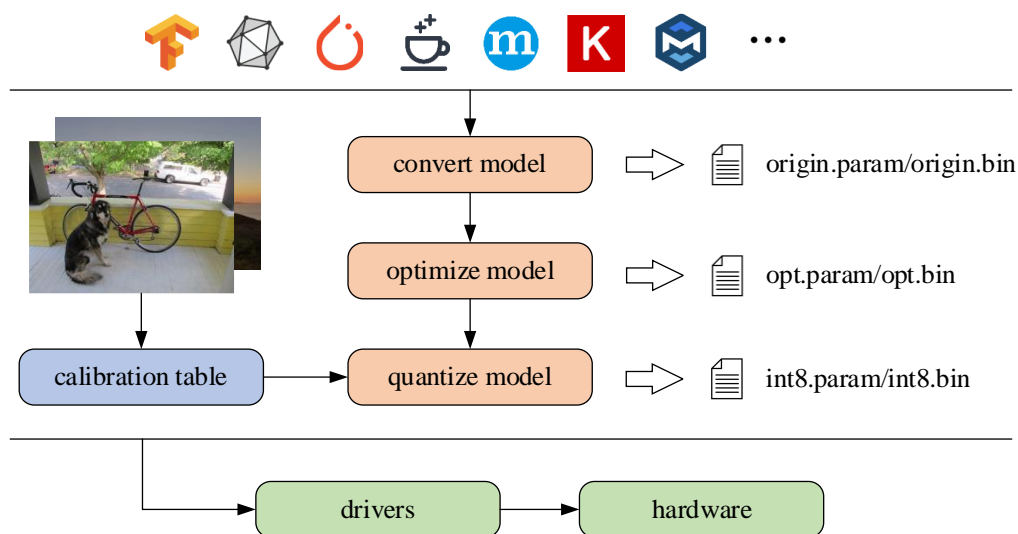


图 4-5 神经网络端到端部署工具链

相比较而言，输入特征图的量化较为复杂，由于数据在输入和推理的过程中存在不确定性，自然无法进行离线量化。此外根据实验表明，由于不同层中特征图的激活值数据分布不规律，直接映射会造成严重的精度下降甚至推理结果错误。为了尽可能降低量化前后数据分布的差异性，量化器采用 KL 散度计算相对熵来寻找最优解。在实际的应用过程中，如图 4-5 所示，使用校准数据集模拟产生输入特征图的数据分布，该数据集需要尽量包含应用场景且数量不宜过多。分析特征图的数据分布并根据搜索算法获得量化校准表，具体实现如图 4-6 所示。

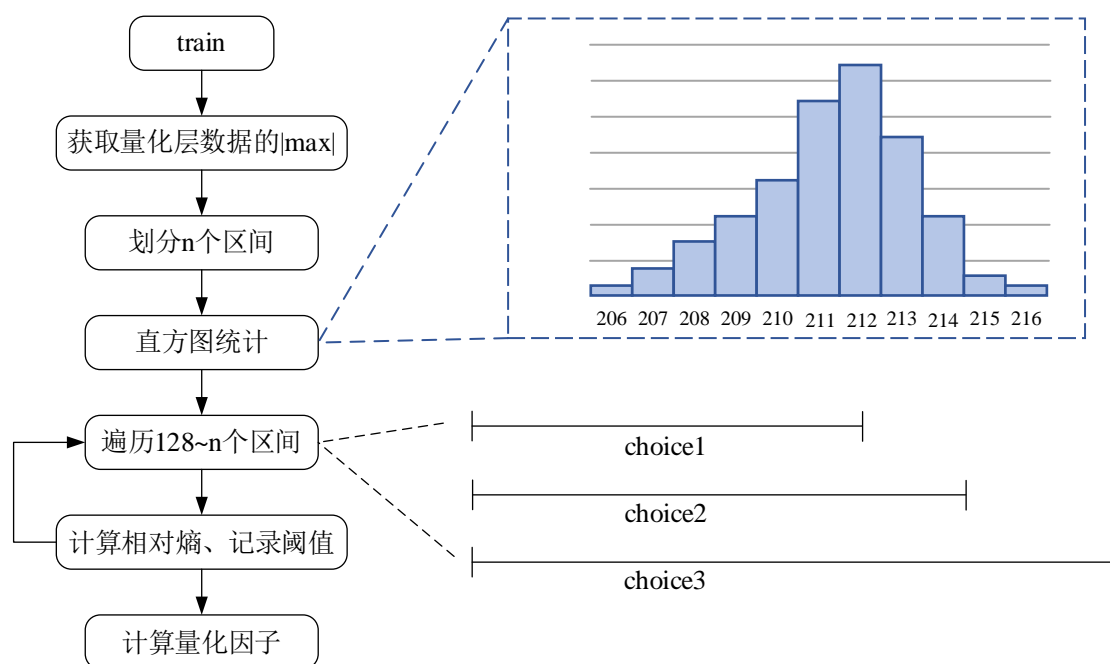


图 4-6 模型量化原理

在具体实现过程中，针对训练后浮点模型的每个量化层进行数据统计以获取数据的最大值，并将 $0 \sim \omega_{\max}$ 的数据分布划分为 n 个区间。一般来讲，划分的区间越精细，量化后的整体性能越好，但随之而来的计算量也越大，例如 TensorRT 量化方案中采用 2048 个区间，MXNet 量化方案中划分 4096 个区间。随后，根据划分的区间为基础对每个量化层的输出进行直方图统计，得到其数据分布。依次选取 $0 \sim 128$ 、 $0 \sim 129$ 、 \dots 、 $0 \sim n-1$ 的区间统计结果进行遍历，并将其编码形式定义为 P_i 。为了减少数据截断带来的精度下降，实际操作中将区间外的数据个数累积到区间边界的位置。对于常见的 INT8 量化，其数据范围为 $-128 \sim 127$ ，本方案中采用对称量化，因此正数的实际范围为 $0 \sim 127$ ，令量化后的编码为 Q ，编码长度为 128。由于 P_i 的编码长度大于等于 128，而相对熵的计算要求 P_i 和 Q 有相同的编码长度，为此需要对 Q 进行扩展。在遍历过程中寻找相对熵最小的 choice，将此时最大区间的中值作为量化的阈值 T ，以此计算得到缩放因子 s_0 ，如公式如 4-3 所示。而在神经网络的量化模型的软硬件部署中常采用缩放因子的倒数作为量化系数，以此将量化的除法运算转换为乘法运算。

$$s_0 = T / 127 \quad (4-3)$$

$$s = 1 / s_0 \quad (4-4)$$

4.4 边缘端软件设计

本文的软硬件方案主要针对边缘端深度学习算法进行加速，系统整体采用异构设计，CPU 负责神经网络运行时调度以及非典型算子加速等工作，NPU 负责计算密集型算子的硬件加速。由于 NPU 采用 IP 化的设计理念，仅通过 AXI 和 AXI-Lite 总线进行数据流和控制流的交互，因此 NPU Core 具有极强的灵活性和兼容性，支持多种指令集架构的 CPU。原型验证阶段采用 ARM A53 裸核环境进行平台搭建，并基于 Xilinx Vitis IDE 完成软件程序的编译、链接、下载。

在实际部署过程中，NCNN 交叉编译工具链和应用对象集中于各大操作系统并采用 glibc 和 glibc++ 进行编译链接，而在嵌入式裸核和实时操作系统中常采用 libc 和 libc++ 的标准库。此外，相比于 Linux OS，裸核开发中常采用开源且轻量化的 FATFS 文件系统，这也导致了 NCNN 底层代码中存在大量与 BSP 不兼容的标准库函数，进而增加了移植难度。虽然本文的原型验证过程中仅针对于 ARM 架构的 CPU 进行部署，但是整体软件程序采用层次化设计，实现了应用层、加速库和驱动层代码的解耦。其中，驱动层代码包括 NPU 调用库函数、NPU 寄存器地址映射、NPU 中断初始化和服务函数、以及 SD 卡、DP 显示、串口等硬件外设库函数。加速库中代码可以进一步细分为 C 算子库和推理引擎底层框架，通过在 C 算子库中实例化硬件对象、调用驱动层中 NPU 硬件 API，实现推理引擎自定义后端的添加。加速库和驱动层中关键参数可以通过配置文件在顶层设计中进行更改，通过层层封装和抽象，二次开发过程中无需关注底层实现细节。应用层代码具体为各种卷积模型的推理主函数，承担模型加载、推

理 API 调用、网络后处理等任务，本文提供了 YOLOv5s、MobileNetv2、Resnet50 等二十多个应用示例。

在具体的原型系统搭建过程中，本文实现了两种演示方案。方案一采用 SD 卡作为神经网络推理的输入输出设备，嵌入式文件系统读取 SD 中的图像和模型，推理完成后将推理信息通过串口打印并将推理图片写回 SD 卡，借助 PC 机完成结果的查看。方案二在 SD 卡存储模型和串口打印信息的基础上，添加了实时视频流进行处理，并将推理结果输出到显示器。两相比较，方案二演示效果更加直观，具备更强的人机交互能力。图 4-7 描述了嵌入式视频流推理的软件执行框图。

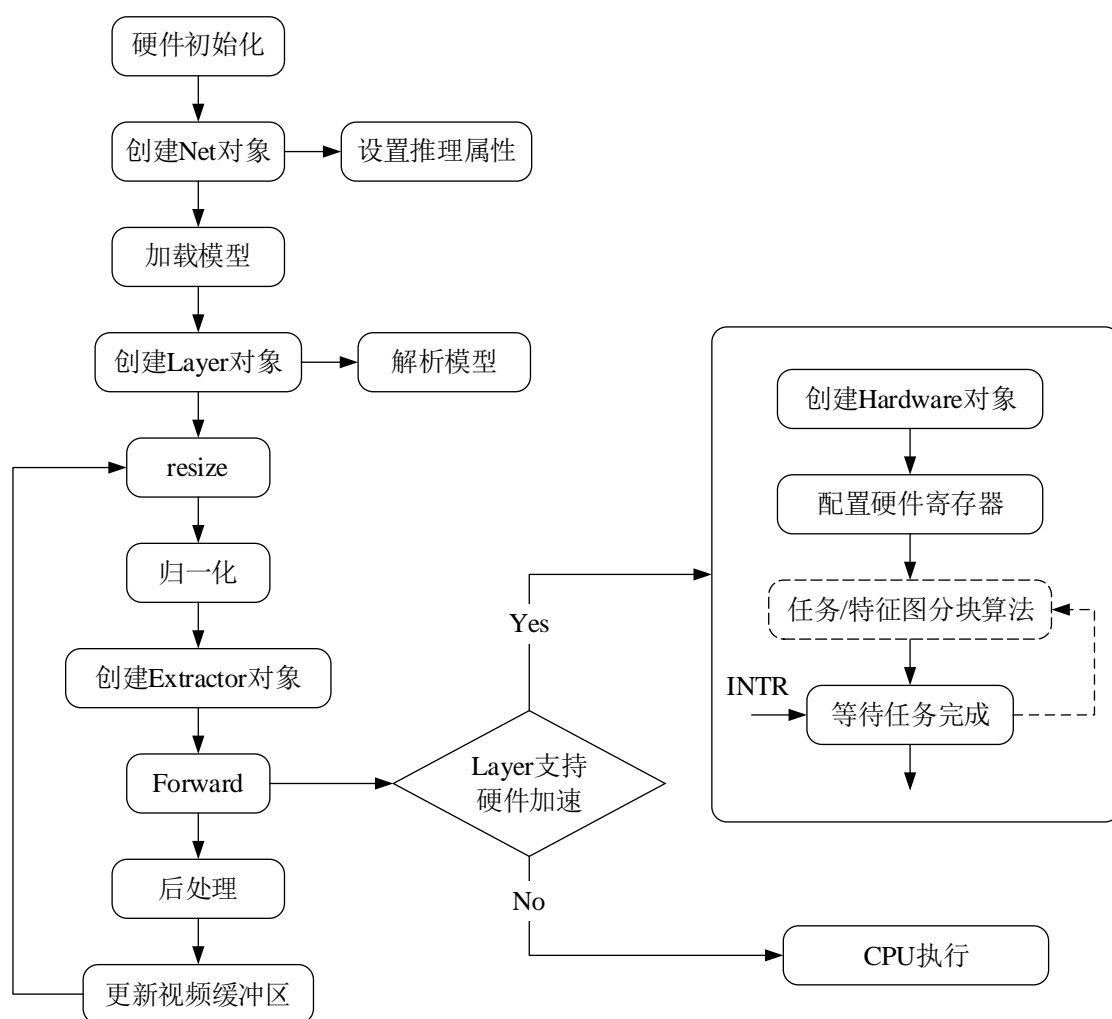


图 4-7 嵌入式视频流推理执行框图

硬件上电之后首先需要完成硬件的初始化，包括 SD 卡、DP 显示、NPU 外部中断的初始化配置。其次创建推理的 Net 对象，并设置推理属性信息，如当前模型的推理精度、推理平台以及其他优化信息等。通过 Net 的 load_param_mem 和 load_model_mem 方法完成模型的加载，并根据 param 文件实例化多个 Layer 对象完成模型解析。现阶段许多的 CNN 模型均需要对输入图像进行 resize 和归一化处理，归一化过程可以借助驱动层提供的 API 进行硬件加速。此外，实例化 Extractor 对象构建推理执行器，指定

输入数据并创建模型输出 Mat 以保存推理结果。在推理过程中，调用加速库中的 C 算子库进行计算，若当前 Layer 支持硬件加速，则调用驱动层 API 实例化 Hardware 对象并对 Hardware 的属性进行配置，NPU 接收到 CPU 的加速请求即启动 DMA 开始运算，计算结束后通过 NPU 中断标识任务完成。对于卷积、深度卷积、池化等算子层可能包含任务和特征图的分块算法，此时该算子层的加速被分解成多个子任务的形式。若当前 Layer 不支持硬件加速，则推理引擎将采用 CPU 进行计算，此时可以借助 SIMD 进行加速，如 ARM 架构的 CPU 可以采用 NEON、RISC-V 架构的 CPU 可以采用 RVV 进行汇编级加速。等待所有算子层 Forward 结束，对推理结果进行后处理并送至输出视频缓冲区，完成显示画面的刷新。重复前处理、模型推理、后处理、刷新显示缓冲区四个过程，即可完成视频流的实时处理和结果显示。

在嵌入式软件推理平台的设计过程中，充分考虑了软件移植和应用二次开发的难度。通过软件程序层层抽象和解耦，可以在不同指令集架构的 CPU 系统中灵活移植。此外对加速库和驱动层代码进行封装，提供大量 API 供开发人员调用，极大降低了深度学习应用开发的难度。

4.5 基于 Verilator 的软硬件协同仿真框架

Verilator 是一款支持 Verilog 和 System Verilog 的周期精确开源仿真器，其通过将 RTL 代码转换成 C++ 或 System C 的形式，并借助编译器进行高效优化实现快速仿真。目前，Verilator 被广泛应用于数字 IC 验证等领域。本文设计的加速器系统采用软硬件协同设计，CPU 中运行深度学习推理引擎，NPU 中执行密集计算加速。两者之间的数据流采用共享内存的方式进行数据交互、控制流采用配置寄存器和中断的方式进行指令传递。为了实现整个系统的仿真工作，采用传统的 RTL 仿真架构需要编写庞大的激励文件，势必带来很大的困难，虽然以 TestBench 为基础的流程较为经典，但是系统整体设计时间有限。为此，本设计中以 Verilator 为基础搭建软硬件联合仿真器框架，软件环境中采用 C++ 编程，移植推理引擎并编写 NPU 驱动程序，直接实现了功能完整的全套软硬件仿真，极大缩短了硬件电路的开发与调试周期。

图 4-8 展示了基于 Verilator 的软硬件联合仿真框架的设计，其主要由接口定义、驱动层、激励发生器和应用层四个部分组成。仿真器整体设计利用 Verilator 工具将 RTL 代码转换成 C++ 代码，并在软件环境中采用 C++ 编写测试用例并产生激励。接口定义即 IO 层的实现，其采用软件模拟双通道 AXI 接口时序实现软硬件的数据流交互，而控制流的接口弱化了 AXI-Lite 的影响，直接通过结构体指针实现对硬件寄存器的读写操作。驱动层和应用层的代码与嵌入式裸核程序大同小异，仅由于运行环境不同，底层标准库略有差异。激励发生器可以在网络模型、算子层、硬件功能单元等不同粒度产生测试用例，并可将硬件推理结果与软件运算实时比对，精确定位异常。除此之外，仿真器框架中集成了神经网络端到端部署脚本，支持 Pytorch、ONNX 等深度学习框架中的模型直接部署到软硬件联合仿真器。因此，该框架具备进行网络推理的能力，并

可以获得与硬件实验相同的输出、波形文件以及性能评估报告等。

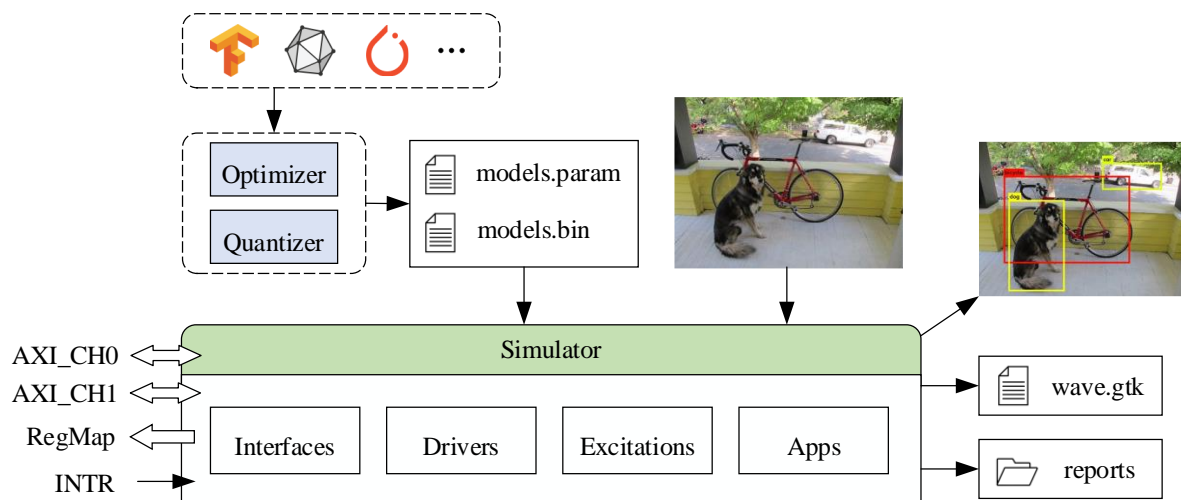


图 4-8 软硬件仿真设计架构

在软硬件联合仿真器的设计过程中，为了最大限度的降低软件环境和硬件电路的验证工作量，仿真平台中添加了大量的断言和校准函数以缩小故障范围。图 4-9 从宏观角度给出了仿真器的工作原理，其支持性能分析、随机测试、批量测试、波形记录、自动校验、异常检测等功能。AXI_Check 模块用于检测双通道 AXI 总线的时序异常和请求异常，如 AXI 突发地址是否满足硬件对齐要求、数据传输是否跨越 4K 边界、Outstanding 深度是否溢出、地址通道和数据通道是否保序、以及 AXI 时序是否满足规范等。

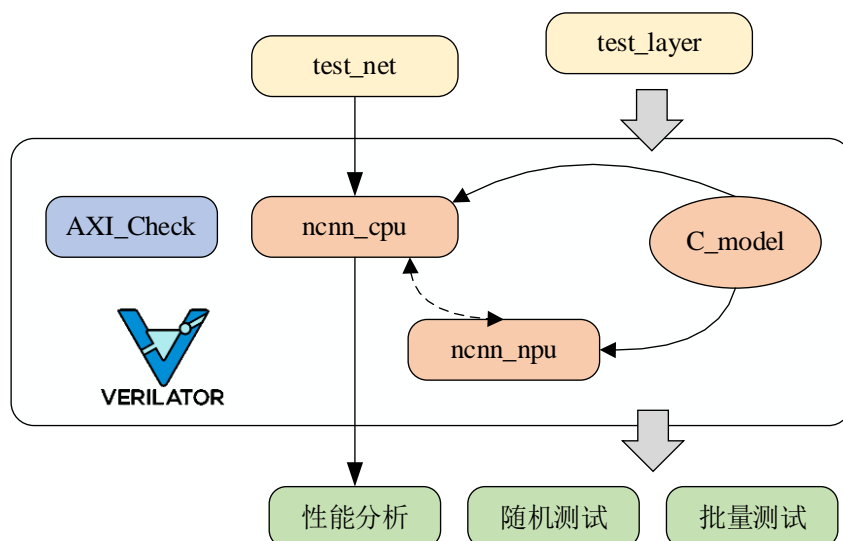


图 4-9 仿真器工作原理

仿真器中支持对 NPU 中的每一个模块的独立仿真、神经网络每一层的独立仿真、以及任意神经网络的整体仿真等任务。此外，在仿真环境中编写 C_model 程序作为数据参考对象，分别支持 NPU 端和 CPU 端推理引擎执行结果与 C_model 的数据比对；在部分测试场景，支持算子层在不同硬件平台的直接实时数据比对工作。通过对各个

算子和硬件功能单元进行随机测试和批量测试, 实现了电路逻辑的分支覆盖和功能覆盖。最后, 在仿真环境中添加了性能分析功能, 通过对访存时间、计算时间、AXI 阻塞时间以及硬件系统 FIFO 状态进行统计, 确定硬件性能瓶颈并不断迭代优化。

图 4-10 展示了基于 Verilator 软硬件联合仿真器的代码功能组成, 整体采用 Makefile 进行项目管理。tools 文件下包含了适用于仿真器和硬件平台的神经网络模型端到端部署工具链, 并使用 cmake 进行构建。log 文件夹下保存了仿真过程中的张量数据, scripts 和 reports 文件夹中包含加速器的性能分析脚本和报告。csrc 文件夹包含了 AI 推理引擎和仿真器的全部 C++ 源码, 而 vsrc 文件夹包含了硬件电路的 RTL 设计。硬件工程采用 Chisel 进行开发, 并使用 sbt 进行项目构建, 最后经 firrtl 中间表示转换为 Verilog 文件。

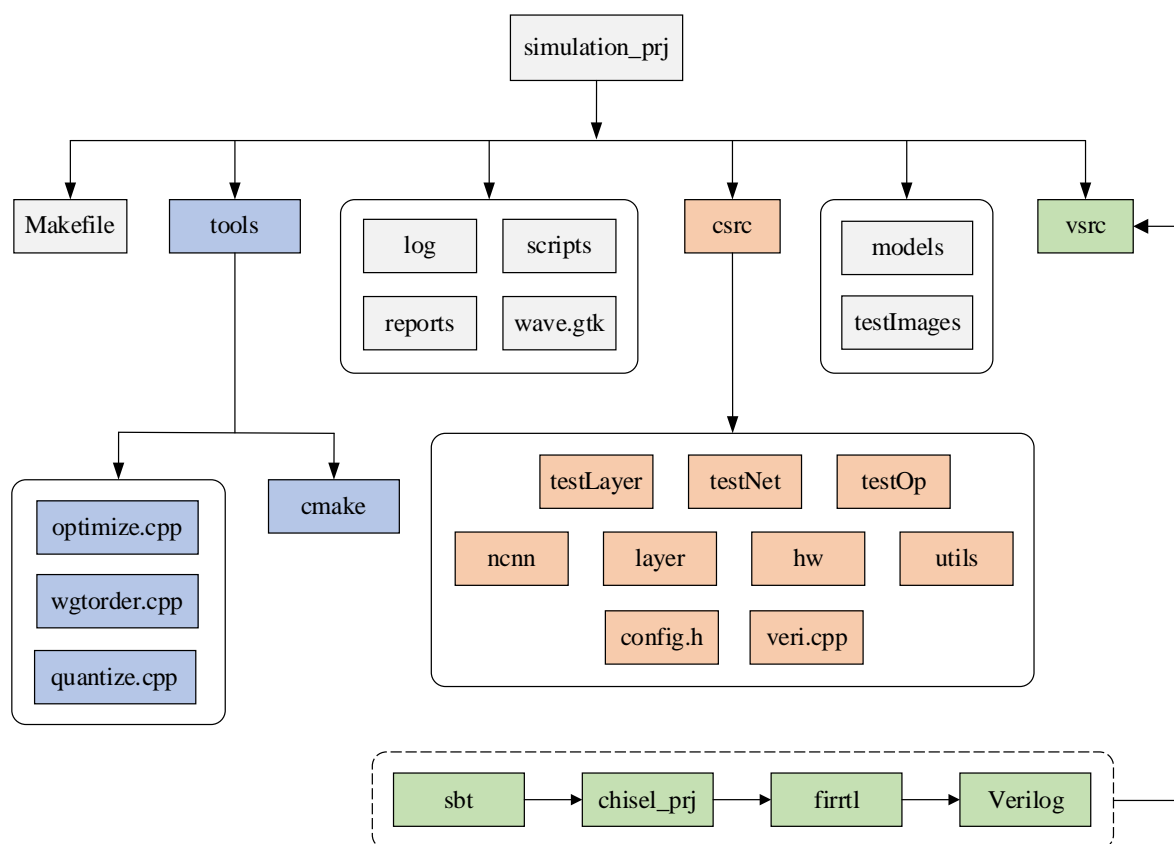


图 4-10 软硬件仿真代码结构框图

4.6 本章小结

本章详细介绍了适用于自定义 NPU 的软件栈设计。首先分析了软硬协同过程中深度学习的技术路线，并介绍了 NCNN 推理引擎的后端支持情况、内存模型以及参数文件格式等内容。其次，开发了神经网络端到端部署工具链以及边缘端裸核软件运行环境。最后，设计了基于 Verilator 的软硬件协同仿真环境以实现快速的故障定位、性能分析以及批量测试等。

5 实验测试与性能分析

5.1 Verilator 仿真性能分析

为了降低软件程序和电路设计的验证难度，本文基于 Verilator 搭建了软硬件联合仿真器，支持各硬件功能单元和各算子层的单独验证，并支持卷积神经网络的端到端部署。Verilator 软硬件联合仿真环境中，不仅支持批量测试、随机测试以及异常定位等功能，还可以在系统层次对目前 NPU 电路设计的性能瓶颈进行分析，为后期设计迭代指明方向。

表 5-1 例举了 NPU 进行卷积层和深度卷积层硬件加速时访存和计算部分的时间消耗，其中 kernel 设置为 3，stride 设置为 1，常规模式下未使用重量化。对于硬件卷积计算，权重缓冲区采用了乒乓设计，并在脉动阵列中添加权重预加载数据通路，配合独特的特征图分块策略可以实现最大限度的硬件权重复用。因此，在脉动阵列特征图流动过程中同时完成了下一轮权重从 DDR 中的预加载，进而掩盖了权重的访存延时。表 5-1 中访存延时主要由输入特征图传输延时组成，常规模式下，输入特征图在 DDR 中的排列采用 NCHW@FP32 的形式，在进行 Layout 变换的过程中，突发长度较短且突发次数较多。虽然在 AXI 设计过程中添加 Outstanding 传输以在宏观上降低 DDR 响应延时，但受限于庞大的数据量和浮点数所占字节数，整体加速效果有限。而对于深度卷积而言，由于计算模型中不存在卷积窗在输入通道方向的累加，计算量相对较小，因此脉动阵列的计算时间小于每轮权重的加载时间。在硬件加速过程中，整体上深度卷积层的访存时间占比明显高于卷积层，主要原因是有限的传输带宽不足以支撑高效的脉动阵列计算，也为后续访存架构优化提供了方向。

表 5-1 常规模式下访存和计算时间占比

| 算子类型 | iw、ih、ic | oc | 各部分周期/时间占比 | |
|------|-------------|-----|-----------------|-----------------|
| | | | 访存 | 计算 |
| 卷积 | 32、32、128 | 128 | 25600(25.56%) | 74542(74.44%) |
| 卷积 | 104、104、256 | 128 | 494592(23.16%) | 1641233(76.84%) |
| 卷积 | 208、208、256 | 64 | 2107904(36.43%) | 3678112(63.57%) |
| 卷积 | 416、416、32 | 64 | 751680(30.72%) | 1695215(69.28%) |
| 深度卷积 | 32、32、32 | 32 | 4114(39.41%) | 6325(60.59%) |
| 深度卷积 | 96、96、128 | 128 | 154840(42.25%) | 211665(57.75%) |
| 深度卷积 | 208、208、256 | 256 | 1835744(46.61%) | 2103070(53.39%) |
| 深度卷积 | 416、416、128 | 128 | 3619552(46.37%) | 4186219(53.63%) |

在本文的软硬件方案中针对特征图的访存延时优化也做出了一些尝试，表 5-2 给

出了采用重量化方案时硬件进行卷积层和深度卷积层加速的访存和计算时间及各部分占比。此时，输入输出特征图在 DDR 中保存为 NHWC@INT8 的张量格式且数据连续紧密排列，与硬件架构需求保持一致，采用 Outstanding 传输机制可以实现数据的快速搬移。与表 5-1 相比较，采用重量化策略节约了特征图四分之三的传输带宽，硬件进行卷积和深度卷积加速的访存延时大幅度降低。现阶段重量化策略仅支持相邻的卷积层和深度卷积层，未来计划从模型计算图优化的角度出发，设计算法并优化硬件结构以实现全部算子的重量化。

表 5-2 重量化模式下访存和计算时间占比

| 算子类型 | iw、ih、ic | oc | 各部分周期/时间占比 | |
|------|-------------|-----|----------------|-----------------|
| | | | 访存 | 计算 |
| 卷积 | 32、32、128 | 128 | 13312(15.85%) | 70672(84.15%) |
| 卷积 | 104、104、256 | 128 | 205824(11.77%) | 1542989(88.23%) |
| 卷积 | 208、208、256 | 64 | 722944(18.39%) | 3208415(81.61%) |
| 卷积 | 416、416、32 | 64 | 190976(10.57%) | 1615411(89.43%) |
| 深度卷积 | 32、32、32 | 32 | 1042(14.87%) | 5967(85.13%) |
| 深度卷积 | 96、96、128 | 128 | 39128(18.23%) | 175532(81.77%) |
| 深度卷积 | 208、208、256 | 256 | 450784(21.63%) | 1633343(78.37%) |
| 深度卷积 | 416、416、128 | 128 | 900576(21.25%) | 3337766(78.75%) |

表 5-3 中针对 AXI 总线数据写回过程中传输阻塞对性能的影响进行分析，其中卷积层和深度卷积层的 kernel 和 stride 均设置为 1。由于脉动阵列的数据吞吐量较大，当 kernel 或输入通道数较小时会出现较为严重的数据阻塞问题。值得庆幸的是，在卷积神经网络中常采用大量的卷积核以提高特征信息的提取能力，此时特征图通道数较多而不会出现 AXI 数据写回阻塞的现象。

表 5-3 AXI 总线数据写回阻塞对性能的影响

| 算子类型 | OFM 排列格式 | iw、ih、ic | oc | 阻塞时间占比 |
|------|-----------|------------|-----|--------|
| 卷积 | NCHW@FP32 | 32、32、32 | 128 | 67.46% |
| 卷积 | NCHW@FP32 | 128、128、32 | 64 | 67.59% |
| 卷积 | NHWC@INT8 | 32、32、32 | 128 | 67.26% |
| 卷积 | NHWC@INT8 | 128、128、32 | 64 | 67.57% |
| 深度卷积 | NCHW@FP32 | 64、64、64 | 64 | 65.12% |
| 深度卷积 | NCHW@FP32 | 128、128、32 | 32 | 55.58% |
| 深度卷积 | NHWC@INT8 | 64、64、64 | 64 | 65.05% |
| 深度卷积 | NHWC@INT8 | 128、128、32 | 32 | 55.54% |

5.2 DC 综合结果

本文采用 Synopsys 的 Design Compiler 工具对 NPU 设计进行电路综合，并根据综合报告对整体电路的资源消耗、面积开销、时序路径等关键问题进行分析，详细参数设置如表 5-4 所示。其中，综合频率设置为 800MHz，输入延时和输出延时设置为时钟周期的 30%，工艺库采用 SMIC 28nm 并在 worst 工艺角下综合设计。现阶段本设计在 800MHz 在时钟频率下已经时序收敛，然而在实际应用中，由于时钟偏斜、时钟抖动等因素的影响，常在电路综合过程中设置时序裕量以降低后端版图布局难度。若采用典型 20% 的时序裕量设计，则可保证 NPU 在 666MHz 的主频下运行，峰值算力高达 2.728TOPS@INT8。

表 5-4 Design Compiler 综合参数设置

| 编号 | 参数 | 值 |
|----|----------------|--|
| 1 | Frequency | 800MHz |
| 2 | Input Delay | 0.375ns |
| 3 | Output Delay | 0.375ns |
| 4 | Corner | Worst |
| 5 | Target Library | scc28nhkcp_hdc35p140_rvt_ssg_v0p81_-40c_ccs.db |

表 5-5 展示 NPU 综合结果中各 cell 的资源消耗，其中，combinational cell 使用量最大，约为 sequential cell 和 buf/inv 使用量的四倍。除此之外，设计中 SRAM 使用 ARM 的 memory compiler 生成，未在统计范围内。综合来看，NPU 整体面积约为 5.5 mm²，各模块面积占比如图 5-1 所示。硬件设计方案中，数据缓冲区主要包括 ifmBuffer、wgtBuffer 以及 ofmBuffer。其中，ifmBuffer 采用 DPRAM 设计，大小为 512KB，占了 NPU 整体面积的 53.7%，资源消耗较大；在未来的研究工作中，需要对 ifmBuffer 在 BANK 层次上细分，根据特征图排列特点设计高效的数据缓存结构。考虑到相同尺寸缓冲区 DPRAM 实现方案的面积消耗约为 SPRAM 方案的两倍，在深入研究中将使用 SPRAM 进行资源类型的替换。其次，ofmBuffer 采用 TPRAM 搭建同步 FIFO，同样占用了较大面积；但硬件设计过程中，各单元采用模块化设计思想，后期可以采用 SPRAM 扩展数据位宽进而搭建同步 FIFO 实现替换，同时针对 ofmBuffer 对整体硬件加速性能影响展开分析，进而减小缓冲区面积。

表 5-5 NPU 资源消耗汇总

| 编号 | 资源 | 使用量 |
|----|--------------------|--------|
| 1 | combinational cell | 998295 |
| 2 | sequential cell | 237568 |
| 3 | buf/inv | 232510 |

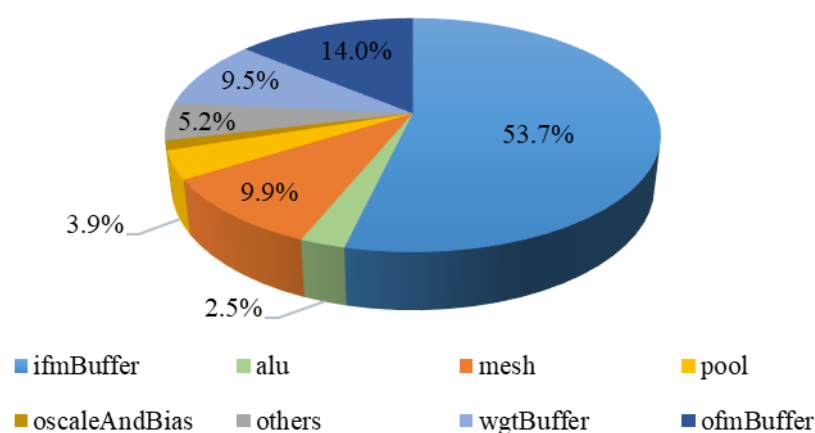


图 5-1 NPU 各模块面积占比

图 5-2 展示了硬件设计综合频率为 800MHz 的关键路径，其起点位于权重寄存器的选择信号，终点位于 PE 累加的部分和。在后续的研究中，将针对时序报告中的关键数据路径进行优化以进一步提高系统性能。

| Startpoint: mesh/r_16508_reg | | |
|---|---------|----------|
| Endpoint: mesh/PE_947/io_out_dl_r_reg[i9] | | |
| Path Group: clock | | |
| Path Type: max | | |
| Point | Incr | Path |
| clock clock (rise edge) | 0.0000 | 0.0000 |
| clock network delay (ideal) | 0.0000 | 0.0000 |
| mesh/r_16508_reg/CK (DQV2_140P7T35R) | 0.0000 | 0.0000 r |
| mesh/r_16508_reg/Q (DQV2_140P7T35R) | 0.0000 | 0.1016 r |
| mesh/PE_947/io_ctl_sel (PE_512_76) | 0.0000 | 0.1016 r |
| | | |
| mesh/PE_947/io_out_dl_reg[19]/D (DQV2_140P7T35R) | 0.0000 | 1.1998 r |
| data arrival time | | 1.1998 |
| clock clock (rise edge) | 1.2500 | 1.2500 |
| clock network delay (ideal) | 0.0000 | 1.2500 |
| mesh/PE_947/io_out_dl_r_reg[19]/CK (DQV2_140P7T35R) | 0.0000 | 1.2500 r |
| library setup time | -0.0502 | 1.1998 |
| data required time | | 1.1998 |
| data arrival time | | -1.1998 |
| slack (MET) | | 0.0000 |

图 5-2 关键路径分析

5.3 FPGA 原型验证

5.3.1 SoC 系统搭建

本文提出的面向深度学习加速的软硬件系统不仅在 Verilator 联合仿真器中进行验证，还实现了 FPGA 原型系统的验证。SoC 系统的搭建采用米联客 ZYNQ UltraScale+

MPSOC 15EG 开发板，其搭载了一颗型号为 Xilinx XCZU15EG-2FFVB1156I 的 FPGA 芯片，内部可用资源如表 5-6 所示。除此之外，该芯片采用异构设计，包含了运行频率最大为 1333MHz 的 4 核 Cortex-A53 ARM CPU 以及运行频率最大为 533MHz 的 Cortex-R5 实时处理器。同时，板载 8GB DDR4（PL 和 PS 侧各 4GB）、64MB Flash 和 8GB EMMC 等丰富的硬件资源，最大功耗 40W。

表 5-6 XCZU15EG-2FFVB1156I 芯片内部资源统计

| 编号 | 资源类型 | 总量 |
|----|-------------|--------|
| 1 | Logic Cells | 747K |
| 2 | LUT | 341K |
| 3 | Flip Flops | 682K |
| 4 | Block RAM | 3.28MB |
| 5 | Ultra RAM | 3.94MB |
| 6 | DSP | 3528 |
| 7 | BUFG | 404 |

FPGA 原型系统的 SoC 级组成如图 5-3 所示，包括 PS 和 PL 两个部分。PS 部分采用单核 ARM A53 进行裸核软件程序开发，软硬件系统借助 MPSOC 双通道的 HP 接口共享 PS 侧 4GB DDR 以进行数据交互。同时，采用 CPU 集成的 UART 和 SDIO 外设实现板级调试、输出信息打印、模型和图像的存储等功能。PL 部分主要为本文设计的 NPU 硬件 IP，借助 AXI Interconnect / AXI Smartconnect 实现 AXI 总线的矩阵互联。由于 NPU 采用双通道 AXI 进行数据传输，并采用单通道 AXI_Lite 进行指令加载，因此三条独立总线需要搭配多个总线互联矩阵。

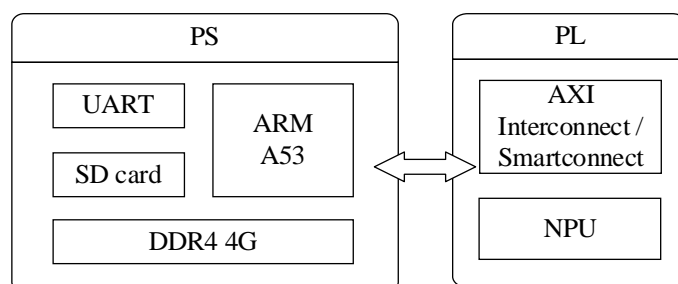


图 5-3 SoC 系统组成

图 5-4 展示了 SoC 系统在 Vivado 软件中 Block Design 的连接图。PL 端时钟源于 PS 端时钟树并通过 DPLL 分频得到 200MHz 的时钟，硬件复位与 CPU 的复位信号相连，借助 Processor System Reset IP 完成复位信号的同步，需要注意的是，硬件电路中仅 NPU 采用高电平复位，其余 IP 复位信号均低电平有效。AXI_Lite 总线借助 AXI Interconnect 与 PS 端的 HPM 低功耗接口相连，数据位宽为 32，并可根据需求扩展多个外设。出于简洁化布局的考虑，对 NPU 及外围电路进行层次化封装。图 5-5 展示了 NPU 层次下 Block Design 的连接图，NPU 的双通道 AXI 分别通过 AXI Smartconnect 与 PS

端的两个 HP 接口相连，外部中断拼接后输出。

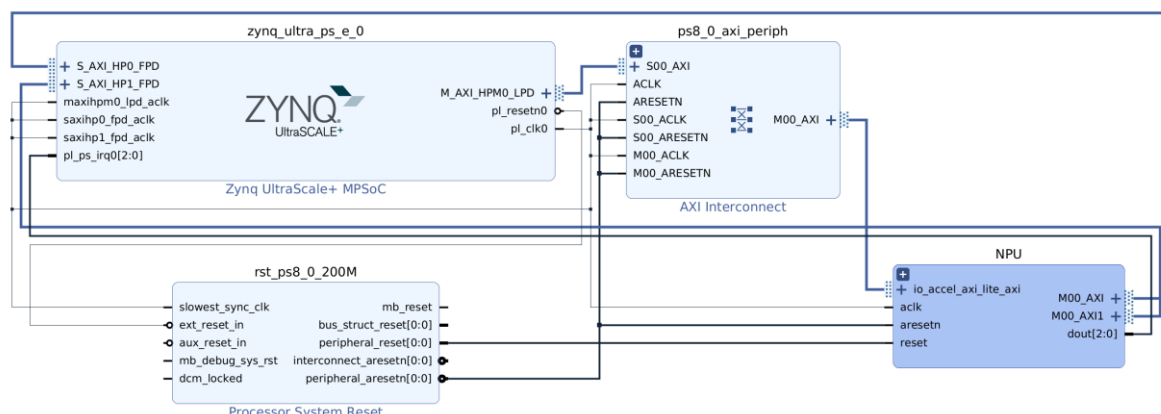


图 5-4 SoC 系统 Block Design 连接图

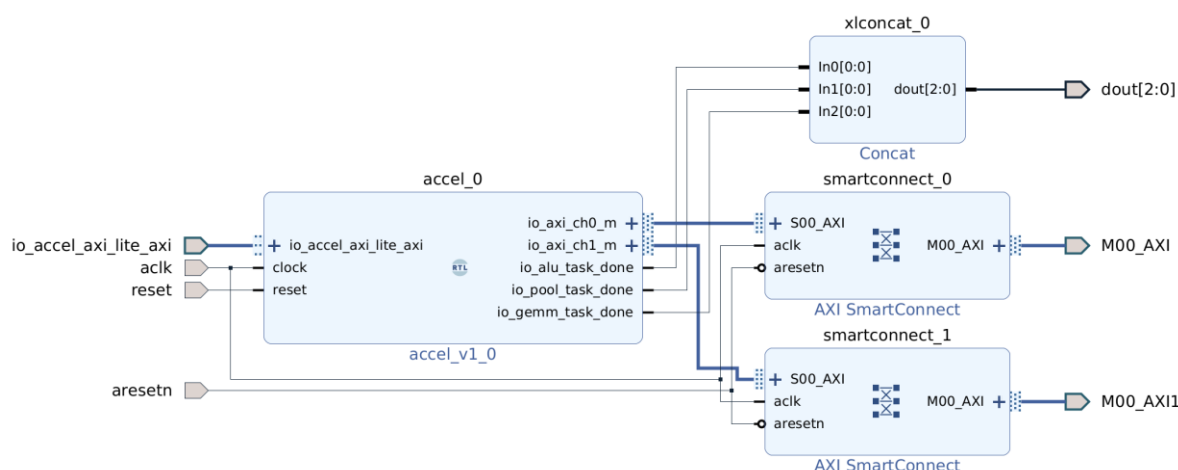


图 5-5 NPU Block Design 连接图

表 5-7 展示了 SoC 系统中总线地址空间的分配情况, CPU 和 NPU 共享 2GB 的 DDR 空间, 两个数据读写通道对应着 NPU_AXI_CH0 和 NPU_AXI_CH1, 软件部分设置堆区和栈区的大小均为 768MB。NPU_CFG 采用 AXI_Lite 总线实现 NPU 指令加载和微量数据传输, 分配地址空间 64KB, 基地址为 0x8000_0000, 其中 NPU 实际使用 512B 的地址段, 其余保留。

表 5-7 FPGA 验证系统 SoC 地址分配

| 名称 | 地址 | 范围 | 功能 |
|-------------|-------------------------|------|-----------|
| NPU_AXI_CH0 | 0x0000_0000~0x7FFF_FFFF | 2GB | 读写 DDR 数据 |
| NPU_AXI_CH1 | 0x0000_0000~0x7FFF_FFFF | 2GB | 读写 DDR 数据 |
| NPU_CFG | 0x8000_0000~0x8000_FFFF | 64KB | NPU 寄存器配置 |

FPGA 系统中包含两组 PL 到 PS 的中断资源可供使用, 共计 16 个中断, 对应中断号为 121~128 和 136~143。NPU IP 采用 3 条外部中断指示 ALU、POOL 和 GEMM 模块的工作状态, 如表 5-8 所示, 均采用上升沿触发, 中断号为 121~123, 通过 concat

的方式接入 PS 端的中断 0 通道。

表 5-8 FPGA 验证系统 SoC 中断号分配

| 名称 | 中断号 | 触发方式 | 功能 |
|----------------|-----|-------|--------------|
| ALU_TASK_DONE | 121 | 上升沿触发 | 指示 ALU 任务完成 |
| POOL_TASK_DONE | 122 | 上升沿触发 | 指示 POOL 任务完成 |
| GEMM_TASK_DONE | 123 | 上升沿触发 | 指示 GEMM 任务完成 |

SoC 系统中 PL 端资源使用情况如表 5-9 所示，整体而言，硬件系统使用了约三分之一的查找表资源，四分之一的触发器资源。其中，Block RAM 和 DSP 消耗较大，主要用于 NPU 中缓冲区和乘法器的映射。

表 5-9 SoC 资源占用情况

| 资源类型 | 总量 | 使用量 | 使用率 |
|------------|--------|--------|--------|
| LUT | 341280 | 114021 | 33.41% |
| LUT RAM | 184320 | 3831 | 2.08% |
| Flip Flops | 682560 | 130738 | 19.15% |
| Block RAM | 744 | 328 | 44.09% |
| Ultra RAM | 112 | 16 | 14.29% |
| DSP | 3528 | 2261 | 64.09% |
| BUFG | 404 | 7 | 1.73% |

表 5-10 例举了 NPU 主要模块的资源消耗情况，Block RAM 的使用主要集中于特征图的输入输出缓冲区，两者合计消耗约占总使用量的 80%；而 DSP 的消耗集中于 Mesh 结构中，消耗量约占总使用量的 90%。硬件方案中 Mesh 微架构包含 1024 个 PE 单元，每个 PE 单元可以在单个周期内完成两组 INT8 的乘加运算，因此，Mesh 结构消耗 DSP 的数量约为 2048 个。Ultra RAM 是 Xilinx UltraScale+ 系列芯片中独有的一类存储资源，开发中可以使用 Xilinx 原语 ram_style 指定，本设计利用该资源搭建权重缓冲区。

表 5-10 NPU 主要模块资源消耗统计

| 名称 | LUT | FF | DSP | BRAM |
|-----------|---------------|---------------|--------------|-------------|
| ALU | 21707(19.04%) | 15310(11.71%) | 126(5.57%) | 4(1.22%) |
| Pool | 18617(16.33%) | 14191(10.85%) | 6(0.26%) | 16(4.88%) |
| Mesh | 13474(11.82%) | 34742(26.57%) | 2049(90.62%) | 0(0.00%) |
| AccMem | 4312(3.78%) | 3331(2.55%) | 0(0.00%) | 32(9.76%) |
| Im2col | 15843(13.89%) | 5997(4.59%) | 27(1.19%) | 128(39.02%) |
| wgtBuffer | 1160(1.02%) | 2130(1.63%) | 4(0.18%) | 0(0.00%) |
| ofmBuffer | 12900(11.31%) | 19200(14.68%) | 0(0.00%) | 128(39.02%) |

表 5-10 NPU 主要模块资源消耗统计（续）

| 名称 | LUT | FF | DSP | BRAM |
|---------------|---------------|---------------|-----------|-----------|
| opFusion | 16801(14.73%) | 23317(17.83%) | 49(2.17%) | 4(1.22%) |
| axiSendBuffer | 1184(1.04%) | 1620(1.24%) | 0(0.00%) | 12(3.66%) |

5.3.2 视频流实时处理系统搭建

本文不仅搭建了 FPGA SoC 级原型验证系统，同时实现了板级的视频流实时处理方案验证。图 5-6 展示了该系统组成，整体仍然采用米联客 MZU15EG 开发套件，并在 SoC 原型系统的基础上添加视频输入输出回路。其中，视频流 HDMI 输入驱动处理在 PL 侧进行，视频流输出则利用与 ARM A53 相连的 DisplayPort 接口实现。目前，该显示方案支持常见目标检测和目标分类算法的实时视频处理。

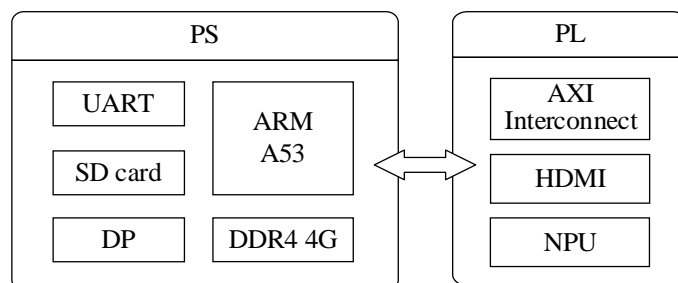


图 5-6 视频流实时处理系统组成

图 5-7 截取了视频流实时处理原型系统在 Vivado Block Design 开发过程中的 IP 级连接，并对 NPU 和 HDMI 输入单元的 RTL 设计进行了层次化封装。NPU 子系统组成如图 5-5 所示，与 SoC 原型系统保持一致。HDMI 子系统主要包含视频帧缓存、DMA、以及 HDMI 信号接收芯片 AD7611 驱动等电路。同时 Address Map 中设置输入视频流在 DDR 中的缓存地址，完成输入数据的软硬件交互。

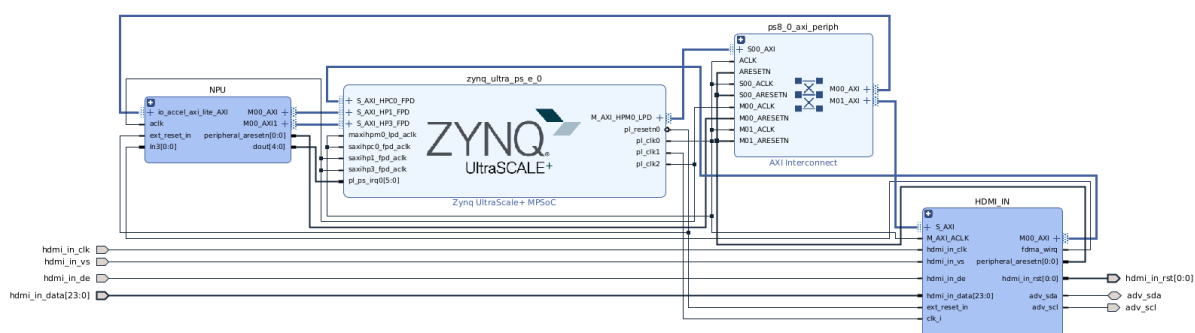


图 5-7 视频流实时处理系统 Block Design 设计

图 5-8 为实验室环境中搭建的视频处理系统实物图，并在右下角的 FPGA 板卡中部署了 MobileNet_SSD 目标检测模型。左侧笔记本电脑通过复制屏幕信息的方式输入实时显示画面至 FPGA 板载的 HDMI 信号接收芯片，采用 NPU 硬件加速逐帧进行识别，并将显示结果通过 DP 视频接口输出至另一台显示器。

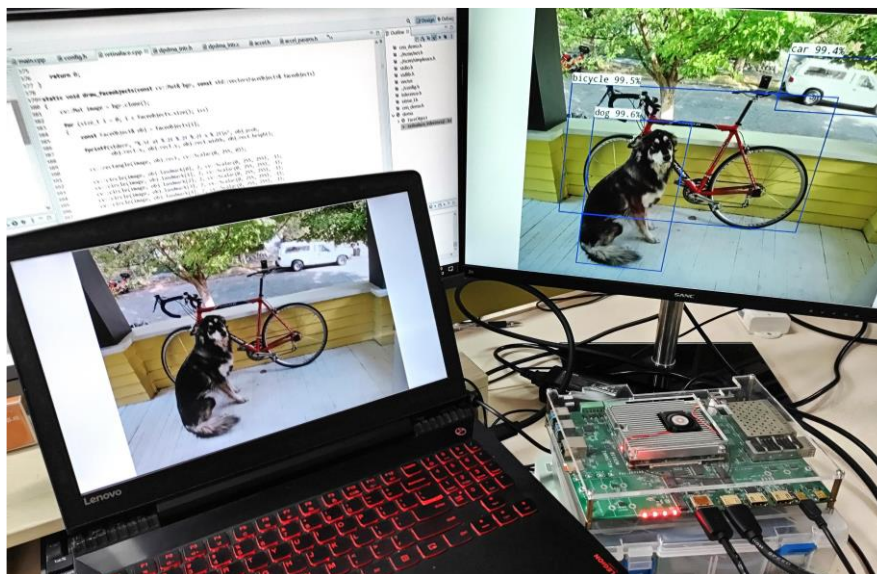


图 5-8 视频流实时处理系统实物图

5.3.3 NPU 推理精度分析

1) MobileNet_SSD NPU 推理精度分析

MobileNet_SSD 是一种融合了 MobileNet 和 SSD 的轻量级深度学习模型，主要用于目标检测任务，特别适用于边缘端和移动端等资源和计算能力有限的设备。精度实验中网络模型使用 PASCAL VOC 进行训练，该数据集包含车辆、家具用品、动物和人 4 个大类、并可细分成自行车汽车、桌椅沙发、猫狗牛羊、人等 20 个小类。图 5-9 和图 5-10 给出了 PC 端和 SoC 原型系统中推理结果对比，其中 PC 端使用未量化 FP32 模型并在 Linux 环境下使用 NCNN 推理引擎完成推理，SoC 原型系统则采用量化后的 INT8 模型进行硬件加速。

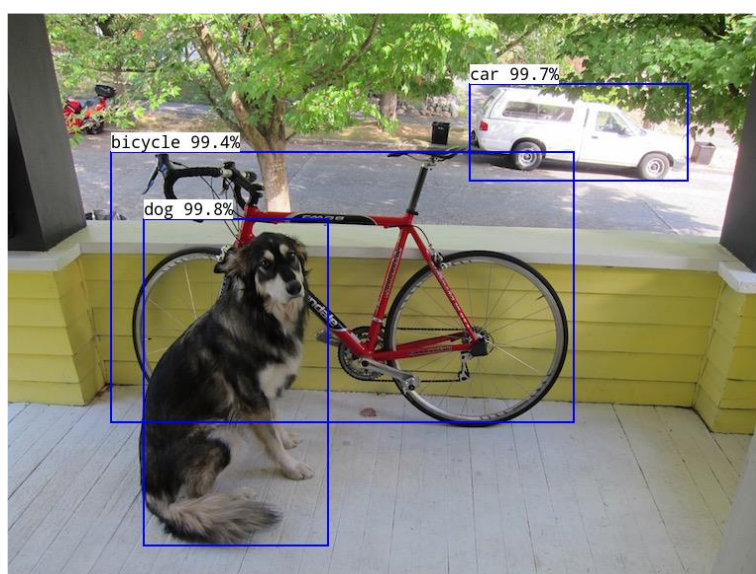


图 5-9 PC 端 MobileNet_SSD 模型未量化时 NCNN 推理结果

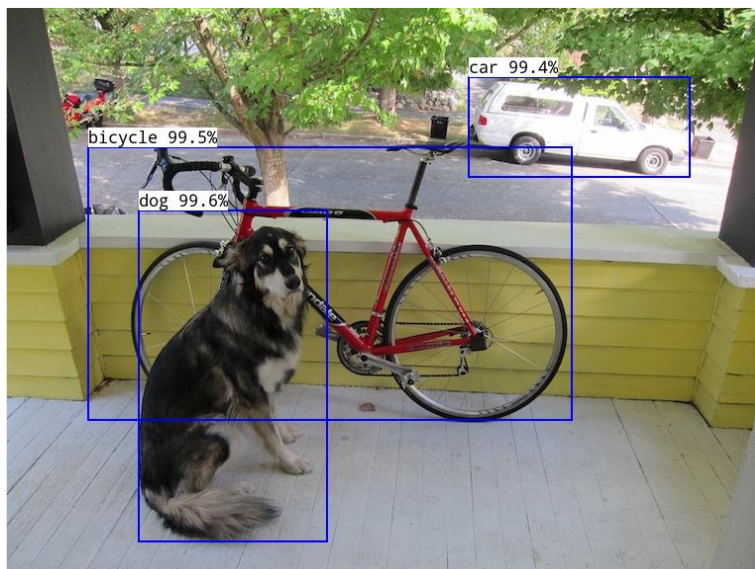


图 5-10 NPU 端 MobileNet_SSD 模型量化后推理结果

在图 5-9 和图 5-10 所示的推理结果中可以明显看出 INT8 量化后的模型仅有轻微的精度下降，满足嵌入式应用需求。图 5-11 截取了 NPU 推理过程中串口的打印信息，主要包含硬件初始化信息、目标图像参数、模型文件加载情况以及检测结果等信息。SoC 软件系统中移植 stbi_image 库和 NCNN 的 simpleCV 库实现了图像的读取、写回和简易的图像处理功能。而串口打印检测结果包含了目标的识别编号、置信度和位置信息。

```
Xilinx Zynq MP First Stage Boot Loader

Release 2021.2 Apr 5 2024 - 20:41:27
PMU-FW is not running, certain applications may not be supported.

[ log ]: SD INIT PASS
[ log ]: INTR INIT PASS
[ log ]: dog.jpg is 163759 bytes
[ log ]: stbi_load w = 768
[ log ]: stbi_load h = 576
[ log ]: stbi_load c = 3
[ log ]: stbi_load desired_channels = 3
[ log ]: mobilenet_ssd_int8.param is 16892 bytes
[ log ]: mobilenet_ssd_int8.bin is 5922816 bytes
12 = 0.99566 at 134.98 206.76 190.64 x 335.11
2 = 0.99463 at 83.08 142.90 488.62 x 276.83
7 = 0.99411 at 467.68 71.63 223.74 x 101.90
[ log ]: imshow save image to image.png
[ log ]: enter imwrite
[ log ]: waitKey stub
```

图 5-11 NPU 端 MobileNet_SSD 模型推理过程中串口打印

2) RetinaFace NPU 推理精度分析

RetinaFace 是商汤科技提出的一种高性能多任务人脸检测模型，可以实现高精度的人脸定位、以及人脸关键点如眼睛、鼻尖、嘴角的提取。该网络架构采用多级特征融合策略以实现不同尺寸的人脸检测，并搭配多任务学习、上下文信息和密集锚点采样

等技术实现了精确的关键点定位。

图 5-12 和图 5-13 给出了 RetinaFace 模型在 PC 端和 SoC 原型系统中的精度对比，与 MobileNet_SSD 采用相同的训练后对称量化方案。从两者的识别结果来看，SoC 原型系统中人脸检测的置信度轻微下降、且面部关键点位置识别准确。

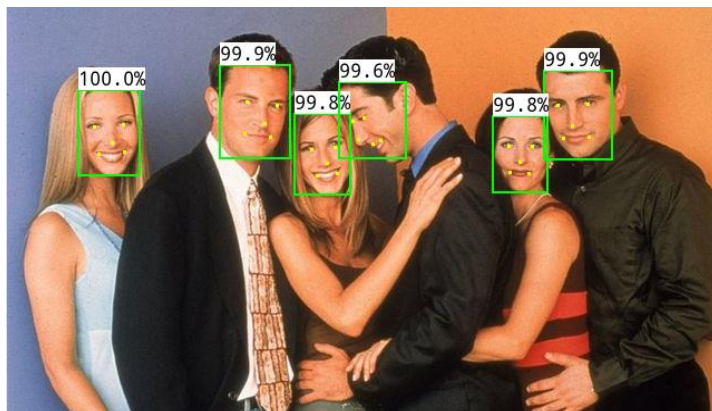


图 5-12 PC 端 RetinaFace 模型未量化时 NCNN 推理结果

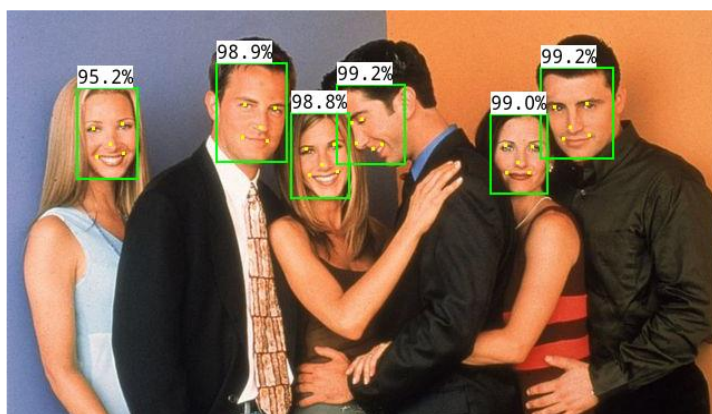


图 5-13 NPU 端 RetinaFace 模型量化后推理结果

5.3.4 NPU 加速性能分析

在 FPGA 原型验证系统中, NPU 单元和 AXI 互联矩阵的时钟频率设置为 200MHz, 时序收敛且仍有较大的裕量。为了方便对 NPU 各功能模块的计算性能进行分析, 本文在 SoC 系统中对各算子层进行独立测试, 并将 NPU@200MHz 的推理时间与 ARM Cortex-A53@1GHz 的结果相比较。ARM A53 采用 ARMv8 架构, 对比测试实验中, 软件程序编译使用 Xilinx ARMv8 GNU Toolchains, 并添加-O3 选项进行优化编译。

表 5-11 和表 5-12 展示了 NPU GEMM 单元进行卷积层和深度卷积层算子加速的推理时间和 CPU 的执行时间对比结果。其中, 硬件卷积和深度卷积方案支持任意特征图尺寸、任意 kernel 以及任意 stride。在卷积加速的对比实验中, CNN 网络中常见尺寸特征图的 NPU 计算时间可以控制在数千微秒及以下, 而单核 ARM A53 则需要数千至数万毫秒, 整体来看, NPU@200MHz 相比 A53@1GHz 约有两千倍的性能提升。而深度卷积的性能对比实验中, 由于单个卷积核仅有一个通道, 数据量较小, 在执行推理

任务时 PE 利用率较低，因此硬件执行速度相比 A53@1GHz 仅有数十倍的性能提升。

表 5-11 Convolution 算子层的推理时间对比

| iw、ih、ic、oc | k | s | NPU@200MHz | A53@1GHz | 加速比 |
|----------------|---|---|------------|------------|-------|
| 32、32、128、512 | 3 | 2 | 704us | 1297.49ms | 1843x |
| 32、32、256、256 | 3 | 1 | 2015us | 5601.58ms | 2780x |
| 32、32、256、512 | 1 | 1 | 812us | 2221.64ms | 2736x |
| 64、64、128、1024 | 3 | 2 | 3779us | 10902.71ms | 2885x |

表 5-12 ConvolutionDepthwise 算子层的推理时间对比

| iw、ih、ic、oc | k | s | NPU@200MHz | A53@1GHz | 加速比 |
|-----------------|---|---|------------|----------|--------|
| 128、128、128、128 | 3 | 2 | 3.14ms | 94.04ms | 29.95x |
| 128、128、256、256 | 3 | 1 | 9.89ms | 667.95ms | 67.54x |
| 128、128、512、512 | 1 | 1 | 16.26ms | 916.09ms | 56.34x |
| 64、64、1024、1024 | 3 | 2 | 7.46ms | 204.91ms | 27.47x |

表 5-13 展示了 NPU ALU 单元与 ARM A53 进行激活函数算子加速的推理时间对比，硬件层向量 ALU 计算核心采用流水线设计，在执行不同复杂度的激活函数时消耗时间相差不大，表中对比实验数据仅对 Swish 算子层进行汇总。特征图激活函数的计算属于逐元素计算的范畴，因此，整体计算时间和特征图元素个数呈正相关。相比于 A53@1GHz，NPU@200MHz 实现了六十余倍的性能提升，且性能提升效果随算子复杂度的增加而增加。除此在外，在 NPU 执行卷积和深度卷积的过程中，实现了激活函数的算子融合，进而掩盖了硬件激活的时间损耗。

表 5-13 Swish 算子层的推理时间对比

| iw、ih、ic | NPU@200MHz | A53@1GHz | 加速比 |
|-------------|------------|-----------|--------|
| 256、256、128 | 8.71ms | 568.55ms | 65.27x |
| 256、256、256 | 17.29ms | 1136.59ms | 65.74x |
| 256、256、512 | 34.36ms | 2273.36ms | 66.16x |

在 FPGA 原型验证系统中，不仅对单一算子层的加速能力进行分析，也对 SoC 系统的网络级验证展开实验。表 5-14 例举了常见 CNN 模型在不同平台的推理时间对比，覆盖图像识别、目标分类、人脸检测等应用场景。SoC 异构的设计方案极大增强了深度学习算法的部署能力，支持任意卷积神经网络的端到端部署。在 NPU@200MHz 的平台中，YOLOv3_Tiny、YOLOv4_Tiny 在输入 416×416 的 RGB 图像时均可实现 25FPS 以上的推理速度；MobileNet 在输入 300×300 的 RGB 图像时可实现 34FPS 以上的推理速度；对轻量级的目标分类算法如 SqueezeNet，甚至可以达到 60FPS 以上的推理速度。而在如 ResNet50 的分类网络中，由于其采用了 NPU 现阶段不支持的全局平均池化、且残差块后激活函数尚未支持算子融合，整体加速仅提升 373 倍，后续优化中，可以

结合重量化策略和图优化算法解决以上大部分计算瓶颈。整体来看，NPU@200MHz 相比 A53@1GHz，实现了数百倍甚至数千倍的性能提升。

表 5-14 不同网络的推理时间对比

| 网络类型 | 输入尺寸 | NPU@200MHz | A53@1GHz | 加速比 |
|------------------|---------------------------|------------|----------|-------|
| YOLOv3_Tiny | $416 \times 416 \times 3$ | 36.72ms | 45.28s | 1233x |
| YOLOv3 | $416 \times 416 \times 3$ | 208.37ms | 571.82s | 2744x |
| YOLOv4_Tiny | $416 \times 416 \times 3$ | 40.75ms | 55.73s | 1368x |
| YOLOv5s | $640 \times 512 \times 3$ | 165.63ms | 152.11s | 918x |
| YOLOv7_Tiny | $640 \times 512 \times 3$ | 144.03ms | 138.54s | 962x |
| MobileNet_SSD | $300 \times 300 \times 3$ | 29.26ms | 28.04s | 958x |
| MobileNet_YOLO | $416 \times 416 \times 3$ | 106.03ms | 72.40s | 683x |
| MobileNetv2_YOLO | $352 \times 352 \times 3$ | 48.59ms | 29.97s | 617x |
| SqueezeNet | $227 \times 227 \times 3$ | 16.54ms | 8.31s | 502x |
| ResNet50 | $224 \times 224 \times 3$ | 231.40ms | 86.35s | 373x |
| RetinaFace | $640 \times 360 \times 3$ | 440.25ms | 624.72s | 1419x |

CNN 硬件加速器资源消耗和性能对比的横向对比结果如表 5-15 所示，其中选取了典型的 VGG-16 模型进行比较。整体来看，由于边缘端计算能力和存储资源的限制，各设计方案均对原始模型进行压缩处理。此外，本文提出的 CNN 加速软硬件解决方案通用性和灵活性较强，支持绝大部分 CNN 模型的端到端部署，并能在性能、资源和功耗上取得较好的表现。

表 5-15 CNN 硬件加速器资源消耗与性能对比

| 对比对象 | 本文 | 文献[51] | 文献[52] | 文献[53] | 文献[54] |
|-------------|-----------------------------|------------------------------|---------------------------|------------------------------|-----------------------------|
| CNN 模型 | VGG-16 | VGG-16 | VGG-16 | VGG-16 | VGG-16 |
| 硬件平台 | Xilinx MPSOC XCZU15EG | Xilinx Virtex-7 VX980T | Xilinx ZYNQ XC7Z045 | Xilinx Virtex-7 VX690T | Intel Arria 10 GX1150 |
| 时钟频率 (MHz) | 200 | 150 | 150 | 200 | 240 |
| 量化精度 | INT8 | FP8/16 | FP16 | FP8 | FP8/16 |
| DSP 消耗 (个) | 2261 | 3395 | 780 | 2877 | 3036 |
| 功耗 (W) | 4.13 | 14.36 | 9.63 | / | 40 |
| 平均性能 (GOPS) | 819 | 1000 | 137 | 1805 | 968 |

5.4 本章小结

本章首先基于 Verilator 仿真环境分析了软硬件系统中目前存在的性能瓶颈。其次

搭建了 FPGA SoC 原型验证系统和视频流实时处理平台，并对 NPU 的加速性能和推理精度进行对比分析。最后，结合 Design Compiler 综合报告对 NPU 的性能、资源占用和面积等情况进行分析。

6 总结与展望

6.1 总结

随着人工智能技术的快速演进，数据量的指数级增长已经成为一种新常态。传统的通用计算架构由于计算密度低、功耗大等问题在边缘端应用受到很大限制，而领域特定架构技术和半导体工艺的进步给硬件底层提供了更多新鲜的途径。本文设计了一款面向 CNN 硬件加速的高性能通用 NPU，搭配软件解决方案可以实现边缘端神经网络的端到端部署，具体研究工作包括以下几个方面：

1) 采用 Chisel 敏捷开发，设计了适用于 CNN 硬件加速的通用计算架构。

在本文的研究工作中，设计了一款以脉动阵列为核心的 CNN 硬件加速器，采用权重固定的数据流格式，最大限度的进行数据复用、资源复用。针对传统脉动阵列中 PE 利用率低等问题，本文增加权重预加载数据通路以实现高效的密集计算。同时在计算核心中添加向量 ALU 单元以支持逐元素算子层的硬件加速。此外，在硬件微架构的设计过程中，充分考虑了多种数据流优化方案，采用算子融合、重量化、大尺寸特征图分块、Layout 变换等方法有效降低了数据访问延时和硬件计算时间。

2) 开发硬件适配的 AI 推理引擎，实现了神经网络算法的端到端部署。

为了简化神经网络部署流程并提高软硬件系统的灵活性和通用性，本文采用异构的设计方案。在神经网络前端工具链中，提供了模型转换、模型优化和模型量化工具，以 NCNN 推理引擎为基础，进行底层框架、硬件后端等优化，实现了 CNN 算法的快速部署。在边缘端的组织架构中，CPU 负责神经网络的调度和非典型算子的计算，NPU 负责常用密集计算的硬件加速。同时，在边缘端的软件栈中实现了驱动层、加速库、应用层代码的解耦设计，极大降低了应用二次开发的难度。

3) 基于 Verilator 开源仿真器，搭建了软硬件联合仿真框架。

针对传统 RTL 仿真验证过程繁琐、耗时耗力等问题，本文结合 Verilator 开源仿真器搭建了软硬件联合仿真框架。在仿真环境中，支持硬件模块级、算子级、网络级仿真粒度，支持神经网络的端到端部署。为了最大限度的降低电路验证工作量，本文在仿真框架中设计了许多数据和时序校验工具，可以实现快速的批量测试、随机测试、数据自动校验、AXI 时序诊断、故障定位、性能分析、波形记录等功能。同时，该仿真框架不仅支持硬件的 Debug，也支持软件驱动层、应用层代码的仿真分析。

最后，本文在 FPGA 平台中搭建了 SoC 原型验证系统和视频流实时处理系统。相比于 ARM A53@1GHz，在主流 CNN 推理速度中 NPU@200MHz 实现了百倍甚至千倍的性能提升。根据 Design Compiler 综合结果，采用 SMIC 28nm 工艺库且保留 20% 时序裕量，NPU 最大运行频率可达 666MHz，提供 2.728TOPS@INT8 的峰值算力。

6.2 展望

本文提出的软硬件方案实现了 CNN 模型简洁高效的端到端部署，具有较强的灵活性和可扩展性。然而由于本人的个人能力和时间精力的限制，该设计仍然存在很多细节值得进一步优化，具体包括以下几个方面：

1) ifmBuffer 设计有待进一步优化。本文的 ifmBuffer 由总大小为 512KB 的 DPRAM 拼接而成，造成较为严重的资源和面积浪费。未来计划采用 SPRAM 实现，并根据特征图内存模型优化缓冲区结构，减小访存延时和资源浪费。

2) 缩小硬件各数据缓冲区的尺寸。在硬件加速方案中，采用较多的数据缓冲区以连接不同吞吐量的模块。然而该过程中可能存在不合理的参数设置，造成资源浪费。未来计划基于本文提出的 Verilator 软硬件联合仿真器，分析各缓冲区深度对整体的性能影响并进行参数修改。

3) 支持的硬件加速算子不够丰富。现阶段 NPU 支持卷积、深度卷积、池化、激活等常用算子映射。未来计划实现反卷积、维度转换等算子的添加和映射。

4) 重量化策略支持范围受限。目前软硬件的重量化策略仅支持相邻的卷积层、相邻的深度卷积层、以及卷积层和深度卷积层之间。未来计划针对神经网络计算图进行深度地优化和融合，将重量化策略推广到全部算子层，进而将有效提高硬件运算效率、缩短访存延时。

5) 网络模型支持种类较少。目前本文提出的软硬件方案仅支持 CNN 的端到端部署，未来计划增加 RNN、Transformer 等模型的部署能力。

致 谢

转眼间已经在求学的道路上奔波了十九年，我也从小学时的淘气包变成了如今的大人模样。高中生、本科生、研究生每一阶段都在努力地完成心中的理想追求，虽然随着年龄的增长，我也曾多次懈怠与颓废，也曾多次改变自己的理想追求。面对即将结束的学生生涯，心中难免有很多不舍，非常幸运在学习和成长的过程中遇到了很多良师益友。

首先最感谢的是我研究生的恩师梁峰教授，是他带我敲开数字 IC 的大门，在我最迷茫的时候给我抛出橄榄枝。梁老师学术严谨、为人谦和、宽容大度、能认真听取学生的意见，作为我们心目中的榜样，指引我们前进的道路。回顾整个研究生生活，竞赛和科研项目贯穿了我整个数字前端的学习历程。感谢梁老师给我们树立正确的价值观，在学术研究中给我们留有自由发挥的空间、也能在我们举步维艰时给出专业的建议。在潜移默化的影响中，我们也怀揣着一个求知的心去探索未知的世界，用发展的眼光去理解行业的变化、去拥抱崭新的领域。

此外，我也非常感谢本科阶段的熊兰老师和廖勇老师，他们严谨的科研、治学态度在我初入大学时备受震撼，也是他们带领我第一次走进嵌入式、电控的世界。感谢他们言传身教，使我逐渐养成了良好的科研习惯。

其次我还要感谢我的教研室小伙伴们，朝夕相处中我们一同学习、一同生活。感谢孙齐伟、付海生、成舒婷、肖飞豹、王宗轩、余文麟等师兄师姐在求职中给我很多帮助，也非常感谢王永强、陈昇杰、曹琪、程才菲、郝渊、张铭旭、官利民等师弟师妹在科研和生活中给我很多的支持和理解。感谢刘硕同学的默默陪伴、理解包容我。

最后，感谢我的父母和家人，他们支持我做的一切决定，感谢他们这么多年来对我的鼓励和呵护，让我始终带着自信和勇气迎接任何挑战。

参考文献

- [1] Chen Y H, Krishna T, Emer J S, et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks[J]. IEEE journal of solid-state circuits, 2016, 52(1): 127-138.
- [2] Chen Y H, Emer J, Sze V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks[J]. ACM SIGARCH computer architecture news, 2016, 44(3): 367-379.
- [3] Chen Y H, Yang T J, Emer J, et al. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices[J]. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2019, 9(2): 292-308.
- [4] Chen T, Du Z, Sun N, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning[J]. ACM SIGARCH Computer Architecture News, 2014, 42(1): 269-284.
- [5] Chen Y, Luo T, Liu S, et al. Dadiannao: A machine-learning supercomputer[C]//2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2014: 609-622.
- [6] Du Z, Fasthuber R, Chen T, et al. ShiDianNao: Shifting vision processing closer to the sensor[C]//Proceedings of the 42nd Annual International Symposium on Computer Architecture. 2015: 92-104.
- [7] Liu D, Chen T, Liu S, et al. Pudiannao: A polyvalent machine learning accelerator[J]. ACM SIGARCH Computer Architecture News, 2015, 43(1): 369-381.
- [8] Zhang S, Du Z, Zhang L, et al. Cambricon-X: An accelerator for sparse neural networks[C]//2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2016: 1-12.
- [9] Zhang C, Li P, Sun G, et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks[C]//Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays. 2015: 161-170.
- [10] 刘彪, 陈长林, 张宇飞等. 稀疏卷积计算高效数据加载与输出缓存策略 [J]. 国防科技大学学报, 2023, 45(05): 212-221.
- [11] 谭龙, 严明玉, 吴欣欣等. 面向稀疏卷积神经网络的 CGRA 加速器研究 [J]. 高技术通讯, 2024, 34(02): 173-186.
- [12] Parashar A, Rhu M, Mukkara A, et al. SCNN: An accelerator for compressed-sparse convolutional neural networks[J]. ACM SIGARCH computer architecture news, 2017, 45(2): 27-40.
- [13] Lu W, Yan G, Li J, et al. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks[C]//2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017: 553-564.
- [14] Kung H T, Leiserson C E. Systolic arrays (for VLSI)[C]//Sparse Matrix Proceedings 1978. Philadelphia, PA, USA: Society for industrial and applied mathematics, 1979, 1: 256-282.
- [15] 徐睿. 基于脉动阵列的卷积神经网络硬件加速器体系结构研究 [D]. 国防科技大学, 2022.
- [16] Shen J, Ren H, Zhang Z, et al. A high-performance systolic array accelerator dedicated for CNN[C]//2019 IEEE 19th International Conference on Communication Technology (ICCT). IEEE, 2019: 1200-1204.
- [17] Bachrach J, Vo H, Richards B, et al. Chisel: constructing hardware in a scala embedded language[C]//Proceedings of the 49th Annual Design Automation Conference. 2012: 1216-1225.
- [18] 梁峰, 吴斌等. 敏捷硬件开发语言 Chisel 与数字系统设计 [M]. 北京: 电子工业出版社, 2022.

- [19] Jia L, Lu L, Wei X, et al. Generating systolic array accelerators with reusable blocks[J]. IEEE Micro, 2020, 40(4): 85-92.
- [20] Xu R, Ma S, Wang Y, et al. Configurable multi-directional systolic array architecture for convolutional neural networks[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2021, 18(4): 1-24.
- [21] Wang B, Ma S, Liu Z, et al. SADD: A Novel Systolic Array Accelerator with Dynamic Dataflow for Sparse GEMM in Deep Learning[C]//IFIP International Conference on Network and Parallel Computing. Cham: Springer Nature Switzerland, 2022: 42-53.
- [22] Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit[C]//Proceedings of the 44th annual international symposium on computer architecture. 2017: 1-12.
- [23] Moreau T, Chen T, Vega L, et al. A hardware–software blueprint for flexible deep learning specialization[J]. IEEE Micro, 2019, 39(5): 8-16.
- [24] Chen T, Moreau T, Jiang Z, et al. TVM: An automated End-to-End optimizing compiler for deep learning[C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018: 578-594.
- [25] Genc H, Kim S, Amid A, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration[C]//2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021: 769-774.
- [26] Genc H, Haj-Ali A, Iyer V, et al. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures[J]. arXiv preprint arXiv:1911.09925, 2019, 3: 25.
- [27] Asanovic K, Avizienis R, Bachrach J, et al. The rocket chip generator[J]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016, 4: 6.2.
- [28] Zhao J, Korpan B, Gonzalez A, et al. Sonicboom: The 3rd generation berkeley out-of-order machine[C]//Fourth Workshop on Computer Architecture Research with RISC-V. 2020, 5.
- [29] 卢北辰, 杨兵. 面向存算架构的神经网络数字系统设计 [J] . 微电子学与计算机, 2024(04): 1-10.
- [30] 唐成峰, 胡炜. 应用于忆阻器阵列存内计算的低延时低能耗新型感知放大器 [J] . 微电子学与计算机, 2024(02): 58-66.
- [31] Jiang H, Peng X, Huang S, et al. CIMAT: A transpose SRAM-based compute-in-memory architecture for deep neural network on-chip training[C]//Proceedings of the International Symposium on Memory Systems. 2019: 490-496.
- [32] Tu F, Wang Y, Wu Z, et al. ReDCIM: Reconfigurable Digital Computing-In-Memory Processor With Unified FP/INT Pipeline for Cloud AI Acceleration[J]. IEEE Journal of Solid-State Circuits, 2022, 58(1): 243-255.
- [33] Lattner C, Amini M, Bondhugula U, et al. MLIR: A compiler infrastructure for the end of Moore's law[J]. arXiv preprint arXiv:2002.11054, 2020.
- [34] Eldridge S, Barua P, Chapyzhenka A, et al. MLIR as hardware compiler infrastructure[C]//Workshop on Open-Source EDA Technology (WOSET). 2021.
- [35] Jiang X, Wang H, Chen Y, et al. Mnn: A universal and efficient inference engine[J]. Proceedings of Machine Learning and Systems, 2020, 2: 1-13.
- [36] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.
- [37] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Advances in neural information processing systems, 2012, 25.
- [38] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2015: 1-9.

-
- [39] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.
- [40] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.
- [41] Huang G, Liu Z, Van Der Maaten L, et al. Densely connected convolutional networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2017: 4700-4708.
- [42] Iandola F N, Han S, Moskewicz M W, et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size[J]. arXiv preprint arXiv:1602.07360, 2016.
- [43] Howard A G, Zhu M, Chen B, et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications[J]. arXiv preprint arXiv:1704.04861, 2017.
- [44] Zhang X, Zhou X, Lin M, et al. Shufflenet: An extremely efficient convolutional neural network for mobile devices[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 6848-6856.
- [45] Redmon J, Farhadi A. Yolov3: An incremental improvement[J]. arXiv preprint arXiv:1804.02767, 2018.
- [46] Han S, Mao H, Dally W J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding[J]. arXiv preprint arXiv:1510.00149, 2015.
- [47] Ding X, Zhou X, Guo Y, et al. Global sparse momentum sgd for pruning very deep neural networks[J]. Advances in Neural Information Processing Systems, 2019, 32.
- [48] Frankle J, Carbin M. The lottery ticket hypothesis: Finding sparse, trainable neural networks[J]. arXiv preprint arXiv:1803.03635, 2018.
- [49] He Y, Kang G, Dong X, et al. Soft filter pruning for accelerating deep convolutional neural networks[J]. arXiv preprint arXiv:1808.06866, 2018.
- [50] Liu Z, Mu H, Zhang X, et al. Metapruning: Meta learning for automatic neural network channel pruning[C]//Proceedings of the IEEE/CVF international conference on computer vision. 2019: 3296-3305.
- [51] Huang W, Wu H, Chen Q, et al. FPGA-based high-throughput CNN hardware accelerator with high computing resource utilization ratio[J]. IEEE Transactions on Neural Networks and Learning Systems, 2021, 33(8): 4069-4083.
- [52] Guo K, Sui L, Qiu J, et al. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA[J]. IEEE transactions on computer-aided design of integrated circuits and systems, 2017, 37(1): 35-47.
- [53] Yin S, Tang S, Lin X, et al. A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 38(4): 678-691.
- [54] Ma Y, Cao Y, Vruthula S, et al. Automatic compilation of diverse CNNs onto high-performance FPGA accelerators[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 39(2): 424-437.

攻读学位期间取得的研究成果

- [1] 梁峰, 秦海鹏, 曹琪, 陈昇杰. 基于蜂鸟 E203 RISC-V 内核的智能识别 SoC 设计与应用系统开发. 第六届全国大学生集成电路创新创业大赛全国二等奖, 工业和信息化部人才交流中心, 2022-08.
- [2] 梁峰, 秦海鹏, 陈昇杰, 曹琪. 高性能实时目标检测加速器设计. 第五届全国大学生 FPGA 创新设计竞赛全国二等奖, 中国电子学会, 2022-07.
- [3] 梁峰, 秦海鹏, 陈昇杰, 曹琪. 基于 FPGA 的高性能实时目标检测硬件加速器设计. 第十七届中国研究生电子设计竞赛西北赛区一等奖, 中国电子学会, 2022-12.

答辩委员会会议决议

论文对 CNN 加速器软硬件实现方案进行了深入的研究与设计，选题具有重要的学术意义和应用价值。论文完成的主要工作如下。

1) 采用 Chisel 语言设计了 CNN 加速的硬件架构，提供了矩阵、向量、标量计算的加速能力，并通过算子融合和重量化等优化策略有效降低了系统访存时间。

2) 开发了与硬件加速器适配的推理引擎，实现了神经网络模型的端到端部署；在边缘端裸核运行环境中，采用层次化理念实现了驱动层、加速库及应用层代码的解耦。

3) 搭建了软硬件联合的仿真框架，支持硬件模块级、算子级、网络级的仿真粒度，并集成了数据自动校验、故障定位等功能。

4) 在 FPGA 平台上搭建了原型验证系统并进行了性能测试与分析，结果达到了预期目标。

论文写作认真，态度科学严肃，工作量大，研究工作表明作者已掌握了本学科的基础理论和专门知识，具备了独立从事科研工作的能力。

答辩中陈述清晰，回答问题准确。经答辩委员会讨论，一致同意通过论文答辩，并建议授予秦海鹏工学硕士学位。

常规评阅人名单

本学位论文共接受 2 位专家评阅，其中常规评阅人 2 名，名单如下：

| | | |
|-----|-----|--------|
| 梅魁志 | 教授 | 西安交通大学 |
| 杨晨 | 副教授 | 西安交通大学 |

学位论文独创性声明（1）

本人声明：所呈交的学位论文系在导师指导下本人独立完成的研究成果。文中依法引用他人的成果，均已做出明确标注或得到许可。论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

本人如违反上述声明，愿意承担以下责任和后果：

1. 交回学校授予的学位证书；
2. 学校可在相关媒体上对作者本人的行为进行通报；
3. 本人按照学校规定的方式，对因不当取得学位给学校造成的名誉损害，进行公开道歉。
4. 本人负责因论文成果不实产生的法律纠纷。

论文作者（签名）： 日期：2024 年 5 月 19 日

学位论文独创性声明（2）

本人声明：研究生所提交的本篇学位论文已经本人审阅，确系在本人指导下由该生独立完成的研究成果。

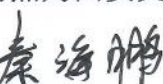

本人如违反上述声明，愿意承担以下责任和后果：

1. 学校可在相关媒体上对本人的失察行为进行通报；
2. 本人按照学校规定的方式，对因失察给学校造成的名誉损害，进行公开道歉。
3. 本人接受学校按照有关规定做出的任何处理。

指导教师（签名）： 日期：2024 年 5 月 19 日

学位论文知识产权权属声明

我们声明，我们提交的学位论文及相关的职务作品，知识产权归属学校。学校享有以任何方式发表、复制、公开阅览、借阅以及申请专利等权利。学位论文作者离校后，或学位论文导师因故离校后，发表或使用学位论文或与该论文直接相关的学术论文或成果时，署名单位仍然为西安交通大学。

论文作者（签名）： 日期：2024 年 5 月 19 日
指导教师（签名）： 日期：2024 年 5 月 19 日

(本声明的版权归西安交通大学所有，未经许可，任何单位及任何个人不得擅自使用)