

学校代号 10532 学号 S2010W0916
分类号 TP391 密级 普通



湖南大学
HUNAN UNIVERSITY

电子信息硕士学位论文

针对国产 ARM 架构 CPU 的深度学习 函数库的设计与优化

学位申请人姓名 谭言西
培养单位 信息科学与工程学院
导师姓名及职称 全哲 教授
学科专业 计算机技术
研究方向 高性能计算、体系结构
论文提交日期 二〇二三年 x 月 xx 日

学校代号： 10532
学 号： S2010W0916
密 级： 普通

湖南大学电子信息硕士学位论文

针对国产 ARM 架构 CPU 的深度学习 函数库的设计与优化

学位申请人姓名： 谭言西
培 养 单 位： 信息科学与工程学院
导师姓名及职称： 全哲 教授
专 业 名 称： 计算机技术
论 文 提 交 日 期： 二〇二三年 x 月 xx 日
论 文 答 辩 日 期： 二〇二三年 x 月 xx 日
答 辩 委 员 会 主 席： 待定

**Design and optimization of deep learning function library for
domestic ARM architecture CPU**

by

Yanxi Tan

B.E. (NanChang University)2018

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of engineering

in

Computer Technology

in the

Graduate School

of

Hunan University

Supervisor

Professor Zhe Quan

June, 2023

湖南大学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名： 日期： 年 月 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于

- 1、保密 ，在_____年解密后适用本授权书。
- 2、不保密 。
(请在以上相应方框内打“□”)

作者签名： 日期： 年 月 日
导师签名： 日期： 年 月 日

摘要

自美国通过芯片法案加强了对与中国的芯片制裁力度以来，让我国在芯片领域的关键核心技术和核心零部件领域被“卡脖子”。为了能够在人工智能深度学习与高性能计算相关领域紧追世界前沿，提升国家自主计算能力，我国在过去几年内自主研发了许多的自研芯片，涵盖了 CPU、GPU、DSP 等。国产 ARM 架构 CPU 是国家在自研 CPU 上的新一次尝试，不仅拥有了更大的缓存空间，还提供了 256 位的向量长度并包含了可变长向量拓展指令集。它继承了 ARM 所属的精简指令集架构的节能且高效的优势，拥有巨大的高性能计算潜力。但是在运行深度学习应用的时候想要完全发挥国产 ARM 架构 CPU 的能力，目前的深度学习应用程序还不足以解决问题，深度学习应用软件栈通常可以在逻辑上分为应用层、支持层和硬件层，根据具体的应用层可以在支持层甚至应用层做相应的是配合优化。比如英伟达半导体公司为自家的 GPU 开发了 CUDA 开发框架以及 CuDNN 深度神经网络库，Intel 专门开发 oneDNN 针对 Intel 的 CPU 以及 GPU 进行了深度的优化。而目前并未有任何深度学习函数库对国产 ARM 架构 CPU 进行深度优化。鉴于这个问题，本文在深度学习应用软件栈的中间层实现了高性能深度学习函数库 FTEngine。为在国产 ARM 架构 CPU 上运行深度学习应用提供了高性能的函数支持。本文主要的工作和创新点包含了以下几个方面：

1. 设计实现了深度学习软件栈中间层的深度学习函数库 FTEngine，通过分析主流卷积神经网络的网络层，在 FTEngine 中实现了基本的卷积层、池化层、归一化层、全连接层以及激活层函数。通过分析网络层的计算逻辑以及计算特征，使用可变长向量拓展指令集的 intrinsic 指令实现了归一化层、全连接层以及激活层函数，使用内联汇编的方式书写了卷积层和池化层的函数内核。利用负载均衡、重构计算等技术优化各网络层函数。

2. 使用 Im2col+GEMM 方式实现高性能卷积层函数，通过分析 Im2col 算法，重新设计了 Im2col 算法在 NCHW 和 NHWC 数据格式下的搬运方式，并使用可变长向量拓展指令集优化实现了 Im2col 内部的高性能拷贝内核。另外，本文借鉴 GOTOBLAS 和 BLIS 的分块方式以及循环框架，根据国产 ARM 架构 CPU 的缓存大小实现了高性能矩阵乘函数。并探讨了高性能矩阵乘的并行优化方式。经实验证明，本文实现的高性能矩阵乘函数能够比 OpenBLAS 在 A64FX 的高性能编译版本快 7%。最终本文实现的 FTEngine 在 16 线程并行度下运行 VGG16 网络模型比 ACL22.08 版本快 6%。

关键词： 国产 ARM 架构 CPU；可变长向量拓展指令集；深度学习；人工智能；

Abstract

Since the United States passed the chip bill to strengthen sanctions against China in the field of chips, our country has been “choked” in the field of key core technologies and core components in the field of chips. In order to keep up with the world’s forefront in the fields of artificial intelligence deep learning and high-performance computing, and to enhance the country’s independent computing capabilities, our country has independently developed many self-developed chips in the past few years, covering CPU, GPU, DSP and so on. The domestic ARM architecture CPU is a new attempt by the country in self-developed CPUs. It not only has a larger cache space but also provides a 256-bit vector length and includes a variable-length vector extension instruction set. It inherits the energy-saving and efficient advantages of the RISC architecture owned by ARM and has huge potential for high-performance computing. However, to fully utilize the capabilities of domestic ARM architecture CPUs when running deep learning applications, current deep learning applications are not sufficient to solve the problem. The deep learning application software stack can usually be logically divided into application layer, support layer and hardware layer. According to specific application layers, corresponding coordination optimization can be done at the support layer or even application layer. For example, Nvidia Semiconductor Company developed CUDA development framework and CuDNN deep neural network library for its own GPUs. Intel specifically developed oneDNN for deep optimization of Intel’s CPUs and GPUs. However, no deep learning function library has been deeply optimized for domestic ARM architecture CPUs. In view of this problem, this paper implements a high-performance deep learning function library FTEngine in the middle layer of the deep learning application software stack. It provides high-performance function support for running deep learning applications on domestic ARM architecture CPUs. The main work and innovations of this paper include the following aspects:

1. The deep learning function library FTEngine in the middle layer of the deep learning software stack was designed and implemented. By analyzing the network layers of mainstream convolutional neural networks, basic convolutional layers, pooling layers, normalization layers, fully connected layers and activation layer functions were implemented in FTEngine. By analyzing the computational logic and characteristics of the network layers, the normalization layer, fully connected layer and activation layer functions were implemented using intrinsic instructions of the variable-length vector extension instruction set. The function kernels of the convolutional layer and pooling layer were written using inline

assembly. Various network layer functions were optimized using technologies such as load balancing and computational restructuring.

2. A high-performance convolutional layer function was implemented using the Im2col+GEMM method. By analyzing the Im2col algorithm, the transportation method of the Im2col algorithm in NCHW and NHWC data formats was redesigned, and a high-performance copy kernel was optimized and implemented using the variable-length vector extension instruction set. In addition, this paper draws on the block method and loop framework of GOTO-BLAS and BLIS to implement a high-performance matrix multiplication function according to the cache size of domestic ARM architecture CPUs. The parallel optimization methods of high-performance matrix multiplication are also discussed. Experiments have shown that the high-performance matrix multiplication function implemented in this paper can be 7% faster than the high-performance compiled version of OpenBLAS on A64FX. Finally, the FTEngine implemented in this paper runs the VGG16 network model at a parallelism of 16 threads 6% faster than ACL22.08 version

Key Words: Domestic ARM architecture CPU; variable long vector expansion instruction set; deep learning; artificial intelligence;

目 录

学位论文原创性声明和学位论文版权使用授权书	I
摘要	II
Abstract	III
插图索引	VII
附表索引	IX
第1章 绪论	1
1.1 研究背景及意义	1
1.2 国内外研究现状	3
1.2.1 人工智能函数库相关研究	3
1.2.2 关于卷积函数优化相关研究	5
1.2.3 基于体系结构优化的相关研究	6
1.2.4 数据并行技术的相关研究	7
1.3 论文组织结构	11
第2章 国产 ARM 架构 CPU 及卷积神经网络介绍分析	13
2.1 国产 ARM 架构 CPU 简介	13
2.2 卷积神经网络层分析	14
2.3 卷积网络层介绍	15
2.3.1 全连接层	15
2.3.2 卷积层	16
2.3.3 池化层	18
2.3.4 归一化层	19
2.3.5 激活层	21
2.4 本章小节	22
第3章 FTEngine 库函数设计实现	23
3.1 FTEngine 总体设计	23
3.2 FTEngine 算子实现	24
3.2.1 池化层函数实现	24
3.2.2 归一化层函数实现	29
3.2.3 全连接层函数实现	31
3.2.4 激活层函数实现	32
3.3 本章小结	35

第 4 章 高性能卷积函数设计与实现	36
4.1 高性能 Im2col 函数实现	36
4.1.1 Im2col 算法分析	36
4.1.2 高性能 Im2col 算法转换设计	37
4.1.3 使用 SIMD 技术实现高性能向量拷贝内核	41
4.2 高性能矩阵乘算法实现	42
4.2.1 分块策略选取	44
4.2.2 缓存分配	45
4.2.3 数据重新排序	48
4.2.4 汇编层面优化	49
4.2.5 多线程优化	51
4.3 本章小节	53
第 5 章 实验与结果分析	54
5.1 国产 ARM 架构 CPU 单核峰值性能实验	54
5.2 矩阵乘性能比较实验	55
5.3 FTEngine 卷积层和池化层性能比较实验	57
5.4 模型运行对比实验	60
5.5 本章小结	61
总结与展望	64
参考文献	67
附录 A 读学位期间所发表的学术论文	71
附录 B 读学位期间所参加的科研项目	72
致 谢	73

插图索引

图 1.1 SIMD 计算示意图	8
图 1.2 Intel SIMD 向量示意图	9
图 1.3 Intel 与 ARM 的 SIMD 发展历史示意图	9
图 1.4 NEON 向量加法示意图	10
图 1.5 ARM SIMD 向量示意图	10
图 2.1 可变长向量寄存器及可变长预测寄存器示意图	13
图 2.2 向量预测求和指令示意图	13
图 2.3 主流卷积神经网络历史发展示意图	14
图 2.4 全连接操作示意图	16
图 2.5 卷积操作示意图	17
图 2.6 池化操作示意图	18
图 2.7 批归一化和层归一化计算作用维度示意图	20
图 3.1 人工智能函数库架构和 FTEngine 的支持层部分	23
图 3.2 池化函数的分层实现抽象图	25
图 3.3 池化函数的分层实现抽象图	26
图 3.4 池化函数的分层实现抽象图	27
图 3.5 最大池化汇编内核核心段汇编代码	28
图 3.6 累加方法示意图	30
图 3.7 重用向量方法的全连接函数示意图 (浅色代表没有计算完全)	31
图 3.8 朴素方法的全连接函数示意图	32
图 3.9 激活层函数类型判断逻辑图	33
图 3.10 intrinsic 指令求最大值函数示例图	33
图 3.11 intrinsic 指令 Softmax 函数核心部分示例图	34
图 4.1 Im2col 算法原图	37

图 4.2 NCHW 数据格式 Im2col 转换示意图.....	38
图 4.3 NHWC 数据格式 Im2col 转换示意图.....	39
图 4.4 Im2col 转换拷贝示意图（虚线代表某个权重访问的元素）.....	39
图 4.5 Im2col 改进算法拷贝边界距离分析图.....	40
图 4.6 stride 为 1 时的向量拷贝内核代码图.....	42
图 4.7 stride 不为 1 时的向量拷贝内核代码图.....	42
图 4.8 矩阵乘示例图.....	43
图 4.9 gotoblas 提出的分块方法.....	44
图 4.10 BLIS 的循环框架图.....	46
图 4.11 32×8 寄存器使用分配图	47
图 4.12 GEBP 数据重排后的数据排布示意图	48
图 4.13 A 矩阵数据重排顺序示意图	48
图 4.14 B 矩阵数据重排顺序示意图	48
图 4.15 核心计算汇编代码示意图	50
图 4.16 分块循环第四次循环示意图	52
图 4.17 分块循环第三次循环示意图	52
图 4.18 分块循环第二次循环示意图	52
图 4.19 分块循环第一次循环示意图	53
图 5.1 峰值性能测试代码核心部分示意图	55
图 5.2 FTEngine GEMM 与 OpenBLAS GEMM 的性能对比图	56
图 5.3 FTEngine GEMM 与 BLIS GEMM 的性能对比图	57
图 5.4 AlexNet 卷积参数下 FTEngine 与 ACL 的卷积性能对比图	60
图 5.5 VGG16 卷积参数下 FTEngine 与 ACL 的卷积性能对比图	60
图 5.6 AlexNet 卷积参数下 FTEngine 与 ACL 的池化性能对比图	61
图 5.7 VGG16 卷积参数下 FTEngine 与 ACL 的池化性能对比图	62
图 5.8 FTEngine 与 ACL 运行 AlexNet 的性能对比图	63
图 5.9 FTEngine 与 ACL 运行 VGG16 的性能对比图	63

附表索引

表 2.1 知名卷积神经网络模型对于网络层的支持表.....	15
表 5.1 本章节测试的环境以及对比软件和工具的详细配置	54
表 5.2 AlexNet 中的池化层计算参数.....	57
表 5.3 AlexNet 中的卷积层计算参数.....	58
表 5.4 VGG16 中的池化层计算参数	58
表 5.5 VGG16 中的卷积层计算参数	58

第1章 绪论

1.1 研究背景及意义

人工智能（Artificial Intelligence, AI）在过去几十年里发展迅猛，源源不断的出现了新颖的算法，基于AI算法不断的涌现了各种技术产品。在生活中，我们有意识无意识的跟AI的接触越来越多了。自动驾驶，语音助手，人脸识别、智能助理，各种技术层出不穷，人工智能在生活发展中扮演着越来越重要的角色。

早在20世纪50年代，图灵在发表的论文《The imitation game》^[1]中提出“机器能否思考？”中提出计算机能否像人脑一样进行思考。图灵提出并只做了第一台“计算机”——图灵机，对于图灵来说，当告诉一台图灵机思考的方式时，图灵机就可以告诉你它思考的终点——结果。图灵是为了让英国赢得第二次世界大战而申请在情报解密部门以解密作为目的研发了图灵机。图灵机的研发，让英国可以破译德国的进攻指示，提前做好部署进而提前的结束第二次世界大战。但实际上在当时对于图灵机来说，它其实是笨重并且迟缓的：即使在知道了部分答案之后再进行计算仍然需要几个小时的时间才能得出结果，它的大小也占据了一个军用库房，但它——世界上第一台“计算机”让第二次世界大战提前了两年结束，让人们意识到了计算科学发展的重要性，也让人们开启了对于人工智能的展望。

受到图灵机的启发和推动，人工智能发展迅速，人们逐渐尝试使用计算机来模拟大脑的神经网络结构，模拟人类大脑的工作原理，从视觉、听觉、触觉等感官接受并且处理信息，再经过过去的经验、当前的猜想进行判断分析，然后通过重复过程的学习吸收，最终得到对于某项事物的认知。后来人们收到了启发不断探索设计出来了人工智能网络模型（Artificial Neural Network），人工智能网络模型包含了输入层、隐藏层、输出层几层单元，在真实的模型环境中，数据通过前期图片处理之后传输到输入层，在内部向输出层计算传播，直到输出层输出结果完成前向传播的整个过程，而如果是在模型训练的情况下，将会将前向传播的输出结果与真实结果计算得到计算误差并根据链式法则计算每一层的误差贡献，根据误差贡献和梯度下降的原理，更新网络中每个参数的值，如此使得下一次前向传播的输出结果更加贴近正确结果。

由于当人工智能算法开始发迹的时候计算机仍处于早期发展时期，能够使用计算机的人较少，并且计算机处理能力低，存储能力有限，往往又是体积庞大到无法灵活携带，所以当时的人工智能算法往往模型简单，处理相对简单的问题并且输出结果比较差。而随着信息时代、物联网时代的到来，在技术的发展下计算机逐渐性能越来越高，体积越来越小，生活处处能够连接上互联网，让人类获得

了越来越多的数据，逐渐探索了越来越复杂的神经网络模型，逐渐发现复杂的神经网络模型具有很优秀的学习泛化能力，可以在对大量数据进行学习后对从未见过的新数据进行正确判断。目前人工智能技术的应用已经十分普遍比如大家常见的手机使用人脸识别、指纹识别解锁；无人飞行机、酒店送餐机器人自动巡航自动避障；部分游戏里面的智能 NPC，能够将视频里的人物进行换脸的视频 AI 换脸。还有在一些特殊领域比如对航空涡轮螺旋桨发动机进行验伤^[2]，或者是应用在工厂流程中对资源步骤进行重新规划，都开始应用上了 AI 技术。然而人工智能模型的推理和运算是需要时间的计算的。在自动驾驶、金融交易等领域中对时间的要求是非常高的，时间的损耗影响的不仅仅是经济的流失更有可能是人身安全的风险，所以对于能够带来更高性能，更高效率的人工智能运算库、人工智能运算计算框架以及人工智能算力加速卡的研发成了近年来的重要研究。

随着中国近几年在多个高技术领域的快速崛起，并且我国开启了对高端技术产业重点发展的战略，以美国为首的西方发达国家出于对于经济形势改变的担忧以及我国技术进步提升国家影响力对他们造成利益威胁，美国率先开展了对中国的关键技术打压，发表了芯片法案。面对对于芯片具有垄断地位的美国进行的技术围剿打压，我们国家必须发展芯片事业，打赢这场关键技术攻坚战。继而，国内近年来涌现了许多的新兴芯片，龙芯、飞腾、申威、壁仞、寒武纪、旷视、瀚博等公司都在逐步流片推出自己的人工智能算力卡，新能源车企蔚来、小鹏、理想等都以启动自研芯片的流程，种种发展都为芯片国产化提供了希望。对一个人工智能算力卡的评估通常有以下几个指标：浮点运算能力、显存容量、显存带宽、功耗、硬件加速特性以及是否有支持的软件框架和运算优化库。在这里，浮点运算和显存容量带宽都是硬件直接可以带来的优势，自芯片产出之后就不能再改变的部分。而功耗和硬件加速特性的使用都依赖于软件框架和运算库的支持。作为软件支持的人工智能运算框架和人工智能运算函数库不仅方便了用户的使用，当优化得当，还可以减少训练推理人工智能模型时候的实际资源消耗。而每个厂商的算力卡之间甚至是同一个厂商不同的算力卡都不能用同一套运算库去使用。

随着 2020 年富岳使用 ARM 架构的 A64FX 芯片^{[3][4]}凭借 415PFLOPS 运算能力成为了新一代的超级计算机榜首，并以快 1 倍多的优势超过第二名超级计算机，成为第一个以 ARM 架构获得世界超级计算机能力排行榜榜首的超级计算机，并接连在 2020 年 11 月一集 2021 年 6 月霸榜。“富岳”让人们认识了 ARM 架构的性能威力；而后苹果推出的基于 ARM 架构研发的 M1 桌面级 CPU 也是让人们眼前一亮，也为 ARM 架构吸引了一大批关注。ARM 属于精简指令集架构（RISC, Reduced Instruction Set Computing），相对于以 Intel X86 为首的复杂指令集（CISC, Complex Instruction Set Computing）架构而言，精简指令集指令条数少，并且功能简单且采取统一的指令格式，指令字长统一，可以更充分的利用流水线来对程序

进行优化。并且作为精简指令集架构的 ARM 芯片和 RISC-V 芯片都具有可定制性和可扩展性高的特点，可以供客户针对自己的需求进行独家定制扩展。

国产 ARM 架构 CPU 是国家对于自研芯片的又一次大胆尝试，具有拥有超强性能的潜力，所以针对国产 ARM 架构进行函数优化适合有必要的，使用可扩展向量拓展指令集从汇编层面优化 AI 库的算子更好的发挥国产 ARM 架构 CPU 的处理器性能是非常有必要的。

1.2 国内外研究现状

1.2.1 人工智能函数库相关研究

为了让人工智能模型运行的更加迅速，人们选择开发出运行速度更快的库，人们开发出高性能库来加速开发运行人工智能模型的时间最早可以追溯到在上世紀五十年代，那时候高性能人工智能函数库主要是以一些独立的软件包的形式存在，这些软件包通常是指的一些工具函数，可以在早期避免从头到尾的重复开发，里面涵盖了例如矩阵运算、向量运算、分类算法、聚类算法等等。其中一个最早的函数库是数学软件包 BLAS^[5] (Basic Linear Algebra Subprogram)，最早在上世紀 70 年代被开发出来，它包含了三个层级：BLAS level1、BLAS level2、BLAS level3，他们代表了向量和向量之间的运算、向量和矩阵之间的运算以及矩阵和矩阵之间的运算。但这并不是一个真正意义上的人工智能函数库。

随着深度学习的崛起与科技和信息时代的到来，衍生出了一系列的技术方向，包括计算机视觉、文字语音识别、自然语言处理、推荐系统、风险预测与评估、智能监控等领域。这些技术的广泛应用深刻的改变了人们的生产和生活方式，也加大对人工智能运行速度的需求，在这样的热潮下，为了提高模型的运行速度，逐渐涌现了许多的深度学习框架和人工智能函数库。

Caffe^[6]是由贾扬清在加州大学伯克利分校攻读博士期间创建和开发的深度学习框架。它采用 C++ 编写，并支持 Python 和 Matlab 接口。底层仅支持 CUDA，因此只能在 NVIDIA 显卡上运行。Caffe 对于基于深度卷积神经网络的算法具有良好的支持效果，可方便地进行 CNN 模型的训练和测试。

TensorFlow^[7]是一款深度学习框架，由谷歌大脑团队于 2015 年开发。它的核心特点是使用有状态的数据流图进行数值计算，其中数学运算表示为结点，不同结点之间通过多维数组进行数据通信，这些多维数组被称为“Tensor”张量，因此得名 TensorFlow。尽管 TensorFlow 是目前最流行的深度学习框架之一，但它是一个底层的框架，需要编写大量代码来完成各种深度学习任务。近年来，TensorFlow 不断发展，引入了动态图、可视化工具包等功能，支持多平台、分布式运行和多 GPU/CPU 等功能，并支持了谷歌自主研发的神经网络加速器 TPU^[8]

(Tensor Process Unit)。

PyTorch^[9]是 Facebook 人工智能研究所推出的深度学习框架，它的前身是 Torch。相比于 TensorFlow，PyTorch 具有动态图的支持，即图是在运行时生成的，允许处理可变长度的输入和输出；PyTorch 还支持动态图的自动微分，可以大大简化代码的编写。PyTorch 使用 Python 语言，开发者只需要掌握 Numpy 语言和基本的深度学习概念就可以使用 PyTorch 搭建网络。相比之下，TensorFlow 需要手动完成许多复杂的任务，通常需要更多的工作才能复现 PyTorch 的功能。

JAX^[10]是 TensorFlow 在面对 PyTorch 的冲击和启发之下推出的新一代深度学习框架，基于开源 Flax 神经网络库结合 Autograd 和 XLA（Accelerated Linear Algebra）。其中 Autograd 可用于计算 numpy 函数组的导数，XLA 则是一款谷歌开发的机器学习编译器，供 JAX 在不同后端获得最优的并行方案。JAX 将成为谷歌大脑团队替换 TensorFlow 的方案。

oneDNN^[11]（oneAPI Deep Neural Network Library）是 Intel 公司开发的深度神经网络函数库，他的前身是 MKLDNN，主要服务于 Intel 平台下的服务器芯片比如：凌动处理器、至强处理器等等，针对它们进行了深度优化，使用 SSE4.1 和 AVX512 指令集进行了向量化优化。oneDNN 具有很优秀的拓展性，它支持使用 OpenCL^[12]（Open Compute Language）对 Intel 的 Gen 和 Xe 架构 GPU 拓展，同时支持链入 ACL^[13]（Arm ComputeLibrary）来获得对 Arm CPU 的支持提升。

cuDNN^[14]（CUDA Deep Neural Network Library）是 NVIDIA 的深度神经网络函数库，针对 NVIDIA 的 GPU 采用 CUDA（Compute Unified Device Architecture）进行实现，并针对 NVIDIA 的 TensorCore 特性进行了专门优化。

ACL（Arm Compute Library）是 Arm 公司的开源人工智能和视觉函数库，包含了深度学习常用函数以及计算机视觉前后处理的常用函数实现，支持 ARM 的 CPU 使用 OpenCL 支持 GPU，使用多线程技术和 NEON 指令集进行优化。

以上列举的是近十年国外主流深度学习框架库，国内近十年也诞生了有 MNN^[15]（Mobile Neural Network），TNN^[16]（Tencent Neural Network），Paddle^[17]（Parallel Distributed Deep Learning）等高性能机器学习深度学习框架，大都主要支持算法模型的训练、推理、部署三大环节。目前的算法开发人员主要还是以 Pytorch 和 TensorFlow 开发算法，再根据需求移植到不同的推理训练框架上。然而根据计算设备不同，计算平台不同，通常对设备的支持程度不同，对于设备的特性的发挥程度往往有很大的差异。

尽管以上的深度学习框架和人工智能深度学习函数库表现十分优秀并经受了市场考验，虽然部分人工智能函数库对 ARM 架构有一定的支持，但是它们对于国产 ARM 架构 CPU 并未做出专门支持，没有针对国产 ARM 架构 CPU 缓存结构和体系结构进行专门优化，所以我们需要针对国产 ARM 架构 CPU 上执行的函数

进行针对性的设计和优化。

1.2.2 关于卷积函数优化相关研究

卷积函数是神经网络深度学习中最重要的图算子。在一些 CNN 模型中，卷积函数所占用的时间可以占据整个网络模型运行时间的 80% 以上。为了加速神经网络的运行速度，提高卷积函数的运行速度是非常重要的。最近几年对神经网络算子优化成为了一股热潮，有许多的工作学者探究有关优化卷积函数的方法技巧。

Andrew 和 Scott^[18]两人基于 Winograd 开创的最小滤波算法，提出了 Winograd 卷积算法，他们将卷积计算的方法进行转换通过增加加法的方式减少乘法的次数来加快卷积运算；Michael Mathieu^[19]等人将离散傅立叶变化应用于卷积的实现上，使用傅立叶域中的乘积操作完成卷积操作。微软研究院的 Kumar Chellapilla 等人提出使用 Im2col+GEMM^[20]的方法对输入特征图进行转换实现了卷积的操作。对比这几个常见的卷积优化实现方法，Andrew 和 Scott 的工作表明 Winograd 算法能在小型卷积核情况获得优秀的性能表现，傅立叶变换卷积算法只能适用于特征映射数较多的情况，Im2col+GEMM 的卷积方法类似于傅立叶变换卷积算法，适用于卷积规模较大的情况下使用。但 Im2col+GEMM 的卷积方法和傅立叶变换卷积算法都有相同的诟病，就是会引入额外的内存开销，在部分的特殊情形中，Im2col 甚至可以占用整个卷积算法中大部分的时间消耗。

为了减少 Im2col 算法耗时长的问题，Wang Haoyu^[21]等人提出了基于连续内存步进访问的方法优化了 Im2col 算法的内存读取加快了 Im2col+GEMM 的卷积速度；苹果的工程师 Minsik Cho^[22]提出了一种优化内存使用的卷积方式，通过探讨数据被重用的特征，尽可能的减少了内存的消耗，并最终仍然使用 GEMM 的高性能内核，它的算法需要在内部将输入图转换成 NHWC 的格式进行。

Marat Dukhan^[23]在 2019 年提出了非直接卷积，使用更小的间接缓冲区替换了 Im2col 算法带来的内存开销，极大的减少了 Im2col 算法带来的时间损失和内存消耗问题，但是非直接卷积算法却无法避免只能支持 NHWC 格式输入的问题和只能支持前向传播的问题。黄春^[24]等人采用了使用高性能批处理矩阵乘的方式优化批量卷积性能，思路相较独特。

庄晨^[25]提出 FastConv，采用了 ARM Neon 指令集优化了 Winograd 卷积和 Im2col+GEMM 卷积，并提出了自动搜寻高性能内核的方法，在华为鲲鹏上获得了非常高的性能表现。

ZhangJunYang^[26]学者针对 FT2000 的边缘计算情况对卷积算法进行优化，将卷积核进行重排维度，对进行向量乘后在输出维度累加，并融合卷积池化操作，但是他的算法比较依赖输出维度的大小。输出维度较低时不利于整体性能发挥。

1.2.3 基于体系结构优化的相关研究

深度学习应用在不同的计算设备上运行时常常会出现运行效率天差地别，甚至无法运行的问题，因为不同的计算设备之间硬件设备差异巨大，所以对于不同厂商或是定制的的 CPU、GPU、DSA 专用芯片、FPGA(Field-programmable gate array) 设备，会专门实现自己的内核 kernel 函数以操纵所拥有的 Tensor Core、CPU 的向量寄存器，或者是 DSA、GPU 中特殊计算单元。总的来说，针对不同架构的优化往往从访存和并行两个方面出发：

1. 在存储方面，需要遵循时间局部性和空间局部性的原则书写程序，在 CPU 中有熟知的内存金字塔结构——内存到常见的多级缓存+TLB (Translate Look-aside Buffer) 再到寄存器的结构；相对应的，在 MT3000^[27]的 CPU+DSP (Digital Signal Processor) 架构中有定制化的 AM (Array Memory)、SM (Scalar Memory)、GSM (Global Shared Memory)，与 CPU 上的内存金字塔架构十分类似，处理单元对于数据的读取访问从前往后依次由快而慢；而在 GPU 中在全局内存到计算核心之间同样也拥有着 L1 缓存、L2 缓存，另外线程也会拥有独占的局部内存。正因为计算设备中的内存结构有越靠近处理器访问越快，越靠近处理器存储容量越小的性质，所以需要合理使用内存结构。对于 GPU 和 DSP 来说，内存可控性较强，可以显示的声明使用共享内存和片上内存，而对于 CPU 设备来说，缓存是不被程序员代码指令控制的，需要利用程序的数据局部性来减少缓存不命中的次数以优化程序提升算力。

2. 在并行方面，在 CPU 中可以分为数据并行、线程并行、指令并行三种。在数据并行 (Data-Level Parallelism) 方面主要是利用向量处理单元或者张量处理单元的硬件设备在一条指令中完成多个数据的操作，具体的并行能力由特定硬件设备提供的向量寄存器以及具体计算时候采用的精度确定。在线程并行 (Thread-Level Parallelism) 方面 AI 库由于运算的特殊性，通常数据是具有连续的规律性，在实际的运算中对多组数据进行相同的操作，所以可以很好的使用线程并行来优化运算，国产 ARM 架构 CPU 中有 64 个核，每个核心之间都互不干扰，都存在独立的缓存单元，而且由于 AI 库中函数通常拥有 4 个甚至 6 个维度，并且大部分的计算是数据不交叉的，所以具有适合并行优化并不存在复杂并发情形，可以比较简单使用多线程进行优化。在高性能计算领域中人们比较熟知的多线程编程环境是 OpenMP^[28]和 MPI^[29]，OpenMP 具有被多款主流编译器诸如 (GCC、Clang、ICC) 都支持的优点，并且操作简单，可以通过外部编译而不修改程序代码的情况自动开启或者关闭多线程的模式，相对 MPI 入手简单、学习简单。在指令集并行 (Instruction-Level Parallelism) 方面，国产 ARM 架构 CPU 属于乱序执行超标量处理器类型，超标量处理器意思是处理器可以在单个周期中取出多条指令，并且利用硬件对指令进行调度，在国产 ARM 架构 CPU 中存在两个向量处理单元，

同一时刻可以完成两组向量寄存器的运算，而乱序执行的特点可以减轻我们对于流水线设计的难度，而流水线需要尽可能地保持满载，以充分利用硬件资源，提高吞吐量。乱序执行可以将多条指令重排执行，避免因为某条指令的延迟而导致流水线停顿，从而保证流水线的顺畅运行。此外，乱序执行还可以减少因为分支预测失败而导致的流水线清空和重装的操作，从而提高 CPU 的效率和性能。

1.2.4 数据并行技术的相关研究

(1) SIMD

1972 年 Michael J. Flynn 根据 CPU 中的硬件之间可以传递的流可以分为指令和数据两种，而进一步的将计算机根据处理指令集和数据流的区别分为了单指令流单数据流^[30](Single Instruction stream Single Data stream)，单指令流多数据流(Single Instruction stream Multiple Data stream)，多指令流多数据流(Multiple Instruction stream Multiple Data stream)以及多指令流单数据流(Multiple Instruction stream Single Data stream)四种计算机类型。其中多指令单数据流机器暂时仅存在于理论之中，而单指令单数据流的结构在最基础最早出现的结构，在流水线化操作流时他同样可以实现一定的并行性。单指令流多数据流常常体现在图形处理器之中，在 GPGPU 上使用一个控制器控制多个处理器来完成计算任务通常可以获得远超 CPU 的效率。但这里我们讨论的是 CPU 端的微处理器的 SIMD 结构。当 CPU 端拥有特定的向量寄存器时候就可以凭借 SIMD 的特性获得很大程度的性能提升。向量体系结构相比标量体系结构主要有以下的优势：

1. 在每个向量载入或存储操作中付出较长的存储器延迟时间，而不需要在载入或存储向量中每个元素时耗费时间。
2. 减少了流水线互锁的频率，多个指令间可能由于相关性而引起停顿，向量体系结构中每条向量指令仅需一次流水线停顿，而不是每个向量元素操作需要一次。
3. 大幅缩减了动态指令带宽需求。向量体系结构使原本在标量体系中每个元素操作需要一条指令，变成了每个向量操作仅需要一条向量指令。

Intel 的 SIMD 拓展最早开始于 1997 年，那时候推出的 SIMD 拓展叫做 MMX^[31](Multi Media eXtensions)，MMX 在当时主要为了提高 CPU 处理图像的能力，但是当时的 MMX 使用的 MM0~MM7 寄存器是 CPU 中浮点寄存器的一部分，所以将会与浮点计算产生冲突，不能同时进行。

接着到了 1999 年，Intel 在奔腾三上做了名为 SSE^[32](Streaming SIMD Extension) 的新的扩展，后来 AMD 公司也在 2001 年支持了 SSE，与 MMX 中不同的是，SSE 采用了独立的寄存器组 XMM0~XMM7，在 64 位工作时为 XMM0~XMM15。并且这些寄存器的长度也增加到了 128-bit，还增加了 32-bit 的控制寄存

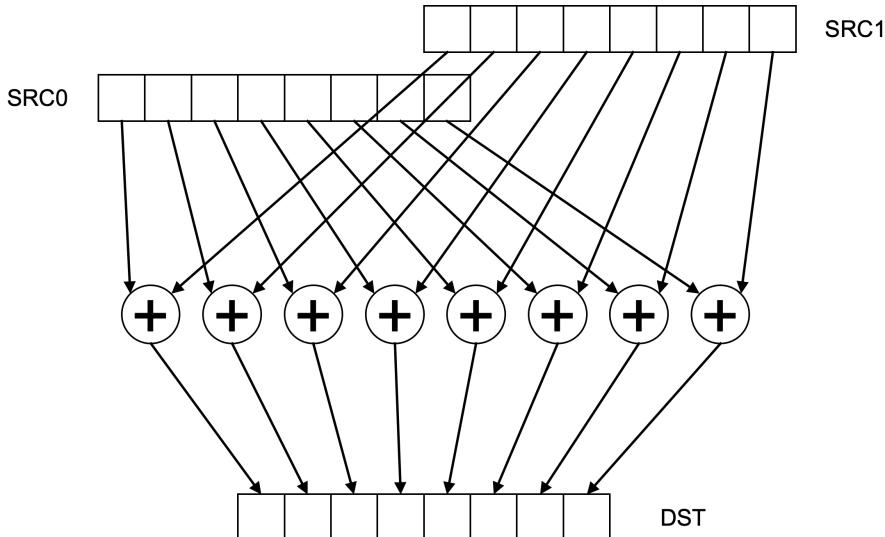


图 1.1 SIMD 计算示意图

器 MXCSR (Multimedia eXtension Control and Status Register)，能够控制 SSE 指令集的精度和摄入方式等等，并且 SSE2 开始支持浮点类型运算，解决了浮点类型无法数据并行的问题。

接着到了 2000 年，SSE2 开始支持 128 位的整数运算和 64 位双精度浮点运算，而后直到 2007 年，SSE 系列的 SIMD 着重于提升指令集的功能性，逐步提供了针对图像视频处理、字符串处理、密码算法相关的新型指令，并未提出对于高性能计算有帮助的提升。

直到 2011 年随 Intel Sandy Bridge 微架构引入全新的 AVX^[33] (Advanced Vector extensions)。将 128-bit 的向量指令拓展到了 256-bit 长向量指令，并且从指令的程度进行了改善，相较于 SSE，AVX 新增了三元操作指令，可以通过新增一个目的操作数用于存储结果从而使得可以一条指令代替之前的两条指令（计算 + 存储），它可以减少一次数据移动加速指令运行，在一个周期内完成多个操作，显著提高计算速度。2011 年推出的 AVX2 则是 AVX 为处理整型数据而进行的迭代，同时增加三个浮点三元操作指令。2016 年推出的 AVX-512^[34]，将原来拓展的 256-bit 的寄存器直接提升到了 512-bit，提高了 CPU 的能效比，并且新的 ZMM 寄存器的低 256-bit 也可以与 256-bit 的 YMM 进行混用，同时 AVX-512 开始支持了 opmask，可以控制特定位的寄存器参与计算，或是使用特定位的计算结果。

总的来说，Intel 在 SIMD 上的发展趋势是，越来越长的向量寄存器，越来越高效，越来越完备的指令集的提供。Intel 在向量拓展指令集体系上迭代更新十分迅速。

ARM (Advanced RISC Machine) 至今为止只更新了两代 SIMD 体系结构，第一代属于 2005 年 Armv7 时候推出的高级 SIMD NEON^[35]，ARMv7 NEON 主要有以下特性：NEON 拥有 16 个 128 位寄存器也可以作为 32 个 64 位寄存器使用。它

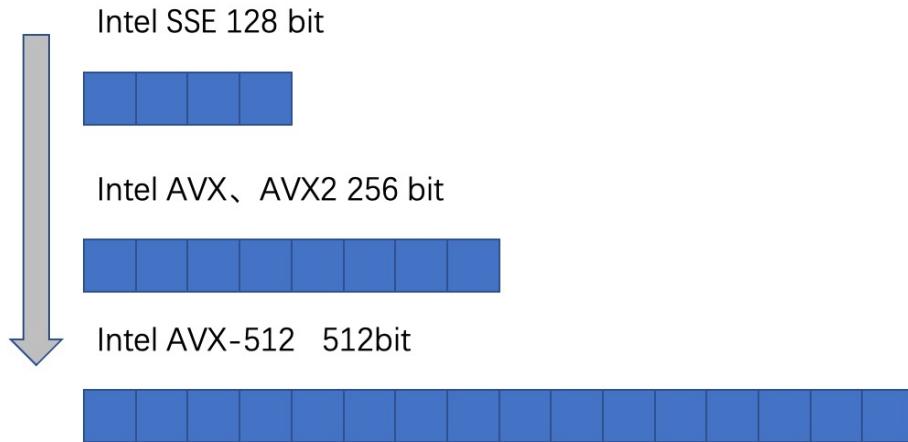


图 1.2 Intel SIMD 向量示意图

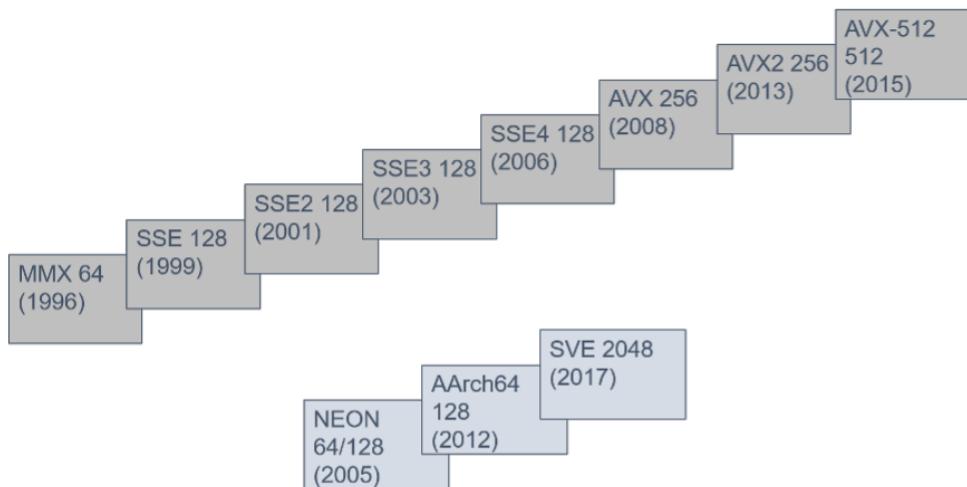


图 1.3 Intel 与 ARM 的 SIMD 发展历史示意图

支持多种整数操作，支持兼容性的单精度浮点操作、半精度以及指令条件执行，它设计出来主要是为了面对在 CPU 端加速多媒体处理任务；当到了 2012 年升级到了 ARMv8 架构时候，NEON 也改进到了 NEONv2，支持双精度的浮点运算以及 64 位的整形运算，支持复数类型的计算需求，NEON 还支持整数与向量直接进行计算。图 1.5 就展示了 NEON 指令“VADD.I16 Q0, Q1, Q2 半精度整型加法”。在这些改进下，NEON 指令集开始更加适用于通用计算而不仅仅只是在 21 世纪初时候需要的多媒体计算了。到了 2017 年，随着市场的需求越来越高，高性能计算和并行计算需要更多非常规的数据类型和复杂的数据结构，而目前的 ARM NEON 无法满足未来人工智能高性能计算多样化的需求。

基于 NEON 指令集不是很好满足高性能计算和人工智能的需求，SVE^[36](Scalable Vector Extension, 可变长向量拓展) 在 2018 年应运而生，可变长向量拓展指令集是支持可变向量长度的指令集，由于 ARM 处理器的可定制性可能让不同的厂商用户根据自己的需求定制不同的 CPU 向量位宽，使用传统的指令集实现的函数库无法在另一种架构上使用，可变长向量拓展指令集支持任意 128

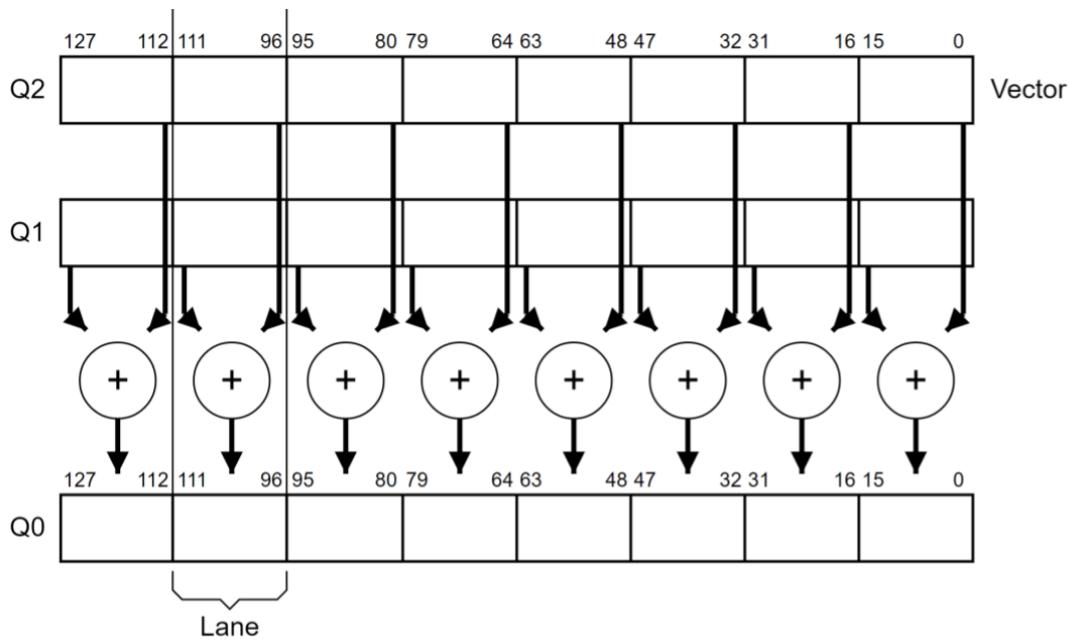


图 1.4 NEON 向量加法示意图

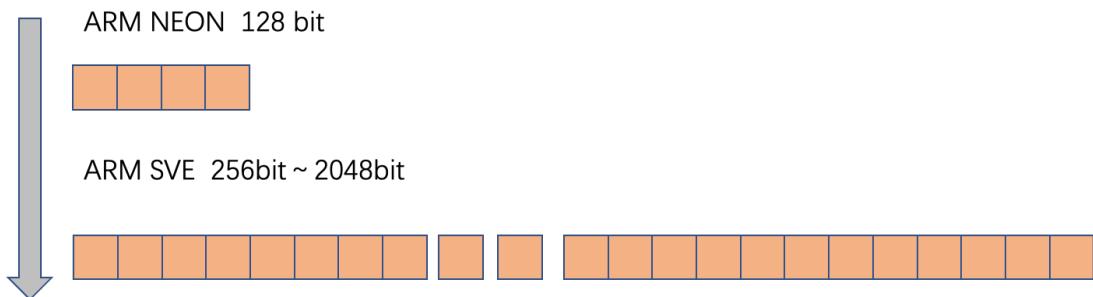


图 1.5 ARM SIMD 向量示意图

位整数倍的向量长度，最高可以支持到 2048 位。在不同向量长度的架构上可以使用同一指令集，这使得用户写出的程序可以具有良好的迁移性。为了提供这种可变向量长度的特性，ARMv8+sve 架构提供两种寄存器：向量寄存器和预测寄存器。预测寄存器是其他架构都未提供的寄存器，预测寄存器与向量寄存器一一对应，预测寄存器可以指导指令是否存取使用向量寄存器相应的位，让 SIMD 的操作更加的灵活也间接辅助提供了每通道预测，统一收集和分发发送等多种功能。

目前为止使用 SIMD 技术可以通过三种方式：

1. 编译器提供的自动向量化优化，部分编译器可以在使用较高优化级别时候识别可以向量化的代码，采取自动向量化的优化方式使用 SIMD 加速程序，但是编译器提供的自动向量化无法针对复杂情况进行优化，必须得通过手动优化。
2. 使用汇编语言或者在程序中内联汇编的方式直接使用向量操作指令。使用汇编语言直接操作运算步骤、操作寄存器的使用，主动排“取算存”流水来获得相较于其他两种方式最快的性能提升，但是汇编语言也会带来编写困难等问题。
3. 使用 Intrinsic^[37]内置函数，调用 SIMD 的 C 拓展函数和 C 拓展操作符。这

是自主便携 SIMD 代码的最简便的方法，也是最被常用的方法，因为用法就如同调用常见的函数，并且在 C++ 代码中还可以支持重载功能，配合 auto 关键字忽略数据类型只用考虑操作函数。

(2) SIMT

SIMT (Single Instruction Multiple Thread) 体系结构是一种多线程处理器体系结构，其中每个线程都执行相同的指令，但是操作不同的数据。SIMT 体系结构最典型的产品是 GPU (图形处理器)，它是一种专门在个人电脑等一些移动设备上做图像和图形相关运算工作的微处理器。与 CPU 相比，GPU 上包含更多的运算单元，且更适合来处理深度学习底层的计算任务；GPU 也拥有更大的带宽，拥有更高的数据吞吐量，支持大批量的数据处理。

目前市场上用于深度学习训练和推理的 SIMD 架构硬件设备主要是 NVIDIA GPU。NVIDIA GPU 在硬件包含多个 SM (Streaming Multiprocessor)，每个 SM 中包含多个 CUDA 核心，NVIDIA GPU 单个 SM 中的线程是以线程束 (Thread Warp) 为单位进行调度的，以一组 32 个线程为单位。因此，理论上 NVIDIA GPU 中可同时运行的线程数量远远超过 CPU 中线程的数量。在程序设计时需要充分挖掘计算任务的并行性，并将计算任务尽可能均衡分配到每个 SM，以发挥最大的硬件计算性能。

NVIDIA GPU 上并未提供类似 CPU 上的向量单元，无法使用类似 CPU 的指令集进行向量运算。考虑到 NVIDIA GPU 上线程访存指令的位宽是 128 位，一次访存可以获取或写入 4 个单精度浮点数或 8 个半精度浮点数。因此，在一次取数计算或数据写回时，尽可能以 128 位为单位以减少访存指令执行的次数，以提高效率。

1.3 论文组织结构

本文基于国产 ARM 架构 CPU 处理器设计并实现了深度学习函数库 FTEngine，通过对多个主流卷积神经网络模型进行分析，本文在 FTEngine 深度学习函数库中使用 Intrinsic 技术以及内联汇编的方式实现了卷积神经网络的核心网络层。并通过使用体系结构优化的技术结合国产 ARM 架构 CPU 的体系结构对卷积层核函数和池化层核函数进行了深度优化。以下是本文的章节安排以及主要内容：

第一章 绪论。本章介绍了本文相关背景和研究意义。本章节介绍了人工智能库的相关研究、有关卷积优化的相关研究、体系结构相关研究以及数据并行技术的相关发展，以及本文的组织结构。

第二章 国产 ARM 架构 CPU 及卷积神经网络介绍分析。本章节介绍了国产

ARM 架构 CPU 的缓存大小、工作算力以及指令集的支持和寄存器的配置。并对几款主流的卷积神经网络进行简单的介绍，以及分析它们所支持的网络层。从主流卷积神经网络中提取了几种基本的网络层，并逐一介绍。

第三章 FTEngine 库函数设计实现。通过对上一章节的主流卷积神经网络分析，在这一章节我们介绍了我们实现的高性能卷积层函数。在这一章节我们介绍了我们在 FTEngine 中对池化层、归一化层、全连接层、激活层四个网络层的函数进行的优化和实现。

第四章 高性能卷积函数设计与实现。我们针对 FTEngine 中的卷积层采用了 Im2col+GEMM 的方式进行实现。我们结合 GOTOBLAS^[38]的分块思想和 BLIS^[39]的循环框架实现了高性能的矩阵乘函数，针对国产 ARM 架构的缓存结构设计了最适宜的分块大小，并使用可变长向量拓展指令集实现了高性能的矩阵乘汇编内核；我们结合高性能矩阵乘的实现对 Im2col 的算法逻辑进行分析并提出优化设计，设计并实现了 Im2col 在 NCHW 和 NHWC 两种维度上的实施策略，并实现了使用可变长向量拓展指令集的向量化可变长拷贝核函数。

第五章 实验与结果分析。本章节首先对本文实现的高性能矩阵乘函数进行了测试比较，然后与开源计算库 ACL 比较了深度优化的卷积层和池化层函数。最后同 ACL 做了多种并行模式下运行 AlexNet^[40]和 VGG16^[41]的性能对比。

第六章 总结与展望。本章节总结了本文中所做的所有工作，并提出了未来需要改进的方向和目标。

第2章 国产ARM架构CPU及卷积神经网络介绍分析

2.1 国产ARM架构CPU简介

国产ARM架构CPU是国防科技大学和飞腾公司合作研发的新一代处理器，使用ARMv8架构，支持可变长向量指令集，使用256位的向量长，具有很大的性能提升潜力。国产ARM架构CPU处理器使用64个兼容ARMv8指令集的处理器核心，每个处理器核心工作频率2.5GHz，并且拥有64KB的指令缓存和64KB的数据缓存，每两个处理器核心共享2MB大小的二级缓存，三级缓存总共64Mb。国产ARM架构CPU处理器不仅拥有更大的缓存数据，还支持更大的算力。在运算能力、存储能力上十分拥有十分强劲的实力，更加的匹配高性能计算和人工智能的需求。



图2.1 可变长向量寄存器及可变长预测寄存器示意图

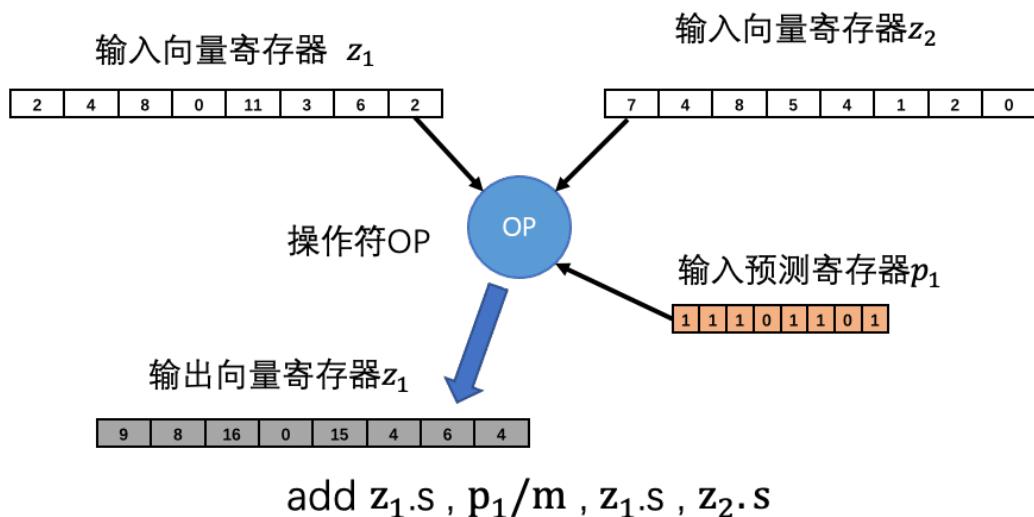


图2.2 向量预测求和指令示意图

国产ARM架构CPU拥有31个通用寄存器，它们在Aarch64位工作情况下用

X0-X31 标识，在当作 32 位的寄存器使用时使用 W0-W31 标识使用，通过寄存器的后缀代表具体使用类型。国产 ARM 架构 CPU 支持可变长向量拓展指令集，拥有 32 个支持 256 位向量长的矢量寄存器 Z0-Z31，支持 64 位、32 位、16 位和 8 位四种类型，低 128 位可以与 NEON 指令集共享使用，拥有 16 个可变长预测寄存器 P0-P15，每个可变长预测寄存器也同向量寄存器一样，可以存储与向量寄存器相同个数的数据，不过可变长预测寄存器大小为向量寄存器的 1/8 大小，使用最低位来标志是否使用位，预测寄存器在可变长向量指令集的指令中作为操作数而被使用，可以使用两种模式工作，Merge 模式在预测寄存器设置为 0 的位置将结果寄存器相应位置保留原来的第一个操作数的值，而 Zero 模式在预测寄存器设置为 0 的位置将结果寄存器相应位置设置为 0。我们用图2.2一个国产 ARM 架构 CPU 处理器中向量预测求和示意图来展示它的作用方式。

2.2 卷积神经网络层分析

目前由人们所熟知的典型卷积神经网络主要包括 LeNet^[42]、AlexNet、VGGnet、InceptionNet^[43]、ResNet^[44]、它们大部分因为 ImageNet 竞赛被人们所熟知。LeNet 在 1998 年由 Yann LeCnn 提出，使用 Sigmoid 作为主流的激活函数；AlexNet 在 2012 年夺得 ImageNet 竞赛冠军，使用了 Relu 激活函数、提出了 LRN^[45]（Local Response Normalization）局部响应标准化；VGG 通过使用小卷积核增加了特征图深度，提高了辨别能力；InceptionNet 增加了 Inception 结构块并通过使用小于输入特征图输入的 1x1 卷积核达成减少深度降低维度的目的，并提出 BN^[46]（BatchNormalization）实现特征标准化缓解梯度消失问题；ResNet 增加了纵向信息联系，引入了前方信息，缓解了模型退化问题。

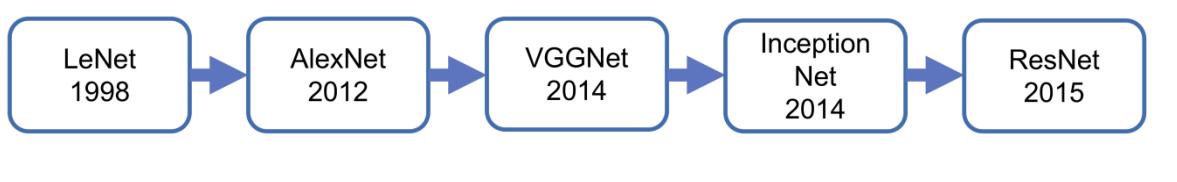


图 2.3 主流卷积神经网络历史发展示意图

综合观察上述多个网络模型结构，总结了它们所包含的网络结构如表2.1，根据图表中对比，可以发现几乎所有的主流 CNN 模型都支持卷积层、池化层、激活层、全连接层、归一化层。而 DepthWiseConcat 只有 IncepotionNet 拥有。

神经网络模型的计算包含了前向计算和反向计算^[47]两个过程，通常一个模型的推理过程只包含了前向计算过程，而训练过程则需要包含前向计算和反向计算，反向计算是根据反向传播算法提出的，针对不同的网络层，反向计算需要计算的对象和方式都有很大的不同。另外，不同的网络模型中对于不同网络层会选

表 2.1 知名卷积神经网络模型对于网络层的支持表

网络模型名称	卷积层 层	池化层	激活层	全连接 层	归一化 层	Inception 层
LeNet	支持	支持	支持	支持	—	—
AlexNet	支持	支持	支持	支持	支持	—
VGGNet	支持	支持	支持	支持	—	—
InceptionNet	支持	支持	支持	支持	支持	支持
ResNet	支持	支持	支持	支持	支持	—

取不同的具体实现，池化层通常提供了最大池化、平均池化、L2 平均池化等等、激活层又有提供 Relu、Softmax、Sigmoid 等等，最后 elementwise 函数并未在上表中提到，因为 elementwise 操作通常不会以网络层的形式提出，通常存在于网络层中间张量之间的计算，使用非常频繁。elementwise 可以细分为包含单个元素的 elementwise-unary，还有双元操作的 elementwise-binary 操作，它们也都包含了许多种变体。

2.3 卷积网络层介绍

在对几种常见卷积神经网络模型结构分析卷积层之后，我们在这小节对卷积层、池化层、全连接层、归一化层和激活层进行介绍，介绍每层特点以及功能作用。并在后续章节介绍在国产 ARM 架构 CPU 上针对不同的网络层函数进行特定的设计实现和优化。

2.3.1 全连接层

全连接层是神经网络中最基本的一种层次结构，它主要有两个作用。首先，全连接层能够从输入数据中提取和转换特征，得到更高层次的抽象特征表示，为后续的分类、回归等任务提供有用的信息。其次，全连接层的结构很简单，就是一个矩阵乘法加上一个偏置向量，即 $y = W \times x + b$ ，其中 W 是权重矩阵， x 是输入向量， b 是偏置向量， y 是输出向量。全连接层的参数个数取决于输入和输出的维度，通常比较多，因此容易造成过拟合。为了解决这个问题，可以使用正则化、dropout、批量归一化等技术来减少参数的自由度或增加泛化能力。

全连接层在深度学习中有很多应用。例如，在卷积神经网络中，通常在最后几层使用全连接层来进行分类；在自编码器中，可以使用全连接层来实现编码和解码；在生成对抗网络中，可以使用全连接层来构建生成器和判别器等。全连接

层的优点是可以实现任意的线性变换，从而提高神经网络的表达能力。全连接层也可以用于分类任务，例如在最后一层连接 Softmax 函数来输出类别概率。全连接层还可以用于回归任务，例如在最后一层连接恒等函数来输出连续值。然而，全连接层也有其缺点：参数个数较多，容易导致过拟合和计算开销大；不具有卷积层的局部感知和平移不变性，也就是对于输入的位置和结构不敏感，这对于处理图像等数据不利。因此，在深度学习中，通常会在全连接层之前使用卷积层或者池化层来提取局部特征和降低维度。

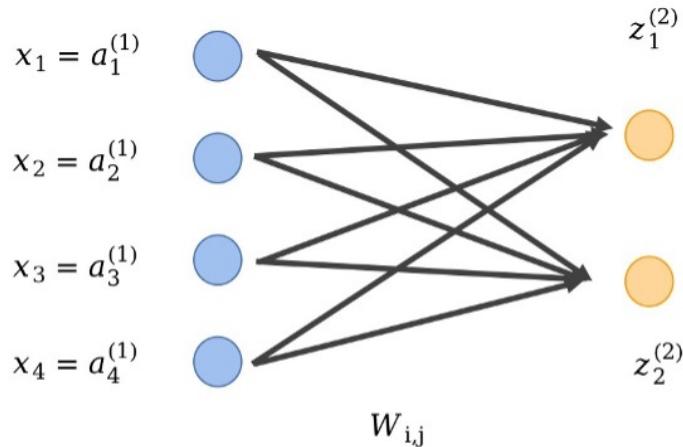


图 2.4 全连接操作示意图

$$\text{dst}(n, oc) = \text{bias}(oc) + \sum_{ic=0}^{IC-1} \text{src}(n, ic) \cdot \text{weights}(oc, ic) \quad (2.1)$$

如图2.4就是一个神经网络全连接层的示意图，全连接层的公式可以使用公式2.1表示，mb 和 ic 代表的是全连接层输入图片的数量和输入通道数，oc 代表的是输出的通道数目，weights 代表的是全连接层的权重，src 是输入层数据，bias 是这一层中的偏移值。全连接层输入 $mb \times ic$ 个数据，与大小为 $oc \times ic$ 的数据相乘得到大小为 $mb \times ic$ 的结果特征。

2.3.2 卷积层

全连接层一次性对所有数据进行处理，会导致数据维度降低以至于部分图像的重要特征丢失问题的出现。为了提取输入数据的局部特征，在动物视觉系统的启发下发明了卷积层，神经网络卷积层具有平移、旋转和缩放不变性等优点，能够有效提取图像中的局部特征。

卷积是深度学习和卷积神经网络中运算占用时间运算量最大，运算最频繁的部分，所以卷积神经网络是最需要进行优化计算的部分。卷积层的基本原理是通过将小矩阵（称为卷积核或滤波器）在输入图像上滑动并点乘每个位置的值，得到输出的特征图，卷积的计算可以用公式2.2表示。其中 oc 表示输出维度，ic 表示

输入维度， KW 表示卷积窗口的高度， KW 表示卷积窗口的宽度， PH 和 PW 分别表示填充的高度和宽度， SH 和 SW 分别表示卷积在高度维度上和宽度维度上的窗口跨度。

$$\begin{aligned} \text{dst}(n, oc, oh, ow) = & \text{bias}(oc) + \\ & \sum_{ic=0}^{IC-1} \sum_{kh=0}^{KH-1} \sum_{kw=0}^{KW-1} \text{src}(n, ic, oh \cdot SH + kh - PH_L, ow \cdot SW + kw - PW_L) \quad (2.2) \\ & \cdot \text{weights}(oc, ic, kh, kw) \end{aligned}$$

在卷积的过程中，卷积层的输入参数需要指定图片数量、输入通道、输入图尺寸、卷积核尺寸、输出通道、卷积的跨度、卷积边缘填充值、卷积空洞值。图片数目和输入通道特征图决定了输入特征图的数量规模，输出通道和输入通道来决定卷积核的数量规模，而通过输入图尺寸、卷积核尺寸和卷积跨度、边缘填充值、卷积空洞值可以决定卷积的执行方式，同时也可以决定输出图的大小。并且，卷积在高级神经网络模型中还可以与其他类型的层结合使用，具有良好的融合性质。输出图的大小可以通过公式2.3决定。

$$\begin{aligned} H_{\text{output}} &= (H_{\text{input}} - H_{\text{kernel}} + \text{padding} \times 2 - (\text{dilation} \times (H_{\text{kernel}} - 1) + 1)) \div \text{stride} + 1 \quad (2.3) \\ W_{\text{output}} &= (W_{\text{input}} - W_{\text{kernel}} + \text{padding} \times 2 - (\text{dilation} \times (W_{\text{kernel}} - 1) + 1)) \div \text{stride} + 1 \end{aligned}$$

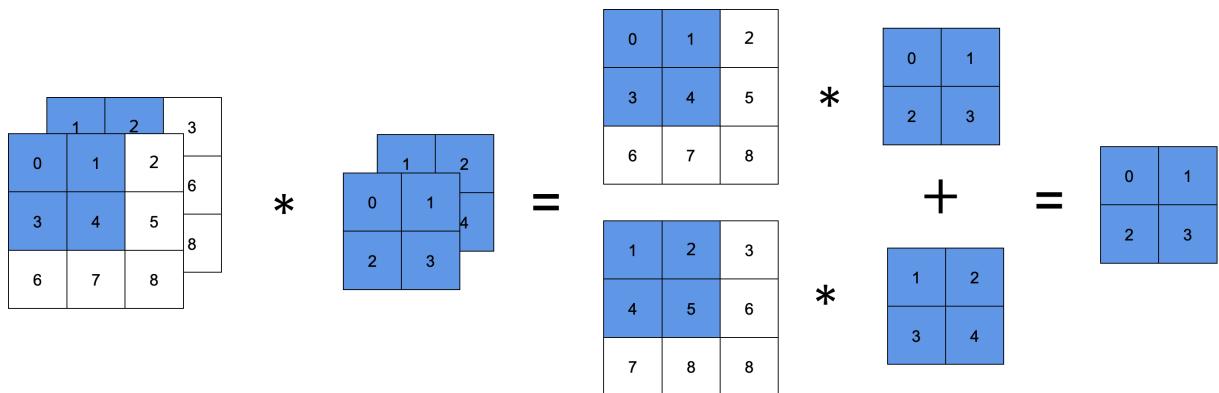


图 2.5 卷积操作示意图

如图2.5就是一个多通道的卷积计算示例，图中特征图规模 $\text{nbatches} \times W \times \text{channels} \times \text{height} \times \text{width} = 1 \times 2 \times 3 \times 3$ ，其中输入图数量为 1 输入通道为 2，高和宽分别为 3，卷积核的输入规模 $\text{channels}_{\text{out}} \times \text{channels}_{\text{in}} \times \text{Kernel}_{\text{height}} \times \text{Kernel}_{\text{width}} = 1 \times 2 \times 2 \times 2$ ，其中输出通道为 1，输入通道为 2，卷积核的高和宽都为 2，另外图中的卷积的空洞 dilation 为 1，填充 padding 为 0，步长 stride 为 1，分别代表图中卷积不是空洞卷积，每次卷积核移动一个像素点以及没有边缘填充。

2.3.3 池化层

池化层是神经网络中一种用于在卷积层之后进行数据降维的操作，也称为下采样层。它能够减少网络的参数量，防止过拟合，增强模型的鲁棒性和不变性。由于在神经网络模型中，特征之间的精确位置远不如它们与其他特征提取块中的相对位置信息重要，因此常常处于一种参数冗余状态。池化可以通过在一个特征区域中，使用特定的池化算法取得一个特征值来代表整个特征区域的特征值，既提取了代表性特征值又缩小了参数量。池化层的主要作用包括增大网络感受野、抑制噪声、降低信息冗余、降低模型维度和计算量、降低网络优化难度，使模型对输入图像中的特征位置变化更具鲁棒性。

常见的池化方法有最大池化、平均池化、全局平均池化、混合池化、随机池化和中值池化等。不同的池化方法各有优缺点。最大池化和平均池化是比较常见的两种方法。最大池化能够提取特征图中响应最强烈的部分，将池化区域的最大值提取出来作为池化层的输出结果。它之所以能够成为最有效的池化算法之一，是因为在卷积层提取了特征之后，某个区域的特征往往可以由最大特征提取值来表示。平均池化则通过将池化区域中所有值求和取得特征平均来描述池化区域的特征。但是最大池化虽然保留了图像的边缘和纹理信息，但也可能丢失一些细节信息。而平均池化可以提取特征图中所有特征的信息，保留图像的背景信息，但也可能导致特征混杂和模糊。

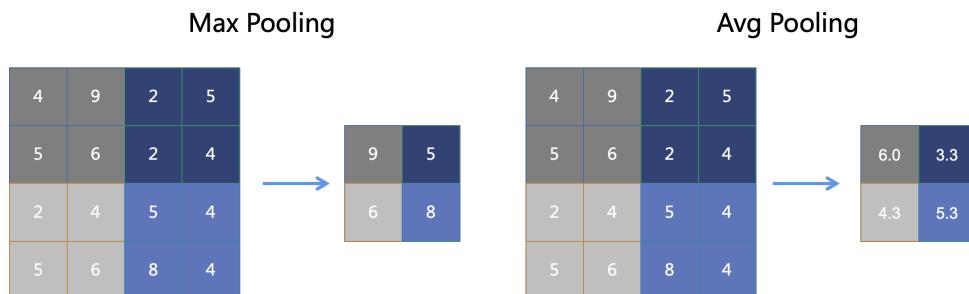


图 2.6 池化操作示意图

图2.6示例展示了一个简单的池化操作，图左边表示了最大池化的计算过程，图右边展示了平均池化的计算过程

对于最大池化可以参照公式2.4它将同一个池化窗口内相同通道数中最大的特征像素点提取出来输出到输出特征图中。

$$\begin{aligned}
 \text{dst}(n, c, oh, ow) = & \\
 \max_{kh, kw} & (\text{src}(n, c, oh \cdot SH + kh \cdot (DH + 1) - PH_L, ow \cdot SW + kw \cdot (DW + 1) \\
 & - PW_L))
 \end{aligned} \tag{2.4}$$

平均池化可以参照公式2.5，平均池化将同一个池化窗口中相同通道数的特征值求和并求平均再输出到特征图中。

$$\begin{aligned} \text{dst}(n, c, oh, ow) = & \\ & \frac{1}{KH \cdot KW} \sum_{kh, kw} \text{src}(n, c, oh \cdot SH + kh \cdot (DH + 1) - PH_L, ow \cdot SW + kw \cdot (DW + 1) - PW) \end{aligned} \quad (2.5)$$

卷积层的输出尺寸为：

$$\begin{aligned} H_{\text{output}} &= (H_{\text{input}} - H_{\text{kernel}} + \text{padding} \times 2) \div \text{stride} + 1 \\ W_{\text{output}} &= (W_{\text{input}} - W_{\text{kernel}} + \text{padding} \times 2) \div \text{stride} + 1 \end{aligned} \quad (2.6)$$

其中 n 代表了输入图的数量，c 代表着输入图的通道数量，SH 和 SW 代表着在高和宽两个维度上的跨度大小，PH 和 PW 代表着高和宽两个方向上边缘的填充大小，KW 和 KH 代表着池化窗口的大小，oh 和 ow 表示这输出图像的高和宽的大小。对于最大池化来说是将池化窗口内的所有元素进行比较返回最大的池化，而对于平均池化则是将池化窗口中的元素求和取平均。

2.3.4 归一化层

归一化是一种常用的数据预处理技术，它通过将不同规格或分布的数据转换为统一的规格或分布，来消除数据之间的量纲或尺度差异，从而提高数据的可比性和稳定性。在卷积神经网络中，归一化可以应用于隐藏层的输入，以便更容易地训练网络。归一化能够加速梯度下降寻找最优解的速度，即加快模型的收敛速度。如果数据取值范围相差较大，目标函数就会变得扁平，梯度下降就需要走很多弯路，导致迭代次数增加。但是，如果对数据进行归一化处理，目标函数就会呈现比较圆形，这样梯度下降就能更快地找到最优解。此外，归一化还可能提高模型的精度。

在分类器或聚类算法中，需要计算样本之间的距离。如果一个特征的取值范围很大，那么距离计算就会主要取决于这个特征，而忽略其他特征的贡献。通过归一化处理，可以使不同特征对距离的影响更加均衡，从而提高模型的泛化能力。

此外，在深度学习中，归一化层还可以适应激活函数。如果使用 Sigmoid 或 Tanh 等激活函数，则需要将数据归一化到接近 0 的区域，以避免数据落入饱和区导致梯度消失。如果使用 Relu 激活函数，则需要将数据归一化到正值区域，以避免数据死亡。在神经网络中常见的归一化层包括批量归一化 (Batch Normalization, BN)、层归一化 (Layer Normalization, LN)、组归一化 (Group Normalization, GN) 和单

例归一化 (Instance Normalization, IN)。

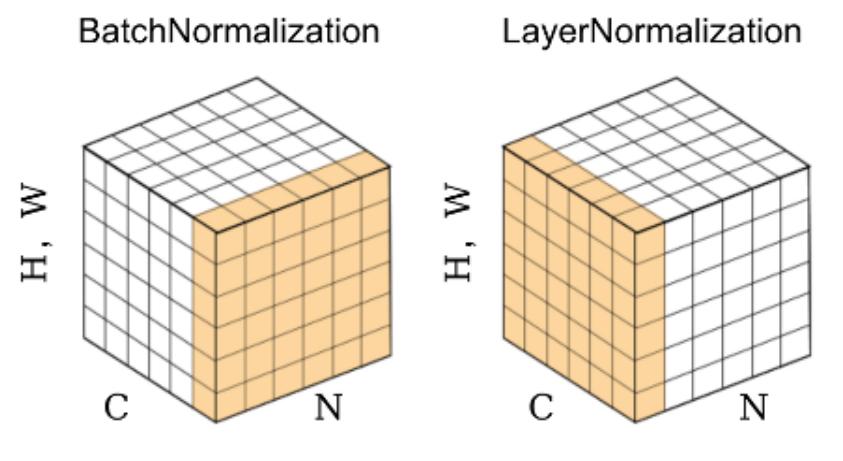


图 2.7 批归一化和层归一化计算作用维度示意图

批量归一化是为了解决神经网络模型训练时，不同输入之间输入特征分布差异太大导致内部协变量偏移问题出现而提出的。所以批量归一化对每个小批量样本集合中每个通道上所有样本做归一化，并引入可学习的缩放和平移参数，图2.7中左侧就是批量归一化的示意。

$$\text{dst}(n, c, h, w) = \gamma(c) \cdot \frac{\text{src}(n, c, h, w) - \mu(c)}{\sqrt{\sigma^2(c) + \epsilon}} + \beta(c) \quad (2.7)$$

其中：

$$\begin{aligned} \mu(c) &= \frac{1}{NHW} \sum_{nhw} \text{src}(n, c, h, w) \\ \sigma^2(c) &= \frac{1}{NHW} \sum_{nhw} (\text{src}(n, c, h, w) - \mu(c))^2 \end{aligned} \quad (2.8)$$

批量归一化的计算方式可以参考公式2.7, x_i 是归一化层的输入数据是某个通道中的所有的数据的规模， μ 表示所有输入数据的平均值， σ^2 表示的是输入数据在通道上的方差，由于方差可能会出现 0 的情况，所以需要 ϵ 防止分母为 0 的出现，最后用归一化的数据乘以缩放变量 γ 再加上偏移变量 β 。

层归一化对某一批次所有的神经元进行归一化，批量归一化需要应对批量不小的情况，而且当面对输入的分布动态变化的情况时候，层归一化是一个解决方案，层归一化不依赖于小批次的统计信息，因此在使用小批次时也能保持良好的性能。层归一化的计算方式可以参考公式2.9，公式参数同批量归一化，图2.7右侧就是层归一化的展示。

$$\text{dst}(t, n, c) = \gamma(c) \cdot \frac{\text{src}(t, n, c) - \mu(t, n)}{\sqrt{\sigma^2(t, n) + \epsilon}} + \beta(c) \quad (2.9)$$

其中：

$$\begin{aligned}\mu(t, n) &= \frac{1}{C} \sum_c \text{src}(t, n, c) \\ \sigma^2(t, n) &= \frac{1}{C} \sum_c (\text{src}(t, n, c) - \mu(t, n))^2\end{aligned}\quad (2.10)$$

2.3.5 激活层

深度学习激活层是指在神经网络中，对输入信号进行非线性变换的函数。激活层的作用是增加神经网络的表达能力，使其能够拟合复杂的函数。激活层的选择会影响神经网络的性能和收敛速度，因此需要根据不同的任务和数据选择合适的激活层。

常见的激活层有以下几种：

Sigmoid 激活层：Sigmoid 函数是一种 S 形曲线，其输出范围为 (0,1)，可以用于表示概率或者二分类问题。Sigmoid 激活层的优点是输出值有界，可以抑制过大或过小的信号；缺点是容易出现梯度消失，计算量较大，且不是零中心的。Sigmoid 函数公式同2.11。

$$d = \frac{1}{1 + e^{-s}} \quad (2.11)$$

Tanh 激活层：Tanh 函数是一种双曲正切函数，其输出范围为 (-1,1)，可以用于表示正负或者多分类问题。Tanh 激活层的优点是输出值为零中心，可以减少参数偏移；缺点是当值偏离零点时倒数将缩小接近 0，仍然容易出现梯度消失，计算量较大。Tanh 函数公式同2.12。

$$d = \tanh s \quad (2.12)$$

Relu 激活层：Relu 函数是一种分段线性函数，其输出范围为 [0, +∞]，可以用于表示非负或者稀疏特征。Relu 激活层的优点是计算量小，在零值附近的斜率较高，收敛速度快，可以缓解梯度消失；缺点是输出值无界，可能导致神经元死亡。Relu 函数公式同2.13。

$$d = \begin{cases} s & \text{if } s > 0 \\ 0 & \text{if } s \leq 0 \end{cases} \quad (2.13)$$

Leaky Relu 激活层：Leaky Relu 函数是对 Relu 函数的改进，其在负半轴上有

一个小的斜率 α , 可以用于表示非负或者稀疏特征。Leaky Relu 激活层的优点是避免了神经元的死亡归零情况, 保持了 Relu 的优势; 缺点是输出值无界, 可能导致数值不稳定。Leaky Relu 函数公式同2.14。

$$d = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{if } s \leq 0 \end{cases} \quad (2.14)$$

Softmax 激活层: Softmax 函数是一种归一化指数函数, 其输出范围为 $(0,1)$, 且所有输出值之和为 1, 用于表示多分类问题或者多项式分布。Softmax 激活层的优点是能够输出类别概率, 便于计算交叉熵损失; 缺点是输出值受最大值影响较大, 可能导致数值溢出或者梯度稀疏。

Softmax 激活层分为两种方案, 一种是不用对结果取对数的 Softmax, 同公式2.15:

$$\text{dst}(\overline{ou}, c, \overline{in}) = \frac{e^{\text{src}(\overline{ou}, c, \overline{in}) - v(\overline{ou}, \overline{in})}}{\sum_{ic} e^{\text{src}(\overline{ou}, ic, \overline{in}) - v(\overline{ou}, \overline{in})}} \quad (2.15)$$

然而在减去最大值之后 Softmax 仍会有出现上下溢出的风险, 于是使用 LogSoftmax 对 Softmax 的结果进行取对数处理, 增强了数据的稳定性。LogSoftmax 的公式同2.16:

$$\begin{aligned} \text{dst}(\overline{ou}, c, \overline{in}) &= \ln \left(\frac{e^{\text{src}(\overline{ou}, c, \overline{in}) - v(\overline{ou}, \overline{in})}}{\sum_{ic} e^{\text{src}(\overline{ou}, ic, \overline{in}) - v(\overline{ou}, \overline{in})}} \right) \\ &= (\text{src}(\overline{ou}, c, \overline{in}) - v(\overline{ou}, \overline{in})) - \ln \left(\sum_{ic} e^{\text{src}(\overline{ou}, ic, \overline{in}) - v(\overline{ou}, \overline{in})} \right) \end{aligned} \quad (2.16)$$

这里 \overline{out} 表示的外部的维度, \overline{in} 表示的是最内部的维度, $v(\overline{ou}, \overline{in}) = \max_{ic} \text{src}(\overline{ou}, ic, \overline{in})$ 表示的是在某个通道数下的最大值。

2.4 本章小节

本章简单描述了国产 ARM 架构 CPU 的基本架构, 然后分析了几种知名卷积神经网络所包含的网络层, 并对所共有的网络层进行了分析介绍。后续将介绍我们在国产 ARM 架构 CPU 上实现的深度学习函数库, 并介绍我们针对本章所介绍的网络层所实现的高性能算子函数。

第3章 FTEngine 库函数设计实现

通过上一章节对国产 ARM 架构 CPU 的了解和对卷积神经网络层的分析了解周，面对市面上暂时不存在针对国产 ARM 架构 CPU 优化实现的人工智能函数库的问题，我们实现了针对国产 ARM 架构 CPU 的体系结构实现了高性能深度学习单精度推理函数库——FTEngine，旨在充分提高国产 ARM 架构 CPU 在深度学习中的计算表现能力；本章节将先介绍 FTEngine 位于 AI 软件栈的位置和作用，然后讲解 FTEngine 中各个算子的实现，对于性能需求最大的卷积函数，本文将它放到下一章节单独讲解。

3.1 FTEngine 总体设计

在人工智能领域中，AI 程序软件栈通常可以自上而下分为软件层、支持层、硬件层三层。通常所说的 AI 软件栈指的软件层和支撑层的组合，算法人员将程序运行在 AI 软件上，这一层的软件涵盖了训练、推理、部署三种 AI 软件框架，包含了神经网络操作接口、基本数据结构定义、设备管理、内存管理、数据加载处理等功能模块，比较庞大并且功能丰富。软件层需要由支持组提供函数功能支持，支持层主要是由针对某款体系结构设计实现的函数构成，本文实现的 FTEngine 处于支持层，利用可变长向量扩展指令集结合通用体系结构优化技术设计、实现高性能深度学习网络层函数，以支持在国产 ARM 架构 CPU 平台上支持运行卷积神经网络模型的函数实现。

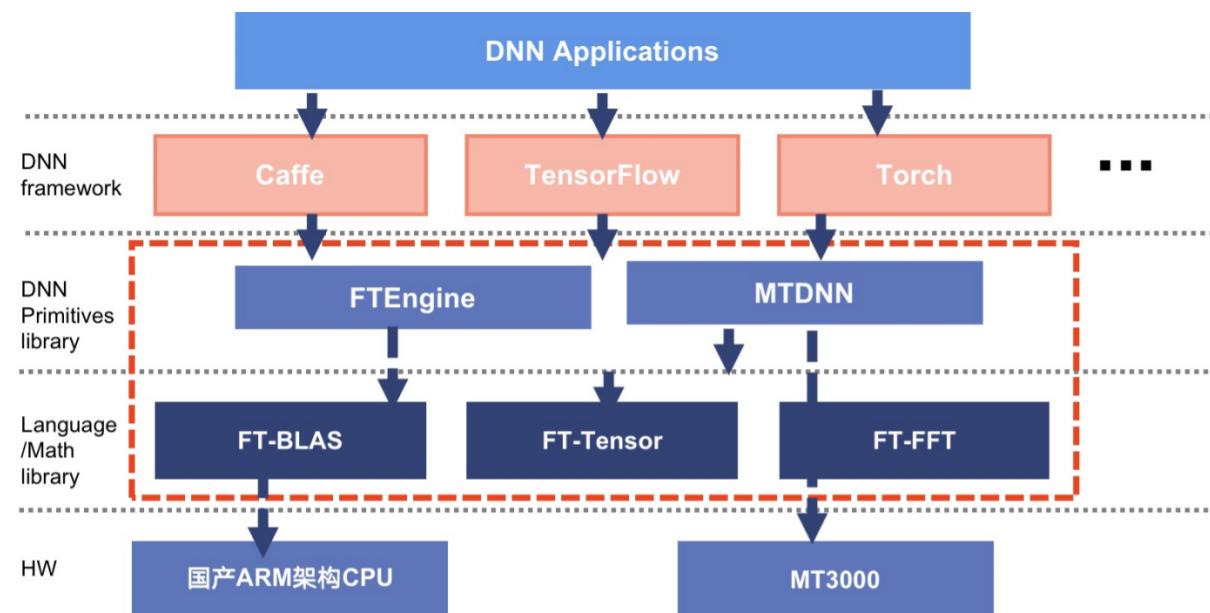


图 3.1 人工智能函数库架构和 FTEngine 的支持层部分

3.2 FTEngine 算子实现

3.2.1 池化层函数实现

(1) 池化函数结构设计

针对池化层的运算特征，我们将池化函数的调用抽象成“接口层—驱动层—内核层”三层结构。在接口层，我们根据池化方法的选择（最大池化或平均池化）和不同的数据格式输入进行分层处理。对于 NCHW 格式的输入，我们会在接口层将其转换为 NHWC 格式。而对于 NHWC 格式的输入，我们会在驱动层使用循环和多线程控制，在内核层分别实现平均池化的求均值算子和最大池化的求最大值算子。

具体来说，在接口层，我们接受池化层的全部参数，并根据输入类型使用函数指针成员变量保存后续将要使用的内核接口。针对输入数据格式，我们可以在外部选择是否需要进行格式转换。如果需要进行格式转换，我们会针对 NCHW 格式的输入数据，在池化之前进行数据格式转换。而对于需要从 NHWC 转换为 NCHW 格式的数据，我们会选择在池化之后进行转换。

内核层包含平均池化和最大池化的两种汇编内核实现。每个内核函数都计算一个池化窗口区域，最终得到一个池化窗口在多个通道上的输出值。因此，作为内核层，参数包括一个浮点二维数组和两个整型值作为输入，一个浮点一维数组作为输出。两个整型值分别表示通道数和数据单元数。这里我们用数据单元表示数据，因为一个数据单元包含了多个通道的数据，每个数据单元实际上都是一个一维数组。作为输入的浮点二维数组存储了所有数据单元的地址，其大小为池化窗口中数据单元的个数。而输出的一维数组代表了最后输出的数据单元。在内核层中，我们主要使用了体系结构优化和向量化技术，在最大限度地加速池化操作。

进入驱动层后，已经确定了汇编内核函数的选用。这一层需要处理池化操作时滑动窗口所需解决的问题，并处理填充、跨度等参数对不同位置池化窗口最终调用汇编内核的影响。驱动层中是以池化输出值的行和列为基础进行循环的。对于池化函数的输出值，一个输出值就可以代表一个池化窗口。通过输出值坐标可以反向推导出窗口涵盖的特征图范围和填充范围。通过填充范围和窗口位置可以确定窗口内有效数据元个数。由于池化窗口内各个数据元存在跨行跨列情况，在驱动层调用汇编层前需要准备保存数据元地址数组。

内核层是包含平均池化和最大池化的两种汇编内核实现，每一个内核函数计算的是一个池化的窗口区域，最终得到一个池化窗口在多个通道上的输出值，所以作为内核层，参数包含了一个浮点二维数组和两个整形值作为输入，一个浮点一维数组作为输出，两个整型值分别是通道数和数据单元数，这里我们用数据单

元表达数据，因为一个数据单元包含了多个通道的数据，每个数据单元实际上都是一个一维数组，而作为输入的浮点二维数组存储的就是所有数据单元的地址，他的大小是池化窗口中数据单元的个数，而输出的一维数组代表的就是最后输出的数据单元。在内核层中我们主要使用了体系结构的优化和向量化技术，在最大限度的加速池化的操作。

进入驱动层已经确定了汇编内核函数的选用，这一层需要处理池化操作时候池化窗口的滑动所需要解决的问题，需要处理填充、跨度等参数对于不同位置的池化窗口最终调用汇编内核的影响。驱动层中是以池化输出值的行和列为基础进行循环的，对于池化函数的输出值，一个输出值就可以代表着一个池化窗口，通过输出值的坐标可以进而反向推导出窗口涵盖的特征图范围和填充范围。通过填充范围和窗口位置可以确定窗口内的有效数据元个数，因为池化的窗口内的各个数据元存在跨行跨列的情况，所以在驱动层调用汇编层的最后一步需要做好准备保存数据元地址数组的工作。

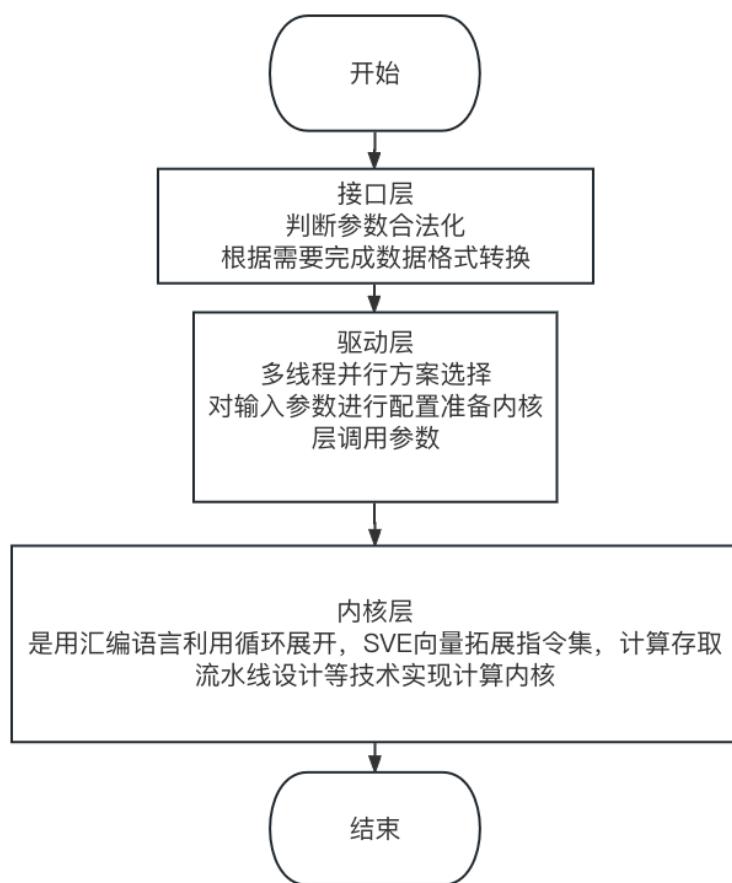


图 3.2 池化函数的分层实现抽象图

图3.2就是我们在 FTEngine 中对池化函数进行的分层实现，使用三层的实现我们大量增加了代码的重用度并且提供了非常高的拓展性，同时也保证性能的高效。

(2) 向量优化 NHWC 格式池化内核

对于输入数据格式为 NCHW 的情况不采用向量化进行内核优化，因为 NCHW 的数据格式中只有卷积窗口个数数据，并且所有数据需要相互计算，那么对于 NCHW 的数据格式的池化操作即使向量化也无法避免最终求池化结果时候需要进行性能较低的累加操作，所以，我们只对 NHWC 的数据格式实现向量化优化内核。

对于池化层函数我们采用内联汇编的形式实现汇编内核，在汇编层函数中可以利用预测寄存器控制循环操作，使用预测寄存器控制循环结束，具体实施时候，我们将使用 4 个预测寄存器 P0-P3 来控制计算数据长度，使用 16 个向量寄存器 Z0-Z15 来存储计算数据，因为数据格式是 NHWC 形式的，所以存储的以通道优先的顺序，优化实现中，一次性可以计算最多 4 个向量长的通道数据，向量与向量之间计算时候（最大池化求最大的方式，平均池化求和），计算会存在于相同的通道数据之中，在向量中刚好对应相同的数据索引。图3.3和图3.4假设我们的寄存器存储 10 个单精度型情况时候的中计算方式：

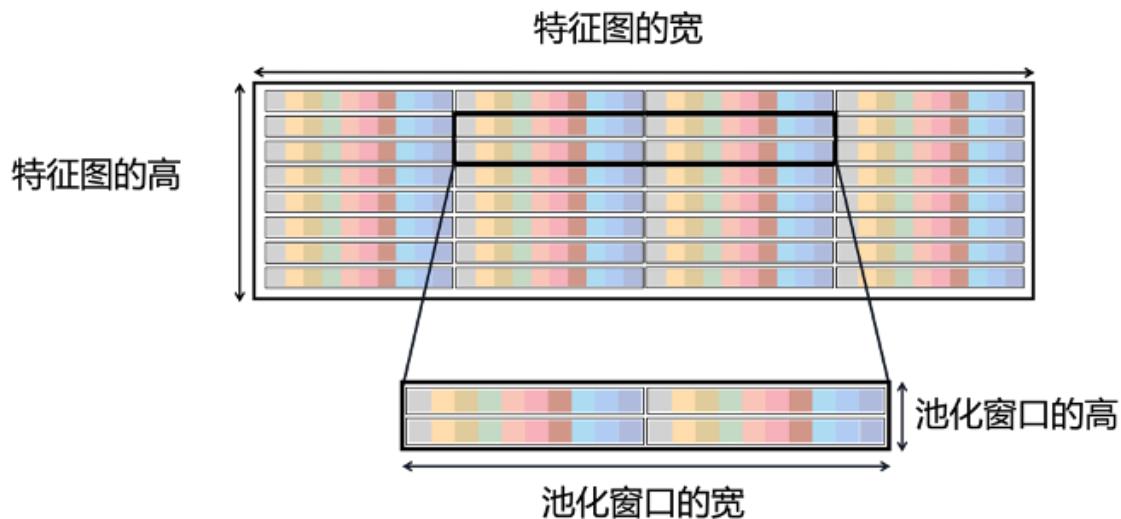


图 3.3 池化函数的分层实现抽象图

具体的我们通过内联汇编的形式实现了内核算子，通过汇编指令 `whilelt` 指令来控制通道数目上的迭代，通过 `ld1w` 指令来将数据载入到向量寄存器中，通过使用 `fmax` 获得两向量寄存器中的数据的较大值，在平均 pooling 中，通过使用 `fadd` 累加向量值并在累加结束之后通过使用 `fmul` 指令乘以统计数量的倒数得到平均值，如下展示部分主要计算部分，以及存算流水设计。

在汇编内核中使用 `movprfx` 指令掩盖两个指令额外延迟，使用 `movprfx` 指令可以将一个运算寄存器做到类似于更名的操作，`movprfx` 指令可以与后序一个指令相结合将一个三元操作汇编指令转换为一个四元操作汇编指。比如：当我想完成一个形似 $z \leftarrow z3 + z1 \times z2$ （包含四个操作数的乘加指令，FMA4）的操作时候，

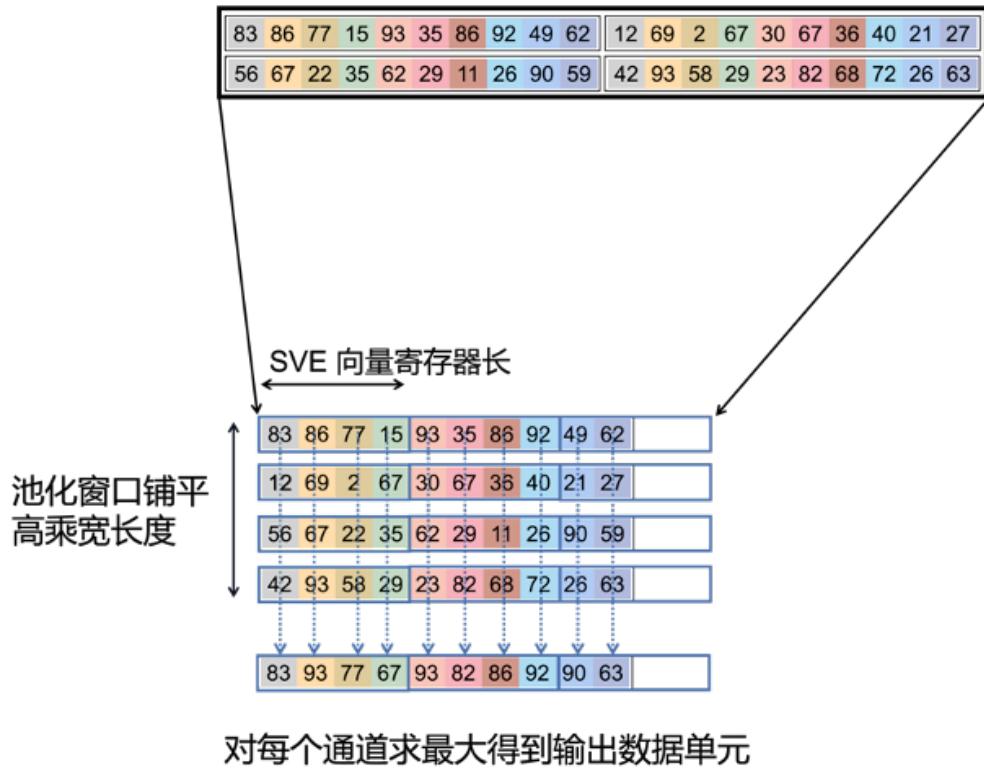


图 3.4 池化函数的分层实现抽象图

在 ARM 可变长向量拓展指令集的汇编指令中需要将它转换成为两个指令来完成操作: $z3 \leftarrow z3 + z1 \times z2$ 和 $z0 \leftarrow z3$, 但是通过使用 movprfx 指令, 这个操作可以达到一步完成。

在汇编内核中使用 whilelt 指令控制循环逻辑, 可变长向量拓展指令集中新增的 whilelt 指令可以在控制向量寄存器的存取过程中搭可变长预测寄存器使用, 可以在我们进行计算的过程中避开繁杂的分支预测。预测寄存器是与向量寄存器进行通道对应的, 在不同的架构底下即使向量寄存器是不同位长的, 而预测寄存器可以使用有效位对应向量寄存器的元素。在实际的汇编内核代码中, 本函数适用的通道优先存储情况中, 用向量寄存器来存储一个像素点位置不同通道的数据, 在存储的过程中就先用连续四次的 whilelt 指令对通道长度进行判断设定相应的预测寄存器。需要注意的是, 这样可以让我们无需处理 SIMD 代码中常见的尾余值。whilelt 指令可以更改状态寄存器, 所以可以通过对于状态寄存器的判断来判断是否终止循环, 而对于状态寄存器的判断要比每次比对要快 $n * x$ 倍, n 为每次使用多少个向量寄存器进行读数, x 为向量寄存器的长度。因此使用 whilelt 指令可以节约大量的分支预测次数。

同样的这个实现是同时支持半精度的计算情况, 在半精度的情况下仅仅需要将所有操作指明单精度的部分改成为支持半精度的方式, 比如 ld1w 载入数据的指令, w 意味着一个字的大小 (在向量寄存器的计量单位中, 1 个字代表的是 32

```

1      "2:"  

2      "movprfx z19, z4\n fmax z19.s, p0/M, z19.s, z3.s\n"  

3      "movprfx z23, z2\n fmax z23.s, p0/M, z23.s, z1.s\n"  

4      "ldp x23, x22, [x19, #0x0]\n"  

5      "ldp x21, x20, [x19, #0x10]\n"  

6      "movprfx z18, z0\n fmax z18.s, p0/M, z18.s, z31.s\n"  

7      "fmax z22.s, p0/M, z22.s, z30.s\n"  

8      "ldlw { z4.s }, p4/Z, [x23, x28, LSL #2]\n"  

9      "ldlw { z3.s }, p4/Z, [x22, x28, LSL #2]\n"  

10     "movprfx z17, z29\n fmax z17.s, p0/M, z17.s, z28.s\n"  

11     "fmax z21.s, p0/M, z21.s, z27.s\n"  

12     "ldlw { z2.s }, p4/Z, [x21, x28, LSL #2]\n"  

13     "ldlw { z1.s }, p4/Z, [x20, x28, LSL #2]\n"  

14     "movprfx z16, z26\n fmax z16.s, p0/M, z16.s, z25.s\n"  

15     "fmax z20.s, p0/M, z20.s, z24.s\n"  

16     "ldlw { z0.s }, p3/Z, [x23, x27, LSL #2]\n"  

17     "ldlw { z31.s }, p3/Z, [x22, x27, LSL #2]\n"  

18     "fmax z19.s, p0/M, z19.s, z23.s\n"  

19     "fmax z18.s, p0/M, z18.s, z22.s\n"  

20     "ldlw { z22.s }, p3/Z, [x21, x27, LSL #2]\n"  

21     "ldlw { z30.s }, p3/Z, [x20, x27, LSL #2]\n"  

22     "fmax z17.s, p0/M, z17.s, z21.s\n"  

23     "fmax z16.s, p0/M, z16.s, z20.s\n"  

24     "ldlw { z29.s }, p2/Z, [x23, x26, LSL #2]\n"  

25     "ldlw { z28.s }, p2/Z, [x22, x26, LSL #2]\n"  

26     "subs x24, x24, #0x1\n"  

27     "fmax z8.s, p0/M, z8.s, z19.s\n"  

28     "ldlw { z21.s }, p2/Z, [x21, x26, LSL #2]\n"  

29     "ldlw { z27.s }, p2/Z, [x20, x26, LSL #2]\n"  

30     "fmax z7.s, p0/M, z7.s, z18.s\n"  

31     "fmax z6.s, p0/M, z6.s, z17.s\n"  

32     "ldlw { z26.s }, p1/Z, [x23, x25, LSL #2]\n"  

33     "ldlw { z25.s }, p1/Z, [x22, x25, LSL #2]\n"  

34     "fmax z5.s, p0/M, z5.s, z16.s\n"  

35     "add x19, x19, #0x20\n"  

36     "ldlw { z20.s }, p1/Z, [x21, x25, LSL #2]\n"  

37     "ldlw { z24.s }, p1/Z, [x20, x25, LSL #2]\n"  

38     "bgt 2b\n"

```

图 3.5 最大池化汇编内核核心段汇编代码

位数据)在半精度中改为 ld1h 指令, h 意味着半个字的大小(也就是 16 位数据大小);而在使用向量寄存器的时候,将向量寄存器的后缀精度标识符号改变即可,最大化的循环体内核计算代码如图3.5所示。

3.2.2 归一化层函数实现

我们目前仅在 FTEngine 中实现了高性能的批归一化函数,批归一化函数的实现实际上是针对每个通道的内所有的数据,对同一个通道内的所有数据进行归一化处理,主要的计算部分都是围绕通道维度进行展开,所以归一化层算法的实现方法在 NHWC 数据格式和 NCHW 数据格式中也有实现上的不同。另外,批量归一化函数在网络模型的执行期间,有可能均值和方差在传入时候便已计算好,也有可能会以传入空指针的方式指导批处理函数内部需要计算均值和方差,所以同时也需要实现对均值和方差的求解函数。

针对 NHWC 格式的输入数据,无论是计算均值和方差,还是计算归一化数据都比较方便,这个方便体现在 NHWC 数据格式在批归一化的函数中的计算对于并行性的支持,批归一化的实现是针对数据通道的维度展开的,需要对个数据通道的所有数据展开计算,而不同通道的数据之间连续存储,相邻数据之间不需要进行计算,这种结构很适合使用向量处理器进行计算,没有计算效率较低的累算操作需求。目前我们在 FTEngine 中实现了 NHWC 数据格式的高性能批归一化函数以及 NHWC 格式下的高性能求均值方差函数。我们将顺序讲解计算我们如何实现均值和方差以及计算归一化值:

(1) 计算均值和方差

在计算均值和方差的函数中,我们将 NHW 三个维度进行展开合并并称之为有效计算单元数,代表每一个数据通道都需要针对有效单元数据数目进行计算,然后我们可以调用平均池化的汇编内核计算平均值,在平均池化的内核中,我们在数据通道上用 4 个向量寄存器进行循环计算,而在有效单元数上面进行大循环。针对我们一个向量寄存器一次性可以存储 256 位长的数据。我们一次可以处理 32 个通道。在求均值时,我们把步骤分解为累加,求平均,对于对有效单元数据数量个向量组进行累加的过程中,因为国产 ARM 架构 CPU 拥有两个浮点 SIMD 操作元件,同一时间可以指令两个浮点乘加运算,所以在计算累加操作时候,我们对累加的方法进行了修改,我们采用每次累加四组元素的方式进行累加数据,分为三个步骤实现,第一步先将四组数据分成两份两两相加,然后第二步将两两相加得到的两组结果数据相加得到四组数据的总和,第三步再累加到全局累加数据。因为国产 ARM 架构 CPU 同一时间可以完成两次浮点加法操作,在这种计算模式下,第一步中四组数据的两组加法中不存在数据冲突,读写寄存器并不重复,

所以第一步中的两个加法操作可以在一个周期中完成，而第二部和第三部各占一个周期，总共就是四个周期可以完成 4 个数据的累加操作。这对比朴素的四组数据的逐次更新具有巨大的性能提升。累加策略如图3.6所示，可以发现不考虑数据迁移的耗时情况下，可以在三个周期的 CPU 运行时间完成四个周期的朴素累加。

在求得了平均值之后，下一步就是使用平均值对于每个输入数据进行求解，使用可变长向量拓展指令集向量化实现计算，外部循环通道方向，在内部循环使用向量求解数据元素与平均值的差平方，并累加到全局累加向量寄存器中，每次内部循环结束就可以获得一组通道数量的方差结果。同样的，对该计算流程使用相同优化手段，虽然对于树状累加方法的加速将减少的加速表现，但是因为相对累加操作需要更多的计算，所以获得的并行状态加速会有更多。

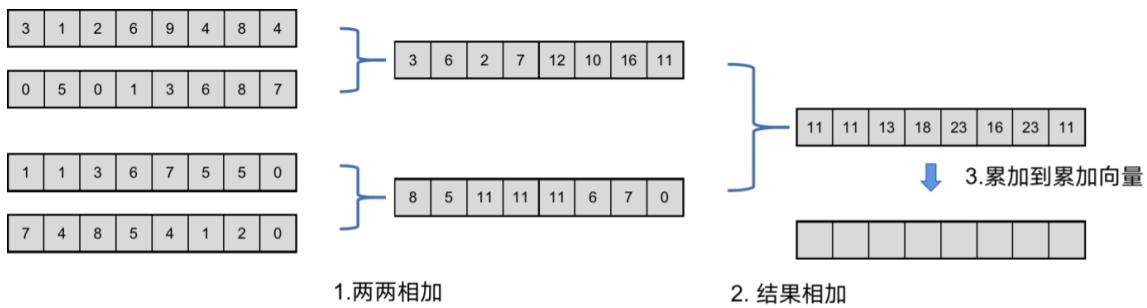


图 3.6 累加方法示意图

对于计算均值和方差这类程序，因为在同一个通道的数据之间存在累算操作，会在最终对于统计的累算变量共同访问，满足产生数据竞争的条件，所以我们选择在数据通道维度进行多线程并行优化，将数据通道维度划分出多份，在不同线程中调用使用。

(2) 计算归一化

计算归一化部分与计算均值和方差部分类似，NHWC 数据格式的通道维度在最后，所以所有的数据访问和运算都可以使用数据并行的方式优化，在 ARM 架构 CPU 中可以通过使用可变长向量拓展指令集使用，通过预测寄存器的指导计算与存取，可以让我们完全不用对尾数进行特殊处理，不足向量长度的尾数同样可以通过向量完成计算。

在高性能批归一化函数实现中，因为批归一化的操作针对每个数据元操作，而平均值、方差值、 γ 和 β 都是数据通道长度，并且连续存储在内存中，所以我们在大循环中对数据单元进行循环，在内部使用可变长向量拓展向量指令集操作向量计算归一化行为。具体的，我们每次对数据通道进行向量长度的迭代，连续通过地址读取向量长度的输入数据、方差数据、 γ 、 β 、到向量中，对于 γ 和 β 的数据地址如果为空地址则默认零值向量，而对于 ϵ 则广播成常量向量；然后在计算中依次使用向量完成方差数据与 ϵ 向量求和，再将求和数据通过使用可变长向量拓

展指令集中的 svrsqrte 指令和 svrsqrts 指令，以及 svmul 乘法指令求的平方根的倒数近似值，因为 ARM 底层是通过近似的方式计算平方根的结果，对于 svrsqrte 指令是求去平方根倒数近似值的作用，然后用它乘以原来的数据得到方差的平方根的数据，再通过 svrsqrts 指令得到平方根的倒数的近似值。最后使用乘法指令乘以倒数完成了除法操作，写回到输出数据地址中就完成了一个数据元的所有通道上的计算。

3.2.3 全连接层函数实现

全连接层函数的输入数据格式是确定的，输入数据格式为图片数据量乘以输入通道维度的数据，即 $MB \times IC$ ，然后权重的数据格式为输出维度乘以输入维度的数据即 $OC \times IC$ ，最后将会得到 $MB \times OC$ 的输出数据。虽然在一定角度上来看是一种转置矩阵乘的操作，但是在 FTEngine 中，我们没有使用矩阵乘的方式完成，因为当单张图片传入数据的时候，转置矩阵乘的操作将会退化为向量矩阵乘的操作，在这个情况下，如果还是通过转置再调用矩阵乘的极端实现将会极大的影响性能。所以我们选择为全连接层函数实现高性能的向量矩阵乘。

针对全连接层的计算模式，针对输入维度和输出维度的计算先后顺序我们实现了两种全连接函数的向量化实现，分别是先遍历 OC 维度的方式和先遍历 IC 维度的计算方式，在遍历 IC 维度的函数中，我们每次用两个向量寄存器存储输入数据的数据，然后内部遍历 OC 维度，使用两个向量寄存器依次存储权重的数据，然后将两组向量相乘，并将存储结果的向量使用 svaddv 指令进行向量内求和并累加到输出。

而另一种方式先遍历输出维度再遍历输入维度，这种方式每次计算出输出维度中的单个维度的最终结果，在循环中使用两个向量寄存器存储输入数据，使用两个向量寄存器存储权重的数据，然后使用两个向量寄存器存储中间数据，不断循环使用乘加指令，在循环结束时候调用向量内求和指令求得结果存入结果中。

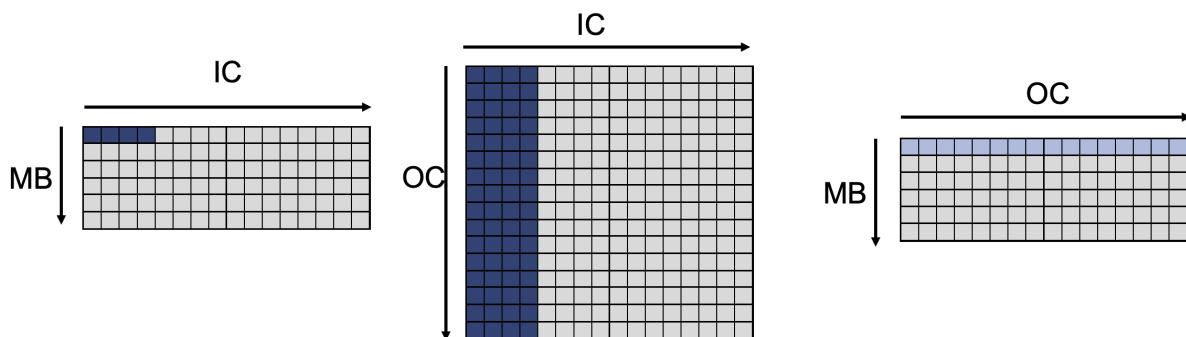


图 3.7 重用向量方法的全连接函数示意图 (浅色代表没有计算完全)

我们将第一种方式称为 A，如图3.7，将第二种方式称为 B 如图3.8，综合比较 A 实现和 B 实现，A 方式具有更加优秀的数据局部性，在数据量较小的情况下具

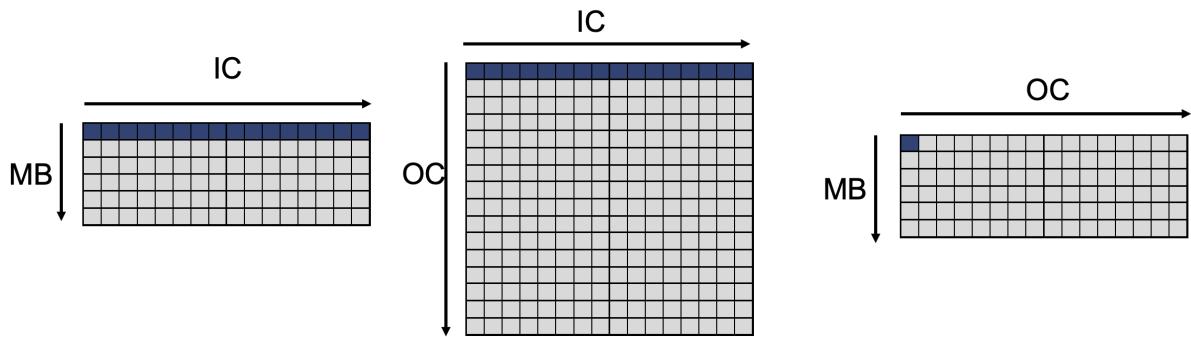


图 3.8 朴素方法的全连接函数示意图

有更加优秀的表现，但是 B 的方式具有更加稳定的性能表现，综合考虑之后，打算在权重数据大小小于二级 cache 大小的时候选择调用 A 方案，在大于二级 cache 大小的时候选择调用 B 方案。

3.2.4 激活层函数实现

激活层的算子实现比较简单，除了 Softmax 激活函数之外，激活层的操作针对数据中的所有数据进行相同的数据操作。针对激活函数，FTEngine 中暂时实现了 Sigmoid, Tanh, Relu, Leaky_Relu, Softmax 五种激活函数，针对前四种激活函数，FTEngine 中采用同一个编程接口，函数接口统一预留 alpha 和 beta 值，算法内部选择是否是用，通过传入枚举变量激活种类进行控制激活函数具体的实现，

具体的方法是：将四种函数实现为参数包含作为输入的支持单个可变长向量指令集的向量寄存器和单个支持可变长向量指令集的预测寄存器以及作为输出的单个支持可变长向量指令集的向量寄存器的核函数，在运行期间将多线程的循环以及任务均衡化部分还有向量存取部分放置在外层，因为激活函数的相同运算特征，这样的设计可以大大提高代码重用度，而调用真正激活函数的方法是，在进入函数的时候通过函数指针获取具体的激活内核函数，然后在循环中通过函数指针完成对激活函数内核的调用。

而针对 Softmax 函数，FTEngine 中选择单独进行实现，对于 Softmax 的实现类似于归一化函数的实现，也需要分为两层进行实现，第一步需要求出局部的最大值，第二步根据求出的最大值计算 Softmax 函数。针对调用 Softmax 的函数的数据通常输入格式为 $N \times IC$ (N 为图片数量， IC 为输入的数据通道)，我们在计算时，图片与图片之间计算隔离，计算存在于每个图片的内部。所以我们很容易给 Softmax 的并行方案设定在外层对图片的循环之中。

首先需要求取每个图片中的最大值，在求取最大值时候，我们使用单个向量对图片的所有通道数据进行遍历，另一个向量全局维护最大值的部分，不断对所有数据进行比较，最终得到一个存储最大值的向量，最后通过向量的横向操作获得向量内的最大值也就是该图片所有通道中的最大数据。我们通过使用 intrinsic

```

1 void (*func)(svfloat32_t src, svbool_t pg, svfloat32_t dst, float alpha,
2           float beta) = NULL;
3 switch (op) {
4     case sigmoid:
5         func = sigmoid_kernel;
6         break;
7     case relu:
8         func = relu_kernel;
9         break;
10    case leaky_relu:
11        func = leaky_relu_kernel;
12        break;
13    case tanh:
14        func = tanh_kernel;
15        break;
16    default:
17        printf("unsupported");
18        break;
19 }

```

图 3.9 激活层函数类型判断逻辑图

```

1 void logits_1d_max(const float* in, float* out, int workSize) {
2     auto vec_max = svdup_n_f32(-FLT_MAX);
3     int x = 0;
4     auto pg = svwhilelt_b32(x, workSize);
5     const auto all_true_pg = svptrue_b32();
6     do {
7         // compute
8         auto current_value = svld1(pg, in + x);
9         vec_max = svmax_m(pg, vec_max, current_value);
10
11        // iterate
12        x += svcntw();
13        pg = svwhilelt_b32(x, workSize);
14    } while (svptest_any(all_true_pg, pg));
15    auto maxVal = svmaxv(all_true_pg, vec_max);
16    *out = maxVal;
17 }

```

图 3.10 intrinsic 指令求最大值函数示例图

函数指令的方式。

需要注意的是，对于每次比较求最大值的部分，我们需要指定使用可变长向量指令集控制运算为 Merge 模式，对于预测寄存器中指示为真的数据单元我们做比较计算，对于预测寄存器中指示为假的数据单元我们保存全局存储的最大值。

在求取完了最大值之后，我们就需要对图像中每个元素计算他的 Softmax 值。因为我们需要实现 Softmax 和 LogSoftmax 函数，我们通过在实现的时候传入布尔变量判断是否需要求对数，在实现上通过判断传入的布尔变量进行变化。之后我们通过 intrinsic 函数指令使用可变长向量拓展指令集首先完成求取单个元素与最大值的差的指数值和所有指数值的累和，并把所有的指数值存储在临时存储空间中，第二步我们从第一步的临时存储空间中取出第一步计算的指数值除以第一步

```

1   { // 计算指数并求和
2     const auto max_val = *max_ptr;
3     const auto vec_max = svdup_n_f32(max_val);
4     const auto vec_beta = svdup_n_f32(beta);
5     auto vec_sum = svdup_n_f32(0.f);
6     int x = 0;
7     svbool_t pg = svwhilelt_b32(x, workSize);
8     do {
9       auto vec_elements = svld1(pg, in_ptr + x);
10      vec_elements =
11        svmul_z(pg, svsub_z(pg, vec_elements, vec_max), vec_beta);
12      if (!is_log) {
13        vec_elements = svexp_f32_z(pg, vec_elements);
14        vec_sum = svadd_m(pg, vec_sum, vec_elements);
15      }
16      svst1(pg, exp_tmp + x, vec_elements);
17      if (is_log) {
18        vec_sum = svadd_m(pg, vec_sum, svexp_f32_z(pg, vec_elements));
19      }
20      x += svcntw();
21      pg = svwhilelt_b32(x, workSize);
22    } while (svptest_any(all_true_pg, pg));
23    sum = svaddv(all_true_pg, vec_sum);
24    if (is_log) {
25      sum = (float)(log(sum));
26    } else {
27      sum = 1.f / sum;
28    }
29  }
30  // 求激活值
31  {
32    int x = 0;
33    svbool_t pg = svwhilelt_b32(x, workSize);
34    auto vec_sum = svdup_n_f32(sum);
35    do {
36      auto vec_in = svld1(pg, exp_tmp + x);
37      auto normalized_value = svdup_n_f32(0.f);
38      if (is_log) {
39        normalized_value = svsub_z(pg, vec_in, vec_sum);
40      } else {
41        normalized_value = svmul_z(pg, vec_in, vec_sum);
42      }
43      svst1(pg, out_ptr + x, normalized_value);
44      x += svcntw();
45      pg = svwhilelt_b32(x, workSize);
46    } while (svptest_any(all_true_pg, pg));
47  }

```

图 3.11 intrinsic 指令 Softmax 函数核心部分示例图

计算出来的求和得到最终的 Softmax 值。对于 LogSoftmax，因为对数函数的向量操作需要额外实现的原因以及对数函数内部的乘加可以转换为对数函数外部的加减的原因，我们把除以和的对数改为了对数外部的减法操作实现。需要注意的是，指数函数并不被原生提供，对于指数函数我们需要额外实现，对于指数函数的实现方法是通过参考^[48]里描述的多项式近似的方法，我们在国产 ARM 架构 CPU 处理器上使用可变长向量拓展指令集进行实现。

3.3 本章小结

本章首先描述了 FTEngine 在 AI 库中的位置及它所扮演的角色，然后介绍了池化层、归一化层、激活层、全连接层在 FTEngine 中的高性能实现方法，我们将在下一章节介绍在 FTEngine 中重点进行优化实现的卷积层。

第 4 章 高性能卷积函数设计与实现

因为卷积层是神经网络中最耗时的部分，所以在 FTEngine 中对它进行了深度的设计和优化。在 FTEngine 中使用 Im2col+GEMM 的策略实现卷积函数。在本章节将介绍如何实现 FTEngine 中的高性能卷积函数。FTEngine 中针对 Im2col 函数进行深度分析和设计，针对列主序的矩阵乘函数重新设计了 NCHW 格式和 NHWC 格式输入图的转换方法，并对 Im2col 的向量化转换方法进行探讨分析。然后基于 GOTOBLAS 提出的方法并借鉴 BLIS 的循环框架实现了高性能矩阵乘函数，并基于国产 ARM 架构 CPU 的缓存大小计算了矩阵分块参数，使用包括但不限于数据重排、矩阵分块、循环展开等方法进行优化，使用汇编语言使用可拓展向量指令集实现了高性能的汇编内核函数，最后探讨分析了高性能矩阵乘的多线程实现方法。

4.1 高性能 Im2col 函数实现

4.1.1 Im2col 算法分析

Im2col 算法原名为 Image to column，原 Im2col 算法的实现方法如图4.1^[20]所示，该算法每次将卷积窗口覆盖的特征图元素转换为连续存储的行的方式组成左乘矩阵，并将卷积核转换为纵向的列并组合成矩阵右乘矩阵，如此组成了矩阵乘的左右矩阵。Im2col 算法将输出图像的行和列压缩成左乘矩阵的行，卷积核权重数量就成了矩阵乘中左乘矩阵的列和右乘矩阵的行，卷积核的输出通道数成为了右乘矩阵的列。将卷积的计算转换成两个矩阵相乘，最终的结果的行就是输出特征图的特征值数量，输出的列就是输出特征图的输出维度数量。

但是原文中这种转换的方式在具体的实现时候并不是一个高效率的实现，首先考虑输入特征图转换为左乘矩阵的步骤，无论矩阵数据是行主序方式存储还是列主序方式存储的模式，以卷积窗口作为转换单元的做法不可避免的会有不连续读取的情况出现；当矩阵数据为行主序方式存储的情形下，单个卷积窗口的数据的存储是连续进行的，而每次到下一个特征图的读取将是一次不连续读取的操作，如果先进行卷积窗口的遍历，那么下一次存储也会是不连续存储的情形。再考虑当矩阵数据为列主序方式储存时，这种转换方法显然是性能较低的方法，对于每一次的存储操作都将是不连续的存储行为，性能更是大打折扣。但是在列主序的情况下，作为需要转换成列的卷积核可以不用更多额外操作。

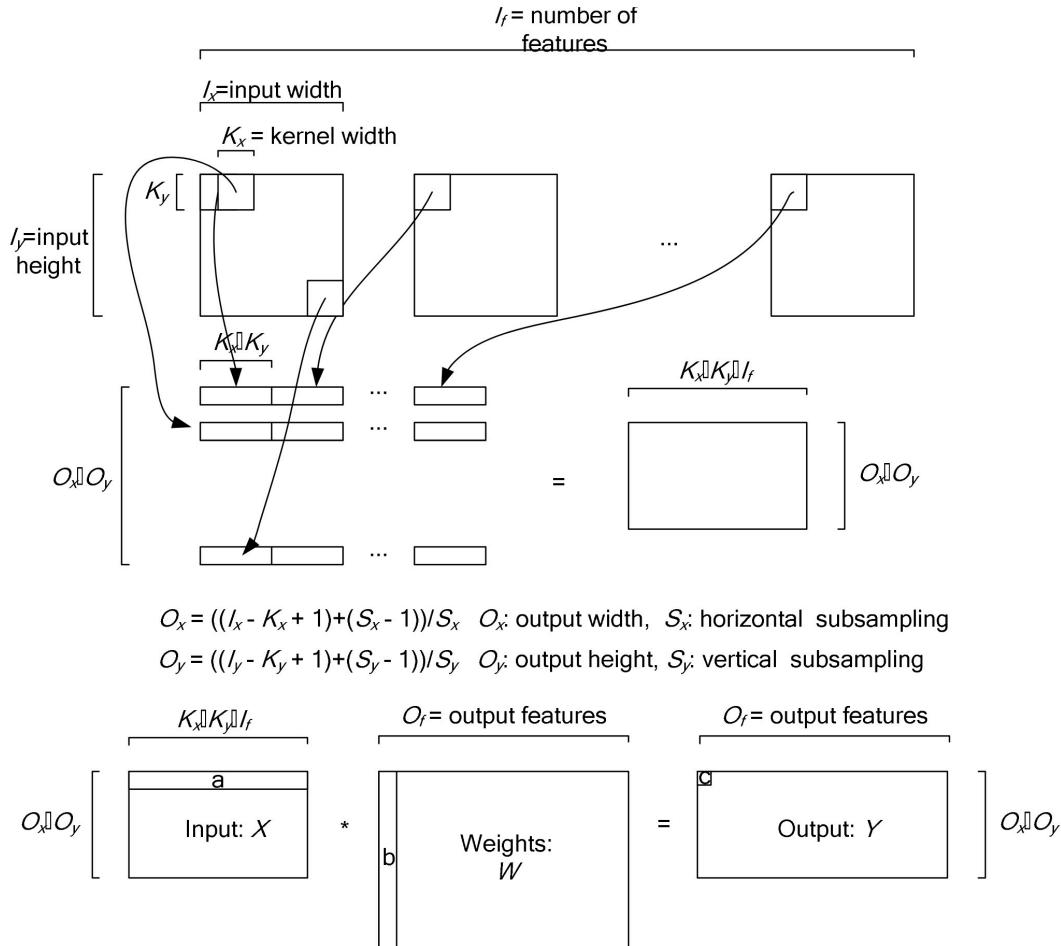


图 4.1 Im2col 算法原图

4.1.2 高性能 Im2col 算法转换设计

我们通过分析 Im2col 原文中转换时遇到的数据访存不连续的问题，做出了改进。我们抛弃了基于卷积窗口作为转换单元的方式，采用了一种新的方式，即针对卷积窗口中特定权重单元作为转换单元。具体来说，我们按照卷积窗口中某个特定位置在特征图像中遍历时所对应的特征图的点作为转换依据。这种改进的好处在于，可以大大减少 cache miss 的概率。例如，假设卷积窗口大小为 $K_w \times K_h$ ，卷积输出的窗口为 $O_w \times O_h$ ，因为输出特征图的宽 O_w 通常远大于卷积窗口的宽 K_w ，当输入特征图足够宽时，卷积窗口中跨行的行为会导致跨长距离读取数据并导致 cache miss。而我们采用的第二种方式，由于上下两行数据中首尾存储连续，所以在跨行读取时不会出现跨长距离读取数据而导致 cache miss 的情况。对应图4.2中转换方法。

并且，卷积窗口位置点作为转换单元的方式在列存储的情形下可以做到存储连续，因为在转换之后的左乘矩阵中，每一列实际上对应的正是某一个卷积和权重在卷积过程中对应的特征图的值，所以是列顺序优先的。而且列主序的存储方式下，对于卷积核可以无需转换直接作为右乘矩阵使用。

最后面对 NCHW 输入格式的卷积转换方式如图4.2所示，转换后我们的左乘矩阵行数为(输出矩阵的行×输出矩阵的列)，左乘矩阵的列为(输入矩阵的通道×卷积核的列×卷积核的列)，右乘矩阵的列数为(输入矩阵的通道×卷积核的列×卷积核的列)，右乘矩阵行数为(卷积的输出通道数目)，左乘矩阵与右乘矩阵相乘得到结果矩阵，结果矩阵的行为(输出矩阵的行×输出矩阵的列)，结果矩阵的列为(卷积的输出通道数目)，矩阵相乘的运算实际上也就完成了卷积的过程

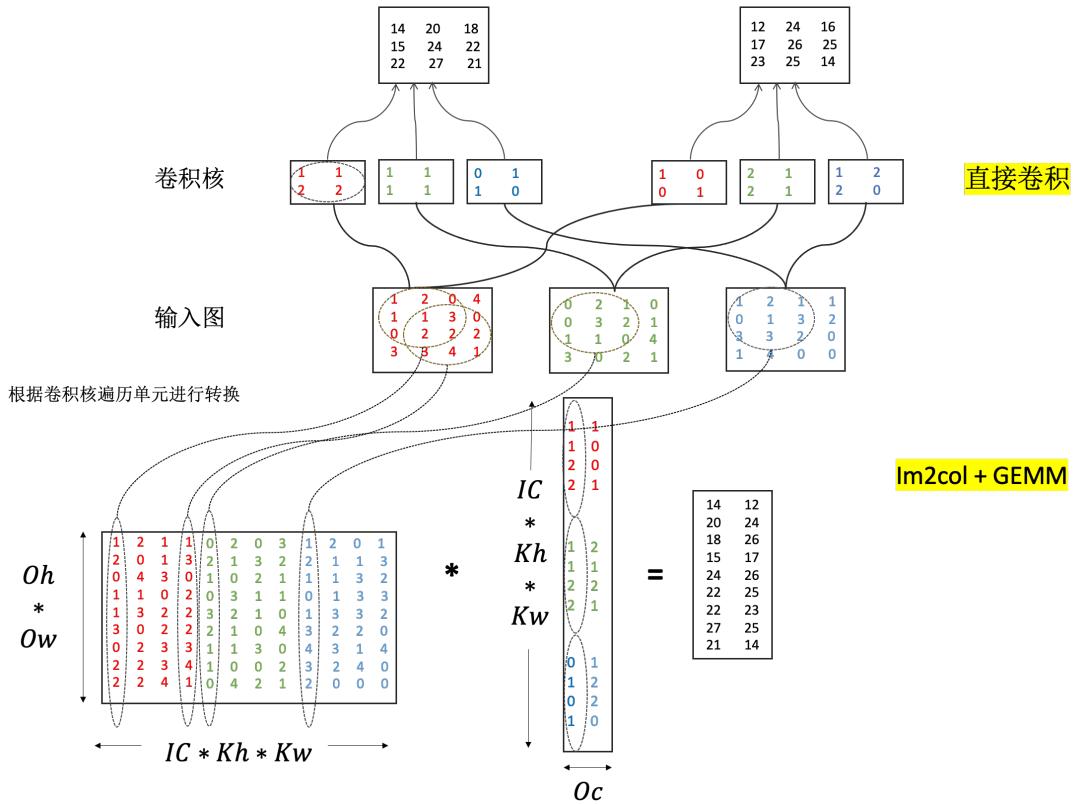


图 4.2 NCHW 数据格式 Im2col 转换示意图

输入特征图格式为 NHWC 格式的数据时，权重的数据格式为（输出通道 OC，卷积核高 KH，卷积核宽 KW，输入通道 IC）格式，面对 NHWC 输入格式的卷积转换方式如图4.3所示，转换的方法相对于 NCHW 输入格式不同，针对 NHWC 的形式，因为卷积核的权重之间相隔输入通道数量个数据，当使用之前的方式放置卷积权重时，输入通道维度和输出通道维度会无法分割。而且因为数据维度的关系，当使用之前的转换方式转换 NHWC 格式的数据时，输入通道的维度也会和输出特征图数目的维度捆绑，将会导致我们的转换失败。

所以在输入格式为 NHWC 格式的数据时我们采取将卷积窗口作为转换单元的依据针对每一个卷积窗口有(输入矩阵的通道×卷积核的列×卷积核的列)个数据进行转换，将它转换成右乘矩阵的一列，一共有(输出矩阵的行×输出矩阵的列)个列进行转换，对于卷积核的转换，我们采取直接对卷积核进行转置操作，转置之后可以直接作为左乘矩阵使用了。

输入格式为 NHWC 进行转换的情况比较容易发现由于通道维度最后的关系，在对卷积窗口中处于同一行的数据进行拷贝的时候，往往每次可以连续拷贝较长的一组数据，并且存储时候也是连续的进行存储，说明我们通过设计转换方式已经获得了非常高性能的拷贝情况。

而对于输入格式为 NCHW 的情况，我们结合卷积核与特征图进行计算的顺序和模式进行分析可以发现，特征图转换的左乘矩阵的行数其实就是卷积核权重的数目，它对应着卷积核权重从左到右从上到下顺序的元素，而特征图转换的左乘矩阵的列则代表的该行相对应的卷积核权重需要乘的对象的值，这个列的大小为(输出矩阵的行 \times 输出矩阵的列 \times 输入矩阵的数量)，也就是我们的卷积窗口中特定位置单元在特征图中所对应的数据，我们后续针对 NCHW 格式的转换方式如图4.3，图中虚线代表着卷积核权重所遍历的特征图数据，以及转换后的位置。

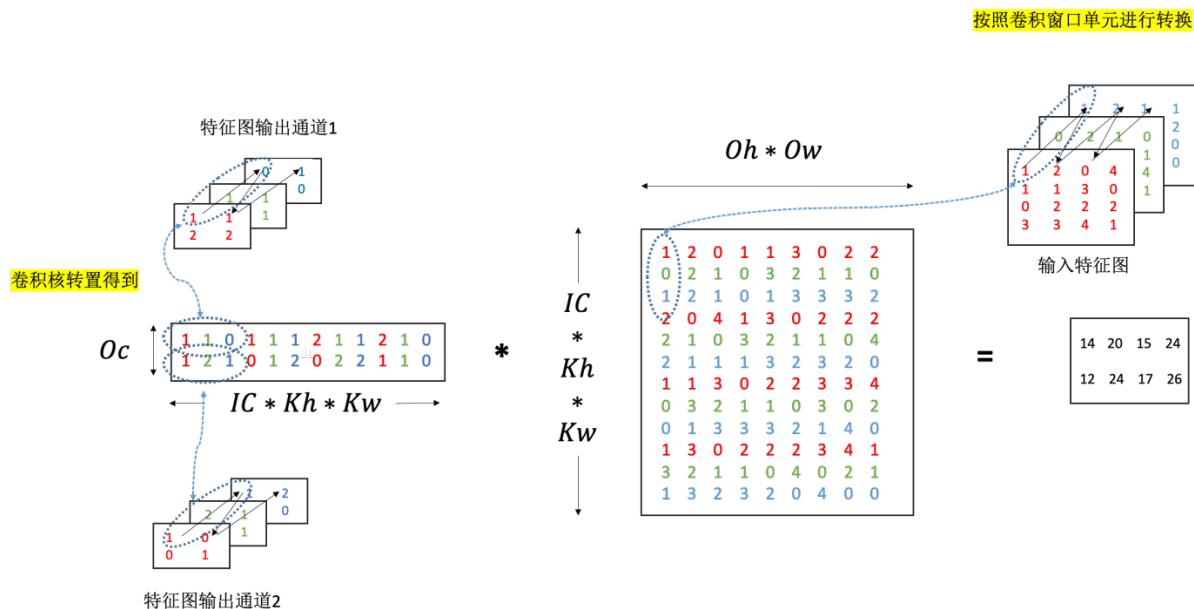


图 4.3 NHWC 数据格式 Im2col 转换示意图

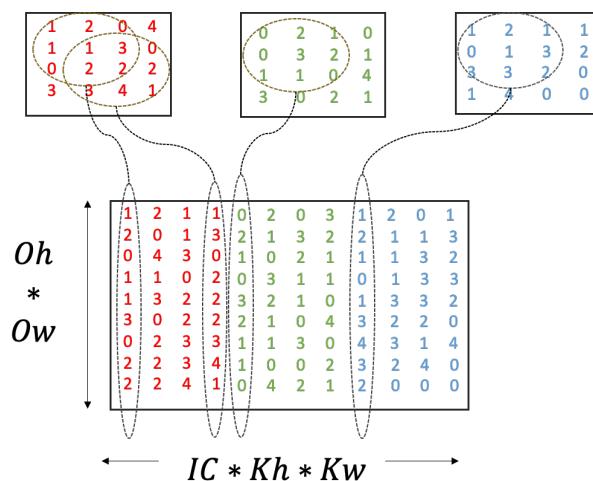


图 4.4 Im2col 转换拷贝示意图（虚线代表某个权重访问的元素）

实现转换有两个方法实现，一种方式是以卷积核每次覆盖的特征图区域作为一次输出，也就是每次转换一行的方式进行；另一种是以卷积核中的单个元素作为单位，在特征图上扫过的特征图元素作为输出，也就是每次转换一列的方式进行。考虑到我们矩阵是以列主序的方式存储数据，另外一列的数据相对一行的数据通常包含更多，我们选择了第二种方式遍历转换。在第二种方式的转换中，通过细致的观察可以发现，当卷积的窗口跨度为 1 的情况下，数据的转换通常为连续的存取，而窗口跨度不为 1 的情况下，我们也可以使用可变长向量指令集支持的分散加载特性，跨步长对数据进行存取。所以这部分的存取在长度足够的情况下使用向量存取进行优化可以得到客观的性能加速。

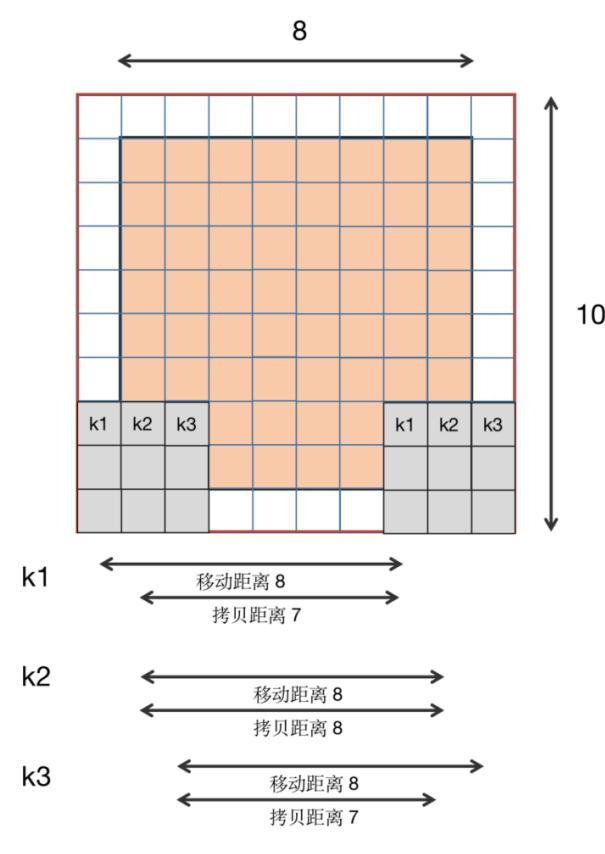


图 4.5 Im2col 改进算法拷贝边界距离分析图

以(输出矩阵的行 \times 输出矩阵的列 \times 输入矩阵的数量)维度作为转换维度进行转换，我们可以按照输出矩阵的行和列进行遍历，将特征图的转换问题转化为简单的数据拷贝操作，我们对这个数据拷贝的调用操作需要求解三个变量：输入数据地址、输出数据地址、数据长度。为了得到这三个变量需要考虑两个因素：第一个因素是填充的影响，因为卷积时特征图的边缘可能需要处理填充。如果需要填充，则需要在输入图像的周围扩展边界，用 0 扩展，在拷贝的时候我们需要考虑到填充 0 的影响，因为填充的 0 并不是真正的存在地址之中，所以这一个步骤我们只有依靠跳过填充 0 进行拷贝进行；第二个因素来自卷积核窗口带来的边缘

影响，需要根据卷积核元素距离卷积核上下左右边缘位置处理进行处理。而这两个因素带来的计算是我们使用向量拷贝操作所必须接受的额外开销。

图4.5上的灰色区域表示的是卷积核窗口的覆盖区域，这里只罗列了最后一行最左和最右的两个区域，对于卷积核中第一个元素 k_1 ，它在卷积窗口的滑动过程中运动了 8 个元素单位、但是覆盖特征图只有 7 个元素单位，所以最终拷贝 7 个元素单位，同理可以计算卷积核中第二个元素 k_2 和 k_3 。于是我们可以推导出公式

$$\begin{aligned} \text{input}_{\text{offset}} &= \text{Max}((\text{col}_{\text{kernel}} - \text{pad}), 0) \\ \text{output}_{\text{offset}} &= \text{Max}((\text{pad} - \text{col}_{\text{kernel}}), 0) \\ \text{left}_{\text{offset}} &= \text{Max}((\text{col}_{\text{kernel}} - \text{pad}), 0) \\ \text{right}_{\text{offset}} &= \text{Max}((W_{\text{kernel}} - (\text{col}_{\text{kernel}} + 1) - \text{pad}), 0) \\ \text{Len} &= \text{Image}_{\text{width}} - \text{left}_{\text{offset}} - \text{right}_{\text{offset}} \end{aligned} \quad (4.1)$$

其中： $\text{input}_{\text{offset}}$ 是输入时候需要跳过区域长度偏移； $\text{output}_{\text{offset}}$ 是目的地址需要跳过区域长度偏移； $\text{left}_{\text{offset}}$ 和 $\text{right}_{\text{offset}}$ 是卷积核移动时候实际能够拷贝的元素边界距离图像边界的距离。拷贝的长度就是图像长度减去左右偏移长度。 $\text{input}_{\text{offset}}$ 和 $\text{output}_{\text{offset}}$ 与我们遍历中得到的原图像地址和目标图像地址之和就可以得到拷贝的两个地址参数，而 Len 长度就是我们的拷贝长度，加上我们的数据跨度的参数就可以调用我们的高性能向量拷贝函数了。

4.1.3 使用 SIMD 技术实现高性能向量拷贝内核

向量拷贝内核是比较简单的内核实现，将一个地址内部连续元素拷贝到另外一个地址所指向的内存区域中，假设有 N 个元素需要拷贝，在标量运算中需要 $O(N)$ 的时间复杂度单位，当使用国产 ARM 架构 CPU 处理器上的向量寄存器进行优化时，可以获得接近 8 倍的加速提升效果，但是考虑到向量的输入输出相对标量来说会需要更多的时钟周期，所以具体的提升并没有那么高的加速比。针对 stride 为 1 的情况我们拷贝内核的核心部分实现如图4.6，针对 stride 不为 1 的情况，我们的拷贝内核核心实现如图4.7。

因为一次性的拷贝数据大小接近一个卷积输出图像的行的长，所以当输出矩阵行较长的时候，我们可以采用每轮循环多个向量进行加载存储的方式，如此我们同时吸收了循环展开带来的优势也利用了更多的向量寄存器。

当使用向量寄存器或者使用循环展开的时候都会碰到尾处理情况，比如对一个需要迭代 15 次的循环进行循环展开 4 次，那么就需要循环 3 次循环展开，还剩下 3 次单独的循环需要单独处理。但是使用可变长向量拓展指令集控制国产 ARM 架构 CPU 的向量寄存器可以不用考虑循环展开所面对的尾处理情况，可变长向量拓展指令集使用预测寄存器进行判断向量寄存器的各个位置元素是否有效，是否对该位置的元素做出行为操作。而对于预测寄存器的更新我们可以使用

```

1 do {
2     svst1_f32(pg0, out_ptr, svld1(pg0, in_ptr));
3     svst1_f32(pg1, out_ptr + vec_len, svld1(pg1, in_ptr + vec_len));
4     svst1_f32(pg2, out_ptr + vec_len2, svld1(pg2, in_ptr + vec_len2));
5     svst1_f32(pg3, out_ptr + vec_len3, svld1(pg3, in_ptr + vec_len3));
6     x += vec_len4;
7     pg0 = svwhilelt_b32(x, valid_len);
8     pg1 = svwhilelt_b32(x + vec_len, valid_len);
9     pg2 = svwhilelt_b32(x + vec_len * 2, valid_len);
10    pg3 = svwhilelt_b32(x + vec_len * 3, valid_len);
11
12    out_ptr += vec_len4;
13    in_ptr += vec_len4;
14 } while (svptest_any(pg0, all_true_pg));

```

图 4.6 stride 为 1 时的向量拷贝内核代码图

```

1 svint32_t idx = svindex_s32(0, stride);
2 do {
3     svst1_f32(pg0, out_ptr,
4             svld1_gather_s32offset_f32(pg0, in_ptr, idx));
5     svst1_f32(pg1, out_ptr + vec_len,
6             svld1_gather_s32offset_f32(pg1, in_ptr + vec_len, idx));
7     svst1_f32(pg2, out_ptr + vec_len2,
8             svld1_gather_s32offset_f32(pg2, in_ptr + vec_len2, idx));
9     svst1_f32(pg3, out_ptr + vec_len3,
10            svld1_gather_s32offset_f32(pg3, in_ptr + vec_len3, idx));
11    x += vec_len4;
12    pg0 = svwhilelt_b32(x, valid_len);
13    pg1 = svwhilelt_b32(x + vec_len, valid_len);
14    pg2 = svwhilelt_b32(x + vec_len * 2, valid_len);
15    pg3 = svwhilelt_b32(x + vec_len * 3, valid_len);
16
17    out_ptr += vec_len4;
18    in_ptr += vec_len4;
19 } while (svptest_any(pg0, all_true_pg));
20 }

```

图 4.7 stride 不为 1 时的向量拷贝内核代码图

“whilelt x, len” 指令，这个指令对 x 进行迭代加 1，同时将预测寄存器上的相应有效位置 1，直到 x 等于 len；而对于 x 大于等于 len 的时候，将会将预测寄存器上的有效位置零。最后它就会得到一个处理好的预测寄存器。使用这个预测寄存器，我们不用担心我们对于数据的读取和写入会发生越界的行为，因为预测寄存器的有效位将限制住向量寄存器的行为范围

4.2 高性能矩阵乘算法实现

矩阵乘法属于 BLAS 规范中的 Level 3 的规范，目前 OpenBLAS^[49]、BLIS (Blas-like Instantiation Software)、MKL (Math Kernel Library)、APL^[50] (Arm Performance Library) 都实现了高性能的矩阵乘算法，针对不同的架构进行了性能调优。目前市面上 CPU 端实现的矩阵乘法几乎都是根据 GOTOBLAS 所提出的高

性能矩阵乘方法框架衍生拓展出来。本项目所实现的高性能矩阵乘也是改自BLIS矩阵乘法的框架，使用了矩阵分块、循环展开、数据重排、数据预取、汇编层面排流水、SIMD向量指令集等优化手段，针对体系结构设计了高性能的矩阵乘法。矩阵乘法的公式同：

$$C = \alpha A \times B + \beta C \quad (4.2)$$

这里C存储结果矩阵维度为M×N，A代表着矩阵乘的左乘矩阵维度为M×K而B代表着矩阵乘的右乘矩阵，维度为K×N， α 和 β 分别是乘法系数和偏移系数。通常将矩阵乘的规模简化称为M×N×K。

如图4.8就是一个基本矩阵乘法的示例：

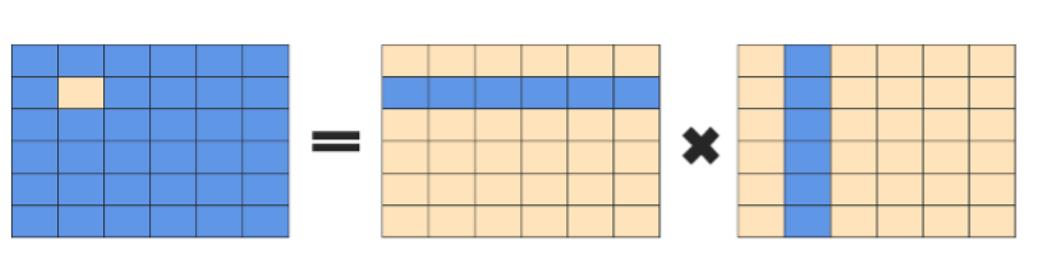


图 4.8 矩阵乘示例图

算法 4.1: 朴素矩阵乘实现代码

```

Input: Matrix A
Input: Matrix B
Input: Matrix C
Input: int M
Input: int N
Input: int K

1 for i = 0; i < M; ++i do
2   | for j = 0; j < N; ++j do
3     |   | for k = 0; k < K; ++k do
4       |     |   C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
5     |   | end
6   | end
7 end
```

最简单的矩阵乘法可以通过3层for循环完成计算，然而在循环中，即使我们采取了最优的迭代次序来实现矩阵乘法，仍然会有非常大的缓存不命中几率，非常的内存不友好，并且未能很好的与硬件进行配合来获得优势。

本文中实现的高性能的 GEMM 的分块算法参照了 GOTOBLAS 论文中介绍的分块方法，又结合了 BLIS 的循环分块框架，将矩阵进行切割分块成小矩阵进行矩阵乘法。本小节将介绍如何选取分块策略以及结合国产 ARM 架构 CPU 选择数据打包参数，再结合分块方式介绍如何通过数据重排，提高内部循环计算中数据连续性，再讲解汇编中使用的优化策略和多线程方法。

4.2.1 分块策略选取

如图4.9^[38]所示，GOTOBLAS 将矩阵分块分为三层六种方式，并将大矩阵代号为 M (Matrix, 矩阵)，将分出的长短边矩形称为 P (Pannel, 面板块)，将短短边矩形称为 B (Block, 小块)，以此对六种分块方式进行了方法区分。首先将通用矩阵乘 GEMM 划分成了 GEPP、GEMP、GEPM，然后再二次划分细化为 GEBP、GEPB、GEDOT 三种内核形式，之所以需要将矩阵划分小块就是为了让小矩阵填入缓存中为此让计算访存耗时比增大。

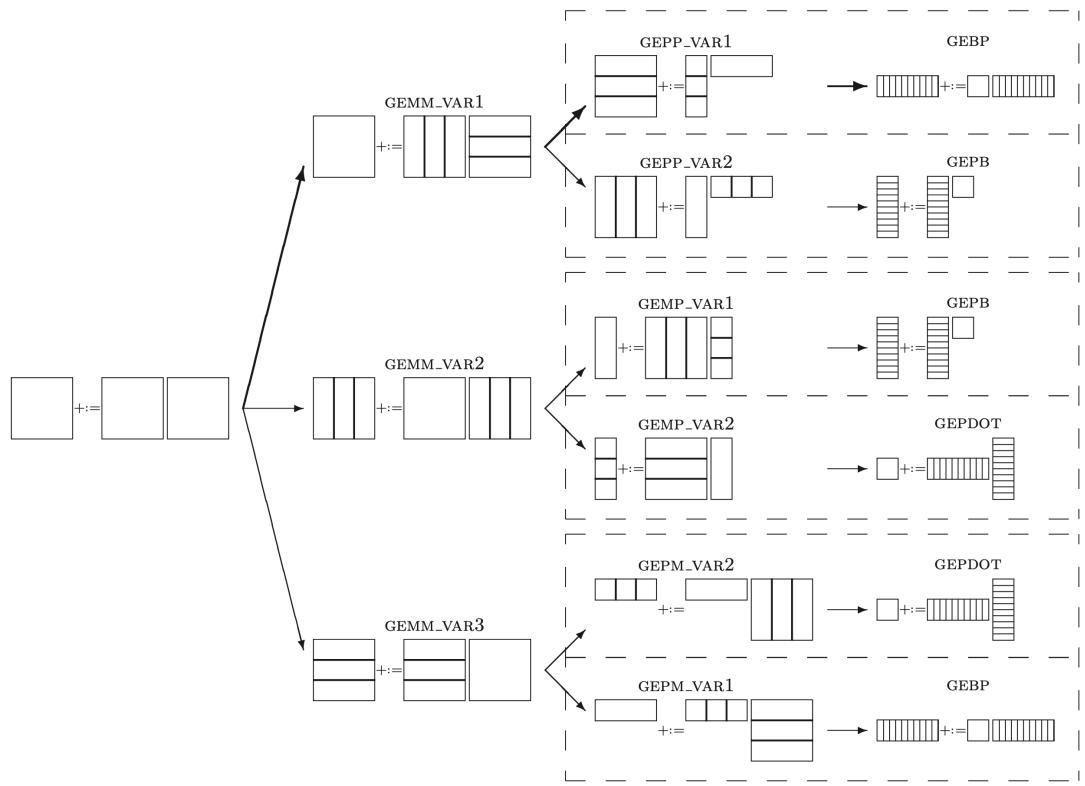


图 4.9 gotoblas 提出的分块方法

对以上六种分块方式进行分析，来挑选出最适宜国产 ARM 架构 CPU 的分块方法。在4.9^[38]图中，可以首先排除 4 和 5 两种分块方式，因为在第 4 和第 5 的分块方式中，需要对 A 矩阵的行和 B 矩阵的列也就是 K 进行迭代，在这种情况下，整个 C 矩阵块都会被循环迭代更新，而 C 矩阵块在这种顺序的循环更新中最多只能整体存入 2 级缓存中，数据的读入和写出都远远慢于寄存器或者 1 级缓存的直

接读写速度，所以 4 和 5 是最先淘汰的选项。

然后对比第 1 种分块方式和第 6 种分块方式，两种分块方式最后都是变成 GEBP 形式，第 1 种从 GEPP 形式分块得到，第 6 种从 GEPM 形式得到，在第 1 种分块方式 GEPP 中，B 矩阵的小 pannel 将会被重排打包起来，在计算中将通过合理配置参数大小将它放入缓冲区中，而 C 矩阵将会从主存中直接提取，但是可以使用寄存器计算 C 矩阵，在最后将 C 矩阵更新回内存，在这个步骤中，C 矩阵的写回可以隐藏在下一组的计算中，而在第 6 种分块方式 GEPM 中，B 矩阵从内存中读入，C 矩阵则可以通过合理配置参数大小放入缓冲区中，在循环的计算中，B 矩阵从内存中传入的时间可以被计算的时间给隐藏，但是最终在 GEPM 的循环结束时候需要将 C 矩阵的缓冲数据放回内存中，这部分的操作将会比第一种方式更加耗时，所以在这种情形下，我们舍弃了第 6 种分块方式。

第 2 种分块方式 GEPP 和第 3 种分块方式 GEMP 与第 1 种分块方式和第 6 种分块方式十分类似，第 3 种方式与第 6 种方式同为 GEMP 所以同理故不予考虑，第 2 种分块方式与第 1 种分块方式之间基本相同，两者之间互为转置模式。

4.2.2 缓存分配

经过以上的分块分析，我们选择好了 GEMM-GEPP-GEBP 的分块方式，然后我们结合图4.10^[51]讲解下如何进行循环拆分到我们的目标分块模式，图 XX 是我们的循环设计图，取自 BLIS 的框架示意图，该循环中的第 5 层、第 4 层、第 3 层循环描述了之前的分块方式，其中在第五层中多描述了 nc 的大小分块设计，对矩阵 B 添加了一步分块，下一步我们将讲述各个部分的内存将会如何放置在不同内存中以及对于数据的重排设计。

现代 CPU 是不支持程序员对于数据的直接控制的，缓存作为内存到寄存器之间数据的暂存器。当数据被请求的时候，CPU 首先会去检查缓存是否存在数据，如果存在则直接取回，否则需要先将数据搬运到低层次缓存中，并逐层搬运到更高级别的缓存中，在这个过程中，他们会挤出其他的数据缓存，针对数据的替换策略中，缓存通常用相联度来描述保持数据的性能，相联度越高则保持数据的能力越强。所以，当我们希望我们的分块后的数据能够按我们所期望的保存在缓存中，我们需要满足两个条件：

1. 期望被放入缓存中的数据在被访问计算期间连续且访问频繁；
2. 期望被放入缓存中的数据总数据能够满足缓存大小限制，不能出现缓存抖动现象（数据总数大于缓存能够装的大小，频繁挤出之后短期内需要使用的数据）。

为了满足这两个条件，我们采取两个措施来完成：

1. 对需要放入更深级别缓存的数据放在最内部循环顺序读取并控制好维度大

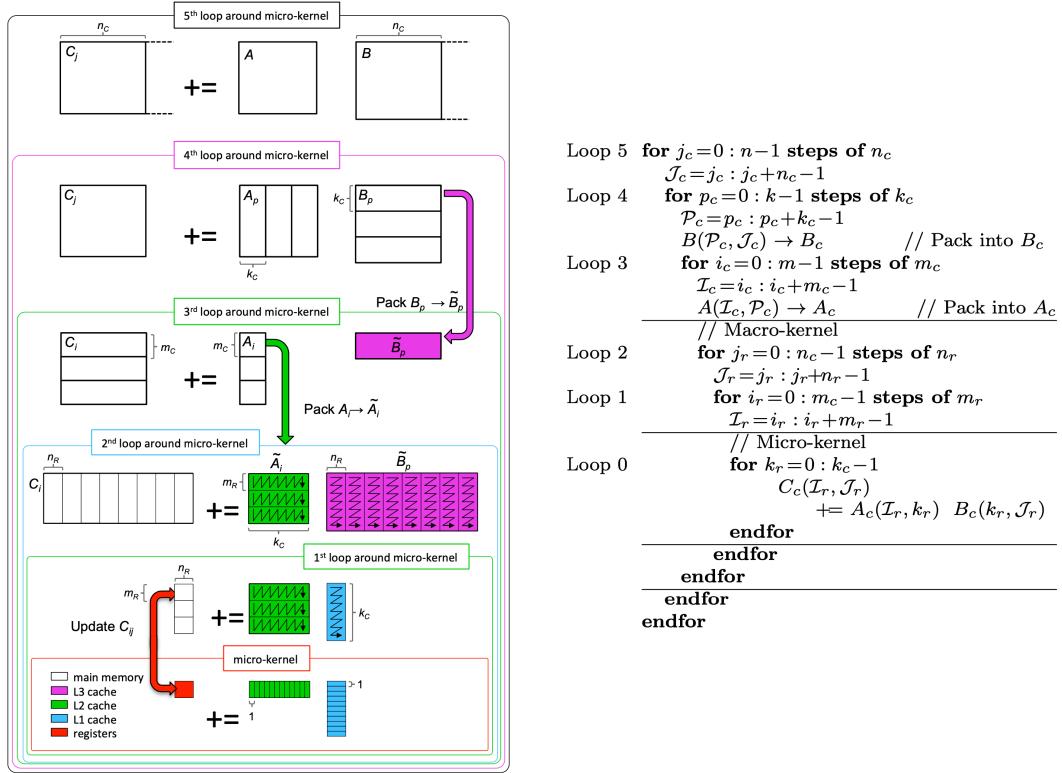


图 4.10 BLIS 的循环框架图

小。

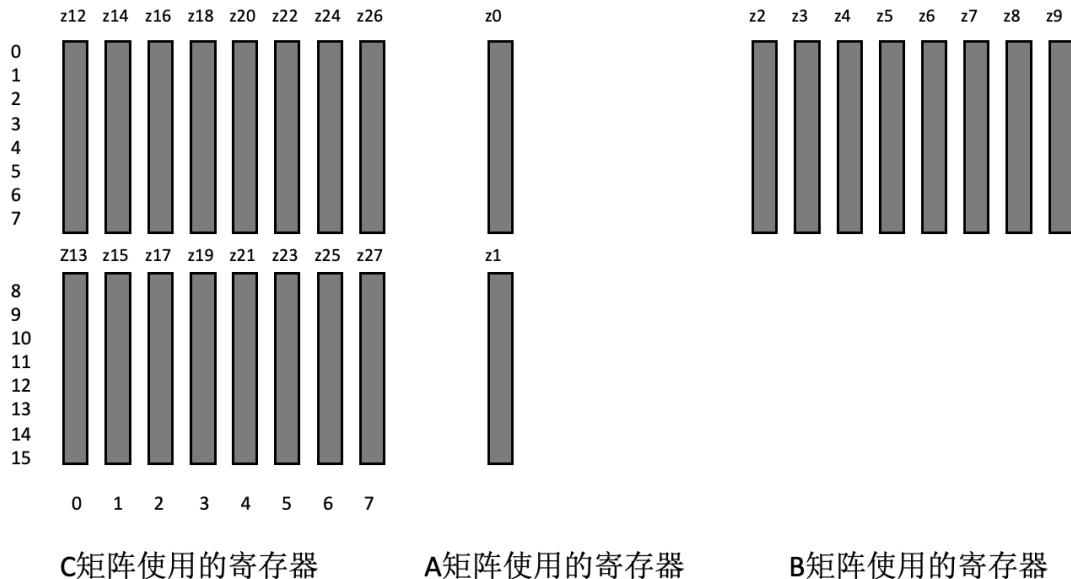
2. 对读取的数据需要进行数据重排，避免数据间跨度太大。

根据 GEBP 的大小，我们设计将 GEBP 定义为 Macro-kernel 外层内核，将最内层的循环叫做 Micro-kernel，数据主要存在寄存器和 1 级缓存。在第四层循环中，对 K 维度按 k_c 为跨度分割 A 矩阵和 B 矩阵并设计将 B 的 pannel 放入 3 级缓存中，大小为 $k_c \times n_c$ ；在第三层循环中 M 维度按 m_c 为跨度分割 A 矩阵为 block 并将 A 的 block 放入 2 级缓存中，大小为 $m_c \times k_c$ ，然后在 Micro-kernel 的内部循环中将 B 的 pannel 和 A 的 block 细分为条分别大小为 $m_r \times k_c$ 和 $k_c \times n_r$ ，B 的条将存在于 1 级缓存中，在寄存器级别计算出大小 $m_r \times n_r$ 为 C 的 block。

现在我们针对第二层循环和第一层循环进行分析设计分块参数采取范围，在第三层循环时候我们分割加载了数据规模为 $m_c \times k_c$ 的 A 矩阵 block，在内部的循环我们加载了数据规模为 $m_c \times n$ 的 C 矩阵 pannel 和数据规模为 $k_c \times n$ 的 B 矩阵 pannel 与 A 的 block 进行计算，将输出的数据规模为 $m_c \times n$ 的数据更新 C 矩阵 pannel。所以我们在这个步骤中搬运了规模为 $m_c \times k_c + (2 \times m_c + k_c) \times n$ 的数据，计算了规模为 $2 \times m_c \times k_c \times n$ 的数据量。为了使我们在搬运数据时候消耗的时间被计算所消耗的时间隐藏，我们对它们做除法可以得到计算数据量与传输数据量之间的比值为：

$$\begin{aligned}
 & \frac{2m_c k_c n}{m_c k_c + (2m_c + k_c) n} \frac{\text{flops}}{\text{memops}} \approx \frac{2m_c k_c n}{(2m_c + k_c) n} \frac{\text{flops}}{\text{memops}} \\
 & = \frac{2m_c k_c}{2m_c + k_c} \quad \text{当 } k_c \ll n
 \end{aligned} \tag{4.3}$$

所以这里可以得出我们需要将计算传输比最大化，因此我们需要让 k_c 远小于 n ，并且 $2m_c$ 与 k_c 之间尽可能接近。因为 k_c 的选取涉及到了汇编向量内核的计算范畴，我们首先选取 m_r 和 n_r 的大小，在这里我们提出了两套 m_r 和 n_r 的选取，分别为 16×8 和 16×10 ；对于 16×8 的选择是基于采用一半的寄存器供更新 $m_r \times n_r$ 规模的 C 小块使用为缘由的，使用 16 个 256 位的支持可变向量长拓展指令集的向量寄存器存储 C，使用两个支持可变向量长拓展指令集的向量寄存器存储 A，使用两个支持可变向量长拓展指令集的向量寄存器 B；而对于 16×10 的分块的理由是为了全部的 32 个 256 位支持可变向量长拓展指令集的向量寄存器，其中 20 个支持可变向量长拓展指令集的向量寄存器存储 C，使用两个支持可变向量长拓展指令集的向量寄存器存储 A，使用 10 个支持可变向量长拓展指令集的向量寄存器 B。图 4.11 就是 16×8 内核的寄存器分配图。

图 4.11 32×8 寄存器使用分配图

这里我们使用 $m_r \times n_r = 16 \times 8$ 为例进行其他参数选取分析。继续以每个部分的分块的数据量大概占据缓存一半大小为基准进行计算，根据 L1 缓存大小 64k 容纳 2048 个浮点数以及 n_r 为 8 得到 k_c 取值为 128，根据 k_c 值与 L2 缓存 2Mb 容纳 65536 浮点数的大小得到 m_c 大小 128，再根据 64MB 的 L3 缓存得到 n_c 可得 4096 数量。最终得到 $m_c \times k_c \times n_c$ 的参考数值为 $128 \times 4096 \times 128$ ，但是在实际的运行期间存在许多的不定因素，所以具体的分块参数需要我们去实际测试得到。

4.2.3 数据重新排序

在分块得到了 B pannel 和 A block 之后，我们需要对数据进行重排处理，这里有两个原因：首先对于分块后的小矩阵实际上是通过循环进行划分的分块，数据与数据之间的处理顺序和存储顺序之间并不匹配，如果没有进行数据重排，实际运算时候会出现内存抖动现象，那么将没法完成我们所设想的矩阵块精准位于对应缓存，而且有可能会导致最终效率低于朴素版矩阵乘。

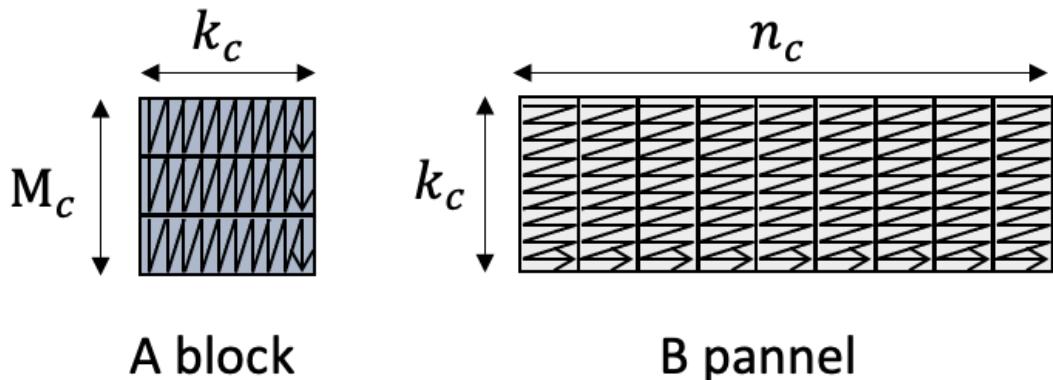


图 4.12 GEBP 数据重排后的数据分布示意图

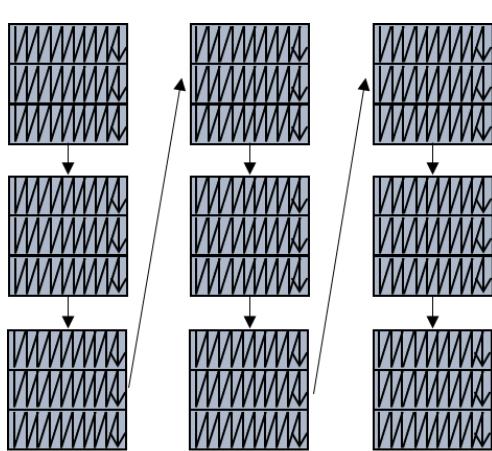


图 4.13 A 矩阵数据重排顺序示意图

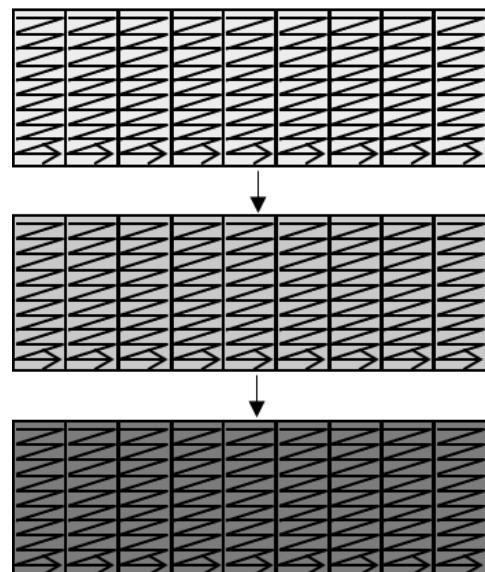


图 4.14 B 矩阵数据重排顺序示意图

具体的数据重新排序方法依照循环的次序排布，通过分块的循环控制分块的顺序，在分块时使用局部内存进行存储重排序后的矩阵数据，通过拷贝重定向的方式，首先对 B 的一整块 pannel 进行重排序，然后再对 A 矩阵的整个 block 进行数据重排序，最终 A 的 block 和 B 的 pannel 排序完如图4.12。针对 A 矩阵的 pack 顺序如图4.13，因为首先对 K 维度进行分块，再对维度进行分块，所以数据

重排序打包的顺序是先从上到下再从左往右；对 B 重排序打包的顺序参考图4.14；按照循环从上到下进行重排打包。

4.2.4 汇编层面优化

在矩阵乘的汇编层我们完成了对于 C 的一小块的计算，通过每次进行一层外积的方式对 C 的一小块进行堆叠计算，在这一部分我们主要通过使用汇编和可变向量长拓展指令集指令的方式提供了部分优化手段。我们主要使用了以下几种方式：

1. 循环展开：我们使用循环展开的方式，将 Micro-kernel 中对 k_c 维度的遍历循环展开，一次遍历四次。通过循环展开我们可以减少分支预测的次数减少分支预测机制所带来的耗时罚款，延长了计算流水线的长度。循环展开是一种以代码量换取性能的方式，相对于普通的循环增加了处理尾循环的部分，通常循环部分代码量会膨胀展开倍数。

2. 数据预取：数据预取可以让数据在使用之前做好存取准备，ARM 架构提供了高级的数据预取指令，使用“PRFM”指令可以对编译器做出建议使它加载提前加载数据到制定的内存层次，“PRFM”指令支持三种模式，对数据为了加载或者存储而预先提取，或者是预先加载指令。另外支持预取数据到三级缓存中，在这里我们使用“prfm pldl1keep, [src]”来建议 CPU 将 src 存储的地址里的数据进行预取到 L1 缓存中，我们在 Micro-kernel 循环内部进行预取以及在每次汇编内核即将结束时候对下一次传入汇编内核中的数据首地址进行预取进行数据预取优化。

3. 向量优化：通过普通的向量化技术我们可以将算法的效率大大增加，比如对于内核中 16×8 外积操作，通过将左乘矩阵的一列 16 个元素乘以右乘矩阵的一行 8 个元素得到一个 16×8 的矩阵，如果使用标量来计算的话那么一次外积总共将乘加 128 次，而我们选择使用两个支持可变向量长拓展指令集的向量寄存器保存左乘矩阵的一列，使用 8 个向量寄存器将右乘矩阵的一行广播保存，那么只需要 16 次向量乘加计就可以完成标量寄存器原先 128 次的计算，在外积计算上提供了 8 倍的加速，里面还未计算对于向量 1 次存取 8 个元素的提升速度

4. 指令并行及流水设计：国产 ARM 架构 CPU 是支持乱序执行超标量处理器，拥有两个 SIMD 浮点处理元件，同一时刻可以做两个浮点乘加运算。在我们的汇编内核中，有 k_c 次的迭代次数，每一次迭代中我们需要将左乘矩阵的两个支持可变向量长拓展指令集的向量寄存器以此与八个右乘矩阵的向量寄存器进行相乘，累加到 16 个临时矩阵的支持可变向量长拓展指令集的向量寄存器中，16 个用于存储结果的临时矩阵向量寄存器除去每次作为运算的结果寄存器之外，只有在所有运算之前的一次置零操作和运算结束的存回操作一次数据传输的操作。这两次操作都是 16 的倍数级别时间消耗但是都是不可摊销的时间消耗。而在计算过程

```

1      " S256LOOP:                                \n\t" // Body
2      "
3      " fmla z12.s, p0/m, z0.s, z2.s          \n\t"
4      " prfm PLDL1KEEP, [x1, #224]            \n\t"
5      " fmla z13.s, p0/m, z1.s, z2.s          \n\t"
6      " prfm PLDL1KEEP, [x1, #288]            \n\t"
7      " ld1rw  z2.s, p0/z, [x1]                \n\t"
8      "
9      " fmla z14.s, p0/m, z0.s, z3.s          \n\t"
10     " fmla z15.s, p0/m, z1.s, z3.s          \n\t"
11     " ld1rw  z3.s, p0/z, [x1, #4]           \n\t"
12     "
13     " fmla z16.s, p0/m, z0.s, z4.s          \n\t"
14     " fmla z17.s, p0/m, z1.s, z4.s          \n\t"
15     " ld1rw  z4.s, p0/z, [x1, #8]           \n\t"
16     "
17     " fmla z18.s, p0/m, z0.s, z5.s          \n\t"
18     " fmla z19.s, p0/m, z1.s, z5.s          \n\t"
19     " ld1rw  z5.s, p0/z, [x1, #12]           \n\t"
20     "
21     " fmla z20.s, p0/m, z0.s, z6.s          \n\t"
22     " fmla z21.s, p0/m, z1.s, z6.s          \n\t"
23     " ld1rw  z6.s, p0/z, [x1, #16]           \n\t"
24     "
25     " fmla z22.s, p0/m, z0.s, z7.s          \n\t"
26     " prfm PLDL1KEEP, [x0, #448]             \n\t" // 448 + 64 -64
27     " fmla z23.s, p0/m, z1.s, z7.s          \n\t"
28     " prfm PLDL1KEEP, [x0, #512]             \n\t"
29     " ld1rw  z7.s, p0/z, [x1, #20]           \n\t"
30     "
31     " fmla z24.s, p0/m, z0.s, z8.s          \n\t"
32     " prfm PLDL1KEEP, [x0, #576]             \n\t"
33     " fmla z25.s, p0/m, z1.s, z8.s          \n\t"
34     " prfm PLDL1KEEP, [x0, #640]             \n\t"
35     " ld1rw  z8.s, p0/z, [x1, #24]           \n\t"
36     "
37     " fmla z26.s, p0/m, z0.s, z9.s          \n\t"
38     " fmla z27.s, p0/m, z1.s, z9.s          \n\t"
39     " ld1rw  z9.s, p0/z, [x1, #28]           \n\t"
40     "
41     " ld1w   z0.s, p0/z, [x0]                \n\t"
42     " ldiw   z1.s, p0/z, [x0, #1, MUL VL]    \n\t"
43

```

图 4.15 核心计算汇编代码示意图

中，我们在 k_c 次迭代中间的每一次迭代都涉及到了 16 次的乘加操作，对左乘矩阵有两次的数据加载操作，对右乘矩阵有八次的数据加载操作。这里十个数据传输的时间消耗可以用两个方式摊销：一种是每次迭代中先完成八个右乘矩阵向量寄存器的广播加载和一个左乘矩阵向量寄存器的加载，然后计算八次乘加，再更新左乘矩阵的向量寄存器，再进行八次乘加的方式；另一种是先加载两个左乘矩阵的向量寄存器和一个右乘矩阵向量寄存器的广播加载，然后以两个乘加操作加一个右乘矩阵一个元素的广播加载为节拍循环四次。只需要考虑时间摊销的价值和 SIMD 浮点处理元件的个数，我们就可以很容易的选择第二种方式，在第二种

方式中，我们可以达到最大程度指令流水，在迭代中将数据传输的时间开销隐藏在计算的时间开销中，只需在进入循环时候承受数据传输开销。而在第一种选择中，则需多承担五次时间开销，内联汇编书写的内核中循环展开计算的第一部分代码如图4.15所示。

4.2.5 多线程优化

通常在许多循环存在的地方，就存在着很大的并行潜力，在矩阵乘中由于分块的原因创造了很多的循环，将原先的三层循环就可以完成的朴素矩阵乘变成了庞大的六层循环（包含汇编内部对 k_c 维度的迭代）。而具有非常大的并行潜力，这里我们参考了 BLIS 提出的对矩阵乘的分析方法^[52]以及 BLIS 的循环图^[51]中循环逻辑自上而下的分析，针对国产 ARM 架构 CPU 的硬件特征找寻最适合多线程优化的方案。

1. 第五层循环：这趟循环是最开始进行的循环，对右乘矩阵 B 的行维度 N 上进行了 n_c 的分割，分割成多个小矩阵块，在这种情况下如果想要使用多线程加速，我们会考虑将左乘矩阵被所有线程共享，每个线程单独持有保存右乘矩阵的一行中一个小块。但是这种并行方式比较特殊，适合 NUMA (non-uniform memory access, 非统一内存访问) 架构，对每个非统一内存访问访问节点都将它的一小块 B 放置在自己独立占据的三级缓存中，因为 n_c 的选取较大，并行度很难高起来。

2. 第四层循环：第四层循环对 K 维度进行了 k_c 大小的拆分，由于结果矩阵 C 的 x 行 y 列的值是由左乘矩阵 A 的 x 行与右乘矩阵 B 的 y 列对应元素的乘积的总和，一共有 K 个元素求和，而如果不满 K 个元素，我们认为它是结果矩阵值的一层皮，最终所有的皮求和才是结果值。这个循环就是迭代计算皮的过程，并将皮进行累加。针对这个循环如果进行遍历，因为存在着所有线程对结果矩阵 C 的更新，将会涉及到线程的数据竞争问题，为了解决数据竞争可能导致的结果不正确问题，我们需要利用使用锁机制来让每个线程可以在更新结果矩阵的时候拥有绝对的独享权。然而一般需要引入锁机制的多线程程序通常性能会打折扣，并行第四层循环，只有在 M、N 维度较小的情况下，或者是最后更新的开销远小于计算开销的时候才可取，由于考虑到 K 维度比较低时候并行效率将很低以及求和更新时候的互斥情况带来的效率降低不可避免，将抛弃在次循环上并行。

3. 第三层循环：第三层循环是对左乘矩阵 A 矩阵进行分块的循环，在这趟循环中将 A 矩阵划分为 block，划分以 m_c 为大小划分 M 维度。目的是将 A 的小矩阵 block 能够存放的进 2 级缓存中，这种条件下并行，每个线程独占 A 的一个小矩阵块，放置在各自的 2 级缓存中，而 B 的 pannel 将放置在 3 级缓存中，这种情况下不会出现线程的数据竞争问题，因为不同线程计算的是结果矩阵 C 的不同行。这种方式需要考虑的是矩阵乘的维度 M 的大小，当 M 太小导致并行度小于

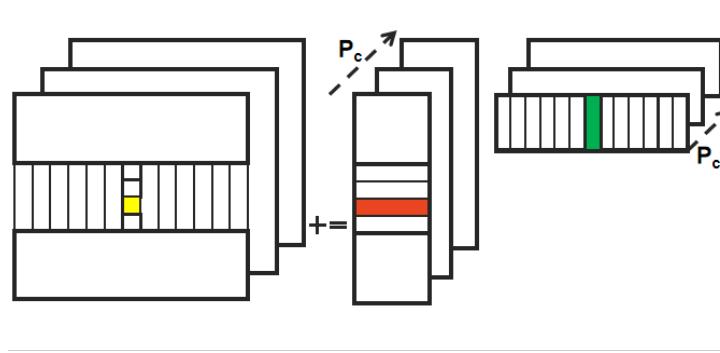


图 4.16 分块循环第四次循环示意图

可使用的线程总数时将影响最终的并行效果。

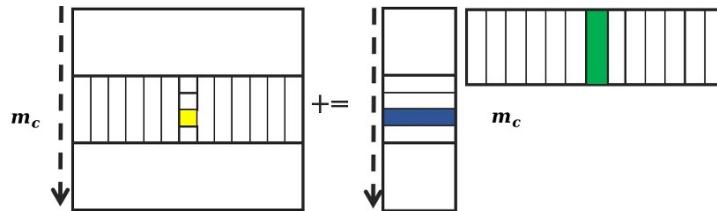


图 4.17 分块循环第三次循环示意图

4. 第二层循环：第二层循环对 n_c 维度进行 n_r 维度的划分，对右乘矩阵 B 的 pannel 进行分割，分割成一个 $k_c \times n_r$ 的裂片在与左乘矩阵 A 的 block 进行计算。这一层循环有 $n_c \div n_r$ 次迭代，在我们的计算中远大于国产 ARM 架构 CPU 实际线程数 64 的倍数，在这一步骤中进行多线程的话如果矩阵乘 n 维度大于 n_c 就一定能够调用所有的线程进行计算，但是在这一层并行需要将左乘矩阵 A 的 block 放置到不同核的 2 级缓存中，需要考虑这部分是否会因为缓存一致性问题影响时间开销。

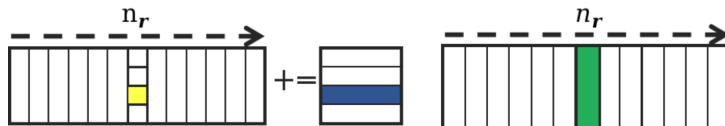


图 4.18 分块循环第二次循环示意图

5. 第一层循环：类似第二层循环，对 m_c 维度进行 m_r 维度的划分，对左乘矩阵 A 的 block 进行行分割，分割成一个 $m_r \times k_c$ 的裂片在与右乘矩阵 B 的裂片进行计算，在这层循环中拥有 $m_c \div m_r$ 次迭代，在我们的分块和内核设计中，这个值大约为 9-10 之间，远小于国产 ARM 架构 CPU 拥有的实体线程数量，可以直接不用考虑在这层循环上进行并行。

6. 寄存器级循环：由于寄存器中做的是 $m_r \times n_r$ 的计算量，计算量较小，并且内核做的是外积，需要迭代累加，同样需要控制数据竞争问题，综合考虑可以直接不考虑并行。

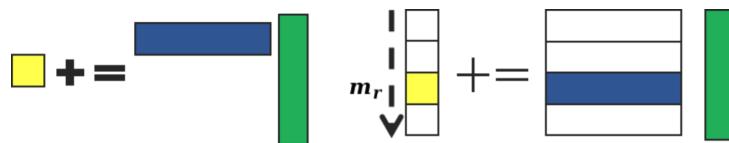


图 4.19 分块循环第一次循环示意图

在综合对所有存在循环的部分进行分析考虑，最终发现只有第三层循环和第二层循环适合进行多线程优化，最终我们在 FTEngine 中实现的高性能矩阵乘函数选择在第三层循环使用 OpenMP 进行多线程优化。

4.3 本章小节

本章介绍了 FTEngine 中优化程度最高的网络层——卷积层，在本章，我们首先介绍了 FTEngine 中对 Im2col 算法的改进后的设计，并介绍了改进后的转换参数的获取方法，以及设计使用可变长向量拓展指令集优化 Im2col 拷贝转换内核的设计。然后介绍了我们在 FTEngine 中优化实现的高性能矩阵乘，介绍了我们针对国产 ARM 架构 CPU 处理器设计的分块方法和分块大小，以及我们在国产 ARM 架构 CPU 上使用可变长拓展指令集实现的向量化内核，最后我们还探讨了我们实现的高性能矩阵乘的并行方案。

第 5 章 实验与结果分析

本章将分为四个部分展开，首先我们测试国产 ARM 架构 CPU 上使用可变长向量拓展指令集实现的高度优化矩阵乘，通过贪心的方式进行遍历找到实际最优的分块参数，并且在性能上对比了几款常见开源线性代数函数库的矩阵乘法实现；然后我们使用了两个知名的深度卷积神经网络模型 AlexNet 和 VGGNet 作为参考，首先我们对 AlexNet 网络和 VGGNet 的网络结构进行分析，并根据网络的结构提取主要的网络层的输入输出参数；然后我们对 AlexNet 和 VGGNet 中使用的卷积层和池化层的常见参数进行单独性能测试；最后我们再使用 TEngine 搭建 AlexNet 和 VGGNet 与进行单线程和多线程情况下的性能测试，分析测试结果。

表 5.1 本章节测试的环境以及对比软件和工具的详细配置

配置选项	配置内容
测试平台：	国产 ARM 架构 CPU 处理器
测试使用软件：	FTEngine、OpenBLAS-0.3.23、BLIS-0.9.0、ACL-22.08、oneDNN3.0
编译器版本：	gcc-11.1.0、cmake-3.10.2、make-4.2.1

5.1 国产 ARM 架构 CPU 单核峰值性能实验

首先我们分析国产 ARM 架构 CPU 上的峰值性能，参考单核理论峰值性能的计算方法同式5.1。

$$\text{单核峰值性能} = \text{单核频率} \times \frac{\text{CPU 向量位宽}}{\text{数据类型位数}} \times 2 \quad (5.1)$$

根据现有的国产 ARM 架构 CPU 信息，我们可以总结国产 ARM 架构 CPU 的单核峰值性能为：

$$2.5 \times 10^9 \frac{\text{cycles}}{\text{second}} \times 16 \frac{\text{FLOPs}}{\text{cycle}} \times 2 = 80\text{GFLOPs/ second} \quad (5.2)$$

2.5×10^9 代表单核核心工作频率； $16 \frac{\text{FLOPs}}{\text{cycle}}$ 由 SIMD 发射数与 SIMD 向量寄存器支持的最高浮点数据个数相乘得出；需要乘 2 是因为国产 ARM 架构 CPU 支持混合乘加指令，每个混合乘加指令可以在一个周期内完成一个乘法操作和一个加法操作可以充当两次浮点操作。

为了验证我们的计算公式，我们同时手动实现了 CPU 性能测试程序，我们采

用内联汇编写一个循环程序，循环外部初始化 8 个预测寄存器和 16 个向量寄存器，然后铺满 16 个连续的向量乘加，并使用不同的向量寄存器作为结果寄存器避免数据竞争，再将向量乘加复制到五次，总共在循环中进行 80 次向量乘加计算。在循环外部使用计时函数掐时间最终可以得到运行时间。计算核心部分如图 5.1

```

1  | "fmla z0.s, p0/m, z20.s, z21.s      \n\t"
2  | "fmla z1.s, p0/m, z20.s, z21.s      \n\t"
3  | "fmla z2.s, p0/m, z20.s, z21.s      \n\t"
4  | "fmla z3.s, p0/m, z20.s, z21.s      \n\t"
5  | "fmla z4.s, p0/m, z20.s, z21.s      \n\t"
6  | "fmla z5.s, p0/m, z20.s, z21.s      \n\t"
7  | "fmla z6.s, p0/m, z20.s, z21.s      \n\t"
8  | "fmla z7.s, p0/m, z20.s, z21.s      \n\t"
9  | "fmla z8.s, p0/m, z20.s, z21.s      \n\t"
10 | "fmla z9.s, p0/m, z20.s, z21.s      \n\t"
11 | "fmla z10.s, p0/m, z20.s, z21.s     \n\t"
12 | "fmla z11.s, p0/m, z20.s, z21.s     \n\t"
13 | "fmla z12.s, p0/m, z20.s, z21.s     \n\t"
14 | "fmla z13.s, p0/m, z20.s, z21.s     \n\t"
15 | "fmla z14.s, p0/m, z20.s, z21.s     \n\t"
16 | "fmla z15.s, p0/m, z20.s, z21.s     \n\t"

```

图 5.1 峰值性能测试代码核心部分示意图

通过公式：

$$\text{实际峰值性能} = \frac{2 \times 16 \times 5 \times \text{循环次数}}{\text{运行时间} \times 10^9} \quad (5.3)$$

我们可以得到最终实际计算的峰值性能，输入循环次数 100000 次，经过多次计算，结果大概在 79.89 左右，考虑到损耗因素，性能与计算峰值性能相差无几。

5.2 矩阵乘性能比较实验

我们实现的高性能矩阵乘与朴素版矩阵乘的精度测试比较误差可以保证在 10^{-4} 以内，基本满足深度学习的要求。然后我们将我们的高性能单精度矩阵乘法对比 OpenBLAS 和 BLIS 的最新版本发行版；我们下载了 OpenBLAS-0.3.23 和 BLIS-0.9.0 的源码，并传输到国产 ARM 架构 CPU 服务器上，针对 OpenBLAS，可以指定编译的架构选项，通过“TARGET=ARCH”指定针对架构进行优化，FT2000 已在 OpenBLAS 中进行单独优化，另外发现支持部分支持可变长向量拓展指令集的架构的特殊优化，比如 A64FX 处理器，CORTEX-A710 处理器，这两款机器的缓存架构与国产 ARM 架构 CPU 比较接近，主要体现在这两款处理器的二级缓存为 1MB，而国产 ARM 架构 CPU 处理器的二级缓存持有 2MB 的大小，但目前 OpenBLAS 支持的架构来看，这两款处理器是最接近国产 ARM 架构 CPU 的选择，最终我们选择 OpenBLAS 编译 A64FX 的版本进行对比。而 BLIS 支持自动判断架构编译，对于 BLIS-0.9.0 版本，支持“arm_sve”的架构策略，我们也选择

BLIS-0.9.0 对 arm_sve 进行编译与 FTEngine 的矩阵乘进行测试。

因为 BLIS 和 OpenBLAS 的单精度矩阵乘函数名相同，放在同一个文件中会相互冲突，所以对于 BLIS 和 OpenBLAS 使用相同的测试程序通过指定不同的编译头文件，链接不同的库文件生成不同的执行文件。

同时使用 shell 命令书写脚本文件，循环执行四种高性能单精度矩阵乘，并测试出运行时间。对于计算数据本文是如此设计的：矩阵乘的 M、N、K 值相同并从 512 到 1024，然后依次递增 1024 直到 8192 大小的规模大小取值，对于左乘矩阵 A 和右乘矩阵 B 进行随机初始化，对于结果矩阵 C 进行相同常量初始化。最终在脚本中轮流执行得到运行时间。

通过计算公式5.4计算得到最终计算性能。

$$\text{实际矩阵乘性能} = \frac{(2 \times M \times N \times K)}{\text{运行时间} \times 10^9} \quad (5.4)$$

如图5.2是我们实现的高性能矩阵乘对比 OpenBLAS 编译 A64FX 库和 OpenBLAS 编译 arm 库下的运行性能对比。

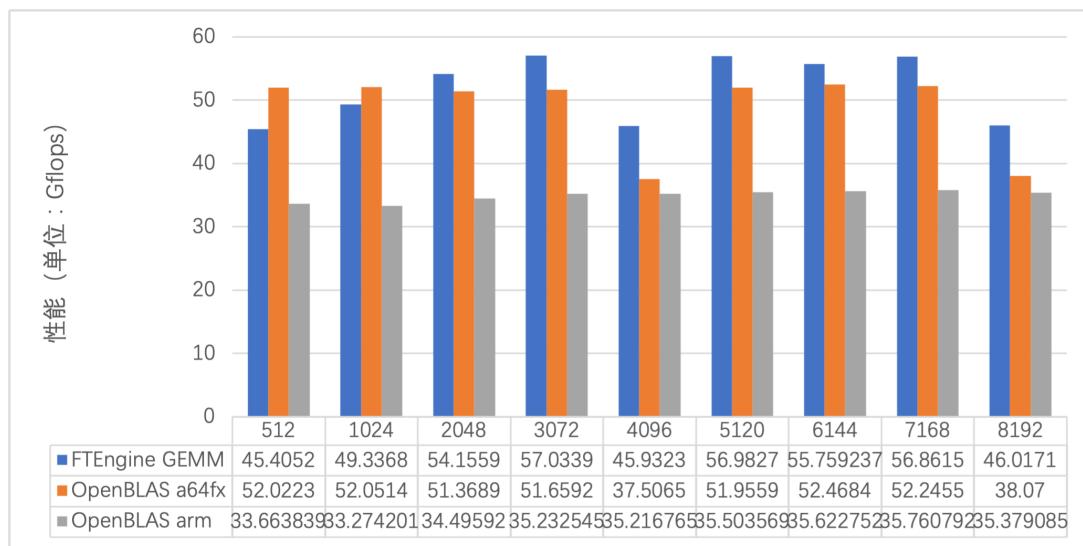


图 5.2 FTEngine GEMM 与 OpenBLAS GEMM 的性能对比图

如图5.3是我们实现的高性能矩阵乘对比 BLIS 下的运行性能对比。

综合来看，在矩阵规模超过 $2048 \times 2048 \times 2048$ 之后，我们实现的矩阵乘性能比 OpenBLAS 两个编译版本还有 BLIS 都要更快。通过比较 OpenBLAS 中针对 A64FX 参数编译和针对 ARM 通用版本的编译，也可以发现针对 A64FX 参数编译比针对 ARM 通用版本编译的性能要优秀很多，这表明了针对特定架构的指令集优化以及特定架构下矩阵乘分块大小的设定所带来的作用。而 BLIS 性能要处于最低的情况，可能因为 BLIS 的默认分块参数将分块分的太小，未能完全填满二级三级缓存空间，无法发挥出国产 ARM 架构 CPU 的全部性能。另外，在维度为

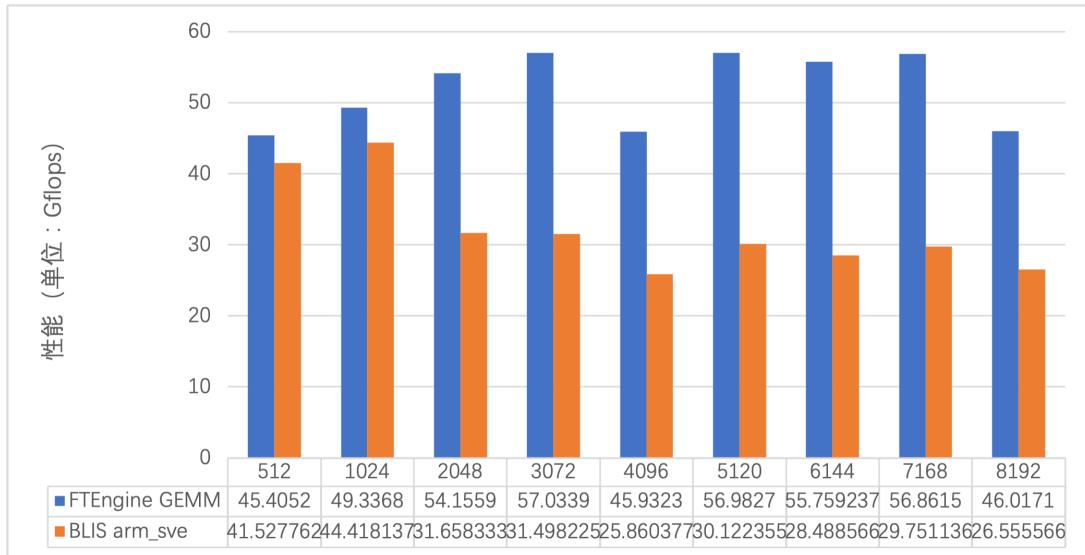


图 5.3 FTEngine GEMM 与 BLIS GEMM 的性能对比图

4096 大小时候，性能出现了骤降，猜测是因为刚好跨过缓存界限，有小部分数据需要从内存中取出，运算取数的时间比较低，造成了性能的损伤，最终我们得到我们实现的高性能矩阵乘函数的性能比 BLIS 的矩阵乘函数的性能平均快 76% 的。比 ARM 通用编译版本的性能平均快 52%，比 A64FX 编译版本平均快 7 %。

5.3 FTEngine 卷积层和池化层性能比较实验

FTEngine 的数据的误差在开发阶段便一直都能控制在 10^{-3} 到 10^{-5} (根据网络层的不同) 的范围内，所以无需顾虑精确度的问题。但是为了对 FTEngine 进行性能上的测试比较，检验我们在 FTEngine 中对深度学习网络层函数优化的效果，我们挑选了 AlexNet 和 VGGNet 作为我们的测试标准，我们认为实际运行时候的算子的运行参数更能代表算子的真正性能表现，我们提取了 AlexNet 和 VGG16 中卷积操作和池化操作的具体参数，在本小节对我们优化程度最高的卷积层函数和池化层函数进行测试。

AlexNet 中池化层的具体参数如表5.2，AlexNet 中卷积层的具体参数如表5.3。

表 5.2 AlexNet 中的池化层计算参数

输入尺寸	输入通道	内核尺寸	跨度	填充	输出尺寸
55	96	3	2	0	27
27	256	3	2	0	13
13	256	3	2	0	6

VGG16 中池化层的具体参数如表5.4，VGG16 中卷积层的具体参数如表5.5。

表 5.3 AlexNet 中的卷积层计算参数

输入尺寸	输入通道	内核尺寸	输出通道	跨度	填充	输出尺寸
227	11	3	96	4	0	55
27	5	96	256	1	2	27
13	3	256	384	1	1	13
13	3	384	384	1	1	13
13	3	384	256	1	1	1

表 5.4 VGG16 中的池化层计算参数

输入尺寸	输入通道	内核尺寸	跨度	填充	输出尺寸
224	64	2	2	0	112
112	128	2	2	0	56
56	256	2	2	0	28
28	512	2	2	0	14
14	512	2	2	0	7

表 5.5 VGG16 中的卷积层计算参数

输入尺寸	输入通道	内核尺寸	输出通道	跨度	填充	输出尺寸
224	3	3	64	1	1	224
224	3	64	64	1	1	224
112	3	64	128	1	1	112
112	3	128	128	1	1	112
56	3	128	256	1	1	56
56	3	256	256	1	1	56
28	3	256	512	1	1	28
28	3	512	512	1	1	28
14	3	512	512	1	1	14

对于池化层的参数统计表，提供了输入尺寸、输入通道、内核尺寸、跨度、填充、输出尺寸等信息，其中输入尺寸代表了输入图像的宽和高，默认相同所以只提供一个值，内核尺寸表明的是池化窗口的尺寸，高和宽相同；而对于卷积层的参数统计，内核尺寸代表的是卷积核窗口的大小，并增加了输出通道的参数。

在 VGG 网络层，有多个网络层卷积中的尺寸相同，在表中没有重复列举，另外我们对 VGG 的网络进行细致观察，发现 VGG 的网络结构中，池化层和卷积层的内核尺寸、跨度、填充的值都是相同的，所以对于 VGG 网络的计算影响参数只有输入通道和输入尺寸。

在获得了 AlexNet 和 VGG16 的卷积层具体参数后，我们对卷积层函数进行性能单测，我们对比 ACL22.08 版本的性能，ACL 并没有直接对网络层进行测试的函数，oneDNN 中可以通过修改 cmake 并增加编译选项链接使用 ACL，在卷积层和池化层支持使用 ACL 算子函数运行，我们使用 AlexNet 和 VGG16 中提取的卷积和池化函数的参数配置来将 FT Engine 和 ACL 进行对比，测试在国产 ARM 架构 CPU 上 FT Engine 的卷积函数和池化函数的具体性能效果。

安装 oneDNN3.0 版本后我们链接 ACL 需要修改 cmake 目录下的 Find.cmake 文件，修改查找库的路径，之后将 ACL 的安装库路径 export 在环境变量上，在编译时候指定 cmake 参数 DNNL_AARCH64_USE_ACL=ON，如此之后可以编译使用 ACL 作为执行函数的 oneDNN。为了减少变量，我们设置 oneDNN 的 CPU 运行时多线程为 OpenMP，与 FT Engine 保持一致。

我们将卷积层函数和池化层函数的要测试的参数组使用二维数组打包，对于图片数目在卷积函数测试中我们统一设定为 3，在池化函数测试中我们统一设定为 10。用循环控制依次对不同参数测试，循环内部对函数执行前后进行时间掐表，循环执行 100 次累加所有执行时间，最后对执行时间取平均值并输出到终端。

图5.4表明了我们在 AlexNet 包含的卷积层参数下 FT Engine 和 ACL 之间的性能比较：

图5.5表明了我们在 VGG16 包含的卷积层参数下 FT Engine 和 ACL 之间的性能比较：

图5.6表明了我们在 AlexNet 包含的池化层参数下 FT Engine 和 ACL 之间的性能比较：

图5.7表明了我们在 VGG16 包含的池化层参数下 FT Engine 和 ACL 之间的性能比较：

通过实验可以发现我们实现的 AI 算子可以以微弱的优势高于 ACL 的实现，比较原因应该是尽管国产 ARM 架构 CPU 的向量长度是 NEON 向量长度的两倍，但是因为程序中逻辑阶段同样占据很多时间，并且 ACL 开源库本身是个性能很不错的开源库，所以我们仅能借着向量长的优势微弱高过 ACL 的执行速度。最

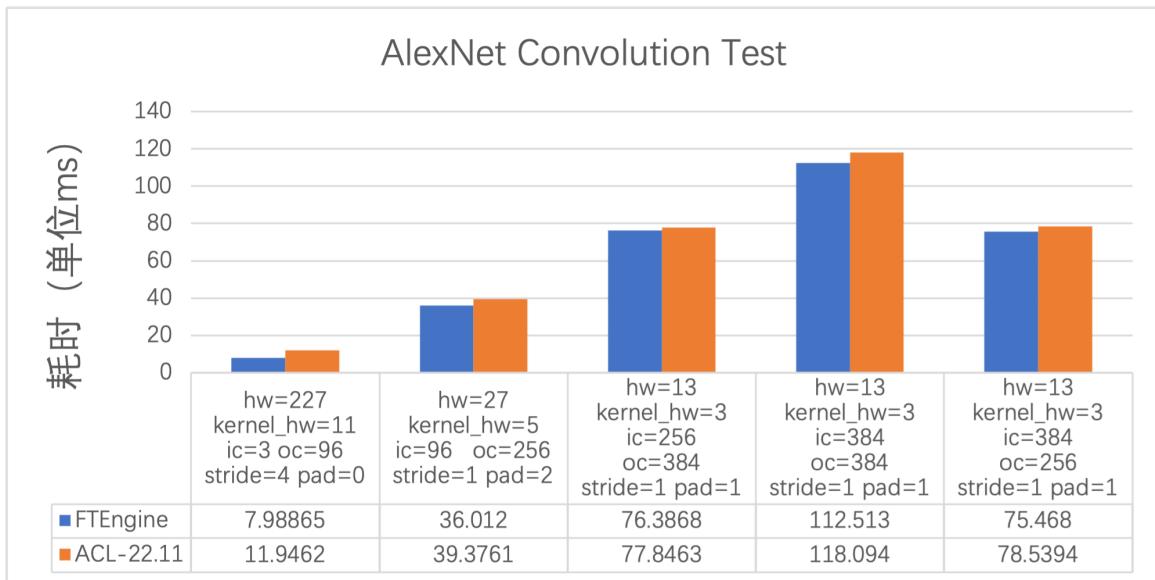


图 5.4 AlexNet 卷积参数下 FTEngine 与 ACL 的卷积性能对比图

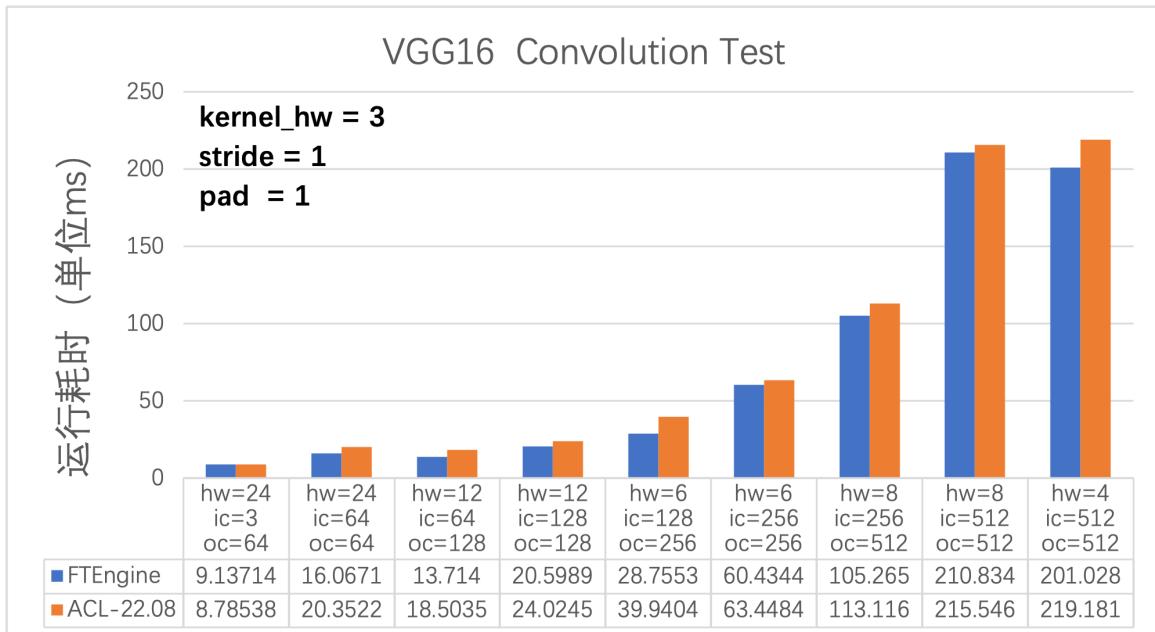


图 5.5 VGG16 卷积参数下 FTEngine 与 ACL 的卷积性能对比图

终我们可以发现在 AlexNet 的卷积参数在 FTEngine 运行速度比 ACL 快 13%，在 VGG16 的卷积层参数下 FTEngine 比 ACL 快 15%。在 AlexNet 的池化层参数中，速度平均提升了 38%，而在 VGG16 的参数中速度平均提升了 11%。

5.4 模型运行对比实验

经过对卷积层和池化层进行实际参数的性能对比测试之后，我们可以看出 FTEngine 性能略优于 ACL 的性能表现，为了比较实际模型的运行能力，我们又对比了 AlexNet、VGG16 两款经典模型的性能数据，我们使用 FTEngine 搭建了简

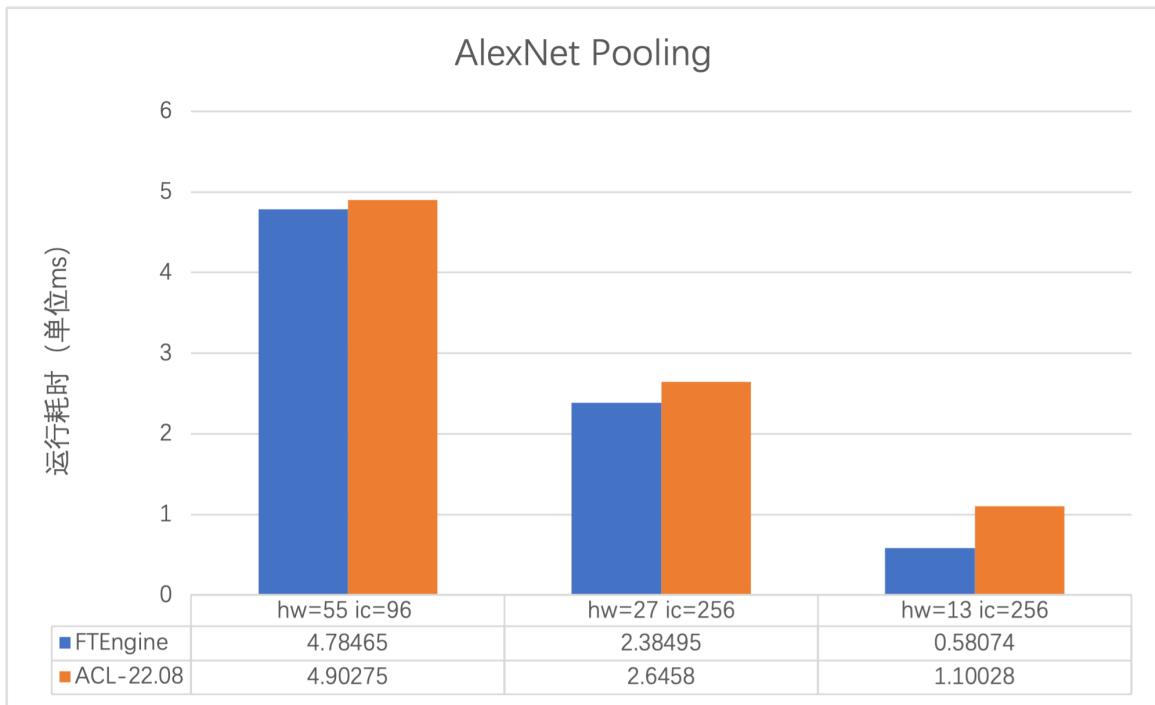


图 5.6 AlexNet 卷积参数下 FTEngine 与 ACL 的池化性能对比图

单的 AlexNet、VGG16、两种卷积神经网络模型，而 ACL 自带了 AlexNet、VGG16、两种网络模型的样例，经过修改加入相同的计时函数并测试了单线程、16 线程、64 线程下的运行耗时。我们使用随机生成的数据进行测试，每次运行的 batchsize 为 1，每个情况测试五次，并记录五次数据结果。并在最终取的平均值。最终我们记录数据并绘制图：

5.5 本章小结

本章节首先测试了国产 ARM 架构 CPU 的峰值性能，通过执行使用大量的混合乘加指令并统计时间计算得到国产 ARM 架构 CPU 每秒最多可以执行 80G 个浮点操作指令；

然后我们使用两种知名的开源线性代数库 OpenBLAS、BLIS，对第四章实现的高性能矩阵乘函数进行比较测试，对 OpenBLAS 尝试了两种编译版本，一种是 ARM 通用版本，一种是针对同样拥有可变长向量指令集的 A64FX 处理器进行深度优化的版本。对 BLIS 编译了支持使用可变长向量指令集的 arm_sve 编译版本。然后我们将相同的测试程序复制三份，针对三种矩阵乘函数实现进行测试，从命令行传入矩阵的维度参数，程序内部对矩阵数据进行随机初始化，我们对三种矩阵乘的维度从 1024 逐次累加 1024 到 8192 大小，记录每次运行时间并通过性能求解公式得到每次的计算性能。最终比较三种高性能矩阵乘的测试结果。我们的矩阵乘在所有的维度上都性能优于 BLIS 的矩阵乘实现和 OpenBLAS 针对 ARM 架

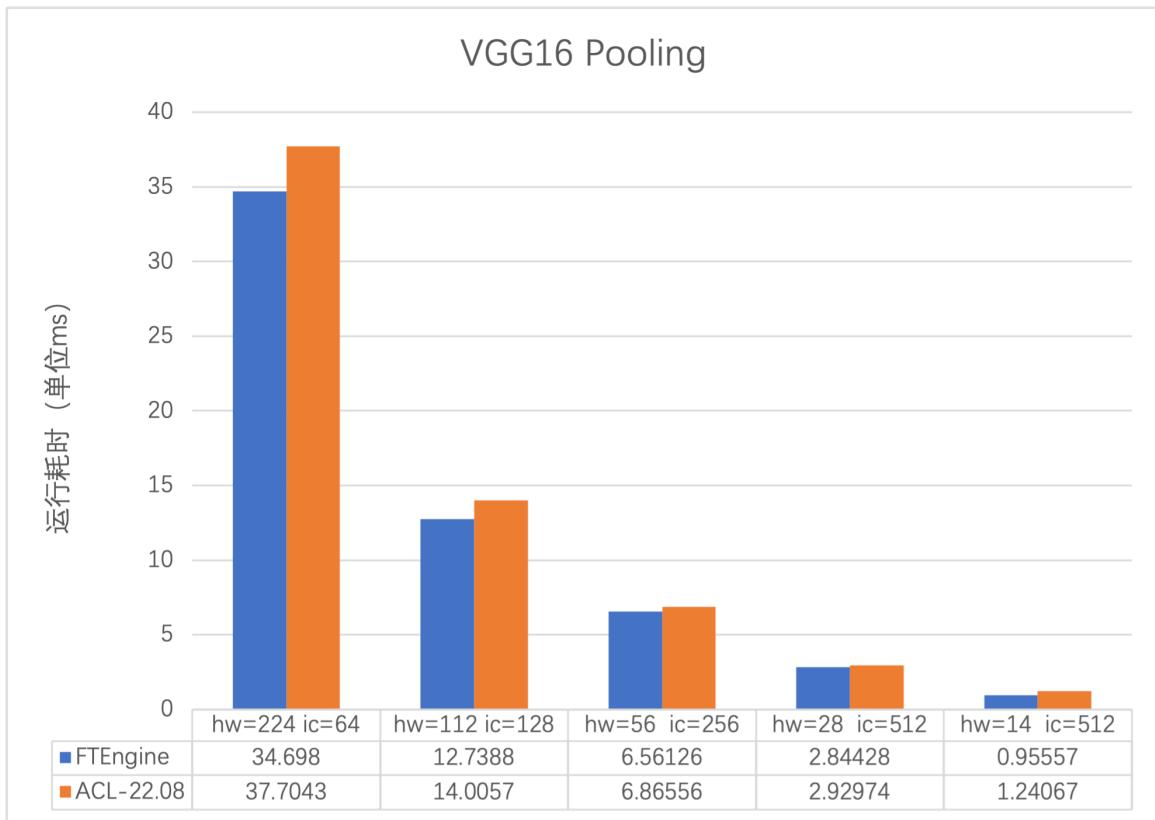


图 5.7 VGG16 卷积参数下 FTEngine 与 ACL 的池化性能对比图

构的通用实现，在低维度上略低于 OpenBLAS 对与我们国产 ARM 架构 CPU 架构参数相似 A64FX 编译的版本，在大于 $2048 \times 2048 \times 2048$ 维度及以上时，我们实现的高性能矩阵乘性能高于 OpenBLAS 针对 A64FX 的编译版本。最终我们得到我们实现的高性能矩阵乘函数的性能比 BLIS 的矩阵乘函数的性能平均快 76% 的。比 ARM 通用编译版本的性能平均快 52%，比 A64FX 编译版本平均快 7%。

接着，我们认为使用网络模型中网络层的参数更能体现 FTEngine 实际运行时候的性能，所以我们记录了 AlexNet 和 VGG16 中的卷积层参数和池化层参数，并使用记录的参数对比了 FTEngine 和 ACL22.08 的执行时间，比较表明在 AlexNet 的卷积参数在 FTEngine 运行速度比 ACL 快 13%，在 VGG16 的卷积层参数下 FTEngine 比 ACL 快 15%。在 AlexNet 的池化层参数中，速度平均提升了 38%，而在 VGG16 的参数中速度平均提升了 11%，不过研究测试结果发现，因为我们是针对输入通道维度做的向量化优化，在输入通道维度较高，其他维度较低的时候提升更加明显。

最后，我们使用 FTEngine 和 ACL 运行比较 AlexNet 和 VGG16 两个网络模型，在单线程、16 线程、64 线程时不同并行程度时候的性能表现。每个情况运行 5 遍，最终求取平均并进行比较。结果表明在三种并行方案下 FTEngine 的平均性能均优于 ACL 的实现情况，不过同时发现 FTEngine 中的并行部分并不成熟，在 64 线程的情况下，平均性能要低于 16 线程的情况，表明了任务分配和线程使用仍

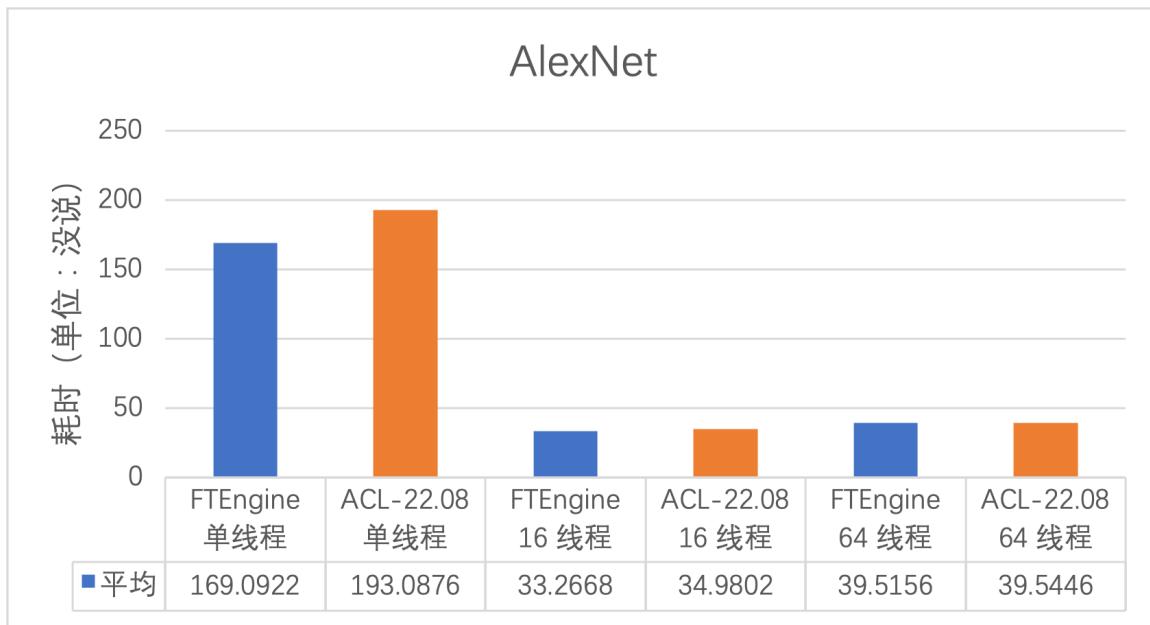


图 5.8 FTEngine 与 ACL 运行 AlexNet 的性能对比图

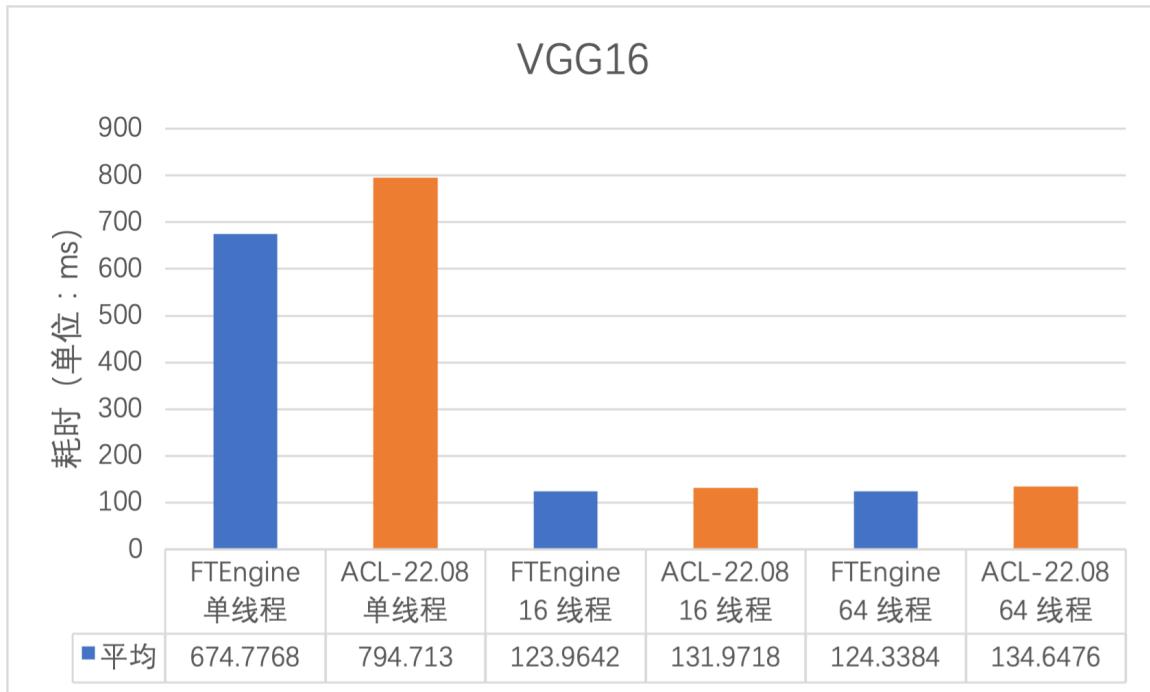


图 5.9 FTEngine 与 ACL 运行 VGG16 的性能对比图

有提升空间。最终我们记录在单线程情况下 FTEngine 执行 AlexNet 网络模型比 ACL22.08 快 14 %，在 16 线程情况执行 AlexNet 网络模型 FTEngine 要比 ACL22.08 快 5 %，而在单线程情况下 FTEngine 执行 VGG16 网络模型比 ACL22.08 快 17 %，在 16 线程情况执行 AlexNet 网络模型 FTEngine 要比 ACL22.08 快 6 %。由于 64 线程的耗时不稳定，所以我们不做比较。

总结与展望

伴随着国家的繁荣昌盛，我们国家在基本建设上已经基本达到国际前列水平了，与此同时我们国家开启了在芯片领域的追逐，发起了一次一次的尝试。然而，一次成功的芯片的诞生离不开运作在它之上的软件。国产 ARM 架构 CPU 处理器是我们国家在 ARM 架构上的新的一次尝试，它包含着成为新一代超级计算机核心的一切要素，我们需要给予它一个适合运作在它之上的软件。深度学习人工智能领域是目前各国技术竞争的一个新战场，对于国产 ARM 架构 CPU 处理器，目前市面上存在的诸如 OpenBLAS、BLIS、Caffe、oneDNN 以及国产的神经网络框架都没有针对国产 ARM 架构 CPU 处理器的向量支持和缓存大小设计实现高性能的深度学习函数内核。所以本文结合体系结构优化技术和深度学习的基本理论，为国产 ARM 架构 CPU 设计实现了高性能深度学习函数库 FTEngine。在 FTEngine 中根据 Im2col+GEMM 方法实现卷积函数，并针对传输转换特征优化了 Im2col 转换函数，并根据现有开源 GEMM 的优化理论优化实现了支持可变长向量指令集的高性能矩阵乘函数。对于其他深度学习函数，FTEngine 中根据具体网络层函数的计算特征进行相应的实现和不同层级的优化，最终实现高性能深度学习函数库 FTEngine。本文的具体工作如下：

1. 本文首先介绍了人工智能神经网络的发展历史，介绍了人工智能的应用以及当代对于算力的追逐和国内目前为了发展计算科技的各种尝试。并介绍了目前在深度学习人工智能领域主流的深度学习计算库和一些新兴的人工智能编译库，以及基于体系结构进行程序优化的基本方法，最后
2. 目前深度学习卷积神经网络的应用最为广泛，也是最火热的研究方向；现存的神经网络函数库没有针对 ARM 架构 CPU 进行深度的优化，所以本文分析了 LeNet、AlexNet、VGGNet、Inception Net、ResNet 五种典型的网络模型结构，总结了需要运行神经网络所必须的网络层最少包含卷积层、激活层、池化层、归一化层、全连接层五种深度学习网络层，并介绍了各个层的作用和原理。
3. 深度学习的软件栈主要从上到下分为应用层、支持层、硬件层三层，我们实现了链接软件层算法人员应用程序与硬件层国产 ARM 架构 CPU 间的支持层函数库——FTEngine。具体的目前实现了卷积神经网络所需要的卷积层、激活层、池化层、归一化层以及全连接层。
4. 针对池化层、全连接层、激活层、归一化层四个网络层，我们针对不同网络的层的计算逻辑采用了不同程度和不同方法的优化手段。针对池化层的计算逻辑，我们将池化层计算抽象成三层模式，分为针对输入特征进行处理的接口层，计算池化窗口的驱动层，以及使用汇编编写使用可变长向量拓展指令集进行向量

优化的内核层；对于全连接层函数，针对计算大小不同，使用 intrinsic 指令集实现了两种方案的向量乘，兼顾大小规模情况下的计算；针对激活层，我们使用 intrinsic 指令集将 Sigmoid、Relu、Leaky_Relu、Tanh 四类通用激活函数实现了通用调用窗口，使用 switch 语句加函数指针的方式完成了四类激活函数共用同一调用流的方式，并提供了优秀的拓展能力，另外对 Softmax 进行单独的实现；针对归一化层，我们实现了批归一化函数，使用平均池化的内联汇编内核函数求得均值，然后使用 intrinsic 指令集操作向量寄存器实现归一化函数。

5. 针对卷积层的函数高性能实现，我们采用 Im2col+GEMM 的方式实现，我们针对国产 ARM 架构的体系结构，借鉴了开源高性能矩阵乘的实现方法实现了使用可变长向量拓展指令集的高性能矩阵乘函数，具体的，采用了矩阵分块、数据重排、循环展开、向量寄存器使用、指令流水等技术实现高性能矩阵乘法，并同时讨论了高性能矩阵乘的并行方案。并针对矩阵乘使用的列主序的存储格式，针对 NCHW 数据格式和 NHWC 数据格式设计实现了高性能 Im2col 函数。并使用可变长向量拓展指令集来采取向量优化的手段。

6. 对本文实现的高性能矩阵乘进行了测试，在方阵矩阵乘规模 512 到 8192 规模下能够实现比 BLIS 快 72% 的性能表现，比 OpenBLAS 的 ARM 通用编译版快 52% 的性能表现，比 OpenBLAS 的 A64FX 编译版本快 7% 的性能表现；因为本文开发阶段就将本文实现的高性能深度学习函数同深度学习函数的朴素实现进行精度对齐，故没有另重复做精度对齐实验。然后本文对使用内联汇编书写内核并深度优化的卷积层函数以及池化层函数进行性能测试，使用 AlexNet 和 VGG16 中的卷积参数和池化参数并将 ACL22.08 作为参照库进行测试，结果表明在 AlexNet 的卷积参数在 FTEngine 运行速度比 ACL 快 13%，在 VGG16 的卷积层参数下 FTEngine 比 ACL 快 15%。在 AlexNet 的池化层参数中，速度平均提升了 38%，而在 VGG16 的参数中速度平均提升了 11%；最终本文将实现的 FTEngine 对比 ACL22.08 运行了 AlexNet 和 VGG16 两个神经网络模型，结果表明在单线程情况下 FTEngine 执行 AlexNet 网络模型比 ACL22.08 快 14%，在 16 线程情况下执行 AlexNet 网络模型 FTEngine 要比 ACL22.08 快 5%，而在单线程情况下 FTEngine 执行 VGG16 网络模型比 ACL22.08 快 17%，在 16 线程情况下执行 VGG16 网络模型 FTEngine 要比 ACL22.08 快 6%。

本文基于国产 ARM 架构 CPU 处理器设计实现的深度学习函数库 FTEngine，结合了国产 ARM 架构 CPU 处理器的指令集特性和内存体系。根据实验结果表明，本文基于国产 ARM 架构 CPU 的缓存结构以及指令集设计的高性能矩阵乘函数相比开源线性代数库 BLIS、OpenBLAS 的矩阵乘函数有显著的性能提升，本文通过 Im2col+GEMM 实现的高性能卷积函数还有通过分层实现高度优化的池化层函数相对于开源人工智能图像处理函数库 ACL 也有更好的性能表现。运行当前比较

知名的网络模型也能获得比 ACL 更好的性能表现。但仍然本文还拥有很多可以继续改进的点。下一阶段的工作可以分为以下几个点：

1. 本文设计的高性能矩阵乘目前仅支持单精度矩阵乘法，对于双精度和低整型暂未实现，后续工作将完成低整型部分的高性能矩阵乘函数。
2. 本文设计的高性能深度学习函数库 FTEngine，设计和实现的核心是围绕利用可变长向量长指令集获取较高性能，是针对性能方面的探索，而在数据格式的支持程度以及拓展程度目前仅处于初期阶段。后续工作将考虑继续补充 FTEngine 深度学习函数库，引入深度学习中的前后处理函数以及拓展各类支持中间层，完善各种数据格式的支持。
3. 本文设计实现的深度学习网络层函数目前仅实现了神经网络推理的函数，如果作为神经网络推理库，后续工作将会考虑针对数据格式量化方面进行研究，来获得更高的吞吐量和计算效率。

参考文献

- [1] Turing A M. The imitation game[Z]. 2006.
- [2] Li L, Quan Z, Wang Z, et al. RIRCNN: A fault diagnosis method for aviation turboprop engine[C]// Peng R, Pantoja C E, Kamthan P. The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022. KSI Research Inc., 2022: 220-224. <https://doi.org/10.18293/SEKE2022-112>.
- [3] Sato M. The supercomputer "fugaku" and arm-sve enabled A64FX processor for energy-efficiency and sustained application performance[C]//19th International Symposium on Parallel and Distributed Computing, ISPDC 2020, Warsaw, Poland, July 5-8, 2020. IEEE, 2020: 1-5. <https://doi.org/10.1109/ISPDC51135.2020.00009>.
- [4] Sato M, Ishikawa Y, Tomita H, et al. Co-design for a64fx manycore processor and " fugaku " [C]// SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020: 1-15.
- [5] van de Geijn R A, Goto K. BLAS (basic linear algebra subprograms)[M]//Padua D A. Encyclopedia of Parallel Computing. Springer, 2011: 157-164. https://doi.org/10.1007/978-0-387-09766-4_84.
- [6] Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]// Proceedings of the 22nd ACM international conference on Multimedia. 2014: 675-678.
- [7] Abadi M, Barham P, Chen J, et al. Tensorflow: a system for large-scale machine learning.[C]//Osd: volume 16. Savannah, GA, USA, 2016: 265-283.
- [8] Jouppi N P, Yoon D H, Ashcraft M, et al. Ten lessons from three generations shaped google's tpuv4i : Industrial product[C]//48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021. IEEE, 2021: 1-14. <https://doi.org/10.1109/ISCA52012.2021.00010>.
- [9] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library [J]. Advances in neural information processing systems, 2019, 32.
- [10] Bradbury J, Frostig R, Hawkins P, et al. JAX: composable transformations of Python+NumPy programs [CP]. 2018. <http://github.com/google/jax>.
- [11] Intel. oneapi deep neural network library (onednn)[M]. GitHub, 2022. <https://github.com/oneapi-src/oneDNN>.
- [12] Stone J E, Gohara D, Shi G. Opencl: A parallel programming standard for heterogeneous computing systems[J]. Computing in science & engineering, 2010, 12(3): 66.
- [13] Arm. Arm-software/computelibrary[M]. GitHub, 2022. <https://github.com/ARM-software/ComputeLibrary>.
- [14] Chetlur S, Woolley C, Vandermersch P, et al. cudnn: Efficient primitives for deep learning[J]. CoRR,

- 2014, abs/1410.0759. <http://arxiv.org/abs/1410.0759>.
- [15] Jiang X, Wang H, Chen Y, et al. Mnn: A universal and efficient inference engine[C]//MLSys. 2020.
- [16] Tencent. Tnn: developed by tencent youtu lab and guangying lab, a uniform deep learning inference framework for mobile、desktop and server.[J]. GitHub repository, 2021-04-26. <https://github.com/Tencent/TNN>.
- [17] Ma Y, Yu D, Wu T, et al. Paddlepaddle: An open-source deep learning platform from industrial practice [J]. Frontiers of Data and Domputing, 2019, 1(1): 105-115.
- [18] Lavin A, Gray S. Fast algorithms for convolutional neural networks[C]//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, 2016: 4013-4021. <https://doi.org/10.1109/CVPR.2016.435>.
- [19] Mathieu M, Henaff M, LeCun Y. Fast training of convolutional networks through ffts[C]//Bengio Y, LeCun Y. 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings. 2014. <http://arxiv.org/abs/1312.5851>.
- [20] Chellapilla K, Puri S, Simard P. High performance convolutional neural networks for document processing[C]//Tenth international workshop on frontiers in handwriting recognition. Suvisoft, 2006.
- [21] Wang H, Ma C. An optimization of im2col, an important method of cnns, based on continuous address access[C]//2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE). 2021: 314-320. DOI: 10.1109/ICCECE51280.2021.9342343.
- [22] Cho M, Brand D. Mec: Memory-efficient convolution for deep neural network[A]. 2017. arXiv: 1706.06873.
- [23] Dukhan M. The indirect convolution algorithm[A]. 2019. arXiv: 1907.02129.
- [24] 黄春, 姜浩, 全哲, 等. 面向深度学习的批处理矩阵乘法设计与实现[J]. 计算机学报, 2022, 45 (225-239).
- [25] 庄晨. 基于 CPU SIMD 指令集的卷积计算优化[Z]. 中国科学院大学 (中国科学院深圳先进技术研究院), 2022.
- [26] Zebin T, Scully P J, Peek N, et al. Design and implementation of a convolutional neural network on an edge computing smartphone for human activity recognition[J]. IEEE Access, 2019, 7: 133509-133520. DOI: 10.1109/ACCESS.2019.2941836.
- [27] Lu K, Wang Y, Guo Y, et al. Mt-3000: a heterogeneous multi-zone processor for hpc[J]. CCF Transactions on High Performance Computing, 2022, 4(2): 150-164.
- [28] Chandra R, Dagum L, Menon R, et al. Parallel programming in openmp[M]. Morgan kaufmann, 2001.
- [29] Gropp W, Gropp W D, Lusk E, et al. Using mpi: portable parallel programming with the message-passing interface: volume 1[M]. MIT press, 1999.
- [30] 解庆春, 张云泉, 王可, 等. SIMD 技术与向量数学库研究[J]. 计算机科学, 2011, 38(298-301).
- [31] Peleg A, Weiser U. Mmx technology extension to the intel architecture[J]. IEEE micro, 1996, 16(4): 42-50.

- [32] Raman S K, Pentkovski V, Keshava J. Implementing streaming simd extensions on the pentium iii processor[J]. IEEE micro, 2000, 20(4): 47-57.
- [33] Lomont C. Introduction to intel advanced vector extensions[J]. Intel white paper, 2011, 23.
- [34] Cornea M. Intel avx-512 instructions and their use in the implementation of math functions[J]. Intel Corporation, 2015: 1-20.
- [35] Reddy V G. Neon technology introduction[J]. ARM Corporation, 2008, 4(1): 1-33.
- [36] Stephens N, Biles S, Boettcher M, et al. The arm scalable vector extension[J]. IEEE micro, 2017, 37(2): 26-39.
- [37] Wang H, Wu P, Tanase I G, et al. Simple, portable and fast simd intrinsic programming: generic simd library[C]//Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing. 2014: 9-16.
- [38] Goto K, Geijn R A v d. Anatomy of high-performance matrix multiplication[J]. ACM Transactions on Mathematical Software (TOMS), 2008, 34(3): 1-25.
- [39] Van Zee F G, Van De Geijn R A. Blis: A framework for rapidly instantiating blas functionality[J]. ACM Transactions on Mathematical Software (TOMS), 2015, 41(3): 1-33.
- [40] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Commun. ACM, 2017, 60(6): 84-90. <https://doi.org/10.1145/3065386>.
- [41] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[C]// Bengio Y, LeCun Y. 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. 2015. <http://arxiv.org/abs/1409.1556>.
- [42] 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. DOI: 10.1109/5.726791.
- [43] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[A]. 2014. arXiv: 1409.4842.
- [44] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[A]. 2015. arXiv: 1512.03385.
- [45] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks[C]//NIPS'12: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1. Red Hook, NY, USA: Curran Associates Inc., 2012: 1097-1105.
- [46] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[A]. 2015. arXiv: 1502.03167.
- [47] Rumelhart D E, Hinton G E, Williams R J. Learning representations by back-propagating errors[J]. nature, 1986, 323(6088): 533-536.
- [48] Malossi A C I, Ineichen Y, Bekas C, et al. Fast exponential computation on simd architectures[J]. Proc. of HIPEAC-WAPCO, Amsterdam NL, 2015, 56.
- [49] 张先轶, 王茜, 张云泉. OpenBLAS: A High Performance BLAS Library on Loongson 3A CPU[J]. Journal of Software, 2012, 22(zk2): 208-216.
- [50] Arm. Arm performance libraries[M]. Arm, 2022. <https://developer.arm.com/downloads/-/arm-perfo>

rmance-libraries.

- [51] Huang J, Van de Geijn R A. Blislab: A sandbox for optimizing gemm[A]. 2016.
- [52] Smith T M, Van De Geijn R, Smelyanskiy M, et al. Anatomy of high-performance many-threaded matrix multiplication[C]//2014 IEEE 28th International Parallel and Distributed Processing Symposium. IEEE, 2014: 1049-1059.

附录 A 读学位期间所发表的学术论文

1. 一种基于 SVE 指令集的池化层函数的高性能实现方法。授权公告号:CN 115878188 B (已授权, 第二作者)

附录 B 读学位期间所参加的科研项目

致 谢

致谢。