中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

深度学习处理器编程方法研究

作者姓名：兰慧盈

指导教师：陈云霁

中国科学院计算技术研究所

学位类别：工学博士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2018 年 12 月

The Study of the programming method for Deep Learning Processors

A dissertation submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Phylosophy

in

Computer Architecture

By

Huiying, Lan

Supervisor: Yunji, Chen

Institute of Computing Technology

Chinese Academy of Sciences

December 2018

中国科学院大学
研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：
日　　期：2018.11.22

中国科学院大学
学位论文使用授权的说明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：
日　　期：2018.11.22　　日　　期：2018.11.22

摘 要

近年来，深度学习算法迅速发展，被广泛应用在图像分类，人脸识别，自然语言处理，语音识别等众多领域中。但深度学习算法在实际应用上却受限于其训练和推理过程中的巨大运算量，传统的计算设备无法满足其在日常应用中的能效需求。因此，研究人员提出了面向深度学习的领域专用加速器，更快速高效的处理深度学习算法。然而，由于缺乏配套的软件栈，深度学习处理器的程序开发很困难，难以持续支持不断涌现的新型深度学习框架及算法，大大限制了其成果的进一步推广与移植。本文提出一种 5 层的深度学习加速器编程软件栈并对其中三个层次进行了研究。

本文的创新点如下：

设计并实现了一种面向深度学习处理器的高性能库（DLPlib），能够在不改动编程框架 Caffe 的应用程序的情况下，集成深度学习处理器（Cambricon-X），并达到手写指令性能的 56%-93%。DLPlib 通过将张量和过滤器数据分开封装，解决了深度学习处理器上神经元数据和权值数据摆放方式不同的问题，增加了数据局部性。此外，DLPlib 还对深度学习中的算子，以及矩阵运算进行了封装，以支持编程框架中的算子。

设计并实现了一种面向深度学习处理器的高级汇编语言（High-level assembly language, HLAL）和汇编器，以提升高性能库在实现新算法时的灵活性和可扩展性。HLAL 由底层汇编语句，宏汇编语句和高级 Block 构成。底层汇编语句是对硬件指令集的封装，宏语句用于提供编程抽象，Block 可以处理任意规模的神经网络算子（如卷积，池化等），进一步提供了编程便利性。评估结果显示，HLAL 在 10 种 benchmark 和手写优化的代码相比，在正向运算和反向运算中分别达到 95% 和 96% 的效率。

提出了一种面向深度学习处理器的中间表示（Deep learning intermediate representation, DLIR），用来解决多加速器之间的可移植性问题。DLIR 包括高级中间表示，低级中间表示，数据管理模块以及指令生成器。高级中间表示用于表示深度学习框架中的算子；低级中间表示用于表示底层硬件中的算子；数据管理模块支持数据自动划分和数据摆放，可以根据硬件资源选择划分策略，优化数据局部性；指令生成器可以利用自动双缓冲技术生成高效的代码。我们在三种不同架构，不同指令粒度的加速器上对中间表示进行评估，采用 6 种常用的全网网络进行实验，实验结果显示，中间表示可以很好的支持不同硬件架构，适应不同硬件规模，产生高效的代码，利用双缓冲技术，可以达到手写优化指令性能的 70.8%-97.7%。

关键词：高性能库，汇编语言，中间表示，深度学习加速器，深度学习

Abstract

In recent years, deep learning algorithms have been applied to a broad range of tasks, including image classification, facial recognition, natural language processing, speech recognition. However, applying deep learning algorithms to practical scenarios are restricted due to the enormous computational workloads consumed by training and inference phases. Traditional computational devices are not able to satisfy the performance and energy restrictions in practical applications. Therefore, many researchers proposed accelerators customized for deep learning algorithms to accelerate the computation. However, due to the lack of a corresponding software stack, it is difficult to develop new algorithms on such accelerators, and also hard to integrate them into the prosperously developing deep learning frameworks. Such difficulties cumber the promotion and transplantation of deep learning accelerators. In this paper, we propose a 5-level software stack and study three levels in the stack.

We make following contributions:

We design and implement a high-performance library for deep learning accelerators, through which, we can integrate the deep learning processor (Cambricon-X) without modifying the programming framework (Caffe), and achieve 56%-93% performance of hand-optimized implementations. The library also addresses the data placement of neurons and weights in deep learning processors and improve the data locality by using two data structures, i.e., Tensor and Filter, to encapsulate neurons and weights. In addition, DLPlib encapsulates a set of neural network operators and matrix operators to support the operations in the programming framework.

We design and implement a high-level assembly language (HLAL) and assembler for deep learning accelerators to enhance the flexibility and scalability of the high-performance library. The assembly language is composed of low-level assembly statements, macro statements and high-level blocks. The low-level assembly statement is the abstraction of hardware ISA. The macro statement provides programming abstraction. The block can process an arbitrary scale operator (e.g., convolution and pooling) and provide better programming convenience. We evaluate the HLAL on 10 benchmarks, and the results show that the assembler can offer 95% and 96% performance on forward and backward phases compared to the hand-optimized implementation.

We propose an intermediate representation (DLIR) to offer portability among different accelerators. The intermediate representation includes a high-level inter-

mediate representation, a low-level intermediate representation, a data management module and an instruction generator. The high-level intermediate representation is used to represent operators in deep learning frameworks. The low-level intermediate representation is used to represent hardware functions. The data management module supports automatic data partitioning and data placement. Partitioning strategy is selected based on hardware resources to optimize data locality. The instruction generator can generate highly efficient code by automatically applying the double buffering technique. We evaluate DLIR on three architectures with different instruction set architectures, adopting six commonly-used networks as benchmarks. The results show that the proposed intermediate representation is able to efficiently support all architectures and adapt to various hardware configurations, producing highly efficient code. By using automatic double buffering, we are able to offer 70.8%-97.7% performance of the hand-optimized implementation.

Key Words: high-performance library, assembly language, intermediate representation, deep learning processor, deep learning

目 录

图目录

表目录

第 1 章 绪论

1.1 深度学习算法发展现状

人工智能（Artificial Intellegence）指通过机器所表现出来的智能，通常是通过计算机程序实现，由于机器是人所制造的，因此称为人工智能。人工智能包含不同领域的问题，涉及范围广泛。其核心问题是通过机器构建出和人类相似，甚至超越人类的的推理，学习，计划，交流等能力。

人工智能学科中的一个分支，机器学习，是一类利用机器对数据进行建模，来解决问题的算法。机器学习在近 30 多年的发展中，已经成为一门多领域的交叉学科，涉及到概率论、统计学、计算复杂性理论等多门学科。机器学习算法通常会构建数学模型，然后利用历史数据，反复调整模型，经过多次迭代，让模型逐渐趋近于可以表达的真正的问题。通过设计这种让计算机可以自动“学习”的算法，就可以自动分析数据，获得规律，并利用规律对未知数据进行预测。如今，机器学习已广泛应用于各种应用场景，包括数据挖掘[11,12]、计算机视觉[4,13–15]、自然语言处理[16–18]、证券市场分析、语音和手写识别[1,19]、战略游戏和机器人[20–24] 等领域。

近年来，机器学习中的深度神经网络算法，开始逐渐表现出非凡的学习能力。深度神经网络（深度学习）是神经网络算法的延伸。神经网络算法起源于对生物大脑神经网络结构的模仿，用于对函数进行估计或者近似。神经网络由人工神经元和突触构成，图 1.1中是一个人工神经元的结构，其中 N1-N3 是输入神经元，它们通过突触 w1-w3 和神经元 x 相连，x 的值被计算出来之后，会经过激活函数的激活，被激活的神经元会被继续向后继续传递。通过反复训练，神经元之间的链接（突触数据）会发生变化，整个网络会逐渐拟合到训练数据所表达的规律。

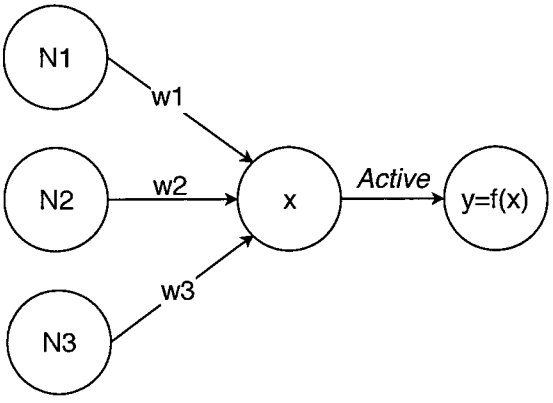


图 1.1: 神经元结构

在神经网络的基础上，人们通过增加神经网络的层数，提升网络的复杂性，构成多层神经网络，被称为深度神经网络，利用深度神经网络构建模型算法称为深

度学习算法。深度学习算法在处理图像[1,4,25–29]，自然语言[16,17,30–37]，语音[38,39]等任务上，相较传统的机器学习算法有明显的提升和进步。

多层的神经网络算法的模型通常比较复杂，需要大量的数据作为样本输入来训练，这些数据在互联网时代到来之前是不易获得的。但是随着互联网的发展，数据呈指数增长，大规模数据集[40]的出现，让深度学习算法成为可能。同时，计算设备（包括 GPU 和 CPU）的计算能力也在不断的增长。NVIDIA 最近推出的 Tesla P4 处理器，可以达到 5.5 TeraFLOPS 的单精度浮点运算能力。相比于 CPU，GPU 在处理深度学习算法上可以达到 60 倍的运算能力。这些条件的成熟，都给深度学习算法的发展提供了基础。

深度学习算法中最流行的算法包括卷积神经网络和递归神经网络，前者主要用于图像处理，后者主要用于输入长度不确定的数据的处理，比如语音数据，自然语言数据等。卷积神经网络最初由 LeCun[41] 等人提出，用于手写数字图像的分类，在手写数字数据集 LeNet5 上取得超过 90% 的识别率，近年来又被扩展到更大的数据集和网络结构上，用于更复杂的学习任务，比如 ILSVRC 比赛中的 ImageNet[40] 图像分类任务。2012 年，Krizhevsky[25] 等人提出的 AlexNet 成功的将 ILSVRC-2012 比赛[42] 图像识别任务的 top-5 的错误率降低到 15.3%。随后的几年内，研究人员们不断改进神经网络的算法和网络结构，规模越来越大，拓扑结构也变得越来越复杂，相应的任务的识别率也显著提高。2016 年，Ren 等人[4] 提出的 ResNet 采用了多达 1001 层的网络结构，将 ILSVRC-2015 图像分类任务的错误率降低到 3.57%。

从学习任务上来说，机器学习算法可以分为三类：有监督学习，无监督学习和半监督学习（增强学习）。在这三类学习任务上，深度学习算法都取得了非常好的效果。

有监督学习（Supervised Learning）。有监督学习的训练数据都是带有标签的，数据集中的每个样本都带有自己的标签，因此可以明确的判断一个样本的预测结果正确与否。一般情况下，在模型的训练过程中，数据集被分为训练数据集和测试数据集。训练数据集用来训练模型，而测试数据集用来验证模型的学习情况，实时的调整学习策略。最常见的有监督学习是分类任务。上文中提到的 AlexNet[25] 等网络都是用于图像分类任务的深度学习算法。通过加深网络拓扑，研究人员们成功的将图像识别任务的正确率推到了人眼识别的极限[43]。

无监督学习（Unsupervised Learning）。无监督学习[44–46,46–48] 中，用来学习的数据集都是没有标签的，因此无法决定一个具体的正确类别，也就是数据在学习中，是没有“正确答案”用来校对的。一般而言，无监督学习的任务会去寻找数据中更加广泛，更加普遍的一些相似点，根据这种相似性将数据分成几个簇（cluster），但是这些簇的具体含义我们并不知道。比如，autoencoder[49] 算法就是一种被广泛应用的无监督学习算法，其输入数据通过多层感知机被转化成一个低

维向量，之后再利用反向算法，将这个向量恢复成原始数据大小，标签被设置为和输入相同的数据，通过多次迭代，这个低维向量就可以用来表示这个输入向量中的重要信息。

增强学习 (Reinforcement Learning)。增强学习[20,50–54] 可以被认为是一种介乎于监督学习和无监督学习之间的学习方式。增强学习的数据可以认为是有标签的——反馈 (reward，相当于标签) 是根据代理 (agent) 的行动 (action，相当于输入) 提供的，但是学习机制是直接和环境进行交互的，环境中的变化会生成一个特定的反馈给代理，代理通过不断的行动-反馈的学习，逐渐达到预想的目标。增强学习系统的目标，是通过学习，可以在每一个状态下都能够采取最好的行动，以最大化每一个状态的反馈。Deep Mind 提出的 AlphaGo[55] 和之后提出的 AlphaGo Zero[56] 就是利用了深度学习算法的增强学习模型来学习下围棋的任务。AlphaGo 通过学习人类棋谱，打败了所有与其对战的人类棋手。而之后的 AlphaGo Zero 可以在不输入人类棋谱的情况下，通过自己和自己博弈进行更快的学习，这都得益于深度学习算法的超强学习能力。

1.2 深度学习加速器

深度学习算法的计算量非常大，其推理和训练过程都需要花费大量时间。训练过程需要进行多次的迭代，不断调整参数，直到模型收敛。而推理过程在实际应用（比如嵌入式设备）中通常有严格的延迟限制，传统的计算设备（CPUs/GPUs）无法提供足够的能效。在这种情况下，研究人员提出了面向深度学习算法的硬件加速器。深度学习加速器可以有效降低功耗，同时成倍的提升深度学习算法的处理性能。

2014 年，Chen[5] 等人提出了 DianNao 神经网络加速器，通过对神经网络算法进行 tiling，以及使用高带宽的片上存储，有效的减少访存。2015 年，Chen 等人提出了 DaDianNao[57]，一个用于机器学习的超级计算机。DaDianNao 为了进一步减少数据搬运的性能开销，提出将所有的权值放在片上，每一个 Node 包含采用 edram 作为存储介质，通过多个 Node 链接的方式，保证加速器的可扩展性。ShiDianNao[58] 是一个第一个基于脉动阵列机的低功耗深度学习加速器，其二维阵列的设计可以高效的处理二维的图像数据。上述的加速器都着重于对深度学习算法的加速，Liu 等人提出的 PuDianNao[59] 可以支持 7 种机器学习算法，尤其是一些传统的机器学习算法，包括 K 均值 (Kmeans) [60]，K 临近 (KNN) [45]，决策树[61]，支持向量机 (SVM) [62] 等。PuDianNao 同样利用的是机器学习算法中数据可以重用的特性，对算法进行 tiling 以减少数据搬运，从而提高效率，减少功耗。Google 提出一款可以用在数据中心和云端的神经网络加速器：Tensor Processor Unit (TPU) [63]，TPU 的结构也是基于脉动阵列原理，它可以通过 Google 的深度学

习框架 TensorFlow[64] 进行编程，方便的部署在深度学习系统中加速计算。Chen[65] 提出了 Eyeriss 加速器，通过分析卷积神经网络的数据流和数据重用，提出一种新的数据重用策略，有效的减少数据搬运，从而提升了性能。为了进一步减少运算量和访存量，人们提出了支持数据压缩的加速器。EIE[66] 是一个针对 LSTM 结构中全连接层的权值进行压缩的加速器；Cambricon-X[8] 是首个可以支持卷积稀疏化的加速器。随着深度学习算法的快速发展，算法越来越多样化，为了支持更多种类的算法，Liu 等人提出了 Cambricon[9]，首个针对深度学习加速器的指令集。除了预测之外，深度学习算法的训练计算量也非常大，Venkataramani[67] 等人提出一种针对训练的加速器 ScaleDeep，将深度学习算子分成计算密集和访存密集两种，分别设计了两种计算模式，用来加速不同类型的计算模式。

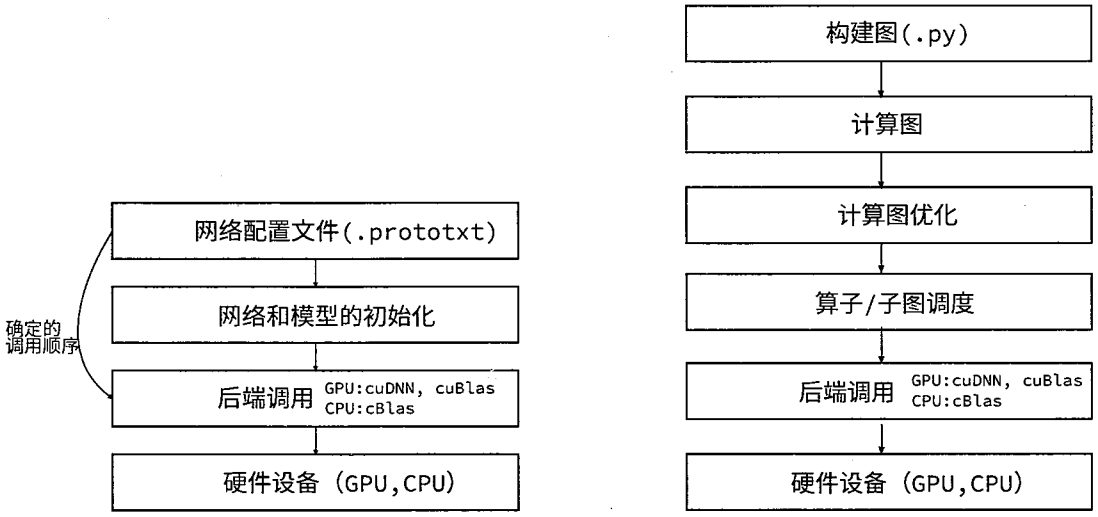
1.3 深度学习编程框架

早期的深度学习网络只包含几种固定的算法，网络结构也较为简单，多为线性结构（比如 AlexNet[25], VGG[13] 网络），对深度学习编程框架的灵活性需求较低。而随着深度学习算法的快速发展，越来越多灵活的算法被提出，深度学习网络的算法和结构都在快速的演化，因此对编程框架的灵活性和性能提出了进一步的需求。深度学习编程框架软件栈的深度也在增加，从功能上可以分成两个方面，一个是网络拓扑结构的构建方式，主要有两种，基于层的拓扑结构和基于计算图的拓扑结构。另一方面是对于单个算子的实现方式，可以分成三种，一种是调用高性能库，比如 cuDNN[68], cuBlas 等来直接实现一个算子；第二种是用基本算子实现新算子，基本算子依然通过调用高性能库实现；第三种方法则通过调用专门的领域专用语言（Domain-specific language, DSL）来描述计算过程。

1.3.1 网络拓扑结构构建

根据网络拓扑结构的构建方式，深度学习编程框架主要可以分为两类[69]，一类是基于层的编程框架，一类是基于计算图的编程框架。图 1.2表示了这两类框架的区别。

基于层的这类框架接收一个由各种层组成的网络配置，执行的顺序严格按照输入配置的层顺序进行，框架所做的工作是分析配置中的每个层，然后调用相应的底层算子接口。这些算子通常是底层硬件提供的高性能库编程接口。比如，由 Berkeley AI Research (BAIR) 和 the Berkeley Vision and Learning Center (BVLC) 提出的 Caffe 深度学习框架，就是一个基于层的框架。它通过接收一个 prototxt 的配置文件，对网络进行配置，之后框架会进行空间分配，网络初始化等工作，之后就是根据配置中的层顺序调用底层的接口，比如，对于 GPU 后端来说，Caffe 会去



(a) 基于层的深度学习编程框架　　(b) 基于图的深度学习编程框架

图 1.2: 深度学习编程框架

调用 cuDNN[68] 和 cuBlas 接口。这类编程框架对性能的优化主要集中在每一个算子的实现部分，而由于没有中间的计算图表达，因此不会对网络图结构进行优化。

另一类编程框架是基于计算图结构的编程框架。这类编程框架会让用户定义一个由 symbolic 节点构成的图结构，每一个节点表示的是操作或者输入数据，然后对这个图进行优化，比如消除无用节点，节点融合等，之后框架会对图中的节点进行调度，和基于层的框架不同，节点的执行顺序，以及执行在哪个设备上都是在调度中动态决定的。这样的执行逻辑在多设备的情况下很有优势，比如，一个分布式系统中，由于有中间图的存在，框架可以动态的进行图分割，将提取出来的不同子图部署到不同的运行设备上运行。部署的依据可能是设备的闲置情况，计算能力，以及算法和设备的匹配程度。当下流行的基于图的编程框架包括 TensorFlow[64]，MXNet[70]，Torch[71] 等。

1.3.2 实现单个算子的方法

根据编程框架实现单个算子的方式，可以分成三大类。

第一类实现方式中，每个算子的实现都是通过直接调用底层设备的高性能库，这种实现方式，一般会在实现每个算子的时候对实现方式进行充分的优化。最有代表性的是 Caffe 框架，其中的每一个层的正向和反向，都是通过调用后端提供的库，比如：对于 GPU 的后端，Caffe 会直接调用其提供的 cuDNN[68] 深度学习算法库生成后端指令。而框架的作用只是进行一些参数的传递，和数据的分配拷贝，基本优化都在算子的实现内部。

第二类实现方式中，框架首先实现一部分核心算子，比如卷积操作，矩阵操

作，算数操作等，这些算子的实现方式同第一类一样是调用后端高性能库进行指令生成，而这类实现不同的地方在于，对于越来越多的新增算子，不是让程序员利用库或者框架外代码实现，而是利用框架内，已经定义好的核心算子，来实现新的算子。新的算子相当于构建一个新的子图，这个子图可以抽取出来，融合到整体图中，构建出一个大的计算图，然后进行优化。由于实际上并没有引入新的后端实现，新算子可以被框架转化为已有的算子，相当于并没有引入新的算子，新算子可以直接被框架的调度器识别和调度，因此也不需要对调度器进行修改，减少了维护开销，也更利于对计算图进行优化。TensorFlow[64] 使用的就是这种实现方式。

第三类实现方式中，框架引入一种领域专用语言来实现一个特定的算子，这种方式的引入来源于对新算子的需求。第二种实现方式虽然可以一定程度上简化实现算子的方式，但是依然有灵活度的限制，同时，性能上也不容易得到保障。而如果想要生成非常高效的算子，依然需要程序员手动的对算子进行优化，这样的人力开销依然很大。在这种情况下，研究人员们开始在框架中嵌入领域专用语言，用更少的人力实现更高效的新算子。一种有代表性的做法是 TVM[7] 深度学习软件架构中，引入了 Halide[72–74] 中计算和调度分离的思路，利用 Halide IR 来实现算子，并且提供了一系列 Schedule（调度）方法优化计算到硬件的映射过程。另一个最近提出的项目，TensorComprehension[75]，同样为了更快更高效的描述和实现一个算法，定义了一种矩阵描述的领域专用语言，利用 Polyhedral[76,77] 模型对代码进行自动的优化，生成高效的可执行代码。

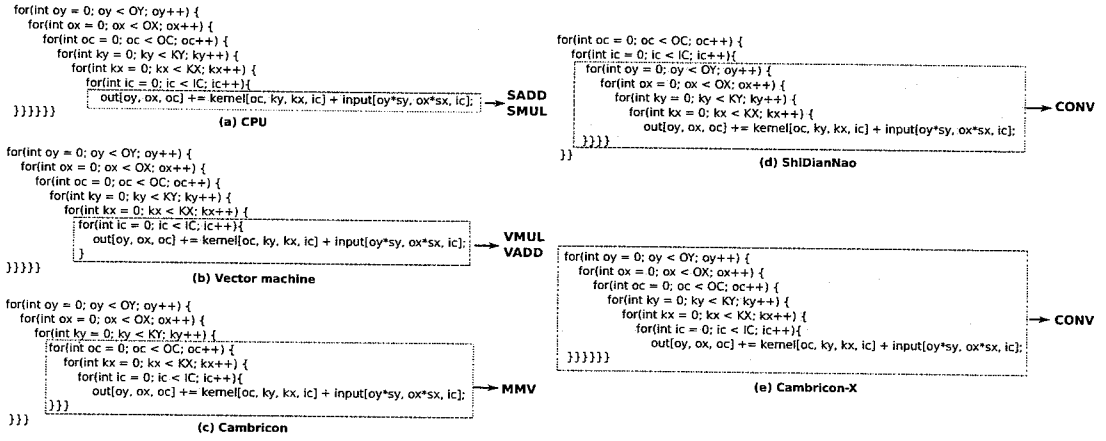


图 1.3: 卷积到不同粒度的加速器上的映射

1.3.3 现存深度学习加速器的编程方法

我们收集了近 5 年内（2012-2017）体系结构顶级会议（即：MICRO，ISCA，HPCA，ASPLOS）中发表的深度学习处理器文章，共 18 篇，并统计了这些加速器所使用的编程方法，如表 1.1所示。我们发现，18 篇中有 6 篇文章并没有提到编

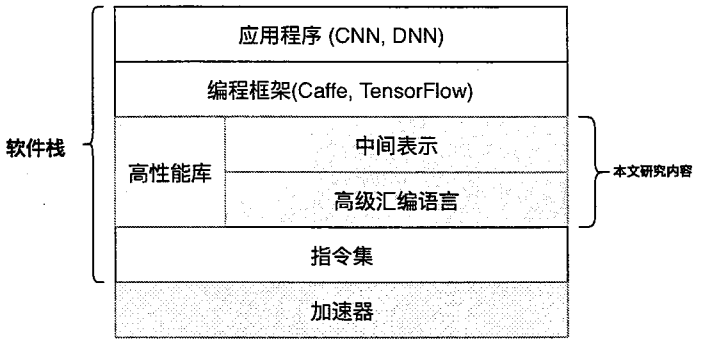


图 1.4: 软件栈

程方式，表中我们标注为 Unknown。剩下的 12 篇采用的编程方法从抽象层次和算子两方面都存在差异。这种差异一方面会增加程序员的学习成本，另一方面增加了将加速器集成到编程框架中的开销。

为了进一步说明深度学习加速器提供的编程接口的差异性，我们将接口按照操作粒度和编程抽象层次两个维度进行分类，并分别介绍它们的优势和劣势。

按照编程抽象层次，我们可以将加速器的编程接口分成五类。

- **指令集/汇编语言**。指令集和底层汇编语言（可读指令）是加速器提供的最底层的编程接口，因为层次低，所以它可以提供最完备的硬件功能，所有硬件能支持的功能都可以通过指令集中的指令来控制，带给用户很大的灵活性，但同时也造成很大的编程难度。因为用户需要自己手动的对片上资源进行管理，同时需要对指令之间的并行执行，同步等操作进行控制，非常容易出错。用汇编语言进行编程对用户的水平要求很高，如果用户对硬件的结构不够了解，很可能写不出高效的程序，因此也无法发挥汇编语言的效果。

- **指令生成器/配置文件**。指令生成器通过接收一个包含层类型、规模、参数信息的网络结构配置，生成相应的指令序列。这样的指令生成器的编程非常方便，只需要按照格式填写相应的参数，就可以生成指令，保证了易用性，但是其缺点也很明显，即缺乏灵活性。用户只能在硬件上运行指令生成器支持的几种算法，而无法让用户自己添加新的算子。除此之外，由于没有提供低层次的编程接口，用户也无法利用硬件特性，去对算法做进一步的优化。

- **库/API**。库通过提供给用户一系列 API 来构建网络。这些 API 包括了各种粒度的算子，每一个算子可以处理任意大小的输入输出规模的计算，每个算子对应一次内存调用。用库来编程的好处在于，学习成本相对低，编程友好，由于算子的粒度比较粗，如果算子的性能优化的好，则容易生成高效的程序。但是缺点在于，灵活性和可扩展性较低，同时由于不提供硬件相关的信息，用户无法自行进行优化。

表 1.1: 现在加速器的编程方法

编程抽象层次	算子粒度	加速器
ISA	Operation	Cambricon[9]
Code generator	Layer	DianNao[5], DaDianNao[57], ShiDianNao[58], Neurocube[78]
Library	Layer	CambriconX[8]
Library	Network	PipeLayer[79]
Compiler	Layer	PRIME[80]
Compiler	Network	SCALEDEEP[67]
Framework	Layer	RedEye[81]
Framework	Network	Minerva[82], TPU[63]
Unknown	Unknown	Stripes[83], ISAAC[84], Cnvlutin[85], EIE[66], FlexFlow[86], SCNN[87]

- **编译器/DSL**。还有一类深度学习加速器采用自己提出的语言作为编程接口，同时需要实现相应的编译器来支持整个编译过程。这样做的好处是，整个编译过程可以非常灵活，做多个 pass 的优化，缺点是实现起来的开销较大，对于小型的项目可能没有这么多人力去开发一个编译器。

- **框架/Symbolic 图表达**。还有一类深度学习加速器使用深度学习框架（比如 TensorFlow[64]）作为编程接口。采用框架的好处是，可以直接利用框架的编程接口，对用户友好，表达灵活。而其缺点是，由于小的操作很多，需要多次的内核调用，导致性能下降。

从另一个维度算子粒度上，我们可以将加速器的编程接口分为三类，基本算子，领域专用算子，以及网络算子。

- **基本算子（Basic Operation）**。基本算子粒度的编程接口提供细粒度的操作，通过算子的组合，我们可以实现一个神经网络中的算法，比如卷积，池化等。由于一种算法可以有多种实现方式，提供算子粒度的接口让用户可以自己定义算法，提供了灵活性，让加速器更具有可编程性和可扩展性，但是相应的也会增加编程难度。比如，Cambricon 指令集提供的一系列矩阵运算就是基本算子。

- **领域专用算子（Domain-specific Operatoin）** 领域专用算子粒度的编程接口提供可以执行神经网络算法的算子，比如卷积算法，池化算法等。这些算法可能是直接由硬件的指令所提供，也可能是由库，或者指令生成器来支持。比如，ShiDianNao 的指令集，和其提供的指令生成器都是领域专用算子粒度

的。这种算子粒度的好处是，可以较快的构建一个深度学习网络，而省去部分编程开销，但是相应的这种便利也会降低灵活性。

- **网络算子（Network Operatoin）** 一些加速器只能支持特定的网络，比如 LeNet[41]，AlexNet[25] 等。这类编程接口可以对网络进行极致的手动优化，达到接近理论值的性能，但是相应的不具备任何灵活性。

可以看出，深度学习加速器的编程方法多种多样，没有被统一到一个软件栈中，这使得使用加速器的学习成本很高，面对日益增长的算法需求，现有的编程手段无法满足性能和编程效率上的要求，本文针对这一问题，提出一个面向深度学习处理器的 5 层的软件栈，让广大的深度学习算法研究者们可以在深度学习框架中利用深度学习处理器来进一步提升开发效率。

1.4 研究内容和主要贡献

本文中，我们设计了一个面向深度学习加速器的 5 层软件栈，如图 1.4所示，包括应用程序层，编程框架层，中间表示层，高级汇编层，指令集层，以及高性能库。本文重点研究了中间表示，汇编语言，以及高性能库，本节中我们简单分析设计软件栈所面临的挑战，并阐述我们是如何通过这个 5 层软件栈解决这些问题的。

在为深度学习加速器设计软件栈的时候，我们面临的主要挑战以及相应的解决思路如下：

框架可移植性。过去几年人们开发了大量的深度学习框架，这些框架已经有大量的用户，它们提供的接口也可以很好的描述机器学习算法。深度学习加速器作为一种计算设备，应该最大限度的利用已有资源，便于集成到现有框架中。我们发现，框架的上层接口虽然差异很大，但是其后端通常会调用底层硬件提供的高性能库，比如 NVDIA 公司提出的专门用于加速 GPU 上深度学习算法的高性能库 cuDNN[68]，以及用于加速矩阵运算的 cuBlas 等。为了满足这种特性，我们在 3章中提出一种面向深度学习加速器的高性能库的设计，并将其集成到深度学习框架 Caffe[88] 中，大幅降低了开发难度。

硬件可移植性。近年来各种深度学习加速器架构也在不断涌现，它们的架构设计差异很大，同样也给软件栈在后端的移植带来很大的困难。硬件虽然架构上差异很大，但是从编程角度来看，我们只需要考虑其指令集所能支持的功能，而不需要在意具体的硬件实现。根据这种原则，我们在 5章中提出一种面向多种深度学习加速器的中间表示，其包含多种粒度的算子，支持标量，向量，矩阵和张量运算。这些算子是对深度学习处理器功能的抽象。

灵活性和运行效率的权衡。使用深度学习加速器主要目的是提高计算速度，性能效率是至关重要的，但是，加速器特殊的硬件特性让高效指令生成变得困难。作

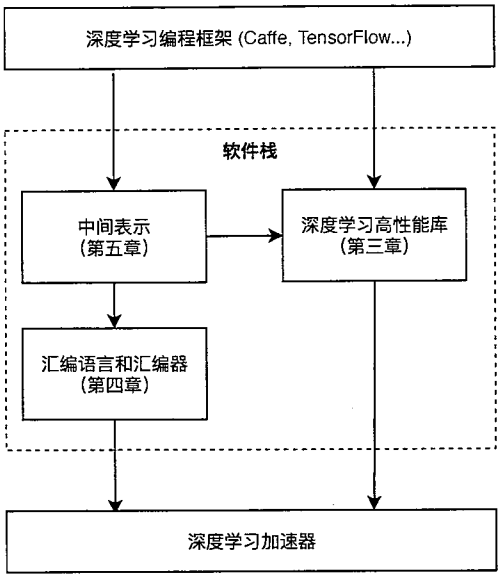


图 1.5: 研究内容

为一种新生的计算设备，深度学习加速器和传统计算设备有很大的区别。为了加速深度学习算法，加速器中引入了很多特殊的定制化设计，让深度学习加速器从功能和组织结构上和传统的 CPU 和 GPU 有较大差异，可归纳为一下三点。

（1）从片上资源的角度来看，由于深度学习算法每一次计算对数据的需求量都很大，因此，深度学习加速器中通常采用可以支持变长数据存取的 scratchpad memory 而不是 cache 作为片上的 buffer。但是 scratchpad memory 虽然可以提供更高的带宽和更灵活的数据存取，却需要程序员手动的进行存储和释放，同时，由于加速器的片上资源非常有限，通常无法容纳一个算法的全部数据，而是需要被分成多次进行加载和计算。这种拆分是交给用户去做，还是提供某种操作原语或者由编译器去做是在设计编程框架或者编译器的时候需要考虑的问题。(2) 为了可以高效率的支持深度学习算子，深度学习加速器中提供的指令多是基于 tensor 的复杂算子，比如，2D 的卷积操作，池化操作等，LRN 操作等。这些操作实际上是将原本的计算循环中的一个较大部分固化到了硬件上来操作。而不同的加速器的操作粒度是不同的，图 1.3表现了不同加速器以及 CPU 对于一个卷积计算的可能的拆分情况。这种基于 tensor 的粗粒度原子操作和传统的 CPU 和 GPU 也是完全不同的。（3）由于深度学习算法具有计算访存密集，但控制流简单的特点，这样的计算模型和一般 CPU 上所执行的有大量控制流变化的程序差别也很大。这种计算模式也给了我们对程序进行优化的机会，同时也启发我们去针对这种特殊的模式来设计编程模型。针对提升效率和编程灵活性的问题，我们在第 4章中提出一种面向深度学习加速器的汇编语言和汇编器，其暴露了更多的硬件功能和底层信息，让程序员可以充分利用加速器底层指令进行优化。

图1.5中呈现了本文各章节的研究内容，以及其相互之间的关系，具体而言，本

文贡献可以归纳如下：

1. 我们设计并实现了一种高性能库的解决方法，既能够方便的集成到编程框架中，同时可以保障执行效率。我们将深度学习加速器中一些常见的数据结构，以及算子进行封装。神经元和权值数据被封装在多维数组 Tensor 和 Filter 数据结构中，以解决神经元和权值数据的摆放问题，并且包含一组预定义的算子，包括深度学习算法和矩阵计算，算子的输入输出数据即为 Tensor 和 Filter。我们通过 DLPLib 将深度学习加速器集成到了 Caffe 中，极大降低了开发难度。在 Cambricon-X 加速器上进行实验的结果显示，DLPlib 可以达到手写优化代码效率的 56%-93% 。

2. 我们设计并实现了一种用于深度学习加速器的高级汇编语言并实现了相应的汇编器，来提升编程的灵活性和可扩展性。高性能库的解决方案虽然可以简单高效的支持加速器，但是相应的，也缺乏灵活性——难以添加新的算子，库的性能也完全依赖于厂商对算子的优化。这是由于高性能库提供的编程接口的抽象层次太高，用户接触不到底层硬件指令，无法通过底层指令的方式直接在加速器上编程，因此也就不能实现高效的新算子。针对这个问题，我们提出一种用于深度学习加速器的高级汇编语言，其中既包括底层的汇编指令，宏汇编指令，同时也包含一组用于方便编程的神经网络扩展库。汇编语言提供可设置分段的 Tensor 和 Filter 数据结构，用来存放神经元和权值，同时支持一系列片上数据存取的原语。我们在 Cambricon 指令集以及相应的原型加速器上进行实验，结果显示，HLAL 在 10 种 Benchmark 上，正向和反向运算分别能够达到手写优化代码的 95% 和 96% 。

3. 我们进一步提出一种面向深度学习处理器的中间表示，以解决程序在不同加速器上的可移植性以及程序的优化问题。汇编语言由于提供了底层的语句，从而给用户提供了足够的灵活性，可以达到较高的运行效率，但是正由于它的抽象层次过低，让实现新算子的编程比较困难，尤其是对于规模较大的算子，这种算子需要被拆分成多个子操作，分时的进行计算，使用汇编语言编写算子拆分很困难，也容易产生错误，此外数据的分段策略的制定也是一个不易解决的问题。针对这个问题，我们提出一个中间表示层，位于软件栈中编程框架和汇编语言之间，具体包含：高级中间表示和低级中间表示，数据管理模块，以及指令生成器。我们在三种不同结构，不同指令粒度的加速器上，采用了 6 种常用的全网网络进行实验，利用双缓冲和自动分段，可以达到手写指令优化效率的 70.8%-97.7%。

1.5 文章结构安排

本文共包括 6 个章节。

第 2章中，我们对本文研究内容的背景进行介绍，包括神经网络算法的发展，各种常用的神经网络算法介绍，以及它们的应用场景；之后介绍了各种最近提出的神经网络加速器，以及这些加速器的编程方法；最后，介绍了在传统设备上进行编程使用的深度学习编程框架，以及它们的特点。第 3章中，我们提出一个用于深度学习加速器的高性能库。介绍了高性能库中包含的数据结构，算子选择，以及 API 的设计。第 4章介绍了面向深度学习加速器的高级汇编语言，包括底层的低级语句，以及高级的 Block 提供类似库的功能，既可以提供足够的灵活性，同时利用 Block 提供编程上的便利性。第 5章中，我们提出一个针对深度学习加速器的中间层，中间层的上层可以连接多种深度学习架构，底层可以对应不同架构的深度学习加速器。中间层包括用来描述算法的中间表示，以及指令生成器。第 6章中，我们对全文做一个总结，并展望未来的工作方向。

第 2 章 背景

2.1 深度学习算法

深度学习算法属于机器学习算法中的一类。近年来，深度学习算法在许多不同领域的任务上取得了非常好的效果，包括如图像识别，图像分类，图像生成，自然语言处理，语音识别等领域。深度学习算法利用多层神经网络构建出一个复杂的模型，同时利用信息时代带来的红利，巨大规模的数据集，减少过拟合作用，使得模型的识别能力显著提升。我们下面将介绍几种常见的网络，包括卷积神经网络，递归神经网络，以及各种改善训练速度提升性能的方法，如权值压缩，低精度计算等。本节中，我们将介绍一些常用的神经网络算法。

2.1.1 卷积神经网络（Convolutional Neural Networks, ConvNets/CNNs）

卷积神经网络是神经网络的一种，它在很多和图像相关的应用上都取得了非常好的效果。比如，图像识别，图像分类，目标检测，人脸识别等，这些任务在自动驾驶，智能机器人等应用中能起到很重要的作用。

最早的卷积神经网络 LeNet5[1] 是 1988 年由 LeCun 提出的卷积神经网络，用于手写数字识别，成功的将识别率提升到接近人眼的识别率。但是由于当时计算设备的计算能力匮乏，而卷积神经网络算法的计算量巨大，同时也缺乏足够的训练数据，导致卷积神经网络只能用在手写数字识别这种小规模的任务上，而无法进一步推广。图 2.1中是一个 LeNet5 卷积神经网络的结构。LeNet5 中包含两个卷积层，两个激活层，两个池化层，和三个全连接层，它们是现代神经网络中运用最为广泛的算法。

随着计算机技术的不断发展，数据变得越来越丰富，计算设备的性能也大幅度提升，人们开始利用神经网络算法去解决更复杂的识别问题。2012 年，Krizhevsky[25] 等人提出的 AlexNet 成功的将 ILSVRC-2012 比赛[42] 图像识别任务的 top-5 的错误率降低到 15.3%。随后的几年内，研究人员们不断改进神经网络的算法和网络结构，规模越来越大，拓扑结构也变得越来越复杂，相应的任务的识别率也显著提高。2016 年，Ren 等人[4] 提出的 ResNet 采用了多达 1001 层的网络结构，将 ILSVRC-2015[42] 图像分类任务的错误率降低到 3.57%。我们将在本节中介绍神经网络中最常用的几种算法层，以及一些具有代表性的整体网络结构。

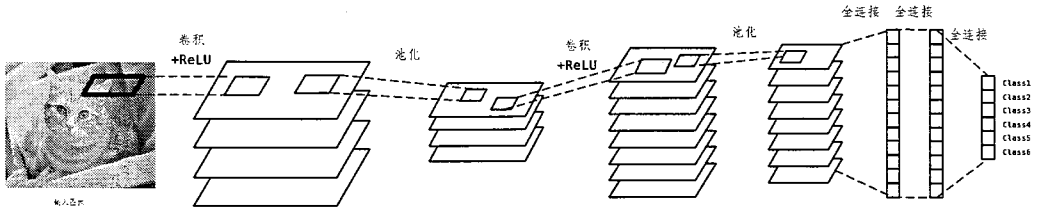


图 2.1: LeNet5 卷积神经网络结构[1]

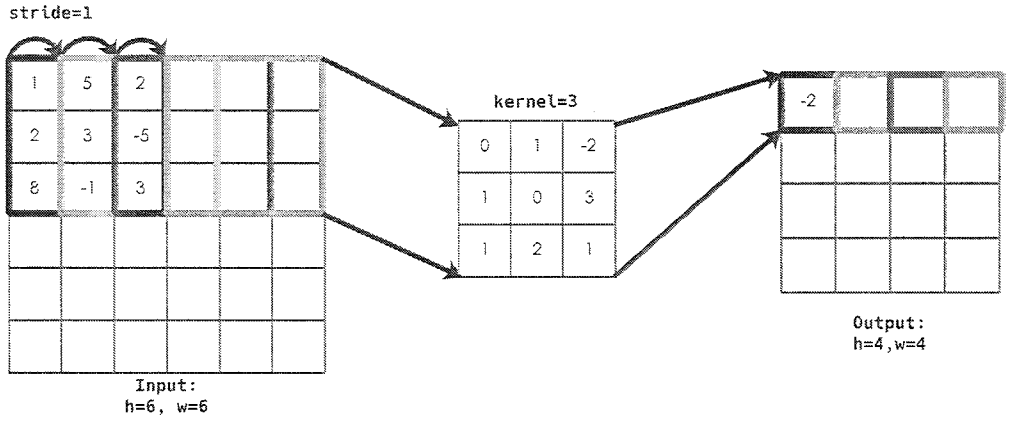


图 2.2: 卷积操作[1]

2.1.1.1 卷积层（Convolution）

卷积层[1] 是卷积神经网络中的一种重要算法，它是卷积神经网络中的重要组成部分。卷积层中包含很多的卷积核，每个卷积核在空间维度上被共享。卷积核在输入的特征图上滑动，可以得到一个被提取过后的特征图。图 2.2中展示了一个在一个特征图上的卷积操作。卷积操作的输入是 6×6 的特征图，卷积核的大小是 3×3，步长为 1×1，输出是 4×4 的特征图。每一个输出点是输入和卷积核做点乘得到的。当输入有多个通道（channel）时，每个输出特征图会对应一组卷积核，每个核对应一个输入通道，计算出来的值相加获得最终的输出值。

2.1.1.2 池化层（Pooling）

池化层，通常接在卷积层后面，对卷积的输出特征图进行降采样，对数据进行降维。池化的取值方式主要有两种，一种是取最大值，即输出池化区域内的最大值，第二种是取平均值，即输出池化区域内数值的平均数。图 2.3中为一个池化层的例子，池化的输入特征图大小为 4×4，池化的区域大小为 2×2，输出大小为 2×2。这个例子中，我们进行的是最大池化操作，即输出为池化区域中的最大值。

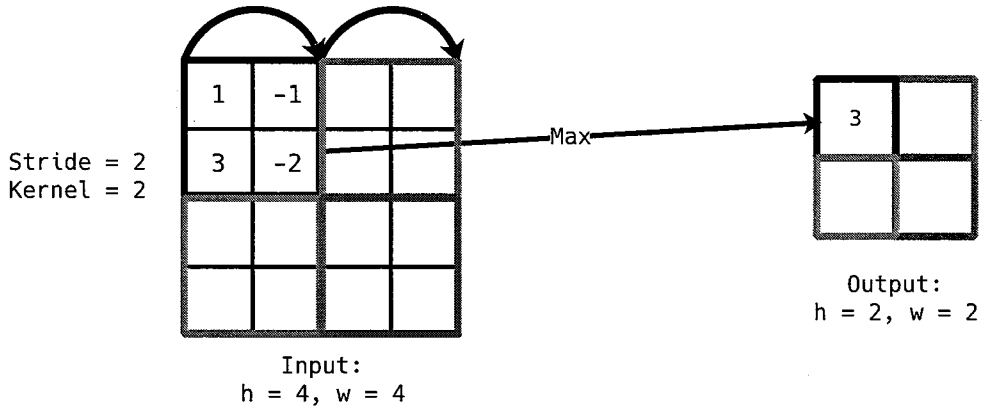


图 2.3: 最大池化层

2.1.1.3 正则化层 (Normalization)

正则化层用于在训练过程中，调整数据的范围，目前提出的正则化层主要包括两种。

2.1.1.4 Local Response Normalization (LRN)

LRN 层[25] 最早由 Krizhevsky 等人提出，用于帮助算法的泛化。LRN 算法的公式如下所示：

$$out^i_{x,y} = in^i_{x,y} / \left(k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (in^j_{x,y})^2 \right)^{\beta} \tag{2.1}$$

其中，i 表示第几个特征图，x, y 表示一个特征图上的一个点的坐标。计算每一个输出 $out^i_{x,y}$ 的值，需要将对应的 N 个点相加。N, K, α 和 β 是在训练之前设置的参数。

2.1.1.5 Batch Normalization (BN)

Batch Normalization 层[89] 用于在训练时对输入数据归一化的处理，以降低网络中间值的协方差偏移 (convariance shift)。研究人员观察到，当输入数据被归一化后会加速训练的速度，因此提出在网络结构中加入归一化操作，对一些层，比如卷积层，全连接层的输出进行归一化，减少数据分布的偏移，提升训练效果。

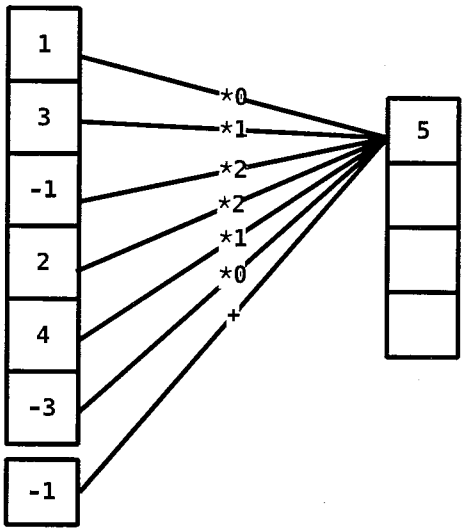


图 2.4: 全连接层

2.1.1.6 **全连接层**（Fully-connected）

全连接层和常规的神经网络中的层类似，其输入神经元和每一个输出神经元之间都会有权值的连接，如图 2.4所示。每一个输出神经元的值是输入神经元和权值加权之后的累加和。在实现中，全连接层通常是通过矩阵乘法来完成。对于一个 b 个样本，m 到 n 个神经元的全连接层，其输入是一个 $b \times m$ 的二维矩阵（batch×输入维度），权值则是 $m \times n$ 的矩阵，输出则是一个 $b \times n$ 的矩阵。

表 2.1: 激活函数

激活函数	公式
Sigmoid	$f(x) = \frac{1}{2+e^{-x}}$
Tanh	$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
ReLU	$f(x) = max(0, x)$
Leaky ReLU	$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x > 0 \end{cases}$，其中 α 是一个固定的很小的值
Parametric ReLU	$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x > 0 \end{cases}$，其中 α 是一个可学习的参数
Randomized ReLU	$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x > 0 \end{cases}$，其中 α 是一个随机数

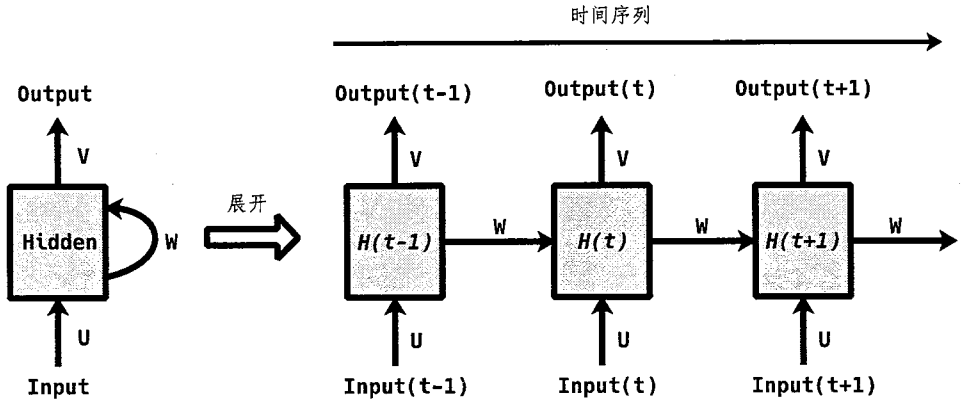


图 2.5: 递归神经网络 (RNN)[2]

2.1.1.7 激活层（activation）

激活层用于处理被加权后的神经元数据，用于筛选其中的活跃神经元，也将输出神经元的值规约到一个范围内。常用的激活函数包括 Sigmoid，Tanh，ReLU。表 2.1中列出了一些常用激活函数的公式，其中 L-ReLU，P-ReLU[43] 以及 R-ReLU 都是 ReLU 函数的变种。

2.1.2 递归神经网络（Recurrent Neural Network, RNN）

递归神经网络是一类人工神经网络，主要用于处理文本，手写字体，语音等输入长度可变的时序列数据。RNN[2] 可以描述动态的时间行为，其在计算某一时刻 t 的输出是，除了 t 时刻的输出之外，还会接受 t-1 时刻的隐层输出，即过去时的状态。单纯的 RNN 无法处理随着不断的递归，权重指数级爆炸或者消失的问题，为了解决这个问题，人们引入了长短期记忆的概念，用来保存一些“记忆”，构建出的网络被称为长短期记忆神经网络（Long short-term memory, LSTM）。LSTM[90] 是一种特殊类型的 RNN，能够学习长期的数据间的依赖。

2.1.3 网络结构

最早的神经网络模型主要是线性的，但是随着算法的不断发展，逐渐出现带有分支的网络模型，其结构也越来越复杂。我们将网络结构分成 3 类，第一类是顺序网络结构，即最传统的神经网络，比如 LeNet5[1]，AlexNet[25] 和 VGG[13] 等。第二类是图状网络结构，比如 GoogleNet[29] 和 ResNet[4]；第三类是有动态规模的网络结构，比如 Fast RCNN[91]，以及 Faster-RCNN[92]。

顺序网络结构。顺序神经网络中层按照顺序结构进行排列，后一个层只和前一个层相连。一个层只有一块输入，一块输出。典型例子包括：LeNet5，AlexNet，

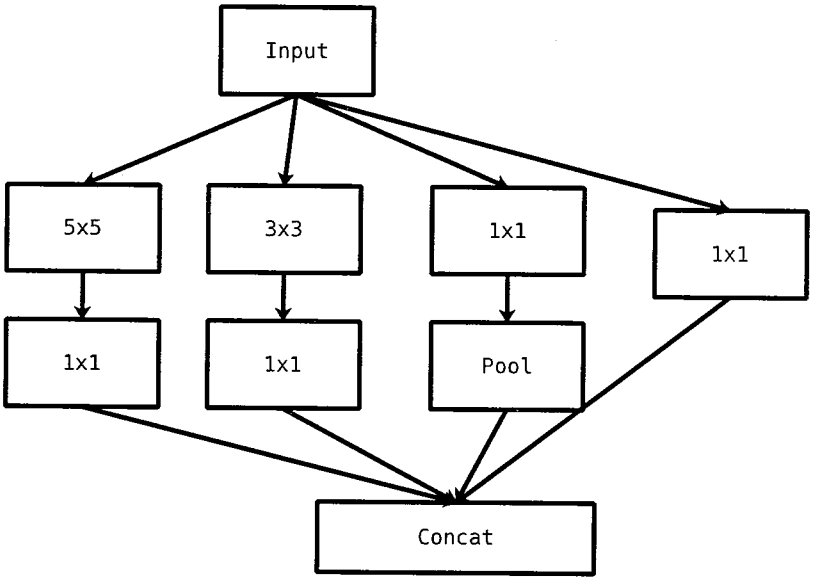


图 2.6: Inception 结构[3]

VGG。

图状网络结构。图状的神经网络中的层不是完全按照顺序组合在一起的，而是类似图的连接方式，存在多输入多输出的情况，比如 GoogleNet[3] 中的 inception 结构，和 ResNet[4] 中的 block 结构。图 2.6是一个 inception 结构，其中输入数据被经过不同的卷积核，i.e., 5x5，3x3，和 1x1，进行处理，得到不同的输出，然后这些输出会被按照特征图的方向被拼接在一起。用这种方法，可以减少权值的数量。

图 2.7中是 ResNet 中的一个基本块的残差传输的结构。残差不是以连续的方式逐层传递，而是跳过几层，跨层传递，这样导致某一层和两个层相连接，也就构成了图的结构。现在，越来越多的网络是通过图的方式构建，图的结构也变得越来越复杂，在这种情况下，神经网络的编程框架的编程接口也逐渐在向图的表达靠拢，TensorFlow[64]，MXNet[70] 这样的基于图结构的编程框架变得越来越流行起来。

动态规模的网络结构。在一些任务，比如图像识别中，我们需要首先标出识别目标的位置，然后在识别过程中动态的将识别到的目标从图片中抠出来，然后再进行图像识别[92–94]。这种情况下，抠出来的图像的大小是不定的，但是卷积神经网络只能处理固定大小的输入图像，因此我们需要对选择出来的图像进行处理之后再送入卷积神经网络中进行分类。这样的操作在图像识别的网络中会被用到。这些网络中用到的，将不同大小的输入图像处理成同样尺寸输入的算法是 Region of interest Pooling（ROI-Pooling）[91]。ROI-Pooling 根据输入输出的大小，动态的计算出池化核大小，然后计算出一个确定的输出特征图。

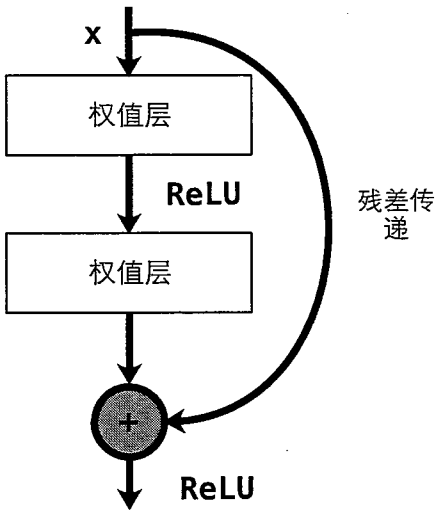


图 2.7: ResNet 中残差传递的基本块[4]

2.1.4 压缩神经网络

随着神经网络结构越来越复杂，导致其计算量越来越大，但是研究人员们发现这些计算中有很多都是冗余的，图 2.8中是一个稀疏连接的全连接层，图 2.8(a) 是原始的全连接层，图 2.8(b) 是进行剪枝后的神经元连接，很多值为 0 的权值连接被去掉，因此减少了计算量。激活后的神经元中，通常会包含大量的零值，通过在训练过程中添加训练条件可以让网络中的零值变得更多，从而使得网络更加稀疏。根据这个特点，人们对神经网络模型的权值和神经元进行压缩，去掉冗余的计算和存储。常用的方法包括剪枝和量化，来减少运算量和数据存储。

比如，DeepCompression[95] 是 Han 等人 2015 年提出的神经网络压缩算法，其中运用到三个阶段的压缩，最多可将原始权值压缩 35×-49×。首先在训练时对权值进行剪枝，只留下重要的权值，减少权值的个数，这一步压缩可以将 AlexNet 的权值减少到原来的 9×-13×；第二步量化权值，每一个权值用 4 比特的索引表示，使得权值的数值可以共享，将权值减少 27×-31×；第三步对权值和权值的索引进行霍夫曼编码，又将存储大小减少 35×-49×。利用神经网络的稀疏性，研究人员们还设计了相应的定制加速器来加速压缩后的网络的计算，可以将计算时间大幅减少。

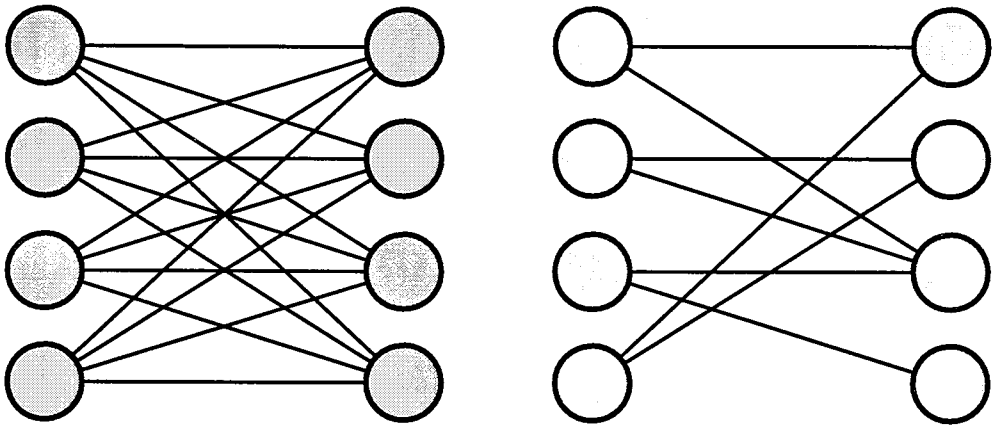


图 2.8: 神经网络剪枝

2.2 深度学习加速器

前文中，我们介绍了深度学习算法，各种不同的层和连接方式，从算法和应用的角度对编程框架的设计提出了要求。本节中，我们将介绍各类深度学习加速器，即本文编程方法所关注的后端设备，其体系结构的设计决定了可以支持的功能，很大程度上决定了深度学习编程软件的设计。我们按照加速器的架构可以根据数据流类型分为 2 种类型，基于向量处理的处理器[5,8,9,57,66,85]，以及基于空间数据流的处理器[58,63,65,87,96]，而根据它们提供的功能操作的粒度又可以分成三种类型，一种提供向量操作[5,8,57,97]，一种提供矩阵操作[9]，另一种是提供张量操作，如卷积操作等的加速器[58]。

2.2.1 向量操作处理器

DianNao[5] 是一款用于处理神经网络算法的加速器，其可以支持卷积，池化，全连接等常用的深度学习算法。DianNao 文章中，作者对最消耗计算资源和访存资源的卷积算法和全连接算法进行了分析，对其计算中的多层循环进行分解，找到数据局部性，并将这种局部性映射到了硬件设计上。这种利用数据局部性和数据重用提高效率的思想在 DianNao 系列的其他加速器上也有所体现。图 2.9中是对全连接层算法的循环分块。经过分块之后，我们对输入数据进行了重用，从而减少了数据访存。

图中，左侧的伪代码描述了一个朴素的没有任何分块的全连接层计算，这种情况下，计算每个输出时，都需要加载一个输入值，即需要 NxM 次对输入的加载。

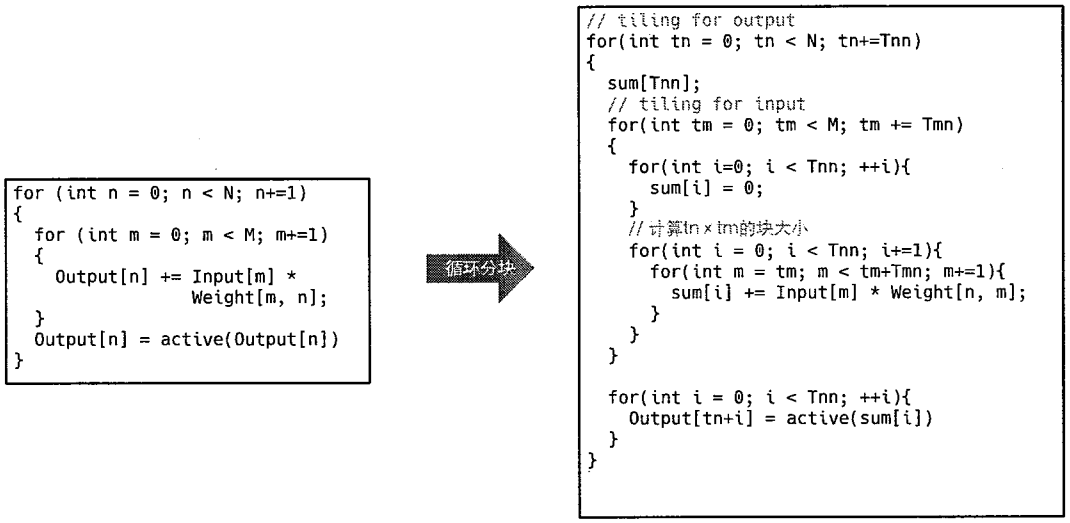


图 2.9: 全连接层的循环分块[5]

然而，每个输入实际上可以被多个输出数据重用，因此，为了增加数据重用，我们对其输入和输出分别进行划分。经过分块之后，如右侧代码所示，我们将输入分成大小为 Tnn 的块，输出分成大小为 Tmn 的块。我们一次计算一个输出块和一个输入块之间的乘法，这样一个输入数据被加载后，会被重复使用 Tnn 次。利用这种方法，我们将输入数据的加载次数降低到：TNxTMxTmn，其中 TN 和 TM 是输出和输入块的数量。DianNao 包含多个流水级，可以一次性计算 TNxTM 个乘法，然后将输出累加，并且进行激活运算。DianNao 中 TN 和 TM 的数量设置为 16x16 个，即包含 256 个乘法器。此外，由于神经元和权值每次读取的数据量不同（神经元每次读取 TM 和数，权值每次读取 TNxTM 个数），为了每次计算提供足够的数据，DianNao 为神经元和权值的读取安排两块不同的片上缓存，以提高带宽。

虽然 DianNao 可以处理不同规模的神经网络算法，但是当网络规模较大时，数据必须被存放在主存中，先被加载到片上缓存之后再进行计算。频繁的访存操作会极大的限制 DianNao 的效率。从机器学习算法的角度来说，一个网络中可能会有十亿个参数，这已经是非常大的规模，但是从硬件角度来说却并不算是非常大。比如 10 亿个数，如果每个数需要 64 位来存储，大约需要 8GB 的容量。虽然，8GB 的数据要放在一个芯片上的话确实太大，但是我们可以设想将这 8GB 的数据映射到多个芯片上，每个芯片容纳一部分数据，多芯片组成起来就可以在片上放下全部的数据，而不需要主存的参与。DaDianNao[57] 的提出就是基于这样的思路。DaDianNao 由一组节点（node）组成，每个节点中有一块芯片，芯片的结构和 DianNao 类似，节点按照网状连接。每个节点都包含大量的存储空间（采用 eDRAM 作为存储介质，主要用于存储权值数据），神经元计算单元（和 DianNao

类似，包含多个流水集的计算单元，比如乘法，加法树，线性插值等）。一个 64 个节点的 DaDianNao 结构可以获得和 NVIDIA K20M GPU 相比 450.65x 的加速比，以及 150.31x 的能耗节省。以上的加速器只针对机器学习中的一类算法，即神经网络算法，进行了加速，并不支持传统的机器学习算法，比如 KNN，Kmeans 等。PuDianNao[59] 分析了 7 种常见的机器学习算法，包括经典机器学习算法和神经网络算法。PuDianNao 对常用的机器学习算法的计算模式和访存模式进行了分析，设计了 7 个流水级，并且包含了一个 ALU，用于处理机器学习算法中的更丰富的计算模式，比如决策树和朴素贝叶斯算法中广泛用到的计数（counting）。

2.2.2 矩阵操作加速器

Cambricon[9] 是 2016 年由 Liu 等人提出的神经网络指令集，也是首个针对神经网络加速器的指令集。其观察到神经网络算法中存在的超过 90% 的标量计算都可以进行向量化，用矩阵/向量计算来完成，而剩下的指令可以通过标量指令完成。因此，Cambricon 指令集更多的利用数据间并行（Data-level Parallal, DLP）而不是指令并行（Instruction-level Parallel, ILP）来对算法进行加速。指令集中包含了包含了 43 条指令，分成计算指令，逻辑指令，控制指令和数据搬运指令 4 类。可以实现对矩阵，向量，以及标量之间的加减乘法，指数，期望，对数等运算。

2.2.3 张量操作加速器

ShiDianNao[58] 是一种低功耗的，基于脉动阵列机的卷积神经网络加速器，其主要利用了 2 维卷积空间上不同输出数据对输入数据的共享，减少了对输入数据的搬运。DianNao 主要针对深度神经网络算法（全连接层），而 DaDianNao 主要针对的是私有核的卷积操作和全连接层，这些操作的主要优化点在于减少数据的读取。而 ShiDianNao 主要用于加速的是共享核的卷积神经网络，通过充分利用卷积运算中对输入数据间的共享来降低功耗。由于卷积核的共享，使得权值数量大幅减少，可以全部放在芯片内部，不需要在计算过程中从主存中读取数据。ShiDianNao 的速度是 NVIDIA K20M GPU 的 30 倍，而能耗少 4688.13 倍。ShiDianNao 的二维矩阵结构天然的支持二维数据的带滑动窗口的计算，因此可以直接提供二维的卷积运算和池化计算，即神经网络算法中最常用到的两种操作。

2.2.4 稀疏神经网络加速器

由于神经网络算法具有稀疏性和可压缩性，而传统的计算设备，比如 GPU，无法很好的利用这种稀疏性对计算进行加速，因此，人们提出了可以更好的利用稀疏性的神经网络加速器。Han 等人提出的 EIE[66] 结构，可以处理稀疏的矩阵乘法，从而加快全连接层的计算速度。Cambricon-X[8] 是 2016 年提出的首个可以处

理稀疏卷积计算的加速器。EIE[66] 也是一种基于向量处理的加速器，可以对全连接层的权值进行压缩。

2.3 深度学习编程框架

深度学习框架的出现促进了深度学习算法的发展，降低了算法开发的成本。深度学习框架向算法开发者提供了友好的编程接口，让程序员能够快速的构建算法。同时，深度学习框架的后端通常会支持多个平台（Windows，Linux，Android 等），以及多种设备（CPU，GPU，领域专用加速器等），良好的可移植性进一步降低了用户的开发成本。

如图 1.2所示，深度学习编程框架一般分为多个层次，最上层的是编程接口，用于定义神经网络的拓扑结构，一般来讲，数据是按照多维数组的形式组织的，每一个算子的输入和输出都是多维矩阵。编程接口的设计除了可以影响编程上的灵活性，同时也影响着底层的优化方式。比如，用数据流图来表示一个网络，则可以在图的层面对网络进行优化，但是用层来描述的接口则无法进行计算图的优化。从编程接口的角度，我们可以将框架分成三类，基于层的框架，基于图的深度学习框架，以及深度学习领域专用语言（Domain specific language, DSL）。下面我们将分别介绍这三类架构。

2.3.1 基于层的深度学习编程框架

基于层的编程框架在构建网络时，是以层为单位，每一个层表示一个粗粒度的算子，比如卷积层，池化层，全连接层等。每个层有一些需要配置的参数，框架提供一个接口，用户通过写接口的配置文件，指定层的计算顺序，即构建了网络。配置文件或被框架解析为对应层的函数调用，通过反复调用这些层的函数接口执行网络。基于层的深度学习框架倾向于只提供粗粒度的算子用来构建网络，这样做的好处是，每一个层的内部实现上，库的开发者们可以进行充分的性能优化，因此可以提供更好的运行效率。但是，相应的，其劣势也很明显，就是缺乏灵活性，当需要增加新的算子时，用户需要从头实现一个新的层而无法重用已经写好的其他算子，同时，通过配置文件的方式编程，对用户并不友好，尤其是当层数量增加，或者链接非常复杂的情况下，这样的写法很容易写错。此外，配置文件的表达力也收到限制，比如循环和控制流的跳转就难以用配置文件来表示。

最典型的基于层的深度学习框架是由伯克利大学的 Jia 等人提出的 Caffe[88]，卷积神经网络编程框架。其前端配置采用 prototxt 文件格式来配置网络拓扑结构，和生成相应的参数定义，用户通过编写 prototxt 文件，对每一个层的参数以及训练参数进行配置，之后运行中，框架程序会按照解析出来的网络结构信息，进行神经网络的训练或者推理。

2.3.2 基于图的深度学习编程框架

基于数据流图（data flow graph）的深度学习编程框架利用节点（node）和边（edge）构造出来的有向图来描述计算过程，即数据流。其中，每一个节点表示一个运算操作，或者表示一块数据的输入起点或者输出终点。边则表示节点之间的输入/输出关系。数据被表示为多维数组（张量）的形式，数据在节点之间沿着边的方向传播。通过一个节点时，数据就会作为该节点运算操作的输入被计算，计算的结果则顺着该节点的输出边流向后面的节点。一旦输入端的所有数据准备好，节点将被分配到各种计算设备，完成异步并行地执行运算。一下是 4 种最流行的基于数据流图的机器学习库[98]。

Theano[99] 是一个基于 Python 语言的机器学习库，其是最早采用计算图构造机器学习计算表达式的编程库，启发了很多其他编程框架。Theano 要求用户先定义计算图结构，链接方式，构成数学表达式，其中表达式的输入输出可以是标量，向量或者多维数组，这些都是机器学习算法中的常用数据类型。Theano 可以处理大量的运算数据，在 CPU 上，其效果可以和 C 代码相媲美。此外，Theano 也可以利用更高效的计算设备，比如 GPU 进行计算，以获得超过 CPU 几个数量级的运行速度。Theano 的另一个特点是利用代数运算那系统 (Computer algebra system, CAS)，将其与优化编译器相结合，就可以为许多数学运算生成定制的 C 代码，进行更加充分的优化[98]。

Tensorflow[64] 也是一款基于数据流图计算的机器学习编程框，由 Google 公司提出，具有编程灵活、支持多种计算平台，在异构系统上部署方便等特点，同时还支持多种语言接口，比如 python, c++ 等。TensorFlow 可以很好的支持跨平台运行，程序员只需要修改很少量的代码，就可以将在 CPU 上执行的代码移植到 GPU 平台上进行运算。此外，Tensorflow 还可以支持自动异构分布式计算，它的模型能够运行在不同的分布式系统上，系统可以包括多个 GPU、CPU、手机节点等[98]。MXNet[70] 也是一款基于计算图模型的机器学习编程库。它也能够支持跨平台（GPU，CPU），以及多 GPU 配置。MXNet 提供了高级模型来构建网络中的块，让编程更加方便。而且还可以在常见的硬件设备上运行（包括手机、服务器等）。MXNet 提供更多的编程接口，除了 Python 之外，还提供了对 R、Julia、C++、Scala、Matlab 和 JavaScript 的编程接口[98]。

Torch[71] 是另一款基于计算图模型的机器学习框架，主要用于科学计算，包括机器学习，深度学习等。Torch 早期采用 Lua 作为编程接口，主要原因 Lua 的底层代码都是用 C++ 实现的，而 Lua 和 C++ 之间的集成调用接口友好，开销很低，因此可以更好地支持内嵌 CUDA-C 优化代码[98]，但是，随着 Python 语言的日益流行和其开发者的活跃程度，现在 Torch 也和 TensorFlow 和 MXNet 一样将接口转向了 Python。发布了新的项目 Pytorch。

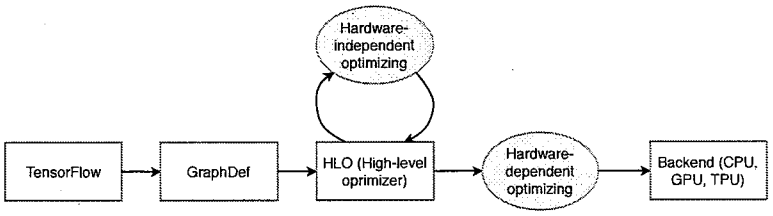


图 2.10: XLA 编译流程[6]

2.3.3 深度学习领域专用语言

随着深度学习算法的不断演进，人们对描述算法的编程接口的灵活性和开发效率要求越来越高，过去的算子已经不够用，经常需要开发新的算子，而开发算子的工作本身需要耗费大量的人力，因此逐渐成为瓶颈。为了解决这个问题，人们提出了深度学习领域专用的编程语言和中间表示，其可以提供更好的灵活性，减轻程序员的编程负担，但同时也给编译的性能带来挑战。

此外，现在的深度学习领域专用语言，主要是针对 CPU 和 GPU 后端进行的优化，对于深度学习加速器的支持则非常局限。本节中，我们将介绍几种不同的深度学习领域专用语言。

2.3.3.1 XLA

XLA（Accelerated Linear Algebra）[6] 是 Google 提出的一种领域专用的编译器，用来对 TensorFlow 的计算进行加速，其可以有效的减少内存使用，提升运算速度，提供平台可移植的性能。XLA 提供 JIT 编译和 AOT 编译两种编译方法，可以支持多种后端，同时开放了集成不同后端的接口，包括深度学习加速器（即 TPU[63]）。

XLA 中提出了一种领域专用的中间表示，HLO（High leve optimizer IR），用于 XLA 内部的计算图描述。如图 2.10所示，当 XLA 被调用，它会首先接收一个从 TensorFlow 的计算图表示，之后这个图会被转化成 HLO IR 描述的中间代码，之后，XLA 的编译器会对这个中间代码进行优化。最后，针对不同的后端，编译器会从 HLO IR 生成不同的后端相关的代码。比如对于 GPU 和 CPU 的后端，XLA 会为其生成 LLVM，然后调用 LLVM 的接口生成二进制可执行文件。XLA 的 HLO IR 中定义了大量的带有高级语义的原子操作（HLOInstruction），输出输出都是 Tensor，每个算子通过手工进行优化，因此可以能够提供较好的性能。

2.3.3.2 TVM

TVM[7] 是一个为深度学习设计的软件栈，其结构如图 2.11所示。TVM 的输入是计算图，首先在计算图的层次对算子进行融合优化，减少算子的个数，来节省

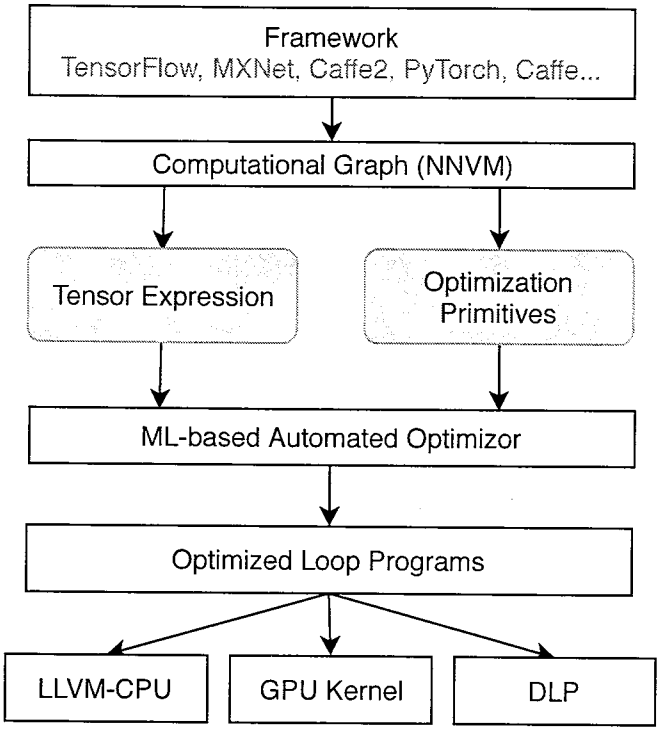


图 2.11: TVM 架构图[7]

算子的内核调用开销。之后，TVM 借鉴了 Halide[72] 语言将计算和调度分开的思想，提供了一种张量描述语言，用来实现一个具体的算子（即描述计算），同时提供一系列优化原语（包括），用来对每个算子进行优化（调度）。每个算子会被翻译成 TVM 的中间语言，最后生成底层设备相关的代码。

2.3.3.3 Tensor Comprehension

TensorComprehension（TC）[73] 是一种基于 Tensor 描述的领域专用语言，用来描述深度学习中用到的计算模型，主要用来帮助用户更快的定义一个新的算子。由 TC 构成的算子会被转化成 Halide IR。TensorComprehension 引入了 Polyhedral 模型，对循环进行优化，包括自动的 tiling，调转循环顺序等。同时，TC 还引入一个基于遗传算法的 Autoturn 机制，对可调整的参数进行自动调优，找到最好的调优策略。

2.3.3.4 DLVM

DLVM[100] 是一个深度学习领域专用的中间语言以及一系列编译工具。DLVM 的核心语言是一个中间表示，采用了静态单一分配 (SSA)，控制流图，高级类型，比如 Tensor，以及一系列高级算子线性代数操作。同时，DLVM 中还包括了大量

的领域专用分析和转化，比如代数简化，线性代数融合等。

2.4 本章小结

本章中，我们介绍了本文研究的背景：深度学习算法的发展，常用的算法和网络结构，这些都是本文所提出的软件栈需要去实现的具体程序；深度学习加速器架构以及指令集，这是本文软件栈所使用的后端设备，其架构对上层软件接口的设计至关重要；以及深度学习编程框架，即我们希望可以支持的前端，目前主流深度学习应用程序都是都是通过各种深度学习框架开发，深度学习社区已经积累了大量应用程序，对于一种新型设备来说，最重要的是要利用现有的资源，在不改变应用程序的情况下将加速器集成到框架中去。

第 3 章 深度学习处理器高性能库

本章中，我们针对深度学习加速器设计一款高性能库（DLPlib），作为深度学习加速器的编程接口。我们首先介绍提出 DLPlib 的动机和设计思路，之后介绍高性能库的整体结构，数据结构和 API 设计。之后，我们介绍 DLPLib 的编程模型，以及如何将 DLPlib 嵌入到流行的深度学习框架 Caffe 中。最后我们在 Cambricon-X[8] 上对 DLPlib 的性能进行评估[101]。

3.1 设计思路

如今，深度学习算法被人们用来解决各种问题，一些过去难以攻克的难题，通过深度学习的方法都取得了很大进步。比如，图像识别[91,92]，图像分类[4,25]，手写数字识别[1]，自然语言处理[16] 等。随着深度学习的模型变得越来越复杂，计算量也变得越来越大，传统设备已经提供不了足够的计算性能，在此情况下，人们提出深度学习处理器来处理深度学习算法。然而，由于程序员们只能通过手写底层指令的方式写程序，在深度学习处理器上进行开发的效率非常低下，人力开销巨大，并且可能造成很多重复劳动。因此，要提升深度学习处理器上的开发效率，我们需要找到一种编程方法，能够解放人力，提升算法开发效率。

一种可能的解决方案是采用高级的深度学习框架作为深度学习加速器的前端，将后端加速器深度集成到这个框架中。比如，Google 提出的张量处理单元（Tensor Processing Unit, TPU[63]）采用机器学习编程框架 TensorFlow[64] 作为前端编程接口。这种编程方法的好处在于，用户的学习成本低，由于 TensorFlow 的易用性较好，用户可以很轻易的在 TPU 上进行编程。同时，由于 TensorFlow 支持跨平台，用户写一份代码，可以在不同平台（GPU，CPU 和 TPU）之间迁移，大大减少开发成本。

但是，将深度学习加速器直接集成到高级深度学习框架中也有不利的地方。因为直接使用最高级的编程接口，而不暴露中间的较低级操作，深度学习加速器无法集成到其他深度学习框架中。尤其是，在现在深度学习框架蓬勃发展的时候，深度学习加速器作为一个底层设备，应该和 GPU，CPU 等传统设备一样，被更多深度学习框架和开发者们利用。

从后端设备的角度出发，我们将编程方法定位在低层库的层次，而不依赖于任何一种编程框架，类似英伟达公司提出的 cuDNN[68] 深度学习加速库。低层次的高性能库是前端无关的，因此可以集成到不同的深度学习框架中，令深度学习加速器得以被利用。在设计深度学习加速库的过程中，我们主要考虑以下三方面问题。

- 数据结构设计。高性能库应该只包含少量的数据类型，这些数据类型应该能够表示深度学习算法中出现的数据类型，同时也是底层硬件可以支持的。更少的数据类型便于我们对数据结构进行优化，同时，也让用户更易于掌握，进行编程。在保持数据类型尽量少的情况下，我们也需要考虑到可扩展性。

- 算子的选择。算子的选择是效率和灵活性之间的平衡。提供充分优化过的高级算子则效率高，但是对算法实现上则有限制，相反如果提供最基本的算子，对于实现算法会很灵活，但是运行效率就会相对低。

- API 设计。API 的设计涉及到对数据结构和算子的支持，这种支持应该灵活并且容易实现。为了让深度学习框架可以利用深度学习处理器，我们在设计 API 的时候，还应该考虑到将深度学习处理器集成到深度学习框架（比如 TensorFlow[64]，MXNet[70]，Caffe[88] 等）中的便利性。

3.2 总览

DLPlib 主要包含两个主要部分，数据结构和算子。DLPlib 中，数据被表示为张量（Tensor）或者过滤器（Filter），而算子被看作是对张量操作的转换。

算子是一系列用来构造神经网络拓扑结构的操作。DLPLib 中，算子是一组互相独立的操作单元，通过调用一系列算子，我们就可以构建出一个神经网络。这些算子是串行执行的，加速器会首先投入全部的资源来运行一个任务，直到这个任务运行结束之后，才会调用下一个任务。这样的串行执行过程，保证了正确性，尤其是当对算子的调用变得越来越复杂，同时也让我们可以专注于对每一个算子的优化。

图3.1中描述了一个 3 层的神经网络结构。张量数据开始被存放在 CPU 的内

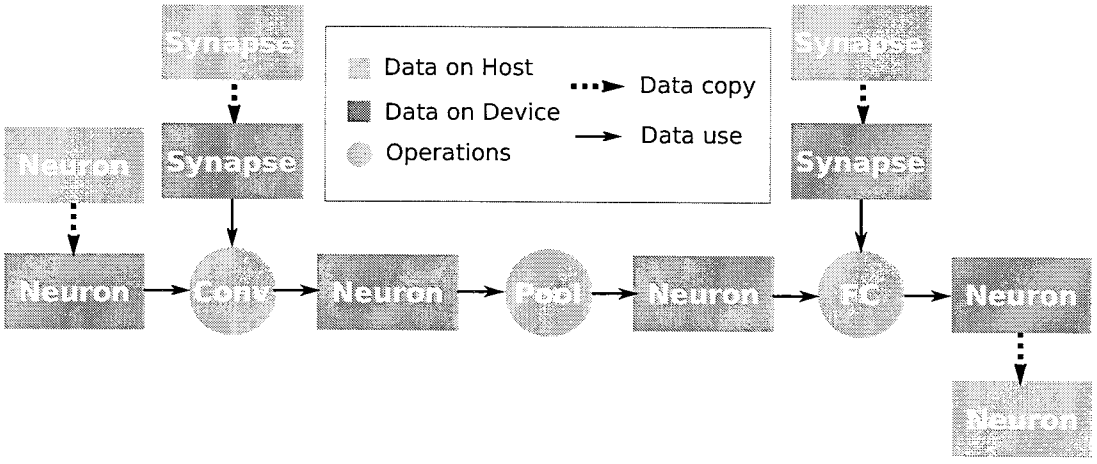


图 3.1: DLPlib 中一个三层神经网络的数据流

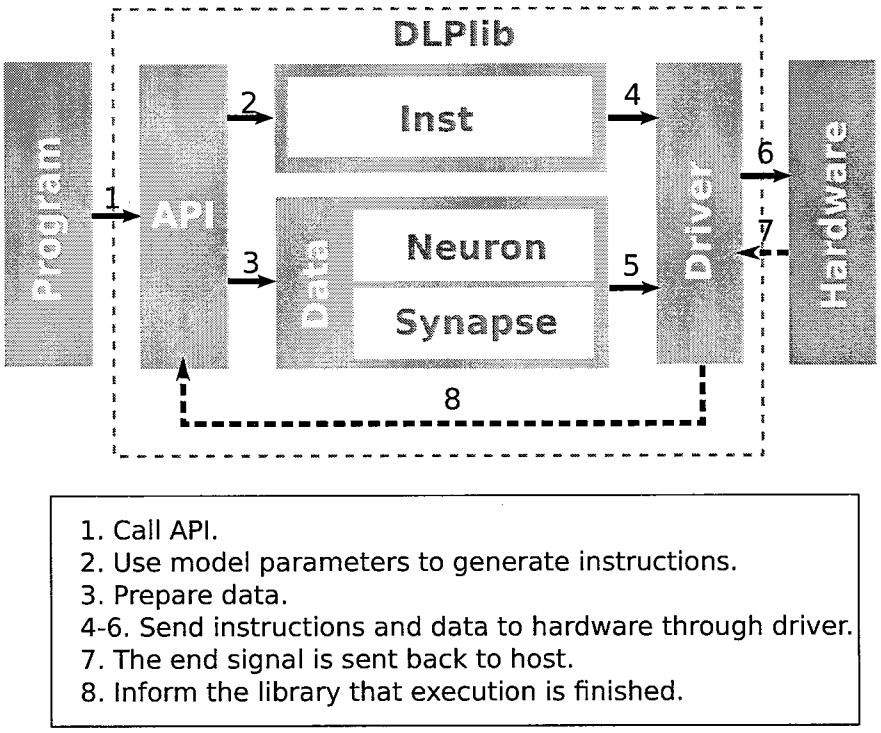


图 3.2: DLPLib 调用过程

存中（host 端），然后被拷贝到设备上进行计算，计算顺序进行，顺序相邻的两个算子会共享相同的一块内存，前一个操作的输出就作为第二个操作的输入。比如，图中卷积层的输出神经元可以作为后面的池化层的输入。对于有突触（权值）数据的层来说，比如卷积层，突触数据也需要在计算发生之前拷贝到设备上。当网络中的每一层计算结束时，最后的输出数据应该拷贝回 CPU，作为数据结果。

整个 DLPlib 的调用过程如图 3.2所示，图中所表示的是操作一个正向的卷积层的调用过程。具体步骤如下：

1. 主机端程序被调用，从而调用库。
2. 通过程序中对 API 的调用，我们可以构造出一个神经网络模型，所有的相关信息都会被打包，为之后的指令生成做准备。
3. 通过分析 API 传递的参数信息，库会生成相应的指令序列，这些指令通过驱动传送给硬件设备，也就是深度学习处理器。
4. 通过调用内存管理的函数接口，数据（比如：输入输出的神经元）也会通过驱动被拷贝到深度学习处理器上。
5. 当执行函数被调用，开始信号被发送到设备上。当硬件设备接受到信号之后，指令会被执行。

6. 当深度学习处理器上的执行结束之后，加速器会发送一个结束信号给主机端，通知主机执行已经结束。之后主机会把计算结果数据拷贝回 CPU 端的内存上。这时整个调用过程就结束了。

3.3 高性能库设计

本节中，我们将具体介绍高性能库中包含的数据类型（张量，过滤器和描述子），以及高性能库中的算子选择。

3.3.1 数据类型

DLPLib 主要包含两种数据结构，张量（Tensor）和过滤器（Filter）。这两种数据类型是来自神经网络算法中的两种核心概念–神经元（Neuron）和突触（Synapse）。我们用 Tensor 来表示算子的输入和输出数据，而用 Filter 来表示突触数据。很多深度学习框架中，神经元和突触都是用一种数据类型来表示的，比如，在 TensorFlow，MXNet 这样的框架中，所有数据都表示为 N 维的数组，Caffe 中则是用 Blob 类封装数据，并不会对神经元和权值数据进行具体的区分，但是在我们针对深度学习处理器的设计中，我们将这两者分开表示，表示成两种不同的数据类型。这是因为考虑到我们所使用的底层硬件深度学习加速器通常会采用两种不同的片上缓存来存储这两部分数据。比如，Cambricon-X 的设计中将神经元和突触分别存放在不同的片上内存上–神经元放在神经元缓存（neuron buffer，NB）上，突触放在突触缓存上（synapse buffer，SB）。因此，为了在进行数据拷贝和分配的时候可以区分目标地址区域，我们在 DLPlib 中，也使用两种数据类型来存放神经元和突触。

3.3.1.1 张量（Tensor）

张量是一个稠密的 n 维向量（其中 $n \leq 4$），用来表示深度学习算法中的神经元数据和偏置数据，实际上，除了突触之外的所有数据都可以用张量来表示。内存中，一个张量中的元素的存储是一个一维数组，而在定义的时候则被看做是一个 n 维的数组。比如，卷积算子的输入输出就是两个 4 维数组，而全连接层的输入输出则是两个 2 维数组。

张量通过 6 个属性来描述，如表 3.1中所述。N，C，H，W 表示一个张量的 4 个维度的尺寸。其中 N 表示样本的数量，C，H，和 W 分别表示一个样本中特征图的数量，高度和宽度。数据格式表示数据排布的存放顺序。字母的顺序表示一个张量数据的维度存储顺序，比如，NCHW 表示这个张量会临接的存放 W 维度的所有数据，之后是 H，C 和 N。数据格式会影响计算时候数据存取的效率，因此我们将其作为一个可以设置的参数放在描述张量的属性里。数值类型表示计算

表 3.1: Tensor 数据类型属性参数

参数	描述
N	样本数
C	特征图数量
H	特征图高度
W	特征图宽度
数据格式	数据维度存放顺序（NCHW，NHWC）
数值类型	计算中使用的数据的精度和类型（32 位浮点，16 位浮点）

表 3.2: Filter 数据类型属性参数

参数	描述
OC	输出特征图数量
IC	输入特征图数量
K_h	核高度
K_w	核宽度
数据格式	数据维度存放顺序（NCHW，NHWC）
数值类型	计算中使用的数据的精度和类型（32 位浮点，16 位浮点）

中使用数据的精度和类型，包括 16 位浮点和 32 位浮点。由于稀疏神经网络和量化神经网络的出现，一个网络中的数据可能不只有一种，而可能由不同精度的数据构成，因此我们需要在描述数据的时候也对数值类型进行描述。为了简化表达，我们只提供 4 维的张量数据，维度少于 4 的数据，比如 2 维和 3 维张量也用 4 维张量表示，只是将 4 个维度中的几个维度的长度设置为 1。

3.3.1.2 过滤器（Filter）

突触是神经网络算法中概念，它连接了不同的神经元，并且这个链接是带有权重的。在实际的计算过程中，突触也和一般数据一样被表示为多维数组，作为一个用于计算的操作数。DLPlib 中，卷积算子和全连接算子的突触权值被表示为过滤器的数据类型。过滤器类型和张量类似，也是被存储为一个 1 维数组，但是被作为 n 维数组使用。表 3.2中列出了过滤器数据类型的 6 个属性参数。其中，OC，IC，K_h，和 K_w 分别表示输出特征图的个数，输入特征图的个数，核的高度和宽度。

DLPlib 中，数据（包括张量和过滤器）需要在使用之前拷贝到设备的内存中。这个拷贝过程由 DLPlib 提供的内存管理函数完成。

Pos	0	1	2	3	4	5	···	31
NB	N0	N1	N2	N3	N4	N5	···	N31

(a) Neuron (Tensor)

Pos	0-15	16-31	···
PE0	S0-S15	S256-S271	···
PE1	S16-S31	S272-S287	···
···			
PE15	S240-S255	S496-S511	···

(b) Synapse (Filter)

图 3.3: 全连接层数据摆放方法

因为引入了过滤器结构，神经网络的输入数据和权值数据就可以分开处理，在指令生成的同时将数据摆放成加速器可以直接使用的格式，以增加数据局部性。比如，一个输入为 32，输出为 32 的全连接层，输入神经元可以直接顺序的摆放，如图 3.3-(a) 所示，这些数据最后会被加载到神经元缓存上，而权值数据，即图 3.3-(b) 中是权值数据的摆放方式，因为权值数据需要被加载到不同的运算单元上，因此需要交叉进行摆放。如果不对权值进行重新摆放，则需要利用多条指令分别加载片上数据，这会导致局部性非常差。

3.3.2 算子

DLPlib 中的另一个重要结构是算子（Operator），算子是 DLPlib 中的基本计算单元，封装了深度学习算法中的常用算法以及矩阵，向量计算。在选择算子时，我们希望这些算子足够具有代表性，可以充分涵盖常神经网络算子，而且尽可能的减少冗余。我们统计了 2012-2016 年机器学习顶级会议（CVPR, EMNLP, ICML, ICRA，以及 NIPS）中出现的深度学习层的算法，它们出现的次数和比率，如表 3.3[102] 所示。

此外，由于神经网络算法中的大部分计算都可以聚合成矩阵/向量操作来完成[9]，因此，我们还提供一组矩阵向量操作，来实现新的算法。为了支持各种带有分支的网络结构，DLPlib 还提供一系列对数据的拼接，分割，变形等操作，这些数据变形操作的选择依据主要是深度学习框架中提供的功能。表 3.4中列出了 DLPlib 中已经支持的算子。

表 3.3: 2012-2016 年 CVPR, EMNLP, ICML, ICRA, 以及 NIPS 会议中不同层算法的出现次数统计

算法	出现次数	比率 (%)
卷积	452	19.28
池化	440	18.76
全连接	473	20.17
LRN	181	7.72
ReLU	425	18.12
Sigmoid	90	3.84
Tanh	100	4.26
反卷积	19	0.81
反池化	16	0.68
LSTM	133	5.67
BN	16	0.68

表 3.4: DLPlib 支持的算子

类型	算子	描述
神经网络计算	Convolution	用于提取图像信息
	Pooling	用于对图像数据进行降采样
	LRN	对数据 channel 方向进行归一化
	Fully-connected	对向量形式的数据进行分类
	BatchNorm	用于将输入数据归一到合理的范围内
矩阵向量计算	Matrix multiplication	矩阵乘法
	Vector Add/Sub/Mul	向量加法，减法，乘法
数据操作	Concat	将多个 Tensor 数据拼接成一个 Tensor
	Reshape	改变 Tensor 或者 Filter 数据的形状

3.4 高性能库实现

本节中，我们将对 DLPlib 的实现方式，各个模块设计，以及运行时过程进行介绍。

3.5 API 和编程模型

DLPlib 采用一种基于描述子的 API，数据类型（张量和过滤器）和算子都通过描述子来定义。比如，一个张量数据可以通过一个张量的描述子，和一个指向内存首地址的指针来描述。

图 3.4中描述了 DLPlib 的编程模型，该模型主要包括三个步骤：1）对设备，内存和算子进行初始化设置；2）调用算子，也就是算子的具体执行；3）资源（包括内存和设备）释放。

- 初始化 (Initialization)。在调用设备之前，我们需要对设备，内存和算子进行初始化。首先，我们为 DLPlib 创建一个句柄（dlpHandle），可以用来指向硬件设备，并且保存关于硬件的上下文（context）信息。句柄是一个全局变量，算子和内存管理函数都会调用句柄来调用硬件设备。其次，所有的内存数据都需要在这一步骤进行分配。存放数据的张量和过滤器的描述子会在这一步骤中定义。DLPlib 为每一个描述子提供 create，set，get 和 destroy 函数，用于创建，设置，获取和销毁一个描述子结构。

- 执行 (Execution)。在设置了数据和描述子之后，我们就开始调用算子，为了保证计算的正确性，算子的调用是顺序的，算子和算子之间不存在并行和资源共享。

- 资源释放 (Release)。当计算执行结束，我们需要释放掉分配的资源，包括 DLP 上分配的内存，描述子和设备句柄。我们要求用户显示的进行资源释放。

我们用一个卷积层作为例子来说明 DLPlib 编程模型的使用方式。图 3.5中的代码可以用于实现一个卷积层。为了方便说明，我们忽略具体的参数，以及描述子的创建，用函数名表示整个调用过程。准备过程中，我们首先调用 createDlpHandle 函数创建一个 DLPHandle，之后分配一个输入张量，一个权值过滤器和一个输出张量。要调用具体的算子之前，也需要先构建算子的描述子，之后调用算子的执行函数 dlpConvolutionForward 进行计算，输入输出等数据作为参数传入算子中。最后，我们需要手动的调用释放函数，释放数据和句柄资源。

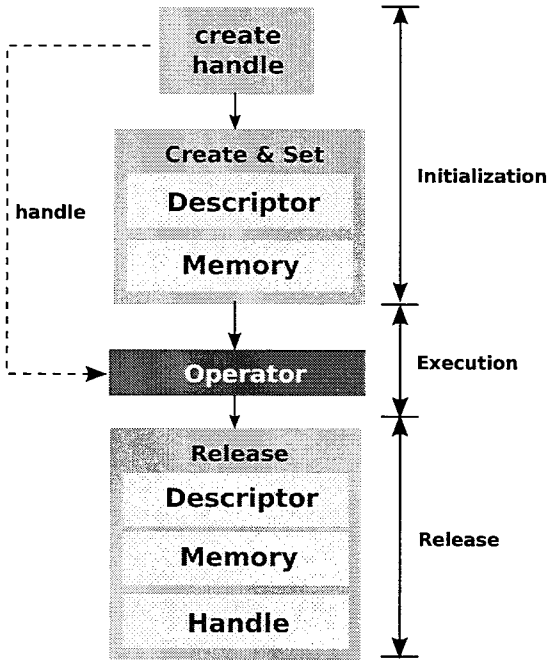


图 3.4: DLPlib 的编程模型

```
// device handle
createDlpHandle(handle);
// create data
dlpMalloc(input);
dlpMemcpy(input);
dlpMalloc(weight);
dlpMemcpy(weight);
dlpMalloc(output);
// create descriptor for
   convolution
createConvDescriptor(desc_conv);
// execute convolution forward
dlpConvolutionForward(desc_conv,
   input, weight, output);
// free
dlpFree(input);
dlpFree(output);
dlpFree(weight);
destroyDlpHandle(handle);
```

图 3.5: 用 DLPlib 实现一个卷积层

3.6 将 DLPlib 集成到编程框架

DLPlib 提出的其中一个目的是为了方便集成到编程框架中。本节中，我们介绍将 DLPlib 集成到框架 Caffe 中的方法。Caffe 是一种被广泛使用的卷积神经网络框架，Caffe 中的数据结构可以和 DLPlib 中的结构很好的对应，我们可以在不修改核心代码的情况下，将 DLPlib 集成到 Caffe 中。

3.6.1 框架结构

图 3.6中描述了 Caffe 一个正向执行过程，这个过程采用两个步骤。第一步 Caffe 会先构建整个神经网络拓扑结构，准备数据，并且分析各层的参数，之后的第二步会根据第一步决定的行为来执行。我们将具体介绍这个过程中的每个步骤，以及 DLPlib 如何集成到这每个步骤中去。

- 准备。Caffe 用 prototxt 格式的文件来描述神经网络的拓扑结构，其可训练参数（即权值数据）存放在 caffemodel 文件中。通过解析这两个文件，就可以构建出一个神经网络模型。对于有分叉的网络，比如 GoogleNet[29]，ResNet[4]。对于这类网络，Caffe 也会解析成顺序的层的序列进行计算，以满足拓扑排序。同时在，解析参数的过程中，各层的输出数据大小会被计算出来，用于之后进行分配数据。

  Caffe 中的数据是采用 Blob 结构来存放的，Blob 中包括了数据在主机端和设备端的地址，负责在主机和设备之间的拷贝工作。Blob 结构中包含着 N，C，H，W 四个维度信息，以及指向不同存储块的指针。神经网络模型中的数据，包括神经元，梯度，以及权值数据，都存放在 Blob 中。神经元和梯度会在层之间进行传递，被不同的层使用，而权值数据则作为每一个层的私有数据被存放在层内部，解析过程中，随着每个层的信息确定，Caffemodel 中

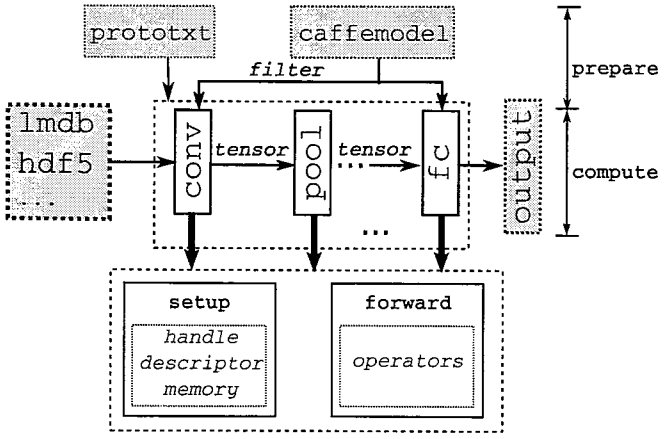


图 3.6: 将 DLPlib 集成到 Caffe 中

的可训练参数，即权值，也会被逐渐加载到每个层的 Blob 里面。当需要用到计算设备（比如 GPU）时，Blob 还需要将数据拷贝到即将要用到的计算设备上。

- 计算。模型构建，数据准备结束后，就开始计算过程。在这一步，Caffe 只需要根据之前构建出的层次拓扑顺序，调用每一层的计算函数，获得最后的结果。

3.6.2 集成

要将 DLPlib 集成到 Caffe 中，我们主要需要做 3 部分的修改：1）修改定义层参数的 proto 文件，添加专门针对深度学习处理器的参数；2）扩展 Blob 类，添加对深度学习处理器设备的支持；3）扩展层。

- 扩展 Prototxt。Proto 文件中定义的数据结构包含了所有层的属性。Caffe 中的后端用 Engin 结构指定，设备包括 cuDNN，Caffe（即 CPU）。作为一个新的后端，深度学习加速器也作为一个枚举类型添加到 Engin 中。

- 扩展 Blob。DLPlib 中的张量和过滤器数据都被包装在 Blob 结构中。我们在 Blob 类中添加一个指向 DLP 上内存空间的指针，用于索引，并扩展内存拷贝函数，以支持 DLP 和 CPU 之间的内存拷贝。

- 扩展 Layer。Caffe 将神经网络中的算法，比如卷积，池化等，包装在 Layer 这个类中。要添加新的设备实现，我们需要定义新的子类，重写 Layer 类中的几个虚函数（Setup，Reshape，以及 Forward 函数）。

  如图 3.6所示，Setup 函数会创建句柄（Handle），描述子（Descriptor）和内存（Memory），之后 Reshape 函数会对它们进行设置。比如，一个卷积层需要 5 个描述子—输入张量，输出张量，权值过滤器，偏置，和卷积算子。利用 DLPlib 提供的帮助函数，或者 Caffe 提供的 computeOutputDim 函数，我们就可以算出输出的特征图的尺寸。

  对于 Forward 函数，我们会调用 DLPlib 中的具体算子来进行计算。比如，要执行一个卷积层，我们会用卷积操作去获得最终的结果。由于所有的数据和参数（如：描述子）都已经在之前的步骤中设置好了，因此，我们可以直接调用算子的计算函数，进行计算。对于没有直接对应算子的操作，我们可以利用 DLPlib 中提供的矩阵向量算子来实现。

  Caffe 中的两个步骤，Setup（准备数据和网络参数）和 Forward（执行计算操作）可以直接对应到 DLPlib 的编程模型上。对于 DLPlib 来说，描述子的配置和内存的分配可以自然对应到准备步骤，而算子则可以在 Forward 函数

中调用。Caffe 还可以支持反向操作（Backward 和 update），但是由于我们的后端加速器 Cambricon-X 不支持反向计算，因此我们在设计 DLPlib 时不考虑这一部分的实现。

3.7 性能评估

3.7.1 实验环境

本节中，我们在 Cambricon-X 后端上对 DLPlib 进行评估，对照的 baseline 是手动优化实现的代码，高性能库的易用性和手写指令相比显而易见，因此我们主要针对性能（加速比）这一个方面进行评估。实验选用了 9 种 Benchmark（如表 3.5所示），包括 7 种单层算子（卷积层，池化层，LRN 层，以及全连接层），和两种大型网络（Alexnet 以及 VGG16）。

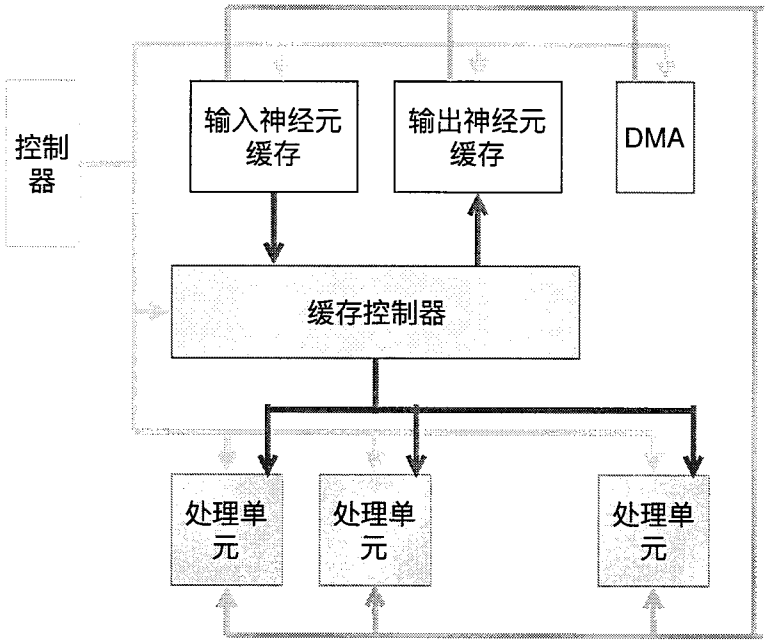


图 3.7: 评估 DLPlib 中所使用的深度学习处理器架构[8]

如前文所述，我们采用 Caffe 作为前端，使用 Caffe 社区提供的神经网络结构定义文件（prototxt 文件）作为输入。我们使用的后端 Cambricon-X[8] 是通过 Verilog 语言实现，使用 Synopsys Verilog Compiler Simulator（VCS）编译得到。Cambricon-X 的结构如图 3.7所示，该加速器包括 16 个处理单元（PE），每一个处理单元包含 16 个乘法器，一个 16 合 1 的加法树，以及一个 2KB 的突触缓存（Synapse Buffer，SB）。处理器上还包含两个片上神经元缓存（输入神经元缓存 NBin 和输出神经元缓存 NBout），每一个神经元缓存大小为 8KB，加速器的频率可以达到 1GHz。

表 3.5: Benchmarks

Benchmark	Op.	In (Size@FM)	Out (Size@FM)	Kernel	Stride	Pad	Active
Conv1	Conv.	112@96	112@384	3	1	1	-
Conv2	Conv.	112@96	112@256	3	1	1	-
Pool1	Pool.	55@96	27@96	3	2	0	-
Pool2	Pool.	224@64	112@64	2	2	0	-
LRN	LRN.	55@96	55@96	-	-	-	-
FC1	FC.	1@4096	1@4096	-	-	-	-
FC2	FC.	1@4096	1@1000	-	-	-	-
AlexNet	Conv.	224@3	55@96	11	4	0	ReLU
	LRN.	55@96	55@96	-	-	-	-
	Pool.	55@96	27@96	3	2	0	-
	Conv.	27@96	27@256	5	1	2	ReLU
	LRN.	27@256	27@256	-	-	-	-
	Pool.	27@256	13@256	3	2	0	-
	Conv.	13@256	13@384	3	1	1	ReLU
	Conv.	13@384	13@384	3	1	1	ReLU
	Conv.	13@384	13@256	3	1	1	ReLU
	FC.	1@9216	1@4096	-	-	-	-
	FC.	1@4096	1@4096	-	-	-	-
	FC.	1@4096	1@1000	-	-	-	-
VGG16	Conv.	224@3	224@64	3	1	1	ReLU
	Conv.	224@64	224@64	2	2	1	ReLU
	Pool.	224@64	112@64	2	2	0	ReLU
	Conv.	112@64	112@128	3	1	1	ReLU
	Conv.	112@128	112@128	3	1	1	ReLU
	Pool.	112@128	56@128	2	2	0	
	Conv.	56@128	56@256	3	1	1	ReLU
	Conv.	56@256	56@256	3	1	1	ReLU
	Conv.	56@256	56@256	3	1	1	ReLU
	Pool.	56@256	28@256	2	2	0	ReLU
	Conv.	28@256	28@512	3	1	1	ReLU
	Conv.	28@512	28@512	3	1	1	ReLU
	Conv.	28@256	28@512	3	1	1	ReLU
	Pool.	28@512	14@512	2	2	0	-
	Conv.	14@512	14@512	3	1	1	ReLU
	Conv.	14@512	14@512	3	1	1	ReLU
	Conv.	14@512	14@512	3	1	1	ReLU
	Pool.	14@512	7@512	2	2	-	ReLU
	FC.	1@25088	1@4096	-	-	-	-
	FC.	1@4096	1@4096	-	-	-	-
	FC.	1@4096	1@1000	-	-	-	-

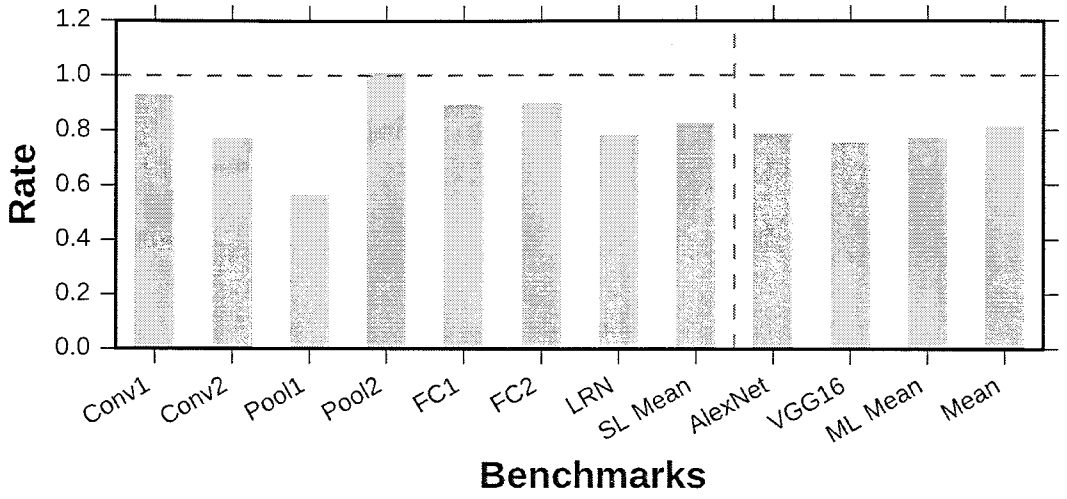


图 3.8: DLPlib 和手写指令的性能比

3.7.2 实验结果

3.7.2.1 性能比较

我们将比较 DLPlib+Caffe，以及手写指令在表 3.5中的 9 种测试例上的运行效率。我们的评估结果如图 3.8所示。图中的所有数据都被归一化到了 DLPlib 的性能数据。可以看到，平均来讲，在 9 个 benchmark 上，DLPlib 能够达到手写指令性能的 79%。

对于单层网络，DLPlib 在卷积，全连接，池化，和 LRN 这几种算法上，分别达到了手写指令性能 77%-93%, 89%-90%, 56%-81%, 以及 78% 的效率。而在全网络上，DLPlib 在 AlexNet 和 VGG16 两个网络能够达到手写指令性能的 78% 和 75%。虽然 DLPlib 比手动优化的指令相比性能有些损失，但是和 Caffe-GPU 相比，还是可以达到平均 4.1× 的加速比，可以起到加速作用。而这些性能上的损失换来的是编程方面的易用性，我们认为这样的损失是可以接受的。下面我们会分别分析单层网络和多层网络中性能损失的原因，这些缺陷也是在未来改进 DLPlib 的一个方向。

- **单层实验结果分析**

深度学习加速器的性能主要受到两方面影响：一方面是数据重用策略，越多数据重用，访存量就会越小，另一方面则是计算和访存之间的并行度，并行度越高，虽然访存和计算量都是一样的，但是设备资源的利用率会越高，总体效率就会越高。对于大规模的操作来说，比如表 3.5中的 Conv2 测试例，由于输入输出的数据规模超过片上缓存容量，需要将这一个卷积操作分割成几个子操作进行计算。DLPlib 中，因为我们需要动态的处理不同规模的卷积层，因此对于某一种特定规

模的卷积操作的优化不够充分。而手写指令则是针对测试例中的特定规模进行了专门的访存计算并行优化，因此在性能上有优势。这样的性能差异，可以通过在库中添加针对不同规模的模板，保证各种规模下的并行执行，进一步优化性能。

- **全网络实验结果分析**

我们观察到，DLPlib 在全网络上的性能比要比单层略差一些。这是由于在手写指令的时候，对于层之间可以进行的一些优化，在 DLPlib 中没有实现。比如层融合的优化，层之间的并行化等等。层融合指的是，对于相邻的两层，前一层的输出可以直接被后一层使用，不需要将这块数据存回片外，读取回来再进行计算，以此节约一次存储（store），一次读取（load）的开销。层之间的并行化则指的是前一层的最后一次计算，可以和下一层的第一个数据读取并行起来，以增加整体的并行性。这样的优化在层数较多的时候可能取得较好的效果。由于 DLPlib 中的每个层都是独立的调用和指令生成，层之间无法融合，导致了调用的开销。

3.7.2.2 易用性

DLPlib 的一个好处在于它极大的降低了程序员在深度学习加速器上开发的成本，这个成本包括了学习成本和开发新算法的成本。Caffe 作为一个被广泛采用的流行的架构，已经被很多程序员掌握，基于其上进行开发可以极大的减少学习成本。比如，我们要利用 Caffe+DLPlib 这套框架实现一个 AlexNet 的网络，只需要从 Caffe 的开源社区上下载一份通用的配置文件（prototxt），然后做少量的修改，比如后端的修改，即可以直接在 DLPlib 上运行。也就是说，我们可以完全复用前端的网络结构构件的这一套工具，而不需要自己重新定义网络结构，这对于大规模的网络，比如 ResNet 这种层比较多，连接关系也复杂的网络结构来说，可以节省巨大的工作量。

3.8 本章小结

本节中，我们提出一种面向深度学习加速器的高性能库，将深度学习中的常用数据结构 Tensor 和 Filter 进行了封装，直接支持一系列深度学习算法的算子，并提出了相应的编程模型，利用该编程模型，我们将高性能库集成到通用卷积神经网络框架 Caffe 中，大幅度降低了深度学习算法的开发难度。

第 4 章　深度学习处理器汇编语言和汇编器

前文中，我们为深度学习处理器设计了一种高性能库，能够在不改变应用程序的情况下集成将深度学习处理器。但是，高性能库的抽象层次仍然较高，也隐藏了诸多的硬件设计细节。这样的抽象使得开发者不需要去考虑这些硬件相关的特性，但这也同时限制了有经验的开发者进一步挖掘加速器的潜力，提升加速器的执行效率。

因此，本节中，我们研究另一种在深度学习加速器上编程的方法，希望可以弥补库在性能优化方面的缺陷。我们希望提出的编程方法可以支持（1）提供高级算子保证编程效率，让开发者们可以快速的搭建一个拓扑网络，以及可以方便的将加速器集成到深度学习框架中去。(2) 提供足够的对底层特性的支持，因此不会限制开发者对硬件各种资源和特性的利用，和进一步的优化程序效率。

我们首先分析了在深度学习加速器上设计这样的语言和生成高效指令序列的挑战和难点，之后通过权衡灵活性和易用性，提出一个领域专用的高层次汇编语言 (High-level assembly language, HLAL)，以及相应的汇编器和运行时系统，并且提出一系列优化运行效率的方案。最后，我们在 Cambricon 指令集 (Cambricon-ISA) 及相应的原型架构 (Cambricon-ACC) 上对 HLAL 进行了评估。值得注意的是，我们提出的汇编语言不仅是一个针对特定硬件结构的编程接口，同时也是一套用于指导深度学习加速器编程接口设计的方法论。

4.1　挑战和目标

在介绍汇编语言之前，我们首先分析在深度学习处理器上编程的挑战。一方面挑战来自于深度学习算法的多样化带来的。深度学习算法具有复杂性，访存计算密集，以及多样性。为了支持更多算子，底层编程接口就需要提供足够的灵活性。其次，深度学习加速器架构中的一些特性，也带来编程困难。比如，深度学习加速器通常采用 scratchpad memory，而不是 cache 作为片上存储，因此需要开发者显式的指定数据在主存储器 (main memory) 和片上缓存 (on-chip buffer) 之间的搬运。这样做的好处是，如果精心优化，合理的调整指令顺序，同步位置，则可以极大的提升计算和访存之间的并行度，但劣势也很明显，就是会增加开发的困难度，如果同步指令的插入不合理，就会导致计算和访存无法并行，性能下降。我们进一步从软件和硬件两方面进一步分析这些挑战。

4.1.1　深度学习算法和硬件的特点

- 算法特点深度学习算法特点让其在加速器上的开发变得非常困难：

1. 大多数算子具有访存密集以及计算密集的特点，因此其对性能的需求很高；
2. 算法的发展非常迅速，新的算子层出不穷。这就要求有一套易用的编程工具，可以让用户快速编写出新算子。
3. 深度学习算法的规模越来越大，因此需要对数据进行划分。

- 硬件特点深度学习加速器在设计上为了能够提升算法的效率和降低功耗，采用了很多的特殊设计，这些特性要求其上的编程接口也要提供相应的支持。具体来说如下所述：

1. 对特殊数据类型的支持，包括对量化，非精确计算，低精度计算，权值和神经元的稀疏等。对于这一类深度学习加速器，上层的数据表达中，就需要有关于相应特殊数据表达的参数设置，以及特殊的数据摆放，量化 / 稀疏的算子的支持等。
2. 深度学习加速器的另一个特点，就是偏好使用 scratchpad memory 而不是 cache 作为片上缓存。这样的做法可以提升对算法的访存的灵活性，提升性能，但是也会导致编程上的困难，以及额外的编程接口的支持，比如显式指定的对片上缓存的读写。
3. 将神经元和突触数据放在不同的片上存储中。这样做的好处是可以提高读取带宽，缓解数据的供应不足问题。但是这导致在上层编程接口中，我们无法用同一个数据结构去表达神经元和突触这两种数据。

4.1.2 挑战

由于深度学习加速器的片上空间有限，而深度学习算法的规模通常都会远远超出片上的空间大小，因此，我们不得不对算法进行计算和数据上的拆分。将一次计算，划分成多次计算，分别送入到加速器中，使得每一次计算的数据量可以放在片上缓存中。然而，这样的数据划分对于程序员来说可能是复杂而且容易出错的。

- **片上地址空间管理**。片上缓存有时候会被划分成多个区域，存放不同类型的数据（比如输入神经元和输出神经元），这也会增加程序员管理片上缓存的难度，尤其是当程序员只能够用低级的指令进行编程的时候。开发者必须要了解片上存储的数据地址，然后再进行数据的分配，这种分配包括去指定具体的地址，非常容易出错，而且很难调试。

- **指令调度**。为了提升性能，我们希望指令可以尽可能的并行执行，而要开发者用一般的汇编指令去编写并行程序，是一件非常困难的事情，而且包含了

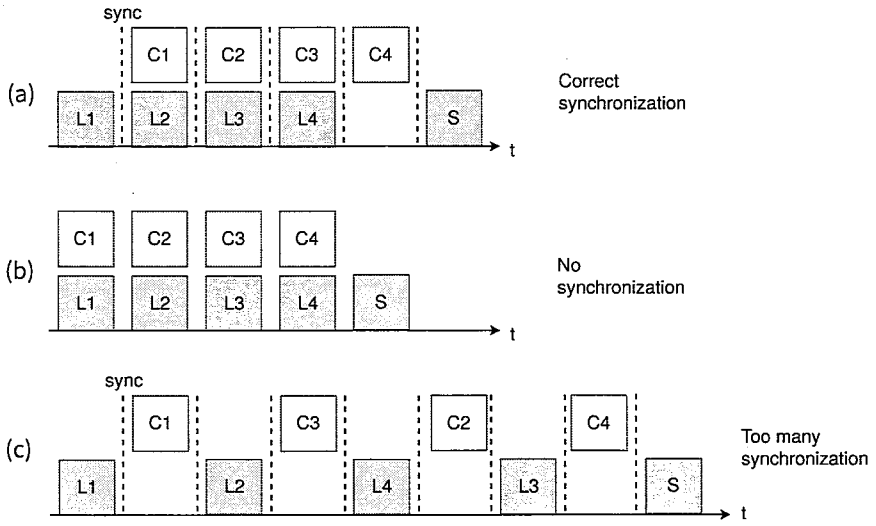


图 4.1: 插入同步指令的位置对执行过程的影响。

并行的程序也很难调试。比如，插入同步指令的位置不正确，就会导致计算错误，而过多的插入计算指令，则可能导致性能严重下降。图 4.1中描述了不同的同步位置对执行过程的影响。其中 C1 ~ C4 表示的四个相互独立的计算操作，L1 ~ L4 为这四次计算操作提供数据，S 表示将计算得出的数据存储到片外。并行性就存在于 C_i 以及 L_{i+1} 之间。

可以看出，同步指令的数量和位置都极大的影响着运行效果。图 4.1(c) 过多的插入了同步指令（在每一个操作之后都插入同步指令），这样做法可以得到正确的结果，但是会极大的影响运行效率；图 4.1（b）中，程序员过少的插入同步指令（不插入同步指令）虽然性能提升了，但是计算结果错误；图 4.1（a）中的执行是正确的，程序员正确的插入了同步指令，结果正确，同时效率较高。

4.1.3 设计目标

- **保证性能**。我们把用户分为两类，一类是对硬件结构并不熟悉的深度学习算法的开发者，这类开发者需要可以快速开发算法，同时获得相对快的运行速度。另一类是对深度学习加速器非常了解的专家，以及一些对性能要求非常高的用户。对于这类用户，运行的性能比开发效率更加重要，他们需要对加速器的性能做深度的挖掘。我们希望提出的编程接口可以满足这两类用户，即提供允许快速开发的高级算子，又提供接近底层硬件的指令集。

- **开发效率**。开发效率指的是在加速器上开发算法的效率，具体包括编程接口的友好性和易用性，调试难度等。开发效率对于第一类用户更加重要，因为他们更加重视开发速度，可以在较短的时间内尝试更多类的算法。

表 4.1: Cambricon 指令集[9]

指令类型		例子	操作数
控制		跳转（J），条件分支（CB）	寄存器（标量值），立即数
数据传输	矩阵	矩阵加载/存储/移动	寄存器（矩阵地址/大小），立即数
	向量	向量加载/存储/移动	寄存器（向量地址/大小），立即数
	标量	标量加载/存储/移动	寄存器（标量值），立即数
计算	矩阵	矩阵乘向量，向量乘矩阵， 矩阵乘标量，外积，矩阵加/减矩阵	寄存器（矩阵/向量/标量） 地址/大小，标量值
	向量	向量对位计算（加/减/乘/除） 向量超越函数（指数，对数） 内积，随机数生成，向量最大最小值	寄存器（向量地址/大小，标量值）
	标量	标量计算（加/减/乘/除），超越函数	寄存器（标量值，立即数）
逻辑操作	向量	向量比较（大于，小于，等于）， 向量逻辑操作（与，或，取反）	寄存器（标量值，立即数）
	标量	标量比较，逻辑操作	寄存器（标量），立即数

4.2 指令集和硬件结构

本节中，我们采用的底层硬件是 Cambricon[9] 指令集和加速器。我们采用 Cambricon 的原因有两点，首先，Cambricon 是第一个提出的针对神经网络加速器的，完整的指令集，同时包含一个相应的加速器架构设计，用这套指令集作为后端具有一定的可扩展性。Cambricon 的设计是基于神经网络算法中超过 90% 的计算都可以分解成向量和矩阵操作的事实，它没有直接提供类似卷积这样的高级指令，而是提供了一系列的矩阵向量指令，以及标量指令，用来组成这些神经网络算法，以此灵活的支持多种算法。

4.2.1 指令集

Cambricon[9] 是一个基于存-取的指令集架构，包括 43 条指令。43 条指令可以分为 4 类：计算指令，逻辑指令，控制指令和数据传输指令。指令例子如表 4.1所示。控制指令提供的跳转和条件分支指令和标量指令，利用标量和跳转指令，理论上 Cambricon 有能力完成所有可能的操作。

4.2.2 架构

Cambricon-ACC[9] 是根据 Cambricon 指令集提出的加速器原型，可以充分的支持指令集。加速器的结构如图 4.2所示。具体来说，Cambricon-ACC 使用 scratchpad memory 而不是寄存器文件，令神经网络算法中的数据存取更加灵活，来支持其中经常需要用到的变长数据访存。向量单元包括 32 个 16 位的加法器和 32 个

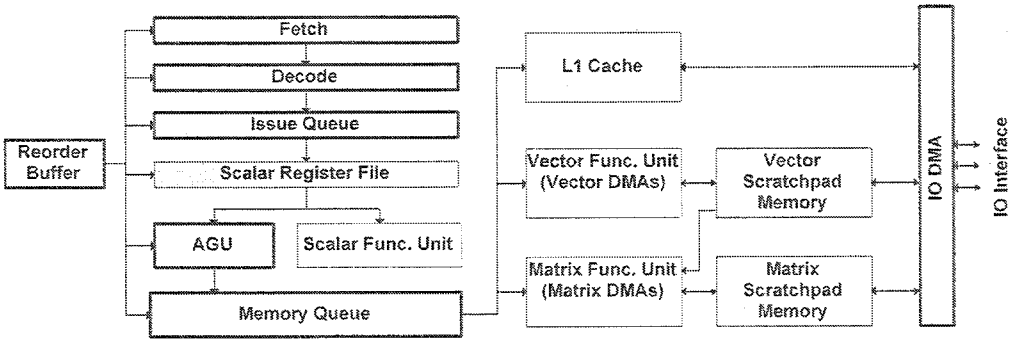


图 4.2: Cambricon-ACC 架构图[9]

16 位的乘法器，向量 scratchpad memory 大小为 64KB。矩阵单元包含 1024 个加法器和 1024 个乘法器，这些加法器和乘法器被划分成 32 个独立的计算单元，每一个计算单元包含 24KB 大小的 scratchpad memory。为了解决并行读写的问题，scratchpad memory 会被分解成 4 个 banks。

4.3 汇编语言

本节中，我们提出一种面向深度学习加速器的一种高级汇编语言（High-level Assembly language, HLAL），其中包括了深度学习专用的数据结构，以及层次化的语句，用于提供编程抽象。我们首先介绍 HLAL 的基本抽象层次和结构，之后分别介绍不同层次的语句，以及数据结构。

4.3.1 概述

传统的汇编语言虽然在性能优化方面有很大的优势，但是也存在一系列问题，比如编程困难，缺乏可移植性等。我们希望可以利用汇编语言和硬件的相似性保证高效性，同时增加不同层次的汇编语言，增加语言的编程友好性。我们希望达到 4.1.3小节中提出的目标，因此，除了一般的底层语句之外，我们还在 HLAL 中提供一些列预定义的高级算子（Block）和高级数据结构（Filer，Tensor），用来封装一些常用的深度学习算法，这样可以帮助用户更有效率的写出高效的代码，同时也增加代码的可重用性和易用性。

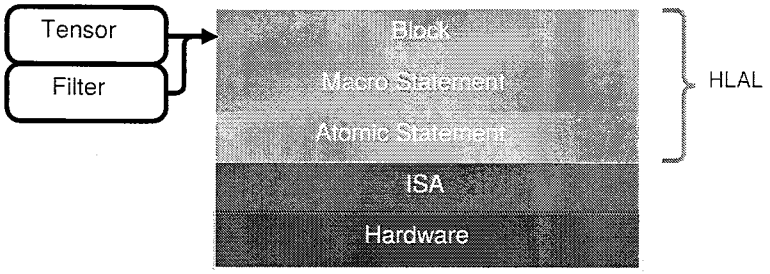


图 4.3: HLAL 抽象层次

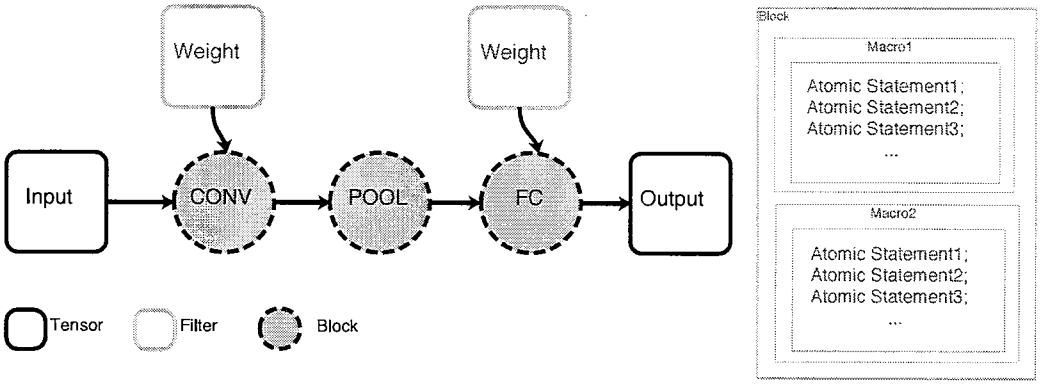


图 4.4: HLAL 抽象层次间的关系概述

Listing 4.1: 样例代码：用 HLAL 实现两层全连接层

```
;; 第一层FC规模：1024个输入神经元，256个输出神经元
;; 第二层FC规模：256个输入神经元，256个输出神经元
.code
_main:
  smove $R1, #1
  smove $R2, #1
  smove $R3, #256
  @block_fw_fc fc1_out, fc1_input, fc1_weight, fc1_bias
  @block_fw_fc fc2_out, fc1_out, fc2_weight, fc2_bias;
.static_rw
  @tensor fc1_out, 1, 1, 256, 1, 1, 256;
  @tensor fc2_out, 1, 1, 256, 1, 1, 256;
.static_ro
  @tensor fc1_input, 1, 1, 1024, 1, 1, 4;
  @tensor fc2_bias, 1, 1, 256, 1, 1, 256;
@filter fc1_weight, 256, 1, 1, 1024, 256, 1, 1, 256;
  @filter fc2_weight, 256, 1, 1, 256, 256, 1, 1, 256;
```

图 4.3中描述了 HLAL 的几个抽象层次，从上到下依次为：Block，宏语句（Macro statement），原子语句（Atomic Statement）。原子语句下面则直接对应后段的 ISA 和硬件设备。我们还提供两种高级数据类型，Tensor 和 Filrter，用于表示神经网路算法中的多维矩阵结构。Tensor 和 Filter 是 Block 的输入。Block 由宏语句和基本语句构成。

图 4.4中是一个更加详细的 HLAL 的不同层次的示意图。图中输入神经元和输出神经元由 Tensor 数据类型表示，卷积层和全连接层的权值（Weight）由 Filter 数据类型表示。Tensor 和 Filter 作为算子卷积，池化和全连接的输入，这类神经网络高级算子在 HLAL 中用 Block 进行定义。一个 Block 由宏指令和原子指令构成。

对于用户来说，要构建一个一般的神经网络，他们只需要调用提供的预定义高级算子 Block，来构造神经网络拓扑结构。此外，开发者也可以利用原子语句，和宏语句定义新的高级算子，以提供足够的灵活性。

4.3.2 源文件结构

代码 4.1中，我们展示了一个例子，实现的是两个全连接层组成的正向神经网络。和传统的汇编语言类似，HLAL 的源程序也是由多个分区（section）构成，在编译和链接加载过程中，不同的 section 会被加载到不同的地址空间中，经过分别处理。HLAL 的源文件由一个代码段和若干数据段构成的：code 代表的是代码段，数据段的标识符为 static_ro, static_rw，分别表示静态只读数据段以及静态读写数据段。

代码段存放的是原子语句，宏语句和 Blocks 构成的源码，静态数据段中的数据在编译时就确定了数据大小，可以预先分配，便于优化，静态数据进一步分为只读和读写，这样的属性标注可以用于调试和检查代码的正确性，比如地址空间分配检查。目前，HLAL 只支持静态的数据分配，包括数据的分段信息，都需要在编译时中确定。

4.3.3 数据类型

HLAL 中，我们提供两种领域专用的高级数据类型，用来表示 4 维数组：Tensor 以及 Filter。通过配置不同的维度信息，这两种结构可以覆盖神经网络算法中所用到的所有向量，矩阵，张量数据。比如，将 4D-Tensor 其中的两个相邻维度都设置为 1，则该 Tensor 就可以当作 2D-Tensor 来使用。这两种特殊的数据类型，可以映射到 Cambricon 中不同的功能单元和片上存储上。

除了高级数据类型之外，我们还为提供更贴近底层的数组类型，这类数组主要用来表示将数据从片外搬运到片内的过程，尤其是表示一段连续的片上地址空间。

matrix scratchpad memory 用标识符 msm 表示，而 vector scratchpad memory 则用标识符 vsm 表示。我们将在本节中具体介绍这几种数据类型。

4.3.3.1 Tensor

Tensor 是 HLAL 中的一种基本结构，Tensor 被设计为 N 维的张量，其中 $1 \leq N \leq 4$。Tensor 可以存放神经元，梯度，以及 bias，和 bias 的梯度。这些数据都会被最终加载到片上的 Neuron buffer 上来使用。Tensor 需要首先被声明，然后被使用。用户需要具体的指定出 tensor 各个域的数值，包括 batch，特征图宽度，特征图长度，特征图个数，以及他们分别对应的分段大小。数据分段指的是，当数据的规模过大，一块数据就需要被划分成好几个小的数据块，依次进行计算。数据的分段我们将在小节 4.3.3.5中做进一步的介绍。Tensor 的声明可以出现在源文件中的 static ro, static rw。其声明格式如下，各参数的含义如表所示。

```
1 @tensor [var_name] [N],[H],[W],[C],
2                    [NS],[HS],[WS],[CS],[Layout];
```

虽然在定义中，我们按照 NHWC 的顺序给数据的维度信息赋值，但实际上，真实的数据维度的排列顺序是由 Layout 参数来确定的。对于维度高于 4 的数据，用户可以首先用 4 维表示最内部的数据维度，剩下的维度信息的可以利用多层循环来进行处理。

表 4.2: Tensor 参数

参数名	含义
Var_name	变量名字
N	batch 的数量
H	特征图高度
W	特征图宽度
C	特征图个数
NS	batch 的数量的分段大小
HS	特征图高度的分段大小
WS	特征图宽度的分段大小
CS	特征图个数的分段大小
Layout	数据摆放维度顺序 (枚举类型：NHWC，NCHW, etc.)

表 4.3: Filter 参数

参数名	含义
Var_name	变量名字
CO	输出卷积核个数
KH	卷积核高度
KW	卷积核宽度
CI	输入卷积核个数
COS	输出卷积核个数的分段大小
KHS	卷积核高度的分段大小
KWS	卷积核宽度的分段大小
CIS	输入卷积核个数的分段大小
Layout	数据摆放维度顺序 (枚举类型：OHWI，IHWO, etc.)

4.3.3.2 Filter

Filter 也是 HLAL 中提供的一种高级数据结构，Filter 被设计为 N 维的张量，其中 $1 \leq N \leq 4$。Filter 可以用来存放神经网络中的权值数据，权值的梯度等等。Filter 和 Tensor 的区别在于两方面：首先，从概念上，Filter 表示的是和突触，也就是神经元的连接方式相关的数据，而 Tensor 表示的则是神经元数据。其次，从到硬件结构的映射角度，Filter 会被映射到和权值 / 矩阵相关的存储单元上，并且被处理权值的指令使用。而 Tensor 会被映射到和神经元 / 向量相关的存储单元上，并且被处理向量 / 神经元的指令使用。Filter 需要被首先声明，然后使用，声明中，我们需要指定各维度大小，以及各唯独相应的分段大小信息。Filter 可以被声明在源文件中的 static ro, static rw。一个 Filter 的定义格式如下所示，各参数的含义如表所示。

```
1 @filter [var_name] [CO],[KH],[KW],[CI],
2                    [COS],[KHS],[KWS],[CIS],[Layout];
```

Filter 主要用于计算卷积和全连接层，一般的卷积层中 KH 和 KW 为卷积核的大小，而对于全连接层来说，KH 和 KW 被设置为 1。

4.3.3.3 标量数据

除了张量数据之外，HLAL 还提供标量数据，主要用于设置对计算的循环，神经网络算法中的一些参数设置，比如学习率，动量，迭代次数等，以及一些标量计算。静态的标量数据，即常数，可以直接写在代码段，即 code 分区中。以如下形式初始化和赋值。标量数据的声明和初始化形式如下所示。

```
@fix16 [var_name] [init_value (optional)]
```

其中 var_name 是变量的名称，init_value 是变量的初始化值。标量数据可以有不同的数值类型，比如 16 位定点（fix16），16 位浮点（fp16）等。这取决于后端硬件支持那些数据类型的计算。对于 Cambricon 来说，只有 fix16 是合法的数据类型。

4.3.3.4 片上数组

Tensor，Filter 和标量数据分配的都是片外的资源，即主存上的空间。它们需要首先被加载到深度学习处理器上之后，再进一步。作为汇编语言，HLAL 希望可以提供给用户对硬件资源的充分的控制能力。所谓的资源当然包括了片上的存储资源，即片上矩阵缓存和片上向量缓存。因此，我们同样定义一种表示片上存储资源的数据类型——片上数组，用来表示深度学习加速器片上的数据块。

片上数组的定义可以出现在一个 block 里，用来定义一块片上数据，tensor 和 filter 数据需要首先拷贝到片上数组中，然后再被具体的宏指令和原子指令调用，它主要用来在定义 block 的计算过程时使用。HLAL 中，针对 Cambricon 架构，我们提供两种片上数组：Vector On-chip Array (VOA) 以及 Matrix On-chip Array (MOA)，分别表示一块分配于 vector scratchpad memory 上的数据和一块分配于 matrix scratchpad memory 上的数据。片上数组的定义格式如下所示，其中 SIZE 是常数。

```
VOA fix16 var_name[SIZE];
MOA fix16 var_name[SIZE];
```

4.3.3.5 数据划分

数据划分是神经网络加速器编程中的重要概念。传统的计算设备通常包含了多级存储结构，比如：硬盘，主存储器，Cache，寄存器等。这些不同等级的存储介质从低到高，现代的编译器已经对这些资源的分配进行了优化。比如 Cache 的替换策略，寄存器分配算法等等。然而，对于神经网络加速器来说，却没有这样的软件基础设施去做相应的片上资源分配。同时，由于神经网络算法具有数据密集的特点，神经网络加速器的片上资源实际上是非常紧张的，如果不能充分利用片上资源，减少数据搬运次数，利用计算和数据存取之间的并行，将很难达到满意的运行效率。

多级存储器中，较大存储器上的数据通常被划分成多个段，被分别加载到上一级存储器上，这个分段的大小是被硬件所限制，这个替换过程也基本上是自动完成（硬件决定）的。而由于神经网络加速器偏好采用 scratchpad memory 而不是

cache 作为片上存储，因此需要程序员去手动将数据加载到片上，这虽然可以提高运行效率，却极大的增加了编程的难度。

传统的向量处理器中，由于处理器内部的寄存器文件大小有限，也不可避免的需要对数据进行划分，其采用的技术被称为条带挖掘 (strip mining) [10]。如图 4.5所示，条带挖掘会将一个一维的向量划分成长度为寄存器文件长度的段，分别加载到处理器中，从而可以处理较大规模的向量。

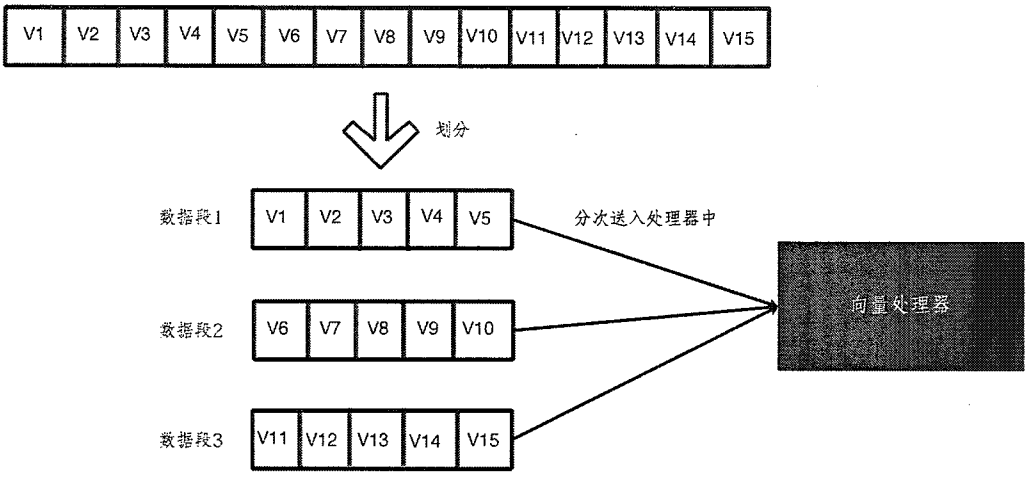


图 4.5: 向量处理器中的条带挖掘[10]

向量处理器中，这样的划分是相对容易的，因为向量数据只有一个维度。然而，在神经网络加速器中，数据的划分就比较复杂了，因为神经网络算法中使用的通常是 4 维的张量数据，而不是向量数据。因此有多个维度都需要进行划分。同时，神经网络算法，相对向量操作而言也更加复杂。比如，卷积操作需要至少 6 个循环完成计算，而每一个循环实际上都可以做划分，以增加数据重用或者满足片上存储大小。

作为一个底层汇编语言，HLAL 并不提供自动的循环 tiling 和数据划分功能，用户需要自己手动的指定数据的分段大小，或者由软件编译栈的更上层来做这样的划分。在汇编语言这一层次，我们允许开发者对分段进行设置，但是不会自动的设置分段参数，上文中，Tensor 和 Filter 的声明参数中的分段信息，就是用来处理神经网络算法中的数据划分的。提供分段信息的一个重要原因是，分段可以影响到数据的摆放，从而影响数据存取效率。图 4.6展示了一个宽度 W 和宽度 W 都为 6 的数据块，从 H 和 W 两个维度进行了数据划分，分成了四个 3×3 的数据块（段）。划分之后，数据的摆放顺序发生变化，图中红色的箭头表示数据存储的顺序。这是由于深度学习加速器在计算时要求加载一段连续数据，如果不进行数

据的重新排列，则需要在运行过程中，利用数据读取指令和跳转指令获取所需要的数据段，导致指令数量的增加。

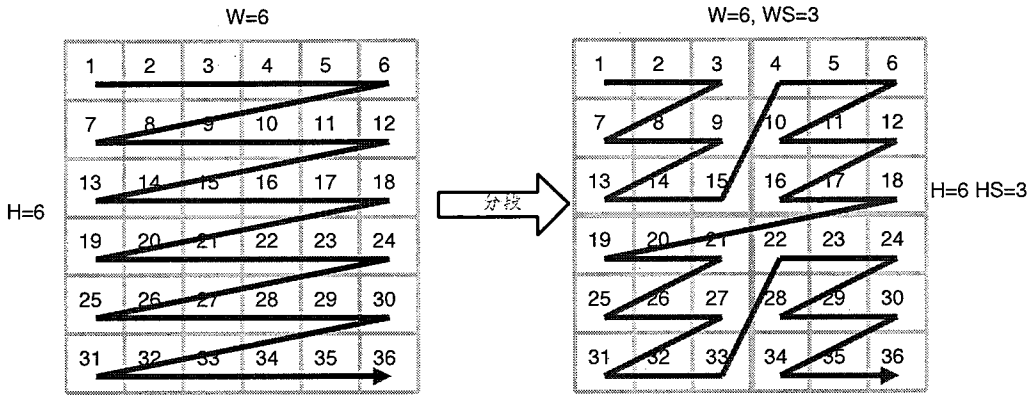


图 4.6: 数据划分

4.3.4 语句

HLAL 提供了层次化的语句，分别表示不同的抽象层次，最下层是最靠近硬件指令集的原子语句，向上依次是，用来表示一个计算或者访存序列的宏语句，以及高级算子 Block。

4.3.4.1 原子语句

HLAL 中，原子语句是抽象层次最低的语句，是对 ISA 的直接封装。原子语句相当于一般汇编语言中的语句，由于原子语句和硬件指令集直接对应，相当于可以直接操作硬件的资源，这就给程序员提供了足够的硬件信息和灵活性。虽然，用户并不会经常的用原子语句写程序，但是其存在保证了汇编语言的完备。在为 Cambricon 所设计的 HLAL 中，依照 Cambricon 的指令集，原子语句被分为 3 类：计算语句，访存 (IO) 语句，和控制语句。其中计算语句可以进一步分为数值计算语句和逻辑计算语句。这样对原子语句进行区分，目的是为了更方便的利用指令的并行性。

4.3.4.2 宏语句

HLAL 中，宏语句被定义为基本的调度单元。传统的汇编语言中，宏指令被广泛运用，主要用于为汇编语言提供一定程度的层次抽象。宏指令在编译时会被替换成一系列的基本指令，其功能和函数类似。在 HLAL 中，宏指令用于抽象一系列原子语句，在编译过程中，宏语句会被替换成定义中的原子语句。

由于深度学习处理器的指令在运行中存在着并行关系，但是编程时指令的编写是顺序的，因此，程序员很难在写程序时设想运行过程。为了解决这个问题，我们设计两种宏语句—计算宏语句（*C-Macro*）和 IO 宏语句（*IO-Macro*）。一个宏语句中包含的指令序列应该属于一类原子语句，即计算原子语句（包括计算指令和逻辑语句），或者 IO 原子语句。被宏指令封装后，代码量减少，程序员就可以更清晰的看到 IO 序列和计算序列的并行性。

一个计算宏语句的声明如以下伪代码所示：

```
@C-Macro macro_name var1, var2,...
    c_atom_inst1;
    c_atom_inst2;
    ...
mend
```

一个 IO 宏语句的声明如下伪代码所示：

```
@IO-Macro macro_name var1, var2,...
    io_atom_inst1;
    io_atom_inst2;
    ...
mend
```

其中，*@C-Macro* 和 *@IO-Macro* 是宏语句的保留关键字。macro_name 是这条宏语句的名字，后面是参数列表。宏语句的参数可以是寄存器名称或者立即数，表示包括片上和片外地址，向量，矩阵的规模尺寸，以及标量数值。宏的声明以 *mend* 关键字结尾。

对宏语句的调用如下伪代码所示：

```
@macro_name param1, param2...
```

汇编语言中，另一种实现代码抽象的方式是使用程序调用（procedure call）。传统汇编语言中，宏和程序调用是并存的，都用来提供代码抽象。程序调用不像宏需要在编译时进行替换，而是在运行时通过一个跳转指令跳转到相应的代码段，执行函数中的指令。HLAL 中，我们采用宏而不提供程序调用，原因如下。首先，宏比程序调用在运行时更加高效。程序调用会带来额外的运行时开销，比如对片上数据的压栈和保存现场等，而宏语句采用替换源码的方式，所有处理都在编译时期完成，不会增加运行时的负担，因此更能满足深度学习加速器对效率的要求。其次，传统的汇编程序更倾向于使用程序调用而不是宏的原因在于，一般的 CPU 函数的指令密度较低，使用了宏替换之后会引起指令爆炸。而深度学习处理器的指令密度较高，不容易引发指令爆炸。因此综合来看，我们认为对于深度学习加速器来说，应该采用宏而不是程序调用来提供编程抽象。

4.3.4.3 同步语句

由于存在计算语句和 IO 语句的并行执行，我们需要通过同步指令在一些存在数据依赖的地方确保语句的执行顺序。HLAL 不支持对数据流和数据依赖的分析，而是要求程序员在编写代码的时候手动的插入同步语句，其格式如下：

```
1 @barrier
```

由于不同的加速器使用的同步语句不同，@barrier 实际上是抽象后的同步指令，在编译过程中，@barrier 会被映射到具体硬件相关的用来阻塞之前指令语句。比如，Cambricon 的指令集并没有提供专用的阻塞指令，我们会在需要同步的地方插入一条和上一条指令相同的指令将其阻塞住。

4.3.4.4 Block

Block 是 HLAL 中的重要结构，用来提供较高抽象层次的算子，一个 block 可以处理任意输入输出规模的一个算子。block 的目标用户是最普遍的深度学习算法开发者，深度学习框架，以及深度学习加速库的开发者们。这些开发者只在算法的层面上进行开发，对性能的关注度不高，因此不需要知道底层的硬件信息，比如存储层次，一个算法的具体实现过程等，而只需要调用最上层的编程接口，即 Block。HLAL 中提供一组内建的 block，包括各种常用的深度学习算法，比如卷积，池化等。同时，开发者也可以自己定义新的 block，其用法和内建的 block 一样。利用这些内建的 block 和自定义的 block，用户就可以很方便的构建出一个神经网络结构。

Block 的定义和调用。 Listing 4.2中展示的是一个全连接层正向运算的 block 的定义代码。首先，我们获得全连接层的分段信息，即将 tensor 和 filter 的参数传输到寄存器里（代码 6-16 行），段数量（seg_n）用来控制循环，段大小（seg_size）用来计算首地址和作为计算语句的传入参数。这个全连接层的正向计算 block 可以处理一个 batch 的，任意输入输出大小的全连接正向运算。为了支持任意规模的计算，输入和输出被分成了多个段，首先计算第一个输出段对应的所有输入计算，将这些部分和加起来，然后计算第二个输出段，直到所有输出段都计算结束。

代码 21-25 行调用了宏语句，实现输入数据的读取（iom_load_input），权值数据的读取（iom_load_weight），以及正向全连接的计算（cm_fc_forward）。由于计算用到的数据是前两个 IO 宏指令读取进来的，因此需要在它们之间插入同步指令（@barrier）来确保运行结果的正确。

预定义的 Block。 HLAL 提供一组预定义的 Block，涵盖常用的深度学习算法（包括正向和反向计算，以及训练算法的指定），以及矩阵/向量操作。在选择包含到预定义集合的算子时，我们希望这些 block 应该足够具有代表性，可以充分涵

盖常神经网络算子，而且尽可能的减少冗余。我们选择的依据同样是通过近年深度学习算法的发展而定，如 3.3.2小节中表 3.3统计数据所示，HLAL 的预定义 block 包括表 3.3中出现的所有算法（如表 4.4所示）。

同时，为了提升对一般用户的友好程度，我们希望提供一组基本算子，新的算法也应该可以通过这些算子实现。根据[9]，神经网络算法中的大部分计算都可以聚合成矩阵/向量操作来完成，因此，HLAL 中包含一组矩阵/向量操作，让用户可以在不接触底层硬件的情况下，自定义新算法。我们所支持的矩阵向量操作如表 4.4所示。

自定义的 block。由于深度学习算法的发展非常迅速，算法开发者们需要实现新算子。我们希望 HLAL 能够满足这一需求，具备足够的灵活性和可扩展性，让开发者们可以利用神经网络加速器进行算法开发。一种构建新算法的方法是利用矩阵向量操作的 block 来实现，这样做的好处是，用户不需要考虑硬件细节，开发起来比较方便。但这样可能导致 block 的调用开销较大，整体效率下降。因此，用户可以利用之前介绍的宏语句，原子语句等自定义新的 block，其用法和一般的 block 相同。

Listing 4.2: 样例代码：正向全连接层 block 定义

```
;; #tensor_buffer: 神经元缓存的初始地址
;; #filter_buffer: 权值缓存的初始地址
def_block @fc_forward_blk(@tensor input,
                          @filter weight,
                          @tensor output) {
  smove $R1, output.seg_size_c
  smove $R2, input.seg_size_c
  ;; R3和R4用来控制循环
  smove $R3, output.seg_n_c
  smove $R4, input.seg_n_c
  ;; R5是输入tensor的片上地址
  smove $R5, #tensor_buffer
  ;; R6是权值的片上地址
  smove $R6, #filter_buffer
  ;; R7是输出tensor的片上地址
  sadd $R7, $R5, input.seg_size_c
OS:
  smove $R4, input.seg_size_c
IS:
  ;; 将输入加载到tensor缓存中
  @iom_load_input input, $R4, $R5
  ;; 将权值加载到filter缓存中
  @iom_load_weight weight, $R1, $R4, $R6
  ;; 计算全连接操作，以及部分和相加
  @cm_fc_forward $R7, $R1, $R5, $R2, $R6
  @barrier
  ssub, $R4, #1;
  CB IS, $R4;
  ;; 将算好的一个输出端存回到主存中
  @m_store_output output, $R3, $R7
  ssub, $R3, #1;
  CB OS, $R3
}
```

表 4.4: 内建 block

类别	内建 block
深度学习	卷积，池化，全连接，LRN，ReLU，Sigmoid，Tanh，反卷积，反池化
矩阵操作	矩阵乘法，矩阵对位加法/减法/乘法/除法
向量操作	向量加法/减法/乘法/除法，内积，外积

4.4 汇编器

本节中，我们为 HLAL 汇编语言设计一个汇编器，用来生成相应的可执行程序。我们首先介绍汇编器的结构，以及编译流程，包括：预处理，地址映射，数据摆放，以及地址分配和重定位。然后，我们会介绍内建 block 的优化策略，以及块之间的调度和融合优化。

4.4.1 编译过程

图 4.7中，我们展示了 HLAL 汇编器的架构。HLAL 汇编器的输入是一个源文件，输出是可执行代码（指令序列）。可执行代码中包含了在设备上以及主机端运行的代码。编译过程主要主要分为 4 个步骤：预处理，静态数据分配，以及重定位。如 4.3.2小节所述的，汇编的源文件通过标识符划分为几个部分（比如：.code, .static_data），汇编器会将这几个段分别送入不同的模块进行处理。代码段会被送入到预处理的模块中，数据段会被送入数据管理模块进行地址的分配以及数据摆放。

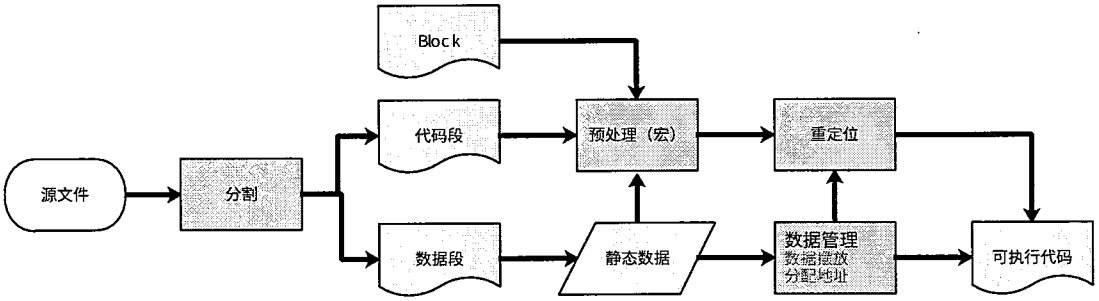


图 4.7: 汇编流程

HLAL 的设计参考了传统的汇编语言，因此其编译流程和传统汇编类似，但作为一个领域专用的汇编语言，同时也由于其目标设备（深度学习处理器）架构的特殊性，HLAL 包含了一些传统汇编没有的特性，比如分段的 Tensor，Filter 数据结构。为了支持这些特性，编译过程也有所不同。我们将在下文中具体介绍汇编器对这些特殊属性的支持。

4.4.2 数据管理

在编译过程中，所有的数据首先在 host 端，即 CPU 上进行分配，然后再拷贝到设备的内存（DDR）上，用于后面的计算。HLAL 中，数据是以 Tensor 和 Filter 类型进行声明和管理的。和一般数据不同的地方在于，Tensor 和 Filter 是多维数组，需要根据分段的参数信息以及 layout 参数，对数据进行重新排列，以适应后面的计算过程。一个 Tensor 类型的分配过程如图 4.8所示。

Tensor 和 Filter 首先根据声明中的分段信息计算出全部的分段参数。Tensor 和 Filter 的声明中，我们确定了某维度的长度 $size$，以及分段的长度 seg_size，我们需要通过这两个值计算出分段的数量 seg_n，以及不能整除的段的部分 seg_r。数据信息会用于地址空间的分配，包括主机端和设备端。之后主机端的数据可能会进行初始化，之后，根据数据的分段信息，地址信息，数据会被拷贝到设备端的地址空间上。

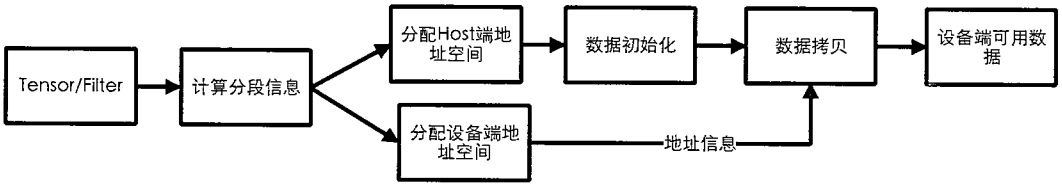


图 4.8: Tensor/Filter 数据分配流程

4.4.2.1 地址分配

HLAL 语言编译中的地址分配包含两部分，在主机端的地址分配，以及设备端的地址分配。神经网络的预测中所使用的样本数据，以及权值数据都需要从主机端读取过来，因此这一部分的数据需要先在主机上分配，然后再从主机端传输到设备端上。主机端上的地址分配由操作系统完成，设备端的地址由相应的驱动完成。由于在神经网络拓扑中，多数的数据都是静态数据，对地址的分配只需要按照.static_data 中声明的数据进行顺序的分配。

4.4.2.2 数据摆放

数据摆放是 HLAL 编译中的重要问题，尤其是在神经网络预测中，输入数据和权值数据无法被完全放到片上存储中，因此需要被划分成多个段，由于维度的顺序问题，划分后的数据会变成不连续的。要将这些不连续的数据加载到片上，一种最直接的方法，就是用多条指令，从不同地址搬运数据。这样的好处是，数据不

需要被重新排列，但缺点是指令数量的增加，会导致延迟和指令解码的开销增加。因此，为了提升性能，我们需要将数据进行重新排列。

图 4.9中是一个数据重排的例子。图中，数据块 1，数据块 3（蓝色部分）是一次计算所需要的数据，在重新排列之前，我们需要用两条指令 load1，load2 分两次加载这些数据，进行数据重排之后，数据块 1，3 被放置在了相邻的地址空间内，因此可以用一条 load 指令将这段数据加载到片上。

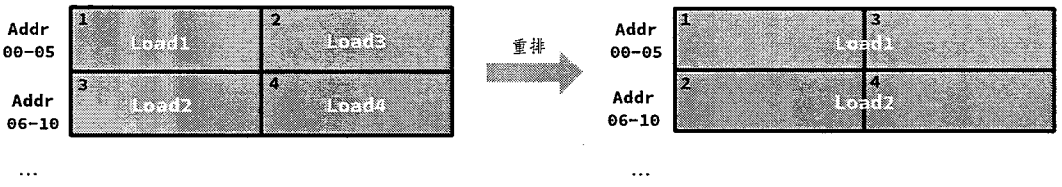


图 4.9: 数据划分可以减少需要的 Load 指令条数

4.4.3 并行模型

虽然 Cambricon 的设计主要利用数据并行（Data-level parallelism）进行加速，但是，由于神经网络算法的计算和访存密集，而且存在大量可以并行的计算和访存，因此，通过合理的利用指令间并行，也可以有效的提升效率。Cambricon 中，并行编程的困难来源于其深度有限的指令队列（深度为 2）。由于一般的算法都需要多条计算指令来完成，而访存指令的数据量较少，当生成指令时，如果访存指令计算指令的前面，则会将后面本来可以并行执行的访存指令卡住，无法并行执行。比如，用 Cambricon 指令实现 Pooling 计算需要 9 条指令[9]，而加载输入数据只需要一条访存指令（从片外加载到神经元缓存），如果要 Pooling 的计算和访存并行执行，我们需要将访存指令调整到计算指令的前面，让他们可以并行执行。

本节中，我们提出一种编程方法帮助程序员更好的利用 HLAL 开发并行程序，并用全连接层举例说明该模型，具体如图 4.10所示。我们提出的编程方法主要分为两步：

1. *用宏语句封装串行执行的计算片段和访存片段。*为了增加访存和计算之间的并行，我们需要代码能够清晰的区分串行部分和可以并行的部分。因此，我们将利用 HLAL 中提供的宏语句，将代码分成计算宏指令和访存宏指令。首先，我们需要将算法映射到计算操作和访存操作上。如图所示，全连接层的正向计算中，包含了两条访存指令（读取权值和输入数据），以及两条计算指令（矩阵乘法和加部分和）。我们将访存指令封装成一条 IO 宏指令，计算指令封装成计算宏指令。

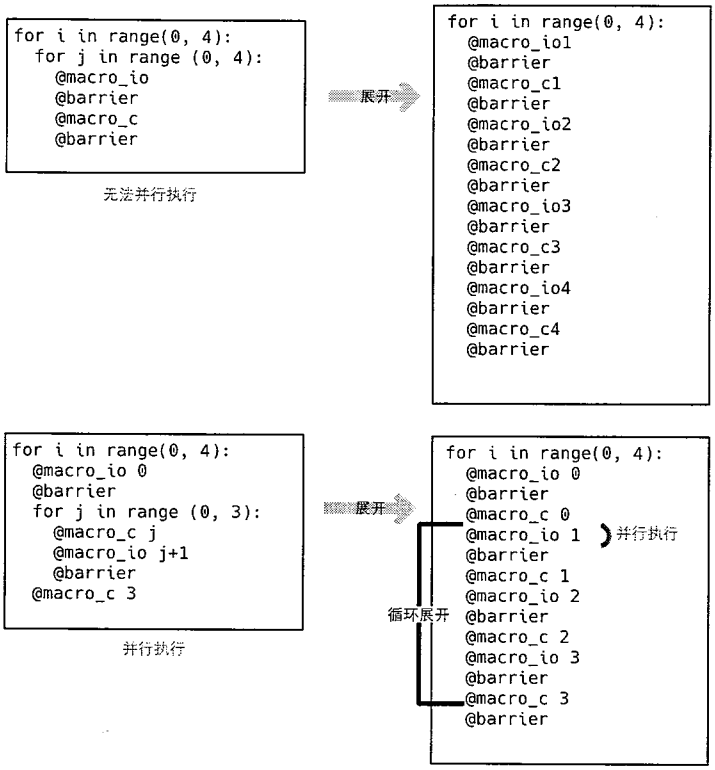


图 4.10: 循环拆解增加并行度

2. 调度宏语句。在封装好宏语句之后，我们需要调度这些宏语句，让其可以尽可能的并行化执行。这中间主要涉及 3 个方面的调度。(1) 拆解循环形成。图 4.10中是一个通过拆分循环增加并行度的例子。上半部分图中是一个朴素的实现方式，循环中直接写入加载和计算，为了保证正确性，我们需要在每条指令后面都插入同步指令 @barrier，这会导致并行度低，计算速度慢。因此，我们调整循环的次数，将第一次的 Load 操作提取到循环外部，让之后的操作可以并行执行，如下半部分图所示。(2) 调整指令顺序。有时候，两个相互独立的宏语句在实际运行中无法并行执行，在 Cambricon 中，这是受到了发射队列的深度的影响。比如如图 4.11所示，全连接层的四条指令，理论上是可以并行执行的，但是由于序列深度的原因，前两条指令会首先被解析，然后串行执行，而后面的指令无法被执行。这种情况下，我们需要将可以并行的四条指令分别装入不同的宏指令块中，然后调整他们的顺序，确保一条访存指令和一条计算指令并行执行。

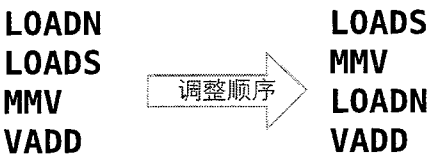


图 4.11: 调整计算指令增加并行度

4.5 运行时

本节中，我们介绍为 HLAL 在主机端和深度学习加速器上运行的运行时行为。对于一般的程序语言来说，已经有非常完整的运行时的支持（操作系统，链接器等），而对于深度学习加速器来说，却还没有这样的基础设施软件。因此，我们为深度学习加速器设计运行时系统，以支持其和主机端的互动。作为一个计算设备，深度学习处理器的运行时操作主要是由主机端负责，比如，分配内存，内存拷贝，数据摆放等行为都是由主机端完成。

HLAL 的运行时过程如下。首先程序被调用，host 端（CPU）会首先在主存上分配所有的静态数据。然后，CPU 将加速器的程序通过驱动发送到设备的内存（global memory）中。然后，CPU 设置加速器的启动寄存器让加速器开始运行。首先，指令直接进行运行，到结束，HLAL 中不支持动态的数据分配，所有的数据都是在运算开始之前在 host 端进行的分配，开始运算之后会直接将所有指令算完。在神经网络算法中，对动态数据分配的需求很少，即便存在，也可以由 CPU 端来实现，动态数据分配对于加速器来说是非常不利的，基本上相当于两次单独的程序执行。

4.6 实验

本节中，我们将对 HLAL 以及相应的汇编器进行评估。我们的评估主要从两方面进行：（1）语言的易用性；（2）HLAL 生成代码的运行效率。

4.6.1 实验环境和方法

4.6.1.1 Baseline

在对运行效率的评估中，我们采用几种 baseline 进行对比。

1. **手动优化的 Cambricon 代码**。手动优化的 Cambricon 代码表示一种接近最优的性能实现。实现中我们充分的考虑了指令的并行。

2. **手写朴素的 Cambricon 代码**。表示没有充分考虑指令并行的手写代码，可能浪费较多的并行性，因此性能相对较差。

3. **GPU**。我们使用 GPU 作为一个性能对比，表明 Cambricon 代码性能的合理性。我们使用 Caffe 作为前端，后端采用 Nvidia K40M 型号的 GPU（包含 12GB 的 DDR5 存储，4.29TFlops 峰值性能，采用 28nm 的工艺）。Caffe 是一个非常流行的高性能神经网络框架，可以支持多种设备，包括 CPU，GPU。其对 GPU 的支持是通过调用 GPU 提供的高性能库接口，本节的实验中，我们采用 cuDNN 的后端接口。

以上的 Cambricon 代码都是运行在 Cambricon-ACC[9] 原型设备上，其结构如图 4.2所示，其参数如表 4.5所示。

表 4.6: Benchmarks

Name	layer	InS#FM	OutS#FM	K	S	Source
conv1	*Conv*	7#512	7#2048	1	1	ResNet[4]
conv2	*Conv*	14#512	14#512	3	1	VGG[13]
conv3	*Conv*	13#256	13#384	3	1	AlexNet[25]
pool1	*Pool*	28#512	14#512	2	2	VGG
pool2	*Pool*	56#256	28#256	2	2	VGG
fc1	*FC*	1#9216	1#4096	-	-	AlexNet
fc2	*FC*	1#4096	1#4096	-	-	AlexNet
fc3	*FC*	1#4096	1#1024	-	-	AlexNet
lrn	*Lrn*	27#256	27#256	-	-	AlexNet
lstm	*LSTM*	input(3)-hidden(400)-output(121)[103]				

表 4.5: Cambricon-ACC 参数配置

参数	配置
PE 数量	32
向量缓存大小	64KB
矩阵缓存大小	723KB
带宽	128GB/s
访存延迟	100ns

4.6.1.2 Benchmark

本节中，我们选择 10 个层作为 Benchmark 进行性能评估，涵盖不同规模的卷积层，池化层，全连接层，LRN 层以及 LSTM 层，这些层都是从真实的网络拓扑（AlexNet, VGG, ResNet）中提取出来的。表 4.6中列出了我们选择的 Benchmark 的参数。除了单层的 Benchmark 之外，我们还在两个大规模全网络（AlexNet, VGG16）上对 HLAL 进行了测试。

4.6.2 性能评估

4.6.2.1 结果

图 4.12中汇报了我们在 10 个单层的 Benchmark 上运行得到的几个加速比，分别是 GPU 比 HLAL，手动优化指令（Hand-optimized）比 HLAL，手写朴素指令（Hand-written）比 HLAL 之间的性能比。和 GPU 相比，HLAL 在正向上获得了平均 2.90× 的加速比，反向获得 3.57× 的加速比。和手写优化的指令相比，HLAL 正向上可以达到其 95% 的性能，反向上可以达到其 96% 的性能。和手写指令相比，HLAL 在正向上可以达到 1.20× 的加速比，反向上可以达到 1.14× 的加速比。HLAL 整体性能比手动优化的指令差一些，最好的一个 Benchmark 是在正向 FC 上达到其 98% 的性能，接近最优。而其最差的性能比则是在 conv3 的正向操作上，只达到其 74% 的性能。HLAL 性能差的主要原因来源于卷积指令复杂的循环嵌套，以及其运算密集的特性，导致并行性的利用比较困难，具体分析见 4.6.2.2小节。而 FC 层的计算更加规整，而且是访存密集型，计算比较容易被隐藏起来，因此性能损失更少。我们进一步评估 HLAL 生成指令在两个真实的全网络（AlexNet, VGG16）上的性能。我们用 HLAL 和手写指令比较，HLAL 在两个网络的推理阶段分别取得 1.16× 和 1.09× 的性能提升。其提升主要也是来源于对指令间并行性的优化（如 4.4.3小节中提出的方法所述）。全网中的部分性能提升来源于对多层融合 Block 的使用，比如 VGG16 中的 Conv-ReLU-Pool 结构。利用层融合算子可以将性能提升 2%，对于一个融合结构来说，访存量可以减少 10.16%。

4.6.2.2 分析

我们发现，和其他 benchmark 相比，卷积层的加速比是最低的，因此我们进一步针对卷积算法进行评估。在理想情况下，卷积这类计算密集的算法的访存操作应该和计算操作充分的并行，但是，通过对执行过程的分析，我们发现，实际中卷积中读取神经元和权值的指令和计算指令并没有充分并行，从而导致了性能损失。这是由于，卷积需要两条加载指令，而这两条指令的运行时间是不定的，我们的实现中，没有根据运行时间来将计算操作进行划分，从而和这两条加载指令并行执

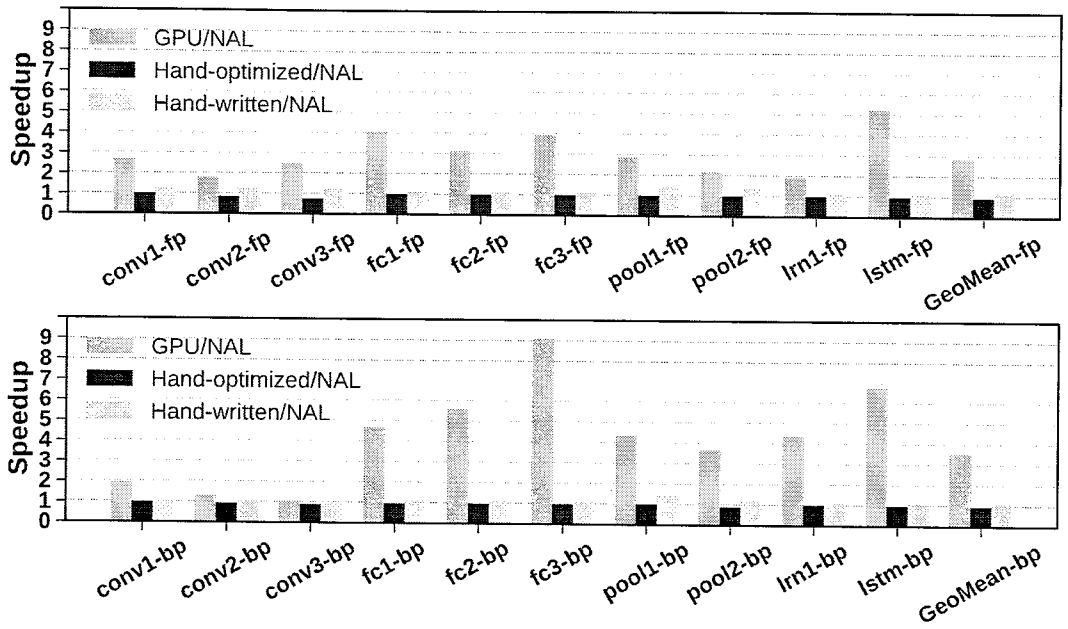


图 4.12: 加速比

行。这种情况在全连接层中并不会出现，因为全连接层是访存密集型的，及时有两条访存指令，但是计算指令的运行时间远小于访存，因此可以和访存并行执行。

4.6.3 开发效率评估

使用 HLAL 进行开发比用指令直接编程极大的提升了开发效率。比如，使用用 HLAL 实现一个两层全连接层的网络只需要 15 行代码，而[9] 中实现一个全连接层的代码则需要超过 50 行代码，单层可以将代码量降低 3.33×，对于实现越复杂的层来说，单层减少的代码就越多。当神经网络的拓扑结构越来越复杂，层数越来越多，HLAL 可以更显著的降低代码长度。在 ResNet34 上[4]，我们可以将代码的行数减少 21.3×，大幅减少了代码量。

4.7 本章小结

本节中，我们提出一种面向深度学习加速器的汇编语言，以及汇编器，提供对硬件功能的更全面的支持，同时可以弥补之前一章中高性能库的编程灵活性。汇编语言中包含一组低级的基本语句，对应硬件的基本功能，，宏语句，用来封装一组计算语句或者一组访存语句，以及高级 block，支持任意规模的深度学习算子，以提升编程效率。汇编语言在以 Cambricon 为后端的情况下，在 10 个 Benchmarks 上，和最优性能相比获得正向 95% 和反向 96% 的效率。

第 5 章　深度学习处理器中间表示

在上节中，我们为深度学习处理器设计了一种高级汇编语言，便于用户利用加速器的硬件特征。汇编语言的灵活性很高，但是也相应的也让编程变得困难，同时，汇编语言的底层性使得它通常会和硬件特性紧密耦合在一起，不容易迁移到别的平台上，尤其是当越来越多的加速器被提出的时候，软件栈的可移植性极大的影响着开发效率。为了解决可移植性问题，我们提出一种面向深度学习处理器的中间表示。设计中间表示的挑战主要来源于两方面：一方面挑战来自上层框架提供的大量算子以及底层硬件提供的算子之间的不匹配；另一方面来自于深度学习加速器的体系结构设计，它们虽然有共性，但也存在差异，此外，加速器提供的可编程接口也不尽相同，这增加了底层硬件抽象的困难。本章中，我们将分析中间层设计的挑战和难点，并根据上述分析提出一种能够支持多种抽象层次编程接口的中间表达，以及相应的数据管理模块和指令生成器。

5.1 挑战

在第 1章中，我们总结了现存加速器使用的编程接口和指令生成方式，可以看出，现有加速器的编程方法非常不统一，在抽象层次以及提供算子的粒度上都有很大区别。这对中间层编程接口的设计带来了挑战，另一方面，深度学习处理器自身特殊的硬件结构，也要求中间表示能够表达这些特殊的结构和操作。

5.1.1 加速器带来的挑战

- **原生编程方法的多样性**。如 1.3.3小节所述，已经提出的深度学习加速器在计算粒度和抽象层次上都有很大的差异。由于我们希望可以尽可能的利用加速器已有的编程接口。因此，我们需要找到一种方式，可以兼容不同抽象层次，不同操作粒度的接口。图 1.3中展示了用不同粒度算子实现卷积的伪代码，我们可以看出，硬件支持的计算粒度不同，卷积的循环拆解方法也不同。ShiDianNao 的结构可以直接运行一个卷积指令，而 Cambricon 指令集则需要将卷积拆解成矩阵乘法之后再进行运算。而其他粒度的处理器，比如 CPU，向量处理器的粒度则更小，需要将卷积拆分成更细粒度的操作之后才能映射到硬件上。

- **硬件中的特殊特征**。深度学习加速器在设计中，为了提升性能，包含了很多特殊特性。首先，深度学习加速器通常支持的是以多维数组为操作数的运算，如卷积，矩阵运算等，和 CPU，GPU 将计算分解成最基本的标量运算不同。

其次，为了提升卷积算法和全连接算法的数据带宽，大多数加速器都会在片上放多块独立的缓存来放不同的操作数，比如，DianNao，Cambricon-X 片上有三块缓存（输入神经元，输出神经元和突触），而 Cambricon 则有两块缓存（向量缓存和矩阵缓存）。然而，深度学习框架通常使用同样的数据结构（多维度 Tensor）表示突触和神经元数据，比如 TensorFlow 中的 Tensor，MXNet 中的 NDArray，以及 Caffe 中的 Blob。这两部分数据，在真实被硬件调用的时候，会被安排到不同的片上空间中，要保障计算的正确性，这样的信息就需要在编程接口中有所体现。此外，很多硬件还为一些和数据类型相关的神经网络算法，比如稀疏神经网络，低精度网络等，提供了特殊的硬件支持，这些特殊属性也给中间层的设计带来了难度。

- **数据管理和缓存复用**。另一个挑战依然来源于加速器的特殊设计，相比于 cache，加速器更偏好于使用 scratchpad memory 作为片上缓存。原因在于，深度学习加速器的数据存取通常是连续而大量的，而且比较有规律，使用 cache 则需要很大的 cacheline，消耗的功耗大，并不划算。采用手动编程的方式管理片上缓存的好处在于，可以减小这部分功耗，同时通过手动管理数据访存，也可以进一步增加访存和计算之间的并行。然而，手动管理缓存的方式显然给编程带来了更大的困难。尤其是，当数据规模增大，片上缓存无法容纳所有数据，而需要进行数据拆分的时候，多块数据需要复用同一块缓存。这时候，如何合理的调度运算顺序，访存顺序就成为一个编程上的挑战。

5.1.2 算子多样化带来的挑战

- **大量的算子**。一个高级的编程框架通常提供大量的算子，比如，TensorFlow 包括了超过 6000 个算子。框架中的算子数量通常远大于深度学习加速器。将全部算子一一映射到加速器提供的原子操作上是不现实的。比如，Cambricon 指令集只能提供矩阵操作，其和框架之间的无法直接映射，而需要通过某种中间转化。

- **不同的计算粒度**。不同框架的算子粒度也是各不相同的。基于层的框架，比如 Caffe，将算子组织为高级的神经网络算子，而基于图的框架中包含更多细粒度算子，比如基本的标量加减乘除。此外，它们实现一种新的层的方式也不同，基于层的框架要求要修改框架的源代码，通过调用库代码，比如后端提供的高性能库（如 cuDNN）来实现新的层，而基于图的框架则允许用户直接调用已经定义的细粒度算子，利用这些算子组合成新的算子。因此，对于基于层的框架，我们需要提供粗粒度的可以直接调用的库级别代码，而对于基于图的框架来说，我们需要提供的细粒度，可以进一步进行组合优化的算子。

- **高抽象层次**。框架的抽象层次通常很高，为了提供可移植性，框架通常只进行平台无关的一些优化，而具体牵扯硬件细节的实现则交给后端提供的库来完成。比如，GPU 提供的 cuDNN 神经网络加速库就提供了一系列常用的深度学习算法，用来进行 GPU 上的加速，被框架广泛的使用。但是，当需要实现一个 cuDNN 不支持的算子时，就需要用户自己去实现了。实现的方式大致分为两种：第一种是通过调用矩阵运算高性能库来实现新的算子，比如在 Caffe 上，用户可以利用 GPU 的 cuBlas 高性能库来添加新算子。另一种方法是，利用已经支持的算子来实现新算子，比如 TensorFlow。然而，这两种方式从本质上都是调用一系列库函数，从优化的角度讲，后者只是增加了对图的优化，而没有提供更具体的硬件相关的优化。对于深度学习加速器来说，高抽象层次要求我们提供一种可以对硬件进行优化，但是又编程友好的接口。

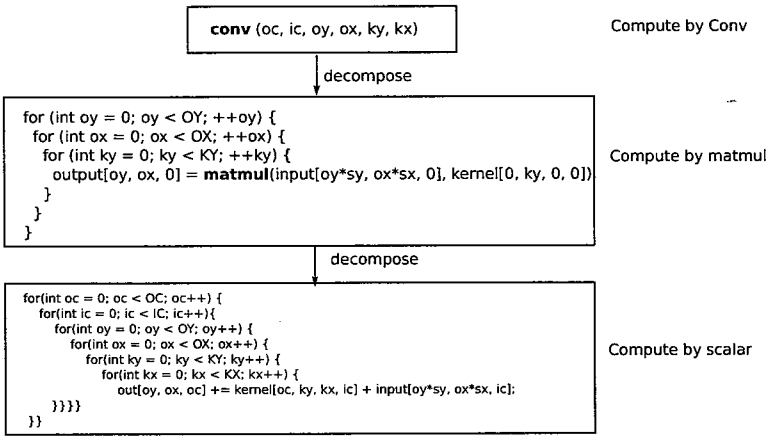


图 5.1: 算法映射：卷积操作可以被拆分成不同粒度的运算

由上分析，我们需要找到一种在加速器上实现算子的编程方法，将算法映射到加速器提供的算子上，同时还需要充分的利用硬件资源，对算法做合理的划分。

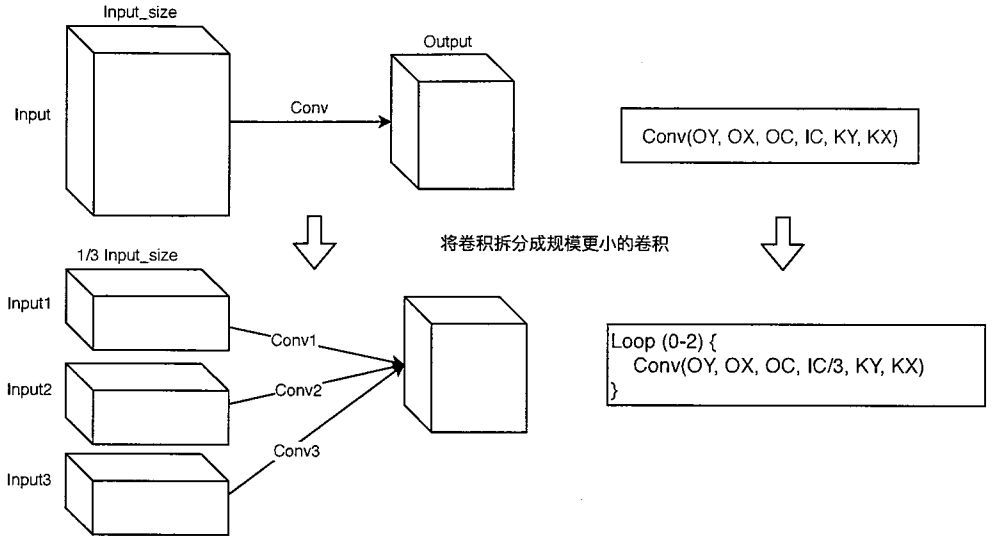


图 5.2: 硬件资源映射：将一个卷积划分成多个卷积操作

5.2 中间层

为了解决上述问题，我们提出一个中间层，位置在深度学习框架和深度学习加速器之间，为深度学习加速器提供一种统一的抽象接口，称为 DLIR (Deep learning intermediate representation)。DLIR 的输入为一个计算图，输出是面向各个平台的可执行代码。中间层中包含了一个基于 tensor 的中间表示（包括高级中间表示和低级中间表示），一系列模板库，以及指令生成器，其可以进行自动的数据划分以及软流水调度。

本节中，我们首先介绍设计思路，然后概括描述中间层的结构，以及其支持的功能，之后对中间表示，各模块的功能进行概述。

5.2.1 设计思路

本节中，我们介绍中间层的设计思路。根据对深度学习加速器指令生成器抽象层次分析，并结合 5.1中提到的关键问题，提出我们的解决思路，介绍将粗粒度的，规模不限的算子，映射到细粒度的，规模受限的算子的策略。

5.2.1.1 基于 Tensor 的计算

深度学习算法多数以 Tensor 作为输入输出数据，因此，过去提出的深度学习框架中都对 Tensor 进行了支持，但是，这些 Tensor 最终都会被拆解成标量计算来描述，这是由于传统的计算设备 CPU 和 GPU 中的计算，还是以标量的计算为单位。而深度学习加速器和传统设备，在编程指令级别的接口上的最大区别在于，

其硬件支持的计算也是以 Tensor 为单位的。因此，在对深度学习加速器进行编程的时候，我们不需要将计算拆解到标量的级别进行描述，而是可以以 Tensor 为单位进行计算。因此，DLIR 支持的主要数据结构即为 Tensor。

一个 Tensor 可以表示一块逻辑上的数据，但是这块数据最终需要被映射到具体的地址空间中。由于硬件的片上空间有限，一个逻辑上的 Tensor 需要被拆分成多个块，不同的块会分时的复用同一块片上存储，这个过程关系到计算调度和优化。我们提出两种层次的 Tensor 结构，HLTensor (High-level Tensor) 和 LLTensor (Low-level Tensor)，分别表示逻辑上任意规模的 Tensor，以及实际可以加载到缓存上的一块 Tensor 数据。HLTensor 可以被拆分成多个 LLTensor，这个拆分过程我们通过设置 HLTensor 的 Dimension 实现。关于 Tensor 的设计我们在 5.2.4中我们将进一步介绍。

5.2.1.2 计算映射和调度

要将一个抽象的，粗粒度的算子映射到具体的硬件指令上，需要经过多级的计算拆分和映射，而由于深度学习加速器的指令集的粒度各不相同，我们很难将这个映射过程统一起来。比如，对于 Cambricon 指令集，卷积算子的实现需要先把卷积映射成矩阵乘法计算，然后再根据 Cambricon 加速器上的资源数量，比如乘法器数量，片上资源大小等，将矩阵乘法拆分成更小的运算，然后再为其生成指令。但是，对于 ShiDianNao 加速器，由于 ShiDianNao 直接支持卷积算子，因此不需要卷积到矩阵乘法的映射，只需要将卷积拆分成硬件可以容纳的规模进行计算即可。这个过程可以进一步分成对两个方面的描述：算法映射以及硬件资源映射。

- **算法映射**。算法层面的映射指的是将算法从语义角度拆分成更简单，粒度更细的操作。比如一个卷积操作，从算法角度上，可以被拆分成多个矩阵乘法操作。一个矩阵乘法操作，又可以被拆分成多个内积操作，最后内积可以被拆分成多个标量操作，如图 5.1所示。对于每一个算子，我们都首先对其进行算法映射，映射过程中，算子的粒度会逐渐变细，直到所有用到的算子都用可以支持的原子算子支持。

- **硬件资源映射**。硬件资源的映射指的是将逻辑上（也就是算法功能上）可以支持的算子，映射成具体的、可以在硬件上执行的指令序列。这个映射过程涉及到对计算的划分，对具体硬件资源的利用（包括计算资源和存储资源），数据重新摆放，对片上数据的管理，以及对数据访存和计算顺序，并行的优化等问题。图 5.2描述了一个卷积算法从逻辑到硬件的映射过程。逻辑上，卷积算法只用一条指令就可以完成，但是实际上，由于片上缓存资源有限，卷积数据无法全部放在片上，而是需要分成三块数据用三条卷积指令计算。

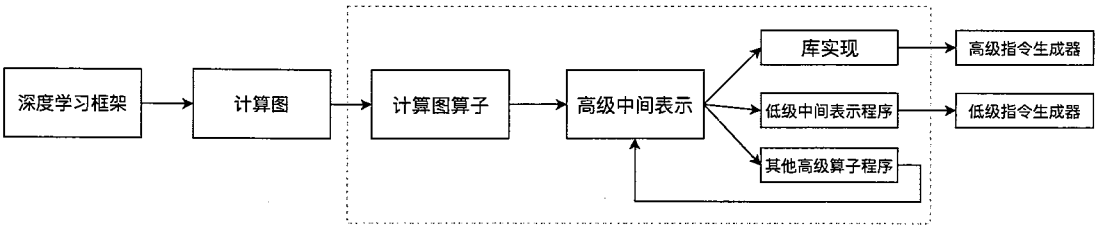


图 5.3: 中间层整体结构

5.2.2 加速器功能抽象

为了支持跨平台，我们首先对深度学习加速器进行抽象，我们的抽象是基于加速器提供的编程接口上的，我们根据其按照接口的抽象层次，或者说和硬件的独立性，分成两大类：高级编程接口以及对应的高级指令生成器；和低级编程接口以及对应的低级指令生成器。高级编程接口屏蔽了和底层硬件相关的信息，其抽象层次和库的接口类似，其可以处理任意规模的算子，第 3章中我们提出的高性能库就是一种高级指令生成器。低级编程接口的抽象层次更低，其算子无法处理任意规模的算子，计算规模受到片上资源的限制，当数据规模较大，就需要程序员来针对算法进行数据划分，比如如何安排片上资源，以及增加指令间的并行等。第 4章中我们提出的汇编语言的基本语句部分就是一种低级的编程接口，而汇编语言中提供的 block 则属于高级编程接口。

5.2.3 整体结构

图 5.3中表示的是中间层的结构。中间层的前端可以是各种不同的深度学习框架，比如，Caffe 或者 TensorFlow。深度学习框架通常可以提供友好的编程接口，让程序员可以快速的构建一个神经网络。深度学习框架虽然都可以被翻译成计算图的模式，但是却使用完全不同的接口，因此，我们提供一个计算图的抽象结构，用来描述计算图，不同框架的计算图都会被转化成中间层中的计算图结构，然后交给后端进行分析。一个计算图表示一个算子，算子的粒度可大可小，算子的输入和输出都是 Tensor 数据。

对计算图中的每一个算子，我们会搜索其是否有注册了的高级中间表示算子，如果有，则命中，进行下一步的转化，如果没有，则表示该算子无法被中间层所支持，则会将其返回给框架，分配给其他的设备。高级中间表示的有三种实现方式：(1) 直接调用底层提供的库；(2) 用其他高级中间表示实现；(3) 映射为低级中间表示。下面我们分别介绍这三种方式。

- **库实现**。高级中间表示的算子可以通过两种方式支持，第一种是通过用户自定义的库，即直接将一个粗粒度的算子映射成加速器的可执行代码。这种方

式让用户可以利用加速器提供的原本的高级指令生成器来生成指令。比如，我们在第 3一章中提出的高性能库，就是一种高级编程接口，当底层接入的设备是 Cambricon-X，我们就可以用高性能库来执行。

- **其他高级算子实现**。另一种实现方式，是利用已经定义的高级算子来实现该算子，类似于 TensorFlow 中算子的实现方式。高级中间算子的输入输出都是多维数组（Tensor）。高级中间表示被转化成多个中间表示构成的一个计算图，然后会被集成到原始的计算图中，新引入的高级算子会被递归的搜索其实现方式，直到全部被映射到库实现或者低级中间表示程序上。

- **低级中间表示实现**。我们在中间层中提供低级中间表示，作为一种实现高级中间表示的方法。低级中间表示中包含了和硬件资源相关的原语和语句，比如对片上缓存的使用等。没有库实现的算子可以用低级中间表示来实现。我们提供一系列优化工具来优化调度。这也是本章中的重点。低级中间表示对应的后端是低级指令生成器和低级编程接口。比如，第 4章中的汇编器就是一种低级指令生成器。

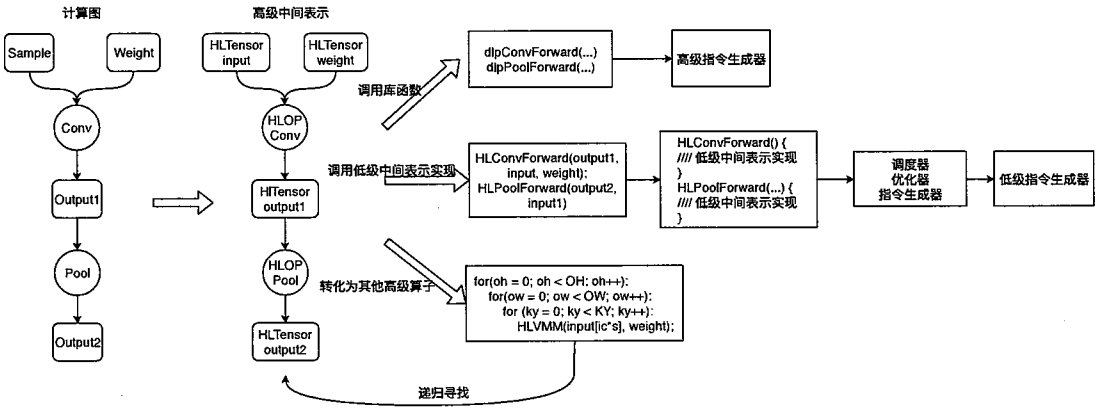


图 5.4: 中间层算子转化过程

5.2.4 中间表示

本节将介绍中间层中的编程接口——中间表示。为了支持不同类型的后端指令生成器，我们引入两个层次的中间表示，高级中间表示和低级中间表示。此处高级和低级指的是和对硬件的抽象层次，高级中间表示和硬件的“距离”更远，是硬件无关的表达，而低级中间表示和硬件关联密切，反映了抽象后的硬件功能和资源信息。相应的，我们还提供两种抽象层次的数据结构，高级张量和低级张量，来表示高级和低级中间表示的操作数。高级和低级中间表示是从抽象层次对功能进行分层，另一个维度，“算法”维度的抽象我们则通过引入多个层次（以数据维度为准分层）的计算语句来实现。

5.2.4.1 数据结构

在 DLIR 中，我们提供为高级中间表示和低级中间表示提供两种对应的数据结构，高级张量（High-level Tensor，HLTensor），以及低级张量（Low-level Tensor，LLTensor），以及一个用来表示维度分段信息的数据结构 Dimension。

•**Dimension**。由于深度学习加速器的片上资源有限，每个 Tensor 要被分成几块分别进行计算，这一技术在向量处理器上被称为条带挖掘（strip mining），而在深度学习加速器上，划分会变得更加困难，主要原因是，深度学习加速器处理的操作的输入输出是多维的 Tensor，而不是一维的向量，Tensor 的每一个维度都可以进行划分，找到合适的划分策略比向量处理器更困难。因为可以划分的维度更多，可以构成的数据块大小也变得更多，这也给数据摆放和定位每一块数据带来挑战。为了更好的表示数据分段，我们将 Tensor 的每个维度用一个特定的数据结构 Dimension 进行包装。每个 Dimension 中包含了某一个维度的分段信息，通过各个维度的 Dimension，我们可以直接计算出 Tensor 被分成的各个数据块的地址和大小。

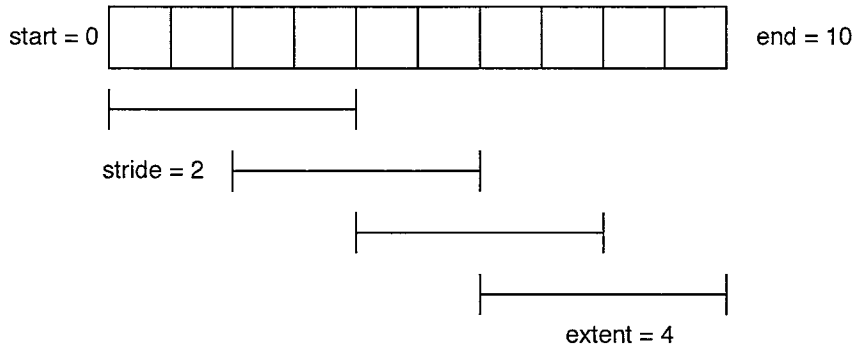


图 5.5: Dimension 结构

Dimension 结构用于表示一个维度上的信息，以及对该维度划分后，访问每一个位置的模式。一个 Dimension 可以用 4 个变量来描述：开始位置（start），结束位置（end），步长（stride），以及每一次访问的长度（extent）。图 5.5中描述了一个 Dimension 描述的维度信息。这个例子中是一个从 0 开始，长度为 10 的维度，在遍历这个维度的时候，我们会从 0 开始，以步长为 2 移动，每次访问长度为 4 的位置。

• **高级张量（High-level Tensor, HLTensor）**。我们提供高级张量作为高级中间表示中算子的操作数。HLTensor 可以表示一个 N 维张量，这个张量的各维度大小可以是任意规模，用来和框架中的多维数组进行对接。HLTensor 中的每个维度都通过一个 Dimension 结构来表示。通过多个 Dimension 的组合我们可以将一个 HLTensor 划分成多个数据块。

图 5.6中是一个二维的 HLTensor，它的两个维度 H 和 W 都被进行了划分。HLTensor 结构除了维度信息之外，还包含了数据的排布（layout）信息，数据的排布顺序被表示为维度的遍历顺序。在深度学习框架中，通常使用预定义的枚举类来指定数据排布，比如，NCHW 表示 batch，channel，height，width 这样的遍历顺序，使用枚举类型指定遍历顺序缺乏可扩展性。HLTensor 中，我们将数据排布和 Dimension 结构相结合在一起，每个维度在定义的时候都被分配一个字符串作为维度的标识名，而数据排布的维度顺序则通过指定标识名顺序来定义。比如，一个 TensorFlow 中按照 NHWC 的维度顺序进行排布的数据。如下所示，我们首先定义四个维度，然后用这四个维度来生命一个高级张量，输入的维度的顺序即其默认顺序的，原始的数据的顺序。之后，我们通过 reorder 函数，可以重新排列这几个维度的顺序。

```
1  // Dim: start, end, stride, extent
2  Dim dim_n("N", 0, 32, 1, 1);
3  Dim dim_c("C", 0, 128, 32, 32);
4  Dim dim_h("H", 0, 10, 1, 3);
5  Dim dim_w("W", 0, 10, 1, 3);
6  HLTensor tensor("T", {dim_n, dim_c, dim_h, dim_w});
7  tensor.reorder({"N", "H", "W", "C"});
```

在进行数据摆放的时候，实际的摆放顺序会按照 reorder 之后的进行排列。

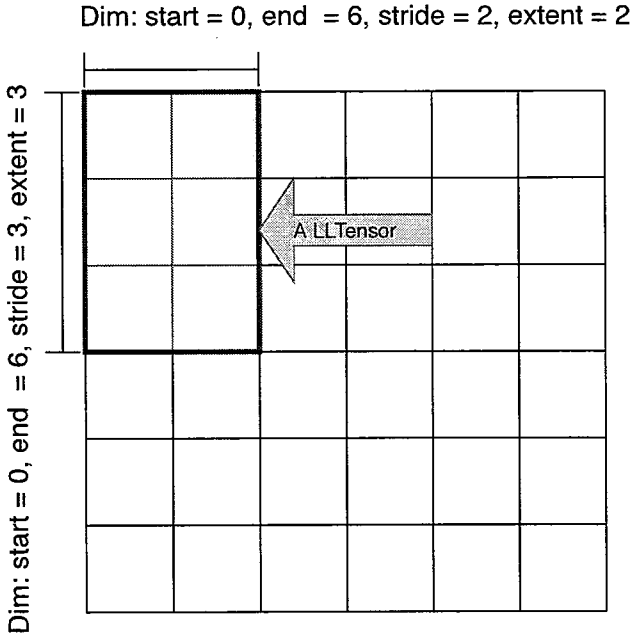


图 5.6: 被分块的 2 维高级张量

• **低级张量（Low-level Tensor，LLTensor）**。当一个 HLTensor 可能被被分成多个块，这每个块被映射成一个 LLTensor，LLTensor 也是一个多维数组，一个

LLTensor 对应一个 HLTensor 中的一块数据，其规模通常是经过划分之后，确保可以在深度学习加速器上放下的。运行过程中，每一条加载指令，会将一个 LLTensor 加载到片上，我们在数据的摆放和调度阶段保障 LLTensor 数据的存放是连续的。LLTensor 实际上是对一块片上缓存的空间的抽象，可以类比为传统计算设备，比如 CPU，中对一个寄存器空间的抽象。CPU 的主要操作是标量计算，标量数据存放在寄存器中。而对于深度学习加速器来说，其主要操作是张量操作，张量被存放在缓存中，而有所差别的是，寄存器的大小是固定的，而一个张量的大小则不固定，因此需要用一个数据结构 LLTensor 对其进行抽象，从而更好的管理。

5.2.4.2 算子

除了数据结构之外，我们提供算子表示计算过程。针对高级中间表示和低级中间表示这两种表示方法，我们的算子也分为高级算子和低级算子。高级算子的输入输出是高级张量，而低级算子的输入输出即为低级张量。

高级算子表示的是一个逻辑层面上的算子，其输入输出是任意规模的高级张量。高级算子可以直接和上层编程框架传过来的计算图中的算子一一对应，并将计算图中算子的输入输出转化为高级张量。高级算子可以通过调用后端加速器提供的高级指令生成器，或者库，直接生成硬件指令，也可以用低级中间表示来实现这个算子，然后利用中间层提供的自动划分模块，调度模块实现生成低级中间表示指令序列，最后调用低级指令生成器来生成具体的二进制代码。

将算子分成高级和低级，只实现了在抽象层次（和硬件的“距离”）上的分类，但是，对于另一个维度“算法”，或者说算子的粒度，仍无法支持。因此，我们将进一步提供多层次算法维度上的支持。层次化的算子粒度和抽象层次无关，因此在高级算子和低级算子中都有体现。我们按照张量的维数（rank）进行划分，分成 0 维度（标量），1 维数据（向量），2 维数据（矩阵），以及大于 1 的 n 维数据（张量）四种层次。中间层中所支持的算子如表 5.1所示。

• **标量（Scalar）**。标量是最细粒度的算子，其输入输出数据都是 0 维数据。现在，大多数加速器并不支持标量计算，但还是有一些加速器，比如 Cambricon 中包括

表 5.1: 不同计算粒度的中间语言算子

粒度	操作
张量	Conv., Max-pool, Avg-pool, LRN, LCN, FC
矩阵	MMV, VMM, matrix add/sub/mul/div
向量	VADD, VSUB, VMUL, VDIV, VGT, VLT, VEQ
标量	SADD, SSUB, SMUL, SDIV, SGT, SLT. SEQ

Listing 5.1: 样例代码：用高级中间语言实现一个卷积层和池化层

```
Dim n(1, 1);
Dim input_channel(96, 64); // partition into 3 segments
Dim input_spatial(57, 57);
Dim conv_channel(384, 64); // partition into 6 segments
Dim conv_spatial(28, 28);
Dim k(3, 3);
Dim pool_spatial(24, 24);
int conv_stride = 2;
int pool_stride = 2;

// define data with Dim
Neuron input(n, input_channel, input_spatial, input_spatial, NHWC);
Neuron conv_output(n, conv_channel, conv_spatial, conv_spatial, NHWC);
Neuron pool_output(n, conv_channel, pool_spatial, pool_spatial, NHWC);
Synapse weight(conv_channel, k, k, input_channel, OHWI);
Neuron bias(conv_channel);

// Call HLOP to perform conv and pool layers
ConvForward(conv_output, input, weight, bias, conv_stride, conv_stride)
    ;
MaxPoolForward(conv_output, pool_output, pool_stride, pool_stride);
```

了标量计算的功能，这部分功能主要用来控制循环，以及在反向计算中用来更新权值的梯度。同事，我们在中间语言中支持标量也是为了保证功能的完整性。

• **向量（Vector）**。向量算子的输入输出数据都是 1 维数据，向量算子可以被转化成标量数据。向量操作是神经网络算法中的基本操作，比如，Batch Normalization，Pooling 算法都可以拆解成向量操作。向量操作也是很多深度学习加速器中会支持的一类操作，比如 Cambricon，ShiDianNao。

• **矩阵（Matrix）**。矩阵算子的输入输出数据都是 2 维数据，矩阵算子可以被转化成向量计算，矩阵操作是实现神经网络算法中的基本操作，比如卷积，全连接都可以被拆分成矩阵操作。矩阵操作是多数深度学习加速器都会支持的一类操作。

• **张量（Tensor）**。张量的输入输出数据都是维度大于 2 的数据。张量的算子可以对应深度学习中算法，比如，一个完整的卷积操作，池化操作，全连接操作等。张量算子是深度学习算子中的主体部分，尤其是类似卷积这种热点操作，多数深度学习加速器都会直接提供相应的张量操作来加速神经网络中的常用算子。

Listing 5.2: 样例代码：用低级中间语言实现卷积网络

```
// define HLOP with LLOP
void HLConvForward(Neuron &output, Neuron &input,  Synapse &weight, int
    stride_h, int stride_w){
    int co_seg = output.co_segn();
    int ci_seg = input.ci_segn();

    for(int o = 0; o < co_seg; ++o){
        NeuronBuffer output_buffer = NBMem->allocate(output.segsize());
        NeuronBuffer partial_buffer = NBMem->allocate(output.segsize())
            ;
        for(int i = 0; i < ci_seg; ++i){
            // allocate space on on-chip buffers
            NeuronBuffer input_buffer = NBMem->allocate(input.segsize
                (1,1,1,i));
            SynapseBuffer synapse_buffer = SBMem->allocate(weight.
                segsize(o,1,1,i));
            // load data from main memory to on-chip buffers
            NBMem->load(input.seg(1,1,1,i),input_buffer);
            SBMem->load(weight.seg(o,1,1,i),synapse_buffer);
            // perform convolution
            LLConvForward(partial_buffer,input_buffer,synapse_buffer,
                stride_h,stride_w);
            LLEltAdd(output_buffer,partial_buffer,output_buffer);
        }
        NBMem->store(output_buffer, output.seg(1, 1, 1, o));
    }
}
```

5.2.4.3 访存操作

深度学习加速器通常包含多块片上缓存，用来存放指令的操作数。每次计算，我们需要将主存中的一块数据加载到片上缓存中。不同的加速器可能有差异很大的存储层次，为了兼容不同的缓存设计，我们采用一种可配置的缓存注册机制，让用户可以自己配置其硬件的缓存属性，包括缓存的数量，大小，延迟，带宽等。指令生成器在调度过程中通过读取缓存的参数也可以更好的评估运行时间进行调度。

配置缓存时，我们首先对加速器的存储结构抽象成一系列可定义的参数（如表 5.2所示），将参数的可配置接口提供给加速器的厂商，让他们可以自定义缓存。每个定义的缓存进行独立的地址空间管理。我们提供一系列操作实现对片上缓存

的操，包括分配（allocate），释放（release），加载（load），存储（store），移动（move）。比如，代码 5.2 中，11,12 行表示在神经元缓存和权值缓存分别分配一块存储空间，14,15 行表示将一块数据加载到刚才分配的空间中。

表 5.2: 定义缓存属性

属性	例子（DianNao 中的 NBin）
名字	NBin
大小	2KB
可加载	设备内存
访存延迟	100ns
带宽	250GB/s

5.2.5 数据管理

深度学习加速器的编程中一个重要问题就是对数据的处理和管理。本节中，我们将介绍 DLIR 中对数据的分配和释放，数据摆放方法，以及对数据进行分段的策略。

5.2.5.1 片上数据的分配和释放

为了管理片上数据，我们提出一种简单的片上数据管理器来管理片上数据的释放和存储。对于每一个注册的片上数据，我们为其生成一个数据管理模块。可以实现对片上空间的分配，释放，以及合并相邻的地址碎片。

- **数据分配**。每当分配一块片上数据，我们会从第一个索引项开始，搜索所有的索引项地址块的大小，找到一块大小最相近的可用空间，对这块空间进行分配，将其标志为“已分配”，并将剩下的空间作为一个新的索引项插入索引表中。

- **数据释放**。当释放一块片上数据，我们会首先将这块地址的状态标志为“可用”，然后检查这块被释放的地址相邻的地址空间是否是“可用”，如果是，则将两块数据拼接在一起，如果两边的两块数据都是可用的，则将三块数据拼成一块。

5.2.5.2 数据摆放

现在的深度学习框架中，数据的排布通常采用枚举类型来指定，比如 cuDNN，TensorFlow 中使用的 NCHW，NHWC。它们分别表示数据的维度顺序为样本数，

特征图数量，特征图高度，特征图宽度，以及样本数，特征图高度，特征图宽度，特征图数量。这样的表达方式非常不灵活，无法满足深度学习加速器对于数据摆放的复杂需求。当进行计算式，深度学习加速器通常需要读取一段较长的数据，如果不进行数据重新摆放，这段所需的数据可能是放在一段不连续的空间的。这种情况下，加载这段数据就需要多条指令才能完成。而进行数据重新排列之后，我们可以讲这几段分离的数据放到连续的地址空间中，因此减少指令读取次数，从而减少访存开销。

为了满足数据的连续读取，一个 4 维的 Tensor 如果最内部的维度被进行了拆分，那么就需要进行数据的重新排列，而且在优化原语的作用下，这些维度的顺序还可能被调换。这种情况下就更需要进行数据重新摆放。我们将 Tensor 数据的重新摆放抽象成用多个循环对 Tensor 进行拷贝。以下面的代码为例，其实现了将一个大小为 16(维度 b)×1024(维度 i) 的 Tensor 的数据进行重新排列的逻辑。i 维度被划分成 32×32 的块，然后调换了计算顺序：从 b 到 i-outter 到 i-inner 调换成了 i-outter, i-inner，b。因此需要对数据进行重新摆放以满足数据的连续读取。

```
produce Reshaped_data{
  for (i-outter, 0, 32, 1){
    for (i-inner, 0, 32, 1){
      for(b, 0, 16, 1){
        rd[x++] = data[b*1024+i-outter*32+i-inner];
      }
    }
  }
}
```

5.2.5.3 数据分段决策

数据分段是对一块大的数据进行合理的划分，使得划分后的每一小段数据可以放到片上的缓存中。分段大小会影响并行性，如果分段非常小，则表示访存的次数增加，由于每次访存需要一个启动时间，当访存次数增加，则总体的访存时间则会增加，这样对访存密集型的算法，比如全连接层，是不利的。但是如果分段比较大，对于计算密集型的运算，在指令没有充分并行的情况下可能导致较长的等待时间。本节中，我们首先定义数据分段问题，然后提出一种贪婪算法可以快速的找到一个解决方案。并且用一个全连接层作为例子解释具体的过程。

问题定义。数据分段策略是要估算出段的大小，即在解空间内搜索最优解，因此我们首先定义问题，提出我们使用的目标函数。

我们将段大小，即需要求解的量，用变量 $x_1, x_2, ...x_n$ 表示，n 是维度的数量。我们的目标是要找到一组 x，能够满足对于所有的 m，$F_m(x_1, x_2, ...x_n) \leq MemSize_m$，

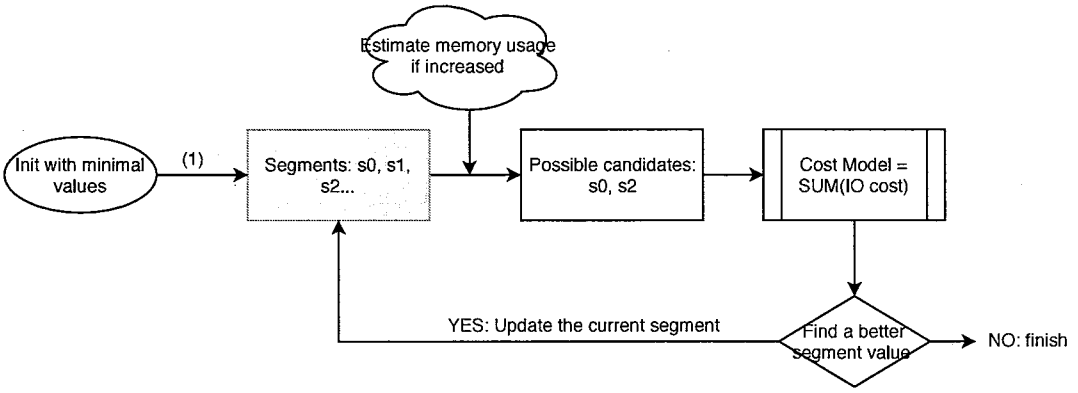


图 5.7: 数据划分算法流程

其中 m 表示一块片上缓存的大小的标识符。F_m 是计算每一种分段策略对于内存 m 所需要的空间大小，这个值的计算可以通过分析 ALLOC 和 RELEASE 得到。

我们用一个输入为 1024，输出为 256 的全连接层作为例子，我们的后端采用的是 Cambricon，其包括两块片上缓存（大小分别是 64KB 和 732KB），利用矩阵操作进行全连接层的运算。其代码如下所示。其中，Dimension 在被分段的时候，分段的大小还没有被确定，其返回值是未决定的变量。维度 $n = 2$，变量 x_1，即 in_seg_size, x_2 即为 out_seg_size, $m = [vmem, mmem]$, $F_{vmem}(x_1, x_2) = x_1 + x_2 \times 2, F_{mmem}(x_1, x_2) = x_1 \times x_2$, 因此, 约束条件即为: $x_1 + x_2 \times 2 < 32K \&\& x_1 \times x_2 < 384K$

```
1  Dim input_dim(1024);
2  Var in_seg_size = input_dim.tile();
3  Var in_parts = input_dim.partn();
4  Dim output_dim(256);
5  Var out_seg_size = output_dim.tile();
6  Var out_parts = output_dim.partn();
7  Tensor input(input_dim, "input");
8  Tensor output(output_dim, "output");
9  Tensor weight(output_dim, input_dim, "weight");
10 VMem in_buffer = Alloc(in_seg_size);
11 VMem out_buffer = Alloc(out_seg_size);
12 VMem temp = Alloc(out_seg_size);
13 MMem w_buffer = Alloc(in_seg_size * out_seg_size);
14 for (o, 0, out_parts, 1) {
15   for (i, 0, in_parts, 1) {
16     Load(in_buffer, input[i]);
17     Load(w_buffer, weight[o][i]);
18     Matmul(temp, in_buffer, w_buffer);
19     Vadd(out_buffer, temp, out_buffer);
20   }
21   Store(output[o], out_buffer);
```

```
22    }
23    Release(in_buffer);
24    Release(out_buffer);
25    Release(w_buffer);
26    Release(temp);
```

之后，我们估算出总的访存时间作为目标函数，我们的目标就是最小化访存时间。我们只评估访存时间而不评估计算时间，原因有两点，首先，分段大小对于总的计算时间来说没有太多影响，第二，中间表示的层次不利于去评估每条计算语句的性能。

5.2.4.3小节中介绍的缓存注册机制，让我们可以获得 Load 和 Store 某一块存储区域的访存延迟和带宽，因此，可以估算出一条 Load 和 Store 的访存时间。比如，在全连接层的例子中，对输入的一次 Load 的访存时间为：in_seg_size $\div B_{vmem} + L_{vmem}$，其中 B_{vmem} 表示缓存的带宽，L_{vmem} 表示的是一次读取的延迟。对于输入数据总的 Load 时间即为 in_parts 乘以 Load 的一次访存时间。再通过循环次数，我们可以计算出整体的访存时间。

贪婪算法。我们将分段策略的目标定义为最小化访存时间，同时尽可能的把存储器用满。为了找到最优解，最直接的方法就是去搜索整个设计空间，然后对每一个空间评估出时间，然后选择一个时间最短的，这个方法可以保证获得最优解，但是由于设计空间非常大，编译的时间会过于长，因此，我们提出一种贪婪算法，可以显著的缩短搜索时间，同时获得较好的内存利用率。在 AlexNet 网络上，我们将搜索步数缩短了 161.44×。当使用双缓冲优化时，分段情况会稍微不同，我们会将内存的大小缩小到原来的一半进行搜索。算法流程如图 5.7所示，具体步骤如下：

- 初始化。首先将段大小设置为 $x_i(i = 1, 2, ..., n)$，设置成一个最小数值。之后的步骤中，这些数值会逐渐增加，直到达到内存限制为止。
- 获得需要被增加的维度。我们用 $M(i)$ 来评估当第 i 个维度增加之后，会造成的访存开销，选择访存开销最小的维度 i 作为要增加的维度。然后增加这个选出维度的分段大小。
- 不断重复第二步直到没有任何一个维度的分段大小可以继续被增加。程序退出。

对于有多个存储结构的架构来说，分段策略可以递归的应用在没两层存储层次上。当对某一级存储的分段结束后，我们会继续对更高一级（更接近计算，分段大小更小）的维度进行分段。

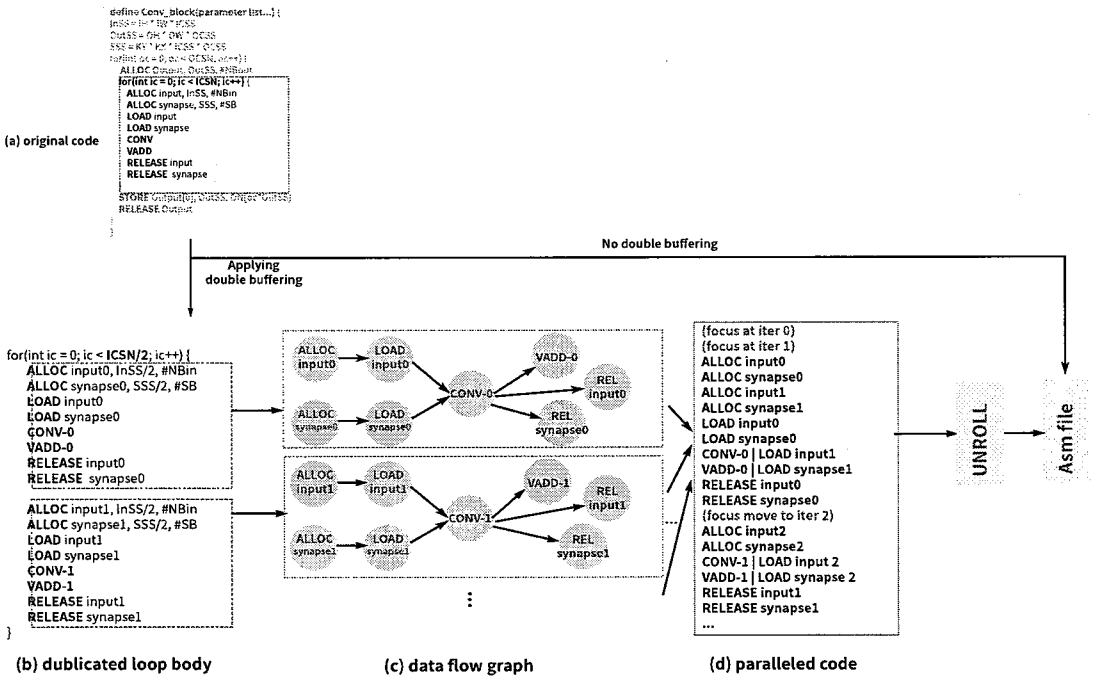


图 5.8: 双缓冲

5.2.6 指令生成

指令生成主要针对用低级算子实现的高级算子，这些算子会首先被展开成循环的形式，然后对其进行相应的优化调度，最后生成指令。在调度中，我们采用双缓冲（double buffering）技术进行优化。因为深度学习算法具有计算和访存密集的特点，进行了分段后，不同数据段的计算之间不存在依赖关系，因此非常适合使用双缓冲技术，来增加访存和计算的并行。没有双缓冲的指令生成比较简单，可以直接看作是对循环的展开。在展开循环时，有两种方式，对于支持条件跳转的架构，比如 Cambricon，我们用跳转指令实现循环，对于不支持条件跳转的架构，由于此时分段的数值都已经确定，我们直接将循环展开，在展开过程中替换循环的变量。

为了自动实现双缓冲技术，我们提出一种基于图的算法，将每两次迭代合成一次，对数据依赖图进行拓扑排序，确定指令的执行顺序。其主要如下所示：

- **循环变换**。双缓冲技术可以将一块内存分成两个部分使用，然后对两个循环之间的访存和计算操作并行执行。我们通过将相邻的两次循环合并起来来实现。这个过程如图 5.8所示。我们用卷积操作作为一个例子进行讲解。首先，我们根据用户标识的原语得到需要进行双缓冲优化的循环（ 5.8中的 (a)）。将迭代展开，然后，我们将两次迭代合并成一次迭代，修改语句的地址（变量名）。比如图 5.8-(b) 中，两个输入都被分配在输入缓存上:input1 和 input2。当 input1 被用来计算的时候，input2 就可以加载。
- **生成图**。之后，我们将循环体的代码翻译成一个有向图，用来表示数据之间的依

赖。两次迭代会被分别翻译成两个独立的图，如图 5.8-(c) 所示。由于这两个有向图是从同一段代码中生成，它们的拓扑结构是相同的，差别只是在地址和迭代数上。

• **生成并行代码**。生成了计算图之后，我们会根据这两个图生成指令。我们先定义生成过程中需要用到的三个不同的数据结构。

1. *焦点窗口*。焦点窗口会指向当前正在生成的有向图，窗口的大小为 2，用来装下两次迭代的大小，这两次迭代可能会同时用到片上资源，因此需要同时进行分配。当一次迭代中的指令都被分配出去之后，窗口就会自动向后滑动一次迭代。图 5.8-(c) 中，迭代 0 和迭代 1 都在窗口中，当有向图 0 中的最后的一个节点（RELEASE synapse0）被释放之后，有向图 0 中的节点都被释放，窗口会向后滑动一次迭代，也就是将有向图 2 放到窗口中。通过使用焦点窗口，我们可以节省保存多个有向图的空间，而且由于有向图的结构都是一样的，我们可以不需要重复的构建删除有向图，而是可以直接重用有向图结构。

2. *顺序容器*。顺序容器是指令调度优化中的原子结构，可以看成是一块顺序的代码段。在调度中，一个顺序容器中的指令会被顺序的生成。在有向图中，顺序节点会当成一个图节点进行调度。计算粒度很低的架构，可以利用顺序容器来保存一段计算指令，这段指令可能是由标量构成的。比如，用矩阵粒度的指令来实现一个卷积操作，因为这些矩阵操作都是在矩阵运算单元上执行的，因此需要顺序执行。这种情况下，一个卷积的语句会被装在顺序容器中，而顺序容器就可以作为一个卷积算子的节点一样进行调度。

3. *并行容器*。并行容器可以用来存放可以并行执行的语句或者顺序容器。并行容器中的所有内容都会被并行的翻译。

4. *资源表*。资源表是用来追踪指令生成过程中的资源使用情况的。由于所有的硬件资源都是通过抽象的方式定义，因此，每当选择了一个原子操作后，资源会相应的发生变化，比如某一运算器的会被占用。我们在调度过程中会通过资源的使用情况寻找可以并行执行的指令。

现在我们来描述具体的将有向图翻译成指令序列的过程。

- *找到可用的调度单元*。这一步中，我们构造一个调度单元的集合 $S = u_1, u_2, ...u_n$，由可用的调度单元构成。一个可用的调度单元需要满足两个条件：

  (1) 这个单元所需要的所有资源都是可用的。这一步通过检查资源表完成。(2) 这个单元没有未完成的依赖节点。比如，图 5.8-（c）中，当 LOAD input0，

LOAD synapse0 执行之后，Conv-0 就准备好了，而这个时候运算单元是可用的。

- *找到可以并行执行的调度单元。*从上一步构建的集合中，我们可以继续辨认出那些可以并行执行的指令，然后将它们融合成并行单元。方法是通过检查资源表，找到两个可以同时执行的运算单元。比如 CONV 和 LOAD 使用的资源完全不同，因此可以并行执行。在图 5.8的例子中 CONV1 和 LOAD input2 是可以并行执行的，它们会被放入并行容器中，作为下一步考虑的一个选项。这种情况下，原始的集合 S 会被转化成 $S' = u_i, u_2, ...[u_i, u_j]_k, ...u_m$，其中 u_i 和 u_j 会被合并成一个并行单元。总的调度单元你数量会从 n 变成 m。

- *选择一个调度单元。*这一步骤中，我们会选择出一个调度单元，将其推入到指令序列中。S' 这个情况下存在三种状态：

  1. $m = 0$，表示当前的计算图翻译已经结束，我们可以直接结束步骤。
  2. $m = 1$，表示我们只有一种可以选择的单元，因此我们选择这唯一的调度单元，更新资源表，然后跳转回第一步，更新集合。
  3. $m > 1$，如果存在并行单元，则选择序号最小的单元（序号越小表明越早被收入集合）；否则，选择序号最小的单元，然后跳转会第一步。

5.3 性能评估

5.3.1 实验环境

5.3.1.1 后端

我们选择了三种最近提出的加速器作为后端进行实验，Cambricon[9]，Cambricon-X[8]，和 ShiDianNao[58]，并且用 GPU 作为比较基准。我们重新实现了这三种加速器的模拟器，实现参数按照加速器的发表论文进行设置。

- **Cambricon[9] (ACC-C)**。Cambricon 是一个 load-store 架构，包含 43 条指令，被分成四类，计算指令，逻辑指令，控制指令和数据搬运指令。Cambricon 指令集的文章中也同时提出了一个加速器的原型，可以充分的支持 ISA。本文的试验中我们使用的参数配置如表 5.3所示。其中，C/H/W 分别表示加速器可以一次处理的 Tensor 的特征图数量/特征图高度/特征图宽度的三个维度大小。Cambricon 指令集由矩阵指令，向量指令和标量指令三种构成，这三类指令正好可以和 LLIR 中的矩阵，向量，标量粒度的算子相互对应

表 5.3: 后端加速器配置参数

加速器	缓存（KB）	C	H	W
ACC-C-S	Matrix Mem(32),Vector Mem(16)	16	1	1
ACC-C-M	Matrix Mem(128),Vector Mem(128)	8	1	1
ACC-C (Origin)	Matrix Mem(768),Vector Mem(64)	32	1	1
ACC-X (Origin)	NBin(8), NBout(8), SB(32)	16	1	1
ACC-X-M	NBin(64), NBout(64), SB(128)	8	1	1
ACC-X-L	NBin(32), NBout(32), SB(768)	32	1	1
ACC-S-S	NBin(8), NBout(8), SB(32)	1	16	16
ACC-S (Origin)	NBin(64), NBout(64), SB(128)	1	8	8
ACC-S-L	NBin(32), NBout(32), SB(768)	1	32	32

的。Cambricon 中包含两个片上缓存，矩阵缓存和向量缓存，缓存的大小如表 5.3所示。

- **Cambricon-X[8] (ACC-X)**。Cambricon-X 是 2016 年提出的一个神经网络加速器，其不仅可以加速一般的神经网络，还可以利用神经网络中的稀疏性进一步进行加速。由于 Cambricon-X 的文章中没有明确的列出其指令集，我们根据其文章中加速器所提供的功能推理其指令集。Cambricon-X 的架构是一种基于向量的架构，因此可以包含高于向量粒度的指令。同时，Cambricon-X 主要支持的深度学习算法，如卷积，池化，全连接等，可以定义为 tensor 粒度的操作。如表 5.3所示，Cambricon-X 片上有三块缓存，NBin，NBout 和 SB。

- **ShiDianNao[58] (ACC-S)**。ShiDianNao 是一款专门为卷积神经网络的图像应用设计的低功耗加速器，主要用于在摄像头内部。原版的 ShiDianNao 针对的是小规模的，可以全部放在片上的网络，比如 LeNet5，因而在计算过程中没有对 DRAM 的访问。本文的试验中，我们将 ShiDianNao 连接到一个全局的 DRAM，同其他的加速器一样，进行数据的划分之后计算。ShiDianNao 的架构是基于脉动阵列机的类型。ShiDianNao 支持 Tensor 粒度的卷积，池化和全连接层，此外还包括矩阵和向量的计算。我们将 ShiDianNao 的指令集定义为 tensor 和 matrix 的粒度。

- **GPU**。GPU 被广泛的用于神经网络计算中。我们提供一组 GPU 实验数据作为基准。我们用的 GPU 卡是 NVIDIA K40M，内存 12GB，28nm 的工艺，可以达到峰值性能 4.29TFlops。所有的 GPU 实验都是在 Caffe[88]（cuDNN[68] 作为后端）平台上实现的。

5.3.1.2 前端

中间表示的输入是一个计算图，可以从任意编程框架生成，这里我们使用 Caffe 作为前端框架进行实验，输入文件即配置网络所使用的 prototxt 文件。

5.3.1.3 Benchmark

我们选择了六个网络作为 benchmark 来评估 DLIR 的性能，即：LeNet-5[1]，Cifar10-quick-model[14]，AlexNet[25]，VGG16[13]，VGG19[13]，以及 ResNet34[4]。其中 LeNet-5 和 Cifar10-quick-model 属于小规模网络，AlexNet，VGG16，VGG19 以及 ResNet34 属于大规模的网络。这些网络涵盖了 5 种不同的神经网络算子：卷积，池化，LRN，全连接以及 Batch Normalization。

5.3.1.4 Baseline

实验中，我们使用三种不同的加速器，为了保证公平，每种加速器我们采用三种不同的规模，如表 5.3所示，相当于采用 9 种不同的硬件后端。软件的维度上，我们评估三种不同实现方式的性能：充分进行过手动优化的代码（Hand-optimized），不采用双缓冲的 DLIR 代码（DLIR-naive），以及采用双缓冲的 DLIR 代码（DLIR-db）。

5.3.2 实验结果

本节中我们对中间层语言的性能进行测试。为了表明我们重新实现的加速器的模拟器可以正确的模拟硬件，我们和原始文章中的数据进行对比。

5.3.2.1 原版性能

为了说明我们重新实现的加速器是正确的，我们和原版的比较了性能。由于 Cambricon 的原始数据是和 K40 相比，而 ShiDianNao 和 Cambricon-X 是和 K20 相比较的，我们使用 K40 和 K20 两种设备作为基准。在 Cambricon 上，原始 Cambricon 和我们重新实现的 Cambricon 对 GPU 的加速比相差无几（3.19× vs. 3.09×）。我们从 Cambricon-X 的原始文章中选择了三种网络，并且，在这三种网络上做了实验，比较了 ACC-X 和 Cambricon-X 之间的性能。我们的性能数据归一化到 K20 的性能数据上，达到了 5.1× 的加速比，和原论文中的 4.9× 接近。为了比较 ACC-S 和原始的 ShiDianNao，我们在原始文章中的 10 种 benchmark 上做了实验，ACC-S 达到了 25.2× 的加速比，原文中的加速比是 28.9×。我们认为我们所实现的加速器是合理的，基本可以和原文中的加速器达到接近的性能。我

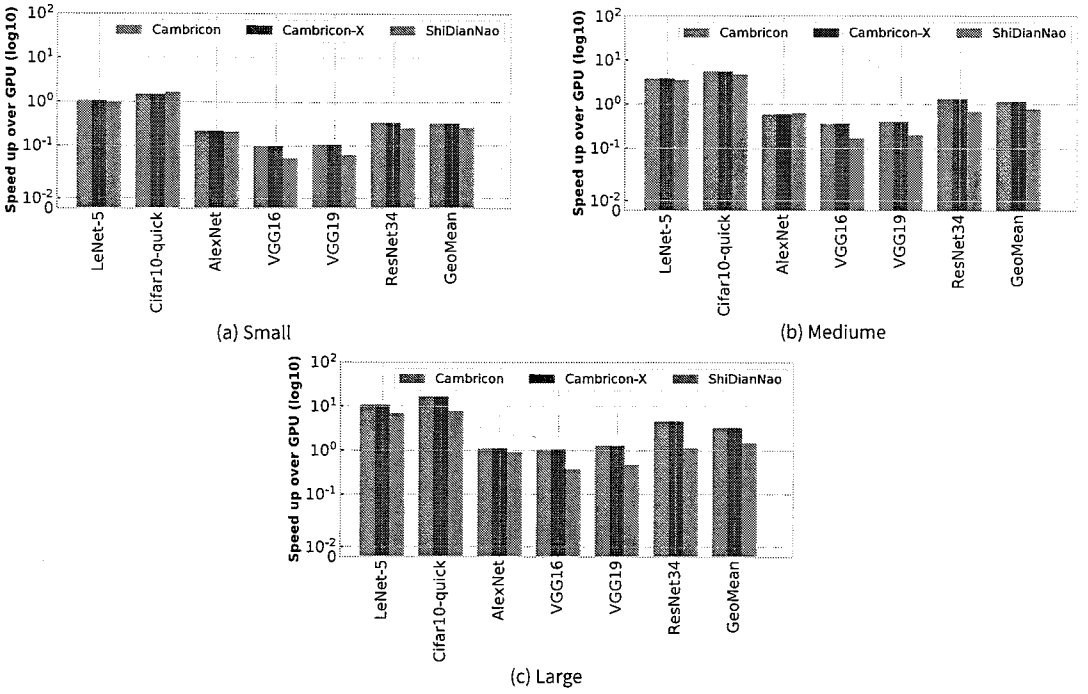


图 5.9: 不同规模硬件运行效率

们实现的加速比和原文中的相比稍微慢一些，这是由于指令生成造成的，而原文中的数据是最优化的数据结果。

5.3.2.2 不同硬件参数下的性能

为了表明中间层对于硬件架构和硬件参数良好的可适应性，我们调整三种架构的硬件参数，分别进行实验比较。我们将三种加速器的计算单元数量，缓存大小等分别设置为三种不同大小，具体数值如表 5.3所示。没有后缀的表明原始大小，M（Mediate）表示中等资源配置，S（Small）表示较少的资源配置，L（Lareg）表示较多的资源配置。

图 5.9中，我们汇报了不同配置的硬件在 6 种网络上的性能加速比。通过观察不同的架构之间的性能差异，我们发现，DLIR 的指令生成器可以很好的适应不同硬件规模，三种加速器都可以从硬件资源的提升中获得性能提升。其中，ACC-S 的性能要比 ACC-C 和 ACC-X 的性能差一些，尤其是 ACC-S-L 的加速比明显低于 ACC-C 和 ACC-X，这是由于其架构是基于脉动阵列机的，需要特征图的长宽两个方向小于其运算单元的数量，才可能获得性能提升，因此，对于 ACC-S 来说，虽然运算单元的数量增加，但是一部分的资源却被浪费掉了。

此外，我们也研究了不同存储参数对硬件性能的影响，包括读取延迟和带宽的影响。我们比较了 9 种延迟时间（从 0 到 400ns）对加速比的影响，在图 5.10中展现了加速比的变化。为了更加明显的展现单个层受影响的情况，我们从 6 种网络

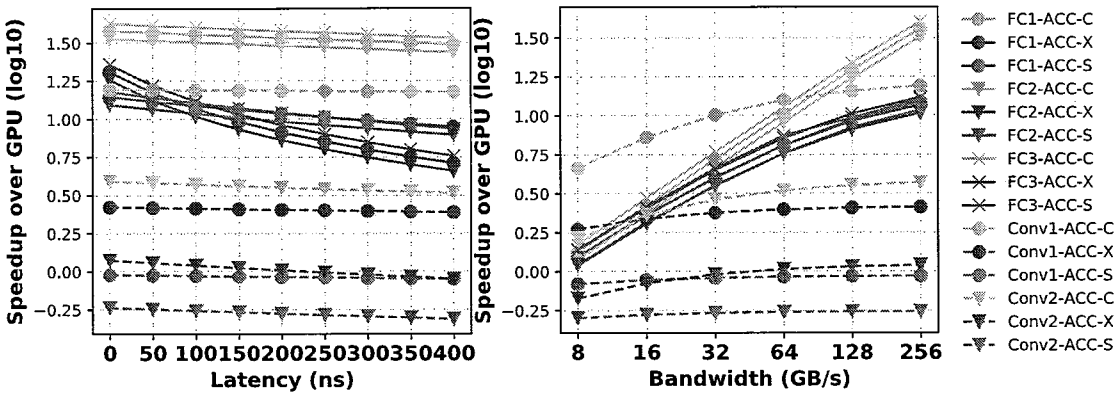


图 5.10: 不同硬件参数对于 GPU 的加速比

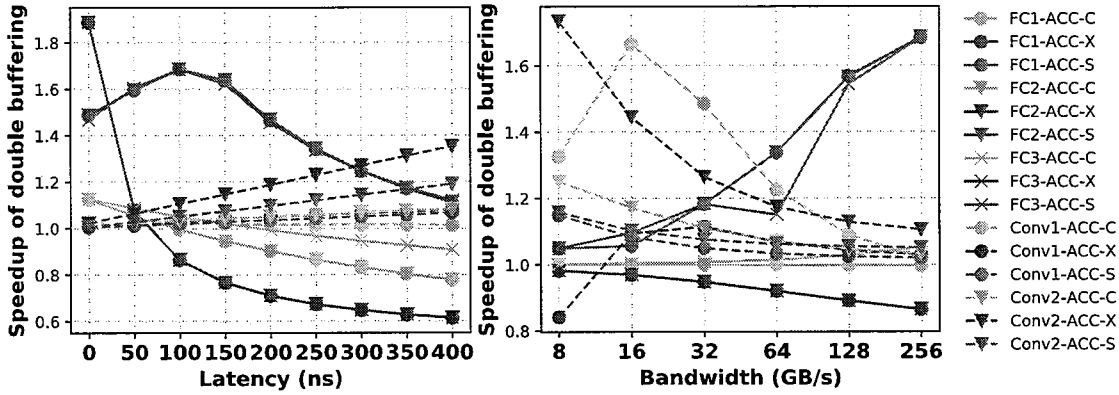


图 5.11: 使用双缓冲和不用双缓冲的加速器比

中，抽取出 5 个有代表性的单层进行实验，2 个全连接层和 3 个卷积层。AlexNet 网络中的 fc1 (9216-4096), fc2 (4096-4096), conv1 (27×27×96-27×27×256)，以及 VGG 网络中的 fc3 (4096-1000), conv2 (112×112×64-112×112×128)。除了延迟，我们还研究了不同带宽（8GB/s 到 256GB/s）对性能的影响。结果在图 5.10中所示。可以看到，全连接层，作为访存密集型的操作比计算密集型的卷基层更能够从延迟的下降和带宽的上升中获益。

5.3.2.3 双缓冲

原始规模。本节中，我们评估使用双缓冲优化后能够取得的加速比，以及各种硬件参数变化对这个优化的影响。图 5.12展示了利用双缓冲取得的性能提升，Cambricon, Cambricon-X 以及 ShiDianNao 分别达到了 6.56%, 3.29%, 以及 1.27%。这三种架构中，ACC-C 在双缓冲的收益最高，而 ACC-C 的收益较少（1.07×）。导致这种差异的原因在于计算资源的差距。ACC-S 的计算资源比较少只有 8x8 的计算单元，而 ACC-C 的计算资源很多，每一拍可以计算 32x32 的计算量，相差 16

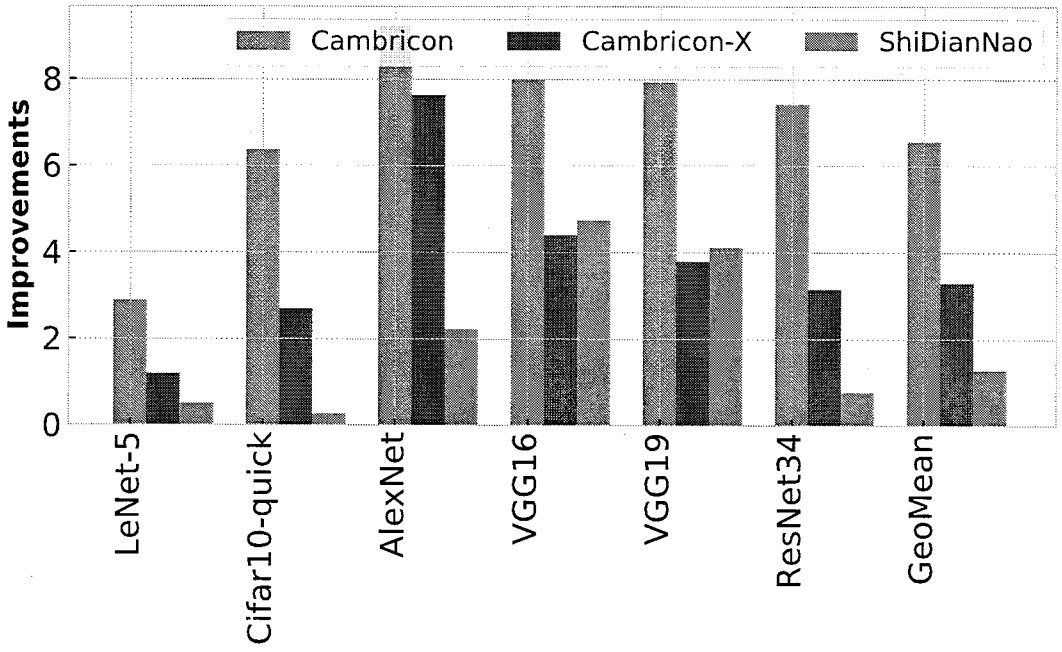


图 5.12: 利用双缓冲取得性能提升

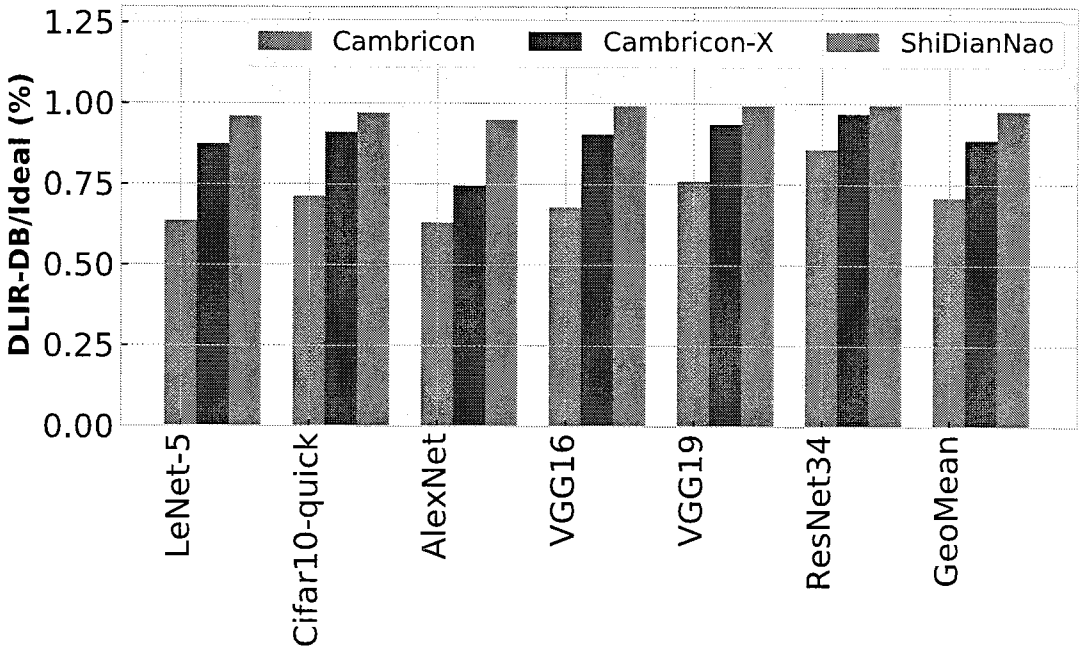


图 5.13: DLIR 双缓冲和手写优化指令相比

倍。因此。ShiDianNao 的计算更加密集，当它们的访存量差距不大的情况下，即使访存全部被覆盖，整体减少的时间占比也比较少。

我们进一步和手写指令优化情况下的性能提升比较。手工优化情况下，利用双缓冲可以基本将访存或者计算完全隐藏，即运算时间是访存和计算中较长的那个，实验结果如图 5.13所示，采用了自动双缓冲的指令生成可以达到手写指令优化效率的 70.8%-97%。

片上缓存。我们观察到，使用双缓冲技术实际上不一定总是能够提升性能。性能的损失是由每一次访存的延迟造成的。双缓冲带来的性能提升是通过让访存和计算并行执行，使得时间短的操作可以隐藏起来。在顺序执行时，总的执行时间可以表示为：$T = T_c + T_{IO}$，而 $T_{IO} = \sigma_{i=0}^{n-1}(S_i/B + L)$，其中，$T_c$ 表示计算时间，n 表示访存操作数，对于每一个访存操作 i，访存的数据大小为 S_i。当使用双缓冲的时候，内存被分成两块让计算和访存操作可以并行执行，因此，当 S_i 减少，也就是分块越小，n 会越大。因为数据传输的时间不变，n 增大就会导致 IO 的时间增加，变化的 IO 时间表示为 $\delta T_{IO} = L \times \delta n$。理想中，计算密集型的层，比如卷积，不应该被这种增长所影响，因为这些层的整体时间都可以被评估为 $Max(T_C, T_{IO})$，其中 T_C 远远大于 T_{IO}，即使 IO 的时间会因为双缓冲而增加，T_C 也可以覆盖 T_{IO}。然而，对于 IO 密集型的算法，使用双缓冲可能会导致 IO 的时间进一步增加，比被隐藏起来的计算时间更多，从而导致性能下降。

从实验数据中，我们观察到以下现象。

- 对于两个卷积层，所有的加速器都可以从双缓冲中获益。这个结果很容易理解，根据之前的分析，我们知道对于像卷积层这样的计算密集型运算，几乎必然是获益的。
- 对于访存密集型的全连接层，加速器 ACC-C 和 ACC-X 会因为访存的延迟而导致加速比下降，甚至随着延迟的增加，性能会进一步恶化。
- 全连接层在 ACC-S 加速器上只有在 100ns 以下的延迟上可以获得加速比，当延迟增加到 100ns 以上时，性能就会下降。关键的原因在于 ACC-S 上有很大的缓存空间，而运算单元比较少。如果加载新的数据可以在处理加载后的。我们将 ACC-S 的缓存大小缩小到 ACC-X 的大小，发现它们的行为变得类似了。
- 更高的带宽可能会导致双缓冲的效果下降。我们观察到在 ACC-C 上，16GB/s 的带宽是一个转折点（conv1）。

5.4 本章小结

本节中，我们提出一种面向深度学习加速器的中间层，包括两个层次的中间表示，数据管理器，以及指令调度器。高级中间表示可以用来表示从上层框架中传递下来的，和硬件独立的算子，高级中间表示可以被转化为低级中间表示，即和硬件指令，硬件资源相关的中间表示此外，中间层中的数据管理模块可以进行自动的数据划分，指令调度器可以进行自动的双缓冲。我们在 3 种不同的深度学习处理器架构，Cambricon, Cambricon-X, ShiDianNao 上评估中间层的性能，实验结果表明，在 6 种大的神经网络模型上，自动指令生成可以达到和手写指令接近的性能，利用双缓冲技术可以达到手写优化指令性能的 70.8%-97.7%。

第 6 章　总结与展望

本文提出一种面向深度学习加速器的软件栈，包括应用程序层，编程框架层，中间表示层，汇编层，加速库，以及指令集层，解决了深度学习处理器的编程问题，着重研究了其中的高性能库，中间表示层和高级汇编层。该软件栈可以在不改动应用程序的情况下，直接运行在加速器上，同时支持了框架中的大量算子，并且能够充分保障执行效率。

软件栈中包含了两种从编程框架调用加速器的方法，一种是直接通过高性能库调用，另一种是通过中间表示和汇编语言两个层次生成指令调用。这二者所满足的需求及设计的出发点各不相同。高性能库主要满足的是对性能的需求，其设计的出发点是能够高效支持一种特定的应用，比如本文中的深度学习算法，而提供一系列操作（API），并且尽可能的优化这些操作，达到更好的效率。而中间表示和汇编语言的提出是为了满足灵活性和可移植性，其设计主要是从硬件的角度出发，把硬件当做一个平台，尽可能充分的抽象出硬件的功能，让开发者可以利用硬件开发各种类型的算法和应用而不仅仅局限于某一类算法上。而中间表达为了保证灵活性，必然会牺牲一部分效率。

随着深度学习处理器的功能变得更加强大，我们可以预想未来会有更多类型的算法在加速器上实现，上层的编程框架也可能会不断变化，出现更多新的领域专用语言和框架。而这些新的编程语言或者框架编写的新算法也都和深度学习算法一样，可以通过中间表示转化为加速器上的程序。同时，只要未来高效实现某种特定应用算法的需求存在，高性能库的存在就是必不可少的。

我们未来的研究方向主要包括 3 点。首先，我们会进一步的完善中间表示层的表达能力，使其能够支持更多深度学习处理器的功能，比如对数据的稀疏化表示，数据压缩，量化，低精度计算等功能的支持。其次，我们希望将现在的编程模型扩展至多核加速器上，进一步研究多核上的数据拆分策略以及优化问题。最后，我们希望可以进一步研究将深度学习处理器集成到分布式系统中的优化方法，从软件栈的角度需要做那些方面的改善。