

# 中国科学技术大学

# 硕士学位论文



## Android 平台的 CNN 模型 能效优化问题研究

作者姓名： 王震

学科专业： 计算机系统结构

导师姓名： 周学海 教授 李曦 副教授

完成时间： 二〇一八年四月



University of Science and Technology of China  
A dissertation for master's degree



**Research about Energy  
Efficiency Optimization of CNN  
Models on Android Platform**

Author: Zhen Wang

Speciality: Computer Architecture

Supervisors: Prof. Xuehai Zhou, Assoc. Prof. Xi Li

Finished time: April, 2018



## 中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：\_\_\_\_\_

签字日期：\_\_\_\_\_

## 中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

公开    保密（\_\_\_\_年）

作者签名：\_\_\_\_\_

导师签名：\_\_\_\_\_

签字日期：\_\_\_\_\_

签字日期：\_\_\_\_\_



## 摘要

近年来，卷积神经网络（CNNs）因其高推断精度和强自适应性而被广泛应用于各种领域，例如：计算机视觉、语音识别等。另一方面，移动手机当前已经成为人类日常生活中的随身携带之物，并且每天都产生着大量与人类相关的传感数据。为了让手机更加智能地服务于人类，许多工程项目也尝试着在移动端利用卷积神经网络处理这些传感数据。然而，由于受到当前移动平台的资源限制（内存、计算能力、电池容量等），基于 CNN 模型的应用在手机移动平台上并不多见。

目前，手机上基于 CNN 模型的应用绝大部分都是采用“客户端-服务器”模式，但是该模式不仅强依赖于网络性能（如网络稳定性等）而且会导致用户隐私泄露。因此，许多研究学者开始探索如何在移动端离线执行卷积神经网络的前向推断过程。针对这一研究课题，本文提出了一系列优化策略并设计与实现了一套可高效运行在 Android 平台的卷积神经网络推断时库。然后，本文利用该推断时库开发了一款生活日志型应用，借以探索从系统层进一步提高该类场景应用运行时能效的策略。论文的主要工作包括：

1. 利用预训练好的卷积神经网络模型权重在移动端重构网络，并使用 OpenCL 异构编程框架开发基于手机 GPU 加速的卷积神经网络推断时库。
2. 基于“剪枝-重训”方法对卷积神经网络模型进行压缩，并在 CNN 推断时库中引入稀疏矩阵向量乘（SpMV）使得运行时库支持经压缩处理的稀疏 CNN 模型。
3. 为了充分利用当前以及未来移动设备 SoC 所提供的异构计算能力，本文提出了一种使用移动平台异构设备处理器并行执行 CNN 推断的能效优化策略。该策略可根据目标平台所配备异构处理器间的能效差异自适应地寻找一个可高效并行执行 CNN 推断的设备处理器组合。
4. 针对基于 CNN 模型的生活日志型应用，本文详细分析了该类应用的运行时负载特征，并进一步提出了在系统层使用动态电压频率调节技术（DVFS）提高该类应用性能或能效的方法。

本文工作的研究意义主要包括如下三点：

1. 设计与实现了一套集成离线模型压缩、异构计算任务分配等功能的移动端 CNN 推断时库。

## 摘 要

---

2. 提出了基于异构设备处理器高能效并行执行 CNN 推断的策略，该策略可在运行时主动对目标平台上的异构处理器能效进行评估。
3. 探索了从系统层使用 DVFS 技术优化基于 CNN 模型移动端智能应用能效的策略。

**关键词：**卷积神经网络；移动平台；能效；异构计算；权值压缩；系统层优化

## ABSTRACT

In recent years, Convolutional Neural Networks (CNNs) have been widely used in various domains, such as computer vision and speech recognition because of their high accuracy and strong self-adaptiveness. On the other hand, mobile phones have become carry-ons to human beings, and generate a large number of sensor data every day. In order to make mobile phones serve people more intelligently, many engineering projects also try to use CNNs to process these sensor data on the mobile phone. However, due to resource limitations (memory, computing power, battery capacity, etc.) of current mobile platforms, CNN-based mobile applications have not become mainstream on mobile platforms.

Currently, most CNN-based mobile applications adopt the client-server computing paradigm. But this paradigm not only depends on network performance (such as network stability) but also leads to privacy leakage. Therefore, many researchers have begun to explore how to perform the inference process of convolutional neural networks directly on the mobile platform. For this research topic, this paper proposes a series of optimization strategies and designs a CNN inference library, which can run on the Android platform in an energy efficient way. Then, this paper uses the inference library to develop a life-logging application to explore ways to further improve the energy efficiency of such applications. In summary, our main contributions of this paper include:

1. Reconstructing the convolutional neural network on the mobile phone by pre-trained weights. Using the mobile GPU acceleration by the OpenCL framework to develop a CNN inference library.
2. Using the pruning-retraining loop to compress CNN models. Introducing Sparse Matrix-Vector Multiplication (SpMV) into the CNN inference library to enable the runtime library to support the compressed sparse CNN model.
3. To take full advantage of the heterogeneous computing environment provided by current and future mobile device SoCs, this paper proposes an optimization strategy which can use heterogeneous device processors to execute the CNN inference. Based on the energy efficiency difference among heterogeneous processors equipped on the target mobile platform, this strategy can adaptively find an energy efficient device processor combination to execute the CNN inference in parallel.

## ABSTRACT

---

4. This paper analyzes the runtime load of the CNN-based life-logging application in detail and further proposes the method of using Dynamic Voltage and Frequency Scaling (DVFS) to improve the CNN-based application performance or energy efficiency in system level.

The research significance of this paper is described as follows:

1. Designing and implementing a mobile CNN inference library that integrates functions such as model compression and heterogeneous computing task assignment.
2. Proposing a strategy for parallel execution of CNN inference based on heterogeneous device processors. This strategy can automatically evaluate the energy efficiency of heterogeneous processors on a target platform at runtime.
3. Exploring strategies of using DVFS to improve the CNN-based smart application energy efficiency in system level.

**Key Words:** CNNs; Mobile platform; Energy efficiency; Heterogeneous computing; Weights compression; System-level optimization

# 目 录

第 1 章 绪论 ······	1
1.1 引言 ······	1
1.2 深度学习模型于移动端能效优化的研究现状 ······	2
1.2.1 早期的探索与尝试 ······	2
1.2.2 基于模型压缩的优化 ······	3
1.2.3 基于异构计算的优化 ······	3
1.2.4 其他优化方法 ······	4
1.3 论文的主要研究工作 ······	4
1.4 论文的组织结构 ······	5
第 2 章 技术背景与实验平台 ······	7
2.1 卷积神经网络 ······	7
2.1.1 卷积层 ······	8
2.1.2 池化层 ······	9
2.1.3 全连接层 ······	10
2.1.4 激活层 ······	10
2.2 反向传播算法 ······	11
2.2.1 梯度下降 ······	11
2.2.2 反向传播 ······	12
2.3 OpenCL 异构编程框架 ······	12
2.3.1 可移植性 ······	13
2.3.2 标准向量处理 ······	13
2.3.3 并行编程 ······	13
2.3.4 OpenCL 基本编程流程 ······	14
2.4 能效度量标准与实验平台 ······	16
2.4.1 能效度量标准 ······	16
2.4.2 ODROID-XU3 ······	17
2.4.3 Open-Q <sup>TM</sup> 820 开发套件 ······	19
2.5 本章小结 ······	19
第 3 章 基于手机 GPU 加速和离线模型压缩的能效优化 ······	21
3.1 CNN 前向推断基本算子的分解与实现 ······	21
3.1.1 全连接层基本算子 ······	21

## 目 录

---

3.1.2 卷积层基本算子 .....	24
3.1.3 池化层基本算子 .....	26
3.1.4 激活层基本算子 .....	28
3.2 基于手机 GPU 加速的 CNN 前向推断 .....	29
3.2.1 预训练 CNN 模型权重参数解析 .....	29
3.2.2 LeNet-5 模型于移动端的重构 .....	30
3.2.3 AlexNet 模型于移动端的重构 .....	31
3.3 基于“剪枝-重训”的层压缩优化 .....	33
3.3.1 模型剪枝流程 .....	34
3.3.2 LeNet-5 模型剪枝 .....	35
3.3.3 AlexNet 模型剪枝 .....	36
3.3.4 移动端 SpMV 的实现 .....	38
3.3.5 与 CNNdroid 的对比 .....	39
3.4 本章小结 .....	40
<b>第 4 章 基于异构设备处理器并行执行推断的能效优化 .....</b>	<b>43</b>
4.1 移动端 SoC 的发展趋势 .....	43
4.2 基于异构计算的 CNN 并行推断研究动机 .....	44
4.3 基于异构计算的自适应计算任务分配策略 .....	45
4.3.1 高能效设备处理器组合的搜索 .....	45
4.3.2 计算任务的划分 .....	47
4.4 实验验证 .....	49
4.5 本章小结 .....	52
<b>第 5 章 基于应用场景的系统层能效优化 .....</b>	<b>53</b>
5.1 动态电压频率调节技术 .....	53
5.2 智能监控系统 Android 应用的负载分析 .....	53
5.3 基于应用场景的系统层能效优化策略 .....	57
5.3.1 动态时间规整 .....	57
5.3.2 基于应用场景的调频策略 .....	58
5.4 实验验证 .....	59
5.5 本章小结 .....	61
<b>第 6 章 总结与展望 .....</b>	<b>63</b>
6.1 本文工作总结 .....	63
6.2 未来工作展望 .....	64

## 目 录

---

参考文献 .....	65
致谢 .....	71
在读期间发表的学术论文与取得的研究成果 .....	73



# 图 目 录

1.1 研究工作的技术路线 .....	5
2.1 卷积神经网络结构示意图 <sup>[43]</sup> .....	7
2.2 图像数据和卷积核 .....	8
2.3 卷积过程示意图 .....	8
2.4 最大池化过程示意图 .....	10
2.5 全连接层示意图 .....	10
2.6 三种非线性激活函数 .....	11
2.7 核函数在不同 OpenCL 兼容设备上的分发过程 .....	14
2.8 ODROID-XU3 .....	17
2.9 功耗和能耗测量 APP 运行界面.....	18
2.10 Open-Q™ 820 开发套件 .....	19
3.1 矩阵乘法定义 .....	21
3.2 矩阵乘法不同实现版本的运行时间、能耗和平均功耗对比 .....	23
3.3 img2col 转换三通道输入特征图示例 .....	24
3.4 img2col 不同实现版本的运行时间、能耗和平均功耗对比 .....	26
3.5 最大池化不同实现版本的运行时间、能耗和平均功耗对比 .....	27
3.6 Relu 激活算子不同实现版本的运行时间、能耗和平均功耗对比 .....	28
3.7 CNN 模型权重参数解析流程.....	29
3.8 LeNet-5 模型结构示意图 <sup>[39]</sup> .....	30
3.9 LeNet-5 单张图片推断运行时间、能耗和平均功耗对比 .....	30

3.10 AlexNet 模型结构示意图 <sup>[2]</sup> .....	31
3.11 AlexNet 单张图片推断运行时间、能耗和平均功耗对比 .....	32
3.12 不同平台上 AlexNet 单张图片推断运行时间对比 .....	33
3.13 “剪枝-重训”流程概览 .....	34
3.14 LeNet-5 模型精度随全连接层权重连接保留率变化的曲线 .....	35
3.15 LeNet-5 模型全连接层剪枝前后权重分布对比 .....	36
3.16 AlexNet 模型精度随全连接层权重连接保留率变化的曲线 .....	36
3.17 AlexNet 模型全连接层剪枝前后权重分布对比 .....	37
3.18 剪枝前后 AlexNet 首次加载时间、能耗和平均功耗对比 .....	37
3.19 稠密矩阵和稀疏矩阵向量乘的运行时间、能耗和平均功耗对比 .....	38
3.20 AlexNet 剪枝前后单张图片推断运行时间、能耗和平均功耗对比 .....	39
3.21 本文 CNN 推断时库与 CNNdroid 的对比 .....	39
4.1 手机 GPU 和 CPU 执行 CNN 推断的运行时间、能耗和平均功耗 .....	44
4.2 基于异构计算的自适应计算任务分配策略工作流程 .....	45
4.3 计算任务划分示意图（其中每一条红线表示一个划分点） .....	48
4.4 每一层的执行时间 .....	50
4.5 每一层的运行时平均功耗 .....	50
4.6 每一层的运行时能耗 .....	51
4.7 不同异构计算组合的运行时间、能耗、平均功耗和能效对比 .....	51
5.1 智能监控系统 Android 应用 .....	54
5.2 智能监控系统 APP 的工作流程 .....	54
5.3 智能监控系统 APP 的负载和 GPU 频率变化 .....	55

## 图 目 录

---

5.4 CNN 推断的运行时间、能耗和平均功耗随 GPU 频率的变化 ······	56
5.5 两段时间序列间的规整 ······	57
5.6 规整路径示例 <sup>[60]</sup> ······	58
5.7 应用场景分类 DTW 阈值的确定 ······	60
5.8 基于应用场景的调频策略运行效果 ······	60



## 表 目 录

3.1 矩阵乘法不同实现版本的能效对比 .....	24
3.2 img2col 不同实现版本的能效对比 .....	26
3.3 最大池化不同实现版本的能效对比 .....	28
3.4 Relu 激活算子不同实现版本的能效对比 .....	29
3.5 LeNet-5 模型结构中的卷积层和全连接层 .....	30
3.6 LeNet-5 分别于手机 GPU 和 CPU 上运行时能效对比 .....	31
3.7 AlexNet 模型结构中的卷积层和全连接层 .....	32
3.8 AlexNet 分别于手机 GPU 和 CPU 上运行时能效对比 .....	32
3.9 AlexNet 模型的存储占用和内存占用需求 .....	40
4.1 分别于手机 GPU 和 CPU 上执行一次完整 CNN 推断的能效 .....	44
4.2 实验所用 CNN 模型的卷积层和全连接层结构参数 .....	49
5.1 智能监控系统 APP 在不同调频策略下的能效 .....	56
5.2 CNN 推断过程中不同调频策略下 CPU 的平均功耗 .....	61



## 算 法 索 引

3.1	CPU 版本矩阵乘法 . . . . .	22
3.2	GPU 版本矩阵乘法 (标量形式) . . . . .	22
3.3	GPU 版本矩阵乘法 (向量形式) . . . . .	23
3.4	img2col 核心操作伪代码 . . . . .	25
3.5	最大池化核心操作伪代码 . . . . .	27
3.6	激活层基本算子实现伪代码 . . . . .	28
3.7	稀疏矩阵向量乘实现伪代码 . . . . .	38
4.1	高能效设备处理器组合的搜索过程 . . . . .	46
4.2	设备处理器组合中每一处理器所分配任务比例的计算过程 . . . . .	48
5.1	基于应用场景的调频策略 . . . . .	59



# 第1章 绪论

## 1.1 引言

随着深度学习<sup>[1]</sup>领域不断取得突破性进展，基于深度学习模型的手机移动应用也如雨后春笋般发展起来。智能手机已经在逐渐改变人们的生活方式，几乎每部手机都集成着若干功能各异的传感器，这些传感器每天都在采集着大量与使用者相关的复杂数据，而深度学习模型无疑是处理这些数据的最佳工具。但是，由于当前嵌入式平台的资源限制（内存、计算能力、电池容量等），基于深度学习模型的应用并没有在手机移动应用市场中成为主流。

虽然深度学习模型对硬件资源的苛刻需求阻碍了它向手机移动平台的“进军”，但是其带来的好处仍然诱惑着人们在物联网设备和移动硬件上采用它。目前，手机上基于深度学习模型的应用（如，图像识别、语音识别）绝大部分都是基于云端服务的，即手机端采集所需的传感数据并通过网络将数据上传到云端服务器，云端服务器在获得数据后，运行深度学习模型推断，最后将所得的推断结果再次通过网络传回至手机端。然而，这种处理方式带来了许多负面影响，主要总结如下：1) 它可能会泄露用户的隐私数据，因为它需要将用户的一些敏感数据（如，图像、音频）发送到第三方服务端进行处理；2) 它的推断执行时间将会与波动的、不可预测的网络质量（如，网络延迟、吞吐量）紧密挂钩。更糟糕的是，当网络条件很差甚至不可联网时，这种推断操作就不能正常进行；3) 由于无线通信能量开销的存在，对于那些需长时期运行的应用（如，增强现实、认知助理等），使用云端执行推断任务也是不切实际的；4) 因为移动网络是按流量收费的，所以用户就不会使用那些需发送大量待处理数据（如视频流）至云端的应用，这便限制了深度学习模型在移动端的应用场景。

考虑到上述云端执行的负面影响，用户可能更希望那些基于深度学习模型的应用在手机本地就可以完成推断任务。另一方面，我们需要意识到高能效移动处理器的计算性能一直处于不间断地发展中。例如，与4年前的iPhone 5S相比，2017年苹果发布的iPhone 8在CPU单核处理性能上拥有着232%的增长，而在多核处理性能上更是提高了373%。许多研究学者认为，在不久的将来，即使没有远程计算的辅助，手机移动端也可以胜任许多基于深度学习模型的计算任务。通过手工改造并简化深度学习模型（如卷积神经网络<sup>[2]</sup>），在移动端本地设备上直接运行一些深度学习模型已经被证明是可行的，但是这样做不但需要大量的设计技巧，而且对于现存的大多数深度学习模型来说都是行不通的。更为重要的，正是因为深度学习模型的复杂性才使得推断准确度和鲁棒性取得了革命性

的飞跃，并且这才是智能移动应用所迫切需要的，所以手工简化模型的方式并非是我们的初衷。

与桌面、服务器端相比，手机移动端的不足不仅表现在较弱的处理能力上，还凸显在较小的内存容量上（如，华为 Mate 9 Pro 的内存容量仅为 4GB）。然而，因为深度学习模型具有结构异常复杂的特点，所以绝大部分模型都拥有着成百万上千万甚至上亿的参数（如，VGG16<sup>[3]</sup> 模型拥有 1.8 亿个参数）。这导致了许多深度模型并不能在手机移动端运行。而且，即使一些模型可以运行，其也会造成大量的内存占用和能耗开销。

综上所述，采用深度学习模型处理移动传感数据并在移动端进行离线推断是未来手机移动应用发展的一种趋势。为了将这一趋势转变为现实，当前研究人员必须要对移动端深度学习模型的本地推断过程进行各种能效优化。目前，研究工作需要解决的主要问题总结如下：

1. 如何利用当前移动设备处理器的特点及其未来的发展趋势，使得深度学习模型可以高能效地于移动端进行离线推断。
2. 受限于当前手机内存容量较小的不足，如何使得结构复杂的深度学习模型也可以正常地在移动端运行。
3. 在保证性能的条件下，如何根据应用场景的特点和系统层提供的信息，进一步降低上层基于深度学习模型应用的运行时功耗和能耗。

## 1.2 深度学习模型于移动端能效优化的研究现状

### 1.2.1. 早期的探索与尝试

Lane 等人<sup>[4]</sup> 设计了一个基于移动设备 CPU 和 DSP 的低功耗深度神经网络 (DNNs)<sup>[5]</sup> 原型推断系统。他们利用该推断系统研究了一些典型的移动感知任务（如行为感知），并将该推断系统与更加通用的传统辨识技术（如 SVM、GMM 和决策树等）相比较。他们的研究表明推断系统中所使用的深度神经网络并不会给现代的移动硬件带来过度的负载压力，而且即使是一个简单的 DNN 模型（如减少 71 倍的输入特征数），与传统的通用学习技巧相比，也可以改善推断精度。Lane 等人<sup>[6]</sup> 还研究了一些深度学习模型 (DNNs 和 CNNs) 在按比例缩小后，在资源受限的嵌入式设备上执行推断阶段的运行时行为和资源特征。Yanai 等人<sup>[7]</sup> 探究了适合在移动端实现的 CNN 结构，并提出了多可扩放的 Network-In-Network(NIN)<sup>[8]</sup>，即用户可以调整识别时间和识别精度的折中比。他们的研究发现 BLAS 库适合加速 IOS 系统上卷积层的计算，而 NEON SIMD<sup>[9]</sup> 更适合在

Android 系统上加速卷积层的计算。

### 1.2.2. 基于模型压缩的优化

Denton 等人<sup>[10]</sup> 通过使用线性压缩技巧去除了 CNN 模型中的冗余参数，有效地加速了大型已训练 CNN 模型的推断执行速度，而为此只需要付出很小的推断精度损失。类似地，Jaderberg 等人<sup>[11]</sup> 利用不同特征通道和卷积核间存在的冗余特征对卷积核进行了低秩近似分解。Lebedev 等人<sup>[12]</sup> 使用非线性最小平方法计算一个四维卷积核的低秩 CP-分解，即使用一些秩为 1 的张量之和表示该四维卷积核。Wu 等人<sup>[13]</sup> 提出了一个量化的卷积模型，可以在加速计算的同时降低模型的存储和内存开销。Wang 等人<sup>[14]</sup> 使用一个基于低秩的、分组稀疏向量分解的方法加速 CNN 模型的推断阶段。该方法的核心思想是将卷积核分解为一些小数量的多线性低秩张量之和，并用这些近似张量代替原始的卷积核执行标准回传过程以微调模型。

### 1.2.3. 基于异构计算的优化

为了验证使用移动端 GPU 执行卷积模型推断的加速效果，Lokhmotov 等人<sup>[15]</sup> 基于三星 Chrome-book 2 平台分析了 AlexNet<sup>[2]</sup> 卷积模型的前向推断在移动端的执行性能。根据实验结果，他们发现带有 OpenBLAS<sup>[16]</sup> 支持的 Caffe<sup>[17]</sup> 要比带有 ViennaCL<sup>[18]</sup> 支持的 Caffe 快大约 4 倍，比带有 cLIBLAS<sup>[19]</sup> 支持的 Caffe 快大约 10 倍。他们认为这可能是因为当前移动端的 GPU 性能较差，并且支持 OpenCL<sup>[20]</sup> 的现有并行库没有针对移动端进行优化。DeepX 是 Lane 等人<sup>[21]</sup> 为在移动平台上运行深度学习模型设计的一个软件加速器。DeepX 利用一个基于网络计算（远程处理器）和本地异构处理器的混合体（包含 CPUs, GPUs, LPUs 等）来降低资源的开销。DeepX 通过两个推断时资源控制算法来提升性能，即运行时层压缩和深度结构分解。然而，DeepX 主要缺点有两点：（1）运行时层压缩不仅没有减少运行时的内存开销，还加重了处理器的计算量；（2）运行时所使用的一些决策参数来源于离线计算好的值，不能适应运行环境的变化。Huynh 等人<sup>[22]</sup> 设计了一个基于手机 GPU 的深度神经网络框架 DeepSense。DeepSense 是基于 OpenCL 的，故而可以在 GPU 上执行 CNN 模型推断。然而，其计算密集型的操作（如，卷积运算）主要运行在 GPU 上，没有充分利用 CPU 的多核处理能力。与此同时，DeepSense 不支持模型压缩，这使得一些大而复杂的模型不能正常运行。文献<sup>[23]</sup> 呈现了一个基于 GPU 加速的库（CNNdroid），其主要使用 Android 官方提供的高性能计算框架 RenderScript<sup>[24]</sup> 来加速已训练 CNN 模型的推断过程。然而，其同样没有对一些较大的模型做压缩处理。

### 1.2.4. 其他优化方法

除了上述三类研究外,还有一些其他的相关研究。文献<sup>[25-27]</sup>通过按比例缩小深度模型的方法,将深度学习模型缩小后运行在手机或 DSP 上。使用低功耗处理器也被证明对于连续传感类型的应用特别有效,该类系统诸如 Speakersense<sup>[28]</sup>、Dsp.ear<sup>[29]</sup>等,它们在应用级进行优化以均衡主处理器和协处理器间的负载。Antoniou 等人<sup>[30]</sup>将深度卷积模型应用在智能监控系统中,分别于 PC 和移动设备上实现了一款可以自动检测、智能识别的在线监控系统。最后,开发深度学习领域的专用硬件是另外一个当前较为热门的研究方向,许多研究人员对此做出了贡献,如 Diannao<sup>[31]</sup>以及 FPGA 神经网络加速器<sup>[32-35]</sup>等。

可以看出深度学习模型于手机移动端的应用是近两年刚刚兴起并快速升温的研究方向。然而,大量研究学者的关注点都是放在如何提高深度学习模型于移动端的运行速度,而本课题的研究重点更多的是能效兼顾,并且会着重考虑能耗问题。

## 1.3 论文的主要研究工作

本文主要针对当前已被工业界广泛应用的深度卷积神经网络模型 (CNNs) 在 Android 平台上的推断执行进行能效优化研究,且主要工作包含以下方面:

1. 分析卷积神经网络进行前向推断时所需的基本算子,并通过 OpenCL 异构编程框架在手机 GPU 上实现计算密集型的算子。离线解析经 Caffe、YOLO<sup>[36]</sup> 和 Tensorflow<sup>[37]</sup> 等深度学习框架训练得到的卷积神经网络模型权重并将这些权重保存成统一的格式,以便在移动端重构不同框架训练出来的 CNN 模型。
2. 在保证推断精度损失极少的条件下,使用“剪枝-重训”算法对预训练好的 CNN 模型权重参数进行压缩,这样不仅可以降低网络模型的存储占用,还减少了模型内存加载时间和重构过程中的访存能耗。针对剪枝得到的稀疏矩阵,使用稀疏矩阵向量乘代替内积操作,进一步加快卷积神经网络的前向推断速度。
3. 为了充分利用当前以及未来移动设备 SoC 所提供的异构计算特征,本文提出了一种简单有效的方法,使得基于 CNN 模型的应用在运行时可以根据其所处运行环境中 CPU、GPU 等异构设备处理器的推断能效差异,自适应地寻找一个高能效的本地异构处理器组合并行执行 CNN 前向推断。
4. 针对需长时期运行的 CNN 模型应用 (Life-logging Apps),本文通过离线

分析其运行时负载特征进一步探索在系统层使用动态电压频率调节技术(DVFS)<sup>[38]</sup>提高该类应用性能或能效的策略。

通过上述第1、2、3点优化策略，本课题设计与实现了一套可高能效运行在Android平台的CNN推断时库，并最终将其运用在一个生活日志型应用(如智能监控系统)中以验证第4点优化策略的有效性。图1.1描述了本课题研究工作的技术路线。

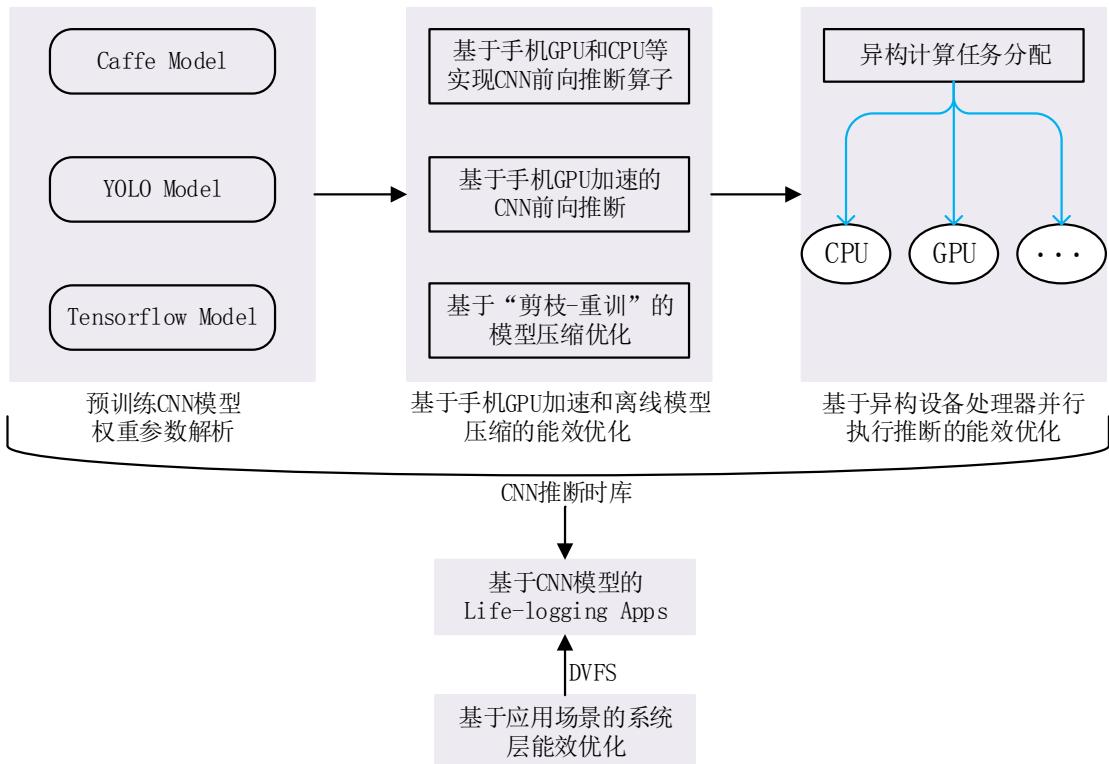


图1.1 研究工作的技术路线

## 1.4 论文的组织结构

本文的章节安排如下：

第1章主要讲述了深度学习模型在手机移动平台上的应用现状，并分析了阻碍基于深度学习模型移动应用发展的原因以及使用手机本地离线推断的必要性。接着，通过详细介绍深度学习模型于移动端进行能效优化的研究现状，引出本文的研究内容与目标。

第2章首先介绍了卷积神经网络的相关概念和组成结构，并对反向传播算法中所使用的梯度下降法和反向传播过程进行了阐述。然后，介绍了OpenCL异构编程框架的基本概念与开发优点，并给出了基于该框架的异构程序设计流程。最

后，描述了本文中所采用的能效度量标准并详细介绍了本研究中所使用的实验平台。

第3章首先对卷积神经网络前向推断过程中所使用的基本算子进行了分解，并详细描述了这些算子的作用与原理。然后，分别给出了这些基本算子在手机 GPU 和 CPU 上的实现方法。接着，描述了基于“剪枝-重训”的权重压缩方法，并使用该方法对卷积神经网络中占存储量主要部分的全连接层权重进行了压缩。对于压缩后的网络，进一步使用稀疏矩阵向量乘（SpMV）代替密集矩阵的内积运算，并给出了 SpMV 在移动端的实现。最后，基于 Caffe、YOLO 和 Tensorflow 等深度学习框架所训练的 CNN 模型权重，本文在移动端重构了 LeNet-5<sup>[39]</sup> 模型和 AlexNet 模型，并比较分析了基于手机 CPU、手机 GPU 和基于 SpMV 实现之间的能效差异。

第4章首先探讨了当前移动端 SoC 的发展趋势。接着，全面分析了在手机 GPU 和 CPU 上分别执行 CNN 前向推断时的能效，进而阐述了利用所有可获得的手机本地处理器去执行 CNN 前向推断并非是一种高能效的方式。本文提出一种简单有效的方法，其可以自适应地评估特定移动平台上所有可获得的本地处理器的能效，并进一步通过计算得到一个高能效的设备处理器组合用以并行执行 CNN 的前向推断过程<sup>[40]</sup>。最后，本文使用一种基于不同处理器计算性能的方法为所选组合中每一个设备处理器分配计算任务。

第5章首先介绍了动态电压频率调节技术（DVFS）的相关概念，并基于第2、3、4章节设计的 CNN 推断时库开发了一款生活日志型 Android 应用——智能监控系统。针对该应用，本文从系统层详细分析了其负载特征，并探索了利用 DVFS 技术进一步提高该类应用性能或能效的策略。

第6章对全文进行了总结，并对论文中尚未解决的问题提供研究线索，以期在未来的工作中加以解决并完善。

## 第2章 技术背景与实验平台

本文主要针对 Android 平台上卷积神经网络的前向推断过程进行能效优化的研究，故而本章首先会对研究工作的主要技术背景进行相关介绍。由于后面章节都会对不同实现方式的能效进行实验评估，故而本章最后一节详细描述了研究中所使用的能效度量标准和实验平台。2.1节介绍了卷积神经网络的基本概念，并对卷积神经网络中的几个基本层进行了详细描述；2.2节简要描述了反向传播算法中涉及的梯度下降<sup>[41]</sup> 和反向传播过程；2.3节主要介绍了 OpenCL 异构编程框架的基本概念和三个主要开发优点，并给出了使用 OpenCL 框架进行程序设计的基本开发流程。2.4节给出了本文评估能效的度量标准，并详细介绍了本文所使用的实验平台。

### 2.1 卷积神经网络

卷积神经网络（CNNs）是一种深度前向反馈人工神经网络<sup>[42]</sup>，其主要用来处理二维或更高维度的数据，如图像数据、音频数据等。近年来，在众多研究学者的努力下，卷积神经网络在计算机视觉领域已经取得了许多突破性进展，如高精度图像识别、目标检测等。卷积神经网络中主要采用了一种名为“卷积”的数学运算，它将深度神经网络模型部分层中的通用矩阵乘法替换为卷积操作。

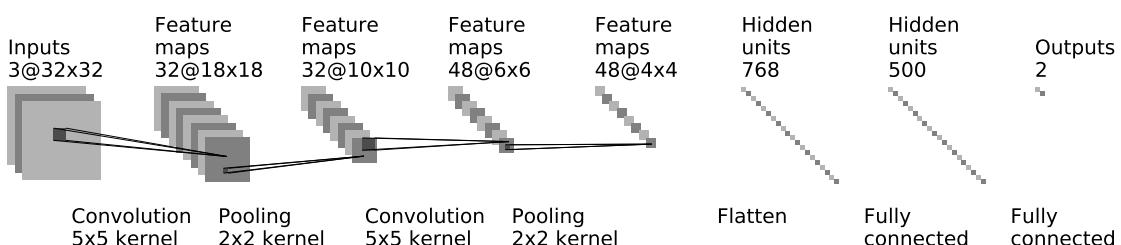


图 2.1 卷积神经网络结构示意图<sup>[43]</sup>

图2.1展示了一个卷积神经网络结构案例，其主要由三个基本层组成，即卷积层、池化层和全连接层。简单来说，每一个卷积层通过多个卷积核将前一层低级别特征转换成高级别特征。池化层用于捕捉一些不变性，如对输入图像的平移、旋转或缩放操作不影响卷积处理的输出结果。最后，全连接层通过聚合前面层提取出的高级别特征完成分类任务。除了这三种基本层外，输入数据在经过卷积层或池化层或全连接层之后还可能再附加一层激活层，其主要使得模型具备非线性，以解决非线性分类等问题。接下来的章节将详细描述这些层的含义与作用。

### 2.1.1. 卷积层

#### 1. 卷积操作

卷积层从字面上来看必定执行卷积操作，其主要通过将多个滤波器作用在输入图像上完成不同特征的提取。这些滤波器被称作卷积核，且被卷积的图像叫做特征图。下面以图2.2表示的图像数据和卷积核为例详细介绍卷积操作的整个过程。

图像数据					卷积核
1	0	0	1	1	1 0 1
0	1	1	1	0	0 1 0
1	1	1	0	1	1 0 1
0	1	0	1	1	
1	0	0	1	1	

图 2.2 图像数据和卷积核

如图2.3所示，卷积核沿着图像从左到右、从上到下按一定步长滑过，并返回卷积核内区域图像像素值与卷积核对应值的乘积之和。通过改变卷积核的值，卷积操作就可以提取到输入图像的不同种类特征。例如，一个中间值为8而其他所有值均为-1的 $3\times 3$ 卷积核可以提取图像的边缘特征，因为它强调了颜色的差异；而一个包含所有值均为0.1的 $3\times 3$ 卷积核会使得图像变得模糊不清，因为它降低了原始图像像素值。卷积神经网络的强大之处在于卷积层的卷积核权值不需要进行手工设计。卷积神经网络一经初始化后，其自身就可以通过一定的学习算法学习到合适的卷积核权重值。

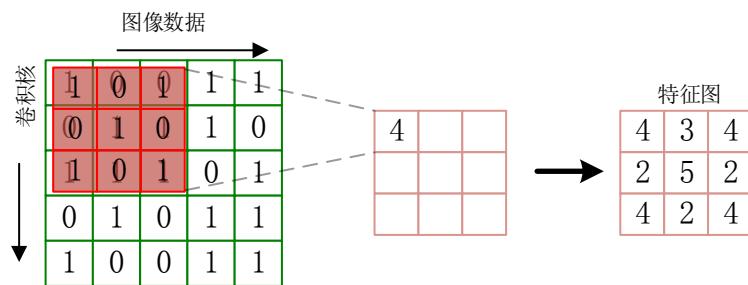


图 2.3 卷积过程示意图

#### 2. 卷积的动机

带有卷积层的神经网络之所以可得到很高的预测精度，是因为卷积操作利用了三个重要的思想：稀疏交互、参数共享和等变化表示。通过这三个思想，卷

积很好地模拟了人类视觉上的局部感受野。下面详述这三个重要思想：

- **稀疏交互:** 传统的神经网络层通过一个将每一输入与每一输出相连接的矩阵参数与输入数据进行矩阵乘法运算来得到该层的输出结果。这意味着每一个输出节点与每一个输入节点都是相交互的。然而，卷积神经网络输入与输出之间的连接是稀疏的。换言之，它通过使用许多远小于输入数据尺寸的卷积核将输入节点与输出节点进行连接。例如，在处理一张图像时，输入图像可能拥有成千上百万的像素，但是我们可以使用一个每次只覆盖几十个像素点的卷积核来检测出小的、有意义的特征（如图像边沿等）。这种稀疏连接方式大大降低了卷积模型的权重存储占用。
- **参数共享:** 这意味着对于模型中不同的功能单元可以使用相同的参数。传统神经网络权重矩阵中的每一个元素在计算输出节点时都只会被使用一次。而在卷积神经网络中，卷积核的每一个元素在输入的每一个位置上都会被使用。这依赖于一个合理的假设：如果一块特征的计算在某个空间位置上是有用的，那么它在其他位置上的计算也应该是有用的。
- **等变化表示:** 卷积操作的参数共享进一步为卷积网络附加了一个属性，即对不同变换的等变化表示。如果对一个函数的输入施加了某种变换，其输出也会响应相同的改变，那么该函数就是等变化的。特别地，一个函数  $f(x)$  是等变化的当且仅当  $f(g(x)) = g(f(x))$ 。对于卷积来说，如果  $g$  是对输入进行的任意一种平移变换，那么卷积函数对  $g$  来说就是等变化的。然而，卷积对于一些其他的变换（如缩放、旋转等）并不是自然等变化的，因此还需要其他的数学机制去处理这些类型的变换。

### 2.1.2. 池化层

与卷积层相比，池化层就相对比较简单。池化层并不直接参与训练学习，其仅仅是对卷积层传来的图像进行非线性下采样操作。最大池化 (max pooling) 是目前所有非线性池化操作中最为常用的。它将输入图片划分成一系列非重叠的矩形区域，对于每一个子区域将其最大值作为输出。池化层可以有效地降低数据表示的空间大小、减少网络模型的参数量和计算量，因此它也可以控制过拟合。在一个 CNN 模型结构中，池化层通常被周期性的插入到紧挨卷积层之后。另外，池化操作也提供了另外一种形式的平移不变性。

池化层独立地作用在输入数据的每一个通道上以调整它的空间尺寸。如图2.4所示，使用  $2 \times 2$  的滤波器在输入数据的每一个通道上沿宽和高两个方向进行步长为 2 的下采样操作是池化层中一个最为常用的形式，其可以丢弃 75%

的激活结果。在该示例中，最大化操作一次作用在4个数字上，并将4个数字中的最大值作为其输出结果。经池化操作后，输入数据的深度维度是保持不变的。

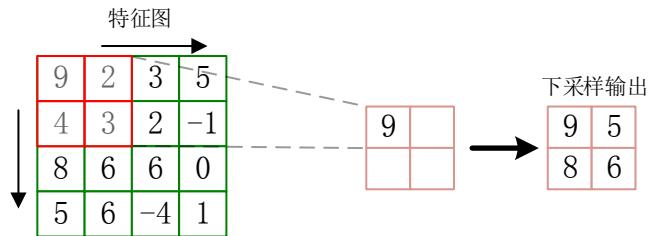


图 2.4 最大池化过程示意图

除了最大池化外，池化单元也可以使用其他函数，如平均池化或 L2-正则池化等。平均池化在之前的研究中使用较多，但在近几年越来越多地被最大池化所代替，最大池化已被实践验证下采样效果更佳。

### 2.1.3. 全连接层

在若干个卷积层和池化层之后，更高级别的逻辑分类是由全连接层完成的。如图2.5所示，全连接层的神经元与前一层的所有激活值进行全连接，其输出值是由输入神经元与权值矩阵进行内积操作并加上偏置矩阵（如果存在的话）而得到的。

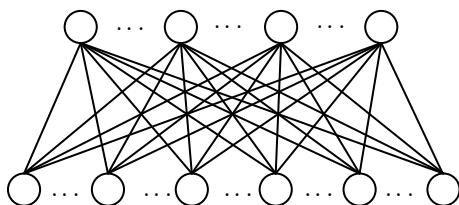


图 2.5 全连接层示意图

### 2.1.4. 激活层

激活层主要使用非线性激活函数来增强决策函数和整个网络模型的非线性属性，同时其不会影响卷积层的感受野。Sigmod 函数 ( $f(x) = (1 + e^{-x})^{-1}$ )、Relu 函数 ( $f(x) = \max(0, x)$ ) 以及双曲正切函数 ( $f(x) = \tanh(x)$ ) 是激活层最为常用的非线性函数，它们的函数图像如图2.6所示。其中，Relu 函数是近年来被研究人员所青睐的非线性激活函数，因为它不仅更接近生物神经激活函数还可以在一定程度上抑制梯度消失问题的出现，而且相对于其他激活函数，使用 Relu 函数的模型训练速度会更快。

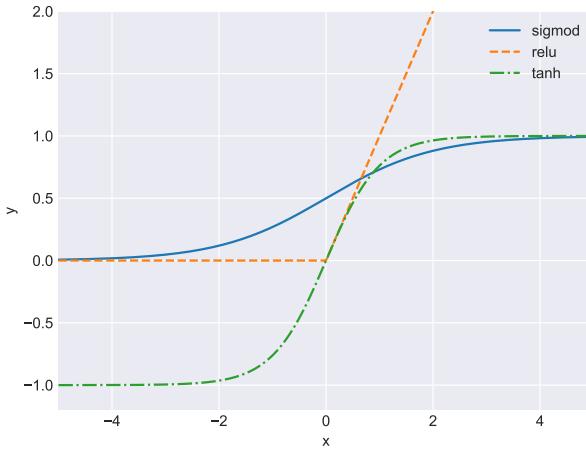


图 2.6 三种非线性激活函数

## 2.2 反向传播算法

在设计好 CNN 模型的结构之后，利用训练样本和损失函数就可以通过反向传播算法求出 CNN 模型中的参数矩阵值。反向传播算法是“误差反向传播”的简称，它是目前训练人工神经网络的最重要方法。反向传播算法主要使用梯度下降法求取模型最优权重，即通过预定义的损失函数计算网络模型中所有权重的梯度，并使用这些梯度来更新权值以最小化损失函数。下面首先介绍梯度下降法，然后详细描述反向传播的计算过程。

### 2.2.1. 梯度下降

在机器学习任务中，需要最小化损失函数  $L(\theta)$ ，其中  $\theta$  是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值  $\theta^0$ ，不断迭代，更新  $\theta$  的值，进行损失函数的极小化。梯度下降的迭代公式为  $\theta^t = \theta^{t-1} - \alpha L'(\theta^{t-1})$ ，其详细推导过程描述如下：

- 参数更新的迭代公式： $\theta^t = \theta^{t-1} + \Delta\theta$

- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行一阶泰勒展开：

$$L(\theta^t) = L(\theta^{t-1} + \Delta\theta) \approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta$$

- 要使得  $L(\theta^t) < L(\theta^{t-1})$ ，可取： $\Delta\theta = -\alpha L'(\theta^{t-1})$ ，则： $\theta^t = \theta^{t-1} - \alpha L'(\theta^{t-1})$ 。  
此处  $\alpha$  是步长，一般取一个较小的数值。

### 2.2.2. 反向传播

反向传播算法主要使用链式求导法则和梯度下降法将损失函数对于每一层的误差沿层回传以更新每层参数的权重值，其主要包括两个阶段：**传播**和**权重更新**。

每一次传播涉及如下步骤：

- 通过网络进行前向传播以生成输出值。
- 通过损失函数计算损失（即推断误差）。
- 沿网络回传所有输出神经元和隐层神经元的增量，即目标值和实际输出值的差值。

每一次权值更新必须遵循如下步骤：

- 将权重的输出增量和输入激活值相乘以得到权重的梯度。
- 一定比重 ( $\alpha$ ) 的权重梯度要从当前权重值中减掉。

比重  $\alpha$  影响着学习速度和学习质量，它也被称为学习率。学习率越高，神经网络的训练速度越快，但是较低的学习率可以使得模型训练更加精确。权重梯度的正负号表示误差是正误差还是负误差，其决定了权重变化方向（增大或减小）。从梯度下降法的推导过程可知，权重必须沿着逆梯度方向增长。上述两个阶段在整个网络模型的学习过程中将不断在不同的批次数据上重复进行直到网络达到预设的性能。

## 2.3 OpenCL 异构编程框架

OpenCL(Open Computing Language)<sup>[20]</sup> 是一个编写跨异构平台可执行程序的编程框架，这些异构平台包括中央处理器 (CPUs)、图形显示器 (GPUs)、数字信号处理器 (DSPs)、现场可编程门阵列 (FPGAs) 以及一些其他处理器或硬件加速器。OpenCL 使用基于 C99 的编程语言为异构设备编写程序，并使用 C/C++ 语言编写主机程序以控制设备程序在计算设备上的运行。OpenCL 为基于任务并行或数据并行的并行计算提供了一个开放的标准接口，并由非盈利技术团队 Khronos Group 维护。可移植性、标准向量处理和并行编程是吸引开发者选择使用 OpenCL 编程的三个主要优势。

### 2.3.1. 可移植性

Java 因其“一次编译，处处运行”的理念而广受青睐，而 OpenCL 可能比 Java 更具可移植性，可以说 OpenCL 是“一次编写，万物运行”。每一个厂商在提供 OpenCL 兼容设备的同时也会提供一些工具链去编译 OpenCL 代码。因此，相同的 OpenCL 代码只需通过使用各个厂商提供的工具链进行编译便可在不同的硬件设备上运行。这对于常规的高性能计算来说是一个极大的便利，因为在没有 OpenCL 之前，高性能计算程序设计人员不得不针对不同厂商的硬件设备学习该厂商指定的编程语言。

### 2.3.2. 标准向量处理

标准向量处理是 OpenCL 的一个巨大优点。与物理或数学上的向量含义不同，这里的向量指的是一种数据结构，其包含多个相同数据类型的元素。在进行向量计算时，对向量中每个元素的操作都是在同一个时钟周期内完成的，可完成这种类型操作的处理器包括超标量处理器或向量处理器等。向量指令通常是由厂商指定且不统一，例如：Intel 处理器使用 SSE 扩展，而英伟达设备使用 PTX 指令。不同的指令集之间的巨大差异性使得开发人员不能使用标准 C/C++ 进行向量编程，而且标准 C/C++ 语言亦没有原生提供向量数据类型。

然而，程序设计人员可以利用 OpenCL 一次编写向量例程，并将它们运行在不同的兼容设备上。这些 OpenCL 程序在不同的平台上会产生不同的编译结果：英伟达的 OpenCL 编译器将会生成 PTX 指令，而 IBM 的编译器会将 OpenCL 程序编译成 AltiVec 指令。由此可见，使用 OpenCL 编写可在不同平台上运行的高性能计算程序是一件省时又省力之事。

### 2.3.3. 并行编程

在 OpenCL 中，开发人员可使用被称为核函数 (*kernels*) 的数据结构将计算任务分配到不同的处理单元上进行并行执行。一个核函数是一段特殊的函数代码，其可以被一个或多个 OpenCL 兼容设备执行。核函数被主机应用 (*host applications*) 发送到一个或多个选定的设备上执行。一个主机应用是使用常规 C/C++ 语言编写的程序，其可以运行在用户使用的主机系统上。主机应用不仅可以将核函数分发到从设备（如 GPUs）上，还可以让核函数直接在主机应用正在使用的 CPU 上运行。

主机应用使用一个被称为上下文 (*context*) 的容器管理着与其相连的从设备。图2.7显示了主机与核函数以及设备之间的交互。为了创建一个核函数，主机应用从一个被称为核函数容器的程序 (*program*) 中选择一个函数。然后，主机应用设

置好该核函数的参数数据并将其分派到一个被称为命令队列的结构中 (*command queue*)。通过命令队列，主机可以告诉设备需要做的事情。当核函数入列后，设备将执行对应的函数逻辑。

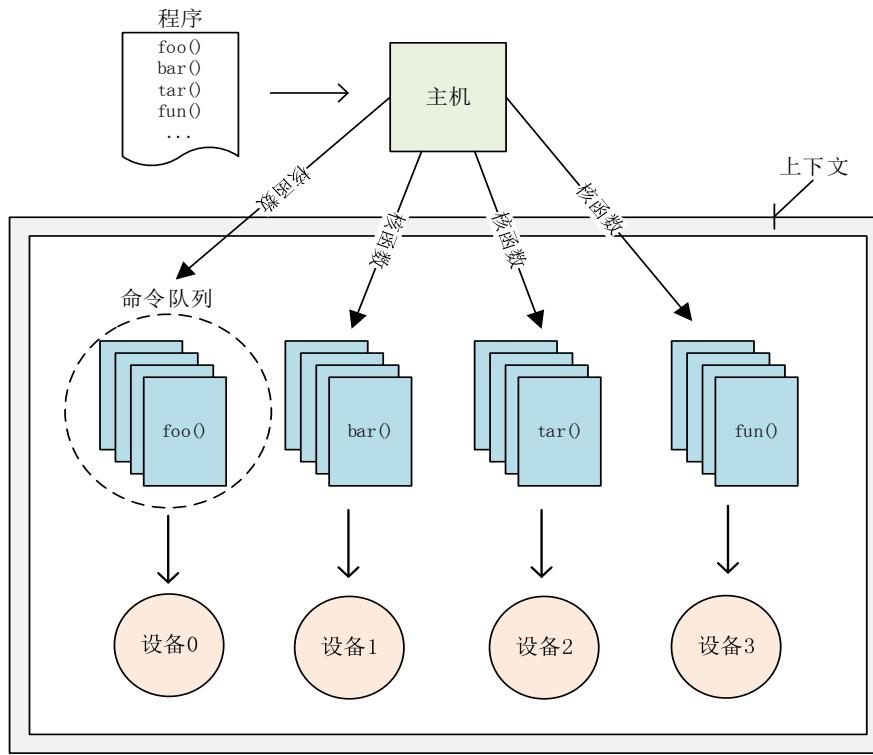


图 2.7 核函数在不同 OpenCL 兼容设备上的分发过程

OpenCL 既支持数据并行操作，也提供完整的任务并行机制。换言之，OpenCL 可在不同的处理单元上执行不同类型的计算任务。图2.7描述了 OpenCL 如何在不同的设备上完成任务并行逻辑。大多数的 OpenCL 设备拥有着不止一个的处理单元，这意味着在每个设备的内部还可以进行进一步的并行计算。

#### 2.3.4. OpenCL 基本编程流程

可移植性、向量处理以及并行编程使得 OpenCL 比常规的 C/C++ 更加强大，但是使用 OpenCL 编写程序也比较复杂。下面给出 OpenCL 编程的基本流程：

1. 访问 OpenCL 平台 (*platform*) 并选择所需平台。平台是由硬件厂商提供的 OpenCL 框架的具体实现，不同的异构硬件生产商（如 Intel、AMD、Nvidia 等）可以在一个系统上分别定义自己的 OpenCL 框架。故而在开发 OpenCL 程序前，程序设计者需要查询系统中可用的 OpenCL 框架实现。这就需要用到 *platform*，其对应的数据结构是 `cl_platform_id`。OpenCL 框架定义的 `clGetPlatformIDs()` API 可以用来访问并获取可用的平台，通过 API

`clGetPlatformInfo()` 可以查看与指定平台相关的信息。

2. 访问指定平台上的设备并选择所需设备。通过厂商提供的平台，程序设计者可以访问每一个与之相连的设备。在 OpenCL 应用中，设备收到的任务和数据来源于主机。在代码中，设备由 `cl_device_id` 数据结构表示。查询并获得设备以及查看与特定设备相关的信息可分别通过 OpenCL 框架定义的 `clGetDeviceIDs()` 和 `clGetDeviceInfo()` 这两个 API 实现。
3. 创建上下文并通过上下文管理设备。如前所述，OpenCL 的上下文主要用来标识并管理一个设备集合。目前，一个上下文只能包含来自同一平台的设备，换言之，AMD 和英伟达提供的设备不能放在同一个平台上。但是，主机应用可以通过创建多个上下文来管理不同厂商提供的设备，并且同一个平台上也可以创建多个上下文。创建上下文主要由 `clCreateContext()` 和 `clCreateContextFromType()` 两个 API 完成。
4. 通过上下文创建命令队列。每个命令队列与特定的某个设备一一对应，主机应用可以通过命令队列对设备进行操作，如执行核函数、设置核函数参数以及读写设备内存等。默认地，命令队列按接收顺序执行命令，但是在创建命令时也可以改变这种默认行为。`clCreateCommandQueue()` API 可完成命令队列的创建工作。
5. 使用程序 (*program*) 存储设备代码。一个程序包含了许多核函数的实现代码，由数据结构 `cl_program` 表示。程序设计人员可以将一个文本字符串通过 `clCreateProgramWithSource()` API 转换为程序这种数据结构，并可使用 API `clBuildProgram()` 对创建的程序进行编译。因为 OpenCL 使用厂商提供的编译器对设备代码进行编译，所以编译信息需要使用 API `clGetProgramBuildInfo()` 得到。
6. 从程序中提取核函数 (*kernel*)。对程序进行编译、链接后，可以使用核函数这种数据结构 (`cl_kernel`) 将每个函数功能进行封装。核函数可以被分发到一个命令队列，进而可被与该命令队列所关联的设备执行。通过 API `clCreateKernel()` 可对程序中的每一个函数进行核函数封装，而 API `clGetKernelInfo()` 可以用来查询指定核函数的相关信息。
7. 设置核函数参数并执行核函数。程序设计人员可以使用 `clEnqueueTask()` 和 `clEnqueueNDRangeKernel()` 这两个 API 将核函数入列执行，而核函数的参数可以使用 API `clSetKernelArg()` 进行设置。核函数入列后并不意味着马上执行，其必须要在相应的命令队列中排队等待。

8. 读取执行结果并释放 OpenCL 资源。OpenCL 提供了许多以 `clEnqueue` 开头的 API，每一个这种函数都是通过命令队列向设备发送命令。其中，可用来读取核函数执行结果的 API 包括 `clEnqueueReadBuffer()` 和 `clEnqueueMapBuffer()`。另外，凡是通过以 `clCreate` 开头的 API 创建的数据结构必须通过以 `clRelease` 开头的对应 API 进行资源的释放。

## 2.4 能效度量标准与实验平台

本文使用 Energy Delay Product(EDP)<sup>[44]</sup> 评估 CNN 前向推断过程中所涉及操作的执行能效，2.4.1节对 *EDP* 的概念以及选择原因进行了阐述。由于本文其他章节的实验部分都是基于 ODROID-XU3<sup>[45]</sup> 和 Open-Q™ 820<sup>[46]</sup> 两个平台进行的，故而2.4.2节和2.4.3节分别对两个平台进行了详细介绍。

### 2.4.1. 能效度量标准

Energy Delay Product(EDP) 是由 Horowitz 提出并被用于评估数字电路设计领域中电路级功耗节约技术提升的能效，之后又被 Brooks 推广使用<sup>[47]</sup>。现在，*EDP* 也经常被用来评估给定某一程序的执行能效。本文主要使用 *EDP* 评估一个计算过程（如 CNN 前向推断）使用不同优化方法所达到的能效值，它反映了该计算过程的计算延迟时间和电池电量消耗之间的折中。因此，本文中 *EDP* 被定义为一个计算过程的所需执行时间与所消耗电量之积，用公式形式化如下：

$$EDP = E \times T \quad (2.1)$$

本文中，*E* 代表某一计算过程的平均能耗，*T* 代表某一计算过程的平均执行时间。很明显，*EDP* 是一个值越小越好的度量标准。

除 *EDP* 可作为能效度量标准外，性能与功耗之比（Performance per Watt, PPW）、性能与能耗之比（Performance per Joule, PPJ）也经常被研究学者用于评估一个计算过程的执行能效。另一方面，在 CNN 前向推断过程中，每秒处理图像数据的帧数（Frame per Second, FPS<sup>[48]</sup>）经常作为 CNN 推断执行性能的度量标准。因此，在 CNN 前向推断过程中，性能与功耗之比可由公式2.2和公式2.3推理得到，而性能与能耗之比可由公式2.2和公式2.4推理得到。三个公式中，*T* 代表平均执行时间，*P* 代表平均功耗，*E* 代表平均能耗。

$$FPS = \frac{1}{T} \quad (2.2)$$

$$PPW = \frac{FPS}{P} = \frac{\frac{1}{T}}{P} = \frac{1}{P \times T} = \frac{1}{E} \quad (2.3)$$

$$PPJ = \frac{FPS}{E} = \frac{\frac{1}{T}}{E} = \frac{1}{E \times T} = \frac{1}{EDP} \quad (2.4)$$

根据公式2.3可知，在 CNN 前向推断过程中，性能与功耗之比即为能耗的倒数。因此，性能与功耗之比仅仅强调了 CNN 前向推断的能耗因素，不能够全面反映推断过程的性能和能耗折中。根据公式2.4可知，在 CNN 前向推断过程中，性能与能耗之比即为  $EDP$  的倒数，其为一个值越大越好的度量标准。无论是选择性能与能耗之比还是  $EDP$ ，度量能效的结果都是一致的。故而，本文最终选用  $EDP$  作为 CNN 前向推断过程中所涉及计算执行能效的度量标准。

#### 2.4.2. ODROID-XU3

图2.8所示即为 Hardkernel 公司生产的 ODROID-XU3 开发平台，其采用三星 Exynos5422 作为 SoC（内部集成 2GB 内存和 ARM Mali-T628 MP6 GPU）。实验中，ODROID-XU3 运行的操作系统为 Android 5.1.1 版本（内核版本号为 3.10.9）。Exynos5422 的处理器使用了 ARM 的 big.LITTLE<sup>[49]</sup> 技术，并由四个 ARM Cortex-A15 大核（频率最高可达 2.0GHz）和四个 ARM Cortex-A7 小核（频率最高可达 1.4GHz）组成。由于使用了 big.LITTLE™ HMP 解决方案，Exynos5422 最多可以使用 8 个核去处理计算密集型的任务。ARM Mali-T628 MP6 GPU 拥有两个 GPU 簇，即 GPU 簇 0 包含 4 个 ARM Mali-T628 核而 GPU 簇 1 包含 2 个 ARM Mali-T628 核<sup>[50]</sup>。故而，对于 ODROID-XU3 来说，所有可获得的本地设备处理器包括一个 CPU 设备和两个 GPU 设备。

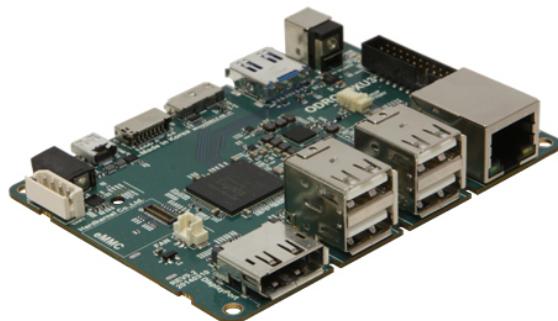


图 2.8 ODROID-XU3

为了方便研究人员测量功耗和能耗，ODROID-XU3 开发平台集成了 4 路电流/电压传感器，它们可以独立地测量大核 A15、小核 A7、GPU 和内存 (DRAMs) 的实时功耗。因为 Hardkernel 公司官网并未提供 Android 平台使用传感器测量不

同部件功耗或能耗的 APP，而本文的实验部分却都是基于 Android 平台的，所以本文基于 Android NDK 开发了两个 APP，即 Power Monitor 和 Energy Update Service。它们可以分别用于测量给定程序某段运行过程的功耗（Power Monitor）和能耗（Energy Update Service）。

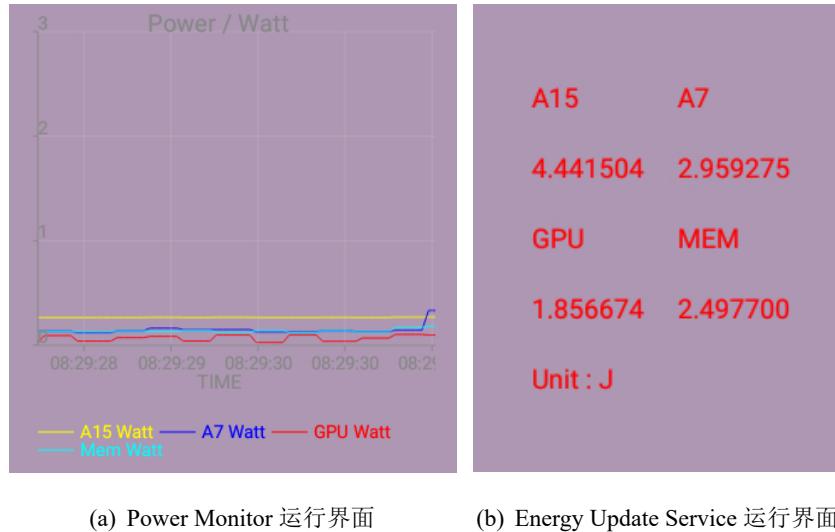


图 2.9 功耗和能耗测量 APP 运行界面

- **Power Monitor APP 工作原理：**后台服务每隔 50 毫秒读取四路传感器的更新值，并将这些更新值追加到 Power.csv 文件中；前端浮动窗口每隔 50 毫秒进行一次界面刷新，并利用后台服务不断传回的四个传感器值绘制大核 A15、小核 A7、GPU 和内存的实时功耗曲线。另外，实验中也可以选择关闭前端的浮动窗口以排除其能耗开销的干扰。浮动窗口关闭后，后台服务会继续运行以不断更新 Power.csv 的文件内容。之后，可以通过离线分析 Power.csv 的内容得到给定 APP 的功耗特征。Power Monitor 的运行界面如图2.9(a)所示。
- **Energy Update Service APP 工作原理：**后台服务每隔 50 毫秒读取四路传感器的更新值，并利用这些更新值求取自 Energy Update Service 应用启动以来大核 A15、小核 A7、GPU 和内存的累积能耗。实验中系统内核经过修改加入了两个系统调用功能：380 号系统调用用于将 4 个最新的累积能耗值（分别对应大核 A15、小核 A7、GPU 和内存的累积能耗）更新到指定的内核空间，而 381 号系统调用用于随时读取该指定内核空间的四个累积能耗值。380 号系统调用主要被后台服务用于将 4 个最新求取的累积能耗值更新至指定的内核空间（更新周期为 50 毫秒）。而其他 Android 应用只需在待测量功耗的代码段前后打上两个桩，通过两次执行 381 号系统调用即

可得到运行前后的两个累积能耗值，两值之差即为运行该段代码所需的能耗。Energy Update Service 应用的前端浮动窗口用于实时显示 4 个当前累积能耗值，其也可以被关闭且不会影响后台服务的正常运行。Energy Update Service 的运行界面如图2.9(b)所示。

### 2.4.3. Open-Q™ 820 开发套件

如图2.10所示，Open-Q™ 820 是 Intrinsyc 公司推出的一款基于骁龙 820 芯片的开发套件。骁龙 820 芯片不仅使用了四个主频可达 2.2GHz 的 64 位 ARMv8 核（Kryo），还配备了高通 Adreno 530 GPU 和 Hexagon 680 DSP。其中，Adreno 530 GPU 虽然只包含了 4 个核心，但是性能要比 ARM Mali-T628 MP6 GPU 强大许多。另外，Open-Q™ 820 还配有 3GB 的 LPDDR4 板载内存。实验中，Open-Q™ 820 运行的 Android 系统版本为 7.0(代号为 Android N)，内核版本号为 3.18.31。



图 2.10 Open-Q™ 820 开发套件

本文的实验部分主要基于 ODROID-XU3 开发平台展开，这是因为使用本文所开发的 Power Monitor 和 Energy Update Service 两款 Android 应用可以很容易地在 ODROID-XU3 平台上获取一段程序执行过程所产生的能耗和功耗。Open-Q™ 820 开发套件主要用于验证本文所开发 CNN 推断时库的可移植性和兼容性，并与 ODROID-XU3 开发平台的 CNN 推断执行性能做对比。

## 2.5 本章小结

本章首先介绍了卷积神经网络的概念和其中一些基本层的含义与作用，这为下一章分解卷积神经网络前向推断过程中的基本算子做足了准备。因为本研究工作中使用“剪枝-重训”方法对卷积神经网络的权重进行压缩，故而本章对

深度学习模型训练过程中所使用的反向传播算法进行了详细描述。另外，2.3节对本研究中所使用的异构编程框架 OpenCL 的基本概念及其具备的三个主要优点进行了介绍，并给出了开发 OpenCL 异构程序的基本编程流程。最后，本章对研究中所使用的能效评估度量标准以及实验平台进行了详细说明。

## 第3章 基于手机GPU加速和离线模型压缩的能效优化

本章详细分析了卷积神经网络前向推断过程中所使用的基本算子，并对每一个基本算子分别给出了基于手机GPU和CPU的实现方式。基于这些基本算子，本文实现了构成CNN推断时库所需的各种网络层。然后，利用实现的CNN推断时库和已训练好的CNN模型权重在手机移动平台上重构了LeNet-5和AlexNet，并分析比较了CPU版本和GPU版本两种实现方式的能效。最后，本章描述了基于“剪枝-重训”的权重压缩方法，并使用该方法对卷积神经网络中占存储量主要部分的全连接层权重进行了压缩。对于压缩后的CNN模型，本章进一步使用稀疏矩阵向量乘（SpMV）代替密集矩阵的内积运算。最后，与CNNdroid<sup>[23]</sup>的对比实验进一步分析了本文所实现CNN推断时库的性能。

### 3.1 CNN前向推断基本算子的分解与实现

卷积神经网络的前向推断过程主要涉及卷积层、池化层、全连接层和激活层等的前向传播，故而对CNN前向推断基本算子的分解即转换为对这些层实现中所需基本算子的剖析。下面针对每层的基本算子，详细介绍其作用机理并分析比较基于手机GPU和CPU实现版本的能效。

#### 3.1.1 全连接层基本算子

$$\begin{bmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ - & - & - & - \\ c_{n1} & c_{n2} & \dots & c_{np} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ - & - & - & - \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ - & - & - & - \\ b_{m1} & b_{m2} & \dots & b_{mp} \end{bmatrix}$$

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + \dots + a_{1m}b_{m1} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + \dots + a_{1m}b_{m2} \\ c_{1p} &= a_{11}b_{1p} + a_{12}b_{2p} + \dots + a_{1m}b_{mp} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} + \dots + a_{2m}b_{m1} \\ c_{np} &= a_{n1}b_{1p} + a_{n2}b_{2p} + \dots + a_{nm}b_{mp} \end{aligned}$$

图3.1 矩阵乘法定义

矩阵乘法操作是全连接层用到的主要基本算子，而且其在本文的卷积层实

现中也被涉及，所以在此先介绍矩阵乘法于手机 GPU 和 CPU 上的实现。图3.1给出了矩阵乘法的定义。根据定义可得 CPU 版本的矩阵乘法实现伪代码如下：

```

Data: mat_left, row_left, col_left, mat_right, row_right, col_right, bias, result
1 for i in 0 ... row_left-1 do
2   | for j in 0 ... col_right-1 do
3     |   res = 0;
4     |   for k in 0 ... col_left-1 do
5       |     res += mat_left[i * col_left + k] * mat_right[k * col_right + j];
6     |   end
7     |   res += bias[i];
8     |   result[i * col_right + j] = res;
9   | end
10 end

```

算法 3.1: CPU 版本矩阵乘法

基于 OpenCL 异构编程框架，下面给出 GPU 版本的矩阵乘法实现。将 CPU 版本中第一重和第二重循环的次数分别设置为核函数的全局工作空间大小，可将 CPU 版本代码直接改成 GPU 版本的标量形式，伪代码表示如下：

```

Data: mat_left, mat_right, col_left, bias, result
1 初始化 res 的值为 0;
2 i = get_global_id(0);
3 j = get_global_id(1);
4 col_right = get_global_size(1);
5 for k in 0 ... col_left-1 do
6   |   res += mat_left[i * col_left + k] * mat_right[k * col_right + j];
7 end
8 res += bias[i];
9 result[i * col_right + j] = res;

```

算法 3.2: GPU 版本矩阵乘法 (标量形式)

上述伪代码中 `get_global_id(dim)` 用于获得第 `dim` 维上的工作项全局 ID，`get_global_size(dim)` 用于获得第 `dim` 维上的全局工作项个数。GPU 版本矩阵乘法的标量形式没有充分利用 OpenCL 的向量处理能力，这会造成一定的 GPU 并行计算性能损失。算法3.3给出了 GPU 版本矩阵乘法向量形式的核心伪代码实现。在向量形式实现版本中使用了 `float4` 向量数据类型，并通过 OpenCL `dot()` 函数完成对两个 `float4` 类型数据的一次内积操作。此处需要注意一点：因为对右矩阵不能使用 `vload4()` 函数连续读取数据，所以需要先读取所需的 4 个数据再将它们转换为一个 `float4` 向量类型，这样才能使得两个包含四元素的向量内积操作可以在一个时钟周期内完成。

**Data:** mat\_left, mat\_right, col\_left, bias, result

```

1 初始话 res 的值为 0;
2 remain 表示核函数剩余未处理的数据量;
3 while remain >= 4 do
4   | 从 mat_left 中使用 vload 函数取出四个数据组成 float4 向量类型 tmp1;
5   | 从 mat_right 中取出与 tmp1 对应的四个数据组成 float4 向量类型 tmp2;
6   | 使用 dot 函数计算 res 的当前值: res += dot(tmp1, tmp2);
7   | 设置 mat_left 和 mat_right 的下标偏移量;
8   | 更新剩余未处理的数据量: remain -= 4;
9 end
10 while remain > 0 do
11   | 从 mat_left 中取出一个 float 数据 tmp1;
12   | 从 mat_right 中取出一个 float 数据 tmp2;
13   | 更新 res 的当前值: res += tmp1 * tmp2;
14   | 设置 mat_left 和 mat_right 的下标偏移量;
15   | 更新剩余未处理的数据量: remain -= 1;
16 end
17 将 res 加上对应偏置值并将结果赋值给 result 对应项;

```

### 算法 3.3: GPU 版本矩阵乘法(向量形式)

基于 ODROID-XU3 实验平台, 本文分析比较了上述三个矩阵乘法实现版本的能效。图3.2显示了  $512 \times 1024$  矩阵与  $1024 \times 512$  矩阵相乘的运行时间、能耗和平均功耗结果。



图 3.2 矩阵乘法不同实现版本的运行时间、能耗和平均功耗对比

由图3.2可知, 使用手机GPU实现矩阵乘法可以明显提升运算性能并降低运行时能耗和功耗。对于  $512 \times 1024$  矩阵与  $1024 \times 512$  矩阵的乘积运算, 相较于 CPU 实现版本, GPU 的标量实现形式加速比为 16.82。另一方面, 通过比较 GPU 版本矩阵乘法的标量实现形式和向量实现形式, 可以发现使用 OpenCL 的

表3.1 矩阵乘法不同实现版本的能效对比

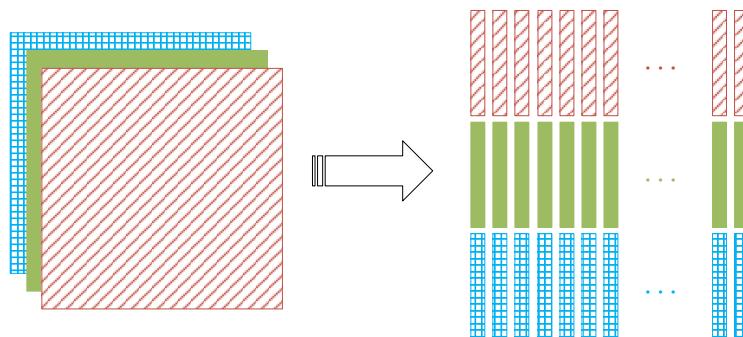
实现版本	EDP(Joules*seconds)
CPU 版本矩阵乘法	1402.763
GPU 版本矩阵乘法 (标量形式)	2.663
GPU 版本矩阵乘法 (向量形式)	0.729

`float4` 向量数据类型可以再获得约 2 倍的性能提升并可进一步降低运行时能耗。观察图3.2显示的平均功耗信息可知，使用手机 CPU 执行矩阵内积操作的功耗大约是使用手机 GPU 时的 2 倍。这也说明了在移动平台 SoC 的设计中，手机 CPU 功耗是高于 GPU 的。另外，与仅使用 GPU 标量数据类型相比，使用 GPU 向量数据类型执行 SIMD（单指令多数据流）指令时功耗会略有上升。由表3.1可知，GPU 版本矩阵乘法在能效方面远高于 CPU 版本矩阵乘法，尤其在使用 OpenCL 向量数据类型实现时 GPU 版本的能效提升了近两千倍（EDP 的定义见2.4.1节，其值越小说明能效越高）。

### 3.1.2. 卷积层基本算子

卷积层的基本算子即为卷积操作，这在第二章进行了相关介绍。为了提高卷积操作的执行性能，本文采用了 `img2col` 结合通用矩阵乘的实现方式。

根据第二章介绍的卷积操作可知，每次滑动卷积窗口，都需要将卷积核的每一个权重值与窗口对应处的图像数据值相乘，并将所有的乘积相加求和，整个操作过程非常类似于向量的内积运算。因此，理论上可以通过矩阵乘法完成卷积操作，而这其中的关键之处即是需要使用 `img2col` 算子对输入矩阵数据进行转换。

图3.3 `img2col` 转换三通道输入特征图示例

将卷积操作转换为矩阵内积运算需要进行如下两步操作：(1) 使用 `img2col`

将输入特征图转换为使用矩阵乘法实现卷积操作时所需的矩阵形式, 图3.3给出了一个使用 `img2col` 将三通道输入特征图转换为所需矩阵形式的示例; (2) 使用矩阵乘法将权重矩阵与第一步转化而来的矩阵进行相乘得到卷积输出结果。算法3.4详细描述了 `img2col` 的执行过程。

```

Data: channels, channel_size, kernel_h, kernel_w, output_h, output_w
1 channels, channel_size 分别表示输入通道数和一个通道的数据量;
2 kernel_h, kernel_w 分别表示卷积核的高度和宽度;
3 output_h, output_w 分别表示卷积层输出特征图的高度和宽度;
4 for channel in 0 ... channels-1 do
5   for kernel_row in 0 ... kernel_h-1 do
6     for kernel_col in 0 ... kernel_w-1 do
7       计算卷积核中第 kernel_row 行首个操作区域在输入特征图上的行索引;
8       for output_rows in output_h ... 1 do
9         if 计算得到的输入特征图的行值索引小于零或者大于输入特征图的
10        高 then
11          for output_cols in output_w ... 1 do
12            | 将该行在输出矩阵上的位置置为 0;
13          end
14        else
15          计算卷积核中第 kernel_col 列首个操作区域在输入特征图上的
16          列索引;
17          for output_cols in output_w ... 1 do
18            if 输入特征图的列值索引大于等于零或者小于输入特征图
19            的宽 then
20              | 将输入特征图上对应的区域放到输出矩阵上;
21            end
22            else
23              | 将该行该列在输出矩阵上的位置置为 0;
24            end
25          将输出列坐标移动到下一个卷积窗口;
26        end
27      将输出行坐标移动到下一个卷积窗口;
28    end
29  end
30  将输出矩阵偏移 channel_size 以操作下一个输入通道的图像数据;
31 end
```

**算法 3.4:** `img2col` 核心操作伪代码

根据算法3.4描述的伪代码,本文分别于手机GPU和CPU上实现了 `img2col` 操作。为了比较 `img2col` 在手机GPU和CPU上的执行性能,本文基于ODROID-XU3 平台测量了 `img2col` 在处理高度和宽度均为 256、通道数为 100 的输入特征图(步长为 1, 卷积核大小为  $3 \times 3$  并且无 padding 处理)的运行时间、能耗和

平均功耗，结果如图3.4所示。

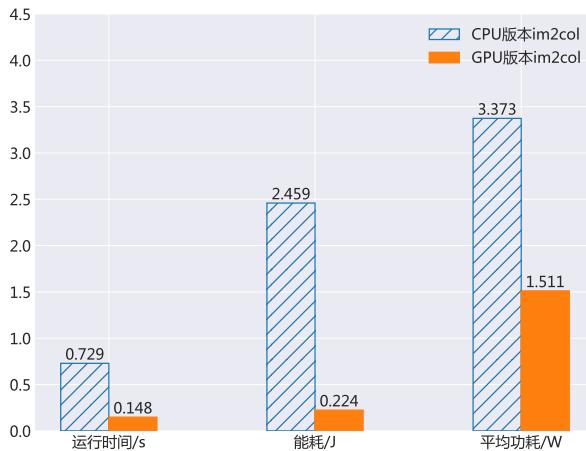


图 3.4 img2col 不同实现版本的运行时间、能耗和平均功耗对比

由图3.4可知，在处理高度和宽度均为 256、通道数为 100 的输入特征图时，相较于 CPU 版本的 img2col，GPU 版本的 img2col 可使得执行速度提升 5 倍左右。与此同时，使用 GPU 执行 img2col 操作的能耗仅为使用 CPU 能耗的 1/10，而平均功耗不到使用 CPU 时的 1/2。表3.2显示了 GPU 版本 img2col 和 CPU 版本 img2col 的 EDP 值。对比两者的 EDP 值可知，基于 GPU 实现的 img2col 可以将能效提升 54 倍以上。

表 3.2 img2col 不同实现版本的能效对比

实现版本	EDP(Joules*seconds)
CPU 版本 im2col	1.792
GPU 版本 im2col	0.033

### 3.1.3. 池化层基本算子

显然，池化层的基本算子即为池化单元。由2.1.2节可知，最大池化是目前研究工作中最为常用的池化单元，因此本文主要实现了最大池化操作。根据第2.1.2节中对最大池化操作的自然语言描述，可形成如算法3.5所示的最大池化操作伪代码。同样地，本文于手机 GPU 和 CPU 上分别实现了算法3.5所描述的伪代码。

对于一张高度和宽度均为 256、通道数为 100 的输入特征图，本文测量了其在 ODROID-XU3 平台上进行最大池化操作所需的运行时间、能耗和平均功耗。实验中，池化操作的步长取为 1、池化核大小取为  $3 \times 3$ ，并且不进行 padding 处理，测量结果如图3.5所示。

```

Data: channels, pooled_h, pooled_w
1 channels 表示输入通道数;
2 pooled_h 和 pooled_w 分别表示池化输出矩阵的高度和宽度;
3 for c in 0 ... channels-1 do
4   for ph in 0 ... pooled_h-1 do
5     for pw in 0 ... pooled_w-1 do
6       计算本次池化窗口的行操作起始位置 hstart 和终止位置 hend;
7       计算本次池化窗口的列操作起始位置 wstart 和终止位置 wend;
8       确保本次行列操作起始位置 hstart 和 wstart 均为非负值;
9       计算本次池化操作输出矩阵的对应下标 pool_index;
10      将本次池化窗口左上角第一个值赋给输出矩阵下标为 pool_index 的项;
11      for h in hstart ... hend-1 do
12        for w in wstart ... wend-1 do
13          利用上面两重循环, 遍历本次池化窗口中每一个值 value;
14          将 value 与输出矩阵下标为 pool_index 的项做比较;
15          将两者较大值重新赋值给输出矩阵下标为 pool_index 的项;
16        end
17      end
18    end
19  end
20  将输入特征图和输出矩阵均偏移到下一个通道;
21 end

```

算法 3.5: 最大池化核心操作伪代码

由图3.5可知, 使用 GPU 进行最大池化操作不仅可以将运算速度提升 12 倍还可以把能耗降为使用 CPU 时的 1/28。最大池化在 CPU 和 GPU 上执行的功耗特征与 img2col 类似, 即 GPU 版本最大池化平均功耗不足 CPU 版本的一半。表3.3从能效的角度分析了最大池化在不同计算设备上的表现, 即 GPU 版本最大池化的能效要高出 CPU 版本约 340 余倍。

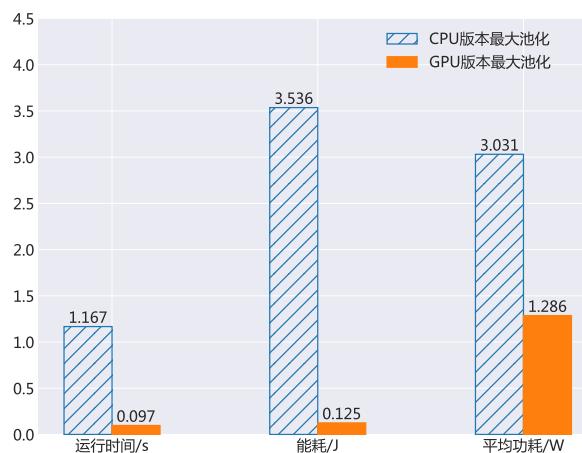


图 3.5 最大池化不同实现版本的运行时间、能耗和平均功耗对比

表3.3 最大池化不同实现版本的能效对比

实现版本	EDP(Joules*seconds)
CPU版本最大池化	4.125
GPU版本最大池化	0.012

### 3.1.4. 激活层基本算子

正如2.1.4节所述，激活层主要使用非线性激活函数对输入数据进行处理，故而激活层的基本算子即为所使用的激活函数。在具体实现中，可选的激活函数主要有 Relu 函数、Sigmoid 函数以及双曲正切函数等。选定激活函数 `activation_fun()`，激活层基本算子的实现伪代码可表示如下：

```

Data: data, length
1 for i in 0 ... length-1 do
2   |   data[i]=activation_fun(data[i]);
3 end

```

算法3.6: 激活层基本算子实现伪代码

根据算法3.6所描述的实现逻辑，本文分别实现了基于手机 GPU 和 CPU 的激活操作。图3.6显示了 Relu 激活算子分别于手机 GPU 和 CPU 上处理  $100 \times 256 \times 256$  个输入数据时的运行时间、能耗和平均功耗。



图3.6 Relu 激活算子不同实现版本的运行时间、能耗和平均功耗对比

从图3.6可以看出，在输入数据量为  $100 \times 256 \times 256$  时，GPU 版本 Relu 激活算子的执行速度可比 CPU 版本的快 6 倍，同时能耗和功耗分别降为使用 CPU 时的  $1/22$  和  $2/7$ 。表3.4给出了 Relu 激活算子分别在 CPU 和 GPU 上执行时的能效。其中，CPU 版本 Relu 激活算子的 EDP 为 0.034，而 GPU 版本 Relu 激活算

子的 EDP 仅为 0.0003。根据 EDP 值越小能效越高的度量准则可知，在输入数据量为  $100 \times 256 \times 256$  时，Relu 激活算子在 GPU 上的运行时能效是其在 CPU 上的 110 多倍。

表 3.4 Relu 激活算子不同实现版本的能效对比

实现版本	EDP(Joules*seconds)
CPU 版本 Relu 激活算子	0.034
GPU 版本 Relu 激活算子	0.0003

## 3.2 基于手机 GPU 加速的 CNN 前向推断

基于3.1节对 CNN 前向推断过程中所涉及基本算子的分解与实现，本文设计了一套可使用手机 GPU 或 CPU 执行 CNN 前向推断的运行时库。该 CNN 推断时库主要包括了卷积层、池化层、全连接层和激活层等网络层的实现。利用所设计的 CNN 推断时库，本章节在手机移动平台上重构了 LeNet-5 模型和 AlexNet 模型，并分析比较了两个模型分别在手机 GPU 和 CPU 上执行推断时的能效特征。

### 3.2.1. 预训练 CNN 模型权重参数解析

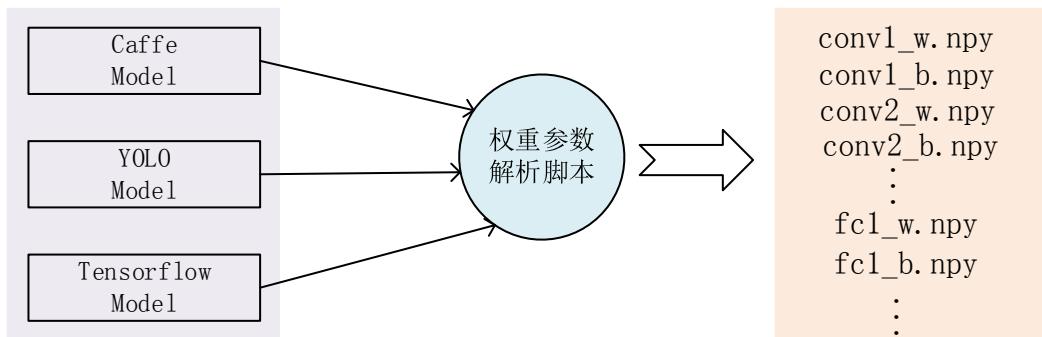
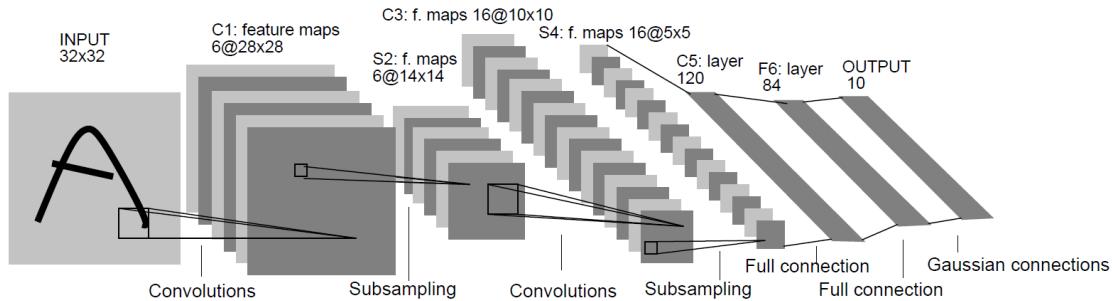


图 3.7 CNN 模型权重参数解析流程

解析预训练 CNN 模型权重参数是在移动端重构卷积神经网络的首要条件，本文针对经 Caffe、YOLO 和 Tensorflow 等深度学习框架训练得到的 CNN 模型给出了权重解析的 python 脚本。该脚本可以提取预训练 CNN 模型的每层权重参数（权值矩阵和偏置矩阵等），并将这些权重参数分层保存为二进制文件格式，整个流程如图3.7所示。

### 3.2.2. LeNet-5模型于移动端的重构

图 3.8 LeNet-5 模型结构示意图<sup>[39]</sup>

LeNet-5 是由卷积神经网络之父 Yan LeCun 提出的，其主要用于手写字符的识别与分类。原始的 LeNet-5 模型主要由 2 个卷积层、2 个池化层、1 个全连接层以及 1 个高斯连接层组成，如图3.8所示。本文所采用的 LeNet-5 模型将最后一层高斯连接层替换了全连接层，且在 MNIST 数据集<sup>[51]</sup> 上的预测精度可达 99.1%。表3.5描述了本文所采用的 LeNet-5 模型结构中卷积层和全连接层的参数信息。

表 3.5 LeNet-5 模型结构中的卷积层和全连接层

名称	类型	描述
conv1	卷积层	输入: $1 \times 28 \times 28$ , 卷积核: $20 \times 1 \times 5 \times 5$ , 步长: 1, padding: 0, 输出: $20 \times 24 \times 24$
conv2	卷积层	输入: $20 \times 12 \times 12$ , 卷积核: $50 \times 20 \times 5 \times 5$ , 步长: 1, padding: 0, 输出: $50 \times 8 \times 8$
ip1	全连接层	输入: 800, 输出: 500
ip2	全连接层	输入: 500, 输出: 10

利用3.2.1节的权重解析脚本，可获取到 LeNet-5 模型各层的权重重矩阵。基于这些权重重矩阵，本文分别于手机 GPU 和 CPU 上重构了 LeNet-5 模型，图3.9显示了 LeNet-5 模型在 ODROID-XU3 平台上执行单张图片前向推断所需的运行时间、能耗和平均功耗。

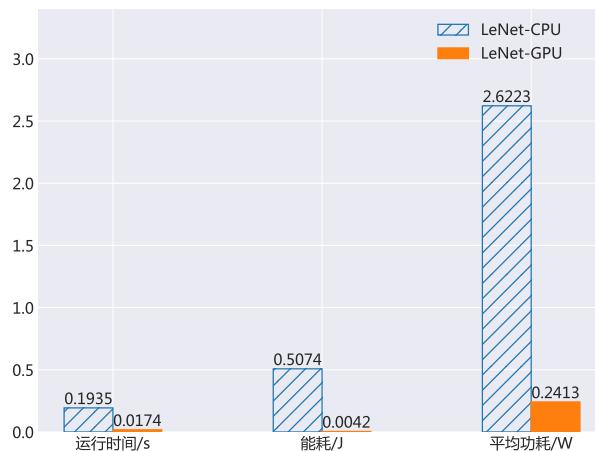


图 3.9 LeNet-5 单张图片推断运行时间、能耗和平均功耗对比

由图3.9可知，LeNet-5模型在手机GPU上进行前向推断的执行速度是在手机CPU上的11倍以上，并且可以节省能耗120多倍。根据表3.6显示的LeNet-5模型分别于手机GPU和CPU上运行时EDP值可知，使用手机GPU进行LeNet-5的前向推断可以将能效提升1400倍以上。

表 3.6 LeNet-5 分别于手机 GPU 和 CPU 上运行时能效对比

运行处理器	EDP(Joules*seconds)
CPU	0.0982
GPU	0.00007

如图3.9所示，使用手机GPU执行LeNet-5前向推断的平均功耗仅为0.2413瓦。这与图3.4显示的使用手机GPU进行im2col运算时的平均功耗1.511瓦相比，可以发现LeNet-5模型在手机GPU上进行前向推断时并未充分利用GPU的计算性能。这可能是因为LeNet-5模型太小，未能充分发挥GPU的并行计算能力。为了验证这一猜想，本文进一步于在手机移动平台上重构了AlexNet模型。

### 3.2.3. AlexNet 模型于移动端的重构

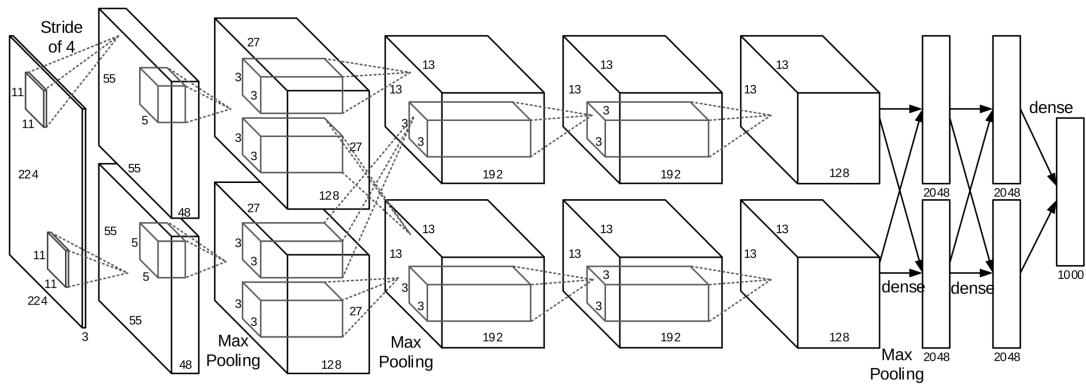


图 3.10 AlexNet 模型结构示意图 [2]

AlexNet 模型在 2012 年 ImageNet 大赛上一举夺冠并由此开启了深度学习的新时代，它也为后续的 CNN 模型研究奠定了基础，其模型结构如图3.10所示。AlexNet 模型主要由 5 个卷积层、3 个池化层和 3 个全连接层组成。其中，AlexNet 的卷积层和全连接层包含了模型所有权重值，它们的参数信息见表3.7。

图3.11显示了 AlexNet 模型分别在 ODROID-XU3 平台 GPU 和 CPU 上进行单张图片推断的运行时间、能耗和平均功耗对比。与图3.9中 LeNet-5 模型的推断运行相比，AlexNet 模型在手机 GPU 上执行前向推断相较于手机 CPU 而言性能加速比可达 15。从功耗角度分析可得，AlexNet 模型在手机 GPU 上执行前向推断

表3.7 AlexNet模型结构中的卷积层和全连接层

名称	类型	描述
conv1	卷积层	输入: $3 \times 227 \times 227$ , 卷积核: $96 \times 3 \times 11 \times 11$ , 步长: 4, padding: 0, 输出: $96 \times 55 \times 55$
conv2	卷积层	输入: $96 \times 27 \times 27$ , 卷积核: $256 \times 96 \times 5 \times 5$ , 步长: 1, padding: 2, 输出: $256 \times 27 \times 27$
conv3	卷积层	输入: $256 \times 13 \times 13$ , 卷积核: $384 \times 256 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $384 \times 13 \times 13$
conv4	卷积层	输入: $384 \times 13 \times 13$ , 卷积核: $384 \times 384 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $384 \times 13 \times 13$
conv5	卷积层	输入: $384 \times 13 \times 13$ , 卷积核: $256 \times 384 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $256 \times 13 \times 13$
fc6	全连接层	输入: 9216, 输出: 4096
fc7	全连接层	输入: 4096, 输出: 4096
fc8	全连接层	输入: 4096, 输出: 1000

时手机GPU的平均功耗(1.31瓦)可接近其峰值。这也说明了手机GPU在执行AlexNet这种结构较为复杂的CNN模型推断时加速效果更加明显。虽然AlexNet模型在手机GPU上执行前向推断的平均功耗比LeNet-5模型增加了1瓦,但是其仍然只是使用CPU执行推断时功耗的一半,也因此其能耗只达到了使用CPU推断的1/30。由表3.8显示的EDP值可知,相较于手机CPU而言,使用手机GPU执行AlexNet模型推断时能效可提升472倍以上。这充分显示了手机GPU在执行卷积神经网络计算上的优势。

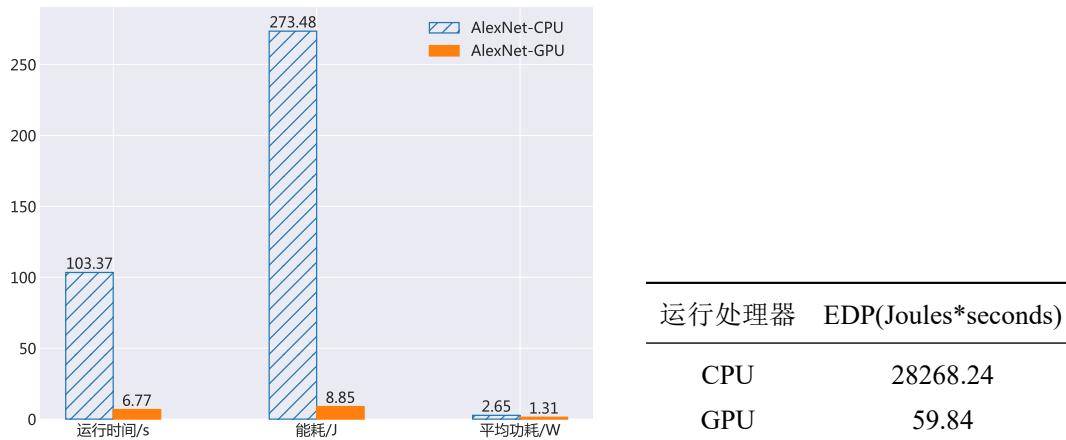


图3.11 AlexNet单张图片推断运行时间、能耗和平均功耗对比

表3.8 AlexNet分别于手机GPU和CPU上运行时能效对比

为了验证本文所开发CNN推断时库的可移植性与兼容性,本文在Open-Q™820开发套件上也重构了AlexNet模型。图3.12显示了AlexNet模型分别于ODROID-XU3平台和Open-Q™820开发套件上进行单张图片推断的CPU运行时间和GPU运行时间。对比在两个平台上使用手机CPU执行AlexNet前向推断的运行时间可知,骁龙820的CPU相较于三星Exynos5422的CPU在计算性能上提升了4倍有余。而当使用手机GPU执行CNN前向推断时,骁龙820的处理速度更是提升了11倍多,这使得在手机移动端完成一次完整的AlexNet前向

推断仅需要580毫秒。由此可以看出，移动平台SoC中的CPU和GPU处理性能都在日趋强大，这为深度学习模型（如卷积神经网络）在移动平台本地完成离线推断奠定了基础。

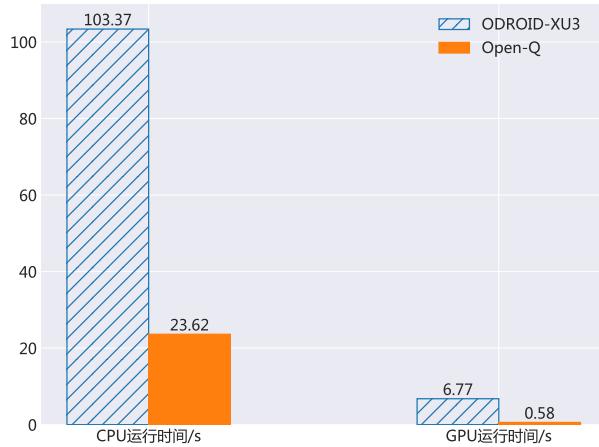


图3.12 不同平台上 AlexNet 单张图片推断运行时间对比

### 3.3 基于“剪枝-重训”的层压缩优化

神经网络是一种典型的过参数化模型，其通常包含着大量的冗余参数，例如：AlexNet模型采用Caffe框架的caffemodel格式进行存储时大小约为244MB。这种过多的冗余参数会导致计算和内存占用两方面的浪费。另一方面，移动手机相较于桌面电脑而言，其往往配备着较低容量的内存，如华为Mate 9 Pro和三星Galaxy S8的内存大小均为4GB。分析卷积神经网络模型权重参数分布可知，全连接层参数占了整个网络参数的绝大部分，如AlexNet模型中全连接层参数占整个网络参数的95%以上。因此，为了降低CNN模型在移动端进行前向推断时的内存占用开销，本文采用“剪枝-重训”方法对CNN模型的全连接层参数进行压缩。

“剪枝”的主体思想就是将不重要的冗余权重连接删除，只保留网络模型中最重要的连接部分。本文参考《Learning both weights and connections for efficient neural network》<sup>[52]</sup>文章中的通用剪枝思想将权值绝对值大小低于某一阈值的连接视为不重要的冗余连接。“重训”即剪枝后重新训练，这主要为了保证剪枝操作不会造成网络模型的精度损失。很明显，如果直接删除网络模型中的部分连接而不做重新训练处理，那么模型的精度就会下降。故而，为了保证网络模型的精度不变，需要对剪枝后的网络进行重新训练。

在对网络模型进行剪枝时，必须要确定待剪枝层的权重连接保留率，即保留待剪枝层的权重连接数量。一般来说，权重连接数越多的层则其需要剪枝掉的连接数也应该越多，即权重连接保留率越低。试错实验法是目前用来确定权重连接

保留率的常用方法，它对每一个待剪枝层 L 进行如下操作：

1. 除 L 层以外，将其他层的权重连接保留率都设置为 1，即除 L 层外不剪枝；
2. 以从高到低的方式设置 L 层权重连接保留率大小（如从 0.95 以步长 0.05 递减到 0.05）；
3. 使用第二步设置的每一个权重连接保留率对 L 层进行剪枝，并对剪枝后的网络模型于测试集上进行推断以获取剪枝后模型的精度；
4. 根据模型精度的变化趋势确定 L 层的最低权重连接保留率。

### 3.3.1. 模型剪枝流程

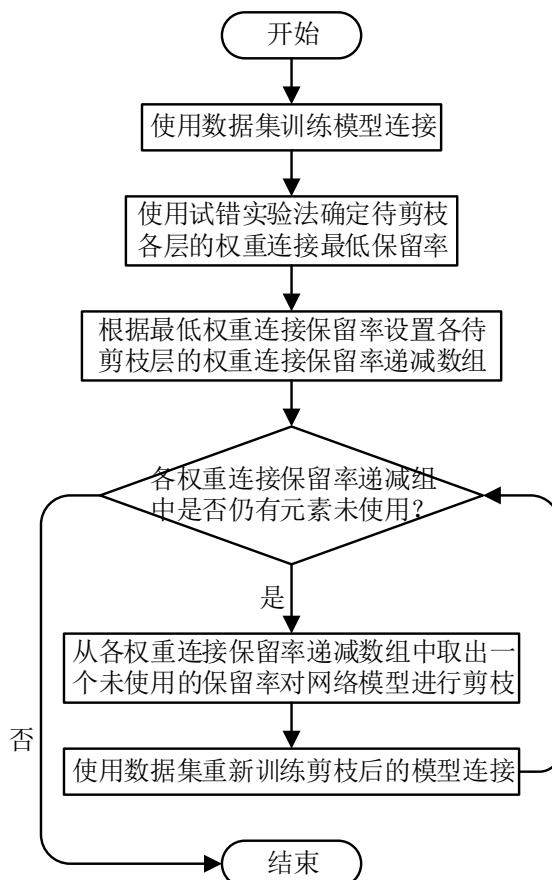


图 3.13 “剪枝-重训”流程概览

图3.13描述了模型“剪枝-重训”的整个过程。首先，使用数据的训练集对CNN模型进行训练以获得稠密权重连接。接着，使用试错实验法确定各待剪枝层的最低权重连接保留率。本文采用逐渐降低权重连接保留率的方式对网络模型进行迭代“剪枝-重训”，而不是使用最低权重连接保留率一次性对模型进行剪枝。因此，本文根据试错实验法得到的最低权重连接保留率来设置各待剪枝层的

权重连接保留率递减数组。举例来说，对于某一层 L，使用试错实验法发现将其保留率设为 0.1 对模型整体精度损失不大，则 L 层的权重连接保留率递减数组可设置成  $[0.95, 0.9, 0.85, \dots, 0.15, 0.1]$ 。最后，对各待剪枝层的权重连接保留率递减数组进行迭代：每次取出一组保留率对模型进行“剪枝-重训”操作，直至各待剪枝层的权重连接达到目标保留数量。

根据图3.13所描述的流程，本文对 Caffe 框架源码进行了修改，使其支持上述的“剪枝-重训”操作，并针对 LeNet-5 模型和 AlexNet 模型进行了实验。另外，对于稀疏矩阵，本文采用 CSR 格式进行存储，即只保存稀疏矩阵的非零值、行的范围以及非零值的列下标。

### 3.3.2. LeNet-5 模型剪枝

图3.14显示了 LeNet-5 模型精度随两个全连接层权重连接保留率变化的趋势。从趋势图上可以看出：全连接层 ip1 包含的冗余连接数过多，所以减少其权重连接对整个模型的前向推断精度影响不大；而全连接层 ip2 受剪枝影响较大，当只保留其 5% 的权重连接时会导致整个模型的推断精度下降到 82% 左右。因此，为了保证剪枝后 LeNet-5 模型的推断精度仍在 90% 以上，本文将全连接层 ip1 和 ip2 的最低权重连接保留率分别设为 0.05 和 0.1。

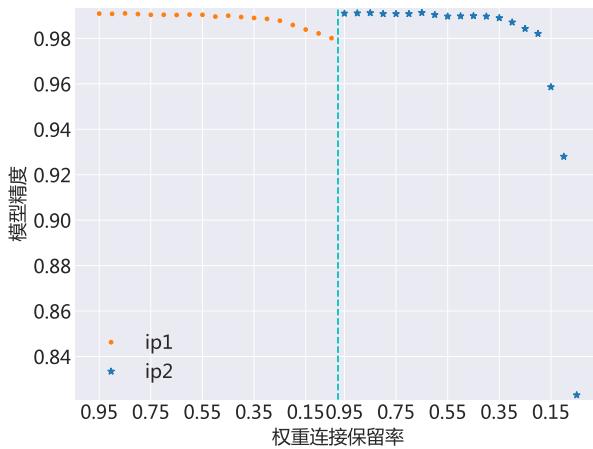


图 3.14 LeNet-5 模型精度随全连接层权重连接保留率变化的曲线

使用图3.13所示的“剪枝-重训”流程对 LeNet-5 模型进行剪枝后，可以发现其模型推断精度仍可高达 99.05%，这几乎没有导致精度上任何损失。剪枝后的模型经 CSR 格式进行存储后，模型大小仅为原始模型的 15.65%。在此需要说明，虽然全连接层 ip1 和 ip2 的权重参数分别减少了 95% 和 90%，但是由于 CSR 存储格式需要存储稀疏矩阵中所有非零值元素的两类信息（元素值和其对应的列下标），所以模型大小仅压缩到原始模型的 15% 左右。由 LeNet-5 模型所有全连接层剪枝前后的权重分布对比图3.15可知，LeNet-5 模型经剪枝后全连接层的权重

只有很少一部分值为非零值，这说明剪枝操作确实将原有的稠密权重矩阵转换为了稀疏权重矩阵。

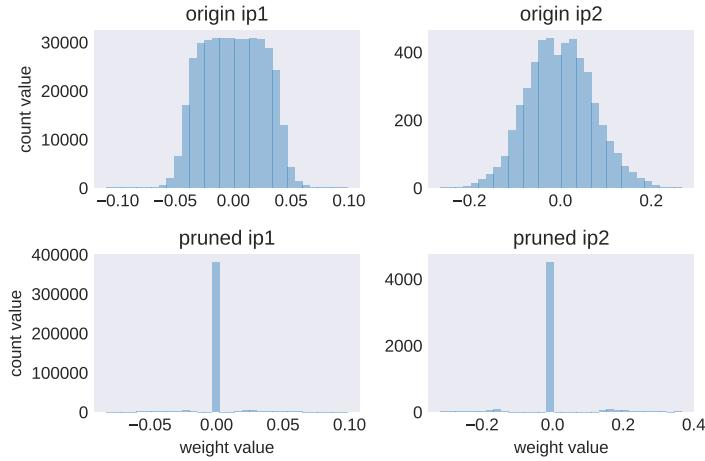


图 3.15 LeNet-5 模型全连接层剪枝前后权重分布对比

### 3.3.3. AlexNet 模型剪枝

与 LeNet 模型剪枝类似，本文首先使用试错实验法确定 AlexNet 模型各全连接层的最低权重连接保留率，其模型精度随不同全连接层权重连接保留率变化的趋势如图3.16所示。通过对比分析可知，fc6 层的剪枝对模型精度影响最大，而 fc8 层的剪枝对模型精度影响较小。为了保证剪枝后模型的 top-1 准确率仍在 50% 以上，本文将全连接层 fc6、fc7 和 fc8 的最低权重连接保留率均设置成 0.15。需要说明的是，本文所使用的 AlexNet 模型剪枝前在 ImageNet<sup>[53]</sup> 数据集上的 top-1 准确率为 55.78%。

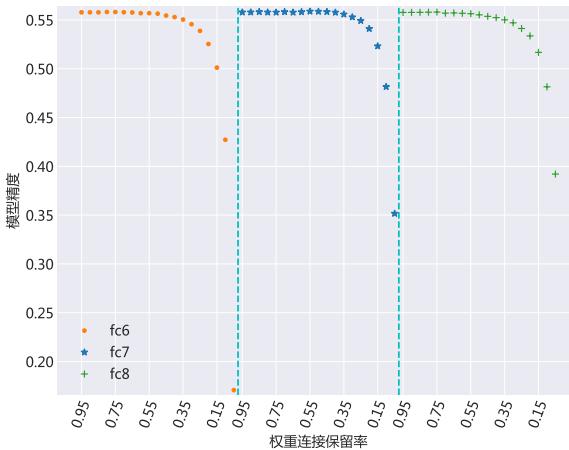


图 3.16 AlexNet 模型精度随全连接层权重连接保留率变化的曲线

经多次“剪枝-重训”操作后，AlexNet 模型的所有全连接层都仅保留了原有权重的 15%，而模型的 top-1 准确率变成了 54.80%，即模型精度仅下降了 0.98%。

之所以会造成模型精度 0.98% 的损失，是因为在模型权重压缩过程中对每次剪枝后的模型仅仅重新训练了 10000 次。如果增加每次剪枝后模型的重新训练次数，模型的精度损失就会减少。使用 CSR 格式对剪枝后的 AlexNet 模型进行存储，其模型大小约为 85MB，这比原始模型大小减少了约 159MB，仅为原始模型大小的 34.8%。图3.17显示了 AlexNet 模型的各全连接层剪枝前后权重分布对比。从分布图中可以明显看出，经剪枝后，AlexNet 模型所有全连接层的权重值仅有很少一部分值不为 0。

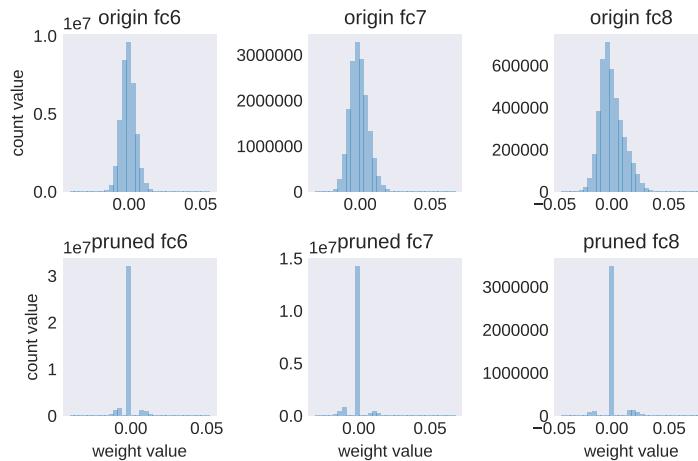


图 3.17 AlexNet 模型全连接层剪枝前后权重分布对比

基于 ODROID-XU3 平台，本文分析比较了 AlexNet 模型在剪枝前后的首次加载时间、能耗和平均功耗变化，结果如图3.18所示。分析图3.18中的数据可知，对 AlexNet 模型的全连接层进行剪枝后（仅保留全连接层原有权重的 15%），模型的首次加载时间减少了 60%，并且内存读取操作所产生的能耗也降为原来的一半。由此可见，对神经网络模型进行剪枝操作不仅可以降低模型的内存占用和内存能耗开销还可以提升模型的加载速度。

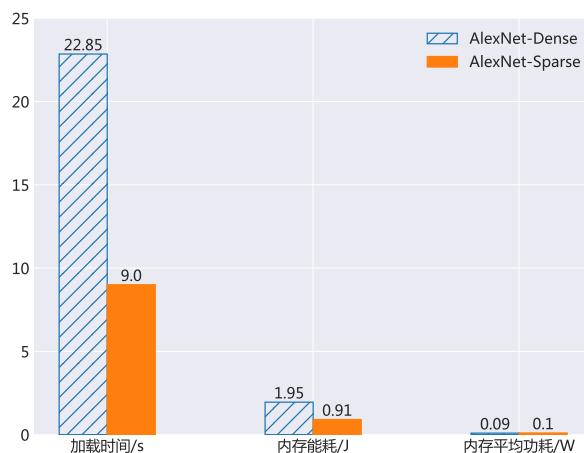


图 3.18 剪枝前后 AlexNet 首次加载时间、能耗和平均功耗对比

### 3.3.4. 移动端 SpMV 的实现

对于剪枝压缩后的卷积神经网络模型，本文使用稀疏矩阵向量乘（SpMV）代替原全连接层的稠密矩阵向量乘（内积操作）。对于一个使用 CSR 格式存储的稀疏矩阵与一个向量的乘积操作，其实现伪代码如算法3.7所示。

```
Data: num_rows, ptr, indices, data, bias, vec, out
1 num_rows 表示稀疏矩阵的总行数;
2 ptr, indices, data 分别表示行的范围、非零值的列下标以及稀疏矩阵的非零值;
3 bias, vec, out 分别表示偏置矩阵、乘数向量以及乘积结果;
4 for row in 0 ... num_rows-1 do
5     tmp = 0;
6     计算第 row 行的非零值下标范围 [start_row, end_row);
7     for j in start_row ... end_row-1 do
8         | temp += data[j] * vec[indices[j]];
9     end
10    out[row] = temp + bias[row];
11 end
```

算法 3.7: 稀疏矩阵向量乘实现伪代码

根据算法3.7所述过程，本文分别于手机 GPU 和 CPU 上实现了稀疏矩阵向量乘操作。图3.19显示了使用 ODROID-XU3 平台 GPU 和 CPU 进行稀疏矩阵向量乘和稠密矩阵向量乘的运行时间、能耗和平均功耗对比。实验中，矩阵维度为  $4096 \times 9216$ ，向量维度为  $9216 \times 1$ ，矩阵的非零值比重为 15%。

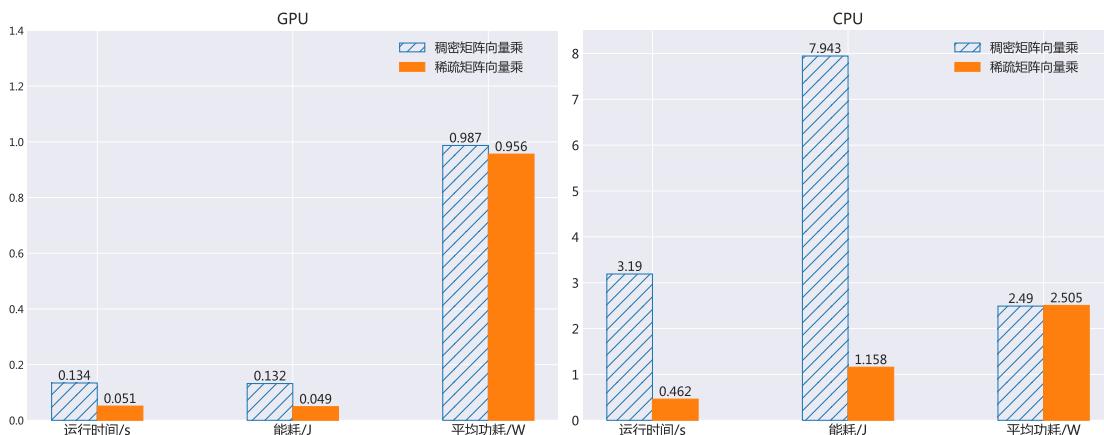


图 3.19 稠密矩阵和稀疏矩阵向量乘的运行时间、能耗和平均功耗对比

由图3.19可知，GPU 上使用 SpMV 代替内积运算，可将矩阵向量乘操作的运行速度提升 2.63 倍，并且完成乘积操作也只需原能耗的 1/3 左右。而在 CPU 上使用 SpMV 操作更可将运行时间和能耗均降为原来的 1/7 左右。这充分说明了矩阵的压缩操作不仅可以带来内存占用和能耗的降低，更可以提升矩阵乘积的计算速度。

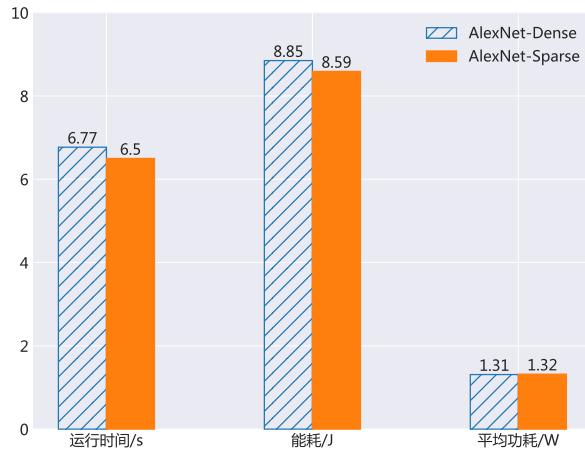


图 3.20 AlexNet 剪枝前后单张图片推断运行时间、能耗和平均功耗对比

对 AlexNet 模型全连接层进行剪枝并使用 SpMV 代替全连接层的内积算子后，再次于 ODROID-XU3 平台上测量其推断单张图片的运行时间、能耗和平均功耗，结果如图3.20中的 AlexNet-Sparse 数据系列所示。与剪枝前的稠密 AlexNet 模型相比，稀疏 AlexNet 模型的前向推断时间、能耗都有所降低，但降低幅度不大。这主要是因为全连接层的计算时间占整个 CNN 推断执行时间的比重较小。

### 3.3.5. 与 CNNdroid 的对比

CNNdroid<sup>[23]</sup> 是 Seyyed 等人开发的基于 GPU 加速的 CNN 推断时库。根据 CNNdroid 作者提供的源码，本文在 Open-Q™ 820 平台上使用其运行了 AlexNet 模型的前向推断过程，并将运行结果与本文所开发的 CNN 推断时库（使用 GPU）做对比，结果如图3.21所示。实验之所以未选择使用 ODROID-XU3 平台，是因为 CNNdroid 在 ODROID-XU3 平台上不能成功执行 AlexNet 模型推断，这可能与 CNNdroid 运行时内存占用过大或其使用的 GPU 编程框架存在限制条件有关。

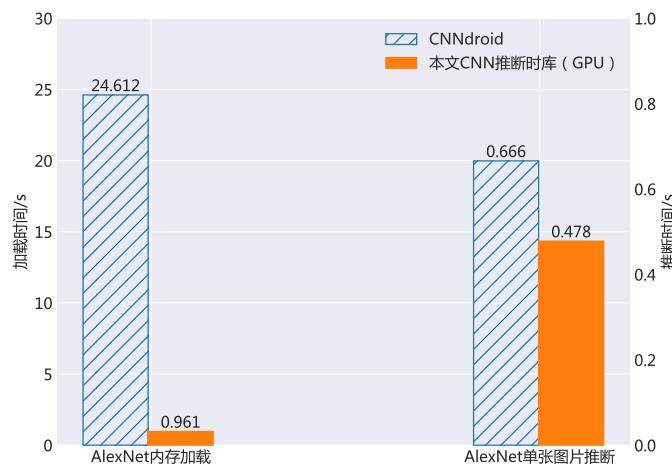


图 3.21 本文 CNN 推断时库与 CNNdroid 的对比

从图3.21中的 AlexNet 模型在两个推断时库中的内存加载时间可以看出，CN-

Ndroid 的 AlexNet 模型内存加载时间远高于本文所开发的 CNN 推断时库（大约是本文 CNN 推断时库的 25.6 倍）。根据表3.9中所示 AlexNet 模型在两个推断时库中的 SD 卡存储占用和内存占用可知，CNNDroid 不仅没有对 AlexNet 模型进行压缩，而且其采用的模型存储格式比 Caffe 框架的 caffemodel 格式所需存储占用更大。这不仅导致了 AlexNet 模型对 SD 卡存储占用和内存占用的需求极高，而且造成了图3.21中模型加载时间过长。由此可见，本文所采用的“剪枝-重训”模型压缩操作的必要性。

表 3.9 AlexNet 模型的存储占用和内存占用需求

	SD 卡存储占用 (MB)	内存占用 (MB)
CNNDroid	290	~ 300
本文 CNN 推断时库	85	~ 90

图3.21也显示了 CNNDroid 和本文 CNN 推断时库的 AlexNet 模型单张图片推断时间，它们分别是两个推断时库执行 100 次 AlexNet 模型推断的平均运行时间。对比两个推断时库的模型推断执行时间可知，本文所开发的 CNN 推断时库因为使用了模型压缩和 SpMV 等优化操作，运行 AlexNet 模型推断的速度要快于 CNNDroid（加速比约为 1.4）。

通过以上分析可以发现，本文所采用的模型压缩和 SpMV 等优化操作使得所开发的 CNN 推断时库无论是在内存加载速度还是在模型推断执行性能上均高于 CNNDroid。虽然对 CNN 模型权重进行“剪枝-重训”压缩可能会造成一定的模型推断精度损失，但是这种精度损失很小（本文压缩后 AlexNet 模型精度下降不足 1%），而且使用合理地剪枝以及充分地再训练操作也可以避免这种模型的精度损失。

### 3.4 本章小结

本章主要介绍了基于手机 GPU 加速和模型压缩的 CNN 前向推断能效优化方法。3.1 节对 CNN 模型前向推断过程中所使用的基本算子进行了分解与移动端的实现，并基于 ODROID-XU3 平台分析比较了各个基本算子在手机 GPU 和 CPU 上的运行时能效。3.2 节根据所实现的各种基本算子设计了一套 CNN 推断时库。基于该推断时库，本文在移动端重构了 LeNet-5 模型和 AlexNet 模型，并分析了两个模型在手机 GPU 和 CPU 上进行单张图片推断时的能效。3.3 节介绍了“剪枝-重训”的模型压缩方法，并实现了对卷积神经网络全连接层权重的压缩。通过对 LeNet-5 和 AlexNet 两个模型的剪枝操作，本文验证了使用压缩方法

提升 CNN 模型于移动端运行时能效的有效性。基于压缩后的稀疏网络模型，本文使用稀疏矩阵向量乘操作 (SpMV) 代替全连接层的内积操作，进一步增强了所开发推断时库在 Android 平台上的可用性。最后，与 CNNdroid 的对比实验展示了本文所采用 SpMV 和模型压缩等优化方法所带来的性能提升。截至本章，本文已经设计与实现了一套可以分别于手机 GPU 和 CPU 上运行的 CNN 推断时库，该库还通过 SpMV 操作支持经“剪枝-重训”压缩处理的稀疏 CNN 模型。



## 第4章 基于异构设备处理器并行执行推断的能效优化

本章首先探讨了移动端 SoC 的发展趋势及其对深度学习模型于移动端进行离线推断的影响。通过对比分析 CNN 前向推断分别在手机 GPU 和 CPU 上的执行能效，4.2节详细阐述了使用移动设备上所有可获得的本地异构处理器运行 CNN 模型推断并非是一种高能效的方式。4.3.1节提出了一种无需人为干预、可自适应计算目标移动平台上所有本地处理器能效的算法。使用该算法可进一步获得目标移动平台上可用于执行 CNN 前向推断的高能效设备处理器组合。4.3.2节给出了一种为所选择设备处理器组合分配计算任务的方法。4.4节基于 ODROID-XU3 平台实现了本文所提出的算法，并验证了这些算法的有效性。

### 4.1 移动端 SoC 的发展趋势

多核异构 CPUs（如 ARM 的 big.LITTLE<sup>[49]</sup>）已然成为当前移动设备处理器的主流架构，而 GPUs 也已集成到绝大多数移动设备中。GPUs 与生俱来的并行计算能力很适合处理深度模型中的常见计算类型。但是，处理能力较强的 GPUs 对移动设备的电池电量消耗也是惊人的。事实上，手机 GPUs 的设计过程更加重视的是低功耗而不是高性能，所以当前商业上使用的移动 GPUs 大部分计算能力并不是很强大，如 Mali™-T628 MP6 的最高运行频率为 600MHz 且核心数仅为 6。因此，单独的 GPUs 解决方案并不能满足移动平台上深度学习模型的运行条件。

除 GPUs 外，移动设备中还集成了一些其他异构处理器，如 DSPs、LPUs、NPUs 等。高通骁龙系列 SoC 不仅配备了 Adreno GPU，还集成了 Hexagon DSP；英伟达 Tegra K1 SoC 除提供高性能的 192 核 GPU、2.3GHz 的 4 核 CPU 外，还提供了一个第五代低功耗核 LPC；华为的海思麒麟 970 内置了神经网络处理单元（NPU），使用 NPU 可进行高效的 AI 相关计算。另外，英伟达如今已跟芯片设计公司 ARM 达成合作，将开源的 NVIDIA 深度学习加速器（NVDLA）架构集成到 ARM 的 Project Trillium 平台上，从而更好地实现移动端机器学习。在 2018 年的世界移动通信大会（MWC）上，联发科亦宣布推出 Helio P60 芯片，该芯片使用的 ARM Cortex™-A73 是专门为移动端 AI 应用设计的处理器架构，其可用于实现基于深度学习的面部检测、物体与场景辨识等算法。由此可见，移动端 SoC 将会越来越多地配备不同用途的专用处理器，并且每一种处理器都会拥有着不同的资源特征。因此，根据层的类型以及模型的资源需求差异，使用不同的处理器组合执行不同的深度学习模型必然可以带来不同的性能-功耗折中<sup>[54]</sup>。

## 4.2 基于异构计算的 CNN 并行推断研究动机

之前的研究工作要么在 CNN 推断中没有很好地利用手机移动设备的异构计算能力（如仅使用 CPU 或 GPU 进行前向推断），要么总是试图利用所有可获得的本地设备处理器进行模型推断，而本章主要探索如何高能效地利用手机移动设备的异构计算能力完成 CNN 模型的离线推断过程。

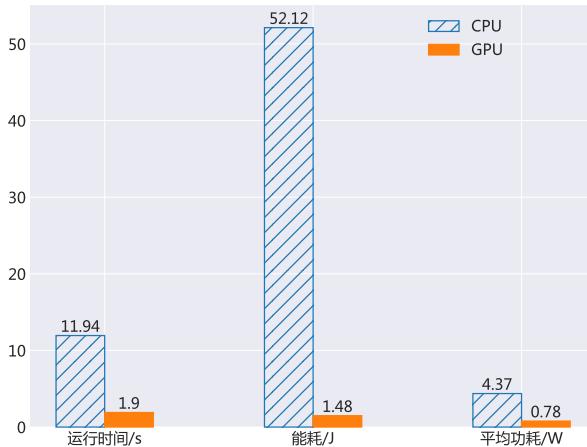


图 4.1 手机 GPU 和 CPU 执行 CNN 推断的运行时间、能耗和平均功耗

图4.1显示了分别在手机 GPU 和 CPU 上运行一次完整 CNN 推断所需时间、能耗和平均功耗。为了充分利用手机 CPU 的计算性能，实验中 CPU 执行 CNN 前向推断时使用了 ARM 处理器提供的 NEON 指令（一种单指令多数据流）并且启用了与处理器核心数相同的线程。由图4.1可知，即使利用了 CPU 所有的并行处理能力，CPU 执行一次完整的 CNN 前向推断过程仍需要耗时 11.94 秒、耗能 52.12 焦。然而，当使用 GPU 去处理相同的一次推断时，运行时间和能耗仅为 1.9 秒和 1.48 焦。由此可见，若只是一味地追求执行速度，可以同时使用手机 GPU 和 CPU 并行地执行 CNN 前向推断。但是，这种方式必然导致一个很低的执行能效。原因可从表4.1所示 CPU 和 GPU 的 EDP 能效值得出，即 GPU 的推断能效是 CPU 的 221 倍。因为手机的电池电量总是有限的，所以手机系统必须在程序执行速度与能耗开销之间维持一个良好的折中。

表 4.1 分别于手机 GPU 和 CPU 上执行一次完整 CNN 推断的能效

运行处理器	EDP(Joules*seconds)
CPU	622.31
GPU	2.81

基于上述分析，本文得出结论：不同的 CNN 模型在手机移动平台上执行前向推断时，需要针对性地选择一个高能效的本地异构计算组合而非简单地利用

所有可获得的异构设备处理器。

### 4.3 基于异构计算的自适应计算任务分配策略

图4.2显示了基于异构计算的自适应计算任务分配策略的基本流程。首先，为了在目标移动平台上寻找一个可高能效并行执行 CNN 前向推断的异构设备处理器组合，手机应用的 CNN 运行时库会根据4.3.1节提出的算法自动评估本地所有不同设备处理器的能效。然后，一旦寻找到所需的设备处理器组合，CNN 运行时库会根据所选择的每一个处理器的性能对计算任务进行划分，划分方法详见4.3.2节。最后，所有的计算任务会被所选择的高能效设备处理器组合并行处理。

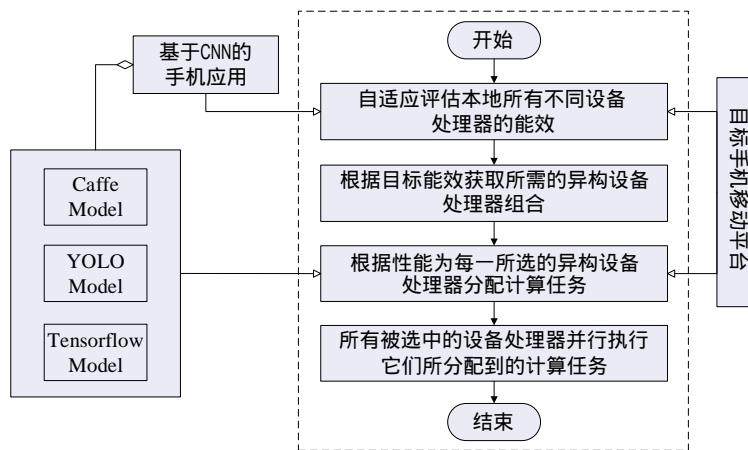


图 4.2 基于异构计算的自适应计算任务分配策略工作流程

#### 4.3.1. 高能效设备处理器组合的搜索

为了获得所期望的高能效设备处理器组合，首先需要了解目标手机移动平台上每一个可访问设备处理器的能效。本文使用一种简单有效的方法来刻画目标平台上所有设备处理器的性能和能耗。简单来说，若目标平台上所有可访问的设备处理器数量为  $n$ ，则在 CNN 推断时库的前  $n$  次运行中，每次都仅使用一个可访问的设备处理器执行完整的 CNN 推断。在每次执行完 CNN 推断后，推断过程在该设备上所需的执行时间、能耗、平均功耗以及 EDP 能效值会被记录下来。这一步会造成一些运行时开销，但是它仅仅发生在程序的前若干次运行中，所以这些开销会被程序的整个运行周期所均摊。为了给接下来的搜索步骤做准备，每一个本地设备处理器的推断性能和推断功耗将会被正则化。最后，根据所期望的能效值不断剔除较低能效的设备处理器便可获得所需的高能效设备处理器组合。需要强调的是，对于同一应用，高能效设备处理器组合可被重复使用，

因此整个搜索过程仅需执行一次。

```

Input: (i) CNN 模型结构参数和预训练的模型权重矩阵;
(ii) 能效阈值  $EDP_{TH}$ (默认值为 1)。
Output: 所期望的高能效处理器组合  $S_{HC}$  以及组合中所有处理器的相对性能集合
 $Perf$ 。
1 遍历目标平台上所有可访问的异构设备处理器(如 GPUs,CPPUs,NPPUs 等)并将它们
放入设备集合  $S_{HC}$  中;
2 for  $processor_i$  in  $S_{HC}$  do
3   使用处理器  $processor_i$  执行一次完整的 CNN 推断;
4   将本次推断的执行时间  $t_i$ , 能耗  $e_i$  以及平均功耗  $p_i$  分别存储到  $T$ ,  $E$  和  $P$  三个
   数组中;
5   使用公式2.1计算本次推断的能效  $EDP_i$  并将其存储在  $EDP$  数组中;
6 end
7 找出具有最小推断执行时间的处理器  $processor_{min}$ ;
8 将  $processor_{min}$  的执行时间和平均功耗分别标记为  $t_{min}$  和  $p_{ref}$ ;
9 for  $(t_i$  in  $T)$  and  $(p_i$  in  $P)$  do
10  计算相对性能:  $perf_{ri} = \frac{t_{min}}{t_i}$ , 将  $perf_{ri}$  添加到数组  $Perf$  中;
11  计算相对功耗:  $p_{ri} = \frac{p_i}{p_{ref}}$ , 将  $p_{ri}$  添加到数组  $P_r$  中;
12 end
13 repeat
14  计算相对执行时间:  $t_r = \frac{1}{\sum_{perf \in Perf} perf}$ ;
15  计算相对总功耗:  $p_{tr} = \sum_{pr \in P_r} pr$ ;
16  计算相对能效 EDP 值:  $edp_r = (p_{tr} * t_r) * t_r$ ;
17  if  $edp_r \geq EDP_{TH}$  then
18    找出具有最大 EDP 值的处理器, 并记录其下标  $i$ ;
19    从  $S_{HC}$  中移除  $processor_i$ ;
20    从  $Perf$  中移除  $perf_{ri}$ ;
21    从  $P_r$  中移除  $p_{ri}$ ;
22  end
23 until  $edp_r < EDP_{TH}$ ;
24 return  $S_{HC}$  和  $Perf$ ;

```

**算法 4.1:** 高能效设备处理器组合的搜索过程

算法4.1描述了在目标手机移动平台上进行高能效设备处理器组合搜索的核心过程。为了在手机本地端完成 CNN 前向推断过程, 模型结构参数和权重矩阵的获取是必须的, 这可以通过第3.2.1节提供的模型权重参数解析脚本实现。因为算法中使用了正则化处理, 所以默认的能效阈值  $EDP_{TH}$  设置为 1。当然, 用户也可以根据需要配置  $EDP_{TH}$  的值以得到不同的能效折中。

如算法4.1所示, CNN 推断时库首先会遍历目标平台上所有可访问的异构设备处理器, 如 CPUs、GPUs 或 NPPUs 等。然后, 所有可获得的设备处理器被收集到一个容器中, 这是整个搜索过程的初始化步骤(行 1)。使用本文提出的策略, 用户不需要提前测量不同设备处理器的性能和功耗。这是因为 CNN 推断时库在

它的前几次执行中会主动对每一个可获得设备处理器的性能和功耗进行评估并进一步计算出它们的能效 EDP 值（行 2-7）。例如，如果目标手机平台上配备有 CPUs、GPUs 和 DSPs，那么每个设备处理器都会完整地运行一次 CNN 推断并且它们的执行性能和功耗等信息都将被记录下来。这个方法也可以扩展应用到新的设备处理器上，如 NPUs、DianNaos<sup>[31]</sup> 等。接着，拥有最小执行时间的处理器会被挑选出来作为一个参考（行 8）。通过该参考处理器的性能和功耗信息，每一个处理器的性能和功耗将被正则化（行 9-12）。因为每个处理器 CNN 推断执行时间的倒数（即每秒处理图像数据的帧数，FPS）可反映它的推断执行性能，所以算法中处理器  $processor_i$  的相对性能可以通过公式  $\frac{\frac{1}{t_i}}{\frac{1}{t_{min}}} = \frac{t_{min}}{t_i}$  得到（行 10）。

为了充分利用并行计算的优势，每一个设备处理器所分配到的计算任务量是通过考虑其性能进行动态调整的。由算法的第 10 行可知，参考处理器（拥有最小推断执行时间的处理器）的相对性能为 1 并且其他处理器的性能都小于 1。因此，参考处理器的任务分配比可通过公式 4.1 进行计算：

$$ratio_{ref} = \frac{1}{\sum_{perf \in Perf} perf} \quad (4.1)$$

假定参考处理器的串行推断执行时间为  $t_s$ ，则其并行处理时间为  $ratio_{ref} \times t_s$ （进一步的讨论详见 4.3.2 节）。因为算法中使用了正则化处理，所以  $t_s$  可被简单地视为 1。这样，整个 CNN 推断的相对并行处理器时间  $t_r$  就等于  $ratio_{ref}$ （行 14）。显然，所选择设备处理器组合的功耗即为组合中所有处理器的功耗总和（行 15）。利用上述得到的相对执行时间和功耗即可通过公式 2.1 计算所选择的每一个设备处理器的相对能效 EDP 值  $edp_r$ （行 16）。然后，通过比较  $edp_r$  与目标能效阈值  $EDP_{TH}$  的大小，不断更新所选择的设备处理器组合。具体来说，若当前所选择设备处理器组合的能效（ $edp_r$ ）低于目标能效，则当前所选择组合中能效最低的设备处理器就会被剔除并且相应的性能和功耗数据集合也会被更新（行 17-22）；重复上述过程直至得到所需的高能效异构计算组合。最后，所选择的设备处理器组合以及它们的性能参数集会作为算法 4.1 的输出返回给 CNN 推断时库。

### 4.3.2. 计算任务的划分

在完成高能效设备处理器组合的搜索步骤之后，CNN 推断时库就可以根据算法 4.1 的返回结果将计算任务分配到所选择的处理器上执行。为了最小化并行处理时间，必须将不同处理器之间的等待时间尽量缩小。因此，一个本地设备处理器所分配到的计算任务量应该与其性能成正比，这即是计算任务划分方法的核心思想。

对于一个给定的预训练 CNN 模型，本文将每一层的一个输出节点定义成一

一个计算任务。换言之，每一层的计算任务量等于该层的输出节点数。因此，卷积层中一个计算任务就是一个卷积操作，而全连接层中一个计算任务就是一个内积操作。图4.3显示了一个计算任务划分的示例，其中每一条红线表示一个任务划分点。

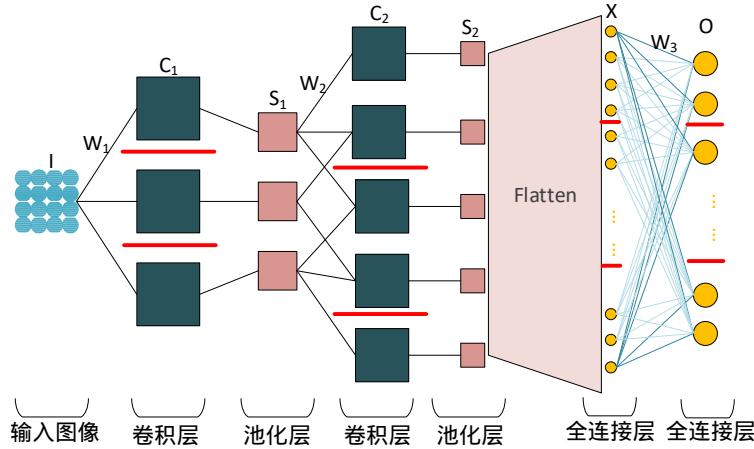


图 4.3 计算任务划分示意图（其中每一条红线表示一个划分点）

算法4.2详细描述了为每一个所选设备处理器计算任务分配比例的方法。如前所述，分配给每一个所选设备处理器的计算任务量取决于该处理器的性能。因此，算法4.1输出的相对性能参数值被用来计算每一个所选设备处理器的任务分配比例（行 1-4）。根据算法4.2输出的任务分配比例数组  $R$ ，每一个所选设备处理器的计算任务量就可通过其任务分配比例与每一层输出节点总数的乘积获得。通过这种划分方法，性能较高的处理器就会分配到更多的计算任务而性能较低的处理器则分配到较少的计算任务。结果使得所有不同大小的计算任务几乎可以在同一时间内完成。这样，系统就可以达到一个负载平衡并在推断过程中高能有效地利用平台所提供的异构计算资源。

**Input:** 所选设备处理器组合中所有处理器的相对性能集合  $Perf$ 。

**Output:** 所选组合中处理器所分配计算任务比例集合  $R$ 。

```

1 for  $perf_i$  in  $Perf$  do
2   | 计算处理器  $process_i$  所分配任务比例:

```

$$r_i = \frac{perf_i}{\sum_{perf \in Perf} perf}$$

| 将  $r_i$  添加到数组  $R$  中;

3 **end**

4 **return**  $R$ ;

**算法 4.2:** 设备处理器组合中每一处理器所分配任务比例的计算过程

对于同一应用而言，算法4.2的执行过程只需进行一次，之后其输出的任务分配比例数组可以保存到手机本地以便重复使用。故而，算法4.2的运行时开销

几乎可以忽略不计。

## 4.4 实验验证

基于第3章所开发的 CNN 推断时库，本章进一步实现了所提出的策略并在 ODROID-XU3 平台上进行了实验验证。为了充分利用 CPU 的并行计算能力，本章也对第3章所开发的 CNN 推断时库进行了优化，如在 CPU 版本的 CNN 推断过程中使用了 ARM 处理器提供的 NEON 指令并且启用了与处理器核心数相同的线程。实验中，本文将 CPU 和 GPU 的运行频率保持在某一固定值（如最大运行频率），这样可以消除设备处理器运行频率的变化对实验结果的影响。

如第2.4.2节所述，ODROID-XU3 平台上所有可获得的本地设备处理器包括一个 CPU 设备和两个 GPU 设备。因此，通过运行算法4.1和算法4.2，CNN 推断时库会得到一个高能效的设备处理器组合（这里即为 ODROID-XU3 平台上的两个 GPU 设备）。另外，实验中所考察的 CNN 模型主要由 9 个卷积层和 3 个全连接层构成，它们的结构参数信息如表4.2所示。为了说明所提策略的有效性，本文将所提策略（*Our work*）与贪心策略（*Greedy*，即总是试图使用所有可获得的异构设备处理器）进行了对比分析。

表 4.2 实验所用 CNN 模型的卷积层和全连接层结构参数

名称	类型	描述
Conv1	卷积层	输入: $3 \times 448 \times 448$ , 卷积核: $16 \times 3 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $16 \times 448 \times 448$
Conv2	卷积层	输入: $16 \times 224 \times 224$ , 卷积核: $32 \times 16 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $32 \times 224 \times 224$
Conv3	卷积层	输入: $32 \times 112 \times 112$ , 卷积核: $64 \times 32 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $64 \times 112 \times 112$
Conv4	卷积层	输入: $64 \times 56 \times 56$ , 卷积核: $128 \times 64 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $128 \times 56 \times 56$
Conv5	卷积层	输入: $128 \times 28 \times 28$ , 卷积核: $256 \times 128 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $256 \times 28 \times 28$
Conv6	卷积层	输入: $256 \times 14 \times 14$ , 卷积核: $512 \times 256 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $512 \times 14 \times 14$
Conv7	卷积层	输入: $512 \times 7 \times 7$ , 卷积核: $1024 \times 512 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $1024 \times 7 \times 7$
Conv8	卷积层	输入: $1024 \times 7 \times 7$ , 卷积核: $1024 \times 1024 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $1024 \times 7 \times 7$
Conv9	卷积层	输入: $1024 \times 7 \times 7$ , 卷积核: $1024 \times 1024 \times 3 \times 3$ , 步长: 1, padding: 1, 输出: $1024 \times 7 \times 7$
FC10	全连接层	输入: 50176, 输出: 256
FC11	全连接层	输入: 256, 输出: 4096
FC12	全连接层	输入: 4096, 输出: 1470

**性能：**图4.4显示了 9 个卷积层和 3 个全连接层在 ODROID-XU3 平台上的执行时间。总体来说，贪心策略的性能高于本文所提出的策略，这是因为贪心策略使用了更多的设备处理器执行计算任务。然而，贪心策略的性能提升幅度并不高，仅仅是略好于本文提出的策略，并且在全连接层上的差距非常小。这主要是因为 GPUs 的并行计算能力比使用了多线程和 NEON 指令的 CPUs 还要强大。由此可见，使用了一个额外 CPU 的贪心策略在性能上并没有表现出显著的提升。

从图4.4中也可以发现贪心策略中一些层的执行时间要长于本文提出的策略。这是因为贪心策略拥有更多的任务间交互开销，其在输出节点较少的层对执行时间的影响较大。

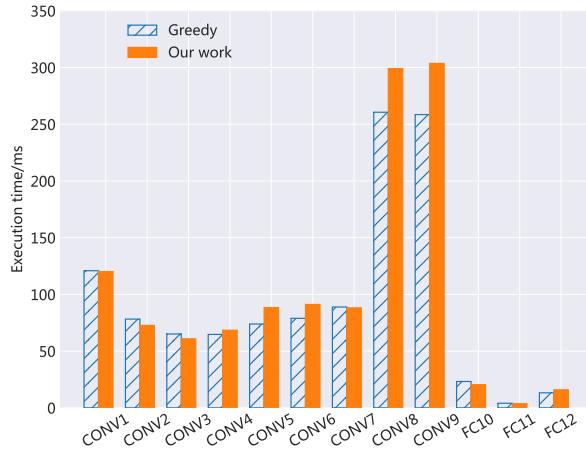


图 4.4 每一层的执行时间

**功耗和能耗：**图4.5描述了两个策略间功耗开销对比。整体上来看，贪心策略的平均功耗约为 5.44 瓦而本文所提策略的平均功耗约为 1.22 瓦。从图4.5也可以看出，对于所有层而言，本文所提策略的功耗要远低于贪心策略的功耗。这主要是因为移动设备 GPUs 在设计时更多地关注低功耗而非高性能。移动 GPUs 通常拥有着较低的运行频率，例如：ARM Mali-T628 MP6 GPU 最大运行频率为 600MHz 而 Samsung Exynos5422 CPU 的最大运行频率可达 2GHz<sup>[45]</sup>。

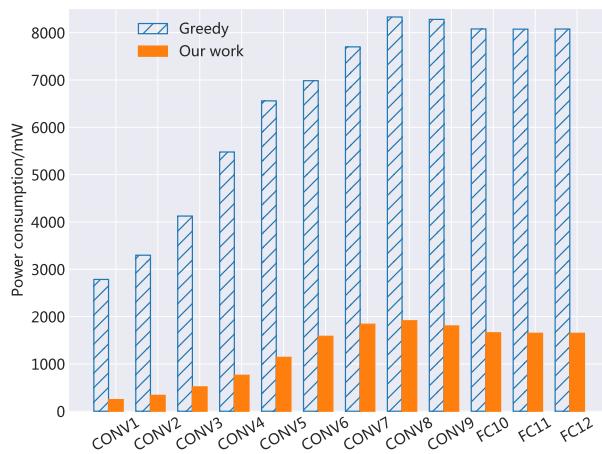


图 4.5 每一层的运行时平均功耗

在各层的运行时能耗方面，图4.6显示了一个与功耗表现类似的趋势，即贪心策略的运行时能耗在各层上都要远高于本文所提出的策略。平均来说，贪心策略的运行时能耗约为 6.59 焦，而本文所提策略的能耗约为 1.63 焦。因此，本文所提策略通过移除一个较低能效的设备处理器有效地降低了 CNN 推断时的功耗

和能耗。另一方面，从之前的性能分析可知，本文所提策略的推断执行性能损失很小。

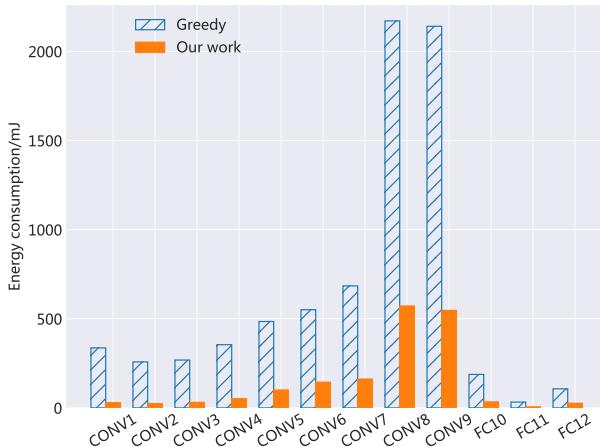


图 4.6 每一层的运行时能耗

**能效：**图4.7显示了不同异构计算组合执行一个完整 CNN 推断过程的四个方面对比，即运行时间、能耗、平均功耗和能效 EDP 值间的比较。*Single GPU* 仅使用一个 GPU 设备处理器执行 CNN 前向推断而 *Our work* 使用由本文所提策略选择的两个 GPU 设备处理器并行执行 CNN 推断。*Greedy* 则使用所有可获得的设备处理器（包含一个 CPU 和两个 GPU 设备处理器）并行执行 CNN 推断。从柱状图4.7可以看出，随着所使用设备处理器数量的增加，CNN 推断的执行时间在不断降低，而能耗和平均功耗却在逐步升高。分析 EDP 值可知，本文所提出的策略（*Our work*）拥有着更好的能效，其大约是贪心策略能效的 3.67 倍。与此同时，本文所提出的策略在推断执行速度上仅降低了 9.7%。



图 4.7 不同异构计算组合的运行时间、能耗、平均功耗和能效对比

上述实验结果证明了利用所有可获得的设备处理器可以最大化加快 CNN 推断速度，但是这种方式也可能会导致一个非常低的推断能效。然而，对于电池容

量受限的手机平台来说，除了需要关注程序的执行性能，程序执行过程中的能耗开销也是非常重要的<sup>[47]</sup>。

## 4.5 本章小结

尽管目前手机移动平台上并没有配备许多的异构设备处理器，但是在手机 SoC 上集成不同的异构处理器以执行不同的任务已经成为了一种趋势。正如 4.1 节所述，手机移动端的处理器架构一直在不断创新。例如，陈天石先生提出的 DianNao 在处理神经网络计算时执行速度是 128-bit 2GHz SIMD 处理器的 117.87 倍，并且功耗仅为 485 毫瓦<sup>[31]</sup>，其能效甚至超过了目前的手机 GPU。华为手机使用的麒麟 970 芯片即嵌入了基于 DianNao 架构的神经网络处理单元（NPU）。在这种手机 SoC 的发展趋势下，合理高效地利用多异构设备处理器执行基于深度学习模型的手机应用将会变得越来越重要。本文对基于异构计算的移动端高能效 CNN 离线推断进行了探索，并对所提出的策略于 ODROID-XU3 平台上进行了实验验证。实验结果发现，与总是试图使用所有可获得异构设备处理器的贪心策略相比，本文所提出的策略可以将能效提升 3.67 倍以上而仅造成 9.7% 的执行性能损失。

## 第5章 基于应用场景的系统层能效优化

本章首先介绍了动态电压频率调节 (DVFS) 的基本概念以及其在能效优化领域所发挥的作用。然后，利用前述章节所实现的 CNN 推断时库，本文开发了一款智能监控系统 Android 应用。5.2节对该应用的负载特征进行了刻画与分析，阐述了当前 Android 系统所采用的功耗管理器在基于深度学习模型应用上表现的不足之处。5.3节提出了一种基于应用场景的系统层能效优化策略，该策略根据应用在系统层表现的负载特征对其进行分类，这样系统便可以针对不同的应用使用不同的功耗管理策略。

### 5.1 动态电压频率调节技术

动态电压频率调节 (Dynamic Voltage and Frequency Scaling, DVFS) 是一种可对芯片电压和频率进行实时动态调节的技术。当前手机 CPU 和 GPU 都支持 DVFS，并且系统层也都存在着相应的功耗管理器 (governor)。每一个功耗管理器都拥有一个执行频率调节的策略，并且这个策略可被配置以取得不同的能效折中。根据配置的策略，功耗管理器可以决定不同状态下的设备处理器所应该运行的频率。设备处理器的负载越高运行频率越高是这些功耗管理器所遵循的基本准则。

动态功耗<sup>[55]</sup>、短路功耗<sup>[56]</sup> 和漏电流功耗<sup>[57]</sup> 是 CMOS 电路的三个主要功耗，因此 CMOS 电路的总功耗可由公式5.1表示：

$$P = P_{Dynamic} + P_{Short} + P_{Leakage} = ACV^2f + AVI_{Short} + VI_{Leakage} \quad (5.1)$$

式中  $C$  代表负载电容的容值， $V$  是工作电压， $A$  是当前频率下电路的平均翻转率， $f$  为工作频率， $I_{Short}$  和  $I_{Leakage}$  分别为短路电流和漏电流。从公式中可知， $C$ 、 $V$ 、 $A$ 、 $f$  决定了整个 CMOS 电路的功耗，而 DVFS 技术就是主要通过改变频率  $f$  和电压  $V$  的值来调节系统功耗的。

### 5.2 智能监控系统 Android 应用的负载分析

基于前述章节所实现的 CNN 推断时库，本文开发了一款智能监控系统 Android 应用，其可以通过手机摄像头自动辨识周围所观察到的物体类别。图5.1显示了两张该应用的运行界面。智能监控系统应用的智能识别功能是由卷积神经

网络 Tiny YOLO<sup>[58]</sup> 模型提供的。



图 5.1 智能监控系统 Android 应用

图5.2显示了智能监控系统 Android 应用的工作流程。在智能监控系统 APP 启动后，它会不断地通过手机摄像头感知周围的场景。当读取到摄像头所拍摄视频中的一帧数据后，该 APP 会立即将该帧数据作为 CNN 模型的输入并执行网络的前向推断过程。推断完成后，侦测结果会在视频框中显示出来。紧接着，监控 APP 会立即获取下一帧图像数据并重复上述过程。

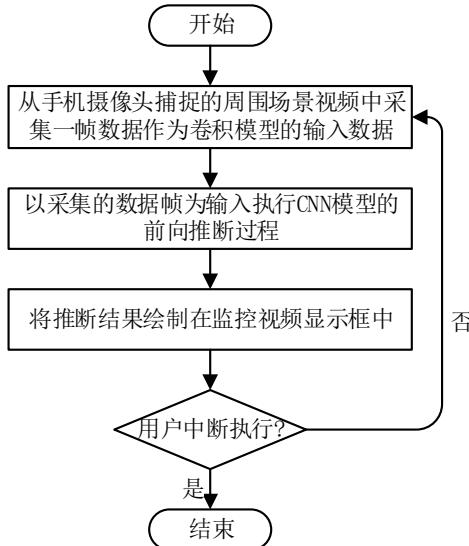


图 5.2 智能监控系统 APP 的工作流程

为了探索从系统层进一步优化基于 CNN 模型手机应用能效的策略，本文基于 ODROID-XU3 平台考察了智能监控系统 APP 的运行时负载特征。由第4章可知，智能监控系统 APP 在 ODROID-XU3 上主要使用 GPU 执行 CNN 的前向推断过程，故而可使用 GPU 的利用率作为该 APP 的负载。图5.3显示了在 Android 系统 GPU 默认功耗管理策略下智能监控系统 APP 的负载和 GPU 频率变化情况。

由图5.3的 GPU 利用率曲线可以看出，智能监控系统 APP 的负载（GPU 利用率）平均值为 86.59%，并且绝大多数的负载值都在平均线以上。智能监控系统 APP 的负载曲线形状与周期脉冲图类似，这与该 APP 的实际工作流程相符合。因为智能监控系统 APP 的工作流程就是周期性的“采样-推断-绘制”。由智能监控系统 APP 的负载曲线上只有一个很窄的波谷可知，“采样-推断-绘制”三个操作中推断占了整个 APP 运行周期的绝大部分。

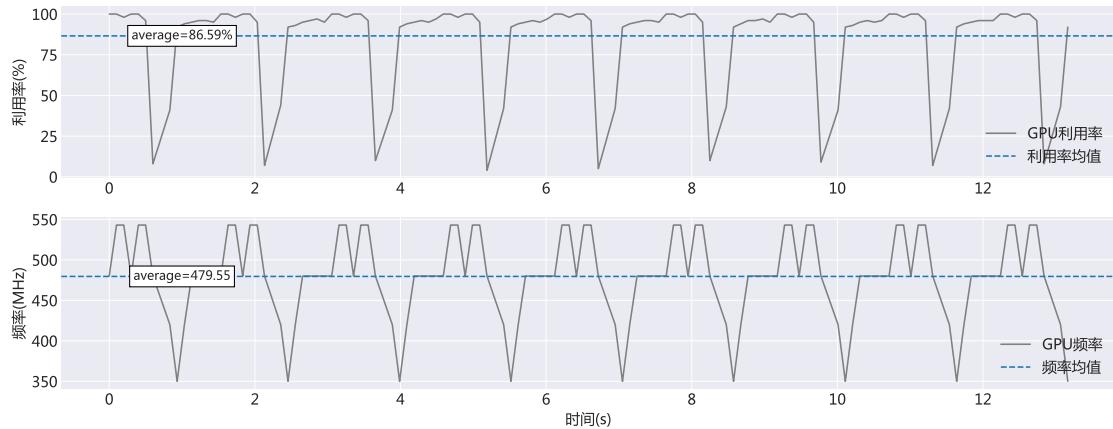


图 5.3 智能监控系统 APP 的负载和 GPU 频率变化

图5.3的 GPU 频率曲线即反映了 GPU 默认功耗管理器所采用的调频策略针对智能监控系统 APP 负载所进行的频率调节变化。由于智能监控系统 APP 负载存在一个尖脉冲波谷，所以在默认调频策略下 GPU 的频率不断在 350MHz 至 543MHz 之间抖动（均值约为 480MHz）。另外，由于频率是根据负载的变化进行调整的，所以调频具有一定的迟滞性。从图5.3中两条曲线的变化可以明显看出，当负载处于较低位置时，频率仍在较高点；当 GPU 几乎处于满负载的状态下，频率却在 480MHz 与 543MHz 间跳变，产生“乒乓效应”。频率调节的迟滞性会造成不必要的功耗开销，而“乒乓效应”不仅带来额外的调频开销还会导致上层应用的性能损失。

ODROID-XU3 平台所配备的 Mali-T628 MP6 GPU 支持六级调频，即 GPU 的运行频率可设置为 177MHz, 266MHz, 350MHz, 420MHz, 480MHz 和 543MHz 六个值。本文将 GPU 频率分别保持在这六个频率值上，考察了智能监控系统 APP 中一次 CNN 推断的运行时间、能耗和平均功耗的变化情况，结果如图5.4所示。从图中三条曲线的走势可以看出，随着 GPU 运行频率的提高，CNN 推断的运行时间在不断减小，而 GPU 的功耗和能耗在逐渐上升。图5.4中的能耗曲线在 266MHz 处上升幅度较大，并且从功耗曲线中也可以发现 266MHz 和 350MHz 处的 GPU 功耗相同，这可能是因为 Mali-T628 MP6 GPU 在 266MHz 的运行时功耗没有优化好。

图5.4中的虚线标注出了在 GPU 默认功耗管理策略下 APP 中 CNN 推断的运

行时间、能耗和平均功耗。根据图5.4显示的三条虚线位置可知，GPU保持运行在480MHz的CNN推断运行时间、能耗及平均功耗与默认功耗管理策略下的对应值十分相近。由图5.3可知，默认功耗管理策略下执行CNN推断时GPU频率均值约为480MHz，这也解释了前述两者在性能和功耗上表现相似的原因。

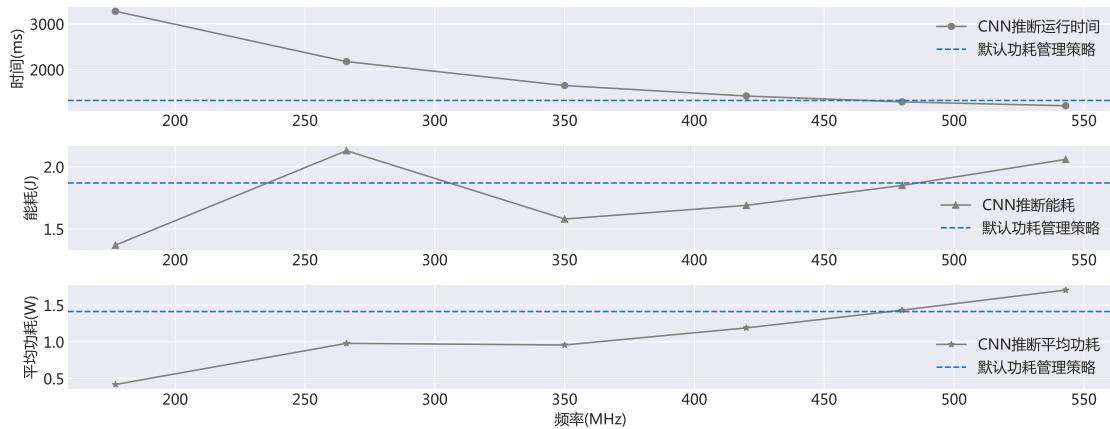


图5.4 CNN推断的运行时间、能耗和平均功耗随GPU频率的变化

表5.1进一步比较了GPU默认功耗管理策略与GPU保持在某一固定频率时执行CNN推断的能效。本文使用FPS（每秒处理图像数据的帧数）作为性能指标，并且使用EDP值评估不同策略的能效。表中的性能提升、能耗和平均功耗降低数据都是相对于默认调频策略而言的。

表5.1 智能监控系统APP在不同调频策略下的能效

策略	运行时间 (ms)	性能 (FPS)	能耗 (J)	功耗 (W)	EDP(Joules*seconds)	性能提升	能耗降低	平均功耗降低
默认调频策略	1323.83	0.76	1.87	1.41	2.48	—	—	—
固定最低频率	3277.89	0.31	1.37	0.42	4.49	-59.61%	26.74%	70.41%
固定最高频率	1207.85	0.83	2.06	1.71	2.49	9.60%	-10.16%	-20.74%
固定480MHz	1292.83	0.77	1.85	1.43	2.39	2.40%	1.07%	-1.30%

由表5.1可以看出，将GPU频率固定到480MHz时，智能监控系统APP的执行性能不仅有所提升而且其能耗也有所降低，这就证明了调频迟滞性和“乒乓效应”确实会造成不必要的能耗开销和性能损失。分析各种策略的EDP值可知，GPU频率固定在最低频率时能效最低，而使用默认调频策略、固定480MHz以及固定最高频率时能效相近。虽然固定GPU运行在480MHz时，与默认调频策略相比，能效提升了3.8%，但是其在性能上的提升量(1.07%)和功耗上的降低量(2.40%)均不太明显。而固定GPU运行在最高频率与默认调频策略的能效EDP值几乎一样，但是其在性能上会带来约10%的提升。综上所述，对于内嵌CNN模型的智能监控系统APP而言，直接将GPU频率调至最高点比使用默认GPU调频策略更佳。

### 5.3 基于应用场景的系统层能效优化策略

根据5.2节的分析可知，针对基于 CNN 模型的生活日志型 Android 应用（如本文开发的智能监控系统 APP），系统默认使用的调频策略会产生“乒乓效应”并造成额外的能耗开销，而将 GPU 频率固定在最大可达频率处 CNN 推断性能更高且几乎不影响推断能效。因此，本文提出基于应用场景的调频策略，其可自动感知系统上层运行的应用类别。具体来说，若该策略感知到系统上层运行的是基于 CNN 模型的生活日志型应用，其会将 GPU 频率调至最高点并保持不变；若该策略感知到系统上层运行的是传统型应用，其会使用 GPU 原来的默认调频策略。基于应用场景的调频策略主要根据系统上层应用的负载特征对应用进行分类，并在分类过程中使用动态时间规整计算上层应用所属的类别。

#### 5.3.1. 动态时间规整

动态时间规整 (Dynamic Time Warping, DTW)<sup>[59]</sup> 主要用于衡量两段时间序列的相似度，并且这两段时间序列的变化速度可以是不一致的。例如，语音识别中不同人的语速不同，同一个字母的发音序列会有所差别，但使用 DTW 就可以有效的区分这种同字母不同语速下的语音序列。动态时间规整主要通过将两段时间序列进行延伸和缩短来计算它们之间的相似性。如图5.5所示，序列 X 和序列 Y 表示两段时间序列，双箭头指向处代表两段序列间的对齐点。

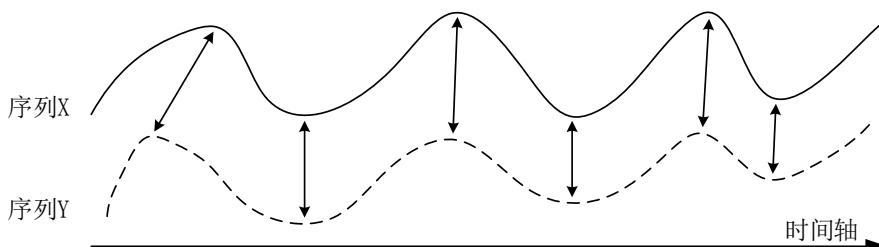


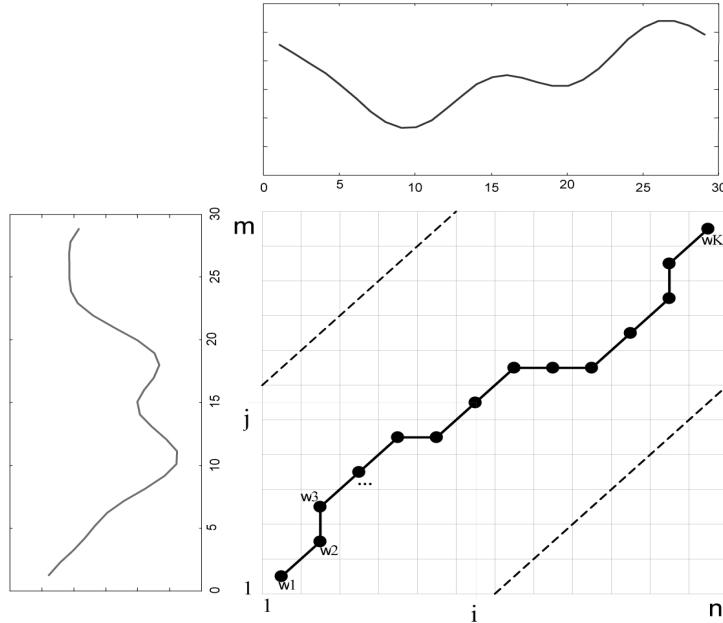
图 5.5 两段时间序列间的规整

$$D(i, j) = d(i, j) + \min\{D(i - 1, j), D(i, j - 1), D(i - 1, j - 1)\} \quad (5.2)$$

图5.6展示了求取两段时间序列间 DTW 值的一个示例，其中 DTW 的主要计算步骤如下所述：

1. 计算两段时间序列各对齐点间的距离  $d(i, j)$ ，这里可以使用欧氏距离、曼哈顿距离等。
2. 设置  $i = 0$  和  $j = 0$  处的规整距离  $D(i, j)$  为正无穷大。

3. 对于  $i = 1 \dots n$ ,  $j = 1 \dots m$ , 利用公式5.2迭代计算各个对齐点间的规整距离  $D(i, j)$ 。
4. 最后一次迭代结果  $D(n, m)$  即为所求两时间序列间的动态时间规整距离。

图 5.6 规整路径示例<sup>[60]</sup>

### 5.3.2. 基于应用场景的调频策略

如前所述, 手机移动平台内嵌 CNN 模型的生活日志型应用主要使用 GPU 执行推断过程, 所以本文针对 GPU 设备处理器提出了基于应用场景的调频策略。算法5.1给出了本文所提策略的伪代码, 其核心思想描述如下:

1. 提取基于 CNN 模型生活日志型应用的特征负载时间序列  $X(x_0, x_1, \dots, x_{N-1})$  作为模板序列。
2. 确定可对应用场景进行分类的 DTW 阈值  $DTW_{TH}$ , 其可用于识别系统上层所运行应用是否为基于 CNN 模型的生活日志型应用。
3. 每次进入调频函数, 在长度为 M 的数组 `load_infos` 中记录当前获得的负载值, 并使计数器值加 1。注意, M 取值应使数组 `load_infos` 能够覆盖上层应用的负载特征, 其可由用户输入也可取一个较大值。
4. 当计数器值等于 M-1 时, 计算已获取的 M 个历史负载序列 `load_infos` 和模板序列 X 之间的 DTW 值 `dtw_dist`。

5. 当  $dtw\_dist$  小于  $DTW_{TH}$  时, 说明系统上层所运行应用为基于 CNN 模型的生活日志型应用, 此时应将 GPU 频率设置到最高(已最高则不变); 反之, 说明系统上层所运行应用为非基于 CNN 模型的其他应用, 此时应采用 GPU 默认的调频策略。

**Data:** 基于 CNN 模型的生活日志型应用负载模板序列  $X(x_0, x_1, \dots, x_{N-1})$ ;

应用场景分类 DTW 阈值  $DTW_{TH}$ 。

```

1 令函数 dtw(X,Y) 的功能为计算两段时间序列 X 和 Y 间的 DTW 值;
2 初始化 count = 0; //计数已获取的负载值数量, 仅全局初始化一次
3 初始化 load_infos[M] = {0}; //存储最近获取的 M 个负载值, 仅全局初始化一次
4 初始化 dtw_dist = 2DTW_{TH}; //仅全局初始化一次
5 if count < M - 1 then
6   | load_infos[count] = 当前 GPU 利用率 (即上层应用负载);
7   | count++;
8 else
9   | load_infos[count] = 当前 GPU 利用率 (即上层应用负载);
10  | count = 0;
11  | dtw_dist = dtw(X,load_infos);
12 end
13 if dtw_dist < DTW_{TH} then
14   | 将 GPU 频率调至最高一级, 已最高则不变; //基于 CNN 模型的生活日志型应用
15 else
16   | 使用默认调频策略; //非基于 CNN 模型的其他应用
17 end

```

算法 5.1: 基于应用场景的调频策略

## 5.4 实验验证

本文利用 ODROID-XU3 平台实现了基于应用场景的调频策略, 并使用所开发的智能监控系统 Android 应用对所提调频策略的有效性进行了验证。

根据图5.3显示的智能监控系统 Android 应用的负载(GPU 利用率)曲线可知, 包含完整 CNN 推断的负载序列时间长度小于 2 秒。因此, 本文从负载曲线中截取时间长度约 4 秒的序列作为基于 CNN 模型的生活日志型应用负载模板序列, 该序列总共包括 40 个负载值。为了确定可对不同应用场景进行分类的 DTW 阈值, 本文分别考察了包括 CNN 推断在内的手机系统可能存在的几个应用场景下应用负载序列与模板序列间的 DTW 值, 结果如图5.7所示。根据图5.7显示的结果可知, 当 DTW 阈值取 480 时, 基于 CNN 模型的生活日志型应用和非基于 CNN

模型的其他应用可以被很好地区分开来。实验中，针对智能监控系统 Android 应用，调频策略中的 M 参数只要取大于 20 的数字即可覆盖监控系统 APP 的负载特征。

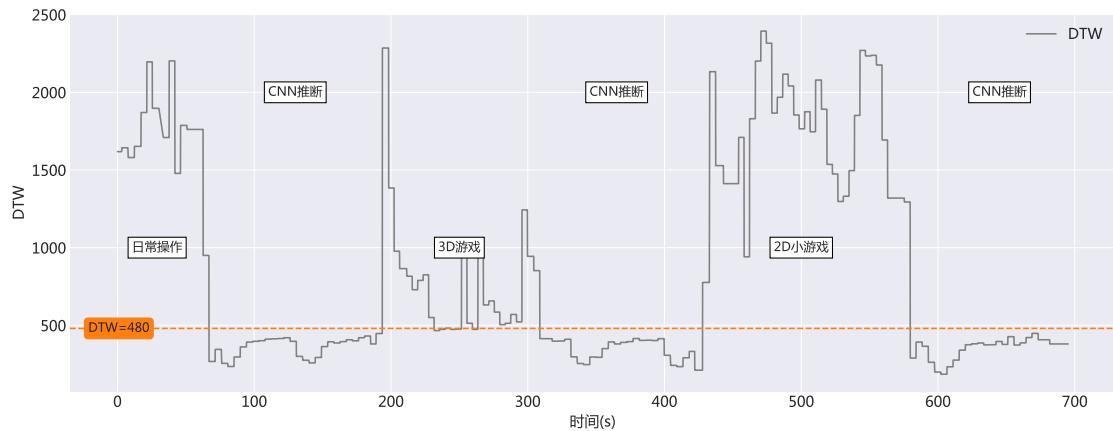


图 5.7 应用场景分类 DTW 阈值的确定

图5.8显示了基于应用场景的调频策略运行效果。由 GPU 的频率变化曲线可以看出，在手机系统处于日常操作（如查看邮件、联系人等）、3D 游戏、2D 游戏等应用场景下时，系统 GPU 功耗管理器使用的均为默认策略，而一旦系统检测到上层运行的应用为基于 CNN 模型的生活日志型应用时，很快会将 GPU 频率调至最高点并保持运行在最高频率。

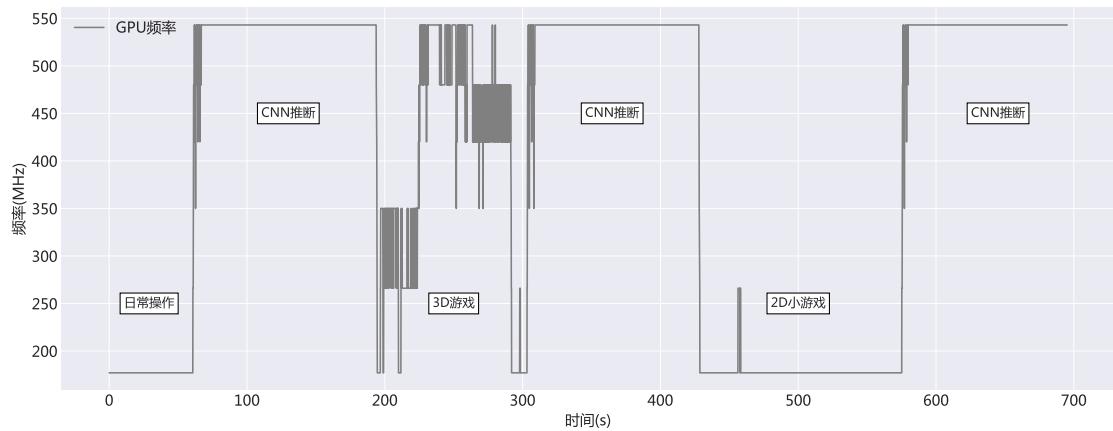


图 5.8 基于应用场景的调频策略运行效果

为了分析基于应用场景的调频策略对系统功耗的影响，本文比较了分别于所提调频策略和默认调频策略下运行 CNN 推断时系统 CPU 功耗的变化。之所以仅比较 CPU 功耗，是因为系统功耗管理器都是运行在 CPU 上的。由表5.2可以看出，在默认调频策略和本文所提调频策略下，运行 CNN 推断时 CPU 的平均功耗分别为 0.71W 和 0.72W。由此得出结论：在功耗测量误差范围内，基于应用场景的调频策略对系统功耗的影响可以忽略不计。

表 5.2 CNN 推断过程中不同调频策略下 CPU 的平均功耗

策略	CPU 平均功耗 (W)
默认调频策略	0.71
基于应用场景的调频策略	0.72

## 5.5 本章小结

本章利用前述章节所实现的 CNN 推断时库开发了一款智能监控系统 Android 应用，并针对该应用详细分析了其在系统层所表现的负载特征。通过对比分析智能监控系统应用分别于 GPU 默认调频策略和固定频率策略下运行时的性能和能效，本文发现系统默认使用的调频策略会产生“乒乓效应”并造成额外的能耗开销，而将 GPU 频率固定在最大可达频率点 CNN 推断性能更高且几乎不影响推断能效。因此，本章的最后提出了一种基于应用场景的调频策略，其可自动感知系统上层运行的应用是否为基于 CNN 模型的生活日志型应用，并根据检测结果及时改变调频行为。本文使用 ODROID-XU3 平台验证了所提调频策略的有效性，并分析了该策略对系统功耗的影响。



## 第6章 总结与展望

### 6.1 本文工作总结

当今社会，移动手机已然成为人类生活中不可或缺的贴身物品，并且用户对其智能化程度要求也越来越高。为此，许多手机生产商将人工智能技术应用于手机平台，借以为使用者提供更加人性化的服务，如苹果 iPhone X、华为 Mate 10 系列等。卷积神经网络（CNNs）是人工智能领域中一种成熟的神经网络模型，其已经被应用于手机平台为人类解决生活中遇到的诸多问题，如人脸识别、机器翻译等。为了更好地保障手机用户的隐私并避免网络性能的不稳定，基于人工智能模型的手机应用愈来愈偏向于使用手机本地设备处理器完成前向推断过程而非上传到云端服务器执行。本文针对 CNN 模型于 Android 手机平台进行离线推断的过程提出了一系列能效优化策略，主要工作包括如下：

1. 基于 OpenCL 异构编程框架设计与实现了一套可在手机 GPU 和 CPU 上执行 CNN 前向推断的运行时库。利用该套 CNN 推断时库，本文在手机移动平台上分别重构了 LeNet-5 模型和 AlexNet 模型。实验发现，LeNet-5 模型在手机 GPU 上进行前向推断的执行速度是其在手机 CPU 上的 11 倍以上，并且能耗也仅为使用 CPU 执行推断时的 1/120。而对于 AlexNet 这种复杂度较高的模型，其在手机 GPU 上执行前向推断相较于手机 CPU 而言性能加速比可达 15，且推断能耗可降为使用 CPU 推断的 1/30。
2. 基于“剪枝-重训”的模型压缩方法对卷积神经网络中占存储量主要部分的全连接层权重进行压缩，并在 CNN 推断时库中引入稀疏矩阵向量乘（SpMV）使得推断时库支持经压缩处理的稀疏 CNN 模型。实验证明，对网络模型进行压缩不仅可以降低手机内存占用开销、提高模型加载速度，还可以在一定程度上加速模型于移动端的推断过程。
3. 结合移动端 SoC 架构的发展趋势，本文提出了一种基于手机平台异构设备处理器高能效并行执行 CNN 推断的优化策略。该策略使得 CNN 推断时库在运行中可自动感知目标平台上不同异构设备处理器的推断能效，并根据这些能效差异自适应地找到一个可高能效执行 CNN 推断的设备处理器组合。实验结果发现，与总是试图使用所有可获得异构设备处理器的贪心策略相比，本文所提出的策略可以获得 3.67 倍更高的能效且仅仅损失 9.7% 的执行速度。
4. 通过分析智能监控系统 Android 应用这种基于 CNN 模型生活日志型 APP

的系统层负载特征，本文发现系统默认使用的 GPU 调频策略会产生“乒乓效应”并造成额外的能耗开销，而将 GPU 频率固定在最大可达频率点时 CNN 推断性能更高且几乎不影响推断能效。因此，本文提出了一种基于应用场景的 GPU 调频策略，其可自动感知系统上层应用是否为基于 CNN 模型的生活日志型应用，并根据检测结果及时改变调频行为。

## 6.2 未来工作展望

本文针对基于 CNN 模型的手机应用如何在移动平台上高能效执行这一研究问题，提出了一系列能效优化策略，包括结合模型压缩的手机 GPU 加速推断、使用异构设备处理器并行执行推断以及根据应用场景特点进行系统层优化等。然而，为了使 CNN 模型广泛地应用于商业手机平台，在本文研究内容的基础上仍然存在着许多待优化的工作。

首先，本文实验发现，手机平台的访存速度普遍较低，所以模型体积较大时其推断速度受访存影响较为明显。在以后的研究工作中，针对手机 GPU 上的 CNN 前向推断过程，可以使用半浮点型数据代替浮点型数据。虽然这种做法可能会造成些许的推断精度损失，但是其不仅能够降低内存占用和访存能耗还可以在一定程度上提高推断执行速度。另外，若手机 GPU 存在着本地内存 (local memory)，应尽量使用本地内存代替全局内存以加快访存速度。本文研究工作仅压缩了卷积模型中占存储量主要部分的全连接层权重，而未来将会对卷积层权重进行压缩并探索为压缩后的卷积层提供移动端运行支持的方法。

其次，本文提出的基于异构设备处理器高能效并行执行 CNN 推断的优化策略中没有考虑任务划分所导致的数据通信开销，这将被加入到未来的研究工作计划中。另外，研究中仅按输出节点对单层的计算任务进行划分，在今后的研究工作中将进一步探索更细粒度的计算任务划分方法。

最后，本文对 CNN 模型推断在系统层进行能效优化的探索是初步的，它为以后的研究工作提供了手机系统层优化 CNN 推断的思路。针对基于 CNN 模型的手机应用，未来的工作将探索更加通用、智能的系统功耗管理器。

## 参 考 文 献

- [1] LECUN Y, BENGIO Y, HINTON G. Deep learning[J]. Nature, 2015, 521(7553): 436.
- [2] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep convolutional neural networks[C]//Advances in neural information processing systems. 2012: 1097–1105.
- [3] SIMONYAN K, ZISSERMAN A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.
- [4] LANE N D, GEORGIEV P. Can deep learning revolutionize mobile sensing?[C]//Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications. [S.l.]: ACM, 2015: 117–122.
- [5] DENG L, HINTON G, KINGSBURY B. New types of deep neural network learning for speech recognition and related applications: An overview[C]//Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on. [S.l.]: IEEE, 2013: 8599–8603.
- [6] LANE N D, BHATTACHARYA S, GEORGIEV P, et al. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices[C]//Proceedings of the 2015 International Workshop on Internet of Things towards Applications. [S.l.]: ACM, 2015: 7–12.
- [7] YANAI K, TANNO R, OKAMOTO K. Efficient mobile implementation of a cnn-based object recognition system[C]//Proceedings of the 2016 ACM on Multimedia Conference. [S.l.]: ACM, 2016: 362–366.
- [8] LIN M, CHEN Q, YAN S. Network in network[J]. arXiv preprint arXiv:1312.4400, 2013.
- [9] MITRA G, JOHNSTON B, RENDELL A P, et al. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms[C]//Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International. [S.l.]: IEEE, 2013: 1107–1116.
- [10] DENTON E L, ZAREMBA W, BRUNA J, et al. Exploiting linear structure within convolutional networks for efficient evaluation[C]//Advances in neural information processing systems. 2014: 1269–1277.
- [11] JADERBERG M, VEDALDI A, ZISSERMAN A. Speeding up convolutional neural networks with low rank expansions[J]. arXiv preprint arXiv:1405.3866, 2014.
- [12] LEBEDEV V, GANIN Y, RAKHUBA M, et al. Speeding-up convolutional neural networks using fine-tuned cp-decomposition[J]. arXiv preprint arXiv:1412.6553, 2014.

- 
- [13] WU J, LENG C, WANG Y, et al. Quantized convolutional neural networks for mobile devices [C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 4820–4828.
  - [14] WANG P, CHENG J. Accelerating convolutional neural networks for mobile applications[C]// Proceedings of the 2016 ACM on Multimedia Conference. [S.l.]: ACM, 2016: 541–545.
  - [15] LOKHMOTOV A, FURSIN G. Optimizing convolutional neural networks on embedded platforms with opencl[C]//Proceedings of the 4th International Workshop on OpenCL. [S.l.]: ACM, 2016: 10.
  - [16] XIANYI Z, QIAN W, CHOTHIA Z. Openblas[J]. URL: <http://xianyi.github.io/OpenBLAS>, 2012: 88.
  - [17] JIA Y, SHELHAMER E, DONAHUE J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. [S.l.]: ACM, 2014: 675–678.
  - [18] RUPP K, RUDOLF F, WEINBUB J. Viennacl-a high level linear algebra library for gpus and multi-core cpus[C]//Intl. Workshop on GPUs and Scientific Applications. 2010: 51–56.
  - [19] NUGTEREN C. Clblast: A tuned opencl blas library[J]. arXiv preprint arXiv:1705.05249, 2017.
  - [20] STONE J E, GOHARA D, SHI G. Opencl: A parallel programming standard for heterogeneous computing systems[J]. Computing in science & engineering, 2010, 12(3): 66–73.
  - [21] LANE N D, BHATTACHARYA S, GEORGIEV P, et al. Deepx: A software accelerator for low-power deep learning inference on mobile devices[C]//Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on. [S.l.]: IEEE, 2016: 1–12.
  - [22] HUYNH L N, BALAN R K, LEE Y. Deepsense: A gpu-based deep convolutional neural network framework on commodity mobile devices[C]//Proceedings of the 2016 Workshop on Wearable Systems and Applications. [S.l.]: ACM, 2016: 25–30.
  - [23] LATIFI OSKOUEI S S, GOLESTANI H, HASHEMI M, et al. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android[C]//Proceedings of the 2016 ACM on Multimedia Conference. [S.l.]: ACM, 2016: 1201–1205.
  - [24] GUIHOT H. Renderscript[M]//Pro Android Apps Performance Optimization. [S.l.]: Springer, 2012: 231–263.
  - [25] LANE N D, GEORGIEV P, QENDRO L. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning[C]//Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing. [S.l.]: ACM, 2015: 283–294.

- 
- [26] CHEN G, PARADA C, HEIGOLD G. Small-footprint keyword spotting using deep neural networks[C]//Acoustics, speech and signal processing (icassp), 2014 ieee international conference on. [S.l.]: IEEE, 2014: 4087–4091.
  - [27] VARIANI E, LEI X, MCDERMOTT E, et al. Deep neural networks for small footprint text-dependent speaker verification[C]//Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. [S.l.]: IEEE, 2014: 4052–4056.
  - [28] LU H, BRUSH A B, PRIYANTHA B, et al. Speakersense: Energy efficient unobtrusive speaker identification on mobile phones[C]//International conference on pervasive computing. [S.l.]: Springer, 2011: 188–205.
  - [29] GEORGIEV P, LANE N D, RACHURI K K, et al. Dsp. ear: Leveraging co-processor support for continuous audio sensing on smartphones[C]//Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems. [S.l.]: ACM, 2014: 295–309.
  - [30] ANTONIOU A, ANGELOV P. A general purpose intelligent surveillance system for mobile devices using deep learning[C]//Neural Networks (IJCNN), 2016 International Joint Conference on. [S.l.]: IEEE, 2016: 2879–2886.
  - [31] CHEN T, DU Z, SUN N, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning[J]. ACM Sigplan Notices, 2014, 49(4): 269–284.
  - [32] ZHANG C, LI P, SUN G, et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks[C]//Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. [S.l.]: ACM, 2015: 161–170.
  - [33] WANG C, GONG L, YU Q, et al. Dlau: A scalable deep learning accelerator unit on FPGA [J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 36(3): 513–517.
  - [34] YU Q, WANG C, MA X, et al. A deep learning prediction process accelerator based FPGA [C]//Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on. [S.l.]: IEEE, 2015: 1159–1162.
  - [35] WANG C, LI X, YU Q, et al. Solar: Services-oriented learning architectures[C]//Web Services (ICWS), 2016 IEEE International Conference on. [S.l.]: IEEE, 2016: 662–665.
  - [36] REDMON J, DIVVALA S, GIRSHICK R, et al. You only look once: Unified, real-time object detection[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 779–788.
  - [37] ABADI M, AGARWAL A, BARHAM P, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems[J]. arXiv preprint arXiv:1603.04467, 2016.
  - [38] LE SUEUR E, HEISER G. Dynamic voltage and frequency scaling: The laws of diminishing returns[C]//Proceedings of the 2010 international conference on Power aware computing and

- systems. 2010: 1–8.
- [39] LECUN Y, BOTTOU L, BENGIO Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278–2324.
- [40] WANG Z, LI X, WANG C, et al. Rethinking energy-efficiency of heterogeneous computing for cnn-based mobile applications[C]//ISPA, 2017 IEEE. [S.l.]: IEEE, 2017: 454–458.
- [41] BOTTOU L. Large-scale machine learning with stochastic gradient descent[M]//Proceedings of COMPSTAT'2010. [S.l.]: Springer, 2010: 177–186.
- [42] SCHALKOFF R J. Artificial neural networks: volume 1[M]. [S.l.]: McGraw-Hill New York, 1997.
- [43] DING W G. Draw Convnet. [https://github.com/gwding/draw\\_convnet](https://github.com/gwding/draw_convnet)[Z].
- [44] HOROWITZ M, INDERMAUR T, GONZALEZ R. Low-power digital design[C]//Low Power Electronics, Digest of Technical Papers, 1994 IEEE Symposium. [S.l.]: IEEE, 1994: 8–11.
- [45] HARDKERNEL. ODROID-XU3. [http://www.hardkernel.com/main/products/prdt\\_info.php?g\\_code=g140448267127](http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127)[Z].
- [46] INTRINSYC. Open-Q™ 820 Development Kit. <https://www.intrinsyc.com/snapdragon-embedded-development-kits/snapdragon-820-development-kit>[Z].
- [47] BROOKS D M, BOSE P, SCHUSTER S E, et al. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors[J]. IEEE Micro, 2000, 20(6): 26–44.
- [48] WANG Z, CHENG Z, LI X, et al. Building a game benchmark for cooperative cpu-gpu with pseudo user-interaction[C]//ISPA, 2017 IEEE. [S.l.]: IEEE, 2017: 574–581.
- [49] CHUNG H, KANG M, CHO H D. Heterogeneous multi-processing solution of exynos 5 octa with arm® big. little™ technology[J]. Samsung White Paper, 2012.
- [50] LOKHMOTOV A. Gemmbench: a framework for reproducible and collaborative benchmarking of matrix multiplication[J]. arXiv preprint arXiv:1511.03742, 2015.
- [51] MNIST. <http://yann.lecun.com/exdb/mnist>[Z].
- [52] HAN S, POOL J, TRAN J, et al. Learning both weights and connections for efficient neural network[C]//Advances in neural information processing systems. 2015: 1135–1143.
- [53] IMAGENET. <http://www.image-net.org>[Z].
- [54] ATTIA K M, EL-HOSSEINI M A, ALI H A. Dynamic power management techniques in multi-core architectures: A survey study[J]. Ain Shams Engineering Journal, 2015.
- [55] BENINI L, BOGLIOLI A, PALEOLOGO G A, et al. Policy optimization for dynamic power management[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1999, 18(6): 813–833.
- [56] 周宽久, 迟宗正, 西方. 嵌入式软硬件低功耗优化研究综述[J]. 计算机应用研究, 2010,

- 27(2).
- [57] YOU Y P, LEE C, LEE J K. Compilers for leakage power reduction[J]. ACM Transactions on Design Automation of Electronic Systems (TODAES), 2006, 11(1): 147–164.
- [58] TINY-YOLO. <https://pjreddie.com/darknet/yolov2>[Z].
- [59] MÜLLER M. Dynamic time warping[J]. Information retrieval for music and motion, 2007: 69–84.
- [60] KEOGH E J, PAZZANI M J. Derivative dynamic time warping[C]//Proceedings of the 2001 SIAM International Conference on Data Mining. [S.l.]: SIAM, 2001: 1–11.



## 致 谢

时光荏苒，三年的研究生学习生涯已悄然接近尾声。三年前的我追随自己内心的兴趣所在毅然选择从其他专业跨入到计算机科学这个领域，三年后的我依然不后悔当初的选择。这三年来，我完成了从对科研的陌生、迷茫到乐在其中的蜕变。期间，我收获到的不仅仅是知识还有道不尽的友情、师生情，而我需要感谢的人也有很多。

首先，我很庆幸三年的学习生活中有着四位导师对我悉心教导。感谢周学海老师，是他将我带入了计算机领域科学的研究的大门。周老师为人和蔼可亲，很少见到他发脾气，但是当我们科研不努力时他也会恨铁不成钢地批评我们。正是因为周老师对学生的严格要求，我学会了主动阅读与自己研究工作相关的论文、认真解决科研中遇到的问题。感谢李曦老师，是他每次在我研究工作遇到瓶颈停滞不前时，及时给予我解决问题的思路。李老师对于科研工作一丝不苟的态度一直是我学习的榜样。虽然李老师平时表现得严厉，但是他其实也很幽默，对待学生更是关爱有加。感谢王超老师在研究工作和论文书写上给我提出的建议。超哥每次对我科研工作的点评都是一针见血，让我及时意识到研究过程中存在的问题。感谢陈香兰老师在科研和学习生活上对我的帮助。在科研或生活上遇到解不开的难题时，陈老师的一番话总能让我恍然大悟，给予我思路和信心去战胜困难。

其次，我要感谢程志南师兄、宋家臣师兄、周金红师兄、徐友军师兄以及赵洋洋师姐。程志南师兄是我在研究生学习生涯中遇到的贵人。从开题选择到小论文写作，对于我每次的叨扰，南哥都会很有耐心地和我讨论，给我指明方向。感谢宋家臣师兄深夜还在和我探讨我的开题工作，他有条不紊的做事风格十分值得我学习。感谢周金红师兄在我研究生选题时给予我的一些建设性意见。感谢徐友军师兄带领我学习 Linux 内核，为我以后的科研工作打下了技术基础。感谢赵洋洋师姐给予我科研问题上的一些解决思路。当然，我还要感谢我的同窗好友张奕玮、罗海钊、徐冲冲、孙凡、鲁云涛、郭玲等人，是他们陪我度过了整个研究生学习生活，是他们在我遇到困难时给予我鼓励，是他们让我在科研道路上走得更远。

最后，我要感谢我的爸爸和妈妈。他们虽然都年过半百，但是对于我的求学之路总是无条件支持。我很愧对我的父母，五十几岁本该是他们安享晚年的年龄，但是因为我还在读书，他们仍在不辞劳苦地工作着。祝愿我的父母身体健康，希望我所学之知识能够报答他们的养育之恩。

王震  
于 2018 年 4 月



## 在读期间发表的学术论文与取得的研究成果

### 已发表论文

1. Zhen Wang, Xi Li, Chao Wang, Zhinan Cheng, Jiachen Song and Xuehai Zhou. Rethinking Energy-Efficiency of Heterogeneous Computing for CNN-Based Mobile Applications[C]//International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2017 IEEE 15th International Conference on. IEEE, 2017: 454-458. (CCF RANK C, ISBN: 978-1-5386-3790-6)
2. Zhen Wang, Zhinan Cheng, Xi Li, Chao Wang, Xianglan Chen and Xuehai Zhou. Building A Game Benchmark for Cooperative CPU-GPU with Pseudo User-interaction[C]//International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2017 IEEE 15th International Conference on. IEEE, 2017: 574-581. (CCF RANK C, ISBN: 978-1-5386-3790-6)

### 参与的主要项目

1. 国家自然科学基金：基于任务行为特征分析的热敏感操作系统技术研究 (No. 61272131)
2. 国家自然科学基金：面向服务的异构多核可重构片上系统任务自动并行化机制 (No. 61202053)
3. 国家自然科学基金：异构多核可重构计算平台上面向服务的操作系统关键技术 (No. 61379040)