北京大学

本科学位论文

题目：**SoCaffe: 基于 Zynq SoC 平台的高性能深度学习框架**

姓　　名：赵睿哲

学　　号：1200012778

院　　系：信息科学技术学院

专　　业：计算机科学与技术

研究方向：计算机系统结构

导　　师：梁云

2016 年 5 月 27 日

版权声明

摘要

pkuthss 文档模版最常见问题：

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

关键词：Zynq, 深度学习, SDSoC, FPGA, 高层次综合, Caffe

SoCaffe: High-Performance Deep Learning Framework on Zynq SoC

Ruizhe Zhao (Computer Science and Technology)
Directed by Prof. Yun Liang

ABSTRACT

Test of the English abstract.

KEYWORDS: Zynq, Deep Learning, SDSoC, FPGA, High-Level Synthesis, Caffe

目录

序言

pkuthss 文档模版最常见问题：

\cite、\parencite 和 \supercite 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 texdoc pkuthss“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

第一章　背景知识

本研究的主要内容是基于 Zynq SoC 平台实现高性能的深度学习框架，因此涉及到三个领域的基本内容：深度学习与神经网络，系统芯片（System-on-Chip，简称 SoC），以及可编程逻辑门阵列（Field Programmable Gate Array，简称 FPGA）和高层次综合（High-Level Synthesis，简称 HLS）。本章分别介绍上述三个领域的基本原理，并选择与本研究密切相关的细分方向进行具体分析。

1.1　深度学习与神经网络

深度学习属于机器学习领域的一个新兴分支。机器学习是实现人工智能的一种方法，简单而言，机器学习往往从大规模的历史数据中学习规律，进而对新的样本进行分类（classification）和预测（prediction）。传统的机器学习算法需要人工指定学习规则，比如从样本中提取哪类特征（feature）、用哪种统计模型进行训练等等。此类方法虽然直观，但需要大量的工程和领域（domain）相关的知识才能达到不错的效果。面对人类认知范围之外的模型时，往往特征提取和模型训练的效果都不尽如人意，有时甚至完全提取不出合适的特征。

深度学习系统的设计是革命性的：特征不再是由专家设计，而是交给通用的算法来自动学习。具体而言，深度学习将特征组织成从低到高的多层堆叠的特征层，使用反向传播算法（Back-propagation，简称 BP）提取特征。深度学习与人脑的工作原理密切相关，堆叠连接的特征层本质上是对神经元的模拟。从这个角度看，深度学习与神经网络是密不可分的：深度学习中的特征层和反向传播算法都来自神经网络。本节接下来首先介绍深度学习的基本概念与基本流程，之后介绍神经网络的几种网络层的特性，其中着重介绍卷积神经网络。

1.1.1　深度学习

2015 年 5 月，深度学习领域的三位代表人物 Yann LeCun①，Yoshua Bengio②与 Geoffrey Hinton③在《自然》杂志发表封面文章《Deep Learning》[5]，介绍深度学习的基本概念与发展现状。文章将深度学习归纳为表征学习（Representation Learning）[1] 的一类方法。表征学习即学习如何从原始数据中提取出可以用来训练与学习的特征的过程，

① Yann LeCun 目前为纽约大学教授，Facebook 人工智能研究院负责人
② Yoshua Bengio 为加拿大蒙特利尔大学教授
③ Geoffrey Hinton 为加拿大多伦多大学教授，并且任职与 Google

深度学习在此之上使用多层表征，每层的特征与上层相比更抽象。此外，每一层都获取上层的输出特征作为输入，进行非线性的转换之后输出到下一层。每层的转换方程都有特定的权值，显著的权值可以放大某些输入特征的“影响”，相应地不显著的权值则“隐藏”了无用的输入特征。深度学习应用经过训练之后，得到的权值可以使得每层提取出最合理的特征。由此看出，深度学习几乎不依赖人类的先验知识，给定特征层的结构和训练数据集之后便可以学习到最合适的特征形式。接下来简单介绍深度学习的训练过程。

所谓训练，即通过历史数据修正深度学习各层中的权值。深度学习的训练有三个基本概念：训练集，损失函数（Loss function）与随机梯度下降（Stochastic Gradient Descendent，简称 SGD）。训练集由大规模的输入数据与预期的输出结果组成，比如图像识别应用往往使用标记好类型的图片集作为训练集。训练集的好坏很大程度上决定了训练结果的好坏。损失函数衡量预期结果与当前系统预测结果的区别，衡量了当前训练阶段的成果，给下一步对权值的调整方向以反馈信息。随机梯度下降根据损失函数相对于权值在小范围内的梯度对权值进行调整。训练集、损失函数与随机梯度下降构成了完整的深度学习训练过程。使用训练集并进行训练的深度学习也称之为有监督学习（Supervised Learning）。

1.1.2 神经网络

神经网络与深度学习不是一类概念，深度学习是一种机器学习的方法，而神经网络则是从生物神经元的构成和连接启发的一种计算过程或组织形式。深度学习侧重方法，神经网络偏重结构。深度学习定义了分层的表征学习结构，而神经网络提供了更具体的计算细节。神经网络起源于 19 世纪 60 年代的感知器（Perceptron）模型，它抽象了生物的神经元：神经元接受一组信号的输入，通过权值加权、激活函数等等的计算得到结果。神经网络将神经元组织成大规模的网络结构，相同类型的神经元组成神经网络层，提供特殊形式的计算。这种分层的神经网络结构与深度学习不谋而合。

深度学习中提到的随机梯度下降计算利用了反向传播算法。反向传播算法于 19 世纪 80 年代左右发明，主要用来推导多层网络结构中每一层的梯度，进而实现随机梯度下降计算。反向传播算法利用复合函数求导的“链式法则”（chain rule），结合网络的分层结构，可以根据网络顶部的输出结果变化反向传播得到每层输入权值的梯度。反向传播算法也因此成为深度学习训练的基础算法。与反向传播相反，前向传播（Forward propagation）是从神经网络的底层逐层计算到顶层输出结果，反向传播用来求梯度，前向传播则用来预测。

神经网络有多种网络层，有些网络层由激活函数构成：比如常用的线性整流层（Rec-

tified Linear Unit，简称 ReLU，$f(x) = max(x, 0)$），tanh 层（$f(x) = tanh(x)$）、Sigmoid 层（$f(x) = \frac{1}{1-exp(x)}$）等等；有些网络层主要是计算，比如卷积层（Convolution Layer）等等。卷积层使用一系列矩阵块（也称为卷积核）对输入值进行卷积计算。神经网络根据组成的网络层以及它们的堆叠方式不同而分类为不同的神经网络类型，卷积神经网络（Convolutional Neural Network，简称 ConvNet 或 CNN）是其中最常用的网络结构之一。

卷积神经网络的设计来源于生物的视网膜工作原理，网络结构主要由卷积层、池化层（pooling layer）和线性整流层构成：卷积层进行卷积计算提取特征（feature map），池化层按照划分好的区域过滤降维，一般求最大值（max pooling）或均值（mean pooling）。卷积神经网络广泛使用于图像识别、视频分析以至于围棋比赛中，并取得了很好的效果。除了卷积神经网络以外，递归神经网络（Reccurrent Neural Network，简称 RNN）在需要“记忆”和上下文信息的学习过程中也起到很大作用，比如文本生成和自然语言处理等。

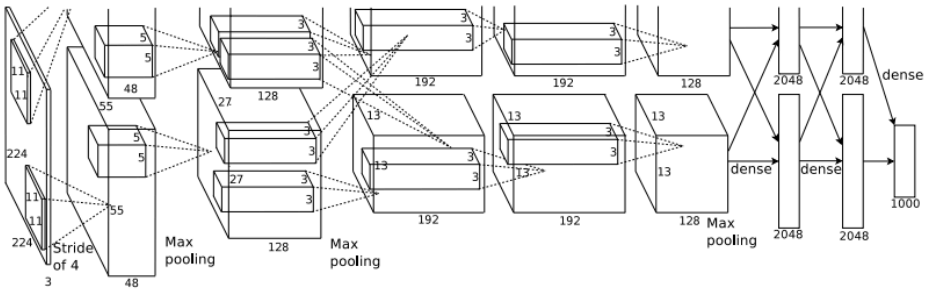

图 1.1　ImageNet LSVRC-2012 比赛中使用的大规模卷积神经网络结构[4]

综上所述，深度学习是机器学习的全新分支，挑战了传统的机器学习方法论。神经网络通过仿生的神经元与网络设计，具象化了深度学习的特征层与堆叠方式。此外，深度学习涉及到的计算，比如卷积神经网络中的卷积层等等，都十分耗费硬件的计算资源和存储资源。本研究的主要目标即为将传统的、部署于大规模 CPU 和 GPU 的深度学习应用，运行于硬件资源和计算能力都十分有限的嵌入式平台上。接下来会对本研究涉及的硬件系统进行详细的介绍。

1.2　系统芯片（SoC）

系统芯片（System-on-Chip，简称 SoC）是一类集成电路，其上包含了处理器芯片和其他电子系统，以及丰富的外部接口和设备等等。SoC 主要被电子工程师用来进行系统开发和验证：SoC 的处理器性能足够强大，一般足以运行常见的操作系统；SoC 一

般也搭载了可编程的 FPGA 硬件系统，可以非常方便修改硬件系统的设计。因此 SoC 可以明显缩短电子产品的开发周期。

SoC 的设计分为软件与硬件两部分。软件部分的开发主要涉及偏底层的硬件驱动、数据传输、硬件控制与基于操作系统的高层次软件等等。硬件系统的设计则更加复杂：工程师一方面开发基于 FPGA 的硬件逻辑，定制系统所要求的特殊功能；另一方面也尽可能基于预定义的、标准的 IP 核（全称知识产权核，Intellectual Property core）进行基础平台的搭建，提高开发效率。此外，硬件与软件不同，在开发完成之后需要进行各种系统级、行为级的功能验证才能正式完成。正因为 SoC 开发难度大，目前在开发工具上和平台设计上都有相应的进展，本研究所使用的 Xilinx Zynq SoC 平台与 SDSoC 开发工具便是例证。

本研究的目标是将深度学习框架部署于嵌入式平台中。嵌入式设备指的是一类针对特定应用场景开发的硬件设备，这类场景往往对实时性要求高，同时也对功耗与设备尺寸进行限制。SoC 作为嵌入式设备的一种实现方式，完整的功能和较低的功耗比兼具，因此非常适合作为本研究的基础开发平台。第二章对本研究使用的 Zynq SoC 平台与 SDSoC 开发工具进行详细的介绍。

1.3 现场可编程逻辑门阵列（FPGA）

FPGA（Field Programmable Gate Array，现场可编程逻辑门阵列）是一种特殊的半导体设备，特点在于可以对硬件逻辑进行重复编程。与 ASIC（Application-Specific Integrated Circuit，专用集成电路）相比，FPGA 比 ASIC 速度要慢，而且占用更多空间。但 ASIC 在设计和实现完成之后就不能修改，而 FPGA 可以不断修改其硬件配置，因此能大大提高硬件开发效率。与 CPU 相比，FPGA 针对特定形式的计算速度更快，而且功耗很低，因此在嵌入式设备和 SoC 中往往将 FPGA 视为重要的组成部分。

FPGA 的基础组件为 CLB（Configurable Logic Block，可配置逻辑块），其中包含 LUT（Look-Up Table，查找表）、全加器和触发器（Flip-Flop）等等基本元素。CLB 十分灵活，既可以用来实现时序与组合逻辑，也可以用来实现 RAM。CLB 之间则通过可编程的线路进行连接。FPGA 的可编程性主要取决于两个部分：一个是查找表的配置，查找表中包含 SRAM 与多路选择器，SRAM 的值随着硬件逻辑设计不同而不同，进而改变 CLB 的计算结果；另一个是连接 CLB 的线路，线路的交叉点也是可以配置进而改变 CLB 之间互联模式的。除此以外，不同的 FPGA 版本具有不同的资源配置，比如 BRAM（Blocked RAM，块内存）与 DSP（Digital Signal Processing，数字信号处理单元）等等在不同的 FPGA 上具有不同的数量，性能也不尽相同。

传统的 FPGA 开发使用的是 VHDL，Verilog，SystemC 等硬件描述语言，这类语言

用于表述硬件的寄存器传输级别（Register Transfer Level，简称 RTL）的抽象。使用这类语言完成设计并仿真验证之后，便可以使用不同 FPGA 厂商提供的工具进行 FPGA 的综合（Sythesis）、实现（Implementation）和比特流生成（Bitstream Generation）。综合把电路的高级抽象转换为底层的逻辑门之间的连接，建立逻辑门网表。实现则是针对具体的 FPGA 硬件，进行逻辑网表的布局（Placement）、布线（Routing）以及满足资源的限制等等。最后的生成可以编程到 FPGA 上的比特流，比特流用来配置 FPGA 上的各类硬件资源。与软件编译相比，FPGA 硬件的综合、实现与比特流生成往往需要花费几十分钟到数小时，依 FPGA 的资源和硬件设计大小而定。

综上所述，FPGA 具有可重构的硬件逻辑、优异的低功耗以及成熟的开发流程，并且广泛应用在各种计算平台中。FPGA 也有其缺点，一方面是开发门槛比较高，没有数字电路基础的软件开发者很难写出性能优秀的 FPGA 应用；另一方面，FPGA 只适用部分的计算过程，涉及到复杂的条件判断、循环等计算过程更适合部署于 CPU 执行。因此，本研究将 FPGA 作为硬件加速器，即部署计算量大但逻辑简单的计算过程于 FPGA 上加速。同时本研究也是用高层次综合工具提高硬件逻辑的开发效率。接下来进一步介绍高层次综合的基本概念和使用方式。

1.4　高层次综合

高层次综合（High-Level Synthesis，简称 HLS）是一种将行为级的、算法级的代码转换为硬件逻辑的过程。高层次综合的输入不再是传统的硬件描述语言，取而代之的是 C/C++ 等高级程序语言。高层次综合使得硬件逻辑开发可以从软件代码开始：传统的开发方式需要等到 RTL 实现完成之后才能验证，迭代周期很长；而高层次综合工具可以令开发者先实现好软件系统，之后直接把需要在 FPGA 上加速的代码进行高层次综合即可。通过高层次综合工具，软件工程师可以不用学习 RTL 级别的描述语言就可以实现硬件逻辑。根据 Xilinx 的调查显示，基于高层次综合的设计开发方法与传统方法相比时间加快 4 倍，结果质量提高 0.7 到 1.2 倍。

一般的高层次综合流程包含如下几步：控制流和数据流图（Control Data Flow Graph）分析，资源分配（Resource Allocation），调度（Scheduling），绑定（Binding）等等。高层次综合的流程本质上是求解在资源和时间的约束下，能达到最优的资源分配和调度策略。这种最优解往往依赖启发式的算法，因此高层次综合工具分析的结果往往比较保守，在资源配置和调度上不一定能取得最佳的结果，因此需要通过开发者手动指定生成硬件的选项。比如指定某一运算需要使用 DSP 实现，某一循环需要流水线化等等。

综上，高层次综合工具提供了一种生产力更高的开发 FPGA 应用的方式，为了取得更高的性能也需要开发者对应用进行分析、手动生成限制条件。本研究使用的 SDSoC

工具中整合了 Xilinx Vivado HLS 工具，在后续的硬件加速器的设计中主要用它进行硬件逻辑的实现和优化。

1.5 总结

第二章　使用技术

本章阐述本研究的整体设计。整体设计部分在基本原理之上，给出更具体的实现方案设计。从本研究的目标出发，为了提供可以运行于 FPGA SoC 平台的深度学习框架，首先应确定用什么 FPGA SoC 平台和使用什么样的开发工具。其次，如果自行从头构建深度学习框架，一方面工作量过于庞大、需要考虑很多优化问题；另一方面也缺乏足够的受众，研究人员更熟悉流行的深度学习框架。因此应选用成熟的流行的深度学习框架作为基础，在其上进行针对特定嵌入式平台的适配与优化。

基于上述的考虑，本章首先介绍 Xilinx Zynq 平台的系统架构、搭载的 ARM 与 FPGA 的硬件特点、以及本研究使用的基于 Zynq 的开发板 Zedboard 的具体特性等等。之后给出 Xilinx SDSoC 工具的基本情况介绍，包括其包含的工具以及支持的开发流程。随后介绍本研究选用的基础深度学习框架 Caffe，主要涉及与后续开发密切相关的软件架构和实现方式。最后，综述整体设计方案，即本研究的最终结果是如何有机结合 ZYNQ，SDSoC 和 Caffe。

2.1　Zynq 平台

Zynq 平台的全称是“全可编程 SoC”（All Programmable SoC），由双核的 ARM Cortex-A9 处理系统（Processing System，简称 PS）与 Xilinx 可编程逻辑（Programmable Logic，简称 PL）组成。除此以外，还包含片上内存（On-chip memory），外部内存接口（External Memory Interface），各类外围设备输入输出接口（I/O Peripherals and Interfaces）以及连接 PS 与 PL 的高速 AXI 总线等等。

本研究基于 Zynq 平台的最新系列 Zynq-7000 进行系统设计与实现。

2.1.1　Zynq 系统架构

Zynq 平台的系统架构如图 2.1 所示。接下来分块介绍 Zynq 的几个组成部分。

ARM 处理系统

Zynq SoC 系统上搭载了双核 ARM Cortex-A9 MPCore 处理器，该处理器可以运行到最高 1GHz，拥有指令和数据的 32KB L1 缓存与 512KB L2 缓存，以及 NEON 媒体处理引擎（media-processing engine）和浮点数向量处理单元（Vector Floating Point Unit，简称 VFPU）等等。Cortex-A9 的性能十分优异，足以满足多种情况下的计算需求。

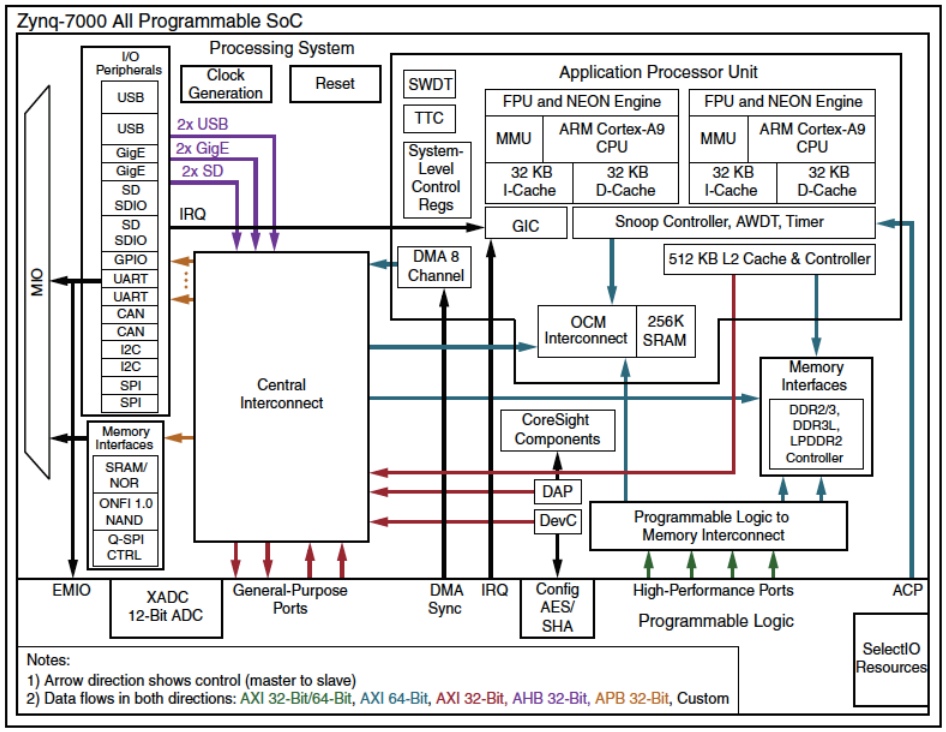

图 2.1　Zynq-7000 平台的系统架构

本研究中依赖该 ARM 处理器提供的 CPU 端环境进行大部分的计算。有关 ARM 处理器与内存系统和 PL 的交互接口在 2.1.4 节介绍。

可编程逻辑

Zynq 的可编程逻辑（PL）与一般的 FPGA 设计一致，主要包含 CLB，BRAM 以及 DSP 处理单元。相关指标如下表所示：

Programmable Logic Cells	LUTs	Flip-Flops	BRAMs	DSP Slices
85K	53,200	106,400	560 KB	220

表 2.1　Zynq-7000 XC7Z020 可编程逻辑上资源容量

本研究使用的 XC7Z020 系列 Zynq 的可编程逻辑的配置与性能大致上与 Artix-7 FPGA 相似，硬件资源比较有限。因此想要取得高效的性能必须做精细的资源分配和优化。相关的优化策略和技巧在第四章详述。

数据通路 AXI 与存储系统

存储系统与数据通路负责连接处理系统与可编程逻辑，以及处理和其他外围设备通信。Zynq 平台的存储系统主要由三个部分组成：处理系统中的缓存（L1 与 L2 cache）和

片上内存（On-chip Memory）；可编程逻辑上的 BRAM；以及大容量的外部存储（External Memory）。Zynq 上的几种数据通路可以完成这三个组成部分之间的不同形式的交互。

Zynq 上的数据通路的基础是 ARM 的 AXI（Advanced eXtensible Interface）总线协议，属于 AMBA（Advanced Microcontroller Bus Architecture）协议的一部分。AXI 的最新版本是 AXI4 协议，Xilinx 设计并实现了基于 AXI4 的 IP 核，使得 AXI 变得更加灵活、高效和便捷。AXI4 协议主要有三类：针对大数据量传输的 AXI4，小数据量的 AXI4-Lite，以及针对流数据的 AXI4-Stream。Zynq 上的数据通路正是基于 Xilinx 提供的 AXI IP 核实现的。

Zynq 基于 AXI 总线协议连接处理系统和可编程逻辑，主要实现两类接口：高性能 AXI 端口（High-Performance AXI ports）与 ACP（Accelerator Coherency Port）接口，二者区别关键在内存数据一致性上。高性能 AXI 端口可以实现处理系统和可编程逻辑高速的数据传输，但不保证 CPU 缓存中的数据与内存读写一致。ACP 接口则在硬件机制上保证缓存中的数据在传输之前被写回内存、内存更新之后缓存同步更新。

在设计 Zynq 应用时具体使用哪种数据通路取决于系统的特性。本研究在第三章给出对数据通路选择的原则，在第四章会具体讨论两种数据通路的性能对比。

2.1.2　Linux 运行模式

基于 Zynq 平台开发的应用有两种运行模式：Standalone 与 Linux 模式。前者代表应用的运行不用启动操作系统，后者则要求先启动 Linux 操作系统再运行应用。考虑到深度学习应用并不是简单的计算，还需要各种数据处理的操作，并且科研人员更熟悉基于操作系统的运行模式，本研究只在 Linux 模式下进行开发。

Linux 模式首先启动预先加载与 ROM 中的引导程序，该程序从 SD 卡中读取第一阶段启动程序（First Stage Boot Loader, 简称 FSBL），FSBL 负责处理可编程逻辑的比特流（bitstream）和 Linux 引导程序（u-boot）的加载与执行。引导程序之后分别加载 Linux 的镜像文件（uImage）、设备树文件（Device Tree）和根系统文件（Rootfs）。Linux 镜像主要包含 Linux 内核，设备树文件定义了硬件设备环境供引导程序动态加载，根系统文件则包含了系统的根目录和必要的程序与库。准备就绪之后即可通过 UART 接口或者 SSH 连接到 Zynq 平台，运行程序或进行开发。

Xilinx 提供了 bootgen 工具来创建 SD 卡引导程序，以及一系列预生成的文件，例如 Linux 镜像，根系统文件等等。SDSoC 工具中包含了 bootgen，以及更多跟硬件平台相关的预生成文件，可以更简便地生成 SD 卡中需要的内容。在 2.3 节中会进一步介绍。

2.1.3 Zedboard

本研究选用的是 Zedboard 作为 Zynq 开发平台。Zedboard 是基于 Zynq-7000 系列的扩展式处理平台，Zedboard 上除了满足 Zynq-7000 所要求的 ARM Cortex-A9 MPCore 处理器与 Xilinx FPGA 之外，还增加了如下配置：512MB DDR3 内存、256MB QSPI 闪存和一系列输入输出接口（HDMI、VGA、OLED 与音频接口）等等。除此以外，Zedboard 成本低廉，还有 Xilinx SDSoC 的特别支持，尤其是预先搭载好的输入输出接口对深度学习应用的开发（计算机视觉等应用）非常方便。因此本研究使用 Zedboard 作为最终深度学习框架所运行的平台。

2.2 SDSoC 开发环境

Xilinx 于 2015 年 3 月推出了 SDSoC 开发环境，提供类似于嵌入式 C/C++ 开发体验。SD 即“软件定义”（Software Defined）：SDSoC 的目标是降低 FPGA 应用开发门槛，让更多的软件开发者，而不是专业的 FPGA 开发人员，可以通过 C/C++ 代码直接生成 Zynq SoC 系统。此外，SDSoC 综合了 Vivado HLS，Xilinx ARM GNU 编译工具链，以及一系列系统生成工具（bootgen, Xillybus 等等），令开发效率得到很大提升。但是 SDSoC 的最大优势在于其对传统 SoC 开发流程的改进。接下来主要介绍 SDSoC 的开发流程与基本使用方式。

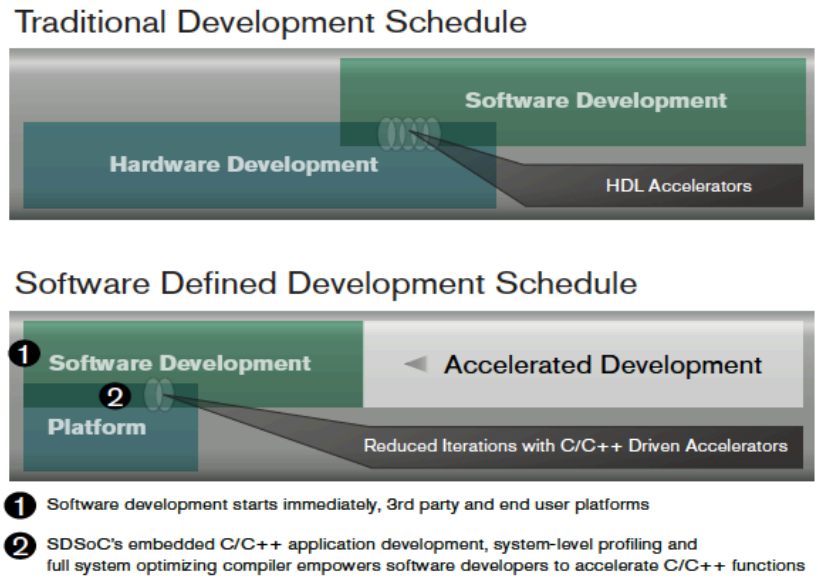

图 2.2 传统的开发流程（Traditional Development Schedule）与软件定义的开发流程（Software Defined Development Schedule）的区别

2.2.1　软件定义的开发流程

传统的 SoC 平台的系统开发流程首先需要完成硬件部分的设计，之后软件开发者才能通过已经综合实现好的硬件接口进行进一步开发。这种开发流程迭代周期非常长，软件部分的开发在硬件设计过程中处于停滞状态，硬件设计只能等软件部分完成才能获得反馈。基于 SDSoC 的开发基于软件定义的开发流程：先用 C/C++ 代码完成整个系统的设计，之后以函数为单元部署于可编程逻辑，进行硬件加速。SDSoC 甚至提供了硬件加速性能预测工具，可以令开发人员在长时间的综合实现完成之前就能大概了解目前的优化策略是否正确。图 2.2 是对两种开发流程的对比。

2.2.2　基本概念与使用方式

数据移动网络

SDSoC 开发环境关键概念是数据移动网络（data motion network），表示 PS 与 PL 之间的数据通路模式：SDSoC 使用预先设计好的数据通路 IP 核来负责 PS 与 PL 之间的数据传输，这些 IP 核称为数据移动单元（data mover）。比如 AXIDMA_SG 就表示通过 AXI 总线用 Scatter-Getter 的方式通过 DMA 传输数据。数据移动单元与数据类型密切相关，标量数据只能用 AXI_LITE 传输，而数组与更复杂的数据类型可以用 AXI_DMA_SG，AXI_DMA_SIMPLE，AXI_DMA_2D 等数据传输单元传输。一般而言 SDSoC 可以根据数据类型，内存分配方式等分析出最适合使用的数据传输单元，但开发者也可以使用预编译指令（pragma）进行优化。

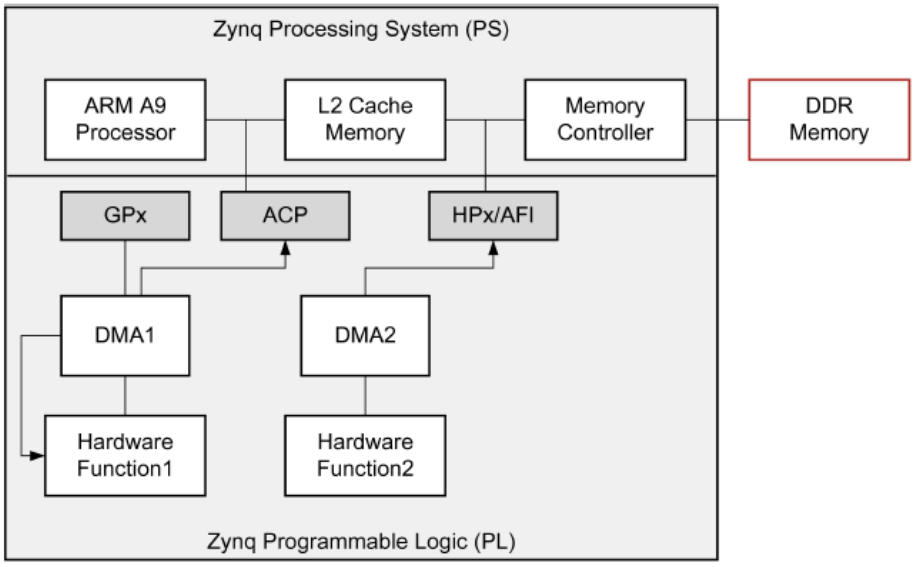

图 2.3　Zynq AXI 总线协议的几种端口

使用方式

SDSoC 的可以通过其 IDE 使用，也可以基于纯粹的 Linux 命令行使用。由于 IDE 的限制更多，本研究完全使用命令行环境开发。SDSoC 的最关键工具是 sdscc/sds++。这两个程序本质上为 C/C++ 编译器，可以完全兼容 gcc/g++ 的命令行选项。但与一般编译器不同，sdscc/sds++ 可以通过命令行选项指定：1）需要 HLS 的函数：包含函数名和要求的时钟频率；2）数据移动网络的时钟频率；3）目标硬件平台：比如 zed（针对 Zedboard）或 zybo（针对 ZYBO 开发板）。因此，通过 sdscc/sds++ 工具可以轻易与原有的 Linux 项目进行兼容。有时候也需要更细粒度的操作，可以使用 arm-xilinx-linux-gnueabi 的 GNU 工具链对 C/C++ 代码进行交叉编译，使用 sdslib 封装 IP 核等等，在第三章对 Caffe 和第三方库的移植中会详细介绍。

本研究的应用完全依赖 SDSoC 进行开发，利用 Vivado HLS 进行硬件逻辑编写和优化，SDSoC 自带的 IP 核进行数据通路的自动生成，以及 Xilinx ARM GNU 工具链编译 CPU 端代码并进行链接。最后 SDSoC 把项目全部整合到 SD 卡中。

2.3 深度学习框架 Caffe

Caffe[3] 是加州伯克利大学计算机视觉与学习中心①开发的深度学习框架，基于 C++ 与 CUDA 开发。Caffe 因其速度快，数据模版清晰易懂，并拥有开源的、高度模块化的代码而广受开发者欢迎。

本研究选用 Caffe 作为移植到 Zynq 上进行移植和加速的对象，主要是因为其清晰的软件架构，非常便于修改和优化。除此以外，与其他深度学习框架（如基于 LuaJIT 的 Torch，基于 Python 的 Theano）相比，Caffe 的 CPU 端代码完全基于 C++ 实现，与高层次综合工具、ARM GNU 编译工具链、SDSoC 等等可以无缝衔接。因此本研究最终选用 Caffe 作为优化的基础。

接下来首先介绍 Caffe 的使用中出现的关键概念，之后着重介绍与移植相关的 Caffe 软件架构和神经网络层的实现。

2.3.1 基本概念

Caffe 中的网络层 Layer 和网络 Net 是两个关键概念。Caffe Layer 是对神经网络层的抽象，通过指定使用的预定义网络类型，上下层连接的网络名称，以及本层的参数就可以构建一个 Caffe Layer。Caffe Layer 之间连接起来的有向无环图（Directed Acyclic Graph，简称 DAG）便是 Caffe Net，即神经网络。Caffe 的神经网络训练是用 Caffe Solver

① Berkeley Vision and Learning Center，http://bvlc.eecs.berkeley.edu/

（求解器）实现的，一个 Caffe Solver 中定义了训练的网络名称、控制训练的参数以及运行的硬件（CPU 或 GPU）等等。Caffe Layer、Caffe Net 和 Caffe Solver 都可以使用 Google Protocol Buffer②文档格式进行定义。

从具体实现的角度出发，理解 Caffe 的代码关键需要理解 Layer 类和 Blob 类。每个 Caffe 的 Layer 类都实现了 Forward 与 Backward 两个方法，相当于神经网络中的前向与反向计算。Forward 与 Backward 的两个参数都是用 Blob 类表示的网络层输入输出。Blob 类本质上是多维数组，因此可以处理各种形状的网络层输入输出。

Caffe 的标准用法有两种，一种是通过编译出的 caffe 命令行工具直接训练 Caffe 模型，其中 Caffe 模型与训练数据需要提前定义好；除此以外，也可以使用 Caffe 动态链接库（libcaffe.so）在其他的 C++ 应用中使用 Caffe 提供的接口。两种方法都十分方便。

2.3.2　软件架构

Caffe 的软件架构主要包含如下几个部分，依赖关系具有清晰的分层次的结构：

1. 基础架构：数据类型的定义，比如 blob.cpp 定义了 Blob 数据结构，net.cpp 与 layer.cpp 分别定义了网络类与网络层类。同时常用数学函数 math_functions.cpp 也十分重要，Caffe 的各种大运算量的计算基本都会调用该文件所定义的函数。
2. 网络层的实现：在 layers/目录下定义了常见的神经网络层的实现，比如卷积神经网络 conv_layer.cpp，ReLU 层 relu_layer.cpp 等等。相关 GPU 代码也包含其中。
3. 求解器实现：在 solvers/目录下定义了多种求解器。
4. 其他：比如 Caffe 使用的 Protobuf 文件等等。

2.3.3　第三方库依赖

Caffe 所依赖的第三方库不多，可以简单分为如下几类：

1. 数据存储：LMDB 与 LevelDB
2. 数据格式和接口：HDF5 与 Protobuf
3. 数学计算：BLAS 库比如 ATLAS 或 OpenBLAS
4. 其他：比如日志 Google glog，基础组件 Boost 等等

总之，Caffe 的分层设计和简单的第三方库依赖都非常便于优化和修改。本研究针对 Caffe 的移植与优化在第三章中详述。

2.4　综合设计方案

② Google Protocol Buffer (`https://developers.google.com/protocol-buffers/docs/overview`）不仅提供了一种接口数据传输的格式，同时还可以生成数据传输与序列化的代码。

第三章　主体工作

从上述整体设计出发，本研究的主要实现了 SoCaffe——一个基于深度学习框架 Caffe 并能运行于任一基于 Zynq 架构的嵌入式 SoC 设备上的深度学习框架。SoCaffe 的名称来源于 SoC 与 Caffe 的结合。

本研究的主体工作可以分为如下三步:

1. 系统分析与架构设计：划分 Caffe 在 SoC 上的软硬件职责，确定 CPU 运行和 FPGA 加速的部分；
2. FPGA 加速器实现：实现针对 GEMM 算法的硬件逻辑与应用优化策略；
3. ARM 系统实现：构建数据通路、综合编译软硬件系统；

本研究的最终成果包含编译好的基于 FPGA 的硬件加速库、使用硬件加速的 Caffe 链接库以及全部依赖的第三方链接库。使用本研究提供的 Caffe 版本与动态链接库，便可以直接实现基于 Zynq SoC 的深度学习应用。本研究的主体工作完全基于 SDSoC 工具实现：FPGA 加速器使用了 SDSoC 包含的 Vivado HLS 工具进行硬件逻辑的编写与优化；数据通路构建依赖 SDSoC 提供的数据接口 IP 库；ARM 系统的综合编译依赖 SDSoC 提供的 ARM GNU 编译工具链。

接下来按步骤介绍主体工作。

3.1　系统分析与架构设计

只用 ARM CPU 完全足以运行 Caffe，但性能不尽如人意。SoC 上的 CPU 本身计算性能有限，而且难以并行化，因此需要 FPGA 硬件进行部分计算的硬件加速。由于 GPU 与 FPGA 在高性能计算的应用中拥有类似的特性，因此可以从 Caffe 的 GPU 加速函数中进行筛选，选择最适合 FPGA 计算的函数进行硬件加速。筛选的原则主要有以下三条：1）函数需要拥有较长的计算时间，占据总运行时间的很大比重，根据阿姆达尔定律（Amdahl's Law）加速类函数可以取得最大的加速比；2）代码逻辑简单，并行度高，适合在 FPGA 上运行；3）数据传输量小，不会花费很多时间在 FPGA 与 CPU 之间的数据传输上。从上述角度出发，本研究选用 GEMM（GEneral Matrix Multiplicaion，通用矩阵乘法）作为在 FPGA 上主要优化的目标函数。

GEMM 计算属于 BLAS（Basic Linear Algebra Subprograms，基本线性代数子程序）

集合，结合了矩阵乘法与加法计算：

$$\mathbf{C} \leftarrow \alpha op(\mathbf{A}) op(\mathbf{B}) + \beta \mathbf{C} \tag{3.1}$$

A，**B**，**C** 分别是输入矩阵，α 与 β 分别为 GEMM 计算的系数。此外，op 函数会随着配置不同，对输入矩阵 **A** 与 **B** 不变换或者进行转置变换。因此 GEMM 具有充分的灵活性，可以配置出各种形式的矩阵乘法与加法组合。

Caffe 中大量使用了 GEMM 计算，主要在网络层的计算过程中，比如卷积层的卷积计算（`conv_layer.cpp`）。Caffe 计算卷积的策略是把复杂的卷积操作变为简单的矩阵乘法问题，进而可以使用效率更高的、充分优化过的 BLAS 计算库。变换的方法主要是使用 im2col 操作对原始输入矩阵进行变换，把分离的矩阵块聚合起来①。总之，Caffe 通过 GEMM 实现的卷积操作取得了不错的优化效果。

综上，整个深度学习框架在 SoC 上的划分如图 3.1 所示。处理系统上运行深度学习应用，并链接第三方链接库（比如 OpenBLAS、LMDB 等等）、caffe 链接库（`libcaffe.so`）与加速器链接库（`libaccel.so`）。硬件加速器链接库提供调用 FPGA 硬件逻辑的高层接口。FPGA 上主要实现了 GEMM 计算的硬件逻辑。PS 与 FPGA 之间通过 AXI 总线和内存进行数据交互。

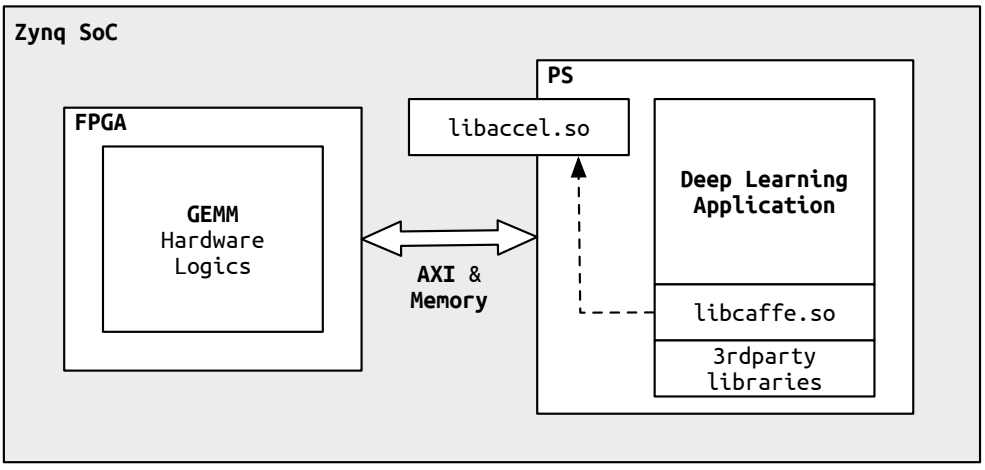

图 3.1　SoCaffe 的系统架构

接下来分别介绍 FPGA 的加速器实现与 ARM 处理系统的实现工作。

① Caffe 的作者贾扬清（Yangqing Jia）在这篇文章中解释了 Caffe 是如何使用 GEMM 与 im2col 来实现卷积层的：`https://github.com/Yangqing/caffe/wiki/Convolution-in-Caffe:-a-memo`

3.2　FPGA 加速器实现

基于 FPGA 实现的 GEMM 加速器是本研究的关键工作。本研究使用 Vivado HLS 工具设计并实现了基于固定大小的矩阵块之间的 GEMM 计算，并使用多种资源分配和调度策略进行优化；处理系统与 FPGA 之间的数据通路使用 SDSoC 工具提供的 IP 核实现；最后通过链接使得 Caffe 可以使用 GEMM 加速器进行计算。整个 GEMM 加速器基本使用了全部的 FPGA 硬件资源，相对于运行于 CPU 上的 OpenBLAS 库有 5.45x 倍速度提升，相应的测试结果在第四章详述。

3.2.1　GEMM 实现

GEMM 本身并不复杂，以传统的三重循环实现的矩阵乘法为基础就可以在 CPU 上实现。下面的算法中以矩阵 **C** 的行、列为优先，假设每次迭代的坐标为 (i, j)。每次迭代首先把 $\beta \times \mathbf{C}(i, j)$ 作为累积变量 *sum* 的初值，进而遍历矩阵 **A** 和 **B** 的 i 行与 j 列（如果 **A** 或 **B** 需要转置则使用 i 列或 j 行，算法的 5、6 行使用了 C 语言的三元条件表达式表示转置条件），将 **A** 与 **B** 的对应元素和 α 相乘并累加。最后把 *sum* 赋给 $\mathbf{C}(i, j)$。

Algorithm 1 GEMM 的原始三重循环算法

```
1:  for i = 0 to M do
2:      for j = 0 to N do
3:          sum ← β × C[i][j]
4:          for k = 0 to K do
5:              a ← (TA) ? A[i][k] : A[k][i]
6:              b ← (TB) ? B[k][j] : B[j][k]
7:              sum ← α × a × b
8:          end for
9:          C[i][j] ← sum
10:     end for
11: end for
```

GEMM 分块矩阵算法

如果要将该算法移植到 FPGA 上，首先需要对矩阵计算进行分块。FPGA 上只有有限的硬件资源，而且在计算开始之前已经固定，所以不可能进行使用该算法在 FPGA 上计算任意大小的矩阵。所谓分块（tiling），是指将原先以矩阵元素为单元的遍历方式，改变为以固定大小的矩阵块为单元的遍历方式。分块矩阵算法的基本原理是分块矩阵乘法与分块矩阵转置的计算公式（如图 3.2，3.3）。

上述公式中的矩阵元素都是矩阵块，每个矩阵中的矩阵块大小一致。m，n，k 分别为按照块大小划分后的矩阵中块的个数。分块矩阵乘法运算的结果中，每个结果矩

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,k} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,k} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k,1} & B_{k,2} & \cdots & B_{k,n} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{m,1} & C_{m,2} & \cdots & C_{m,n} \end{pmatrix}$$

图 3.2　分块矩阵的乘法运算

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,k} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,k} \end{pmatrix}^T = \begin{pmatrix} A_{1,1}^T & A_{2,1}^T & \cdots & A_{m,1}^T \\ A_{1,2}^T & A_{2,2}^T & \cdots & A_{m,2}^T \\ \vdots & \vdots & \ddots & \vdots \\ A_{1,k}^T & A_{2,k}^T & \cdots & A_{m,k}^T \end{pmatrix}$$

图 3.3　分块矩阵的转置运算

阵块都是由块之间的乘法和累加构成的：

$$\mathbf{C}_{i,j} = \sum_{t=0}^{k} \mathbf{A}_{i,t} \times \mathbf{B}_{t,j}$$

而分块矩阵转置运算的结果相当于先以矩阵块为单元进行转置，之后分别对每个矩阵块进行内部转置。

从上述两个公式出发便可以得到 GEMM 的分块矩阵算法。BLK_M，BLK_N，BLK_K 分别为矩阵的块尺寸参数。本算法的框架基本与 GEMM 原始算法类似，但是其中计算参数都是矩阵块而不是矩阵元素。

Algorithm 2 GEMM 的分块矩阵算法

```
for bi = 0 to ⌈M/BLK_M⌉ do
  for bj = 0 to ⌈N/BLK_N⌉ do
    S ← β × C_{bi,bj}
    for bk = 0 to ⌈K/BLK_K⌉ do
      A_b ← A_{bi,bk} or transposed A^T_{bk,bi}
      B_b ← B_{bk,bj} or transposed B^T_{bj,bk}
      S ← α × A_b × B_b + S
    end for
    C_{bi,bj} ← S
  end for
end for
```

接下来具体介绍如何把分块矩阵算法实现于 FPGA 上。

硬件加速实现

FPGA 上主要加速的是 GEMM 分块矩阵算法的第 7 行，对两个固定大小的矩阵块求乘法并进行累加。因此，矩阵块的大小不能任意取值，主要受限于 FPGA 的硬件资源数量：越大的矩阵需要越多的板上计算资源和存储资源。此外，为了适配 Caffe 的接口，该算法主要针对单精度浮点数格式设计实现。最后，为了简化 FPGA 实现的接口，CPU 端会分配几个固定大小的矩阵块用来保存传输到硬件的参数。

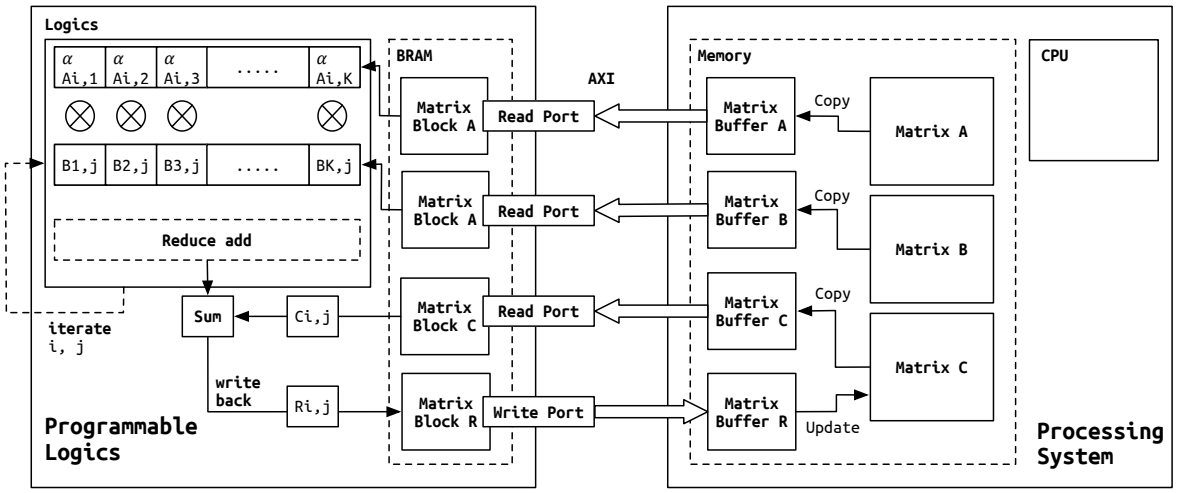

图 3.4　GEMM 的软硬件设计

图 3.4 是具体的硬件加速设计架构：左侧为 FPGA 可编程逻辑，右侧为处理系统，二者之间通过 AXI 总线通信。如图所示，每次计算会先从矩阵中复制矩阵块到专门划分的缓存区中，如果有需要转置则在复制过程中进行转置（如算法 2 的第 5，6 行）。之后，FPGA 把缓存的矩阵块放入 BRAM 中，在每个迭代步都进行矩阵乘法运算中的点积操作。矩阵 **C** 也被读入，和点积结果进行累加。FPGA 上的数据写回到矩阵 **R** 中，防止与对 **C** 的访问冲突。当计算完成之后，处理系统使用矩阵 **R** 的结果对内存中的矩阵 **C** 进行更新，每次更新一个矩阵块大小的数据。

上述 GEMM 加速系统的设计的软件端代码可以很容易地实现，但是硬件端代码主要有两部分组成：将总线中的数据复制到 BRAM，以及从 BRAM 中读取数据进行计算。硬件逻辑代码如代码1和代码2所示。

这两份代码都默认矩阵块的大小必须完全相同，这是一种简化，在实际实现的代码中尺寸可以不一致（相应的论述参见第3.2.4节）。`gemm_accel` 是被指定为要被 HLS 生成硬件的函数，因此 `A_buf`，`B_buf` 与 `C_buf` 都会被实现为 FPGA 上的 BRAM②，`gemm_accel` 所调用的 `gemm_accel_kernel` 中的计算过程也会被实现为硬件逻辑。

② 有的时候 HLS 工具也会用 LUT 来生成存储，选择 BRAM 还是 LUT 可以在程序中显式指定

Listing 1 gemm_accel 函数的定义（未优化）

```
void gemm_accel(float A[BLK_M*BLK_K], float B[BLK_N*BLK_K], float C[BLK_M*BLK_N], float ALPHA,
  ↪  float R[BLK_M*BLK_N]) {
  int i, j, k;
  float A_buf[BLK_M][BLK_K], B_buf[BLK_K][BLK_N], C_buf[BLK_M][BLK_N];
  A_RowCopy: for (i = 0; i < BLK_M; i ++)
    A_ColCopy: for (j = 0; j < BLK_K; j ++)
    #pragma HLS pipeline II=1
      A_buf[i][j] = ALPHA * A[i*BLK_K+j];
  B_RowCopy: for (i = 0; i < BLK_K; i ++)
    B_ColCopy: for (j = 0; j < BLK_N; j ++)
    #pragma HLS pipeline II=1
      B_buf[i][j] = B[i*BLK_N+j];
  C_RowCopy: for (i = 0; i < BLK_M; i ++)
    C_ColCopy: for (j = 0; j < BLK_N; j ++)
    #pragma HLS pipeline II=1
      C_buf[i][j] = C[i*BLK_N+j];
  gemm_accel_kernel(A_buf,B_buf,C_buf,ALPHA,R);
}
```

Listing 2 gemm_accel_kernel 函数的定义（未优化）

```
void gemm_accel_kernel(float A[BLK_M][BLK_K], float B[BLK_K][BLK_N], float C[BLK_M][BLK_N],
  ↪  float ALPHA, float R[BLK_M*BLK_N]) {
#pragma HLS inline
#pragma HLS array_partition variable=A block factor=16 dim=2
#pragma HLS array_partition variable=B block factor=16 dim=1
  int i, j, k;
  float sum;
  Row: for (i = 0; i < BLK_M; i ++) {
    Col: for (j = 0; j < BLK_N; j ++) {
    #pragma HLS pipeline II=1
      sum = C[i][j];
      Prod: for (k = 0; k < BLK_K; k ++)
        sum += ALPHA * A[i][k] * B[k][j];
      R[i*DIM+j] = sum;
    }
  }
}
```

基本优化技巧

两个函数中出现的循环一般会被实现为硬件逐步迭代进行的计算，但因为在代码中使用 HLS 的流水线预编译指令（pipeline），因此循环的迭代步之间可以同时执行：上一步迭代的计算与这一步迭代的 BRAM 读取不冲突，可以并行。流水线化是实现高效硬件逻辑的关键步骤，在 Vivado HLS 工具中，流水线化与循环控制流密切相关，被流水线化的循环内部的嵌套循环会被默认进行循环展开（Loop Unrolling）。同时，初始化间隔（Initialization Interval，简称 II）指的是流水线需要几个时钟周期才能进行初始化，当 II=1 时流水线化达到最佳水平。II=1 的结果主要因为对 A 和 B 的数组划分（Array partition）：对 A 与 B 的访问可以同时通过多个端口进行，这样流水线便不会阻塞。

数据通路

数据通路主要由 SDSoC 生成，使用的是 axis_accelerator_adapter 这一 AXI 总线 IP 核[6]。该 IP 核可以支持 AXI4-Stream 与 AXI4-Lite 两种模式，其中 AXI4-Stream 可以支持 BRAM 和 FIFO 接口。使用 BRAM 还是 FIFO 一方面可以由开发者通过预编译指令主动指定，另一方面也可以由 SDSoC 进行判断，如果内存块会被顺序访问的话，则使用 FIFO，否则使用随机访问的 BRAM。根据1中对几个输入数组的访问模式，这些参数都是按照顺序进行访问的，因此可以使用预编译指令强行指定顺序访问。

总结

本节给出了基于分块矩阵算法的简单 GEMM 实现，其中用到了基本的流水线化、数组划分以及数据通路分析的技巧，而且也通过指定矩阵尺寸为 32 得到了性能和资源占用的报告。

但是存在两个问题：一是该实现的延迟过长（大于 4000），最终得到的硬件效率不高；二是尺寸为 32 不一定是最好的结果，FPGA 上还有很多空余资源可以使用。下一小节会对性能给出基本的分析和预测。

3.2.2　GEMM 性能分析与预测

本节从 GEMM 算法的特性出发，找出性能与 GEMM 分块算法中的块大小之间的数学关系。性能指标的单位为 GFLOPS（Giga floating-point operations per second，每秒 1G 次浮点运算操作个数）。由于 GEMM 分块算法有多种分块方式，为了简便起见本节中使用正方形划分的分块方式，即从矩阵 **A**，**B**，**C** 划分出的任意块 $\mathbf{A_{ik}}$，$\mathbf{B_{jk}}$ 与 $\mathbf{C_{ij}}$ 都是正方形。此外，本节性能分析只涉及 FPGA 硬件逻辑的计算时间，数据传输延迟暂不考虑。

理论上，FPGA 的运行时间（$time_{HW}$）延迟（$latency$，单位为时钟周期数）和时钟频率（$freq$，单位为 MHz）相关：

$$time_{HW} = \frac{latency}{freq} \tag{3.2}$$

延迟的大小与 GEMM 使用的矩阵块大小密切相关。矩阵块大小由三个参数决定：Dim_M，Dim_N，Dim_K③。原始实现（如3.2.1节所示）中延迟由两个子过程决定，分别为 BRAM 复制与 GEMM 计算。BRAM 复制过程包含三个独立的、流水线化的循环，总延迟（L_{BRAM}）等于三个循环的延迟之和。假设初始化间隔已经达到目标 1，则流水线

③ M，N，K 的语义与描述矩阵乘法的场景中使用的语义一致，即 Dim_M 为矩阵块 $\mathbf{A_{ij}}$ 的高，Dim_K 为其宽。其余可以类推得到。

化的循环延迟约等于循环体的延迟④加上循环周期数，因此：

$$\begin{aligned} L_{BRAM} &= L_{AMultLoop} + L_{BCopyLoop} + L_{CCopyLoop} \\ &= Dim_M \times Dim_K + L_{mult} + Dim_N \times Dim_K + L_{copy} \\ &+ Dim_M \times Dim_N + L_{copy} \\ &= S_A + S_B + S_C + 2L_{copy} + L_{mult} \end{aligned} \tag{3.3}$$

即 BRAM 复制过程的延迟等于三个矩阵块的面积和与两个复制的延迟（L_{copy}）、一个乘法运算的延迟（L_{mult}）的和。GEMM 计算延迟（L_{comp}）取决于流水线化的 GEMM 三重循环计算，由于其中第二层循环被流水线化，这部分的总延迟等于前两层的循环周期数加上第三层循环完全展开后的点积计算延迟（L_{prod}）：

$$\begin{aligned} L_{comp} &= Dim_M \times Dim_N + L_{prod} \\ &= S_C + Dim_K \times L_{add} + L_{others} \end{aligned} \tag{3.4}$$

简单对数据流进行分析可以发现，由于点积运算中包含存在依赖的求和运算，且这部分的延迟与 Dim_K 成正比⑤，因此 GEMM 计算延迟与 Dim_K 相关。

综上，假定矩阵块面积的大小是主要决定因素（其他延迟变量的值暂且忽略不计），且令 Dim 为最大矩阵块尺寸，则综合公式3.3与3.4，得到总延迟的表达式为：

$$\begin{aligned} latency &= L_{BRAM} + L_{comp} \\ &\approx S_A + S_B + 2S_C \leq 4Dim^2 \end{aligned} \tag{3.5}$$

即总延迟的大小的上界为 $4Dim^2$。

GEMM 的浮点数操作数总数（$FLOP$）也取决与矩阵块尺寸。从 GEMM 的公式出发3.1，得到（公式3.6中的下标为特定的运算步，Dim 依然为矩阵块尺寸中的最大值）：

$$\begin{aligned} FLOP &= FLOP_{\mathbf{AB}} + FLOP_{\alpha\mathbf{AB}} + FLOP_{\beta\mathbf{C}} + FLOP_{\alpha\mathbf{AB}+\beta\mathbf{C}} \\ &= 2Dim_M \times Dim_N \times Dim_K + Dim_M \times Dim_N \\ &+ Dim_M \times Dim_N + Dim_M \times Dim_K \\ &= 2S_C \times Dim_K + 3S_C \\ &\leq 2Dim^3 + 3Dim^2 \end{aligned} \tag{3.6}$$

④ **A** 矩阵块的 BRAM 复制循环与其他循环不同，延迟主要取决于与 α 的乘法计算延迟，因为乘法比赋值更慢

⑤ 根据7第 183 页，对于浮点数格式的加法，为了防止出现精度问题，不会使用平衡过的加法树（Adder Tree），因此延迟与 Dim_K 的关系是线性而不是对数的。

综合公式3.5与3.6可得：

$$\begin{aligned} GFLOPS &= \frac{FLOP}{latency} \times frequency \times 10^{-9} \\ &= \frac{2Dim^3 + 3Dim^2}{4Dim^2} \times frequency \times 10^{-9} \\ &= \frac{2Dim + 3}{4} \times frequency \times 10^{-9} \end{aligned} \tag{3.7}$$

随 *Dim* 的增加，*GFLOPS* 上升。

综上，GEMM 计算的硬件性能与使用的矩阵块的最大尺寸成正比，因而优化的目标应为最大化 GEMM 中使用的矩阵块大小。矩阵块大小的上限取决于 FPGA 上各类资源的总量，和 GEMM 硬件设计使用各类资源的方式。接下来具体阐述如何通过修改 HLS 生成硬件设计的方式来优化 GEMM 的计算效率。

3.2.3　GEMM 硬件资源约束分析

本节给出各个 FPGA 资源的约束条件，通过求解一个线性规划模型来得到在给定 FPGA 资源个数情况下，能得到的最大 GEMM 矩阵块大小。由3.7可知，GEMM 矩阵块越大性能越好，因此本节所得到的 GEMM 矩阵块大小也是对计算性能而言最优的。本节的 FPGA 资源约束条件是针对原始实现3.2.1分析得到的，其他约束条件在下一章（3.2.4）的优化技巧中给出。

必须指出的是，本节的资源约束条件在精度上有误差。首先，因为 GEMM 计算只是 FPGA 上硬件逻辑之一，还有数据通路和其他控制逻辑也占用 FPGA 的资源，所以本节求解的最大矩阵块尺寸可能最终无法实现；其次，HLS 过程给出的硬件资源报告会略大于实际的实现结果，因为实际实现过程中会有资源共享等优化；再次，本节只求解了 GEMM 算法主要使用的 FPGA 资源（DSP，LUT，FFF 和 BRAM），对其他的资源占用暂不考虑；最后，本节只计算了主要占用资源的操作或变量，其他资源占用也没有计入。尽管如此，本节的结论基本上可给出 GEMM 矩阵块的理论上限，可以指导对矩阵块大小的设置。

BRAM 资源约束

在高层次综合阶段，BRAM 资源使用的都是 BRAM18K，即一个 BRAM 有 18Kb 存储空间的配置。每个 BRAM 有两个端口，既可以配置为两个读端口，也可以配置为一个写端口、一个读端口。但是，BRAM18K 的端口带宽是 18bit，如果读取的数据宽

度大于 18bit 则只能有一个读端口⑥。

原始实现（3.2.1）中，主要占用 BRAM 资源的是代码1中的板上缓存：A_buf，B_buf 与 C_buf，还有 R。R 也默认实现为 BRAM，占用空间与 C_buf 相同。此外，为了能实现达到初始化间隔（II）为 1 的目标，必须对 A_buf 与 B_buf 均进行了数组划分，划分结果必须满足 A_buf 的第二维、B_buf 的第一维能同时访问全部的数据。假定数据类型为单精度浮点数（float），则根据前面所述这里只能由一个读端口。因此，令获取数据类型字节数的函数为 sizeof()，并且假定数组划分后的每个子数组都能用一个 BRAM18Kb 装下⑦，则可以得到 A_buf 与 B_buf 使用的 BRAM 数量（分别为 A_{BRAM} 和 B_{BRAM}）：

$$A_{BRAM} = Dim_K \times \lceil \frac{Dim_M \times \texttt{sizeof(float)} \times 8}{18 \times 1024} \rceil \approx Dim_K \lceil 1.7 Dim_M \times 10^{-3} \rceil \approx Dim_K \quad (3.8)$$

$$B_{BRAM} = Dim_K \times \lceil \frac{Dim_N \times \texttt{sizeof(float)} \times 8}{18 \times 1024} \rceil \approx Dim_K \lceil 1.7 Dim_N \times 10^{-3} \rceil \approx Dim_K \quad (3.9)$$

即二者使用的 BRAM 其实与数组划分的对应维度大小，乘上由其他维度组成的数组所占用的 BRAM 个数相同。

C_buf 没有进行数据划分，因此可以直接按照其数据量大小进行计算，其使用 BRAM18K 的个数为：

$$C_{BRAM} = R_{BRAM} = \lceil \frac{S_C \times \texttt{sizeof(float)} \times 8}{18 \times 1024\ bits} \rceil \approx \lceil 1.7 S_C \times 10^{-3} \rceil \quad (3.10)$$

此外，尽管在代码中没有体现，但是 SDSoC 工具会为 gemm_accel 函数生成处理输入与输出的 IP 核，如第3.2.1节所示。本节暂时不考虑这部分的 BRAM 使用。

综上，假设 FPGA 上 BRAM18K 的资源为 Num_{BRAM} 个，则针对 BRAM 的约束条件为：

$$\begin{aligned} A_{BRAM} + B_{BRAM} + C_{BRAM} + R_{BRAM} &= 2Dim + \lceil 3.4 Dim^2 \times 10^{-3} \rceil \\ &\leq Num_{BRAM} \end{aligned} \quad (3.11)$$

计算资源约束

本节只讨论针对 DSP，FF 和 LUT 的计算资源约束，其中 DSP 资源的类型为 DSP48E。GEMM 算法对计算资源的使用主要来自2中第三层循环的点积计算。由于流水线化的目标是达到 II 为 1，因此这里的乘法与加法必须满足延迟的要求。根据 HLS

⑥ 根据8第 16 页对"Programmable Data Width" 的描述，提到只有 SDP 模式才能对 BRAM18K 读取大于 18b 的数据。SDP 即 Simple Dual-Port，要求只能配置一个端口为读端口

⑦ 本研究中没有使用到比 BRAM18Kb 限定的范围还要大的子数组，即没有矩阵块的任一维度大小大于 576，所以这里假定暂时成立。

的综合报告发现，针对这一部分的加法和乘法操作使用的资源如下表所示：

	DSP48E	FF	LUT
fadd	2	324	424
fmul	3	151	325

表 3.1　单一加法操作（fadd）与乘法（fmul）操作使用的资源个数

由于点积运算的次数为 Dim_K，因此针对运算资源的约束可以轻易得到：

$$Dim_K \times 2 + Dim_K \times 3 = 5Dim_K \leq Num_{DSP} \tag{3.12}$$

$$Dim_K \times 324 + Dim_K \times 151 = 475Dim_K \leq Num_{FF} \tag{3.13}$$

$$Dim_K \times 424 + Dim_K \times 325 = 749Dim_K \leq Num_{LUT} \tag{3.14}$$

求解最大矩阵块尺寸

综合3.11、3.12、3.13、3.14等约束条件，以及本研究使用的 Zynq 系列 XC7Z020 的资源限制[8]：

BRAM18K	DSP48E	FF	LUT
140	220	106400	53200

表 3.2　XC7Z0202 的资源

代入上述数值，可以得到 Dim 的四个上界：44（DSP48E)，71（LUT），224（FF）。并且将 44 代入 BRAM 约束条件3.11发现满足约束。因此，原始实现主要受限于 DSP 的个数，最大能达到的矩阵尺寸上限为 44。

综上，根据硬件资源约束条件和实际数据求解可以发现，原始实现没有很好地利用板上的全部资源，在达到最大矩阵尺寸时，只有 LUT 的资源占用为 61.9%，其它都小于 40%。下一节会利用本节的约束条件以及相关结论优化 GEMM 对资源的使用。

3.2.4　GEMM 优化

本节首先解决第一小节提出的高延迟问题，通过合并几个复制循环即可实现。之后主要解决上一节的硬件资源占用不均匀的问题，总体而言有如下几种解决方案：一种想法是将使用 DSP 资源的计算用 LUT 等资源实现。此外，通过牺牲精度、使用更小的数据类型来降低每个操作所占用的资源，进而提高矩阵块的大小也是一种关键优化策略。本节使用了固定小数点数据类型（fixed-point data type）进行优化。此外，上节的约束条件只使用了最大矩阵块尺寸 Dim，尽管模型更清晰简单，但是实际上如果矩

阵块形状不规则，那么在满足约束条件的前提下也可以得到更大的矩阵面积（GFLOPS与矩阵面积相关）。

优化延迟

在延迟的表达式3.5中，三个复制循环占据了大量的时间（基本为 3/4）。如果在编译代码的过程中对矩阵尺寸进行判断，强制 $Dim_M = Dim_N = Dim_K$，则可以将三个复制循环合并为一个：

Listing 3 gemm_accel 函数：优化延迟

```
void gemm_accel(float A[DIM*DIM], float B[DIM*DIM], float C[DIM*DIM], float ALPHA, float
 ↪  R[DIM*DIM]) {
  int i, j;
  float A_buf[DIM][DIM], B_buf[DIM][DIM], C_buf[DIM][DIM];
  for (i = 0; i < DIM; i ++)
    for (j = 0; j < DIM; j ++) {
    #pragma HLS pipeline II=1
      A_buf[i][j] = ALPHA * A[i*VEC+j];
      B_buf[i][j] = B[i*DIM+j];
      C_buf[i][j] = C[i*DIM+j];
    }
  gemm_accel_kernel(A_buf,B_buf,C_buf,ALPHA,R);
}
```

修改式3.3可以得到 BRAM 复制步骤的延迟下降为 $Dim^2 + L_{mult}$，即只需要循环 Dim^2 次并且循环主体的延迟为最长的乘法计算。

但是上述要求三个尺寸都相等的条件过于严格了，不利于对硬件资源的分配优化。第3.2.4小节给出一种令 Dim_K 的值不同的不规则矩阵计算方式，对本小节的过程给出了改进。

优化硬件资源分配

为了在不增加延迟的基础上降低 DSP 的使用，本研究将部分计算强制使用非 DSP 资源实现。默认的浮点数乘法与加法运算都需要同时占据 DSP，FF 与 LUT 来实现其功能（如表3.1），三者之间是有固定比例的，因此根据 FPGA 资源比例和计算使用的硬件资源比例可以确定 DSP 是决定矩阵尺寸上限的主要因素。

解决该问题的主要思路是改变默认的计算资源使用方式。通过使用 Vivado HLS 的 resource 预编译指令，就可以指定某个参数的计算一定要使用某一种资源来实现。一般情况下这种做法可以改善对 DSP 的依赖情况。但是，对于包含循环的设计而言，一个操作往往会在多次迭代中频繁使用。对于这种情况，如果将该运算完全用 LUT 或其他资源实现，可能会造成 DSP 的使用与 LUT 的使用再次失衡，甚至会导致 LUT 的使用超出系统上限。

综上，本研究提出一种有效地优化硬件资源分配，均衡不同资源使用的方法：通过手动指定使用 DSP 的操作的迭代范围，来最大化对硬件资源的利用（如代码4所示）。

Listing 4 gemm_accel_kernel 函数片段：优化硬件资源分配

```
float tmp_LUT, res_LUT, tmp;
#pragma HLS resource variable=tmp_LUT core=FAddSub_nodsp
#pragma HLS resource variable=res_LUT core=FMul_nodsp
FullLoop: for (k = 0; k < GEMM_FULL_UPPER; k ++) {
  res = A[i][k] * B[k][j];
  tmp = sum + res;
  sum = dsp;
}
TmpLUTLoop: for (k = GEMM_FULL_UPPER; k < GEMM_TMPLUT_UPPER; k ++) {
  res = A[i][k] * B[k][j];
  tmp_LUT = sum + res;
  sum = tmp_LUT;
}
ResLUTLoop: for (k = GEMM_TMPLUT_UPPER; k < GEMM_RESLUT_UPPER; k ++) {
  res_LUT = A[i][k] * B[k][j];
  tmp = sum + res_LUT;
  sum = tmp;
}
FullLUTLoop: for (k = GEMM_RESLUT_UPPER; k < DIM_K; k ++) {
  res_LUT = A[i][k] * B[k][j];
  tmp_LUT = sum + res_LUT;
  sum = tmp_LUT;
}
```

上述代码有四个循环，分别为只使用综合资源、使用加法 LUT 实现、使用乘法 LUT 实现，都使用 LUT 实现四种情况。确定每个循环长度的具体数值依然可以用约束条件求解实现。在表3.1添加不使用 DSP 的运算单元得到：

	DSP48E	FF	LUT
fadd	2	324	424
fmul	3	151	325
fadd_no_dsp	0	374	556
fmul_no_dsp	0	523	798

表 3.3　添加无 DSP 占用的加法与乘法操作实现

根据表3.3，代入循环长度 d_1, d_2, d_3, d_4，得到四个约束条件如下：

$$5d_1 + 3d_2 + 2d_3 \leq Num_{DSP} \tag{3.15}$$

$$475d_1 + 525d_2 + 847d_3 + 897d_4 \leq Num_{FF} \tag{3.16}$$

$$749d_1 + 881d_2 + 1222d_3 + 1354d_4 \leq Num_{LUT} \tag{3.17}$$

$$d_1 + d_2 + d_3 + d_4 = Dim_K \tag{3.18}$$

求解上述模型，可以发现当 Dim_K 的值为 56 的时候上述约束条件是无解的⑧。考虑到不规则的数值有可能不利于矩阵块的划分，最后本研究使用 $Dim_K = 56$ 的情况作为该优化策略的最优矩阵块。同时，由于对所有矩阵尺寸有长度一致的假定，因此 $Dim_M = Dim_N = 56$。此时的资源占用为⑨：

资源类型	BRAM18K	DSP48E	FF	LUT
HLS 资源占用 (%)	84	96	34	89
最终资源占用 (%)	63.57	96.36	40.27	56.59

表 3.4　最优化硬件资源分配

上述资源占用中 DSP 资源与 LUT 资源基本均匀使用，但是 BRAM 和 FF 的占用还是相对较少，需要考虑其他的优化策略。有关该最优分配的性能参见第4.1.1章。

优化矩阵块形状

之前的假定是矩阵块的大小尺寸必须完全一致，主要是为了在数据复制的步骤中使用一套循环逻辑来降低延迟。但如果将条件放松一些，当 M 与 N 对应的块尺寸相等，K 对应的块尺寸相异时，循环逻辑依然可以保持不变，只需要增加循环内部的条件判断即可。

允许大小不等的矩阵块的优点在可以更精细地调整硬件资源的使用。BRAM 的使用由三个块尺寸同时决定，而计算单元只是由 Dim_K 此在 DSP 和 LUT 的资源达到上限的时候，还可以通过增加 Dim_M 与 Dim_N 来增大 BRAM 的资源占用，从而提高 FPGA 每次运算的运算量。

代码5中 `DIM` 与 Dim_M 和 Dim_N 相等，`VEC` 与 Dim_K 相等。修改约束3.11为：

$$\begin{aligned} A_{BRAM} + B_{BRAM} + C_{BRAM} + R_{BRAM} &= 2VEC + \lceil 3.4DIM^2 \times 10^{-3} \rceil \\ &\leq Num_{BRAM} \end{aligned} \tag{3.19}$$

由于 `VEC` 主要受限于第3.2.4小节的计算，值取为 56。代入 BRAM 的个数 140，并且预留 5% 空间给其它逻辑，则 `DIM` 的上界可以求得为 78 ⑩。但是最终实现的时候，

⑧ 本研究使用 WolframAlpha(`http://www.wolframalpha.com`)语言进行上述模型的求解，代码为：

```
Reduce[{
5x+3y+2z <= 220,
475x+525y+847z+897t <= 106400,
749x+881y+1222z+1354t <= 53200,
x+y+z+t = dimK,
x>= 0, y >= 0, z >= 0, t >= 0}, {x,y,z,t}];
```

`dimK` 通过尝试的方法发现 $\texttt{dimK} \geq 58$ 时，程序给出无解提示。此外，LUT 与 FF 资源可能被应用于别的应用逻辑，在实际求解的时候对两个资源的上界乘上 0.9，降低可用资源上限。

⑨ HLS 的资源占用报告中没有给出 IP 核与其他逻辑的资源占用，因此本表在最后一行也给出最终资源占用

⑩ 即求解一个 $2 * 56 + 3.4\texttt{DIM}^2 \times 10^{-3} \leq 140 * 0.95$ 不等式。

Listing 5 gemm_accel 函数：优化矩阵块形状

```
void gemm_accel(float A[DIM*VEC], float B[DIM*VEC], float C[DIM*DIM], float ALPHA, float
  ↪  R[DIM*DIM]) {
  int i, j;
  float A_buf[DIM][VEC], B_buf[VEC][DIM], C_buf[DIM][DIM];
  for (i = 0; i < DIM; i ++)
    for (j = 0; j < DIM; j ++) {
    #pragma HLS pipeline II=1
      if (j < VEC) A_buf[i][j] = ALPHA * A[i*VEC+j];
      if (i < VEC) B_buf[i][j] = B[i*DIM+j];
      C_buf[i][j] = C[i*DIM+j];
    }
  gemm_accel_kernel(A_buf,B_buf,C_buf,ALPHA,R);
}
```

如果 DIM 数值越大，则对时钟频率的要求会更高。在满足时钟要求的情况下，这里能取到最大的 DIM 数值为。最终的报告如下表：

资源类型	BRAM18K	DSP48E	FF	LUT
HLS 资源占用 (%)	90	87	34	92
最终资源占用 (%)	42	96	34	89

表 3.5　最优化矩阵尺寸

优化数据类型—半精度浮点数

尽管上述几种优化策略已经能在 FPGA 上部署很大的矩阵块，执行很快的 GEMM 算法了。但是针对某些特殊的场景，比如要求实时性的计算时，之前的优化结果尚并不如人意。在这些场合中，对速度的要求高过对精确度的要求，因此可以使用更低精度的数据类型来降低平均每次计算所用到的资源，进而增加板上部署的矩阵块的大小。

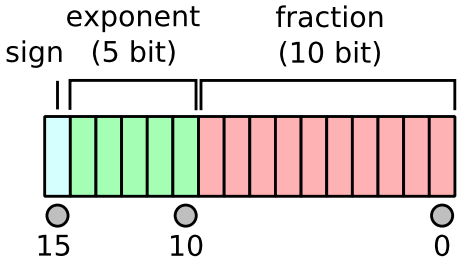

图 3.5　IEEE-754 半精度浮点数的格式描述，图片来源(https://en.wikipedia.org/wiki/File:IEEE_754r_Half_Floating_Point_Format.svg)

HLS 提供了对 IEEE-754 标准[2] 中的半精度浮点数（Half-precision floating point）的支持，其格式包含 1 个符号位，5 个指数位和 10 个尾数位（图3.5），共 16 位。半精

度浮点数已经广泛地应用于计算机图形学等领域了。这里的优化实现方法只是将板上 BRAM 存储的矩阵块都用半精度浮点数声明，不改变输入参数的类型与处理系统中声明的数据类型。在 FPGA 的计算内核中使用半精度浮点数进行强制类型转换。代码如下：

Listing 6 `gemm_accel` 函数片段：优化数据类型

```
1  void gemm_accel(float A[DIM*VEC], float B[DIM*VEC], float C[DIM*DIM], float ALPHA, float
   ↪  R[DIM*DIM]) {
2    int i, j;
3  #ifdef GEMM_HALF_DATA
4  #include "hls_half.h"
5  #define data_t half
6  #else
7  #define data_t float
8  #endif
9    data_t A_buf[DIM][VEC], B_buf[VEC][DIM], C_buf[DIM][DIM];
10   for (i = 0; i < DIM; i ++)
11     for (j = 0; j < DIM; j ++) {
12     #pragma HLS pipeline II=1
13       if (j < VEC) A_buf[i][j] = ALPHA * A[i*VEC+j];
14       if (i < VEC) B_buf[i][j] = B[i*DIM+j];
15       C_buf[i][j] = C[i*DIM+j];
16     }
17   gemm_accel_kernel(A_buf,B_buf,C_buf,ALPHA,R);
18 }
```

可以看出第 3 行对缓存的数据类型修改为 `half`，在内部进行复制操作的时候发生了强制类型转换。

当数据格式发生改变的时候，原先基于 `float` 假设设计的约束条件都需要进行修改：1）BRAM：由于此时单一数据只占用 16 个 bit，则一个 BRAM18K 可以拥有两个读端口，并且 `C_buf` 占用的数据空间也需要修改，而由于 `R` 依然使用浮点类型，其数据占用不变。2）计算资源：此时使用的计算单元为 `hadd` 与 `hmul`，基本不使用 DSP 资源。

经过测试，使用半精度浮点数的 GEMM 算法可以使用最高达到 $Dim = 96$[11]的矩阵块。

资源类型	BRAM18K	DSP48E	FF	LUT
HLS 资源占用 (%)	80	43	30	40
最终资源占用 (%)	92.50	43.64	38.51	52.17

表 3.6　修改数据类型的综合报告

综上，GEMM 的优化目标是尽可能地利用 FPGA 的资源，以部署更大的矩阵块来

[11] 此时依然可以使用不规则矩阵块的优化，但是如果把 Dim_K 增到很大，则会造成传输的数据量过大，不能满足时钟频率的情况

提高计算效率。使用了这一系列优化方式之后，GEMM 自身相对于纯 CPU 的实现已经取得了很大的加速比。下一节具体描述如何令 Caffe 运行于 Zynq SoC 上，并与 GEMM 加速器相结合。

3.3　ARM 系统实现

SoCaffe 在处理系统上的工作（如图3.1）主要分为三个部分：编译与优化使用硬件加速库、交叉编译 Caffe 与其他第三方库、生成 SD 卡文件包。本节的工作主要依赖 SDSoC 工具，使用其提供的 IP 核、代码生成来构建与 FPGA 加速器的通信机制，ARM GNU 工具链构建动态链接库和交叉编译，bootgen 等工具生成包含比特流的 Linux 系统镜像等等。

3.3.1　构建与使用硬件加速库

尽管在 Caffe 中使用 GEMM 加速库只用调用封装好的硬件加速器入口函数即可，但还有些问题需要考虑。首先，GEMM 加速器并不适合全部形状的矩阵参数：当矩阵尺寸比较小时，使用 GEMM 加速器甚至比软件版本慢很多，主要原因是 GEMM 加速器的额外开销与计算对小尺寸矩阵计算时间而言是显著的。其次，在执行 GEMM 硬件加速函数的同时，CPU 也应该利用起来，进行一些内存分配和复制等等简单操作，这就要求对 GEMM 硬件函数的调用代码需要异步执行。最后，GEMM 硬件加速库与一般软件库不同，其中包含了一个硬件的比特流，选择何时加载、如何加载比特流也是需要考虑的问题。接下来会逐一讨论如何构建与使用 GEMM 硬件加速库。

PS-PL 接口生成

调用 GEMM 加速器函数时，会以处理系统分配好的内存地址作为参数，SDSoC 主要根据参数对应的内存块的分配方式来确定生成什么样的接口。如果内存块是以 `malloc` 分配生成的，则 SDSoC 使用 Scatter-Gatter 类型的接口访问内存中的数据。如果内存块以 `sds_alloc` 分配，则 SDSoC 可以认识到该内存块是地址连续的，因此会使用顺序访问的接口（比如 AXIDMA_SIMPLE）。除此以外，如果不需要保证缓存的数据一致性，可以指定 SDSoC 生成使用 AFI 端口的高速访问数据通路，否则 SDSoC 会自动使用 ACP 端口。

调用方式与条件

正如本节序言所述，GEMM 加速器是针对固定大小的矩阵块实现的，因此无法根据输入矩阵的形状进行调整。尤其是当输入矩阵形状非常不规律时，比如一条边很短

（~ 10）而另一些边很长（≥ 1000）时，使用 GEMM 加速器会带来非常大的性能损耗。本研究使用一种简单条件来判断是否应该执行 GEMM：只有当输入矩阵的三个尺寸都大于 128[12]时才调用 GEMM 加速器（如代码7所示，其中当定义了 SDS 时才会编译 gemm_sds 接口），否则使用 CPU 端的 BLAS 库。

Listing 7 caffe_cpu_gemm 函数

```
template<>
void caffe_cpu_gemm<float>(const CBLAS_TRANSPOSE TransA,
    const CBLAS_TRANSPOSE TransB, const int M, const int N, const int K,
    const float alpha, const float* A, const float* B, const float beta,
    float* C) {
  int lda = (TransA == CblasNoTrans) ? K : M;
  int ldb = (TransB == CblasNoTrans) ? N : K;
  int TA  = (TransA == CblasNoTrans) ? 0 : 1;
  int TB  = (TransB == CblasNoTrans) ? 0 : 1;
#ifdef SDS
  if (M >= 128 && N >= 128 && K >= 128)
    gemm_sds(TA, TB, M, N, K, alpha,
        (float *)A, lda, (float *)B, ldb, beta, (float *)C, N);
  else
#endif
  cblas_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha, A, lda, B,
      ldb, beta, C, N);
}
```

代码中的 gemm_sds 接口是对算法2的直接实现，参数列表与 cblas_sgemm 的标准一致，这里不加赘述。

链接库生成

传统的 Zynq SoC 应用开发在完成硬件函数的设计之后，是无法使用类似于软件方式调用该硬件逻辑的：必须要通过配置 AXI 总线的数据并传输数据才能执行硬件逻辑。但使用 SDSoC 进行开发时却可以直接调用硬件函数，究其原因，主要是因为 SDSoC 在编译过程中对原始的硬件函数进行了封装，并将封装后的版本替换了原始的函数调用。封装的函数包含了额外的数据准备，信号传输与等待命令完成等一系列过程。正因为如此，SDSoC 可以为 FPGA 硬件逻辑自动构造出一个软件动态链接库。

但硬件逻辑的比特流却不包含在动态链接库中，而是放在一个独立的数据文件中。比特流文件可以在系统引导的过程中加载，也可以在系统运行过程使用 xdevcfg 设备动态配置。

⑫ 128 的数值主要根据第 4 章中对 OpenBLAS 的对比测试得到，当矩阵尺寸大于 128 时，GEMM 加速器相对于 OpenBLAS 的性能才有了明显提升

3.3.2　交叉编译

所谓交叉编译，指的是一种编译出来的程序是运行在其他环境中的编译过程。SDSoC 中提供的 ARM GNU 工具链就是在 x86 环境中交叉编译出 ARM 环境中可执行代码的交叉编译工具。使用该工具链编译 GEMM 硬件加速库，以及 Caffe 本身都很简便，但是涉及到 Caffe 的第三方库依赖便会遇到很多问题。针对第三方库的交叉编译主要有三种策略：

1. 直接修改环境变量中的 `CC`，`LD` 等为 ARM GNU 工具链中对应的工具，可以用来处理比较简单的库的编译；
2. 使用第三方库提供的配置方案进行交叉编译配置：大部分第三方库都会使用 configure、CMake 等工具生成 `Makefile`，以方便交叉编译；OpenBLAS 在编译工具链的配置之外也会要求指定对应系统架构的名称，进而可以充分优化性能[13]。
3. 其他：有的第三方库完全不支持交叉编译，主要原因是编译过程需要得到编译结果测试的反馈。比如 HDF5 在编译的过程中，会调用编译出来的程序来生成与目标系统相关的代码，处理起来非常麻烦[14]。

因为第三方库的交叉编译标准不一，而且目标环境相对固定，因此本研究提供一套已经编译好的第三方动态链接库供 SoCaffe 使用，不需要重复编译，直接链接即可运行。

3.3.3　系统生成

系统生成过程需要得到最终的、可用的 SoCaffe 系统，其中要包含编译好的动态链接库，FPGA 比特流，以及其他工具等等。本研究按照如下流程构建 SoCaffe 系统：

1. 编译硬件加速器动态链接库 (`libaccel.so`)，生成包含 FPGA 比特流的 SD 卡与相应的引导文件；
2. 编译 Caffe 链接库（`libcaffe.so`），并链接第三方库函数和硬件加速器函数；
3. 编译使用 SoCaffe 环境的代码，比如各种基于 Caffe 的深度学习应用等等；
4. 将所有生成的链接库，可执行程序复制到第一步生成的 SD 卡目录中，进而复制到 Zynq SoC 使用的 SD 卡；
5. 启动系统，将链接库复制到系统库目录下 (`/lib`)，运行需要执行的程序；

---

[13] 根据 OpenBLAS 的文档(`https://github.com/xianyi/OpenBLAS/blob/develop/README.md`)，Zynq-7000 搭载的 Cortex-A9 ARM CPU 应该使用 `ARMV7` 模式配置，该配置可以使用处理器上的浮点数计算硬件。但是由于 SDSoC 中附带的 ARM GCC 是 Lite 版本，不支持编译使用硬件浮点数计算单元，因此本研究中编译 OpenBLAS 为 `ARMV5` 模式，即所有运算都是软件实现的，没有使用浮点数处理器加速。

[14] HDF5 的官方文档明确指出不支持交叉编译(`https://www.hdfgroup.org/HDF5/faq/compile.html`)，但根据邮件列表中的一封指南(`https://lists.hdfgroup.org/pipermail/hdf-forum_lists.hdfgroup.org/2013-September/007104.html`)，经过一些复杂的 CMake 设置是可以完成编译的。本研究就是采用上述方法最终完成 HDF5 的交叉编译。

3.4 总结

本章首先给出了系统架构划分的思路，根据 Caffe 的软件架构和计算特性，决定只把 GEMM 操作移植到可编程逻辑中，其余功能在处理系统中实现。GEMM 在 FPGA 上的移植分为三步，首先对传统软件算法进行改进，实现对 FPGA 优化的分块矩阵算法；然后对原始实现版本进行性能分析与预测，并给出优化的方向—修改硬件资源的占用方式；最后实现了各种不同的优化策略。此外，也需要对 Caffe 中对 GEMM 算法的调用进行修改，编译第三方链接库，并完成整个系统的生成。

下一章对 SoCaffe 进行完整详细的测试。

第四章　实验结果

本章对 SoCaffe 的性能进行测试，首先给出 GEMM 加速器的独立测试结果，接着给出综合的 SoCaffe 性能测试。GEMM 测试中主要包含 GEMM 设计过程中的各种优化策略的对比，从硬件资源使用和性能两个角度分析；以及包含与运行于 CPU 的 OpenBLAS 的性能对比。SoCaffe 测试首先使用 Caffe 的单元测试对 Caffe 的功能进行测试，给出一些关键计算的运行时间；之后使用 convnet-benchmarks[①]作为基准进行卷积层的性能测试；最终通过两个深度学习应用：MNIST 与 CIFAR-10 网络对 SoCaffe 进行整体评估。

4.1　GEMM 测试

4.1.1　优化策略对比

		Baseline	ResAlloc	IrrShape	HalfFloat
Largest Matrix Shape	M	32	56	64	96
	N	32	56	64	96
	K	32	56	56	96
FPGA Resource Usage (%)	BRAM18K	23	42	42	40
	DSP48E	72	76	90	43
	FF	18	35	34	30
	LUTs	46	95	91	40
Latency (cycles)		2352	6736	8673	19030
Clock frequency (MHz)		143	143	143	143
Esti. GFLOPS		4.1713197	7.65610451	7.76648449	13.50436994
Block Test GFLOPS		3.31	6.77	7.2	12.31
Full Test GFLOPS		1.45	2.61	3.03	4.44

图 4.1　GEMM 优化策略对比结果

根据3.2.4节，得到如下四个版本实现：原始实现（Baseline）、资源分配优化（ResAlloc）、矩阵尺寸优化（IrrShape）以及半精度浮点数优化（HalfFloat）四个版本。本节选取四个版本的设计在最优参数配置下得到的实现作为对比样本，对比的项目包括：

1. 最大矩阵尺寸：根据目标版本，能得到的最大的矩阵尺寸；

① https://github.com/pku-ceca-research/convnet-benchmarks

2. FPGA 资源利用：BRAM，DSP，LUT 等指标所占用的百分比；
3. 延迟与时钟频率：时钟频率选用能综合得到的最快的时钟频率；
4. 预测性能与实际性能：用 GFLOPS 作为性能指标；

对比结果参见图4.1。可以发现最快的优化版本是使用半精度浮点数优化的版本，GFLOPS 与矩阵块的大小密切相关。尽管资源分配优化与矩阵尺寸优化在 K 对应的矩阵块尺寸上一致，但因为 M 与 N 的不同造成性能上矩阵尺寸优化略优。此外，实际性能测试包含两个，一个是块测试，即单纯测试从 CPU 调用 FPGA 上 GEMM 操作的时间；另一个是完整测试，包含完整的 GEMM 算法，从矩阵拷贝数据到缓存中的时间也要计入。可以看出 CPU 端的计算确实比较影响最终的计算性能。

4.1.2 OpenBLAS 对比测试

OpenBLAS 是非常成熟的 BLAS 计算库，而且也是 Caffe 兼容的 BLAS 库之一，因此其在 ARM CPU 上的运行时间可以作为 GEMM 硬件加速比的计算基准。

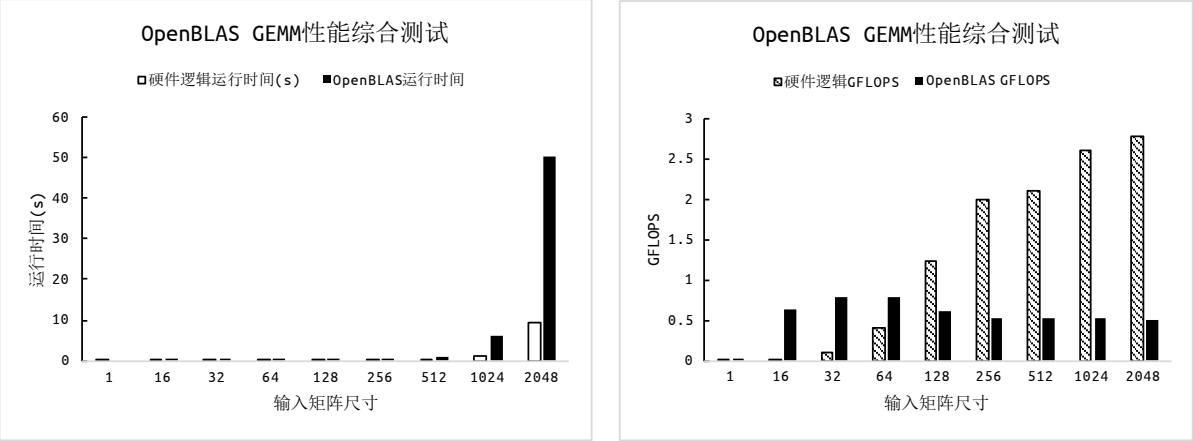

图 4.2　GEMM 与 OpenBLAS 的性能对比

这里测试使用的是GEMM在单精度浮点数精度下最快的版本（IrrShape）。与OpenBLAS在多种矩阵尺度下相比，GEMM硬件加速版本确实在小尺寸矩阵上速度很慢，但随着矩阵尺度大于 128 时，GEMM 硬件加速比越来越大。最终测试结果显示 GEMM 的硬件加速版本是 OpenBLAS 的 5.42 倍。

4.2 Caffe 测试

4.2.1 单元测试

Caffe 的单元测试使用 Google Test 测试框架进行搭建，在功能性测试的基础上也加上简单的对性能的测试。

4.2.2　网络性能测试

4.2.3　深度学习应用测试

MNIST

CIFAR-10

结论

pkuthss 文档模版最常见问题：

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

参考文献

[1] Yoshua Bengio, Aaron Courville and Pierre Vincent. "*Representation learning: A review and new perspectives*". *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **2013**, *35*(8): 1798–1828.

[2] IEEE Standards Committee *et al.* "*754-2008 IEEE standard for floating-point arithmetic*". *IEEE Computer Society Std*, **2008**, *2008*.

[3] Yangqing Jia, Evan Shelhamer, Jeff Donahue *et al.* "*Caffe: Convolutional Architecture for Fast Feature Embedding*". *arXiv preprint arXiv:1408.5093*, **2014**.

[4] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. "*Imagenet classification with deep convolutional neural networks*". In: *Advances in neural information processing systems*, **2012**: 1097–1105.

[5] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. "*Deep learning*". *Nature*, **2015**, *521*(7553): 436–444.

[6] *SDSoC Environment User Guide (UG1027)*, 2016-1.

[7] "*Vivado Design Suite User Guide: High-level Synthesis (UG902)*". **2016**.

[8] *Zynq-7000 All Programmable SoC Overview (DS190)*, 2016-1.

附录 A　附件

pkuthss 文档模版最常见问题：

\cite、\parencite 和 \supercite 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 texdoc pkuthss“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

致谢

pkuthss 文档模版最常见问题：

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，**[test-zh]**、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

北京大学学位论文原创性声明和使用授权说明

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。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在□一年/□两年/□三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日