高性能实时目标识别与检测系统

摘要

目标识别与目标检测是计算机视觉领域的重要任务。由于深度神经网络模型强大的表示能力，基于深度神经网络模型的目标识别与目标检测算法成为主流。在嵌入式设备中，实现高性能的目标识别与检测任务，往往受算力，成本，功耗等约束。主流的 CPU 与 GPU 的方案虽然较为成熟，但往往能耗效率较低。FPGA 低功耗的特点，相比起 CPU 和 GPU 能达到更高的能耗效率，更加适合部署于边缘嵌入式平台。

但是，业界主流目标识别与检测算法在 FPGA 上进行部署，有着较高的难度，需要经过漫长的开发周期。面对着 FPGA 开发设计中的挑战，本系统基于 RTL 语言，设计了一种高性能的卷积加速器，在软核 Cortex-M3 的辅助控制下，在 FPGA 上实现目标识别与目标检测的深度神经网络模型的实时推理，并在 Genesys ZU 3EG 板卡上得到验证。

关键词：FPGA 部署神经网络，目标识别与检测，软核任务调度

目录

1 项目背景

近年来,FPGA 的神经网络加速的研究逐渐成为了研究热点。随着算力的提升，深度神经网络逐渐流行，适合各种任务的算法模型不断被提出。

大约从 2015 年开始，基于 FPGA 的卷积加速器的研究，也如同雨后春笋般迅速拔地而起，许许多多优秀的加速器实现方案被提出，但是基本只停留在理论与实验阶段。深鉴科技，基于 XILINX FPGA 芯片，设计了一套开发套件 DNNDK（Deep Neural Network Development Kit），是第一个向用户提供统一 API 接口的 FPGA 神经网络快速部署的工具链。2018 年，XILINX 收购了深鉴科技，在 DNNDK 的基础上进行改进，于 2019 年推出 Vitis AI，简化了 FPGA 神经网络的部署。

目前，Vitis AI 已经推出 1.3 版本，支持主流框架和能够执行各种深度学习任务的最新模型；提供一组全面的预优化模型，可随时部署在 Xilinx 设备；提供功能强大的量化器，支持模型量化，校准和微调；AI 库提供统一的高级 C ++和 Python API，实现从边缘到云的最大可移植性。

然而，无论是 Vitis AI 还是 DNNDK，实现的源码并不开源，而且几乎只对 XILINX Zynq Ultrascale 系列的 FPGA，如 ZCU102,ZCU104,ZCU106 等板卡，有着相对较好的支持。整个 Vitis AI 工具链的搭建工程，交叉编译，生成镜像的过程繁杂易出错，官方教程也在不断地变更修正。用户若想在 FPGA 上部署自定义的神经网络，依旧十分困难。FPGA 部署神经网络的研究在当前，仍处于“无轮子可用”的状态。

面对如上的诸多挑战，该系统基于 RTL 语言，自主设计了一种高性能的卷积加速器，在软核 Cortex-M3 的辅助控制下，能够实现主流神经网络模型的推理，并且配合摄像头采集与显示回显的回路，构建了一个高性能实时目标识别与检测系统。

2 系统框架与实现功能介绍

该系统是完成的是一个常见的视频采集->处理->视频回显任务：

摄像头采集视频流后，经过预处理，通过 AXI-VDMA（Video Direct Memory Access）以多缓存机制缓存至外部存储器 DDR3/DDR4 后，VDMA 输出稳定的视频

流流向显示器驱动进行视频的实时回显。

卷积加速器通过软核或硬核 CPU 的控制，从 VDMA 帧缓存区域获取输入特征图，进行目标识别/目标检测神经网络模型的推理，推理结果经软核或硬核 CPU 处理后，与 VDMA 输出的视频流通过 Video Mixer 输出至显示器。

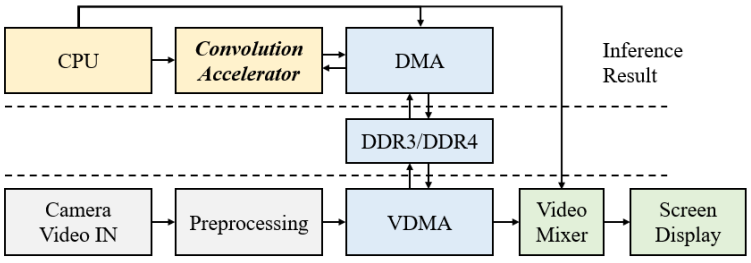


图 1 系统框架图

3 开发板选型

该系统目前使用 Digilent 设计的 Genesys ZU 3EG 开发板进行验证。

Genesys ZU 3EG 开发板搭载 Zynq Ultrascale 系列的 XCU3EG 芯片，板载 MIPI 摄像头接口，DisplayPort 视频输出接口，4G DDR4 内存条与其它丰富的外设。

Ultrascale 系列的 FPGA 芯片，相比起普通的 7 系列带有硬核 CPU 的 Zynq 或者不带硬核 CPU 的 Artix 系列，有着卓越的优势。Ultrascale 系列的 FPGA 芯片在硬件资源上，相比起 7 系列的 FPGA 更加丰富；改进的制作工艺与芯片架构，使得硬件逻辑能工作在更高的频率，达到更好的性能；Ultrascale 系列 FPGA 的 DDR4 的接口，相比起 DDR3 能达到更高的内存带宽；MIPI 摄像头接口和 DisplayPort 接口，能实现高清视频的实时采集并回显，且简化了摄像头采集显示器回显的回路搭建。

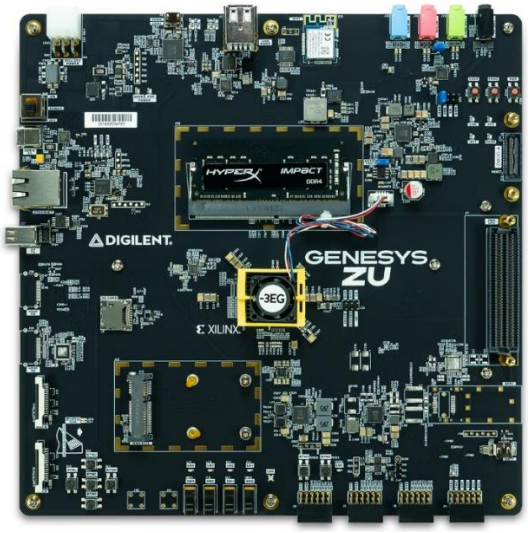


图 2 Genesys ZU 3EG 开发板

4 开源软核与硬核的使用情况

开源软核使用情况：ARM DesignStart Cortex-M3

硬核使用情况：使用 Zynq Ultrascale 的 MPSoC 辅助开发设计

该系统的最终目标，是实现一个纯 FPGA 硬件逻辑实现的高性能实时目标识别检测系统：软核 CPU Cortex-M3 控制调度 RTL 语言描述的卷积加速器，实现目标识别与目标检测的神经网络模型的推理。但是，该系统的实现无法一蹴而就，需要一个循序渐进的过程。所以，该系统的实现将分为两个阶段：

前期，将进行卷积加速器的基本功能验证，软核 CPU 开发流程的探索，与摄像头显示器回路的搭建。在该阶段，硬核 CPU 能够简化设计，加速功能的验证。所以，在该阶段采取硬核为主，软核为辅的设计方案。

后期，卷积加速器的功能验证完毕后，将逐步“去硬核化”，逐步让软核承担更多的任务，逐步向纯 FPGA 硬件逻辑实现迈进。

通过这样一个由硬核逐步过渡到软核的系统开发过程，能够降低开发难度。

目前，系统已经完成了卷积加速器的基本功能验证，搭建好了软核 CPU 并能完成基本的功能任务，搭建好了摄像头采集与显示器回显的回路。

5 系统方案论证

5.1 摄像头显示器回路构建方案论证

方案一 选用传统的 OV5640 摄像头，连接至 PMOD 端口，进行视频采集。采用 HDMI 或 VGA 显接口，对采集到的视频流的进行回显。该方案的优点是方案成熟简单，参考案例较多；缺点是性能较差，无法充分发挥 Genesys ZU 3EG 开发板的特点。

方案二 选用 Digilent 官方推荐的 Pcam 5C 摄像头，与板卡的 MIPI 接口相连实现视频的采集。采用 PS 端的 DisplayPort 接口，对采集到的视频流的进行回显。该方案的优点是摄像头采集到的视频流能达到较高帧率与分辨率，充分发挥了 Genesys ZU 3EG 板卡的性能；缺点是方案较不成熟，参考资料较少。

为了实现高性能的实时摄像头采集+显示的功能，采用方案二 MIPI 摄像头+DisplayPort 的方案。

5.2 卷积加速器的实现方案论证

方案一 采用高层次综合对卷积加速器进行设计。该方案的优点是开发周期短，开发难度相对较低。缺点是高层次综合往往会生成冗余的逻辑，难以充分利用片上资源，难以进行细粒度的优化，难以达到较高的时钟频率。

方案二 部署 Xilinx 的 Vitis AI 套件，使用 Vitis AI 的 DPU 进行神经网络的加速推理。该方案的优点是使用 Vitis AI 套件对官方模型的推理能达到非常好的性能，无需自己设计卷积加速器。该方案的缺点是 Vitis AI 尚未成熟，且支持性较好的板卡较少，自定义硬件平台较难实现。

方案三 使用 RTL 语言从零开始设计卷积加速器。该方案的优点是能够有着最大的设计自由度，能进行细粒度的优化，能通过优良的设计避免冗余逻辑资源的生成，充分利用片上资源。该方案的缺点是开发周期极长，开发难度极大。

为了达到较高的设计自由度与较好的加速性能，采用方案三，使用 Verilog 开展对卷积加速器的 RTL 设计实现。

5.3 卷积加速器的调度与控制方案论证

方案一 使用 Genesys ZU 的硬核 CPU 对卷积加速器进行调度和控制。该方案的优点是硬核 CPU 具有较高的性能，能提高调度与控制的实时性。该方案的缺点是可扩展性差，无法用于无硬核 CPU 的 FPGA 板卡中，且无法体现比赛中队软核设计的要求。

方案二 搭载 ARM Cortex-M3，使用软核 CPU 对卷积加速器进行调度与控制。该方案的优点是软核 CPU 与卷积加速器耦合性强，扩展性通用性高，该系统能够应用在纯 FPGA 的板卡中，符合比赛的设定与要求。该方案的缺点是，软核 CPU 的性能远低于硬核 CPU，无法达到较高的性能。

前期功能验证阶段，将采用方案一实现卷积加速器的调度。在该阶段中，对卷积加速器的调度控制部分避免使用一切 SDK 函数库，从最底层操作寄存器读写的方式进行控制与调度，便于后期向软核 CPU Cortex-M3 的转移；后期，将逐步摆脱硬核 CPU 的依赖，采用方案二，使用软核 CPU 实现对卷积加速器的控制。

事实上，初步的验证证明，使用软核 CPU 或硬核 CPU 对卷积加速器进行调度控制，性能相差无异。在剩余的报告中，该系统默认使用软核 CPU Cortex-M3 实现硬件加速器的控制。

6 系统模块的搭建

系统模块的搭建分为：摄像头显示器回路搭建，ARM Design Start Cortex-M3 片上系统搭建，硬件加速系统的搭建三个部分。

6.1 摄像头显示器回路搭建

前期功能验证阶段，采用 Genesys ZU 3EG 的 MIPI 摄像头采集，DisplayPort 显示器回显的方案。

Digilent 官方推荐的 Pcam 5C 摄像头，通过 MIPI 接口连接至 Genesys ZU 3eg 开发板。Pcam 5C 采集到的视频流为 RAW 格式的数据，需要经过 Sensor Demosaic 与 Gamma LUT 两个 IP 核进行校正。校正得到的视频流通过 VDMA IP 核多帧缓存机制缓存入 DDR，VDMA 输出的稳定的视频流数据由 Video Timing

Controller 控制时序，通过 AXI-Stream To Video Out IP 核，输出实时视频流至 DisplayPort 接口进行回显。

通过 PS 端对上述 IP 核进行配置，即可完成摄像头采集->显示器回显回路的搭建。

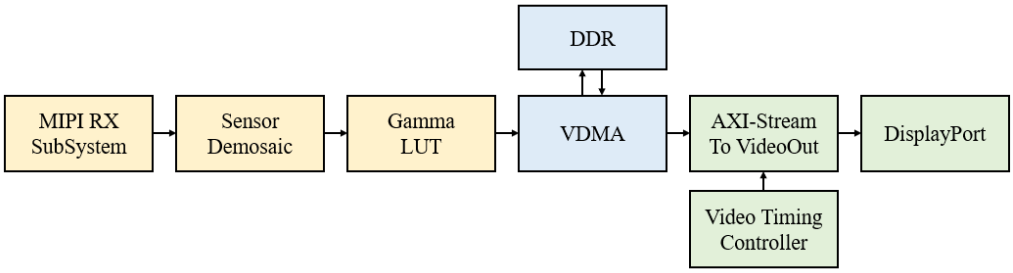


图 3 MIPI 摄像头->DisplayPort 显示器的采集显示回路

MIPI 摄像头与 Displayport 显示器回路最终搭建的 Block Design 如下所示。

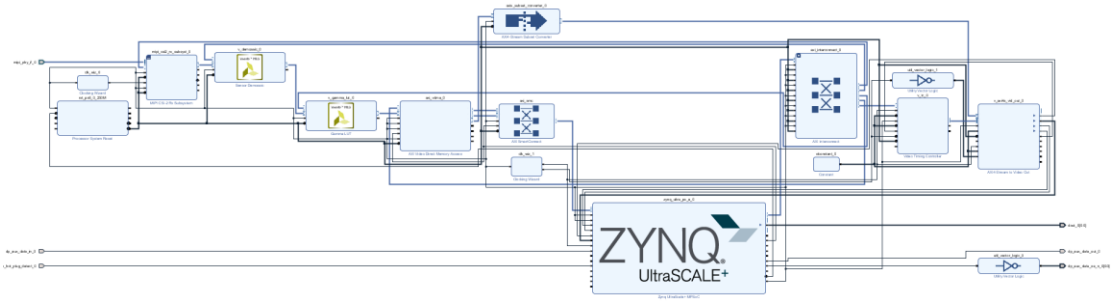


图 4 MIPI 摄像头->DisplayPort 显示器的 Block Design 设计

事实上，若使用 OV5640+HDMI/VGA 实现采集回显回路，搭建流程与上述设计相差不大。OV5640 直接输出 RGB565 数据，可以去除 Sensor Demosaic 与 Gamma LUT 两个 IP 核的校正。另外，采用不同的视频输出接口，或者输出不同分辨率的图像，都需要改变 Video Timing Controller 的具体参数。在此则不进行赘述。

6.2 ARM Cortex-M3 软核的系统搭建

ARM Cortex-M3 软核系统的搭建参照了官方手册的说明，并进行了一定的简化，删减了许多非必要的功能。Cortex-M3 片上系统必要的模块包括：时钟与复位，SWD 调试接口，AXI 总线桥与连接的外设三个部分。如下图所示。当前，总线桥所连接的外设模块仅有 GPIO 与串口模块。在后续的硬件加速系统的搭建中，

将会添加更多的模块。

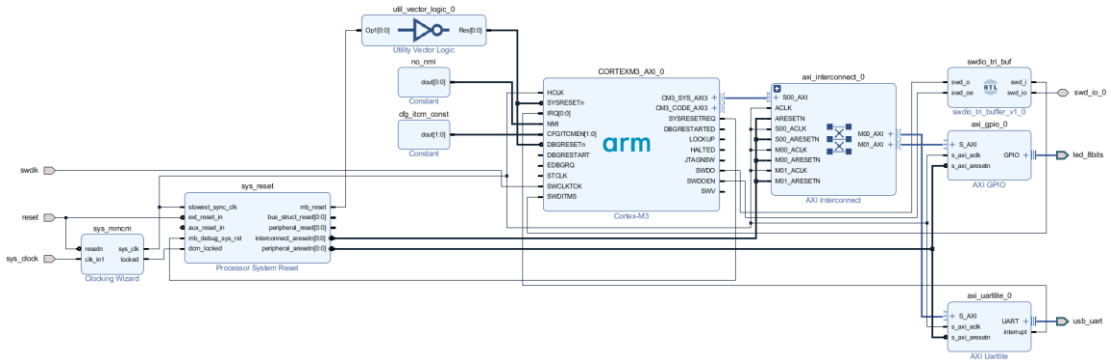
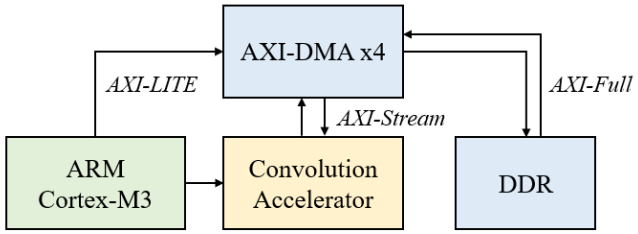


图 5 ARM Cortex-M3 片上最小系统的搭建

6.3 硬件加速系统的搭建

硬件加速系统由 RTL 描述的卷积加速器，AXI-DMA IP 核与 ARM Cortex-M3 软核 CPU 组成。其中，卷积加速器负责运算加速，AXI-DMA 核负责数据的传输，软核 CPU 负责任务的调度与寄存器的配置。系统框图如下所示。



Cortex-M3->Accelerator&AXI-DMA

图 6 硬件加速系统框图

四个 AXI-DMA 与卷积加速器通过 AXI Interconnect 与 Cortex-M3 的数据总线相连，由 Cortex-M3 进行控制。四个 AXI-DMA 的 Memory Map 端口通过 AXI Smartconnect 与 Zynq 的 AXI HP 口相连，实现对 DDR 的读写。如下图所示。

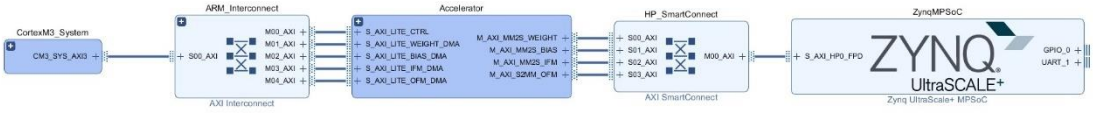


图 7 硬件加速系统 Block Design 的 Interface Viewer

ARM Cortex-M3 软核与卷积加速器的搭建最终的 Block Design 如下图所示。

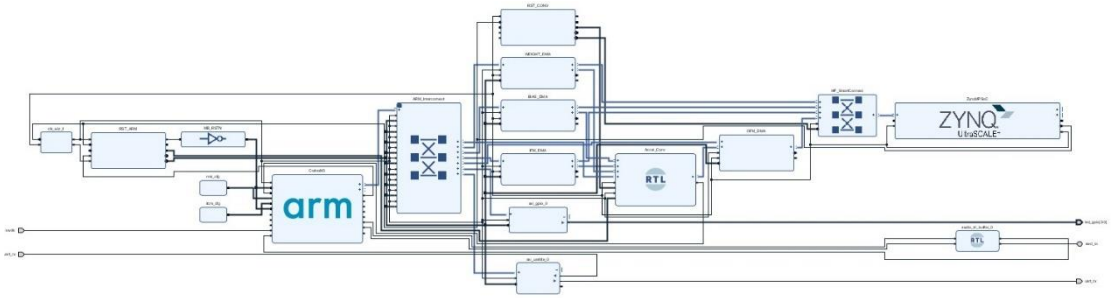


图 8 硬件加速系统 Block Design 的 Default Viewer

7 硬件加速器的详细设计

硬件加速器的设计是整个系统的核心。

该系统中卷积加速器的设计，是整个系统的最大特色与亮点。为了设计该卷积加速器，前期开展了深入的调研，中期经过了数月的开发，后期经过了无数的测试迭代更新。

该卷积加速器由软核 CPU 进行调度，硬件电路进行加速，以流的形式进行数据的输入输出，能够通用于目前主流的深度神经网络模型的推理。

整个加速器框架由四个部分组成：软核 CPU ARM Cortex-M3，四个负责数据传输的 AXI-DMA IP 核，外部的 DDR3/DDR4 存储器，与 RTL 语言描述的卷积加速器。

软核 ARM Cortex-M3 负责的四个 AXI DMA IP 核与卷积加速器的寄存器配置。IFM,BIAS,WEIGHT AXI-DMA 三个 DMA 核通过 AXI 总线连接到 AXI HP 口或者 Memory Interface Generator，将外部 DDR3/DDR4 的数据转换成 AXI-Stream 形式的流式数据，输入进卷积加速器；而卷积加速器计算得到的流式数据，通过 OFM AXI-DMA IP 核回存外部 DDR3/DDR4。系统框架如下所示。

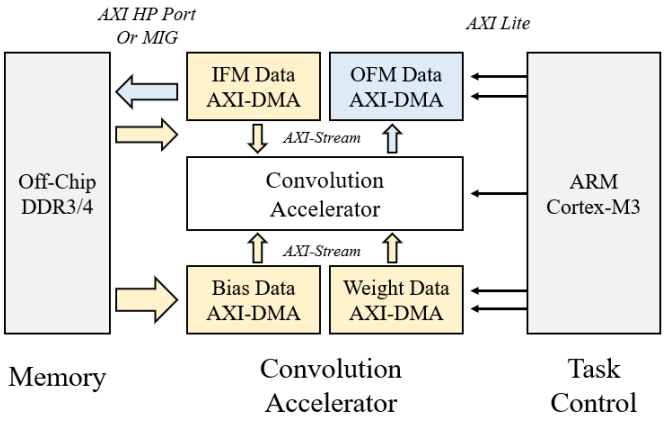


图 9 基于软核调度的流式卷积加速器

该卷积加速器的实现参考了论文[1] Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA 和论文[2] Going Deeper with Embedded FPGA Platform for Convolutional Neural Network 中的卷积核架构，并在此基础上进行了改进和创新。

该卷积加速器采用 Verilog 纯 RTL 语言实现，对卷积中的乘加运算进行了细粒度的优化，对加速并行的架构进行了深入的探索，在消耗相对较少的资源的情况下，能达到较高的性能。下面，将详细介绍该卷积加速器的设计。

7.1 充分利用硬件乘法器

乘法是卷积的基本运算。卷积神经网络在计算终端的部署，往往需要经过量化，压缩模型大小，将浮点运算转变成定点数的运算，从而提升推理速度。INT8 线性量化，则是最常见的一种量化方法。

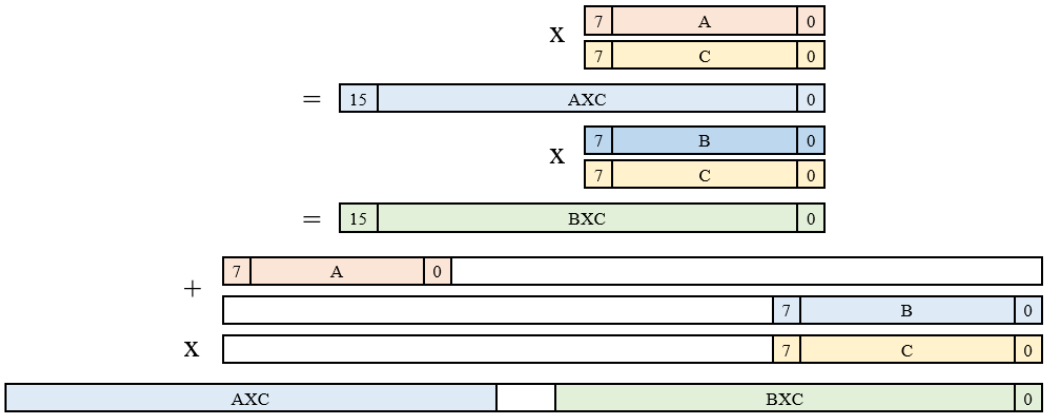
卷积神经网络经过 INT8 量化后，所有的乘法运算的都以 INT8 形式进行。Xilinx 的 FPGA 片上的硬件乘法器资源，能完成比 INT8 数据位宽更大的乘法。例如 7 系列的 FPGA 拥有 DSP48E1 资源，能够完成 25 位乘 18 位的有符号数的乘法。Zynq Ultrascale 系列的 FPGA 拥有 DSP48E2 资源，能够完成 27 位乘 18 位有符号数的乘法。

可见单个 DSP48，完成单个 INT8 乘法运算还有许多冗余位没有被充分利用。我们可以合理地利用 DSP48 的特点，将两个 INT8 乘法，映射到只有一个乘法单

元的 DSP48 上。

两个 INT8 乘法映射到一个 DSP48 的前提是两个乘法具有一个相同因数。在卷积运算中，共享一个相同的因子的运算非常常见。例如，同一个 IFM 与不同的一套权值相乘，IFM 数据则作为共享的因子；同一套权值与不同的 IFM 相乘，该权值则作为共享的因子。

假如要进行两个 INT8 乘法 AXB 和 DXB，二者等效于(A+D)XB，这就可以利用到 DSP48 中的预加器。但同时还要注意到，我们最终期望得到的是两个乘积 AXB 和 DXB，而不是乘积和。因此，我们按如下图所示操作。以 Ultrascale 的 DSP48E2 为例：DSP48E2 的端口 A 为 27 位，将其高 9 位填充数据 A，并对 a 做了符号位扩展，由 8 位变为 9 位，低 18 位填充 0。DSP48E2 的 D 端口为 27 位，将数据 D 进行符号位扩展，低 9 位填充数据 D。DSP48E2 的 B 端口为 18 位，将数据 B 进行符号位扩展，低 8 位填充数据 B。最终，DSP48E2 的 48 位 P 端口将输出 AXB 和 DXB，如下图所示。



Implemeting Two INT8 Multiplication
Using a Single DSP48E1/E2

图 10 将两个 INT8 乘法映射到一个 DSP48 中

7.2 多个操作数加法的方案选择

加法是卷积的另一个基本运算。卷积运算中，往往需要进行多个数的相加。

譬如，一个 3 乘 3 的卷积窗口经过乘法运算后，需要进行一个 9 个操作数的加法，才能得到最终的输出值；多个输入通道卷积计算出来的结果，需要经过相加才能得到一个输出通道。

多数相加有多种方法。方法一，多个操作数直接相加。加法器所在路径，将是时序收敛的瓶颈。方法二，通过加法链相加，使用多级寄存器保证数据的对齐。方法三，通过加法树相加，降低逻辑级数。

三种方法中，直接相加消耗的 Flipflops 少，消耗的 LUT 最多，逻辑级数最高，时序最差；加法链相加消耗的 Flipflops 最多，逻辑级数最小，时序最好；而加法树的性能在二者之间。最终，选取加法树实现基本的多数相加单元。

在该卷积加速器中，9 个操作数的相加操作最为常见：3 乘 3 的乘积和，8 输入通道+Bias 的求和，都是 9 个操作数的相加操作。使用“三叉树”结构完成 9 数相加，相比起“二叉树”结构能减少一个时钟的延时，并且大大减少了 LUT 和 FF 的消耗，时序也不至于有太大影响，如下图所示。很明显，三路加法树在该卷积加速器中更加具有优势。

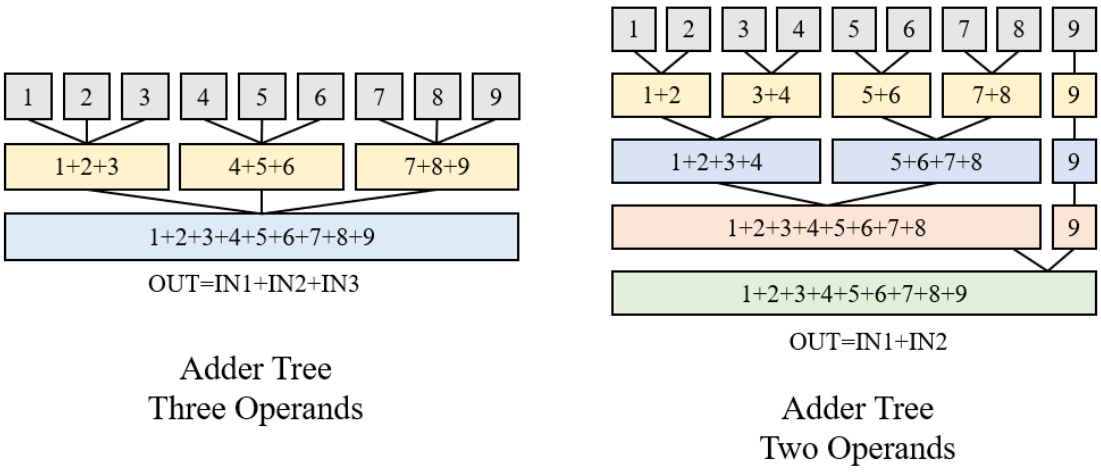


图 11 两种类型的加法树结构

7.3 深度可动态配置的行缓冲机制

行缓冲机制在 FPGA 数字图像处理中，使用非常频繁。图像过程中经常遇到需要对于像素按照行对齐的输出，例如图像的均值滤波，中值滤波，高斯滤波以及 Sobel 边缘查找等，这些算法都需要行缓冲设计。

行缓冲机制能够使得数据在单端口以流水形式单行输入时，通过缓存历史输入数据，实现多行数据并行输出，从而实现并行化操作。例如一副图片每行有 M 个数据，则 Linebuffer 当中每行则有 M 个缓存，如若要实现 N 行数据并行输出，

则需要 N 行缓存。其在 MX（N-1）个时钟周期后，则可以实现 N 行数据同步并行输出。

而卷积神经网络的计算基本运算单元与图像处理中的均值滤波，中值滤波，高斯滤波以及 Sobel 边缘查找非常类似，都需要像素的按照行对齐，并实现数据并行输出，所以，行缓冲机制是卷积加速器设计的常见基本架构，如下图所示。该卷积加速器的设计，也是基于行缓冲机制进行展开。

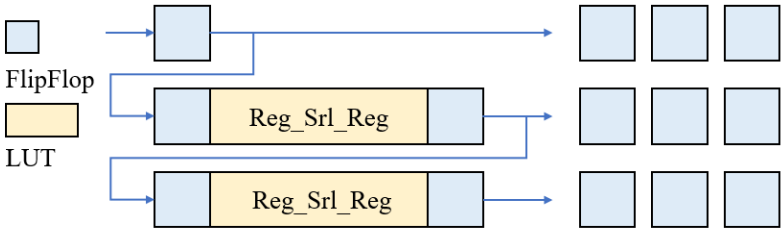


图 12 基本的行缓冲结构

然而，传统的简单的 LineBuffer，往往只能实现固定大小的特征图的多行对齐。LineBuffer 最终将映射成 FPGA 中的 RAM，不同大小的特征图则需要不同大小的 RAM 进行行缓冲。FPGA 本身的特点，决定了无法为不同大小的特征图分配不同大小。

在图像处理中，往往是对固定大小的图像进行处理，对缓冲深度可配置的 LineBuffer 没有迫切的需求；而在一个深层的神经网络卷积运算中，往往具有多种不同大小的特征图。于是，在该卷积加速器设计中，对传统的行缓冲结构进行了改进扩展，使其更加适合应用于卷积神经网络的推理计算中。

当下，主流的神经网络往往采用较小的卷积核，如 3X3，1X1 的卷积核，并对输入特征图进行 Padding，使得输出特征图大小与输入特征图相同。并且，2X2 且 Stride 为 2 的池化运算最为常见。如 VGG16 与以 VGG16 为 BackBone 的众多卷积神经网络。这样，整个神经网络的推理中的特征图的所有大小类型，其实并不算太多，且特征图的大小随着层数的递增，在每一次 2X2 的池化中特征图尺寸以 1/2 递减。这样，我们就可以根据我们的神经网络模型，例化深度最长的 LineBuffer，并且在中间逐级抽头，通过 Multiplexer 来配置当前卷积中所需要的 LineBuffer 深度，从而匹配当前的特征图的尺寸，具体结构如下图所示。

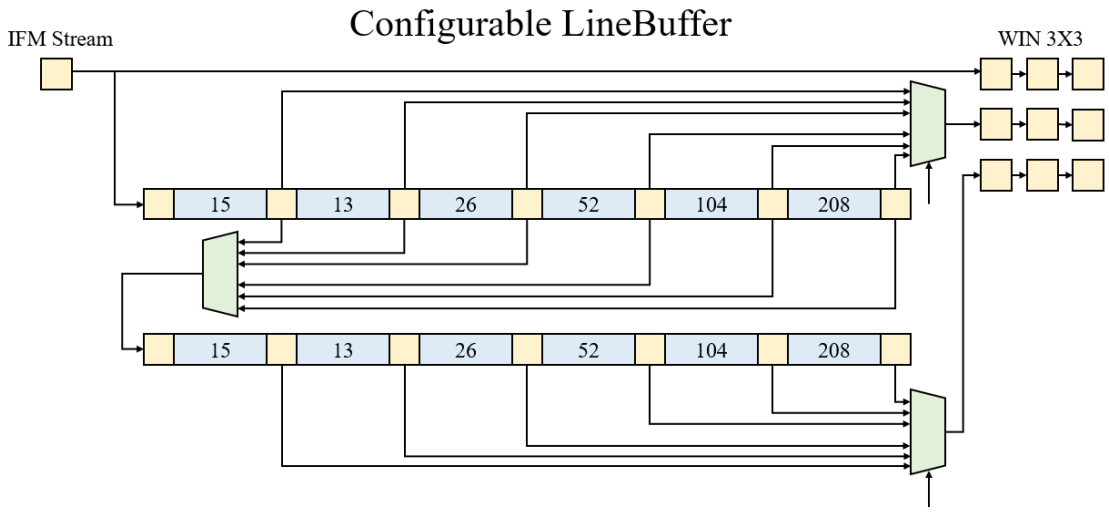


图 13 缓冲深度可动态配置的行缓冲机制

上图的 LineBuffer，就根据 Yolov3-Tiny 的 BackBone，大小分别为[13,26,52,104,218,416]的特征图进行配置。通过多路选择器选通不同的 Datapath，从而实现缓冲深度的可配置。

7.4 大尺寸特征图的分块策略

特征图从片外的 DDR 内存输入进来后，需要缓存至 FPGA 片上存储器资源。而片上存储器资源往往较小，无法存储较大尺寸的特征图。以 Yolov3-Tiny 的输入特征图为例，尺寸为 416X416X3,将一个完整的特征图存入片上，最少就得需要 120 个 BRAM36K，如果采用 PingPong Buffer，则需要 240 个 BRAM36K。Artix 7 系列容量最大的芯片，最多也只有 316 个 BRAM36K。所以，FPGA 片上存储器，往往只能存储较小的特征图，而较大的特征图，则可以通过分成多块的形式逐块送入片上存储器进行卷积。

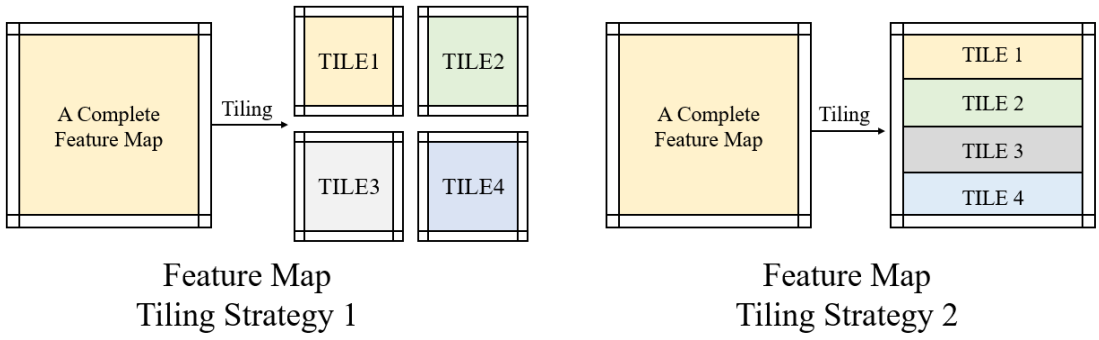


图 14 两种大尺寸特征图的分块策略

特征图分块的策略有两种，如上图所示。第一种是“方形分割”，将一个大的特征图，均分成多个等大的较小的方形，送入片上 RAM。

这一种方法适合固定深度的 LineBuffer 架构。例如，在某个卷积核的设计中，采用了固定深度的 LineBuffer 架构，只能进行特征图大小为 28X28 的卷积。大于 28X28 的特征图，则可以通过策略一的分块方式，等分成若干个 28X28 大小的特征图。该方法的缺点在于，分块的特征图地址不连续，且分割前要对每一个块进行 Padding，否则每一小块的边缘像素计算结果有误。不连续的寻址与 Padding 操作，对 FPGA 并不友好，往往需要强力的硬核 CPU 辅助，如 Zynq 系列的 FPGA。

第二种策略则更加适合于本卷积加速器的设计。由于本卷积加速器设计了深度可配置的 LineBuffer 架构，无需将特征图分块成固定的大小。

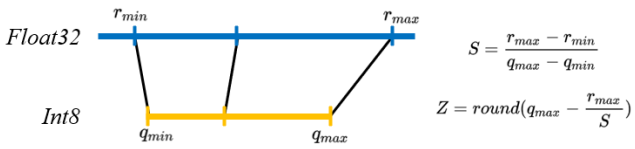
在这种情况下，对于较大的特征图，我们可以进行“矩形分块”，每次只进行若干行的卷积。例如，片上存储器只能够存储 52X52 大小的特征图，而在进行特征图尺寸为 104X104 的卷积时，则课将该特征图分为四个 104X26 的矩形，分别送入 FPGA 进行卷积。该策略的优点在于，特征图分块后的地址连续，进行 Padding 时，边缘像素的地址就在分块的一前一后，在 FPGA 上无需复杂的寻址，更加易于实现。

7.5 模型的量化与量化推理

主流的深度学习框架下训练得到的模型，Float32 的数据类型最为常见。而在 FPGA 上进行浮点运算需要较大的开销，往往需要将浮点数量化成定点数进行运算。众多研究发现，在神经网络的前向推理中，将原本 Float32 进行推理的卷积运算量化成 16 位甚至 8 位，并不会对推理结果的准确率有较大的损失。所以，在能接受精度稍微下降的 trade-off 下，量化往往是 FPGA 神经网络推理的必要步骤。

目前主流的深度神经网络框架，如 Pytorch 与 Tensorflow，皆有神经网络量化的 API 可供调用。本卷积加速器的设计，能对 Pytorch 框架下的 Post Trainning Static Quantization 的量化方法量化得到的网络模型，进行 INT8 的推理。接下来将详细介绍该量化方法的原理，并详细讲解 FPGA 上实现该量化推理的具体计算细节。

Pytorch 框架的 Post Trainning Static Quantization 是一种线性量化。该线性量化的原理，论文[3]中进行了详细的介绍。量化之前，先在每一层都插入 Histogram Observer,并为网络模型准备一定量的输入数据样本进行前向推理；在推理过程中，Histogram Observer 记录下数据的分布情况，并计算出线性量化的相关参数;在输入一定量的样例后,即可根据观察得到的量化参数伸缩因子，将 Float32 模型压缩成 INT8，如下图所示。



$$q_3^{i,k} - Z_3 = \frac{S_1 S_2}{S_3} \sum_{j=1}^{N} (q_1^{i,j} - Z_1)(q_2^{j,k} - Z_2)$$

INT8 Linear Post Traing Static Quantization

图 15 INT8 线性量化原理

其中 r 为浮点数，S 为伸缩因子，q 为量化后的整数，Z 为量化后的零点的位置。

量化之后模型的推理过程，可以由以下的例子进行说明。卷积运算是一种乘累加运算，我们以用下面的式子对其进行表示。

$$r_3^{i,k} = \sum_{j=1}^{N} r_1^{i,j} r_2^{j,k}$$

将上述式子的浮点数替换成量化后的整数，再稍作变换，得到以下的式子。

$$S_3(q_3^{i,k} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{i,j} - Z_1) S_2(q_2^{j,k} - Z_2)$$

$$q_3^{i,k} - Z_3 = \frac{S_1 S_2}{S_3} \sum_{j=1}^{N} (q_1^{i,j} - Z_1)(q_2^{j,k} - Z_2)$$

可见上述式子中除了 M=S1*S2/S3 是浮点数之外，其它的运算都是 INT8 的运算。根据浮点数本身的存储方式，因子 M 可以通过 frexp 函数，把该浮点数分解成尾数和指数。

$$M = \frac{S_1 S_2}{S_3}$$

$$M = 2^{-n} M_0$$

这样，与 M0 的乘积可以转换成定点运算，与 2 的负 n 次方的乘积可以转换成右移运算。经过上面的推导可以发现，整个量化后模型的推理，都转换成了定点运算。

然而，在 FPGA 实现上述计算，需要注意若干细节：

1 卷积运算中，INT8 与 INT8 的乘法得到 INT16，理论上在进行累加的时候，需要做 INT32 的累加运算。但是在大量实验观察发现，量化后的模型在 INT8 相乘累加之后，往往远小于 INT32 能表示的最大值。事实上，INT8 与 INT8 的乘法得到 INT16 进行累加，只需要 INT18 就能保证绝大部分的激励值不溢出。所以，为了节约 FPGA 的资源，该卷积加速器仅用 INT18 做累加。

2 所有的累加结束后，得到的 INT18 需要经过伸缩变换，重新变回 INT8。伸缩变换的过程，做了以下的操作：

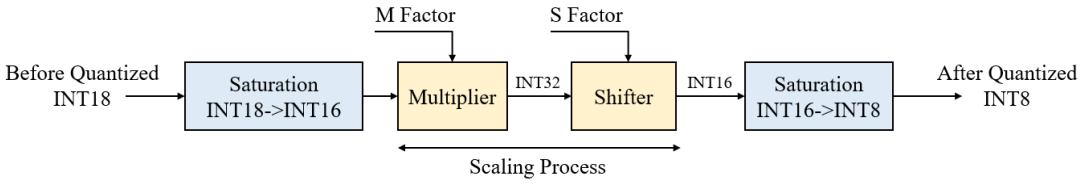
首先，做饱和处理将 INT18 饱和截断至 INT16；

其次，与 UINT16 的 Scale_Factor M0 系数相乘，得到 INT32；

接着，“向最近的整数舍入右移”（15+Shift_Factor（即 n)），重新得到 INT16；

最后，对 INT16 再做饱和处理，得到 INT8。

上述的操作，能与 Pytorch 的量化推理结果基本保证一致，如下图所示。



INT8 Linear Quantized Inference

图 16 INT 量化推理过程

3 “向最近的整数舍入右移”是一种特殊的右移操作。普通的右移，是向负无穷舍入，而如果直接除以 2 的（15+Shift_Factor）次方后做 rounding 再转成 INT8，是向最近的整数舍入。实验证明，二者在最终的计算误差会逐级累加，结果会有很大的偏差。所以，需要设计一种特殊的“朝零舍入右移”。

解决方法的思路非常的朴素，即在右移丢失后（15+Shift_Factor）位的数据之前，记录下 INT32 乘积结果中第[15+Shift_Factor-1]位的值。若该值为 1，则最终右移的值需要加 1；若为 0，则计算结果不变。这样的操作即可将普通的右移的向负无穷舍入，变成向零舍入，从而保证了计算的正确性。

7.6 流水运算

上述所介绍的乘法运算，加法树运算，行缓冲机制，量化计算，池化计算等，皆为非阻塞的流水运算，数据从 IFM Buffer 流向 OFM Buffer，产生的延时为：Total Latency=LineBuffer 的延时+Multiplier 的延时+Adder Tree x2 的延时+Accumulate 的延时+Quantization 的延时+Pooling 池化的延时。每个时钟接受一个数据并输出一个数据，实现较高吞吐率。

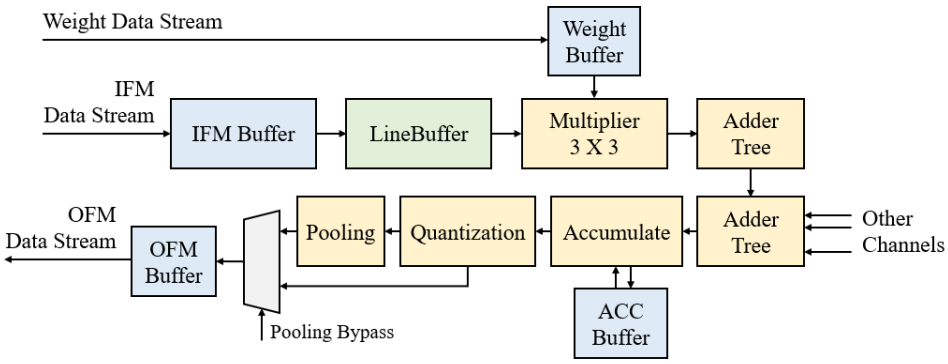


图 17 流水运算

7.7 并行维度的探索

卷积运算的加速，主要体现在“并行”。使用 C 语言对卷积运算进行最原始朴素的描述，如下面的代码所示。对于下面的六重循环进行不同方式的展开，则可以实现不同的并行策略[5]。

```
for(int o=0;o<CHO;o++)                                  // ofm
    for(int i=0;i<CHI;i++)                              // ifm
        for(int x=0;x<OFM_SIZE;x++)                     // ofm pixel rows
            for(int y=0;y<OFM_SIZE;y++)                 // ofm pixel cols
                for(int u=0;u<KERNEL_SIZE;u++)          // kernel rows
                    for(int v=0;v<KERNEL_SIZE;v++)      // kernel cols
                        out[o][x][y]+=in[i][x+u][y+v]*weight[u][v];
```

图 18 卷积运算的 C 代码描述

在本卷积加速器的设计中，对[kernel rows]和[kernel cols]这两个维度进行了展开，对[ofm]和[ifm]这两个维度实现了并行。

[kernel rows]和[kernel cols]这两个维度上的展开，是通过 LineBuffer 机制，在每一个时钟到来时产生一组与 Kernel 相同尺寸的数据，并例化相应数目的乘法器和加法器来实现的。例如卷积 Kernel 为 3X3，则 LineBuffer 机制可以在每个 Clk 到来时产生一个窗口大小为 3X3 的数据，并例化 9 个乘法器数据与权重对应相乘，通过加法树对 9 个数据进行相加，最终得到输出数据流。如下图所示。

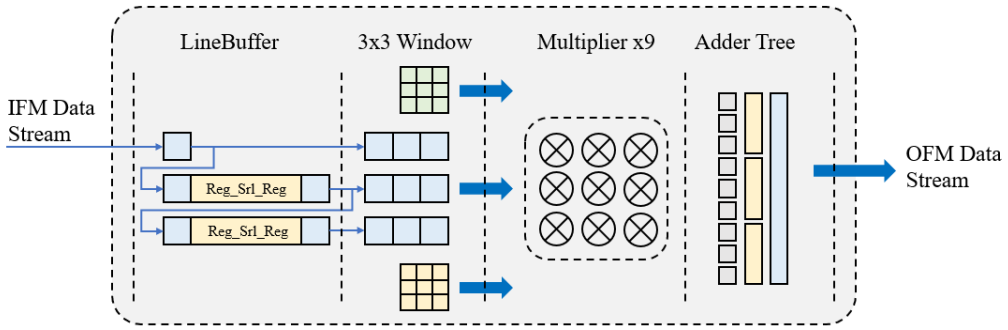


图 19 [kernel rows]和[kernel cols]维度的展开

无论是 LineBuffer，乘法器还是加法树，都采用了流水的结构，能在每一个时钟都能接收一个 IFM Data，并输出一个 OFM Data，中间计算无阻塞，从而实现[kernel rows]和[kernel cols]这两个维度上的展开。该卷积模块的总 Latency=行缓冲的时延+乘法器的时延+加法树的时延。

[ofm]和[ifm]这两个维度的并行，则要综合考虑输入输出数据的位宽，片上资源的开销，存储读写带宽的占用，与数据的具体组织形式。

首先，考虑输入输出数据的位宽。由于模型经过量化后，数据以 INT8 的形式进行卷积运算，单个通道的数据位宽为 8。若采用 AXI DMA 来实现流式数据的传输，AXI DMA 可以支持[8,16,32,64,128]等位宽的数据传输，分别可以实现[1,2,4,8,16]个通道的数据的同时传输。充分利用好 AXI DMA 的数据位宽，设计合适的并行度与 AXI DMA 的带宽进行匹配，则可以提高内存带宽的利用率，简化设计。

其次，考虑片上资源的开销。并行，其实就是多个相同卷积模块的例化，根据单个卷积模块的资源耗用情况，对比片上的资源总量，可以大致估算出可以实现的最大并行度。卷积运算中，最关键的资源是乘法器资源 DSP48。若在[kernel rows]和[kernel cols]这两个维度上进行展开，默认为 3X3 的卷积核大小，单个

卷积单元则需要耗用掉 9 个 DSP。假设片上总共有 360 个 DSP，保留 20%的余量，则剩余的 288 个 DSP 能够最多支持例化 32 个上述的卷积单元。若每个卷积单元能够完成两组 INT8 的卷积运算，则最大的并行度为 32X2=64，即在每一时刻，可以同时进行 64 组卷积运算。

考虑以上两个因素之后，基本就可以根据具体的硬件资源条件，确定并行策略。在本卷积加速器的设计中，采用了 64 位的输入输出带宽，可以同时输入输出 8 组特征图。并行度设置为 8X8=64，每一时刻同时进行 64 组卷积运算。

64 组卷积运算，就要为 64 组卷积单元分配独立的权重。权重数据同样的通过 AXI DMA 以 64Bits 流水形式传入。64Bits 的数据分成 8 组，每组数据再通过串转并的方式，分配到 8 个 Block RAM，从而实现 8X8=64 组权重数据的分配。对于 3X3 的卷积核，将权重展平并拼接，将得到 72 位的数据。72 位的数据，也正好匹配 BRAM36K 的内部硬件结构，能实现无冗余的存储。

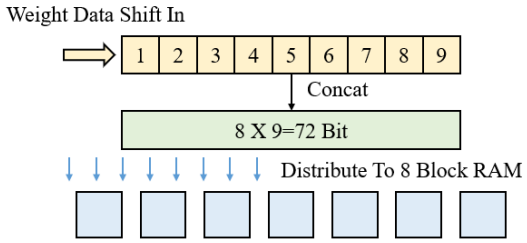


图 20 单个通道的权重数据 串转并存入 8 个 BRAM

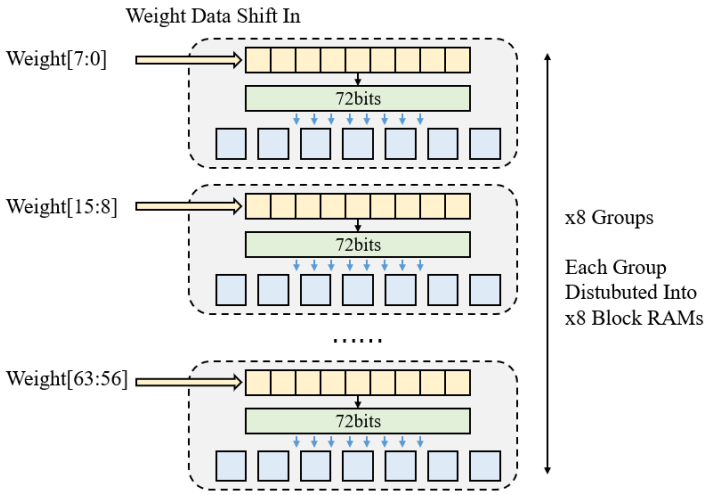


图 21 多个通道的权重数据 串转并存入 8X8=64 个 BRAM

64 组卷积运算，分成 8 组，每一组共享 8 路输入特征图数据。每组的输入数据相同，但权重不同，则可以对应产生 8 组输出特征图的数据。如下图所示。

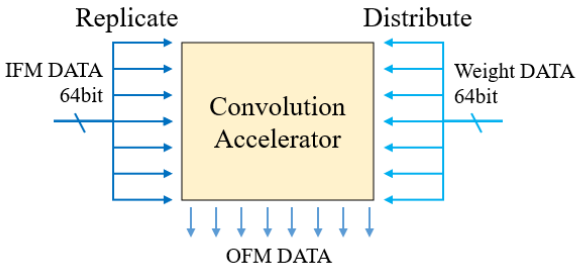


图 22 “8 进 8 出”的并行策略

7.8 软核 CPU 的调度与控制

卷积加速器仅完成卷积计算，需要外部对其进行调度与控制。理论上，可以通过设计状态机来进行控制，但是这样将会增加设计的复杂度。通过外部构建 SoC 系统，搭载软核 CPU，通过 AXI-Lite 总线对卷积加速器与 AXI-DMA IP 核进行控制，则会更加灵活轻便。

该卷积加速器有 7 个寄存器。软核 CPU ARM Cortex-M3 通过 AXI-Lite 总线对这 7 个寄存器进行读写，从而实现对卷积加速器的控制。

Accelerator Control Reg Address: 0x00

0	1	2	3	4	5	6	7	8	9	Reserved

0 IFM Recv Start
1 Weight Recv Start
2 Bias Recv Start
3 OFM Send Start
4 Conv Start
5 Pooling Enable
6 First Conv Flag
7 Last Conv Flag
8 Pingpong Buffer Sel
9 Task Valid

图 23 卷积加速器的寄存器 1

第一个寄存器是控制寄存器。前 5 位为控制字，分别控制 IFM 数据，Weight 数据，Bias 数据的接收，OFM 数据的发送，和卷积运算的使能。后五位为状态字，分别指示当前的卷积结果是否需要池化，是否是第一次或者是最后一次卷积，Pingpong Buffer 的指针指向的内存区域，和当前数据是否有效。

由前 5 位的控制字可见，该卷积加速器有 5 种类型的任务。每一种任务使能之后，需要外部进行手动复位。任务完成之后，该卷积加速核通过拉高 AP_DONE 信号触发软核 CPU 的中断，告知 CPU 完成当前设定的任务，从而实现反馈。

Convolution Parameter Reg Address: 0x04

[0:15]	[16:27]	[28:30]

[0:15] Convolution Data Length
[16:27] Column Length
[28:30] LineBuffer Type Select

图 24 卷积加速器的寄存器 2

第二个寄存器用于设置卷积运算的具体参数。前 16 指示要对缓存中长度为多少的数据进行卷积，后 12 位指示当前的特征图一行有多少个数据。最后三位指示当前特征图所对应的 LineBuffer 的类型。

Quantization Parameter 1 Reg Address: 0x08

[0:15]	[16:19]	Reserved

[0:15] Scaling Factor: Mult
[16:19] Scaling Factor: Shift

图 25 卷积加速器的寄存器 3

第三个寄存器用于设置量化参数中的 Scaling Factor。

Quantization Parameter 2 Reg Address: 0x0C

[0:7]	[8:15]	Reserved

[0:7] Input Zero Point
[8:15] Output Zero Point

图 26 卷积加速器的寄存器 4

第四个寄存器用于设置量化参数中的输入零点与输出零点。

Address Reg Address: 0x10-1C

[0:31] Weight Read Address
[0:31] Bias Read Address
[0:31] OFM Send Address

图 27 卷积加速器的寄存器 5 6 7

第五，第六，第七个寄存器提供卷积加速器所需的一些地址信息。

通过配置这七个寄存器，可以实现对该卷积加速器的完全控制。

但同时，软核 CPU ARM Cortex-M3 还需要对 AXI-DMA 核进行配置，来实现 IFM 数据，Weight 数据，Bias 数据的接收，和 OFM 数据的发送。查阅 AXI-DMA 核的数据手册后发现，若需要对其进行简单控制，完成对 DDR 的读写操作，只需要进行四个寄存器的读写。下面是通过 DMA 对 DDR 进行读操作的代码。

```
AXI_Out32(DMA_BASEADDR,0x04);
AXI_Out32(DMA_BASEADDR+SRCADDR_OFFSET,bufaddr);
AXI_Out32(DMA_BASEADDR+CR_OFFSET, 0x10003);
AXI_Out32(DMA_BASEADDR+BUFFLEN_OFFSET,buflen);
```

图 28 使用底层寄存器操作读写的方式控制 DMA

第一行通过对 DMA 基地址写入 0X04 实现对 DMA 的复位。上电之后，只需对 DMA 进行一次复位；第二行配置对 DDR 读取的首地址；第四行配置对 DDR 读取的总长度。

通过 DMA 对 DDR 进行写入的配置操作与上述的描述类似。

7.9 资源耗用率与性能指标

该卷积加速器的设计耗用情况如下所示

	LUT	FF	BRAM36K	DSP48	LUTRAM
Accelerator	24K	16K	40	296	14K
Cortex-M3	10K	3.5K	16	3	0
AXI-DMA x4	5K	8K	34	0	0.5K
Total	30K	27.5K	90	299	14.5K

表 1 卷积加速器的资源耗用情况

该设计在 Digilent Genesys ZU 3eg 的开发板上得到验证，资源占用率情况如下所示

	LUT	FF	BRAM36K	DSP48	功耗
Total	70K	14K	216	360	3.915W
Utilization	60%	23%	41%	83%	

表 2 卷积加速器的资源占用率

实测该卷积加速器的性能如下所示

指标	性能
最高时钟频率	超过 250MHz
峰值 GOP/s	144GOP/s
平均 GOP/s	73GOP/s

表 3 卷积加速器的性能指标

以 VGG16 的 BackBone 做测试，该卷积加速器的性能如下

	IFM Channel	OFM Channel	FM SIZE	Total OP	Time(ms)
Test 1	64	64	224	1.85G	25.3ms
Test 2	64	128	112	0.92G	12.4ms
Test 3	256	256	56	1.85G	23.1ms
Test 4	512	512	30	1.85G	30.7ms
VGG16 BackBone					230ms

表 4 性能测试

现阶段，该卷积加速器完成了初步的功能性测试。上述的测试结果可以发现，该卷积加速器性能较强，对于多输入输出通道的不同尺寸的卷积运算，皆能有较明显的计算加速效果。

后续的工作中，在完成完整的功能性测试后，将会部署 Yolov3-tiny 目标检测算法，实现简单的实时目标检测任务。

8 系统集成测试

8.1 软核基本功能测试

赛题要求能够通过 SoC 的数字 GPIO 在外部硬件逻辑分析仪仪器上（虚拟仪器）显示出对应 SoC 的内核名称。

在 AXI 总线桥上挂载 AXI-GPIO 外设，采取 System ILA 直连 AXI-GPIO 的输出端口，程序烧录完毕后进行波形的抓取。实现定义字模，存储进 CORTEX-M3[][]数组；软核实现 Cortex-M3 字样的程序可以通过对数据口的循环输出进行实现；对同一个地址多次读写时，需要采用 Volatile 关键字避免编译器的“过优化”，否则无法输出正确的波形。

具体的程序与结果如下图所示。

```
void Display_CortexM3(void){
  int i,j,t;
  for(i=0;i<ROW;i++){
    for(j=0;j<COL;j++){
      for(t=0;t<6;t++){
        *(volatile uint32_t *)(ILA_BASEADDR)=0x00;
        *(volatile uint32_t *)(ILA_BASEADDR)=CORTEXM3[i][j];
      }
    }
  }
  return;
}
```

图 29 实现 Cortex-M3 字样 IO 输出 C 程序

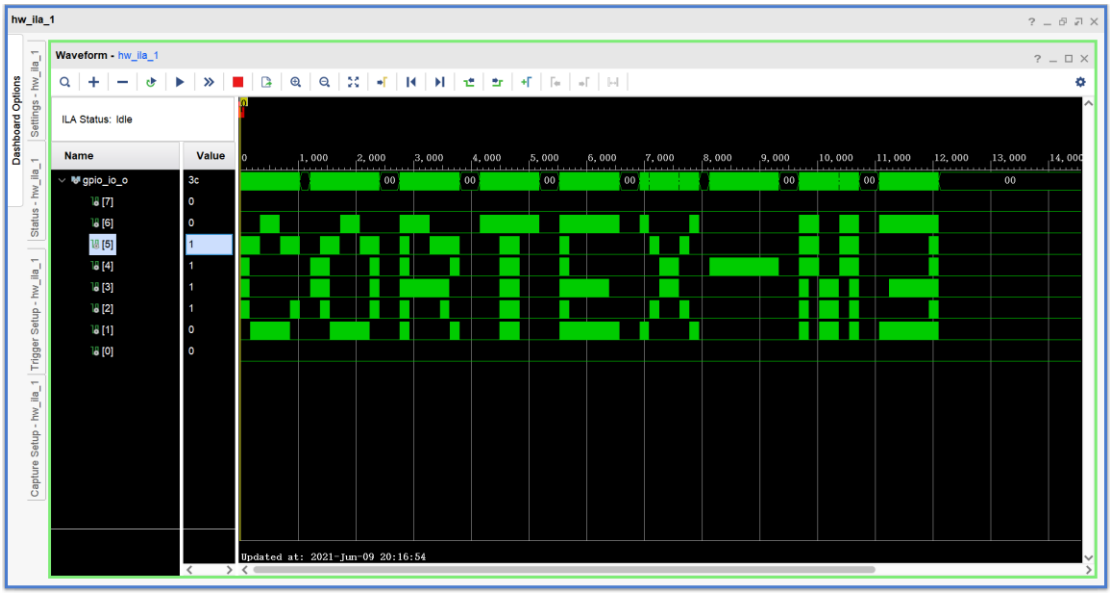


图 30 ILA 观察 Cortex-M3 字样输出

8.2 系统基本功能测试

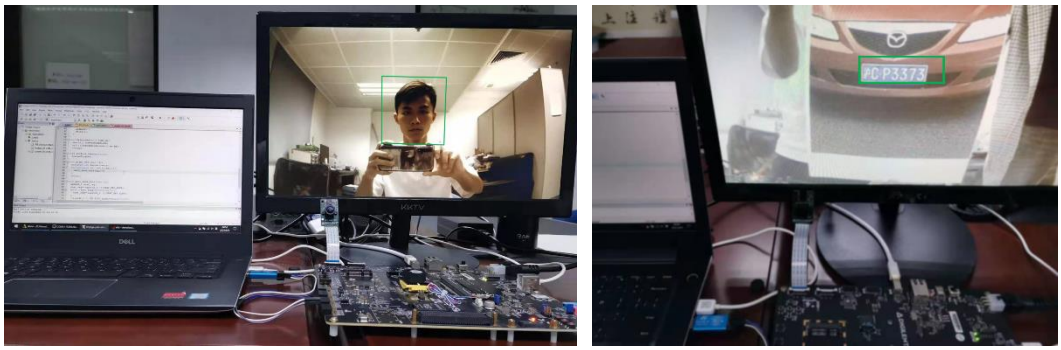


图 31 人脸识别与车牌识别的具体应用场景

YOLO 是当下最流行的目标识别与检测算法。该系统选用易于边缘计算平台

部署实现的 Yolov3-Tiny 进行验证，构建视频采集->Yolov3-Tiny 目标检测->视频回显的基本应用。

人脸识别与车牌识别，则是目标识别与检测的常见应用场景。于是便在 Yolov3-Tiny 模型的基础上自定义数据集并进行训练，实现人脸识别与车牌识别的具体应用，部署至该系统。

系统运行实物图如上所示。

9 项目创新点

9.1 提出并实现了一种新的卷积加速方案

该系统提出基于软核调度的流式卷积加速方案。

系统以 Cortex-M3 作为任务调度器，RTL 描述的卷积加速器进行加速运算，DMA 核实现流式数据的读写。三个部分各司其职，软硬件设计划分明确，互相紧密耦合，形成完整的一个系统。

9.2 卷积加速器的设计采用多种优化策略

从基本的乘加运算开始优化

合理进行乘法器与加法器的设计，从最基本的运算开始细粒度优化：充分利用片上资源，实现资源的复用，减少资源的消耗

提出了一种深度可配置的行缓冲机制

对传统的行缓冲机制进行改进，使得卷积加速器不再受限于单一尺寸的特征图卷积运算，实现了多尺寸的特征图的卷积运算。并在此机制上提出大尺寸特征图“矩形”分块策略，避免地址的跳变与 Padding 时的复杂寻址。

流水运算提高运行时钟频率

计算过程皆为流水运算，RTL 描述与具体硬件相匹配，充分利用 DSP48 内部的寄存器进行流水运算，使其达到最佳的性能，达到更高的时钟频率。

乒乓缓存机制提高数据吞吐率

数据通过乒乓缓存机制实现读入读出，在数据交互的同时也在进行卷积运算，使得数据吞吐率大大提高。

采用易于实现的 INT8 量化

选用了易于硬件电路实现的 INT8 线性量化方案，推理的全过程计算均为定点运算，无需进行反量化。

合理探索计算并行方案

充分考虑了输入输出数据的位宽，片上资源的开销，存储读写带宽的占用，与数据的具体组织形式，最终确定下来最优的 8X8 并行加速方案。

10 总结、体会与期望

该系统是完成的是一个常见的视频采集->处理->视频回显任务：摄像头采集的视频流缓存至外部存储器 DDR，卷积加速系统从帧缓存区域获取输入特征图，进行目标识别/检测神经网络模型的推理，推理结果与实时视频流在显示器共同回显。

该系统的亮点在于卷积加速系统的设计：软核 CPU 的调度+RTL 描述的加速器设计+DMA 流式数据的传输，三者构成一个有机结合的系统。该系统的难点在于卷积加速器的设计：无论是卷积计算的具体细节实现，还是数据的交互与并行方案的确定，都具有较大难度，需要较长的开发周期。

该系统从整个项目方案的制定，到所有模块的开发调试，再到最终整体系统的搭建，皆由队内唯一的一位成员单独完成。笔者先前有过在 FPGA 上部署神经网络算法的相关研究经历[4]，并在此次比赛中做了更深入的研究和探索。

该系统在 Digilent 设计的 Genesys ZU 3EG 的开发板上进行验证展示。赛后，我队愿意将系统进行开源，分享该开发板的调试经验，将卷积加速系统的具体实现细节进行开源，以实际行动奉行 Digilent“分享创新的乐趣，发现更好的自己”的理念。

参考文献

[1] K. Guo et al., "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 1, pp. 35-47, Jan. 2018, doi: 10.1109/TCAD.2017.2705069.

[2] Qiu, Jiantao & Song, Sen & Wang, Yu & Yang, Huazhong & Wang, Jie & Yao, Song & Guo, Kaiyuan & Li, Boxun & Zhou, Erjin & Yu, Jincheng & Tang, Tianqi & Xu, Ningyi. (2016). Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. 26-35. 10.1145/2847263.2847265.

[3] Jacob, Benoit & Kligys, Skirmantas & Chen, Bo & Zhu, Menglong & Tang, Matthew & Howard, Andrew & Adam, Hartwig & Kalenichenko, Dmitry. (2017). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference.

[4] FPGA-Based Neural Network Acceleration for Handwritten Digit Recognition

Shen, Guobin (School of Electronics and Information Technology, Sun Yat-sen University, Guangzhou; 510006, China); Li, Jindong; Zhou, Zhi; Chen, Xiang Source: Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST, v 346, p 182-195, 2021, IoT as a Service - 6th EAI International Conference, IoTaaS 2020, Proceedings.

[5] J. Chang and S. Kang, "Optimizing FPGA-based convolutional neural networks accelerator for image super-resolution," 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), 2018, pp. 343-348, doi: 10.1109/ASPDAC.2018.8297347.