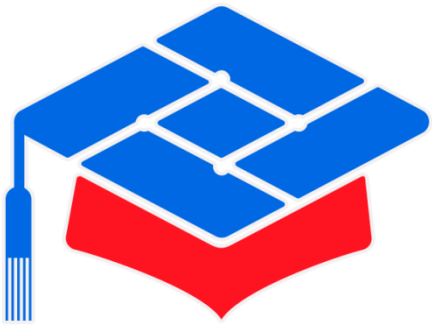


第七届

全国大学生集成电路创新创业大赛

报告类型*：方案描述文档

参赛杯赛*：海云捷讯杯

作品名称*：基于 FPGA 机器视觉缺陷检测的实现

队伍编号*：CICC1577

团队名称*：冰糖葫芦儿

目录

1 系统架构分析 1
1.1 总体系统架构 1
1.2 程序架构 3
1.2.1 总决赛方案 3
1.2.2 分区赛方案 4
2 关键技术分析 5
2.1 FPGA 端优化 5
2.1.1 重写 dvp_ddr3 与 ddr3_vga 5
2.1.2 双 Buffer 连续传输 11
2.1.3 使用单通道进行显示与推理 12
2.1.4 使用 FPGA 实现图像预处理中的 resize 13
2.1.5 PL 端绘制预测框 20
2.1.6 CNN 加速器优化 24
2.1.7 数据重排 28
2.2 模型优化 32
2.2.1 智能归因 32
2.2.2 数据增强 34
2.2.3 基于敏感度分析的剪枝量化 36
2.2.4 基于 L1 和 FPGM 法则的综合剪枝策略优化 37
2.2.5 输出层缩放 38
2.2.6 单通道模型 39
2.3 SDK 与 ssd_detction 优化 40
2.3.1 程序流程设计 40
2.3.2 数据重排 ARM 端优化 41
2.3.3 开启编译器 O3 优化 46
2.3.4 循环展开 47
2.3.5 PaddleLite 优化 48
2.3.6 修改 PaddleLite 为单通道 50

2.3.7NHWC->NCHW 通道转换的重写 51
2.3.8 图片预处理过程的重写 53
2.4 驱动优化 54
3 性能指标 55
3.1 推理速率指标（官方 Demo） 55
3.2 HDMI 显示刷新性能指标 55
3.3 实时推理刷新性能指标（决赛所要求系统） 55
3.4 数据重排优化指标 56
3.5 加速器优化指标 57
3.6 模型剪枝指标 57
3.7 预处理优化指标（改单通道） 58

1 系统架构分析

1.1 总体系统架构

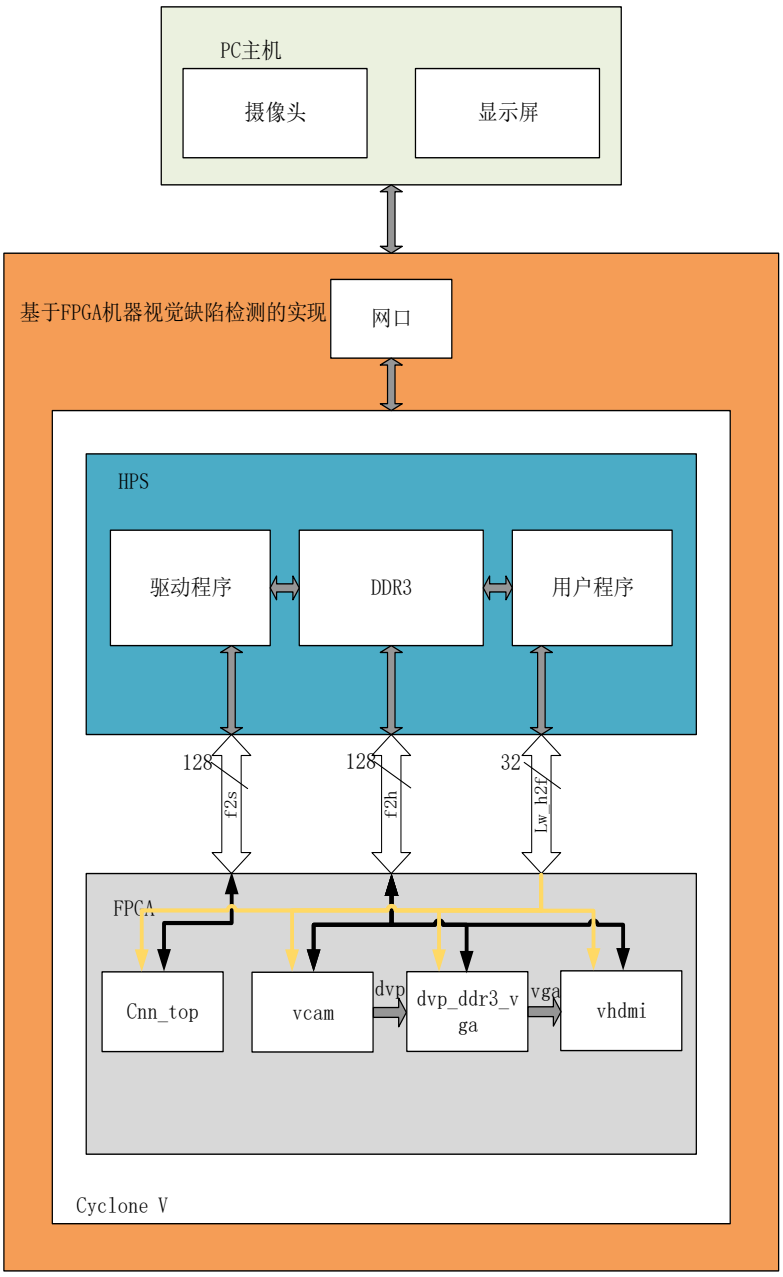


图 1.1　系统总体框图

系统架构如图 1.1，系统总体运行由 HPS 端调度，FPGA 端作为 HPS 端并行运行加速部分，主要作用是为 HPS 端执行卷积加速、HDMI 显示等功能，因此，我们可以通过将 HPS 端执行的数据重排、预处理、resize、plot 等操作移动到 FPGA 端进行并行运行，这样就可以加速整个系统的运行速度。

FPGA 端与 HPS 端沟通主要通过多个总线，有由 FPGA 端发起的数据传输 f2h_sdram 总线，f2h_axi 总线，也有由 HPS 下达指令的 lw_axi 总线。FPGA 数据通过总线传输到 PS 端 ddr3 中。在 Linux 系统中 FPGA 相当于其外设，因此使用的是 DMA 等驱动对数据进行搬运，传输的数据在 Linux 系统中处于内核空间，将其搬移到用户空间后我们就可以使用用户程序对数据进行操作。

表 1.1　FPGA 与 HPS 的桥接

组件名	桥接	Hps 连接端口
mm_bridge_sdram0	FPGA to SDRAM	f2h_sdram0_data
mm_bridge_axi	FPGA-to-HPS	f2h_axi_slave
mm_bridge_lw_axi	Lightweight HPS-to-FPGA	h2f_lw_axi_master

模型部分是整个系统进行推理的基础，主要分为模型训练以及模型部署，训练框架使用 PaddleDetection，推理框架使用 Paddle-Lite。针对训练的模型优化，主要集中于模型精度的提升以及模型大小的压缩，针对部署的模型优化，就是在 PS 端对 intel_SDK 以及推理框架 Paddle-Lite 的优化。

本系统通过**重写** dvp_ddr3、ddr3_vga 等官方 ip，利用 FPGA 并行计算的优势，将 HPS 部分推理流程中的多个任务**并行**到 FPGA 实现，例如对图像的 **resize** 以及推理结果的 **plot** 叠加。同时通过**时序约束**、**逻辑固化**、**去除无用组件**等方式提高 cnn 加速器的运行频率，加速了 HDMI 显示刷新速度以及推理速度。

在模型方面，我们通过使用**本地数据增强**、使用**剪枝策略**进行剪枝、优化**小目标检测精度**、进行**输出层缩放**、修改**单通道模型**等方式提升模型精度并压缩模型大小。

在 SDK 以及推理框架 Paddle-Lite 方面，我们使用 **Neon** 指令集、开启 **O3** 优化、使用 **OpenMP**、进行**循环展开**等方式优化了 PS 端的数据重排。在 Paddle-Lite 中**开启多核心推理**，推理过程使用**单通道模型与数据**，加速推理速度。

在构建实时推理系统时，我们在总决赛中，使用一个 SSD_Detection 进程，完成实时推理的刷新工作，每完成一帧的推理，将推理结果写入 PL_Plot 控制寄存器中，从而完成 HDMI 上的实时推理结果叠加显示。而推理工作，将完全由 FPGA 进行，当 DVP_DDR 模块将图像写入 BUF0 时，DDR_VGA 模块从 BUF1 中读取数据并使用 PL_Plot 模块叠加预测框，完成推理结果的显示以及视频流刷新工作。

1.2 程序架构

在总决赛中，我们抛弃了分区赛使用的双进程方案，引入 PL 端绘制预测框思想，从而可将双核 ARM-A9 全部用于推理，有效提升 CPU 的利用率，并且解决了双进程读取 DDR 冲突的问题，通过改方法，可以提升推理刷新时间稳定性，并可将推理时间提升 10ms。下面我们将两种方案都进行阐述，进行比较。

1. 2. 1 总决赛方案

我们设计了 PL 端绘制预测框的 IP，通过 h2f_lw_bridge 控制状态寄存器，从而控制 PL_Plot IP 的绘制，其状态寄存器说明如下图所示。

```
plot_virtual_base[0] = 0 ;      //总体传输寄存器
plot_virtual_base[1] = 0 ;      //单个框传输传输寄存器,可以使用该寄存器表示数据准备好一次
plot_virtual_base[2] = 0 ;      //传输个数寄存器
plot_virtual_base[3] = 0 ;      //框左上x坐标寄存器
plot_virtual_base[4] = 0 ;      //框左上y坐标寄存器
plot_virtual_base[5] = 0 ;      //框右下x坐标寄存器
plot_virtual_base[6] = 0 ;      //框右下y坐标寄存器
plot_virtual_base[7] = 0 ;      //框标签寄存器
plot_virtual_base[8] = 0 ;      //框置信度寄存器
```

图 1.2.1　PL_PLOT 控制寄存器

我们设计了一个 SSD_Detction 进程，其从 BUF2~BUF3 中读取经过 PL 侧 Resize 后的 300*300 单通道图像，转换为 Tensor 结构后进行使用 PaddleLite 推理，将推理结果写入 PL_Plot 寄存器，从而实现推理结果的叠加显示。

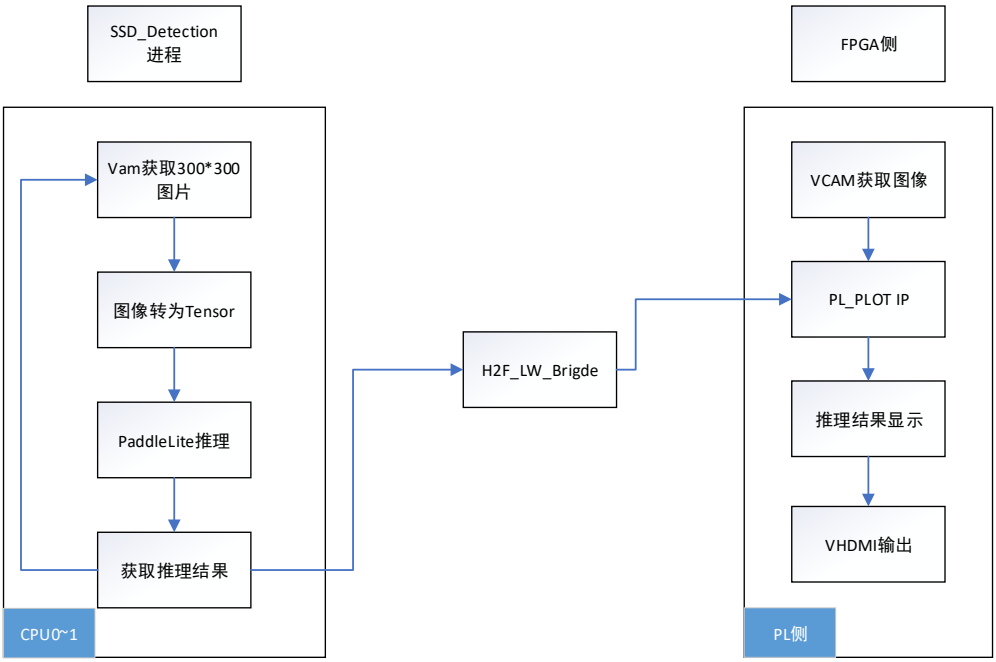


图 1.2.2　总决赛程序设计框图

1.2.2 分区赛方案

我们的系统将使用**两个进程**完成，他们中间通过共享内存的方式进行进程间通信。他们分别的工作是：

1. 进程 ssd_detection 用于从 ddr 中获取原始图像数据，进行模型的推理。并将 PaddleLite 推理结果拷贝入共享内存，给进程 2 使用。
2. 进程 ssd_hdmi 用于从 ddr 中获取图像原始数据，从共享内存中拷贝推理结果，并使用 visualize_result()函数将推理结果叠加到原图像，通过 ddr_vga IP 将叠加后的图像推流到 HDMI 显示。

由于使用两个进程，其推理与 HDMI 视频流推流过程是**独立**的，在推理结果并未刷新前，原图像叠加上一次的推理结果进行输出，其相互独立互不影响。

注：由于 ssd_hdmi 进程运行较快，为了降低 CPU 资源占用，我们使用 usleep() 函数将进程休眠，并且由于系统中大部分进程均运行于 cpu0 上，为了保证推理速率，我们将 ssd_hdmi 视频推流进程绑定于 cpu0 运行，推理进程 ssd_detction 独占 cpu1。并提高推理进程 ssd_detction 的任务优先级为 0，降低视频推流进程 ssd_hdmi 的任务优先级为 **39**。如此可减少两个进程对于资源的抢占，在不降低视频流帧率的情况下，加快推理速度。

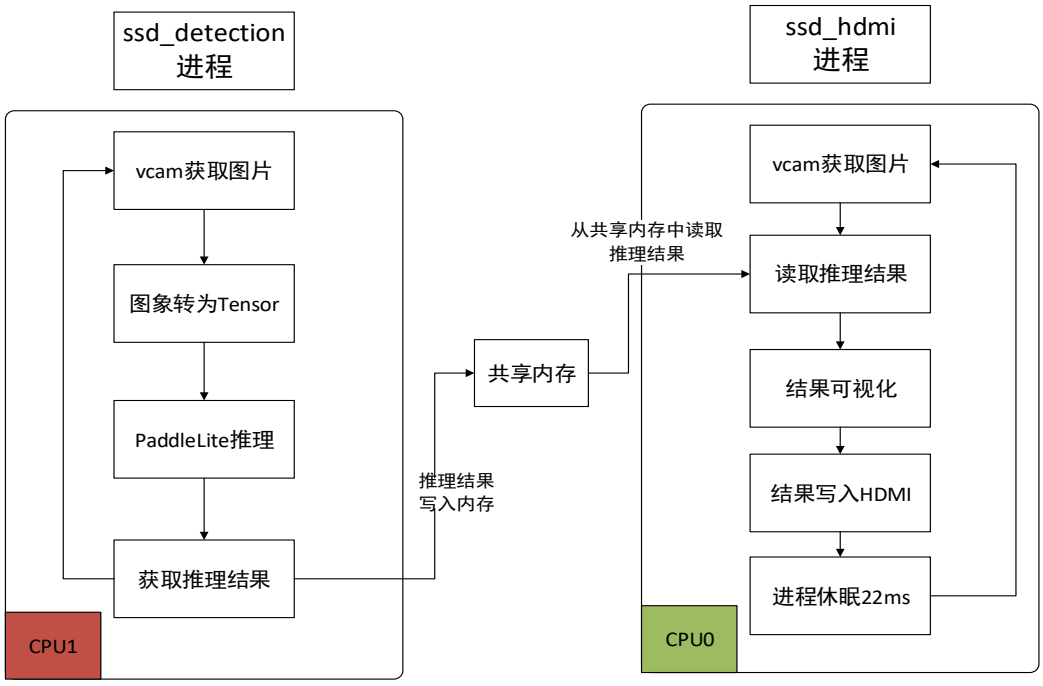


图 1.2.3　分区赛程序设计框图

2 关键技术分析

2.1 FPGA 端优化

2. 1. 1 重写 dvp_ddr3 与 ddr3_vga

由于官方提供的 dvp_ddr3 以及 ddr3_vga 的两个 ip，其需要使用 HPS 端进行调度，严重影响推理程序的速度，同时 HDMI 速度也受到损失，因此最好能够将 HDMI 显示完全与 HPS 端分离，得到的帧数据直接传输到 HPS 端 DDR3 的指定双 buffer 中，从 buffer 直接读取数据然后推理，这样无需 PS 调度，加快了推理速度同时加快了 HDMI 显示速度。

同时，官方提供的 ip 我们无法进行自定义的修改，如修改为单通道、将 resize、图像预处理模块添加进入 ip 中。通过自定义 ip，我们可以将想要实现的操作加入模块 ip 中，以提升推理准确度、推理速度等指标。

为了便于移植，我们将 dvp_ddr3 与 ddr3_vga 整合为一个 ip 即 dvp_ddr3_vga。

•绘制 dvp_ddr3_vga 总体框图如下：

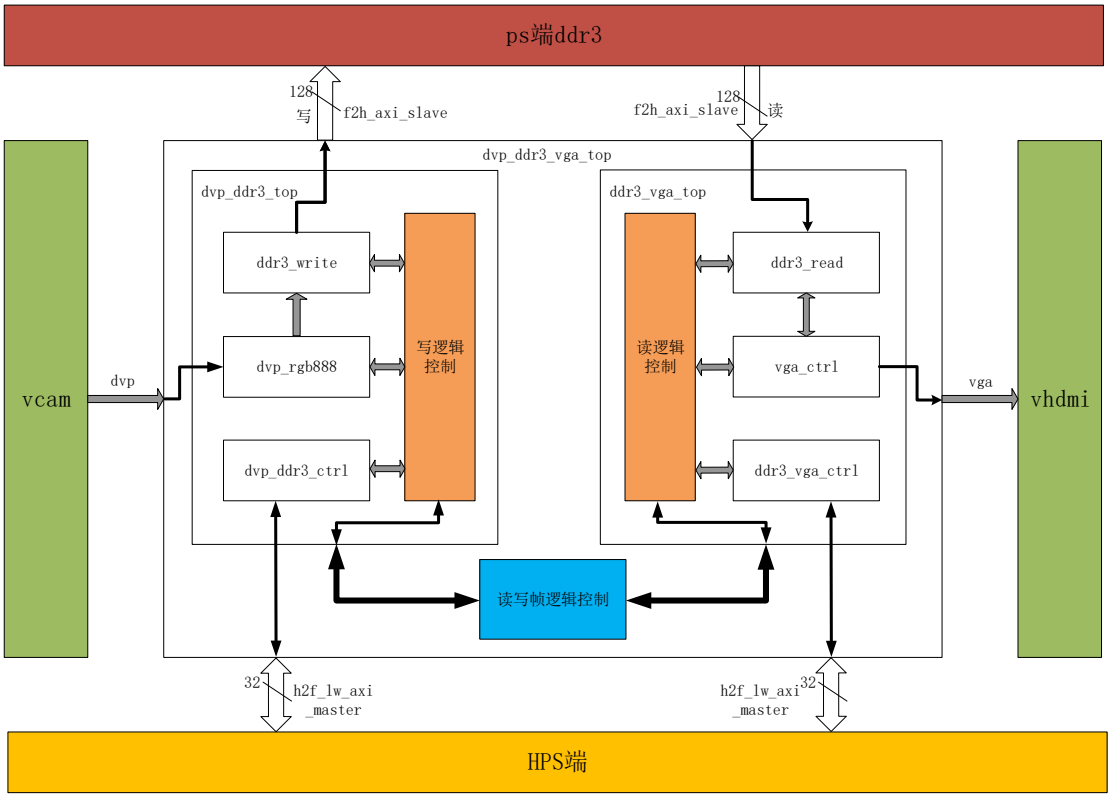


图 2.1.1　dvp_ddr3_vga 模块框图

•dvp_ddr3_vga 模块组成如下：

表 2.1.1　dvp_ddr3_vga 模块组成

模块名称	功能描述
Top 模块	
dvp_ddr3_vga_top	例化 dvp_ddr3_top、ddr3_vga_top 模块，并完成相应读写逻辑的控制。
dvp_ddr3 模块	
dvp_ddr3_top	例化 dvp_rgb888、ddr3_write、dvp_ddr3_ctrl，并产生相应帧写逻辑。
dvp_rgb888	DVP 数据采集模块，将 8 位的 DVP 数据转换为 128 位并生成相应的写请求送给 ddr3_write 模块，此外还需要产生场同步开始与结束信号交给 dvp_ddr3_top 模块产生相应帧写逻辑。
ddr3_write	接收 dvp_rgb888 传来的 128 位数据，在 dvp_ddr3_top 控制逻辑下通过 f2h_axi_slave 完成对 ps 端 ddr3 的突发写操作。
dvp_ddr3_ctrl	dvp_ddr3 控制 IP，包含了状态控制寄存器，可启动 IP 运行并读取写状态。
ddr3_vga 模块	
ddr3_vga_top	例化 vga_ctrl、ddr3_read、ddr3_vga_ctrl，并产生相应帧读逻辑。
vga_ctrl	从 ddr3_read 中获取 128 位数据将其转换位 24 位 rgb 数据并以 vga 时序输出，vga 驱动时钟为 5M 时钟。此外还需要产生场同步开始与结束信号交给 ddr3_vga_top 模块产生相应帧读逻辑。
ddr3_read	在 vga_ctrl 的读请求下提供相应的 128 位数据。在 ddr3_vga_top 控制逻辑下通过 f2h_axi_slave 完成对 ps 端 ddr3 的突发读操作。
ddr3_vga_ctrl	ddr3_vga 控制 IP，包含了状态控制寄存器，可启动 IP 运行并读取写状态。

图 2.1.1 以及表 2.1.1，展示了 dvp_ddr3_vga 模块其组成以及各模块作用，其中，最重要的模块就是接收 dvp 时序的 dvp_rgb888 以及转换 vga 时序的 vga_ctrl 模块，以及负责使用 avalon 总线写入读出 ddr3 的 ddr3_read、ddr3_write 模块。

dvp_rgb888 模块接收 vcam 传来的 dvp 数据，将其寄存为 ddr3_write 的 128 位数据写入 ddr3_write 的 FIFO 中，当 ddr3_write 的 FIFO 中数据深度达到突发长度 16 时，ddr3_write 进行一次突发写，将连续 16 个 128 位数据通过 f2h_axi_slave 总线写入 PS 端 ddr3 的 buffer 中，其中 ddr3_write 写入是以帧为开始和结束。

ddr3_read 模块通过 f2h_axi_slave 总线从 ps 端 ddr3 中的 buffer 读取 dvp_ddr3 模块写入的帧，一次突发读取 16 个 128 位数据，将其存储在模块内部的 FIFO 中，当 vga_ctrl 发起一次读请求时，在读时钟下，每一个时钟周期向外输出 128 位数据。

由于 vga_ctrl 在 vga 时钟下需要输出的是 24 位 rgb_888 像素，因此使用移位寄存器向外输出 24 位数据，需要满足位宽可以被 24 整除。读请求有效情况下，ddr3_read 一个时钟输出 128 位数据，128 无法被 24 整除，而 384 可以被 24 整除。因此 vga_ctrl 一个读请求我们需要持续三个时钟周期，也就是连续输出 3 个 128 位数据，即 384 位数据。

在 vga 时钟驱动下，每个时钟周期 vga_ctrl 模块向外输出 24 位数据，同时产生同步的 vga_vsync、vga_de 等同步信号。

vga 时钟我们按满帧计算，一个时钟输出一个 24 位数据：

$$400*320*30=3840000=3.84MHz$$

上述时钟不满足 5M 时钟，我们定义 vga 时序如下：

表 2.1.2　400*320 分辨率 30 帧 vga 时序

400*320@30	行同步	场同步
同步 SYNC	40	40
后沿 BACK	0	20
有效 VALID	400	320
前沿 FRONT	0	0
周期 TOTAL	440	380

计算实际的 vga 时钟如下：

$$440*380*30=5016000\approx 5MHz$$

该时钟可以保证显示帧率为满 30 帧。

•**dvp_ddr3_vga 端口如下：**

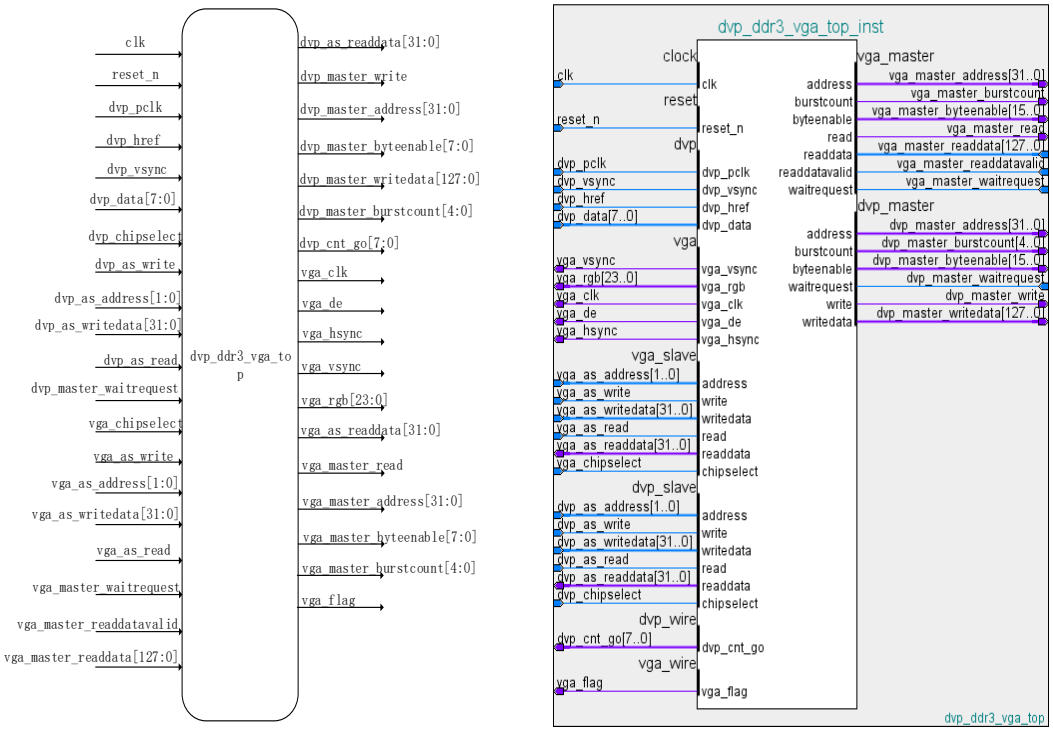


图 2.1.2　dvp_ddr3_vga 模块端口

•**dvp_ddr3_vga 模块信号功能说明如下：**

表 2.1.3　dvp_ddr3_vga 模块信号功能说明

信号	位宽	I/O	功能
clk	1	I	系统时钟
reset_n	1	I	低电平复位
dvp 时序输入接口			
dvp_pclk	1	I	摄像头像素时钟
dvp_href	1	I	摄像头行同步信号
dvp_vsync	1	I	摄像头场同步信号
dvp_data	8	I	摄像头图像数据
dvp_slave 控制接口(avalon_mm_slave)			
dvp_chipselect	1	I	片选信号，当该信号有效时，IP 核的所有寄存器才能被访问。
dvp_as_address	2	I	地址，该地址信号指定了 IP 核被访问的寄存器编号，通过给该信号赋予不同的值，就能选择访问不同的寄存器。
dvp_as_write	1	I	写请求信号，当该信号有效时，writedata 端口上的数据会被写入 address 指定的寄存器中。

dvp_as_writedata	32	I	写数据信号，当 write 信号有效时，该端口上的数据会被写入 address 指定的寄存器中。
dvp_as_read	1	I	读请求信号，当该信号有效时，readdata 端口上的数据会被写入 address 指定的寄存器中。
dvp_as_readdata	32	O	读数据端口，当 chipselect 有效时，该端口上的值为 address 指定的寄存器中的值。
dvp_master 写接口(avalon_mm_master)			
dvp_master_address	32	O	address 信号代表一个字节地址
dvp_master_write	1	O	置位表示一个 write 传输，如果存在，则需要 writedata
dvp_master_byteenable	16	O	字节使能。byteenable 中的每个比特对应于 writedata 和 readdata 中一个字节。
dvp_master_writedata	128	O	写传输的数据
dvp_master_burstcount	5	O	突发的传输数量，dvp 写突发长度为 16
dvp_master_waitrequest	1	I	当 slave 无法响应读写请求时，置位 waitrequest，强制 master 等待；
dvp_wire 测试接口			
dvp_cnt_go	8	O	记录 hps 发出的请求次数
vga 时序输入接口			
vga_clk	1	O	输出 vga 时钟,频率 5MHz
vga_de	1	O	有效数据选通信号 DE
vga_vsync	1	O	输出场同步信号
vga_hsync	1	O	输出行同步信号
vga_rgb	24	O	输出 24bit 像素信息
vga_slave 控制接口(avalon_mm_slave)			
vga_chipselect	1	I	片选信号
vga_as_address	2	I	地址
vga_as_write	1	I	写请求信号
vga_as_writedata	1	I	写数据端口
vga_as_read	1	I	读请求信号
vga_as_readdata	32	O	读数据端口
vga_master 读接口(avalon_mm_master)			
vga_master_address	32	O	address 信号代表一个字节地址
vga_master_read	1	O	置位表示一个 read 传输，如果存在，则需要 readdata

vga_master_byteenable	16	O	字节使能
vga_master_readdata	128	I	写传输的数据
vga_master_burstcount	1	O	突发的传输数量，vga 读突发长度为 1
dvp_master_waitrequest	1	O	当 slave 无法响应读写请求时，置位 waitrequest，强制 master 等待；
vga_master_readdatavalid	1	I	读数据有效信号

图 2.1.2 以及表 2.1.3，展示了 dvp_ddr3_vga 模块的端口信号，并详细描述了端口信号的功能，对于细节的 dvp_rgb888、vga_ctrl 的时序等问题，我们将其放在 2.1.2 节中进行详细叙述。

•dvp_ddr3_vga 状态控制寄存器说明如下：

表 2.1.4　dvp_ddr3 状态控制寄存器说明

寄存器名称	地址	位宽	R/W	功能说明
control	0	32	W	bit0:单次传输触发位，该位仅在主机写该寄存器且写入数据的 bit0 位为 1 时为高电平，否则保持低电平，对应于 control_go 信号。对于双 buffer 连续读写情况，无需 ps 控制时，该位无效。 bit[2:1]:预留的使能控制信号，对应于 control_en。
control_user_base	1	32	W	PL 从 PS 侧 DDR3 突发读数据的物理首地址。
control_user_length	2	32	W	一帧图像数据量，以 Byte 为单位。
control_state	3	32	W/R	bit0:为方便 PS 端搬运图像，dvp 连续写双 buffer 的状态位，利用该位 PS 端可读取空闲 buffer; dvp 向 buffer0 写完一帧图像，则向 control_state 写 1; dvp 向 buffer1 写完一帧图像，则向 control_state 写 0;

表 2.1.4 是针对 dvp_ddr3 这个模块的状态控制寄存器的说明，ddr3_vga 模块和该模块类似，此处不再赘述。

上述寄存器（除 control_state）只有在不使用双 buffer 连续传输时起用，使用双 buffer 连续传输时，由于需要写的两个 buffer 地址以及写长度在写 ip 时已经固定了，无需通过 PS 进行启动与赋地址。

2.1.2 双 Buffer 连续传输

如果在 PS 端发起传输一帧的命令给 dvp_ddr3 模块以获取一帧图像，需要等待的时间是：

dvp_ddr3 模块一帧传输完成时间+memcpy 的时间

为了 ps 端获取帧图像的即时性以及减少 PS 端复制内存的时间开销，我们使得 FPGA 端不需要通过 ps 端进行写入。因为 dvp 时钟为 11.5MHz，vga 时钟为 5MHz，而 dvp 一个时钟周期传输 8 位，vga 一个时钟周期传输 24 位，因此 vga 的速度快于 dvp 速度，也就是 dvp 写完一帧的速度更慢，因此我们可以以 dvp 写入一帧是否完成为标志位，使用乒乓操作的原理进行双 buffer 连续写操作。

双 buffer 写状态机如下：

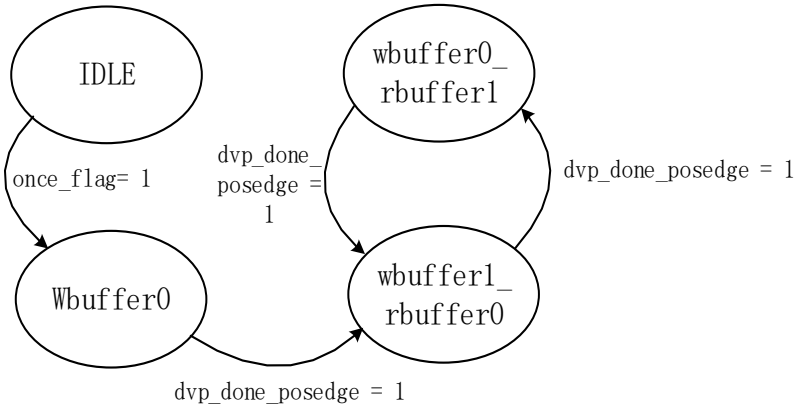


图 2.1.3　dvp_ddr3_vga_top 的读写状态机

该状态机初始状态为 IDLE，启动后自动进入 Wbuffer0 状态，dvp_ddr3 模块写入一帧给 buffer0，当写完 buffer0 时，ddr3_write 置位 dvp_done 表示一帧写完，使用边沿检测 dvp_done 上升沿得到 dvp_done_posedge 信号，该信号只持续一个时钟周期，检测到该信号代表一帧写完。因此写完 buffer0 时，转换状态为 wbuffer1_rbuffer0，即 dvp 写 buffer1，vga 读 buffer0。再次检测到 dvp_done_posedge 时，转换状态为 Wbuffer0_rbuffer1，即 dvp 写 buffer0，vga 读 buffer1，如此一来就可以使得双 Buffer 连续传输得以实现。

使用双 buffer 连续传输意味着 HDMI 显示完全由 PL 端控制，无需 PS 端进行控制，但是由于我们需要在显示上叠加推理结果框等，想要完全实现双 buffer 连续传输，我们需要实现 PL 端绘制推理结果框。在没有实现上述模块时，我们只让 dvp_ddr3 由 PL 端控制，ddr3_vga 的地址切换由 PS 端控制，因此 buffer 结构如下：

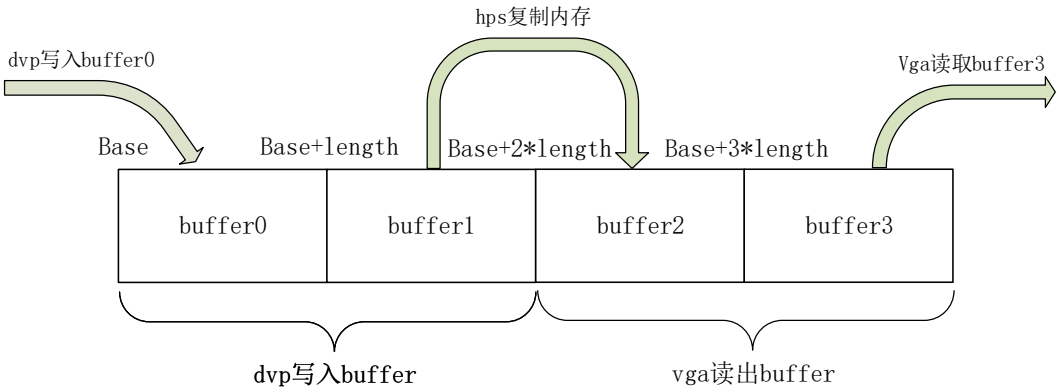


图 2.1.4　普通的四 buffer

使用上面结构主要是为了防止 hps 读取与 dvp 写入到同一片空间产生冲突，同时可以使得 dvp 写入满帧率，vga 读出满帧率(hps 的 memcpy 速度远大于显示速率)。

2. 1. 3 使用单通道进行显示与推理

•dvp_rgb888 时序

对于三通道：

三通道需要传输 400*320*3 个字节，读取一个 1 个 128 位数据可以传 16 个字节，一行 400 个像素有 1200 个字节，一共需要写入 75 个 128 位数据才能传输完成 400 个像素。

三通道信号时序如下图：

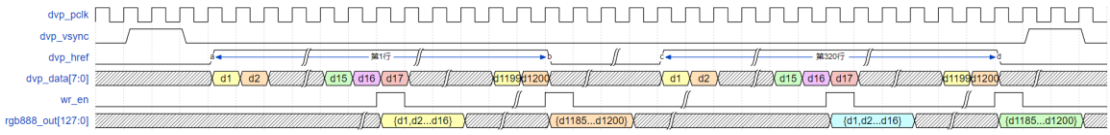


图 2.1.5　三通道 dvp_rgb888 的时序

对于单通道：

单通道只需要传输 400 个字节，一共需要写入 25 个 128 位数据才能传输完成 400 个字节。与三通道时序不同的是，隔一个通道取一次，因此我们将原 dvp_pclk 进行三分频再使用该三分频时钟进行采样，在 fpga 工程中将该时钟约束为全局时钟，即可使得该时钟与 dvp_pclk 时钟同步。我们也可以使用 pll 产生 dvp_pclk 的三分频时钟，这样可以保证该时钟与 dvp_pclk 时钟是同步的。

单通道采样信号时序如下图：

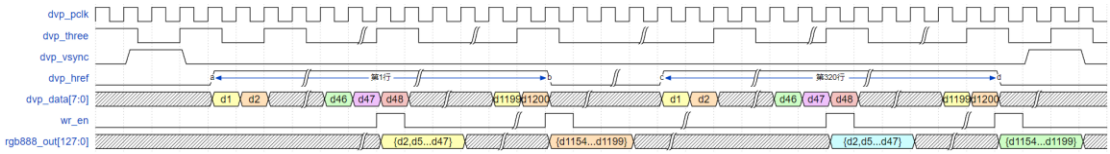


图 2.1.6 单通道 dvp_rgb888 的时序

•vga_ctrl 时序图

对于三通道：

由于是三通道，一个像素 24 位，而我们一次读取的数据位宽是 128 位，无法形成整除关系，因此需要将读取的 128 位数据凑成 384 位，这样 384/24=16 就可以被整除传输 16 个像素。对于一行 400 个像素，我们一共需要读 25 次 384 位数据，一次读请求为三个时钟周期宽。

需要注意，此处 vsynv 场同步信号高电平为有效，低电平为同步。

以下是第一行对应的时序图：

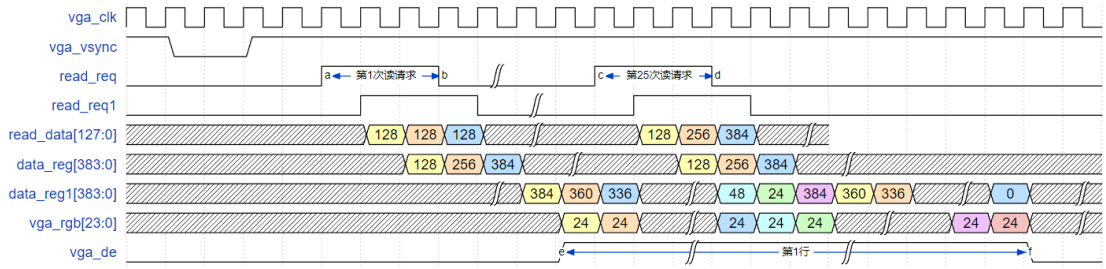


图 2.1.7　三通道 vga_ctrl 的时序

对于单通道：

与三通道不同的是，读请求宽度为一个时钟周期，读请求只超前 vga_de 四个时钟周期，即请求一次读 128 位，不是三通道的 384 位，这是因为单通道一个时钟周期移位寄存器只输出 8 位，输出的 24 位像素的三个通道是一样的，因此，128 位可以输出 16 个 24 位像素。其时序与三通道相似，因此不再放出时序图。

使用单通道后，每帧图像写入减少了 2/3 的大小，原来需要写入 400*320*3，现在只需要写入 400*320 大小的图像。这样在 DDR3 中占用的空间直接减少了 2/3，同时大大减轻了 F2H 总线的负担。

2.1.4 使用 FPGA 实现图像预处理中的 resize

在推理时，SSD_Monlienetv1 模型要求所有输入图像都必须为同样大小，这是由于其特侦图大小以及先验框数目都必须根据输入图像大小确定，因此我们在进行推理时所有输入图像在输入网络前都必须经过 resize 操作将其变为统一大小 300*300。根据我们的测量，400*320*3 图像 resize 为 300*300*3 大小所花费的

时间为 7ms。在踏入了 100ms 的推理时间后，每一点提升都弥足珍贵，更不必说 7ms 的提升了，因此我们完成了将 resize 操作转移到 PL 端实现的方式完成图像预处理的并行化。

在原推理程序中，我们使用 CV 库中的双线性插值法对图像进行 resize，因此我们为了追求与原程序效果一致，我们在 PL 端实现的 resize 操作原理也是双线性插值。

- **具体原理如下：**

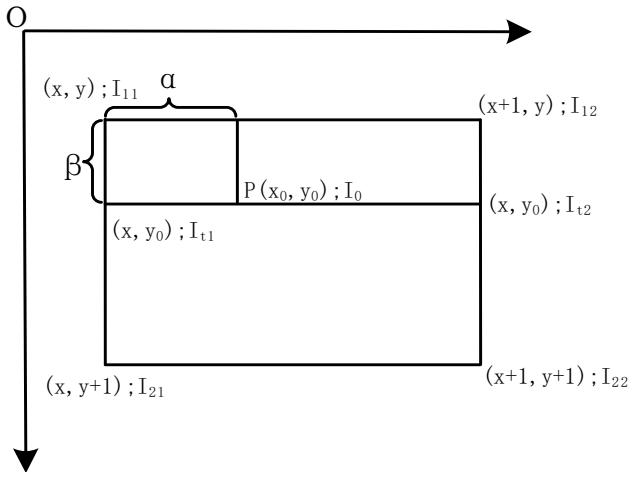


图 2.1.8 双线性插值原理

双线性插值算法经常用来对图像缩放，其核心思想是在水平和垂直两个方向上分别进行一次线性插值。

双线性插值算法根据点 P(x0,y0)的四个相邻点(x,y)、(x+1,y)、(x,y+1)、(x+1,y+1)的灰度值 I_{11}、I_{12}、I_{21}、I_{22}，通过两次插值计算得出点 P(x0,y0)处的灰度值 I_0，双线性插值的示意图如上。

而具体的计算过程如下：

(1)计算 α 和 β，分别为 x_0 和 y_0 的小数部分

$$\alpha = x_0 - x$$
$$\beta = y_0 - y$$

(2)根据 I_{11} 和 I_{21} 的插值求点(x,y0)的灰度值 I_{t1}

$$I_{t1} = I_{11} + \beta(I_{21} - I_{11})$$

(3)根据 I_{12} 和 I_{22} 的插值求点(x+1,y0)的灰度值 I_{t2}

$$I_{t2} = I_{12} + \beta(I_{22} - I_{12})$$

(4)根据 I_{t1} 和 I_{t2} 的插值求点(x0,y0)的灰度值 I_0

$$I_0 = I_{t1} + \alpha\left(I_{t2} - I_{t1}\right) = (1-\alpha)(1-\beta)I_{11} + (1-\alpha)\beta I_{21} + \alpha(1-\beta)I_{12} + \alpha\beta I_{22}$$

上述计算过程用矩阵表示为：

$$I_0 = ABC$$

其中有：

$$A = \left[\left(1-\beta\right)\beta\right]$$

$$B = \begin{bmatrix} I_{11} & I_{12} \\ I_{21} & I_{22} \end{bmatrix}$$

$$C = \left[\left(1-\alpha\right)\alpha\right]^T$$

实际上的 α 和 β 是计算当前缩放图像和原始图像映射的小数部分，因为我们需要根据比例计算缩放图像像素应该使用原始图像的哪四个像素求灰度，但是比例乘以行数列数未必能得到一个整数，因此整数部分我们作为映射的坐标，小数部分作为在进行插值时的加权也就是 α 和 β。

在实际的使用中：

(1) 我们令 x_ratio 和 y_ratio 代表原始图像与缩放后图像在水平和垂直方向上的比率，即步进。

(2) 根据 x_ratio 和 y_ratio，计算缩放后图像坐标(i,j)到到原始图像坐标的映射，其中整数部分作为原始图像的像素坐标(x,y)，小数部分作为像素灰度值的权重也就是 α 和 β。实际上以行为例，本步的操作就是利用缩放后图像前 i-1 行乘上竖直的缩放比例 y_ratio，计算缩放后图像的第 i 行在原始图像中的行位置，然后取其整数为原始图像的起始行，小数部分作为加权值 α 和 β。

(3) 根据像素坐标(x,y)我们可以得到原始图像四邻域的像素灰度值 img1(y:y+1,x:x+1)，根据 α 和 β 可以得到四邻域水平方向上的像素权重1-α 和 α 以及四邻域竖直方向上的像素权重1-β 和 β，四邻域像素与权重的乘累加和，作为放大后的图像的近似灰度值 img2(i,j)。

- **根据以上原理，我们可以实现其 FPGA 版本操作:**

(1)对原始图像进行行缓存

由于双线性插值需要利用近邻 4 个像素进行计算并获得目标像素值，所以至少需要缓存两行像素。我们利用双端口 BRAM 对原始图像进行行缓存。我们需要将 400*320 的像素缩小为 300*300，因此缓存一行原始图像像素至少需要 400 个 BRAM 地址空间，为了使得我们的 IP 可以在一定范围内适应输入图像大小变化，此处将 BRAM 一行像素空间定义为 1024，同时定义 BRAM 最大可以存储 4 行像素，因此 BRAM 的深度为 4096，而 BRAM 的宽度等于像素位宽也就是 8。

同时我们进行双线性插值时，需要同时从 BRAM 中读出近邻 4 个像素,我们为了降低设计难度，采用 4 个 BRAM 对图像进行行缓存，每个 BRAM 分别输出近邻 4 个像素中的其中 1 个。如下图所示：

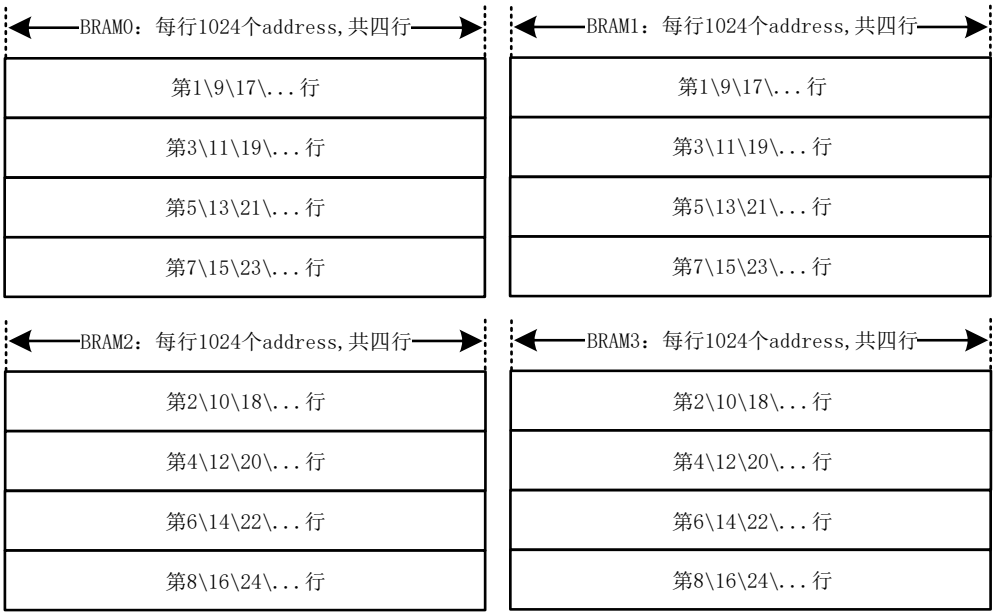


图 2.1.9 BRAM 的存储结果

可以看到其中图像奇数行像素同时存于 BRAM0 和 BRAM1 中，偶数行像素同时存于 BRAM2 和 BRAM3 中。因此 BRAM0、BRAM1、BRAM2 和 BRAM3 分别输出左上角、右上角、左下角、右下角的像素。

将原始图像存入 BRAM 之前，需要产生相应的 BRAM 地址。根据原始图像的场信号 per_img_vsync 和行信号 per_img_href，对原始图像进行列统计 img_hs_cnt ∈ [0,C_SRC_IMG_WIDTH-1]和行统计 img_vs_cnt ∈ [0,C_SRC_IMG_HEIGHT-1]，其中 C_SRC_IMG_WIDTH 和 C_SRC_IMG_HEIGHT 分别表示原始图像的宽度和高度。BRAM 地址的计算公式如下所示，其中 img_hs_cnt 用于产生行内像素的地址、img_vs_cnt[2:1]用于产生不同行的基地址。

$$\text{bram\_}a\text{\_waddr} = \{\text{img\_vs\_cnt}[2:1], 10'\text{b0}\} + \text{img\_hs\_cnt}$$

一个 BRAM 大小为 4*1024 代表 4 行。因此使用 img_vs_cnt[2:1]产生这四行的地址，然后再使用 img_hs_cnt 产生其行内像素的地址。img_hs_cnt 计数的行上限受输入图像 dvp 行有效信号限制，其上限为输入原始图像行长度。为什么使用 img_vs_cnt[2:1]产生这四行的地址我们讲完写使能就知道了。

写地址解决了，接下来是写使能问题，由于 BRAM0 和 BRAM1 存储的行是奇数行，BRAM2 和 BRAM3 存储的行是偶数行，因此我们只需要生成两个写使能信号可以对当前有效行数是奇数还是偶数进行判断即可，这很简单。因此我们的写使能信号在场同步有效以及行同步有效时，使用 img_vs_cnt[0]判断当前行是奇偶，这是因为 img_vs_cnt[0]在最开始第 1 行时为 0，代表奇数行，第二行时为 1 代表偶数行，如此往复我们就可以通过 img_vs_cnt[0]判断奇偶行。

因此我们使用 img_vs_cnt[0]以及其取反信号分别可以生成奇数行和偶数行的写使能信号，在我们的代码里 bram1_a_wenb 为 BRAM0 和 BRAM1 的写使能，bram2_a_wenb 为 BRAM2 和 BRAM3 的写使能。以上两个写使能信号分别控制两个 BRAM。

而写地址信号则同时控制 4 个 BRAM，因此如果要使用写地址信号对 BRAM0 和 BRAM1 以及 BRAM2 和 BRAM3 分别进行写地址的变化重点就在于使用 img_vs_cnt[2:1]。因为两组 BRAM 的写使能信号是轮流有效的，因此写地址信号在前一行属于第一组 BRAM，后一行属于第二组 BRAM，而 img_vs_cnt[2:1]每经过两行增加 1，刚好与写使能信号匹配，这样两组 BRAM 在写入行地址时是同步变化的，这样经过 8 个行周期，一共有四对奇偶行就分别写入了两组 BRAM。

(2)当每行像素缓存到 BRAM 后，将行统计 img_vs_cnt 作为标签存入异步 FIFO 中后续进行双线性插值放大算法时，会根据该标签判断 BRAM 中是否已经缓存了插值所需要的两行像素。

(3)在进行双线性插值算法之前，需要计算原始图像与目标图像在水平和垂直方向上的比率(即目标图像映射到原始图像的坐标步进)C_X_RATIO 和 C_Y_RATI O。已知原始图像的分辨率为 400x320、目标图像的分辨率为 300x300，且要求将比率定标为 16 位小数，故 C_X_RATIO 和 C_Y_RATIO 的计算结果如下:

$$C\_X\_\mathrm{RATIO} = \mathrm{floor}\left(\frac{C\_SRC\_IMG\_\mathrm{WIDTH}}{\mathrm{C}\_DST\_IMG\_\mathrm{WIDTH}} \times 2^{16}\right) = \mathrm{floor}\left(\frac{400}{300} \times 2^{16}\right) = 87381$$

$$C\_Y\_\mathrm{RATIO} = \mathrm{floor}\left(\frac{C\_SRC\_IMG\_\mathrm{HEIGHT}}{\mathrm{C}\_DST\_IMG\_\mathrm{HEIGHT}} \times 2^{16}\right) = \mathrm{floor}\left(\frac{320}{300} \times 2^{16}\right) = 69905$$

注意 C_X_RATIO 和 C_Y_RATIO 需要使用 17’进行存储

(4)目标图像的坐标(y_cnt,x_cnt)及目标图像映射到原始图像的坐标(y_dec,x_dec)均由控制器负责完成。下图所示为控制器状态机，控制器状态跳转的说明，如表所示。

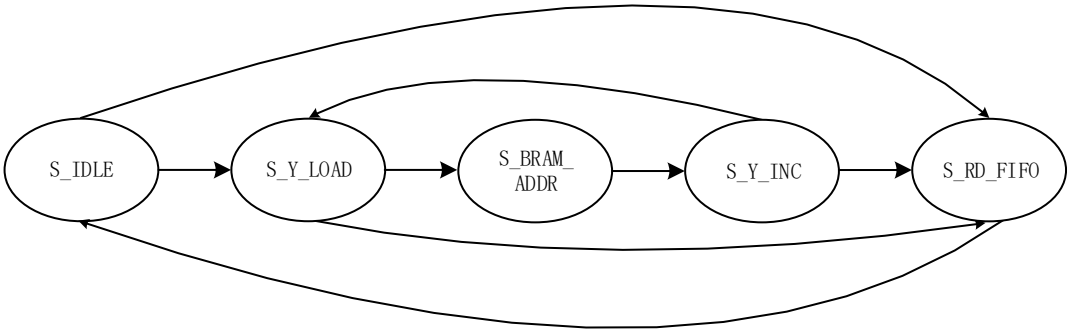


图 2.1.10 坐标映射的状态机控制

表 2.1.5 状态跳转说明

状态名	功能描述
S_IDLE	S_IDLE 状态中，当 FIFO 非空时，若 FIFO 中的标签 img_vs_cnt 不为 0，且目标图像最后一行像素的最近邻插值已经完成(即 y_cnt==C_DST_IMG_HEIGHT)，则进入 S_RD_FIFO 状态;否则进入 S_Y_LOAD 状态
S_Y_LOAD	S_Y_LOAD 状态中，对目标图像映射到原始图像的 Y 标 y_dec 进行四舍五入计算。由于 y_dec[26:16]是整数部分、y_dec[15:0]是小数部分，故四舍五入结果为 y_dec[26:16]+y_dec[15:0]。若结果小于等于 img_vs_cnt，则说明 BRAM 已经缓存了插值所需要的两行像素，进入 S_BRAM_ADDR 状态;否则进入 S_RD_FIFO 状态
S_BRAM_ ADDR	S_BRAM_ ADDR 状态中，生成目标图像的 X 坐标 x_cnt、目标图像映射到原始图像的 X 坐标 x_dec，完成后进入 S_Y_INC 状态
S_Y_INC	S_Y_INC 状态中，生成目标图像的 Y 坐标 y_cnt、目标图像映射到原始图像的 Y 坐标 y_dec，若 y_cnt 等于目标图像最后一行 C_DST_IMG_HEIGHT-1 时，则进入 S_RD_FIFO 状态;否则，进入 S_Y_LOAD 状态
S_RD_FIFO	S_RD_FIFO 状态中，将 FIFO 中的标签读出，进入 S_IDLE 状态

(5)根据(y_dec,x_dec)可以得到近邻 4 个像素中左上角像素的像素级坐标整数部分(y_int_c1,x_int_c1，以及(y_dec,x_dec)与近邻 4 个像素的水平和垂直距离 x_fra_c1、inv_x_fra_c1、y_fra_c1、inv_y_fra_c1，也就是小数部分，相当于我们

上面的1-α和α， 1-β和β。

(6)将(y_int_c1,x_int_c1)转为BRAM的读地址bram_addr_c2，以及根据x_fra_c1、inv_x_fra_c1、y_fra_c1、inv_y_fra_c1 计算近邻 4 个像素的权重 frac_00_c2、frac_01_c2、frac_10_c2、frac_11_c2，分别代表左上右上左下右下角的像素。其中 bram_mode_c2 用于指示左上角像素位于图像奇数行或偶数行，其值为 0 时表示奇数行，为 1 时表示偶数行。

(7)根据 bram_mode_c2 产生近邻 4 个像素在 4 个 BRAM 中的读地址，注意，如果左上角像素位于奇数行，则可以将下一行取为偶数行，地址是顺着取的。如果左上角像素位于偶数行，则下一行应该为奇数行，地址需要跳一行取才行。

(8) 根据 4 个 BRAM 的读地址从 BRAM 中读出 4 个像素并映射到近邻 4 个像素的位置上。

(9) 得到的近邻 4 个像素中可能存在像素超出图像边界的情况，需要进行像素的边界复制，像素越界处理。如果边界超出图像边界，则对像素进行复制处理。

(10)将近邻 4 个像素与(6)中计算的各自的权重相乘后累加，得到目标像素的灰度值

(11)由于计算得到的目标像素灰度值值可能大于 255 为了避免像素值越界，将灰度值大于 255 的像素，直接输出 255，灰度值越界处理。

注意，以上所有操作，只有写入使用原始图像时钟，其余所有操作使用缩放图像时钟。

- **resize 模块的组成：**

表 2.1.6 resize 模块组成

模块名称	功能描述
resize_top	例化 rgb2gray、bilinear_interpolation、dvp_gray
rgb2gray	将输入三通道的 dvp 时序转为单通道 dvp 时序输出，也就是将 rgb 转为灰度图的 dvp 时序输出
bilinear_interpolation	实现双线性插值，输入为灰度的 400*320 原图像，输出为经过 resize 的 300*300 的单通道灰度图像，其帧时序与原
dvp_gray	就是原来的 dvp_rgb888，为了防止模块冲突进行了重命名，作用是将 bilinear_interpolation 模块输出的 dvp 时序

	转换为 ddr3_write 可以接收的输入

- **使用 resize 后的 buffer 结构：**

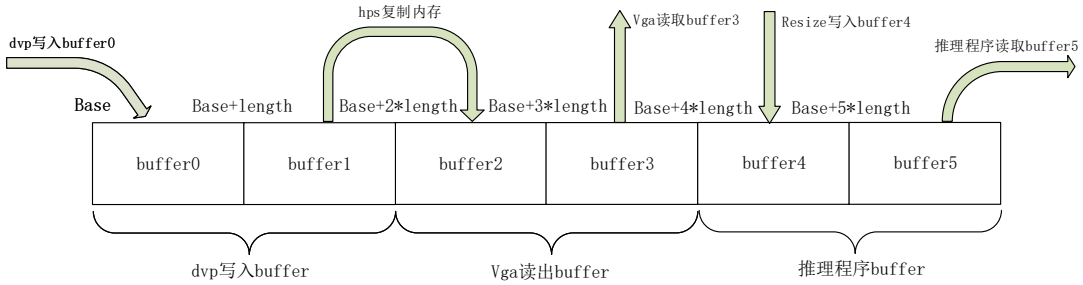


图 2.1.11　使用 resize 后的 buffer 结构

使用 resize 的 buffer 结构如上，使用单通道的 resize 图像大小为 300*300，因此只占用 buffer4 和 buffer5 的前 300*300 的空间。resize 与 dvp 是帧同步的。

2.1.5 PL 端绘制预测框

在上文中，我们为了保证推理刷新以及 HDMI 刷新互不影响，我们使用了两个核心分别运行两个程序并行执行达到同时完成两个任务的功能。其根本就是为了达到并行工作，既然能在 PS 端并行，为何我们不能在 FPGA 端并行呢？这与我们使用 FPGA 端进行 resize 是一样的思路。如果我们需要 HDMI 刷新与推理刷新完全独立的话，我们需要使用双 buffer 连续传输，但是阻碍我们将 HDMI 刷新并行到 FPGA 的唯一阻碍就是我们需要在 HDMI 显示叠加了推理结果的图像，目前我们是在 PS 端将推理结果绘制在原图像上再写入 DDR 供 vga 读取，因此在这里我们将 PS 端推理得到的推理结果传入 FPGA，FPGA 根据结果实时的在 PL 端的 vga 图像中叠加结果，这样 PS 端只用单独运行推理刷新程序，将其推理结果单方面的给 FPGA 即可实现绘框。

使用 PL 端绘制预测框，根本改进在于，完全不需要 ps 端进行内存复制，也就是 dvp 和 vga 是完全相关的，也就是 dvp 连续写，vga 连续读，dvp 写以及 vga 读地址切换直接由 PL 端控制，HDMI 显示唯一需要与 ps 端进行通信的就是推理结果的通信，同时该通信是单方向的，即推理程序发给 FPGA 无需管是否接收。

搭建的 pl_plot 模块其构成如下：

表 2.1.7　pl_plot 模块组成

模块名称	功能描述
vga_ctrl	生成 VGA 有效显示像素坐标交给 flag_generate，并利用传

	入框数控制框的显示，同时作为顶层模块例化 flag_generate，并产生相应 VGA 时序
flag_generate	生成一个由框、标签、置信度组成的显示单元
bin2bcd	将传入的置信度二进制转换为 BCD 码给 flag_generate 模块选择字模

实现该模块的思路就是，我们以框、标签、置信度作为一个显示单元，使用一个模块对其进行控制，通过重复例化该模块，我们可以控制显示单元的个数，也就是最多可以显示多少个框、标签以及置信度显示单元，同时将各个模块输出进行或操作就可以在显示图像中显示多个显示单元。

每一次显示我们需要 PS 端传入的参数有框数、每个框的标签、框的位置以及该标签的置信度。一个显示单元需要显示框、标签以及置信度。其中，框的显示我们只需要传入框的左上角坐标以及右下角坐标即可绘制矩形框，标签一共有四个，而置信度的数字我们限制为四位小数，因此需要显示的字模如下：

数字：0，1，2，3，4，5，6，7，8，9，.

标签：ca_shang、zhen_kong、zhe_zhou、zang_wu

其中，对于标签我们可以直接将这四种标签直接整体转换为字模进行显示，对于置信度我们就需要将 PS 端传入的二进制置信度转换为五个二进制 BCD 码，分别对应个位以及四个小数位，通过 BCD 码对当前位显示的数字字模进行选择。

数字字模大小为 16*16，字宽和字高设置为 16*16。对于标签，我们直接生成四个标签对应的字模，无需分开生成，所有字符的高度都必须相等，因此我们需要将其生成的字模调整高度统一为 16。以上字模我们将其存储在一维数组中，在综合时会被综合为 ROM。

表 2.1.8　标签字模

ca_shang	**zhen_kong**	**zhe_zhou**	**zang_wu**
0000000000000000;	0000000000000000;	0000000000000000;	0000000000000;
0000000000000000;	0000000000000000;	0000000000000000;	0000000000000;
0000000000000000;	0000000000000000;	0000000000000000;	0000000000000;
0000000040000000;	0040000040000000;	0040000000400000;	0000000000000;
0000000040000000;	0040000040000000;	0040000000400000;	0000000000000;
0000000040000000;	0040000040000000;	0040000000400000;	0000000000000;
0000000040000000;	0040000040000000;	7E40000000400000;	0000000000000;
3C3C003C5E3C5E3E;	7E5E3C5E004C3C5E3E;	FE5E3C007E5E3C42;	7E3C5E3E00DB42;
4646006662466244;	0C6246620058466244;	0E6246000C624642;	0C46624400DA42;

400E0060420E4244;	08427E420078424244;	1C427E0008424242;	080E4244005A42;
4076001C4276423C;	10424042006842423C;	3842400010424242;	1076423C006A42;
4246004242464240;	204242420044424240;	7042420020424246;	20464240006646;
664E0066424E427C;	60426642004666427C;	FF4266006042666E;	604E427C00646E;
383A003C423A4246;	7E421C420042184246;	FF421C007E42183A;	7E3A424600243A;
0000000000000042;	00000000000000042;	0000000000000000;	00000042000000;
0000FF000000007C;	00000000FF0000007C;	000000FF00000000;	0000007CFF0000;

表 2.1.9　数字字符字模

小数点.	0	1	2	3	4	5	6	7	8	9
00;	00;	00;	00;	00;	00;	00;	00;	00;	00;	00;
00;	00;	00;	00;	00;	00;	00;	00;	00;	00;	00;
00;	18;	00;	18;	18;	04;	00;	0C;	00;	10;	18;
00;	7E;	1C;	7E;	7E;	0E;	7E;	1C;	FF;	7E;	7E;
00;	7E;	3C;	FE;	FF;	1E;	7E;	1C;	7F;	EE;	EE;
00;	E7;	7C;	E7;	E7;	1E;	E0;	38;	06;	E7;	E7;
00;	E7;	7C;	07;	06;	3E;	FC;	78;	0E;	E6;	C7;
00;	E7;	1C;	0E;	1E;	7E;	FE;	7E;	0C;	7E;	E7;
00;	E7;	1C;	0E;	1E;	6E;	C7;	E7;	1C;	FE;	FE;
00;	E7;	1C;	1C;	07;	EE;	07;	E7;	18;	E7;	7E;
00;	E7;	1C;	38;	47;	FF;	47;	E7;	38;	C7;	1C;
00;	E7;	1C;	70;	E7;	FF;	C7;	E7;	38;	E7;	18;
F0;	7E;	1C;	FF;	7E;	0E;	FE;	FF;	38;	FE;	38;
60;	3C;	1C;	FF;	7C;	0E;	7C;	7E;	70;	7E;	70;
00;	00;	00;	00;	00;	00;	00;	00;	00;	00;	00;
00;	00;	00;	00;	00;	00;	00;	00;	00;	00;	00;

除了上述问题外，我们需要着重解决的一个问题是，从 PS 端得到的置信度是二进制的，我们在发送给 fpga 端前将其乘上 10000 取整发送，也就是相当于只取小数点后四位。因此我们需要提取置信度的个十百千万位用以选择在置信度区域需要显示的数字组合。

二进制转 BCD 我们使用“加 3 移位法”。

首先，我们需要得到五个 BCD 码，因此二进制码输入位数我们设置为 16 位，因为 5 个 BCD 码表示的范围为 0-99999，而 2 的 16 次方为 65536，对于输入小于 65536 的二进制数，我们都可以得到其对应的 5 个 BCD 码。

二进制转 BCD 转换步骤如下：

1. 对于 BCD 移位寄存器中的每 4 位的 BCD 数字，检测这个数是否大于 4。如果是，就在这个数字上加上一个 3。
2. 将整个 BCD 寄存器向左移动一位，将输入二进制序列的最高有效位（MSB）

移入到 BCD 寄存器的最低位（LSB）。

3. 重复步骤 1 和步骤 2，直到所有的输入位都被使用了。

为什么检查每一个 BCD 码是否大于 4，因为如果大于 4（比如 5、6），下一步左移就要溢出了，所以加 3，等于左移后的加 6，起到十进制调节的作用。

表 2.1.10 BCD 时序

时钟脉冲	移位结果（移位方向←）			输入的二进制码
	BCD 码高位	BCD 码次高位	BCD 码最低位	
	0000	0000	0000	11101011
1	0000	0000	0001	1101011
2	0000	0000	0011	101011
3	0000	0000	0111	01011
修正			+0011	
	0000	0000	1010	01011
4	0000	0001	0100	1011
5	0000	0010	1001	011
修正			+0011	
	0000	0010	1100	011
6	0000	0101	1000	11
修正		+0011	+0011	
	0000	1000	1011	11
7	0001	0001	0111	1
修正			+0011	
	0001	0001	1010	1
8	0010	0011	0101	
结果（十进制）	2	3	5	

在代码中，我们使用一个 FSMD 控制整个操作，循环访问二进制数的每一位，并且将调整电路与次态逻辑分开，使用单独的代码描述。

- **使用 resize 以及 pl_plot 后的 buffer 结构：**

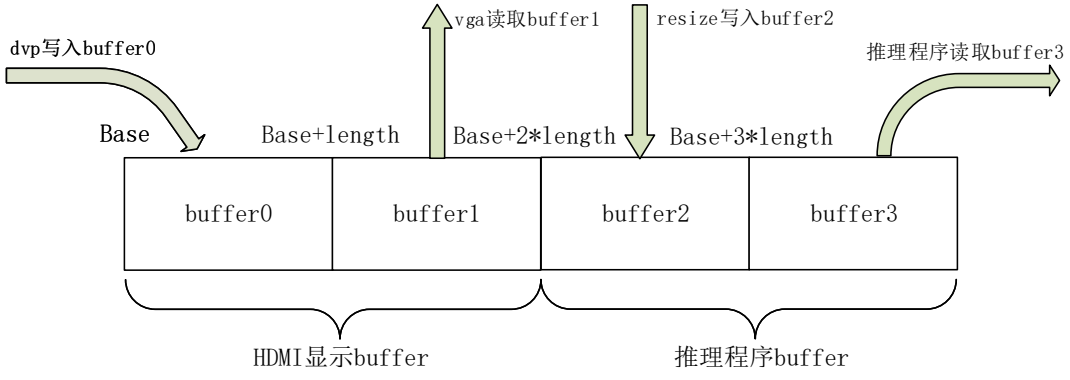


图 2.1.12 使用 resize 以及 pl_plot 后的 buffer 结构

使用 resize 以及 pl_plot 后的 buffer 结构如上，HDMI 显示完全由 PL 端控制，因此 HDMI 显示 buffer 是 PL 端使用。而推理程序 buffer 的写入是 PL 端写入 resize 图像，大小为 300*300，读出是 HPS 端推理程序读出 resize 图像用于推理。该 buffer 结构是最终的工程使用的结构。

- **使用 resize 以及 pl_plot 的 dvp_ddr3_vga 模块框图：**

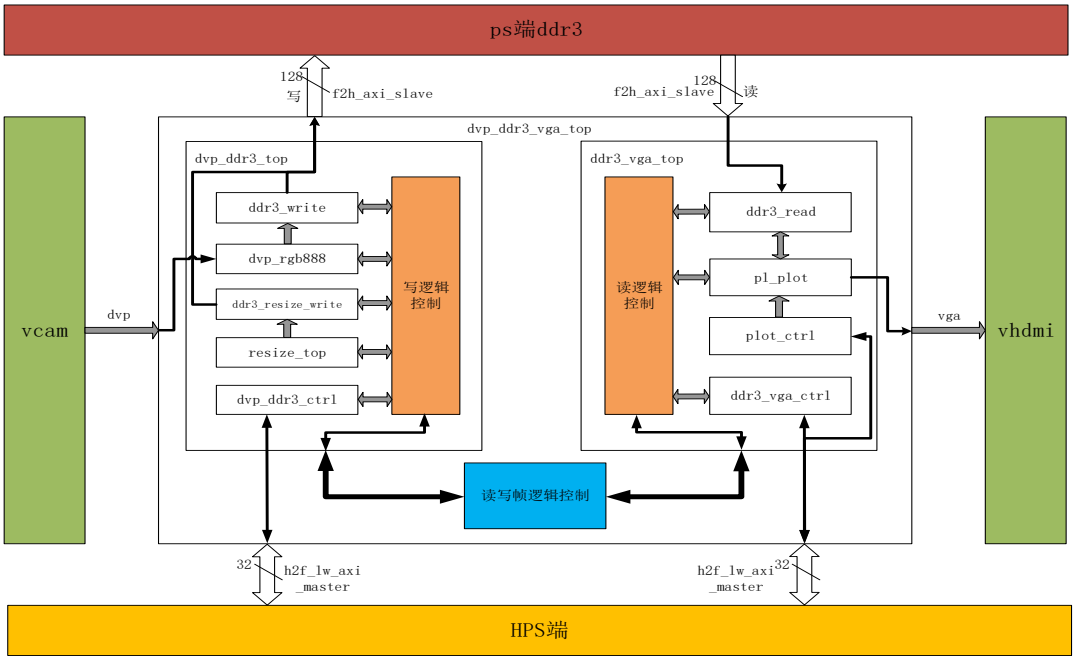


图 2.1.13　使用 resize 以及 pl_plot 的 dvp_ddr3_vga 模块框图

为了更方便的使用 resize 以及 pl_plot 模块，我们将其整合进入 dvp_ddr3_vga 模块。与原 dvp_ddr3_vga 模块相比较，增加了 resize 以及 pl_plot 模块后，对于写入 ddr，增加了一个 resize 图像的写主机，其帧写入控制与原图像写入是一样的，同时由于是两个写主机，Avalon 总线会自己进行仲裁，不需要我们进行控制先后传输。对于读 ddr，仍然和原来一样，只不过我们将 vga_ctrl 模块替换为了 pl_plot 模块，该模块通过读入 plot_ctrl 接收的推理结果信息，将其转换为推理结果显示单元在 vga 显示时与原图像叠加显示。

2.1.6 CNN 加速器优化

经过测试，推理过程中大部分时间花在了卷积的计算上，而 CNN 加速器承担着卷积计算的任务。因此我们可以通过时序约束，并利用 Chip Planner 进行逻辑固化，通过提升 CNN 加速器的运行频率以提升卷积运算的速度。同时考虑到

CNN 加速器通过 f2h_sdram 总线访问 PS 侧 DDR，完成数据的读取操作，我们也对 f2h_sdram 总线频率进行提升。

通过查阅文档发现，f2h_sdram 总线的最高频率可达到 200Mhz，但考虑到 CNN 访问 f2h_sdram 总线时，当两者同步时，才能获得最大的性能，因此我们需保证两者频率相同时，其相位也相同，故应使用同一时钟信号。

• CNN 运行频率提升

我们使用自建 PLL 替换 Demo 中所提供的 clk_cnn 组件，通过时序约束，Chip Planner 逻辑固化等手段，将 CNN 加速器时钟提升至 158Mhz，但为了保证其稳定性，我们在初赛验收中使用 **155Mhz** 进行。

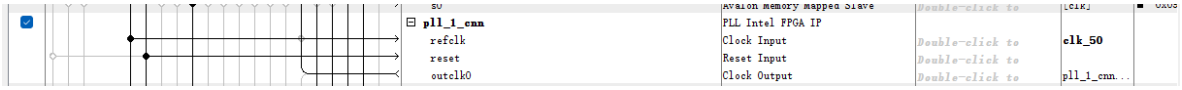


图 2.1.14 自建 PLL 锁相环产生时钟

其中 PLL 的输出 155Mhz 时钟将送给三个组件使用：

1. f2h_sdram0_clock 提高 f2h_sdram 总线时钟。
2. mm_bridge_sdram0 分线器组件。
3. cnn_top_0 CNN 加速器组件。

这里经过测试，我们需要保证 CNN 加速器组件的时钟与 f2h_sdram 总线的时钟一致，需保证相位、频率均相同。这样才能使 CNN 加速器组件与 sdram 组件的时钟同步，满足时序一致性，获得更大的性能。

• 时序约束

进行时序分析时，我们采用 Fast 1100mv 85C Model，相对来说更贴近我们的实验环境。

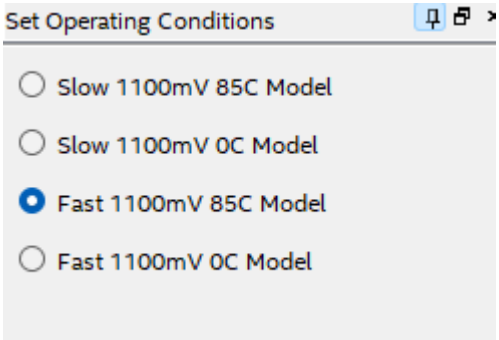


图 2.1.15 时序分析环境设置

Timing Analyzer 中的 Fmax 选项，可以帮助我们快速查看时钟信号的最高频率，如下图所示。我们可以看到 PLL1 的时钟可达到 177Mhz，满足我们手动约束的 155Mhz 时钟，可以看到我们的约束是有限的。

	Fmax	Restricted Fmax	Clock Name	Note
1	159.8 MHz	159.8 MHz	FPGA_CLK1_50	
2	177.81 MHz	177.81 MHz	u0|pll_1_cnn|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|divclk	
3	1503.76 MHz	1200.48 MHz	soc_system:u0|soc_system_hps_0:hps_0|soc_sy...am_inst|hps_sdram_pll:pll|afi_clk_write_clk	limit due to minimum period restriction (tmin)

图 2.1.16 时序分析 Fmax

我们通过 Timing Analyzer 对 PLL 产生的 150Mhz 时钟进行时序约束，让其在布线时尽可能满足该时钟的时序要求。

20	u0|pll_1_cnn|altera_pll_i|general[0].gpll~FRACTIONAL_PLL|vcoph[0]	Generated	2.857	350.02 MHz	0.000	1.428	50.00	1	7
21	u0|pll_1_cnn|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|divclk	Generated	5.714	175.01 MHz	0.000	2.857	50.00	2	1

图 2.1.17

并将其写入 sdc 文件，令 Quartus 自动识别。

```
create_generated_clock -name {u0|pll_1_cnn|altera_pll_i|general[0].gpll~FRACTIONAL_PLL|vcoph[0]}
-source [get_pins {u0|pll_1_cnn|altera_pll_i|general[0].gpll~FRACTIONAL_PLL|refclkin}] -duty_cycle
50.000 -multiply_by 7 -master_clock {FPGA_CLK1_50} [get_pins {
u0|pll_1_cnn|altera_pll_i|general[0].gpll~FRACTIONAL_PLL|vcoph[0] }]
create_generated_clock -name {u0|pll_1_cnn|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|divclk}
-source [get_pins {u0|pll_1_cnn|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|vco0ph[0]}]
-duty_cycle 50.000 -multiply_by 1 -divide_by 2 -master_clock
{u0|pll_1_cnn|altera_pll_i|general[0].gpll~FRACTIONAL_PLL|vcoph[0]} [get_pins {
u0|pll_1_cnn|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|divclk }]
create_generated_clock -name
{u0|pll_0_sdram|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|divclk} -source [get_pins
{u0|pll_0_sdram|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|vco0ph[0]}] -duty_cycle 50.000
-multiply_by 1 -divide_by 3 -master_clock
{u0|pll_1_cnn|altera_pll_i|general[0].gpll~FRACTIONAL_PLL|vcoph[0]} [get_pins {
u0|pll_0_sdram|altera_pll_i|general[0].gpll~PLL_OUTPUT_COUNTER|divclk }]
```

图 2.1.18

- **逻辑固化**

对于时序分析时，其建立时间不满足要求的信号，我们使用 Chip Planner 进行逻辑固化，**提高其时序收敛**。

	Slack	From Node	To Node	
	-5.753	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~17_OTERM1958	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.749	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~17_OTERM1958	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.734	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~25_OTERM1952	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.730	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~25_OTERM1952	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.724	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~17_OTERM1958	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.705	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~25_OTERM1952	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.672	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~13_OTERM1961	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u
	-5.668	soc_system:u0|cnn_top:cnn_top_0|load:load...ad_data:load_data_inst|Add20~13_OTERM1961	soc_system:u0|cnn_top:cnn_top_0|load:load_i...|load_data:load_data_inst|input_row_num[13]	u

图 2.1.19

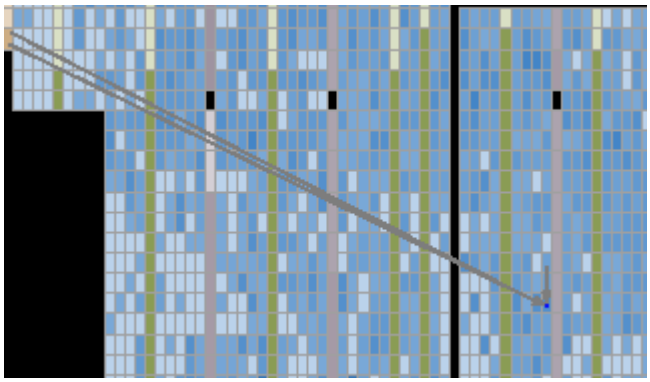


图 2.1.20

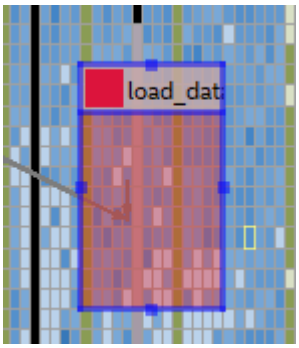


图 2.1.21

- **无用组件去除**

这里我们将无用组件移除，**减少逻辑资源**的使用，增强时序的收敛性，移除组件如下图所示。

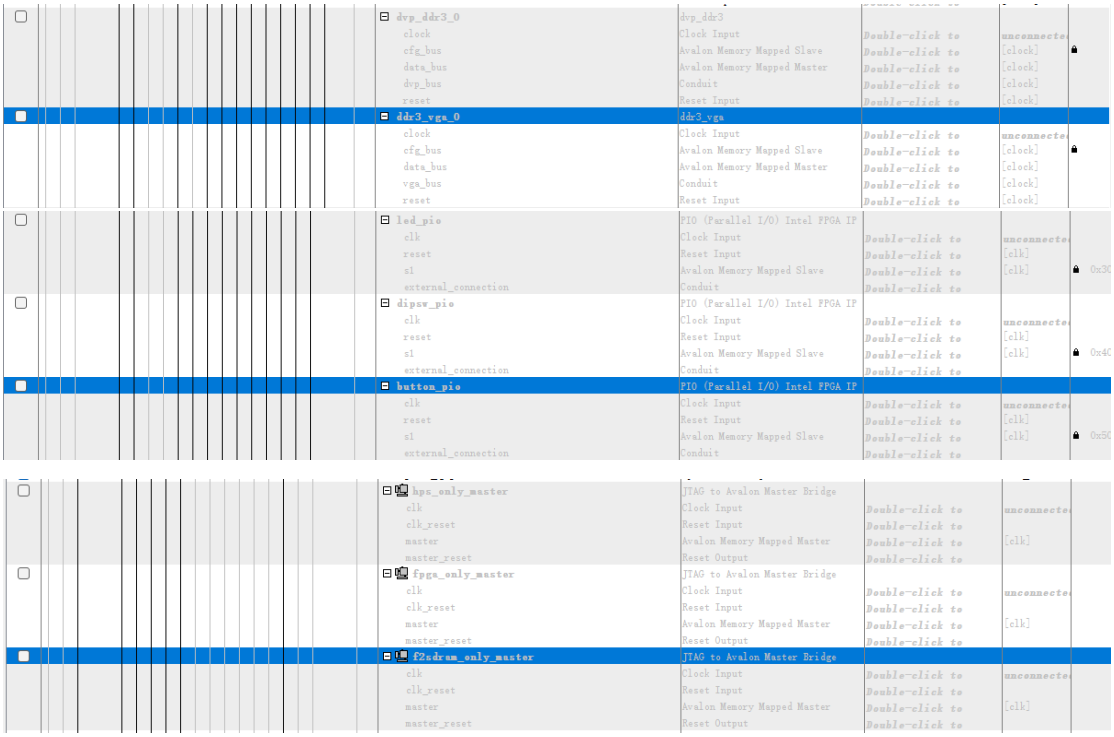


图 2.1.22

2.1.7 数据重排

通过阅读 ssd_deteciton.cc 文件，了解到框架提供了 Data_reoragnize_ip 的接口，如下图所示：

```
#if (REOGANIZE_TYPE == REOGANIZE_FPGA)
void data_reorganize(int mode, int in_c, int feature_map, int input_offset, int output_offset)
{
  foo_set(data_reorganize_ip, FPGAREG_REORG_MODE, mode);
  foo_set(data_reorganize_ip, FPGAREG_REORG_IN_C, in_c);
  foo_set(data_reorganize_ip, FPGAREG_REORG_FEATURE_MAP_SIZE, feature_map);
  foo_set(data_reorganize_ip, FPGAREG_REORG_DDR_INPUT_OFFSET, input_offset);
  foo_set(data_reorganize_ip, FPGAREG_REORG_DDR_OUTPUT_OFFSET, output_offset);

  int32_t status;
  status = foo_get(data_reorganize_ip, FPGAREG_REORG_START);
  status |= 0x1;
  foo_set(data_reorganize_ip, FPGAREG_REORG_START, status);
  status = foo_get(data_reorganize_ip, FPGAREG_REORG_START);
  #if (FPGA_SOURCE == FPGA_SOURCE_NK)
    while (!(status & 2)){
  #elif (FPGA_SOURCE == FPGA_SOURCE_AW)
    while (status & 1){
  #endif
      status = foo_get(data_reorganize_ip, FPGAREG_REORG_START);
    }
}
```

图 2.1.23 data_reorganize 接口函数

表 2.1.11 IP 对应接口控制寄存器

PS 端变量名	PL 端寄存器	功能
Mode	FPGAREG_REORG_MODE	模式选择
In_c	FPGAREG_REORG_IN_C	输入通道数
Feature_map	FPGAREG_REORG_FEATURE_MAP_SIZE	输入特征图尺寸
Input_offset	FPGAREG_REORG_DDR_INPUT_OFFSET	输入地址偏移
Output_offset	FPGAREG_REORG_DDR_OUTPUT_OFFSET	输出地址偏移
Status	FPGAREG_REORG_START	运行状态

我们根据上述所给控制寄存器以及 Avalon-MM 接口设计模板编写数据重排 IP。

- **设计思路**

我们通过阅读相关文档，确定此次数据重排为 8 通道重排。数据重排的操作，实质是变换输入数据的存储位置，使得**并行计算的卷积加速器能够读入正确的数据进行运算**。以 3*300*300 特征图为例，原始数据排列如图 2.1.24 所示，M 表示特征图尺寸，由于加速器并行运算，所以需要**并行读入数据**，例如第一次读入数据为每个通道的第一个元素（以蓝色标注），而按照原始排列，则会按顺序读

取 1~300*300~M+1。

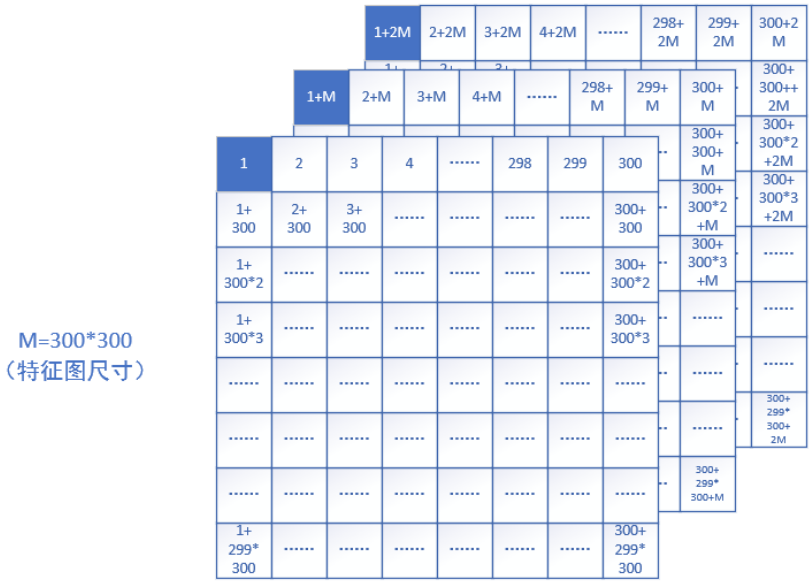


图 2.1.24 原始数据排列图

因此，我们需要用到数据重排，变换元素存储位置，并使其符合通道数（不足通道数填 0 补充），即更改为如图 2.1.25 所示。这使得每次读入是之前并行排列的数据。

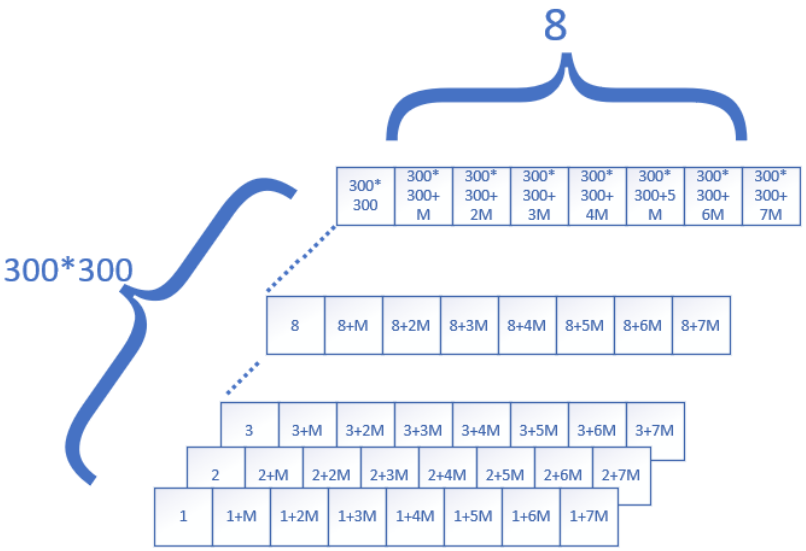


图 2.1.25 重排后数据排列图

设计框图如图 2.1.26 所示，主体我们分为三部分，所需参数控制寄存器读写模块，基于 Avalon_MM 接口的 DDR 内存读写模块以及关键的 DDR 读写控制模块。

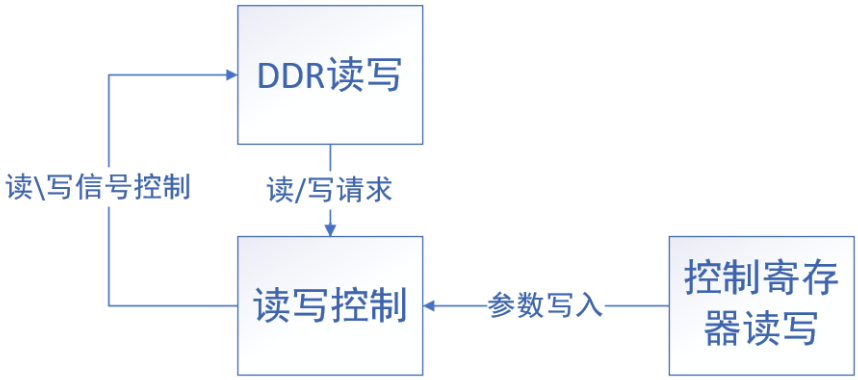
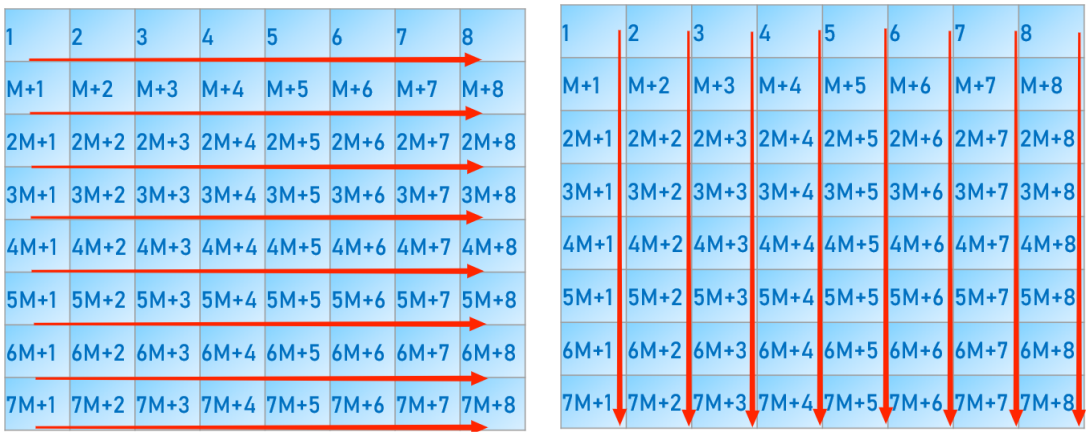


图 2.1.26 数据重排 IP 设计框图

我们的基本思路是设法**构建一个 8*8 的数据矩阵**，对于该矩阵，数据**按行读入**，之后**按列写出**，如图 2.1.27 所示。值得注意的是，按行读入，**每次读完 8 个连续地址的数据后需要变换读入地址**，偏移量为特征图尺寸，以符合数据重排的需要。DDR 内存读写模块基于异步 FIFO 设计,FIFO 深度我们设置为 8，这样能够加快读写速度并且符合每次读写 8 个数据的设计要求。



(a)读入　　(b)写出

图 2.1.27 读入写出示意图

如此遍历，直到 FPGAREG_REORG_FEATURE_MAP_SIZE 个位置遍历结束。而对于特征图尺寸大小不是 8 的整数倍情况，我们通过计算剩余的位置个数进行特殊对待。例如对于 32*3*3 的输入。8 次连续读入后剩余 1 个数据，这时我们便需要更改连续读入长度为 1，读取完成后，写入的长度也随之变更。

最终实现的 data_recognize 模块组成如下：

表 2.1.12　data_recognize 模块组成

模块名称	功能描述
Data_top	例化 ddr3_data_re_ctrl，ddr3_write，ddr3_read，data_ctrl

	四个模块
data_ctrl	对遍历位置、读写地址、control_go 信号进行控制，确定当前通道组数是否遍历完成，产生终止信号
ddr3_data_re_ctrl	当 HPS 需要查询状态时，与 HPS 通信获取相应的寄存器值
ddr3_read	将读 fifo 时序转换为读 avalon 时序
ddr3_write	将写 fifo 时序转换为写 avalon 时序

2.2 模型优化

2.2.1 智能归因

利用百度 BML 全功能 AI 开发平台，我们使用原始数据集以 SSD-Mobilenetv1 为网络进行预置模型调参。BML 平台能够在模型训练结束后给出智能归因报告，指出模型存在的一些问题与相应的解决方案。根据所给报告分析归因，主要得出如下几点：

1.针孔和擦伤的精度较低，由于四种类别特征相差较大，故将精度损失主要归因于漏识别。设法提高召回率；

图 2.2.1 预置模型调参训练各标签平均精度图

图 2.2.2 预置模型调参训练精度评估报告

2.针对擦伤单独分析，根据漏检率与目标框大小关系的样本分布，我们不难发现主要集中于小目标的漏检。并且，亮度对擦伤的检测精度影响也较大，过亮和过暗都导致目标的漏检。

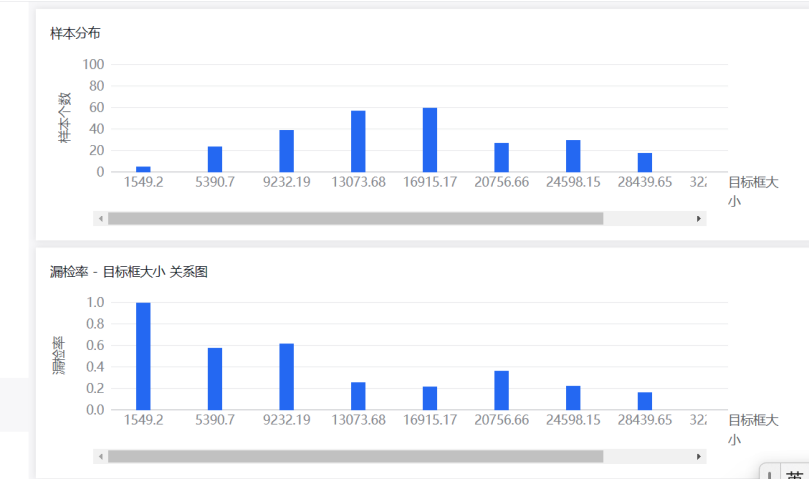


图 2.2.3 预置模型调参训练擦伤目标框大小关系分析报告

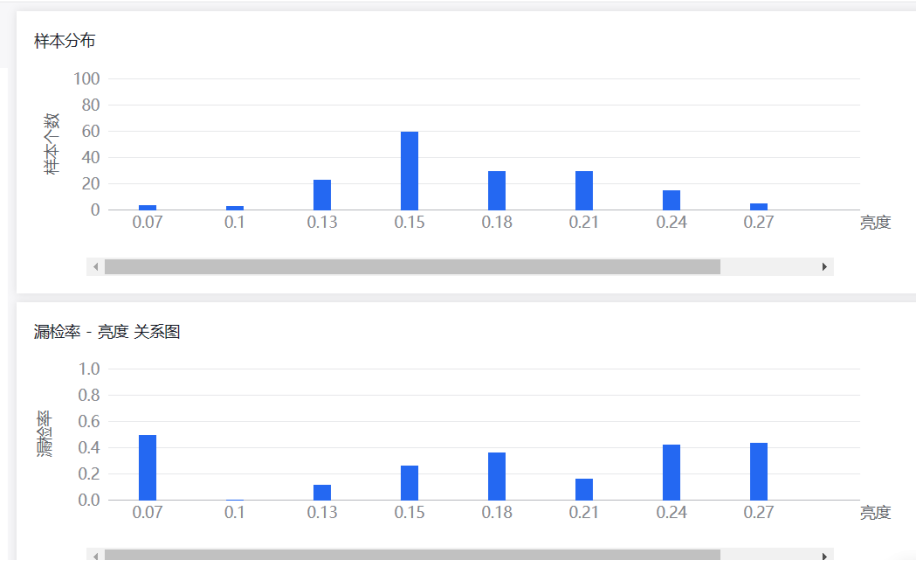


图 2.2.4 预置模型调参训练擦伤亮度关系分析报告

3.对针孔单独分析归因，不难发现针孔精度低主要是小目标检测困难，漏检率高。从漏检率与目标框大小的分布关系上，看出小目标在原始 SSD-MBV1 上识别困难。为此模型优化主要考虑小目标检测的问题。

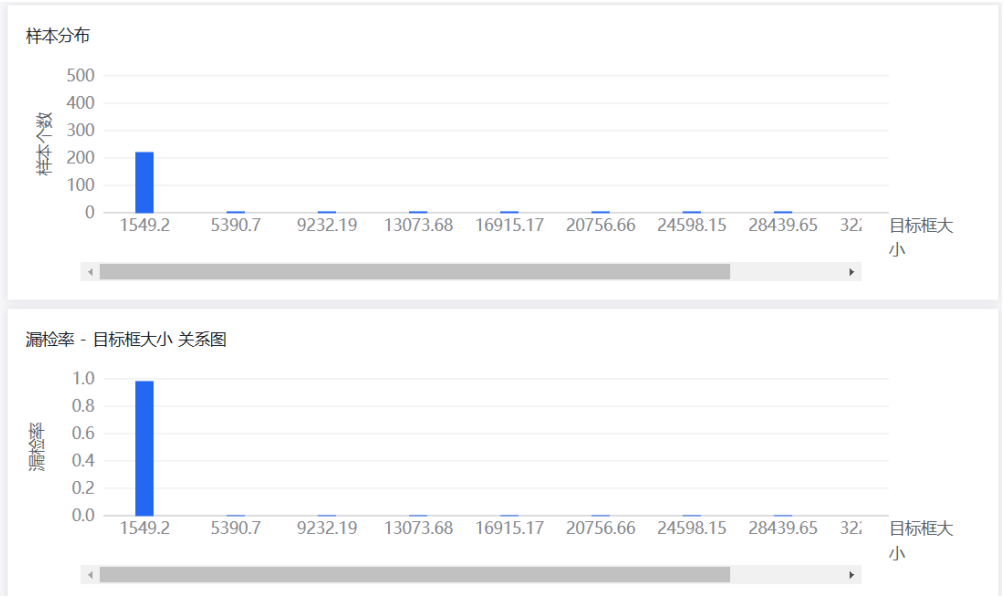


图 2.2.5　预置模型调参训练针孔目标框大小分析报告

我们据此对模型参数进行修改与制定相应的数据增强策略。

2.2.2 数据增强

根据上述对原始数据集存在问题的分析，确定数据增强的主要目的是让模型根据轮廓来识别目标物体，并能适应不同亮度环境。

百度 EasyData 智能数据服务平台提供数据采集、标注、清洗、加工等一站式数据服务，能够助力开发者高效获取 AI 开发所需高质量数据。我们利用 EasyData 平台对原始数据集进行数据增强操作。

我们采用图像全局以及标注框局部分别增强，在全局上采用 AutoContrast(自动对比度)、Posterize(减少每个颜色通道的 bits 至指定位数)、Brightness(亮度调整)和 Sharpness(锐度调整)增强方式串行叠加。

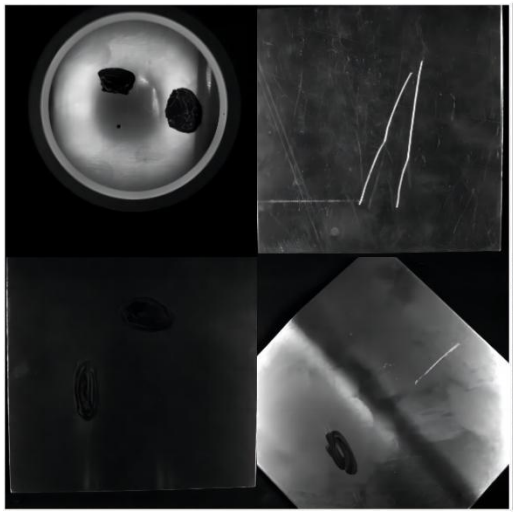


图 2.2.6　原始图片

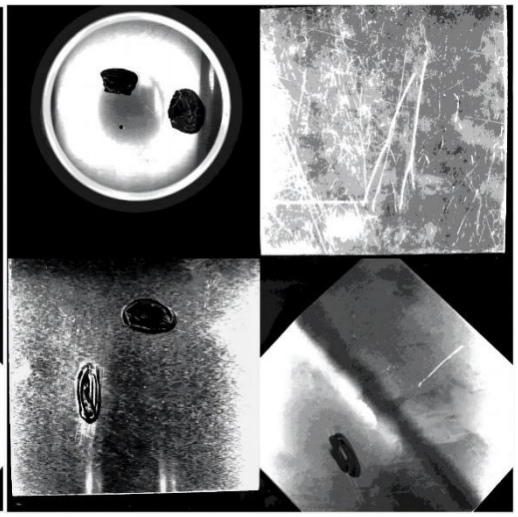


图 2.2.7　全局增强效果图

标注框局部增强更注重于能更好地根据轮廓而不是色彩来识别目标物体，因此采用 Equalize_Only_BBoxes(标注框图像直方图均衡)和 Solarize_Only_BBoxes(标注框图像色彩调整)方式。

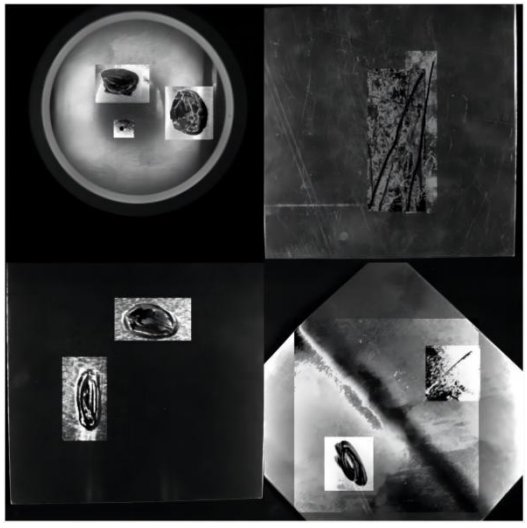


图 2.2.8　局部增强效果图

同时，在模型训练过程中发现，PaddleDetection 在 SSD-MobilenetV1 训练中采用的数据增强策略会导致预测框置信度偏低的现象。由于我们已经对数据集进行了数据增强操作，并且得到了较好的训练效果，故舍弃 RandomDistort、RandomExpand、RandomCrop 和 RandomFlip 这些数据增强方式。

```
sample_transforms:
  - Decode: {}
  # - RandomDistort: {brightness: [0.5, 1.125, 0.875], random_apply: False}
  # - RandomExpand: {fill_value: [127.5, 127.5, 127.5]}
  # - RandomCrop: {allow_no_crop: Fasle}
  # - RandomFlip: {}
  - Resize: {target_size: [300, 300], keep_ratio: False, interp: 1}
  - NormalizeBox: {}
  - PadBox: {num_max_boxes: 90}
```

图 2.2.9 sample_transforms 策略优化

2.2.3 基于敏感度分析的剪枝量化

在使用赛事方提供的 PaddleDetection 替代文件后，我们发现替换后仅支持训练量化模型，无法与剪枝策略同时使用。为使模型能够进行剪枝操作，在训练时采用分布进行的方式，首先训练全精度与剪枝模型，然后在另一个 PaddleDetection 下训练量化模型。为使训练量化模型时网络参数对应，修改 mobilenet_v1.py 文件。

如下图所示，在 mobilenet_v1.py 中对每个卷积层的输入输出通道数乘以对应的剪枝后保留率，并将保留率与剪枝率对应，以数组形式存储便于在训练过程中，当剪枝率发生变化后，量化训练的模型通道数能迅速对应，提升修改效率。

```
        dws21 = self.add_sublayer(
            "conv2_1",
            sublayer=DepthwiseSeparable(
                in_channels=int(32*prune_ratios[0] * scale),
                out_channels1=32*prune_ratios[0],
                out_channels2=64*prune_ratios[1],
                num_groups=32*prune_ratios[0],
                stride=1,
                scale=scale,
                conv_lr=conv_learning_rate,
                conv_decay=conv_decay,
                norm_decay=norm_decay,
                norm_type=norm_type,
                name="conv2_1"))
        self.dwsl.append(dws21)
        self._update_out_channels(64*prune_ratios[1], len(self.dwsl), feature_maps)
```

图 2.2.10 卷积层通道修改

```
prune_ratios=np.array([0.75,0.8125,0.625,0.5,   0.5,0.5625,0.5625,0.625,   0.4375,0.3125,0.6875,0.875,   0.875,0.875,0.9375,0.9375],dtype=float)
prune_ratios=1-prune_ratios
```

图 2.2.11 量化训练通道缩放比例

在剪枝算法选择上，采用 FPGM 剪枝算法对网络进行裁剪。由于每个卷积层

对整个网络在特定数据集的贡献权重不同，因此在选择合适的剪枝率时需要对每个卷积层进行敏感度分析。在剪枝率步进的选择上，由于加速器是 16 通道的，因此我们选择 1/16 为步进分析。

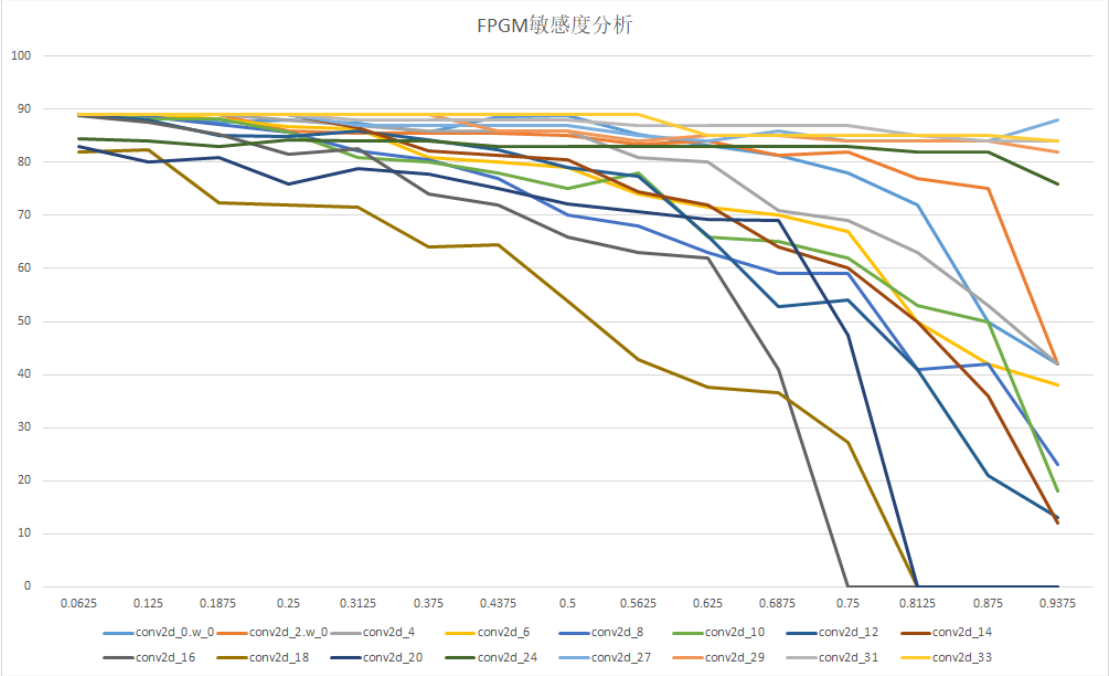


图 2.2.12　敏感度分析图

根据各层敏感度确定剪枝率，在仅主干网络剪枝的情况下，FLOPS 由原先的 **5.09GFlops** 下降为 **1.07GFlops，剪枝率达 78.8%**。

```
Pruner:
  criterion: fpgm
  pruned_params: [conv2d_0.w_0,conv2d_2.w_0,conv2d_4.w_0,conv2d_6.w_0,    conv2d_8.w_0,conv2d_10.w_0,'conv2d_12.w_0',
'conv2d_14.w_0',
  'conv2d_16.w_0','conv2d_18.w_0','conv2d_20.w_0','conv2d_24.w_0',    'conv2d_27.w_0','conv2d_29.w_0','conv2d_31.w_0',
'conv2d_33.w_0']
  pruned_ratios: [0.75,0.8125,0.625,0.5,  0.5,0.5625,0.5625,0.625,  0.4375,0.3125,0.6875,0.875,  0.875,0.875,0.9375,0.9375]
```

图 2.2.13 剪枝率设置图

2.2.4 基于 L1 和 FPGM 法则的综合剪枝策略优化

了解到《The Pruning Method of the CNN Based on the Comprehensive Assessment of the Convolutional Kernel》中提出一种基于 FPGM、L1 范数、BNSCALE 的混合剪枝方法。根据 PaddleDetection 默认剪枝程序流程，在卷积剪枝中会对每个卷积层各个通道通过不同计算方法得出重要性分数，并在排序后对重要性小的通道根据剪枝率进行裁剪，我们基于此思想提出在计算通道重要性时选择 **FPGM 和 L1 法则加权**得到重要性分数。

我们更改 paddleslim 源码，并通过源码安装 paddleslim 即可完成更改。

```
# ------------------------------------------------------------
        reduce_dims = [i for i in range(len(value.shape)) if i != pruned_axis]
        l1norm = np.mean(np.abs(value), axis=tuple(reduce_dims))
        if groups > 1:
            l1norm = l1norm.reshape([groups, -1])
            l1norm = np.mean(l1norm, axis=1)

        # sorted_idx = l1norm.argsort()

        dist_sum_list = []
        for out_i in range(value.shape[0]):
            dist_sum = self.get_distance_sum(value, out_i)
            dist_sum_list.append(dist_sum)
        scores = np.array(dist_sum_list)

        if groups > 1:
            scores = scores.reshape([groups, -1])
            scores = np.mean(scores, axis=1)

        sorted_idx = (scores/2+l1norm/2).argsort()
    # ------------------------------------------------------------
```

图 2.2.14 通道重要性分数加权

在模型训练中，由于分为剪枝侧和量化侧，在剪枝侧，我们在训练集与测试集划分比例为 9：1 情况下比较剪枝策略优化前后的 mAP 指标，发现 mAP 由原有的 86%提高到 91%，证明混合不同剪枝策略能够一定程度上提升检测精度。

2.2.5 输出层缩放

由于剪枝算法仅对主干网络 MobilenetV1 进行裁剪，输出层网络并未进行裁剪，这可能导致输出层的头部网络存在着较大的冗余。修改 mobilenet_v1.py 的源码对 6 个输出卷积层的输出通道数进行缩放，从而达到输出层剪枝的目的。我们选择 6 个输出卷积层通道数缩减为原有的 1/2，减少至 **4.27GFlops**。与主干网络剪枝结合后，整体模型 FLOPS 降低为 **0.93GFlops**，剪枝率达到 **81%**。

```
output_prune_ratios=np.array([0.5,0.5,0.5,0.5,0.5,0.5],dtype=float)
```

图 2.2.15 输出层缩放比例

```
            elif(i==4):
                tmp = self.add_sublayer(
                "conv5_" + str(i + 1),
                    sublayer=DepthwiseSeparable(
                        in_channels=int(512*prune_ratios[10]),
                        # in_channels=128,#128
                        out_channels1=int(512*prune_ratios[10]),
                        out_channels2=int(512*output_prune_ratios[0]),
                        num_groups=int(512*prune_ratios[10]),
                        stride=1,
                        scale=scale,
                        conv_lr=conv_learning_rate,
                        conv_decay=conv_decay,
                        norm_decay=norm_decay,
                        norm_type=norm_type,
                        name="conv5_" + str(i + 1)))
                self.dwsl.append(tmp)
                self._update_out_channels(int(512*output_prune_ratios[0]), len(self.dwsl), feature_maps)
        dws56 = self.add_sublayer(
            "conv5_6",
            sublayer=DepthwiseSeparable(
                in_channels=int(512* output_prune_ratios[0]* scale),
                out_channels1=int(512*output_prune_ratios[0]),
                out_channels2=1024*prune_ratios[11],
                num_groups=int(512*output_prune_ratios[0]),
```

图 2.2.16　输出卷积层通道修改举例

2.2.6 单通道模型

通过查看数据集各通道像素值，我们发现数据集中图片各通道像素值相同，即灰度图，只是以三通道形式存储。为利用其灰度图特点，缩短图像预处理时间，我们欲将图片单通道输入进行推理预测。为此，我们需要产生单通道模型，我们修改 PaddleDetection 源码即可实现。我们主要修改 mobilenetv1 的第一个卷积层与图像读取代码。

首先，我们将 mobilenetv1 第一个卷积层的输入通道数修改为 1，以保证图像仅输入一个通道数据。

```
self.conv1 = ConvBNLayer(
    in_channels=1,
```

图 2.2.17 卷积层修改

之后修改 ppdet/data/transform/operators.py 代码，将转换通道数的 cv2.COLOR_BGR2RGB 修改为 cv2.COLOR_BGR2GRAY，即转换为灰度图。由于转化后像素以二维数组存储，为使能够以正确格式输入网络，新建一个空的三维数组，通道数为 1，长宽与之前相同，并用得到的灰度值填充。并对 Resize，NormalizeImage 做相应修改，最终训练得到单通道输入模型。

```
im_copy = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
im= np.zeros((im.shape[0], im.shape[1], 1), np.uint8)
im[:,:,0]=im_copy
```

图 2.2.18 单通道灰度图填充

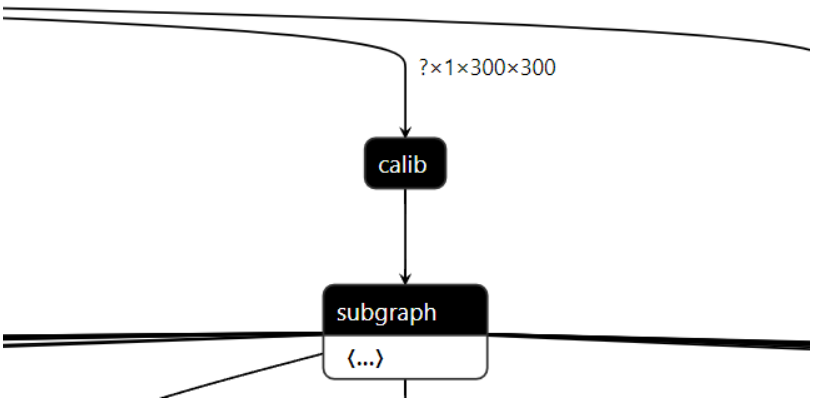


图 2.2.19 模型输入通道更改图

2.3 SDK 与 ssd_detction 优化

2.3.1 程序流程设计

由于赛题要求构建实时推理系统，因此我们重构了 ssd_detction 流程，系统程序架构我们上述已经讲到，这里来讲述具体的实现过程。

• ssd_detection 进程

程序运行过程如下图 2.3.2 所示，官方所提供 Demo 在进行推理前，除加载图像到 Tensor1 外，还需加载图像相关信息到 Tensor0 与 Tensor2 中（例如长宽、缩放因子等）。由于图像相关信息均相同，因此我们仅需加载一次即可，新的一帧图像到来仅需替换 Tensor1，如此便可加快系统推理刷新率。并且我们通过 Linux 的 CPU 绑定技术、修改进程优先级，以及修改 PaddleLite 底层源码，成功将推理进程绑定与双核心上使用双线程运行，极大程度提高了推理刷新速度以及 CPU 资源利用率。

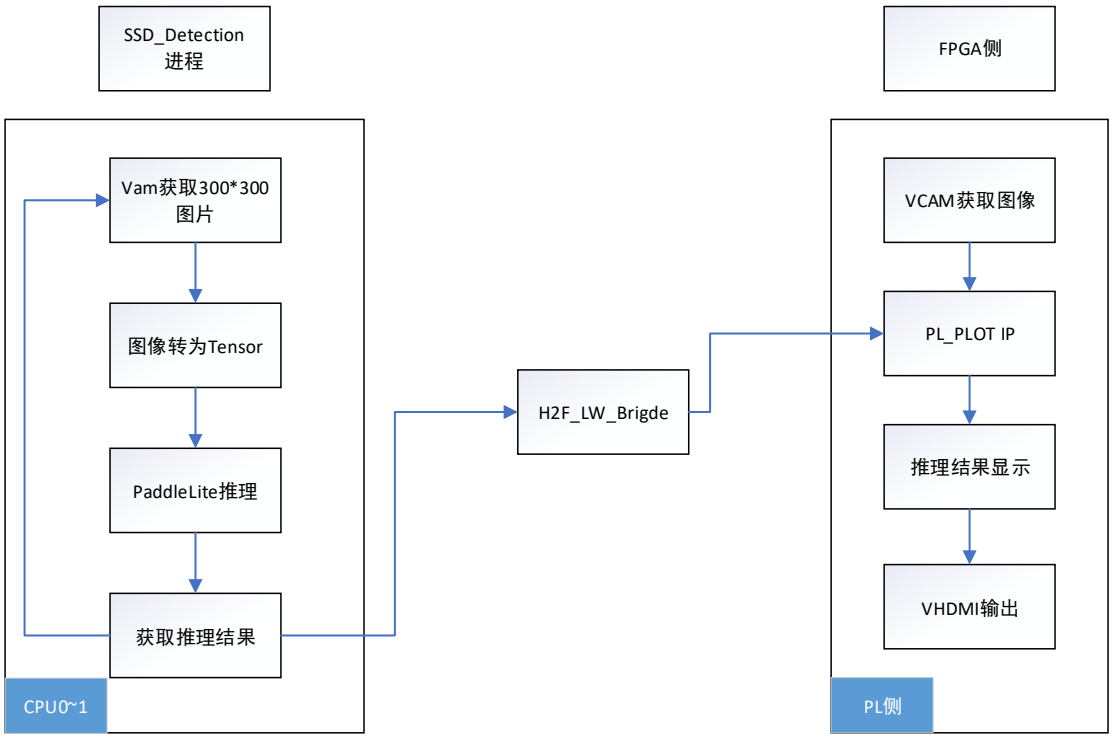


图 2.3.1 推理进程流程框图

• 2.3.2 数据重排 ARM 端优化

该部分内容主要讲述基于 NEON 指令集的数据重排优化，目前我们已将数据重排在 PL 侧完成初步设计，该部分内容将在未来展望部分讲解。

• 优化前后对比

可以看到优化前输入输出数据重排时间共 86ms+，优化后达到了 13ms。

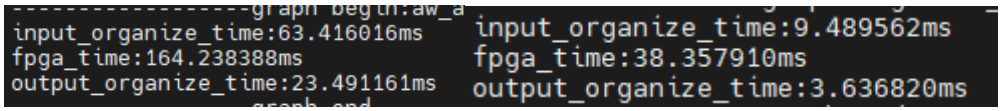


图 2.3.2 优化前后对比图

• 输入输出重排

对于 CNN 加速器读取参数时，每次访问 DDR 都是列优先访问，如遍历以下矩阵，则按列有限访问 1，5，9，13。每次读取的跨度大，访问数据内存不连续，由于 CPU 的 Cache 块大小有限，不能一次加载所有数据，在读取时会由于内存不连续出现大量的 Cache miss，将会花费大量的时间。

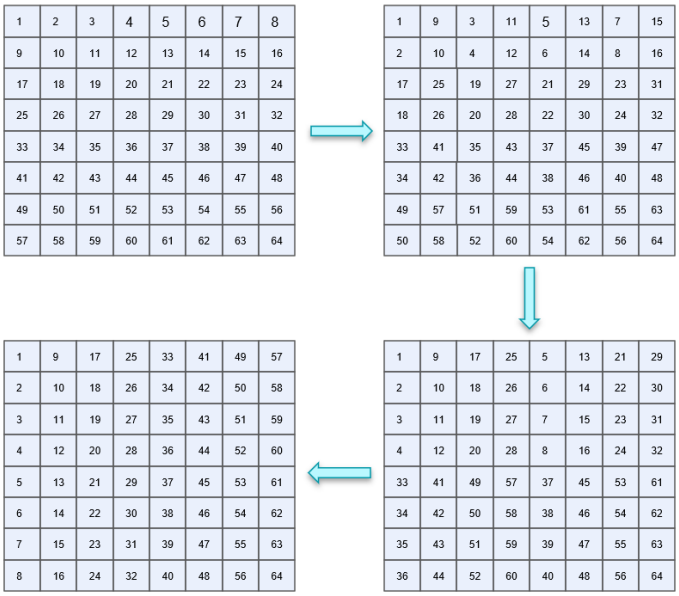


图 2.3.3 重排示意图

因此可以首先将数据进行重排，将矩阵进行一个转置后存储，此时数据相邻，地址连续，可以充分利用数据总线带宽，减少内存的访问次数，如上图所示。

• Neon

Arm Neon 技术，即高级 SIMD（单指令多数据）用于实施 Armv8-A 或 Armv8-R 架构配置文件的架构扩展。

Neon 技术为指令集架构提供了专门的扩展，提供可以对多个数据并行执行数学运算的附加指令流。

这可以通过加速音频和视频编码/解码、用户界面、2D/3D 图形或游戏来改善多媒体用户体验。 Neon 还可以加速信号处理算法和功能，以加速音频和视频处理、语音和面部识别、计算机视觉和深度学习等应用。

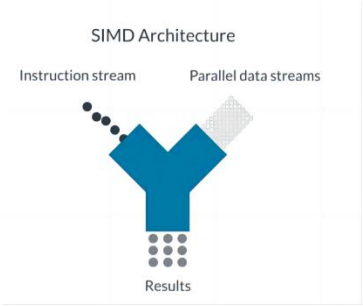


图 2.3.4 NEON 技术示意图

在移动平台上进行一些复杂算法的开发，一般需要用到指令集来进行加速。NEON 技术是 ARM Cortex™-A 系列处理器的 128 位 SIMD（单指令，多数据）架

构扩展，专门针对大规模并行运算设计的，旨在为消费性多媒体应用程序提供灵活、强大的加速功能，从而显著改善用户体验。

• 矩阵运算加速重排

数据重排本质上就是一个矩阵转置运算，我们可以通过线性代数相关知识，将大矩阵分解为若干个小矩阵，对小矩阵进行单独的数据重排，而小矩阵的数据重排，我们可以利用 ARM 的 Neon 指令集进行加速，如此便可将 60+ms 的数据重排时间缩短到 16ms。

运算过程如下所示：

1. 原始 8*8 矩阵如图 2.3.5 所示。

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

图 2.3.5 原始数据排列图

2. 对相邻两行。即 1、2 行，3、4 行，5、6 行，7、8 行进行矩阵转置操作，结果如下图 2.3.6 所示。

1	9	3	11	5	13	7	15
2	10	4	12	6	14	8	16
17	25	19	27	21	29	23	31
18	26	20	28	22	30	24	32
33	41	35	43	37	45	39	47
34	42	36	44	38	46	40	48
49	57	51	59	53	61	55	63
50	58	52	60	54	62	56	64

图 2.3.6 转置后数据排列图

3. 将第 1、3 行，2、4 行，5、7 行使用 vturn 指令进行矩阵转置操作，结果如下图 2.3.7 所示。

1	9	17	25	5	13	21	29
2	10	18	26	6	14	22	30
3	11	19	27	7	15	23	31
4	12	20	28	8	16	24	32
33	41	49	57	37	45	53	61
34	42	50	58	38	46	54	62
35	43	51	59	39	47	55	63
36	44	52	60	40	48	56	64

图 2.3.7

4. 将每四个行相邻的地址，打包成一个元素，对 1、5 行，2、6 行，3、7 行进行矩阵转置操作，结果如下图 2.3.8 所示。如此一来，便通过 SIMD 技术完成了数据重排的加速处理

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

图 2.3.8

• OpenMP

OpenMP 是一种用于共享内存并行系统的多线程程序设计方案，支持的编程语言包括 C、C++和 Fortran。OpenMP 提供了对并行算法的高层抽象描述，特别适合在多核 CPU 机器上的并行程序设计。编译器根据程序中添加的 pragma 指令，自动将程序并行处理，使用 OpenMP 降低了并行编程的难度和复杂度。

OpenMP 是一种显式（非自动）编程模型，为程序员提供对并行化的完全控制。 一方面，并行化可像执行串行程序和插入编译指令那样简单。另一方面，像插入子程序来设置多级并行、锁、甚至嵌套锁一样复杂。

OPENMP 编程模型：

内存共享模型：OpenMP 是专为多处理器/核，共享内存机器所设计的。底层架构可以是 UMA 和 NUMA。即(Uniform Memory Access 和 Non-Uniform Memory Access)，如下图所示

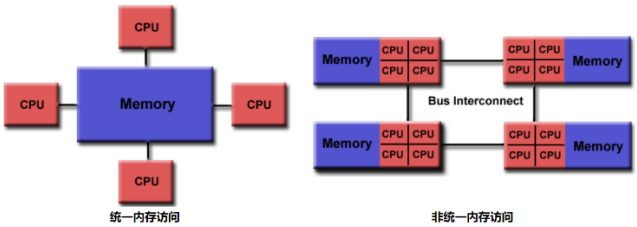


图 2.3.9 OPENMP 示意图

具体实现：

1. 考虑到输出数据重排多层 for 循环嵌套了 tran_8()函数，对于 for 循环可以使用 OpenMP 进行多线程加速。使用编译制导指令，将串行程序转变为并行程序，如下图 2.3.10 所示。

```
#pragma omp parallel for
for (int i = 0; i < up_round(c, INPUT_EXTEND_SCALE); i++)
{
  tran_8((uint8_t *)din + i * area * INPUT_EXTEND_SCALE, (uint8_t *)dout + i * area * INPUT_EXTEND_SCALE, INPUT_EXTEND_SCALE, area);
}
```

图 2.3.10

2. 由于输入输出重排的循环调用有不同之处，这里我们分别对输入和输出重排使用不同的 tran_8()函数，以寻求最优化，对于输入重排，我们将函数内重排循环的最外层，使用#pragma omp parallel for 编译制导指令，利用 OpenMP 进行多线程优化。

```
#pragma omp parallel for
for (y = 0; y < gy_l; y += 8)
{
  #pragma unroll(32)
  // #pragma omp parallel for
    for (x = 0; x < gx_l; x += 8)
    {
        // laden 8 Reihen Daten
        reg882_0.val[0] = vld1_u8(&gbild[y][x]);
        reg882_0.val[1] = vld1_u8(&gbild[y + 1][x]);
        reg882_1.val[0] = vld1_u8(&gbild[y + 2][x]);
        reg882_1.val[1] = vld1_u8(&gbild[y + 3][x]);
        reg882_2.val[0] = vld1_u8(&gbild[y + 4][x]);
        reg882_2.val[1] = vld1_u8(&gbild[y + 5][x]);
        reg882_3.val[0] = vld1_u8(&gbild[y + 6][x]);
        reg882_3.val[1] = vld1_u8(&gbild[y + 7][x]);

        // je 2 Reihen transponieren
        reg882_0 = vtrn_u8(reg882_0.val[0], reg882_0.val[1]);
        reg882_1 = vtrn_u8(reg882_1.val[0], reg882_1.val[1]);
        reg882_2 = vtrn_u8(reg882_2.val[0], reg882_2.val[1]);
```

图 2.3.11 利用 OpenMp 多进程优化

- 去除 memset

输入数据重排中，官方代码利用 memset 函数进行置 0 操作，经过调试分析，这部分执行过程并未有实际意义，可以进行省略。

```
#elif (ARMREOG_TYPE == ARMREOG_NEON)
  int high = h + 2 * pad;
  int width = w + 2 * pad;
  int area = high * width;
  // int channel_count = up_round(c, INPUT_EXTEND_SCALE) * INPUT_EXTEND_SCALE;

  tran_8_1((uint8_t *)din, (uint8_t *)dout, area, up_round(c, INPUT_EXTEND_SCALE) * INPUT_EXTE
  // tran_8((uint8_t *)din,(uint8_t *)dout,area,c);

  for (int cc = 0; cc < area; cc++)
  {
    memset(dout + cc * INPUT_EXTEND_SCALE + c, 0, INPUT_EXTEND_SCALE - c);
  }

#endif
}
```

图 2.3.12 涉及 memset 部分代码

修改后如下图所示，由于输入输出数据重排都是基于 tran_8 函数实现，但两者过程有些许不同，其中的 tran_8_1 是利用 OpenMP 技术单独对输入数据重排进行优化的代码。去除 memset 前后数据重拍所需时间对比如下图 2.3.13 所示。

```
------------------graph begin------
input_organize_time:44.553375ms
fpga_time:37.366497ms
output_organize_time:4.027653ms
------------------graph end------
```

```
------------------graph begin------
input_organize_time:9.831853ms
fpga_time:37.320518ms
output_organize_time:3.698724ms
------------------graph end------
```

图 2.3.13 去除 memset 时间对比图

2.3.3 开启编译器 03 优化

由于编译器自带优化功能，而默认状态下使用 O1 级别优化，不同级别的优化区别如下所示：

O1 优化：在不影响编译速度的前提下，尽量采用一些优化算法，降低可执行代码的大小。

O2 优化：牺牲部分编译速度，除了执行 -O1 所执行的所有优化之外，还会采用几乎所有的目标配置支持的优化算法，用以提高目标代码的运行速度。

O3 优化：除了执行 -O2 所有的优化选项之外，一般都是采取很多向量化算法，提高代码的并行执行程度，利用现代 CPU 中的流水线、Cache 等。这个选项会提高执行代码的大小，而可以提高目标代码的运行速度。

开启 O3 优化前后输入输出重排时间对比如下图所示。

```
------------------graph begin------
input_organize_time:34.168159ms
fpga_time:37.798752ms
output_organize_time:6.577938ms
------------------graph end------
```

```
------------------graph begin------
input_organize_time:9.831853ms
fpga_time:37.320518ms
output_organize_time:3.698724ms
------------------graph end------
```

图 2.3.14 开启 03 优化对比图

2.3.4 循环展开

在C语言中，循环展开技术是一种提升程序执行速度的非常有效的优化方法，它可以由程序员手工编写，也可由编译器自动优化。循环展开的本质是，利用CPU指令级并行，来降低循环的开销，当然，同时也有利于指令流水线的高效调度。

循环展开的优点：

1. 减少了分支预测失败的可能性。
2. 增加了循环体内语句并发执行的可能性，当然，这需要循环体内各语句不存在数据相关性。
3. 能够显著对tight loop，比较紧凑的，循环体内代码较少的循环带来速度提升。假如unroll factor = 5，那么loop指令的开销将降为之前的1/5，是很大的一个性能提升。
4. CPU架构中EX阶段的运算单元，比如浮点运算单元，往往是pipeline的，多个stage才能完成。通过unrolling，能够让运算单元的pipeline排得更满。

- **具体实现**

由于现代编译器已经足够优异，我们不需要手动进行循环展开，只需要对需要展开的循环前加入编译前导指令#pragma unroll (16) ，便可以利用编译器对其进行循环展开，此外由于我们上述已经开启了O3优化，编译器其实已经自动帮我们进行了循环展开，我们手动插入#pragma unroll的意义是寻找最佳的展开深度，进一步进行优化。

例如在输出数据重排中，我们共同利用了OpenMP技术以及循环展开技术，如下图所示，成功的将输出数据重排优化到3.6ms。

```
#elif (ARMREOG_TYPE == ARMREOG_NEON)
  int area = h * w;
  #pragma omp parallel for
  #pragma unroll(32)
  for (int i = 0; i < up_round(c, INPUT_EXTEND_SCALE); i++)
  {
    tran_8((uint8_t *)din + i * area * INPUT_EXTEND_SCALE, (uint8_t *)dout + i * area * INPUT_
  }

#endif
}
#endif
```

图 2.3.15 OpenMP与循环展开技术联合

2.3.5 PaddleLite 优化

• 修复 PaddleLite 能耗模式 BUG

通过阅读 PaddleLite 源码，发现 PaddleLite 底层很大程度上利用 OpenMP 进行多线程优化，将串行执行的程序进行并行执行，但官方提供的 PaddleLite 实际上并未有 bug，并未开启多线程模式，因此我们给出了如下解决方案。

PaddleLite 加载模型时，会选择能耗模式，如下图所示，可根据需求选择不同的能耗，由于该 Cyclone V 代芯片具有双核 ARM-A9 处理器，因此我们选择 LITE_POWER_HIGH 模式，并开启多线程运行。

```
std::shared_ptr<PaddlePredictor> LoadModel(std::string model_file,
                                           int num_theads) {
  MobileConfig config;
  config.set_model_from_file(model_file);
  config.set_threads(num_theads);
  config.set_power_mode(CPU_POWER_MODE);
  std::shared_ptr<PaddlePredictor> predictor =
  CreatePaddlePredictor<MobileConfig>(config);
 return predictor;

}
```

图 2.3.16

选项	说明
LITE_POWER_HIGH	绑定大核运行模式。如果ARM CPU支持big.LITTLE，则优先使用并绑定Big cluster。如果设置的线程数大于大核数量，则会将线程数自动缩放到大核数量。如果系统不存在大核或者在一些手机的低电量情况下会出现绑核失败，如果失败则进入不绑核模式。
LITE_POWER_LOW	绑定小核运行模式。如果ARM CPU支持big.LITTLE，则优先使用并绑定Little cluster。如果设置的线程数大于小核数量，则会将线程数自动缩放到小核数量。如果找不到小核，则自动进入不绑核模式。
LITE_POWER_FULL	大小核混用模式。线程数可以大于大核数量。当线程数大于核心数量时，则会自动将线程数缩放到核心数量。
LITE_POWER_NO_BIND	不绑核运行模式（推荐）。系统根据负载自动调度任务到空闲的CPU核心上。
LITE_POWER_RAND_HIGH	轮流绑定大核模式。如果Big cluster有多个核心，则每预测10次后切换绑定到下一个核心。
LITE_POWER_RAND_LOW	轮流绑定小核模式。如果Little cluster有多个核心，则每预测10次后切换绑定到下一个核心。

图 2.3.17

但加载模型时，会出现如下图 2.3.18 提示，即监测 CPU0 状态失败，自动切换到不绑核运行模式，使用 taskset -p 指令，可以查到该进程只在一个 CPU 上运行，如图 2.3.21 所示，以上信息说明，我们的模型运行时，并未按照设定的多核运行，考虑到 PaddleLite 底层有部分代码使用 OpenMP 实现，使用多核心进行推

理可以提升部分推理速度。

```
[I  5/24 12:50:20.521 ...nna/Paddle-Lite/lite/core/device_info.cc:509 check_cpu_online] Failed to query the online statu
e of CPU id:0
[I  5/24 12:50:20.521 ...nna/Paddle-Lite/lite/core/device_info.cc:514 check_cpu_online] CPU id:0 is offine
[W  5/24 12:50:20.522 ...nna/Paddle-Lite/lite/core/device_info.cc:1352 SetRunMode] Some cores are offline, switch to NO
BIND MODE
```

图 2.3.18 CPU0 监测报错

```
root@awcloud:/opt/paddle_frame# taskset -p 2429
pid 2429's current affinity mask: 2
```

图 2.3.19

解决方案

通过调试，定位到 PaddleLite 用于绑定核心的程序 device_info.cc 位于 Paddle-Lite/lite/core 目录下，绑定线程的程序如下图所示。

```
bool bind_threads(const std::vector<int> cpu_ids) {
#ifdef ARM_WITH_OMP
  int thread_num = cpu_ids.size();
  omp_set_num_threads(thread_num);
  std::vector<int> ssarets(thread_num, 0);
#pragma omp parallel for
  for (int i = 0; i < thread_num; i++) {
    ssarets[i] = set_sched_affinity(cpu_ids);
  }
  for (int i = 0; i < thread_num; i++) {
    if (ssarets[i] != 0) {
      LOG(ERROR) << "Set cpu affinity failed, core id: " << cpu_ids[i];
      return false;
    }
  }
#else   // ARM_WITH_OMP
  std::vector<int> first_cpu_id;
  first_cpu_id.push_back(cpu_ids[0]);
  int ssaret = set_sched_affinity(first_cpu_id);
  if (ssaret != 0) {
    LOG(ERROR) << "Set cpu affinity failed, core id: " << cpu_ids[0];
    return false;
  }
#endif  // ARM_WITH_OMP
  return true;
}
```

图 2.3.20

阅读以上程序，我们可以看到 PaddleLite 是通过查询 /sys/devices/system/cpu/cpu0 目录下的 online 文件，来确认该核心是否可用，并加以绑定。但由于 HPS 侧的 CPU0 为主 CPU，其默认在线，因此文件夹中并未含有 online 文件，如下图所示。

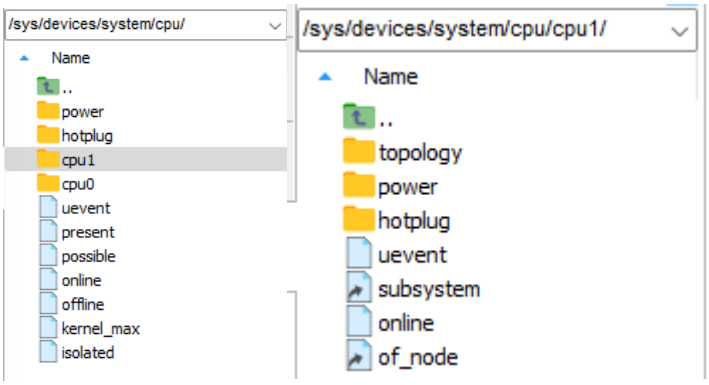


图 2.3.21

因此当 PaddleLite 查询 online 时，误以为 CPU0 offline，故默认将 CPU 绑定在 CPU1 上运行，并且自动将 PaddleLite 能耗模式切换为 LITE_POWER_NO_BIND 模式，该 bug 极大程度限制了系统性能。

由于我们确认两个 CPU 均在线，这里我们将其返回 True，并默认将 PaddleLite 绑定到两个 CPU。修改后通过 tasket -p 指令查询进程的运行核心，可以看到 mask 为 3，转换为二进制数为 11。代表我们成功将模型的推理过程运行在两个核心上，充分利用了硬件资源。

修改前后推理刷新时间如下图所示 (图中 FPS run 代表推理结果刷新一帧的速率，该时间为推理进程全流程时间，包括数据的读取、图片的预处理、数据重拍、推理过程、以及推理结果输出。该时间即为比赛规则中的推理刷新速率)

推理刷新速度	25分	推理刷新速度达到1帧/秒，可得基础分5分； 每超出100ms，扣0.5分，0分为限； 每减少30ms，加0.7分，25分为限

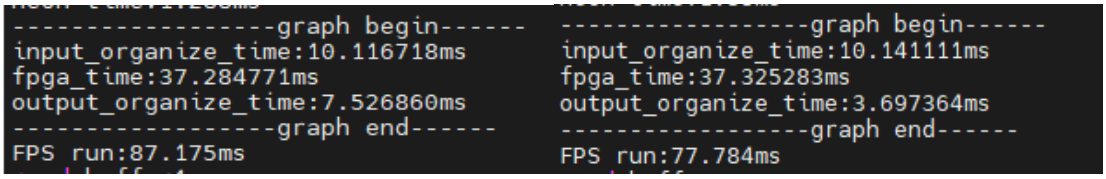


图 2.3.22 修改前后对比

可以看到，经过修复 PaddleLite 底层 bug，将推理刷新速度提升了 10ms。

2.3.6 修改 PaddleLite 为单通道

通过对数据集的调研，可以发现数据集为灰度图，但数据集图片仍为 RGB888 格式，24bit 位深，这也意味着图片中有 2/3 的数据为无效数据，我们要做的就是剔除这些无效数据。此外我们也将重写 dvp_ddr，将传入 DDR 的图像数据直接转换为单通道，可进一步节省数据读入读出时间。

单通道、三通道的处理时间主要差距在图片的预处理上，对于推理时间并未有较大的影响，因为将三通道模型与单通道模型进行对比，其只减少了 1%的 FLOPS，下面我将从以下几个方面分析单通道系统的优势：

1. 首先若 dvp_ddr 输入为单通道，将 ddr 原始数据转换为 OpenCv 的 MAT 格式时，由于单通道，其像素间连续，可以通过 memcpy 一次读取一行的像素点，而三通道由于 RGB 排列，需要利用指针对每一个像素进行单独赋值，这部分由于

地址连续可以节省几 ms 的时间。

2. Resize 时间，对于 400*320 的像素，我为了保证输入图片与数据集长宽比相同，我只使用了 400*300 的像素点，并通过 resize 缩放为 300*300 的图像，单通道可以减少 resize 的时间。

3. Convert 转换时间，由于兼容 PaddleLite 的 tensor 输入需要将像素值放缩为（-1，1）的 float 范围内，需要使用 cv::convert 函数将 u8 图片转换为 f32 格式，这部分需要较大的时间。

4. Nhwc->nchw 格式转换时，这部分需要根据均值和方差进行图片的放缩，并且进行数据格式的转换，从 RGBRGBRGB 格式转换为 RRRGGGBBB 格式，如下图 2.3.23 所示这部分使用 neon 指令集进行加速，并使用循环展开进一步缩短时间。

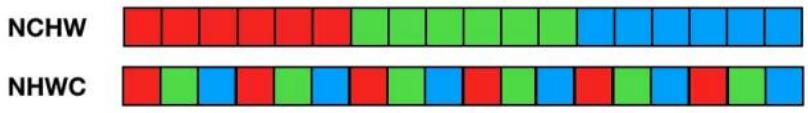


图 2.3.23　Nhwc NHWC 排列示意图

2.3.7NHWC->NCHW 通道转换的重写

- **官方 demo 的错误修改**

修改前后如下图所示，送入 PaddleLite 的 Tensor 应归一化为(-1,1)范围内，Demo 中在 NHWC->NCWH 通道转换的同时进行归一化操作，但经过阅读源码发现，这里应该是除以 scale，而非乘法。这是官方提供 Demo 中的一点小错误，这里我们进行更正。后续我们在 Github 阅读 PaddleLite 源码时，也证实了这一点。

```
for (; i < size; i++) {
  *(dout_c0++) = (*(din++) - mean[0]) * scale[0];
  *(dout_c0++) = (*(din++) - mean[1]) * scale[1];
  *(dout_c0++) = (*(din++) - mean[2]) * scale[2];
}
```

```
for (; i < size; i++) {
  *(dout_c0++) = (*(din++) - mean[0]) / scale[0];
  *(dout_c0++) = (*(din++) - mean[0]) / scale[0];
  *(dout_c0++) = (*(din++) - mean[0]) / scale[0];
}
```

图 2.3.24　Tensor 归一化

修改方式

该部分代码主要完成了两个功能,并且利用 NEON 指令进行并行加速：

1. 归一化，（当前像素值-均值）/方差

2. 通道转化。

原代码该过程如下图所示：

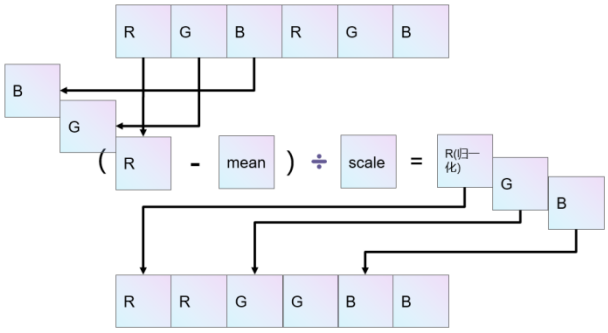


图 2.3.25

单通道代码示意图如下图所示（此时仅需要归一化即可，并不需要重排）：

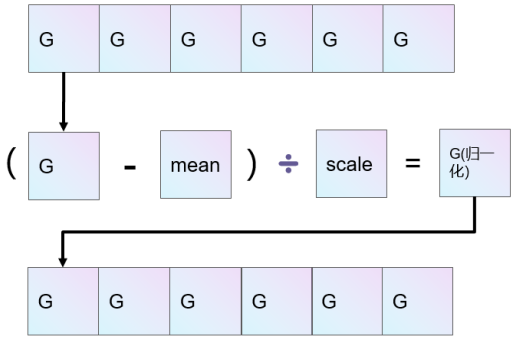


图 2.3.26

代码修改前后如下图图 2.3.27，图 2.3.28 所示：

```
// fill tensor with mean and scale and trans layout: nhwc -> nchw, neon speed up
void neon_mean_scale(const float* din,
                     float* dout,
                     int size,
                     const std::vector<float> mean,
                     const std::vector<float> scale) {
  if (mean.size() != 3 || scale.size() != 3) {
    std::cerr << "[ERROR] mean or scale size must equal to 3\n";
    exit(1);
  }
  float32x4_t vmean0 = vdupq_n_f32(mean[0]);
  float32x4_t vmean1 = vdupq_n_f32(mean[1]);
  float32x4_t vmean2 = vdupq_n_f32(mean[2]);
  float32x4_t vscale0 = vdupq_n_f32(1.f / scale[0]);
  float32x4_t vscale1 = vdupq_n_f32(1.f / scale[1]);
  float32x4_t vscale2 = vdupq_n_f32(1.f / scale[2]);
  float* dout_c0 = dout;
  float* dout_c1 = dout + size;
  float* dout_c2 = dout + size * 2;
  int i = 0;
  for (; i < size - 3; i += 4) {
    float32x4x3_t vin3 = vld3q_f32(din);
    float32x4_t vsub0 = vsubq_f32(vin3.val[0], vmean0);
    float32x4_t vsub1 = vsubq_f32(vin3.val[1], vmean1);
    float32x4_t vsub2 = vsubq_f32(vin3.val[2], vmean2);
    float32x4_t vs0 = vmulq_f32(vsub0, vscale0);
    float32x4_t vs1 = vmulq_f32(vsub1, vscale1);
    float32x4_t vs2 = vmulq_f32(vsub2, vscale2);
    vst1q_f32(dout_c0, vs0);
    vst1q_f32(dout_c1, vs1);
    vst1q_f32(dout_c2, vs2);

    din += 12;
    dout_c0 += 4;
    dout_c1 += 4;
    dout_c2 += 4;
  }

  for (; i < size; i++) {
    *(dout_c0++) = (*(din++) - mean[0]) * scale[0];
    *(dout_c0++) = (*(din++) - mean[1]) * scale[1];
    *(dout_c0++) = (*(din++) - mean[2]) * scale[2];
  }
}
```

图 2.3.27

```cpp
// fill tensor with mean and scale and trans layout: nhwc -> nchw, neon speed up
void neon_mean_scale(const float* din,
                     float* dout,
                     int size,
                     const std::vector<float> mean,
                     const std::vector<float> scale) {
  if (mean.size() != 3 || scale.size() != 3) {
    std::cerr << "[ERROR] mean or scale size must equal to 3\n";
    exit(1);
  }
  float32x4_t vmean0 = vdupq_n_f32(mean[0]);
  float32x4_t vscale0 = vdupq_n_f32(1.f / scale[0]);
  float* dout_c0 = dout;
  int i = 0;
  #pragma unroll(4)
  // #pragma omp parallel for num_threads(2)
  for (i=0; i < size - 3; i += 4) {
    float32x4_t vin3 = vld1q_f32(din);
    float32x4_t vsub0 = vsubq_f32(vin3, vmean0);
    float32x4_t vs0 = vmulq_f32(vsub0, vscale0);
    vst1q_f32(dout_c0, vs0);
    din += 4;
    dout_c0 += 4;
  }
  for (; i < size; i++) {
    *(dout_c0++) = (*(din++) - mean[0]) / scale[0];
  }
}
```

图 2.3.28

2.3.8 图片预处理过程的重写

由于上述 NHWC 代码的重构，我们将 PaddleLite 原先只支持三通道输入，重构为单通道输入，此外图片的预处理过程也应进行重构。

图片预处理函数主要完成了以下功能：

1. 将图片 Resize 为 300*300 大小，以便送入模型。
2. 将图片像素转换为 float32 格式，便于后续的归一化。
3. 进行归一化以及 NHWC 数据重排。

代码分析

由于模型训练时，我们使用双线性插值算法进行 Resize，这里我们为了保证数据一致性，也使用双线性插值算法进行 Resize。

此外由于是单通道图片，在调用 converTo 函数进行格式转换时，要修改为 CV_32FC1 格式。

```cpp
void preprocess(const cv::Mat& img, const ImageBlob img_data, float* data) {
  cv::Mat rgb_img;
  Timer timer;
  timer.startTimer();

  cv::resize(
      img, rgb_img, cv::Size(img_data.im_shape_[0],img_data.im_shape_[1]),
      0.f, 0.f, cv::INTER_CUBIC);
  std::cout << "resize time:" << timer.getCostTimer() << "ms" << "\n";

  cv::Mat imgf;
  timer.startTimer();
  rgb_img.convertTo(imgf, CV_32FC1, 1/*1 / 255.fi*/);
  const float* dimg = reinterpret_cast<const float*>(imgf.data);
  std::cout << "convert time:" << timer.getCostTimer() << "ms" << "\n";
  timer.startTimer();
  neon_mean_scale(
    dimg, data, int(img_data.im_shape_[0] * img_data.im_shape_[1]),
    img_data.mean_, img_data.scale_);
  std::cout << "neon time:" << timer.getCostTimer() << "ms" << "\n";
}
```

图 2.3.29 图像预处理代码

- **修改前后时间对比**

我们通过创建定时器，对预处理的每步进行计时，可以看到修改前后所花时间如下图所示（这里的 resize time 代表图片缩放时间；convert time 代表图片数据由 uint8 类型转换为 float32 格式所需时间；mean scale time 代表 NHWC 通道转换所需时间。）

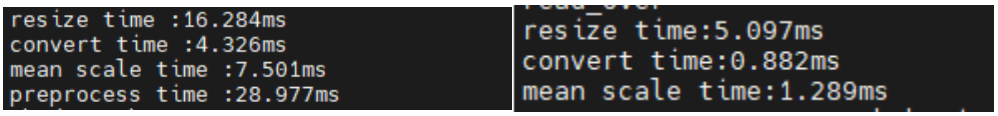


图 2.3.30 预处理修改前后对比

经过对比，我们发现我们总共进行了如下优化：

1. 将图片 Resize 时间从 16.2ms 缩短到 5ms。
2. 将图片从 uint8 格式转换为 float32 格式所化时间从 4.3ms 缩短到 0.88ms。
3. 通道 NHWC 转换从 7.5ms 降低到 1.2ms。

我们成功的将图片预处理时间从 29ms，缩短至 7ms，带来了较大的性能提升。

2.4 驱动优化

如下图 2.4.1 所示，其官方提供的驱动只能在 DDR 上申请 1024*768*2 空间，即针对 400*320*3 的图片大小 buffer，只能申请 4 个 buf 大小的连续空间，我们为了避免 ssd_hdmi 以及 ssd_detection 读写 ddr 冲突，提出了使用 6 个 buf 进行两个 IP 的设计，但在编写应用程序时发现无法申请 6 个 buf 大小的连续空间，因此我们查看驱动源码，并对其进行更改，使其能申请到更大的连续空间。

```
void *my_kernel_buffer=NULL;
unsigned long my_kernel_buffer_phy;

static int dma_size=(1024*768*2);
```

图 2.4.1

3 性能指标

3.1 推理速率指标（官方 Demo）

在使用 155Mhz 时钟频率，82%剪枝率的单通道模型，获得了 inference_time : 64ms 的推理速度。

```
------------------graph begin------
input_organize_time:10.588405ms
fpga_time:37.580349ms
output_organize_time:3.950532ms
------------------graph end------
------------------graph begin------
input_organize_time:9.738510ms
fpga_time:37.172298ms
output_organize_time:3.657476ms
------------------graph end------
predictor run:64.015ms
success detection, image size: 640, 480, detect object: zhen_kong, score: 0.996985, location: x
=236, y=157, width=23, height=25
success detection, image size: 640, 480, detect object: zhen_kong, score: 0.784001, location: x
=267, y=186, width=35, height=30
------------------ Config info ------------------
runtime_device: armv8
precision: int8/float3
num_threads: 2
---------------- Data info ----------------------
batch_size: 1
---------------- Model info ---------------------
Model_name: new1ch_3.nb
---------------- Perf info ----------------------
Total number of predicted data: 1 and total time spent(s): 77
preproce_time(ms): 12.7588, inference_time(ms): 64.1086, postprocess_time(ms): 1.05696
fpga release
root@awcloud:/opt/paddle_frame#
```

图 3.1.1

3.2 HDMI 显示刷新性能指标

经过测试，官方提供的 dvp_ddr ip，其最高帧率只能达到 15 帧，因此我们通过重写 dvp_ddr3 以及 ddr3_vga 的两个 ip，并将其整合为 dvp_ddr3_vga 模块,使其能够连续传输从而无需 HPS 控制。将 HDMI 显示刷新速率达到 48 帧帧率,如图 3.2.1 所示。

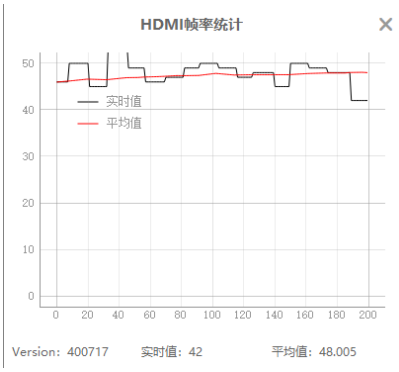


图 3.2.1

3.3 实时推理刷新性能指标（决赛所要求系统）

我们编写了 ssd_detection 程序，用于实时推理刷新，该程序仅完成推理结果

的刷新，然后将推理结果写入 PL_Plot 模块的控制寄存器中，从而完成预测框的绘制，我们推理结果的刷新速率如下图 3.3.1 所示，推理结果刷新速率为 69ms。

```
[2023-8-16 10:34:5.497]
success detection, image size: 300, 300, detect object: , score: 1, location: x=279, y=134, width=95, heig
ht=70
--------------------------------------------------------------------------------
------------------graph begin------
input_organize_time:10.023458ms
fpga_time:38.502155ms
output_organize_time:3.674703ms
------------------graph end------
[2023-8-16 10:34:5.566]
success detection, image size: 300, 300, detect object: , score: 1, location: x=276, y=135, width=95, heig
ht=69
--------------------------------------------------------------------------------
------------------graph begin------
input_organize_time:10.228271ms
fpga_time:38.563095ms
output_organize_time:3.669041ms
------------------graph end------
[2023-8-16 10:34:5.637]
success detection, image size: 300, 300, detect object: , score: 1, location: x=276, y=135, width=94, heig
ht=69
--------------------------------------------------------------------------------
```

图 3. 3. 1

3.4 数据重排优化指标

表 3.4.1

具体操作	输入数据重排时间(MS)	输出数据重排时间(MS)	总时间(MS)
原 NEON	63.416	23.491	86.82
O3 优化	35.606	12.489	48.095
删除无用置零	9.242	12.518	21.76
OPENMP	8.948	3.636	12.584
循环展开	8.932	3.567	12.499

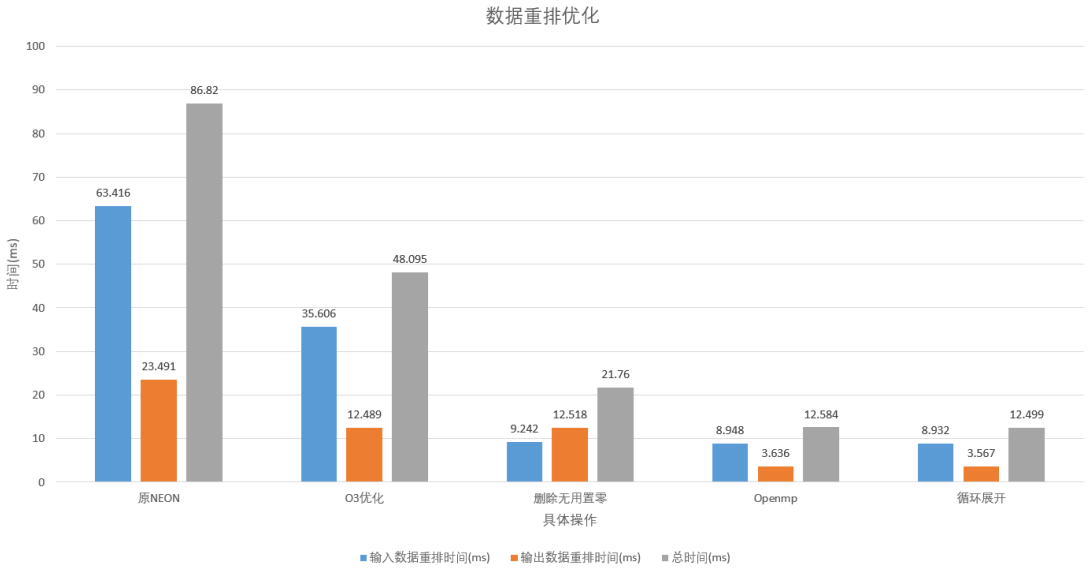


图 3.4.1

3.5 加速器优化指标

表 3.5.1

频率(MHZ)	平均推理时间(MS)	最快推理时间(MS)	加速器运行时间(MS)
50	133.382	133.104	105.849
75	99.356	99.285	72.058
100	82.695	82.554	55.170
125	73.058	72.242	44.982
150	65.785	65.206	38.414
155	64.854	64.485	37.294

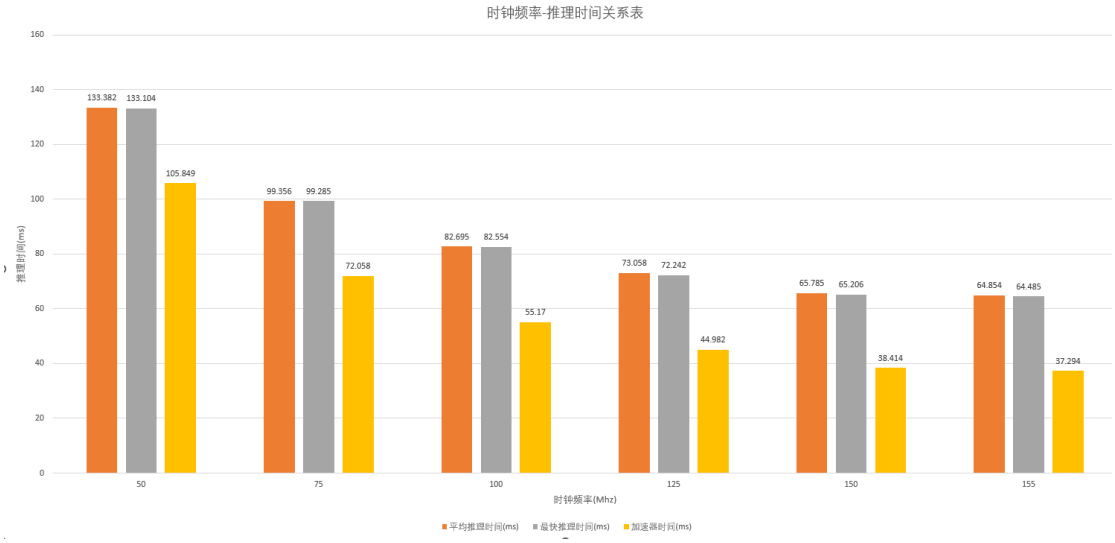


图 3.5.1

3.6 模型剪枝指标

表 3.6.1

剪枝率(%)	输出层缩放 全量化	最快推理时间(MS)	加速器运行时间(MS)
0	是	190.328	158.439
21	是	172.98	137.345
43	是	143.275	107.680
60	是	112.48	75.334
82	是	64.485	37.294

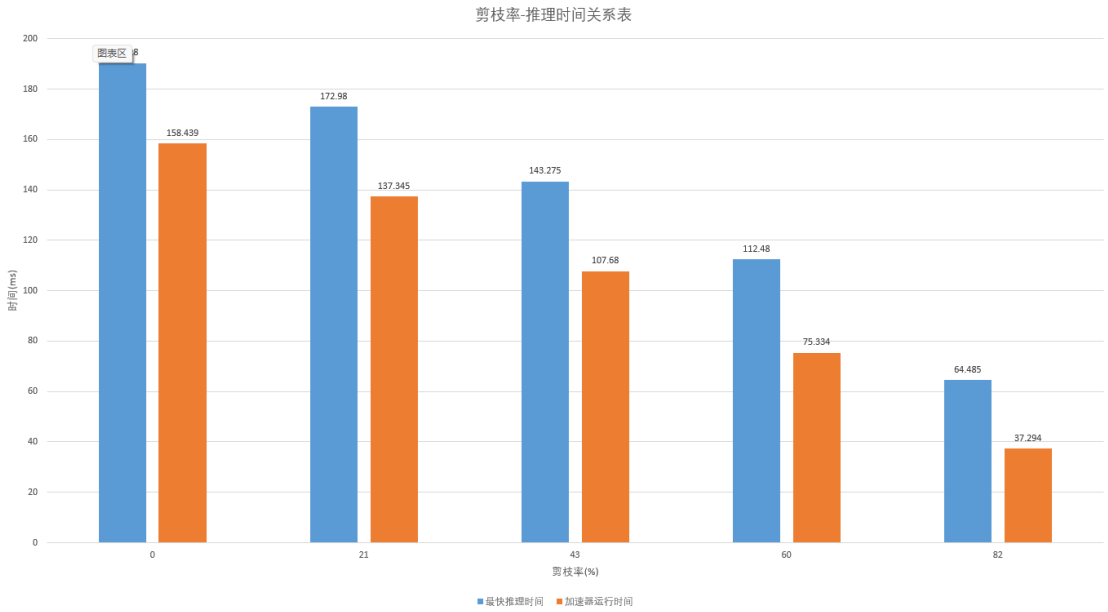


图 3.6.1

3.7 预处理优化指标（改单通道）

表 3.7.1

预处理过程	修改前时间(MS)	修改后时间(MS)
RESIZE	63.416	23.491
UINT8->FLOAT32	35.606	0.882
NHWC->NCHW	9.242	1.289
预处理总时间	28.977	7.268

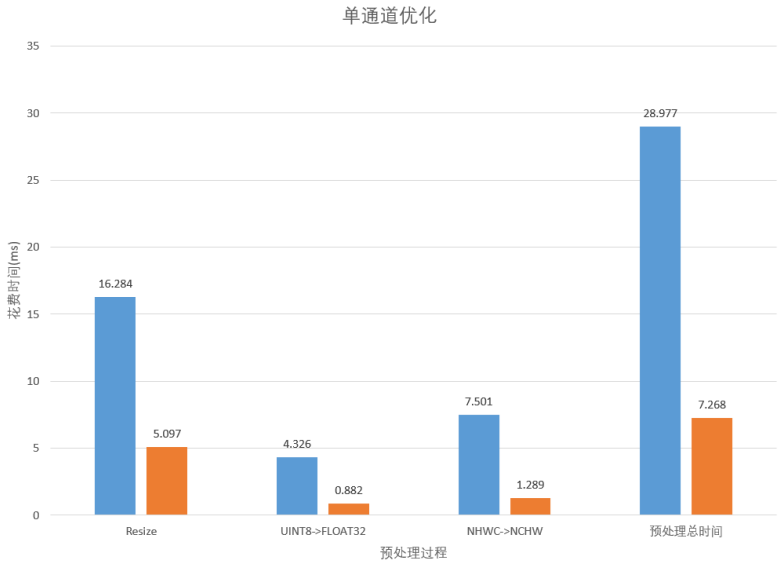


图 3.7.1