CCFSys 定制计算挑战赛

CCFSys Customized Computing Challenge 2023

技 术 报 告

学　　校： 北京工业大学

队 伍 名 称： 极光

参 赛 队 员： 刘一祺、王天硕、李一鸣

指 导 教 师： 张文博、包振山

目　　录

第 1 章　概述

1.1 问题重述

赛题：Filter2D

1、基本要求：使用 AIE API 或 AIE Intrinsic 在 64*64 图像上实现 3*3 卷积核的 filter2D 函数，完成 AIE 仿真。

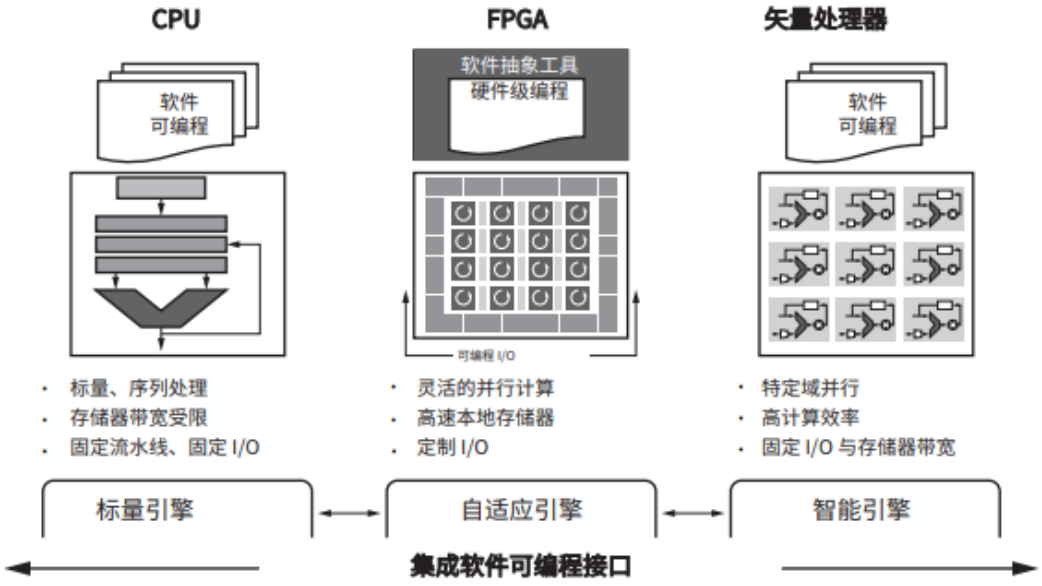
2、高级要求：在高清图像上的图像信号处理流水线，能在 VCK5000 上运行成功，输入图像尺寸越大（HD~4K），帧率越高，得分越高。

1.2 设计平台

（1）硬件平台

本次设计基于 AMD Versal 自适应加速平台(ACAP)系列的 VCK5000 板卡。Versal ACAP 系列基于 台积电（TSMC）的 7 nm FinFET 工艺技术，是第一个将软件可编程性与特定领域硬件加速和灵活应变能力相结合的平台。

Versal ACAP 系列是一款革命性的异构计算架构，其通过高带宽片上网络(NoC)有机的将下图中三种传统架构统合为一个加速引擎。这种紧密耦合的混合架构比任何一种单独架构的实现都支持更高的定制水平和性能提升[5]。



（2）软件平台

本设计使用 Vitis 工具流程来集成应用。AMD Vitis™ 工具流程可以凭借类似软件的编译和链接流程来集成 AMD Versal™ 器件的下列四大领域，从而简化硬件设计和集成：AI 引擎阵列、可编程逻辑 (PL) 区域、片上网络 (NoC) 和处

理器系统 (CIPS)。将自己已编译的 AI 引擎设计 graph (libadf.a) 与器件的 PL 区域内实现的其他内核（包括 HLS 和 RTL 内核）加以集成，并将其链接在一起以供在目标平台上使用[1]。

1.3 问题分析

为满足赛题基础要求和高级要求，必须要兼顾计算速度和内存消耗，也必须充分使用 AIE API 来加速运算。在基础要求中，图像较小，可以把所有图像都调入内存中运算,实现更高的运算效率,但在更大的图像中,因为 AIE 的内存较小，不可能把全部图像放进内存，所以就要求使用数据流方式来处理。数据流和卷积计算的设计必须合理，否则就会导致运算效率低下或者内存消耗过大的情况。在使用数据流减少内存消耗的情况下保证高运算速度是此赛题的难点所在。

1.4 解决方案

为解决上述难题，我们把图像切分为多份，每份图像都有两个 AIE kernel 来运算，输入和输出使用数据流的方式，两个核负责不同部分的运算，这样能够提高计算速率，减少内存消耗，并且处理速度可以接近一条数据流的极限输入速率,尽可能利用上全部资源,我们把能够处理一份图像的部分叫做一个处理单元。一幅图像可以有多个处理单元来并行运算,比如 64*64 图像可以分配 8 个处理单元来计算，更高分辨率的图像可以分配更多处理单元来计算，但要注意处理单元数不能过多,因为 NOC 资源有限。这种方式即提高了计算效率,又充分利用了 AIE 的硬件资源。

在搬运数据方面，使用 PL 核来搬运，PL 核负责向 AIE 的输入数据流中推送数据，同时在 AIE 内核运行时，PL 内核也负责接收输出流的数据并进行存储。

在 PS 端中，PS 端主要负责切割图像、调用 PL 内核和 AIE 内核、AIE 运算结果核验、AIE 运行时间统计等功能。

1.5 文档组织结构

在本文档中第二章主要描述了 AIE 详细设计，第三章描述了 PL 的设计以及 PL 与 AIE 的连接，第四章描述了 HOST 设计，其中包括图像分割、PL 内核的调用、数据同步以及运算结果统计，第五章主要描述整个系统的编译运行以及运行结果展示。

第 2 章　AIE 设计

2.1 设计目标

为了能在更快的运算速度、更小的内存资源消耗、更高的资源利用率的情况下满足功能要求，提出了以下设计目标：

（1）高运算速度

充分使用 AI Engine API、AI Engine Intrinsics 在硬件级别上对计算进行加速。使用并行处理和流水线，实现多个卷积计算任务的并行处理，提高整体计算效率，从而尽可能地提高卷积计算的效率和速度。通过多种优化技术实现高度并行的计算、快速的数据传输和高效的内存管理，提高计算速度。

（2）低内存消耗

整体采用流的设计，将数据流划分为多个小的数据块，每个数据块都可以在流水线中被处理，从而减少了数据在内存中的存储和传输，从而尽可能地减少内存消耗，也可以进一步提高数据的处理效率。

（3）高资源利用率

在设计过程中，要充分考虑每个核心与核心之间、核心与流之间的任务分配，不能出现一部分核心任务重，一部分核心任务较轻，一直处在阻塞的状态，这些都是资源利用率不高的表现。核心每时每刻都有运算任务，每个流都能够达到极限输入输出速率，这是最理想的情况，能够以最快的速度完成计算任务。

2.2 接口设计

AIE 中有两种类型的接口：窗口(window)和流(stream)。

流的连接方式可以将数据流化，划分为多个小的数据块，并且将这些小的数据块分别传输到处理器中进行计算。这种方式可以减少数据在内存和总线中的传输次数，同时提高了数据传输的效率和速度。在此设计中，目标是最高性能的 filter2D 实现，所以在输入和输出以及在内核与内核之间形成流接口是一种比较好的连接方式。

AI Engine 中每个数据流的速率为 1Gsps，单个数据流输入数据远远不能达到期望的速度，高采样率输入（1Gsps 以上）要被分解成多个阶段进行处理[3]。所以对于 64x64 图像，我们使用把图像分为 8 块，8 条数据流同时输入的方式来提高数据传输速率，对于 HD（1280x720）图像，由于 NOC 资源数量限制，我们选择把图像分为 12 块，12 条数据流同时输入和输出。除图像数据输入输出流之外，还有一条数据流专门用来传输卷积核。64x64 图像与 HD 图像的数据流定义如表 2-1、表 2-2 所示：

表 2-1　64x64 图像数据流定义

数据流	类型	名称	比特数
coeff_in	input_plio	CoeffData	plio_32_bits
in[0]	input_plio	StreamIn0	plio_32_bits
in[1]	input_plio	StreamIn1	plio_32_bits
in[2]	input_plio	StreamIn2	plio_32_bits
in[3]	input_plio	StreamIn3	plio_32_bits
in[4]	input_plio	StreamIn4	plio_32_bits
in[5]	input_plio	StreamIn5	plio_32_bits
in[6]	input_plio	StreamIn6	plio_32_bits
in[7]	input_plio	StreamIn7	plio_32_bits
out[0]	output_plio	StreamOut0	plio_32_bits
out[1]	output_plio	StreamOut1	plio_32_bits
out[2]	output_plio	StreamOut2	plio_32_bits
out[3]	output_plio	StreamOut3	plio_32_bits
out[4]	output_plio	StreamOut4	plio_32_bits
out[5]	output_plio	StreamOut5	plio_32_bits
out[6]	output_plio	StreamOut6	plio_32_bits
out[7]	output_plio	StreamOut7	plio_32_bits

表 2-2　HD 图像数据流定义

数据流	类型	名称	比特数
coeff_in	input_plio	CoeffData	plio_32_bits
in[0]	input_plio	StreamIn0	plio_32_bits
in[1]	input_plio	StreamIn1	plio_32_bits
in[2]	input_plio	StreamIn2	plio_32_bits
in[3]	input_plio	StreamIn3	plio_32_bits
in[4]	input_plio	StreamIn4	plio_32_bits
in[5]	input_plio	StreamIn5	plio_32_bits
in[6]	input_plio	StreamIn6	plio_32_bits
in[7]	input_plio	StreamIn7	plio_32_bits
in[8]	input_plio	StreamIn8	plio_32_bits
in[9]	input_plio	StreamIn9	plio_32_bits
in[10]	input_plio	StreamIn10	plio_32_bits
in[11]	input_plio	StreamIn11	plio_32_bits

表 2-2 （续）

数据流	类型	名称	比特数
out[0]	output_plio	StreamOut0	plio_32_bits
out[1]	output_plio	StreamOut1	plio_32_bits
out[2]	output_plio	StreamOut2	plio_32_bits
out[3]	output_plio	StreamOut3	plio_32_bits
out[4]	output_plio	StreamOut4	plio_32_bits
out[5]	output_plio	StreamOut5	plio_32_bits
out[6]	output_plio	StreamOut6	plio_32_bits
out[7]	output_plio	StreamOut7	plio_32_bits
out[8]	output_plio	StreamOut8	plio_32_bits
out[9]	output_plio	StreamOut9	plio_32_bits
out[10]	output_plio	StreamOut10	plio_32_bits
out[11]	output_plio	StreamOut11	plio_32_bits

2.3 数据类型选择

在设计时选择 int32 作为运算和存储的数据类型。int32 在 AIE 的矢量运算器中最高支持 8 个 int32 同时计算，这也就确定了一个 AIE 内核的运算并行度为八。

在计算时，使用 AI Engine Intrinsics 中的两个内部函数来计算，分别为 lmul8 和 lmac8，实现并行度为 8 的计算[2]。两个函数如下图 2-1、图 2-2 所示：

v8acc80 **lmul8** (v32int32 xbuff, int **xstart**, unsigned int xoffsets, v8int32 zbuff, int zstart, unsigned int zoffsets)
Multiply intrinsic function . More...

图 2-1 lmul8 函数

v8acc80 **lmac8** (v8acc80 acc, v32int32 xbuff, int **xstart**, unsigned int xoffsets, v8int32 zbuff, int zstart, unsigned int zoffsets)
Multiply-accumulate intrinsic function . More...

图 2-2 lmac8 函数

在 input_stream_int32 和 output_stream_int32 两种流接口中，一次性最大读取和写入量为 4 个 int32，即 readincr_v4 和 writeincr_v4，又因为 AIE 支持 8 个 int32 并行运算，所以在数据读入和写出时，需要分两次从流中读取或写入到流中，这里使用 AI Engine Intrinsics 中的 upd_v 函数实现 v4int32 到 v8int32 的合并，使用 ext_v 函数实现 v8int32 到 v4int32 的分割[2]。两个函数如下图 2-3、图 2-4 所示：

v4int32 **ext_v** (v8int32 buf, int idx)
Extract a v4int32 vector from a v8int32 vector. More...

图 2-3 ext_v 函数

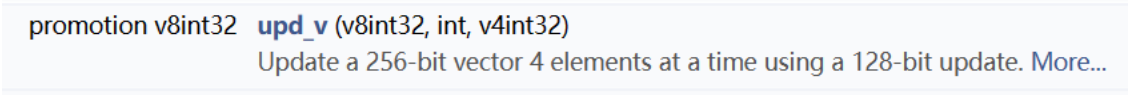


图 2-4 upd_v 函数

在 acc 流中，input_stream_acc80 和 output_stream_acc80 也是最高支持 4 个 acc80 同时写入和读取，所以这里使用 ext_hi 和 ext_lo 函数实现 v8acc80 到 v4acc80 的分割，使用 upd_hi 和 upd_lo 函数实现 v4acc80 到 v8acc80 的合并[2]。upd_lo 与 upd_hi 函数如下图 2-5 所示，ext_hi 与 ext_lo 函数如下图 2-6 所示：

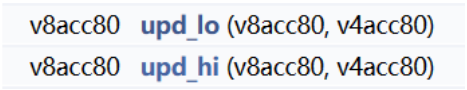


图 2-5 upd_lo 与 upd_hi 函数

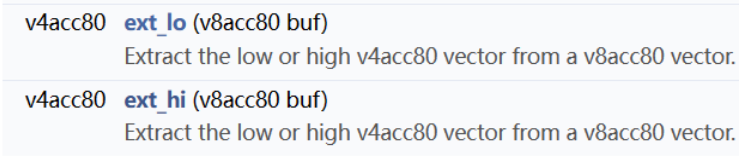


图 2-6 ext_hi 与 ext_lo 函数

2.4 AIE 内核设计

为了不受数据流速率的限制，我们把图像分为多个块。每块图像都由两个 AIE 内核来处理，第一个内核为 calc_step1 主要负责卷积 0，1，2 部分的计算，同时给第二个核整理并发送数据，第二个内核为 calc_step2，主要负责卷积 3，4，5，6，7，8 部分的计算，以及把结果写入到输出流。这里把拥有两个内核、一条输入流、一条输出流，能够处理一块图像的 Graph 称作一个处理单元[3]。处理单元如下图 2-7 所示：

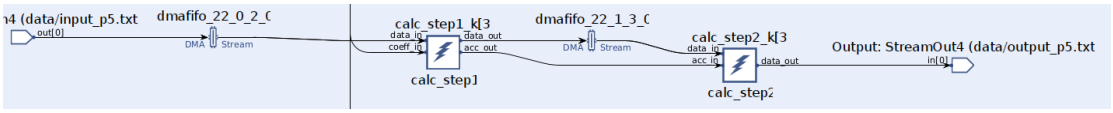


图 2-7 处理单元

因为每块图像都需要一个处理单元，也就是每块图像都需要两个内核，所以需要的总内核数为处理单元数*2，例如 64*64 图像分割 8 份并行计算，则需要消耗 16 个 AIE 内核来完成计算。

2.4.1 图像分割

为适应多个流并行输入数据，需要把图像做分割处理。这里定义图像行数为

HEIGHT，列数为 WIDTH，分的份数为 PART，每份行数为 PART_LINE_NUMBER。

对于 64*64 图像：

HEIGHT = 64;

WIDTH = 64;

PART = 8;

PART_LINE_NUMBER = HEIGHT/PART = 8;

在图像分块时，需要分两种情况，一种是第一块和最后一块，另一种是中间块，因为一行的卷积运算需要用到上中下三行的数据，并且不考虑图像边界，卷积后会缩小，所以第一块和最后一块的行数为 PART_LINE_NUMBER+1，中间块的行数为 PART_LINE_NUMBER+2。第一种情况的卷积输出行数为 PART_LINE_NUMBER-1 行，第二种情况的卷积输出为 PART_LINE_NUMBER 行，最终输出的图像大小为（HEIGHT-2）*（WIDTH-2）。

对于 HD 图像（1280*720）：

HEIGHT = 720;

WIDTH = 1280;

PART = 12;

PART_LINE_NUMBER = HEIGHT/PART = 60;

对于高分辨率图像可以按照实际情况设置分多少份，HEIGHT 必须是 PART 的倍数，否则无法运算，在这里分为 12 份，即 12 条流并行输入，也可以分更多份，份数越多，计算速度越快，但同时要注意其他资源是否足够支持这么多流的并行输入，需要做出权衡。具体分割方法如下图 2-8 所示：

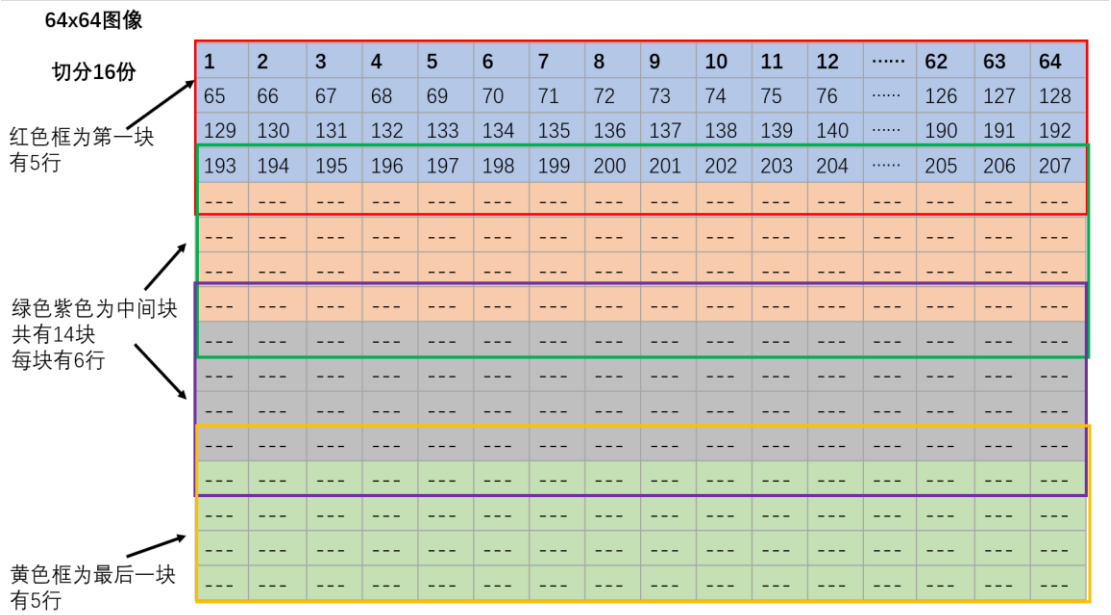


图 2-8 图像分割方法

2.4.2 calc_step1 内核设计

（1）职责

计算卷积的 kernel 0 ,1 ,2 部分,传输 calc_step2 需要的图像数据和 acc。

（2）数据流接口

图像输入流 input_stream_int32 data_in，卷积核输入流 input_stream_int32 coeff_in，图像输出流 output_stream_int32 data_out，ACC 输出流 output_stream_acc80 acc_out。

（3）处理流程

首先读入卷积核，并通过 data_out 输出流发送给 calc_step2。其次，读取核存储第一行、第二行的图像数据，通过 data_put 流输出第二行图像给 calc_step2。最后，循环 PART_LINE_NUMBER 次，每次循环中先运算出上上行的 acc，再读取下一行的新数据并更新缓存，向 data_out 流写入新数据，并向 acc_out 流写入运算出的 acc。calc_step1 的程序流程图如下图 2-9 所示：

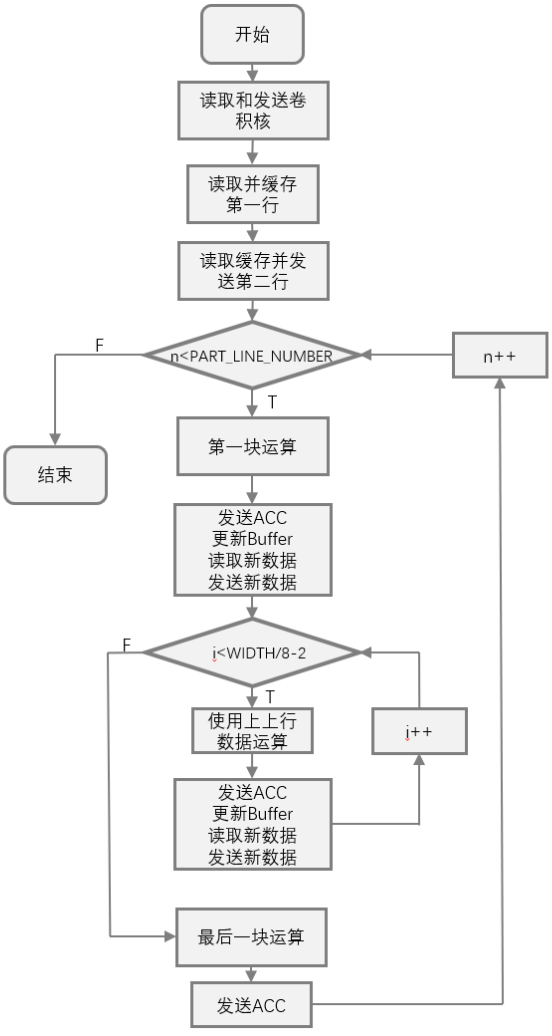


图 2-9 calc_step1 程序流程图

（4）八并行度运算详细设计

若要做卷积运算，则需要 3*10 的图像数据与 3*3 的卷积核数据，计算 8 个卷积需要的数据如图 2-10 中蓝色框中数据所示：

1	2	3	4	5	6	7	8	9	10	11	12
65	66	67	68	69	70	71	72	73	74	75	76
129	130	131	132	133	134	135	136	137	138	139	140

图 2-10 卷积需要的数据

而 calc_step1 负责的卷积数据只需要第一行数据，即图 2-10 中黄色框中的数据，图 2-11 中蓝色框中数据与 kernel[0]相乘，红色框中数据与 kernel[1]相乘，绿色框中数据与 kernel[2]相乘,并将结果相加，即达到可同时运算 8 个数据的效果，所需做的运算如下图 2-11 所示：

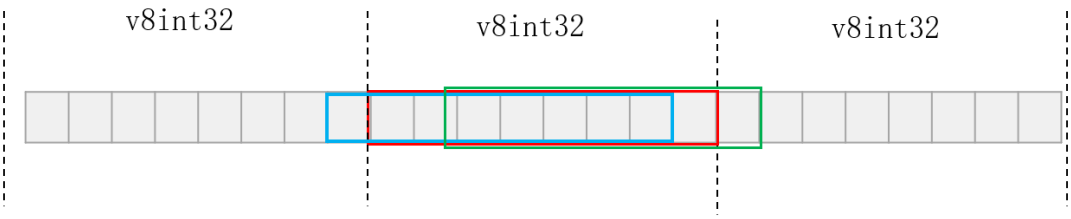


图 2-11 第一行卷积运算

从图中可看出 calc_step1 中卷积计算公式如下：

式中 acc——输出的结果

buf——存储图像的缓冲区

kernel——卷积核

$$\begin{cases} acc[0] = buf[7] * kernel[0] + buf[8] * kernel[1] + buf[9] * kernel[2] \\ acc[1] = buf[8] * kernel[0] + buf[9] * kernel[1] + buf[10] * kernel[2] \\ acc[2] = buf[9] * kernel[0] + buf[10] * kernel[1] + buf[11] * kernel[2] \\ acc[3] = buf[10] * kernel[0] + buf[11] * kernel[1] + buf[12] * kernel[2] \\ acc[4] = buf[11] * kernel[0] + buf[12] * kernel[1] + buf[13] * kernel[2] \\ acc[5] = buf[12] * kernel[0] + buf[13] * kernel[1] + buf[14] * kernel[2] \\ acc[6] = buf[13] * kernel[0] + buf[14] * kernel[1] + buf[15] * kernel[2] \\ acc[7] = buf[14] * kernel[0] + buf[15] * kernel[1] + buf[16] * kernel[2] \end{cases}$$

在运算过程中，数据是以 v8int32 为单位存储的，所以在计算时需要三块 v8int32 来完成计算。使用一个 v32int32 作为缓冲区，把三块 v8int32 放进缓冲区后，使用 luml8 核 lmac8 进行计算，在这次计算完毕后，下次计算还需要这次计算的后两个 v8int32，所以只需要把缓冲区以向前移一位，再新读取一块 v8int32 放进来，即可继续做下一次卷积运算，缓冲区在运算时的更新情况如下图 2-12 所示：

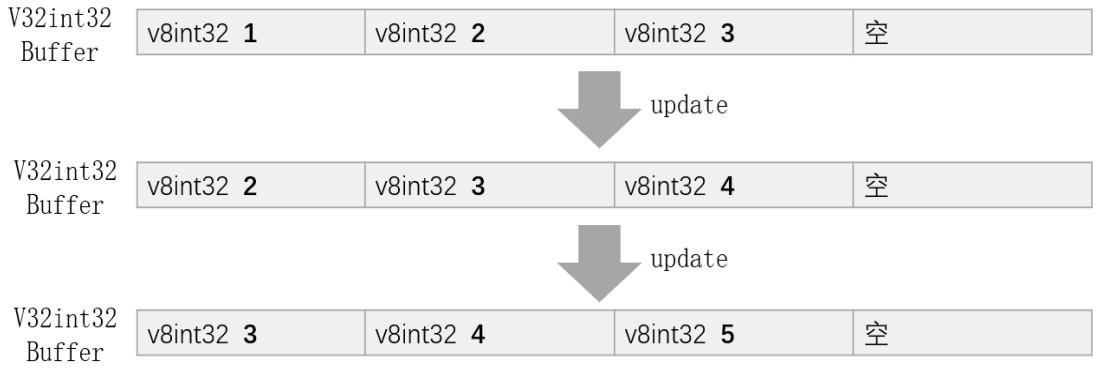


图 2-12 缓冲区更新

除了每一行的第一个以及最后一个 v8int32 需要单独处理，中间部分的所有卷积结果都可以由上述运算过程得出。

（5）为 calc_step2 整理和输出数据

calc_step1 需要[0, PART_LINE_NUMBER-1]范围的行做运算，calc_step2 需要[1, PART_LINE_NUMBER+1]范围的行做运算。所以 calc_step1 需要发送 PART_LINE_NUMBER 行的 ACC，以及 PART_LINE_NUMBER+1 行的图像数据给 calc_step2。而 calc_step2 需要的 ACC 和图像数据是需要错开两行的，所以在 calc_step1 中需要缓存两行数据。在主循环种使用上上行的数据进行运算，再把新读取进来的数据发送，这就做到了 ACC 和图像数据错开两行输出的效果。在 calc_step2 中只需按顺序输入，不需要做其他数据整理。两个内核间的数据交互如图 2-13 所示：

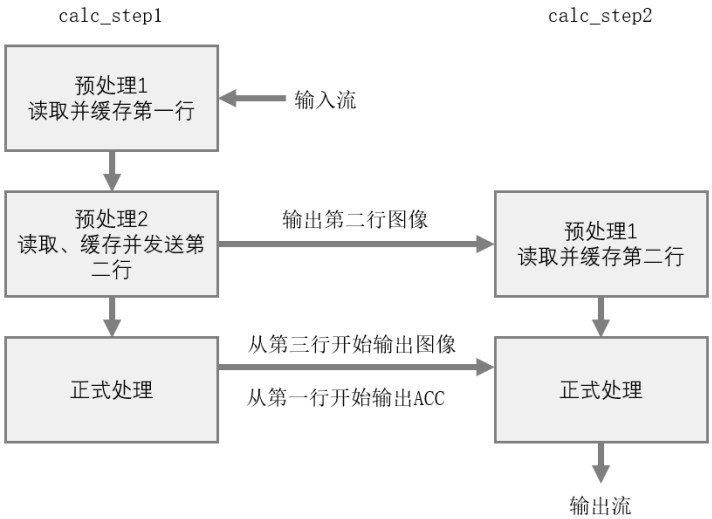


图 2-13 内核数据交互

2.4.3 calc_step2 内核设计

（1）职责

计算卷积的 kernel 3,4,5,6,7,8 部分，把结果写入输出流。

（2）数据流接口

图像输入流 input_stream_int32 data_in，ACC 输入流 input_stream_acc80 acc_in，结果输出流 output_stream_int32 data_out。

（3）处理流程

读取 calc_step1 第二个预处理阶段的图像数据并存储。循环 PART_LINE_NUMBER 次，直接按顺序读取来自 calc_step1 输出的图像和 acc 进行计算和输出。calc_step2 的程序流程图如下图 2-14 所示：

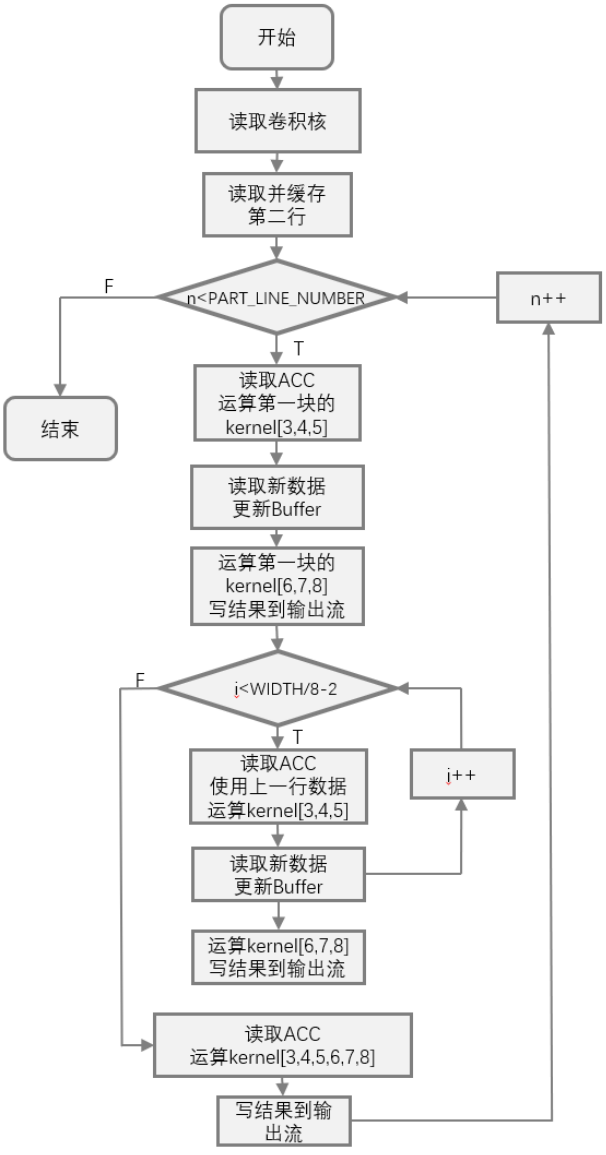


图 2-14 calc_step2 程序流程图

（4）八并行度运算详细设计

calc_step2 的计算与 calc_step1 的计算大体相同，只是需要多计算一行的数据，计算量是 calc_step1 的两倍。图中蓝色框中数据与 kernel[3]相乘，红色框中数据与 kernel[4]相乘，绿色框中数据与 kernel[5]相乘，紫色框中数据与 kernel[6]相乘，黄色框中数据与 kernel[7]相乘，橙色框中数据与 kernel[8]相乘并将结果相加，即达到可同时运算 8 个数据的效果，所需做的运算如下图 2-15 所示：

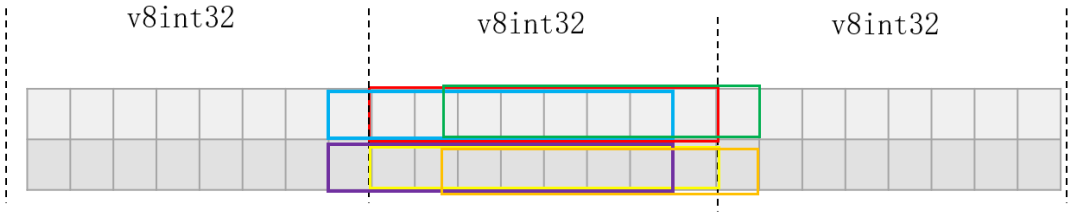


图 2-15 第 2、3 行卷积运算

从图中可看出 calc_step2 中卷积计算公式如下：

式中 acc1——输出的结果

acc——传入的结果

buf——存储图像的缓冲区

kernel——卷积核

$$
\begin{cases}
acc1[0] = acc[0] + buf[1][7] * kernel[3] + buf[1][8] * kernel[4] + buf[1][9] * kernel[5] + buf[0][7] * kernel[6] + buf[0][8] * kernel[7] + buf[0][9] * kernel[8] \\
acc1[1] = acc[1] + buf[1][8] * kernel[3] + buf[1][9] * kernel[4] + buf[1][10] * kernel[5] + buf[0][8] * kernel[6] + buf[0][9] * kernel[7] + buf[0][10] * kernel[8] \\
acc1[2] = acc[2] + buf[1][9] * kernel[3] + buf[1][10] * kernel[4] + buf[1][11] * kernel[5] + buf[0][9] * kernel[6] + buf[0][10] * kernel[7] + buf[0][11] * kernel[8] \\
acc1[3] = acc[3] + buf[1][10] * kernel[3] + buf[1][11] * kernel[4] + buf[1][12] * kernel[5] + buf[0][10] * kernel[6] + buf[0][11] * kernel[7] + buf[0][12] * kernel[8] \\
acc1[4] = acc[4] + buf[1][11] * kernel[3] + buf[1][12] * kernel[4] + buf[1][13] * kernel[5] + buf[0][11] * kernel[6] + buf[0][12] * kernel[7] + buf[0][13] * kernel[8] \\
acc1[5] = acc[5] + buf[1][12] * kernel[3] + buf[1][13] * kernel[4] + buf[1][14] * kernel[5] + buf[0][12] * kernel[6] + buf[0][13] * kernel[7] + buf[0][14] * kernel[8] \\
acc1[6] = acc[6] + buf[1][13] * kernel[3] + buf[1][14] * kernel[4] + buf[1][15] * kernel[5] + buf[0][13] * kernel[6] + buf[0][14] * kernel[7] + buf[0][15] * kernel[8] \\
acc1[7] = acc[7] + buf[1][14] * kernel[3] + buf[1][15] * kernel[4] + buf[1][16] * kernel[5] + buf[0][14] * kernel[6] + buf[0][15] * kernel[7] + buf[0][16] * kernel[8]
\end{cases}
$$

（5）输出计算结果

在计算完毕后，把 acc 使用 srs 函数转化为 v8int32 后即可输出。

在一行结果中，结果的个数应该比图像的列数少二，所以运算结果中的第一个和最后一个 int32 需要丢弃，最后输出图像的尺寸为(HEIGHT-2)*(WIDTH-2)。

因为需要丢弃单个数据，所以在向输出流写入一行的第一块 v8int32 和最后

一块 v8int32 时需要使用 ext_elem 函数来提取，在写入中间块时使用 ext_v 函数把 v8int32 分成两份，依次写入输出流即可。

2.5 仿真验证

2.5.1 性能测试

在内核设计完成后，对于 64x64 图像拆分为 8 路、HD 图像拆分为 12 路的情况进行了 AIE 仿真对其性能进行了评估。8 路的 graph 和 12 路的 graph 如图 2-16、图 2-17 所示：

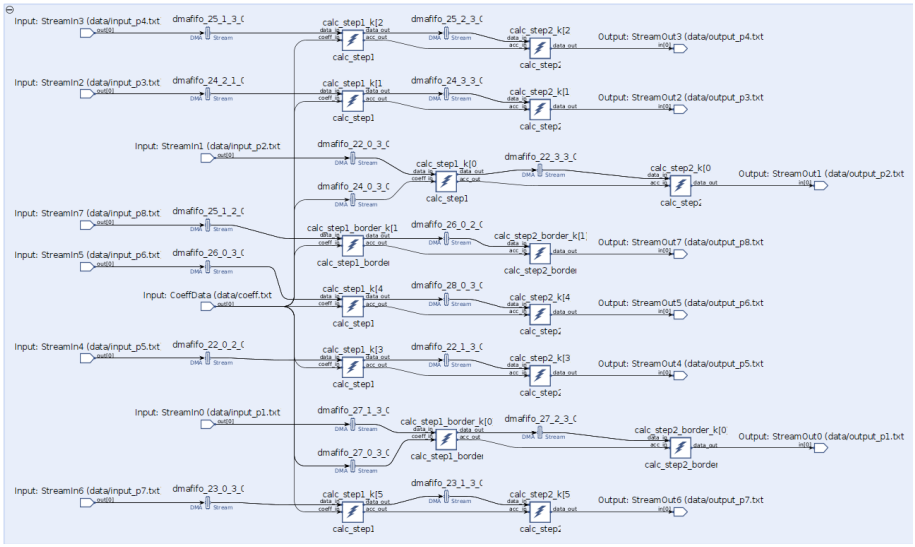


图 2-16 64x64 图像 graph

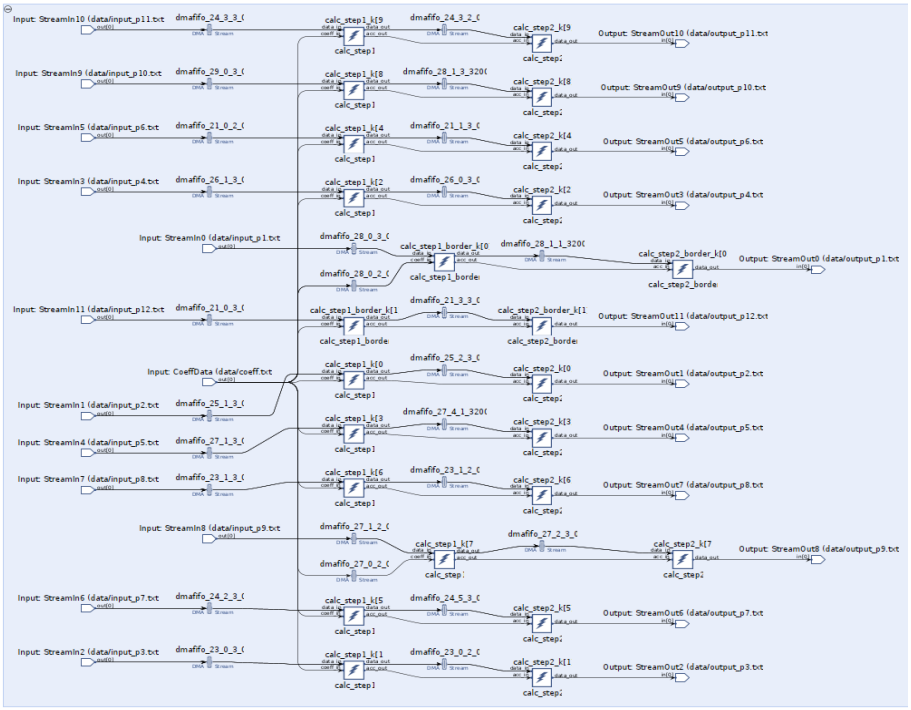


图 2-17 HD 图像 graph

下文中只对于 8 路 64x64 图像进行描述，因为 12 路的 HD 图像过大导致 AIE 仿真时间过长，所以未对其进行 AIE 仿真，12 路 HD 图像性能指标将在第五章系统部署部分展示在 VCK5000 板卡上的实际运行速度。

在 AIE 仿真完毕后，分别查看 calc_step1 和 calc_step2 的 Trace，观察运行时间和流阻塞情况[8]，两个核的 Trace 如图 2-18 所示：

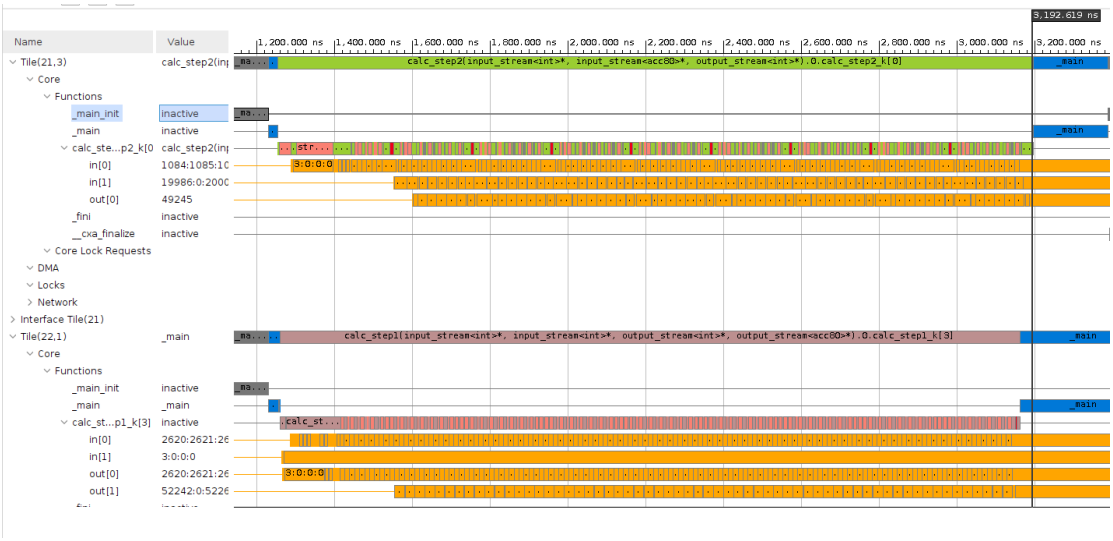


图 2-18 Trace 图

在图中可以看出，calc_step1 和 calc_step2 的运行时间基本相同，任务分配较为平均。在 calc_step1 中 stream stall 出现很频繁，因为每个单元的处理速率已经超过流的极限输入速率，流的利用率很高，在数据流准备数据的时间里已经把上次读取进来的数据处理完毕。

所有 AIE kernel 的运行情况如下图 2-19 所示：

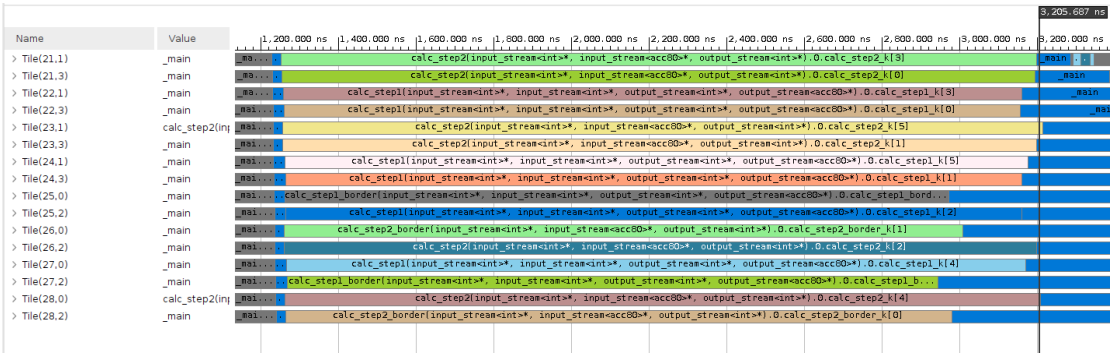


图 2-19 AIE kernel 运行情况

在内存消耗方面：通过 vitis analyzer 中可以得出 64*64 图像每个核心只消耗 4KB 内存，HD 图像每个核心只消耗 11KB 内存，符合设计要求。

在速度方面：图 2-19 中可以计算出整个系统计算完毕一幅 64*64 图像需要消耗 1951ns，同时可以计算出整个系统的处理帧率为 512557.66FPS。但因为每块图像行数较少，流水线启动需要时间，满速运行的时间不是很长，如果在每块

图像行数足够的情况下，整个系统的吞吐率能更高。经过 AIE 仿真验证，此设计符合预期目标。

2.5.2 运算结果验证

AIE 仿真结束后会对应生成多个 output 文件，这里使用一个 C 语言外部程序实现输出卷积标准运算结果，与 AIE 运行结果进行比对。生成标准结果的关键代码如下：

```
for(int i=0;i<HEIGHT-2;i++){
  for(int j=0;j<WIDTH-2;j++){
    int tmp = 0;
    for(int m=0;m<3;m++){
      for(int n=0;n<3;n++){
        tmp+=IMG[i+m][j+n] * kernel[m][n];
      }
    }
  }
}
```

把标准输出结果拷贝到 Vitis 的 AIE 工程中，使用 Vitis 自带的 Compare With 功能进行比对结果，在弹出的 enter transformation commands 窗口中输入命令去除 AIE 仿真输出的时间戳[8]。去除时间戳命令如下图 2-20 所示：

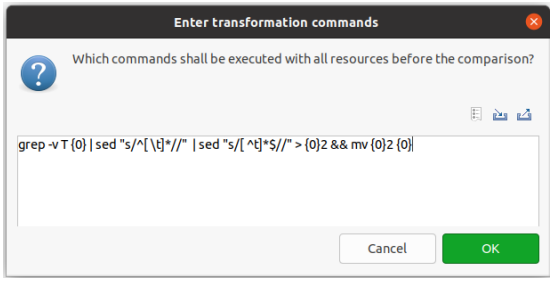


图 2-20 去除时间戳命令

比较后对比结果如下图 2-21 所示：

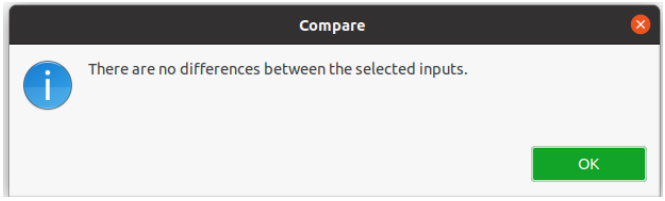


图 2-21 比对结果

在对 16 组数据进行逐一比对后，结果显示所有输出数据与标准输出数据相符，代表 AIE 输出结果正确。

第 3 章　PL 内核设计

3.1 功能描述

实现将块状的内存与 AXI-stream 的流式数据互相转换。采用两种 PL 核，一种是内存到流的转换(pl_mm2s)，一种是流到内存的转换(pl_s2mm)，分别用于 AIE 数据输入和 AIE 数据输出。

3.2.1 PL 内核接口设计

pl_mm2s 需要的参数为：

ap_int<32>* mem：输入数据的内存空间

hls::stream<qdma_axis<32, 0, 0, 0> >& s：AXI-stream 流

int size：要处理的数据量

pl_s2mm 需要的参数为：

ap_int<32>* mem:要存储输出数据的内存空间

hls::stream<qdma_axis<32, 0, 0, 0> >& s：AXI-stream 流

int size：要处理的数据量

3.2.2 PL 内核设计

在 64x64 图像中，pl_mm2s 和 pl_s2mm 各有 8 个，分别负责 8 个输入数据流和 8 个输出数据流，在 HD 图像中则各有 12 个，除此之外还另需要一个 pl_mm2s 负责卷积核的传递[4]。其中 pl_mm2s 和 pl_s2mm 的核心代码如图 3-1、3-2 所示：

```
void pl_mm2s(ap_int<32>* mem, hls::stream<qdma_axis<32, 0, 0, 0> >& s, int size) {
data_mover:
    for (int i = 0; i < size; i++) {
        qdma_axis<32, 0, 0, 0> x;
        x.data = mem[i];
        x.keep_all();
        s.write(x);
    }
}
}
```

图 3-1 pl_mm2s 代码

```
void pl_s2mm(ap_int<32>* mem, hls::stream<qdma_axis<32, 0, 0, 0> >& s, int size) {
data_mover:
    for (int i = 0; i < size; i++) {
        qdma_axis<32, 0, 0, 0> x = s.read();
        mem[i] = x.data;
    }
}
}
```

图 3-2 pl_s2mm 代码

3.3 PL 与 AIE 连接

对于 PL 和 AIE 的流连接，我们使用 config.cfg 文件来指定 PL 与 AIE 之间的连接。

64x64 图像的 config.cfg 如下图 3-3 所示：

```
[connectivity]
#Number of Kernels
nk=pl_mm2s:9
nk=pl_s2mm:8

#Input Image to AIE Graph
stream_connect=pl_mm2s_1.s:ai_engine_0.StreamIn0
stream_connect=pl_mm2s_2.s:ai_engine_0.StreamIn1
stream_connect=pl_mm2s_3.s:ai_engine_0.StreamIn2
stream_connect=pl_mm2s_4.s:ai_engine_0.StreamIn3
stream_connect=pl_mm2s_5.s:ai_engine_0.StreamIn4
stream_connect=pl_mm2s_6.s:ai_engine_0.StreamIn5
stream_connect=pl_mm2s_7.s:ai_engine_0.StreamIn6
stream_connect=pl_mm2s_8.s:ai_engine_0.StreamIn7

#Input filter2D kernel to AIE Graph
stream_connect=pl_mm2s_9.s:ai_engine_0.CoeffData

#Access Graph Output
stream_connect=ai_engine_0.StreamOut0:pl_s2mm_1.s
stream_connect=ai_engine_0.StreamOut1:pl_s2mm_2.s
stream_connect=ai_engine_0.StreamOut2:pl_s2mm_3.s
stream_connect=ai_engine_0.StreamOut3:pl_s2mm_4.s
stream_connect=ai_engine_0.StreamOut4:pl_s2mm_5.s
stream_connect=ai_engine_0.StreamOut5:pl_s2mm_6.s
stream_connect=ai_engine_0.StreamOut6:pl_s2mm_7.s
stream_connect=ai_engine_0.StreamOut7:pl_s2mm_8.s
```

图 3-3 64x64 图像 config.cfg

HD 图像的 config.cfg 如下图 3-4 所示：

```
[connectivity]
#Number of Kernels
nk=pl_mm2s:13
nk=pl_s2mm:12

#Input Image to AIE Graph
stream_connect=pl_mm2s_1.s:ai_engine_0.StreamIn0
stream_connect=pl_mm2s_2.s:ai_engine_0.StreamIn1
stream_connect=pl_mm2s_3.s:ai_engine_0.StreamIn2
stream_connect=pl_mm2s_4.s:ai_engine_0.StreamIn3
stream_connect=pl_mm2s_5.s:ai_engine_0.StreamIn4
stream_connect=pl_mm2s_6.s:ai_engine_0.StreamIn5
stream_connect=pl_mm2s_7.s:ai_engine_0.StreamIn6
stream_connect=pl_mm2s_8.s:ai_engine_0.StreamIn7
stream_connect=pl_mm2s_9.s:ai_engine_0.StreamIn8
stream_connect=pl_mm2s_10.s:ai_engine_0.StreamIn9
stream_connect=pl_mm2s_11.s:ai_engine_0.StreamIn10
stream_connect=pl_mm2s_12.s:ai_engine_0.StreamIn11

#Input filter2D kernel to AIE Graph
stream_connect=pl_mm2s_13.s:ai_engine_0.CoeffData

#Access Graph Output
stream_connect=ai_engine_0.StreamOut0:pl_s2mm_1.s
stream_connect=ai_engine_0.StreamOut1:pl_s2mm_2.s
stream_connect=ai_engine_0.StreamOut2:pl_s2mm_3.s
stream_connect=ai_engine_0.StreamOut3:pl_s2mm_4.s
stream_connect=ai_engine_0.StreamOut4:pl_s2mm_5.s
stream_connect=ai_engine_0.StreamOut5:pl_s2mm_6.s
stream_connect=ai_engine_0.StreamOut6:pl_s2mm_7.s
stream_connect=ai_engine_0.StreamOut7:pl_s2mm_8.s
stream_connect=ai_engine_0.StreamOut8:pl_s2mm_9.s
stream_connect=ai_engine_0.StreamOut9:pl_s2mm_10.s
stream_connect=ai_engine_0.StreamOut10:pl_s2mm_11.s
stream_connect=ai_engine_0.StreamOut11:pl_s2mm_12.s
```

图 3-4 HD 图像 config.cfg

第 4 章　HOST 设计

4.1 设计目标

Host 部分代码用于在 Cortex®-A72 上运行，负责通过 Xilinx 运行时库(XRT)管理、控制 AIE graph 和 PL 内核，故对其提出以下设计目标

（1）高运算速度：通过命令队列等，将程序尽可能的并行化

（2）内存安全：申请的内存空间在不需要时必须释放

（3）计算结果验证：PS 部分需要独立执行算法，并将结果与 AIE 部分运算结果比对，以验证结果的正确性

4.2 HOST 详细设计

HOST 程序采用 C++17 标准以及 XRT Native API[1]对整体架构进行时序管理，HOST 程序流程图如下图 4-1 所示：

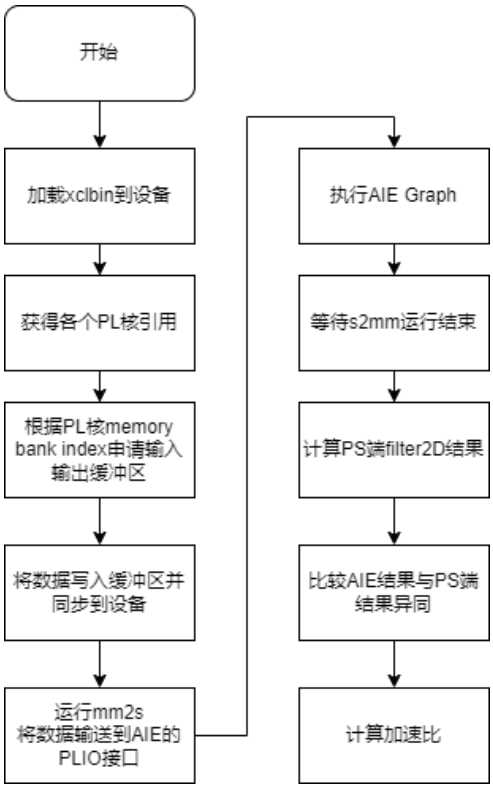


图 4-1 HOST 程序流程图

4.2.1 图像数据分割

分割图像数据是在 PS 端中实现的，具体分割方式在 AIE 设计部分已经说明，算法代码如下图所示：

```
for(int i=1;i<pl_kernel_num;++i){
    DataInput[i]=new int[inputDataLength[i]];
    memcpy(DataInput[i],img+((img_size/pl_kernel_num)*i-1)*img_size,inputDataLength[i]*sizeof(int));
    input_buff[i].write(DataInput[i]);
    input_buff[i].sync(XCL_BO_SYNC_BO_TO_DEVICE);
    std::cout<<"input_buffer_"<<i<<" finished"<<std::endl;
}
```

图 4-2 图像数据分割代码

4.2.2 PL 内核引用

通过 XRT::kernel 接口获得每个 PL 核的引用，注意必须与 hw_link 部分 cfg 文件中的名字一一对应。

4.2.3 数据缓冲区与数据同步

通过 XRT::bo 接口分配 4K 对齐的缓冲区对象。

其被用于在 host 与 device 之前进行数据传输：

当 host 要向 device 写入数据时，需要先调用 xrt::bo::write()更新 host 侧 buffer，再通过 xrt::bo::sync(XCL_BO_SYNC_BO_TO_DEVICE)将 host 侧的数据同步到 device 侧完成数据同步。

当 host 要从 device 读出数据时，需要先调用 xrt::bo::sync(XCL_BO_SYNC_BO_FROM_DEVICE)将 device 侧的数据同步到 host 侧，再通过 xrt::bo::read()将数据从 host 侧 buffer 中取出[6,7]。

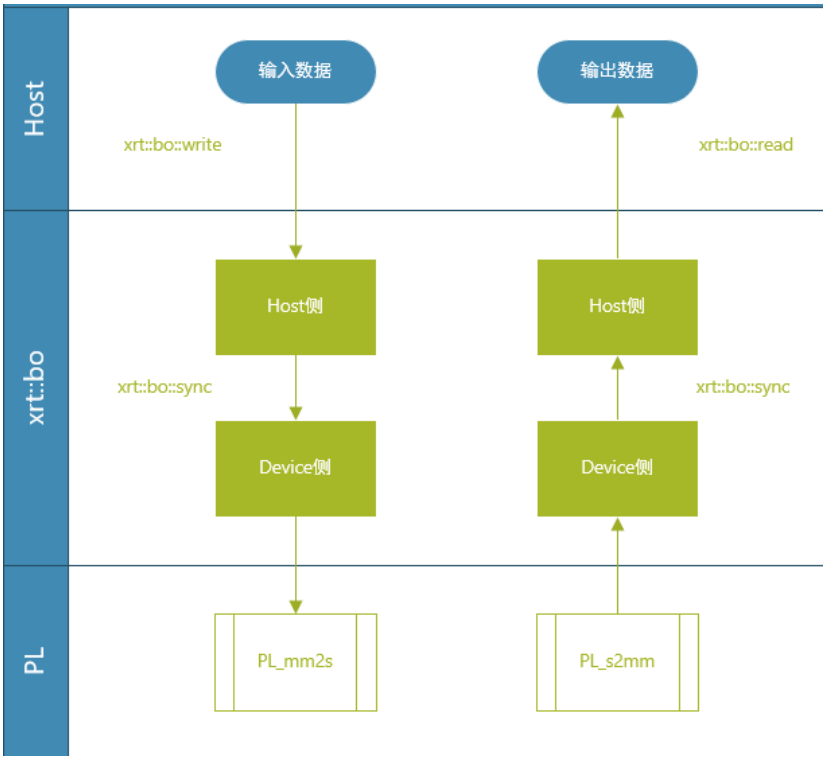


图 4-3 通过 XRT::bo 完成 host 与 device 数据传输

此外，为了保证能与 PL 核相连，在创建 XRT::bo 时需要通过 xrt::kernel::group_id()接口获得 PL 核的 memory bank index。故步骤必须在获得 PL 内核的引用后进行。

4.2.4 调用 PL 核执行与等待完成

在 XRT native API 中对 xrt::kernel 的()操作符。只需将这些 PL 核的输入填入其中即可开始当前 PL 核的运行。

xrt::kernel 的()操作符会返回一个 xrt::run 类的对象，如果需要维护时序逻辑，只需调用 xrt::run::wait()函数即可阻塞该进程直至 PL 核完成运行。

4.2.5 运算结果对比

```
//filter2D algorithm on PS
int* filter2D(int* img, unsigned int img_width, unsigned int img_length, const int* kernel, unsigned int kernel_size) {
    //compute the result size
    unsigned int result_length=img_length-kernel_size+1;
    unsigned int result_width=img_width-kernel_size+1;
    //Allocate memory for result
    int* result = new int[result_length * result_width];
    memset(result, 0, sizeof(int) * result_length * result_width);
    //compute the result
    for (unsigned int row = 0; row < result_width; ++row) {
        for (unsigned int col = 0; col < result_length; ++col) {
            unsigned int iter_kernel = 0;
            //compute the result[row][column]
            for (unsigned int iter_r = 0; iter_r < kernel_size; ++iter_r) {
                for (unsigned int iter_c = 0; iter_c < kernel_size; ++iter_c) {
                    result[row * result_length + col] += img[(row + iter_r) * img_length + col + iter_c] * kernel[iter_kernel++];
                }
            }
        }
    }
    return result;
}
```

图 4-4 PS 端 filter2D 代码

在 HOST 中是独立实现了 filter2D 算法，并在最后将该部分运算结果与 AIE 部分的运算结果进行一一比对，并统计不同结果的数量。算法代码如下图所示：

算法复杂度分析：

设图片长宽为 m,n，卷积核大小为 k。

时间复杂度：$O(k^2 \times m \times n)$

空间复杂度：考虑结果为$O(m \times n)$，不考虑存放结果的空间则为$O(1)$

4.2.6 运算时间统计

通过 C++标准 chrono 库对运行时间进行统计。

在算法开始和结束时通过 std::chrono::steady_clock::now()获得当前时间点，并将两者相减得到 std::chrono::duration 对象表示这一时间段长度。本程序中将时间度量衡选为毫秒。

第 5 章　系统部署

5.1 系统编译

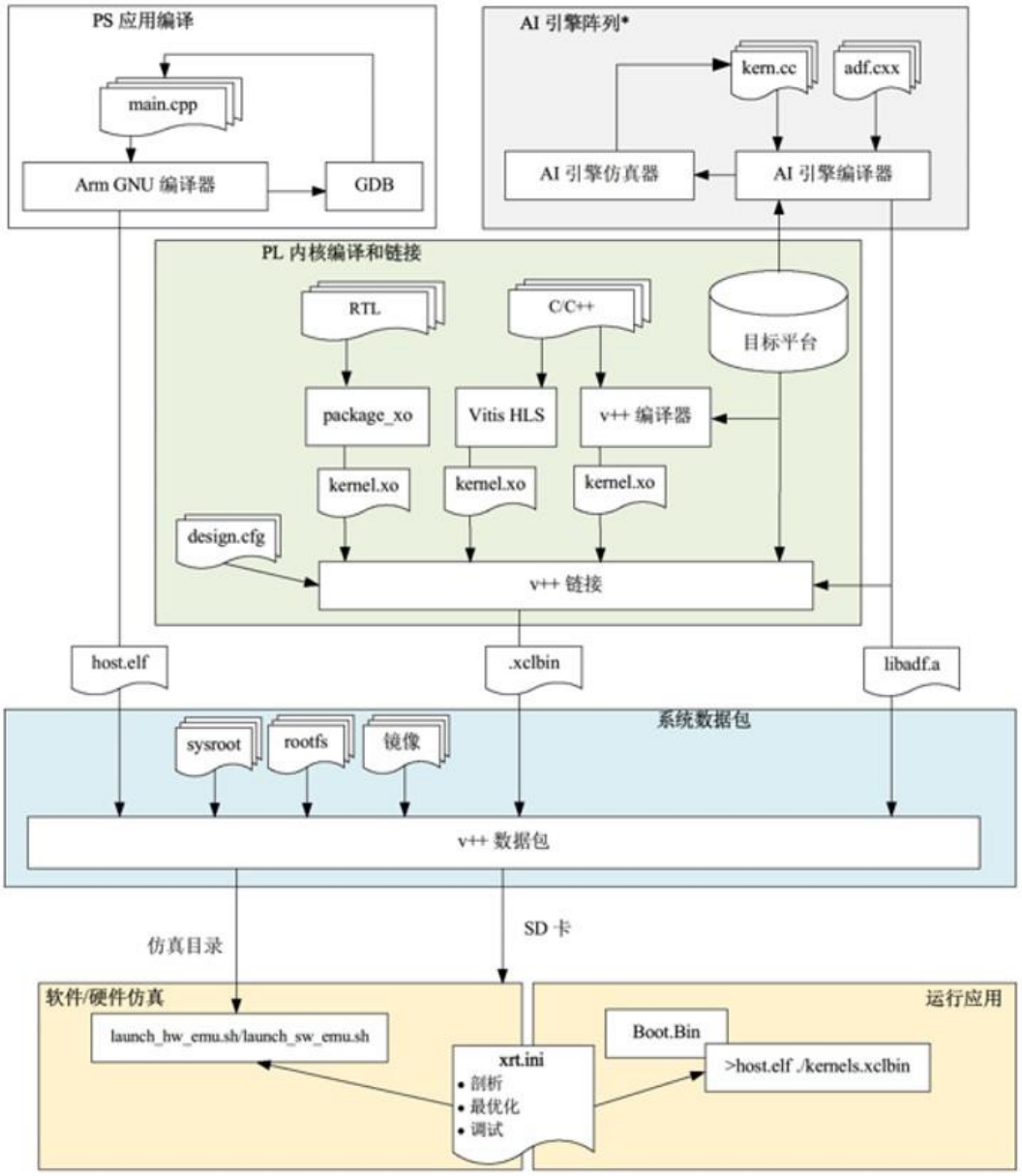


图 5-1 构建和封装嵌入式系统的流程

AIE Graph 部分经 aiecompiler 编译为 libadf.a 文件[1]

每个 PL 核均编译为.xo 文件

V++编译器将上面的所有中间件以及 hw_link 部分的.cfg 封装为器件二进制文件(.xclbin)[8]

Host 程序编译为可执行文件文件(.exe)，在运行中读取器件二进制文件，获得控制硬件设计所需的元数据。

5.2 Makefile

一个工程中通常还有大量源文件，其按类型、功能、模块分别放在若干个目录中。可以在 makefile 中定义一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译。通过 makefile 文件，开发者可以仅仅通过一行简单的 make 命令即可完成需要大量命令才能完成的项目编译。

5.3 系统运行

本次实验借助于 HACC@NUS 的板卡支持。

在 HACC@NUS 使用 Slurm 开源工作调度工具对计算机集群进行计算机节点的分配。在使用前首先应使用 sinfo 查看当前计算节点的占用情况，如下图所示，如果节点状态是 idle 表示当前节点可用，alloc 表示节点在用[9,10]。

```
[redacted]@hacchead:~$ sinfo
PARTITION                              AVAIL  TIMELIMIT  NODES  STATE NODELIST
cpu_only*                                 up 7-00:00:00      1   idle hacc-node2
vck5000_standard_reservation_pool         up      30:00      5   idle hacc-vck5000-[1-5]
vck5000_compile                           up 7-00:00:00      1   idle hacc-node0
vck5000_qdma                              up      30:00      5   idle hacc-vck5000-[1-5]
xilinx_vck5000_gen4x8_qdma_2_202220_1     up      30:00      5   idle hacc-vck5000-[1-5]
```

图 5-2 sinfo 查询计算节点占用情况

决定使用哪个节点后即可使用官方提供的 vck5000_alloc 脚本去申请当前该节点使用权。手动执行 vck5000_exit 脚本或者使用时间到达 30min 会自动对节点进行回收[9,10]。

5.4 运行结果

两部分均采用三次重复实验，计算平均加速比。

5.4.1 64x64 图像运行结果

表 5-1 64x64 图像运行结果

	AIE 运行时间(ms)	PS 运行时间(ms)	加速比
实验一	0.7090592	0.157073	0.198678
实验二	0.824778	0.160741	0.19489
实验三	0.849412	0.164867	0.194095
平均	0.7944164	0.160894	0.195888

5.4.2 HD 图像运行结果

表 5-2 HD 图像运行结果

	AIE 运行时间(ms)	PS 运行时间(ms)	加速比
实验一	1.50515	47.4129	31.5005
实验二	1.69701	43.6752	25.7365
实验三	1.56071	43.7315	28.0202
平均	1.58762	44.9399	28.4191

5.4.3 实验分析

注意到 HD 图像实验取得了平均 28.4191 的加速比，证明我们的思路是正确的。但是我们发现对于数据量较少的 64x64 实验，使用 AIE 进行运算甚至是负优化。考虑到使用 AIE 进行 HD 图像平均 1.58762ms 与 64x64 图像平均 0.7944164ms 的运算时间相差无几，以及 64x64 图像的 AIE 仿真结果为 1951ns，我们初步分析可能是因为 PL 数据搬运及进程同步等固定开销推高了使用 AIE 完成 64x64 图像处理的时间，也有可能是因为目前的测试方式对于极短的运行时间有误差。

第 6 章　总结与展望

1、比赛总结

本设计针对 filter2D 赛题，完成了此赛题的基础要求以及高级要求，使用了分割图像并用多路同时处理所有图像块的方式，在使用 VCK5000 板卡的情况下使用 HD 图像达到平均 28.4191 的加速比，取得了比较不错的效果，但使用 64x64 图像只获得了 0.195888 的加速比，而在 AIE 仿真中只耗时 1951ns，我们分析可能是因为 PL 数据搬运及进程同步等固定开销推高了运行时间，也有可能是目前的测试方式对于极短的运行时间有误差。同时，在开发过程中遇到了许多困难，数据传输逻辑和运算方式也频繁修改，从单内核到双内核，从双内核到多路同时处理，再到最后的内存优化，并且在编译和部署时也处处碰壁，但最终解决了所有问题。

2、未来工作展望

虽然完成了本赛题，但依然存在很多可以优化的地方。

（1）没有考虑图像边界，输出的图像会比原图像小。

（2）没有考虑三通道的图像处理，目前有很多图像都是有 RGB 三个通道，但此设计只能处理灰度图像。

（3）没有考虑不同数据类型，因为不同数据类型在 API 中的计算方式都有变化，数据读取逻辑也会随之变化，这就导致没有很多时间来研究其他数据类型的运算。

3、个人建议

在开发过程中遇到有一些不方便的地方，对此提出了一些建议。

（1）在文档方面：文档数量比较多，比较散，建议多突出和整理重点部分，便于查阅和开发。

（2）在编译方面：AIE 和 SW 编译和运行耗时和内存消耗情况还能接受，但在 windows 中的分配 12GB 内存的 ubuntu 虚拟机中运行 hardware 编译无法完成，内存会爆满导致崩溃，而在服务器中编译需要五六个小时，比较耗时，所以建议可以优化编译流程或方式，让大部分电脑可以承受编译所需的资源。

4、致谢

最后，感谢竞赛组织方提供了硬件平台以及线上培训让我们团队能够接触到业界最先进的异构系统架构。

参考文献

[1] AMD/Xilinx. AI Engine Tools and Flows User Guide (UG1076)

[2] AMD/Xilinx. AI Engine Intrinsics User Guide

[3] AMD/Xilinx. AI Engine Kernel and Graph Programming Guide (UG1079)

[4] AMD/Xilinx. Vivado Design Suite User Guide High-Level Synthesis (UG902)

[5] AMD/Xilinx. Versal ACAP AI Engine Architecture Manual (AM009)

[6] AMD/Xilinx. XRT Native Library C++ API

[7] AMD/Xilinx. XRT and Vitis™ Platform Overview

[8] AMD/Xilinx. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)

[9] HACC@NUS. Heterogeneous Accelerated Compute Cluster (HACC) at NUS

[10] Slurm. Slurm workload manager Version23.02