

# 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
1.1 赛题要求.....	1
1.2 开发平台介绍.....	1
1.2.1 硬件平台.....	1
1.2.2 软件平台.....	2
1.2.3 AIE 架构优势.....	2
1.3 竞题难点分析.....	3
1.4 解决方案.....	5
1.5 文档组织结构.....	6
第 2 章 系统设计.....	7
2.1 介绍.....	7
2.2 系统架构.....	7
第 3 章 AI Engine 设计.....	9
3.1 技术难点分析及设计目标.....	9
3.2 功能概述.....	10
3.3 AIE 核心设计.....	10
3.4 Graph 设计.....	12
3.5 内存与速度的权衡.....	13
3.6 性能评估.....	14
3.7 本章小结.....	15
第 4 章 PL 设计.....	16
4.1 技术难点分析及设计目标.....	16
4.2 功能描述.....	16
4.3 核心设计.....	17
4.3.1 PL_SEND 核心设计.....	17
4.3.2 PL_RECEIVE 核心设计.....	19
4.4 存在的不足.....	20
4.5 本章小结.....	21
第 5 章 PS 设计.....	22
5.1 设计目标.....	22
5.2 功能概述.....	22
5.3 JPEG 图像解码.....	23
5.4 图像数据分割.....	24
5.5 运算结果重塑与合并.....	24
5.6 信息统计.....	25
5.7 本章小结.....	25
第 6 章 系统部署.....	26
6.1 系统编译.....	26
6.2 Makefile 文件.....	27
6.3 系统运行验证.....	27
6.4 本章小结.....	29
第 7 章 总结与展望.....	31

参考文献.....	34
-----------	----

# 第 1 章 概述

## 1.1 赛题要求

赛题名称：CV 方向——基于 VCK5000 平台实现 Filter2D 算子优化设计

### 性能评分（40%）

#### 基础要求：

- 要求使用 AIE API 或 AIE Intrinsic 设计 3x3 kernel on 64x64 image 的 filter2D function
- AIE Emulation 仿真成功并提交设计报告

#### 提高要求：

- 要求使用 AIE API, AIE Intrinsic 或 Vitis Vision Library 设计高清图像处理流水线设计
- 系统级别仿真成功，在 VCK5000 平台测试成功并提交设计报告

#### 评分规则

- 基础要求
  - 使用相同数量的 AIE 以及相同输入数据类型的情况下，设计占用的存储资源越少，计算周期越短的队伍得分越高

#### 提高要求

- 设计的输入图像尺寸越大(HD~4K)，帧率越高(FPS)得分越高
- 设计的创新度越高，系统基本的图像处理流水线的应用设计越合理得分越高

#### 工程测试

### 设计报告（40%）

#### 系统架构合理度 30%

如 AIE, PL 等计算资源合理使用  
设计的硬件架构合理度和性能优化

#### 架构分析和创新 10%

提出此新架构的优劣势，并提出改进想法  
提出此架构适合的应用场景  
挖掘设计的性能极限

#### 预赛得分 20%

## 1.2 开发平台介绍

### 1.2.1 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, 搭载了高性能加速核心——自适应智能化引擎 (Adaptive Intelligent Engine, AIE) 引擎阵列。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) 有机的将三种传统架构统合为一个加速引擎 (如图 1-2 所示), 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，以实现最大限度地提高编程的灵活性和硬件的可重构性<sup>[1]</sup>。

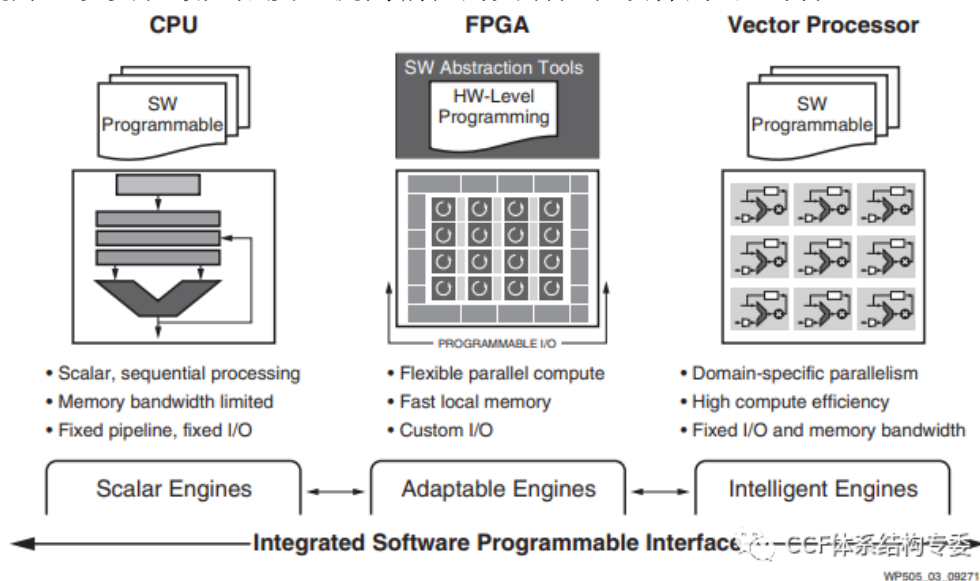


图 1-2 Versal ACAP 异构计算架构

## 1.2.2 软件平台

本设计采用 Vitis 工具（如图 1-3 所示）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™ 器件从而简化硬件设计和集成，主要涵盖四个方面：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<sup>[2]</sup>。

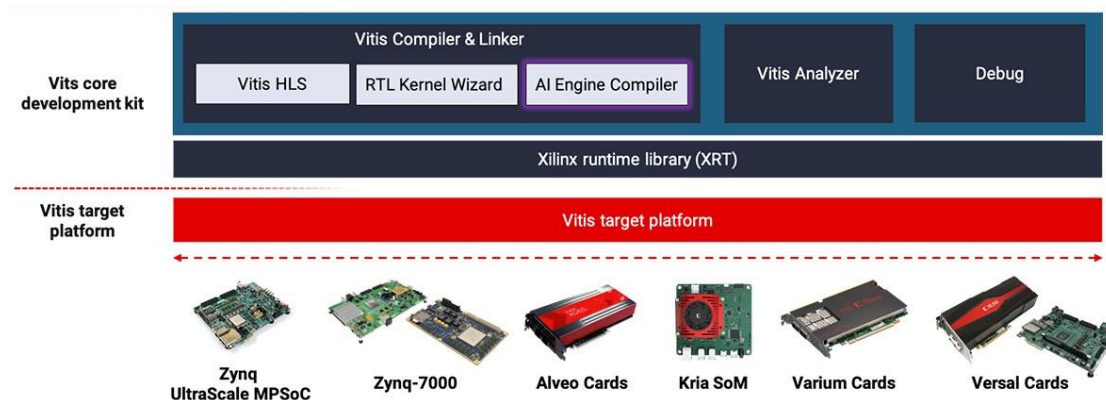


图 1-3 Vitis 软件平台

## 1.2.3 AIE 架构优势

在 VCK5000 平台中最为亮眼的当属 AIE 架构（如图 1-4 所示）。

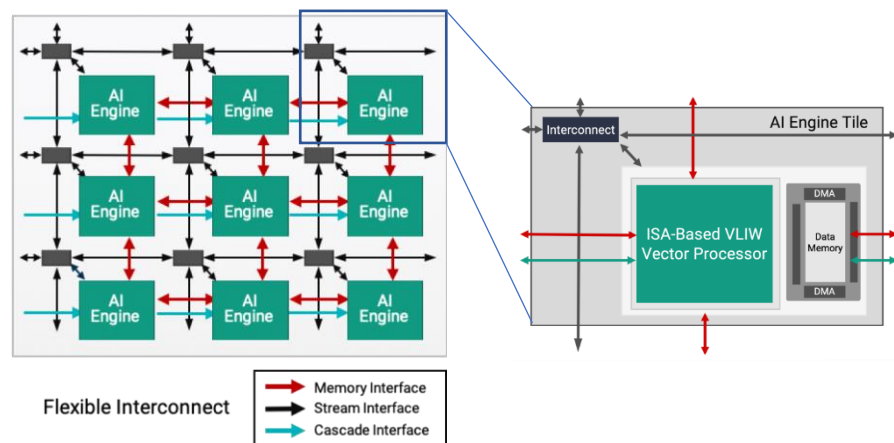


图 1-4 AIE 架构和 AIE Tile

其主要针对数字信号处理和人工智能应用提供加速，相比传统的 FPGA 架构和 GPU/CPU 等通用处理器，具有以下优势：

- **高效性：**一方面采用了 AIE tile array 拓扑结构，通过集群方式提供算力，以满足 AI 应用对于大规模的并行计算的需求；另一方面，采用 NoC 使得各大硬件结构之间可以高效互联，特别是可实现外部 DRAM 直接输入 AI Core 阵列，大大提升了芯片内部的数据传输能力；此外，由于提供了 VLIW 机制，可进一步提升性能；
- **灵活性：**AIE 架构提供多种互联通信机制（核间级联、直接内存访问、全局 AXI 等）；提供软件可编程(PS)和动态自适应重配置(PL)具有高度的可编程性；支持混合精度计算，可在保证精度前提下提高计算效率；提供可编程存储器层级的重配置机制，从而优化层次化存储器设计；
- **低功耗：**AIE tile 中的存储靠近计算端的设计，减少通讯开销节省能耗、降低运行成本；
- **快速部署：**AIE 架构与其他软件工具集成紧密，可以快速开发和部署，且 AIE 支持多种接口和标准，可以与不同系统高效集成。

### 1.3 竞题难点分析

针对特定硬件平台的算子开发和优化需要结合算子的数学计算特征和计算平台的软硬件资源进行深度定制化，以实现高效率 and 低资源占用率的解决方案。如图 1-4 所示，团队针对 Filter2D 算子的特点，我们对该算子进行常规的并行性和局部性分析，挖掘其对计算效率和存储消耗等方面的优化可能性，而后结合赛题要求，分析 VCK5000 计算平台的架构特点和软硬件资源条件能否支持此类定制计算和优化的实现以及可能出现的设计难点。

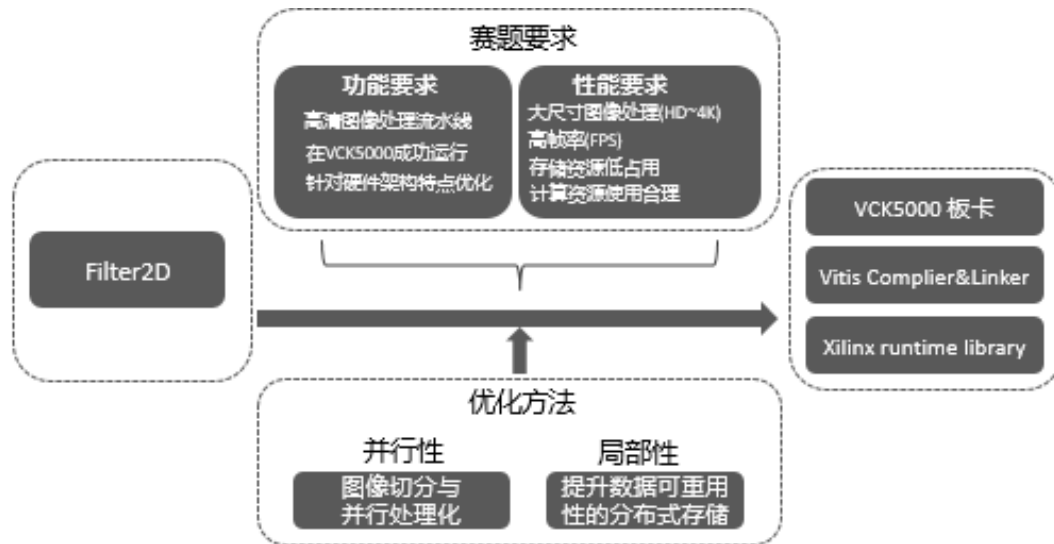


图 1-5 赛题难点问题分析示意图

Filter2D 的计算模式是一种典型的模版计算 (stencil computation)，因其在图像的每个像素位置执行的操作相同，即卷积核模版与图像对应像素区域之间的乘累加操作，且每次重复的操作之间并无执行顺序依赖，于是该算法天然适合并行计算，如果能对原始输入图像进行合理的并行任务粒度的切分，理论上可以最大化地利用多核计算体系的硬件资源，大大提高计算效率；若以行优先遍历的顺序对原始图像进行遍历计算，则前后两次的卷积运算所涉及的数据在原始图像上的空间位置十分接近，映射到硬件存储上则意味着已缓存的这部分数据会在极短时间内被重复访问，即具有时间局部性和空间局部性。通过设计该算法的存储结构和访存机制可以大大提高数据复用，减少不同存储结构之间的数据频繁交换，这样可最大程度减少片上存储资源的开销。

AIE 由一组创新的超长指令字 (VLIW) 和单指令、多数据 (SIMD) 的处理引擎以及独立的存储器构成 (即 AIE Tile)，且彼此之间具有灵活且高速通信模式和存储带宽。这些排列密集的计算核心共同构成了一个可灵活配置的并行计算矩阵，为实现 Filter2D 算法的并行化提供了硬件级别的支持，使我们可以通过 C++ 编程将定制的并行计算模式以计算图的形式映射到 AIE 中。算法的局部性优化需要设置合理的缓存策略和内存换入换出策略，得益于 Versal ACAP 上丰富的可编程逻辑结构 (即 PL 和 PS) 以及互联这些核以及 AIE 的高速片上网络 (NoC) 以及其接口，我们在设计存储器层级上具有较高的自由度；除此之外，多种数据传输方式，如窗口 (window) 和流 (stream) 等，也方便了我们根据输入数据的处理难度进行存储策略的调整，获得理想的效果。

考虑到并行性和局部性在特殊情况下可能存在冲突，比如在使用图像拆分方法实现并行计算时，需要把卷积的大量计算任务平均分配到 AIE 密集的核心中，

里面往往存在一些重复的图像数据。这需要根据设计的实际情况以及卷积算法的特性来权衡是否使用空间来换取时间上的优势，即损失掉一些数据的复用率来换取加速效率。

针对 8K 分辨率图像的处理情况，由于图像较大，所以无法使用 window 的传输方式进行图像数据的交互，需要使用 stream 方式进行数据交流并采用流水线处理。但由于卷积计算所需跳跃性地访问图像数据，如何使用 stream 的顺序访问方式来完成卷积计算则成为新的问题。并且在 AI Engine 中每个 Tile 能够使用的内存最大为 32KB，也不可能缓存大量图像到内存中，所以如何把数据流和数据缓存两种方式完美地配合起来是设计中的重中之重。

综上，我们分析了 Filter2D 的计算模式，并从底层硬件架构和资源的角度证明我们的并行性和局部性优化的初衷是可行的，并在最后阐述了待解决的设计难点。

## 1.4 解决方案

经过上述分析，本团队制定 filter2D 赛题的解决方案(如图 1-5 所示)。

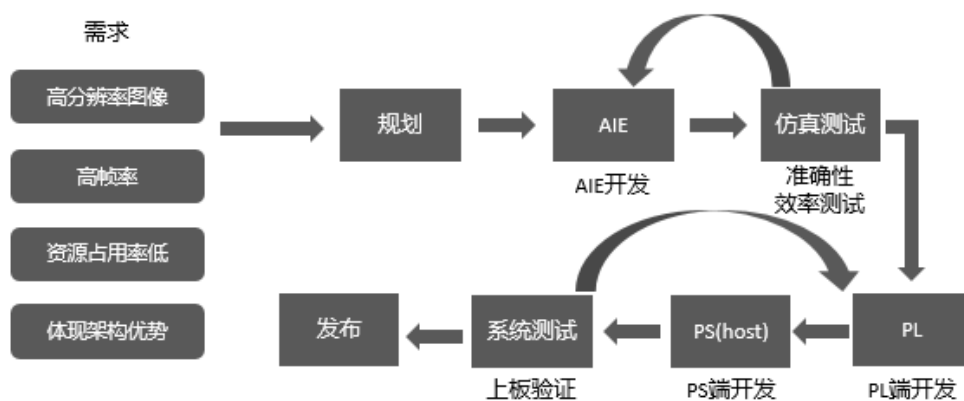


图 1-6 赛题解决方案流程示意图

首先，梳理本次竞赛对于高分辨率、高帧率、资源占用率低、展现架构优势等要求，并由此对整个赛题进行有针对性的规划；

而后，发挥 Versal ACAP 平台中 AIE 在处理高并行度矢量运算的优势，探索基于 AIE 实现 filter 算子高并行度处理的上限，之后由仿真测试的结果反馈调优 AIE 开发，探索性能上限。从提升并行化的角度考虑，图像切分的块数越多，AIE 的并行处理能力越强，计算时间越短，需要的 AIE 内核也越多，但在扩大并行度的同时不能超过硬件资源数量的限制，因此需要合理设计并行计算子任务的粒度，以及各个 AIE 内核之间的交互，达到整体的 AIR Array 的负载均衡，最大化地利用 AIE 资源。

之后，依据 AIE 并行化处理的要求，在平台硬件资源限制下，配合完成 PL



端的开发，并在 PS 端实现 JPEG 图像的解码、图像切分等预处理操作。这些切分得到的各个数据块将送到 AIE 中并行处理，在处理过程中使用数据流的访问模型并采用流水线的计算方法，计算完毕后再合成为一副完整图像，最终进行结果的输出。由于 Filter2D 的卷积计算的特性，每计算出一个结果都需要其上中下三行的图像数据，所以在图像切分时，为保证每一块图像中的每一行都能正常运算，所以需要多保留两行数据，与另外两块图像数据有交叉的部分。为了提高并行度，我们决定采用空间换取时间的策略，即每块数据保留冗余的数据，尽管分块后总的数据量变大，但能大大提高并行度，且随着图像分辨率越高，存储冗余数据所造成的存储开销成本与并行度的提高所带来的计算效率收益而言显得更微不足道。

在存储器层次设计方面，我们决定使用数据流与缓存配合的方式来解决卷积计算所需数据访问的跨越性问题，并且在 AIE 中的缓存要尽可能地平均分配到每个核心中，避免一个核心的内存压力过大，而其他核心的内存压力很小这种分布不均匀的问题。

## 1.5 文档组织结构

文档中第二章为系统设计，对该系统架构进行了简述，第三章为 AI Engine 设计，描述了 AIE 内核与 Graph 的设计与连接，第四章为 PL 设计，阐述了系统所需 PL 内核的功能以及性能指标，第五章为 PS 设计，描述了系统的运行流程，第六章为系统部署及运行评估。

## 第 2 章 系统设计

### 2.1 介绍

Filter2D 算法是一种常用的图像处理算法，主要用于图像滤波处理，包括噪声去除、边缘检测、图像增强等。它在计算机视觉、图像处理、模式识别等领域有着广泛的应用。

而 Filter2D 算法对硬件资源的需求量相对较大。第一，需要使用大量的存储空间来存储图像数据和卷积核数据，第二，需要进行大量的乘法和加法运算，因此需要较高的计算能力。第三，Filter2D 算法还需要大量的数据传输带宽，以支持图像数据和卷积核数据的传输。

基于以上原因，研究人员大部分时间都在 CPU 端及 GPU 端上做相关工作。然而，CPU 性能受限于运算速度和并行能力，GPU 的功耗较高，CPU 和 GPU 很难满足应用的延迟和功耗要求，特别是嵌入式场景。

随着近年来 FPGA 技术的发展，特别是随着片上存储存储器和其他资源的大量使用，FPGA 作为提高运算效率的专业加速器得到了广泛的应用。与 CPU 和 GPU 相比，FPGA 具有低时延和低功耗的优点。可编程性使得 FPGA 比 ASIC 更灵活。

尽管在 FPGA 中有这些优势，但较高的编程难度仍然阻碍了大多数开发人员在 FPGA 上部署网络。Xilinx 提出了一类新的可重构设备，称为自适应计算加速平台 (ACAP)，以高性能和低功耗为现代应用的计算和通信需求提供解决方案。并提出了在 ACAP 平台上开发提高用户抽象层次的软件解决方案。ACAP 是一种混合计算平台，它集成了传统的可编程结构、软件处理器和软件加速器引擎 (AI Engine)。ACAP 将标量引擎、适应性引擎和智能引擎与领先的内存和接口技术相结合，为任何应用提供强大的异构加速。ACAP 通过一系列工具、软件、库、IP、中间件和框架来实现所有行业标准设计流程。同时降低了软件工程师的开发难度。

为了充分利用 AIE 驱动的通用平台优势，我们的设计在 Xilinx ACAP 器件上提供了一个基于 AIE 的高性能 filter2D 加速器。

### 2.2 系统架构

Xilinx Versal ACAP (Adaptive Compute Acceleration Platform) 是一种具有灵活计算加速能力的平台，其系统架构包括 AI Engine、PL 端 (Programmable Logic) 和 PS 端 (Processing System)。

我们提出的设计方案是为了提高 filter2D 卷积处理速度，基于由 AI Engine 驱动的 Xilinx Versal ACAP 设备。所设计的加速器针对 filter2D 算法本身以及

针对新型 AIE 架构进行了优化。

该系统充分利用了通用 ACAP 异构平台的优势，系统架构图如图 2-1 所示。

AIE 核心是实现高性能卷积计算的关键。AI 引擎核心可以在 1.33GHz 下每个周期执行 8 个 int32 类型的乘法累积操作。其他操作使用 PL(可编程逻辑，传统的 FPGA 部件)端，用于对 AIE 输入输出的控制。

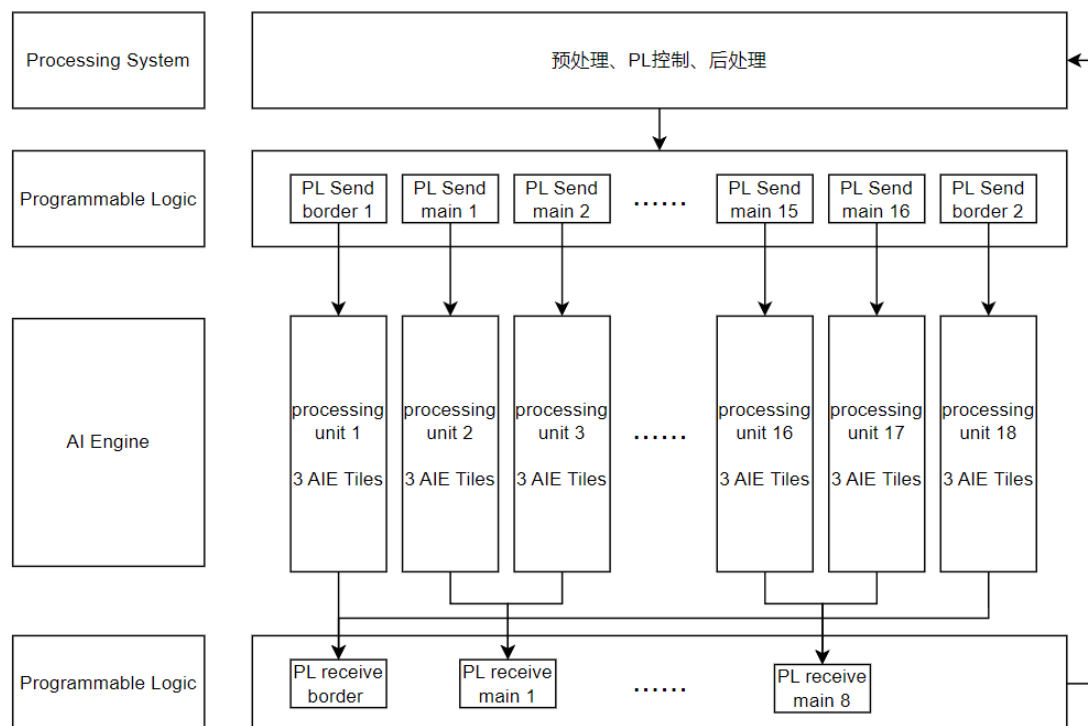


图 2-1 系统架构图

每个 AIE 处理单元所需的图像及卷积核数据通过 PL\_SEND 核心进行缓存和发送到 AI Engine，每个 AIE 处理单元输出的结果数据通过 PL\_RECEIVE 核心接收。主机控件在 PS (Processing System, X86 CPU) 端使用 Xilinx 运行时库 (XRT) 管理、控制系统运行流程。

## 第 3 章 AI Engine 设计

### 3.1 技术难点分析及设计目标

Filter2D 在图像处理等领域有着十分重要的地位，能够在自己设定卷积核的情况下进行图像卷积运算。其最显著的内存访问特征是需要跨行访问图像，这就引出了一个问题：如何使用在数据流的方式顺序数据访问的情况下实现高速卷积计算？

一个普遍的方法是在处理时先缓存两行图像在内存中，在读取第三行的时候开始进行卷积计算。但在 AIE 这种内存较小的硬件结构上，运算高分辨率图像时存储两行图像所需的内存容量是远远不够的，因此需要做更细粒度的切分和数据预处理操作。

缓存图像的另一种替代方法是在数据输入到 AIE 之前就做好了图像重塑和整理等操作，如把运算一次卷积所需的数据按照顺序依次排列好再输入 AIE，以实现不需要在 AIE 内存中缓存的效果。但经过实验后，该方法与缓存图像的方式相比，输入数据流需要传输的数据量是其他方式的三倍，与加速的理念相违背。

在实验中暴露了两个相互冲突的设计目标。首先我们希望能够达到尽可能高的卷积计算速度。其次，我们也希望能够在保证卷积的高速运算的情况下同时最小化内存和通信开销。因此，第二个研究问题出现了：如何在这两个设计目标之间进行权衡？

为了解决这些研究问题，我们设计了以下 AIE 架构用来解决上述问题。并根据 filter2D 本身的特点，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<sup>[3]</sup>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

Filter2D 的内存访问特征为需要跨行访问，若要使用数据流顺序读取必然要进行缓存，所以整体采用数据流与缓存部分数据配合的设计。每个图像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，

我们希望在设计过程中能够保证此 AIE 架构可以支持最高 8K 分辨率图像。目前 4K 分辨率已经普及，8K 分辨率也将在 2025 年普及，所以支持 8K 图像处理是非常有必要的。

我们使用了 AIE 阵列负责 8K 图像下 3x3 卷积核 filter2D 的核心运算, 这里把能够运算出一整幅图像或图像连续的一部分结果的模块叫做一个处理单元, 一个处理单元包含 3 个 AIE 内核。决定一个处理单元需要处理数据量的参数包括 PART\_NUMBER (图像切分块数)、HEIGHT (图像纵向分辨率) 与 WIDTH (图像横向分辨率), 其中 PART\_NUMBER 数值越大, 处理单元越多, 同时处理速度也会提高, 但系统开销也会增大, 需要根据任务所需的芯片资源核性能要求来确定此值, 以达到最佳效率。若 PART\_NUMBER 值为 1, 则仅由一个处理单元运算整幅图像。在此方案中 PART\_NUMBER 值为 18, 即总共使用了 54 个 AIE 内核, 每个处理单元负责源图像的 1/18。其中图片切分方法如下图 3-1 所示:

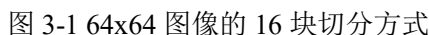


图 3-2 中显示了一个包含 3 个 AIE 核心的处理单元在输出一组 v8int32<sup>[2]</sup>结

果时的运行情况，因为 AIE 矢量运算器可以在一周期内完成 8 个 int32 类型的乘法运算，所以处理单元都是以 8 个 int32 为单位来输入和输出数据的。想要运算出一个 v8int32 卷积结果，需要存储 9 块 v8int32 来完成运算，也因此图像的横向分辨率必须是 8 的倍数。为了达到更高的运算效率，我们把卷积拆分到两个 AIE 核心中进行计算，并在其之间传递 ACC 来实现累加。

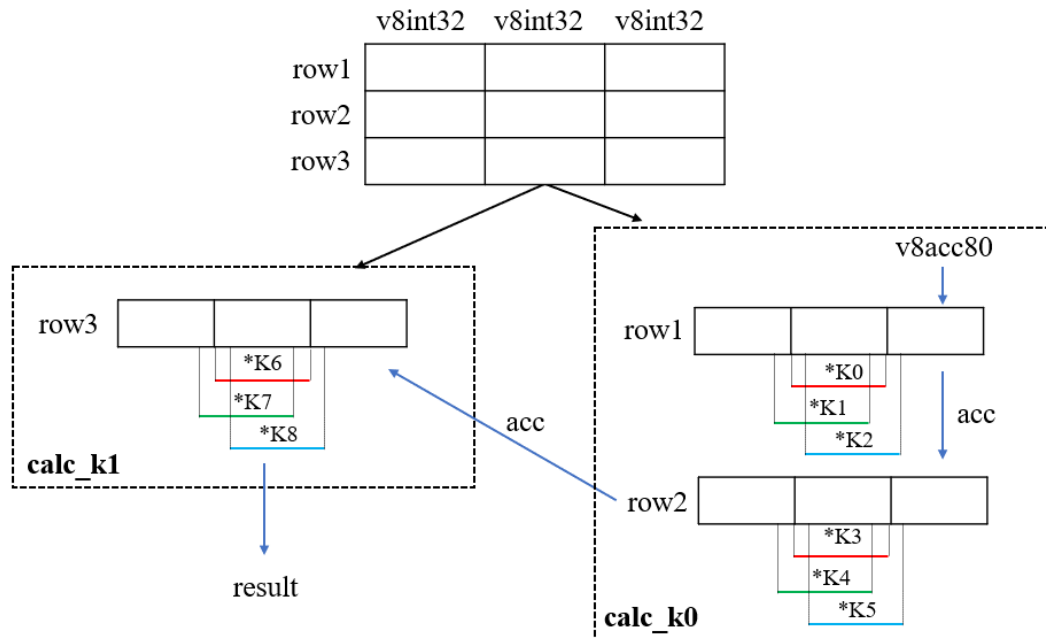


图 3-2 处理单元的运行情况

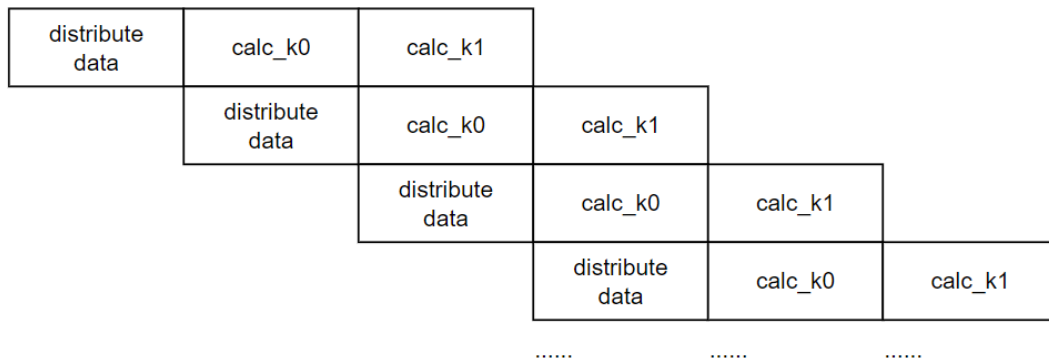


图 3-3 AIE 处理流水线

AIE 的数据路径使用流水线的方式，其中处理单元的三个 AIE 内核数据流水原理图如图 3-3 所示。在流水线启动前，因为两部分卷积运算所需数据范围不同，所以存在一个启动过程，首先给 calc\_k0 发送第一行图像数据并缓存起来，之后在 distribute 核读取第二行并缓存，至此启动过程完成。在 distribute 核心正式分发数据的时候给 calc\_k0 分发缓存内的一行图像，给 calc\_k1 分发新读取进来的一行图像，并更新自己的缓存区，这样就都符合了两个计算核所需的数据范围并可以启动流水线。

在处理的过程中，虽然运算一组结果需要 9 块 v8int32 数据，每行需要前中后三块 v8int32，但在实际运算过程中，可以为每一行设置一个缓冲区，缓冲区内已经存储了先前读入的两组数据，需要运算新一组数据时，只需要把缓冲区以向前移一组，再新读取一组 v8int32 放进来，即可继续做下一次卷积运算。这样就可以使用很少的空间来很好地减少通讯数据量。

在流水线启动后运算完毕一块图像的时间就约等于读取完毕一块的图像的时间，图像块数据量越大此现象越明显。在流水线运行时，distribute 核心与 calc\_k0 分别存储一行图像，calc\_k0 的缓存中存储的是目标运算行的上上行图像，distribute 的缓存中存储的是目标运算行的上一行图像，在输出时即可做到两个计算核错行输出，并且这种方式缓存的两行图像分布在两个 AIE 内核中，达到了节省内存的目标，同时这种方式可以支持 8K 图像，也达到了处理超高清分辨率的目标。在经过实验验证后，证明目前的方案确实解决了在 8K 图像范围内使用流式顺序数据访问的情况下实现高速卷积计算的研究问题，具体实验结果数据会在后续章节体现。

### 3.4 Graph 设计

在三种 AIE 内核设计完成并进行数据流的连接后，我们就可以得到一个处理单元的 Graph。数据分配核心从总输入数据流中读入图像数据，并通过其两条数据流用来给两个计算核心发送所需的图像数据，两个计算核心之间有一条级联数据流用作 ACC 的传递，并在计算核 1 进行最后的累加并向总输出数据流中输出计算结果。

由于计算核 1 的数据输入流是直连数据分配核心，而它运算所需的 ACC 则由计算核 0 发送出来，尽管 AIE 核心运行非常快速，但是相比直连路径也会有一些延迟存在，所以在 distribute 核心的两个分发路径中分别设置了两个 DMA FIFO 用来平衡速度上的细小差别，防止影响运行流畅度<sup>[4]</sup>。处理单元的 Graph 如下图 3-4 所示：

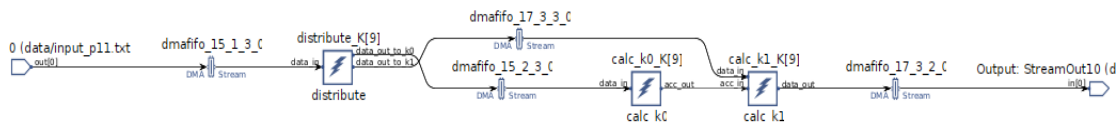


图 3-4 处理单元 Graph

在得到一个处理单元之后，若把单元看作一个整体，则这个整体只有一条外部输入数据流和输出到外部的数据流，对应着图像输入和结果输出，目前已经可以依靠其实现一整幅图像的卷积运算，但处理单元存在的意义就是为了方便复制多个单元并行计算一副图像，从而提高处理速度，所以复制多个处理单元同时运



行是非常必要的。

在上文中提到，我们采用的图像分块数量为 18，也就对应存在 18 个处理单元并行计算。在对应的处理单元输入流中输入分块后的图像数据，多个输入图像块可以被同时处理。整体的 Graph 如下图 3-5 所示：



图 3-5 整体 Graph

### 3.5 内存与速度的权衡

在前期设计中，从 64x64 到 HD 图像，从 HD 到 8K 图像，为了在使用 8K 图像时保证高运算速度，内存问题一直是困扰我们的主要因素。既不能为了减少内存消耗而大幅牺牲速度，也不能在追求速度时无节制地使用内存。在 AIE 中每个 Tile 最大能够使用的内存为 32KB，PL 端能够使用的 BRAM 也非常少。所以在处理高分辨率图像时，内存消耗与运行速度必须做出权衡<sup>[4]</sup>。

在 filter2D 中，由于运算出一个像素的结果必然要跨行访问图像的特性，导致卷积运算对于数据流这种流式输入方式十分不友好，所以大部分的做法是缓存两行图像后再开始运算，并在运算的过程中不断更新缓存，能够达到算法的效果。但是这种方式非常消耗内存，低分辨率图像 AIE 的内存还能够承受，但分辨率一旦上升到 4K 或者 8K，AIE 就不能承受缓存两行图像所需的内存空间，所以这就需要使用另外的方式来处理内存问题。

在上文 3.3 小节中阐述了对于高分辨率图像的图像缓存策略，AIE 运算单元的两个运算核所需的图像范围不同，其中一个计算核不需要图像的最后一行，第二个计算核不需要图像的前两行，但是中间部分的图像是两个核都需要的。所以我们采用的方式为：整个 AIE 运算单元总共存储两行，但分布在不同的 AIE Tile 中。所以不论需要计算多少分辨率的图像，每个核所需的内存大小仅为一行图像数据的大小，对于 8K 图像来说一行图像的大小为 30720B，加上其他的一些内存



开销，AIE 的 32KB 内存刚刚好能够承受。AIE 中一个处理单元的内存使用情况如下图所示 3-6 所示：

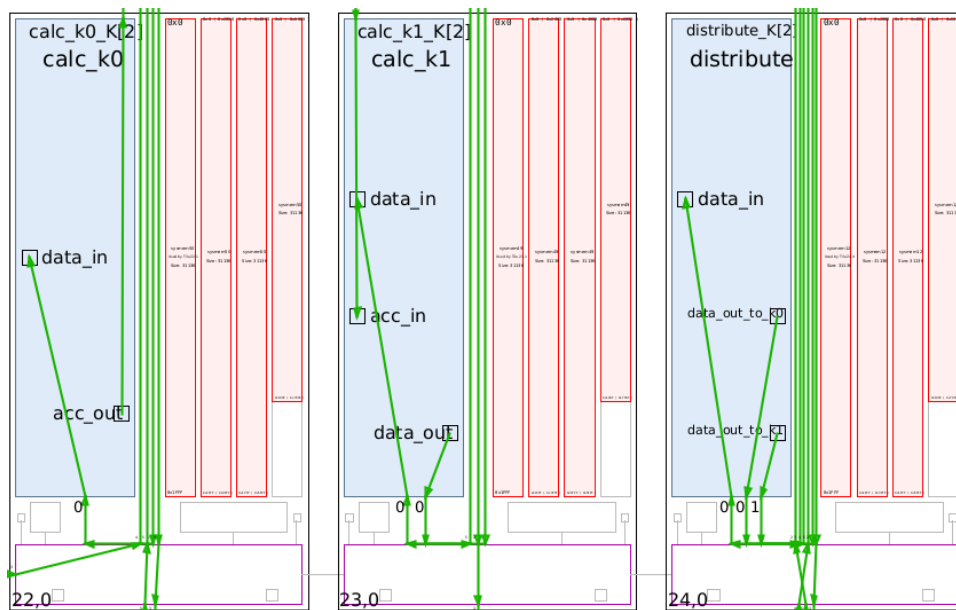


图 3-6 处理单元内存使用

在此章节的开头我们提出了有一种可以不消耗 AIE 内存的方式：在 PS 端或者 PL 端把运算每个像素所需的数据都按顺序排列好后再送入 AIE 中，这样 AIE 就免去了数据访问的烦恼，直接按照顺序读取即可。

但这种方式最大的问题在于输入数据流的数据量比直接输入图像要大三倍，这也就代表着运算效率要低三倍，经过实验分析后，真实情况也确实如此。经过内存消耗与速度的折中考虑，我们没有选择这种方案，因为目前的内存分配方案能够承受 8K 图像，已经满足设计要求，没有必要牺牲大量速度去换取内存的低利用率。

经过后续实验证实，目前的 AIE 设计方案可以在 8K 图像范围内做到运行速度与资源消耗的平衡，在 8K 图像下的内存使用没有超出 AIE 内存限制，并且处理完毕一副图像仅需要 9ms 左右，具体上板运行数据会在后续章节描述。说明此方案解决了在速度和内存这两个设计目标之间的冲突的研究问题。

### 3.6 性能评估

对 AIE 的性能评估我们选择使用 Vitis 工具链自带的软件仿真器来仿真 AIE 运行，从而得到运行时间以及运行结果。

由于 AIE 的软件仿真十分耗费时间，在高分辨率图像下需要运行很长时间才能仿真完毕，所以我们在 AIE 仿真中采用 8K 图像。在此次性能评估中使用的参数为 64x64 图像，16 路并行运算，对于真正运算 8K 图像的运行结果，在后续章节中会详细描述在 VCK5000 板卡下运行 8K 图像的实际运行结果。

在 AIE 仿真完毕后，经过验证计算结果无误，使用 Vitis analyzer 工具在 Trace 视图中查看所有 AIE 内核的运行时间统计。运行情况如图 3-7 所示：

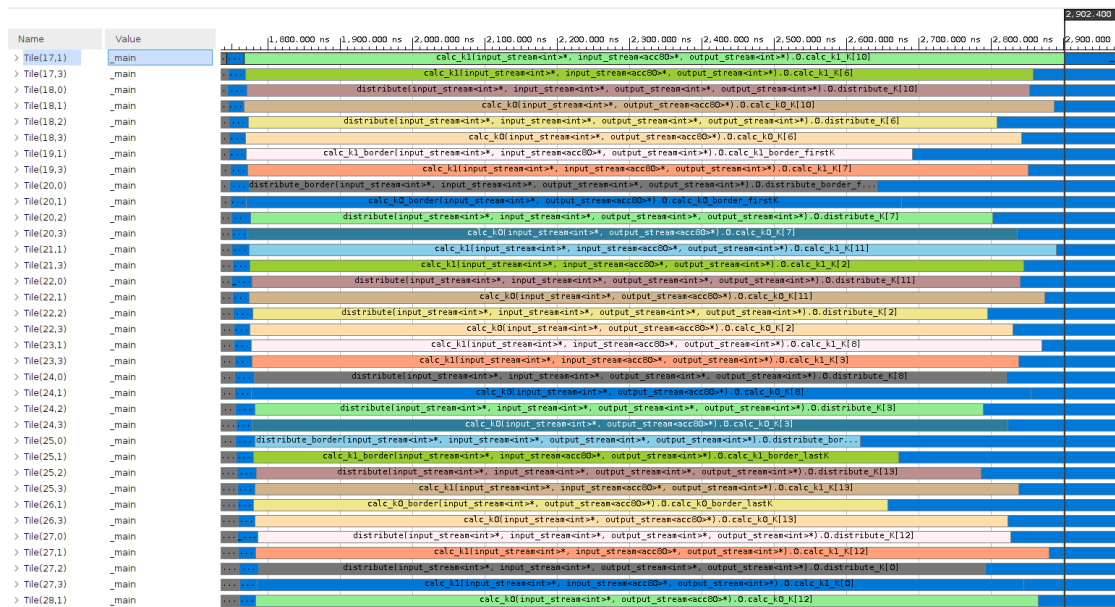


图 3-7 AIE 内核运行情况

通过观察图像得出了以下结论：通过图中显示的数据可以计算出整个系统在 16 个计算单元的情况下计算完毕一幅 64x64 图像需要消耗 1138ns，我们可以使用这个运行时间来推算出 8K 图像的运算时间： $7680/64 \times 4320/64 \times 1138 \approx 9.2\text{ms}$ ，此推算数据与我们的实际在 VCK5000 中得出的运行时间高度吻合。

同时可以计算出系统在 64x64 图像下的处理帧率为 878734.62FPS，在 8K 图像下的处理帧率为 108.69FPS。但因为仿真使用的每块图像行数较少，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达到更高，若在实际运行 8K 图像时，则流水线启动对时间的影响微乎其微。经过 AIE 仿真验证，此设计符合预期目标。

### 3.7 本章小结

本章主要介绍了此系统在 AI Engine 方面的详细设计。首先，我们提出了 filter2D 在 AIE 上实现的技术难点，提出了设计目标，说明了图像分块方法。在 AIE 核心设计部分，我们详细阐述了 AI Engine 内核的设计原理和思路，说明了一个在处理单元内 3 个 AIE 内核的数据流动和运算流程。在 Graph 设计部分，我们介绍了在处理单元内部的 AIE 核心是如何连接的，并对处理单元做了复制。在 AIE 内存占用优化部分，我们说明了如何对超高分辨率图像做内存优化，使内存消耗小于 32KB。最后，我们对于此设计使用软件仿真做了性能评估。总之，本章在 AIE 设计上克服了一系列研究难题，做到了在 8K 图像范围内运行速度及内存占用的权衡。

## 第 4 章 PL 设计

### 4.1 技术难点分析及设计目标

AIE 阵列通过 AXI-stream 接口与 PL 侧连接, PL 端的数据移动引擎负责通过 AXI4 接口在 DDR 和片上 ram 之间传输图像数据以及卷积核。由于 PL 端是与 AIE 数据流直接交互的模块, 所以首先可以考虑到的研究问题是: 如何保证 PL 端的数据处理速度能够支持 AIE 满速运行?

首先, 在 VCK5000 板卡中 DDR 内存能够提供 102.4GB/s 的带宽, 这里会出现两种情况, 第一种是 AIE 的处理速度小于 DDR 内存带宽; 第二种是 AIE 处理速度大于 DDR 内存带宽。对于第二种情况的其中一种解决方案是把图像缓存到 BRAM 中, 再输出到 AXI-stream 接口, 但由于 PL 端的 BRAM 资源不足以存储高分辨率图像, 所以此方式不能作为解决方案, 可以考虑缓存小部分图像的方式。

在目前的方案中, AIE 所需带宽是可以计算的, 计算数据来源于上一章中 AIE 仿真时的数据, 因为 AIE 仿真的数据输入速率是足够的, 所以可以作为参考。目前一路处理单元所需的内存带宽为:  $7680 \times 4320 \times 4 / 0.009 / 16 / 1024^3 \approx 0.86\text{GB/s}$ 。由计算可以得出 AIE 所需带宽与图像分块数成正比, 所以又引出了第二个研究问题: 目前硬件资源最多可以支撑多少路运算单元同时运行?

对于以上研究问题, 我们提出了以下 PL 端的设计目标:

(1) 必须满足 AIE 处理单元的处理速度。

(2) 在硬件条件不变、保证速度和稳定性的情况下, 能够支持同时运行 AIE 处理单元数量越多越好。

### 4.2 功能描述

我们以负责向 AIE 发送数据和负责从 AIE 接收数据作为根据, 把 PL 核心分为了两大类: pl\_send 核心与 pl\_receive 核心。其中一个 pl\_send 核心仅负责一个数据单元的数据输入, 并且在输入图像数据前, 需要先发送卷积核数据, 从上文中可以了解到图像分块数量决定了 AIE 处理单元的数量, 同时也决定了 pl\_send 核心的数量。一个 pl\_receive 核心负责两路 AIE 运算单元的输出。

在根据发送和接收数据对 PL 核心进行分类后, 还需要根据处理图像数据块的大小对这两大类进行再次分类。由于图像切分后, 第一块图像以及最后一块图像都比中间图像块少一行数据, 所以这两类发送的数据量不同, 需要再次进行分类。最终分类为四种 PL 核心: pl\_send\_border (边界块发送 PL)、pl\_send\_main (中间块发送 PL)、pl\_receive\_border (边界块接收 PL)、pl\_receive\_main (中间块接收 PL)。

在目前的方案中，共有 18 路处理单元同时运行，对应着四种核心所需的数量为：2 个、16 个、1 个、8 个，总计 27 个 PL 内核。PL 核心与 AIE 处理单元的关系如下图 4-1 所示：

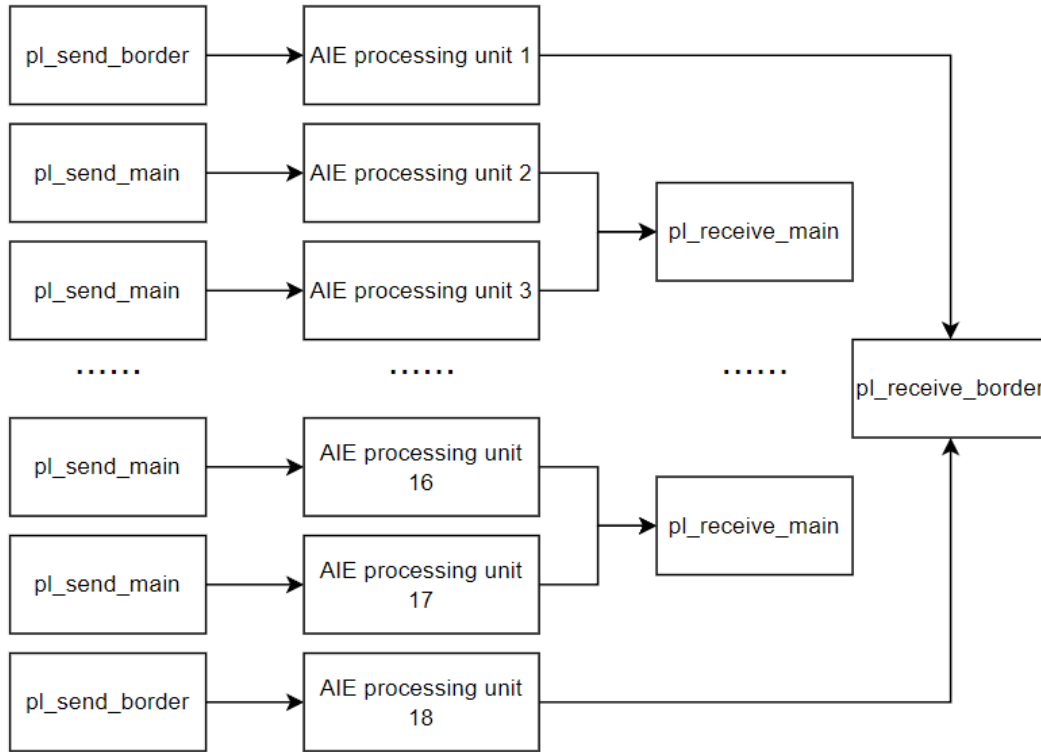


图 4-1 PL 核心与 AIE 处理单元的关系图

## 4.3 核心设计

### 4.3.1 PL\_SEND 核心设计

此 PL 核心主要负责 AIE 所需的卷积核以及图像数据。每个核心需要发送的数据量由以下参数决定：PART\_NUMBER (图像切分块数)、HEIGHT (图像纵向分辨率) 与 WIDTH (图像横向分辨率)。

其中 main 类 PL\_SEND 核心能够访问到的 DDR 内存空间大小为  $((\text{HEIGHT}/\text{PART\_NUMBER}+2)*\text{WIDTH}+12)*4\text{B}$ ，border 类 PL\_SEND 核心能够访问到的 DDR 内存空间大小为  $((\text{HEIGHT}/\text{PART\_NUMBER}+1)*\text{WIDTH}+12)*4\text{B}$ 。对于算式中 12 的意义是卷积核的发送数量，虽然 3x3 卷积核只有 9 个数据，但是在 AIE 中 int32 的数据流读取是以四个 int32 为单位读取的，所以需要 3 个空白数据做填充。

在核心中申请了一个有 16 个 int32 大小的数组作为缓冲区，有两个目的，第一个目的是为了一次从 DDR 读取进 16 个数据再统一发送，避免对 DDR 的频繁访问。第二个目的是方便使用 memcpy 函数触发总线突发传输，提高访问效率<sup>[5]</sup>。

PL\_SEND 核心的程序流程图如下图 4-2 所示：

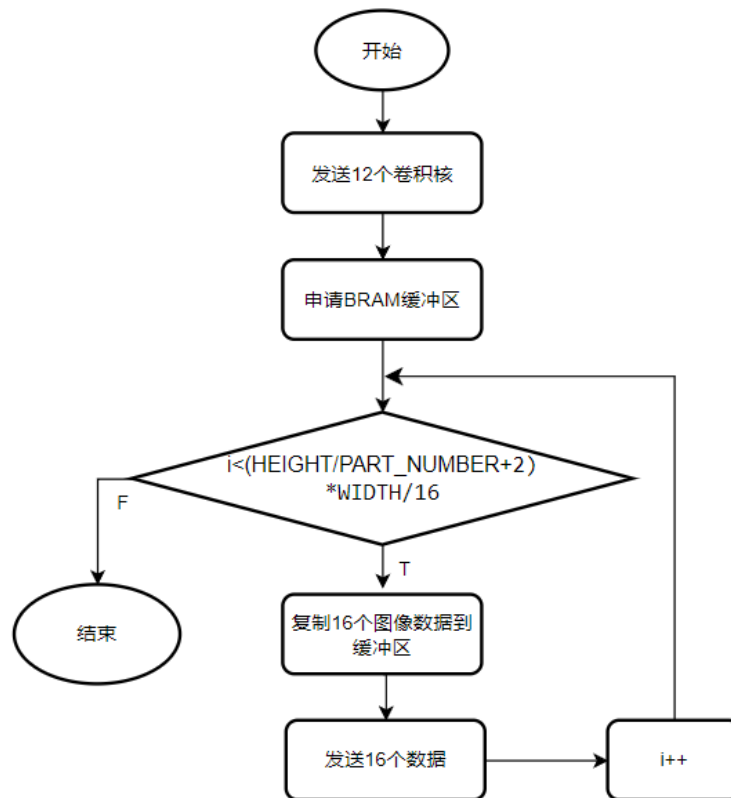


图 4-2 PL\_SEND 流程图

在上文中可知，共有 18 路 AIE 处理单元，由此可以计算出所有输入所需的总带宽为： $18 \times 0.86 = 15.48 \text{GB/s}$ ，DDR 内存带宽可满足此要求。在内核编写完成后，使用 Vitis analyzer 中查看综合报告，可得知 PL\_SEND 内核的 Latency 与资源消耗，下图 4-3 为编译报告详情：

Target Frequency	Estimated Frequency
250	342.465759
250	342.465759
250	342.465759

Start Interval	Best (cycles)	Avg (cycles)	Worst (cycles)	Best (absolute)	Avg (absolute)	Worst (absolute)
15	15	15	15	60.000 ns	60.000 ns	60.000 ns
1858563	1858563	1858563	1858563	7.434 ms	7.434 ms	7.434 ms
1858724	1858723	1858723	1858723	7.435 ms	7.435 ms	7.435 ms

FF	LUT	DSP	BRAM	URAM
237	259	0	0	0
421	305	0	0	0
2476	3772	0	0	0

图 4-3 PL\_SEND 编译相关信息

从图中可以看出，PL\_SEND\_MAIN 内核总共花费 7.435ms 来搬运数据，明显小于 AIE 的 9ms 处理时间，PL\_SEND\_BORDER 内核信息极为相似，故不展示。在数据中可以分析出目前 PL 核心能够支持 AIE 满速运行。

### 4.3.2 PL\_RECEIVE 核心设计

此 PL 核心主要负责接收 AIE 运算单元输出的结果。每个核心需要接收的数据量由以下参数决定：PART\_NUMBER(图像切分块数)、HEIGHT(图像纵向分辨率)与 WIDTH(图像横向分辨率)。

其中 main 类 PL\_RECEIVE 核心能够访问到的 DDR 内存空间大小为  $(HEIGHT/PART\_NUMBER * WIDTH + 12) * 4B$ , border 类 PL\_RECEIVE 核心能够访问到的 DDR 内存空间大小为  $((HEIGHT/PART\_NUMBER - 1) * WIDTH + 12) * 4B$ 。从计算中能够分析出 PL\_RECEIVE 核心对比于 PL\_SEND 对于每一路处理单元所需要处理的数据量要少很多。

PL\_RECEIVE 核心对比于 PL\_SEND 核心最大的不同点是 PL\_RECEIVE 核心需要同时负责两路 AIE 处理单元的输出，而 PL\_SEND 只需要处理一路。这两路输出数据在 DDR 中的存储位置不一样。为了顺序访问 DDR 内存，我们使用了一个大小为 16 的缓存数组，把一路接收的 8 个数据放在缓存的高 8 位上，把另外一路接收的 8 个数据放在缓存的低 8 位上，然后再把缓存中的 16 个数据按顺序写入到 DDR 中，但此方式虽然做到了内存顺序访问，但没有把数据存放到预期的位置，所以在运行结束后需要 PS 端做一次数据重塑，把顺序还原回正常的的数据排列顺序。

PL\_RECEIVE 核心的程序流程图如下图 4-3 所示：

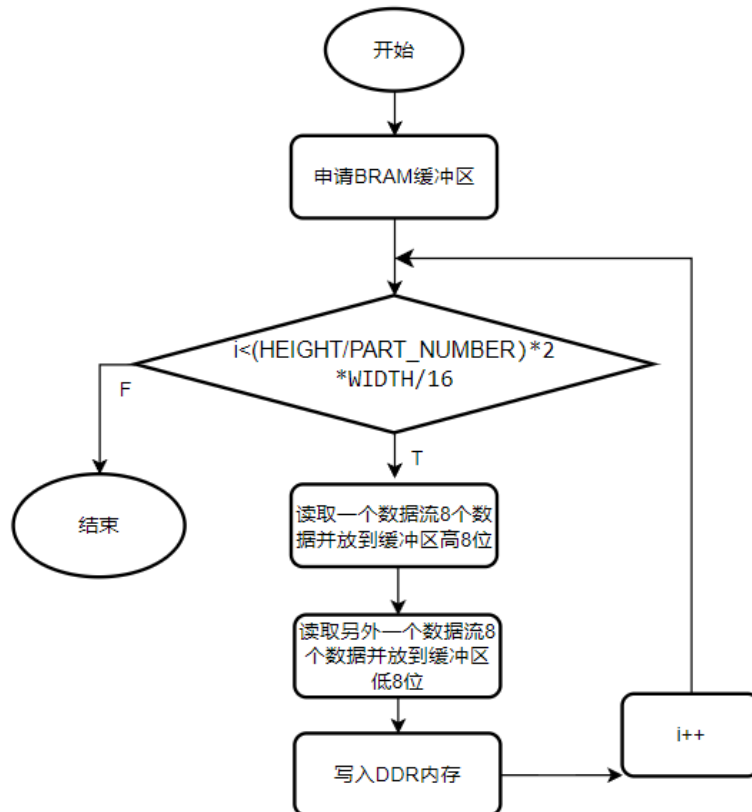


图 4-4 PL\_RECEIVE 流程图

在此我们可以使用输出图像尺寸等参数来计算出 18 路 AIE 运算单元输出数据所需的 DDR 内存带宽： $7678 \times 4318 \times 4 / 0.009 / 1024^3 \approx 13.72 \text{GB/s}$ ，结合 PL\_SEND 所需的  $15.48 \text{GB/s}$ ，总共需要 DDR 内存带宽为  $29.2 \text{GB/s}$ ，DDR 内存带宽可满足此要求。

在内核编写完成后，使用 Vitis analyzer 中查看综合报告，可得知 PL\_RECEIVE 内核的 Latency 与资源消耗，下图 4-5 为编译报告详情：

Target Frequency	Estimated Frequency
250	342.465759
250	342.465759

Start Interval	Best (cycles)	Avg (cycles)	Worst (cycles)	Best (absolute)	Avg (absolute)	Worst (absolute)
1843202	1843202	1843202	1843202	7.373 ms	7.373 ms	7.373 ms
1843274	1843273	1843273	1843273	7.373 ms	7.373 ms	7.373 ms

FF	LUT	DSP	BRAM	URAM
475	97	0	0	0
5284	8542	0	0	0

图 4-5 PL\_SEND 编译相关信息

从图中可以看出，PL\_RECEIV\_MAIN 内核总共花费  $7.373 \text{ms}$  来搬运数据，小于 8K 图像的运行时间  $9 \text{ms}$ ，能够支持 AIE 满速运行，在结合 PL\_SEND 的编译报告的情况下，可以确定此方案解决了保证 PL 端的数据处理速度能够支持 AIE 满速运行的研究问题，并且没有 DDR 内存带宽瓶颈的顾虑。

## 4.4 存在的不足

在上文中提到，我们总共使用了 18 路 AIE 处理单元、PL\_SEND 使用一对一方式服务、PL\_RECEIVE 使用一对二方式服务的方案。那我们为什么会选择这种方案呢？事实上限制主要来源于 PL 端，对于 AIE 方面，处理单元可以复制很多路，直到 AIE 阵列的核心不够用为止，而 18 路所需的 AIE 核心远远没有达到 AIE 限制。

在起初，我们的 PL 方案采用输入和输出全部都是一对一的方式来服务，18 路则需要 36 个 PL 核心，对应着需要消耗 36 个 NOC 资源，但 NOC 资源总共能够支持 28 个 PL 核心同时运行，这也就是为什么我们前期只使用 12 路的原因。之后我们查阅资料后提出了一种方案：能否仅仅使用两个 PL 核心，一个 PL 核心负责所有输入，一个 PL 核心负责所有输出呢？我们开始了尝试，编写完毕进行上板的实际实验结果是，计算可以顺利完成，但速度十分不理想，下图是我们使用在 4K 图像下仅使用两个 PL 核心的运行截图：



```
TEST PASSED !  
AIE filter2D take up 162.901ms  
PS filter2D take up 381.264ms  
Speedup ratio:2.34046
```

图 4-6 实验方案运行截图

在运行结果中可以看出，实际运行速度非常慢，在 4K 图像下甚至达到了 162ms，我们又继续探索，但因为知识储备以及时间紧迫等原因，此方案没有被使用。

后来，我们又提出了一个方案：既然一个 PL 核心负责不了全部数据流，只让一个 PL 核心负责两个数据流是否可行呢？这种方案可以把处理单元的个数比原来增加两倍。在经过一系列的尝试过后，我们确实可以使用 24 路处理单元达到对比 PS 端 180 倍的加速比，但十分不稳定，能跑出这个效果的次数只占运行总次数的四分之一左右，最终因为不稳定，导致最后也没有采用。

目前的方案则是上文方案中的改版，我们发现使用 18 个处理单元、PL\_SEND 使用一对一方式服务、PL\_RECEIVE 使用一对二方式服务的方案运行效率十分稳定，并且也有很好的加速效果。并最终采用此方案。

但目前这种方案依然限制了 AIE 发挥，不能利用全部的 AIE 阵列资源，所以我们会继续研究更加优化的方案，努力做到充分发挥 AIE 阵列的硬件优势、利用全部的硬件资源。

## 4.5 本章小结

本章主要介绍了此系统在 PL 端的详细设计。首先，我们阐述了 PL 端为服务 AIE 良好运行的技术难点，提出了设计目标。描述了 PL 内核的分类，需要完成什么功能。其次我们对 PL\_SEND、PL\_RECEIVE 类型内核的详细实现做了阐述。最后，我们描述了目前 PL 端存在的一些不足以及对处理单元数量的一些考虑和实践。总之，我们解决了在最大 18 个计算单元的情况下，保证 PL 端的数据处理速度能够支持 AIE 满速运行的研究问题。



## 第 5 章 PS 设计

### 5.1 设计目标

VCK5000 板卡使用 x86 主机作为 PS 端，开发人员可以使用常见的 x86 开发工具和库来开发和优化应用程序。为开发者提供了熟悉的软件开发环境，可以轻松进行软件开发和调试。

PS 端通过 PCIE 通道与 VCK5000 板卡进行通讯，负责通过 Xilinx 运行时库 (XRT) 管理、控制 AIE graph 和 PL 内核。为适配 filter2D 应用以及上述设计，我们对其提出以下设计目标：

(1) 图片文件操作：运算时需要的图像数据来源于 JPEG 图片，解码图片有两种方案，一种是使用 OpenCV 库，另外一种是使用官方 PL 内核进行 JPEG 解码。同时 PS 端负责把运算结果输出到图像中。

(2) 正确的数据预处理和后处理：为保证 PL 以及 AIE 的正确运行，需要对图像做拆分、重塑等操作，确保计算流程的正确进行。

(3) 计算结果验证：PS 部分需要独立执行 filter2D 算法，并将运算出的正确结果与 AIE 运算结果比对，以验证结果的正确性。

(4) 计算信息统计：在运行过程中，PS 端需要对 PL 的运行时间、PS 端 filter2D 运行时间做统计，以提供运行时间、加速比、FPS 等报告信息。

### 5.2 功能概述

PS 端程序采用 C++17 标准以及 XRT Native API<sup>[6]</sup> 对整体架构进行时序管理。首先，PS 端要使用 XRT API 打开 VCK5000 设备，把编译好的 xclbin 文件加载进设备中，等待文件加载成功后即可开始运行。

首先需要读取输入图像，解码完毕 JPEG 图像后，在 PS 端转为灰度作为输入图像，并转化为 int32 类型的一维数组存储。之后就可以对其进行图像切分，切分方法在第三章 AI Engine 设计中进行了介绍。切分完毕后可以获得多个小型数组，可供 PL 内核创建时使用。

数据预处理完成后，PS 端就可以获取对 PL 内核的引用，为每个 PL 内核分配 `xrt::bo[6,7]`，把对应的输入数据写入到 `xrt::bo` 中，最后把数据同步到设备侧。PS 端不需要对 AIE 进行直接的控制以及调用，在硬件电路中 PL 内核已经与 AIE 阵列连接好，只需控制 PL 内核即可。在数据同步完成后，即可启动所有 PL 内核的运行，并开始计时。等待所有内核运行结束后结束计时。至此在 PL 和 AIE 端的所有运算操作已经执行完毕，只需要把设备侧的数据同步回 PS 端的内存中，就可以获得 AIE 的运算结果。

获得所有 PL\_RECEIVE 的输出结果后，在 PS 端需要对其进行重塑及合并，即数据后处理之后，才能得到完整的一幅图像。数据重塑完成后，PS 端存储的即为完整的一幅输出图像，在最后就可以输出运行过程中的一些性能指标，并把运算结果输出到 JPEG 图像中。下图为 PS 端的运行流程图：

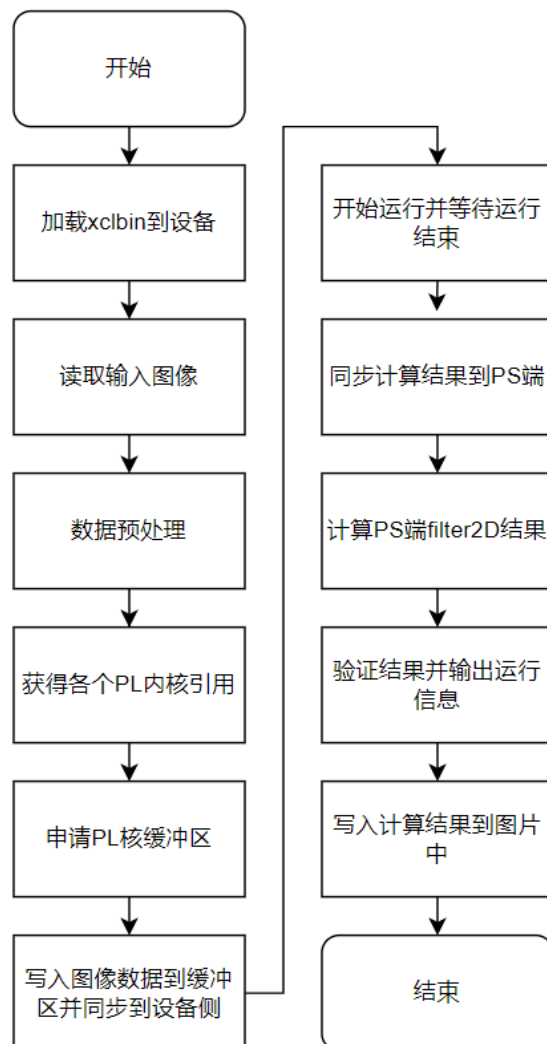


图 5-1 PS 端运行流程图

### 5.3 JPEG 图像解码

在图像解码方面，我们实现了两种方案：OpenCV 方案与 JPEG 解码 PL 内核 knr1\_jpeg 方案。

(1) OpenCV 方案：此方案的优点是调用方便，代码简短，缺点是需要依赖 OpenCV 库，会对部署造成一系列麻烦。对于此方案，我们使用 `cv::imread` 把 JPEG 读取为存储灰度数据的 Mat 格式，之后提取出所有的像素点保存到一维数组中作为输入图像。

(2) PL 解码方案：此方案使用官方的 JPEG 解码内核 knr1\_jpeg 来解码图

像，解码器可以将 JPEG 图像解码为 YUV 格式，该方案的优点是硬件利用率高、不需要依赖其他库文件。要使用此方案解码图像，首先通过 PS 端读取图像文件的二进制数据并同步到解码内核中，等待 PL 内核执行完成后，可以获得该图像的 YUV 数据以及图像信息数据，图像信息数据中包含图像大小等信息，在解码出 YUV 数据后，我们在 PS 端中提取出其中的 Y 通道作为输入图像。

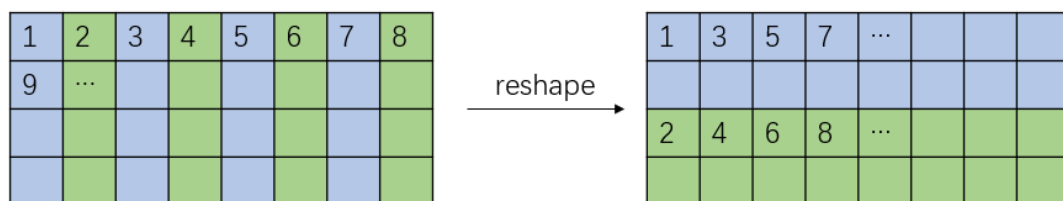
我们还使用了 OpenCV 进行运算结果的输出，若不需要输出图像，在 PS 端注释掉相关代码即可。

## 5.4 图像数据分割

分割图像数据在 PS 端中实现，分割方式图解在 AI Engine 设计部分的功能概述中进行了展示。在此方案中 PART\_NUMBER(切分块数)为 18，对应需要申请 18 个块内存来存储分割后的数据。根据图像切分方法中的描述，首先需要计算每块所存储的数据个数，第一块与最后一块存储的数据个数为  $(HEIGHT/PART\_NUMBER+1)*WIDTH$ ，中间块为  $(HEIGHT/PART\_NUMBER+2)*WIDTH$ 。计算数据量后就可以计算出每块在原始图像中的起始行，最后使用 memcpy 函数来拷贝源图中的数据到各个缓冲区中。

## 5.5 运算结果重塑与合并

在 PL 汇集 AIE 处理单元的运算结果时，两路处理单元的结果是交替保存的，需要在 PS 端进行重塑，下图 5-2 为数据重塑的操作过程，下图 5-2 中以一个 PL 内核的输出数据，数据总共有四行，每行 8 个 v8int32 作为举例。



说明:蓝色框对应一路AIE运算单元输出的数据，绿色框对应另外一路AIE运算单元输出数据。

图 5-2 数据重塑过程

在重塑操作之前，需要计算出每块输出图像在最终存储图像结果的数组中所在的起始索引地址，以便后续在重塑中从此地址开始向后输出数据。在重塑过程中，需要从 PL 内核结果中取出 16 个数据，把其中的高 8 位 int32 与低 8 位 int32 分别放到对应的输出图像起始地址中，并对每个起始地址增加 8，为下次存储做准备，最终完成重塑操作，形成输出图像。

## 5.6 信息统计

PS 端的统计信息包括：运算结果准确性、VCK5000 实际运行时间、PS 端 filter2D 运行时间、对比 PS 端加速比、每秒帧数。

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，并使用此时间结果计算加速比，帧数等信息。

运算结果对比是在 PS 端中是独立实现了 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  $O(1)$

若测试结果无误，则会输出“TEST PASSED!”。

## 5.7 本章小结

本章主要介绍了此系统在 PS 端的详细设计。首先，我们提出了 PS 端了设计目标，描述了需要完成的功能并对执行流程做了分析。其次我们对两种图像解码方式、图像数据分割、图像数据重塑合并的方法做了阐述。最后，我们总结了 PS 端的输出信息内容、计时方式、运算结果对比方法等内同。总之，通过此 PS 端设计方案，整个系统可以快速并有序地运行完毕。

## 第 6 章 系统部署

### 6.1 系统编译

在经过每个模块的变成后，就可以为其进行编译核部署。AIE 部分经 aiecompiler 编译为 libadf.a 文件，每个种类的 PL 核编译为对应.xo 文件，最后使用 V++编译器将上面的所有中间件以及 hw\_link 部分的.cfg 配置文件封装为器件二进制文件(.xclbin)<sup>[8]</sup>。

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。构建和封装嵌入式系统的流程如下图 6-1 所示：

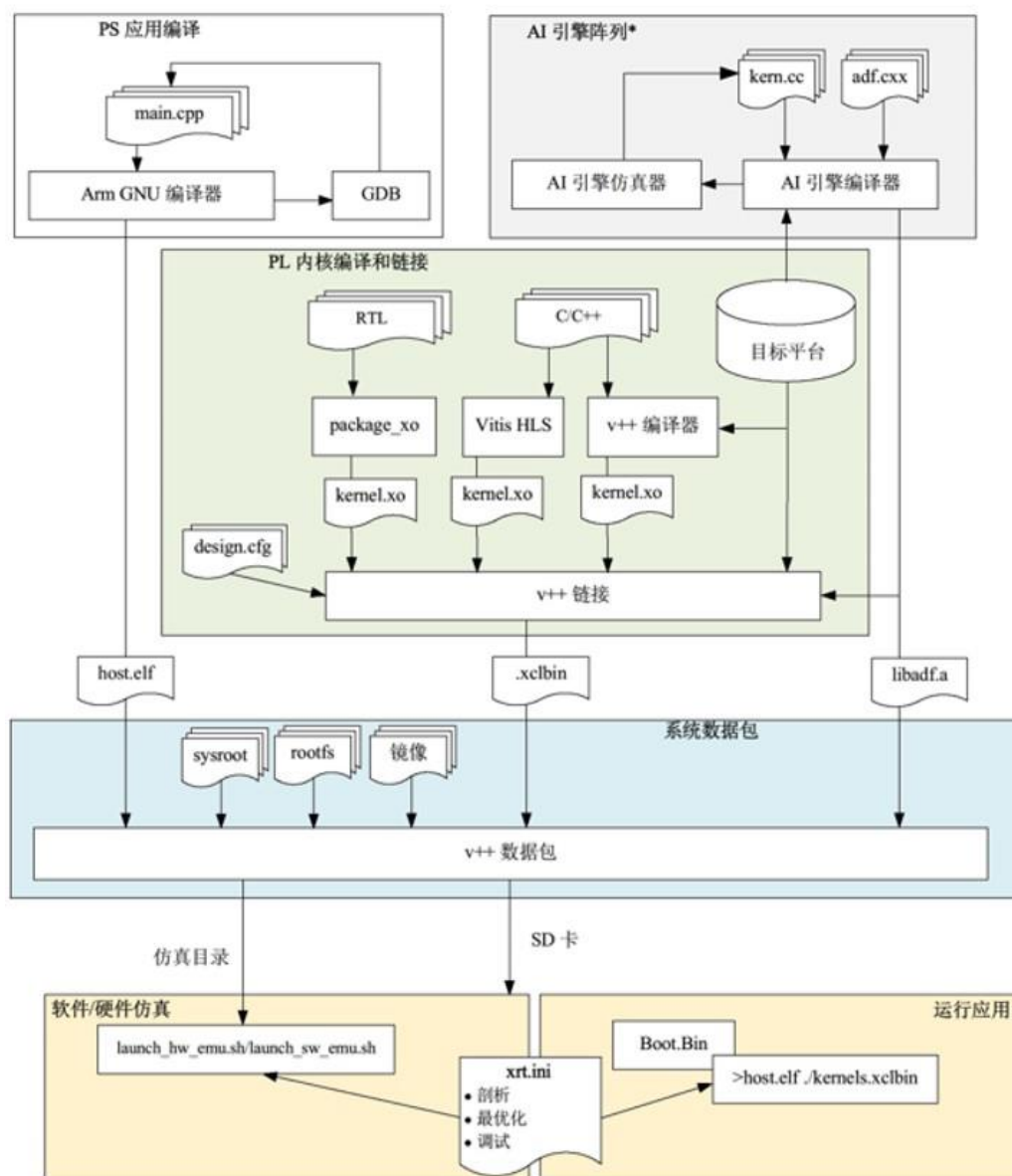


图 6-1 构建和封装嵌入式系统的流程

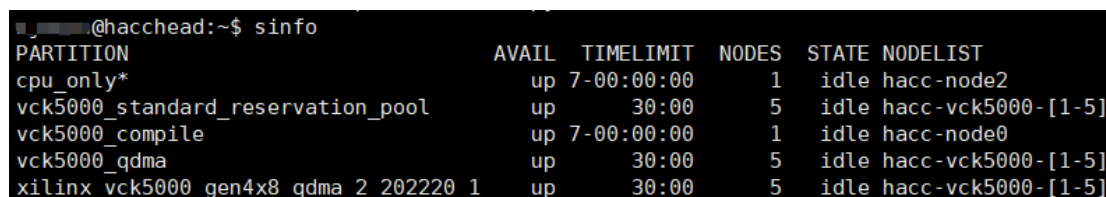
## 6.2 Makefile 文件

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。

对于 AIE 部分、PL 部分、hw\_link 部分都有对应的 Makefile 文件。在编译时使用终端进入对应目录输入 make all 即可开始编译流程。若要编译整个系统，可以先单独对 AIE 和 PL 部分进行编译，之后再使用总的 Makefile 进行编译，也可以使用根目录下的 Makefile 文件一键完成编译，编译过程大概需要 2 至 3 个小时。

## 6.3 系统运行验证

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 6-2 所示，如果节点状态是 idle 表示当前节点可用，alloc 表示节点被占用<sup>[9,10]</sup>。



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 6-2 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。

进入 VCK5000 节点后，需要使用 FTP 工具把需要的 XCLBIN、HOST 端的编译文件、源图像 image.jpg 导入到 HACC 中。在终端中输入 ./template.exe ./template.xclbin 即可开始执行项目。

在经过一段时间的执行后，可以看到运行的详细信息以及运行结果报告，并在此文件夹下输出了 output.jpg 文件。在 PS 端的输出结果中可以看到 filter2D 在 VCK5000 板卡上的运行时间、在 PS 端运行时间、结果正确性对比情况、加速比、FPS 等信息。

本次实验采用三次重复实验，计算平均加速比、平均运行时间以及平均帧率。测试所使用的图像分辨率为 7680x4320，测试数据采用 int32 类型，测试结果保留两位小数，下表 2-1 为运行结果统计：

表 6-1 运行结果统计表

运行次数	AIE 运行时间(ms)	PS 运行时间(ms)	加速比	帧率(FPS)
第一次	9.97	1376.35	137.91	100.20
第二次	9.37	1340.04	142.88	106.62
第三次	10.12	1391.26	137.43	98.78
平均	9.82	1369.22	139.41	101.87

观察三次运行结果后，每次的正确性报告均为 TEST PASSED，这证明我们的计算结果无误。并且在 8K 图像下实验取得了对比主机端平均 139.41 倍的加速比以及平均 101.87FPS 的帧率，证明我们在上文中的设计方案和思路是正确的。下图 6-3 为实际运行的输出截图。

```

Image WIDTH:7680
Image HEIGHT:4320
open the device
load the xclbin ./template.xclbin
Input JPEG file size = 4549700
Create input and output device buffers for JPEG decoder
Create runner
Read JPEG file and transfer JPEG data to device
Run krnl_jpeg kernel
Transfer YUV data to host
Transfer Info data to host
Allocate memory for the image
get Kernel Reference
calculate the required data length for each section
split the image and put into PL input buffers
allocate xrt buffers for kernels
write and synchronize data to device buffer
delete dataInput buffers
running...
run completed
allocate buffers for PL output
read PL output data to PS buffer
calculate the store data length and index for each section
reshape and merge PL output to full image
compute the filter2D result: ps_result
compare aie_result with ps_result
TEST PASSED !
VCK5000 filter2D take up 9.77466ms
PS filter2D take up 1348.27ms
Speedup ratio:137.935
Frames Per Second:102.305 FPS
write output.jpg image finish!

```

图 6-3 实机运行结果

在观察终端输出报告后，我们可以使用图像查看器观察程序输出的 JPEG 文件。下图 6-4 是使用卷积核  $\{0, -2, 0, -2, 12, -2, 0, -2, 0\}$  处理在 hacc\_demo 中自带的 sand.jpg 图像，此图像默认分辨率为 36480x2736，为了能够使用 8K 图像进行处理，我们使用图像处理工具对其进行了分辨率扩充，扩充至 7680x4320。



图 6-4 输出图像

把输出图像放大观察后可以看到，整体达到了边缘提取的效果，边缘提取是该卷积核的处理效果。以此可以验证，我们的图像解码以及卷积计算均达到了预期目标。

在测试时，我们认为把 AIE 的运行时间与 PS 端的运行时间作比较来计算加速比不是非常合理，所以我们在测试过程中想要把我们的加速方案与 Vitis Libraries 中的 filter2d 官方库方案的运行时间作比较，并计算出加速比。

但在实施的过程中我们发现 Vitis Libraries filter2d 的存储图像方式为把所有图像信息都调入 input\_window 中，并且在计算过程中使用指针来访问像素数据，没有使用数据流的方式，而 input\_window 最大能够分配的大小为 16KB，所以库中的方案最大只能支持运行共 4096 个像素点的 int32 数据类型图像进行卷积运算，无法处理 8K 分辨率图像。

虽然无法使用此方式与官方库的方案进行直接对比，但是在运行结果中除加速比之外的运行时间和帧率是两个十分有说服力的数据。

## 6.4 本章小结

本章主要介绍了此系统的部署即运行结果。首先，我们介绍了构建和封装系统的流程。其次我们 Makefile 文件以及编译方式做了阐述。最后，我们借助



HACC@NUS 完成了系统在 VCK5000 上的实际运行,并获得了运行报告及输出图像。  
总之,通过编译和运行,在统计运行结果之后,证明实际运行的效果与设计时的  
预期相符,符合设计要求。

## 第 7 章 总结与展望

### 1、比赛总结

定制计算算法实现挑战赛 (Customized Computing Challenge, CCC) 是中国计算机学会体系结构专委会针对高性能异构计算技术交流的平台。很庆幸, 今年我们团队参加了这届比赛, 让我们亲身体验到了异构计算领域的业界最新技术, 第一次尝试在结合了 AI Engine (矢量和标量引擎), 可编程逻辑 (PL) 和可编程片上网络 (NoC) 的异构平台上, 应用高层次综合技术, 高效完成从算法到硬件应用开发, 虽然从报名到决赛仅仅短短三个月, 尽管现在看来这个作品仍显拙劣, 但这次巅峰级的科研前训练确实让我们收益颇丰, 总结下来包括如下几方面:

#### a) 基于 filter2D 突出 ACAP 架构特色

Filter2D 是计算机视觉领域的经典算法, 这个算法虽然看似简洁, 但却是典型的卷积运算, 而卷积运算是 CNN、DNN 等一系列深度学习模型的基本算子, 因此该函数也是各类深度学习函数库中的基本函数, 比如在著名的 OpenCV 中就有该函数 filter2D。

通过 CCC 培训了解到 AIE 的架构特征后, 我们深切体会该架构的高效性、灵活性、低功耗特性和快速部署的优势, 在结合 filter2D 实现特征分析后, 我们体会到 AIE 架构可以充分利用矢量处理单元在一周期内一次性计算多个乘法结果, 这对于卷积的计算十分有利; AIE 阵列庞大的规模、独特的架构以及特殊的数据传输方式使其可以很好地支持图像的并行处理。

#### b) ACAP 的适用范围

VCK5000 属于 ACAP 架构的一款代表性产品, 它将标量计算、矢量计算和可编程逻辑结合在一起, 便于在其上发挥各自优势, 更高效、更低功耗地达成算法的功能目标。以我们完成 filter2D 为例, 在使用 int32 类型 8K 分辨率图像的情况下可以达到平均 101.87FPS 的帧率, 获得相比 PS 端平均 139.41 倍的加速比。由此可以预见, ACAP 未来将会在对不同类型算力均有较大需求的 AI 领域中发挥更大的作用。

然而, 通过初步使用, 我们也体会到部分资源仍存在瓶颈, 比如 NoC 资源不足导致处理单元不能很好地扩大规模、单 PL 负责多路性能不稳定等, 这让我们不得不将设计调整为使用 18 路并行处理, PL\_SEND 核心一对一服务、PL\_RECEIVE 核心一对二服务的方案。从软硬件协同设计的角度来看, 若能总结出某类掘金应用的关键特征, 部分硬件设计可以做出新的改进; 反之, 若能对硬件逐层抽象描述, 提升编程效率, 乃至实现系统设计自动化。由此可见, 站在硬件设计的角度对现有 AI 算法进行重新审视, 进一步优化高层次综合的研究势在必行。

### c) VCK5000 开发的难点

工具链问题：对软件工程师来说仍显得不够友好，似乎在针对类似 ML 的软硬件协同设计工作中，对于底层硬件的理解和知识最终决定了其最终代码实现质量，这意味着软件工程师往往需要跟硬件工程师协同工作才可以写出高质量的代码，这无疑拖累了工作效率。且在开发过程中，AIE、PL 和 PS 端的编程语言和编程模式都不尽相同，在多种编程模式之间频繁切换具有一定的开发和调试难度。

## 2、未来工作展望

通过参与此次 CCC 竞赛，我们领略到 Versal ACAP，特别是 AIE 架构的优势。同时也坚定了我们在其上展开进一步研究的信念，下一步我们试图展开的工作包括如下三方面：

### a) 基于 Versal ACAP 的 filter2D 深度优化

此次竞赛让我们小试牛刀，体会到了 VCK5000 的威力，然而站在产品化的角度来看，目前的作品还略显稚嫩。目前的作品可以借助于 OpenCV 库达成较为完美的实现；在我们尝试采用 Vitis Vision Library 加载图片时，对于不同类别图片的适应性处理仍显欠缺。比如，目前我们仅针对 YUV(4:4:4) 的图片进行了处理下一步还可以进一步拓展。另外，目前我们的处理仅停留在灰度图，下一步可以如法炮制拓展到 RGB 领域。

同时，针对 filter2D 函数来说，目前对于 PL 端的性能优化挖掘度不够，虽然结合 AIE 做过一些高性能瓶颈测试，但性能还不够稳定，下一步仍需要在提升鲁棒性方面开展进一步研究和探索。

### b) 针对 Versal ACAP 异构系统设计自动化水平提升的研究

此次竞赛让我们初步领略到 VCK5000 的难点，AIE、PL 和 PS 分属不同的开发框架，如何优化以实现异构系统设计自动化有待进一步探索。从设计工具链角度来看，目前 Vitis 开发平台虽然已经有了很大的提升，但对比 CUDA 来说仍有较大的距离，对底层硬件架构了解较少的人员来说，目前的编程环境仍有较大的难度，这无疑会大大限制该架构在相关领域的推广应用。

### c) 面向 Versal ACAP 架构的定制化软硬件优化设计的研究

通过此次比赛了解到，VCK5000 可以部署在云端作为性能评测、设计优化的评估平台，以此指导对软硬件资源进行定制化设计。虽然，此次比赛我们只尝试了 CV 领域中的常用函数 filter2D，但其设计思想可以有效推广至各类卷积类深度学习模型的优化设计实现中。目前卷积神经网络为代表的深度模型已应用在包括 CV 和 NLP 在内的很多应用中，并已形成了若干深度学习库函数，未来可以考虑将本竞赛中探索的方法应用在各类深度学习库函数经典算子的优化设计中，提升针对不同类型应用的定制化优化设计的效率。

### 3、致谢

非常感谢 CCC2023 组委会为此次比赛做出的辛勤努力!从大赛宣讲阶段的高水平技术培训,到后期耐心的答疑和技术指导,让我们在短短几个月中从技术小白提升成为初学者,也让我们初次领略到异构系统开发的效率,并立志投身相关的科研工作。期待以后能有机会继续向组委会和与会专家学习!

## 参考文献

- [1] AMD/Xilinx. Versal ACAP AI Engine Architecture Manual (AM009)
- [2] AMD/Xilinx. AI Engine Tools and Flows User Guide (UG1076)
- [3] AMD/Xilinx. AI Engine Intrinsic User Guide
- [4] AMD/Xilinx. AI Engine Kernel and Graph Programming Guide (UG1079)
- [5] AMD/Xilinx. Vivado Design Suite User Guide High-Level Synthesis (UG902)
- [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