

# Lab1 设计一个 FIR 滤波器分离鸟类声音

张文瑞 2011881

## 一、实验目标

- 1、如何使用 Vitis HLS 构建一个项目
- 2、在 Vitis HLS 中进行仿真、综合与 IP 导出
- 3、使用 Vivado 对 HLS 导出的 IP 进行集成
- 4、使用 PYNQ 构建一个简单的应用

## 二、实验环境

- 1、PYNQ-Z2 远程实验室服务或物理板卡
- 2、Vitis HLS
- 3、Vivado

## 三、实验过程遇到的问题

注 1：由于 VitisHLS 和 Vivado 部分的具体实验步骤在指导文件中进行了详细阐述，因此不做重复说明，该部分内容主要基于听课中的困惑及实操中遇到的各类报错、解决方式。

注 2：为方便区分具体内容便于观看，规定如下：

◆查阅及引用资料内容——黑色

◆报错问题说明——红色

◆解决方案——蓝色

### 【查阅资料】

#### 1、Vitis HLS 相关工作机制

Vitis HLS 是一个高级综合工具。用户可以通过该工具直接将 C、C++编写的函数翻译成 HDL 硬件描述语言，最终再映射成 FPGA 内部的 LUT、DSP 资源以及 RAM 资源等。用户通过 Vitis HLS，使用 C/C++代码来开发 RTL IP 核，可以缩短整个 FPGA 项目的开发和验证时间。

Vitis HLS 工作主要分为两个阶段，第一个阶段为调度和控制逻辑的提取；第二个阶段为捆绑映射。

调度（Scheduling），调度主要完成的任务是判定每个时钟周期要完成哪些操作、每个操作又需要多少个时钟周期来完成、以及调度等工作。

控制逻辑的提取（Control Logic Extraction），该步骤主要是生成状态机。

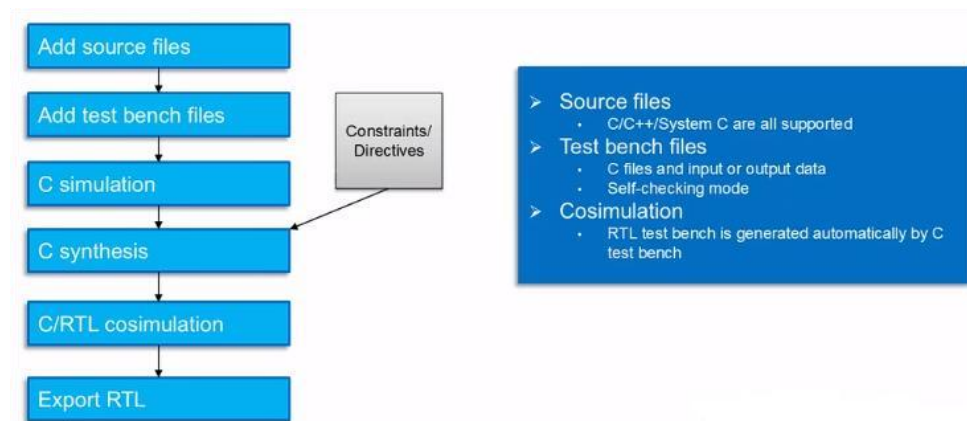
捆绑映射（Binding），判定每个操作需要什么资源来实验，完成资源的过程。

#### 2、基于 Vitis HLS 的 IP 设计流程

首先，我们采用 C/C++语言来描述算法以及 Test Bench，同时我们还要准备 Constraints 和 Directives。Vitis HLS 有专门的图形化界面来设置 Directives。以上这些组成了整个设计的输入。值得一提的是，Vitis HLS 也集成和提供了 C 代码库。这些库涵盖了算数运算、视频处理、信号/数据处理、线性代数等。开发人员可以直接调用这些库函数来加速自己 C 算法的描述。

随后，通过 Vitis HLS 平台将以上设计输出为 VHDL/Verilog 代码。开发人员并不直接使用这些代码，而是将这些代码封装成 IP 核，然后将 IP 核添加到 Vivado 的 IP Catalog 中进行调

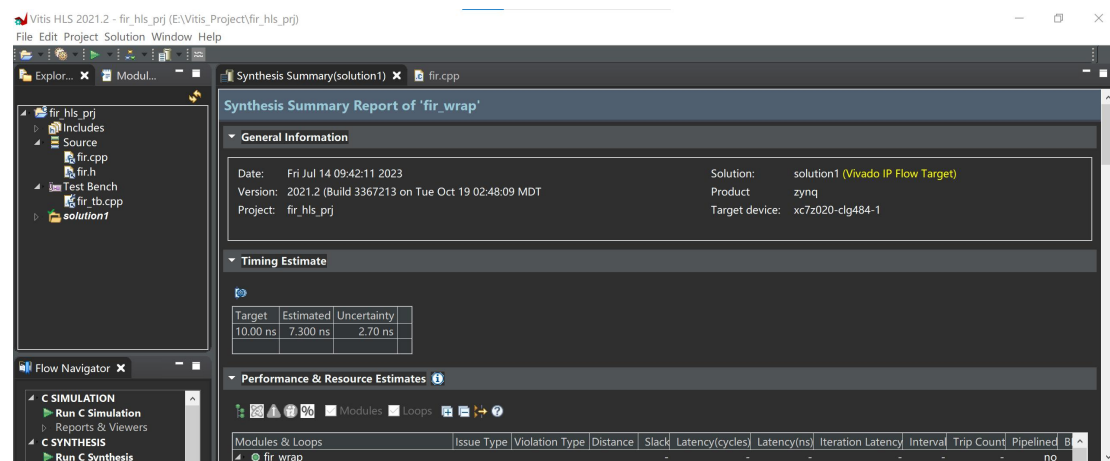
用。这也和 Vivado 提出的以 IP 为核心的设计理念是一致的。



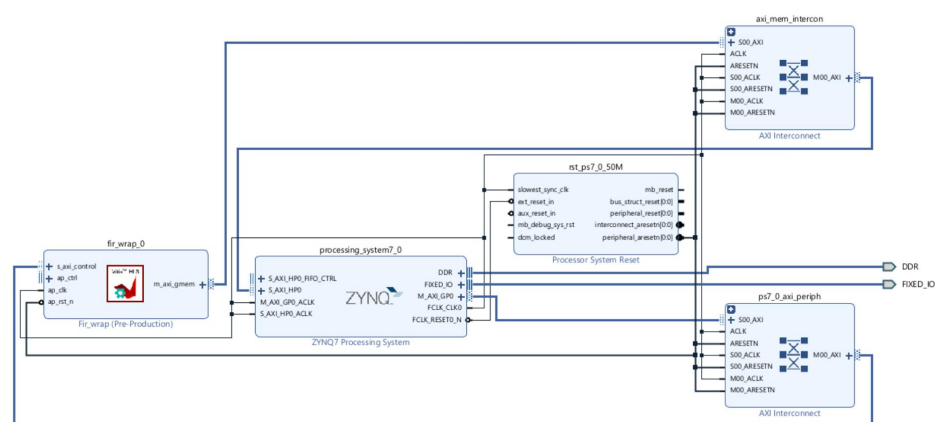
通过 Vitis 的联合仿真（Cosimulation）功能，C/C++语言描述的 Test Bench 可以自动转换为 RTL 的 Test Bench。

### 【实验重要步骤】

#### 1、Vitis HLS——C Synthesis 结果



#### 2、Vivado——block design



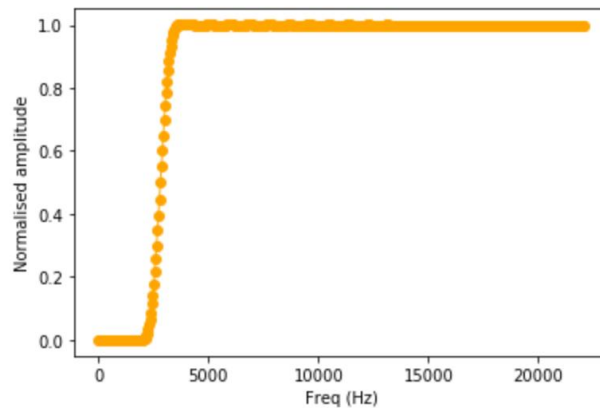
#### 3、jupyter notebook 部分可视化结果

---

```
In [7]: import matplotlib.pyplot as plt
```

```
plt.plot(sample_freqs, abs(resp), linewidth=1, color="orange", marker="o")
plt.xlabel("Freq (Hz)")
plt.ylabel("Normalised amplitude")
```

```
Out[7]: Text(0, 0.5, 'Normalised amplitude')
```



---

#### 【程序报错】

1、构建 Vitis HLS 工程,进行 C simulation 时,提示报错:**[HLS 200-1023]Part '-' is not installed.**

**Solution:** 经在 <https://support.xilinx.com> 平台查阅解决方案及咨询助教老师后,第一次确认该报错的直接原因是板卡不匹配。在构建工程时由于没有找到 xc7z020clg484-1 板卡,因此我选择 xc7z020clg400-1 为代替先行实验,实验无法进行。因此第二次开始考虑软件版本原因,经官网查阅后,最终确认目前下载的 Vitis HLS 版本(2020.3)并不支持 ZYNQ-Z2 板卡。重新下载后,后续实验全部由 2021.2 版本进行。

2、Vitis HLS,提示报错:**ERROR: [IMPL 213-28] Failed to generate IP.**

**Solution:** 自 2022 年 1 月 1 日起, Vivado HLS 和 Vitis HLS 使用的 export\_ip 命令将无法导出 IP。在后台使用 HLS 的 Vivado 和 Vitis 工具也会受到此问题的影响。HLS 工具以 YYYYMMDDHHMM 格式设置 ip\_version, 此值作为有符号整数(32 位)进行访问,这会导致溢出并生成下面的错误(或类似错误)。Xilinx 建议所有客户应用此补丁以确保安全。(下载补丁: y2k22\_patch)

## 四、实验收获与总结

- 1、基本掌握 Vitis HLS、Vivado、jupyter notebook 的使用方法。
- 2、了解了如何使用 Vitis HLS 构建一个项目,在 Vitis HLS 中进行仿真、综合与 IP 导出,使用 Vivado 对 HLS 导出的 IP 进行集成
- 3、学会使用 PYNQ 构建一个简单的应用

## Lab2 使用 Sobel 算子提取 Lena 的边缘实验目标

张文瑞 2011881

### 五、实验目标

- 1、在 Vitis HLS 中创建使用 AXI Stream 接口的 Sobel IP
- 2、在 Vivado 中构建包含 DMA 的 IP 集成
- 3、在 PYNQ 中学习 DMA 等接口的使用
- 4、在 PYNQ 中构建一个高效的 Sobel 图像处理应用

### 六、实验环境

- 1、PYNQ-Z2 远程实验室服务或物理板卡
- 2、Vitis HLS
- 3、Vivado

### 七、实验过程遇到的问题

#### 【查阅资料】

#### 1、sobel 算子

在边缘检测中，常用的一种模板是 Sobel 算子。Sobel 算子有两个，一个是检测水平边缘的；另一个是检测垂直边缘的。与 Prewitt 算子相比，Sobel 算子对于像素的位置的影响做了加权，可以降低边缘模糊程度，因此效果更好。

Sobel 算子另一种形式是各向同性 Sobel(Isotropic Sobel)算子，也有两个，一个是检测水平边缘的，另一个是检测垂直边缘的。各向同性 Sobel 算子和普通 Sobel 算子相比，它的位置加权系数更为准确，在检测不同方向的边沿时梯度的幅度一致。将 Sobel 算子矩阵中的所有 2 改为根号 2，就能得到各向同性 Sobel 的矩阵。

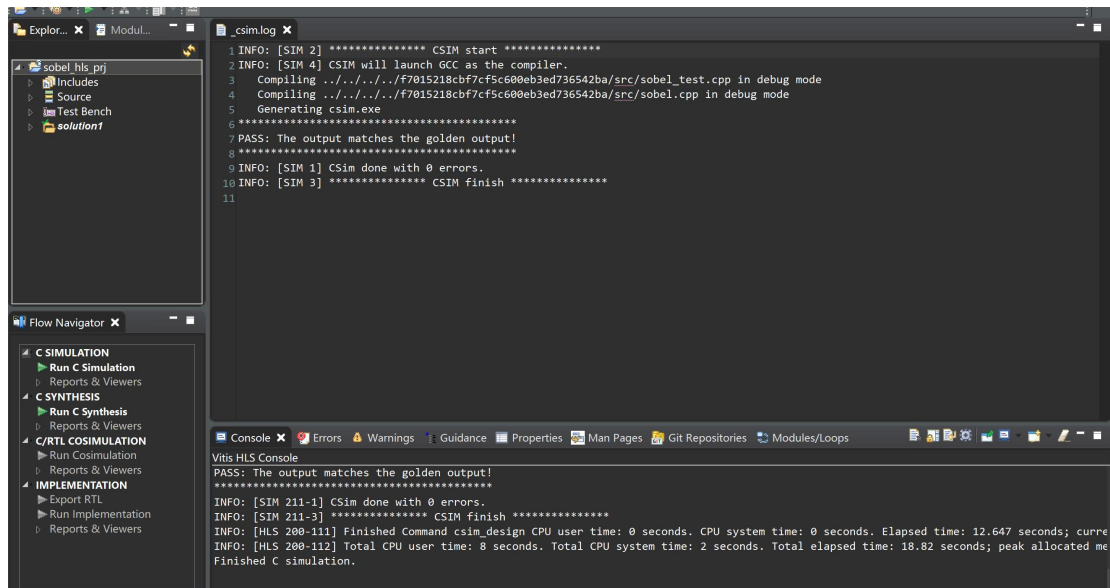
由于 Sobel 算子是滤波算子的形式，用于提取边缘，可以利用快速卷积函数，简单有效，因此应用广泛。美中不足的是，Sobel 算子并没有将图像的主体与背景严格地区分开来，换言之就是 Sobel 算子没有基于图像灰度进行处理，由于 Sobel 算子没有严格地模拟人的视觉生理特征，所以提取的图像轮廓有时并不能令人满意。在观测一幅图像的时候，我们往往首先注意的是图像与背景不同的部分，正是这个部分将主体突出显示，基于该理论，我们给出了下面阈值化轮廓提取算法，该算法已在数学上证明当像素点满足正态分布时所求解是最优的。

#### 2、常用算子比较

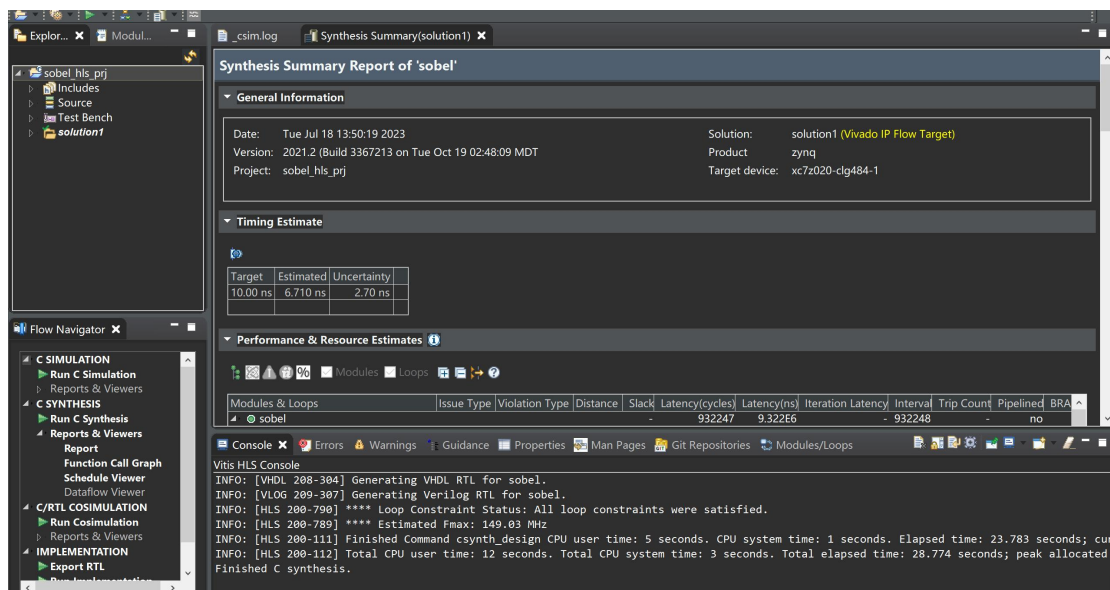
算子	优缺点比较
Roberts	对具有陡峭的低噪声的图像处理效果较好，但利用 Roberts 算子提取边缘的结果是边缘比较粗，因此边缘定位不是很准确。
Sobel	对灰度渐变和噪声较多的图像处理效果比较好，Sobel 算子对边缘定位比较准确。
Kirsch	对灰度渐变和噪声较多的图像处理效果较好。
Prewitt	对灰度渐变和噪声较多的图像处理效果较好。
Laplacian	对图像中的阶跃性边缘点定位准确，对噪声非常敏感，丢失一部分边缘的方向信息，造成一些不连续的检测边缘。
LoG	LoG 算子经常出现双边边缘像素边界，而且该检测方法对噪声比较敏感，所以很少用 LoG 算子检测边缘，而是用来判断边缘像素是位于图像的明区还是暗区。
Canny	此方法不容易受噪声的干扰，能够检测到真正的弱边缘。在 edge 函数中，最有效的边缘检测方法是 Canny 方法。该方法的优点在于使用两种不同的阈值分别检测强边缘和弱边缘，并且仅当弱边缘与强边缘相连时，才将弱边缘包含在输出图像中。因此，这种方法不容易被噪声“填充”，跟容易检测出真正的弱边缘。

## 【实验重要步骤】

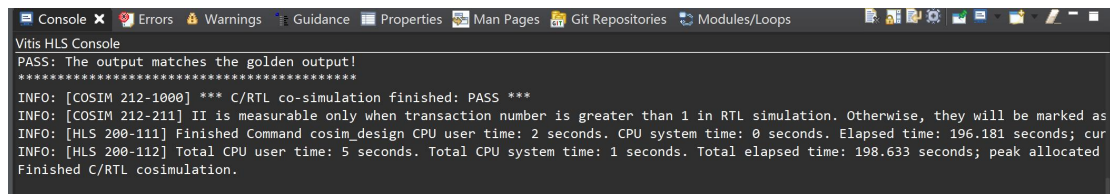
### 1、Vitis HLS——C simulation



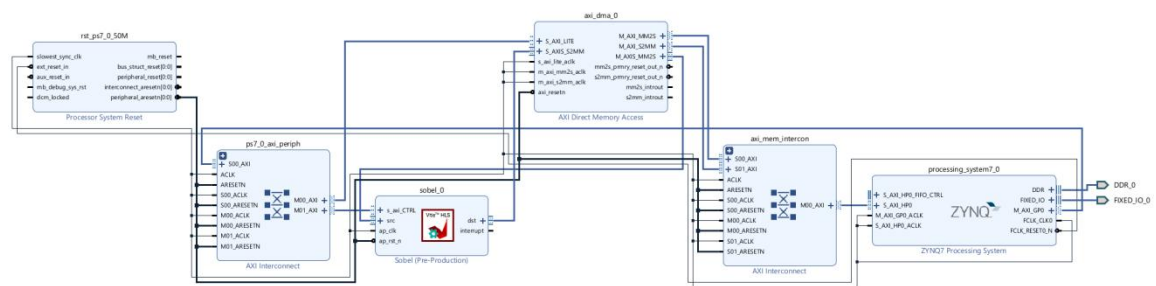
### 2、Vitis HLS——C synthesis



### 3、Vitis HLS——C/RTL Co-simulation



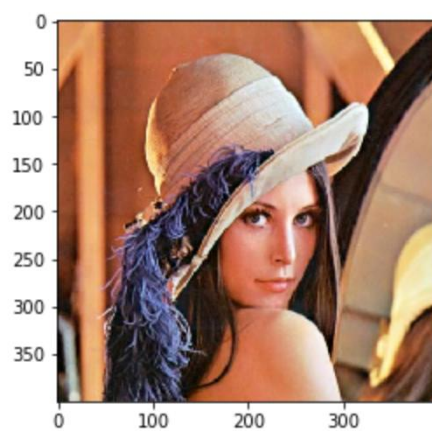
### 4、Vivado——block design



#### 4、jupyter notebook 部分可视化结果

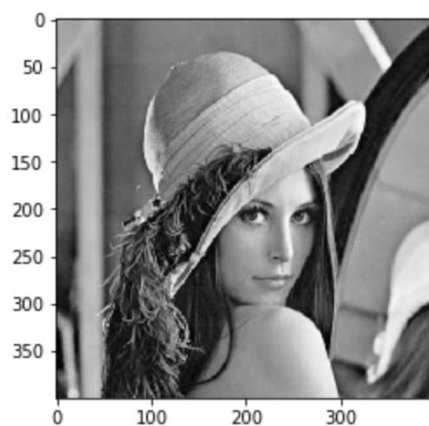
```
In [3]: plt.imshow(img[:, :, ::-1])
```

```
Out[3]: <matplotlib.image.AxesImage at 0xa2811be0>
```



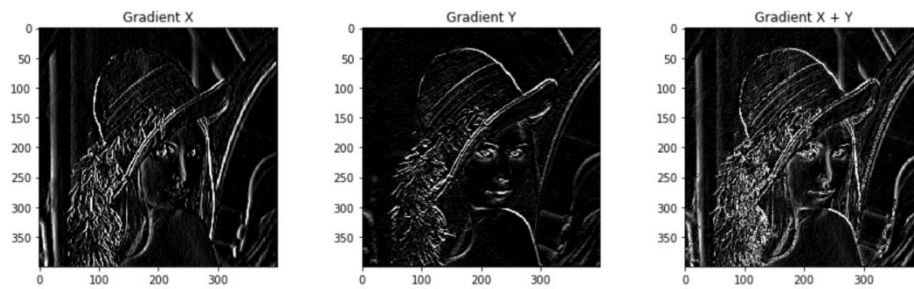
```
In [5]: plt.imshow(gray, cmap='gray')
```

```
Out[5]: <matplotlib.image.AxesImage at 0xa2635418>
```



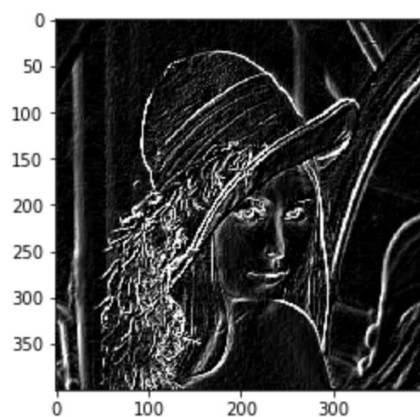


Out[7]: <matplotlib.image.AxesImage at 0xa05eb088>



```
In [15]: plt.imshow(output_buffer.reshape(rows, cols), cmap='gray')
```

Out[15]: <matplotlib.image.AxesImage at 0x9fd19f88>



## 八、实验收获与总结

- 1、对 sobel 算法及其应用有了一定的理解。
- 2、学会在 Vitis HLS 中创建使用 AXI Stream 接口的 Sobel IP
- 3、学会在 Vivado 中构建包含 DMA 的 IP 集成
- 4、学会在 PYNQ 中学习 DMA 等接口的使用在 PYNQ 中构建一个高效的 Sobel 图像处理应用
- 5、学会利用 Python 语言实现一些结果的可视化语句。

## Lab3 MNIST 分类器

张文瑞 2011881

### 一、实验目标

- 1、如何使用 Vitis HLS 构建一个项目
- 2、在 Vitis HLS 中进行仿真、综合与 IP 导出
- 3、使用 Vivado 对 HLS 导出的 IP 进行集成
- 4、使用 PYNQ 构建一个简单的应用

### 二、实验环境

- 1、PYNQ-Z2 远程实验室服务或物理板卡
- 2、Vitis HLS
- 3、Vivado

### 三、实验过程遇到的问题

#### 【查阅资料】

vitis\_ai 是基于 fpga 中的 dpu（深度学习处理单元）统一的全栈式 ai 推断解决方案，其主要包含 3 个主要部分：

- 1、vitis\_ai model zoo，支持 tf、caffe 和 pytorch，包括自动驾驶、监控和金融等领域的已训练好的模型，可用于快速概念验证和生产。
- 2、vitis\_ai development kit，提供了 5 种工具用于部署网络。5 种工具如下：
  - ai 优化器：通过 fine\_turn 的方法减少模型计算量，可以提高 5-20 倍计算速度。
  - ai 量化器（vai\_q\_tensorflow/caffe）：  
将 float32 的模型转化为 int8 模型，甚至更低精度。  
执行层融合和其他硬件友好型策略，加速网络推断效率。
  - ai 编译器（vai\_c\_tensorflow/caffe，可支持 model zoo 和自定义模型）：  
生成在 dpu ip 上运行的高效代码  
提供 api，实现机器学习预处理和调用 dpu 进行网络前向推断。
  - ai 检测器：用于分析 ai 性能，集成到 vitis analyzer 中
  - ai library
- 3、deeplearning processing unit（dpu）：高效执行前向计算。不同类型的 dpu 处理不同网络，如 cnn/lstm/bert/mlp 等。

#### 【实验步骤】

- 1、使用 Vitis AI 激活 TensorFlow 2.x 的环境  
`conda activate vitis-ai-tensorflow2`
- 2、（可选）训练模型  
float\_model.h5 是一个训练好的模型，你也可以选择自己训练  
`cd SummerSchool-Vitis-AI`  
`python train.py`
- 3、量化  
`./1_quantize.sh`



脚本中调用了 `vitis_ai_tf2_quantize.py`, 使用 `python` 的 API 进行量化:

首先加载模型并创建量化器对象

```
float_model=tf.keras.models.load_model(args.model)quantizer=vitis_quantize.VitisQuantize
r(float_model)
```

#### 4、加载数据集用于模型校准 (Calibration)

```
(train_img, train_label), (test_img, test_label) = mnist.load_data()test_img =
test_img.reshape(-1, 28, 28, 1) / 255
```

5、量化模型, 需指定用作校准的数据集 (`calib_dataset` 参数), 可以使用部分的训练集或测试集, 通常 100 ~ 1000 个就够了

```
quantized_model = quantizer.quantize_model(calib_dataset=test_img)
```

#### 6、量化完之后模型依旧被保存为 .h5 格式

```
quantized_model.save(os.path.join(args.output, args.name+'.h5'))
```

#### 7、编译

```
./2_compile.sh
```

脚本使用 `vai_c_tensorflow2` 命令进行模型的编译, 需指定以下参数:

--model 量化之后的模型

--arch 指定 DPU 架构, 每个板卡都不一样, 可以在  
`/opt/vitis_ai/compiler/arch/DPUCZDX8G` 目录下寻找

--output\_dir 输出目录

--net\_name 模型的名字

输出的 .xmodel 文件被保存在 `compile_output` 目录下

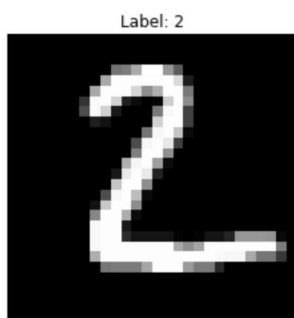
使用 DPU-PYNQ 部署模型

使用 PYNQ 2.7 或 3.0.1 版本的镜像启动板卡

```
$ sudo pip3 install pynq-dpu --no-build-isolation
```

#### 8、部分可视化结果


```
plt.imshow(test_data[1, :, :, 0], 'gray')
plt.title('Label: {}'.format(test_label[1]))
plt.axis('off')
plt.show()
```



```
plt.tight_layout()
for i in range(num_pics):
    image[0,...] = test_data[i]
    job_id = dpu.execute_async(input_data, output_data)
    dpu.wait(job_id)
    temp = [j.reshape(1, outputSize) for j in output_data]
    softmax = calculate_softmax(temp[0][0])
    prediction = softmax.argmax()

    ax[i].set_title('Prediction: {}'.format(prediction))
    ax[i].axis('off')
    ax[i].imshow(test_data[i,:, :, 0], 'gray')
```

Prediction: 7 Prediction: 2 Prediction: 1 Prediction: 0 Prediction: 4 Prediction: 1 Prediction: 4 Prediction: 9 Prediction: 5 Prediction: 9



```
stop = time()
correct = np.sum(predictions==test_label)
execution_time = stop-start
print("Overall accuracy: {}".format(correct/total))
print(" Execution time: {:.4f}s".format(execution_time))
print(" Throughput: {:.4f}FPS".format(total/execution_time))
```

Classifying 10000 digit pictures ...

Overall accuracy: 0.9871

Execution time: 3.6281s

Throughput: 2756.2394FPS

## 四、实验收获与总结

1、MNIST 数据集（Mixed National Institute of Standards and Technology database）是美国国家标准与技术研究院收集整理的大型手写数字数据集，包含了 60,000 个样本的训练集以及 10,000 个样本的测试集。

MNIST 中所有样本都会将原本 28\*28 的灰度图转换为长度为 784 的一维向量作为输入，其中每个元素分别对应了灰度图中的灰度值。MNIST 使用一个长度为 10 的 one-hot 向量作为该样本所对应的标签，其中向量索引值对应了该样本以该索引为结果的预测概率。