密级：

TinyML 逻辑

详细设计

编写：_____

校对：_____

审核：_____

批准：_____

易灵思科技有限公司

2022-9-20

修订记录

日期	修订版本	描述	作者
2022-9-20	V1.0	添加 EVSOC 模块概要设计	Tate Li

目录

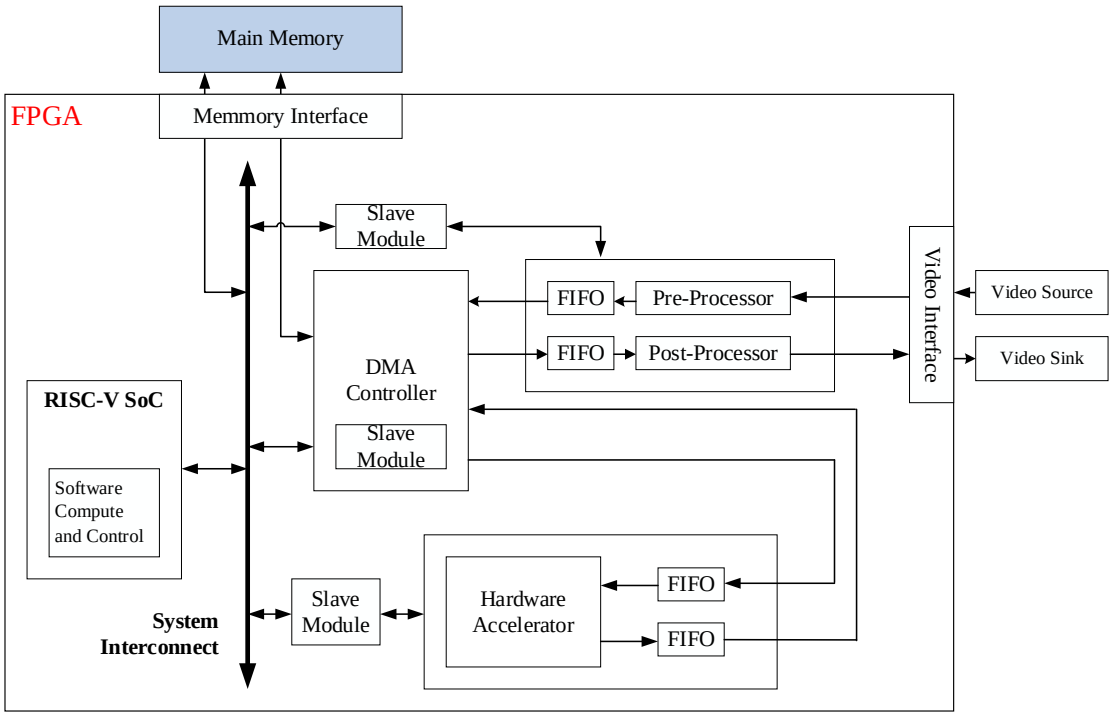
图目录

表目录

1 产品特性简介

人工智能（AI）和边缘视觉是一个新兴领域，涵盖了汽车、工业视觉、零售和机器人等广泛的应用。易灵思的 Edge Vision SoC 框架使用 RISC-V SoC 软核和硬件相互结合的方式来实现，包含了特定的嵌入式软件功能、硬件加速器模块、AI 算法加速器模块、存储和 I/O 相关接口、外围设备和控制逻辑。其 Edge Vision SoC 框架如图所示。

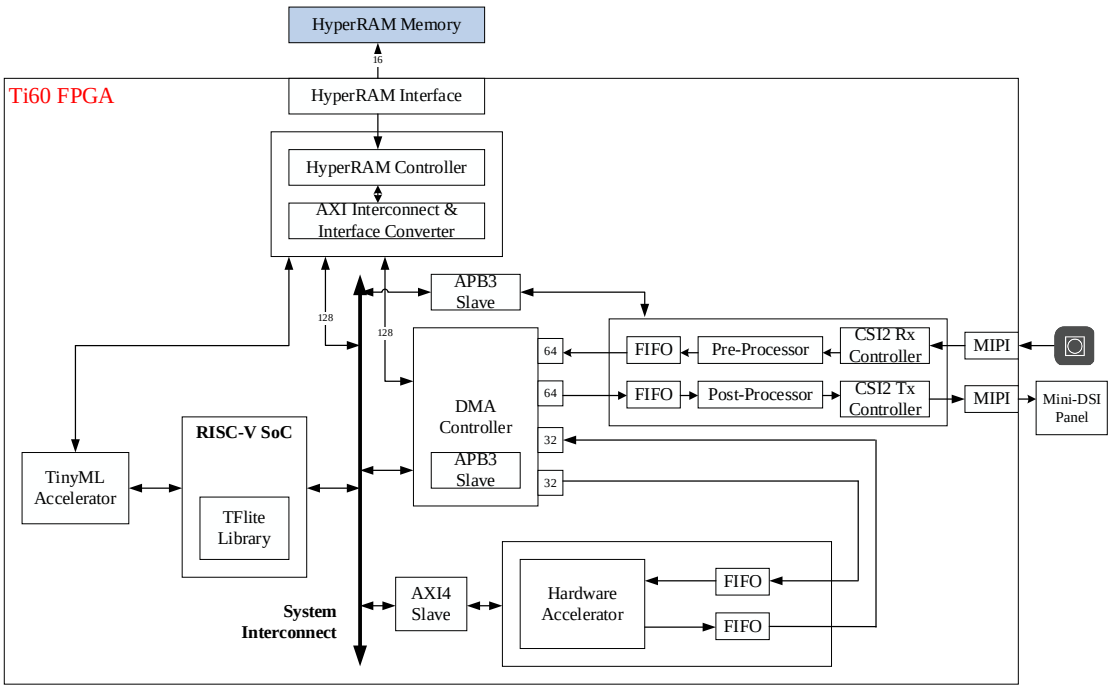
在此框架内，易灵思选择为某些特定功能提供了示例设计，例如视频处理、AI 对象检测、机器学习、多摄像机融合等。你可以自由地使用 Edge Vision SoC 框架编写 C/C++代码，构建不同的加速算法或者 AI 模型，或者你也可以完全使用自己的定制插件。



本用户指南主要描述了如何使用以 Edge Vision SoC 框架构建的示例设计。这些设计说明了本框架的用例，特别是视频处理的硬件/软件协同处理设计。此外，本设计还显示了如何使用软件控制 FPGA 硬件，也就是说，您可以通过更改 RISC-V 处理器中的固件来启用不同的加速功能。

2 设计框架

如下图，针对易灵思 Titanium Ti60 F225 开发板所设计的图像信号处理实例设计框图如下。本次设计主要依靠易灵思的 Edge Vision SoC 框架，在本次设计中添加了 TinyML Accelerator，主要用于对视频流的 AI 加速算法。



本次设计使用 Raspberry Pi V2 相机模块作为视频流的输入来源，然后将视频流经 MIPI 接口传输到 CSI-2 协议控制器里。此时预处理模块（Pre-Processor）可根据 RISC-V 处理器的配置对视频流的帧率、分辨率、像素格式和是否转为灰度像素做出处理。最后再将处理过后的数据经过 DMA 存储到外部的 Hyper RAM 里。

而在发送方向，通过 DMA 从外部的 Hyper RAM 里读取视频数据送入到后处理模块（Post-Processor）的缓存 FIFO 里。此时后处理模块根据 RISC-V 处理器的配置信息对视频流做出分辨率和帧率的调整处理以便于能够在 Mini-DSI 面板上成功显示，然后再将该数据流转换为 CSI-2 协议控制器所需要的格式，最终通过 MIPI 接口传输到外部的显示单元上。

在本设计中，Hardware Accelerator 和 TinyML Accelerator 都是实现加速的两个功能模块。其中 Hardware Accelerator 主要实现对视频数据的格式处理，例如可以实现灰度转换、二进制腐蚀（binary erosion）和二进制膨胀（binary dilation）等功能；而 TinyML Accelerator 主要实现对视频数据的 AI 算法加速，例如算法中常出现的卷积、加法、乘法和大小比较等运算，体现在对数据的数学运算上面。两个加速功能都是将相机输入的视频流作出处理后再存入外部 Hyper RAM 中，也都可以在 RISC-V 处理器上运行。

下表显示了本次设计在 Ti60 上使用的逻辑资源分布情况

Block	**Flipflops**	**Adders**	**LUTs**	**Memory Blocks (M10K)**	**DSP**
Complete Design	19246	2673	28265	211	4
RISC-V SoC	6196	702	7301	59	4
DMA Controller	6162	745	8,393	67	0
CSI-2 RX Controller Core	871	60	2149	15	0
DSI TX Controller Core	1574	111	4506	21	0
HyperRAM Controller	1738	263	2601	18	0
Camera	814	618	1039	11	0

Display	489	0	716	12	0
Hardware Accelerator	630	162	606	8	0
TinyML Accelerator					

2.1 Sapphire RISC-V SoC

本设计实例支持高度灵活的硬件和软件的协同设计方法，因此您可以选择是在 RISC-V 处理器中执行算法运算，还是在硬件中执行算法。RISC-V SoC 可用于执行整体系统控制并执行固定的或者灵活多变的的算法。

在本设计实例的框架中，其包含的 Sapphire Vision RISC-V SoC 是 Sapphire SoC 标准版本的变体。详情请参阅《Sapphire RISC-V SoC Data Sheet》规格数据手册。

2.2 DMA Controller

DMA 控制器主要实现便于外部存储器和本设计中的其他模块之间的通信功能。实现了将数据帧存储到外部存储器中、向 Hardware Acceleration 发送和接收数据，并向 Post-Processing 发送数据。DMA 控制器对每个端口都有专用的、固定方向的通道。

DMA 控制器有 direct 模式和 scatter and gather (SG)模式。在这两种模式中，RISC-V 处理器使用轮询(默认)或中断的方式来确定 DMA 传输是否完成。

•轮询——RISC-V 处理器检查 BUSY 信号，直到其被取消断言。

•中断——RISC-V 处理器可以在 DMA 传输过程中执行其他操作，当它接收到一个中断时代表传输完成。

在本次 RTL 设计中， DMA 控制器连接到外部 256 位的 DDR DRAM 接口。

2.3 HyperRAM Controller

Hyper RAM 控制器主要实现了对片外存储芯片（Hyper RAM）的的读写管理，其一端为 AXI4 接口，一端为 Hyper RAM 的管脚。可以支持三个 master 端同时对 Hyper RAM 的读写操作。

2.4 Video Capture and Pre-Processing

本次设计使用 Raspberry Pi V2 相机模块作为系统图像数据的输入来源。视频从摄像机传输到 MIPI CSY-2 RX 控制器内核（Titanium），分辨率为 1920*1080，RAW10 格式（每个原始像素 10 位），每个时钟 4 个像素。帧速率根据相机驱动程序设置而变化。

Pre-Preprocessor 接收来自相机的视频流并执行以下功能：

•根据您在 RISC-V 固件中选择的设置调整 RGB 增益。

•将每个时钟的像素数从 4 更改为 2。

•将 RAW 转换为 RGB.

•裁剪或缩放视频以进行后续处理。

•可选择将 RGB 像素数据转换为灰度像素数据。

对于 Titanium Ti60 F225 开发板设计，输入视频分辨率被裁剪为 1080*1080，然后使用 nearest-neighbor 法缩小到 540*540。

完成这些功能后，Pre-Preprocessor 通过 DMA 控制器将数据存储到外部存

储器中。如果 RISC-V 处理器上运行的固件指定将 RGB 或灰度数据存储在外部存储器中。此时您可以选择在硬件中实现灰度转换，或者在 RISC-V 处理器中执行该功能。

2.5 Post-Processing and Display

Post-Processor 根据最终的显示器分辨率生成视频信号（HSYNC、VSYNC 和 VALID），并从显示器 DMA FIFO 检索像素数据（红色、蓝色和绿色）。对于 Trion 开发板设计，视频信号和像素数据通过 LVDS 接口打包和输出。对于 Titanium Ti60 F225 开发板设计，视频信号和像素数据被传递到 DSI TX 控制器内核，便于在 Mini-DSI 面板上显示。

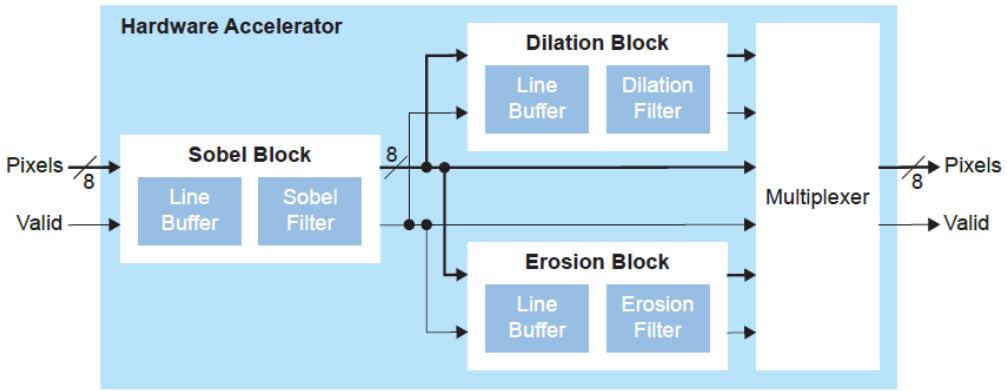
2.6 Hardware Accelerator

本次设计在硬件加速器中提供了几个标准功能，这些功能也可以使用软件代码来实现。在默认情况下，本设计演示了硬件/软件协同设计：RISC-V 处理器作为嵌入式软件执行 RGB 到灰度的转换，而硬件加速器执行 Sobel 边缘检测、二进制腐蚀（binary erosion）和二进制膨胀（binary dilation）。

- Sobel filter：执行边缘检测。
- Binary erosion：通过对所有窗口像素进行“与”运算来删除线条细节。
- Binary dilation：通过对所有窗口像素进行“或”运算来增强线条细节。

硬件加速器模块在流水线流式体系结构中实现，并以三种模式运行：Sobel、扩展 Sobel 或侵蚀 Sobel。您可以在 RISC-V 处理器上运行的固件中指定模式。

Figure 9: Hardware Acceration Block Diagram

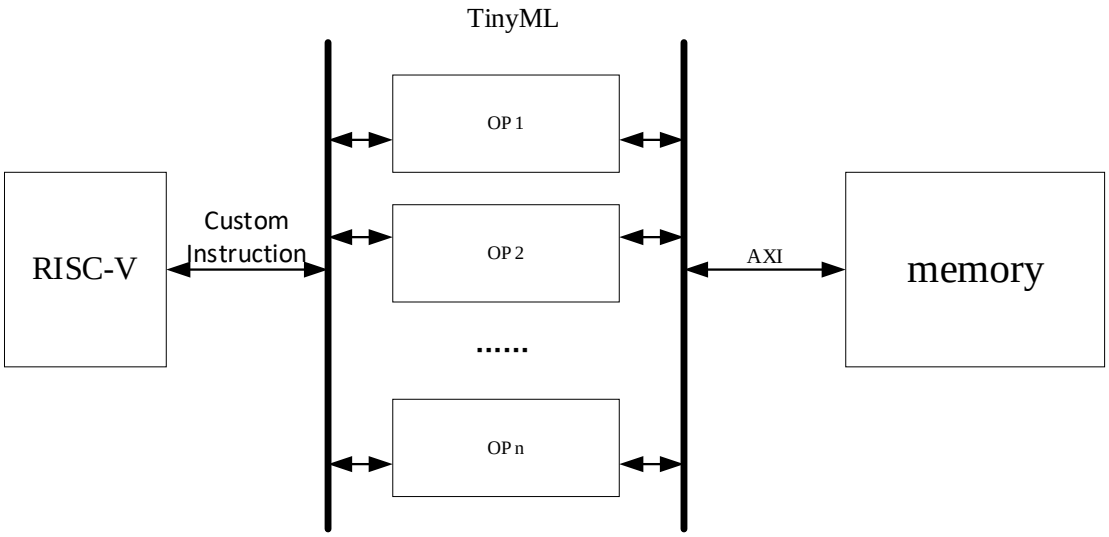


在 RTL 设计中，标准包装器包含加速功能、输入和输出 FIFO，以及与连接到 RISC-V 处理器的 AXI4 从机模块接口的调试和控制寄存器。FIFO 和控制逻辑从外部存储器获取输入数据，并通过 DMA 控制器将输出数据存储到外部存储器。此包装器旨在提供一个标准接口，以便您可以轻松地放入自己的加速函数。

2.7 TinyML Accelerator

微型机器学习 Tiny Machine Learning (TinyML) 是机器学习的一个研究领域，专注于在超低功耗的微控制器设备上开发和部署机器学习模型。TinyML 使机器学习可以在在安全、低延迟、低功耗和低带宽的边缘设备上运行。本次设计提供了几个在 TinyML 上的实际加速应用，例如 face_landmark、

yolo_person、resnet_image_classification、mobilenetv1_person_detection 和 ds_cnn_keyword_spotting。你可以灵活地选择自己所需要的加速应用。



在 RTL 设计中，TinyML 加速器主要是根据这些加速应用在软件上耗时较长的操作转变而来，将其由软件实现变为硬件实现以此加速。故 TinyML 加速器主要由几个加速操作模块（OP）构成，其一端接着 RISC-V 处理器的 custom instruction 接口，一端使用 AXI4 接口与外部的存储空间 Hyper RAM 相连。你也可以轻松的使用这些标准接口来加入自己的加速算法。

在本设计中，主要实现了以下几个加速操作：分别为 Convolution & Depthwise、ADD、MAXIMUM（比较最大最小）、MULT 和 FC（Fully Connected）。可以通过修改对应的参数使能来打开或者关闭这些功能（注：当你在软件上关掉此功能时，在 RTL 代码中应该也关掉该功能），每一个操作模块内又有单独的参数可以来调整模块的性能与资源使用情况，以此来适应你的设计。

2.8 固件选择

当你将基本的视频流框架搭建好以后，就可以开始定制软件了。在 RISC-V 处理器上可以选择或者定制自己将要实现的功能。在本设计中提供了两个加速模块以供你使用，分别为 TinyML Accelerator 和 Hardware Accelerator。你可以在软件代码中的 main.c 文件里去使用它。该文件中包含了注释代码，您可以打开它使其执行各种功能。这些功能可以再软件里实现或者使用硬件实现。在你修改 main.c 文件以后，则必须重新编译软件并将新固件文件（.elf）下载到主板。下面将介绍本次设计中的一些预设功能。

2.8.1 Hardware Accelerator

此节将叙述如何在开发板上运行不同的固件。在硬件加速器模块里定义了三个不同功能的固件，你可以通过修改软件主程序（main.c）来选择运行的固件，具体实现方式可参考《edge-vision-soc-ug-v3.0.pdf》。

2.8.2 TinyML Accelerator

对于 TinyML 加速器的固件选择，在 RTL 设计上，只需要修改文件 define.v 里的参数就可以了。其中具体的参数含义可以参考 TinyML 模块接口信号定义。

```
`define AXI_DW  128
`define CONV_DEPTH_MODE "STANDARD"
`define CONV_DEPTH_STD_IN_PARALLEL  8
`define CONV_DEPTH_STD_OUT_PARALLEL 4
`define CONV_DEPTH_STD_OUT_CH_FIFO_A    1404
`define CONV_DEPTH_STD_FILTER_FIFO_A    375
`define CONV_DEPTH_STD_CNT_DTH  1404
`define CONV_DEPTH_LITE_PARALLEL    4
`define CONV_DEPTH_LITE_AW  3
`define ADD_MODE    "STANDARD"
`define FC_MODE "LITE"
`define FC_MAX_IN_NODE  64
`define FC_MAX_OUT_NODE 12
`define MULT_MODE   "LITE"
`define MIN_MAX_MODE    "LITE"
```

而在软件程序上，你需要修改 define.h 文件里参数，使得软件和硬件使用的功能能够对应上。或者你也可以使用易灵思提供的工具（tinyml_参数生成器）来生成两个 define 文件，然后再重新编译工程。

2.9 使用您自己的加速器

在本实例中有一个执行过滤功能的硬件加速器（Hardware Accelerator）和执行 AI 算法的加速器（TinyML Accelerator）。

其硬件加速器的顶层文件里（hw_accel_wrapper.v）包含了硬件加速器、FIFO、一些控制逻辑和调试寄存器。本顶层具有连接到 DMA Controller 和 AXI4 slave 的接口。

AXI4 slave 接口用于 RISC-V 处理器和硬件加速器之间的通信，主要用于调试和控制的功能。虽然本设计使用了 AXI4 slave 接口，但您也可以在顶层中更改为使用其它 slave 接口（如 APB3 或 AXI4-lite）。

顶层文件中的控制逻辑主要是为了产生 FIFO 和 DMA 读/写相关信号，这些读写信号根据硬件加速器的行为（例如，对数据流的读写请求、DMA 传输的固件设置等）来产生。

AI 算法的加速器的顶层文件里（tinyml_accelerator.v）拥有 AXI4 master 接口和 custom instruction 接口，其中 AXI4 接口是用来连接 DMA 控制器的，以便于从存储空间里读写数据；而 custom instruction 则是用来连接 RISC-V 处理器，可以接收相关参数，命令和数据等。

如果您想加入自己的自定义加速器，则需要对顶层文件和 RISC-V 的固件进行调整。以下各节介绍了您应该考虑的方面。

2.9.1 接口修改

首先决定要使用哪种类型的接口，例如 AXI4、APB3 或 AXI4-lite。接下来，确定 RISC-V 处理器和加速器之间需要什么通信来对控制和调试寄存器进行设置。

例如，您可以使用 RISC-V 处理器触发计算和握手的开始，设置不同的操作模式，或探测关键信号以进行调试。最后，根据您的设计更新的加速器顶层中

的控制逻辑和调试寄存器。

2.9.2 DMA Controller 修改

第一步是确定输入和输出数据要求。DMA 控制器使用 AXI-ST 接口连接到硬件加速器。默认情况下，该接口具有 32bit 宽度，可用于从片外存储器中读写数据流。

DMA 控制器支持各个通道的异步模式。如果您需要以不同的时钟速率读写数据流，请将相应 DMA 控制器通道道的时钟信号连接正确。（参见第 33 页的 DMA 控制器。）

2.9.3 FIFO 修改

根据自定义加速器的需要来确定 FIFO 要求（数据宽度、深度、模式等）和读/写行为，并相应地修改 FIFO。接下来，检查 FIFO 下溢和上溢，并相应地调整 FIFO 控制信号或 DMA 传输设置。

2.9.4 RISC-V 修改

当你修改了本设计的 RTL 代码后，您必须更新 RISC-V 固件以匹配您的 RTL 更改。例如，当地址偏移量增加后则需要去修改 axi4_hw_accel.h 文件中的内容。

如果您使用不同的 slave 接口，请在 main.c 中添加相应的标题以便于访问。请参阅 main.c 中的 Camera Capture、HW Accelerator、Check AXI4 slave Status（HW Accessor）和 Check APB3 slave States（Camera&Display）部分的 APB3 和 AXI4 从机接口的使用。

RISC-V Soc 使用易灵思的 IP 生成，可以在生成时选择自己需要的接口和运行频率，当使用自定义加速器时，应更新固件中的 DMA 传输设置。例如，根据加速器的行为更新每个捕获帧的 DMA 传输长度。（dmasg_direct_start（）的第三个参数是以字节数表示的传输长度。）默认情况下，示例设计使用 direct 模式 DMA 传输。

3 TinyML Accelerator 详细设计

3.1 TinyML 框图设计

下图为 TinyML 模块系统框图：

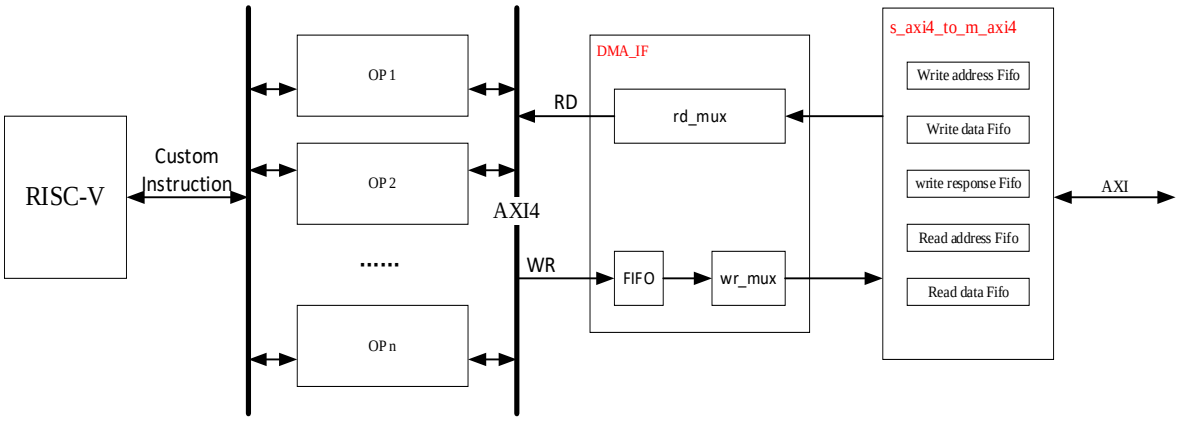


图 1 TinyML 系统框图

如上图所示 TinyML 功能模块通过 Custom Instruction 与 RISC-V 之间进行通信，通过 DMA 方式与 HyperRAM 之间进行通信。各子模块功能描述如下：

1) OP：各个加速算子。直接参与运算的加速单元。
2) DMA IF：完成各个 OP 模块之间的读写调度与缓存。
3) s_axi4_to_m_axi4：主要完成对 AXI4 接口的数据通道进行字节寻址和时钟域的转换。

3.1.1 OP 一级模块功能描述

在本设计中，OP 模块主要实现了以下几个加速操作：分别为 Convolution & Depthwise、ADD、MIN_ MAX（比较最大最小）、MULT 和 FC（Fully Connected）。其中所有加速操作都有两种模式，分为 lite 模式和 standard 模式。两者不可共存，处于互斥状态，用户可根据需要进行参数配置来决定使用那种模式。

Lite 模式：仅使用 Custom Instruction 接口与 RISC-V 通信，数据直接与 RISC-V 进行交互，不直接对 HyperRAM 进行读写操作。此模式下资源占用率低，但是性能不足。

standard 模式：使用 Custom Instruction 接口接收来自 RISC-V 的命令，然后使用 DMA 方式来读写 HyperRAM 以提高性能。此模式由于直接对数据进行读取，不再经过 RISC-V 传输，所以性能提升较大，但是资源耗费较多。

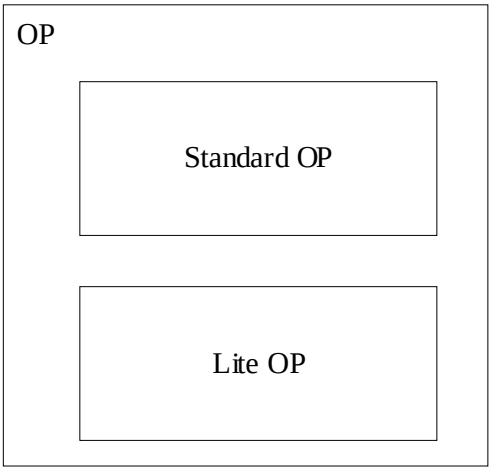


图 2 OP 结构框图

3.1.1.1 Lite OP 模块功能描述

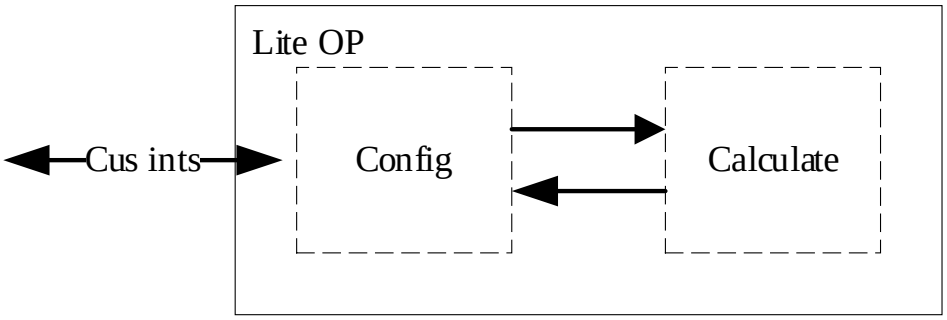


图 3 Lite OP 结构框图

Lite 模式下，OP 模块主要分为两个功能点：Config 和 Calculate。

Config：主要实现对模块的参数配置；

Calculate：实现加速算法的完整运算流程。

3.1.1.2 Standard OP 模块功能描述

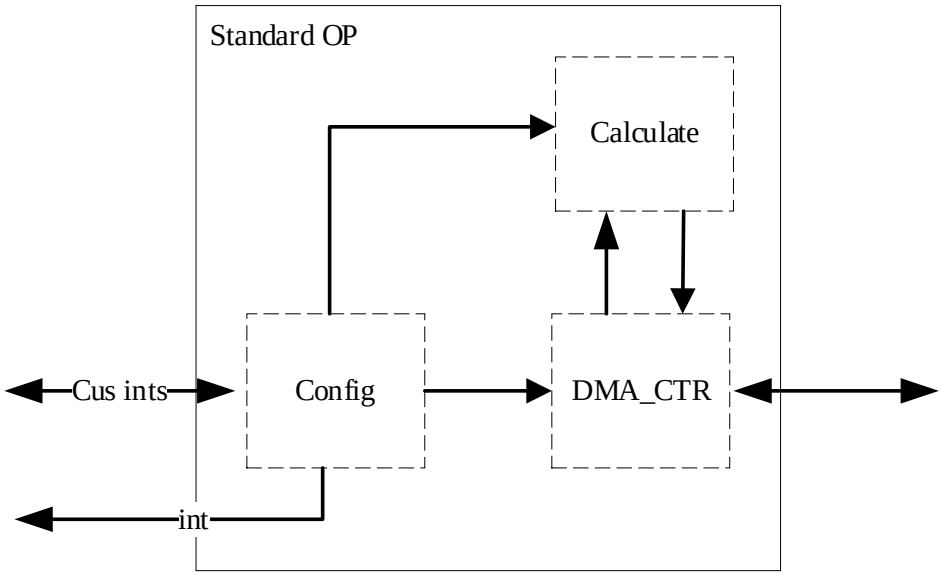


图 4 Standard OP 结构框图

Standard 模式与 lite 模式最主要的区别在于多了一个 DMA_CTR 功能子模块，该功能模块主要实现对 RISC-V 命令的解析，然后发出 DMA 读写命令的状态机。该功能模块会通过中断的方式来告诉 RISC-V 命令的执行情况。

3.1.2 dma_if 一级模块功能描述

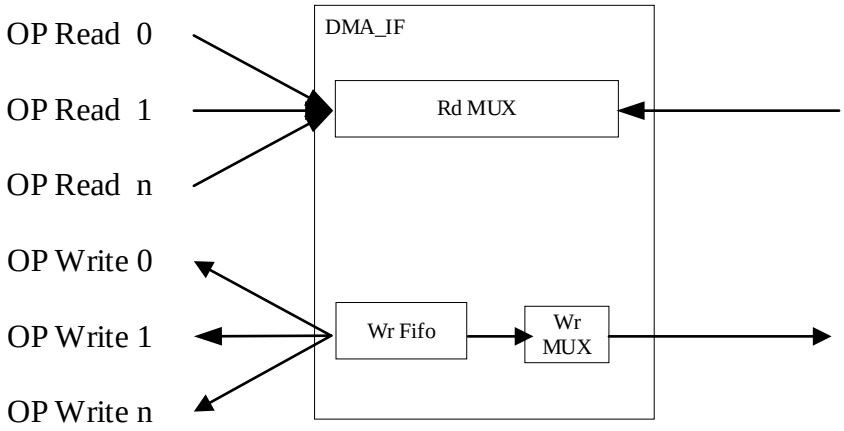


图 5 dma_if 结构框图

dma_if 模块主要实现两个功能：一是接收来自各个 OP 模块的 DMA 读写请求，当有两个及以上的 OP 模块时，需要对这些读写请求进行调度处理。同一时刻不会有两个 OP 模块同时发起读写请求。二是对读写命令的缓存并将其转换为 AXI4 接口的读写请求。

3.1.3 s_axi4_to_m_axi4 一级模块功能描述

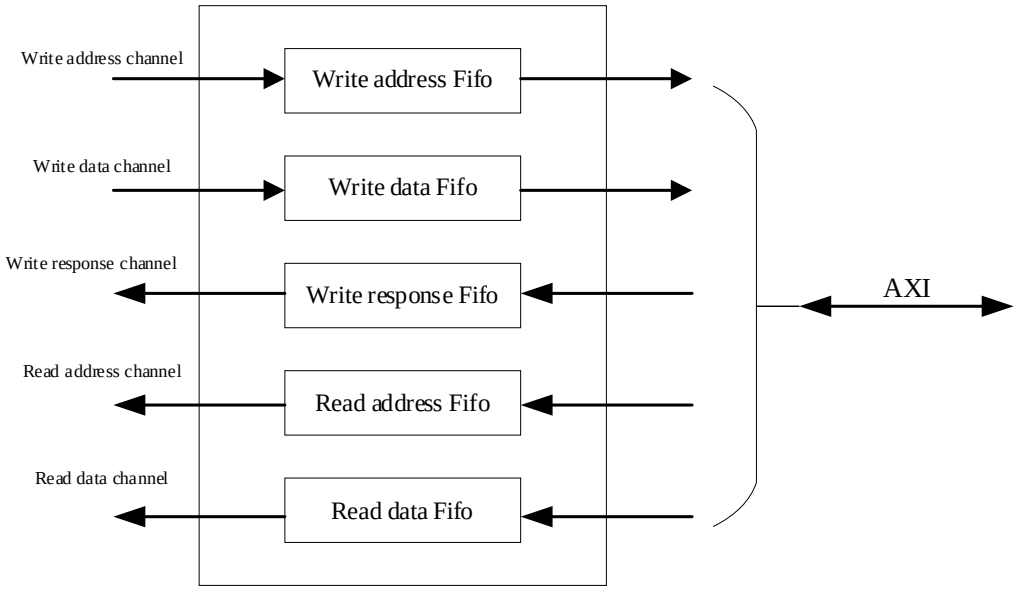


图 6 s_axi4_to_m_axi4 结构框图

本模块主要完成对 AXI4 接口上的数据通道字节寻址，由于数据通道位宽为 128bit，但是读写时可能不会同时对 128bit 的数据进行读写，故需要进行字节寻址。5 个 FIFO 分别对应了 AXI4 的 5 个通道，这些 FIFO 都是异步 FIFO，也可以进行时钟域的转换。

3.2 TinyML 模块参数定义

表 1TinyML 一级模块常量定义

parameter	值	描述
AXI_DW	128	AXI4 接口的数据位宽
ADD_MODE	LITE	ADD 加速算子模式选择开关；

		DISABLE：关闭整个 ADD 加速功能。 LITE：使用 lite 模式。 STANDARD：使用 standard 模式。
MIN_MAX_MODE	LITE	MIN_ MAX 加速算子模式选择开关；与 ADD 含义一致
MULT_MODE	LITE	MULT 加速算子模式选择开关；与 ADD 含义一致
FC_MODE	LITE	FC 加速算子模式选择开关；与 ADD 含义一致
CONV_DEPTH_MODE	LITE	Convolution 加速算子模式选择开关；与 ADD 含义一致
CONV_DEPTH_LITE_PARALLEL	4	Lite 模式下的 Convolution 加速算子并行通道数量
CONV_DEPTH_LITE_AW	7	Lite 模式下的 Convolution 加速算子并行通道数量宽度
CONV_DEPTH_STD_IN_PARALLEL	8	STANDARD 模式下的 Convolution 加速算子输入通道数量设置
CONV_DEPTH_STD_OUT_PARALLEL	4	STANDARD 模式下的 Convolution 加速算子输出通道数量设置
CONV_DEPTH_STD_OUT_CH_FIFO_A	1404	STANDARD 模式下的 Convolution 加速算子参数，不可更改，由参数生成器生成
CONV_DEPTH_STD_FILTER_FIFO_A	288	STANDARD 模式下的 Convolution 加速算子参数，不可更改，由参数生成器生成
CONV_DEPTH_STD_CNT_DTH	1404	STANDARD 模式下的 Convolution 加速算子参数，不可更改，由参数生成器生成
FC_MAX_IN_NODE	640	Lite 模式下的 FC 加速算子参数，不可更改，由参数生成器生成
FC_MAX_OUT_NODE	640	Lite 模式下的 FC 加速算子参数，不可更改，由参数生成器生成

3.3 TinyML 模块接口信号定义

表 2 TinyML 一级模块接口信号定义

信号名	方向	描述
全局信号		
clk	Input	全局时钟
reset	Input	全局复位
Coustom Instruction 总线		
cmd _ready	Output	接收数据 ready 信号，高有效
cmd _valid	Input	接收数据 valid 信号，高有效
cmd _function_id	Input	区分数据信号和参数信号的标志 ID
cmd _inputs_0	Input	参与计算对的数据 X 或者参数值
cmd _inputs_1	Input	参与计算对的数据 Y 或者参数值
cmd_int	Output	输出的中断信号
rsp_ready	Input	回应通道的 ready 信号

rsp_valid	Output	输出值的有效信号，高有效
rsp_outputs_0	Output	最终计算输出的值
AXI4 总线		
Write address channel		
m_axi_awready	input	从设备已准备好接收地址和相关的控制信号
m_axi_awvalid	output	主设备给出的地址和相关控制信号有效
m_axi_awaddr	output	写突发操作中第一次数据传输的地址
m_axi_awlen	output	标识每次突发传输的传输次数
m_axi_awsize	output	每次突发传输的大小
m_axi_awburst	output	突发类型，包括突发类型和突发大小信息，该字段决定了每次突发传输时地址的计算方法
m_axi_awprot	output	保护类型
m_axi_awlock	output	锁定类型，提供关于传输时原子特性的额外信息
m_axi_awcache	output	存储器类型
Write data channel		
m_axi_wready,	input	从设备已准备好接收数据选通信号
m_axi_wvalid	output	主设备给出的数据和字节选通信号有效
m_axi_wdata	output	写出的数据
m_axi_wstrb	output	数据的字节选通，数据中每 8bit 对应这里的 1bit
m_axi_wlast	output	主设备给出的数据和字节选通信号有效
Write response channel		
m_axi_bready	output	主设备已准备好接收写响应信号
m_axi_bvalid	input	从设备给出的写响应信号有效
m_axi_bresp	input	写响应，该信号表示写传输的状态
Read address channel		
m_axi_arvalid	output	主设备给出的地址和相关控制信号有效
m_axi_arready	input	主设备给出的地址和相关控制信号有效
m_axi_arcache,	output	存储器类型
m_axi_arlock	output	锁定类型，提供关于传输时原子特性的额外信息
m_axi_arprot	output	保护类型
m_axi_arburst	output	突发类型，包括突发类型和突发大小信息，该字段决定了每次突发传输时地址的计算方法
m_axi_arsize	output	表示每次突发传输的大小
m_axi_arlen	output	标识每次突发传输的传输次数
m_axi_araddr	output	读突发操作中第一次数据传输的地址
Read data channel		
m_axi_rready	output	主设备已准备好接收读取的数据和响应信息
m_axi_rvalid	input	从设备给出的数据和响应信息有效
m_axi_rdata	input	存储器类型
m_axi_rlast	input	读响应，这信号表示读传输的状态
m_axi_rresp	input	读响应，这信号表示读传输的状态

4 环境搭建

4.1 运行环境安装

4.1.1 安装 Efinity®软件

如果从未使用过我们的产品，请先从支持中心下载 Efinity®软件并进行安装。要编译本用户指南中所描述的设计，您必须拥有 2021.1.165.4.10 或更高版本的 Efinity®软件（注意：请不要在安装路径里使用中文或者空格）。

4.1.2 Install the Edge Vision SoC

4.1.3 Install the RISC-V SDK

4.1.4 Install the Java JRE

请执行以下操作安装 JRE：

1. 从下载适用于您的操作系统的 64 位版本的 JRE 或 JDK
   https://www.java.com/en/download/manual.jsp（Java 8 官方版本）
   https://developers.redhat.com/products/openjdk/download（OpenJDK 8 或 11）
   http://jdk.java.net/16/（OpenJDK 16）
2. 按照网站上的安装说明安装 JRE。

4.2 硬件安装

4.2.1 硬件组件需求

本示例设计使用了 Titanium Ti60 F225 开发板工具包中的以下硬件：

- Titanium Ti60 F225 Development Board
- Mini-DSI 面板连接器子卡 * 1
- 双 MIPI 到 DSI 转换器子卡* 1
- Raspberry Pi 相机接口子卡*1
- Raspberry Pi v2 摄像头模块* 1
- Mini-DSI 面板* 1
- 15 针扁平电缆*1
- 30 针扁平电缆*1
- USB type-C 电缆*1
- 通用 AC-DC 电源适配器
- 跳线

您还需要自己提供以下硬件：

- 安装了 Efinity®软件的计算机

4.2.2 硬件组装步骤

硬件安装步骤请参考《ti60_evsoc_hw_setup.pdf》

4.2.3 安装 USB 驱动

4.2.4 下载程序到开发板

要使用 Edge Vision SoC 框架的示例设计，必须将 RTL 设计编程到板中。

1. 生成 Edge Vision SoC 框架的 bit 文件有两种方法：
- 打开要使用的工程（.xml），查看并在 Efinity®软件中编译生成。
- 使用位于 efinity_project 目录中已编译好的 edge_vision_soc.hex 文件。

注：提供的 bit 流文件是为 SPI Active 模式编译的。有关为其他模式生成 bit 文件的详细信息，请参阅《Efinity 软件用户指南》。

2. 使用 Efinity®Programmer 和 SPI Active 模式将比特流文件下载到开发板上。如果使用 SPI Active 模式，则您需要复位 FPGA。
3. 按下开发板上的 CRESET 按钮来复位 FPGA。此复位可确保在用户应用程序运行之前进行外部内存初始化。

4.3 Eclipse 和 OpenOCD 软件的使用

您可以使用 Eclipse 来管理软件工程，使用 OpenOCD 来进行调试。以下各节介绍如何设置 Eclipse 环境、创建工程和编译工程。此外，本节还将指导您如何设置 OpenOCD 调试器，以便与 Edge Vision SoC 配合使用。

RISC-V SDK 包含 run_eclipse.bat 文件（Windows）或 run_eclipse.sh 文件（Linux），它将可执行文件添加到路径中，为 Edge Vision BSP 设置环境变量，并启动 Eclipse。始终使用此可执行文件启动 Eclipse；不要直接启动 Eclipse。

Eclipse 软件使用步骤和方法请参考《riscv-ruby-ug-v1.2.pdf》

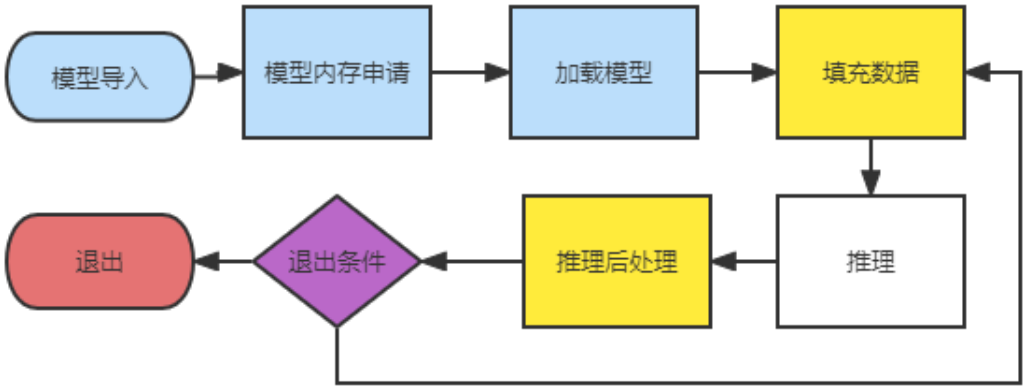
4.4 创建 TinyML 工程

TinyML 示例工程已经在附件中包含，您可以使用 Risc-V SDK 进行导入，下面是一份文件目录：

文件	说明
tools	Tinyml 模型解析和参数化工具
tinyml_vision	示例工程
model_zoo	模型训练示例
Ti60F225 开发板	用于执行 TinyML 工程

4.4.1 TinyML 开发流程和工程结构

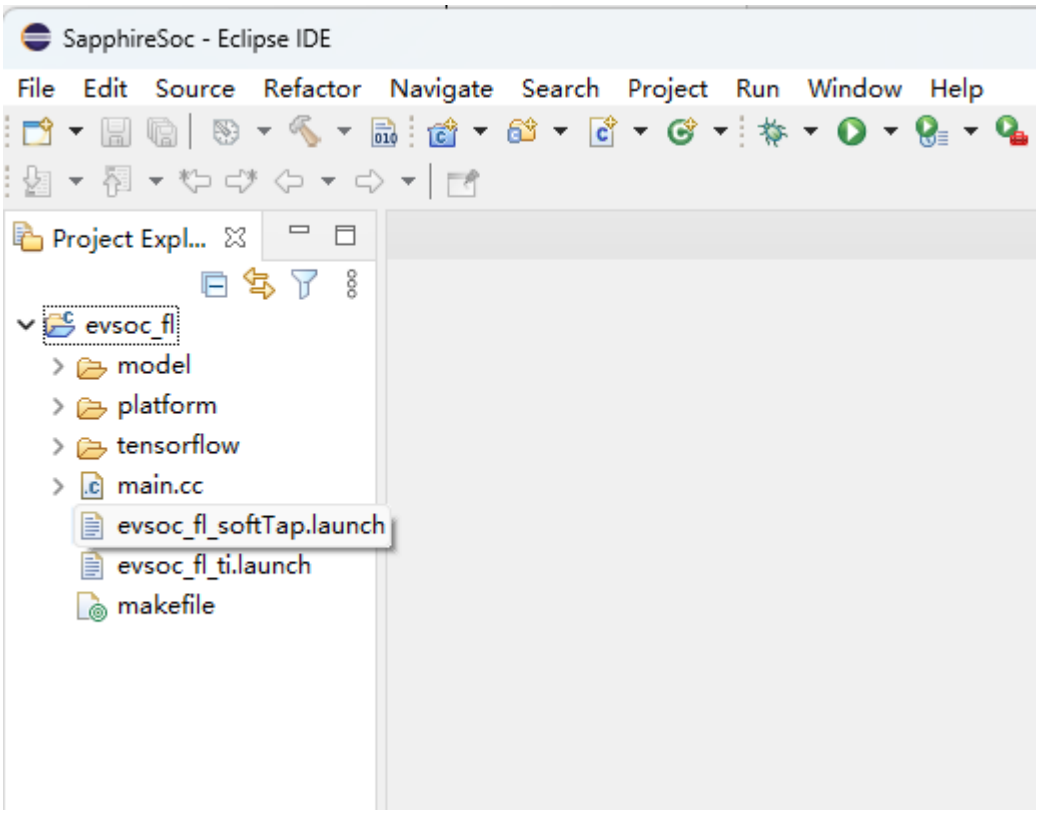
TinyML 工程开发框架基于 TensorFlow-Lite 进行开发，整体的软件编写步骤如下图，其中，蓝色部分的三个流程，只需重复执行单次，并且不同项目高度一致，填充数据和数据后处理，则会跟具体项目不同有较大区别。



TinyML 软件模板工程目录结构如下：

文件名	说明	备注
build	编译输出目录	
platform	Tinyml 驱动目录	
tensorflow	TFlite 库目录	
model	模型文件目录	
main.cc	工程 main 文件	
makefile	编译文件	

导入后的我工程，如图所示：

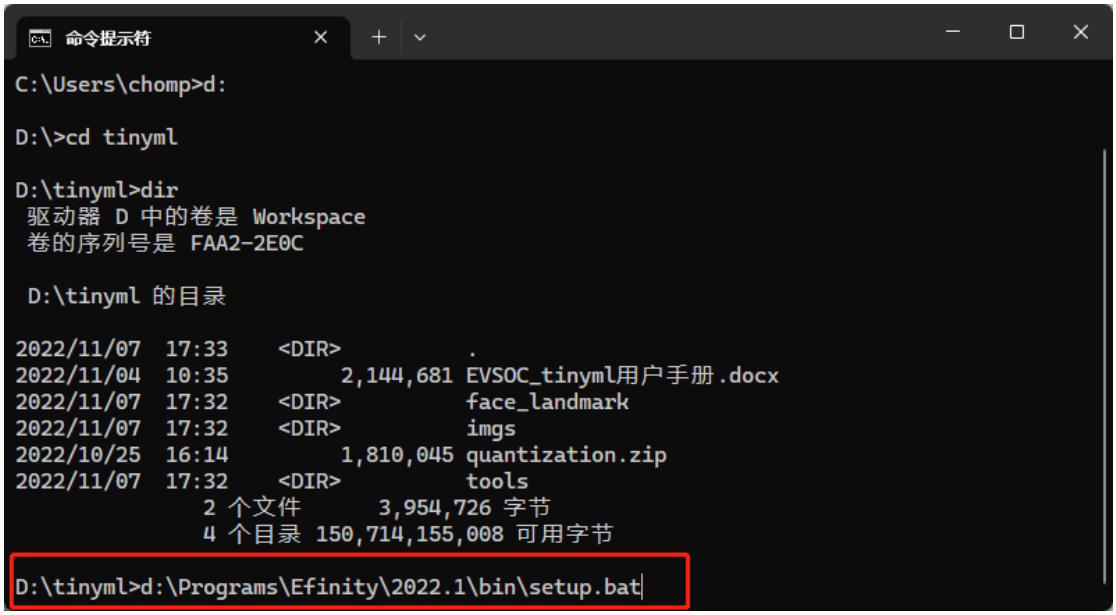


4.4.2 Tflite 模型导入

该不走进针对训练后量化的 tensorflow-lite 文件，该文件类型的后缀名称为：model.tflite，如目前没有 tflite 文件，请参考 4.4.3《训练后量化》章节。

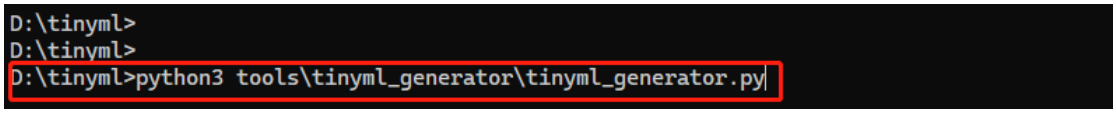
4.4.2.1 配置运行环境

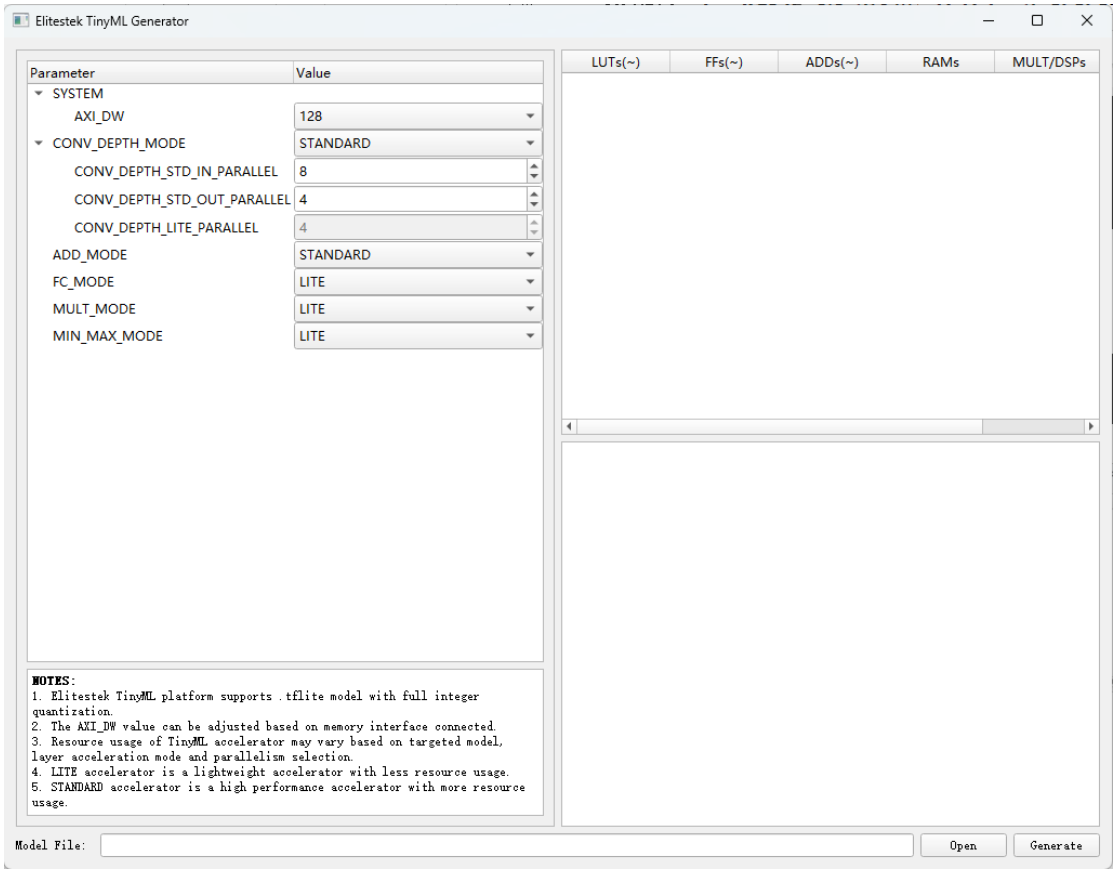
该脚本依赖 Efinity 工具中提供的 setup.bat(Linux 环境为 setup.sh)，打开命令行窗口并执行该命令:



4.4.2.2 TinyML 参数配置

在刚才的命令行窗口，键入并执行：
python3 tools\tinyml_generator\tinyml_generator.py 命令





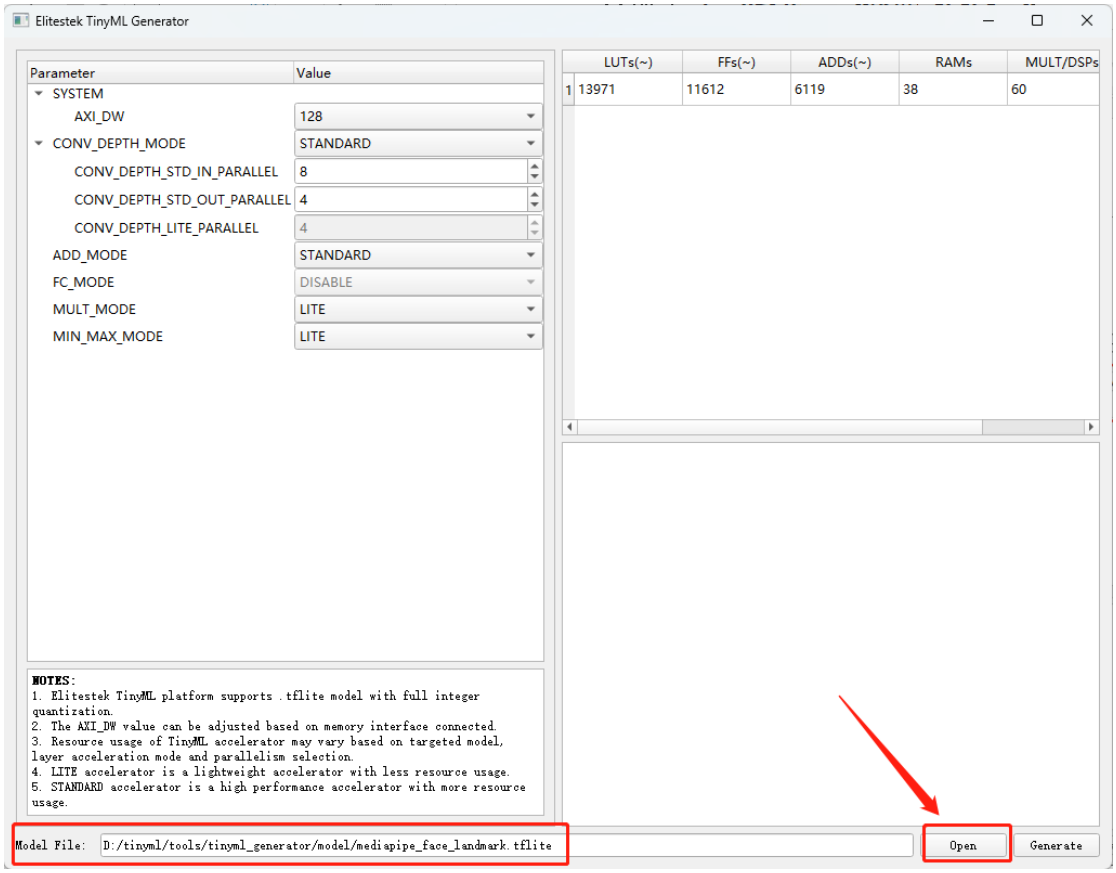
4.4.2.3 配置 TinyML 加速算子参数

在应用过程中用户可使用 TinyML Generater 工具进行参数化配置，可根据资源大小进行调节，在资源占用和性能间选择最佳配置。

工程文件夹(tools/tinyml_generator/model 中包含了如下几个模型的示例代码，分别对应关系为：

模型文件	Efinity 工程
mediapipe_face_landmark.tflite	Ti60F225_mediapipe_face_landmark_demo
mobilenetv1_person_detect.tflite	Ti60F225_mobilenetv1_person_detect_demo
yolo_person_detect.tflite	Ti60F225_yolo_person_detect_demo

我们选择 mediapipe_face_landmark.tflite 作为示例进行演示，首先点击 Open 按钮，选中并加载 Tensorflow-Lite 的 tflite 文件。



有关 TinyML 参数，这里有一份简单的说明：

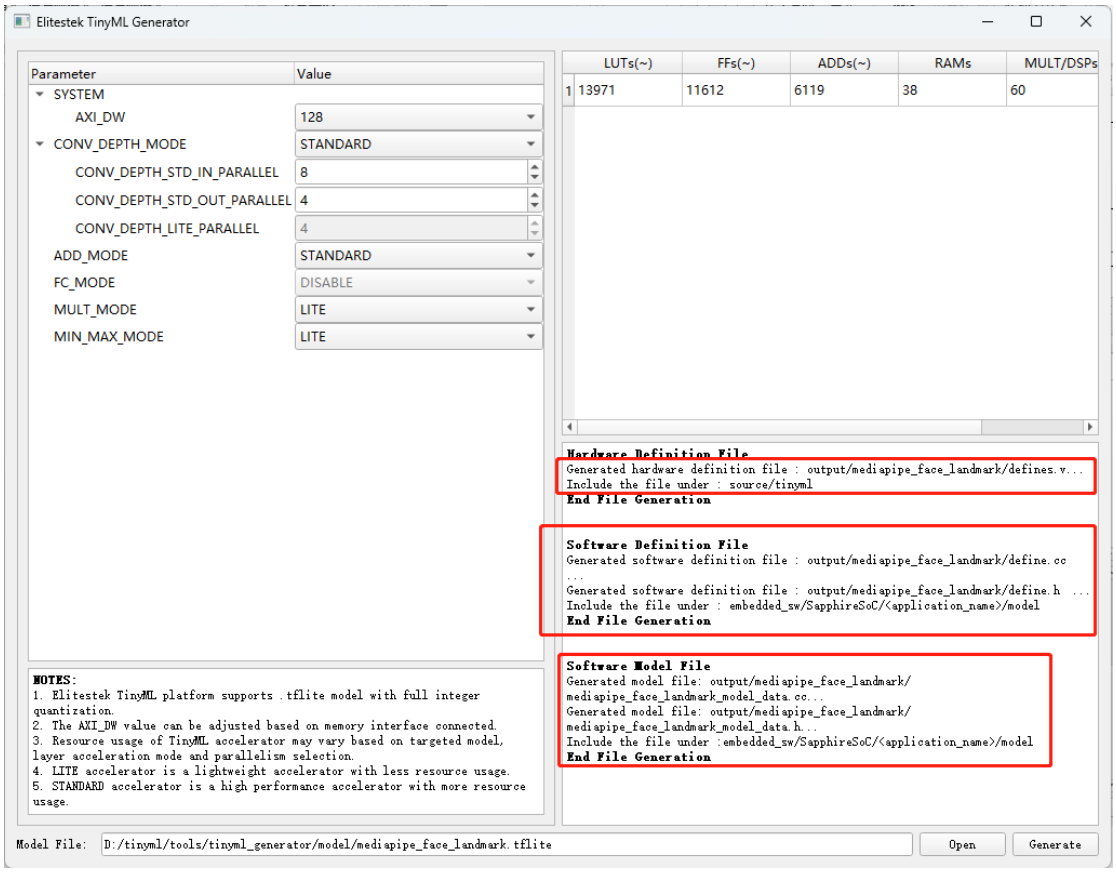
参数	子参数	默认值	备注
CONV_DEPTH_MODE		STANDARD	Conv、Depthwise 层的工作模式
	CONV_DEPTH_STD_IN_PARALLEL	4	标准加速模式下的输入/输出通道设置
	CONV_DEPTH_STD_OUT_PARA€LLEL	4	
	CONV_DEPTH_LITE_PARALLEL	4	精简模式下的并行通道数量
	CONV_DEPTH_LITE_AW	7	
ADD_MODE	无	STANDARD,LITE	Add 层工作模式
FC_MODE		LITE	只支持精简模式
MULT_MODE		LITE	
MIN_MAX_MODE		LITE	

- Standard 模式：为高性能模式，资源占用较多
- Lite 模式：资源占用较低

4.4.2.4 生成模型参数文件

点击“Generate”按钮生成工程所需要的文件，如下图工具中会提示文件最终的生成路径，这些文件按照提示，放入指定位置：

文件名	用途	存放路径
define.h define.cc	加速器软件参数	RISCV-SDK 工程目录文件夹： embedded_sw/SapphireSoC/$project_name /model /
model_name_model_data.cc	导出的 tflite 模型文件	
model_name_model_data.h		
define.v	TinyML RTL 硬件工程的参数	将该文件放入 rtl 工程目录的： $project_name/source/tinyml

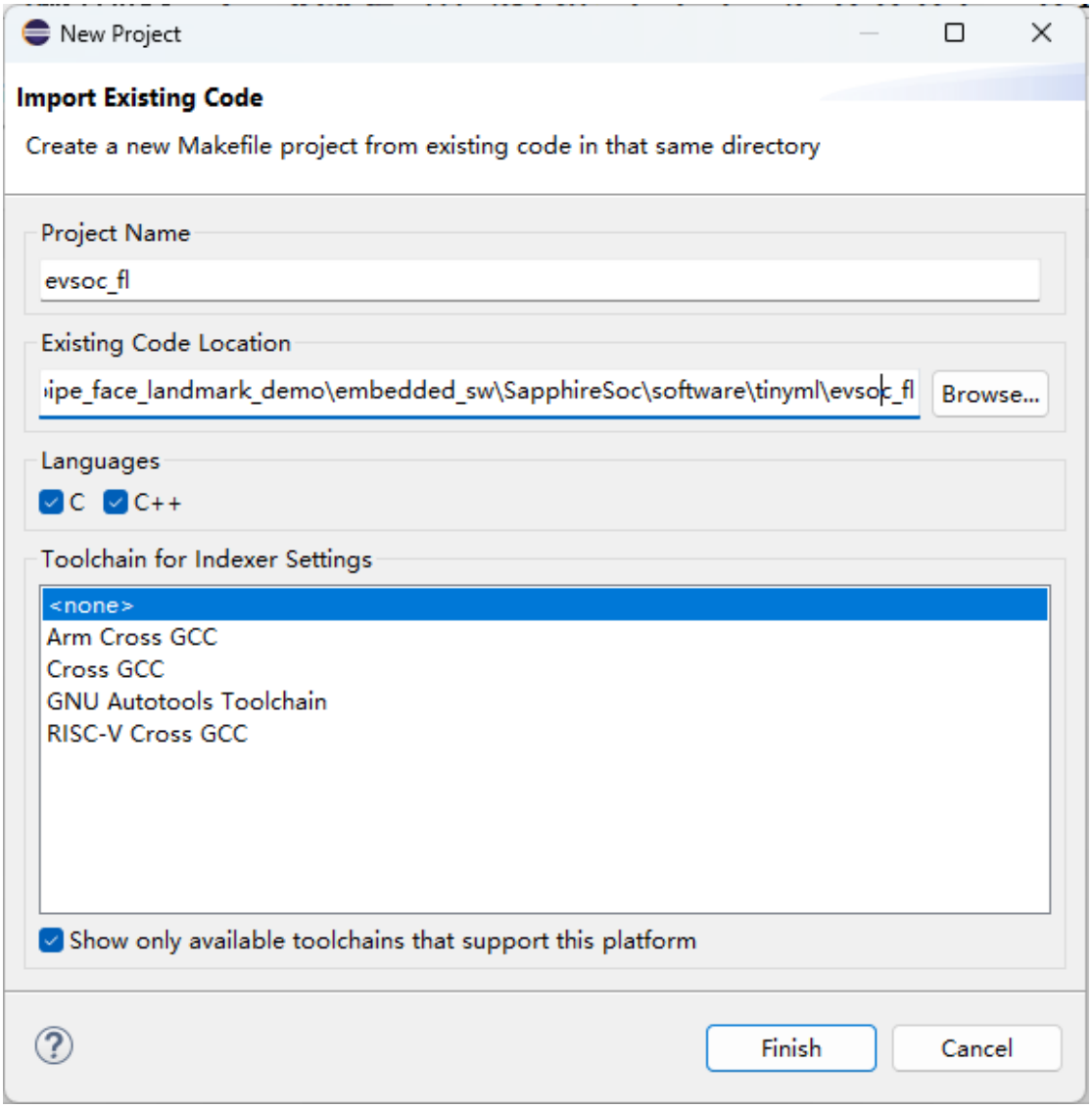


4.4.2.5 添加工程

在导入软件工程前，需要进行工程导入和 workspace 切换，具体操作如下：

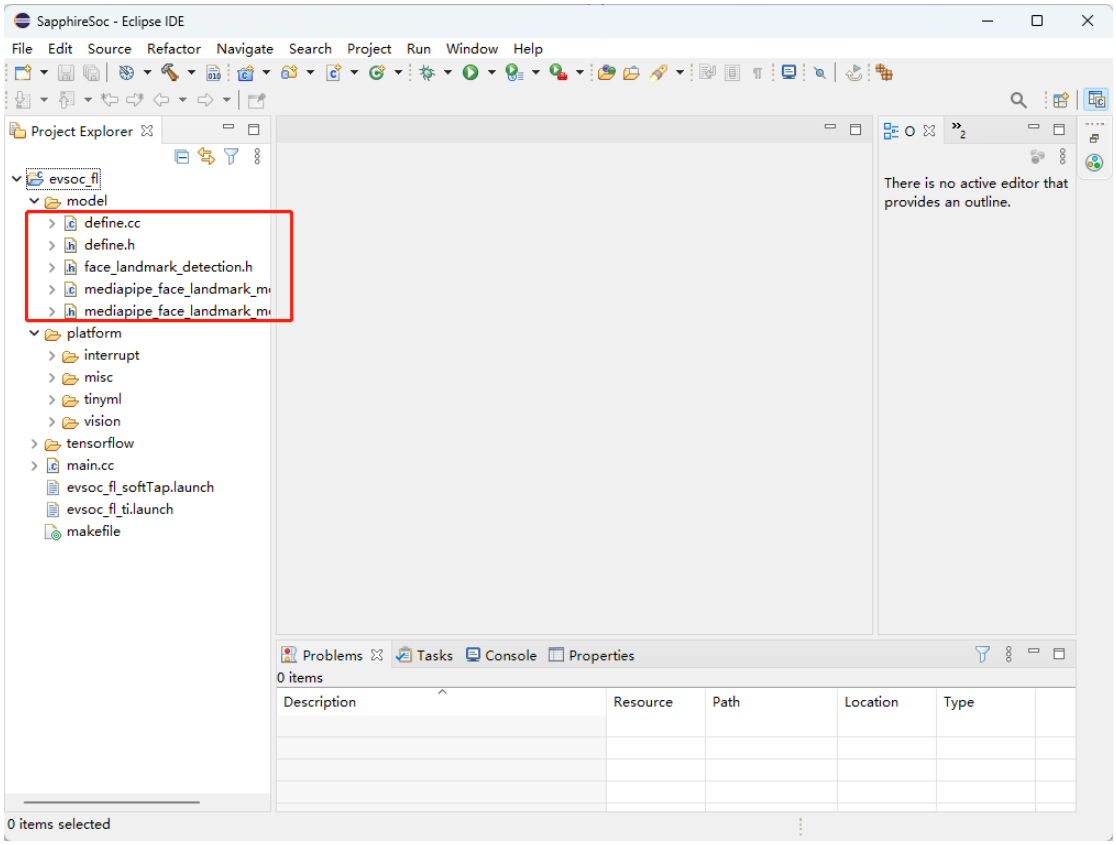
1. 打开 RISCV-SDK，并选择 Sapphire 启动。
2. 点击 File -> switch Workspace –> Other。
3. 选中软件工程$project/embedded_sw/ SapphireSoc
4. 选择导入 File -> New -> Makefile Project with Existing Code，选择内容如

下，完成后点击 Finish。



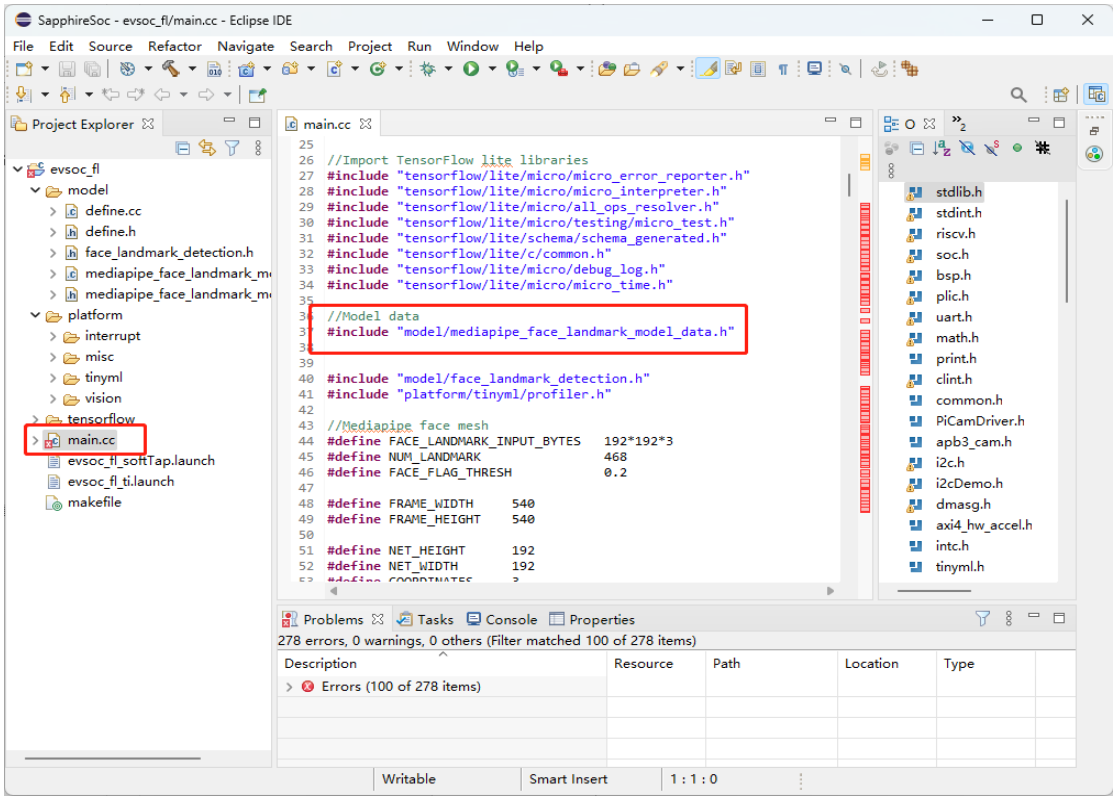
4.4.2.6 添加文件

将上述步骤生成的模型文件，放入工程目录的 model 文件夹中。



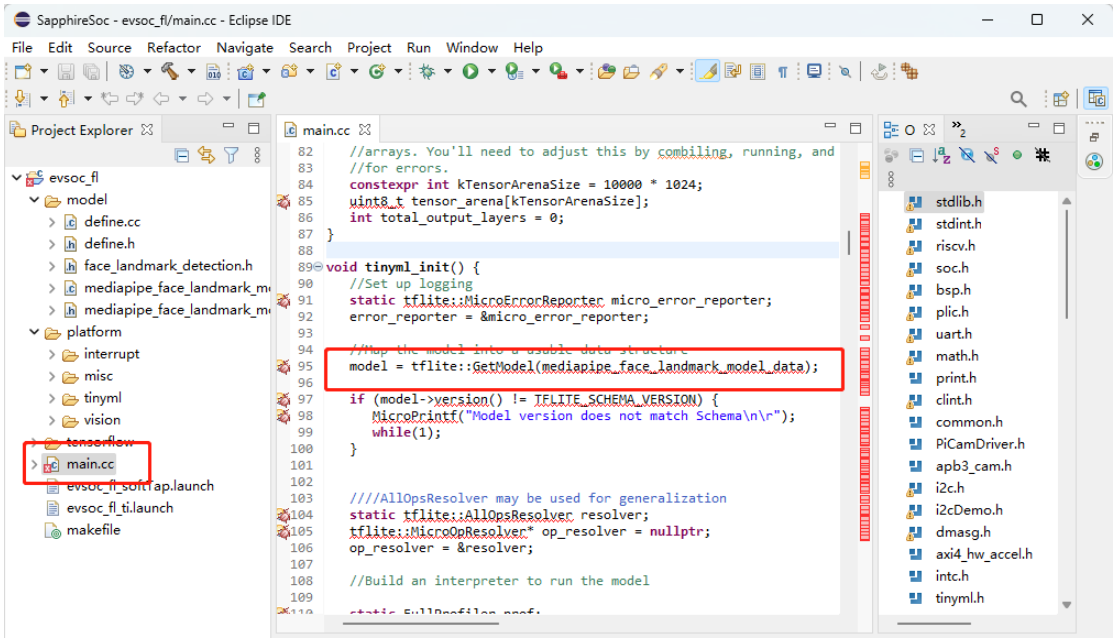
4.4.2.7 代码修改

将模型头文件 face_landmark_model_data.h 引用添加入 main.cc：

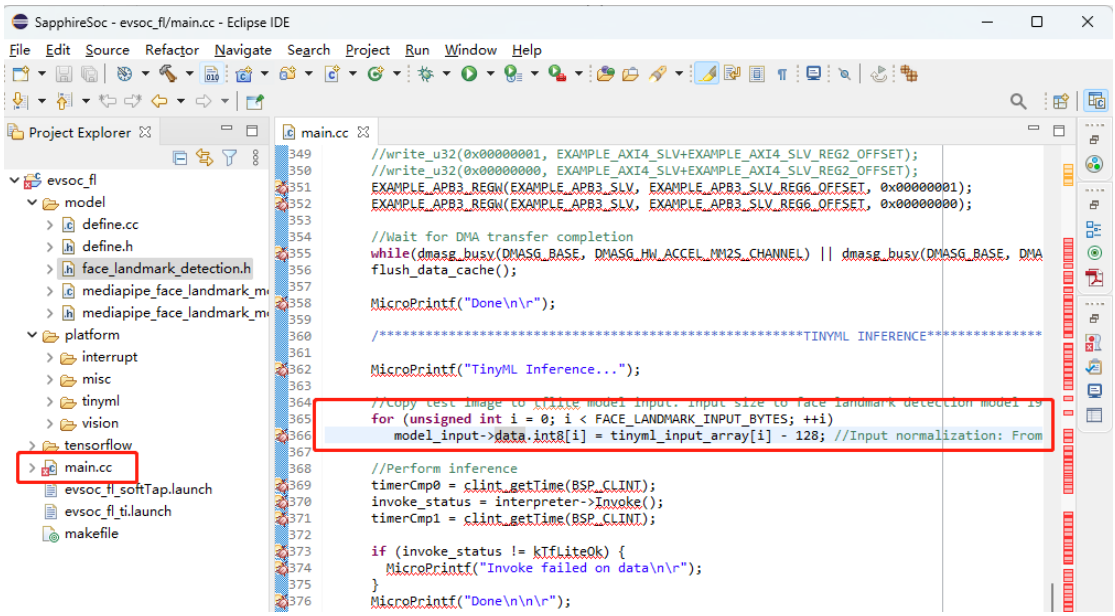


使用 tflite 库加载该模型数据，该代码为：

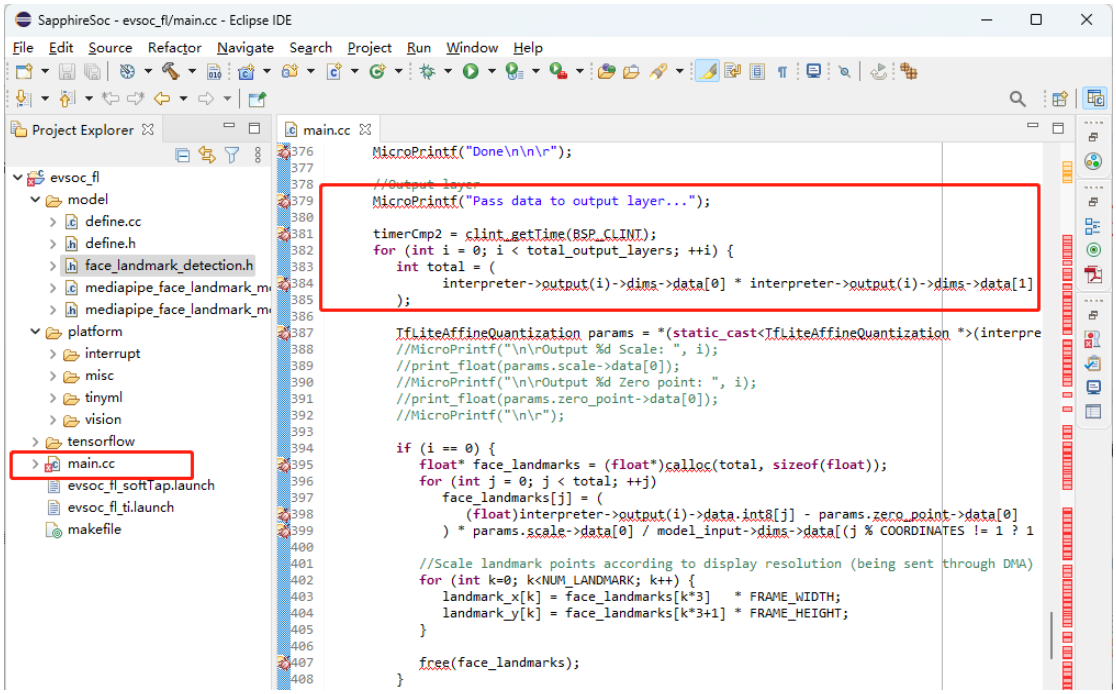
```
tflite::GetModel(mediapipe_face_landmark_model_data);
```



调用 tflite interpreter->Invoke()函数，来启动模型推断。注意每次 Invoke 前，需要更新输入 Tensor，也就是 interpreter->input(0)的数据，该项目使用 DMA 填充来自摄像头的数据。

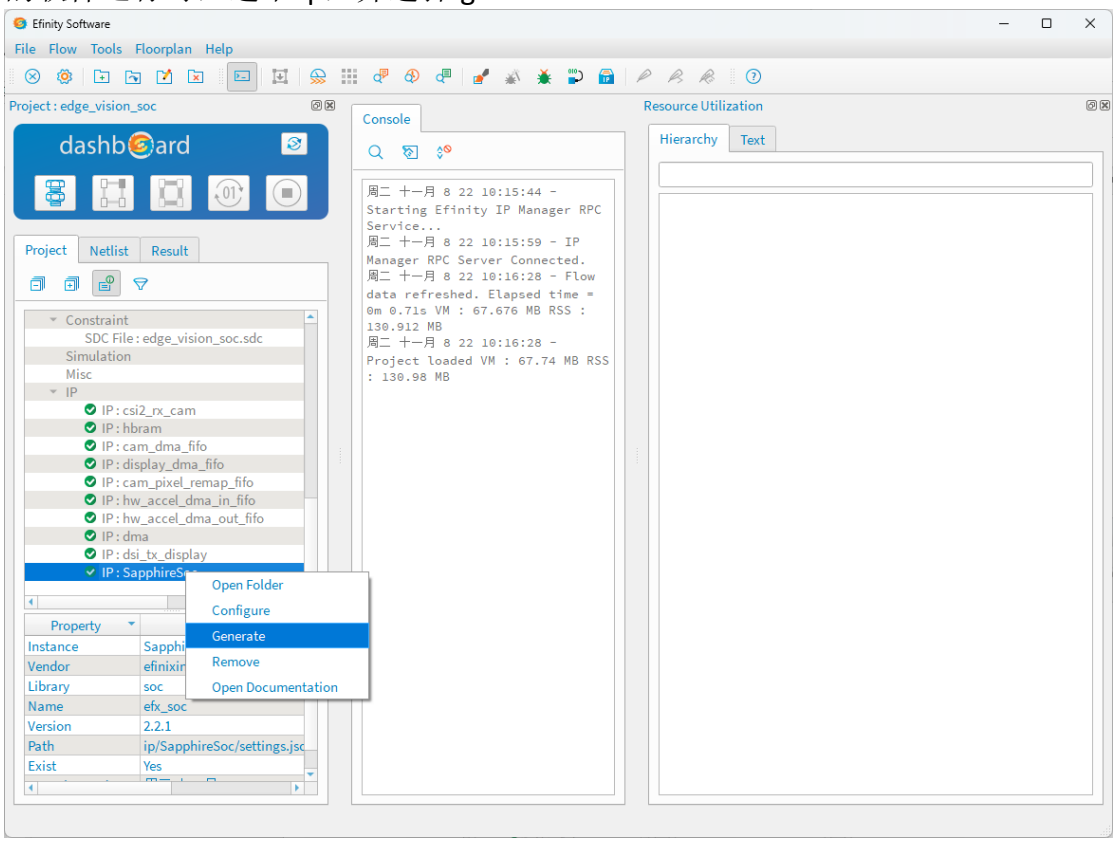


添加推理数据和推理代码之后，就是获取结果输出，Tensorflow-Lite 推理的结果保存在 interpreter->output 这个 Tensor 中，该 Tensor 的输出内容，与网络结构相关，通常如果只输出一组数据，那么结果就是 interpreter->output(0)。该例程中使用的 face_landmark 输出为面部点阵，数据稍复杂，如图所示：



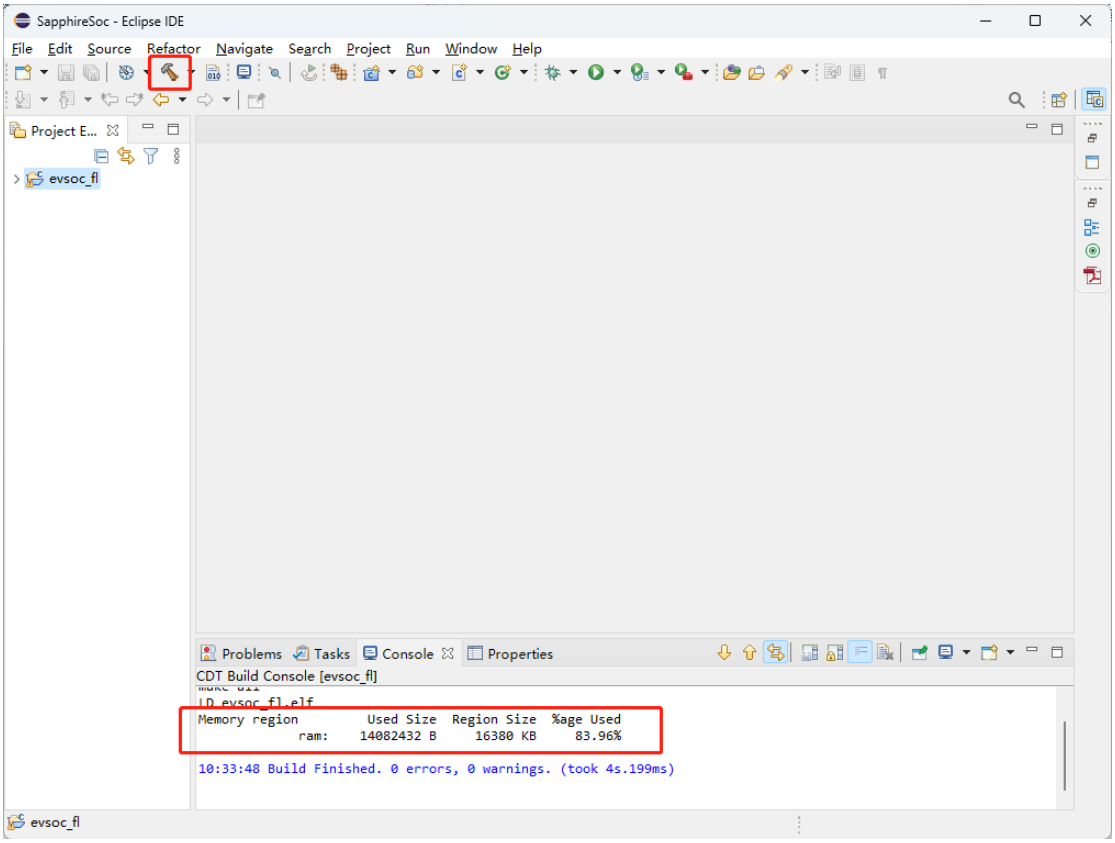
4.4.2.8 编译工程

在编译软件工程之前，需要根据当前工程（tinyml_vision/Ti60F225_mediapipe_face_landmark_demo），生成 sapphire soc 的软件运行时，选中 ip，并选择 generate。



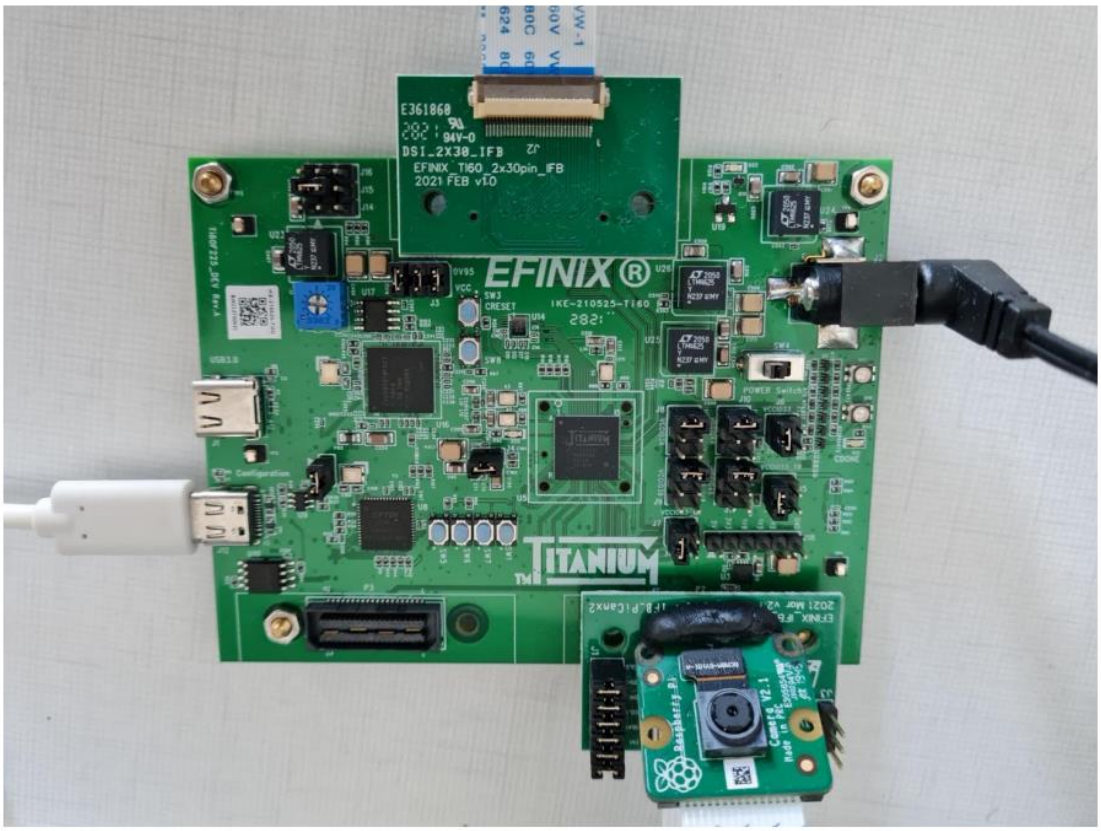
软件代码合并成功后，点击编译按钮编译该工程，编译完成后会输出该工程的内存占用大小，可根据实际情况判断是否需要减小 kTensorArenaSize 预分

配空间的值，该值默认为（10M）。点击 build 按钮编译该工程：

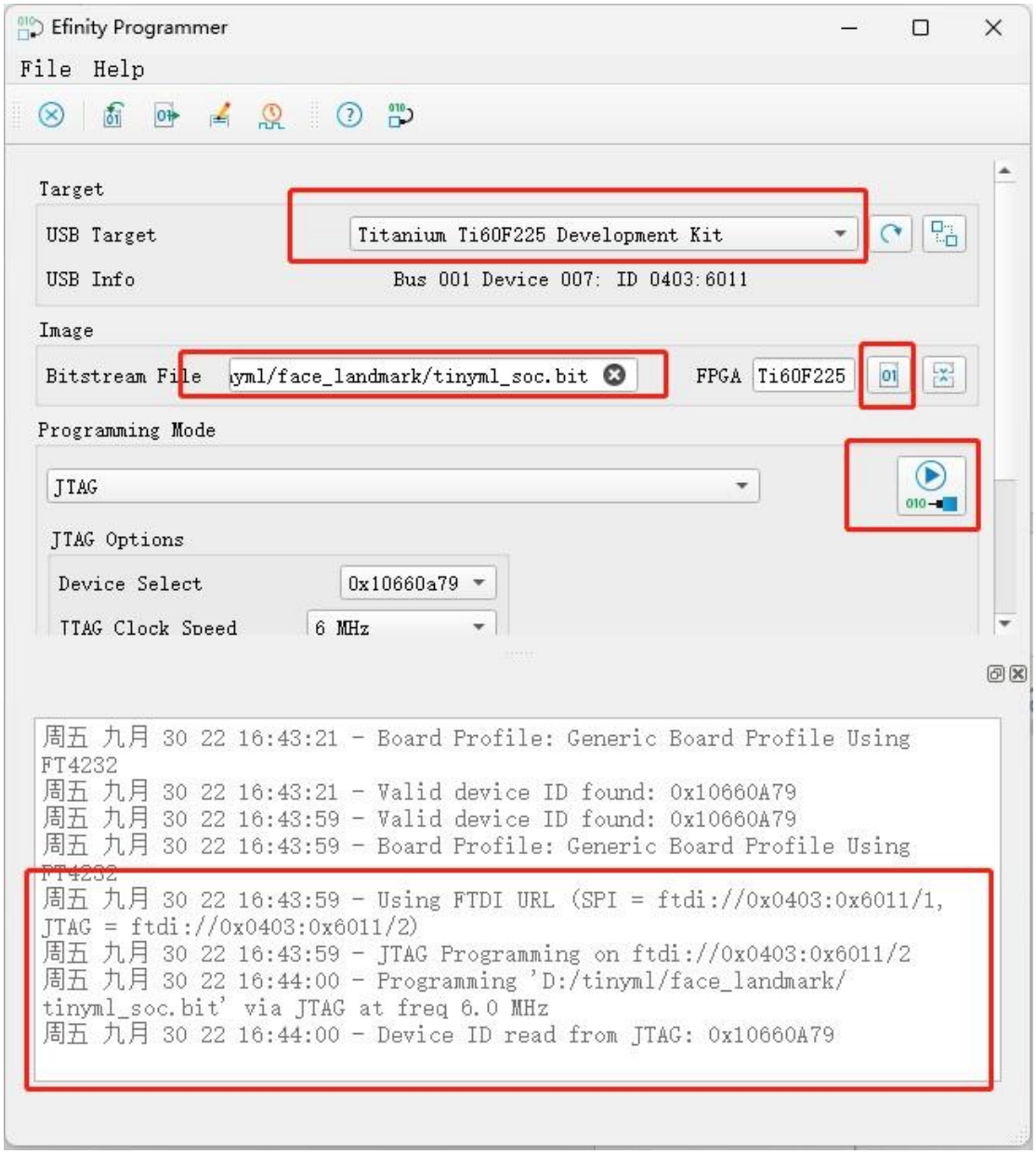


4.4.2.9 运行和调试

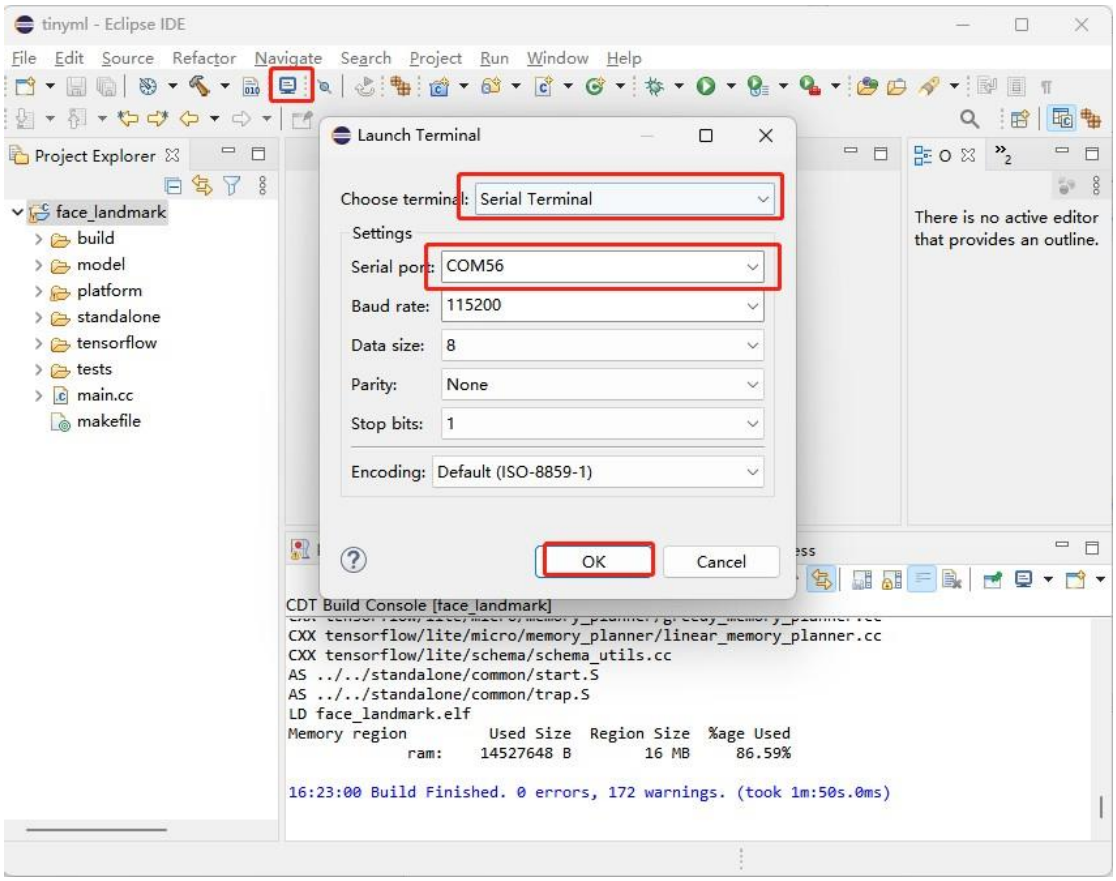
在运行之前，首先需要使用 Efinity 工具综合和编译继承好的硬件 bitstream 烧写进入开发板。如需使用工程预编译好的 bitstream，可使用 efinity 烧写工具，参考下图直接进行烧写。这里给出一份 Ti60F225 DevKit 的连接图，详细的外设连接，请参考手册《ti60_evsoc_hw_setup.pdf》


E361868
DSI_2X30_IFB
EFINIX_Ti60_2x30pin_IFB
2021 FEB v10
EFINIX®
IKE-210525-Ti60
TITANIUM

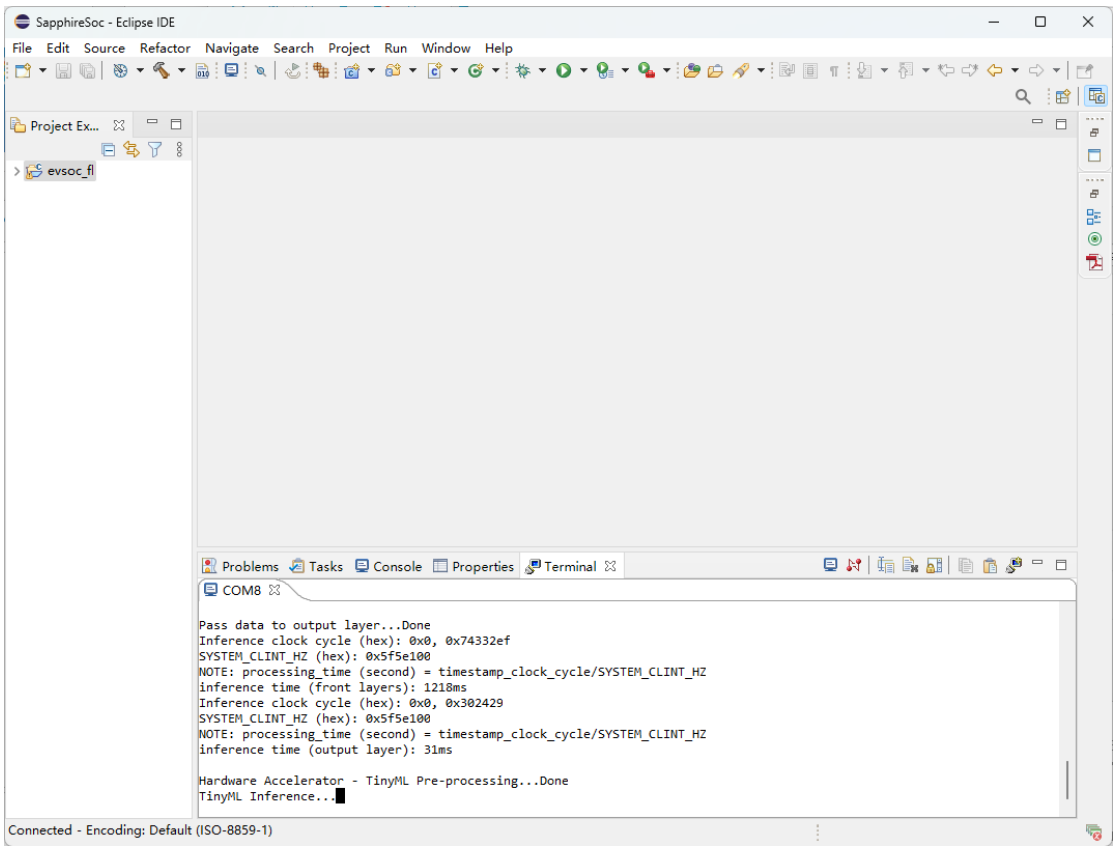
打开 Efinity Programer，并烧写 bitstream：



bitsteam 烧写成功后，需要在 RISCV-SDK 中打开并配置终端，特别注意 COM 的名称会有所变动，Ti60F225 Devkit 有两个 COM 接口，选择第一个即可，下图仅作参考



运行该工程，等待大约 10 秒，会打印如下结果，可以看到一次推理耗时约为 940ms：



4.4.3 TensorFlow 训练后量化

4.4.3.1 概念

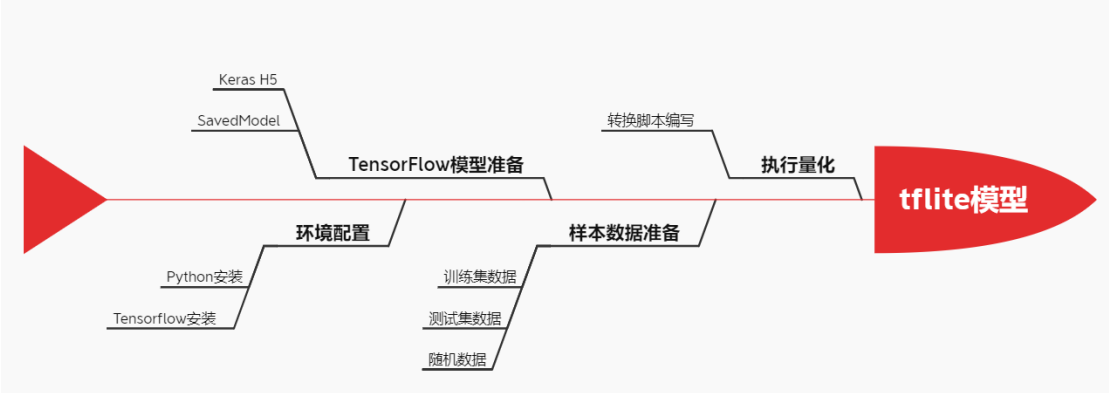
训练后量化的作用主要是为了节省计算资源和加速推断过程，在尽量不影响模型精度的前提下，提供更好的性能和更低的资源占用。实际的 AI 项目种，所使用的模型大多为 int8 量化模型，下图为一个性能参考：

技术	优势	硬件
动态范围	大小缩减至原来的四分之一，速度加快 2-3 倍	CPU
全整数	大小缩减至原来的四分之一，速度加快 3+ 倍	CPU、Edge TPU、
Float16 量化	大小缩减至原来的二分之一，速度加快 2 倍	CPU、GPU

- 全整数量化是将 FP32[$\pm 2^{-126}$， $\pm(2^{128} - 2^{104})$]的取值范围，量化在 Int8[**-128, 127**]

  范围内进行计算。将存储和带宽需求降低为原来的 1/4，计算能力提升的同时可节省大量资源。同样由于动态范围的缩小，在量化时就需要考虑裁剪/滑动取值窗口来进行参数校准。因此量化时，需要最接近实际场景的推理数据来对模型进行优化。

- 模型量化的整体流程为：

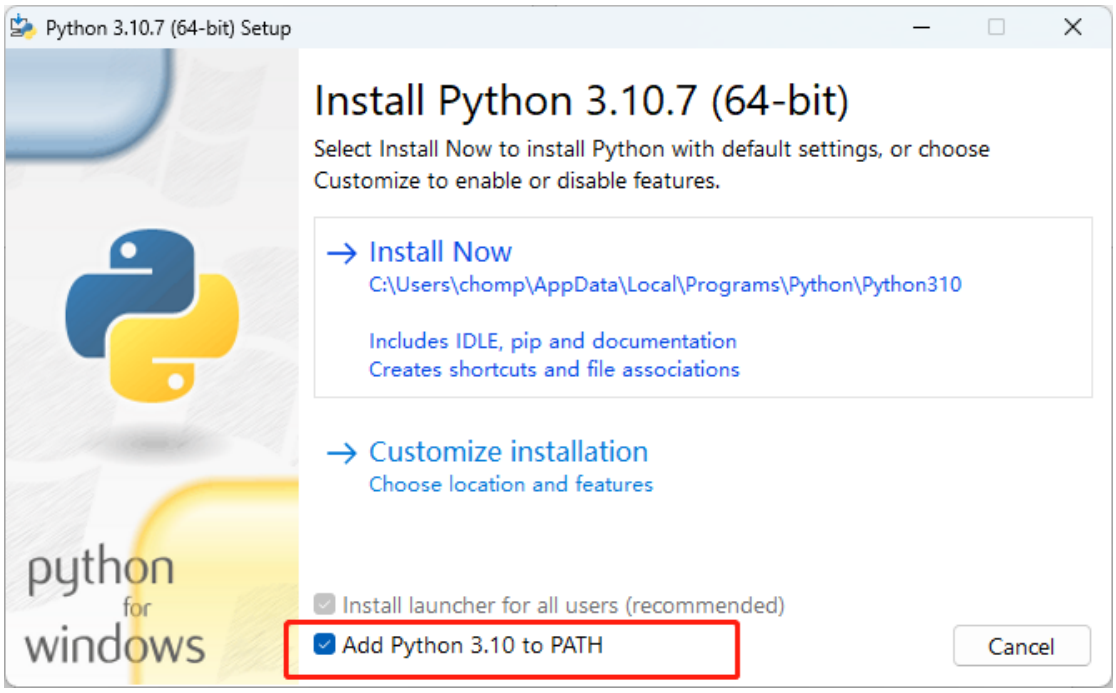


4.4.3.2 环境配置

Google 在 TensorFlow 库中，集成了 tflite 模型的导出工具，因此，在进行以下工作之前，首先需要安装 tensorflow 工具包。

Tensorflow 安装依赖 Python，首先需要下载安装 Python。

Windows 系统需下载安装包: Python3.10.7，另外如图所示，需要勾选 Add Python 3.10 to PATH 选项，用于命令行使用。



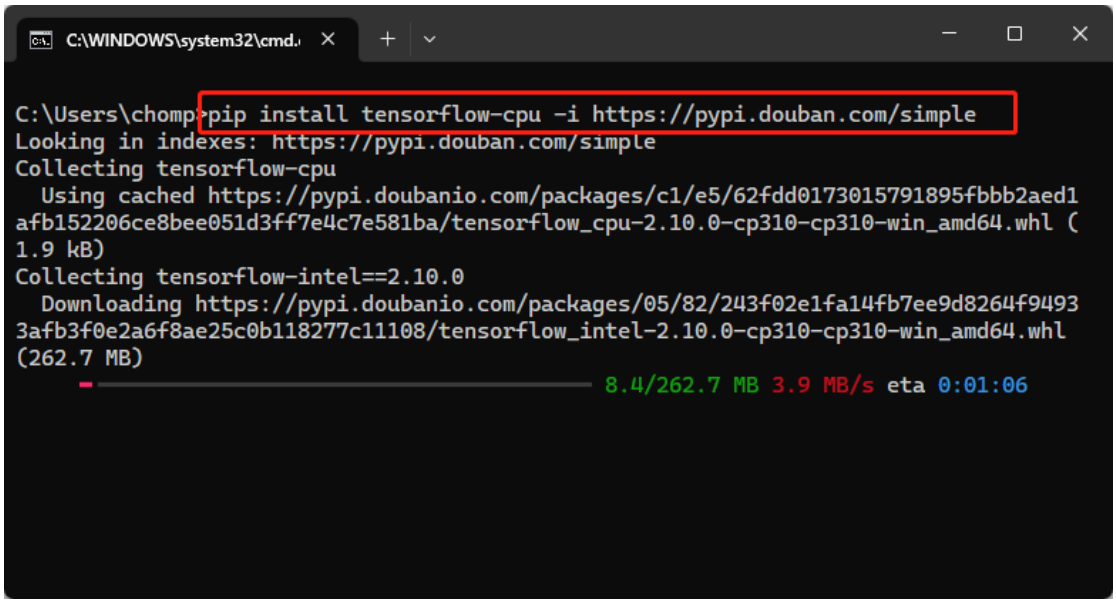
Linux 系统可直接使用命令行进行安装：

```
sudo apt-get install python3 python-is-python3
```

Python 安装完成后，即可使用命令行工具进行 tensorflow 安装，首先需要打开命令行窗口，Windows 系统下，为命令提示符，Linux 下则为终端模拟器，打开后键入如下命令：

```
pip install tensorflow-cpu -i https://pypi.douban.com/simple
```

Windows 命令提示符如图，等待安装完成即可。



4.4.3.3 TensorFlow 模型准备

TensorFlow2.0 目前支持常见的导出模型为 saved_model 和 keras h5 两种模型。前者通常为一个文件夹/压缩包。后者为.h5 结尾的单独文件，工程应用中，这两种模型文件都较为常见。

另外，在模型量化过程中，为了提升量化准度，也需要准备一些校准样本文件，校准文件是指对项目使用中的真实数据有代表性的数据，越符合实际分布越好。

如没有准备好校准数据，通常建议使用如 ImageNet、Cifar10、CoCo 等标准数据集中的数据。当然也可以直接生成随机数 tensor 进行填充，但是可能影响模型的精度。

4.4.3.4 编写量化脚本

Tensorflow 进行量化时，需要编写对应的 python 脚本，流程上比较固定，只是校准数据预取部分的函数需要根据情况自行实现，以下代码为范例代码，可在该脚本目录新建 imgs 文件夹，将校准图片放入该目录，然后通过命令行执行该脚本即可：

```
import tensorflow as tf
import numpy as np
import pathlib
# 该函数实现为整个模型转换过程中最关键的部分，需要根据实际情况提供量化时的校准数据。
# 理论上这里的数据应该接近实际数据分布越好，需求 100~500 份。
# 如果使用标准模型如 mnist, cafir10, imagenet, coco 等标准库进行模型训练，那么可以直接使用 test 标签下的数据。
# 如果为自定义图片， 则需要根据场景提供合适的图片集
# 将图片加载放入 train_images 变量中
# def representative_dataset(): #需要根据模型数据构造编写
#     #从训练数据中抽取 100 份，实现参考数据输入
#     data = tf.data.Dataset.from_tensor_slices(train_images).batch(1).take(100)
#     for tensor in data:
#         yield [tensor]
img_height = 0
img_height = 0
def representative_dataset():
  data_dir = pathlib.Path("imgs").glob("*.jpg")
  for i in data_dir:
      img = tf.io.read_file(i.as_posix())
      tensor = tf.io.decode_image(img, channels=3, dtype=tf.dtypes.float32)
      tensor = tf.image.resize(tensor, [img_width, img_height])
      yield [tf.expand_dims(tensor, 0)]

#训练后的 FP32 模型加载
```

```
model_name = 'trained_models' #saved_model 文件格式
# model_name= 'trained_models.h5' #Keras H5 文件格式

model = tf.keras.models.load_model(model_name)
converter = tf.lite.TFLiteConverter.from_keras_model(model)
[(_, img_width, img_height, _)] = model.layers[0].input_shape

converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_quant_model = converter.convert()
open(model_name + ".tflite", "wb").write(tflite_quant_model)
```

4.4.3.5 实例参考（Resnet）

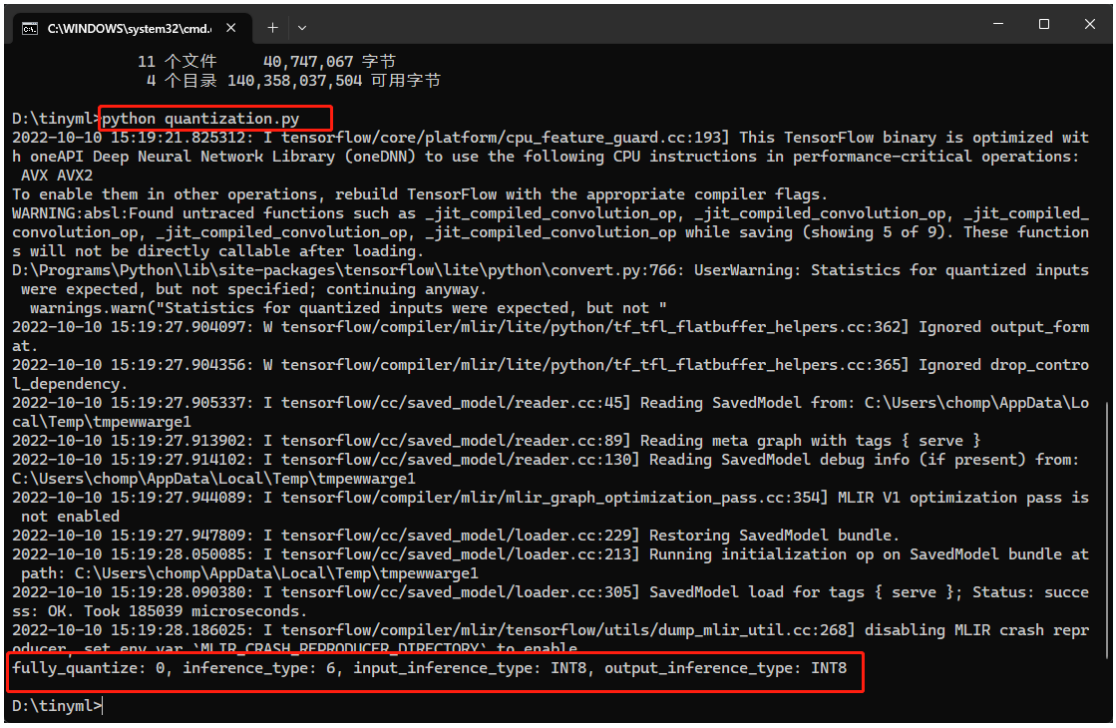
附件文件夹中 quantization.zip 压缩包，提供了该实例所需要用到的文件目录，该实例使用一个已经训练好的 Resnet 模型进行训练后量化，文件内容为：

文件类型	描述	示例文件
模型文件	TensorFlow 所支持的两种模型格式,save_model 和 H5 格式	trained_models trained_model.h5
参考数据集	用于 int8 量化时的参数矫正，提高模型准确率。来源为模型训练时的训练集、测试集、表达集。	Imgs/*.jpg
量化脚本	简单模型转换代码	quantization.py
Tflite 模型	量化后自动生成	trained_models.tflite

附件文件进行解压，并使用命令行执行如下命：

```
python quantization.py
```

执行结果：



最终在同目录下，生成 trained_model.tflite 文件。可参考 4.4.2《创建 TinyML 工程》使用该模型文件。

4.5 加载应用程序

您可以通过两种方式加载应用程序的二进制文件：

- 使用 OpenOCD 调试器引导
- 从 flash 芯片启动

本节主要介绍从 flash 芯片引导的步骤。在第 20 页的中描述了使用 OpenOCD 调试器引导的步骤。

当 FPGA 启动时，Ruby SoC 将二进制应用程序从 SPI flash 芯片复制到外部的内存模块里，然后开始执行应用程序。SPI flash 的二进制地址从 0x0038_0000 开始。

从 SPI flash 芯片里加载程序的步骤如下：

1. 打开电源，FPGA 从板载 flash 芯片里加载配置信息。
2. 配置完成后，引导程序开始将 flash 芯片里（地址为 0x0038_0000，大小为 124 KB）的用户二进制文件复制到片外 DRAM（地址为 0x0000_1000）。
3. Edge Vision SoC 跳转到逻辑地址 0x0000_1000 去执行用户的二进制文件。