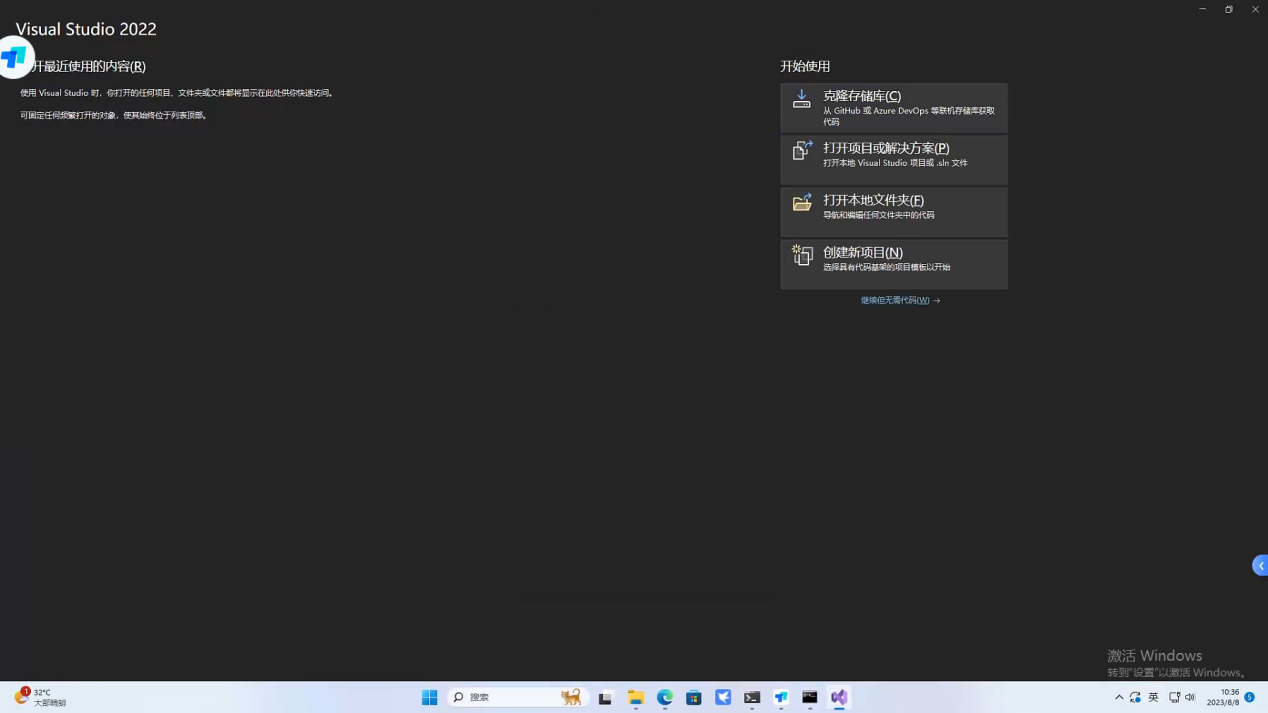
**Windows下使用onnxruntime部署yolov8-cls分类模型**

**一、安装配置**

1. **安装Vistual Studio2022**

参考博客：http://t.csdn.cn/fY9zJ

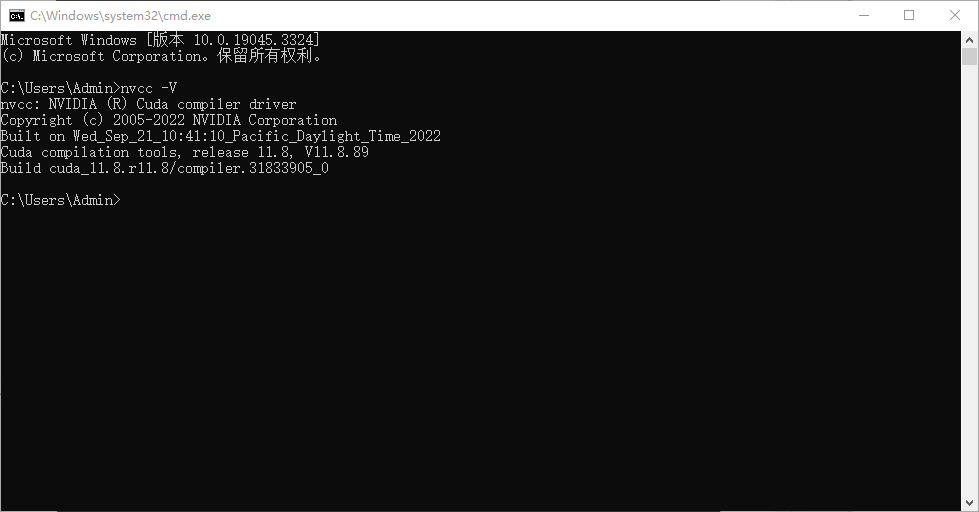
下载地址：<https://visualstudio.microsoft.com/zh-hans/>



1. **安装CUDA11.8及cudnn**

参考博客：<http://t.csdn.cn/PQcis>

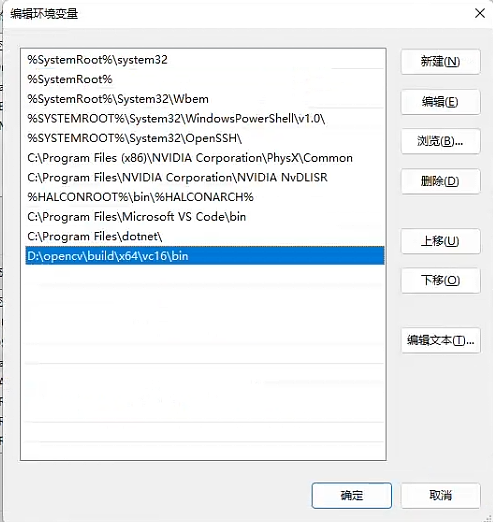
下载地址：https://developer.nvidia.com/cuda-downloads



1. **安装opencv4.8.0**

参考博客：<http://t.csdn.cn/NHQXk>

下载地址：<https://opencv.org/>



1. **安装onnxruntime1.15.1**

参考博客：http://t.csdn.cn/txYqv

下载地址：https://github.com/microsoft/onnxruntime/releases

下载windows版onnxruntime-osx-x86\_64-1.15.1.tgz解压即可

**注意必须与CUDA版本对应！**可参看网址：

https://onnxruntime.ai/docs/execution-providers/CUDA-ExecutionProvider.html

1. **安装cmake-GUI**

参考博客：http://t.csdn.cn/ATphP

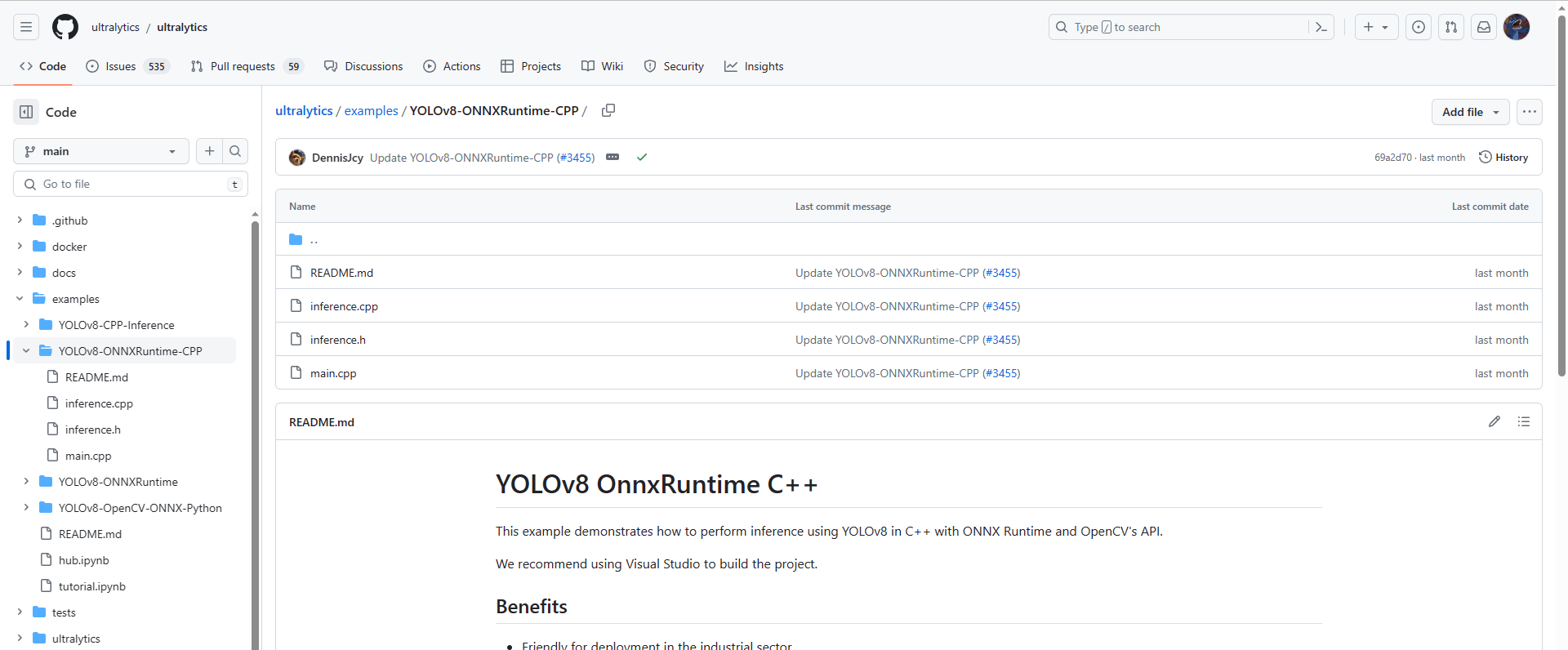
下载地址：https://cmake.org/download/

**二、项目配置**

1. **下载yolov8-detect检测模型c++推理代码**

YOLOv8官方仓库：https://github.com/ultralytics/ultralytics

yolov8-detect检测代码路径：./examples/YOLOv8-ONNXRuntime-CPP/



1. **配置CmakeList**

创建CmakeList文件CMakeLists.txt，添加内容如下：

cmake\_minimum\_required(VERSION 3.5)

project(Yolov8CPPInference VERSION 0.1)

set(CMAKE\_INCLUDE\_CURRENT\_DIR ON)

# CUDA

option(CUDA\_USE\_STATIC\_CUDA\_RUNTIME OFF)

set(CMAKE\_CXX\_STANDARD 11)

set(CMAKE\_BUILD\_TYPE Debug)

# TODO(Call for PR): make cmake compatible with Windows

set(CMAKE\_CUDA\_COMPILER "C:\\Program Files\\NVIDIA GPU Computing Toolkit\\CUDA\\v11.8\\bin\\nvcc") #1

enable\_language(CUDA)

set(CMAKE\_CUDA\_STANDARD 11)

set(CMAKE\_CUDA\_STANDARD\_REQUIRED ON)

# !CUDA

# OpenCV

find\_package(OpenCV REQUIRED)

include\_directories(${OpenCV\_INCLUDE\_DIRS})

# !OpenCV

set(PROJECT\_SOURCES

    main.cpp

    inference.h

    inference.cpp

)

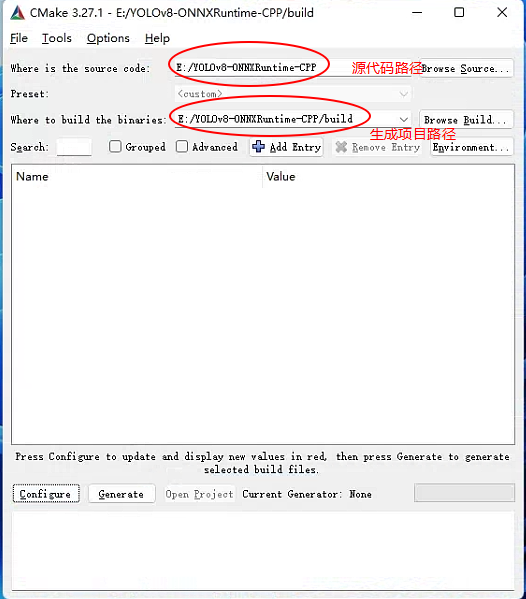
add\_executable(Yolov8CPPInference ${PROJECT\_SOURCES})

target\_link\_libraries(Yolov8CPPInference ${OpenCV\_LIBS})

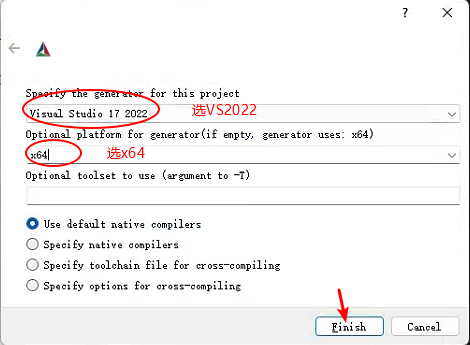
#1：需要更换为当前设备中CUDA对应的位置

1. **使用cmake-GUI生成项目**

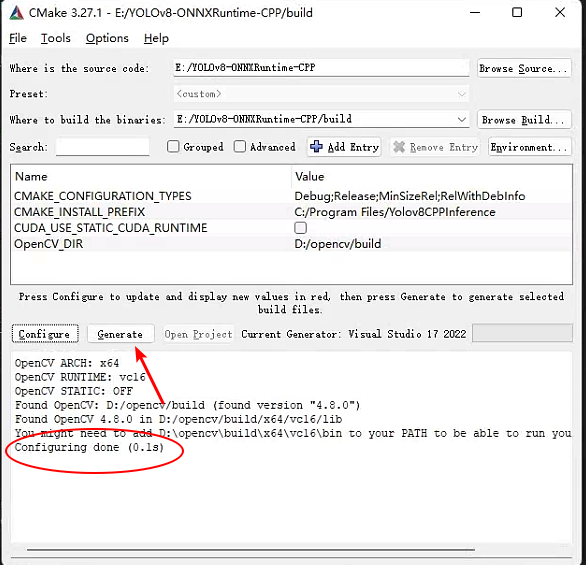
**（1）**源代码路径选择下载后的YOLOv8-ONNXRuntime-CPP目录；生成项目路径选择YOLOv8-ONNXRuntime-CPP/build(未创建会自动生成)；点击configure；



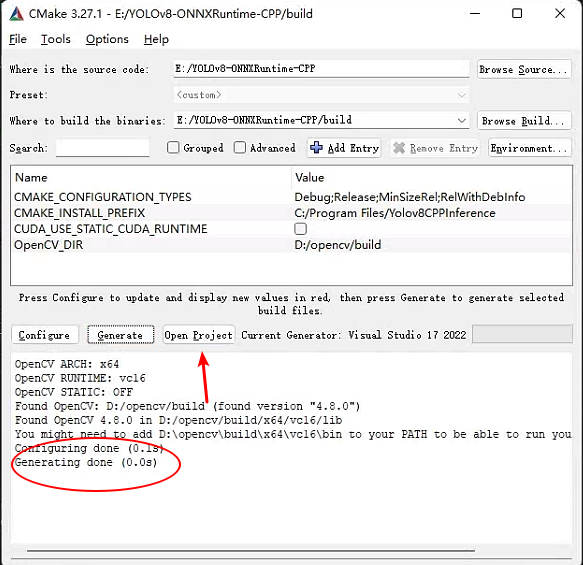
**（2）**生成器选择Visual Studio 17 2022，平台选择x64，点击finish；



**（3）**点击Generate生成项目；

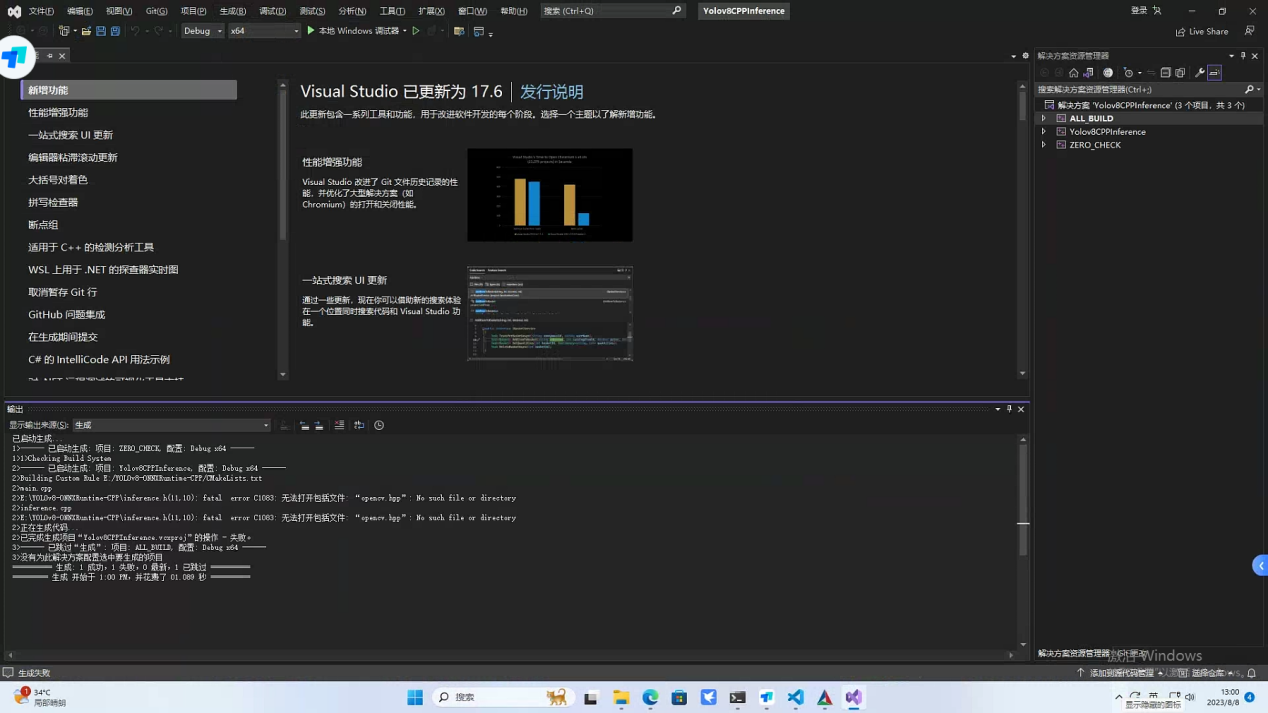


**（4）**生成完成后点击open project打开项目。

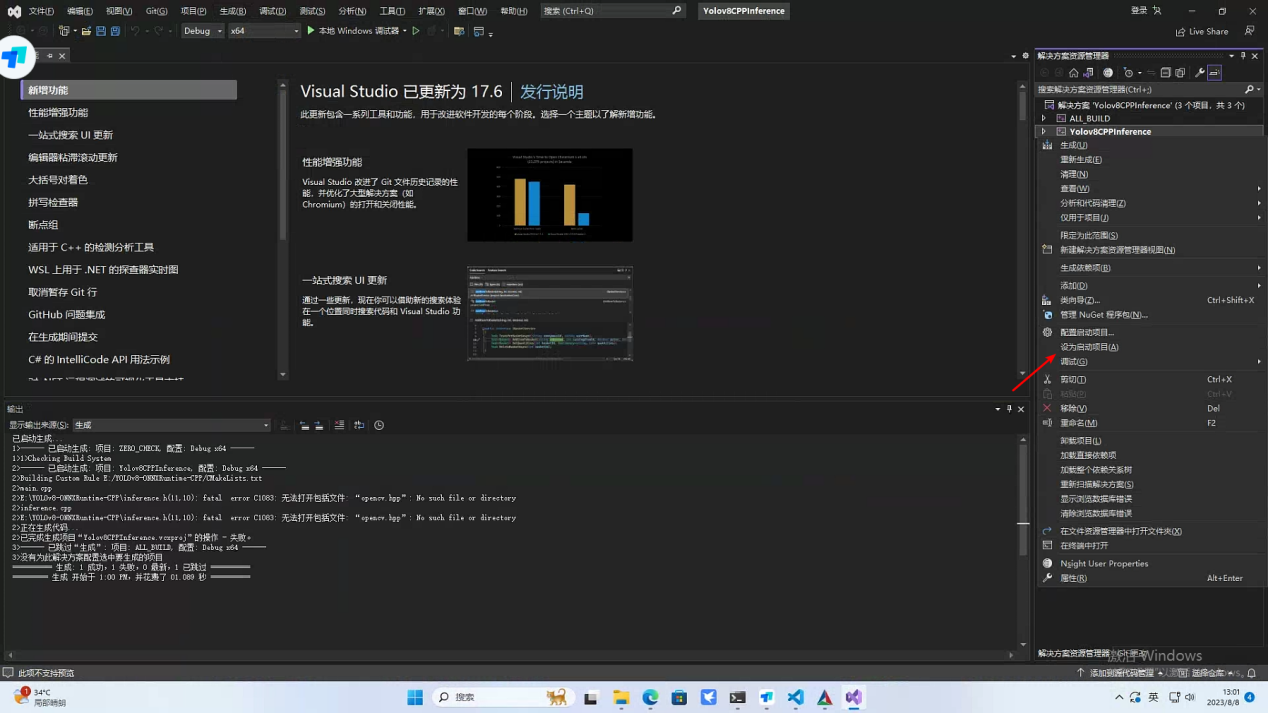


1. **配置项目依赖项**

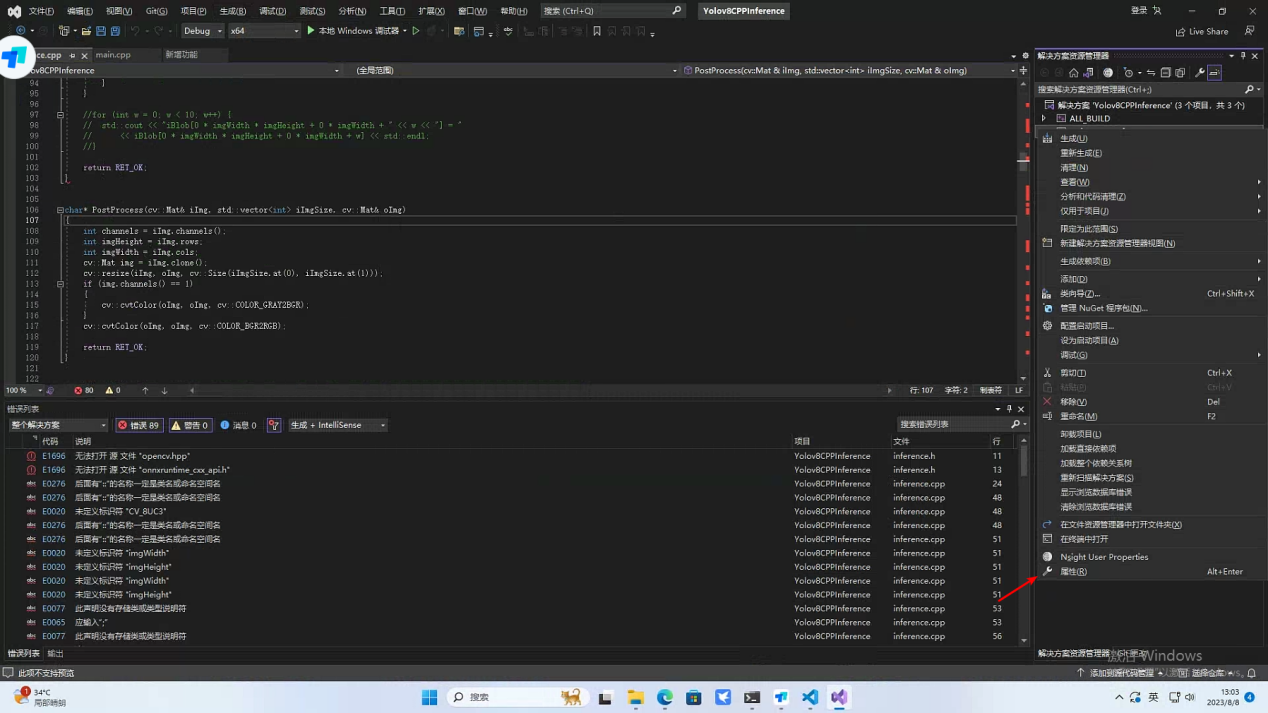
**（1）**生成解决方案；



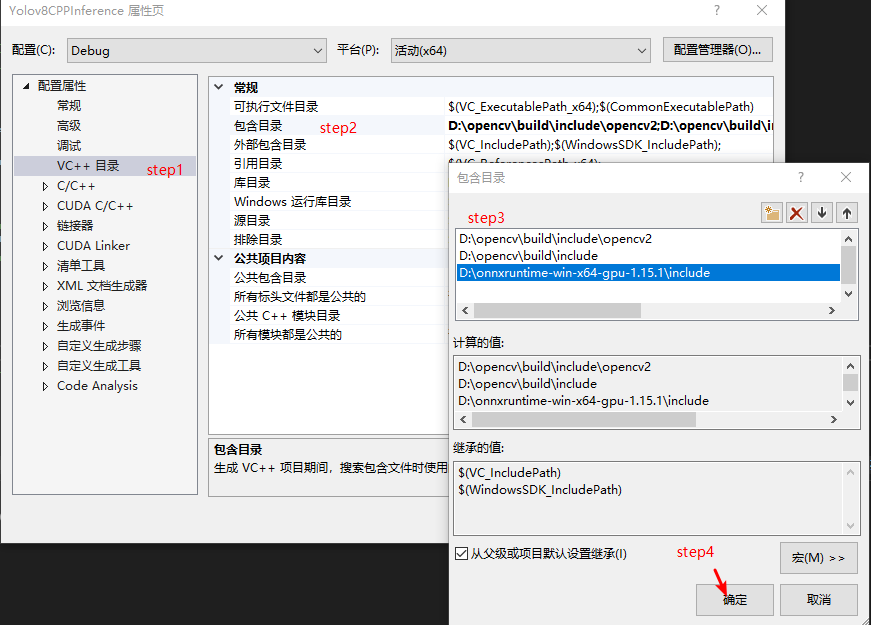
**（2）**设置三个项目中的Yolov8CPPInference为启动项目；



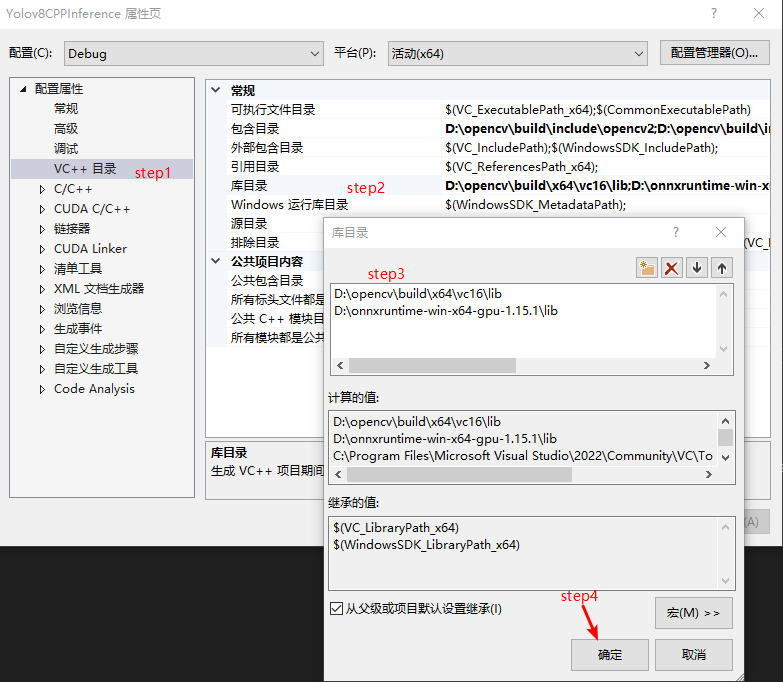
**（3）**点击属性添加项目配置项；



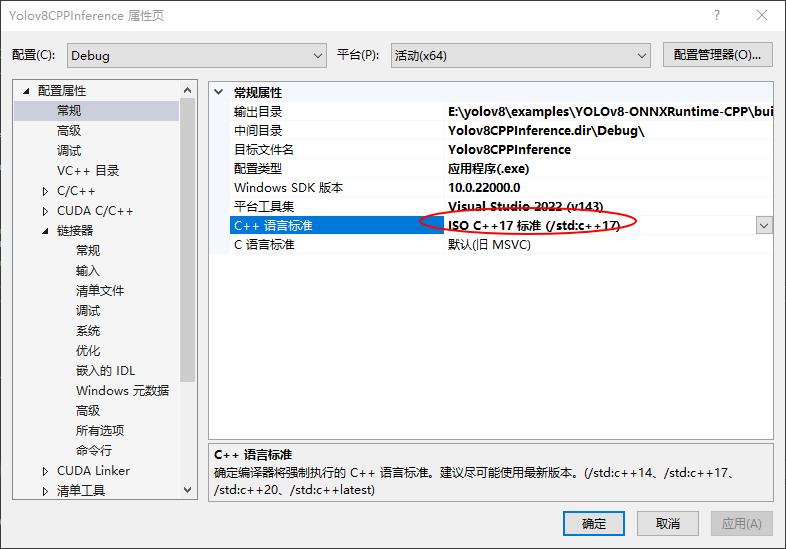
**（4）**按如下步骤添加opencv和onnxruntime的包含目录，路径替换为设备上具体的路径；

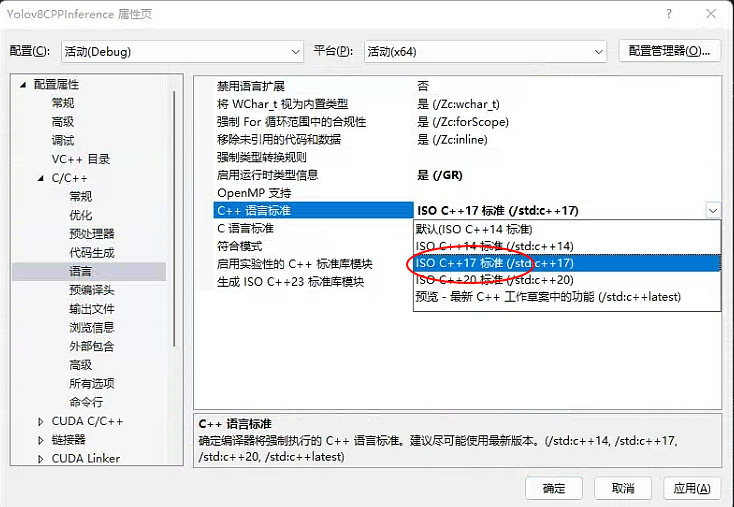


**（5）**按如下步骤添加opencv和onnxruntime的库目录，路径替换为设备上具体的路径；

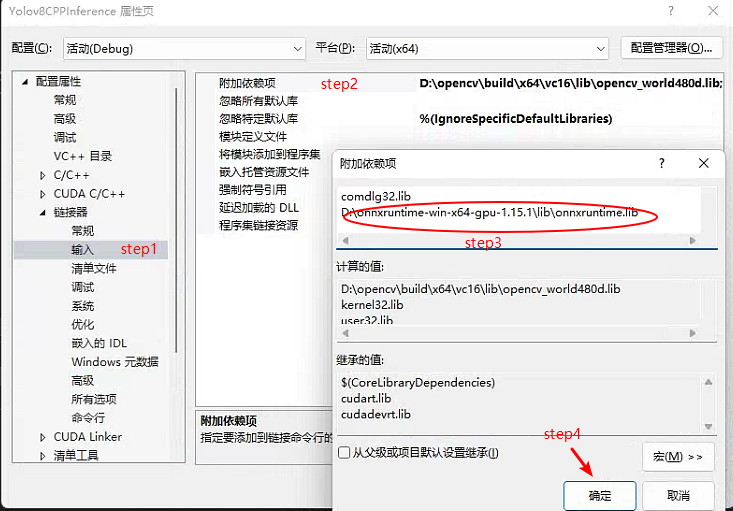


**（6）**因为官方c++推理代码使用了c++17新特性，需按下图将语言标准设定为c++17；

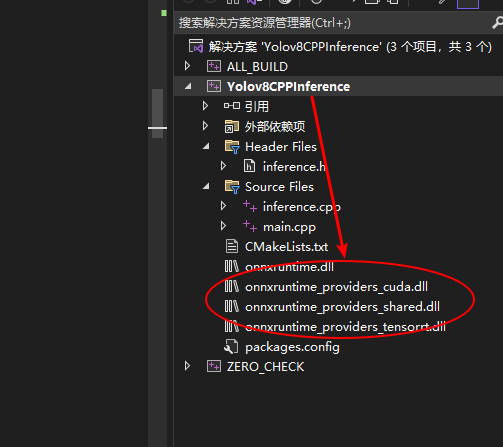


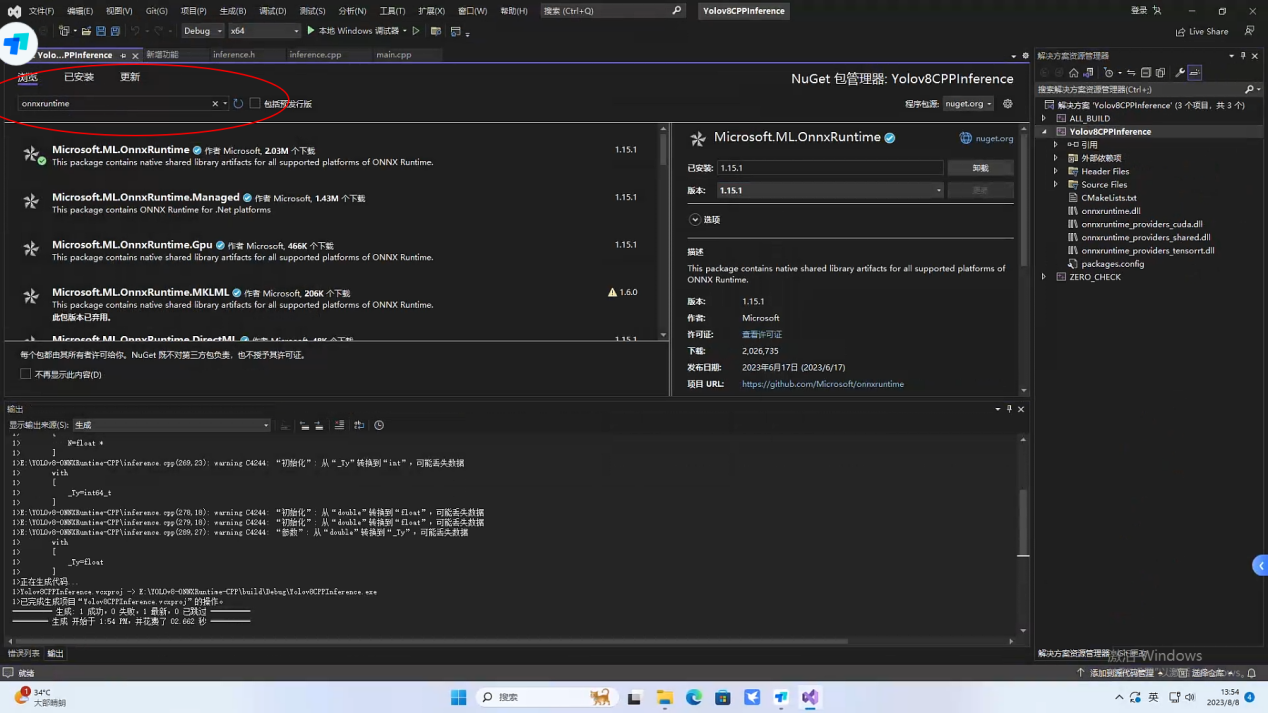


**（7）**链接器中的附加依赖项中已自动添加opencv库，需手动添加onnxruntime.lib；



**（7）**将./onnxruntime-win-x64-gpu-1.15.1/lib/目录下的四个dll文件拷入项目根目录下，或者通过“管理Nuget程序包”下载onnxruntime依赖包，否则会报版本不支持的错误：“The given version [15] is not supported, only version 1 to 7 is supported in this build.”。





**三、分类推理**

1. **yolov8-cls.pt转yolov8-cls.onnx**

需提前配置好conda环境并pip安装yolov8，完成后可使用命令将pt模型转换为onnx模型，或者创建transform.py（推荐）转换，内容如下：

from ultralytics import YOLO

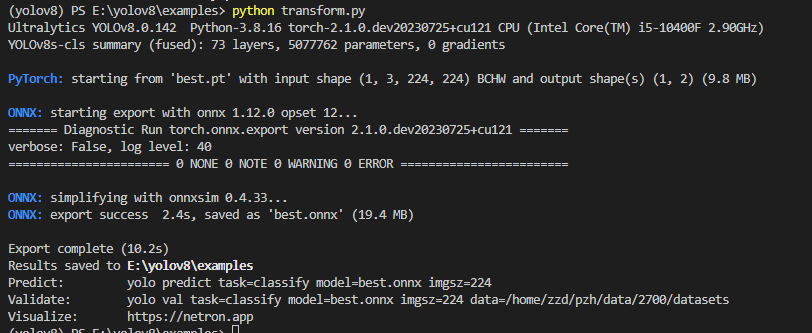
# Load a YOLOv8 model

model = YOLO("./best.pt")

# Export the model

model.export(format="onnx", opset=12, simplify=True, dynamic=False, imgsz=224)

替换原始模型路径，imgsz根据训练的pt模型输入大小更换，运行上述代码即可得到best.onnx文件。



1. **查看onnx网络架构与输入输出，便于实现预/后处理**

进入网站https://netron.app，加载转换后的onnx模型，查看yolo模型的任务类别、输入输出等重要信息，便于之后用c++实现预处理与后处理。

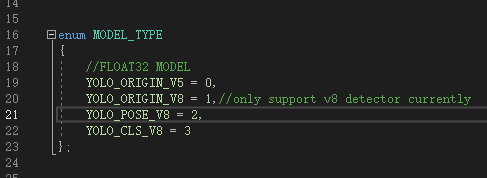


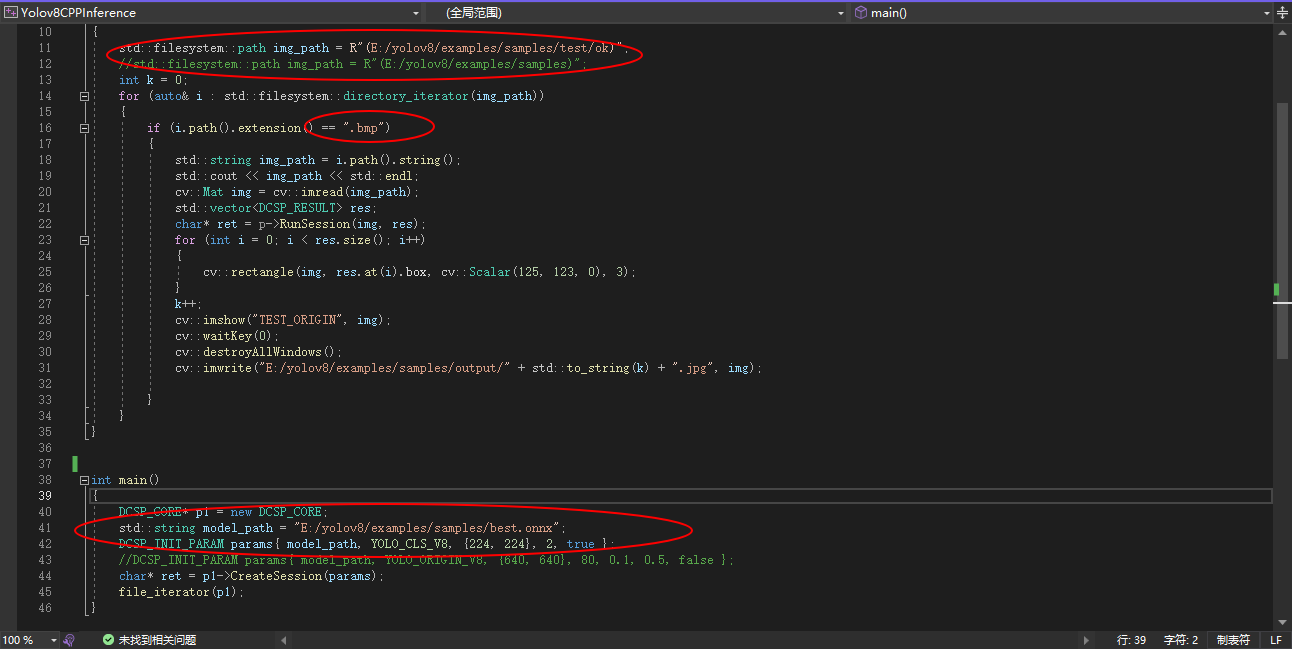
1. **修改主函数配置模型参数和测试图片路径**

进入项目的main.cpp修改要使用的模型、需要测试的图片路径与图片格式，

DCSP\_INIT\_PARAM params{ model\_path, YOLO\_CLS\_V8, {224, 224}, 2, true };

如图DCSP\_INIT\_PARAM中第二个参数为推理的任务，有分类、分割、检测三种，目前官方仅支持检测，此处改为分类YOLO\_CLS\_V8，后续自行用c++实现yolov8分类的预处理与后处理即可；第三个参数为模型的输入图像大小{H, W};第4哥参数为分类数。





1. **添加官方c++推理代码的预处理与后处理适应分类模型**

进入项目的inference.cpp文件，依照yolov8预处理的python代码添加相应的预处理，首先添加PreProcess函数对输入图像进行resize和通道变换；

char\* PreProcess(cv::Mat& iImg, std::vector<int> iImgSize, cv::Mat& oImg)

{

    cv::Mat img = iImg.clone();

    cv::resize(iImg, oImg, cv::Size(iImgSize.at(0), iImgSize.at(1)));

    if (img.channels() == 1)

    {

        cv::cvtColor(oImg, oImg, cv::COLOR\_GRAY2BGR);

    }

    cv::cvtColor(oImg, oImg, cv::COLOR\_BGR2RGB);

    return RET\_OK;

}

修改BlobFromImage函数添加使用ImageNet的均值和方差进行归一化；

template<typename T>

char\* BlobFromImage(cv::Mat& iImg, T& iBlob)

{

    int channels = iImg.channels();

    int imgHeight = iImg.rows;

    int imgWidth = iImg.cols;

    for (int c = 0; c < channels; c++)

    {

        for (int h = 0; h < imgHeight; h++)

        {

            for (int w = 0; w < imgWidth; w++)

            {

                iBlob[c \* imgWidth \* imgHeight + h \* imgWidth + w] = (std::remove\_pointer<T>::type)((iImg.at<cv::Vec3b>(h, w)[c]) / 255.0f);

            }

        }

    }

    cv::Mat paddedImage(224, 224, CV\_8UC3, cv::Scalar(0, 0, 0));

    // Copy the original image into the center of the padded image

    cv::Rect roi((224 - imgWidth) / 2, (224 - imgHeight) / 2, imgWidth, imgHeight);

    iImg.copyTo(paddedImage(roi));

    // Convert to float and normalize

    paddedImage.convertTo(paddedImage, CV\_32FC3);

    cv::Scalar mean(0.485, 0.456, 0.406);

    cv::Scalar stdDev(0.229, 0.224, 0.225);

    paddedImage /= 255.0f;

    paddedImage -= mean;

paddedImage /= stdDev;

    for (int c = 0; c < channels; c++)

    {

        for (int h = 0; h < imgHeight; h++)

        {

            for (int w = 0; w < imgWidth; w++)

            {

                iBlob[c \* imgWidth \* imgHeight + h \* imgWidth + w] = (std::remove\_pointer<T>::type)((paddedImage.at<cv::Vec3f>(h, w)[c]));

}

        }

    }

    return RET\_OK;

}

最后在TensorProcess添加后处理，此处仅输出分类后各个类别的置信度和推理各个阶段所用时间；添加部分如下(case 3是因为设置的参数YOLO\_CLS\_V8即模型类型为3)

    case 3:

    {

        //后处理

        std::cout << "ng: " << output[0] << std::endl;

        std::cout << "ok: " << output[1] << std::endl;

#ifdef benchmark

        clock\_t starttime\_4 = clock();

        double pre\_process\_time = (double)(starttime\_2 - starttime\_1) / CLOCKS\_PER\_SEC \* 1000;

        double process\_time = (double)(starttime\_3 - starttime\_2) / CLOCKS\_PER\_SEC \* 1000;

        double post\_process\_time = (double)(starttime\_4 - starttime\_3) / CLOCKS\_PER\_SEC \* 1000;

        if (cudaEnable)

        {

            std::cout << "[DCSP\_ONNX(CUDA)]: " << pre\_process\_time << "ms pre-process, " << process\_time << "ms inference, " << post\_process\_time << "ms post-process." << std::endl;

        }

        else

        {

            std::cout << "[DCSP\_ONNX(CPU)]: " << pre\_process\_time << "ms pre-process, " << process\_time << "ms inference, " << post\_process\_time << "ms post-process." << std::endl;

        }

#endif // benchmark

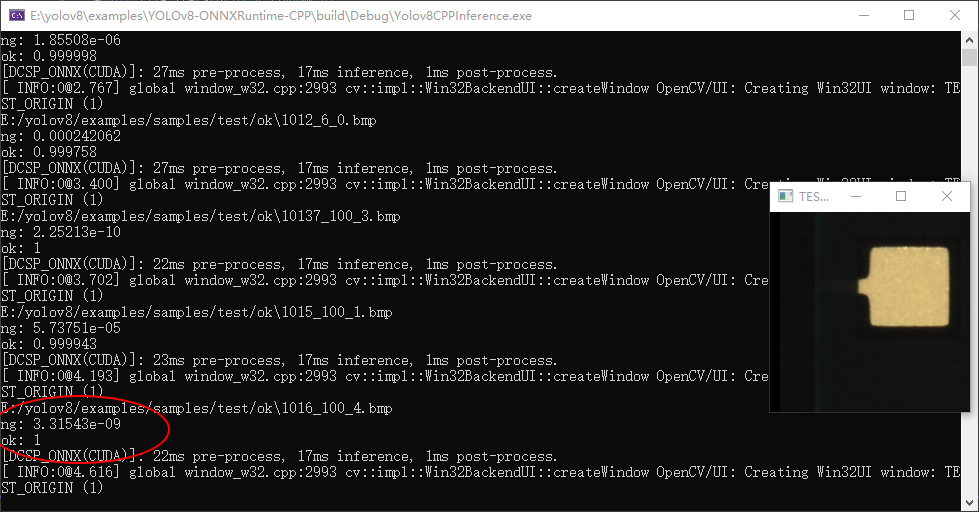
        break;

    }

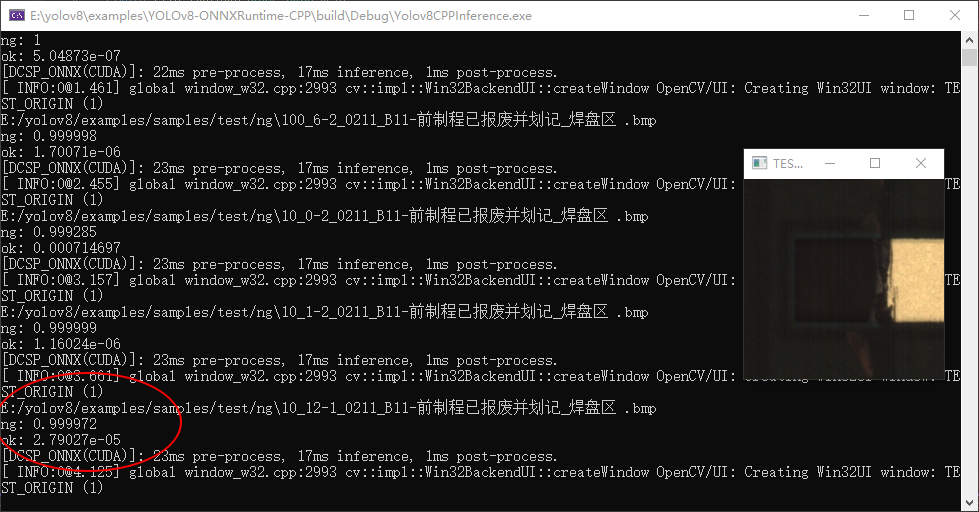
1. **测试结果**

设置好启动项目后，点击开始执行即可，在半导体缺陷复判的分类任务上，与pt模型输出结果一致。

ok测试样例：



ng测试样例：



cpu运行：DCSP\_INIT\_PARAM params{ model\_path, YOLO\_CLS\_V8, {224, 224}, 2, false };

gpu运行：DCSP\_INIT\_PARAM params{ model\_path, YOLO\_CLS\_V8, {224, 224}, 2, true};

或者直接修改inference.h头文件中CudaEnable = true的值。

模型释放：

void DCSP\_CORE::DestroySession()

{ if (session) {delete session;} }