Implementation Manual: YOLOv8-CPP Detector

Lead TA: Muhammad Haseeb (26100253)

September 15, 2025

Contents

1 Project Overview **2**
1.1 Directory Structure . . . 2

2 Implementation Tasks **2**
2.1 Preprocessor (src/preprocess.cpp) . . . 2
2.2 Inference Engine (src/infer_engine.cpp) . . . 3
2.3 Post-processor (src/nms.cpp) . . . 3
2.4 Frame Queue (src/frame_queue.cpp) . . . 3
2.5 Thread Logic (src/frame.cpp) . . . 3

3 Building and Testing **4**
3.1 Building the Docker Environment . . . 4
3.2 Entering the Docker Container . . . 4
3.3 MacOS . . . 5
3.4 Running Individual Tests . . . 5

4 Running the Final Application **5**
4.1 Building the Main Executable . . . 5
4.2 Running the Detector . . . 5

5 Submission and Grading **5**
5.1 Submission Requirements . . . 6
5.2 Grading Criteria . . . 6

1 Project Overview

The goal of this project is to build a multi-threaded C++ application that performs real-time object detection using a YOLOv8 model. The application will read from a video file or webcam, process frames in a pipeline, and save the output as a new video file with bounding boxes drawn on detected objects.

The architecture is based on a **Producer-Consumer** model:

- **Producer Thread**: Reads frames from a video source and places them into a shared, thread-safe queue.
- **Consumer Thread**: Retrieves frames from the queue and performs the computationally expensive detection pipeline (preprocess, infer, post-process).

1.1 Directory Structure

The project is organized as follows:

/headers Contains all header files (.h). **Do not modify these.**

/src Contains C++ implementation files (.cpp). **This is where you will write your code.**

/tests Contains pre-written tests for each component.

/data A suggested location for input video files.

/models Script to build .onnx model from a PyTorch model file.

2 Implementation Tasks

You must implement the C++ source files corresponding to the provided headers.

2.1 Preprocessor (src/preprocess.cpp)

Objective: Implement the Preprocessor class to prepare an image for the neural network.

Preprocessor::process(const cv::Mat& image) This function must transform a raw cv::Mat frame into a tensor (represented as a cv::Mat) suitable for the model. The required steps are:

1. **Resize**: Perform a direct, non-proportional resize of the input image to the model's required input dimensions (e.g., 640x640). This will stretch or squash the image.
2. **Normalize**: Convert the pixel values from the [0, 255] integer range to the [0, 1] floating-point range.
3. **Reformat**: Convert the image from its default OpenCV format (BGR color order, HWC layout) to the format required by the model (RGB color order, NCHW layout).

Preprocessor::getScaleAndPadding() This function is part of a different preprocessing strategy (letterboxing) and is not required for the direct resize method. You may return default values.

2.2 Inference Engine (src/infer_engine.cpp)

Objective: Implement the InferEngine class to manage the ONNX Runtime and perform inference.

InferEngine::loadModel(const std::string& model_path) This function must initialize the ONNX Runtime environment and create a session by loading the .onnx model from the given path.

InferEngine::infer(const cv::Mat& input_blob) This function must execute the model. The steps are:

1. Create an ONNX Runtime input tensor
2. Execute the session's Run() method.
3. Retrieve the output tensor from the results.
4. Wrap the output tensor's data in a cv::Mat and return it.

2.3 Post-processor (src/nms.cpp)

Objective: Implement the postprocess function to decode the model's raw output.

postprocess(...) This function takes the raw prediction matrix from the inference engine and converts it into a clean list of final detections. The logical steps are:

1. **Decode**:For each row of prediction matrix, find the class with the highest confidence score.
2. **Filter**: Discard any detections where the highest confidence score is below the conf_threshold.
3. **Scale**: For the remaining detections, extract the bounding box coordinates (cx, cy, w, h).
4. **NMS**: Perform class-wise Non-Maximum Suppression to eliminate redundant, overlapping bounding boxes for the same object class

2.4 Frame Queue (src/frame_queue.cpp)

Objective: Implement the FrameQueue class to create a thread-safe data structure.

FrameQueue::push(const cv::Mat& frame) This function must add a frame to the queue. It should be thread-safe using a std::mutex. If the queue is full, it must block until space becomes available.

FrameQueue::pop(cv::Mat& frame) This function must retrieve a frame from the queue. It must be thread-safe. If the queue is empty, it must block until a frame is pushed or the queue is closed.

FrameQueue::close() This function must set a flag to stop all operations and notify any waiting threads to unblock.

2.5 Thread Logic (src/frame.cpp)

Objective: Implement the producer and consumer thread functions. A skeleton is provided in the directory.

producer(...) This function's loop should:

1. Open the video specified by video_path.

2. Continuously read frames from the video.
3. Push each valid frame into the shared FrameQueue.
4. Stop when the video ends or the global running flag is false.
5. Close the queue before exiting to signal the consumer.

consumer(...) This function's loop should:

1. Pop a frame from the shared FrameQueue.
2. Pass the frame to the Preprocessor.
3. Pass the resulting blob to the InferEngine.
4. Make sure output matrix is valid and in correct shape for post-processing.
5. Pass the output predictions to the postprocess function.
6. Draw the final detections onto the original frame.
7. Write the annotated frame to a cv::VideoWriter.
8. Continue as long as the running flag is true or the queue is not empty.

You will have to write optimal code here and your choice of queue size will determine how efficient your code is :

3 Building and Testing

You will use Docker and the provided Makefile to create a consistent build environment and test your code.

3.1 Building the Docker Environment

First, build the Docker image. This command only needs to be run once.

```
make docker-build
```

3.2 Entering the Docker Container

To compile and test your code, you must enter the Docker container. This command mounts your current project directory inside the container at /app.

```
docker run --rm -it -v $(PWD):/app inference_engine bash
```

Inside docker first run:

```
export LD_LIBRARY_PATH=./onnxruntime-linux-x64-1.17.0/lib:$LD_LIBRARY_PATH
```

And then to create onnx model:

```
python3 convert_model.py
```

All subsequent make **commands must be run from inside this container's shell.**

3.3 MacOS

Below are the commands to build and run the Docker container for the inference engine, targeting the linux/amd64 platform. These commands assume you are running on macOS, using $(pwd) for the host path.

```
docker buildx build --platform=linux/amd64 “
    -t inference˙engine:latest --load .
```

```
docker run --rm -it --platform=linux/amd64 “
    -v ”$(pwd)”:/app inference˙engine:latest bash
```

3.4 Running Individual Tests

As you complete each component, you can test it individually. These tests will help you debug your implementation for each part of the pipeline.

```
# Test your preprocessor implementation
make test-preprocess

# Test your NMS/post-processing implementation
make test-nms

# Test your inference engine implementation
make test-inferengine

# Test your thread-safe queue implementation
make test-framequeue
```

4 Running the Final Application

Once all tests pass, you can build the main application executable.

4.1 Building the Main Executable

```
make
```

4.2 Running the Detector

Execute the compiled program with the required arguments.

```
./inference˙engine --model yolov8n.onnx --video data/sample˙video.mp4
```

The output video will be saved as output.mp4 in the project’s root directory.

5 Submission and Grading

Please follow the instructions below carefully to ensure your submission is graded correctly.

5.1 Submission Requirements

You are required to submit a single compressed file containing only the src directory.

- **Format**: YourRollNumber.zip (e.g., 26100253.zip)
- **Contents**: The zip file must contain only the src directory with your implemented .cpp files. Do not include headers, models, data, the Makefile, or any build artifacts.

The directory structure inside your submitted .zip file should be as follows:

```
Part2.zip
—
        src/
            frame.cpp
            frame_queue.cpp
            infer_engine.cpp
            main.cpp
            nms.cpp
            preprocess.cpp
        output.mp4
```

5.2 Grading Criteria

Your submission will be evaluated based on the following criteria:

1. **Correctness (Component Tests)**: The primary component of your grade will be determined by the successful compilation and passing of the provided unit tests. We will run the following commands, and each must pass:
   - make test-preprocess
   - make test-nms
   - make test-inferengine
   - make test-framequeue

   Failure to compile will result in a ZERO.
   Hidden Cases: Final marks will be based on visible + hidden cases
2. **End-to-End Functionality**: After individual tests, your entire application will be compiled and run on a confidential test video. The evaluation will be based on the quality of the generated output.mp4.
3. **Output Video Quality**: The generated output video will be manually inspected for:
   - **Accuracy**: Bounding boxes should be tightly wrapped around the correct objects.

- **Completeness**: Correct class labels and confidence scores must be displayed. **Robustness**: The application must run to completion without crashing and produce a valid, non-corrupt video file.

Code Quality: While not a direct percentage, clean, readable, and well-structured code is expected. Points may be deducted for exceptionally poor code quality or memory management issues.