# 如何用 OpenVINO™让 YOLOv8 获得 1000+ FPS 性能?

作者:武卓

YOLO 家族又添新成员了! 作为目标检测领域著名的模型家族, you only look once (YOLO) 推出新 模型的速度可谓是越来越快。就在刚刚过去的1月份, YOLO 又推出了最新的 YOLOv8 模型, 其 模型结构和架构上的创新以及所提供的性能提升, 使得它刚刚面世, 就获得了广大开发者的关 注。

YOLOv8 的性能到底怎么样?如果说利用 OpenVINO™的量化和加速,利用英特尔®CPU、集成显卡以及独立显卡与同一代码库无缝协作,可以获得 1000+ FPS 的性能,你相信吗?那不妨继续往下看,我们将手把手的教你在利用 OpenVINO™在英特尔®处理器上实现这一性能。



图 1. YOLOv8 推理结果示例

好的,让我们开始吧。

注意:以下步骤中的所有代码来自 OpenVINO Notebooks 开源仓库中的 230-yolov8-optimization notebook 代码示例,您可以点击以下链接直达源代码。 <u>https://github.com/openvinotoolkit/openvino\_notebooks/blob/main/notebooks/230-yolov8-optimization.jpynb</u>

## 第一步: 安装相应工具包及加载模型

本次代码示例我们使用的是 Ultralytics YOLOv8 模型,因此需要首先安装相应工具包。

```
1. !pip install "ultralytics==8.0.5"
```

然后下载及加载相应的 PyTorch 模型。

```
1. from ultralytics import YOLO
2.
3. MODEL_NAME = "yolov8n"
4.
5. model = YOLO(f'{MODEL_NAME}.pt')
6.
7. label map = model.model.names
```

定义测试图片的地址,获得原始 PyTorch 模型的推理结果

1. IMAGE\_PATH = "../data/image/coco\_bike.jpg"
2. results = model(IMAGE\_PATH, return\_outputs=True)

其运行效果如下

```
Ultralytics YOLOv8.0.5 🚀 Python-3.8.10 torch-1.13.1+cpu CPU
Fusing layers...
YOLOv8n summary: 168 layers, 3151904 parameters, 0 gradients, 8.7 GFLOPs
```

为将目标检测的效果以可视化的形式呈现出来,需要定义相应的函数,最终运行效果如下图所示



# 第二步: 将模型转换为 OpenVINO IR 格式

为获得良好的模型推理加速,并更方便的部署在不同的硬件平台上,接下来我们首先将 YOLO v8 模型转换为 OpenVINO IR 模型格式。YOLOv8 提供了用于将模型导出到不同格式(包括 OpenVINO IR 格式)的 API。model.export 负责模型转换。我们需要在这里指定格式,此外,我们 还可以在模型中保留动态输入。

```
1. from pathlib import Path
2.
3. model_path = Path(f"{MODEL_NAME}_openvino_model/{MODEL_NAME}.xml")
4. if not model_path.exists():
5. model.export(format="openvino", dynamic=True, half=False)
```

接下来我们来测试一下转换后模型的准确度如何。运行以下代码,并定义相应的前处理、后处理 函数,

```
1. from openvino.runtime import Core, Model
2.
3. core = Core()
4. ov_model = core.read_model(model_path)
5. device = "CPU" # GPU
6. if device != "CPU":
7.    ov_model.reshape({0: [1, 3, 640, 640]})
8. compiled_model = core.compile_model(ov_model, device)
```

在单张测试图片上进行推理,可以得到如下推理结果



### 第三步: 在数据集上验证模型准确度

YOLOv8 是在 COCO 数据集上进行预训练的,因此为了评估模型的准确性,我们需要下载该数据 集。根据 YOLOv8 GitHub 仓库中提供的说明,我们还需要下载模型作者使用的格式的标注,以便 与原始模型评估功能一起使用。

```
1. import sys
2. from zipfile import ZipFile
3.
4. sys.path.append("../utils")
5. from notebook utils import download file
6.
7. DATA_URL = "http://images.cocodataset.org/zips/val2017.zip"

    LABELS URL = "https://github.com/ultralytics/yolov5/releases/downloa

   d/v1.0/coco2017labels-segments.zip"
9.
10.OUT DIR = Path('./datasets')
11.
12.download file(DATA URL, directory=OUT DIR, show progress=True)
13.download file(LABELS URL, directory=OUT DIR, show progress=True)
14.
15.if not (OUT DIR / "coco/labels").exists():
       with ZipFile(OUT DIR / 'coco2017labels-
16.
   segments.zip' , "r") as zip_ref:
17.
           zip ref.extractall(OUT DIR)
       with ZipFile(OUT DIR / 'val2017.zip' , "r") as zip ref:
18.
          zip_ref.extractall(OUT_DIR / 'coco/images')
19.
```

接下来,我们配置 DetectionValidator 并创建 DataLoader。原始模型存储库使用 DetectionValidator 包装器,它表示精度验证的过程。它创建 DataLoader 和评估标准,并更新 DataLoader 生成的每个数据批的度量标准。此外,它还负责数据预处理和结果后处理。对于类初 始化,应提供配置。我们将使用默认设置,但可以用一些参数替代,以测试自定义数据,代码如 下。

```
    from ultralytics.yolo.utils import DEFAULT_CONFIG
    from ultralytics.yolo.configs import get_config
    args = get_config(config=DEFAULT_CONFIG)
    args.data = "coco.yml"
    validator = model.ValidatorClass(args)
    data_loader = validator.get_dataloader("datasets/coco", 1)
```

Validator 配置代码如下

```
1. from tqdm.notebook import tqdm
2. from ultralytics.yolo.utils.metrics import ConfusionMatrix
3.
4. validator.is_coco = True
5. validator.class_map = ops.coco80_to_coco91_class()
6. validator.names = model.model.names
7. validator.metrics.names = validator.names
8. validator.nc = model.model[-1].nc
```

定义验证函数, 以及打印相应测试结果的函数, 结果如下

```
print_stats(fp_stats, validator.seen, validator.nt_per_class.sum())

Class Images Labels Precision Recall mAP@.5 mAP@.5:.95

all 5000 36335 0.633 0.474 0.521 0.371
```

# 第四步:利用 NNCF POT 量化 API 进行模型优化

Neural network compression framework (NNCF) 为 OpenVINO 中的神经网络推理优化提供了一套先进的算法,精度下降最小。我们将在后训练(Post-training)模式中使用 8 位量化(无需微调)来优化 YOLOv8。

优化过程包括以下三个步骤:

```
1) 建立量化数据集 Dataset;
```

- 2) 运行 nncf.quantize 来得到优化模型
- 3) 使用串行化函数 openvino.runtime.serialize 来得到 OpenVINO IR 模型。

建立量化数据集代码如下

```
1. import nncf # noqa: F811
2. from typing import Dict
3.
4.
5. def transform_fn(data_item:Dict):
       .....
6.
7.
       Quantization transform function. Extracts and preprocess input d
   ata from dataloader item for quantization.
8.
       Parameters:
9.
          data item: Dict with data item produced by DataLoader during
   iteration
10.
       Returns:
           input_tensor: Input data for quantization
11.
       .....
12.
       input_tensor = validator.preprocess(data_item)['img'].numpy()
13.
```

```
14. return input_tensor
15.
16.
17.quantization_dataset = nncf.Dataset(data_loader, transform_fn)
```

### 运行 nncf.quantize 代码如下

```
1. quantized model = nncf.quantize(
2.
       ov model,
3.
       quantization dataset,
       preset=nncf.QuantizationPreset.MIXED,
4.
5.
       ignored scope=nncf.IgnoredScope(
           types=["Multiply", "Subtract", "Sigmoid"], # ignore operati
6.
   ons
7.
           names=["/model.22/dfl/conv/Conv",
                                                         # in the post-
   processing subgraph
8.
                   "/model.22/Add",
9.
                   "/model.22/Add 1",
10.
                   "/model.22/Add 2"
11.
                   "/model.22/Add 3"
12.
                   "/model.22/Add 4",
                   "/model.22/Add 5",
13.
                   "/model.22/Add 6",
14.
15.
                   "/model.22/Add_7",
16.
                   "/model.22/Add 8",
                   "/model.22/Add_9",
17.
                   "/model.22/Add 10"]
18.
19.
       ))
```

最终串行化函数代码如下

```
1. from openvino.runtime import serialize
```

```
3. print(f"Quantized model will be saved to {int8_model_path}")
```

```
serialize(quantized_model, str(int8_model_path))
```

运行后得到的优化的 YOLOv8 模型保存在以下路径

#### yolov8n\_openvino\_int8\_model/yolov8n.xml

```
接下来,运行以下代码在单张测试图片上验证优化模型的推理结果
```

```
1. if device != "CPU":
```

- 2. quantized\_model.reshape({0, [1, 3, 640, 640]})
- 3. quantized\_compiled\_model = core.compile\_model(quantized\_model, devic
   e)

```
4. input_image = np.array(Image.open(IMAGE_PATH))
5. detections = detect(input_image, quantized_compiled_model)[0]
6. image_with_boxes = draw_boxes(detections, input_image)
7.
8. Image.fromarray(image_with_boxes)
```

### 运行结果如下



验证下优化后模型的精度,运行如下代码:

```
    print("FP32 model accuracy")
    print_stats(fp_stats, validator.seen, validator.nt_per_class.sum())
    a
    print("INT8 model accuracy")
    print_stats(int8_stats, validator.seen, validator.nt_per_class.sum())
    )
```

得到结果如下:

FP32 model accuracy						
Class	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
all	5000	36335	0.633	0.474	0.521	0.371
INT8 model accuracy						
Class	Images	Labels	Precision	Recall	mAP@.5	mAP@.5:.95
all	5000	36335	0.634	0.473	0.519	0.369

可以看到模型精度相较于优化前,并没有明显的下降。

# 第五步:比较优化前后模型的性能

接着,我们利用 OpenVINO 基线测试工具

https://docs.openvino.ai/latest/openvino\_inference\_engine\_tools\_benchmark\_tool\_README.html 来 比较优化前(FP<sub>3</sub>2)和优化后(INT8)模型的性能。在这里,我们分别在英特尔®至强®第三代处 理器(Xeon Ice Lake Gold Intel 6348 2.6 GHz 42 MB 235W 28 cores)上运行 CPU 端的性能比 较。针对优化前模型的测试代码和运行结果如下

> # Inference FP32 model (OpenVINO IR)
>  !benchmark\_app -m \$model\_path -d CPU -api async shape "[1,3,640,640]"

FP32 模型性能:

С	File Edit View Run Kernel Tabs Settings He	lp
h.	+ 🗈 ± C	230-yolov8-optimization.ipyn × +
	Filter files hu name	E + % □ □ ► ■ C → Markdown ∨
	Filter files by flame	[Step 6/11] Configuring input of the model
	/ 230-yolov8-optimization /	[ INFO ] Model inputs:
_	Name A Last Modified	[ INFO ] Model outputs:
=	datasets 23 days ago	[ INFO ] output0 (node: output0) : f32 / [] / [1,84,8400]
	volov8n openvino int8 model 23 davs ago	[Step 7/11] Loading the model to the device
÷.	volov8n openvino model 23 davs ago	[ Step 8/11] Querying optimal runtime parameters
	Job volovil optimization inveh     20 days age	[ INFO ] Model:
	M DEADME and	[INFO] NETWORK_NAME: torch_jit
	README.md 23 days ago	[ INFO ] OPIINAL NUMBER_OF_INFER_KEQUESIS: 14 [ INFO ] NUM STREAMS: 14
	yolov8n.onnx 23 days ago	[ INFO ] AFFINITY: Affinity.CORE
	yolov8n.pt 23 days ago	[ INFO ] INFERENCE_NUM_THREADS: 56
		[ INFO ] PERF_COUNT: False
		[ INFO ] INFERENCEKELISION_HINI: < (ype: TLOBISZ >
		[Step 9/11] Creating infer requests and preparing input tensors
		[WARNING ] No input files were given for input 'images'!. This input will be filled with random values!
		[ INFO ] Fill input 'images' with random values
		[Step 10/11] Measuring performance (Start inference asynchronously, 14 inference requests, limits: 60000 ms duration)
		[ INF0 ] Benchmarking in inference only mode (inputs filling are not included in measurement loop).
		[ INFO ] First inference took 44.83 ms
		[Step 11/11] Dumping statistics report
		[ INFO ] Count: 2010 Iterations
		[ INFO ] DUFATION: 60037.39 ms
		[INFO] Latency:
		[INFO] Average 311 mc
		[TING] Min- 20.43 ms
		[ INFO ] Throughput: 434.90 FPS
		[ INFO ] INFOUGHPUT: 434.90 FPS

INT8 模型性能:

$\mathbf{c}$	File Edit View Run Kernel	Tabs Settings He	lp III III III III III III III III III I	
	+ 🗈 ± C		230-yolov8-optimization.ipyn ×     +	
-	Filter files by name	Q	B + X □ □ → ■ C → Markdown ∨	
0	/ 230-yolov8-optimization /		[INFO] output0 (node: output0) : +32 / [] / [1,84,3] [Step 5/11] Resizing model to match image sizes and given batch	
	Name 🍝	Last Modified	LINFU J MODEL DATCH SIZE: I [ TNFO ] Reshaping model: 'images': [1.3.640.640]	
:=	datasets	23 days ago	[ INFO ] Reshape model took 13.91 ms	
	volov?n ononvino int? model	22 days ago	[Step 6/11] Configuring input of the model	
*	yoloval_openvilo_inta_inidei	23 days ago	[ INFO ] Model inputs:	
	yolov8n_openvino_model	23 days ago	[ INFO ] Images (node: Images) : u8 / [N,C,H,W] / [1,3,640,640]	
	<ul> <li>230-yolov8-optimization.ipynb</li> </ul>	23 days ago	[ INFO ] output0 (nde: output0) : f32 / [] / [1.84.8400]	
	README.md	23 days ago	[Step 7/11] Loading the model to the device	
	volov8n.onnx	23 days ado	[ INFO ] Compile model took 5265.90 ms	
	D units of at	00 dour our	[Step 8/11] Querying optimal runtime parameters	
	🔄 yolovan.pt	23 days ago	[ INFO ] MODEL:	
			[INFO] NEIWORK_WARE COLL_IF	
			[INFO] NUM STREAMS: 56	
			[ INFO ] AFFINITY: Affinity.CORE	
			[ INFO ] INFERENCE_NUM_THREADS: 56	
			[ INF0 ] PERF_COUNT: False	
			[ INF0 ] INFERENCE_PRECISION_HINT: <type: 'float32'=""></type:>	
			[ INF0 ] PERFORMANCE_HINT: PerformanceMode.THR0UGHPUT	
			[ INFO ] PERFORMANCE_HINT_NUM_REQUESTS: 0	
[Step 9/11] Cre			[Step 9/11] Creating infer requests and preparing input tensors	
			[ WARNING ] No input files were given for input 'images'!. This input will be filled with random values!	
		[ INFU ] Fill input 'Images' with random Values		
			[Step 10/11] measuring performance (start interence asynchronously, so interence requests, timits: 00000 ms dura	
			[ INFO ] Denoting in inference only 57 12 ms	
			[Step 1]/1] Dumping statistics report	
			[ INFO ] Count: 87192 iterations	
			[ INFO ] Duration: 60050.88 ms	
			[ INFO ] Latency:	
			[INFO] Median: 38.38 ms	
			[INFO] Average: 38.50 ms	
			[ INF0 ] Min: 32.03 ms	
			[ INF0 ] Max: 78.47 ms	
			[ INFO ] Throughput: 1451.97 FPS	

### 已经达到了 1400+ FPS!

在英特尔<sup>®</sup>独立显卡上的性能又如何呢?我们在 Arc<sup>™</sup> A770m 上测试效果如下:

30 yalov8 e > høytertab × + → C O localhost 8888/ab/tree/230 -yolov8-optimization/230-yolov8-optimization.jpynb	
ile Edit View Run Kernel Tabs Settings Help El Launcher X ▼ 230-yolov8-optimization.jpy1X + ■ + X □ □ → ■ O → Code ∨ [INFO ] Reshape model 1 *inages*; [1,3,640,640] [INFO ] Reshape model Took 35.82 ms [Sten Gill3] Content for the model	Task Manager     File Options View     Processes Performance App history Startup Users Details Services     CPU     CPU     GPU     Intel(R) Arc(TM) A770M Graphic
<pre>[Step 0.11; Contagering input or the model [IHF0] Model inputs: [IHF0] Model outputs: [IHF0] outputs: [INF0] output0 (node: output0) : f32 / [] / [1,84,8400] [Step 7.11] Loading the model to the device [INF0] Compile model took 16385.35 ms [Step 8.11] Querying optimal runtime parameters [INF0] Model: [INF0] Model: [INF0] METMORK_IMMEE. Torch_it [INF0] METMORK_IMMEE. Torch_it [INF0] METMORK_IMMEE torch_it [INF0] METMORK_IMMEE torch_it [INF0] METMORK_IMMEE torch_it [INF0] METMORK_IMMEE torch_it [INF0] HetMORK_IMMEE torch_it [INF0] METMORK_IMMEE torch_it [INF0] Fill Creating infer requests and preparing input tensors [MAMINE] No input files user given for input 'images'!. This input will be filled with randc [INF0] Fill input 'images' with randon values [Step 9.11] Menuming performance (Start inference asynchronously, S12 inference requests, lis [INF0] Benchmarking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in Inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in Inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in Inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in Inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in Inference only node (inputs filling are not included in measurement is [INF0] Elemicharking in Inference only node</pre>	Memory 11.6/63.6 G8 (18%)               · 3D             1% - Copy             11              ////             ///
[Step 11/11] Dumping statistics report [ INFO ] Count: 6924 iterations [ INFO ] Louration: 60545.43 ms [ INFO ] Latency: [ INFO ] Median: 476.80 ms [ INFO ] Median: 476.80 ms [ INFO ] Median: 296.64 ms [ INFO ] Max: 538.02 ms [ INFO ] Throughput: 1073.97 FP5 [ ]:	GPU 1 Shared GPU memory usage 31.6 GPU GPU 2 Intel(IR) Arc(TM) A770 Graphics 0% (34 °C) Utilization Dedicated GPU memory 1% 0.7/16.0 GB GPU Memory Shared GPU memory 1.5/47.8 GB 0.8/31.8 GB GPU Temperature

也超过了 1000 FPS!

需要注意的是要想获得如此的高性能,需要将推理运行在吞吐量模式下,并使用多流和多个推理 请求(即并行运行多个)。同样,仍然需要确保对预处理和后处理管道进行微调,以确保没有性 能瓶颈。

### 第六步:利用网络摄像头运行实时测试

除了基线测试工具外,如果你想利用自己的网络摄像头,体验一下实时推理的效果,可以运行我 们提供的实时运行目标检测函数

```
1. run_object_detection(source=0, flip=True, use_popup=False, model=ov_
    model, device="AUTO")
```

获得类似如下图的效果:



### 第七步: 进一步提升性能的小技巧

• 非同步推理流水线

在进行目标检测的推理时,推理性能常常会因为数据输入量的限制而受到影响。此时,采 用异步推理的模型,可以进一步提升推理的性能。异步 API 的主要优点是,当设备忙于推 理时,应用程序可以并行执行其他任务(例如填充输入或调度其他请求),而不是等待当 前推理首先完成。要了解如何使用 openvino 执行异步推理,请参阅 AsyncAPI 教程 <u>https://github.com/openvinotoolkit/openvino\_notebooks/blob/97f25b1697ob6fe2287ca47bba6</u> <u>4f31cffg8e795/notebooks/115-async-api/115-async-api.ipynb</u>。

### • 使用预处理 API

预处理 API 允许将预处理作为模型的一部分,从而减少应用程序代码和对其他图像处理库的依赖。预处理 API 的主要优点是将预处理步骤集成到执行图中,并将在选定的设备

(CPU/GPU/VPU/等)上执行,而不是作为应用程序的一部分始终在 CPU 上执行。这将提高所选设备的利用率。更详细的预处理 API 信息,请参阅预处理教程 <u>https://docs.openvino.ai/latest/openvino\_docs\_OV\_Runtime\_UG\_Preprocessing\_Overview.ht</u> <u>ml</u>。

对于本次 YOLOv8 示例来说, 预处理 API 的使用包含以下几个步骤:

### 1. 初始化 PrePostProcessing 对象

20.from openvino.preprocess import PrePostProcessor
21.
22.ppp = PrePostProcessor(quantized model)

### 2. 定义输入数据格式

```
2. from openvino.runtime import Type, Layout
3.
4. ppp.input(0).tensor().set_shape([1, 640, 640, 3]).set_element_type(Type.u8
     ).set_layout(Layout('NHWC'))
5. pass
```

```
3. 描述预处理步骤
```

预处理步骤主要包括以下三步:

- 将数据类型从 U8 转换为 FP32
- 将数据布局从 NHWC 转换为 NCHW 格式
- 通过按比例因子 255 进行除法来归一化每个像素

代码如下:

```
1. ppp.input(0).preprocess().convert_element_type(Type.f32).convert_lay
    out(Layout('NCHW')).scale([255., 255., 255.])
2.
```

```
3. print(ppp)
```

### 4. 将步骤集成到模型中

- 1. quantized\_model\_with\_preprocess = ppp.build()
- 2. serialize(quantized\_model\_with\_preprocess, str(int8\_model\_path.with\_ name(f"{MODEL\_NAME}\_with\_preprocess.xml")))

具有集成预处理的模型已准备好加载到设备。现在,我们可以跳过检测函数中的这些预处 理步骤,直接运行如下推理

1. def detect\_without\_preprocess(image:np.ndarray, model:Model):
2. """

```
3.
       OpenVINO YOLOv8 model with integrated preprocessing inference fu
   nction. Preprocess image, runs model inference and postprocess resul
  ts using NMS.
4.
       Parameters:
5.
           image (np.ndarray): input image.
6.
           model (Model): OpenVINO compiled model.
7.
       Returns:
           detections (np.ndarray): detected boxes in format [x1, y1, x
8.
   2, y2, score, label]
       .....
9.
       output_layer = model.output(0)
10.
11.
     img = letterbox(image)[0]
12.
       input_tensor = np.expand_dims(img, 0)
13.
       input hw = img.shape[:2]
14.
       result = model(input tensor)[output layer]
       detections = postprocess(result, input hw, image)
15.
16.
       return detections
17.
18.
19. compiled model = core.compile model(quantized model with preprocess,
    device)
20.input_image = np.array(Image.open(IMAGE_PATH))
21. detections = detect without preprocess(input image, compiled model)[
   0]
22.image with boxes = draw boxes(detections, input image)
23.
24. Image.fromarray(img with boxes)
```

# 小结:

整个的步骤就是这样!现在就开始跟着我们提供的代码和步骤,动手试试用 Open VINO<sup>™</sup>优化和 加速 YOLOv8 吧。

关于英特尔 OpenVINO<sup>™</sup> 开源工具套件的详细资料,包括其中我们提供的三百多个经验证并优化 的预训练模型的详细资料,请您点击

https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html

除此之外,为了方便大家了解并快速掌握 OpenVINO<sup>™</sup>的使用,我们还提供了一系列开源的 Jupyter notebook demo。运行这些 notebook,就能快速了解在不同场景下如何利用 OpenVINO<sup>™</sup> 实现一系列、包括计算机视觉、语音及自然语言处理任务。OpenVINO<sup>™</sup> notebooks 的资源可以在 Github 这 里下载安装: <u>https://github.com/openvinotoolkit/openvino\_notebooks</u>。

### 通知和免责声明

Intel 技术可能需要启用硬件、软件或服务激活。

任何产品或组件都不能绝对安全。

您的成本和结果可能有所不同。

©英特尔公司。Intel、Intel 徽标和其他 Intel 标志是 Intel Corporation 或其子公司的商标。其他名称和品牌可能被称为他人的财产。