High-Performance Computing Project: Comparing CPU and GPU Performance for Object Detection

Avdhesh Gaur

December 16, 2024

Contents

1 Introduction

The objective of this project is to compare the performance of CPU and GPU for object detection tasks using different computational approaches. Specifically, we evaluate:

- A Haar Cascade-based object detection system running entirely on the CPU.
- A YOLOv5-based deep learning model optimized for GPU execution.

The focus is on measuring processing time, memory usage, and hardware utilization for videos of varying resolutions and evaluating their accuracy.

2 System Specifications

2.1 Hardware

- **Operating System**: Windows 11 Home
- **CPU**: Intel Core i7 (8 Cores, 3.6 GHz)
- **GPU**: NVIDIA GeForce RTX 4060, 8 GB GDDR6
- **RAM**: 16 GB DDR5
- **Storage**: 1 TB PCI-e NVMe SSD

2.2 Software

- **Python Version**: 3.10.12
- **PyTorch Version**: 2.5.1+cu124
- **OpenCV Version**: 4.10.0
- **psutil Version**: 5.9.0
- **OS (Testing)**: Ubuntu 22.04.5 LTS

3 Performance Analysis

3.1 Results Overview

Table 1: Performance Metrics for CPU and GPU Object Detection

Approach	Frames	Process Time (s)	Avg Time/Frame (s)	CPU Mem Diff (MB)	GPU Mem Diff (MB)	GPU Util (%)
CPU (Large)	1800	518.73	0.2882	1008.77	N/A	N/A
CPU (Small)	500	5.10	0.0102	99.48	N/A	N/A
GPU (Small)	496	20.77	0.0419	306.14	-344.00	97.29
GPU (Large)	1792	74.44	0.0415	792.48	-344.00	99.11

3.2 Accuracy: Haar vs YOLO

Haar Cascade Limitations:

- In high-resolution videos (3840x2160), Haar cascades failed to detect cars accurately. The algorithm produced false positives and struggled with smaller objects due to its fixed-scale scanning and reliance on grayscale features.

YOLO Advantages:

- YOLOv5, executed on the GPU, consistently detected cars across varying scales, achieving high accuracy even in high-resolution videos.

4 GPU Code Explanation

4.1 Memory Management

```
torch.cuda.empty_cache()  # Free unused GPU memory
gc.collect()  # Trigger garbage collection for CPU memory
```

Listing 1: GPU Memory Management

4.2 Batch Processing for Inference

```
def detect_objects_batch(frames):
    frames_rgb = [cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) for
       frame in frames]
    with torch.no_grad():
        results = model(frames_rgb)
        torch.cuda.synchronize()
    return results.pandas().xyxy
```

Listing 2: Batch Inference Using GPU

4.3 Real-Time Single Frame Processing

```python
def detect_objects_single(frame):
    frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    with torch.no_grad():
        result = model([frame_rgb])  # Single frame inference
        torch.cuda.synchronize()
    return result.pandas().xyxy
```

Listing 3: Real-Time Single Frame Inference

5 Pros and Cons of CPU and GPU

CPU:

- **Pros:**
  - Low memory usage for small tasks.
  - No need for additional hardware setup.
  - Suitable for lightweight applications.
- **Cons:**
  - Inefficient for large workloads.
  - Slower sequential processing.
  - Accuracy is poor for complex datasets.

GPU:

- **Pros:**
  - High speed with parallel processing.
  - Efficient for real-time and large-scale tasks.
  - Handles high-resolution videos and complex backgrounds.
- **Cons:**
  - Requires expensive hardware setup.
  - Memory overhead for small tasks.
  - Additional software dependencies.

6 Future Work

- Optimize CPU code with multi-threading and parallel processing.
- Deploy YOLOv5 models on edge devices like NVIDIA Jetson.
- Use lightweight models (e.g., YOLOv5s) for reduced GPU memory consumption.

7 Conclusion

This comparison shows that YOLOv5 on GPUs significantly outperforms Haar cascades on CPUs in both speed and accuracy. For high-resolution videos, GPU utilization reached an average of 99.11% for large workloads and 97.29% for smaller workloads, demonstrating its capability for real-time applications.