Task 01 - Build an Object detection pipeline

To accomplish Task 01, we need to build an object detection pipeline that processes videos one by one, detects objects (specifically persons and vehicles), and generates annotated videos with detections. We should also evaluate and profile the pipeline in terms of time, memory, and GPU usage, providing a comprehensive report for both average and worst-case scenarios. Additionally, we should consider how to improve the pipeline using Nvidia's tools, such as TensorRT inference, and discuss optimization considerations.

1. Designing the Pipeline:

- The pipeline processes videos one by one, ensuring sequential processing.
- It performs object detection using YOLOv2 (You Only Look Once version 2), a model suitable for identifying persons and vehicles. YOLOv2 is chosen for its real-time capabilities and good object detection performance.

2. Choice of Object Detection Model:

- YOLOv2 is chosen for its efficiency and accuracy. It is known for real-time object detection and has been widely used in applications involving persons and vehicles. It balances speed and accuracy, making it a suitable choice for this pipeline.

3. Evaluation and Profiling:

- To evaluate the pipeline's performance, we can use various profiling tools. For time profiling, we can use Python's **cProfile** or external profiling tools like NVIDIA's Nsight Systems. For memory and GPU usage, we can use system monitoring commands such as **top**, **nvidia-smi**, and **nvprof** for GPU profiling.

3. Comprehensive Report:

- The report should contain processing time for each video, the average frames per second (FPS), resolution, and number of channels (typically 3 for RGB).
- Example metrics for each video could include:
    - Video Name: person_only_video.mp4
    - Average FPS: 25
    - Resolution: 1280x720
    - Channels: 3 (RGB)
    - Processing Time: 100 seconds

4. Optimizing with TensorRT:

- To improve the pipeline using NVIDIA's TensorRT inference, consider the following steps:
    1. Model Conversion: Convert the YOLOv2 model to TensorRT format using the TensorRT toolkit.
    2. TensorRT Inference: Use the optimized TensorRT model for inference. TensorRT provides optimizations for faster inference.
    3. Batch Processing: Process multiple frames in a batch to take advantage of GPU parallelism.
    4. Precision Control: Adjust the precision of the model (e.g., FP16) to improve performance while maintaining accuracy.
    5. Streamlining: Optimize the code by reducing unnecessary operations and ensuring efficient use of GPU memory.

5. Improvement Considerations:

- While optimizing the models for performance, consider the trade-off between speed and accuracy.
- Choose appropriate TensorRT precision settings (e.g., FP16) based on the desired trade-off.
- Experiment with different batch sizes to find the optimal balance between throughput and latency.
- Minimize data transfer between CPU and GPU to reduce overhead.
- Ensure the TensorRT model is compatible with the specific GPU hardware for maximum performance gains.

Overall, with the right profiling and optimization techniques, the pipeline can achieve improved real-time object detection performance while identifying persons and vehicles in videos.

Here's a simple code structure for the implementation part of Task 01 using YOLOv2 as the object detection model:

```
# Import necessary libraries

# Path to the labeled dataset zip file

# Directory where you want to extract the dataset

# Extract the contents of the zip file

# Load the object detection model (e.g., YOLOv2)

# Open video file for processing

# Create an output video file

# Process each frame of the video

while video_has_frames:

    # Read a frame from the video

    # Perform object detection on the frame
    # Annotate the detected objects on the frame
    # Write the annotated frame to the output video
# Release video files and clean up
```

Task 02 - Working on Constrained devices

Designing the Parallel Processing Pipeline:

In Task 2, the objective is to process four videos in parallel while staying within specific hardware constraints, including 8GB of GPU VRAM, 4GB of memory, and a maximum CPU usage of 20%. Here's how the pipeline can be designed:

1. **Video Processing in Parallel:**
    - The pipeline should process multiple videos simultaneously to increase throughput.
    - This can be achieved by creating separate processing threads or subprocesses for each video.
2. **Efficient Resource Utilization:**
    - To stay within the hardware constraints, careful resource management is required.
    - Threads or subprocesses should be designed to efficiently use GPU and CPU resources, taking care not to exceed the specified limits.
3. **Batch Processing:**
    - Implement batch processing of frames to maximize GPU utilization.
    - Divide each video into batches of frames, which can be processed in parallel, and ensure that batch sizes are optimal for GPU memory.
4. **Shared Resources:**
    - Shared resources, like the GPU, should be carefully allocated among parallel threads to avoid resource contention and potential crashes.

Design Choices and Trade-Offs:

- **Concurrency**: Using threads or subprocesses for parallel video processing allows for efficient utilization of CPU cores but may require careful synchronization to prevent resource conflicts.
- **Batch Size**: Determining the optimal batch size is crucial. Larger batches improve GPU utilization but can lead to memory overflow. Smaller batches consume less memory but may underutilize the GPU.
- **Resource Management**: The pipeline should carefully manage GPU memory, ensuring that all videos and batches fit within the 8GB VRAM limit. Overshooting this limit can lead to GPU crashes.
- **Performance vs. Accuracy**: Balancing the number of threads or subprocesses to maximize throughput while maintaining accuracy is essential. Too many parallel processes can cause contention for resources, affecting accuracy.

Profiling and Staying Within Constraints:

- To profile the pipeline, use system monitoring commands, GPU profiling tools, and memory monitoring tools.
- Ensure that GPU memory, CPU usage, and system memory remain within the specified constraints during parallel processing.
- Continuously monitor and adjust resource allocation as needed to maintain resource limits.

Report and Analysis:

- The report should include processing times for each video, average FPS, resolution, and channels, similar to Task 1.
- Example metrics for each video could include:
  - Video Name: vehicle_only_video.mp4
  - Average FPS: 30
  - Resolution: 1280x720
  - Channels: 3 (RGB)
  - Processing Time: 6 seconds
- The analysis should focus on resource usage:
  - GPU VRAM: Ensure that VRAM usage remains below 8GB.
  - CPU Usage: Stay within the 20% CPU usage constraint.
  - Memory: Ensure that system memory (RAM) stays within the 4GB limit.

Improvement Considerations:

- Optimize batch sizes: Experiment with different batch sizes to maximize GPU and CPU utilization while staying within memory constraints.
- Resource-aware scheduling: Implement dynamic scheduling that adapts resource allocation based on the available resources and load.
- GPU Memory Management: Utilize GPU memory efficiently by reusing memory as needed and releasing resources promptly.
- Load balancing: Implement load balancing to ensure that all threads or subprocesses contribute equally to the processing workload.
- Asynchronous Processing: Implement asynchronous processing to overlap computation and communication, reducing idle time.

By addressing these considerations, the parallel processing pipeline can achieve higher throughput while staying within the specified hardware constraints.