CSCI-582 Project Report: Benchmarking Multi-Object Tracking with YOLOX and ByteTrack on NVIDIA Xavier NX and NVIDIA Orin AGX

Ethan Sims
Colorado School of Mines
Golden, CO, USA

Sara Larson
Colorado School of Mines
Golden, CO, USA

1 INTRODUCTION

Multi-Object Tracking (MOT) is a computer vision task for object detection in videos. It plays a critical role in application areas such as autonomous driving / vehicles, sports analytics, surveillance, and robotics. The most effective paradigm is detection-based tracking, where objects are detected in frames, and these detections are used to guide the tracking process.

2 TARGETED DOMAIN [2 PTS]

Detection and tracking algorithms work together to identify objects. In this project, the detection algorithm utilized is YOLOX. YOLO stands for "You Only Look Once," and is a family of CNN (Convolutional Neural Network) object detection models. Specifically, these models aim to balance speed and accuracy, delivering real-time performance without sacrificing detection quality. YOLOX introduces anchor-free detection, simplifying model design by not requiring predefined anchor boxes [2].

For tracking, the algorithm used is called ByteTrack. Traditionally, tracking algorithms will pick high-scoring (high-confidence) detection boxes to use in the tracking. ByteTrack, however, utilizes some of the low-scoring (low-confidence) detection boxes to improve result accuracy. In addition to higher accuracy results (as shown in [1]), ByteTrack has proven to handle occlusion and motion blur better than other methods through the inclusion of low-scoring detection boxes. It is a simple and fast method that can be introduced into existing pipelines [1].

The dataset used in this project is MOT20, which features a range of videos in highly crowded settings, focusing primarily on tracking people in challenging environments.

3 MOTIVATION: THE NEED FOR ACCELERATION [2 PTS]

The performance of detection and tracking algorithms is highly dependent on the computational capabilities of the underlying hardware. In the case of MOT, parallelization can dramatically improve performance. This parallelization is better suited for GPUs, where thousands of CUDA cores are optimized to efficiently handle pixel-wise operations. During detection, each pixel within a video frame must be processed to determine the detection boxes. By leveraging GPUs, these computations can be performed in parallel, significantly accelerating the detection process.

4 RELATED WORK [8 PTS]

Our project is based on the work done in [5]. This paper's key idea was to perform the data association step of tracking over all detection boxes rather than dropping the low-confidence detections. Specifically, the paper presented a data association algorithm called BYTE. This algorithm starts by matching existing tracklets to high-confidence detection boxes. Then, all unmatched tracklets are matched to the remaining low-confidence detections. At the end of this process, any unmatched tracklets get deleted, and any unmatched high-confidence detections are turned into new tracklets. The generality of this algorithm is a great advantage, as it allows it to be applied to a variety of existing approaches. As this is only part of the tracking process, it is fairly trivial to substitute in any detector. This allows for a lot of freedom and experimentation with various detectors such as YOLOX [2]. Moreover, the BYTE algorithm is general enough to apply to other trackers such as JDE [3].

The paper performed some analysis of ByteTrack against other existing approaches. They found that applying the BYTE algorithm to existing trackers gave noticeable accuracy improvements. ByteTrack was also compared directly to other trackers, such as DeepSort [4], and found that it outperformed all other trackers in every accuracy metric other than false positives and ID switches.

4.1 Current Limitations

As mentioned previously, ByteTrack is not the best approach for minimizing ID switches or false positives in multi-object tracking. Since it allows for low-confidence detections to affect the tracking, there is more room for error when it comes to the two previously mentioned metrics. Moreover, some experiments in [5] revealed that, while ByteTrack is a strong candidate for multi-class tracking scenarios, it did not stand out quite as much from other approaches as it did in the single-class domain.

5 PROJECT DESCRIPTION [2 PTS]

This project analyzes the acceleration effects of running YOLOX and ByteTrack on two different GPUs: the NVIDIA Jetson Xavier NX and NVIDIA Jetson Orin AGX. Three major system differences examined are: CPU Count (6 and 12, respectively), CUDA Core Count (384 and 2048, respectively), Tensor Core Count (48 and 64, respectively), and DRAM Memory Size (8GB with 51.2 GB/s bandwidth and 32GB/64GB with 204.8 GB/s bandwidth, respectively).

To deploy the YOLOX neural network model effectively, the TensorRT framework was used, which is specifically optimized for NVIDIA devices. The model's image preprocessing and object detection inference were both accelerated using CUDA, allowing the GPU to handle the bulk of the computational load.

5.1 Challenges

There are specific torch and torchvision versions which correspond to the Jetson device firmware and drivers. As such, when installing prebuilt PyTorch and TorchVision versions (using a command like "pip"), they often do not match the CUDA version of the Jetson device. This was an issue which caused major delay at the beginning of the project. To solve it, the necessary pipeline was creating a PyTorch model for YOLOX, exporting it to an open ONNX format, then exporting this to TensorRT format in order to run the model on the Jetson GPUs.

Another challenge faced in this project was regarding the NVRTC (NVIDIA Runtime Compilation Library) versions. This library is for compiling C++ CUDA code at runtime as opposed to at build-time. There were version dependency issues which arose when running the YOLOX model, specifically due to incompatibilities between CUDA versions. These were resolved by downgrading the `cuda-python` package from version 12 to 11 to match the expected runtime environment.

6 EXPERIMENT SETUP [2 PTS]

6.1 Methodology

The set up for our experiments was as follows: First, we created separate Python scripts for running the detector on a video and the tracker on the detection results. Keeping these two processes separate was an intentional choice so that we could compare the tracker and the detector directly. We then created two bash scripts (one for YOLOX detection, one for ByteTrack tracking) which would run the detector and the tracker multiple times on several different videos and resolutions, taking different measurements each time. The reason we reran the tracker and the detector for each measurement was to ensure that no noise was added by the other measurement tools. Some results were fairly noisy, so we prioritized analysis on average values and maximum values.

6.2 Metrics Measured and Comparisons Made

We aimed to assess both the YOLOX detector and the ByteTrack tracker separately based on several metrics. These metrics included: CPU and GPU utilization (as a percentage), RAM, CPU and GPU temperature, CPU and GPU power usage, and FPS/run-time. In addition, we looked into the time spent in each major step of the detector (loading, resizing, preprocessing, and inferring).

All of these metrics were gathered for several inputs across two different NVIDIA Jetson systems. We performed these measurements on 4 different input videos from the MOT20 dataset with the intention of observing how our chosen performance metrics differ depending on scene complexity. We also obtained measurements on a video from MOT20 at three varying resolutions so that we could observe how performance changes at different resolutions.

7 RESULTS [9 PTS]

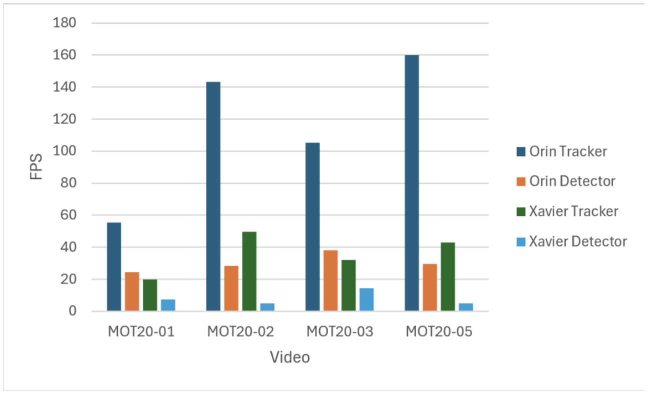

Figure 1

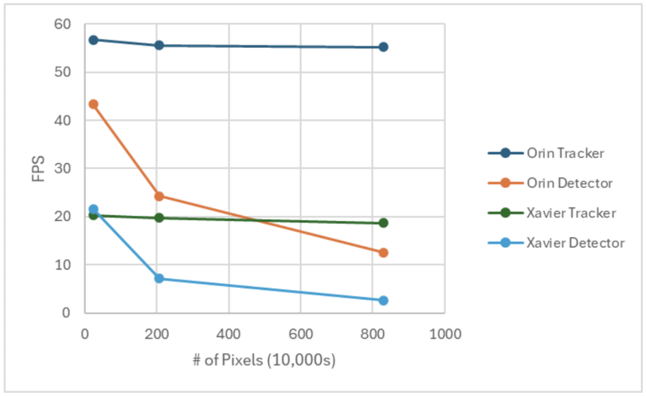

Figure 2

Figures 1 and 2 show the framerates of the tracker and the detector on both systems across different videos and different resolutions. The most obvious result here is that the tracker is consistently faster than the detector on both systems, meaning that the detector is the more significant bottleneck in the full pipeline. We can also note that unlike the detector, the tracker's speed is robust to increases in video resolution.

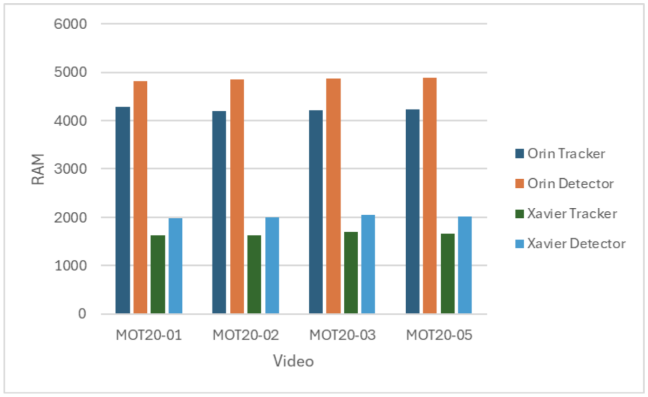

Figure 3

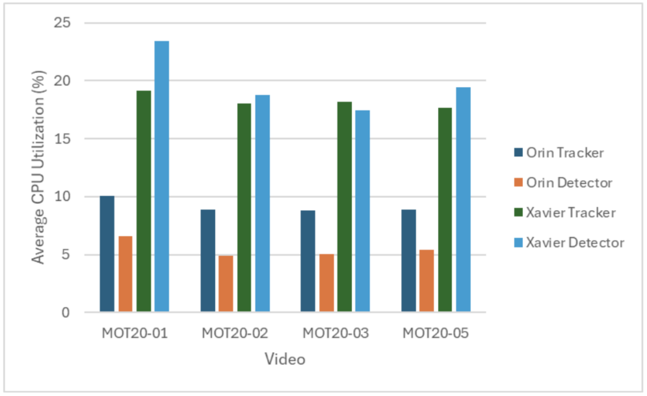

Figure 5

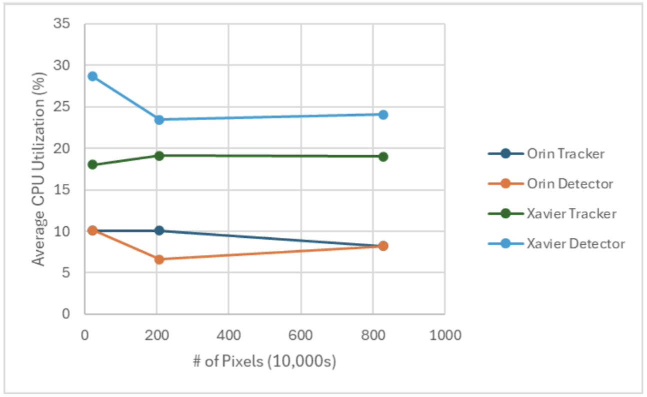

Figure 6

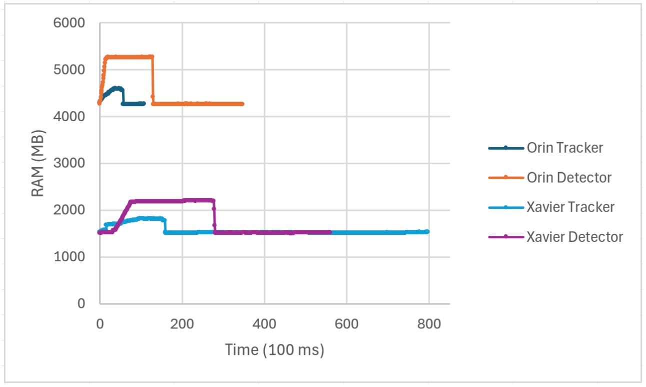

Figure 4

Figures 5 and 6 show CPU utilization for the tracker and detector on both systems. There did not seem to be any significant effects on average CPU utilization as we varied the inputs. We can note that the Xavier system had a higher CPU utilization which makes sense because its CPU was weaker. Because of this, it had to use a larger percentage of its CPU to perform all of the required computations.

Figures 3 and 4 show RAM usage of the tracker and the detector on both systems. Figure 3 shows RAM for different input videos, and figure 4 shows RAM usage over time for one video. We found no notable difference in RAM usage as we varied the input video (as seen in Figure 3) or as we varied the resolution. Figure 4 lets us see that both the tracker and the detector have a short warm-up time as they allocate memory before plateauing at some maximum memory value. Then, at some point, the memory usage drops back roughly to its starting value. We also see that the Orin system starts and ends at a much higher RAM usage which suggests that the Orin system is using much more memory on unrelated tasks.

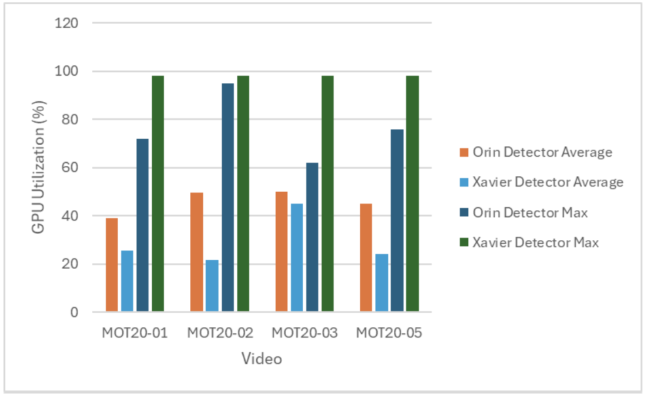

Figure 7

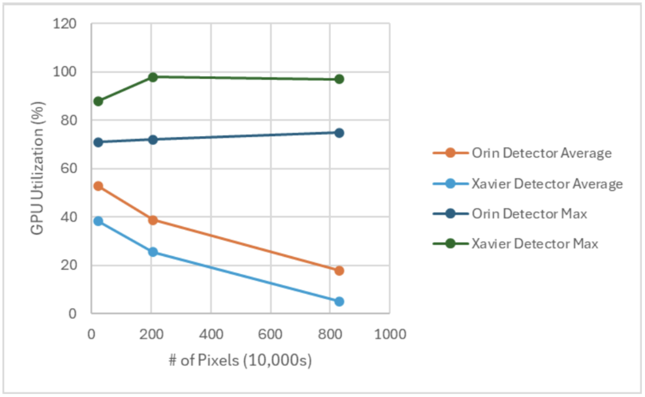

Figure 8

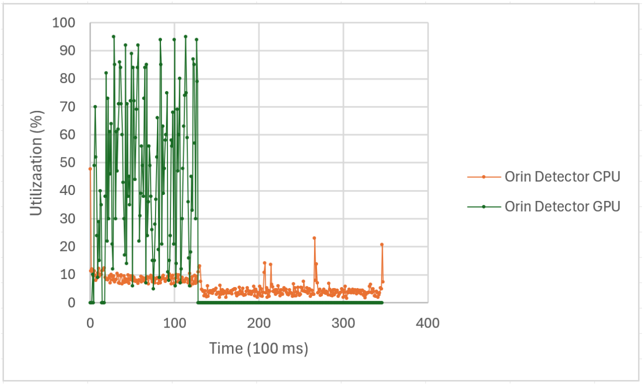

Figure 10

Figures 7 and 8 yield some seemingly counter-intuitive findings. We see that the average GPU utilization was lower on the weaker GPU of the Xavier system and that it similarly decreased as we increased the resolution. The reason for this is likely that the CPU serves as a larger bottleneck than the GPU. Although both mentioned cases result in the GPU having to be at a higher percentage of utilization for more time, the CPU bottleneck means that the GPU also has to spend much more time waiting. Since the time spent waiting for the CPU increases more than the time spent working in the GPU, we see a decrease in the average GPU utilization.

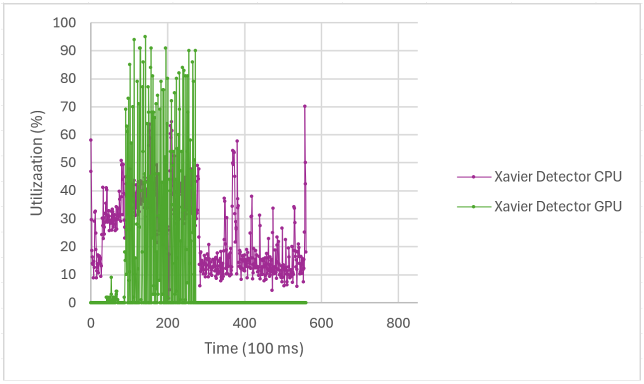

Figure 11

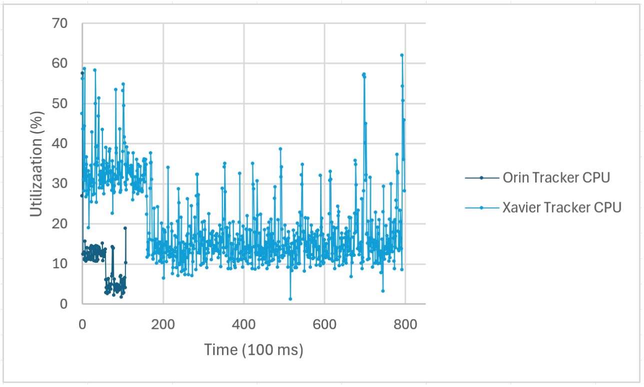

Figure 9

Figure 9 shows that the tracker's work is front-loaded, having the highest CPU utilization before dropping off notably. This is likely due to set-up costs when starting the tracker.

Figure 10 hosts some notable insights. The first thing to note is that while the CPU utilization stays relatively steady, the GPU utilization is very sporadic, frequently jumping between very high utilization and very low utilization. We can also note that the GPU utilization is generally much higher than the CPU utilization throughout the first chunk of the runtime. However, after a certain point, the GPU stops being used and the CPU decreases its utilization as well. This drop-off is similar to the ones seen in Figures 4, 9 and 11. All of these drop-offs imply that the major inference part of the detector finishes at these points, leaving only slow, sequential computations to finish.

Comparing figure 11 to figure 10, we can see that there is a significant increase in CPU utilization on the Xavier system over the Orin system. This reinforces the idea suggested by figures 7 and 8 that the CPU is a larger bottleneck than the GPU.

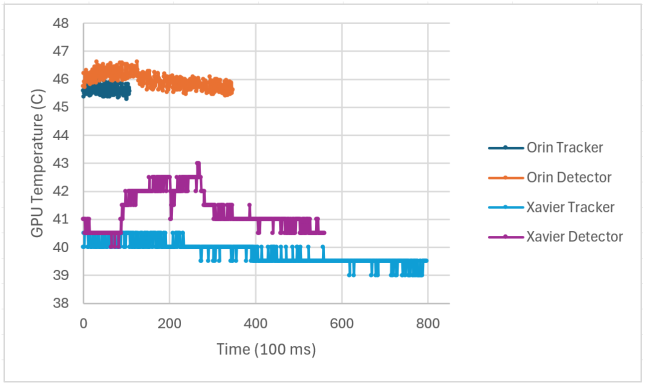

Figure 12

Figure 12 shows the GPU temperature over time for both systems. Since the tracker does not use the GPU, the lines representing the tracker temperature serve as a baseline for our observations. There is not much to be gleaned from these measurements that we have not already seen, but this figure does reinforce the idea of a dropoff in GPU usage after a certain point in the execution of the detector.

We also examined CPU temperature, but there were no significant findings.

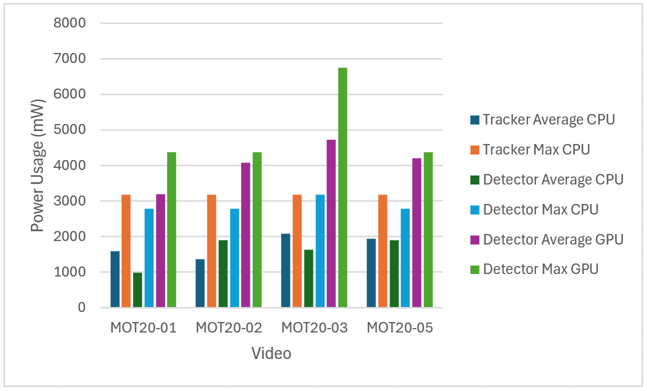

Figure 13

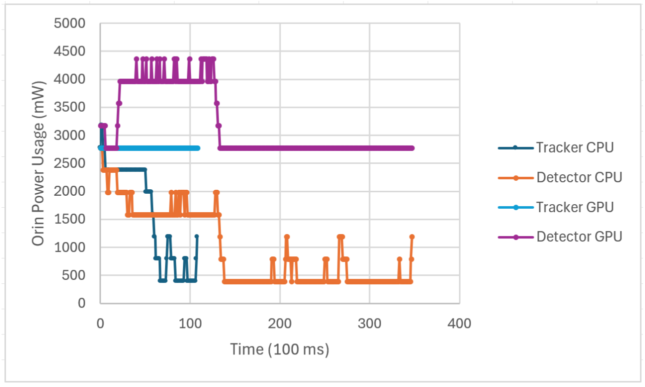

Figure 14

Unfortunately, we were unable to obtain power measurements from the Xavier system, so figures 13 and 14 only concern the Orin system. Figure 13 shows that the tracker and detector both use roughly the same amount of power for the CPU parts of their computations, but the detector's GPU power usage stands far above both CPU power usages.

Figure 14 further supports the idea of a drop-off in computational intensity in both the tracker and the detector. Another interesting note is that the detector seems to increase its GPU power usage after the first few seconds, but it decreases its CPU power usage over that same time.

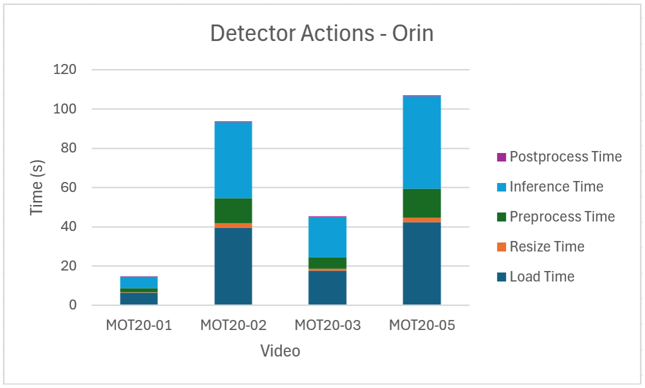

Figure 15

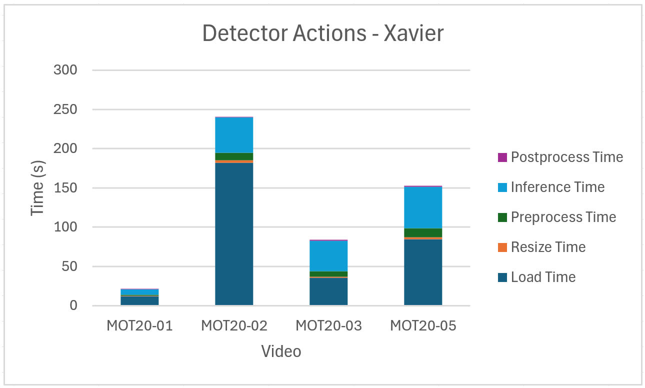

Figure 16

Figures 15 and 16 show that loading and inference are by far the biggest two tasks for the detector on both systems across all videos. This makes sense as memory tends to serve as a bottleneck in many computational domains, and inference is the most computationally intensive task in the detector. We can note that on the weaker Xavier system, both load and inference times become even more significant. This suggests that there may be further speedups that can be achieved by simply adding more computational power.

8 LINK TO SOURCE AND OTHER RESOURCES [3 PTS]

Here is the public GitHub repository which contains the scripts used, generated engines, and results of the experiment:

https://github.com/sara-l333/multi-object-tracking.git

See the README.md for information on other repositories and utilities used.

9 PROJECT/CLASS TAKEAWAY [2 PTS]

One key takeaway from this project is the importance of variety in measurements to accurately assess performance of some hardware or application. In order to identify trends and understand their underlying causes, it is important to look at many different metrics on a variety of inputs. When optimizing code / applications, there is no silver bullet; examining different metrics and seeing which factors play major roles in the optimization is key.

We have also obtained from this class many takeaways that can be generalized to almost any hardware and application. One such takeaway is the importance of parallelization when it comes to accelerating a computation. By assigning simple computations across many CUDA cores, while generalization is not optimized, speed is dramatically improved. In application areas such as computer vision (our project domain) and computer graphics which consist of numerous pixel-wise operations, the power of the GPU is essential. Another takeaway has to do with limitations imposed by bottlenecks such as sequential computations and memory accesses. Our project results demonstrated how loading from memory comprised a majority of the Jetson device time, showcasing how this is a consistent bottleneck in computing.

One final important takeaway from this project and this class is the use cases of different hardware. Specifically, how some hardware architectures (such as CPUs and GPUs) are very generalizable, but not very efficient for specific tasks. But, some other hardware architectures (like domain-specific FPGAs) can be tailored to some specific computations to perform a task very efficiently. There is a spectrum when it comes to computing devices, ranging from very generalizable and slow to very rigid and fast. By recognizing this, projects can be optimized based on specific needs.

These takeaways could help us in our future careers by helping us to accelerate certain tasks we may be trying to implement. Knowing how to find and understand performance trends, general tactics for speeding up computations, and the importance of using the right kind of hardware are all very important when it comes to optimizing any computational task.

10 FEEDBACK [+1 PTS (GOES TO PARTICIPATION)]

The projects were a great way to introduce students to many different domains of computation while enforcing the importance of speed-up techniques such as parallelization in all of these domains. Moreover, the project provided a great introduction to the full pipeline of obtaining many different measurements to assess the performance of some software.

The only downside of this project is the freedom of choice in the application area. Because each project is fairly different, it is understandable that there is no road map which the professor can provide to complete this project. Without this, it was very difficult to get the project running initially. Many hours were spent on compatibility issues, even after contacting the instructor and PhD students for assistance. This flexibility of the project allows students to pursue areas of study / interest, but it does make getting everything up and running quite difficult.

REFERENCES

[1] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. 2022. YOLOX: Exceeding YOLO Series in 2021. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops (Lecture Notes in Computer Science, Vol. 13681)*. Springer, 1–18. https://doi.org/10.1007/978-3-031-20047-2_1

[2] Viso Suite. [n. d.]. YOLOX: You Only Look Once X Explained - Real-Time Object Detection. https://viso.ai/computer-vision/yolox/. Accessed: 2025-04-22.

[3] Zhongdao Wang, Liang Zheng, Yixuan Liu, Yali Li, and Shengjin Wang. 2020. Towards Real-Time Multi-Object Tracking. arXiv:1909.12605 [cs.CV] https://arxiv.org/abs/1909.12605

[4] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. 2017. Simple online and realtime tracking with a deep association metric. In *2017 IEEE International Conference on Image Processing (ICIP)*. 3645–3649. https://doi.org/10.1109/ICIP.2017.8296962

[5] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. 2022. ByteTrack: Multi-Object Tracking by Associating Every Detection Box. arXiv:2110.06864 [cs.CV] https://arxiv.org/abs/2110.06864