End-to-End Delay Analysis and Optimization of Object Detection Module for Autonomous Driving

Wootae Jeon
Graduate School of
Automotive Engineering
Kookmin University, Korea
wjsdnxo@kookmin.ac.kr

Kyungtae Kang
Department of Computer
Science and Engineering
Hanyang University, Korea
ktkang@hanyang.ac.kr

Jong-Chan Kim
Department of Automobile
and IT Convergence
Kookmin University, Korea
jongchank@kookmin.ac.kr

Abstract

To develop a safe autonomous driving system, the delay from camera to output of its object detection module should be thoroughly analyzed and optimized. However, surprisingly little attention has been paid to this end-to-end delay of DNN-based object detectors. To the best of our knowledge, this study is one of the first attempts to analyze the internal architecture of Darknet YOLO object detector in terms of its end-to-end delay. Based on the analysis result, we propose an optimized system architecture by our modified pipeline strategy that satisfies the object detector's needs in general. Our empirical evaluation shows that the end-to-end delay drops to 229 ms (from 1,335 ms) on our evaluation platform, which is an 83% reduction. Note that we only modify the system architecture and do not change the DNN architecture itself; hence there is no penalty on the detection accuracy. Our demo is available at https://youtu.be/n3pr3s09Fs4.

1 Introduction

For safe autonomous driving, a vehicle's camera-based perception should detect hazardous on-road obstacles as early as possible to reduce the risk of collision. In that sense, the delay from camera capture to detection should be thoroughly analyzed and optimized. In this regard, lots of studies try to reduce *inference delay* by developing light-weight *deep neural network* (DNN) architectures. On the contrary, however, surprisingly little attention has been paid to *end-to-end delay* optimization, which includes not only the inference delay but also other delay components such as the time taken from the image capture by the camera to the start of the inference engine.

Table 1: End-to-end delay and frame rate measurement results

		YOLOv3	YOLOv2	YOLOv3-tiny	YOLOv2-tiny
Jetson TX2	DELAY(ms)	3,710	1,881	328	317
	FPS	1.62	3.22	20.6	21.1
Jetson AGX Xavier	DELAY(ms)	1,335	761	99	198
	FPS	4.48	8.33	30.2	29.7
Drive PX2	DELAY(ms)	1,206	590	122	108
	FPS	4.9	10	31.2	31
PC with 2080ti GPU	DELAY(ms)	199	76	83	80
	FPS	28.6	30.2	30.3	30.2

Table 1 shows our measurement results for the end-to-end delay and frames per second (fps) of various versions of Darknet YOLO (You Only Look Once) object detector [1, 2, 3] on four different

Machine Learning for Autonomous Driving Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Nvidia graphics processing unit (GPU)-based hardware platforms. Note that Darknet YOLO is the most well-known reference implementation of object detection in the research community. Moreover, it has been readily used for developing real autonomous vehicles to be deployed on the road [4, 5]. As shown in the table, besides the fact that the delays are significantly affected by the GPU hardware performance and specific DNN architectures, another important finding is that the end-to-end delay is too long in relation to the corresponding fps values. For example, let's take a look at the Jetson AGX Xavier and YOLO v3 case. Since its fps is 4.48, we can naively expect the end-to-end delay to be approximately 223 ms, which is less than one-fourth of one second. However, the actual measurement result of the end-to-end delay is surprisingly high, that is 1,335 ms, which is almost six times compared to the expected one. Starting from this observation, our research questions are given in the following:

- What is happening inside the object detection system within the end-to-end delay?

- How can we reduce this end-to-end delay, and by how much?

To answer the first question, we first analyzed the internal architecture of Darknet YOLO v3 and thoroughly measured every delay component. From the investigation, we found out that Darknet is currently employing a multithreaded four-stage pipeline architecture, which is optimized for multicore systems with the aim of maximizing its frame rate, while sacrificing the end-to-end delay. In this architecture, the reason behind the long delay is the misbalanced pipeline architecture with different-length stages; the pipeline cycle time should be aligned to the longest one, which causes significant idle times to short-length stages. Also, we have found that between the camera driver and Darknet, there was an unnecessary queuing delay of considerable length that was caused by an inappropriate queuing strategy.

To answer the second question, we first removed the unnecessary queuing delay by eliminating the queue between the camera driver and Darknet. This simple hack reduced the average delay on the Nvidia Jetson AGX embedded platform from 1,335 ms to 438 ms, which was a 67% reduction. Second, we empirically searched for the optimal pipeline architecture that could minimize the end-to-end delay. The resulting optimal architecture was a two-stage pipeline architecture with an offset optimization technique. With this optimization, the end-to-end delay was even further reduced to 229 ms, which was an 83% reduction compared with that of the original architecture. While adapting the four stages into the final two-stage architecture, cycle time increase was inevitable, which affects the frame rate. However, the impact on the frame rate was minimal, that is, from 4.48 to 4.3 (4% decrease) compared to the end-to-end delay reduction.

Related Work. In the recent deep learning literature, there are many studies that aim to develop faster deep neural network architectures. For example, to target efficient inference on embedded systems, single-stage object detectors have been developed [6, 7]. Also, to reduce the computational load, techniques such as fixed-point quantization [8], model compression [9, 10], and neural networks for mobile systems [11, 12, 13] have been proposed. They share the same goal as our work of reducing the inference delay. To achieve this goal, the above studies tried to advance neural network architectures; however, our study does not try to make a better neural network architecture. Instead, we investigate the neural network inference system from the system's perspective to eliminate unnecessary delays within the end-to-end delay. This way, we hope to extract the best possible performance for a given neural network architecture. The other studies tried to develop a more efficient inference system by exploiting the internal (generally unknown) behaviors of the Nvidia GPU drivers [14]. Recently, they modified the internal architecture of a Darknet YOLO object detector to improve real-time performance [15] on Nvidia Drive PX2 embedded systems. Although their work and our research use the same code base to improve real-time performance, their objective was to maximize the throughput, that is, the number of concurrently supported cameras, slightly sacrificing the delay. On the contrary, our work tries to reduce the end-to-end delay assuming only a single camera.

The remainder of this paper is organized as follows: The next section gives the analysis result of Darknet's internal architecture and clarifies the reason why we have such a long end-to-end delay. In Section 3, we determine an alternative pipeline architecture with a minimized end-to-end delay. Finally, Section 4 concludes this paper.

2 End-to-end delay analysis

This section explains the internal architecture of Darknet YOLO v3 from the perspective of its end-to-end delay. We had a number of reasons for choosing Darknet as the base reference implementation. One reason is that it is written purely in the C language with minimal dependencies, which makes it very easy to port from a high-end desktop to a minimal embedded system. The other reason is Darknet is being used in practice by many autonomous driving systems like Autoware and Apollo. The repository of the baseline Darknet source code is at `https://github.com/pjreddie/darknet`.

2.1 Four stages of object detection

This subsection explains the four stages of object detection from camera capture to the display of the object detection result.

Capture. For the object detection module to perceive an external scene, the scene image first enters through the camera lens, making an image pattern on the CCD or CMOS sensor, and the digitized image is finally sent to the operating system's kernel driver memory through a communication bus, which is USB in our experiments. Usually, image capture occurs continually at the camera's predefined frequency. For our camera, the frequency is 30 fps.

Fetch. Next, the object detection module fetches the image from the driver's memory to its memory buffer. Specifically, Linux's Video4Linux framework uses streaming I/O to directly map the driver memory to Darknet's address space. Thus there is no actual memory copy. Instead only the pointer is passed between the driver and Darknet. Additionally, at this stage, the resizing is done, which changes the size of the input image so that it is acceptable by the backend neural network.

Detect. This stage feeds the resized input image to the neural network which runs on the GPU. First, the image is copied from the host memory to the GPU device memory. The neural network code written in the CUDA programming language with the pretrained neural network parameters executes the forward propagation process to find relevant objects in the input image. The result is composed of a set of detected object classes, e.g., car and bicycle, and their locations in the input image. Finally, the result is copied to the host memory.

Display. Finally, the result of the object detection is visualized by drawing bounding boxes around the detected objects on top of the input image. One could argue that the display task is not necessary for production vehicles. However, in most autonomous vehicles, the visualization of how the vehicle is recognizing the environment is crucial to making the passenger feel confident that the autonomous driving system is working correctly and is traveling in the right direction. For this reason, we decided to include the display stage in the experiment.

Assuming the above stages happen consecutively, the average end-to-end delay D can be given simply as

$$D = d_{capture} + d_{fetch} + d_{detect} + d_{display}, \tag{1}$$

where each d_* represents the average delay component for each stage. Each d_* was believed to be very close to the actual execution times because there was no other workload in the experimental platform. However, this statement is not always guaranteed since we are using Linux with its fair scheduling policy instead of a real-time operating system. Thus, we use the term *delay* instead of *execution time*. Besides the end-to-end delay, frame rate or fps is also a crucial performance metric. Let us define the average cycle time C instead of fps, which is the inverse of fps or frequency. C is defined as the time distance between two consecutive image captures, which can also be simply calculated as

$$C = d_{capture} + d_{fetch} + d_{detect} + d_{display}. \tag{2}$$

Note that C is inherently the same as D in the sequential architecture. Figure 1 shows this simple sequential architecture. In the figure, we have four image frames from i to $i + 3$. Each image frame goes through the four stages, which are capture, fetch, detect, and display. After completing all the stages, the next image frame is captured, and the four stages are completed consecutively.

In Figure 1, a small time gap is depicted between the beginning of the capture stage and the image arrivals. When the capture stage begins, there is a wait for the next frame according to the camera's fps. The precise definition of the end-to-end delay should be from when the image arrives at the driver buffer. However, for the simplicity of explanation, the end-to-end delay is depicted from the very beginning of the capture stage in our figures.

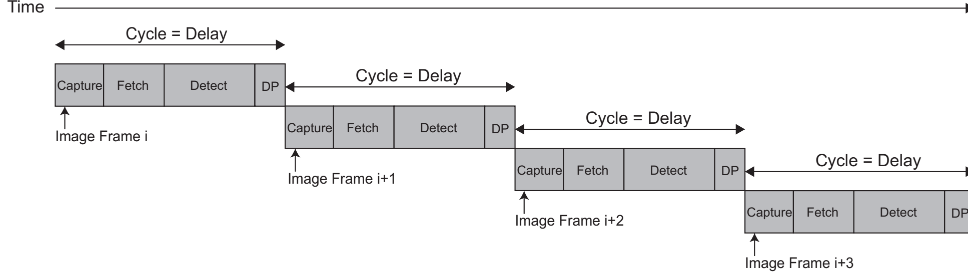

Figure 1: Sequential architecture (DP = display)

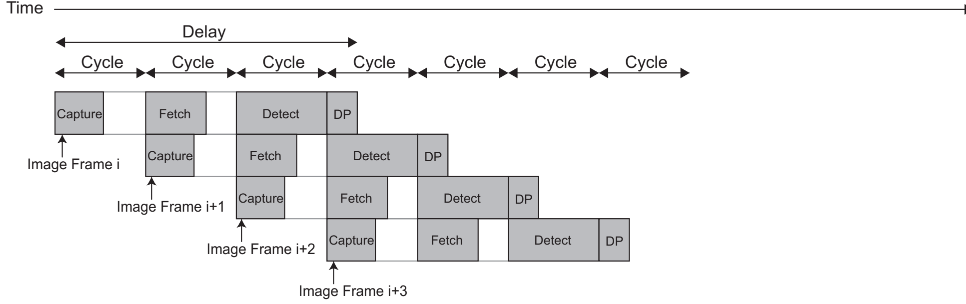

Figure 2: Four-stage pipeline architecture (DP = display)

2.2 Multithreaded pipeline architecture

By looking into Darknet's internal thread architecture, we found that Darknet employs a multithreaded pipeline architecture where there are separate threads for each of the capture, fetch, detect, and display stages. These threads are designed to run in parallel on multicore hardware to maximize the fps. Figure 2 shows an ideal pipeline architecture. In the figure, four threads are looping within a cycle time, which is a popular thread architecture called *fork-join* model. Each image frame should experience four pipeline cycles to finish a single object detection job. The cycle time C is given as the longest value of $\{d_{capture}, d_{fetch}, d_{detect}, d_{display}\}$. In other words,

$$C = \max\{d_{capture}, d_{fetch}, d_{detect}, d_{display}\}. \tag{3}$$

In the figure, for example, the detect stage is the longest, and all the other stages should loop following this common cycle time. Then the average end-to-end delay D is given as the sum of the three cycle times plus the delay caused by the display stage which is at the end of the pipeline, that is,

$$D = 3 \times C + d_{display}. \tag{4}$$

From the above observation, we understand that object detection can attain a very high fps with an unreasonably long end-to-end delay in a pipeline architecture. However, even with this explanation, we still cannot tell why the end-to-end delay is more than six times longer than the cycle time.

Among the four stages, the fetch, detect, and display stages are implemented in Darknet. However, the capture stage is performed by the operating system kernel driver and OpenCV library [16]. The fetch thread calls the driver function `ioctl`, which dequeues an image frame from the camera driver queue. This is a typical *producer-consumer* model. The camera driver produces image frames into the camera driver queue, and Darknet consumes the image at the head of the queue. In this model, if Darknet can consume the image frames at a rate above the camera's fps, there will no additional delay caused by the queue. However, if the consumption is slower than the production speed, the queue eventually fills up and the driver can no longer put a fresh image frame into the queue.

Let the queue length be denoted by N_q. It then takes $N_q \times C$ time for the new image at the tail to reach the head of the queue. After the fetch thread retrieves the image frame, the remaining cycles continue, that are, fetch, detect, and display. Thus the average end-to-end delay D can be given as

$$D = N_q \times C + 2 \times C + d_{display} = (N_q + 2) \times C + d_{display}, \tag{5}$$

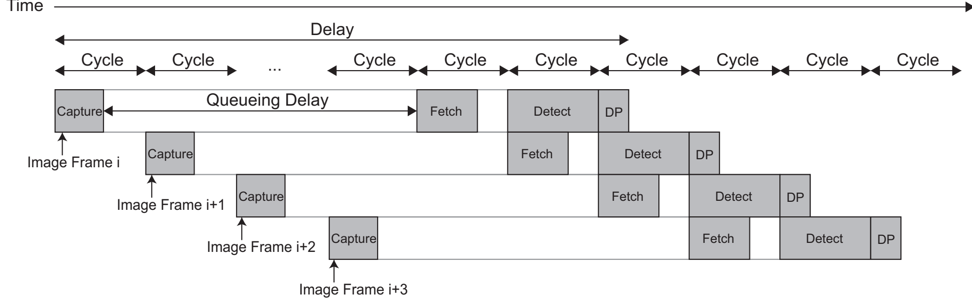

Figure 3: Four-stage pipeline architecture with kernel driver queue (DP = display)

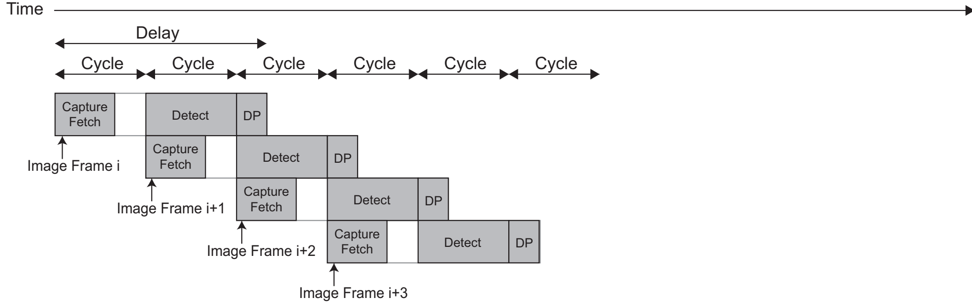

Figure 4: Synchronous fetch architecture without kernel driver queue (DP = display)

assuming $C = \max\{d_{capture}, d_{fetch}, d_{detect}, d_{display}\}$. The default driver queue length is $N_q = 4$. With this observation, we can see the reason why the end-to-end delay is more than six times longer than the cycle time in our measurement results in Table 1. Figure 3 shows this pipeline architecture with the kernel driver queue.

3 End-to-end delay optimization

3.1 Eliminating unnecessary queueing delay

To reduce the end-to-end delay, the first approach is to eliminate the unnecessary queuing delay in the kernel driver queue. To do so, we combine the capture stage and the fetch stage into a single capture/fetch thread such that the fetch thread retrieves the image frame from the camera in its own execution context. Figure 4 shows this synchronous fetch architecture. With this enhancement, the end-to-end delay D is reduced to

$$D = 2 \times C + d_{display}. \tag{6}$$

Note that even with the original Darknet architecture, if the consumption (fetch) is faster than the production (capture), it performs just like the synchronous fetch architecture. Thus, the original architecture might work well with a high-speed GPU. However, with a low-performance embedded system, we cannot expect this scenario. Therefore, it is a natural design choice to eliminate the queue between the capture and the fetch stages.

3.2 Modified pipeline architecture

While carefully looking at Figure 4, we can see repeated overlapped executions of the display thread and the detect thread. Another critical finding from our measurement study is that the parallel execution of the display and detect threads badly hurts each other's execution delays, since they use common GPU resources. Thus, our next design change is to remove this overlap by combining the detect thread and the display thread into a single thread.

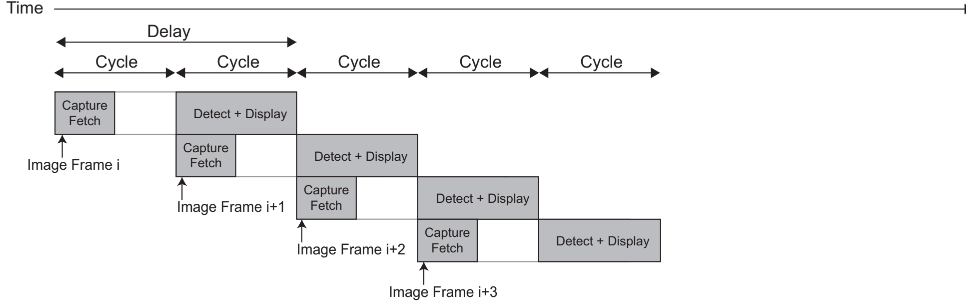

Figure 5: Two-stage pipeline architecture

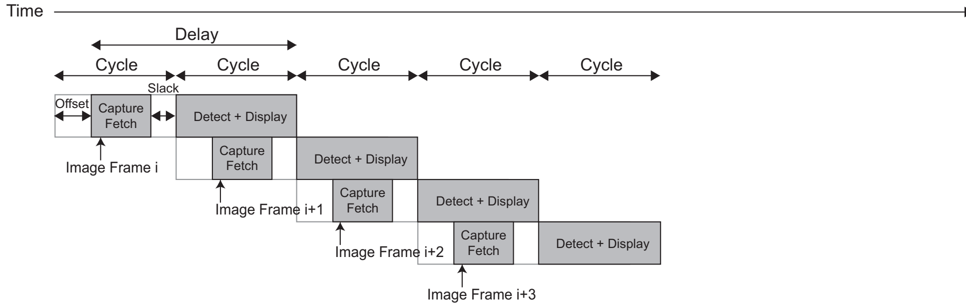

Figure 6: Two-stage pipeline architecture with offset optimization

Figure 5 shows our new two-stage pipeline architecture with the combined detect and display stage, where the end-to-end delay becomes

$$D = 2 \times C. \tag{7}$$

In the figure, note that the cycle time looks a little bit lengthened compared to the previous architectures since we now have a combined detect/display stage, which is naively expected to have its length equal to the sum of the two stages. However, we have a counter-intuitive result about the cycle time after combining the two stages. Before combining them, d_{detect} = 223 ms and $d_{display}$ = 15 ms. Thus the combined stage is expected to have its length of 238 ms. However, it is only increased from 223 ms to 228 ms, which is a very minor increase. The reason is that the two stages no longer compete for the shared GPU resource. Thanks to the elimination of this resource conflict, the detect stage's length is decreased from 223 ms to 216 ms, and the length of the display stage is decreased from 15 ms to 12 ms. As a result, the final end-to-end delay is slightly decreased from 461 ms to 456 ms after combining the two stages. Note that we have a very slight penalty on the cycle time, anyway.

3.3 Offset optimization

In Figure 5, we still see unnecessary time gaps between the capture/fetch stage and the detect/display stage. In this subsection, we try to remove this time gap. To do so, Figure 6 introduces two new notions, which are *offset* and *slack*. By controlling the offset, we try to release the capture/fetch thread sometime later than the beginning of the cycle. Then, increasing the offset eventually reduces the slack, which is the unnecessary time gap between the end of the capture/fetch thread and the beginning of the next cycle. Then the end-to-end delay is reduced to

$$D = 2 \times C - \theta_{capture+fetch}, \tag{8}$$

where $\theta_{capture+fetch}$ is the offset of the capture/fetch stage. Then we can simply expect the optimal end-to-end delay at the point when the slack touches zero. If we further increase the offset, the capture/fetch thread will finish even after the detect/display thread, which will eventually increase the cycle time and the delay. Note that the cycle time is defined as

$$C = \max\{d_{capture+fetch} + \theta_{capture+fetch}, d_{detect+display}\}. \tag{9}$$

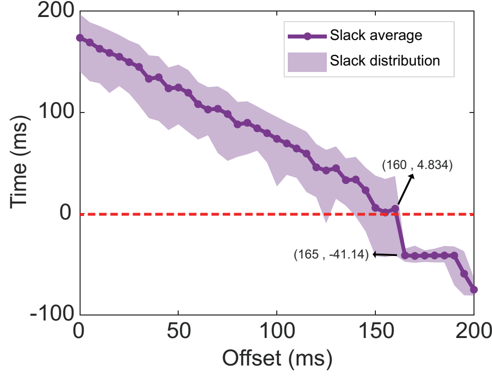

Figure 7: Slack times with varying offset values

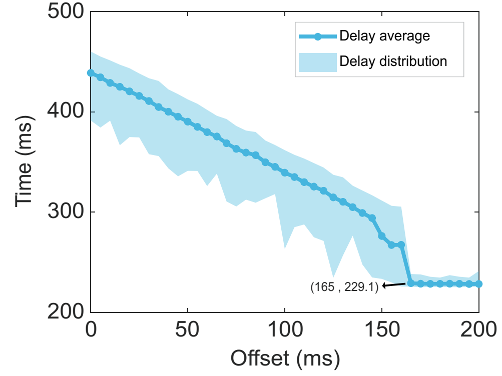

Figure 8: End-to-end delays with varying offset values

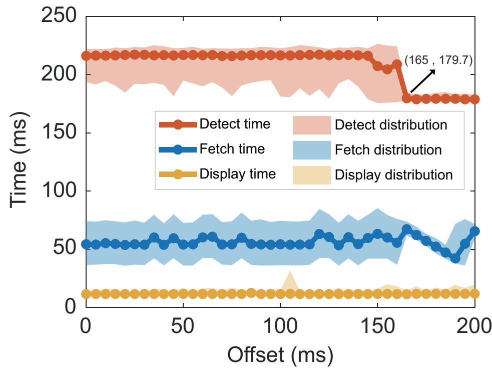

Figure 9: Execution delays with varying offset values

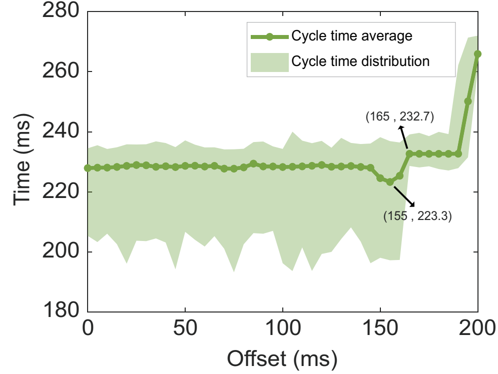

Figure 10: Cycle times with varying offset values

From the above assumption, let us define the problem as an optimization problem which minimizes D with a free variable $\theta_{capture+fetch}$. Although the problem looks fairly simple, what makes the problem challenging is that $d_{capture+fetch}$ and $d_{detect+display}$ also vary with $\theta_{capture+fetch}$ due to the shared resource conflict caused by the overlapped execution of the two threads. To empirically solve the problem, we measured the slack times, end-to-end delays, and cycle times with varying offset values from 0 ms to 200 ms at 5 ms intervals. The measurement was conducted 1,000 times for each offset value. Additionally, the individual delays of detect, fetch, and display stages were also measured. Note the fetch stage itself includes the capture stage.

Figure 7 shows the actual measurement result of slack times with varying offset values. The shaded area shows the distribution of the repeatedly measured results while the line with dots is the average values. In the figure, the slack reaches zero right after the offset 150 ms. After the offset 160 ms, the slack becomes negative, which means the cycle time is extended by that value by the finishing time of the capture/fetch thread. Figure 8 shows that the end-to-end delay is linearly decreasing according to the increasing offset before the slack reaches zero, that is, until the offset 145 ms. After the offset 145 ms, the end-to-end delay is rapidly decreasing until when it becomes the optimal value 229.1 ms at the offset 165 ms.

This rapid decrease is due to the rapid decrease of the detect stage's execution time between the offset 160 ms and 165 ms as shown in Figure 9. Note that, at the offset 165 ms, its slack time is -41.14 ms as shown in Figure 7, which means the capture/fetch thread has less overlapped execution with the detect/display thread. This minimized overlapped execution has a significant effect to the detect/display thread's execution time since it incurs less shared resource conflict. Figure 10 shows the cycle times with varying offset values, which is constant until the offset 145 ms. Between the offset 150 ms and 160 ms, the cycle time is a little bit less than the initial cycle time. It is also due to

the decreased execution time of the detect stage. At the offset 165 ms, the cycle time is 232.7 ms, which is marginally longer than the initial cycle time 223 ms.

From the above experiment, we conclude that the optimal end-to-end delay is 229.1 ms at the offset 165 ms. Although it has a minor effect on the cycle time and also the frame rate, it is marginal compared to the 83% reduction of its end-to-end delay.

3.4 Evaluation

Our evaluation hardware platform is Nvidia Jetson AGX Xavier, which has 16 GB of RAM with an 8-core ARM CPU and a 512-core Volta GPU. For the camera, Logitech USB camera C920 with 30 fps frequency was used. As our software platform, we used Nvidia Ubuntu Linux-18.04, Jetpack-4.2.2, and custom compiled OpenCV-3.3.1.

As the baseline implementation, Darknet source code from `https://github.com/pjreddie/darknet` was used. From the source code, we modified the internal architecture trying various system configurations and pipeline architectures. To measure the end-to-end delay, each image frame's meta data was tagged with the exact time it arrives at the driver buffer. For the comparison, we specifically implemented the following three architectures:

- The unmodified architecture with the four-stage pipeline and the kernel driver queue as in Figure 3, which is denoted as *Original*.
- The modified architecture after removing the unnecessary kernel driver queue as in Figure 4, which is denoted as *SyncFetch*.
- The final architecture with the two-stage pipeline with the offset optimization as in Figure 6, which is denoted as *OurOptimal*.

Figure 11 shows the average end-to-end delays and cycle times after processing 1,000 real-time camera images in a highway driving scenario for the above three architecture. Note that *SyncFetch* dramatically reduces the end-to-end delay from 1,335 ms to 438 ms comapred to *Original*. This is due to the elimination of the unnecessary queueing delay between the camera driver and Darknet. Also, *OurOptimal* was able to further reduce it down to 229 ms by our modified pipeline architecture. Although *OurOptimal* shows little decrease in its frame rate, the effect (about 0.2 fps) is minimal. A visual comparison between *Original* and *OurOptimal* is available at `https://youtu.be/n3pr3s09Fs4`.

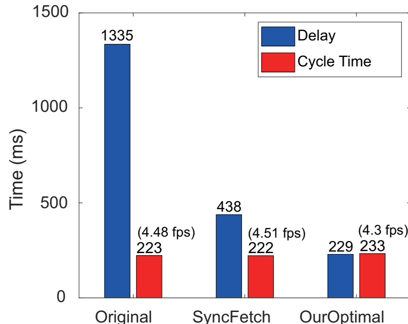

Figure 11: Evaluation results

4 Conclusion

Our study simply began by looking at the first installation of Darknet YOLO object detector, which was believed to be very fast. However, the first impression was it's so slow. By saying slow, we have two different meanings; one is the time lag from the real world through the camera to the output of the object detection, which is called the end-to-end delay. The other is how fast the object detection is looping, that is, the cycle time or the frame rate. By a preliminary measurement study, we concluded that we have a good enough frame rate considering the hardware performance. Meanwhile the end-to-end delay is too long considering the frame rate. With this motivation, this paper presents a detailed measurement and analysis study of the internal software architecture of Darknet. Based on that, we developed an enhanced pipeline architecture that exhibits an optimal end-to-end delay, which proved to be an 83% reduction on our evaluation platform.

Although we developed a promising object detection architecture, one limitation is that we considered only the average delays and did not take the worst-case scenario into consideration. In the autonomous driving application, however, the worst-case scenario is important to guarantee the safety of the system. Thus, in our future work, we plan to consider the worst-case delays.

Acknowledgments

This work was supported partly by the Korea Evaluation Institute Of Industrial Technology (KEIT) grant funded by the Ministry of Trade, Industry and Energy (MOTIE) (20000316, Scene Understanding and Threat Assessment based on Deep Learning for Automatic Emergency Steering) and partly by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2014-0-00065, Resilient Cyber-Physical Systems Research). Jong-Chan Kim is the corresponding author.

References

[1] J. Redmon, "Darknet: Open source neural networks in c." `http://pjreddie.com/darknet/`, 2013–2016.

[2] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[3] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[4] T. E. Choe, G. Chen, W. Zhang, Y. Guo, and K. W. Tsoi, "Perception of low-cost autonomous driving." 2019 US Frontiers of Engineering Symposium. `https://www.naefrontiers.org/194948/Paper`, 2019.

[5] "Autoware.ai." `https://gitlab.com/autowarefoundation/autoware.ai`, 2019.

[6] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[8] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 2849–2858, JMLR.org, 2016.

[9] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$0.5mb model size," *arXiv:1602.07360*, 2016.

[10] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[11] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, (New York, NY, USA), pp. 82–95, ACM, 2017.

[12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017. cite arxiv:1704.04861.

[13] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[14] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 104–115, Dec 2017.

[15] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J. Frahm, "Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 305–317, April 2019.

[16] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.