Data-parallel Real-Time Perception System with Partial GPU Acceleration for Autonomous Driving

Sol Ahn*, **Seungha Kim***, **Ho Kang, and Jong-Chan Kim**
Graduate School of Automotive Engineering, Kookmin University, Korea
`{solahn00, hailee98, rkdghrk12, jongchank}@kookmin.ac.kr`

Abstract

Real-time perception systems should maximize system performance by efficiently utilizing given computing resources. To this end, many perception systems employ the multithreaded pipeline architecture to leverage its task-level parallelism. In this architecture, however, imbalanced pipeline stages cause significant pipeline stalls. Moreover, limited task-level parallelism prevents us from fully utilizing the often more than a dozen CPU cores, which are common in recent SoCs. With this observation, we shift from the aforementioned task-parallel architecture to data-parallel architecture. Instead of assigning pipeline stages to each CPU core, we dispatch sequential sensor data arrivals across all the CPU cores in a round-robin manner. By exploiting this temporal parallelism, our perception architecture achieves frame rates that scale with the number of CPU cores, yielding near-optimal perception delays. Additionally, we apply partial GPU acceleration instead of conventional full GPU acceleration that is too biased towards delay optimizations. In contrast, our perception architecture selectively accelerates only a portion of given DNNs, offering flexibility in providing both frame rate-optimal and delay-optimal system configurations.

1 Introduction

Real-time perception systems should maximally utilize the given computing resources to provide the maximum system performance, targeting the following two performance metrics: (i) *frame rate* and (ii) *perception delay*. The frame rate indicates the system's throughput, while the perception delay indicates the input-to-output latency. Both have crucial impacts on mission-critical applications like autonomous driving, where many perception systems employ the multithreaded pipeline architecture that exploits the task-level parallelism in perception tasks. This *task-parallel* architecture assigns each pipeline stage to dedicated CPU cores running in parallel to improve the frame rate, however, at the cost of additional pipeline stall delays by imbalanced pipeline stages [1]. Moreover, if the degree of task-level parallelism is less than the number of given CPU cores, the remaining CPU cores cannot be utilized.

Based on the above observations, this study introduces a perception system architecture designed to maximize the utilization of available computing resources, thereby maximizing the system performance. Our basic idea is to migrate from the *task-parallel* architecture to the *data-parallel* architecture to exploit the data-level parallelism instead of the limited task-level parallelism. The original meaning of data-level parallelism is to (i) partition input data, (ii) process each of them in parallel, and (iii) compose an output (i.e., *spatial* parallelism). We extend this notion along the time domain by dispatching repeatedly arriving sensor data to CPU cores in a round-robin manner (i.e., *temporal* parallelism). This way, the frame rate becomes scalable to the number of CPU cores while minimizing the perception delay.

*These authors contributed equally to this work.

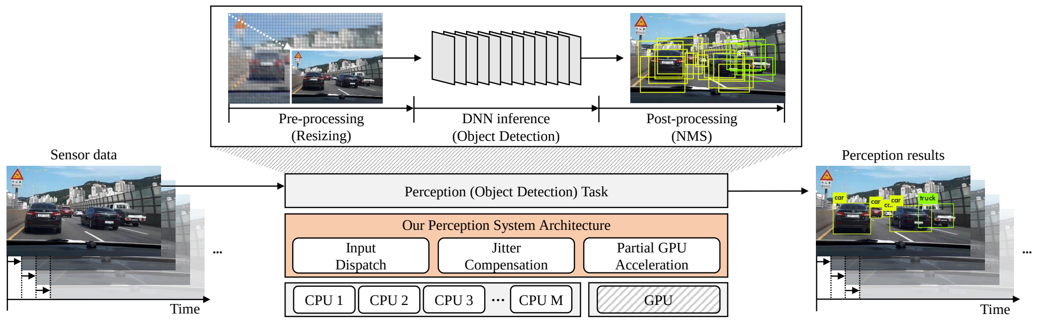


Figure 1: DNN-based perception system architecture.

Fig. 1 shows a typical perception system architecture, where multiple CPU cores and a GPU are given as computing resources. We assume a deep neural network (DNN)-based perception task that consists of three stages of pre-processing (e.g., resizing), inference (e.g., object detection), and post-processing (e.g., non-maximum suppression (NMS)). The perception task should efficiently utilize the given computing resources. For that, our perception system architecture provides three key features of (i) input dispatch, (ii) jitter compensation, and (iii) partial GPU acceleration.

Input dispatch. Our input dispatcher receives periodically arriving sensor data (e.g., camera images) while monitoring the independent progress of perception tasks on the CPU cores. Then a new sensor data arrival is assigned to an available CPU core in a round-robin manner, trying to maximally utilize the CPU cores for the maximum frame rate. This way, the frame rate can easily scale to the number of CPU cores. Once the system's achievable frame rate is decided, the same sensor data arrival rate can be processed without any additional delays like the pipeline stall delays in the task-parallel architecture.

Jitter compensation. Since the data-parallel architecture makes all the CPU cores busy at the same time, memory contention delays are unavoidable to some extent. Moreover, due to the low predictability of such contentions, the perception task's execution time is with random variations (i.e., jitter). To minimize this unpredictability, we employ the logical execution time (LET) concept [2, 3] to emulate a constant execution time by adding a busy loop that waits until a predefined (measured in our case) worst-case execution time (WCET). In this manner, we can remove most jitters. Please note that the exact WCET analysis or measurement method is out of this study's scope.

Partial GPU acceleration. Recent systems on chip (SoCs) typically include an integrated GPU, which can be utilized to accelerate the DNN-based inference stage. Here, we assume that the pre- and post-processing stages do not need GPU accelerations. The conventional method is to accelerate the whole DNN layers at the same time. This *full* GPU acceleration can dramatically reduce the perception delay to the optimal point. Besides, the frame rate is also significantly enhanced. However, we found that the resulting frame rate is suboptimal, since the overutilized shared GPU quickly becomes a bottleneck. With this observation, we propose a *partial* GPU acceleration that finds an optimal subset of DNN layers to be accelerated that maximizes the frame rate to the optimal point. This method implements both the frame rate-optimal and the delay-optimal system configurations.

Our perception system architecture is implemented based on the Darknet DNN framework [4], which is a state-of-the-art DNN framework widely used in autonomous driving vehicles due to its high performance and portability. Since the Darknet framework is based on the task-parallel architecture, we modify it based on our proposed architecture. Our experimental results with the DenseNet [5] classification network on an Nvidia Jetson AGX Orin platform show scalable frame rates with near-optimal perception delays by the data-parallel architecture. Our implementation also demonstrates the practicality of the partial GPU acceleration method by selectively achieving both the frame rate-optimal and delay-optimal results.

The contributions of this study can be summarized as:

- We present the data-parallel perception system architecture that maximally utilizes given CPU and GPU computing resources for the maximum system performance.
- We present the partial GPU acceleration method that selectively accelerates given DNN layers for the flexible optimization of frame rates and perception delays.

2 System Model and Problem Description

2.1 System Model

This study considers a real-time perception system with M identical CPU cores and a single GPU accelerator, denoted by $\{C_1, C_2, \cdots, C_M\}$ and G, respectively. We assume only a single GPU since most embedded hardware platforms use an integrated GPU without supporting external GPUs. A perception sensor (e.g., camera, LiDAR, and radar) is connected to the system through a dedicated bus (e.g., USB and Ethernet). This study assumes a single sensor configuration, and the sensor is assumed to have the freedom to adjust its sampling frequency according to the perception system's provided frame rate. Note that this study is interested in the maximum achievable system performance without assuming prefixed discrete sensor frequencies.

In the system, there is a DNN-based perception task that processes the sensor data to produce its perception results. Object detection and image classification can be typical examples. From now on, we narrow down our focus on camera-based perception tasks, which are generally composed of the following three stages:

- **Pre-processing:** This stage reads the sensor data, which are then prepared according to the DNN's input requirements. For example, raw camera images can be resized and quantized before going through a DNN.
- **Inference:** This stage does the forward propagation through the DNN layers, sequentially from the first layer to the last layer, to produce raw DNN outputs. This layer-by-layer execution is commonly found in most DNN models.
- **Post-processing:** This stage produces the final perception results from the raw DNN outputs. For example, non-maximum suppression (NMS) is typically applied to remove redundant detections in many object detection systems.

We assume that the pre-processing and post-processing stages can be only executed by a CPU core, while the inference stage can be selectively executed by a CPU core or by a GPU. In some cases, the pre- and post-processing stages can also be accelerated by GPUs when there are multiple GPUs in the system. However, since we assume only a single GPU, it is a natural choice to dedicate the GPU to accelerate the inference stage.

The worst-case execution time (WCET) of the pre- and post-processing stages are denoted by e_{pre} and e_{post}, respectively, assuming they run on CPU cores. When the inference stage is executed on a CPU core, its WCET is denoted by e_{infer}^{CPU}. In contrast, when the inference stage is accelerated by the GPU, its WCET is denoted by e_{infer}^{GPU}. Without considering any specific resource allocation, the WCET of the inference stage is generally denoted by e_{infer}. Then the total WCET is also given by

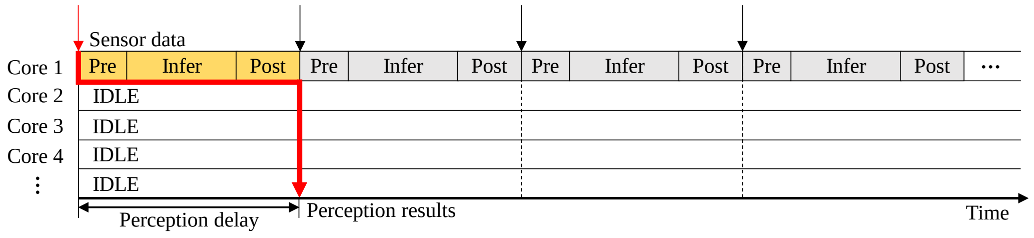
$$e_{total} = e_{pre} + e_{infer} + e_{post}. \tag{1}$$

During the inference stage, a DNN model is given as N layers, denoted by $\{l_1, l_2, \cdots, l_N\}$, where each layer represents a DNN operation such as fully-connected layers and convolutional layers. Due to the independence of each layer execution, each layer can select its computing resource. Each i-th layer's per-layer WCET is generally denoted by $e_{infer(i)}$. When specifying a specific computing resource, we use $e_{infer(i)}^{CPU}$ and $e_{infer(i)}^{GPU}$, respectively.

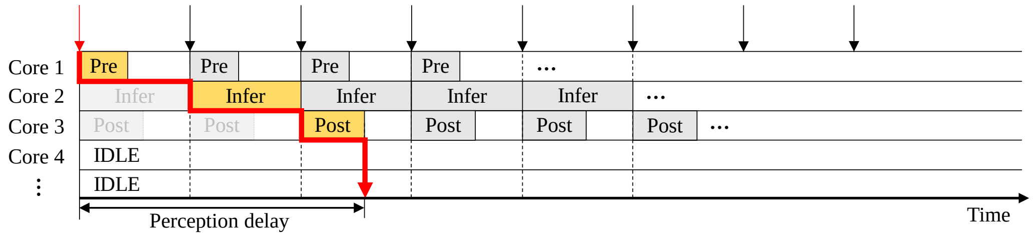
A perception system's performance is represented by a tuple

$$(f, d), \tag{2}$$

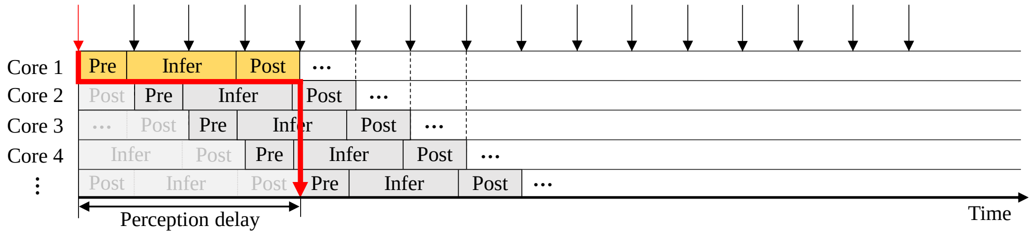
where f is the frame rate and d is the perception delay. The frame rate indicates the number of images the system can process in a unit time, usually expressed by a frames per second (FPS) value. The perception delay is defined as the elapsed time from the arrival of sensor data to the output of the final perception results. If no additional delay factor exists, the perception delay can be as short as e_{total}. The frame rate and the perception delay are crucial to autonomous systems' overall performance and safety. Also, note that there are various timing requirements. For example, some applications require a very high frame rate with a somewhat relaxed perception delay. In another case, the frame rate can be relaxed with a tighter delay requirement.



(a) *Sequential* architecture.



(b) *Task-parallel* architecture, exploiting task-level parallelism.



(c) *Data-parallel* architecture, exploiting data-level parallelism.

Figure 2: Comparison of perception system architectures.

2.2 Problem Description

Based on the above system model, our problem is to design a perception system architecture that provides the maximum system performance, considering both the frame rate and the perception delay. This study limits our problem to running the perception task as the only real-time task in the system because our objective is to make the perception task maximally utilize the computing resources for the maximal system performance.

3 Migrating Data-parallel Architecture

3.1 Sequential Architecture

Fig. 2a depicts the *sequential* architecture, where the pre-processing (*Pre*), inference (*Infer*), and post-processing (*Post*) stages are sequentially executed by a single thread with a limited utilization of just a single CPU core. The downward arrows indicate sensor data arrivals that go through the three stages to produce the perception results. The red thick arrow illustrates the data flow, indicating the perception delay. In the sequential architecture, the perception delay, denoted by d_{seq}, is given as

$$d_{seq} = e_{total} = e_{pre} + e_{infer} + e_{post} \tag{3}$$

since it does not experience any extra delays besides each stage's execution. Also, the frame rate, denoted by f_{seq}, is simply the inverse of the perception delay, as in

$$f_{seq} = \frac{1}{d_{seq}}. \tag{4}$$

Although the sequential architecture is the most naïve approach, it exhibits the *optimal* perception delay that cannot be further reduced in any way [1]. Meanwhile, the sequential architecture provides the *worst* frame rate since it utilizes only a single CPU core out of the given CPU cores without exploiting any parallelism in the perception task.

3.2 Task-parallel Architecture (a.k.a. Multithreaded Pipeline Architecture)

For more effective utilization of the given computing resources, many state-of-the-art perception systems employ the multithreaded pipeline architecture to exploit the task-level parallelism in perception tasks (hereafter referred to as *task-parallel* architecture). Fig. 2b depicts the task-parallel architecture, where the three perception stages are decomposed into three independent pipeline stages that run in parallel on three different CPU cores. Unfortunately, however, the synchronous pipeline cycle of the task-parallel architecture, denoted by s_{tpa}, is determined by the longest among the three pipeline stages (the inference stage in the figure), as in

$$s_{tpa} = \max(\{e_{pre}, e_{infer}, e_{post}\}). \tag{5}$$

Then, to estimate the perception delay, we have to take the pipeline stall delay into account, which can be observed between the pre-processing stage and the inference stage caused by the imbalanced pipeline stages. As a result, the perception delay, denoted by d_{tpa}, becomes the sum of two pipeline cycles added by one post-processing execution time, as in

$$d_{tpa} = 2s_{tpa} + e_{post}. \tag{6}$$

Although the perception delay is inferior to the sequential architecture, the frame rate is significantly enhanced by exploiting the task-level parallelism. In the task-parallel architecture, the frame rate, denoted as f_{tpa}, is given as the inverse of the pipeline cycle time, as in

$$f_{tpa} = \frac{1}{s_{tpa}} = \frac{1}{\max(\{e_{pre}, e_{infer}, e_{post}\})}. \tag{7}$$

Here, we realize that sometimes there is a tradeoff relation between the frame rate and the perception delay when optimizing the system performance. The task-parallel architecture in particular provides enhanced frame rates at the cost of deteriorated perception delays. Furthermore, unfortunately, the task-parallel architecture also fails to fully utilize the given CPU cores, utilizing only three of them. If we have more than three CPU cores, which is highly probable, the remaining CPU cores cannot be utilized by the task-parallel architecture, significantly underutilizing the given computing resources.

3.3 Data-parallel Architecture

To overcome the aforementioned limitations, we shift from the *task-parallel* architecture to the *data-parallel* architecture to exploit the data-level parallelism instead of the limited task-level parallelism. In our perception tasks, there are two kinds of noticeable data-level parallelism that can be exploited: (i) *spatial* parallelism and (ii) *temporal* parallelism. By exploiting the spatial parallelism, we can decompose an input data into multiple pieces that can be processed independently, which, however, can cause significant perception errors along the border lines. Alternatively, our choice is to exploit the temporal parallelism. Fig. 2c shows our data-parallel architecture particularly exploiting the temporal parallelism, where each sensor data arrival is dispatched to an available CPU core among the M CPU cores in a round-robin manner. Since they are independently processed on multiple CPU cores in parallel, the frame rate can scale linearly proportional to the number of CPU cores. Besides, there are no extra delays like the pipeline stall delays found in the task-parallel architecture. Ideally, the perception delay can be as small as the sequential architecture that guarantees the optimal perception delay (i.e., *near-optimal* perception delay). In practice, however, we have unavoidable memory contention delays by making all the CPU cores busy at the same time. Thus, the perception delay by our data-parallel architecture, denoted by d_{dpa}, is given by

$$d_{dpa} = e_{pre} + e_{infer} + e_{post} + \epsilon, \tag{8}$$

where ϵ denotes the collective memory contention delay, which depends on many factors such as workload characteristics and cache architectures. Here, our argument is that since we execute the

[1]Here, we do not consider the GPU acceleration, which will be discussed in Section 4.

exactly identical task code on every CPU core, the memory contention delay can be minimized in contrast to usual multi-tasking scenarios with more complex mixture of applications. In the meantime, the frame rate of our data-parallel architecture, denoted by f_{dpa}, is the inverse of the perception delay multiplied by M, given by

$$f_{dpa} = \frac{M}{d_{dpa}}, \tag{9}$$

since each CPU core independently produces the frame rate of $1/d_{dpa}$. Thus, by a large number of CPU cores, which is common in many modern SoCs, our data-parallel architecture can realize significantly increased frame rates by maximally utilizing all the given CPU cores.

With the above observations, we claim that with modern multicore SoCs, our proposed data-parallel architecture significantly outperforms the conventional task-parallel architecture in terms of the frame rate and the perception delay at the same time. To be more specific, the frame rate is almost linearly scalable to the number of CPU cores, while the perception delay approaches the optimal point of the sequential architecture with just a small increase caused by memory contentions.

LET-based jitter compensation. One extra issue is the jitter problem, which is vivid in the data-parallel architecture, due to the unpredictable memory contention among the busy CPU cores being forced to be fully utilized. The jitter makes it difficult to estimate the completion time of currently running perception tasks. Also, the jitter makes it difficult to clearly pinpoint the achievable frame rate if it has non-negligible fluctuations. With this challenge, we use the LET concept to make a constant perception delay by adding an artificial busy loop counting until the predefined WCET. Then we can consider perception tasks having constant execution times, significantly simplifying the system design.

4 Partial GPU Acceleration

4.1 GPU-based DNN Acceleration for Optimal Perception Delay

Thus far, we have utilized only the CPU cores without considering the GPU, as depicted in Fig. 3a. To further enhance the system performance, the inference stage needs to be accelerated by the GPU. Regarding the GPU acceleration, the usual approach is to accelerate the whole DNN layers at the same time (i.e., *full* GPU acceleration), as depicted in Fig. 3b, which will significantly reduce the perception delay from (8) to

$$d_{dpa}^{GPU} = e_{pre} + e_{infer}^{GPU} + e_{post}, \tag{10}$$

where e_{infer}^{GPU} must be much smaller than e_{infer}^{CPU}. Also, since e_{infer}^{GPU} cannot be further reduced by already accelerating all the layers, we can say that the perception delay is *optimal* by the full GPU acceleration.

However, even with the dramatic effect by the full GPU acceleration on the perception delay, the same dramatic effect is not guaranteed on the frame rate. While sounding counter-intuitive, recall that we have only a single GPU, which is a shared resource among all the perception instances running in parallel on many CPU cores. Thus, the GPU easily becomes a bottleneck, significantly limiting the scalability benefit of the frame rate by the data-parallel architecture. Imagine we begin with a very low frame rate, then the GPU will not be a bottleneck. However, by gradually increasing the frame rate, the perception instances are eventually forced to be *serialized* by the GPU at some point. Considering this GPU bottleneck effect, the frame rate after the full GPU acceleration, denoted by f_{dpa}^{GPU}, is updated from (9) to

$$f_{dpa}^{GPU} = \min\left(\left\{\frac{M}{d_{dpa}^{GPU}}, \frac{1}{e_{infer}^{GPU}}\right\}\right), \tag{11}$$

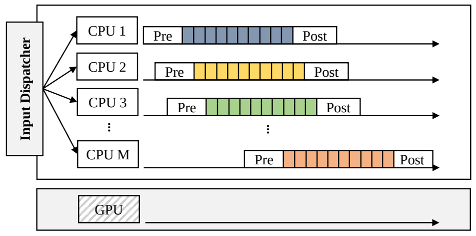
where $\frac{M}{d_{dpa}^{GPU}}$ is the maximum frame rate the M CPU cores can maximally produce before reaching the GPU bottleneck, while $\frac{1}{e_{infer}^{GPU}}$ is the bottlenecked frame rate, with all the inference stages of the perception instances strictly serialized by the GPU. Then the minimum of the two values will be the actual frame rate. If the system is bottlenecked by the GPU, the scalability benefit of multiplying M disappears, which means that the frame rate is suboptimal by the full GPU acceleration.

4.2 Partial DNN Acceleration for Optimal Frame Rate

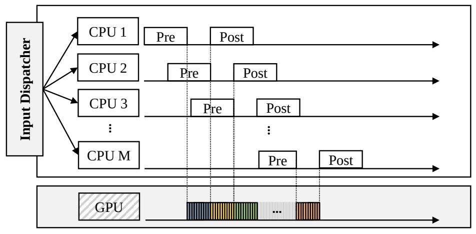
To maximize the frame rate by more efficiently utilizing the GPU resource, we present a *partial* GPU acceleration scheme, as depicted in Fig. 3c. Recall that the GPU acceleration only applies to the inference stage, where N DNN layers are executed in a layer-by-layer manner. Here, each layer has a freedom to choose the computing resource upon which it executes. In our system model, we have two choices of CPU and GPU. Although we can try every combination of mixed CPU layers and GPU layers, this study downsizes the problem space by defining the problem as splitting the N layers into the first k *GPU-accelerated* layers and the remaining $N-k$ *non-accelerated* layers. Thus, our problem is to find the optimal k $(0 \leq k \leq N)$ that maximizes the frame rate. In our problem, the execution time of the inference stage with k GPU-accelerated layers can be represented as

$$e_{infer}^{GPU(k)} = \sum_{i=1}^{k} e_{infer(i)}^{GPU} + \sum_{i=k+1}^{N} e_{infer(i)}^{CPU}. \quad (12)$$

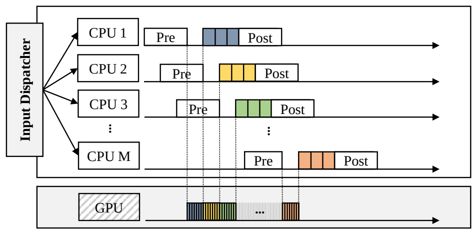
We propose an iterative optimization process that begins with $k = 0$ meaning no GPU acceleration. If we gradually increase k, the execution time decreases by accelerating more and more layers. Some layers will experience dramatic effects, while some others with relatively minor effects. The impact of the GPU acceleration heavily depends on the layer type and size. In our experience, for example, the larger layer (i.e., larger number of parameters) is better accelerated. Thus, we try every possible k, from 0 to N, while measuring the perception delay and the frame rate to find the *delay-optimal* k and the *frame rate-optimal* k. As noted in Section 4.1, the delay-optimal k is sure to be N, while the frame rate-optimal k cannot be simply said before trying the iterative optimization. However, the optimal k will make an equilibrium system state, where the GPU is barely but fully utilized and the CPU cores never waits for the GPU.



(a) No GPU acceleration.



(b) Full GPU acceleration.



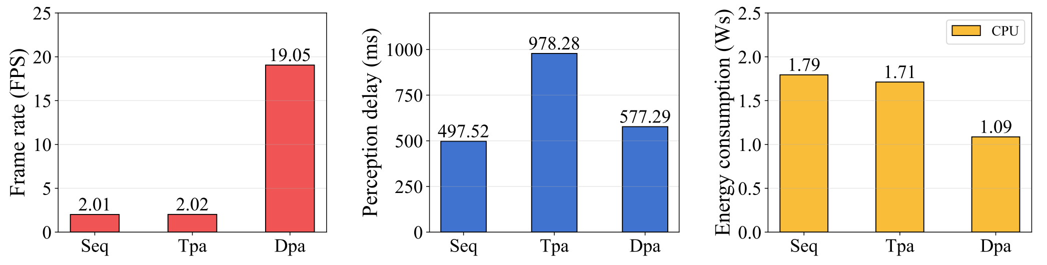
(c) Partial GPU acceleration.

Figure 3: Comparison of Data-Parallel Architectures for Partial DNN Acceleration.

5 Experiments

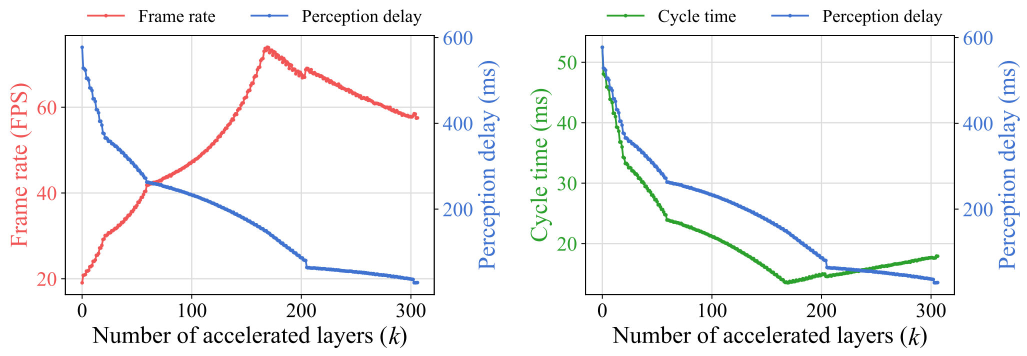
5.1 Implementation

We implemented our proposed architecture on an Nvidia embedded hardware platform. Our experimental platform is Nvidia Jetson AGX Orin with 32 GB RAM, a 12-core 2.2 GHz ARM CPU, and an integrated Ampere GPU with 2048 CUDA cores. Out of the 12 CPU cores, the first CPU core is dedicated to essential OS processes by the `isolcpus` Linux kernel parameter such that the remaining 11 cores (i.e., M=11) as well as the GPU are completely dedicated to the execution of the perception task. As our software platform, we use Nvidia Ubuntu 20.04.6 LTS with CUDA 11.4.239 and JetPack 5.0.2. Our implementation is based on Darknet [4, 6], which is one of the most well-known DNN frameworks originally developed for the YOLO object detection networks. Darknet employs the task-parallel architecture with three pipeline stages, exactly following our system model in Section 2.1. We modified Darknet to reflect our system design including the data-parallel architecture, the jitter compensation, and the partial GPU acceleration.



(a) Frame rate. (b) Perception delay. (c) Per-image energy consumption.

Figure 4: Performance comparison with 11 CPU cores.



(a) Frame-rate perspective. (b) Cycle-time perspective.

Figure 5: Iterative optimization for finding the frame-rate optimum by partial GPU acceleration.

5.2 Evaluation Results

Using our perception architecture implementation, we deploy DenseNet [5] with 306 layers as our workload. DenseNet is one of the most well-known convolutional neural networks (CNNs) for image classification. The system performance (i.e., frame rate and perception delay) and energy consumption is measured to compare the following architectures:

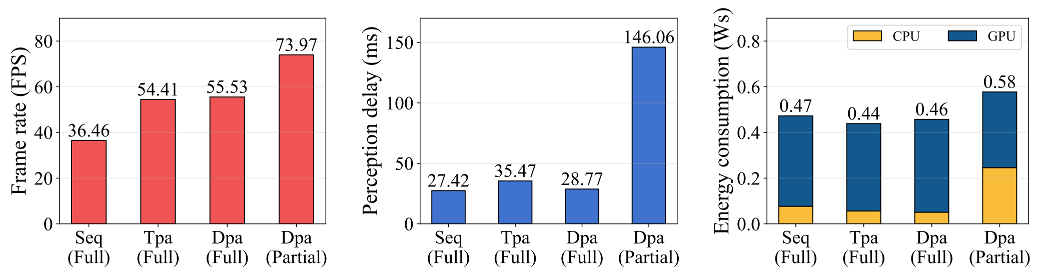
- **Seq**: **Seq**uential architecture (Section 3.1),
- **Tpa**: **T**ask-**p**arallel **a**rchitecture (Section 3.2), and
- **Dpa**: **D**ata-**p**arallel **a**rchitecture (Section 3.3).

Fig. 4a compares the frame rate, where **Dpa** significantly outperforms **Seq** (948%). The theoretical speed-up can be 1100% since **Dpa** utilizes 11 CPU cores simultaneously while **Seq** is with a single CPU core. Even with the non-negligible memory contention overhead, the result shows that **Dpa** successfully utilizes all the CPU cores for the maximum frame rate. Another interesting observation is that **Tpa** is not meaningfully enhanced from **Seq**. It is due to the extremely imbalanced pipeline stages of DenseNet.

Fig. 4b compares the perception delay. Note that **Dpa** is comparable to **Seq**'s optimal perception delay with a marginal overhead (+16%). In contrast, **Tpa** shows a significantly increased perception delay (+97%) caused by long pipeline stall delays. As a result, **Tpa** is the worst choice for DenseNet since it provides almost the same frame rate with a significantly increased perception delay from **Seq**.

Fig. 4c compares the per-image energy consumption. The figure shows that **Dpa** is the most energy-efficient method, which may sound counter-intuitive in the first place. However, it can be understood that the frame rate increase multiplier is significantly greater than the energy consumption increase multiplier, making **Dpa** the most energy-efficient choice.

Fig. 5 shows the iterative optimization process to find the frame rate-optimal configuration with our partial GPU acceleration. The two figures show the same results with different perspectives. Fig. 5a particularly shows the frame rate, while Fig. 5b shows its inverse (i.e., cycle time). The figures show



(a) Frame rate. (b) Perception delay. (c) Per-image energy consumption.

Figure 6: System performance comparison with full and partial GPU acceleration (11 CPU cores and a GPU).

that by accelerating more and more layers, the perception delay monotonically decreases. In contrast, the frame rate increases until reaching the optimal frame rate (73.97 FPS) with k=169. Then the frame rate begins to decrease meaning that it is bottlenecked by the GPU.

Fig. 6 compares the perception architectures with possibly different GPU accelerations. The full GPU acceleration is denoted by *(Full)*, while the partial GPU acceleration is denoted by *(Partial)* that specifically means the frame rate-optimal configuration. Fig. 6a shows that **Dpa (Partial)** achieves the optimal frame rate (73.97 FPS). However, Fig. 6b shows that **Dpa (Partial)** has a significant side-effect of an increased perception delay. In contrast, **Dpa (Full)** exhibits a near-optimal perception delay comparable to **Seq (Full)** while providing a comparable frame rate to **Tpa (Full)**. Thus, **Dpa (Full)** is the best choice among the four if the perception delay is more important than the frame rate. Also note that Fig. 6c shows that **Dpa (Full)** is also very energy efficient. In contrast, **Dpa (Partial)** is the worst in terms of energy consumption since it makes all the computing resources (i.e, CPU and GPU) busy.

6 Related Work

The system optimization in DNN-based applications plays important roles in autonomous driving [7, 8, 9, 10, 11, 12]. Among them, this study focuses on the perception system. Many such perception systems are based on the task-parallel architecture. In [13], a parallel-pipeline architecture is proposed to improve the object detection throughput by slightly sacrificing the delay. In contrast, R-TOD [1] minimizes the object detection delay by optimizing every end-to-end delay component at the cost of the slightly reduced frame rate. Both studies are based on the task-parallel architecture without exploiting the data-level parallelism as our proposed architecture. DART [14] exploits the data-level parallelism for the deterministic execution of real-time tasks. However, DART exploits only the spatial parallelism without considering the temporal parallelism as our proposed architecture.

7 Conclusion

This study presents a data-parallel real-time perception system with partial GPU acceleration. Our perception architecture maximally utilizes the given computing resources including CPUs and GPUs for the maximum system performance. For that, we migrate from the conventional task-parallel architecture to the data-parallel architecture that specifically exploits the temporal data parallelism while processing the perception tasks. Also, our partial GPU acceleration framework accelerates only a fractional subset of given DNN layers that maximizes the frame rate. While the conventional full GPU acceleration always provides the optimal perception delay, it does not guarantee the optimal frame rate. An iterative optimization is used to find the frame rate-optimal and delay-optimal system configurations based on the partial GPU acceleration framework. Our perception system architecture is implemented with an Nvidia embedded platform and evaluated with the Darknet DNN framework and the DenseNet image classification network. In the future, we plan to extend our framework to multi-DNN systems by considering preemptive scheduling among multiple DNNs.

Acknowledgment

This work was supported by the BK21 Four Program (5199990814084) of the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Korea. J.-C. Kim is the corresponding author of this paper.

References

[1] W. Jang, H. Jeong, K. Kang, N. Dutt, and J.-C. Kim, "R-tod: Real-time object detector with minimized end-to-end delay for autonomous driving," in *Proc. 41st IEEE Real-Time Systems Symposium (RTSS)*, pp. 191–204, 2020.

[2] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[3] C. M. Kirsch and A. Sokolova, "The logical execution time paradigm," *Advances in Real-Time Systems*, pp. 103–120, 2012.

[4] J. Redmon, "Darknet: Open source neural networks in C." `http://pjreddie.com/darknet/`, 2013–2016. Accessed: 2023-11-16.

[5] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. 30th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4700–4708, 2017.

[6] A. Bochkovskiy, "Darknet: Open source neural networks in C." `https://github.com/AlexeyAB/darknet`. Accessed: 2023-11-16.

[7] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proc. 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 751–766, 2018.

[8] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks," in *Proc. 42nd IEEE Real-Time Systems Symposium (RTSS)*, pp. 329–341, 2021.

[9] W. Kang, S. Chung, J. Y. Kim, Y. Lee, K. Lee, J. Lee, K. G. Shin, and H. S. Chwa, "Dnn-sam: Split-and-merge dnn execution for real-time object detection," in *Proc. 28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 160–172, 2022.

[10] S. Liu, S. Yao, X. Fu, R. Tabish, S. Yu, A. Bansal, H. Yun, L. Sha, and T. Abdelzaher, "On removing algorithmic priority inversion from mission-critical machine inference pipelines," in *Proc. 41st IEEE Real-Time Systems Symposium (RTSS)*, pp. 319–332, 2020.

[11] S. Liu, X. Fu, M. Wigness, P. David, S. Yao, L. Sha, and T. Abdelzaher, "Self-cueing real-time attention scheduling in criticality-aware visual machine perception," in *Proc. 28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 173–186, 2022.

[12] S. Heo, S. Jeong, and H. Kim, "Rtscale: Sensitivity-aware adaptive image scaling for real-time object detection," in *Proc. 34th Euromicro Conference on Real-Time Systems (ECRTS)*, 2022.

[13] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *Proc. 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 305–317, 2019.

[14] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *Proc. 40th IEEE Real-Time Systems Symposium (RTSS)*, pp. 392–405, 2019.