2021 IEEE Real-Time Systems Symposium (RTSS) | 978-1-6654-2802-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/RTSS52674.2021.00038

LaLaRAND: Flexible Layer-by-Layer CPU/GPU Scheduling for Real-Time DNN Tasks

Woosung Kang[†], Kilho Lee[‡], Jinkyu Lee[§*], Insik Shin[¶] and Hoon Sung Chwa[†*]
Dept. of Information and Communication Engineering, DGIST, Republic of Korea[†]
School of AI Convergence, Soongsil University, Republic of Korea[‡]
Dept. of Computer Science and Engineering, Sungkyunkwan University (SKKU), Republic of Korea[§]
School of Computing, KAIST, Republic of Korea[¶]

***Abstract*—Deep neural networks (DNNs) have shown remarkable success in various machine-learning (ML) tasks useful for many safety-critical, real-time embedded systems. The foremost design goal for enabling DNN execution on real-time embedded systems is to provide worst-case timing guarantees with limited computing resources. Yet, the state-of-the-art ML frameworks hardly leverage heterogeneous computing resources (i.e., CPU, GPU) to improve the schedulability of real-time DNN tasks due to several factors, which include a coarse-grained resource allocation model (one-resource-per-task), the asymmetric nature of DNN execution on CPU and GPU, and lack of schedulability-aware CPU/GPU allocation scheme. This paper presents, to the best of our knowledge, the first study of addressing the above three major barriers and examining their cooperative effect on schedulability improvement. In this paper, we propose LaLaRAND, a real-time layer-level DNN scheduling framework, that enables flexible CPU/GPU scheduling of individual DNN layers by tightly coupling CPU-friendly quantization with fine-grained CPU/GPU allocation schemes (one-resource-per-layer) while mitigating accuracy loss without compromising timing guarantees. We have implemented and evaluated LaLaRAND on top of the state-of-the-art ML framework to demonstrate its effectiveness in making more DNN task sets schedulable by 56% and 80% over an existing approach and a baseline (vanilla PyTorch), respectively, with only up to -0.4% of performance (inference accuracy) difference.**

I. Introduction

As deep neural networks (DNNs) have emerged as the fundamental element and core enabler in machine learning applications, well-trained DNN models are increasingly deployed for inference, perception, and control tasks in many safety-critical, real-time embedded systems (e.g., autonomous driving [1], robotics [2], and healthcare systems [3]). Recently, along with the rapid advent of high-performance system-on-chips (SoCs) equipped with heterogeneous computing resources, such as GPUs, executing multiple DNNs on a shared computing platform gains popularity and is quickly becoming the mainstream for real-time embedded systems [1], [4]–[14]. The major design goal for such real-time embedded systems is to provide worst-case timing guarantees for real-time DNN tasks with efficient use of the underlying computing resources.

Although the state-of-the-art machine learning (ML) frameworks, such as PyTorch [15], TensorFlow [16], and Caffe [17], provide well-defined programming models and highly optimized internal implementation, it is still challenging to efficiently utilize heterogeneous computing resources for typical multi-layer DNNs so as to fulfill their timing requirements. They employ a monolithic resource allocation model that statically allocates all layers of a DNN task into a single type of pre-defined resource (e.g., CPUs or GPUs) and use canonical scheduling principles (i.e., FIFO) that sequentially schedule DNN tasks without prioritizing them according to their timing requirements. Moreover, modern heterogeneous SoCs have shown the performance of different processor types, i.e., the CPU and the GPU (the two most prevalent types of processors), to be unbalanced for DNN execution, posing another challenge for efficient use of heterogeneous resources in meeting timing requirements. Our experimentation with popular DNN models on a representative CPUs–GPU SoC has demonstrated an up to $296.6\times$ slowdown of CPU execution over GPU execution.

A rich number of prior studies have been focused on splitting DNN computations across heterogeneous processors in order to improve the performance of DNN inference [18]–[22]. They partition each DNN into smaller units and distribute them across heterogeneous computing resources, such as GPUs, with some DNN optimization techniques (e.g., compression, quantization, and pruning) [18]–[20]. Some studies further reduce the latency by executing each layer on different computing resources [19] or by offloading DNN computations to the cloud [20], [21]. Although all of these studies have made valuable contributions on lowering inference latency by utilizing multiple resources, they did not consider how to leverage heterogeneous resources to improve the schedulability of real-time DNN tasks. A recent study [6], most relevant to our work, focused on ensuring the timing constraints of DNN tasks. The work in [6] employs a pipeline-based resource allocation and scheduling architecture, called DART, where CPUs and a GPU are arranged into nodes, and subsets of consecutive layers of each DNN task are allocated to different nodes and executed in a pipeline manner. Although DART provides deterministic response time to DNN tasks, the performance imbalance issue pertaining to heterogeneous resources and subsequent related issues are not fully considered, which significantly affects schedulability performance.

In this paper, we aim to improve the schedulability of real-time DNN tasks while leveraging heterogeneous resources. To achieve this goal, we propose a new real-time layer-level DNN scheduling framework, LaLaRAND (**La**yer-by-**La**yer **R**esource **A**llocation for **N**-**D**NNs). One of the key features of LaLaRAND is the use of CPU-friendly quantization, one of the powerful techniques to optimize the CPU performance, to mitigate the performance imbalance between CPU and GPU resources. Another key feature is a transparent, fine-grained (layer-level) CPU/GPU scheduling mechanism that offers maximum flexibility in coordinating the allocation of

*Corresponding authors: Hoon Sung Chwa (chwahs@dgist.ac.kr); Jinkyu Lee (jinkyu.lee@skku.edu).

978-1-6654-2802-6/21/$31.00 ©2021 IEEE
DOI 10.1109/RTSS52674.2021.00038

CPU and GPU to DNN tasks, without requiring any code modification of DNN applications. Building upon the above two features, LaLaRAND develops a new scheduling policy that determines the allocation of individual DNN layers to CPU and GPU in a way to improve schedulability. To this end, we derive a response time analysis to verify whether DNN tasks satisfy their timing constraints with careful consideration of inherent overheads pertaining to our proposed layer-level CPU/GPU allocation scheme, and we develop a layer-by-layer CPU/GPU allocation algorithm that takes into account the effect of each layer's resource allocation on the schedulability with respect to the proposed schedulability analysis. Tightly integrating all the above features is not only novel but also crucial to achieve the goal. This is because a lack of either quantization or layer-level allocation typically inflates the execution time of a DNN task by an order of magnitude, making the task hardly schedulable.

It is worth noting that CPU-friendly quantization improves inference latency at the expense of accuracy. From our experimental results, we observed quantized layers make different contributions to the loss of accuracy of the DNN model; a few quantized layers cause a large accuracy drop by up to 29.6 percentage points (%p) relative to full precision (without quantization), while most of the quantized layers cause a marginal accuracy drop by up to 0.3%p (to be detailed in Sec. II). To mitigate this problem, LaLaRAND enables runtime layer migration that reclaims and reallocates unused CPU/GPU resources in a way that reduces the accuracy loss without violating any timing constraint.

We implement a prototype of LaLaRAND on top of PyTorch [15], one of the most popular ML frameworks. We evaluate LaLaRAND with four standard DNN models, i.e., GoogLeNet [23], SqueezeNet [24], MnasNet [25], and MobileNetV2 [26], on a representative CPUs–GPU board, i.e., Nvidia Jetson Xavier [27]. The evaluation results show that LaLaRAND outperforms the existing resource allocation approaches significantly in terms of the schedulability without significant accuracy losses for real-time DNN tasks. LaLaRAND is shown to make 56% and 80% more real-time DNN task sets schedulable over an existing approach [6] and a default resource allocation mechanism on vanilla PyTorch, respectively, while only an up to 0.4%p inference accuracy drop or even an up to 0.7%p accuracy improvement is observed.

Contribution. To the best of our knowledge, this paper presents a first approach that combines CPU-friendly quantization with layer-level CPU/GPU allocation schemes to create a collaborative contribution to improving the schedulability of real-time DNN tasks. The contributions of this paper can be summarized as follows:

- We demonstrate, via in-depth case studies, the significant performance imbalance between CPU and GPU, an opportunity of employing CPU-friendly quantization, and the importance of layer-level resource allocation while accounting for its impact on schedulability and inference accuracy (Sec. II).
- We present a new system abstraction that supports the scheduling of individual DNN layers on CPU and GPU in a transparent way (Sec. III).
- We create an offline layer-by-layer CPU/GPU allocation algorithm that not only provides timing guarantees but also increases schedulability performance (Sec. IV).
- We develop an online CPU/GPU allocation scheme that supports the reallocation of DNN layers across CPU and GPU to mitigate the accuracy loss due to CPU-friendly quantization without hurting any timing guarantees (Sec. V).
- From our experimental results, we found that LaLaRAND significantly outperforms a state-of-the-art ML framework with standard DNN models by up to 80% more DNN taskset schedulability (Sec. VI).

II. Motivation

In this section, we first examine, via a set of measurement-based case studies, the performance imbalance between CPU and GPU as a main barrier for efficient use of both CPU and GPU so as to accommodate as many real-time DNN inference tasks as possible without violating any timing constraints. We then explore an opportunity of employing CPU-friendly quantization for DNNs, one of the popular techniques to optimize the CPU execution performance, to hurdle the barrier and introduce the important issues faced therein.

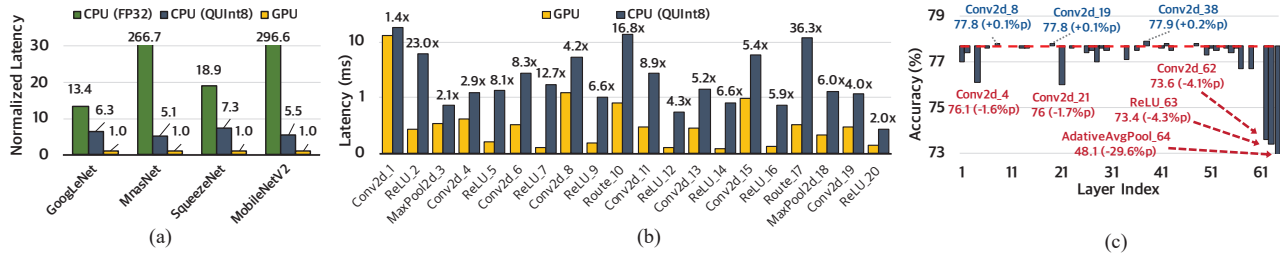
A. CPU vs GPU Performance Imbalance in DNN Execution

To demonstrate the performance imbalance issue, we conducted experiments to measure the execution times of representative DNNs (GoogLeNet, SqueezeNet, MnasNet, and MobileNetV2) running either on the CPU or on the GPU of Nvidia Jetson Xavier. We note that Nvidia Jetson Xavier is equipped with 8 CPUs and 1 GPU, and in the case of CPU execution, each DNN is executed in parallel on 8 CPUs to maximize performance. As shown in Fig. 1(a), our experimental results show a significant performance gap between the CPU and GPU executions. Executing DNNs on CPUs yields an average slowdown of $148\times$ over executing on a GPU. For MnasNet and MobileNetV2, their execution times (green bars) on the CPU are even increased by $266.7\times$ and $296.6\times$ than those (yellow bars) on the GPU, respectively. To verify whether the results also hold for other computing platforms, we perform a similar experiment using two additional representative embedded boards, Nvidia Jetson TX2 [28] and Nano [29]. The results show a similar conclusion, inferring that such a significant performance imbalance between CPU and GPU becomes a main barrier for efficient use of heterogeneous resources to accommodate as many real-time DNN tasks as possible without violating any timing constraints.

B. CPU-friendly Quantization

To increase the resource efficiency, prior studies [30]–[32] proposed to employ *quantization* which shrinks the default 32-bit single floating-point values (`FP32`) of DNNs to occupy fewer bits through some transformations. Reducing the bit width can greatly improve the speed of executing a DNN.

To resolve the CPU/GPU performance imbalance problem, we consider an 8-bit linear quantization scheme [33] involving 8-bit integer values for CPUs because each CPU core is equipped with vector Arithmetic Logic Units (ALUs) capable of processing multiple 8-bit integers in parallel. The 8-bit linear quantization scheme maps a set of `FP32` values to 8-bit unsigned integers (`QUInt8`) where 0 and 255 map to the minimum and the maximum `FP32` values, respectively. This scheme is not only able to reduce the space by 75%, but also allows to convert floating-point numbers to integer numbers leading to a great reduction in execution time.



(a) (b) (c)

Fig. 1. (a) DNN execution times on CPU (`FP32`), CPU (`QUInt8`), and GPU in Nvidia Jetson Xavier, (b) per-layer execution times of SqueezeNet on CPU and GPU, and (c) layer-level sensitivity analysis of SqueezeNet.

Fig. 1(a) shows the reduction in the CPU execution time of four DNN models when the quantization scheme is applied. For example, the CPU execution times of MnasNet and MobileNetV2 are decreased by 52.2× and 53.9× with the 8-bit linear quantization scheme, respectively, which are comparable to their GPU execution times as 5.1× and 5.5× respectively.

From now on, throughout the paper, we assume that the CPU and GPU cores perform computations using `QUInt8` integers and `FP32` values, respectively. The rationale behind this is an expectation that both CPU and GPU cores can greatly benefit from the data types using their native ALUs; GPU cores are optimized for graphics applications which heavily utilize floating points (e.g., `FP32`) and CPU cores are equipped with vector ALUs capable of processing multiple 8-bit integers (e.g., `QUInt8`) in parallel.

Although the results indicate that CPU-friendly quantization is beneficial to resolve the significant CPU/GPU performance imbalance problem, we found two key issues to efficiently utilize both CPU and GPU cores for multi-DNN real-time inference: (1) layer-level different execution characteristics and (2) quantization impacts on inference accuracy.

Layer-level execution characteristics. We first investigate the execution characteristics of each layer in SqueezeNet. Fig. 1(b) shows the execution time of each layer on CPU and GPU cores in log scale, arranged from left to right in their sequential execution order.[1] We observed that the execution times of CPU and GPU and their ratios of CPU to GPU vary greatly by layers depending on layers' type and location in the DNN. Among all layers, their CPU/GPU execution times and execution time ratios of CPU to GPU differ by up to 41.5× and up to 128.9×, respectively. In particular, the convolution layers (Conv) show relatively longer execution time on GPU by up to 104.7× than the activation layers (ReLU). The execution time ratio of CPU to GPU for the first layer is 1.4, while that for the second layer is 23.0. Among the convolution layers, their execution time ratios of CPU to GPU differ by up to 6.3×. Note that typical DNNs are composed of alternating layers of various types each of which shows different execution characteristics. Such a large variation of layer-level execution times calls for CPU/GPU allocation at the layer level granularity, which will be discussed in Sec. IV.

Quantization impacts on inference accuracy. Although the 8-bit quantization scheme improves the CPU execution performance comparable to GPU's, as a side effect, it may incur an inference accuracy loss for DNNs. We conduct another case study to measure how reducing the precision of each layer by quantization affects the DNN's overall top-1 inference accuracy, referred to as *sensitivity analysis*. In sensitivity analysis, a single layer is quantized at a time while the others are unquantized (i.e., leaving their inputs and computation in floating-point), and inference accuracy is evaluated over 1,000 images on the ImageNet dataset [34]. Fig. 1(c) shows the accuracy difference between a single-layer quantization and full precision (without quantization). The results show that just a few quantized layers, e.g., 62nd, 63rd, and 64th layers, cause a large accuracy drop by 4.1%p, 4.3%p, and 29.6%p, respectively, relative to full precision (which has 77.7% accuracy), while most of layers cause a marginal accuracy drop by up to 0.3%p when quantization is applied. Note that quantizing some layers, e.g., 8th, 19th, and 38th layers, even slightly improves the accuracy. We infer via this case study that the accuracy loss can be minimized or the accuracy can be even improved by selectively quantizing layers according to the sensitivity analysis, which will be discussed in Sec. V. To verify whether the results also hold for other DNNs, we perform a similar experiment using three other DNNs, i.e., GoogLeNet, MnasNet, and MobileNetV2. We confirmed that the results draw a similar conclusion.

Note that, although retraining DNN models is not required for applying CPU-friendly quantization, there exist quantization overheads which will be discussed in Sec. VI.

III. LaLaRAND

A. System Goal and Overview

Recently, DNN applications are typically implemented on top of the ML frameworks [15]–[17], [35], thanks to their well-defined programming model and highly-optimized internal implementation. However, it is challenging for such frameworks to provide flexible CPU/GPU allocation schemes to guarantee and improve the schedulability of real-time DNN tasks due to several factors. In this paper, We aim to develop a layer-level CPU/GPU scheduling framework with CPU-friendly quantization, achieving the following goals for real-time DNN inference tasks:

- G1. It not only provides timing guarantees for real-time DNN tasks but also improves schedulability performance, and
- G2. It minimizes the overall accuracy loss due to CPU-friendly quantization, i.e., preserving the overall accuracy close to that of the full precision (without quantization).

Regarding G1, we propose a new system abstraction, named LaLaRAND, that enables transparent and flexible CPU/GPU scheduling of individual DNN layers. In particular, LaLaRAND addresses several major barriers in leveraging CPU and GPU to improve the schedulability of DNN tasks, which

[1] We only depict the results of the first 20 layers out of 64 layers, since a similar trend can be seen for the rest of layers.

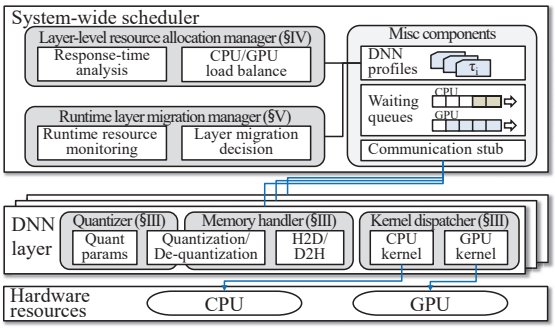


Fig. 2. System-level overview of LaLaRAND

include performance imbalance between CPU and GPU execution, monolithic resource allocation, separate CPU and GPU memory spaces, and lack of system-wide scheduling decision.

We then develop schedulability analysis for the proposed layer-level CPU/GPU scheduling abstraction, addressing the following challenges. Depending on which resource (CPU and GPU) individual layers are allocated to, i) the execution time of a DNN task varies and ii) new overheads (i.e., the cost of maintaining data consistency between two consecutive layers) can be introduced, both of which affect the response time of the DNN task. Moreover, the amount of interference or blocking delay imposed by other tasks may also vary depending on layer-level allocation. We introduce a new efficient layer-by-layer CPU/GPU allocation algorithm that finds a mapping between individual layers and CPU/GPU cores such that all DNN tasks are schedulable based on the proposed response time analysis, increasing schedulability performance.

Regarding G2, CPU-friendly quantization, one of the key features of LaLaRAND, typically comes with the trade-off between the speedup of CPU execution and inference accuracy drop. Furthermore, CPU-friendly quantization brings different effects on the accuracy of a DNN task depending on which layers it is applied to. Thus, this makes it quite complicated to apply CPU-friendly quantization for finding an optimal layer-level CPU/GPU resource allocation that achieves both G1 and G2. Thus, we first address G1 and then G2. To this end, LaLaRAND supports runtime layer migration to mitigate the accuracy loss due to CPU-friendly quantization. It reclaims unused CPU and GPU resources at runtime and utilizes them to selectively reallocate the layers that contribute to most of the accuracy loss when quantized from CPU to GPU cores without violating any timing constraint.

B. Design of LaLaRAND

LaLaRAND supports flexible CPU/GPU scheduling for DNN layers while providing real-time guarantees, without having to modify the code of DNN applications. The core design features of LaLaRAND include CPU-friendly quantizer, Dynamic kernel dispatcher, Transparent memory handler, and System-wide scheduler as depicted in Fig. 2.

CPU-friendly quantizer. Typically, quantization is applied to the entire DNN model at design time. In contrast to such task-level quantization, CPU-friendly quantizer supports dynamic layer-level quantization. When initializing a DNN task, the quantizer employs the qnnpack library [36] in PyTorch, performs input calibration at each layer to determine necessary

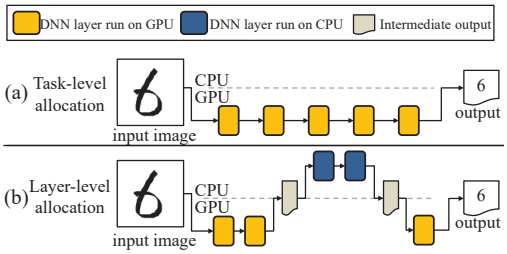


Fig. 3. Task-level allocation and layer-level allocation examples

parameters, i.e., scales and zero points, and stores them for layer-level quantization/dequantization. With the preloaded parameters, Transparent memory handler can dynamically quantize/dequantize input data for each layer at runtime.

Dynamic kernel dispatcher. As shown in Fig. 3(a), the current ML frameworks adopt a task-level resource allocation model that statically allocates all the layers of a DNN task into a single pre-defined resource (i.e., CPU or GPU). Such a task-level resource allocation model inherently limits an opportunity of optimizing the overall performance despite the availability of another resource. Dynamic kernel dispatcher relaxes this task-level resource assumption, enabling each layer to be executed on different resources as shown in Fig. 3(b). To do this, LaLaRAND generates multiple *kernels* corresponding to each available resource and selectively runs a single kernel per layer. Note that a kernel represents a unit of execution that contains codes and data for each layer. The kernel dispatcher is implemented as a wrapper that hooks the initialization and invocation phases of each layer. When initializing a DNN task, the kernel dispatcher generates both CPU and GPU kernels for each layer and preloads the metadata (e.g., weight tensors) for the layer into both CPU and GPU memories. Note that the CPU kernel is quantized at this step to mitigate the CPU-GPU performance imbalance. After that, when each layer is invoked to execute, the kernel dispatcher can dynamically allocate either CPU or GPU to the layer by simply selecting a proper kernel to run. This multi-kernel architecture is effective to achieve our goals, since it can provide runtime layer-level dynamic resource allocation with minimized overhead. Without this, the framework should generate a new kernel every time upon any change to resource allocation, incurring large computation and memory copy overheads. Concerning the spatial overhead imposed by the multi-kernel architecture, additional kernels only occupy a relatively small memory space (i.e., up to 6.9 MB in our experiment) since they typically utilize quantized data (to be detailed in Sec. VI).

Transparent memory handler. Under the task-level resource allocation model, many ML frameworks implement the feature of passing data between layers by simply sharing a specific memory buffer on the designated resource's memory. A preceding layer simply writes its output to the buffer and its succeeding one reads as input the output from the buffer. However, this scheme is no longer valid when two consecutive layers running on different resources, since CPU and GPU typically have separate memory spaces. Thus, Transparent memory handler maintains data consistency, when two consecutive layers are allocated to different resources, through dynamic (de)-quantization and memory buffer management. Specifically, the data handler employs explicit memory transfer

between a *host* (i.e., CPU) and a *device* (i.e., GPU); it copies the memory buffer from host to device (i.e., *H2D*) or from device to host (i.e., *D2H*) according to the resource allocation.

In addition, it dynamically quantizes/dequantizes intermediate data. If per-layer quantization is applied, each layer requires a different data type as input; for instance, a quantized layer uses an integer type (e.g., `QUInt8`), while a dequantized one uses a floating-point type (e.g., `FP32`). Note that such data handling happens right before executing a kernel; thereby, each layer has only one or no data handling as long as the layer executes on a single resource, even runtime migration happens. As the data handler performs runtime data processing, it may incur additional computation and memory copy overhead. However, this overhead is not large as the layer computation itself (to be detailed in Sec. VI). In addition, since the overhead is predictable through offline profiling, the scheduler can take this overhead into account as presented in Sec. IV.

System-wide scheduler. To optimize the overall scheduling performance, it is essential to make system-wide scheduling decision. To this end, LaLaRAND introduces System-wide scheduler which keeps track of system-wide information gathered from all running DNN tasks, operating as an independent process while communicating individual DNN layers running in the system. Note that it communicates with each layer through the IPC-based *communication stub* interface implemented within each layer and the scheduler.

To develop an offline schedule of DNN tasks, the scheduler maintains each DNN's profile including its real-time requirement, CPU/GPU execution time, CPU/GPU memory copy cost, and quantization/dequantization cost. According to those profiles, System-wide scheduler decides the CPU/GPU allocation of each individual layer in a way that meets timing constraints (see Sec. IV for details). System-wide scheduler then controls the execution timing of each layer through waiting queues corresponding to each resource. The waiting queues are implemented as priority queues to support fixed-priority scheduling. Whenever the first layer of each DNN task is ready to execute for an inference request, it sends a *ready* message to System-wide scheduler. The scheduler then enqueues the message into the proper resource's queue to which the layer is allocated and selects the highest-priority layer in the waiting queue to run on a corresponding resource. Whenever a layer finishes its execution, its subsequent layer (if any) is *immediately* enqueued into the waiting queue to which the layer is allocated. If the subsequent layer is allocated to the same resource as its previous layer, System-wide scheduler does not execute any layer of a lower-priority DNN task between the completion of the previous layer and the release of the subsequent layer. This way, System-wide scheduler allows the preemption of a currently-executing DNN task by a higher-priority DNN task at the layer level granularity, while preventing unnecessary blocking from lower-priority DNN tasks.

For the runtime migration of layers, the scheduler continuously monitors the presence of unused resources (i.e., *slack*), by capturing earlier completion of each layer's execution than its worst-case execution time. Any layer can be reallocated from an already assigned resource to the other one if the other resource has enough slack and the layer expects to result in higher accuracy after migration. The details of scheduling and migration policy will be presented in Sec. IV.

IV. Layer-by-Layer Resource Allocation

In this section, we present schedulability analysis for the proposed LaLaRAND scheduling framework and propose a new layer-level CPU/GPU allocation algorithm based on the analysis. To this end, we first explain our system model with notations. We then develop a response time analysis that determines the schedulability of a set of real-time DNN tasks under any given layer-level resource allocation. Finally, we develop an algorithm to find a layer-by-layer allocation for a target task set offline, which yields the schedulability of the task set by the response time analysis.

A. System Model and Notations

We consider a CPUs/GPU platform π configured as two clusters π^{C} and π^{G}, where π^{C} comprises m^{C} (≥ 1) identical CPU cores and π^{G} comprises one CPU core and one GPU device. Note that cluster π^{G} has a dedicated CPU core to prevent unnecessary delays on π^{C} due to the GPU-related CPU operations, such as a kernel launch and data transmission.

Real-time DNN inference tasks are presented by the sporadic task model which is widely used in various real-time systems. We assume each task uses one DNN model and makes one inference request per job. Each DNN task $\tau_i \in \tau$ can be specified as $\tau_i = (T_i, V_i, N_i)$, where T_i is the minimum separation between successive job releases, V_i is a sequence of DNN layers, and N_i is the number of layers. Each DNN layer $\tau_{i,j} \in V_i$ is characterized by $(C^{\mathrm{C}}_{i,j}, O^{\mathrm{C}}_{i,j}, C^{\mathrm{G}}_{i,j}, O^{\mathrm{G}}_{i,j})$, where

- $C^{\mathrm{C}}_{i,j}$: the worst-case execution time (WCET) when running on the CPU cluster with its quantized model;
- $O^{\mathrm{C}}_{i,j}$: the sum of the maximum time to quantize the input of $\tau_{i,j}$ from `FP32` to `QUInt8`, denoted as $O^{\mathrm{C1}}_{i,j}$, and the maximum time to dequantize the output of $\tau_{i,j}$ from `QUInt8` to `FP32`, denoted as $O^{\mathrm{C2}}_{i,j}$;
- $C^{\mathrm{G}}_{i,j}$: the worst-case GPU kernel execution time; and
- $O^{\mathrm{G}}_{i,j}$: the sum of the maximum time to copy the input of $\tau_{i,j}$ from the host (CPU) memory to the device (GPU) memory, denoted as $O^{\mathrm{G1}}_{i,j}$, and the maximum time to copy the output of $\tau_{i,j}$ back to CPU memory, denoted as $O^{\mathrm{G2}}_{i,j}$.

Each cluster executes one layer at a time. If CPU cluster π^{C} consists of more than one CPU core, each layer $\tau_{i,j}$ is executed on m^{C} cores in parallel and $C^{\mathrm{C}}_{i,j}$ is determined by the slowest thread. Since GPU cluster π^{G} consists of a dedicated CPU core and one GPU device, $C^{\mathrm{G}}_{i,j}$ includes the WCET of CPU operations for kernel launching, in addition to the worst-case GPU kernel execution time. Note that the values of each layer's parameters can be estimated from DNN profiles proactively stored in System-wide scheduler.

Due to the precedence dependency, $\tau_{i,j}$ cannot start execution unless $\tau_{i,j-1}$ has finished execution (except the first layer of τ_i). Such a DNN task τ_i is assumed to generate a potentially infinite sequence of jobs, each of which is separated from its predecessor by at least T_i time-units and is required to complete the execution of N_i layers within a relative deadline of T_i from its release.

We consider task-level fixed-priority scheduling where each task τ_i has a single static priority shared over all of its layers. Execution of each layer is non-preemptive, so the preemption of a higher-priority task is only allowed after the completion of

a currently-executing layer of a lower-priority task. We assume a quantum-based time; let one time unit a quantum length without loss of generality.

B. Schedulability Analysis and Allocation Algorithm

We state the *layer-by-layer CPU/GPU allocation* problem.

Definition 1 (Layer-by-layer CPU/GPU Allocation): Given a task set τ and a CPUs/GPU platform π, find a mapping from all the layers of $\tau_i \in \tau$ to π^{C} and π^{G} such that all tasks whose subsets of layers are mapped onto either π^{C} or π^{G} meet their deadlines under fixed-priority scheduling.

We first derive a response time analysis to verify whether all real-time DNN tasks satisfy their timing constraints under a given layer-level CPU/GPU allocation. We then propose the layer-by-layer CPU/GPU allocation algorithm to find a mapping from layers of each DNN task to CPU and GPU clusters such that all tasks are schedulable by the response time analysis.

Response time analysis. For a given layer-by-layer CPU/GPU allocation $\Lambda = \{\{\Lambda_i^{\mathrm{C}}, \Lambda_i^{\mathrm{G}}\}_{\tau_i \in \tau}\}$, where Λ_i^{C} and Λ_i^{G} are the subsets of layers of τ_i assigned to π^{C} and π^{G}, respectively, we use C_i^{C} and C_i^{G} to denote the cumulative CPU and GPU execution times of the layers of τ_i in Λ_i^{C} and Λ_i^{G}, respectively, and they are calculated as

$$C_i^{\mathrm{C}} = \sum_{\tau_{i,j} \in \Lambda_i^{\mathrm{C}}} C_{i,j}^{\mathrm{C}} \text{ and } C_i^{\mathrm{G}} = \sum_{\tau_{i,j} \in \Lambda_i^{\mathrm{G}}} C_{i,j}^{\mathrm{G}}. \tag{1}$$

In order to derive a tight upper bound on the worst-case response time of a DNN task by reducing pessimism on the quantization/dequantization overhead on the CPU cluster and the data transfer overhead on the GPU cluster, we define the CPU and GPU segments of a task. A CPU segment $\phi_{i,p}^{\mathrm{C}}$ of τ_i is defined as the p-th subset of *consecutive* layers of τ_i assigned to CPU cluster π^{C}. Likewise, a GPU segment $\phi_{i,q}^{\mathrm{G}}$ of τ_i is defined as the q-th subset of *consecutive* layers of τ_i assigned to GPU cluster π^{G}. If all layers of a task τ_i are assigned to the CPU (GPU) cluster, τ_i has one CPU (GPU) segment. Otherwise, τ_i consists of an alternate sequence of CPU and GPU segments or the other way around. Let ϕ_i^{C} and ϕ_i^{G} denote the entire sets of CPU segments and GPU segments of τ_i, respectively, i.e., $\phi_i^{\mathrm{C}} = \bigcup \phi_{i,p}^{\mathrm{C}}$ and $\phi_i^{\mathrm{G}} = \bigcup \phi_{i,q}^{\mathrm{G}}$. Intermediate layers except the first and last layers in each $\phi_{i,p}^{\mathrm{C}}$ involve no quantization/dequantization overhead because the output of an intermediate layer can remain on the same bit precision and be directly used as input to the next layer. Similarly, intermediate layers except the first and the last layers in each $\phi_{i,q}^{\mathrm{G}}$ involve no data transfer overhead because the output of an intermediate layer can remain on the GPU cluster and be directly used as input to the next layer. Therefore, we can eliminate unnecessary overheads and thus derive a tight upper bound on the worst-case response time of a DNN task. Based on this, we use $O(\phi_{i,p}^{\mathrm{C}})$ and $O(\phi_{i,q}^{\mathrm{G}})$ to denote the cumulative overheads on a CPU segment $\phi_{i,p}^{\mathrm{C}}$ and a GPU segment $\phi_{i,q}^{\mathrm{G}}$, respectively, and they are calculated as

$$O(\phi_{i,p}^{\mathrm{C}}) = O_{i,p1}^{\mathrm{C}1} + O_{i,p2}^{\mathrm{C}2} \text{ and } O(\phi_{i,q}^{\mathrm{G}}) = O_{i,q1}^{\mathrm{G}1} + O_{i,q2}^{\mathrm{G}2}, \tag{2}$$

where $p1$ ($q1$) and $p2$ ($q2$) denote the first and the last layers of $\phi_{i,p}^{\mathrm{C}}$ ($\phi_{i,q}^{\mathrm{G}}$), respectively.

Then, the cumulative overheads O_i on τ_i is derived as

$$O_i = \sum_{\phi_{i,p}^{\mathrm{C}} \in \phi_i^{\mathrm{C}}} O(\phi_{i,p}^{\mathrm{C}}) + \sum_{\phi_{i,q}^{\mathrm{G}} \in \phi_i^{\mathrm{G}}} O(\phi_{i,q}^{\mathrm{G}}). \tag{3}$$

Under fixed-priority scheduling with layer-by-layer CPU/GPU allocation Λ, the worst-case response time (WCRT), w_i, of τ_i can be calculated iteratively in the following expression:

$$w_i^{a+1} = C_i^{\mathrm{C}} + C_i^{\mathrm{G}} + O_i + I_i(w_i^a) + B_i, \tag{4}$$

where $I_i(w_i^a)$ is the cumulative interference on layers in τ_i caused by higher-priority tasks and B_i is the cumulative blocking delay of layers in τ_i caused by lower-priority tasks. Note that the initial value w_i^0 is set to $C_i^{\mathrm{C}} + C_i^{\mathrm{G}} + O_i + B_i$, and the iteration halts when $w_i^{a+1} > D_i$ (unschedulable) or $w_i^{a+1} = w_i^a$ (the response time no larger than w_i^a).

Let $hp(\tau_i)$ be the set of all tasks with a priority higher than τ_i. The CPU and GPU segments of a task τ_i can be delayed by CPU and GPU segments of higher-priority tasks τ_h, i.e., $hp(\tau_i)$, respectively. The number of jobs of τ_h released during the execution of a single job of τ_i is at most $\left(1 + \left\lfloor \frac{w_i^a}{T_h} \right\rfloor\right)$. Then, the cumulative interference on τ_i, denoted by $I_i(w_i^a)$, is derived as

$$I_i(w_i^a) = \begin{cases} \sum_{\tau_h \in hp(\tau_i)} \left(1 + \left\lfloor \frac{w_i^a}{T_h} \right\rfloor\right) \cdot \left(C_h^{\mathrm{C}} + \sum_{\phi_{h,p}^{\mathrm{C}} \in \phi_h^{\mathrm{C}}} O(\phi_{h,p}^{\mathrm{C}})\right) & \text{if } \phi_i^{\mathrm{G}} = \emptyset, \quad \text{(5a)} \\ \sum_{\tau_h \in hp(\tau_i)} \left(1 + \left\lfloor \frac{w_i^a}{T_h} \right\rfloor\right) \cdot \left(C_h^{\mathrm{G}} + \sum_{\phi_{h,q}^{\mathrm{G}} \in \phi_h^{\mathrm{G}}} O(\phi_{h,q}^{\mathrm{G}})\right) & \text{if } \phi_i^{\mathrm{C}} = \emptyset, \quad \text{(5b)} \\ \text{(5a) + (5b),} & \text{otherwise.} \quad \text{(5c)} \end{cases}$$

In addition to the interference by higher-priority tasks, each CPU or GPU segment of a task τ_i can be further blocked by a currently-executing layer of a lower-priority task on the CPU or GPU cluster, respectively, because the execution of each layer is non-preemptive. Based on the design of System-wide scheduler, blocking may occur on only the first layer of a segment but not on the subsequent layers of the same segment. Thus, the cumulative blocking delay B_i of τ_i is derived as

$$B_i = |\phi_i^{\mathrm{C}}| \cdot B_{max}^{\mathrm{C}}(\tau_i) + |\phi_i^{\mathrm{G}}| \cdot B_{max}^{\mathrm{G}}(\tau_i), \tag{6}$$

where

$$B_{max}^{\mathrm{C}}(\tau_i) = \max_{j:\tau_l \in lp(\tau_i) \bigwedge \tau_{l,j} \in \Lambda_l^{\mathrm{C}}} (C_{l,j}^{\mathrm{C}} - 1), \tag{7}$$

$$B_{max}^{\mathrm{G}}(\tau_i) = \max_{k:\tau_m \in lp(\tau_i) \bigwedge \tau_{m,k} \in \Lambda_m^{\mathrm{G}}} (C_{m,k}^{\mathrm{G}} - 1), \tag{8}$$

while $lp(\tau_i)$ is the set of tasks with a priority lower than τ_i.

Then, we can check the schedulability of a task set as presented in the following lemma:

Lemma 1: A task set τ is schedulable if

$$\forall \tau_i \in \tau, w_i \leq T_i. \tag{9}$$

Proof: There are at most $\left(1 + \left\lfloor \frac{w_i^a}{T_h} \right\rfloor\right)$ jobs of any $\tau_h \in \tau$ that can be executed within an interval of length w_i^a, and thus, the maximum interference time on τ_i by tasks in $hp(\tau_i)$ in an interval of length w_i^a is upper-bounded by Eqs. (5a)-(5c). Each CPU or GPU segment is blocked at most once by the execution of a layer of a lower-priority task, and thus, the maximum blocking time of τ_i by tasks in $lp(\tau_i)$ is upper-bounded by Eq. (6). This means, if the right-hand-side (RHS) of Eq. (4) is equal to w_i^{a+1}, τ_i finishes its execution within w_i^{a+1} in any case. Therefore, w_i resulting from the iterative

process is an upper-bound of the response time of τ_i's jobs. Thus, if $w_i \leq T_i$ holds for all $\tau_i \in \tau$, τ is schedulable. ■

Allocation algorithm. We present how to assign a mapping from the layers of each individual task to CPU and GPU clusters such that an entire task set τ is deemed schedulable by the response time analysis under fixed-priority scheduling. Basically, our proposed layer-by-layer CPU/GPU allocation algorithm shown in Alg. 1 sets an initial assignment to the GPU-only assignment, where all layers of all tasks are allocated to the GPU cluster (Line 3), and checks from the highest-priority task whether a subset of its layers can be reallocated to the CPU cluster in an iterative way so that τ is deemed schedulable by the response time analysis (Lines 4–20).

Let us discuss the key intuition behind how our layer-by-layer CPU/GPU allocation algorithm works. First, most of layers show less execution time when executed on the GPU cluster than when executed on the CPU cluster even with quantization as observed in Fig. 1(b). This is because, in general, each layer in typical DNNs exhibits massive parallelism which is commonly exploited with the use of GPU cores. Thus, the sum of execution of layers in a task on both CPU and GPU clusters tends to increase when some layers are moved from the GPU cluster to the CPU cluster. Second, when some layers of a higher-priority task are moved to the CPU cluster, the interference on all lower-priority tasks whose layers are assigned to the GPU cluster only is decreased. Third, for each individual task, if multiple adjacent layers are allocated on the same resource type, i.e. composing a segment, each intermediate layer has no overhead and no blocking delay.

At the beginning of the k-th iteration step, $(k-1)$ highest-priority tasks have finished their resource allocation while guaranteeing their schedulability. During the k-th step, our allocation algorithm then seeks to select τ_k's layers to be allocated to the CPU cluster one-by-one (Lines 5–19). Let us discuss the effect of allocating a layer $\tau_{k,j}$ to the CPU cluster on a higher-priority task $\tau_h \in hp(\tau_k)$, itself (i.e., τ_k), and a lower-priority task $\tau_l \in lp(\tau_k)$, respectively. For a higher-priority task τ_h, its blocking delay on the CPU cluster increases if $C^{\mathrm{C}}_{k,j}$ is the largest non-preemptive chunk of lower-priority tasks allocated on the CPU cluster. For a task τ_k itself, its total execution time on the GPU cluster is decreased by $C^{\mathrm{G}}_{k,j}$, and its total execution time on the CPU cluster is increased by $C^{\mathrm{C}}_{k,j}$. In addition, its blocking delay B_k is increased if both the previous layer $\tau_{k,j-1}$ and next layer $\tau_{k,j+1}$ are allocated on the GPU cluster due to the increased number of CPU and GPU segments by one, respectively (to be addressed as $\Delta B_k(\tau_{k,j})$ in Eq. (10)). Likewise, its overhead O_k may also be increased depending on the allocation of $\tau_{k,j-1}$ and $\tau_{k,j+1}$ (to be addressed as $\Delta O_k(\tau_{k,j})$ in Eq. (10)). For a lower-priority task τ_l, the interference of $\tau_{k,j}$ on τ_l is decreased since it is assumed that all layers of τ_l are allocated on the GPU cluster in the k-th step.

Considering the effect of allocating one layer of τ_k to the CPU cluster, we now determine which layer to be allocated to the CPU cluster. A layer $\tau_{k,j}$ is said to be *CPU-eligible* in the k-th step if τ_k and $\forall \tau_h \in hp(\tau_i)$ are deemed schedulable by the response time analysis presented in Eq. (9) under the assumption that $\tau_{k,j}$ is allocated to the CPU cluster, while all other layers of all tasks are allocated as it is in the k-th step

Algorithm 1 Layer-by-layer CPU/GPU allocation

1: Input: $\tau = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$
2: Output: $\Lambda = \{\{\Lambda^{\mathrm{C}}_1, \Lambda^{\mathrm{G}}_1\}, \{\Lambda^{\mathrm{C}}_2, \Lambda^{\mathrm{G}}_2\}, \{\Lambda^{\mathrm{C}}_3, \Lambda^{\mathrm{G}}_3\}, ..., \{\Lambda^{\mathrm{C}}_n, \Lambda^{\mathrm{G}}_n\}\}$
3: $\forall \tau_k \in \tau, \Lambda^{\mathrm{G}}_k = \bigcup_{\forall \tau_{k,j} \in V_k} \{\tau_{k,j}\}$, $\Lambda^{\mathrm{C}}_k = \emptyset$ and $E(k) = \emptyset$
4: **for** $\tau_k \in \tau$ in descending order of priority **do**
5: **if** τ_k is unschedulable **or** $\forall \tau_l \in lp(\tau_k)$ is schedulable with Λ by Eq. (9) **then**
6: **return** Λ
7: **end if**
8: **for** $\tau_{k,j} \in \Lambda^{\mathrm{G}}_k$ **do**
9: Set $\hat{\Lambda}^{\mathrm{G}}_k = \Lambda^{\mathrm{G}}_k \setminus \{\tau_{k,j}\}$ and $\hat{\Lambda}^{\mathrm{C}}_k = \Lambda^{\mathrm{C}}_k \bigcup \{\tau_{k,j}\}$
10: Set $\hat{\Lambda} = \{\hat{\Lambda}^{\mathrm{C}}_k, \hat{\Lambda}^{\mathrm{G}}_k\} \bigcup \Lambda \setminus \{\Lambda^{\mathrm{C}}_k, \Lambda^{\mathrm{G}}_k\}$
11: **if** τ_k **and** $\forall \tau_h \in hp(\tau_k)$ is schedulable with $\hat{\Lambda}$ by Eq. (9) **then**
12: $E(k) = E(k) \bigcup \{\tau_{k,j}\}$
13: **end if**
14: **end for**
15: **if** $E(k) \neq \emptyset$ **then**
16: Find $\tau_{k,s} \in E(k)$ according to Eq. (10)
17: $\Lambda^{\mathrm{G}}_k = \Lambda^{\mathrm{G}}_k \setminus \{\tau_{k,s}\}$, $\Lambda^{\mathrm{C}}_k = \Lambda^{\mathrm{C}}_k \bigcup \{\tau_{k,s}\}$ and $E(k) = \emptyset$
18: **go to** Line 5
19: **end if**
20: **end for**

(Lines 8–14). Note that, by the assumption, $\tau_{k,j}$ might additionally impose the blocking delay on its higher-priority tasks that have finished their allocation on the CPU cluster, which requires to check whether $\forall \tau_h \in hp(\tau_i)$ remains schedulable (Line 11). With this, we can ensure that the schedulability of τ_k is not affected by resource allocation in the next iteration steps. Thus, our resource allocation algorithm builds a solution incrementally without backtracking. Let denote $E(k)$ as a subset of layers of τ_k that is CPU-eligible in the k-th step.[2] We then select one CPU-eligible layer $\tau_{k,s}$ in $E(k)$ to be allocated to the CPU cluster such that the increase in the response time of τ_k is minimized (Lines 15–19), and it is presented as

$$\tau_{k,s} \coloneqq \underset{\tau_{k,j} \in E(k)}{\arg\min} \left(C^{\mathrm{C}}_{k,j} - C^{\mathrm{G}}_{k,j} + \Delta B_k(\tau_{k,j}) + \Delta O_k(\tau_{k,j})\right), \quad (10)$$

where $\Delta B_k(\tau_{k,j})$ and $\Delta O_k(\tau_{k,j})$ are the amounts of varied blocking delay and overhead, respectively, when $\tau_{k,j}$'s allocation is changed from the GPU to CPU cluster, derived as

$$\Delta B_k(\tau_{k,j}) = \begin{cases} B^{\mathrm{C}}_{max}(\tau_k) + B^{\mathrm{G}}_{max}(\tau_k) & \text{if } \tau_{k,j\text{-}1}, \tau_{k,j+1} \in \Lambda^{\mathrm{G}}_k, \\ -B^{\mathrm{C}}_{max}(\tau_k) - B^{\mathrm{G}}_{max}(\tau_k) & \text{if } \tau_{k,j\text{-}1}, \tau_{k,j+1} \in \Lambda^{\mathrm{C}}_k, \\ 0, & \text{otherwise,} \end{cases}$$

$$\Delta O_k(\tau_{k,j}) = \begin{cases} O^{\mathrm{G2}}_{k,j\text{-}1} + O^{\mathrm{C}}_{k,j} + O^{\mathrm{G1}}_{k,j+1} & \text{if } \tau_{k,j\text{-}1}, \tau_{k,j+1} \in \Lambda^{\mathrm{G}}_k, \\ -O^{\mathrm{C2}}_{k,j\text{-}1} - O^{\mathrm{G}}_{k,j} - O^{\mathrm{C1}}_{k,j+1} & \text{if } \tau_{k,j\text{-}1}, \tau_{k,j+1} \in \Lambda^{\mathrm{C}}_k, \\ O^{\mathrm{G2}}_{k,j\text{-}1} + O^{\mathrm{C1}}_{k,j} - O^{\mathrm{G2}}_{k,j} - O^{\mathrm{C1}}_{k,j+1} & \text{if } \tau_{k,j\text{-}1} \in \Lambda^{\mathrm{G}}_k, \tau_{k,j+1} \in \Lambda^{\mathrm{C}}_k, \\ -O^{\mathrm{C2}}_{k,j\text{-}1} + O^{\mathrm{C2}}_{k,j} - O^{\mathrm{G1}}_{k,j} + O^{\mathrm{G1}}_{k,j+1} & \text{if } \tau_{k,j\text{-}1} \in \Lambda^{\mathrm{C}}_k, \tau_{k,j+1} \in \Lambda^{\mathrm{G}}_k. \end{cases}$$

Note that both $\Delta B_k(\tau_{k,j})$ and $\Delta O_k(\tau_{k,j})$ depend on the allocation of $\tau_{k,j\text{-}1}$ and $\tau_{k,j+1}$. As an example, let us consider a case when both $\tau_{k,j\text{-}1}$ and $\tau_{k,j+1}$ are allocated on the GPU cluster. Allocating $\tau_{k,j}$ to the CPU cluster increases the CPU and GPU segments by one, respectively, since the three layers become belonging to different segments, respectively, yielding the increase of the blocking delay by $B^{\mathrm{C}}_{max}(\tau_k) + B^{\mathrm{G}}_{max}(\tau_k)$. Also, it additionally causes i) the time to copy the output of $\tau_{k,j\text{-}1}$ to the CPU memory ($O^{\mathrm{G2}}_{k,j\text{-}1}$), ii) time to quantize and dequantize the input and output of $\tau_{k,j}$ ($O^{\mathrm{C}}_{k,j}$), and iii) the time to copy the input of $\tau_{k,j+1}$ to the GPU memory ($O^{\mathrm{G1}}_{k,j+1}$).

[2]Although our resource allocation algorithm focuses on G1, we can also make a few highly sensitive layers according to the sensitivity analysis shown in Fig. 1(c) forced to run on the GPU cluster only by simply excluding them from E(k), preventing a task from a significant accuracy loss (considering G2) offline. In evaluation shown in Section VI, no layer is excluded from E(k).

In the k-th iteration step, we repeat selecting a layer $\tau_{k,s}$ one-by-one in $E(k)$ (by Line 18) until there is no CPU-eligible layer (i.e., $E(k)$ is empty in Line 15).

Runtime Complexity. Let n denote the number of DNN tasks in a task set. At each iteration step k in Alg. 1, our algorithm tries to find a subset $E(k)$ among all layers of τ_k by checking Eq. (9) with the complexity of $O(N_k \cdot n^3)$. Repeating this step until $E(k)$ becomes empty for all tasks, Alg. 1 requires $O(\max_{\tau_k \in \tau} N_k^2 \cdot n^4)$.

V. Runtime Layer Migration

Although the layer-by-layer CPU/GPU allocation obtained by Alg. 1 can ensure the execution of DNN tasks without violating their timing constraints, it may suffer from the accuracy loss if some layers that result in low accuracy when quantized are executed on the CPU cluster. Therefore, this section addresses the following *runtime layer migration decision* problem.

Definition 2 (Runtime Layer Migration Decision): Given the layer-by-layer CPU/GPU allocation Λ obtained by Alg. 1, decide an online migration (resource reallocation) of each layer between CPU and GPU clusters such that the overall accuracy loss of τ is minimized (i.e., achieving G2) without violating any timing constraint (i.e., achieving G1).

Our solution consists of (I1) the slack management part that reclaims unused CPU and GPU resources (called slack) caused by early completion of a layer's execution than its worst-case execution time, and (I2) the resource reallocation part that effectively utilizes them to change the resource allocation (from CPU to GPU, or from GPU to CPU) of some layers, such that I1 and I2 should achieve both G1 and G2 in Sec. III-A.

For I1, we propose the following slack reclamation scheduler. The scheduler is invoked upon (i) release of a new layer or (ii) completion of a layer on either CPU or GPU cluster. Let $S^{\mathrm{C}}(t_{cur})$ and $S^{\mathrm{G}}(t_{cur})$ denote available slacks on the CPU and GPU clusters at the current time instant t_{cur}, respectively, with $S^{\mathrm{C}}(0) = S^{\mathrm{G}}(0) = 0$. Upon completion of a layer $\tau_{i,j}$ of τ_i on the GPU cluster at time t_{cur}, we compare the actual execution time (denoted as $AC^{\mathrm{G}}_{i,j}$) with its worst-case time allotment $C^{\mathrm{G}}_{i,j}$ and increase the GPU slack $S^{\mathrm{G}}(t_{cur})$ by $C^{\mathrm{G}}_{i,j} - AC^{\mathrm{G}}_{i,j}$. If the GPU cluster has been idle during successive invocations of either (i) or (ii), its slack value is decreased by the amount of idle time until it becomes zero. We update the CPU slack $S^{\mathrm{C}}(t_{cur})$ in the same way.

As to I2, we need to judge which layer should change the resource allocation from CPU to GPU (or from GPU to CPU). According to the sensitivity analysis shown in Fig. 1(c), we refer to a layer as an *unfavorable* one if the accuracy is lowered when quantized as compared to full precision (i.e., when the accuracy difference between a single-layer quantization and full precision becomes negative in Fig. 1(c)) and as a *favorable* one, otherwise. Upon release of an unfavorable layer $\tau_{i,u}$ of τ_i on the CPU cluster at time t_{cur}, our migration algorithm checks whether the unfavorable layer $\tau_{i,u}$ can be reallocated to the GPU cluster and scheduled by using the GPU slack $S^{\mathrm{G}}(t_{cur})$. Once $\tau_{i,u}$ is reallocated to the GPU cluster, the GPU slack $S^{\mathrm{G}}(t_{cur})$ will be decreased by the amount of execution time of $\tau_{i,u}$ on the GPU cluster. Likewise, our migration algorithm checks whether the favorable layer $\tau_{i,f}$ can be reallocated to the CPU cluster.

The remaining step is to develop the migration algorithm, which selects a layer whose resource allocation is changed to achieve G2 without compromising G1. Let $\tau_{i,c}$ denote a candidate unfavorable (favorable) layer of τ_i to be reallocated to the GPU (CPU) cluster at time t_{cur} and J_i^* denote an active job of τ_i that includes $\tau_{i,c}$. Regarding G1, we need to guarantee that the resource allocation change for $\tau_{i,c}$ does not compromise (G1a) the schedulability of J_i^* and (G1b) the schedulability of all active jobs except J_i^* and all jobs of tasks in τ to be released in future. When it comes to G2, we should consider the degree of being unfavorable/favorable of each layer to decide one among candidate layers whose resource allocation change does not violate both G1a and G1b.

We first explain how to address G1a. We derive a sufficient condition for the job J_i^* to remain schedulable after changing $\tau_{i,c}$'s resource allocation from one to the other cluster at time t_{cur}. Suppose no other layer will change its allocation after t_{cur}. Consider a new task τ_x with $T_x = d_i^* - t_{cur}$ and $V_x = \{\tau_{i,c}, ..., \tau_{i,|N_i|}\}$, where d_i^* is the absolute deadline of J_i^* and its layer-by-layer CPU/GPU allocation $\{\Lambda_x^{\mathrm{C}}, \Lambda_x^{\mathrm{G}}\}$ obtained from $\{\Lambda_i^{\mathrm{C}}, \Lambda_i^{\mathrm{G}}\}$ by i) removing already executed layers $\{\tau_{i,1}, ..., \tau_{i,c-1}\}$ of J_i^* and ii) newly allocating a candidate layer $\tau_{i,c}$ to the cluster to be migrated. Also consider that the new task τ_x has the same priority as that of τ_i. Using $\{\Lambda_x^{\mathrm{C}}, \Lambda_x^{\mathrm{G}}\}$, the following lemma can guarantee the schedulability of J_i^* after the migration of $\tau_{i,c}$ at t_{cur}.

Lemma 2: Consider a new layer-by-layer CPU/GPU allocation $\Lambda' = \{\Lambda_x^{\mathrm{C}}, \Lambda_x^{\mathrm{G}}\} \bigcup \Lambda \setminus \{\Lambda_i^{\mathrm{C}}, \Lambda_i^{\mathrm{G}}\}$. Suppose an active job J_i^* satisfies the following condition under Λ'. Then, it is schedulable under fixed-priority scheduling unless any layer migration other than $\tau_{i,c}$ occurs after t_{cur}.

$$C_x^{\mathrm{C}} + C_x^{\mathrm{G}} + O_x + I_i(T_x) + B_x \leq T_x. \tag{13}$$

where $I_i(T_x)$ is the maximum interference on τ_x in any interval of length T_x calculated by Eqs. (5a)-(5c), and B_x is the maximum blocking delay of τ_x calculated by Eq. (6).

Proof: The term $C_x^{\mathrm{C}} + C_x^{\mathrm{G}} + O_x$ is the sum of J_i^*'s remaining execution time on the CPU and GPU clusters and overheads at t_{cur}. T_x is the remaining time to the deadline of J_i^* at t_{cur}. Based on the same reasoning shown in the proof of Lemma 1, $I_i(T_x)$ and B_x are upper-bounds on the maximum interference on J_i^* and the maximum blocking time of J_i^*, respectively. This implies that J_i^* will finish its execution by d_i^* if Inequality (13) holds, thus proving the lemma. ■

For G1b, we must guarantee the schedulability of all active jobs except J_i^* at time t_{cur} and all jobs released in future. While this can be achieved by allowing J_i^* to consume all the remaining slack in a greedy manner, we may reserve some of the remaining slack for more (un)favorable layers, which can address G2. To this end, we first construct a candidate list for GPU migration, referred as M^{G} as a set of all unfavorable layers that currently allocated on the CPU cluster and a candidate list for CPU migration, referred as M^{C} as a set of all favorable layers that currently allocated on the GPU cluster. We refer to layers that result in lower and higher accuracy when quantized as being more unfavorable and favorable to quantization, respectively, according to the sensitivity analysis. Then,

we assign the GPU and CPU slacks, $S^{\mathrm{G}}(t_{cur})$ and $S^{\mathrm{C}}(t_{cur})$, to the layers in M^{G} and M^{C} *in descending orders* of more unfavorable and favorable layers, respectively. The rationale for such a prioritized slack distribution is that reallocating a more unfavorable layer to the GPU cluster and reallocating a more favorable layer to the CPU cluster yield higher accuracy improvement. To do so, for CPU-to-GPU migration, upon release of an unfavorable layer $\tau_{i,u}$ of τ_i on the CPU cluster at time t_{cur}, we first reserve $S^{\mathrm{G}}(t_{cur})$ for more unfavorable layers than $\tau_{i,u}$ in M^{G}, denoted as $\hat{M}^{\mathrm{G}}(\tau_{i,u})$, and check whether a layer $\tau_{i,u}$ can be reallocated and executed on the GPU cluster by using the remaining GPU slack. The same policy can be applied upon release of a favorable layer $\tau_{i,f}$ of τ_i on the GPU cluster at time t_{cur}. Therefore, the following lemma addresses not only G1b, but also G2.

Lemma 3: Under a layer-by-layer CPU/GPU allocation Λ, suppose i) a task set τ is deemed schedulable by satisfying Eq. (9) and ii) an unfavorable layer $\tau_{i,u}$ of J_i^* changes its resource allocation from the CPU to GPU cluster at time t_{cur} and satisfies Eq. (14). Then, all active jobs except J_i^* and all jobs released in future are schedulable under fixed-priority scheduling unless any layer migration other than $\tau_{i,u}$ occurs after time t_{cur}.

$$C^{\mathrm{G}}_{i,u} - \Delta O_i(\tau_{i,u}) \leq S^{\mathrm{G}}(t_{cur}) - \sum_{\tau_{k,h} \in \hat{M}^{\mathrm{G}}(\tau_{i,u})} (C^{\mathrm{G}}_{k,h} - \Delta O_i(\tau_{k,h})). \quad (14)$$

Likewise, under a layer-by-layer CPU/GPU allocation Λ, suppose i) a task set τ satisfies Eq. (9) and ii) a favorable layer $\tau_{i,f}$ of J_i^* changes its resource allocation from the GPU to CPU cluster at time t_{cur} and satisfies Eq. (15). Then, all active jobs except J_i^* and all jobs released in future are schedulable under fixed-priority scheduling unless any layer migration other than $\tau_{i,f}$ occurs after time t_{cur}.

$$C^{\mathrm{C}}_{i,f} + \Delta O_i(\tau_{i,f}) \leq S^{\mathrm{C}}(t_{cur}) - \sum_{\tau_{k,h} \in \hat{M}^{\mathrm{C}}(\tau_{i,f})} (C^{\mathrm{C}}_{k,h} + \Delta O_i(\tau_{k,h})), \quad (15)$$

where $\hat{M}^{\mathrm{C}}(\tau_{i,f})$ is a set of more favorable layers than $\tau_{i,f}$ in M^{C}.

Proof: Suppose that i) and ii) hold but there exists a job (other than J_i^*) that misses its deadline; we show the contradiction of this supposition. Let us first consider the case of reallocating an unfavorable layer $\tau_{i,u}$ of J_i^* to the GPU cluster at t_{cur}. Suppose a job of a task τ_k (whose release time and deadline are t_a and $t_a + T_k$) misses its deadline. Consider an interval $[t_a - l_k, t_a + T_k)$ such that $[t_a - l_k, t_a)$ is the maximum consecutive busy interval (in which the GPU cluster is occupied when all layers execute for their WCETs), and the GPU cluster is idle in $[t_a - l_k - 1, t_a - l_k)$. Then, $S^G(t_{cur}) > 0$, where $t_{cur} \in [t_a - l_k, t_a + T_k)$, implies that some layers whose executions are in $[t_{cur}, t_a + T_k)$ under the worst-case execution scenario are completed earlier than their WCETs by $S^G(t_{cur})$. Let us consider two cases where (a) $t_{cur} \leq t_a - l_k$ and (b) $t_{cur} > t_a - l_k$.

For Case (a), we claim that $S^G(t_a - l_k)$ becomes zero based on the following reasoning. Let t_x $(\leq t_a - l_k - 1)$ denote the end of the busy interval right before $[t_a - l_k, t_a)$. $S^G(t_{cur})$ is at most $\max(t_x - t_{cur}, 0)$ because $S^G(t_{cur})$ is increased by the amount of early completion of each layer's execution than its WCET (i.e., no larger than $\max(t_x - t_{cur}, 0)$). Thus, any layer migration at $t_{cur} \leq t_a - l_k$ does not affect the execution in $[t_a - l_k, t_a + T_k)$. This contradicts the supposition that τ is schedulable but there exists a job that misses its deadline.

For Case (b), we consider two sub-cases where (b1) $t_a \leq t_{cur}$ and (b2) $t_{cur} < t_a$. For Cases (b1) and (b2), $S^G(t_{cur})(> 0)$ implies that the sum of an actual interference and blocking time imposed on the job missing its deadline in $(t_a, t_a + T_k]$ is decreased by $S^G(t_{cur})$ and $S^G(t_{cur}) - X$, respectively, where X is the amount of idle time in $[t_{cur}, t_a)$ without migrating $\tau_{i,u}$. For Case (b2), the amount of execution of a migrating layer $\tau_{i,u}$ in the idle interval that belongs to $[t_{cur}, t_a)$ does not impose additional delay on the job missing its deadline in $[t_a, t_a + T_k)$. A layer migration at t_{cur} can delay the execution of the job by at most $S^G(t_{cur})$ and $S^G(t_{cur}) - X$ for Cases (b1) and (b2), respectively. Similar to Case (a), this contradicts the supposition.

We can similarly prove the case of reallocating a favorable layer $\tau_{i,f}$ of J_i^* to the CPU cluster at t_{cur}. ■

Upon release of either an unfavorable layer or a favorable layer on the CPU or GPU cluster, respectively, we reallocate the layer to the GPU or CPU cluster if both Lemmas 2 and 3 are satisfied, i.e., Eqs. (13) and (14) for an unfavorable layer and Eqs. (13) and (15) for a favorable layer. Based on Lemmas 2 and 3, our runtime migration algorithm is able to efficiently mitigate the overall accuracy loss of τ due to quantization while guaranteeing no deadline miss.

Runtime Complexity. At each invocation (either release or completion of a layer), our algorithm updates the slack with the complexity of $O(1)$. Then, our algorithm checks the possibility of a layer migration with Eqs. (13)–(15) with the complexity of $O(n^2)$. Thus, the total complexity is $O(n^2)$.

VI. Evaluation

We have implemented and evaluated LaLaRAND on top of PyTorch, one of the state-of-the-art ML frameworks. DNNs designed with PyTorch are easily adapted to LaLaRAND without requiring any modifications, which makes LaLaRAND universal and applicable to existing DNNs. We demonstrate the capability of LaLaRAND making a significant improvement of schedulability and accuracy. We use two metrics: *schedulability ratio* and *accuracy*. The schedulability ratio is defined as the percentage of schedulable task sets of the total number of generated task sets by using an algorithm. The accuracy is defined as the percentage of correctly inferred images over total tested images. We have also measured the overhead of LaLaRAND.

A. Experimental Setup

Hardware and Software. We conduct experiments on the NVIDIA Jetson Xavier board with 8-core Carmel CPUs and a 512-core Volta GPU. We use PyTorch v1.4.0 with CUDA 10.2 and cuDNN 8.0.2 for implementation. LaLaRAND assumes that the CPU and GPU cores perform computations using `QUInt8` and `FP32` values, respectively, that are supported in hardware by common processors. LaLaRAND employs the qnnpack library [36] provided in PyTorch for layer-level quantization/dequantization.[3] For CPU/GPU scheduling on the

[3]Although the GPU cores in the Xavier board support both `FP16` (half-precision floating-point) and `QUInt8` formats as well, PyTorch does not yet support quantized operator implementations for GPU. Thus, LaLaRAND applies 8-bit linear quantization for CPU execution only.

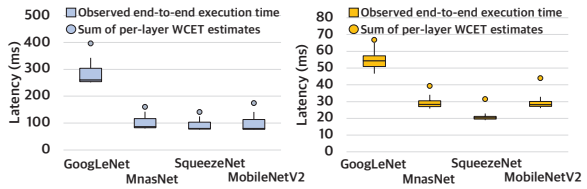


Fig. 4. Sum of each layer's WCET estimates (marked as a dot) and distribution of measured end-to-end execution time (marked as a candlestick) on (a) CPU (`QUInt8`) and (b) GPU in Nvidia Jetson Xavier for GoogLeNet, MnasNet, SqueezeNet, and MobileNetV2 with a 602KB image

Xavier board, LaLaRAND utilizes seven CPU cores as a CPU cluster and one GPU device with one CPU core as a GPU cluster. Each cluster executes a single layer at a time by utilizing all available cores in parallel.[4] For transparent memory handler, LaLaRAND explicitly copies the memory buffer between CPU (host) and GPU (device) through `CudaMemcpy()` calls. Note that CUDA provides other GPU memory management methods, such as pinned memory and unified memory. LaLaRAND can also support those memory management methods by replacing `CudaMemcpy()` with corresponding CUDA API calls, i.e., `CudaHostAlloc()` for pinned memory and `CudaMallocManaged()` for unified memory.

DNN Models and Datasets. We use four popular DNN models for image classification in our evaluation, i.e, GoogLeNet, MnasNet, SqueezeNet, and MobileNetV2, that are tested with the ImageNet dataset for 1,000 images.

Time measurement. We take a measurement-based approach to estimate the worst-case execution time of individual layers of DNN models on CPU and GPU clusters as well as their quantization/dequantization and data transfer overheads. Our experiment measures each layer's execution time on CPU and GPU clusters by running 1,000 times, respectively, and takes the maximum value among the measured ones as the WCET. The same applies to estimate quantization/dequantization and data transfer overheads at layer level. Fig. 4 shows the sum of each layer's worst-case execution time estimates (marked as a dot) and the distribution of observed end-to-end execution time (marked as a candlestick) on CPU and GPU clusters, respectively, for four DNN models. The box in each candlestick represents the range of values between quartiles (25 and 75 percentiles). The horizontal line in the middle of the box is the median and the vertical line shows the maximum and minimum values. For example, the sums of each layer's WCET estimates of GoogLeNet on CPU and GPU clusters are 396.6ms and 66.9ms, respectively, while its average end-to-end execution times on CPU and GPU clusters are 260.6ms and 54.4ms, respectively.

Approaches to be compared. We compare the following five different approaches:

- LaLaRAND: layer-by-layer CPU/GPU allocation in Alg. 1 and runtime layer migration in Sec. V;
- LaLaRAND w/o QU: layer-by-layer CPU/GPU allocation in Alg. 1 without CPU-friendly quantization;
- SOWD: resource-efficient schedulability-oblivious CPU/GPU workload distribution;
- DART: pipelined data-parallel CPU/GPU scheduling [6]; and
- BaseGPU: GPU-only allocation (vanilla PyTorch).

SOWD sorts all layers of all tasks in increasing order of their execution time ratios of CPU to GPU and allocates layers having higher ratios to GPU and layers having lower ratios to CPU until the utilization of the CPU and GPU clusters is balanced to fully utilize resources efficiently. DART employs a pipeline-based resource allocation and scheduling architecture, where CPUs and a GPU are arranged into nodes, and subsets of consecutive layers of each task are allocated to different nodes and executed in a pipeline manner. In comparison of the schedulability ratio, we use its own schedulability analysis based on the delay composition theorem for DART and our proposed analysis in Eq. (9) for BaseGPU. Note that, to make a fair comparison with LaLaRAND, we also employ CPU-friendly quantization for SOWD and DART. Note that other related techniques [18]–[22] mentioned in Sec. VII are not included in this comparison, since they did not deal with the timing constraints, rendering them infeasible for real-time DNN tasks.

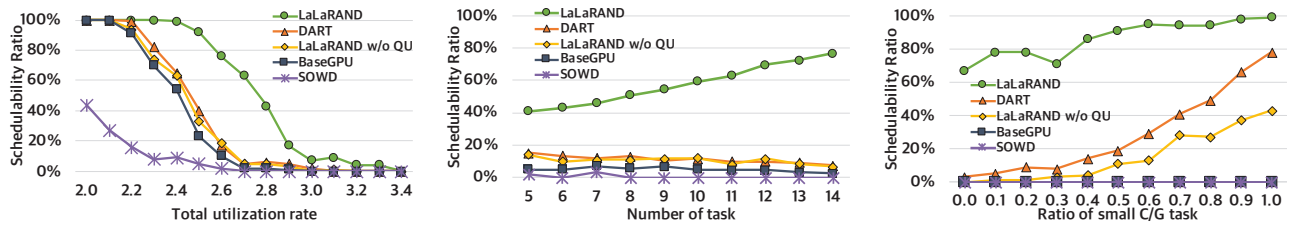
B. Experimental Results

Schedulability. We generate DNN task sets by using the worst-case execution time of each layer and overheads measured via the experiments. We randomly generate 7,500 task sets, where the period of each task is determined according to the UUniFast algorithm [37], which has been widely used for the generation of synthetic task sets. Each task set has 10 DNN tasks whose DNN models are randomly chosen among the four image classification DNN models and priorities are determined by rate-monotonic (RM) [38].

Fig. 5(a) compares the percentage of schedulable task sets by five resource allocation approaches while varying the total utilization rate $U_{\tau,\pi}$ of a task set from 2.0 to 3.4.[5] LaLaRAND exhibits high capability in finding schedulable task sets in that LaLaRAND finds 56%, 64%, 80%, and 633% more schedulable task sets than DART, LaLaRAND w/o QU, BaseGPU, and SOWD, respectively. Note that LaLaRAND is shown to dominate BaseGPU, because our layer-by-layer CPU/GPU allocation in Alg. 1 uses the solution of BaseGPU as an initial input to the allocation process. The performance gap between LaLaRAND and the other approaches is shown to become larger as $U_{\tau,\pi}$ increases. As $U_{\tau,\pi}$ increases, it becomes more difficult to make task sets schedulable by utilizing the GPU cluster only. Thus, BaseGPU hardly finds any schedulable task sets when $U_{\tau,\pi} \geq 2.7$. Although SOWD can fully utilize all resources in an efficient way through a fine-grained layer-level resource allocation, it does not consider schedulability also affected by execution dependencies and overheads among layers in each task, yielding worse schedulability performance than BaseGPU. Under LaLaRAND w/o QU, the execution time ratio of CPU to GPU for each layer is too high to allocate just a few layers of a task to the CPU cluster, making the task easily deemed unschedulable. Under DART, the workload of each DNN task is distributed among resource nodes so that the utilization of nodes is balanced. In contrast, LaLaRAND tries to allocate as many layers of higher-priority tasks to the CPU cluster, while leaving all layers of lower-priority tasks on the GPU cluster so as to effectively reduce the interference imposed on lower-priority tasks. Therefore, the performance gap between

[4]PyTorch supports parallel processing of a single kernel through the OpenMP API for CPUs.

[5]We define the total utilization rate $U_{\tau,\pi}$ of a task set on a CPU/GPU platform as $\sum_i \left(\sum_j (C^{\mathrm{C}}_{i,j} + C^{\mathrm{G}}_{i,j})/2 \right) / T_i$.



(a) (b) (c)

Fig. 5. Schedulability ratios with (a) varying $U_{\tau,\pi}$ when the number of tasks (n) is 10, (b) varying n when $U_{\tau,\pi} = 2.7$, and (c) varying the ratio of small C/G ratio task when $U_{\tau,\pi} = 2.7$ and $n = 10$

LaLaRAND and the other approaches can be interpreted as the benefit of taking the effect of allocating a layer to the CPU cluster on the schedulability into account by our layer-level resource allocation.

We also varied the number of tasks (n) in a task set to identify its effect on schedulability. We generate 5,000 additional task sets in total while varying the number of tasks from 5 to 14 when $U_{\tau,\pi}$ is set to 2.7 as shown in Fig. 5(b). LaLaRAND is shown to outperform the other approaches for all values of the number of tasks. One interesting observation is that the performance of LaLaRAND is getting better as the number of tasks increases, while the other approaches show the opposite trend. For example, as the number of tasks increases from 5 to 14, the schedulability ratio of LaLaRAND is increased from 41% to 77%, but those of DART, LaLaRAND w/o QU, BaseGPU, and SOWD are decreased from 15% to 8%, from 14% to 7%, from 5% to 3%, and from 2% to 0%, respectively. As the number of tasks increases, each task in a task set is more likely to have a smaller utilization. LaLaRAND can allocate more layers of higher-priority tasks to the CPU cluster without compromising their schedulability and effectively reduce the interference on lower-priority tasks allocated on the GPU cluster only, resulting in better schedulability.

We also varied the ratio of small C/G ratio tasks to identify the effect of task's execution time ratio of CPU to GPU on schedulability. We consider tasks using GoogLeNet as small C/G ratio tasks because GoogLeNet shows the smallest average execution time ratio of CPU to GPU at the layer level among four DNN models. We generate 5,500 additional task sets while varying the ratio of small C/G ratio tasks from 0 to 100% when $U_{\tau,\pi} = 2.7$ and $n = 10$ as shown in Fig. 5(c). LaLaRAND is shown to outperform the other approaches for all ratios. The performance gap between LaLaRAND and the others becomes larger as the ratio of small C/G ratio tasks decreases. For example, under LaLaRAND, 78% of the task sets are schedulable when the ratio of small C/G ratio tasks is 0.1, while DART, LaLaRAND w/o QU, BaseGPU, and SOWD only make 5%, 1%, 0%, and 0% of the task sets schedulable, respectively. Such an improvement can be interpreted as the benefit of efficient use of both CPU and GPU resources by our fine-grained layer-by-layer CPU/GPU allocation algorithm. As a result, LaLaRAND can find more schedulable task sets under limited computing resources.

In addition, we conducted a case study on the Xavier board to demonstrate the effect of our flexible layer-by-layer CPU/GPU scheduling on the end-to-end response time of DNN tasks and schedulability. The task set used in this case study consists of the following four tasks: $\tau_1(T_1 = 120$, GoogLeNet), $\tau_2(320$, MnasNet), $\tau_3(320$, GoogLeNet), and

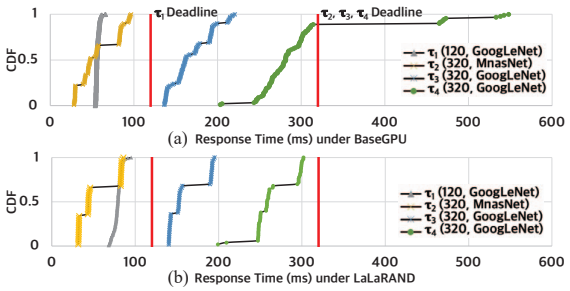


Fig. 6. Response time CDF of four DNN tasks by (a) BaseGPU and (b) LaLaRAND

$\tau_4(320$, GoogLeNet). The tasks are scheduled under RM. Fig. 6 shows the measured response time CDF of four tasks by BaseGPU and LaLaRAND. Under BaseGPU, task τ_4 having the lowest priority misses its deadline and shows long tail latency with the maximum observed response time of $548ms$. This is because all tasks are executed on GPU only under BaseGPU, so all higher-priority tasks impose a larger interference on τ_4. On the other hand, LaLaRAND achieves $1.8\times$ reduction in the maximum observed response time of τ_4 over BaseGPU, making all tasks schedulable. Although LaLaRAND yields $1.4\times$ increment in the maximum observed response time of τ_1 over BaseGPU by scheduling some of τ_1's layers on the CPU cluster, LaLaRAND can still guarantee the schedulability of all tasks and effectively reduce the interference on lower-priority tasks including τ_4, resulting in better schedulability. Note that the worst-case response time of τ_4 derived by our response time analysis in Eq. (4) is $319.9ms$, while the maximum observed response time of τ_4 by LaLaRAND is $301.7ms$.

Accuracy. We now show how LaLaRAND effectively mitigates the accuracy loss due to quantization by runtime layer migration. With the generated task sets, we consider 100 schedulable task sets by both LaLaRAND and DART but not by BaseGPU whose total utilization rates are between 2.5 and 3.0 and $n = 5$. For each task in a task set, we measure the top-1 classification accuracy with the ImageNet dataset for 1,000 images. Fig. 7 shows the distributions of difference in accuracy for LaLaRAND with and without runtime layer migration and DART relative to BaseGPU (full precision), respectively. The vertical green line represents a reference accuracy by BaseGPU, and the red dot is marked at the corresponding difference in accuracy for each task under each method based on the green line while its brightness represents the frequency. Note that we exclude the results showing no accuracy difference. LaLaRAND with our runtime layer migration algorithm shows a very small accuracy drop (less than 0.4%p), while the case of no runtime layer migration shows a significant accuracy loss over a wide range from -5.3%p to -18.2%p. Our runtime layer migration

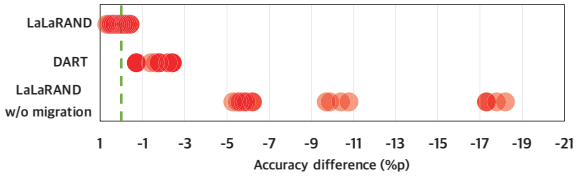


Fig. 7. Distribution of accuracy difference with LaLaRAND, DART, and LaLaRAND without layer migration relative to BaseGPU (the green line)

algorithm effectively utilizes 36,068ms and 13,293ms of the GPU and CPU slacks (6.2% and 23.8% of the total amounts of worst-case workloads allocated on the GPU and CPU clusters), respectively, to reallocate 93.4% of unfavorable layers to the GPU cluster and 0.27% of favorable layers to the CPU cluster on average for a task set. We observed that all unfavorable layers reallocated to the GPU cluster belong to higher-priority tasks, while favorable layers reallocated to the CPU cluster are from tasks with different priorities; this is because, our offline CPU/GPU allocation algorithm allocates higher-priority tasks' layers to the CPU cluster first. Nevertheless, our algorithm prioritizes the distribution of the GPU and CPU slacks to more unfavorable and favorable layers in a task set, respectively, resulting that none of the tasks in a task set suffers from a severe accuracy drop (only up to -0.4%p). DART shows an accuracy loss over a range from -0.7%p to -2.4%p, yielding a higher accuracy drop than LaLaRAND. We interestingly note that our runtime layer migration algorithm can even improve the accuracy by up to 0.7%p for 11% of all tasks against BaseGPU. Such an accuracy improvement can be interpreted as the benefit of efficiently utilizing available slacks on the CPU and GPU clusters at runtime to migrate unfavorable and favorable layers to GPU and CPU clusters, respectively, without violating any timing constraint.

Overheads analysis. We analyze the overhead of LaLaRAND for 1) DNN initialization, 2) per-layer scheduling, 3) intermediate data handling, and 4) memory usage. Note that we measure the overhead with four DNN models used in the performance evaluation. First of all, to initialize a DNN task, LaLaRAND incurs an average delay of 53.7 seconds. It seems quite long but still affordable since it happens only once before starting the DNN task. After initialization, LaLaRAND imposes per-layer scheduling overhead composed of IPC communication and decision making costs that respectively take 67.9μs and 2.3μs on average (up to 98.3μs and 57μs, respectively). This delay is not too critical since it is relatively short compared to the layer computation. And, this delay does not hinder the timing guarantee, since the LaLaRAND scheduler takes this delay into account for scheduling decisions. A few layers may incur data handling overhead to keep data consistency between two layers assigned to different resources. It only takes 2.3ms on average, relatively short delay compared to the computation time of each DNN layer. In addition, it happens not frequently, for instance, only 6 out of 139 layers of GoogLeNet require the data handling. Note that an extreme case may require a high cost for data handling, e.g., up to 73ms in our measurement. However, this is not critical since LaLaRAND scheduler will allocate resources while avoiding such a large cost.

Table I shows the memory overhead; LaLaRAND requires up to 27% more memory (up to only 6.8 MB) than a vanilla PyTorch, which is affordable for better real-time schedulability.

TABLE I. MEMORY OVERHEAD PROFILE ON NVIDIA XAVIER

Memory (MB)	GoogLeNet	MobileNetV2	MnasNet	SqueezeNet
PyTorch	26.5	13.97	17.48	4.95
LaLaRAND	33.38	17.61	22	6.29

VII. RELATED WORK

As modern embedded systems are equipped with CPUs along with heterogeneous processors such as GPUs, a rich number of prior studies have been focused on splitting DNN computations across heterogeneous processors to improve the performance of DNN inference. DeepX [18] splits a DNN into multiple groups of layers, applies some optimizations (e.g., pruning) to the groups, and then distributes the groups to the heterogeneous processors. Heimdall [22] partitions each DNN into smaller units, schedules multiple DNNs on GPU, and offloads some DNNs to CPU in case there is a high contention on GPU. μLayer [19] accelerates a single DNN layer by simultaneously utilizing heterogeneous processors and performing computations using processor-friendly quantization. Some other studies focused on offloading DNN computations to the cloud. MCDNN [20] executes a DNN either on the mobile device or in the cloud depending on the remaining energy and cash budget in the cloud. Neurosurgeon [21] executes earlier layers on the mobile device and the following layers in the cloud to reduce the latency and the energy consumption. Although all of these studies have made valuable contributions on lowering inference latency by utilizing multiple resources, they did not deal with the timing constraints when allocating resources or scheduling DNN tasks, rendering them infeasible for real-time multi-DNN systems.

A recent study [6], most relevant to our work, focused on ensuring the timing constraints of DNN tasks. The work in [6] employs a pipeline-based resource allocation and scheduling architecture, called DART, that partitions the DNN inference execution of each task into stages, configures a set of computing nodes, and allocates computing nodes to stages so as to meet timing constraints. Although DART provides deterministic response time to DNN tasks, the performance imbalance issue pertaining to heterogeneous resources and subsequent related issues are not explicitly considered, which significantly affects schedulability performance.

VIII. CONCLUSION

This paper aims to improve the schedulability of real-time DNN tasks while leveraging heterogeneous resources. We presented LaLaRAND, a new system abstraction, that enables transparent and flexible CPU/GPU scheduling of individual DNN layers. By tightly coupling CPU-friendly quantization with layer-level CPU/GPU allocation schemes, LaLaRAND can not only provide timing guarantees for real-time DNN tasks but also significantly improve schedulability performance while mitigating the overall accuracy loss due to CPU-friendly quantization. Our implementation and evaluation of LaLaRAND on top of a state-of-the-art ML framework demonstrated the effectiveness of LaLaRAND in meeting timing constraints without any significant accuracy drop for real-time DNN tasks. In future, we would like to extend LaLaRAND towards multiple GPUs with concurrent kernel executions and develop efficient task-level or layer-level priority assignment in order to further enhance schedulability as well as resource-efficiency.

ACKNOWLEDGEMENT

This work was supported in part by the National Research Foundation of Korea (NRF) grant (2017M3A9G8084463, 2020R1F1A1076058, 2021R1A2B5B02001758, 2021R1A4A1032252, 2018R1A5A1059921 (ERC), 2020R1A2C2005479) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant (2014-3-00065: Resilient Cyber-Physical Systems Research, 2020-0-00209) funded by the Korea government (MSIT), as well as the DGIST R&D Program of MSIT (20-CoE-IT-01).

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 *RTSS*, 2020.

[2] A. Singh, D. Patil, and S. Omkar, "Eye in the sky: Real-time drone surveillance system (dss) for violent individuals identification using scatternet hybrid deep learning network," in *CVPR Workshops*, 2018.

[3] X. Song, B. Yang, G. Yang, R. Chen, E. Forno, W. Chen, and W. Gao, "Spirosonic: Monitoring human lung function via acoustic sensing on commodity smartphones," in *MobiCom*, 2020.

[4] 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 *RTAS*, 2019.

[5] S. Bateni and C. Liu, "ApNet: Approximation-aware real-time neural network," in *RTSS*, 2018.

[6] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *RTSS*, 2019.

[7] S. Bateni, H. Zhou, Y. Zhu, and C. Liu, "PredJoule: A timing-predictable energy optimization framework for deep neural networks," in *RTSS*, 2018.

[8] S. Lee and S. Nirjon, "SubFlow: A dynamic induced-subgraph strategy toward real-time DNN inference and training," in *RTAS*, 2020.

[9] S. Heo, S. Cho, Y. Kim, and H. Kim, "Real-time object detection system with multi-path neural networks," in *RTAS*, 2020.

[10] H. Zhou, S. Bateni, and C. Liu, "S^3DNN: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads," in *RTAS*, 2018.

[11] 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 *RTSS*, 2020.

[12] R. Pujol, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Generating and Exploiting Deep Learning Variants to Increase Heterogeneous Resource Utilization in the NVIDIA Xavier," in *ECRTS*, 2019.

[13] D. Zhang, N. Vance, Y. Zhang, M. T. Rashid, and D. Wang, "Edge-batch: Towards ai-empowered optimal task batching in intelligent edge systems," in *RTSS*, 2019.

[14] W. Kang and J. Chung, "DeepRT: predictable deep learning inference for cyber-physical systems," *Real-Time Systems*, vol. 55, pp. 106–135, 2019.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019. [Online]. Available: https://pytorch.org

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *MM*, 2014. [Online]. Available: https://caffe.berkeleyvision.org

[18] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *IPSN*, 2016.

[19] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "µlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *EuroSys*, 2019.

[20] H. Shen, M. Philipose, S. Agarwal, and A. Wolman, "MCDNN: An approximation-based execution framework for deep stream processing under resource constraints," in *MobiSys*, 2016.

[21] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *ASPLOS*, 2017.

[22] J. Yi and Y. Lee, "Heimdall: Mobile gpu coordination platform for augmented reality applications," in *MobiCom*, 2020.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[24] 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," in *ICLR*, 2017.

[25] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019.

[26] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.

[27] Jetson AGX Xavier Developer Kit. [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[28] Jetson TX2 Module. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2

[29] Jetson Nano Developer Kit. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[30] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018.

[31] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *CVPR*, 2019.

[32] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks," in *NIPS*, 2016.

[33] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *CVPR*, 2018.

[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

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

[36] M. Dukhan, Y. Wu, and H. Lu, "Qnnpack: Open source library for optimized mobile deep learning," Mar 2020. [Online]. Available: https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/

[37] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[38] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *RTSS*, 1989.