CAMDNN: Content-Aware Mapping of a Network of Deep Neural Networks on Edge MPSoCs

Soroush Heidari, Mehdi Ghasemi, Young Geun Kim, Carole-Jean Wu, and Sarma Vrudhula

Abstract—Machine Learning (ML) workloads are increasingly deployed at the edge. Enabling efficient inference execution while considering model and system heterogeneity remains challenging, especially for ML tasks built with a network of deep neural networks (DNNs). The challenge is to maximize the utilization of all available resources on the multiprocessor system on a chip (MPSoC) at the same time. This becomes even more complicated because the optimal mapping for the network of DNNs can vary with input batch sizes and scene complexity. In this paper, a holistic hierarchical scheduling framework is presented to optimize the execution time for a network of DNN models on an edge MPSoC at runtime, considering varying input characteristics. The framework consists of a local and a global scheduler. The local scheduler maps individual DNNs in the inference pipeline to the best-performing hardware unit while the global scheduler customizes an Integer Linear Programming (ILP) solution to instantiate DNN remapping. To minimize scheduler runtime overhead, an imitation learning (IL) based scheduler is used that approximates the ILP solutions. The proposed scheduling framework (CAMDNN) was implemented on a Qualcomm Robotic RB5 platform. CAMDNN resulted in lower execution time of up to 32% than heterogeneous earliest finish time, and by factors of 6.67X, 5.6X and 2.17X than the CPU-only, GPU-only and Central Queue schedulers.

Index Terms—Machine learning, scheduling, edge, IoT, deep neural networks, DNN serving

1 Introduction

Deep Neural Networks (DNNs) are increasingly used in a variety of applications such as computer vision, speech recognition, and recommender systems [4], [14], [15], [16], [31]. The increasing complexity of machine learning (ML) tasks drives the adoption of an end-to-end inference pipeline using multiple models, which are best represented as a network of DNNs [6], [25], [36], [37]. An example of a network of DNNs is shown in Fig. 1. A video stream is first subject to object detection after which custom DNNs are invoked to extract extra information specific to that object type.

A conventional approach to process such networks of DNNs is to use a cloud server where the user-end device sends only the input data to the cloud and receives the processed data [9], [27], [42]. Their primary disadvantages are large communication latency and diminished security. With the advances in the MPSoC design, there has been an increasing trend to run ML workloads entirely at the edge [11], [21], [24], [41]. However, efficient mapping of a network of DNN models onto an MPSoC poses several challenges. First, it has to be performed on a frame-by-frame basis as each frame can present different numbers of objects and different numbers of each object type, reflecting changes in the scene complexity. Fig. 2 shows an example scene where the *complexity* of the scene is reflected in both *the number of objects* as well as *the number of object types*. Second, modern MPSoCs are heterogeneous multi-core processors that consist of CPUs, graphic processing units (GPUs), digital signal processors (DSPs) and possibly custom accelerators, with different performance characteristics, presenting various acceleration opportunities. Third, the practice of batching input requests of a model to exploit data locality is made more complicated due to the heterogeneity of the MPSoC and the dynamically changing scene. Fourth, the runtime scheduling can be considered as a sequential decision-making problem where each scheduling decision depends on both current scene complexity and the current state of resources. This makes static scheduling solutions suboptimal.

An optimal (or near-optimal) workload-specific scheduling policy that uses the information mentioned above can be found using constrained optimization techniques such as mixed integer programming [3], [5] and constrained programming [34]. However, these approaches are not practical in a dynamic environment where scene complexity can change on a frame-by-frame basis and the execution overhead of these policies becomes prohibitive at runtime. To resolve these issues, imitation learning (IL) techniques [35] can be utilized to approximate the optimal policy (a.k.a. Oracle) with minimum runtime overhead.

In this paper a new approach, called *CAMDNN*, is proposed for mapping a network of DNNs onto a heterogeneous MPSoC. Its objective is to minimize the processing delay, thereby reducing the number of dropped frames. *CAMDNN* is a hierarchical scheduling framework including two main components: a *local* scheduler which maps individual DNNs in the inference queue to the best execution

- *Soroush Heidari, Mehdi Ghasemi, Carole-Jean Wu, and Sarma Vrudhula are with the School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ 85281 USA. E-mail: {sheidar1, mghasem1, caroleje, vrudhula}@asu.edu.*
- *Young Geun Kim is with the Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea. E-mail: younggeu_kim@korea.ac.kr.*

Manuscript received 18 January 2022; revised 1 August 2022; accepted 28 August 2022. Date of publication 15 September 2022; date of current version 11 November 2022.
This work was supported in part by NSF under Grants 2008244, CCF-1652132 and CCF-1618039, and in part by the Center for Embedded Systems, NSF under Grant 1361926.
(Corresponding author: Soroush Heidari.)
Digital Object Identifier no. 10.1109/TC.2022.3207137

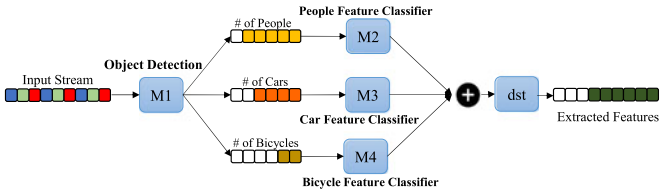

Fig. 1. The network of DNN models in a surveillance application. After identifying objects, specific DNN models are used for further processing of car and person and bicycles.

target in a greedy fashion and a *global* scheduler that is invoked infrequently to adjust the existing allocation. The global scheduler initiates DNN remapping on the basis of the overall reduction in latency when compared to simply running the local scheduler with current allocation. The global scheduler customizes an Integer Linear Programming solution (ILP) incorporating dependency structure of the network of DNNs, scene complexity, DNN-specific features and hardware heterogeneity. To minimize scheduler runtime overhead, an imitation learning (IL) based scheduler is used that approximates the ILP solutions at runtime. The IL-based scheduler imitates the optimal scheduler devised and implemented at design time. To construct the IL-based scheduler, a set of optimal scheduling demonstrations are captured along with relevant system states. Different scene complexities are simulated by enumerating the number of objects in the scene.

The proposed approach is demonstrated on a real platform. The mobile MPSoC is a Qualcomm Robotic RB5 which is equipped with CPU, GPU, and DSP units. Experiments demonstrate *CAMDNN* improves the execution time by up to 32% compared to a baseline scheduler. The baseline scheduler maps individual DNN to a unit on the MPSoC based on the unit's time of availability and the predetermined execution time of the DNN on that unit. *CAMDNN* also leads to 6.67x, 5.60x, and 2.17x improvement compared with other schedulers including *CPU-only*, *GPU-only* and *Central Queue*.

2 Background and Motivation

In this section, principal characteristics of DNNs that must be included in the optimization problem are described. These include different performance profiles of models on hardware units, their loading time, and the impact of batch size on performance. *Optimal scheduling decisions depend on the scene complexity of inputs to the network of DNNs, the design and implementation of specific synchronization mechanism for input batching and previous model mapping decisions on hardware units.*

Fig. 2. An example of input frame [2]. The objects of interest are cars, people, and bikes. The other detected objects are ignored.

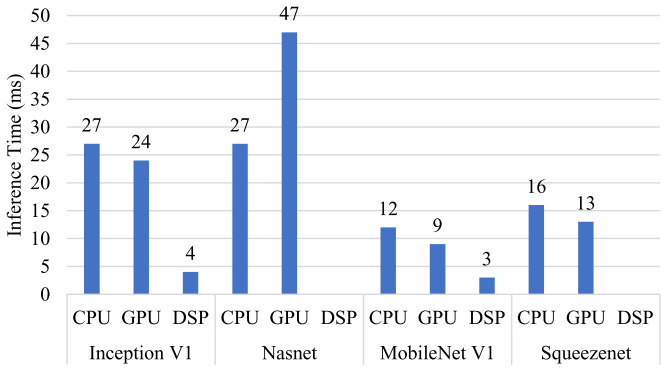

Fig. 3. The inference time of different models on CPU, GPU, and DSP. Nasnet and SqueezeNet were not supported on DSP.

DNNs and Hardware Characteristics. DNN kernel execution time is determined by neural network architectures and specific hardware. Fig. 3 shows that the different models have different performance profiles on different hardware units of Qualcomm RB5 development kit. In addition, not all models are available on every processor. Optimal mapping has to account for such differences.

*Kernel Loading. Model loading time is significant and varies across DNNs and hardware execution targets.*Even though different hardware accelerators show lower execution times for different models, they are usually associated with a significant overhead of loading time for each model. This limits the migration of DNN models between different processors at runtime. Although the loading cost can be avoided by initialization of all DNN models on each hardware resource before runtime, this is not practical due to limited resources on mobile MPSoCs. Fig. 4 shows that loading time depends on the model size and hardware type.

*Input Content Characteristics. Performance can be improved by as much as 2 times if the model mapping decision considers input contents explicitly.*In a scene, multiple objects of the same type (e.g., people) will invoke multiple instances of the same DNN. Therefore, as the scene changes, both the total number of objects and the number of each object type change. Not every model is available on each processor. Thus, all three of these factors need to be accounted for in determining the optimal mapping, from scene to scene.

*Inference Batching. Additional improvement in performance can be achieved with an optimal setting of batch size.*To make the most efficient use of different hardware resources, several inference requests must be executed in batches. Besides improving the execution time per inference, batching can decrease the overhead associated with context switching of different DNN models on the same hardware resource. Therefore, it is better to avoid switching between different DNN models and run different inference requests from the same DNN in a batch.

The choice of optimal batch size (i.e., the batch size that minimizes the execution time per inference for a specific model on a specific hardware type) depends on the hardware type, model's size and architecture. Fig. 5 shows the average inference time of four representative image classification

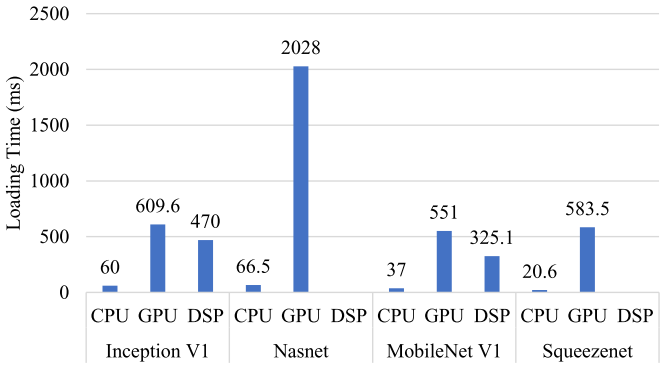

Fig. 4. The loading time of different models on CPU, GPU, and DSP. Nasnet and SqueezeNet were not supported on DSP.

DNN models with varying batch size. For example, the optimal batch size for running Inception V1 on CPU is three. Note that although performance versus batch size are monotonic for some models, the best *joint* batch size for a scene may not correspond to the minimum point of each curve.

Additional improvement in performance related to batching may be realized by waiting for a sufficient number of images to be prepared prior to dispatching them as inputs of the DNN models. Delaying the scheduling, typically by a small amount is beneficial due to the asynchronous arrival patterns of the inference requests. This happens because after object detection, the images are prepared in parallel by multiple CPU cores [8]. To avoid the increase in latency due to batching inference requests from multiple frames, our approach is limited to batching inference requests that originated from the same frame and model.

Fig. 6 shows the result of an optimal mapping for an example scene with 13 people, 11 cars and 2 bikes. The mapping algorithm should select the best execution target and also the combination of batch sizes. In this example, the optimal batch size combination on the DSP for people is (0,0,1,0,2), which denotes one batch of three requests and two batches of five requests. Similarly, three batches of cars each of size one are processed by the CPU and two batches of cars each of size four are processed by the GPU. Both bicycles are processed by the DSP as a batch of size 2. The optimal choice of batch size is dependent on the profiling shown in Fig. 5.

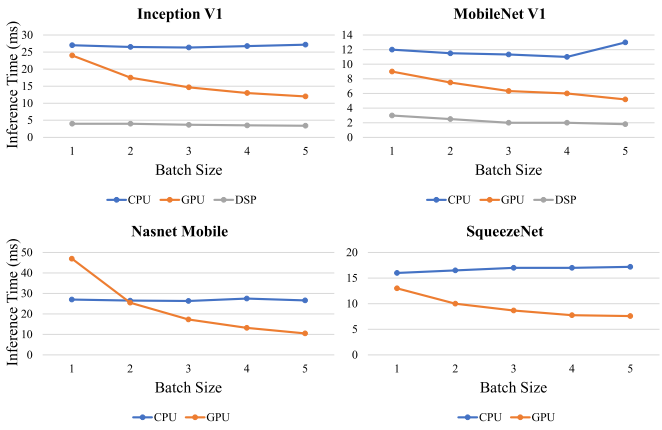

Fig. 5. The effect of batch size on the inference time per request is dependent on the model type and the hardware unit.

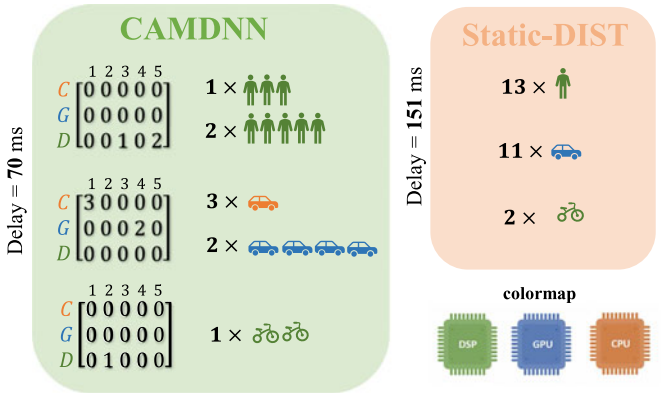

Fig. 6. The left figure shows optimal mapping of models to processors and the best combination of batch size is shown in matrices for each object type. The rows in the matrices correspond to computing resources and the columns correspond to different batch sizes. CAMDNN can process the given frame in Fig. 2 more than 2x faster compared to optimal static scheduler which ignores scene complexity and batching opportunities.

In contrast, *Static-DIST* scheduler assigns the network of DNNs to different processors considering only the dependency structure and heterogeneous execution time. However, scene complexity and batching are not considered. Therefore, all 13 people and two bicycles are being processed by DSP (one-by-one), all 11 cars being processed by GPU. That results in more than 2x higher processing time compared to the proposed approach.

3 Related Work

There has been an increasing interest in running DNNs on mobile devices. Several platforms such as TensorFlow Lite [1] and Qualcomm SNPE [30] are available.

In [7], the capabilities of mobile devices for running deep learning workloads are explored and an extension to the TensorFlow framework to support training is proposed. In [18], [19], the performance analysis of running DNNs on several mobile devices (chipsets from Qualcomm, HiSilicon, Samsung, MediaTek, and Unisoc) is presented. There has also been prior work on mapping a single DNN onto hardware units on a mobile device. A framework called *Pipe-it* is described in [39] where the convolution layers in a single DNN are partitioned between big and LITTLE cores on a multi-core mobile device.

Reinforcement learning (RL) and Imitation learning (IL) have witnessed growing popularity for sequential decision making problems such as runtime task scheduling. However, lack of simulation environment makes RL-based solutions not practical for runtime scheduling on resource-constrained devices due to slow convergence [26], [27]. In contrast, the use of imitation learning can be a better approach where the optimal solution can be formulated and generated offline. In [23], an imitation learning based approach for task scheduling in heterogeneous many-core processors is proposed which aims at wireless streaming application with static application graph. However, the proposed runtime scheduler distributes the workload while ignoring data locality and hardware compatibility/memory constraints.

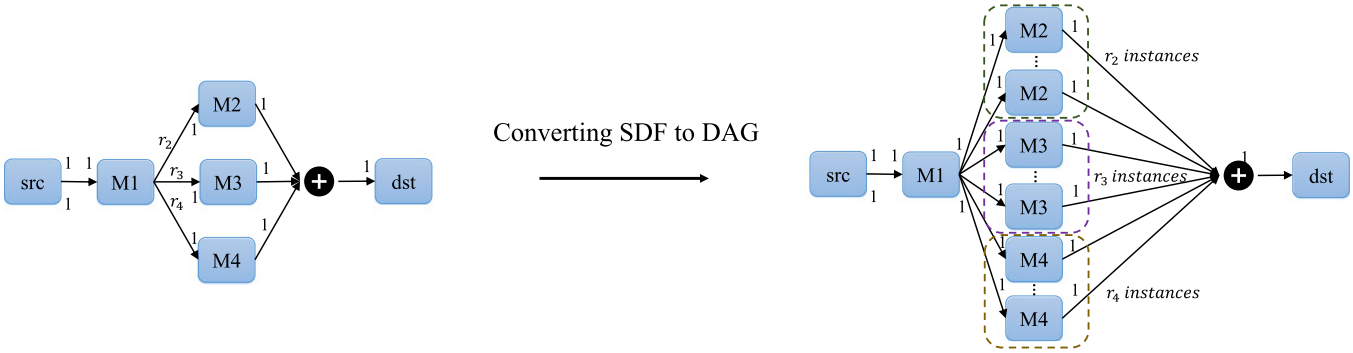

Fig. 7. The data flow graph of the application. In order to improve the processing delay, duplicates of the models are generated depending on how many objects are detected after object detection model (M1).

Another category of works partition the computation of a single or multiple DNNs between devices in an IoT environment. *MoDNN* [28] and *DeepThings* [44] were designed to execute DNNs on resource-constrained mobile devices. They partition the input data of a DNN model and offload the computation of a segment of the input to a device. In [13], the computation of a network of DNNs is partitioned between a user-end device and a cloudlet. The goal is to reduce the energy consumption of the user-end device by the optimal collaboration between the user-end device and the cloudlet. However, this work is not aware of the variation in the content of input data.

AutoScale [21] targets energy-efficiency of DNN inference. It employs reinforcement learning to select the suitable computing unit with the options of CPU, co-processors on the mobile device, and a connected edge device in the network. *MABSTA* [20] distributes the stages of a computation flow graph to different devices on the network to minimize the total processing delay. The method is also based on a reinforcement learning to select the suitable execution target considering the heterogeneity of devices and the variable network delays.

Efforts to improve the execution of ML workloads are now focusing on running several DNNs in the form of network of DNNs. Google's MediaPipe framework [25] and visual question answering (VQA) [6], [36] are recent examples. The frameworks presented in [9], [10], [32], [33] are cloud-based DNN serving solutions which either do not consider or make general assumption in regards to scene complexity, reallocation overhead, hardware heterogeneity and application graph dependency. For instance, Inferline [9] tries to meet end-to-end tail latency constraints of prediction pipelines while minimizing the overall hardware cost. Inferline schedules different stages of ML pipeline with multiple DNNs onto hardware resources on the cloud server. A low-frequency planner selects the hardware type for each stage of pipeline and a high frequency planner reacts to the changes in the input arrival patterns. However, this work does not address the problems of workload mapping on resource-constrained edge MPSoCs.

To the best of the authors' knowledge, this paper is the first work that addresses the mapping of a network of DNNs on an edge MPSoC that is content-aware and exploits the DNN properties such as batching, and model load time. In addition, the proposed method is evaluated on a state-of-the-art MPSoC.

4 System Model and Optimization Objective

4.1 Hardware Model

The mobile MPSoC used in this work had several processing units, namely, CPU, GPU and DSP. Each processing unit is denoted by p_j.

4.2 Application Model

The application consists of running several DNN models for each input frame where each model M_i is designed specifically for a recognition task. For example, as it is shown in Fig. 1, distinct models are used for processing different types of objects. The application can be modeled as a data flow graph $G = (\mathcal{M}, E)$ where $\mathcal{M}$ is the set of DNNs and E denotes the precedence between different models. Depending on the content of input data, the number of times that the models are executed (r_i) changes. Therefore, modeling the application graph as a directed acyclic graph (DAG) is not correct as it ignores the dynamic nature of the workload and model's repetition in a single frame. Typically, the real-time streaming applications that perform a set of periodic tasks with fixed data production and consumption rates are modeled as a synchronous dataflow graph (SDF). For example, the production rate for an object detection model is the number of objects found in the scene and consumption rate is the input batch size which represents the number of frames that can be processed at each model's invocation. The consumption and production rates are not known at design-time and rates are solely dependent on each frame's content and the scheduling decisions in our application.

Assuming an acyclic application graph, the acyclic SDF can be transformed to DAG in polynomial time where each model is duplicated based on its repetition number. For example, in the SDF graph on the left in Fig. 7, parameters r_2, r_3 and r_4 represent the number of objects for three types in the scene. These parameters are known only after running the object detection model.

Requests for the same model are batched together and sent to a computing resource. The execution time of model i on processing unit j with batch size k is denoted with $t_e(i, j, k)$. The list of notations can be seen in Table 1.

4.3 Optimization Objective

The goal is to map different DNNs to processing units and determine the batching size for models (considering the

TABLE 1
List of Notations

Notation	Description
M_i	model i
$\mathcal{M}$	set of all the models in application graph $\{M_i \mid 1 \leq i \leq n_m\}$
src, dst	source and destination nodes
r_i	total number of invocations for model i
ρ_i	priority of model i
p_j	processor j
$\mathcal{P}$	set of all the available processing units $\{p_j \mid 1 \leq j \leq n_p\}$
N_j	maximum number of distinct running models on processor j
rem_j	remaining time that takes for processor j to become idle
T_j	total execution time on processor j
T_{max}	maximum execution time on all processors (makespan)
k	the batch size $(1 \leq k \leq n_b)$
$b_k(i,j)$	the number of instances of M_i running on p_j with batch k
$a(i,j)$	set to 1, if $\sum_k b_k(i,j) \neq 0$ for a given i, j
$t_e(i,j,k)$	execution time of model i on p_j with batch size of k
$t_l(i,j)$	loading time of model i on processor j
$t_s(i,j)$	start time of model i on processor j
$t_f(i,j)$	finish time of model i on processor j
$epoch$	scheduling interval

change in the content of input data) such that the frame processing time (T_{max}) is minimized.

5 Proposed Approach

Fig. 8 shows the overall scheme of the proposed approach *CAMDNN*. It shows the global and local schedulers and their interaction. The time horizon is divided into an infinite number of time intervals of fixed duration equal to $epoch$. Aside from initial mapping, the global scheduler operates every $epoch$ and re-allocates DNN models (considering all the DNNs in the flow graph) whenever there are dramatic changes in the scene complexity or hardware availability (rem_j). As it can be seen, the local scheduler is the final decision maker in the scheduling pipeline and the global scheduler is invoked infrequently in a fixed scheduling interval to re-calibrate the existing allocation. In between two such invocations, the local scheduler tweaks that allocation by making local changes without knowledge of the whole graph.

Even though the global scheduler produces optimal results, it is impossible to imitate the optimal scheduler at runtime without any regression error. Therefore, the local scheduler is proposed to act on each DNN inference requests in a greedy fashion without any information about inference requests of other models. Its decision is made based on the current system state (hardware availability rem_j) and collected profiling information $t_e(i,j,k)$ and $t_l(i,j)$. The local scheduler also utilizes the new allocation and number of duplicates of each DNN from the global scheduler (explained more in Section 5.2).

5.1 Global Scheduler

The goal of the global scheduler is to obtain the optimal mapping and scheduling of DNNs to computing resources with the knowledge of all the DNNs inside a frame. The problem can be formulated precisely as an Integer Linear Programming (ILP) problem. However, the execution overhead of the ILP solver becomes prohibitive at runtime. This challenge is addressed by using imitation learning methods. Learning optimal scheduler by imitation can give us an scheduler with minimal runtime overhead compared to well-known heuristics. The design of both ILP-based optimal scheduler and the IL-based scheduler are explained as follows.

5.1.1 ILP-Based Optimal Scheduler

The ILP formulation includes the DNN-specific profiling information, the number of objects, the number of object types and previous allocation in the frame. The ILP formulation can be used for general heterogeneous multiprocessor task assignments with task/data dependency. However, embedded DNN-specific characteristics in the formulation result in higher performance improvement for DNN serving. The DNN-specific profiling information includes execution time $t_e(i,j,k)$ of model i on processor j with batch size k. The ILP formulation is given below. The *decision variables* are the start and finish time ($\boldsymbol{t_s(i,j)}$, $\boldsymbol{t_f(i,j)}$) of model i on processor j and $\boldsymbol{b_k(i,j)}$ (shown bold in the ILP formulation). The number of decision variables in the ILP solution depends on the application graph and the given target device. Therefore, we have $((n_m \times n_p) \times (2 + n_b))$ decision variables. The auxiliary variable T_{max} is the upper bound on the total delay in processing a given frame. Minimizing T_{max} minimizes the finish time of a given frame among all processors.

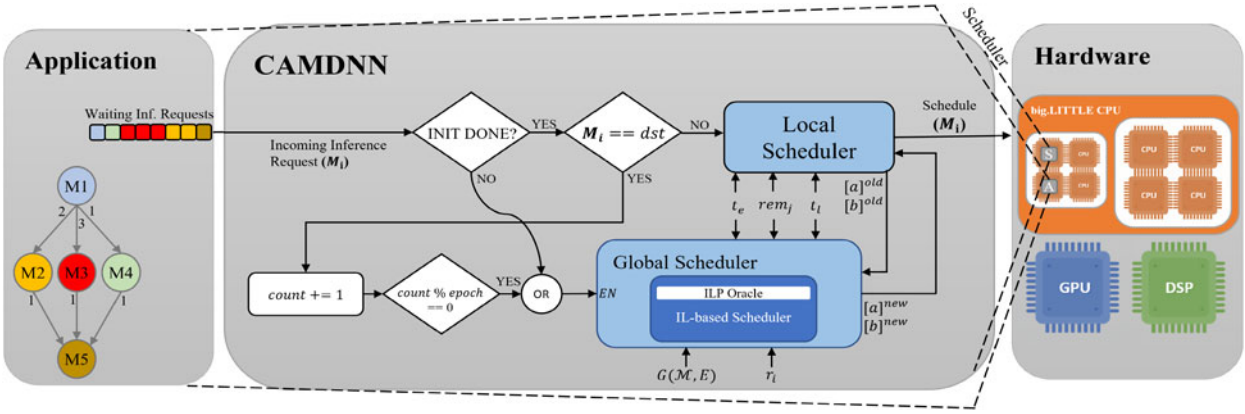

Fig. 8. The overall scheme of our proposed approach.

$min(T_{max})$,
// Finish time on every processor j is less than T_{max}
$T_j + rem_j \leq T_{max} \quad \forall j \quad (c1)$
// Finish time of model i on processor j is equal to its start time, execution time based on the mapped batching, and the loading time if a model is needed to be loaded.

$$e(i,j) = \sum_{k=1}^{n_b} \boldsymbol{b_k(i,j)} \times t_e(i,j,k)$$

$$\boldsymbol{t_f(i,j)} = \boldsymbol{t_s(i,j)} + e(i,j) + \frac{diff(i,j) \times t_l(i,j)}{epoch} \quad (c2)$$

$$diff(i,j) = ([a(i,j)]^{new}.\overline{[a(i,j)]^{old}})$$

// Precedence constraint in the flow graph
$i_1 \prec i_2 : \boldsymbol{t_s(i_2,j)} \geq \boldsymbol{t_f(i_1,j)} \quad \forall j \quad (c3)$
// Priority constraints based on the critical path
$\rho_{i_1} > \rho_{i_2}$:
$\boldsymbol{t_s(i_2,j)} \geq \boldsymbol{t_s(i_1,j)} + e(i_1,j) \quad (c4)$
// Sum of batches for each model over all processors equals to the total number of objects for that model

$$\sum_{j=1}^{n_p} \sum_{k=1}^{n_b} k \times \boldsymbol{b_k(i,j)} = r_i \quad (c5)$$

// Setting allocation variables based on batch variables
Ω is a big integer number (e.g., maximum execution time)

$$\sum_{k=1}^{n_b} \boldsymbol{b_k(i,j)} \geq (1 - \Omega \times (1 - a(i,j)) \quad \forall i,j \quad (c6)$$

$$\sum_{k=1}^{n_b} \boldsymbol{b_k(i,j)} \leq \Omega \times a(i,j) \quad \forall i,j \quad (c7)$$

// Processor constraints for number of mapped model

$$\sum_{i=1}^{n_m} a(i,j) \leq N_j \quad \forall j \quad (c8)$$

//$b_k(i,j)$ is an integer variable

$$\boldsymbol{b_k(i,j)} \in \mathbb{N}^+ \quad (c9)$$

// Start and finish times are positive continuous variables

$$\boldsymbol{t_f(i,j)}, \boldsymbol{t_s(i,j)} \in \mathbb{R}^+ \quad (c10)$$

There are three distinct elements considered in the above ILP formulation.

- Variations across scenes result in variations in the workloads associated with each model. Consequently, a fixed assignment of models to processing units will be suboptimal, and often, significantly so. The ILP formulation will result in the best distribution of the models, including multiple instances of the same model, across the processors.
- The ILP solver will find the optimal combination of batch sizes of each model on different processors. This can have significant impact on the latency.
- Embedding the reallocation overhead in the ILP formulation will result in the optimal allocation which minimizes the average processing delay over the next scheduling epoch.
- The ILP formulation will reduce the overhead of context switching between different models through the assignment of priorities to models. For example, all instances of a given model are assigned a unique priority, resulting in each being scheduled one after another, and thereby reducing the overhead of context switching. The priority is derived based on the critical path at design time. The unique priority of each DNN model is designed to be the summation of its upward rank (maximum distance from model i to terminal node) and downward rank (maximum distance from start node to model i) [38], [43].

5.1.2 IL-Based Scheduler

IL-based scheduler aims to learn a scheduling policy $\hat{\pi}$ that approximates the performance of the optimal scheduler π^* when either the optimal scheduler is not available or it is computationally expensive to run it at runtime. The first step is to produce a set of scheduling demonstration P^* induced by the optimal scheduler (ILP-based scheduler) π^*. Given this pre-collected set of demonstration, the IL-based scheduler ($\hat{\pi}_\theta$) parameterized by θ learns to take the correct scheduling decision $\pi^*(s) = a^*$ for a given input state s minimizing the imitation loss L. The imitation learning problem can be formulated as follows:

$$\theta^* = argmin_\theta E_{(s,a^*)\sim P^*} L[a^*, \hat{\pi}_\theta(s)].$$

We have used off-the-shelf machine learning method of deep neural networks to construct the IL-based scheduler. The imitation learning problem is modeled as supervised multi-output regression problem which maps the input feature vector to the batching matrix $b_k(i,j)$. The output matrix encompasses both allocation and number of instances of each DNN on different processors. To solve the regression problem, we construct an MLP (Multi Layer Perceptron) neural network with weight parameter θ.

State Representation. The ability to learn the scheduling policy greatly depends on state and action representations and the choice of imitation loss. One of the key challenges is designing a node embedding which preserves and aggregates all DNN features and their dependency structure in the network of DNNs. In our scheduling problem, DNN features that are used in the ILP formulation are execution time t_e on different heterogeneous processing elements, priority, and total number of invocations. The priority assignment encompasses the information about the dependency structure. Also, the number of invocations encodes scene complexity which is the dynamic component of the input feature vector changing at runtime. Besides DNN features, remaining time that takes for processor j to become idle rem_j is considered to represent current hardware state.

*Loss Function.*The next challenge is designing a loss function which minimizes the regression error which represents the Mean of Absolute Errors (MAE) over the entire training set. As it was shown in Fig. 6, the batching matrix which is the expected output of the proposed regression model is inflated with zero values. This is what is referred to as zero-inflated multi-target regression problem [22]. The meta-learning [40] is one of common approaches to tackle zero-inflated regression. The basic idea is to train a

multi-stage network with a combination of a Classifier and Regressor. The multi-label classifier decides if the output is zero. After classifier, a multi-target regression model is used to predict the non-zero values. Also, we have imposed the data dependent constraint ($c5$) introduced in Section 5.1.1 by adding extra penalty term [29] to the the loss function.

5.2 Local Scheduler

As mentioned earlier, it is impossible to replicate the optimal scheduler decisions at runtime without any regression error. In case of *zero loading time*, running global scheduler for every new frame would be sufficient to fix prior incorrect scheduling decisions due to regression error. However, DNN loading time is significantly high and remapping DNNs after every change in scene complexity and hardware availability is impossible. Therefore, the local scheduler is devised to cope with incorrect scheduling decisions from the IL-based scheduler.

Algorithm 1. Local Scheduler

Input: Inference request from model M_i, ρ_i, $[b]$
Output: Update $b_k(i,j), 1 \leq j \leq n_p, 1 \leq k \leq n_b$
Check w_i (The number of waiting requests for M_i)
if $w_i \neq b_k(i,j)$ **then**
  Start SyncTimer
end
while $w_i \neq b_k(i,j) || SyncTimer < WaitTime$ **do**
  wait()
end
Find the host based on preferred batch size $k \leq w_i$ for M_i
$HOST = argmin_j\{(rem_j + t_e(i,j,k)/k)$
$+ t_l(i,j) \times (1 - a(i,j))\}$
Wait until selected processor becomes ready, then batch all available requests and update matrix $[b]$
Dispatch the inference request

The local scheduler is a modified version of *Heterogeneous Earliest Finish Time (HEFT)* policy [38] which takes both loading time and inference synchronization into account. The runtime *HEFT* heuristic takes an inference request as input, the time to execute it on different hardware resources and the ready time of each hardware resource. The *Local Scheduler* is illustrated in Algorithm 1. Given matrix $t_e(i,j,k)$, a list of preferred batch sizes for a given model on each processor can be obtained by taking into account the average inference time per request shown in Fig. 5. For example, the preferred batch sizes for MobileNet V1 on CPU are $[4,3,2,1,5]$. The algorithm looks up the first batch size k in the sorted list that is less than or equal to the number of waiting requests for model i (w_i). To find the host, local scheduler considers both execution time of model i on processor j for a batch size of k and the loading time for model i on processor j. If the model is already on the hardware unit, the loading overhead is multiplied by $(1 - a(i,j))$ which makes it zero. The value of rem_j is the time it takes for processor j to become idle. Since both the execution time and the starting time of the model is known to the scheduler, the remaining time can be calculated based on the elapsed time of the current running model on each processor. The local scheduler implements inference synchronization by introducing fixed wait time ($\approx 5ms$) to let other inference requests from the same DNN model arrive. The local scheduler decisions match the sequence of schedules generated by IL-based scheduler unless either there is a change in scene complexity during current scheduling epoch or the IL-based scheduler approximates the optimal scheduler incorrectly (e.g., the output of IL-based scheduler does not meet data dependent constraints mentioned in Section 5.1.1). The time complexity of local scheduler is $O(1)$ which makes it quite responsive with minimal overhead.

5.3 CAMDNN Example

Fig. 1 shows an example application graph where Inception V1, SqueezeNet and MobileNet are used for processing people, cars, and bicycles. It is assumed that the camera captures 14 frames per second (fps) and the scheduling epoch is fixed to $5\ sec$. As it can be seen in Fig. 8, CAMDNN starts with an initialization step in which the initial allocation is assigned based on the optimal ILP-based solution. After the initial step, the local scheduler is the final decision maker in the scheduling pipeline and the global scheduler is only invoked infrequently in a fixed scheduling interval to recalibrate the existing allocation. Fig. 9 illustrates the CAMDNN steps corresponding to the scenario depicted in Fig. 2 after the object detection has completed. Below, the details of important steps in CAMDNN are explained:

1) Initialization step: In this step, because there is no prior frame to take into account, the the number of instances of each model is assumed to be one, i.e., $r = (1,1,1)$. In general, in any scheduling epoch, the number of instances of each model collected in the most recently processed frame is taken into account. In addition, the loading time of the initial frame is ignored.

   The initial allocation assigns a CPU to only serve inference requests for the object detection model and the other models are allocated to the GPU and DSP units. The GPU serves the inference requests for the car model and the DSP serves the inference requests for people and bicycle models. *Note:* The object detection can only be executed on the CPU and it is invoked just once for each frame.
2) Assume that the first frame ($f1$) in the scene shown in Fig. 2 has 13 people, 11 cars and 2 bicycles. The corresponding repetition vector is $r = (13,11,2)$. The local scheduler schedules the individual DNN requests in a greedy fashion (explained in Section 5.2) with respect to the initial allocation. In this frame, the people inference requests are processed in one batch of three requests and two batches of five requests on the DSP. The 11 cars are processed by the GPU in two batches of size five and one batch of size one. Both bicycles are processed by the DSP as a batch of size two. This results in a total processing delay of $97ms$ with a corresponding remaining time $rem_j = (0,25,0)$.
3) After processing the first frame, no model is assigned to the CPU. The loading time of car model on the CPU is 20.6 ms, while the GPU still has 25 ms of execution (remaining time $rem_{gpu} = 25$). Hence, the local scheduler adjusts the current allocation and loads car model on CPU. Now the car model is served by the GPU and CPU. This allocation results in a total

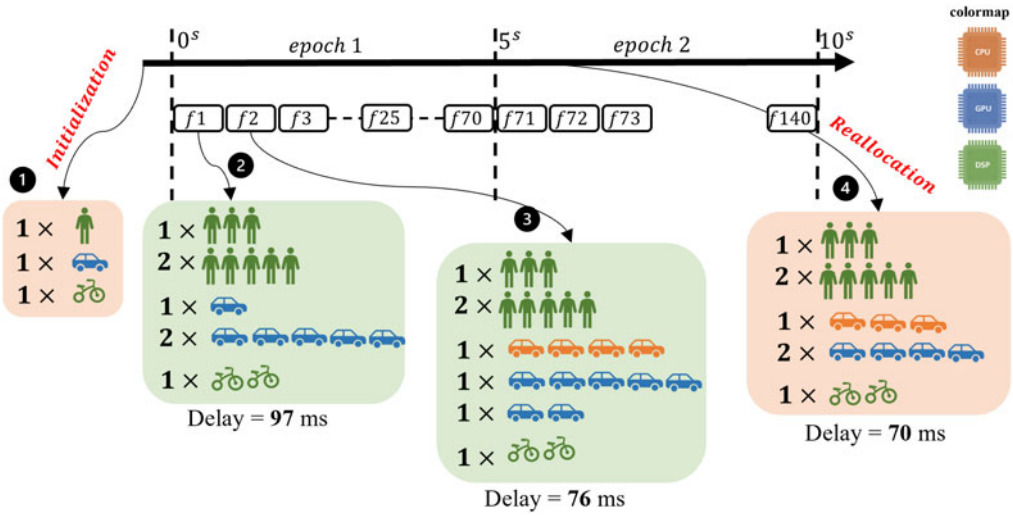

Fig. 9. CAMDNN runtime example.

processing delay of $76ms$ with the corresponding remaining time $rem_j = (5, 0, 0)$.

Note that although the local scheduler considers the accumulated remaining time on each processor and its loading time, it acts in a greedy fashion by allocating one inference request at a time. The resource allocated may not be the optimal choice in the long run as it does not consider other model inference requests.

4) After reaching the next scheduling epoch (time 5), the ILP-based solution is called to re-allocate the current allocation based on the most recently processed frame ($f70$). This new allocation results in the total processing delay of $70ms$ with corresponding remaining time $rem_j = (0, 0, 0)$. Had the ILP not been invoked at time $t = 5$ and the existing allocation remained intact, then the remaining time increases to $rem_j = (340, 335, 321)$. The remaining time on CPU delays the execution of object detection for the incoming frames and consequently delays the execution of other models assigned to the GPU and DSP. This causes significant remaining time on all processors. This shows the importance of reallocating the models after sufficient time has elapsed due to possible changes in the scene complexity.

6 Experimental Methodology

6.1 Hardware

Qualcomm Snapdragon RB5 development kit was selected to demonstrate the efficiency of our proposed approach. As shown in Table 2, this device is equipped with CPU, GPU, and DSP.

6.2 Application

The applications consist of an object detection model and several classification models (Fig. 1). The object detection model is based on a MobileNet-V1 SSD (single-shot detector) model (model M1). The classification models include Inception V1, Inception V2, MobileNet V1, Nasnet Mobile, SqueezeNet, and MobileNet V2. Different models with different precisions including INT8, FP16, and FP32 were tested. The pool of DNNs were simply selected as good representatives of small/medium/large DNN workloads from the Google tflite model hub with different precision support. These models were executed on the MPSoC using TFLite framework. The CPU, GPU, and DSP implementation were done using *xnnpack*, *OpenCL*, and *Hexagon* delegates. Also, the global scheduler is compiled and implemented on ARM processor using Cbc (Coin-or branch and cut) which is an open-source mixed integer programming solver written in C++ [12]. The IL-based solution is a multi-layer perceptron which is trained over 12,000 scheduling demonstrations using Tensorflow framework. The training set is obtained based on the enumeration of a reasonable range of scene complexity (e.g., from 0 to 14 objects for each object type) and a random set of remaining time for a given application graph and target device. The extra latency caused by errors in the neural network approximation of the ILP is about 3.5% compared to the optimal ILP solution tested with 100 random scene complexities. To run the trained model on the mobile device, the model is converted to TFLite model [1].

6.3 Effect of Content Change

The effect of content change on the processing delay was evaluated. A tuple (r_2, r_3, r_4) denotes the number of objects for each type. Two cases of content change were evaluated. The first case was changing the total number of objects in the scene. The second one was keeping the total number of objects fixed and evaluating different permutations of object numbers.

6.4 Comparison With Other Approaches

CAMDNN was compared with a set of *static* and *dynamic* solutions. In the static approach, the model allocation is

TABLE 2
The Hardware Characteristics of the Mobile MPSoC

CPU	Qualcomm Kyro 585 ARM CPU
GPU	Qualcomm Adreno 650
DSP	Qualcomm Hexagon
DRAM	LPDDR5/LPDDR4X SDRAM
Operating System	Linux

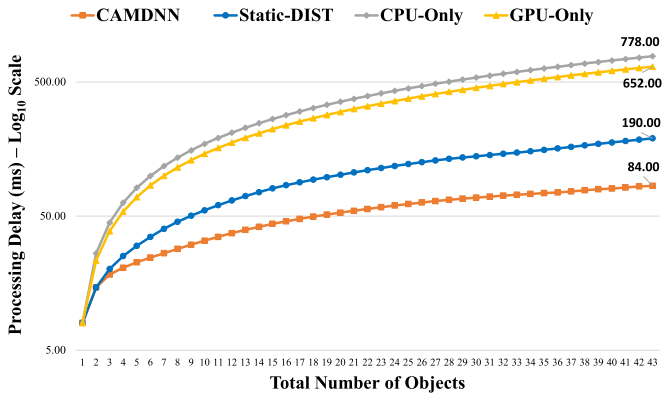

Fig. 10. Comparison between processing delay of *CAMDNN*, *Static-DIST*, *CPU-only*, and *GPU-only* over different number of objects. Each value on the abscissa represents the total number of objects. The corresponding value on the ordinate shows the average delay for all permutations of objects with the given total number of objects.

TABLE 3
Comparison Between Average Processing Delay of *CAMDNN*, *Static-DIST*, *GPU-Only*, and *CPU-Only* Over Different Combinations of Number of Objects

	Processing Delay (CAMDNN =1)	
	MEAN	STD
Static-DIST	1.83	2.57
CPU-only	6.67	8.04
GPU-only	5.60	6.87

The presented data are normalized to CAMDNN.

determined once before runtime, and remains fixed throughout the processing of the frame. Three static approaches were evaluated: *Static-DIST*, *CPU-only* and *GPU-only*. *Static-DIST* ignores the content of the scene and simply distributes the network of DNN models onto an MPSoC based on their heterogeneous execution time and their precedence structure. *CPU-only* and *GPU-only* run all the workloads on the CPUs and GPUs respectively. The dynamic scheduling policies include *HEFT* [38] and Central Queue (*CQ*) [17]. *HEFT* is a greedy algorithm in which the content of input data and the batching are not considered and scheduling decisions are based on the execution time of models and the remaining workload on each hardware unit. *CQ* only selects the fastest hardware unit among those that are available. As simple as these schemes are, they are indeed the common approaches described in the literature. Also, CAMDNN was compared with the IL-based scheduler proposed in [23] which in case of small graph sizes shows insignificant difference compared to *HEFT* performance.

7 Experimental Results

7.1 Static Solutions

In Fig. 10, the performance of three static solutions are compared with *CAMDNN* over different permutations of tuple (r_2, r_3, r_4). The x-axis represents the total number of objects and the corresponding ordinate value is the average completion time or delay over all numbers of object with the given total. As it can be seen, when there is only one of each object type in the scene, *CAMDNN* and *static-DIST* have identical performance which is also better than *CPU-only* and *GPU-only*. However, when the number of objects in the scene starts growing the performance gap between our approach compared to others becomes more significant.

Table 3 shows *CAMDNN* outperforms *Static-DIST*, *CPU-only*, and *GPU-only* solutions as much as 1.83x, 6.67x, and 5.60x. Moreover, the *CAMDNN* shows much lower performance variation (lower standard deviation) compared to other static solutions.

Table 4 shows the performance sensitivity to different distribution of objects. In this case, the total number of objects was set to 21, which is the most frequent sum over all permutations of tuple (r_2, r_3, r_4). *CAMDNN* shows a much lower mean and variance in the processing delay compared to the others. Also, multiple interesting observations can be noted as follow.

- Objects associated with SqueezeNet have a greater impact on performance. Due to DSP being limited to only INT8 models, SqueezeNet can only be executed on CPU and GPU. Therefore, the performance variation can be dependent on accuracy constraints.
- Performance variation for *CPU-only* and *GPU-only* shows linear relationship with the execution time of the model on CPU and GPU respectively. However, the performance variation for *CAMDNN* is related to the execution times, allocation and hardware compatibility.

These observations further emphasize the need for a content-aware scheduler which is also aware of hardware heterogeneity.

TABLE 4
Performance Sensitivity to Object Distribution With a Fixed Total Number of Objects

Total = 21	$r_2 = 7$		$r_3 = 7$		$r_4 = 7$	
	Mean	STD	Mean	STD	Mean	STD
CAMDNN	60.05	15.73	51.01	3.94	61.59	12.92
Static-DIST	111.67	45.84	99.00	3.36	113.33	40.48
CPU-only	393.00	52.62	393.00	93.56	393.00	49.19
GPU-only	330.00	42.43	330.00	85.57	330.00	49.19

The result for each column corresponds to fixing the number of objects for one type (r_i) and considering different combinations of objects for other types.

TABLE 5
Comparison Between Average Processing Delay of *CAMDNN*, *HEFT*, *CQ* Over 16 Different Scene Complexities

	Processing Delay (CAMDNN =1)	
	MEAN	STD
HEFT	1.32	1.45
CQ	2.17	2.56

The presented data are normalized

7.2 Dynamic Solutions

CAMDNN was compared to two runtime schedulers, *HEFT* and *CQ*, over 16 different scene complexities. *HEFT* and *CQ* are suboptimal due to lack of batching support and ignoring the large overheads of model loading on different processing units. Also, they ignore the dependencies in the application graphs and the cost of context switching between different models. All of these contribute to their suboptimal results.

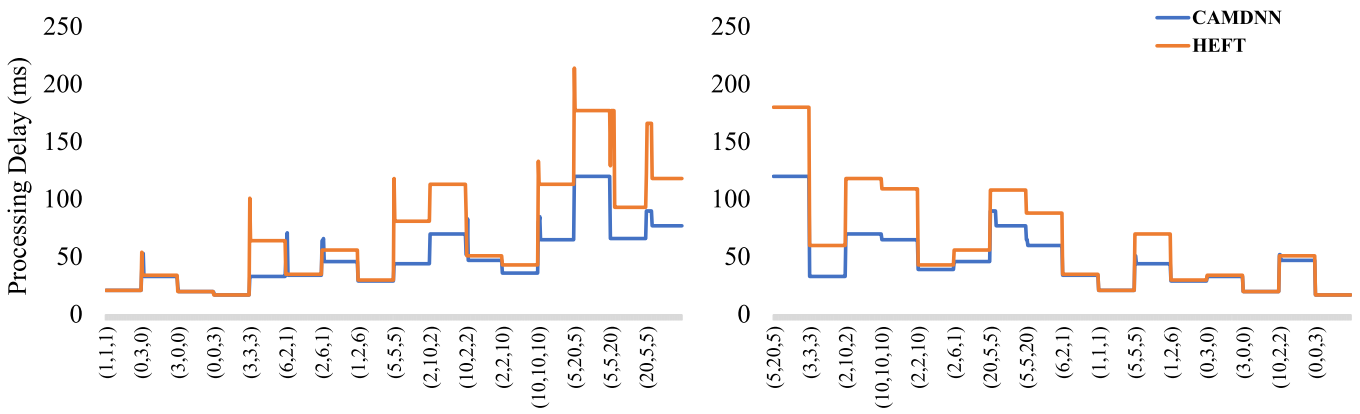

Fig. 11. The variation in the processing delay using our proposed approach versus *HEFT*. The tuple shows the number of objects for each type. There were 50 frames between each content change $epoch = 5sec$.

Table 5 shows the average and standard deviation of delay for 16 cases of scene complexity. As shown in Table 5, *CAMDNN* shows 32% and 2.17x lower processing delay on average compared with the *HEFT*, and *CQ* approaches respectively. Moreover, the proposed approach leads to lower standard deviation and more stable performance compared with other approaches.

Fig. 11 shows the performance delay for 800 frames using our approach versus (*HEFT*). The proposed approach is close to *HEFT* for the tuple (1,1,1). However, when the number of objects increases, the gap between the performance of two approaches increases. The performance is evaluated over two different sequences of scene complexity. As mentioned earlier, the previous allocation of models has a significant impact on subsequent allocations. For example, the transition from $(20, 5, 5) \rightarrow (5, 5, 20)$ triggers reallocation and *CAMDNN* runs (5,5,20) combination in 60 ms but $(5, 20, 5) \rightarrow (5, 5, 20)$ does not trigger reallocation because the cost of reallocation is higher than its benefit and runs (5,5,20) combination in 65ms.

7.3 Performance Sensitivity Analysis

Table 6 shows the sensitivity analysis of our content-aware mapping algorithm for different rates of content change. The cell values are the average processing delay for CAMDNN and the HEFT over 10,000 random scene complexities. As it can be seen, when the interval of content change is decreased (faster rate of content change), the processing delay is increased in both approaches.

However, CAMDNN outperforms HEFT in all rates of content change. Moreover, the rate of increase in processing delay when the interval of content change is reduced is lower for CAMDNN. This is due to the fact that CAMDNN considers the loading time of the models and also optimizes the batch size for running different models. To further improve the proposed approach, the scheduling epoch can be adjusted at runtime based on the rate of content change.

TABLE 6
Performance Sensitivity Analysis for a Fixed Scheduling Epoch (Epoch=5000ms) and Different Rates of Content Change

	Interval of Content Change (ms)			
	5000	1000	500	100
CAMDNN	58.70	76.97	83.20	84.08
HEFT	75.20	106.69	111.90	114.04

7.4 Overhead of Scheduler

The overhead of global and local scheduler were also measured on the mobile device. Recall, that the local scheduler has a time complexity of $O(1)$ since it only selects the best target for the current DNN. The measured time for the local scheduler was less than 1 ms. Also, the global scheduler which is based on an IL-based solution using a multi-layer perceptron (MLP) takes approximately 2.2 ms. The training phase takes about 2 hours on a laptop with a discrete mobile GPU RTX2060.

8 Conclusion

This paper presented a framework to map a network of DNNs onto resources on a heterogeneous MPSoC. In order to improve the processing delay, the framework considers the performance profile of models on computing units, loading time of models, and the batching of requests. Moreover, the presented method considers the variation in the content of input and adapt the decisions at run-time. The framework consists of a local and a global scheduler. The local scheduler works on a granularity of a DNN model inside the frame and chooses the suitable computing unit while the global scheduler customizes an Integer Linear Programming (ILP) solution to instantiate DNN remapping. Experiment results on a Qualcomm RB5 development demonstrates that the proposed approach shows on average up to 32% and 2.17x lower processing delay compared with (*HEFT*) and (*CQ*). Also, it shows on average up to 1.83x, 6.67x, and 5.60x lower processing delay compared to *static-DIST*, *CPU-only* and *GPU-only* respectively.

In our approach, we did not explicitly include thermal modeling in our decision making policy for two reasons: 1) The device manufacturers do not share any information about their thermal models. 2) The system that we have has a heat sink and a fan. Therefore, any thermal issues could not be observed. However, considering the power and thermal constraints due to a limited battery capacity and/or lack of active cooling, both thermal and power constraints can be embedded into our problem formulation. This is a part of our future work.

REFERENCES

[1] Tensorflow lite. 2021. [Online]. Available: https://www.tensorflow.org/lite

[2] Aleksey Bochkovskiy. YOLOv4 object detection. 2020. [Online]. Available: https://alexeyab84.medium.com/yolov4-the-most-accurate-real-time-neural-network-on-ms-coco-dataset-73adfd3602fe

[3] K. R. Baker and D. Trietsch, *Principles of Sequencing and Scheduling*. Hoboken, NJ, USA: Wiley, 2013.

[4] E. Bank-Tavakoli, S. A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, and M. Pedram, "POLAR: A pipelined/overlapped FPGA-based LSTM accelerator," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 28, no. 3, pp. 838–842, Mar. 2020.

[5] S. Baruah, "An ILP representation of a DAG scheduling problem," *Real-Time Syst.*, vol. 58, pp. 85–102, 2022.

[6] A. F. Biten et al., "Scene text visual question answering," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 4291–4301.

[7] Y. Chen, S. Biookaghazadeh, and M. Zhao, "Exploring the capabilities of mobile devices in supporting deep learning," in *Proc. ACM/IEEE 4th Symp. Edge Comput.*, 2019, pp. 127–138.

[8] Y. Choi, Y. Kim, and M. Rhu, "Lazy batching: An SLA-aware batching system for cloud machine learning inference," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 493–506.

[9] D. Crankshaw et al., "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 477–491.

[10] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 613–627.

[11] M. Farhadi, M. Ghasemi, S. Vrudhula, and Y. Yang, "Enabling incremental knowledge transfer for object detection at the edge," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops*, 2020, pp. 396–397.

[12] J. Forrest et al. coin-or/Cbc: Release releases/2.10.8 (releases/2.10.8). Zenodo, 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6522795

[13] M. Ghasemi, S. Heidari, Y. G. Kim, A. Lamb, C.-J. Wu, and S. Vrudhula, "Energy-efficient mapping for a network of DNN models at the edge," in *Proc. IEEE Int. Conf. Smart Comput.*, 2021, pp. 25–30.

[14] S. A. Ghasemzadeh, E. B. Tavakoli, M. Kamal, A. Afzali-Kusha, and M. Pedram, "BRDS: An FPGA-based LSTM accelerator with row-balanced dual-ratio sparsification," 2021, *arXiv:2101.02667*.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[16] U. Gupta et al., "The architectural implications of Facebook's DNN-based personalized recommendation," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architecture*, 2020, pp. 488–501.

[17] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*, Cambridge, U.K.: Cambridge Univ. Press, 2013.

[18] A. Ignatov et al., "AI benchmark: Running deep neural networks on Android smartphones," in *Proc. Eur. Conf. Comput. Vis. Workshops*, 2018, pp. 288–314.

[19] A. Ignatov et al., "AI benchmark: All about deep learning on smartphones in 2019," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. Workshops*, 2019, pp. 3617–3635.

[20] Y.-H. Kao, K. Wright, P.-H. Huang, B. Krishnamachari, and F. Bai, "MABSTA: Collaborative computing over heterogeneous devices in dynamic environments," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 169–178.

[21] Y. G. Kim and C.-J. Wu, "AutoScale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchit.*, 2020, pp. 1082–1096.

[22] S. Kong et al., "Deep hurdle networks for zero-inflated multi-target regression: Application to multiple species abundance estimation," 2020, *arXiv:2010.16040*.

[23] A. Krishnakumar et al., "Runtime task scheduling using imitation learning for heterogeneous many-core systems," 2020, *arXiv:2007.09361*.

[24] L. Liu, J. Tang, S. Liu, B. Yu, Y. Xie, and J.-L. Gaudiot, "π-rt: A runtime framework to enable energy-efficient real-time robotic vision applications on heterogeneous architectures," *Computer*, vol. 54, no. 4, pp. 14–25, Apr. 2021.

[25] C. Lugaresi et al., "MediaPipe: A framework for building perception pipelines," 2019, *arXiv:1906.08172*.

[26] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.

[27] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 270–288.

[28] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. IEEE Des. Autom. Test Europe Conf. Exhib.*, 2017, pp. 1396–1401.

[29] P. Márquez-Neila, M. Salzmann, and P. Fua, "Imposing hard constraints on deep networks: Promises and limitations," 2017, *arXiv:1706.02025*.

[30] Qualcomm Inc. Snapdragon Neural Processing Engine SDK,. 2022. [Online]. Available: https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk

[31] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 779–788.

[32] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 397–411.

[33] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," 2021, *arXiv:2102.01887*.

[34] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of Constraint Programming*, Amsterdam, Netherlands: Elsevier, 2006.

[35] S. Schaal, "Is imitation learning the route to humanoid robots?," *Trends Cogn. Sci.*, vol. 3, no. 6, pp. 233–242, 1999.

[36] A. Singh et al., "Towards VQA models that can read," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 8317–8326.

[37] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield, "Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2017, pp. 4241–4247.

[38] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[39] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput CNN inference on embedded ARM big LITTLE multicore processors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2254–2267, Oct. 2020.

[40] D. H. Wolpert, "Stacked generalization," *Neural Netw.*, vol. 5, no. 2, pp. 241–259, 1992.

[41] C.-J. Wu et al., "Machine learning at Facebook: Understanding inference at the edge," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2019, pp. 331–344.

[42] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1270–1278.

[43] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm," in *Proc. Eur. Conf. Parallel Process.*, 2003, pp. 189–194.

[44] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.

Soroush Heidari received the BS degree in electrical engineering from the Sadjad University of Technology, Mashhad, Iran, in 2010, and the MS degree in electrical engineering from Oklahoma State University, Stillwater, Oklahoma, in 2015. He is currently working toward the PhD degree with the School of Computing and Augmented Intelligence, Arizona State University, Tempe, Arizona, since 2017. He is a graduate research associate with Arizona State University. His current research interests focuses on enabling near real-time content-aware edge computing and scheduling machine learning workloads on heterogeneous edge MPSoCs. He spent his graduate research internship with RadiusAI, Inc., Tempe, Arizona, in 2019.

Mehdi Ghasemi received the BS degree in computer engineering from the Ferdowsi University of Mashhad, Mashhad, Iran, and the MSc degree in computer engineering from Shahid Beheshti University, Tehran, Iran. He is currently working toward the PhD degree with the School of Computing and Augmented Intelligence, Arizona State University, Tempe, Arizona, since 2017. He is a graduate research associate with Arizona State University. His research interests include Internet of Things (IoT), energy-aware computing, and computation offloading at the edge.

Young Geun Kim received the BS and PhD degrees from the Department of Computer Science, Korea University, in 2014 and 2018, respectively. He is currently an assistant professor with the Department of Computer Science and Engineering, Korea University. His research focus lies in the domain of computer system architecture with particular emphasis on energy-efficient and low-power systems. His research has pivoted into designing efficient systems for machine learning execution at the edge.

Carole-Jean Wu received the BSc degree from Cornell University, and the MA and PhD degrees from Princeton University. She is currently an associate professor with Arizona State University, and is currently a research scientist with Facebook AI Research. Her research lies in the domain of computer systems. Her recent research focuses on designing systems for machine learning execution at-scale and on tackling system challenges to enable efficient AI execution in a responsible way. She chairs the MLPerf Recommendation Benchmark Advisory Board and co-chaired MLPerf Inference. She was the recipient of the NSF CAREER Award, the Facebook AI Infrastructure Mentorship Award, the IEEE Young Engineer of the Year Award, the Science Foundation Arizona Bisgrove Early Career Scholarship, and the Intel PhD Fellowship, among a number of best paper awards.

Sarma Vrudhula received the BMath degree from the University of Waterloo, Waterloo, ON, Canada, and the MSEE and PhD degrees in electrical and computer engineering from the University of Southern California, Los Angeles, California. He is a professor of computer science and engineering with Arizona State University, and and the director of the NSF I/UCRC Center for Embedded Systems. Prior to ASU, he was a professor with the ECE Department, University of Arizona, Tucson Arizona, and was on the faculty of the EE-Systems Department, University of Southern California. He was also the founding director of the NSF Center for Low Power Electronics, University of Arizona. His work spans several areas in design automation and computer aided design for digital integrated circuit and systems, focusing on low power circuit design, and energy management of circuits and systems.