Thermal-Aware Scheduling for Deep Learning on Mobile Devices with NPU

Tianxiang Tan, *Student Member, IEEE,* and Guohong Cao, *Fellow, IEEE,*

Abstract—As Deep Neural Networks (DNNs) have been successfully applied to various fields, there is a tremendous demand for running DNNs on mobile devices. Although mobile GPU can be leveraged to improve performance, it consumes a large amount of energy. After a short period of time, the mobile device may become overheated and the processors are forced to reduce the clock speed, significantly reducing the processing speed. A different approach to support DNNs on mobile device is to leverage the Neural Processing Units (NPUs). Compared to GPU, NPU is much faster and more energy efficient, but with lower accuracy due to the use of low precision floating-point numbers. We propose to combine these two approaches to improve the performance of running DNNs on mobile devices by studying the thermal-aware scheduling problem, where the goal is to achieve a better tradeoff between processing time and accuracy while ensuring that the mobile device is not overheated. To solve the problem, we propose a heuristic-based scheduling algorithm to determine when to run DNNs on GPU and when to run DNNs on NPU based on the current states of the mobile device. The heuristic-based algorithm makes scheduling decisions greedily and ignores their future impacts. Thus, we propose a deep reinforcement learning based scheduling algorithm to further improve performance. Extensive evaluation results show that the proposed algorithms can significantly improve the performance of running DNNs on mobile devices while avoiding overheating.

Index Terms—deep learning, mobile computing, power management

1 Introduction

Over the past years, Deep Neural Networks (DNNs) have made tremendous progress and have been applied to various fields such as natural language processing [1], computer vision [2], [3], speech recognition [4], augmented reality [5], health care [6], etc., and many applications based on DNNs have been developed for mobile devices. For example, some apps can recognize users' voice as text input, help tourists identify well known landmarks, and monitor users' health conditions. Although DNNs can improve user experience by providing more accurate results, deep learning models are by nature computationally intensive, making them challenging to deploy on battery powered mobile devices. To address this problem, mobile devices can run DNNs on mobile GPUs. Some deep learning frameworks such as Tensorflow and Torch include tools and libraries to run existing DNNs on mobile GPU. Although the processing time can be significantly reduced, running DNNs on mobile GPUs consumes a large amount of energy. After a short period of time, the mobile device may become overheated, in which case the processors are forced to reduce the clock speed, significantly reducing the processing speed.

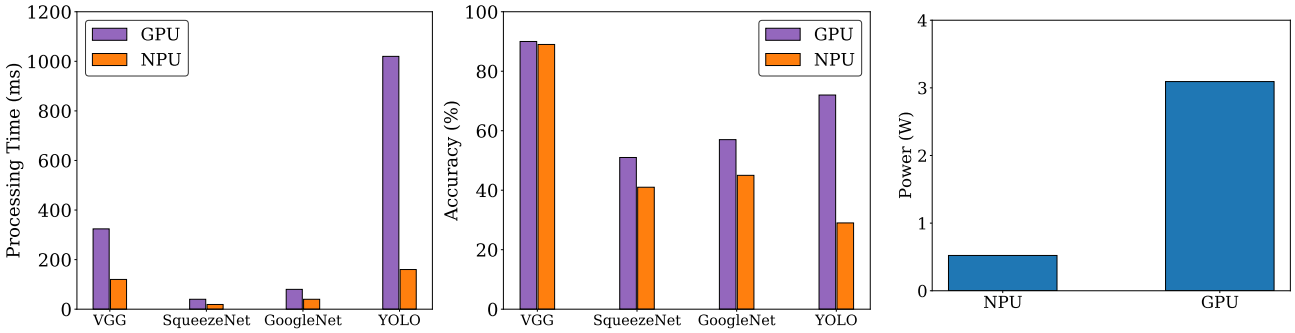
A different approach is to leverage the Neural Processing Units (NPUs) equipped with mobile devices. In recent years, many companies such as Huawei, Qualcomm, and Samsung have developed dedicated NPUs for mobile devices, which can process AI features [7]–[9]. Compared to GPU, NPU is much faster and more energy efficient. With NPU, the mobile device is unlikely to overheat even after continuously running DNNs for a long time. However, NPU has some fundamental limitations, such as low precision of floating-point numbers and limited memory space [10]. The DNNs have to be compressed and run with 8-bit integers or half precision floating-point numbers on NPU. As a result, the accuracy of running some DNNs on NPU is lower than that of GPU or CPU.

There is a performance tradeoff between the GPU-based approach and the NPU-based approach. The GPU based approach has higher accuracy, but it has higher energy consumption. Running DNNs on GPU continuously will cause the mobile device to overheat and result in poor performance. On the other hand, the mobile device is unlikely to be overheated in the NPU based approach, which is faster and more energy efficient. However, running some DNNs on NPU may result in low accuracy due to the use of low precision floating-point numbers.

To address this problem, we propose to combine these two approaches to improve the performance of running DNNs on mobile devices. The major challenge is to determine when to run DNNs on GPU to achieve better performance and when to run DNNs on NPU to avoid overheating. Consider an example of a flying drone. Its camera captures videos which are processed in real time to detect nearby objects to avoid crashing into trees or buildings. If the device becomes overheated, the processing speed will suddenly drop, which increases the risk of collisions. As a result, besides considering processing time and accuracy, it is also critical to avoid overheating (thermal throttling).

In this paper, we study the *Thermal-Aware Scheduling* (TAS) problem, where the goal is to achieve a better tradeoff between processing time and accuracy while ensuring that the mobile device does not overheat. To solve the problem, we propose a Heuristic-Based Scheduling (HBS) algorithm

• *The authors are with School of Electrical Engineering and Computer Science, Pennsylvania State University, University Park, PA 16802. E-mail: {txt51, gxc27}@psu.edu.*



(a) Processing time (b) Accuracy (c) Power consumption

Fig. 1: Performance comparisons of running DNNs on NPU and GPU.

to determine when to run DNN on GPU and when to run DNN on NPU based on the processor usage, the clock speed, and the temperature of the processor. The heuristic-based algorithm makes scheduling decisions greedily and ignores their future impact. To address this issue, we propose a Deep Reinforcement Learning based Scheduling (DRLS) algorithm to further improve the performance. In DRLS, deep Q-learning [11] is used to evaluate the future impact of a scheduling decision. In the training procedure of DRLS, we carefully design the reward function and propose techniques for replay buffer management.

Our contributions can be summarized as follows.

- We measure the performance of running DNNs on GPU and NPU, and identify the limitation of the current mobile architecture such as overheating when running computationally intensive DNNs.
- We formulate the Thermal-Aware Scheduling (TAS) problem and propose Heuristic-Based Scheduling (HBS) algorithm and Deep Reinforcement Learning Scheduling (DRLS) algorithm to solve it.
- We implement the proposed algorithms on mobile devices with NPU. Extensive evaluation results show that the proposed algorithms can significantly improve the performance of running DNNs on mobile devices while avoiding overheating.

The rest of the paper is organized as follows. Section 2 compares GPU-based approach and NPU-based approach, and provides the motivation of Thermal-Aware Scheduling. In Section 3, we formalize the Thermal-Aware Scheduling problem. Section 4 presents the HBS algorithm and Section 5 presents the DRLS algorithm. In Section 6, we give the evaluation results. Section 7 presents related work and Section 8 concludes the paper.

2 Preliminary

In this section, we first use some measurement results to identify the strengths and weaknesses of GPU and NPU, and then give the motivation of thermal aware scheduling.

2.1 Comparison Between GPU and NPU

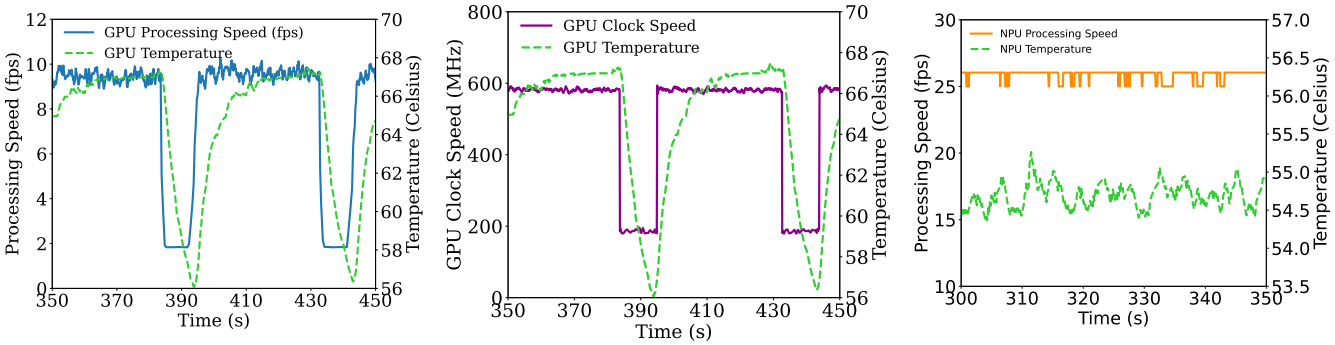
To have a better understanding of the characteristics of GPU and NPU, we compare the processing time, accuracy and power consumption of running different DNNs on GPU and NPU.

The experiment was conducted on HUAWEI mate 10 pro which has NPU and GPU, and the evaluations are based on the following DNN models: 1) VGG model [12] with the LFW dataset, 2) SqueezeNet model [13] with 4000 object images randomly chosen from the FCVID dataset [14], 3) GoogleNet model [15] with the same dataset as the SqueezeNet model, 4) YOLO Small model [3] with 100 images randomly chosen from the COCO dataset [16]. As shown in Figure 1(a), compared to GPU, running VGG, SqueezeNet, and GoogleNet on NPU can reduce the processing time by 50-65% compared to using GPU. From Figure 1(b), we can see that the speed improvement of NPU is at the cost of accuracy loss, although the accuracy loss of using NPU is different for different DNN models. For example, compared to GPU, using NPU has similar accuracy when running VGG, 15% accuracy loss when running SqueezeNet and GoogleNet, and the accuracy drops by 45% when running YOLO Small.

The accuracy loss is mainly because NPU only supports FP16 operations. This design option can reduce the processing time but may increase the accuracy loss because of floating-point number overflow or underflow. Although well-known deep learning frameworks such as Tensorflow Lite also use FP16 operations, these FP16 operations are limited to store model parameters and input data, and FP32 are still used to store intermediate results and for most operations (e.g., accumulation).

The amount of accuracy loss is related to the DNN model. In VGG, the extracted feature vectors are compared, and they represent the same person if the similarity is above a predefined threshold. Although NPU introduced error may change some values, the relationship between the similarity and the threshold will remain and thus keep the same level of accuracy. SqueezeNet and GoogleNet classify images based on the largest element in the extracted feature vector. The NPU introduced errors may increase or decrease the value of some elements, and then classify an object as something else. In YOLO Small, there is more information such as location, category and size of the objects in the feature vector. Since any error in the feature vector can totally change the detection result, its accuracy loss is much higher.

We also compared the power consumption of GPU and NPU. We use the fuel gauge on the mobile device to retrieve battery voltage and current information. Such information can be obtained using the Android SDK and we have devel-



(a) (b) (c)

Fig. 2: The relationship among processing speed, temperature and processor clock speed

oped a mobile app to record the instantaneous voltage and current of the battery every 100ms. As shown in Figure 1(c), the power consumption of NPU is about 85% less than that of GPU. Considering that the processing time of running DNN on GPU is also much longer than NPU (see Figure 1(a)), the NPU-based approach is about 10-20 times more energy efficient than the GPU-based approach.

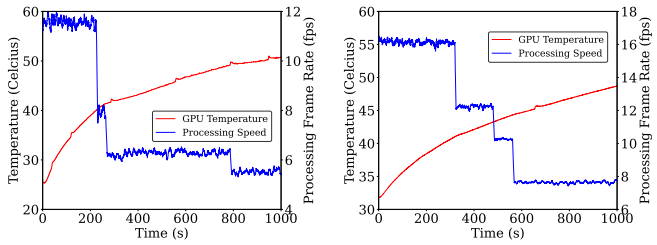
2.2 Motivation

The current mobile architecture is not designed for running computationally intensive DNNs, especially when the DNNs are required to run for some amount of time. This poses research challenges for supporting some deep learning based applications (e.g., video analytics). When running these deep learning applications, the processor works at high clock speed and its temperature increases sharply. Within a short period of time, the mobile device will overheat and the processor will be forced to reduce the clock speed, significantly reducing the processing speed.

To illustrate this problem, we conducted some experiments using HUAWEI Mate 10 pro. We developed an Android application which runs as a background process to record the processing speed and the temperature of the processors every 100ms. Moreover, we also use ARM Streamline, which is a professional performance profiling software, to monitor the clock speed of GPU. In the experiment, GoogleNet is run on GPU to process videos randomly selected from the FCVID dataset.

Figure 2(a) shows the processing speed and the temperature of the GPU over time. In the beginning, the temperature is low, and GoogleNet is processed quickly on GPU. After time 385s, the GPU temperature increases to 67°C, and the device overheats. The system cools down the device by reducing the power consumption of the processors and forcing them to work at a low frequency, i.e., the GPU clock speed drops from 600 to 200MHz as shown in Figure 2 (b). As a result, the GPU processing speed becomes much slower; i.e., drops from 10 frames per second (fps) to 2 fps as shown in Figure 2 (a). When the GPU temperature drops to 56°C, the system increases the GPU clock speed and the processing speed increases back to 10 fps. This phenomenon is not limited to a specific type of smartphone. In Figure 3, we performed the same experiment on two different smartphones, Samsung S20 and Pixel 4. On Pixel 4, the data from the GPU temperature sensor (obtained from the /sys folder) consists of random numbers. Therefore, we approximate the GPU temperature using CPU temperature for both devices. As shown in the figure, the performance of running DNN on GPU drops as the GPU temperature increases. In Samsung S20 and Pixel 4, automatic shutdown occurs after multiple clock speed reductions during overheating. As a result, Figure 2 looks different from Figure 3, where Mate 10 Pro recovers from overheating by significantly reducing the clock speed.

Based on Figures 2 and 3, we can see that the GPU processing speed experiences a significant drop when the system overheats. To address this problem, we leverage NPU to avoid overheating and improve performance. Since NPU consumes much less power and generates much less heat, it does not overheat and can maintain the high processing speed. As shown in Figure 1(c), NPU only consumes about 16% of power compared to GPU. Moreover, the heat generated by NPU can be absorbed and dissipated quickly, as shown in 2(c). Note that we use the CPU temperature to approximate the temperature of NPU since it cannot be directly obtained from the system. In this experiment, we run GoogleNet on NPU, and measure its processing speed. Since there is no temperature sensor on NPU, we show the temperature of CPU which is in the same System on a Chip (SoC). As shown in the figure, there is no significant temperature increase for CPU since it works at low utilization, and there is no overheating. The processing speed of NPU (25 fps) is much faster than GPU.



(a) Samsung S20 (b) Pixel4

Fig. 3: The performance of GPUs under different temperatures.

From Figures 1 and 2, we can see that NPU is much faster and more energy efficient than GPU. When DNN is run on NPU, the device does not overheat since the temperature of the processors remains low and stable. However, the accuracy of some DNNs drops on NPU and it is difficult to satisfy the accuracy requirements of some mobile applications. On the other hand, the accuracy of the GPU based approach is higher, although the device may be overheated and the performance becomes much worse after continuously running DNN for a period of time. To address this issue, we combine these two approaches to achieve better and robust performance under thermal constraints. We first formulate the Thermal-Aware Scheduling problem and propose a heuristic-based scheduling algorithm to solve it. Then, we propose a deep reinforcement learning scheduling (DRLS) algorithm to further improve the performance.

3 The Thermal-Aware Scheduling (TAS) Problem

In this section, we study the Thermal-Aware Scheduling (TAS) problem. The utility is defined as a weighted function of accuracy and processing speed, and our goal is to maximize the utility while ensuring that the device does not overheat.

3.1 Problem Formulation

Let n denote the total number of video frames that needs to be processed, and each frame can be skipped or processed on GPU or NPU. Let P denote the set of processors (NPU or GPU) on which the DNNs can be run. Let $A(j)$ denote the accuracy of running DNN on the j^{th} processor. More specifically, $A(1)$ is the accuracy of running the DNN on GPU and $A(0)$ is the accuracy of running the DNN on NPU. Let α denote the tradeoff parameter between accuracy and the processing speed. The utility can be computed as $\sum_{i=1}^{n}\sum_{j} A(j)X_i^j + \alpha\frac{\sum_i\sum_j X_i^j}{n}$. X_i^j is a variable to show which processor is used to process the frame. If $X_i^j = 0$, the i^{th} frame will not be processed on the j^{th} processor. If $X_i^1 = 1$, the i^{th} frame will be processed on the j^{th} processor. Let $T(t)$ denote the temperature of the CPU at time t and let T_o denote the overheating threshold of the CPU. Then, we have $T(t) \leq T_o$.

Although the DNN is run on GPU or NPU, we only consider the temperature of CPU (T_t and T_o) for the following reasons. First, even though the DNN is run on GPU, the temperature of the CPU is the highest among all the processors. This is because the CPU needs to perform many operations such as data preprocessing (i.e., scaling and normalization) and copy data from main memory to GPU or NPU [9], [17], [18]. Moreover, all the processors are located on the same SoC in mobile devices and the temperature of GPU also affects that of the CPU. As a result, the temperature difference between CPU and GPU is usually within 2 degrees Celsius. Most importantly, the mobile devices determine whether the device is overheated or not based on the temperature of the CPU instead of GPU or NPU. Therefore, we only consider the CPU temperature in this problem formulation.

Notation	Description
I_i	The i^{th} frame
$A(j)$	The accuracy of running the DNN on the j^{th} processor
f	Processing speed (fps)
$T(t)$	The temperature of the CPU at time t
T_o	The overheating temperature threshold.
T_h	The predefined temperature threshold used in HBS and DRLS
n	The number of video frames that needs to be processed
X_i^j	a variable indicating that the i^{th} frame is processed on the j^{th} processor.

TABLE 1: Notations

The notations used in the problem formulation and the algorithm design are shown in Table 1. The problem can be formulated in the following way.

$$\max \quad \sum_{i=1}^{n}\sum_{j\in P} A(j)X_i^j + \alpha\frac{\sum_{i=1}^{n}\sum_{j\in P} X_i^j}{n} \tag{1}$$

$$s.t. \quad T(t) \leq T_o, \forall t \tag{2}$$

$$\sum_{j\in P} X_i^j \leq 1, \forall i \in [1, n] \tag{3}$$

Objective (1) is to maximize the utility of the processed frames. Constraint (2) ensures that the device temperature is below a predefined threshold at any time. Constraint (3) specifies that each frame can be either skipped or processed on GPU or NPU.

The formulation is difficult to solve directly due to the following reason. First, it is hard to model the thermal changes of the mobile device directly [19]. The temperature change of the mobile device depends on many factors such as the clock speed of the processors, the environment, and other components of the mobile device, which are difficult to model [19]–[21]. For example, it is hard to model the heat exchange among different components of the mobile device. The temperature of a processor is not only based on its clock speed and utilization but also affected by other processors due to the physical proximity. Similarly, it is hard to model the heat exchange between the mobile device and the environment, since it is affected by the environment temperature, the device material and its heat dissipation. Moreover, even if a device temperature model can be built, solving the nonlinear optimization problem will take too much time. With limited computational power, it is impractical to implement such a complex algorithm on mobile devices. Therefore, we first propose a heuristic-based solution to solve the problem and then design a deep reinforcement learning based solution to further improve the performance.

3.2 Overview of the Thermal-Aware Scheduling Framework

An overview of the Thermal-Aware Scheduling framework is shown in Figure 4. Since some applications continuously work for a long time, the mobile device may overheat after several minutes and the performance drops significantly. The thermal-aware scheduling algorithms ensure that the

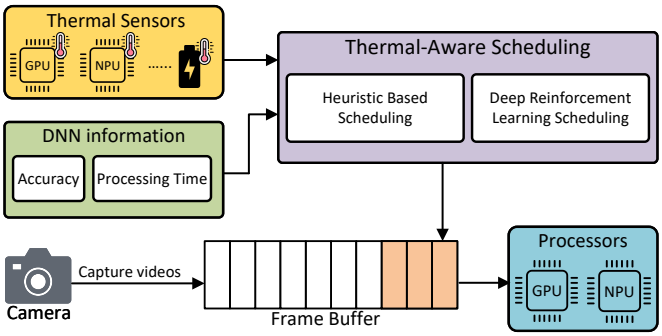


Fig. 4: Overview of Thermal-Aware Scheduling.

temperature of the device does not exceed a predefined threshold. To avoid overheating, the scheduling algorithm needs to adjust the processing frame rate by skipping some video frames and determine which processor should be used for processing the frames to achieve better performance.

4 Heuristic-Based Scheduling Algorithm

Since it is impractical to solve the TAS problem directly, we propose a heuristic-based scheduling (HBS) algorithm. In HBS, time is divided into time intervals. At the beginning of each time interval, the HBS algorithm makes a scheduling decision to determine where to run the computation and how many frames should be processed, through the following procedure. First, HBS predicts the temperature change based on the processing speed, the processor usage, and the history data. Then, a scheduling decision is made to maximize the utility within the temperature threshold.

4.1 Predicting the Temperature Change

Many factors such as DNN architecture, clock speed of the processor, and the environment affect the temperature of the mobile device. To predict the temperature change, we extract features related to the DNN model and the mobile device. Our feature vector $\overrightarrow{x} = (x_1, x_2, \ldots, x_m)$ contains three parts: the DNN features, the processor features, and the thermal features. The DNN features include the processing time of the DNN and the processing frame rate. The processor features include the clock speed of the processor and the processor usage information. The thermal features include the temperature of different components in the mobile devices (e.g., processor, display and battery). Since the extracted features have different ranges, we use Min-Max normalization to re-scale the range of features to $[0, 1]$.

With the extracted features, HBS predicts the temperature change based on the historical data. More specifically, let f denote the processing speed and let p denote the processor being used. HBS maintains the historical data in a set of lists $\{l(f_1, p_1), l(f_2, p_2), \ldots l(f_m, p_m)\}$. $l(f_i, p_i)$ is a list of pairs $(\overrightarrow{x}, \Delta T)$, where the feature vector $\overrightarrow{x}$ represents the current state of the mobile device and ΔT is the CPU temperature change after running DNN on processor p_i with processing speed f_i. A new pair is added to the list $l(f_i, p_i)$ as follows. Suppose at the beginning of a time interval, the extracted feature vector is $\overrightarrow{x}$, and HBS determines that the processing speed should be f_i and the computation should be run on processor p_i. At the end of the time interval, HBS obtains the CPU temperature change ΔT. Then, a pair $(\overrightarrow{x}, \Delta T)$ is added to the list $l(f_i, p_i)$.

Assume that the feature vector representing the current state of the mobile device is $\overrightarrow{x'}$. If the processing speed is f' and the DNN is run on p', HBS predicts the temperature change as follows. It searches the list $l(f', p')$ to find the pair $(\overrightarrow{x^*}, \Delta T^*)$ with the smallest $d(\overrightarrow{x}, \overrightarrow{x'})$. Here, $d(\overrightarrow{x}, \overrightarrow{x'})$ is the Euclidean distance of vectors, and it is defined as $\sqrt{\sum_i (x_i - x_i')^2}$. Then, the predicted temperature change is ΔT^*.

Since HBS is frequently called and the mobile applications may run for a long time, The efficiency of HBS may become worse when a large number of pairs are added to the list $l(f, p)$. To address this issue, HBS only keeps 50 historical data pairs in $l(f, p)$. When $|l(f, p)| = 50$ and a new pair is added to $l(f, p)$, HBS calculates the similarity among pairs based on the Euclidean distance and keeps the 50 most different pairs. More specifically, a minimum spanning tree algorithm is run to remove a pair from $l(f, p)$. Each pair in $l(f, p)$ is a node in the graph. The distance between two nodes $(\overrightarrow{x}, \Delta T)$ and $(\overrightarrow{x}, \Delta T)$ is defined as $\frac{1}{d(\overrightarrow{x}, \overrightarrow{x'})}$. HBS runs Prim's algorithm to create the spanning tree and the last pair added to the tree will be removed from $l(f, p)$.

4.2 Making a Scheduling Decision

In the HBS algorithm, because the temperature prediction may have errors and a small error may cause overheating and reduce the performance, instead of using T_o, HBS defines a threshold T_h which is several degrees lower than T_o. HBS ensures that the device temperature is around T_h. In Section 6, we will experimentally choose the value of T_h.

Based on the temperature estimation, HBS determines the processor p and its processing speed f to maximize the utility. More specifically, for each possible scheduling decision f and p, HBS predicts the temperature T. If $T > T_h$, HBS will not consider this scheduling decision since HBS needs to control the temperature of the device to avoid overheating. If $T \leq T_h$, HBS will calculate the utility as $A(p) + \alpha f$.

The HBS algorithm is summarized in Algorithm 1. Lines 2-5 maintain the list $l(f, p)$. In Lines 6-14, HBS finds the scheduling decision to maximize the utility. In the algorithm, $U_{best}, f_{best}, p_{best}$ are used to keep track of the best utility and the best scheduling decision that have been found so far. The running time of the algorithm is $O(n_f)$, where n_f is the number of processing speeds for running the DNN.

5 Deep Reinforcement Learning Based Scheduling Algorithm

The HBS algorithm can be used to improve the performance and the robustness of running DNNs on mobile devices by ensuring that the device temperature does not exceed the overheating threshold. However, HBS has some limitations. For example, HBS makes greedy scheduling decisions based on the current state of the mobile device, and ignores the future impact of the scheduling decision. To address this issue, reinforcement learning technique is used to find a

Algorithm 1: Heuristic-Based Scheduling Algorithm

Input : The processing speed f^{prev} and the processor p^{prev} used in the last time interval.

Output: The processing speed f, the processor p used for running DNN

Extract the feature vector $\overrightarrow{x'}$ and the current device temperature T'.
Add $(\overrightarrow{x'}, T')$ to $l(f^{prev}, p^{prev})$.
if $|l(f^{prev}, p^{prev})| > 50$ **then**
  Use Prim's algorithm to create minimum spanning tree.
  Remove the last pair added to the spanning tree.
$f_{best} \leftarrow f_{max}, p_{best} \leftarrow 0$
$U_{best} \leftarrow A(p_{best}) + \alpha f_{best}$
for $p \leftarrow 0$ *to* 1 **do**
  for $f \leftarrow 0$ *to* f_{max} **do**
    $(\overrightarrow{x^*}, T^*) \leftarrow \arg\min_{(\overrightarrow{x}, T)} d(\overrightarrow{x'}, \overrightarrow{x})$
    if $T^* \leq T_h$ **then**
      $u \leftarrow A(p) + \alpha f$
      if $u > U_{best}$ **then**
        $U_{best} \leftarrow u$, $f_{best} \leftarrow f$, $p_{best} \leftarrow p$
return f_{best}, p_{best}

better schedule decision based on the status of the mobile devices.

5.1 Markov Decision Process (MDP)

Typically, a reinforcement learning problem can be modeled as a Markov Decision Process (MDP) [22] which consists of an environment and an agent. In our problem, the environment is the mobile device, which runs a program to process the frames on GPU or NPU, and the agent is our scheduler which gets information from the environment and makes scheduling decisions.

The information which is obtained from the environment is called the state. Let $\boldsymbol{s}_i$ denote the state of the environment at time i, and $\boldsymbol{s}_i$ is a feature vector which includes the temperature of different components in mobile devices and the clock speed of the processors.

Given the state $\boldsymbol{s}_i$, the agent generates an action a_i based on a policy $\pi(a_i|\boldsymbol{s}_i)$. In our problem, the actions a_i is the schedule decision which determines the processing speed and the processor used for i^{th} time interval, and the policy $\pi(a_i|\boldsymbol{s}_i)$ is the probability $P(a_i|\boldsymbol{s}_i)$ that the agent takes action a_i at state $\boldsymbol{s}_i$.

By performing an action a_i, the agent changes the state of the mobile device and gets a reward $R(\boldsymbol{s}_i, a_i)$, which is equivalent to the utility of the processed frames. To evaluate how well the agent performs from time $t = 0$ to T, a value function $V_\pi(\boldsymbol{s}_0)$ is used. It represents the expected reward and can be computed as follows $V_\pi(\boldsymbol{s}_0) = E[\sum_{t=0}^{T} \gamma^t R(\boldsymbol{s}_i, a_i)|\boldsymbol{s}_0]$, where $\gamma \in [0, 1]$ is the discount factor. The goal of our problem is to maximize the value function $V_\pi(\boldsymbol{s}_0)$ by finding the optimal policy $\pi^* = \arg\max V_\pi(\boldsymbol{s}_0)$. Based on Bellman's equation, the optimal value function can be expressed as follows $V^*(\boldsymbol{s}_i) = \max_{a_i} Q(s_i, a_i) = R(\boldsymbol{s}_i, a_i) + \max_{a_i} \gamma \sum_{\boldsymbol{s}_{i+1} \in S} P_{a_i}(\boldsymbol{s}_{i+1}, \boldsymbol{s}_i) V^*(\boldsymbol{s}_{i+1})$, where $Q(s_i, a)$ is the Q-value which represents the expected discounted reward that can be achieved if perform a_i at state s_i and then follow optimal policy π^* from then on. If an action can result in higher Q-value, the agent should be trained to favor this action. $P_{a_i}(\boldsymbol{s}_{i+1}, \boldsymbol{s}_i)$ is the probability of transiting from state s_i to s_{i+1} under action a_i. In other words, the state transition depends on the schedule decision and the current state of the mobile device.

However, we could not apply the MDP to our problem directly for the following reasons. Firstly, the state of the mobile device is continuous instead of discrete as required by the MDP model. For example, the temperature of the mobile devices is not discrete and there are infinite possible states. Moreover, the state transition $P_{a_i}(\boldsymbol{s}_{i+1}, \boldsymbol{s}_i)$ is a known prior in the MDP but it is unknown in our problem. To address this issue, we propose a Deep Reinforcement Learning Scheduling (DRLS) algorithm which is based on deep Q-learning to solve the problem.

5.2 Deep Reinforcement Learning Scheduling (DRLS)

The overview of DRLS is shown in Figure 5. Compared to the MDP model, DRLS is based on deep Q-learning, it uses a deep neural network to evaluate the Q-value. Different from MDP, the reward in DRLS cannot be simply defined as the utility used in the problem formulation since DRLS needs to avoid overheating and utility does not include the penalty for overheating.

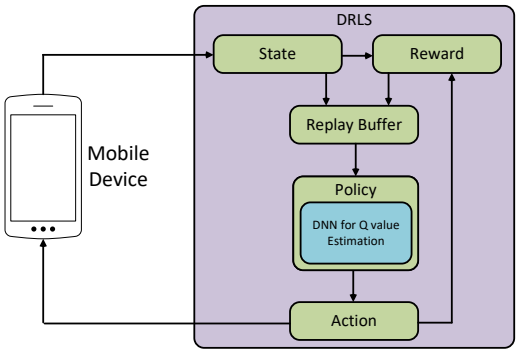


Fig. 5: Overview of DRLS.

Intuitively, the reward should encourage DRLS to maximize the utility under the temperature threshold T_o and prevent DRLS from making scheduling decisions that can cause the device to overheat. When the temperature is approaching the threshold, the reward should be smaller or even be negative. More specifically, similar to the HBS algorithm, DRLS aims to control the temperature around T_h ($T_h < T_o$). When the temperature is below T_h, the mobile device is considered to be cool and the reward is the same as the utility. When $T_h < T \leq T_h + \frac{3}{4}(T_o - T_h) = \frac{3}{4}T_o + \frac{1}{4}T_h$, the device temperature is already above the temperature threshold, the agent should get a negative reward when the temperature increases and get a positive reward if the temperature drops. When $T_h + \frac{3}{4}(T_o - T_h) < T \leq T_o$, the temperature is close to the overheating threshold, the agent should get a negative reward. If the temperature is above T_o,

the algorithm fails to control the temperature of the mobile device and the agent will get a large negative reward. The definition of the reward in DRLS can be summarized as follows.

$$\boldsymbol{r}_t(f,p) = \begin{cases} A(p) + \alpha * f, & T \in [0, T_h) \\ -0.2 * (\frac{3}{4}T_o + \frac{1}{4}T_h - T), & T \in [T_h, \frac{3}{4}T_o + \frac{1}{4}T_h) \\ -10, & T \in [\frac{3}{4}T_o + \frac{1}{4}T_h, T_o) \\ -99, & T \in [T_o, +\infty) \end{cases}$$

For the reward parameters, we heuristically tried different options using beam search algorithm and use the best one in reward function.

In deep Q-learning, a DNN is used to estimate the Q-value for each possible action based on the current state. DRLS implements a 2-layers fully connected feedforward DNN, which has 128 units in the first layer and 64 units in the second layer, for Q-value estimation. To train the DNN, the Bellman equation is used to update the Q-values.

More specifically, the parameters of the DNN are initialized to random values. At the i^{th} time interval, the agent makes a scheduling decision $\boldsymbol{a}_i$ and gets a reward R_i. The state is changed from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$. The loss $L_i(\theta_i)$ can be expressed as follows $L_i(\theta_i) = R_i + \gamma \max_{\boldsymbol{a}} Q(\boldsymbol{x}_{i+1}, \boldsymbol{a}) - Q(\boldsymbol{x}_i, \boldsymbol{a}_i)$, where the θ_i is the weights in the DNN. With the loss, the gradients in each layer can be computed, and the parameters are adjusted to better fit the new Q-values. The agent continuously makes scheduling decisions and observes rewards and state changes of the mobile device. After thousands of iterations, the loss is minimized and the DNN is trained for estimating the Q-values.

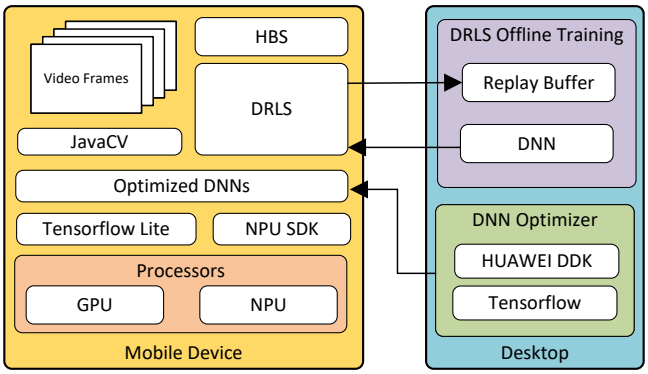


Fig. 6: The implementation details.

To improve the accuracy of Q-value estimation, a replay buffer is used to remember the old sample data and the DNN is trained using both new and old sample data. The replay buffer must be carefully designed in order to achieve high performance. For example, if DRLS is trained with a FIFO replay buffer, it will not process the frames with the highest processing speed on GPU even though the mobile device has a very low temperature. This is because, as time goes by, the replay buffer is dominated by the sample data which represents that the device has a high temperature. The device keeps running computationally intensive applications and it is impossible to cool down the device to a low temperature. DRLS keeps learning how to make good scheduling decisions when the device temperature is high but forgets how to find a good scheduling decision when the temperature of the device is low. This phenomenon is called catastrophic forgetting, which means the model learns new knowledge but forget the previous knowledge.

To avoid catastrophic forgetting, the replay buffer in DRLS should not be a FIFO queue. Instead, we use similar idea in the HBS algorithm (i.e., using Prim's algorithm to remove redundant pairs in the list $l(f,p)$) to maintain the replay buffer. More specifically, the items in replay buffer can be represented as $b_i = (\overrightarrow{x}_i, f, p, \boldsymbol{r}_i, \overrightarrow{x}_{i+1})$. Suppose the replay buffer is full and a new sample arrives. Similar to HBS, DRLS runs the Prim's algorithm to generate minimum spanning trees of the items. Each item can be considered as a node in the graph and the distance between item b_i and b_j is $\frac{1}{d(\overrightarrow{x}_i, \overrightarrow{x}_j)}$. Then, the last item added to the minimum spanning tree will be removed from the replay buffer. If the last item is newly added, it will be kept in the replay buffer and the second to last item added to the tree will be removed.

We also set a timer in the training procedure. When the time is up, the device will sleep for 10 minutes so that it can be completely cooled down before next iteration. When the device restarts the training procedure, more sample data which represents the device has low temperature, can be collected. With the above method, the DRLS can be trained to make good scheduling decisions.

6 Performance Evaluation

In this section, we evaluate the performance of the proposed thermal-aware scheduling algorithms on smartphones. We first present the evaluation setup and then present the evaluation result.

6.1 Evaluation Setup

In the evaluations, we use a HUAWEI mate 10 pro, which is equipped with 6 GB memory, octa-core CPU with big.LITTLE architecture (4x2.4 GHz and 4x1.8 GHz) [23] and a Cambricon NPU. To run the DNNs on GPU, we leverage the Tensorflow Lite library [24]. Since the architecture of NPU is different from CPU or GPU, the existing DNNs must be optimized before running on NPU. HUAWEI has published the SDK called HUAWEI DDK which includes toolsets to perform such optimization. DNNs trained with Caffe [25] and Tensorflow deep learning framework can be optimized and run on NPU.

To make scheduling decisions, both HBS and DRLS need information related to the mobile device status (e.g., processor clock speed and temperature). Since some information cannot be accessed through the Android SDK, we wrote a program to obtain such information by reading the system files. More specifically, our program reads the files in the directory /sys/devices/virtual/thermal/ to obtain temperature information, and it takes about 10ms to get all information from the system files.

In the experiment, the smartphone continuously runs object classification tasks which are commonly used in many deep learning based mobile applications. We use

SqueezeNet [13] and GoogleNet [15] in the evaluation because they are well known models and are widely adopted by many researchers for different tasks. Moreover, both SqueezeNet and GoogleNet have compact structures and they are very suitable to be run on smartphones which have limited computational power.

To measure performance, we use video frames from the FCVID dataset [14], which is used for training DNNs related to object classification and activity recognition. The FCVID dataset includes many real-world videos, with data size about 1.9 TB. Since the dataset is very large, it is impossible to process all video frames on smartphones.

We randomly choose 60 videos from the dataset for object classification, and we filter out the noisy data. Before running the DNNs, the video frames should be preprocessed, for example, the frames need to be normalized or be resized to the input size of the DNNs. In the experiment, we use JavaCV to perform such frame preprocessing on smartphones.

Since the HBS algorithm is easy to implement, no additional library is required. In contrast, the DRLS algorithm is more complex, where a DNN is trained to estimate the Q-values for every scheduling decision. Training the DNN on Android is not easy due to the following reasons. First, most existing Android deep learning frameworks such as Torch, Tensorflow Lite have limited support on running a pre-trained DNN on smartphones, and the API is not flexible for training the deep reinforcement learning models in these frameworks on mobile. Even though some frameworks such as Deeplearning4j [26], can be run on Android devices to train deep reinforcement learning models, the API is not flexible enough to manage the replay buffer, which is important for DRLS to achieve better performance. Second, the smartphones have limited computational resources and it is time consuming to train DNNs on smartphones.

To address this issue, the training procedure includes multiple rounds of offline training. More specifically, in each round, the DRLS algorithm is run to make scheduling decisions and the sample data is collected from the mobile devices. After 30 minutes, the sample data is offloaded to a powerful desktop which is equipped with Intel i7-10700K, NVIDIA GeForce RTX 3080 and 32GB memory. The DRLS algorithm is fine-tuned with the latest sample data. Then, the updated DRLS is deployed on the smartphone and a new round of offline training is started. The implementation of our framework is shown in Figure 6.

In the experiment, our algorithms are tested in two different environments. The first one is in the open space, where the smartphone is placed on the table. The other one is inside the VR headsets, such as Google cardboard where the smartphone is tightly wrapped by the cardboard. Many mobile VR applications have been developed and they may run DNNs in the background to provide better user experience. For example, a DNN may be run in the background to accelerate the VR scene rendering speed by predicting the users' head movement. Compared to open space, it is more difficult for the smartphone to dissipate heat and cool down in the confined space.

Method	MSE	ME
Android	0.013	1.25
HBS	0.001	0.45

TABLE 2: The comparison between Android based approach and HBS

6.2 Temperature Estimation in HBS

The HBS algorithm leverages the historical data to estimate the temperature changes, and we evaluate the effectiveness of our temperature prediction in this subsection. We compare our method with the one used in Android SDK which estimates the thermal headroom with a linear regression model based on the recent temperature data. We first run the HBS algorithm on the smartphone and collect the data trace. Then, we implement the same Android SDK (API 30) algorithm on a desktop and replay the data trace to predict the temperature of the next time interval. The evaluation is based on two commonly used metrics, Maximum Squared Error (MSE) and Maximum Error (ME). MSE measures the difference between the ground truth temperature and the predicted temperature. It measures the average performance of temperature estimation on all the data and can be expressed as $\frac{\sum_i^n (y_i - \hat{y}_i)^2}{n}$, where y_i represents the ground truth in the i^{th} time interval and $\hat{y}_i$ represents the predicted temperature. Different from MSE, ME focuses on the maximum difference between the ground truth and the predicted temperature. ME is defined as $\max |y_i - \hat{y}_i|$, and it is important in thermal-aware scheduling. This is because the processors are forced to run at a low clock speed when the temperature reaches the overheating threshold. Even though most predictions are correct, a large error in the estimation may still cause thermal throttling. As a result, the ME should be as small as possible.

We compare the MSE and ME of both methods in Table 2 and plot the CDF of the prediction errors in these two methods in Figure 7.

As shown in the figure and table, the temperature prediction in HBS outperforms the linear regression algorithm due to the following reasons. The linear regression is based on the most recent temperature data, but the device temperature is not growing linearly since the scheduling algorithm keeps changing the processing speed and the processor running the computation. When a new scheduling decision is made, the device temperature may not keep increasing and decreasing linearly. As a result, the Android based approach underperforms.

6.3 Determining Threshold T_h in HBS

In the HBS algorithm, the threshold T_h is used to control the temperature of the device. To set the threshold properly, we use different thresholds T_h to evaluate the performance of the HBS. In the experiment, we test the HBS algorithm in the open space environment.

Figure 8 shows the performance of HBS under different thresholds T_h. As shown in the figures, the performance of the HBS algorithm is better when T_h becomes larger. This is because HBS can improve the utility by processing more frames on GPU and the device temperature can drop more quickly as T_h increases. When $T_h \geq 52°C$, the performance

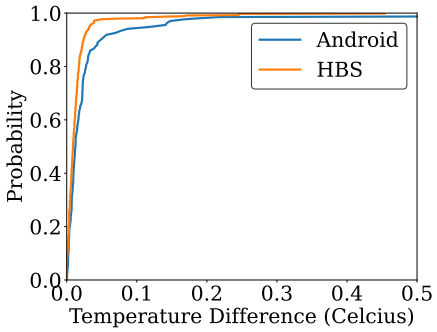
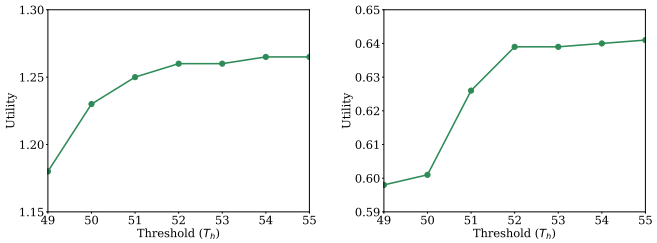


Fig. 7: The CDF of the temperature prediction errors in HBS and Android based approach.

of HBS only improves slightly as T_h increases. This is because the heat dissipation rate is almost the same when $T_h \geq 52°C$ and HBS makes similar scheduling decisions.



(a) GoogleNet (b) SqueezeNet

Fig. 8: The performance of HBS under different thresholds T_h

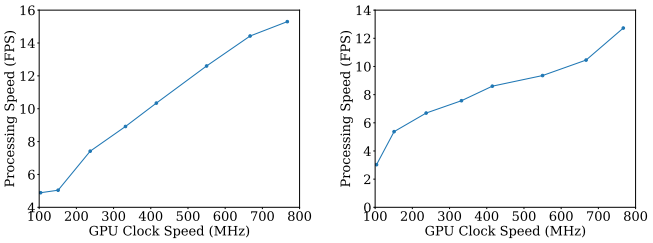
Based on the experiment results, we set T_h to be 52 in the experiment for the following reasons. First, the performance of HBS with $T_h = 52°C$ can achieve high utility (only 1% lower than the maximum utility in the experiments). Moreover, in the real world, the smartphone may run other programs in the background. Setting a higher threshold will leave HBS with less chance to adjust the device temperature and increase the risk of overheating. Therefore, we set T_h to be $52°C$ such that HBS can achieve high performance and minimize the risk of overheating.

6.4 Comparisons of Different Algorithms

Since SqueezeNet and GoogleNet have different model structure, their processing time is different. Based on their processing time limitations, we set the maximum processing speed for GoogleNet and SqueezeNet to be 10 fps and 15 fps, respectively. The running time of HBS and DRLS is about 1ms, and they only run at the beginning of each time interval which is about 1 second long. As a result, the scheduling overhead is negligible.

To evaluate the performance of our algorithms, we compare them with the following methods, *All-GPU*, *All-NPU* and *GPU-DVFS*.

All-GPU: All frames are processed on GPU. There is no scheduling algorithm used to control the temperature, and the frames are processed at the highest speed. The All-GPU method does not control the device temperature and



(a) GoogleNet (b) SqueezeNet

Fig. 9: The GPU processing speed in relation to different clock speeds.

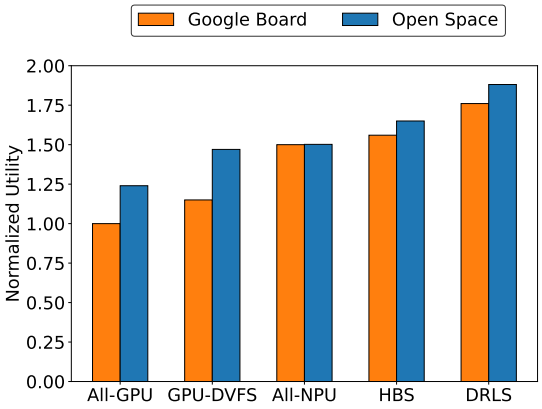
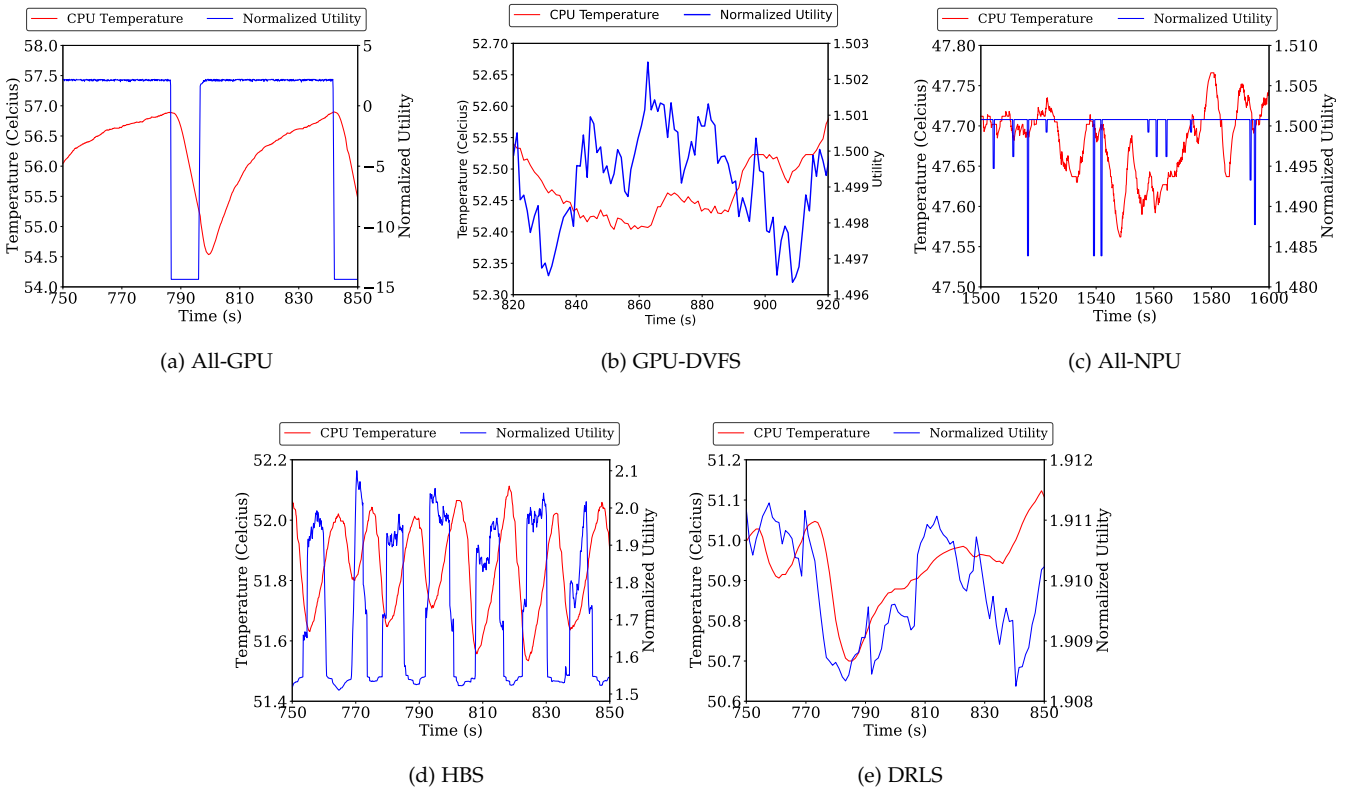


Fig. 10: The performance comparison of different approaches using GoogleNet.

the smartphone becomes overheated after some period of time. Thus, we set the utility to be -5 as penalty when the smartphone is overheated.

All-NPU: All the frames are processed on NPU. No scheduling algorithm is used to control the temperature of the device, and the frames are processed at the highest speed on NPU.

GPU-DVFS: Dynamic voltage and frequency scaling (DVFS) [19], [27]–[29] is a widely used power management technique to save energy and it also be used to reduce the amount of heat generated to avoid overheating. To apply DVFS techniques on mobile devices, root permission is needed to change the GPU frequency setting. On the Huawei Mate 10 Pro, the GPU clock speed can be adjusted to eight different frequencies, ranging from 103MHz to 767MHz. The GPU frequency is controlled by the gpu_scene_aware governor, which is used by the system and will be changed over time. To manually set a certain frequency and avoid the intervene from the system governor, a background process is created to overwrite a user-define frequency into the max_freq and min_freq files in the system folder. This background process will overwrite the setting every 200ms. The running time of this task is less than 2ms and is negligible compared to the computationally intensive tasks executed on the GPU. Figure 9 shows the processing time of running different DNN models using different GPU clock speed. As shown in the figure, the processing speed increases when the GPU frequency



(a) All-GPU (b) GPU-DVFS (c) All-NPU

(d) HBS (e) DRLS

Fig. 11: Detailed performance comparisons under open space environment when running GoogleNet.

increases. In the evaluation, the GPU-DVFS method utilizes a beam search to identify the optimal GPU frequency and processing speed combination to maximize utilization while avoiding overheating.

In Figure 10, we compare our algorithms with All-GPU, GPU-DVPS, and All-NPU using GoogleNet in different environments. In the evaluation, the parameter α is set to be 0.05. The All-GPU method has the worst performance among all methods. The reason can be better explained with Figures 11 and 12, which shows the temperature and normalized utility of different methods. As shown in the figure, the smartphone overheats only in the All-GPU method. Due to the large penalty for overheating, the utility of All-GPU is low. Compared to the open space environment, the utility is lower when the smartphone is placed in the VR headset. This is because the smartphone is wrapped by the card box and it is more difficult to dissipate the heat. The smartphone overheats more frequently and the processors are forced to work at a low clock speed for longer time. As a result, the utility is lower.

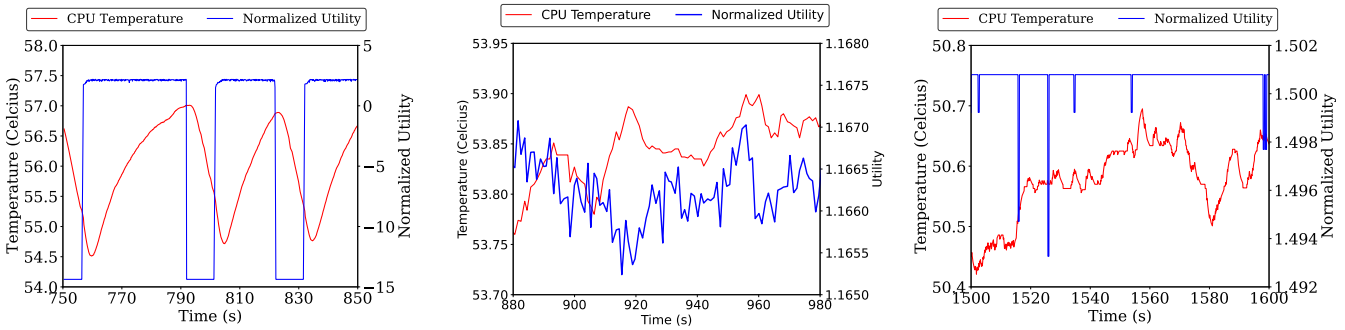
Compared to the All-GPU method, the GPU-DVFS technique has better performance. This can be explained by the Figures 11 and 12. As shown in the figures, the temperature of GPU-DVFS method remains stable and does not reach the overheating threshold. Therefore, the GPU-DVFS method does not have overheating penalty and it outperforms All-GPU significantly.

The performance of All-NPU remains the same under different environments. This is because running DNN on NPU generates much less heat than GPU and the smartphone does not overheat. All-NPU keeps running the DNN with the maximum processing speed on NPU. Compared to Figure 11(c), the CPU temperature in Figure 12(c) is higher. This is because it is more difficult to dissipate heat in the VR headset.

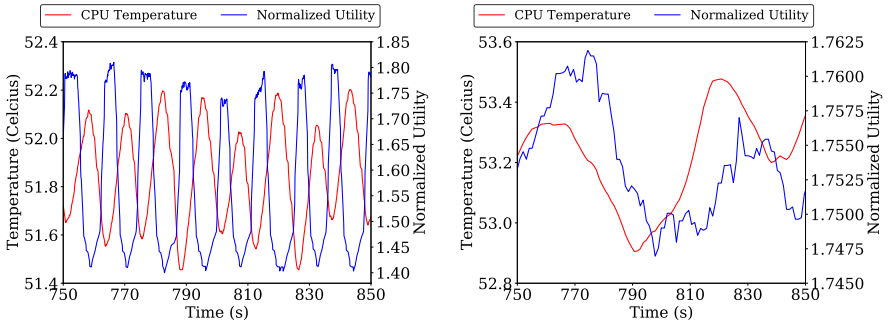
The performance of GPU-DVFS and All-NPU is nearly identical in an open space environment, but the performance gap widens when the mobile device is placed within VR headsets. This is because the energy consumption of running DNNs on GPU is much larger than that of NPU. In open space, the device can dissipate heat more efficiently and GPU-DVFS can achieve better performance by processing more frames. However, when the device is placed within a VR headset, the processing speed of GPU-DVFS has to be reduced and the utilization becomes lower. In contrast, since NPU generates much less heat than GPU, the performance of All-NPU remains the same.

Compared to All-NPU, HBS and DRLS also consider running DNNs on GPU in addition to NPU, and then their performance is better. Moreover, DRLS outperforms HBS in both environments due to the following reasons. The HBS algorithm focuses on maximizeing utility for the current time interval without considering the future impact of the scheduling decision. In contrast, DRLS considers the future impact of the scheduling decision. As shown in Figure 11(e) and Figure 12(e), DRLS does not try to achieve the best performance for every time interval. Instead, it tries to achieve better performance for all time intervals and thus its average utility is higher.

Compared to HBS and DRLS, the CPU temperature of



(a) All-GPU (b) GPU-DVFS (c) All-NPU



(d) HBS (e) DRLS

Fig. 12: Detailed performance comparisons under VR headsets environment when running GoogleNet.

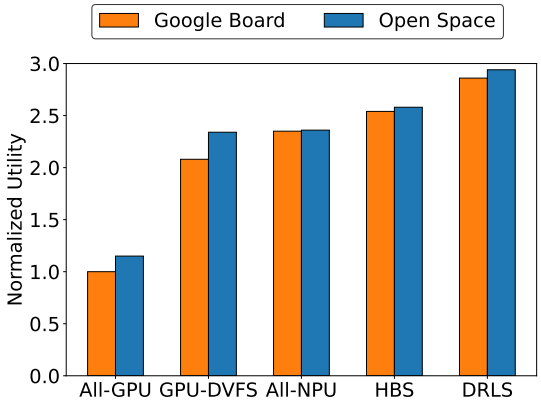


Fig. 13: The performance comparison of different approaches using SqueezeNet.

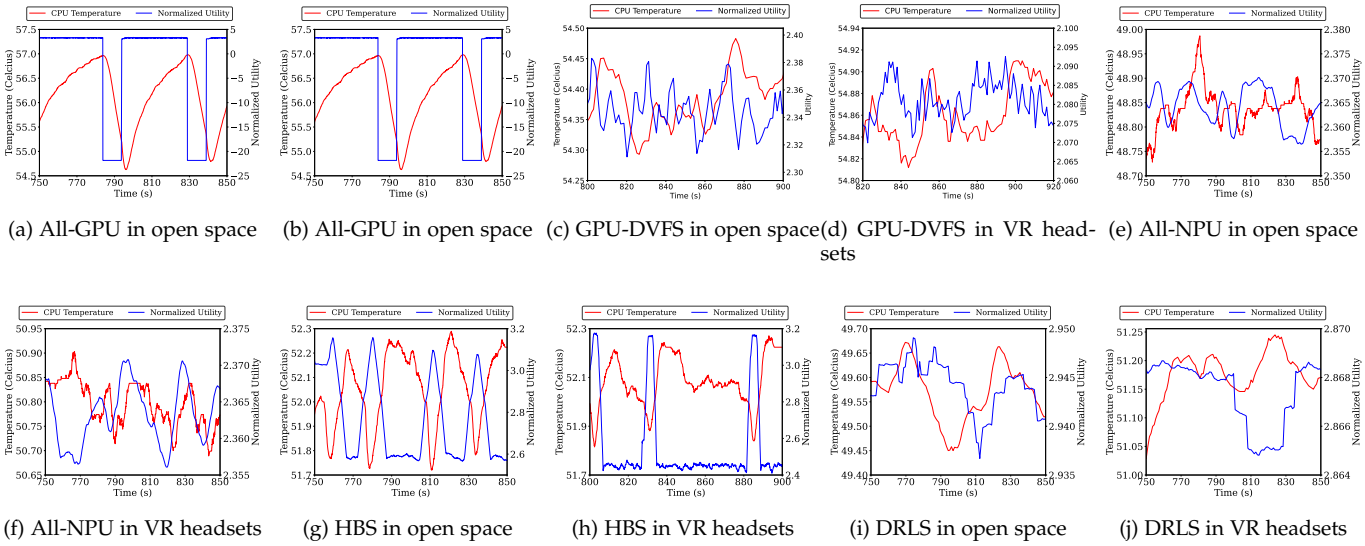
GPU-DVFS is much higher and is close to the overheating threshold T_o. This is because HBS and DRLS avoid overheating by controlling the device temperature around T_h, which is equal to $52°$ in the experiments and is smaller than T_o. Therefore, GPU-DVFS method is less robust and the risk of overheating will increase significantly when the device temperature is changed by other factors (e.g., execution of other background programs or environment temperature changes).

We also use SqueezeNet to evaluate the performance of All-GPU, GPU-DVFS, All-NPU, HBS and DRLS under different environments. In the evaluation, the parameter α is set to be 0.01 and the result is shown in Figure 13. Compared to GoogleNet, SqueezeNet is simpler and can be run faster. Since the maximum processing speed is increased to 15 fps, more frames need to be processed on GPU and the smartphone still gets overheated in All-GPU. As a result, All-GPU has the worst performance among all methods. As shown in Figure 14(c) and (d), the performance of GPU-DVFS is much better than that of All-GPU. This is because the temperature of GPU-DVFS remains below the overheating threshold, leading to enhanced utility by avoiding overheating penalties.

All-NPU outperforms All-GPU, and its performance remains the same under different environments. This can be better explained by Figures 14(e) and (f). As shown in the figures, the CPU temperature is stable and below the overheating threshold. Different from ALL-GPU, All-NPU has the same utility in both environments since frames can be processed at the highest speed and there is no overheating. Since All-NPU does not leverage GPU to further improve performance, it under-performs HBS and DRLS.

All-NPU also outperforms GPU-DVFS and the performance difference is larger when the device is placed within a VR headset. This is because the energy consumption of GPU is much higher than that of NPU. When the device is placed within a VR headset, the GPU processing speed of GPU-DVFS has to be reduced to avoid overheating. However, the All-NPU performance remains the same in this environment since the NPU generates much less heat than GPU.

As shown in Figures 14(g)-(j), although the peak perfor-



(a) All-GPU in open space (b) All-GPU in open space (c) GPU-DVFS in open space (d) GPU-DVFS in VR headsets (e) All-NPU in open space

(f) All-NPU in VR headsets (g) HBS in open space (h) HBS in VR headsets (i) DRLS in open space (j) DRLS in VR headsets

Fig. 14: Detailed performance comparisons under different environments when running SqueezeNet.

mance of HBS is higher than DRLS in some time intervals, HBS under-performs DRLS considering all time intervals. This is because HBS makes scheduling decisions to maximize utility for the current time interval, without taking into account its future impacts. In contrast, DRLS considers the future impact of a scheduling decision and tends to make decisions that are beneficial across all time intervals. As shown in Figures 14(i) and (j), DRLS maximizes utility by controlling the phone temperature at various levels based on the environmental conditions.

7 Related Work

Recently, there has been considerable research progress on object recognition using DNN models [30], [31]. However, these research results rely on machines with powerful GPU, and most of them cannot be applied to mobile devices with limited computation capability. Offloading techniques have been proposed to address this problem, where computational intensive tasks are offloaded to the cloud or edge server. MAUI [32] and many other research [33], [34] provide general offloading techniques to optimize energy or reduce the computation time for mobile applications. To further reduce the amount of data to be offloaded, some local processing techniques have been proposed to filter out less important or redundant data [5], [35]. Other research focuses on selecting different DNN models to run locally to satisfy the time constraints based on the network conditions [36]–[39].

Researchers also address the resource limitations of mobile devices through model compression and hardware support. For example, in [40], [41], convolutional layers and fully-connected layers are compressed to reduce the processing time of DNN models. In [42], the authors compress DNN models by optimizing the neural network configuration. Although these model compression techniques can be applied to improve efficiency, the accuracy also drops. As another approach, researchers leverage hardware techniques or model partition techniques to improve the execution efficiency of running DNN models on mobile devices. For example, in [43]–[45], the authors developed various techniques to leverage the mobile GPU to improve performance. In [17], Tan *et al.* have developed model partitioning techniques to schedule some neural network layers on CPU while executing other layers on NPU to achieve better tradeoffs between processing time and accuracy. This model partition technique can be applied to our problem, but there are many challenges to address. Since the problem here is about avoiding overheating and improving accuracy, the problem formation will be different. Moreover, to apply model partition, we also need to consider the extra data transmission time between NPU memory and the main memory.

There exists a large amount of research on thermal management for multi-core processors based on Dynamic Voltage and Frequency Scaling (DVFS) and task migration techniques [19], [27]–[29], [46]. Researchers have studied this problem on heterogeneous processors [47]–[51], and have also studied how to apply these techniques to mobile devices to achieve better tradeoffs between performance and energy under thermal constraints [52], [53]. In recent years, deep reinforcement learning (deep RL) has been applied to many decision-making tasks in various fields and some researchers have applied it to thermal management on mobile devices. For example, Kim *et al.* [19] proposed a deep RL based algorithm to maximize the QoE for mobile devices under different environments. However, none of them considers the special characteristics of NPU, which is the focus of this paper.

8 Conclusions

In this paper, we proposed thermal-aware scheduling algorithms to improve the performance of running DNNs on mobile devices. By measuring the performance of running DNNs on GPU and NPU, we identified the performance tradeoffs between GPU-based approach and NPU-based approach. The GPU-based approach has higher accuracy,

but also higher energy consumption. Running DNNs on GPU continuously will cause the mobile device to overheat and result in poor performance. On the other hand, NPU-based approach is faster and more energy efficient, but with lower accuracy. We proposed to combine these two approaches by studying the thermal-aware scheduling problem, where the goal is to achieve a better tradeoff between processing speed and accuracy for DNN based applications while ensuring that the mobile device does not overheat. The major challenge is to determine when to run DNNs on GPU to achieve better performance and when to run DNNs on NPU to avoid overheating. We first proposed a Heuristic-Based Scheduling (HBS) algorithm to solve the problem. HBS predicts the temperature based on historical data and makes scheduling decisions based on the current state of the mobile device. Since HBS ignores the future impact of a scheduling decision, we proposed Deep Reinforcement Learning Scheduling (DRLS) algorithm to further improve the performance. We have implemented our algorithms on smartphones and evaluation results show that the proposed algorithms can significantly improve the performance of running DNNs on mobile devices.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants 2125208 and 2215043.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems(NIPS)*, 2017.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *IEEE International Conference on Computer Vision (ICCV)*, 2015.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[4] J. Park, Y. Boo, I. Choi, S. Shin, and W. Sung. Fully Neural Network Based Speech Recognition on Mobile and Embedded Devices. *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[5] K. Chen, T. Li, H. Kim, D. Culler, and R. Katz. MARVEL: Enabling Mobile Augmented Reality with Low Energy and Low Latency. *ACM Sensys*, 2018.

[6] J. Hernandez, D. McDuff and R. Picard. Biowatch: Estimation of heart and breathing rates from wrist motions. *IEEE International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth)*, 2015.

[7] HUAWEI. Kirin 990. https://consumer.huawei.com/en/campaign/kirin-990-series/.

[8] Samsung. Samsung Electronics to Strengthen its Neural Processing Capabilities for Future AI Applications. https://news.samsung.com/global/samsung-electronics-to-strengthen-its-neural-processing-capabilities-for-future-ai-applications.

[9] Qualcomm. Snapdragon 855. https://www.qualcomm.com/products/snapdragon-855-mobile-platform.

[10] T. Tan and G. Cao. FastVA: Deep Learning Video Analytics Through Edge Processing and NPU in Mobile. *IEEE INFOCOM*, 2020.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, P. Stig, B. Charles, S. Amir, A. Ioannis, K. Helen, K. Dharshan, W. Daan, L. Shane, and H. Demis. Human-Level Control through Deep Reinforcement Learning. *nature*, 2015.

[12] O. M. Parkhi, A. Vedaldi, A. Zisserman. Deep Face Recognition. *British Machine Vision Conference*, 2015.

[13] F. Iandola, and S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer. SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and 0.5 MB Model Size. *arXiv preprint arXiv:1602.07360*, 2016.

[14] Y. Jiang, Z. Wu, J, Wang, X. Xue and S. Chang. Exploiting Feature and Class Relationships in Video Categorization with Regularized Deep Neural Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[16] T. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár and C. Zitnick. Microsoft COCO: Common Objects in Context. *European Conference on Computer Vision (ECCV)*, 2014.

[17] T. Tan and G. Cao. Efficient Execution of Deep Neural Networks on Mobile Devices with NPU. *ACM IPSN*, 2021.

[18] HUAWEI. HiAI. https://developer.huawei.com/consumer/en/devservice/doc/2020315.

[19] S. Kim, K. Bin, S. Ha, K. Lee, and S. Chong. zTT: Learning-based DVFS with Zero Thermal Throttling for Mobile Devices. *ACM Mobisys*, 2021.

[20] Ganapati Bhat, Gaurav Singla, Ali K Unver, and Umit Y Ogras. Algorithmic optimization of thermal and power management for heterogeneous mobile platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[21] S. Wang, G. Ananthanarayanan, and T. Mitra. OPTiC: Optimizing Collaborative CPU–GPU Computing on Mobile Devices with Thermal Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[22] Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 1957.

[23] ARM. big. little technology: The future of mobile.

[24] Google. Tensorflow. https://www.tensorflow.org/.

[25] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *ACM International Conference on Multimedia*, 2014.

[26] Konduit K.K. Deeplearning4j. https://deeplearning4j.konduit.ai/.

[27] A. Iranfar, M. Zapater, and D. Atienza. Machine Learning-Based Quality-Aware Power and Thermal Management of Multistream HEVC Encoding on Multicore Servers. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[28] C. Lin, K. Wang, Z. Li, and Y. Pu. A Workload-Aware DVFS Robust to Concurrent Tasks for Mobile Devices. *ACM Mobicom*, 2023.

[29] R. Chen, Q. Wan, X. Zhang, X. Qin, Y. Hou, D. Wang, X. Fu, and M. Pan. EEFL: High-Speed Wireless Communications Inspired Energy Efficient Federated Learning over Mobile Devices. *ACM Mobisys*, 2023.

[30] G. Huang, Z. Liu, L. Van Der Maaten, and K. Weinberger. Densely Connected Convolutional Networks. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[31] J. Zhang, Z. Tang, M. Li, D. Fang, P. Nurmi, and Z. Wang. CrossSense: Towards Cross-Site and Large-Scale WiFi Sensing. *ACM Mobicom*, 2018.

[32] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. *ACM Int'l Conf. on Mobile Systems Applications and Services (MobiSys)*, 2010.

[33] I. Zhang, A. Szekeres, A. Van, I. Ackerman, S. Gribble, A. Krishnamurthy, and H. Levy. Customizable and Extensible Deployment for Mobile/Cloud Applications. *ACM USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[34] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao. Energy-Efficient Computation Offloading in Cellular Networks. *IEEE International Conference on Network Protocols (ICNP)*, 2015.

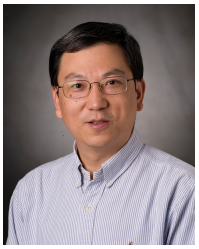
[35] Y. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. *ACM Sensys*, 2015.

[36] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics. *IEEE INFOCOM*, 2018.

[37] T. Tan and G. Cao. Deep Learning on Mobile Devices Through Neural Processing Units and Edge Computing. *IEEE INFOCOM*, 2022.

[38] Y. Kim J. Lim. Real-time DNN Model Partitioning for QoE Enhancement in Mobile Vision Applications. *IEEE SECON*, 2022.
[39] T. Tan and G. Cao. Deep Learning Video Analytics Through Edge Computing and Neural Processing Units on Mobile Devices. *IEEE Transactions on Mobile Computing*, 2023.
[40] S. Bhattacharya and N. Lane. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. *ACM Sensys*, 2016.
[41] B. Fang, X. Zeng, and M. Zhang. Nestdnn: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. *ACM Mobicom*, 2018.
[42] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, Lu and T. Abdelzaher. FastDeepIoT: Towards Understanding and Optimizing Neural Network Execution Time on Mobile and Embedded Devices. *ACM Sensys*, 2018.
[43] L. Oskouei, S. Salar, H. Golestani, M. Hashemi and S. Ghiasi. CNNdroid: GPU-Accelerated Execution of Trained Deep Convolutional Neural Networks on Android. *ACM International Conference on Multimedia*, 2016.
[44] N. 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. *IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2016.
[45] L. Huynh, Y. Lee, and R. Balan. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. *ACM International Conference on Mobile Systems Applications and Services (MobiSys)*, 2017.
[46] Y. Lee, H. Chwa, K. Shin, and S. Wang. Thermal-Aware Resource Management for Embedded Real-time Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
[47] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.
[48] M. Cho, W. Song, S. Yalamanchili, and S. Mukhopadhyay. Thermal System Identification (TSI): A Methodology for Post-Silicon Characterization and Trediction of the Transient Thermal Field in Multicore Chips. *2012 28th Annual IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*, 2012.
[49] G. Singla, G. Kaur, A. Unver, and U. Ogras. Predictive Dynamic Thermal and Power Management for Heterogeneous Mobile Platforms. *IEEE Design Automation and Test in Europe*, 2015.
[50] D. Lee, S. Das, J. Doppa, P. Pande, and K. Chakrabarty. Performance and thermal tradeoffs for energy-efficient monolithic 3d network-on-chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2018.
[51] J. Jeong, J. Lee, D. Kim, C. Jeon, C. Jeong, Y. Lee, and B. Chun. Band: Coordinated Multi-DNN Inference on Heterogeneous Mobile Processors. *ACM Mobisys*, 2022.
[52] s. Mandal, G. Bhat, c. Patil, J. Doppa, P. Pande, and U. Ogras. Dynamic Resource Management of Heterogeneous Mobile Platforms via Limitation Learning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.
[53] J. Haj-Yahya, M. Alser, J. Kim, G. Yağlıkçı, N. Vijaykumar, E. Rotem, and O. Mutlu. SysScale: Exploiting Multi-Domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors. *ACM/IEEE ISCA*, 2020.

Tianxiang Tan received the BE degree from Sun Yat-sen University, the MS degree in computer science from University of Southern California, and the PhD degree in computer science from the Pennsylvania State University. His research interests include mobile cloud computing, edge computing and deep learning. He is a student member of the IEEE.



Guohong Cao received his B.S. degree in computer science from Xi'an Jiaotong University, and his Ph.D. in computer science from the Ohio State University in 1999. Since then, he has been with the Department of Computer Science and Engineering at the Pennsylvania State University, where he is currently a Distinguished Professor. He has published more than 250 papers in the areas of wireless networks, mobile computing, machine learning, wireless security and privacy, and Internet of Things, which have been cited over 25000 times. He has served on the editorial board of IEEE Transactions on Mobile Computing, IEEE Transactions on Wireless Communications, and IEEE Transactions on Vehicular Technology, and has served on the organizing and technical program committees of many conferences, including the TPC Chair/Co-Chair of IEEE SRDS, MASS, and INFOCOM. He has received several best paper awards, the IEEE INFOCOM Test of Time award, and the NSF CAREER award. He is a Fellow of the AAAS and a Fellow of the IEEE.