DynCNN: Application Dynamism and Ambient Temperature Aware Neural Network Scheduler in Edge Devices for Traffic Control

OMAIS SHAFI, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India
SACHIN KUMAR CHAUHAN, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India
GAYATHRI ANANTHANARAYANAN, Department of Computer Science and Engineering
Indian Institute of Technology, Dharwad, Karnataka, India
RIJUREKHA SEN, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India

Road traffic congestion increases vehicular emissions and air pollution. Traffic rule violation causes road accidents. Both pollution and accidents take tremendous social and economic toll worldwide, and more so in developing countries where the skewed vehicle to road infrastructure ratio amplifies the problems. Automating traffic intersection management to detect and penalize traffic rule violations and reduce traffic congestion, is the focus of this paper, using state-of-the-art Convolutional Neural Network (CNN) on traffic camera feeds. There are however non-trivial challenges in handling the chaotic, non-laned traffic scenes in developing countries. Maintaining high throughput is one of the challenges, as broadband connectivity to remote GPU servers is absent in developing countries, and embedded GPU platforms on roads need to be low cost due to budget constraints. Additionally, ambient temperatures in developing country cities can go to 45-50 degree Celsius in summer, where continuous embedded processing can lead to lower lifetimes of the embedded platforms. In this paper, we present DynCNN, an application dynamism and ambient temperature aware controller for Neural Network concurrency. DynCNN effectively uses processor heterogeneity to control the number of threads and frequencies on the accelerator to manage application utility under strict thermal and power thresholds. We evaluate the efficiency of DynCNN on three different commercially available embedded GPUs (Jetson TX2^{TM}, Xavier NX^{TM} and Xavier AGX^{TM}) using a real traffic intersection's 40 days' dataset. Experimental results show that in comparison to all existing state-of-the art- GPU governors for two different CPU settings, DynCNN reduces the average temperature and power by ~12°C and 68.82% respectively for one CPU setting (Baseline1) and similarly, it improves the performance by around 31.2% compared to the other CPU setting (Baseline2).

ACM Reference Format:
Omais Shafi, Sachin Kumar Chauhan, Gayathri Ananthanarayanan, and Rijurekha Sen. 2022. DynCNN: Application Dynamism and Ambient Temperature Aware Neural Network Scheduler in Edge Devices for Traffic Control. In *ACM SIGCAS/SIGCHI Conference on Computing and Sustainable Societies (COMPASS) (COMPASS '22), June 29-July 1, 2022, Seattle, WA, USA*. ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3530190.3534823

1 Introduction

With 1.25 million people killed on the world's roads each year and another 20-50 million seriously injured [9], road traffic injuries currently inflict social and economic losses of epidemic proportions. This is especially true in developing

countries, where the road-to-vehicles ratio is so skewed that drivers resort to non-laned chaotic driving, regularly flouting traffic rules to dangerously move ahead of each other. In addition to accidents, the road-to-vehicles skewed ratio also causes daily traffic congestion, that in turn frustrates drivers, increasing their propensity to reckless driving. Congestion also increases vehicular emissions and air pollution, another health hazard of epidemic scale in developing countries [3, 7].

Given the bleak situation, authorities in developing countries are trying to adopt automated sensing and computations, to augment their current traffic police based manual intersection management and traffic rule enforcement systems [36]. While over-hauling drivers' nonchalance towards rules is difficult, a somewhat practical goal is to instill fear of economic penalties and enforce more discipline. In this paper, we work with such a traffic management urban agency, in the Indian capital city of Delhi. High FPS and high resolution intersection cameras, Convolutional Neural Networks (CNN) for real-time video processing, Reinforcement Learning (RL) based intersection control etc. are fancy terms we could propose to them, as positive examples from the developed countries and recent academic literature [2, 35]. But our close collaboration reveals multiple real constraints such deployments in developing countries have, that need to be examined carefully before automated road traffic management becomes practical.

Budget is the first constraint. Lack of reliable broadband connectivity from road to remote GPU servers is the second constraint. These two constraints call for using embedded platforms (costing as low as possible) to process camera frames in the road itself. In addition, ambient temperatures in our envisioned deployment scenarios can rise up to 45-50 degree Celsius in summer. An on-road deployed embedded platform will not have air-conditioning, and therefore the platform's life-time needs to be maximized with effective thermal control. Edge computing devices show a great promise in supporting state-of-the-art neural network based applications. Traditional hardware accelerators like Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA), and custom accelerators for neural networks like Tensor Processing Unit (TPU) or Vision Processing Unit (VPU) are now included in SoCs of many embedded devices [19–21].

While a single neural network inference seems easy now on the edge with the said hardware accelerator and software support, concurrently running multiple neural networks seems the next logical step for many edge applications. For example, in this paper, we deploy traffic cameras in a megacity of a developing country for traffic rule violation detection and penalization. Cameras look at vehicles from the back and the front to detect speed and signal violations. The purpose of the cameras is to detect rule violations, and also to detect and read the number plates to fine the rule violating vehicles. As shown in Section 3, one road approach needs six cameras in our deployment, overall comprising of 18-24 cameras for a 3-way or 4-way intersection. This number will go up if the road approaches are wider than in our setting. If all these camera feeds need to be processed using state of the art neural network inference models (e.g. CNN for vehicle and number plate detection), then neural network *concurrency* is a requirement for such an edge device.

Neural network concurrency at the edge is explored in Section 4 using both multi-processing and multi-threading, on real state-of-the-art edge computing platforms with NVIDIA GPU cores, namely NVIDIA Jetson TX2^{TM} [19], NVIDIA Xavier NX^{TM} [21] and NVIDIA Xavier AGX^{TM} [20]. We experiment with both large and small CNN models from standard neural network model zoo [1] and show the feasibility of running upto 36 concurrent neural networks on these embedded GPU platforms, before memory bandwidth starts becoming a bottleneck. Similar observations have been reported in [32]. However, the biggest challenge we observe to this tremendous promise of neural network concurrency is the increased temperature rise and power consumption on the platforms. We empirically demonstrate for each embedded GPU platform how increasing GPU thread count to handle more concurrent neural network inferences and boosting the GPU clock frequency for higher throughput in each concurrent inference task, affects the thermal (upto 25 degree Celsius temperature rise) and power (upto 10000 mW power rise) characteristics in Section 5. We additionally experiment at multiple ambient temperatures of 35 and 45 degree Celsius, and show how higher ambient temperatures causes the

absolute platform temperatures to rise even more and leads to device failures (rebooting arbitrarily or not powering up at all) after only 15-20 minutes of experiments.

Based on our empirical observations of running concurrent neural networks at different ambient temperatures, we finally present DynCNN. DynCNN handles the performance-thermal-power trade-off problem, using heterogeneous processor cores on edge devices. In heterogeneous SoCs, neural network accelerators like GPU, FPGA, TPU etc. co-exist with one or more CPU cores. DynCNN uses these CPU cores to continuously monitor the application state, and controls the accelerator's thread count and operating frequency, based on the application's current requirements. For example, DynCNN can run traffic density estimation using computer vision methods like Optical Flow and Background Subtraction on the CPU cores. Based on the traffic density seen by a particular traffic camera as measured by the CPU, DynCNN starts or stops the GPU thread handling that camera's input on our hardware platform and also sets the GPU frequency appropriately. We describe this concrete multi-threaded traffic monitoring application, with results from a deployed intersection in Section 3. As we empirically show with 40 days' of traffic data collected from our deployment, DynCNN always achieves better application performance at much lower thermal and power levels, compared to all existing GPU governors that either aggressively throttle and lose utility while maintaining thermal-power budget, or aggressively run the application at the highest possible frames per second (FPS) but reach unsafe temperature and power consumption levels. Exploiting processor core heterogeneity, DynCNN better controls accelerators like GPU using CPU, for more optimal performance-power-thermal trade-off control. Let us summarize the contributions of the paper.

❶ We show why concurrent CNNs are required in traffic monitoring for rule violation detection with the help of 40 days' data collected from our deployment at a busy traffic intersection Lajpatnagar, in New Delhi, India.

❷ We analyze the rise in temperature and power with the increase in the number of GPU threads and GPU frequencies, while computer vision tasks run on embedded GPU platforms. In addition, we also study the rise in platform temperature with the change in ambient temperatures.

❸ We design and implement an intersection controller, DynCNN, that dynamically adjusts the number of GPU threads and frequencies based on the traffic density. We show experimentally that DynCNN reduces the average temperature and power by ~12°C and 68.82% respectively for one CPU setting (Baseline1) and similarly, it improves the performance by around 31.2% compared to the other CPU setting (Baseline2).

The rest of the treatise is as follows. Section 2 discusses the related work, Section 3 describes our traffic-intersection application, neural network concurrency is presented in Section 4, Section 5 discusses the thermal and power constraints in neural network concurrency, Section 6 discusses and evaluates our DynCNN controller and finally we conclude in Section 8.

2 Related Work

Non-laned heterogeneous traffic in developing regions has excited the research community to design automated traffic monitoring systems using a wide variety of embedded sensors like cameras [26, 37, 38], microphones [27, 30] and RF [28, 29]. All these works are on congestion estimation, that output the level of traffic density or the length of vehicle queue on a given road stretch. Our work gives a superset of these outputs. We detect and classify vehicles for a given road stretch, which in summation can give the traffic density and queue length on the roads. We additionally output the location of each detected vehicle, which is necessary for traffic rule violation detection. In addition, the ambient temperatures in developing cities can go to around 45-50°C where continuous embedded processing can lead to lower lifetimes of the embedded platforms. Thus, in addition to efficient traffic management, we also need to maintain the temperature of the

setup deployed on the roads in developing regions. *To the best of our knowledge, we are the first to consider thermal control for efficient traffic management in developing regions.*

We now study the application aware controllers that have been proposed in the literature. Choi et al. [6] propose a CPU/GPU governor to optimise Android graphics pipeline and user QoE by changing maximum CPU frequency and minimum GPU frequency depending on frame load prediction. The CPU/GPU frequency capping method was proposed by Park et al. [23] to increase the energy efficiency of game applications. By training a large number of gaming applications, it creates a lookup table with CPU and GPU costs. Individual user profiles were employed by Yang et al. [39] to increase energy efficiency by managing CPU clocks and learning the user experience of an application. The authors in [15, 18] propose DVFS governors for web browsing. Chuang et.al [8] propose an online adaptive CPU-GPU governor for mobile devices for minimizing the energy consumption. Similarly, Hsieh et.al [13] propose a memory-aware cooperative CPU-GPU governor to maximise the energy efficiency of high-end mobile game workloads. ML-Gov [22] propose a a machine learning enhanced integrated CPU-GPU governor that builds tree-based piecewise linear models offline, and deploys these models for online estimation into an integrated CPU-GPU Dynamic Voltage Frequency Scaling (DVFS) governor. By constructing a model based on extensive observations to estimate the overall power budget within the temperature threshold, Bhat et al. [4] and Gupta et al. [11] proposed a predictive thermal and power management system. Sahin et al. [31] divided applications into two categories: throttling-prone continuous calculations and latency-sensitive bursty workloads. To reduce thermal throttling length, they use a separate policy based on the power profile for different types of workloads. Similarly, Sharifi et al. [33] built a predictive task scheduler using the physical information of CPUs to build a thermal model.

Recent literature has shifted the paradigm to the use of Reinforcement Learning to solve such problems. Gupta et al. [12] proposed a Deep Q-learning framework for runtime power management optimization under dynamically changing workloads. [34] introduced an online power management technique based on Q-learning that does not require any prior workload knowledge. Iranfar et al. [14] proposed a Q-learning based power and thermal management algorithm for frequency scaling and thread allocation by limiting state-action space. The recent work by [16] propose a deep reinforcement learning based technique to achieve maximum performance while ensuring zero thermal throttling. In this paper, we propose an application aware based controller with the use case as traffic violation which to the best of our knowledge has not been considered before. Moreover, the recent works in the literature use the RL based techniques to balance the performance and the temperature. However, these RL techniques add a significant amount of computational load at the edge on top of the application. *In comparison, our* `DynCNN`*, a heuristic based approach is a simplistic design with the least requirement of system resources and it outperforms the most advanced DVFS schemes from the Linux kernel.*

3 Developing Region Traffic Monitoring Case-study

We partner with an urban municipality in the Indian capital city of Delhi and deploy traffic cameras at real intersections. We build computer vision based methods to continuously gauge traffic densities using embedded CPUs to process the camera inputs, and subsequently analyze the application dynamism over an extended period of 40 days at one particular intersection. This section describes this application and our deployment, and how it drives various data driven empirical design choices for `DynCNN`.

3.1 Concurrent CNNs for traffic monitoring and rule violation detection

Figure 1 shows a typical four-way intersection with four road approaches, common in developing countries in South Asia, with left-side driving of vehicles (shown with red arrows on the roads). In multiple such intersections, our deployment partners installed two rule violation detection cameras, per road approach of the intersection. As seen from Figure 1, the first camera CAM1, looks away from the intersection, towards incoming vehicles coming towards the intersection. Vehicles are detected at two fixed locations along the road, at a gap of distance d, at times t_1 and t_2. **Speed violation** occurs, if the vehicle speed = $\frac{d}{t_2 - t_1}$ exceeds some threshold. The second camera CAM2, looks towards the intersection, towards outgoing vehicles crossing the intersection. **Signal violation** occurs, if the current traffic signal for an approach is red, and vehicles are detected in that approach on or beyond the zebra crossing. For each rule violation camera, there are two Automatic Number Plate Recognition (ANPR) cameras, which detect and read number plates to automatically issue fines, for vehicles violating rules. ANPR cameras are double the number of rule violation cameras, as number plates are smaller than cars. Each ANPR camera, therefore, needs to focus better on a smaller road region, to reduce false negatives in number plate detection and improve Optical Character Recognition (OCR) accuracy.

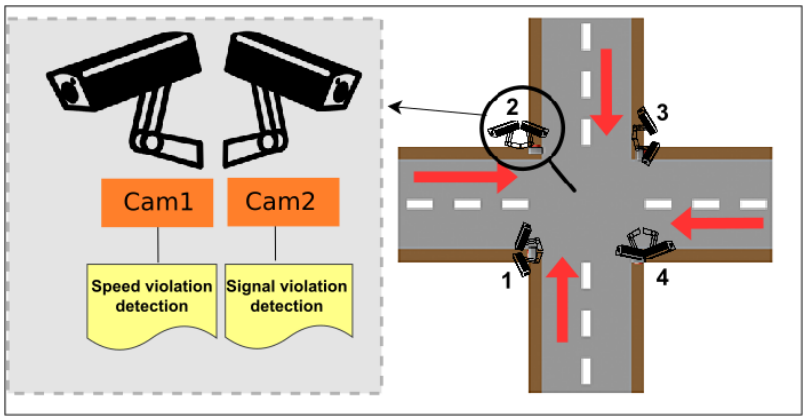


Fig. 1. A Typical 4-way intersection

Table 1 shows for each approach road, the number of cameras for different purposes. As object detection (either vehicle or number plate) forms a core part of the work, CNNs can be very useful because of their reported high accuracy in object detection tasks [5, 10, 17]. The challenge here is that each approach has 6 cameras requiring 6 CNN tasks, which together creates 18 and 24 CNN tasks for three-way and four-way intersections respectively. While Section 4 shows these high number of concurrent CNN threads can be supported in state-of-the-art edge computing devices with embedded GPUs, Section 5 shows the thermal and power overheads of running such threads at high GPU frequencies all the time. We therefore examine this application in more detail, to see if application specific dynamisms can be used for better GPU concurrency control under thermal and power constraints.

Purpose	Cameras	CNN task
Detect speed violation	1	Vehicle detection, tracking
Detect signal violation	1	Vehicle detection
ANPR	4	Number plate detection, OCR

Table 1. 6 CNN tasks per approach at an intersection, adding up to 18 or 24 CNN tasks for a 3-way or 4-way intersection

To examine whether road traffic in these intersections have enough dynamism or not that might be relevant for GPU concurrency control, we devise a mechanism of traffic density estimation. Being non-CNN based computer vision algorithms, this density estimation can run on CPU cores and do not need GPUs. It also handles non-laned chaotic roads of developing regions. If the traffic densities vary considerably to aid application aware thermal and power control for the GPU, the CPU can execute such intelligent control by running the density estimation algorithm and changing the GPU threads and frequencies as needed. Current heterogeneous edge computing devices like Jetson TX2 or Xavier AGX and NX have quad-core or octa-core ARM CPUs in addition to the NVIDIA embedded GPUs. This CPU based application state monitoring for GPU concurrency control on edge devices with heterogeneous processor cores, forms the crux of `DynCNN`. We next describe our CPU based density estimation algorithm and demonstrate its output on real camera data from our deployment.

3.2 Traffic density estimation on embedded CPU cores

To estimate traffic density, a background filter is subtracted from each camera frame, to compute the foreground, and *foreground/background* indicates queue density. The filter is periodically updated, the period τ denoting the learning rate. Such updates ensure that changing lighting conditions over the day, shadows etc. are correctly incorporated in the background filter. Background subtraction based density estimates comprise both standing and moving traffic. We additionally use an optical flow algorithm, to detect moving pixels between frames, and use that to compute dynamic traffic density i.e. density of only vehicles that are moving. Table 2 summarizes the CPU based density estimation tasks, each of which need to process CAM1 and CAM2 video feeds, as they together cover the span of a given approach. Thus there are 2 cameras per approach, on which these non-CNN tasks have to run, together requiring 6 to 8 cameras for three-way and four-way intersections respectively.

Purpose	Cameras	Computer Vision task
Queue density estimation	2	Background subtraction
Dynamic density estimation	2	Optical flow

Table 2. Traffic density estimation at an intersection that can execute on CPU

Figure 2 shows sample congested images from CAM1 and CAM2 on the left, at one approach in one of the intersections where we deploy and examine our traffic monitoring system. The right side shows: (a) background subtraction based density (queue density in blue curve) and (b) optical flow based density (dynamic density in orange curve), over a span of over 10 minutes. As seen from the graph, queue density starts to rise as vehicles start accumulating when signal turns red (indicated by vertical red lines), and starts to fall as vehicles start to clear when signal turns green (indicated by vertical green lines). Dynamic density is zero when red signal is on as all vehicles are static (between red and green vertical lines) and rises when signal turns green and vehicles start moving. CAM1 also shows similar patterns, but with less periods of high queue density than CAM2. CAM1 is monitoring the tail of the incoming queue while CAM2 monitors the queue head. The head gets filled earlier and therefore shows higher density. CAM1 also has more dynamic density even when signal is red for the same reason. The tail of the queue monitored by it keeps growing with incoming vehicles, even after signal is red.

3.3 Long term traffic density monitoring to understand application dynamism

Even within a short span of 10 minutes in Figure 2, the traffic shows some changing behaviors based on the traffic signal state and incoming vehicle load. We next check these application level dynamisms over a longer span of 40 days at our

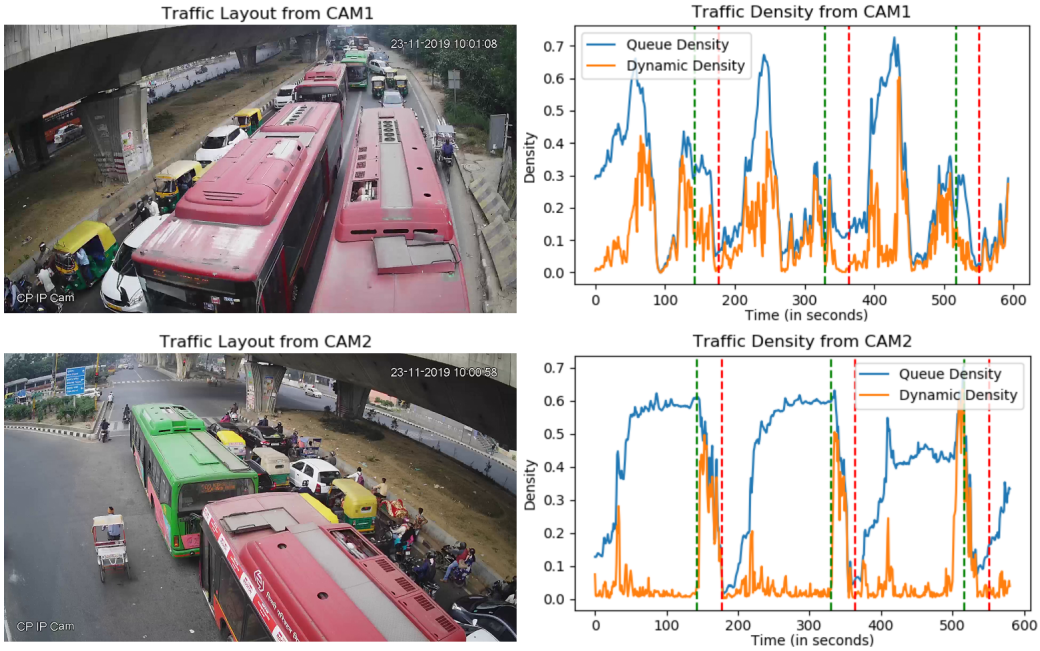


Fig. 2. View from traffic cameras on the left and traffic densities measured on road deployed CPU on the right

deployment, to understand how such dynamisms might be useful for GPU concurrency and frequency control for power and thermal benefits. Figure 3 shows box plots for traffic queue densities across all three road approaches over 40 days at our deployment intersection. The horizontal axis denotes the hour of the day and the vertical axis denotes the traffic queue density. We observe from the graph that for a particular hour along the x-axis, there is non-negligible box height along

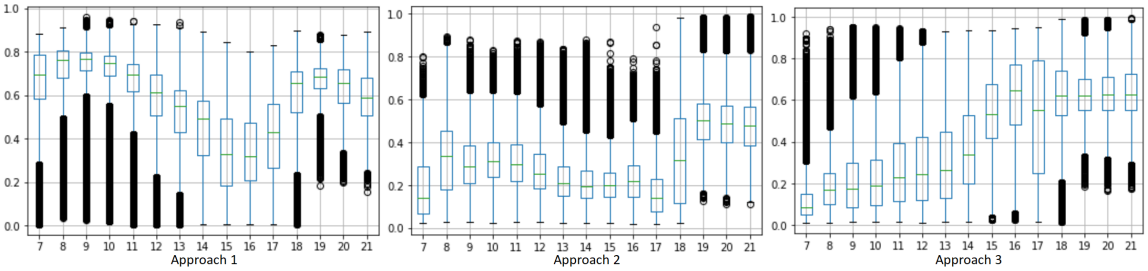


Fig. 3. Box plot showing traffic queue density for 40 days. Traffic density increases or decreases at different hours on different days. Road approach 1 and 2 have higher traffic densities compared to approach 3 where traffic gradually increases and then continues to remain high.

y-axis. This indicates variable traffic densities within that particular hour over the 40 days. Thus density does not start to rise or fall exactly at the same time every day during peak hours, but varies from one day to another. This observation of traffic congestion increasing or decreasing at different hours on different days, motivate us to built a runtime controller for GPU concurrency. A static concurrency controller, based on say time of the day will not be effective. Setting the time statically before deployment will not adapt well to the day to day variations in congestion patterns. DynCNN however, running continuously on CPU cores to monitor real time densities will be able to adapt to these changes.

We additionally observe from the graphs that the road approach 1 sees higher traffic density than approach 2. The hourly pattern of traffic in both approach 1 and approach 2 comprises higher values during morning peak hours, then a drop at noon, and again higher densities in the evening peak hours. However, approach 3, which joins the other approaches to form a T junction, has an independent pattern where the traffic gradually increases and then continues to remain high, as the day progresses. These different levels of traffic densities in different approaches and the diurnal patterns depend on the feeder road networks for these approaches and the volume of vehicles pushed based on workplace and residential blocks' locations with respect to this intersection.

The observation of variable traffic densities spatially across road approaches and temporally across hours of a day give us an indication of how this application's dynamisms can be used to vary GPU concurrency, and therefore keep temperature and power under much better control at no loss of application performance. For example, it should be possible to use different number of GPU threads based on traffic dynamism. Spawning the maximum number of GPU threads always is wasteful, as a GPU thread for a certain camera seeing very low or no traffic density is unnecessary. Similarly GPU frequency can be controlled based on application's varying FPS requirements at different traffic densities. Our deployment and long term data shows the necessity of a dynamic controller like `DynCNN` and hints at control opportunities based on fluctuating density values. We discuss the concrete system parameters like GPU thread counts and frequencies that `DynCNN` can control in real time, based on traffic densities measured on the CPU, in Section 6.

Summary:
Based on the analysis of the 40 days data, we observe that the traffic congestion keeps on increasing or decreasing at different hours on different days and additionally, different approaches of the roads observe different levels of traffic densities as shown in Figure 3

4 The Promise of Concurrent Neural Networks on Embedded GPU Devices

Edge devices are being packed with more accelerator cores like GPU and are provided with software support for creating and running many threads and processes for concurrency. In this section, we demonstrate the promise of concurrent neural networks on many such NVIDIA GPU based embedded platforms, namely Jetson TX2^{TM} [19] (256 CUDA cores, 8GB RAM), Xavier NX^{TM} [21] (384 CUDA cores, 8GB RAM) and Xavier AGX^{TM} [20] (512 CUDA cores, 32GB RAM). We start with multi-processing based concurrency, wherein we see a significant overhead due to RAM size and context-switching limitations. We then discuss multi-threading based concurrency in more detail, which we carry forward in subsequent sections to examine thermal and power trade-offs and eventually design our `DynCNN` controller to control the thread count and frequencies based on application state.

4.1 Neural Network Concurrency with Multi-processing

To explore concurrency on the embedded GPU platforms, we start with multi process execution, each process running one neural network inference task. We use TensorRT optimized neural network inference tasks from model zoo [1] as benchmarks for this experiment. Figure 4 shows the number of concurrent processes that can be run using the widely popular TinyYOLO object detection model for CNN inferences. The process count is bound by the RAM size, as each process has disjoint memory address space. For example, in TX2 with 8 GB RAM, 8 processes can be run concurrently. This number will reduce if a larger CNN network with more model weights is used.

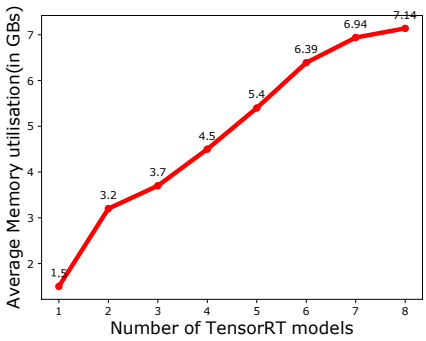


Fig. 4. Number of concurrent processes is limited by RAM size. The maximum processes that are supported on TX2 are 8.

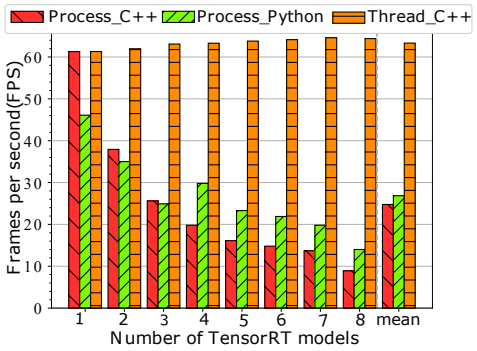


Fig. 5. Different GPU concurrency techniques where multi-processing sees reduction in FPS due to context-switching overhead.

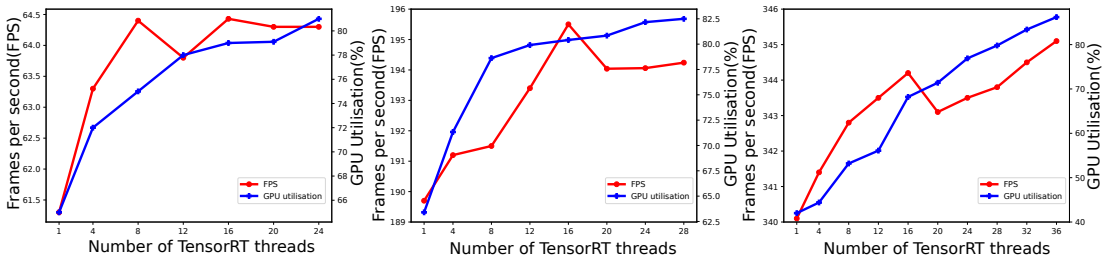
Figure 5 further examines the FPS obtained by the 8 processes, when the CPU application calling the GPU CUDA kernels is written in C++ vs. Python. While a single process starts at 60 FPS, increasing the number of processes reduces the FPS to 20 at 8 processes, due to inter-process context switching overhead. Figure 5 shows the FPS also for C++ threads, which can maintain a stable FPS at 60. We will examine C++ multi-threading next as that will be shown to support much higher levels of concurrency than multi-processing. We will also use multi-threading for all subsequent experiments. However, multi-processing, though limited by RAM size and context-switching overhead, is nonetheless important, as it allows each process to run with a different CNN model. Multi-threading shares the CNN models across all threads in a CUDA context, and therefore concurrent running of different CNNs is infeasible.

4.2 Neural Network Concurrency with Multi-threading

To support multi-threading on the embedded GPU platforms, we used CUDA *Contexts* and *Streams*. A CUDA *Context* represents a virtual address space on a GPU and holds management data (e.g. memory allocation list, CPU-GPU mapping etc.). A CUDA *Stream* masks the memory copy latency and enables concurrent execution of multiple GPU kernels to increase GPU utilization. We use a single context and bind multiple streams within that context. Each neural network inference task is connected to a specific stream and all the streams are executed in parallel.

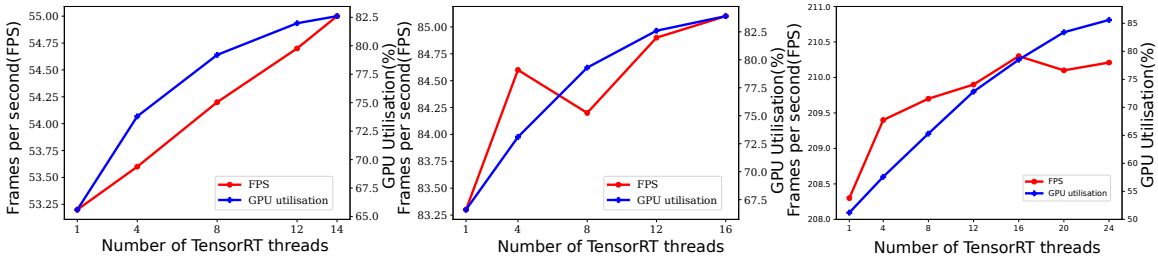
In multi-threading, all threads in the same stream use the same neural network model in a shared memory address space. So unlike multi-processing, multi-threading based concurrency cannot handle different neural networks tasks with separate models. If applications can be designed to reuse the same neural network model across different threads, then multi-threading is a viable concurrency option. We will demonstrate such an application from a developing region traffic intersection, where many traffic cameras pointing in different directions, can all use the same fine-tuned CNN model like TinyYolo for vehicle and number plate detection tasks. This application, and similarly many others where the same model needs to applied to many inputs, can use multi-threading based concurrency.

We show the GPU utilization and FPS characteristics with increasing number of threads for a small CNN *Tiny-Yolo* and a larger CNN *Googlenet*, both of which are widely used CNNs for object detection applications [24, 25]. For Tiny-YOLO, TX2 and NX boards show GPU utilization to saturate at 82% (refer to Figure 6a and Figure 6b). The maximum number of threads that are supported for TX2 and NX are 24 and 28 respectively. For Xavier AGX, the GPU utilisation increases to 86.2% for 36 threads (refer to Figure 6c) as more resources are available in Xavier AGX, compared to TX2 and NX.



(a) TX2-Tiny-YOLO (b) NX-Tiny-YOLO (c) AGX-Tiny-YOLO

Fig. 6. FPS and GPU utilizations on TX2, NX and AGX when Tiny-YOLO CNN is run. TX2 saturates at 24 GPU threads, NX at 28 GPU threads and AGX at 36 GPU threads, with GPU utilization slightly above 80% in all three cases. RAM bandwidth bottleneck marks this thread saturation points. FPS is around 64, 196 and 346 per thread for the saturation thread count, depending on the supported clock frequencies of the three platforms.



(a) TX2-Googlenet (b) NX-Googlenet (c) AGX-Googlenet

Fig. 7. FPS and GPU utilizations on TX2, NX and AGX when Googlenet CNN is run. TX2 saturates at 14 GPU threads, NX at 16 GPU threads and AGX at 24 GPU threads, with GPU utilization slightly between 82-85% for the three boards. RAM bandwidth bottleneck marks this thread saturation points. FPS is around 55, 85 and 210 per thread for the saturation thread count, depending on the supported clock frequencies of the three platforms.

However, for a heavier model Googlenet, the number of threads supported are 14/16/24 (less compared to Tiny-YOLO) for TX2, NX and AGX respectively (refer to Figure 7a, Figure 7b and Figure 7c). For the similar boards, TX2 and NX, the GPU utilisation increases on increasing the number of threads and saturates at 82.1% and 83.2% respectively. For Xavier AGX the GPU utilisation increases to 85.6% and the number of threads saturate at 24. We observe that for the heavier model the number of CNN threads supported are less compared to a lighter model. This is because the computation in heavier model is more and as such with increasing number of threads, the GPU utilisation reaches the saturation more quickly compared to a lighter model. Additionally, the RAM bandwidth can saturate earlier as more model weights are read by the increasing number of threads. Note that the similar results have been reported in [32] for NX and AGX, we reproduce the results in our paper with the additional results on TX2.

Summary:

❶ The RAM size and overhead of context switching limits the neural network concurrency using multiprocessing on edge devices.

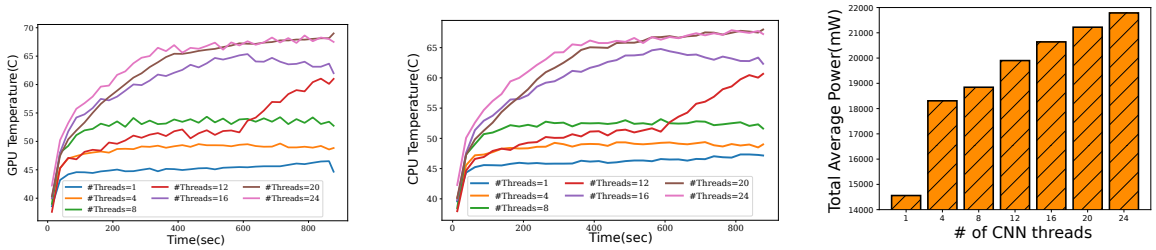
❷ Using multithreading, we can run maximum of 36 threads and 24 threads for the smaller model (like TinyYolo) and heavier model (like googlenet) respectively.

5 Thermal and Power Constraints in Neural Network Concurrency

While the state-of-the-art edge devices are shown to support excellent levels of CNN concurrency at very high throughput, continuous operations on such edge devices can lead to rising temperatures and higher power consumption which in turn can reduce the longevity of the platforms. Avoiding this by dynamically changing the GPU concurrency levels based on application state and input is the motivation behind DynCNN. In this section, we demonstrate this with a motivational example in which we run the TensorRT optimized TinyYOLO CNN on three different embedded GPUs with different GPU thread counts and frequencies.

5.1 Platform temperature and power consumption rises as GPU thread count increases

Figures 8a and 8b show the GPU and CPU temperature rise with increase in GPU thread count for AGX. We can see that temperature rises upto 20°C for AGX on increasing the threads from 1 to 24. This is evident by comparing the flat portions of the lowest and the highest curves in Figures 8a and 8b. On similar lines, with the increase in the number of threads from 1 to 24, the total power increases by roughly around 7500 mW for AGX (refer to Figures 8c). We observe the similar trends for TX2 (6-8°C rise in temperature and 1350 mW increase in power) and NX (6-8°C rise in temperature and 1300 mW increase in power). However, due to the lack of space, we show the representative plots of AGX only. Moreover, we want to mention here that we show the total combined power (CPU and GPU) for all the three boards to maintain the uniformity as NX does not provide us the CPU and GPU power separately. Both temperature and power rises, corresponding to GPU thread count increase, are significant.

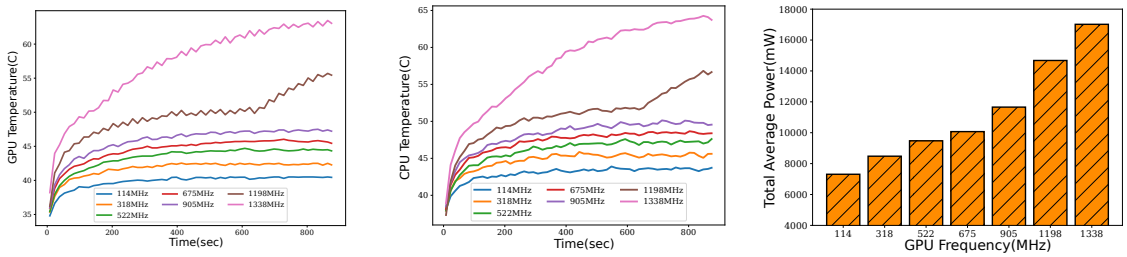


(a) AGX GPU Temperature (b) AGX CPU Temperature (c) AGX Power

Fig. 8. Temperature and Power with varying CNN thread counts for AGX (Representative graph). Temperature rises upto 20 degree Celsius and Power varies by 7500 mW on AGX, as GPU thread count changes from 1 to 24.

5.2 Platform temperature and power consumption rises as GPU frequency increases

To obtain the maximum throughput in terms of FPS, we can run the CNN threads at the maximum GPU frequency. However, running the CNN threads at the maximum frequency will lead to additional temperature rise. We observe from the Figures 9a and 9b that with increase in GPU frequencies, both the GPU and CPU temperatures rise for the AGX board. The temperature rises upto 25°C on AGX, and varies by 10-12°C for TX2 and NX. In addition, we also observe that on increasing the GPU frequencies from minimum to maximum with a fixed CPU frequency, the total power (refer to Figure 9c) increases roughly by a factor of 2X for AGX which is significant. We observe the similar increase in power for TX2 and NX, however, we do not show them due to space limitations. The additional power consumption and increase in the temperatures can lead to lower longevity of the platforms.

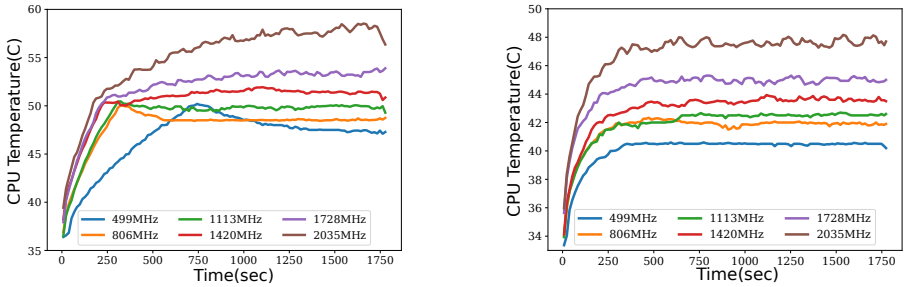


(a) AGX GPU Temperature (b) AGX CPU Temperature (c) AGX Power

Fig. 9. Temperature and Power with varying GPU frequencies for AGX (Representative graph). Temperature rises upto 25 degree Celsius and Power varies by 10000 mW on AGX, as GPU frequency changes from the minimum to the maximum.

5.3 Platform temperature rises more as ambient temperature rises

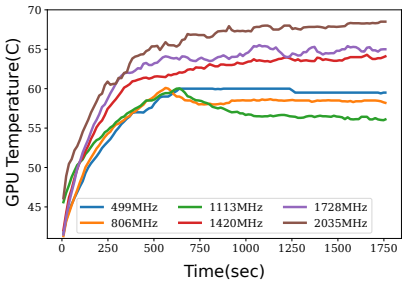
While number of concurrent GPU threads and GPU operating frequency affect the temperature rise, the thermal effect becomes more pronounced as ambient temperatures increase. The experiments above for Figures 8 and 9 were conducted at around 30°C ambient temperature. When the same experiment is conducted at 35 degree or 45°C ambient temperature, the platform temperatures go up to 60 and 70 °C respectively. This is shown in Figure 10 and Figure 11 for Jetson TX2. The left plots show the default behavior of the platform, when the embedded fan is kept off in the beginning, and it starts after a Linux kernel defined threshold temperature (around 50°C). The right plots show the temperature rise, when the embedded fan is turned on beforehand and kept on throughout the experiment. The fan keeps the temperatures lower by around 10 degrees. However, continuous operations even with the embedded fan on, has high thermal overhead. We have anecdotal evidence of all our platforms malfunctioning after the 45°C ambient temperature experiment. They would arbitrarily reboot or not get powered on, indicating that continuous operation near the boards' thermal safety thresholds is inadvisable. Note that the CPU and GPU temperature rise is almost similar at the respective ambient temperature for both fan off and on modes. Due to the space limitations, we show CPU temperature for 35°C and GPU/CPU temperature for 45°C.



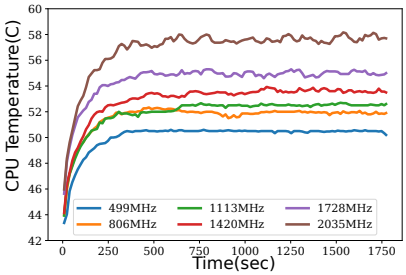
(a) CPU Temperature with fan off at 35 degree ambient temperature

(b) CPU Temperature with fan on throughout at 35 degree ambient temperature

Fig. 10. Temperature rise at 35 degree ambient temperature. Keeping the fan on throughout maintains the temperature lower by around 10 degrees.



(a) GPU Temperature with fan off at 45 degree ambient temperature



(b) CPU Temperature with fan on throughout at 45 degree ambient temperature

Fig. 11. Temperature rise at 45 degree ambient temperature. Keeping the fan on throughout maintains the temperature lower by around 10 degrees.

Thus state-of-the-art embedded GPU platforms can run 24/28/36 CNN threads for small CNN models like TinyYolo, each with 64/190/340 FPS, as seen in Section 4. For larger CNN models like GoogleNet, they can still run 14/16/24 threads each with 55/85/210 FPS. However, the corresponding rise in platform temperature, especially at high ambient temperatures is significant (upto 20-25 degree Celsius rise within 15 minutes on AGX). In addition, power consumption also goes up by a factor of 2X. These edge computing devices might be deployed in scenarios where thermal and power constraints are real. For example, a drone flying over a desert to locate a lost person or a traffic controller deployed in a tropical country intersection can easily see 45-50 degree Celsius ambient temperatures in summer. In absence of careful thermal control, the platform will be destroyed even after 30 minutes of operation. Power similarly is scarce in a battery operated drone or limited in a solar powered traffic intersection. Power budget further reduces cooling options for these boards as running a fan continuously will increase power consumption. Air-conditioners are infeasible unlike in a lab or data-center setting. Passive cooling using metal casing etc. might be the only viable option. GPU concurrency is thus a tremendous promise that needs to be carefully tamed for thermal and power constraints.

This performance-thermal-power trade-off motivates DynCNN, an intelligent neural network concurrency controller that adapts to an application's dynamic needs. Specifically for the camera based road traffic monitoring in developing countries discussed in Section 3, road-side deployments at high ambient temperatures and limited power for solar sustainability, make DynCNN a necessity. We discuss the design, implementation and evaluation of DynCNN next.

Let us summarize the insights.

Summary:

❶ The platform temperature and power consumption increases by 20°C and 7500mW as we increase the number of threads from 1 to 24 (refer to Figures 8a, 8b, 8c).

❷ Both the GPU and CPU temperatures rise with the increase in GPU frequencies. Also, with a fixed CPU frequency, we observe the total power increases roughly by a factor of 2X on increasing the GPU frequencies from minimum to maximum(Refer to Figures 9a, 9b, 9c).

❸ The thermal effect becomes more pronounced as ambient temperatures increase. When keeping the fan-off, the platform temperature increases more by a 10°C compared to when keeping the fan-on for the whole execution (refer to Figures 10a, 10b, 11a, 11b).

6 DynCNN: Application dynamism aware CNN concurrency controller

The main idea behind DynCNN is to exploit the heterogeneous processor cores available in state-of-the-art edge devices with CNN accelerators e.g. CPU and GPU cores present in the NVIDIA Jetson and Xavier platforms we use. The main computations for the edge application requiring concurrent CNNs run on the GPU cores. For instance, (a) vehicle and number plate detection and OCR in traffic monitoring application, or (b) suspicious object detection in drone based surveillance or (c) speech to text conversion from multiple microphones in a conference room setting, will all use neural networks running on the GPU cores. Concurrent neural networks might also be necessary, based on how many cameras or microphone feeds needs to be handled at what throughput.

However, the CPU cores on the same edge platforms can additionally monitor the application state. For example, CPU cores can monitor (a) traffic density in intersection control and traffic monitoring application, (b) presence of suspicious movement in drone based intrusion detection application or (c) presence or absence of acoustic signal in particular microphones in a speech to text transcription for conference room scenario. Based on the CPU based processing results, the CPU can then dynamically control the level of concurrency on the GPU cores (e.g. GPU thread count and frequency), as needed to optimally support the application's current state. Instead of running maximum possible GPU threads at maximum possible frequencies and risk platform lifetime with high power surge and temperature rise, application aware concurrency control can make edge computing safer, while not compromising on application performance. This section explores this idea of DynCNN concurrency controller design, for the specific use-case of traffic monitoring at intersections. Other applications can use a similar approach to control GPU concurrency based on CPU based monitoring of the application state, changing what to monitor and what to control in the CPU as necessary.

6.1 Can variable traffic densities measured by CPU control GPU concurrency?

As seen in Section 3, the traffic densities differ spatially across different road approaches of the same intersection, and also temporally over the course of the day depending on changing vehicle volumes in morning peak vs. evening peak vs. non-peak hours. In this section, we first explore whether these variable densities can be translated to different needs of GPU concurrencies, and then design the DynCNN controller based on those insights.

6.1.1 Varying number of GPU threads depending on traffic density: Each GPU thread handles a particular camera's frame inputs and processes it with object detection CNNs. There are six cameras per road approach in our deployment (see Figure 1). Three cameras (one for vehicle detection and two for ANPR) monitor the head of the traffic queue for that approach and remaining three cameras (again one for vehicle detection and two for ANPR) monitor the tail of the traffic queue. With six cameras per road approach, a three or four-way intersection needs 18 or 24 GPU threads. When the road approaches are wider than in our deployment setting, requiring more than 6 cameras per road approach to cover the greater road width, then the total number of cameras and corresponding maximum number of GPU threads will be more.

Since the head of the queue starts getting filled before the tail, when traffic densities are low depending on the time of the day, the tail of the queue might be empty. Thus the three cameras monitoring the queue tail should ideally not need GPU threads to run their CNN tasks when they see low to no traffic densities. If one road approach can save three GPU threads for three cameras when its queue tail is empty, a three or four way intersection can save 9 to 12 GPU threads out of the 18 or 24 maximum GPU threads respectively, if all their queue tails are empty. When road widths are more with higher number of maximum GPU threads, opportunity to stop some threads at low traffic densities will be even more.

As seen in Section 5, lower number of GPU threads uses less power (Figure 8c) and keeps platform temperatures lower (Figure 8a, 8b). Thus if the application can use lower number of GPU threads based on dynamic traffic densities measured

on the CPU, controlling the number of GPU threads can be a very effective way of power and thermal control. We use this first insight of traffic density aware GPU thread start stop control in DynCNN.

6.1.2 Varying GPU frequencies depending on traffic density: We next examine whether different traffic densities can use different GPU frequencies. When traffic densities are high, vehicle movements are slower. Thus processing the video frames at lower FPS with corresponding lower GPU frequencies can be acceptable, as the monitored scene is changing at a slower rate. However, when traffic densities are lower, vehicles move at high speeds. Thus maximum supported FPS at the maximum possible GPU frequency must be used under such circumstances, to avoid missing over-speeding vehicles that should be detected and penalized. Thus different traffic densities see different vehicle dynamics on the road, which can be exploited to vary the GPU frequencies to support just the correct level of FPS needed at that instant.

As seen in Section 5, lower GPU frequencies directly translate to lower temperatures (Figure 9a, 9b) and power (Figure 9c). Simultaneously running many GPU threads, each at a different GPU frequency, based on the traffic density seen by that camera, is not feasible on current embedded GPU devices, as there are not so many individual clock domains. Still, based on the FPS requirements of the running CNN threads, a lower frequency can be set for all simultaneous threads, and this common GPU frequency can be varied based on dynamic traffic densities. The resulting power and temperature rise compared to maximum GPU frequencies might be less in such a scenario. In addition to starting and stopping GPU threads, DynCNN also therefore controls the GPU frequencies for those threads, based on the traffic density inputs from the CPU.

6.2 DynCNN Thermal Control Algorithm

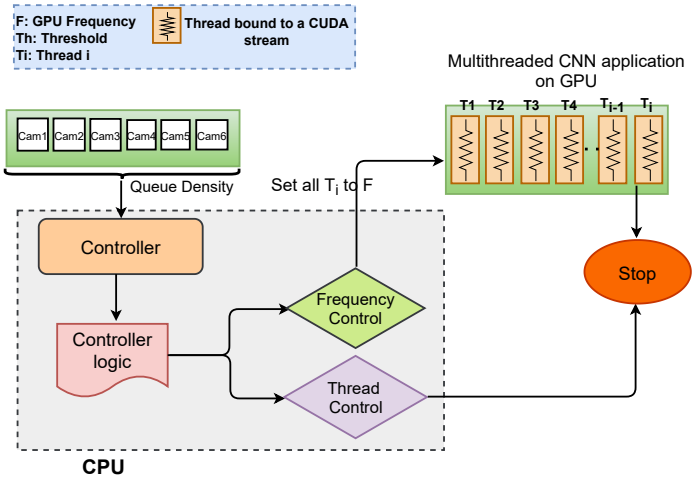


Fig. 12. DynCNN Thermal Controller Workflow.

Based on the intuitions for application aware thermal and power control, we present DynCNN controller next. DynCNN comprises three components: (a) traffic density measurement on CPU for all cameras to assess the application's current state, (b) determining which CNN threads to run at what common GPU frequency based on the densities, and (c) actually giving the control signal to the GPU to start/stop the necessary threads and change the GPU frequencies. Figure 12 shows how DynCNN works in our embedded edge devices. The camera inputs come to the CPU cores (for instance, over Ethernet LAN at different IP addresses in our deployed units). The CPU runs the traffic density estimation algorithms as described

in Section 3 using Background Subtraction and Optical Flow based Computer vision methods. These densities are input to the DynCNN control algorithm (Algorithm 1), which then determines the appropriate GPU frequencies needed to support the optimal FPS (*Frequency Control Unit*), and also which GPU threads to stop if traffic densities for those cameras fall under a threshold (*Thread Control Unit*). Based on the algorithm output, the GPU DVFS is invoked to set the appropriate GPU frequency, and unnecessary GPU threads are stopped.

Algorithm 1 DynCNN Thermal Control Algorithm

Input: *List of Queue density values (d1,d2,...,d24) from different cameras*
Output: *GPU frequency (F) and the list of CNN threads*
Initialisation: D ={d1,d2,d3.....,d24}
Epoch:= 60s ▷ *Determined Experimentally*
Threshold: T= {0.3,0.6} ▷ *Threshold values for three bins*

1: **procedure** FREQUENCY CONTROL($D, F_{min}, F_{mid}, F_{max}, T$)
2: $[bin_1, bin_2, bin_3] \leftarrow binCount(D, T)$ ▷ *obtains the count of each bin*
3: $index \leftarrow argmax\big([bin_1, bin_2, bin_3]\big)$ ▷ *Retrieves the index of the max count bin*
4: $F \leftarrow [F_{max}, F_{mid}, F_{min}]\,[index]$ ▷ *Sets the frequency corresponding to the bin*
5: **procedure** THREAD CONTROL(D)
6: **for each** $d \in D$ **do** ▷ *Iterate through the density values in set D*
7: **if** $d_i < \mathscr{T}$ **then** ▷ *Density value of thread i is less than threshold*
8: **Stop** Th_i ▷ *Stop the thread*
9: **end for each**

DynCNN can be run periodically at certain epochs. In our experiments, we set the epoch to be 60 seconds as we see traffic densities seem to be stable over that window size based on our 40 days' deployment (Figure 3). In our algorithm, we divide the density values into three bins based on the threshold value (T) where bin_1,bin_2 and bin_3 refers to the density values in the range 0-0.3,0.3-0.6,0.6-1 respectively. The count of each bin is calculated using *binCount* and the function *argmax* gives the index of the maximum count bin (Lines 2-3). The GPU frequency is set according to the bin with the maximum count (Line 4). Note that bin_1,bin_2 and bin_3 corresponds to lower, intermediate and larger density respectively, hence we set F_{max}, F_{mid} and F_{min} for these three bins in our algorithm. Additionally, if the maximum counts of two bins are equal, we choose the bin which corresponds to the larger frequency. The various thresholds in Algorithm 1, for what FPS is needed at what traffic density, and correspondingly what minimum GPU frequency can support that FPS, are empirically learnt using 1-2 days' data collected from our deployment intersection. Similarly, for the *Thread Control*, we check if the density value for a particular thread is less than some threshold, we stop the thread otherwise it keeps on running (Lines 7-9). A similar traffic monitoring system deployed elsewhere or a DynCNN running for a different application, can set the necessary thresholds in a similar way from empirical data.

6.3 DynCNN Controller Evaluation

6.3.1 Existing GPU governors as baselines: Modern edge devices with embedded GPUs include multiple GPU governors as part of the system software, since power and temperature are important metrics to balance, in addition to application performance metrics like throughput and latency. These governors control the GPU frequency using the Dynamic Voltage Frequency Scaling (DVFS) drivers following different heuristics. Some governors are more considerate about the application performance and therefore will set higher GPU frequencies to get maximum throughput and minimum latency. Other governors might more aggressively control power and temperature and therefore set lower GPU frequencies. Based on deployment setting, developers can choose a governor for their system, depending on what trade-off they require between application performance and power control. It is important to compare DynCNN with these already deployed GPU governors, to see whether application awareness can bring any additional benefit to balance the performance-power-thermal trade-offs. We list five standard GPU governors along with our proposed DynCNN controller in Table 3, with the

		Priority metric	
GPU Governor	*DVFS Heuristic*	*App Performance*	*Power/Temperature*
Performance	Sets to the maxiumum frequency	✓	✗
On-demand	System load decides the frequency	✓	✓
Userspace	User specified desired frequency	✓	✓
Wmark	Generates interrupts based on load	✓	✓
Nvhost_ podmark	Frequency scaled based on throughput hints	✓	✗
DynCNN	**Frequency and threads scaled based on application input**	✓	✓

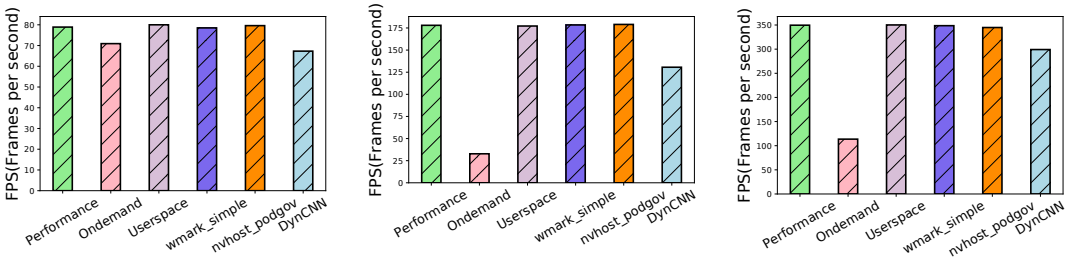
Table 3. Existing GPU governors used as Baselines to evaluate DynCNN metric trade-offs

heuristics they use for GPU frequency control and which metric among application performance and power/temperature they prioritize.

6.3.2 Experimental Methodology We split our 40 day traffic density data into 20 minute chunks, to ensure each experiment runs for less than 20 minutes. This is necessary for platform safety as some baseline GPU governors push power and temperatures to dangerous levels even in 15-20 minutes. We cluster the 20 minute chunks based on traffic densities observed over those 20 minutes. For instance, morning peak forms a cluster, evening peak another, non-peak a third. These clusters allow us to take some 20 minute data traces that are representative of that cluster, and obtain statistics for that window. This avoids processing all the traffic density traces which is repetitive. We run DynCNN on some 20 minute traces and obtain the various statistics (FPS, Temperature and Power) and repeat this for all the clusters. Finally, we take a weighted average of the statistics obtained per cluster where weight is the number of 20 min instances present in a particular cluster. We plot this average for three metrics: ❶ *FPS (Frames per second)* ❷ *GPU and CPU Temperature* and ❸ *Total Power* over 40 days of traffic data, and compare DynCNN with five baseline GPU governors under two CPU settings.

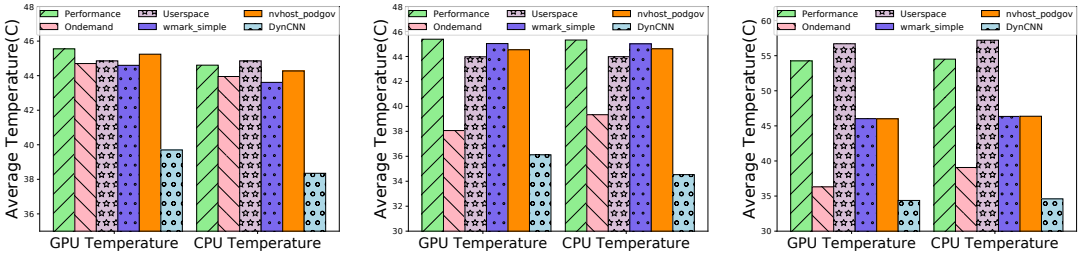
6.3.3 CPU Baseline1: SchedUtil governor We first compare the DynCNN controller with the CPU Baseline1. We set the CPU to the SchedUtil governor that dynamically adjusts the minimum and the maximum frequency. Figures 13a, 13b and 13c show the comparison of FPS (Frames per second) of existing GPU governors with our DynCNN controller for the Baseline1. We observe from the three graphs that majority of the existing governors(except OnDemand) maintain the higher application FPS (more by 10 in TX2, 50 in NX and AGX) compared to our DynCNN controller. This is because most of these governors prioritize the application performance (refer to Table 3). However, the OnDemand governor maintains a good balance between the performance and the battery savings and thus maintains itself at the lower frequency compared to other governors(as observed from experiments). This leads to a very low FPS in the OnDemand GPU governor compared to the other governors and our DynCNN controller. In addition, all these existing GPU governors (except OnDemand) maintain the FPS quite high which is more than required by the application. Our proposed DynCNN controller instead dynamically sets the GPU frequency based on the application needs as guided by the traffic densities measured by the CPU. This makes rest of the GPU governors application agnostic compared to our DynCNN controller which is application dependent. Note that for the userspace governor, we do not set any frequency (keep it default). We want to mention here that even if the user specified frequency was set for userspace governor, it would still not adjust to the application behaviour dynamically unlike DynCNN.

We next plot the average temperatures for each of the existing governors (using Baseline1) and compare these values with our DynCNN controller. We observe from the Figures 14a, 14b and 14c that the average CPU and GPU Temperatures using our DynCNN controller is the least among all. For TX2, the average CPU and GPU Temperatures for DynCNN are



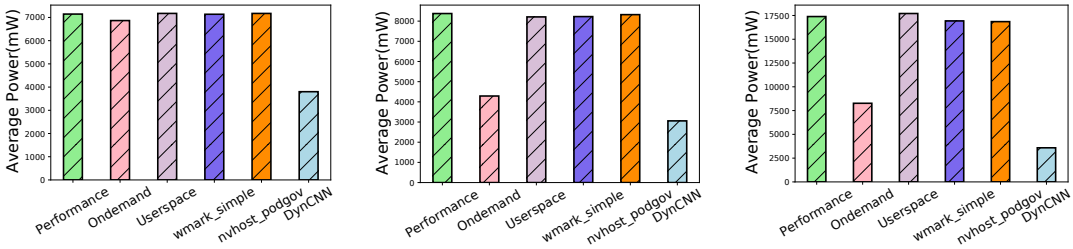
(a) TX2 FPS (b) NX FPS (c) AGX FPS

Fig. 13. Application performance in FPS with SchedUtil CPU governor as CPU Baseline1 and multiple GPU DVFS governors given in Table 3. All the governors except OnDemand maintain the higher application FPS (more by 10 in TX2, 50 in NX and AGX) than DynCNN. OnDemand maintains itself at a lower frequency due to battery savings and hence has lower FPS



(a) TX2 Temperature (b) NX Temperature (c) AGX Temperature

Fig. 14. Average temperature with SchedUtil CPU governor as CPU Baseline1 and multiple GPU DVFS governors given in Table 3. Average CPU and GPU temperatures for DynCNN are less by 5-6°C for TX2, 2-10 °C for NX and 2-22 °C for AGX as all the existing governors target the application performance by keeping the frequency maximum most of the times.



(a) TX2 Power (b) NX Power (c) AGX Power

Fig. 15. Average power with SchedUtil CPU governor as CPU Baseline1 and multiple GPU DVFS governors given in Table 3. The average power consumed by DynCNN is less by 4000 mW,1300-5000 mW and 5000-14500 mW for TX2, NX and AGX respectively. DynCNN dynamically adjusts the frequency and the count of threads leading to less power consumption.

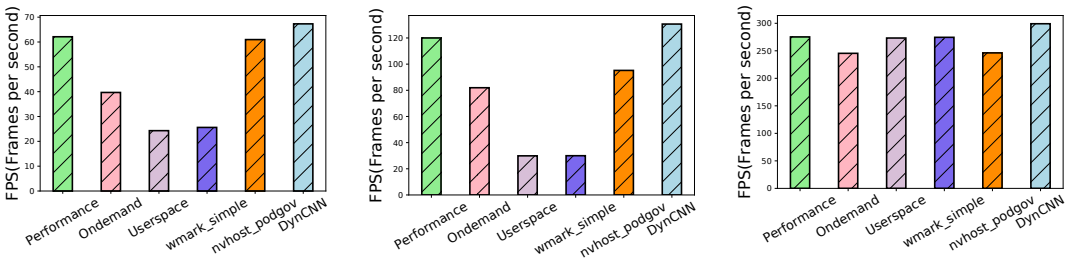
less by around 5-6 °C than the existing GPU governors. For NX, the average CPU and GPU Temperatures are less by 2-10 °C and similarly for AGX, the values are less by 2-22 °C. On an average, the average temperature is less by around

12°C for DynCNN compared to the existing governors. Clearly, DynCNN outperforms all the existing GPU governors in the average Temperature. This is intuitive as most existing GPU governors prioritize application performance for which they need to keep the frequency at the maximum level most of the times. Even though this ensures the high application FPS (most of the time unnecessary for the application), it leads to a significant GPU and CPU temperature rise. Similarly, all these governors are agnostic to the number of CNN threads that are required at any particular instance based on the traffic density values and hence it adds to the temperature rise. Thus DynCNN manages temperatures remarkably well at the right level of application throughput.

We finally compare the total power (GPU and CPU) consumed by the existing governors vs. DynCNN in Figures 15a, 15b and 15c. The total power consumed by the existing governors is comparatively much higher than our proposed DynCNN controller. The average power consumed by DynCNN is roughly less by 4000 mW (Figure 15a), 1300-5000 mW (Figure 15b) and 5000-14500 mW (Figure 15c) for TX2, NX and AGX respectively which on an average is roughly around 68.82% less power compared to the existing governors. The primary reason that DynCNN consumes less power is because we dynamically adjust the frequency and the number of threads based on the need of an application rather than setting it always at the maximum as done by these existing governors.

In summary, existing GPU governors are application agnostic and thus maintain much higher GPU frequency than needed by the application, based on the DVFS heuristic they follow (Table 3). However, DynCNN controller takes into consideration the application requirement. Based on the dynamism of the application, it adjusts the GPU frequency and additionally the number of GPU threads. DynCNN thereby uses much less power at much lower temperatures, while maintaining the application throughput at the correct FPS.

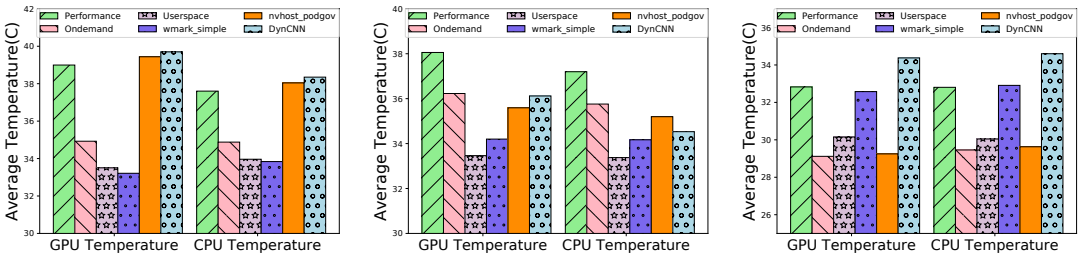
6.3.4 CPU Baseline2: DVFS manually set based on application FPS requirements We also study the behaviour of existing GPU governors when used with CPU Baseline2 and compare it with DynCNN. In CPU Baseline2, we disable the DVFS and manually set the CPU frequency that is used to monitor the traffic density in intersection control. We set the CPU frequency to the value at which we obtain the FPS roughly equal to 10 as for density measurement on CPU, 10 FPS is good enough. Once we set the CPU frequency, we compare the performance of the GPU governors with DynCNN.



(a) TX2 FPS (b) NX FPS (c) AGX FPS

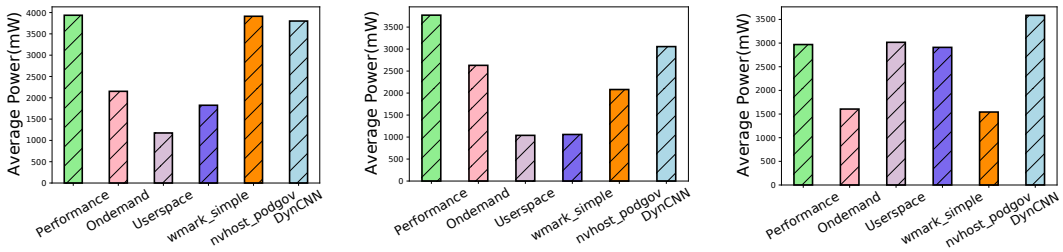
Fig. 16. Application performance in FPS with fixed CPU frequencies as CPU Baseline2 and multiple GPU DVFS governors given in Table 3. Existing DVFS governors adjust the GPU frequency based on the setting of CPU frequency. Performance improvement of 31.2% is observed compared to the existing governors.

Figures 16a, 16b and 16c show the comparison of FPS of DynCNN with the existing GPU governors for CPU Baseline2. We observe that we outperform all the existing GPU governors in the FPS metric. On an average, we have a performance improvement of roughly around 31.2% compared to the Baseline2. DynCNN performs better than these governors in FPS



(a) TX2 Temperature (b) NX Temperature (c) AGX Temperature

Fig. 17. Average temperature with fixed CPU frequencies as CPU Baseline2 and multiple GPU DVFS governors given in Table 3. All the existing governors maintain the GPU at a lower frequency due to setting of CPU at a fixed frequency and hence average temperature is less by 2-4°C compared to DynCNN.



(a) TX2 Power (b) NX Power (c) AGX Power

Fig. 18. Average power with fixed CPU frequencies as CPU Baseline2 and multiple GPU DVFS governors given in Table 3. GPU frequency maintained is lower in existing governors and hence power consumed is less compared to DynCNN.

because all of these GPU governors are affected by the frequency that is set in the CPU. Our observation is that all GPU governors adjust GPU frequency according to the frequency set up in the CPU. We also observe during our experiments if we increase the CPU frequency, the GPU frequency for all the GPU governors does increase. In both ways, DynCNN will outperform the existing governors, either in performance or in Temperature/Power as all these governors do not take into consideration the application behaviour which DynCNN does.

As mentioned above, in CPU Baseline2, the GPU frequencies are adjusted by the existing governors according to the fixed frequency set in the CPU. All GPU governors maintain the frequency at a lower value compared to that in CPU Baseline1 and hence the average GPU and CPU Temperature is slightly less(2-4 °C)(refer to Figures 17a, 17b, 17c) compared to DynCNN. Power also is lower than DynCNN (refer to Figures 18a, 18b and 18c). However the lower temperatures and powers here are at a large cost to FPS, whereas DynCNN maintains required FPS while also respecting thermal and power thresholds. In summary,DynCNN outperforms existing GPU governors for CPU Baseline2 in FPS at good thermal-power trade-off.

Summary of evaluation results:
❶ The average temperature is less roughly by around 12°C for DynCNN compared to the existing GPU governors with SchedUtil CPU governor as baseline. Moreover, the average power consumption is less by around 68.82% compared to the existing governors. However, the application FPS is higher in most of the existing governors with SchedUtil CPU governor as baseline because majority of these governors priortize the application performance.
❷ When the CPU frequency is manually set based on application FPS requirements, we observe a performance improvement of around 31.2% in DynCNN compared to when the existing GPU governors are used. However, the average temperature and power consumption are less in existing GPU governors compared to DynCNN as setting the CPU frequency manually affects the GPU frequency and we observe that GPU frequency is maintained at a lower value in all the existing governors when the CPU frequency is set manually which we call as Baseline2.

7 Discussion

Platform Temperature and Power increase: With the increase in the number of the GPU threads, we observe that there is a significant increase in the platform temperature and the power consumption leading to the heating of the boards. In addition, there is an additional increase in the temperature and power if the GPU frequency is increased. Moreover, we also analyze the effect of varying ambient temperatures on the platform temperature. We analyze the two cases here - fan on and fan off. By keeping the fan on throughout, we observe that there is a temperature decrease of around 10 degrees compared to fan off mode.
DynCNN for Thermal Control: Traffic densities on the roads keep on increasing or decreasing at different hours of the day. Moreover, different intersections of the roads observe different levels of traffic densities(refer to Figure 3). DynCNN can handle both the scenarios by dynamically adjusting the GPU frequency based on the traffic congestion on the roads. If the congestion is high, GPU frequency can be lowered, thus reducing the platform temperature. In addition, DynCNN can dynamically adjust the number of GPU threads by looking at the different intersections of the roads. If the density for an intersection is less than a threshold, the GPU thread for the intersection is stopped otherwise it keeps on running. This also aids in reducing the temperature of the platform.

8 Conclusion and Future Work

Efficient traffic management using edge device deployment has various associated constraints with it such as power constraint, ambient temperature etc. In this paper, we examine the trade-off between the maximum throughput on edge devices and the associated power and thermal characteristics. We, then propose DynCNN, an application dynamism and ambient temperature aware controller for neural network concurrency. DynCNN efficiently uses the processor heterogeneity to control the GPU frequency and the number of GPU threads executing on the GPU. Using this, we are able to reduce the average temperature and power by ~12°C and 68.82% for Baseline1 and improve the performance by 31.2% for Baseline2 with the existing GPU governors. This is a promising step towards real world adoption of automated intersection control in development countries, in collaboration with our urban traffic management partners in New Delhi, India. While the urban agency collaborating with us start testing DynCNN on the road, we will extend the system to support night vision cameras, and see the long term impact of traffic intersection management algorithms on travel time, intersection throughput and air quality.

References

[1] [n.d.]. TensorRT models. https://elinux.org/Jetson_Zoo.

[2] Itamar Arel, Cong Liu, Tom Urbanik, and Airton G Kohls. 2010. Reinforcement learning-based multi-agent system for network traffic signal control. *IET Intelligent Transport Systems* 4, 2 (2010), 128–135.

[3] Shashank Bharadwaj, Sudheer Ballare, Munish K Chandel, et al. 2017. Impact of congestion on greenhouse gas emissions for road transport in Mumbai metropolitan region. *Transportation Research Procedia* 25 (2017), 3538–3551.

[4] Ganapati Bhat, Gaurav Singla, Ali K Unver, and Umit Y Ogras. 2017. Algorithmic optimization of thermal and power management for heterogeneous mobile platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 3 (2017), 544–557.

[5] Luis Castrillo. 2020. Critical Evaluation of object recognition models and Their Speed and Accuracy on Autonomous Vehicles. *Selected Computing Research Papers* (2020), 9.

[6] Yonghun Choi, Seonghoon Park, and Hojung Cha. 2019. Graphics-aware power governing for mobile devices. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 469–481.

[7] Arti Choudhary and Sharad Gokhale. 2016. Urban real-world driving traffic emissions during interruption and congestion. *Transportation Research Part D: Transport and Environment* 43 (2016), 59–70.

[8] P. Chuang, Y. Chen, and P. Huang. 2017. An adaptive on-line CPU-GPU governor for games on mobile devices. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 653–658. https://doi.org/10.1109/ASPDAC.2017.7858398

[9] E Fumagalli, Dipan Bose, Patricio Marquez, Lorenzo Rocco, Andrew Mirelman, Marc Suhrcke, and Alexander Irvin. 2017. *The high toll of traffic injuries: unacceptable and preventable*. World Bank.

[10] Reagan L Galvez, Argel A Bandala, Elmer P Dadios, Ryan Rhay P Vicerra, and Jose Martin Z Maningo. 2018. Object detection using convolutional neural networks. In *TENCON 2018-2018 IEEE Region 10 Conference*. IEEE, 2023–2027.

[11] Ujjwal Gupta, Raid Ayoub, Michael Kishinevsky, David Kadjo, Niranjan Soundararajan, Ugurkan Tursun, and Umit Y Ogras. 2017. Dynamic power budgeting for mobile systems running graphics workloads. *IEEE Transactions on Multi-Scale Computing Systems* 4, 1 (2017), 30–40.

[12] Ujjwal Gupta, Sumit K Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y Ogras. 2019. A deep Q-learning approach for dynamic management of heterogeneous processors. *IEEE Computer Architecture Letters* 18, 1 (2019), 14–17.

[13] C. Hsieh, J. Park, N. Dutt, and S. Lim. 2015. Memory-aware cooperative CPU-GPU DVFS governor for mobile games. In *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*. 1–8. https://doi.org/10.1109/ESTIMedia.2015.7351775

[14] Arman Iranfar, Soheil Nazar Shahsavani, Mehdi Kamal, and Ali Afzali-Kusha. 2015. A heuristic machine learning-based algorithm for power and thermal management of heterogeneous MPSoCs. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 291–296.

[15] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. 2012. DevScope: a nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 353–362.

[16] Seyeon Kim, Kyungmin Bin, Sangtae Ha, Kyunghan Lee, and Song Chong. 2021. zTT: learning-based DVFS with zero thermal throttling for mobile devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 41–53.

[17] Shraddha Mane and Supriya Mangale. 2018. Moving object detection and tracking using convolutional neural networks. In *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 1809–1813.

[18] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, Ravi Iyer, and Chita R Das. 2015. Domain knowledge based energy management in handhelds. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 150–160.

[19] Nvidia. [n.d.]. Jetson TX2. https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-tx2/.

[20] Nvidia. [n.d.]. Xavier AGX. https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[21] Nvidia. [n.d.]. Xavier NX. https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[22] Jurn-Gyu Park, Nikil Dutt, and Sung-Soo Lim. 2017. ML-Gov: A machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming. In *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia*. 12–21.

[23] Jurn-Gyu Park, Chen-Ying Hsieh, Nikil Dutt, and Sung-Soo Lim. 2017. Synergistic CPU-GPU frequency capping for energy-efficient mobile games. *ACM Transactions on Embedded Computing Systems (TECS)* 17, 2 (2017), 1–24.

[24] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[25] Shaoqing Ren, Kaiming He, Ross Girshick, Xiangyu Zhang, and Jian Sun. 2016. Object detection networks on convolutional feature maps. *IEEE transactions on pattern analysis and machine intelligence* 39, 7 (2016), 1476–1481.

[26] Aditya Vashishtha Venkata N. Padmanabhan Edward Cuttrell Rijurekha Sen, Andrew Cross and William Thies. 2013. Accurate Speed and Density Measurement for Road Traffic in India. In *Proceedings of the 3rd Annual Symposium on Computing for Development*.

[27] Bhaskaran Raman Rijurekha Sen and Prashima Sharma. 2010. Horn-Ok-Please. In *Proceedings of the 8th Annual International Conference on Mobile Systems, Applications and Services (Mobisys)*.

[28] Bhaskaran Raman Amarjeet Singh Rupesh Mehta Rijurekha Sen, Abhinav Maurya and Ramakrishnan Kalyanaraman. 2015. Road-RFSense: A Practical RF-Sensing Based Road Traffic Estimation System for Developing Regions. In *Transactions on Sensor Networks*.

[29] Bhaskaran Raman Rupesh Mehta Ramakrishnan Kalyanaraman Nagamanoj Vankadhara Swaroop Roy Rijurekha Sen, Abhinav Maurya and Prashima Sharma. 2012. KyunQueue: A Sensor Network System To Monitor Road Traffic Queues. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems (Sensys)*.

[30] Pankaj Siriah Rijurekha Sen and Bhaskaran Raman. 2011. RoadSoundSense: Acoustic Sensing based Road Congestion Monitoring in Developing Regions. In *Proceedings of the 8th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*.

[31] Onur Sahin, Lothar Thiele, and Ayse K Coskun. 2018. Maestro: Autonomous qos management for mobile applications under thermal constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 8 (2018), 1557–1570.

[32] Omais Shafi, Chinmay Rai, Rijurekha Sen, and Gayathri Ananthanarayanan. 2021. Demystifying TensorRT: Characterizing Neural Network Inference Engine on Nvidia Edge Devices. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 226–237.

[33] Shervin Sharifi, Dilip Krishnaswamy, and Tajana Šimunić Rosing. 2013. PROMETHEUS: A proactive method for thermal management of heterogeneous MPSoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 7 (2013), 1110–1123.

[34] Hao Shen, Ying Tan, Jun Lu, Qing Wu, and Qinru Qiu. 2013. Achieving autonomous power management using reinforcement learning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18, 2 (2013), 1–32.

[35] Alexander Shustanov and Pavel Yakimov. 2017. CNN design for real-time traffic sign recognition. *Procedia engineering* 201 (2017), 718–725.

[36] Rajeshwari Sundar, Santhoshs Hebbar, and Varaprasad Golla. 2014. Implementing intelligent traffic control system for congestion control, ambulance clearance, and stolen vehicle detection. *IEEE Sensors Journal* 15, 2 (2014), 1109–1113.

[37] Ashlesh Sharma Lakshminarayanan Subramanian Vipin Jain, Aditya Dhananjay. 2012. Traffic Density Estimation from Highly Noise Image Sources. In *Transportation Research Board Annual Summit*.

[38] Lakshminarayanan Subramanian Vipin Jain, Ashlesh Sharma. 2012. Road Traffic Congestion in the Developing World. In *Proceedings of the 2nd ACM Symposium on Computing for Development (DEV)*.

[39] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. 2019. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 305–317.