2017 IEEE International Symposium on High Performance Computer Architecture

Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures

Mingcong Song, Yang Hu, Huixiang Chen, Tao Li
Department of Electrical and Computer Engineering, University of Florida
Gainesville, USA
{songmingcong, huyang.ece, stanley.chen}@ufl.edu, taoli@ece.ufl.edu

***Abstract*—Accelerating Convolutional Neural Networks (CNNs) on GPUs usually involves two stages: training and inference. Traditionally, this two-stage process is deployed on high-end GPU-equipped servers. Driven by the increase in compute power of desktop and mobile GPUs, there is growing interest in performing inference on various kinds of platforms. In contrast to the requirements of high throughput and accuracy during the training stage, end-users will face diverse requirements related to inference tasks. To address this emerging trend and new requirements, we propose *Pervasive CNN (P-CNN)*, a user satisfaction-aware CNN inference framework. P-CNN is composed of two phases: cross-platform offline compilation and run-time management. Based on users' requirements, offline compilation generates the optimal kernel using architecture-independent techniques, such as adaptive batch size selection and coordinated fine-tuning. The runtime management phase consists of accuracy tuning, execution, and calibration. First, accuracy tuning dynamically identifies the fastest kernels with acceptable accuracy. Next, the run-time kernel scheduler partitions the optimal computing resource for each layer and schedules the GPU thread blocks. If its accuracy is not acceptable to the end-user, the calibration stage selects a slower but more precise kernel to improve the accuracy. Finally, we design a user satisfaction metric for CNNs to evaluate our Pervasive deign. Our evaluation results show P-CNN can provide the best user satisfaction for different inference tasks.**

I. Introduction

Recently, deep learning based approaches, especially deep convolutional neural networks (CNNs) [1], have emerged as indispensable tools in many fields, ranging from image and video recognition to natural language processing. This is primarily due to their increasing capability of achieving high accuracy on many challenging machine learning problems. For instance, Microsoft recently announced that its newest deep learning network (PReLU-nets) [2] had outperformed human-level accuracy on a classification task using the 1000-class ImageNet dataset [3]. The improved accuracy comes with significantly increased computation demands. For example, VGGNet [4] (one of the most preferred CNNs in the community of deep learning) requires $1.5x10^{10}$ floating point multiplications per image to perform object recognition. To tackle this challenge, researchers have begun to look to CNN accelerators. The general purpose graphic process unit (GPGPU), with its massively parallel computational capacity, has become the most popular deep learning accelerator. For instance, since 2013, more than 80 percent of the teams participating in the ImageNet image-recognition competition utilized GPUs.

Accelerating CNNs on GPUs usually involves two stages. First, a CNN model is trained on a large-scale labeled training dataset. The training stage, which is mostly time-consuming (e.g., it takes three weeks to train VGGNet on four high-end NVIDIA Titan Black GPUs [4]), is delegated to high-end GPUs. Once complete, the network is deployed to run inference, using its previously trained parameters to classify, recognize, and process unknown inputs. The high accuracy of CNNs has given birth to a large number of artificial intelligence (AI) powered applications, such as DeepFace [5] and Prisma [6]. DeepFace, a deep learning facial recognition system created by Facebook, nears human accuracy in identifying faces. Prisma transforms ordinary photos into the styles of famous artwork. Photos created by Prisma are now taking over Instagram. Normally, these applications send data acquired on mobile platforms to a data center which utilizes high-end GPUs to perform the inference. Recently, with the increasing compute power of mobile GPUs, there is growing interest in performing inference on mobile platforms [7] (e.g. IoT sensors, mobile and smart phones), which enable applications such as Facebook Moments [8] (identifies which friends are in a photo) and Google Translation app (translates food menus and signage in a live view through the phone's camera). As a result, CNN-based applications are becoming pervasive across all GPU platforms. However, due to differences between GPU hardware resources, the optimal configuration of CNNs on one GPU may not be suitable on another. This motivates us to understand the specific implementation of CNNs on GPUs and craft an analytical model to predict the performance of CNNs among different GPU architectures. With a platform-independent analytical model, CNN models trained on high-end GPUs can be efficiently deployed on different platforms without time-consuming retraining.

It is important to note that the optimization goals of these two stages (i.e., training and inference) are different. The goal of the training stage is to obtain high accuracy as soon as possible. The inference stage, however, is closer to the end-user and its goal varies based upon the specific task. Real-time tasks need a small latency to perform inference, such as real-time surveillance and autonomous driving. Background and batch-processing tasks, such as DeepFace deployed in data centers, are not sensitive to runtime, but are more concerned about energy consumption. Some interactive tasks, such as Google Translate app and Prisma, are sensitive to runtime, but can tolerate some delay. In addition to runtime and energy,

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.52

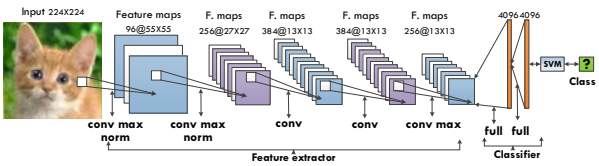

Figure 1. A representative CNN architecture - AlexNet

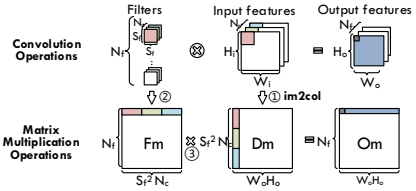

Figure 2. Convolutional operation in a CNN layer

CNN-based applications have a distinctive signature: accuracy. Interestingly, high accuracy is not always preferred in the inference stage, especially when the accuracy of a state-of-the-art CNN can outperform human-level accuracy. For example, some entertainment applications do not need high accuracy, such as Moments. Lowering the accuracy can even improve user experience since the unnoticeable accuracy decrease leads to faster response time. For some applications in security and scientific research, higher accuracy is preferred. Thus, the influence of accuracy on user experience is also determined by the specific task. To evaluate the user experience of the inference, we first investigate the influence of runtime, power, and accuracy on user satisfaction, and then provide a user satisfaction-of-CNN metric (i.e., *SoC*) by combining these three factors.

To ensure ideal user satisfaction on all kinds of platforms, we propose *Pervasive CNN*, a user satisfaction-aware CNN inference framework. P-CNN has two phases: cross-platform offline compilation and run-time management. Based on the specifications of CNN-based applications, P-CNN can infer the requirements of end-users. Then, according to these requirements, offline compilation generates the optimal kernel for the deployed platforms and provides scheduling information for run-time management. The runtime management phase consists of accuracy tuning, execution, and calibration. First, accuracy tuning dynamically identifies the fastest kernels with acceptable accuracy. During this phase, a series of tuning tables are generated for the following calibration. Based on the tuning table, the run-time kernel scheduler partitions the optimal GPU computing resource to each layer and then schedules the GPU thread blocks. As the input data can change during run-time, P-CNN monitors the accuracy of the output. If its accuracy is not tolerated by the end-user, calibration chooses a less aggressive tuning table (corresponds to a slower but more precise kernel) to improve the accuracy of output.

To ensure that the actual response time of CNN-based applications is within the tolerance of end-users, we first develop a platform-independent time model to estimate the execution time of CNNs. We then use this time model to guide cross-platform offline compilation. To generate the optimal kernels for a specific GPU architecture, we first characterize the deep learning libraries to reveal two main tuning parameters: sub-matrix size and register consumption per thread, which dominate the performance of the convolutional kernel. We then design an analytical metric to select the optimal kernel by coordinately fine-tuning these two parameters.

By comparing with multiple run-time scheduler schemes, P-CNN can obtain the best user satisfaction for different inference tasks across various GPU platforms. Evaluation results also show that our entropy-based approximate method achieves the same effect (i.e. 1.8x speedup within 10% accuracy loss) as the accuracy-based method.

In summary, the main contributions of this work are:

- We present Pervasive CNN, a user satisfaction-aware CNN inference framework, to efficiently deploy CNN trained models to all kinds of platforms without time-consuming retraining.
- We propose a platform-independent analytical model (i.e., time and resource model) to guide cross-platform offline compilation.
- We take the first step in helping the computer architecture community understand the inefficiencies of current GPU architectures in supporting CNN-based applications.
- We develop a run-time accuracy tuning technique, which can dynamically change the accuracy of CNNs based on the deployed scenario.

The rest of this paper is organized as follows. Section II introduces background. Section III characterizes deep learning based applications across different GPUs. Section IV describes our Pervasive CNN, including cross-platform offline compilation and run-time management. Section V evaluates our pervasive design. Related work and conclusion are discussed in Section VI and VII, respectively.

II. BACKGROUND

A. CNN

We introduce essential preliminaries of CNN using AlexNet [1] as an example, shown in Fig. 1. It consists of five convolutional layers, three max-pooling layers, and two classifier layers. The convolutional layers perform dot products between the convolutional filters and local regions of the input feature maps. These operations dominate the execution time of CNN computation. The convolutional operations in a convolutional layer benefit from the optimized matrix multiplication libraries. cuBLAS [9] is one such library that implements Basic Linear Algebra Subprograms (BLAS) on top of the Nvidia CUDA runtime library. We demonstrate a typical matrix multiplication based convolutional operation in Fig. 2.

In step 1, an operation called *im2col* stretches out the local regions in the input image into a column-major matrix (D_m). Similarly, in step 2 the weights of the CONV layer are stretched out into a filter matrix (F_m). Then the original convolutional operation can be transferred into a matrix multiplication ($F_m \times D_m$) in step 3. In Fig. 2, the filter matrix F_m has dimensions $N_f \times S_f^2 N_c$, while the data matrix D_m has

dimensions $S_f^2N_c \times W_oH_o$. The output matrix O_m has dimensions $N_f \times W_oH_o$. Therefore, we can calculate the number of floating point operations (i.e., FLOPs) in a convolutional layer through the number of multiply-accumulate operations of $F_m \times D_m$:

$$Conv_{FLOPs} = 2N_f \times S_f^2N_c \times W_oH_o, \quad (1)$$

where a single multiply-accumulate operation counts as 2 FLOPs. The $Conv_{FLOPs}$ is usually used to measure the computational intensity of a convolutional layer.

B. User Satisfaction-of-CNN

Although we know the user satisfaction of CNN is related to runtime, energy, and accuracy, the influence of these three factors on user satisfaction is diverse among different tasks. To accurately reflect how these three factors determine user satisfaction, we classify CNN-based applications into three classes: interactive task, real-time task, and background task. First, we analyze the influence of runtime and energy on user satisfaction using these three classes respectively. Then, we discuss the distinctive factor of CNN-based applications (i.e., accuracy) in detail.

1) Interactive task

As most interactive tasks are user-facing, such as AI-powered mobile apps (e.g., Prisma and Google translate), they are sensitive to runtime. If the app's response time exceeds the human acceptable limits, users may abandon the app. However; users can also tolerate some delay since they understand that AI-powered applications involve complex computations. We illustrate the relationship between user satisfaction, runtime, and energy consumption in Fig. 3. Runtime increases from left to right on the x-axis. Previous work [10] indicates that the user satisfaction of interactive applications can be classified into three distinct states as application runtime increases: imperceptible $(0, T_i]$, tolerable $(T_i, T_t]$, and unusable (T_t, ∞).

In the imperceptible region, runtime increase will not jeopardize user satisfaction, yet can achieve more energy savings. Therefore, there is no need to achieve the fastest response time in this region. Instead, we should try to minimize energy consumption by lowering the performance so that runtime is close to T_i. Beyond T_i, user satisfaction enters the tolerable region, where user satisfaction deteriorates as the user starts to notice the slow response time. However, the runtime is still acceptable in this region. As the runtime increases further, user satisfaction is eventually violated at T_t, where the user deems the response time unacceptable. At T_t and beyond, users abandon the task. As a result, user satisfaction decreases to 0 in the unusable region. We should try to avoid running the task in this region.

2) Real-time task

Real-time tasks have a strict runtime requirement. For example, pedestrian detection in self-driving vehicles has a hard deadline and missing the deadline can lead to a serious accident. Thus, real-time tasks do not have the tolerable region as shown in Fig. 3. Its imperceptible and unusable regions are nearly the same as the interactive task. However, the end of the imperceptible region (i.e., T_i) should be less than the deadline [11].

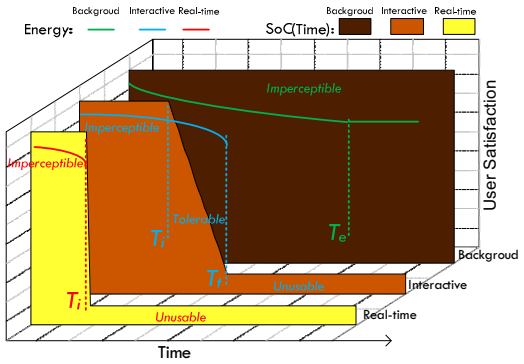

Figure 3. User satisfaction of CNN-based applications

3) Background task

Background tasks are normally not sensitive to the runtime since the user does not expect the results for a longer duration of time. Instead, these tasks are concerned more about energy consumption. For example, after taking many photos, users will switch to other apps, such as web browsing, while Moments is identifying which friends are in these photos. Although users do not need to send the photos to their friends immediately, they pay more attention to the power consumption of Moments since they desire a long battery life. Therefore, the runtime of background tasks does not affect user satisfaction. The entire region in Fig. 3 for a background task is the imperceptible region. Note that the energy consumption first decreases then plateaus as the runtime increases. This is because energy consumption is the product of power and runtime. At T_e and beyond, the decrease in power is offset by the increase in runtime.

4) Accuracy of CNN-based applications

Besides runtime and energy, CNN based applications have a distinctive signature: accuracy. Interestingly, high accuracy is not always preferred in inference, especially when the accuracy of a state-of-the-art CNN can outperform human-level accuracy. For example, some entertainment applications do not need high accuracy, such as Moments. Lowering the accuracy can even improve user experience since the unnoticeable accuracy decrease can lead to faster response time. In this case, it is important to find the optimal accuracy. Low accuracy will affect the user satisfaction while high accuracy will lead to a heavier computational burden.

Traditionally, researchers use a labeled test dataset to evaluate the accuracy of CNN-based applications. However, it is impossible to create a labeled dataset that encompasses all applications. One viable option is to enable self-diagnosis capability within CNN based applications. During runtime, they can leverage their outputs to diagnose themselves. Due to the lack of a labeled test dataset, we use network uncertainty [12] instead of accuracy as the metric. Here we take a classification task as an example. Each input data should belong to a certain class denoted by $Y \in \{1, \cdots, k\}$. However, we do not know true class labels during run-time. The output of a CNN classifier layer (Y) is a probability distribution P. For example, P (0.6, 0.3, 0.1) means the probability of output belonging to class1 is 60% and 30% of output is class2. We

TABLE I. ACCURACY VS ENTROPY

CNNs	AlexNet [1]	VGGNet [4]	GoogLeNet [13]
Accuracy	79.4%	86.6%	88.5%
Entropy	1.05	0.88	0.83

TABLE II. GPU CONFIGURATIONS

GPU	Platform	CUDA Cores	Memory
K20c	Server	2496 CUDA cores, 706 MHz	5GB, 2600MHz, 320-bit GDDR5
Titan X	Desktop	3072 CUDA cores, 1000 MHz	12GB, 7Gbps, 384-bit GDDR5
GTX 970m	Notebook	1280 CUDA cores, 924 MHz	3GB, 2500MHz, 192-bit GDDR5
Jetson TX1	Mobile	256 CUDA cores, 998 MHz	4GB LPDDR4, 25.6GB/s

then use the entropy of Y (i.e., $CNN_{entropy}$) to measure the uncertainty of the output. The discrete entropy of Y can be estimated by

$$H(Y) = -\sum_{i=1}^{k} p_i \log(p_i). \quad (2)$$

Higher values of entropy imply more uncertainty in the distribution P. For example, the entropy of P1 (0.4, 0.4, 0.2) is higher than P2 (0.7, 0.2, 0.1) since P1 is highly confused between class1 and class2 and P2 is more confident that the output belongs to class1. Higher uncertainty (i.e., higher entropy) further indicates low accuracy. As shown in Table I, the accuracy of the network improves with the decrease of network entropy. Therefore, we leverage network entropy (i.e., $CNN_{entropy}$) as the metric of accuracy to judge whether the output of a CNN is reliable or not.

III. CHARACTERIZATION

A. Experimental Setup and Methodology

To characterize the actual deployment of CNNs and cover all possible application scenarios, we deploy three ImageNet winners' models (AlexNet [1], GoogLeNet [13], and VGGNet [4]) to four different platforms (Server, Desktop, Notebook, and Mobile). The detailed parameters are listed in Table II. We gather GPU runtime information using Nvidia Visual Profiler [14]. The CNN models are trained by Caffe [15], an open-source deep learning framework widely used in both academia and industry. We characterize three state-of-the-art deep learning libraries: cuBLAS (adopted by Caffe), cuDNN [16] (the leading industrial library developed by Nvidia) and Nervana [17] (the fastest implementation), on the acceleration of CNN-based applications. The test data is selected from ImageNet [3].

B. Inference Prefers a Small Batch Size

For the training stage, it is common to batch hundreds of training inputs and process them simultaneously. This batching method not only prevents overfitting but also significantly improves GPU efficiency by amortizing the overhead of loading weights from GPU memory. For example, Karen et al. [4] train VGGNet using stochastic gradient descent with a batch size of 256. Though the batching method also benefits the processing throughput of the inference, it significantly increases network latency (i.e. network's end-to-end response time), especially on mobile platforms. Table III compares different networks' latencies (millisecond) w/ and w/o the batching method across three platforms. Considering the limited computing capacity of mobile GPUs, we opt to a smaller batch size than that used in training: 128 for AlexNet, 64 for GoogLeNet and 32 for VGGNet. As shown in Table III, though Nervana achieves the highest performance among the three state-of-the-art deep learning libraries, it still takes about 400ms to run the naive CNN (i.e., AlexNet) on TX1 using the small batch size, which is far from the real-time requirement of end users. Even worse, cuDNN and Nervana fail to run GoogLeNet and VGGNet on mobile GPUs with the batching method due to out-of-memory issues (marked as x in Table III).

TABLE III. LATENCIES W/ AND W/O BATCHING

CNNs	GPUs	Batching			No-Batching		
		cuBLAS	*cuDNN*	*Nervana*	*cuBLAS*	*cuDNN*	*Nervana*
AlexNet	TitanX	131	68	31	3	3	**15**
	970m	301	171	77	5	5	**38**
	TX1	1269	1183	397	25	24	**216**
GoogLeNet	TitanX	381	118	48	10	17	**43**
	970m	585	301	121	12	18	**107**
	TX1	2622	x	611	64	67	**527**
VGGNet	TitanX	165	152	164	8	10	**162**
	970m	457	407	405	20	23	**410**
	TX1	2428	x	x	104	116	x

What is more, there is not enough available data generated during the tolerant response time of end-users. For example, most interactive tasks only submit one request each time, and then wait for a response. Therefore, inference prefers a smaller batch size due to fast response time requirements and limited available input data, especially for real-time and interactive tasks.

C. Inefficiency of Current GPU Implementation

Although inference prefers a smaller batch size to obtain better user satisfaction, the highly optimized deep learning libraries, such as Nervana and cuDNN, do not perform well on the smaller batch size. For example, the batch size of Nervana must be a multiple of 32. For this reason, non-batching for Nervana (i.e., font-bold in Table III) corresponds to its supported minimum batch size: 32. With its supported minimum batch size, the smallest latency of GoogLeNet on TX1 is still more than 500ms. Although the latency of the non-batching method on laptop and desktop platforms can meet the requirements of end users (e.g., <30ms), their throughputs (i.e. the number of processed images per second) are not optimal. Fig. 4 shows the ratio of throughput without batching to that of using the batching method. The ratios are below 50% for cuDNN among various platforms.

As a CNN consists of multiple convolutional layers, we delve deeper into the convolutional layers to find the cause of low throughput. To better quantify the performance of CNN convolutional layers, we define computation efficiency, *cpE*, which indicates the ratio of the achieved throughput to the peak throughput of GPU in each convolutional layer:

TABLE IV. DETAILED INFORMATION OF CNN DOMINATED KERNELS

GPU	Library	COV Layer	Result-matrix	Sub-matrix	Reg-ister	Shared Memory	Block Size	#blocks (register)	#blocks (shmem)	maxBlocks	Grid Size
TX1	*cuBLAS*	CONV2	128x729	128x64	120	12544	128	8	14	min(14, 8)=8	12
		CONV5	128x169	128x64	120	12544	128	8	14	min(14, 8)=8	4
	cuDNN	CONV2	128x729	32x32	48	2304	64	40	84	min(84, 40)=40	92
		CONV5	128x169	32x32	48	2304	64	40	84	min(84, 40)=40	24
K20	*cuBLAS*	CONV2	128x729	64x64	79	8468	256	39	65	min(65, 39)=39	24
		CONV5	128x169	64x64	79	8468	256	39	65	min(65, 39)=39	6
	cuDNN	CONV2	128x729	64x64	79	8468	256	39	65	min(65, 39)=39	24
		CONV5	128x169	64x64	79	8468	256	39	65	min(65, 39)=39	6

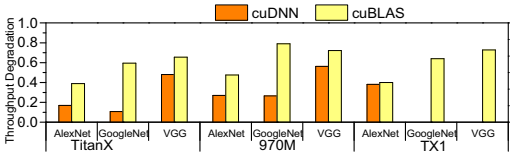

Figure 4. The ratio of throughput w/o batching to batching

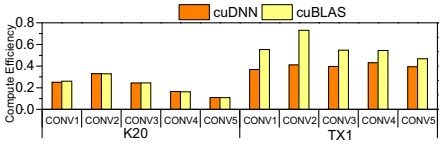

Figure 5. Compute efficiency among CONV layers

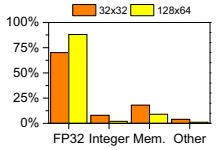

Figure 6. Inst. breakdown

$$\text{cpE}=\frac{\text{Throupghput}}{\text{GPU peak througput}}=\frac{\text{Conv}_{\text{flops}}/\text{t}_{\text{conv}}}{2\cdot\text{GPUfreq}\times\text{nSMs}\times\text{nCores}}. \quad (3)$$

In (3), the actual throughput in each convolutional layer is the number of FLOPs per second. As each core is capable of executing 1 multiply-accumulate (2 FLOPs) per clock cycle, the GPU peak throughput is two times the product of GPU frequency, the number of Streaming Multiprocessors (i.e., SMs) and number of CUDA cores per SM.

Fig. 5 shows the *cpE* of AlexNet using cuBLAS and cuDNN on a data center GPU (i.e., K20) and mobile GPU (i.e., TX1). On the data center platform, *cpE* is less than 35%, especially for the last two convolutional layers (<15%). This low *cpE* even exists in the mobile GPU. The average *cpE* of cuDNN on TX1 is about 40%. Next, we choose two representative layers, CONV2 (the layer with largest computational load) and CONV5 (the layer with least *cpE*), to further analyze the bottlenecks of convolutional layers.

Table IV highlights the detailed kernel information of these two layers. Since the convolutional operation is implemented by matrix multiplication, the convolutional kernel is a Single Precision Floating General Matrix Multiply (i.e., SGEMM). The computational procedure of SGEMM is similar to the algorithm developed by Volkov and Demmel [18]. It divides the result matrix (M×N) into sub-matrices, with each sub-matrix (m×n) mapped to a GPU thread block. Therefore, the number of blocks (i.e., GridSize) is equal to the number of sub-matrix, which is determined by the sizes of result matrix and the sub-matrix:

$$\text{GridSize}=\left\lceil\frac{M}{m}\right\rceil\times\left\lceil\frac{N}{n}\right\rceil. \quad (4)$$

In Table IV, SGEMM is a register-intensive kernel, which consumes more than 48 registers per thread (i.e., *r*). Therefore, the register resource (i.e., *R*) largely determines the number of blocks (i.e., maxBlocks) that can be simultaneously executed on GPU:

$$\text{maxBlocks}=\text{nSMs}\times\left\lfloor\frac{R}{\text{block size}\times r}\right\rfloor. \quad (5)$$

With small batch sizes, GridSize is less than maxBloks, which leads to underutilization of GPU resources. Resource underutilization is a severe problem if end-users are sensitive to response time. To better quantify the degree of resource underutilization, we define *Util* as:

TABLE V. UTILS OF ALEXNET

Size	CONV1	CONV2	CONV3	CONV4	CONV5
K20	0.82	0.62	0.46	0.23	0.15
970m	0.6	0.3	0.3	0.15	0.1
TX1	1	0.75	0.75	0.75	0.5

$$\text{Util}=\frac{\text{GridSize}}{\text{nCycle}\times\text{maxBlocks}}=\frac{\text{GridSize}}{\left\lceil\frac{\text{GridSize}}{\text{maxBlocks}}\right\rceil\times\text{maxBlocks}}. \quad (6)$$

Table V shows the *Util* of AlexNet across different GPU platforms with the non-batching method. Resource underutilization even exists in the mobile GPU (i.e., TX1) that has the least computing resources. Moreover, different layers have various *Utils*, which demands a per-layer optimization. Low *Util* not only leads to lower processing throughput but also challenges current GPU thread blocks (also called Cooperative Thread Arrays, CTAs) scheduler.

CTAs are assigned to each SM in a Round-Robin (i.e., RR) manner, and each SM is allocated to the maximum number of CTAs [19][20]. The maximum number of CTAs that can be assigned to each SM depends on kernel resource usage, including registers, shared memory, etc. Once a CTA completes, the CTAs scheduler assigns another CTA to the just-freed SM until all CTAs have been assigned. However, the RR CTA scheduler is not efficient for layers with low *Util*. We illustrate this situation in Fig. 7. The GPU consists of four SMs, and a kernel has four CTAs. The optimal number of CTAs for each SM (i.e., *optTLP*) is 2. Fig. 7 demonstrates two CTA schedulers: Priority-SM (i.e., PSM) and Round-Robin (i.e., RR). PSM first allocates 2 CTAs to the first SM, and then assigns the remaining 2 CTAs to the second SM. In this way, the remaining two SMs can be released to run other kernels or powered off to save energy. For RR, four SMs are occupied by four CTAs respectively. Therefore, PSM is better than RR since it achieves nearly the same performance with half the SM computing resources.

D. Current Methods to Improve Util

1) Small sub-matrix size

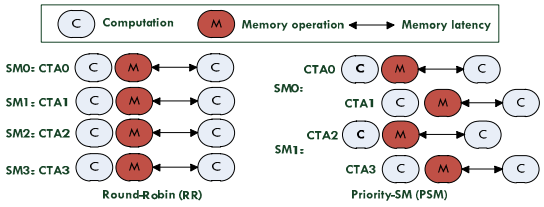

Figure 7. RR vs PSM

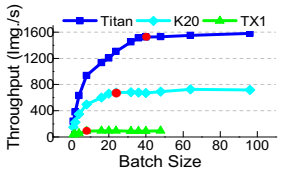

Figure 8. Commuting Throughput

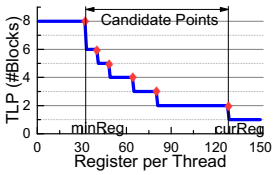

Figure 9. TLP Vs Registers

Although cuDNN adopts a small sub-matrix size (i.e., 32x32) in TX1 to increase resource utilization, its performance is still inferior to cuBLAS, as shown in Fig. 5. This is because the small sub-matrix in cuDNN leads to reduced register consumption as shown in Table IV, which will further incur more global memory transactions and decrease the computation density. Fig. 6 illustrates the comparison of computation density: ratio of floating point instructions to the total instructions among different sub-matrix sizes. Therefore, there is a trade-off between higher resource utilization (i.e., smaller sub-matrix) and higher computation density (i.e., bigger sub-matrix). The actual performance of a layer is determined by the size of sub-matrix and register usage. We can obtain the optimal throughput by using cross-platform offline compilation to fine-tune the sub-matrix size and register consumption per thread in each convolutional layer across different platforms.

2) Multi-process mechanism

Although a multi-process mechanism, such as MPS [21], can improve GPU resource utilization, it has no control over how the TBs (i.e., Thread Blocks) are dispatched. It is possible that more TBs are allocated to other concurrent tasks using MPS. Therefore, there is no guarantee on runtime for time-sensitive CNN-based applications. Note that although techniques such as Spatial-Multitasking GPUs [22], could satisfy the end user's requirement by dedicatedly allocating enough SMs to CNN-based applications, the diversity in *Util* challenge such approaches. If we allocate the maximum number of SMs (i.e., SM resource used in the layer with the highest *Util*) for all the layers, there is still resource underutilization for the layers with low *Util*. Instead, we should allocate SMs according to the *Util* in each layer. Specifically, we first try to find the optimal number of SMs to meet the run-time requirement and then leverage the spatial partition technique to allocate optimal SMs to each layer and power gate the otherwise-idle SMs to reduce energy consumption.

3) Big batch size

For background tasks, we can improve GPU utilization by increasing the batch size. However, a batching method with a large batch size easily runs out of GPU memory because CNN-based applications are memory-intensive [23]. We should pay attention to the batch size selection. When GridSize is equal to maxBlocks, the GPU resource is fully utilized and the batch size at this point is optimal. However, due to the variation of hardware resource (e.g. *nSMs* and *R*), different platforms have various maxBlocks, which causes the optimal batch size to vary among different GPU platforms (marked as red in Fig. 8). Worse, since the network parameters are different for each layer of a CNN, such as W_oH_o and N_f, various layers have different optimal batch sizes even for the same platform. Therefore, it is imperative for us to craft an analytical model to predict the performance of CNNs among different GPU architectures. With this model, we can easily infer the optimal batch size for each CNN-based application on different GPU platforms.

IV. PERVASIVE CNN

We propose Pervasive CNN (P-CNN) to efficiently deploy CNN-based applications to different platforms while maintaining the highest user satisfaction. As discussed in Section II, response time, energy consumption, and accuracy determine user satisfaction. Among these three factors, end-users are more sensitive to response time and accuracy. Thus, our scheduling policy gives time and accuracy the highest priority. Having satisfied the requirements on response time and accuracy, P-CNN tries to save energy by optimally using the GPU resources. The optimal GPU resource can be derived by a resource model. To ensure the response time is under the user's tolerance, P-CNN features a time model to predict the response time. Based on the predicted performance, we can adjust the response time during offline compilation. To allow the accuracy of CNN-based applications to dynamically adapt to real-world scenarios, P-CNN performs accuracy tuning during the run-time phase. Therefore, our P-CNN framework consists of two main steps: cross-platform offline compilation and run-time management, as shown in Fig. 10. Based on the deployed GPU architecture and users' requirements, offline compilation generates the optimal kernel for each layer and provides scheduling information (i.e., scheduling configurations), including the optimal number of occupied SMs (i.e., *optSM*) and *optTLP* of each layer, for the run-time management. The run-time management phase consists of accuracy tuning, execution, and calibration. Based on the scheduling configurations, accuracy tuning dynamically identifies the fastest kernels with acceptable $CNN_{entropy}$ and generates a series of tuning tables for the following calibration. During the tuning and execution phase, the run-time kernel scheduler partitions the optimal SM resource to each layer. As the input data can change during run-time, P-CNN monitors the uncertainty of the output. If its uncertainty is larger than the threshold, calibration chooses a less aggressive tuning table to improve the output accuracy. Next, we introduce our P-CNN in detail.

A. User Input

The user input module is responsible for providing the deployed GPU architecture, application specifications (e.g., input data generation rate) and user requirements to our P-

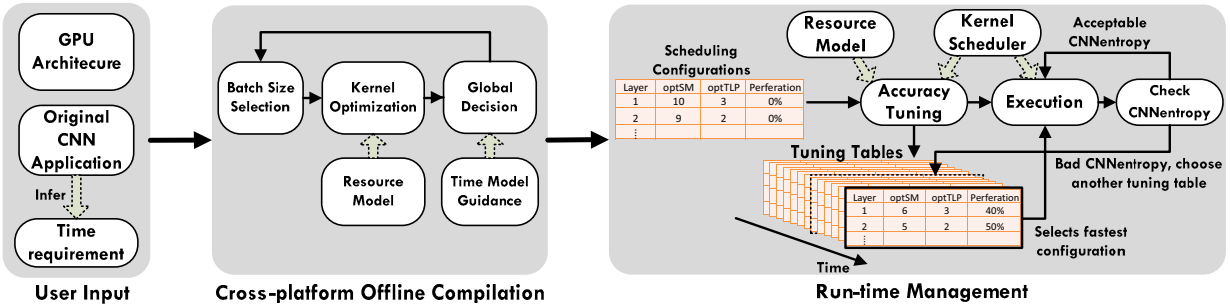

Figure 10. An overview of Pervasive CNN framework

CNN. However, it is annoying for end-users to clearly describe their requirement each time they submit a request. Instead, we design an inferring mechanism to infer end-users' requirements automatically. Specifically, we first classify the application to different tasks, such as real-time task, background task, and interactive task, based on the application specifications. Then we write a look-up table to record the time requirement for each task. The time requirement can be derived from some statistical information of user experience in research reports [24][25]. In the future, we can create a more fine-grained time requirement table for each user using machine learning techniques to learn user experience. Finally, we can use the task type of the application to look up this table and find the requirement for each application.

B. Cross-platform Offline Compilation

1) Batch size selection

a) Background task: Since a background task does not care about response time, we can use the batching method to achieve maximum processing throughput and reduce energy consumption. However, we should pay attention to the batch size selection since CNN-based applications are memory-intensive and big batch sizes can easily run out of GPU memory. Therefore, the optimal batch size should be the minimum batch size, which allows all the layers to fully utilize the GPU resource. As shown in Table V, the last layer manifests the minimum *Util*. Therefore, the optimal batch size should ensure the *Util* of the last layer equals 1. When the batch size is more than the optimal batch size, the processing throughput can not be further improved since all the computing resources have been utilized.

b) Other tasks: Since these tasks (e.g., real-time and interactive tasks) are sensitive to response time and there is not enough available data generated during the time requirement (i.e., T), we set the initial batch size as the number of available data generated during T (i.e., T/data generate rate).

2) Kernel optimization

After the initial batch size is determined, we start to optimize the kernel for each layer. As discussed in Section III.D, the kernel's performance is mainly determined by sub-matrix size and register consumption per thread. For CNNs, the common sub-matrix sizes are 128×128, 128×64 and 128×32 [17]. The selection of sub-matrix not only determines the number of blocks (i.e., GridSize) but also affects the register usage per thread (i.e., *curReg*). For register-intensive kernels, such as SGEMM, reducing the register usage can increase thread-level parallelism (i.e., TLP) as shown in Fig. 9. The minimum register usage per thread (i.e., *minReg*) is determined by the ratio of register resource of SM to the maximum number of threads supported by SM. Therefore, the range of register adjustment is from *minReg* to *curReg*.

To achieve maximum performance, we should coordinately fine-tune sub-matrix size and register per thread. Fig. 9 shows the impact of varying register count per thread on TLP with the sub-matrix size (i.e., 128×128) on K20. Here *curReg* is 127 and *minReg* is 32. In Fig. 9, each stair consists of a few design points with the same TLP but different numbers of registers per thread. Within the same TLP, the design with the most register usage per thread has the best performance. Therefore, we only need to consider the rightmost point on each stair (marked in red color). With the pruned design space, we can easily explore the performance of the candidate points. Here we use the register spilling technique in [26] to decrease the number of registers to these candidate points.

Although less register consumption means high TLP, the single-thread performance will be affected because more local variables are spilled to the slower global memory. Considering that kernels of convolutional layers fail to utilize all the shared memory space and the shared memory is faster than the global memory, we first leverage the spare shared memory instead of global memory for register spilling. Note that we only utilize the spare shared memory for spilling so that the TLP is not decreased. After the spare shared memory is exhausted, we continue to spill the remaining registers to global memory. During register spilling, the overhead (i.e., $Spill_{cost}$) due to the extra global and share memory accesses triggered by register spilling is:

$$Spill_{cost}=N_{global}\times Cost_{global}+N_{shm}\times Cost_{shm}+N_{others}, \quad (7)$$

where N_{global}, N_{shm}, and N_{others} represent the number of inserted global memory instructions, shared memory instructions, and extra instructions (used for address computation for spilling) respectively. $Cost_{global}$ and $Cost_{shm}$ represent the delay of per access to the global memory and shared memory respectively.

Next, we model the influence of improved TLP on the performance of the kernel. Combining the selection of sub-matrix size and TLP, we define a metric, *nInvocation*, to

quantify the number of invocations needed to process a convolutional layer:

$$\text{nInvocations}=\left\lceil\frac{\text{GridSize}}{\text{TLP}\times\text{nSMs}}\right\rceil. \quad (8)$$

In (8), the GPU needs fewer invocations to process a convolutional layer by increasing TLP and sub-matrix size (m·n). Moreover, considering the remainder in (4), we define a metric, *rEC*, to evaluate the ratio of effective computation to overall computation with different sub-matrix sizes:

$$\text{rEC}=\frac{\text{M}\times\text{N}}{\left\lceil\frac{\text{M}}{\text{m}}\right\rceil\times\left\lceil\frac{\text{N}}{\text{n}}\right\rceil\times\text{m}\times\text{n}}. \quad (9)$$

Since the performance of GPU kernels depends on the single thread performance and the TLP, we use an analytical metric S_{kernel}, which involves both to capture the performance tradeoff:

$$\text{S}_{\text{kernel}}=(1\text{-rEC})\times\text{Spill}_{\text{cost}}\times\text{nInvocations}. \quad (10)$$

We compare S_{kernel} of different design points and select the one with the smallest value. The TLP with the smallest S_{kernel} corresponds to *optTLP*.

3) Global decision using analytical models

Resource model: Due to resource underutilization in the inference, there is no need to use all the SMs to perform the inference. In Fig. 10, the run-time kernel management needs two factors: *optSM* and *optTLP* for each layer, to do the scheduling. Section IV.B.2 has discussed the selection of *optTLP*. Next, we introduce how to choose *optSM* to maximize the efficiency of SMs. We can achieve this goal by utilizing fewer SMs. At the same time, we want to ensure that the *nInvocations* is not increased as follows:

$$\left\lceil\frac{\text{GridSize}}{\text{optTLP}\times\text{optSM}}\right\rceil=\left\lceil\frac{\text{GridSize}}{\text{optTLP}\times\text{numSMs}}\right\rceil. \quad (11)$$

Note that the *optSM* is chosen as the minimum value to satisfy (11). For example, if GridSize is 40, *opTLP* is 3 and the total number of SMs in GPU is 10, the minimum value (i.e., *optSM*) in (11) is 7 and we can release 3 SMs to perform other tasks. After finding the optimal configurations (e.g., *optSM* and *optTLP*), the offline compilation of a background task is complete since it does not care about response time. For real-time and interactive tasks, we need to further check whether the response time meets the user's requirement or not. Next, we craft a time model to predict the performance and use the predicted response time to guide the adjustment of batch size.

Time model: After determining the *optSM* in each layer, we can derive a time model to calculate the processing time of a given layer as:

$$\text{t}=\frac{\text{Conv}_{\text{flops}}\times\text{Batch Size}}{\text{peakFlops}\times\text{optSM}\times\text{rCE}\times\frac{\text{FMA Insts}}{\text{Total Insts}}}, \quad (12)$$

where *FFMA Insts* and *Total Insts* represent the number of fused multiply-add instructions and total instructions, respectively. *peakFlops* can be approximated by the twice product of the GPU frequency and number of CUDA cores per SM. Finally, the total time of a CNN (i.e., *T*) is the sum of time spent on each layer.

Global decision: After obtaining the estimated runtime, we can compare it to the user's requirement of time (i.e., T_{user}). If T_{user} is larger than *T*, the offline compilation is over. Otherwise, we need to decrease the batch size to reduce *T*. The new batch size is:

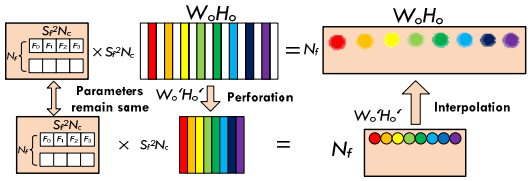

Figure 11. Perforation-interpolation based approximation

$$\text{new Bach size}=\frac{\text{T}_{\text{user}}}{\text{T}}\text{Batch size}. \quad (13)$$

Due to modified batch size, the computational load in each layer has also changed. We need to optimize the kernel of each layer from Section IV.B.2 again.

C. Run-time Management

1) Entropy-based accuracy tuning

As discussed in Section II.B, accuracy is a distinctive signature for CNN-based applications. Although state-of-the-art CNNs can achieve greater than human-level accuracy, there is no need to always run the CNN-based applications with the highest accuracy. Lowering the accuracy even can improve user experience since the unnoticeable accuracy degradation leads to faster response time. Therefore, it is important for us to find the optimal accuracy (i.e., *optAccuracy*, not the highest accuracy) for each CNN-based application. However, different applications will have various requirements about accuracy. Even for the same task, different application scenarios may lead to varying *optAccuracy*. For example, face detection deployed around a busy square needs higher accuracy than that in a quiet room. As the training of a CNN is a time-consuming procedure, it is impossible to train a new CNN model for each application to meet the diverse *optAccuracy*. Instead, we introduce an entropy-based approximate method to tackle this challenge and make the accuracy of a CNN dynamically adapt to the different scenarios during run-time.

Since CNNs mainly focus on image or video processing where neighboring pixels tend to have similar values [27], there is spatial redundancy in CNNs. Modern CNNs, such as AlexNet, exploit this redundancy through the use of strides in the convolutional layers [1]. Although increasing the convolutional strides can reduce the computational costs of CNNs, it also alters the intermediate representations size and number of weights in the first fully-connected layers, which requires retraining and is therefore undesirable. Here we still leverage spatial redundancy to reduce the computational load. Instead of increasing strides to reduce the output of convolutional layers, we directly sample the output of convolutional layers (i.e., *perforation*) and interpolate the remaining values with the neighboring values (i.e., *interpolation*), as shown in Fig. 11. The perforation reduces the number of operations required for the matrix multiplication using a perforation rate ($1\text{-}W_o'H_o'/W_oH_o$) and the interpolation ensures the architecture of CNNs remains unchanged. The number of sampled points ($W_o'H_o'$) play an important role in this approximation. Smaller $W_o'H_o'$ will lead to reduced runtime, but also lowers the accuracy of CNNs. Therefore, we should find a balance between runtime and

accuracy. Moreover, various layers affect CNNs' runtime and accuracy differently. To quantify the tradeoff between runtime and accuracy, we define a metric, *TE*, to select the layer achieving a high speedup with low error:

$$TE=\frac{T_{ori}-T_{Layer\ i}}{CNNentropy_{Layer\ i}-CNNentropy_{ori}}. \quad (14)$$

Since we adjust the accuracy during run-time, we do not have the labeled data to evaluate the accuracy. Instead, we use the network uncertainty (i.e., $CNN_{entropy}$) to depict the accuracy. As various layers affect CNNs' runtime and accuracy differently, we employ a simple greedy strategy to perform the accuracy tuning in Fig. 12. During each adjustment, we only sample one layer's outputs ($W_o^{'}H_o^{'}$) and keep other layers untouched. After all the layers complete the adjustment (this procedure is named an iteration), we choose the layer with the maximum *TE* as the optimal adjustment in this iteration. To maximize the ratio of effective computation to overall computation (i.e., *rEC*), we make sure that $W_o^{'}H_o^{'}$ (which corresponds to N in (9)) is a multiple of *n* (one dimensional size of sub-matrix). As shown in Fig. 12, since Layer2 has the maximum *TE*, the perforation rate becomes [0, 0.1, 0, …, 0] after iteration 1. After obtaining the optimal adjustment, we can create a new tuning table for this iteration using our resource model. If the error is still within the tolerance of the user, we continue to approximate the network using a smaller $W_o^{'}H_o^{'}$ in the next iteration. This process will continue until tuning finds the output uncertainty is bigger than the defined threshold.

Note that the tuning phase does not introduce overheads since the offline compilation has ensured the response time is under the user tolerance and the output of tuning phase will not be abandoned by end-users.

2) Run-time kernel management

To reduce the overhead of the run-time scheduler, the scheduling information is directly obtained from the tuning tables, including *optSM* and *optTLP*. For each layer, *optSM* SMs are allocated to perform convolutional operations and the remaining SMs (*maxSM-optSM*) are power gated to save energy. Our scheduler first allocates *optTLP* CTAs to the first SM, and then allocates another *optTLP* CTAs to the second SM until *optSM* SMs have been occupied. Once a particular CTA completes on an SM, the CTAs scheduler assigns another CTA to this SM, until all CTAs have been assigned.

3) Calibration

As the input data can change during run-time, P-CNN monitors the uncertainty of the output. If its uncertainty is larger than the threshold, P-CNN switches to a slower but more precise version of the CNN. These decisions are based on the tuning path previously described above. By backtracking along the tuning path in Fig. 12, P-CNN finds a more accurate kernel and reduces the output uncertainty. This process will continue until the output uncertainty is less than the pre-defined threshold.

V. EVALUATION

In this section, we use real GPU-based systems and a simulator based framework to comprehensively evaluate our P-CNN design. Our simulator framework is implemented based on GPGPU-Sim [28]. GPUWatch [29] is used to measure the energy consumption. We first introduce our metric, Satisfaction of CNN (i.e., *SoC*) for evaluating our P-CNN. We then describe the baseline schemes, which only consider one or two factors when scheduling CNN-based applications, such as an energy-efficient scheduler, performance-preferred scheduler and QoS per energy (i.e., QPE) scheduler [10]. Next, we compare them with our SoC-oriented P-CNN and present a comprehensive evaluation of our P-CNN on three important scenarios across different platforms. We also justify the procedure of entropy-based accuracy tuning using labeled test data.

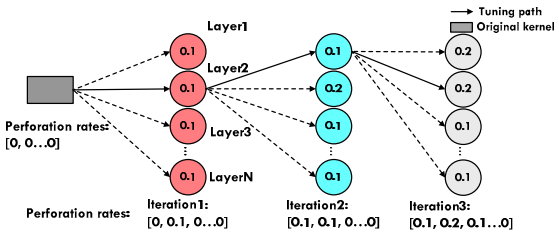

Figure 12. Accuracy tuning procedure

A. Metric for Evaluating User Satisfaction-of-CNN

To evaluate how P-CNN benefits user satisfaction of CNN-based applications, we propose a new metric, Satisfaction-of-CNN (i.e., *SoC*):

$$SoC=\frac{SoC_{time}\times SoC_{accuracy}}{Energy\ Consumption}. \quad (15)$$

SoC_{time} is a measurement of how response time affects user satisfaction as discussed in Section II.B. Imperceptible SoC_{time} is defined as 1. Unusable SoC_{time} is defined as 0. In the tolerable region, SoC_{time} degrades linearly with the response time [30]. We use $SoC_{accuracy}$ to describe the influence of the accuracy on user satisfaction. Here we leverage network uncertainty (i.e., $CNN_{entropy}$) to depict the accuracy. If the $CNN_{entropy}$ is smaller than the pre-defined threshold (i.e., $CNN_{threshold}$), $SoC_{accuracy}$ is 1. Otherwise, $SoC_{accuracy}$ is defined as $CNN_{threshold}/CNN_{entropy}$.

B. Baseline Schedulers

We compare our P-CNN with the following five baseline schemes:

1) Performance-preferred scheduler: To obtain fast response time, this scheme normally employs a non-batching method to perform the CNN-based applications.

2) Energy-efficient scheduler: This scheme only considers energy consumption. It leverages the batching method in the training phase. With the high processing throughput from the big batch size, it can consume less energy to perform the task. It is same as our P-CNN without accuracy tuning when processing background tasks.

3) QoS per energy (QPE) scheduler: This scheme takes response time and energy consumption into account when scheduling CNN-based applications. It attempts to consume the least energy under a given runtime requirement.

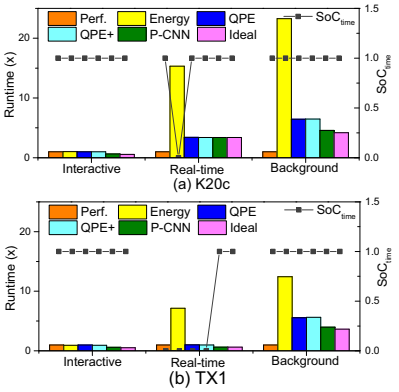

Figure 13. Norm. runtime and SoC_{time}

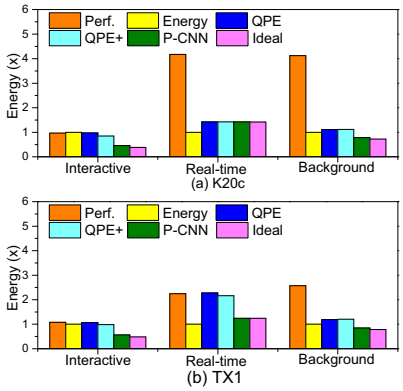

Figure 14. Norm. energy

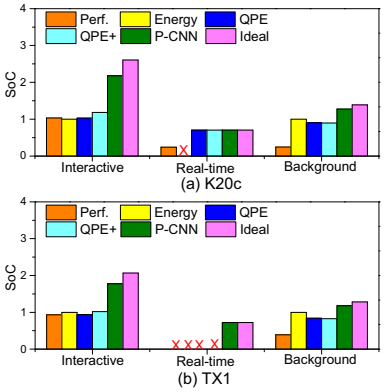

Figure 15. Satisfaction of CNN (i.e., *SoC*)

4) QPE+ scheduler: Compared with QPE, QPE+ can utilize the optimal computing resource to do the inference. It can save energy when *Util* is low. QPE+ is similar to our P-CNN without accuracy tuning.

5) Ideal scheduler: It is the oracle scheduler that has a priori knowledge of all the requirements of end-users. It can obtain the maximum *SoC* by profiling all the tuning points.

C. Comprehensive Evaluations of P-CNN

We evaluate various schedulers under three typical tasks: age detection (interactive task), video surveillance (real-time task) and image tagging (background task). After a user submits a selfie, age detection could estimate his/her age using CNN. Due to the limited available data, this interactive task normally takes the non-batch method. As human-computer interaction research shows that users can tolerate 100ms latency for certain interactive tasks [31] and abandon an app that takes more than 3s to load [32], Ti and Tt are defined as 100ms and 3s respectively for the age detection task. For the real-time video surveillance task, the deadline to process a frame is the frame rate. For example, the deadline is 1/60s if the frame rate of surveillance video is 60FPS. For the background image tagging, there is no timing restriction.

We compare these five schedulers using the latest version of GPGPU-Sim. The simulation parameters in Table VI are from the specifications of Nvidia K20c and TX1. As discussed in Section II.A, the convolutional operation is implemented by matrix multiplication. Therefore, we run the kernel of matrix multiplication (i.e., sgemm [33]) to perform the convolutional operations in GPGPU-Sim.

We first compare the runtime and energy consumption of P-CNN against other schedulers. We then use *SoC* to summarize our P-CNN in user satisfaction. Fig. 13, 14 and 15 show the runtime, energy consumption, and *SoC* scores under different scheduling schemes respectively. To compare with the baseline scheduler that performs the best in a certain metric, the runtime is normalized to the Performance-preferred scheduler while energy consumption is normalized to the Energy-efficient scheduler.

We first discuss the results of K20c. As our time model can accurately predict the response time, the response times of the three applications are within the imperceptible region

TABLE VI. SIMULATION PARAMETERS FOR GPGPU-SIM

Parameters	K20c	TX1
SM	13 SMs, 706MHz	2 SMs, 998MHz
Register	64Kx32bit, 16 banks	
TLP Limitation	16 CTAs, 2048 Threads	
Shared Memory	48KB, 32 banks	
Warp	32-threads, GTO warp scheduler	

for the schedulers equipped with a time model (i.e., all the schedulers except for energy-efficient scheduler). Therefore, all the schedulers achieve the maximum SoC_{time} except energy-efficient scheduler in Fig. 13(a). For the real-time task, the runtime of the energy-efficient scheduler exceeds the deadline, which leads to the 0 in its SoC_{time} and *SoC* (i.e.,'x' in Fig. 15(a)). In Fig. 14(a), P-CNN consumes the least energy among these schedulers (nearly equal to the Ideal scheduler). We explain this phenomenon step-by-step. For the interactive task, QPE+ consumes less energy than QPE because QPE+ can power-off the idle SMs to save energy. However, for the real-time and background tasks, the energies of QPE and QPE+ are the same since the GPU resource has been utilized in QPE and no idle SM can be closed by QPE+. For the accuracy non-sensitive tasks, such as age detection and image tagging, P-CNN can save more energy than QPE+ by choosing the fastest kernels (1.5X speedup) with acceptable accuracy (5% loss). However, for the applications require high accuracy (i.e., video surveillance), P-CNN consumes similar energy as QPE+.

With the ideal SoC_{time} and least energy consumption, P-CNN obtains the high *SoC* in Fig. 15(a). However, it still does not achieve as high a *SoC* as Ideal scheduler, especially for the interactive task (i.e., age detection). This is because the entertainment app, such as age detection, can work well with a low accuracy while P-CNN takes a conservative accuracy to perform this task. In the future, we can improve prediction of accuracy requirements for the interactive task by learning user experience. We then compare these five schedulers on the Nvidia TX1. Evaluation results show that P-CNN also achieves the ideal SoC_{time} (Fig. 13(b)), the least energy consumption (Fig. 14(b)) and the high *SoC* (Fig. 15(b)) on TX1. Note that all of the schedulers fail to meet user satisfaction (i.e., 'x' in Fig. 15(b)) for the real-time task except our P-CNN and Ideal scheduler. Even though it uses

the non-batch method, as shown in Fig. 13(b), the runtime of the real-time task still exceeds the deadline on TX1, which is not acceptable for end-users. Our P-CNN can further accelerate the procedure of inference on TX1 to meet user satisfaction by levering the approximate method.

D. Accuracy Tuning

As discussed in Section IV.C, we use network uncertainty (i.e., $CNN_{entropy}$) to guide run-time accuracy tuning. Here we use a labeled test dataset to evaluate the accuracy variation in the procedure of entropy-based accuracy tuning. Fig. 16 shows the speedup (orange bar), the entropy (blue line) and accuracy (green line) during this tuning procedure. We observe the speedup increases monotonically until the network uncertainty is larger than the pre-defined threshold. At the same time, the increase of network uncertainty (i.e., ΔE) corresponds to the decrease of network accuracy (i.e., ΔA), which demonstrates that CNN_{entroy} is an effective metric to evaluate the accuracy of CNN. After accuracy tuning, we achieve 1.8X speedup with a small decrease (10%) in accuracy. We also include the accuracy based approximation in Fig. 16, where the CNN accuracy is measured using the labeled data, then the approximation is guided by the accuracy. Fig. 16 illustrates how our entropy-based approximation obtains the same effect (i.e. similar speedup and accuracy) as the accuracy-based method. Compared with the accuracy-based method, our entropy-based method is unsupervised (without labeled data to measure accuracy), which is applicable to the run-time accuracy.

VI. Related Work

Deep Learning: Sirius [34] is an open end-to-end intelligent personal assistant based on DNN services. The DjiNN [35] further brings the community the characterization of GPU acceleration server system running DNN services and provides insights into designing future warehouse-scale computer architectures for DNN services. D^{3}NN [23] also develops a high-throughput deep learning system for high-end servers. Not only focusing on data center platform, our P-CNN can efficiently deploy CNN-based applications to different platforms with high user satisfaction.

GPU: Recently, architectural extensions have been proposed for GPU multi-tasking. Adriaens et al. demonstrate that not all GPU applications fully use GPU resources [22]. They propose a few heuristics for spatial multitasking and demonstrate performance improvements compared to sequential kernel execution using GPGPU-sim. Liang et al. further improve the GPU multi-tasking. They propose GPU Spatial-Temporal Multitasking [20], which allows different kernels to share the GPU resources both spatially and temporally. However, the diverse GPU resource utilization of CNNs among different layers challenges these multi-tasking mechanisms, which results in inappropriate SM allocation among various layers. Moreover, their methods do not take user satisfaction into account when allocating SMs. Although our P-CNN is also based on spatial partitioning, we first use our analytical model to evaluate the *Util* of each layer and then allocate optimal SMs according to the *Util* in each layer, which can enable high user satisfaction with fewer SMs.

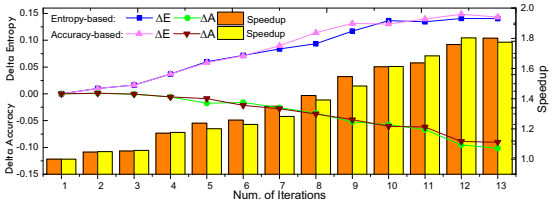

Figure 16. Entropy-based and Accuracy-based approximations

Approximate Computing: Recently, researchers in the deep learning community have developed several methods to accelerate CNNs, including low-rank approximations [36] and network pruning [37]. Among these, network pruning mainly focuses on reducing the storage of CNNs so that it is easily deployed into mobile platforms. Pruning based approaches mainly compress the parameters of fully-connected layers (where most CNN parameters reside), while our method mainly focuses to accelerate the convolutional layers (where most processing time is spent). What is more, it requires the modification of the CNN architecture, which results in a time-consuming retraining. Therefore, network pruning cannot be used in run-time accuracy adjustment. PerforatedCNNs [38] also leverages the loop perforation technique to do the approximation. While its approximate procedure is guided as a supervised method (i.e., loss function), it is still not applicable to run-time accuracy tuning.

VII. Conclusion

This work proposes Pervasive CNN, a user satisfaction aware CNN inference framework. P-CNN consists of cross-platform offline compilation and run-time management. First, P-CNN infers the requirements of end-users based on the specification of CNN-based applications. According to this requirement, offline compilation generates the optimal kernel for the deployed platform. Then, accuracy tuning finds the optimal accuracy for this deployed scenario. During execution, the run-time kernel management partitions the optimal SM resource to each layer, then schedules the GPU thread blocks. If its accuracy is not tolerated by end-users, calibration chooses a slower but more precise kernel to improve the accuracy. In particular, P-CNN features an accurate time model to guarantee the requirement of response time. Our evaluation results show P-CNN can efficiently deploy CNN-based applications to all kinds of platforms with the best user satisfaction when compared with other runtime scheduler schemes.

Acknowledgment

We would like to thank James Poe and Amer Qouneh for making this paper better. This work is supported in part by NSF grants 1527535, 1423090, 1320100,1117261, 0937869, 0916384, 0845721(CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multicore Computing Awards, and by three IBM Faculty Awards.

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems (NIPS), 2012.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In IEEE International Conference on Computer Vision (ICCV), 2015.

[3] Jia Deng, Wei Dong, R Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In IEEE Conference on Computer Vision and Pattern Recognition, 2009.

[4] A Simonyan, K. and Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, CoRR, vol. abs/1409.1, 2014.

[5] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014.

[6] Prisma: http://prisma-ai.com/.

[7] Chao Li, Yang Hu, Longjun Liu, Juncheng Gu, Mingcong Song, Xiaoyao Liang, Jingling Yuan, and Tao Li. Towards Sustainable In-situ Server Systems in the Big Data Era. In Proceedings of the 42Nd Annual International Symposium on Computer Architecture, 2015.

[8] Moments: http://www.momentsapp.com/.

[9] cuBLAS: https://developer.nvidia.com/cuBLAS.

[10] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient QoS (eQoS) in mobile Web applications. In International Symposium on High Performance Computer Architecture (HPCA), 2015.

[11] Yunlong Xu, Rui Wang, Tao Li, Mingcong Song, Lan Gao, Zhongzhi Luan, and Depei Qian. Scheduling Tasks with Mixed Timing Constraints in GPU-Powered Real-Time Systems. In Proceedings of the 2016 International Conference on Supercomputing (ICS), 2016.

[12] AJ Joshi, F Porikli, and N Papanikolopoulos. Multi-class active learning for image classification. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009.

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In IEEE Conference on Computer Vision and Pattern Recognition, 2015.

[14] NVIDIA Visual Profiler: https://developer.nvidia.com/nvidia-visual-profiler.

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, Trevor Darrell, and UC Berkeley Eecs. Caffe : Convolutional Architecture for Fast Feature Embedding. In ACM Multimedia, 2014.

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, arXiv Prepr. arXiv1410.0759, Oct. 2014.

[17] Nervana: https://github.com/NervanaSystems/nervanagpu.

[18] Vasily Volkov and James W Demmel. Benchmarking GPUs to tune dense linear algebra. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2008.

[19] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In International Symposium on High-Performance Computer Architecture, 2014.

[20] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. Efficient GPU spatial-temporal multitasking, IEEE Trans. Parallel Distrib. Syst., vol. 26, no. 3, pp. 748–760, 2015.

[21] MULTI-PROCESS SERVICE: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[22] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for GPGPU spatial multitasking. In International Symposium on High-Performance Computer Architecture, 2012.

[23] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. Bridging the Semantic Gaps of GPU Acceleration for Scale-out CNN-based Big Data Processing: Think Big, See Small. In International Conference on Parallel Architectures and Compilation Techniques (PACT), 2016.

[24] B Gaudette, CJ Wu, and S Vrudhula. Improving smartphone user experience by balancing performance and energy with probabilistic QoS guarantee. In International Symposium on High Performance Computer Architecture (HPCA), 2016.

[25] Kaige Yan, Xingyao Zhang, and Xin Fu. Characterizing, Modeling, and Improving the QoE of Mobile Devices with Low Battery Level. In International Symposium on Microarchitecture (MICRO), 2015.

[26] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In International Symposium on Microarchitecture (MICRO), 2015.

[27] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. SAGE: Self-Tuning Approximation for Graphics Engines. In International Symposium on Microarchitecture, 2013.

[28] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and TM Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In International Symposium on Performance Analysis of Systems and Software (ISPASS), 2009.

[29] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In International Symposium on Computer Architecture (ISCA), 2013.

[30] Zhijia Zhao, Mingzhou Zhou, and Xipeng Shen. Satscore: Uncovering and avoiding a principled pitfall in responsiveness measurements of app launches. In International Joint Conference on Pervasive and Ubiquitous Computing (Ubicomp), 2014.

[31] Yasuhiro Endo, Zheng Wang, J Bradley Chen, and Margo Seltzer. Using Latency to Evaluate Interactive System Performance. In USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1996.

[32] How loading time affects your bottom line: https://blog.kissmetrics.com/loading-time/.

[33] John A Stratton, Christopher Rodrigues, I-jui Sung, Nady Obeid, Li-wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing, Cent. Reliab. High-Performance Comput., vol. 127, 2012.

[34] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

[35] Johann Hauswald, Yiping Kang, Michael A Laurenzano, Quan Chen, Cheng Li, Ronald Dreslinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Djinn and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In International Symposium on Computer Architecture (ISCA), 2015.

[36] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating Very Deep Convolutional Networks for Classification and Detection, IEEE Trans. Pattern Anal. Mach. Intell., pp. 1943–1955, 2015.

[37] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In International Symposium on Computer Architecture (ISCA), 2016.

[38] Michael Figurnov, Dmitry Vetrov, and Pushmeet Kohli. PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions, arXiv Prepr. arXiv1504.08362, Apr. 2015.