Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge

Yiping Kang Johann Hauswald Cao Gao Austin Rovinski
Trevor Mudge Jason Mars Lingjia Tang

Clarity Lab
University of Michigan - Ann Arbor, MI, USA
{ypkang, jahausw, caogao, rovinski, tnm, profmars, lingjia}@umich.edu

Abstract

The computation for today's intelligent personal assistants such as Apple Siri, Google Now, and Microsoft Cortana, is performed in the cloud. This cloud-only approach requires significant amounts of data to be sent to the cloud over the wireless network and puts significant computational pressure on the datacenter. However, as the computational resources in mobile devices become more powerful and energy efficient, questions arise as to whether this cloud-only processing is desirable moving forward, and what are the implications of pushing some or all of this compute to the mobile devices on the edge.

In this paper, we examine the status quo approach of cloud-only processing and investigate computation partitioning strategies that effectively leverage both the cycles in the cloud and on the mobile device to achieve low latency, low energy consumption, and high datacenter throughput for this class of intelligent applications. Our study uses 8 intelligent applications spanning computer vision, speech, and natural language domains, all employing state-of-the-art Deep Neural Networks (DNNs) as the core machine learning technique. We find that given the characteristics of DNN algorithms, a fine-grained, layer-level computation partitioning strategy based on the data and computation variations of each layer within a DNN has significant latency and energy advantages over the status quo approach.

Using this insight, we design `Neurosurgeon`, a lightweight scheduler to automatically partition DNN computation between mobile devices and datacenters at the granularity of neural network layers. `Neurosurgeon` does not require per-application profiling. It adapts to various DNN architectures, hardware platforms, wireless networks, and server load levels, intelligently partitioning computation for best latency or best mobile energy. We evaluate `Neurosurgeon` on a state-of-the-art mobile development platform and show that it improves end-to-end latency by 3.1× on average and up to 40.7×, reduces mobile energy consumption by 59.5% on average and up to 94.7%, and improves datacenter throughput by 1.5× on average and up to 6.7×.

Keywords mobile computing; cloud computing; deep neural networks; intelligent applications

1. Introduction

The way we interact with today's mobile devices is rapidly changing as these devices are increasingly personal and knowledgeable. Intelligent Personal Assistants (IPAs), such as Apple Siri, Google Now, and Microsoft Cortana, are integrated by default on mobile devices and are expected to grow in popularity as wearables and smart home devices continue to gain traction [1, 2]. The primary interface with these intelligent mobile applications is using speech or images to navigate the device and ask questions. Demand for this mode of interaction is expected to replace the traditional text based inputs [3–5].

Processing speech and image inputs for IPA applications requires accurate and highly sophisticated machine learning techniques, the most common of which are Deep Neural Networks (DNNs). DNNs have become increasingly popular as the core machine learning technique in these applications due to their ability to achieve high accuracy for tasks such as speech recognition, image classification and natural language understanding. Many companies, including Google, Microsoft, Facebook, and Baidu, are using DNNs as the machine learning component for numerous applications in their production systems [6–8].

Prior work has shown that speech or image queries for DNN-based intelligent applications require orders of magnitude more processing than text based inputs [9]. The common wisdom has been that traditional mobile devices cannot support this large amount of computation with reasonable latency and energy consumption. Thus, the status quo approach used by web service providers for intelligent applications has been to host all the computation on high-end cloud servers [10–13]. Queries generated from a user's mo-

ASPLOS '17, April 08-12, 2017, Xi'an, China

DOI: http://dx.doi.org/10.1145/3037697.3037698

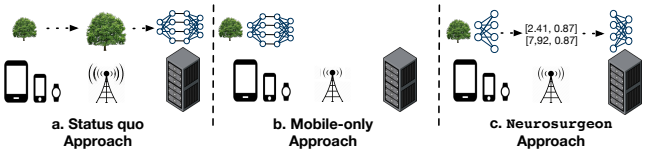


Figure 1: Status quo, mobile-only and the Neurosurgeon approach. Status quo approach performs all computation remotely in the cloud, the mobile-only approach performs all computation locally on the mobile device, and the Neurosurgeon approach partitions computation between the cloud and mobile device.

bile device are sent to the cloud for processing, as shown in Figure 1a. However, with this approach, large amounts of data (e.g., images, video and audio) are uploaded to the server via the wireless network, resulting in high latency and energy costs.

While data transfer becomes the latency and energy bottleneck, performance and energy efficiency of modern mobile hardware have continued to improve through powerful mobile SoC integration [14, 15]. Motivated by this observation, this work re-examines the computation breakdown for intelligent applications between mobile and cloud. In particular, we investigate how computation can be pushed out of the cloud and onto the mobile devices on the edge to execute all or parts of these conventionally cloud-only applications. Key questions we address in this work include:

1. How feasible it is to execute large-scale intelligent workloads on today's mobile platforms?

2. At what point is the cost of transferring speech and image data over the wireless network too high to justify cloud processing?

3. What role should the mobile edge play in providing processing support for intelligent applications requiring heavy computation?

Based on our investigation using 8 DNN-based intelligent applications spanning the domains of vision, speech, and natural language, we discover that, for some applications, due to the high data transfer overhead, locally executing on the mobile device (Figure 1b) can be up to 11× faster than the cloud-only approach (Figure 1a). Furthermore, we find that instead of limiting the computation to be either executed entirely in the cloud or entirely on the mobile, a fine-grained layer-level partitioning strategy based on a DNN's topology and constituent layers can achieve far superior end-to-end latency performance and mobile energy efficiency. By pushing compute out of the cloud and onto the mobile devices, we also improve datacenter throughput, allowing a given datacenter to support many more user queries, and creating a win-win situation for both the mobile and cloud systems.

Given the observation that ideal fine-grained DNN partition points depend on the layer compositions of the DNN, the particular mobile platform used, the wireless network configuration and the server load, we design a lightweight dynamic scheduler, Neurosurgeon. Neurosurgeon is a runtime system spanning cloud and mobile platforms that automatically identifies the ideal partition points in DNNs and orchestrates the distribution of computation between the mobile device and the datacenter. As Figure 1c shows, Neurosurgeon partitions the DNN computation and takes advantage of the processing power of both the mobile and the cloud while reducing data transfer overhead. The detailed contributions of this paper are as follows:

- **In-depth examination of the status quo –** We show the latency and energy consumption of executing state-of-the-art DNNs in the cloud and on the mobile device. We observe that uploading via the wireless network is the bottleneck of the status quo approach, and mobile execution often provides better latency and energy consumption than the status quo approach. (Section 3)
- **DNN compute and data size characteristics study –** We provide an in-depth layer-level characterization of the compute and data size of 8 DNNs spanning across computer vision, speech and natural language processing. Our investigation reveals that DNN layers have significantly different compute and data size characteristics depending on their type and configurations. (Section 4)
- **DNN computation partitioning across the cloud and mobile edge –** Based on the compute and data characterization of DNN layers, we show that partitioning DNN at layer granularity offers significant performance benefits. We then design a systematic approach to identify the optimal points to partition computation for reduced latency and mobile energy consumption across a suite of applications. (Section 4)
- **Neurosurgeon runtime system and layer performance prediction models –** We develop a set of models to predict the latency and power consumption of a DNN layer based on its type and configuration, and create Neurosurgeon, a system to intelligently partition DNN computation between the mobile and cloud. We demonstrate that Neurosurgeon significantly improves end-to-end latency, reduces mobile energy consumption, and improves datacenter throughput. (Sections 5 and 6)

Our evaluation on a suite of 8 DNN applications shows that using Neurosurgeon on average improves end-to-end latency by 3.1×, reduces mobile energy consumption by 59.5%, and improves datacenter throughput by 1.5×.

2. Background

In this section, we provide an overview of Deep Neural Network (DNN) and describe how computer vision, speech, and natural language processing applications leverage DNNs as their core machine learning algorithm.

DNNs are organized in a directed graph where each node is a processing element (a neuron) that applies a function to its input and generates an output. Figure 2 depicts a 5 layer

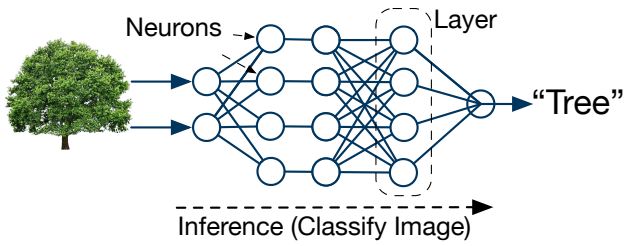


Figure 2: A 5-layer Deep Neural Network (DNN) classifies input image into one of the pre-defined classes.

DNN for image classification where computation flows from left to right. The edges of the graph are the connections between each neuron defining the flow of data. Multiple neurons applying the same function to different parts of the input define a layer. For a forward pass through a DNN, the output of a layer is the input to the next layer. The depth of a DNN is determined by the number of layers. Computer Vision (CV) applications use DNNs to extract features from an input image and classify the image into one of the predefined classes. Automatic Speech Recognition (ASR) applications use DNNs to generate predictions for speech feature vectors, which will then be post-processed to produce the most-likely text transcript. Natural Language Processing (NLP) applications use DNNs to analyze and extract semantic and syntactic information from word embedding vectors generated from input text.

3. Cloud-only Processing: The Status Quo

Currently, the status quo approach used by cloud providers for intelligent applications is to perform all DNN processing in the cloud [10–13]. A large overhead of this approach is in sending data over the wireless network. In this section, we investigate the feasibility of executing large DNNs entirely on a state-of-the-art mobile device, and compare with the status quo.

3.1 Experimental setup

We use a real hardware platform, representative of today's state-of-the-art mobile devices, the Jetson TK1 mobile platform developed by NVIDIA [16] and used in the Nexus 9 tablet [17]. The Jetson TK1 is equipped with one of NVIDIA's latest mobile SoC, Tegra K1: a quad-core ARM A15 and a Kepler mobile GPU with a single streaming multiprocessor (Table 1).

Table 1: Mobile Platform Specifications

Hardware	Specifications
System	Tegra K1 SoC
CPU	4-Plus-1 quad-core ARM Cortex A15 CPU
Memory	2 GB DDR3L 933MHz
GPU	NVIDIA Kepler with 192 CUDA Cores

Table 2: Server Platform Specifications

Hardware	Specifications
System	4U Intel Dual CPU Chassis, 8 × PCIe 3.0 × 16 slots
CPU	2 × Intel Xeon E5-2620 V2, 6C, 2.10 GHz
HDD	1TB 2.5" HDD
Memory	16 × 16GB DDR3 1866MHz ECC/Server Memory
GPU	NVIDIA Tesla K40 M-Class 12 GB PCIe

Our server platform is equipped with an NVIDIA Tesla K40 GPU, one of NVIDIA's latest offering in server class GPUs (Table 2).

We use Caffe [18], an actively developed open-source deep learning library, for the mobile and server platform. For the mobile CPU, we use OpenBLAS [19], a NEON-vectorized matrix multiplication library and use the 4 cores available. For both GPUs, we use cuDNN [20], an optimized NVIDIA library that accelerates key layers in Caffe, and use Caffe's CUDA implementations for rest of the layers.

3.2 Examining the Mobile Edge

We investigate the capability of the mobile platform to execute a traditionally cloud-only DNN workload. We use AlexNet [21] as our application, a state-of-the-art Convolutional Neural Network for image classification. Prior work has noted that AlexNet is representative of today's DNNs deployed in server environments [22].

In Figure 3, we break down the latency of an AlexNet query, a single inference on a 152KB image. For wireless communication, we measure the bandwidth of 3G, LTE, and Wi-Fi on several mobile devices using TestMyNet [23].

Communication Latency – Figure 3a shows the latency to upload the input image via 3G, LTE, and Wi-Fi. The slowest is 3G connection taking over 870ms. LTE and Wi-Fi connection require 180ms and 95ms to upload, respectively, showing that the network type is critical for achieving low latency for the status quo approach.

Computation Latency – Figure 3b shows the computation latency on mobile CPU, GPU and cloud GPU. The slowest platform is the mobile CPU taking 382ms to process while the mobile GPU and cloud GPU take 81ms and 6ms, respectively. Note that the mobile CPU's time to process the image is still 2.3× faster than uploading input via 3G.

End-to-end Latency – Figure 3c shows the total latency required by the status quo and the mobile-only approach. Annotated on top of each bar is the fraction of the end-to-end latency spent on computation. The status quo approach spends less than 6% of the time computing on the server and over 94% of the time transferring data. The mobile GPU achieves a lower end-to-end latency than the status quo approach using LTE and 3G, while the status quo approach using LTE and Wi-Fi performs better than mobile CPU execution.

Energy Consumption – We measure the energy consumption of the mobile device using a Watts Up? meter [24] and

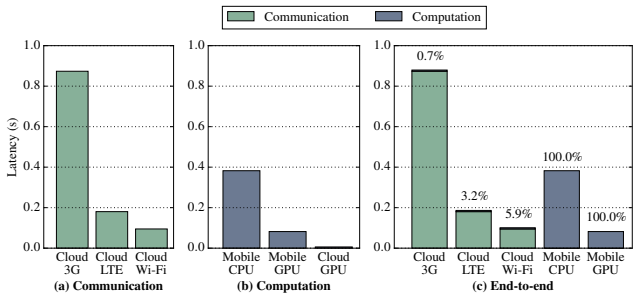


Figure 3: Latency breakdown for AlexNet (image classification). The cloud-only approach is often slower than mobile execution due to the high data transfer overhead.

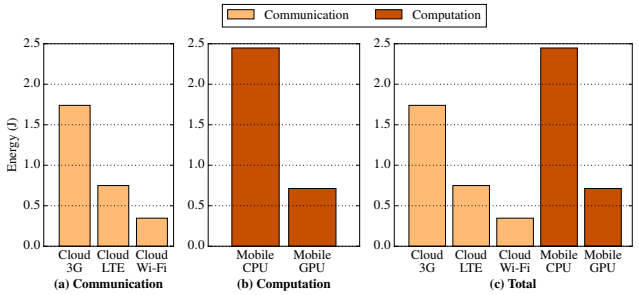


Figure 4: Mobile energy breakdown for AlexNet (image classification). Mobile device consumes more energy transferring data via LTE and 3G than computing locally on the GPU.

techniques described by Huang et al. [25]. Similar to the trends shown in Figure 3a, Figure 4a shows that the communication energy is heavily dependent on the type of wireless network used. In Figure 4b, the mobile device's energy consumption is higher on the CPU than the GPU (while the GPU needs more power, the device is used for a shorter burst thus it consumes less total energy). Figure 4c shows the total mobile energy consumption for the cloud-only approach and mobile execution where the energy in the cloud-only approach is dominated by communication. The mobile GPU consumes less energy than transferring input via LTE or 3G for cloud processing, while cloud processing via Wi-Fi consumes less energy than mobile execution.

Key Observations – 1) The data transfer latency is often higher than mobile computation latency, especially on 3G and LTE. 2) Cloud processing has a significant computational advantage over mobile processing, but it does not always translate to end-to-end latency/energy advantage due to the dominating data transfer overhead. 3) Local mobile execution often leads to lower latency and energy consumption than the cloud-only approach, while the cloud-only approach achieves better performance if using fast Wi-Fi connection.

4. Fine-grained Computation Partitioning

Based on the findings in Section 3, the question arises as to whether it is advantageous to partition DNN computation between the mobile device and cloud. Based on the observation that DNN layers provide an abstraction suitable for partitioning computation, we begin with an analysis of the data and computation characteristics of state-of-the-art DNN architectures at the layer granularity.

4.1 Layer Taxonomy

Before the layer-level analysis, it is important to understand the various types of layers present in today's DNNs.

Fully-connected Layer (`fc`) – All the neurons in a fully-connected layer are exhaustively connected to all the neurons in the previous layer. The layer computes the weighted sum of the inputs using a set of learned weights.

Convolution & Local Layer (`conv`, `local`) – Convolution and local layers convolve the image with a set of learned filters to produce a set of feature maps. These layers mainly differ in the dimensions of their input feature maps, the number and size of their filters, and the stride with which the filters are being applied.

Pooling Layer (`pool`) – Pooling layers apply a pre-defined function (e.g., max or average) over regions of input feature maps to group features together. These layers mainly differ in the dimension of their input, size of the pooling region, and the stride with which the pooling is applied.

Activation Layer – Activation layers apply a non-linear function to each of its input data individually, producing the same amount of data as output. Activation layers present in the neural networks studied in this work include sigmoid layer (`sig`), rectified-linear layer (`relu`), and hard Tanh layer (`htanh`).

Other layers studied in this work include: **normalization layer (`norm`)** normalizes features across spatially grouped feature maps; **softmax layer (`softmax`)** produces a probability distribution over the number of possible classes for classification; **argmax layer (`argmax`)** chooses the class with the highest probability; and **dropout layer (`dropout`)** randomly ignores neurons during training to avoid model over-fitting and are passed through during prediction.

4.2 Characterizing Layers in AlexNet

We first investigate the data and computation characteristics of each layer in AlexNet. These characteristics provide insights to identify a better computation partitioning between mobile and cloud at the layer level. In the remainder of this and subsequent sections, we use the GPU in both mobile and server platforms.

Per-layer Latency – The left bars (light-colored) in Figure 5 show the latency of each layer on the mobile platform, arranged from left to right in their sequential execution order. The convolution (`conv`) and fully-connected layers (`fc`) are the most time-consuming layers, representing over 90% of the total execution time. Convolution layers in the middle (`conv3` and `conv4`) takes longer to execute than the early convolution layers (`conv1` and `conv2`). Larger number of filters are applied by the convolution layers later in the DNN to progressively extract more robust and representative fea-

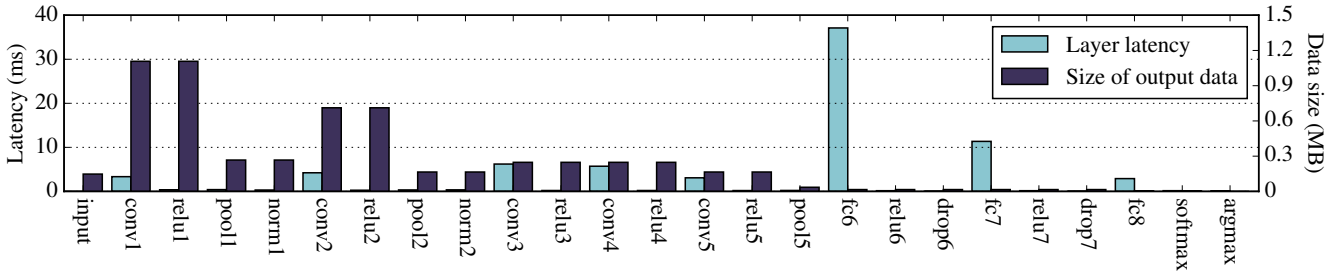
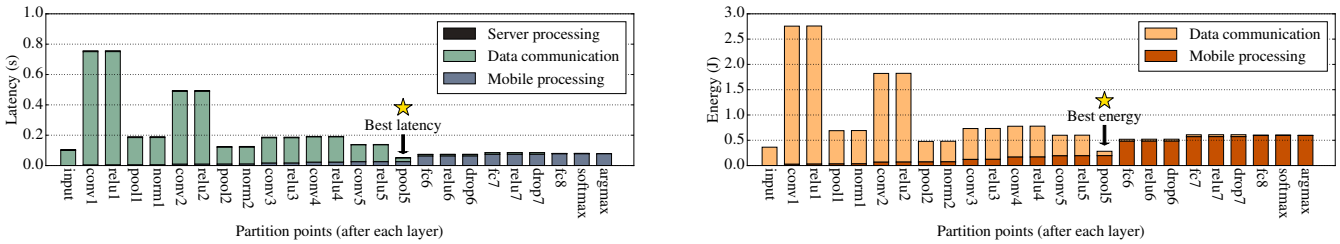


Figure 5: The per layer execution time (the light-colored left bar) and size of data (the dark-colored right bar) after each layer's execution (input for next layer) in AlexNet. Data size sharply increases then decreases while computation generally increases through the network's execution.



(a) AlexNet latency **(b) AlexNet energy consumption**

Figure 6: End-to-end latency and mobile energy consumption when choosing different partition points. After the execution of every layer is considered a partition point. Each bar represents the total latency (a) or mobile energy (b) if the DNN is partitioned after the layer marked on the X-axis. The left-most bar represents cloud-only processing and the right-most bar represents mobile-only processing. The partition points for best latency and mobile energy are annotated.

tures, increasing the amount of computation. On the other hand, fully-connected layers are up to one magnitude slower than the convolution layers in the network. The most time-consuming layer is the layer `fc6`, a fully-connected layer deep in the DNN, taking 45% of the total execution time.

Data Size Variations – The right bars (dark-colored) in Figure 5 shows the size of each layer's output data, which is also the input to the next layer. The first three convolution layers (`conv1`, `conv2` and `conv3`) generate large amounts of output data (shown as the largest dark bars) as they apply hundreds of filters over their input feature maps to extract interesting features. The data size stays constant through the activation layers (`relu1 - relu5`). The pooling layers sharply reduce the data size by up to 4.7× as they summarize regions of neighboring features by taking the maximum. The fully-connected layers deeper in the network (`fc6 - fc8`) gradually reduce the data size until the softmax layer (`softmax`) and argmax layer (`argmax`) at the end reduce the data to be one classification label.

Key Observations – 1) Depending on its type and location in the network, each layer has a different computation and data profile. 2) The latency of convolution and pooling layers on the mobile GPU are relatively small, while fully-connected layers incur high latency. 3) Convolution and pooling layers are mostly at the front-end of the network, while fully-connected layers are at the back-end. 4) With convolution layers increasing data and then pooling layers reducing data, the front-end layers altogether reduce the size of data gradually. Data size in the last few layers are smaller than the original input. 5) The findings that data size is generally decreasing at the front-end, and per-layer mobile latency is generally higher at the back-end, indicates the unique opportunity for computation partitioning in the middle of the DNN between the mobile and cloud.

4.3 Layer-granularity Computation Partitioning

The analysis in Section 4.2 indicates that there exist interesting points within a neural network to partition computation. In this section, we explore partitioning AlexNet at each layer between the mobile and cloud. In this section, we use Wi-Fi as the wireless network configuration.

Each bar in Figure 6a represents the end-to-end latency of AlexNet, partitioned after each layer. Similarly, each bar in Figure 6b represents the mobile energy consumption of Alexnet, partitioned after each layer. Partitioning computation after a specific layer means executing the DNN on the mobile up to that layer, transferring the output of that layer to the cloud via wireless network, and executing the remaining layers in the cloud. The leftmost bar represents sending the original input for cloud-only processing. As partition point moves from left to right, more layers are executed on the mobile device thus there is an increasingly larger mobile processing component. The rightmost bar is the latency of executing the entire DNN locally on the mobile device.

Partition for Latency – If partitioning at the front-end, the data transfer dominates the end-to-end latency, which is consistent with our observation in Section 4.2 that the data size is the largest at the early stage of the DNN. Partitioning at the back-end provides better performance since the application can minimize the data transfer overhead, while taking ad-

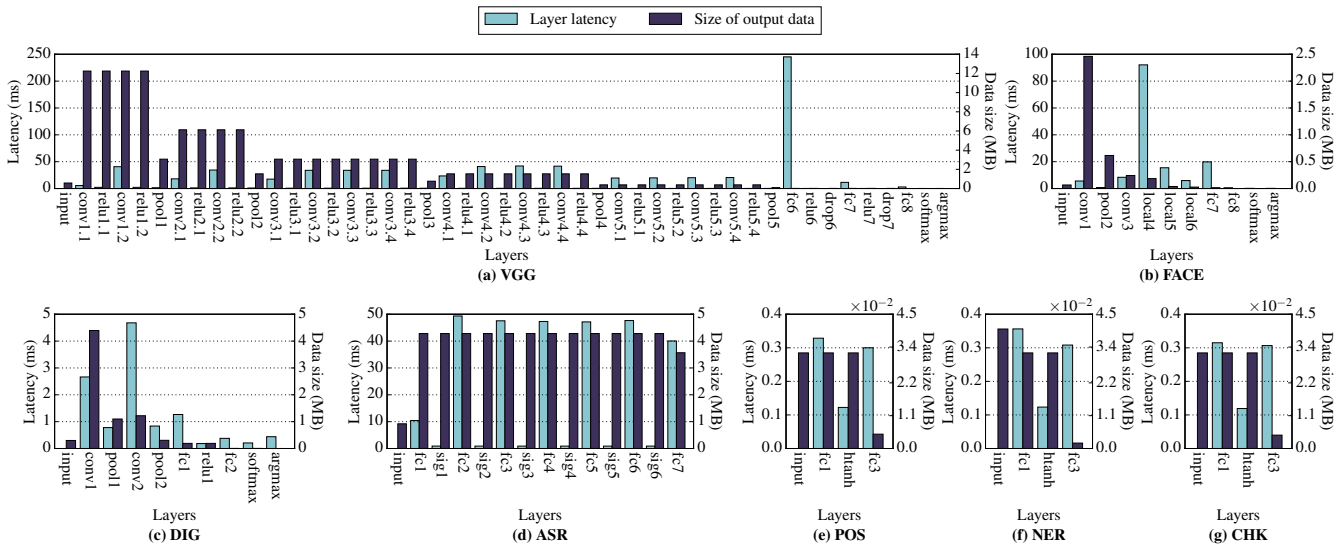


Figure 7: The per layer latency on the mobile GPU (left light-color bar) and size of data (right dark-color bar) after each layer's execution.

vantage of the powerful server to execute the more compute-heavy layers at the back-end. In the case of AlexNet using the mobile GPU and Wi-Fi, partitioning between the last pooling layer (`pool5`) and the first fully-connected layer (`fc6`) achieves the lowest latency, as marked in Figure 6a, improving 2.0× over cloud-only processing.

Partition for Energy – Similar to latency, due to the high energy cost of wireless data transfer, transferring the input for cloud-only processing is not the most energy-efficiency approach. As marked in Figure 6b, partitioning in the middle of the DNN achieves the best mobile energy consumption, 18% more energy efficient than the cloud-only approach.

***Key Observations* –** Partitioning at the layer granularity can provide significant latency and energy efficiency improvements. For AlexNet using the GPU and Wi-Fi, the best partition points are between the intermediate layers of the DNN.

4.4 Generalizing to More DNNs

We expand our investigation to 7 more intelligent applications to study their data and computation characteristics and their impact on computation partitioning opportunity. We use the DNNs provided in the Tonic suite [9], as well as VGG, a state-of-the-art image classification DNN, and LTE as the wireless network configuration. Details about the benchmarks are listed in Table 3. We count the number of layers of each DNN starting from the first non-input layer to the last layer, including `argmax` if present.

CV Applications – The three remaining computer vision DNNs (VGG, FACE and DIG) have similar characteristics as AlexNet (Figure 5), as shown in Figures 7a –7c. The front-end layers are convolution layers increasing data, and pooling layers reducing data. The data size in the back-end layers are similar or smaller than the original input data. The latency for the back-end layers are higher than most of the

Table 3: Benchmark Specifications

App	Abbr.	Network	Input	Layers
Image classification	IMC	AlexNet [21]	Image	24
	VGG	VGG [26]	Image	46
Facial recognition	FACE	DeepFace [27]	Image	10
Digit recognition	DIG	MNIST [28]	Image	9
Speech recognition	ASR	Kaldi [29]	Speech features	13
Part-of-speech tagging	POS	SENNA [30]	Word vectors	3
Named entity recognition	NER	SENNA [30]	Word vectors	3
Word chunking	CHK	SENNA [30]	Word vectors	3

front-end layers (e.g., `fc6` is the most time-consuming layer in VGG), except for DIG where convolution layers are most time-consuming. Similar to AlexNet, these characteristics indicate partitioning opportunities in the middle of the DNN. Figure 8a shows that the partition point for best latency for VGG is in the intermediate layers. In addition, Figures 8a - 8c show that different CV applications have different partition points for best latency, and Figures 9a - 9c show the different partition points for best energy for these DNNs.

ASR and NLP Applications – The remaining four DNNs in the suite (ASR, POS, NER and CHK) only consist of fully-connected layers and activation layers. The layer breakdowns are shown in Figures 7d - 7g, where, throughout the execution, layers of the same type incur similar latency and the data size stay relatively constant except for the very first and last layer of each DNN. These DNNs do not have data-increasing layers (i.e., convolution layers) or data-reducing layers (i.e., pooling layers). As a result, there only exist opportunities for partitioning the computation at the extremities of these networks. Figures 8d - 8g and Figures 9d - 9g show the different partition points for best latency and energy for these DNNs, respectively. There are data communication components in the right-most bars (mobile-only processing) for these applications because the output of the DNN is sent to the cloud for post-processing steps required by these applications.

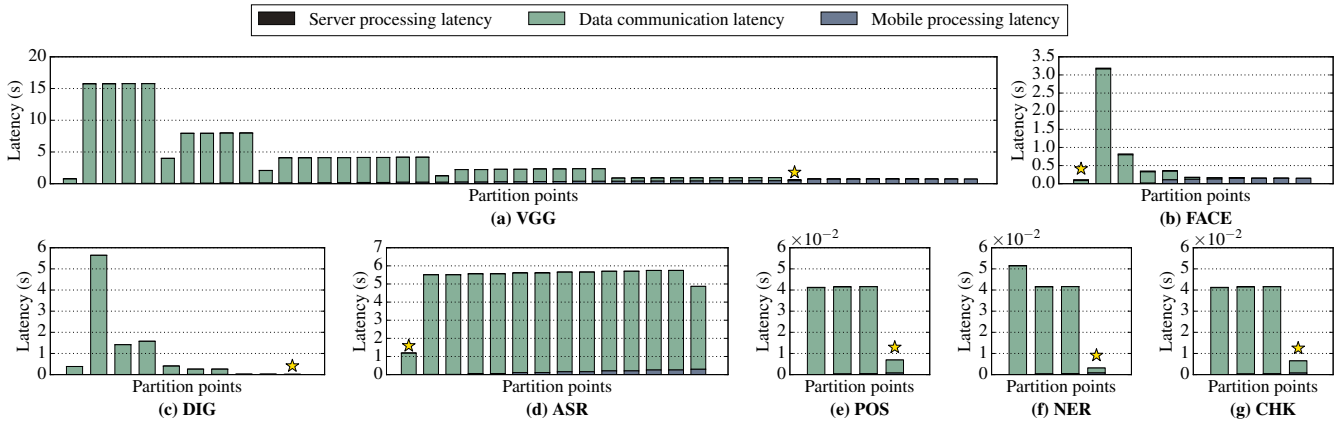


Figure 8: End-to-end latency when choosing different partition points. Each bar represents the end-to-end latency if the DNN is partitioned after each layer, where the left-most bar represents cloud-only processing (i.e., partitioning at the beginning) while the right-most bar represents mobile-only execution (i.e., partitioning at the end). The wireless network configuration is LTE. The partition points for best latency are each marked by ★.

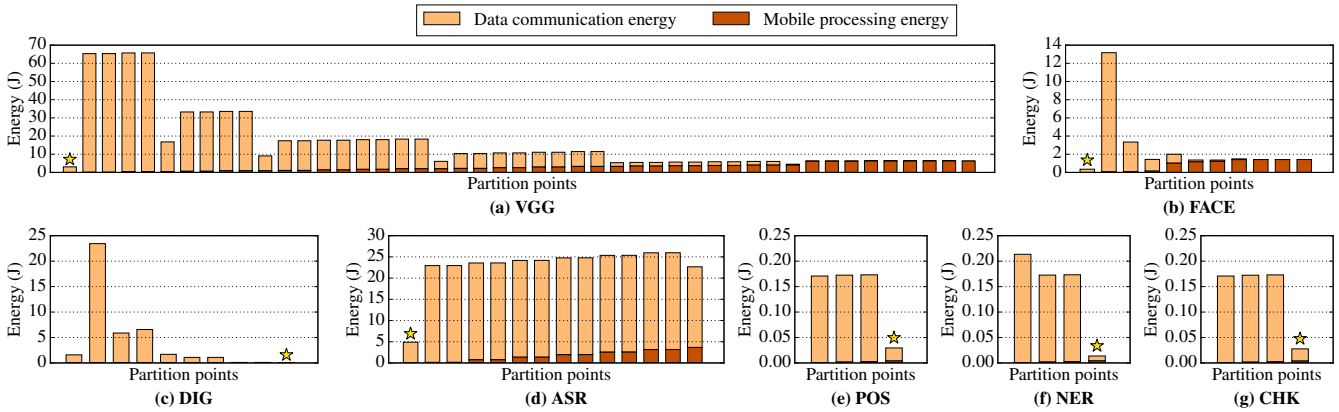


Figure 9: Mobile energy consumption when choosing different partition points. Each bar represents the mobile energy consumption if the DNN is partitioned after each layer, where the left-most bar represents cloud-only processing (i.e., partitioning at the beginning) while the right-most bar represents mobile-only execution (i.e., partitioning at the end). The wireless network configuration is LTE. The partition points for best energy are each marked by ★.

Key Observations – 1) In DNNs with convolution and pooling layers (e.g. Computer Vision applications), the data size increases after convolution layers and decreases after pooling layers, while the per-layer computation generally increases through the execution. 2) DNNs with only fully-connected layers of similar size and activation layers see small variations in per-layer latency and data size (e.g., ASR and NLP DNNs). 3) The best way to partition a DNN depends on its topology and constituent layers. Computer vision DNNs sometimes have better partition points in the middle of the DNN, while it is more beneficial to partition at the beginning or the end for ASR and NLP DNNs. The strong variations in the best partition point suggest there is a need for a system to partition DNN computation between the mobile and cloud based on the neural network architecture.

5. Neurosurgeon

The best partition point for a DNN architecture depends on the DNN's topology, which manifests itself in the computation and data size variations of each layer. In addition, dynamic factors such as state of the wireless network and datacenter load affect the best partition point even for the same DNN architecture. For example, mobile devices' wireless connections often experience high variances [31], directly affecting the data transfer latency. Datacenters typically experience diurnal load patterns [32], leading to high variance in its DNN query service time. Due to these dynamic factors, there is a need for an automatic system to intelligently select the best point to partition the DNN to optimize for end-to-end latency or mobile device energy consumption. To address this need, we present the design of `Neurosurgeon`, an intelligent DNN partitioning engine. `Neurosurgeon` consists of a deployment phase and a runtime system that manages the partitioned execution of an intelligent application. Figure 10 shows the design of `Neurosurgeon`, which has two stages: deployment and runtime.

At Deployment – `Neurosurgeon` profiles the mobile device and the server to generate performance prediction models for the spectrum of DNN layer types (enumerated in Section 4.1). Note that `Neurosurgeon`'s profiling is application

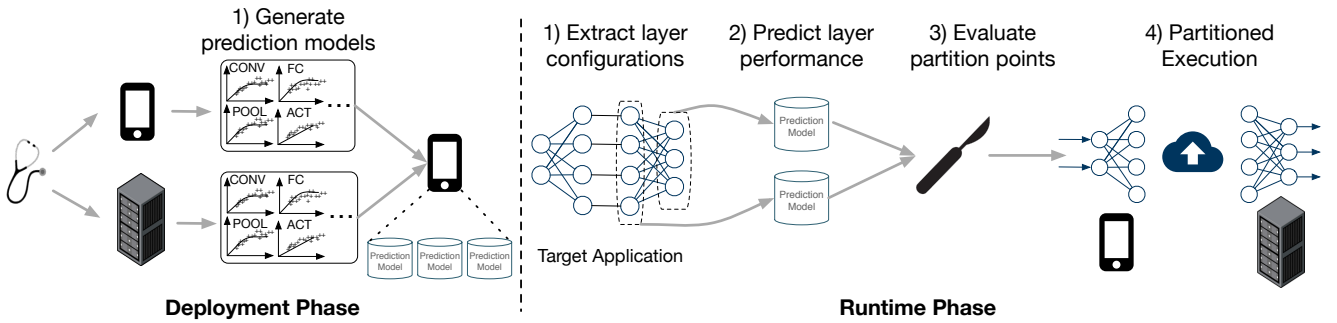


Figure 10: Overview of `Neurosurgeon`. At deployment, `Neurosurgeon` generates prediction models for each layer type. During runtime, `Neurosurgeon` predicts each layer's latency/energy cost based on the layer's type and configuration, and selects the best partition point based on various dynamic factors.

agnostic and only needs to be done once for a given set of mobile and server platforms; per-application profiling is not needed. This set of prediction models are stored on the mobile device and later used to predict the latency and energy cost of each layer (Section 5.1).

During Runtime – During the execution of an DNN-based intelligent application on the mobile device, `Neurosurgeon` dynamically decides the best partition point for the DNN. As illustrated in Figure 10, the steps are as follows: 1) `Neurosurgeon` analyzes and extracts the DNN architecture's layer types and configurations; 2) the system uses the stored layer performance prediction models to estimate the latency and energy consumption for executing each layer on the mobile and cloud; 3) with these predictions, combined with the current wireless connection bandwidth and datacenter load level, `Neurosurgeon` selects the best partition point, optimizing for best end-to-end latency or best mobile energy consumption; 4) `Neurosurgeon` executes the DNN, partitioning work between the mobile and cloud.

5.1 Performance Prediction Model

`Neurosurgeon` models the per-layer latency and the energy consumption of arbitrary neural network architecture. This approach allows `Neurosurgeon` to estimate the latency and energy consumption of a DNN's constituent layers without executing the DNN.

We observe that for each layer type, there is a large latency variation across layer configurations. Thus, to construct the prediction model for each layer type, we vary the configurable parameters of the layer and measure the latency and power consumption for each configuration. Using these profiles, we establish a regression model for each layer type to predict the latency and power of the layer based on its configuration. We describe each layer's regression model variables later in this section. We use GFLOPS (Giga Floating Point Operations per Second) as our performance metric. Based on the layer type, we use either a logarithmic or linear function as the regression function. The logarithmic-based regression is used to model the performance plateau as the computation requirement of the layer approaches the limit of the available hardware resources.

Convolution, local and pooling layers' configurable parameters include the input feature map dimension, number, size and stride of the filters. The regression model for convolution layer is based on two variables: the number of features in the input feature maps, and $(filter\ size/stride)^2 \times (\#\ of\ filters)$, which represents the amount of computation applied to each pixel in the input feature maps. For local and pooling layers, we use the size of the input and output feature maps as the regression model variables.

In a **fully-connected** layer, the input data is multiplied by the learned weight matrix to generate the output vector. We use the number of input neurons and number of output neurons as the regression model variables. **Softmax** and **argmax** layers are handled similarly.

Activation layers have fewer configurable parameters compared to other layers because activation layers have a one-to-one mapping between their input data and output. We use the number of neurons as the regression model variable. We apply the same approach to **normalization** layers.

As previously mentioned, it is a one-time profiling step required for each mobile and server hardware platform to generate a set of prediction models. The models enable `Neurosurgeon` to estimate the latency and energy cost of each layer based its configuration, which allows `Neurosurgeon` to support future neural network architectures without additional profiling overhead.

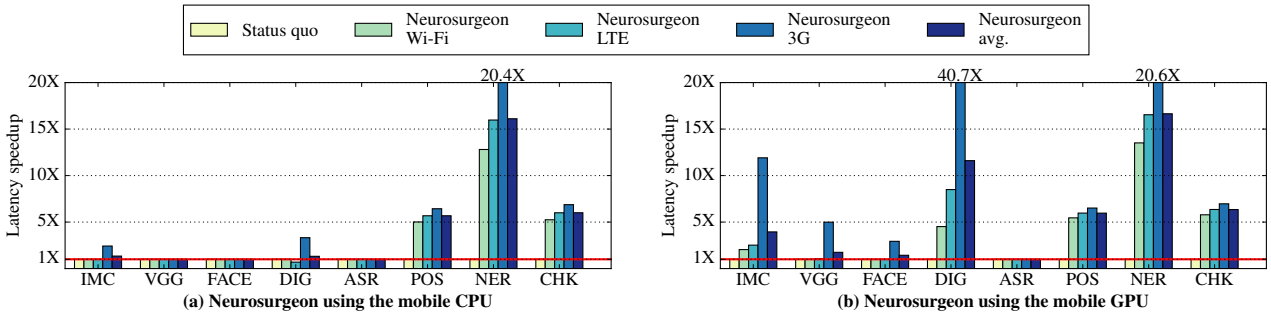
5.2 Dynamic DNN Partitioning

Utilizing the layer performance prediction models, `Neurosurgeon` dynamically selects the best DNN partition points, as described in Algorithm 1. The algorithm has two-steps: analysis of the target DNN and partition point selection.

Analysis of the Target DNN – `Neurosurgeon` analyzes the target DNN's constituent layers, and uses the prediction models to estimate, for each layer, the latency on mobile and cloud, and power consumption on the mobile. Specifically, at lines 11 and 12 of Algorithm 1, `Neurosurgeon` extracts each layer's type and configuration (L_i) and uses the regression models to predict the latency of executing layer L_i on

Table 4: `Neurosurgeon`'s partition point selections for best end-to-end latency. Green block indicates `Neurosurgeon` makes the optimal partition choice and white block means a suboptimal partition point is picked. On average, `Neurosurgeon` achieves within 98.5% of the optimal performance.

Mobile	Wireless network	Benchmarks							
		IMC	VGG	FACE	DIG	ASR	POS	NER	CHK
CPU	Wi-Fi	input	input	input	input	input	fc3		
	LTE	input	input	input	argmax	input	fc3		
	3G	argmax	input	input	argmax	input	fc3		
GPU	Wi-Fi	pool5	input	input	argmax	input	fc3		
	LTE	argmax	argmax	input	argmax	input	fc3		
	3G	argmax	argmax	argmax	argmax	input	fc3		



(a) Neurosurgeon using the mobile CPU

(b) Neurosurgeon using the mobile GPU

Figure 11: Latency speedup achieved by `Neurosurgeon` normalized to status quo approach (executing entire DNN in the cloud). Results for three wireless networks (Wi-Fi, LTE and 3G) and mobile CPU and GPU are shown here. `Neurosurgeon` improves the end-to-end DNN inference latency by 3.1× on average (geometric mean) and up to 40.7×.

mobile (TM_i) and cloud (TC_i), while taking into consideration of current datacenter load level (K). Line 13 estimates the power of executing layer L_i on the mobile device (PM_i) and line 14 calculates the wireless data transfer latency (TU_i) based on the latest wireless network bandwidth.

Partition Point Selection – `Neurosurgeon` then selects the best partition point. The candidate points are after each layer. Lines 16 and 18 evaluate the performance when partitioning at each candidate point and select the point for either best end-to-end latency or best mobile energy consumption. Because of the simplicity of the regression models, this evaluation is lightweight and efficient.

Algorithm 1 `Neurosurgeon` DNN partitioning algorithm

1: **Input:**
2: N: number of layers in the DNN
3: $\{L_i | i = 1 \cdots N\}$: layers in the DNN
4: $\{D_i | i = 1 \cdots N\}$: data size at each layer
5: $f, g(L_i)$: regression models predicting the latency and power of executing L_i
6: K: current datacenter load level
7: B: current wireless network uplink bandwidth
8: PU: wireless network uplink power consumption
9: **procedure** PARTITIONDECISION
10: **for each** i *in* $1 \cdots N$ **do**
11: $TM_i \leftarrow f_{mobile}(L_i)$
12: $TC_i \leftarrow f_{cloud}(L_i, K)$
13: $PM_i \leftarrow g_{mobile}(L_i)$
14: $TU_i \leftarrow D_i / B$
15: **if** $OptTarget == latency$ **then**
16: **return** $\arg\min_{j=1\cdots N}(\sum_{i=1}^{j} TM_i + \sum_{k=j+1}^{N} TC_k + TU_j)$
17: **else if** $OptTarget == energy$ **then**
18: **return** $\arg\min_{j=1\cdots N}(\sum_{i=1}^{j} TM_i \times PM_i + TU_j \times PU)$

5.3 Partitioned Execution

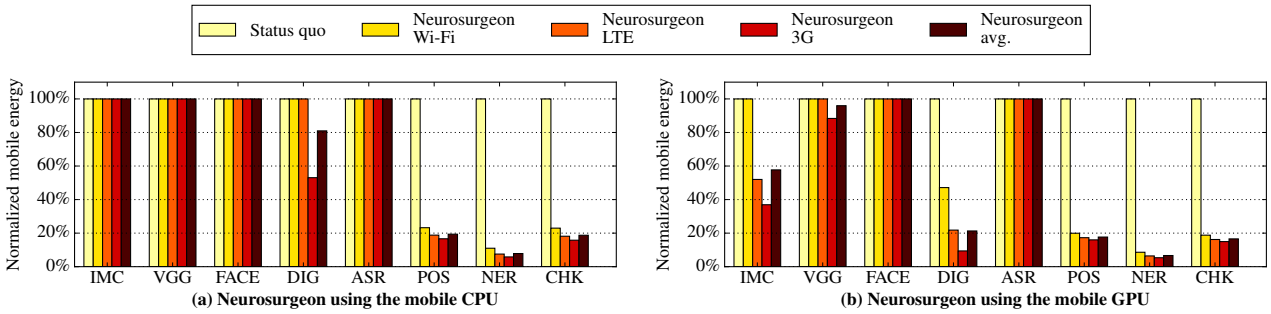
We prototype `Neurosurgeon` by creating modified instances of Caffe [18] to serve as our mobile-side (`NSmobile`) and server-side (`NSserver`) infrastructures. Through these two variations of Caffe, we implement our client-server interface using Thrift [33], an open source flexible RPC interface for inter-process communication. To allow for flexibility in the dynamic selection of partition points, both `NSmobile` and `NSserver` host complete DNN models, and partition points are enforced by `NSmobile` and `NSserver` runtime. Given a partition decision by `NSmobile`, execution begins on the mobile device and cascades through the layers of the DNN leading up to that partition point. Upon completion of that layer, `NSmobile` sends the output of that layer from the mobile device to `NSserver` residing on the server side. `NSserver` then executes the remaining DNN layers. Upon the completion of the DNN execution, the final result is sent back to `NSmobile` on the mobile device from `NSserver`. Note that there is exactly one partition point within the DNN for which information is sent from the mobile device to the cloud.

6. Evaluation

We evaluate `Neurosurgeon` using 8 DNNs (Table 3) as our benchmarks across Wi-Fi, LTE and 3G wireless connections with both CPU-only and GPU mobile platforms. We demonstrate `Neurosurgeon` achieves significant end-to-end latency and mobile energy improvements over the status quo cloud-only approach (Sections 6.1 and 6.2). We then compare `Neurosurgeon` against MAUI [34], a well-known

Table 5: `Neurosurgeon` partition point selections for best mobile energy consumption. Green block indicates `Neurosurgeon` makes the optimal partition choice and white block means a suboptimal partition point is picked. On average, `Neurosurgeon` achieves a mobile energy reduction within 98.8% of the optimal reduction.

Mobile	Wireless network	Benchmarks							
		IMC	VGG	FACE	DIG	ASR	POS	NER	CHK
CPU	Wi-Fi	input	input	input	input	input	fc3		
	LTE	input	input	input	input	input	fc3		
	3G	input	input	input	argmax	input	fc3		
GPU	Wi-Fi	input	input	input	argmax	input	fc3		
	LTE	pool5	input	input	argmax	input	fc3		
	3G	argmax	argmax	input	argmax	input	fc3		



(a) Neurosurgeon using the mobile CPU

(b) Neurosurgeon using the mobile GPU

Figure 12: Mobile energy consumption achieved by `Neurosurgeon` normalized to status quo approach (executing entire DNN in the cloud). Results for three wireless networks (Wi-Fi, LTE and 3G) and mobile CPU and GPU are shown here. `Neurosurgeon` reduces the mobile energy consumption by 59.5% on average (geometric mean) and up to 94.7%.

computation offloading framework (Section 6.3). We also evaluate `Neurosurgeon`'s robustness to variations in wireless network connections (Section 6.4) and server load (Section 6.5), demonstrating the need for such a dynamic runtime system. Finally, we evaluate the datacenter throughput improvement `Neurosurgeon` achieves by pushing compute out of the cloud to the mobile device (Section 6.6).

6.1 Latency Improvement

Partition Point Selection – Table 4 summarizes the partition points selected by `Neurosurgeon` optimizing for latency across the 48 configurations (i.e., 8 benchmarks, 3 wireless network types, mobile CPU and GPU). The green cells indicate when `Neurosurgeon` selects the optimal partition point and achieves the best speedup while the white cells indicate `Neurosurgeon` selects a suboptimal point. `Neurosurgeon` selects the best partition point for 44 out of the 48 configurations. The mispredictions occur because the partition points and its associated performance are very close to one another and thus a small difference in `Neurosurgeon`'s latency prediction shifts the selection. Across all benchmarks and configurations, `Neurosurgeon` achieves latency speedup within 98.5% of optimal speedup.

Latency Improvement – Figure 11 shows `Neurosurgeon`'s latency improvement over the status quo approach, across the 8 benchmarks on Wi-Fi, LTE, and 3G. Figure 11a shows the latency improvement when applying `Neurosurgeon` to a mobile platform equipped with a CPU, and Figure 11b shows that of a mobile platform with a GPU. For CV applications, `Neurosurgeon` identifies the best partition points for 20 out of 24 cases and achieves significant latency speedups, especially when the mobile GPU is available. For the NLP applications, `Neurosurgeon` achieves significant latency speedups even when Wi-Fi is available. For ASR, `Neurosurgeon` successfully identifies that it is best to execute the DNN entirely on the server and, therefore `Neurosurgeon` performs similar to the status quo for that particular benchmark. Across all benchmarks and configurations, `Neurosurgeon` achieves a latency speedup of 3.1× on average and up to 40.7× over the status quo approach.

6.2 Energy Improvement

Partition Point Selection – Table 5 summarizes the partition points identified by `Neurosurgeon` for best mobile energy. `Neurosurgeon` selects the best partition point for 44 out of the 48 configurations. For the suboptimal choices, `Neurosurgeon` consumes 24.2% less energy on average than the status quo approach.

Energy Improvement – Figure 12 shows the mobile energy consumption achieved by `Neurosurgeon`, normalized to the status quo approach. Figure 12a and 12b present results for CPU-only mobile platform and GPU-equipped mobile platform, respectively. When optimizing for best energy consumption, `Neurosurgeon` achieves on average a 59.5% reduction in mobile energy and up to 94.7% reduction over the status quo. Similar to the improvement for latency, the energy reduction is also higher for most benchmarks when the mobile platform is equipped with a GPU.

6.3 Comparing Neurosurgeon to MAUI

In this section, we compare `Neurosurgeon` to MAUI [34], a general offloading framework. Note that MAUI is control-

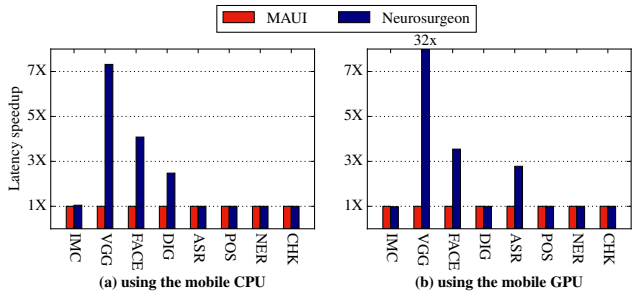


Figure 13: Latency speedup achieved by `Neurosurgeon` vs. MAUI [34]. For MAUI, we assume the optimal programmer annotation that achieves minimal program state transfer. `Neurosurgeon` outperforms MAUI by up to 32× and 1.9× on average.

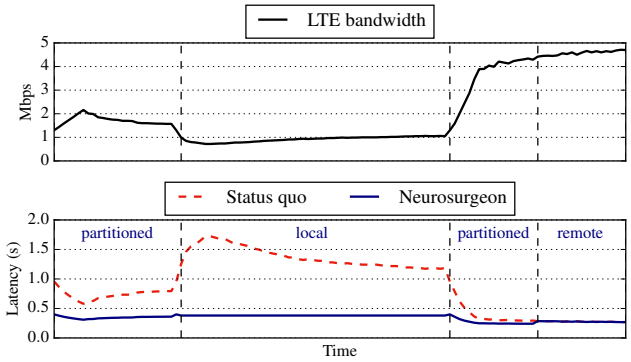


Figure 14: The top graph shows bandwidth variance using a LTE network. The bottom graph shows the latency of AlexNet (IMC) of the status quo and `Neurosurgeon`. `Neurosurgeon`'s decisions are annotated on the bottom graph. `Neurosurgeon` provides consistent latency by adjusting its partitioned execution based on the available bandwidth.

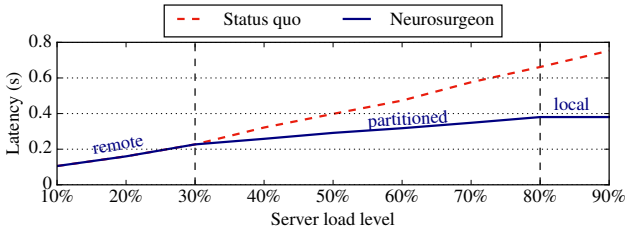


Figure 15: `Neurosurgeon` adjusts its partitioned execution as the result of varying datacenter load.

centric, reasoning and making decisions about regions of code (functions), whereas `Neurosurgeon` is data-centric, making partition decisions based on the structure of the data topology that can differ even if the same code region (function) is called.

Figure 13 presents the latency speedup achieved by `Neurosurgeon` normalized to MAUI when executing the 8 DNN benchmarks, averaged across three wireless network types. Figure 13a presents the result when applying MAUI and `Neurosurgeon` on a CPU-only mobile platform and Figure 13b presents the result on a mobile platform equipped with a GPU. In this experiment, we assume that for MAUI, programmers have optimally annotated the minimal program states that need to be transferred.

Figure 13 shows that `Neurosurgeon` significantly outperforms MAUI on the computer vision applications. For the NLP applications, both `Neurosurgeon` and MAUI correctly decide that local computation on the mobile device is optimal. However, MAUI makes incorrect offloading choices for more complicated scenarios (e.g., VGG, FACE, DIG and ASR). This is because MAUI relies on past invocation of a certain DNN layer type to predict the latency and data size of the future invocations of that layer type, leading to mispredictions. This control-centric prediction mechanism is not suitable for DNN layers because the latency and data size of layers of the same type can be drastically different within one DNN, and `Neurosurgeon`'s DNN analysis step and prediction model correctly captures this variation. For instance, in VGG, the input data size for the first and second convolution layers are significantly different: 0.57MB for `conv1.1`, and 12.25MB for `conv1.2`. For the mobile CPU and LTE, MAUI decides to offload the DNN before `conv1.2` due to its misprediction, uploading large amount of data and resulting in a 20.5× slowdown over the status quo approach. Meanwhile, `Neurosurgeon` successfully identifies that for this case it is best to execute the DNN entirely in the cloud, and thus achieves similar performance as the status quo and a 20.5× speedup over MAUI.

6.4 Network Variation

In this section, we evaluate `Neurosurgeon`'s resilience to real-world measured wireless network variations. In Figure 14, the top graph shows measured wireless bandwidth of T-Mobile LTE network over a period of time. The bottom graph shows the end-to-end latency of the status quo approach and `Neurosurgeon` executing AlexNet (IMC) on the mobile CPU platform. Annotated on the bottom graph is `Neurosurgeon`'s dynamic execution choice, categorized as either local, remote or partitioned. The status quo approach is highly susceptible to network variations and consequently the application suffers significant latency increases during the low bandwidth phase. Conversely, `Neurosurgeon` successfully mitigates the effects of large variations and provides consistent low latency by shifting partition choice to adjust the amount of data transfer based on the available bandwidth.

6.5 Server Load Variation

In this section, we evaluate how `Neurosurgeon` makes dynamic decision as the server load varies. Datacenters typically experience diurnal load patterns and high server utilization leads to increased service time for DNN queries. `Neurosurgeon` determines the best partition point based on the current server load level obtained by periodically pinging

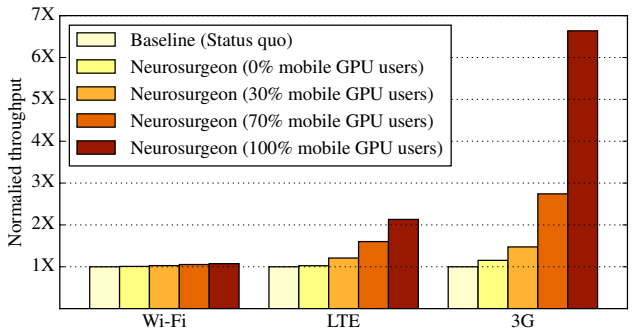


Figure 16: Datacenter throughput improvement achieved by `Neurosurgeon` over the status quo approach. Higher throughput improvement is achieved by `Neurosurgeon` for cellular networks (LTE and 3G) and as more mobile devices are equipped with GPUs.

the server during idle period, and thus avoids long latency caused by high user demand and the resulting high load.

Figure 15 presents the end-to-end latency of AlexNet (IMC) achieved by the status quo approach and `Neurosurgeon` as the server load increases. The mobile device is equipped with a CPU and transfers data via Wi-Fi. As shown in the figure, the status quo approach does not dynamically adapt to varying server load and thus suffers from significant performance degradation when the server load is high. The end-to-end latency of the status quo approach increases from 105ms to 753ms as the server approaches its peak load level. On the other hand, by taking server load into consideration, `Neurosurgeon` dynamically adapts the partition point. In Figure 15, two vertical dashed lines represent the points where `Neurosurgeon` changes its selection: from complete cloud execution at low load, to partitioning the DNN between mobile and cloud at medium load, and eventually completely onloading to mobile at peak load. Regardless of the server load, `Neurosurgeon` keeps the end-to-end latency of executing image classification below 380ms. By considering server load and its impact on the server performance, `Neurosurgeon` consistently delivers the best latency regardless of the variation in server load.

6.6 Datacenter Throughput Improvement

`Neurosurgeon` onloads part or all of the computation from the cloud to mobile devices to improve end-to-end latency and reduce mobile energy consumption. This new compute paradigm reduces the computation required on the datacenter, leading to shorter query service time and higher query throughput. In this section, we evaluate `Neurosurgeon`'s effectiveness in this aspect. We use BigHouse [38] to compare the achieved datacenter throughput between status quo and `Neurosurgeon`. The incoming DNN queries are composed evenly of the 8 DNNs in the benchmark suite. We use the measured mean service time of DNN queries combined with Google web search query distribution for the query inter-arrival rate.

Figure 16 presents the datacenter throughput improvement achieved by `Neurosurgeon`, normalized to the baseline status quo approach of executing the entire computation on the server. Each cluster presents results for a given wireless network type. Within each cluster, the first bar represents the status quo cloud-only approach, while the other four bars represent `Neurosurgeon` with different compositions of the mobile hardware. For example, "30% Mobile GPU users" indicates 30% of the incoming requests are from mobile devices equipped with a GPU while the remaining 70% are from devices equipped only with a CPU.

When the mobile clients are connected to the server via fast Wi-Fi network, `Neurosurgeon` achieves on average 1.04× throughput improvement. As the wireless connection changes to LTE and 3G, the throughput improvement becomes more significant: 1.43× for LTE and 2.36× for 3G. `Neurosurgeon` adapts its partition choice and pushes larger portions of the DNN computation to the mobile devices as the wireless connection quality becomes less ideal. Therefore the average request query service time is reduced and a higher throughput is achieved in the datacenter. We also observe that as the percentage of mobile devices with GPU increases, `Neurosurgeon` increases the computation onloading from the cloud to mobile, leading to higher datacenter throughput improvement.

7. Related Work

Previous research efforts focus on offloading computation from the mobile to cloud. In Table 6, we compare `Neurosurgeon` with the most relevant techniques on properties including whether there is heavy data transfer overhead, data-centric or control-centric partitioning, low run-time overhead, whether application-specific profiling is required, and whether programmer's annotation is needed.

In addition to these key differences, computation partition frameworks have to make predictions as to when to offload computation and the correctness of the prediction dictates the final performance improvements for the application. COMET [35] offloads a thread when its execution time exceeds a pre-defined threshold, ignoring any other information (amount of data to transfer, wireless network available, etc.). Odessa [36] makes computation partition decisions only considering the execution time and data requirements of part of the function, without taking the entire application into consideration. CloneCloud [37] makes the same offloading decisions for all invocations of the same function. MAUI's [34] offloading decision mechanism is better in that it makes predictions for each function invocation separately and considers the entire application when choosing which function to offload. However, MAUI is not applicable for the computation partition performed by `Neurosurgeon` for a number of reasons: 1) MAUI requires a profiling step for each individual application, whereas predictions are required to perform DNN partitioning. `Neurosurgeon` makes deci-

Table 6: Comparing Neurosurgeon to popular computation offloading/partition frameworks

	MAUI [34]	Comet [35]	Odessa [36]	CloneCloud [37]	Neurosurgeon
No need to transfer program state			✓		✓
Data-centric compute partitioning					✓
Low/no runtime overhead	✓		✓	✓	✓
Requires no application-specific profiling		✓			✓
No programmer annotation needed		✓	✓	✓	✓
Server load sensitive			✓		✓

sions based on the DNN topology without any runtime profiling. 2) MAUI is control-centric, making decisions about regions of code (functions), whereas Neurosurgeon makes partition decisions based on the structure of the data topology that can differ even if the same code region (function) is executed. Layers of a given type (even if mapped to the same function) within one DNN can have significantly different compute and data characteristics. 3) Neurosurgeon transfers only the data that is being processed in contrast to transferring all program state. 4) MAUI requires the programmer to annotate their programs to identify which methods are "offload-able".

In addition to prior work investigating the utilization and efficiency of datacenter systems [39–52], there has been growing interest in building large scale datacenter systems for Deep Neural Network workloads. Various accelerators, such as GPUs, ASICs, and FPGAs, have been proposed for datacenters to better handle DNN computation [9, 53–55]. There has also been effort in designing compact DNNs suitable for the mobile edge. Microsoft and Google explore small-scale DNNs for speech recognition on mobile platforms [56, 57]. MCDNN [58] proposes generating alternative DNN models to trade-off accuracy for performance/energy and choosing to execute either in the cloud or on the mobile. This work investigates intelligent collaboration between the mobile device and cloud for executing traditionally cloud-only large-scale DNNs for reduced latency and energy consumption without sacrificing the DNNs' high prediction accuracy.

8. Conclusion

As an essential component of today's intelligent applications, Deep Neural Networks have been traditionally executed in the cloud. In this work, we examine the efficacy of this status quo approach of cloud-only processing and show that it is not always optimal to transfer the input data to the server and remotely execute the DNN. We investigate the compute and data characteristics of 8 DNN architectures spanning computer vision, speech, and natural language processing applications and show the trade-off of partitioning computation at different points within the neural network. With these insights, we develop Neurosurgeon, a system that can automatically partition DNN between the mobile device and cloud at the granularity of neural network layers. Neurosurgeon adapts to various DNN architectures, hardware platforms, wireless connections, and server load levels, and chooses the partition point for best latency and best mobile energy consumption. Across 8 benchmarks, when compared to cloud-only processing, Neurosurgeon achieves on average 3.1× and up to 40.7× latency speedup, reduces mobile energy consumption by on average 59.5% and up to 94.7%, and improves datacenter throughput by on average 1.5× and up to 6.7×.

9. Acknowledgment

We thank our anonymous reviewers for their feedback and suggestions. This work was sponsored by ARM, Intel, and the National Science Foundation under grants IIS-VEC-1539011, CCF-SHF-1302682, CNS-CSR-1321047 and NSF CAREER SHF-1553485.

References

[1] Wearables market to be worth $25 billion by 2019. http://www.ccsinsight.com/press/company-news/2332-wearables-market-to-be-worth-25-billion-by-2019-reveals-ccs-insight. Accessed: 2017-01.

[2] Rapid Expansion Projected for Smart Home Devices, IHS Markit Says. http://news.ihsmarkit.com/press-release/technology/rapid-expansion-projected-smart-home-devices-ihs-markit-says. Accessed: 2017-01.

[3] Intelligent Virtual Assistant Market Worth $3.07Bn By 2020. https://globenewswire.com/news-release/2015/12/17/796353/0/en/Intelligent-Virtual-Assistant-Market-Worth-3-07Bn-By-2020.html. Accessed: 2016-08.

[4] Intelligent Virtual Assistant Market Analysis And Segment Forecasts 2015 To 2022. https://www.hexaresearch.com/research-report/intelligent-virtual-assistant-industry/. Accessed: 2016-08.

[5] Growing Focus on Strengthening Customer Relations Spurs Adoption of Intelligent Virtual Assistant Technology. http://www.transparencymarketresearch.com/pressrelease/intelligent-virtual-assistant-industry.html/. Accessed: 2016-08.

[6] Google Brain. https://backchannel.com/google-search-will-be-your-next-brain-5207c26e4523#.x9n2ajota. Accessed: 2017-01.

[7] Microsoft Deep Learning Outperforms Humans in Image Recognition. http://www.forbes.com/sites/michaelthomsen/2015/02/19/microsofts-deep-learning-project-outperforms-humans-in-image-recognition/. Accessed: 2016-08.

[8] Baidu Supercomputer. https://gigaom.com/2015/01/14/baidu-has-built-a-supercomputer-for-deep-learning/. Accessed: 2016-08.

[9] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISCA '15, New York, NY, USA, 2015. ACM.

[10] The 'Google Brain' is a real thing but very few people have seen it. http://www.businessinsider.com/what-is-google-brain-2016-9. Accessed: 2017-01.

[11] Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html. Accessed: 2017-01.

[12] Apple's Massive New Data Center Set To Host Nuance Tech. http://techcrunch.com/2011/05/09/apple-nuance-data-center-deal/. Accessed: 2016-08.

[13] Apple moves to third-generation Siri back-end, built on open-source Mesos platform. http://9to5mac.com/2015/04/27/siri-backend-mesos/. Accessed: 2016-08.

[14] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 64–76. IEEE, 2016.

[15] Whitepaper: NVIDIA Tegra X1. Technical report. Accessed: 2017-01.

[16] NVIDIA Jetson TK1 Development Kit: Bringing GPU-accelerated computing to Embedded Systems. Technical report. Accessed: 2017-01.

[17] Nvidia's Tegra K1 at the Heart of Google's Nexus 9. http://www.pcmag.com/article2/0,2817,2470740,00.asp. Accessed: 2016-08.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[19] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2013.

[20] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[22] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Andrew Ng. Deep learning with cots hpc systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.

[23] TestMyNet: Internet Speed Test. http://testmy.net/. Accessed: 2015-02.

[24] Watts Up? Power Meter. https://www.wattsupmeters.com/. Accessed: 2015-05.

[25] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[26] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[27] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR)*, 2014.

[28] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[29] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. The kaldi speech recognition toolkit. In *Proc. ASRU*, 2011.

[30] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 2011.

[31] Ashkan Nikravesh, David R Choffnes, Ethan Katz-Bassett, Z Morley Mao, and Matt Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *International Conference on Passive and Active Network Measurement*, pages 12–22. Springer, 2014.

[32] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 301–312. IEEE Press, 2014.

[33] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[34] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[35] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently.

[36] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011.

[37] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[38] David Meisner, Junjie Wu, and Thomas F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. *ISPASS '12: International Symposium on Performance Analysis of Systems and Software*, April 2012.

[39] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ronald G. Dreslinski. Adrenaline: Pinpointing and reigning in tail queries with quick voltage boosting. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[40] Michael A. Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[41] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[42] Vinicius Petrucci, Michael A. Laurenzano, Yunqi Zhang, John Doherty, Daniel Mosse, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicore in warehouse scale computers. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[43] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[44] Johann Hauswald, Tom Manville, Qi Zheng, Ronald G. Dreslinski, Chaitali Chakrabarti, and Trevor Mudge. A hybrid approach to offloading mobile image classification. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[45] 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.

[46] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[47] Matt Skach, Manish Arora, Chang-Hong Hsu, Qi Li, Dean Tullsen, Lingjia Tang, and Jason Mars. Thermal time shifting: Leveraging phase change materials to reduce cooling costs in warehouse-scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISCA '15, 2015.

[48] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real system smt processors to improve utilization in warehouse scale computers. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[49] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *ACM SIGPLAN Notices*, volume 51, pages 681–696. ACM, 2016.

[50] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 456–468. IEEE, 2016.

[51] Animesh Jain, Michael A Laurenzano, Lingjia Tang, and Jason Mars. Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[52] Michael A. Laurenzano, Yunqi Zhang, Jiang Chen, Lingjia Tang, and Jason Mars. Powerchop: Identifying and managing non-critical units in hybrid processor architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 140–152, Piscataway, NJ, USA, 2016. IEEE Press.

[53] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 269–284. ACM, 2014.

[54] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–381. ACM, 2015.

[55] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11), 2015.

[56] Xin Lei, Andrew Senior, Alexander Gruenstein, and Jeffrey Sorensen. Accurate and Compact Large vocabulary speech recognition on mobile devices. In *INTERSPEECH*, pages 662–665, 2013.

[57] Xin Lei, Andrew Senior, Alexander Gruenstein, and Jeffrey Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. In *INTERSPEECH*, pages 662–665, 2013.

[58] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An execution framework for deep neural networks on resource-constrained devices. In *MobiSys*, 2016.