2023 IEEE International Conference on Edge Computing and Communications (EDGE) | 979-8-3503-0483-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/EDGE60047.2023.00037

Offload Shaping for Wearable Cognitive Assistance

Roger Iyengar, Qifei Dong, Chanh Nguyen
Carnegie Mellon University
{raiyenga, qifeid, chanhn}@cs.cmu.edu

Padmanabhan Pillai
Intel Labs
padmanabhan.s.pillai@intel.com

Mahadev Satyanarayanan
Carnegie Mellon University
satya@cs.cmu.edu

***Abstract*—Edge computing has much lower elasticity than cloud computing because cloudlets have much smaller physical and electrical footprints than a data center. This hurts the scalability of applications that involve low-latency edge offload. We show how this problem can be addressed by leveraging the growing sophistication and compute capability of recent wearable devices. We investigate four Wearable Cognitive Assistance applications on three wearable devices, and show that the technique of *offload shaping* can significantly reduce network utilization and cloudlet load without compromising accuracy or performance.**

***Index Terms*—Computer Vision, Machine Learning, Offloading, Wearable Computing, Mobile Computing, Edge Computing, IoT, Cloudlet, Augmented Reality, 5G, Wi-Fi**

I. Introduction

Offloading compute-intensive operations at low latency from underpowered wearable devices over a wireless network to a nearby cloudlet was one of the original motivations for edge computing [1]. Today, it continues to be an important driver of edge computing, but faces the challenge of *limited elasticity.* A cloudlet is designed for a much smaller physical and electrical footprint than a cloud data center. Hence, modest load spikes can overwhelm a cloudlet and its wireless network. Since low end-to-end latency is non-negotiable for many edge-native applications [2], shifting load to the cloud is not feasible. Techniques that reduce the average utilization of shared resources for edge offload are therefore valuable.

Such a technique is *offload shaping,* originally described in 2015 by Hu et al [3]. That work presented empirical evidence that many instances of offloading are wasted work because of imperfect real-time sensing (e.g., blurry image capture or near-duplicate frames). Offload shaping eliminates the resource demand of this useless work via *early discard* in the processing pipeline that starts at the wearable device. Hu et al showed that offload shaping was possible even with the limited capability of wearable devices. They also demonstrated significant reduction in the utilization of shared resources.

In this paper, we extend the concept of offload shaping by leveraging the growing sophistication and compute capability of recent wearable devices. We observe that on-device hardware accelerators for tasks such as deep learning inference [4], super-resolution [5] and scene analysis [6] have emerged. At the same time, we observe that the complexity and resource demand of the end-to-end processing pipelines have also grown (e.g., because of larger deep neural networks (DNNs)). Consequently, edge offload is still necessary in the worst case. However, we show that the average-case burden of edge offload can be reduced by offload shaping. We further show that this savings can be achieved on diverse wearable devices, without compromising the accuracy or the end-to-end latency of typical processing pipelines.

We explore offload shaping in the context of *wearable cognitive assistance (WCA)* applications for assembly tasks. Originally described in 2014 [7], this genre of applications has emerged as a "killer app" for edge computing because (a) they transmit large volumes of video data from device to cloudlet; (b) they have stringent end-to-end latency requirements; and (c) they make substantial compute demands of the cloudlet, often requiring a GPU. A WCA application runs on a wearable device such as Google Glass® or Microsoft® Hololens®, leaving the user's hands free for task performance. It provides visual and verbal guidance and error detection for a user who is performing an unfamiliar task. We investigate four WCA applications on three wearable devices: Google Glass® Enterprise Edition 2, Vuzix Blade® 2, and Magic Leap 2.

The main contribution of this work is to show that offload shaping can significantly reduce the network utilization and cloudlet load of WCA applications on diverse wearable devices, without compromising accuracy or performance. Our results show that the savings achievable varies across devices, but the concept is robust. Offload shaping is thus a valuable technique for reconciling the conflicting demands of scalability and end-to-end performance for WCA tasks.

The rest of the paper is organized as follows. Section II describes the four WCA applications studied in this work, and the computer vision pipelines associated with them. Sections III and IV describe the two forms of offload shaping we examined, and they present the main experimental results of this paper. Section V concludes the paper.

II. Image Processing in WCA

Table I lists the four WCA applications studied in this paper. Progress on an assembly task is determined via computer vision using DNN models, that were fine-tuned on data for each specific task. For each application, we collect training images depicting each step of the assembly task. We label each image to indicate the step of the task that is displayed, and draw a bounding box around the section of the image that contains the object being assembled. We also collect and label separate sets of test data to evaluate the accuracy of the DNN models.

DNNs with low accuracy will result in a poor user experience. When a frame is misclassified, an application either fails to recognize that the user has successfully completed a step,

2767-9918/23/$31.00 ©2023 IEEE
DOI 10.1109/EDGE60047.2023.00037

TABLE I
THE FOUR WCA APPLICATIONS WE DEVELOPED.

Name	Description
Stirling	Assemble a heat engine from metal parts
Meccano	Build a model bike from metal parts
Toyplane	Build a model helicopter from 3D printed plastic parts
Sanitizer	Assemble a sanitizer for a smartphone from metal and plastic parts

TABLE II
TOP-1 CLASSIFICATION ACCURACY FOR STANDALONE DNN MODELS. THIS IS THE PERCENTAGE OF IMAGES THAT THE MODEL CLASSIFIED CORRECTLY. THE HIGHEST ACCURACY FOR EACH APPLICATION IS IN BOLD.

	Meccano	Stirling	Sanitizer	Toyplane
Resnet 50	69.8%	26.3%	68.3%	56.4%
EfficientDet-Lite0	**75.2%**	53.7%	79.3%	51.1%
EfficientDet-Lite1	71.1%	53.8%	84.1%	63.5%
EfficientDet-Lite2	**75.2%**	**57.8%**	84.9%	59.8%
Fast MPN-COV	73.5%	52.0%	84.0%	**78.0%**
Faster R-CNN	72.3%	50.7%	**91.0%**	67.5%

or it detects that a step has been completed when it hasn't and gives the user a new instruction prematurely.

A. Standalone DNN

We train and test standalone DNNs on the data that we collected for each application. These include both image classifiers and object detectors. Image classifiers are given an image, and assign a label indicating the type of object that is shown in the whole image. When training our image classifiers, we ignore bounding box labels and just train the models using class labels. The image classifiers we tried were Resnet 50 [8] and Fast MPN-COV [9]. Object detectors can find multiple objects present in an image, as opposed to just one. They return bounding box coordinates and class labels for each object present in an image. We evaluate the Faster R-CNN [10] and EfficientDet [11] object detectors. Our evaluation looks at three different versions of EfficientDet. EfficientDet-Lite0 has the smallest number of learned parameters, and EfficientDet-Lite2 has the largest number of learned parameters.

For each application and object detector or image classifier combination, we train a DNN on the training data and test it on the test data. Each training and test image contain exactly one instance of the object being assembled. We compute Top-1 accuracy for the image classifiers by comparing the highest confidence label from the model with the ground truth label we assigned, for each image. Top-1 accuracy is computed for the object detector by comparing the class label of the object that the model detected with the highest confidence score, and the ground truth class label. The bounding box coordinates returned by the object detector are ignored for this evaluation, because the applications do not need to know the location of the object being assembled. They just need to know the step of the assembly task that is shown in an image. As the results in Table II show, the low accuracy of this approach suggests that a standalone DNN is insufficient.

TABLE III
TOP-1 CLASSIFICATION ACCURACY FOR PIPELINES. THE HIGHEST ACCURACY FOR EACH APPLICATION IS IN BOLD.

	Meccano	Stirling	Sanitizer	Toyplane
EfficientDet-Lite0 and Resnet 50	75.0%	85.1%	87.9%	69.8%
EfficientDet-Lite0 and Fast MPN-COV	82.0%	78.4%	79.3%	77.2%
EfficientDet-Lite1 and Resnet 50	74.6%	70.3%	87.7%	70.9%
EfficientDet-Lite1 and Fast MPN-COV	81.7%	66.6%	79.3%	77.7%
EfficientDet-Lite2 and Resnet 50	75.0%	**91.0%**	89.1%	70.1%
EfficientDet-Lite2 and Fast MPN-COV	81.5%	86.0%	80.6%	76.6%
Faster R-CNN and Fast MPN-COV	**84.5%**	80.9%	**92.9%**	**81.9%**

B. Pipeline

In an attempt to increase accuracy, we use a two stage process inspired by [12]. An object detector first finds the region of an image that contains the section of the object that the user is currently assembling. The application then crops the image around this region, and then determines the step of the task that is shown, using an image classifier. As with the standalone DNN implementations of our applications, the image classifier has one class for each step of the task.

We train Faster R-CNN [10] and EfficientDet [11] object detectors with modified versions of our training data for each application. Images were labeled with bounding boxes, but all objects were assigned a single class label.

Our Resnet 50 [8] and Fast MPN-COV [9] image classifiers are trained on images that were cropped to only include the regions inside of our bounding box labels. This allows these models to classify cropped images, rather than the original full images that also contained part of the empty table around the object that was being assembled.

We test the pipelines of models trained for each application on the test set for that application. These results are listed in Table III. The pipeline consisting of Faster R-CNN and Fast MPN-COV achieves the highest accuracy for all applications except Stirling. The best pipeline outperforms the best standalone DNN for all four applications.

III. MAPPING PROCESSING TO COMPUTING TIERS

All of the models and pipelines described in Section II can be run on a cloudlet, with the applications implemented as thin clients. However, cloudlets are a limited resource [13]. Heavy cloudlet load limits the number of other users that can share the cloudlet. We thus examined how wearable devices can reduce the amount of processing on cloudlets. Network bandwidth is also a shared limited resource. Our experiments examine how running some computations locally, instead of offloading them to a cloudlet, can reduce the amount of bandwidth used by WCA applications. Figure 1 shows the three partitioning strategies we explore.

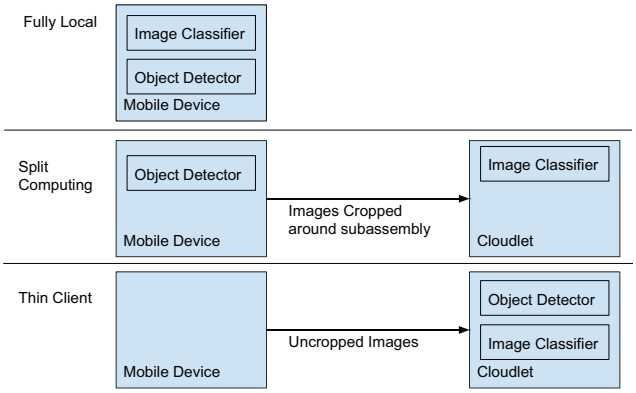

Fig. 1. Fully local, split computing, and thin client implementations of WCA

A. Fully Local Computation

Carrying out all computations locally avoids consuming any cloudlet resources or network bandwidth. Unfortunately, fully local computation limits an application's image processing capabilities to the what the wearable devices are capable of running. In addition, the battery in the devices must power all of these computations. We run experiments to find the accuracy, inference time, and power consumption for models running directly on wearable devices. This indicates whether or not it is practical to run all computations for WCA applications locally on the wearable device.

EfficientDet and Resnet 50 can be run on Android devices using PyTorch Mobile and TensorFlow Lite. However, Fast MPN-COV and Faster R-CNN cannot currently run on mobile devices [14], [15]. All of our pipelines that run locally on the wearable devices were thus limited to using EfficientDet and Resnet 50.

B. Thin Client

Offloading all computations to a cloudlet allows us to process images using Faster R-CNN and Fast MPN-COV. This enables us to use the most accurate pipelines listed in Table III. We measured the inference time and power consumption of a thin client that offloads all computations to a cloudlet. We implemented a flow control mechanism in the thin client that only allows the client to send one image to the cloudlet at a time. After the client sends an image to the cloudlet, it waits for the cloudlet to transmit the result from processing that image before sending the next image. This prevents a buffer of stale frames from building up on the cloudlet, while the cloudlet is busy processing a frame.

All of the systems were Internet-connected: the wearable devices via Wi-Fi, and the cloudlet via 1 Gbps Ethernet. The ping time between the wearable device and the cloudlet was under 5 ms. The cloudlet had two Intel® Xeon® E5-2699v3 processors and an Nvidia® GeForce® GTX 1080 Ti GPU.

The thin client experiments represent the best case scenario: a user with a high bandwidth and low latency connection to a server with a GPU. The wearable device can offload all expensive computations to this server. This saves power on the wearable device, and supports compute-intensive DNNs. Our thin client experiments achieve the highest accuracy, and lowest possible execution time and device power consumption.

C. Split Computing

Split computing is a form of offload shaping to reduce network bandwidth and/or cloudlet resource usage. Preprocessing on the wearable device can reduce the amount of data that must be sent to the cloudlet, and/or replace some of the computations that would have been run on the cloudlet.

As we showed in Section II, a standalone object detector or image classifier is not sufficient for our applications. In addition, implementing the pipeline described in Section II-B using an object detector that is split across a client and a cloudlet is impractical. A split object detector has a head DNN which outputs an embedding, that gets sent to the cloudlet. The cloudlet feeds this embedding to the split object detector's tail, and the tail's output just contains the bounding box coordinates and class labels for the detected objects. There is no way for the cloudlet to obtain a cropped image from the original embedding that was sent to the cloudlet. Our application would either have to send the entire image to the cloudlet along with the embedding, or it would have to send the bounding box coordinates back to the wearable device, and have the wearable device send the cropped image in some form to run the classifier on the cloudlet. The former approach eliminates all of the bandwidth savings that split computing offers; while the latter approach requires a second round trip to the wearable device, which increases latency.

One possibility is running an object detector on the wearable device, cropping the image there, and then feeding the cropped image to a split image classifier. However, this requires running both the object detector and the head of the split image classifier on the wearable device. We instead opt to run just the object detector on the wearable device, transmit the cropped image to the cloudlet, and run the full image classifier there. This allows us to use the Fast MPN-COV image classifier, which cannot be run on a wearable device. In addition, it saves the wearable device from having to run the head of a split image classifier. This relatively simple implementation of split computing does not require ML expertise in DNN splitting. Can such a simple implementation still offer a significant bandwidth savings without unreasonably harming battery life or classification accuracy?

D. Results

Classification accuracy, inference time, and power consumption all impact a user's experience with a WCA application. A model with low accuracy might result in the application failing to recognize a completed step, or prematurely giving the user a new instruction. High inference time results in a large delay between a user completing a step, and the application providing the next instruction. High power consumption will drain the wearable device's battery quickly. The experiments in this section measure these quantities for fully local execution, thin clients, and split computing.

TABLE IV
CLASSIFICATION ACCURACY FOR THE BEST PERFORMING PIPELINE IN EACH SETTING

	Meccano	Stirling	Sanitizer	Toyplane
Fully Local	75.0%	91.0%	89.1%	70.9%
Split Computing	82.0%	91.0%	89.1%	77.7%
Thin client	84.5%	91.0%	92.9%	81.9%

TABLE V
THE BANDWIDTH SAVED BY TRANSMITTING CROPPED IMAGES. IMAGES WERE CROPPED AROUND THE BOUNDING BOXES RETURNED BY EFFICIENTDET-LITE0.

Stirling	Meccano	Toyplane	Sanitizer
79.2%	52.3%	86.8%	94.2%

1) Classification Accuracy: Table IV lists the highest accuracy possible for each implementation type, based on the data from Sections II and III-D3. The best performance for three out of our four applications requires offloading all computations to a cloudlet.

2) Bandwidth Savings: As Table V shows, split computing offers significant bandwidth savings. This is expressed as: $(\text{Bytes}_{\text{full images}} - \text{Bytes}_{\text{cropped images}})/\text{Bytes}_{\text{full images}}$

The bandwidth savings achieved is content-dependant. The distance between the camera and the object being assembled will change how large a subassembly appears in the image, and this will directly impact the number of bytes required to transmit the cropped image. The bandwidth savings of techniques such as image compression or DNN-based split computing vary less based on the specific content in an image.

Transmitting cropped images requires less than 50% of the bandwidth that the uncropped images require, for all of our datasets. The savings is over 90% for the Sanitizer dataset. In many cases, this significant bandwidth savings will be worth the reduction in accuracy that comes along with using split computing instead of offloading all computations.

3) Inference Time: We measured image processing times for pipelines running directly on the wearable devices. We did this by storing our test set on the devices, and running code that looped through each image. Inside the loop, our code ran the pipeline of models that was being timed. The code logged the elapsed time every 20 frames, based on Android's uptime counter. The elapsed times were divided by 20, to get the per-frame inference time. Each pipeline was run for five minutes. Table VI shows our results.

Table VII lists the largest pipeline that meets the latency bounds from [16], on each device. The accuracies of each pipeline, for all applications are listed in Table III. There is a large gap between the accuracy of the pipeline that can meet the latency bounds and the accuracy of the most accurate pipeline, for most device and application combinations. This indicates that fully local computation is not an acceptable strategy in most of our cases.

TABLE VI
INFERENCE TIME FOR ONE FRAME, IN MILLISECONDS, FOR FULLY LOCAL COMPUTATION. FOR EACH CELL, THE AVERAGE COMES BEFORE THE ± SIGN AND THE STANDARD DEVIATION COMES AFTER.

	Google Glass	Magic Leap	Vuzix Blade
EfficientDet-Lite0 and Resnet 50	480 ± 14	161 ± 3	2031 ± 27
EfficientDet-Lite1 and Resnet 50	661 ± 46	183 ± 7	2423 ± 155
EfficientDet-Lite2 and Resnet 50	958 ± 9	222 ± 3	3072 ± 73

TABLE VII
THE LARGEST PIPELINE THAT MEETS TIGHT AND LOOSE LATENCY BOUNDS. "LARGEST" REFERS TO THE NUMBER OF PARAMETERS USED FOR THE PIPELINE'S VERSION OF EFFICIENTDET.

	Tight Bound	Loose Bound
Google Glass®	EfficientDet-Lite0 and Resnet 50	EfficientDet-Lite2 and Resnet 50
Magic Leap	EfficientDet-Lite2 and Resnet 50	EfficientDet-Lite2 and Resnet 50
Vuzix Blade®	None	EfficientDet-Lite1 and Resnet 50

TABLE VIII
SINGLE-FRAME INFERENCE TIME FOR THE THIN CLIENT, IN MILLISECONDS

Google Glass	Magic Leap	Vuzix Blade
166 ± 8	150 ± 8	203 ± 9

Inference time measurements for thin clients are listed in Table VIII. These measurements include the time to transmit images to the cloudlet, process them there, and then transmit results back to the wearable device. As with our other time measurements, the applications were run for five minutes, and elapsed time was recorded every 20 frames. These values were well below the tight latency bounds on all three devices. The thin client offers the highest possible accuracy and the lowest inference time. However, it consumes the largest amount of bandwidth and cloudlet resources.

Table IX lists the per-frame inference times of our split computing pipelines, across all three devices. As with accuracy, the inference time for split computing is in between fully local computations and the thin clients. The pipeline that uses EfficientDet-Lite0 runs on the Google Glass within the tight latency bound from [16]. The pipeline that uses EfficientDet-Lite1 is almost under the tight latency bound when run on Google Glass. All three pipelines run within the tight latency bound on Magic Leap. However, none of the pipelines run within the latency bound on Vuzix Blade.

4) Power Consumption: We measure the amount of power that each of these devices use while running the pipelines fully locally, in a loop. As with our previous experiments, we run each pipeline for five minutes. None of these devices have user serviceable batteries, so we cannot measure power

TABLE IX
SINGLE-FRAME INFERENCE TIME OF SPLIT COMPUTING PIPELINES, IN MILLISECONDS

	Google Glass	Magic Leap	Vuzix Blade
EfficientDet-Lite0 and Fast MPN-COV	308 ± 22	106 ± 7	1120 ± 47
EfficientDet-Lite1 and Fast MPN-COV	622 ± 186	133 ± 6	1778 ± 210
EfficientDet-Lite2 and Fast MPN-COV	779 ± 38	154 ± 4	2156 ± 65

TABLE X
AVERAGE POWER CONSUMPTION, IN WATTS, FOR FULLY LOCAL EXECUTION. THESE MEASUREMENTS ARE RECORDED WHILE THE WEARABLE DEVICE IS RUNNING THE FULL PIPELINE. THE BASELINE APPLICATION DOES NOT CARRY OUT ANY COMPUTATION.

	Google Glass	Magic Leap	Vuzix Blade
Baseline	0.61 ± 0.13	15.1 ± 0.46	0.84 ± 0.15
EfficientDet-Lite0 and Resnet 50	1.43 ± 0.35	18.54 ± 0.37	1.18 ± 0.10
EfficientDet-Lite1 and Resnet 50	1.26 ± 0.29	18.41 ± 0.22	1.24 ± 0.16
EfficientDet-Lite2 and Resnet 50	1.26 ± 0.27	18.55 ± 0.16	1.22 ± 0.13

TABLE XI
AVERAGE POWER CONSUMPTION FOR THIN CLIENTS, IN WATTS

	Google Glass	Magic Leap	Vuzix Blade
Baseline	0.61 ± 0.13	15.1 ± 0.46	0.84 ± 0.15
Thin client	1.30 ± 0.41	14.26 ± 0.33	1.17 ± 0.09

consumption based on the current and voltage that is being supplied to the device by its charger. Instead, we run our code with the devices unplugged, and query for current and voltage readings from Android, using the BatteryManager class. We multiply the voltage and current to compute power. Our code contains a background thread which logs the current and voltage every 100 ms.

Table X lists the power values for each pipeline, running on all three devices. The *baseline* measurements were recorded for an application that shows an empty Android activity, but does not do anything aside from recording current and voltage values in a background thread.

The Magic Leap 2 consumes over 15 watts running the baseline application. The device's depth sensors might consume some of this power, or it could have been spatial mapping code running in the background. None of the pipelines increase the Magic Leap 2's power usage by more than 25% of the power consumed by the baseline. The most dramatic increase over baseline power usage is for EfficientDet-Lite0 and Resnet 50 on Google Glass, with an average power usage of 1.43 Watts. However, this still implies a reasonable battery life. A 3.2 Wh battery can supply 1.43 Watts for over two hours.

The power consumption experiments for thin clients measure the power consumed on the wearable device, but they do not measure the power consumed on the cloudlet. These results are presented in Table XI. Running the thin client on the Vuzix Blade 2 consumes more power than running the baseline application, but less power than running any of the on-device pipelines. The Google Glass consumes slightly more power running the thin client as it does when running the on-device pipelines. However, all three of these clients consume significantly more power than the baseline. The thin client on the Magic Leap consumes slightly less power than the baseline application. There isn't a clear explanation for this, but the difference is fairly small.

Split computing power consumption measurements are listed in Table XII. As with our other power measurements, these measurements were made on the wearable devices, and do not include the power consumed by the cloudlet.

TABLE XII
AVERAGE POWER CONSUMPTION OF WEARABLE DEVICES RUNNING DNN PIPELINES, IN WATTS

	Google Glass	Magic Leap	Vuzix Blade
Baseline	0.61 ± 0.13	15.1 ± 0.46	0.84 ± 0.15
EfficientDet-Lite0 and Fast MPN-COV	1.37 ± 0.11	18.03 ± 0.32	1.27 ± 0.10
EfficientDet-Lite1 and Fast MPN-COV	1.23 ± 0.15	18.31 ± 0.21	1.22 ± 0.10
EfficientDet-Lite2 and Fast MPN-COV	1.21 ± 0.16	18.76 ± 0.25	1.24 ± 0.12

These power measurements are similar to our measurements for fully local execution in Table X. The increase in power consumption, above the baseline, is reasonable for all of three devices running all of the pipelines we tested.

IV. GATING

Most of a user's time running a WCA application is spent completing assembly steps. Applications only need to check if a step has been completed after a user thinks the step is done. A step cannot possibly be complete while a user is in the middle of working on it. This section considers modifying applications, so that users have a way to indicate when they have completed a step. This prevents the applications from having to process any images in between when an instruction is given, and when the user indicates that a step is completed. We will henceforth refer to this strategy as *gating*. During the periods that the applications do not have to process frames, they do not use cloudlet resources or network bandwidth. Gating is thus a form of offload shaping.

When a user indicates that they think a step has been completed, the application will begin processing camera images to verify if this is true. If the application determines that the step has in fact been completed, it will give the user the next instruction and then stop processing frames until the user indicates that this next step has also been completed. If the application determines that a user was mistaken, and a step has not actually been completed, it continues processing camera images until the step is actually completed.

The simplest form of gating requires the user to press a button on the wearable device to indicate that they think a step has been completed. This is trivial to implement. However, it requires the user to move one hand all the way from the object they are assembling to the side of their wearable device. An alternative form of gating we implement uses MediaPipe [17] to determine when a user shows a thumbs up gesture to the camera. The thumbs up gesture is the user's way of indicating that they think a step has been completed. Our last form of gating uses automated speech recognition. The user speaks the words "ready for detection," when they believe that a step is complete. This does not require the user to move their hands away from the object that they are assembling. But it is unlikely to work well in a noisy environment.

A. Experiments

A practical gating method will not significantly increase the amount of time it takes for a user to complete a task

TABLE XIII
AVERAGE POWER CONSUMPTION (IN WATTS) OF DEVICES USING DIFFERENT GATING OPTIONS

	Google Glass	Magic Leap	Vuzix Blade
Baseline	1.63 ± 0.23	22.08 ± 0.80	1.98 ± 0.25
Button	1.32 ± 0.16	21.87 ± 0.70	1.80 ± 0.16
Thumbs Up	1.54 ± 0.29	22.13 ± 0.60	2.29 ± 0.34
Speech	1.40 ± 0.18	23.18 ± 0.58	1.80 ± 0.15

or the amount of power that a wearable device consumes. In addition, a good gating method will reduce network bandwidth substantially. We therefore measure the power consumption, task completion time, and bandwidth usage.

We implement four versions of our Toyplane application. The *baseline* version does not use gating. The *button* version has the user press a button to suggest step completion. The *thumbs up* version has the user make a thumbs up gesture. The *speech* version has the user say "ready for detection."

We implement speech gating using the PocketSphinx continuous speech recognition engine [18]. As the Sphinx developers note, their engine does not use state of the art methods for speech recognition. However, Android does not natively include a continuous speech recognition engine that third party developers can access. The Azure Cognitive Services Speech container has to be run on a server [19]. We note that there may exist a better continuous speech recognition engine that can be run entirely on an Android-based wearable device.

Three users assemble the toy plane using all four implementations of the Toyplane application. All implementations are run using a Vuzix Blade 2 headset. The headset records traces of the user completing the task with each of the applications. We then play back these traces on all three headsets to measure power consumption, task completion time, and bandwidth usage. Our playback application processes frames at the rate they were recorded at. For example, if another wearable device can process a frame with MediaPipe faster than the Vuzix Blade did when the trace was recorded, the playback app will pause until it reaches the timestamp when the frame had been processed in the original trace. This allows us to play back traces in a reproducible way on the Vuzix Blade itself, as well as other headsets that have faster hardware. Computations for gating (such as detecting a thumbs up gesture or recognizing speech) were run on the wearable device. Determining the task step shown in an image was done on the cloudlet.

B. Results

Table XIII lists average power consumption. None of the gating strategies result in significantly more power being consumed. The button gating consumed less power than the baseline on all devices. This was likely a result of the large amount of data that was sent between the cloudlet and the wearable device while the baseline application was running. The button-based gating application does not require any expensive computations to be run on the wearable device.

Table XIV lists average task completion time for each gating strategy. The button gating adds almost no time to the task. The thumbs up and speech gating do lead to a noticeable increases in average completion time. Thumbs up gating increased average completion time by over 40%.

TABLE XIV
AVERAGE TASK COMPLETION TIME (IN MILLISECONDS) FOR EACH GATING STRATEGY

Baseline	Button	Thumbs Up	Speech
55.70 ± 1.88	58.32 ± 10.32	78.37 ± 11.61	67.86 ± 8.92

TABLE XV
AVERAGE BANDWIDTH SAVINGS OVER THE BASELINE FOR EACH GATING STRATEGY

Button	Thumbs Up	Speech
93.5% ± 1.2%	88.1% ± 6%	93.4% ± 1.2%

Table XV lists average bandwidth across our three traces for each gating strategy. All three gating strategies significantly reduce the bandwidth usage.

V. CONCLUSION

In this paper, we have explored the concept of offload shaping for WCA. Our results show that a two stage pipeline consisting of an object detector and an image classifier is effective in four WCA applications. A thin client strategy offers the highest accuracy and lowest latency. However, split computing offers significant bandwidth savings. Gating also offers sizable bandwidth savings.

In terms of mobile hardware, our results show that the Vuzix Blade 2 is too slow to run anything other than a thin client. However the Google Glass Enterprise Edition 2 and the Magic Leap 2 are capable of running some version of each technique discussed in this paper. None of our techniques increase power consumption by a significant percentage on any of the wearable devices.

Looking to the future, mobile device hardware is likely to improve. More compute-intensive DNNs for object detection and image classification may also emerge. The accuracy of DNNs that can be run on future mobile devices within the tight latency bound may also improve. While it is hard to predict the net effect of all these changes, it is clear that offload shaping will continue to be valuable. Quantitative comparisons of fully local, split, and thin client approaches should therefore inform the optimal partitioning strategy at each point in time.

ACKNOWLEDGMENTS

This research was sponsored by the National Science Foundation under award number CNS-2106862. This work was done in the CMU Living Edge Lab, which is supported by Intel, ARM, Vodafone, Deutsche Telekom, CableLabs, Crown Castle, InterDigital, Seagate, Microsoft, the VMware University Research Fund, and the Conklin Kistler family fund. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring entity or the U.S. government.

REFERENCES

[1] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, October-December 2009.

[2] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, "The Seminal Role of Edge-Native Applications," in *2019 IEEE International Conference on Edge Computing (EDGE)*, Milan, Italy, 2019.

[3] W. Hu, B. Amos, Z. Chen, K. Ha, W. Richter, P. Pillai, B. Gilbert, J. Harkes, and M. Satyanarayanan, "The Case for Offload Shaping," in *Proceedings of HotMobile 2015*, Santa Fe, NM, 2015.

[4] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices," in *2016 15th ACM/IEEE Intl. Conf. on Information Processing in Sensor Networks*, 2016.

[5] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "MobiSR: Efficient On-Device Super-Resolution through Heterogeneous Mobile Processors," in *Proceedings of MobiCom 2019*, 2019.

[6] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models Using Wearable Commodity Hardware," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys2017)*, 2017.

[7] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards Wearable Cognitive Assistance," in *MobiSys*, Bretton Woods, NH, June 2014.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[9] P. Li, J. Xie, Q. Wang, and Z. Gao, "Towards faster training of global covariance pooling networks by iterative matrix square root normalization," in *IEEE Int. Conf. on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[10] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems*, 2015.

[11] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[12] T. Gebru, J. Krause, Y. Wang, D. Chen, J. Deng, and L. Fei-Fei, "Fine-grained car detection for visual census estimation," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017, p. 45024508.

[13] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan, "Towards Scalable Edge-Native Applications," in *Proceedings of the Fourth IEEE/ACM Symposium on Edge Computing (SEC 2019)*, Washington, DC, November 2019.

[14] TensorFlow, "Running TF2 Detection API Models on mobile," https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/running_on_mobile_tf2.md, Last accessed May 19, 2023.

[15] PyTorch, "autodiff for user script functions aka torch.jit.script for autograd.Function," https://github.com/pytorch/pytorch/issues/22329, Last accessed May 19, 2023.

[16] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, and M. Satyanarayanan, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, San Jose, California, 2017.

[17] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C.-L. Chang, M. Yong, J. Lee, W.-T. Chang, W. Hua, M. Georg, and M. Grundmann, "Mediapipe: A framework for perceiving and processing reality," in *Third Workshop on Computer Vision for AR/VR at IEEE Computer Vision and Pattern Recognition (CVPR) 2019*, 2019.

[18] CMUSphinx, "PocketSphinx," https://github.com/cmusphinx/pocketsphinx, Last accessed May 19, 2023.

[19] Azure Cognitive Services, "Speech Containers," https://learn.microsoft.com/en-us/azure/cognitive-services/cognitive-services-container-support#speech-containers, Last accessed May 19, 2023.