CLIO: Enabling automatic compilation of deep learning pipelines across IoT and Cloud

Jin Huang[*], Colin Samplawski[*], Deepak Ganesan[*], Benjamin Marlin[*], Heesung Kwon[†]
[*]University of Massachusetts Amherst, [†]US Army Research Laboratory
{jinhuang,csamplawski,dganesan,marlin}@cs.umass.edu,heesung.kwon.civ@mail.mil

ABSTRACT

Recent years have seen dramatic advances in low-power neural accelerators that aim to bring deep learning analytics to IoT devices; simultaneously, there have been considerable advances in the design of low-power radios to enable efficient compute offload from IoT devices to the cloud. Neither is a panacea — deep learning models are often too large for low-power accelerators and bandwidth needs are often too high for low-power radios. While there has been considerable work on deep learning for smartphone-class devices, these methods do not work well for small battery-powered IoT devices that are considerably more resource-constrained.

In this paper, we bridge this gap by designing a continuously tunable method for leveraging both local and remote resources to optimize performance of a deep learning model. **Clio** presents a novel approach to split machine learning models between an IoT device and cloud in a *progressive* manner that adapts to wireless dynamics. We show that this method can be combined with model compression and adaptive model partitioning to create an integrated system for IoT-cloud partitioning. We implement **Clio** on the GAP8 low-power neural accelerator, provide an exhaustive characterization of the operating regimes where each method performs best and show that **Clio** can enable graceful performance degradation as resources diminish.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

edge computing, cloud computing, computation off-loading, deep neural networks

ACM Reference Format:
Jin Huang[*], Colin Samplawski[*], Deepak Ganesan[*], Benjamin Marlin[*], Heesung Kwon[†]. 2020. CLIO: Enabling automatic compilation of deep learning pipelines across IoT and Cloud. In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20), September 21–25, 2020, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372224.3419215

MobiCom '20, September 21–25, 2020, London, United Kingdom

ACM ISBN 978-1-4503-7085-1/20/09…$15.00
https://doi.org/10.1145/3372224.3419215

1 INTRODUCTION

While traditional IoT and sensor networks have focused on connecting simple sensors (e.g., thermostats, humidity sensors, etc), we are seeing a shift towards richer, always-on sensors (e.g. cameras, microphones, IMUs) which have become considerably more energy-efficient in recent years [9, 18, 35, 55]). In turn, these capabilities are driving more sophisticated applications of battery-powered sensors (e.g. home security, mobile health).

These trends have led to an emerging marketplace for associated compute elements that support continuous analytics at low power consumption. Several recently announced neural accelerator chipsets such as GreenWaves GAP8 [51], EtaCompute Tensai [50] and Syntiant NDP [52]) consume less than 100mW, thereby allowing us to envisage Mote-class devices equipped with neural accelerators.

Despite these developments, there remains a substantial gap between available resources on low-power accelerators and resources needed for efficient execution of state-of-art deep learning models. While model compression methods have been successful at squeezing large models into embedded devices like smartphones and Raspberry PIs, low-power accelerators represent a huge step up since they have two to three orders of magnitude less resources than these embedded processors (e.g. the GAP8 has a convolutional accelerator with 1 GFlops whereas Raspberry Pi has 24 GFlops and iPhone X has 350 GFlops). The successes of most model compression methods to date has been with relatively small compression ratios (e.g. less than 2× in compute and memory footprint without significant loss of accuracy [17]) but fitting even the smallest versions of state-of-art deep learning models like ResNet-18, VGG-16, and MobileNetV2 into an IoT device requires a 5–20× compression in compute and memory footprint, which is beyond the capability of existing methods. As a result, existing work on optimizing machine learning models for IoT has used simpler models and relied on hand-crafted optimizations rather state-of-art deep learning models and automated model compression techniques [26]. With either approach, the price is a substantial reduction in prediction accuracy due to the extreme levels of compression required to fit the model into available resources.

An alternative to model compression is to leverage wireless communication for transmitting data to the cloud for processing. Rather than compressing the model, we can instead compress the raw data using lossy compression techniques, transmit it to the cloud, and use cloud resources for prediction. This is particularly attractive since the efficiency of low-power radio front-ends and their duty-cycling efficiency has steadily improved over the last two decades (e.g. BLE, Zigbee, etc [1, 36, 49]). However, the challenge is that low-power radios have limited bandwidth and are also sensitive

to range and environmental dynamics since they transmit at low power levels. This can be problematic in low bandwidth settings since squeezing data into a narrow wireless link via lossy compression can lead to poor inference performance due to information loss.

These developments raise an interesting question — can we simultaneously leverage both model compression (where the model is squeezed to fit local compute resources) and data compression (where the data is squeezed to fit the narrow wireless link) to maximize inference quality? A growing body of research is exploring the benefits of partitioning deep learning models such that the IoT device executes the initial layers of a convolutional neural network (CNN) and transmits the intermediate results to the cloud where the remaining stages of execution occur (e.g. [21, 24, 40]). However, for most state-of-art models, intermediate results from various layers are often quite large and inefficient to transmit using low-bandwidth IoT radios. This issue is exacerbated by the fact that only the early layers of a deep learning model are viable as partition points given compute and memory constraints on the IoT device. In mobile settings, wireless dynamics also presents a significant issue — existing techniques need to be carefully trained for specific bandwidth settings and cannot adjust on-the-fly to dynamic changes in bandwidth.

The CLIO Approach: In this work, we seek to tackle these problems and answer the important fundamental question of how ultra-low compute and communication components can be simultaneously leveraged to improve inference accuracy. We propose **Clio**[1], a novel model compilation framework that provides a continuum of options for deep learning models to be partitioned across a low-power IoT device and the cloud while gracefully degrading performance in the presence of limited and dynamically varying resources. **Clio** achieves this by combining the idea of partitioning a deep learning model together with progressive transmission of intermediate results — put together, these techniques can deal with both resource constraints and resource dynamics. Importantly, **Clio** integrates progressive transmission with model compression and adaptive partitioning in response to wireless dynamics. By doing so, **Clio** significantly expands the possible endpoints for model compilation for IoT-cloud networks.

One of the key benefits of **Clio** is that it allows model compilation techniques to be used even for stringently resource-constrained IoT devices which may have orders of magnitude less resources than the original model requires. By fully utilizing available bandwidth for remote execution, **Clio** limits the portion of the original model that needs to be compiled to the resource-constrained device thereby reducing the extent of compression that is required. This has significant advantages from a deployment perspective — rather than having to hand-tune a model for deployment, automatic model compilation allows us to deploy large models over resource-limited networks of devices.

We present an exhaustive evaluation of **Clio** across a spectrum of compute architectures [6, 51] radio architectures [1, 3, 5] and datasets [14, 25, 37]. We show that **Clio** can be used to compile state-of-art models like MobileNetV2 [38] and ResNet-18 [16] into resource-constrained IoT settings, and can reduce both end-to-end latency and power consumption without a noticeable drop in inference accuracy. We demonstrate significant gains over state-of-art approaches for data compression [32], model compression [17, 33] and partitioned execution [21, 29].

[1]CLIO stands for **C**loud-**IO**T partitioning

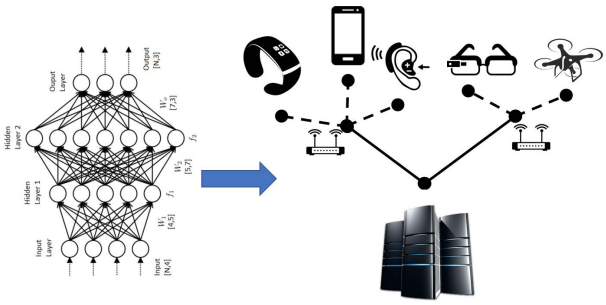


Figure 1: Many IoT devices with rich sensors like cameras are continuously connected to the cloud. We need new methods to execute deep learning models across IoT-Cloud while respecting resource constraints and adapting to wireless variability.

2 CASE FOR CLIO

While there has been significant prior work on deep learning in embedded systems, low-power IoT systems present unique challenges due to their extreme resource constraints.

From a radio perspective, the main challenge is dealing with uncertain wireless conditions. IoT radios operate in a highly duty-cycled manner to save power; however, this comes at the cost of not knowing available bandwidth upon wakeup since the channel may have changed since the previous wake period. The situation is exacerbated by the fact that IoT radios transmit at low power (typically 0dBm) and uplink bandwidth is therefore more sensitive to factors like body blockage, device mobility, and attenuation due to walls. In addition, it is not practical to pre-train and deploy a large number of different models for different available bandwidths in resource constrained settings. Finally, due to loss of synchronization during sleep, there is additional connection latency to start transferring data when an event occurs. This in turn adds to the end-to-end delay for remote computation approaches that rely on the cloud.

From a computation perspective, the key challenge is the lack of sufficient local resources. While low-power accelerators have the architectural building blocks to execute deep learning models quickly, they are still limited by computation ability, memory size and maximum operating frequency. For example, the GAP8 operates at 1 GFlops (with convolutional accelerator), 64KB L1 cache and 512KB L2 cache, and a maximum operating frequency of 175 MHz. This is a challenge since state-of-art deep learning models need two to three orders of magnitude more computational resources than are available.

We now look at canonical approaches for execution of deep learning models at the edge and describe why they are inadequate for low-power IoT scenarios.

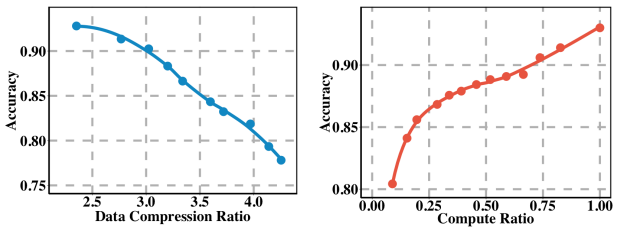


Figure 2: (a) Remote execution is bottlenecked by bandwidth availability. High data compression ratios lead to poor accuracy. (b) Local execution is bottlenecked by compute/memory availability. At low compute, model compression perform particularly poorly.

Data compression vs model compression: Two common methods for dealing with limited resources are data compression (i.e., we squeeze the data volume by transmitting a lossy version of the data for remote execution) and model compression (i.e., we squeeze larger models to execute locally within desired latency bounds). However, in severely resource-constrained scenarios, both approaches can have low prediction performance.

We illustrate this tradeoff in Figure 2 — the figure shows the effect of reducing data transmission size and reducing model size on accuracy when executing the MobileNetV2 model on CIFAR-10 images. On the left is the effect of data compression using JPEG, a popular lossy compression method for images. The curve corresponds to accuracy when executing the MobileNetV2 model on images compressed at different JPEG compression qualities. We train a separate classification model for each JPEG quality to show best-case performance given images of a particular JPEG quality. On the right is the effect of a particular model compression approach based on scaling down the layer widths of the MobileNetV2 model, which reduces the required computation.[2]

The figure shows that accuracy degrades rather quickly once the data or model are reduced in size. Even at relatively low compression levels, the accuracy drops below 90% and steadily reduces thereafter. We see a similar trend for ImageNet.

We note that replacing model compression with hand-crafted models that are specifically designed to fit into resource-constrained MCUs does not resolve the problem. For example, the CMSIS-NN is a convolutional neural network specifically optimized for the ARM STM32 F7 [27], but the model only achieves around 80% accuracy for the CIFAR-10 dataset compared to the 93% accuracy that the full MobileNetV2 model achieves. Hence, such hand-crafted methods can also come with significant performance degradation.

Model partitioning: Another approach that has become popular is edge-cloud model partitioning to take advantage of both remote and local compute resources to optimize performance [15, 21, 29, 41]. While partitioning is appealing, existing methods assume knowledge of bandwidth to determine how to partition the model and do not directly tackle the issues of bandwidth uncertainty and variability that we described earlier. Also, resource constraints and high dynamic range of bandwidth also make it infeasible to store a large number of models optimized for different bandwidths.

The Clio approach: **Clio** is a model compilation technique that allows us to execute large state-of-art models over resource-constrained IoT networks while gracefully degrading performance in the face of uncertain wireless bandwidth. At the core of **Clio** is a joint IoT-cloud optimization technique for progressively transmitting intermediate results from a partitioned model in-order to deal with bandwidth variations. In addition to introducing a new technique for enhancing the operation of partitioned models, **Clio** also presents an integrated solution that combines various pieces of the puzzle — model compression, model partitioning and progressive transmission — to create an integrated system for partitioned execution under dynamic bandwidth settings. Thus, our work is holistic and leverages multiple methods to provide the best overall performance. We describe our proposed solution in more detail in the following section.

3 CLIO DESIGN

We start with an overview of the **Clio** run-time and then describe the core techniques involved.

3.1 Overview of Clio run-time

The **Clio** runtime operates differently in response to short-term and long-term bandwidth dynamics. There are two cases where rapid adaptation to bandwidth is required. The first is when channel conditions are unknown when a node wakes up after operating in deep sleep (i.e. during duty-cycled operation). The second is when link bandwidth varies unpredictably due to mobility. **Clio** adapts on-the-fly to these conditions by operating in a progressive manner.

When bandwidth changes persist for longer periods, then **Clio** adapts the partition point so that it can operate at the best partition for the current conditions. For example, say the model is partitioned at layer 5 and progressive slicing can allow it to deal with bandwidth swings from 250 kbps to 1 Mbps. If the average bandwidth now drops to 100 kbps, then switching to a different partition point is needed in-order to further shrink the size of intermediate results and better cope with the lower available bandwidth.

3.2 Progressive slicing

Preliminaries: A deep learning model $\mathbf{M}$ can be represented by a directed acyclic computation graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consisting of a set of layers $\mathbf{V}$ and a set of directed edges $\mathbf{E}$. The **partition point** of the graph is a cut $\mathbf{C} \subset \mathbf{E}$ in the computation graph $\mathbf{G}$ that separate the model $\mathbf{M}$ into two parts.

To simplify this optimization problem, we restrict the cut to occur between two successive layers in the model. We assume a model with K layers and a cut occurs between $\mathbf{L}_k$ and $\mathbf{L}_{k+1}$, $k \in \{1, ..., K-1\}$. Thus, in the layered case, the choice of the cut $\mathbf{C} \in C$ determines the trade-off between end-to-end latency and total energy consumption. At this point, accuracy is unchanged as we are simply distributing a computation graph across the edge device and the cloud.

[2]We note that there are many approaches to model compression including weight quantization, weight sparsification, compact filters, and so on. The example we provide is one illustration of the general trend.

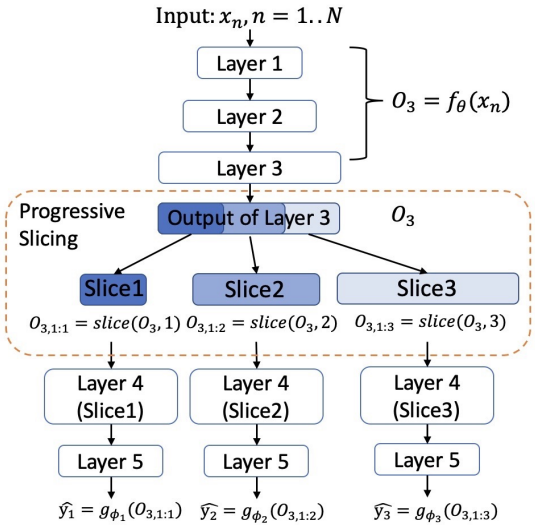


Figure 3: Example of progressive slicing.

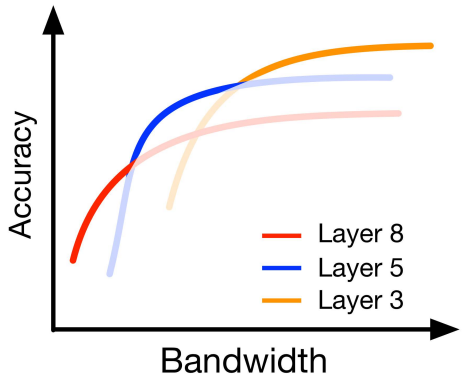


Figure 4: For each choice of partition point, progressive slicing offers a range of accuracy–bandwidth tradeoffs. The envelope (in bold) is the best operating regime for given bandwidth.

Learning Ordered Hidden Representations: **Clio** continuously adapts the size of the intermediate activations transmitted in a progressive manner to maximize best inference under the given bandwidth conditions. We achieve this via a novel training procedure that learns a single model that can continually adapt the size of the hidden representation that it transmits from layer $\mathbf{L}_k$ on the edge device to layer $\mathbf{L}_{k+1}$ on the cloud. We learn a separate model for the cloud for each hidden representation size. These models receive and decode the transmitted representation from layer $\mathbf{L}_k$ of each size, producing a final output.

Let $\mathbf{O}_k = f_\theta(\mathbf{x})$ be a function implementing the computation graph for a model M up to the output of layer k and assume that this layer includes W_k units. Now, define the slicing function $\mathbf{O}_{1:i} = \text{slice}(i, \mathbf{O})$ to map the input vector $\mathbf{O}$ to the sub-vector of $\mathbf{O}$ consisting of its first i channels, $\mathbf{O}_{1:i}$. Define $y = g_{\phi_i}(\mathbf{O}_{1:i,k})$ to be a function implementing the computation graph for a model M that receives as input only the first i hidden unit values from layer k and produces the final output of the network. Note that this function has a separate set of parameters for each value of i.

The model transmits as many hidden units as possible from layer k to layer $k+1$ across the wireless link given the latency constraint and available bandwidth. The final prediction output by the model is then given by: $y = g_{\phi_i}(\text{slice}(i, f_\theta(\mathbf{x})))$

The key to the success of this approach is to train the model using a loss function that takes into account a distribution over available bandwidths and thus over the representation sizes that can be transmitted. Let π_i be the probability that there is sufficient bandwidth available to send $\text{slice}(i, f_\theta(\mathbf{x}))$. Given a data set $\mathcal{D} = \{(\mathbf{x}_n, y_n), n = 1 : N\}$ and a prediction loss function $\ell(y, y')$, we define the loss based on the intermediate representation $\text{slice}(i, f_\theta(\mathbf{x}))$ by $\mathcal{L}_i(\mathcal{D}; \theta, \phi_i)$. The overall loss that we optimize is the expected value of $\mathcal{L}_i(\mathcal{D}; \theta, \phi_i)$ where the expectation is taken over the probability π_i that we can send the representation $\text{slice}(i, f_\theta(\mathbf{x}))$. The model thus learns parameters that result in an ordered hidden representation with the property that it minimizes expected error under the progressive slicing operation for the given distribution over representation sizes π. The details are provided below.

$$\theta^*, \phi^*_{1:W_k} = \arg\min_{\theta, \phi_{1:W_k}} \mathbb{E}_\pi \left[\mathcal{L}_i(\mathcal{D}; \theta, \phi_i)\right] \quad (1)$$

$$\mathcal{L}_i(\mathcal{D}; \theta, \phi_i) = \sum_{n=1}^{N} \ell(y_n, g_{\phi_i}(\text{slice}(i, f_\theta(\mathbf{x}_n)))) \quad (2)$$

Example: We illustrate progressive slicing with the example shown in Figure 3. Here, the IoT device executes the initial layers until layer 3, and the rest of the pipeline is executed at the cloud. In this example, the output of layer 3 is partitioned into three slices, and depending on the bandwidth, a different number of slices are transmitted to the cloud. There are three corresponding versions of layers on the cloud to handle the three possible slices as input. During inference time, IoT devices will execute layers $1-3$ locally and pass the intermediate result of layer 3 to radio for transmission. Due to network dynamics, the radio will send as much as possible in the available time and the cloud will activate the corresponding path for the received progressively sliced intermediate results.

3.3 Adaptive partitioning

So far, we have assumed that we know the best partition point. But the choice of partition point impacts performance since changing it affects both the latency for computation, as well as the size of the activations. While progressive slicing does provide the ability to deal with variations in the data transmission size, a single partition point may not provide sufficient dynamic range across bandwidth conditions.

Figure 4 illustrates this interplay between partitioning and progressive transmission with an example. For each partition point, progressive transmission is a curve that offers different accuracy - bandwidth tradeoffs. Thus, the best operating point given some specific bandwidth is the ⟨partition, slice⟩ tuple that has the highest accuracy at that bandwidth.

Thus, we build a performance profile similar to that shown in Figure 4 i.e. a set of performance curves offered by different progressive slices across different partition points. These profiles can be calculated using platform-level benchmarking as we do on the GAP8 device in §4.2. Once we have such a profile, we can select the optimal partition point and progressive slice for a given bandwidth. We note that building such a profile may be expensive in terms of training time due to the number of possible partition points and progressive slices; we discuss how to optimize training time in §3.4.

Incorporating model compression: Augmenting progressive slicing with model compression can improve performance, particularly for computation of the IoT-portion of the model which is slower due to resource constraints at the IoT node. There are several approaches used for model compression onto embedded devices [10] including weight quantization, weight sparsification, compact filters and others. We apply these model compression methods at different stages of our pipeline — pruning happens prior to progressive slicing whereas weight quantization and sparsification happens after the progressively sliced model has been trained. This allows us to easily integrate progressive slicing with state-of-the-art model compression techniques (e.g. our implementation integrates with AutoML [17] and Meta-pruning [33]).

3.4 Reducing training time

So far, we assume that we train across all possible slices of the intermediate layer (for progressive transmission) and across all possible layers as potential partition points. Clearly, this can incur significant training time. In this section, we describe several optimizations to reduce training time.

Let $C(\mathbf{L_{1:k}})$ be the computation cost of the model $\mathbf{M}$ of layers $\mathbf{L_{1:k}}$ and k be the partition points. For the training of progressive slicing, the early layers $\mathbf{L_{1:k}}$ will be shared (since they are unaffected by slicing), and the set of layers $\mathbf{L_{k+1:K}}$ will be different for each slice. Thus, the computation cost of the model will be $C(\mathbf{M}) = C(\mathbf{L_{1:k}}) + \#\text{slices} \times C(\mathbf{L_{k+1:K}})$. Therefore, training cost is affected by two parameters — the number of slices and the number of partition points (i.e. k). We now look at how to optimize these two aspects.

Reducing training time for progressive slicing: There are two optimization that we use in our training to greatly reduce training complexity for training across progressive slices. First, we limit the number of different hidden representation sizes that are considered by setting some values of π_i to 0. For example, in our work we consider only sending a number of hidden units that is a multiple or power of a given integer. This approach can be used to restrict the number of different parameter sets required in the cloud to a more manageable number for large models. Second, we tie the parameters starting at layer $k+2$ so that we are only expanding the required parameters at the first layer on the cloud. For example, in Figure 3 we see that the cloud uses separate pipelines depending on which slice is transmitted. Instead, the cloud can only have different versions of Layer 4 to deal with the different number of intermediate activations transmitted and use the same Layer 5 across all these Layer 4 options. Our implementation uses both of these optimizations to accelerate training.

Reducing potential partition points: To reduce training across partition points, we use a bandwidth coverage heuristic to choose a few promising partition points to reduce training time. For each layer where we can partition the model, progressive slicing can provide some tolerance to bandwidth swings by changing how many slices are transmitted. So, we use a greedy algorithm to select only layers that can extend the bandwidth range to which the model can adapt. We now describe this process in more detail.

In-order to estimate how much bandwidth range can be supported via progressive slicing on a layer, we first need to know how much time is available for communication. Let $C_{iot}(\cdot)$ and $C_{cloud}(\cdot)$ denote the computation latency at the IoT device and cloud respectively. These functions can be estimated for each layer through one-time offline profiling.

For a model partitioned at layer K, $\mathbf{L}_{1:k}$ are executed at the IoT device and $\mathbf{L}_{k+1:K}$ at the cloud. The total latency for computation, ϵ_k^C, is $C_{iot}(\mathbf{L}_{1:k}) + C_{cloud}(\mathbf{L}_{k+1:K})$; total energy for computation will be ϵ_k^C, is $C_{iot}(\mathbf{L}_{1:k})$. Given an end-to-end latency target or total energy consumption of ϵ_{target}, we can now only use $\epsilon_{target} - \epsilon_k^C$ for communication to transmit intermediate results. If $\mathbf{O}_k$ denotes the different sizes of the progressively sliced intermediate results from layer k and $R(\cdot)$ denotes the function that maps the transmission sizes back into the required bandwidths based on target latency, we can calculate the bandwidth range $\mathbf{B}_k$ that progressive slicing at layer k can dynamically adjust to as:

$$\mathbf{B}_k = R(\mathbf{O}_k, \epsilon_{target} - \epsilon_k^C) \tag{3}$$

Given this range, we just select the most promising partitions as the minimum set of intervals that cover the bandwidth range of interest (e.g. from 50kbps to 2Mbps for Bluetooth). This is a simple interval coverage problem that can be solved using a greedy algorithm [12].

4 IMPLEMENTATION

We instantiate **Clio** for several state-of-the-art neural network models and IoT accelerators and processors, and evaluate on different datasets. We describe the implementation details below.

4.1 Compiling Clio to Models

To prove the generalization of our approach, we apply **Clio** to several different state-of-the-art neural network models including **MobileNetV2** [38], **VGG** [42] and **ResNet** [16]. We partition after any layer or blocks (residual blocks in **ResNet** or bottleneck structures in **MobileNetV2**) and do not partition within each block structure.

Skip connections: MobileNetV2 uses skip connections to connect the beginning and end of a convolutional block; by doing so, the network has the opportunity of accessing earlier activations that were not modified in the convolutional block. Skip connections usually exist when the input and output have the same size. Since **Clio** progressively slices the intermediate results into different sizes during training, we change the skip connection from an addition operation to a 1×1 convolutional operation so that we can deal with different sizes of input and output.

Model compression: As mentioned in §3.4, **Clio** can be incorporated with state-of-the-art compression techniques. In our implementation, we apply AutoML[17] and Meta Pruning [33] to MobileNetV2. We also use width multiplier as a baseline to compress the MobileNetV2 model.

Reducing training time: In order to reduce training time when compiling **Clio** to MobileNetV2, we choose every second channel number for early partition points and every fourth channel number for later partition points for progressive slicing. For example, when we use a target latency of 300ms in our implementation, partition points after layer 3, 5 or 8 of MobileNetV2 are chosen based on our heuristic because they can extend our bandwidth range to 195 kbps, 175 kbps and 98 kbps respectively.

4.2 Compiling Models to IoT Devices

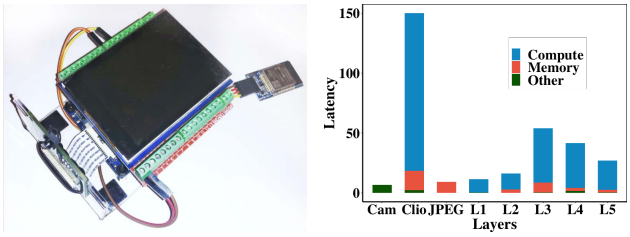
Clio compiles models into two sub-models — a sub-model that is locally executed on IoT devices and a sub-model that is remotely executed on cloud servers. We look at two IoT processors. The **GAP 8** is an ultra-low power neural accelerator that has the Hardware Convolutional Engine (HWCE) that can accelerate the convolutional operations [51]. The **STM32F7** is a popular ARM7-based IoT processor without a convolutional accelerator [6]. Our evaluation of MobileNetV2 on the GAP8 accelerator is based on a full end-to-end implementation (described in more detail below). For the ARM7 processor, we estimate the computational cost based on our implementation on the GAP8 but using ARM7 frequency and Risc-V cycles per instructions. In the ARM7, operations such as multiplication, addition and comparison are executed in parallel in the MCU.

GAP8 implementation: The GAP8 platform only has basic support for shallow models and wireless communication, and we were using both larger models as well as integration of IoT and cloud models. We implemented the necessary drivers and optimized them to achieve continuous operation over BLE. We also wrote software to take the model from PyTorch/TFLite and split it/load it onto the GAP8, which dealt with changes in format and optimized it to make it fit in memory.

We use TFLite [53] files for model deployment on GAP8. When converting a model into a TFLite file, some of the operations such as batch normalization are fused for deployment. As GAP8 only supports fixed-point arithmetic operations, all the weights and inputs are transformed into 8-bit Q-format fixed-point numbers during quantization. Meanwhile, the intermediate results are also stored and transmitted as 8-bit fixed point numbers.

There are two kinds of convolutions in MobileNetV2 – the first is basic convolution and the second is expanded convolution, which in-turn consists of three basic convolutions — expansion, depthwise and projection convolutions. Even though the GAP8 has a hardware convolution engine to speed up convolution operations, some operations are not supported due to the hardware design and are executed on the computing cores in parallel. Also, operations such as ReLU6 and padding are fused and executed on the computing cores in parallel.

On the hardware side, we integrated the GAP8 board with a HIMAX camera, LCD screen, and a radio shield which connects via UART interface (shown in Figure 5). The integrated board captures an image from camera, shows it on the LCD screen, executes the early layers of the model and then transmits the intermediate results via radios.



(a) Integrated GAP8 node (b) Latency Breakdown

Figure 5: (a) Integrated GAP8 node with Greyscale HIMAX HM01B0 camera, BLE shield, and LCD display, (b) latency specs based on empirical measurement.

Cloud Execution: Execution at the cloud is considerably faster than at the IoT device. For our benchmarks, we use a Xeon Silver 4116 2.10 GHz with 32 GB DDR4 2400 MHz and GTX 1080 Ti GPU.

Demo of real-time operation: We have integrated the above pieces to support continuous live operation. We made a full video demo of our system (see code[2]) — to make the demo, we had to re-train and optimize the model for the greyscale format coming from the GAP8 board's HIMAX camera which was different from the ImageNet/CIFAR-10 data that we used for our results.

Benchmarking MobileNetV2 on GAP8: We profiled the GAP8 to obtain latency and energy benchmarks for different computational blocks.

For latency profiling, we use the hardware counters to evaluate total cycles for computation, memory copying and other operations. We transform the total cycles to latency by dividing by the GAP8 clock frequency setting. Figure 5 shows the breakdown of latency consumed by the execution of our GAP8 implementation of the MobileNetV2 early layers. Based on the latency breakdown, we estimate the equivalent compute ability of the GAP8 when executing MobileNetV2 to be approximately 525 MMAC/s.

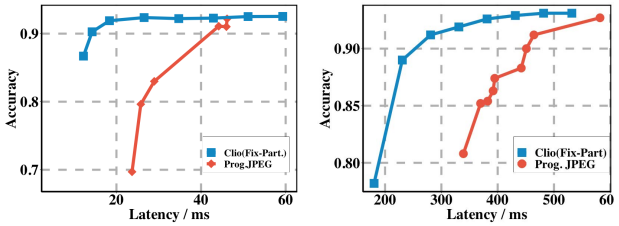
For wireless, we found that the maximum speed at which we could transfer data from the GAP8 to the BLE shield is 250kbps (due to UART speed limitations). We use these numbers in our evaluation when evaluating latency and energy.

5 EVALUATION

We start by describing the datasets we use and schemes we compare against before evaluating **Clio** against these methods.

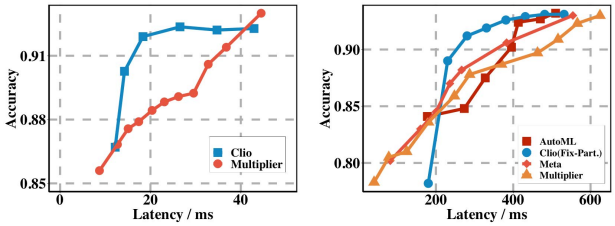
5.1 Datasets

The two main datasets that we will use for evaluation are based on **CIFAR-10** and **ImageNet** — these datasets have different image sizes and demonstrate the performance of our methods under different workloads. The **CIFAR-10** [25] dataset consists of 60000 32×32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The **ImageNet** [14] dataset has the 224×224 color images. The original **ImageNet**



(a) CIFAR-10 (b) ImageNet-20

Figure 6: Prog. Slicing vs Prog. JPEG for CIFAR-10 and ImageNet-20 datasets. Latency corresponds to GAP8 and BLE radio (numbers in §4.2). Prog. Slicing has significantly better performance than Prog. JPEG for both CIFAR10 and ImageNet-20 datasets.



(a) CIFAR-10 (b) ImageNet-20

Figure 7: Clio vs Model Compression. Latency corresponds to GAP8 and BLE radio @ 250kbps. On CIFAR10, only width multipliers are effective and Clio is better than this method. On ImageNet-20, Clio is better than AutoML, meta-pruning and width multipliers except at low latencies where there is insufficient time for communication.

dataset consists of 1000 classes. In practice, however, IoT devices are typically focused on specific tasks. Rather than classify over 1000 classes, so we create a smaller dataset, ImageNet-20, with a random set of 20 classes and use this for our evaluation. We have 26000 images for training in 20 classes with 1300 images per class in training; 1000 images for evaluation.

5.2 Comparisons

We compare against several methods in this evaluation.

Data Compression: In terms of **data compression**, we compare **Clio** against transmission of JPEG-compressed raw data to the cloud for remote processing. We use a state-of-art JPEG-based encoder, DeepN-JPEG [32], which is specifically engineering for deep learning workloads and therefore performs better than traditional JPEG for deep learning tasks.

We evaluate against two versions of JPEG: a) *Baseline JPEG* where the image is compressed to a desired quality and the coefficients are sequentially transmitted, and b) *Progressive JPEG* where the compressed coefficients are interlaced in different passes of progressively higher detail.

Model Compression: In terms of **model compression**, we compare against several state-of-art local compression options: meta-pruning [33], adapting the width-multiplier and AutoML [17].

Partitioning: Finally, we also compare against state-of-art model partitioning methods including Neurosurgeon [21], JALAD [29], and 2-step Pruning [41].

5.3 Prog. Slicing versus Prog. JPEG

We now compare Progressive Slicing in **Clio** for a fixed choice of partition point (layer 5) against Progressive JPEG that uses the state-of-art DeepN-JPEG encoder [32]. Both operate in an interlaced manner and can stop at any time.

We first look at the performance benefits when compared with remote computation on JPEG-compressed data. Figure 6 shows latency results for two datasets (CIFAR-10 and ImageNet-20) and a BLE radio operating at 250kbps. We see that **Clio** with fixed partition point clearly outperforms Progressive JPEG, particularly at low latencies. Results are almost identical when the metric is energy instead of latency, hence we do not show these results.

5.4 Clio versus Model Compression

We now evaluate the benefits of **Clio** over model compression approaches that seek to squeeze the entire MobileNetV2 model into the resource constraints of the GAP8.

Figure 7 shows the results. For small image sizes in CIFAR-10, **Clio** is significantly better than the width multiplier. The results for ImageNet-20 are more instructive. At low latency, we see that local compression methods generally perform better as expected since **Clio** can only transmit a very small amount of intermediate results to the cloud, which diminishes accuracy. Once latency increases above about 220ms, **Clio** performs better than all local model compression methods. Finally, when the latency is sufficiently high, the techniques start to converge since there is sufficient compute and sufficient data transfer for both local and remote processing to work well.

We note that these results are only based on progressive slicing and do not adapt the partition point. Augmenting progressive slicing with adaptive partitioning further improves the benefits of **Clio** over model compression (we defer this discussion to §5.6)

5.5 Comparing Local, Remote, and Clio

So far, we have separately evaluated **Clio** against model compression and data compression. We now look more holistically at these three methods to try to understand the operating regimes where each of them works best. The design space, however, is rather large since different factors affect the different methods — local execution depends on compute/memory resources and compute efficiency; remote execution depends on available bandwidth and radio efficiency; and **Clio** depends on all of the above factors. Rather than looking at each design parameter separately, we illustrate the general trends and tradeoffs by taking a few example architectures and the MobileNetV2 model.

We now compare end-to-end latency across a few exemplar architectures corresponding to different processors (GAP8 – 50mW @ 150MHz and STM32F7 ARM7 – 60mW @ 200MHz), radio power/bandwidth combinations (LoRa – 50kbps @ 60mW, ZigBee – 200kbps @ 60mW, BLE – 250kbps @ 15mW, WiFiax – 1 Mbps @ 100 mW), and image sizes (CIFAR-10 vs ImageNet-20). Since we cannot show the

entire envelope for each setting, we look at the performance ratio between each method and **Clio** corresponding to 90% accuracy (trends are similar for other choices of accuracy). For each method, we evaluate the ratio of its performance against **Clio** — so a latency or energy ratio greater than one means that the method has higher latency/energy compared to **Clio**.

Figure 8 shows the results for CIFAR-10 and ImageNet-20 — the caption describes how to interpret the table and color code. We see that as we go from left to right (low to high bandwidth), performance gains of **Clio** over remote processing diminishes whereas performance gains over local processing improves. As we go from a general-purpose ARM7 processor to a specialized GAP8 accelerator, local processing improves in performance. We see a significant band of configurations where **Clio** has better performance than local and remote processing. The results are intuitive and generally follows the trend of available compute and bandwith resources — when compute is more plentiful than bandwidth, local compute is better; when bandwidth is more than local compute resources, remote computation is better and **Clio** when there is roughly equivalent compute and bandwidth resources hence using both together provides advantages.

5.6 Benefits of adapting the partition point

So far, our results have focused exclusively on progressive slicing. We now turn to optimizing the selection of the partition point together with progressive slicing. We consider two versions of **Clio** — a fixed partition version as before (layer 5) and an adaptive version that uses the average bandwidth over the last minute to determine what partition point to use (as described in §3.3). We also show the performance of two versions of JPEG — Progressive JPEG which makes no bandwidth assumptions and Baseline JPEG which compresses based on the recent bandwidth history.

We look at performance across a range of dynamic bandwidth scenarios on synthetic and real traces to allow us to understand the performance of **Clio** in a dynamic environment. We focus on ImageNet-20 images due to their larger size and consider the case where images need to be processed within 300ms.

Synthetic Trace-driven Evaluation: To demonstrate the ability of **Clio** to adapt to dynamics, we use synthetic bandwidth traces where we systematically vary the amount of dynamics. The mechanism for constructing traces is based on the procedure in [34] and uses a Markovian model in which each state represented an average bandwidth in the selected range. State transitions were performed at a 1 second granularity and followed a geometric distribution. Each bandwidth value was then drawn from a Gaussian distribution centered around the average bandwidth for the current state, with variance uniformly distributed in $[0.05, \sigma_{MAX}]$. We vary σ_{MAX} to control the amount of variance in the trace and plot the performance of different schemes to show how they adapt to dynamics. Our synthetic dataset covers typical wireless network conditions for low-power radios, with average bandwidth ranging from 50 Kbps to 1 Mbps.

Figure 9 shows how gracefully different methods degrade in accuracy as a result of divergence between the real bandwidth and estimated bandwidth. The larger this gap is, the more likely it is that the partition point or compression ratio is improperly chosen.

We see that **Clio** with adaptive partitioning has the least degradation in accuracy as the bandwidth variability increases. The accuracy degradation for Baseline JPEG is 2–3× worse than that for **Clio** with adaptive partitioning. In fact, even **Clio** with a fixed partition point performs marginally better than Baseline JPEG albeit. Progressive JPEG performs significantly worse than the **Clio**-based methods.

Real-world Trace-driven Evaluation: We now evaluate **Clio** across several real-world bandwidth traces. Specifically, we look at HSDPA mobility traces which were collected in different mobile environments such as buses, trains, and ferries [37].

Figure 10 shows the results. We note that when the bandwidth is less than the minimum bandwidth required for the technique used (e.g. first scan/slice) to generate a prediction result, then we record the corresponding accuracy as zero. We see that Progressive Slicing is better than Progressive JPEG across all traces; in fact, our method is also comparable to Baseline JPEG even though the method makes no assumptions about available bandwidth. **Clio** augmented with partition point adaptation improves performance and is as good or better than JPEG-based methods.

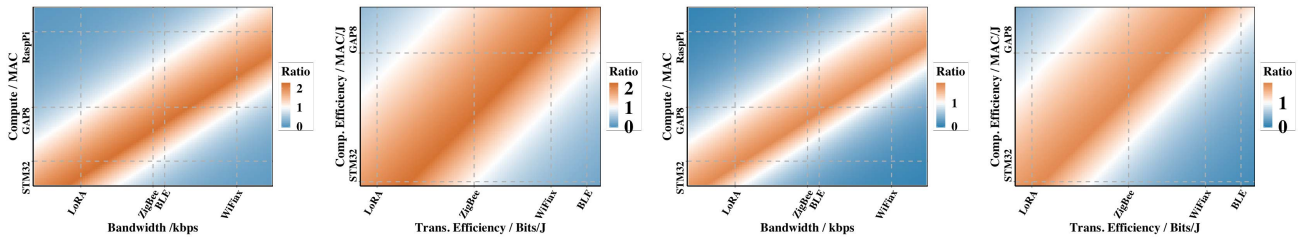
We expand on the above results in two ways. First, we take all the instances from the trace where there the predicted bandwidth does not match the actual bandwidth, and look at what effect that has on accuracy. Figure 11(a) shows the loss in accuracy for different methods across different gaps between predicted and actual bandwidths. We see that **Clio** with adaptive partitioning can achieve the best overall accuracy and minimal accuracy degradation even when there is a substantial gap between real and estimation bandwidths result from network dynamics.

Second, we illustrate the difference between adaptive partitioning versus fixed partitioning using a time-series snippet from the traces with bandwidth dynamics. We see that adaptive partitioning adapts to larger bandwidth changes effectively to minimize the loss in performance due to dynamics.

Performance under known bandwidth: In the trace-driven evaluation, we had to predict bandwidth based on past measurements; now, we take a more favorable scenario for Baseline JPEG where bandwidth is known and quality of compressed JPEG is optimized for the given bandwidth. These results are for the ImageNet-20 dataset.

Figure 12 shows the results. We see that choosing the optimal partition point improves the classification performance particularly at lower latencies. We see that Baseline JPEG compression is significantly better than Progressive JPEG. The graph also shows interesting tradeoffs between Baseline JPEG versus **Clio**. We see that for very low latency, Baseline JPEG is superior since JPEG computation has lower latency than that incurred by **Clio** to compute the initial layers. As available latency increases above a minimum threshold to execute a few layers locally, **Clio** starts to perform better. Both the progressive-only and optimal-partitioning + progressive-based versions of **Clio** perform better than JPEG-based options. Once the latency is long enough to transmit sufficient data, all approaches converge in performance.

Compute overhead of switching partition: Changing the partition point incurs some additional overhead since the IoT-portion of the model needs to be changed. We assume that the different



(a) CIFAR10: Latency (b) CIFAR10: Energy (c) ImageNet-20: Latency (d) ImageNet-20: Energy

Figure 8: Heatmaps representing the regimes where local, remote and partitioned computation offer benefits. The left two graphs correspond to CIFAR-10 and the right two correspond to ImageNet-20. The band in the middle of each graph corresponds to the regime where Clio is the best approach; to the left of the band is the region where remote computation (on jpeg-compressed data) is better and the right of the band is the region where local computation (via a compressed model) is more efficient.

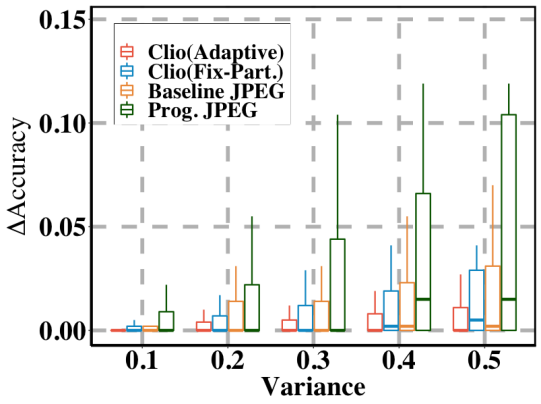


Figure 9: We use synthetic traces and gradually increase variance in bandwidth to show robustness of different methods. Clio with adaptive partitioning is most robust to bandwidth variations and degrades more gracefully than Baseline JPEG.

models are stored on the GAP8 flash and the cost only involves switching between the models. On the GAP8, the weights of the model are stored as files in HyperFlash and need be copied to cache or HyperRAM before the model can be run. Therefore, when a partition point change is triggered, we need to first free the cache and HyperRAM for the current model and allocate new space for the new model. We empirically evaluated that this process only takes around 20ms on the GAP8, which is a small overhead given relatively infrequent switching between partition points (for example, the 20ms overhead is barely visible in Figure 11(b)).

5.7 Evaluation over GAP8 Hardware

We now look at an end-to-end evaluation over the actual GAP8 platform and BLE radio. One key difference between the results so far and our real experiments over the GAP8 is the effect of radio duty-cycling. So far, we have assumed that the radio can be instantly connected to the cloud whenever needed. This is, of course, unrealistic for duty-cycled radios — for example, a BLE sender and receiver wakeup at periodic intervals to see if there is new data to transmit and this interval determines how quickly data transmission can begin. The choice of wakeup interval influences duty-cycling efficiency — if the interval is high, then the radio consumes less power (e.g. BLE consumes only 45μW if connection latency is 1 second), and if the interval is low, frequent wakeups increase power consumption (e.g. BLE consumes 460μW at 40ms connection latency) [46].

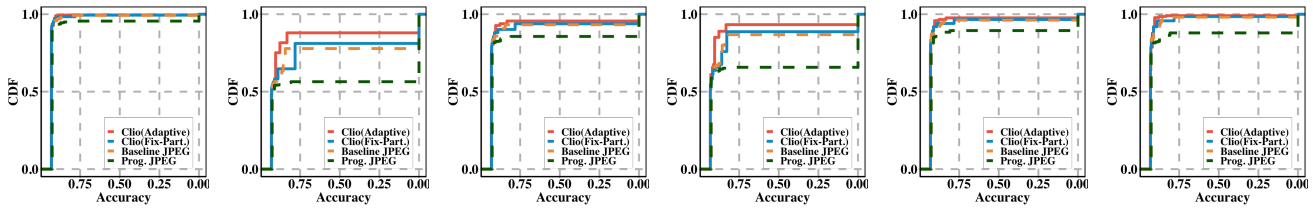
Figure 13 shows the results when we use BLE 4.0 and BLE 4.2 as the radio. The difference between these is that BLE 4.0 can only transmit data via notification packets whose maximum size is 20 bytes while BLE 4.2 enlarges the maximum size to 251 bytes. We see that the end-to-end latency of **Clio** increases very little even as connection interval increases significantly from 7.5ms to 40ms whereas the end-to-end latency for JPEG increases steeply in a step-wise manner (corresponding to each additional layer being transmitted). One advantage for **Clio** in the presence of connection latency is that since **Clio** processes several initial layers before transmitting to the cloud, it is able to mask a significant fraction of the end-to-end latency.

Thus, an important advantage of **Clio** in the context of deployments that use duty-cycled radios is that **Clio** is effective at masking the wake-up delays by locally processing the data until the connection becomes available. In contrast, the computation time of JPEG is small and does not benefit as much from the wake-up delay.

5.8 Other models and methods

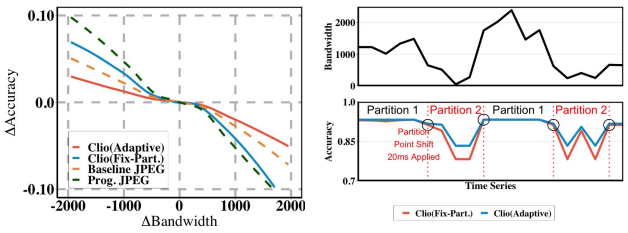
We now look at how **Clio** performs for models other than MobileNetV2 and compare against several other partitioning methods.

Compatibility with other models and techniques: So far, we have looked only at MobileNetV2 but **Clio** can also be used with other deep learning models. Figure 14 compares Progressive Slicing, Progressive JPEG and Model Compression for another popular model, ResNet [16]. Since ResNet has higher compute requirements, the bandwidth point at which it becomes better than remote computation is lower than for MobileNetV2. We generally see that ResNet is most effective at bandwidth lower than 200kps, so we show results for ResNet when the wireless interface is a LoRa radio operating at 50kbps. For this radio, we see that the latency envelope from ResNet is better than local and remote computation. We see



(a) Bus (b) Car (c) Ferry - CLIO vs. JPEG (d) Metro (e) Train (f) Tram

Figure 10: CDF of performance of Clio and JPEG for different mobility traces.



(a) Effect of estimation error (b) Time Series

Figure 11: (a) Even when there is a large gap between estimated and actual bandwidth, Clio with partition point adaptation performs best. (b) Time series showing bandwidth changes and partition point adaptation to minimize loss in accuracy

	Bandwidth →			
	LoRA	ZigBee	BLE	WiFi
STM32F7	2.3x|3.6x	1.5x|1.8x	1.3x|1.7x	1.1x|1.2x
GAP 8	2.8x|6.3x	2.2x|3.4x	2.0x|3.0x	1.3x|1.5x

Table 1: Comparison against Neurosurgeon on end-to-end latency on ImageNet-20.

	Bandwidth →			
	LoRA	ZigBee	BLE	WiFi
STM32F7	1.65x	1.39x	1.32x	1.08x
GAP 8	1.64x	1.64x	1.63x	1.27x

Table 2: Comparison against 2-step pruning on end-to-end latency on VGG/CIFAR-10.

similar trends with VGG-19 which is also a large model but do not show results due to space constraints.

Another technique to reduce communication is to use early exit which allows the model to stop execution at the IoT device if the prediction result can be predicted with high confidence [44, 45]. This technique is compatible with **Clio** and early exit can be folded into **Clio** pipeline. However, we found that early exit at the IoT device has poor accuracy (less than 70%) for models like MobileNetV2 given the IoT device constraints so we did not incorporate this into **Clio**.

Comparison against JALAD: Figure 15 shows results of our comparison against JALAD[29], a technique for reducing the size of intermediate results by quantizing with fewer bits. **Clio** performs better across the entire range, particularly at low compression ratios. This is because **Clio** can leverage the sparsity of the intermediate results while JALAD only leverages the sparsity in the fixed-point representation. We note that the quantization optimizations in JALAD can also be incorporated with **Clio** to improve results. For example, performance is unaffected by 6-bit quantization as opposed to 8-bit quantization and this can help improve **Clio**'s results.

Comparison against NeuroSurgeon: Table 1 compares **Clio** against Neurosurgeon across different radios and presents two numbers — the first is for similar accuracy as Neurosurgeon (i.e. less than 0.5% accuracy loss) and the second is for lower accuracy (roughly 3% accuracy loss). We see that even at almost identical accuracy, **Clio** performs better since it uses progressive slicing and adaptive partitioning to improve performance. **Clio** provides significantly better performance if the application can tolerate additional accuracy loss.

Comparison against 2-step Pruning: We also compare **Clio** with a two-step pruning method proposed in [41]. This method prunes the original model in two passes and partitions the pruned model into two parts (edge and cloud). In the first pruning pass, all layers of the model are pruned by using state-of-art pruning methods; in the second pass, each layer is pruned one-by-one while keeping other layers intact. Finally, the partition point is determined such that end-to-end latency requirements are met.

The 2-step pruning method is evaluated using VGGNet and CIFAR-10, so we compare with **Clio** for the same model/dataset. We compare end-to-end latency of these methods for comparable accuracy. Table 2 shows that **Clio** performs better than 2-step pruning across the different bandwidth settings.

Overall, we see that **Clio** performs better than alternate partitioning methods since the progressive slicing method manipulates the intermediate feature maps, which is critical for dealing with highly constrained network conditions.

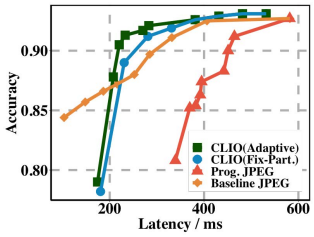


Figure 12: Adaptive Partitioning Benefits

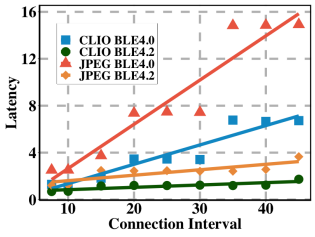


Figure 13: Experimental Results on GAP8

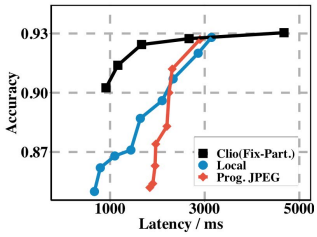


Figure 14: Clio vs JPEG for ResNet18

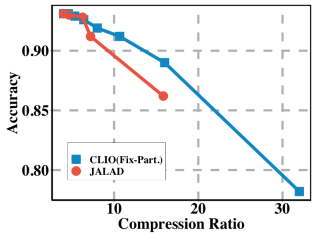


Figure 15: Comparison between Clio and JALAD

6 RELATED WORK

Related efforts can be largely classified into three categories — deep learning for resource-constrained devices, cloud offload and partitioned execution.

Model compilation for single device: There has been much recent interest on optimizing deep learning models for embedded MCUs [8, 27, 28, 31, 56]. While most of this work assumes less stringent constraints than required by IoT devices, some specifically target such devices [27, 31, 56]. Model compression is a more mature field for smartphone-class devices — companies like Google and Amazon offer model compression tools to make it easy for developers to optimize deep learning models for mobile devices [47, 48]. Our work is complementary to these efforts and enables model compilation techniques to also leverage low-power wireless radios and compile large models to distributed IoT-Cloud networks.

Cloud offload: There has been more than a decade of work on optimizing cloud offload and this is one of the most common ways of deploying IoT systems [7, 11, 13, 19, 23, 30, 39]. Generally, cloud offload methods are agnostic to the specific computation performed at the edge (IoT) device versus the edge cloud. They primarily deal with variations in network delay and bandwidth at the network layer. In contrast, we develop an end-to-end method for compiling machine learning models such that they can seamlessly adapt to fluctuations in network conditions.

Partitioning analytics: Several recent efforts have looked at partitioning deep learning models across the cloud and edge (e.g. Neurosurgeon [21]). There has been some work that also extends the idea of partitioning — [29] quantizes and compresses intermediate results, [24] also lossily compresses intermediate results, and [40] describes additional model pruning of the local IoT model to reduce bandwidth. We show that progressive slices can provide benefits over compression of intermediate results and over model compression techniques.

7 CONCLUSIONS

In conclusion, we present a new approach to model compilation for low-power IoT where we automatically split a deep learning model between the IoT device and the cloud to optimize latency and energy while also being resilient to bandwidth dynamics. Our work is a departure from prior work in that **Clio** operates in a progressive manner and simultaneously reasons about model compression (for local computation) and data compression (for cloud computation). Our approach is end-to-end and optimizes overall inference performance while taking into consideration resource limitations such as bandwidth, energy, and computation ability. We believe that our approach is generalizable and paves the way for leveraging both low-power accelerators and low-power radios. It can be used in a host of IoT and sensor network deployments.

Our work also opens up several research directions. While we focused on images, we suspect similar methods will work for complex audio processing models, for example, methods to enable speech processing in noisy environments [4]. Also, we are currently working on extending **Clio** to models for video analytics – these models are quite different since they learn temporal dependencies via LSTMS, 3D convolutions, or some combination of CNNs and optical flow tracking [20, 22, 43, 54, 57]. Our expectation is that as the complexity of these models increases, partitioning between edge and cloud becomes more essential and model compilation methods such as **Clio** will be even more important.

ACKNOWLEDGMENTS

The research reported in this paper was sponsored in part by the CCDC Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196 (ARL IoBT CRA) and by National Science Foundation under Grant No. 1719386 and No. 1815347. 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 the ARL, NSF or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] cc2531 zigbee chip. http://www.ti.com/lit/ds/symlink/cc2531.pdf.
[2] Github repo. https://github.com/jinhuang01/CLIO.
[3] Long range, low power rf transceiver 860-1000 mhz with lora technology. https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1272.
[4] Nsf: Hearables challenge. https://www.nasa.gov/feature/nsf-hearables-challenge.
[5] St spbt2632c2a bluetooth module. http://www.st.com/web/en/resource/technical/document/datasheet/DM00048919.pdf.
[6] Stm32f7: A very high-performance mcus with arm cortex-m7 core. https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html.
[7] A. Ashok, P. Steenkiste, and F. Bai. Enabling vehicular applications using cloud services through adaptive computation offloading. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, pages 1–7. ACM, 2015.
[8] S. Bhattacharya and N. D. Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 176–189. ACM, 2016.
[9] Bosch. *BMI160: Ultra Low Power Inertial Measurement Unit.*

[10] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[13] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *In Proceedings of ACM MobiSys*, 2010.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[15] Y. Han, X. Wang, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen. Convergence of edge computing and deep learning: A comprehensive survey. *CoRR*, abs/1907.08349, 2019.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[18] Himax. *HM01B0: Ultra Low Power Image Sensor*.

[19] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale. Weardrive: Fast and energy-efficient storage for wearables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 613–625, Santa Clara, CA, July 2015. USENIX Association.

[20] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.

[21] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 615–629. ACM, 2017.

[22] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[23] E. C. Kevin Boos, David Chu. Flashback: Bringing immersive virtual reality to mobile devices through aggressive rendering memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, 2016.

[24] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay. Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms. In *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2018.

[25] A. Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[26] L. Lai and N. Suda. Enabling deep learning at the iot edge. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, 2018.

[27] L. Lai, N. Suda, and V. Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[28] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 283–294. ACM, 2015.

[29] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu. Jalad: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 671–678. IEEE, 2018.

[30] F. Liu, P. Shu, H. Jin, L. Ding, J. Yu, D. Niu, and B. Li. Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications. *Wireless Communications, IEEE*, 20(3):14–22, 2013.

[31] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. 2018.

[32] Z. Liu, T. Liu, W. Wen, L. Jiang, J. Xu, Y. Wang, and G. Quan. Deepn-jpeg: a deep neural network favorable jpeg-based image compression framework. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[33] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3296–3305, 2019.

[34] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210, 2017.

[35] Maxim Integrated. *Ultra-Low Power, Single-Channel Integrated Biopotential (ECG, R to R Detection) AFE*.

[36] Nordic Semiconductor. *NRF51822: Bluetooth low energy and 2.4GHz proprietary SoC*.

[37] H. Riiser, T. Endestad, P. Vigmostad, C. Griwodz, and P. Halvorsen. Video streaming using a location-based bandwidth-lookup service for bitrate planning. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 8(3):24, 2012.

[38] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. IEEE, 2018.

[39] M. Satyanarayanan. A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *GetMobile: Mobile Computing and Communications*, 18(4):19–23, 2015.

[40] W. Shi, Y. Hou, S. Zhou, Z. Niu, Y. Zhang, and L. Geng. Improving device-edge cooperative inference of deep learning via 2-step pruning. *arXiv preprint arXiv:1903.03472*, 2019.

[41] W. Shi, Y. Hou, S. Zhou, Z. Niu, Y. Zhang, and L. Geng. Improving device-edge cooperative inference of deep learning via 2-step pruning. *CoRR*, abs/1903.03472, 2019.

[42] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[43] L. Sun, K. Jia, D.-Y. Yeung, and B. E. Shi. Human action recognition using factorized spatio-temporal convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4597–4605, 2015.

[44] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[45] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339. IEEE, 2017.

[46] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino, and D. Formica. Performance evaluation of bluetooth low energy: a systematic review. *Sensors*, 17(12):2898, 2017.

[47] https://aws.amazon.com/sagemaker/neo/. Amazon SageMaker Neo: Train models once, run anywhere with up to 2x performance improvement.

[48] https://developers.google.com/ml-kit/. Google ML Kit: Bringing Google's ML expertise to mobile developers.

[49] https://en.wikipedia.org/wiki/IEEE_802.11ah. 802.11ah WiFi HaLow: WiFi for Internet of Things.

[50] https://etacompute.com/. Etacompute TENSAI: Machine Learning Platform with Ultra-Low-Power Consumption for Edge Devices.

[51] https://greenwaves-technologies.com/gap8-product/. GAP8: Ultra-low power, always-on processor for embedded artificial intelligence.

[52] https://www.syntiant.com/. SYNTIANT: Always-on machine learning solutions for battery-powered devices.

[53] https://www.tensorflow.org/lite. Tensor Flow Lite: Deploy machine learning models on mobile and IoT devices .

[54] G. Varol, I. Laptev, and C. Schmid. Long-term temporal convolutions for action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 40(6):1510–1517, 2018.

[55] Vesper. *VM1010: Wake-on-Sound Piezoelectric MEMS Microphone*.

[56] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291. ACM, 2018.

[57] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4694–4702, 2015.