Flamingo: A User-Centric System for Fast and Energy-Efficient DNN Training on Smartphones

Sanjay Sri Vallabh Singapuram
University of Michigan - Ann Arbor
singam@umich.edu

Chuheng Hu
John Hopkins University
chu29@jhu.edu

Fan Lai
University of Illinois
Urbana-Champaign
fanlai@illinois.edu

Chengsong Zhang
University of Illinois
Urbana-Champaign
cz81@illinois.edu

Mosharaf Chowdhury
University of Michigan - Ann Arbor
mosharaf@umich.edu

ABSTRACT

Training DNNs on a smartphone system-on-a-chip (SoC) without carefully considering its resource constraints leads to suboptimal training performance and significantly affects user experience. To this end, we present Flamingo, a system for smartphones that optimizes DNN training for time and energy under dynamic resource availability, by scaling parallelism and exploiting compute heterogeneity in real-time. As AI becomes a part of the mainstream smartphone experience, the need to train on-device becomes crucial to fine-tune predictive models while ensuring data privacy. Our experiments show that Flamingo achieves significant improvement in reducing time ($\tilde{1}2\times$) and energy ($\tilde{8}\times$) for on-device training, while nearly eliminating detrimental user experience. Extensive large-scale evaluations show that Flamingo can improve end-to-end training performance by 1.2–23.3× and energy efficiency by 1.6–7× over the state-of-the-art.

CCS CONCEPTS

• Computer systems organization → Real-time system architecture; **• Computing methodologies → Distributed artificial intelligence**.

KEYWORDS

Federated Learning, User Experience, Training Latency, Energy Efficiency

ACM Reference Format:
Sanjay Sri Vallabh Singapuram, Chuheng Hu, Fan Lai, Chengsong Zhang, and Mosharaf Chowdhury. 2023. Flamingo: A User-Centric System for Fast and Energy-Efficient DNN Training on Smartphones. In *4th International Workshop on Distributed Machine Learning (DistributedML '23), December 8, 2023, Paris, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3630048.3630183

1 INTRODUCTION

Model training and inference at the edge are becoming ubiquitous, allowing for better privacy [44], localized customization [39], low-latency prediction [39] etc. Google [29] and Meta [43] deploy federated learning (FL) across potentially millions of end-user devices to mitigate privacy concerns with user-sensitive data migration; Apple performs federated evaluation and tuning of automatic speech recognition models on mobile devices [53]. Furthermore, communication constraints like intermittent network connectivity and bandwidth limitations (e.g.driving data) also necessitate the capability to train models closer to the user [42].

Recent advances for on-device model execution focus on theoretical optimizations to ML models [40, 55, 60]. However, the execution engines used to demonstrate these systems are ill-suited for resource-constrained devices like smartphones. Due to the lack of easily-extensible mobile backends, today's on-device efforts often resort to either traditional in-cluster ML frameworks (e.g.PyTorch [18] or TensorFlow [27]) or operation-limited mobile engines (e.g.DL4J [5],TFLite [23]). The former prioritizes performance over resource usage, while the latter limits which models can train. Overall, existing on-device training solutions are either suboptimal in performance, detrimental to user experience, or limited in capability.

Unlike cloud or datacenter training devices (i.e., GPUs), smartphones are constrained in terms of the maximum electrical power and energy consumption. Modern smartphones use a system-on-a-chip (SoC) architecture with a small number of heterogeneous cores, each with different strengths and weaknesses. These constraints are exacerbated by dynamic resource availability in smartphones: the impact on user-facing applications must be minimal when end-users are actively using the device.

Therefore, efficiently performing training requires careful consideration of multiple constraints: while training on low-performance, low-power core(s) can meet energy and power constraints without interfering with user applications, this comes at the cost of time; in some cases, it may be energy-inefficient owing to the long training duration. Statically allocating cores to applications leads to resource under-utilization, and existing proposals to offload training to unused cores do not have practical implementations [45]. Further, running a computationally-intensive workload like DNN training can significantly degrade user experience due to resource contention. In short, we need a *practical, bespoke and adaptive system* for on-device DNN training on smartphone SoCs.

In this paper, we propose Flamingo, a real-time adaptive system to train DNN models on smartphones in real-world settings, while considering resource and temperature constraints without hurting user experience. Our key contributions are as follows:

- We propose a novel dynamic resource assignment algorithm that tightly couples the DNN model and the heterogeneity of SoCs, to both optimize resource usage *and* improve user-experience.
- We demonstrate Flamingo's efficacy on user-experience across a set of popular devices of varying performance capabilities with a real-world on-device benchmark.
- We also show Flamingo's efficacy in a large-scale Federated Learning setting, leading to faster convergence and energy savings.
- Flamingo can train *unmodifed* PyTorch models, allowing access to the full PyTorch operator set and obviating model conversion for deployment. This is particularly valuable when considering edge deployment of emerging use cases, such as generative AI applications.
- Flamingo is implemented in user-space and works across multiple Android platforms, eliminating platform-specific dependence and dangerous rooting procedures.

2 BACKGROUND & RELATED WORK

Federated Learning algorithms have made considerable progress to train the model on the edge. FedProx [49], Fed-ensemble [61] and FedYoGi [55] reinvent the vanilla model aggregation algorithm, FedAvg [52], to mitigate the data heterogeneity. Oort [47] orchestrates global-scale FL clients, and cherry-picks participants to improve the time-to-accuracy training performance, while other advances are reducing network traffics [57], enhancing client privacy via differential privacy [33, 67], personalizing models for different clients [37, 39], and benchmarking FL runtime using realistic FL workloads (e.g., FedScale [46] and Flower [28]).

Application Sandboxing on Android isolates applications from each other [2], preventing them from accessing system-information (e.g. CPU-load) and modifying system-parameters (e.g. CPU frequency). This makes it impossible for user-space apps to monitor the system and enable intelligent scheduling. "Rooting" the device could disable sandboxing, but incurs the risk of bricking the device and making the user data vulnerable to malicious applications [30].

On-device Training on mobile operating systems has been enabled with frameworks like Apple's CoreML [4] and Google's TFLite [23], with capabilities of offloading compute to the mobile GPU or a specialized Neural Processing Unit (NPU). Deeplearning4J [5] and PyTorch offer Java binaries to include with Android applications for on-device training, but are not space-optimized (up to 400 MB) and cannot offload training to the mobile GPU. MNN [1] addresses the first issue by eliminating dependencies with its lightweight footprint of a few megabytes. [45, 48, 65] propose solutions to optimize energy consumption by modifying the clock frequency of the processor. These proposals require platform-specific rooting

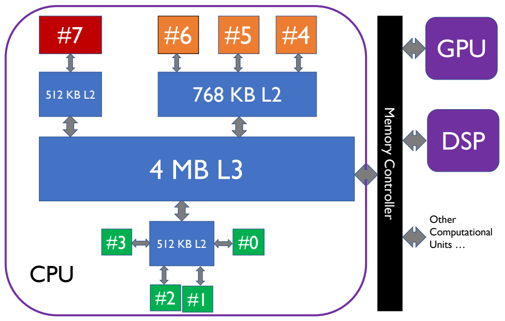


Figure 1: SD865's Heterogenous SoC Architecture

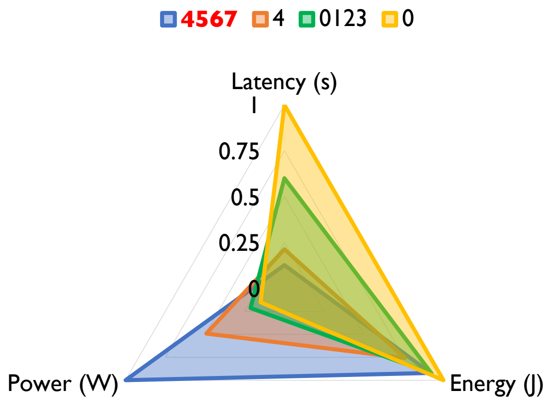


Figure 2: Training Resnet34.

procedures to modify such system parameters, making them infeasible for deployment due to risks outlined previously 2. Melon [62] and Sage [38] reduce the memory footprint of training through recomputation and micro-batching. Mandheling exploits platform-specific methods to improve energy efficiency, but is limited to operators that support mixed-precision training [64].

Heterogeneity of Smartphone SoCs stems from packing the CPU, GPU, memory, and other compute elements into a single chip, and can vary according to the application. The Snapdragon SD865 [22] SoC outlined in Fig 1 found in high-end devices has four low-powered cores (#0-3), four low-latency cores (#4-7), an embedded GPU and a DSP. One of the low-latency cores ("Prime" core #7) can run at higher speeds to achieve the lowest latencies. The SD855 [21] follows the same topology but with slower cores, while the SD845 [20] lacks a Prime core. These SoCs are thus suited for mid-range and entry-level devices respectively.

3 MOTIVATION

Factors: SoC and Model Architectures. The choice of CPU cores and DNN model architecture can affect performance metrics such as training latency, average power and energy consumed. Fig 2 details the resource usage to train Resnet34 on a Pixel 3 using PyTorch, on a variety of CPU core combinations. PyTorch uses a *greedy* strategy to pick as many threads as there are low-latency cores. The fastest choice to train the network is to use all the low-latency cores (i.e.,4567), with higher compute scaling improving

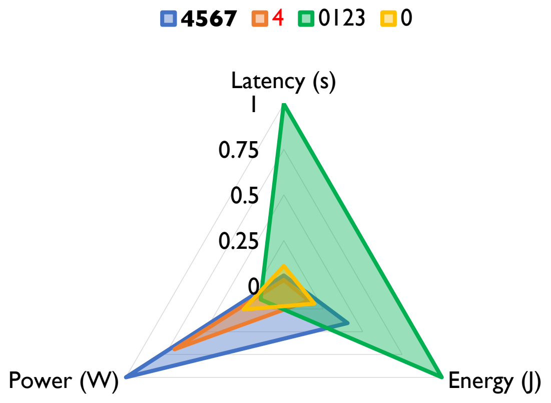


Figure 3: Training ShuffleNetv2.

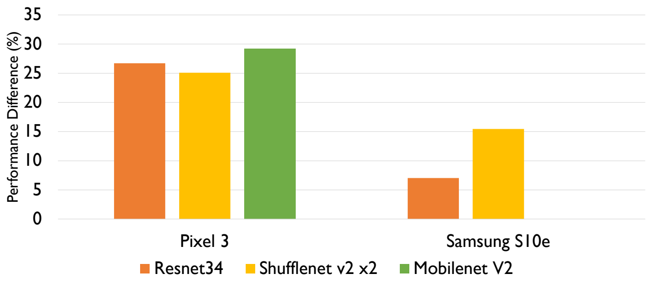


Figure 4: Impact of training on PCMark score.

latency. In contrast, the most energy-efficient choice is to involve any one of the low-latency cores. This brings up an interesting observation: *low power usage does not translate to low energy usage*; while combinations involving the low-power cores (i.e., 0–3) are always more power-efficient, they need not be more energy-efficient. Lower power leads to slower execution, increasing the total energy spent over a longer period of time.

However, these observations are not universal. Training ShuffleNet on the Pixel 3 (Fig 3) results in using one of the low-latency cores as both the fastest and most energy-efficient choice. Compute scaling becomes ineffective due to the memory-intensive depthwise convolution operations [54, 66]. Multiple threads running memory-intensive operations compete for the cache leading to cache-thrashing, reducing overall performance. Using just one thread allows the cache to be used in an exclusive manner. This is a known issue that has been addressed in GPUs [54] and Intel CPUs [6], but is yet to be addressed for ARM CPUs. In the absence of a pre-optimized training backend, it is necessary to customize the execution for every model and SoC combination.

Consequence: User Experience. On-device training adversely impacts user-experience due to its resource-intensive profile, restricting FL deployments to train only when the device is idle [7, 58]. The adverse user experience can manifest as slower device responses, delayed video playback etc. We can quantify this impact by comparing the performance difference of a representative benchmark of real-world usage, like PCMark Work 3.0 [17]. Training in the background has a measurable impact on the PCMark score Fig 4, with the entry-level Pixel 3 being impacted more than the Samsung S10e. With a majority of Android applications only using 1–2 threads [36], this presents an opportunity to exploit other CPU cores that are either under lower load or are being used by low-priority background services, enabling the training to run even the phone is being used.

Implications for system design. The execution strategy must work across the multitude of hardware and model combinations while considering the impact on user experience.

4 FLAMINGO

Flamingo is a real-time adaptive system for on-device DNN training on smartphone SoCs, and is the *first* to improve performance and energy efficiency of training *while minimizing the impact on user experience*. This leads to improved performance for mobile applications locally as well as quicker model convergence for distributed applications such as FL. Figure 5 outlines the overall architecture of Flamingo for an FL application. A local application would omit the Central Coordinator and FL Aggregator.

4.1 Design Overview

At its core, Flamingo explores combinations of CPU cores as execution choices to optimize for training time, and identifies alternative choices when training interferes with user-facing foreground applications.

Flamingo exploits the heterogeneity in smartphone SoCs to provide many execution choices to ensure that the device can continue training under a wider range of resource constraints. Flamingo infers interference with other applications without rooting the device and does not need invasive power monitors to measure the energy expenditure of on-device training, thus enabling large-scale deployment on Android devices. We also implement a standardized communication interface for the client to be compatible with existing distributed frameworks , e.g.PySyft (See App B for details).

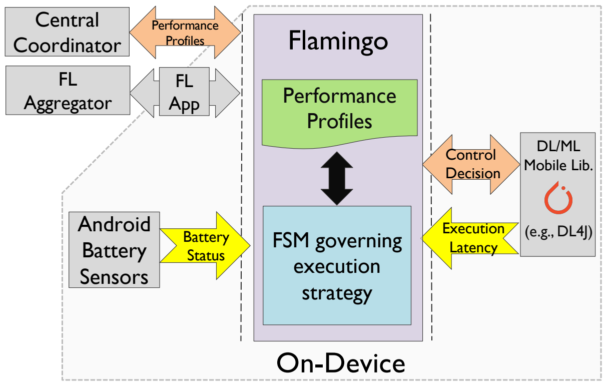
Here, we summarize the sequence of steps required to involve a smartphone under Flamingo and elaborate each step (except Monitoring) in the following subsections:

(1) Monitoring: Once installed, Flamingo monitors the battery and device states to decide on accepting a training request. Flamingo declines the request if the battery is above 35°C, preventing battery life reduction and thermal pain [10, 35, 50]. When the device is idle, Flamingo monitors battery drain to determine the background services' power usage.

(2) Exploration: Flamingo explores execution choices by profiling their resource usage when the phone is idle and unplugged, attributing the battery drain to the training and background services alone. It picks a different execution choice for each training request until all choices are explored.

(3) Training: Once exploration is complete, Flamingo can accept a training request even when the phone is not idle, but only if the battery is charging or is above a minimum level to prevent low battery levels. The request can originate from a remote coordinator or a local app that needs training services (e.g. next-word prediction for keyboards). It uses the performance profiles of execution choices to dynamically migrate the execution based on inferred interference (Fig 5b).

Flamingo is implemented in user-space[1] within Termux [24], a Linux Terminal emulator for Android. We envision Flamingo as an OS-integrated service to enable on-device training for any app.



(a) Architecture of Flamingo

```
for _ in range(num_batches):
    battery_status = flamingo.get_battery()
    if battery_status != "charging" or battery_status <
        MIN_BAT_THRESHOLD:
        break

    latency = train(execution_choice)
    # Detect interference
    if latency > latencies[execution_choice] * 1.1:
        # Downgrade Execution Choice
        execution_choice -= 1
        grade_time = time.now()
    # Explore to upgrade choice
    elif time.now() - grade_time > _2_minutes:
        execution_choice += 1

    flamingo.migrate(execution_choice)
```

(b) Control Loop

Figure 5: Design of Flamingo

4.2 Exploring Execution Choices

In order to accommodate varying compute resource availability during training, we explore running the training on different combinations of compute units. These combinations can be a selection of CPU cores, or other execution units like the mobile GPU. Since the PyTorch backend we use is implemented only for CPUs, we limit the exploration to a combination of CPU cores, but design our system to be agnostic to the execution choice to scale to other execution units in the future (Figure 5a). Each choice is profiled by training on a fixed number of batches, and amortizing the resource usage for one local step. For e.g., the state-space for a Pixel 3 has 8 choices, with 4 choices of low-latency cores and 4 choices of low-powered cores and are enumerated in 4.3. We delve deeper into the general state-space of choices in Appendix B. Flamingo explores the state-space in a *work-conserving* manner, by profiling while participating in model training.

[1]https://github.com/SymbioticLab/FedScale/tree/master/fedscale/edge/termux

This exploration can be eliminated by leveraging the central coordinator(s) in distributed learning systems. For e.g., the coordinator in federated learning can store the performance profiles for a specific device model once the first client running on that model completes its exploration. The aggregator can then redistribute these performance profiles to new clients running on the same device model, allowing them to skip exploration to immediately becoming available to participate.

4.3 On-Device Training

Flamingo maximizes training performance in the presence of user-facing foreground applications by dynamically modifying its compute footprint.

Our strategy prefers shorter training latencies by using low-latency cores (large footprint), but relinquishes them (leading to a smaller footprint) to foreground applications when necessary. Flamingo only relinquishes as many cores as is necessary to minimize interference, thus making *effective use of unused compute* in the resource constrained environment. It sorts the execution choices according to their performance profile in the decreasing order of compute footprint, enabling Flamingo to downgrade its footprint when detecting interference. Flamingo then *tailors the order to the SoC-DNN combination* at hand by avoiding sub-optimal choices and those that do not present a viable trade-off, i.e.increasing the footprint does not reduce training latency. The following rules define a relative order between execution choices,

(1) Using more cores of the same type is preferred (e.g. `4567` preferred over `4`).
(2) Using any number of low-latency cores is preferred over using any number of low-power cores (e.g. `4` preferred over `0123`).
(3) Prime cores (#7), if present, are preferred over other low-latency cores (#4-6) (i.e. order[47] < order[45]).

Following these rules, the total order of choices for a Pixel 3 (Prime core absent) is `4567`, `456`, `45`, `4`, `0123`, `012`, `01`, `0`.

Flamingo then prunes choices that present no viable tradeoff. For e.g. choosing `4567` to train ShuffleNet on a Pixel 3 uses more cores than choosing just `4`, but also worsens both latency and energy efficiency(Figure 3). Pruning finds choices that work best for a SoC–model combination.

The performance profile of an SoC–model combination can in turn inform the design of the DL model based on whether it is able to maximize the utilization of all compute resources. For e.g., ShuffleNetv2 could switch to a different convolution operation to benefit from compute scaling.

5 EVALUATION

We evaluate Flamingo in real-world settings on smartphones and in a simulated setting of federated learning to gauge its large-scale impact in distributed settings, by training three deep-learning models on CV and NLP datasets.

5.1 Methodology

Experimental Setup We benchmark (see App B) the energy usage and latency for each DL model on 5 mobile-devices: Galaxy Tab S6, OnePlus 8, Samsung S10e, Google Pixel 3 and Xiaomi Mi 10, to

Device	Speedup (Avg:12×)			Energy Efficiency (Avg:8×)		
	Resnet34	ShuffleNetv2_x2	MobileNetv2	Resnet34	ShuffleNetv2_x2	MobileNetv2
Galaxy Tab S6	1.9×	21×	14.5×	1.9×	12.2×	9.4×
OnePlus 8	2.1×	17×	13.9×	2.4×	8.5×	7.5×
Google Pixel 3	1×	1.8×	1.6×	1×	1.8×	2.3×
Samsung S10e	1.9×	39×	31.8×	2.1×	39×	17.4×
Xiaomi Mi 10	2.1×	17.2×	14×	2.2×	7.8×	5.8×

Table 1: On-device Speedups and Energy Efficiency improvements over baseline.

Task	Dataset	#FL Clients	#Samples	Model	Target Acc.	Speedup	Energy Eff.
Classification	OpenImage [9]	14,477	1,672,231	MobileNetv2 [60]	52.8%	23.3×	7.0×
				ShuffleNetv2_x2 [66]	46.3%	6.5×	5.8×
Speech Recognition	Google Speech [63]	2,618	105,829	ResNet-34 [41]	60.8%	1.2×	1.6×

Table 2: Summary of improvements on time to accuracy and energy usage in large-scale evaluation.

obtain their performance profiles. We use FedScale [46] to evaluate Flamingo in a large-scale federated-learning setting, using 20 A40 GPUs [34]. We emulate device behavior using pre-processed device traces (see App A) from the GreenHub dataset [51] (300k devices). We use the PCMark Work 3.0 benchmark score [17], which includes *realistic* workloads like web-browsing, video streaming, etc. as opposed to stress testing viz. [8, 16], to measure the impact of background training on the user-experience. We compare the efficacy of Flamingo against the Energy-Aware Scheduler [32] to reduce the impact on the PCMark score.

Calculating Energy Usage The energy is calculated by logging the drop in battery SoC. Instantaneous Power is calculated as Voltage * Current. This can be approximated by averaging the current and voltage over an interval of 1 % battery level drop. Average Power = $(V_{start} + V_{end})/2 * (battery\_capacity/100)/\Delta T$, where V_{start} and V_{end} are the battery voltages at the start and end of the interval, and $battery\_capacity$ is the charge capacity of the smartphone's battery in Coulombs, and ΔT being the length of the time interval. The energy can be calculated across every drop in battery level, and thus can be summed up in a piece-wise manner across intervals that overlapped with the benchmark of concern to produce a total energy usage estimate.

Datasets and Models We run two categories of applications with real-world datasets of different scales, detailed in Table 2. The ResNet34 model is trained on the GoogleSpeech dataset, and MobileNetv2 and ShuffleNetv2_x2 are trained on the OpenImage dataset.

Choice of On-device Backend We considered PyTorch [18], TFLite [23] and MNN [1] as potential on-device backends.While PyTorch could train all models used in our experiments, TFLite and MNN failed due to known model conversion issues, lack of operator support, or run-time issues related to memory bounds or GPU support [15, 26]. We selected PyTorch as a backend and will expand to other backends once these issues are addressed. PyTorch can also be used as a backend to train on iOS devices [19]. iOS offers a Thread Affinity API which hints the scheduler to bind two or more threads to the same L2 cache to improve locality, but currently cannot bind a thread to a CPU [11]. However, the QoS value of a process has been shown as an effective way to migrate the process between performance and efficiency cores and could be used towards porting Flamingo to iOS devices [12, 14].

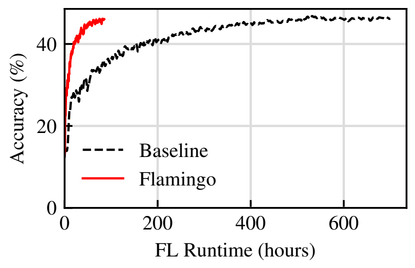
Hyperparameters The minibatch size is set to 16 for all tasks, with a learning rate of 0.05, and a SGD optimizer. The baseline uses the PyTorch execution choice, which uses all low-latency cores in the SoC configuration. We use the FedAvg [52] averaging algorithm to combine model updates.

Realistic Energy Budget We model FL-client device-failures due to the additional energy cost of on-device training, using real-world battery charge-discharge traces [51]. We calculate total available energy by tracking the energy added through charging, and energy spent due to daily usage and on-device training. The device is considered offline when the total available energy falls below a minimum threshold.

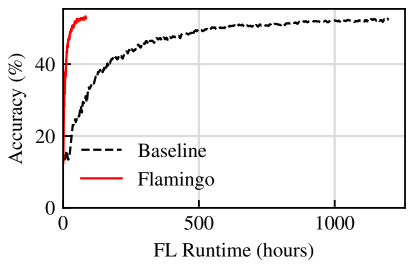
Metrics For the real-world local evaluation, we want to reduce execution latency while increasing energy efficiency. For the large-scale FL simulation, we want to reduce the training time to reach target accuracy, while reducing energy usage and number of clients lost. We set the target accuracy to be the highest achieved by either the baseline or Flamingo.

Device	Baseline	Flamingo	Improvement
Galaxy Tab S6	-10.2 %	-5.8 %	43 %
OnePlus 8	-12.5 %	0 %	100 %
Google Pixel 3	-27 %	-3.1 %	88 %
Samsung S10e	-11.2 %	0 %	100 %

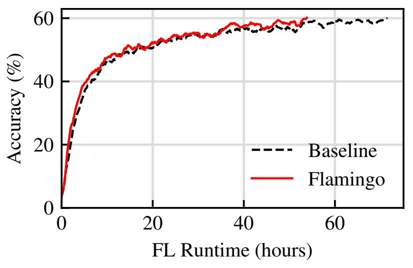
Table 3: Impact of background-training on user-experience as measured by PCMark.



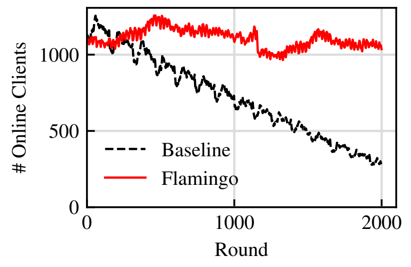
(a) ShuffleNet: Accuracy vs Time



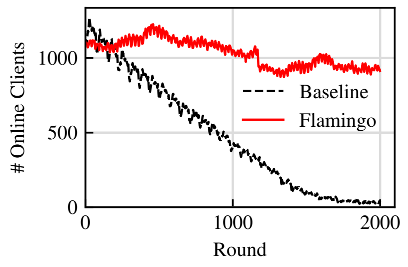
(b) MobileNet: Accuracy vs Time



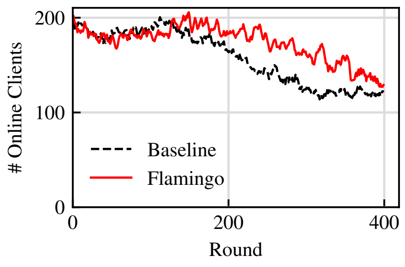
(c) ResNet34: Accuracy vs Time



(d) ShuffleNet: #OnClients vs Rounds



(e) MobileNet: #OnClients vs Rounds



(f) ResNet34: #OnClients vs Rounds

Figure 6: Large-Scale Federated Learning Results

5.2 Real-World Evaluation

Flamingo achieves 12× on-device training speedups and is 8× more energy efficient than the baseline on average (Table 1). Flamingo picks optimal choices, especially for ShuffleNet and MobileNet models that are affected by cache-thrashing (Sec 3), overlooked by the baseline.

Flamingo reduces impact on user-experience by 43-100 % measured by the impact of training on the PCMark Work 3.0 benchmark as shown in Table 3, entirely eliminating the interference in OnePlus8 and S10e devices. This is due to the fact that Flamingo is tightly-coupled to the on-device backend to scale down its compute footprint using alternative choices discovered through state-space exploration and relinquishing higher performing compute to foreground applications, which the baseline cannot. The improvement is particularly stark for the Pixel 3 since it is the lowest-end entry-level device, implying Flamingo can benefit a majority of smartphones [56] that are entry-level devices.

5.3 Large-Scale Evaluation

Flamingo reduces time to accuracy by 1.2-23× and improves energy efficiency 1.6-7× for federated learning. Table 2 summarizes the overall time-to-accuracy improvement, and Fig 6a-6c report the corresponding training performance over time. Flamingo's improved energy efficiency directly translates to a larger pool of participants available to perform training, unlike the baseline which steadily loses devices with every passing round due to exhausting the total available energy (Fig 6d, 6e). This indicates that Flamingo allows clients to remain online for longer which helps the model train faster, and device **owners can simultaneously maintain their regular usage behavior while participating in FL**. Fig 6c reports the ResNet-34 performance on the speech recognition task. Due to the small number of clients that can be emulated in this task, Flamingo and baseline achieve comparable time-to-accuracy performance Fig 6f. However, Flamingo achieves better energy efficiency by tailoring the execution choice to each device training Resnet34.

6 CONCLUSION

The need to train DNN models on end-user devices such as smartphones is only going to increase with privacy requirements becoming more prevalent. We proposed Flamingo, a real-time adaptive system specialized to efficiently train DNNs on Android smartphones, and to the best of our knowledge is the first to prioritize user experience. By tightly coupling the scheduling strategy with the DNN and SoC architecture, Flamingo achieved accelerated training times and reduced energy consumption across various tasks. This enhancement proved advantageous in distributed training scenarios like federated learning, and offered an effective on-device training solution that operates seamlessly alongside user applications without any interference. With this paper being a successful first step, Flamingo will enable further research in this domain and future practitioners can build on top of its toolchain. In order to enable such advances, we will be releasing Flamingo's source, and will regularly update it to accommodate to diverse backends based on the input from the community.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback which improved the final paper. We thank the CloudLab team for providing GPU servers for our experiments. We also thank members of the SymbioticLab, Vasudha Varadarajan and Nishil Talati for their valuable comments and suggestions. This work was supported in part by NSF grant CNS-2106184 and a grant from Cisco.

REFERENCES

[1] Alibaba MNN. https://github.com/alibaba/MNN.
[2] Android Sandbox. https://source.android.com/security/app-sandbox.
[3] Android WakeLock. https://developer.android.com/reference/android/os/PowerManager.WakeLock.
[4] Apple Core ML. https://developer.apple.com/documentation/coreml.
[5] Deeplearning for Java. https://deeplearning4j.konduit.ai/.
[6] Depthwise convolution in intel mkl-dnn. https://oneapi-src.github.io/oneDNN/dev_guide_convolution.html?highlight=depthwise.
[7] Federated learning: Collaborative machine learning without centralized training Data. *Google AI Blog*.
[8] GeekBench Android. https://browser.geekbench.com/android-benchmarks.
[9] Google Open Images Dataset. https://storage.googleapis.com/openimages/web/index.html.
[10] Huawei battery recommendations. https://consumer.huawei.com/za/support/battery-charging/lithium-ion-battery/.
[11] iOS Thread Affinity API. https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/mach/thread_policy.h.
[12] iOS/macOS DispactchQoS API. https://developer.apple.com/documentation/dispatch/dispatchqos.
[13] Linux CPU affinity. https://man7.org/linux/man-pages/man2/sched_getaffinity.2.html.
[14] macOS DispatchQoS Experiments. https://eclecticlight.co/2022/01/07/how-macos-controls-performance-qos-on-intel-and-m1-processors/.
[15] ONNX->MNN conversion issue. https://github.com/alibaba/MNN/issues/2298.
[16] PassMark Android. "https://www.androidbenchmark.net/".
[17] PCMark for Android. https://benchmarks.ul.com/pcmark-android.
[18] PyTorch. https://pytorch.org/.
[19] PyTorch on iOS. https://pytorch.org/mobile/ios/.
[20] Snapdragon 845 specs. https://en.wikichip.org/wiki/qualcomm/snapdragon_800/845.
[21] Snapdragon 855 specs. https://en.wikichip.org/wiki/qualcomm/snapdragon_800/855.
[22] Snapdragon 865 specs. https://en.wikichip.org/wiki/qualcomm/snapdragon_800/865.
[23] TensorFlow Lite. https://www.tensorflow.org/lite/models.
[24] Termux. https://termux.com/.
[25] Termux API. https://wiki.termux.com/wiki/Termux:API.
[26] TFLite GPU Delegate Issue. https://github.com/tensorflow/tensorflow/issues/53335.
[27] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
[28] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, Pedro P. B. de Gusmão, and Nicholas D. Lane. Flower: A friendly federated learning research framework. In *arxiv.org/abs/2007.14390*, 2020.
[29] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. In *MLSys*, 2019.
[30] Luca Casati and Andrea Visconti. The dangers of rooting: data leakage detection in android applications. *Mobile Information Systems*, 2018, 2018.
[31] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.
[32] ARM Developer. Energy aware scheduling (eas).
[33] Apple Differential Privacy Team. Learning with privacy at scale. In *Apple Machine Learning Journal*, 2017.
[34] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
[35] Begum Egilmez, Gokhan Memik, Seda Ogrenci-Memik, and Oguz Ergin. User-specific skin temperature-aware dvfs for smartphones. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1217–1220. IEEE, 2015.
[36] Cao Gao, Anthony Gutierrez, Madhav Rajan, Ronald G. Dreslinski, Trevor Mudge, and Carole-Jean Wu. A study of mobile device utilization. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 225–234, 2015.
[37] Avishek Ghosh, Jichan Chung, Dong Yin, and Kannan Ramchandran. An efficient framework for clustered federated learning. In *NeurIPS*, 2020.
[38] In Gim and JeongGil Ko. Memory-efficient dnn training on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 464–476, New York, NY, USA, 2022. Association for Computing Machinery.
[39] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating dnn model porting for on-device inference at the edge. In *NSDI*, 2021.
[40] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.
[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.
[42] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.
[43] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, Kaikai Wang, Anthony Shoumikhin, Jesik Min, and Mani Malek. Papaya: Practical, private, and scalable federated learning. In *MLSys*, 2022.
[44] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. In *Foundations and Trends in Machine Learning*, 2021.
[45] Young Geun Kim and Carole-Jean Wu. Autofl: Enabling heterogeneity-aware energy efficient federated learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–198, 2021.
[46] Fan Lai, Yinwei Dai, Sanjay S. Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. FedScale: Benchmarking model and system performance of federated learning at scale. In *International Conference on Machine Learning (ICML)*, 2022.
[47] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021.
[48] Li Li, Haoyi Xiong, Zhishan Guo, Jun Wang, and Cheng-Zhong Xu. Smartpc: Hierarchical pace control in real-time federated learning system. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 406–418, 2019.
[49] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *MLSys*, 2020.
[50] Shuai Ma, Modi Jiang, Peng Tao, Chengyi Song, Jianbo Wu, Jun Wang, Tao Deng, and Wen Shang. Temperature effect and thermal impact in lithium-ion batteries: A review. *Progress in Natural Science: Materials International*, 28(6):653–666, 2018.
[51] Hugo Matalonga, Bruno Cabral, Fernando Castor, Marco Couto, Rui Pereira, Simão Melo de Sousa, and João Paulo Fernandes. Greenhub farmer: Real-world data for android energy mining. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 171–175. IEEE Press, 2019.
[52] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.
[53] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier C. van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, Sudeep Agarwal, Julien Freudiger, Andrew Byde, Abhishek Bhowmick, Gaurav Kapoor, Si Beaumont, Áine Cahill, Dominic Hughes, Omid Javidbakht, Fei Dong, Rehan Rishi, and Stanley Hung. Federated evaluation and tuning for on-device personalization: System design & applications. *CoRR*, abs/2102.08503, 2021.
[54] Zheng Qin, Zhaoning Zhang, Dongsheng Li, Yiming Zhang, and Yuxing Peng. Diagonalwise refactorization: An efficient training method for depthwise convolutions. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.
[55] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. In *ICLR*, 2021.
[56] Vijay Janapa Reddi, Hongil Yoon, and Allan Knies. 2 billion devices and counting: An industry perspective on the state of mobile computer architecture. *IEEE Micro*, 38:6–21, 2018.
[57] Daniel Rothchild, Ashwinee Panda, Enayat Ullah, Nikita Ivkin, Ion Stoica, Vladimir Braverman, Joseph Gonzalez, and Raman Arora. Fetchsgd: Communication-efficient federated learning with sketching. In *ICML*, 2020.
[58] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. A generic framework for privacy preserving deep learning. *arXiv preprint arXiv:1811.04017*, 2018.
[59] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. A generic framework for privacy preserving deep learning. https://github.com/OpenMined/PySyft, 2018.

[60] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.
[61] Naichen Shi, Fan Lai, Raed Al Kontar, and Mosharaf Chowdhury. Fed-ensemble: Improving generalization through model ensembling in federated learning. In *arxiv.org/abs/2107.10663*, 2021.
[62] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 450–463, New York, NY, USA, 2022. Association for Computing Machinery.
[63] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. In *arxiv.org/abs/1804.03209*, 2018.
[64] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, MobiCom '22, page 214–227, New York, NY, USA, 2022. Association for Computing Machinery.
[65] Qiyang Zhang, Zuo Zhu, Ao Zhou, Qibo Sun, Schahram Dustdar, and Shangguang Wang. Energy-efficient federated training on mobile device. *IEEE Network*, pages 1–7, 2023.
[66] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.
[67] Wennan Zhu, Peter Kairouz, Brendan McMahan, and Wei Li Haicheng Sun. Federated heavy hitters discovery with differential privacy. In *AISTATS*, 2020.

A DATA PRE-PROCESSING

A.1 Monitoring and modeling resource usage

Due to the inherent scalability and logistical issues associated with collecting the resource usage data of smartphones, we emulate the background resource usage logging mentioned in Section 4 by using a system trace dataset provided by GreenHub [51]. This data was collected by a background app logging the usage of various resources, resulting in collecting 50 million samples from 0.3 million Android devices that are highly heterogeneous in terms of device models and geographical locations. Each sample contains values for different resources, including the battery_level and battery_state, at a particular timestamp. We then filter the dataset for high quality traces and then pre-process the data, detailed in Appendix A.

A.2 Trace Selection and Re-Sampling

We observe that the sampling frequencies and sampling periods for users are not consistent given the complicated real-world settings. To utilize the data, we first pre-processed the data. We selected 100 high-quality user traces out of 0.3 million users with the following criteria: 1) The user has a sampling period that is no less than 28 days; 2) The user has an overall sampling frequency no less $\frac{5}{432}$ Hz, which is equivalent to 100 samples one day on average across the whole sampling period; 3) The maximum time gap between two adjacent samples is no larger than 24 hours; 4) The number of time gaps between two adjacent samples that are larger than 6 hours should be no more than 15. We resample the non-uniform traces using Piecewise Cubic Hermite Interpolating Polynomial (i.e. scipy.interpolate.PchipInterpolator) to a fixed rate of 10min frequency.

After the resampling of "battery_level", we set the "battery_state" to reflect whether the battery is charging(1), not discharging(0), or discharging(-1). This depends on the sign of the difference between the current "battery_level" and the previous "battery_level" for each pair of consecutive data points.

Data augmentation for temporal heterogeneity. In order to simulate client availability across all time zones, we select sub-intervals of 100 traces shifted by 1 hour, 23 times. This results in 2400 clients spread across the planet.

B IMPLEMENTATION

In this section, we discuss the implementation we followed corresponding to each step outlined in Section 4.

State-space of Execution Choices. For a given SoC, the state-space of execution choices explores the effect on running the training on a) different scales and b) different types of CPU cores. We explain this using the SD865 SoC 1 as an example, which has 4 low-power cores and 4 high-power cores. A brute-force exploration of the state-space would result in $8! = 40320$ choices. We exploit the homogeniety of cores (i.e. using cores "01" is the same as using cores "23") to avoid redundant choices and shrink the state-space to $(4+1)*(4+1)-1 = 24$ choices. We conducted experiments to study the efficacy of choices involving heterogeneous cores, e.g. "40", "5410", by running DNN training of the evaluated models 2 and parallelized matrix multiplication. The heterogeneous choices (e.g. "5410") were consistently outperformed by their low-latency subset choice (e.g. "54") in energy and in latency. This is due to the fact that the low-power cores become stragglers, thus preventing higher utilization of the low-latency cores and increase the overall latency. Pruning away the heterogeneous choices reduces the state space to $4+4 = 8$, resulting in the following state space: "0", "01", "012", "0123", "4", "45", "456", "4567". We also explore the energy-latency tradeoff between the standard low-latency core and the "Prime" low-latency core when present, e.g. in SD865 and SD855, since the "Prime" latency cores consume more power but can take less time to complete tasks. The 3 additional choices involving the "Prime" core with other low-latency are "457", "47" and "7", bringing the size of state-space to 11. The SD845 SoC used in the Pixel 3 only has 8 choices since it does not have a "Prime" core.

Implements Standardized Interface. We intend the client communication interface of Flamingo to follow the existing standard (i.e., client implements `run_local_step` and `isActive`) in order to seamlessly work with existing distributed solutions such as federated learning server-client frameworks (e.g., PySyft). This also reduces the possibility of introducing unintentional privacy leaks by deviating from standardized client-coordinator interfaces.

Scheduling in Android. The first hurdle of running the training process on Android was to ensure that the scheduler does not put the process to sleep once the phone's screen turns off. We solve this issue by acquiring a "WakeLock" [3], an Android level feature that allows the app unrestricted use of the processor. In order to be able to explore the performance and energy usage of different core combinations, we needed low-level control to limit the scheduling of the training processes to a specific core or a set of cores and change the number of threads at run-time. This required access to the Linux scheduling API function calls `sched_setaffinity` and `sched_getaffinity` [13]. The choice of the deep-learning library implementation can determine native access to these APIs.

Mobile Deep Learning Library. We considered mobile-oriented versions of predominant deep-learning frameworks, like PyTorch and Deeplearning4J(DL4J), since they are already used in client-side executors like KotlinSyft [59]. Although Android's JNI API offers access to system calls, the threading API used by the Android builds of PyTorch and DL4J do not offer a way to change the number of threads at run-time. On the other hand, PyTorch's Linux backend did have a way dynamically control the number of threads when compiled with OpenMP [31].

Execution Environment. We utilize a Linux-like environment created by Termux [24], a Linux terminal emulator app running on an Android phone. This lets us access many low-level system calls, including `sched_setaffinity` and `sched_getaffinity`. We use PyTorch v1.8.1 with OpenMP, compiled on the device for the 64-bit ARM architecture to run the local training. We use the termux-api [25] interface to monotor the system-state, including the battery state and charge level.

Performance and Energy Benchmarking. After setting the CPU affinity, training costs associated with a particular deep-learning model is benchmarked by amortizing the battery usage across multiple runs detailed. We then subtract power required to run other processes and components of the phone to arrive at the energy and power usage of the training. In order to minimize the effect of the external environment and other applications on the benchmark, we stop all unnecessary services and background processes to isolate the execution from any interference.

In a distributed setting like Federated Learning, this benchmarking is performed during the first rounds of the device's participation when the phone is idle and is discharging. This is to minimize interference with other applications and accurately calculate the energy usage of the training. After benchmarking the entire state-space and pruning choices that do not present any viable tradeoff 4.2, the performance profile of the device is compiled and sent to the Central Coordinator 5a. This profile can be reused by other FL-clients running on the same device (e.g. other Pixel 3 devices) and skip the benchmarking process altogether, leading to efficient and user-experience friendly execution from the start.