



Mandhel ing: Mixed-Precision On-Device DNN Training with DSP Offloading

Daliang Xu^{1*}, Mengwei Xu^{2**}, Qipeng Wang¹, Shangguang Wang², Yun Ma¹, Kang Huang³, Gang Huang¹, Xin Jin^{1#}, Xuanzhe Liu^{1#}

¹Key Lab of High Confidence Software Technologies (Peking University), Beijing, China

²State Key Laboratory of Networking and Switching Technology (BUPT), Beijing, China

³Linggui Tech Company, Beijing, China.

{xudaliang,mayun,hg,xinjin@pku.edu.cn,wangqipeng@stu.pku.edu.cn}

{mwx,sgwang}@bupt.edu.cn

kang.huang@nlptech.com

ABSTRACT

This paper proposes Mandhel ing, the first system that enables highly resource-efficient on-device training by orchestrating mixed-precision training with on-chip Digital Signal Processor (DSP) offloading. Mandhel ing fully explores the advantages of DSP in integer-based numerical calculations using four novel techniques: (1) a CPU-DSP co-scheduling scheme to situationally mitigate the overhead from DSP-unfriendly operators; (2) a self-adaptive rescaling algorithm to reduce the overhead of dynamic rescaling in backward propagation; (3) a batch-splitting algorithm to improve DSP cache efficiency; (4) a DSP compute subgraph-reusing mechanism to eliminate the preparation overhead on DSP. We have fully implemented Mandhel ing and demonstrated its effectiveness through extensive experiments. The results show that, compared to the state-of-the-art DNN engines from TFLite and MNN, Mandhel ing reduces per-batch training time by 5.5 \times and energy consumption by 8.9 \times on average. In end-to-end training tasks, Mandhel ing reduces convergence time by up to 10.7 \times and energy consumption by 13.1 \times , with only 1.9%–2.7% accuracy loss compared to the FP32 precision setting.

CCS CONCEPTS

- Human-centered computing → Ubiquitous and mobile computing;
- Computing methodologies → Artificial intelligence.

KEYWORDS

Mobile device, deep learning, DSP offloading

ACM Reference Format:

Daliang Xu^{1*}, Mengwei Xu^{2**}, Qipeng Wang¹, Shangguang Wang², Yun Ma¹, Kang Huang³, Gang Huang¹, Xin Jin^{1#}, Xuanzhe Liu^{1#}. 2022. Mandhel ing: Mixed-Precision On-Device DNN Training with DSP Offloading. In *The 28th Annual International Conference on Mobile Computing and Networking (ACM*

*Equal contributions; #Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM MobiCom '22, October 17–21, 2022, Sydney, NSW, Australia

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9181-8/22/10...\$15.00

<https://doi.org/10.1145/3495243.3560545>

MobiCom '22, October 17–21, 2022, Sydney, NSW, Australia. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3495243.3560545>

1 INTRODUCTION

With ever-increasing data privacy concerns [5], empowering a mobile device to train a deep neural network (DNN) locally (i.e., *on-device training*) has recently attracted attention from both academia and industry [1, 2, 79]. Without giving away the training data out of controlled domain, on-device training enables (i) geo-distributed devices to collaboratively establish a high-accuracy model [16, 32] or (ii) a single device to personalize and adapt its model to its environment [79].

However, a key obstacle to practical on-device training is its huge resource cost. According to our preliminary measurements with two popular DL libraries (TFLite [52] and MNN [39]), training ResNet-50 with one batch (batch size 32) takes 4.6 GB of memory and 36.4 seconds on a Xiaomi 10 smartphone equipped with a Snapdragon 865 CPU. The consumed energy equals watching a 1080P-definition video for 111.2 seconds. In an end-to-end learning scenario, it typically takes thousands or even more such batches of training and the accumulated cost becomes prohibitively expensive. Unfortunately, this issue has not been well explored by the research community. Existing studies gaining impressive benefits for on-device inference tasks [19, 35, 53, 73, 81] can hardly be applied to on-device training due to the huge gap between inference and training workloads, e.g., the different computation patterns and accuracy requirements.

To optimize the performance of on-device training, this work is motivated by two key observations. First, traditional DNN training is mostly performed in FP32 data format to achieve good model accuracy. However, the ML community has recently proposed various *mixed-precision training* algorithms [21, 36, 50, 51, 62, 74, 78, 83, 88, 90, 91, 93], where the weights and activations generated during training are represented not only by FP32 but also by lower-precision formats such as INT8 and INT16. By exploiting the hardware features in accelerating integer operations, these algorithms are effective in reducing the training-time resource cost while guaranteeing convergence accuracy, i.e., only a 1.3% loss on CIFAR-10 [83]. Second, modern mobile SoCs often consist of heterogeneous processors, among which the Digital Signal Processor (DSP) is ubiquitously available and particularly suits integer operations, i.e., INT8-based matrix multiplication. For example, Hexagon 698 DSP [8] is adequate to execute 128 INT8 operations in one cycle

and has been demonstrated to be $11.3\times/4.0\times$ more energy-efficient than CPU/GPU in DL inference tasks [87], respectively. Intuitively, under the mixed-precision training setting, we are interested in the question of whether we can partially offload the training workloads, especially those integer-based operations, from CPU to DSP to reduce the cost of on-device training.

In this paper, we propose a first-of-its-kind system, namely **Mandhel ing**, which enables highly resource-efficient, mixed-precision on-device training with on-chip DSP offloading. To facilitate developers in using different types of training algorithms on **Mandhel ing**, we investigate popular mixed-precision training algorithms and extract the key principles from them. Based on those principles, we incorporate a unified abstraction into **Mandhel ing**, as we will discuss in §3.2. With a given training algorithm and the model to be trained, **Mandhel ing** aims to minimize the training cost by judiciously co-scheduling various training operators to mobile DSP (mostly) and CPU. When designing **Mandhel ing**, however, we need to address the following major challenges that have not been explored in the existing literature.

- *Dealing with DSP-unfriendly operators.* In a typical mixed-precision training algorithm, some operators like Transpose and Normalization run slowly on DSP due to their irregular memory accesses [56] or lack of architecture-level support. A judicious scheme to determine what operators to offload to DSP while others are placed on CPU is needed to fully exploit those heterogeneous processors.
- *Dynamic rescaling does not fit DSP.* During the training, dynamic rescaling [74, 83, 88, 90] is a critical operation to quantize/dequantize among different data types. We observe that this operation runs slowly on DSP and can easily compromise the benefits of using it. Because dynamic scaling is inserted into each layer with the low-precision data type, simply scheduling it to CPU like other DSP-unfriendly operators incurs a high context-switching overhead and so needs to be optimized exclusively.
- *Exhausted data cache.* Training workloads impose high pressure on the DSP cache and a vanilla implementation leads to a low cache hit ratio. The reasons are twofold. First, training tasks often require a large batch size, which results in frequent memory accesses to access intermediate data. Second, the DSP cache is often smaller than the CPU cache, e.g., only half for the L2 cache on the Snapdragon 865. Considering that fully utilizing the processor cache is a killing factor for memory-intensive operations such as convolution weight gradients [73], an exhausted data cache on DSP is likely to act as a bottleneck in the training process.
- *Costly compute-graph preparation.* Unlike inference tasks, on-device training tasks usually use dynamic graphs to facilitate developers in developing and debugging [59]. However, preparing the compute graph on DSP takes a considerable amount of time to allocate DSP memory, build graph-to-operation references, etc. Therefore, eliminating the compute-graph preparation under DSP's tight memory budget is critical to **Mandhel ing**.

Key techniques of Mandhel ing. To address the aforementioned challenges, **Mandhel ing** presents the following novel techniques to fully unleash DSP's computing capacity:

- **CPU-DSP co-scheduling** (§3.3) is proposed to mitigate the overhead of DSP-unfriendly operators. The key idea is to reduce the number of context switchings brought by DSP-unfriendly operators and overlap CPU and DSP execution as much as possible. This is

achieved through our novel scheduling algorithm, which considers the latency of the operator executing on different processors and the overhead of CPU-DSP context switching. Unlike DeepX [46] and HERTI [28], which maximize parallel processing capability, **Mandhel ing**'s co-scheduling aims to use DSP offloading as much as possible when reducing the context-switching overhead.

- **Self-adaptive rescaling** (§3.4) is proposed to significantly reduce the overhead of dynamic rescaling by adaptively lowering its invoking frequency. This is motivated by our micro-experiments which demonstrate that, after the early stage of training, the actual changing frequency of the scale factor becomes low and its value becomes fairly stable.

- **Batch splitting** (§3.5) is proposed to reduce the cache pressure on DSP and thereby increase the cache hit ratio. **Mandhel ing** runs the intra-operator partition at the batch dimension because this solution does not influence the inputs and weights of the original convolution operation, and thus causes no redundant computations. **Mandhel ing** uses an intuitive yet effective method to identify the batch size splitting point. It also provides an integer-only scheme to efficiently concatenate the output from the split batch.

- **DSP compute subgraph reuse** (§3.6) is proposed to eliminate the preparation overhead of the DSP compute graph. This is motivated by a key observation that the model structure rarely changes during the training phase. However, directly reusing subgraphs [9, 26] can easily exceed the DSP memory budget, since training requires far more memory than inference. To further tackle DSP's memory constraint, we provide a practical subgraph-reusing algorithm based on the minimum dynamic memory allocation/deallocation principle.

Implementation and evaluation. We have fully implemented **Mandhel ing** with 15k LoC in C/C++ and 800 LoC in assembly language. **Mandhel ing** is a standalone framework that supports models exported from different front-end frameworks (e.g., TensorFlow [13] and PyTorch [59]) and is compatible with various mixed-precision training algorithms. We then conducted extensive experiments on six typical DNN models (VGG-11/16/19 [67], ResNet-18/34 [34], and InceptionV3 [69]) and three commodity mobile devices (Xiaomi 11 Pro, Xiaomi 10, and Redmi Note9 Pro). The results demonstrate that, compared to native support from TFLite and MNN, **Mandhel ing** can reduce per-batch training time and energy consumption by $5.5/8.9\times$ on average and up to $8.3\times/12.5\times$, respectively. Furthermore, compared to GPU-enhanced training, **Mandhel ing** speeds up per-batch training time by $7.1\times$ and reduces energy consumption by $5.8\times$ on average. In end-to-end training on a single device, comparing the FP32-based training algorithm and MNN, **Mandhel ing** accelerates model convergence by $5.7\times$ on average and reduces total energy consumption by $7.8\times$ on average. The improvements are even more profound in a federated learning scenario with $8.0\times$ convergence speedup and $10.6\times$ energy reduction on average. Meanwhile, the accuracy of trained models drops marginally with only 1.9%-2.7% loss compared to the accuracy of the FP32 precision setting, which is consistent with the theoretical results adopted by the ML community [21, 36, 50, 51, 62, 74, 78, 91-93]. The ablation study further distinguishes the effectiveness of every single key technique of **Mandhel ing**.

Contributions are summarized as follows.

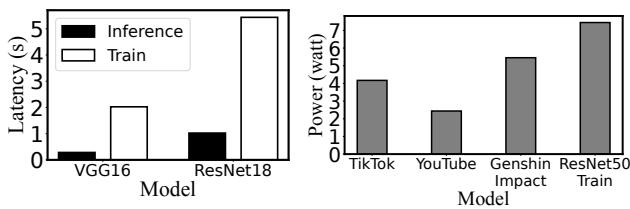


Figure 1: Preliminary measurements that highlight the huge resource cost of on-device training.

- We thoroughly explore the opportunities and challenges of DSP offloading for mixed-precision on-device training.
- We design and implement the first DSP-offloading-based mixed-precision on-device training framework, which incorporates four novel techniques: self-adaptive rescaling, batch splitting, CPU-DSP co-scheduling, and DSP compute subgraph reuse. The system has been fully open sourced¹.
- We evaluate Mandhelng with representative DNN models and commodity mobile devices. The results demonstrate Mandhelng's superior effectiveness and practical value.

2 BACKGROUND AND MOTIVATIONS

We briefly introduce some background and our motivations.

2.1 On-Device DNN Training

A trend is emerging to deploy DNN training locally on devices, to help especially with data privacy concerns (like GDPR [5]) in AI applications. Similar to DNN training on the cloud/server, on-device training also conducts a mini-batch sampling strategy, where each batch's training comprises three stages: the forward pass, the backward pass, and the weight update. The forward pass loads inputs and calculates the loss; the backward pass usually employs a specific optimizer like Stochastic Gradient Descent (SGD) [17] to obtain the gradients; lastly, the gradients are applied to the weights for the model update. Compared to on-device inference, the training task consumes more resources because: (i) the backward pass contains about $2\times$ FLOPs as the forward pass (the same as inference); (ii) a training process often involves hundreds or even thousands of mini-batches.

We conducted two measurement studies highlighting the prohibitively high resource cost of on-device training. Figure 1(a) shows that DNN training takes $7.14\times$ longer than inference with the same batch size, which is larger than the gap for theoretical FLOPs (about $3\times$). That is because training ops, especially weight-gradient ops, are more difficult to optimize because of the feature map's dynamic input width and height. Besides, Figure 1(b) shows that on-device training consumes more energy than typical video players (TikTok [10], YouTube [12]) and gaming apps (Genshin Impact [6]). For instance, training one batch (BS=32) of the ResNet-50 model costs the same energy as watching 36.4 seconds of videos on YouTube. Considering the minimal granularity for training is usually one epoch, which, let's say, contains 1,000 mini-batches, the energy cost is as high as watching 7.91 hours of videos on YouTube.

¹<https://github.com/UbiquitousLearning/Mandhelng-DSP-Training>

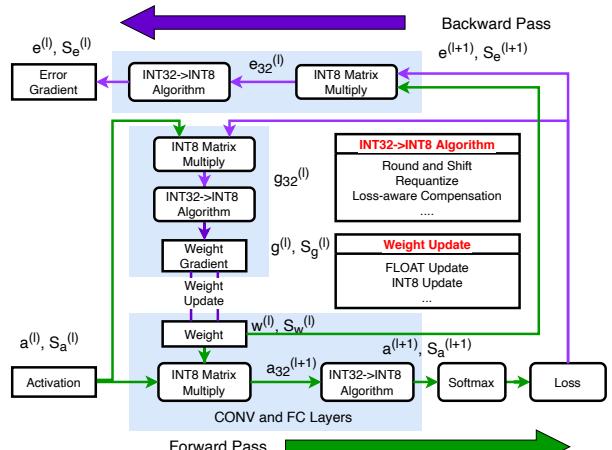


Figure 2: Workflow of mixed-precision training.

2.2 Mixed-Precision Training

Mixed-precision training algorithms have been proposed to reduce the resource cost of DNN training [21, 36, 50, 51, 62, 74, 78, 91–93]. These algorithms mainly exploit the high redundancy of data in DNNs (including the activation, weights, bias, and gradients) [54], which means that their representation precision can be reduced, e.g., from FP32 to FP16/INT8. Fewer bits per number enables DNN training to run faster through the single-instruction-multiple-data (SIMD) hardware parallelism, which is commonly available on mobile processors [72]. Here, we use INT8 training as an example to illustrate how typical mixed-precision training algorithms work. Figure 2 shows an exemplified workflow.

Forward pass. After quantization, activation $a^{(l)}$ and weight $w^{(l)}$ are INT8 numbers with scale factors $S_a^{(l)}$ and $S_w^{(l)}$. With INT8 matrix multiplication, which is used to replace traditionally FP32-based multiplication, we can obtain intermediate INT32 activation $a_{32}^{(l+1)}$. To transform the intermediate results into INT8 numbers, an INT32-to-INT8 algorithm is needed, such as Round and Shift [74], Loss-aware Compensation [91], or Requantize [13]. When forwarding to the final layer, the activations are input to the softmax and loss layer.

Backward pass and weight update. The obtained INT8 error gradients $e^{(l+1)}$ will multiply with $w^{(l)}$ and $a^{(l)}$ to obtain intermediate INT32 error gradients $e_{32}^{(l)}$ and weight gradients $g_{32}^{(l)}$. Through the INT32 to INT8 algorithm, we can also get the error and weight gradients of the l -th layer $e^{(l)}$ and $g^{(l)}$ with their scale factor $S_e^{(l)}$ and $S_g^{(l)}$. Finally, the model weights are updated by gradient $g^{(l)}$ with the global learning rate and other hyperparameters. The update method can be divided into two categories – FLOAT update and INT8 update – which means the former can support changing scale factors.

2.3 Mobile DSP Offloading

DSP was originally designed for processing digital signals like audio with high energy efficiency. Almost every mobile SoC includes DSP, and the most common is Qualcomm's Hexagon [8]. In 2016, Qualcomm announced the Hexagon 680 DSP – the first DSP with Hexagon Vector Extensions (HVX) designed to allow significant

compute workloads for advanced image processing and computer vision [4]. Given its popularity, our work utilizes Hexagon DSP to run the DNN training workloads, but the techniques are compatible with other DSP hardwares as well.

Hexagon DSP architecture. Nowadays, the Hexagon DSP contains hexagon cores and a SIMD co-processor. The former performs general-purpose processing while the latter is good at vector computation [7]. *Mandhel* mainly exploits the SIMD co-processor, which can process 1024-bit fixed-point data inside one HVX instruction, or 128 INT8 mathematical functions like add and multiply in one cycle. Besides, the hexagon core's clock frequency is 500 MHz, which is much lower than the CPU ones, so it is much more energy friendly. However, its hexagon cores are too weak to perform heavy general processing and its SIMD co-processor does not include float processing units to perform FP32/FP16 operations. Therefore, we need to carefully design the training operations on the Hexagon DSP to gain the expected benefits.

Hexagon DSP programming model. The Hexagon DSP and CPU cores share the main memory, but do not share the cache. The two processors have their own memory space, indicating that data needs to be copied between them. Therefore, a typical program contains two parts: the application logic code running on the CPU and the data processing code running on DSP. DSP code is dynamically loaded on invocation of a synchronous Remote Procedure Call (FastRPC) [7].

3 THE DESIGN

3.1 Overview

Design goal. *Mandhel* aims to minimize latency and energy consumption during a given training task through on-device DSP offloading. *Mandhel* is designed as a generic framework to support different kinds of mixed-precision algorithms and allows users to customize the algorithms through its exposed abstraction and configurations, as we will discuss in §3.2. Convergence accuracy is guaranteed by the used mixed-precision training algorithm.

Workflow. Figure 3 illustrates the workflow of *Mandhel*. The input of *Mandhel* includes the mixed-precision training algorithms and the model file which can be either pre-trained or randomly initialized. Once *Mandhel* is deployed on a device, it works in two stages. (1) In the preparation stage, *Mandhel* translates models from different front-end frameworks (e.g., TensorFlow [13] and PyTorch [59]) to intermediate models in the form of a FlatBuffer-format model file. (2) In the execution stage, *Mandhel* generates CPU and DSP compute subgraphs and performs compute-subgraph execution on Android devices. Note that both stages run on devices, and the preparation stage is automatically triggered before the first-time execution of one shot. Hence, such a design does not introduce any additional programming efforts for app developers.

• **Preparation stage.** When a to-be-trained model is downloaded to a device, *Mandhel* translates the model to an intermediate model via the *Intermediate Model Builder* according to the mixed-precision training configuration. The intermediate model contains the operator type, hyperparameters, inputs and outputs as well as the memory regions of intermediate model inputs and outputs. Then, *Mandhel* runs a profiling iteration to obtain each layer's

optimal configuration via the *Split Batch Profiler* to further optimize the intermediate model and gain higher performance, as we will show in §3.5. Lastly, a FlatBuffer-format model file will be generated to be processed directly by the *Mandhel* runtime.

- **Execution stage.** Once a training task starts, the *Mandhel* runtime first loads datasets and weights from the disk to the required memory regions, and then generates CPU and DSP compute subgraphs via the *CPU-DSP co-scheduling controller* (§3.3). All subgraphs will execute on Android devices' CPU or DSP via *Mandhel*'s *Hexagon DSP and CPU Training backend*, which include operator implementation optimized for mixed-precision data types on CPU and DSP.

Key techniques. While DSP has proven useful in DNN inference [48], we observe a disparity between using DSP to serve training and inference tasks. Compared to the CPU, a vanilla DSP training engine can achieve very limited or even negative performance gain. To this end, *Mandhel* incorporates four key techniques to augment DSP's computing capacity. The techniques can be divided into two major classes: *intra-operator* and *inter-operator*. At the intra-op level, since convolution and weight-gradient layers would result in excessive memory access and exhaust the DSP cache, we use self-adaptive rescaling (§3.4) and batch splitting (§3.5) to reduce memory access and increase the cache hit ratio. At the inter-op level, *Mandhel* overlaps CPU and DSP execution as much as possible to mitigate the overhead of DSP-unfriendly operators (§3.3) and reuses the DSP compute graph to eliminate the preparation overhead (§3.6).

3.2 Mixed-Precision Training Abstraction

We thoroughly investigate the typical mixed-precision training algorithms [21, 36, 50, 51, 62, 74, 78, 91–93] and summarize how they manipulate data in Table 1. We extract the basic concepts from those algorithms and identify four key elements that jointly define a mixed-precision training algorithm.

- *Translation from FP32 operators to mixed-precision operators.* To support end-to-end training with mixed precision, a normal FP32 operator often needs to be translated into a combination of new operators that operate on data with different kinds of precision. Such a translation is elaborately designed by algorithm developers to ensure good accuracy. Note that *Mandhel* has implemented lots of mixed-precision operators internally.
- *Backpropagation rules* are used to illustrate how to calculate operators' weight and error gradients. For instance, the NITI algorithm uses INT8 deconvolution to calculate the FP32 convolution error gradients.
- *Weight information* includes weight type, weight initializing methods, and weight update algorithms.
- *Optimizer information* includes the loss function (e.g., cross-entropy) and the optimizer, like SGD and ADAM.

Table 2 gives an example of how to use the above abstraction to specify the NITI algorithm. For example, an FP32 convolution needs to be translated into: (1) an INT8-based convolution; (2) a Max operation to obtain the scale factor; (3) a Shift operation to convert the INT32 value to INT8. Such a configuration will

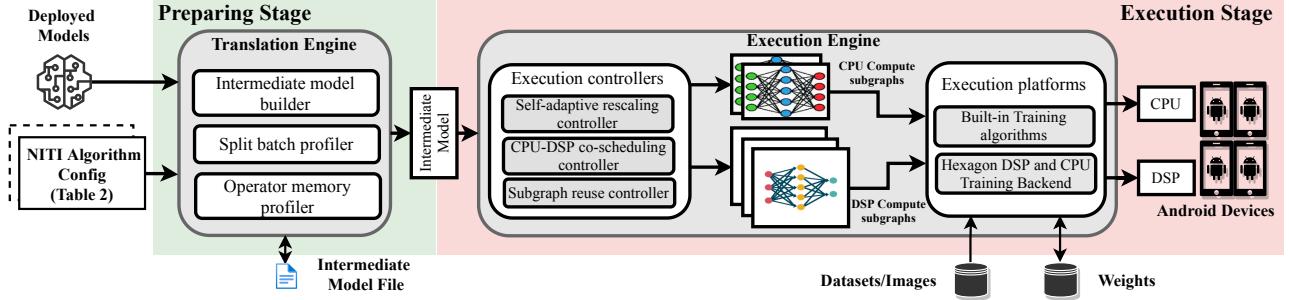


Figure 3: The overall workflow of Mandheling.

Mixed-precision algo.	W	A	G	WU	support
NITI [74]	INT8	INT8	INT8	INT8	✓
Octo [91]	INT8	INT8	INT8	INT8	✓
Adaptive Fixed-Point [88]	INT8/INT16	INT8	INT8	FP32	✓
WAGEUBN [83]	INT8	INT8	INT8	FP24	✓
MLS Format [90]	INT8	INT8	INT8	FP32	✓
Chunk-based [75]	FP8	FP8	FP8	FP16	✗
Unified INT8 Training[95]	INT8	INT8	INT8	FP32	✗

"W", "A", "G", and "WU" represent weight, activation, gradient and weight update.

Table 1: Data types of mixed-precision training algorithms.

	Op	CPU	DSP
Latency compare	Transpose	3 ms	25 ms
	WeightRotate	4 ms	20 ms
	Slice	4 ms	17 ms
DSP unsupported ops	Normalization, Quantization, Round, Sqrt, etc.		

Table 3: Operators that do not fit DSP.

be input to Mandheling along with the model to be trained. Currently, Mandheling has already incorporated many built-in mixed-precision training algorithms, i.e., 5 out of 7 in Table 1. The other two are currently not yet supported due to the lack of support for certain low-precision operators such as FP8-based convolution, and will be considered in future work.

3.3 CPU-DSP Co-Scheduling

DSP-unfriendly operators. Though DSP is adequate for INT8 vector arithmetic operations, there are still some irregular memory access operations or float operations that are unsuitable for DSP to run, referred to as “DSP-unfriendly operators” in this work. As shown in Table 3, some operators’ latency on DSP is more than 8× slower than on CPU, and some FP32-only operators, such as Normalization and Quantization, lack architecture-level support on DSP [74, 83]. Therefore, they need to be executed on the CPU.

To partition the model across DSP and CPU, an intuitive approach is to merge the adjacent DSP-unfriendly operators into subgraphs and place them on the CPU. However, since CPU-DSP context switching incurs a high overhead, mainly due to the duplicate data between their memory space (i.e., around 25ms on the Xiaomi 10), this approach could lead to non-optimal performance. Conceptually, some DSP-friendly operators should be placed on the CPU as well to reduce the CPU-DSP context-switching frequency. Therefore, we need a context-switching-aware scheduling strategy that wisely maps the operators to CPU and DSP.

Attribute	Contents	
	key	value
Translation	FP32 Conv	INT8 Conv+ReduceMax+Shift
	FP32 MaxPool	INT8 MaxPool
Backprop.	FP32 Conv Error Grad.	INT8 Deconv
	FP32 Conv Weight Grad.	INT8 ConvBackpropFilter
Weight	Initializer	Xavier_normal
	Type	INT8
Optimizer	Update	INT8
	Loss	Cross-Entropy
	Optimizer	SGD

Table 2: A typical NITI algorithm training config.

Operator-to-hardware scheduling. To solve the scheduling problem, Mandheling first uses topological sort to get an execution order for all operators and profiles to obtain each operator’s latency on CPU and DSP. Then, it uses a dynamic programming algorithm to find the approximately optimal scheduling solution while obeying the execution order.

$$T[i + 1, \text{CPU}] = \min \begin{cases} T[i, \text{CPU}] + L_{i+1}^{\text{CPU}} \\ T[i, \text{DSP}] + L_{i+1}^{\text{CPU}} + L_{\text{switch}} \end{cases} \quad (1)$$

$$T[i + 1, \text{DSP}] = \min \begin{cases} T[i, \text{CPU}] + L_{i+1}^{\text{DSP}} + L_{\text{switch}} \\ T[i, \text{DSP}] + L_{i+1}^{\text{DSP}} \end{cases} \quad (2)$$

$T[i + 1, \text{CPU}]$ is the lowest latency of finishing O_{i+1} if O_{i+1} is running on CPU. L_{i+1}^{CPU} means O_{i+1} running on CPU and L_{switch} represents the latency of context switching. The init state is set to $T[1, \text{CPU}] = L_1^{\text{CPU}}$ and $T[1, \text{DSP}] = L_1^{\text{DSP}}$. When both O_{i+1} and O_i are running on CPU, there is **no context switching**. Otherwise, a context-switching overhead is added. $T[i + 1, \text{CPU}]$ is the minimum value of the above two. Similarly, the latency of finishing O_{i+1} running on DSP also has two circumstances, as shown in Eq 2. The objective T_{model} can be formulated as

$$T_{\text{model}} = \min\{T[N, \text{CPU}], T[N, \text{DSP}]\}. \quad (3)$$

N is the number of operators. Based on the recursion formula and the objective, we can find the optimal scheduling plan.

Noticeably, our co-scheduling algorithm is designed to reduce the context-switching overhead of DSP-unfriendly operators rather than to provide maximal CPU-DSP parallelism. In fact, since DSP is generally 3–5× more energy efficient than CPU, allocating more computations to CPUs and parallelizing them with DSP can only potentially bring a modest improvement in speed but with huge energy consumption.

3.4 Self-Adaptive Rescaling

```

1 int scale = 0;
2 /* Calculate INT32 temporal results */
3 for(int i = 0; i < length; i++) {
4     Tensor x = input[i];
5     Tensor w = weight[i];
6     // CONV or matrix multiply
7     Tensor temp_result = x * w;
8     // count leading zero
9     Tensor clz = clz(temp_result);
10    int tscale = 32 - max(clz) - 7;
11    scale = scale > tscale ? scale : tscale;
12    temp_output[i] = temp_result;
13 }
14 /* Cast the INT32 to INT8 values */
15 for(int i = 0; i < length; i++) {
16     Tensor temp = temp_output[i];
17     // Downscale
18     Tensor int8_result = temp / scale;
19     result[i] = int8_result;
20 }

```

Listing 1: Key C code snippet of dynamic rescaling

```

scale = 0

loop0:
v0 = vmem ptr_i
v1 = vmem ptr_w

v2 = vmpy v0, v1

v3 = vclz v2
tscale = vmax v3
scale = mux scale >
tscale, scale,
tscale

vmem ptr_t, v2
end loop0
loop1:
v0 = vmem ptr_t
v3 = vmpye v0, scale
vmem ptr_v, v3
end loop1

```

Listing 2: Asm code version

Static scaling vs. dynamic rescaling. Once a quantized model is deployed for inference, the scale factor (i.e., $S_a^{(l)}$ and $S_w^{(l)}$ in Figure 2) per layer is a static value. Therefore, the data flow is simple as it just needs to multiply the matrix after loading the input and weight and store the scaled result. During training, however, the scale factor also needs to be dynamically adapted, just like the trainable weights. An unreasonable scale factor can noticeably lower model accuracy and the optimal scale factor cannot be known prior to training completion.

Mismatch of dynamic rescaling and DSP. Such dynamic scaling runs slow on DSP due to its excessive memory access. For each batch of training, we have to store the temporal outputs and reload them after obtaining the scale factor to downscale the temporal outputs to final INT8 results, as shown in Listings 1 and 2. Note that each layer with trainable weights collocates with a scale factor. Therefore, there can be at most hundreds of dynamic scale factors in a typical DNN model. As we have measured, dynamic rescaling will add at least $2\times$ latency compared with static rescaling.

Insights & Opportunities. Fortunately, we observed two useful patterns of how scale factors change during training. Figure 4 illustrates (a) the concrete value of a scale factor of one layer and (b) its changing frequency in training the VGG11 model on the CIFAR-10 dataset. We find that after the initialization period: (1) the scale factor jumps between two alternative values – i.e., 10 and 11 – and using either of them does not affect model accuracy; and (2) the actual changing frequency of the scale factor is low, e.g., per 10–60 batches. Those two phenomena are common in training because when the model approaches convergence, the gradients decrease and the scale factor is less likely to be changed.

Based on the above observations, we propose a self-adaptive rescaling technique to mitigate the overhead of rescaling. Its key idea is to periodically enable rescaling, instead of per batch. The rescaling frequency is adaptively configured based on the observed history of how frequently the scale factors change after training, e.g., last K batches with rescaling enabled. We heuristically map the observed frequency of changed scale factor f to the periodic

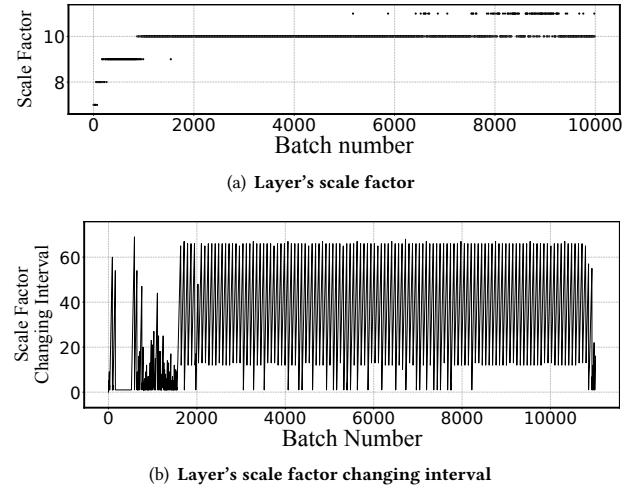


Figure 4: The scale factor and its changing interval of the first CONV layer in training the VGG11 model (batch size = 64) on the CIFAR-10 dataset.

Input Size	latencies of different batch sizes (ms)					
	2	4	8	16	32	64
8×8	0.63	0.63	0.85	1.03	1.27	1.84
16×16	0.84	0.89	4.23	3.98	4.64	12.24
32×32	1.69	2.50	59.11	62.35	68.13	152.89

Table 4: The latency of a convolution layer on DSP with different batch/input sizes (input channel = 64, output channel = 64).

frequency in calculating the new scale factor $f/2$. Our experiments in §4 show that such a heuristic policy works well for different kinds of models (VGG, ResNet, Inception, etc.) and datasets (CIFAR10, FEMNIST), reducing the training time by 10.8%–65.4% with negligible accuracy degradation (up to 0.3%).

3.5 Batch Splitting

Exhausted data cache. A well-known factor for fast on-device inference is full utilization of the processor cache [73]. For inference tasks, using a large batch is very likely to improve CPU utilization and, therefore, the processing throughput [25, 68]. However, for DSP-based training tasks, we observe a huge performance decline for large operator and batch sizes, especially for weight-gradient calculation. Table 4 illustrates this phenomenon with a convolution operator on the Xiaomi 10 (Snapdragon 865 SoC). When the input data size is 32×32, the latency of batch size = 32 is 27.3× more than that of batch size = 4, which means 8× theoretical workload incurs a 27.3× delay².

The performance drop comes from the exhausted DSP cache. There are two reasons for the disparate behavior in training and inference tasks. First, training tasks often require a large batch size, which leads to numerous amounts of memory access for intermediate data. Second, the DSP cache is often smaller than the CPU cache, e.g., only half for the L2 cache on the Snapdragon 865 (1 MB vs. 2 MB).

²When batch size > 4, the actual batch size will be padded to a multiple of 32 as required by the Hexagon NN, so the latency of batch size = 8 or 16 will be similar to that of batch size = 32.

To reach a high cache hit ratio, we need to partition the intra-operator workloads. We choose to split the operation at the batch dimension, i.e., the first dimension of input data, as it is simpler to implement and causes no redundant computations. We refer to an operator behaving “abnormally” if its latency-to-workload (in FLOPs) ratio is noticeably higher than the same configuration but with a smaller batch size. Through offline profiling, as in Table 4, Mandheling can identify all abnormal operators in a model and split them into normal ones.

To ensure efficient locality, Mandheling splits an abnormal batch into multiple micro-batches and executes them individually. The final weight gradients are the accumulation of their output. The accumulation formula for FP32 operations is $W^g = \sum_{i=1}^n W_{batch_i}^g$. When it comes to INT8, the formula will be changed into $W^g * S^g = \sum_{i=1}^n W_{batch_i}^g * S_{batch_i}^g$. This formula can finally be transformed to

$$W^g = \sum_{i=1}^n W_{batch_i}^g * S_{batch_i}^g / S^g \quad (4)$$

According to Eq 4, if $S_{batch_i}^g / S^g = 1$, we can avoid the FP32 add operation. Referring to the non-split gradient algorithm, we can know that $S^g = \max\{S_{batch_i}^g, i \in split_num\}$. Therefore, all the temporary micro-batch results should rescale from $S_{batch_i}^g$ to S^g . Our experiments show that when splitting the batch, in most cases, $S_{batch_i}^g$ is the same as S^g , so rescaling will not compromise the benefits of batch splitting.

3.6 Compute Subgraph Reuse

Costly compute-graph preparation. Our experiments also show that preparing the compute graph takes a considerable amount of time, e.g., 304ms on TFLite and 212ms on MNN for the VGG16 [67] model, respectively. Preparation includes the following steps. First, the training engine needs to build the DSP compute subgraph, which consists of operators with inputs, outputs, and parameters. To ensure that the subgraph can execute correctly, the engine also maintains a reference relationship graph between operators. Before invoking the DSP training, the memory space needs to be allocated on DSP as well. The current on-device training engines always prepare a new compute subgraph for each batch of training. The dynamic graphs are rather easy for developers to debug and develop, but they unfortunately incur high overheads on resource-constrained devices, which are not suitable for on-device training.

Since the models are rarely modified during on-device training, we propose to reuse DSP compute subgraphs to eliminate their preparation overheads. However, directly reusing subgraphs can easily exceed the DSP memory budget since substantial memory regions cannot be released. To this end, Mandheling seeks to minimize the memory allocation/deallocation operations under the memory constraint – a common memory management problem in the operating system [15, 60, 71, 89]. An opportunity is that, unlike OS, Mandheling’s memory allocation/deallocation for subgraph reuse always follows a DNN execution order, so the most recently used (MRU) memory region has the longest reuse distance. Therefore, the key idea is to release the MRU memory regions which best fit memory needs.

- In the preparation stage, Mandheling profiles all memory regions used by compute subgraphs, as shown in Figure 3. Since the number

Devices	CPU	GPU	DSP
Xiaomi 11 Pro Snapdragon 888	2.84GHz Cortex-X1 3x 2.4GHz Cortex A78 4x 1.8GHz Cortex A55	Adreno 660 GPU 700MHz	Hexagon 780 DSP 500MHz
Xiaomi 10 Snapdragon 865	2.84GHz A77 3x 2.4GHz Cortex A77 4x 1.8GHz Cortex A55	Adreno 650 GPU 587MHz	Hexagon 698 DSP 500MHz
Redmi Note9 Pro Snapdragon 750G	2x 2.2GHz Cortex A77 6x 1.8GHz Cortex A55	Adreno 619 GPU 950MHz	Hexagon 694 DSP 500MHz

Table 5: Devices used in the experiments.

Model	Input Data	FLOPs	# of CONVs
VGG-11 [67]	CIFAR-10	914 M	8
VGG-16 [67]	CIFAR-10	1.35 G	13
VGG-19 [67]	ImageNet	26.92 G	16
ResNet-34 [34]	CIFAR-10	7.26 G	36
ResNet-18 [34]	ImageNet	11.66 G	20
InceptionV3 [69]	CIFAR-10	2.43 G	16

Table 6: DNN models used in the experiments.

of compute subgraphs is rather small (<100), we can exhaustively explore all circumstances to find all possible solutions satisfying different memory size requirements.

- When the system is about to exceed the memory budget, Mandheling releases the MRU memory regions identified and marked in the preparation stage and allocates space for subgraphs that are yet to come.

4 IMPLEMENTATION AND EVALUATION

We have fully implemented Mandheling with 15k LoC in C/C++ and 800 LoC of assembly language in total. The prototype is a standalone framework supporting models exported from MNN [39], TFLite [52], and Pytorch Mobile [59]. Mandheling leverages the Hexagon NN [3] as the DSP backend, which is the only open-source library supporting inference on DSP, developed by Qualcomm. While Mandheling has supported many different mixed-precision training algorithms, as shown in Table 2, we use NITI [74] as the default in our experiments because it extensively uses INT8 operations that are quite suitable for DSP. Since the Hexagon DSP architecture is constantly changing, we mainly optimize our implementation for the V66 architecture using assembly code. The prototype reuses FP32-based operators running on the CPU from MNN.

4.1 Experimental Methodology

Hardware setup. We test the performance of Mandheling on three smartphones with different Qualcomm SoCs: the Xiaomi 11 Pro (Snapdragon 888), the Xiaomi 10 (Snapdragon 865), and the Redmi Note9 Pro (Snapdragon 750G). The Xiaomi 11 Pro device is equipped with the latest Hexagon 780 DSP, which is said to have huge performance improvements over previous ones. The hardware details are shown in Table 5. All devices run Android OS 10. By default, we always run the baselines on 4 BIG CPU cores and Mandheling’s CPU workloads on 2 BIG cores. The CPU frequency is controlled by the OS’s dynamic voltage and frequency scaling (DVFS) controller. **Models.** We test with a range of typical CNN models with various input sizes: VGG11/16/19 [67], ResNet18/34 [34], and InceptionV3 [69], as listed in Table 6. The input data for those models

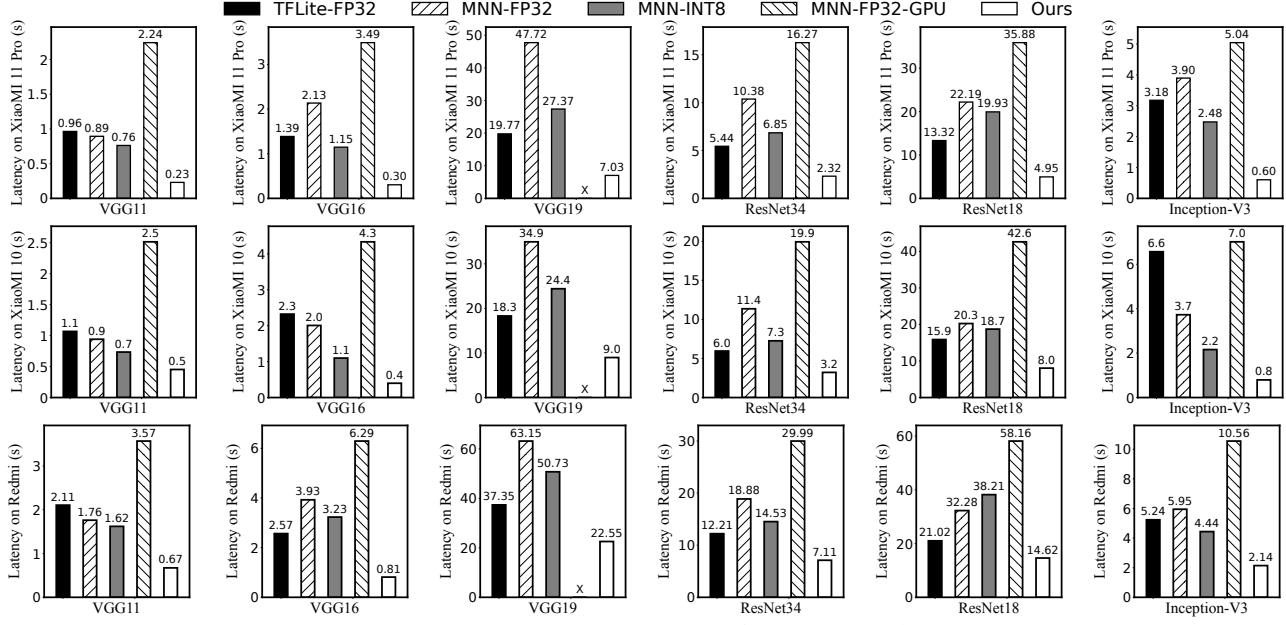


Figure 5: Per-batch training time on different models (batch size = 64) on different devices.

are either CIFAR-10 [44] (input size 32x32) or ImageNet [45] (input size 224x224).

Baselines. We mainly compare Mandheling with two frameworks. One is MNN [39], which is one of the earliest frameworks that has supported on-device training since late 2019. Note that Mandheling also reuses some operator implementation from MNN. The other one is TFLite [52], which is the most popular DL framework on smartphones and recently added training support in November 2021. Both MNN and TFLite only support training in FP32 format. To make a fairer comparison, we also extend MNN to train with the INT8 format using the same training algorithm as Mandheling on CPU. More specifically, we compare Mandheling with four baselines: (1) TFLite-FP32: the traditional FP32-based training method provided by TFLite; (2) MNN-FP32: the traditional FP32-based training method provided by MNN; (3) MNN-INT8: the INT8-based training method implemented by us based on MNN (NEON [64] is extensively used to optimize the performance of this baseline); and (4) MNN-FP32-GPU: FP32-based training on mobile GPU through the OpenCL backend.

We do not compare Mandheling with NPU approaches. That is because, to our best knowledge, existing mobile NPUs are designed for inference rather than training. Though MNN [39] and TNN [11] support inference on NPUs, there has been no NPU training support and open-source operator-level NPU implementation for on-device learning until now.

Metrics and configurations. We mainly measure training time and energy consumption during training. Energy consumption is calculated through Android's vFS (/sys/class/power_supply) by profiling every 100ms. In addition, we also evaluate thermal and CPU frequency through /sys/class/thermal/thermal_zone and /sys/devices/system/cpu/cpufreq to show our power efficiency over the long duration of intensive computation. All experiments are repeated three times and we report the average numbers.

4.2 Per-Batch Performance

Overall performance. We first comprehensively investigate the per-batch training performance of Mandheling with batch size 64. The latency and energy consumption results are illustrated in Figures 5 and 6, respectively. Our key observation is that Mandheling *consistently and remarkably outperforms other baselines on both metrics*.

- *Training latency of Mandheling vs. FP32 baselines.* Compared with MNN-FP32 and TFLite-FP32, Mandheling achieves a 2.08-7.1× and 1.44-8.25× speedup in per-batch training time, respectively. Comparing different devices, we observe that Mandheling's improvements are relatively less profound on the Redmi Note9 Pro than on the other two devices. This is because the Redmi Note9 Pro is equipped with an outdated SoC where the performance gap between CPU and DSP is much smaller than the other two high-end SoCs.

- *Energy consumption of Mandheling vs. FP32 baselines.* As Figure 6 shows, Mandheling's improvements in energy consumption are even more profound than in training speed. Specifically, Mandheling reduces energy consumption by 3.21-11.2× and 2.01-12.5× compared with MNN-FP32 and TFLite-FP32, respectively. Such a huge benefit comes from both the training speedup and the higher power efficiency of DSP.

- *Mandheling vs. MNN-FP32-GPU.* Mandheling can reduce latency 3.98-11.63× and energy consumption 3.46-10.95× compared with MNN-FP32-GPU. The reason for such a huge improvement is that (1) as far as we know, MNN is the only framework supporting on-device GPU training and is not fully optimized yet (GPU utilization during training is only around 30-50%); and (2) DSP is more power-efficient than GPU. Besides, training the VGG19 model on ImageNet with MNN-FP32-GPU encounters out-of-memory failure.

- *Mandheling vs. MNN-INT8.* According to Figures 5 and Figure 6, Mandheling can reduce latency by up to 4.13× and energy consumption by 6.5× compared to MNN-INT8, respectively. Since both

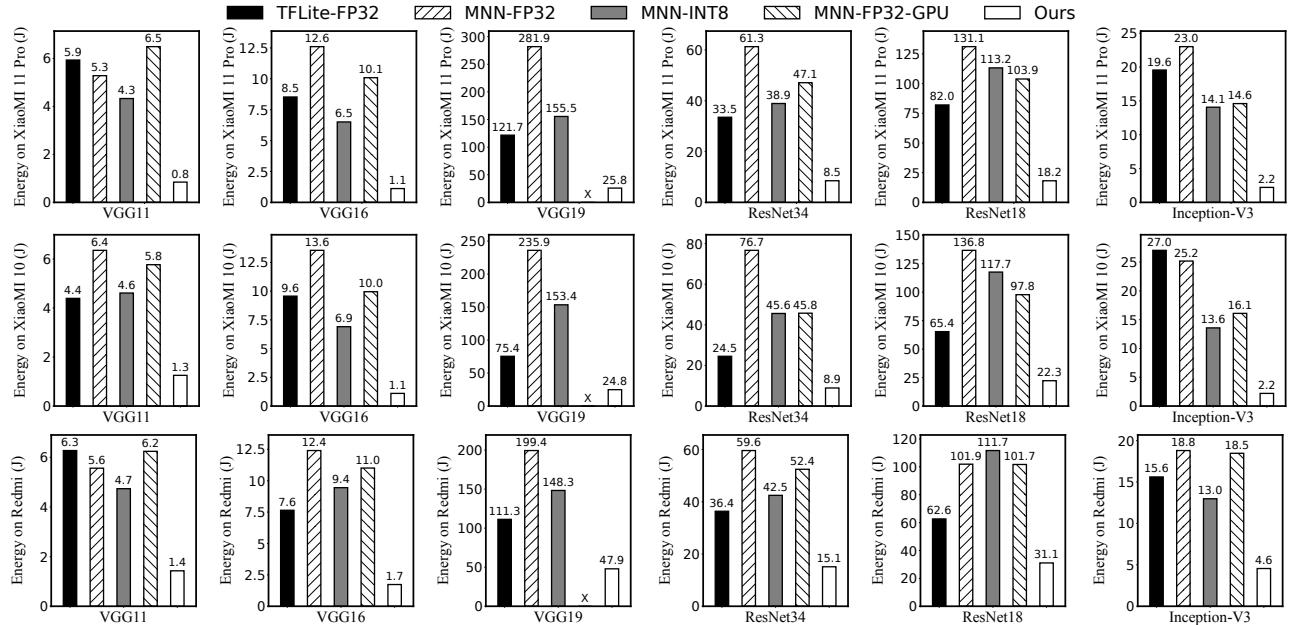


Figure 6: Per-batch energy consumption of different models (batch size = 64) on different devices.

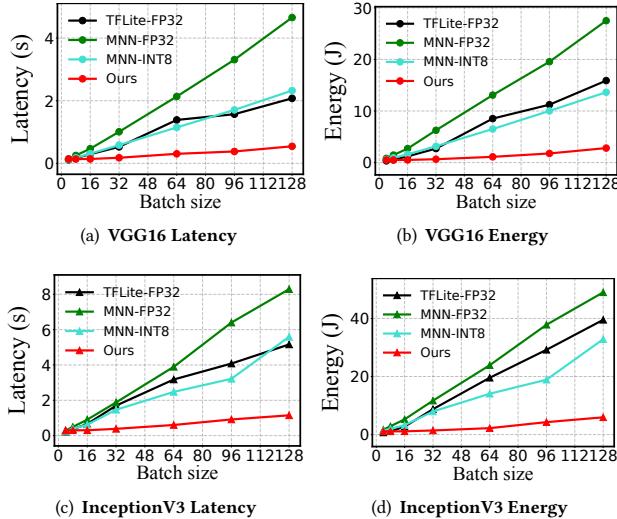


Figure 7: Per-batch training time and energy consumption under various batch sizes.

of them use the same mixed-precision training algorithm, the improvements come from Mandheling's ability to fully utilize the DSP hardware. Note that the DSP HVX vector instruction can calculate up to 128 INT8 arithmetic operations, while CPU NEON can only perform four operations.

Impacts of batch size. We then evaluate Mandheling with various batch sizes from 4 to 128 with the VGG16 and InceptionV3 models on the Xiaomi 11 Pro. As shown in Figure 7, Mandheling consistently outperforms other baselines on each batch size, e.g., 4.81× lower latency and 6.90× lower energy consumption on average. Besides, the performance gap between Mandheling and baselines is bigger as batch sizes increase. For instance, Mandheling reduces latency by up to 8.56× and energy consumption by 12.67× for batch size 128 thanks to the batch-splitting technique (§3.5).

Method	CPU Conf.	Time (s)		Energy (J)	
		H	L	H	L
MNN-FP32	BIG 1×	4.88	10.81	11.05	6.90
	BIG 2×	2.87	5.89	12.85	7.50
	BIG 4× (default)	2.14	3.86	11.90	7.10
	LITTLE 1×	25.20	42.57	13.35	21.95
	LITTLE 4×	10.75	17.06	14.85	9.55
	Hybrid 8×	4.50	7.58	25.40	12.45
MNN-INT8	BIG 1×	5.59	14.75	9.85	4.35
	BIG 2×	3.07	7.37	10.60	5.25
	BIG 4× (default)	1.79	3.8	7.90	4.55
	LITTLE 1×	40.13	81.20	12.65	25.85
	LITTLE 4×	10.36	20.64	10.35	5.95
	Hybrid 8×	2.71	4.96	12.10	6.20
Ours	BIG 1×	0.29	0.41	0.70	0.65
	BIG 2× (default)	0.25	0.32	0.90	0.70
	BIG 4×	0.24	0.31	1.20	0.90
	LITTLE 1×	0.79	1.61	0.80	1.15
	LITTLE 4×	0.49	0.68	0.90	0.75
	Hybrid 8×	0.34	0.39	1.60	0.90

Table 7: The performance impacts of the selection of CPU cores and their frequency (H/L: highest/lowest frequency available). Highlighted numbers indicate the best performance or least energy consumption.

Impacts of CPU cores. Since most baselines and part of Mandheling run on CPUs, it's intuitive to test how the choice of CPU cores affects their performance. In this experiment, we use the VGG16 model with batch size 64. We vary the CPU core numbers and the frequency for each core on the Xiaomi 11 Pro. The results are summarized in Table 7.

As observed, the choice of CPU cores opens rich tradeoffs for Mandheling and baselines between training speed and energy consumption. Running on 4 BIG CPU cores with the highest frequency enables Mandheling to train with the fastest speed (0.24s per batch), yet its energy consumption is 1.9× higher than 1 BIG CPU core with the lowest frequency. The default configuration for Mandheling is to use 2 BIG CPU cores with DVFS to balance the two key metrics. In reality, a developer or the OS might control the CPU cores to harness such a tradeoff. For instance, on a low-power device, the OS

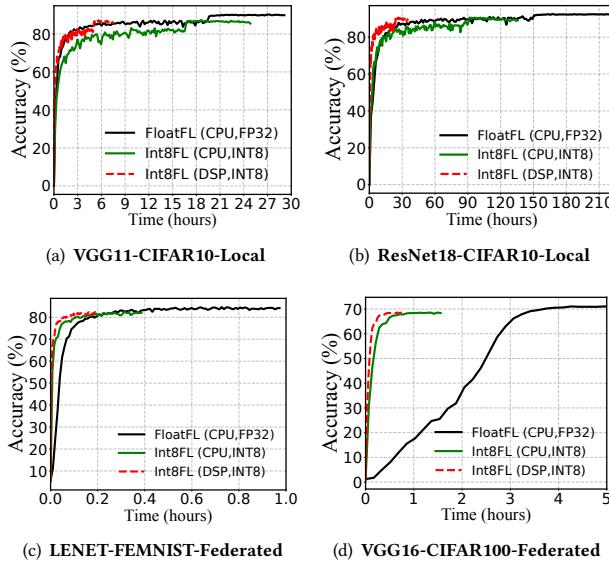


Figure 8: Convergence accuracy across clock time under single device and federated scenarios.

Dataset	Model	Methods	Acc.	Training Cost to Convergence		
				Round number	Clock Hours	Energy (WH)
Centralized CIFAR-10	VGG11	MNN-FP32	89.87%	150	29.13	187.01
		MNN-INT8	87.17%	150	24.77	153.33
		Ours	87.17%	150	7.50	31.39
Centralized CIFAR-10	ResNet18	MNN-FP32	92.49%	150	223.55	1,435.19
		MNN-INT8	90.62%	150	135.71	840.04
		Ours	90.62%	150	35.68	149.32
Federated FEMNIST	LeNet	MNN-FP32	84.18%	990	0.97	0.00057
		MNN-INT8	82.04%	4,960	0.39	0.00029
		Ours	82.04%	4,960	0.19	0.00007
Federated CIFAR-100	VGG16	MNN-FP32	71.15%	1,960	8.35	2.74
		MNN-INT8	68.42%	2,200	1.56	1.26
		Ours	68.42%	2,200	0.78	0.21

Table 8: A summary of end-to-end training costs up to convergence under different training scenarios.

might instruct Mandheling to use only one BIG CPU core for training. We should note that Mandheling still significantly outperforms other baselines with any settings.

4.3 End-to-End Convergence

We now demonstrate that Mandheling can significantly accelerate model convergence while guaranteeing model accuracy in end-to-end training experiments. We focus on the time-to-accuracy metric [74, 83, 88, 90, 95].

Learning on a single device is the case when all training data resides in a single device. We train the VGG16 and the ResNet18 with training set CIFAR-10 on the Xiaomi 11 Pro and verify the accuracy after each epoch. We fix the CPU frequency to the max value and 10 minutes of sleep after training 10 epochs to avoid shutdown due to overheating. Note that sleep time does not count in the time-to-accuracy metric.

As illustrated in Figures 8(a) and (b) and summarized in the first two rows of Table 8, Mandheling's convergence accuracy is only 1.9–2.7% lower than training with FP32. This accuracy drop is consistent with the numbers reported by the original algorithm paper [74] and

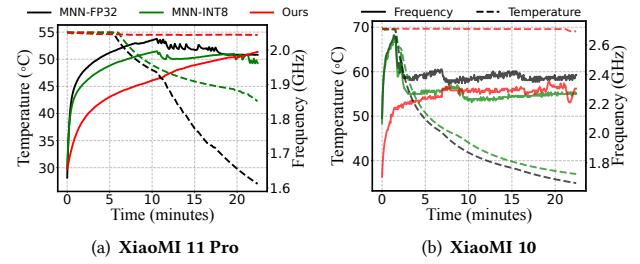


Figure 9: The temperature and CPU frequency dynamics during on-device training on different devices.

is generally accepted by the relevant ML community [14, 20, 21, 36, 50, 51, 62, 74, 75, 77, 78, 83, 88, 90–95]. However, it takes 5.06–6.27× less time and consumes 5.96–9.62× less energy for Mandheling to achieve 87.17% and 90.40% convergence accuracy compared with MNN-FP32. Compared to MNN-INT8, Mandheling converges to the same accuracy as they both use the NITI training algorithm, but it takes 3.55× less time and consumes 5.46× less energy on average. **Cross-device federated learning** is another killer use case that allows many devices to collaboratively train a model without giving away their training data. In our experiments, we use a popular FL simulation platform [82] and plug in the tested on-device training performance of Mandheling and baselines. Both FloatFL and Int8FL use the traditional FL protocol FedAvg. For a fair comparison, we set the number of the local epoch as 1 for all experiments. We use FEMNIST [63] and CIFAR-100 [44] as the testing datasets and follow prior work [33] to partition them into non-IID distribution.

As demonstrated in Figures 8(c) and (d) and the last three rows of Table 8, Mandheling's accuracy is 2.14% and 2.73% lower than FloatFL on FEMNIST and CIFAR-100, respectively. However, it only takes 19.58% and 9.3% of clock time to converge (i.e., 5.26× and 10.75× speedup) with Mandheling, respectively. Such tremendous improvement over the FloatFL protocol comes from both the reduced on-device training time and the reduced communication time for applying INT8-based training. Table 8 also shows that Mandheling reduces the energy consumption of a single client by 8.14× and 13.1×, respectively.

Thermal impacts. To reach a usable accuracy level, on-device training often takes a substantial amount of time [16], e.g., minutes for each round of federated learning or even hours for continuous local transfer learning [79]. Such a long duration of intensive computation may lead to thermal issues and, therefore, a CPU frequency change due to DVFS. Thus, we investigate the thermal dynamics of on-device training on two devices and illustrate the results in Figure 9. On both tested devices, we observe the temperature rising and the CPU underclocking, but the trend for Mandheling is much milder than for the other two baselines. Using MMN-FP32 and the Xiaomi 10 as an example, the device temperature rises sharply from 29°C to 68°C in 2 minutes, and the CPU frequency drops from 2.8GHz to 1.3GHz. On the other hand, it takes about 18 minutes for Mandheling to raise the temperature from 29°C to 58°C, which leads to almost no CPU underclocking. This is because DSP frequency is 4× lower than CPU frequency and is designed for low-power scenarios.

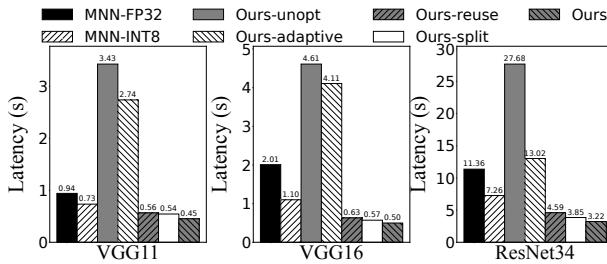


Figure 10: Ablation study of Mandheling.

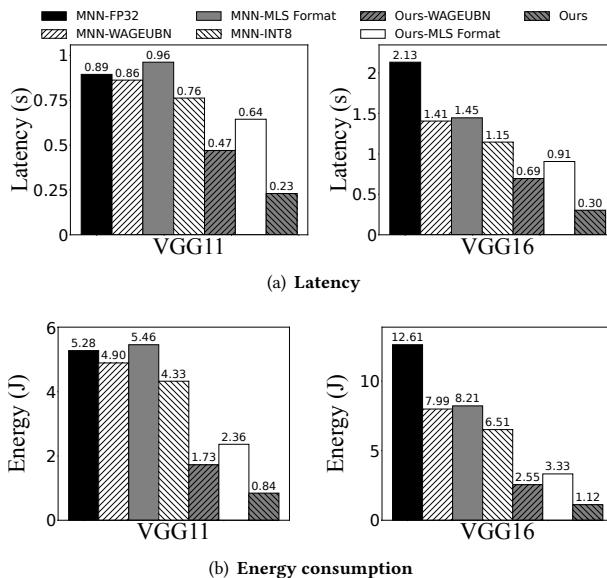


Figure 11: Mandheling's performance using different mixed-precision training algorithms. "MNN-**" are baselines running on CPU; while "Ours-**" are running on DSP using Mandheling.

4.4 Ablation Study

We further conduct a breakdown analysis of the benefit brought by each of Mandheling's techniques. The experiments are performed with the VGG11, VGG16 and ResNet34 models on the Xiaomi 10. The results are illustrated in Figure 10.

We observe that all techniques make a non-trivial contribution to the improvement. For instance, the per-batch training time for the ResNet34 model is 27.68s without any optimizations. When the self-adaptive rescaling is applied, the latency reduces to 13.02s. Adding the compute subgraph-reusing technique further decreases the latency to 4.59s. The other two techniques, batch splitting and CPU-DSP co-scheduling, also result in 22.9% and 19.5% lower latency, respectively. Since the four key techniques optimize the training cost from different aspects, they can be well orchestrated to provide accumulative optimization. Besides, the benefits of batch splitting are rather small for VGG11 compared to the other two models. That is because the workload of the VGG11 model is smaller, leading to the lack of high cache pressure that would motivate the use of the batch-splitting technique.

4.5 Fitting to Various Training Algorithms

Recall that Mandheling is an underlying independent framework that supports different kinds of mixed-precision training algorithms. Therefore, we also test Mandheling with different training algorithms: NITI [74] (default), MLS Format [90], and WAGEUBN [83]. The experiments are performed with VGG11 and VGG16 on the Xiaomi 11 Pro.

As shown in Figure 11, the training speeds of three mixed-precision algorithms are all faster than that of MNN-FP32 (3.08×, 2.34×, and 7.1× speedup). Mandheling's energy consumption improvement is more significant, i.e., 3.64× and 6.85× average reduction for VGG11 and VGG16, respectively. That is because Mandheling's key techniques aim to solve generic challenges to support different kinds of mixed-precision training algorithms. When comparing the same mixed-precision training algorithms running on CPU and DSP, Mandheling is still 2.04×, 1.59×, and 3.83× faster, respectively. Such benefits come from Mandheling's effective DSP offloading. Among the different training algorithms, NITI has the lowest training latency because it maximizes the usage of INT8 in its data flow.

5 DISCUSSION

Applicability and comparison to more hardware Mandheling currently leverages mobile DSP and CPU for training. We now discuss how Mandheling can incorporate more types of hardware.

NPUs and XPPUs are equipped on high-end mobile platforms nowadays and can deliver significantly higher computing capacity than general-purpose processors. Mandheling's abstraction of mixed-precision training algorithms can ease efforts to support training on such hardware. Moreover, its four key techniques are highly suitable for solving NPUs'/XPPUs' common performance issues. For instance, *CPU-XPU co-scheduling* is useful for tackling the challenge that XPPUs typically support only a small set of NN operators [37, 87]. Nevertheless, extra engineering effort should be put in to extend Mandheling for NPUs/XPPUs because they have their own architecture-level specific features that differ from DSPs. For example, Kirin's NPU supports hardware-level matrix multiplication operations, while DSP only supports vector multiplication. Fully optimizing operator implementation and improving memory access efficiency to unleash its matrix operation capability is crucial to improving its training performance. Offloading on-device training tasks on such hardware demands a new hardware-software co-design mixed-precision on-device training algorithm to improve training accuracy and efficiency.

Moreover, Mandheling currently does not leverage mobile GPU for training. This is because mobile GPU is not as efficient as CPU for on-device training, as shown in prior work [18, 87], not to mention DSP. The observation is also confirmed in our experiments: Figure 5 shows that the latency of MNN-FP32-GPU is 1.6–3.0× higher than that of MNN-FP32.

The practicality of Mandheling. Mandheling takes on-device training one step closer to real applications by significantly reducing its energy cost. Even with the performance boost of Mandheling, there may be concerns that it is still too heavy to train modern DNNs on mobile devices. We have the following comments. First, end-to-end training does not necessarily execute on one device. For

example, the model could be pre-trained on the cloud with public data and fine-tuned or personalized on each device [76, 79, 80]. Alternatively, in federated learning, the training cost is amortized across millions of devices [42, 43, 82]. Second, devices have plenty of idle time for local training. Previous research shows that a device has an average of 2.7 hours available for on-device training per day [79]. Meanwhile, training the Inception-V3 model locally on the Xiaomi 11 Pro with 20K personalized 32×32 labeled pictures takes only 31.25 seconds every day. The training process only consumes 6.36 mAh (0.13% battery) in total.

6 RELATED WORK

Mixed-precision DNN training. Recently, many mixed-precision DNN training algorithms have been proposed to reduce training costs [14, 20, 21, 36, 50, 51, 62, 74, 77, 78, 91–94]. The key idea is to replace the default numerical format of FP32 with lower precision for activations and/or weights, e.g., FP16, INT8, or even BOOL. Octo [91] further proposes an INT8-based training algorithm and also builds a system for edge GPUs. Instead of contributing new mixed-precision training algorithms, *Mandheling* is designed as a generic, underlying system to efficiently support those algorithms. **On-chip DNN offloading** has been studied to enable faster DNN inference on heterogeneous mobile processors like GPU and DSP. Most studies focus on how to partition and schedule the workloads on different processors, such as intra-layer [41, 85], inter-layer [27], block-layer level [29, 46], and model level [24, 48, 86]. However, they only focus on DNN inference. Besides ML tasks, GPU/DSP are also leveraged for signal, sound and image processing [23, 47, 61]. *Mandheling* is motivated by those efforts and is the first framework to support the offloading of training workloads to mobile DSP.

DNN inference optimizations. Besides on-chip DNN offloading, there have been many other research efforts towards improving DNN inference performance on devices. For instance, some apply structured pruning techniques to trade off latency and model accuracy [22, 30, 31]. Some improve low-level kernel implementation through a compiler, core scheduling, etc. [19, 53, 73]. Some of them optimize DNN inference in specific scenarios [35, 81, 84, 86]. *Mandheling* is inspired by those works, yet focuses on training instead of inference. As previously discussed, DNN training faces many unique challenges as compared to inference, so *Mandheling* contributes novel techniques in addressing those challenges.

Federated learning is an emerging machine-learning paradigm [43, 55, 57, 58] that is built atop on-device training and requires many clients to collaboratively train a DNN model. The communication bottleneck seriously affects system efficiency and model accuracy [40]. Therefore, prior work mostly focuses on the model compression technique [38, 49, 65, 66, 70] to reduce communication traffic. As an underlying framework, *Mandheling* is orthogonal to and compatible with those algorithm-level optimizations.

7 CONCLUSIONS

In this paper, we have proposed *Mandheling*, the first system that enables highly resource-efficient on-device training by orchestrating mixed-precision training with on-chip DSP offloading. *Mandheling* incorporates novel techniques such as self-adaptive rescaling and

CPU-DSP co-scheduling to fully unleash the power of DSP. We conducted extensive experiments to evaluate *Mandheling*.

8 ACKNOWLEDGEMENT

This work was partly supported by the National Key R&D Program of China under grant number 2020YFB1805500, the National Natural Science Foundation of China under the grant numbers 62172008 and 62102009, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and the PKU-Baidu Fund Project under the grant number 2020BD007. Mengwei Xu was partly supported by NSFC (No. 62102045), Beijing Nova Program (No.Z21110002121118), and Young Elite Scientists Sponsorship Program by CAST (No.2021QNRC001).

REFERENCES

- [1] Federated learning: Collaborative machine learning without centralized training data. <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>, 2017.
- [2] How apple personalizes siri without hoovering up your data. <https://www.technologyreview.com/2019/12/11/131629/apple-ai-personalizes-siri-federated-learning/>, 2019.
- [3] Qualcomm hexagon nn offload framework. https://source.codeaurora.org/quic/hexagon_nn/, 2020.
- [4] dsp-processor. <https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor>, 2021.
- [5] General data protection regulation. <https://gdpr-info.eu/>, 2021.
- [6] Genshin. <https://genshin.mihooyo.com/>, 2021.
- [7] /hexagon-dsp-sdk. <https://developer.qualcomm.com/software/hexagon-dsp-sdk>, 2021.
- [8] Qualcomm hexagon. https://en.wikipedia.org/wiki/Qualcomm_Hexagon, 2021.
- [9] Tensorflow graph reusing. <https://www.tensorflow.org/guide/function>, 2021.
- [10] Tiktok. <https://www.tiktok.com>, 2021.
- [11] Tnn. <https://github.com/Tencent/TNN>, 2021.
- [12] Youtube. <https://www.youtube.com>, 2021.
- [13] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.
- [14] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.
- [15] Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 202–209, 2007.
- [16] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dmitry Huba, Alex Ingberman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.
- [17] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of 19th International Conference on Computational Statistics*, pages 177–186. Springer, 2010.
- [18] Dongqi Cai, Qipeng Wang, Yuanqiang Liu, Yunxin Liu, Shangguang Wang, and Mengwei Xu. Towards ubiquitous learning: A first measurement of on-device training performance. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pages 31–36, 2021.
- [19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Liannmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. *TVM*: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 578–594, 2018.
- [20] Xi Chen, Xiaolin Hu, Hucheng Zhou, and Ningyi Xu. Fxpnet: Training a deep convolutional neural network in fixed-point representation. In *2017 International Joint Conference on Neural Networks*, pages 2494–2501. IEEE, 2017.
- [21] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.
- [22] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdrnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127, 2018.

- [23] Petko Georgiev, Nicholas D Lane, Cecilia Mascolo, and David Chu. Accelerating mobile audio sensing algorithms through on-chip gpu offloading. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 306–318, 2017.
- [24] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. Dsp-ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pages 295–309, 2014.
- [25] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
- [26] Wentian Guo, Yuchen Li, and Kian-Lee Tan. Exploiting reuse for gpu subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering*, 2020.
- [27] Donghee Ha, Mooseop Kim, KyeongDeok Moon, and Chi Yoon Jeong. Accelerating on-device learning with layer-wise processor selection method on unified memory. *IEEE Sensors*, 21(7):2364, 2021.
- [28] Myeonggyun Han and Woongki Baek. Herti: A reinforcement learning-augmented system for efficient real-time inference on heterogeneous embedded systems. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 90–102. IEEE, 2021.
- [29] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques*, pages 165–177. IEEE, 2019.
- [30] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y Chen. Legodnn: block-grained scaling of deep neural networks for mobile vision. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 406–419, 2021.
- [31] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdmn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136, 2016.
- [32] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.
- [33] Chaoyang He, Songze Li, Jinyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.
- [34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [35] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017.
- [36] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.
- [37] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 235–247, 2022.
- [38] Divyansh Jhunjhunwala, Advait Gadikar, Gauri Joshi, and Yonina C Eldar. Adaptive quantization of model updates for communication-efficient federated learning. In *2021 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3110–3114. IEEE, 2021.
- [39] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, et al. Mn: A universal and efficient inference engine. *arXiv preprint arXiv:2002.12418*, 2020.
- [40] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Ben{-}Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.
- [41] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μ layer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.
- [42] Jakub Konečný, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.
- [43] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.
- [44] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
- [45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.
- [46] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawzar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 1–12. IEEE, 2016.
- [47] Nicholas D Lane, Petko Georgiev, and Lorena 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, 2015.
- [48] Royson Lee, Stylianos I Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D Lane. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.
- [49] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: an efficient federated learning framework for heterogeneous mobile clients. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 420–437, 2021.
- [50] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. *Advances in neural information processing systems*, 30, 2017.
- [51] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.
- [52] TensorFlow Lite. Deploy machine learning models on mobile and iot devices, 2019.
- [53] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing cnn model inference on cpus. In *2019 USENIX Annual Technical Conference*, pages 1025–1040, 2019.
- [54] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. *arXiv preprint arXiv:1810.01875*, 2018.
- [55] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.
- [56] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys*, 49(2):1–35, 2016.
- [57] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 94–108, 2021.
- [58] Chaoyue Niu, Fan Wu, Shaohu Tang, Lifeng Hua, Rongfei Jia, Chengfei Lv, Zhihua Wu, and Guihai Chen. Billion-scale federated learning on mobile clients: A submodel design with tunable privacy. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.
- [59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.
- [60] Robert Pyka, Christoph Faßbach, Manish Verma, Heiko Falk, and Peter Marwedel. Operating system integrated energy aware scratchpad allocation strategies for multiprocessor applications. In *Proceedings of the 10th international workshop on Software & compilers for embedded systems*, pages 41–50, 2007.
- [61] Rajib Rana, Margee Hume, John Reilly, Raja Jurdak, and Jeffrey Soar. Opportunistic and context-aware affect sensing on smartphones. *IEEE Pervasive Computing*, 15(02):60–69, 2016.
- [62] Mohammad Rastegari, Vicente Ordóñez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.
- [63] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.
- [64] Venu Gopal Reddy. Neon technology introduction. *ARM Corporation*, 4(1):1–33, 2008.
- [65] Amirhossein Reisizadeh, Aryan Mokhtari, Hamed Hassani, Ali Jadbabaie, and Ramtin Pedarsani. Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization. In *International Conference on Artificial Intelligence and Statistics*, pages 2021–2031. PMLR, 2020.

- [66] Nir Shlezinger, Mingzhe Chen, Yonina C Eldar, H Vincent Poor, and Shuguang Cui. Federated learning with quantization constraints. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8851–8855. IEEE, 2020.
- [67] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [68] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.
- [69] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.
- [70] Hanlin Tang, Shaoduo Gan, Ce Zhang, Tong Zhang, and Ji Liu. Communication compression for decentralized training. *Advances in Neural Information Processing Systems*, 31, 2018.
- [71] Devesh Tiwari, Sanghoon Lee, James Tuck, and Yan Solihin. Mmt: Exploiting fine-grained parallelism in dynamic memory management. In *2010 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2010.
- [72] Haozhao Wang, Zhihao Qu, Qihua Zhou, Haobo Zhang, Boyuan Luo, Wenchao Xu, Song Guo, and Ruixuan Li. A comprehensive survey on training acceleration for large machine learning models in iots. *IEEE Internet of Things Journal*, 2021.
- [73] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 215–228, 2021.
- [74] Maolin Wang, Seyedramin Rasoulinezhad, Philip HW Leong, and Hayden KH So. Niti: Training integer neural networks using integer-only arithmetic. *arXiv preprint arXiv:2009.13108*, 2020.
- [75] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.
- [76] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jiliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. Melon: Breaking the memory wall for resource-efficient on-device machine learning. 2022.
- [77] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.
- [78] Xundong Wu, Yong Wu, and Yong Zhao. Binarized neural networks on the imagenet classification task. *arXiv preprint arXiv:1604.03058*, 2016.
- [79] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. Deep-type: On-device deep learning for input personalization service with minimal privacy concern. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*.
- [80] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. Deepwear: Adaptive local offloading for on-wearable deep learning. *IEEE Transactions on Mobile Computing*, 19(2):314–330, 2019.
- [81] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144, 2018.
- [82] Chengxu Yang, Qipeng Wang, Mengwei Xu, Zhenpeng Chen, Kaigui Bian, Yunxin Liu, and Xuanzhe Liu. Characterizing impacts of heterogeneity in federated learning upon large-scale smartphone data. In *Proceedings of the Web Conference 2021*, pages 935–946, 2021.
- [83] Yukuan Yang, Lei Deng, Shuang Wu, Tianyi Yan, Yuan Xie, and Guoqi Li. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks*, 125:70–82, 2020.
- [84] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.
- [85] Qunsong Zeng, Yuqing Du, Kaibin Huang, and Kin K Leung. Energy-efficient resource management for federated edge learning with cpu-gpu heterogeneous computing. *IEEE Transactions on Wireless Communications*, 20(12):7947–7962, 2021.
- [86] Jinrui Zhang, Deyu Zhang, Xiaohui Xu, Fucheng Jia, Yunxin Liu, Xuanzhe Liu, Ju Ren, and Yaoxue Zhang. Mobipose: Real-time multi-person pose estimation on mobile devices. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 136–149, 2020.
- [87] Qiyang Zhang, Xiang Li, Xiangying Che, Xiao Ma, Ao Zhou, Mengwei Xu, Shangguang Wang, Yun Ma, and Xuanzhe Liu. A comprehensive benchmark of deep learning libraries on mobile devices. *arXiv preprint arXiv:2202.06512*, 2022.
- [88] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, et al. Fixed-point back-propagation training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2330–2338, 2020.
- [89] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. *ACM SIGARCH Computer Architecture News*, 33(5):27–32, 2005.
- [90] Kai Zhong, Tianchen Zhao, Xuefei Ning, Shulin Zeng, Kaiyuan Guo, Yu Wang, and Huazhong Yang. Towards lower bit multiplication for convolutional neural network training. *arXiv preprint arXiv:2006.02804*, 3(4), 2020.
- [91] Qihua Zhou, Song Guo, Zhihao Qu, Jingcai Guo, Zhenda Xu, Jiewei Zhang, Tao Guo, Boyuan Luo, and Jingren Zhou. Octo:{INT8} training with loss-aware compensation and backward quantization for tiny on-device learning. In *2021 USENIX Annual Technical Conference*, pages 177–191, 2021.
- [92] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.
- [93] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive quantization for deep neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [94] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.
- [95] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1969–1979, 2020.