Characterizing Parameter Scaling with Quantization for Deployment of CNNs on Real-Time Systems

CALVIN B. GEALY, University of Pittsburgh, Pittsburgh, United States
ALAN D. GEORGE, University of Pittsburgh, Pittsburgh, United States

Modern deep-learning models tend to include billions of parameters, reducing real-time performance. Embedded systems are compute-constrained while frequently used to deploy these models for real-time systems given size, weight, and power requirements. Tools like parameter-scaling methods help to shrink models to ease deployment. This research compares two scaling methods for convolutional neural networks, uniform scaling and NeuralScale, and analyzes their impact on inference latency, memory utilization, and power. Uniform scaling scales the number of filters evenly across a network. NeuralScale adaptively scales the model to theoretically achieve the highest accuracy for a target parameter count. In this study, VGG-11, MobileNetV2, and ResNet-50 models were scaled to four ratios: 0.25×, 0.50×, 0.75×, 1.00×. These models were benchmarked on an ARM Cortex-A72 CPU, an NVIDIA Jetson AGX Xavier GPU, and a Xilinx ZCU104 FPGA. Additionally, quantization was applied to meet real-time objectives. The CIFAR-10 and tinyImageNet datasets were studied. On CIFAR-10, NeuralScale creates more computationally intensive models than uniform scaling for the same parameter count, with relative speeds of 41% on the CPU, 72% on the GPU, and 96% on the FPGA. The additional computational complexity is a tradeoff for accuracy improvements in VGG-11 and MobileNetV2 NeuralScale models but reduced ResNet-50 NeuralScale accuracy. Furthermore, quantization alone achieves similar or better performance on the CPU and GPU devices when compared with models scaled to 0.50×, despite slight reductions in accuracy. On the GPU, quantization reduces latency by 2.7× and memory consumption by 4.3×. Uniform-scaling models are 1.8× faster and use 2.8× less memory. NeuralScale reduces latency by 1.3× and dropped memory by 1.1×. We find quantization to be a practical first tool for improved performance. Uniform scaling can easily be applied for additional improvements. NeuralScale may improve accuracy but tends to negatively impact performance, so more care must be taken with it.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Computing methodologies → Computer vision**; **Neural networks;**

Additional Key Words and Phrases: Computer vision, benchmarking, machine learning, parameter scaling, convolutional neural networks

ACM Reference Format:
Calvin B. Gealy and Alan D. George. 2024. Characterizing Parameter Scaling with Quantization for Deployment of CNNs on Real-Time Systems. *ACM Trans. Embedd. Comput. Syst.* 23, 3, Article 38 (May 2024), 35 pages. https://doi.org/10.1145/3654799

This research was supported by SHREC industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783.

Authors' address: C. B. Gealy and A. D. George, University of Pittsburgh, Electrical and Computer Engineering, 4420 Bayard St, Suite 560, Pittsburgh, PA 15213; e-mails: c.gealy@pitt.edu, alan.george@pitt.edu.

ACM 1539-9087/2024/05-ART38
https://doi.org/10.1145/3654799

1 INTRODUCTION

Convolutional neural networks (**CNNs**) are a popular **deep-learning** (**DL**) architecture and provide the backbone for many high-accuracy, real-time computer-vision applications. Over the past decade, these models have grown larger, both in terms of the number of parameters and the number of **floating-point operations** (**FLOPs**) intrinsic to a model. This increase in parameters and FLOPs tends to lead to models that can achieve higher accuracies but at the cost of latency for end-to-end inference or memory required to perform inference. Real-time apps, by contrast, often have to meet stringent time constraints on less computationally capable embedded systems, which are often used for their reduced size, weight, and power consumption. In order to reduce the size of these models, researchers have designed methods to scale the number of parameters in a model to make them more amenable to resource-constrained systems. Therefore, it is important to study the tradeoffs in terms of model complexity and on-device performance.

In this research, we study the impact of two parameter-scaling methods, uniform scaling and NeuralScale, on inference performance when considering the deployment of a real-time application. Uniform scaling is a simple method that uniformly changes the number of filters in a CNN across layers [12]. NeuralScale, a form of **neural architecture search** (**NAS**), is a recently published method that attempts to learn the importance of parameters; based on the importance of the parameters, it can then selectively remove filters from each layer to achieve the best accuracy for a target parameter count [18].

To evaluate the two methods, measurements of inference latency, runtime-memory usage, and power consumption were gathered across three models: VGG-11, MobileNetV2, and ResNet-50, at four scaling ratios: 0.25×, 0.50×, 0.75×, 1.00×. These models were chosen to highlight three common baseline architectures with initial parameter counts ranging from approximately 2 million to over 20 million and to mirror several of the models from Reference [18]. To understand the implication of scaling across a wide variety of devices, results were gathered on embedded CPU, GPU, and FPGA platforms. In order to evaluate the tangible impacts of these methods, we consider a hypothetical streaming camera application with a frame rate of 60 **frames per second** (**FPS**), which can be required for advanced apps [14, 20]. Any model that achieves a latency less than 16.67 ms is considered to have achieved real-time processing; any further reductions in latency indicate that there is extra time in the processing budget for other applications on the real-time system.

We also consider the impact of quantization on models to determine the efficacy of combining both parameter-scaling methods and quantization toward reaching the goal of real-time performance. The CIFAR-10 dataset [15] is used for evaluation to enable easy comparisons to other classification studies. Additional results in this study compare the effect of dataset on performance by evaluating MobileNetV2 models trained on both CIFAR-10 and tinyImageNet [19].

This research provides significant insight into the performance of models scaled using uniform scaling and NeuralScale on embedded devices. Previous research in [9] explored parameter scaling on an embedded CPU and embedded GPUs by using models trained by the authors of [18]. As an extension of [9], this article provides several major additions. First, we retrained all models and gathered new results on a different dataset, CIFAR-10, for novel discussion on the tradeoffs between accuracy and device performance. We also present new comparisons of MobileNetV2 trained on the tinyImageNet dataset. The second, and largest, addition to this study is our testing of the impact of quantization on model performance. Quantization is a tool that can be used in conjunction with scaling methods, as well as on its own. We, therefore, consider the impact of quantization in two ways: first, on performance when quantization is applied to scaled models, and second, as an alternative method to scaling for improving the performance of models towards reaching real-time performance goals. Third, to further guide the use of parameter scaling, we expand our

initial inference analysis to include details on the training processes for both uniform scaling and NeuralScale, including the time of generating these models with all methods. Fourth, we added a new class of device to this research by characterizing the runtime behavior of inference with scaled models on an FPGA-based embedded system, which contextualizes performance on latency-sensitive, real-time systems. Fifth, we measured the power consumption for each of the models tested in this research. Finally, we use the collective results of our study to provide new guidance on selecting a scaling ratio, a non-trivial task without the data gathered from this research that relates device performance to parameter count. Based on our observations, we outline a process for choosing a scaling ratio with a latency or memory target given one of the devices we tested. In summary, the contributions of this article include an analysis of:

— Latency, runtime-memory usage, and power measurements across three devices at four scaling ratios using two scaling methods.
— The effect of the dataset on hardware performance.
— Using quantization in conjunction with parameter scaling and its effect on runtime performance.
— Tradeoffs during training when using NeuralScale.
— NeuralScale's restructuring of models and limitations to performance due to increased FLOPs.
— Guidance for scaling ratio selection based on real-world observations.

2 BACKGROUND AND RELATED RESEARCH

This section contains background details on methods utilized by this research, including uniform scaling and NeuralScale. It also provides context about these methods and how they relate to other research. Additionally, details about quantization methods are included.

2.1 Neural-Network Performance Metrics

The goal behind neural-network metrics is to capture some of the complexity of a model to understand its inference performance on a particular hardware system. Two common metrics, parameters and FLOPs, are described. The background on these metrics is followed by **Critical Datapath Length** (**CDL**), which is a recently developed metric for understanding model performance.

2.1.1 Parameters and FLOPs. Parameters and FLOPs are two common metrics reported with newly published models. These two metrics help capture some of the complexity of a model. The number of parameters is a count of all of the trainable values used to generate a prediction. FLOPs are a measure of the FLOPs in a model. This value can be derived from the number of multiply-accumulates in the model, though this estimate does ignore operations like the sigmoid activation function. However, this value is imperfect in terms of predicting model performance given discrepancies in how lower-level software may fuse kernels or change the execution of the kernels, especially when highly parallel accelerators like GPUs are used [16].

2.1.2 Critical Datapath Length. CDL was proposed by Langerman et al. in Reference [16] and can be found as the diameter of the directed-acyclic graph representing a model. It represents the length of the serial path through which data flows during inference. As noted in their research, the standard metric of FLOPs is not indicative of real-world performance, especially on massively parallel devices like GPUs. CDL is included in this research as it is another metric that has been introduced to better understand and predict model performance. A larger CDL indicates that there is a longer path through which the data must flow during inference, leading to relatively longer execution times on highly parallel devices.

2.2 Neural Architecture Search

Designing neural networks that perform well on resource-constrained devices has become a research topic of interest in the machine-learning and computer-vision fields. Recently, there has been a significant amount of interest in NAS, where a neural-network architecture is evolved algorithmically to adapt to specific constraints. Within NAS, there is research interest in creating networks optimized for specific devices. Cai et al. perform a proxyless NAS by generating an over-parameterized network and then conducting path-level pruning to create a network that is efficient for a given platform [3]. This method is performed while avoiding simpler tasks that do not relate to a given network, dataset, or target platform. Cai et al. demonstrate NAS by choosing a sub-network from a supernet for deployment on a given device [2]. They do this operation by training the supernet and then fine-tuning subsets of the supernet. With this method, they were able to achieve a new state of the art on ImageNet. Another method for pruning is ResRep [7]. This method uses structural pruning to remove entire filters from CNN layers to achieve a minimal loss in accuracy for a target FLOPs reduction. The authors of ResRep claim to achieve near lossless accuracy performance with a ResNet-50 model that has only 45% of the FLOPs of the original model.

2.3 NeuralScale

Another NAS-based reduction method called NeuralScale was proposed by Lee and Lee [18]. With NeuralScale, only the width component of the network is scaled. NeuralScale attempts to select the best number of filters for each individual layer at a given scaling factor rather than scaling uniformly across layers. The authors showed accuracy improvements over other scaling methods [18].

To target a specific number of parameters, NeuralScale first performs an iterative pruning method to determine the *importance* of each filter in the network [18]. The importance is defined as a measurement of the increase in error caused by the removal of that filter. When removing a filter causes a large decrease in accuracy, the filter is considered to be more important. This metric of importance is explained in the research of Molchanov et al. [21].

As the pruning method is performed across many iterations, the change in the number of filters per layer compared with the number of parameters in the network is learned. With this knowledge, a model of the number of filters in a layer given the total number of parameters can be generated. Essentially, a curve is fit which shows how the parameter count changes as the number of filters is increased for each layer. Using the learned curve, stochastic gradient descent is then used to refine the model until the target parameter count is reached. An in-depth, mathematical explanation can be found in the NeuralScale article [18]. The steps of iteratively pruning the model, searching for the parameters for the curve, and then generating the network with stochastic gradient descent represent one iteration of what the authors define as architecture descent. Architecture descent can then be run iteratively for a set number of steps or until convergence.

After completing several iterations of architecture descent, the method requires a scaling ratio to be set. The filter layout for the model is then generated using the previously learned curve. The target parameter count is set so that the number of parameters in the generated model is similar to the number of parameters that would exist in the original model when each layer is uniformly multiplied by the scaling ratio. The final step is to train the newly scaled model architecture on the target dataset. The entire NeuralScale process is depicted in Figure 1.

2.4 MobileNets and Uniform Scaling

MobileNets are a group of networks aimed at making an efficient architecture for mobile devices [12]. MobileNetV1 and MobileNetV2 use a depthwise-separable convolution to reduce the

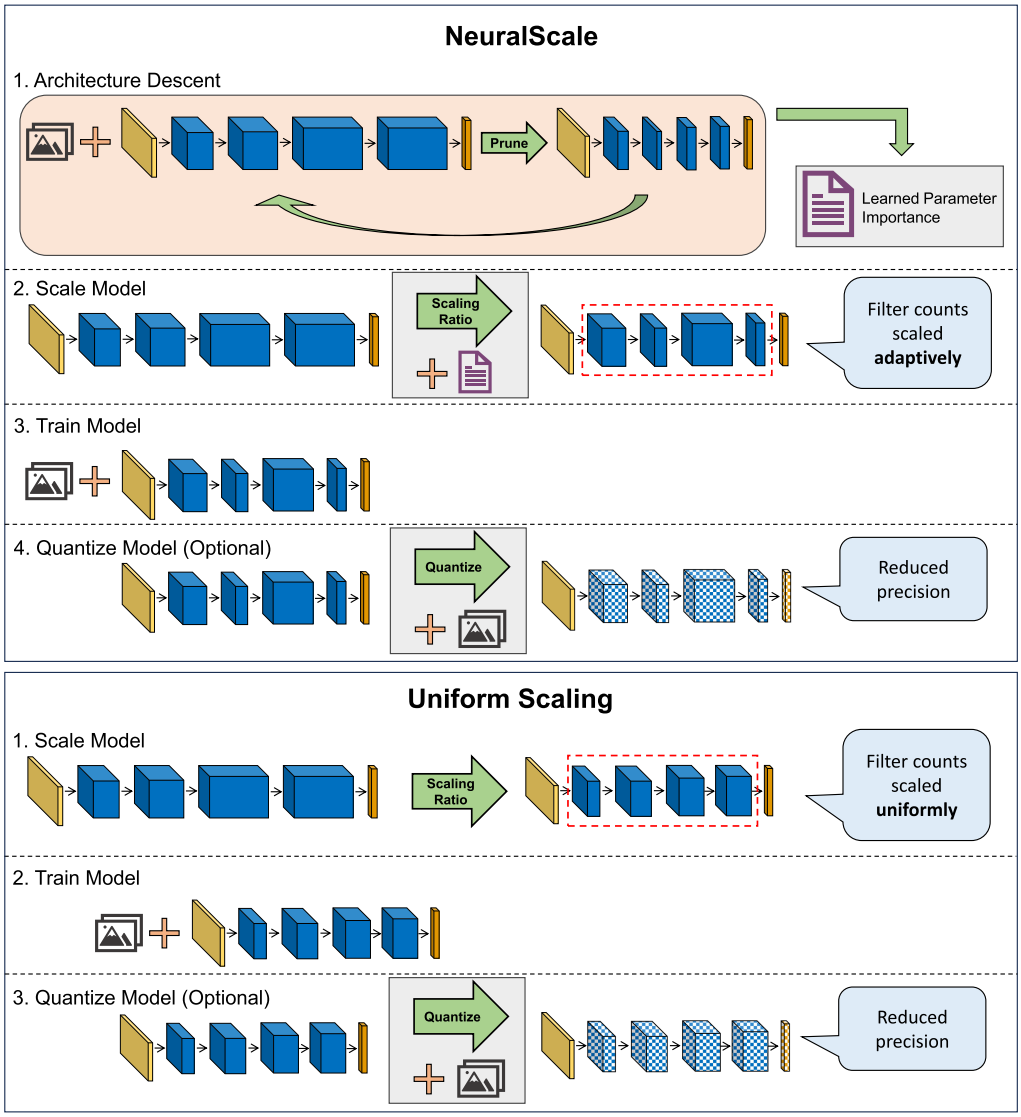


Fig. 1. This figure shows diagrams of the NeuralScale and uniform-scaling processes. (Top diagram) The first step in the NeuralScale process is architecture descent [18]. Here, the target dataset is used to determine the importance of the parameters of each filter to the overall accuracy of the model. This importance is then used to create a new base model, which is then again iteratively pruned. This process is repeated for 15 iterations in this research. After architecture descent, the importance of the parameters has been learned. Second, the model is scaled using a scaling ratio and the learned parameter importance. Note that it generates a model where the output layers are not a uniform multiple of the original model, shown in the dashed red box. In the third step, the model must be trained on the target dataset. The final, optional step is to quantize the model to reduce datatype precision. (Bottom diagram) For uniform scaling, the first step is to uniformly multiply all filter counts by the same scaling ratio. The second step is to train the model on the target dataset. The final, optional step is to quantize the model.

computational complexity of the models [12, 32]. Hyperparameters of the network include the width multiplier and image resolution. The effect of the width multiplier on the network is the same as the effect of the scaling ratio with uniform scaling in this research. The individual number of filters in each layer are all multiplied, uniformly, by the scaling ratio to shrink the model to target fewer parameters and make it more amenable to resource-constrained systems. Figure 1 highlights the main steps in uniform scaling.

2.5 Quantization for Improved Inference

Modern computer-vision models tend to be over-parameterized [6], meaning they are a good candidate for quantization. Through quantization, the precision of a model can be lowered from standard **32-bit floating-point** (**float32**) values to **half-precision floating point (float16)**, **8-bit integer (int8)**, or even binary. Quantization can improve the runtime performance of models on target platforms [10]. Additionally, some hardware accelerators like Xilinx's DPU run only int8 models [37]. In our research, int8 quantization is used as it is supported by the tools of all tested platforms.

Quantization can take many forms. Of interest to this research is **post-training quantization** (**PTQ**). PTQ has exceptionally low engineering overhead as models are not retrained or fine-tuned. Part of the training dataset may be used to determine scaling values for the activations in the model. This process is known as calibration. If PTQ does not achieve satisfactory performance, further methods like **quantization-aware training** (**QAT**) can be used to fine-tune the model on the training dataset [10].

2.6 Real-Time Computer Vision

There are many cases where a computer-vision application would need to be performant on an embedded system. One of the most relevant examples is the vision processing system for a self-driving car. Consider a car moving at 80 miles per hour, roughly 117 feet per second. If an obstruction suddenly enters the roadway 20 feet ahead of the vehicle, an object detector with 100 milliseconds of processing time will produce its first result when the car has already traversed half of the distance to the object in question. Therefore, it is critically important that low latency is achievable on embedded devices. Furthermore, the high-resolution vision necessary for detecting small obstructions will be even slower due to the added computational cost from the increase in the number of pixels and limited memory bandwidth on the embedded devices.

For this research, we consider a real-time latency constraint of 16.67 ms, or a frame rate of 60 FPS. Many state-of-the-art computer-vision apps target this rate even with more complex goals than classification [14, 20]; it follows that simple classification models should also be able to reach this real-time goal. Ideally, models will only use a fraction of this time as often the processing system needs to perform other operations with this time budget. Therefore, achieving smaller latencies is prudent as less of the time budget spent on the model enables more of the budget to be used for other important applications in the pipeline.

3 APPROACH

In this research, we analyze the performance of three different models using uniform scaling and NeuralScale on three embedded systems. We test these models on one CPU-based device, one GPU-based device, and one FPGA-based device to understand the performance tradeoffs when using parameter scaling for models deployed on resource-constrained systems. Finally, we also explore the effect of quantizing scaled models and explore the tradeoffs between using scaling and quantization separately.

Table 1. Overview of the Three Embedded Devices and the Accompanying Inference Frameworks

Platform	Type	Inference Framework	Datatype Precisions Tested
ARM Cortex-A72 (Raspberry Pi 4)	CPU	ONNX Runtime	float32, int8
NVIDIA Jetson AGX Xavier	GPU	NVIDIA TensorRT	float32, mixed float32/int8
Xilinx ZCU104	FPGA SoC	Vitis AI & Xilinx DPU	int8

Table 2. Specifications of the AGX Xavier [25, 35] and GTX 1080 Ti [24, 34]

Metric	AGX Xavier	GTX 1080 Ti
Architecture	Volta	Pascal
Peak Perf. (TFLOPS)	1.4	11.3
CUDA Cores	512	3584
TDP (W)	30	250
Memory (GB)	16	11
Mem. BW (GB/s)	136	484

3.1 Test Platforms

Details about the three embedded device platforms are included below. An overview of the test platforms is shown in Table 1. Each device runs a separate inference framework designed for improved performance on its architecture. A brief overview of the three frameworks studied is also included.

3.1.1 CPU Platform and ONNX Runtime. This research used the Raspberry Pi 4, which features a quad-core ARM Cortex-A72 CPU [31]. We used it as an embedded CPU baseline due to its simplicity and strong open-source community support. Its uses in rapid prototyping also make it an attractive solution. Though several years old, this processor architecture is still in active use and is therefore still relevant for research purposes. For example, the ARM Cortex-A72 is featured in the Xilinx Versal AI Core **system-on-chip** (**SoC**), which is a new platform for embedded AI applications [38]. The board used in this research features 4.0 GB of RAM. A 64-bit version of the Ubuntu GNU/Linux operating system was used.

On the CPU device, we tested with ONNX Runtime, which is designed for efficient inference with **Open Neural Network Exchange** (**ONNX**) models. ONNX Runtime is an open-source project backed by Microsoft and includes a Python API which allows for optimized execution of ONNX models on various accelerator types as well as standard CPU inference. ONNX Runtime supports different quantization schemes. For this research, we studied static PTQ for int8 models. For more information, the reader is directed to the ONNX Runtime documentation [27].

3.1.2 GPU Platform and NVIDIA TensorRT. The specifications for the Jetson AGX Xavier platform tested in this research are listed in Table 2. For reference to a desktop-class GPU, the specifications of a GTX 1080 Ti are also listed. The GTX 1080 Ti was used in the original NeuralScale research [18]. Note the significant limits to the core count, thermal profile, and memory bandwidth on the embedded device. Also, note that while the Jetson AGX does have more memory than the GTX 1080 Ti, this memory is shared across the GPU and CPU cores. The shared memory architecture can offer performance improvements for GPU memory accesses as data does not have to be copied between the host memory and device memory like on most GPUs.

NVIDIA TensorRT is a tool used for high-performance inference on NVIDIA GPUs [22]. It allows for several performance optimizations such as mixed precision, layer fusion, automatic kernel

selection, and others. Using an ONNX model as input, TensorRT can create the appropriate high-performance engine automatically [1, 26]. For this study, we tested the models with float32 and mixed float32/int8 precision enabled.

3.1.3 FPGA Platform and Xilinx DPU. The Xilinx Zynq UltraScale+ MPSoC ZCU104 evaluation kit served as the FPGA platform in this research. The ZCU104 device features a SoC with a quad-core ARM Cortex-A53 CPU and a Xilinx 16 nm FPGA fabric [40]. The Xilinx-distributed ZCU104 DPU v2022.2-v3.0.0 board image was used for testing on the device.

This device supports the Xilinx **Deep-Learning Processing Unit** (**DPU**). This **intellectual property** (**IP**) core is designed to accelerate the inference performance of CNNs by leveraging the compute resources of the FPGA fabric in the combined CPU/FPGA SoC [37]. The Vitis AI software platform is used to convert PyTorch models into a format that can be deployed on the DPU [13].

3.2 Model Training and Setup

In this subsection, we summarize the training device, framework, and hyperparameters. First, we outline the method for training models on the CIFAR-10 dataset. Next, we detail the additional steps for training MobileNetV2 on tinyImageNet. Model training and achieving the highest accuracy is not the primary focus of our study. Instead, we consider the impact of model structure on device performance. Therefore, before using scaling to deploy a model, additional investigation into the best hyperparameters should be performed to improve model accuracy.

3.2.1 Comparing Scaling Methods with Three Models on CIFAR-10. To study the impact of the base model structure, we trained versions of VGG-11 [33], MobileNetV2 [32], and ResNet-50 [11] using uniform scaling and NeuralScale. The VGG-11 and MobileNetV2 implementations we used were from the original NeuralScale repository [17], but the models were fully retrained in this research. Models were trained on a desktop computer featuring an AMD Ryzen 5 2600 CPU, 16.0 GB of RAM, and an NVIDIA GTX 1080 Ti GPU.

We performed the NeuralScale architecture descent method with a batch size of 128, a learning rate of 0.09, and a weight decay of 5×10^{-4}. The CIFAR-10 dataset [15] was used for training. Data augmentations were used to artificially increase the training dataset size; these augmentations resized the images to 32×32, randomly cropped, randomly horizontally flipped, and randomly erased sections of the image. Following the method of the original NeuralScale research [18], we performed 15 iterations of architecture descent.

Once the search space was explored through architecture descent, we scaled the three models using uniform scaling and NeuralScale to four ratios: 0.25×, 0.50×, 0.75×, 1.00×. We generated five models at each of these ratios using the two scaling methods to create a more fair accuracy comparison between the two methods. The same hyperparameters were used for this training setup. 150 epochs were used. Additionally, the learning ratio was reduced by a factor of 10 at epochs 50, 100, and 125. A training/validation split of 80/20 was used to save the best weights on the validation set during the training process.

We also gathered metrics inherent to each model. FLOPs were found by **measuring the multiply accumulates (MACs)** using [42] and multiplying the MACs by two. While this method is not a perfect way to calculate FLOPs, it does provide an estimate. The CDL was measured by exporting the model to the ONNX format and by then finding the diameter of the graph of the model [16].

Training was performed with PyTorch version 1.12.1. Models were exported to ONNX using opset version 13. Only the model that performed best on the validation split was exported to the ONNX format. The ONNX models are ingested by both ONNX Runtime and NVIDIA TensorRT. The trained PyTorch models were deployed via Vitis AI.

3.2.2 Comparing Scaling Methods with Two Datasets on MobileNetV2. We also investigate the impact of datasets on real-time performance to determine how performance scales with larger inputs. The same training method previously used on the three models with CIFAR-10 was used to train MobileNetV2 models with the tinyImageNet dataset [19] installed via the Python package from [30]. CIFAR-10 contains 32×32-pixel RGB images with 10 classes. The tinyImageNet dataset contains 64×64-pixel RGB images with 200 classes, representing a subset of the data in the ImageNet dataset [5]. By holding the model configuration and training parameters consistent when comparing with two datasets, we can investigate the impact of dataset on the performance of the reduction methods. We chose to perform this investigation on MobileNetV2 as testing showed good performance across the devices when trained on CIFAR-10, and [18] also performed testing between MobileNetV2 and tinyImageNet.

The larger image sizes caused the GPU to run out of memory during architecture descent on the NVIDIA GTX 1080 Ti GPU. Therefore, when training the tinyImageNet models, a reduced batch size of 64 had to be used. Otherwise, for simplicity of comparison, the same hyperparameters from CIFAR-10 training were used. Note that more advanced hyperparameter selection could improve accuracy, but hyperparameter tuning in general is dataset-dependent and therefore outside the scope of this study. Consistent with our other study across models, comparisons will only be made between the reduction methods within a dataset and not between datasets.

3.3 CPU Benchmarking

The first CPU-specific task is the generation of quantized ONNX models. For simplicity, quantization was performed on the same device as the original training. ONNX Runtime static PTQ [28] was used to convert the float32-based ONNX model into an int8 model. To perform quantization, 1,000 random training image samples were used as a calibration dataset. The ONNX Runtime quantizer was then run on the model, and a final accuracy on the test dataset was gathered.

A separate Docker container with ONNX Runtime version 1.12.1 was built for testing on the ARM Cortex-A72 device. The Raspberry Pi 4 default maximum clock rate of 1.5 GHz was used for testing. We gathered latency measurements by sending random data through the models with 20 cycles of warmup followed by 80 cycles of measured inferences. Memory measurements were gathered using the Python Memory Profiler [29]. This tool measures the amount of runtime memory by querying the OS to determine the amount of memory used. It also can report the specific amount of memory used by a specific line of Python code. The profiler was used to record the amount of memory needed for the ONNX Runtime InferenceSession generation. Both latency and memory measurements were averaged over 50 trials to better sample inference latency and runtime-memory utilization. All measurements were gathered using a batch size of 1 to represent a streaming app performing inference on 1 image at a time, a common use case on embedded systems.

3.4 GPU Benchmarking

NVIDIA JetPack version 5.1, which includes TensorRT version 8.5.2, was prepared on the NVIDIA AGX Xavier. The AGX Xavier includes several different power modes. For this testing, we configured the device for an approximate max power budget of 30 W, with two CPU cores enabled at a frequency of 2,100 MHz, a GPU frequency of 900 MHz, and a memory frequency of 1,600 MHz. This mode was chosen to enable the fastest clock frequencies for the GPU.

Using the TensorRT Python API, we converted the original float32 ONNX models into serialized TensorRT engines. We also enabled the conversion of the model using int8 operations by creating a calibration cache using 1,000 training images as described in the TensorRT documentation, which allows TensorRT to perform implicit quantization [23]. Note that TensorRT engine generation is non-deterministic due to the use of profiling to optimize kernel selection at the time of generation.

The accuracy of both unquantized and quantized engines was measured using the test datasets through the Python TensorRT API.

Latency measurements were gathered on all serialized engines using NVIDIA's *trtexec* program, details of which can be found in the TensorRT documentation [23]. We used this program to load the serialized engines generated through the Python API. This program then profiled the latency of inference using a batch size of 1 by sampling the performance for a total of 10 seconds with 2 seconds of warmup. Memory measurements were also gathered using *trtexec*. To estimate the runtime memory required, the TensorRT documentation recommends summing the size of the serialized engine, the persistent memory allocated, and the memory allocated to store the activations [22]. We gather the size of the serialized engine using the *du -b* Bash command. The size of persistent memory and activation memory are reported in the verbose mode of the *trtexec* program. Since all steps from engine generation to runtime performance can have variance, we performed this pipeline of engine generation and analysis 50 times to sample the overall latency and memory requirements of each model when using TensorRT for inference.

3.5 FPGA Benchmarking

The Xilinx-released Vitis-AI Docker container, version 3.0.0.106, was used for converting the PyTorch models into DPU models. This container was run on the same systems used for model training. The pipeline found in Xilinx's sample code at [36, 39] was used for reference. The DPU only runs on quantized models, therefore no float32 results are gathered. We use 500 sample images from the training dataset to calibrate the quantizer. Then we export the model and set it to target the ZCU104.

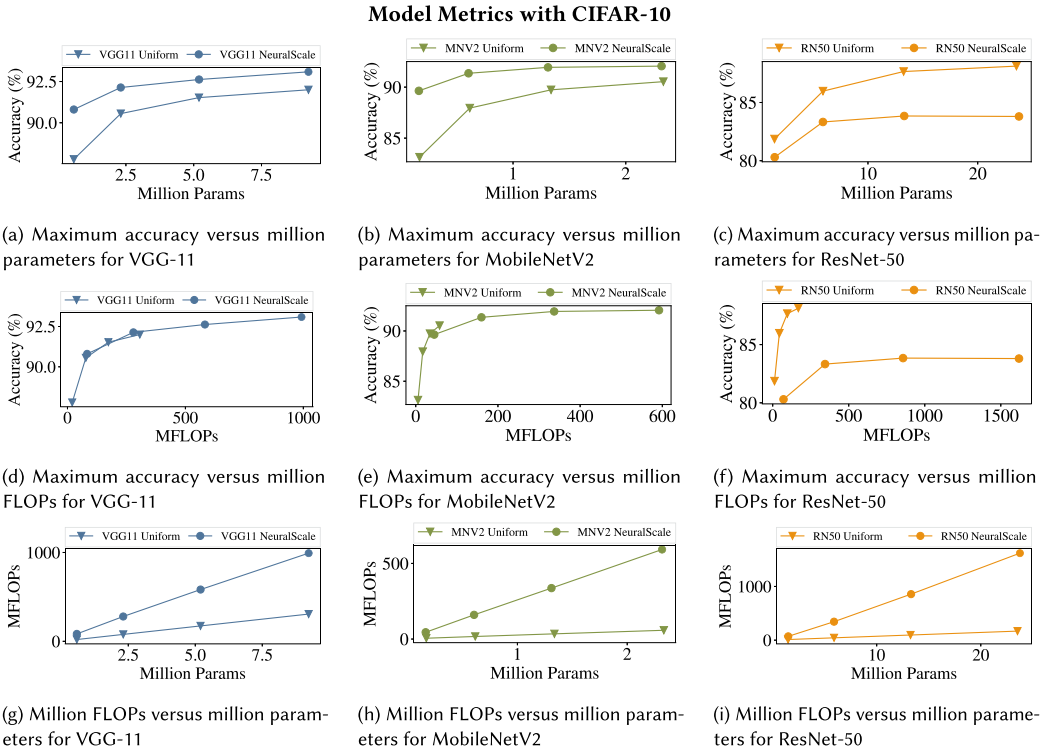
A modified version of the app from Reference [36] was created to load the 10,000 test images in the test datasets. This app reports the latency of inference when using only 1 thread, essentially setting the batch size to 1 as the thread concurrency represents the number of inferences that can be performed in parallel on the DPU. Accuracy on the test dataset was recorded on the device. Again, the latency performance was averaged over 50 trials.

3.6 Power Benchmarks

Power results are reported in terms of the dynamic power to compare how power utilization varies across the models. Additionally, the results are reported as the throughput (images/sec) per watt of peak load power. This metric helps to identify the overall efficiency of inference on the devices. To avoid device-dependent frameworks, an external power meter was used to measure the total device power for the three devices under test. Specifically, we used the Poniie PN2000 Electricity Usage Monitor. Idle power measurements were gathered when no foreground processes were running on the devices. Then, we measured the power during the execution of each model and recorded the peak load power. The dynamic power consumed during the execution of each model can be found by subtracting the idle power from the peak load power measurements. We found these measurements to be precise enough to identify trends in power across scaling methods and scaling ratios.

4 RESULTS

The results section details model metrics, device performance, and quantization results. For graph legibility, VGG-11, MobileNetV2, and ResNet-50 are abbreviated as VGG11, MNV2, and RN50, respectively. This research aims to compare scaling methods rather than model types. Therefore, we will only draw comparisons between the two methods and not between model types. The uniform-scaling 1.00× models represent the original models before any scaling. Error bars representing the standard deviation are included on all latency and memory usage charts, as well as the accuracy



(a) Maximum accuracy versus million parameters for VGG-11

(b) Maximum accuracy versus million parameters for MobileNetV2

(c) Maximum accuracy versus million parameters for ResNet-50

(d) Maximum accuracy versus million FLOPs for VGG-11

(e) Maximum accuracy versus million FLOPs for MobileNetV2

(f) Maximum accuracy versus million FLOPs for ResNet-50

(g) Million FLOPs versus million parameters for VGG-11

(h) Million FLOPs versus million parameters for MobileNetV2

(i) Million FLOPs versus million parameters for ResNet-50

Fig. 2. Model metrics for three models under study with CIFAR-10. Each marker represents the four scaling ratios from left to right: 0.25×, 0.50×, 0.75×, 1.00×.

bars for the NVIDIA AGX Xavier due to the non-deterministic nature of engine generation. For latency, memory, and dynamic power, a *lower* value is better. For accuracy and throughput per watt, a *higher* value is better.

Accuracy measurements are included in this study to relate changes in accuracy given the scaling methods. Note that all models used the same hyperparameters to make the study of multiple methods, models, and scaling ratios possible. VGG-11 accuracy results in this study are similar to the original NeuralScale research [18]. Additional discussion on accuracy will follow in Section 5.2.

4.1 Model Metrics on CIFAR-10

The maximum accuracy achieved on each model versus the number of parameters are displayed in Figures 2(a), 2(b), 2(c). Due to the wide range in the number of parameters in each model, each graph is displayed separately. Note that the marked points from left to right represent the four scaling ratios in order: 0.25×, 0.50×, 0.75×, and 1.00×. The goal of the NeuralScale method is to achieve a higher accuracy for the same number of parameters. This goal was achieved for VGG-11 and MobileNetV2 as the NeuralScale line is always above the uniform-scaling line. However, that is not the case for the ResNet-50 model tested. While likely a limitation of using the same hyperparameters for all models, it is important to note that NeuralScale alone is not guaranteed to generate a more accurate model than uniform scaling.

Figures 2(d), 2(e), 2(f) show the accuracy compared with the number of FLOPs in each model. For VGG-11 and MobileNetV2, the two lines begin to overlap. When comparing the NeuralScale 0.50×

Table 3. CDL of the CIFAR-10 Models Tested Compared with the Range of MFLOPs for the Scaling Methods

Model	CDL	Uniform Scaling MFLOPs range	NeuralScale MFLOPs range
VGG-11	23	20.06–306.75	83.37–992.31
MobileNetV2	111	5.51–58.01	44.98–591.98
ResNet-50	118	11.80–168.69	70.99–1619.63

The MFLOPs value of the original model is represented by the top of the range for the uniform-scaling models.

VGG-11 model to the original, equivalent to uniform-scaling 1.00×, the large reduction in parameters only leads to a 0.9× increase in FLOPs with a near identical accuracy, only 0.14% different. For MobileNetV2, the NeuralScale 0.25× may reduce the parameter count over the uniform-scaling 0.75× model, but the NeuralScale model is actually 0.10% less accurate with 1.3× more FLOPs. With ResNet-50, all uniform-scaling models used far fewer FLOPs than their NeuralScale counterparts, often at significant accuracy improvements. Overall, these results indicate that simply scaling to target a number of parameters ignores the potential impact on the number of FLOPs in the model, which is influential to the overall accuracy of the models.

Figures 2(g), 2(h), 2(i) show the number of FLOPs in a model versus the number of parameters. By fitting a line to the number of FLOPs required for each parameter, we can see the rate at which the FLOPs increase relative to the parameter count. All lines have an $R^2 > 0.999$. Compared with uniform scaling, the NeuralScale models increase the FLOPs per parameter at a rate of 3.2× for VGG-11, 10.5× for MobileNetV2, and 9.8× for ResNet-50. This result indicates that NeuralScale learns to increase the number of FLOPs in a model for a target parameter count to try to achieve a more accurate model.

Finally, the CDL for each model and the FLOPs range for the models generated by uniform scaling and NeuralScale are displayed in Table 3. Parameter scaling does not affect the depth of the networks; therefore, all models with the same base have the same CDL. As noted in [16], CDL is a more meaningful metric for memory-bound devices such as GPUs, while FLOPs are more meaningful on compute-bound devices such as CPUs.

4.2 CPU Results

The CPU results are first presented with a comparison of scaling methods while varying the best model. Next, a comparison of scaling methods while varying the dataset is presented using the MobileNetV2 model. Finally, a summary is provided for the reader's benefit. For the CPU device, the idle power was measured to be 3.1 W.

4.2.1 Comparing Scaling Methods with Three Models on CIFAR-10. The inference results for the float32-based models on the ARM Cortex-A72 cores are shown in Figure 3. Across all three devices, the dashed teal lines represent the performance of the original models, while the dashed red line represents a real-time target of 60 FPS. Latency results for a batch size of 1 are shown in Figure 3(a). Note that the uniform-scaling models are always faster than their NeuralScale counterparts. On the CPU, many models struggle to meet real-time performance, represented by the dashed red line on the latency graph. For uniform scaling, one VGG-11 model, three MobileNetV2 models, and one ResNet-50 model achieve this constraint. However, with NeuralScale models, only the 0.25× models achieve real-time latency. NeuralScale is designed for resource-constrained systems, but it does not optimize for latency, thus it is important to consider the impact of the scaling method toward reaching real-time performance.

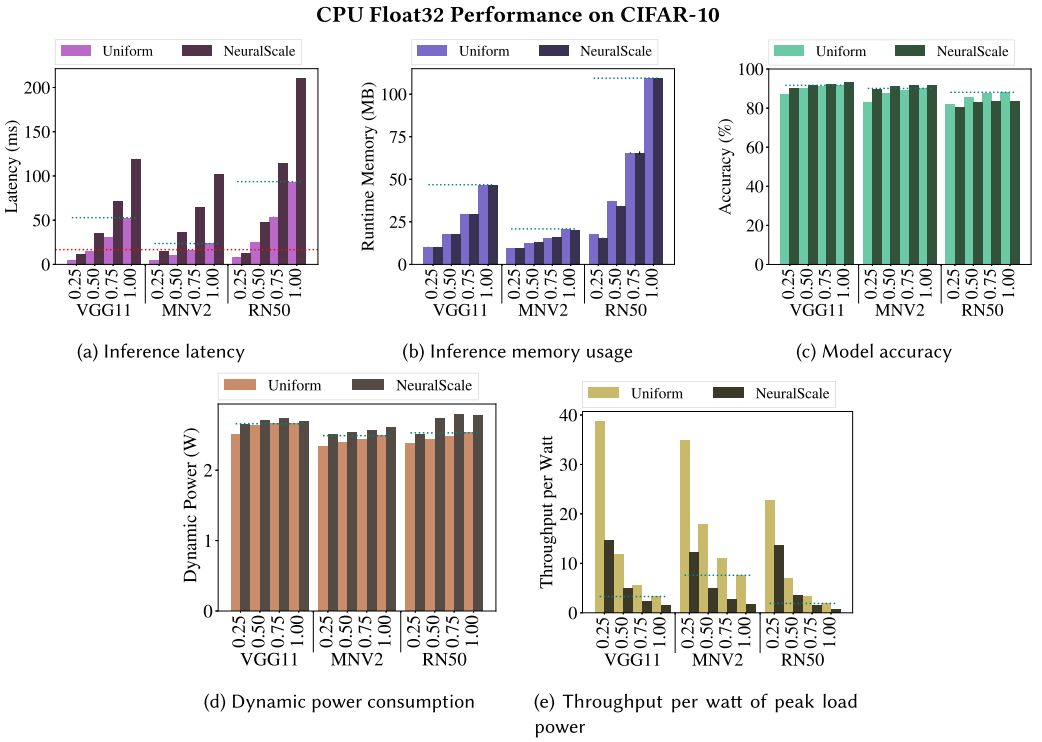
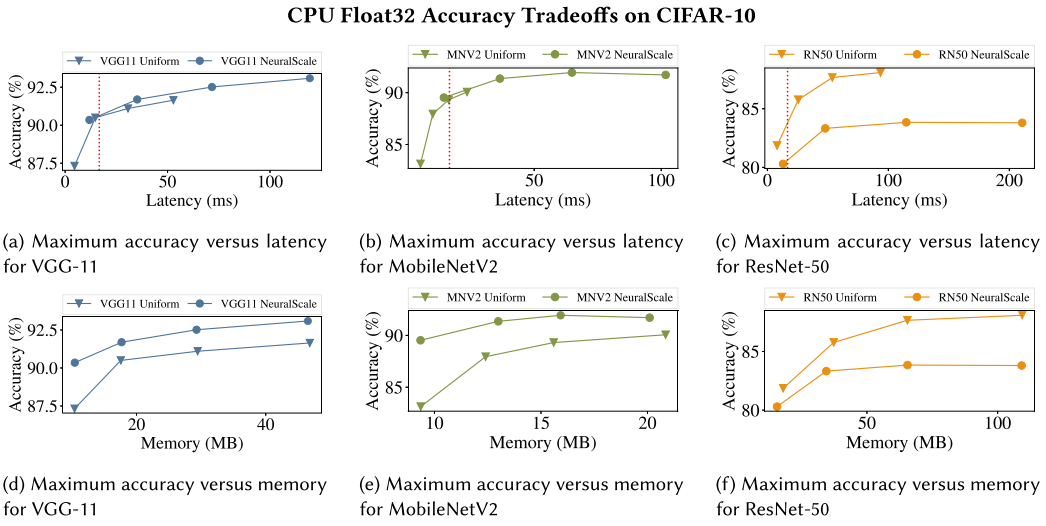


Fig. 3. Arm Cortex-A72 float32 model performance using ONNX Runtime. The dashed red line represents a real-time latency target of 60 FPS. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

Also note that, as shown in Figure 3(b), the memory utilization between the two scaling types is always approximately equivalent. The memory consumption has a Pearson correlation coefficient of 0.999 (p=1.6 · 10^{-30}) with the number of parameters in the model. This result indicates that, by adjusting the number of parameters, it is possible to estimate the amount of memory that will be used by the device. However, given the larger number of FLOPs in the NeuralScale models, they tend to run more slowly on a less parallel device like a CPU.

In terms of power, uniform-scaling models always use less dynamic power than their NeuralScale counterparts, though not by large margins. Given the much faster inference speeds of the uniform-scaling models, throughput per watt of peak load power is significantly higher for uniform-scaling models over NeuralScale. Therefore, we conclude that uniform-scaling models are more efficient for processing on a CPU than their NeuralScale counterparts. For example, to achieve any improvement over the baseline model with MobileNetV2, the scaling ratio has to be set to 0.25× for NeuralScale, while all ratios tested are more efficient for uniform scaling.

Figures 4(a), 4(b), 4(c) show the accuracy versus average latency performance on the three models tested. For VGG-11 and MobileNetV2, we see that the increases in accuracy come at the clear tradeoff of latency on the CPU device. As the uniform scaling and NeuralScale lines overlap, we can see how higher accuracies are only achieved with higher latencies. For the ResNet-50 model, the uniform scaling implementations are always more accurate, often at lower latencies. Figures 4(d), 4(e), 4(f) show the accuracy versus average memory utilization on the three models tested. Given that parameter count is highly correlated with memory utilization on the CPU, we do not see a



(a) Maximum accuracy versus latency for VGG-11

(b) Maximum accuracy versus latency for MobileNetV2

(c) Maximum accuracy versus latency for ResNet-50

(d) Maximum accuracy versus memory for VGG-11

(e) Maximum accuracy versus memory for MobileNetV2

(f) Maximum accuracy versus memory for ResNet-50

Fig. 4. Arm Cortex-A72 float32 model accuracy tradeoffs using ONNX Runtime. The dashed red line represents a real-time latency target of 60 FPS.

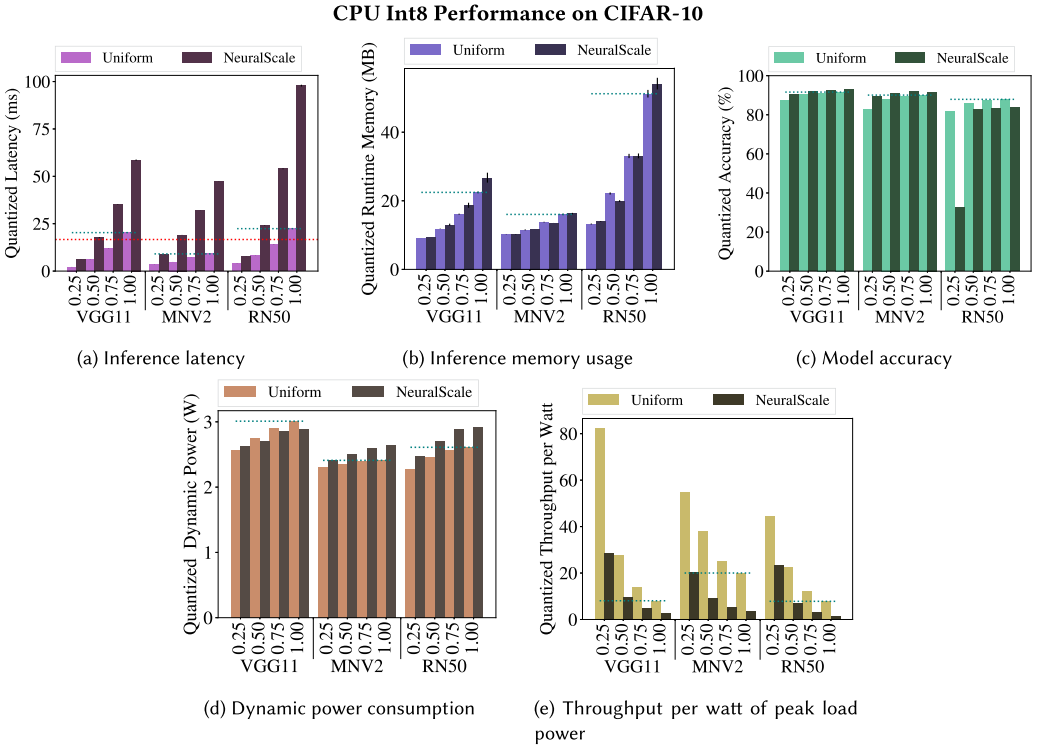
tradeoff between memory and accuracy. On the CPU, NeuralScale models achieve higher accuracies for the same memory for VGG-11 and MobileNetV2, while uniform scaling achieves a higher accuracy for the same memory for ResNet-50.

Next are the results for the int8-based models on the ARM Cortex-A72, shown in Figure 5. Again, similar trends are evident to those found in the float32 models. The uniform-scaling models always, and quite often significantly, outperform their NeuralScale counterparts in terms of inference latency. On average, the quantized NeuralScale models are 3.3× slower than their uniform-scaled counterparts. Several more models now achieve real-time performance. For VGG-11, the three smallest uniform-scaling models are faster than 16.67 ms, while only one NeuralScale model is. For MobileNetV2, all uniform-scaling models meet the real-time constraint, while only the smallest NeuralScale model does. Finally, for ResNet-50, three uniform-scaling models are real-time, while only one NeuralScale model is.

Memory utilization between the two scaling types again tends to be approximately the same. However, an unexpected result in terms of model accuracy occurred for the NeuralScale 0.25× ResNet-50 model. Its uniform-scaling counterpart was able to quantize well, but the NeuralScale version of the model was not able to do so, with an accuracy drop of approximately 48% due to quantization. This drop is likely due to the change in model structure resulting from the NeuralScale process, discussed in Section 5.1.

Different from the float32 models, the int8 uniform-scaling VGG-11 models use more power than their NeuralScale counterparts. Uniform-scaling int8 MobileNetV2 and ResNet-50 models still use less power than NeuralScale versions, as shown in Figure 5(d). The uniform-scaling VGG-11 model can operate more efficiently on the hardware than NeuralScale, as shown by its significantly higher throughput per watt in Figure 5(e). Additionally, the other uniform-scaling models always remain significantly more efficient in terms of throughput per watt than their NeuralScale counterparts.

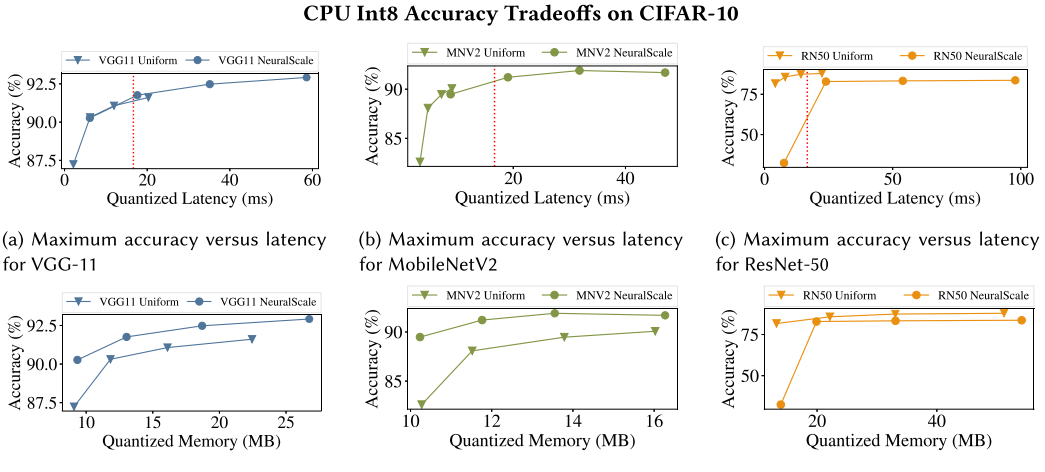
The accuracy versus latency charts shown in Figures 6(a), 6(b), 6(c) are similar to the trends shown on the float32 models. Again we see that accuracy gains for NeuralScale VGG-11 and MobileNetV2 models come at the cost of latency, while the uniform-scaling ResNet-50 models



(a) Inference latency (b) Inference memory usage (c) Model accuracy

(d) Dynamic power consumption (e) Throughput per watt of peak load power

Fig. 5. Arm Cortex-A72 int8 model performance using ONNX Runtime. The large error in quantization accuracy for NeuralScale 0.25× ResNet-50 is likely caused by the change in model structure resulting from the NeuralScale process. The dashed red line represents a real-time latency target of 60 FPS. The dashed teal lines on the bar graphs denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).



(a) Maximum accuracy versus latency for VGG-11

(b) Maximum accuracy versus latency for MobileNetV2

(c) Maximum accuracy versus latency for ResNet-50

(d) Maximum accuracy versus memory for VGG-11

(e) Maximum accuracy versus memory for MobileNetV2

(f) Maximum accuracy versus memory for ResNet-50

Fig. 6. Arm Cortex-A72 int8 model accuracy tradeoffs using ONNX Runtime. The dashed red line represents a real-time latency target of 60 FPS.

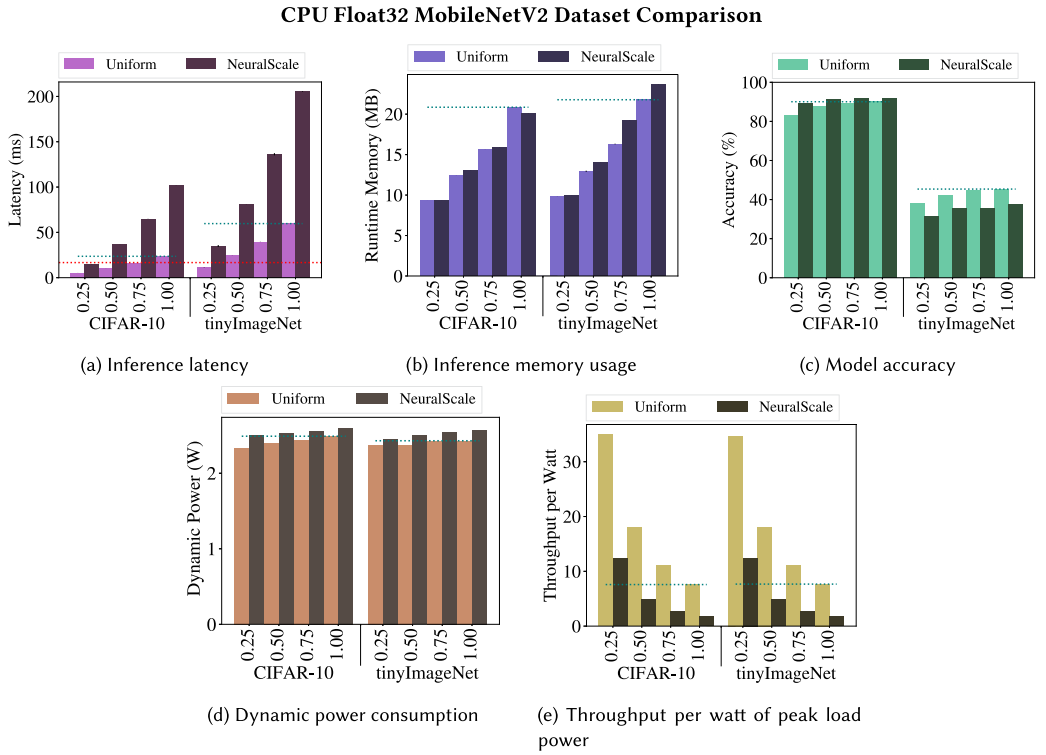


Fig. 7. Arm Cortex-A72 float32 dataset comparison using ONNX Runtime. The dashed red line represents a real-time latency target of 60 FPS. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

outperform their NeuralScale counterparts. In Figures 6(d), 6(e), 6(f), we see memory utilization again is highly correlated with parameter count, and therefore, the most accurate model should be used when targeting memory utilization.

4.2.2 Comparing Scaling Methods with Two Datasets on MobileNetV2. The results for CIFAR-10 and tinyImageNet with float32 MobileNetV2 models are shown in Figure 7. In terms of latency, NeuralScale models are again slower than their uniform-scaling counterparts. For CIFAR-10, NeuralScale models are 3.6× slower, while for tinyImageNet, they are 3.3× slower. The difference between the memory consumptions of NeuralScale and uniform scaling is similar and not significantly impacted by the dataset.

Interestingly, in this study, the NeuralScale models are less accurate on tinyImageNet than the uniform-scaling models. This trend differs from the results of [18]. While it is likely caused by non-optimized hyperparameters, it does indicate that the NeuralScale method is potentially more sensitive to hyperparameter choices, making it more difficult to use than uniform scaling.

Finally, for the power measurements, there are no significant differences in dynamic power, though NeuralScale models do tend to use slightly more power than their uniform-scaling counterparts. Our results also indicate that dataset has essentially no effect on throughput per watt of peak load power. Uniform scaling always significantly outperforms NeuralScale.

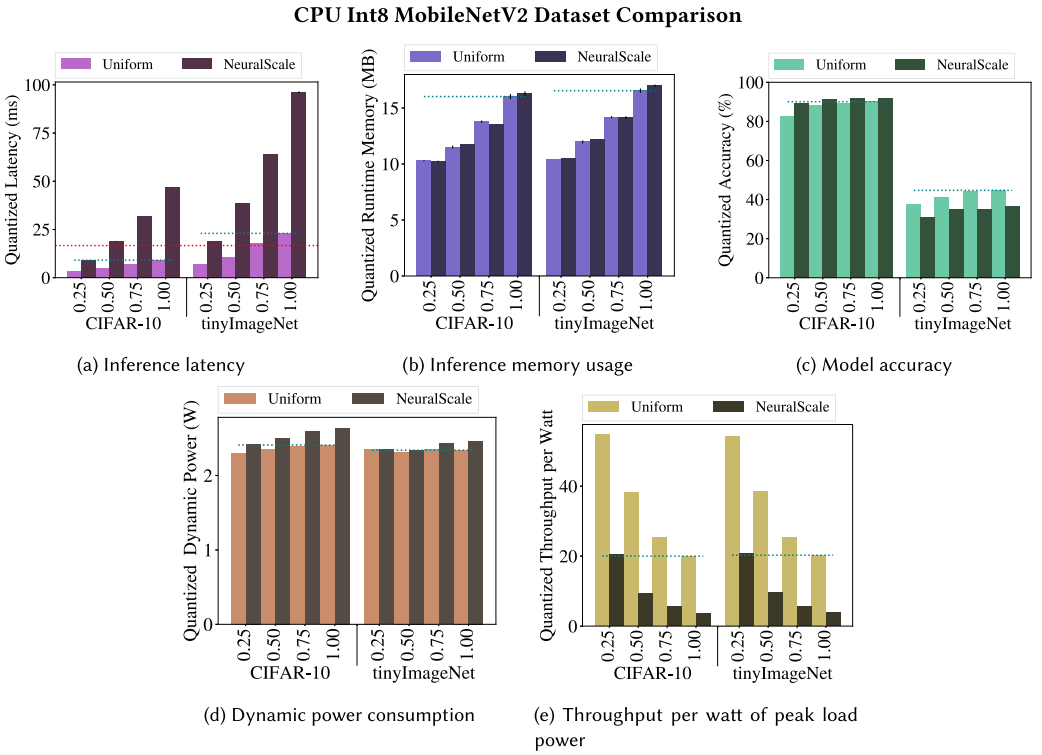


Fig. 8. Arm Cortex-A72 int8 dataset comparison using ONNX Runtime. The dashed red line represents a real-time latency target of 60 FPS. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

Similar results were gathered for the quantized models, shown in Figure 8. NeuralScale models were 3.9× and 3.5× slower than their uniform-scaling counterparts for CIFAR-10 and tinyImageNet, respectively. The trends in memory utilization, accuracy, power, and throughput per watt match the trends seen on float32 models.

4.2.3 CPU Summary. On the CPU, uniform-scaling models are consistently faster than NeuralScale models. Memory usage is approximately equivalent. The difference in accuracy is dependent on the model and dataset. The dynamic power is approximately consistent, though uniform-scaling models often use slightly less power. Finally, the throughput per watt is consistently much higher for uniform scaling than NeuralScale.

4.3 GPU Results

In this section, we report the results from the NVIDIA Jetson AGX Xavier. Similar to the CPU results, we first report the results for the three base models trained on CIFAR-10, followed by a comparison of MobileNetV2 trained on the two datasets. Finally, we provide a summary for review. For the GPU device, the idle power was measured to be 8.1 W.

4.3.1 Comparing Scaling Methods with Three Models on CIFAR-10. On the NVIDIA Jetson AGX Xavier, the uniform-scaling models once again tend to perform faster than their NeuralScale counterparts, but often by a small margin, as shown in Figure 9(a). As the device has greater parallel computational capacity than the CPU testbed, the CDL is the more impactful metric. The GPU is

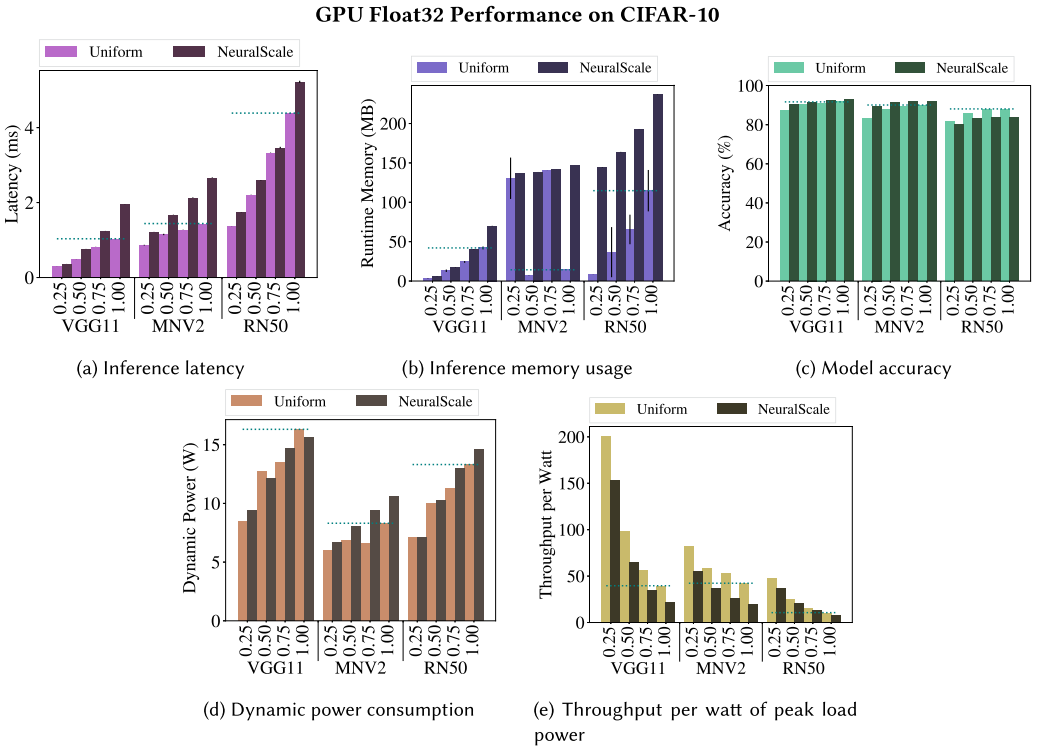
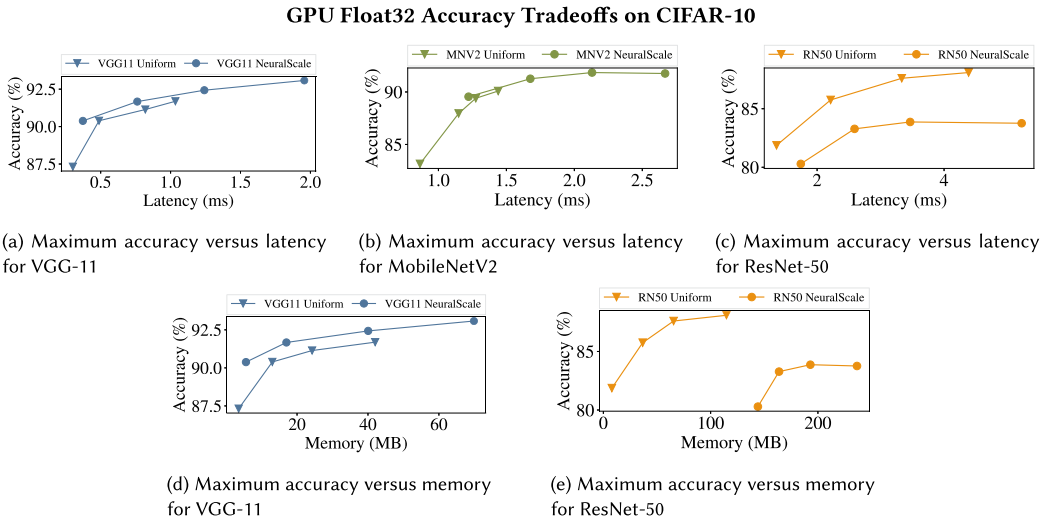


Fig. 9. NVIDIA Jetson AGX Xavier float32 model performance using TensorRT. Error bars for the inference memory footprint are non-negligible. This variance is attributed to the non-deterministic behavior of engine generation caused by large variations in the memory required for the activations when running the model. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

therefore less sensitive to the additional FLOPs of the NeuralScale models, and the execution time difference is less pronounced. The slowest model is roughly 4.5 ms, placing its percent of the overall real-time processing budget at approximately 27%. Therefore, in the worst case, these models leave plenty of processing time for other tasks across all three model types. The memory results on the AGX are more irregular, as shown in Figure 9(b). The trend on the VGG-11 model shows that NeuralScale models tend to use slightly more memory than their uniform-scaling counterparts. The ResNet-50 results also similarly follow this trend, though there is more variance in the memory usage for the uniform-scaling models. This variance is caused by the differing memory usage of the 50 trials of TensorRT engine generation, showing that TensorRT will conduct tradeoffs between memory usage and runtime depending on the kernels it selects for the engine to optimize performance.

For MobileNetV2, the uniform-scaling models at 0.25× and 0.75× use an unexpectedly large amount of memory. This large utilization is attributed to large amounts of memory being used for the activations, one of the three components summed to provide the estimate of overall runtime memory on the GPU devices. This results indicate that, especially given the large variance at 0.25×, sometimes the engine generated by TensorRT uses far more memory than even the same model on different runs of the engine generation. We generated 50 separate engines for each individual model, resulting in large differences in estimated memory consumption for some of the models, shown by the large error bars in Figure 9(b). Given the non-deterministic nature of engine



(a) Maximum accuracy versus latency for VGG-11

(b) Maximum accuracy versus latency for MobileNetV2

(c) Maximum accuracy versus latency for ResNet-50

(d) Maximum accuracy versus memory for VGG-11

(e) Maximum accuracy versus memory for ResNet-50

Fig. 10. NVIDIA Jetson AGX Xavier float32 model accuracy tradeoffs using TensorRT.

generation, it is therefore wise to attempt engine generation multiple times to confirm the most efficient engine is created. It also indicates that TensorRT, on certain attempts to generate the engines, concluded that allocating additional memory for activations was worth the cost for improvements to latency given which kernels it decided to use.

Next are the dynamic-power results, shown in Figure 9(d). On VGG-11, there is not a consistent difference in dynamic-power utilization between uniform scaling and NeuralScale. However, uniform scaling always achieves consistently better throughput per watt than NeuralScale as shown in Figure 9(e). On MobileNetV2 and ResNet-50, the uniform-scaling models always use less power and have better throughput per watt than their NeuralScale equivalents.

Figures 10(b), 10(a), 10(c) show the accuracy versus latency of the models. Once again, we see that the increase in accuracy comes at the cost of latency for VGG-11 and MobileNetV2. Additionally, the NeuralScale line tends to be slightly above the uniform scaling line for the same latency, though the values are close. For ResNet-50, we again show that uniform scaling is best due to its higher accuracy. Figures 10(d), 10(e) show the accuracy versus memory trends. Note that MobileNetV2 is not included due to the large variations in memory. Based on these graphs, NeuralScale achieves better accuracy for the same memory utilization for VGG-11, while uniform scaling outperforms NeuralScale on ResNet-50.

Quantized-model inference results are shown in Figure 11. TensorRT failed to generate a quantized engine for the NeuralScale 0.25× ResNet-50 model as it was unable to calibrate the activations in the model. For the NeuralScale 1.00× ResNet-50 model, it failed to generate an engine as it ran out of memory on the device. These results are therefore not included in the bar charts. Again, the inference latency of the uniform-scaling models is less than that of the NeuralScale models. Across all model types, inference latency is now near or below 1.2 ms. In the worst case, the models are only using about 7.2% of the real-time latency budget, leaving plenty of processing time for other tasks. In general, the NeuralScale versions are 1.2× slower than their uniform-scaling counterparts. In terms of runtime-memory usage, the NeuralScale models use more memory than their uniform-scaling counterparts, but not by as large of a margin as seen in the float32 models.

The dynamic power of the quantized uniform-scaling VGG-11 models, displayed in Figure 11(d), is consistently higher than their NeuralScale counterparts. This larger dynamic power is likely

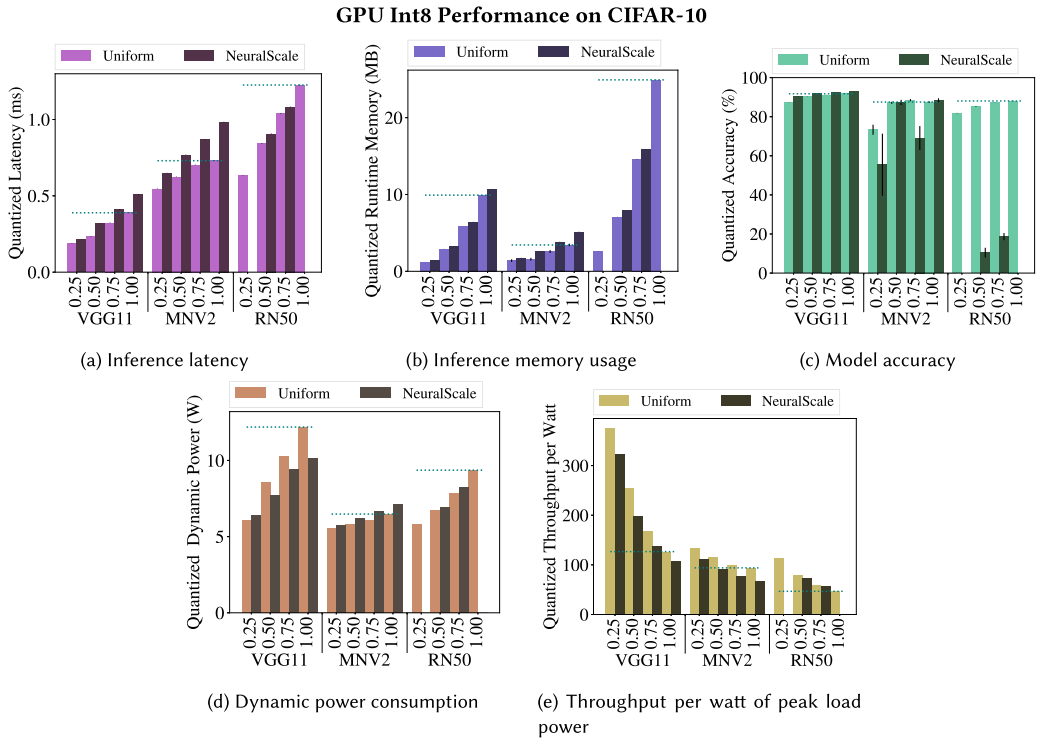
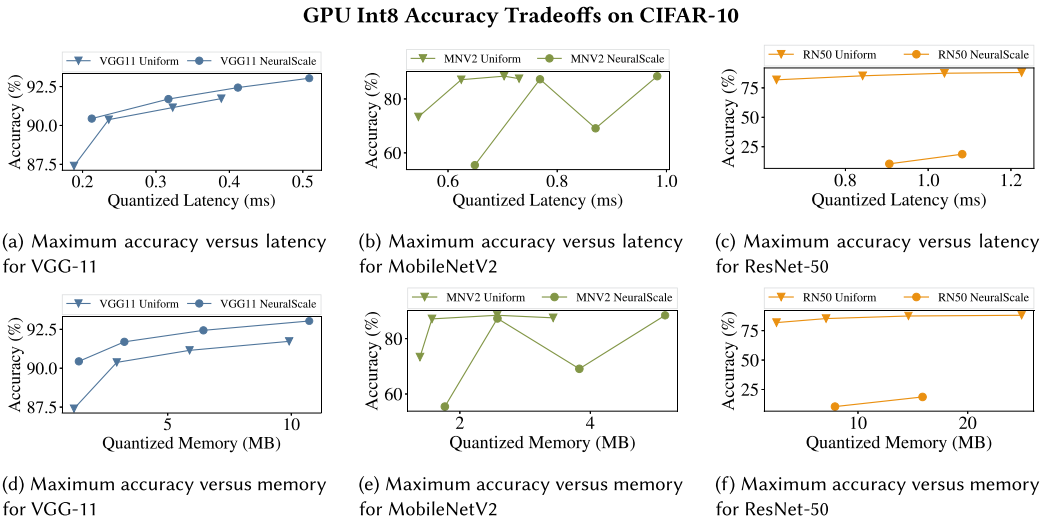


Fig. 11. NVIDIA Jetson AGX Xavier float32/int8 model performance using TensorRT. TensorRT failed to generate models for NeuralScale 0.25× and 1.00× ResNet-50 due to calibration and memory errors, respectively. The low accuracy performance of the other NeuralScale ResNet-50 models is likely caused by the change in model structure due to the NeuralScale process. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

caused by the model better utilizing the device, as uniform scaling is more efficient in terms of throughput per watt, shown in Figure 11(e). Similar to the float32 models, the uniform-scaling MobileNetV2 and ResNet-50 models consistently use less power and are more efficient than the NeuralScale versions of these models.

When quantized, Figure 12(a) indicates that uniform scaling and NeuralScale have similar accuracy for each millisecond of latency for VGG-11. Figures 12(b),12(c), however, show that uniform scaling outperforms NeuralScale in terms of accuracy versus latency on MobileNetV2 and ResNet-50. The results are similar for accuracy versus memory as Figure 12(d) shows NeuralScale outperforming uniform scaling for VGG-11 and the opposite for MobileNetV2, Figure 12(e), and ResNet-50, Figure 12(f). Also note that for NeuralScale 0.75× MobileNetV2, the tools struggled to quantize as well, leading to a drop in accuracy. This result indicates that QAT would likely be needed to improve accuracy performance.

Figure 13 shows the memory usage per parameter for all models on the AGX Xavier. Note the log scale on the y-axis. With float32 models, NeuralScale uses an average of 4.6× the runtime memory of uniform-scaling models. For the int8 models, the increase is not nearly as large at 1.3× the memory usage. Essentially, on the GPU, the modified parameter layout of the NeuralScale models becomes less efficient than simply scaling the layers uniformly with float32 values. This result is further evidence that, especially on the GPU, simply scaling to a set number of parameters does



(a) Maximum accuracy versus latency for VGG-11

(b) Maximum accuracy versus latency for MobileNetV2

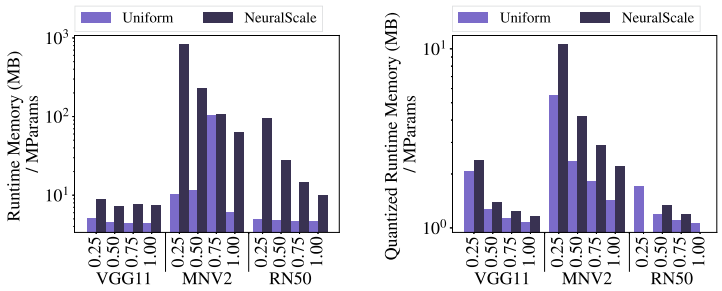
(c) Maximum accuracy versus latency for ResNet-50

(d) Maximum accuracy versus memory for VGG-11

(e) Maximum accuracy versus memory for MobileNetV2

(f) Maximum accuracy versus memory for ResNet-50

Fig. 12. NVIDIA Jetson AGX Xavier float32/int8 accuracy tradeoffs using TensorRT.



(a) AGX unquantized memory usage per parameter

(b) AGX quantized memory usage per parameter

Fig. 13. AGX memory usage per parameter. Note the log scale on the y-axis.

not guarantee equivalent or efficient runtime-memory usage when using float32 values, while the trend observed on the CPU showed parameter count correlated well with memory usage.

Another result that is important to note is that neither of the two working NeuralScale ResNet-50 models, 0.50× and 0.75×, could be quantized with high accuracies shown in Figure 11(c). Their uniform-scale counterparts had no quantization issues. Even when we attempted to use all 50,000 training images for the PTQ calibration dataset, the achieved accuracy was only 25% for the NeuralScale 0.75× model. Given this result, uniform-scaled models are the better option when using quantization on the AGX Xavier.

4.3.2 Comparing Scaling Methods with Two Datasets on MobileNetV2. When comparing the two scaling methods across the two datasets, uniform-scaling float32 models continue to be faster than their NeuralScale counterparts, displayed in Figure 14(a). For CIFAR-10, the NeuralScale models are an average of 1.6× slower. With the larger images of tinyImageNet, NeuralScale models are 1.8× slower than their uniform-scaling counterparts.

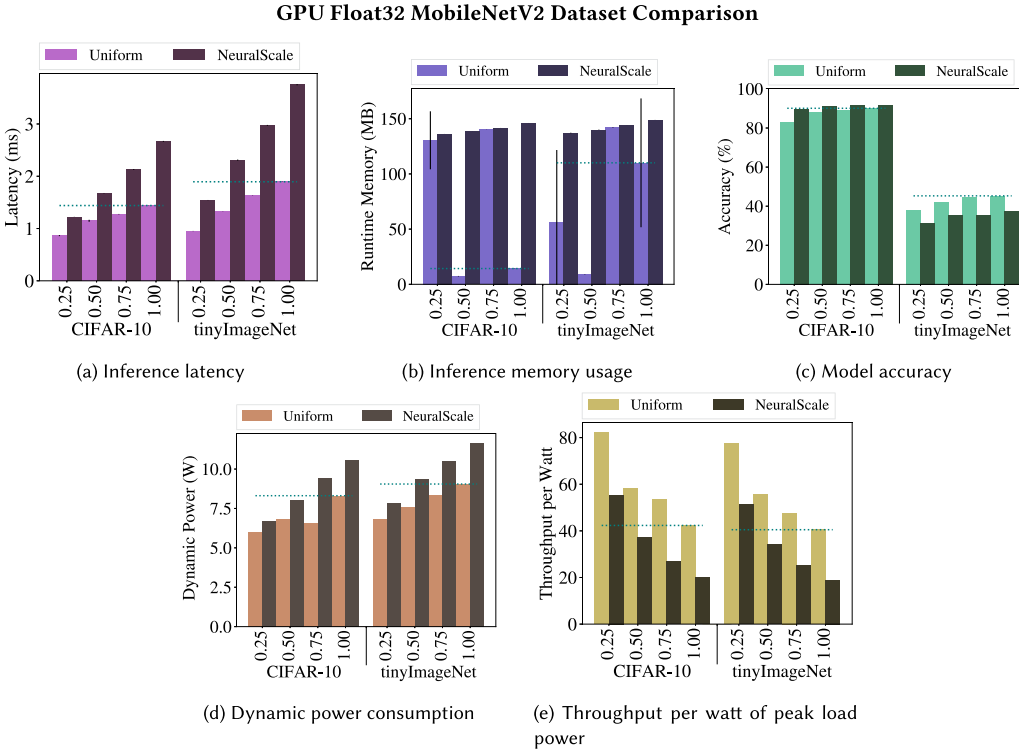


Fig. 14. NVIDIA Jetson AGX Xavier float32 MobileNetV2 dataset comparison using TensorRT. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

The estimated runtime-memory usage for the tinyImagNet MobileNetV2 model contains similar variations to those seen with the CIFAR-10 models, Figure 14(b). These results indicate that the memory utilization can vary in terms of activation memory when using MobileNetV2 with this version of TensorRT. However, for models with lower variation, uniform-scaling models tend to use less memory than the NeuralScale models.

In terms of power utilization, the dataset does not change the relationship between NeuralScale and uniform scaling, Figure 14(d). Once again, the uniform-scaling models consistently use less power than their NeuralScale counterparts. However, the dataset itself can change dynamic power as the tinyImageNet models use more power than the CIFAR-10 models. With a more complex dataset, the models have more operations to do, likely leading to an increase in power. In terms of throughput per watt, Figure 14(e), uniform scaling is more efficient than NeuralScale. Additionally, performance across datasets is similar.

Finally, the results for the int8 models are shown in Figure 15. NeuralScale models are an average of 1.2× and 1.4× slower for CIFAR-10 and tinyImageNet, respectively. Additionally, there was significant memory variation, though NeuralScale models consistently use more memory than uniform scaling. Finally, power results are similar to the float32 models with approximately similar throughput per watt between the scaling methods.

4.3.3 GPU Summary. On the GPU, the uniform-scaling models are consistently faster than their NeuralScale counterparts. Estimated memory utilization can vary based on the implementation chosen by TensorRT. Additionally, power utilization is lower for uniform scaling, which also achieves higher throughput per watt.

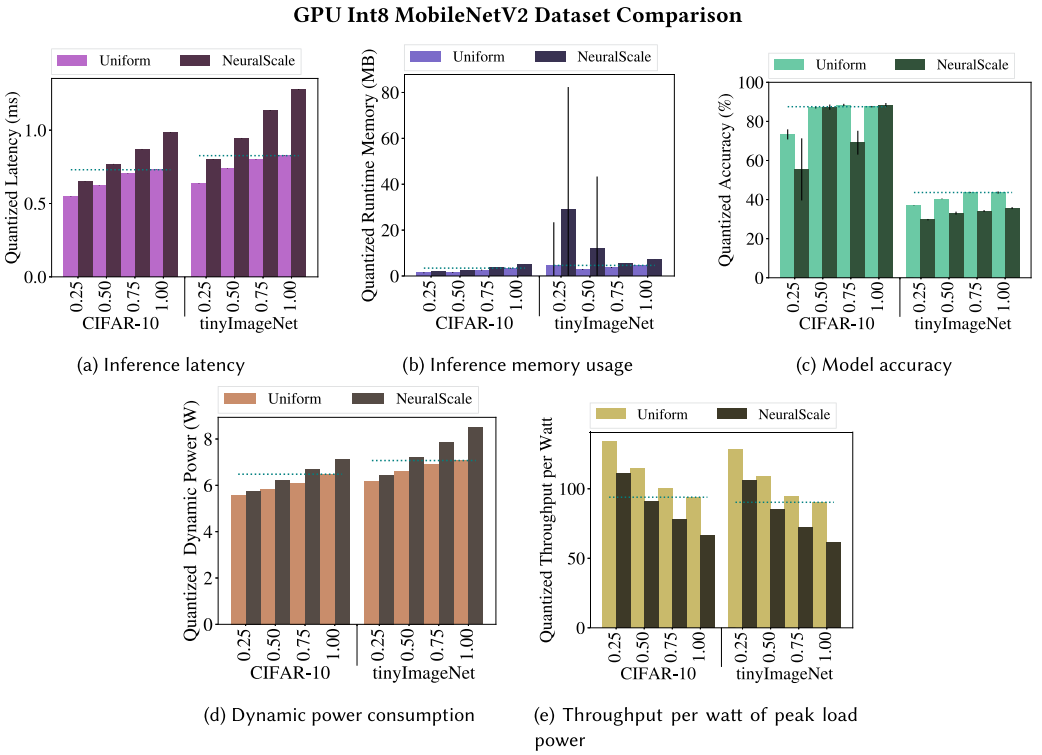


Fig. 15. NVIDIA Jetson AGX Xavier float32/int8 MobileNetV2 Dataset Comparison using TensorRT. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

4.4 FPGA Results

This subsection presents the results gathered on the Xilinx ZCU104. Again, we first present results comparing the scaling methods across three models with CIFAR-10, followed by results across datasets and a device summary. For the FPGA device, the idle power was measured to be 16.6 W.

4.4.1 Comparing Scaling Methods with Three Models on CIFAR-10. The Xilinx ZCU104 using Vitis AI and Xilinx's DPU only support int8-based models. Inference results are shown in Figure 16. For VGG-11 and MobileNetV2, the uniform-scaling models are slightly faster than the NeuralScale models, displayed in Figure 16(a). However, for ResNet-50, the performance of some of the uniform-scaling models, 0.25× and 0.50×, is slightly slower than the NeuralScale models. Overall, the number of parameters in the model is highly correlated with the runtime on the FPGA with a Pearson correlation of 0.924 (p=1.2 · 10^{-10}). This result indicates that parameter scaling is an appropriate technique for FPGA-based devices.

Given that the ZCU104 FPGA fabric is not reconfigured to run different models and always has the same DPU-based bitstream, memory measurements were less sensible for this device. On the FPGA, the accuracy performance of the generated models is shown in Figure 16(b). The NeuralScale ResNet-50 models did not reach the expected accuracies. Based on these results, the structure of the NeuralScale models was more challenging for Vitis AI to quantize satisfactorily using PTQ. Techniques like QAT are therefore needed for improved accuracy performance with NeuralScale.

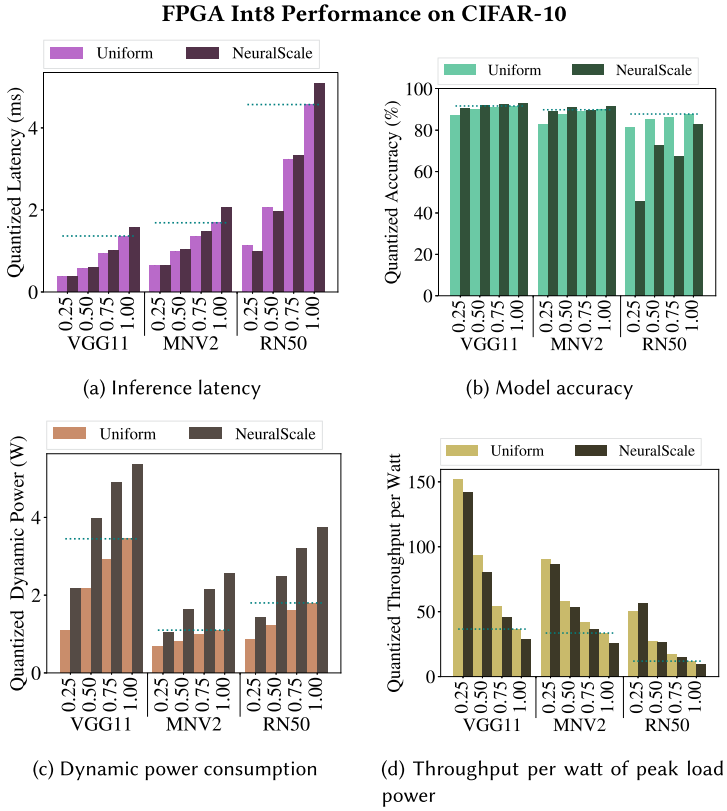


Fig. 16. Xilinx ZCU104 int8 model performance using Vitis AI and Xilinx's DPU IP-core. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

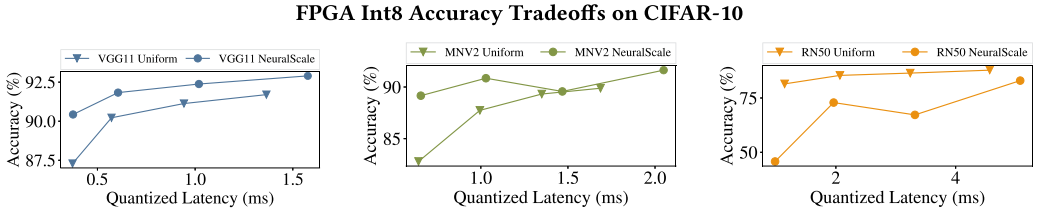


Fig. 17. Xilinx ZCU104 int8 model accuracy tradeoffs using Vitis AI and Xilinx's DPU IP-core.

Next, the power results are presented in Figure 16(c). The dynamic-power trends on the FPGA are again similar to the other devices with uniform scaling consistently outperforming NeuralScale. However, in terms of throughput per watt, Figure 16(d), there is no clear advantage between uniform scaling and NeuralScale. The efficiency of the models tends to be very similar, with NeuralScale being more efficient than uniform scaling for some of the ResNet-50 models.

Finally, we compare the accuracy versus latency results for the ZCU104. Figure 17(a) shows NeuralScale outperforming uniform scaling as it is always more accurate for the same latency on VGG-11. For MobileNetV2, at the two smaller scaling ratios, Figure 17(b) indicates that NeuralScale

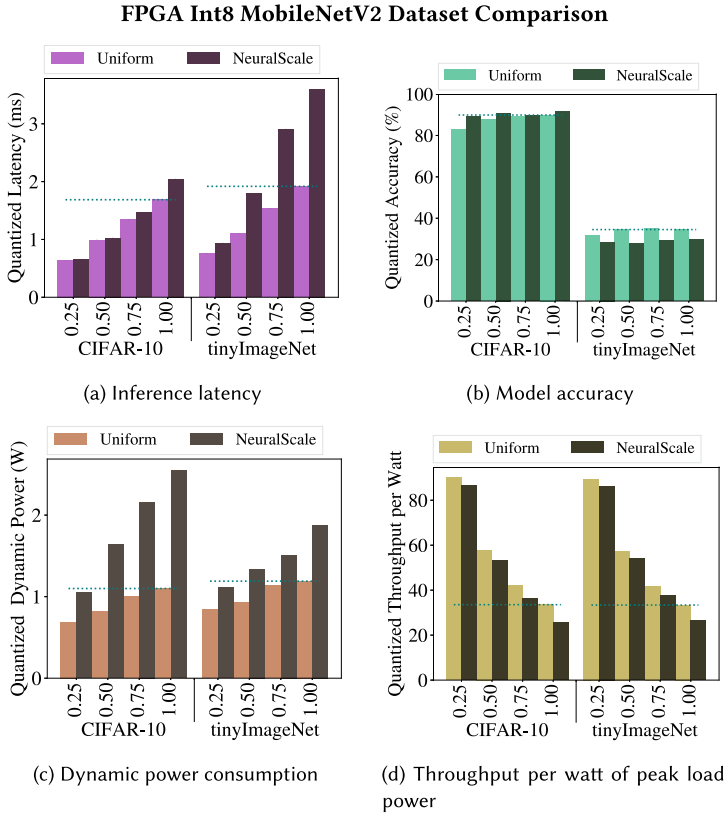


Fig. 18. Xilinx ZCU104 int8 dataset comparison using Vitis AI and Xilinx's DPU IP-core. The dashed teal lines denote the performance of the original, unscaled models (equivalent to uniform scaling 1.00×).

is more accurate than uniform scaling, though performance becomes similar at larger values. Given that the latencies of these models are only a minuscule part of the overall real-time processing budget, it is evident that NeuralScale models are the best option for FPGAs. Finally, Figure 17(c) indicates that uniform scaling is once again faster with higher accuracies than NeuralScale on ResNet-50.

4.4.2 Comparing Scaling Methods with Two Datasets on MobileNetV2. The results comparing scaling methods on MobileNetV2 across datasets are shown in Figure 18. In terms of latency, the NeuralScale models are more impacted by the growth in image size of tinyImageNet compared with CIFAR-10. NeuralScale models are only 1.1× slower than their uniform-scaling counterparts on CIFAR-10. However, with the larger image size and additional growth in operations, they are 1.6× slower than uniform scaling on tinyImageNet. These results indicate that the growth in image size was more detrimental for the more complicated NeuralScale models.

Finally, in terms of dynamic power, NeuralScale models continue to use more power than uniform scaling, Figure 18(c). Interestingly, the dynamic power of the tinyImageNet models is lower than the CIFAR-10 models. In terms of throughput per watt, the dataset does not have a large impact on the scaling methods as efficiency is similar.

4.4.3 FPGA Summary. For the FPGA, NeuralScale models are generally slower than uniform-scaling models, with larger images slowing down NeuralScale models more. However, latency

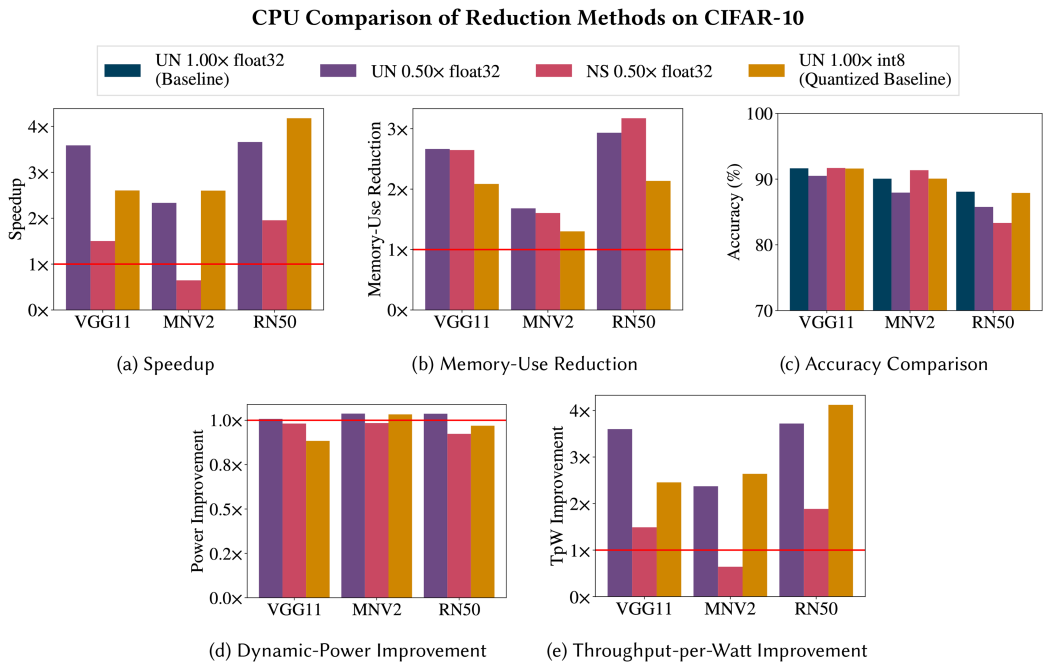


Fig. 19. Using ARM Cortex-A72, comparison of the performance of select models to the original model (Uniform scaling, 1.00×, float32).

and parameter count have a large correlation when analyzed on the CIFAR-10 dataset. Uniform-scaling models use less power. However, depending on the model, throughput per watt is relatively consistent between the two scaling methods.

4.5 Parameter Scaling versus Quantization Performance on CIFAR-10

In this subsection, we compare an original float32 model to uniform and NeuralScale float32 models at 0.50×. We also compare these models to an int8 version of the original model. We chose a scaling ratio of 0.50× for this case study as it represents the middle of the range for the ratios tested. We calculate the speedup for each of the optimized models over the original float32 model. We also compare the reduction in memory usage by dividing the original model's memory usage by the value for each of the three optimizations. The dynamic-power improvement is also compared by dividing the original model's dynamic power by the reduced model's power. As a reminder, the dynamic power is the peak load power minus the idle power. The improvement to the **throughput per watt** (**TpW**) is also presented. Additionally, we report the accuracy of all models in these comparisons. Please note that the accuracy range depicted on the charts in this section is reduced to 70–100% to make it easier to compare the variations between the optimization methods.

Figure 19 shows these results on the ARM Cortex-A72 device. The NeuralScale models see the least improvement in latency, including a slowdown for the MobileNetV2 model. Uniform scaling improves the latency more than quantization for VGG-11, but quantization has a larger speedup for MobileNetV2 and ResNet50. Across the three models tested, quantization improves latency performance by 3.1×. Uniform scaling improves latencies by slightly more at 3.2×, while NeuralScale only improves latencies by 1.4×. However, in terms of memory usage, Figure 19(a), both NeuralScale and uniform scaling have a larger improvement. In terms of dynamic power, none of the reduction methods provide a significant improvement, Figure 19(c). Finally, in terms of throughput

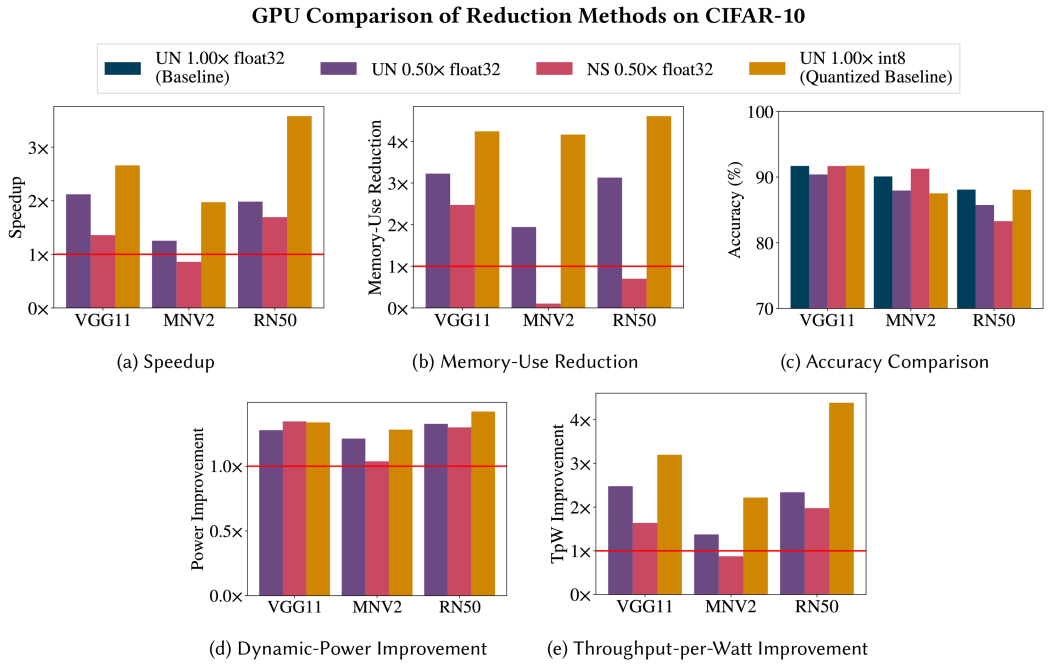


Fig. 20. Using the Jetson AGX Xavier, comparison of the performance of select models to the original model (Uniform scaling, 1.00×, float32).

per watt, shown in Figure 19(d), NeuralScale is consistently the worst. Uniform scaling performs best with VGG-11, but quantization produces the best improvement for the other two models. These results indicate that quantization is generally better for improving latencies on the ARM Cortex-A72 while the scaling methods are better for improving memory usage.

On the AGX Xavier, quantization always outperforms the scaling methods in terms of latency, Figure 20(a), and memory usage, Figure 20(b). Latencies are improved by 2.7× with quantization alone, while uniform scaling improves performance by 1.8×, and NeuralScale improves by 1.3×. Dynamic power is improved by 1.3× with quantization. With uniform scaling it is also improved by 1.3×, while with NeuralScale it is improved by 1.2×. Note that if the peak load power were to be considered, these differences would be smaller. Additionally, quantization always produces the largest improvement to throughput per watt, as shown in Figure 20(e). These results indicate that TensorRT and the AGX are well optimized for quantization. There is also minimal overall change to accuracy, shown in Figure 20(c), though a more complex scheme like QAT could be used to improve the performance of the quantized MobileNetV2 model.

Essentially, quantization was able to achieve similar or better results to that of both uniform scaling and NeuralScale at a ratio of 0.50× on the three models tested. This result indicates that quantization can be considered as an alternative to parameter scaling for model deployment. It is equally important, therefore, to consider the practicality of training models with the two different optimizations to decide which tool should be used first, discussed in Section 5.5.

5 DISCUSSION

The discussion section highlights the main lessons learned from this research. First, insight is provided as to why NeuralScale models have significantly more FLOPs than their uniform-scaling counterparts. Next, we discuss tradeoffs for the accuracy of the models in this study. This

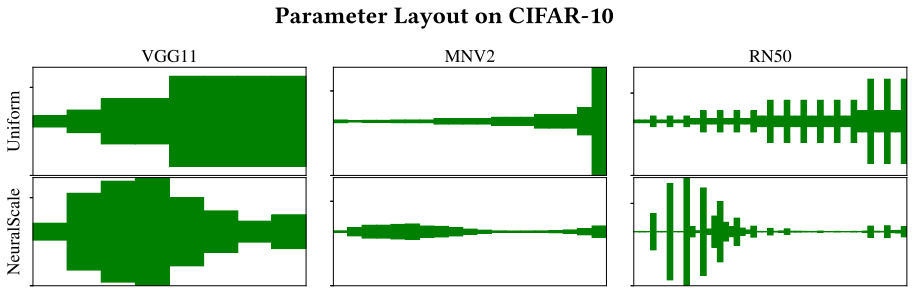


Fig. 21. Layer widths at 1.00× scaling for models trained on CIFAR-10. *Y*-axis represents layer widths, *x*-axis represents depth in network.

discussion is followed by details about selecting a scaling ratio and the challenges of training scaled models. Finally, we provide a discussion of lessons for using parameter scaling when deploying models.

5.1 Model Layout

One of the interesting aspects of the NeuralScale method is how it adjusts the widths of the layers in models, as shown in Figure 21. In this figure, the *y*-axis represents the width of a layer, while the *x*-axis represents the layer number, or depth, in the network. The layer widths are shown for a scaling ratio of 1.00× as this ratio depicts the model layout with the number of parameters chosen by the original model designers. As the scaling ratio is 1.00×, the uniform-scaling method simply returns the original network layout.

Figure 21 is insightful because it shows the structure for a more efficient architecture, in terms of accuracy per parameter, as determined by NeuralScale. NeuralScale tends to shift the parameters from the end of the network to the beginning or middle. In CNNs, earlier layers capture more general features like textures while later layers capture more specific details [41]. Figure 21 displays how NeuralScale infers that general features are more useful on CIFAR-10. This result is interesting because most of the models as designed by the original sources tend to place more parameters near the end of the network.

A possible reason why NeuralScale may shift so many parameters to the beginning of the network is that when it trains and attempts to prune an early layer, it discovers a larger drop in accuracy, and therefore identifies it as more important. As these earlier layers tend to hold more fundamental shapes [41], removing one of them would likely remove a main component of higher-level features. NeuralScale perhaps overparameterizes the early layers even further as it continues to place more emphasis on having duplicates of more fundamental features rather than learning more complex high-level features. We also hypothesize that the vastly different model layouts, especially for ResNet-50, meant that some of the quantization tools struggled with the NeuralScale versions of these models. Given that the uniform-scaled ResNet-50 models did not struggle to quantize with ONNX Runtime, TensorRT, or Vitis AI, it seems likely that the non-standard structure in the NeuralScale models made it more difficult for these tools to perform PTQ well.

For the NeuralScale models, there is an observed large increase in FLOPs for the same parameter count as uniform scaling. This result can be explained by the relationship between FLOPs and the model layout. Earlier in the network, there is also a larger spatial dimension. The relationship of FLOPs in a convolutional layer is on the order of $O(n^2)$ to the spatial dimension due to the height and width of the image factoring into the FLOPs calculation. Therefore, there is a large increase in the number of FLOPs to be performed when more of the parameters are used for these earlier filters.

5.2 Accuracy Tradeoffs

Given that the same hyperparameters were used for training all models, we cannot guarantee that all models are the most accurate possible. However, it is also important to note that several tools, namely TensorRT and Vitis AI, failed to generate accurate models with NeuralScale ResNet-50 models when attempting quantization with CIFAR-10. Therefore, extra care needs to be taken in the form of QAT to try to improve performance. Doing so, though, will result in increased complexity, which shows the value of a simpler scaling method like uniform scaling.

Accuracy in this study is further limited because parameter scaling often removes the ability to perform transfer learning. When using transfer learning, weights from a model pre-trained on a more complex dataset like ImageNet [5] are used as an initialization for the model being trained on a new dataset. However, without using standard model sizes, it is often difficult to use pretrained weights to help improve the accuracy of a model. There are several methods [4, 8] that attempt to transfer weights between models of different sizes, but these methods have other limitations and are outside the scope of this study. Not using transfer learning can increase the challenge of creating the most accurate model for a given task.

Additionally, NeuralScale ResNet-50 models were less accurate than their uniform-scaling counterparts. This result is perhaps most interesting because NeuralScale is designed to produce more accurate models per parameter, but it failed to do so on CIFAR-10 in this study. Therefore, the additional complexity of the NeuralScale pipeline is not always justified. NeuralScale was also less accurate than uniform scaling on MobileNetV2 with tinyImageNet, differing from the results of Reference [18]. While additional accuracy could likely be gained via an exhaustive hyperparameter search, this would be prohibitively expensive given the training times of architecture descent and NeuralScale as discussed in Section 5.4. It also shows that NeuralScale is sensitive to hyperparameter selection in practice, making it more difficult to recommend for other models and datasets due to these tradeoffs.

Finally, we consider VGG-11 and MobileNetV2, which both showed consistent accuracy versus latency tradeoffs on our test platforms with CIFAR-10. On the embedded CPU, when targeting latency, both methods produced similar results. When targeting runtime-memory utilization, NeuralScale outperforms uniform scaling. For the embedded GPU, again both methods perform similarly in terms of latency, while results are mixed for memory utilization. Finally, on the embedded FPGA, NeuralScale slightly outperforms uniform scaling with generally higher accuracies at the same latency.

5.3 Scaling Ratio Selection

A major challenge with parameter scaling methods for deployment is setting the scaling ratio. While a smaller ratio reduces the parameter count, in the case of NeuralScale, it does not necessarily lead to better on-device performance. One of our main contributions is, therefore, guidance on setting a scaling ratio based on the empirical results of this study.

Table 4 shows the estimated linear relationship between *latency* and the number of parameters, while Table 5 shows the estimated linear relationship between *device memory* and parameter count. Using one of the equations from the tables, we can solve for a scaling ratio, S, with Equation (1). P_t represents the target parameter count, which can be estimated using the tables. In this equation, y represents the target latency (in ms) or memory usage (in MB). P_o is the value of the parameter count in the original model.

$$S \approx \sqrt{\frac{P_t}{P_o}} = \sqrt{\frac{(y-b)/m}{P_o}}. \tag{1}$$

Table 4. Linear Equations (y = mx+b) for Estimating Latency Given a Parameter Count

Device	Precision	Pearson Correlation	Slope (m)	Bias (b)
CPU	float32	0.786 ($p = 5.2 \cdot 10^{-6}$)	$5.7 \cdot 10^{-6}$	17.9
	int8	0.648 ($p = 6.1 \cdot 10^{-4}$)	$2.2 \cdot 10^{-6}$	9.7
GPU	float32	0.850 ($p = 1.5 \cdot 10^{-7}$)	$1.6 \cdot 10^{-7}$	$9.6 \cdot 10^{-1}$
	int8	0.546 ($p = 8.6 \cdot 10^{-3}$)	$2.8 \cdot 10^{-8}$	$5.1 \cdot 10^{-1}$
FPGA	int8	0.924 ($p = 1.2 \cdot 10^{-10}$)	$1.7 \cdot 10^{-7}$	$6.9 \cdot 10^{-1}$

Results assume 32 × 32-pixel images, such as CIFAR-10. (y = latency (ms), x = parameters).

Table 5. Linear Equations (y=mx+b) for Estimating Memory Usage Given a Parameter Count

Device	Precision	Pearson Correlation	Slope (m)	Bias (b)
CPU	float32	0.999 ($p = 1.6 \cdot 10^{-30}$)	$4.2 \cdot 10^{-6}$	9.3
	int8	0.991 ($p = 8.6 \cdot 10^{-21}$)	$1.8 \cdot 10^{-6}$	9.9
GPU	float32	0.391 ($p = 5.9 \cdot 10^{-2}$)	$4.0 \cdot 10^{-6}$	62.5
	int8	0.994 ($p = 7.8 \cdot 10^{-21}$)	$1.0 \cdot 10^{-6}$	1.3

Results assume 32 × 32-pixel images, such as CIFAR-10. (y = memory (MB), x = parameters).

It is, of course, important to note the Pearson correlation coefficients between the latency and parameters and memory utilization and parameters. Given a lower Pearson correlation coefficient, the less useful the linear equation will be at estimating a scaling ratio. In terms of latency requirements and estimating a scaling ratio, the linear equations are best for estimating float32 performance on the GPU and int8 performance on the FPGA. For targeting a set memory utilization, the linear equations are useful at both precisions on the CPU and with int8 on the GPU. Given these equations, and the accompanying correlation values, we can now estimate a target scaling ratio given a latency or memory constraint.

To compare the usability of these equations, we performed a brief case study with the CIFAR-100 dataset [15], which is similar to CIFAR-10 but instead contains 100 classes. We attempted to achieve real-time performance with uniform scaling on the ARM Cortex-A72 embedded CPU. The result of Equation (1) is a scaling ratio of ≈ 0.82. Therefore, we chose a slightly more conservative ratio of 0.80×. We then trained a MobileNetV2 model with uniform scaling at this ratio and also attempted quantization on the unscaled model. Once the models were trained, we found that the uniform scaling achieved a latency of 18.2 ms and an accuracy of 64 %. Therefore, even with an approximation for the optimal scaling ratio, we still failed to meet real-time performance of 16.67 ms and would need to attempt training again with a smaller scaling ratio. On the other hand, the quantized base MobileNetV2 model reached real-time performance on CIFAR-10 and continued to do so with CIFAR-100 with a latency of 9.1 ms and an accuracy of 65.3 %, an improvement over the float32 uniform-scaling model at a ratio of 0.80×. Again, we see the challenges of using scaling methods for deployment. If quantization alone is not capable of meeting the latency requirements, scaling is a useful tool. However, we conclude that quantization remains the best first step for improving runtime performance.

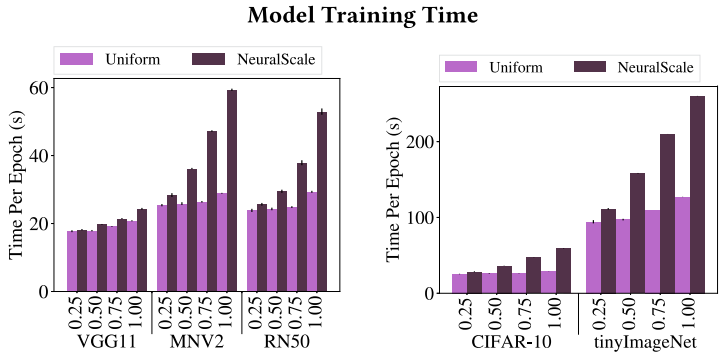
5.4 Training Challenges

While this article is focused on the deployment performance of scaled models, it is also important to note the changes to the training process when using these scaling methods. NeuralScale includes

Table 6. Total Time to Run Architecture Descent on Each Model for the 15 Iterations

Model	Dataset	Time
VGG-11	CIFAR-10	5h 22m 57s
MobileNetV2	CIFAR-10	7h 57m 52s
MobileNetV2	tinyImageNet	1d 6h 5m 52s
ResNet-50	CIFAR-10	1d 18h 1m 51s

Training times measured on an NVIDIA GTX 1080 Ti.



(a) Training times across models on CIFAR-10

(b) Training times on MobileNetV2 for CIFAR-10 and tinyImageNet

Fig. 22. Averaging training time per epoch of each model utilizing an NVIDIA GTX 1080 Ti, averaged over 30 epochs.

the pre-scaling step of architecture descent, which can be costly depending on the number of parameters in a model. Architecture descent is dependent upon a dataset, and therefore, cannot be performed once per model, but needs to be performed once per dataset+model combination. For reference, Table 6 shows the times for architecture descent for each of the three models on CIFAR-10 and the time of MobileNetV2 with tinyImageNet. As is evident on MobileNetV2, the larger image sizes have a big impact on architecture descent time as the tinyImageNet dataset is 3.8× slower than the CIFAR-10 version. ResNet-50 has significantly more parameters and is theoretically therefore a model that would be more likely to be scaled to achieve better on-device performance. However, having more parameters leads to much longer pruning times with architecture descent.

For both uniform scaling and NeuralScale, the models must be completely retrained if changing the scaling ratio or changing the dataset. This could be costly if you do not know what scaling ratio to use, though Section 5.3 provides intuition for selecting a value. Figure 22 shows the average training time per epoch for each of the models. Note that since the NeuralScale models tend to have more FLOPs, they are significantly costlier to train than their uniform-scaling counterparts. For example, with 1.00× MobileNetV2, the NeuralScale model takes over 2.05× as long to train as the uniform-scaling model on CIFAR-10. This tradeoff in training time is important to consider depending on how many scaling ratio combinations may need to be tested to reach a target latency or memory utilization and how often retraining is expected.

5.5 Recommendations for Model Deployment

When considering model deployment, reduction methods aim at improving a model's inference performance. As studied in this research, the results of the parameter scaling reduction method can

be of mixed utility. A simple version like uniform scaling tends to give good latency and memory improvements over the original model, but often at the cost of accuracy. NeuralScale generates models that can be more accurate than the original but can perform worse in terms of latency and memory usage.

One of the main challenges of parameter scaling is that training is performed after scaling the model. Other reduction methods like structured pruning use a fully trained model that is then pruned to reduce the model size [7]. This difference means that training techniques like transfer learning can still be effectively used for achieving more accurate models.

In contrast to parameter scaling and similar to structured pruning, quantization is also done after training the model. As shown in Figure 19, quantization is able to achieve similar speedups to uniform scaling and better speedups compared with NeuralScale on the CPU, though it is not as efficient in terms of memory usage. On the GPU in Figure 20, quantization is always better than the two scaling methods in terms of speedup and memory-use reduction. All three of the frameworks tested in this research had native support for quantization through PTQ. While parameter scaling requires changing the model definition in the PyTorch code, quantization can be easily used to reduce the size of a model. Quantization is therefore a natural first reduction method to use since parameter scaling and structured pruning both require additional setup in the model definition to enable the removal of filters.

6 CONCLUSION

Computer-vision models have continued to grow in size and complexity, while the desire to deploy models on resource-constrained, embedded devices increases as machine learning is deployed on edge devices. Therefore, it is important to explore methods that can help achieve better real-time performance across devices. Parameter-scaling methods aim at adjusting the size of these large and complex vision models by reducing the number of filters in the layers of the model. Uniform scaling, originally used in MobileNets [12], uniformly multiplies the number of filters in a layer by a consistent scaling value. NeuralScale, proposed in [18], attempts to fine-tune this process to determine the importance of the different layers, allowing it to adaptively adjust the widths of the layers. This research analyzed the performance of these models on embedded CPU, GPU, and FPGA systems.

Models scaled using NeuralScale tend to be slower than their uniform-scaling counterparts on the CPU and GPU testbeds. This slowdown is because NeuralScale tends to shift parameters earlier in the model, resulting in more computationally intensive inference. On CIFAR-10, NeuralScale models have 6.4× more FLOPs on average. Combining parameter scaling with quantization enables additional performance gains. For VGG-11 and MobileNetV2, uniform scaling and NeuralScale tend to have similar accuracy versus latency trends with CIFAR-10, showing how the increase in accuracy for NeuralScale comes at the cost of latency reductions. On the FPGA testbed, both methods performed similarly, with NeuralScale achieving higher accuracies for the same latencies on VGG-11 and MobileNetV2. When analyzing the tinyImageNet results, we found that NeuralScale is very sensitive to hyperparameter selection, which can reduce the achieved accuracy. Additionally, the increased computational intensity of the NeuralScale models reduces its utility for providing improved performance.

Averaging over all the models tested on CIFAR-10, on the embedded CPU, NeuralScale models perform inference at a speed that is 2.9× *slower* than their uniform-scaling counterparts while consuming the same amount of memory. On the embedded GPU, NeuralScale models are 1.3× *slower* than their uniform-scaling counterparts, while they also consume 2.0× *more* memory. Inference speed on the FPGA is the same on average. When optimizing for power, uniform scaling generally produces models that require less power, though results are mixed for VGG-11. Overall, we

recommend using NeuralScale with VGG-11 and MobileNetV2 and uniform scaling with ResNet-50 to realize the most accurate models for CIFAR-10. On tinyImageNet, however, we found that our uniform-scaling versions were more accurate than NeuralScale, differing from the accuracies achieved in [18]. While likely a result of non-optimal hyperparameters, it highlights the sensitivity of the NeuralScale training process.

Between quantization and parameter scaling, this research demonstrated that quantization is the best first step for model deployment on resource-constrained platforms. When comparing the results on CIFAR-10, on the CPU, quantization alone *reduces* latency by 3.1× and memory consumption by 1.8× over the original model while having similar dynamic-power consumption. On the GPU, quantization *reduces* latency by 2.7×, memory consumption by 4.3×, and dynamic-power consumption by 1.3× over the original model. It also easily enables FPGA-based inference using tools like Vitis AI and Xilinx's DPU. If additional latency performance is needed, combining quantization with parameter scaling can result in faster models, but doing so may prevent the use of other training techniques that enable more accurate models, such as transfer learning. Overall, uniform scaling tends to be the easier and more viable method for improved latency and memory performance, while NeuralScale can produce more accurate models but adds additional complexity to the training process and additional operations to the models.

ACKNOWLEDGMENTS

We would also like to acknowledge the students of SHREC, past and present, for their assistance and guidance with this journal article. Specifically, we thank Dr. David Langerman, Evan Gretok, Marika Schubert, and Dr. Seth Roffe.

REFERENCES

[1] ONNX Developers. 2019. ONNX: Open Neural Network Exchange. Retrieved from https://github.com/onnx/onnx

[2] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-all: Train One network and specialize it for efficient deployment. In *Proceedings of the International Conference on Learning Representations*. International Conference on Learning Representations, ICLR, Addis Ababa, Ethiopia, 1–15. Retrieved from https://github.com/mit-han-lab/once-for-all

[3] Han Cai, Ligeng Zhu, and Song Han. 2019. Proxylessnas: Direct neural architecture search on target task and hardware. In *Proceedings of the 7th International Conference on Learning Representations, ICLR 2019*. International Conference on Learning Representations, ICLR, New Orleans, Louisiana, USA, 1–13.

[4] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. 2016. Net2Net: Accelerating learning via knowledge transfer. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*. International Conference on Learning Representations, ICLR, San Juan, Puerto Rico, 1–12.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Miami, FL, 248–255. DOI: https://doi.org/10.1109/CVPR.2009.5206848

[6] Misha Denil, Babak Shakibi, Laurent Dinh, MarcAurelio Ranzato, and Nando de Freitas. 2013. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*. C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc., Lake Tahoe, 2148–2156. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2013/file/7fec306d1e665bc9c748b5d2b99a6e97-Paper.pdf

[7] Xiaohan Ding, Tianxiang Hao, Jianchao Tan, Ji Liu, Jungong Han, Yuchen Guo, and Guiguang Ding. 2021. ResRep: Lossless CNN pruning via Decoupling Remembering and Forgetting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. IEEE, Online, 4510–4520. Retrieved from https://openaccess.thecvf.com/content/ICCV2021/html/Ding_ResRep_Lossless_CNN_Pruning_via_Decoupling_Remembering_and_Forgetting_ICCV_2021_paper.html?ref=https://githubhelp.com

[8] Jiemin Fang, Yuzhu Sun, Qian Zhang, Kangjian Peng, Yuan Li, Wenyu Liu, and Xinggang Wang. 2021. FNA++: Fast network adaptation via parameter remapping and architecture search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 9 (Sept. 2021), 2990–3004. DOI: https://doi.org/10.1109/TPAMI.2020.3044416

[9] Calvin B. Gealy and Alan D. George. 2022. Evaluation of parameter-scaling for efficient deep learning on small satellites. In *Proceedings of the AIAA/USU Conference on Small Satellites*. USU, Logan, Utah, 1–14. Retrieved from https://digitalcommons.usu.edu/smallsat/2022/all2022/348/

[10] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2022. *A Survey of Quantization Methods for Efficient Neural Network Inference* (1 ed.). Chapman and Hall/CRC, Boca Raton, 291–326. DOI: https://doi.org/10.1201/9781003162810-13

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 2016-Decem. IEEE Computer Society, Las Vegas, Nevada, USA, 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90

[12] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861. Retrieved from https://arxiv.org/abs/1704.04861

[13] Vinod Kathail. 2020. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Seaside CA USA, 173–174. DOI: https://doi.org/10.1145/3373087.3375887

[14] Sameh Khamis, Sean Fanello, Christoph Rhemann, Adarsh Kowdle, Julien Valentin, and Shahram Izadi. 2018. StereoNet: Guided hierarchical refinement for real-time edge-aware depth prediction. In *Computer Vision—ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Vol. 11219. Springer International Publishing, Cham, 596–613. DOI: https://doi.org/10.1007/978-3-030-01267-0_35

[15] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2009. CIFAR-10 and CIFAR-100 datasets. Retrieved from https://www.cs.toronto.edu/~kriz/cifar.html

[16] David Langerman, Alex Johnson, Kyle Buettner, and Alan D. George. 2020. Beyond floating-point ops: CNN performance prediction with critical datapath length. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–9. DOI: https://doi.org/10.1109/HPEC43674.2020.9286182

[17] Eugene Lee and Chen-Yi Lee. 2020. GitHub—NeuralScale: Efficient Scaling of Neurons for Resource-Constrained Deep Neural Networks. Retrieved from https://github.com/eugenelet/NeuralScale

[18] Eugene Lee and Chen-Yi Lee. 2020. NeuralScale: Efficient scaling of neurons for resource-constrained deep neural networks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Seattle, WA, USA, 1475–1484. DOI: https://doi.org/10.1109/CVPR42600.2020.00155

[19] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. 2017. tinyImageNet from Stanford University CS231n. Retrieved from http://cs231n.stanford.edu/tiny-imagenet-200.zip

[20] Shanchuan Lin, Andrey Ryabtsev, Soumyadip Sengupta, Brian Curless, Steve Seitz, and Ira Kemelmacher-Shlizerman. 2021. Real-time high-resolution background matting. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Nashville, TN, USA, 8758–8767. DOI: https://doi.org/10.1109/CVPR46437.2021.00865

[21] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. 2019. Importance estimation for neural network pruning. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 2019-June. IEEE Computer Society, Long Beach, California, 11256–11264. DOI: https://doi.org/10.1109/CVPR.2019.01152

[22] NVIDIA. 2021. Developer Guide—NVIDIA Deep Learning TensorRT Documentation v8.0.3. Retrieved from https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-803/developer-guide/index.html

[23] NVIDIA. 2022. Developer Guide :: NVIDIA Deep Learning TensorRT Documentation v8.5.2. Retrieved from https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-852/developer-guide/index.html#intro-quantization

[24] NVIDIA. 2023. GeForce GTX 1080 Ti Graphics Cards. Retrieved from https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1080-ti/

[25] NVIDIA. 2023. Jetson Modules. Retrieved from https://developer.nvidia.com/embedded/jetson-modules

[26] NVIDIA. 2023. ONNX-TensorRT: TensorRT Backend For ONNX. Retrieved from https://github.com/onnx/onnx-tensorrt

[27] ONNX Runtime developers. 2021. ONNX Runtime. Retrieved from https://onnxruntime.ai/

[28] ONNX Runtime developers. 2023. Quantize ONNX Models. Retrieved from https://onnxruntime.ai/docs/performance/quantization.html

[29] Fabian Pedregosa and Philippe Gervais. 2022. Memory_profiler: Monitor Memory Usage of Python Code. Retrieved from https://github.com/pythonprofilers/memory_profiler

[30] Facundo Quiroga. 2023. Facundoq/Tinyimagenet. Retrieved from https://github.com/facundoq/tinyimagenet

[31] Raspberry Pi Foundation. 2023. Raspberry Pi 4 Model B Specifications—Raspberry Pi. Retrieved from https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[32] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Salt Lake City, UT, 4510–4520. DOI: https://doi.org/10.1109/CVPR.2018.00474

[33] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*. International Conference on Learning Representations, ICLR, San Diego, California, USA, 1–14.

[34] TechPowerUp. 2023. NVIDIA GeForce GTX 1080 Ti Specs. Retrieved from https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877
[35] TechPowerUp. 2023. NVIDIA Jetson AGX Xavier GPU Specs. Retrieved from https://www.techpowerup.com/gpu-specs/jetson-agx-xavier-gpu.c3232
[36] Xilinx. 2022. Vitis-AI-Tutorials/Design_Tutorials/09-Mnist_p yt at 1.4 · Xilinx/Vitis-AI-Tutorials. Retrieved from https://github.com/Xilinx/Vitis-AI-Tutorials
[37] Xilinx. 2023. IP Facts • DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338) • Reader • AMD Adaptive Computing Documentation Portal. Retrieved from https://docs.xilinx.com/r/en-US/pg338-dpu/IP-Facts
[38] Xilinx. 2023. Versal AI Core Series. Retrieved from https://www.xilinx.com/products/silicon-devices/acap/versal-ai-core.html
[39] Xilinx. 2023. Xilinx/Vitis-AI at v3.0. Retrieved from https://github.com/Xilinx/Vitis-AI/tree/v3.0
[40] Xilinx. 2023. Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. Retrieved from https://www.xilinx.com/products/boards-and-kits/zcu104.html
[41] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Vol. 8689. Springer International Publishing, Cham, 818–833. DOI: https://doi.org/10.1007/978-3-319-10590-1_53
[42] Ligeng Zhu. 2022. PyTorch-OpCounter. Retrieved from https://github.com/Lyken17/pytorch-OpCounter

Received 10 October 2023; revised 30 January 2024; accepted 21 March 2024