Fast Retraining of Approximate CNNs for High Accuracy

Elias Trommer, Bernd Waschneck, Akash Kumar

***Abstract*—One technique for approximating neural networks when deploying to resource-constrained systems is the use of approximate multiplications. Giving up full mathematical accuracy opens new opportunities for more efficient hardware implementations. Modeling the effects of inaccurate hardware already in the training stage improves performance but significantly slows down the training due to expensive type conversions and memory access operations. We propose a method to speed up the simulation of inaccurate hardware by using a composition of floating-point functions. Both an analytical and a data-driven method for finding these functions are provided. We further provide a study and implementation of per-channel quantization, a scheme that enhances the granularity of converting neural network parameters to integers. This helps boost the application's accuracy. In our evaluation, our floating-point models achieve up to a $4\times$ speed-up over the commonly-used lookup table implementation, while providing a high fidelity simulation of the target function. Extending quantization with per-channel granularity yields a median accuracy improvement of 0.87 p.p. for ResNet8/CIFAR10 with 4-bit weight quantization in combination with hardware using approximate multipliers. Our extended software toolkit for the study of approximate multipliers in PyTorch is publicly available and provides a variety of building blocks for applying inaccurate product functions to neural networks.**

***Index Terms*—neural networks, pruning, microcontrollers**

I. Introduction

The success of deep learning is driven by vast amounts of data and increasingly large neural network architectures [1]. As the demand outpaces hardware improvements, lowering the resource consumption of a deployed neural network becomes more and more important. Various *approximation* techniques have been proposed to improve the trade-off between resource consumption and performance. Both pruning and quantization alter the network on the parameter level. Pruning [2] identifies connections that disproportionately affect accuracy and retains them, while other connections are eliminated, which leads to a decrease in the number of active parameters. Quantization [3] converts a network trained in a floating-point representation into a lower bit-width fixed-point integer representation.

A different approach to approximation concerns the numerical operations of the neural network algorithm. Because multiplications are the main driver of hardware complexity in a neural network, approximations of the arithmetic product function circuitry itself have been proposed in the past [4]. Relaxing the requirement for mathematically exact results leverages the error tolerance of neural networks to allow for more efficient hardware designs.

Earlier work focuses on developing Approximate Multipliers (AMs) that maximize accuracy when running neural networks that were trained using accurate product functions. Later work demonstrated that accuracy can be improved significantly if the network is trained with awareness of the approximate product function [5]. Since the induced error is an effect of the hardware design, it is deterministic and predictable, allowing for it to be accurately modeled. By incorporating the inaccuracy of the approximate product function in the training stage, the network's parameters can converge to a configuration that minimizes the approximation error that gets propagated during inference.

Constructing an approximation-aware training pipeline is complicated by the fact that the requirements of the training and the inference stage vary widely: Training is carried out once at design time on capable desktop or High-Performance Compute (HPC) systems, typically employing graphics processing units (GPUs). Due to better convergence on a smooth numerical surface, floating-point numerical formats are the common choice for neural network training, leading to GPUs being highly optimized for floating-point operations. Inference, in contrast, runs continuously and often on systems with minimal available resources, making approximation techniques especially relevant for this stage.

Several software solutions have been developed to integrate AM simulation with deep learning training [6]–[8]. However, these solutions rely on the same mechanism for AM simulation: Assuming that approximate product implementations are combined with low bit-width integer quantization allows for pre-computing all AM results, and then storing them as a 2D array. During training, multiplication inputs are converted to integers, which are used as indices to retrieve the pre-computed output from this lookup table (LUT) of results. Adapting the matrix multiplication—the fundamental operation in neural networks—at this low level also entails that layer implementations must be rewritten to incorporate AM. This added complexity increases the cost of implementation and decreases throughput, compared to standard neural network training frameworks. So far, this has limited the study of arithmetic approximation in neural network to very small networks. Most available toolkits build on Tensorflow, while submissions to machine learning conferences suggest that the research community has come to prefer PyTorch [9]. Only one publicly available implementation is based on PyTorch [7], yet it lacks both GPU acceleration and advanced quantization schemes.

The intersection of approximate computing and neural network (NN) quantization remains underexplored. Most work using AMs for NNs focuses on 8-Bit operands throughout the network. A large body of work in the field of quantization, however, suggests that fewer than eight Bits might be sufficient to provide the same accuracy as the baseline model [10]. This is important for resource-constrained NN accelerators, since

smaller operands reduce the application's need for expensive memory accesses [11]. Especially for lower bit-widths, the choice of quantization scheme becomes significant for the application's final accuracy. By providing flexible support for more sophisticated quantization schemes, TorchApprox aims to enable the study of low bit-width AMs in deep learning pipelines.

We initially proposed TorchApprox [12], a feature-rich toolkit for integrating AM into neural network training as a solution to some of these issues. TorchApprox introduces High-Throughput (HTP) models, a new type of approximate multiplication operator during training: Instead of using a LUT, HTP models emulate the approximation error using a combination of primitive floating-point functions. We extend the concept in this work by applying it to logarithmic multipliers, an entirely different class of AMs. All HTP models are compared to several other retraining methods reported in the literature in terms of throughput and reproduction of the target function. We further extend TorchApprox with more complex quantization schemes that enhance the granularity of the conversion from floating-point to integer. Through this higher granularity, it may be possible to use smaller numerical representations while achieving comparable accuracy.

In summary, this work provides the following novel contributions:

- We extend the previously proposed HTP model approach to the commonly used logarithmic AMs by deriving a floating-point behavioral model from their algorithmic description.
- All proposed HTP models are thoroughly benchmarked for their throughput and accuracy on three different Convolutional Neural Network (CNN) tasks. The results are compared with other state-of the art methods that aim to improve neural network performance when using approximate product implementations.
- We provide a formal definition of an approximate matrix multiplication that uses per-channel and affine quantization and an experimental study of the achievable increase in accuracy when CNN weights are quantized with per-channel granularity instead of global per-tensor parameters.

With HTP models, we provide both fast and faithful modeling of the effects of AM during retraining. Extending quantization with increased granularity further boosts the accuracy of approximate neural networks after retraining. Our library is publicly available under www.github.com/etrommer/torch-approx.

II. Background and Related Work

Using approximate multiplications to replace the resource-intensive accurate multiplications in a neural network has been discussed in several studies in the past. We provide an overview of relevant AM architectures and the currently available implementations of AMs in deep learning training pipelines.

A. Approximate Multipliers

One of the first algorithms to use approximation to simplify the implementation of multiplications on hardware was proposed by John N. Mitchell in 1962 [14]. Mitchell's algorithm is based on the observation that a multiplication can be converted to an addition by applying logarithmic and inverse logarithmic transformations to the operands and the result, respectively. We give a short summary of the algorithm as provided by Liu et al. [15] which will help understand the concepts discussed in Section III-C. Two unsigned fixed-point operands A and B can be expressed as

$$A = 2^{k_1}(1 + x_1),\ 0 \leq x_1 < 1 \tag{1}$$

$$B = 2^{k_2}(1 + x_2),\ 0 \leq x_2 < 1 \tag{2}$$

which can be converted to the binary logarithm of A and B

$$\log_2(A) = k_1 + \log_2(1 + x_1) \tag{3}$$

$$\log_2(B) = k_2 + \log_2(1 + x_2) \tag{4}$$

for which the logarithm of the product $P = A \cdot B$ can be rewritten to give

$$\log_2(P) = k_1 + k_2 + \log_2(1 + x_1) + \log_2(1 + x_2) \tag{5}$$

Mitchell then applies the linear approximation $x \approx \log_2(1+x)$ for $0 \leq x < 1$, which allows for the product to be calculated using a simple addition:

$$\log_2(P) \approx k_1 + k_2 + x_1 + x_2 \tag{6}$$

This algorithm is only applicable to unsigned operands. The most common approach to extending this method to signed multiplication is an initial sign extraction for both operands, calculating the product using absolute values and a re-insertion of the appropriate sign into the result.

A whole class of approximate multiplication algorithms build upon Mitchell's early work and rely on the simplicity of approximate multiplications in the logarithmic domain, but provide additional simplifications. Hashemi et al. [4] propose dynamic truncation of the log-transformed operands to their most-significant bits, leading to reduced hardware complexity. In the same work, the authors discuss how to modify the truncation scheme so that the resulting error is unbiased—an improvement on Mitchell's original algorithm which always underestimated the correct results. Truncation of operands in the logarithmic domain for CNNs is also discussed by Kim et al. [16], who conclude that they can help reduce the arithmetic resource consumption of deep learning models.

A different technique that is commonly used for the approximation of hardware multipliers is partial product truncation [17]. A truncated multiplier omits the generation and accumulation of the least significant partial products. The technique decreases the result's resolution, but can significantly improve the multiplier's area and power consumption.

EvoApproxlib, a popular implementation of approximate hardware multipliers, was first proposed in [18]. The authors present a library of 8-bit AMs that were designed from scratch using a genetic algorithm. AMs are selected to target different Pareto points of accuracy and resource consumption. The library was later extended to include AMs in other integer formats [19]. No simple model of the error function is available for these algorithmically generatedAMs.

Table I
APPROXIMATE NEURAL NETWORK LIBRARIES FEATURE COMPARISON

	ProxSim [8]	TFApprox [6]	TFApprox4IL [13]	AdaPT [7]	TorchApprox (this work)
Based on	TF	TF	TF	PyTorch	PyTorch
GPU Acceleration	✓	✓	✓	—	✓
Grouped Convolutions	—	—	✓	(✓)*	✓
Open Source	—	✓	—	✓	✓
affine & per-channel quantization	—	affine only	affine only	—	✓

*most common configuration

Fitting a linear regression model to the error function of an AM in order to obtain a condensed representation was first proposed by Ullah et al. [20]. In this work, the performance of an AM on a task is predicted from its linear regression coefficients using a Multi-Layer Perceptron (MLP). However, the models are not used as a functional equivalent of the AM. For a high-throughput simulation of the approximation error during training, additive gaussian noise (AGN) is frequently used as a low fidelity surrogate model [21]. Several other error models discussed in the literature allow for characterizing certain error properties, but not a full emulation of the output function [22], [23].

B. Approximate Neural Network Training

The enormous complexity of recent neural network models makes it infeasible to implement them from scratch for non-trivial applications. This has given rise to deep learning frameworks, which provide the building blocks required for implementing and training a neural network. TensorFlow [24] popularized the use of a dataflow graph representation to describe and optimize neural network computations. Additionally, the authors emphasize parallel execution across large numbers of GPU-enabled nodes. The more recent PyTorch [25] addresses a similar set of problems but is primarily designed to speed up research and experimentation through its accessible programming interface.

Studying the effects of quantization and approximation on a neural network is a task that is already supported by several solutions which we compare in Table I. ProxSim [8] is built around a GPU-accelerated implementation of matrix multiplications using an approximate product function (ApproxGeMM). The ApproxGeMM implementation serves as a primitive that is used to evaluate Convolutional layers (which are unfolded to batched 2D matrices using the im2col operator) as well as Fully-Connected (FC) layers. TFApprox [6] only supports Convolutional layers but optimizes GPU throughput further; to speed up memory accesses, the GPU's texture memory serves as a cache that holds the pre-computed LUT of the AM's output space. TFApprox has recently been expanded to cover retraining and grouped convolutions [13]. Of these libraries, only TFApprox is publicly available. ApproxTrain, another toolkit based on Tensorflow, implements a simulation of floating-point AMs with large operand bit-widths. As the LUT size increases quadratically with the operand bit-width, storing it in memory becomes infeasible for large operands. The authors instead propose only storing the mantissa part in a LUT, while performing sign and exponent computation at runtime. Using this method, only AMs with floating-point operands can be simulated during training. This is rarely useful when the model is meant to be deployed to a constrained accelerator where integer operands are standard practice, and thus the simulation of *integer* AMs is required during training.

Currently, the only other implementation in PyTorch is AdaPT [7]. The authors propose several optimizations for inference on a central processing unit (CPU), including the use of JIT-compiled kernels, loop unrolling to maximize the use of SIMD capabilities, as well as multithreading. AdaPT does, however, not provide GPU acceleration for approximate layers.

All other frameworks only provide a fixed and limited choice of simple quantization algorithms, with ProxSim and AdaPT only providing symmetric quantization. TFApprox and TFApprox4IL provide affine operand quantization, but can only determine a single parameter pair for an entire operand tensor. For all frameworks, the quantization is fused with the LUT simulation kernel. TorchApprox, in contrast, has been extended in this work to also support schemes that are able to express quantization using a dedicated parameter pair for each *channel*. These more complex schemes can help improve accuracy, especially when the distribution of weight values differs across channels [26]. TorchApprox also relies on the high-level Quantization interface provided by PyTorch itself. This enables further experimentation with different quantization algorithms by the user without the need for adapting low-level primitives.

III. PROPOSED METHODOLOGY

We derive an approximate neuron multiplication operator with affine and per-channel quantization, starting from the definition of the accurate operator. Moreover, we provide a rationale for using floating-point models instead of the commonly used LUT simulation of AMs. We show how to derive these models analytically from the definition of logarithmic multipliers and in a black-box fashion using a data-driven approach for AMs that are algorithmically generated.

A. Affine and per-channel quantization

Converting a trained neural network model to a numerical format of a lower bit-width than was used for training has become standard practice for deploying to devices outside of

data center environments [26]. A common choice of target numerical format are 8-bit integers, since they provide decent savings in memory and compute at virtually no accuracy loss. The quantization scheme chosen determines how the neural network parameters are converted to lower-precision integers, using either Post-Training Quantization (PTQ) once the model is fully-trained or iteratively during training, a technique known as Quantization-Aware Training (QAT). The conversion has to balance the added complexity during inference with minimizing the error introduced by the quantization process. For the remainder of this paper, two distinctions are important: First, symmetric quantization and affine quantization; and second, per-tensor and per-channel quantization. In both cases, the former is a subset of the latter. The simplest case, per-tensor symmetric quantization, scales all real numbers to integers using the same scaling factor throughout an entire tensor, but keeps the zero-point identical. Affine quantization, in contrast, adds an additional offset Z, so that the mapping from quantized integers q to real numbers r takes the form

$$r = S\,(q - Z) \tag{7}$$

as discussed by Jacob et al. [27]. This representation is beneficial if the distribution of values is not centered around zero, as is often the case for activations. For approximate implementations, this has the added benefit that it can map neural network parameters to both signed and unsigned ranges. While a detailed derivation can be found in the original work, we will briefly reiterate the most important fact for this work, namely that the resulting entry $r_{i,k}$ from multiplying two quantized $N \times N$ matrices can be expressed as

$$r_{i,k} = S_x S_w \sum_{j=1}^{N} (x^{(i,j)} - Z_x) \cdot (w^{(j,k)} - Z_w) \tag{8}$$

from which the subtraction from the multiplicands can be factored out to give

$$\begin{aligned} &= S_x S_w \Big(N Z_x Z_w \\ &- Z_x \sum_{i=1}^{N} w^{(j,k)} - Z_w \sum_{j=1}^{N} x^{(i,j)} \\ &+ \sum_{j=1}^{N} x^{(i,j)} w^{(j,k)} \Big) \end{aligned} \tag{9}$$

Previous research suggests that ‘the multiplications in CNNs can be approximated while the additions have to be accurate’ [28]. In addition, the main driver of energy consumption is multiplication hardware [29]. This work therefore follows the commonly used approach of replacing multiplications with an approximate operator while additions remain accurate [30], [31]. We can replace the accurate product $x^{(i,j)} \cdot w^{(j,k)}$ with an arbitrary integer operation $f(x^{(i,j)}, w^{(j,k)}) \approx x^{(i,j)} \cdot w^{(j,k)}$ to obtain an approximate matrix multiplication operator with affine input quantization. Implementing affine quantization, as opposed to the simpler symmetric quantization, allows us to natively support both signed and unsigned AMs while maximizing the numerical range that is used by the quantized values.

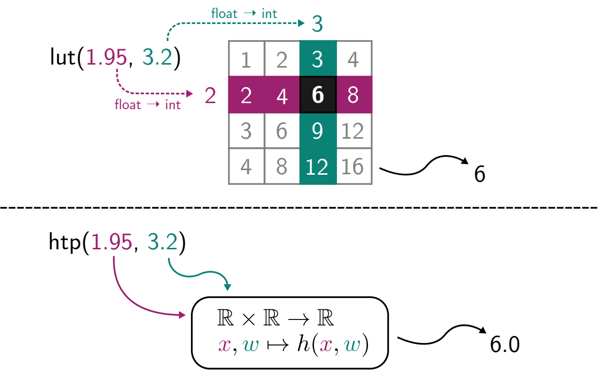

Figure 1. Schematic comparison of LUT (top) and HTP (bottom) approach to simulation of approximate multiplications. Finding an HTP function $h(x, w)$ requires domain-specific knowledge of the underlying AM.

We hypothesize that optimizing quantization granularity is highly important to achieve good accuracy when targeting a combination of low bit-width quantization and approximate multiplications. Optimizing the numerical range of parameters is likely to help convergence by maximizing the amount of information that can be conveyed in each operand. To this end, we extend support to also cover per-channel quantization. Per-channel quantization is a scheme in which the granularity of the quantization process is increased so that *each channel* in a weight tensor can be quantized with individual scale and zero point parameters. This helps minimize the numerical error introduced by the quantization process, particularly for lower bit-widths. By substituting the scalars S_w, Z_w with vectors $\vec{S_w} \in \mathbb{R}^N$, $\vec{Z_w} \in \mathbb{Z}^N$, we can adapt Equation (10) to obtain the formulation of an approximate matrix multiplication with affine and per-channel quantization as:

$$\begin{aligned} r_{i,k} \approx\; & S_x S_w^{(k)} \Big(N Z_x Z_w^{(k)} \\ & - Z_x \sum_{i=1}^{N} w^{(j,k)} - Z_w^{(k)} \sum_{j=1}^{N} x^{(i,j)} \\ & + \sum_{j=1}^{N} f(x^{(i,j)}, w^{(j,k)}) \Big) \end{aligned} \tag{10}$$

Our toolkit implements this scheme for all supported layer types.

B. From Fake-Quantization to Fake-Approximation

When running inference of neural networks on resource-constrained embedded systems or accelerators, integer quantization is the preferred representation for activations and weights [32]. In contrast, floating-point formats dominate the domain of neural network training. This is a natural fit for GPUs, which are highly optimized for peak throughput when performing floating-point operations, while integer performance is typically poor in comparison. For the training of quantized networks, this disparity can be alleviated through the use of *fake quantization*: The desired quantization algorithm is applied to the operands; the quantized values, however, remain in a floating-point representation throughout the entire training procedure [27]. This allows for an accurate modeling of the

Figure 2. Comparison of LUT (left) and HTP (right) approach to simulation of approximate multiplications in the context of a neural network layer. Integer operation domain is highlighted in green. Note that the quantization operation $\mathbf{Q}$ is replaced with fake-quantization Q' on the right side, leading to a composition of only floating-point functions.

effects of quantization while keeping a fast floating-point training pipeline. In contrast, a LUT-based ApproxGeMM implementation must materialize the operands as integers to carry out the array lookup operation that retrieves the approximate multiplication result.

To incorporate AMs in a neural network layer during model preparation, we must replace the accurate matrix product in each neuron with a custom multiplication followed by a summation of the results. Numerous optimizations have been developed for the *accurate* matrix product, many of which are specific to certain applications or target hardware. Maintaining an equally large number of optimized *approximate* matrix product implementations that target different platforms and use cases is not a realistic option. Instead, the common approach is to map the various neural network layer tasks to a single ApproxGeMM primitive, leading to reduced throughput compared to that of accurate neural networks. We propose an alternative approach to this problem: Could the AM simulation be carried out in the floating-point domain by a function that matches the AM's output but decomposes it into a series of operand transformations and *accurate* matrix products? Through this, highly-optimized neural network layer implementations could serve as a building block instead of having to resort to re-implementing them from scratch.

For an accurate neuron, the pre-activation output, denoted as y, for activations x and weights w is defined as

$$y = \sum_{i=1}^{n} x_i \cdot w_i \tag{11}$$

while the inner product is replaced with some approximate operation

$$f(x, w) \approx x \cdot w \tag{12}$$

for an approximate neuron

$$\tilde{y} = \sum_{i=1}^{n} f(x_i, w_i) \tag{13}$$

We omit the neuron's bias term in this derivation for brevity. For a fast software simulation, the approximate product $f(x, w)$ is typically expressed as a 2D array lookup with integer indices x and w. In contrast, an HTP model replaces the costly type conversion of x and w and the subsequent memory access with an HTP function $h(x, w)$. A comparison of how LUT and HTP operators differ fundamentally is provided in Figure 1. Instead of a 2D array lookup, the HTP function expresses the target function as a weighted sum of transformations applied to x and w. Each of these transformations g_x, g_w is a simple real-valued function $\mathbb{R} \to \mathbb{R}$ that might help the overall HTP model $\mathbb{R} \times \mathbb{R} \to \mathbb{R}; x, w \mapsto h(x, w)$ in recreating the AM's original inaccuracy. The weight coefficient c_j then determines the amount with which each transformation pair $(g_{x,j}, g_{w,j})$ contributes to the HTP model. It is important to note that picking an appropriate set of transformations requires some insight into the internal function of the AM. Finding the specific combination of transformations as determined by the values of c_j can, however, partially be automated as we will demonstrate in Section III-D. This gives us the formulation of an HTP function $h(x, w)$, composed of a set of k input transformations and their respective weight coefficients $\{(c_1, g_{w,1}, g_{x,1}), (c_2, g_{w,2}, g_{x,2}), \dots, (c_k, g_{w,k}, g_{x,k})\}$ as:

$$\begin{aligned} h(x, w) &\approx f(x, w) \\ &= \sum_{j=1}^{k} c_j \cdot g_{w,j}(w) \cdot g_{x,j}(x) \end{aligned} \tag{14}$$

which we can insert into Equation (13) to obtain

$$\tilde{y} \approx \sum_{i=1}^{n} \sum_{j=1}^{k} c_j \cdot g_{w,j}(w_i) \cdot g_{x,j}(x_i) \tag{15}$$

Importantly, this equation can be rephrased as

$$\tilde{y} \approx \sum_{j=1}^{k} c_j \cdot \sum_{i=1}^{n} g_{w,j}(w_i) \cdot g_{x,j}(x_i) \tag{16}$$

The inner sum is then simply the operation of the accurate neuron from Equation (11) with transformations $g_{x,j}$ and $g_{w,j}$ applied to its operands. Achieving AM simulation through repeated transformation of the input operands alone allows us to treat the underlying implementation of the neural network layer as a black box. We achieve the output simulation of the AM by repeatedly performing the layer operation with different transformations applied to its inputs. The output tensors are then aggregated by forming the weighted sum to give the operation's result. The approach is compared to the regular LUT operation in Figure 2.

We leverage a certain property of many AM designs for the discovery of HTP models: A large number of hardware implementations approximate the product function by truncating the least-significant Bit positions [33], [34]. This process happens early on in the multiplication algorithm, which means that subsequent logic can be omitted. Because the least-significant positions have the lowest impact on the magnitude of the result, the resulting hardware can be simplified significantly while maintaining a result close to that of an accurate multiplication.

The truncation of these Bit positions leads to a mapping of several different operand values onto a single value. The resulting product is then a piecewise constant function of the transformed multiplicands multiplied together. By capturing the dynamics of the operand truncation, the HTP model can retain an accurate product implementation with transformed operands.

In the remainder of this section, we demonstrate the application of this principle to two different AM designs.

C. Logarithmic Multipliers

With Equation (16), we have obtained a basis that allows us to use existing layer implementations in order to simulate the inaccuracy produced by an AM. However, it is not yet clear how to determine the transformations $g_{w,j}$ and $g_{x,j}$ for a specific hardware implementation. We illustrate the process of deriving an HTP model by hand using an analytical method in this subsection. In the following subsection, we describe a data-driven method that partially automates model discovery when analytical methods are not applicable.

We model the truncated Mitchell AM [4], [16] as an important representative of the wider class of logarithmic AMs. Past studies have discussed several variants of logarithmic AMs that specifically target NNs [35], [36]. The aim of this section is to provide a description of how these AM models can be efficiently simulated in the floating-point domain. With the provided derivation, we hope that researchers will be able to quickly adapt the proposed methodology for similar use cases. An advantage of logarithmic AMs is the availability of a full algorithmic description. We can use this description as a basis for our implementation. From a high-level perspective, a logarithmic AM transforms the operands x, w to their base-2 logarithms

$$X = \log_2(x), \quad W = log_2(w) \tag{17}$$

which are subsequently added and transformed back to the original domain using an exponential function:

$$x \cdot w = 2^{(X+W)} \tag{18}$$

This is simply a reformulation of an accurate product, without approximation. It is not immediately obvious how a sum followed by exponentiation of the result can be transformed to allow applying the equality between Equation (15) and Equation (16). Indeed, our previous work on this topic concludes with the remark that this is likely not possible at all and instead requires a custom kernel implementation [12]. However, an important detail is that implementations typically consist of applying a deterministic approximation to the logarithm conversion process, such that $X' \approx \log_2(x), \quad W' \approx \log_2(w)$. It is therefore sufficient to capture the effects of the conversion process and then convert the operands back to the original domain, before performing a regular multiplication. Given this, we can rewrite Equation (18) in the desired format to yield

$$w \cdot x \approx 2^{(X'+W')} \tag{19}$$

$$= 2^{X'} \cdot 2^{W'} \tag{20}$$

which leaves the task of finding transformations $g_x(x) = 2^{X'}$, $g_w(w) = 2^{W'}$. For all logarithmic multipliers in our study, the applied approximation of the logarithm transformation is the same for both operands. This allows us to further use the equality $g_x = g_w$. Extending the model for $g_x \neq g_w$ is, however, trivial in general since both transformations are applied independently of each other.

An approximation technique used by both the truncated Mitchell multiplier [16], [36] and DRUM [4] is the truncation of the operand's lowermost bits. Our goal is to emulate this binary integer operation in the floating-point domain for $x' \in \mathbb{R}_+$. In principle, truncating an n-bit binary number to the leading k bits (thus setting the lowest $n-k$ bits to zero) is equivalent to rounding down to the nearest multiple of $2^{(n-k)}$ in the floating-point domain. While n is derived from binary integers and thus not well-defined in the floating-point domain, it represents the same numerical range in both domains. Consequently, the value of n is not directly accessible in the floating-point representation. However, we know that if the leading one is in the n-th position of the unsigned binary integer representation, x' must satisfy $2^n \leq x' < 2^{n+1}$. We can therefore use the equality $n = \lfloor \log_2(x') \rfloor$. Using this identity, we can emulate the k-leading bit truncation $\lfloor x \rfloor_k$ by subtracting the modulus of x and $2^{(n-k)}$. This gives us the emulated truncation of the lowermost bits in the floating-point domain as:

$$g(x) := \begin{cases} x & \text{if } x < 2^k \\ x - x \bmod 2^{\lfloor \log_2(x) \rfloor - k + 1} & \text{otherwise} \end{cases} \tag{21}$$

DRUM [4] applies an additional correction by setting the $(n-k)$-th bit of the operand to one when truncating. Following the same logic as above, this correction factor can be simulated by truncating to k bits and then adding a constant $2^{\lfloor \log_2(x) \rfloor - k}$ to the operand.

When translating this to parallel code, there are two branches that need to be considered. First the decision of whether an operand is truncated, and second the sign extraction that is omitted here for brevity. Furthermore, the base-2 logarithm is an expensive function to evaluate, even with GPU acceleration. However, keeping values as floating-point throughout saves two type conversion to and from integer and the LUT memory access that retrieves the pre-computed result. For Convolutional layers, a potentially sub-optimal `im2col/GeMM` implementation is avoided in favor of the optimized implementation maintained by the framework itself. We expect that, despite some added overhead through operand transformations, these savings will overall be larger and speed up the entire training pipeline.

D. Truncated Multipliers

An analytical design of HTP models falls short when there is no explicit description of the underlying algorithm available. This limitation applies to AMs that were designed algorithmically. For selecting the appropriate transformations g and their corresponding weights c for algorithmic designs of truncated AMs, we propose the use of an equally algorithmic search procedure. Our method relies on selecting a superset

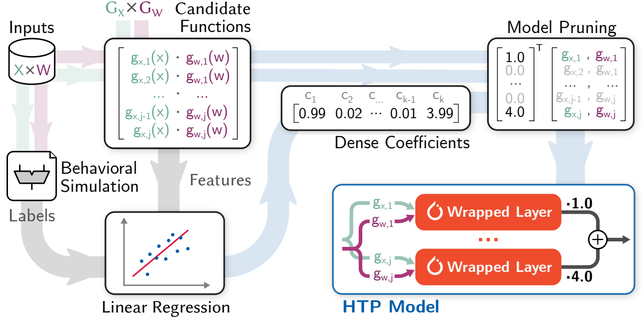

Figure 3. Construction of High-Throughput Model from a set of a candidate functions. A Linear Regression determines the weighting coefficients for the candidate functions that best describe the functional simulation of an Approximate Multiplier. The resulting coefficients are rounded and low-impact transformations are removed from the final model.

of potentially useful operand transformations G. The relevant subset of these candidate functions, respective hyperparameters and coefficients are determined using a data-driven approach.

The approximate multiplication output function $\tilde{z} \approx x \cdot w$ for a truncated multiplier that omits the l least significant partial products from one operand can be expressed as $\tilde{z} = x \cdot \left(w - w \bmod 2^l\right)$ [37]. Note that a similar transformation might also be applied to the other operand x, which is omitted here for simplicity. As previously, the rationale remains the same when applying these transformations to both operands, because both operands are transformed independently of each other. $\tilde{z}$ is a piecewise-constant function with uniform interval length 2^l. Accurately modeling these patterns using only polynomial input transformations is not possible. Instead, we require an additional transformation to capture the piecewise component of the output function. From the model of the truncated multiplication, we can see that a modulo function is a suitable choice. Adding a transformation of $\bmod 2^l$ to our set of candidate functions allows the model to capture the effects of partial product truncation in the floating-point domain. Our basis functions, applied to the inputs x, w, therefore are:

$$G_x = \left\{1, x, x \bmod 2^l\right\},\ G_w = \left\{1, w, w \bmod 2^m\right\}$$

where $l, m \in \mathbb{N}^+$ correspond to the partial product truncation for each operand. The user is not required to supply these parameters because the range of values for l, m can be covered exhaustively by a grid search. For most practical designs, it is reasonable to assume that l, m are of similar magnitude, allowing a further shrinking of the search space.

Rather than creating each model by hand, we propose using a linear regression that optimizes the selection of g and c over the set of candidate functions for each AM. Using a data-driven approach makes the method independent of knowledge of the approximate product error function and generalizes to AM designs that extend partial product elimination with error correction schemes [37], [38] or other approximation techniques.

To identify the relevant subset of the candidate function space $G_x \times G_w$ and the corresponding coefficients c_j, we fit a regression model to the output space of each target approximate multiplier. For each combination of input operands $(x, w) \in X \times W$, we obtain:

- a *feature vector* described by the multiplication of all possible combinations of the input transformations in Section III-D $g_x(x) \cdot g_w(w);\ (g_x, g_w) \in G_x \times G_w$ applied to x, w.
- a *label* containing the result of a behavioral simulation of the approximate multiplier's output $f(x, w)$

The task of the regression is to optimize the weight c_j for each input transformation to best capture the behavior of the target AM. For AMs with an input space that is too large to cover exhaustively (16-bit multipliers), a randomly sampled subset of the input space $X \times W$ can be used instead.

Different from classical regression problems, the inputs and expected outputs of the model are entirely known beforehand. The goal of the regression model is to find a *condensed* function that maps inputs to the expected outputs, rather than generalizing to unseen data. After fitting the model, the coefficients are rounded to one decimal place. This helps the model account for coefficients that have not fully converged. Transformations with near-zero coefficients are omitted from the final model as shown in Figure 3. Reducing the model to only the most relevant features eliminates the need to compute feature transformations with little influence on the overall result later on.

E. Generalization

In principle, any Boolean function can be expressed as a composition of functions in the floating-point domain using the same logic that was applied in Section III-C. For a functionally complete logic system, it is sufficient to define only a NAND operator. For two numbers $A, B \in \mathbb{R}_+$, the NAND operation of their bit position k can be expressed as

$$\begin{aligned}\overline{A \wedge B}_k :=\ & 1 - \left(A \bmod 2^k - A \bmod 2^{(k-1)}\right) \\ & \cdot \left(B \bmod 2^k - B \bmod 2^{(k-1)}\right) \cdot \frac{1}{2^{2(k-1)}}\end{aligned} \tag{22}$$

with this equivalence, any logic circuit can—in theory—be expressed as an HTP model. It is important to note that this is not necessarily always practical. Actual gains in performance can only be expected when the model can be reduced to a handful of input transformation. A property that enables this—and that both the studied logarithmic and truncated AM designs possess—is that the output surface of the approximated product function gives the same result for a large number of operand combinations. Input candidate functions then need to be selected in a way that they (partially) replicate this constant dynamic when their transformed inputs $g_x(x)$ and $g_w(w)$ are multiplied together.

IV. Results and Discussion

In this section, we provide an experimental evaluation of the effects of a per-channel quantization scheme on the accuracy of the combination of AMs and low bit-width neural networks. We also compare both versions of the HTP operator in end-to-end training scenarios for various CNNs. Finally, we evaluate

the throughput of both LUT-based kernels and HTP models provided by TorchApprox with the LUT-based kernels from two other AM retraining toolkits.

A. *Quantization Schemes*

TorchApprox introduces support for per-channel quantization in combination with AMs. Providing a more complex quantization scheme is based on the assumption that increasing the granularity of the quantization process will decrease the numerical error introduced by the integer conversion, leading to higher accuracy for the same AM designs, since it can make better use of the available numerical range for a given quantization bit-width. Because per-channel quantization adds complexity to training and inference which has to be justified, we conduct an experimental analysis of how performance is impacted by each scheme. We assess the effects of per-channel and per-tensor quantization on quantized neural networks on the LeNet-5 architecture and the MNIST dataset [39] as well as ResNet8 [40] and the CIFAR10 [41] image classification task. For both test cases, a baseline model is trained in FP32. From this, both a model with per-channel and per-tensor quantization are fine-tuned using QAT for each bit-width. To produce a single data point, an AM design is applied to every layer of each quantized model. The resulting network is then retrained with simulated approximation and the resulting model's Top-1 accuracy is reported. This is repeated for all $8 \times n$-Bit AM instances in the EvoApprox library, where $n = \{2, 3, 4\}$ for LeNet-5, since it is a very simple problem, and $n = \{4, 5, 6, 7\}$ for ResNet8/CIFAR10, because it is more complex. We show the comparison of the achieved Top-1 accuracy for every combination of test case, quantization scheme, bit-width and AM used throughout the network's layers along with a box plot that visualizes the *aggregate* statistics for each bit-width and quantization scheme in Figure 4. We notice that some combinations—particularly those employing AMs with very low precision—do not converge at all during the approximate retraining phase, but remain at an accuracy that is close to randomly guessing the output prediction. The reason for this is that some AM designs simply do not provide sufficient accuracy for the training process to produce gradients that would allow for the model parameters to converge at all. Both MNIST and CIFAR10 distinguish between ten target classes, putting the accuracy of randomly guessing at $10\,\%$ for both. We define a Top-1 Accuracy of $15\,\%$ as a threshold for convergence. To gain a better insight into the behavior of models that produce *useful* accuracy, the non-converged cases below the threshold are excluded in Figure 4. The number of non-converged networks for each experimental setup is, however, reported in Table II.

Table II
NUMBER OF CONVERGED MULTIPLIER CONFIGURATIONS AND MEDIAN ACCURACY FOR DIFFERENT BIT-WIDTHS AND QUANTIZATION SCHEMES

Model	Quantization		Multipliers		Median Top-1 Acc. [%]
	bit-width	Granularity	Total	Converged	
LeNet-5 & MNIST	2	tensor	13	11	94.67
		channel	13	11	99.13
	3	tensor	29	24	99.14
		channel	29	26	99.22
	4	tensor	29	26	99.24
		channel	29	25	99.25
ResNet8 & CIFAR10	4	tensor	29	19	84.62
		channel	29	19	86.10
	5	tensor	34	24	86.65
		channel	34	26	86.81
	6	tensor	25	15	87.00
		channel	25	16	87.06
	7	tensor	34	25	87.01
		channel	34	26	87.25

The results show that per-channel quantization helps with both accuracy and convergence. The number of AMs above the convergence threshold is the same or higher, when per-channel quantization is used in all cases except for one. Similarly, the median Top-1 accuracy that was achieved across the converged networks is above that of the models using per-tensor quantization for all test cases. This suggests that using per-channel quantization also helps boost accuracy. The effect is particularly pronounced for smaller bit-widths, while it tends to disappear for weight quantization with larger bit-widths. This can be explained by the fact that per-channel quantization can help reduce the numerical error introduced by the quantization process. The initial quantization error, however, is largest for small bit-widths, making higher granularity most effective in these scenarios, as was reasoned in Section III-A.

These experimental results show that there is a general tendency towards higher accuracy for approximate neural networks with per-channel quantization, but the evaluation so far does not allow for assessing the impact on *individual* approximate neural networks. To compare the effect on each individual AM, we further analyze the achieved accuracy for the 8×4-bit configurations in Figure 5. We limit this evaluation to the results of 14 out of the initial 29 AMs that achieve an accuracy of more than $80\,\%$, since models with an accuracy close to that of the baseline model are most relevant for real-world applications. In the chosen representation in Figure 5, the accuracy of the approximate neural network variant that uses per-channel quantization is on the y-axis, while the x-axis corresponds to the variant using per-tensor quantization. If a point lies *below* the main diagonal, it indicates that per-tensor quantization led to higher accuracy than per-channel quantization. Every point *above* the main diagonal shows that the per-channel configuration has converged to higher accuracy. From the fact that only a single point is marginally below the main diagonal, we can infer that switching from per-tensor to per-channel quantization yields an improvement in accuracy for almost every case in our evaluation. The mean improvement for the surveyed models is 1.09 percentage points, while the median improvement is 0.87 percentage points for the reported models with more than $80\,\%$ accuracy.

B. *HTP Model Fidelity*

HTP models themselves are not necessarily a faithful representation of the AM they were constructed for. This makes

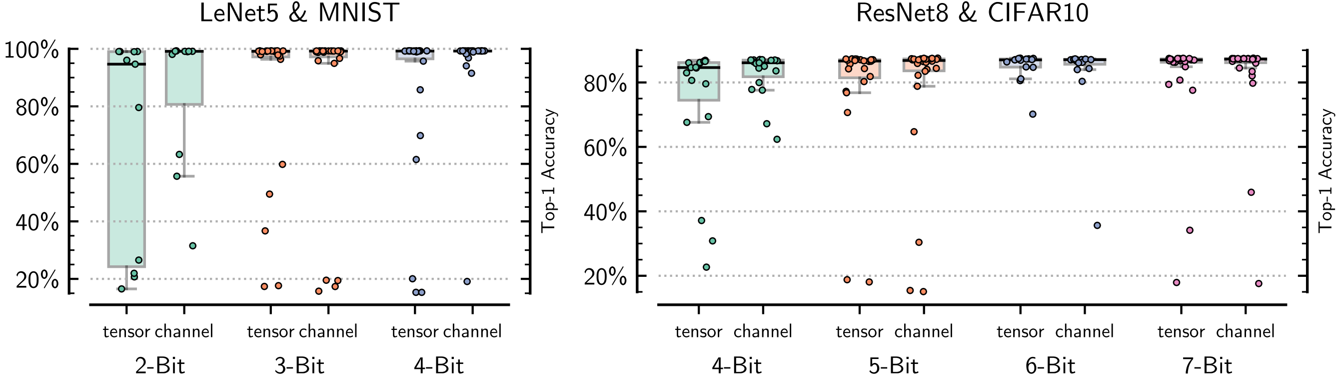

Figure 4. Comparison of per-tensor and per-channel quantization on LeNet-5/MNIST and ResNet8/CIFAR10 for different weight quantization bit-widths. Evaluation is carried out across all $8 \times n$-bit unsigned EvoApprox multipliers.

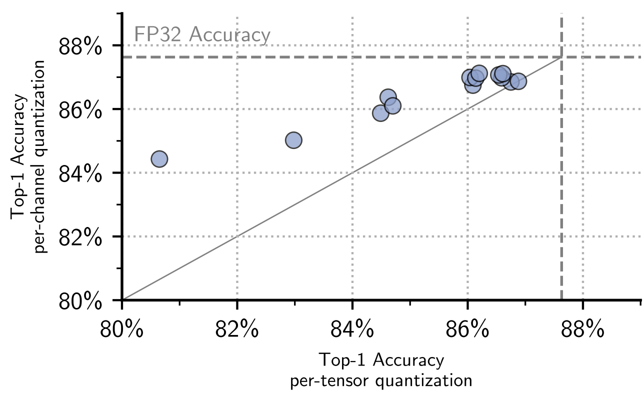

Figure 5. Comparison of per-tensor and per-channel quantization on ResNet/CIFAR10 across 8×4-bit unsigned EvoApprox multipliers. Results below $80\,\%$ accuracy are omitted. Main diagonal marks equivalent performance for per-tensor and per-channel quantization. Points above main diagonal mark improvement through per-channel quantization.

Table III
APPROXIMATE NEURAL NETWORK RETRAINING METHODS

Format	Description
Baseline	Accurate Product Function
Noise	Accurate Product Function with AGN that is characterized by the mean and standard deviation of the target AM's error distribution [42]
Linear	Linear regression model with polynomial feature expansion to five regression coefficients [43]
HTP	High-throughput model of the target AM (this work)
LUT	Faithful simulation of the target AM using a LUT of pre-computed results

it important to confirm that they recreate the target function sufficiently well. The aim of the experiment is to evaluate how closely each method tracks the results of behavioral simulation.

To establish the fidelity of HTP models in a training pipeline, we retrain several common neural network models for approximate inference using a number of different retraining methods and compare the results of each. The retraining methodology for all networks is described in detail in Appendix A.

For every architecture, we take a pre-trained and quantized model as a starting point. A copy of this baseline model is then retrained using a variety of methods that are described in detail in Table III. Optimizer configuration, epochs and all other hyperparameters are identical for all methods in the evaluation. By retraining the accurate baseline using the same hyperparameters, we ensure that all evaluated models have received the same amount of training. This procedure is repeated for a wide range of AMs. All combinations of experimental setup and retraining method are retrained for each of the 13 signed 8-bit AMs in the EvoApprox library as well as the DRUM [4] and Mitchell truncated [16] logarithmic multipliers for truncation bit-widths of 3, 4 and 5 bits respectively (thus totaling six logarithmic AM configurations). The principle of logarithmic AMs is independent of the operand bit-width. For the experiments, 12-bit quantization of weights and activations is used for all logarithmic AMs.

Behavioral simulation of the target AM is used to evaluate and compare retrained models on the test set, as it accurately reproduces the behavior seen after deployment to the target system. The top-k accuracy (with $k = 1$ for LeNet-5 and ResNet8 and $k = 5$ for VGG16) of the model trained using behavioral simulation is used as the ground truth, to which all operators are compared. The results of the MAE between ground truth accuracy and the respective method's achieved accuracy for each population of AM are shown in Figure 6. The exact numerical value of the MAE varies between models depending on the architecture, complexity of the task, etc. Since the aim of the evaluation is a *relative* comparison between the different methods, we normalize the MAE to one across all experiments in order to allow for a side-by-side comparison.

The evaluation shows that the HTP approach reproduces behavioral simulation with high fidelity. Despite the HTP model of the logarithmic AM being mathematically equivalent to the simulation, it does not achieve perfect replication of its behavior. This is explained by the lack of conversion between floating-point and integer representations in the HTP model, which uses

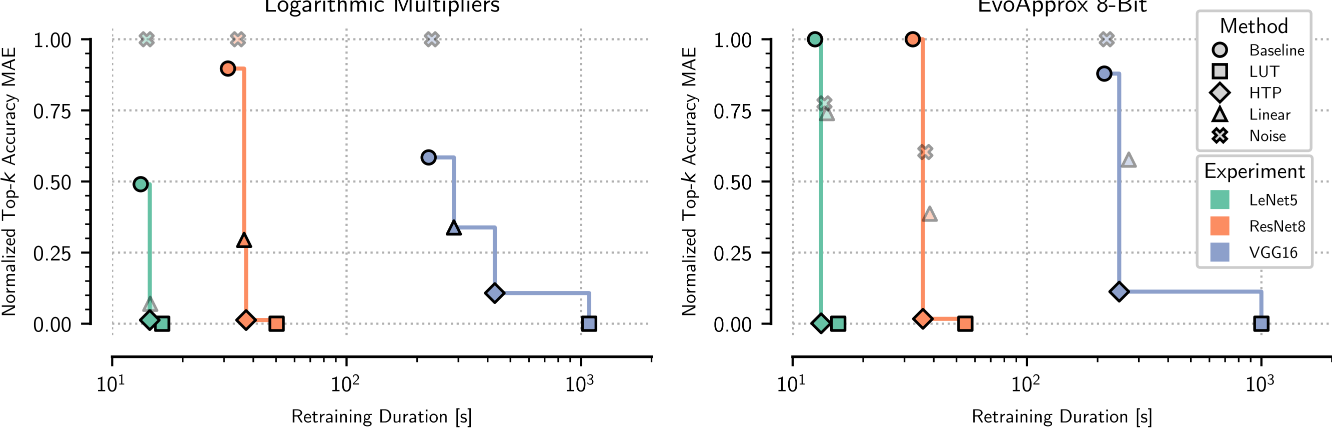

Figure 6. Pareto Front of retraining method fidelity compared to behavioral simulation and throughput for various retraining methods. Methods are compared for three different CNN experimental setups and two populations of AMs. The MAE between retrained model and behavioral simulation of the Top-k accuracy is aggregated across the evaluated AM population and normalized to one to make the values comparable across experiments. An ideal method would be found on the lower left corner, i.e. it would minimize both the Top-k accuracy error and the training time.

a floating-point numerical representation throughout the entire pipeline, causing a slightly different numerical behavior. The absolute numerical values of the MAE in Table VI reveal that the HTP models of EvoApprox AMs, which were generated using a data-driven linear regression, show a significantly higher deviation from the ground-truth because they can only partially capture the target AM's behavior. Of all retraining methods HTP, however, consistently provides the highest fidelity simulation of target AMs. In terms of throughput, HTP models are close to the baseline in all experiments, yielding an up to $4\times$ speedup over behavioral simulation for VGG16 and the simpler truncated multiplier models. HTP models of logarithmic multipliers are more time-consuming, due to their reliance on an expensive log transformation of operands in the floating-point domain. Nevertheless, the approach still provides nearly a $2.5\times$ speedup over behavioral simulation in the case of VGG16. For all experiments, HTP models achieve a Pareto-optimal balance between faithful recreation of the target AM and a minimal penalty on throughput in the retraining pipeline.

C. Throughput

Benchmarking is carried out on a system equipped with an AMD Ryzen 9 3900X CPU and an nVidia RTX 2080Ti GPU. The 24 logical cores available on the CPU are used for parallelizing CPU-based computations. We compare the performance of both our GPU-based LUT kernels and our HTP simulation models of the 8-bit EvoApprox AMs to AdaPT's accelerated CPU kernels. When using a LUT-based approach, throughput is independent of the chosen AM. Conversely, the computational complexity of HTP models varies depending on their number of coefficients and required input transformations. We also include 12-bit and 16-bit EvoApprox HTP models to evaluate the impact of simulating AMs with larger input spaces. A network that does not simulate any AM but only performs fake quantization is included as a baseline. The quantized network does not model the error from approximate hardware but serves as a reference for the overhead of such product implementations. The inference time for a single batch of ImageNet-sized data (224×224 pixels, 3 channels, batch size of 32) for several common CNNs is measured over 50 runs, with the median reported.

The benchmark results in Figure 7 suggest that there is indeed a performance degradation for all approximate hardware simulations compared to an accurate model that only uses fake quantization. For TorchApprox' LUT-based approach, this performance penalty ranges from $1.9\times$ to $16.7\times$ with EfficientNetB0 and MobileNetV2 showing the lowest overhead due to their use of the dedicated Approximate Depthwise Convolution operator. TorchApprox' dedicated Depthwise Convolution operator also outperforms the lower bound of the throughput achievable with HTP floating-point models. Across the different networks, the geometric mean of the speedup achieved by the 8-bit HTP models over the LUT-based kernels is $2.7\times$. Some performance penalty is unavoidable when comparing HTP models to the quantized baseline; HTP models must perform each layer's operation multiple times with different transformations applied to the inputs, and the results have to be scaled and accumulated. All TorchApprox inference modes outperform the CPU-based AdaPT with a geometric mean speedup of $96\times$ for the HTP models, and $36\times$ for the LUT-based approach, demonstrating that GPU acceleration is essential for modern neural network applications. Because the HTP operator is unique to TorchApprox, there is no equivalent operator available in AdaPT that would allow for a direct comparison. The performance of the LUT operator which, in terms of algorithm, is identical in both AdaPT and TorchApprox does, however allow for an indirect comparison. Thanks to a CPU's more sophisticated cache hierarchy and instruction-level support for array lookups, the difference between HTP models and an optimized LUT-implementation would likely be smaller when run on a CPU, compared to a GPU. The avoidance of expensive type conversions and handwritten kernels when using

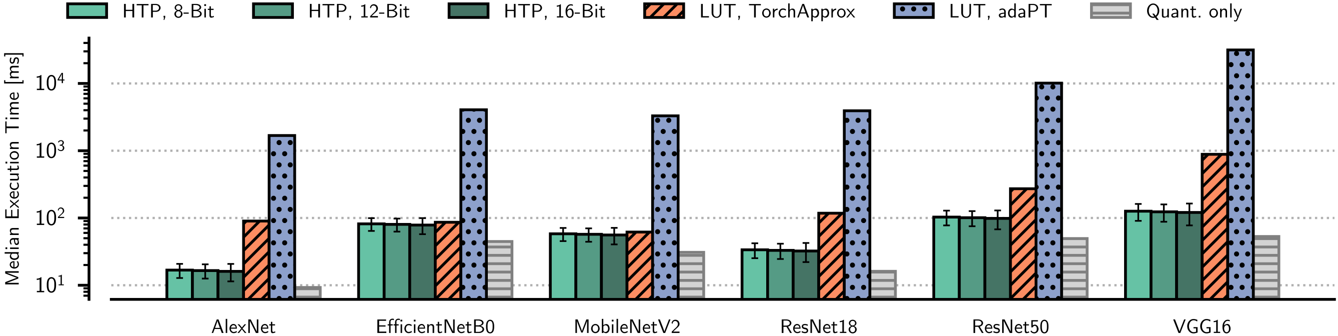

Figure 7. ImageNet inference single batch execution time (batch size 32) for several CNN models.

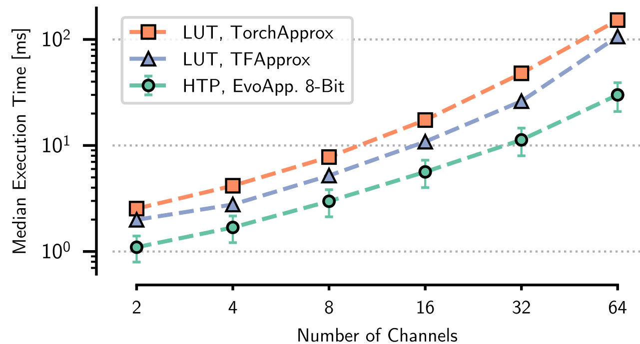

Figure 8. 2D Convolution inference single batch execution time for NCHW tensor with a varying number of channels $C_{\text{in}} = C_{\text{out}}$ and $N = 64$, $H = W = 128$.

HTP models would, however, still be beneficial, regardless of the compute device. In terms of absolute performance, it is highly unlikely that HTP models on a CPU be able to close the large performance gap between a CPU and a GPU.

The performance of HTP models with 12-bit and 16-bit operands is on par with those derived from 8-bit AMs. HTP models targeting larger bit-width AMs are slightly ahead in terms of throughput, compared to 8-Bit HTP models. This is explained by the fact these models are a simulation, where the input format is the same floating-point representation, regardless of the specific AM being simulated. The complexity and throughput of these models is entirely determined by the number and complexity of their required input transformations $g_x(x), g_w(w)$. Coincidentally, 16-Bit HTP models are slightly less complex, allowing for higher average throughput. The HTP model's independence of the operand bit-width of the AM being simulated is an important benefit for the study of AMs with larger operand bit-widths, since LUTs of more than 8×8-bit are typically not used due to impractical memory consumption and poor cache utilization. HTP models, in contrast, are applicable to AMs of any bit-width.

We also compare the performance of our ApproxGeMM kernels to those from TFApprox [6]. A full comparison of both libraries on a complete neural network task is difficult. The lack of a Depth-wise Convolution operator in TFApprox limits any benchmark to using outdated CNN architectures. Furthermore, both libraries are built on different frameworks, which could significantly impact end-to-end measurements. To mitigate this effect, we analyze the inference throughput of a single 2D Convolution layer while varying the number of input and output channels in Figure 8. For LUT-based implementations, TFApprox shows higher throughput for all configurations, which might be explained by the larger number of handwritten and optimized kernels in TFApprox. We would like to point out that our more modular approach reduces throughput, but simplifies maintenance and experimentation. The HTP models provide a geometric mean speedup of 2× when compared to TFApprox's 2D Convolution.

V. Conclusion and Outlook

In this work, we discussed and evaluated several novel techniques that help improve the speed and performance of neural networks using operators with approximate multiplications. By replacing expensive type conversion and memory lookup operations for the simulation of AMs with primitive floating-point functions, we achieved up to a 4× improvement in throughput for the largest evaluated benchmark model. Unlike a LUT-based simulation, our HTP approach does not depend on input operand bit-width, making it particularly attractive for AMs with higher-bit-width inputs where a LUT becomes infeasible due to the quadratic growth in memory usage. This operator has also been extended to cover logarithmic AMs. With this extension, we demonstrate that the concept is applicable to a wider range of AM designs. Despite relying on expensive operations like a floating-point logarithm and modulus, HTP models still provide significant speedups over commonly used AM simulation approaches with throughput approaching that of a training pipeline using accurate operations. A faster simulation of AMs during the retraining of NN applications can help in studying the effects of approximate hardware implementations on neural networks after deployment quicker and in more detail, increasing the accessibility of AM as another parameter for NN optimization to the more widely used pruning and quantization.

A remaining limitation is the process of deriving HTP models. While some model parameters can be discovered in a partially automated fashion, at least the search space still needs to be designed by hand, based on insights into the function of the respective subclass of AMs. Furthermore, the use of a composition of input transformations relies on AMs having an output space that is constant for large numbers of operand combinations and does not trivially generalize to other AM designs while still providing useful acceleration. While a generalization to any Boolean function is theoretically possible, how these generalized models can be simplified to the extent that they are faster than LUT operations has not been discussed in this work. Similarly, our evaluation finds that a retraining with simulated AMs might lead to non-convergence in a larger number of cases. In this study, only a relatively simple retraining scenario was evaluated. More complex retraining methods that provide better convergence are a promising direction for future research.

We also provided the theoretical background for combining AMs and integer quantization, both for affine per-tensor and per-channel schemes. Comparing weight quantization with per-channel and per-tensor granularity showed that increased granularity of quantization parameters provides a significant boost in accuracy over the commonly used per-tensor quantization scheme. This is especially true when combining AMs and quantization to very low bit-widths.

By improving support for AM operators during neural network training, we provide tools that improve the performance of deep learning models when they are deployed on inaccurate hardware. We hope our results will inspire future research into the representation of simple logic operations using optimized surrogate models that take advantage of modern GPUs's advanced floating-point capabilities.

ACKNOWLEDGMENTS

The project on which this report is based was funded by the German Ministry of Education and Research (BMBF) under the project number 16ME0542K. The responsibility for the content of this publication lies with the author.

APPENDIX

A. Network Training Setup

To improve reproducibility, we discuss in detail the model training process that was used for the experiments in Section IV-A and Section IV-B. An overview of the most important hyperparameters is given in Table IV. All experiments start from a baseline model in FP32, without any approximation or quantization. For LeNet-5 and ResNet8, this model is trained from scratch, while a pre-trained model is used and fine-tuned in the case of VGG16. A quantized version is then fine-tuned using QAT. Because Section IV-B considers 8×8-Bit AMs, the simpler symmetric per-tensor scheme is used in this experiment.

Section IV-A aims to establish whether quantization with per-channel granularity improves over a per-tensor scheme (and if so, by how much). Because the quantization error grows for smaller bit-widths, the effect is expected to be largest when quantizing to small bit-widths. To quickly evaluate a large space of AMs, an LeNet-5 instance is used. From the baseline model, weights are quantized to 2, 3 and 4 bits using both per-tensor and per-channel quantization, while activations remain in an 8-bit per-tensor quantization for all experiments. For each combination of bit-width and granularity, we generate a quantized baseline model using QAT. Each quantized model is then retrained with simulated approximate multiplications in all convolutional and FC layers for all 2-, 3-, and 4-bit AMs from the EvoApprox library in PyTorch, using the TorchApprox simulation library with LUTs. The quantization parameters are updated alongside other network parameters to produce neural network configurations with the highest accuracy.

Table IV
EXPERIMENTAL SETUPS USED IN SECTIONS IV-A TO IV-B

	Mode	Parameters
LeNet5 MNIST	Baseline	10 epochs, SGD, L_2 Reg. $= 1 \cdot 10^{-4}$, Initial LR = 0.1, $\gamma = 0.75$ epochs (3,6,9), momentum = 0.9
	QAT	6 epochs, SGD, Initial LR = $5 \cdot 10^{-3}$, $\gamma = 0.9$ (epochs 4,5)
	Approx.	6 epochs, SGD, Initial LR = $5 \cdot 10^{-3}$, $\gamma = 0.9$ (epochs 4,5)
ResNet8 CIFAR10	Baseline	180 epochs, SGD, L_2 Regularization $= 1 \cdot 10^{-4}$, Initial LR = 0.1, $\gamma = 0.9$ epochs (90,130,160), momentum = 0.9
	QAT	12 epochs, SGD, Initial LR = $1 \cdot 10^{-2}$, $\gamma = 0.9$ (epochs 6,9), momentum = 0.9
	Approx.	8 epochs, SGD, Initial LR = $1 \cdot 10^{-2}$, $\gamma = 0.9$ (epochs 5,7), momentum = 0.9
VGG16-BN TinyImageNet	Baseline	from pretrained, 30 epochs, SGD, L_2 Regularization $= 1 \cdot 10^{-4}$, Initial LR = $1 \cdot 10^{-2}$, $\gamma = 0.8$ (Val. Acc Top-5 Plateau Scheduler w. patience = 3 epochs), momentum = 0.9
	QAT	2 epochs, SGD, Initial LR = $5 \cdot 10^{-4}$
	Approx.	Linear layers only, 1 epoch, SGD, Initial LR = $1 \cdot 10^{-3}$

For ResNet8, the range of quantization bit-widths is set to be higher, encompassing 4-, 5-, 6- and 7-bit AM designs, because of the higher complexity for ResNet8

In addition to the LeNet-5 and ResNet8 models described above, a more complex task is added and evaluated in Section IV-B. To gain insights into model performance on larger CNNs, the VGG16 architecture [44] is evaluated on the TinyImageNet dataset [45]. For VGG16, we apply approximate computation only to the three FC layers at the end of the network, rather than the full network because we find that full approximation leads to non-convergence in many instances. This is not directly representative of any real-world workload but serves as a synthetic test case that allows us to evaluate the retraining performance of the various AM simulations on a much larger number of models than if we applied AMs to all layers. Due to the complex task, VGG16 is retrained for a single epoch with a learning rate of $1 \cdot 10^{-3}$. In line with previous work [23], [46], we find that most accuracy is either recovered very early on in the fine-tuning process, or that the model does not converge at all. This is especially true for architectures without residual branches like VGG16. The experiment conducted Section IV-B also does not necessarily require full convergence of the evaluated network, since the aim of the experiment is to examine how well different models follow the convergence of behavioral simulation in a training

Table V
RETRAINING DURATION FOR EXPERIMENTS IN FIGURE 6

	Experiment	Baseline	Behavioral	HTP	Linear	Noise
Logarithmic Multipliers	LeNet5	13.2s	16.3s	14.5s	14.5s	14.0s
	ResNet8	31.2s	50.3s	37.3s	36.6s	34.4s
	VGG16	224.0s	1084.0s	429.0s	287.0s	231.0s
Evoapprox 8-Bit	LeNet5	12.5s	15.6s	13.2s	14.0s	13.7s
	Resnet8	32.6s	54.6s	36.0s	38.5s	36.8s
	VGG16	213.7s	997.7s	248.6s	271.8s	218.6s

Table VI
MEAN ABSOLUTE ERROR OF TOP-k ACCURACY IN PERCENTAGE POINTS AFTER RETRAINING WITH VARIOUS METHODS FOR EXPERIMENTS IN FIGURE 6

	Experiment	Baseline	Behavioral	HTP	Linear	Noise
Logarithmic Multipliers	LeNet5	3.85e-01	0.00e+00	1.00e-02	5.50e-02	7.85e-01
	ResNet8	9.91e+00	0.00e+00	1.42e-01	3.26e+00	1.11e+01
	VGG16	6.33e-02	0.00e+00	1.17e-02	3.67e-02	1.08e-01
Evoapprox 8-Bit	LeNet5	8.53e+00	0.00e+00	1.15e-02	6.32e+00	6.61e+00
	ResNet8	1.23e+01	0.00e+00	2.10e-01	4.76e+00	7.42e+00
	VGG16	9.24e-01	0.00e+00	1.18e-01	6.06e-01	1.05e+00

Table VII
MEDIAN SINGLE BATCH INFERENCE TIME IN MILLISECONDS FOR DIFFERENT CNN MODELS AND AM RETRAINING METHODS AS SHOWN IN FIGURE 7.

Experiment	HTP 8-Bit	HTP 12-Bit	HTP 16-Bit	LUT, TA	LUT, adaPT	Quant.
AlexNet	17±4	17±4	16±5	90	1682	9
EfficientNetB0	82±18	80±18	79±21	86	4062	45
MobileNetV2	58±13	57±13	56±15	62	3300	31
ResNet18	34±8	33±8	32±10	118	3925	16
ResNet50	103±26	101±26	99±31	272	10110	49
VGG16	126±36	124±36	121±43	884	31467	53

Table VIII
SINGLE BATCH INFERENCE TIME IN MILLISECONDS FOR DIFFERENT 2D CONVOLUTION CHANNEL COUNTS AS SHOWN IN FIGURE 8.

Channels	HTP, 8-Bit	TorchApprox, LUT	TFApprox, LUT
2	1.097±0.303	2.538	1.991
4	1.687±0.475	4.171	2.774
8	2.977±0.852	7.780	5.200
16	5.634±1.629	17.376	10.854
32	11.292±3.297	47.935	26.159
64	30.001±9.094	152.319	106.572

pipeline scenario.

For the evaluation of throughput in Section IV-C, the model's accuracy is not considered at all, because the inference process is not influenced by the model's parameters. For this reason, all evaluated models are taken directly from the TorchVision model zoo, without further adaptation.

B. Experimental results

The exact values of the experimental results from Sections IV-A to IV-C are provided in Tables V to VIII. Note that HTP model values in Table VII and Table VIII are aggregated across *all* HTP models of the given bit-width, with mean and standard deviation reported.

REFERENCES

[1] N. Maslej, L. Fattorini, E. Brynjolfsson, J. Etchemendy, K. Ligett, T. Lyons, J. Manyika, H. Ngo, J. C. Niebles, V. Parli, Y. Shoham, R. Wald, J. Clark, and R. Perrault, "Artificial Intelligence Index Report 2023," Oct. 2023.

[2] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS Conference, Denver, Colorado, USA, November 27-30, 1989*. Morgan Kaufmann, 1989, pp. 598–605.

[3] D. Lin, S. Talathi, and S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, Jun. 2016, pp. 2849–2858.

[4] S. Hashemi, R. I. Bahar, and S. Reda, "DRUM: A dynamic range unbiased multiplier for approximate applications," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015*. IEEE, 2015, pp. 418–425.

[5] Y. Fan, X. Wu, J. Dong, and Z. Qi, "AxDNN: Towards the cross-layer design of approximate DNNs," *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 317–322, Jan. 2019.

[6] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina, "TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU," *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 294–297, Mar. 2020.

[7] D. Danopoulos, G. Zervakis, K. Siozios, D. Soudris, and J. Henkel, "AdaPT: Fast Emulation of Approximate DNN Accelerators in PyTorch," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 6, pp. 2074–2078, Jun. 2023.

[8] C. De La Parra, A. Guntoro, and A. Kumar, "ProxSim: GPU-based Simulation Framework for Cross-Layer Approximate DNN Optimization," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Grenoble, France: IEEE, Mar. 2020, pp. 1193–1198.

[9] H. He, "The state of machine learning frameworks in 2019," *The Gradient*, 2019.

[10] B. Rokh, A. Azarpeyvand, and A. Khanteymoori, "A Comprehensive Survey on Model Quantization for Deep Neural Networks in Image Classification," *ACM Transactions on Intelligent Systems and Technology*, vol. 14, no. 6, pp. 1–50, Dec. 2023.

[11] M. Horowitz, "Computing's energy problem (and what we can do about it)," *2014 {IEEE} International Conference on Solid-State Circuits Conference, {ISSCC} 2014, Digest of Technical Papers, San Francisco, CA, USA, February 9-13, 2014*, pp. 10–14, Feb. 2014.

[12] E. Trommer, B. Waschneck, and A. Kumar, "High-Throughput Approximate Multiplication Models in PyTorch," *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 79–82, May 2023.

[13] M. Pinos, V. Mrazek, F. Vaverka, Z. Vasicek, and L. Sekanina, "Acceleration Techniques for Automated Design of Approximate Convolutional Neural Networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 13, no. 1, pp. 212–224, Mar. 2023.

[14] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," *IEEE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, Aug. 1962.

[15] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, Sep. 2018.

[16] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, May 2019.

[17] G. Zervakis, K. Tsoumanis, S. Xydis, N. Axelos, and K. Pekmestzi, "Approximate Multiplier Architectures Through Partial Product Perforation: Power-Area Tradeoffs Analysis," *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pp. 229–232, May 2015.

[18] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 258–261, Mar. 2017.

[19] V. Mrazek, L. Sekanina, and Z. Vasicek, "Libraries of Approximate Circuits: Automated Design and Application in CNN Accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 4, pp. 406–418, Dec. 2020.

[20] S. Ullah, S. S. Murthy, and A. Kumar, "*SMApproxlib*: Library of FPGA-based approximate multipliers," *Proceedings of the 55th Annual Design Automation Conference*, pp. 1–6, Jun. 2018.

[21] X. He, L. Ke, W. Lu, G. Yan, and X. Zhang, “AxTrain: Hardware-oriented neural network training for approximate inference,” in *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED 2018, Seattle, WA, USA, July 23-25, 2018*. ACM, 2018, pp. 20:1–20:6.

[22] S. Mazahir, M. K. Ayub, O. Hasan, and M. Shafique, “Probabilistic Error Analysis of Approximate Adders and Multipliers,” in *Approximate Circuits*. Cham: Springer International Publishing, 2019, pp. 99–120.

[23] E. Trommer, B. Waschneck, and A. Kumar, “Combining Gradients and Probabilities for Heterogeneous Approximation of Neural Networks,” *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, Oct. 2022.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 265–283.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 2019, pp. 8024–8035.

[26] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, “A White Paper on Neural Network Quantization,” Jun. 2021.

[27] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 2704–2713.

[28] M. S. Kim, A. A. Del Barrio, H. Kim, and N. Bagherzadeh, “The Effects of Approximate Multiplication on Convolutional Neural Networks,” *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 2, pp. 904–916, Apr. 2022.

[29] G. Zervakis, H. Saadat, H. Amrouch, A. Gerstlauer, S. Parameswaran, and J. Henkel, “Approximate Computing for ML: State-of-the-art, Challenges and Visions,” in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. Tokyo Japan: ACM, Jan. 2021, pp. 189–196.

[30] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, “Energy-Efficient Neural Computing with Approximate Multipliers,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 14, no. 2, pp. 1–23, Apr. 2018.

[31] C. De La Parra, A. Guntoro, and A. Kumar, “Full Approximation of Deep Neural Networks through Efficient Optimization,” *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, Oct. 2020.

[32] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding,” in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[33] M. Ahmadinejad and M. H. Moaiyeri, “Energy- and Quality-Efficient Approximate Multipliers for Neural Network and Image Processing Applications,” *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2021.

[34] F.-Y. Gu, I.-C. Lin, and J.-W. Lin, “A Low-Power and High-Accuracy Approximate Multiplier With Reconfigurable Truncation,” *IEEE Access*, vol. 10, pp. 60 447–60 458, 2022.

[35] M. S. Ansari, B. F. Cockburn, and J. Han, “An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing,” *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 614–625, Apr. 2021.

[36] M. S. Kim, A. A. Del Barrio, R. Hermida, and N. Bagherzadeh, “Low-power implementation of Mitchell’s approximate logarithmic multiplication for convolutional neural networks,” *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 617–622, Jan. 2018.

[37] G. Zervakis, O. Spantidi, I. Anagnostopoulos, H. Amrouch, and J. Henkel, “Control Variate Approximation for DNN Accelerators,” *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 481–486, Dec. 2021.

[38] O. Spantidi, G. Zervakis, I. Anagnostopoulos, H. Amrouch, and J. Henkel, “Positive/Negative Approximate Multipliers for DNN Accelerators,” *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, Nov. 2021.

[39] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[40] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778.

[41] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” 2009.

[42] M. A. Hanif, R. Hafiz, and M. Shafique, “Error resilience analysis for systematically employing approximate computing in convolutional neural networks,” in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*. IEEE, 2018, pp. 913–916.

[43] S. Ullah, S. S. Sahoo, and A. Kumar, “CLAppED: A Design Framework for Implementing Cross-Layer Approximation in FPGA-based Embedded Systems,” *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 475–480, Dec. 2021.

[44] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[45] Y. Le and X. S. Yang, “Tiny ImageNet Visual Recognition Challenge,” Tech. Rep., 2015.

[46] C. De La Parra, A. Guntoro, and A. Kumar, “Improving approximate neural networks for perception tasks through specialized optimization,” *Future Generation Computer Systems*, vol. 113, pp. 597–606, Dec. 2020.

Elias Trommer received an M.Sc. in Computer Engineering from Technische Universität Berlin in 2020. He is currently pursuing a Ph.D. degree at Technische Universität Dresden, in cooperation with Infineon Technologies Dresden. His research interests include the application of gradient-based optimization techniques and the inference of neural networks on resource-constrained devices like microcontrollers.

Bernd Waschneck received an M.Sc. in semiconductor physics from Ludwig-Maximilians-Universität Munich in 2013 and a Ph.D. in manufacturing engineering from the University of Stuttgart in 2020. Since 2014, he works at the semiconductor company Infineon Technologies in the fields of data science and artificial intelligence. He currently works as Director of System Innovation & Software in the Infineon Development Center Dresden, where he leads the System Innovation team. His research interests include hardware and software AI acceleration and embedded AI on microcontrollers for smart sensors.

Akash Kumar received the joint PhD degree in electrical engineering and embedded systems from the Eindhoven University of Technology and the National University of Singapore (NUS) in 2009. From 2009 to 2015, he was with NUS. He is currently a Professor with Technische Universität Dresden, where he is directing the Chair for Processor Design. His current research interests include the Design, Analysis, and Resource Management of Low-Power and Fault-Tolerant Embedded Multiprocessor Systems.