Hardware-Aware Training for Multiplierless Convolutional Neural Networks

Rémi Garcia, Léo Pradels, Silviu-Ioan Filip, Olivier Sentieys
Université de Rennes, IRISA, Inria
Rennes, France
{firstname.lastname}@irisa.fr

***Abstract*—Many computer vision tasks use convolutional neural networks (CNNs). These networks have a significant computational cost and complex implementations, in particular on embedded systems. A common way to implement CNNs on integrated circuits is to use low-precision quantized weights and activations instead of *de facto* floating-point (FP) ones. This is important to reduce the implementation cost. However, this has drawbacks regarding accuracy, and Quantization-Aware Training (QAT) is one of the most popular approaches to mitigate this issue. In this article, we introduce a multiplierless-aware training approach that significantly reduces hardware resource consumption. We propose to incrementally fix weights to their current value based on their implementation cost. To compute this cost, we base our approach on a Multiple Constant Multiplication (MCM) shift-and-add solving technique. With this idea, we show a global implementation cost reduction by around 25% w. r. t. a vanilla QAT approach without hardware usage in the loop. Compared to state-of-the-art multiplierless-aware training methods, the network accuracy of our designs is closer to that of a vanilla QAT baseline.**

***Index Terms*—Convolutional Neural Network (CNN), Multiple Constant Multiplication (MCM), Hardware accelerator**

I. Introduction

Convolutional Neural Networks (CNNs) have become a mainstay in computer vision tasks such as image classification or image restoration. While CNNs can easily be deployed on powerful hardware for offline evaluation, there is an increasing need to deploy such solutions at the edge. Indeed, in cases such as self-driving cars, CNN inference should be carried out as close as possible to the sensors with minimal latency. However, CNNs have a significant computational cost, and their implementation on embedded systems is complex and prone to stringent constraints. As such, a common way to implement CNNs on Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs) is to use low-precision quantized weights and activations instead of de facto floating-point ones. This avoids potentially costly floating-point units in favor of smaller, dedicated, and optimized integer-based arithmetic operators.

Knowing that the trained neural network will be implemented using quantized weights and activations, it is possible to adapt the quantization accordingly [1]. This reduces the overall hardware cost of the implementation. However, quantization comes with a penalty in terms of accuracy loss of the neural network [2], and additional constraints on the quantization step can further degrade it severely. To avoid a significant drop in accuracy when using quantization, it is possible to optimize it during the training process, and adapt weights accordingly. This approach is called Quantization Aware Training (QAT) [3] and is agnostic to the specifics of the neural network architecture being optimized.

This work was funded by the HOLIGRAIL project (ANR-23-PEIA-0010).

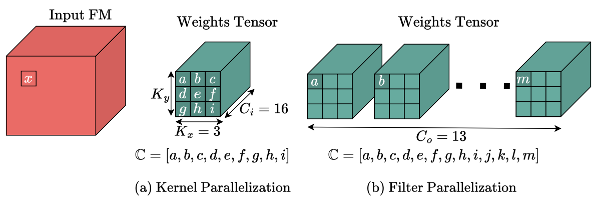
Input FM
x
Weights Tensor
K_y
a b c
d e f
g h i
$C_i = 16$
$K_x = 3$
$\mathbb{C} = [a, b, c, d, e, f, g, h, i]$
(a) Kernel Parallelization
Weights Tensor
a
b
m
$C_o = 13$
$\mathbb{C} = [a, b, c, d, e, f, g, h, i, j, k, l, m]$
(b) Filter Parallelization

Fig. 1. Representation of different parallelization opportunities within a convolution layer: (a) kernel parallelization, and (b) filter parallelization.

When looking at CNN implementation possibilities, it is helpful to note that each input is multiplied by many quantized weights, as illustrated in Fig. 1, where the input x is multiplied by: (a) all the weights inside a kernel (to construct different output elements), and (b) by the weight at the same position from each filter. Due to memory transfer limits, we usually do not have a fully parallel implementation [4]. Instead, we usually either parallelize all the kernel multiplications (as in Fig. 1a) or all the filter multiplications (as in Fig. 1b). In both cases, as each input is multiplied by multiple constant weights, minimizing this implementation cost is equivalent to solving a corresponding Multiple Constant Multiplication (MCM) problem. This problem involves minimizing the cost of multiplying a single variable with multiple constants [5], or in this particular case, the weights. It is possible to parallelize multiplications differently, using a loop interchange method [6]. This is applied after training, while the focus of this work is on hardware-aware training.

In this work, we present a novel hardware-aware quantized training approach that also takes advantage of the MCM approach, and show its efficiency directly through synthesis experiments, with corresponding accuracy and hardware usage metrics. We produced a new mathematical model, implemented with Pyomo [7], to guide our quantized training flow. The new problem we define and solve with our approach is inspired by both QAT and progressive pruning methods. Ultimately, we implemented a corresponding accelerator using

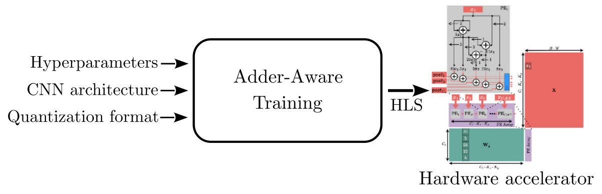

Fig. 2. Our training-to-hardware flow. From user-given parameters, we start our quantization-aware training approach. Once trained, we produce HLS code for the corresponding hardware accelerator.

High-Level Synthesis (HLS). Overall, we propose a complete training-to-implementation flow, as illustrated in Fig. 2. From a given CNN architecture pre-trained using floating-point arithmetic, we use a quantization-aware training method to find appropriate quantized weights. Then, we apply our hardware-aware method for the refinement process, and we use PACT [8] for the activation function quantization. Finally, we use HLS to generate an optimized hardware accelerator.

In Section II, we provide the necessary background and related work information. Then, in Section III, we present our multiplierless-guided QAT approach. The proposed hardware accelerator is described in Section IV, whereas training and hardware synthesis results are discussed in Section V.

II. Background & Related Work

A. Quantization-Aware Training

Quantizing an already trained model may introduce perturbations that push it away from the region it had converged to when using floating-point arithmetic. Re-training the model with quantized parameters can generally improve accuracy, and QAT is the most widely used approach to do this. Popularized by the likes of BinaryConnect [3], the *modus operandi* of QAT consists of performing forward and backward passes using the quantized model in floating-point, with the particularity that model parameters are quantized after each gradient update, similar to projected gradient descent methods.

An important aspect to be aware of is that quantization operators are non-differentiable. This creates issues during backpropagation, since the gradient of such operators will be zero almost everywhere (due to the piecewise flat nature of quantizers), making progress during re-training impossible. A common way to address this is to approximate the gradient of this operator by a so-called Straight Through Estimator (STE) [9] that essentially ignores the quantizer and approximates it with an identity function. This approach can also be extended to learn quantization parameters during QAT as well, such as activation clipping ranges [8] and activation scaling factors [10], [11].

In this work, we used the DoReFa-Net method [12] as a starting point for our QAT flow. This approach quantizes weights to a given bit width and projects them on the range $[-1, 1]$ using the $\tanh(\cdot)$ operator. Values are also normalized to reach at least one of the bounds, -1 or 1. Then, ReLU-based activation outputs are quantized using PACT [8].

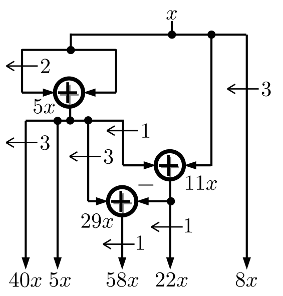

Fig. 3. MCM implementation with the weight set $\{5, 8, 22, 40, 58\}$ using a single adder graph.

B. Shift-and-add-based Multiple Constant Multiplication

One common way to address MCM problems is to replace multiplications with shifts, additions, and subtractions (adders), the so-called shift-and-add approach [13], [14]. For example, the multiplication of an integer variable x by the constant 7 can be done as $7x = 2^3x - x$, replacing the costly generic multiplication with a bit-shift and a subtraction.

Using the least number of adders (bit-shifts can be hardwired at a negligible cost) is a typical intermediate goal to obtain a minimal cost hardware implementation. Various MCM adder minimization methods exist, such as greedy algorithms [13], heuristics [15], [16] or Integer Linear Programming-based (ILP) approaches [14], [17]. Compared to greedy algorithms based on the Canonical Signed Digit (CSD) representation [13], heuristic methods generally provide better solutions. However, these solutions still lack optimality guarantees or certificates, w. r. t. the optimized metric, and are not easily extendable. ILP approaches, on the other hand, provide optimality guarantees or a certifiable gap to the best known bound and can be conveniently modified [18], [19] to incorporate different metrics.

In Section III, we present specifics of the problem we aim to solve and show that it does not exactly map to an MCM instance. Even so, an ILP-based approach is the most flexible and the easiest to adapt to handle such changes. While slower than heuristics for instance, put in the context of training, the ILP runtime is usually dominated by the DNN optimization time, making the ILP overhead acceptable.

Applying one of the above-mentioned methods on a set $Y = \{y_i\}_{i=1}^n$ of n target constants, we can produce a “shift-and-add chain” to perform all the multiplications $y_i x$. For example, the multiplications $x \times \{5, 8, 22, 40, 58\}$ can be done with the following shift-and-add sequence:

$$\begin{aligned} y_1 &= x \ll 2 + x = 5x, & y_2 &= x \ll 3 = 8x, \\ t_1 &= y_1 \ll 1 + x = 11x, & y_3 &= t_1 \ll 1 = 22x, \\ y_4 &= y_1 \ll 3 = 40x, & t_2 &= y_1 \ll 3 - t_1 = 29x, \\ y_5 &= t_2 \ll 1 = 58x, \end{aligned}$$

where the t_i are intermediate products. At each step, the value by which x is multiplied with, *e. g.*, 11 in the case of t_1 or 58 in the case of y_5, is called a *fundamental*. The same MCM

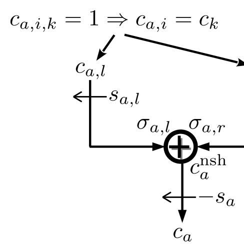

Fig. 4. Adder in MCM ILP modeling. Integer variable c_a is the fundamental associated to the adder, its inputs $c_{a,l}$ and $c_{a,r}$ are chosen from previous fundamentals, based on the value of the binary variables $c_{a,i,k}$'s. The shifts and signs of the adder are determined by $s_{a,l}$, s_a and $\sigma_{a,l}$, $\sigma_{a,r}$, respectively.

can be advantageously represented graphically as in Fig. 3 using an adder graph, where horizontal arrows are bit-shifts and adders with a minus sign above perform subtractions.

Finding the best adder graph, *i. e.*, one with the least number of adders, is the same problem as using the least number of adders in the shift-and-add chain. Usually, modern MCM solvers use adder graph representations [14]–[16] instead of the shift-and-add sequence approach. Heuristics, such as Hcub [15] or RPAG [16], build adder graphs, adder after adder or layer of adders after layer of adders, until the produced adder graph successfully outputs all the target multiplications.

The ILP approach is based on the mathematical modeling of an adder. For each adder, $a \in [\![1; N]\!]$, where N is a known upper bound on the total number of adders required to solve the MCM instance, an integer variable c_a encoding the fundamental computed by the adder, is used. The value of this fundamental directly stems from the input values of the adder, their associated shifts and if the adder performs an addition or a subtraction. This can be summed up as the following equation:

$$c_a = 2^{-s_a} \left((-1)^{\sigma_{a,l}} \, 2^{s_{a,l}} c_{a,l} + (-1)^{\sigma_{a,r}} \, c_{a,r} \right), \qquad (1)$$

where integer variables s encode the shifts, binary variables σ encode the signs, and $c_{a,l}$ and $c_{a,r}$ correspond to the previous left and right fundamentals of the adder graph. These previous fundamentals, including 1 as the input, are linked to the current adder with binary variables $c_{a,i,k}$. A graphic representation of the equation above is presented in Fig. 4. Although (1) is nonlinear, it has been shown that it can be linearized and used in an ILP approach [14].

For our work, we used the MCM ILP model from [14] to obtain optimized implementations of trained CNNs. We also used a modified version of the model during training. The overall approach is presented in Section III-B.

C. Shift-and-Add Aware Training

Multiple approaches to reduce the multiplication cost of CNNs have been proposed. In particular, using power of two (PoT) weights [20], [21] has a significant impact on the implementation cost. However, restraining the weight values so much comes with a notable accuracy loss [21]. Recent work [22] has shown that it is possible to merge QAT with a more complex shift-and-add aware quantization, called Adder-Aware Training (AAT). This has been done by precomputing sets of possible quantized values for which the overall implementation cost is limited to 1 or 2 adders per adder graph. Then the quantization steps during training are guided towards one of these sets. Precomputing sets is costly, thus limited to sets which can be implemented with a single or with two adders. To get an idea of the number of sets and the challenge of enumerating them, it is helpful to note that the number of odd outputs of an adder graph is bounded by its number of adders. Then, considering that 3-adder adder graphs output 3 odd values, which can be shifted to produce additional even values, for 8-bit quantized weights, the number of possible adder graph sets of outputs is $\binom{127}{3} = 333\,375$. This number is an upper bound as there are triples of odd fundamentals that actually cannot be computed with a 3-adder adder graph, but the order of magnitude remains. Hence, it is not reasonable to precompute all the sets of quantized values requiring 3 adders or more to implement.

Another work, called More AddNet [23] proposes an AAT approach where the implementation is based on Reconfigurable Constant-Coefficient Multipliers (RCCMs). The proposed RCCM implementation uses adder graphs with up to 4 reconfigurable adders. In their work, Hardieck *et al.* were able to precompute sets with 3 and 4 adders as additional constraints on the adder graph structure were imposed. This reduced the search space to a reasonable size but this also leads to a partial covering of all the possible 4-adder adder graphs instead of a complete enumeration. Finally, we note that their approach is not meant for parallelizing computations. Our approach, on the other hand, benefits from parallelism, thus does not target the same overall implementation problem. For this reason, we do not precisely compare hardware implementation between More AddNet and our approach.

In our work, we do not rely on precomputed sets, we compute the implementation cost only when needed. Although we increase the training time by incorporating it as an optimization component, it does not tend to bottleneck global training time. Our goal is to guide the quantization progressively towards a small implementation cost without limiting possible weight values *a priori*. Progressively fixing weights is a method that already proved its efficiency in the case of pruning [24]. In that case, it consists of zeroing out weights and activation signals, usually in a progressive manner [25, Fig. 7]. The choice of weights to be zeroed out is made based on a computed score. In our case, instead of fixing weights to zero, we will fix them to their current value.

III. Progressive MCM-based Quantization Flow

A. Our Strategy

The core idea of our MCM-based QAT flow is to fix the neural network weights incrementally based on a shift-and-add complexity score. Our weight-fixing process is illustrated in Fig. 5. This way, weights with the smallest implementation cost are fixed for the rest of the training. Then, the other

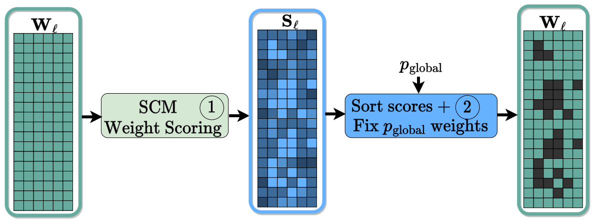

Fig. 5. Progressive weight fixation during network QAT. Every n-th epoch, a percentage of all the weights is fixed. The flow takes a network layer as input, then the Single Constant Multiplication (SCM) score is calculated for each weight which is not yet fixed, taking into account previously fixed weights. Then p_{global} percent of the weights are fixed (represented in black).

weights are fine-tuned, and the process is repeated until the end of the training, when all weights have been fixed.

As we target a multiplierless implementation, it is natural to consider the Single Constant Multiplication (SCM) problem for every weight and take the adder count from its solution as the score. To do this, we apply the ILP-based approach from [14] on each quantized weight. For example, suppose that we have the following list of quantized weights:

$$\mathbf{W} = [5, 40, 22, 8, 58]\,. \tag{2}$$

Taking the adder counts, we obtain the scores

$$\mathbf{S} = [1, 1, 2, 0, 2]\,. \tag{3}$$

Hence, we should fix the weight 8 first, as zero adders are necessary to implement it. Then 5, 40 (with one adder each), and 22 and 58 (with two adders each).

In this example, we did not consider that the weights will ultimately be implemented together, solving an MCM problem. Indeed, to implement the weights $\mathbf{W}$, we can use 3 adders in total (as illustrated in Fig. 3), instead of $0+1+1+2+2 = 6$ if each weight were to be implemented independently. However, there is no straightforward way to group weights together *a priori*. Hence, we cannot compute scores by considering the possible inter-dependencies.

However, we can compute scores by considering past information on fixed weights. This way, we can use already-fixed weights to more accurately compute the score of quantized weights. In (2), if we suppose that the first weight, 5, was fixed in a previous step, the score $\mathbf{S}$ would be

$$\mathbf{S} = [0, 0, 1, 0, 2]\,, \tag{4}$$

instead of the one reported in (3). Indeed, in some cases, fixed weights can lead to score reductions in weights that will be fixed later. Here 40 is directly implemented from 5 as

$$40 = 5 \ll 3, \tag{5}$$

and 22 only costs one additional adder, since

$$22 = 5 \ll 2 + 1 \ll 1. \tag{6}$$

However, not all scores are impacted: 8 was already cost-free as a power of 2, and 58 does not benefit from the fixed weight 5 since it still requires two adders.

To automatically compute these scores, we need to solve an SCM problem for each weight, considering that some fundamentals are already available and not just the input 1. This new problem is similar to the SCM/MCM problems, and its complexity is identical to that of the SCM problem[1], which is conjectured to be NP-hard [26]. To solve this problem, instead of simply using the ILP-based approach from [14], we first simplify it for SCM instead of MCM. Then, we modify the model to use the knowledge of already fixed weights.

B. SCM Model with Fixed Weights

Our goal is to find the score for each weight W and, in our case, this score basically corresponds to the minimal number of adders that will be used to implement the adder graph which outputs the constant C. In the following, we present our SCM model for finding the implementation cost of the positive and odd target constant C, stored with b bits, given the set $\mathbf{F}$ of already-fixed weights. For each weight W, we can easily compute its associated positive and odd target constant C as

$$C = \text{odd}\left(|W|\right), \tag{7}$$

where odd $(\cdot)$ corresponds to dividing the input by two until the result is odd. Then, it is reasonable to consider the implementation cost of Cx is close to the one of Wx: from Cx, a simple shift permits to obtain $|W|\,x$ and the sign can usually be retrieved with minimal cost.

For our model, similarly to previous work [14] on MCM, we use integer variables $c_a \in [\![0; 2^{\lceil \log_2 |W| \rceil} + 1]\!]$, $\forall a \in [\![0; N]\!]$, for storing each fundamental. The first variable, c_0, corresponds to the input and is fixed to 1. The upper bound N on the number of possibly useful adders is obtained using a greedy algorithm [13] on the SCM problem, *i.e.*, this bound is obtained without considering already fixed weights. We do not know *a priori* if all the N adders will be used or not. Thus, we encode that the adder a is used in the adder graph with binary variables $u_a \in \{0, 1\}$, $\forall a \in [\![1; N]\!]$.

We aim to minimize the number of actually used adders to get the most realistic score for our problem. This is done with the following objective:

$$\min \sum_{a=1}^{N} u_a. \tag{8}$$

Then, we add constraints to fix the adder graph topology and to ensure that the output is produced by one of the used adders. The latter constraint is obtained by using a binary variable for each adder, $o_a \in \{0, 1\}$, $\forall a \in [\![1; N]\!]$, where

$$c_a = C \quad \text{if } o_a = 1, \qquad \forall a \in [\![1; N]\!], \tag{9}$$

$$\sum_{a=1}^{N} o_a = 1, \tag{10}$$

ensure that the adder c_a corresponds to the target constant for the binary variable o_a equal to 1. At this point, the only

[1]The case where no weights are fixed, which is the SCM problem, is an instance of our problem.

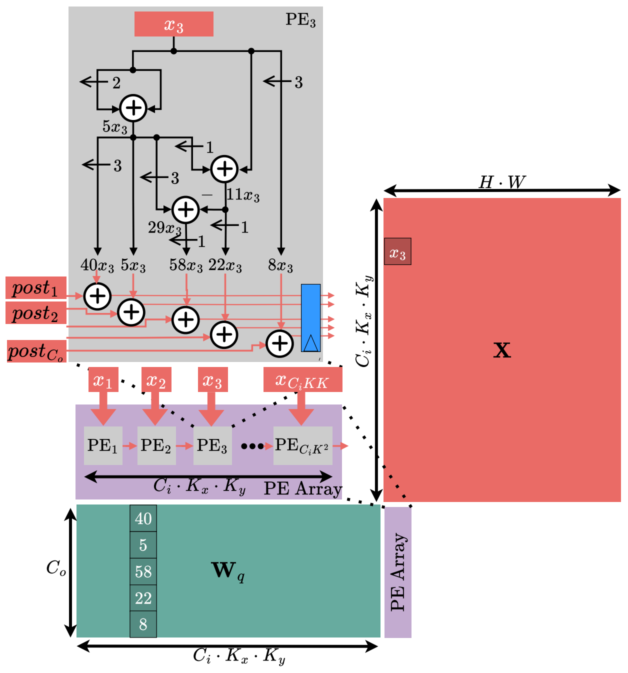
x_3
PE$_3$
2
3
$5x_3$
1
3
3
$11x_3$
$29x_3$
1
1
$40x_3$ $5x_3$ $58x_3$ $22x_3$ $8x_3$
$post_1$
$post_2$
$post_{C_o}$
x_1
x_2
x_3
x_{C_iKK}
PE$_1$ PE$_2$ PE$_3$ PE$_{C_iK^2}$
$C_i \cdot K_x \cdot K_y$ PE Array
$H \cdot W$
$C_i \cdot K_x \cdot K_y$
x_3
$\mathbf{X}$
C_o
40
5
58
22
8
$\mathbf{W}_q$
PE Array
$C_i \cdot K_x \cdot K_y$

Fig. 6. Our global architecture of a convolutional layer accelerator employs a Weight Stationary strategy [27]. This architecture includes an array of $C_i \cdot K_x \cdot K_y$ PEs, which hold stationary weights, while each PE passes the post-accumulation result to the next. Within each PE, the weights are hard-coded, here with an adder graph. Additionally, a post-accumulation input from the previous PE and a register (in blue) are used to store the current post-accumulation.

difference with the original MCM model [14] lies with the constraints on the outputs, which are simplified.

Our main change is taking fixed weights into account in the model. This can be done by increasing the number of variables c_a from $a \in [\![0; N]\!]$ to $a \in [\![-|\mathbf{F}|; N]\!]$. Then, we fix

$$c_a = F_{-a}, \qquad \forall a \in [\![-|\mathbf{F}|; -1]\!], \tag{11}$$

to store the fixed fundamentals in the model.

In the original model, adders are linked together using binary variables $c_{a,i,k} \in \{0,1\}$, $\forall a \in [\![1; N]\!], i \in \{l, r\}, \forall k \in [\![0; a-1]\!]$, where $c_{a,i,k} = 1$ implies that the i-th input (left or right) of the adder a is the adder k. In our case, we have to modify the indices to include fixed weights as possible inputs: $c_{a,i,k} \in \{0,1\}$, $\forall a \in [\![1; N]\!], i \in \{l, r\}, \forall k \in [\![-|\mathbf{F}|; a-1]\!]$. This way, we are able to adjust the topology to include fixed weights without any other changes to the other sets of variables. Finally, our change only impacts the sets of constraints involving $c_{a,i,k}$.

These modifications to the original MCM model give rise to an SCM model that takes fixed weights into account. As we will see in Section V, solving this model with the Gurobi optimizer is quite fast and can be included in the CNN training phase.

IV. CNN Inference Accelerator Design

For the accelerator, we use a systolic array for each layer which is based on a weight stationary dataflow [27] design. The design fully unrolls the layer, employing the `Im2Col` algorithm to execute both convolution and fully connected layers as matrix-matrix multiplications [28], [29]. A high-level view of the accelerator architecture is shown in Fig. 6, where Processing Elements (PEs) use adder graphs. The input X is fed into the PE array column per column. The systolic array, here called "PE array", operates with a WS strategy. Each PE corresponds to an MCM problem instance obtained from filter parallelization (see Fig. 1), and post-accumulation results are passed from one PE to the next.

We use HLS for our implementation. To do so, we integrated HLS generation for adder graphs in our tool `AdderGraphs`[2]. We utilized Xilinx AMD VITIS HLS 2022.2 and started by verifying that adder graph generation with HLS has the expected effect: our high-level description of an adder graph should lead to a *better* circuit than basic multiplications.

To test this, we compared adder graphs against simple constant/variable products ($c \times x$). For every 8-bit constant, we solved the associated SCM problem to obtain the best adder graph, and we produced the corresponding HLS. For example, the multiplication by 23 led to the following C++ code:

```
y = (x << 1 + x) << 3 - x;
```

We found that the number of LUTs in the resulting circuit was always lower when using adder graphs than the simple product `y = c*x;`. Hence, we used HLS for all our implementations and benefited from its ease of use for the complete CNN description. In particular, although in Fig. 6 PEs are represented with a shift-and-add implementation, HLS allowed us to easily switch from one implementation strategy to another and to compare different solutions (generic product vs shift-and-add).

V. Experiments

A. Experimental Setup

We evaluated our method using three CNN models: ResNet-18 and ResNet-20 [30] on the ImageNet-1K [31] and CIFAR-10 [32] datasets, respectively, for image classification, and VDSR-10 [33] on the Set5 [34] and Set291 (comprising 200 images from [35] and 91 images from [36]) datasets, for super-resolution.

For ResNet-18 and ResNet-20, we started our training with pretrained floating-point weights from timm [37] and from scratch, respectively. In both cases, we employed a Stochastic Gradient Descent (SGD) optimizer with a momentum of 0.9. For ResNet-18, we started with a learning rate of 7×10^{-3} and a weight decay of 4×10^{-5}. The learning rate was reduced by a factor of 0.1 every 5 epochs. The process took 20 epochs with batch size set to 256. For ResNet-20, we started with a learning rate of 3×10^{-3}, and a weight decay of 5×10^{-5}. The learning rate was reduced by a factor of 0.1 every 20 epochs. Training lasted for 100 epochs, with a batch size of

[2]https://github.com/remi-garcia/AdderGraphs

128. In case of VDSR-10, we used locally pretrained weights. Then, we ran the training over 30 epochs with a learning rate of 10^{-3}, SGD with 0.9 momentum as the optimizer, and a batch size of 4. The learning rate was reduced by a factor of 0.1 every 12 epochs, and the weight decay was set to 10^{-3}.

For the quantization, we targeted 8-bit fixed-point weights using DoReFa-Net [12], and we used PACT [8] for activations. For ResNet-18, every 4 epochs we fixed $p_{\text{global}} = 20\%$ of the weights based on the scoring strategy presented in Section III-B. For ResNet-20, we fixed $p_{\text{global}} = 10\%$ of the weights every 10 epochs. For VDSR-10 experiments, we also added gradient clipping in the training loop, as in [33], and we applied our strategy every 5 epochs, fixing $p_{\text{global}} = 16.6\%$ of the weights each time.

We have implemented the ILP model for SCM with fixed weights using the modeling language Pyomo [7], and we used the open-source tool `jMCM` [14] for the final implementation. In both cases, we used the Gurobi 12.0 ILP solver. We did not set any time limit on the solving time as models were small enough to be optimally solved in a few seconds, at most.

Finally, the hardware synthesis and simulation results have been done using Xilinx AMD VITIS HLS 2022.2 targeting a ZCU104 board. We have also developed an open-source tool `AdderGraphs` that automatically generates HLS for the adder graphs, and then combines them in a complete HLS accelerator flow for each CNN.

B. *Results*

In this section, we compare our approach with classic QAT and different AAT [20], [22], [23] approaches, confirming that a shift-and-add implementation of CNNs has a positive impact on the final hardware cost. We also want to verify that our progressive adder-aware training approach does not excessively degrade model accuracy, all while significantly reducing the overall cost. This investigation is directed by three main research questions:

RQ1 Does using a shift-and-add implementation significantly reduce FPGA hardware cost compared to using DSP arithmetic blocks or vanilla product operators?

RQ2 Does our shift-and-add aware training approach lead to smaller circuits than vanilla QAT?

RQ3 How does the progressive adder-aware training method perform in terms of accuracy compared to QAT and precomputed set-based approaches?

It would be interesting to compare the hardware resource consumption of our approach with state-of-the-art AAT methods to elaborate on *RQ2*. However, More AddNet [23] is not meant for the parallelization we included in our accelerator, thus a direct comparison would not be fair. Also, [22] did not target large CNNs as we do.

1) RQ1: Using Adder Graphs, LUTs or DSPs: In previous sections, we proposed an adder-aware training strategy. However, adder graphs might not be interesting for hardware implementation of CNNs and we need to assess their suitability first. To that end, we trained a ResNet-18 [30] CNN on ImageNet-1K [31] using QAT and compared the HLS synthesis results of (i) the vanilla multiplication (*Simple* $*$ lines in Table I), (ii) the multiplication with an additional pragma enforcing the use of DSPs ($*$ + *pragma* lines), and (iii) our adder graph description (*Adder graph* lines). In each case, we compare implementations of the second layer of the networks. Their dimensions for ResNet-20, ResNet-18 and VDSR-10 are $16\times16\times3\times3$, $32\times32\times3\times3$ and $54\times54\times3\times3$, respectively. The results are summarized in Table I.

TABLE I

LUT, FF and DSP consumption comparison of three different implementations on a Xilinx ZCU104 board for a convolution layer of multiple CNNs. Results are given in percentage of utilization of the complete board and include the implementation of the systolic array. Weights on 8 bits were obtained from a classic QAT flow.

CNN	Method	LUT (%)	FF (%)	DSP (%)	Latency (μs)
ResNet-20	Simple $*$	15.17	1.90	0	5.15
	$*$ + pragma	3.91	0.90	68.06	5.15
	Adder graph	8.49	1.54	0	5.15
ResNet-18	Simple $*$	64.94	9.44	0	5.16
	$*$ + pragma	–	–	$308.4^{\dagger}$	–
	Adder graph	41.82	7.05	0	5.16
VDSR-10	Simple $*$	$>100^{\dagger}$	–	–	–
	$*$ + pragma	–	–	$615.57^{\dagger}$	–
	Adder graph	97.87	14.59	0	5.16

†indicates an estimation as place and route was not possible.

In all cases, we obtained the same latency. Using HLS, we provide a target frequency and the synthesis tool added registers as needed. Thus, we explain that latency does not differ between methods due to the fact that most registers have probably been inserted in other parts of the accelerator, and not in the PEs. Hence, we believe that the latency is not driven by the PEs, but rather by the surrounding logic.

Enforcing the use of DSPs with a pragma is not reasonable: more than half of the DSPs of the board are used for a single layer of ResNet-20, a 20-layer CNN. For ResNet-18 and VDSR-10, the number of DSPs required for a single layer exceeds the available resources. Using adder graphs, on the other hand, is promising. Even with weights trained using simple QAT, *i. e.*, not specifically targeting this implementation, the shift-and-add implementation has a very low resource consumption. Our shift-and-add implementation uses half the resources of the simple $*$ operator in HLS.

The adder graph implementation is still too costly for a complete implementation of the ResNet-20 network on a single Xilinx ZCU104 board. Indeed, if all the layers had the same cost, it would require around 170% of the available LUT resources. Although the assumption of "same cost layers" is not correct, the need for LUTs largely exceeds the 100% limit, and thus the conclusion certainly holds. Our AAT approach could however reduce the resource consumption down to a reasonable level. For the other CNNs we evaluate, implementing the complete network on a single Xilinx ZCU104 board is certainly out-of-reach as almost half of the LUT resources are used just for a convolutional layer of ResNet-18 and the synthesis of a single layer of VDSR-10 does not even finish.

TABLE II

Comparison of the adder graph implementation cost between a convolution layer of two CNNs, with 8-bit weights obtained from QAT or with our approach. LUT and FF consumption is given in percentage of utilization of a Xilinx ZCU104 board, including the implementation of the systolic array. No DSPs were used.

CNN	Method	LUT (%)	FF (%)	Latency (μs)
ResNet-20	Classic QAT	8.49	1.54	5.15
	Our AAT	6.19	1.20	5.15
ResNet-18	Classic QAT	41.82	7.05	5.16
	Our AAT	23.46	4.23	5.16
VDSR-10	Classic QAT	97.87	14.59	5.16
	Our AAT	81.54	12.15	5.16

2) RQ2: Implementation Cost of the Shift-and-Add Aware Training Approach: We demonstrated that using adder graphs for the hardware implementation permits to reduce resource consumption w. r. t. automatic synthesis choices. Using a progressive adder-aware weight-fixing method, we expect that our approach should further reduce the hardware utilization.

To verify this hypothesis, we compare the implementation cost of a layer with weights obtained using QAT and our approach. Results are reported in Table II. Our progressive approach reduces the LUT consumption by 27% and the FF consumption by 22% for a layer of ResNet-20. If all the 20 layers were the same cost, implementing this CNN on the Xilinx ZCU104 board would lead to a 123.8% hardware utilization. Although this still exceeds available resources, we are closer to being able to implement a complete ResNet-20 network on a single Xilinx ZCU104 board. For a ResNet-18 layer, we achieve a reduction in LUT usage of over 43%. Although the complete implementation of a ResNet-18 network on a Xilinx ZCU104 board cannot be considered for the moment, this large reduction of LUT and FF utilization is very encouraging and demonstrates the efficiency of the method.

Finally, for a VDSR-10 layer, we obtain a hardware reduction of around 16%. For all these comparison, the cost of the systolic array is included and is the same for all methods. Hence, the gain on the part of the circuit we actually target is even larger.

3) RQ3: Accuracy of the Shift-and-Add Aware Training Approach: With this last research question, we want to measure the effect of our method on accuracy. It is well-documented [3] that quantization negatively impacts the accuracy of the network. Using QAT reduces the accuracy loss, but still degrades accuracy w. r. t. floating-point training and inference.

Our approach has the same hyperparameter settings as vanilla QAT, and incorporates an additional one by fixing weights during training. Hence, we lose a bit of flexibility compared to QAT and we cannot reasonably expect to have a better, or even the same, accuracy as QAT. However, compared to existing adder-aware techniques [20], [22], [23], which rely on precomputed sets of quantized values to round towards, we propose a more flexible method. Thus, we expect to have a better accuracy, or at least similar.

TABLE III

Accuracy results with ResNet-18 on ImageNet-1K, ResNet-20 on CIFAR-10 and VDSR-10 on Set5. Weights are quantized with 8 bits. The accuracy of ResNet-18 and ResNet-20 is the top-1 accuracy given as a percentage, and the accuracy of VDSR-10 is calculated for a $\times 3$ upscaling with PSNR expressed in dB.

	ResNet-18				ResNet-20		VDSR-10	
	[23]		Our		Our		Our	
	Top-1	rel.	Top-1	rel.	Top-1	rel.	Top-1	rel.
Float	73.2	100	73.1	100	92.8	100	34.03	100
QAT	72.7	99.3	73.0	99.9	92.7	99.9	33.97	99.8
Our	–	–	72.3	98.9	92.4	99.6	33.97	99.8
PoT	–	–	70.0	95.8	92.4	99.6	33.88	99.6
3-Add	72.0	98.4	–	–	–	–	–	–
2-Add	71.0	97.0	–	–	–	–	–	–
1-Add	61.9	84.6	–	–	–	–	–	–

Results are reported in Table III. On ResNet-18, with our set of training hyperparameters, we obtain a similar reference accuracy (using floating-point) as More AddNet. Compared to QAT accuracy results, we lose 0.7 percentage points of accuracy using our more constrained method. However, using a progressive approach allowed us to keep a top-1 accuracy above the 3-Add More AddNet results. On ResNet-20, our accuracy results are very close to the accuracy obtained with a vanilla QAT. For VDSR-10, our method and vanilla QAT led to the same PSNR results, and these results are only slightly below the reference floating-point training ones. Compared to a PoT approach [20], our method leads to a significantly better accuracy on the large ImageNet-1K dataset with ResNet-18. However, when using simpler datasets and models, ResNet-20 on CIFAR-10 and Super-Resolution with VDSR-10 in our case, both approaches performed similarly.

VI. Conclusion

Adder-Aware Training (AAT) methods, and hardware implementation-aware training in general, have been getting increased attention over the last few years. There exists an approach adapted to small neural network controllers [22] and another for larger CNN implementations [23]. This second approach is not meant for parallel implementation. With our work, we fill this gap in the literature. We propose a progressive training method to obtain CNN weights that are well-suited for a parallel implementation of the neural network. With our hardware experiments on ResNet-18, ResNet-20, and VDSR-10, we demonstrate that our method does not degrade network accuracy much, and that it leads to a significant reduction in implementation cost, by around 25% w. r. t. vanilla QAT. Our method can be directly applied to any neural network in which computations can be parallelized. In particular, it would be interesting to have a direct comparison with previous AAT work [22] on smaller neural networks.

To compute scores at each fixing step, we solve an ILP model per weight. We show that, although this increases training times, solving ILP models does not bottleneck training time. Still, an increase in the number of weights will automatically induce a longer optimization time. To overcome this issue, it might be interesting to develop a dedicated heuristic for this problem or to adapt existing ones [15], [16]. Finally, as CNNs are robust to small weight changes [38], these weights could be adjusted to lead to an implementation with a smaller cost [6]. In our work, we first quantize weights and then compute their score. An interesting path to further reduce the implementation cost would be to merge both ideas by computing the score of multiple approximations of each weight and only retain the best solution.

References

[1] O. Gustafsson and F. Qureshi, "Addition Aware Quantization for Low Complexity and High Precision Constant Multiplication," *IEEE Signal Processing Letters*, vol. 17, no. 2, pp. 173–176, Feb. 2010.

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

[3] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 3123–3131.

[4] K. Goetschalckx and M. Verhelst, "Breaking High-Resolution CNN Bandwidth Barriers With Enhanced Depth-First Execution," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 323–331, Jun. 2019.

[5] A. D. Booth, "A Signed Binary Multiplication Technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[6] S.-H. Bae, H.-J. Lee, and H. Kim, "MCM-SR: Multiple Constant Multiplication-Based CNN Streaming Hardware Architecture for Super-Resolution," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–13, 2024.

[7] M. L. Bynum, G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff, *Pyomo–optimization modeling in python*, 3rd ed. Springer Science & Business Media, 2021, vol. 67.

[8] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized Clipping Activation for Quantized Neural Networks," 2018.

[9] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[10] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," *arXiv:1902.08153*, 2019.

[11] Y. Bhalgat, J. Lee, M. Nagel, T. Blankevoort, and N. Kwak, "LSQ+: Improving low-bit quantization through learnable offsets and better initialization," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, Jun. 2020, pp. 2978–2985.

[12] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," 2016.

[13] R. Bernstein, "Multiplication by integer constants," *Software: Practice and Experience*, vol. 16, no. 7, pp. 641–652, Jul. 1986.

[14] R. Garcia and A. Volkova, "Toward the Multiple Constant Multiplication at Minimal Hardware Cost," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 5, pp. 1976–1988, 2023.

[15] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms*, vol. 3, no. 2, p. 11, May 2007.

[16] M. Kumm, P. Zipf, M. Faust, and C.-H. Chang, "Pipelined adder graph optimization for high speed multiple constant multiplication," in *2012 IEEE International Symposium on Circuits and Systems*. IEEE, May 2012.

[17] M. Kumm, *Multiple Constant Multiplication Optimizations for Field Programmable Gate Arrays*. Wiesbaden: Springer Fachmedien Wiesbaden, 2016.

[18] ——, "Optimal Constant Multiplication Using Integer Linear Programming," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 567–571, May 2018.

[19] R. Garcia and A. Volkova, "Multiple Constant Multiplication: From Target Constants to Optimized Pipelined Adder Graphs," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. Gothenburg, Sweden: IEEE, Sep. 2023.

[20] M. Elhoushi, Z. Chen, F. Shafiq, Y. H. Tian, and J. Y. Li, "DeepShift: Towards Multiplication-Less Neural Networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Jun. 2021, pp. 2359–2368.

[21] J. Li, M. Yanagisawa, and Y. Shi, "An Area-Power-Efficient Multiplierless Processing Element Design for CNN Accelerators," in *2023 IEEE 15th International Conference on ASIC (ASICON)*. IEEE, Oct. 2023, pp. 1–4.

[22] T. Habermann, J. Kühle, M. Kumm, and A. Volkova, "Hardware-Aware Quantization for Multiplierless Neural Network Controllers," in *2022 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, Nov. 2022.

[23] M. Hardieck, T. Habermann, F. Wagner, M. Mecik, M. Kumm, and P. Zipf, "More AddNet: A deeper insight into DNNs using FPGA-optimized multipliers," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. abs/1308.3432. IEEE, May 2023, pp. 1–5.

[24] H. Wang, C. Qin, Y. Bai, and Y. Fu, "Why is the State of Neural Network Pruning so Confusing? On the Fairness, Comparison Setup, and Trainability in Network Pruning," *arXiv:2301.05219*, 2023.

[25] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 10 882–11 005, 2021.

[26] J. Thong and N. Nicolici, "An Optimal and Practical Approach to Single Constant Multiplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 9, pp. 1373–1386, 2011.

[27] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*. Springer International Publishing, 2020.

[28] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006.

[29] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, ser. MM '14. ACM, Nov. 2014.

[30] 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)*. IEEE, Jun. 2016, pp. 770–778.

[31] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Apr. 2015.

[32] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 (Canadian Institute for Advanced Research)."

[33] J. Kim, J. K. Lee, and K. M. Lee, "Accurate Image Super-Resolution Using Very Deep Convolutional Networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Jun. 2016.

[34] M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. Alberi-Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor embedding," in *Proceedings of the 23rd British Machine Vision Conference*. BMVA press, 2012.

[35] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, ser. ICCV-01, vol. 2. IEEE Comput. Soc, 2001, pp. 416–423.

[36] R. Zeyde, M. Elad, and M. Protter, *On Single Image Scale-Up Using Sparse-Representations*. Springer Berlin Heidelberg, 2012, pp. 711–730.

[37] R. Wightman, H. Touvron, and H. Jégou, "Resnet strikes back: An improved training procedure in timm," *arXiv:2110.00476*, 2021.

[38] D. T. Nguyen, N. H. Hung, H. Kim, and H.-J. Lee, "An Approximate Memory Architecture for Energy Saving in Deep Learning Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1588–1601, May 2020.