Lossless Neural Network Model Compression Through Exponent Sharing

Prachi Kashikar, Olivier Sentieys, Sharad Sinha

Lossless Neural Network Model Compression Through Exponent Sharing

Prachi Kashikar, Olivier Sentieys, *Member, IEEE*, and Sharad Sinha, *Senior Member, IEEE*

***Abstract*— Artificial intelligence (AI) on the edge has emerged as an important research area in the last decade to deploy different applications in the domains of computer vision and natural language processing on tiny devices. These devices have limited on-chip memory and are battery-powered. On the other hand, neural network (NN) models require large memory to store model parameters and intermediate activation values. Thus, it is critical to make the models smaller so that their on-chip memory requirements are reduced. Various existing techniques like quantization and weight-sharing reduce model sizes at the expense of some loss in accuracy. We propose a lossless technique of model size reduction by focusing on the sharing of exponents in weights, which is different from the sharing of weights. We present results based on generalized matrix multiplication (GEMM) in NN models. Our method achieves at least a 20% reduction in memory when using Bfloat16 and around 10% reduction when using IEEE single-precision floating point, for models, in general, with a very small impact (up to 10% on the processor and less than 1% on FPGA) on the execution time with no loss in accuracy. On specific models from HLS4ML, about 20% reduction in memory is observed in single precision with little execution overhead.**

***Index Terms*— Convolutional neural network (CNN), FPGA, generalized matrix multiplication (GEMM), model compression, Vivado HLS.**

I. Introduction

THE convolutional neural networks (CNNs) in machine learning are increasingly used in different applications ranging from image classification [1] to object detection [2], activity recognition [3] in medical diagnosis [4], military [5], agriculture [6], sports [7], and so on. Convolutional and fully connected layers in CNNs have a huge number of parameters and feature maps. During training as well as inference, a lot of memory and power are consumed by these compute-intensive layers. Therefore, for training, high-end GPUs or cloud systems are used. Still, the training time ranges from days to weeks for complex state-of-the-art models such as ResNet, GoogleNet, and so on. Their inferences also have thousands of parameters that require storage in the orders of MBs.

The performance improvement in neural network (NN)-based applications has been accelerating in this last decade. However, the software side advances for the deployment of such models have not considered the characteristics of underlying hardware while adding multiple layers to improve the performance. Thus, to deploy models on the edge, hardware–software co-design and co-development become needful [8]. With the advent of the Internet of Things (IoT) and mobile platforms, it is expected that artificial intelligence (AI) applications will run on such devices. The target hardware platforms to run machine-learning models, in this case, are small in size having limited memory and power budgets. It is challenging to deploy existing big models on such tiny devices [9]. This has motivated researchers to compress models and find the best tradeoffs with accuracy.

Model compression is possible during training as well as posttraining. The existing model compression methods like quantization [10], weight sharing, and pruning [11], reduce the model size by sacrificing some accuracy. We propose a lossless model compression technique that exploits exponent distribution in weights for a trained model and show how exponent sharing in weights, an intuitive but powerful method, helps in reducing memory requirements. It is worthwhile to note that the proposed "exponent sharing in weights" is different from the existing method of "sharing of weights." The latter targets weight sharing (i.e., making weights common) by sacrificing accuracy, while the former does not sacrifice accuracy. It is logical to assume that one may apply exponent sharing on top of weight sharing or other existing compression methods, one combination of which (pruning followed by proposed exponent sharing) we present results for in Section IV. We discuss the impact of exponent sharing on the memory and execution time of processors and FPGAs. Ultralow power machine-learning implementation of tasks, such as keyword spotting and visual wake words [12], can benefit from model compression approaches.

Additionally, we present results on various combinations of GEMM dimensions, a few pretrained standard models, and models from HLS4ML [13]. We discuss the impact on resource usage when the reading of weights is sequential and parallel. The impact of using different initiation intervals during pipelining in FPGA implementations is also discussed. We also present the application of our method on a few

Manuscript received 16 April 2023; revised 6 July 2023 and 2 August 2023; accepted 8 August 2023. This work was supported by DST-INRIA through CEFIPRA, India, under Project IFC/4131/DST-Inria/2018-2019/1. *(Corresponding author: Prachi Kashikar.)*

Prachi Kashikar is with the Indian Institute of Technology Goa, Goa 403401, India (e-mail: prachi183311004@iitgoa.ac.in).

Olivier Sentieys is with Univ. Rennes, INRIA, 35042 Rennes, France (e-mail: olivier.sentieys@inria.fr).

Sharad Sinha is with the School of Mathematics and Computer Science, Indian Institute of Technology Goa, Goa 403401, India (e-mail: sharad@iitgoa.ac.in).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2023.3307607.

Digital Object Identifier 10.1109/TVLSI.2023.3307607

relatively large models, though it is not the focus of the work in this article.

The article is arranged as follows. Section II discusses some existing model compression techniques and their shortfalls. In Section III, we discuss our proposed method of exponent sharing to reduce on-chip memory requirements. We also present a model to find out the impact of exponent sharing on the execution overhead (in terms of clock cycles) on FPGAs. We demonstrate our approach in Section IV with experiments and results and conclude in Section V.

II. Related Work: Model Size Compression

The accuracy of a trained model depends on the precision it uses to store its parameters. In general, higher accuracy needs higher precision and, therefore, greater storage volume. Existing model compression techniques focus on reducing the precision of parameters while minimizing the tradeoff with model performance. We discuss some of the mainstream techniques in the following subsections.

A. Algorithmic Approaches of Model Compression

1) Quantization: Quantization is one of the most popular methods for model size compression. It maps (bins) the range of the weights to a smaller set of finite values. As the size of the bins in quantization increases, the precision to save the weights reduces. Researchers could reduce the precision to even a single bit [14] in certain application areas. On the other hand, as the bin sizes increase, accuracy loss also increases due to bigger approximations.

All bins in quantization can be of fixed or variable sizes. In the fixed-size approach, weights are binned independent of their impact on the accuracy. This can cause more loss in accuracy. As a solution, genetic algorithms are used to optimize bin sizes for the best accuracy [15]. These algorithms bin weights depending on their impact on model accuracy. This results in the variable size of bins. As this impact of weights on accuracy is not known during training, quantization using variable bin size is done during inference.

The range of weights in all layers of an NN may not be the same. Besides applying the same quantization policy for a complete model, layer-wise different quantization scheme [10] shows more compression for the same accuracy. Neural architectural search (NAS) [16] helps in identifying the best bit widths for weights per layer during training. As a result, the model gets quantized in mixed precision where each layer needs different bit-widths.

Quantization can be done during as well as posttraining. Incremental network quantization [17] combines both. It divides weights into two sets after training. One set holds the low precision weights. Viable bit lengths are found for this set. The other set having higher precision is retrained and the same process continues until all weights take lower bits in memory. After multiple passes, the model gets quantized.

Along with the weights, the number of activations also increases the memory requirement of a model. For more storage savings, the joint statistics of weights and activation can be considered. The transform quantization [18] does this and improves the suboptimal storage savings obtained after only weight quantization.

Quantization is a magnitude-aware process, in general. However, along with memory storage, few works also look at different constraints such as energy [19], sparsity [20], and hardware [21] for quantization.

2) Pruning: Pruning removes the weights, neurons, or feature maps from the model for its size reduction. A few weight tensors are sparse, redundant, and sometimes have the least impact on the accuracy of the network. Removal of these weights does not lead to a significant loss of accuracy. Pruning removes such redundant weights and neurons and the model size is reduced. It results in accelerated training and inference speed. Pruning can also remove a complete filter from a model. To identify the least valued filters, a model can be fed to Network Pruning Network [22] architecture. This gives binary-valued output that indicates the least significant filters to be pruned which, in turn, saves model storage requirement. Similarly, ThinNet [23] proposes to discard the least impacting filters during convolution completely. It results in more memory savings. It proposes to prune filters depending on the features they propagate to the next layer. So, the least important features are traced through back-propagation, and filters are pruned. Similarly, filter pruning using activation maximization is proposed in [24], which removes the least important convolutional filters.

In layer-wise weight pruning, the least significant weights in a layer are removed and the error propagated to the very next layer is only minimized. However, in practice, this error reaches the output layer of the model causing a substantial loss of accuracy. Yu et al. [11] proposed a neuron importance score propagation (NISP) algorithm to prune neurons jointly from a complete network to minimize the error. Also, the layer-wise weight pruning breaks the structure of state-of-the-art models like VGG as they have dimensional dependencies. It makes the training intractable. Such complex models cannot use single compression techniques for optimal performance. Aghli and Ribeiro [25] make use of the combination of pruning on selective layers with knowledge distillation (discussed in Section II-A3) on the remaining ones.

With the storage reduction, power and energy budgets are also of concern while compressing a model. The layer-wise weight pruning followed by the fine-tuning results in moderately less energy consuming [26]. After layer-wise pruning, the global fine-tuning improves performance.

Pruning can also remove the redundant channels generated during training [27] which does not need any further retraining. For a layer, the channels to prune are identified by regression-based channel selections and least-square reconstruction [28].

3) Knowledge Distillation: Knowledge distillation follows a student–teacher model. The learning from a complex structured teacher model generates a student model (i.e., an ensemble). Ensembles are lightweight and generated in a block-wise manner. An ensemble can combine different features of different teacher models. The student network is distilled by learning application-specific resources such as floating-point

operations (FLOPs) and model parameters. This ensures the best compression-performance tradeoff [29].

These block-wise knowledge distillation methods are complex and take longer training times. However, in depth-wise separable layers, the independent blocks are trained in parallel. Experiments on the state-of-the-art models have shown up to 3.5× speedup using this method during training [30]. As a result of compression, models may also show acceleration in performance. Most of the time the convolutional and fully connected layers accelerate separately. Lin et al. [31] propose to jointly distill layers by first removing redundancies in both convolutional and fully connected layers and then doing the knowledge transfer.

All these popular methods of knowledge distillation, pruning, and quantization can be used in combination with each other for more compression and optimal performance [32]. Usage-driven model selection framework AdaDeep [33] compressed models using a combination of existing techniques. For optimal compression, a model can pass through a series of compression techniques in a pipeline [34], achieving the best of all inferences. The pipeline can contain methods like tensor decomposition, graph pruning, knowledge distillation, and so on.

B. Implementation Oriented Approaches for Model Compression

Communication with off-chip memory affects the speed of inference. Even after compression, the model size can reach several megabytes. Hence, running inference in three phases, that is, layer partition, compression, and scheduling, showed promising results [35] in terms of storage budgets. Storing the parameters in the block floating-point format [36] in the off-chip memory saves some storage and also reduces the communication delay with off-chip memory. Although this format improves the energy and efficiency of the hardware, it also loses some accuracy. The energy efficiency can be further improved by 5.6× focusing on memory accesses made during edge computing [37].

The existing techniques do not always emphasize the tradeoff between performance and resource constraints. The architectures of CNNs are different from each other. Other than using a general method for all of them, a model-specific approach results in better compression. For example, Tiny-PULP-Dronets [38] based on ResNet aims at thinning the number of channels per layer and the neuron which never get activated during the entire validation process without much penalizing the accuracy. It could reduce the model size by more than one order of magnitude (50× fewer parameters) at the cost of 4% accuracy loss. Navardi et al. [39] propose a complete framework to deploy such tiny vision-based models on drones that also reduce the energy requirement by 53%, while preserving 97% of the model accuracy. The parallel execution of DNNs [40] on drones can be exploited to perform real-time navigation with only 64 mW of power.

Most existing methods compress models at the cost of some accuracy loss. In this work, we propose a novel model compression method that keeps the accuracy intact, by studying sharing of exponents in weights. This method can be used for saving both on-chip and off-chip memory requirements for the network weights.

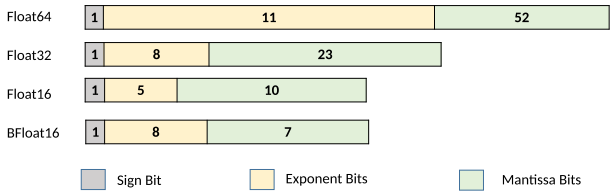

Fig. 1. IEEE and Bfloat16 floating-point representation formats.

III. Exponent Sharing for Model Compression

The size of a model is determined by the number and the representation (integer, float, etc.) of its parameters. The parameters comprising weights and biases are generally floating-point values. In contrast with fixed-point representations, floating-point formats offer greater precision and represent a wider range of real numbers. As per the IEEE 754 standard for floating-point representation [41], every floating-point value is stored as a combination of sign, exponent, and mantissa, as shown in Fig. 1. BFloat16 [42] is a reduced-precision floating-point format where data are stored on 16 bits, multiplications are 16 bits, and accumulations are usually 32-bit wide. The difference between IEEE half-precision (Float16) and BFloat16 is that BFloat16 has a higher dynamic range with 8 bits of exponents, which also makes it easier to cast to Float32. The range in half-precision is reduced because of the decrease in the number of exponent bits, as shown in Fig. 1. We exploit these standard floating-point storage formats to come up with the new storage format.

A. Proposed Floating-Point Storage Format

In IEEE single-precision format, an exponent can have 256 different values, but, in practice, there are thousands of weights in a model. Hence, many of them have the same exponent values. Fig. 2 shows an example of the exponent frequency distribution in weights of a trained LeNet. Out of all the trained weights, only 6.25% exponents are distinct, and the rest of the values are not used for weights.

Hence, we propose a new floating-point storage format based on an algorithmic pass following Algorithm 1 that exploits the presence of several exponents with the same values. The proposed floating-point storage format is illustrated in Fig. 3. Along with IEEE 754 and Bfloat16, it applies to any real value representation format that makes use of the mantissa-exponent method for number representation.

In this novel method, we store only the distinct exponents of floating-point weights in a separate exponent table. The exponents from IEEE floating-point formats are replaced by respective indices referring to the exponent table. Each floating-point weight then becomes a combination of *sign*, *index*, and *mantissa*, where the *index* replaces the *exponent* from the original representation.

If there are k distinct exponents in the weights of a layer, the number of index bits required is $i = \lceil \log_2 k \rceil$, which we found

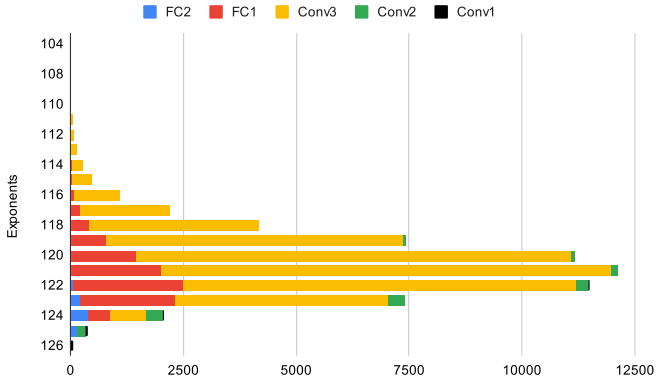

Fig. 2. Exponent frequency distribution in LeNet.

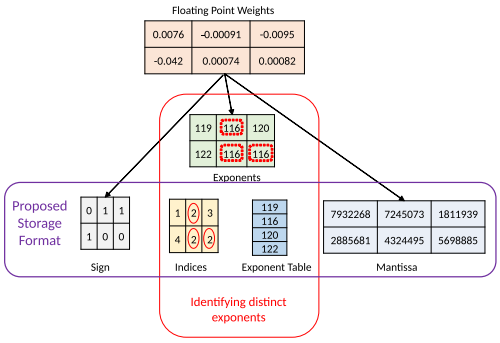

Fig. 3. Illustration of storing weights in the proposed format.

will be always less than 8 bits during our experiments. Hence, if s, e, and m denote the number of bits for sign, exponent, and mantissa, respectively, in the IEEE floating-point or Bfloat16 representations, then the memory needed after exponent sharing (M_{comp}) is

$$M_{\text{comp}} = N \times (s + i + m) + e \times k \tag{1}$$

where N is the total number of weights in a layer and i is the number of index bits.

B. Proposed Optimization Pass for Trained Models

Algorithm 1 presents the proposed optimization pass. It can be applied as a *follow on* pass after any other existing model compression methods as it will not change the accuracy.

Like other compression techniques [43], our method also shows better results on layer-wise compression than on compressing a complete model. This optimization pass is applied to the floating-point weights of every layer which are initially stored in standard formats like the IEEE floating-point standard as shown in Fig. 1. In Algorithm 1, the weight tensor of a layer is segregated into three components: Sign (S_o), Exponent (E), and Mantissa (M_o), in lines 2–6. The distinct exponents (E_o) are identified from E in line 7. All exponents in E are henceforth referred to by their indices in E_0. These indices are stored in a separate Index (I_o) tensor as per lines 8–10. In the end, the algorithm returns tensors S_o, I_o, and E_o of sign, index, and mantissa, respectively, for every weight in the model. The algorithm also reports memory requirements before and post sharing, in lines 12 and 13.

Algorithm 1 Optimization Pass on Trained Models

Data: Weight Tensor (W_i)
Result: Tensors of Sign (S_o), Index (I_o), Mantissa (M_o) and Exponent List (E_o)

1 Calculate memory requirement (M_{orig}) for W_i ;
2 **while** w *in* W_i **do**
3 $S_o \leftarrow$ sign of w ;
4 $E \leftarrow$ exponent of w ;
5 $M_o \leftarrow$ mantissa of w ;
6 **end**
7 Find distinct exponents (E_o) in E ;
8 **while** e *in* E_o **do**
9 Find Index i for e ;
`/* i is an index of e in E */`
10 Place Index i in I_o corresponding to e in E_o ;
11 **end**
12 Calculate memory requirement after Exponent Sharing (M_{comp}) using equation 1 ;
13 Report memory savings (M_{orig} - M_{comp}) Return S_o, I_o, M_o and E_o

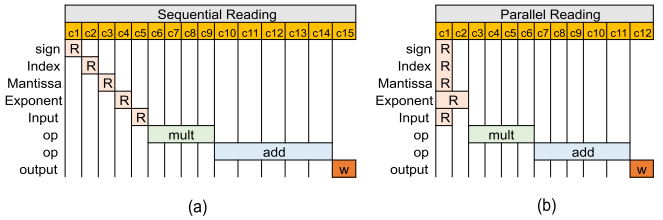

Fig. 4. Clock cycles during exponent-sharing in matrix multiplication with and without pipeline. (a) Sequential Reading. (b) Parallel Reading.

C. Hardware Implementation of Model Postoptimization Pass

The application of the optimization pass results in a model with a different storage method for weights, as shown in Fig. 3. This affects the execution time of the model. The execution time of a model depends primarily on the number of generalized matrix multiplications (GEMMs) in each layer. However, it is possible to reduce the execution time of each layer using GEMM optimizations present in the research literature, though we do not discuss or implement such methods as part of this article. In our proposed indexing-based method, three reads are required compared to a single read when such indexing is absent. This would, in general, increase the execution time in the case of single-threaded implementations on microprocessors. On the other hand, this overhead in execution can be made negligible by doing parallel reads on accelerators like FPGAs [44].

We discuss sequential versus parallel architectures in the next two subsections.

1) Sequential Architecture: Sequential reads are performed in sequential architecture or execution model where a single weight value is fetched from on-chip memory at a time. In our proposed method, three additional reads are necessary to read the same weight, as shown in Fig. 4(a), where *c1*, *c2*, and so on, refer to clock cycle number in the schedule, compared to presharing. For a GEMM of any layer, if the weight tensor is

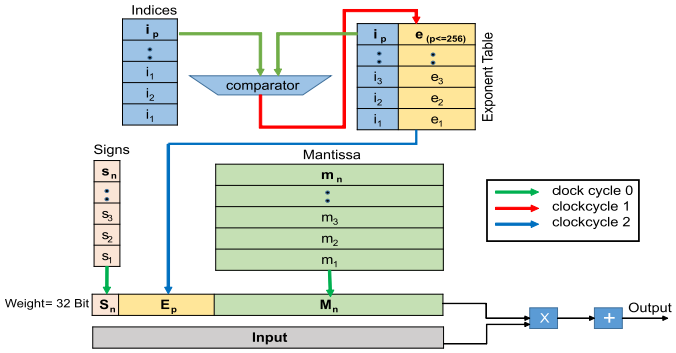

Fig. 5. Hardware implementation of proposed storage scheme (32-bit example).

$M \times N$ and the input tensor is $N \times O$, then the cycle count can be given by

$$C_{\text{expSharing}} = C_{\text{orig}} + M \times N \times O \tag{2}$$

where C_{orig} is the number of clock cycles before exponent sharing and $C_{\text{expSharing}}$ is the number of clock cycles after exponent sharing.

2) Parallel Architecture: We observed about a 9%–10% impact of exponent sharing on the execution cycles when non-pipelined/sequential hardware is used. As mentioned earlier in this section, this impact is due to multiple reads happening to read a single floating-point weight. To reduce this impact, we designed parallel architecture where reads are done in parallel as shown in Fig. 5. Here, multiple reads are possible in the same clock cycle due to multiport on-chip memories like Block RAM, as shown in Fig. 4(b). We used pragmas like *Pipeline* in the high-level synthesis tool Vivado HLS to parallelize operations, including memory reads. In this scenario, the cycle count is given by

$$C_{\text{expSharing}} = C_{\text{orig}} + M \times O. \tag{3}$$

Using (2) and (3), we can find out the execution cycle overhead that would result because of exponent sharing in a GEMM of any layer.

IV. Experiments and Results

In this section, we present results for small models, like tiny-tiny-tiny YOLO (TTT-YOLO), which are used for smaller computing devices, as well as for some relatively large models. This is to show that the proposed work can be used for the reduction in the size of large models as well.

A. Compression Results

We have demonstrated our approach on the layers of the TTT-YOLO model from the Darknet framework with pretrained weights [45]. This model is a tinier version of YOLO having eight convolutional layers. Table I shows the dimensions of GEMM in each of the layers of this model.

Table II reports the total weight memory in bits *Before* exponent sharing and *After* exponent sharing as well as the compression ratio achieved in percentage, both for Float32 and Bfloat16, on all layers of a TTT-YOLO model. The exponent

TABLE I
GEMM Sizes of Layers in Tiny-Tiny-Tiny Yolo

Layer	Input	Filters	GEMM Size
Conv	224x160x3	(3x3x3)x16	[16x27][27x35840]
Conv_1	112x80x32	(3x3x16)x32	[32x144][144x8960]
Conv_2	28x28x32	(3x3x32)x128	[128x288][288x560]
Conv_3	7x5x128	(3x3x128)x512	[512x1152][1152x35]
Conv_4	7x5x512	(3x3x512)x512	[512x4608][4608x35]
Conv_5	7x5x512	(1x1x512)x256	[256x512][512x35]
Conv_6	7x5x256	(3x3x256)x512	[512x2304][2304x35]
Conv_7	7x5x512	(1x1x512)x125	[125x512][512x35]

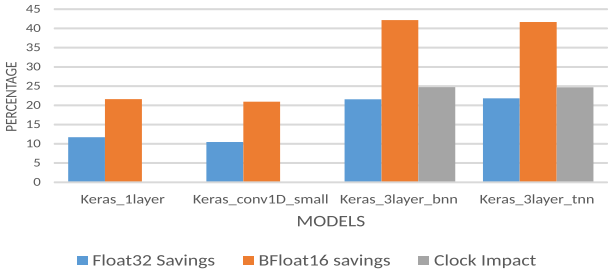

Fig. 6. Exponent sharing in models from HLS4ML.

sharing saved 9.37% and 18.75% on average for Float32 and Bfloat16, respectively.

Along with the layers of a TTT-YOLO, we have experimented with GEMMs of various sizes and random weights to generalize our analysis as shown in Table III. This table also reports the % distinct exponents out of 256. E1–E8 are different GEMMs with randomly generated weights. The compression achieved by our exponent-sharing method reaches 12.5% and 25% for Float32 and Bfloat16, respectively. The memory savings depends on the degree of distinctness in the exponents. Hence, the proposed method is best suited when a layer has less number of distinct exponents but with high frequency so that the indexing (i) will require fewer bits and hence more memory savings as expressed in (1).

We also evaluated exponent sharing on some models from HLS4ML [13]. This tool generates HLS models having fixed ⟨16,6⟩ as a default precision. We converted it to Float32 and BFloat16 and then performed exponent sharing. The memory savings (in %) observed on different models after exponent sharing are shown in Fig. 6. The gains range from 12% to 21.5% for Float32 and from 20% to 43% for BFloat16.

B. Execution Time and Resource Overhead

To find out the applicability of proposed (2) and (3) reporting the execution time overhead, we measured the execution time with and without exponent sharing on various architectures.

1) Execution Time Overhead for Hardware Implementations: Table III reports the impact on the number of clock cycles (%) for the exponent sharing with and without pipelining, for GEMMs of different sizes and random weights. The performance reported is using Vivado HLS set for part xc7z020clg484-1 on Zynq Zedboard with a default clock period of 10 ns. From the sets of experiments, we observe that the clock cycle impact is independent of the exponent

TABLE II
COMPRESSION ON THE LAYERS OF TTT-YOLO

Layer	Memory in Bits					
	32 Bits			Bfloat16		
	Before	After	% Saved	Before	After	% Saved
Conv	13824	12200	11.748	6912	5288	23.495
Conv_1	147456	133776	9.277	73728	60048	18.555
Conv_2	1179648	1069220	9.361	589824	479392	18.723
Conv_3	18874368	17105100	9.374	9437184	7667920	18.748
Conv_4	75497472	68419800	9.375	37748736	30671000	18.75
Conv_5	4194304	3801250	9.371	2097152	1704100	18.742
Conv_6	37748736	34210000	9.374	18874368	15335600	18.749
Conv_7	2048000	1856160	9.367	1024000	832160	18.734
Total	**139703808**	**126607506**	**9.374**	**69851904**	**56755508**	**18.749**

TABLE III
COMPRESSION (%) AND CLOCK CYCLE IMPACT (%) OF EXPONENT SHARING IN THE GEMMS OF SAME DIMENSIONS AND DIFFERENT WEIGHTS

Label	GEMM Dimensions	% Distinct Exponents	Memory in Bits						% Impact on Clock Cycles (Float32)	
			Float32			Bfloat 16			Without Pipelining	With Pipelining
			Before	After	% Saved	Before	After	% Saved		
E1	[250x256][256x16]	6.64	2048000	1856140	9.37	1024000	832136	18.74	9.084	0.077
E2	[128x256][256x169]	6.25	1048576	917632	12.49	524288	393344	24.98	9.084	0.077
E3	[64x256][256x16]	5.47	524288	458864	12.48	262144	196720	24.96	9.084	0.077
E4	[250x256][256x16]	6.25	2048000	1792130	12.49	1024000	768128	24.99	9.084	0.077
E5	[128x256][256x169]	5.86	1048576	917624	12.49	524288	393336	24.98	9.084	0.077
E6	[64x256][256x16]	5.08	524288	458856	12.48	262144	196712	24.96	9.084	0.077

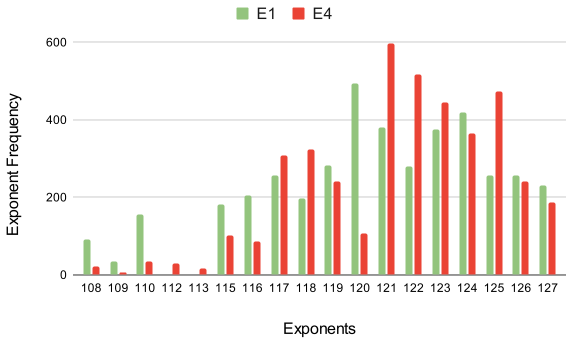

Fig. 7. Exponent frequency distribution in E1 and E4 of Table III.

TABLE IV
IMPACT (%) ON FPGA RESOURCES POSTEXPONENT SHARING IN TTT-YOLO

Layer & GEMM Dimensions		Without Pipeline		With Pipeline	
		FF	LUT	FF	LUT
conv_2	[128x288][288x560]	2.91	3.11	2.41	2.57
conv_5	[256x512][512x35]	2.55	3.22	2.08	2.66
conv_7	[125x512][512x35]	2.59	3.25	2.1	2.68

frequency distribution but dependent on the GEMM sizes of that layer. For example, in Table III, E1 and E4 have the same GEMM dimensions ([250 × 256][256 × 16]) and the same impact on clock cycles (9.084), although the exponent frequency distribution is different, as shown in Fig. 7. Depending on the GEMM dimensions, the number of multiplications and additions are different leading to different impacts on clock cycle count. Here, both E1 and E4 have 1 024 000 multiplications and 1 020 000 additions, and hence the clock cycle impact is also the same. This is also validated using (1) and (2).

As sequential reading after exponent sharing needs three more reads for every weight, the performance is impacted as per (2). The impact on clock cycles is less than 10% percent in all of these cases as summarized in Table III. We have demonstrated exponent sharing on GEMM with *Pipeline* pragma from Vivado HLS resulting in parallel reads. The performance impact abides by the relationship shown in (3). In all of the cases, the impact is less than 1%.

Second, Fig. 6 also reports the execution speed impacts observed on different models from HLS4ML generated for the Virtex7 FPGA xc7vx1140t-flg1926-2. It is important to note that the proposed exponent sharing did not show any execution overhead in small models like *Kerasllayer* and *KerasConv1d*. These models differ in architecture as one has a dense layer and the other has a convolutional layer. Both models have 704 weights in total which is smaller than any of the GEMMs used in the earlier experiments. However, the memory savings in both cases are different as they depend upon the spread of exponents in weights. As the number of layers (GEMMs) in a model increased, we observed there is more impact on execution speed like in the case with the *Keras3LayerBinarySmall* and *Keras3LayerTernarySmall*. The cumulative impact is because of loading weights between two consecutive GEMMs. In both cases, the memory savings are 21% and 43% in Float32 and Bfloat16, respectively. The impact on clock cycles in both models was 24%. In this case, the weights are read in sequential architecture, that is, without using pipelining.

2) Resource Overhead: Along with the clock cycles, resource utilization plays a vital role in choosing an algorithm for model compression. We chose layers II, V, and VII as

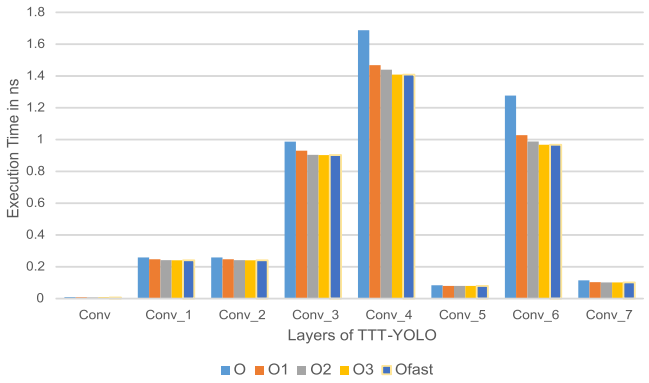

Fig. 8. Optimization on the layers of TTT-YOLO after exponent sharing on default processor in N2D2.

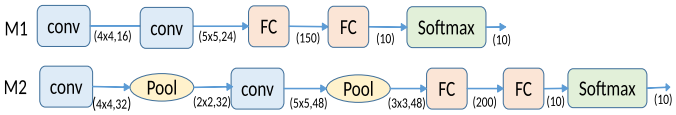

Fig. 9. Architecture of custom models M1 and M2 in N2D2.

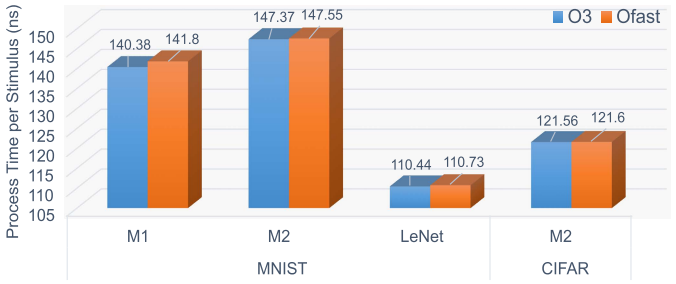

Fig. 10. Levels of optimization on the models of N2D2 after exponent-sharing.

representative of layers having a number of elements in their matrices that can fit in the on-chip memory of the Zynq device on the ZedBoard. This approach is complementary to existing works on CNN accelerators on FPGA where the entire model is not implemented on the FPGA [46]. Table IV summarizes the resource utilization of some of the GEMM kernels from the TTT-YOLO benchmark, in the sequential (without pipeline) and parallel (with pipeline) reading scenarios. Without a pipeline, the increase in the number of LUTs and FF is around 3% in all cases. When relying on a pipeline, the impact on FFs and LUTs is lower than 3% in all cases. For both architectures, there is no change in BRAM and DSP utilization.

3) Execution Time Overhead for Software Implementations: The execution time on a processor is impacted by the architecture, the language, and the compiler used [47]. Languages like C by default save matrices in *row-major* order as they show better execution time because of the principle of locality. We demonstrate the effect of compiler optimizations on *intel-i7* processor on the layers of TTT-YOLO in Fig. 8. In all layers, we observed that the execution times with exponent sharing are the least when optimized with the *O3* flag in *gcc*.

We have demonstrated our method on models trained and exported from N2D2 [48]. We chose models M1 and M2

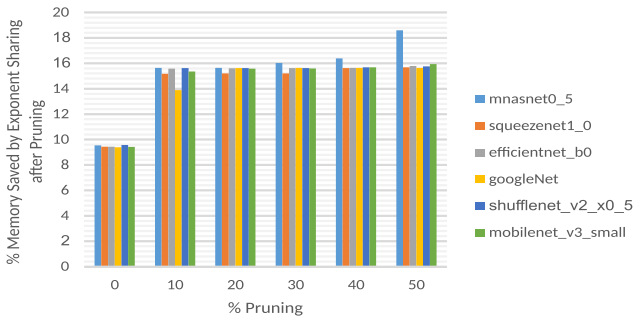

Fig. 11. Exponent sharing after weight pruning on standard models.

having architectures as shown in Fig. 9. N2D2 by default exports the model with the *O3* flag in gcc. We observed that there was no further improvement in execution time with optimization using *Ofast* flag as shown in Fig. 10. The default results of execution time for these models are presented in our prior work [49].

C. Incremental Pipeline Initiation Intervals on FPGA

The processing of two consecutive inputs can be decided using the loop pipeline initiation interval (II). Other than processing every II number of cycles, the interval can be varied depending on the availability of the resources. We found that with exponent sharing the impact after the initiation interval of 5 decreases but the total clock cycles keep increasing. This is expected behavior because, with higher II, there is delayed processing of inputs leading to increased clock cycle count. However, it creates the opportunity to parallelize more reads.

On three layers of the TTT-YOLO, we show the effect of incremental pipeline initiation intervals in Table V. It can be seen that with higher II, the clock cycle count overhead decreases. For instance, the overhead decreases from 0.069% to 0.035%, almost by half when II changes from 5 to 10.

D. Results on Relatively Large Models

Though not the focus of this work, in this section, we highlight that the proposed method can also be applied to any model, whether large or small, in general. A few state-of-the-art image classification models pretrained in PyTorch [50], viz. squeezenet1_0 (1.2 million weights), efficientnet_b0 (5.3 million weights), googleNet (6.6 million weights), shufflenet_v2_x0_5 (1.4 million weights), and mobilenet_v3_small (2.5 million weights) were passed through the proposed exponent sharing method (Algorithm 1). Consequently, this resulted in a reduction in memory requirement for weights by about 10%. Using a single-precision floating point, the storage required by weights became: squeezenet1_0 (4.8 MB), efficientnet_b0 (21.2 MB), googleNet (26.4 MB), shufflenet_v2_x0_5 (5.6 MB), and mobilenet_v3_small (10 MB). At 10% savings in memory, the memory requirement reduces by approximately 0.5–2 MB in these models.

We show how our method can be applied as a *follow on* method on other existing model compression methods using the example of weight pruning. The models in Fig. 11 were

TABLE V
IMPACT OF PIPELINE INITIATION INTERVAL IN FPGA ON CLOCK CYCLES IN TTT-YOLO

Initiation Interval	Layer2			Layer 5			Layer 7		
	Before	After	%Increase	Before	After	% Increase	Before	After	% Increase
1 to 5	103936001	104007681	0.069	23027201	23036161	0.039	11243751	11248126	0.039
6	124508161	124579841	0.058	27605761	27614721	0.032	13479376	13483751	0.032
7	145080321	145152001	0.049	32184321	32193281	0.028	15715001	15719376	0.028
8	165652481	165724161	0.043	36762881	36771841	0.024	17950626	17955001	0.024
9	186224641	186296321	0.038	41341441	41350401	0.022	20186251	20190626	0.022
10	206796801	206868481	0.035	45920001	45928961	0.02	22421876	22426251	0.02

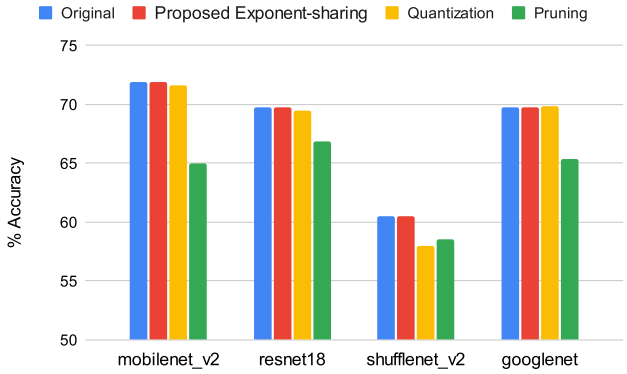

Fig. 12. Impact on accuracy after exponent-sharing, quantization, and pruning. The percentage memory savings using exponent sharing or pruning are 19.13%, 18.74%, 19.16%, and 18.8% for mobilenetv2, resnet18, shufflenetv2, and googlenet, respectively. Quantization saves 50% of memory when the model is converted to INT8.

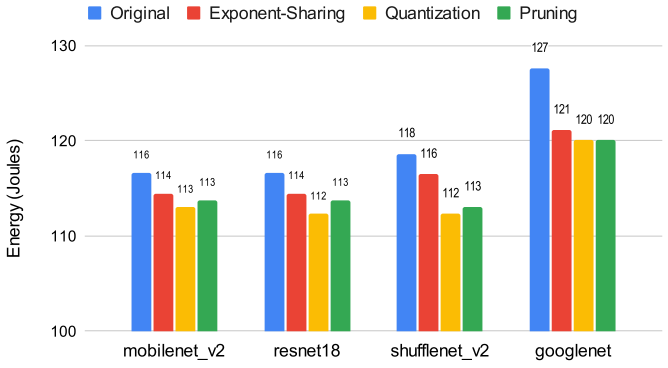

Fig. 13. Energy required for processing a batch of ten images after using different model compression methods, for mobilenetv2, resnet18, shufflenetv2, and googlenet models, using the codecarbon analysis tool [51].

trained and then pruned in PyTorch by setting different pruning levels. The pruning levels (10%, etc.) result in commensurate memory savings (i.e., 10%, etc.). The pruned models were then processed by Algorithm 1. We observe the added advantage of exponent sharing for different levels of pruning in all models shown in Fig. 11. With Float32 precision, application of Algorithm 1 resulted in an *additional* 12% memory savings, on average, with about 15% savings for all models for pruning between 10% and 50%. There is no further loss in accuracy except that incurred by the pruning technique itself.

E. Comparison With Other Model Compression Techniques

To compare our method with quantization and pruning, we first convert the models to the BFloat16 format using PyTorch. We prune models in such a way that the memory saved is equal to the memory saved by exponent sharing. This is done at the cost of some accuracy loss, whereas our method does not compromise the accuracy. On the other hand, although quantization saves 50% of memory in all models, it also loses some accuracy. We show the results on four popular models (mobilenetv2, resnet18, shufflenetv2, and googlenet) in Fig. 12. To understand the impact of exponent sharing on energy consumption we used CodeCarbon [51]. In Fig. 13, we report the energy consumed while processing a batch of ten images by four different models compressed using different methods. Our proposed approach results in comparable energy consumption but without any accuracy loss which is separately shown in Fig. 12.

V. CONCLUSION

The best method for model compression depends on the desired accuracy. The method proposed here compresses models without impacting their accuracy. It does not have any overhead of fine-tuning postmodel compression. The parallel reads, possible in FPGA implementations, bridge the gap in execution time before and after exponent sharing. The code of the GEMM with our implementations is available on github.[1] Our future work involves (1) improving memory savings with exponent approximations during the training phase itself and (2) an extended study of the proposed method for ultralow power ML applications on Jetson Nano.

REFERENCES

[1] M. S. Hasan, "An application of pre-trained CNN for image classification," in *Proc. 20th Int. Conf. Comput. Inf. Technol. (ICCIT)*, Dec. 2017, pp. 1–6.

[2] Y. Liu, H. Li, J. Yan, F. Wei, X. Wang, and X. Tang, "Recurrent scale approximation for object detection in CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 571–579.

[3] M. Zeng et al., "Convolutional neural networks for human activity recognition using mobile sensors," in *Proc. 6th Int. Conf. Mobile Comput., Appl. Services*, Nov. 2014, pp. 197–205.

[4] P. Khatamino, I. Canturk, and L. Özyilmaz, "A deep learning-CNN based system for medical diagnosis: An application on Parkinson's disease handwriting drawings," in *Proc. 6th Int. Conf. Control Eng. Inf. Technol. (CEIT)*, Oct. 2018, pp. 1–6.

[5] C. Chen, J. Huang, C. Pan, and X. Yuan, "Military image scene recognition based on CNN and semantic information," in *Proc. 3rd Int. Conf. Mech., Control Comput. Eng. (ICMCCE)*, Sep. 2018, pp. 573–577.

[6] A. Barbosa, T. Marinho, N. Martin, and N. Hovakimyan, "Multi-stream CNN for spatial resource allocation: A crop management application," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2020, pp. 258–266.

[1] https://github.com/prachikashikar/Expo-Share-In-GEMM

[7] W. R. Johnson, J. Alderson, D. Lloyd, and A. Mian, "Predicting athlete ground reaction forces and moments from spatio-temporal driven CNN models," *IEEE Trans. Biomed. Eng.*, vol. 66, no. 3, pp. 689–694, Mar. 2019.

[8] K. Kwon, A. Amid, A. Gholami, B. Wu, K. Asanovic, and K. Keutzer, "Invited: Co-design of deep neural nets and neural net accelerators for embedded vision applications," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6.

[9] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: Current progress, research challenges, and future roadmap," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 1303–1306, doi: 10.1109/DAC18074.2021.9586232.

[10] X. Wei, H. Chen, W. Liu, and Y. Xie, "Mixed-precision quantization for CNN-based remote sensing scene classification," *IEEE Geosci. Remote Sens. Lett.*, vol. 18, no. 10, pp. 1721–1725, Oct. 2021.

[11] R. Yu et al., "NISP: Pruning networks using neuron importance score propagation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 9194–9203.

[12] C. R. Banbury et al., "Benchmarking TinyML systems: Challenges and direction," 2020, *arXiv:2003.04821*.

[13] (2021). *FastML Team*. [Online]. Available: https://github.com/fastmachinelearning/hls4ml

[14] A. J. Redfern, L. Zhu, and M. K. Newquist, "BCNN: A binary CNN with all matrix OPS quantized to 1 bit precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2021, pp. 4599–4607.

[15] W. Nogami, T. Ikegami, S.-I. O'uchi, R. Takano, and T. Kudoh, "Optimizing weight value quantization for CNN inference," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2019, pp. 1–8.

[16] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of ConvNets via differentiable neural architecture search," 2018, *arXiv:1812.00090*.

[17] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless CNNs with low-precision weights," 2017, *arXiv:1702.03044*.

[18] S. I. Young, W. Zhe, D. Taubman, and B. Girod, "Transform quantization for CNN compression," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 9, pp. 5700–5714, Sep. 2022.

[19] B. Kang, A. Lu, Y. Long, D. Kim, S. Yu, and S. Mukhopadhyay, "Genetic algorithm-based energy-aware CNN quantization for processing-in-memory architecture," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 11, no. 4, pp. 649–662, Dec. 2021.

[20] G. Shomron, F. Gabbay, S. Kurzum, and U. Weiser, "Post-training sparsity-aware quantization," 2021, *arXiv:2105.11010*.

[21] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 8604–8612.

[22] V. Verma, P. Singh, V. Namboodri, and P. Rai, "A 'network pruning network' approach to deep model compression," in *Proc. IEEE/CVF Winter Conf. Appl. Comput. Vis. (WACV)*, Mar. 2020, pp. 3009–3018.

[23] J.-H. Luo, H. Zhang, H.-Y. Zhou, C.-W. Xie, J. Wu, and W. Lin, "ThiNet: Pruning CNN filters for a thinner net," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 10, pp. 2525–2538, Oct. 2019.

[24] Y. Lin, Y. Tu, and Z. Dou, "An improved neural network pruning technology for automatic modulation classification in edge devices," *IEEE Trans. Veh. Technol.*, vol. 69, no. 5, pp. 5703–5706, May 2020.

[25] N. Aghli and E. Ribeiro, "Combining weight pruning and knowledge distillation for CNN compression," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2021, pp. 3185–3192.

[26] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 6071–6079.

[27] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian, "Variational convolutional neural network pruning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2775–2784.

[28] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 1398–1406.

[29] W. Ahmed, A. Zunino, P. Morerio, and V. Murino, "Compact CNN structure learning by knowledge distillation," in *Proc. 25th Int. Conf. Pattern Recognit. (ICPR)*, Jan. 2021, pp. 6554–6561.

[30] C. Blakeney, X. Li, Y. Yan, and Z. Zong, "Parallel blockwise knowledge distillation for deep neural network compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1765–1776, Jul. 2021.

[31] S. Lin, R. Ji, C. Chen, D. Tao, and J. Luo, "Holistic CNN compression via low-rank decomposition with knowledge transfer," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 12, pp. 2889–2905, Dec. 2019.

[32] Y. Wei, X. Pan, H. Qin, W. Ouyang, and J. Yan, "Quantization mimic: Towards very tiny CNN for object detection," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, Sep. 2018, pp. 267–283.

[33] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proc. 16th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2018, pp. 389–400.

[34] Q. Zhang et al., "Efficient deep learning inference based on model compression," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2018, pp. 1695–1702.

[35] D. Hu and B. Krishnamachari, "Fast and accurate streaming CNN inference via communication compression on the edge," in *Proc. IEEE/ACM 5th Int. Conf. Internet-Things Design Implement. (IoTDI)*, Apr. 2020, pp. 157–163.

[36] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance FPGA-based CNN accelerator with block-floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1874–1885, Aug. 2019.

[37] M. Navardi, E. Humes, and T. Mohsenin, "E2EdgeAI: Energy-efficient edge computing for deployment of vision-based DNNs on autonomous tiny drones," in *Proc. IEEE/ACM 7th Symp. Edge Comput. (SEC)*, Dec. 2022, pp. 504–509, doi: 10.1109/SEC54971.2022.00077.

[38] L. Lamberti et al., "Tiny-PULP-dronets: Squeezing neural networks for faster and lighter inference on multi-tasking autonomous nano-drones," in *Proc. IEEE 4th Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Jun. 2022, pp. 287–290, doi: 10.1109/AICAS54282.2022.9869931.

[39] M. Navardi, A. Shiri, E. Humes, N. R. Waytowich, and T. Mohsenin, "An optimization framework for efficient vision-based autonomous drone navigation," in *Proc. IEEE 4th Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Jun. 2022, pp. 304–307, doi: 10.1109/AICAS54282.2022.9869975.

[40] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "A 64-mW DNN-based visual navigation engine for autonomous nano-drones," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8357–8371, Oct. 2019, doi: 10.1109/JIOT.2019.2917066.

[41] W. Kahan, "IEEE standard 754 for binary floating-point arithmetic," *Lect. Notes Status IEEE*, vol. 754, no. 1776, p. 11, May 1996.

[42] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Jun. 2019, pp. 69–76.

[43] K. Nan, S. Liu, J. Du, and H. Liu, "Deep model compression for mobile platforms: A survey," *Tsinghua Sci. Technol.*, vol. 24, no. 6, pp. 677–693, Dec. 2019.

[44] W. J. MacLean, "An evaluation of the suitability of FPGAs for embedded vision systems," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Sep. 2005, p. 131.

[45] *Tiny-Tiny-Tiny YOLO*. Accessed: Feb. 18, 2023. [Online]. Available: https://github.com/k5iogura/darknetttt, last

[46] J. Liao, L. Cai, Y. Xu, and M. He, "Design of accelerator for MobileNet convolutional neural network based on FPGA," in *Proc. IEEE 4th Adv. Inf. Technol., Electron. Autom. Control Conf. (IAEAC)*, vol. 1, Dec. 2019, pp. 1392–1396, doi: 10.1109/IAEAC47372.2019.8997842.

[47] J. Thiyagalingam, O. Beckmann, and P. Kelly, "An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays," in *Proc. 19th Annu. UK Perform. Eng. Workshop*, 2003, pp. 340–351.

[48] CEA LIST. *N2D2*. Accessed: Feb. 28, 2023. [Online]. Available: https://github.com/CEA-LIST/N2D2

[49] P. Kashikar, S. Sinha, and A. K. Verma, "Exploiting weight statistics for compressed neural network implementation on hardware," in *Proc. IEEE 3rd Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Jun. 2021, pp. 1–4.

[50] *Pytorch Vision Models*. Accessed: 28, Feb. 2023. [Online]. Available: https://pytorch.org/vision/stable/models.html

[51] V. Schmidt et al., "CodeCarbon: Estimate and track carbon emissions from machine learning computing," *Cited*, p. 20, May 2021. [Online]. Available: https://codecarbon.io/

Prachi Kashikar received the B.E. degree in computer science and engineering from the M.G.M. College of Engineering, Nanded, India, in 2011, and the M.Tech. degree in information technology from the S.G.G.S. College of Engineering and Technology, Nanded, in 2013. She is currently pursuing the Ph.D. degree with the Indian Institute of Technology Goa (IIT Goa), Goa, India.

Her research interests are mainly in computer architecture and Edge AI.

Mrs. Kashikar has received VLSID 2022 and 2023, GHC 2021, and Mobisys 2021 student fellowships. She is an HLF 2022 Young Researcher.

Olivier Sentieys (Member, IEEE) is currently a Professor with the University of Rennes, Rennes, France, holding an INRIA Research Chair on Energy-Efficient Computing Systems. He is leading the Cairn Team common to INRIA (French research institute dedicated to computational sciences) and the IRISA Laboratory, Rennes, where he is also the Head of the Computer Architecture Department, IRISA. His research interests are in the area of computer architectures, embedded systems, and signal processing, with a focus on system-level design, energy efficiency, reconfigurable systems, hardware acceleration, approximate computing, and power management of energy harvesting sensor networks.

Sharad Sinha (Senior Member, IEEE) received the B.Tech. degree in electronics and communication engineering from CUSAT, Kochi, India, in 2007, and the Ph.D. degree in computer engineering from NTU, Singapore, in 2014.

He is an Associate Professor of Computer Science and Engineering with the Indian Institute of Technology Goa (IIT Goa) Goa, India. His research and teaching interests are in computer architecture, embedded systems, and reconfigurable computing.

Dr. Sinha has received the Best Paper Awards at ICCAD 2022, ICCAD 2017, and IEEE INDICON 2022, and the Best Paper Award nomination at CASES 2018 and FCCM 2019.