4

Inside the AI Accelerators: From High Performance to Energy Efficiency

Ana Pinzari, Adrien Prost-Boucle, Christelle Rabache, and Frédéric Pétrot

Institute of Engineering Univ. Grenoble Alpes, France

Abstract

This chapter overviews current technologies for high-performance, low-power neural networks. To cope with the high computational and storage resources, hardware optimisation techniques are proposed: Deep Learning (DL) compilers and frameworks, DL hardware coupled with hardware-specific code generators. More specifically, we explore the quantization mechanism in deep learning, based on a deep-CNN classification model. We highlight the accuracy of quantized models and explore their efficiency on a variety of hardware platforms. Through experiments, we show the performance achieved using general-purpose hardware (CPU and GPU) and a custom ASIC (TPU), as well as the simulated performance for a reduced bit-width representation of 4 bits, 2 bits (ternary) down to 1-bit heterogeneous quantization (FPGA).

Keywords: Deep Learning, hardware accelerators, DL Compilers, CPU, TPU, GPU, quantization aware training, binary neural network.

4.1 Introduction and Background

AI-based solutions are constantly emerging in our daily life. AI solutions already dominate across all social fields; their remarkable success

bringing comfort and quality, and saving time. However, the difficulty of deploying these solutions raises open questions for both industry and research communities.

The use of the most recent neural networks generally requires a lot of computation and resources, as the rule of thumb is - the deeper the model, the more accurate it is. Various DL frameworks such as TensorFlow, MXNet and PyTorch are meant to simplify the definition and implementation of neural network architectures. To accelerate the performance of these models and achieve high energy efficiency, various DL hardware are proposed. CPU and GPU are general-purpose hardware embracing SIMD and vector-oriented logical components which can be used to facilitate and accelerate neural networks computation.

Application-specific integrated circuits, such as the custom dedicated hardware Google Coral TPU and FPGA, are designed to increase neural network performance and leverage the energy efficiency. Each hardware architecture has its own specificity in term of computational requirements and memory complexity. To cope with these requirements and to adapt the DL models to the wide variety of DL chips, DL compilers have been proposed.

The most recent DL compilers, such as TVM, Glow, XLA, Tensor Comprehension [6] have the objective of optimizing the NN for specific hardware architectures. They include in their flow a front-end intermediate representation (IR) and dedicated back ends, which allows the portability of a model across diverse target hardware.

To enable and facilitate the portability to AI edge devices, various optimization techniques must be applied. The most known methods involve reducing the parameter count and representational precision, while others use tensor decomposition techniques.

The number of parameters can be reduced by pruning the weights and nodes, or to lighten the topology of the neural network architecture. To cope with the memory complexity and to leverage hardware requirements, models need to be represented in lower precision, such as 8-bit integer representations or extremely low-bit precisions (ternary {-1, 0, 1}, binary {-1, 1}). This is referred to as quantization.

In this paper, we propose to show the implementation of a small neural network defined and designed to be deployed on a wide range of small edge-AI devices. To evaluate these edge platforms, we implemented an end-to-end inference design based on a quantized neural network architecture.

These experiments aim at demonstrating that an AI-based classification solution is feasible on these types of low-power and limited resource devices,

by only applying quantization techniques. Other optimizations are of course feasible, and their efficiency is studied in Section 1.2.

For the rest of the article, we show the performance our model achieves for real-time inference on CPU, GPU, TPU and FPGA boards. We are specifically interested to compare the power consumption and the logical and physical resources allocated for these edge devices. These criteria and the model's performance will be examined in our study.

4.2 Related Work

To enable rapid deployment and exploit the performance of hardware accelerators, a great time and effort has been dedicated to DL compilers. A recent overview of these compilers to enable the automatic transformation of DNN to hardware accelerators is well explained in [7].

For specialized DL accelerators, a hardware programmable architecture integrating JIT compiler and runtime, is proposed to the community [4]. The VTA is part of Apache TVM and offers more flexibility and versatility for diverse models to hardware back ends (FPGAs).

A comparison of various type of neural networks (MLPs, CNNs, RNNs) on Google TPU ASIC is done in [5]. Experiments show that the performance is limited by memory bandwidth rather than by peak computational need. This is due to the use of systolic execution (a row matrix is limited to 256-element multiply-accumulate operations) in order to save energy (reading large SRAM uses much more power than arithmetic operations).

Tensor decomposition is another acceleration method. A well-explained study of higher-order tensor decompositions and their applications is reviewed in [3][6]. The authors [2] propose an asymmetric 3D decomposition for different models. In their study, they show that shallower models can achieve 3.5x speed-up on the CPU and 3.3x speed-up on the GPU, with an insignificant loss of accuracy. Experiments on much deeper models, such as the VGG-16, showed that the GPU remains more sensitive to speed-up than the CPU. This gap is explained by the fact that for particular kernels used in tensor decomposition (e.g., 1x3, 3x1 convolutions), there is a lack of parallelism and therefore optimization in CPUs. This problem has boosted the research of many scientists, for example the authors [8] propose a CT decomposition that is up to 5.56x faster than the current Tensor Lab library.

The work of [1] explains in detail the efficiency of using QKeras library for ultra-low-latency inference. The authors use the hls4ml library for a fully automated deployment of quantized model on FPGA and show that the

amount of resource consumption can be reduced by up to 98%. Among various optimizations techniques, such as pruning and 6-bit precision for weights and activations, the best energy efficiency is achieved by the heterogeneous quantization method (be it post-training or quantization aware training).

The first authors to explore the training of neural networks with binary activations were introduced in [20]. An efficient way to map a binary CNN to reconfigurable logic is presented in [21]. Authors use FINN [22] framework to build a scalable and fast binary neural network, achieving a high throughput but a limited accuracy.

In the vast field of hardware accelerators, quantization techniques and models with limited number of weights are our primary research pillars. We are studying how heterogeneous quantization can be applied to achieve fairly high-performance with under 8-bit precision models, as some applications show [23]. In comparison, we do not neglect models with 8-bit integer quantization and show their performance on the most popular AI-edge boards. Indeed, the smallest items that CPUs manipulate is a byte, and there is no point in using smaller bit widths, as they require more instructions to process, and it is even counter-productive from a computation point of view.

4.3 Classification Model

The model we consider for our experiments has been developed for a multi-class classification problem.

To reduce the cost and energy consumption of the inference process as much as possible, we have considered the right balance between resources and accuracy, as a prior criterion. We performed the search for the appropriate network architecture using floating-point representation, keeping in mind that parameter size will be reduced by quantization. The definition of our model is mostly empirical, as the current pre-defined neural networks are mainly intended for very complex problems, and these large models are simply not appropriate for inference on small electronic devices. More details about our particular defined model can be found in [9].

Our neural network has been trained on mono-channel 224×224 images applying as learning method the supervised learning algorithm. Table 4.1 shows an overall description of each layer of the model, the number of parameters and the output size for the resultant feature maps.

We continue with optimization techniques regarding computational and memory requirements necessary to enable the execution of our model on small edge devices.

Table 4.1 Neural Network Description

Layer	Output size / Nr of Parameters
Input (224×224×1)	
Conv2D, 32 (7×7), s=2	109×109×32 / 1600
MaxPool2D (2×2)	54×54×32
Inception Block	54×54×32 / 1056
32 (1×1),	54×54×8 / 264, 54×54×8 / 264,
8 (1×1), 8 (1×1), MaxPool2D (3×3)	54×54×32
32 (3×3), 32 (5×5),	54×54×32 / 2336, 54×54×32 / 6432
32 (1×1)	54×54×32 / 1056
	54×54×128 / 11408
MaxPool2D (2×2)	27×27×128
Conv2D, 12 (1×1)	27×27×12 / 1548
Conv2D, 116 (3×3), s=2	14×14×116 / 12644
Conv2D, 116 (3×3), s=2	7×7×116 / 121220
Flatten	(5684)
FC / Softmax, 58	58 neurons / 329730
Total number of parameters: 478.150 (478 neurons and 477.672 weights)	
Total number of FLOPs: 125.518.940	

4.4 Quantization

Quantization consists of reducing the number of bits necessary to represent a value. Its use in neural networks is not new [12, 13] but using it on deep convolutional neural network raises new challenges. There are now many different quantization approaches, ranging from quantizing only the parameters, quantizing both parameters (often only weights, not biases) and activations, quantizing on 16, 8, or even 2 or 1 bit. Approaches using the smallest bit sizes are meaningful for hardware implementations [14, 15, 16, 17]. For comparison reasons, we performed experiments targeting off-the-shelf microcontroller-based boards using 8-bit quantization and custom hardware accelerators such as FPGA, for lower bit-width representations.

On micro-controllers, the most demanding part of the neuron output computation ($v_j = \sum_{i=0}^{n-1} x_i w_{ij}$) uses only 8-bit integer multiplications.

This is key because the area and power complexity of a multiplier is in $O\left(b^2\right)$ where b is the number of bits of the inputs. Each multiplication produces a $2b$-bit result, that is accumulated with the adder to produce a $(2b + \log_2 n\ )$–bit result, n being the number of inputs of the neuron. Using a 32-bit addition is a safe guess here, as there are very few chances that the accumulation takes place with more than 2^{16} inputs. It is also safe to have a bias b_j on 32-bit, as this is a single addition performed after all integer multiplications ($o_j = v_j + b_j$).

TensorFlow has been the first widely available framework to provide fine-tuned 8-bit integer arithmetic implementations for micro-controllers (using e.g. SIMD instructions) and Google TPU [18], we opted to use it given our high power-efficiency goal. We briefly summarize here the quantization approach that is advocated by and implemented in this framework, which is thoroughly detailed in [19]. For a given convolution layer, the quantization process produces, in addition, an offset (called zero-point, *zp*), and for each output channel of the layer a scale under the form of an integer multiplicand M and a shift s. The scale factor and offset must be applied before the activation function, leading (roughly, as the idea is to divide by 2^s which is not a raw shift for negative values) to $y_j = ((o_j \times M) \gg s) + zp$. These operations, done only once per kernel, typically fit in 32-bit, and the result is saturated to -128 or 127.

From a practical point of view, there are two main ways for quantizing a network: Post-training quantization (PTQ) and quantization-aware training (QAT). PTQ consists of finding offsets and scale values to approximate the weights of an already trained network. Post-training works quite well on large networks, especially when lowering weight size to 8 bits or more. To further reduce bit size without incurring high accuracy losses, it is usually necessary to use QAT. This consists of training the network by considering the low precision behaviour during the process.

Google's TensorFlow-Lite (TF-Lite) open-source framework provides an API to convert and interpret quantized networks. Given our target that is micro-controllers possibly backed by an accelerator, for which lower than 8-bit precision is useless, we use the PTQ method. It produces weights and biases quantized to a fixed-point precision of 8-bit using the approach mentioned above and required by integer-only accelerators. PTQ takes a fully trained model and doesn't require additional modifications for conversion into a quantized model. Nevertheless, an important point for the conversion process is to provide a representative data set, i.e., a small subset of the original data set which covers the entire value space. This gives the quantization process the range of inputs values and it can then find the most appropriate 8-bit fixed-point representation (multiplicand M and shift s) for each weight and activation value. To achieve the best possible performance, i.e., ensure that all computations are done using SIMD instructions or outsourced to the TPU, it is recommended to strictly stick to the 8-bit data type. For this purpose, we perform a full integer optimization with the TF-Lite converter, i.e., the inputs and the outputs use 8 bits too.

The accuracy with the quantization process activated is given Table 4.2.

Table 4.2 Inference Accuracy Of The Quantized Model Before (QAT) and After (PTQ) Training

	Quantization-aware Training	Post-training Quantization
Accuracy	97.63 %	97.35 %

4.5 Experiments and Results

The following experiments are conducted using software implementation of our quantized neural network model as well as the unquantized version. They are each using the available kernel implementation provided with the development kit without modification or optimization from our side.

Further optimization is described in Section 4.5.2, though we show through this type of experiment that solely optimizing the neural network model is enough to deliver the required performances using general purpose hardware.

Experiments are conducted on the following hardware targets.

- X86 Desktop CPU 48 cores / 96 threads (float and int)
- Google Coral TPU coprocessor 4 TOPS (int)
- Google Coral CPU quad Cortex-A53 and a Cortex-M4F (int and float)
- Jetson CPU (int and float) Quad Cortex-a53
- Jetson Maxwell GPU (float), 128 CUDA cores
- STM32MP1 CPU Cortex-A7 (int and float)
- Zynq-7000 SoC *XC7Z010* FPGA

Figure 4.1 describes the workflow to create a TensorFlow Lite model for inference on the above-mentioned edge devices. Our conversion focuses on creating a floating-point quantization model (for inference especially on GPU) and an 8-bit fixed point model for CPU and TPU acceleration. For optimal use of Coral's TPU, the tflite model must be compiled at the end with the edge-tpu compiler to check the compatibility of the quantized operations and then map them onto the TPU.

Once we have the models, we analyse the real-time performance of our model for different systems. The experiments target the number of inferences our model can perform per second, by measuring the latency for different scenarios: unquantized TensorFlow model (binary32 The

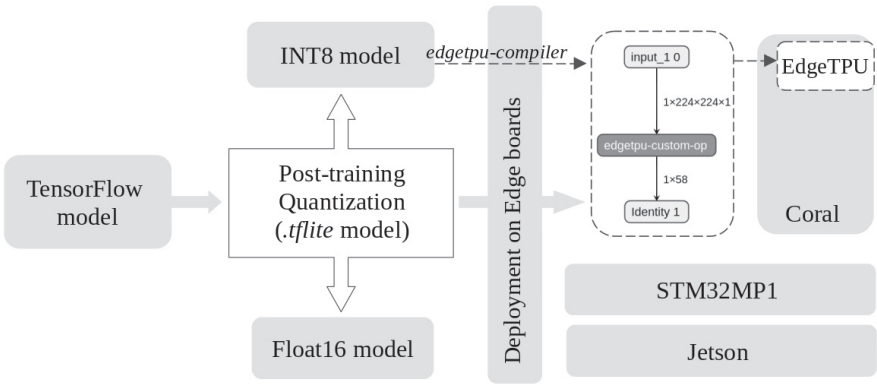

Figure 4.1 Workflow to Create a Tflite Model (Int8 And Binary16) for Inference on Edge Boards: Google Coral Including the Compiled Model for the EdgeTPU, STM32MP1 and Jetson.

*binary*256/128/64/32/16 types correspond to the floating-point representations defined in the IEEE 754-2008 standard on the number of bits indicated in their name.), tflite model (binary16 and int8) and edgetpu model (int8). Inference is performed one image at a time, i.e., the batch size is set to 1.

4.5.1 Time and power consumption

Table 4.3 shows the performance of our model for each target. An x86 CPU desktop machine uses binary32 floats by default to infer a neural network. With quantization, there is a gain in memory resources and therefore a higher inference speed, at the expense of a lower precision. The MP1 board performs faster for integer arithmetic, due to flexible dual cores dedicated for real-time low-power tasks. For the Coral SoC, the best performance is achieved by the TPU ML accelerator, the performance is more than 30x higher (902 i/s) than on its CPU. The Jetson GPU shows good inference performance for models at half precision. The binary16 operations are faster than the binary32 ones, so these quantized models should be considered for future evaluation.

In the following, we present a general analysis by taking higher throughputs and focusing not only on hardware optimizations but also on power consumption. The following experiments are performed on a batch size of 100 images and within the range of 1 to 32 batches processed at a time.

Table 4.3 Inference Performance and Latency Measurements for Randomly Selected Images. Experiments Done on x86 Standalone Server, Google Coral, STM32P1 and NVIDIA Jetson Boards.

	Performance (inferences/s)				
	Float	Float (tflite)	Int CPU	TPU	GPU
x86	52.5	322.5	312.5	-	-
Coral	-	20	31.8	902	-
MP1	-	4.5	5.5	-	-
Jetson	26	38	56	-	47
	Latency (ms)				
x86	19	3.1	3.2	-	-
Coral	-	49.4	31.4	1.11	-
MP1	-	223	181	-	-
Jetson	38.5	26.4	17.8	-	21.2
	Accuracy: 97 %				

4.5.1.1 Google Coral Board

Figure 4.2 shows the performance achieved by the TPU and the CPU of the Coral board. We can observe that for large batch sizes, the TPU hardware accelerator achieves performance up to 1600 inferences/s for a power consumption of 4.2 W. Running the tflite model on the CPU (ARM vector instructions), and without edge-tpu optimization, we obtain a performance of 33 inferences/s (ips) for the int8 model leading to a power consumption of 4.3 W, and a lower consumption of 3.8 W for the binary32 model, with 21 ips. In the power curves, we can observe a repetitive power overshoot of a bit less than 1 W per batch. This is due to the cooling fan that starts when using larger batches. Note that for inferences at a batch size of 1, the fan was never activated.

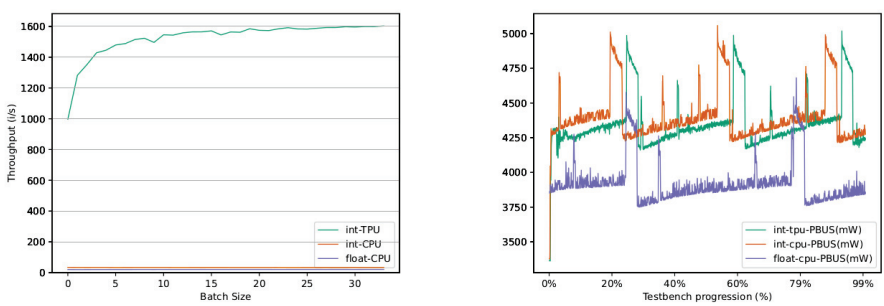

Figure 4.2 Coral Performance and Power Measurements

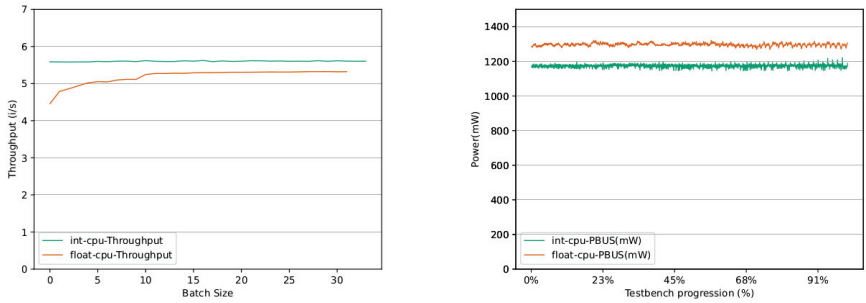

Figure 4.3 MP1 Performance and Power Measurements

4.5.1.2 STM32MP1 Board

The STM32 MP1 board is efficiently designed for low power mode. The float throughput improves when we increase the batch size, taking thus the advantages of the ARM SIMD instructions. For the integer model, there is not much improvement in performance, see Figure 4.3

We can also report that it was not possible to exceed a batch size of 32 with floats due to memory limitations. But we were able to go up to batches of 128 for 8-bit integers due to their much smaller memory footprint.

4.5.1.3 NVIDIA Jetson

Figure 4.4 shows GPU float experiments with two inference kernels. One available is the TensorFlow base interpreter, the other is the TensorFlow lite implementation. Both have similar throughput (a little lower for tflite) but there's a non-negligible change in power consumption going from 5W to 3.5W. The latter being close to integers which are even more interesting with a little more throughput for a little less power consumption. "The non-linearity

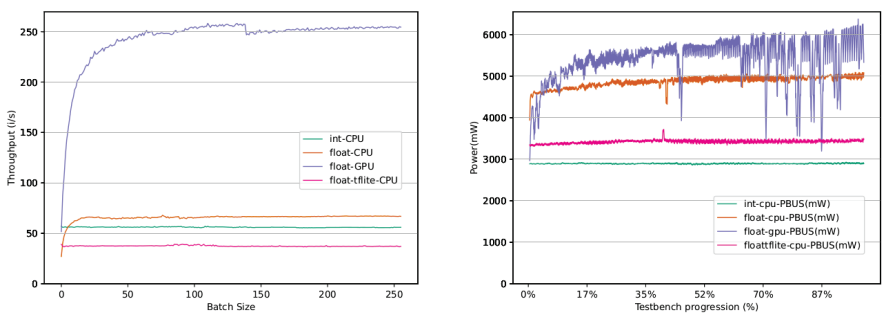

Figure 4.4 Jetson Performance and Power Measurements.

in the GPU curve occurs for a batch ...” a batch size of 128 which is the number of CUDA core to feed with images. This is why we lose some throughput at 129 before slowly catching up the maximum throughput.

4.5.2 FPGA

4.5.2.1 QKeras Library

QKeras [10, 11] is an extension to Keras, a high-level API to define and train neural networks. It has been implemented to perform a drop-in replacement for certain layers of the model, related to weights and activation functions with a deep quantized version of Keras neural network.

QKeras is designed to remain a simple and consistent interface optimized for common functionalities in accordance with Keras design principles. For this purpose, the following set of layers have been implemented: Qconv2D, QActivation, QDense etc, to enable the conversion between non-quantized to quantized networks. To make your own quantization (QAT) it is needed to replace all variables and weights/bias created by Keras as well as output of arithmetic layers by quantized functions. Qactivation is used in both convolution (Qconv2D) and dense (Qdense) layers and acts at the end as a merging function for activation and quantization. For these layers, some parameters are interesting to mention.

Alpha is a parameter concerning the scale factor and should be applied before the activation function. This parameter by default is *None*. It can also indicate that the scale is computed as a floating-point number by the learning process. It can also force the scale to be an integral power of 2, which ends up for a hardware implementation in shifting the result of a convolution or dense layer to the right or left (positive shifts left, negative shifts right). For these practical reasons, in our experiments we opt for the latter setting.

Symmetric if set to, if set to *True,* ensures the trainable parameters to get the same maximum and minimum values after the clipping operation during quantization. The use of *stochastic_rounding* reveals to be useful in practice for improving accuracy. However, computing stochastic rounding might be quite heavy, so we set this parameter to *False.*

Table 4.4 describes the results obtained by our model after quantizing for different precisions. For example, the first convolution *q_conv2d* is set this way: bits=4, integer=0, symmetric=1. The 4-bit quantization of the entire model (weights, biases, and activations) achieves the better accuracy. When further reducing to 2 bits, the accuracy of the model decreases drastically.

Table 4.4 QKeras quantization for different precisions

	Precision			**Sparsity**		
Layer	4 bits	2 bits	Heterogeneous (4 bits, binary)	4 bits	2 bits	Heterogeneous (4 bits, binary)
q_conv2d W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	(4,1,1), (4,1,1) (4,1)	0.1156	0.5131	0.1350
q_conv2d_1 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.1023	0.5341	-
q_conv2d_2 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.0985	0.5720	-
q_conv2d_3 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.1108	0.5483	-
q_conv2d_4 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.1973	0.5330	-
q_conv2d_5 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.2243	0.6550	-
q_conv2d_6 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.0975	0.5445	-
q_conv2d_7 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.1389	0.5975	-
q_conv2d_8 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.1754	0.5540	-
q_conv2d_9 W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	binary, (4,1,1) (4,1)	0.2935	0.6840	-
q_dense W,b ReLU	(4,0,1) (4,1)	(2,0,1) (2,0)	(4,1,1), (4,1,1) (4,1)	0.5152	0.8756	0.4806
	Model Performance			**Total Sparsity**		
Accuracy Loss	96 % 0.148	76.1 % 0.747	94.6 % 0.163	0.4397	0.81	0.3318

The table shows, as an additional information, the model sparsity for various quantization scenario. This is a valuable metric to compare and see the trade-off between the accuracy and the computational cost of the model. The weights sparsity plays the role in reducing the number of calculations during the inference. When the sparsity is the same, the level of FLOPs remains constant. When the sparsity is too important (2 bits precision), the quantization becomes less effective, and the accuracy of the model is reduced. The sparsity for the last fully connected layer is like the pruning technique, where synapses between neurons are reduced.

These reasons led us to search for heterogeneous quantization, the trick being to find the right trade-off between accuracy requirements and hardware performance.

From a practical point of view, for the weights of the intermediate layers, we opted for an extremely low-bit quantization (we used *binary* quantization). The first and last classification layers were quantized to 4 bits, as well as the biases of the entire network. The activations of each quantized layer play an equally important role, so these neurons have not decreased in number of bits, the precision is maintained 4 bits.

This last model was implemented on the small Zybo board. The precision for each quantized layer and the accuracy of the model are described Table 4 (heterogeneous quantization). This method enabled us to achieve an accuracy slightly lower than the performance of the 4-bit model, more precisely a rate of 94.6%.

4.5.2.2 Quantized model and Experimental Setup

The quantized network used in our experiment, targets the small board Zybo Z7010 and explores the advantages of low-bit quantization. The major advantage of binary precision is that the pre-trained weights of the model (1.9 MB) fit very well within the on-chip memory. To achieve high memory throughput and very lightweight control paths, our hardware implementation does not leverage weight sparsity or compression. The low resource usage of multipliers with binary weights also enable to use a larger bit width for activations (4 bits), keeping accuracy high. Each network layer is an independent hardware block with its own dedicated resources and implementation, which enables to optimize parallelism and memory usage on a per-layer basis. The entire network fits inside the FPGA.

The approach of this efficient neural network implementation is presented in [17]. The hardware architecture generated by this method presents a total of 40 layers, with the following type: Sliding Window Layer, Neuron Layer,

Table 4.5 FPGA performance and resource utilization

LUT (logic)	LUTRAM	Slice Registers	Block RAM	DSP cores
9030 / 17600	4830 / 6000	11796 / 35200	60 / 60	37 / 80
(51.3 %)	(80.50 %)	(33.51 %)	(100 %)	(46.25 %)

Table 4.6 Model perpormance on FPGA

Performance	Latency	Power (FPGA only)	Power (Entire Chip)
178 images/s	26 ms	0.24 W $\sim$ 134 mJ/ image	1.75 W $\sim$ 983 mJ/ image

ReLU Layer, MaxPooling Layer, and Fork and Cat layers for synchronization of the parallel branches in the *inception* part of the network.

The resource utilization and performance of our quantized network implementation, is described Table 4.5 .

The BRAM is used for read-only memories of weights in neuron layers. All quantized MAC operations (multiply-accumulate) in neuron layers are implemented in distributed logic with LUTs. The MAC operations of most layers have a 1 b operand, which reduce the multiplications to tiny AND operations. Only the last layer actually implements a 4 b multiplication (0.5% of all MACs). The DSP cores are only used for address calculations within Sliding Window Layers.

The power consumption number is the total power estimation performed by the Xilinx Vivado synthesis suite. The processor subsystem of the Zynq chip would actually be mostly idle, so we report power both for the whole chip and for the FPGA only. The performance at 150MHz is summarized in following Table 4.6.

4.6 Conclusion

A selection of edge-AI boards and some optimization techniques have enabled us to investigate the possibilities of achieving high performance on a low budget. With a deep CNN model defined for a classification task, the accuracy achieved on 8-bit operating systems is around 97%. The efficiency of each board depends on processing speed and RAM availability. In our experiments, we found that performance is more limited by memory usage than by the number of neurons. In addition, we show how performance

and energy efficiency can be affected by the cost of each board. To these measurements, further experiments using binary operations were carried out to address the option of a more energy efficiency at the expense of slightly degraded accuracy (94.6%). To find a suitable model, we used a hybrid aware quantization and described the methods enabling the maintain of an acceptable accuracy.

By focusing on this type of optimization related to the memory usage, i.e., an appropriate number of weights and limited bit widths, we have shown that high-performance inference can be achieved very efficiently. More specifically, the energy efficiency and power consumption achieved by each evaluation board is summarized as follows:

- Coral TPU: $3.12\ \mathrm{mJ/image}$ or $320\ \mathrm{images/s/W}$
- STM32MP1: $232\ \mathrm{mJ/image}$ or $4.7\ \mathrm{images/s/W}$
- Jetson GPU: $22.7\ \mathrm{mJ/image}$ or $44\ \mathrm{image/s/W}$
- Zybo Z-7010: $983\ \mathrm{mJ/image}$ or $101.7\ \mathrm{image/s/W}$

For further work, we plan to try out other optimization techniques linked to specific applications, for which these methodologies are of the utmost interest.

Acknowledgements

This work was supported by Key Digital Technologies Joint Undertaking (KDT JU) in EdgeAI "Edge AI Technologies for Optimised Performance Embedded Processing" project, grant agreement No 101097300.

References

[1] Claudionor N. Coelho Jr, Aki Kuusela, Shan Li, Hao Zhuang, Jennifer Ngadiuba, Thea Klacboe Aarrestad, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors", Nature Machine Intelligence (2021)

[2] Zhang, Xiangyu & Zou, Jianhua & He, Kaiming & Sun, Jian. (2015). Accelerating Very Deep Convolutional Networks for Classification and Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[3] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. SIAM Rev, 51, 3 (August 2009), 455-500.

[4] T. Moreau *et al.,* "A Hardware-Software Blueprint for Flexible Deep Learning Specialization," in *IEEE Micro*, vol. 39, no. 5, pp. 8-16, 1 Sept.Oct. 2019, doi: 10.1109/MM.2019.2928962.

[5] Norman P. Jouppi, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. SIGARCH Comput. Archit. News 45, 2 (May 2017), 1-12.

[6] Vasilache, Nicolas, et al. "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions.", 2018.

[7] R. Zhao *et al.,* "Hardware Compilation of Deep Neural Networks: An Overview," *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Milan, Italy, 2018, pp. 1-8, doi: 10.1109/ASAP.2018.8445088.

[8] Zhang, Tao *et al.* "cuTensor-Tubal: Efficient Primitives for Tubal-Rank Tensor Learning Operations on GPUs." IEEE Transactions on Parallel and Distributed Systems 31 (2020): 595-610.

[9] Pinzari, Ana et al. (2023). Power Optimized Wafer map Classification for Semiconductor Process Monitoring.

[10] Moons, Bert, et al. "Minimum energy quantized neural networks." *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017.

[11] Zhou, Shuchang, *et al.* "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients." *arXiv preprint arXiv:1606.06160* (2016).

[12] G. Dundar and K. Rose, "The effects of quantization on multilayer neural networks," in IEEE Transactions on Neural Networks, vol. 6, no. 6, pp. 1446-1451, Nov. 1995.

[13] B. G. Hoskins, M. R. Haskard and G. R. Curkowicz, "A VLSI implementation of multi-layer neural network with ternary activation functions and limited integer weights," Proceedings of International Conference on Microelectronics, Nis, Serbia, 1995, pp. 843-846 vol.2.

[14] R. Andri, L. Cavigelli, D. Rossi and L. Benini, "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights," 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Pittsburgh, PA, USA, 2016, pp. 236-241

[15] Umuroglu, Yaman & Fraser, Nicholas & Gambardella, Giulio & Blott, Michaela & Leong, Philip & Jahre, Magnus & Vissers, Kees. (2017). FINN: A Framework for Fast, Scalable Binarized Neural Network Inference.

[16] Ritchie Zhao, et al. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17). Association for Computing Machinery, New York, NY, USA.

[17] Adrien Prost-Boucle, Alban Bourge, and Frédéric Pétrot. 2018. High-Efficiency Convolutional Ternary Neural Networks with Custom Adder Trees and Weight Compression. ACM Trans. Reconfigurable Technol. Syst. 11, 3, Article 15 (September 2018).

[18] N. P. Jouppi, et al. "Ten Lessons From Three Generations Shaped Google's TPUv4i: Industrial Product," 2021 ACM/IEEE 48th Annual Intemational Symposium on Computer Architecture (ISCA), Valencia, Spain, 2021, pp. 1-14.

[19] Jacob, Benoit, et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference." Proceedings of the IEEE conference on computer vision and pattem recognition. 2018.

[20] Courbariaux, Matthieu, Bengio, Yoshua, et David, Jean-Pierre. Binaryconnect: Training deep neural networks with binary weights during propagations. Advances in neural information processing systems, 2015 , vol. 28 .

[21] Fraser, Nicholas J., et al. "Sealing binarized neural networks on reconfigurable logic." Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms. 2017.

[22] Umuroglu, Yaman, et al. "Finn: A framework for fast, scalable binarized neural network inference." Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays. 2017.

[23] A. D. Vita, D. Pau, L. D. Benedetto, A. Rubino, F. PÃl'trot and G. D. Licciardo, "Low Power Tiny Binary Neural Network with improved accuracy in Human Recognition Systems," 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 2020, pp. 309-315.