AdequateDL: Approximating Deep Learning Accelerators

Olivier Sentieys, Silviu-Ioan Filip, David Briand, David Novo, Etienne Dupuis, Ian O'Connor, Alberto Bosio

AdequateDL: Approximating Deep Learning Accelerators*

Olivier Sentieys[1], Silviu Filip[1], David Briand[2], David Novo[3], Etienne Dupuis[4], Ian O'Connor[4], and Alberto Bosio[4]
[1]*INRIA, Rennes, France*
[2]*CEA LIST, France*
[3]*LIRMM, Université de Montpellier, CNRS, France*
[4]*Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, France*

***Abstract*—The design and implementation of Convolutional Neural Networks (CNNs) for deep learning (DL) is currently receiving a lot of attention from both industrials and academics. However, the computational workload involved with CNNs is often out of reach for low power embedded devices and is still very costly when running on datacenters. By relaxing the need for fully precise operations, approximate computing substantially improves performance and energy efficiency. Deep learning is very relevant in this context, since playing with the accuracy to reach adequate computations will significantly enhance performance, while keeping quality of results in a user-constrained range. AdequateDL is a project aiming to explore how approximations can improve performance and energy efficiency of hardware accelerators in DL applications. This paper presents the main concepts and techniques related to approximation of CNNs and preliminary results obtained in the AdequateDL framework.**

***Index Terms*—Deep Learning, CNN, Approximate Computing**

I. Introduction

Deep Learning [1] models, and in particular CNNs, are currently one of the most intensively and widely used predictive approaches in the field of machine learning. CNNs have shown to give very good results for many complex tasks such as object recognition in images/videos, drug discovery, natural language processing, autonomous driving, and playing complex games [2]–[5].

In spite of these benefits, the computational workload involved in CNNs is often out of reach for low-power embedded devices, and/or is still very costly when running on datacenter-style Component-Off-The-Shell (COTS) hardware platforms. To give an example, the amazing performance of AlphaGo [5] required 4 to 6 weeks of training executed on 2000 CPUs and 250 GPUs for a total of about 600kW of power consumption (while the human brain of a go player requires about 20W). Thus, a lot of research effort from both industrials and academics has gone into defining/designing custom hardware platforms supporting this type of algorithms, with the goal of improving performance and/or energy efficiency [6]–[8].

In recent years, Approximate Computing (AxC) has become a major field of research to improve both speed and energy consumption in embedded and high-performance systems [9]. By relaxing the need for fully precise or completely deterministic operations, AxC substantially improves energy efficiency.

CNNs show inherent resilience to insignificant errors due to their iterative nature and learning process. Therefore, an intrinsic tolerance to inexact computation is evidenced, and using the approximate computing paradigm to improve power and delay characteristics is relevant [10]. Indeed, CNNs mesh well with AxC techniques, especially with fixed-point arithmetic or low-precision floating-point implementations (it was also demonstrated that even binary or ternary weights could be used), which moreover exposes large fine-grain parallelism. They are therefore ideally suited for hardware acceleration using FPGA and/or ASIC implementations, as acknowledged by the large body of work on this topic. Although accelerators have demonstrated significant performance/energy gains compared to GPU/CPU implementation, they are still not sufficient to address future performance requirements [11].

The goal of this paper is to explore how approximation techniques can improve the performance of hardware accelerators for CNN inference in DL applications. In particular, weight-sharing techniques will be presented and discussed through preliminary experimental results. The paper is structured as follows. Section II provides the basics about approximate computing techniques for CNN inference. Section III presents the open source framework we use for CNN design, while Section IV details the preliminary results. Section VI concludes the paper.

II. AxC Techniques for CNN inference acceleration

Specific industrial constraints are related to the optimization of CNN execution time and energy consumption. One relevant technique complementary to algorithmic optimization is to play with the precision (*i.e.,* bit-width) and number representation of the data. For example, neural networks are often trained using 32-bit floating-point arithmetic. However, for the inference phase, the precision of these parameters can be greatly reduced, enabling huge performance gains on reduced precision hardware architectures, such as those provided by FPGAs, recent NVidia GPUs, or custom computing architectures such as CEA PNeuro [12]. The issue is that for large networks, this precision reduction is far from trivial. There exist several methods that are already implemented in existing frameworks and libraries that help with this.

We consider some examples. This first one is the use 8-bit inference as integrated into the Nvidia TensorRT library [13]. The TensorRT builder must perform a process called calibration to determine how best to represent the weights and activations as 8-bit integers. During this step, a heuristic will determine a suitable 8-bit scaling factor for each layer so as to cover as best as possible the range of required values. A second example is to use TensorFlow

*This work has been funded by the French National Research Agency (ANR) through the AdequatedDL research project (ANR-18-CE23-0012).

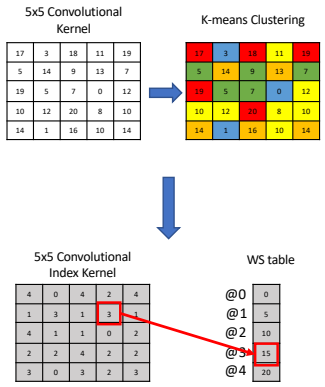

Fig. 1. WS technique applied to a 5×5 convolutional kernel.

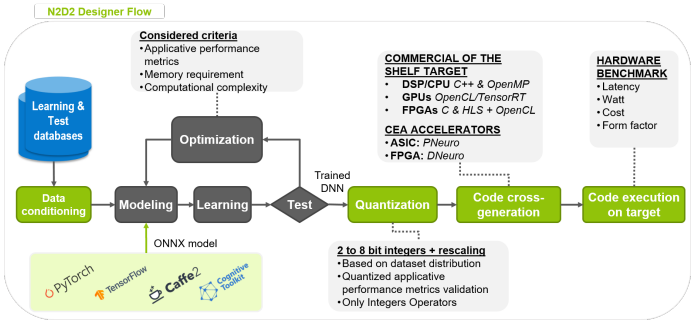

Fig. 2. System design and optimization process using N2D2.

to retrain an already designed network to work with 8-bit inference. While working on the network parameters, it also quantizes layer input/outputs. It does so by analyzing the extremal $(\min, \max)$ values that can pass through the network in order to choose suitable scale and range parameters for the quantization format.

The third option is Ristretto [14], a five step quantization tool: (1) analysis of weight dynamic range for fixed-point representation; (2) a calibration step in the forward path to analyze and determine statistical parameters of the layer activations; (3) binary search to find the number of bits for convolutional & fully connected weights, and layer outputs; (4) iteratively quantize each layer as a function of classification accuracy; (5) retrain the resulting fixed-point network. However, only a subpart of the network (inputs and weights) is quantized, whereas the rest remains in floating-point. This still leaves room for significant reductions in implementation cost.

A recent review of the main techniques to reduce precision is provided in [15], [16]. It concludes that, although much work has been done, CNNs remain an important area of research with many promising applications and opportunities for innovation at various levels of the hardware design process.

Another efficient technique to approximate CNN inference is the Weight-Sharing (WS) approach. Broadly, it identifies **clusters of weights** and aims at replacing all weights within a cluster by its centroid. To better understand this, consider Fig. 1, where starting from a 5×5 convolutional kernel matrix, clustering is applied. Clusters are usually identified with the K-means algorithm [17].

III. The N2D2 deep learning platform

Current research on Deep Neural Networks extensively uses different DL frameworks that implement various types of neural network layers and different learning algorithms. They also implement visualisation tools to ease neural network building and database handling. Among these frameworks, one can cite PyTorch [18] and Tensorflow [19]. Some other frameworks, such as Keras, are just interfaces to these frameworks. The goal of these aforementioned tools is to develop neural networks and machine learning algorithms to solve challenges, for example, related to machine vision. Unfortunately, once the network has been designed and trained, there is a gap towards its optimized deployment in an embedded and hardware constrained system. The recent development of libraries built to ease the deployment and optimization of deep neural networks, such as TensorRT [13], TFLite [19] or OpenVINO [20], reflects the rising trend of hardware integration requirements. The main weakness of these libraries is the limitation to certain target platforms and the possible optimizations that can be applied, which are greatly linked to proprietary solutions.

The N2D2 deep learning framework from CEA [21] reduces this gap by providing an innovative optimization method to the system designer. Leveraging on the CEA expertise related to embedded and high-performance computing, N2D2 is hardware agnostic while being able to directly target most common computing architectures and parallel run-time software.

Fig. 2 shows a high-level view of the system design process enabled by N2D2. The N2D2 framework integrates a generic database handling and data processing dataflow. Fig. 3 describes a typical example possible during the pre-processing step. Transformations can be applied to data, pixel-wise labels, and geometrical labels on a wide range of data types (images, sounds, etc.).

The N2D2 learning core is close to the standard DL frameworks with the support of typical layers, operators and learning rules. The N2D2 learning core execution on x86 and ARM processors is accelerated thanks to C++/OpenMP kernels, while execution on NVidia GPUs is supported thanks to cuDNN and custom CUDA Kernels. Moreover, the N2D2 core also supports spike simulations modelling. The input model representation can be given through an INI description file or in the Open Neural Network Exchange (ONNX) format [22]. The ONNX format enhances DL framework interoperability by unifying the representation model. One of the typical use cases of ONNX in N2D2 is to load a pre-trained neural network from another DL framework.

Among the key features of the N2D2 framework, the integrated quantization module remains one promising technique to optimize a DL model for a wide range of hardware accelerators. This module aims to produce a quantized network version from an already trained floating-point network. The post-training quantization method used relies on a dataset of representative data points to calculate an approximation of the range of the outputs for each network layer. The first step rescales weights to the $[-1, 1]$ range. In the same way, activations are rescaled inside the $[-1, 1]$ range for signed output and $[0, 1]$ for unsigned output. To achieve this goal, the N2D2 quantizer module finds the maximum absolute value of the outputs of the activation for each layer on the whole dataset. Optionally, the N2D2 quantizer can also find the maximum absolute value per output channel. Unfortunately, using the maximum absolute value as scaling factor in the quantized activations often results in a loss of application performance. The N2D2 quantizer addresses this issue by calculating the most appropriate threshold that mini-

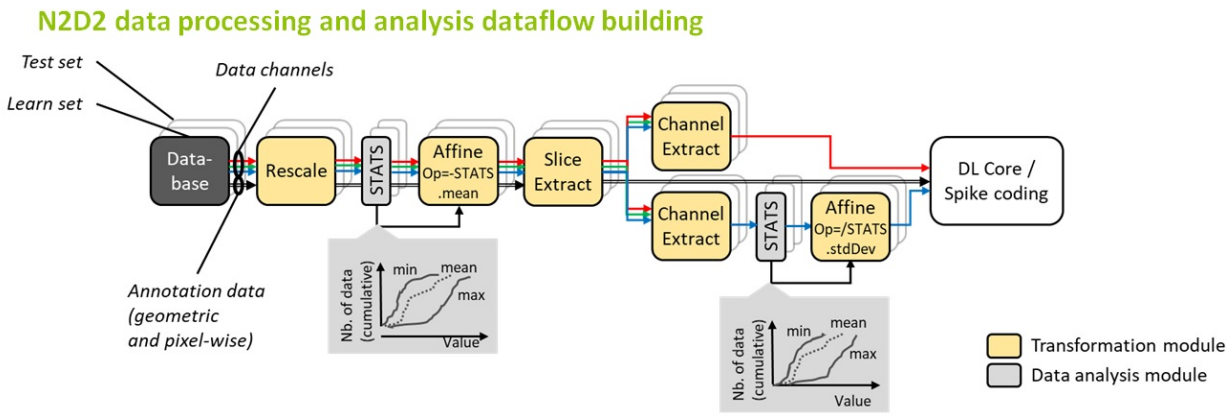

Fig. 3. Preprocessing dataflow with N2D2.

mizes the difference between the origin distribution and the clipped quantized distribution. It does this through an error estimator based on mean square error (MSE) or Kullback-Leibler divergence (KL-divergence). Weights and activations can be represented using 8-bits integers (INT8) without incurring significant loss in accuracy thanks to this method. Once the neural network performances meet the system designer requirements, the code generator of N2D2 can be used to generate optimized code targeting most common COTS using their respective run-time software.

IV. Exploring Weight-Sharing for CNN

We can express WS as an N variable optimization problem, with N the number of layers of the CNN [23]. Each k_i represents the number of shared values of a given layer taken from a set of values k_{range}, and let $k_{\text{tuple}} = \{k_i, i = 1, \ldots, N\}$. The full exploration space has size $\mathcal{O}(|k_{\text{range}}|^N)$. Each solution has to be evaluated in terms of *Accuracy Loss* (AL) and *Compression Ratio* (CR). The objective is to maximize CR for a given AL constraint.

For example, given a small CNN like LeNet [24], composed of 5 layers, with a $k_{\text{range}} = [1, 256]$ allows index values to be encoded using up to 8 bits. The sequential exhaustive exploration of all k_{tuple} combinations will take $256^5 \approx 1.1 \times 10^{12}$ seconds (more than 3000 decades) assuming the clustering (*i.e.,* the selection of shared weights) and the scoring (*i.e.,* CNN AL evaluation) takes 1 second. The evaluation complexity becomes exponentially worse for modern CNNs, which include significantly more layers (*e.g.,* ResNet-152 [25] includes 152 layers). Thus, we need a better method to find the optimal k_{tuple}.

In this work, we split the optimization problem into two orthogonal sub-problems solved in sequence. The first one, **layer optimization**, is to select Pareto efficient k_i local to each layer of the network and the second one, **network optimization**, is to find Pareto efficient combinations k_{tuple} of these selected k_i.

A. Layer optimization

To find the layer Pareto efficient k_i, we perform a layer approximation sensitivity analysis by changing the number of shared values and studying the resulting AL and CR. AL is obtained by scoring the CNN using the test set, while CR is obtained as the ratio between the number of shared values over the number of initial values. An example is Fig. 4, obtained by varying k_i inside $k_{\text{range}} = [1, 256]$ in the third layer of LeNet. The weight sharing AL

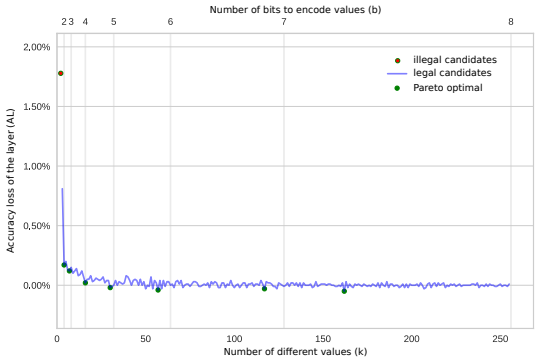

Fig. 4. Studying layer sensitivity to compression by varying the number of shared values.

highly depends on the number of clusters. In order to evaluate the CR of the layer, we rely on the number of bits needed to encode the values $b_{\text{index}} = \lceil \log_2(k_i) \rceil$. We then select Pareto efficient k_i by taking into account the AL and the b_{index}. We thus obtain a restrained set of Pareto efficient approximated layer versions ϕ_i with $|\phi_i| \leq \lceil \log_2(\max(k_{\text{range}}) \rceil$ ($|\phi_i| \leq 8$ in our example). The runtime complexity of this first sub-problem is $\mathcal{O}(N|k_{\text{range}}|)$.

B. Network optimization

The second sub-problem is to combine the k_i together in a k_{tuple} to obtain an optimal approximated CNN. The issue is the still large number of possible combinations: $\mathcal{O}(\prod_{i=1}^{N} |\phi_i|)$. To drastically reduce the required number of CNN evaluations, we use a prediction model giving an analytical approximation of the resulting AL for the CNN compressed with a specific k_{tuple}, noted $AL_{k_{\text{tuple}}}$. It is defined as: $AL_{k_{\text{tuple}}} = \sum_{i=1}^{N} \alpha_i \cdot AL_{k_i}$, with AL_{k_i} the measured AL during the local optimization for the layer i compressed using k_i shared values, and α_i trained coefficients. This prediction model is trained using standard multivariate linear regression.

V. Preliminary Results

We started by validating our method on LeNet trained with the MNIST data set. We did this because the reduced number of layers allows for the exhaustive search of all the layer combinations to

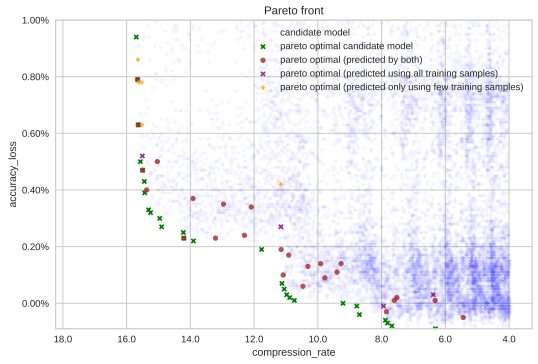

Fig. 5. Comparing Pareto front obtained using CNN evaluation and using the prediction model.

validate the precision of the prediction model. We used as input a self-trained regular 0.9% top-1 error rate LeNet, and the affordable accuracy loss used is 1%.

By solving the first sub-problem, we obtained the following layer candidates (only the selected k_i are displayed):

- **layer1** [2, 4, 6, 9, 23, 37]
- **layer2** [3, 7, 14, 26, 47, 78, 130]
- **layer3** [4, 8, 14, 22, 62, 90, 198]
- **layer4** [2, 3, 8, 10, 28, 50, 83]
- **layer5** [2, 3, 8, 13, 28, 52, 66]

The combination of the different k_i into k_{tuple} introduces $14,406$ possible candidate solutions. To validate our method we have performed the scoring of each of them, taking 1 hour on an NVIDIA V100. From the results, it appears that almost 78% of the candidate solution lead to $AL_{k_{\text{tuple}}}$ lower than 1%. Among these, 27 are Pareto optimal, and thus, the most interesting for our optimization problem. To validate that the $AL_{k_{\text{tuple}}}$ can be inferred using the prediction model, we fitted it using regression on 80% of the valid candidates for training and the other 20% for testing. We obtained the following set of α_i trained coefficients: $[1.26, 0.78, 0.95, 0.80, 0.92]$. Fig. 5 shows that the Pareto frontier obtained using the prediction model is close to the actual one obtained using the CNN evaluation.

VI. Conclusion & Future work

Our efficient exploration method reduced the number of CNN scoring operations required to optimize WS for a CNN to a few hundred from an initial 10^{12}. With promising results on small CNNs, we look forward to applying it to larger, state-of-the-art models, such as ResNet and MobileNet, targetting larger datasets such as CIFAR-10/100 and ImageNet.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, *et al.*, "Recent advances in deep learning for speech research at microsoft," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8604–8608, IEEE, 2013.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[4] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2722–2730, 2015.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan 2016.

[6] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2016.

[7] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[8] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-optimized fpga accelerator for deep convolutional neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 3, pp. 1–23, 2017.

[9] Q. Xu, T. Mytkowicz, and N. Sung Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, pp. 1–1, 01 2015.

[10] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," *arXiv preprint arXiv:1511.06488*, 2015.

[11] H. Tann, S. Hashemi, R. I. Bahar, and S. Reda, "Hardware-software codesign of accurate, multiplier-free deep neural networks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017.

[12] A. Carbon, J.-M. Philippe, O. Bichler, R. Schmit, B. Tain, D. Briand, N. Ventroux, M. Paindavoine, and O. Brousse, "Pneuro: A scalable energy-efficient programmable hardware accelerator for neural networks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1039–1044, IEEE, 2018.

[13] "Nvidia tensorrt: Programmable inference accelerator," 2021.

[14] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks," *arXiv e-prints*, p. arXiv:1604.03168, Apr. 2016.

[15] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, dec 2017.

[16] W. J. D. Song Han, Huizi Mao, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv*, 2016.

[17] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-Means: Re-Training and Parameter Sharing with Harder Cluster Assignments for Compressing Deep Convolutions," *arXiv*, June 2018.

[18] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[19] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[20] Y. Gorbachev, M. Fedorov, I. Slavutin, A. Tugarev, M. Fatekhov, and Y. Tarkan, "OpenVINO Deep Learning Workbench: Comprehensive Analysis and Tuning of Neural Networks Inference," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[21] O. Bichler, D. Briand, V. Gacoin, B. Bertelone, T. Allenet, and J. Thiele, "N2d2-neural network design & deployment," 2017.

[22] J. Bai, F. Lu, K. Zhang, *et al.*, "Onnx: Open neural network exchange." https://github.com/onnx/onnx, 2019.

[23] E. Dupuis, D. Novo, I. O'Connor, and A. Bosio, "Sensitivity analysis and compression opportunities in dnns using weight sharing," *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pp. 1–6, 2020.

[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.