Efficient Neural Networks: from SW optimization to specialized HW accelerators

Marcello Traiola, Angeliki Kritikakou, Silviu-Ioan Filip, Olivier Sentieys
Univ Rennes, CNRS, Inria, IRISA - UMR 6074, F-35000 Rennes, France

***Abstract*—Artificial Neural Networks (ANNs) appear to be one of the technological revolutions of recent human history. The capability of such systems does not come at a low cost, which led researchers to develop more and more efficient techniques to implement them. Optimization approaches have been developed, such as pruning and quantization, leading to reduced memory and computation requirements. Furthermore, such approaches are adapted to the specific hardware platform features to further increase efficiency. To improve it further, the HW programmability can be traded off in favor of more specialized custom HW ANN accelerators. In this education abstract, we illustrate how optimizing operations execution at different levels, from SW to HW, can improve the efficiency of ANN execution.**

***Index Terms*—HLS, Machine learning, hardware accelerators, FPGA**

I. Introduction

Artificial Neural Networks (ANNs) are one of the most intensively and widely used predictive models in the field of Machine Learning (ML) [1]. ANNs have proven outstanding results for many complex tasks and applications, such as object recognition in images/videos, natural language processing, satellite image recognition, robotics, aerospace, smart healthcare, and autonomous driving.

As ANNs have enormous algorithmic complexity, highly flexible and powerful software frameworks (e.g., Pytorch, Tensorflow) have been developed to increase productivity. Unfortunately, programming flexibility and high-level abstraction come at the cost of energy efficiency, especially when hardware characteristics are not taken into account. As illustrated in [2], dealing with the simple problem of multiplying two 4096-by-4096 matrices through a Python implementation wastes much of the performance available on modern computers. Indeed, Python uses additional operations to simplify programming and enhance productivity. Simply using a C implementation drastically reduces the number of operations, yielding an execution time 47 times faster, according to [2]. Further tailoring the code to exploit specific hardware platform features makes it run even faster. For instance, parallelizing the code to run on all the available processing cores, exploiting the processor's memory hierarchy, vectorizing the code, and using special instructions (e.g., Intel's Advanced Vector Extensions, or AVX) makes the final code perform more than 60,000 times faster than the original Python code [2].

Software frameworks exist that optimize the code and tailor it to the hardware features of existing platforms. For instance, TensorFlow Lite Converter converts a model into a memory-efficient format for use on memory-constrained CPU devices. Major software frameworks support code compilation to Graphics Processing Units (GPUs) kernels for accelerating ANN algorithms, leveraging, for example, the massively parallel nature of matrix multiplication. However, such highly programmable platforms often introduce overheads that are not always useful for ANN computations.

The key to further improving energy efficiency, while maintaining performance, is the design of hardware components that are specialized for ANN computations. As shown in the analysis of [3], the energy efficiency of chips increased as the amount of programming flexibility decreased. The analysis considered several chips, such as general-purpose microprocessors, software programmable DSPs, and dedicated signal processing designs with very limited programmability. Energy efficiency differences of four orders of magnitude were observed between the most flexible solutions and the most dedicated ones. As a result, by trading off programmability, specialized hardware can be used to save a lot of energy.

In this education abstract, we first discuss hardware-aware optimization approaches to tailor ANNs to specific device hardware features and then efficient methods for designing specialized hardware components for ANNs.

II. HardWare-aware optimization

Various optimization techniques, such as pruning and quantization, have been utilized to reduce energy consumption. Pruning and quantization are complementary techniques that can be applied together to achieve greater energy efficiency and memory savings. Although these techniques reduce the memory footprint and computation requirements, they may lead to a potential loss in the model's accuracy.

Pruning consists of intelligently sparsifying a dense ANN, which can be achieved through fine-grained and coarse-grained pruning. Fine-grained pruning usually removes connections [4], as it is inspired by the observation that removing weights with small magnitude, e.g., close to zero, marginally affects the ANN accuracy [5]. Coarse-grained pruning usually removes regular structures within convolutional layers (e.g., filters and channels) [6], thus significantly reducing the model's size and the number of operations [5].

Quantization reduces the memory footprint required for the ANN's parameters and activations by reducing the bits used for the representation of the arithmetic values. Floating-point with 32 bits (FP32) is the most commonly used format for training AI models. To improve energy efficiency, the format can be selected based on the hardware computation units of

2643-1726/24/$31.00 ©2024 IEEE
DOI 10.1109/CASES60062.2024.00009

the target architecture, e.g., the Turing GPU architecture from NVIDIA supports 1-bit, 4-bit, 8-bit and 16-bit arithmetic operations. It can be even optimized for ASIC or FPGA designs, leading to the most efficient hardware-accelerated solutions. Several quantization techniques exist to better fit modern ANN architectures [7]. Various libraries provide quantization functionalities, e.g., Qkeras, TensorFlow lite, LarQ, AIMET, Brevitas, TorchQuant, PyTorch quantization module etc.

Compilers, such as Tensorflow XLA and TVM, have been proposed to alleviate the burden of manually optimizing the ANN models for each hardware platform. These compilers highly optimize the transformation between model definition and specific code implementation, targeting the model specification and hardware architecture, leading to more efficient code for a given model and target device.

III. HW acceleration of basic ANN operations

Basic tasks of ANNs, such as convolutions, are easy to accelerate. Indeed, they can be implemented by using the well-known GEMM (General Matrix Multiply) operation. Specifically, through the *Image to Column*, or *Im2col*, operation, data are arranged so that the convolution output can be achieved by GEMM. In SW, highly optimized libraries for CPU or GPU speed up GEMM execution in different ways, such as reordering the loops, improving data locality (better cache usage), and tiling (looping on small enough submatrices to fit in the cache). Moreover, deeply specialized hardware accelerators can push the limits further to execute GEMM/CONV more efficiently. However, this may also require increased effort from the programmer/designer. Indeed, deep knowledge of the hardware is required to propose energy-efficient models. Number representations and precisions are key techniques, as well as memory access since execution is often memory-bound.

Once data are fetched from the main memory, it is very important to reuse it as much as possible, given the high cost of moving data and the performance bottleneck that memory access introduces. Let us consider a Convolution Kernel having $\mathcal{C}$ input channels, $\mathcal{K}$ output channels, batch size $\mathcal{N}$, filter dimensions $\mathcal{R} * \mathcal{S}$ and output activations dimension $\mathcal{T} * \mathcal{U}$. Different data-reuse opportunities are available, such as (i) input reuse: different filters are applied to the same input; each input is reused $\mathcal{K}$ times; (ii) filter (weight) reuse: when processing a batch of size N, all inputs are applied to the same filter, and each filter weight is reused $\mathcal{N}$ times; (iii) conv. reuse: filters slide across different positions of the same input; each weight is reused $\approx \mathcal{T} * \mathcal{U}$ times, and each input is reused $\approx \mathcal{R} * \mathcal{S}$ times.

Accelerators use multiple Processing Elements (PEs), including some logic and local memory registers, to enable data reuse. Different approaches can be adopted, such as (i) temporal reuse: using cache memories/registers so the same data is used more than once over time by the same PE; (ii) spatial reuse: using systolic/multicast architectures where the same data is used by more than one PE at different spatial locations of the hw; (iii) hybrid temporal and spatial Reuse: both cache memories/registers and multiple PEs are used. On top of that, accelerator architectures can be sequential or pipelined to further boost the performance.

Commercially available ANN accelerators combine matrices of PEs in different architectures and mainly aim at parallelizing the second and third inner loops of matrix multiplication. For instance, the Nvidia NVDLA architecture uses Adder trees with weight (sub)line multicasting, while Google TPU utilizes systolic Multiply-And-Accumulate (MAC) with systolic multicast. Such accelerators are engineered to be very efficient in executing basic ANN operations while allowing the programmer some programming flexibility.

IV. Specialized ANN HW accelerators

As also mentioned in Section I, more specialized accelerators, with very low programmability, can lead to very efficient ANN implementations. This is the case of *Streaming Dataflow* architectures, which enable highly customized datapath and custom arithmetic precision for both weights and activations. While matrices of processing elements (as described in Section III) are customized for typical ANN operations (e.g., GEMM) and aim to offer also programmability/flexibility, streaming dataflow architectures are customized/adapted for specific ANN topologies to provide higher efficiency, lower latency, and higher throughput. This is possible thanks to the extensive pipelining and the absence of intermediate buffering between consecutive NN layers. Given the low programmability/flexibility of such solutions, streaming dataflow architectures are usually deployed on FPGA devices, where more flexibility can be achieved through reconfiguration. If hardware resources are enough, a circuit producing one inference per clock cycle can be deployed. However, given the large dimension of ANNs, a trade-off between available resources and throughput is usually necessary. High-level frameworks and compilers, such as FINN [8] and HLS4ML [9], provide an end-to-end flow to create such streaming dataflow accelerators, from high-level definition (e.g., in Pytorch, Tensorflow, or Keras) to Hardware Description Language (HDL) code ready to synthesize and deploy on a hardware target.

References

[1] Y. LeCun *et al.*, "Deep learning," *Nature*, vol. 22, no. 3, pp. 436–44, May 2015.

[2] C. E. Leiserson *et al.*, "There's plenty of room at the Top: What will drive computer performance after Moore's law?" *Science*, vol. 368, no. 6495, p. eaam9744, Jun. 2020.

[3] N. Zhang *et al.*, "The Cost of Flexibility in Systems on a Chip Design for Signal Processing Applications."

[4] N. Lee *et al.*, "SNIP: single-shot network pruning based on connection sensitivity," *CoRR*, vol. abs/1810.02340, 2018.

[5] K. Balaskas *et al.*, "Hardware-aware dnn compression via diverse pruning and mixed-precision quantization," *IEEE TETC*, p. 1–14, 2024.

[6] Y. Wang *et al.*, "Non-structured DNN weight pruning considered harmful," *CoRR*, vol. abs/1907.02124, 2019.

[7] M. Nagel *et al.*, "A white paper on neural network quantization," 2021.

[8] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *FPGA*, Feb 2017, pp. 65–74.

[9] J. Duarte *et al.*, "Fast Inference of Deep Neural Networks for Real-Time Particle Physics Applications," *FPGA*, pp. 305–335, Feb 2019.