FPGA BASED IN-MEMORY AI COMPUTING

Domenik Helms[(1)], Quentin Dariol[(1)], Kim Grüttner[(1)], Behnam R Perjikolaei[(2)], Lukas Einhaus[(3)], Gregor Schiele[(3)]
[(1)] German Aerospace Center (DLR), Oldenburg, Germany, Email: first.last@dlr.de, [(2)] OFFIS, Oldenburg, Germany, Email: behnam.razi.perjikolaei@offis.de, [(3)]University of Duisburg-Essen, Duisburg, Germany, Email: first.last@uni-due.de

Extended Abstract

The advent of AI in vehicles of all kinds is simultaneously creating the need for more and most often also very large computing capacities. Depending on the type of vehicle, this gives rise to various problems: while overall hardware and engineering costs dominate for airplanes, in fully electrical cars the costs for computing hardware are more of a matter. Common in both domains are tight requirements on the size, weight and space of the hardware, especially for drones and satellites, where this is most challenging. For airplanes and especially for satellites, an additional challenge is the radiation resistance of the usually very memory-intensive AI systems.

We therefore propose an FPGA-based in-memory AI computation methodology, which is so far only applicable for small AI systems, but works exclusively with the local memory elements of FPGAs: lookup tables (LUTs) and registers. By not using external and thus slow, inefficient and radiation-sensitive DRAM, but only local SRAM, we can make AI systems faster, lighter and more efficient than is possible with conventional GPUs or AI accelerators. All known radiation hardening techniques for FPGAs also work for our systems.

The small size and low energy profile of our solution fits well with a near-sensor implementation, also reducing cost, weight, and energy of the internal communication busses. Our AI systems are capable of processing any one-dimensional sensor output (a small amount c of values $x_c(t)$ per time), either at extremely high sampling rates of several *MHz* with mere μs of latency, or can work in bursts on data with lower sampling rates, realizing very low power consumption.

In embedded neural networks, timing and performance are dominated by access time and energy cost per memory access. Strictly speaking, an FPGA consists of hundreds of thousands of small, fast and energy-efficient memory blocks (LUTs). If block memory does not need to be accessed, the resulting neural network will compute much faster and with much lower energy cost.

Our method for FPGA based in-memory computing uses precomputed convolutions and stores them in the LUTs. This allows one-dimensional convolutional neural networks (CNNs) to execute without global memory accesses: activations are stored in local registers, and weights and biases of all neurons are encoded in LUTs. Each neuron is assigned its exclusive share of logic circuits. This avoids reconfiguration overhead, but limits the applicability of the overall method to comparatively small CNN, since we need several LUTs per neuron and even the largest FPGAs only provide hundreds of thousands of LUTs.

To enable this "in-LUT processing," we had to limit the set of available neural network layers. We identified and implemented a set which is sufficient for the neural network to function, but which can be efficiently implemented as FPGA without memory access. Our philosophy is that it is better to adapt the neural network during training to make the best use of the limited resources available than to try to optimize the functions in hardware, resulting in an unconstrained neural network.

When being limited to 1 dimensional CNNs, which are sufficient for streaming processing, e.g .for sensor preprocessing, a Conv1D layer can be represented as:

$$y_f(t) = max\left(0, \sum_{c=0}^{C-1} \sum_{i=0}^{N-1} w_{i,c,f} \cdot x_c(t - i \cdot \Delta t) + b_f\right) \quad (1)$$

where $y_f(t)$ is the layer's output vector (one value per filter) and $x_c(t)$ is the layer's input vector, which is only relevant at discrete and equidistant time steps $x_{c,i} = x_c(t - i \cdot \Delta t)$. Also $y_f(t)$ is only defined at certain discrete timesteps $y_{f,j} = y_f(j \cdot \Delta t)$.

Assuming, that the $x_{c,i}$ can be quantized to a very low bitwidth (e.g. 2bit), a push register is a hardware friendly and convenient complexity reduction, making a hardware implementing Equation 1 only depending on the recent-most values $x_c(t)$.

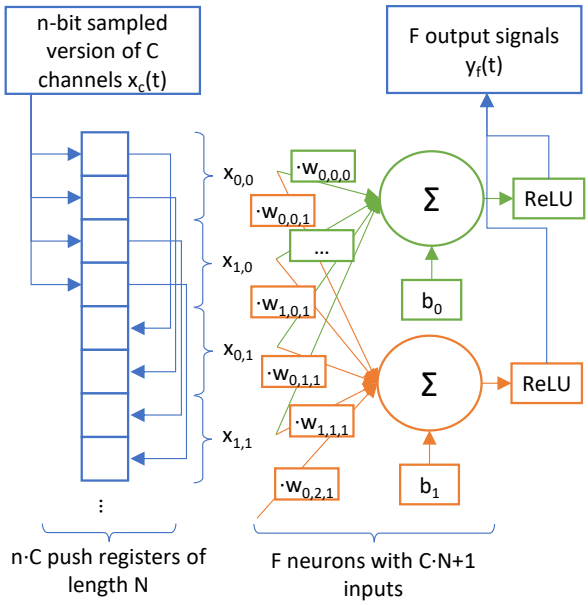

Figure 1: A shift register at the input side simplifies the Conv1D layer to a series of fully connected neurons.

As presented in *Figure 1*, the shift register not only takes over the storage of the older input values, but it also implements the entire shift and recompute aspect of the convolutional layer. The remainder to be implemented is a pure dense layer. Stride values larger than 1 can also be

easily realized by reducing the activation frequency of the dense part of the system, and thus also reducing the sampling frequency of the output signals.

In order to implement the dense neurons efficiently and in a hardware friendly way, we need to apply two steps: The first step is to strictly reduce the number of inputs to each neuron. Network topologies with huge input counts have to be replaced with a tree like structure of neurons, each with only a fraction of the number of inputs of the original neuron. Several different methods are available for such a reduction, but we focus in our work on depthwise separable convolutions, as shown in *Figure 2*, which allows to process the per channel convolution first (full kernel size, but per one channel) and the per channel convolution afterwards (kernel size already reduced to 1, but all channels).

The implementation of the filters is straight forward: For each filter, a version of the separated neurons is instantiated, reading from the same push register, but resulting in a separate output structure (rf. *Figure 2* indicated in green for filter 0 and orange for filter 1).

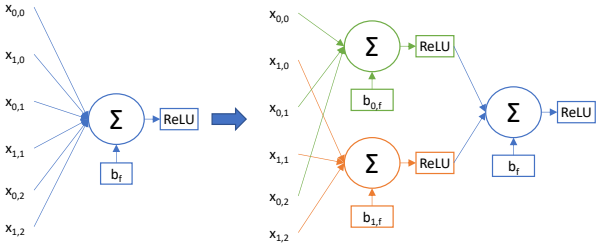

Figure 2: Depthwise separable convolution reduces the number of inputs per neuron

In the second step, which is the core idea of the entire methodology, we exploit the low bitwidth of the input and output signals and the low number of inputs and thus the finite amount of possible input combinations: Instead of actually implementing a series of low bitwidth multiplications, we precompute the per neuron output for all possible input states, downsample them to the output bitwidth n_y and store them in a number of look-up tables.

For a neuron with N inputs of n_x bit input width and n_y bit output width, we can precompute the function

$$y = max(0, \sum_{i=0}^{N-1} w_i \cdot x_i + b) \qquad (2)$$

as an n_y bit value for all $2^{N \cdot n_x}$ possible input states, requiring $n_y \cdot 2^{N \cdot n_x}$ bit of memory, or $n_y \cdot 2^{N \cdot n_x - 6}$ recent FPGA look up tables. Such a structure is referred to as an n-to-m cell with $n = N \cdot n_x$ the number of overall input bits and m the number of output bits (rf. *Figure 3* left).

The major advantage of this approach in comparison to all other approaches is, that it allows the weights and biases to remain real values. Only the input- and output values have to be quantized, which allows for quick and easy training, avoiding techniques such as straight through estimators.

An optional max or average pooling layer can be implemented by yet another push register in combination with either a low bitwidth summation operation for average pooling or a max function for maximum pooling (rf. *Figure 3* right).

Both, a stride larger than 1 as well as a pooling effectively result in a frequency reduction of the signal to be analysed, which then leads to a lower power consumption of the respective hardware blocks, as they operate and thus switch and thus dissipate energy less frequently.

A much more relevant aspect of this frequency reduction is, that each value of the later, lower frequency layers represents a larger interval of time. Due to the restrictions in the input number for synapses, it is not feasible, to do a very long convolution in order to observe features, occurring on a much larger timescale than the sampling frequency. Even with depthwise separation, the kernel size is limited to $N \approx 10$ for binary ($n_x = 1$) and $N \approx 5$ for 2bit values ($n_x = 2$), as otherwise, the LUT count for the $(N \cdot n_x)$-to-n_y block would rise exponentially.

Thus, for an input signal entering with a sampling frequency of f_s, the initial convolution layer can only observe a timeframe N/f_s . Each stride or pooling size multiplies this time so that single values in the later layers can represent arbitrarily large time intervals.

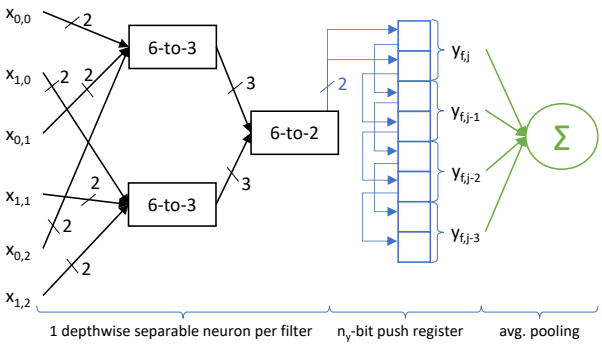

Figure 3 left: The separated sub-neurons can be represented as n-to-m lookup tables. Right: A push register and a sum (or max) function implement an average (or maximum) pooling.

Early Evaluation

To demonstrate and evaluate the performance of our method, we implemented CNN-based ECG recognition. Our implementation used only 40% of the available LUTs on the Spartan S15 chip and none of the block RAM or DSP circuitry. The system processed 500 pre-recorded ECGs with 5575 samples in 281ms, consuming a total of only 73mJ, resulting in 10 million samples per second and an energy cost of 26.2nJ per sample.