EUROPEAN WORKSHOP ON ON-BOARD DATA PROCESSING (OBDP2021), 14-17 JUNE 2021

FPGA BASED LOW LATENCY, LOW POWER STREAM PROCESSING AI

Domenik Helms[(1)], Mark Kettner[(1)], Behnam Razi Perjikolaei[(1)], Lukas Einhaus[(2)], Christopher Ringhofer[(2)], Chao Qian[(2)], Gregor Schiele[(2)]

[(1)] OFFIS, Escherweg 2, 26121 Oldenburg, Germany, Email: first.[second].last@offis.de
[(2)]University of Duisburg-Essen, Forsthausweg 2, 47057 Duisburg, Germany, Email:first.last@uni-due.de

ABSTRACT

The timing and power of an embedded neural network application is usually dominated by the access time and the energy cost per memory access. From a technical point of view, the hundreds of thousands of look-up tables (LUT) of a field programmable gate array (FPGA) circuit are nothing more than small but fast and energy-efficiently accessible memory blocks. If the accesses to the block memory can be reduced or, as in our case, avoided altogether, the resulting neural network would compute much faster and with far lower energy costs.

We have therefore developed a design scheme that uses precomputed convolutions and stores them in the LUT memories. This allows small (mostly one-dimensional) convolutional neural networks (CNN) to be executed without block memory accesses: Activations are stored in the local per LUT registers and the weights and biases of all neurons are encoded in the lookup tables. Each neuron is assigned its exclusive share of logic circuits. This completely avoids the need for memory accesses to reconfigure a neuron with new weights and allows us to perform weight optimisations at design time. However, it limits the applicability of the overall method to comparatively small neural networks, since we need several LUTs per neuron and even the largest FPGAs only provide hundreds of thousands of LUTs.

To make this "in LUT processing" possible, we had to limit the set of available neural network functions. We have identified and implemented a set of functions that are sufficient to make the neural network work, but which can all be implemented efficiently in an 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 optimise the functions in hardware, resulting in a non-limited neural network.

To realise this design scheme, we developed a set of design tools, helping the AI designer to convert a given reference AI in TensorFlow into an equivalent network of the available hardware functions. Our tools also allow to finetune the AI to compensate the accuracy loss from changing the implementation. The two most powerful optimization techniques we applied are variable bitwidth quantization and depth-wise separation of convolutions.

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

1. INTRODUCTION

Artifical intelligence (AI) technologies crossed the threshold from interesting object of study to relevant application almost ten years ago. This is due to three main reasons: (1) the availability of large amounts of data for training; (2) new, more efficient algorithms for training and inference; and (3) new, very efficient hardware components that can handle the enormous amounts of data access and computation in little time and with little power dissipation. Although all three aspects have evolved even further since then, it is the efficiency of the hardware that – following Moore's Law – allows the most significant improvements in AI. To function properly, different categories of AI applications require different computational speeds and, more crucially, corresponding memory access times, as shown in Figure 1.

Power efficiency is fundamentally limited by the energy cost of computing an arithmetic operation and the energy cost of accessing and transporting stored information. With each technology generation this limit still increases exponentially. Each category of AI application operates in a certain speed band and accordingly has a certain minimum power dissipation that makes it applicable to a certain class of system.

As Figure 1 shows, simple AI applications are possible today even for edge or IoT applications. Currently, however, this is only possible if dedicated hardware is adapted for the simplest AI applications at very low operating speeds. Furthermore, FPGAs – as they are

currently used as AI accelerators – fall well short of the technological possibilities due to their high hardware overhead.

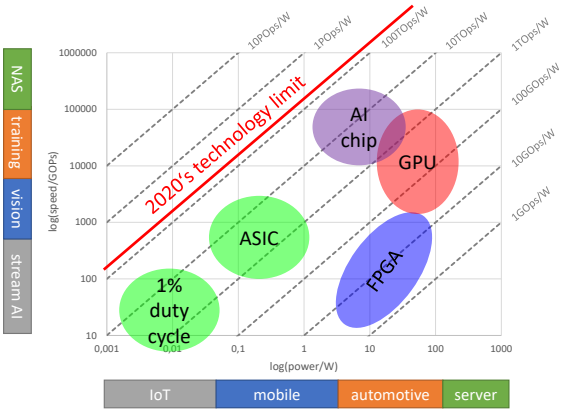

Figure 1 Power efficiency of computations is dictating the applicability of AI applications (qualitative visualization)

The main reason for this is that in today's FPGA-based implementations, the very versatile but also very complex Look-Up Tables (LUT) are used to realise mathematical operations or, at best, logical operations. However, LUT cells are much more powerful and capable of realising highly efficient AI applications very close to the technological limit.

Our paper is structured as follows. In the following chapter, we will review the relevant state of the art in embedded and edge AI technologies. Afterwards, we will describe our idea, give some details on the implementation and report on the initial application's performance.

2. RELATED WORK

There are various ideas on how FPGAs can be used to implement AI algorithms. The two most important classes so far are the generation of a dedicated, bitwidth-optimised computing kernel [1] and the efficient implementation of binary neural networks [2,3].

The key idea of the dedicated kernels is to analyse the neural network to be implemented in hardware and to optimize the bitwidth to an unrestricted value. The OpenVINO synthesis engine for instance, will set up a computation kernel which is highly optimized to compute all layers of the AI model to be implemented one after the other and is implemented in exactly the required bitwidth.

The key idea for implementing binary neural networks is, that in the binary form, a multiplication of two values falls back to a simple Boolean operation (either XOR or AND), which can be very efficiently implemented in FPGA. In order to mitigate the accuracy loss, introduced by the extreme quantization, the network topology can be enlarged before quantization [4].

Both schemes still require the use of LUT cells, typically implemented as 32bit or 64bit programmable memory tables for arithmetic or binary operations. An ASIC implementation of these topologies could approach the technology limit in terms of efficiency, but as an FPGA implementation these ideas still fall far short due to the overhead of replacing a 6-input gate (typically 12 transistors) with a 64bit 5T SRAM array (>320 transistors) plus additional logic.

As we will describe in the next section, we instead try to exploit the full potential of these small local memory arrays. We follow the philosophy of developing structures that can be efficiently represented in an FPGA and then using AI training to make the best use of the available resources.

However, we have to impose strict constraints on the AI topologies, which have to be compensated for in the synthesis and training of the AI applications. The following work describes useful techniques for implementing such topology constraints with little loss of system accuracy:

[5] presents the PACT activation function, which introduces clipping, which normally only occurs during quantisation for hardware conversion, already during training. In this way, the unavoidable clipping artefacts resulting from quantisation can be compensated for or even exploited during training.

There are many ways of reducing the complexity, in our case especially the synapses count of a convolutional layer. One of the most effective ones is depth-wise separable convolution [6], which is also used in many high efficiency applications such as MobileNetV2 and Xception.

Extreme quantisation down to 2 bits must be done as cleverly as possible to prevent a large loss in accuracy [7]. A good possibility for this is statistics-aware weight binning [8], in which statistical moments of the weight distribution are used to determine the optimal quantisation levels.

3. CORE IDEA

Our goal is to develop an approach with which small, simple neural networks can be implemented highly energy efficiently on an FPGA. In order to do so, we restrict ourselves to stream processing based problems, i.e. to one-dimensional convolutional neural networks (Conv1D). In terms of activation function, we only support the most common and most simple of them all,

which is the ReLU function. Besides this, only the one dimensional pooling and the dense layers are supported, yet. A Conv1D layer in a stream processing environment implements the following general equation:

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

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 $y_{f,j} = y_f(j \cdot \Delta t)$.

3.1. Convolution

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 $x_c(t)$.

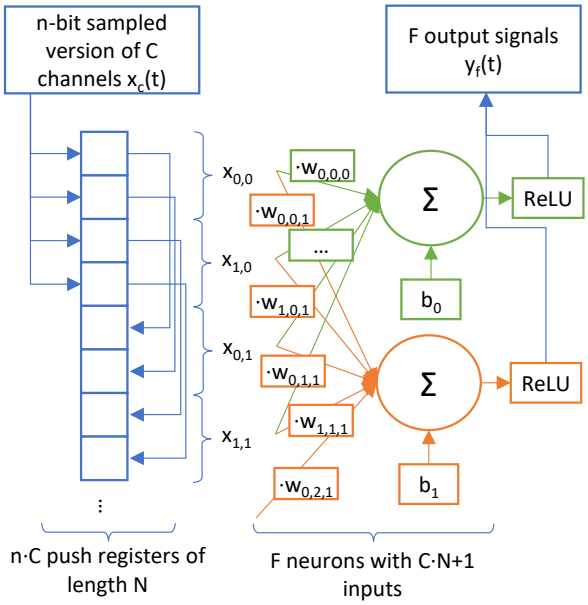

Figure 2: A shift register at the input side simplifies the Conv1D layer to a series of fully connected neurons.

As presented in *Figure 2*, 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.

3.2. Neuron implementation

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 3*, 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 1*, indicated in green for filter 0 and orange for filter 1).

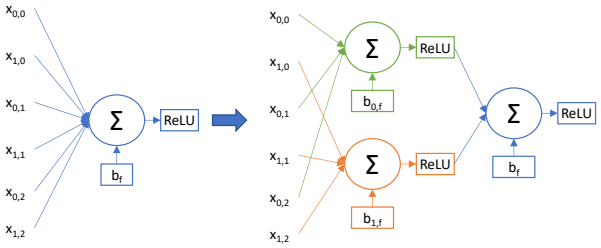

Figure 3: Depthwise separable convolution reduces the number of inputs per neuron

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

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

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 4* 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 [9].

Additionally, it is not even needed to interpret the quantized inputs as integer values or round the output values to integer values, but instead each of the possible

input- and output-states can code one arbitrary real value, allowing the much more powerful codebook quantization [10] for zero-effort.

3.3. Pooling

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 4* 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.

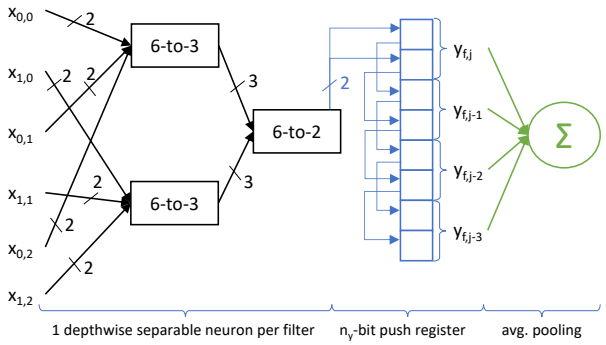

Figure 4 left: The separated sub-neurons can be represented as n-to-m lookup tables. Right: A push register and a sum (or max) function implement an average (or maximum) pooling.

4. SYNTHESIS AUTOMATIZATION

The principles presented above allow to design small AI systems on FPGA in an exclusive and memory free way: As each logical neuron is realized as some explicit and exclusive hardware, i.e. a few LUT cells somewhere on the chip, it is possible for the weights and biases to permanently remain coded implicitly into the LUT cell's configuration. The activations are passed from layer to layer through the configurable metal connections of the FPGA and they are stored between the layers in local registers. Thus, at no point is it necessary to read or write any kind of data from or to the block RAM of the FPGA. This allows for extremely high sampling rates to be processed, as all computations are done in parallel. Additionally, this also allows neural network execution at very low energy per computation costs.

In typical AI implementations, inference time as well as energy costs are dominated by the time and energy costs for memory accesses. Optimizing hardware execution of neural networks usually means reorganization of the execution in order to reduce memory accesses or to replace a global memory access by a more local one, exploiting the various memory hierarchies. In this approach we have taken this to the extreme by completely shifting the weights and bias storage from the main memory into the tens of thousands local 64bit storages, the LUT cells are. Storing activations is shifted even further from the main memory into the registers of the FPGA.

As a drawback, the resulting structure is extremely hard to program, requiring the usage of a hardware description language and the translation of millions of training parameters from a training framework into parameters of the hardware description itself.

In order to make this idea applicable, we thus had to automate the transition from a trained model in a high-level AI framework such as TensorFlow into a configuration bitfile for the FPGA without the need of manual processing in between.

4.1. Network simplification

We started with defining a list of valid TensorFlow layers (the LUTNet library), for which a hardware efficient FPGA implementation is possible. For our first demonstrator, we limited ourselves here to the one-dimensional convolution and the pooling as described above as well as a batch normalization layer, several implementations of dense layers and a few variations of the convolution layer.

The designer can then train and evaluate a reference model, still computing on float values and test and try different configurations, trying to maximize accuracy and/or efficiency. Afterwards the designer can initiate an automated variable quantization and a depthwise separation of the convolution.

The final simplified TensorFlow model can be tested and if necessary updated. The extreme quantization is represented in Tensorflow by customized and

configurable activations, representing the effects of quantization in TensorFlow. The final network can then be translated into our hardware independent intermediate description language, called macro transfer language (MTL).

MTL can be simulated in our tool flow to determine if the current neural network will meet all resource constraints. Otherwise, the user can adapt the reference model, restart the conversion flow and recheck if the hardware requirements are met.

Once this step is converged, the MTL together with pre-implemented hardware templates for each valid layer in the LUTNet library can be send to the final tool we developed, which is a VHDL code generator. This generator is combining and configuring the library templates using the structure and parameters described in the MTL into a fully pipelined VHDL hardware description of the neural network. From there the design can be handed over to a standard FPGA design tool such as Xilinx Vivado.

5. EXAMPLE APPLICATION

In order to test the methodology and to assess the resulting hardware, we applied it to a medical application which was part of the national KI-Sprung challenge [11], where the task was to detect artefacts in a stereo 500Hz 16bit ECG signal.

The demonstrator uses a reference neural network consisting of the following sequence of layers:
- a regular (not separated) CONV1D as input layer
- a one-dimensional max pooling layer
- three depth-wise separated CONV1D layers
- a one-dimensional max pooling layer
- one depth-wise separated CONV1D layer
- a one-dimensional max pooling layer
- a final dense classifier

This neural network was described in TensorFlow, then trained using a quantization-aware training scheme, and finally evaluated. The final AI model is then translated into MTL and finally synthesized into a bitfile for the Spartan 7 S15 FPGA in 28nm technology.

The final design uses 2155 LUT cells, which is 26% of all available LUTs as well as 7.6% of the available Flip-Flops. An additional 14% of LUTs is used to interface with the environment of the FPGA.

The final system was able to detect over 90% of all artefacts in the ECG data and could infer a total of 2.8 million samples in 0.28s while consuming 258mW of power of which one quarter (63mW) is due to a flash buffer, which is used to process the low frequency input data in bursts.

In a high frequency application, the same system would be able to apply a six layer neural network on a 10 MHz stereo input signal, only dissipating 182mW.

In this low frequency scenario, the system is working in bursts, duty cycling in between and needs only 26nJ of energy for a complete inference.

For comparison, recent low power AI accelerators such as the Intel Neural Compute Stick 2 (16nm technology node) ranks around 2W/Tops, which translates to 2fJ per operation. In conventional hardware, our neural network would need roughly 30,000 operations to perform and thus would need 60nJ per inference, but admittedly in 16 bit, instead of 2-3 bit.

6. CONCLUSION

In this paper, we presented a radically new scheme for using FPGAs for AI inference. We showed that our implementation can keep pace with highly optimized AI accelerators having two generation better technology.

There are still many open issues in our flow that we can use to further optimize produced neural networks. First, we used the smallest available FPGA and still only needed a quarter of it for our network. Thus, there is a huge potential for much larger applications and redundant components, which we will focus on in future work.

Additionally, there are the block RAM and DSP cells, completely unused so far, which could be used for instance for a high bitwidth input layer, a high bitwidth dense classifier or a single unconstrained 1D or 2D convolution layer. These would increase the accuracy and applicability of this approach a lot.

Finally, if reconfigurability in field is not necessary, the VHDL file, describing the inference engine could of course also be synthesized into a full custom ASIC chip. Such a device, even though having high development cost, would by far outperform any available AI accelerator hardware in terms of inference speed and efficiency. Using radiation hardened Flip-Flops it would also be a good candidate for Aeronautic and Space applications as an ASIC implementation would be completely memory-free.

ACKNOWLEDGEMENTS

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the KI-Sprung framework (project number 16ES1124K).

REFERENCESS

[1] Intel OpenVINO framework. https://docs.openvinotoolkit.org/latest/openvino_docs_IE_DG_supported_plugins_FPGA.html, 2020.

[2] Corey Lammie, Wei Xiang, Mostafa Rahimi Azghadi: Training Progressively Binarizing Deep Networks using FPGAs, ISCAS 2020.

[3] E. Wang, J. J. Davis, P. Y. K. Cheung, G. A. Constantinides: LUTNet: Rethinking Inference in FPGA Soft Logic. FCCM, 2019.

[4] Xiaofan Lin, Cong Zhao, Wei Pan: Towards Accurate Binary Convolutional Neural Network. arXiv:1711.11294, 2017.

[5] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan: PACT: parameterized clipping activation for quantized neural networks. arXiv:1805.06085v2, 2018.

[6] L. Bai, Y. Zhao, X. Huang: A CNN Accelerator on FPGA Using Depthwise Separable Convolution. doi: 10.1109/TCSII.2018.2865896.

[7] D. Helms, K. Amende, S. Bukhari, T. de Graaff, A. Frickenstein, F. Hafner, T. Hirscher, S. Mantowsky, G. Schneider, M.-R. Vemparala: Optimizing Neural Networks for Embedded Hardware. SMACD 2021

[8] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, Pierce Chuang: Accurate and efficient 2-bit quantized neural networks. MLSys 2019.

[9] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, Jack Xin: Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets. arXiv:1903.05662, 2019.

[10] S. Han, H. Mao, and W. J. Dally: Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. ICLR 2016.

[11] https://www.elektronikforschung.de/projekte/ki-sprung