A C++ Library for Rapid Exploration of Binary Neural Networks on Reconfigurable Logic

Yaman Umuroglu*†, Nicholas J. Fraser*‡, Giulio Gambardella*, Michaela Blott*

*Xilinx Research Labs; †Norwegian University of Science and Technology; ‡University of Sydney

ABSTRACT

Convolutional neural networks (CNNs) with floating point parameters have been successfully applied to a wide range of computer vision tasks. It is desirable to use CNNs in both embedded applications, which typically require fast response time, low power and lean memory requirements, as well as in datacenter deployments that seek high performance and power efficiency. However, these requirements clash with the characteristics of today's CNNs, which may contain millions of floating-point parameters and require billions of operations to recognize one image. The resulting cost of floating point computation and the need to read many parameters from off-chip memory translate into severe limitations to deployment. Many different types of hardware architectures are being harnessed to address the requirements. For example, GPUs offer high floating point performance, but are limited in power efficiency. Field programmable gate arrays (FPGAs) typically deliver lower performance on floating point data types, but can provide much higher computational capacity and power efficiency using reduced precision arithmetic. Recent studies have shown [1, 2, 3] that neural networks can classify accurately even under extreme quantization, using one-bit values to represent parameters and perform arithmetic, making them suitable for energy-efficient image classification on FPGAs. Nonetheless, FPGA deployment has been historically considered a time-consuming task. Recent advances in High-Level Synthesis (HLS) show how development times can be reduced significantly in numerous application domains. We explore how to leverage Vivado HLS to build a library and toolflow that generates binary neural network inference accelerators, both for peak and user-defined performance requirements. The library targets the most common CNN layers (convolutional, fully connected and max pooling) to allow the user to implement the desired neural network. To permit customized implementation while taking advantage of compile-time optimizations, we adopted templated C++ functions, allowing the user to specify both the neural network topology (e.g., number of neurons and synapses) and implementation (e.g., number of compute resources onto which neurons are time-multiplexed).

Figure 1 shows an excerpt from the definition of the Matrix–Vector–Threshold Unit (MVTU), which is used as a library function for building convolutional and fully connected layers. The customized values of folding factors (e.g. the `for` loops `nm` and `sf`) are evaluated at synthesis time depending on the layer parameters, while the processing elements (PEs) working in parallel are instantiated by the inner `for` loop with the UNROLL pragma. Each processing element performs a

```
for (unsigned int nm = 0; nm < neuronFold; nm++) {
    for (unsigned int sf = 0; sf < synapseFold; sf++) {
#pragma HLS PIPELINE II=1
            ap_uint<SIMDWidth> inElem;
            if (nm == 0) {
               inElem = in.read();
               inputBuf[sf] = inElem;
            } else {
               inElem = inputBuf[sf];
            }
            for (unsigned int pe = 0; pe < PECount; pe++) {
#pragma HLS UNROLL
               ap_uint<SIMDWidth> weight = weightMem[pe][nm * synapseFold + sf];
               ap_uint<SIMDWidth> masked = ~(weight ^ inElem);
               accPopCount[pe] += NaivePopCount<SIMDWidth, PopCountWidth>(masked);
            }
    }
    ap_uint<PECount> outElem = 0;
    for (unsigned int pe = 0; pe < PECount; pe++) {
#pragma HLS UNROLL
            outElem[pe, pe) = accPopCount[pe] > thresMem[pe][nm] ? 1 : 0;
            accPopCount[pe] = 0;          // clear the accumulator
    }
```

Figure 1: Excerpt from MVTU definition in Vivado HLS.

binarized dot product (bitwise XNOR and popcount) and activation by comparing the results with a threshold. Figure 2

```
void DoCompute(ap_uint<64> * in, ap_uint<64> * out) {
#pragma HLS DATAFLOW
    stream<ap_uint<64> > memInStrm("memInStrm");
    stream<ap_uint<64> > InStrm("InStrm");
                .
                .
                .
    stream<ap_uint<64> > memOutStrm("memOutStrm");
    Mem2Stream<64, inBytesPadded>(in, memInStrm);
    StreamingMatrixVector<L0_SIMD, L0_PE, 16, L0_MW, L0_MH, L0_WMEM, L0_TMEM>
            (InStrm, inter0, weightMem0, thresMem0);
    StreamingMatrixVector<L1_SIMD, L1_PE, 16, L1_MW, L1_MH, L1_WMEM, L1_TMEM>
            (inter0, inter1, weightMem1, thresMem1);
    StreamingMatrixVector<L2_SIMD, L2_PE, 16, L2_MW, L2_MH, L2_WMEM, L2_TMEM>
            (inter1, inter2, weightMem2, thresMem2);
    StreamingMatrixVector<L3_SIMD, L3_PE, 16, L3_MW, L3_MH, L3_WMEM, L3_TMEM>
            (inter2, outstream, weightMem3, thresMem3);
    Stream2Mem<64, outBytesPadded>(memOutStrm, out);
}
```

Figure 2: Combining layers to form a BNN implementation.

shows the definition of a fully connected neural network with input layer and 3 hidden layers. To build a binarized neural network, each layer is instantiated with the desired number of processing elements, and connected into the network via on-chip streaming channels. We also built a model based on characterizations of each library component, which computes the necessary parameters to achieve the desired design target in terms of resources, classification performance and latency. The first prototype from the implementation of a 3-layer fully connected network for inference on the MNIST dataset is able to reach a peak frame rate of 12.3 million FPS with 0.31µs latency with ~30% of hardware resources on a ZC7045 SoC, while 12 k FPS can be achieved with 2 % of resources on the same device by scaling down the parallelism.

1. REFERENCES

[1] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[2] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[3] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.