ALPHA DATA

An Open Source FPGA CNN Library

An Open Source FPGA CNN Library

Introduction

Convolutional neural networks have become the core component of a large number of hyperscale deployed machine learning algorithms used in image and vision recognition tasks. Low-bitwidth FPGA implementations of these networks provide a potential path to higher throughput and lower power machine learning inference solutions.

The primary purpose of this toolbox is to provide an easy path for developers to investigate the accuracy implications of switching from floating point defined network weights to low bitwidth fixed point weights on each of the layers making up the network.

The second purpose of this toolbox is educational as it allows exploration of structural implications of the different network layers. This applies both at the implications of fixed width bitwidths on arithmetic, which have changing requirements through even the simple multiply accumulation arithmetic required to implement a neuron, and to the structural and parallelization requirements of moving the weight, input and intermediate data on, off and around the chip.

A third purpose is to provide an open source reference code (with a BSD-like licence) to fully allow users to explore these structures and modify the code to meet their requirements. The code has been designed to work with an Open Source Compiler to allow the user to explore the designs on a CPU based system, without needing to purchase FPGA hardware and FPGA tools up-front

The final purpose is to provide some reference designs for hardware implementations of these algorithms on Xilinx FPGA based Alpha Data hardware, running in both an x86 CPU PCIe plug in card configuration and an IBM Power8 CAPI plug in card configuration.

By allowing developers to explore FPGA CNN implementations without having to purchase hardware or software up-front, this package also aims to inform users about the possible benefits of using FPGAs for their machine learning inference and give them confidence to use FPGAs in machine learning, even if they choose not to design the FPGA network circuit themselves, but choose an off the shelf commercial machine learning FPGA library such as those now available from Xilinx or IBM.

The toolbox code can be downloaded from *ftp://ftp.alpha-data.com/pub/appnotes/cnn/adcnnlib-v1_0_0.zip* and the full paper from *ftp://ftp.alpha-data.com/pub/appnotes/cnn/ad-an-0055_v1_0.pdf*

Structural Neural Network Description using a Hardware Description Language

The example code in the toolbox is provided in the ADA derived hardware description language VHDL.

Why VHDL? Why not OpenCL?

For machine language inference there are a number of advantages in using an HDL over a traditional software description. Firstly, the most basic descriptions of neural networks are parallel hardware descriptions and, therefore, it is actually quite easy to model these networks as parallel neurons without looking at a lower level serial execution implementation details such as matrix and tensor product representations. Also, a primary objective of this toolbox is educational: highlighting the structure of the underlying FPGA hardware will benefit the developers understanding of the tasks involved in fitting a neural network into an FPGA implemented circuit, even if later they use a language that abstracts away some of the timing and structural detail. Using a language primarily intended to allow code portability between FPGA, GPU, CPU and other parallel hardware accelerators abstracts away the differences between platforms, rather than allow the user to explore and exploit them. Having an understanding of the clock timing and the fully parallel operation of FPGA resources will help the user appreciate whether or not their solution is fully optimized for the hardware. Looking in detail at on and off chip data movement and the implication of this on caching requirements as well as the relatively simple computational algorithms, will give better understanding for all development even if the user chooses to build their deployable application using a higher level framework, or even a pre-optimized library solution.

The following code section describes a neuron in VHDL, matching the structure shown in Figure 1. Real data types are used for inputs, outputs and weights. The code shows parallel multiplication of the inputs by the weights. Signals (e.g. *products*) and entity ports hold their value and are used to communicate between parallel processes. Variables (e.g. *sum*) only have scope within the process. However the code specifies the summation of all products and the bias within the same timestep. Alternative ways of describing the parallel add are also possible. Timesteps, specified by signal *timestep* here is normally specified and implemented using a clock signal.

```
...
    type real_array is array(natural range <>) of real;
...
    entity neuron is
     generic (
       number_of_synapses : integer := 32);
     port (
       next_timestep : in boolean;
       synapses : in real_array(1 to number_of_synapses);
       output : out real);
    end entity;
     architecture behave of neuron is
       constant bias : real := BIAS_VALUE;
       constant weights : real_array(1 to number_of_synapses) := (1=>WEIGHT1, 2=>WEIGHT2 ...
       signal products : real_array(1 to number_of_synapses);

      function ReLU(x : real) return real is
       begin
         if x<0 then
           return 0;
         else
           return x;
         end if;
       end function;
     begin

       -- Parallel Instantiation of multiplier
       gen mults: for i to 1 to number_of_synapses generate
         products(i) <= weights(i) * synapses(i);
       end generate;

       p neuron: process(next_timestep)
         variable sum : real;
       begin
         -- Note, everything computed in same timestep
         if (next_timestep) then
```

```
          sum := -bias;
          for i in 1 to number_of_synapses loop
            sum :=  sum + products(i);
          end loop;
          output <= ReLU(sum);
        end if;
      end if;
    end architecture;
```

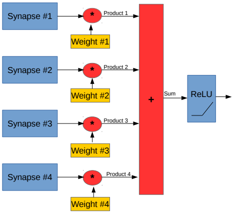

Figure 1 : Neuron Structure

VHDL (and other HDLs) are highly modular and hierarchical. To implement a neural network, a number of those parallel neurons can then be instantiated in parallel in another module. This allows the code to match the structure shown in Figure 2.

```
-- Generate Neural Network layer
gen neurons: for i to 1 to number_of_neurons generate
  neu_i: neuron
    generic map(
       number_of_synapses => number_of_synapses)
    port map(
       next_timestep => next_timestep,
       synapses => layer_inputs,
       output => layer_outputs(i));
end generate;
```

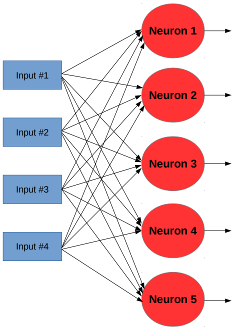

Figure 2 : Neural Network Layer Structure

The previous examples are useful to illustrates some basic properties of the VHDL language and the ease in which its descriptive power can model the structures used in neural networks. These models shown so far are however not sufficient to describe FPGA hardware and cannot be synthesized to run on an FPGA. They can however be compiled to run on a CPU, for simulation and modelling purposes. One major benefit of this modelling really comes into play when the objective is to reduce a pre-trained convolutional neural network with real (floating point) specified weights to a low energy, power efficient fixed point solution with the minimum bitwidth able to achieve the same classification accuracy. VHDL natively supports fixed point signed and unsigned arithmetic in any bitwidth, allowing the bitwidths of different parts of the neuron arithmetic to be minimized, thus allowing overall resource optimization.

For example, if the neuron uses 8 bit signed weights, and 8 bit unsigned data, the signed multiplication needs to be 8x9 bits to add a sign bit to the data. The product will then be 17 bits. Depending on the number of inputs the accumulator sum will need extra bits log2(number of inputs) to avoid potential overflow. The output may not need to have the full resolution, and can have fewer bits, but some scaling and saturation will be required to get behaviour as close to the reference floating point behaviour.

GHDL : an Open Source VHDL Compiler

Compilation of HDL code can be done for 2 purposes. Firstly, it can be synthesized to produce a hardware ASIC or FPGA design. Secondly, it can be compiled as a model of the hardware to run on a CPU based system to simulate the hardware behaviour. There are some restrictions on what parts of the language can be used in the first case. In the second case additional functionality and libraries can be used that do not have equivalent functionality in FPGA hardware such as accessing files from the OS, and printing information to the command line.

FPGA Synthesis from an HDL (or other design entry method) to produce an executable circuit is a considerably time consuming exercise as fundamentally, synthesising, placing and routing the circuit is a Travelling Salesman type Problem and, therefore, has no easy solution. Synthesis runs therefore should be avoided in the development flow until the behaviour has been verified by simulation.

Simulation of the design, effectively treats the VHDL code as a software language, and compiles it to run on a CPU. There are a large number of commercial simulation packages out there including those included with FPGA design environments. These can either interpret or compile the VHDL code. Typically these tools prioritize the logging of all the signals in the design for display, and therefore do not have the most efficient execution. There is also an Open-Source alternative simulation GHDL. This is a VHDL compiler built on top of GCC (it performs VHDL to C cross-compilation first), and as it compiles code rather than being and interpreter, it can run faster than some commercial simulation packages.

All the testbench based designs in this Alpha Data library have been tested using this package, as it provides a software like development flow : compile, link, run, analyse behaviour, fix bugs, repeat ...

The package can be downloaded from http://ghdl.free.fr. This will allow the developer to begin their exploration of FPGA based Machine Learning Inference without needing to purchase any hardware or software. Once confidence that the FPGA is a suitable choice of hardware for the specific Inference application, implementation can be achieved in many ways, re-using this code, using code in another language or using a pre-optimized Machine Learning Stack, such as those available from IBM, Xilinx and others.

A non-timed VHDL model for bit-width Exploration

In this section a non-timed VHDL model of a CNN layer is described. By using a single process and no-clock stimulus, the VHDL code is reduced in complexity to that of an ADA sequential process. This should be easily understandable by a software developer.

The code for this is located in the archive in as file *sim_only/sim_zfnet_layer0.vhd*. The simulation models the input layer of the ZFNet CNN **[1]**. Tensor types are defined for the different fixed point precision data types used in the network layer.

The understanding of the different data bit widths for different arithmetic operations is essential in minimizing energy use per computation and maximizing throughput. Coming from a floating point description, input data, weights, bias, products, accumulator values, and post rectified linear output are all represented by the same 32 bit type. This is relatively inefficient, as inputs typically are unsigned 8 bit image data or unsigned values in the range from 0 to 1. Weights are signed values, but also of limited range and precision. The product of these requires higher resolution and the accumulation of these extra range to avoid overflowing any arithmetic. Before the output, the rectified linear operation will squash the range, reducing the required resolution at that stage, and also removes the sign of the accumulated value, outputting an unsigned value to the next layer. Figure 3 shows the different bitwidths

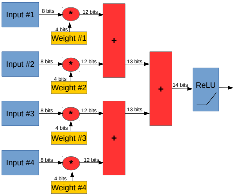

Figure 3 : Fixed Point Neuron Implementation

Therefore using a single arithmetic type throughout the neuron computation is therefore wasteful in energy. To produce an optimal solution, using fixed point arithmetic, requires identifying the minimal bit widths required for each of these variables in the arithmetic. For example, using 8 bit input image data rather than 32 bit floating point will reduce the memory bandwidth requirement, and on-chip data re-use memory cache, and input data routing requirements all by a factor of 4. Using 4 bit weights, rather than 8 bit (or 32 bit) will reduce the weight memory cache requirement, and the multiply and accumulator logic footprint (or allow more multiplys to share the fixed size multiply in the DSP tile), allowing more neuron computations to be packed into the FPGA in parallel. The example sets the bias bit width to the same width as the weights, although allowing some shifting to increase dynamic range. Optionally the bias could have the same bit width resoultion as the accumulator. Output bit resolution will depend on the requirements of the next layer, and scaling by a factor of 2 (bit shifting) before the rectified linear unit can change the gradient of the linear region, so that any significant large values generated are not incorrectly saturated.

The non-timed model allows significant exploration of the effects of changing the different bitwidths, and these are easily modified by changing constant parameters. The code is structured to read in a file of input data, representing a single image in a batch (modifying the code to handle a file containing multiple images is straight forward). The example code currently randomized the weights for all the neurons. If a pre-trained network of weights is available, this could be read in to the simulation in the same way as image data.

With the weights and data read into tensor variables, the code then runs sequentially looping across each pixel, extracting a region of interest for each filter. This is read into a region mask tensor. The region mask tensor elements are then convolved (multiply accumulated) with the weights for every neuron and the sum linearly rectified.

The post filter tensor is then passed through a max pooling loop, which reduces the output data size by selecting the maximum value over a small 2D range of pixels. This output is then written out to a file for analysis.

This code only provides an example of a single layer. The same code could be modified to simulate other layers in the network, and allow analysis of their fixed point performance.

This code in no-way provides a practical template for an FPGA implementation. It forces some sequential dependencies and data access ordering that would be best avoided in an FPGA. It does however provide a very valuable mechanism of validating choices of arithmetic data type within a reduced bit-width fixed point implementation, in a relatively easy to understand high level format.

To compile and run this code using GHDL is very simple, simply compile **-a**, link **-e** and then run the code:

```
[user@machine sim_only]$ ghdl -a sim_zfnet_layer0.vhd
[user@machine sim_only]$ ghdl -e sim_zfnet_layer0
[user@machine sim_only]$ ./sim_zfnet_layer0
```

The code reads in *input_data.txt* which is text file of integers which can be displayed as a colour image. The data is in channel (RGB), column, row order. The output data is also written out in this format, although the number of channels is 96, with channels not relating to any specific colour field.

The code can also be imported into any standard EDA simulation tool such as the XSim simulator built into Xilinx Vivado, Mentor Graphics Modelsim, or Cadence NCSIM.

A second example based on using pre-trained weights from the Caffe Model Zoo *http://caffe.berkeleyvision.org/model_zoo.html* is also provided. This layer simulation is based on the input layer of AlexNet **[2]**. Weights were exported from Caffe in integer format, scaled by 2^30 to extend the range beyond -1:1, and saved as a text file. The simulation VHDL imports these and scales them down to fit in the simulation specified bit width. Figure 4 shows the input image, and Figure 5 shows the 96 neuron outputs after layer 0.

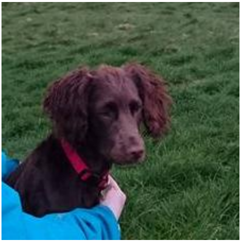

Figure 4 : Input Image

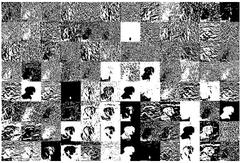

Figure 5 : Layer 0 Output

A hardware synthesizable neuron

The fully parallel neuron structure and the sequentially specified network in previous sections will not generally be useful for FPGA implementation. While a fully sequential solution will obviously not exploit the parallel nature of FPGA logic, a fully parallel structure will also be limited in scale by the number of multiplier and accumulator units available in the chip. Even with over 5500 DSP blocks available, a fully parallel network would only be able to implement about 37 of the neurons of the previously described ZFNet network layer. The throughput would be very fast, but the full network would need to be implemented across many chips.

A more practical degree of parallelization can be achieved at a 1 multiplier-accumulate unit to 1 neuron ratio. This sits in the middle of the parallelization options, as sharing a single multiplier-accumulate unit between many neurons can be considered if computational resources are more limited, as can parallelization within the neuron to increase throughput.

A key priority in producing an energy efficient implementation is minimizing the movement of the input data to the multiply unit. For example, in a Xilinx KU115 device there are 5500 DSP tiles, even assuming a conservative 16 bit operational width at 250MHz, the memory bandwidth to keep both inputs full, with every tile operating independently would be 2.75TB/s. Data re-use and caching is therefore essential. Since all neurons in the same layer use the same input data, having a layer of parallel neurons will reduce the input bandwidth requirement for that half of the computation by a factor related to the number of neurons. The other half of the input bandwidth to the multiply units will be the weights: caching of these will be required to achieve the data re-use necessary to achieve a practical input bandwidth. At the input end of a convolutional network, typically each weight value is used many times for every pixel position output as well as for every frame in the batch. For the example ZFNet input layer this works out at 12100 (110x110) re-uses per frame in the batch. Therefore for this layer, keeping the weight value as close to the multiplier as possible is desirable. For later layers in, for example, ZFNet, especially fully connected layers, this structural argument changes due to the different re-use ratio of the weight values.

In the synthesizable example code *conv_neuron.vhd* the weight memory is implemented as a dual ported memory, as FPGAs typically have a large number of these blocks, each independently addressable with around ~2kB of space. They also have smaller and larger memories available, which may also be used in some cases. However the most useful memory for the weights are the FPGA block memories of approximately 2kB. Typically in an FPGA there is one of these 2kB memory blocks for each multiplier unit.

Since in an inference model, the weights are typically written once and then the data to process is then streamed through, with a potentially very large batch size, the weight input bandwidth is low, with one weight written at a time, sent as a contiguous stream with a first and last weight strobe, with the first weight actually being used as the bias. To removed an unnecessary subtraction, or sign change, it is assumed that the negative of the bias is written into the memory, when the weights are initialized. Note that if the weights were entirely staticly pre-defined, the weight memory could in theory be implemented as a ROM ni the FPGA saving further resources, but at the cost of flexibility.

The processing is the driven by the feature inputs. This is assumed to arrive as a contiguous stream of data, with first and last features signalled independently. This allows the first element to be used to reset the accumulator to the bias, and the memory read address to access the first weight. The multiplication and accumulation can then stream through the weights and features till the final input sample is indicated by the last, and the accumulated value can be linearly rectified and output. The feature data can be continuously input into these neurons with no dead time in between. The output will be generated at this rate divided by the number of weights. Note that the output scaling in the example code is fixed to the same value for the entire layer in the example code. It may be useful to extend the dynamic range of the network, by allowing different scalings on different neurons, either statically defined, or as a run-time configuration parameter.

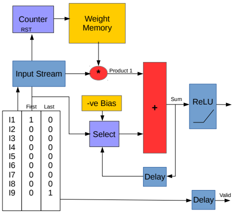

Figure 6 : Example Hardware Synthesizable Neuron Implementation

Figure 6 shows the general structure of the Multiply-Accumulate based hardware synthesizable neuron implementation.

In the example code, throughput and maximum clock rate has been traded against latency. Both Block RAM and DSP tile primitives benefit in maximum clock performance if data is registered at their inputs and outputs. Therefore the control signals, such as first and last strobes get delayed so that the accumulation and output of data are timed to best match the built in register structure of the BRAM and DSP tile primitives.

Testing the example code can be done from the conv_neuron folder, using files *conv_neuron.vhd* and *tb_conv_neuron.vhd* a testbench file that generates random data to run through the neuron model and print out the results to the command line.

```
[user@machine sim_only]$ ghdl -a conv_neuron.vhd
[user@machine sim_only]$ ghdl -a tb_conv_neuron.vhd
[user@machine sim_only]$ ghdl -e tb_conv_neuron
[user@machine sim_only]$ ./tb_conv_neuron
```

As in theory the conv_neuron block is specified to run forever, it will be necessary to use Ctrl-C to exit the simulation, or you can specify a stop time:

```
[user@machine sim_only]$ ./tb_conv_neuron --stop-time=100us
```

A hardware synthesizable neural network layer

A network layer can be constructed easily by specifying a number of the previously described neurons in parallel. All neurons each receive the same input feature data stream. For a very large number of neurons this could potentially create a routing congestion issue. The first example of a neuron layer (*conv_neuron_layer.vhd*) avoids this by fanning the data out to each neuron via a shift register. This will produce an FPGA design that can be clocked very fast and will have few routing problems. The neurons will produce outputs in sequence each delayed by 1 clock cycle. If these are then collated back into a single stream using shift registers on the outputs, the data will arrive as a burst with valid data every second clock cycle.

A limitation of this is that the number of neuron weights must be larger than twice the number of neurons in the layer. Most internal network layers will actually do this, although the input layer of ZFnet does not. A second option, with potentially lower maximum clock rate, is to use a flat fanout on the input side (*conv_neuron_layer_ffanout.vhd*). This will then produce all neuron outputs on the same clock cycle, which can be output as a contiguous stream using the shift register. This layer can have as many neurons as weights, and so can implement the ZFnet input layer.

The restriction is due to the output bandwidth, and another way of working with a higher number of neurons to number of weights ratio, is to implement this structure multiple times in parallel, with each structure implementing a subset of the neurons, and outputting streams of outputs in parallel.

```
[user@machine sim_only]$ ghdl -a ../conv_neuron/conv_neuron.vhd
[user@machine sim_only]$ ghdl -a conv_neuron_layer.vhd
[user@machine sim_only]$ ghdl -a tb_conv_neuron_layer.vhd
[user@machine sim_only]$ ghdl -e tb_conv_neuron_layer
[user@machine sim_only]$ ./tb_conv_neuron_layer
```

A hardware synthesizable convolutional neural network layer

The previous layer implements a fully connected layer. However to implement a convolutional layer requires several extra blocks to be added to the data pipeline.

With convolutional networks, the input data stream is typically for each frame in a batch, a 3D tensor, made up of 2D "images" of features (channels). For the input layer, the input is often a 2D image with 3 channels, for Red, Green and Blue colour data. For deeper layers, the channels can be more numerous and do not necessarily correspond to anything as easily describable.

Efficient reading of this image data from memory requires a rather different approach to the one simulated in the earlier section. That code specifies byte wide accesses to memory addresses, some sequential, and some on a following line. A naive implementation on an FPGA using byte wide reads on 72 bits wide ECC DDR4 DIMMS, with an internal useful data bus width of 512bits, would result in 1.6% efficiency. Since each pixel will be reused approximately 10 times, in combination this would result in lowering a 19.2GB/s memory interface down to 30MB/s, which might actually present memory bandwidth problems.

In a CPU or GPU, the built in hardware cache hierarchy would alleviate this significantly, reading cache-lines at a time, so that the chip memory bandwidth is less critical. In an FPGA there is no built in cache, and the options are to use a CPU-like general purpose cache or an application specific cache. In the case of implementing a CNN, the data access pattern is very predictable and therefore it is relatively easy to implement an application specific cache.

The input data for a frame in a batch will typically be stored in a contiguous buffer, and therefore burst reading this data from DDR4 (or even over PCIe from the host) at maximum width is required. This will create a stream of pixels than can be buffered in a local on-chip cache memory, of a size just larger than the filter mask size, to allow the required tensor mask region to be extracted for every one of the 12100 positions on the input image

The code for this cache is captured in *feature_buffer.vhd*. The cache is deep enough to hold the required number of lines, plus two, to allow a flow controlled input stream to write data into lines while the filter region data is being read from the other lines at the required rate.

Some networks and layers specify reading data into filters off the edge of the image. This zero padding, adding an outer border can be built into the read logic of the feature buffer. This may however make the code quite complex and more difficult to understand, and it may not be the best solution with regards to timing. Therefore for this example code, the zero padding is applied to the input stream before the buffer when required. This is slightly wasteful in requiring extra memory resources. As it is the read side of the buffer that controls processing rate, this will have no effect on performance.

In the example, data is output from the buffer 3 pixels at a time. However there is a gap between the end of each region read and the next. A stream narrowing buffer is placed on this output, which reduces the data width to one feature per clock cycle. This is fed into the convolution neural network layer one feature per clock cycle as a continuous stream ensuring that the multipliers are fully utilized, with an operation every clock cycle.

After the neural network layer, the data is then optionally widened to provide a more manageable data stream. Some layers are often followed by a MaxPool layer, which is a simple non-linear filtering technique, which reduces the amount of data output, by selecting the maximum value from a small region of the neuron output plane. This re-uses the zero padding and feature buffer cache modules to provide similar functionality. The maxpool filter is however significantly simpler, performing comparison operations rather than multiply accumulates.

After the maxpool stage, the data is then output as a stream that can be written back to memory, or passed to the next layer.

Data Streams and Parallelization

The previous sections have referred to data streams extensively. The reason for this is
that to properly understand the FPGA implementation, and identify if a solution is efficient, it is necessary to understand how the data moves around the chip, rather than considering the data to be a static object in memory.

Streams can have several different definitions and meanings. In the simplest case, a stream can be a signal containing data that changes every clock cycle. Adding a data valid signal, to indicate when data is present, and when the signal can be ignored adds a first and almost essential level of control. Adding first and last strobes to mark the beginning and end of a packet (object, feature, frame) of data provides more control. These signals are however of a non-flow controlled type. The logic receiving the data must always be able to accept this data. Simple arithmetic units such as the neuron multiply-accumulate units can easily be designed with this assumption. Assuming non-contiguous data, within packets, can make the logic more complex.

In some cases it is necessary to provide back pressure to streams. Data coming over PCIe or from memory (or being sent to PCIe or memory) typically falls into this case. These streams not only have valid signals, but also have ready signals to indicate ability to accept. Modules in the design such as the feature_buffer cache memory which delay data until ready are examples of logic that uses flow controlled streams. Flow controlled streams are useful in many circumstances, and there are standards such as the ARM/Xilinx AXI4-Stream protocol out there.

A third type of stream used in these example designs is a packet-level flow controlled stream. The feature buffer to stream narrowing connection is an example of this. The ready flag is only checked before the start of each packet of data. This greatly simplifies the connection logic between the two modules and avoids the need for flow control on the data on every clock cycle - which can lead to timing closure difficulties.

By packaging the data to be processed into streams, it can flow around the chip to the different modules requiring it. If non-flow controlled however the destination must always be able to accept the data. This might be an issue at a specific width, but by widening the stream, and creating parallel instances of the destination hardware, the destination should not overflow with received data.

A PCIe FPGA implementation of the CNN Layer

In this section a practical implementation of a CNN layer on an FPGA in a deployable PCIe form factor board is demonstrated. One major key to practical FPGA deployment is handling the input and output of data, from storage, the network or even just from an application running on the host CPU in an efficient manner. This example uses a simple streaming approach to DMA data at an efficient rate across the PCIe bus and back. The streaming approach generally works best with large contiguous data sets, which fortunately matches the requirements for machine learning inference, processing batches of images. Therefore this approach is used in place of more complex memory mapped structures, which offer more flexibility at the cost of complexity.

Figure 7 shows the top level structure of the design. The design is targeted at the Alpha Data ADM-PCIe-8K5 board. A board specific PCIe IP core is used to interface between PCIe and the user design. This core can be configured to provide and consume AXI4 DMA streams, of width 256bits at a clock rate of 250MHz in response to host API functions. Channel 0 is used to provide the input data. This is width reduced to 3 bytes wide to match the input width of the feature buffer cache in the Layer 0 module. The weights are initialized using a stream on a different DMA channel. A stream narrowing module specific to the task of initializing weights is used. This module will buffer the stream data and burst contiguously for each neuron's weights, and generate the first and last strobes, to match the expected input behaviour. This narrow stream structure, gives a low bandwidth means of sending our the weight data to all neurons on the chip without requiring excessive FPGA fabric routing.

The layer output data is sent back over PCIe using another DMA channel. A memory mapped direct slave port is used via some off-the-shelf Xilinx AXI4 width and protocol conversion IP cores to access a bank of registers which the software can read to report on performance counter values.

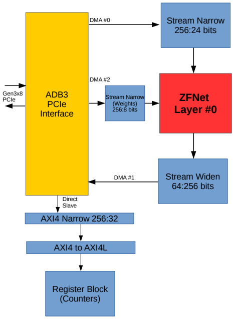

Figure 7 : Example PCIe FPGA Implementation

To build the example requires Xilinx Vivado Synthesis and Implementation tools. Version 2017.1 is the default target, but older versions may be used. With older version it may be necessary to modify the **use_dsp** attribute to **use_dsp48** in the conv_neuron VHDL code. The example also requires the file ip_repo/ adb3_admpcie8k5_x8_axi4_v1_2.zip which is available to ADM-PCIE-8K5 customers when purchasing the RD-8K5 support package. This file must be copied into the ip_repo location before running the project build script.

The project is built using the TCL script: *build_cnn_fpga_pcie.tcl*. This can be run using the command line mode of Vivado.

```
vivado -mode batch -source build_cnn_fpg_pcie.tcl
```

This will create a project and open the GUI. The GUI can then be used to run synthesis and implementation to create a bitstream for the card

In the source code, some optimizations are explored. With Xilinx devices it is easy to fit two 8 bit multiply operations into a single DSP48 tile **[3]**. This however assumes that one of the inputs is the same for both multiplications: **P1+2^16*P2 = A*B1+A*B2*2^16**

This situation does arise in neural network layers as the same data is passed to every neuron, and therefore the simplest way to exploit this is to share the multiplication (DSP tile) between two neighbouring neurons. The files *conv_neuron_shmu.vhd* and *conv_neuron_layer_ffanout_shmu.vhd* implement this optimization, halving the number of multipliers required. This optimization does not necessarily lead to better accumulator use, however additions can also be implemented in fabric as they are less performance critical.

Table 1 shows the resource utilization of the single layer.

Resource	Utilization	Available	Utilization %
LUT	36290	663360	5.47
LUTRAM	6051	293760	2.06
FF	64557	1326720	4.87
BRAM	169.5	2160	7.85
DSP	96	5520	1.74
IO	48	624	7.69
GT	8	48	16.67
BUFG	8	1248	0.64
MMCM	1	24	4.17
PCIe	1	6	16.67

Table 1 : ZFNet Layer 0 Implementation in KU115

The estimated chip power is 7.24W.

These results show a number of things. Firstly that the 96 neurons, while only using 48 shared multiplications, still use 96 independent accumulations. However if multiplication was in short supply, the additions could be implemented in LUTs with almost the same performance. Another major insight is that the layer is using only a small fraction of the resources available in the KU115 chip. This means that there is scope for fitting the full network in this device, or fitting the network layer in a smaller low power device, which might possibly be connected direct to a camera input stream. The ratio of DSPs to BRAM is ~1.75% to ~8% which implies that if scaled up, BRAM resource would be the factor that limits the size. Restructuring the layer to have a higher DSP to BRAM (Multiplier to Weight) ratio is therefore desirable in this case.

CAPI FPGA implementations of the CNN Layer

In this section, the targeting of the same CNN layer into a CAPI based FPGA design is discussed. CAPI is a protocol in IBM Power8 servers that runs over the physical PCIe bus but provides far higher levels of memory coherence and security. It allows the FPGA to access the host applications user space memory as if it were a CPU, using the same user space virtual addresses allowing host memory access with PCIe driver overhead, and possibly most critically a level of security not available in a basic PCIe design which in theory can access any physical host address and potentially subvert a system. CAPI also allow full reconfiguration of the Therefore CAPI and Power 8 deployment has significant security advantages in cloud FPGA deployment.

To port the existing PCIe design to CAPI, the design uses the same stream based design. A top level, off-the-shelf AFU template design which converts from the Native CAPI PSL interfaces to AXI-Stream interfaces is used to provide a similar DMA like interface to the user code. The bulk data streams are by default 512 bits wide in CAPI and therefore slightly different stream conversion logic is required, however the central CNN core can be used unmodified.

Building this design require the user to have purchased a CAPI Developer Kit with an ADM-PCIE-KU3 or ADM-PCIE-8K5 CAPI card. The archive contains all the source code to build an afu in the folder *xilinx_fpga_capi/Sources/afu* with the project file in *xilinx_fpga_capi/Sources/prj*. These files can be plugged into the CAPI build directories and Vivado run with the usual TCL build scripts to generate a deployable CAPI bitstream.

The same design can also be re-targetted at the recently launched CAPI-SNAP framework. Some example action files are provided in the *SNAP* folder that can be used to encapsulate the ZFNet layer in a CAPI SNAP action. A similar 512 bit wide streaming interface is used to transfers the data and weights to and from host memory.

An optimized PCIe FPGA implementation of the multiple CNN Layers

In this section a more optimized implementation of the full ZFNet network is considered. Before going anywhere near coding some spreadsheet analysis of the network should be done. A number of spreadsheets for different CNNs are contained in the folder *spreadsheets*. The main parameters of ZFNet are listed in Table 2 below.

Eight layers are considered in this network, 5 Convolutional Layers and 3 Fully Connected layers. One of the main purposes of the spreadsheet is to calculate the number of times each data element (input or weight) is required to be used in a computation, as well as to calculate how to balance the computational requirements of the different network layers, as the maximise throughput if layers operate in parallel, their performance must be matched.

The input sizes depend on the actual input sizes (zero padded), however the multiplications actually depend on the "effective" rows and columns values, which takes into account the stride and mask size to return the actual number of input points used for each neuron computation, per frame.

The input pixel re-use ratio identifies the level of benefit gained by caching input pixels for convolutional filtering. The actual re-use of these pixels is this multiplied by he number of neurons. This gives an indication of the input data bandwidth relative to layers multiplication performance, which needs to be balanced at a practical level.

The multiplications per neuron and per layer give an indication of the performance required for each layer. If it is desired to feed one layer into the next in a pipeline, these performances must be match by increasing or decreasing the parallelization of the layers. If the inter-layer data are to be stored in external memory between layers, and layers ran sequentially, switching weight values, (an alternative implementation strategy, less power efficient as it requires data and weight values to be buffered off-chip) then this balancing is less important.

Layer	0	1	2	3	4	5	6	7
Type	Convolution Layers					FC	FC	FC Softmax
Input Rows	224	55	13	13	13	1	1	1
Input Cols	224	55	13	13	13	1	1	1
ZP Rows	227	55	15	15	15	1	1	1
ZP Cols	227	55	15	15	15	1	1	1
Skip	2	2	1	1	1	1	1	1
Mask Rows	7	5	3	3	3	1	1	1
Mask Size	7	5	3	3	3	1	1	1
Channels	3	96	256	384	384	9216	4096	4096
Input Size	150528	290400	43264	64896	64896	9216	4096	4096
Neurons	96	256	384	384	256	4096	4096	1000
Effective Rows	110	26	13	13	13	1	1	1
Effective Cols	110	26	13	13	13	1	1	1
Mults per neuron-per pixel	147	2400	2304	3456	3456	9216	4096	4096
Mults per neuron	1778700	1622400	389376	584064	584064	9216	4096	4096
kMults for layer	170755	415334	149520	224280	149520	37748	16777	4096
Input Pixel re-use ratio	12.25	6.25	9	9	9	1	1	1
Layer Performance Requirement	14.62%	35.56%	12.80%	19.20%	12.80%	3.23%	1.44%	0.35%
Weight re-use per frame	12100	676	169	169	169	1	1	1
Weight Memory Size (kB)	14	614	884	1327	884	37748	16777	4096

Table 2 : ZFNet Resource User Spreadsheet

The weight memory and memory bandwidth requirements are also calculated. The high weight re-use per frame values for the convolutional networks point to the major advantages in storing these values on-chip. The relatively small memory footprint of these parameters in the convolutional layers allows them to fit fully in internal memory for the computation.

Total weight memory size for the Convolutional Network Layers will be 3.7MB (assuming 8 bit coefficients, 1 byte per coefficient). The total weight memory size for the fully connected layers will be 58MB.

This much larger memory requirement for the fully connected layers, indicates that the convolutional layer structure and code cannot be simply re-used. Looking at the layer performance requirements, the fully connected layers only account for 5% of the computational load. Therefore it makes a lot of sense to treat these layers separately, and design a different circuit, possibly running on a different FPGA to process these. This task could also be handled by the host system CPU, if it had few other workloads. The memory bandwidth for the data output at this stage is relatively low and so transferring it to a different resource, or storing it in external RAM for later batch processing will be relatively inexpensive in energy.

The previous table shows the general network structure as defined by the user who specified and trained the original network. Many of the parameters such as layer input sizes, numbers of neurons and weight memory size will be fixed. However the layer structure and parallelization can be optimized to allow data flow through with the best utilization.

The following table shows the part of the spreadsheet that allows exploration of parallelization. By increasing the parallelization factor it is possible to increase the DSP to Weight ratio, with the aim of using as many DSPs as are available in the device. The number of clock cycles to push a frame through the layer is calculated. As all layers are implemented as a pipeline, the slowest layer will determine the throughput. Using different parallelization factors at different layers can by specified to attempt to utilize the DSP with the highest load factor possible, by attempting to get the throughput number of cycles for all layers as close as possible.

Layer	0	1	2	3	4
Parallelization Factor	4	4	1	2	2
DSPs Required	384	1024	384	768	512
Cycles	447700	405600	389376	292032	292032
Utilization	100%	90.60%	86.97%	65.23%	65.23%
DSP Utilization	384	927.7	333.9	500.9	333.9

Table 3 : ZFNet Implementation

Total DSP Hardware Requirement for this configuration is 3072 DSP multipliers. These will be utilized at an efficiency of 80%. If the clock rate is assumed to be 250MHz, the network will be able to process at a rate of 80MB/s and accept a new frame every 1.79ms. This is a fairly conservative utilization for a KU115 device, using only just over half the DSP tiles, clocking the at a relatively low rate, and not exploiting the technique of 2 neurons sharing a single multiplier block, each of which might allow doubling of performance. The design does however place and route relatively quickly, which is useful in understanding what is possible.

The project is built using the TCL script: *build_cnn_fpga_pcie.tcl*. This can be run using the command line mode of Vivado.

```
vivado -mode batch -source build_zfnet_fpga_pcie_opt.tcl
```

This will create a project and open the GUI. The GUI can then be used to run synthesis and implementation to create a bitstream for the card

Implementation results are shown in the following table. The number of DSP tiles is higher than estimated. This is due to the extra addition circuits used in the par2 and par4 neurons. These additions could be performed in slice logic to save resources as LUTs are not extensively used. Alternatively, they could be shared between all neurons in the layer along with the ReLU circuit, which would save addition logic/DSP resources, but increase the routing and register use in the layer output shift register.

Resource	Utilization	Available	Utilization %
LUT	128784	663360	19.41
LUTRAM	6825	293760	2.32
FF	267608	1326720	20.17
BRAM	1228.5	2160	56.87
DSP	3760	5520	68.12
IO	48	624	7.69
GT	8	48	16.67
BUFG	8	1248	0.64
MMCM	1	24	4.17

Table 4 : ZFNet Implementation in KU115 (continued on next page)

PCIe	1	6	16.67

Table 4 : ZFNet Implementation in KU115

The estimated power consumption was indicated as 29W.

This network only implemented the 5 convolutional layers. To implement the fully connected layers in the same FPGA, the weights for these layers would need to be stored off-chip, probably in DDR4 memory. While this has fairly high bandwidth on the ADM-PCIE-8K5 board 2 banks at 19.2 GB/s loading this data in every frame would require 30GB/s, so while just possible, would be quite power intensive. A lower bandwidth approach would be to batch up frames for processing. Each layer 5 frame would require 9kB of input memory, and about 8kB of accumulator memory. Layer 6 would require 4kB of input memory and 8kB of accumulator, and Layer 7 4kB of input and 2kB of accumulator memory, resulting in approximately 35kB of memory per frame (about 8x 36kB BRAMs). Therefore batching up to process 32 frames per batch would require around 256 BRAMs, but would reduce the DDR4 bandwidth requirement down to less than 1GB/s.

Spreadsheets are also provided for some other common networks. Using the same general structures and targeting the KU115 device on and ADM-PCIE-8K5 board these indicate that throughput time per frame for AlexNet would be around 584us for a conservative 250MHz clock, with no multiplier sharing. VGG Net would need to be split across 2 FPGAs, due to a 14MB on-chip memory requirement, to run fully pipelined in parallel, but could achieve a throughput time of 5.4ms. The basic CIFAR10 Tensor Flow example with 2 convolutional layers can fit easily in the chip with a very high degree of parallelization, and there is space to keep the fully connected layers weights on-chip.

Summary

This white paper documents an Open Source resource for evaluating the use of FPGA technology in Machine Learning Inference deployment. Open Source code, compatible with open source tools can be used for structural and arithmetic evaluation of pre-trained networks to estimate their suitability for FPGA deployment. One primary objective is to provide an easy straight forward CPU compilable method of testing the effect of fixed precision arithmetic on network accuracy, allowing each layer to be optimzied to the desired precision. Once a fixed precision implementation size and structure has been chosen, example code for implementing this is provided, however this can be used for comparative reference with other available FPGA CNN library solutions. Examples of these networks targeting real FPGA hardware are given, to allow Alpha Data FPGA customers to optimize layers as required. Possibly the most powerful tool provided is not the code, but the example analysis spreadsheets, which break down the networks performance and memory requirements to estimate appropriate FPGA resource use. Although these can be used with the example HDL Open Source code, they can also provide useful indication of potential performance on the hardware using 3rd party CNN libraries. They can be particularly useful in comparing different FPGA devices to determine the suitability or each device for deployment.

References

This section provides supplimental information for this document.

[1] Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," *Lecture Notes in Computer Science*, vol. 8689, pp. 818–833, 2014, Springer

[2] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25*, pp. 1106–1114, 2012

[3] Yao Fu, Ephrem Wu, Varun Santhase elan, Kristof Denolf, Kamran Khan, and Vinod Kathail, *WP490: Embedded Vision with INT8 Optimization on Xilinx Devices,* Xilinx, April 19, 2017

Revision History

Date	Revision	Nature of Change
19/05/17	1.0	First release

Page Intentionally left blank