Space Tolerant CNN FPGA Deployment
Part 1: An Open Source DPU Layer

A. C. McCormick

Space Tolerant CNN FPGA Deployment, Part 1

This paper is the first part of a four part series of white papers providing an educational overview of the issues surrounding the deployment of Convolutional Neural Network solutions on FPGAs in a radiation susceptible environment. This paper builds on the Open Source code documented in our previous white paper **[1]**, and it is recommended that that is consulted for an introduction to Convolutional Neural Network concepts in the context of FPGAs. This first part documents a practical CNN processing core, which can be used to implement a wide range of CNN solutions. The second part, will discuss the Space Hardening of this core, adding in Triple Mode Redundancy for radiation effect tolerance to control path circuitry. The third part will document the higher level control structures necessary to move data to and from the CNN core and dynamically reconfigure its operation to match the functional requirements of a part of a network. The fourth part of this series will document the deployment of this FPGA solution on the Alpha Data ADA-SDEV-KIT3 Space Development kit for the Xilinx XQRKU060 FPGA device.

An open Source DPU Layer

Our previous white paper covered the full implementation of multi-layer neural networks within FPGAs as this provides the lowest latency and is the most memory bandwidth efficient approach. However a quick analysis of the memory footprint of many popular CNN based AI models reveals that except for a small number of mobile application focussed solutions, these cannot fit in their entirety into a single Space deployable FPGA. The most capable of these, the Xilinx XQRKU060, although 5-10x the size and performance of previous generations of Space deployable devices is still only around 20% of the size of the top end data centre deployable FPGAs. Therefore a solution must be deployed that allows the AI model representation to be moved in and out of the FPGA dynamically allowing the same logic to implement the different layers of the network in a time division multiplexed order. A single layer DPU (data processing unit) which sequentially loads the configuration for each layer and then processes the data for that layer (possibly batching a number of frames for efficiency) is therefore an appropriate option to deploy here.

In this part of the paper, the open source code for a versatile single layer DPU will be described, including the key components and system structure. Accurate simulation is essential for reliable FPGA deployment, and a significant proportion of this section will be devoted to the test bench structure, and especially to the functions which allow the configuration of the DPU to be driven by Caffe files from a model library such as those in the Xilinx Vitis AI Model Zoo. The choice of DPU size is a tricky balance. Too large a layer may be inefficient for CNN layers with lower numbers of neurons. Too small a layer will not allow a sufficient number of weights to be cached in the layer to allow non-memory bound efficient processing. However as the DPU must implement many different sized layers on the same hardware a compromise size must be selected.

Structure of the Open Source DPU

Figure 1 shows the general structure of the DPU core. This core processing unit has at its heart an array of neurons which can be dynamically configured with weights and biases. The ReLU non-linearity can be optionally applied to this cores output data stream. On its own, this neuron layer can implement a 1x1 convolutional layer or a fully connected layer. For most image based CNN processing, the input data stream consists of a sequence of pixels, with a number of features per pixel. Each neuron computes the scalar product of these features with each neuron weight to generate a new feature per neuron for that pixel in the next layer. There may be a non-linear transformation applied to this product. The number of active neurons in the layer can be selected dynamically, as can the number of weights pre neuron, stored in local memory within the core. This allows the layer to implement a variety of different CNN layer structures. It will operate most efficiently when all neurons are in use, and can support larger layers through reconfiguration with new coefficients and re-sending the previous layer data.

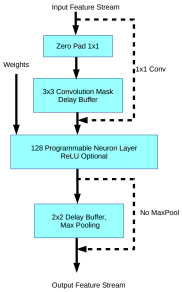

Input Feature Stream
Zero Pad 1x1
1x1 Conv
Weights
3x3 Convolution Mask Delay Buffer
128 Programmable Neuron Layer ReLU Optional
2x2 Delay Buffer, Max Pooling
No MaxPool
Output Feature Stream

Figure 1 : Fixed Point Neuron Implementation

Surrounding this core layer, there are some optional transforms which can be enabled in the data flow path. On the input side there are some transforms to support 3x3 convolution. In comparison to the previous open source library code, this feature buffer block only supports 3x3 convoluton and is not configurable for larger masks, as these appear to have fallen out of fashion with newer CNN models. The feature buffer is however now dynamically programmable and can support run-time configurable changes in image size and number of features. A helper zero padding block adds a surrounding layer of zeros to make the output image size match the input image size if required. The buffer also supports an optional stride of 2 for layers that require that to be implemented without the max pool operation.

Also before the CNN core layer, there is a rate control block, which limits the input data rate if it is going to cause an overflow in the processing. The neuron output rate is currently limited to just 1 neuron output every 2 clock cycles, which helps keep the circuit efficient for pipelining and maximising FPGA clock rate, but however might not be optimal for all CNN configurations. With layers with a lot of features, especially with a 3x3 mask, the input rate will always be the bottleneck, but for some early layers the CNN core might have to operate sub-optimally due to this data rate limitation. If this was observed to be a major limitation on processing efficiency, system wide, the core could use a more parallel implementation of its output logic to work round the bottleneck, however the serial stream output is kept here for now for code readability.

On the output of the core CNN layer is an optional max pooling operation. This selects the maximum value from each 2x2 pixel block in the image, reducing the resolution of the output layer by a factor of 2 in each dimension. Again the stride and window size is fixed to 2x2, but the image width and number of features is dynamically programmable.

DPU Size Choices for YoloV3 DK Tiny Implementation

To test out the capabilities of this DPU layer, a small model from the Xilinx Vitis AI Model Zoo was chosen: dk_tiny-yolov3_416_416. This model is used to provide the example layer structure, and weights and biases for the simulation. This model appears to be a cut down version of the YoloV3 model, but with only outputs at 2 scales used for object classification and bounding box description, with the Layers identified as 14 and 21 containing the output information.

This model consists of 13 CNN layers as shown in table 1.

Layer	Neurons	Conv	Input	Input Size	Output Size	ReLU	Maxpool
Layer 0/1	16	3x3	input	416x416x3	208x208x16	Y	2x2
Layer 2/3	32	3x3	L0/1	208x208x16	104x104x32	Y	2x2
Layer 4/5	64	3x3	L2/3	104x104x32	52x52x64	Y	2x2
Layer 6/7	128	3x3	L4/5	52x52x64	26x26x128	Y	2x2
Layer 8/9	256	3x3	L6/7	26x26x128	13x13x256*	Y	2x2*
Layer 10	512	3x3	L8/9	13x13x256	13x13x512	Y	N
Layer 11	1024	3x3	L10	13x13x512	13x13x1024	Y	N
Layer 12	256	1x1	L11	13x13x1024	13x13x256	Y	N
Layer 13	512	3x3	L12	13x13x256	13x13x512	Y	N
Layer 14	45	1x1	L13	13x13x512	13x13x45	N	N
Layer 17	128	1x1	L12	13x13x256	13x13x128	Y	N
Layer 20	256	3x3	L8+2x2xL17	26x26x384	26x26x256	Y	N
Layer 21	45	1x1	L20	26x26x256	26x26x45	N	N

Table 1 : DK Tiny Yolo V3 Model Layers

Note that in the table, the Caffe layer descriptions for weights, batch normalization, scaling, biases, strides and max pooling are all merged into single layers. This does create a complication at the output of layer 8 which is used pre-max pooling by layer 20 as well as post-max pooling used by layer 10. However ignoring this issue, which will be dealt with in the 3rd paper of this series, we can map the model to our dynamically configurable DPU as follows:

Layer	Seq Runs	Neurons per Run	Neuron Input Size	Efficiency	MACs
Layer 0/1	1	16	27	10.5%	74M
Layer 2/3	1	32	144	25%	199M
Layer 4/5	1	64	288	50%	199M
Layer 6/7	1	128	576	100%	199M
Layer 8/9	2	128	1152	100%	199M
Layer 10	4	128	2304	100%	199M
Layer 11	8	128	4608	100%	797M
Layer 12	2	128	1024	100%	443M
Layer 13	4	128	2304	100%	199M
Layer 14	1	45	512	35%	4M
Layer 17	1	128	256	100%	5.5M
Layer 20	2	128	3456	100%	598M
Layer 21	1	45	256	35%	7.8M

Table 2 : Model Layer Mappings

Table 2 shows how each layer in the CNN is mapped to the 128 neuron layer. For layers with less than 128 neurons, not all are used, lowering the efficiency of the hardware use for those layers. The number of features input to each neuron (this is either previous layer number of neurons, or 9 times this for the 3x3 convolutional layers) also affects the efficiency although only in the case of the first layer for this model. For layers larger than 128 neurons, sequentially re-running the input data through of 128 neurons is the approach used here allowing the DPU to support larger CNN layers. The choice of a 128 neuron DPU is somewhat arbitrary, but does fit fairly well here with most layers operating efficiently. Larger DPUs could be used, to improve parallization and increase performance, but may be of lower efficiency for some layers in the network. Smaller DPUs may be more efficient but have lower throughput potential. It may be possible to implement multiple DPU cores within the same FPGA, and possibly a heterogeneous selection might allow for a more optimal solution for a particular network. However even if 100% MAC efficiency can be achieved, the improvement will only be a a speed up of 1.46 as currently it takes the time for ~4571M MACs to process the ~3213M MACs required.

The 100% efficiency of neuron DSP use, only applies when processing. The configuration of the DPU layer by uploading the weights will reduce this significantly. The impact of this down time can be reduced by batch processing a number of frames, before reconfiguring the DPU, however as with any batch processing, this efficiency gain is traded off for worse latency.

There are also a couple of simple operations that are not covered by the open source DPU at this stage and these suggest features that could be added to enhance its capability. The first is the need for both the pre- and post- maxpooling outputs of layer 8 to be used by later layers. This would be a simple fix to the code, allowing 2 output streams from the core to be sent to memory while the DPU runs this layer. This change is planned for part 3 of this paper series, and so for now the simulation code actually runs layer 8 twice (one with and once without the maxpooling option enabled.)

Another operation not covered is the rescaling of layer 17 output from a 13x13 size up to a 26x26 size so that it can be processed by layer 20 along with the layer 8 output. This operation is not currently included in the DPU, however a single line buffer should be able to repeat the data, then repeat the line to rescale the data up. In the larger layers which require multiple sequential runs to complete, the output data needs merged. In the simulation this is achieved by file manipulation, however in the FPGA implementation it is expected that the writes to memory will be handled by a DMA engine, and this should be capable of striped writes which will write the output for the 128 active neurons and skip over memory addresses the remaining neuron outputs for that layer.

Simulation of DPU using YoloV3 DK Tiny Caffe Model

The simulation of this DPU code is performed using a VHDL testbench which operates at a level of simulating the surrounding FPGA infrastructure by provides data into the DPU core and recording the output of the DPU core by means of flow controlled streams. In the real FPGA simulation these streams would be generated or received by DMA engines handling the reads and writes to off-chip memory. However the details of how this works is not included in this level of the simulation. Instead the source streams are read from files and the output streams are written to files. The simulation can run across many layers in sequence, with the output files from one layer being used by later layers as the input stream.

Configuration of the DPU is set dynamically in the file. This is driven directly by testbench signals. For the FPGA implementation, these signals could be set by an internal CPU, and external CPU, or poissibly by a custom state machine, which runs a program of DPU configurations. Once the dynamic signals are set, the weights need to be uploaded to the DPU internal memory. This is also driven by a data stream, however in this case, the data is read from a text version of the Caffe model. Fortunately in the Xilinx VitisAI Github repo is a version of Caffe which allows easy export of Caffe model parameters in text format. The *tools/binary_to_text* utility with the *-remove_blobs* option enabled will convert any Caffe model file into a text description easily parsed by the VHDL testbench functions.

A VHDL library is provided with sub-programs capable of reading the exported Caffe model and outputting the data as a stream of weights and biases for use by the DPU. The Caffe model has a weights layer, a batch normalization layer and a scaling and biasing layer. For training the batch normalization layer values are based on the batch processed. However for inference, these values are fixed from the output of the training. These 3 linear transforms can then mathematically collapse down to a single layer or weights and biases. The function *read_xcaffe_file* in *cnn_tools_pkg.vhd* reads in the specified layer blobs, combines any layers and outputs a stream of weights to configure the DPU to implement that layer.

The code is provided in the *onelayerdpu_v1_0_0.zip* file. This includes the VHDL source code as well as a *data* and *prj* folders.

The main folder contains the files:

```
cnn_tools_pkg.vhd
dpu_core_tb.vhd
  dpu_core.vhd
    zero_pad_dynamic.vhd
    feature_buffer_dynamic_3x3.vhd
    conv_neuron_layer_drl.vhd
      conv_neuron_drl.vhd
    maxpool22_dynamic.vhd
      prog_length_sr.vhd
```

Note that the *cnn_tools_pkg.vhd* defines the data and weight aritmetic bit widths, which are set by default to 8 bits for the data and 16 bits for the weights resulting a DPU built with multipliers that form the 24 bit product of 8 bit data and 16 bit weights.

Packed in the *data* folder is the source data required for the simulation:

tmp_layer0-conv_filter.txt	Simple Weight Only Text File for non-XCaffe simulation
input_data416x416.txt	Arbitrary 416x416x3 image - not related to DK Tiny training data
dk_tiny-yolov3_416_416_5.txt	DK Tiny Caffe Model in Text Format : 19MB file

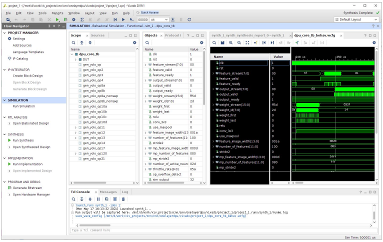
Table 3 : Data Files

Packed in the *prj* folder is a tcl script for generating a Vivado project for simulation and modification: *mkxpr-1ldpu-cnn.tcl*

This should be called from Vivado TCL command line, or used when launching Vivado from the command line:

```
> vivado  -source mkxpr-1ldpu-cnn.tcl
```

This will create a Vivado project with copies of all the source files, which can then be edited within Vivado without affecting the top level source. The GUI is launched and to start the simulation, it is simply a questions of selecting run simulation. As supplied the code will run all layers (except for the layer 0 sim that does not use the Xilinx Caffe model as that clashes with the other layer 0 sim). This simulation could take many hours to run, and for better understading, it is better to run each section at a time by setting the enable constant in *dpu_core_tb.vhd* to *true* or *false* as required. Note that although the input files for the simulation are stored and read from the data directory, layer outputs, which become layer inputs for the next layers are stored locally in the Vivado project simulation directory.

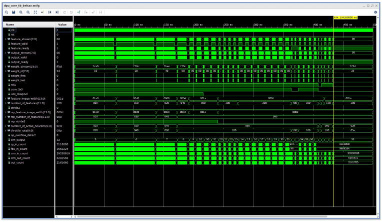


Name	Value
clk	1
rst	0
feature_stream[7:0]	00
feature_valid	0
feature_ready	1
output_stream[7:0]	00
output_valid	0
output_ready	1
weight_stream[15:0]	ff5d
weight_id[7:0]	2d
weight_first	0
weight_last	0
relu	0
conv_3x3	0
use_maxpool	0
feature_image_width[13	001a
number_of_features[11:	100
stride2	0
mp_feature_image_widtl	000d
mp_number_of_features	080
mp_stride2	0
number_of_active_neurc	02d
throttle_rate[9:0]	05a
op_overflow_detect	0
sim output	32

```
launch_runs synth_1 -jobs 2
[Mon May 17 16:13:32 2021] Launched synth_1...
Run output will be captured here: /mnt/d/work/viv_projects/cnn/cnn/onelayerdpu/vivado/project_1/project_1.runs/synth_1/runme.log
save_wave_config {/mnt/d/work/viv_projects/cnn/cnn/onelayerdpu/vivado/project_1/dpu_core_tb_behav.wcfg}
```

Sim Time: 500001 us

Figure 2 : Vivado GUI View of the Simulation


dpu_core_tb_behav.wcfg

Name	Value
clk	1
rst	0
feature_stream[7:0]	00
feature_valid	1
feature_ready	1
output_stream[7:0]	00
output_valid	0
output_ready	1
weight_stream[15:0]	ff5d
weight_id[7:0]	2d
weight_first	0
weight_last	0
relu	0
conv_3x3	0
use_maxpool	0
feature_image_width[13:0]	001a
number_of_features[11:0]	100
stride2	0
mp_feature_image_width[13:0]	000d
mp_number_of_features[11:0]	080
mp_stride2	0
number_of_active_neurons[9:0]	02d
throttle_rate[9:0]	05a
op_overflow_detect	0
sim_output	32
zp_in_count	3118080
fb0_in_count	3563224
cnn_in_count	29238919
cnn_out_count	6201566
out_count	2161660

Figure 3 : Full view of the Simulation Waveform

Outputs of each layer can be visualized and analyzed in tools such as **Matlab** or **Octave**, with the following code:

```
>>x=textread('output_data2.txt');
>>q=zeros(4*208,4*208);
>>for k=1:4;
>>   for l=1:4;
>>     q(k*208-207:K*208,l*208-207:l*208) = reshape(x((k-1)*4+l:16:end),208,208);
>>   end;
>>end
>>imagesc(q');
```

This code displays the 16 layer 0 neuron outputs in a 4x4 array, giving an indication of how the 1st stage of the processing is operating as shown in figure 4. The input image is not related to the training set used to train the weights extracted from the Caffe model, therefore it does not produce a distinctive result in layers 14 and 21 that would indicate a known object and its bounding box. Unfortunately this limits the amount of evaluation it can be used for with regards to the choice of weight and bias scaling and bitwidths selected.

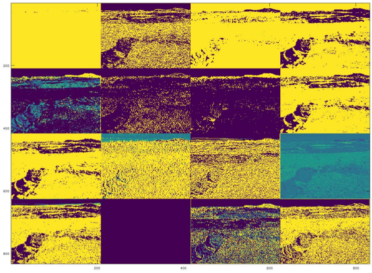
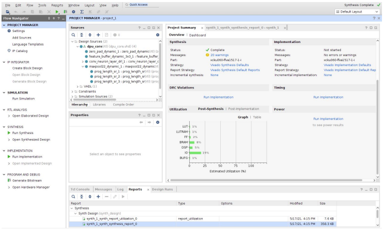


Figure 4 : Layer 0 Simulation Output

The complete simulation of all layers takes around 430ms to run, using the chosen 100MHz clock rate. This has been chosen to make the simulation easier to read. For real deployment on a KU060 device the expected clock rate could be between 200MHz and 400MHz depending on how easy it is to place the design. The reconfiguration of the DPU, and re-writing of the layer weights in simulation has been set to take longer than strictly required however minimising this will only improve performance by a few percent, and it is likely that the core could operate at best at around 9 frames per second. Increasing the number of neurons in the DPU or running multiple DPUs in parallel is an option to further increase throughput. Running synthesis, gives an estimate of the FPGA resources required to implement this core. This indicates that the DPU requires around 8% of the KU060 block RAM and 5% of the DSP tiles, and so a 10-12 times rate speed up might be expected by using multiple DPUs, although this may depend on what other functions the FPGA is required to perform. The resource requirements will depend on the bit widths selected for the arithmetic, and the accuracy of the network can be traded off with selecting lower bitwidths allowing more DPU performance per FPGA. However investigating different bit widths is beyond the scope of this paper.



```
Design Sources (2)
  dpu_core(rtl) (dpu_core.vhd) (4)
    zero_pad_dynamic_1 : zero_pad_dynamic(rtl) (z
    feature_buffer_dynamic_3x3_1 : feature_buffer_
    conv_neuron_layer_drl_1 : conv_neuron_layer_c
    maxpool22_dynamic_1 : maxpool22_dynamic(rt
      prog_length_sr_1 : prog_length_sr(rtl) (proc
      prog_length_sr_2 : prog_length_sr(rtl) (proc
      prog_length_sr_3 : prog_length_sr(rtl) (proc
  VHDL (1)
Constraints
Simulation Sources (2)
```

Synthesis

Status:	✓ Complete
Messages:	20 warnings
Part:	xcku060-ffva1517-1-i
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

Implementation

Status:	Not started
Messages:	No errors or warnings
Part:	xcku060-ffva1517-1-i
Strategy:	Vivado Implementation Defaults
Report Strategy:	Vivado Implementation Default Rep
Incremental implementation:	None

Utilization (Post-Synthesis), Estimated Utilization (%)

LUT	1%
LUTRAM	1%
FF	2%
BRAM	8%
DSP	5%
IO	19%
BUFG	1%

Report	Type	Options	Modified	Size
Synthesis				
Synth Design (synth_design)				
synth_1_synth_report_utilization_0	report_utilization		5/17/21, 4:15 PM	7.6 KB
synth_1_synth_synthesis_report_0			5/17/21, 4:15 PM	358.3 KB

Figure 5 : Vivado Synthesis of DPU

Conclusions and Next Steps

The purpose of this paper was to introduce an Open Source DPU code suitable for implementation as part of a Space FPGA solution using the Xilinx XQRKU060 device. This basic core has been described and the simulation of it explained, including configuration using an off-the-shelf Caffe model from the Xilinx Vitis-AI Model Zoo. This is however just the starting point for further investigation of this core, and its suitability (and the suitability of FPGA based DPUs in general) for deployment in high radiation environments such as space. The next paper in this series will cover the introduction of radiation effects mitigation techniques into the core, primarily covering triple mode redundancy on control paths within the DPU.

References

This section provides supplimental information for this document.

[1] A. C. McCormick *AD-AN-0055: An Open Source FPGA CNN Library,* Alpha Data, May 19, 2017

Revision History

Date	Revision	Nature of Change
26/04/21	1.0	First draft