Mentor®
A Siemens Business

Eyeriss Architecture PE Array
With Catapult Synthesis

February 2019

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com /doc_feedback_form

Introduction

Two of the most important algorithms in image processing/CNN are Convolution and Max-pooling. Convolution is used to filter/activate specific feature in an image. While max-pooling is used to down-sample the image spatially. This document shows an implementation of a processing element array that can compute the convolution using an architecture called Eyeriss.

Toolkit Contents

This toolkit contains the following files:

- ***Eyeriss_PE_Array.pdf*** – This document
- ***directives.tcl*** – This file is a Catapult script that synthesizes the PE array design in Catapult
- ***eyeriss.h –*** Source code for Image Processing design utilizing a PE array

- ***types.h –*** This file contains the datatypes and constants necessary for the design
- ***testbench.cpp –*** This file is used to test the design by using weight matrix of blurring, sharpening, outlining, and edge detection over an input image
- ***sw_conv.h –*** Software implementation of Convolution and Max pooling

Background

Eyeriss is an accelerator for state-of-the-art deep convolutional neural networks (CNNs). The Eyeriss architecture focuses on improving the energy efficiency by exploiting local data reuse of the filter weights and the feature map pixels.

Basic 1D Convolution

1D Convolution is implemented using a shift register loop along with a multiply-accumulate loop with the mac result gets shifted through the output registers. This forms the architecture of a processing element for the Convolution.

```
    // Shift feature data in
    PIXWINDOW: for (int i=0; i<KSIZE-1; i++) {
      PixelWindow[i] = PixelWindow[i+1];
    }
    PixelWindow[KSIZE-1] = dataIn;
    // Shift output register
    SHIFT_out_reg: for (int i=IFMAP*KSIZE-1; i>0; i--) {
      OutputShiftReg[i] = OutputShiftReg[i-1];
    }
    // Compute convolution
    macResult = 0;
    {
      MAC: for (int i=0; i<KSIZE; i++) {
        macResult += PixelWindow[i] * WeightsRegs[i];
      }
```

```
    }
    OutputShiftReg[0] = macResult;
    // Compute partial sum output (taps output reg at inded 'idY')
    psumOut = OutputShiftReg[idY] + psumIn;
```

The following is a block diagram of a processing element.

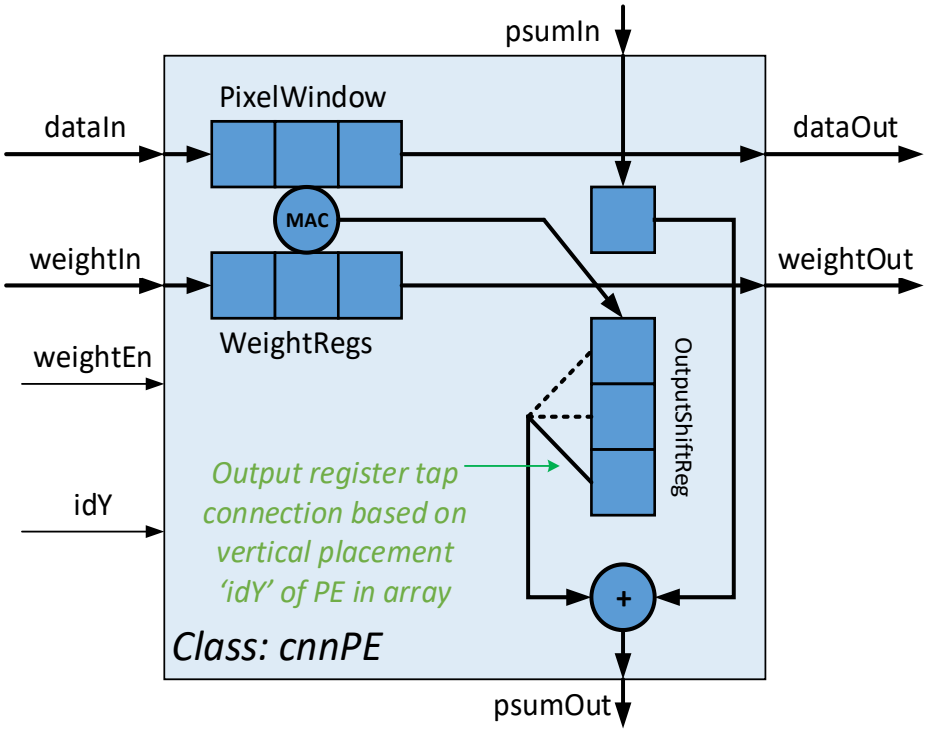


Figure 1 Single Processing Element

From 1D Convolution to 2D

2D convolution is just an extension of 1D convolution stacked vertically accumulating the results across to get the final output (for this example *IFMAP=OFMAP*=1). For 3x3 Convolution, we need to interconnect three processing elements, feeding the *psumIn* of a ***cnnPE*** with *psumOut* of the previous ***cnnPE***.

```
    OFMAP: for (int x=0; x<OFMAP; x++) {
      ARRAY_INTERCONNECT: for (int y=0; y<KSIZE*IFMAP; y++) {
        // (x,y) represents the "position" of this element in the array
        peInstArray[x][y].run(buffIN[x][y], psumIn[x][y], buffWeight[x][y], y,
weightEnbuff[x], dataOut[x][y], weightOut[x][y], psumOut[x][y]);
      }
    }
```

We will provide zero or bias as input to the *psumIn* for the first ***cnnPE*** in the chain. Each ***cnnPE*** taps one index in the output shift register for its *psumOut* based on the ***cnnPE*** placement in the array as shown in the figure.

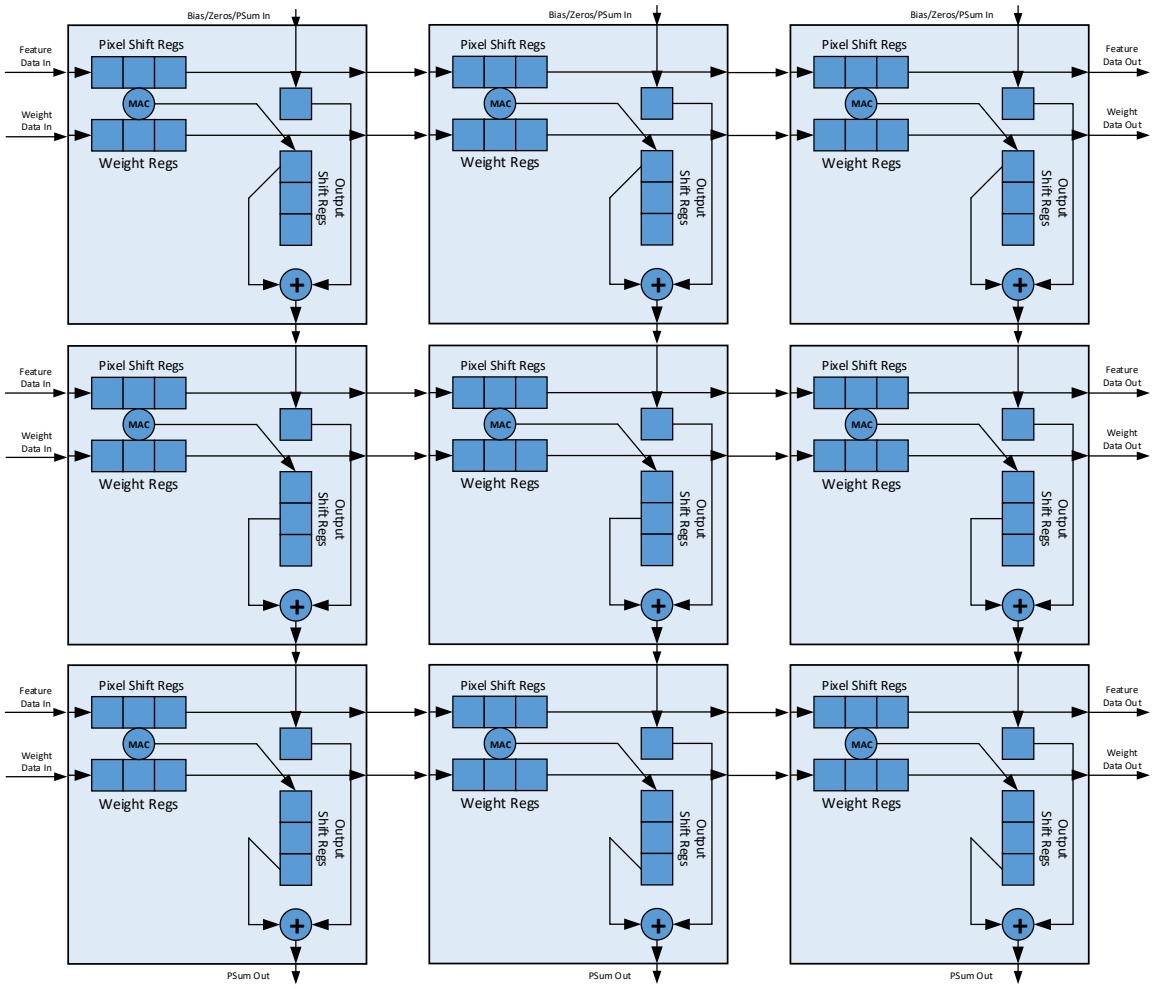
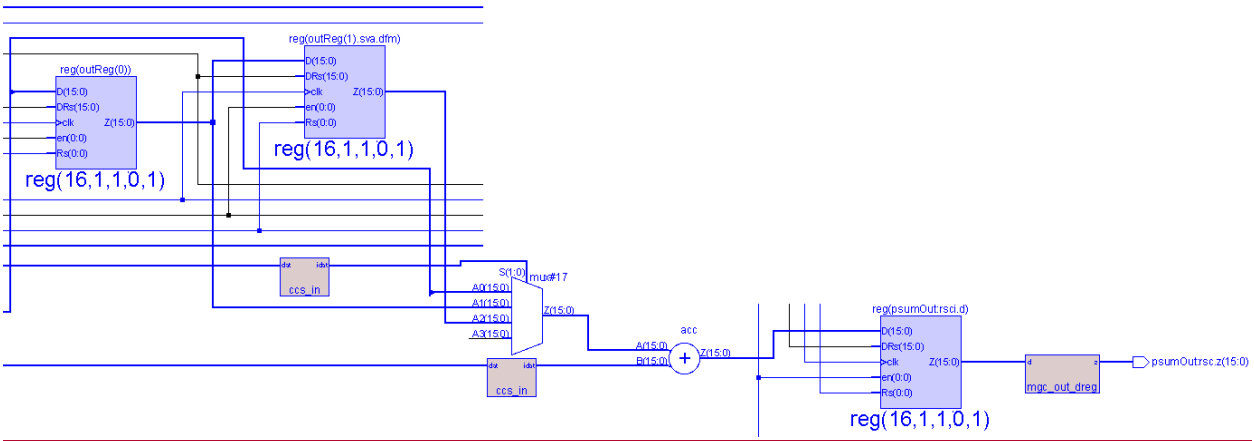


Figure 2 Array of PE's stacked vertically for 2-D Convolution



The ***cnnPE*** array does not differentiate the valid output pixels from the output stream, especially if there is a stride factor involved. The ***vldCNN*** block is used to provide a valid signal for the output sequence.

The ***lineBuffer*** block is used to feed input pixel data into the processing element array. It is made of (Kernel Row Size - 1) blocks of RAM and the control logic to handle images of different shapes. The RAM size determines the maximum image size that can be processed by the array.

The ***imageProcessor*** block coordinates the array, the ***lineBuffer*** and the ***vldCNN*** blocks. It is responsible for loading the configuration and filter weights and then streaming the input and output image pixels. It also provides a reset mechanism for all the blocks.

Reconfigurable Width, Height and CORE Unit

The design as coded will be synthesized to handle image with maximum width and height of 1024. During runtime, the design can be reconfigured to work on any widths and heights which are less than the maximum limit. The configuration gets loaded into the ***imageProcessor*** block either through channel interface or memory interface. When loading the configuration data, ***imageProcessor*** block resets all the other blocks.

Configuration Data	Details
stride	The amount by which the filter shifts
width	Width of the input image
height	Height of the input image
widthKsize	Width of the image x Kernel Row size
imgsz	Total pixel in the image

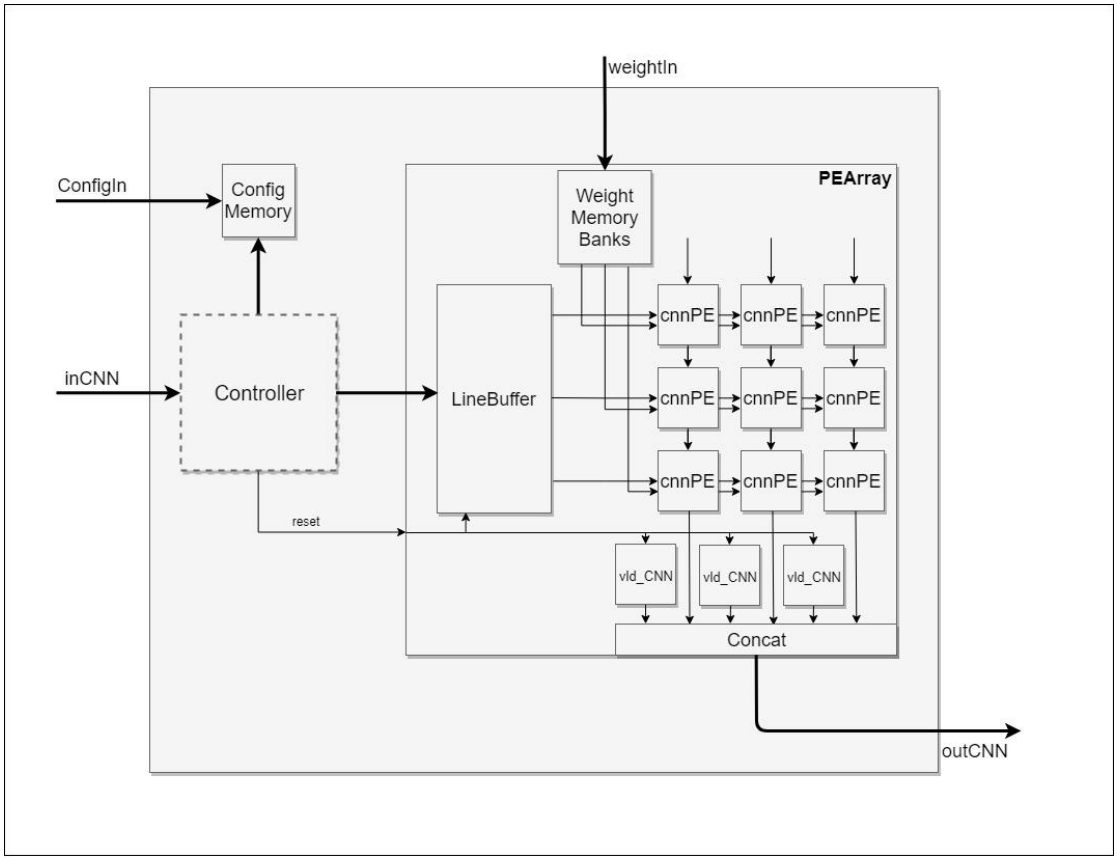


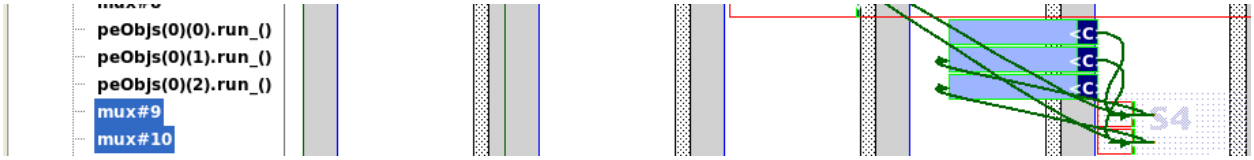
Figure 3 Interconnection between various blocks of the design

Synthesizing the Design

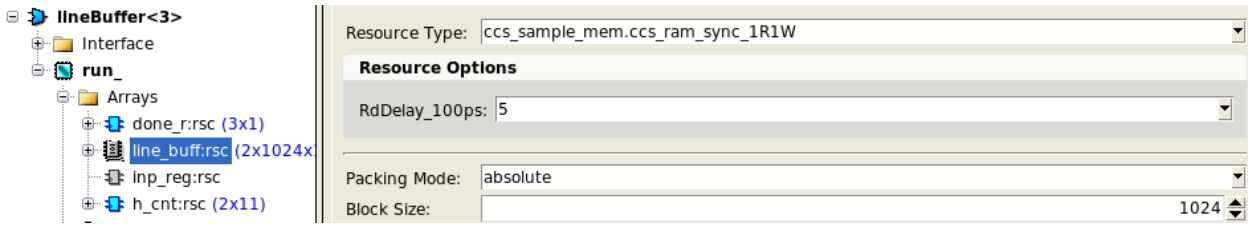
To synthesize the design in Catapult, simply run the directives.tcl script from the shell:

```
% catapult -file directives.tcl
```

Processing element ***cnnPE*** is synthesized as CCORE with the design goal as latency, the clock overhead as 0% and the input delay of 1.0 ns. This design constraints give a neat schedule of all the PEs executing in the same C-step which improves the latency of the overall design.

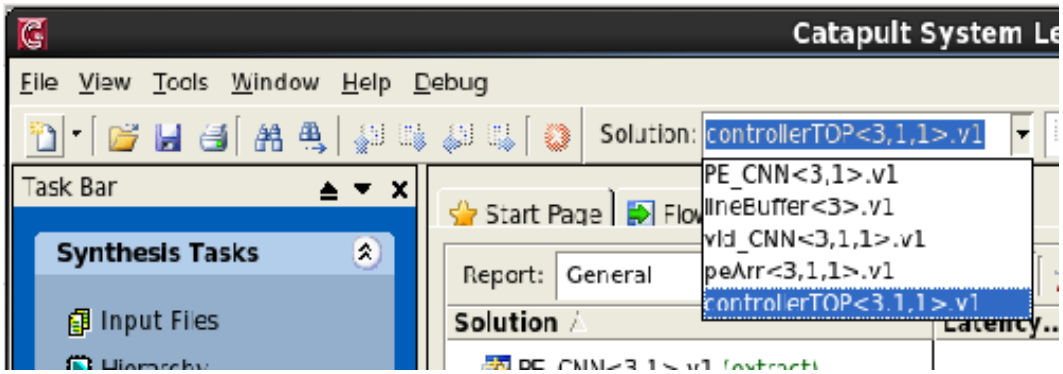


In the ***lineBuffer*** CCORE design, the RAM gets split into blocks which is equal to the Kernel row size-1 for the parallel feeding of the input pixel data to the PE array. For 3x3 Convolution, we need two blocks of RAM for the processing.

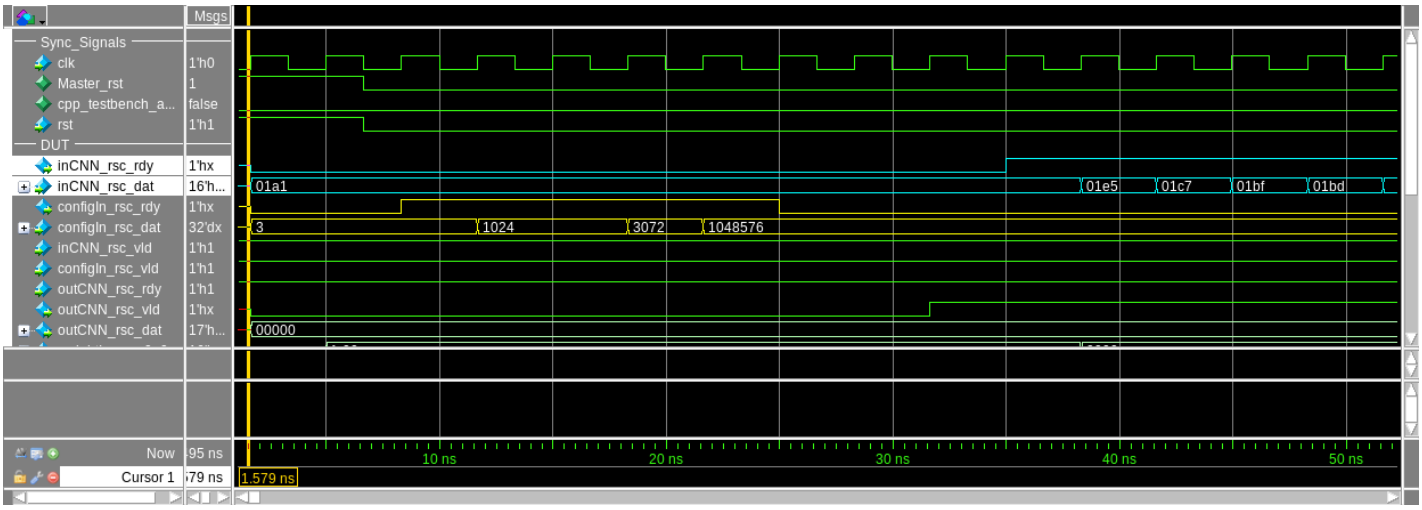


Running the Design

The TCL script will generate the Eyeriss example design in a bottom-up fashion. Once Catapult has finished executing the script you can go back and select the different blocks in the Catapult GUI as shown in Figure.



Simulating the ***imageProcessor*** block in QuestaSim results in the waveform shown in Figure



In the simulation, the configuration data gets loaded first followed by the input pixel data.

Resource Utilization

We synthesized the convolution design for Xilinx Zynq UltraScale+ FPGA (ZU9EG) which has 18Kb BRAMS and 2520 DSP48E2. Table I compares the amount of hardware for different kernel size for 16-bit ac_fixed pixel data.

Table I

Kernel Size	18 Kb BRAMS	DSP48E2	FFs
3x3	2	9	1470
5x5	4	25	2307
7x7	6	49	3285

Table II compares the ac_fixed bit width with respect to the DSP utilization and the error difference between the design and the software outputs. This table is for the sharpen kernel of size 3x3.

Table II

Bit width	DSP48E2	Mean absolute error
8	0	0.45
16	9	0.008
32	36	2.16e-07