TECHNISCHE UNIVERSITÄT DRESDEN

EMBEDDED HARDWARE SYSTEM DESIGN PRACTICAL

Project 1: Conv2D filter for Convolutional layer and accuracy-performance tradeoff using precision scaling

Kazi Shihabul Islam
Matriculation Number: 5044909
Winter Semester 23/24

2D Convolution

2D convolution is just an extension of 1D convolution by convolving both horizontal and vertical directions in two-dimensional spatial domain. Convolution is frequently used for image processing, such as smoothing, sharpening, and edge detection of images.

The impulse (delta) function is also in 2D space, so δ [m, n] has 1 where m and n are zero and zeros at m, n ≠ 0. The impulse response in 2D is usually called "kernel" or "filter" in image processing.

The definition of 2D convolution and the method how to convolve in 2D can be given from the following equation:

$$y[m,n] = x[m,n] * h[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i,j] \cdot h[m-i, n-j]$$

Aim of the project

In this project, the aim is to design and implement a convolution 2d filter for Convolutional layer and accuracy-performance tradeoff using precision scaling on the Zynq Ultra96-V2 board.

The generation of IP is intended to be carried out through Vitis HLS, while the bitstream is produced using Vivado HLS. Additionally, the project implements a kernel/filter on the source stream and the application of approximate optimization techniques to balance output accuracy against performance enhancements. In this project, we optimized the 2D convolution by using approximation techniques such as 1D separable filters.

High Level Synthesis with Vitis HLS

Initially, the number of data_size was selected as 32. However, due to constraints in the tool's resource availability, the number of data_size is scaled down to 9. We have also considered the kernel of size 3X3 for this implementation.

Characteristics of implemented 2D convolution filter.

01. Separable Filter: A filter is separable if its kernel K can be obtained as the outer product of two vectors a * b.
02. The kernel has a rank of 1.
03. Not always equivalent results but acceptable approximation adopting appropriate strategies.
04. Simple and efficient implementation on embedded systems including FPGAs.
05. For separable filters the 2D convolution can be replaced with two distinct 1D convolutions (associative property)
06. Same results, less computations:
    a. Conventional ≈ WxHxMxN
    b. Separable ≈ WxHx(M+N)

Source & Test Bench for the design

The content includes C++ code for designing the filter, accompanied by a header file for incorporating data sets and macros. The bit precision is determined by specifying the kernel size i.e, 3X3.

```
#include "conv2d.h"

void conv2d(hls::stream<axis_data> &in_A, hls::stream<axis_data> &out_C) {
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis register both port=in_A
    #pragma HLS INTERFACE axis register both port=out_C

    // matrices to store inputs and outputs
    Mat_Dtype input_A[MATCOL][MATROW];
#pragma HLS ARRAY_PARTITION variable=input_A complete dim=2
    Mat_Dtype output_C[MATCOL][MATROW];
    Mat_Dtype temp[MATROW][MATCOL];
#pragma HLS ARRAY_PARTITION variable=input_temp complete dim=2

    // 2D convolution using 1D seperable filter
    Mat_Dtype filt33_coeff[3] = {
         1.0, 2.0, 1.0
        };
```

Fig 1. source code conv2d.cpp: Port initialization.

We have initialized two ports using HLS IP with an AXI Input and AXI Output stream given in figure 1.

```
    int rows = MATROW;
    int cols = MATCOL;
    int filterSize = FILTERSIZE;
    int filterRadius = filterSize / 2;
    axis_data local_stream, local_stream_out;

    // read data for Matrix A
    loop_input_A1: for(int i=0; i < MATROW; i++){
        loop_input_A2: for(int j=0; j < MATCOL; j++){
        #pragma HLS PIPELINE
            local_stream = in_A.read();
            input_A[i][j] = local_stream.data;
        }
    }

    R1: for (int i = 0; i < rows; ++i) {
        R2: for (int j = 0; j < cols; ++j) {
#pragma HLS PIPELINE
            for (int k = 0; k < filterSize; ++k) {
                int colIdx = j - filterRadius + k;
                if (colIdx >= 0 && colIdx < cols) {
                    temp[i][j] += input_A[i][colIdx] * filt33_coeff[k];
                }
            }
        }
    }

    C1: for (int i = 0; i < cols; ++i) {
        C2: for (int j = 0; j < rows; ++j) {
#pragma HLS PIPELINE
            for (int k = 0; k < filterSize; ++k) {
                int rowIdx = i - filterRadius + k;
                if (rowIdx >= 0 && rowIdx < rows) {
                    output_C[i][j] += temp[rowIdx][j] * filt33_coeff[k];
                }
            }
        }
    }
```

Fig 2. Source conv2d.cpp: Separable 1D convolution for row and column matrices of the stream input

In figure 2, the source code of the gaussian filter employs a single loop for each row and column for the source matrix, which is pipelined with a pragma directives #pragma HLS PIPELINE to achieve a more favorable trade-off between performance and hardware resources. Despite the loop being unrolled, a substantial overhead is evident due to the involvement of floating-point operations.

Steps for the complete HLS

1. C Simulation

We checked the output of the code using c simulation to identify any errors before synthesis.

2. C Synthesis

Synthesis is performed to map the high-level code to netlist implementation. From the figure 3 given below the core is operating at 105.54 MHz with a latency of 113 clock cycles. II values for loop input A1-A2 and loop output C1-C2 was achieved at 1. Due to complexity of the design the II values for loops R1-R2 and C1-C2 are optimized to 3 using HLS pragma directives. Other utilization metrics can also be observed from figure 4.

General Information

Date: Mon Dec 11 21:44:42 2023
Version: 2018.3.1 (Build 2489210 on Tue Mar 26 04:40:43 MDT 2019)
Project: conv2d_v3
Solution: solution1
Product family: zynquplus
Target device: xczu3eg-sbva484-1-i

Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.475	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
113	113	113	113	none

  - **Detail**
    - **Instance**

N/A

    - **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- loop_input_A1_loop_input_A2	9	9	1	1	1	9	yes
- R1_R2	43	43	20	3	1	9	yes
- C1_C2	43	43	20	3	1	9	yes
- loop_output_C1_loop_output_C2	10	10	3	1	1	9	yes

Fig 3. Performance estimates from synthesis

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	590	-
FIFO	-	-	-	-	-
Instance	-	14	744	1044	-
Memory	8	-	0	0	-
Multiplexer	-	-	-	587	-
Register	2	-	1788	193	-
Total	10	14	2532	2414	0
Available	432	360	141120	70560	0
Utilization (%)	2	3	1	3	0

Fig 4. Utilization estimates from synthesis

3. C/RTL Co-simulation

C/RTL Co-simulation was executed to successfully resolve HDL representation and to make sure no issues arise while exporting the RTL as an IP to Vivado for block implementation.

4. Export RTL as part of implementation

We export the synthesized design to Verilog HDL and configure it as an IP. The IP was exported for input data stream of 9 and kernel size of 3x3 matrix.

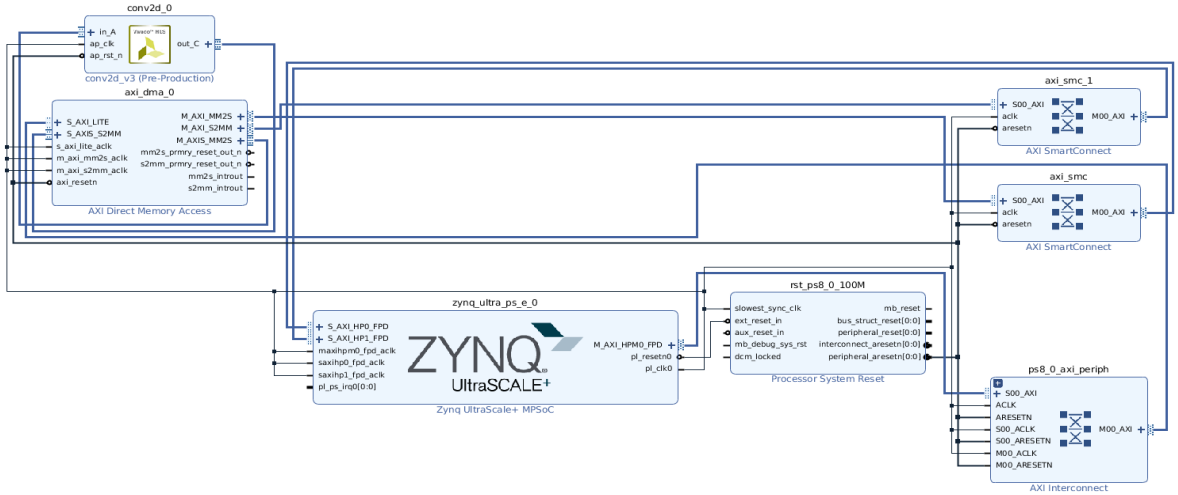

Fig 5. Vivado block diagram

Bitstream generation with Vivado HDL

The synthesized RTL is imported in the HDL to generate the block diagram with Ultra96-V2 MPSOC. The block diagram uses 1 Master & 2 slave configurations along with AXI stream. We also utilized the AXI Direct Memory Access IP for our design. The DMA allows you to stream data from memory, PS DRAM in this case, to an AXI stream interface. This is called the READ channel of the DMA. The DMA can also receive data from an AXI stream and write it back to PS DRAM. This is the WRITE channel.

The DMA has AXI Master ports for the read channel, and another for the write channel, and are also referred to as memory-mapped ports - they can access the PS memory. The ports are labelled MM2S (Memory-Mapped to Stream) and S2MM (Stream to Memory-Mapped). You can consider these as the read or write ports to the DRAM for now.

After validating the diagram, the bitstream is generated using data width of 9 and kernel size of 3. Figure 6 shows the timing constraints for the implemented core in the MPSoc.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.480 ns	Worst Hold Slack (WHS):	0.010 ns	Worst Pulse Width Slack (WPWS):	3.498 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	24751	Total Number of Endpoints:	24751	Total Number of Endpoints:	9135

All user specified timing constraints are met.

Fig 6. Timing constraints of the core.

Programming FPGA with PYNQ Notebook

The Jupiter Notebook (PYNQ) requires three files to interact the implemented design with Ultra96-V2 PL PS subsystem.

a. .bit file: This is utilized to load the overlay, which includes the IP.
b. .hwh file: PYNQ overlays necessitate the inclusion of the hardware handoff (hwh) file. This file is employed by PYNQ to autonomously recognize the Zynq system configuration, IP versions, interrupts, resets, and other control signals.
c. Conv2d.ipynq: Python code for FPGA programming.

The primary objectives of the Jupiter Notebook are described as follows:

At first, from figure 7 we define an overlay for the generated bit stream file to the map IP, DMA. Secondly, we allocate buffers of required data size for sending and receiving data stream as shown in figure 8.

```
from pynq import Overlay
from pynq import allocate
import numpy as np
import math
import cv2
import time


# ### Program the FPGA with the bit file

# ol = Overlay("/home/xilinx/jupyter_notebooks/bit_files/matrix_mult_16.bit")
# ol = Overlay("/home/xilinx/jupyter_notebooks/bit_files/matrix_mult_16_pipelined.bit")
ol = Overlay("/home/xilinx/jupyter_notebooks/kconv2d/design_1.bit")


# ### Check the IPs in the overaly (configuration provided by the bit file)

ol.ip_dict


# ### Create an instance of the DMA and define functions for sending and receiving the data

dma = ol.axi_dma_0

dma_send = ol.axi_dma_0.sendchannel
dma_recv = ol.axi_dma_0.recvchannel
```

Fig 7. Setting up the overlays for the FPGA

```
# ### Define three arrays in the PS memory to store the data transfer
data_size = 9
input_buffer1 = allocate(shape=(data_size,), dtype=float)
output_buffer = allocate(shape=(data_size,), dtype=float)

# ### Generate some test data

for i in range(data_size):
    input_buffer1[i] = (i+1) * 10

print('First sample of data')
for i in range(data_size):
    print(input_buffer1[i])
```

Fig 8. Allocation of buffers

```
# ### Generate some test data

for i in range(data_size):
    input_buffer1[i] = (i + 1) * 5

print('First sample of data')
for i in range(data_size):
    print(input_buffer1[i])


# ### Send the data

start = time.time()

dma_send.transfer(input_buffer1)


dma_send.idle
```

Fig 9. Create signals for filtering

```
# ### Send the data

start = time.time()

dma_send.transfer(input_buffer1)


dma_send.idle

# ### Receive the data from the Streaming FIFO
dma_recv.transfer(output_buffer)
dma.recvchannel.wait()

end = time.time()
fpga_run_time = end - start

print("Output buffer from FPGA")

for i in range(data_size):
    print(output_buffer[i])
```

Fig 10. HW Accelerated function implementation

Figure 9 and 10 depicts the data streams are send and received after a certain interval time and the execution time was recorded. The software implementation was observed using python cv2 library and simulation runtime was also recorded for performance analysis.

```
start = time. time()
#padding needed to considered when using cv2.
#0 padding used
src = np.array([
    [0, 0, 0, 0, 0],
    [0, 5, 10, 15, 0],
    [0, 20, 25, 30, 0],
    [0, 35, 40, 45, 0],
    [0, 0, 0, 0, 0]], np.int16)

kernel = np.array([
    [1, 2, 1],
    [2, 4, 2],
    [1, 2, 1]
])

filtered = cv2.filter2D(src=src, kernel=kernel, ddepth=-1)
print("Expected result")
rows = len(filtered)
cols = len(filtered[0])
for i in range (1, cols-1):
    for j in range (1, cols-1):
        print(filtered[i][j], end = " ")
    print()
end = time.time()
ps_run_time = end -start

dma_recv.idle
print('FPGA run time: ', fpga_run_time)
print('ARM PS run time: ', ps_run_time)
```

```
First sample of data
5.0
10.0
15.0
20.0
25.0
30.0
35.0
40.0
45.0
Output buffer from FPGA
105.0
180.0
165.0
260.0
400.0
340.0
285.0
420.0
345.0
Expected result
105 180 165
260 400 340
285 420 345
FPGA run time:  0.0015707015991210938
ARM PS run time:  0.008320093154907227
```

Fig 11. SW Implementation of the separable conv2d Filter

Performance Analysis

We analyzed 2d convolution using the python library cv2 and our implemented design. We calculated the both the FPGA runtime and PS system runtime, and it was evident that the hardware implementation managed to outperform the PS subsystem at runtime.

Conclusion

The 3x3 2d Gaussian filter design for 2d convolution with the FPGA has the most optimal accuracy. And the expected increase of 6.75ms in the execution time leads to better accuracy. As a result, the performance gain in HW has increased.