Embedded Hardware Systems Design Practical

Task description - Implement a Conv2D operator and then utilize it to implement a Convolutional layer. The implementation should be tested on an Ultra96 board using a test dataset. Furthermore, precision scaling would be considered to explore accuracy-performance trade-offs. Both HDL and HLS can be used for the implementation.

VIVADO HLS - For this task I used custom IP, which performs convolution operation and has parameters like Image and kernel matrix size, stride and padding using Vivado HLS software. I choose HLS as it is easy and convenient to compile, simulate, debug and optimise algorithm implement in C/C++. The parameters are fixed but can be changed in the header file according to the requirements and again export the IP.

header.h

```
#include <stdio.h>
#include <hls_stream.h>
#include <ap_int.h>       // For last signal (1-bit) of streaming protocol


#define image_size 3      // Size of Input image
#define kernal_size 3     // Kernal matrix size
#define stride 1          // Step size of kernel movemnent (Assumed equal in both dimension)
#define padding 1         // Additional pixel added to input image (Assumed equal in both dimension)


// Define the data type of matrix inputs and outputs
typedef double Mat_Dtype;


// Define the axis data structure
struct axis_data {
    Mat_Dtype data;
    ap_uint<1> last;
};


void Conv(hls::stream<axis_data> &data_in, hls::stream<axis_data> &data_out);

```

- The data is read and sent by IP using AXI input stream and AXI output stream channels create using below pragmas.
  - #pragma HLS INTERFACE ap_ctrl_none port=return
  - #pragma HLS INTERFACE axis register both port=data_in
  - #pragma HLS INTERFACE axis register both port=data_out

- Image and kernel matrix are created and fed in values using stream input. Then padding is applied to the input image matrix. Then the convolution operation is carried out as below and then output image is sent via stream output channel.

```
// Read data for Input Image Matrix
L1_Image_Matrix: for(i=0; i < image_size; i++){
    L2_Image_Matrix: for(j=0; j < image_size; j++){
        local_stream = data_in.read();
        input_mat[i][j] = local_stream.data;}}
```

```
// Read data for Kernal Matrix
 L1_Kernal_Matrix: for(i=0; i < kernal_size; i++){
     L2_Kernal_Matrix: for(j=0; j < kernal_size; j++){
        local_stream = data_in.read();
        kernal_mat[i][j] = local_stream.data;}}
```

```
// Convolution operation
C1:  for (i = 0; i < output_size; ++i) {
    C2:  for (j = 0; j < output_size; ++j) {
        output_image[i][j] = 0;
        C3:  for (m = 0; m < kernal_size; ++m) {
            C4:  for (n = 0; n < kernal_size; ++n) {
                output_image[i][j] += padded_image[i * stride + m][j * stride + n] * kernal_mat[m][n]; } } } }
```

```
// Output stream the result
output_loop_1: for(i=0; i < output_size; i++){
    output_loop_2: for(j=0; j < output_size; j++){
        local_stream.data = output_image[i][j];

            // Last signal and the strobe signal
            if((i==output_size-1) && (j==output_size-1))
                local_stream.last = 1;
            else
                local_stream.last = 0;
            data_out.write(local_stream);}}
```

Un-optimized code

- 2 different types of IP were implemented and their performance were compared. One was without any optimisations using directives and other was fully optimised. For optimisation, I used following pragmas.

  - **#pragma HLS LOOP_FLATTEN** - Allows nested loops to be flattened into a single loop hierarchy with improved latency. This saves clock cycles, potentially allowing for greater optimization of the loop body logic.

  - **#pragma HLS LOOP_MERGE** - It merges consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization. Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations.

  - **#pragma HLS PIPELINE II=5** – A pipelined function or loop can process new inputs every 5 clock cycles, where 5 is the II (initiation interval) of the loop or function. It allows the concurrent execution of operations.

```
// Read data for Input Image Matrix
L1_Image_Matrix: for(i=0; i < image_size; i++){
    L2_Image_Matrix: for(j=0; j < image_size; j++){
        #pragma HLS LOOP_FLATTEN
        local_stream = data_in.read();
        input_mat[i][j] = local_stream.data;}}
```

```
// Read data for Kernal Matrix
 L1_Kernal_Matrix: for(i=0; i < kernal_size; i++){
     #pragma HLS LOOP_MERGE
     L2_Kernal_Matrix: for(j=0; j < kernal_size; j++){
        local_stream = data_in.read();
        kernal_mat[i][j] = local_stream.data;}}
```

```
// Convolution operation
C1:  for (i = 0; i < output_size; ++i) {
    C2:  for (j = 0; j < output_size; ++j) {
        #pragma HLS PIPELINE II=5
        output_image[i][j] = 0;
        C3:  for (m = 0; m < kernal_size; ++m) {
            C4:  for (n = 0; n < kernal_size; ++n) {
                    output_image[i][j] += padded_image[i * stride + m][j * stride + n] * kernal_mat[m][n]; } } } }
```

```
// Output stream the result
output_loop_1: for(i=0; i < output_size; i++){
    #pragma HLS LOOP_MERGE
    output_loop_2: for(j=0; j < output_size; j++){
        local_stream.data = output_image[i][j];

            // Last signal and the strobe signal
            if((i==output_size-1) && (j==output_size-1))
                local_stream.last = 1;
            else
                local_stream.last = 0;
            data_out.write(local_stream);}}
```

Optimized code

- Analysing both design after synthesis

Table -1 Comparison between both design

Not – Optimised design	Optimised design
• **Latency (clock cycles)** ◦ **Summary** Latency (min 1172, max 1172) / Interval (min 1172, max 1172) / Type: none	• **Latency (clock cycles)** ◦ **Summary** Latency (min 217, max 217) / Interval (min 217, max 217) / Type: none

- **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- L1_Image_Matrix	15	15	5	-	-	3	no
+ L2_Image_Matrix	3	3	1	-	-	3	no
- L1_Kernal_Matrix	15	15	5	-	-	3	no
+ L2_Kernal_Matrix	3	3	1	-	-	3	no
- memset_padded_image	29	29	6	-	-	5	no
+ memset_padded_image	4	4	1	-	-	5	no
- Loop 4	24	24	8	-	-	3	no
+ Loop 4.1	6	6	2	-	-	3	no
- C1	1050	1050	350	-	-	3	no
+ C2	348	348	116	-	-	3	no
++ C3	114	114	38	-	-	3	no
+++ C4	36	36	12	-	-	3	no
- output_loop_1	33	33	11	-	-	3	no
+ output_loop_2	9	9	3	-	-	3	no

- **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- L1_Image_Matrix_L2_Image_Matrix	9	9	1	-	-	9	no
- L1_Kernal_Matrix	15	15	5	-	-	3	no
+ L2_Kernal_Matrix	3	3	1	-	-	3	no
- memset_padded_image	29	29	6	-	-	5	no
+ memset_padded_image	4	4	1	-	-	5	no
- Loop 4	24	24	8	-	-	3	no
+ Loop 4.1	6	6	2	-	-	3	no
- C1_C2	95	95	56	5	5	9	yes
- output_loop_1	33	33	11	-	-	3	no
+ output_loop_2	9	9	3	-	-	3	no

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	598	-
FIFO	-	-	-	-	-
Instance	-	14	744	984	-
Memory	2	-	384	27	-
Multiplexer	-	-	-	488	-
Register	-	-	736	-	-
Total	2	14	1864	2097	0
Available	432	360	141120	70560	0
Utilization (%)	~0	3	1	2	0

Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	735	-
FIFO	-	-	-	-	-
Instance	-	28	1488	1968	-
Memory	4	-	384	27	-
Multiplexer	-	-	-	805	-
Register	14	-	2880	71	-
Total	18	28	4752	3606	0
Available	432	360	141120	70560	0
Utilization (%)	4	7	3	5	0

VIVADO Design suite - For this task I used Xilinx AXI DMA and controlled it from PYNQ using Jupiter notebook. DMA allows to stream data from memory to an AXI stream interface. This is called READ channel of the DMA. It can also read data from an AXI stream and write it back to memory using WRITE channel.

- This DMA is connected to Zynq Processing Subsystem (PS) with more peripherals like AXI interconnect, AXI smart connect and processor system reset.

- Configuring AXI DMA IP as follows.
  - Width of buffer length register – 26 bits *(determines the maximum packet size for a single DMA transfer).*
  - Address width – 64 bits *(because we are using Zynq Ultrascale+).*
  - Read and write channels enabled
  - Memory map data width - 64 bits *(defined in the PYNQ image and applied at boot time).*
  - Stream data width – 64 bits *(match with IP stream width, for best results - match the settings of the DMA to your datapath).*

- After adding the IP to our Vivado block design is completed as below.

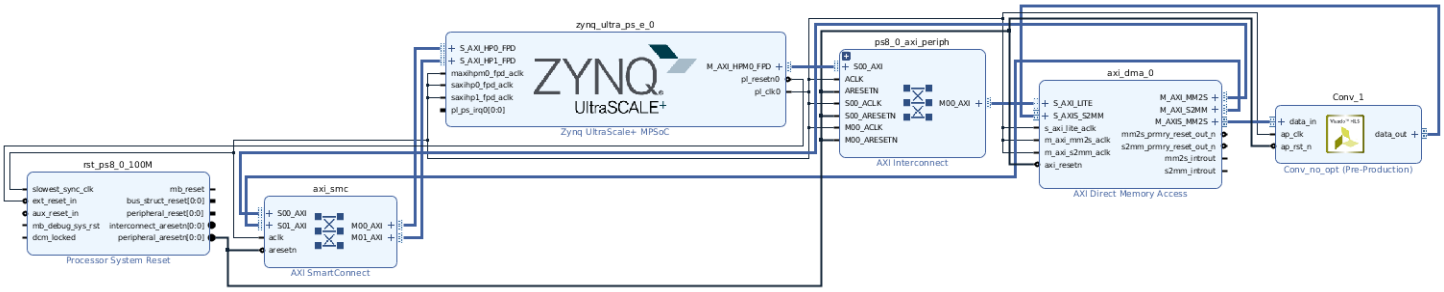

Figure-1 Vivado block design

- Creating HDL wrapper and generating bitstream files. Once completed repeated the same procedure for IP with optimised code.

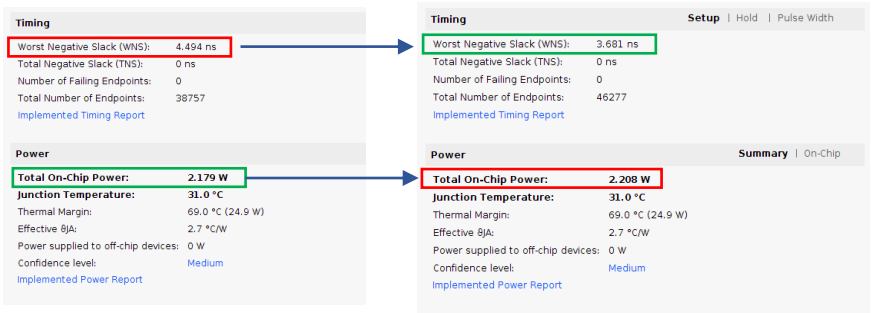

Figure-2 Timing and power report un-optimized and optimized design

- Uploading the .bit and .hwh files in Jupiter notebook and running the python script.

- The .py scripts first send the image and kernal data via send channel and receive output in output buffer. Then convolution function implemented in python using numpy library calculates the output.

- Both results are compared for equality. The end result are as follows.

```
print("Input Matrix:")
print(A_matrix)
print("\nKernel Matrix:")
print(B_matrix)
print("\nConvolution result by python script:")
print(result)

Input Matrix:
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]

Kernel Matrix:
[[-1. -2. -1.]
 [ 0.  0.  0.]
 [ 1.  2.  1.]]

Convolution result by python script:
[[ 13.  20.  17.]
 [ 18.  24.  18.]
 [-13. -20. -17.]]

In [9]: # Checking if both output matrix are equal

if np.array_equal(Output_Matrix, result):
    print("The matrices are equal.")
else:
    print("Both matrix doesnt match")

The matrices are equal.

In [10]: print('FPGA run time: ', fpga_run_time)
print('ARM PS run time: ', ps_run_time)

FPGA run time:  29.145351886749268
ARM PS run time:  0.0034723281860351562
```

Result of Unoptimized bit file

```
result = convolution(A_matrix,B_matrix, stride, padding)

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

print("Input Matrix:")
print(A_matrix)
print("\nKernel Matrix:")
print(B_matrix)
print("\nConvolution result by python script:")
print(result)

Input Matrix:
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]

Kernel Matrix:
[[-1. -2. -1.]
 [ 0.  0.  0.]
 [ 1.  2.  1.]]

Convolution result by python script:
[[ 13.  20.  17.]
 [ 18.  24.  18.]
 [-13. -20. -17.]]

In [20]: # Checking if both output matrix are equal

if np.array_equal(Output_Matrix, result):
    print("The matrices are equal.")
else:
    print("Both matrix doesnt match")

The matrices are equal.

In [21]: print('FPGA run time: ', fpga_run_time)
print('ARM PS run time: ', ps_run_time)

FPGA run time:  35.36079216003418
ARM PS run time:  0.00314068794250488
```

Result of Optimized bit file

Observation and Conclusion

- Pragmas were added to reduce the critical path and latency. It significantly **reduced the latency** by 81% as seen in table-1.

- We can also observe from table-1 that **no. of loop iteration also significantly decreased** by using Loop flatten and merge pragma.

- But then **resource utilization increased** in the optimised case that means overall design area increased too.

- Total on-chip **power increased slightly** as well. Which was expected as no of resource increased, they would utilize more power. Refer figure-2.

- So, there is always trade-off between timing, power and area. All cannot be optimised at the same time. In this project timing was prioritised over others.

Submitted by,
Kishan Dadhania
Matriculation no. - 5112720