CSIE DIC Lab 2024

Lab06. Depth-wise separable convolution

Description

In this lab, you are tasked with designing a circuit to perform a series of convolution operations on an input image, followed by activation functions. Specifically, you will implement depth-wise convolutions, point-wise convolutions, and apply the Rectified Linear Unit (ReLU) function after each convolutional step.

The input image is 1×100×3, which represents the height, width, and number of channels of the image, respectively. The processing steps include two rounds of depth-wise convolution, each using a set of three 1×7 filters tailored to each of the image's channels. Instead of applying the ReLU function after each depth-wise convolution, it is applied only once after completing both rounds of depth-wise convolution. The ReLU function is instrumental in introducing non-linearity to the process by converting all negative pixel values to zero while leaving positive values unchanged.

Following the ReLU function, the circuit should perform a point-wise convolution. This step utilizes three 1×1 filters, one for each channel, effectively integrating information across the channels at each pixel location. The output from the point-wise convolution undergoes one final transformation with the ReLU function, ensuring that any remaining negative values are set to zero.

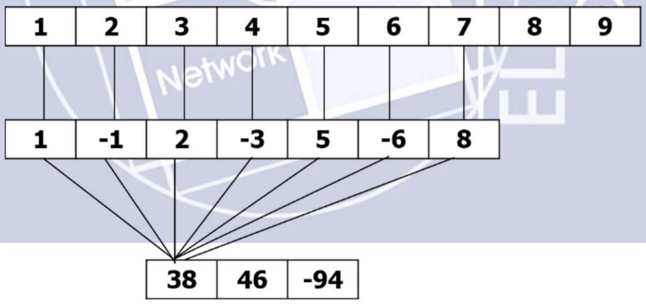

Figure 1: Example of the 1-D convolution operation.

Figure 1 illustrates an example of a one-dimensional (1-D) convolution operation. In this example, a 1×9×1 input image, composed of nine pixels, undergoes convolution with a 1×7 filter kernel. This process involves multiplying each of the seven pixels in a segment of the input image by the corresponding values in the 1×7 filter kernel, then summing these products to produce a single pixel in the output image. The convolution operation progresses across the input image one pixel at a time, a method referred to as using a stride of 1. As demonstrated in Figure 1, with the input dimensions being 1×9×1, the resulting output image from this convolution process is of size 1×3×1, showcasing the effect of the convolution kernel size and stride on the dimensions of the output image.

Figure 2 illustrates an example of depth-wise convolution on an input image with dimensions of 1×15×3 (height, width, channels). For this process, fifteen pixels from each channel of the input image are convolved with a 1×7 kernel filter designated for each channel, leading to the output of the first convolutional layer with 9 pixels from each channel. Following this, a second convolution operation is performed using the same 1×7 kernel filter for each channel, resulting in a final output of three pixels per channel. Therefore, the final output map size is 1×3×3 (height, width, channel), with a stride of 1.

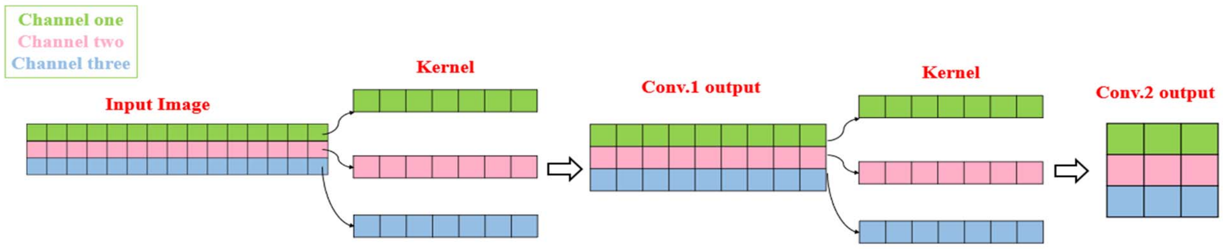

Figure 2: Example of performing the depth-wise convolution operation twice.

Afterward, the output from the depth-wise convolution, which has dimensions of 1×3×3, undergoes processing by the ReLU function. The formula for the ReLU function is provided below.

$$f(x) = \begin{cases} 0, & \text{if } x \leqq 0 \\ x, & \text{if } x > 0 \end{cases}$$

Figure 3 depicts an example of the ReLU function in action. Following the application of the ReLU function, all negative values in the data are set to zero, while positive values are left unchanged.

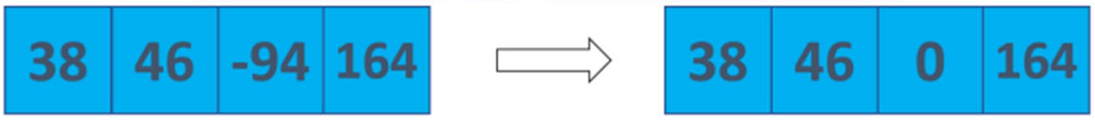

Figure 3: Example of the ReLU operation.

The output from the ReLU function serves as the input for the point-wise convolution. Figure 4 illustrates a point-wise convolution operation, where the pixels across each channel are individually multiplied by a 1×1 filter kernel, and the results are then summed up to yield three final pixel values. For instance, the value of the first pixel in the output map is derived by computing 1×3 + 4×2 + (-7) ×1, which gives a result of 4. In a similar manner, the value for the second pixel in the output map is determined by 2×3 + 5×2 + 8×1, resulting in a total of 24. Subsequently, the output from the point-wise convolution undergoes processing by the ReLU function.

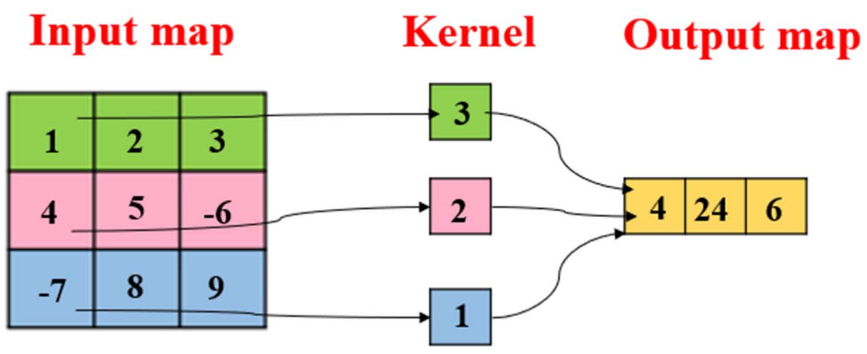

Figure 4: Example of the point-wise convolution operation

Specification

1. Top module name: Convolution
2. Top module filename: Convolution.v
3. Input/output definition

Signal Name	Direction	Bit Width	Description
CLK	Input	1	Clock signal
RESET	Input	1	Asynchronous reset signal (active high)
IN_VALID	Input	1	Asserted when IN_DATA is valid
IN_DATA_1	Input	5	Input pixel values of channel one (unsigned numbers)
IN_DATA_2	Input	5	Input pixel values of channel two (unsigned numbers)
IN_DATA_3	Input	5	Input pixel values of channel three (unsigned numbers)
KERNEL_VALID	Input	1	Asserted when KERNEL is valid
KERNEL	Input	8	Input filter kernels, include three 1×7 depth-wise filter kernels and three 1×1 point-wise filter kernels consecutively (both signed numbers)
OUT_VALID	Output	1	Asserted when OUT_DATA is valid
OUT_DATA	Output	32	The final output image (unsigned number)

4. All input signals are **synchronized at the clock rising edge**.
5. The reset scheme is an **active-high asynchronous reset**.

6. For an input image with three channels, each channel receives 1×100 input pixel values (IN_DATA_1, IN_DATA_2, IN_DATA_3) fed consecutively from left to right. Additionally, the IN_VALID signal will be asserted for 100 cycles to indicate the valid input periods. A detailed input timing diagram illustrating the pixel sequence is presented in Figure 5.
7. When the KERNEL_VALID signal is asserted, the filter kernels are inputted through the KERNEL input. To input the filter kernels, a total of 24 cycles are required to consecutively enter the three 1×7 depth-wise filter kernels for channels 1 to 3 and the three 1×1 point-wise filter kernels for channels 1 to 3. A detailed input timing diagram illustrating the sequence for kernel values is also depicted in Figure 5.

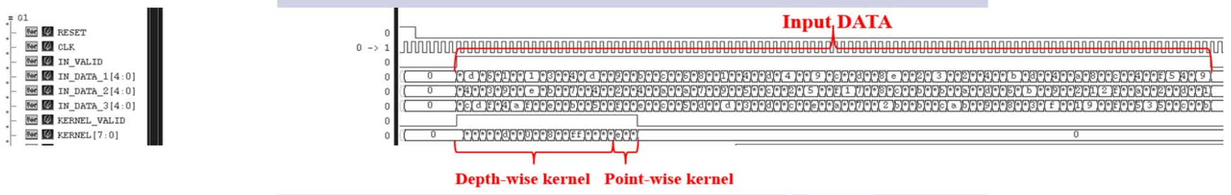

Figure 5: Input timing diagram

8. Upon the availability of the point-wise convolution results, an output valid signal (OUT_VALID) should be asserted, as illustrated in Figure 6. Subsequently, the point-wise convolution results processing by the ReLU function are outputted.

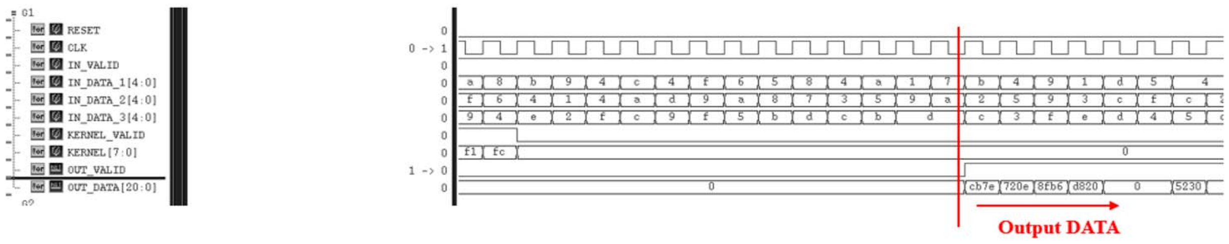

Figure 6: Output timing diagram

9. **Ensure that the latency of your design is less than 15 clock cycles.** Latency is defined as the number of cycles between the last KERNEL input and the first output, as illustrated in Figure 7.

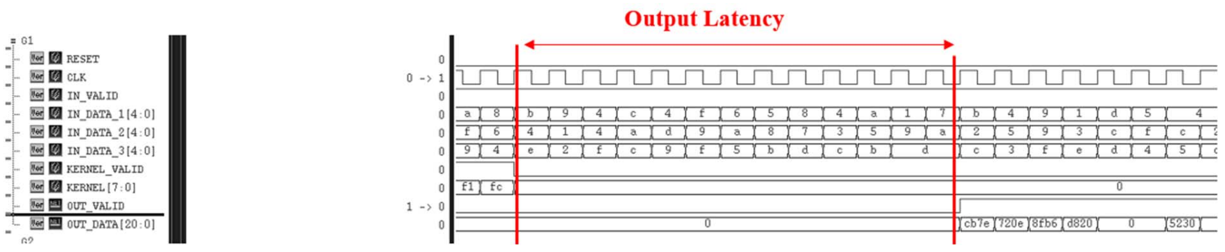

Figure 7: Output latency timing diagram

10. In logic synthesis, the timing constraint for the clock period **MUST** be **0.37ns**. Thus

your design should operate correctly at this clock speed.

Lab Instructions

1. Extract LAB data from TA's directory:
   ```
   % tar   xzvf   ~dicta/lab06.tar.gz
   ```
2. The extracted LAB directory (**lab06**) contains the following:
   - a. **Democase/**   :Design example, including RTL code and synthesis script
   - b. **Exercise/**   :Test pattern for this lab

Democase:

3. Change to the directory: **Democase/**
4. This directory contains:
   - a. **RTL/**   :RTL source code of design example
   - b. **Synthesis/** :Synthesis script
   - c. **Netlist/**   :Synthesized netlist and test pattern for gate-level simulation
5. Change to the directory: **RTL/**
6. Run RTL simulation
   ```
   % ncverilog   –f   run.f
   ```
7. Change to the directory: **Synthesis/**
8. Open and read the synthesis script and synthesis constraint file.
   ```
   % gedit   SYN_ADFP_RC.tcl   &
   % gedit   SYN_RTL.sdc   &
   ```
9. Start logic synthesis,
   ```
   % ./01_run_synthesis
   ```
10. Open and read the synthesis reports.
    ```
    % gedit   report.area   &
    % gedit   report.timing   &
    % gedit   report.datapath   &
    % gedit   report.summary   &
    ```
11. Change to directory: **Netlist/**
12. Check the difference between **(TEST.v & run.f)** in **RTL/** and **(TEST_gate.v & run.f)** in **Netlist/**.
13. Run gate-level simulation:
    ```
    % ncverilog   –f   run.f
    ```

Exercise:

14. Change to the directory: **Exercise**
15. This directory contains:
    - a. **TEST.v**   : Design test bench for RTL simulation

b.	**TEST_gate.v**	: Design test bench for Gate-level simulation
c.	**Convolution.v**	: Empty design module
d.	**lab06.rc**	: nWave saved signal file
e.	**SYN_RTL.sdc**	: Design constraint file for logic synthesis
f.	**weight_1.dat~weight_3.dat**	: Input image for answer 1
g.	**weight_1_2.dat~weight_3_2.dat**	: Input image for answer 2
h.	**kernel.dat**	: Input kernel for answer 1
i.	**kernel_2.dat**	: Input kernel for answer 2

16. Write your RTL code in a synthesizable coding style:

    % gedit Convolution.v &

    or % gvim Convolution.v &

    or % joe Convolution.v

    or % vim Convolution.v

17. Run RTL simulation:

    % ncverilog –f run.f

18. Open waveform:

    % nWave &

    Choose **File** → **Open,** then select **" Convolution.fsdb"** and press **OK**

19. Restore signal recorded for debugging:

    Choose **File** → **Restore signal**, then select **" lab06.rc"** and press **OK**

20. Meaning of signals in TEST group:

cycle_cnt	: cycle count
count_out	: accumulated number of output
error_cnt	: accumulated error number

Logic Synthesis:

21. Repeat steps 7 to 13 as described in the Demo case to synthesize your design. Additionally, you must prepare all files required for logic synthesis. We provide only the design constraint file (**SYN_RTL.sdc**) for logic synthesis and the test bench (**TEST_gate.v**) for gate-level simulation. Ensure that your design passes the gate-level simulation without any timing violations.