15: Hardware Pipelining II

ENGR 315: Hardware/Software CoDesign

Andrew Lukefahr

Indiana University

Some material taken from:
https://github.com/trekhleb/homemade-machine-learning/tree/master/homemade/neural_network
http://cs231n.github.io/neural-networks-1/

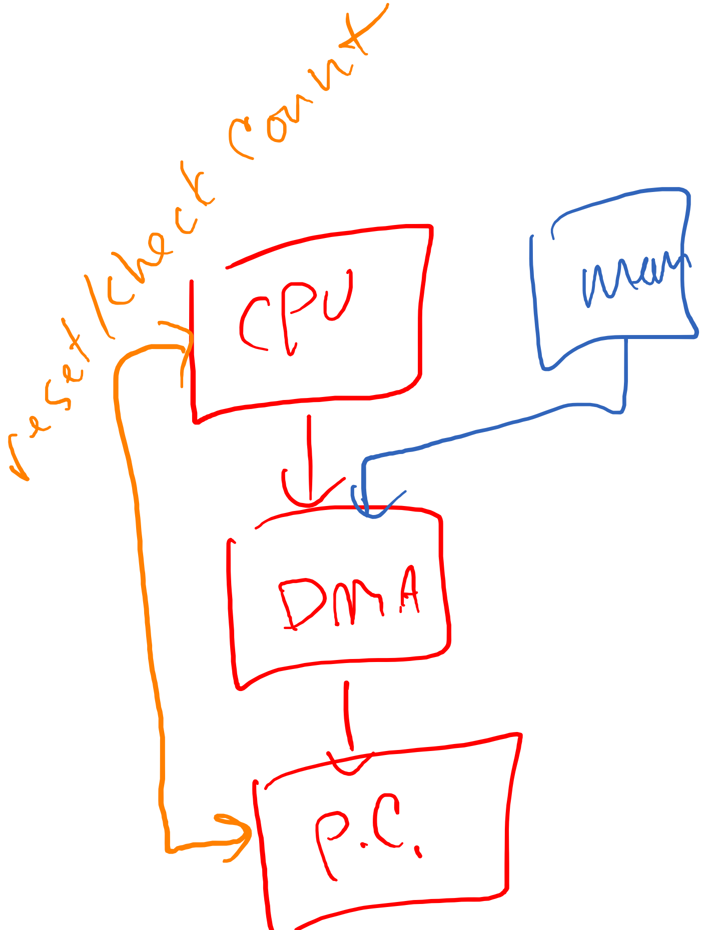
Announcements

- P5 is out

- P6 is out (I think)

- Exam in 3 weeks

P5: Adds DMA + AXI-Stream to Popcount



- DMA
  - Add DMA engine to move data via AXI4-Full to AXI-Stream interface
- Popcount.sv:
  - Add AXI-Stream Interface
  - Keep AXI4-Lite Interface to read result

Example DMA

https://pynq.readthedocs.io/en/v2.6.1/pynq_libraries/dma.html

```
import numpy as np
from pynq import allocate
from pynq import Overlay

overlay = Overlay('example.bit')
dma = overlay.axi_dma

input_buffer = allocate(shape=(5,), dtype=np.uint32)
output_buffer = allocate(shape=(5,), dtype=np.uint32)
```

```
for i in range(5):
    input_buffer[i] = i
```

```
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```

```
Output buffer will contain: [0 1 2 3 4]
```

in ✓ → $\boxed{\text{FPGA}}$ → ~~out~~

count (MMIO)

P6 – DMA from C

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS = 1). The halted bit (DMASR.Halted) should deassert indicating the MM2S channel is running.

2. Skip

3. Write a valid source address to the MM2S_SA register.

4. Write the number of bytes to transfer in the MM2S_LENGTH register.

   The MM2S_LENGTH register must be written last.

5. ~~Poll~~ MM2S_DMASR.Idle bit for completion

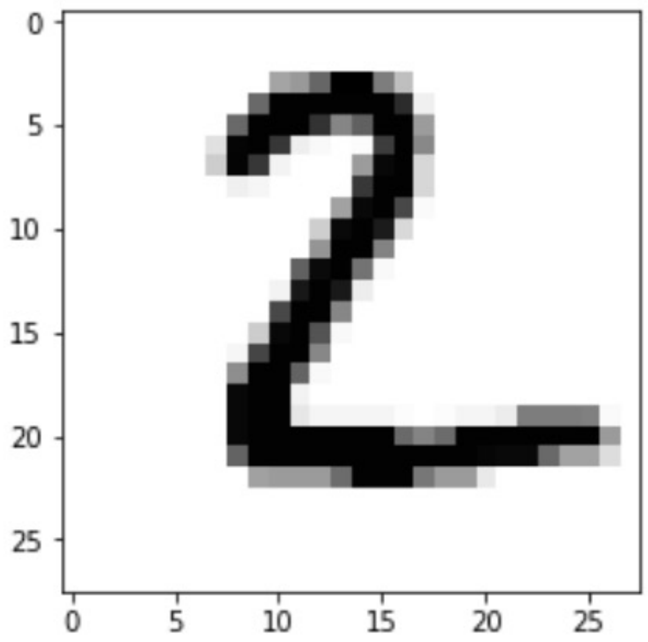
Wait until MM2S_DMASR. Idle = 1

P7+ Accelerate Machine Learning

- Goal: Accelerate reference neural network
- Harder, more open-ended projects
- Groups of 2 allowed.

Simple Neural Network

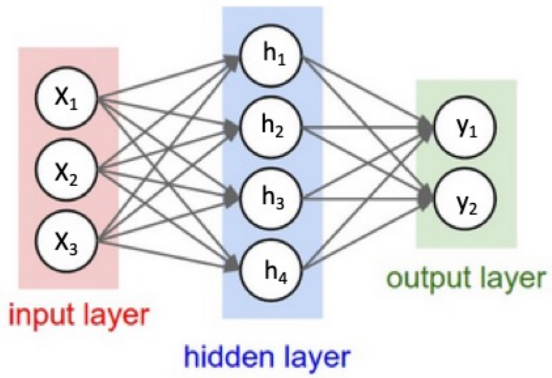
```
================================
Index: 0
Image:
```




```
ML Classification Result: 2
Real Value: 2
Correct Result: True
================================
```

- Takes in image of number
- Returns integer value
- How? artificial neural network

Why Dot Product?




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Matrix Multiplication (Dot Product)

$$\begin{bmatrix} i_0 & i_1 \end{bmatrix} \times \begin{bmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \end{bmatrix} = \begin{bmatrix} o_0 & o_1 & o_2 \end{bmatrix}$$

$$o_0 = i_0 \cdot w_{00} + i_1 \cdot w_{01}$$

$$o_1 = i_0 \cdot w_{10} + i_1 \cdot w_{11}$$

$$o_2 = i_0 \cdot w_{20} + i_1 \cdot w_{21}$$

Alternative Dot Computations

$$[0.1 \quad 0.2] \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = [0.9 \quad 1.2 \quad 1.5]$$

$$0.1 \cdot 1 \qquad 0.1 \cdot 2 \qquad 0.1 \cdot 3 = [0.1 \quad 0.2 \quad 0.3]$$

$$0.2 \cdot 4 \qquad 0.2 \cdot 5 \qquad 0.2 \cdot 6 = [0.8 \quad 1.0 \quad 1.2]$$

1 input @ a time?

$$\begin{array}{r} [0.1 \quad 0.2 \quad 0.3] \\ +\ [0.8 \quad 1.0 \quad 1.2] \\ \hline 0.9 \quad 1.2 \quad 1.5 \end{array}$$

Alternative Dot Computations

$$[0.1 \quad 0.2] \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = [0.9 \quad 1.2 \quad 1.5]$$

round 1

$$[0.1 \cdot 1 \quad 0.1 \cdot 2 \quad 0.1 \cdot 3] + [0 \quad 0 \quad 0] = [0.1 \quad 0.2 \quad 0.3]$$

round 2

$$[0.2 \cdot 4 \quad 0.2 \cdot 5 \quad 0.2 \cdot 6] + [0.1 \quad 0.2 \quad 0.3] = [0.9 \quad 1.2 \quad 1.5]$$

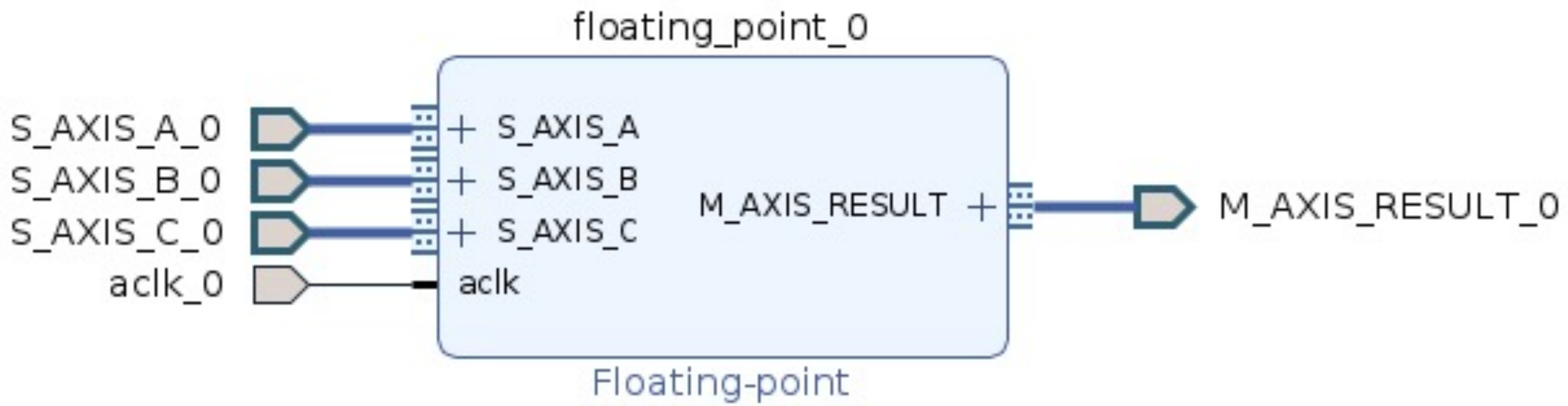
Alternative Dot

```
Input                  Weights                 Output
[[0.1 0.2 0.3]]   .   [1. 2. 3. 4.]        =  [3.8000002 4.4       5.        5.6000004]
                      [5. 6. 7. 8.]
                      [ 9. 10. 11. 12.]
```

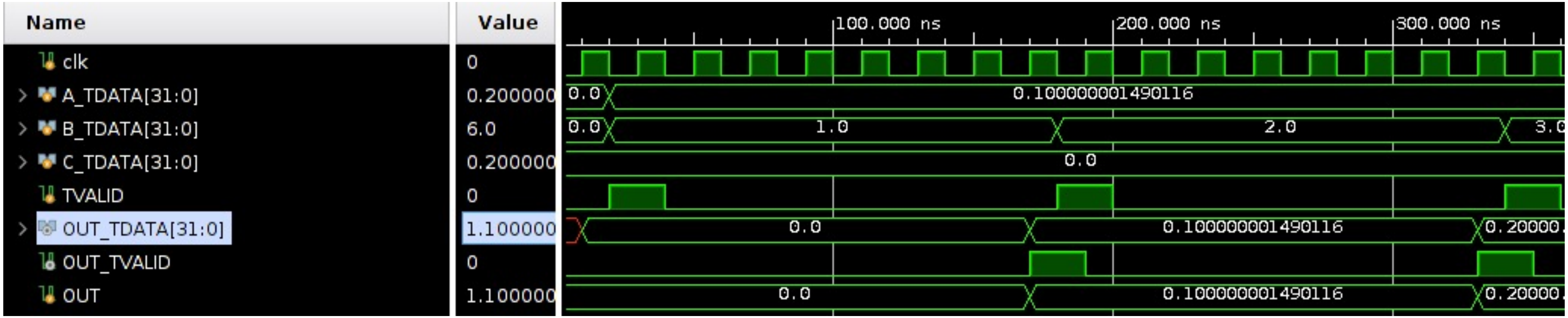
```
Inputs (Shape):
 (1, 3)
Output (Shape):
 (1, 4)
Weights (Shape):
 (3, 4)
```

```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j] + weights[i][j] * inputs[i]
    return outs
```

Hardware Dot Product



Demo Time


Name
Value
clk
A_TDATA[31:0]
B_TDATA[31:0]
C_TDATA[31:0]
TVALID
OUT_TDATA[31:0]
OUT_TVALID
OUT
0
0.200000
6.0
0.200000
0
1.100000
0
1.100000
100.000 ns
200.000 ns
300.000 ns
0.0
0.100000001490116
1.0
2.0
0.20000

Floating-Point Multiply-Accumulate (FMAC)

- Math: a * b + c ← 8 cycles

Floating-Point math takes 8 cycles.

- Floating-Point is complicated.
- How do we work around an 8 cycle latency?
- Pipelining!

Pipelining

- FMAC takes 8 cycles for 1 value
- But can accept a new value every cycle.

- Latency: 8 cycles / value
- Throughput: 1 value / cycle

Latency vs. Throughput

- Latency: How long does an individual operation take?

- Throughput: How many operations can you complete in a given time?

Pipelining

Pipelined Dot Computations

$$\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

Problems with Pipelined FMAC

Latency!

Latency on Pipelined FMAC

- Solution: Stall at the end of a row.

- Drain the pipeline.

Hardware Parallelism

- CPU: 1 Floating-Point Unit
- FPGA? 10 Floating-Point Units?

  20?

  100?

Finding Parallelism

- Some some computation that doesn't depend on other computation's results

- Shared Inputs are OK.

Next Time: Can we use 2+ FMACs?

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

Option 1

Stopped here!

cycle	FMAC comp
0	$0.1 \cdot 1 + 0$
1	$0.1 \cdot 2 + 0$
2	$0.1 \cdot 3 + 0$
3	$0.1 \cdot 4 + 0$

Cycle	FMAC 1	FMAC 2
0	$0.1 \cdot 1 + 0$	$0.1 \cdot 2 + 0$
1	$0.1 \cdot 3 + 0$	$0.1 \cdot 3 + 0$

Option 2

→ $0.1 * 1 + 0$	$0.2 * 5 + 0$
$0.1 * 2 + 0$	$0.2 * 6 + 0$
$0.1 * 3 + 0$	$0.2 * 7 + 0$
$0.1 * 4 + 0$	$0.2 * 8 + 0$

reset pop-count in HW?

→ reset w/ MMIO

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

$$0.1 \cdot \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix}$$

$$0.2 \cdot \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} = \begin{bmatrix} 1.1 & 1.4 & 1.7 & 2.0 \end{bmatrix}$$

$$0.3 \cdot \begin{bmatrix} 9 & 10 & 11 & 12 \end{bmatrix} + \begin{bmatrix} 1.1 & 1.4 & 1.7 & 2.0 \end{bmatrix} = \underline{\begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}}$$

Parallize Alternative Dot Computations?

$$\begin{bmatrix} 0.1 & 0.2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

$$\begin{bmatrix} 0, & 0, & 0 \end{bmatrix} \leftarrow \text{result}$$

$$0.1 \cdot \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \leftarrow \text{temp result}$$

$$0.2 \cdot \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} =$$

$$\begin{bmatrix} 0.8 & 1.0 & 1.2 \end{bmatrix} + \begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} = \begin{bmatrix} 0.9 & 1.2 & 1.5 \end{bmatrix}$$

Can we parallelize Dot?

```
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j] + weights[i][j] * inputs[i]
    return outs
```

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.2 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 3.8 & 4.4 & 5 & 5.6 \end{bmatrix}$$

$$\begin{bmatrix} 0.1 & 0.3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 2.8 & 3.2 & 3.6 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 0.2 \end{bmatrix} \begin{bmatrix} 5 & 6 & 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 1.2 & 1.4 & 1.6 \end{bmatrix}$$

$+$

Can we parallelize Dot?

```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j] + weights[i][j] * inputs[i]
    return outs
```

```python
def par_pydot(inputs, weights):
    par_inputs = [inputs[:,::2], inputs[:,1::2]]
    par_weights = [weights[::2,:], weights[1::2,:]]

    par_outputs = [pydot(par_inputs[0], par_weights[0]),
                   pydot(par_inputs[1], par_weights[1])]

    outputs = par_outputs[0] + par_outputs[1]
    return outputs
```

Can we parallelize Dot?

```python
# how its done in dot.sv
def pydot(inputs,weights):
    inputs = inputs[0] # remove outer nesting
    outs = np.zeros(weights.shape[1], dtype=np.float32)
    for i in range(weights.shape[0]): # input length
        for j in range(weights.shape[1]): # output length
            outs[j] = outs[j] + weights[i][j] * inputs[i]
    return outs
```

```python
def par_pydot(inputs, weights):
    par_inputs = [inputs[:,::2], inputs[:,1::2]]
    par_weights = [weights[::2,:], weights[1::2,:]]

    par_outputs = [pydot(par_inputs[0], par_weights[0]),
                   pydot(par_inputs[1], par_weights[1])]

    outputs = par_outputs[0] + par_outputs[1]
    return outputs
```

19: Hardware Acceleration III

Engr 315: Hardware / Software Codesign
Andrew Lukefahr
Indiana University