ECE 498NSU/598NSG:
Deep Learning in Hardware Fall 2020

HW 4:

Abhi Kamboj

Table of Contents:

This is your last homework this semester and it contains 4 problems. Some questions and problems are marked with a star. These are intended for graduate students enrolled in the 598NSG section. Undergraduates enrolled in the 498NSU section are welcome to solve them for extra credit. The topics covered in this homework include: architectural mapping, importance of data reuse, systolic architectures, and algorithm transforms. This homework is due on Oct. 27, by 5pm, via Gradescope. No extension will be provided, so make sure to get started as soon as possible.

Figure 1: The considered hardware architecture in Problem 1.

Problem 1: Architectural Mapping Warm-up

In this problem, consider the simple compute architecture shown in Figure 1, constituting of a processor **P** and a memory **M**. The processor has a peak performance of 450 GMACs/sec with an arithmetic efficiency of 0.5 pJ/MAC. The memory read (write) bandwidth is 50 (20) Mbits/sec while consuming 10 (20) nJ/bit. The processor and memory are linked with a two-way communication bus, which is assumed to be ideal, that is, it has unlimited data transfer throughput and zero energy consumption.

The compute architecture is designed to process neural network layers in a layer by layer fashion. The processing of a layer is sequenced as follows:

- The layer weights and input activations are fetched from **M** to **P**
- **P** executes the computation required by the layer
- The resultant output activations are stored back in **M**

To simplify the problem, we assume that **P** can compute all types of neural network layers (CONV, FC) and that pooling/activation layers are performed automatically before the result is sent back to **M** at zero cost (latency or energy). Furthermore, it is assumed that the input activations can be discarded as soon as the output activations of that layer are written in **M**.

Let's consider mapping AlexNet (see link: https://github.com/pytorch/vision/tree/master/torchvision/models, input size: $3 \times 224 \times 224$) onto this compute architecture.

1.1

1. Assuming all data types are quantized to 8b, what is the minimum allowable size of **M** (in bytes) needed to ensure that the architecture can perform inference without any off-chip data access. Hint: you should consider both activations and weights.

For this problem, I am trying to find the layer with the largest input + output size, then I will add the number of weights and that should determine the minimum size needed.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Activations:
Here we see that with an input image of 224x224, line (8) would have 224x224x384=19,267,584 bytes input and 224x224x256= 12,845,056 bytes output.

Weights:
From the precision profiler code modified for Alexnet:

activations: 355.200K, **weight: 61.101M**, num_dp: 494.184K, macs: 714.188M

Answer:
Largest input bytes + largest output bytes + weights = 19,267,584 + 12,845,056 + 61,101,000 = 93,213,640 bytes needed at minimum for inference

1.2

2. Determine the total energy consumed by the compute architecture for a single inference. How much of the total energy is consumed by:

  - MAC operations:

From the precision profiler code modified for Alexnet:
activations: 355.200K, weight: 61.101M, num_dp: 494.184K, **macs: 714.188M**
And from the problem statement we have an efficiency of 0.5 pJ/MAC.
Energy for Macs = 714,188,000 MACs * .5 pJ/MAC= 357,094,000 pJ = .357 mJ per inference
.000357/(8.5 + .000357 + 0.0791) = 0.00004161102. So MACs consume .0042% of the overall energy.

  - Memory read operations

The number of bytes that need to be read is the number of inputs at each layer + the number of weights.

The number of weights for conv layers is the kernel dimension times the number of input channels. But that is for only one filter, so we need to multiply that by the number of outputs there are because that's how many filters there are.

For each convolutional layer:
inputs = in_channel x 224 x 224
Weights = in_channel x k_size^2 x out_channel

For each linear layer:
Inputs = number_columns
Weights = number_columns x number_rows

After adding all the layers with code I get:
Input Bytes: 45117416 Weight Bytes: 61090496 Total Read Bytes: 106207912
Energy consumption of 10 nJ/bit for read, so total energy = 10 nJ/bit x 106,207,912 bytes x 8 bits/byte = 8,496,632,960 nJ = 8.49663296 Joules
8.5/(8.5 + .000357 + 0.0791) = 0.99073869127, so read operations consume 99% of the total energy consumption

  - Memory write operations

For write operations, we only write the output bytes of each layer. During inference, every dot product corresponds to exactly one output value so we can just use the number of dot products.

From the precision profiler code modified for Alexnet:
activations: 355.200K, weight: 61.101M, **num_dp: 494.184K**, macs: 714.188M

Energy consumption of 20 nJ/bit, so total energy consumption: 20 nJ/bit x 494,184 bytes x 8 bits/byte = 0.07906944 Joules
0.0791/(8.5 + .000357 + 0.0791) = .0092197 so write operations consume .9% of the energy.

1.3

3. Determine the inference latency by the compute architecture. How much of the total latency is due to:

   - MAC operations

     This is largely similar to part 2, except now instead of energy we look at latency.

     For MACS we have: 450 GMACs/sec.

     Total latency = 714,188,000 MACs / (450 x 10 ^9 MACs/sec.) = 0.00158708444 seconds

     0.00158708444/(0.00158708444 + 0.1976736 + 16.9932659) = 0.00009231246, so .009% of total latency is MACs.

   - Memory read operations

     Read Latency = 106,207,912 bytes x 8 bits/byte / (50 x 10^6 bits/sec) = 16.9932659 seconds

     16.9932659/(0.00158708444 + 0.1976736 + 16.9932659) = 0.98841004063, so 98.8% of the latency is due to read operations

   - Memory write operations

     Write Latency = 494,184 bytes x 8 bits/byte / (20 x 10^6 bits/sec) = 0.1976736 seconds

     0.1976736/(0.00158708444 + 0.1976736 + 16.9932659) = 0.0114976469, so 1.15% of the latency is due to write operations

1.4

4. Assume now that a newer memory **M’** has been developed and will replace **M** in our compute architecture. Thanks to its smarter design **M’** can read and write 2× faster

than **M** with the same energy consumption. By how much will the inference latency change with the new memory block?

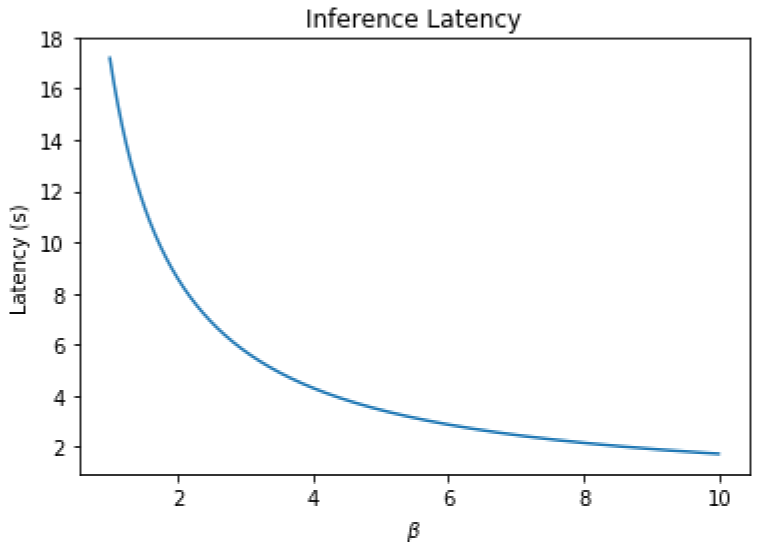
From part 3, we can see the: total latency = MAC latency + read latency + write latency = (0.00158708444 + 0.1976736 + 16.9932659) = 17.1925265844

If we update our memory to be 2x as fast, the read and write latency will decrease by a factor of ½ but the MAC latency will remain the same: (0.00158708444 + 0.1976736/2 + 16.9932659/2) = 8.59705683444.

Thus, the inference latency will decrease by about 8.6 seconds.

1.5

5. If **M'** is instead $\beta\times$ faster than **M** with the same energy consumption. Plot the inference latency as a function of β. What do you observe?



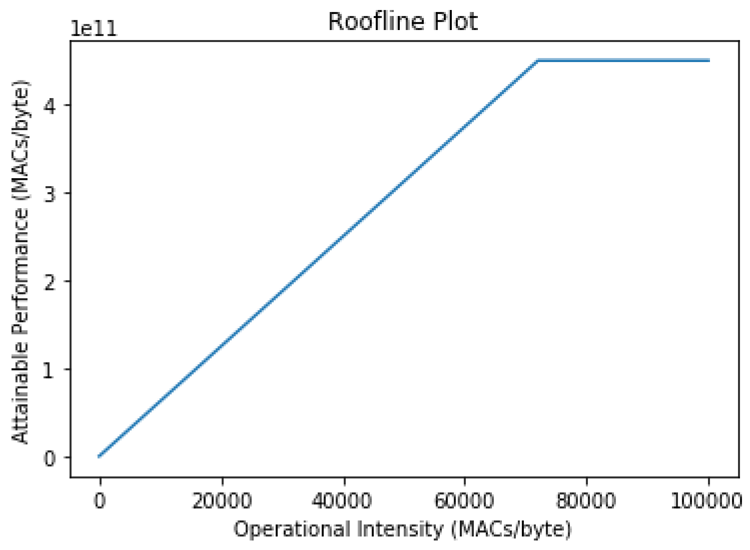
From the graph we can observe that the latency is inversely proportional to beta, ie as beta increases, latency decreases. The graph as well as my answer from part 2 show that the latency is largely dependent on the read operation speed, as so many weights need to be read during inference.

1.6

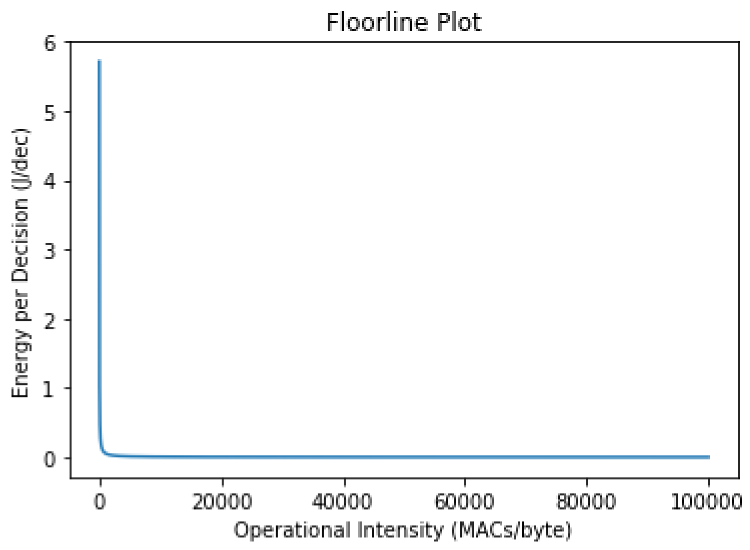
6. Plot the roofline and floorline plots for this architecture. Using the answer to Part 2, estimate the operational intensity (OI) for this workload.

The operational intensity is the number of MACs per byte read, so we have OI = 714,188,000 MACs / 106,207,912 bytes = <u>6.72443311003 MACs/byte</u>

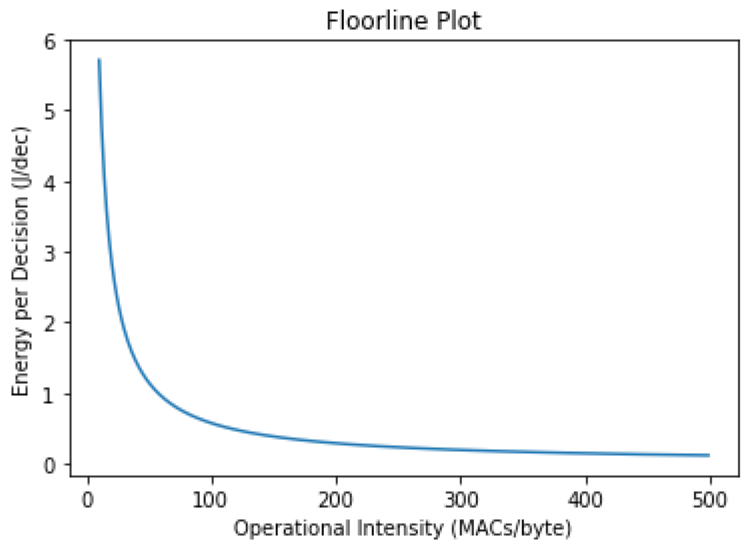
To construct the roofline plot note: BW = 50 Mbits/s / (8bits/byte), PP = 450 GMAC/s, CI = 714,188,000 MACs/decision, and we have attainable performance = min(PP,OI x BW) assuming the architecture is limited by M.



For the floorline plot we have: Ep = .5 pJ/MAC, CI = 714,188,000 MACs , Er = 10 nJ/bit * 8bit/byte = 80 nJ/byte and Edec = CI x Ep (1 + Er/OI)



Floorline plot zoomed in more:



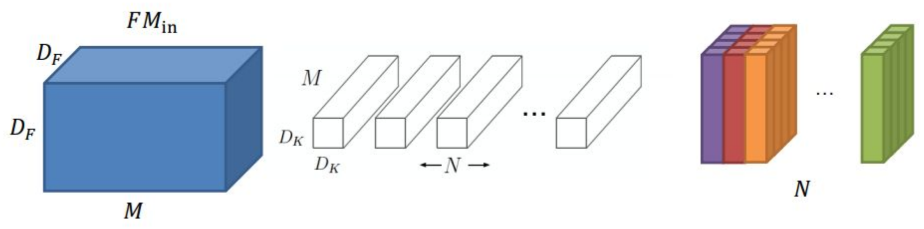
Problem 2: Importance of Data Reuse

In this problem, we will consider again the compute architecture presented in Figure 1. Some of the simplifying assumptions presented in Problem 1 for NN mapping will be removed. Let's consider the fifth convolutional layer (**C5**) in VGG-16 (see https://github.com/pytorch/vision/tree/master/torchvision/models), which comprises of 256 kernels, each of size 3 × 3 × 128 (we will ignore the bias in this problem) and takes in input feature maps of size 56 × 56 × 128. Unless specified otherwise, assume all datatypes are quantized to 8b.

2.1

1. Calculate the amount of data reuse for each of the input activations and weights. For a given datatype, the amount of data reuse is defined as the number of MACs that each data value is used for (i.e., MACs/data).

We can use the equations from lecture 2:



- Kernel size: $D_K^2 \times M \times N$; Assuming stride of 1 below
- Computational cost in terms of # of MACs (multiply-accumulates) = $D_K^2 \times M \times N \times D_F^2$
- Each weight contributes to D_F^2 outputs = "weight reuse factor"
- Each input contributes to $D_K^2 \times N$ outputs = "data reuse factor"

Where Df = 56, Dk = 3, M = 128, N =256

weight reuse = 9 MACs/byte, and data reuse = 56 x 56 x 256 = 802,816 MACs/byte

2.2

2. Assuming that there is zero input and weight data reuse in our architecture, that is, for every MAC operation that needs to be computed, both operands (weights and input activations) have to be read from the same external memory **M** (all partial sums generated by these operations are locally buffered and are readily available). If **P** has a peak MAC throughput of 450 GMACs/sec, what is the minimum required memory read bandwidth (bits/sec) needed to maximize utilization in **P** while executing **C5**?

We can see that every MAC operation uses 2 bytes which are multiplied and then added to an accumulator, so the memory needs to pull 2 bytes of data in the time it takes for the processor to do a MAC. Hence we have Read bandwidth = 450 x 10^9 MACs/sec x 2 bytes/MAC x 8 bits/byte = <u>7.2 x 10^12 bits/sec.</u> This would provide enough data from memory to allow for **P** to run at 450 GMAC/sec.

2.3

3. Assume now that **P** is equipped with an internal buffer that can store all the weights of **C5** for instant data access. The internal buffer allows us to reuse weights across different operations. What is the size (in bytes) needed for the internal buffer to fit the weights? What is the minimum required memory read bandwidth (bits/sec) needed to maximize utilization in **P** while executing **C5**?

For each kernel we would need to store 3 x 3 x 128 = 1,152 weights and we have 256 kernels so to store all the weights of C5 for instant data access we would need a buffer of size: 1,152 x 256 = <u>294,912 bytes.</u>

Now since the processor can store data locally it has more data to work with while the memory is retrieving the next batch of data so the memory can run slower than in part 2. Notice that we are ONLY storing the weights so we still have to retrieve the 1 byte of input data from the memory every time we do a MAC.

Read bandwidth = 450 x 10^9 MACs/sec x 1 bytes/MAC x 8 bits/byte = <u>3.6 x 10^12 bits/sec</u>

If we assume that there are MAC accumulators (partial sums) for every output pixel being stored locally in the processor, then we know that Dk^2 x N weights would touch every pixel and we can multiply and accumulate each of those pixels with the byte input we retrieved from the memory. Thus, Read bandwidth = 450 x 10^9 MACs/sec x 1 bytes / (9 x 256) MACs x 8 bits/byte = 1,562,500,000 bits/sec = 1.6 Gbits/sec

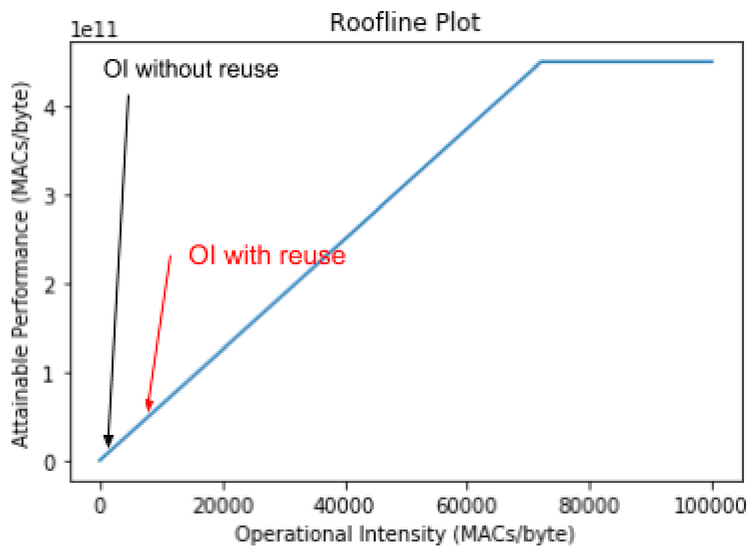
2.4

4. Mark the operating point of the architecture on the roofline plot of Problem 1 assuming: a) no data reuse as in Part 2, and b) with data reuse from Part 3.

Are we using the architecture in problem 2? If so, wouldn't the data reuse be different for each layer so would we be marking it differently for every layer?

Operational intensity = number of MACs per bytes read. Assuming no data reuse, we would do 1 MAC for every byte we read, so <u>OI = 1 MAC/byte</u>.

Assuming we do have data reuse and that there are MAC accumulators (partial sums) for every output pixel being stored locally in the processor, we would do Dk^2 x N = 9 x 256 = 2,304 MACs for every byte of data read, so <u>OI = 2,304 MAC/byte</u>.



2.5

5. Would it be feasible to internally buffer input activations instead of weights? Briefly justify your answer.

Number of weights: 9 x 128 x 256 = 294,912 bytes

Number of inputs: 56 x 56 x 128 = 401,408 bytes

It really depends on the size of the cache. According to this website: "As far as the size goes, the L1 cache typically goes up to 256KB. However, some really powerful CPUs are now taking it close to 1MB". Thus if we were using a 256 KB L1 cache then buffering the inputs would not be feasible as we would have twice as many inputs as our buffer can hold. If we had a powerful CPU with a 1MB L1 cache, then it would be very feasible to hold our 401 KB of data input data in the buffer.

Figure 2: The considered hardware architecture in Problem 3.

Problem 3: A 2D Systolic Architecture

Consider the 2D systolic architecture (SA) presented in Figure 2, which consists of:

- A mesh of 3 × 3 processing elements (PEs) **P0**,...,**P8**. Each PE contains its own local storage via a scratchpad which can be used to store **two data values** at a time, and can communicate with its neighboring PEs to send and receive one datatype each cycle (weights, inputs, or output partial sums). To be more precise, at each cycle, a single PE will first receive data from its neighbors (if any), perform one MAC operation, and send out one datatype to its neighbors (if required).
- A main memory block **M** which is directly connected to **P0**, **P3**, and **P6**. **M** can also send and receive one datatype each cycle.

We would like to implement a simple 2D convolution using this proposed SA. The 2D convolution operates on a 5 × 5 input feature map, uses one 3 × 3 filter, and generates a 3 × 3 output feature map.

Note that output partial sums cannot be sent to **M**. In other words **M** can only store inputs, weights, and complete outputs. Furthermore, the execution of the convolution is considered complete when the last output is stored back into **M**. Assume that the contents of the PEs are *empty* before execution, and that **M** contains both the inputs and weights.

Three engineers proposed three different mappings listed below. For each mapping, Calculate the *number of cycles needed to finish execution*.

3.1

1. Mapping A: The first engineer noticed that the PE mesh has the same size as the output that needs to be computed, and proposed that each PE should process one output pixel locally. That is, partial sums are accumulated locally in each PE, and only weights/inputs are shared amongst PEs.

Output stationary:

a) Add all inputs for the first weight: 3 cycles
b) Broadcast first weight across immediately followed by the next set of inputs: 4 cycles (first cycle carry the same weight value in a column and the next 3 cycles shifting in the next set of inputs)
c) Repeat step b) for 9 times (one for each of the weights). Note: during the last round you can just shift the w9 into the first column (1 cycle) and as you shift the partial sums back to memory doing the MAC for w9 (3 cycles)

Overall: 3 + 9x4 = 39 cycles

3.2

2. Mapping B: The second engineer noticed that the PE mesh has the same size as the filter being used, and proposed that each PE should store one filter weight locally and share the other two datatypes (inputs and partial sums) amongst PEs.

Here we can imagine that the PE mesh is the filter sliding across the inputs. This is weight stationary.

a) First load all the weights into the PEs: 3 cycles
b) Load all the inputs into the PEs and do the partial sum: 3 cycles
c) Accumulate all the partial sums and send to memory: 4 cycles
d) Repeat steps b) and c) 9 times total for each output (so 8 more times)

Overall: 3 + 9x(3+4) = 66 cycles

3.3

3. Mapping C: The third engineer was concerned about frequent fetches of input features from **M**, and proposed loading a 3 × 3 patch of inputs onto the SA to process them completely. Once the corresponding output has been computed, another 3 × 3 input patch is sent again and the process is repeated.

This is like an input stationary method.

a) First load first set of inputs: 3 cycles
b) Next load all the weights: 3 cycles
c) Accumulate partial sums: 4 cycles
d) Now assuming we don't store weights in the PEs then we would repeat steps a) b) c) 9 times total. If we did store weights locally this would be just like weight stationary in part 2 mapping B.

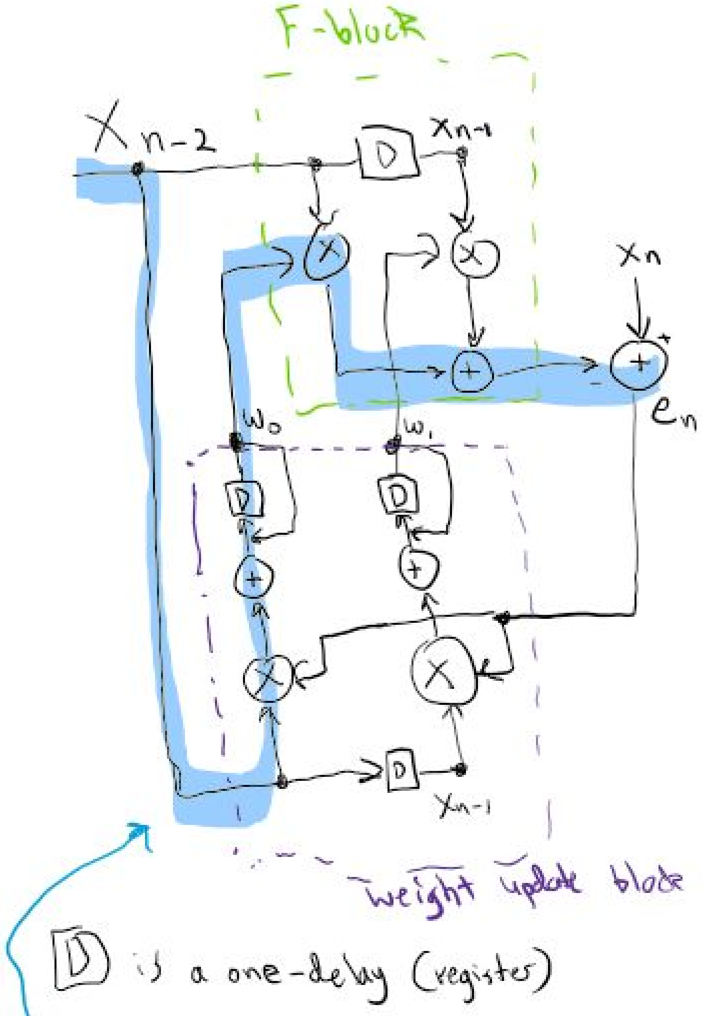
Overall: 9x(3+3+4) = 90 Cycles

Problem 4: Algorithm Transforms *

Consider a 2-tap SGD-based adaptive time series predictor. Assume $T_M = 2T_A$, where T_M and T_A are the multiply and add delay times for all multipliers and adders (including the one in the WUD-block).

4.1

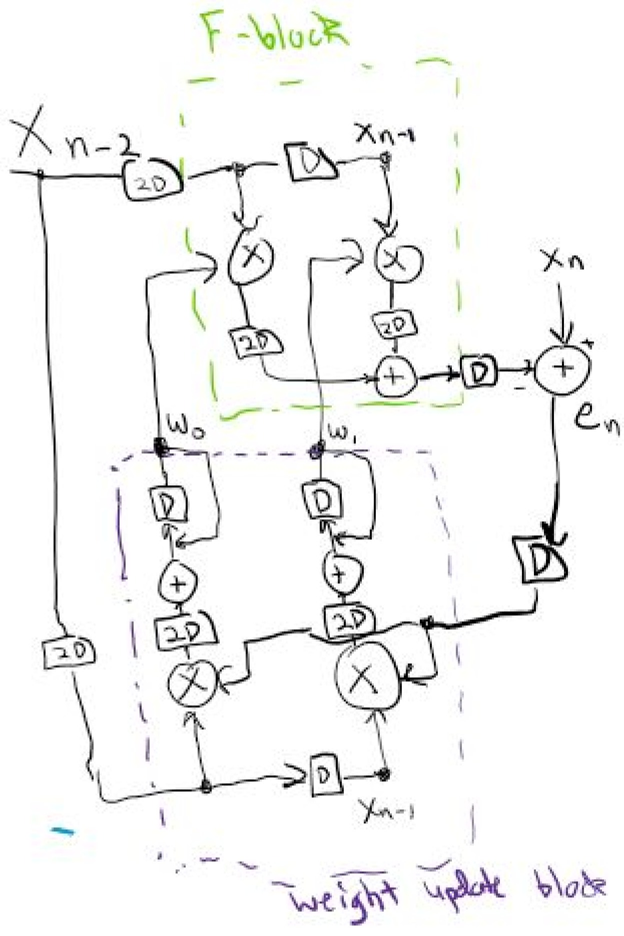
1. Draw the architecture of the SGD-based predictor. Mark the critical path and write an expression for the critical path delay T_{cp} in terms of T_A. This is the serial architecture.



$T_{cp} = 3T_A + 2T_m = 7T_A$

2. Pipeline and retime the serial architecture so that the critical path delay is T_A. The resulting architecture should have 2Ds at the output of each multiplier and 1D at the

output of each adder after retiming. What is the minimum value of M_1 needed to achieve this level of pipelining?



$M_1 = 2$

3. What is the critical path delay in terms of T_A for a 2-unfolded version of the serial architecture?

2-unfolding increases Tcp by a factor of 2, since unfolding preserves the number of delays, and doubles the number of nodes. So Tcp = 14T_A

4. Draw a 2-unfolded version of the serial architecture and mark the critical path.