



Bridge of Life  
Education

# NN Hardware

Lecturer: Hua-Yang Weng

Date: 2022/4/13

[FPGA'17: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference]  
(<https://arxiv.org/abs/1612.07119>)

# From AI to Gate Textbook

## From AI to Gate

Preface ▾

Getting Started ▾

Network Define ▾

Compiler ▾

Verification ▾

**NN Hardware** ▲

### Introduction

Architecture

BNN Algorithm

Fully Connected Layer

Convolution Layer

Pooling Layer

HLS ▾

Case Study ▾

Code Repository ▾

## NN Hardware

In this Chapter, we are going to explain the neural network hardware part. This is based on the Xilinx paper "FINN: A Framework for Fast, Scalable, Binarized Neural Network Inference." This link is listed in the references.

The objective of the hardware part is to leverage FINN compiler and build fast, flexible FPGA accelerators. The heterogeneous streaming methodology further optimized the throughput of this neural network inference hardware. Although this paper focuses on binarized neural networks and their optimized mapping strategies as we will explain in the following sections, the FINN compiler itself can support more general quantized neural networks.

The sections in this chapter are organized as follows. We will first explain the overall architecture of the hardware, then we dive into the algorithms of mapping binarized layers in section 2. The last two sections correspond to the implementation of the fully connected layers and convolution layers.

## References:

- [FPGA'17: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference](#) ↗
- [FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks](#) ↗

← Previous

Next →

## Table of Contents

**NN Hardware**

References:

# Overview

- Introduction
- BNNs inference hardware architecture
- BNN specific Operator Optimizations
- FINN Design Flow and Hardware Library
- Folding
- Evaluation

# Overview

- Introduction
- BNNs inference hardware architecture
- BNN specific Operator Optimizations
- FINN Design Flow and Hardware Library
- Folding
- Evaluation

# Neural Networks in Hardware(1/3)

- Single processing engine
    - Systolic array like processor
  - **Streaming architecture**
    - Dedicated hardware per layer
  - Vector processor
    - Process with instructions
  - Neurosynaptic processor
    - Neurosynaptic like digital neurons and interconnections
- Recent networks evolved very fast
  - New categories of networks coming up while some gradually vanished.

To push energy efficiency:

  - Zero-skipping
  - Weight pruning
  - ...

Replaces simple systolic array to customized computing algorithm.

# Neural Networks in Hardware(2/3)

- Weight stationary
- Output stationary
- Input stationary
- (Row) stationary

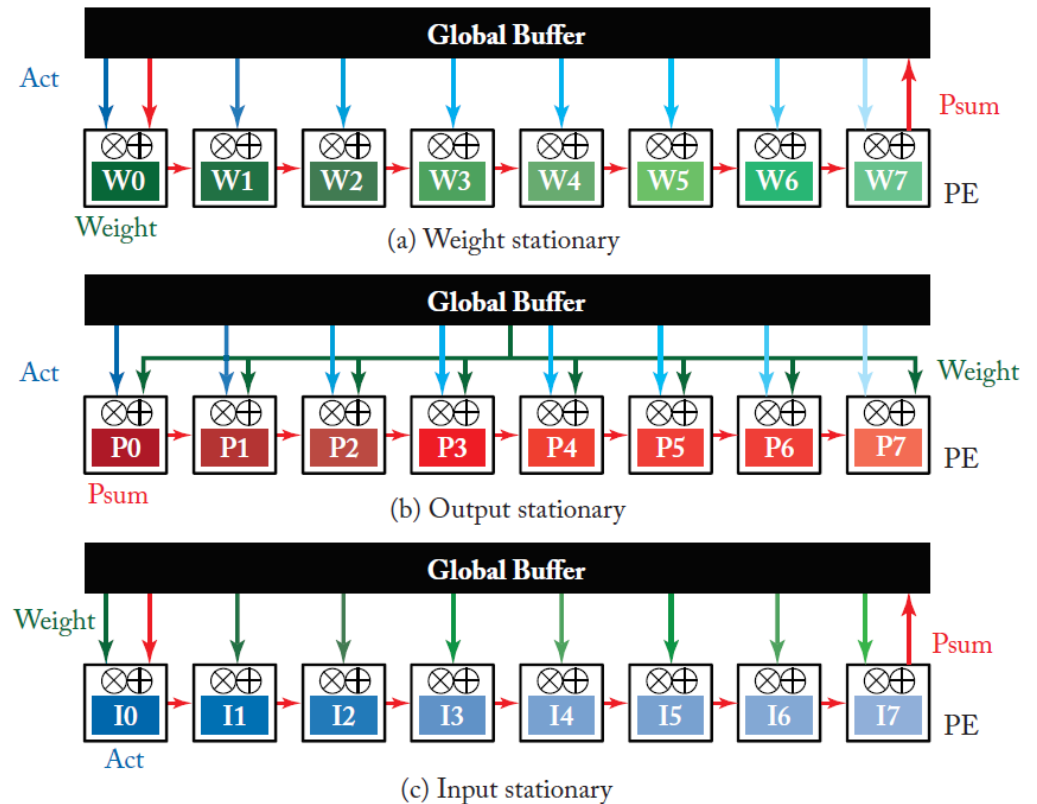
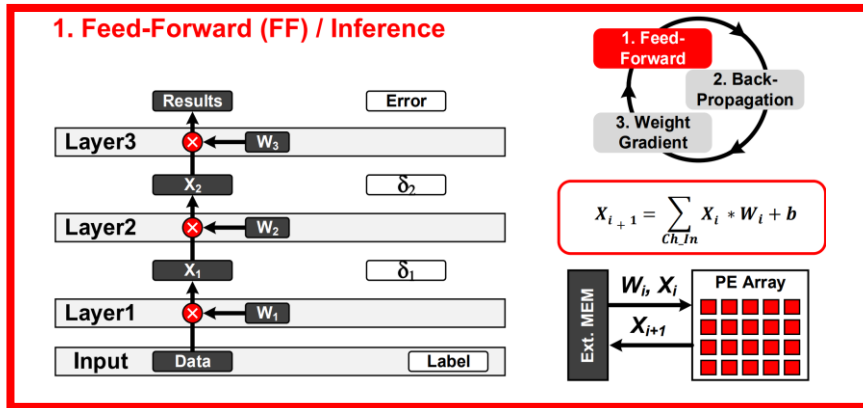


Figure 5.15: The taxonomy of commonly seen dataflows for DNN processing. *Act* means input activation. The color gradient is used to note different values of the same data type.

# Neural Networks in Hardware(3/3)

## • Inference Hardware

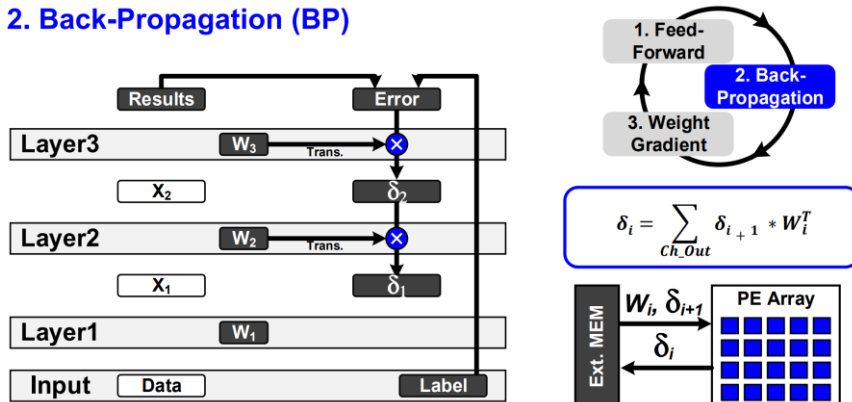
### 1. Feed-Forward (FF) / Inference



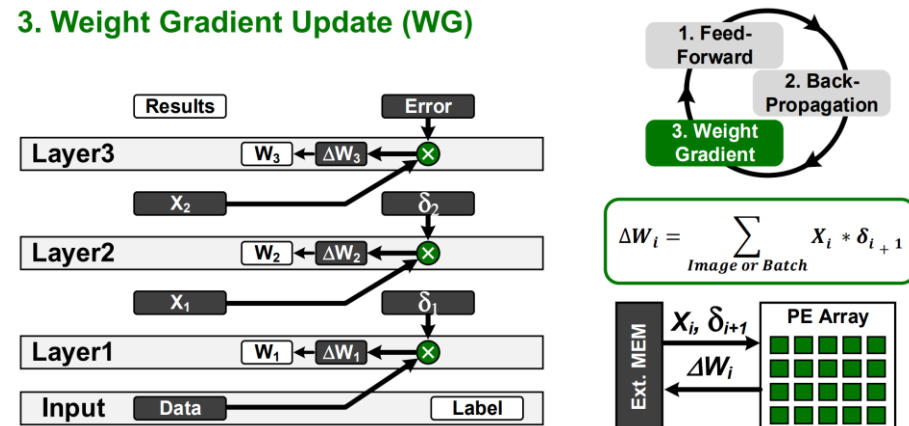
FINN is currently designed for inference hardware

## • Training Hardware

### 2. Back-Propagation (BP)



### 3. Weight Gradient Update (WG)



# Roofline model

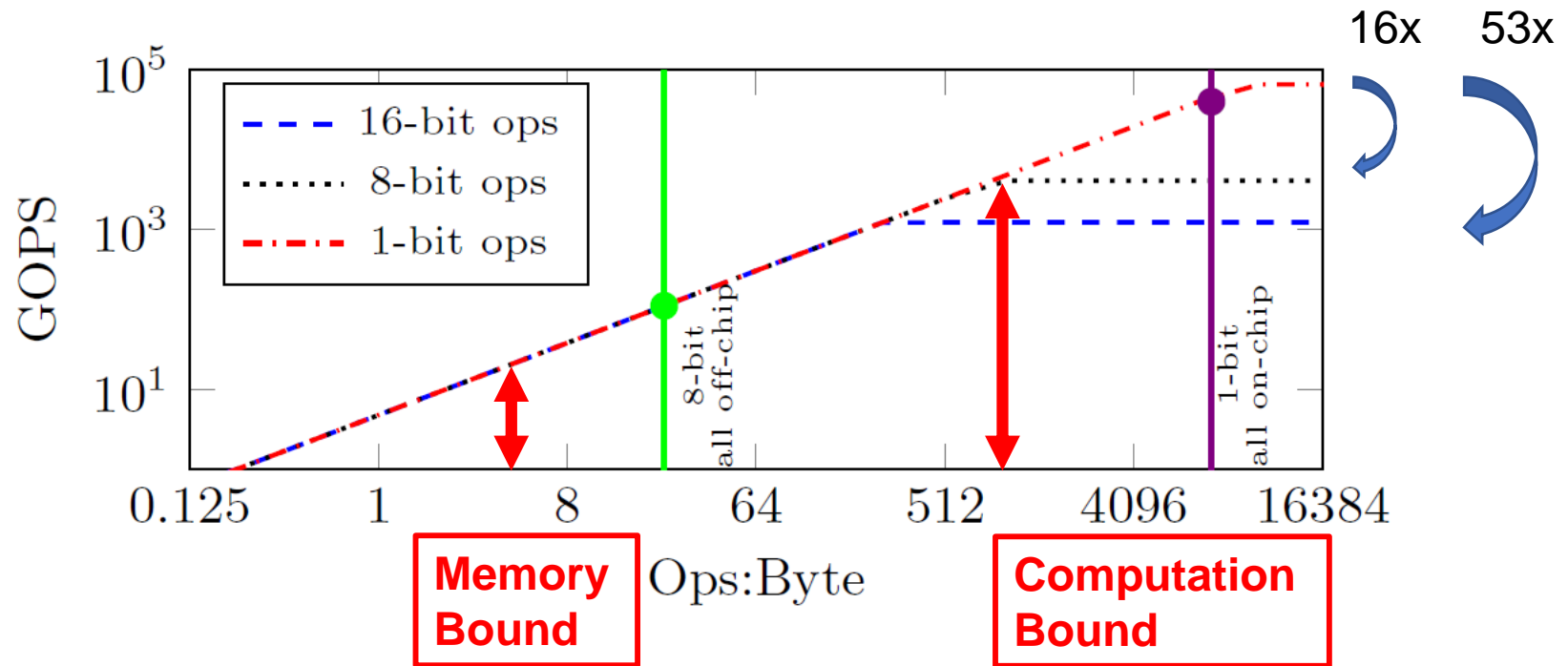


Figure 1: Roofline model for a ZU19EG.



# Accuracy–Computation Tradeoffs

- Fix to 3 fully connected layers
- Scaling the number of neurons in each layer
- As the **network size increases**, the **difference in accuracy** between low precision networks and floating point networks **decreases**

Table 1: Accuracy results - BNN vs NN.

Neurons/layer	Binary Err. (%)	Float Err. (%)	# Params	Ops/frame
128	6.58	2.70	134,794	268,800
256	4.17	1.78	335,114	668,672
512	2.31	1.25	932,362	1,861,632
1024	1.60	1.13	2,913,290	5,820,416
2048	1.32	0.97	10,020,874	20,029,440
4096	1.17	0.91	36,818,954	73,613,312

# Overview

- Introduction
- **BNNs inference hardware architecture**
- BNN specific Operator Optimizations
- FINN Design Flow and Hardware Library
- Folding
- Evaluation

# FINN Hardware Architecture

- Custom architecture for each layer
  - Rather than scheduling a operations to a fixed architecture
- Separate compute engines are dedicated to each layer
- All neural network **parameters** are kept in **on-chip memory**

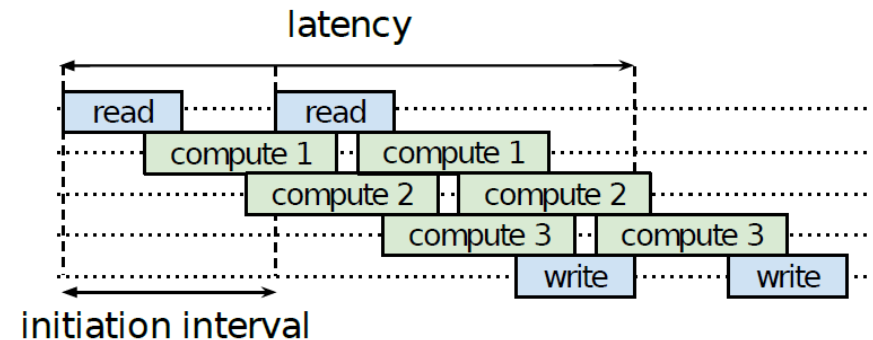
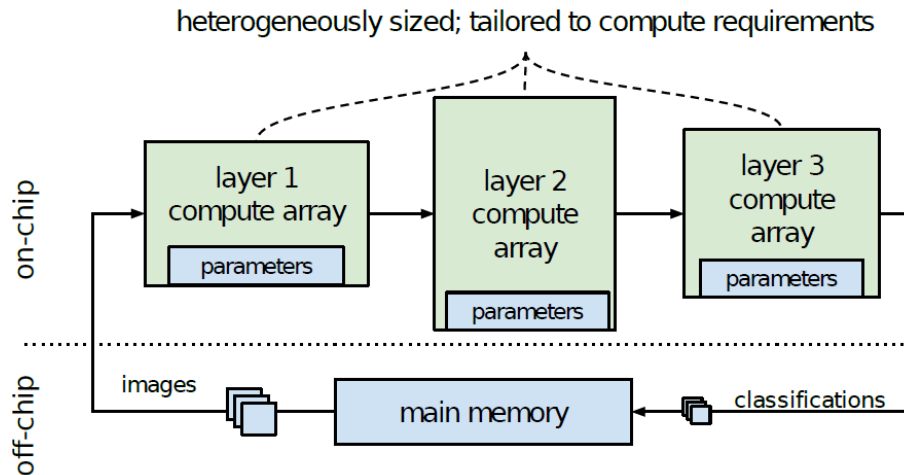


Figure 2: Heterogeneous streaming.

# Overview

- Introduction
- BNNs inference hardware architecture
- **BNN specific Operator Optimizations**
  - Popcount for Accumulation
  - Batchnorm-activation as Threshold
  - Boolean OR for Max-pooling
- FINN Design Flow and Hardware Library
- Folding
- Evaluation

# BNN-specific Operator Optimizations

- Using 1-bit bipolar values for all input activations, weights and output activations (full binarization)
  - **Unset bit** represents **-1** and a **set bit** represents **+1**.
- Batch normalization prior to the activation function.
- Using the following activation function:

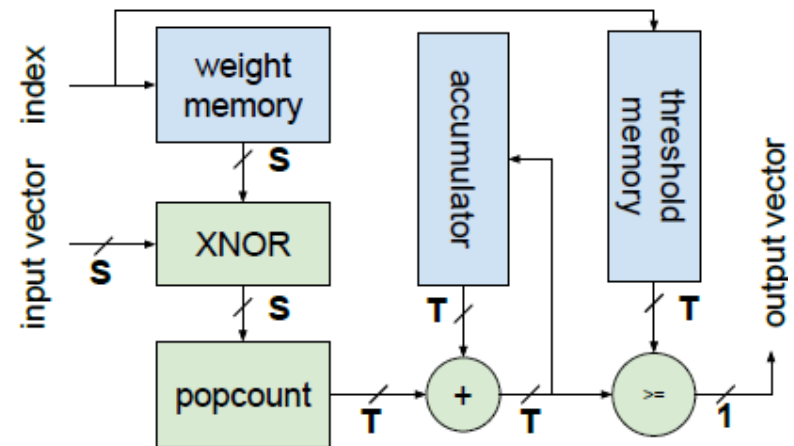
$$\text{Sign}(x) = \{+1 \text{ if } x \geq 0; -1 \text{ if } x < 0\}$$

## ***Some additional optimizations***

1. Popcount for Accumulation
2. Batchnorm-activation as Threshold
3. Boolean OR for Max-pooling

# Optimizations: Popcount for Accumulation

- XNOR gate to compute bipolar multiplication
  - $(1, 1) \rightarrow 1 \quad 1 \times 1 = 1$
  - $(1, 0) \rightarrow 0 \quad 1 \times -1 = -1$
  - $(0, 1) \rightarrow 0 \quad -1 \times 1 = -1$
  - $(0, 0) \rightarrow 1 \quad -1 \times -1 = 1$
- Binary dot product: **popcount** (counts the number of set bits)
  - Instead of signed arithmetic.
- Resource utilization
  - Comparing signed-accumulate.
  - LUT and FF resources x 0.5



## Optimizations: Batchnorm-activation as Threshold

- $a_k$ : Dot product (pre-activation) output of neuron k
- $\theta_k = (\gamma_k, \mu_k, i_k, B_k)$ : Batch normalization parameters
- $a_k^b = \text{Sign}(\text{BatchNorm}(a_k, \theta_k))$ : Output of this neuron
- $\text{BatchNorm}(a_k, \theta_k) = \gamma_k(a_k - \mu_k)i_k + B_k$ : BatchNorm

$\gamma_k, B_k$ : Learnable affine parameters

$a_k$ : input data

$\mu_k$ : mean of the data

$i_k$ : equals to  $\frac{1}{\sqrt{\text{Var}[a_k] + \epsilon}}$

A threshold  $\tau_k$  for the output activation is always present.

- Solving  $\text{BatchNorm}(a_k, \theta_k) = 0$  to deduce that
 
$$\tau_k = \mu_k - B_k / (\gamma_k \cdot i_k)$$
- Avoid computing the batch normalized value during inference  
compute the output activation using an unsigned comparison

# Optimizations: Boolean OR for Max-pooling

- Origin: activations after max-pooling
- FINN: max-pooling after the activations
- $a_1, a_2, \dots, a_Y$ : positive dot product outputs

$$a^b = \text{Max}(a_1, a_2, \dots, a_Y) > \gamma^+$$

- Distributivity of  $\text{Max}(\cdot)$   
$$a^b = (a_1 > \gamma^+) \vee (a_2 > \gamma^+) \dots \vee (a_Y > \gamma^+)$$
- As the threshold comparisons are already computed for the activations, max-pooling can be effectively implemented with the Boolean OR-operator



# Overview

- Introduction
- BNNs inference hardware architecture
- BNN specific Operator Optimizations
- **FINN Design Flow and Hardware Library**
  - Matrix–Vector–Threshold Unit (MVTU)
  - Convolution: The Sliding Window Unit (SWU)
  - Pooling Unit (PU)
- Folding
- Evaluation

# FINN Design Flow and Hardware Library(1/2)

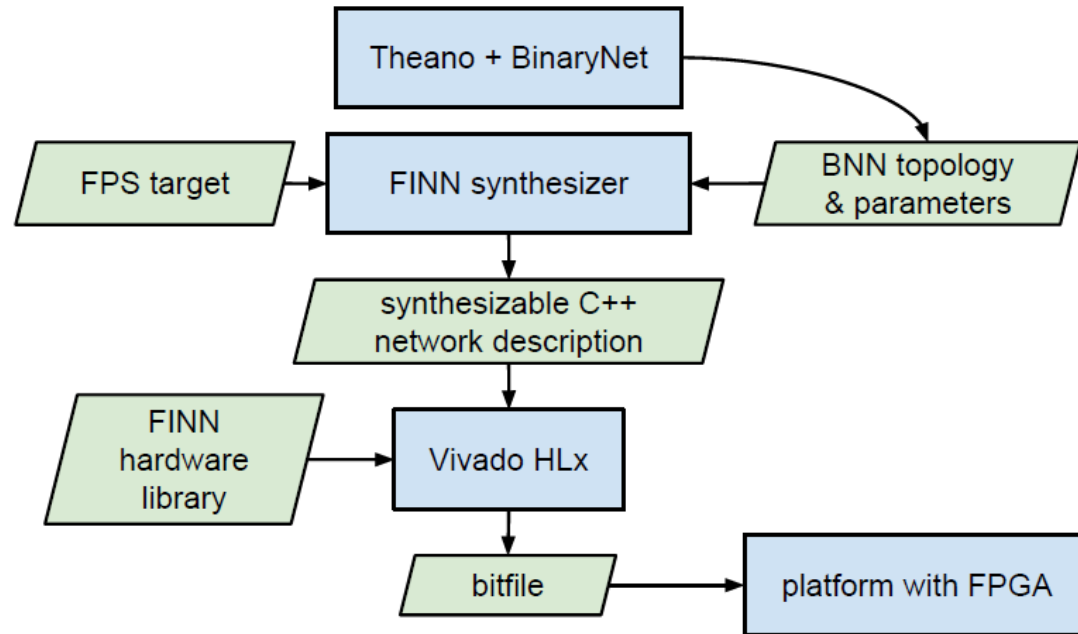
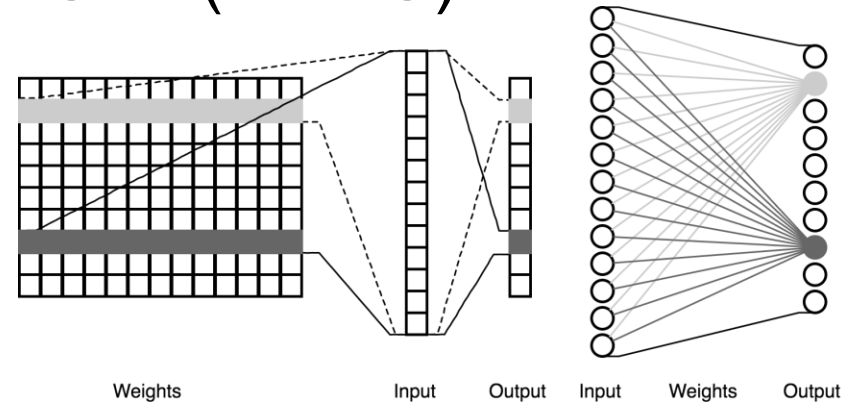


Figure 4: Generating an FPGA accelerator from a trained BNN.

## FINN Design Flow and Hardware Library(2/2)

### 1. Matrix–Vector–Threshold Unit (MVTU)

- Fully-connected layer
- Matrix-vector multiplication



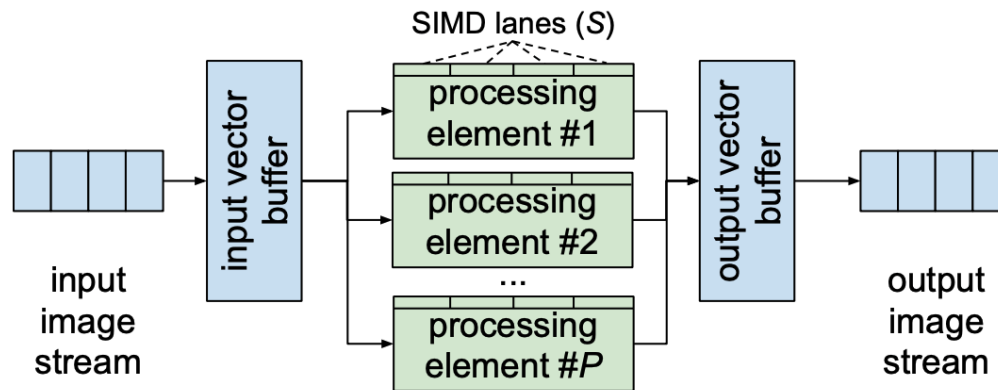
### 2. Convolution: The Sliding Window Unit (SWU)

- Convolutions can be lowered to matrix-matrix multiplications

### 3. Pooling Unit (PU)

# 1. Matrix Vector Threshold Unit (MVTU)

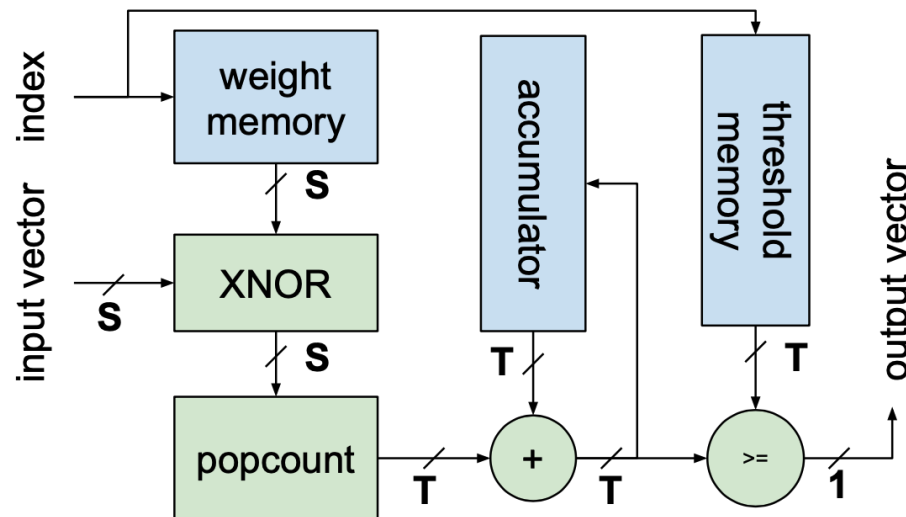
- Forms the computational **core** for our accelerator designs
- BNN can be expressed as matrix–vector operations



The number of PEs (**P**) and number of SIMD lanes (**S**) are configurable to control throughput

# 1. Matrix Vector Threshold Unit (MVTU): PE datapath

- ① Fan-in  $S$  XNOR computes dot product
- ② Popcount with bitwidth  $T = 1 + \log_2 S$
- ③ Accumulate the partial inner-product value
- ④ compares the result to a threshold  $\tau_k^+ \rightarrow 1$  bit output



The weights are stored  
in On-Chip Memory

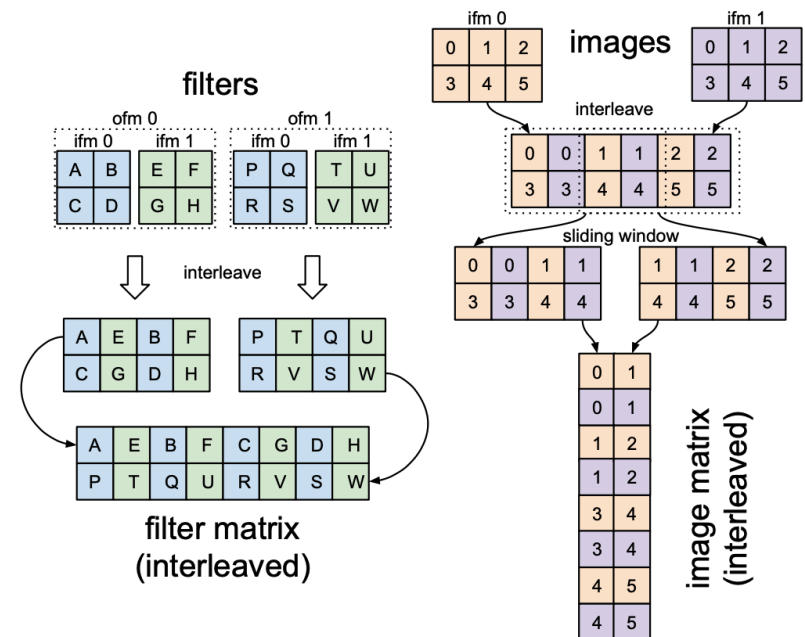
Weight stationary  
Output stationary

## 2. Convolution: The Sliding Window Unit

- Convolutions can be lowered to matrix-matrix multiplications
- Interleave the feature maps
- Each pixel contains all the Input Feature Map (IFM) channel data for that position

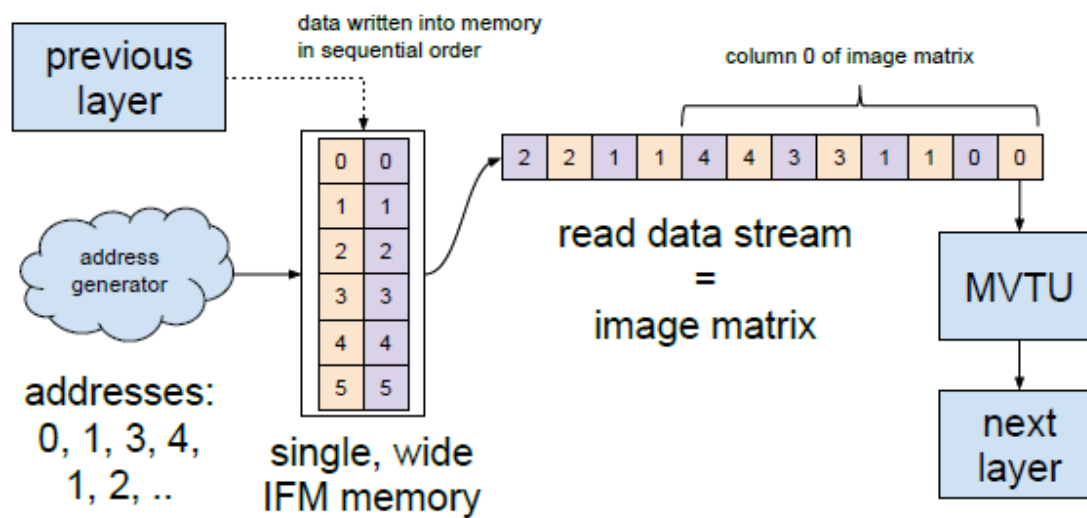
This fashion allows:

- implement the SWU with a single wide OCM instead of multiple narrow OCMs
- Enables the output of the MVTU to be directly fed to the next layer without any transposition.

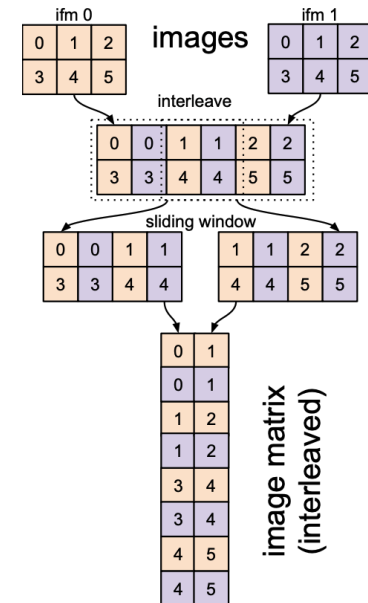


## 2. Convolution: The Sliding Window Unit

- Filter matrix interleaving is computed offline
- The memory locations corresponding to each sliding window
- Read out previous layer to produce the image matrix.

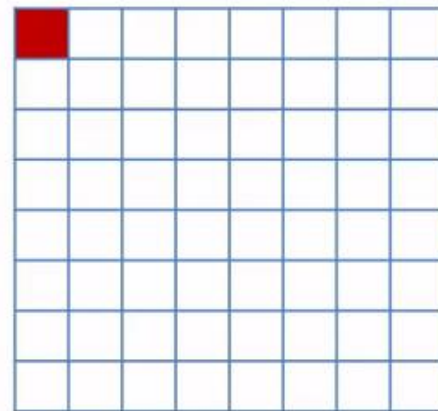


(b) SWU operation.



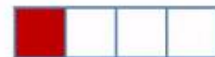
### 3. Pooling Unit (PU)

- $k \times k$  max-pooling kernel
- $D_H \times D_W \times C$  binary feature map
- A  $D_H/k$  line buffer with  $D_W$  bits.

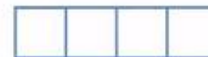


Input Feature map: 7x7  
Max pooling kernel: 2x2

Line buffer



output FIFO



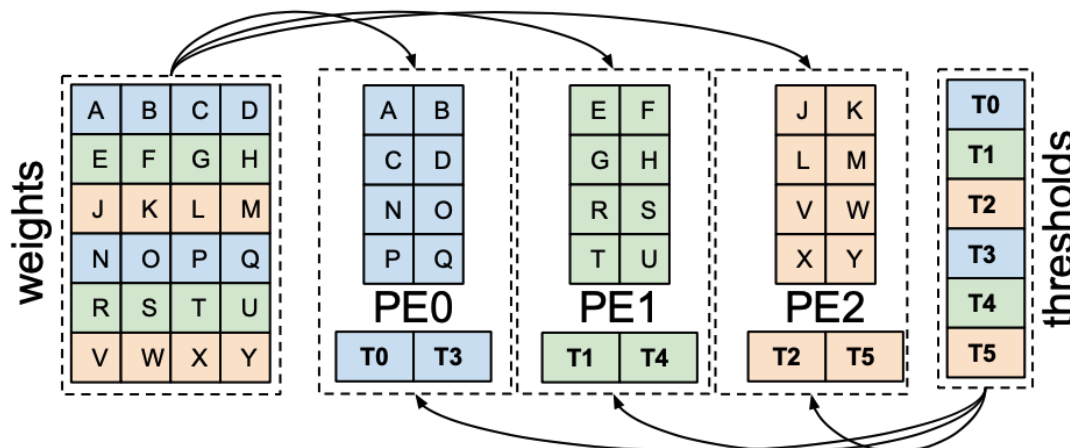


# Overview

- Introduction
- BNNs inference hardware architecture
- BNN specific Operator Optimizations
- FINN Design Flow and Hardware Library
- **Folding**
  - Folding Matrix–Vector Products
  - Determining  $F^n$  and  $F^s$
- Evaluation

# Folding Matrix-Vector Products

- hardware resources on an FPGA is limited
- Use time-multiplex (or fold) save hardware
- Folding is achieved by controlling two parameters of the MVTU
  - P: the number of PEs
  - S: the number of SIMD lanes per PE



P: 3

S: 2

Matrix: 6x4

Fold neuron: 6/3

Fold synapse: 4/2

$$F_n \cdot F_s = (6/3) \cdot (4/2) = 4 \text{ cycles.}$$

# Determining $F^n$ and $F^s$

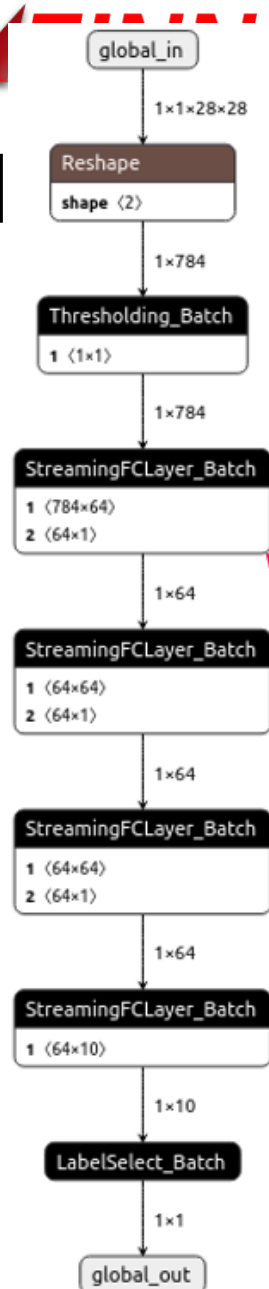
- Avoiding the “one-size-fits-all”
- Guiding principle: **rate-balancing**
- **Slowest layer (with  $II_{max}$ ) will determine the overall throughput**
- For this streaming system,  $FPS \approx \frac{F_{clk}}{II_{max}}$
- Balancing a fully-connected BNN can be achieved by using  $F^n$  and  $F^s$  such that  $F^n \cdot F^s = \frac{F_{clk}}{FPS}$  for each layer.
- **Match the throughput of all other layers to the bottleneck**

# Examples: 4-Layer Fully-connected

- The PE, SIMD factors are set in a manner
  - What is the II for each layer?

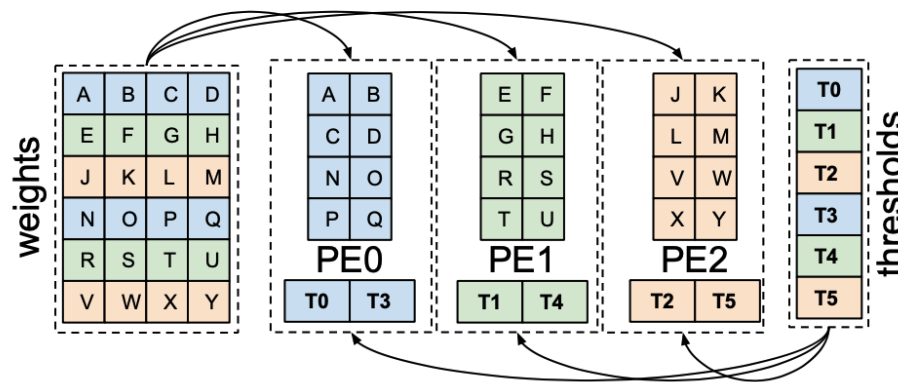
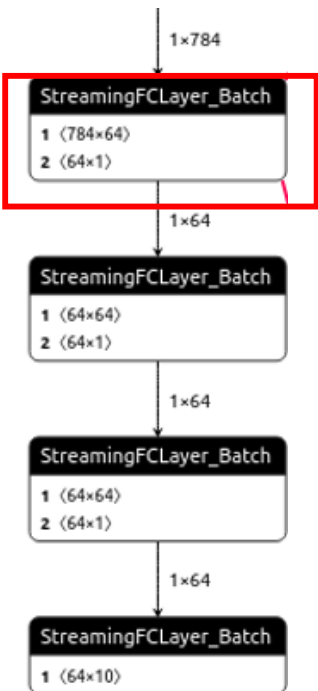
```
fc_layers = model.get_nodes_by_op_type("StreamingFCLayer_Batch")
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
]
for fcl, (pe, simd, ififo, ofifo, ramstyle) in zip(fc_layers, config):
    fcl_inst = getCustomOp(fcl)
    fcl_inst.set_nodeattr("PE", pe)
    fcl_inst.set_nodeattr("SIMD", simd)
    fcl_inst.set_nodeattr("inFIFOdepth", ififo)
    fcl_inst.set_nodeattr("outFIFOdepth", ofifo)
    fcl_inst.set_nodeattr("ram_style", ramstyle)

# set parallelism for input quantizer to be same as first layer's SIMD
inp_qnt_node = model.get_nodes_by_op_type("Thresholding_Batch")[0]
inp_qnt = getCustomOp(inp_qnt_node)
inp_qnt.set_nodeattr("PE", 49)
```



# Examples: 1<sup>st</sup> Layer

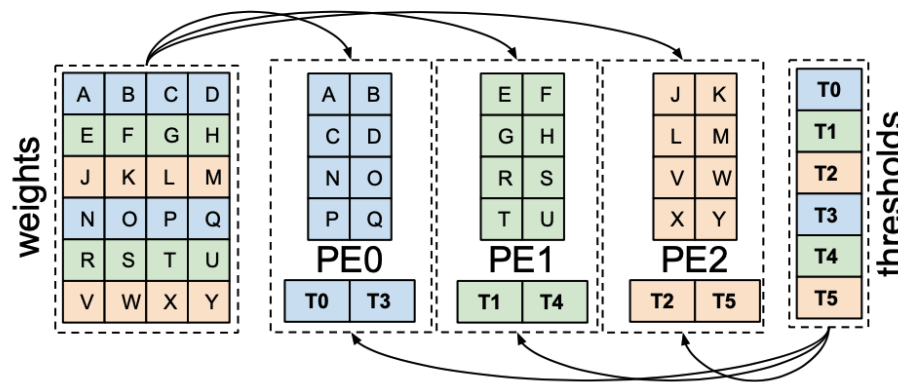
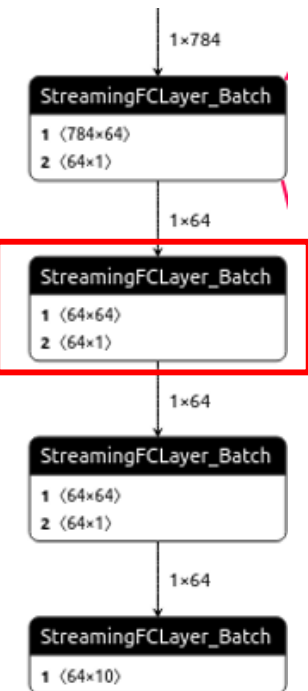
```
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
```



P: 16  
S: 49  
Matrix: 64x784  
Fold neuron: 64/16  
Fold synapse: 784/49  
 $F_n \cdot F_s = (64/16) \cdot (784/49)$   
 $= 4 \times 16$   
 $= 64 \text{ cycles}$

# Examples: 2<sup>nd</sup> Layer

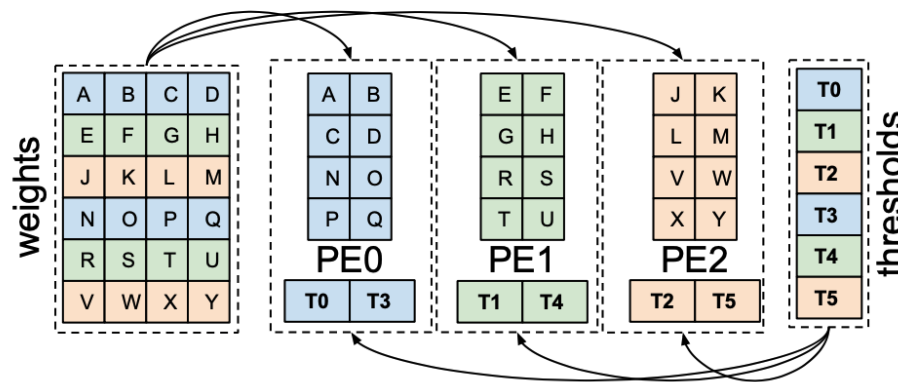
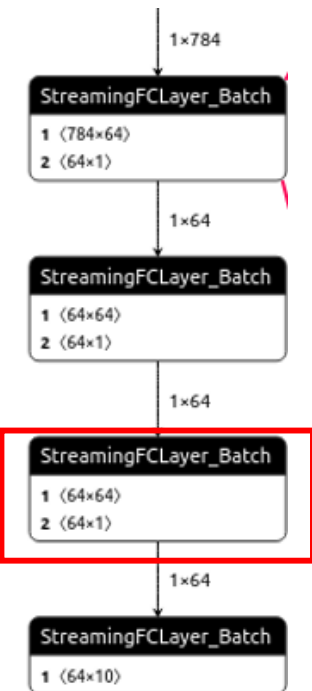
```
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
]
```



P: 8  
 S: 8  
 Matrix: 64x64  
 Fold neuron: 64/8  
 Fold synapse: 64/8  
 $F_n \cdot F_s = (64/8) \cdot (64/8)$   
 $= 8 \times 8$   
 $= 64 \text{ cycles}$

# Examples: 3<sup>rd</sup> Layer

```
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
```

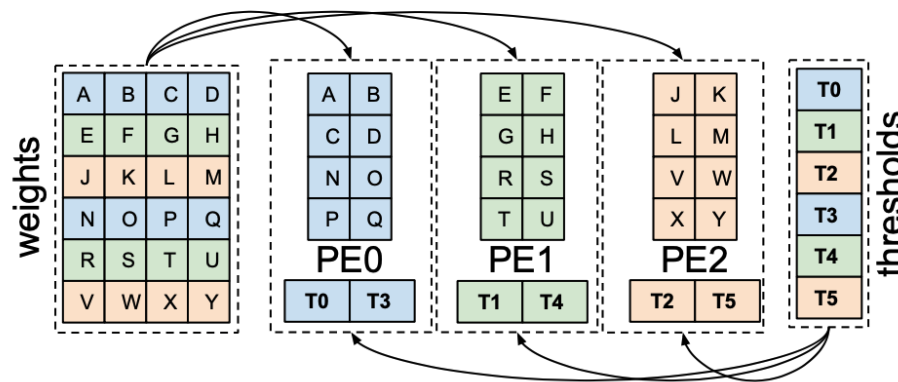
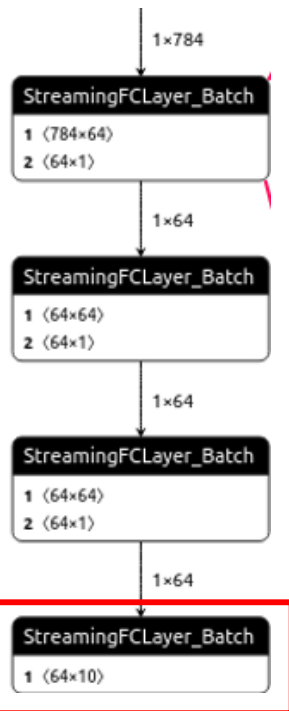


P: 8  
S: 8  
Matrix: 64x64  
Fold neuron: 64/8  
Fold synapse: 64/8

$$F_n \cdot F_s = (64/8) \cdot (64/8) = 8 \times 8 = 64 \text{ cycles}$$

# Examples: Last Layer

```
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (8, 8, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
```



P: 10  
S: 8  
Matrix: 10x64  
Fold neuron: 10/10  
Fold synapse: 64/8

$$F_n \cdot F_s = (10/10) \cdot (64/8)$$

$$= 1 \times 8$$

$$= 8 \text{ cycles}$$



# Examples:

- The PE, SIMD factors are set in a manner s.t.  $II = 64$  for each layer (except the last)

```
# change this if you have a different PYNQ board, see list above
pynq_board = "Pynq-Z2"
fpga_part = pynq_part_map[pynq_board]
target_clk_ns = 10
```

- For this streaming system,  $FPS \approx \frac{F_{clk}}{II_{max}} = \frac{100M}{64}$ 
  - Throughput measurement: 958983
  - This is around 61.37% the ideal case

```
Network metrics:
runtime[ms]: 10.427713394165039
throughput[images/s]: 958983.0120950225
DRAM_in_bandwidth[Mb/s]: 751.8426814824976
DRAM_out_bandwidth[Mb/s]: 0.9589830120950226
fclk[mhz]: 100.0
batch_size: 10000
fold_input[ms]: 0.17762184143066406
pack_input[ms]: 0.1728534698486328
copy_input_data_to_device[ms]: 52.44183540344238
copy_output_data_from_device[ms]: 0.5877017974853516
unpack_output[ms]: 1.1982917785644531
unfold_output[ms]: 0.19979476928710938
```

# Questions

- What if we modify the parameters as follows?

```
# (PE, SIMD, in_fifo_depth, out_fifo_depth, ramstyle) for each layer
config = [
    (16, 49, 16, 64, "block"),
    (4, 4, 64, 64, "auto"),
    (8, 8, 64, 64, "auto"),
    (10, 8, 64, 10, "distributed"),
]
```

- What would you expect?