

# Lecture 2

## Case study :

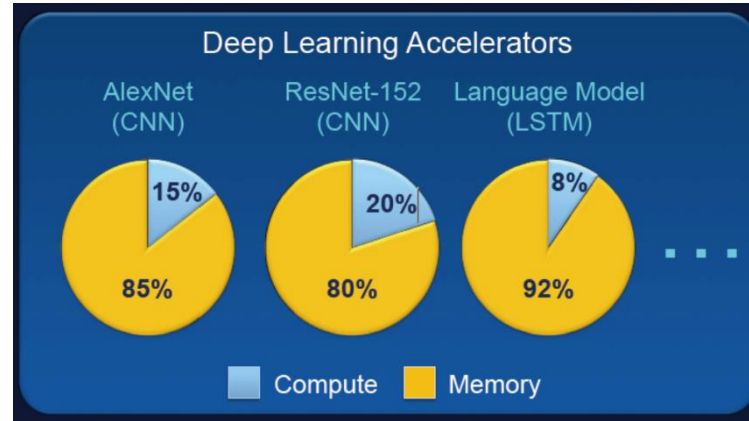
### 1D-PE Design for Convolution Layer work from T-C Wu

Chung-Ho Chen  
NCKU EE

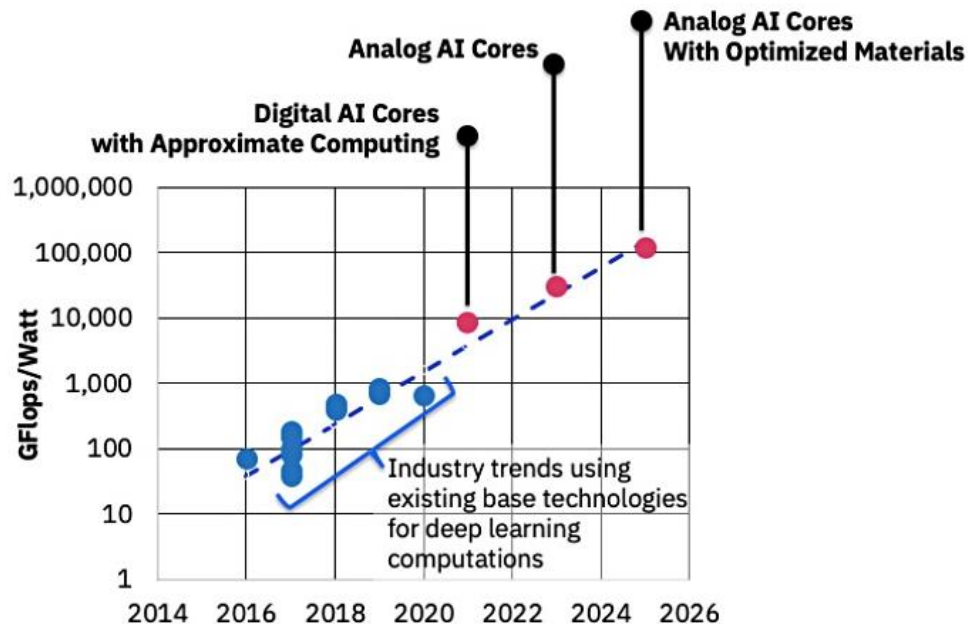
# Deep Learning Accelerators

## ► Intensive Memory Access and Computation

S. Mitra/Stanford  
Abundant-Data  
Computing  
[https://beyondcmos.ornl.gov/documents/Panel\\_Session-Mitra.pdf](https://beyondcmos.ornl.gov/documents/Panel_Session-Mitra.pdf)



Source:  
<https://blocksandfiles.com/2019/02/11/ibms-ai-chips-change-phase/>

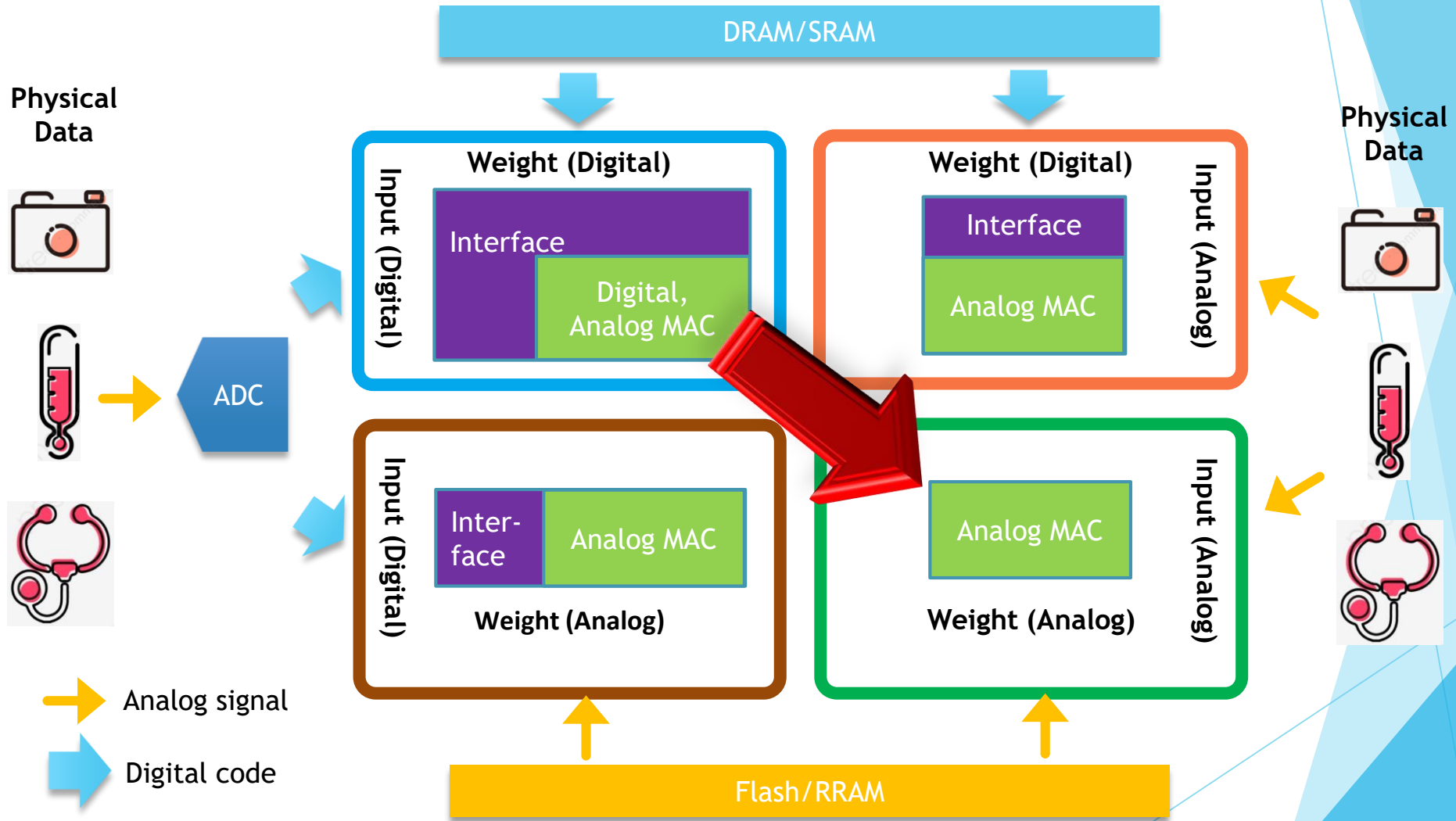


## AI accelerator development challenges

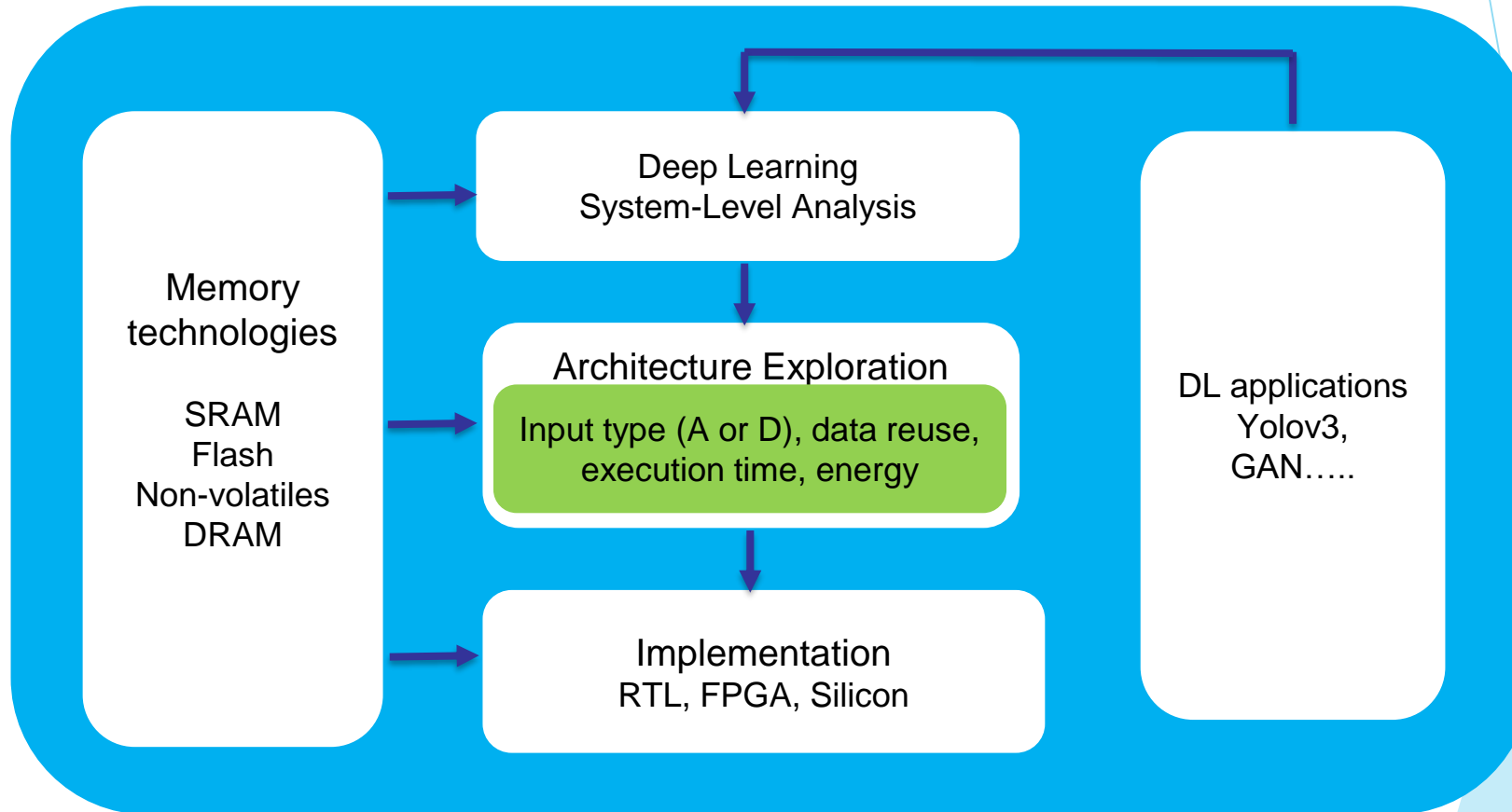
1. Intelligent data fetcher
2. Data reuse strategies in MAC (Multiply-Accumulate) PE arrays
3. Quantization scheme in MAC
4. Advanced in-memory computing analog PE
5. Design and simulation platform



# Putting into Perspective



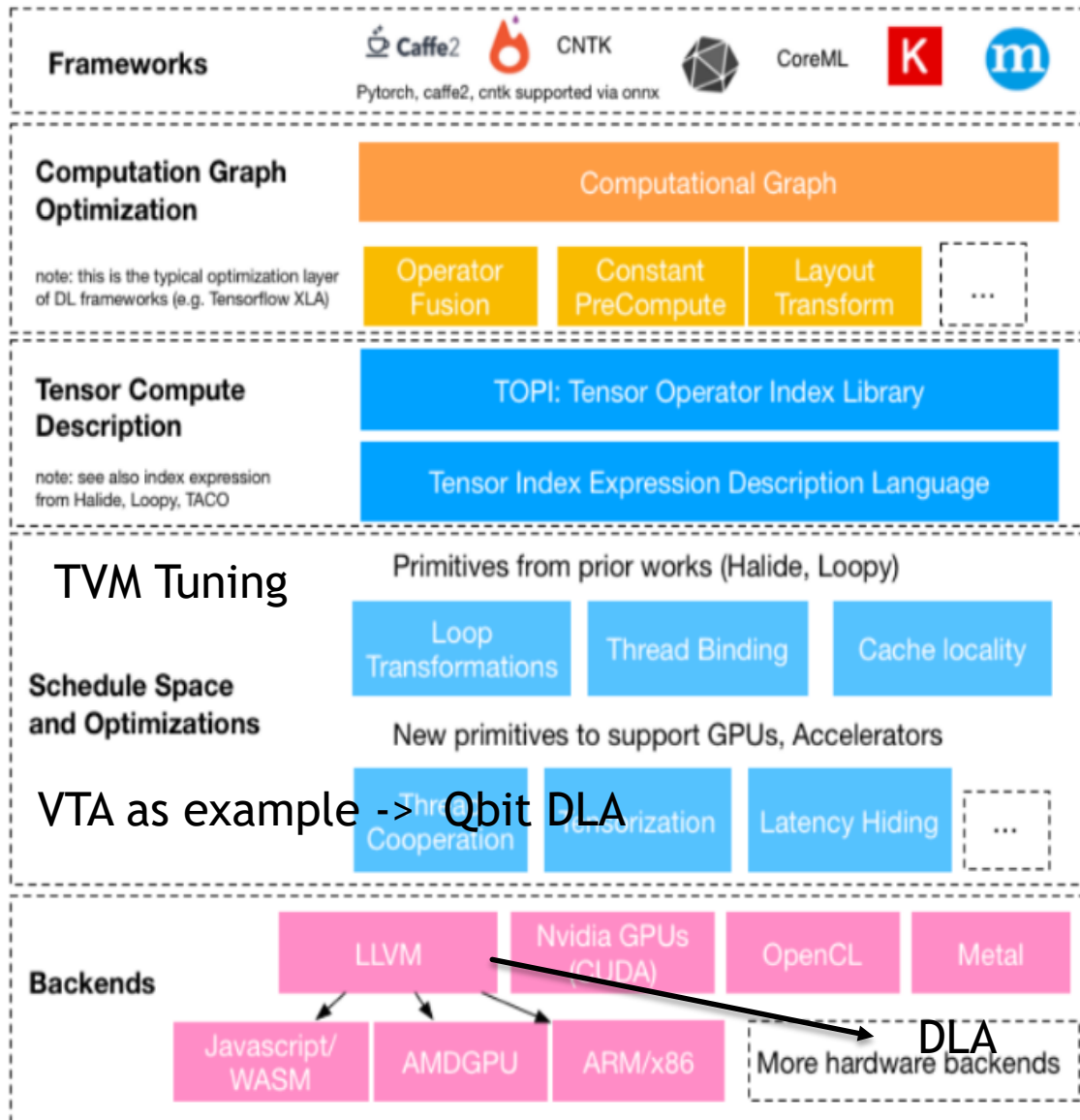
# AI Accelerator Design/Simulation Framework



# TVM: An Automated End-to-End Optimizing Compiler for Deep Learning

University of Washington- CSE

TVM Stack



Graph optimization  
operator fusion, constant precompute, layout transform

Tensor Compute Description: tensor compute library,  
tensor index expression (description language for scheduling and optimization)

LLVM intrinsic support for NNP instruction  
Define intrinsic function for convolution, for instance, in LLVM, so that it can generate the assembly code for NNP.

to go with RISC-V or ARM

# DL Graph Compiler for AI ASIC

- ▶ C Code Based Compiler
  - ▶ Given the interface spec or API of DMA and DL NNP
  - ▶ Read DL optimizer IR, or DL model config./weight, generate the sequence control C code
  - ▶ Use C compiler of the target CPU
- ▶ Assembly Code Based
  - ▶ LLVM intrinsic functions for the target CPU
    - ▶ Adding the instructions for DMA and NNP
  - ▶ LLVM compiler for the target CPU

# Neural-Net Processor (NNP) Code Generation

- ▶ Operators in AI graph are executed in either the RISC-v (or ARM) processor or the NNP
- ▶ Define instructions for NNP operators.
- ▶ Define operation granularity for NNP instructions
  - ▶ Large (Convolution)
  - ▶ Small (MAC), should be an enhanced version of RISC-V or ARM's MAC

# Revisit Convolution Operation



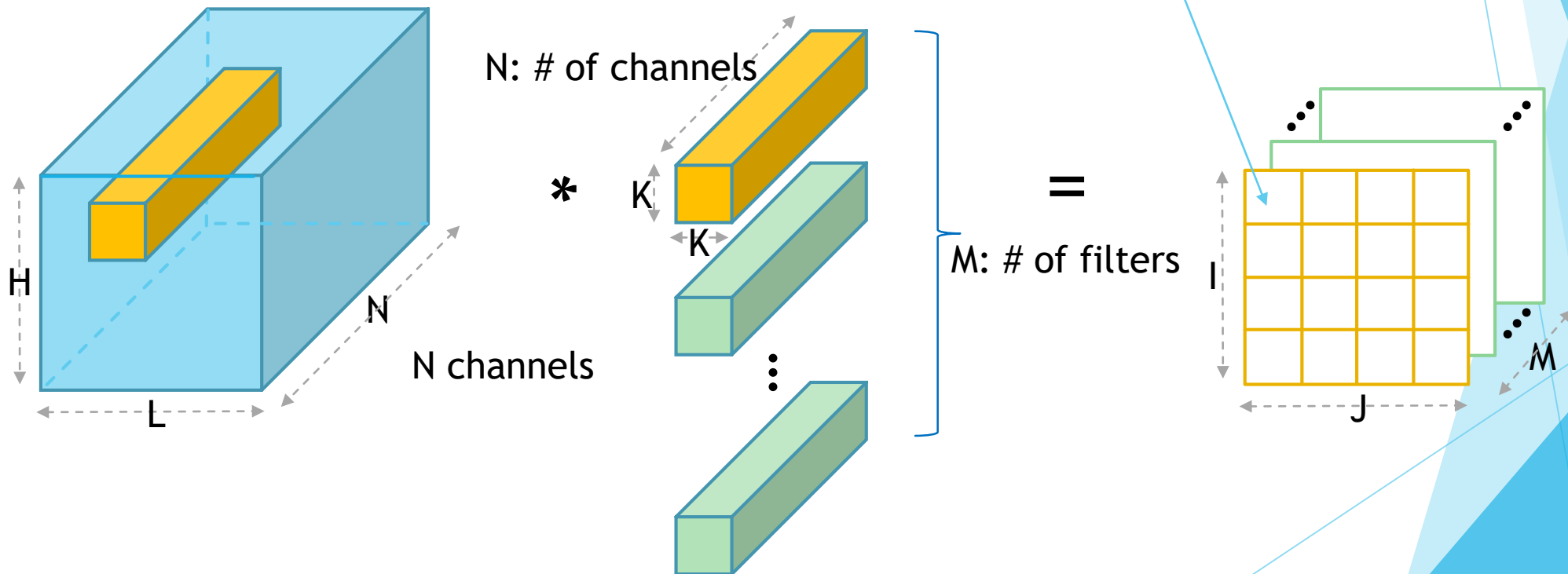
N: # of channels for input map as well as kernel

M: # of filters or kernels, K: size of a filter, which keeps weights

The number of output channels is the number of filters used.

Partial sum of channel  $i$ ,  $PS_i$ : conv of channel  $i$  (the  $i^{\text{th}}$  input map \* the  $i^{\text{th}}$  channel of a filter)

Total sum =  $\sum_{i=1}^N PS_i$ , where N is the number of channels



■ Input Size =  $H * L * N$

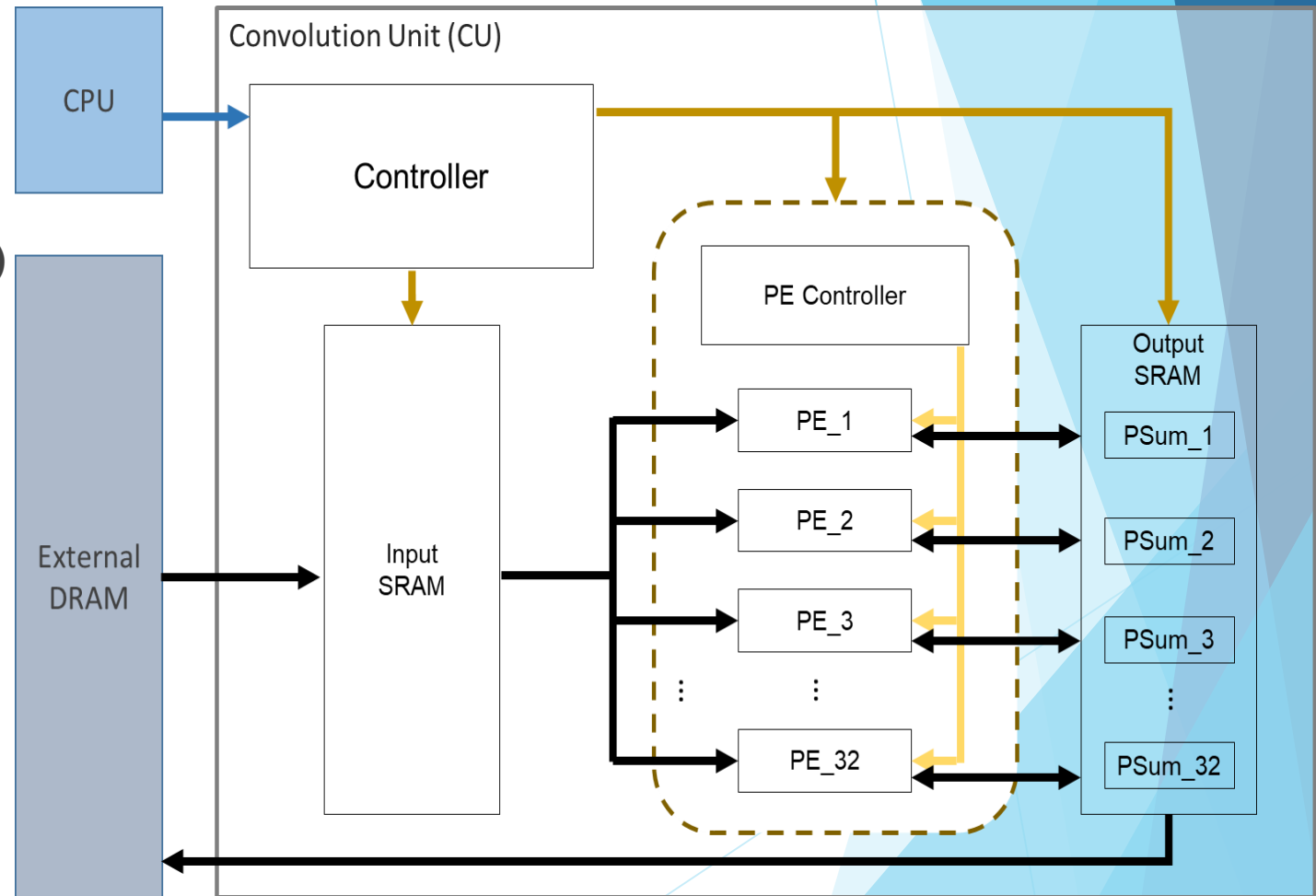
■ Kernel Size =  $K * K * N * M$

■ Output Size =  $I * J * M$



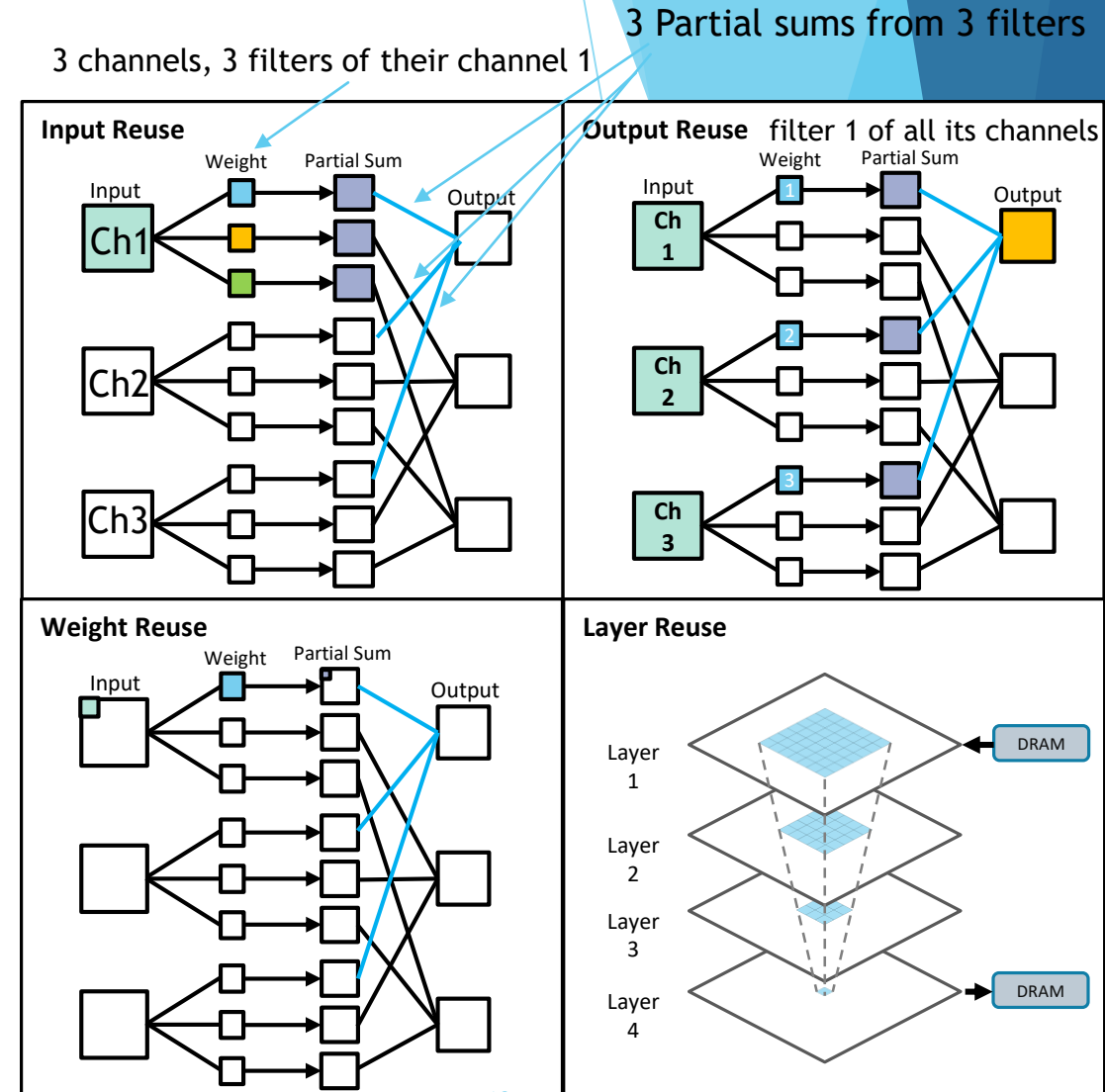
# A Convolution Unit; things to ponder

- ▶ PE structure (fixed MACs or not)
  - ▶ Determine how many MACs in a PE
- ▶ Single PE array structure (parallelism)
- ▶ Multiple PE arrays (scalable performance)
- ▶ More importantly,
  - ▶ how to reuse data
  - ▶ how to operate on various size of kernels
  - ▶ how to effectively use the PEs
- ▶ Hardwired or software (CPU) controller
  - ▶ So that runtime can call, or APIs that a code generator can use and hence produce the runtime library



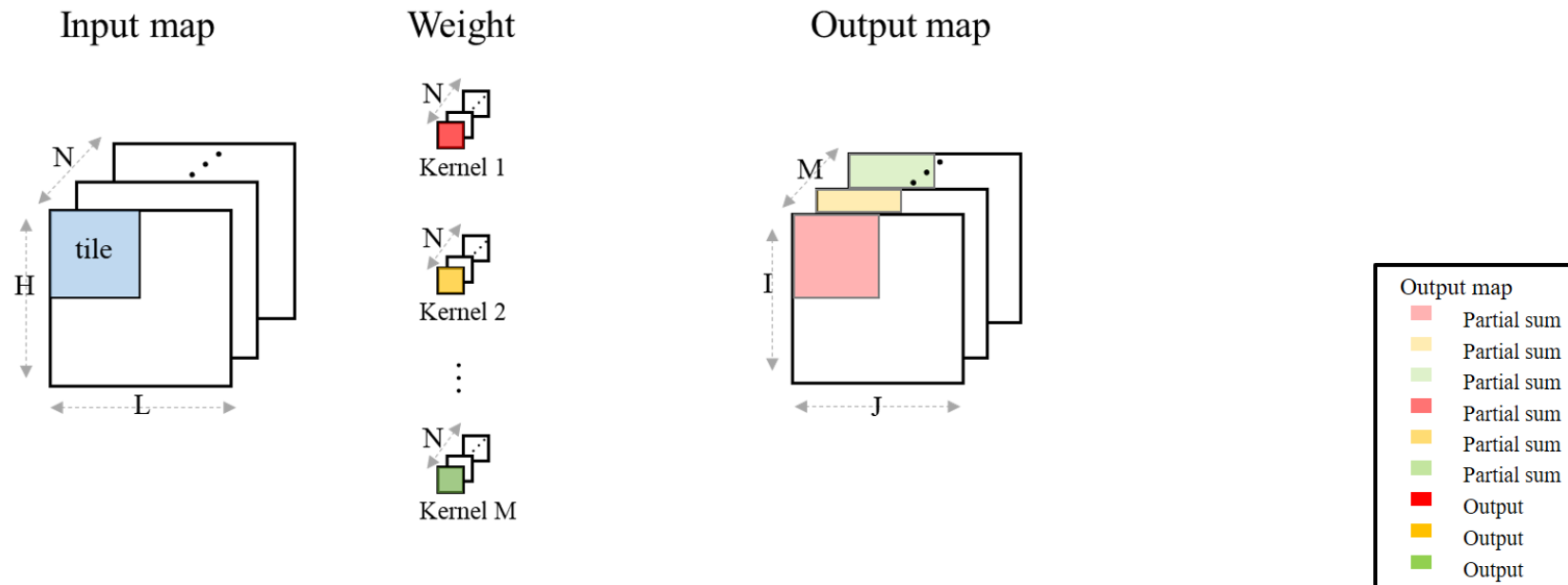
# Data Reuse

- ▶ **Input Reuse (Input stationary)**
  - ▶ Load input data only once.
  - ▶ Load all filters of the same channels at the same time.
  - ▶ Need storing all partial sums for all filters.
- ▶ **Output Reuse (Output stationary)**
  - ▶ One filter (of its all channels) at a time.
- ▶ **Weight Reuse (Weight stationary)**
  - ▶ Weight are loaded only once, and stored in the Weight SRAM.
- ▶ **Layer Reuse**
  - ▶ The input and output of the two adjacent layers are the same.
  - ▶ Only the first input and the last output need to transmission with the external memory.
  - ▶ May need a huge internal memory space.



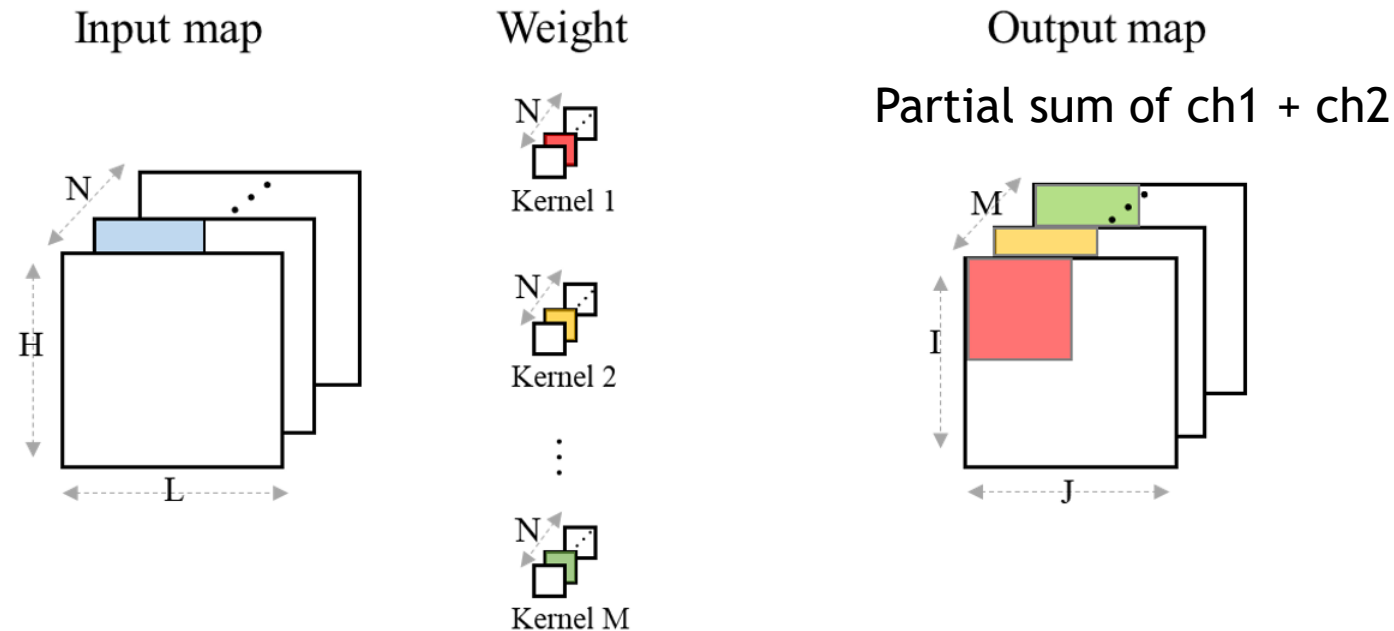
# Tile Reuse: Operation from Tile View

- ▶ Tile reuse: An input map is divided into Tiles.
- ▶ Tile Reuse: A tile is brought in from DRAM and stays stationary.
- ▶ To do this,  $M$  copies of weights of the same channel are made ready. Weights are stationary for the same input tile. (what if  $M \leq \#$  of PEs?)
- ▶ How to allocate convolution task to PEs?



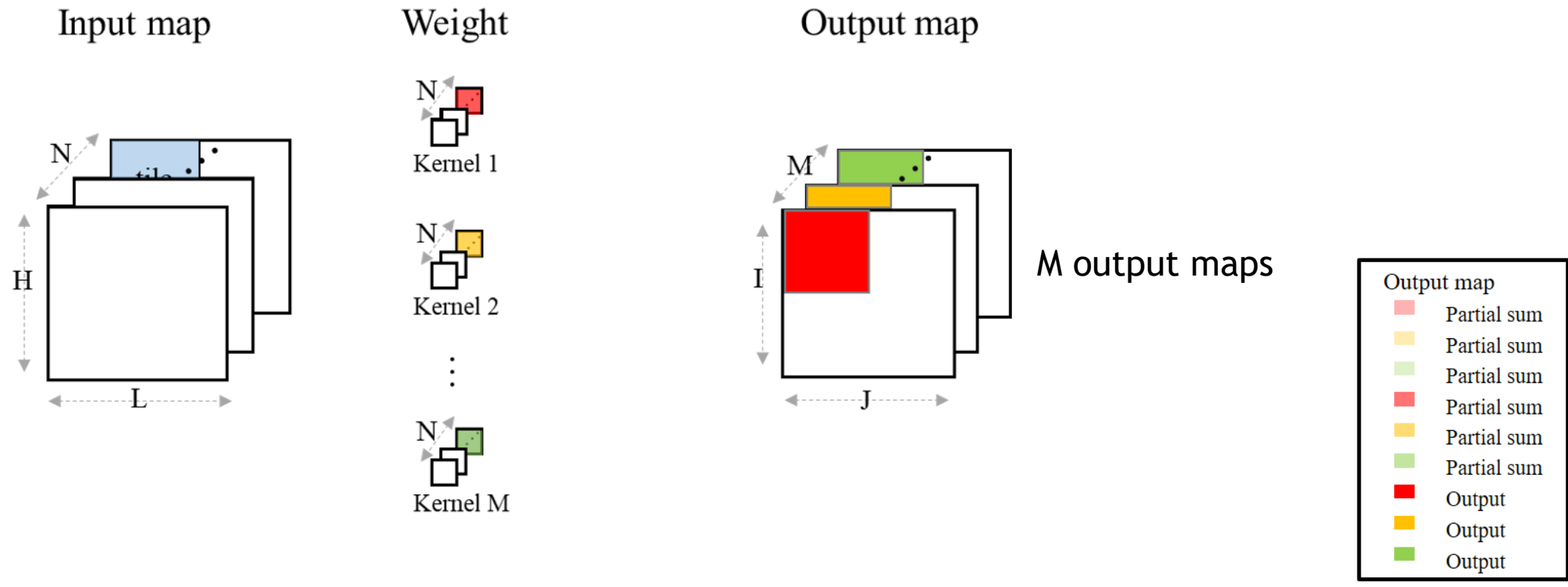
# Tile Reuse : Next tile

- ▶ Order of tile processing: Same tile position of the next channel
- ▶ Load new weights for the corresponding channel
- ▶ For this channel, **we can accumulate partial sum from the previous channel**
- ▶ M (number of filters) copies of the partial sum are produced

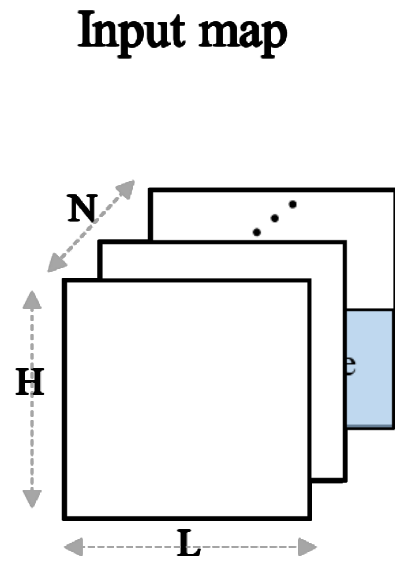


# Tile view: Last tile of the last channel

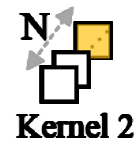
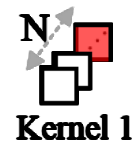
- ▶ At the end of last tile of the last channel: accumulate the total SUM for output for that tile.



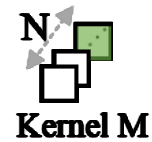
# Tile Reuse in Action



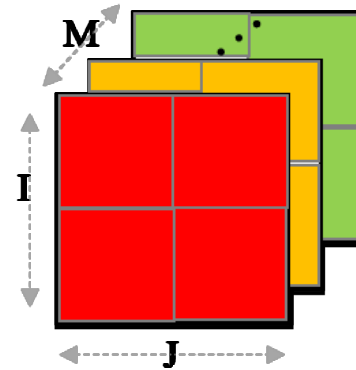
**Weight**



⋮



**Output map**

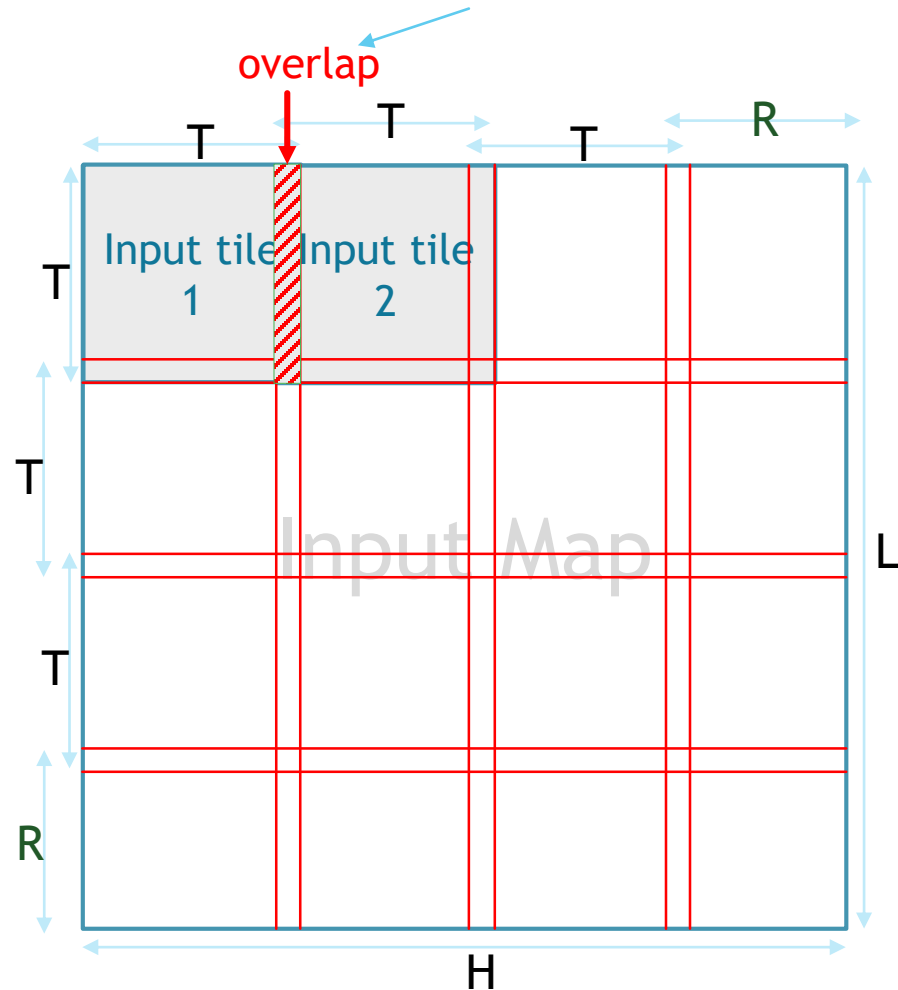


**Output map**

- Partial sum
- Partial sum
- Partial sum
- Partial sum
- Partial sum
- Partial sum
- Output
- Output
- Output

# Tile Reuse - Tile Overlap

Overhead for tile 2= 7.7 % (4/52) for T = 52, K= 3, S= 1



H : input map height

L : input map width

T : tile size

R : remainder tile height or width

K : kernel size

S : stride

$$overlap = K - S + [(T - K) \bmod S]$$

$$R = (H \bmod T) + overlap \cdot \text{floor}\left(\frac{H}{T}\right)$$

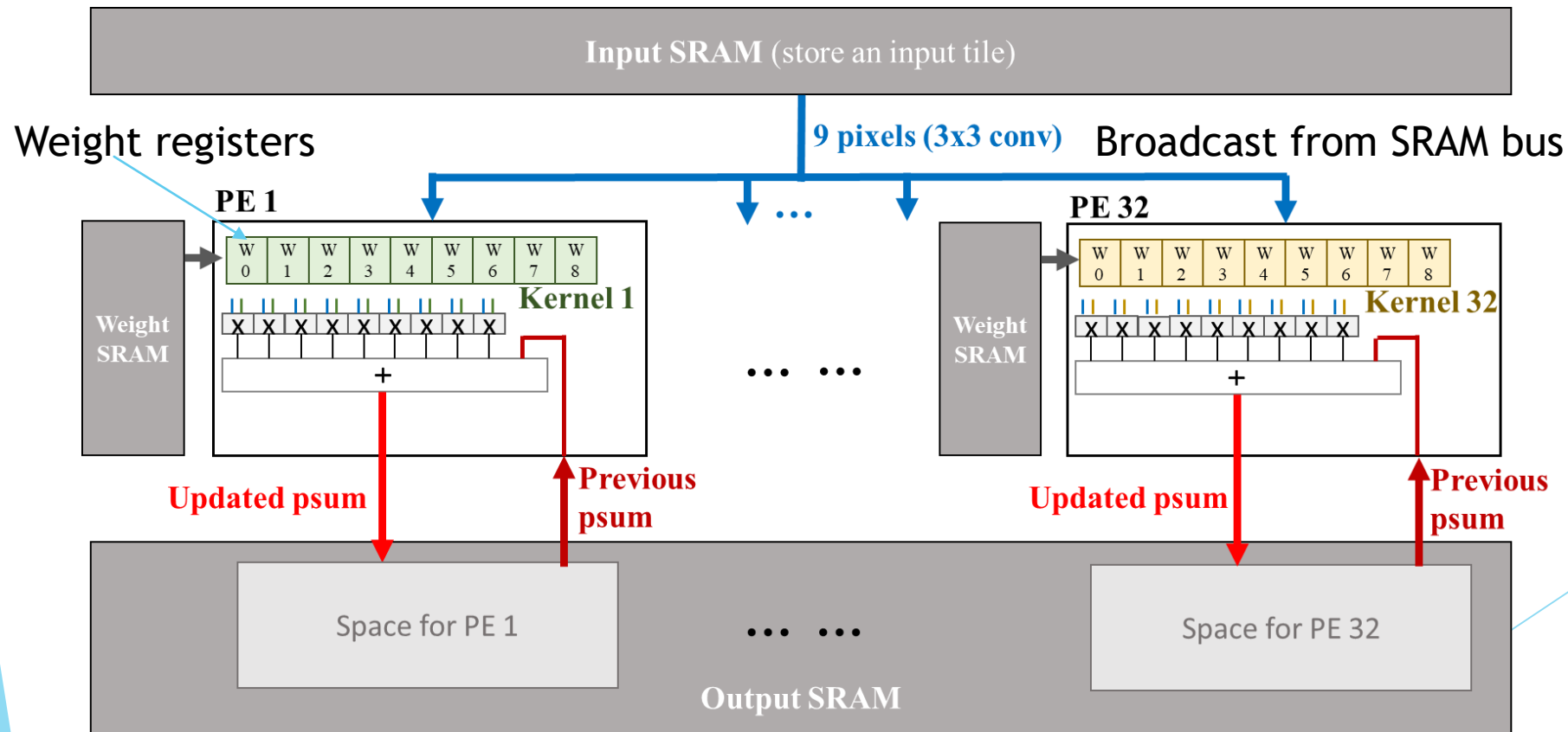
$total_{overlap}$

$$= \left[ overlap \cdot T \cdot \text{floor}\left(\frac{H}{T}\right) \right]^2 \cdot 2 + overlap \cdot R \cdot \text{floor}\left(\frac{H}{T}\right) \cdot 2$$

in term of pixels

# Convolution PE Unit

- ▶ Per-filter based convolution task allocation to a PE
- ▶ The same input is broadcast to all PEs for convolution operations
- ▶ Parallel convolution operations on multiple filters for the same input data

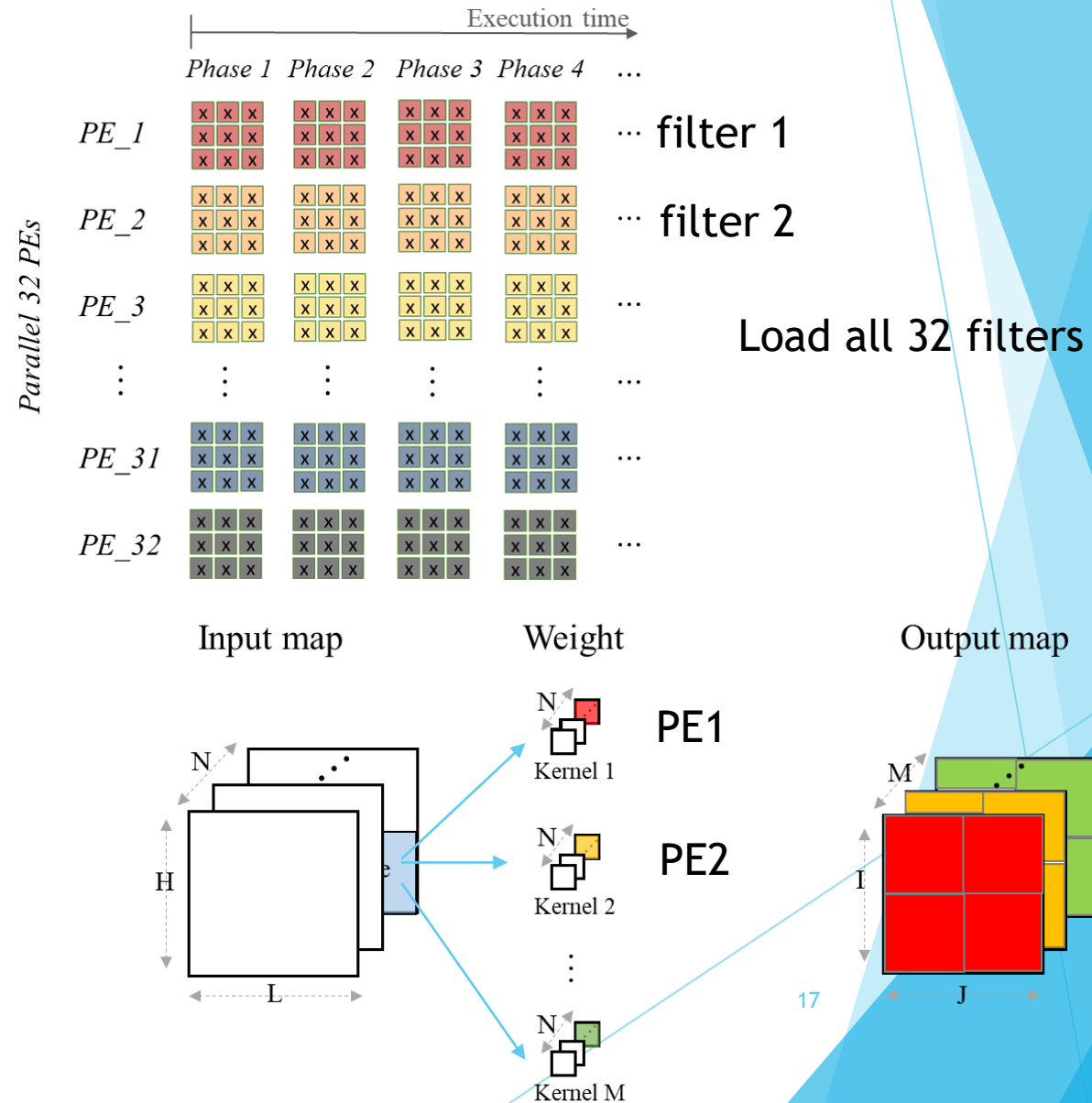




# Per-filter based conv. task allocation

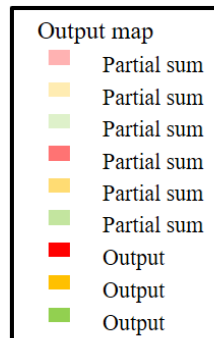
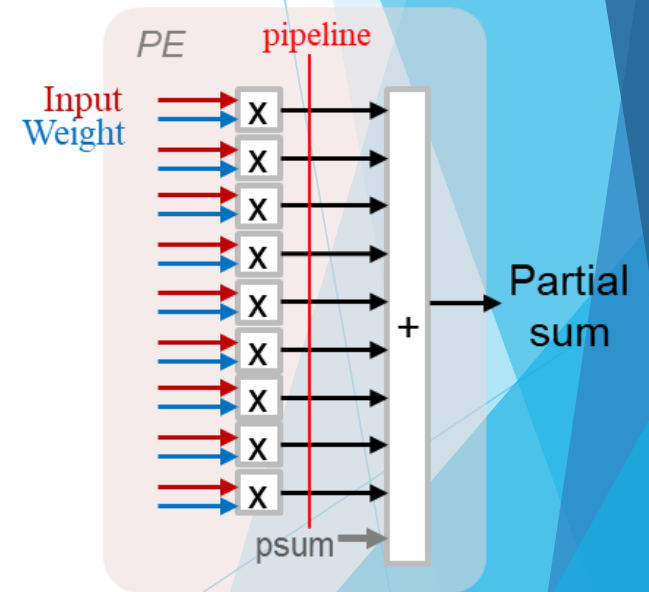
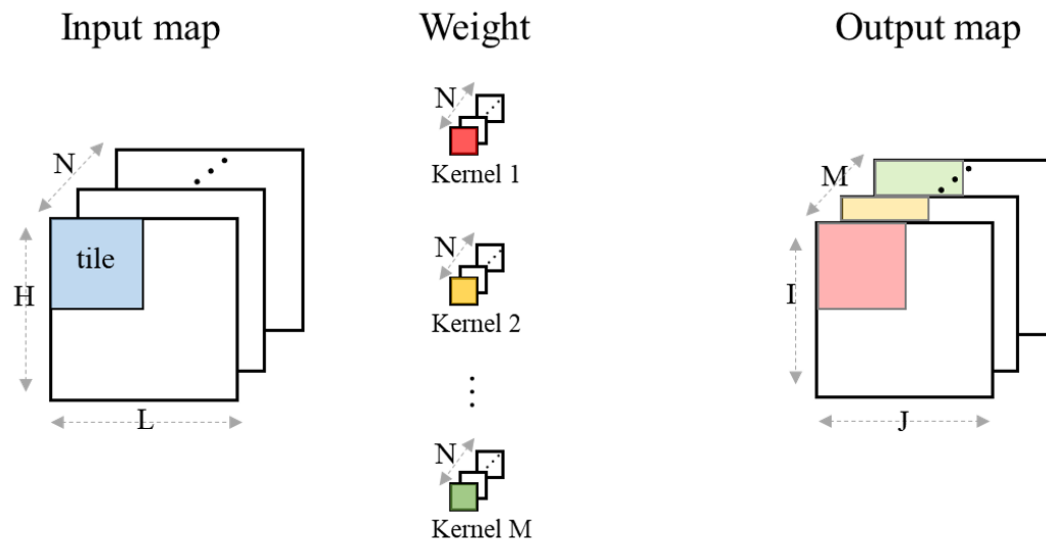


- ▶ Assume # of filters  $M=32$ , # of PEs = 32
  - ▶ If  $M \geq 32$ , then batch
  - ▶ If  $M < 32$ , 32-M PEs idle
- ▶ Allocate one filter to a PE and compute **channel by channel** for tile-sized input
- ▶ Synchronization of all PEs to the same input.
  - ▶ Each PE receives the same 9 input pixels
- ▶ Tile order: channel by channel tile order (why? easy to accumulate PS)

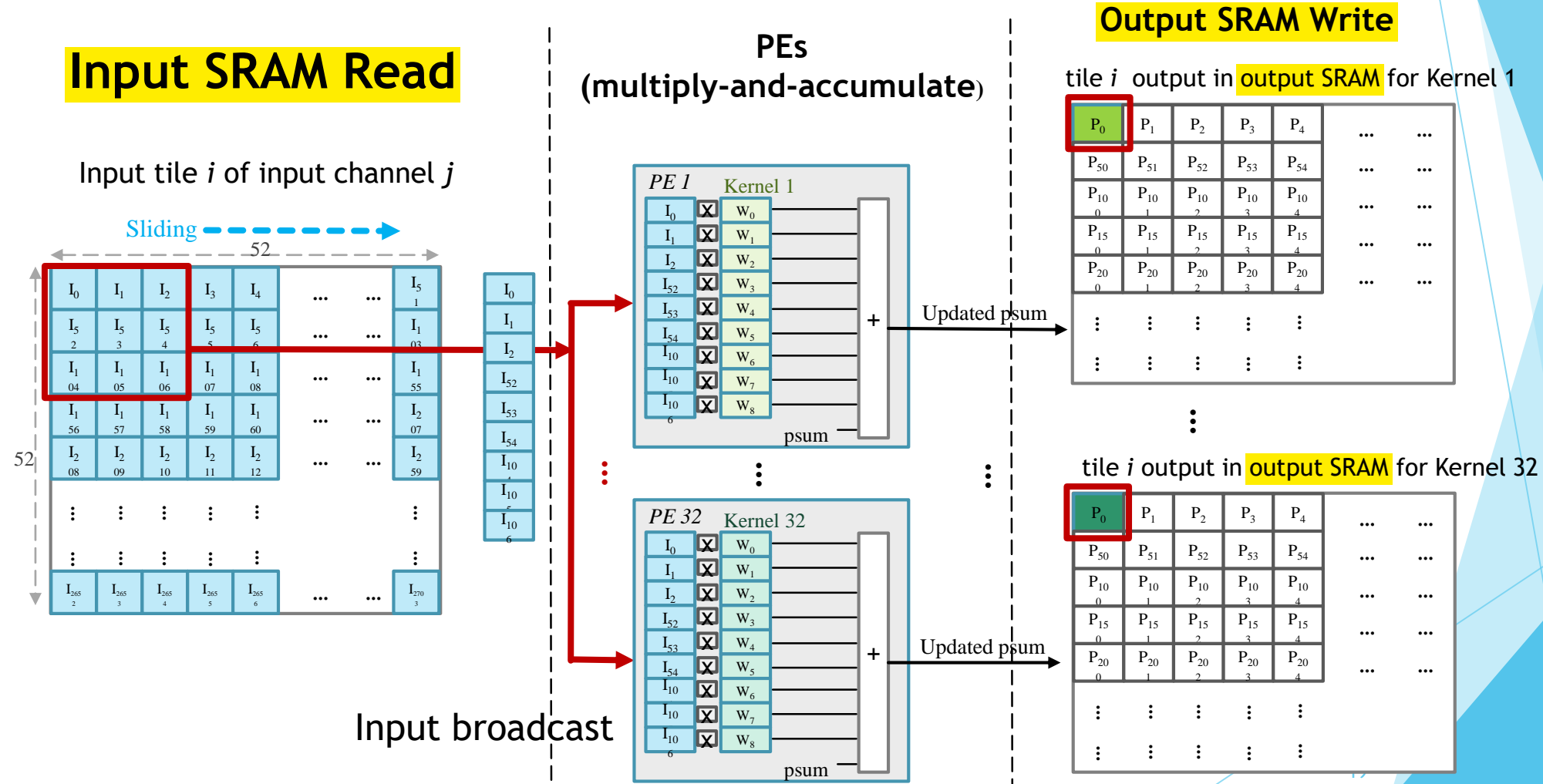


# Partial sum accumulated PE design

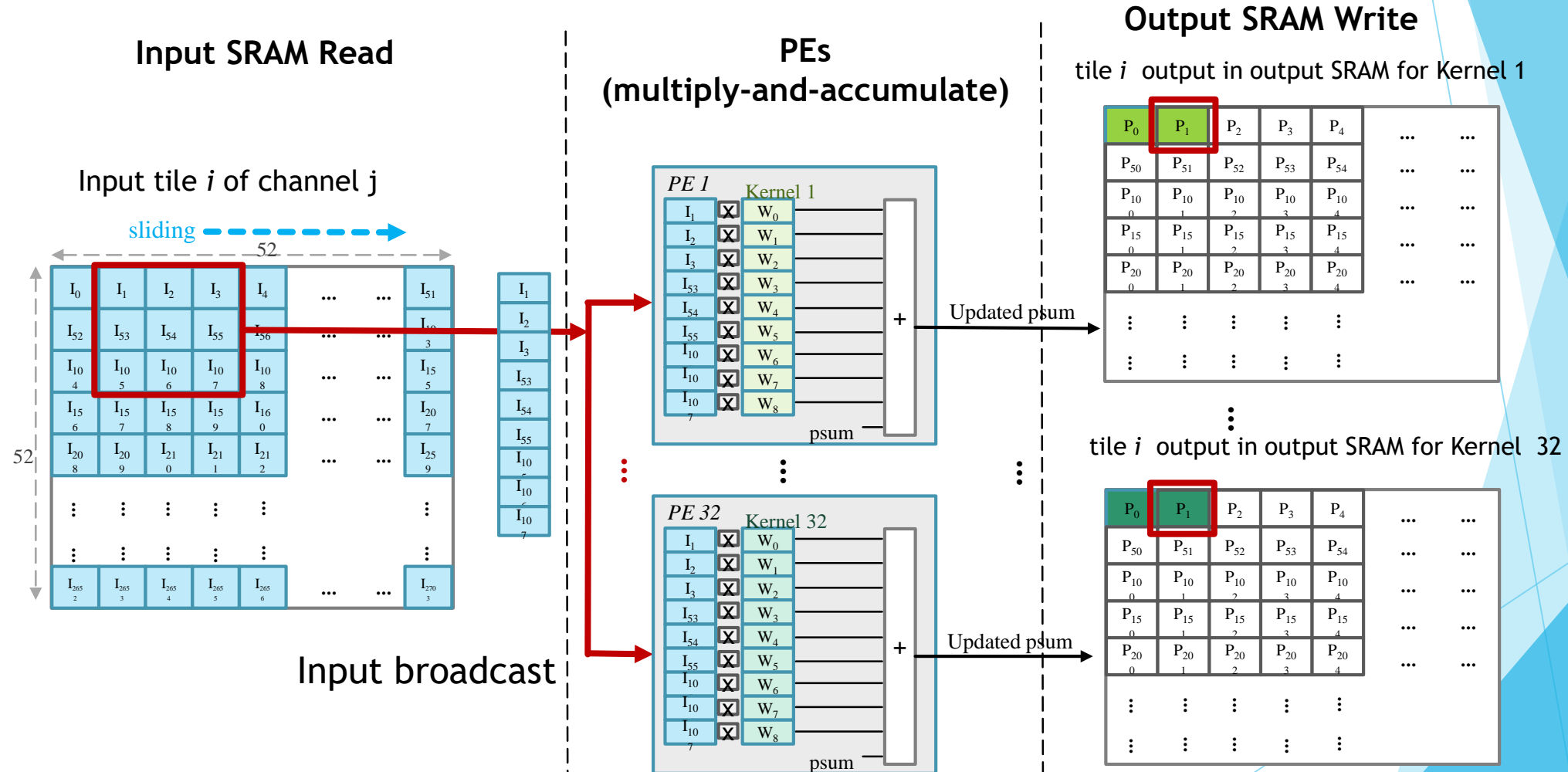
- ▶ Partial sum from the previous channel tile is added with the current channel convolution result.
- ▶ For a PE, it needs only one partial sum output memory which is reused until the total sum is accumulated.



# Tile view: Operation within a Tile (1/2)



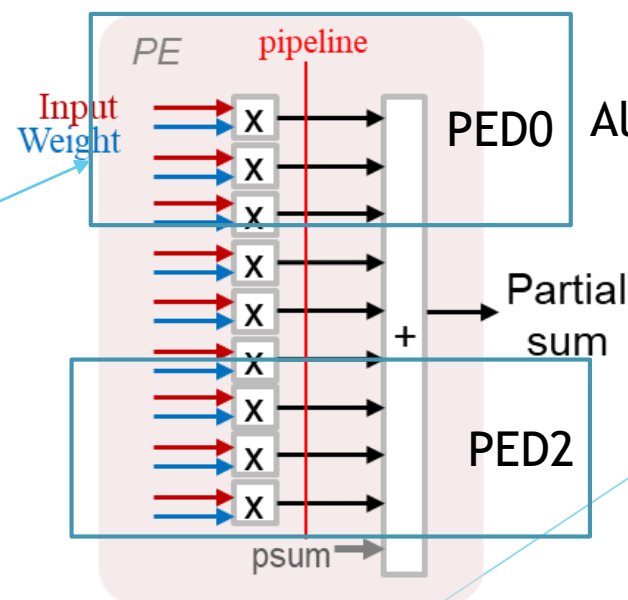
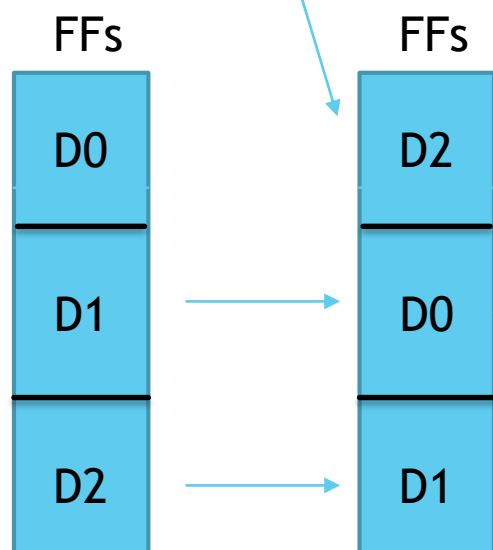
# Tile view: Operation within a Tile (2/2)



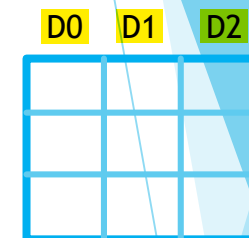


# Optimization of Data Streaming From SRAM

- ▶ A scheme to reduce 2/3 fetches of data from SRAM
- ▶ New column data D2 are loaded into the FFs that release D0 for  $S=1$



Convolution block: Column data



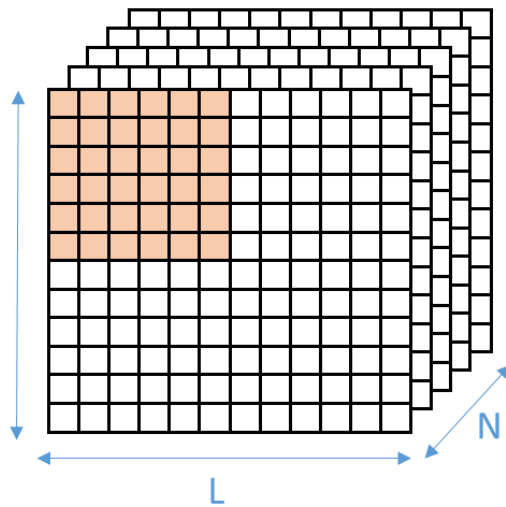
Allocated to process D0

Weights remain at their position

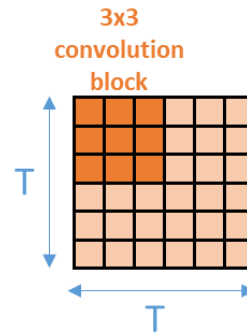
# Global view: Per-filter Based Computation

## In a tile 1/5

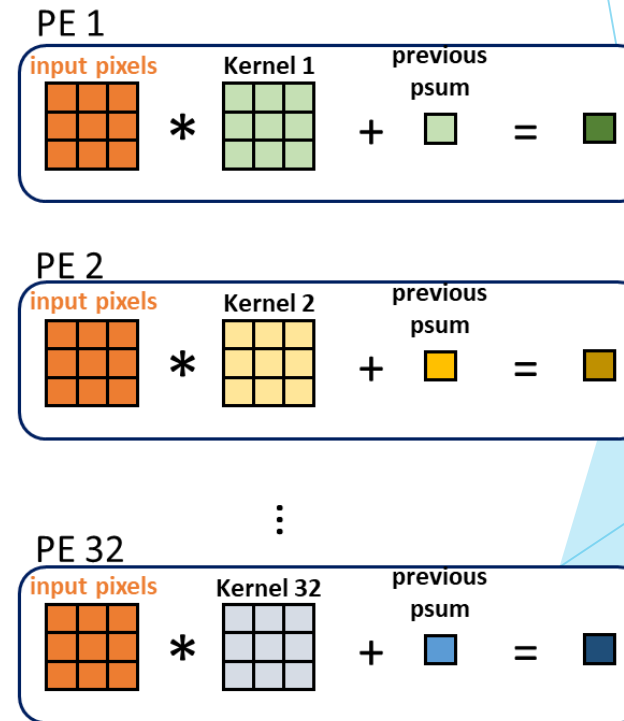
- ▶ 3x3 kernel Input maps in DRAM



An input tile in Input SRAM



Convolution block in 32 PEs



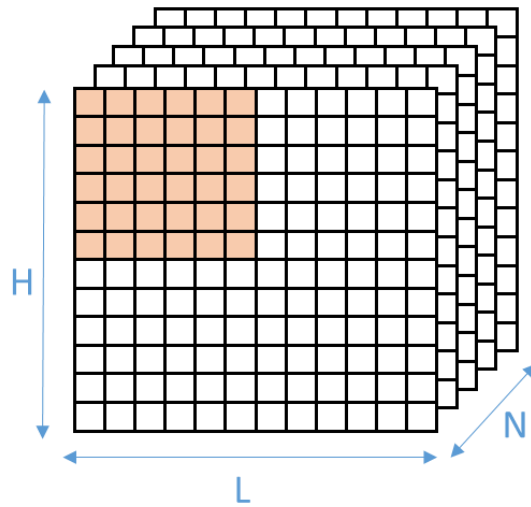
Partial sum in Output SRAM



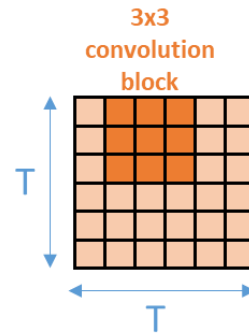
# Global view: Per-filter Based Computation in the same tile 2/5

## ► 3x3 kernel

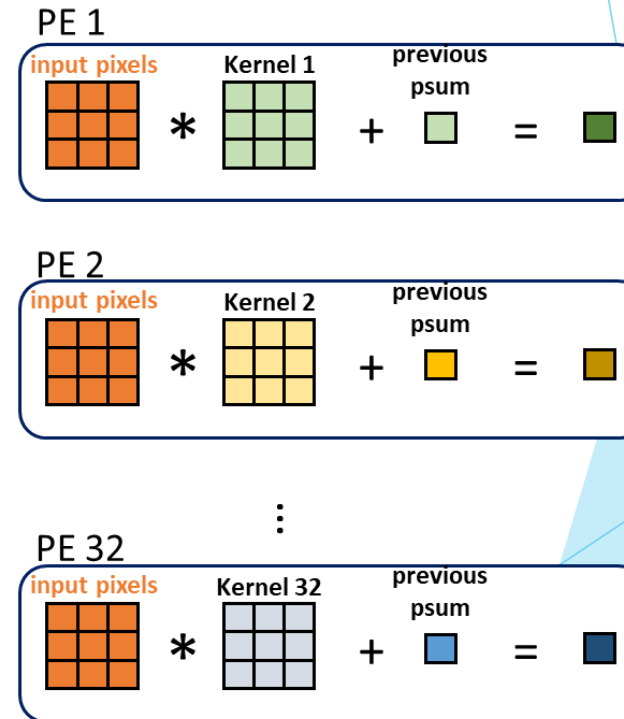
Input maps in DRAM



An input tile in Input SRAM



Convolution block in 32 PEs



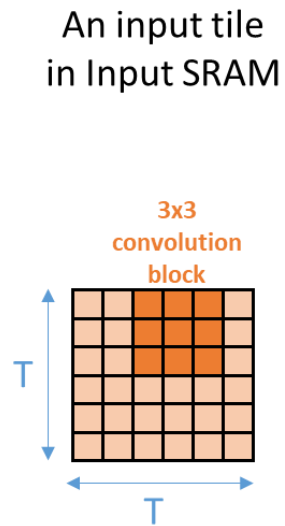
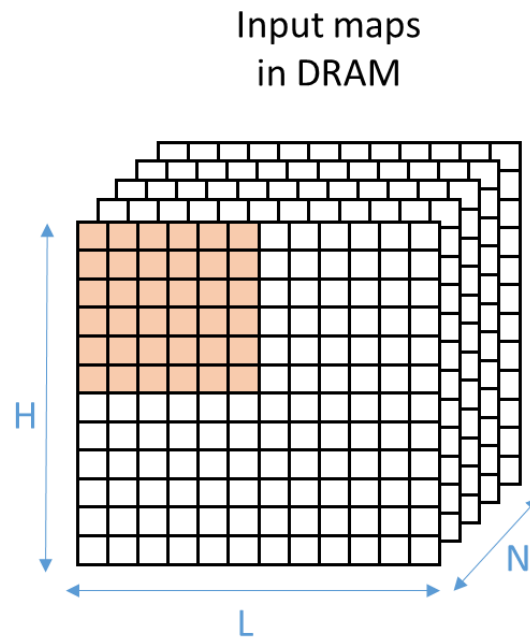
Partial sum in Output SRAM



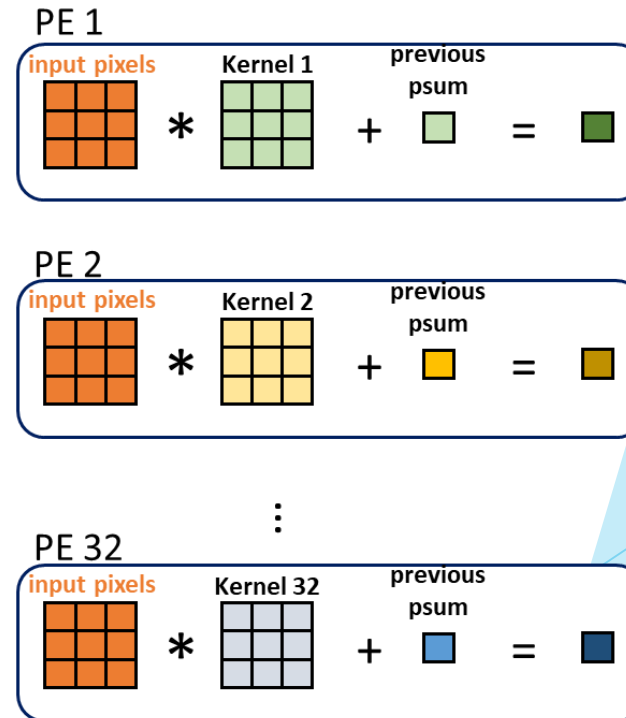


# Global view: Per-filter Based Computation in the same tile 3/5

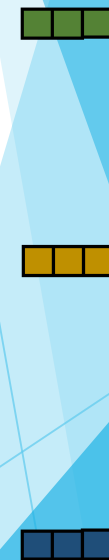
## ► 3x3 kernel



## Convolution block in 32 PEs



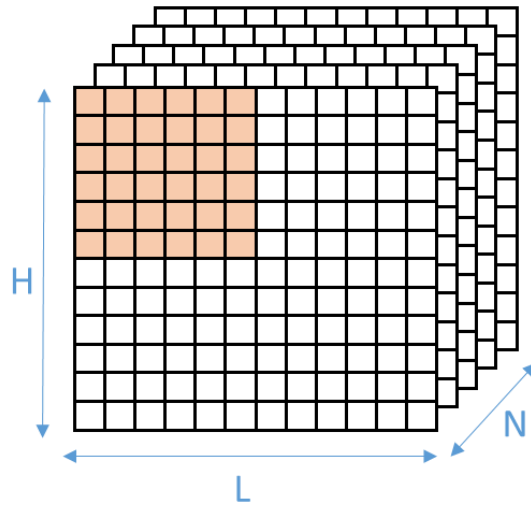
## Partial sum in Output SRAM



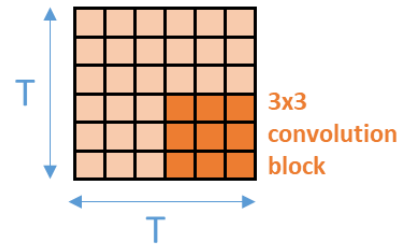
# Global view: Per-filter Based Computation in the same tile 4/5

## ► 3x3 kernel

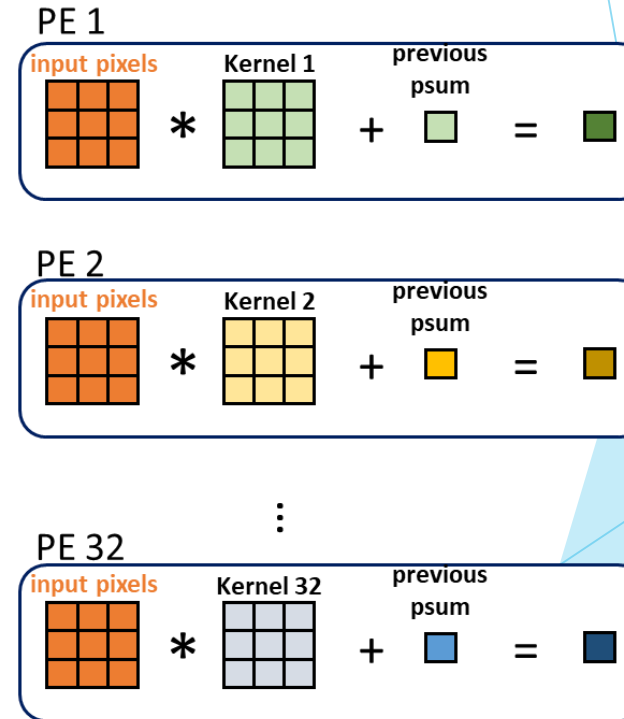
Input maps in DRAM



An input tile in Input SRAM



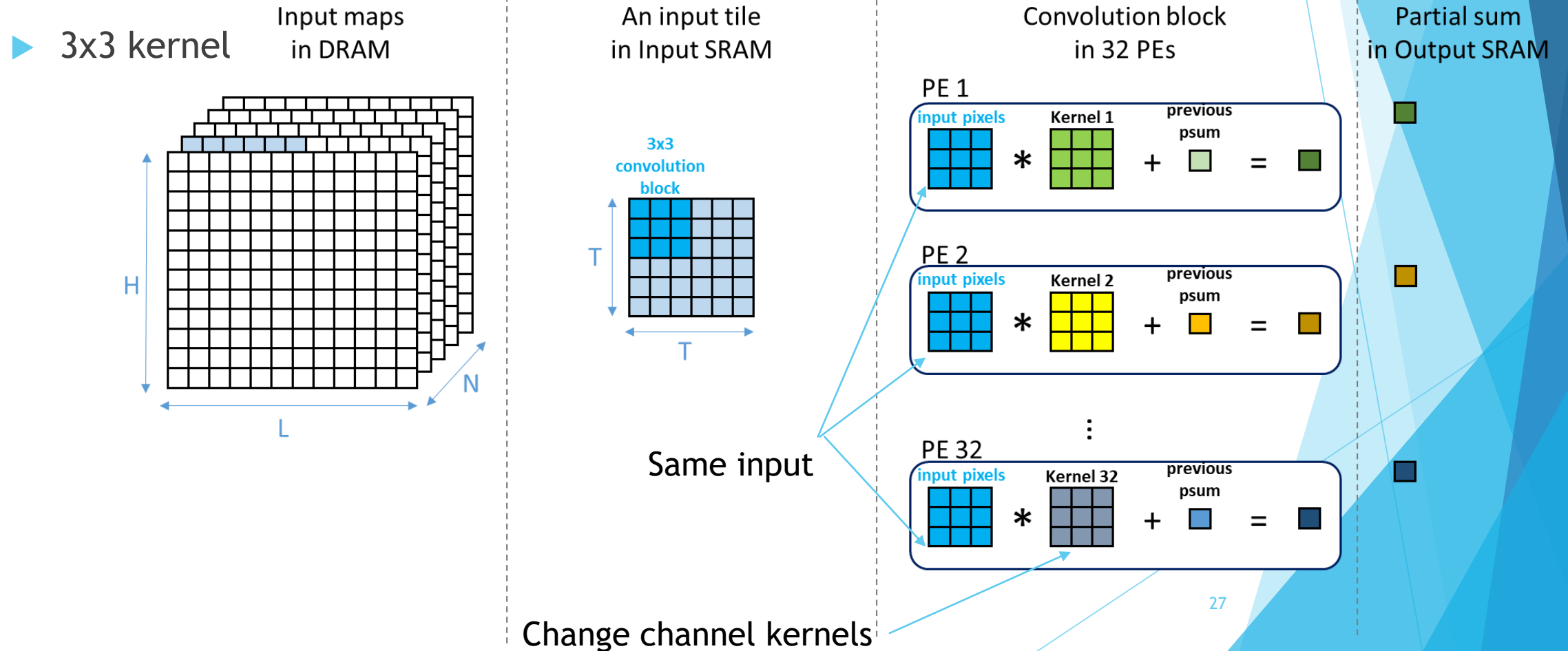
Convolution block in 32 PEs



Partial sum in Output SRAM




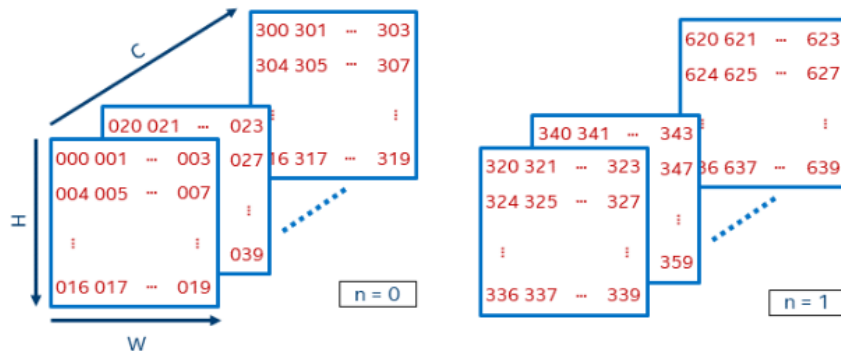
# Global view: Per-filter Based Computation change channel 5/5



# Project - part A:

## Intelligent Data Fetcher (1/3)

- ▶ Problem statement: Propose a SRAM memory system and intelligent data fetcher design  that can effectively prepare and align data to the PEs.
- ▶ At the beginning we will provide you the DRAM model of Verilog and SystemC code. You need to put the **input data and weights** we provided into DRAM.
- ▶ Different framework's data formats will be placed in memory in different ways. The two most common data formats are NCHW and NHWC, so you need to choose one of the following data formats and your data should be placed like the form below.



**NCHW:** 000|001|002|003|004|...|018|019|020|...|318|319|320|...|...

**NHWC:** 000|020|...|300|001|021|...|283|303|004|...|319|320|340|...|...

# Project - part A:

## Intelligent Data Fetcher (2/3)

- ▶ Propose a SRAM memory system. You can analyze which data reuse method is better and evaluate the SRAM size before designing, because this may affect your performance.(ex. Ping-pong Input SRAM, Weight SRAM)
- ▶ Design an intelligent data fetcher(DMA) which can fetch data from DRAM then stores it into the SRAM in the proper order needed for the PE unit.
- ▶ Since project part-B and part-C require you to design a PE accelerator, you need to think about a complete process to allow PE to receive data efficiently.

# Project - part A:

## Intelligent Data Fetcher (3/3)

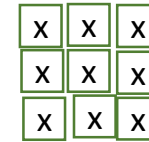
- ▶ Input image size :  $416(\text{height}) * 416(\text{width}) * 3(\text{channel})$
- ▶ Filter :  $16 * 3(\text{height}) * 3(\text{width}) * 3(\text{channel})$
- ▶ Padding = 0 , Stride = 1
- ▶ Size of input SRAM and weight SRAM :

You can decide the SRAM size and how to design it, but you need to explain why you want to do this and write it into the report.
- ▶ DMA :

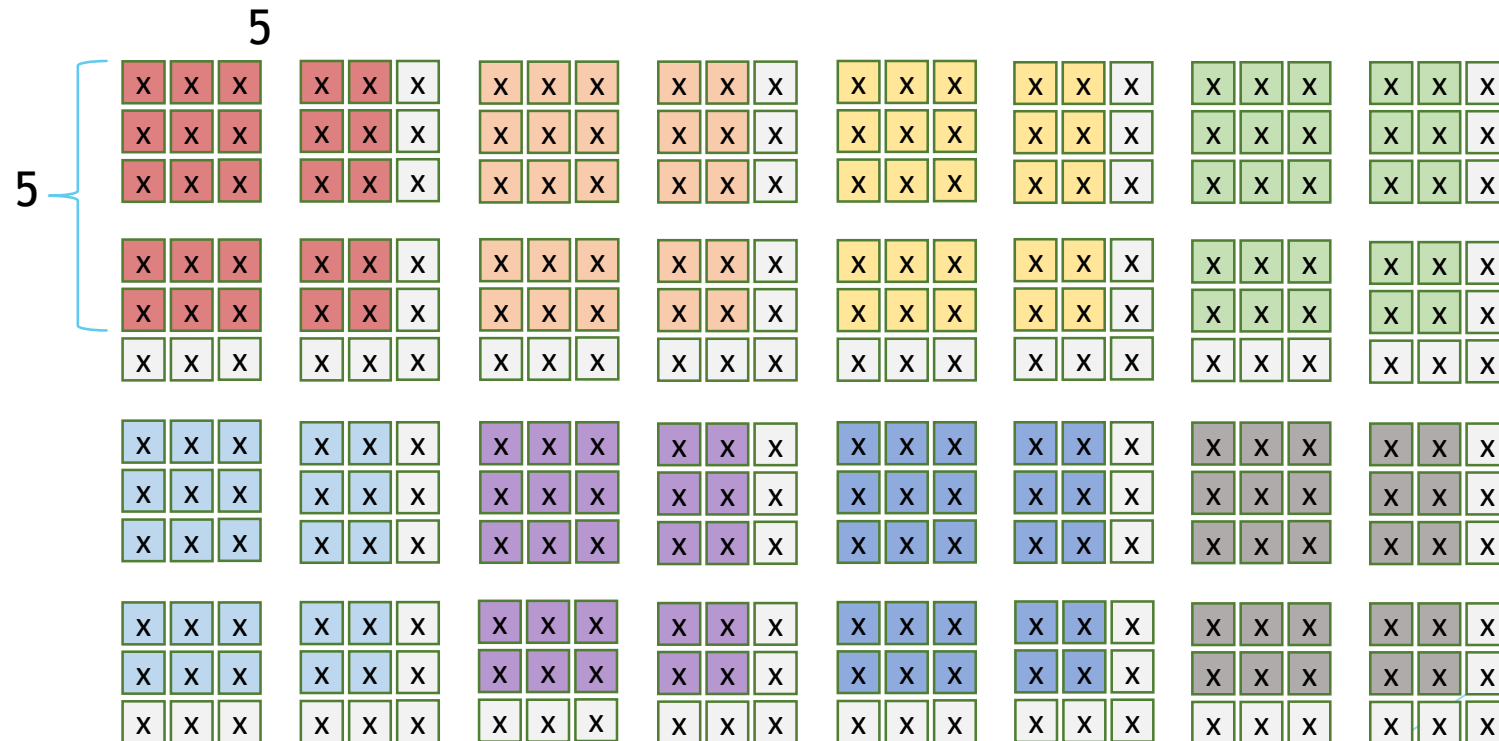
You can use a start signal to tell DMA to start moving data. When all data is moved, you need an interrupt signal to inform the CPU that the task is complete.
- ▶ Can be designed with RTL or SystemC code
- ▶ Report due in 4 weeks.

# Kernel View: Oversize Kernel Handling (1/2)

- ▶ Geometric directly mapping on 3x3 PEs
- ▶ Example: 5x5 kernels (filters are colored)
- ▶ Low PE utilization =  $25/36 = 69.4\%$

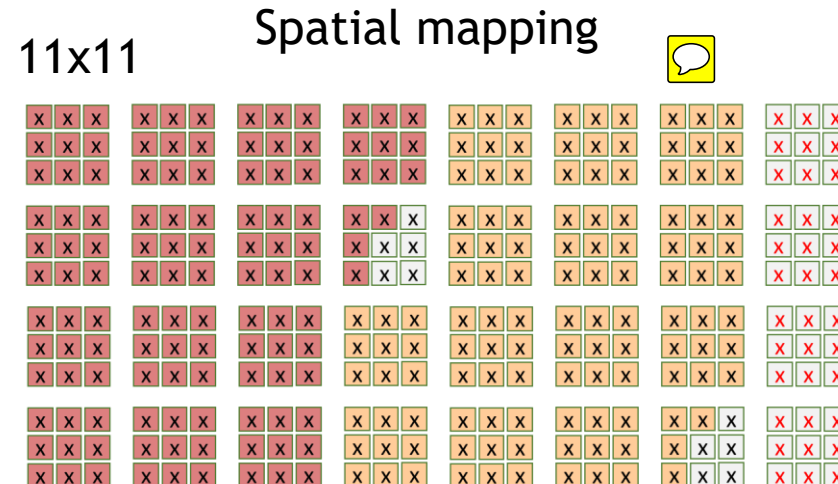
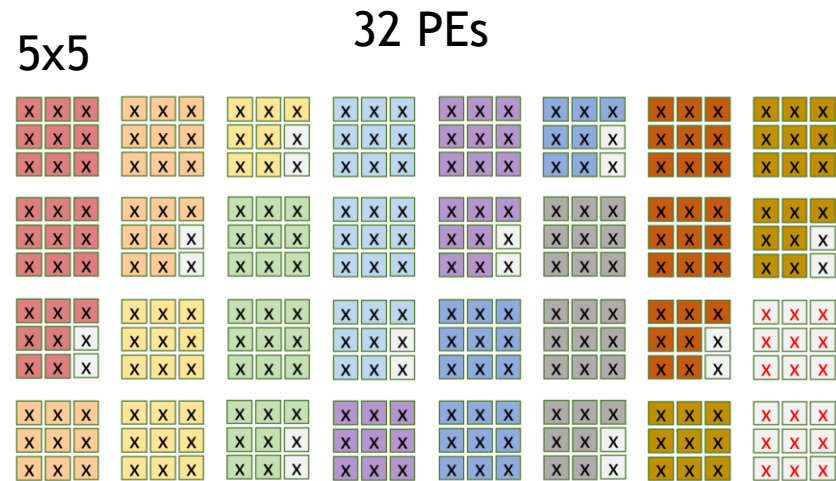


A 3x3 PE core



# Kernel View: Row-major Order Kernel in Geometric Mapping, on different PEs (2/2)

32 PEs, each 9 MACs

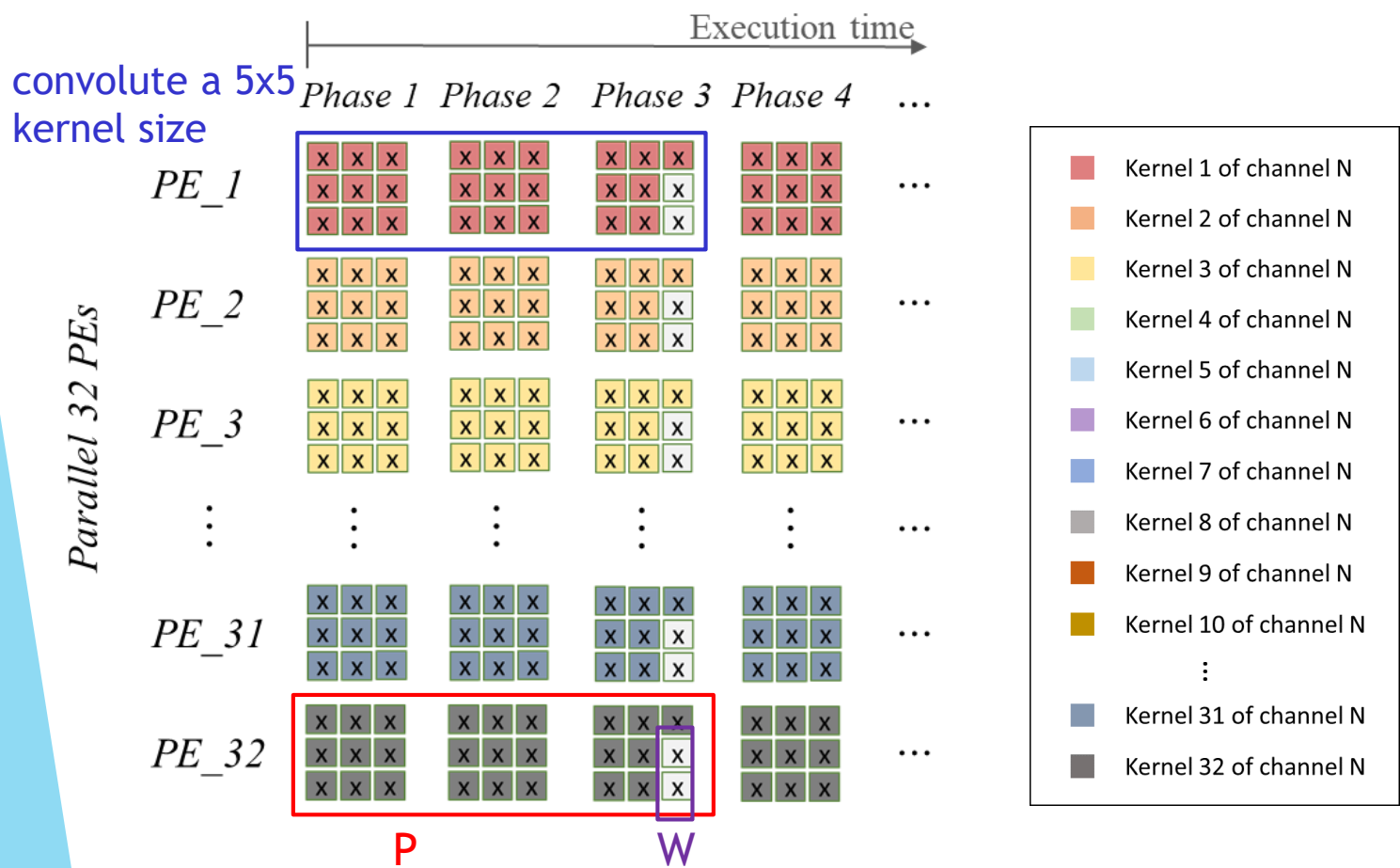


Different color means different kernels  
10 kernels at a time  
Run from channel 1 to N for the given kernel. Then go for another 10 kernels if any

- Kernel 1 of channel N
- Kernel 2 of channel N
- Kernel 3 of channel N
- Kernel 4 of channel N
- Kernel 5 of channel N
- Kernel 6 of channel N
- Kernel 7 of channel N
- Kernel 8 of channel N
- Kernel 9 of channel N
- Kernel 10 of channel N
- ⋮
- Kernel 31 of channel N
- Kernel 32 of channel N



# Kernel View: Row-major Order Kernel Mapping on the Same PE for Oversize Kernel (1 / 2)



P: needed phases for oversize kernel  
W: waste multiplier number for oversize kernel  
k: kernel size

$$P = \text{ceil}(\frac{k \cdot k}{9})$$


$$W = 9 - \left[ (k \cdot k - \text{floor}(\frac{k \cdot k}{9}) \cdot 9) \right]$$

Kernel size	P	W
5	3	2
7	6	5
11	14	5

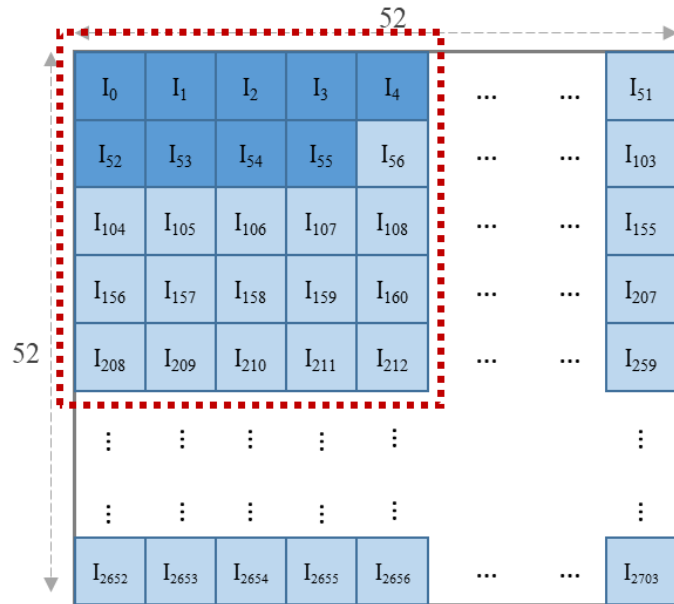
Same input broadcast to all PEs. Suggesting that weights are loaded to PEs of the same row-major order.

# Kernel Row-major Order Mapping on the same 3x3 PE for Oversize Kernel (kernel + input view)

Get 9 tile data in row-major ordering of the kernel and send to all 32 PEs.

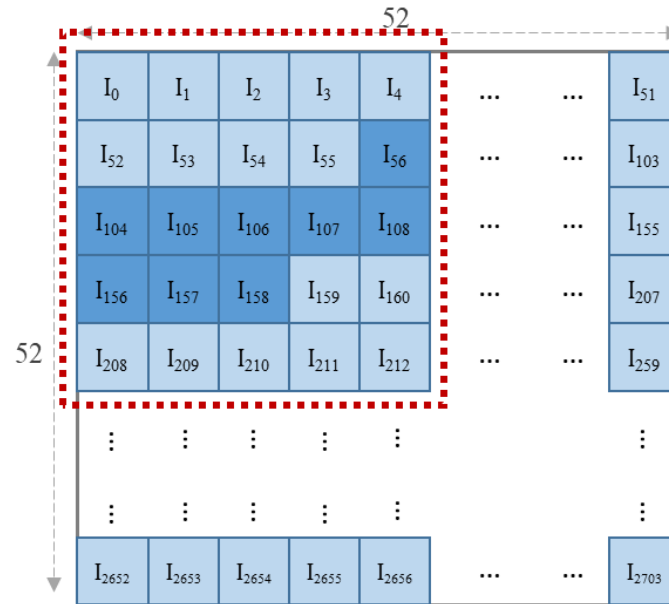
PE utilization = 25/27 

5x5 kernel example: 1<sup>st</sup> batch

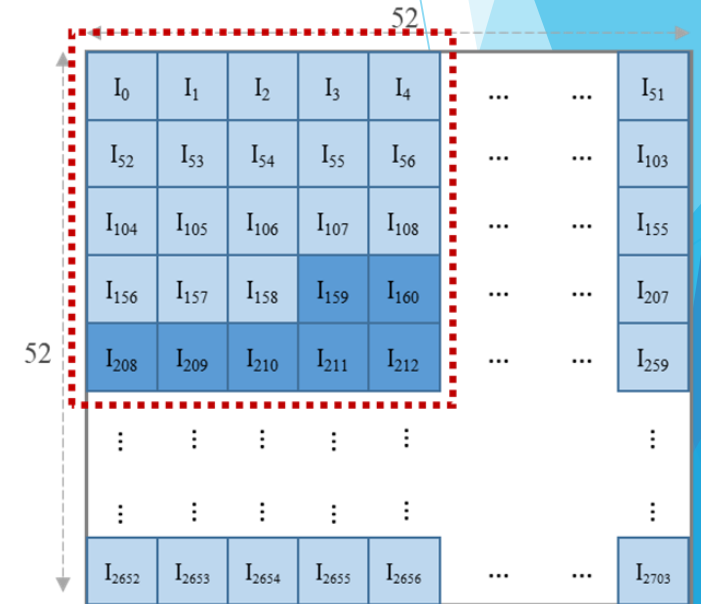


2<sup>nd</sup> batch

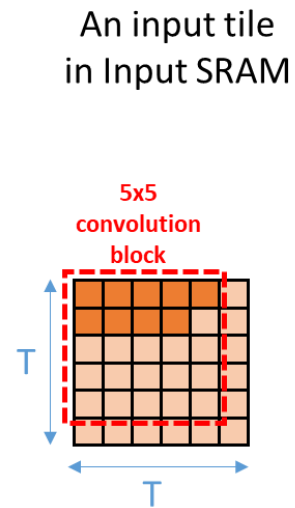
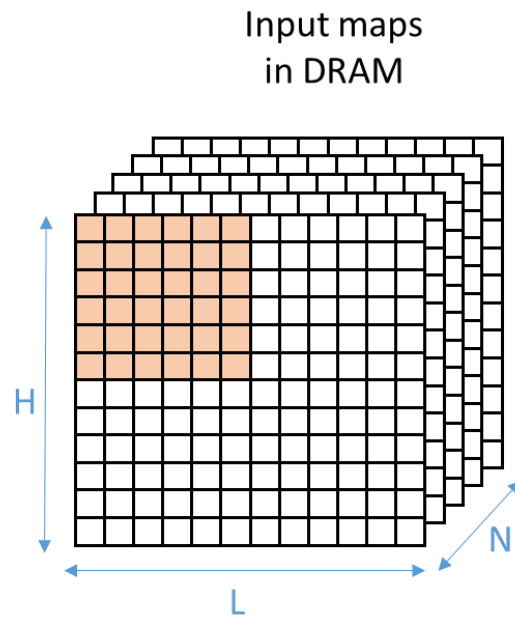
get the part of weights for 2<sup>nd</sup> batch



3<sup>rd</sup> batch

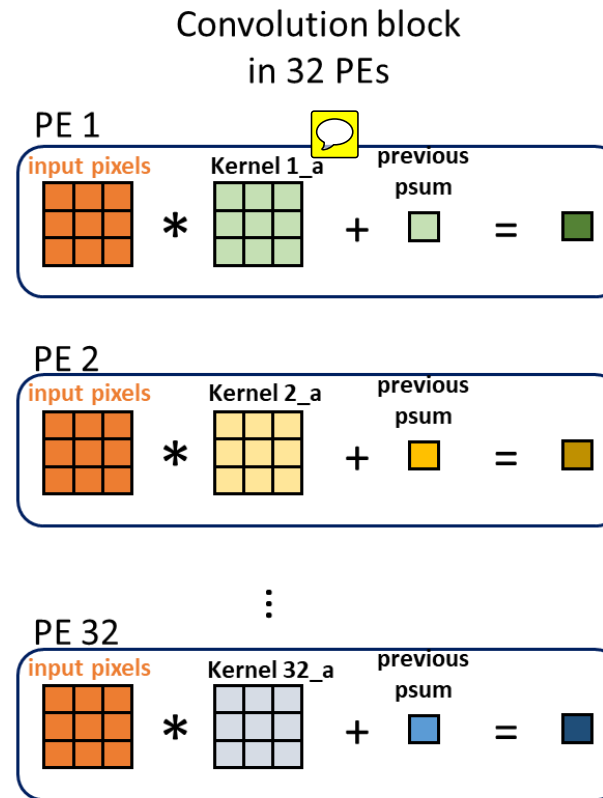


# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel in Action (1/6)



Input broadcast to all PEs

Get the first 9 weights for each PE

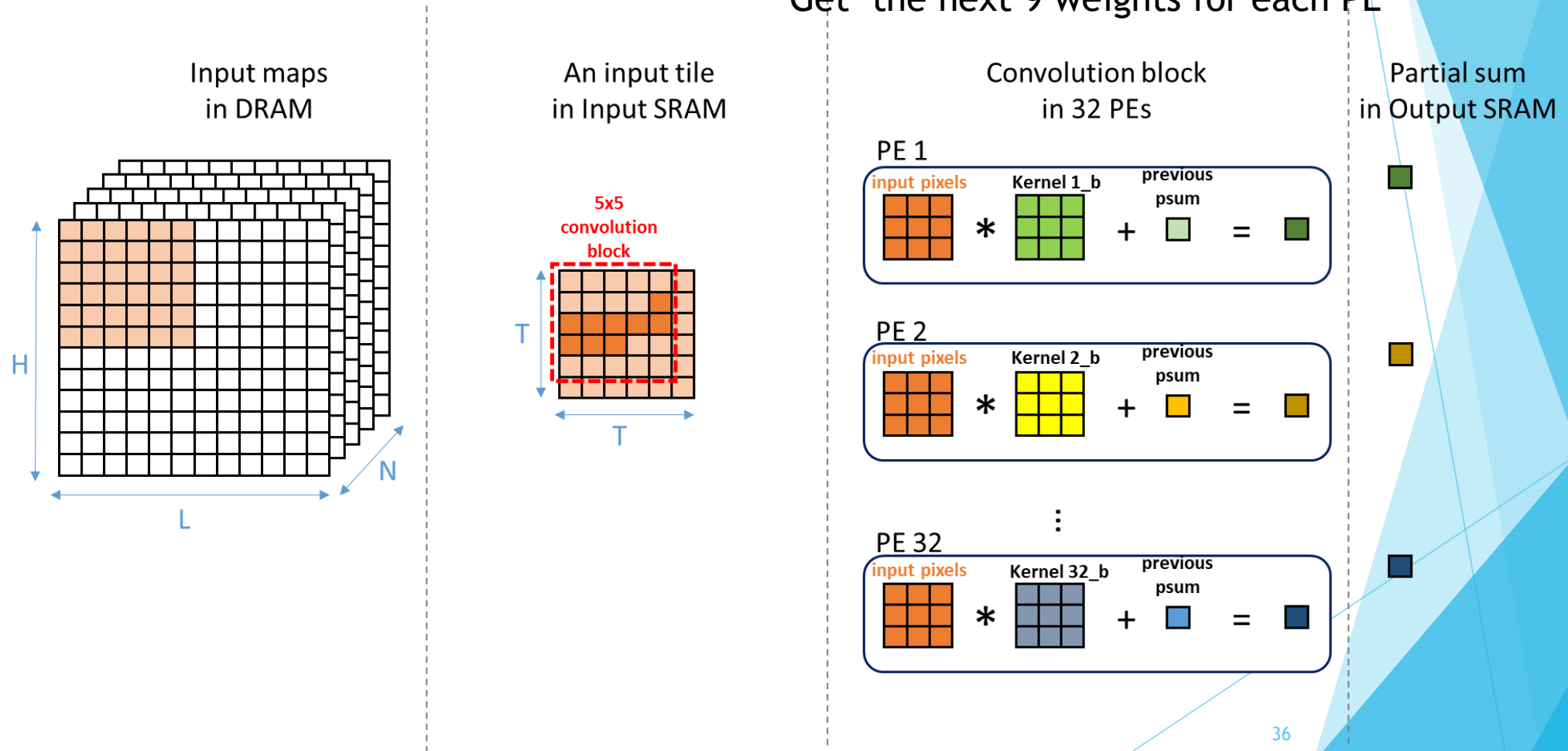


A filter on a PE

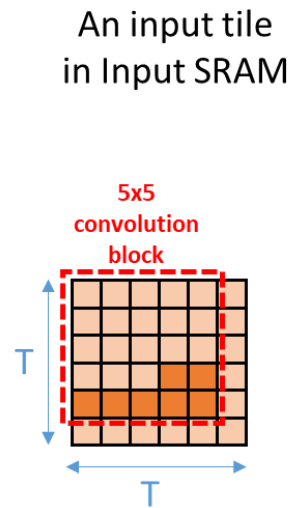
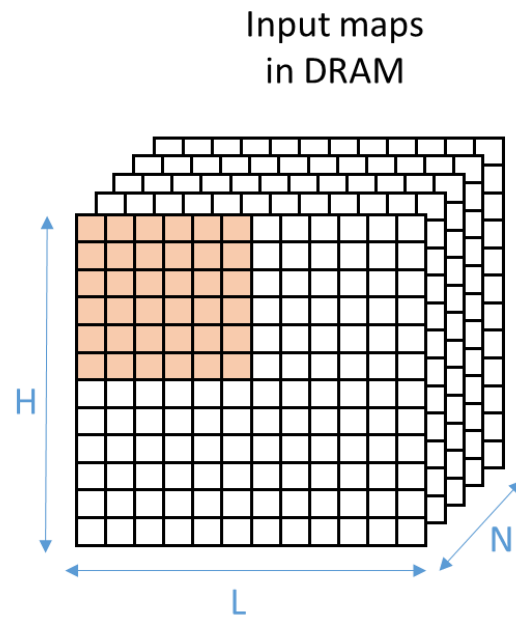
Partial sum in Output SRAM



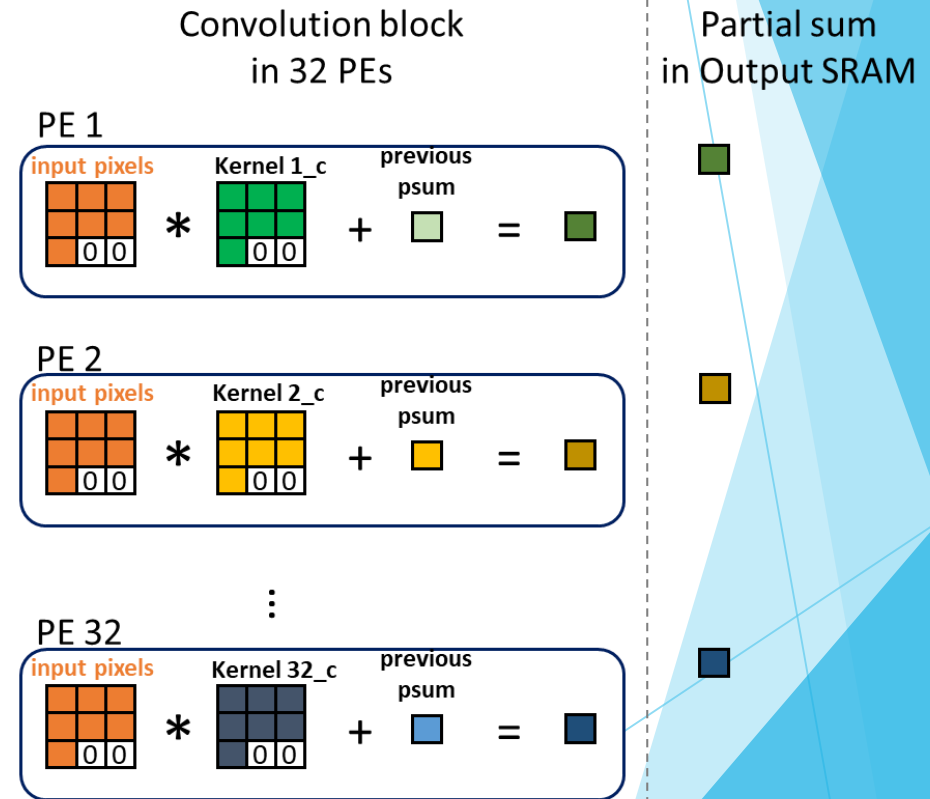
# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel in Action (2/6)



# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel in Action (3/6)

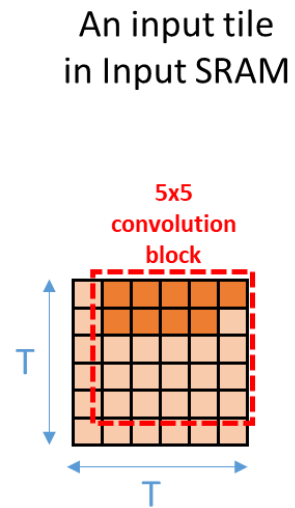
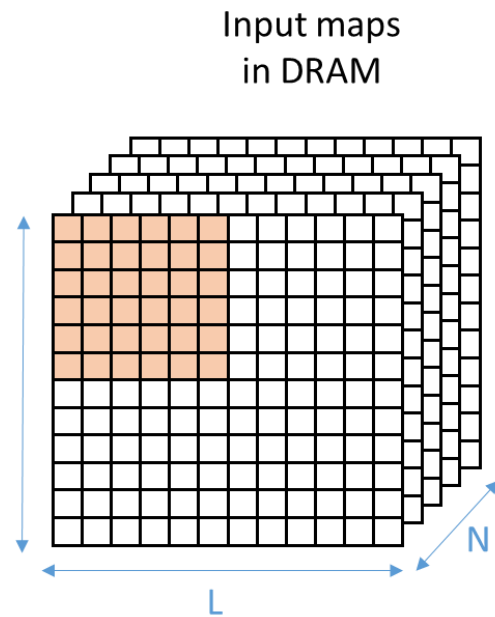


Get the last 7 weights for each PE



# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel (4/6)

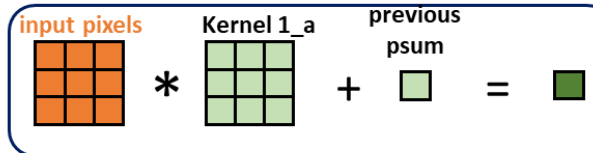
Get the first 9 weights for each PE again



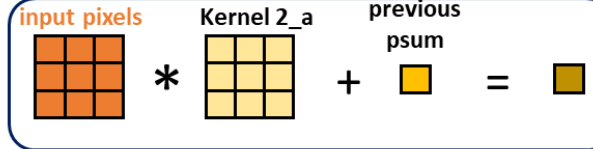
Let's move the box for 1 pixel  
Stride = 1

Convolution block  
in 32 PEs

PE 1

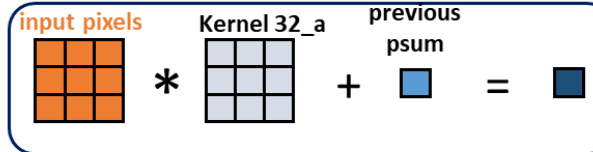


PE 2



⋮

PE 32

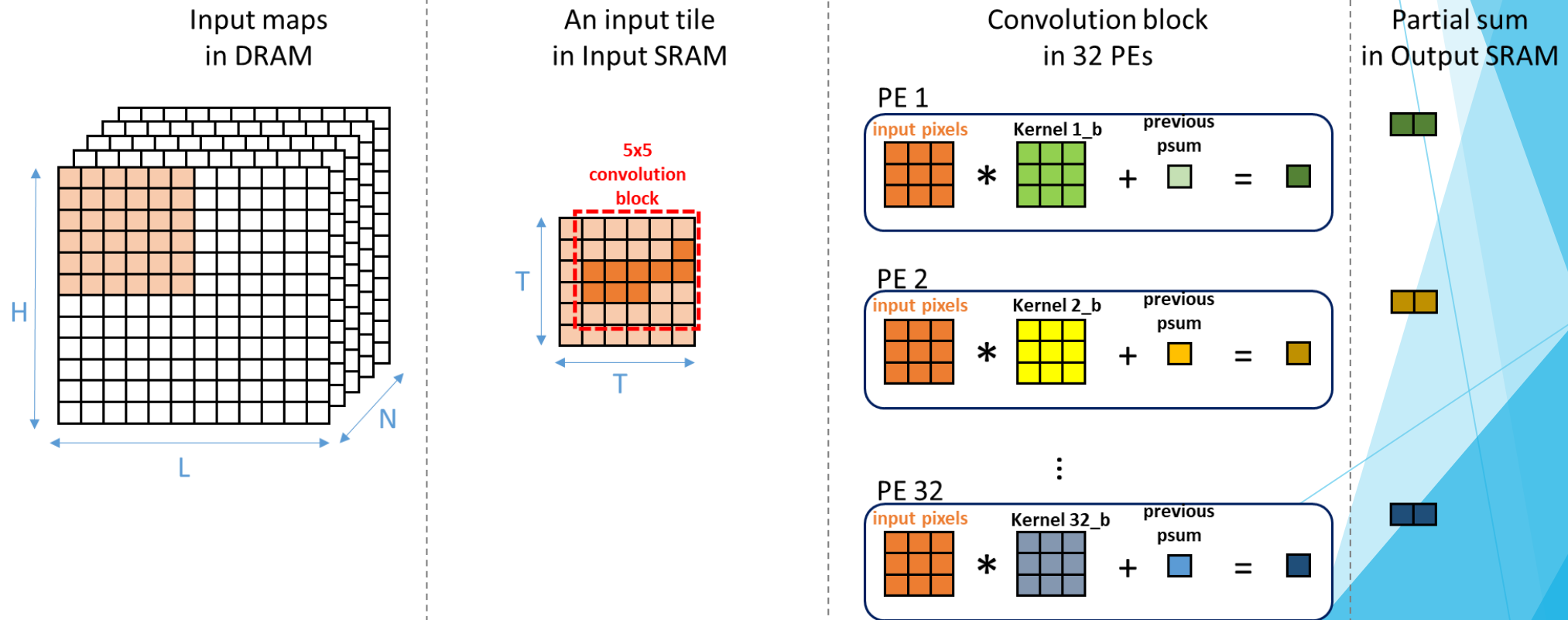


Partial sum  
in Output SRAM



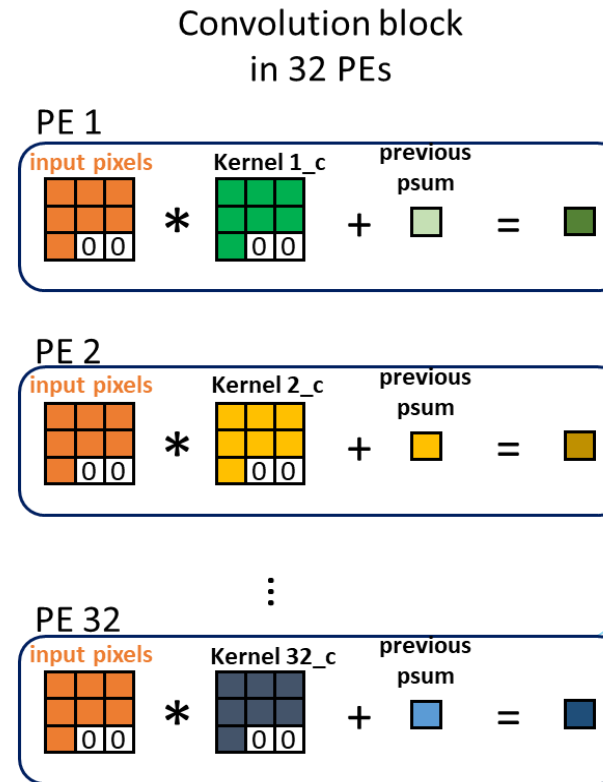
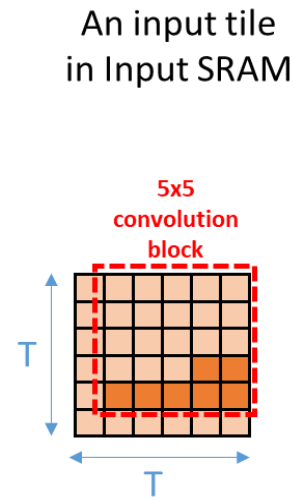
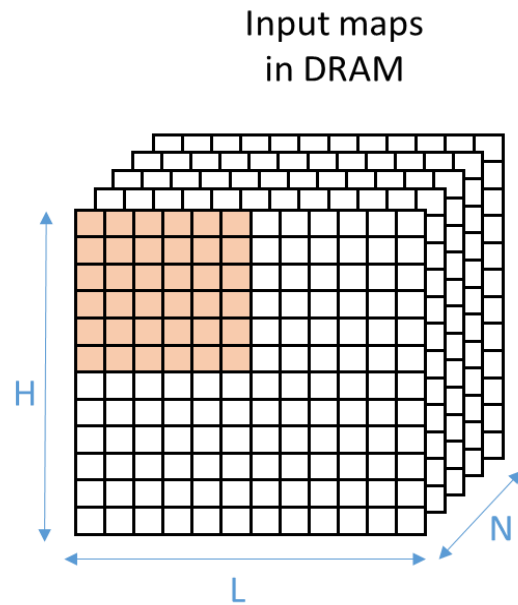
# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel (5/6)

Get the next 9 weights for each PE again



# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel (6/6)

Get the last 7 weights for each PE again



Partial sum in Output SRAM



A stack of partial sums is shown in Output SRAM. The partial sums are shown as small colored squares (green, yellow, blue) corresponding to the results of the convolution operations in the PEs.



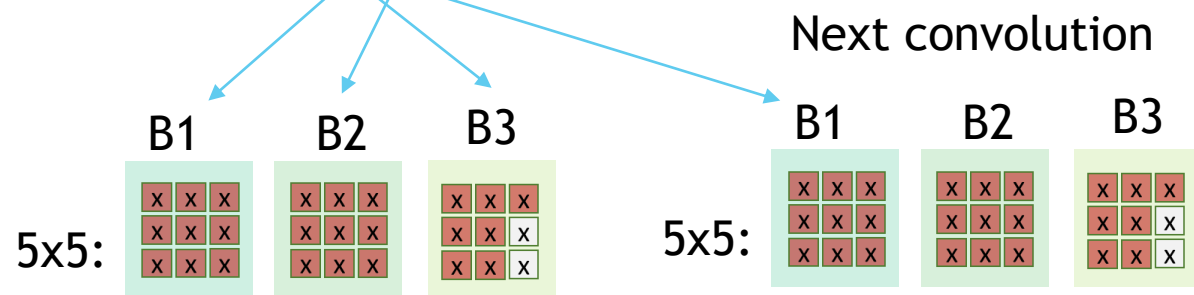
# Kernel Row-major Order Mapping on the Same PE for Oversize Kernel


## ► Observations:

► Reload 9 weights when switching to the next batch

► Reload weights when striding

►



After first batch B1, need to load the next 9 weights for batch B2. This is a read for weight SRAM. 

# Optimization: Weight Stationary for Oversize Kernel

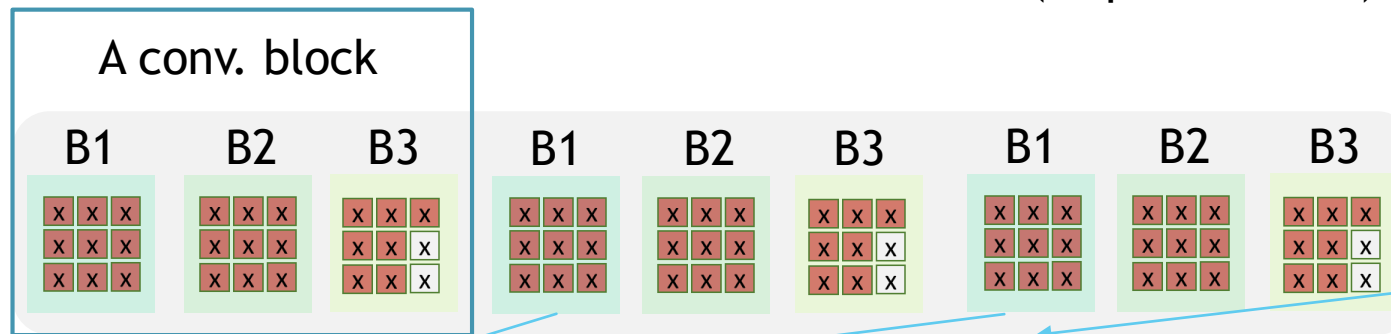
- Optimization: Group the same batches in all different conv blocks **in a tile together**



Striding

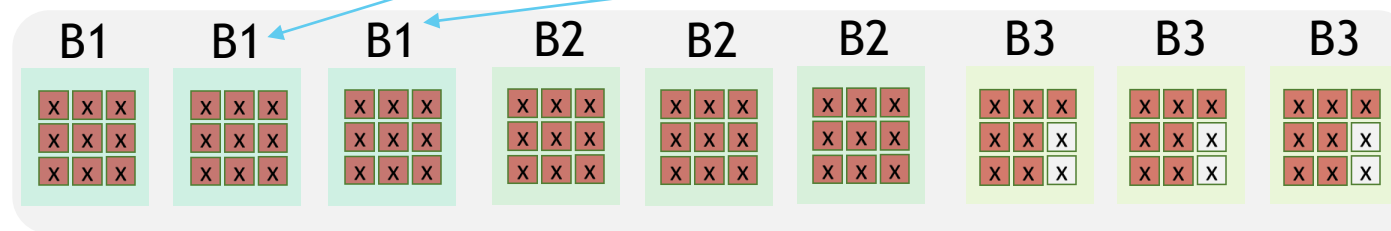
Bi: batch order (or phase order) for a oversize kernel

Original



=>  $3 * 3 = 9$  times

Optimized



=> 3 times

# Optimization: Weight Stationary for Oversize Kernel

- ▶ Optimization: Group the same batches in all different conv blocks **in a tile together**
- ▶ Advantage:
  - ▶ 1: Address generation for input data becomes easy by using conv. block size as an offset.
  - ▶ 2: **Avoid weight thrashing**

# How about 1x1 kernels? The PE is 3x3.

YOLOv3-tiny	Model parameter			Utilization rate of 32 PEs	Utilization rate of 288 multipliers
	Input map	Kernel size	Output map		
conv1	416 x 416 x 3	3	416 x 416 x 16	50 %	50 %
conv2	208 x 208 x 16	3	208 x 208 x 32	100 %	100 %
conv3	104 x 104 x 32	3	104 x 104 x 64	100 %	100 %
conv4	52 x 52 x 64	3	52 x 52 x 128	100 %	100 %
conv5	26 x 26 x 128	3	26 x 26 x 256	100 %	100 %
conv6	13 x 13 x 256	3	13 x 13 x 512	100 %	100 %
conv7	13 x 13 x 512	3	13 x 13 x 1024	100 %	100 %
conv8	13 x 13 x 1024	1	13 x 13 x 256	100 %	11.11 %
conv9	13 x 13 x 256	3	13 x 13 x 512	100 %	100 %
conv10	13 x 13 x 512	1	13 x 13 x 255	99.6 %	11.07 %
conv11	13 x 13 x 256	1	13 x 13 x 128	100 %	11.11 %
conv12	26 x 26 x 384	3	26 x 26 x 256	100 %	100 %
conv13	26 x 26 x 256	1	26 x 26 x 255	99.6 %	11.07 %

99.09 %

68.8 %

# Optimization for 1x1 Kernel Size

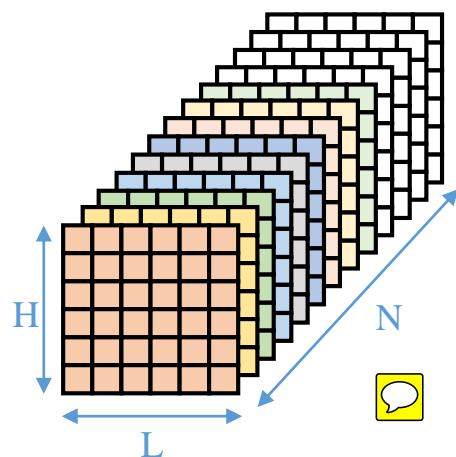
- ▶ When input map size is smaller than 52x52 (input tile size) and kernel size is 1x1, transfer more than **1 input map of different input channels** to Input SRAM. The convolution of these input maps in the input SRAM would be executed at the same time. Assume Input SRAM size = tile size.

## Example

YOLOv3-tiny	Input map	Kernel size	Max input map # that Input SRAM can store	Input map # stored in Input SRAM
conv8	13x13	1	16	9
conv10	13x13	1	16	9
conv11	13x13	1	16	9
conv13	26x26	1	4	4

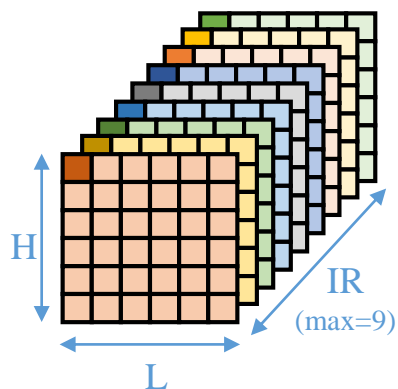
# Optimization for 1x1 Kernel Size (1/4)

Input maps  
in DRAM



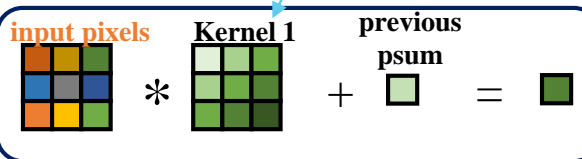
H x L is the dimension  
of input map, e.g., 13x13  
Less than tile size

IR input maps  
in Input SRAM

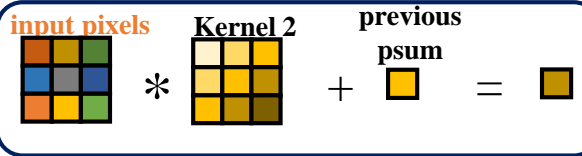


Get at max 9 channels of  
kernel 1

PE 1

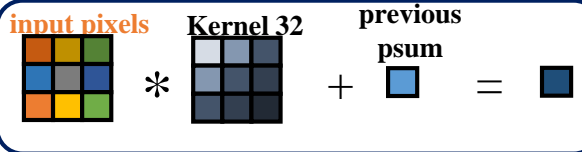


PE 2



⋮

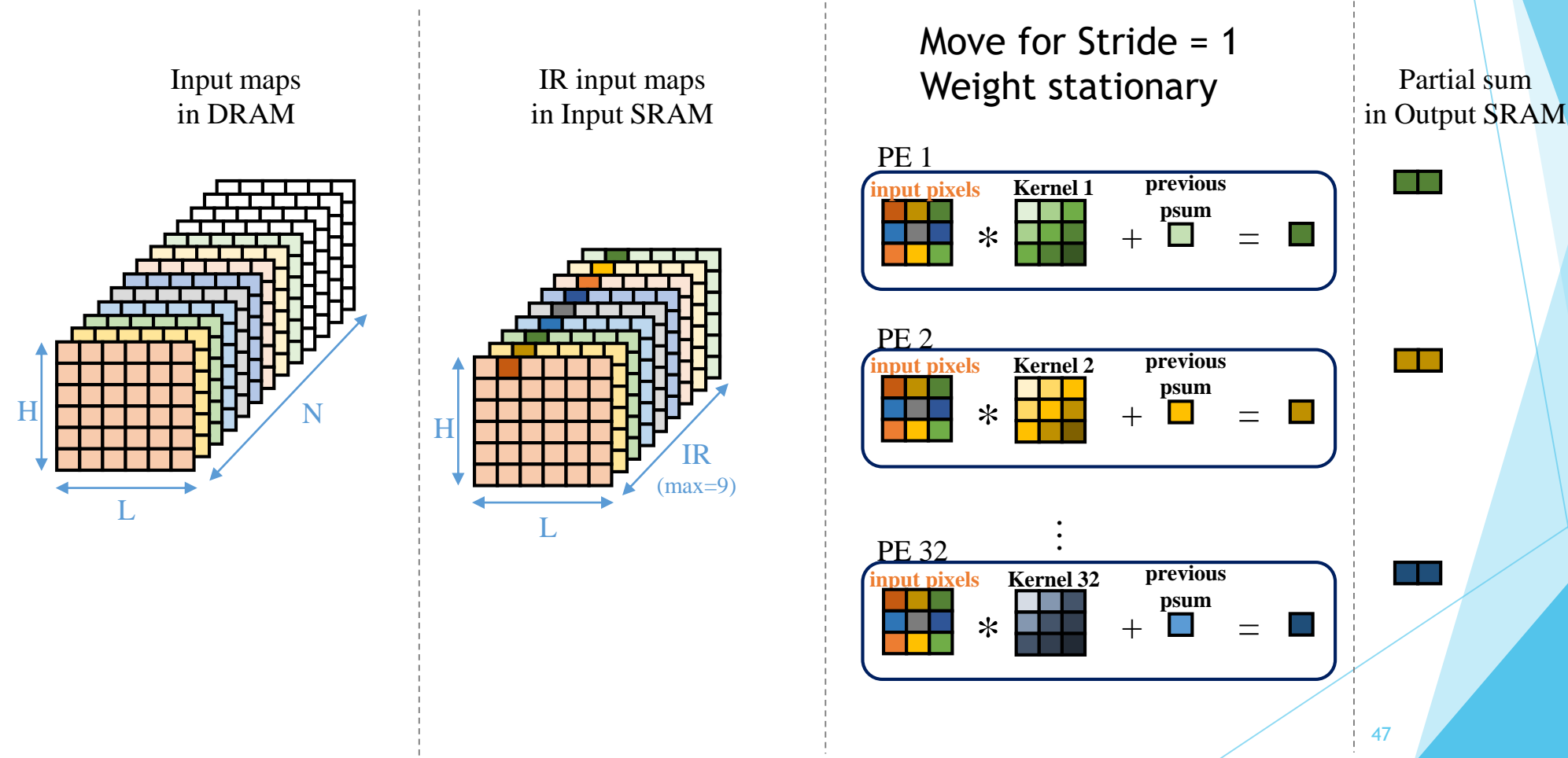
PE 32



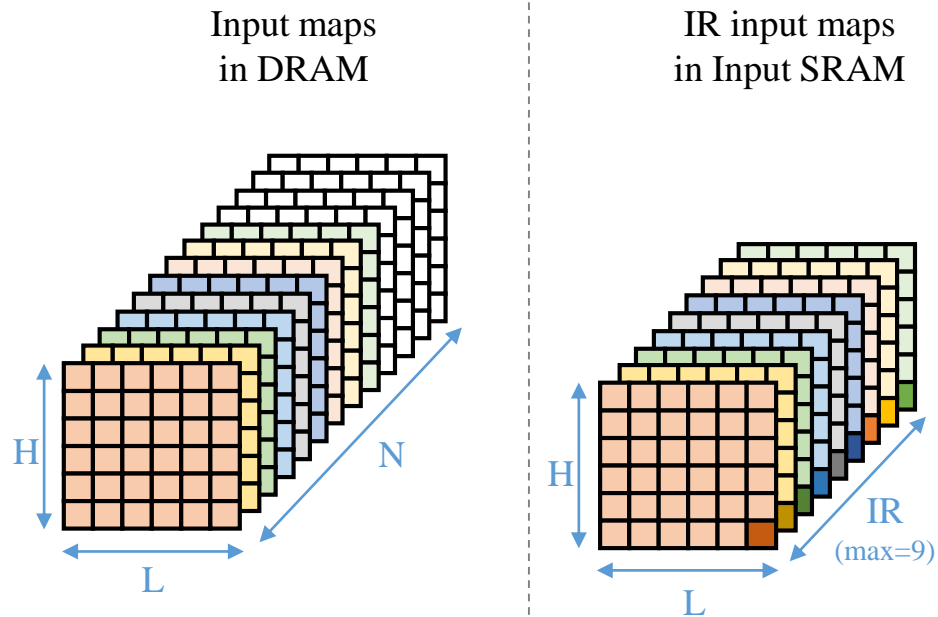
Partial sum  
in Output SRAM



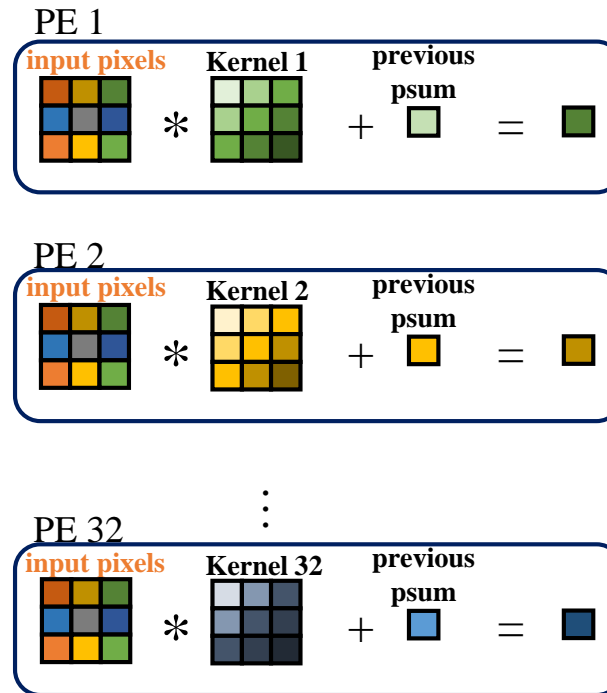
# Optimization for 1x1 Kernel Size (2/4)



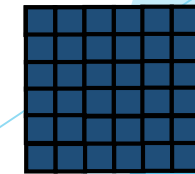
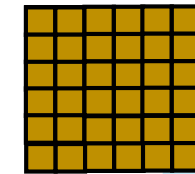
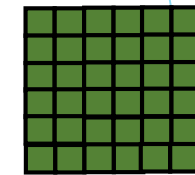
# Optimization for 1x1 Kernel Size (3/4)



## End of H x L convolutions



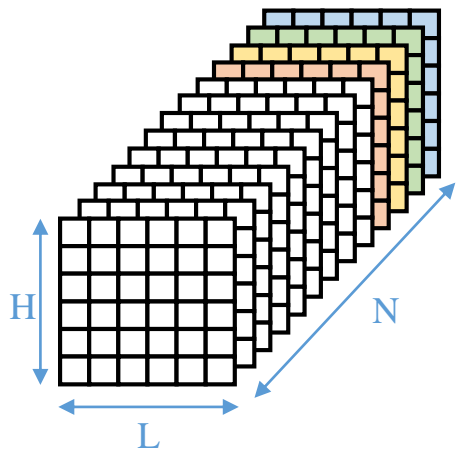
## Partial sum in Output SRAM



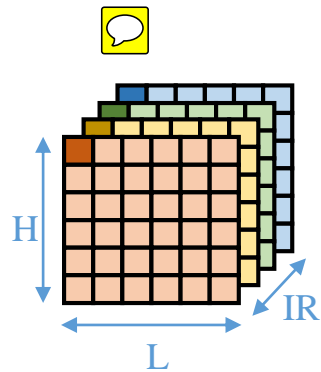


# Optimization for 1x1 Kernel Size (4/4)

Input maps in DRAM



IR input maps in Input SRAM



The rest of channels  
Load channel weights

PE 1

input pixels \* Kernel 1 + previous psum =

0	0
0	0

0	0
0	0

0	0
0	0

PE 2

input pixels \* Kernel 2 + previous psum =

0	0
0	0

0	0
0	0

0	0
0	0

PE 32

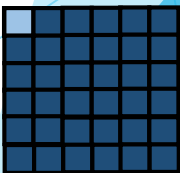
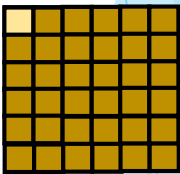
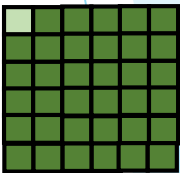
input pixels \* Kernel 32 + previous psum =

0	0
0	0

0	0
0	0

0	0
0	0

Partial sum in Output SRAM

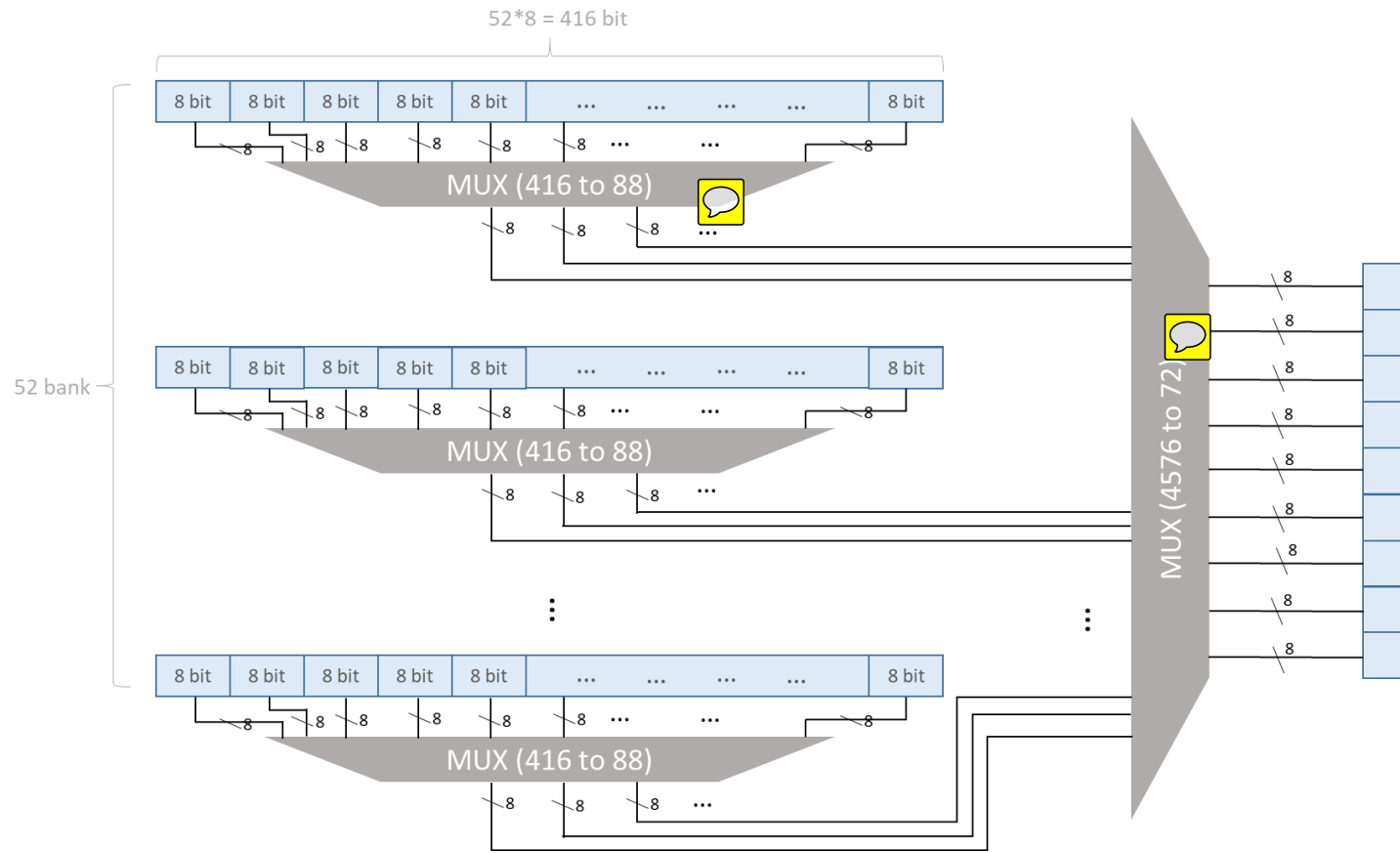


# Project - part B:

## PE accelerator

- ▶ Continuing what you did last time with project-partA. You need to add PE unit to your design, then you need an output SRAM to store the partial sum or the result of the operation, and finally write it back to the DRAM.
- ▶ You need to consider the following issues:
  1. how to dispatch different kernel size to pe for calculation(3x3, 5x5)
  2. PE array structure
  3. PE utilization
  4. data reuse
  5. data streaming from SRAM to PE
- ▶ RTL or SystemC code design
- ▶ Report due in 4 weeks.

# Input SRAM Example

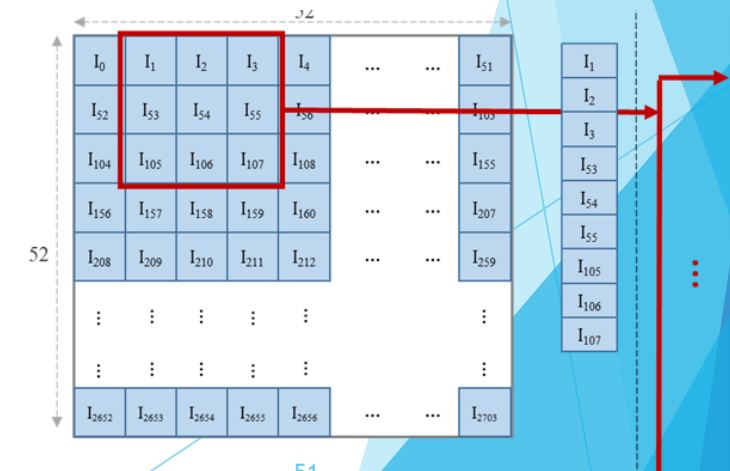


## Pros

- Can get all data in a cycle

## Cons

- Power consumption, datapath,
- Mux tree delay



# PEs' utilization rate of YOLOv3-tiny



YOLOv3-tiny	Model parameter			Utilization rate of 32 PEs	Utilization rate of 288 multipliers	2D PE- Utilization rate of 32x32 PEs
	Input map	Kernel size	Output map			
conv1	416 x 416 x 3	3	416 x 416 x 16	50 %	50 %	49.992 %
conv2	208 x 208 x 16	3	208 x 208 x 32	100 %	100 %	99.968 %
conv3	104 x 104 x 32	3	104 x 104 x 64	100 %	100 %	99.935 %
conv4	52 x 52 x 64	3	52 x 52 x 128	100 %	100 %	99.871 %
conv5	26 x 26 x 128	3	26 x 26 x 256	100 %	100 %	99.742 %
conv6	13 x 13 x 256	3	13 x 13 x 512	100 %	100 %	99.485 %
conv7	13 x 13 x 512	3	13 x 13 x 1024	100 %	100 %	99.742 %
conv8	13 x 13 x 1024	1	13 x 13 x 256	100 %	11.11 % → 99.81 %	98.848 %
conv9	13 x 13 x 256	3	13 x 13 x 512	100 %	100 %	99.485 %
conv10	13 x 13 x 512	1	13 x 13 x 255	99.6 %	11.07 % → 99.41 %	97.722 %
conv11	13 x 13 x 256	1	13 x 13 x 128	100 %	11.11 % → 98.08 %	95.545 %
conv12	26 x 26 x 384	3	26 x 26 x 256	100 %	100 %	99.914 %
conv13	26 x 26 x 256	1	26 x 26 x 255	99.6 %	11.07 % → 44.27 %	98.848 %
				99.09 %	91.66 %	95.952 %

# PEs' utilization rate of VGG16

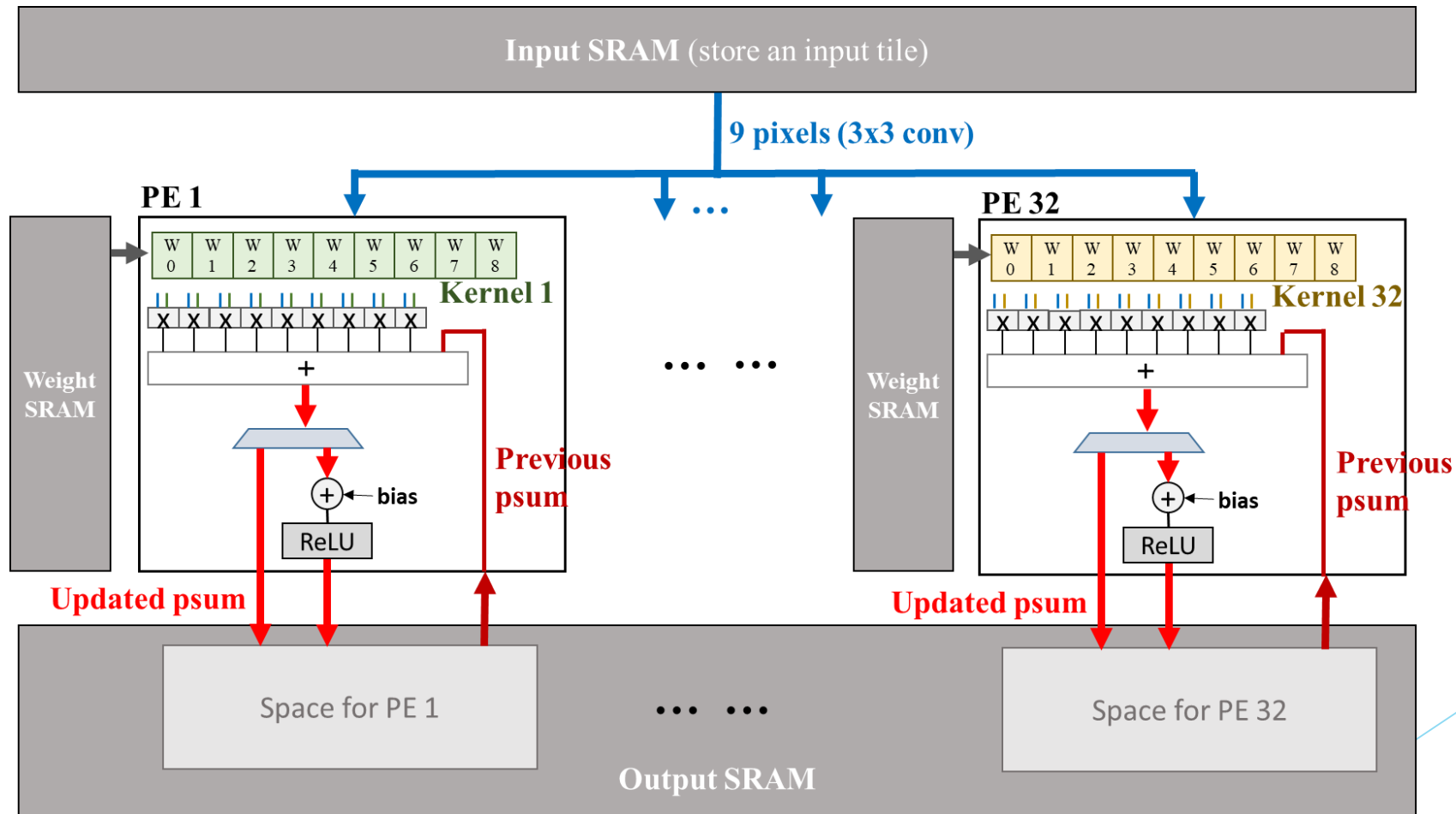
VGG16	Model parameter			Utilization rate of 32 PEs	Utilization rate of 288 multipliers	2D PE- Utilization rate of 32x32 PEs	Eyeriss Num. of active PEs
	Input map	Kernel size	Output map				
conv1	224 x 224 x 3	3	224 x 224 x 64	100 %	100 %	86.182 %	156 (93%)
conv2	224 x 224 x 64	3	224 x 224 x 64	100 %	100 %	99.995 %	156 (93%)
conv3	112 x 112 x 64	3	112 x 112 x 128	100 %	100 %	99.992 %	156 (93%)
conv4	112 x 112 x 128	3	112 x 112 x 128	100 %	100 %	99.996 %	156 (93%)
conv5	56 x 56 x 128	3	56 x 56 x 256	100 %	100 %	99.990 %	156 (93%)
conv6	56 x 56 x 256	3	56 x 56 x 256	100 %	100 %	99.995 %	156 (93%)
conv7	56 x 56 x 256	3	56 x 56 x 256	100 %	100 %	99.995 %	156 (93%)
conv8	28 x 28 x 256	3	28 x 28 x 512	100 %	100 %	99.990 %	168 (100%)
conv9	28 x 28 x 512	3	28 x 28 x 512	100 %	100 %	99.995 %	168 (100%)
conv10	28 x 28 x 512	3	28 x 28 x 512	100 %	100 %	99.995 %	168 (100%)
conv11	14 x 14 x 512	3	14 x 14 x 512	100 %	100 %	99.982 %	168 (100%)
conv12	14 x 14 x 512	3	14 x 14 x 512	100 %	100 %	99.982 %	168 (100%)
conv13	14 x 14 x 512	3	14 x 14 x 512	100 %	100 %	99.982 %	168 (100%)
				100 %	100 %	98.928 %	96.23 %

# PEs' utilization rate of AlexNet

AlexNet	Model parameter			Utilization rate of 32 PEs	Utilization rate of 288 multipliers	2D PE-Utilization rate of 32x32 PEs	Eyeriss Num. of active PEs
	Input map	Kernel size	Output map				
conv1	227 x 227 x 3	11	55 x 55 x 96	100 %	96.03 %	70.542 %	154 (92%)
conv2	31 x 31 x 48	5	27 x 27 x 256	100 %	92.6 %	99.957 %	135 (80%)
conv3	15 x 15 x 256	3	13 x 13 x 384	100 %	100 %	99.943 %	156 (93%)
conv4	15 x 15 x 192	3	13 x 13 x 384	100 %	100 %	99.924 %	156 (93%)
conv5	15 x 15 x 192	3	13 x 13 x 256	100 %	100 %	99.886 %	156 (93%)
				100 %	97.726 %	94.050 %	90.2 %

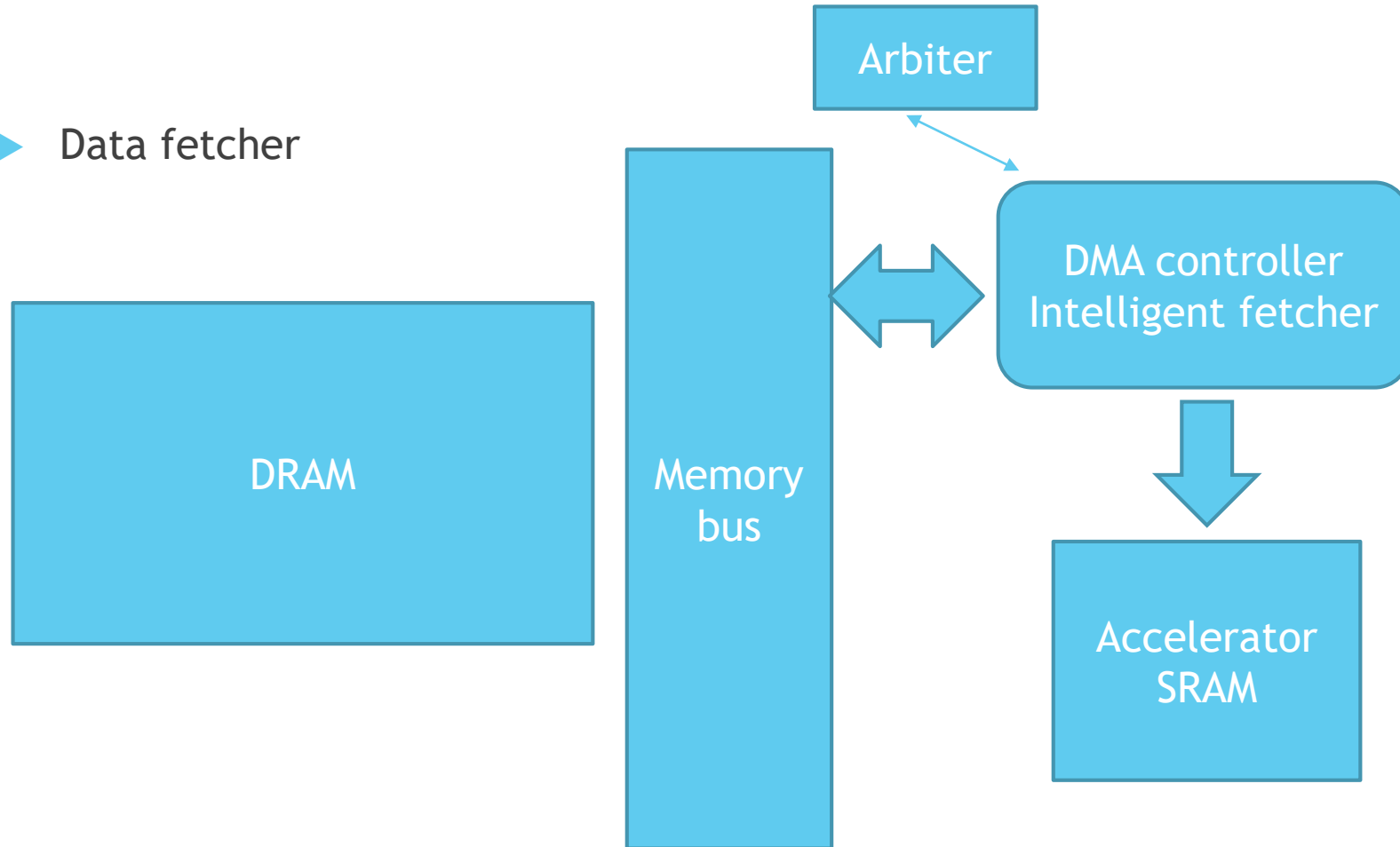
# Convolution, Bias Addition, and ReLU PE Core

- Convolution, bias ,and ReLU in PE



# Memory System

## ► Data fetcher





# Project - part C:

## Complete 1D PE accelerator

- ▶ Continue project partA and partB, this time you need to finish a complete PE accelerator.
- ▶ Your accelerator needs to be able to apply to different input map size and kernel size, and able to do the following things:
  1. convolution
  2. bias
  3. Relu
- ▶ In order to make your design more complete, you need to add bus and arbiter to your design.
- ▶ Enhance your design by considering all the optimizations addressed so far.
- ▶ Can be designed with RTL or SystemC code
- ▶ Presentation due in final week.

# Machine Learning

Hello!! R2-D2



Thanks You