Lecture 10: Machine Learning Compiler and Introduction to AI Accelerator

Some Notes

- Midterm grade will be posted by this weekend.
- There will be a around of project meeting on April 24.
- Quiz today on Federated Learning.

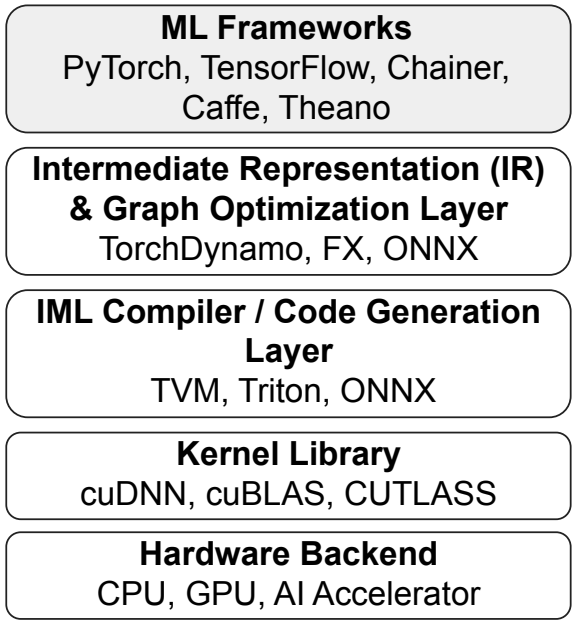
Recap

- Federated Learning
- Distributed DNN Training
- Distributed DNN Inference

Topics

- Machine Learning Compiler
- Introduction to AI Accelerator
- CNN Accelerator

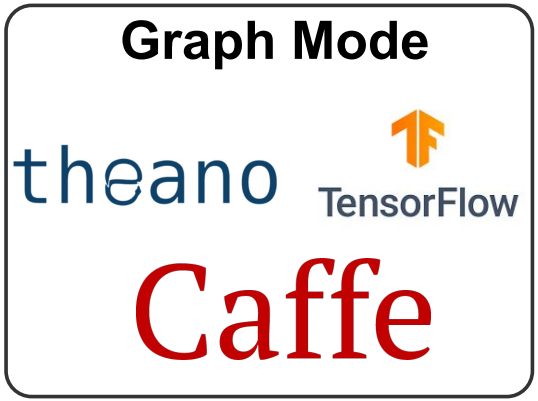
ML Software–Compiler stack

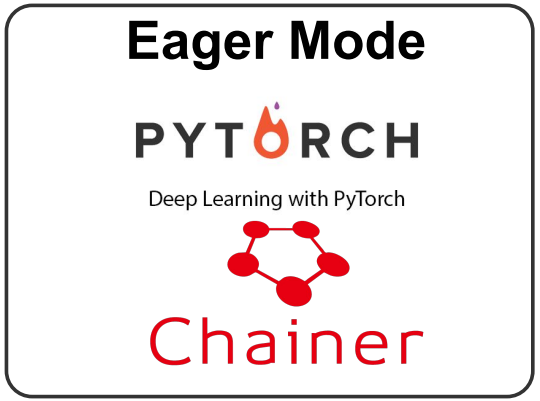


- High-level interfaces where models are defined, trained, and debugged.
- Converts model definitions into an analyzable graph IR for optimization.
- Translates the IR into optimized kernels and executable code for specific devices.
- Executes optimized kernels, manages device memory, and provides primitives.
- The physical devices executing the compiled and optimized models.

NYU SAI LAB

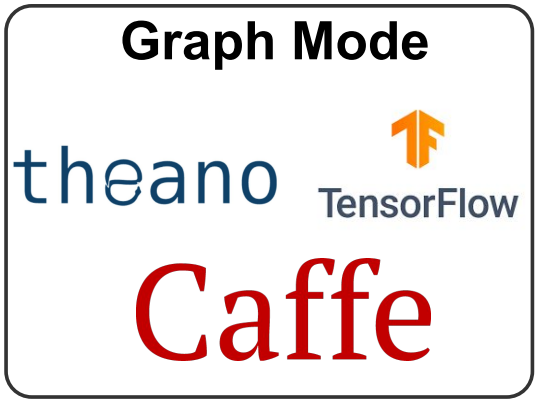
Deep Learning Frameworks





- **Graph mode:** where they expose a graph building API that requires users to first construct a graph and then later execute that graph.
- **Eager mode:** meaning operations are executed immediately as they are called in Python, rather than being added to a static computation graph (as in early TensorFlow).

Deep Learning Frameworks



- **Graph mode:** where they expose a graph building API that requires users to first construct a graph and then later execute that graph.
- **Eager mode:** meaning operations are executed immediately as they are called in Python, rather than being added to a static computation graph (as in early TensorFlow).

Pytorch

```
class LinearLayer(Module):
   def __init__(self, in_sz, out_sz):
      super().__init__()
      t1 = torch.randn(in_sz, out_sz)
      self.w = nn.Parameter(t1)
      t2 = torch.randn(out_sz)
      self.b = nn.Parameter(t2)

   def forward(self, activations):
      t = torch.mm(activations, self.w)
      return t + self.b
```

```
class FullBasicModel(nn.Module):
   def __init__(self):
      super().__init__()
      self.conv = nn.Conv2d(1, 128, 3)
      self.fc = LinearLayer(128, 10)

   def forward(self, x):
      t1 = self.conv(x)
      t2 = nn.functional.relu(t1)
      t3 = self.fc(t1)
      return nn.functional.softmax(t3)
```

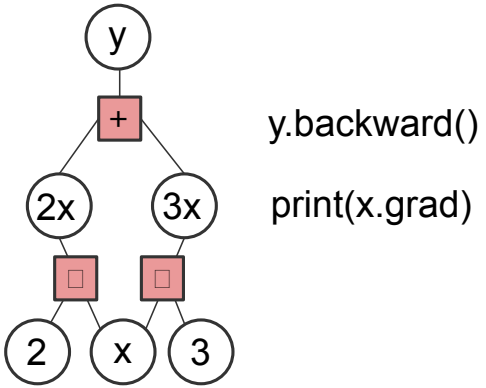
```
discriminator = create_discriminator()
generator = create_generator()
optimD = optim.Adam(discriminator.parameters())
optimG = optim.Adam(generator.parameters())

def step(real_sample):
  # (1) Update Discriminator
  errD_real = loss(discriminator(real_sample), real_label)
  errD_real.backward()
  fake = generator(get_noise())
  errD_fake = loss(discriminator(fake.detach(), fake_label)
  errD_fake.backward()
  optimD.step()
  # (2) Update Generator
  errG = loss(discriminator(fake), real_label)
  errG.backward()
  optimG.step()
```

- Inherit from Chainer.
- DNN models can be programmed. API is defined to make user freely select the layer configuration.
- This “everything is a just a program” philosophy is not limited to just the models, and applies to optimizers and data loaders as well.

Pytorch

- PyTorch allows for bidirectional exchange of data with external libraries.
  - For example, it provides a mechanism to convert between NumPy arrays and PyTorch tensors using the torch.from_numpy() function and .numpy() tensor method.
  - Pytorch has implemented an automatic differentiation functionality to automatically performance the gradient computation.

```
import torch
x = torch.tensor(2.0, requires_grad=True)
y = x*2+x*3
```



NYU SAI LAB

Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." *Advances in neural information processing systems* 32 (2019).

Pytorch

- What if the function is not differentiable? How to generate gradient?
  - Subgradients or Piecewise Derivatives
  - Approximation Gradients
  - Customized gradient definition
  - Compilation Error

Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." *Advances in neural information processing systems* 32 (2019).

Pytorch

- Subgradients or Piecewise Derivatives

```
import torch
x = torch.tensor(0.0, requires_grad=True)
y = torch.abs(x)
y.backward()
print(x.grad)  # prints 0.0
```

- Straight-Through Estimator (STE)

```
def binary_step_ste(x):
    y = (x > 0).float()
    y.backward = lambda grad: grad  #
approximate grad as 1
```

- Customized gradient definition

```
class CustomOp(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return torch.sign(x)
    @staticmethod
    def backward(ctx, grad_output):
        x, = ctx.saved_tensors
        grad_input = grad_output * (x.abs()
<= 1).float()  # custom surrogate
        return grad_input
```

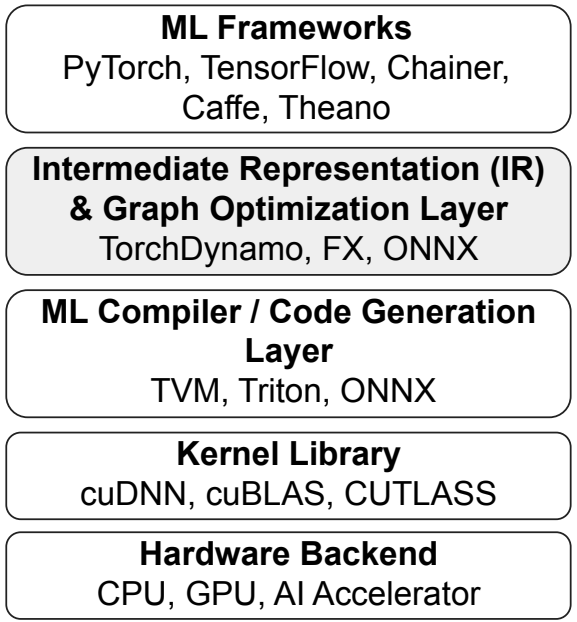

Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." *Advances in neural information processing systems* 32 (2019).

Pytorch

- PyTorch maintains a strict separation between its control (i.e. program branches, loops) and data flow (i.e. tensors and the operations performed on them).
- The resolution of the control flow is handled by Python and optimized C++ code executed on the host CPU, and result in a linear sequence of operator invocations on the device. Operators can be run either on CPU or on GPU.
- Memory allocation is first performed on the CPU, which subsequently manages the allocation and mapping on the GPU.

Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." *Advances in neural information processing systems* 32 (2019).

ML Software–Compiler stack



- High-level interfaces where models are defined, trained, and debugged.
- Converts model definitions into an analyzable graph Intermediate Representation (IR) for optimization.
- Translates the IR into optimized kernels and executable code for specific devices.
- Executes optimized kernels, manages device memory, and provides primitives.
- The physical devices executing the compiled and optimized models.

TensorFlow

- Open-source ML framework by Google (2015)
- Supports CPU, GPU, TPU acceleration
  - Neural networks (CNNs, RNNs, Transformers)
  - Reinforcement learning
  - Signal processing and scientific computing
- TensorFlow builds a **static computation graph** (predefined dataflow)
  - Graph nodes = operations
  - Edges = data (tensors)

- Optimizable and deployable on different hardware
- Enables **parallelism** and **graph-level optimizations**

Abadi, Martín, et al. "{TensorFlow}: a system for {Large-Scale} machine learning." *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016.

TensorFlow

```
import tensorflow as tf

x = tf.constant(3.0)
y = tf.constant(4.0)
z = x * y  # <-- nothing runs yet, just builds the graph
print(z)
```

```
Tensor("mul:0", shape=(), dtype=float32)
```

```
with tf.Session() as sess:
    print(sess.run(z))  # Output: 12.0
```

```
import tensorflow as tf

x = tf.constant(3.0)
y = tf.constant(4.0)
z = x * y  # executed immediately
print(z)
```

```
tf.Tensor(12.0, shape=(), dtype=float32)
```

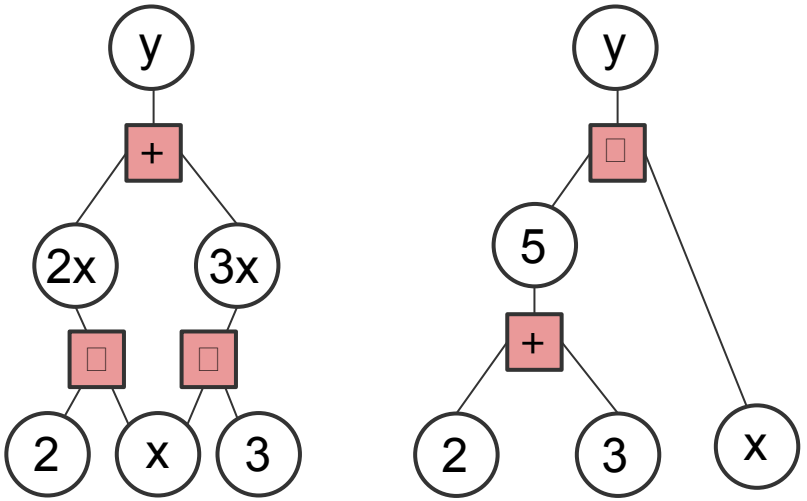

Abadi, Martín, et al. "{TensorFlow}: a system for {Large-Scale} machine learning." *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016.

Graph-level Optimization

```
import torch
x = torch.tensor(2.0, requires_grad=True)
y = x*2+x*3
```

```
import torch
x = torch.tensor(2.0, requires_grad=True)
y = x*(2+3)
```

The downside of eager mode frameworks is that they make it harder to apply graph-level optimizations through compilers.



Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." *Advances in neural information processing systems* 32 (2019).

Torch.fx

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class RedundantModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(4, 8)
        self.fc2 = nn.Linear(8, 8)
        self.fc3 = nn.Linear(8, 2)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = F.relu(x)     # redundant ReLU
        x = self.fc2(x)
        x = F.relu(x)
        return self.fc3(x)
```

```
from torch.fx import symbolic_trace

model = RedundantModel()
gm = symbolic_trace(model)
print(gm.graph)
```

```
graph():
    %x = placeholder[target=x]
    %fc1 = call_module[target=fc1](args=(%x,), kwargs={})
    %relu_1 = call_function[target=torch.nn.functional.relu](args=(%fc1,), kwargs={})
    %relu_2 = call_function[target=torch.nn.functional.relu](args=(%relu_1,), kwargs={})
    %fc2 = call_module[target=fc2](args=(%relu_2,), kwargs={})
    %relu_3 = call_function[target=torch.nn.functional.relu](args=(%fc2,), kwargs={})
    %fc3 = call_module[target=fc3](args=(%relu_3,), kwargs={})
    return %fc3
```


Ansel, Jason, et al. "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation." *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2024.

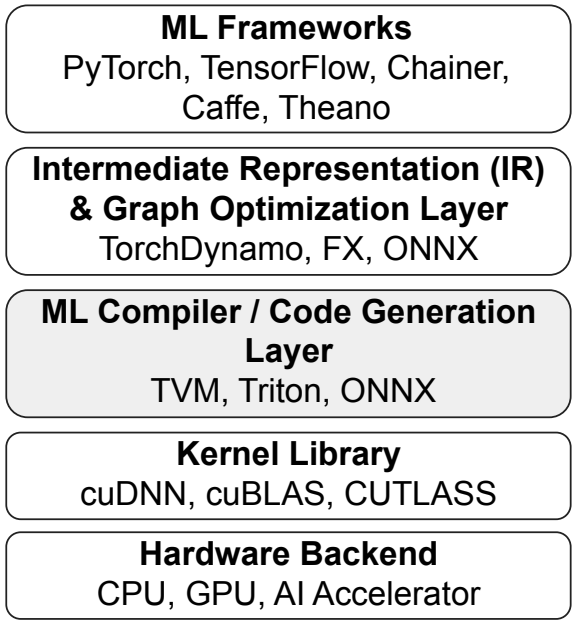
Torch.fx

```
def remove_redundant_relu(gm):
    new_graph = torch.fx.Graph()
    env = {}

    last_op = None
    for node in gm.graph.nodes:
        if node.op == 'call_function' and node.target == torch.nn.functional.relu:
            # Skip consecutive ReLU nodes
            if last_op == torch.nn.functional.relu:
                continue
        new_node = new_graph.node_copy(node, lambda x: env[x])
        env[node] = new_node
        last_op = node.target if node.op == 'call_function' else None

    gm_optimized = torch.fx.GraphModule(gm, new_graph)
    return gm_optimized
```

Torch.fx provides the flexibility to modify and transform the computation graph, enabling performance and efficiency improvements.

Ansel, Jason, et al. "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation." *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2024.

ML Software–Compiler stack



- High-level interfaces where models are defined, trained, and debugged.
- Converts model definitions into an analyzable graph IR for optimization.
- Translates the IR into optimized kernels and executable code for specific devices.
- Executes optimized kernels, manages device memory, and provides primitives.
- The physical devices executing the compiled and optimized models.

NYU SAI LAB

Triton

- An open-source compiler and language (originally by Harvard/OpenAI, now integrated into PyTorch)
- Allows writing custom GPU kernels in Python, achieving CUDA-level performance with much simpler code
- **Automatic tiling & vectorization** for performance portability
- **Fusion-friendly**: easily integrates with PyTorch's graph optimizers

Tillet, Philippe, Hsiang-Tsung Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations." *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019.

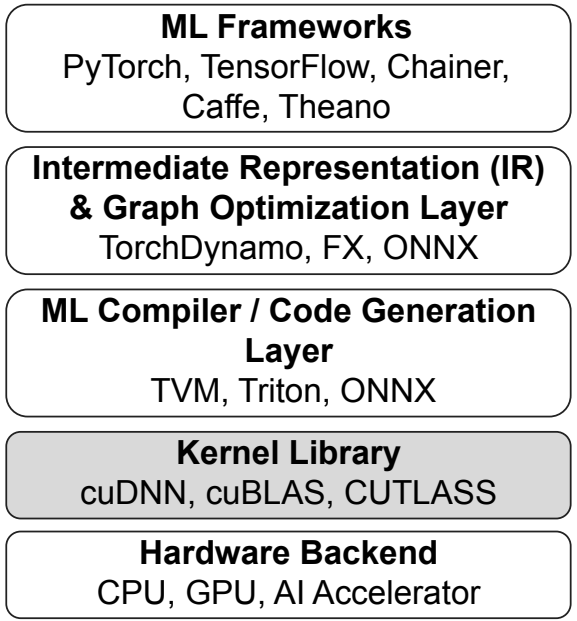
Triton

```
import triton
import triton.language as tl

@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K):
    pid = tl.program_id(0)
    row = pid * 16 + tl.arange(0, 16)
    col = tl.arange(0, 16)
    a = tl.load(a_ptr + row[:, None] * K + tl.arange(0, K))
    b = tl.load(b_ptr + tl.arange(0, K)[:, None] * N + col)
    c = tl.dot(a, b)
    tl.store(c_ptr + row[:, None] * N + col, c)
```

- Writing efficient GPU kernels in CUDA is complex and error-prone.
- Researchers often need custom fused kernels beyond what cuDNN/cuBLAS offer.
- Frameworks like PyTorch needed a flexible but high-performance solution.
- Triton bridges this gap: Python-like syntax with compiler-grade optimization.

Tillet, Philippe, Hsiang-Tsung Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations." *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2019.

ML Software–Compiler stack



- High-level interfaces where models are defined, trained, and debugged.
- Converts model definitions into an analyzable graph IR for optimization.
- Translates the IR into optimized kernels and executable code for specific devices.
- Executes optimized kernels, manages device memory, and provides primitives.
- The physical devices executing the compiled and optimized models.

NYU SAI LAB

cuDNN (CUDA Deep Neural Network Library)

- High-performance GPU library for deep learning primitives
- Optimized implementations for:
  - Convolutions, pooling, normalization
  - Activation functions (ReLU, tanh, sigmoid)
  - RNN/LSTM layers
- Automatically used by TensorFlow, PyTorch, and JAX
- Enables Tensor Cores, mixed precision, and algorithm autotuning for speedups

Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." *arXiv preprint arXiv:1410.0759* (2014).

cuBLAS (CUDA Basic Linear Algebra Subprograms)

- GPU-accelerated version of BLAS (Basic Linear Algebra Subroutines)
- Provides fast operations for:
  - Matrix–vector and matrix–matrix multiplications (`GEMM`)
  - Vector scaling, addition, dot products
- Underpins many deep learning operations (e.g., dense layers, attention mechanisms)
- Also supports **FP16 / BF16 precision** for performance on modern GPUs
- cuDNN accelerates *deep learning-specific ops*, while cuBLAS accelerates *general linear algebra*.
- Both are critical layers in the GPU software stack that make frameworks like TensorFlow and PyTorch fast.

Topics

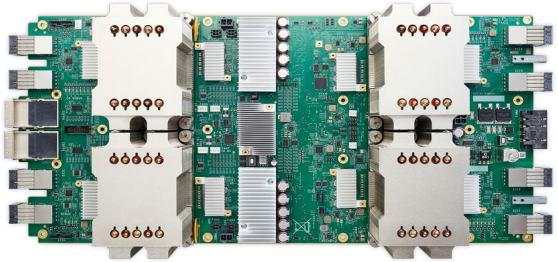
- Machine Learning Compiler
- Introduction to AI Accelerator
- CNN Accelerator

Hardware Support for DNN

- GPU is better than CPU in terms of throughput for both Neural Network training and inference.
  - GPU leverages the highly parallelized architecture of its computing units to handle computational intensive operations.
  - GPU has 10x-20x higher throughput than CPU.
- However, GPU:
  - General purpose.
  - Power consumption and latency is high.
  - Does not support sophisticated pruning and quantization algorithms.

Hardware Support for DNN

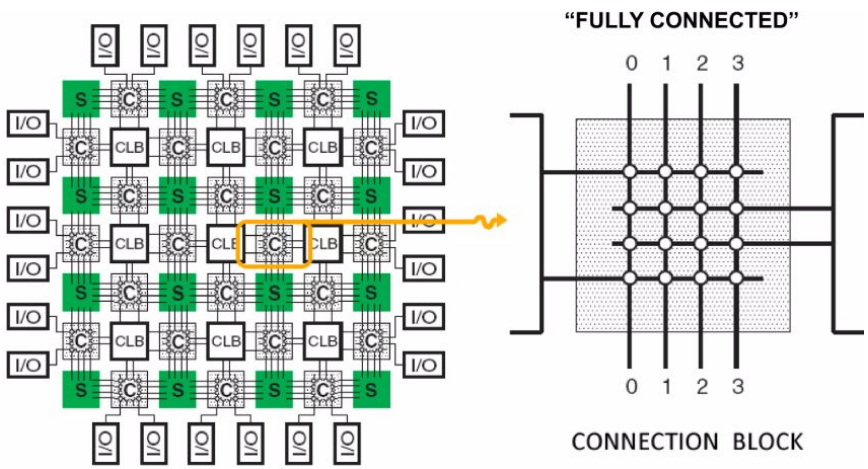
- ASIC-based implementations have been recently explored to accelerate the DNN inference.
  - Google's TPU, Apple's Neural Engine, Cerebras AI chip, …
- FPGA-based accelerators for DNN inference have been recently developed.
  - Has good programmability and flexibility
  - Short development cycles
  - Can be used as a benchmark before implementing on ASIC



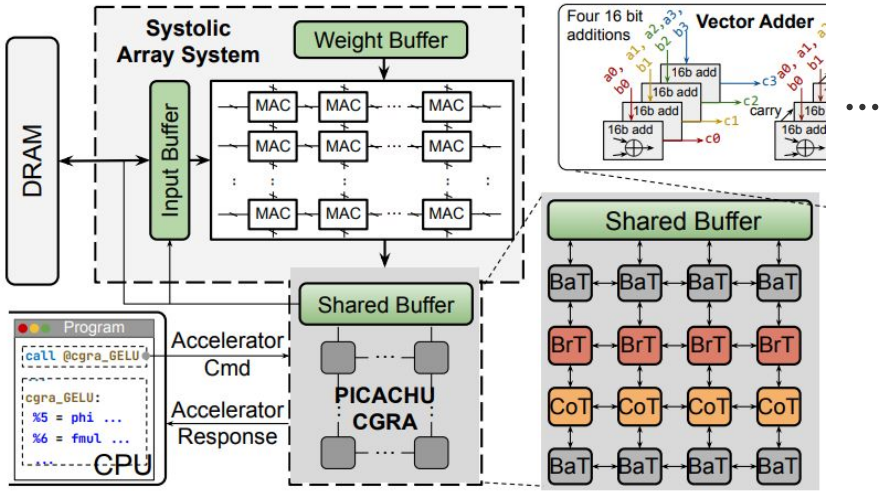
Tensor Processing Unit (Google)

Alveo Accelerator Card (Xilinx)

FPGA and CGRA

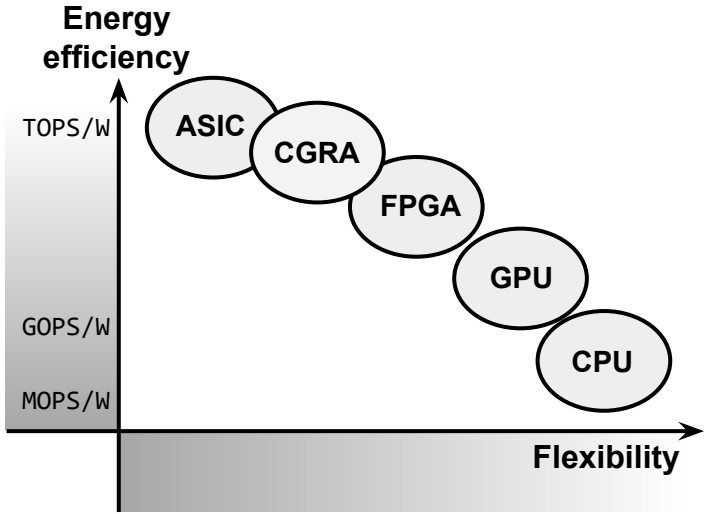


FPGA



CGRA

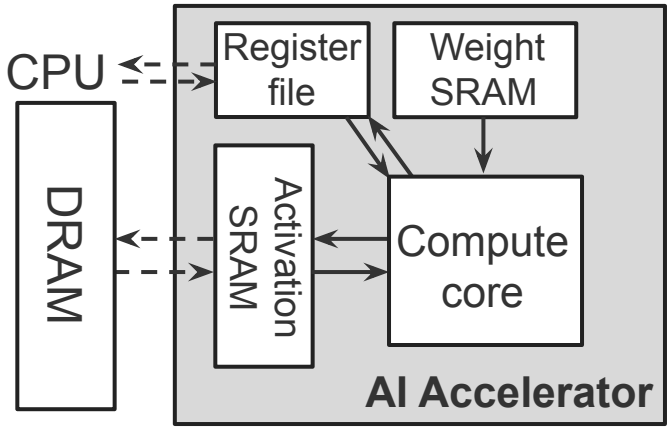
Flexibility & Performance

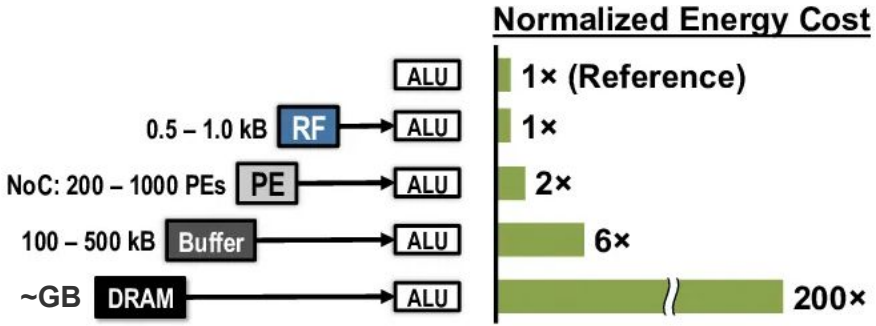


- ASIC offers the highest energy efficiency but is only suitable for specific applications.
- The CPU is a general-purpose processor but has the lowest energy efficiency.

AI Accelerator

- Making any chip is a costly, difficult and lengthy process typically done by teams of 10 to 1000's of people depending on the size and complexity of the chip.

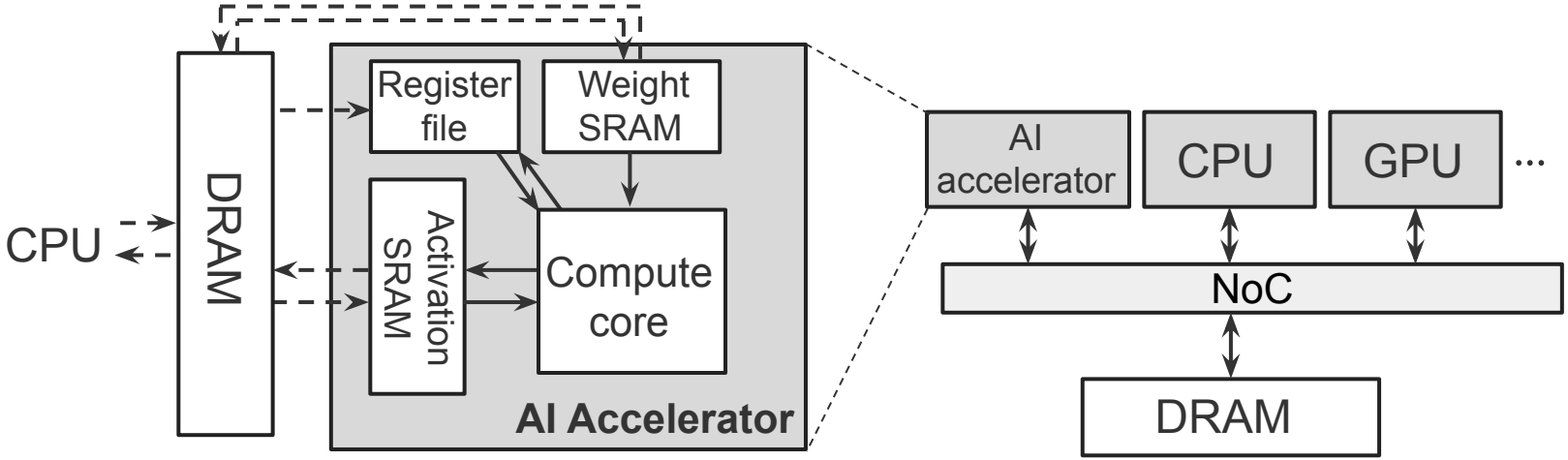




NYU SAI LAB

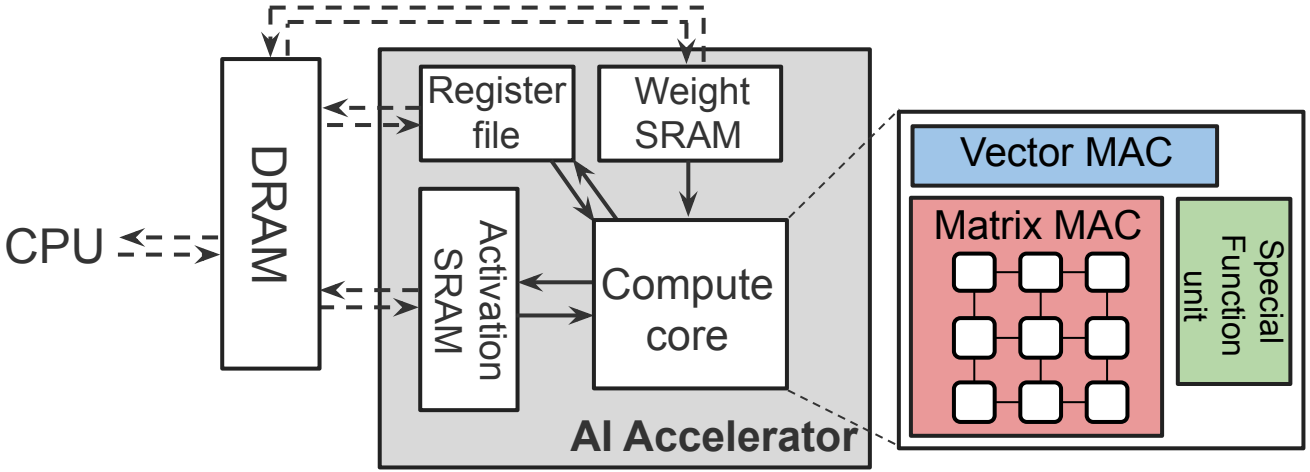
AI Accelerator

- The AI accelerator can execute part of the machine code that is related to the AI workload.



NYU SAI LAB

AI Accelerator



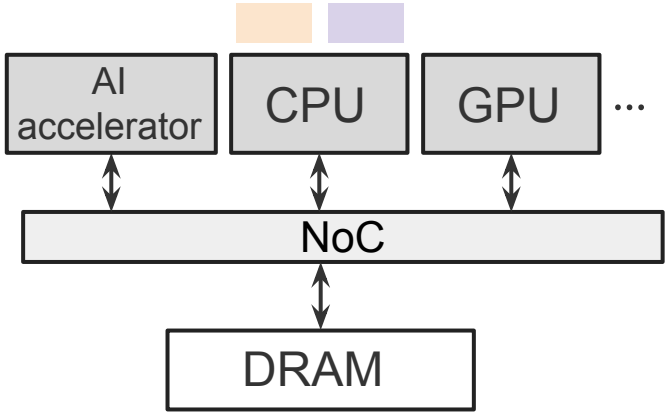
- The compute core consists of Multiply and accumulator (MAC) engine for 2D matrix multiplication.
- It also contains vector multiplier MAC as well as special function unit.

AI Accelerator

```
import torch
import torch.nn as nn
input = torch.randn(1, 1, 5, 5)
conv = nn.Conv2d(1, 1, kernel_size=3)
output = conv(input)
print(output)
```

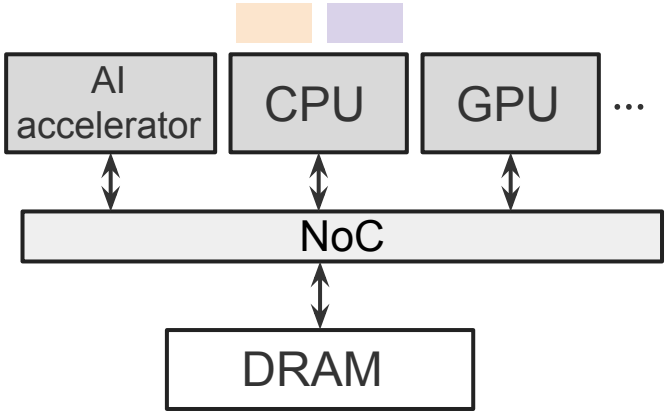
Compiler

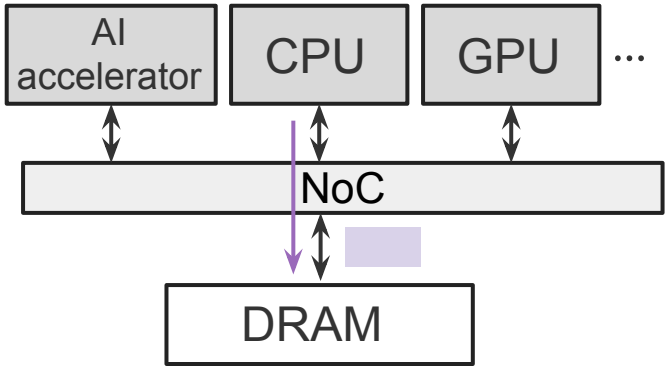
```
01001000 10001001 11011000
01001000 10000011 11000000 00001010
01001000 10000011 11101011 00000101
01001000 00111001 11011000
01110100 00000101
01001000 10001001 11000001
01001000 01101001 11001001 00000010
01001000 10001001 00001111
11101011 00000011
```



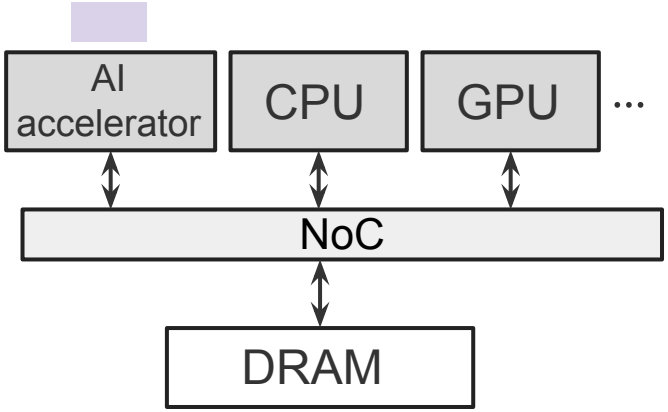
NYU SAI LAB

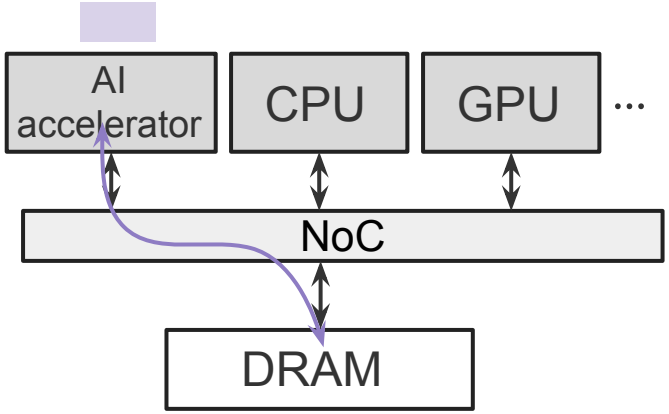
AI Accelerator



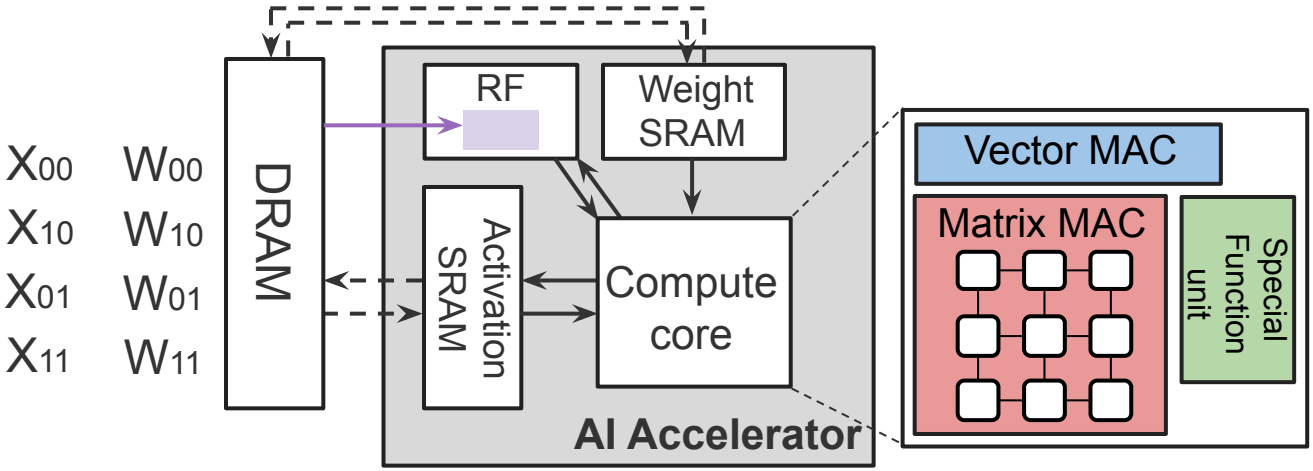


AI Accelerator



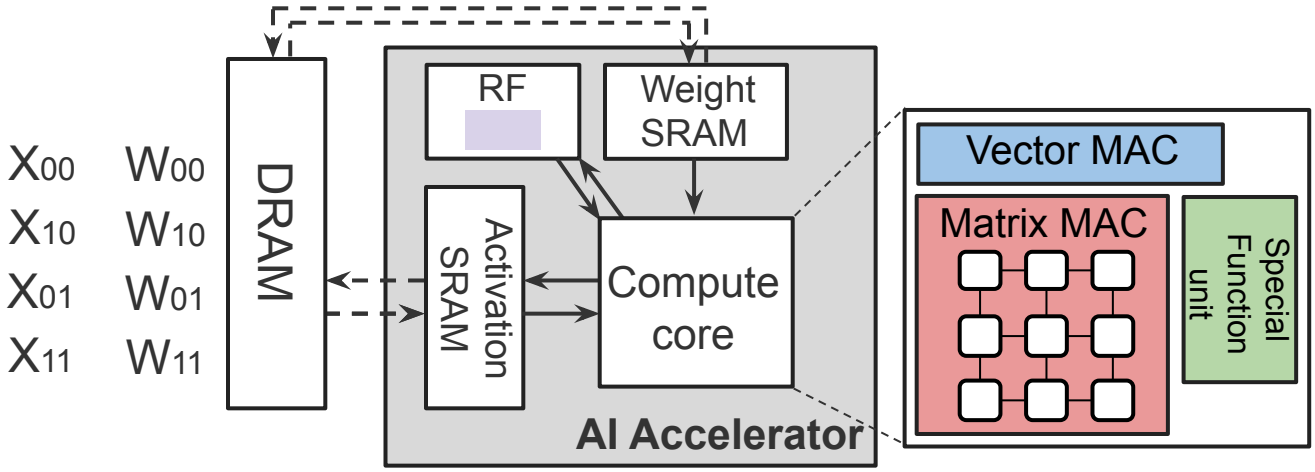


AI Accelerator



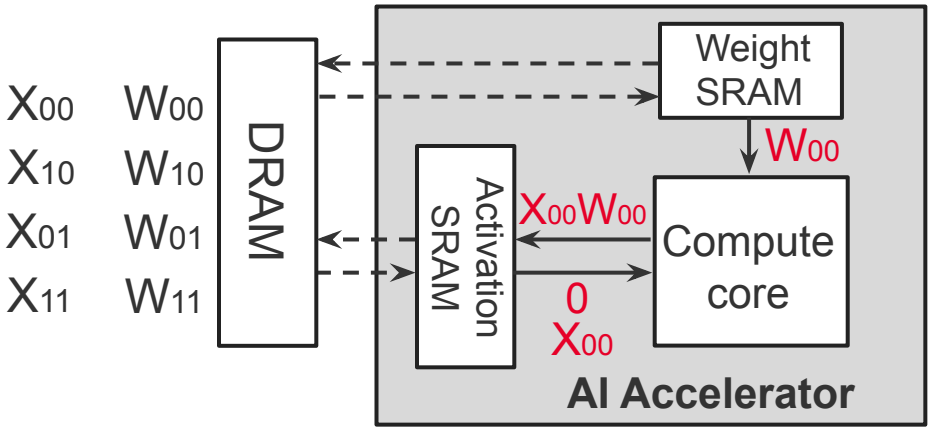
$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

AI Accelerator



$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

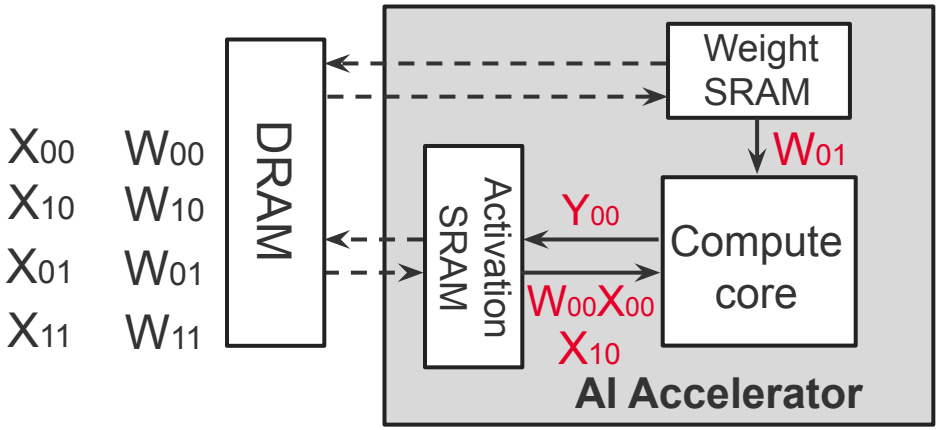
AI Accelerator



$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

NYU SAI LAB

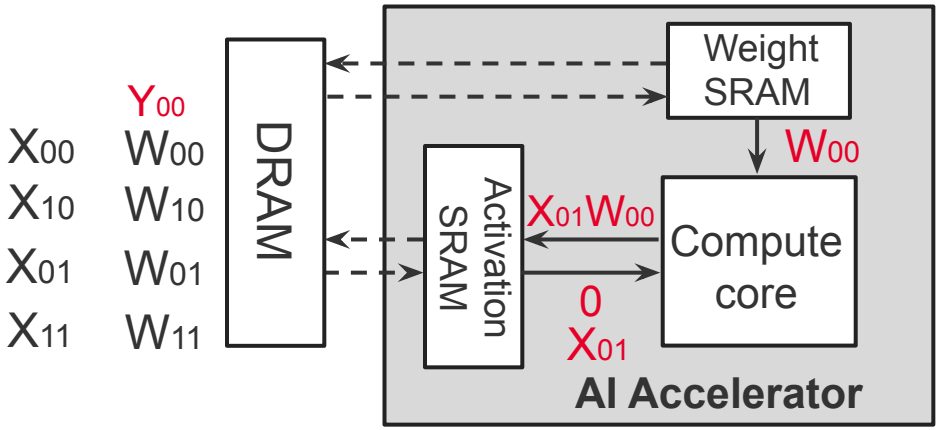
AI Accelerator



$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

NYU SAI LAB

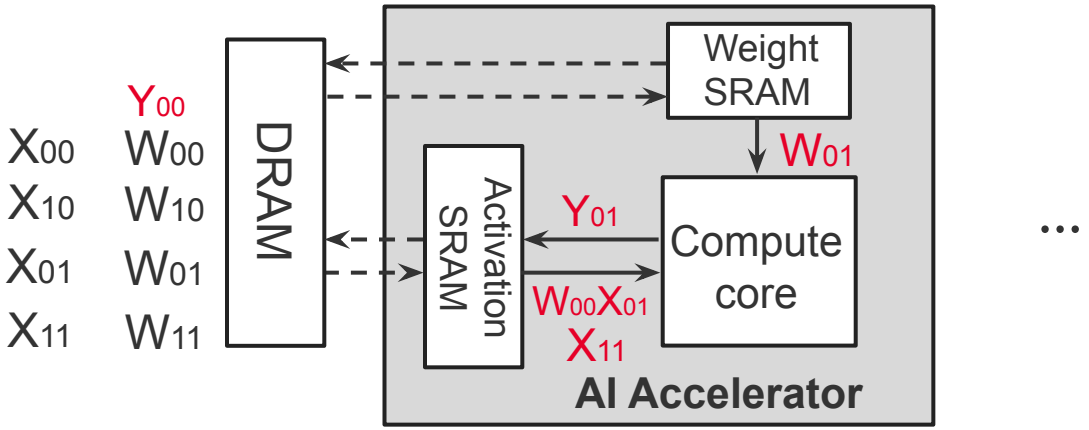
AI Accelerator



$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

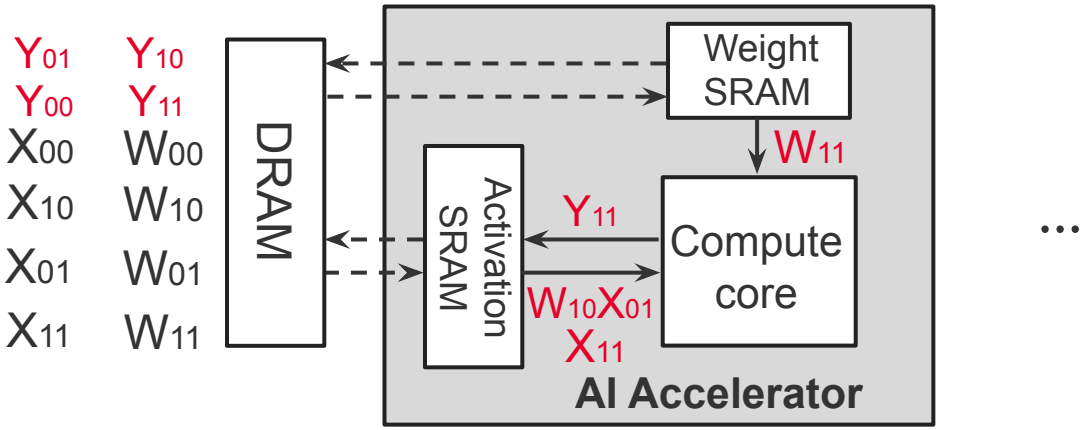
NYU SAI LAB

AI Accelerator



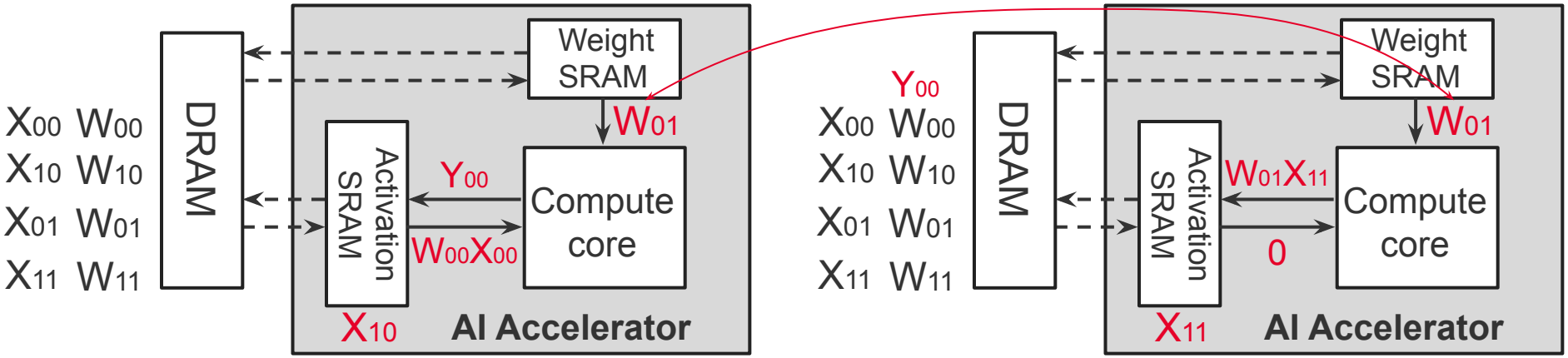
$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

AI Accelerator



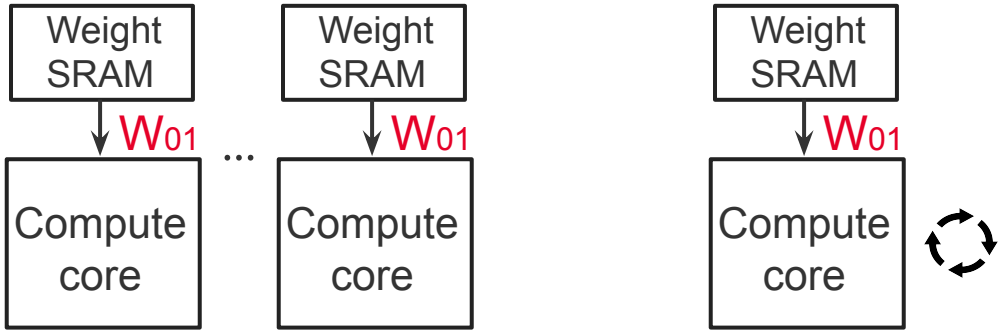
$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

Memory Access Reduction



- The computation and memory access pattern can be changed to minimize the computational cost without impacting the results.

Memory Access Reduction

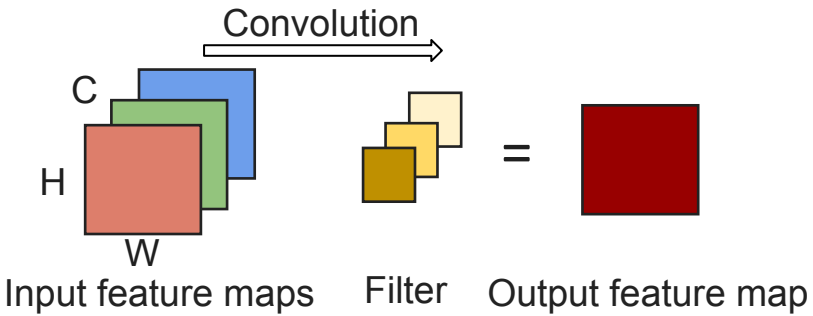


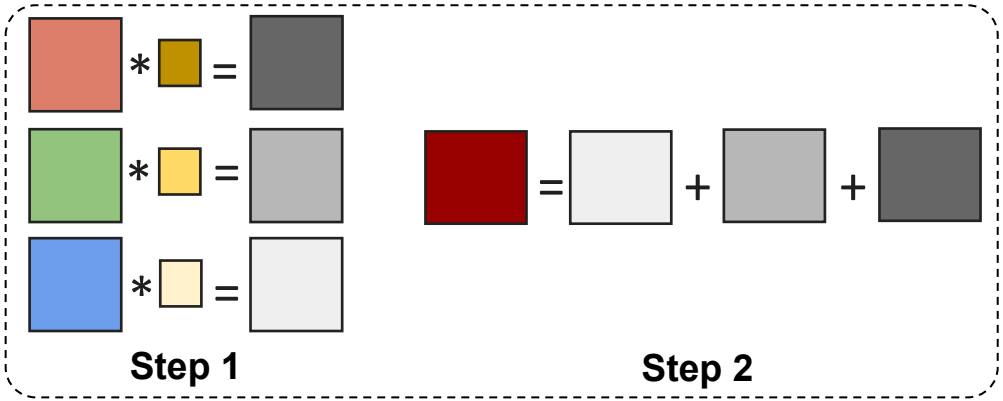
- It is preferable to minimize memory access by maximizing the reuse of loaded data.
- We will explore methods for scheduling neural network layer accesses to minimize memory usage in the next lecture.

Topics

- Machine Learning Compiler
- Introduction to AI Accelerator
- CNN Accelerator

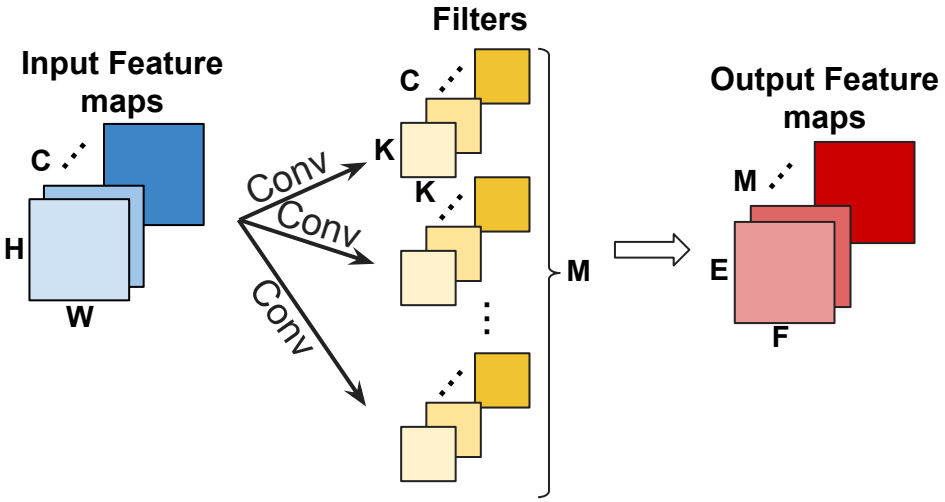
Convolutional Layers





- Core building block of a CNN, it is also the most computational intensive layer.

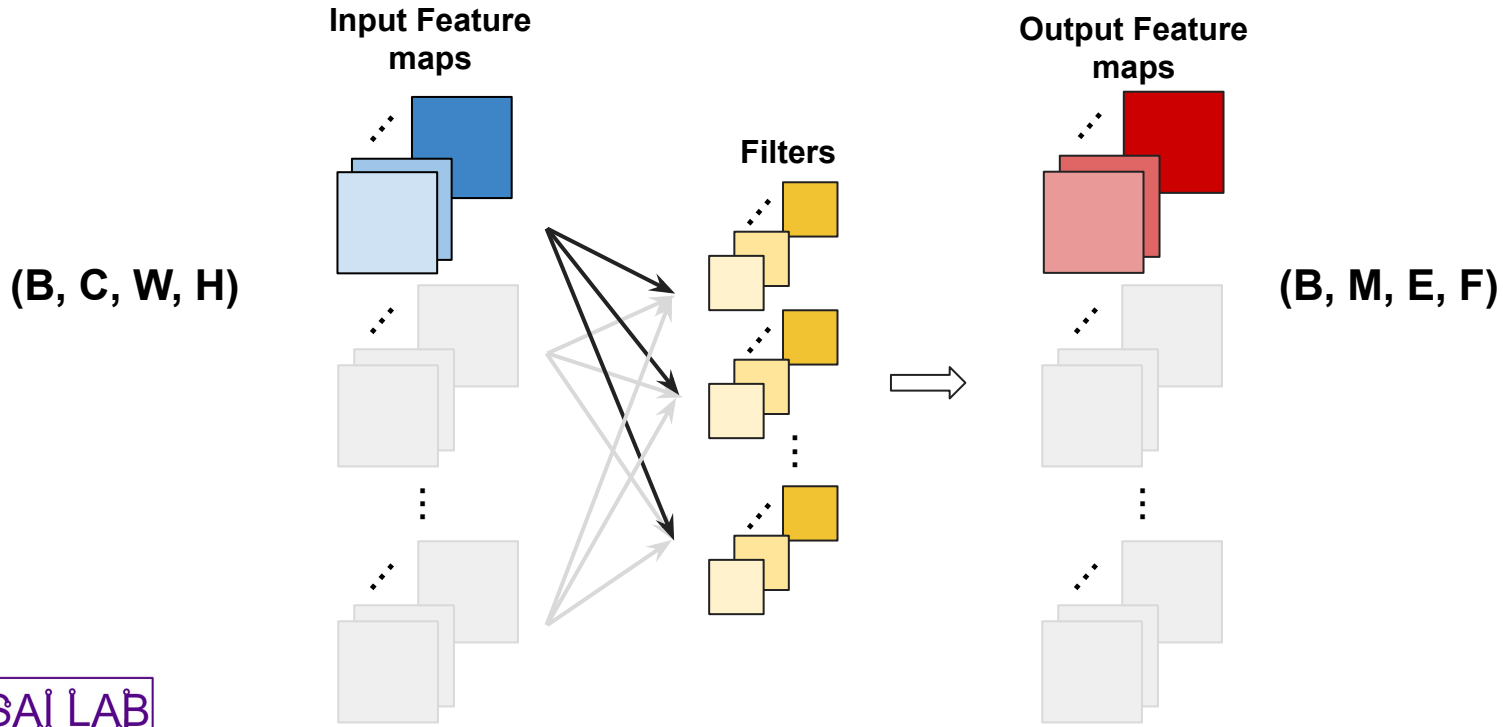
Convolution



- Number of MACs: M×K×K×C×E×F
- Storage cost: 32×(M×C×K×K+C×H×W+M×E×F)

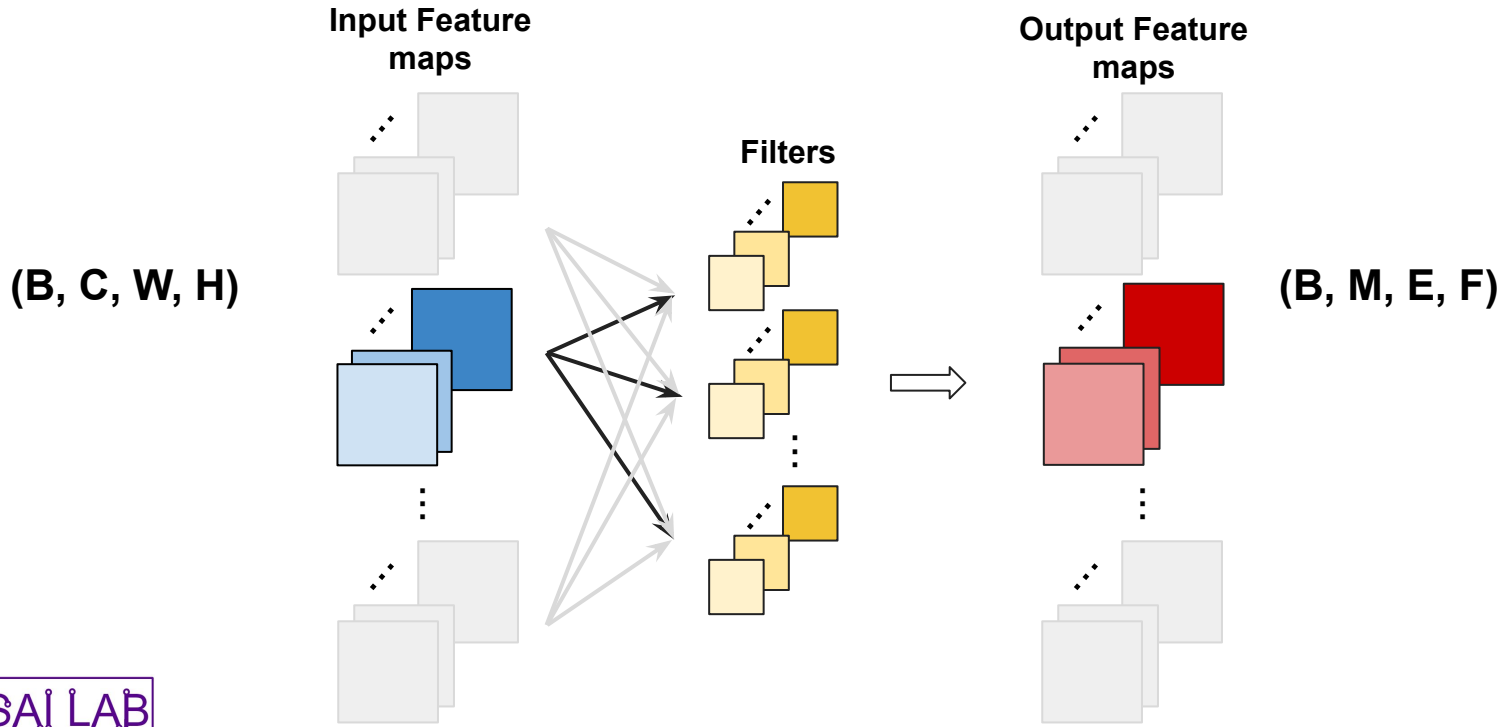
C: number of input channels
H,W: size of the input feature maps
M: number of weight filters
K: weight kernel size
E,F: size of the output feature maps

Convolution



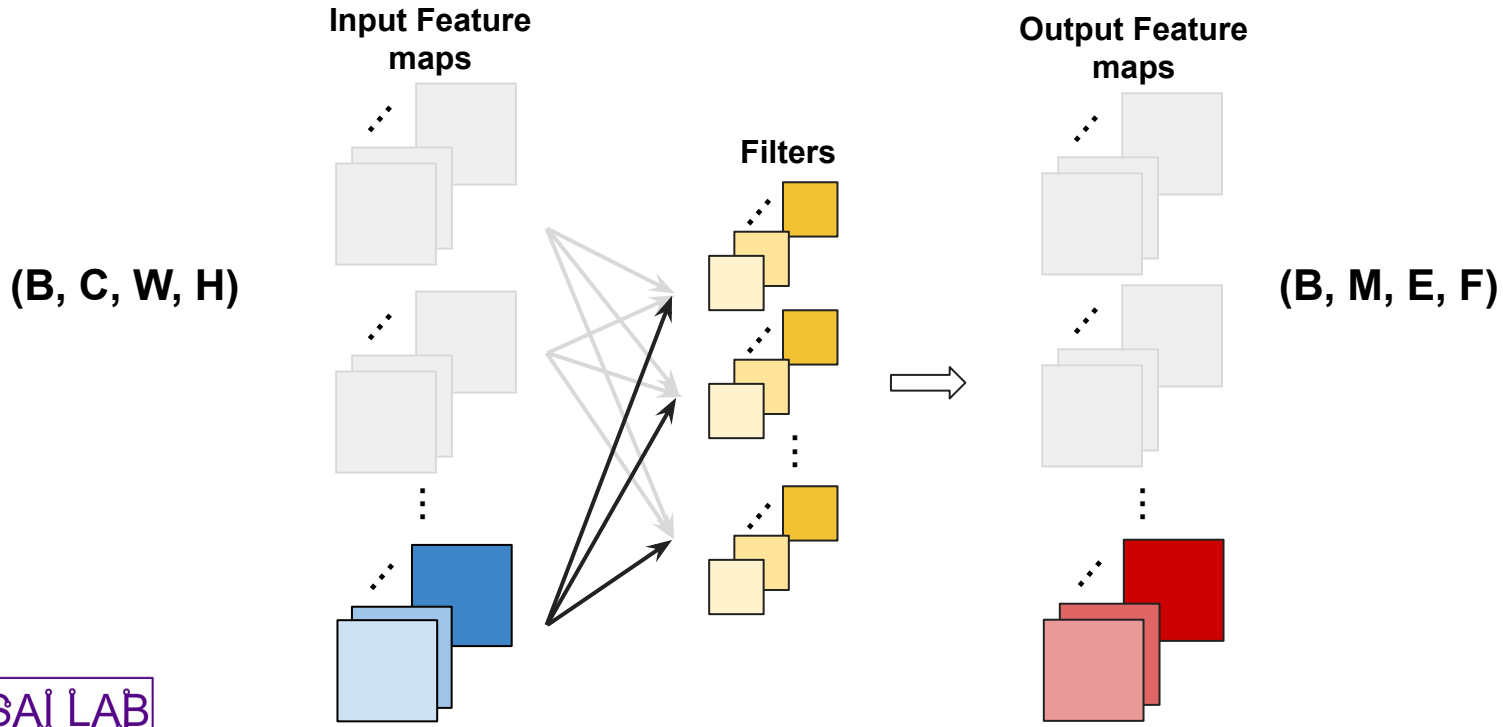
NYU SAI LAB

Convolution

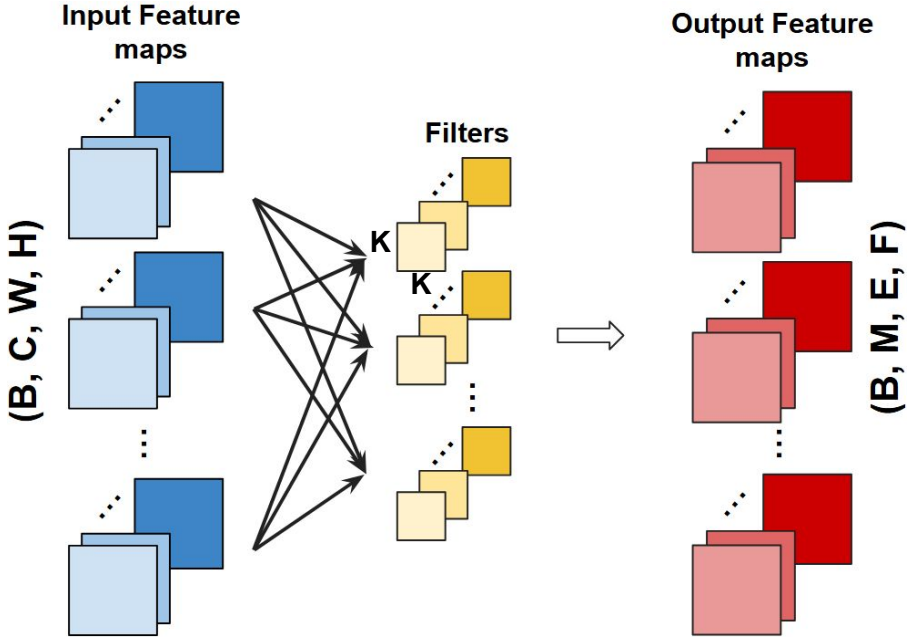


NYU SAI LAB

Convolution



Computational Cost: Standard Convolution



- Number of MACs: B×M×K×K×C×E×F
- Storage cost: 32×(M×C×K×K+B×C×H×W+B×M×E×F)

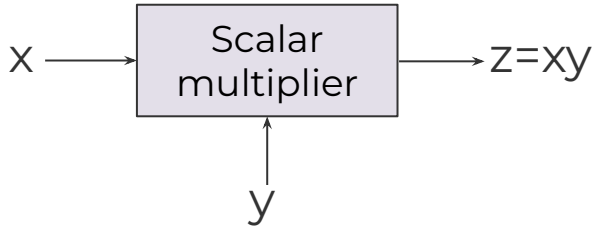
  B: batch size
  C: number of input channels
  H,W: size of the input feature maps
  M: number of weight filters
  K: weight kernel size
  E,F: size of the output feature maps

- We need to iterate over seven dimensions:
  - B, M, C, E, F, K(kernel width), K (kernel height)

NYU SAI LAB

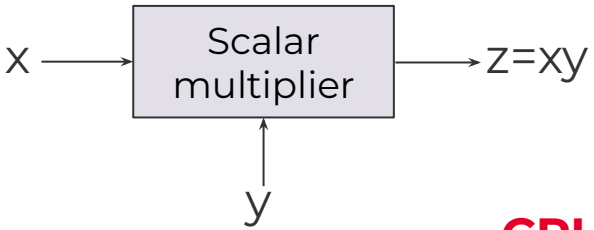
Computational Dataflow for CNN

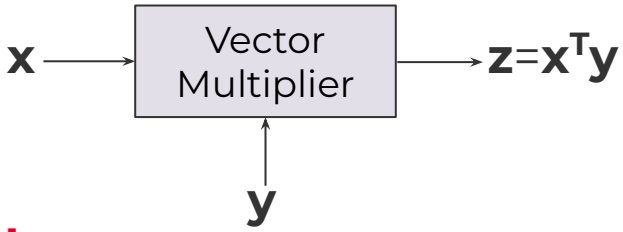
```
for b = 1 to B
   for m = 1 to M
      for c = 1 to C
         for w = 1 to E
            for h = 1 to F
               for k1 = 1 to K
                  for k2 = 1 to K
                     out[b][m][e][f] += in[b][c][e+k1-(K+1)/2][f+k2-(K+1)/2] * filter[m][c][k1][k2];
```



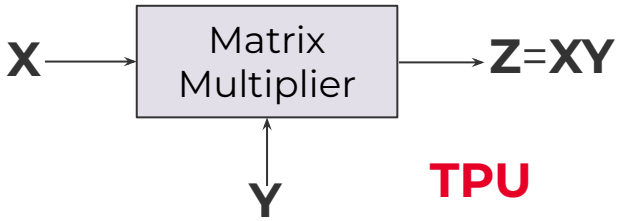
- This simple loop nest can be transformed in numerous ways to capture different reuse patterns of the activations and weights and to map the computation to a hardware accelerator implementation.
- A CNN's dataflow defines how the loops are ordered, partitioned, and parallelized
- We can use the scaler machine to compute the results of CNN using this for loop

Computational Dataflow for CNN

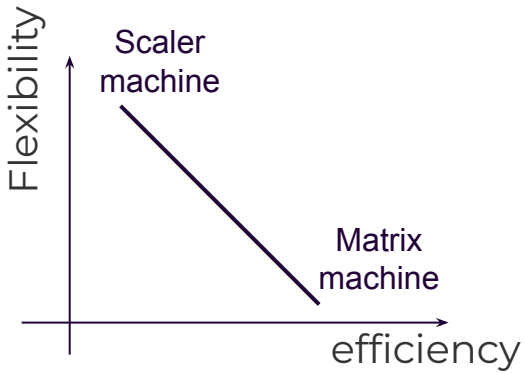




CPU & GPU

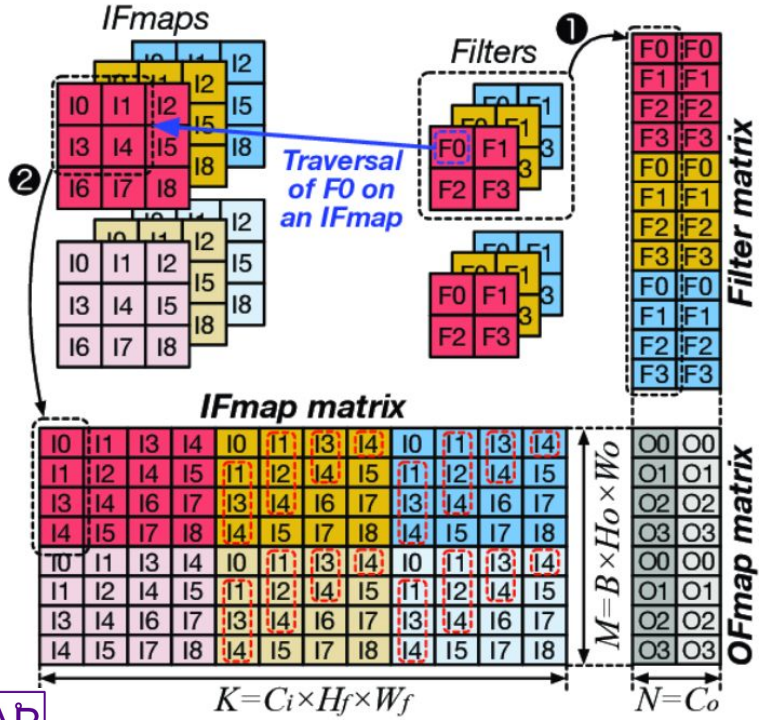


TPU



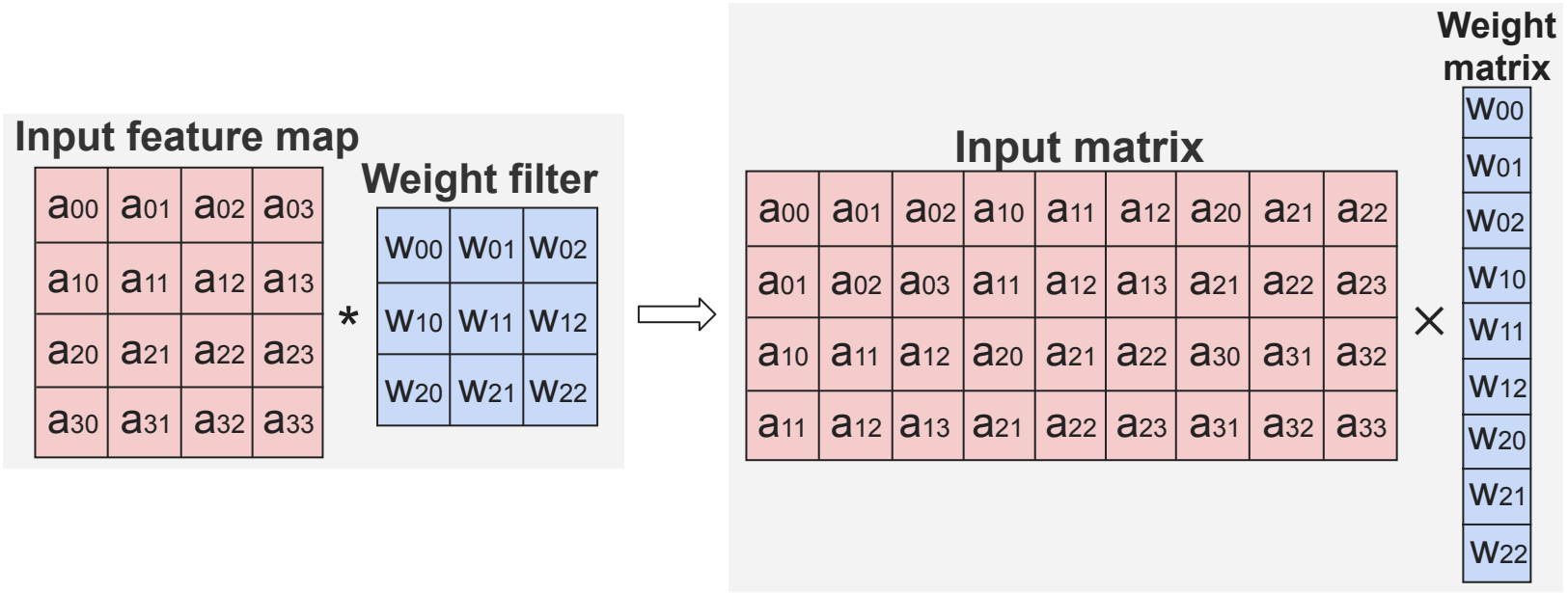
NYU SAI LAB

How to Convert to Matrix Multiplication?

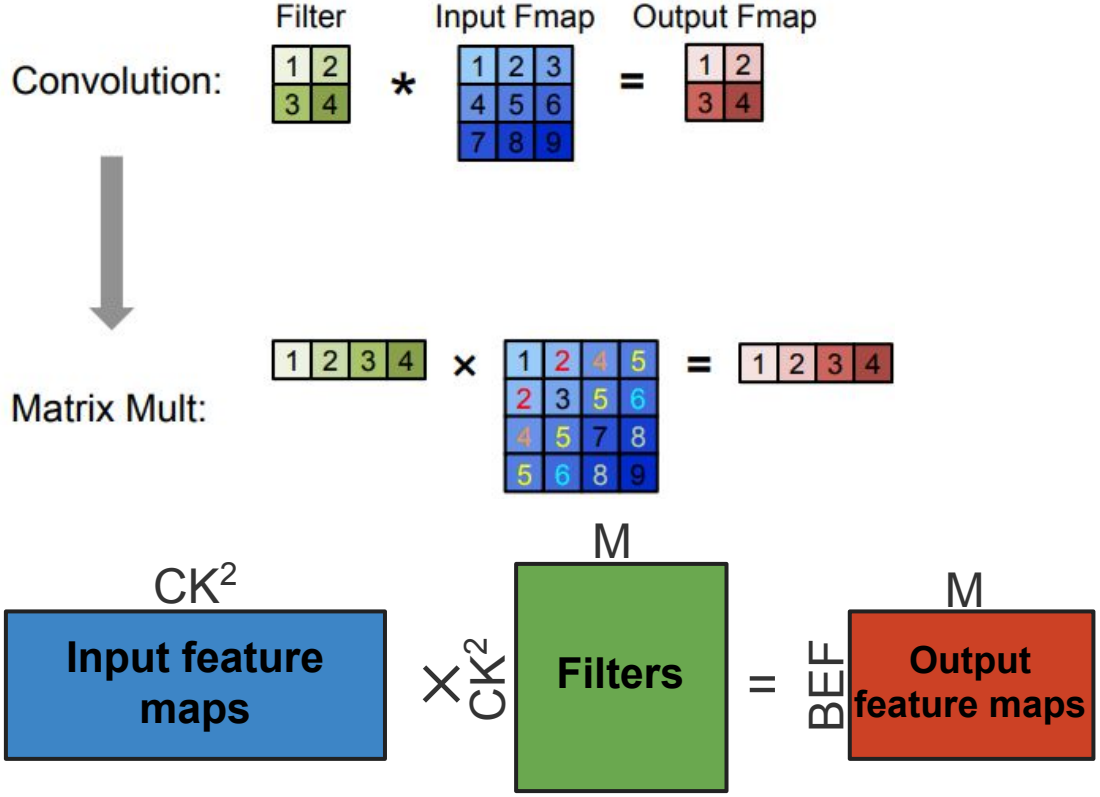


- A standard Convolutional operation can be converted to 2D matrix multiplication using **Im2Col** operations.

How to Convert to Matrix Multiplication?



How to Convert to Matrix Multiplication?



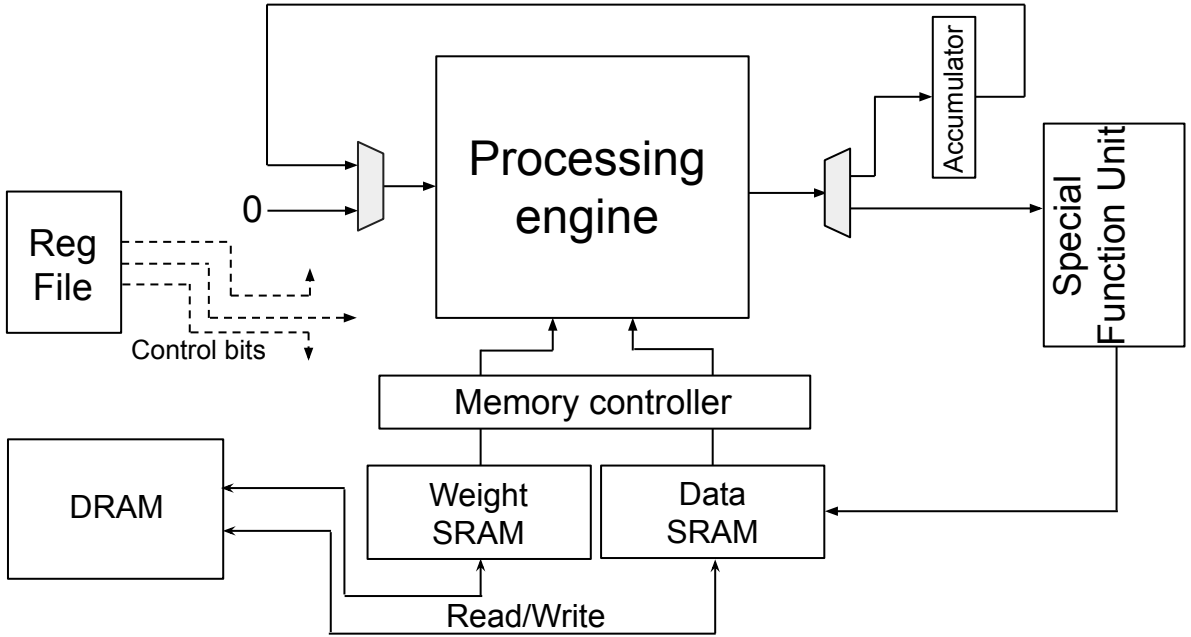
Tiling

- In order to handle matrix multiplication with large size, it is usually decomposed into tiles.

$$\begin{bmatrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{bmatrix} \times \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix} = \begin{bmatrix} W_{00}X_{00}+W_{01}X_{10} & W_{00}X_{01}+W_{01}X_{11} \\ W_{10}X_{00}+W_{11}X_{10} & W_{10}X_{01}+W_{11}X_{11} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{bmatrix}$$

- Each of W_{ij} and X_{ij} can be a sub-matrix.

Hardware Architectures for DNN Processing

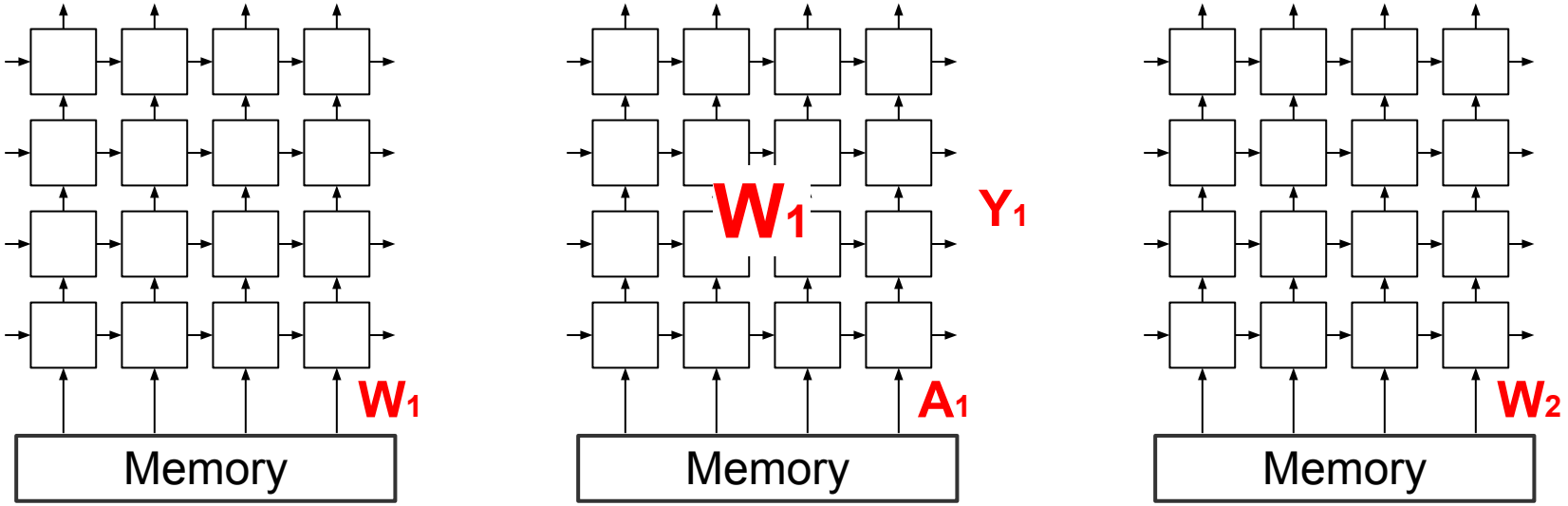


Major building blocks:

- Processing engine
- Accumulator
- Reg file
- Special function unit
- Memory subsystem
  - Weight SRAM
  - Data SRAM
  - DRAM

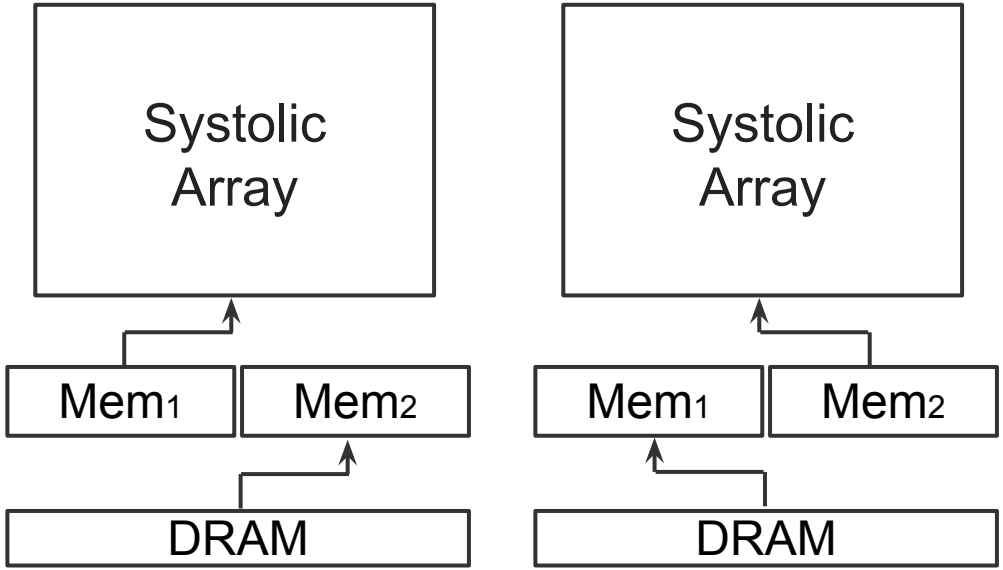
NYU SAI LAB

Computing Paradigms



- Spatial architecture can achieve great reuse of the extracted content, leading to a reduced memory access cost.

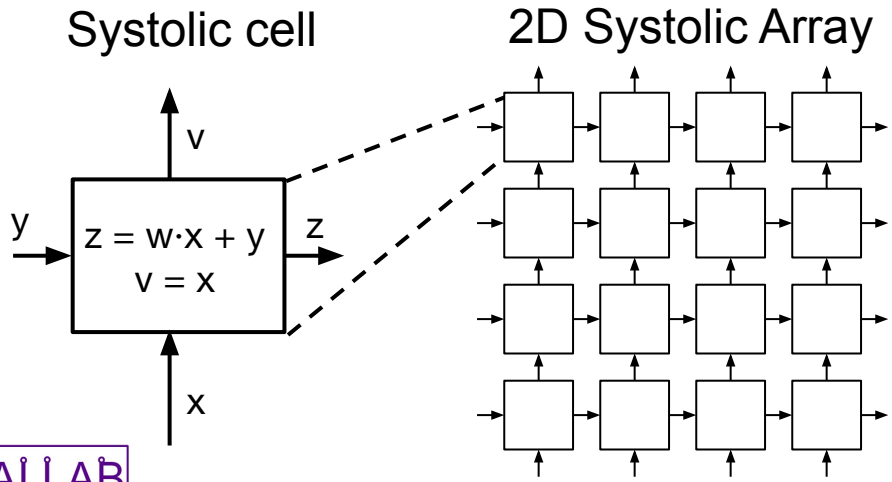
Double Buffering



- Double buffering in hardware design is a technique used to improve the efficiency and performance of data processing, especially in systems that require smooth and continuous data transfer.
- The idea is to overlap the data production and consumption processes to avoid delays.

Systolic Array (Weight Stationary Version)

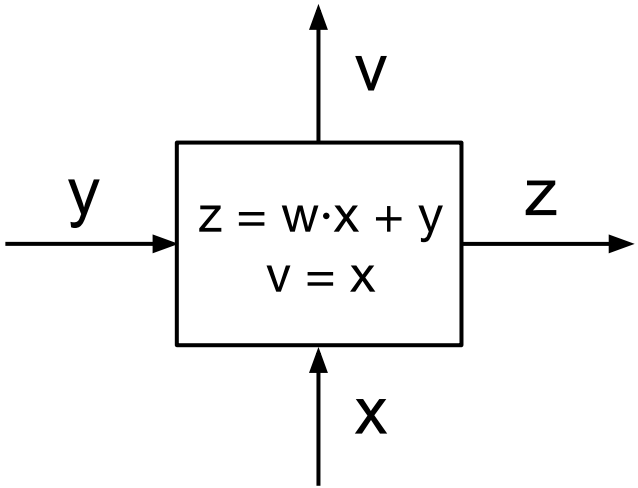
- Kung and Leiserson, "Systolic Arrays for VLSI," 1978 and Kung, "Why systolic architectures?' 1982
- 2D grid of multiplier-accumulators (MACs) for matrix multiplication
- Used by Google TPU for deep learning (2017), etc



TPU (Google)

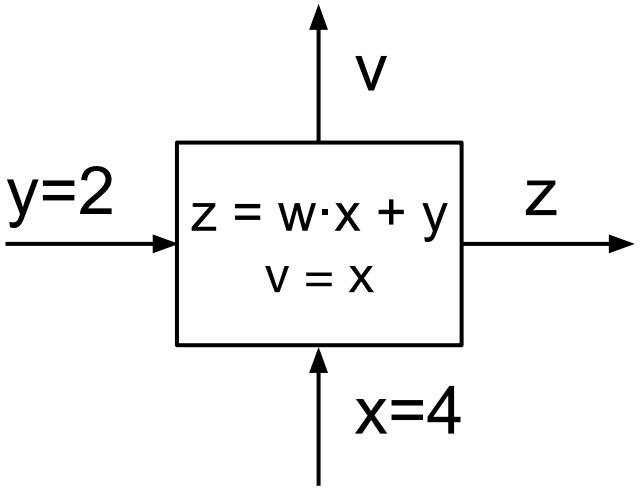
NYU SAI LAB

Systolic Cell



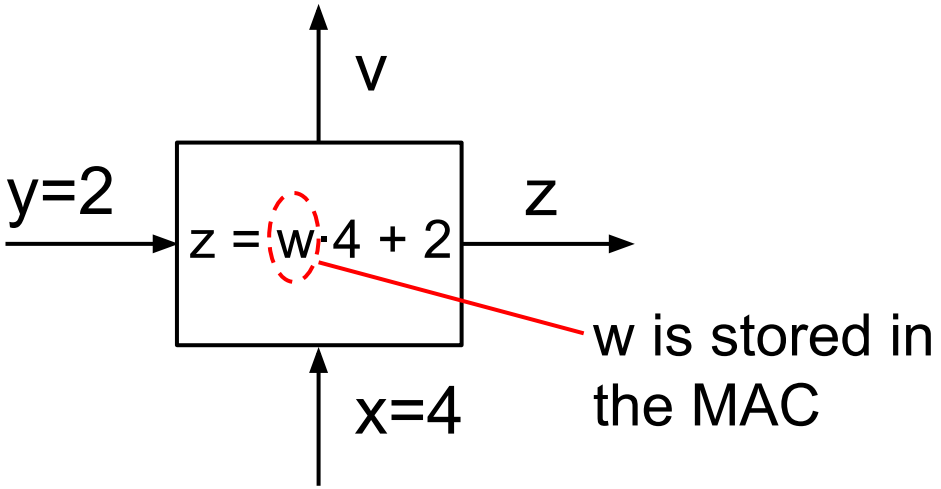
- Takes data (x and y) as input
- w stays in the systolic cell
- Performs a multiply-accumulate operation

Systolic Cell

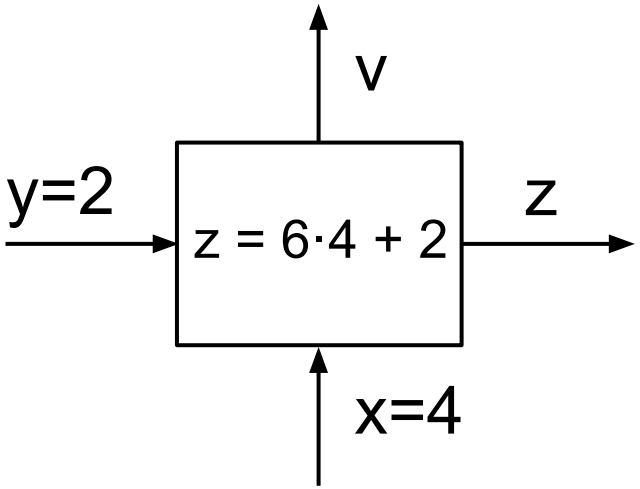


NYU SAI LAB

Systolic Cell

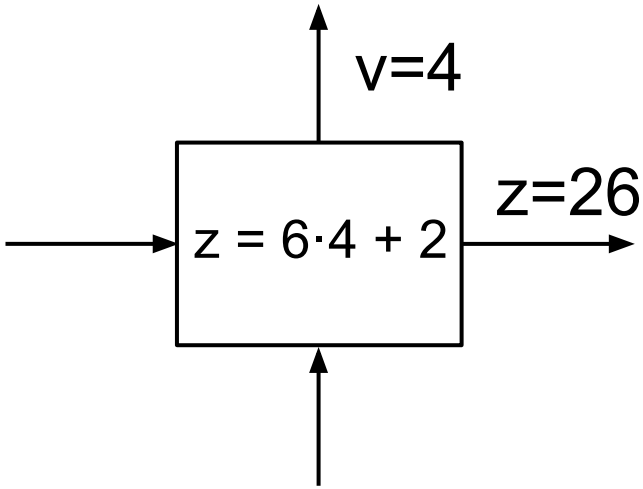


Systolic Cell



NYU SAI LAB

Systolic Cell

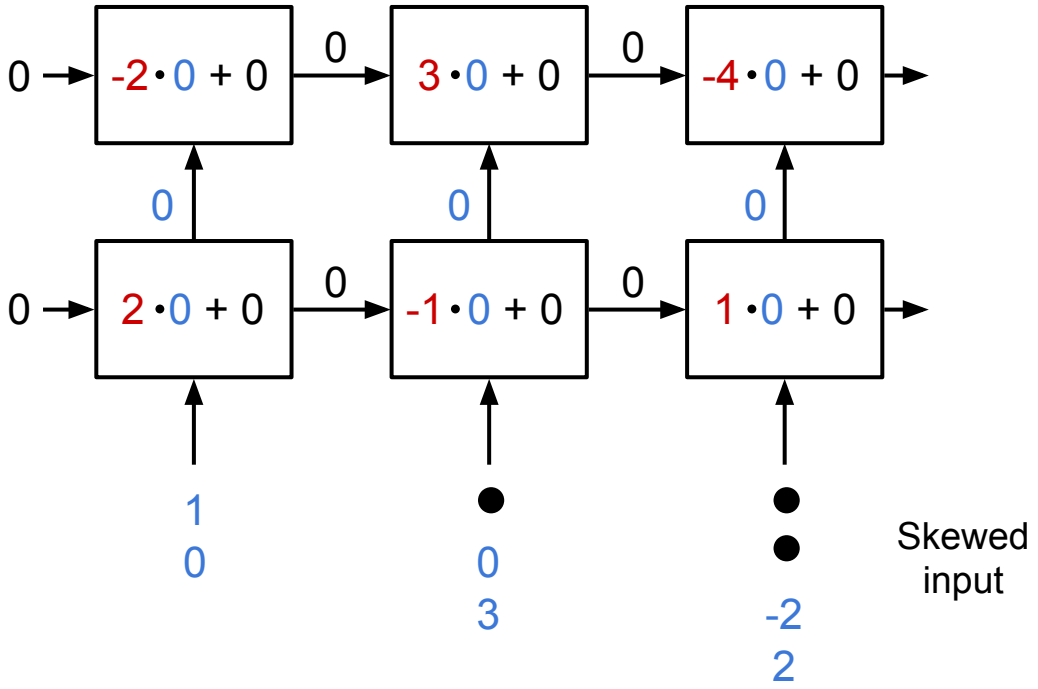


Visualizing Systolic Array Multiplication

Weight Matrix × Data Matrix = Result Matrix

$$\begin{bmatrix} 2 & -1 & 1 \\ -2 & 3 & -4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 6 & 1 \end{bmatrix}$$

Weights in red are preloaded into the systolic array

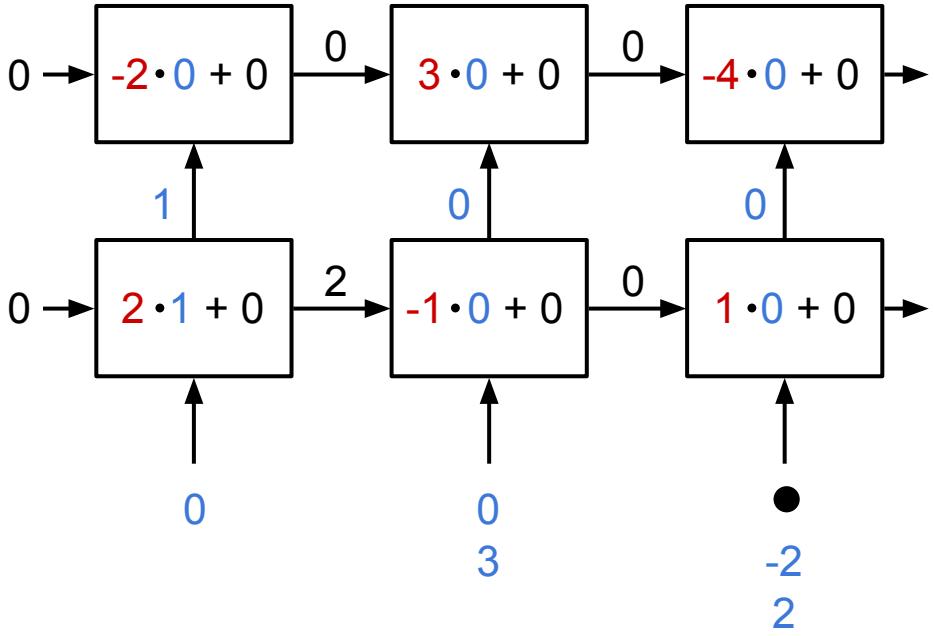


Visualizing Systolic Array Multiplication

Weight Matrix × Data Matrix = Result Matrix

$$\begin{bmatrix} 2 & -1 & 1 \\ -2 & 3 & -4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 6 & 1 \end{bmatrix}$$

Weights in red are preloaded into the systolic array

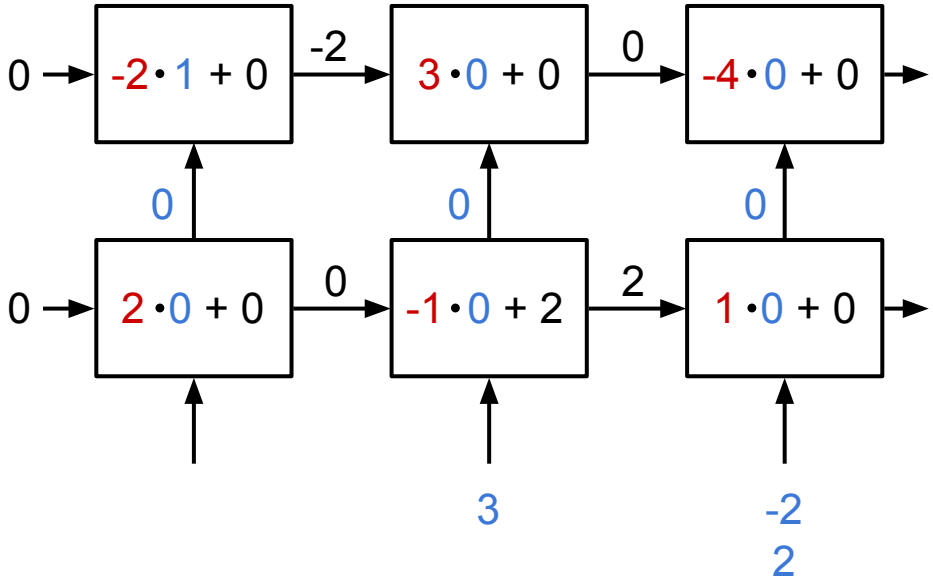


NYU SAI LAB

Visualizing Systolic Array Multiplication

Weight Matrix × Data Matrix = Result Matrix

$$\begin{bmatrix} 2 & -1 & 1 \\ -2 & 3 & -4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 6 & 1 \end{bmatrix}$$



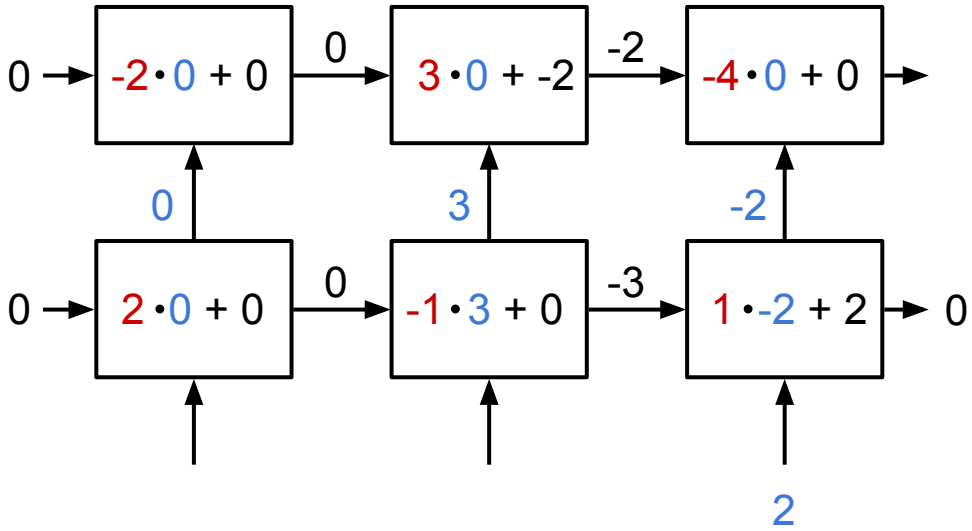
Weights in red are preloaded into the systolic array

NYU SAI LAB

Visualizing Systolic Array Multiplication

Weight Matrix × Data Matrix = Result Matrix

$$\begin{bmatrix} 2 & -1 & 1 \\ -2 & 3 & -4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 6 & 1 \end{bmatrix}$$



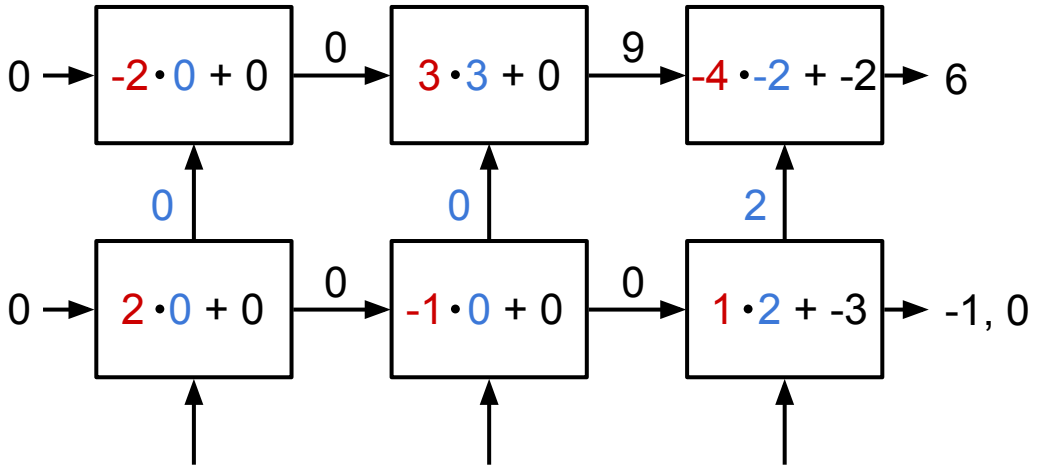
Weights in red are preloaded into the systolic array

NYU SAI LAB

Visualizing Systolic Array Multiplication

Weight Matrix × Data Matrix = Result Matrix

$$\begin{bmatrix} 2 & -1 & 1 \\ -2 & 3 & -4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 6 & 1 \end{bmatrix}$$



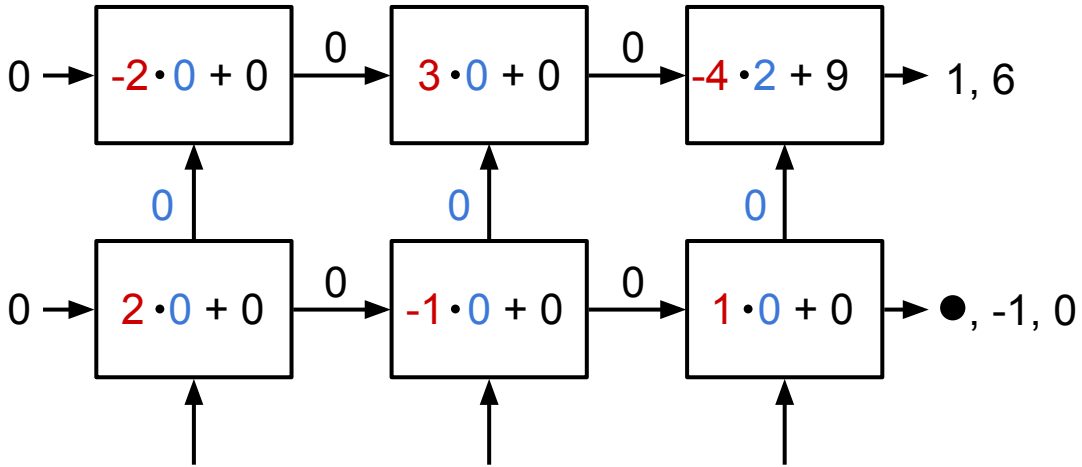
Weights in red are preloaded into the systolic array

NYU SAI LAB

Visualizing Systolic Array Multiplication

Weight Matrix × Data Matrix = Result Matrix

$$\begin{bmatrix} 2 & -1 & 1 \\ -2 & 3 & -4 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 3 \\ -2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 6 & 1 \end{bmatrix}$$



Weights in red are preloaded into the systolic array

NYU SAI LAB

Presentation

- Compressing Optimizer States and Activation for Memory-efficient FP8 Training
  - Pragnay Mandavilli, Raju Kanumuri, Ayush Yadav
- Federated Optimization in Heterogeneous Networks
  - Andrew Hennessee, Buddhi Perera, and Johnson Yang