EE-508: Hardware Foundations for Machine Learning
Modeling Accelerators

University of Southern California

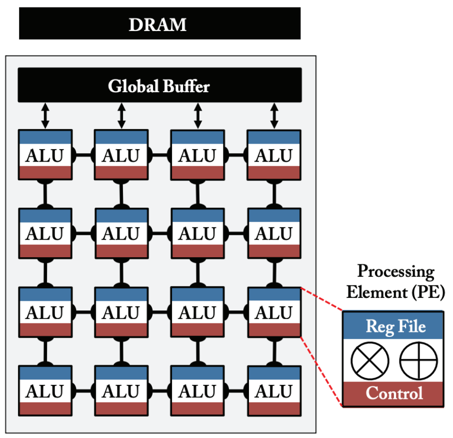
Ming Hsieh Department of Electrical and Computer Engineering

Instructor:
Arash Saifhashemi

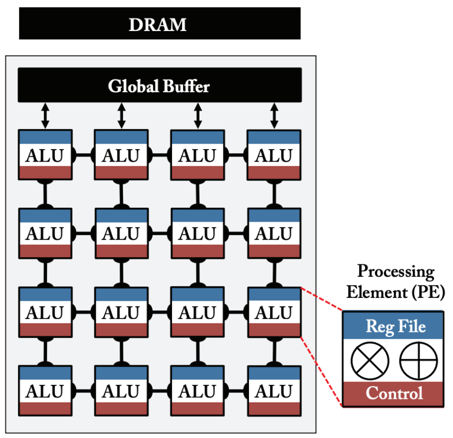
Modeling Accelerators

Modeling

- High Level Languages
  - Python, C/C++
- Domain-Specific Languages (DSLs).
  - Examples:
    - Halide:
      - Allows separation of the algorithm from its schedule, enabling performance portability across different hardware architectures.
    - TVM:
      - Open-source compiler framework that abstracts the details of hardware accelerators.
- Hardware Description Languages (HDLs):
  - SystemC, SystemVerilog, VHDL
  - HLS (High-Level Synthesis) Languages



Modeling



- High Level Languages
  - Python, C/C++
- Domain-Specific Languages (DSLs).
  - Examples:
    - Halide:
      - Allows separation of the algorithm from its schedule, enabling performance portability across different hardware architectures.
    - TVM:
      - Open-source compiler framework that abstracts the details of hardware accelerators.
- Hardware Description Languages (HDLs):
  - SystemC, SystemVerilog, VHDL
  - HLS (High-Level Synthesis) Languages

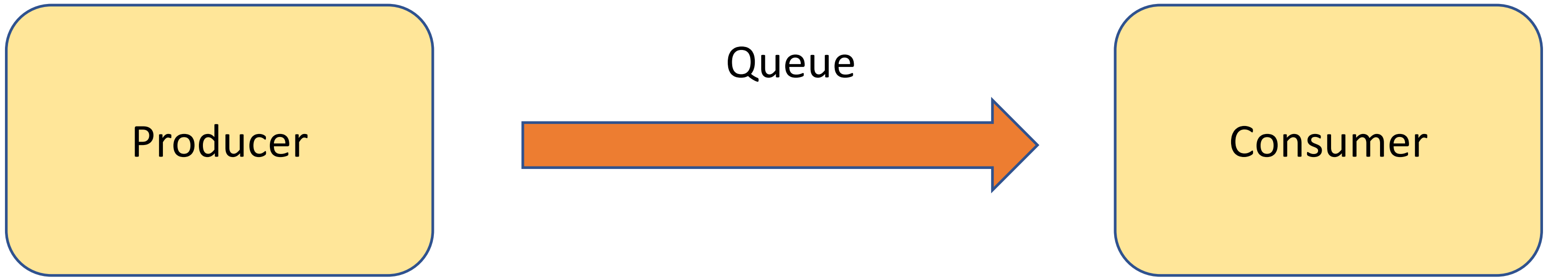
It's more common to use HDLs for hardware design, but we use Python for high level modeling for simplicity.

Python Multiprocessing: Using Process and Queue

- Process:
  - A class for spawning processes in Python, similar to threading.
- Queue:
  - A safe way to pass messages between processes.
    - The maximum size depends on the OS.

```python
def producer(queue):
    for i in range(5):
        item = f'Item {i}'
        queue.put(item)
        print(f'Produced {item}')
        time.sleep(1)
```

```python
def consumer(queue):
    while True:
        item = queue.get()
        if item is None:
            break
        print(f'Consumed {item}')
        time.sleep(1.5)
```

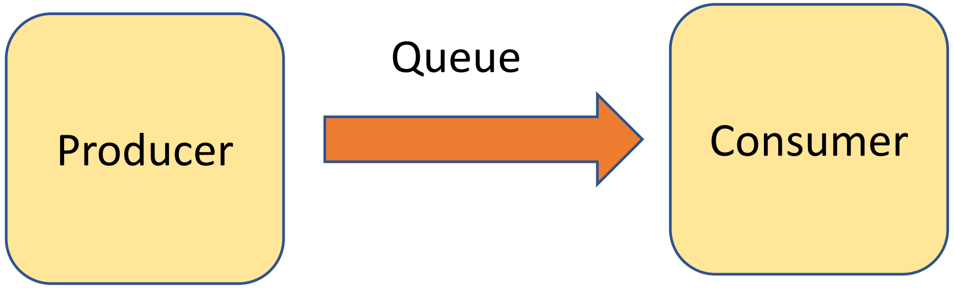


The Put Function

- `put(item[, block[, timeout]]):`
  - Used to enqueue (or add) an item to the queue.
  - Optionally, you can specify whether to **block** if the queue is full using the block parameter (**default is True**).
    - If block is set to True and the queue is full, the method will wait until there's space available in the queue.
  - `timeout` parameter specifies the maximum amount of time (in seconds) to wait if blocking is enabled.
    - If the timeout is reached and the queue is still full, a `Queue.Full` exception will be raised.

```python
def producer(queue):
    for i in range(5):
        item = f'Item {i}'
        queue.put(item)
        print(f'Produced {item}')
        time.sleep(1)
```

```python
def consumer(queue):
    while True:
        item = queue.get()
        if item is None:
            break
        print(f'Consumed {item}')
        time.sleep(1.5)
```

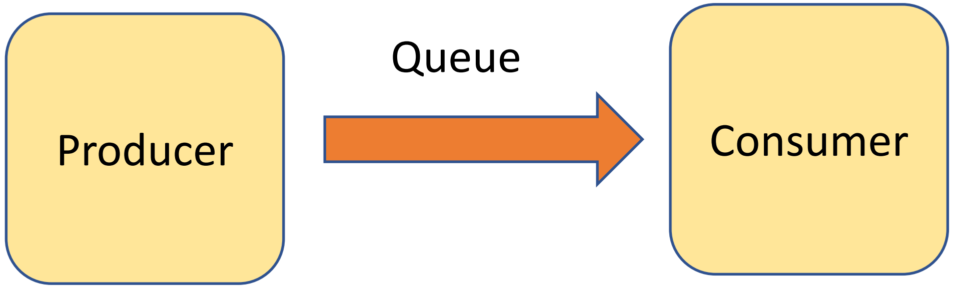


The Get Function

- `get([block[, timeout]]):`
  - Used to dequeue (or retrieve) an item from the queue.
  - Optionally, you can specify whether to **block** if the queue is empty using the block parameter (**default is True**).
    - If block is set to True and the queue is empty, the method will wait until there's an item available in the queue.
  - `timeout` parameter specifies the maximum amount of time (in seconds) to wait if blocking is enabled.
    - If the timeout is reached and the queue is still empty, a `Queue.Empty` exception will be raised.

```
def producer(queue):
    for i in range(5):
        item = f'Item {i}'
        queue.put(item)
        print(f'Produced {item}')
        time.sleep(1)
```

```
def consumer(queue):
  while True:
    item = queue.get()
    if item is None:
        break
    print(f'Consumed {item}')
    time.sleep(1.5)
```



No Wait Version of Put and Get

- `put_nowait(item)`:
  - Similar to put(), but it does not block.
  - It attempts to enqueue the item into the queue immediately.
    - If the queue is full, it raises a queue.Full exception immediately rather than waiting for space to become available.
- `get_nowait()`:
  - Similar to get(), but it does not block.
  - It attempts to dequeue an item from the queue immediately.
    - If the queue is empty, it raises a queue.Empty exception immediately rather than waiting for an item to become available.

Python Multithreading: Using Thread and Queue

```python
def producer(queue):
    for i in range(5):
        item = f'Item {i}'
        queue.put(item)
        print(f'Produced {item}')
        time.sleep(1)
```

```python
def consumer(queue):
 while True:
    item = queue.get()
    if item is None:
        break
    print(f'Consumed {item}')
    time.sleep(1.5)
```

```python
if __name__ == '__main__':
    q = queue.Queue()
    p = threading.Thread(target=producer, args=(q,))
    c = threading.Thread(target=consumer, args=(q,))

    p.start()
    c.start()

    p.join()
    q.put(None) # Signal the consumer to terminate
    c.join()
```

Python Multiprocessing: Using Process and Queue

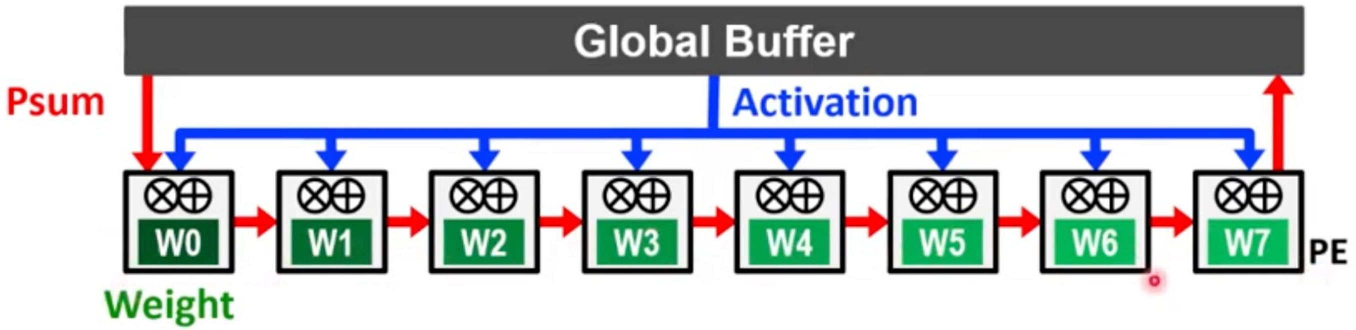
```python
def producer(queue):
    for i in range(5):
        item = f'Item {i}'
        queue.put(item)
        print(f'Produced {item}')
        time.sleep(1)
```

```python
def consumer(queue):
  while True:
      item = queue.get()
      if item is None:
          break
      print(f'Consumed {item}')
      time.sleep(1.5)
```

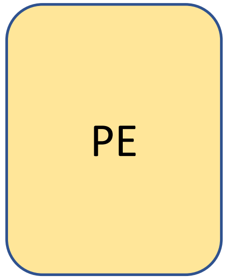
```python
if __name__ == '__main__':
    q = Queue()
    p = Process(target=producer, args=(q,))
    c = Process(target=consumer, args=(q,))
    p.start()
    c.start()
    p.join()
    q.put(None) # Signal the consumer to terminate
    c.join()
```

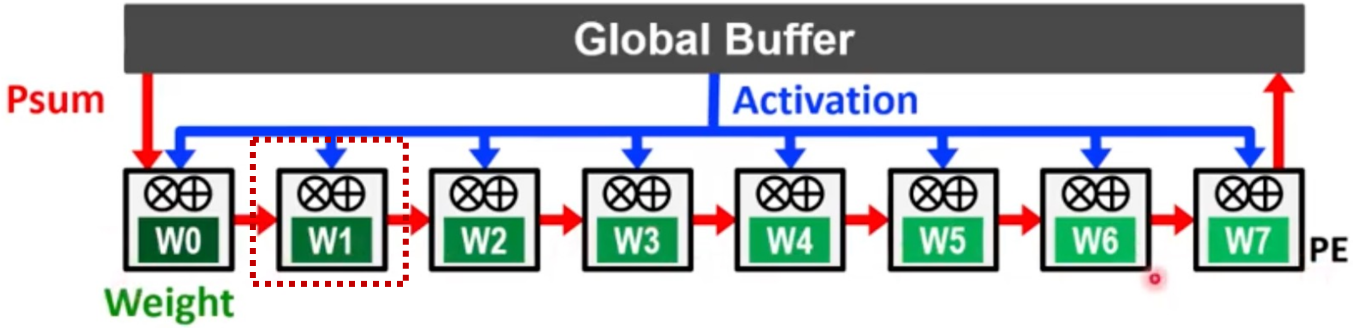
Can be run on multiple cores

What Does Each PE Do?

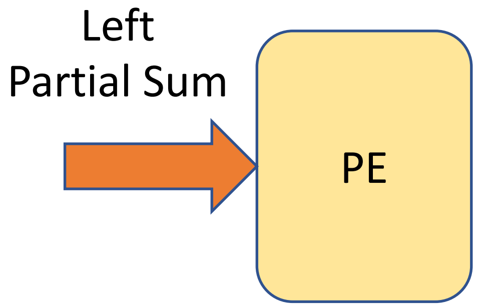


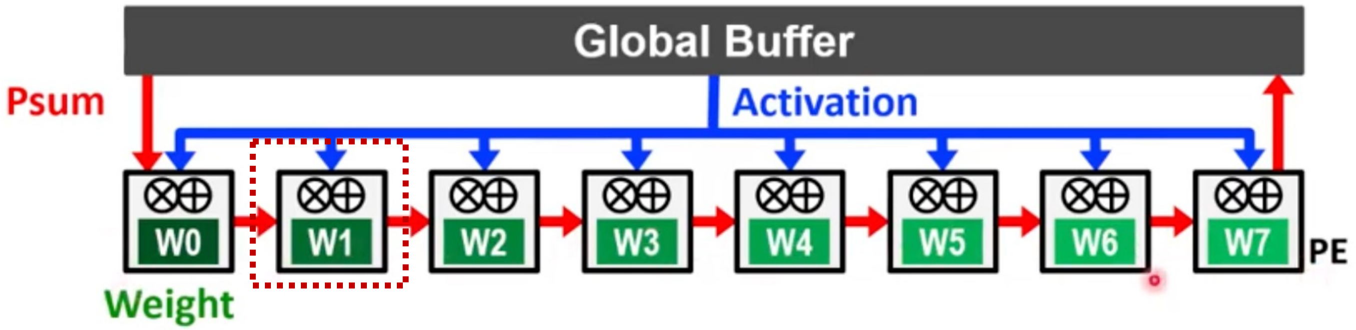
What Does Each PE Do?



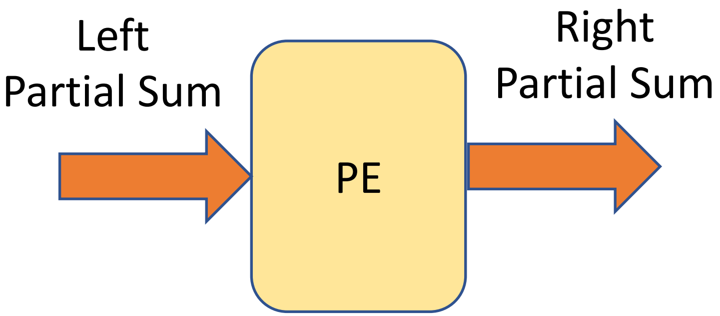


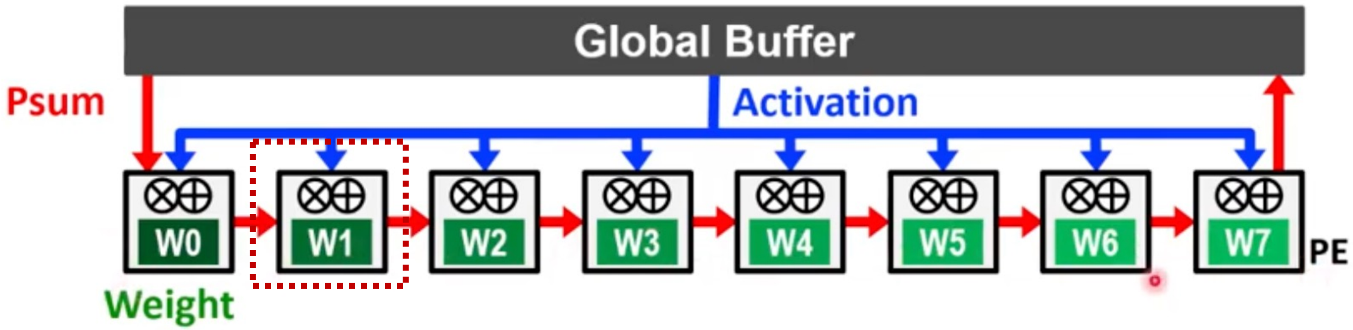
What Does Each PE Do?



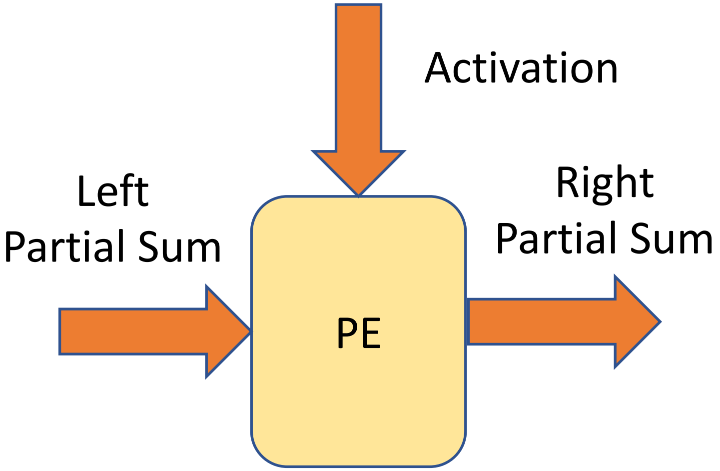


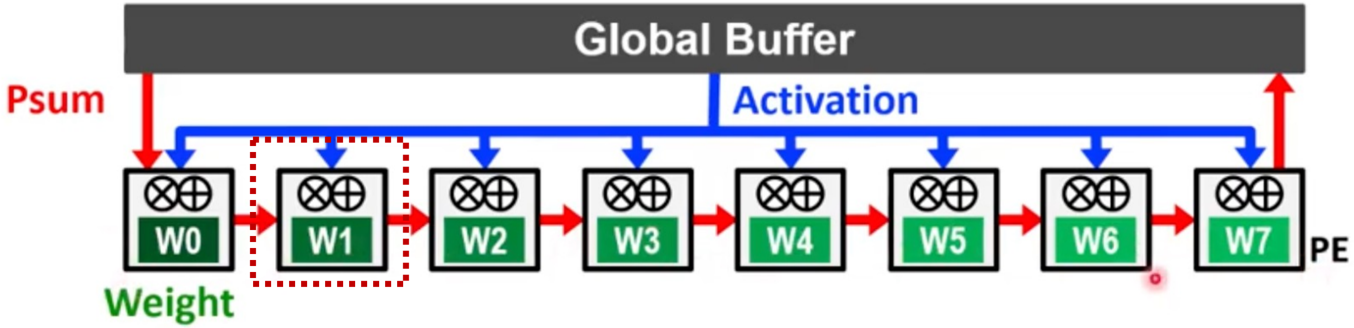
What Does Each PE Do?



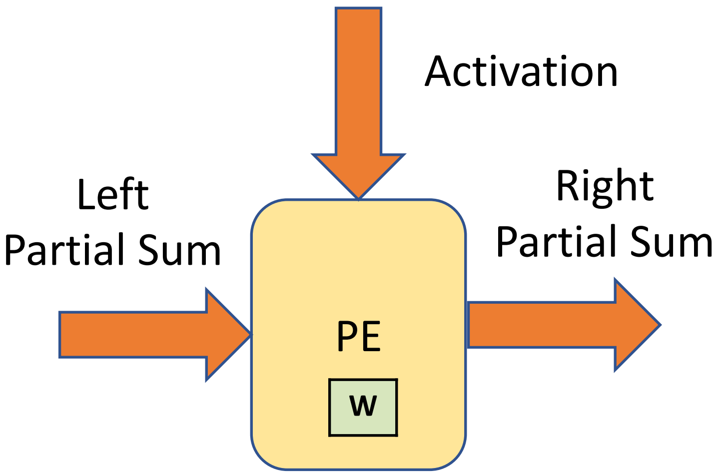


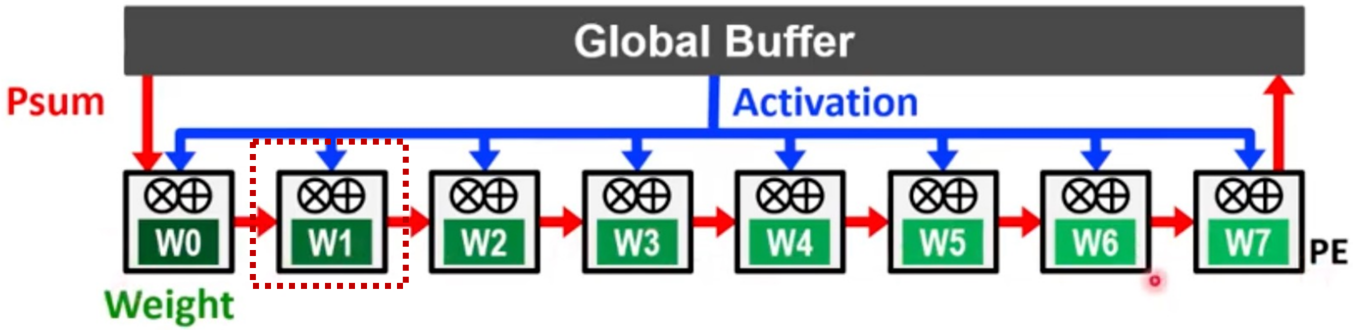
What Does Each PE Do?



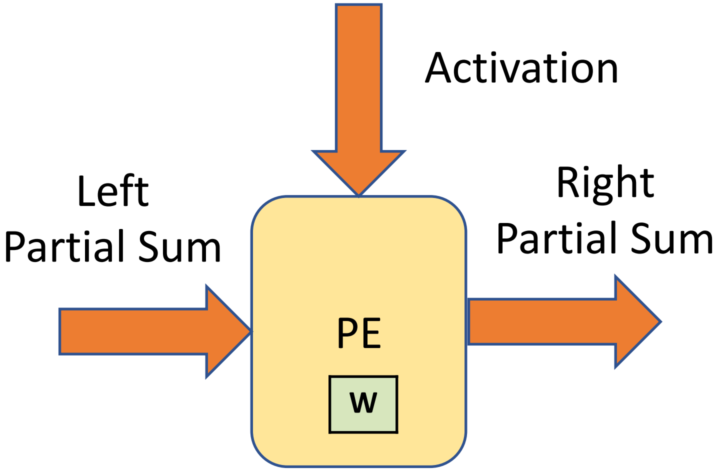


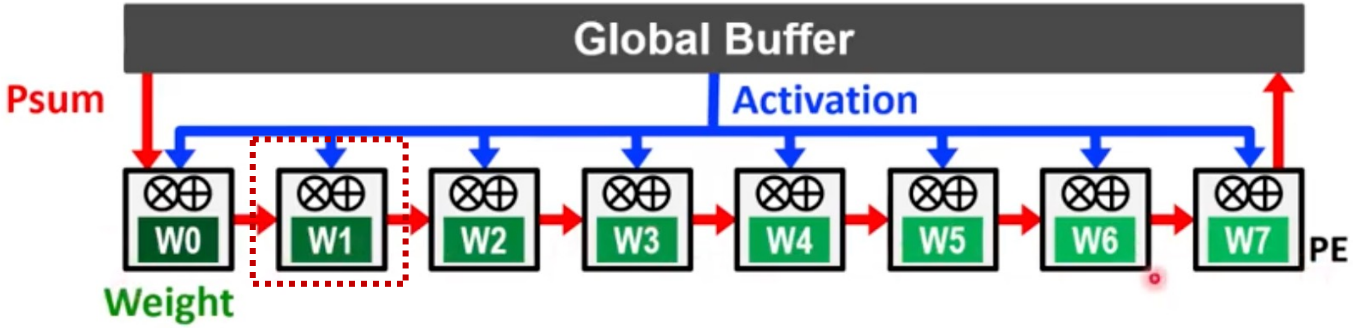
What Does Each PE Do?





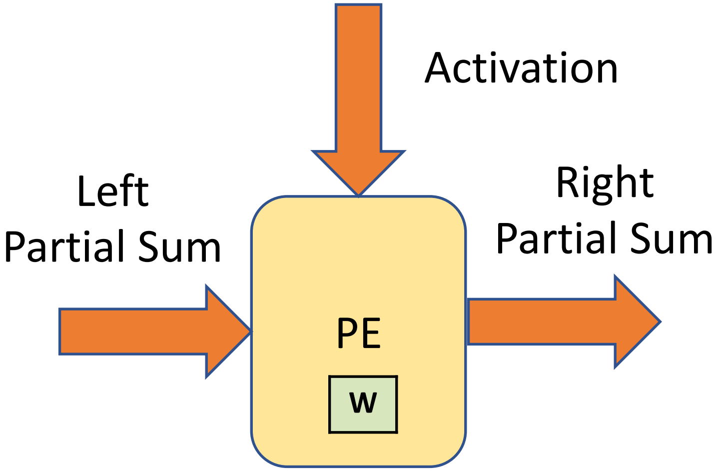
What Does Each PE Do?

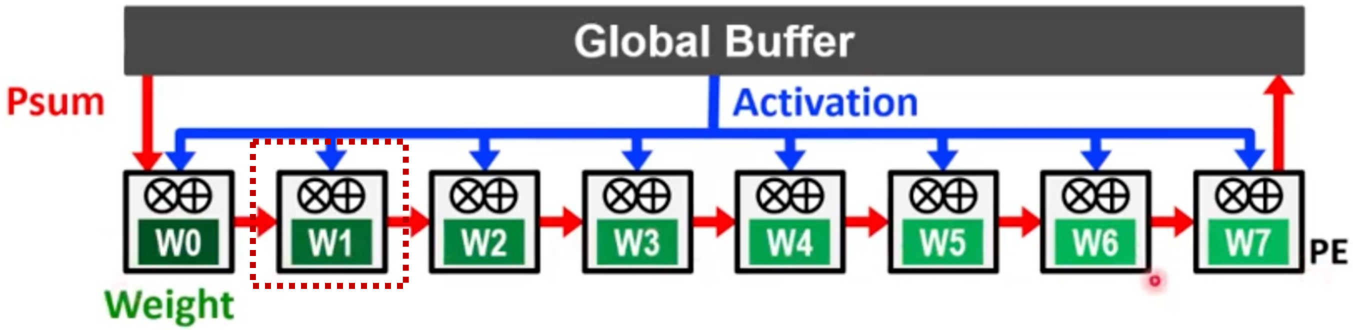





```python
def mac_worker(weight, input_queue,
               left_partial_sum_queue, right_partial_sum_queue):
```

What Does Each PE Do?

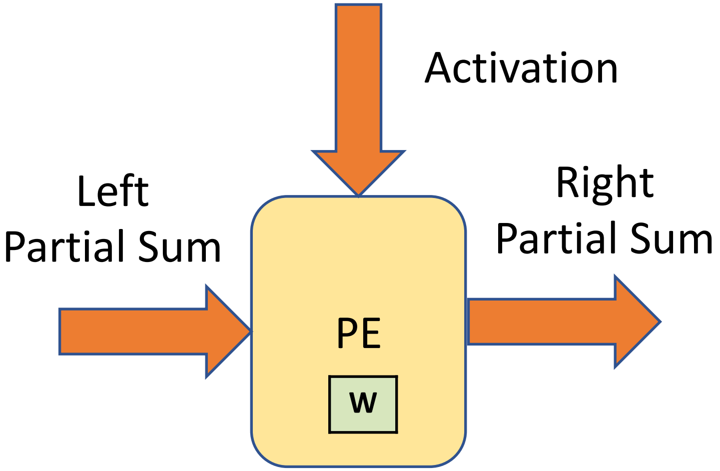


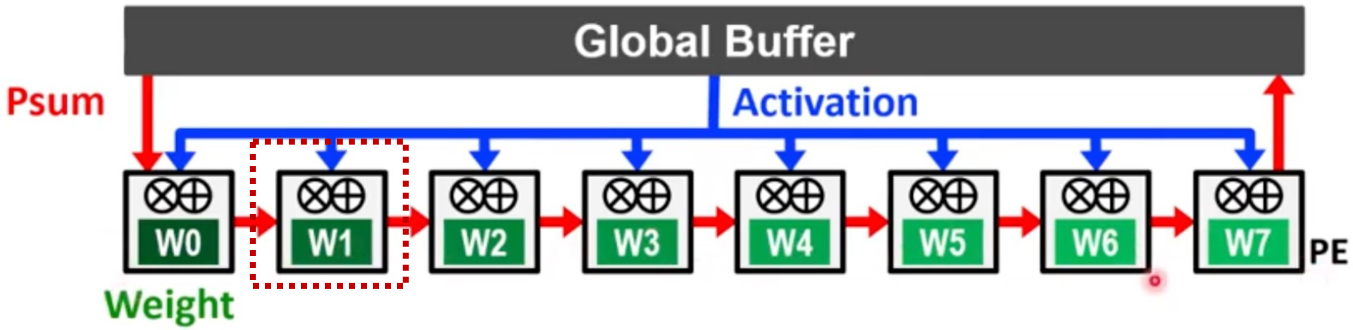



```python
def mac_worker(weight, input_queue,
               left_partial_sum_queue, right_partial_sum_queue):

    while True:
```

What Does Each PE Do?

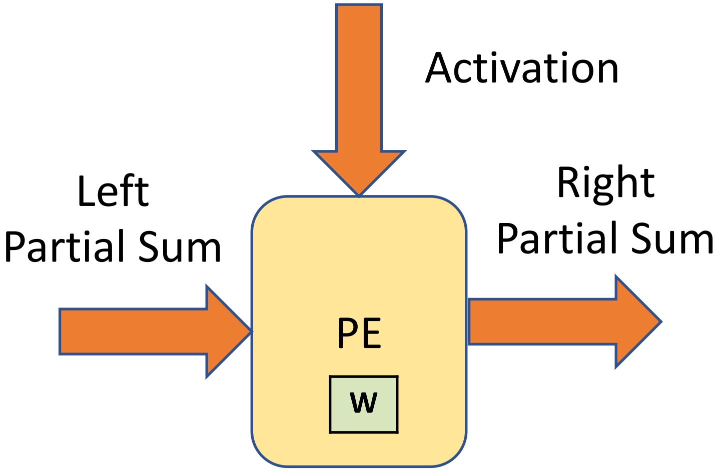


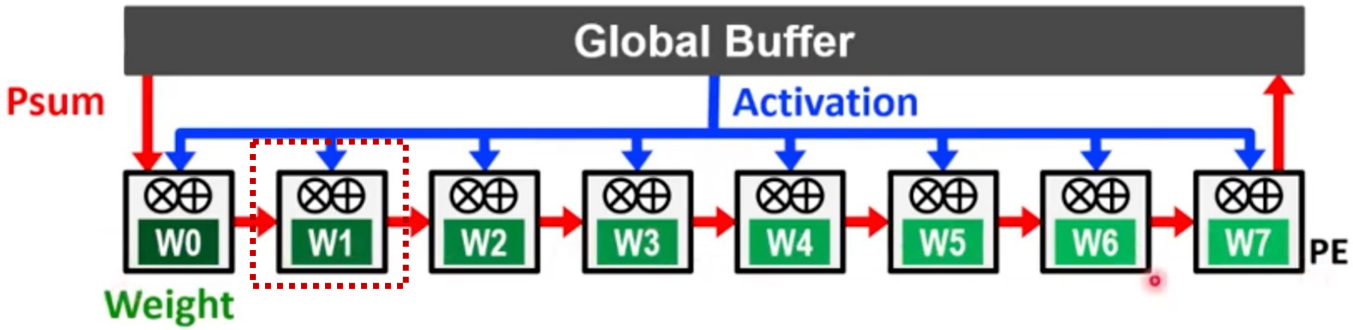



```python
def mac_worker(weight, input_queue,
               left_partial_sum_queue, right_partial_sum_queue):

    while True:
        activation = input_queue.get()
        if activation is None:
            break
```

What Does Each PE Do?

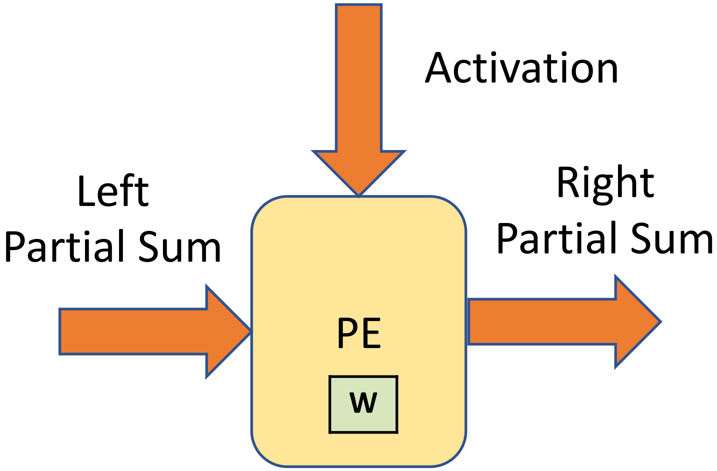


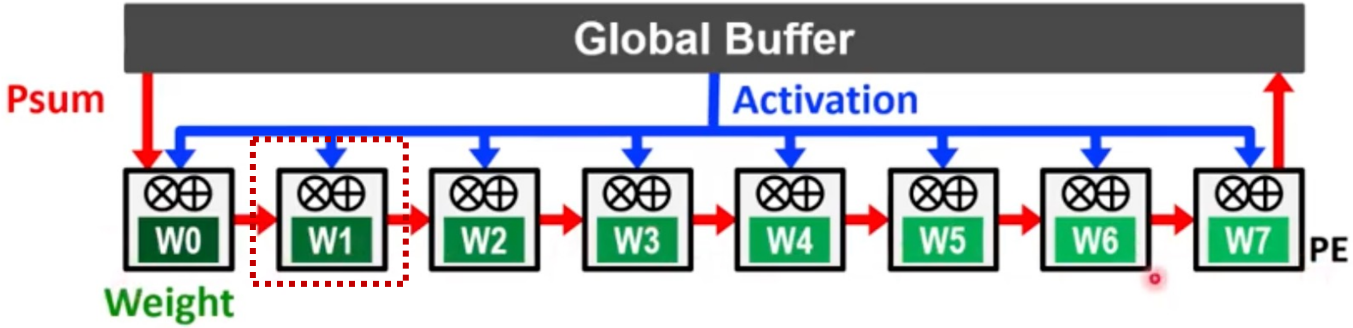



```python
def mac_worker(weight, input_queue,
               left_partial_sum_queue, right_partial_sum_queue):

    while True:
        activation = input_queue.get()
        if activation is None:
            break
        left_partial_sum = left_partial_sum_queue.get()
```

What Does Each PE Do?



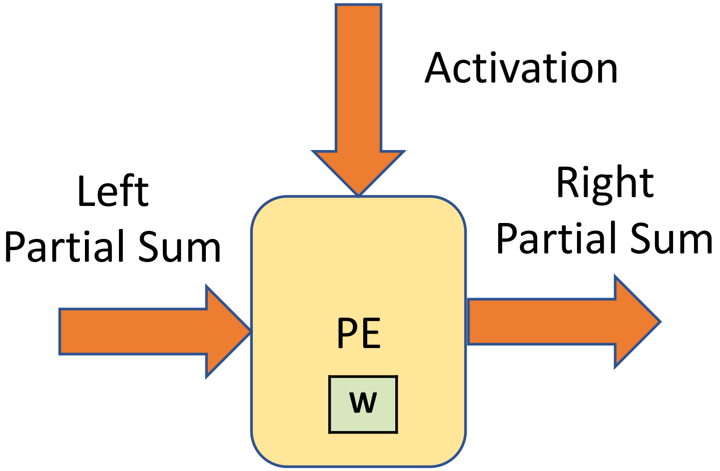


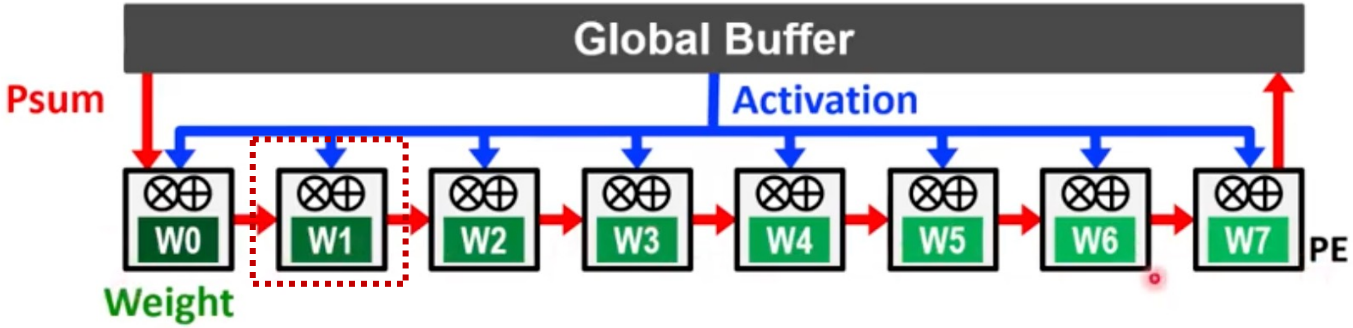

```python
def mac_worker(weight, input_queue,
               left_partial_sum_queue, right_partial_sum_queue):

    while True:
        activation = input_queue.get()
        if activation is None:
                break
        left_partial_sum = left_partial_sum_queue.get()

        right_partial_sum = left_partial_sum + activation * weight
```

What Does Each PE Do?





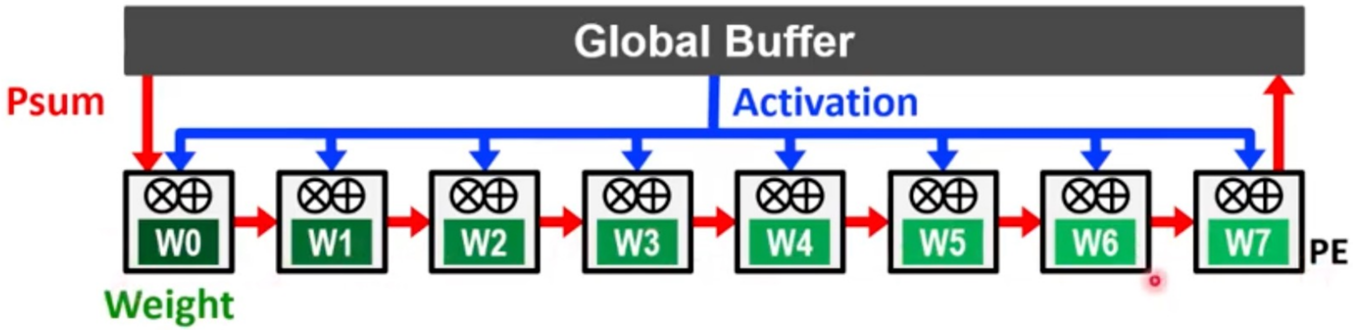

```python
def mac_worker(weight, input_queue,
               left_partial_sum_queue, right_partial_sum_queue):

    while True:
        activation = input_queue.get()
        if activation is None:
                break
        left_partial_sum = left_partial_sum_queue.get()

        right_partial_sum = left_partial_sum + activation * weight

        right_partial_sum_queue.put(right_partial_sum)
```

What Does Each PE Do?




```
num_workers = 8
weights = [1.0 * i for i in range(num_workers)] # Example weights

input_queues = [Queue() for _ in range(num_workers)]
partial_sum_queues = [Queue() for _ in range(num_workers + 1)] # Extra one for the initial input

# Initialize the leftmost partial sum queue
partial_sum_queues[0].put(0) # Initial partial sum


# Instantiate and start workers
workers = []
for i in range(num_workers):
        w = Process(target=mac_worker, args=(
        weights[i], input_queues[i], partial_sum_queues[i], partial_sum_queues[i+1]))
        workers.append(w)
        w.start()
```

Synchronous SystemVerilog Model

- Our Abstract Python models are asynchronous for simplicity.
- In practice PEs are commonly implemented synchronously using a clock signal

```
module MacWorker (
  input wire clk,
  input wire reset,
  input wire [31:0] activation, // Activation input
  input wire [31:0] weight, // Weight (stationary)
  input wire [31:0] left_partial_sum_signal, // Incoming partial sum
  output reg [31:0] right_partial_sum_signal // Output partial sum
);
  reg [31:0] left_partial_sum; // Stored partial sum (weight-stationary)

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      left_partial_sum <= 0;
      right_partial_sum_signal <= 0;
    end else begin
      left_partial_sum <= left_partial_sum_signal;
      right_partial_sum_signal <= left_partial_sum
                 + (activation * weight);
    end
  end
endmodule
```

Synchronous SystemC Model

- For high level modeling, SystemC can be used.
- Use SystemC if:
  - You need high-level system modeling (e.g., processor simulation, transaction-level modeling).
  - You want to write HLS-based designs (C++ → Verilog).
  - You are working on algorithm validation before RTL implementation.

```
#include <systemc.h>

SC_MODULE(MacWorker) {
  // Ports
  sc_in<bool> clk;
  sc_in<bool> reset;
  sc_in<sc_int<32>> activation; // Activation input
  sc_in<sc_int<32>> weight; // Stationary weight input
  sc_in<sc_int<32>> left_partial_sum_signal; // Incoming partial sum
  sc_out<sc_int<32>> right_partial_sum_signal; // Output partial sum
  // Internal register for pipeline efficiency
  sc_signal<sc_int<32>> left_partial_sum;

  // MAC process
  void mac_process() {
    if (reset.read() == 1) {
      left_partial_sum.write(0);
      right_partial_sum_signal.write(0);
    } else {
      // Store the left partial sum (stabilizing input)
      left_partial_sum.write(left_partial_sum_signal.read());

      // Perform MAC operation
      right_partial_sum_signal.write(left_partial_sum.read() +
              (activation.read() * weight.read()));
    }
  }

  // Constructor
  SC_CTOR(MacWorker) {
  SC_METHOD(mac_process);
  sensitive << clk.pos(); // Triggered on the positive clock edge
  sensitive << reset; // Also sensitive to reset
  }
};
```

Modeling in PyTorch or TensorFlow

```
import torch

activation = torch.tensor(3, dtype=torch.int32)
weight = torch.tensor(2, dtype=torch.int32) # Stationary weight
left_partial_sum = torch.tensor(5, dtype=torch.int32)

# Compute MAC operation
right_partial_sum = left_partial_sum + (activation * weight)

print("Right Partial Sum (PyTorch):", right_partial_sum.item())
```

Best for **research, flexible ML models, and GPU execution**

```
import tensorflow as tf

# Define input tensors
activation = tf.constant(3, dtype=tf.int32)
weight = tf.constant(2, dtype=tf.int32) # Stationary weight
left_partial_sum = tf.constant(5, dtype=tf.int32)

# Compute Right Partial Sum
right_partial_sum = left_partial_sum + (activation * weight)

# Run in TensorFlow
print("Right Partial Sum:", right_partial_sum.numpy())
```

Best for **training and deploying models on GPUs/TPUs**.

Note that these are used for mapping to existing hardware

Modeling in TVM

- **High-Level ML Model**
  - TVM takes models from **TensorFlow, PyTorch, ONNX, etc.**
  - Converts them into **Relay IR** (Intermediate Representation).
- **Optimizations and Lowering**
  - TVM applies **auto-scheduling, memory layout transformations, loop optimizations**.
  - Lowers computations to **hardware-specific backends**.
- **Generates Target-Specific Code**
  - **For CPUs → LLVM IR** (Compiles to assembly/machine code).
  - **For NVIDIA GPUs → CUDA** (Optimized for tensor cores).
  - **For FPGAs → OpenCL / HLS C++** (Can be synthesized into Verilog/VHDL).
  - **For Custom ASICs → Maps to low-level tensor instructions** (But does not generate Verilog directly).

```python
import tvm
from tvm import te
import numpy as np

A = te.var("A") # Activation
W = te.var("W") # Stationary weight
L = te.var("L") # Left partial sum

# Compute Right Partial Sum
Right_Partial_Sum = te.compute(
(1,), lambda i: L + (A * W), name="Right_Partial_Sum"
)

# Create a schedule
s = te.create_schedule(Right_Partial_Sum.op)

# Compile for CPU (Can change to CUDA for GPU or FPGA backend)
target = "llvm"
f = tvm.build(s, [A, W, L, Right_Partial_Sum], target=target)

# Run with sample values
ctx = tvm.cpu()
A_val = tvm.nd.array(np.array([3], dtype="int32"), ctx)
W_val = tvm.nd.array(np.array([2], dtype="int32"), ctx) # Stationary
L_val = tvm.nd.array(np.array([5], dtype="int32"), ctx)
R_out = tvm.nd.empty((1,), dtype="int32", ctx)

f(A_val, W_val, L_val, R_out)
print("Right Partial Sum (TVM):", R_out.asnumpy()[0])
```

TVM is mostly used for mapping to existing hardware, it does not replace SystemVerilog

Best for **optimizing ML models for custom hardware (FPGAs, TPUs, ASICs, Edge devices).**

TensorFlow vs. PyTorch vs. TVM

Feature	**TensorFlow**	**PyTorch**	**TVM**
Purpose	ML model definition & execution	ML model definition & execution	ML model compilation & optimization for hardware
Hardware Optimization	Uses **XLA** (Accelerates on CPU, GPU, TPU)	Uses **TorchScript** & ONNX for model optimization	Uses **Relay IR** to map models onto **CPU, GPU, FPGA, ASICs**
Custom Hardware Support	❌ No, limited to TensorFlow-supported hardware	❌ No direct hardware mapping, but can export to TVM	✅ Yes, supports custom **ASICs, FPGAs, TPUs**
Programming Model	High-level ML framework (Keras, TF functions)	High-level ML framework (Eager execution, TorchScript)	Low-level tensor scheduling and compilation
Fine-Grained Scheduling	❌ No	❌ No	✅ Yes, allows **custom scheduling for different hardware**
Deployment	Cloud (TPU), GPU, Edge (TF Lite)	Cloud (GPU, CPU), Edge (TorchScript)	Can optimize for **embedded devices, TPUs, GPUs, and FPGAs**
Example Use Case	Training & deploying ML models on **NVIDIA GPUs & TPUs**	Training & deploying ML models on **NVIDIA GPUs**	**Optimizing models for hardware accelerators (e.g., FPGAs, ASICs, Edge TPUs)**