TU WIEN Informatics

Advanced Computer Architecture

GP-GPUs, TPUs, NPUs

Daniel Mueller-Gritschneder

Content

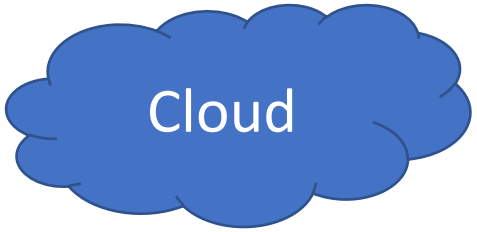
- Motivation: Era of Deep Learning
- GP GPUs
- TPUs / NPUs

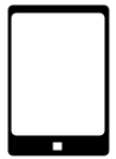
Motivation: Era of Deep Learning

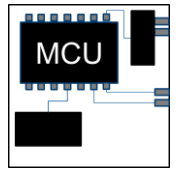
Use of Data-level Parallelism (DLP)

ML Plattforms are Heterogeneous

- Large computing continuum with possibly connectivity:







Datacenter:
Multi-Servers
with Multi-GPUs

Hundreds of CPUs
Hundreds of GBs of DRAM
Several GPUs with
Tens of GB of DRAM
Several TB of Storage

Desktop/Workstation /Fog:
PC with GPU

2-128 CPUs
Tens of GBs of DRAM
1-2 GPUs with Tens of GB of DRAM
A few TB of Storage

Edge/Mobile:
Mobile Phone
Raspberry PI
Embedded GPU
Specialized SoCs

1-4 CPUs
1-4 GBs of DRAM
1 GPUs with a few GB of DRAM
Specialized Accelerators
Tens to Hundreds of GB of Storage

Extreme Edge / TinyML:
MCU
Specialized low-power SoC

1 CPU
Hundreds of kB to a few MB of
embedded SRAM
Low-power Acceleration / Co-proccesors
A few MB of Storage, e.g. embedded Flash

Cloud ML

Desktop ML

Embedded Machine Learning / Edge AI

Deep Learning Models are Heterogeneous

- **In type**: Deep Neural Networks, Convolutional Neural Networks, Transformers, Graph Neural Networks, Recursive Neural Networks
- **In computing demand**: often measured in MAC operations
- **In size**: often measured in number of parameters
- Examples:
  - Large Language Models (LLMs) -produces human-like text
    - GPT-4: 170 trillion (10e12) parameters
    - GPT-3: 175 billion (10e9) parameters
  - ResNet18 – 11 million (10e6) parameters – Image classification e.g. for autonomous driving
  - Keyword Spotting (KWS): 16k-300k (10e3) parameters – Detects keyword in an audio stream, e.g. for Audio wakeup (TinyML)

Embedded ML Applications

- Data is generated at the edge by several sensors.
- ML application is executed on an embedded device “close to” the sensor.

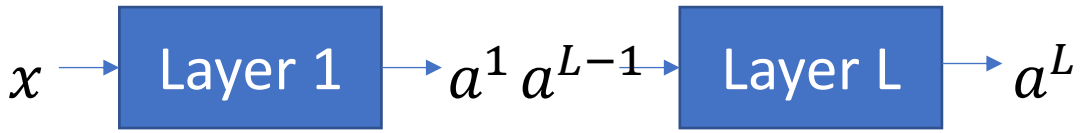
- Examples:
  - Autonomous driving based on HD camera, Lidar and radar
  - Wearable human activity tracking using Gyros, accelerometers
  - Visual wake up from camera
  - Audio wake up (keyword spotting) from microphone
  - Gesture recognition from radar sensor

ANN Architectures

- Layered computation:

$$a^l = f^l(a^{l-1})$$

$$a^L = f^L(f^{L-1}(\dots f^1(x)\dots))$$



- Example Layer: Fully-connected
  - Weights: W
  - Activations: a
  - Activation function: f^l
    - Typical activation functions: ReLU, tanh, softmax

$$a^l = f^l(Wa^{l-1}) + b^{l-1})$$

Design of Neural Network Architecture & Automated Network Architecture Search

- Design of the ANN model architecture must consider target system
  - ROM/RAM Memory resources (weights, activations)
  - Computational power: Operations (Support for nonlinear operations)
  - Acceleration features (type of layers, layer configurations)

- Network Architecture Search (NAS)
  - Algorithms that systematically explore different ANN model architectures in an automatic way
  - Computationally very expensive (training of many candidates to evaluate the accuracy)

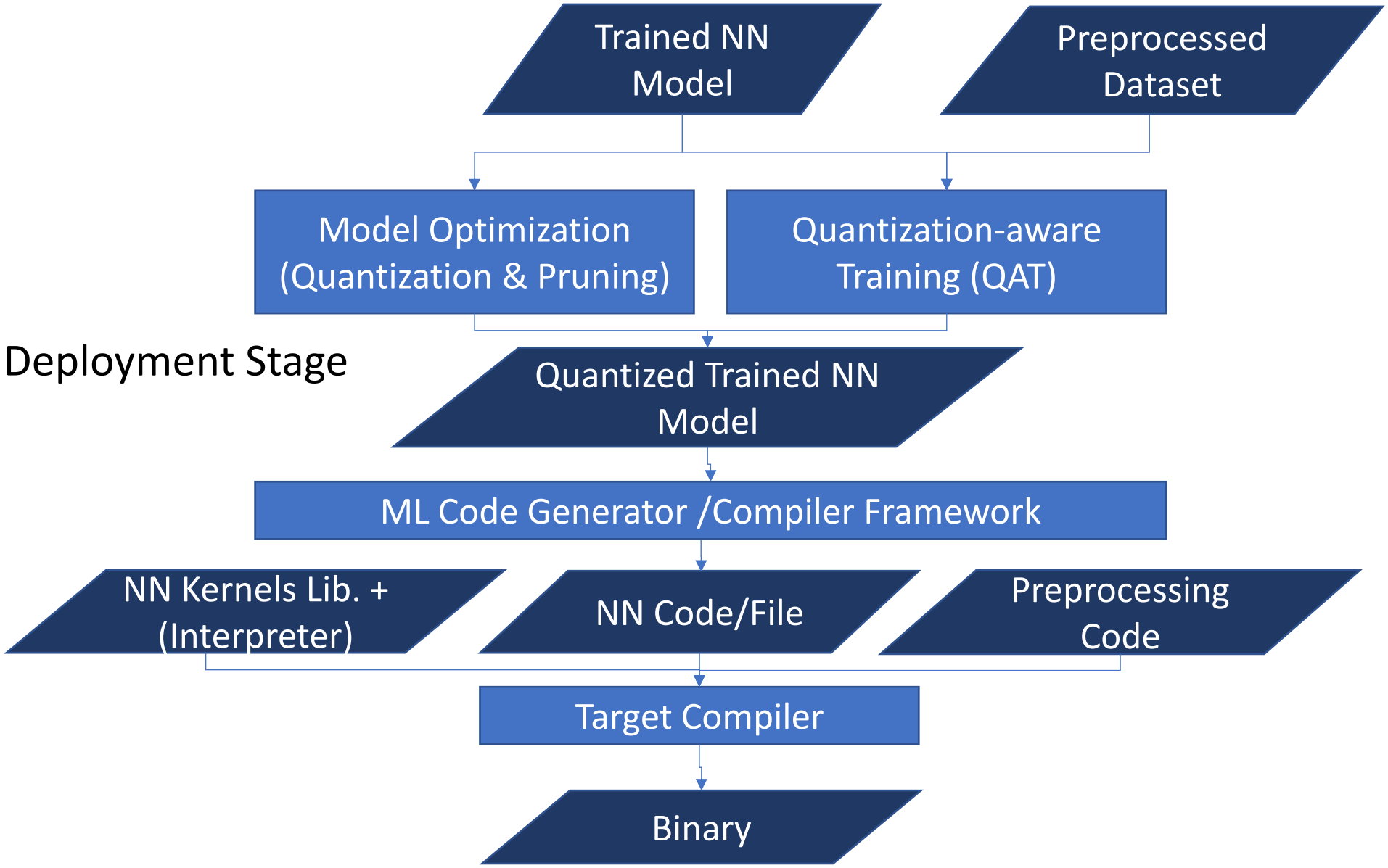
Training

- Training of the ANN model is done on a powerful machine (GPU)
- Trained model is deployed on the embedded device
- Embedded device executes the trained model (inference task)

- Training:
  - Selection of the hyperparameters
  - Optimization of the trainable parameters of the ANN model
  - Using usually a backpropagation algorithm
-
-

Flow(2/2)

- Deployment Stage



Quantization

- Model are usually trained with floating-point (FP) precision (float, double).

- Inference (execution of trained model on embedded device)
  - Full precision (FP) computation (multiplication, addition) expensive
  - HW Floating Point Units expensive (area, energy)

  - For inference the model is transferred to a quantized variant
  - Integer computations (less expensive)
  - Many challenges: Rounding, Overflow, Rescaling, Shifting
  - Simple Example (8bit integer [-127 … 128]):

$$z^l = \begin{bmatrix} 1.4 & 150.5 \\ 8.3 & 2.6 \end{bmatrix} a^{l-1} + \begin{bmatrix} 3.7 \\ 2.4 \end{bmatrix} \quad \Rightarrow \text{Quantization} \quad z^l = \begin{bmatrix} 1 & 128 \\ 8 & 3 \end{bmatrix} a^{l-1} + \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

Quantization Formats

- Many possible formats:
- **Integer formats** for weights/activations usually given by bit-width: x-bit
- On many embedded processors (byte-type quantization simplest 8bit, 16bit)
  - Byte /uint8 (8bit) quantization range: [0 … 255]
- Accumulation variables usually larger size
- Sub-byte integer quantization <8bit

- Binary quantization *w* in {0,1}
- Ternary quantization *w* in {-1,0,1}

- Also reduced-precision floating-point possible (many formats)

Pruning

- Unstructured pruning: Small weight values are set to zero
  - Skip computation with zero values (might require additional logic in program)
  - Simple example:

$$z^l = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \\ 5 & 12 & 3 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \\ 7 \end{bmatrix}$$

Unstructured Pruning

$$z^l = \begin{bmatrix} 155 & 0 & 8 \\ 17 & 38 & 234 \\ 0 & 12 & 0 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 0 \\ 7 \end{bmatrix}$$

- Structured pruning: A column, row, kernel is removed from the operator
  - Operator is modified
  - Example:

$$z^l = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \\ 5 & 12 & 3 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \\ 7 \end{bmatrix}$$

Structured Pruning

$$z^l = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \\ 0 & 0 & 0 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \\ 0 \end{bmatrix}$$

$$z^{*l} = \begin{bmatrix} 155 & 1 & 8 \\ 17 & 38 & 234 \end{bmatrix} a^{l-1} + \begin{bmatrix} 66 \\ 7 \end{bmatrix}$$

Example: Convolutional Neural Network

- Consists of layers (structure reprented by data flow graph)

$$
\begin{aligned}
\mathbf{A}^1 &= \text{Conv2D}(\mathbf{X}, \mathbf{W}^1, \mathbf{b}^1, \sigma_s^1, \sigma_r^1, \delta_s^1, \delta_r^1, P^1, \text{ReLu}) \\
\mathbf{A}^2 &= \text{maxpool}(\mathbf{A}^1 \pi_r^2, \pi_s^2) \\
\mathbf{A}^3 &= \text{Conv2D}(\mathbf{A}^2, \mathbf{W}^3, \mathbf{b}^3, \sigma_s^3, \sigma_r^3, \delta_s^3, \delta_r^3, P^3, \text{ReLu}) \\
\mathbf{A}^4 &= \text{maxpool}(\mathbf{A}^3, \pi_r^4, \pi_s^4) \\
\mathbf{a}^5 &= \text{flatten}(\mathbf{A}^4) \\
\mathbf{a}^6 &= \text{Dense}(\mathbf{a}^5, \mathbf{W}^6, \mathbf{b}^6, \text{Softmax})
\end{aligned}
$$

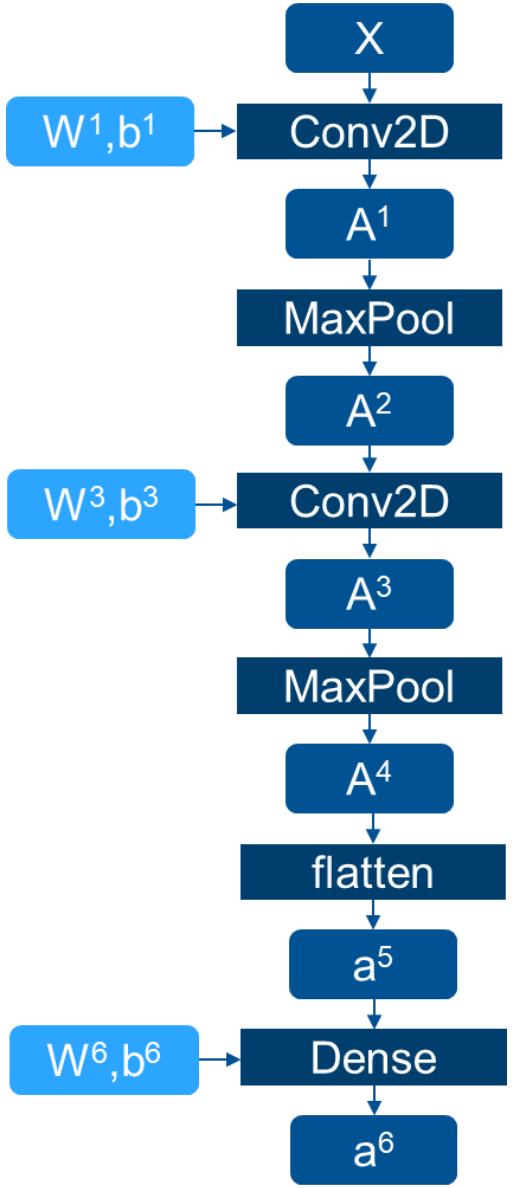


Image to Column (Img2Col) Transformation

- For many targets there exist a very optimized implementation of matrix-matrix-multiply computation e.g. accelerators, for CPUs with some SIMD support, GPUs, but also single-issue CPUs

a) Fully connected layers and convolution layers require to multiply activation and weights and accumulate the result, which basically results in many required MAC operations.

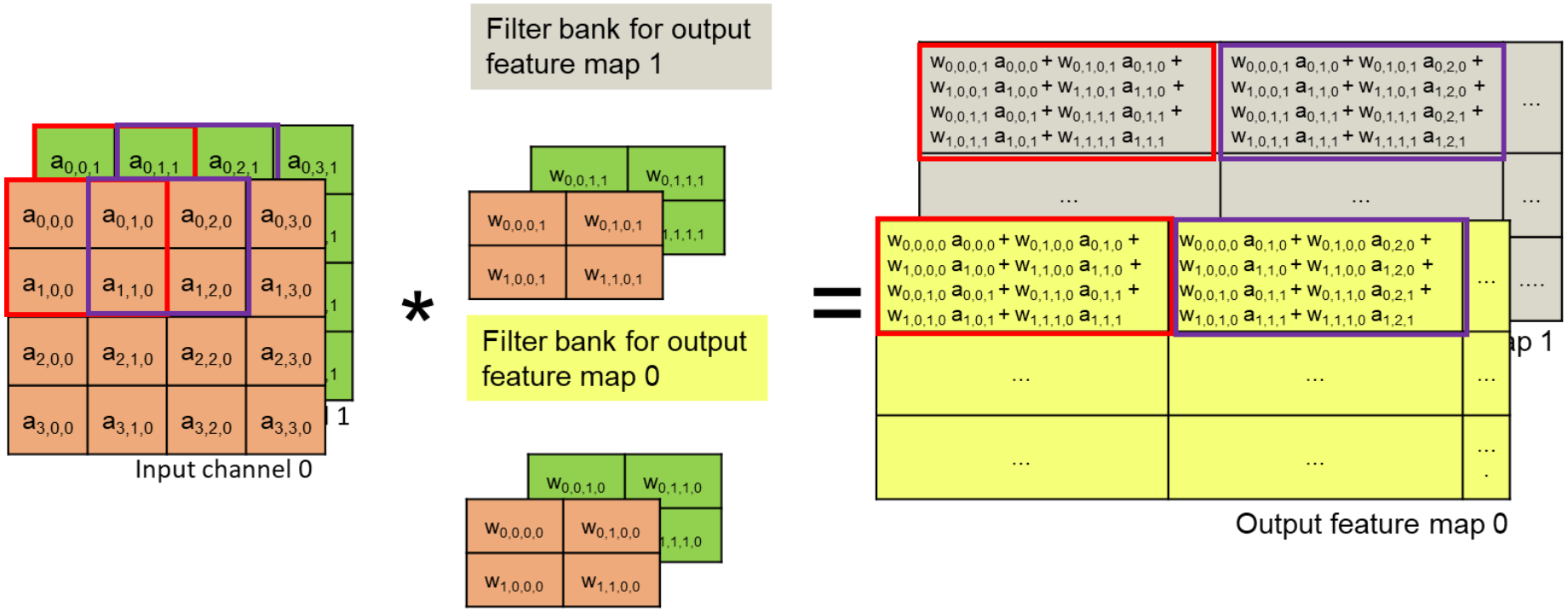
- Img2Col transforms a convolution operation into a matrix-matrix-multiply operation

- Img2Col requires to build up a batch matrix, which is larger than the original activation tensor, because it holds duplicates of some values

➢ Usually Img2Col is not done on the full input activation tensor but inside the convolution loop on some part of the tensor in order to avoid building up the full batch matrix

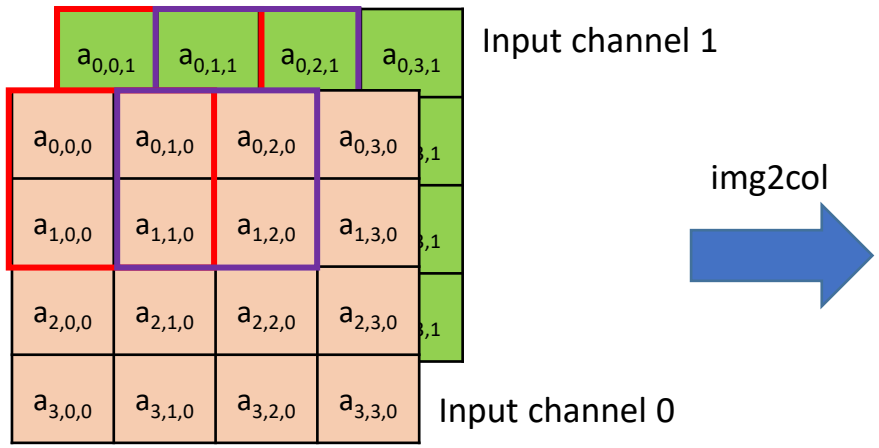
Example for Img2Col (1/5)

- For reference: This is the Standard Convolution



Example for Img2Col (2/5)

- Step 1 for Img2Col: Create col-based batch matrix
- Each line holds the activation values under one kernel position for all channels



img2col

batch 1	batch 2	
$a_{0,0,0}$	$a_{0,1,0}$	…
$a_{0,1,0}$	$a_{0,2,0}$	…
$a_{1,0,0}$	$a_{1,1,0}$	…
$a_{1,1,0}$	$a_{1,2,0}$	…
$a_{0,0,1}$	$a_{0,1,1}$	…
$a_{0,1,1}$	$a_{0,2,1}$	…
$a_{1,0,1}$	$a_{1,1,1}$	…
$a_{1,1,1}$	$a_{1,2,1}$	…

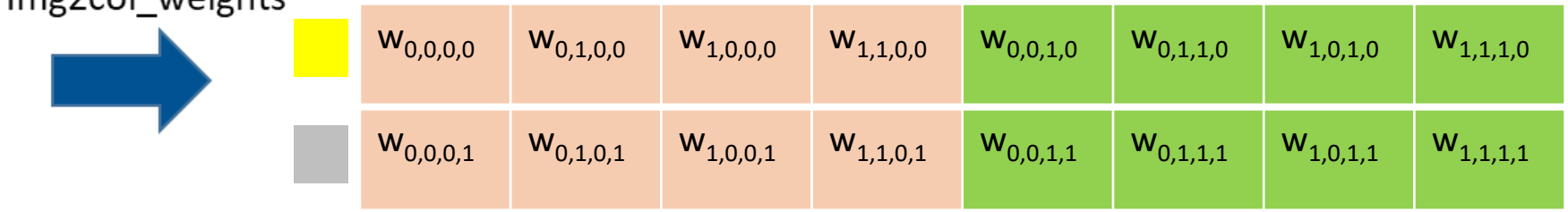
Example for Img2Col (3/5)

- Step 2: Create a row-based filter matrix. (Can be done already offline, is already existing with just storing weight tensor in ROM memory)

Filter bank for output feature map 1 (FM1)

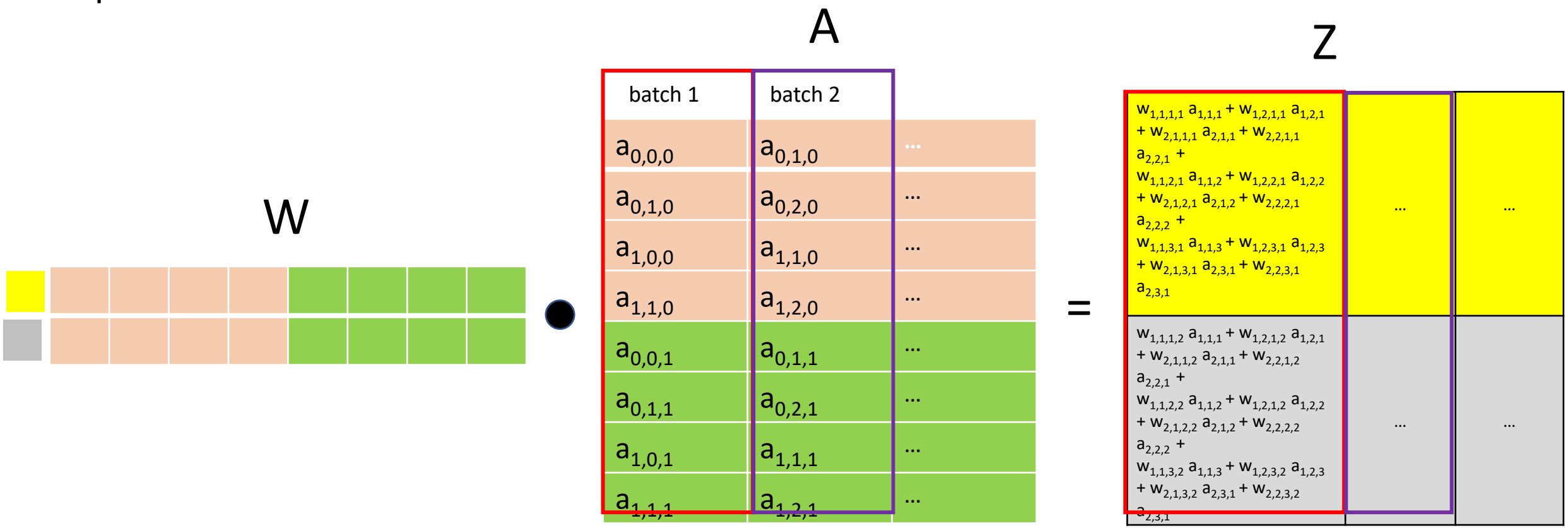
Filter bank for output feature map 0 (FM0)

Img2col_weights



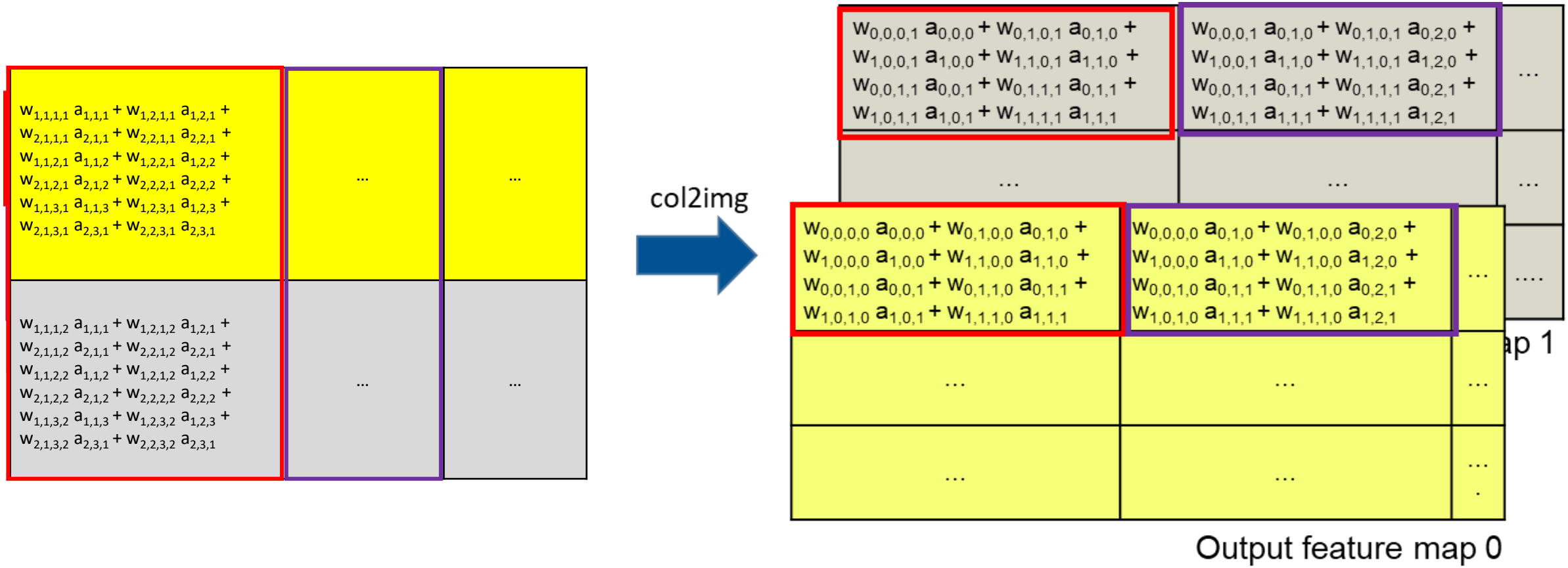
Example for Img2Col (4/5)

- Step 3: Run a matrix-matrix multiplication with target-specific optimized GEMM kernel



Example for Img2Col (5/5)

- Step 4: Reshape the output to recover the output feature maps using the inverse col2img transformation.



GEMM Algorithm

- Basic linear algebra algorithm for matrix-matrix-multiply

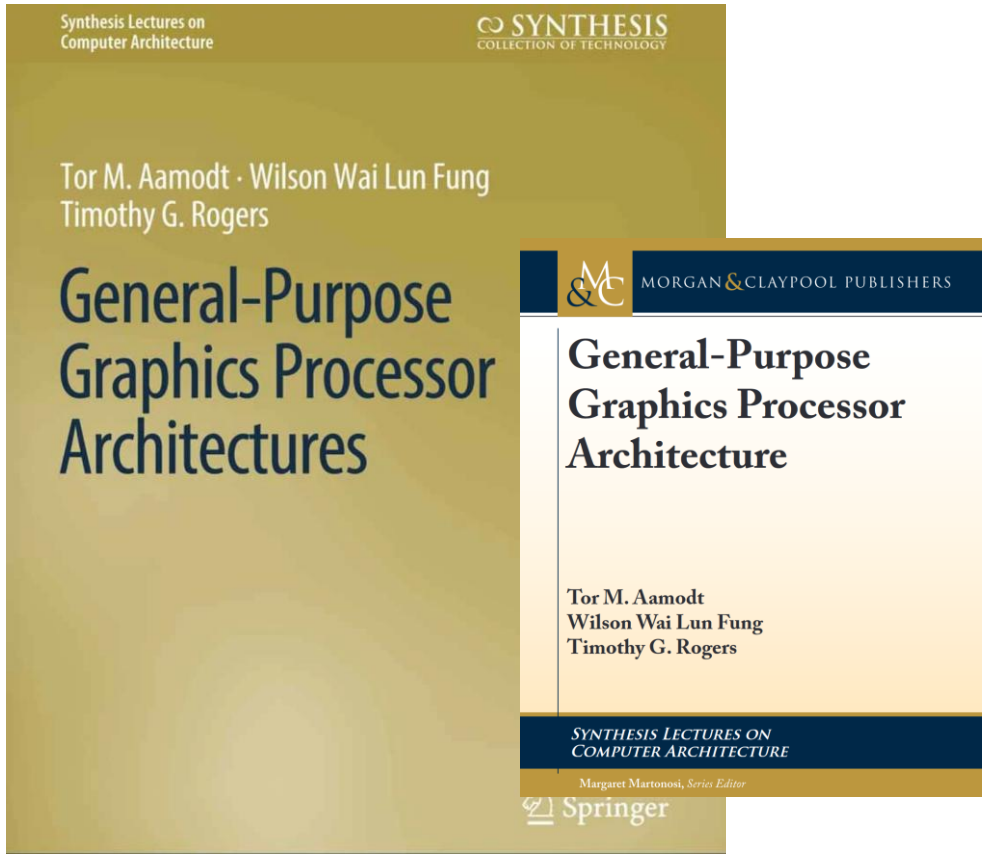
- Optimized versions exist for many hardware platforms e.g.
  - Considering block-wise computation depending on cache sizes
  - Exploiting **data-level parallelism (DLP)**

•GEMM is seen as „*at the heart of deep learning*“ especially when acceleration is considered.

Further reading:
https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

General-Purpose Graphics Processor Units (GPGPUs)

Source



Inspired by:

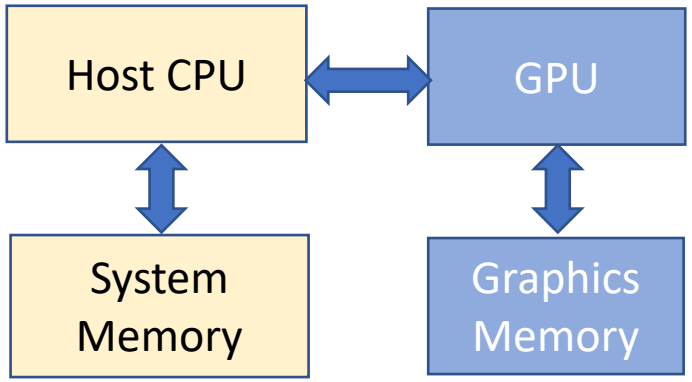
- Book: Aamodt, Fung & Rogers – Generap-Purpose Graphics Processor Architectures
- Book: Hennesy &Patterson: Computer Architecture – A Qualitative Approach
- CA Course: Sophia Shao, UC Berkeley

GPUs

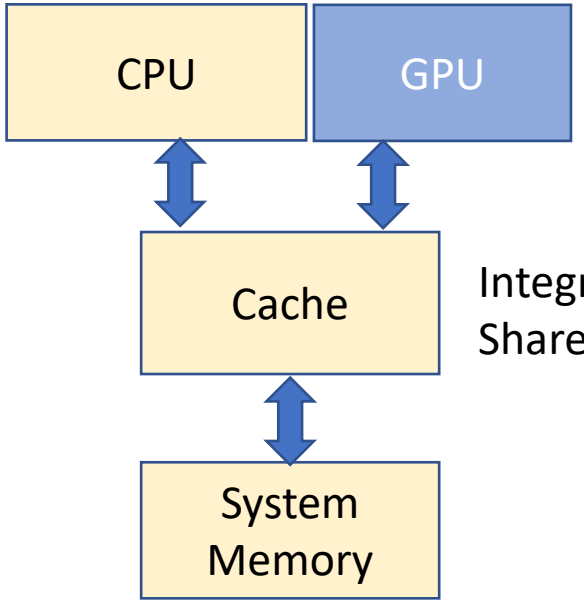
- GPUs were initially introduced for rendering in real time especially for video games.
- Nowadays GPUs can be found in many devices (Data Centers, PCs, Laptop, Phones, Embedded GPUs...)
- General Purpose (GP-GPU): Programming Language CUDA from NVIDIA allowed to use GPUs for other compute besides rendering (now especially used for ML)

GPU (Discrete vs. Integrated)

- GPUs are combined with a CPU either on a single chip or by inserting an additional card (e.g. via PCIe).
- The CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU. The CPU is often called "*the host*".



Discrete GPU: Own memory



Integrated GPU: Shared memory

Basic Programming Model

- CPU (Example Code):

```
void saxpy_serial(int n, float a, float *x, float *y) {
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

```
…
saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
…
```

Basic Programming Model

- GPU (CUDA):

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i<n)
y[i] = a*x[i] + y[i];
}
```

Compute Kernel

```
…
float *d_x, *d_y;
int nblocks = (n + 255) / 256;
cudaMalloc( &d_x, n * sizeof(float) );
cudaMalloc( &d_y, n * sizeof(float) );
cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
...
```

Setup and call kernel from CPU program

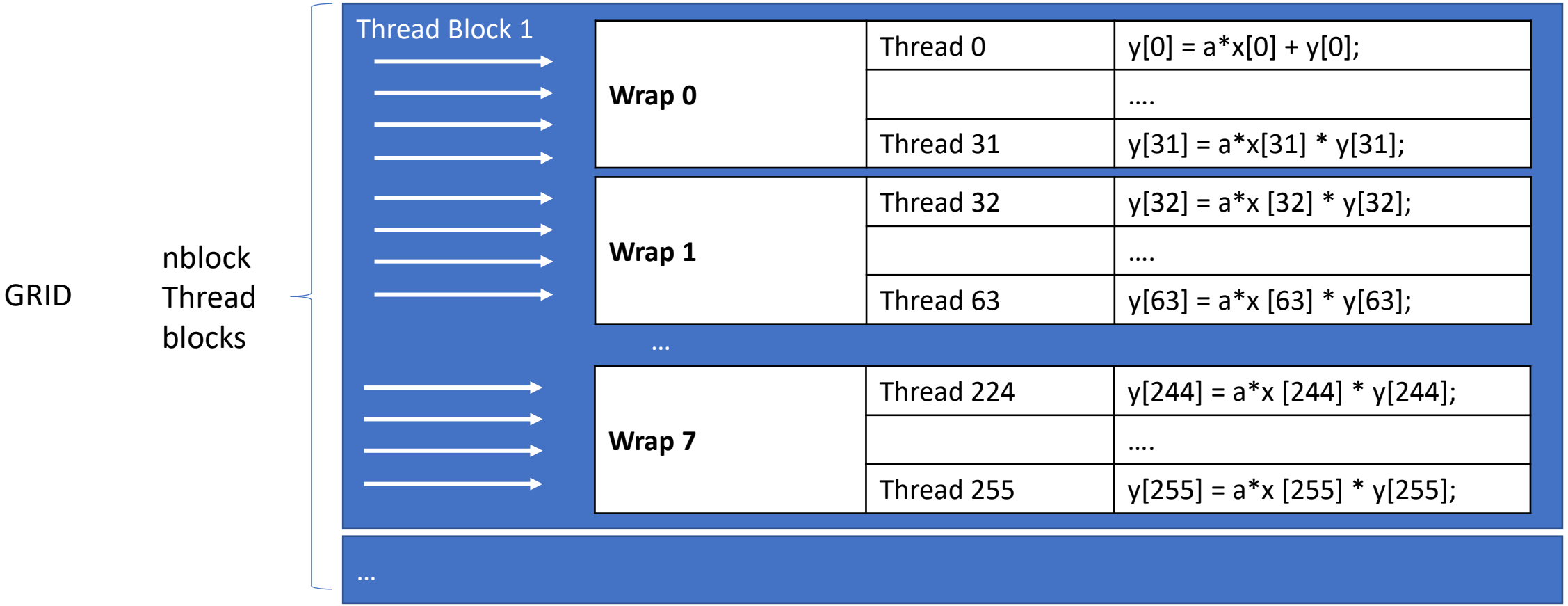
Threads, Warps, Thread block

- The threads that make up a compute kernel are organized into a hierarchy composed of a *grid* of *thread blocks* consisting of *warps*.
- In the CUDA programming model, individual threads execute instructions whose operands are scalar values (e.g., 32-bit floating-point).
- To improve efficiency typical GPU hardware executes groups of threads together in lock-step (SIMD). These groups are called *warps, which* consist of 32 threads
- Warps are grouped into a larger unit called *thread block* by NVIDIA.

Example:

saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

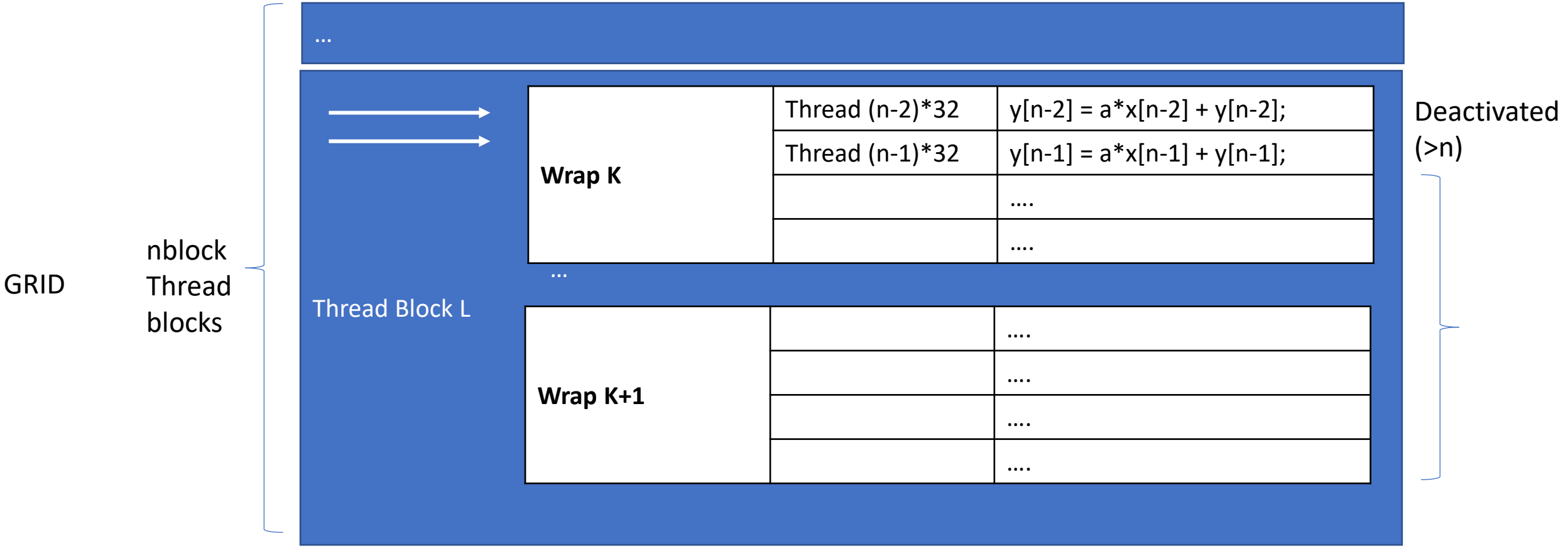
- Launch a single grid, consisting of nblocks thread blocks
- Each thread block contains 256 threads (8 warps).



Example:

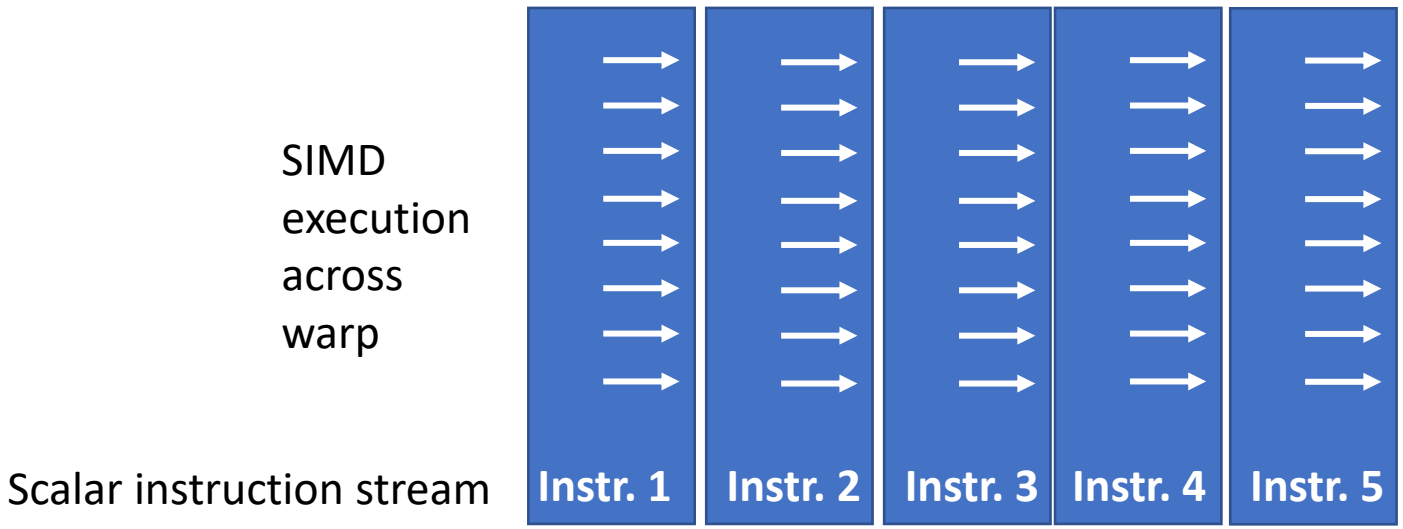
saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

- Threads with thread_idx.x > n are deactivated



Single Instruction, Multiple Thread (SIMT)

- GPUs uses the **Single Instruction, Multiple Thread (SIMT)** model
- Scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware
- Loads and stores are scatter-gather, as threads perform scalar loads and stores

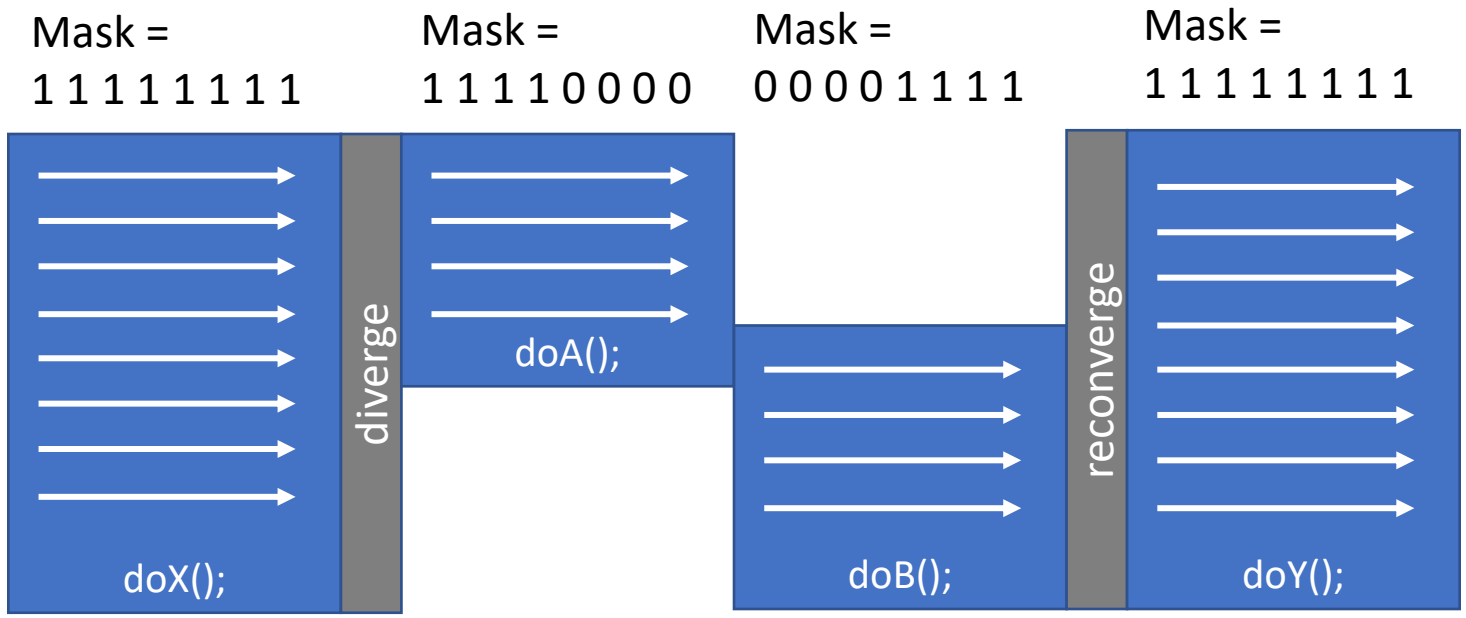


Divergence and Reconvergence of Threads

- Warps execute in lock-step SIMD fashion
- Threads may diverge/reconverge due to control flow

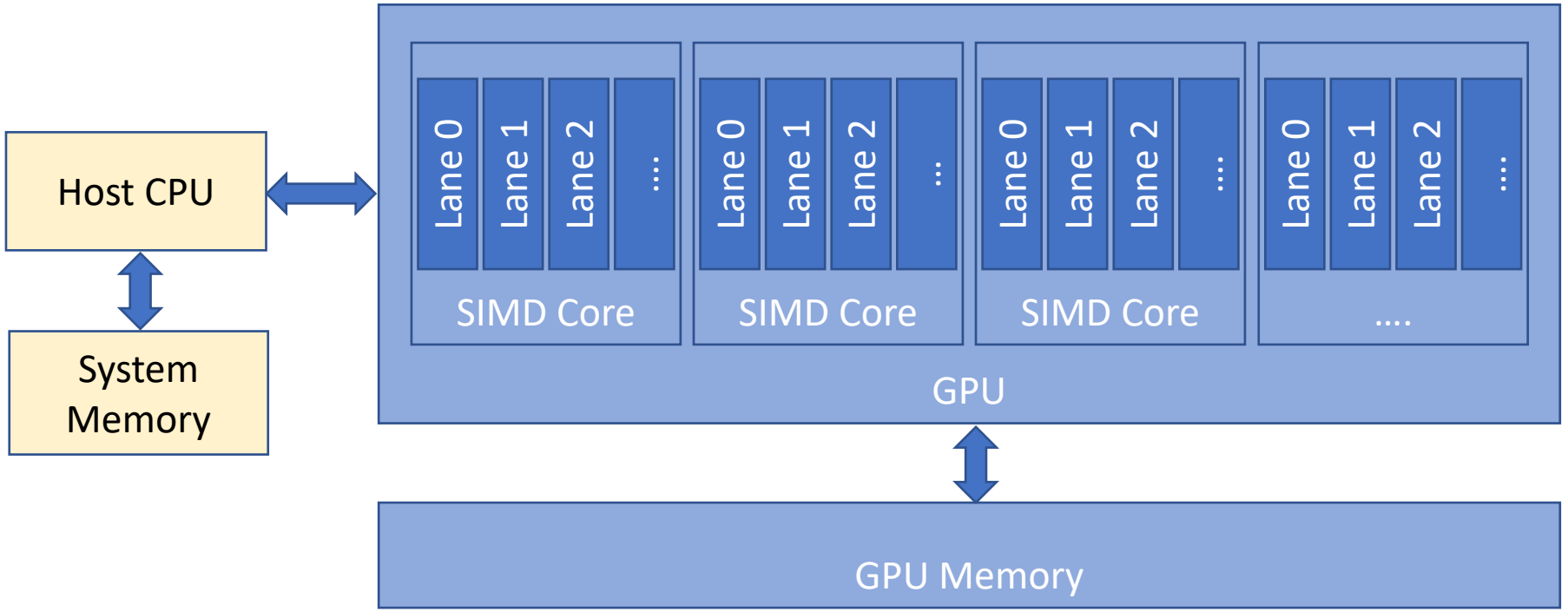
- Simplified illustration (arrows are threads in a thread block):

```
doX();
if (threadIdx.x < 4) {
  doA();
} else {
  doB();
}
doY();
```



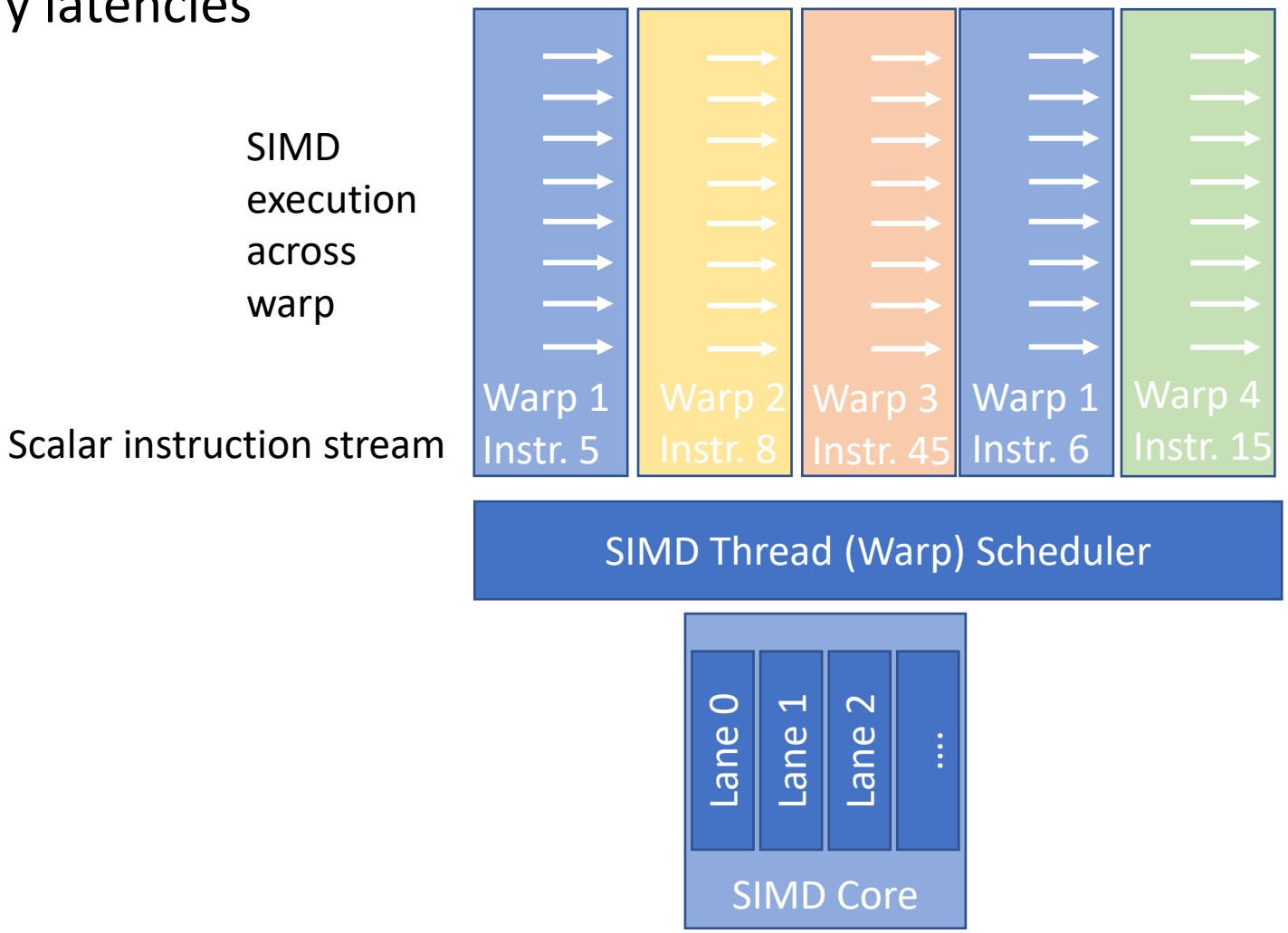
Hardware Execution Model

- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)

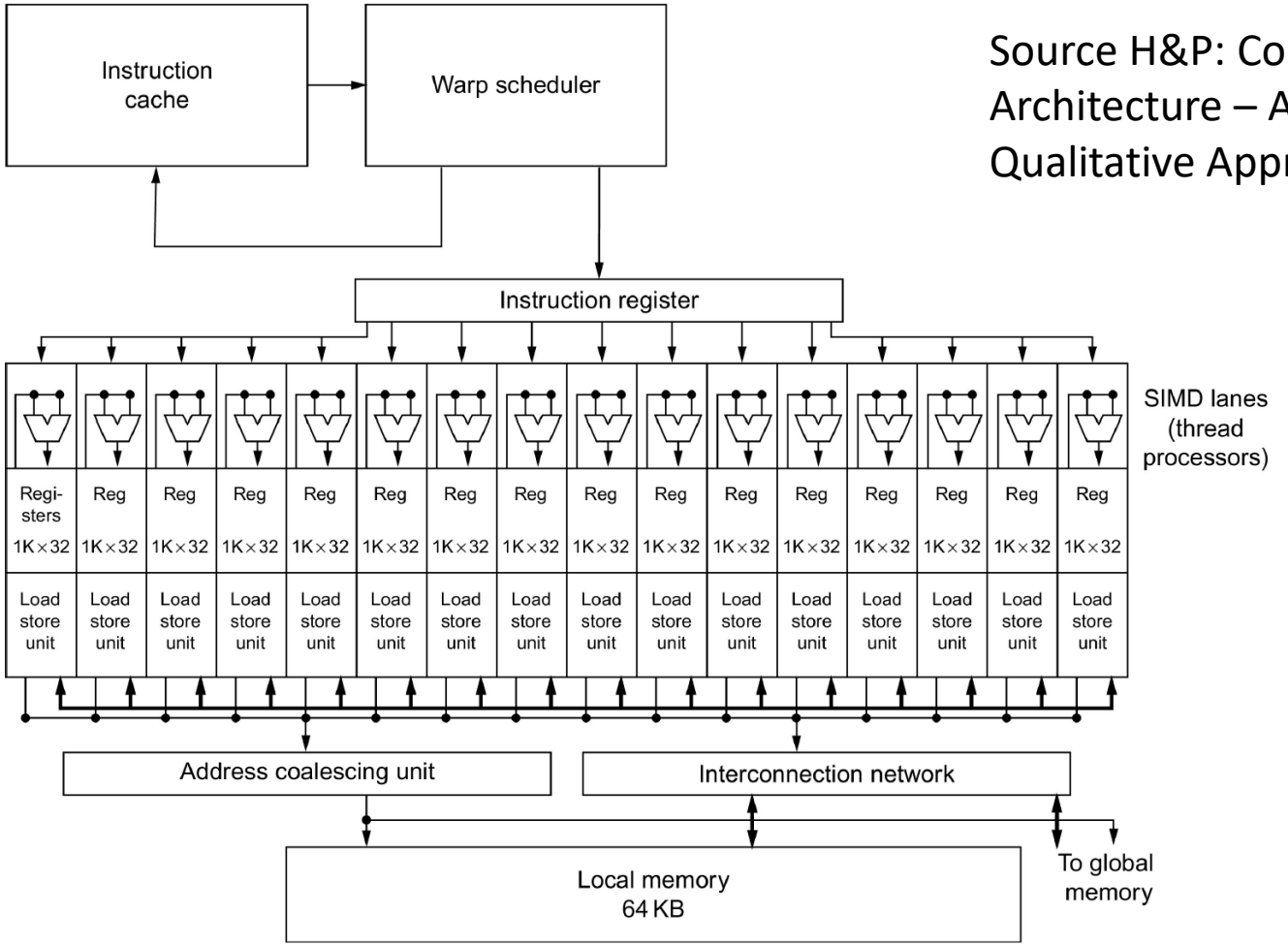


Multithreading on SIMD Processor

- SIMD cores execute instructions of independent warps in multithreaded fashion
- E.g. can hide memory latencies



Multithreaded SIMD Processor



Source H&P: Computer Architecture – A Qualitative Approach

Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

Look at a Real GPU: A100

Optional, not relevant for exam

A100 GPU -128 Streaming Multiprocessor

NVIDEA calls
SIMD processors
Streaming Multiprocessors (SMs)

Source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

SM

- “A100 has four Tensor Cores per SM, which together deliver 1024 dense FP16/FP32 FMA operations per clock”
- “432 Third-generation Tensor Cores per GPU” (108 SMs)

Table 1. NVIDIA A100 Tensor Core GPU Performance Specs

Peak FP64[1]	9.7 TFLOPS
Peak FP64 Tensor Core[1]	19.5 TFLOPS
Peak FP32[1]	19.5 TFLOPS
Peak FP16[1]	78 TFLOPS
Peak BF16[1]	39 TFLOPS
Peak TF32 Tensor Core[1]	156 TFLOPS | 312 TFLOPS[2]
Peak FP16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak BF16 Tensor Core[1]	312 TFLOPS | 624 TFLOPS[2]
Peak INT8 Tensor Core[1]	624 TOPS | 1,248 TOPS[2]
Peak INT4 Tensor Core[1]	1,248 TOPS | 2,496 TOPS[2]

1 - Peak rates are based on GPU Boost Clock.
2 - Effective TFLOPS / TOPS using the new Sparsity feature

Source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Accelerators - Systolic Array

Systolic Array

Concept:

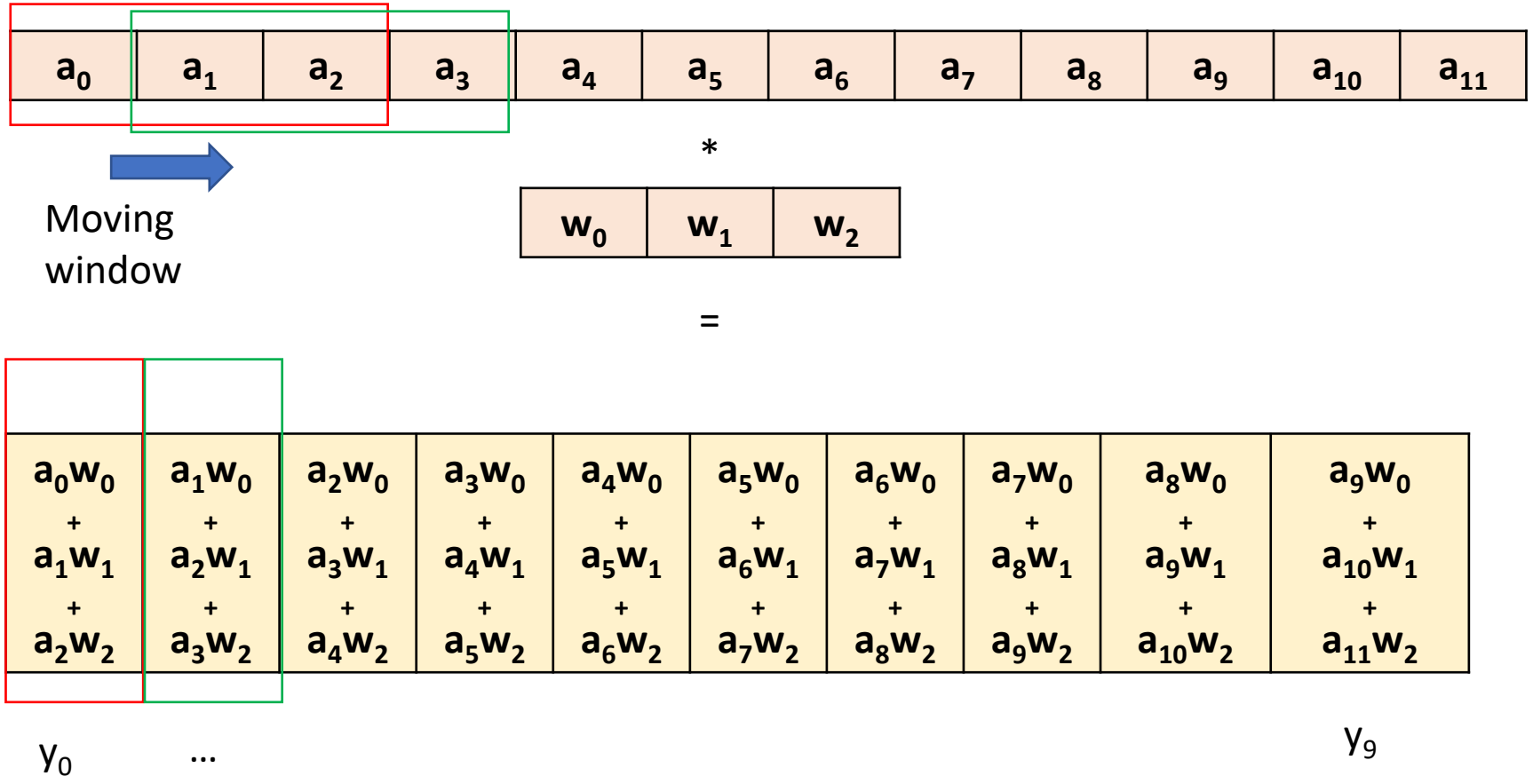
- Functional Units (FUs) are chained to implement a fixed type of computation
- Flow inside systolic array needs to be carefully orchestrated
- Intermediate results are directly moved to next FU

- 2D systolic arrays often used for deep learning for Matrix-matrix multiply (GEMM), called *Tensor Cores*, *GEMM Core, Matrix Multiply Unit*
- Systolic arrays can be designed for many other computations

Example: 1D Convolution

```
void conv1D_12_3(int* x, int* w, int* y) {
for (i=0; i<10;i++) {
  y[i]=0;
  for (j=0;j<3;j++) {
    y[i] += x[i+j] * w[j];
  }
}
}
```

- Simple 1D convolution (A1x12)*(1x3):



Example: 1D Convolution

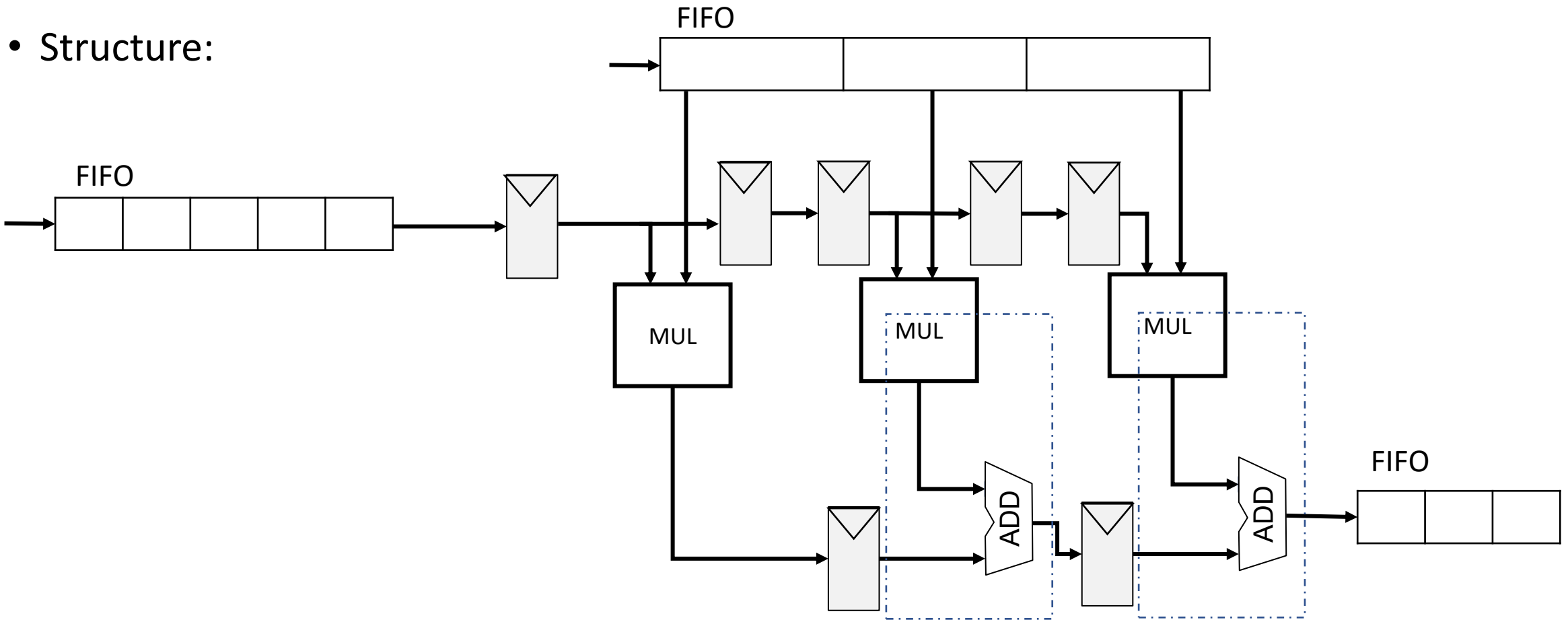
- Code

```
void conv1D_12_3(int* x, int* w, int* y) {
for (i=0; i<10;i++) {
  y[i]=0;
  for (j=0;j<3;j++) {
    y[i] += x[i+j] * w[j];
  }
}
}
```

```
conv1D_12_3:
  LW t1,0(a1)   # w0
  LW t2,4(a1)   # w1
  LW t3,8(a1)   # w2
  LI t4,0
conv1D_12_3_loop:
  LW a4,0(a0)    # x[i+0]
  LW a4,4(a0)    # x[i+1]
  MUL a1,a4,t1  # x[i+0] * w[0]
  MUL a4,a4,t2  # x[i+1] * w[1]
  LW a5,8(a0)    # x[i+2]
  ADD a1,a1,a4
  MUL a5,a5,t3  # x[i+2] * w[2]
  ADD a1,a1,a5
  SW a1,0(a2)  # Store y[i]
  ADDI a0,a0,4
  ADDI a1,a1,4
  ADDI t4,t4,1
  BNE t4,10, conv1D_12_3_loop
RET
```

Example: 1D Convolution - Systolic Array (1D) - Structure

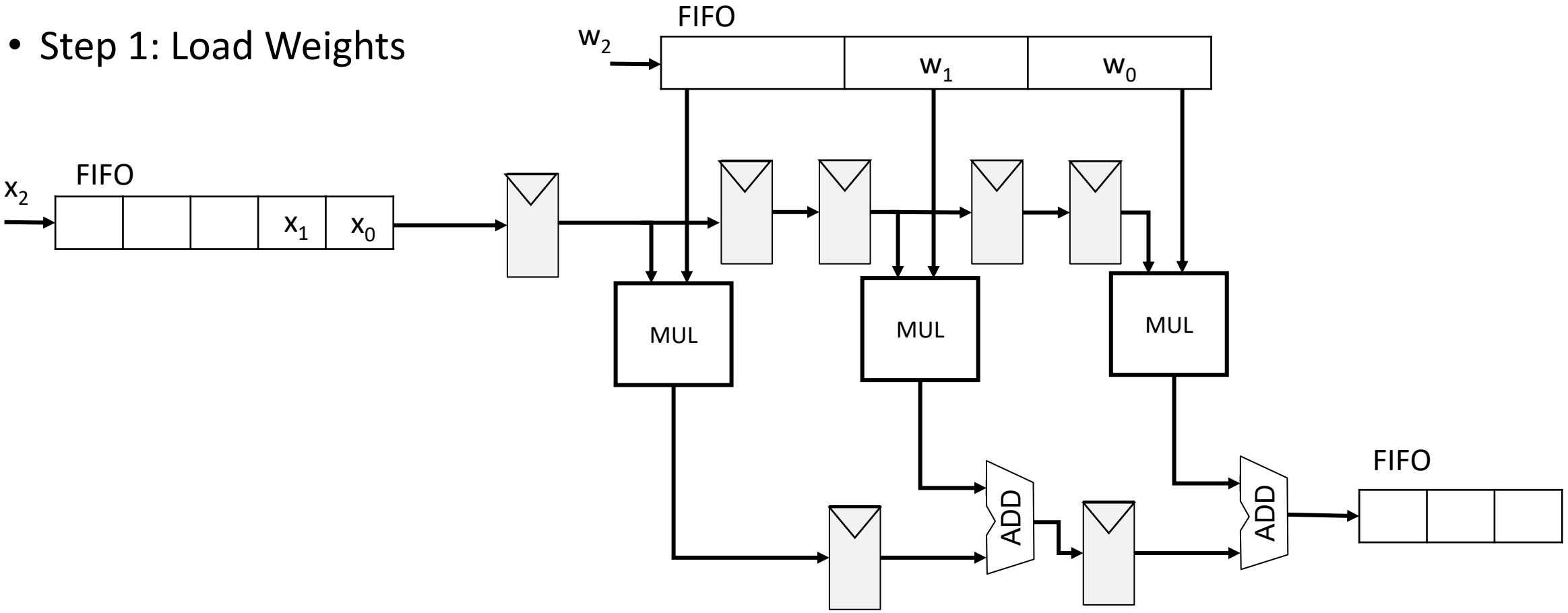
- Structure:



MUL+ ADD FU is called
Multiply-Accumulate (MAC) Unit

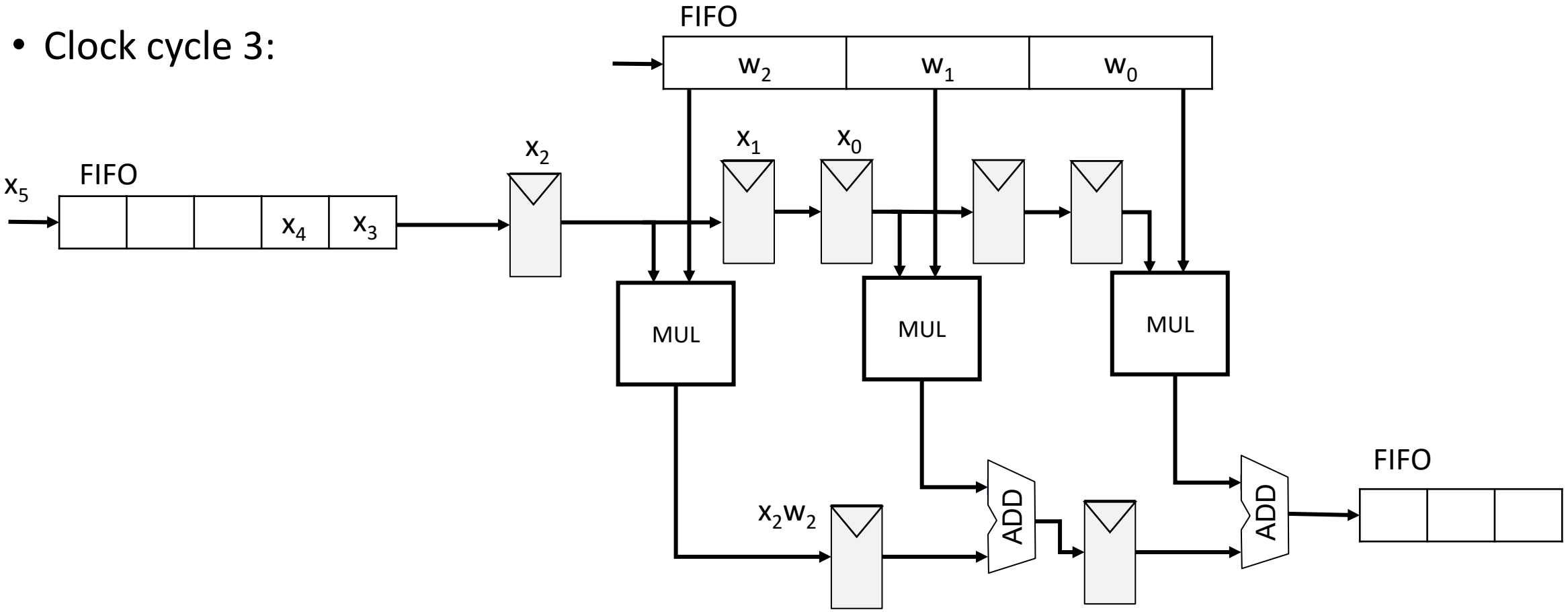
Example: 1D Convolution - Systolic Array (1D) - Structure

- Step 1: Load Weights



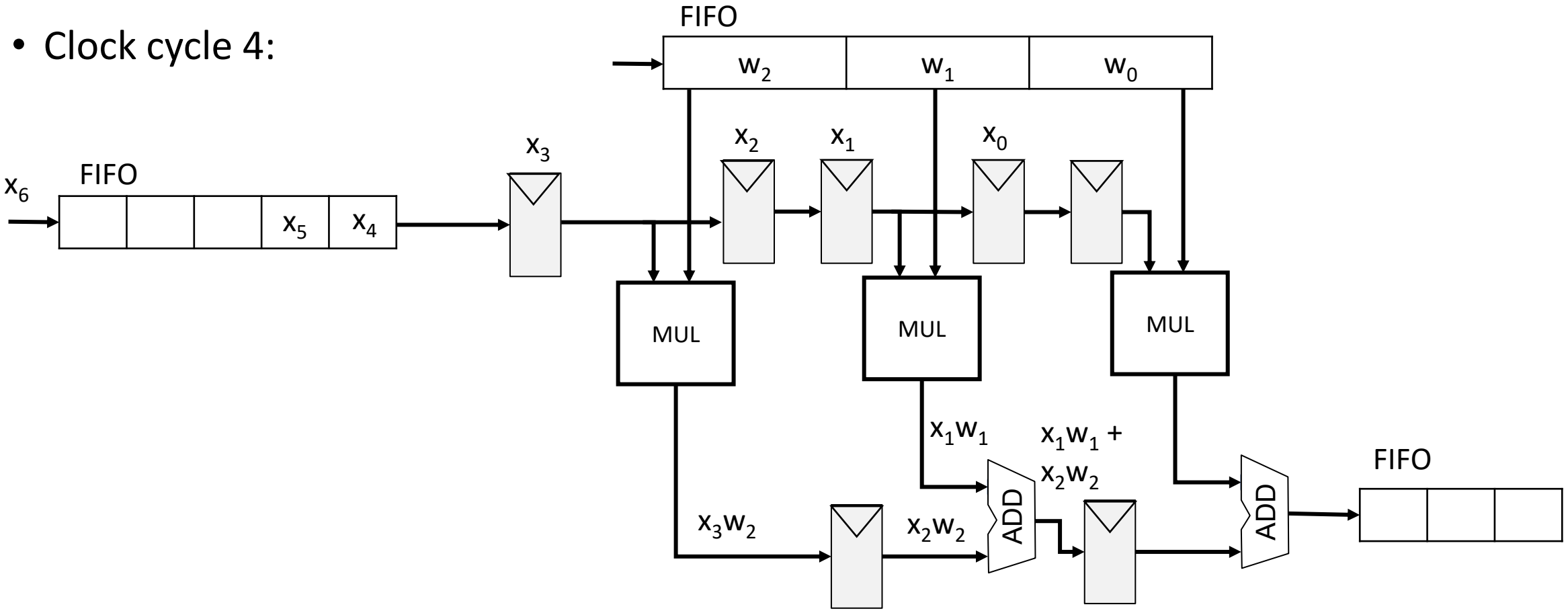
Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 3:



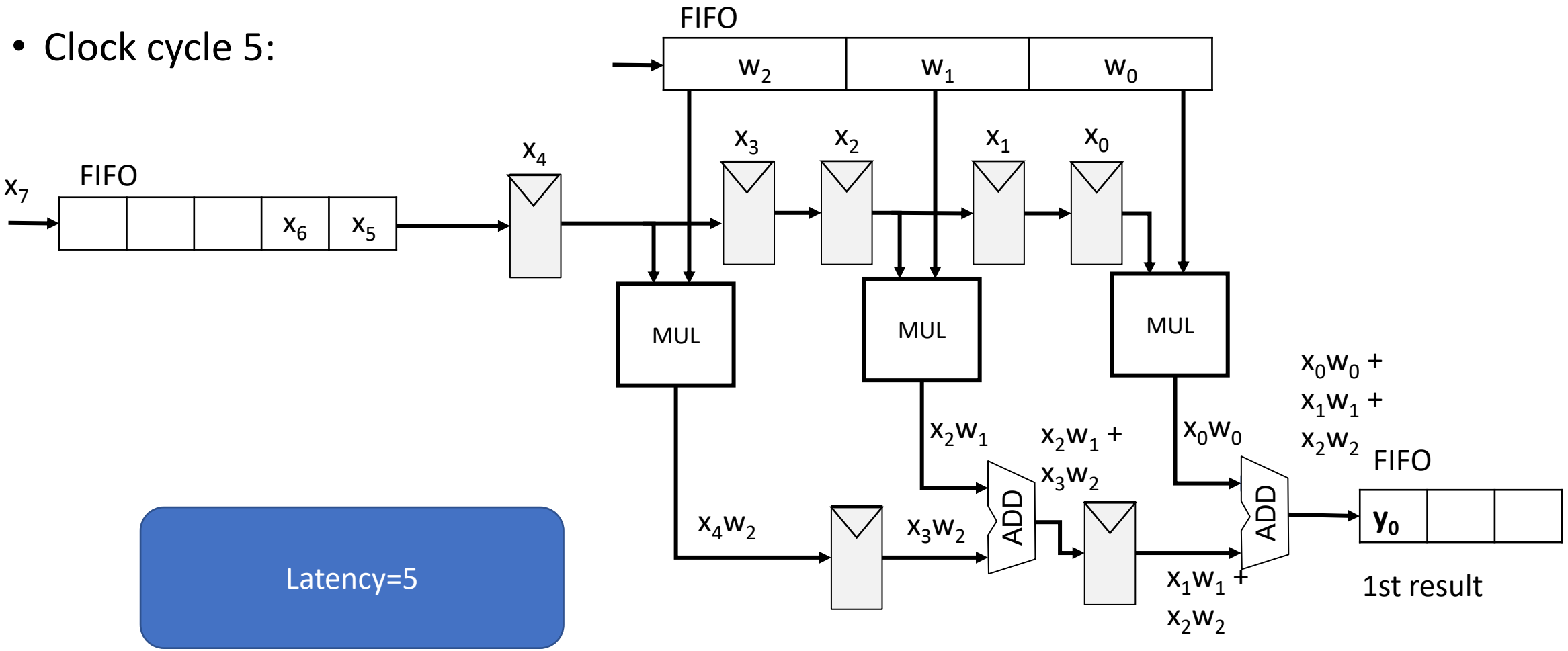
Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 4:



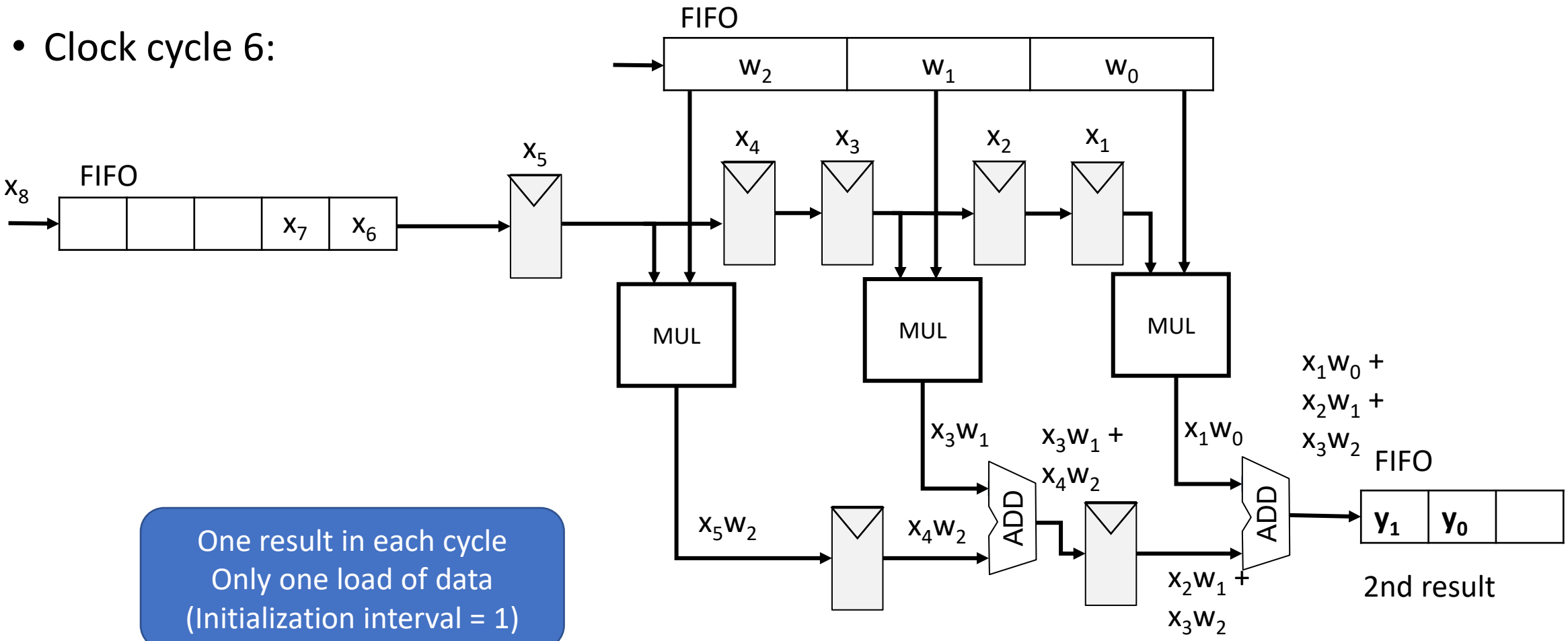
Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 5:



Example: 1D Convolution - Systolic Array (1D) - Structure

- Clock cycle 6:



One result in each cycle
Only one load of data
(Initialization interval = 1)

Systolic Arrays Pros-Cons

- Advantages:
  - Move intermediate results between FUs to reduce memory access
  - Balance between computation and memory bandwidth
  - Simple design to exploit **data-level parallelism** (DLP)

  - Different systolic arrays can be combined for multi-stage computations

- Disadvantage
  - Specialized: computation needs to fit FU arrangement

A look at Real ML Accelerators

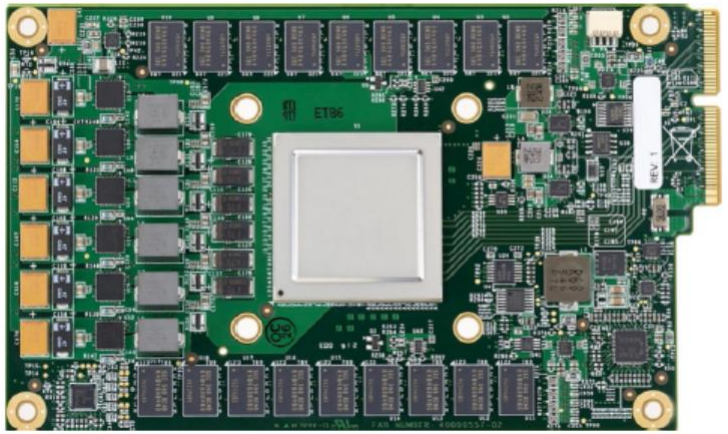
Optional, not relevant for exam

Google Tensor Processing Unit (TPU)

VTA Neural Processing Unit (NPU)

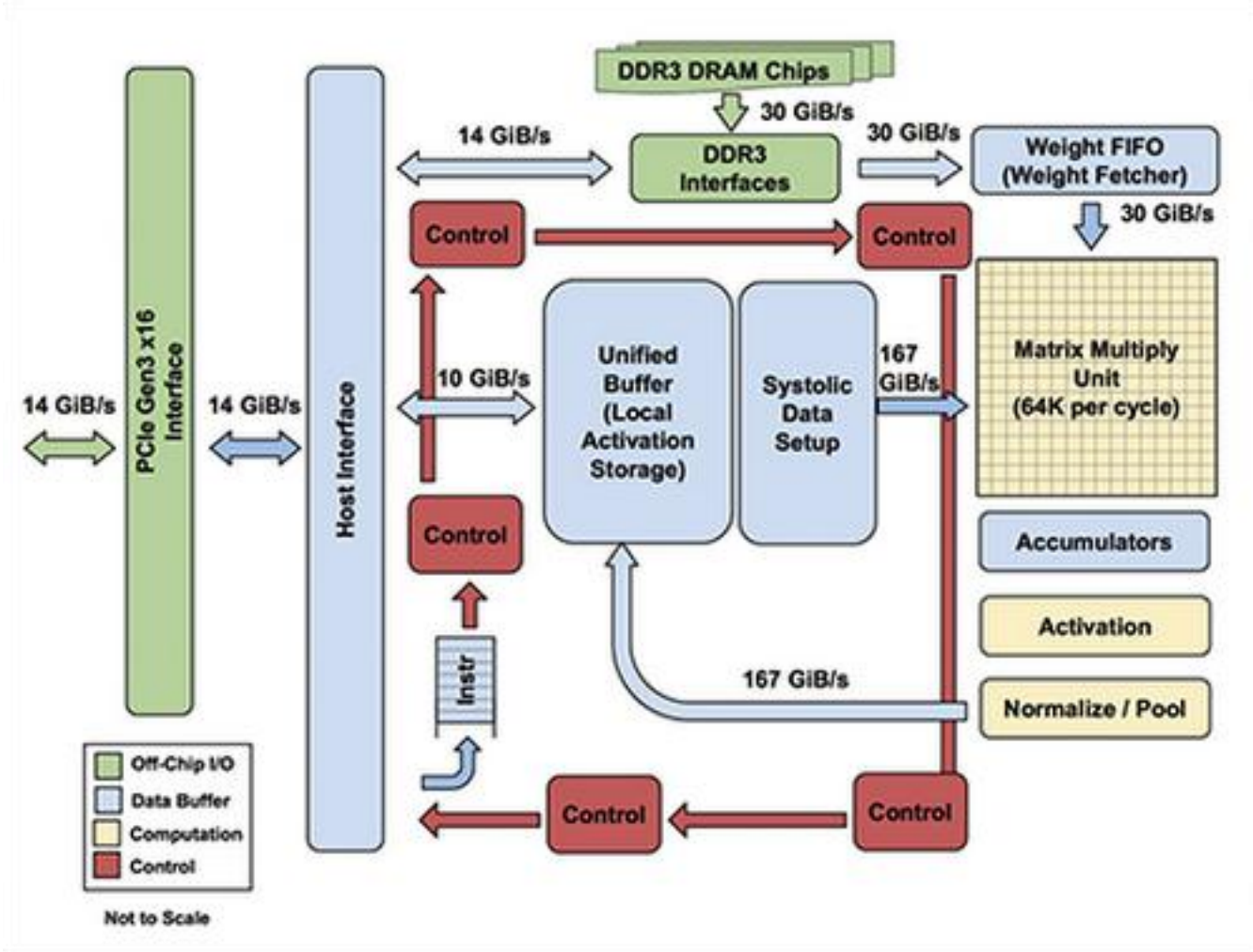
TPU V1: Tensor Processing Unit (2017)

- Application-specific Integrated Circuit (ASIC) – Chip from Google
- Specialized to accelerate Deep Neural Network (DNN) computations
- PCB board with PCIe Interface to Host processor



Source: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu?hl=en

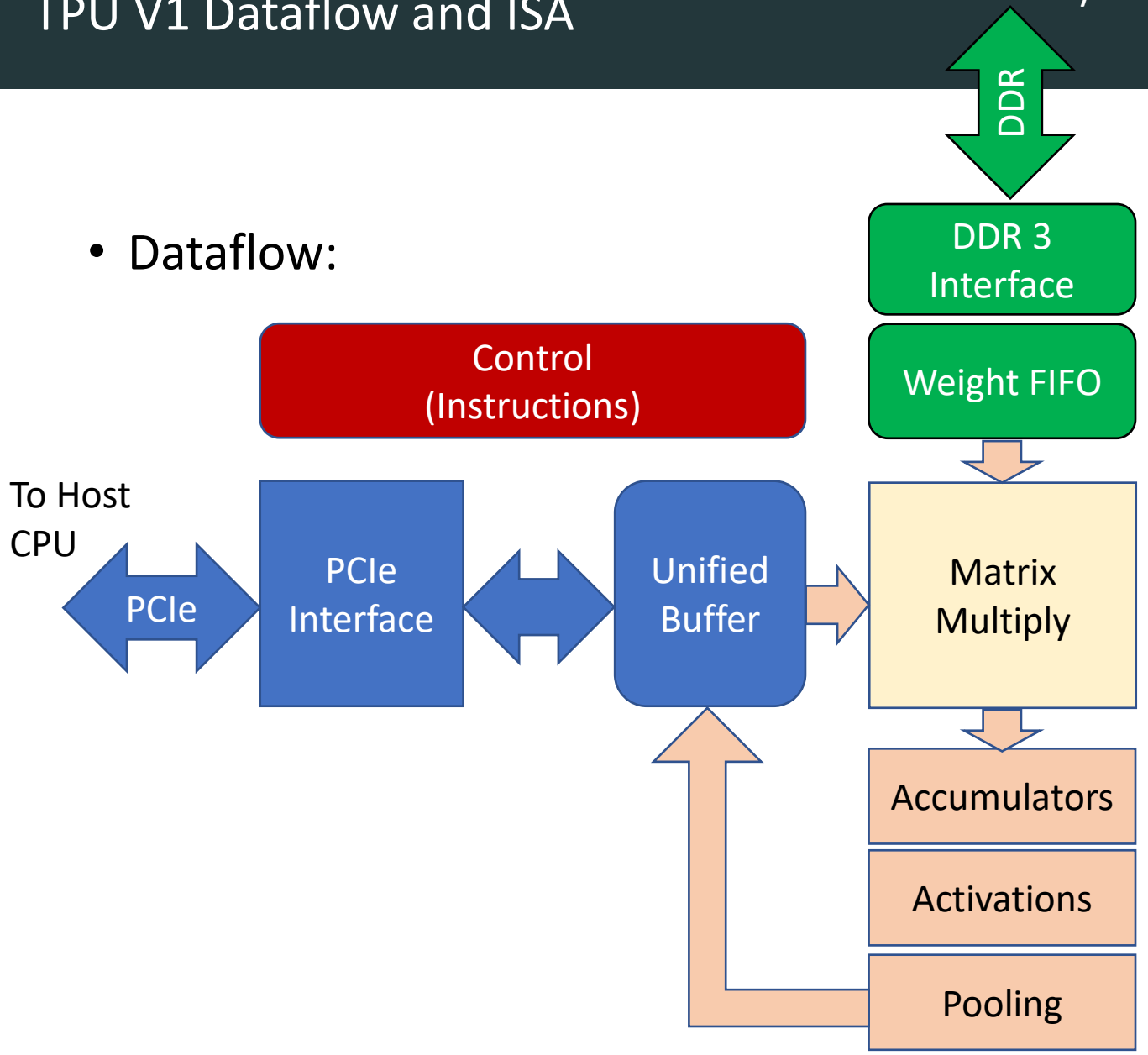
TPU Data Rates



Source: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu?hl=en

TPU V1 Dataflow and ISA

- Dataflow:

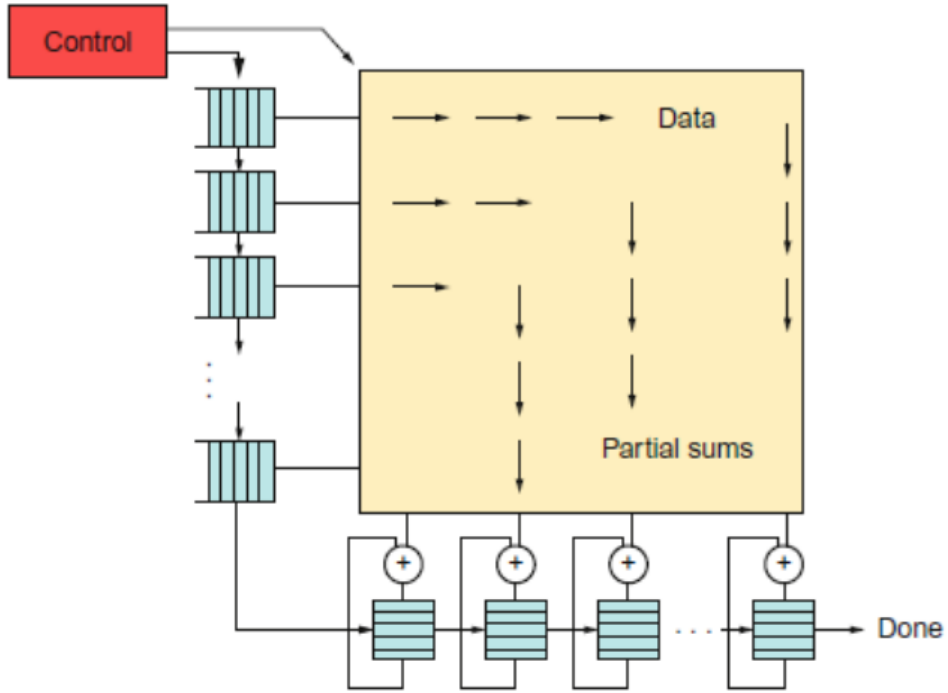


Instructions:

- Read Weights
  - Reads weights from the DDR into the Weight FIFO
- Read from Host Memory:
  - Reads data from the CPU (Host) memory into the unified TPU buffer
- Execute Matrix Matrix Multiply for Convolution + Activation + Pooling
- Write to Host Memory
  - Writes data from unified buffer into CPU memory

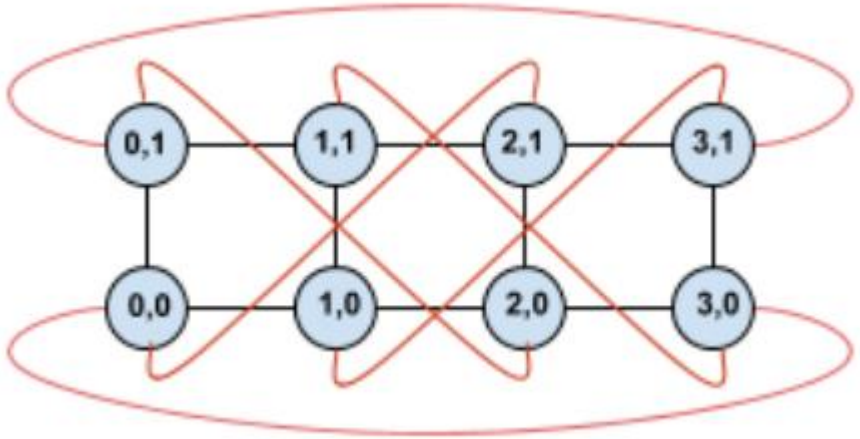
TPU: Matrix Matrix Multiply

- Core of the TPU is matrix-matrix-multiply
- 2D Systolic Array:
  - Input 1: Matrix size Sx256 (Unified buffer)
  - Input 2: Constant matrix 256x256 (Weight FIFO)
  - Output: Input1 multiplied Input 2
  - Latency: S cycles
  - Initialization interval: 1

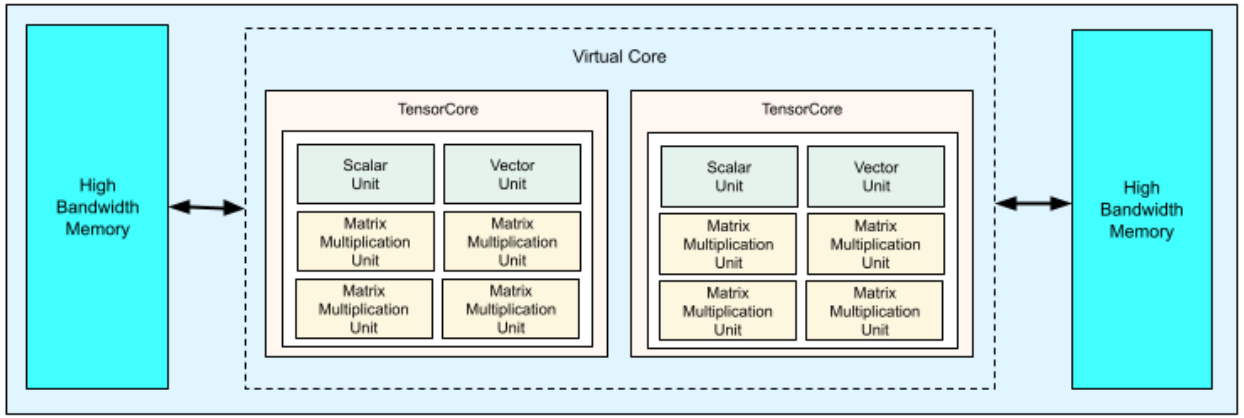


Google TPU V4 for Cloud

Key specifications	v4 Pod values
Peak compute per chip	275 teraflops (bf16 or int8)
HBM2 capacity and bandwidth	32 GiB, 1200 GBps
Measured min/mean/max power	90/170/192 W
TPU Pod size	4096 chips
Interconnect topology	3D mesh
Peak compute per Pod	1.1 exaflops (bf16 or int8)
All-reduce bandwidth per Pod	1.1 PB/s
Bisection bandwidth per Pod	24 TB/s

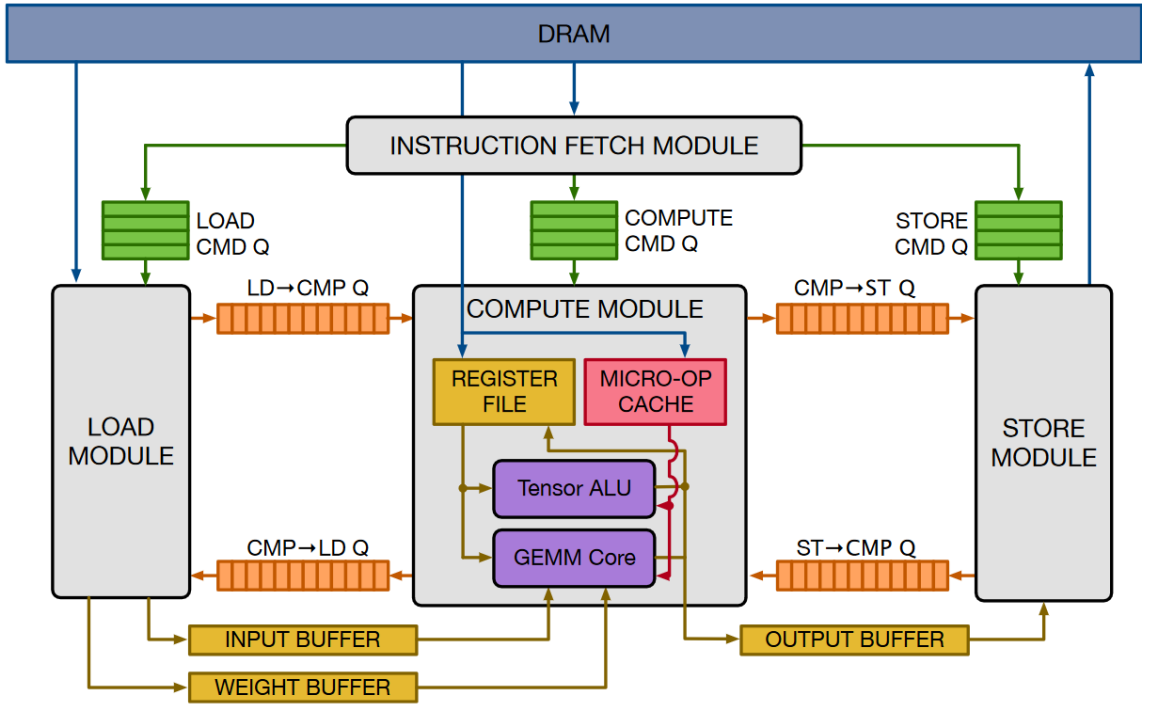


Chips can be arranged in Twisted Torus interconnect



Source: https://cloud.google.com/tpu/docs/v4

Embedded NPU: Versatile Tensor Accelerator (VTA)



- Source: http://arxiv.org/pdf/1807.04188
- Open Source: https://github.com/apache/tvm-vta

Summary

Covered Topics

- General-Purpose Processor Cores
  - Pipelining
  - Speculation and Branch Prediction
  - Instruction-Level Parallelism: Superscalar, VLIW
  - Thread-Level Parallelism: Multi-threading, Multi-Core
  - Data-Level Parallelism: Vector

- Specialized Cores :
  - GP-GPUs
  - Domain Accelerators: TPU, NPU
  - Application-specific Accelerators: HLS-generated

Thank you for your attention!