DL Optimizations for IoT Devices

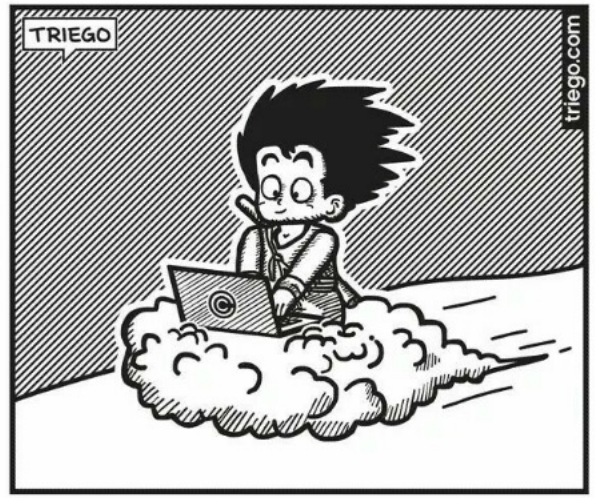
Introduction

Why Run Deep Learning on IoT Devices?

- A reminder…

- Machine Learning models are at the core of many emerging applications in the IoT world:
  - Smart city/district/factory/building/etc.
  - Bio-signals processing
  - Context-aware interactions (activity tracking, location-awareness, etc.)

- **But why don't we just run everything in the cloud?**



Why Run Deep Learning on IoT Devices?

- Sending raw data to the cloud and getting back the results is often sub-optimal

- **Reason #1: A lot of pressure on the network**

  - Imagine we want to process 224x224 videos from IoT cameras in real time at 30fps (1 frame every 30ms). We need to transmit 224x224x3 bytes to the cloud in 30ms, (plus 1 byte for the class index, but we can neglet it)

  - The upload bandwidth required (for a single camera) is (224x224x3x8)/0.03 = **40Mbps**

  - This is a **very low resolution** for todays' standards, and we could have **100s of cameras** in the same network.

Why Run Deep Learning on IoT Devices?

- Sending raw data to the cloud and getting back the results is often sub-optimal

- **Reason #2: High and unpredictable latency (low responsiveness)**

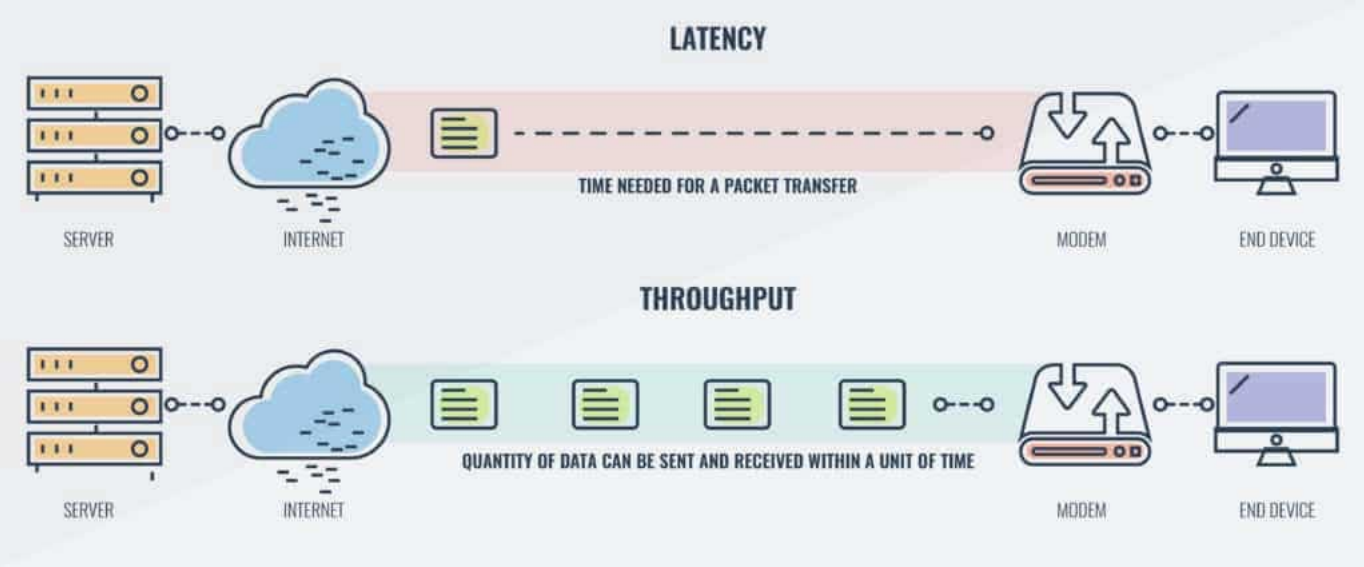
  - Connected with the previous one, but distinct.

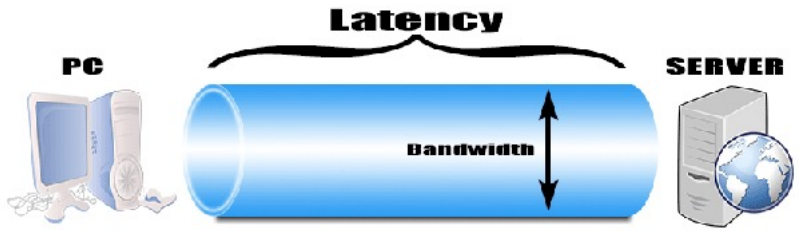
  - Wireless WAN links (4G, LoRA, etc.) have significant Round-Trip Times (RTTs). Even if the bandwidth is high, the time to get the first packet back is 10-100 of ms.

  - Wireless connections can be unstable or unavailable in some places.

Why Run Deep Learning on IoT Devices?

- The bandwidth issue is related to **scale** (it happens because you have 1000s or millions of connected devices)
- The latency problem is there even for a **single** device.





Why Run Deep Learning on IoT Devices?

- Sending raw data to the cloud and getting back the results is often sub-optimal

- **Reason #3: High energy consumption**

  - The energy required to perform computing tasks is constantly decreasing
    - Technology scaling, new types of devices, architectural improvements, software improvements

  - The energy required for transmitting data is not decreasing at the same pace

  - Transmitting large amounts of data over a wireless link from edge to cloud consumes **a lot of energy**. It is much more efficient to do at least part of the computation locally, so that the amount of data to transmit to the cloud is reduced (sort of compression).

Why Run Deep Learning on IoT Devices?

- Sending raw data to the cloud and getting back the results is often sub-optimal

- **Reason #4: Privacy**

  - Transmitting raw data to the cloud raises privacy concerns. Users must fully trust cloud providers for ensuring that their private data is stored and processed in a secure way.

  - This is true both for personal devices (audio from microphone, video from camera, etc.) but also, for example, for industrial assets (data collected by sensors in a plant). Companies are often reluctant to let important data leave their premises.

Why Not?

- **So, let's execute everything on edge devices...!**

- Different definitions of "edge"…
  1. **Edge "gateway":** embedded GPUs or CPUs (Cortex-A class), like the one present in the Raspberry Pi used in the labs
  2. **End-node (sensor):** typically microcontroller-based systems (Cortex-M class)

- In both cases, the hardware specs are orders of magnitude different from those available in the cloud.

Why Not?

Metric	Single cloud server	Edge gateway	Sensor
Number of compute units (cores)	50 (CPU cores) 5000 (GPU cores)	1-4 (CPU cores) 200 (GPU cores)	1 (CPU core) **No GPU**
Operating Frequency	GHz	100s of MHz	kHz/MHz
Memory (main)	100s of GBs	GBs	**MBs**
Memory (mass)	100s of TBs	GBs	**Absent / kBs / MBs**
Power consumption	KWs	10s of W	**mW/W (active)**
Cost	10000 €	50 €	**1 €**

Why Not?

- Large ML models (especially deep NNs) simply cannot be executed "as is"

Metric	Single cloud server	Edge gateway	Sensor
Number of compute units (cores)	50 (CPU cores) 5000 (GPU cores)	1-4 (CPU cores) 200 (GPU cores)	1 (CPU core) **No GPU**
Operating Frequency	GHz	100s of MHz	kHz/MHz
Memory (main)	100s of GBs	GBs	**MBs**
Memory (mass)	100s of TBs	GBs	**Absent / kBs / MBs**
Power consumption	KWs	10s of W	**mW/W (active)**
Cost	10000 €	50 €	

At 1GHz and assuming 1 MAC/cycle (very optimistic): **20s per inference**

Model	Parameters (Memory)	Floating Point Operations - FLOP
VGGNet	**138M (552MB)**	**19.6B**
ResNet - 152	**60M (240MB)**	**11B**

Parameters **only**!
No inputs/activations, no inference engine, no operating system, etc.

Goal

- To summarize, running ML/DL-based **inference** (…or training) directly on edge devices may provide benefits in terms of latency, scalability, energy consumption, and privacy.
- But doing so is not easy! Mainly because of the limited hardware capabilities of these devices, which are determined by **cost** and **energy efficiency** reasons.
- So, we need specific techniques to optimize the inference **memory occupation**, **execution time** and **energy consumption** on cost- and energy-constrained IoT devices.
- In the rest of this part of the course, we'll overview some of these techniques

Not Just for Edge Devices…

- Cloud devices are powerful, but serve millions of requests.
- Many of the optimizations analyzed are beneficial also to make cloud inference lighter (faster, more energy efficienct, etc.)
- Some of them can also be used to speed-up training…

DL Optimizations for IoT Devices

Computational Aspects of DL Models

Computational Aspects of DL Models

- Let's analyze a DL model from the point of view of **computations**
- We'll focus only on the computations needed for inference (i.e. the forward pass), but similar analyses can also be done for training.
- We'll focus on **fully-connected (dense)** and **convolutional** layers
  - RNN cells can be analyzed similarly
  - Other layers (Pooling, BatchNorm, etc) are less relevant from a computational p.o.v. and can be optionally "fused" with dense/conv.

DL Optimizations for IoT Devices

Fully Connected Layers

Fully Connected Layer

- Fully-connected (Dense) layer: $b = f(W \cdot a + k)$
  - a = input activations
  - b = output activations
  - W = layer weights
  - k = layer biases (optional)
  - $f()$ = activation function (sigmoid, tanh, ReLU, etc.)

[Source] Benini, Micropower Deep Learning for Cognitive Cyberphysical Systems

Fully Connected Layer

- Fully-connected (Dense) layer: $b = f(W \cdot a + k)$

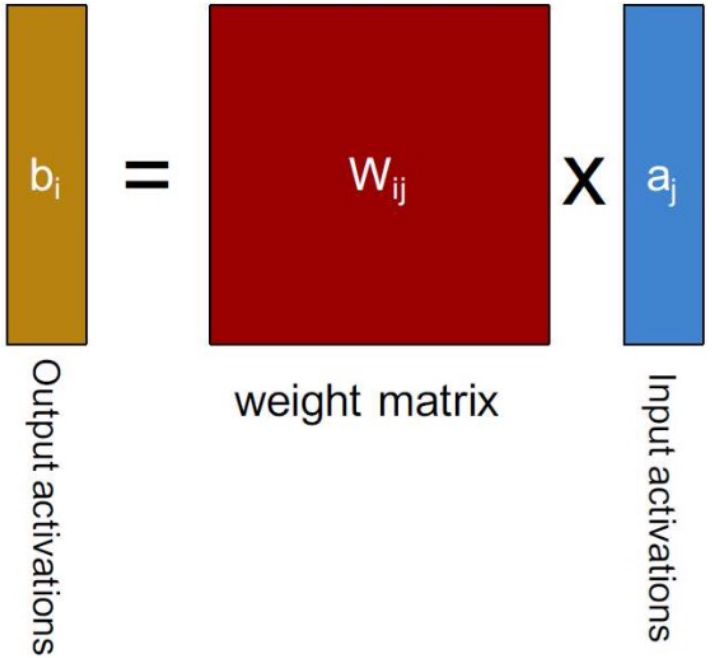
Most critical Part,
Millions of ops per layer

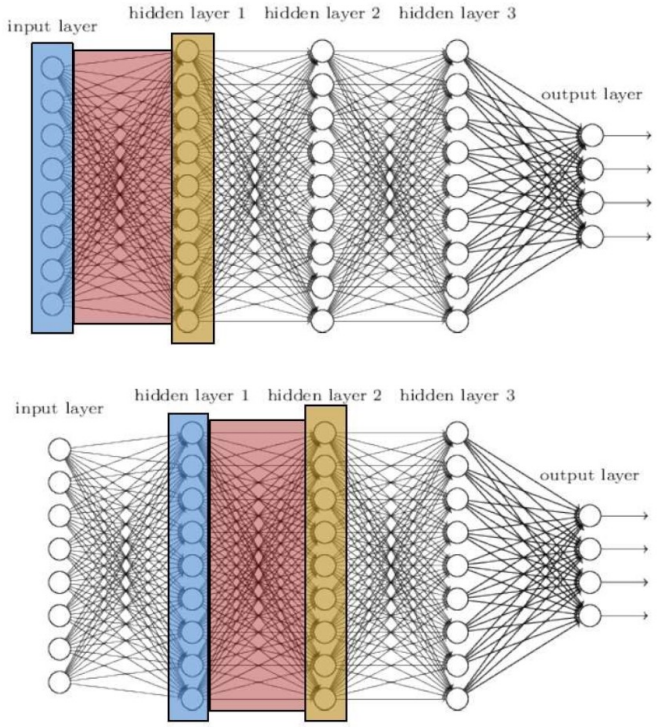
  - $W \cdot a$ requires $O(NM)$ multiply and accumulate (MAC) operations, where N is the number of input activations and M the number of output activations.
  - $\ldots + k$ requires $O(M)$ sums
  - $f(\ldots)$ requires $O(M)$ function evaluations, implemented either with software approximations, or table look-ups, etc.

[Source] Benini, Micropower Deep Learning for Cognitive Cyberphysical Systems

Fully Connected Layer

- Fully-connected (Dense) layer: $b = f(W \cdot a + k)$
  - The key operation is a **dense** Matrix-Vector (MxV) multiplication

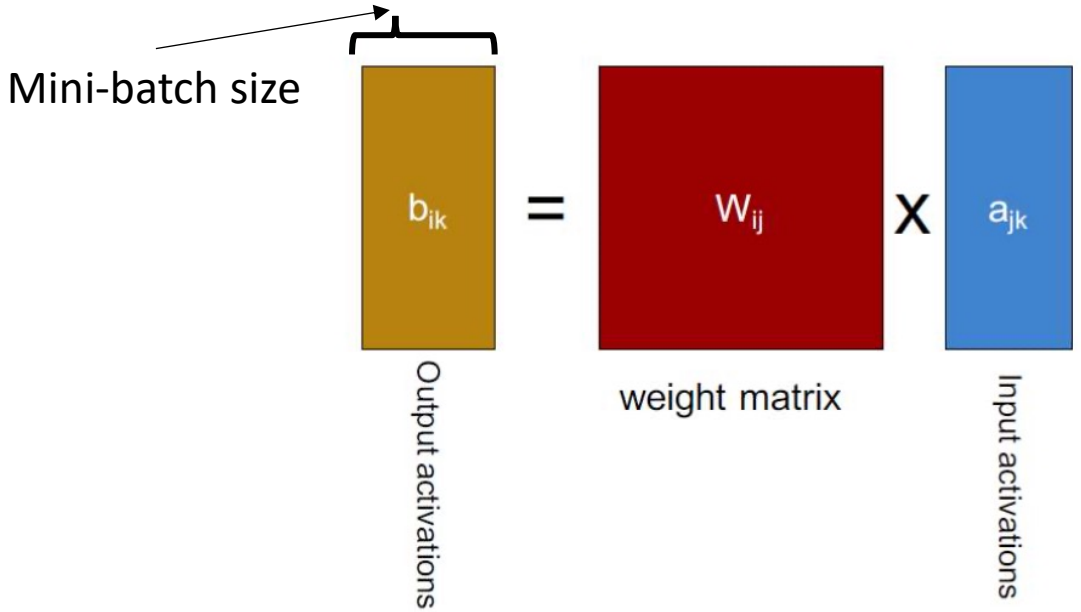




[Source] Benini, Micropower Deep Learning for Cognitive Cyberphysical Systems

Fully Connected Layer

- Batching enables a higher degree of **weight reuse**
  - More than one activation multiplied with the same weight



[Source] Benini, Micropower Deep Learning for Cognitive Cyberphysical Systems

Operational Intensity and the Roofline Model

- We want to maximize weight reuse (or any possible data reuse) because fully-connected layers are **strongly memory bound.** The overall processing speed is not limited by the compute speed, but rather by the transfer of data from/to the memory to/from the CPU.

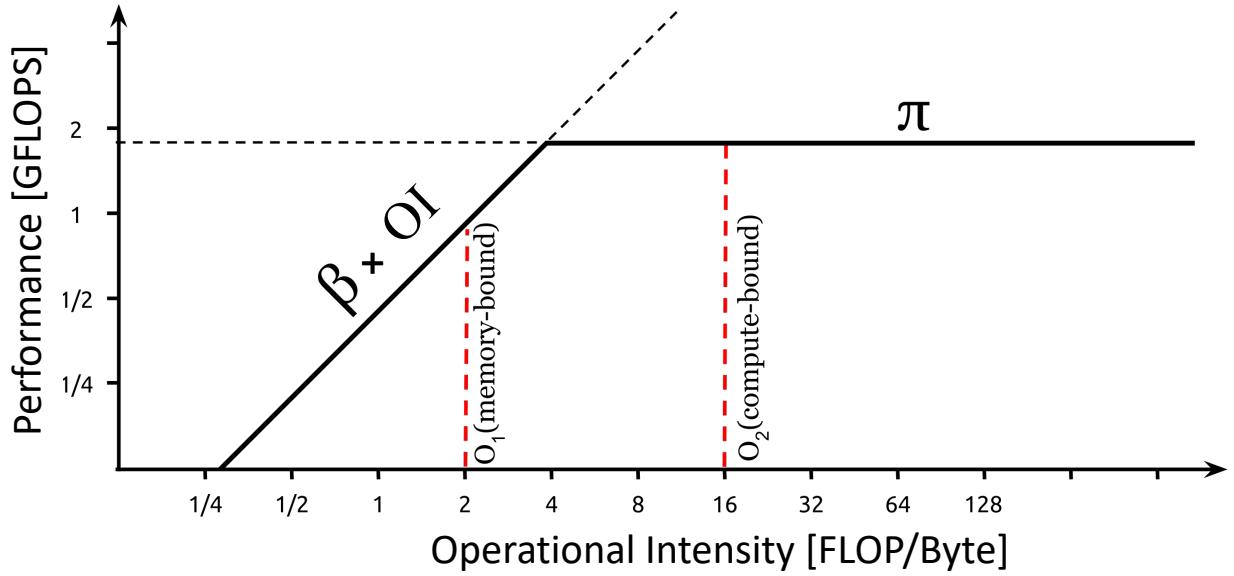
- This can be visualized using the so-called "**roofline model**", which shows the performance bound of an application as a function of its **operational (or arithmetic) intensity (OI)**

$$OI = \frac{N.of\ Operations}{Bytes\ Transferred}\ [\frac{FLOP}{Byte}]$$

- The OI is a characteristic of a given implementation of a computing task. Different implementations of the same computational "kernel" have different OIs in general.

Operational Intensity and the Roofline Model

- The **roofline model** provides an upper bound to the performance of a task based on the physical characteristics of the hardware, in particular its peak performance and peak memory bandwidth. Depending on the OI, we can easily visualize what is the limiting factor for performance, for a particular implementation.

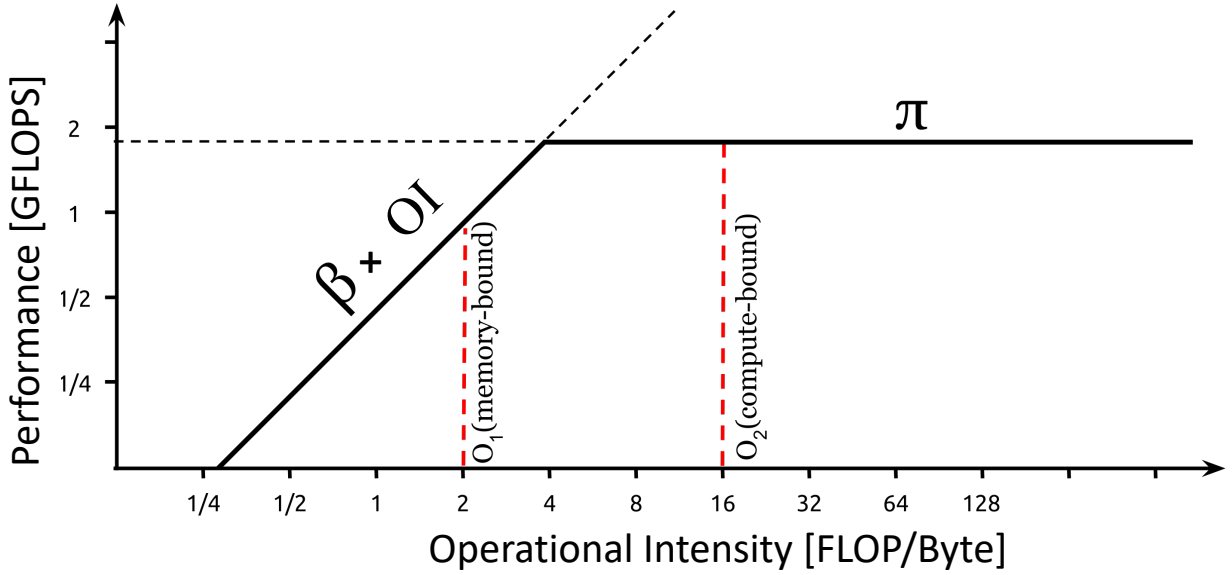


π = peak performance [FLOP/s or GFLOPS – Giga FLOP per Second]

β = peak bandwidth [Byte/s]

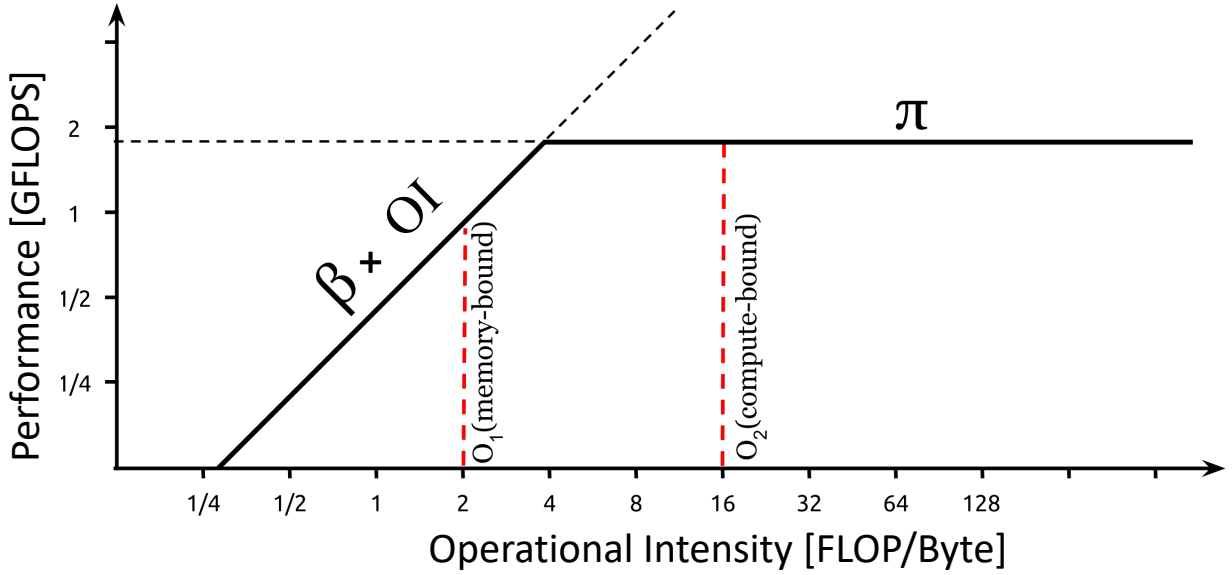
Operational Intensity and the Roofline Model

- **Example:** Assume a system that can perform 2 GFLOPS and can load data from memory at around 0.5GB/s.



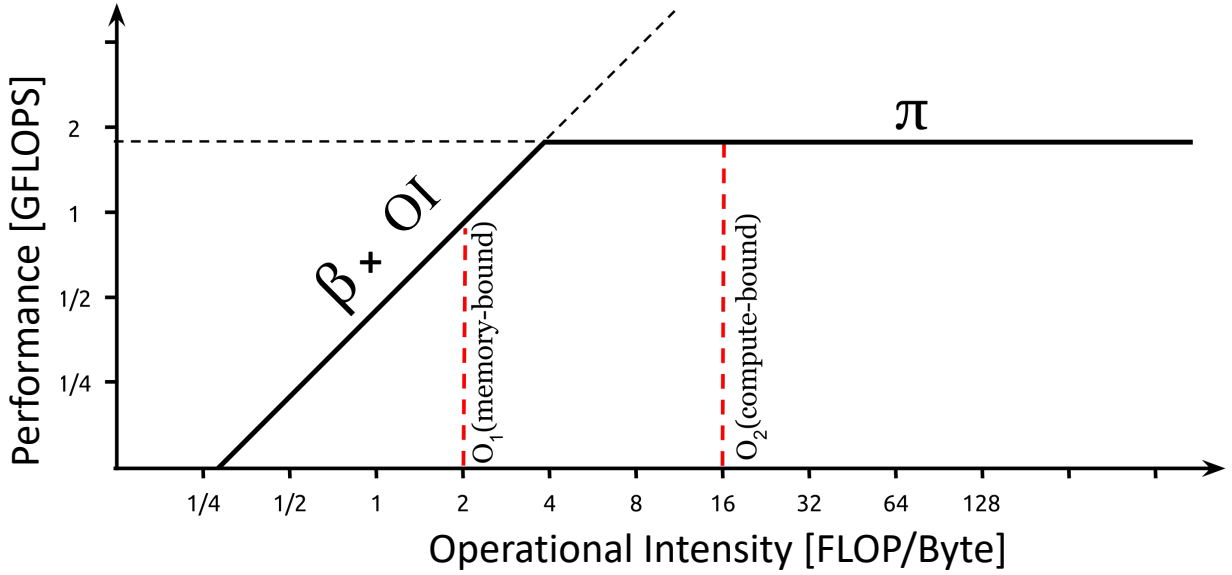
Operational Intensity and the Roofline Model

- **Case 1:** The target computation performs 2 FLOPs for each new loaded byte of data.
- We have OI = O_1 = 2, i.e. the leftmost red line
- We are on the left side of the roofline model, which means that our performance will be limited by memory. Intuitively, since we can only load data at 0.5GB/s, and we need 1 new byte every 2 operations, we can perform at max 2*0.5 = 1 GFLOP per second.

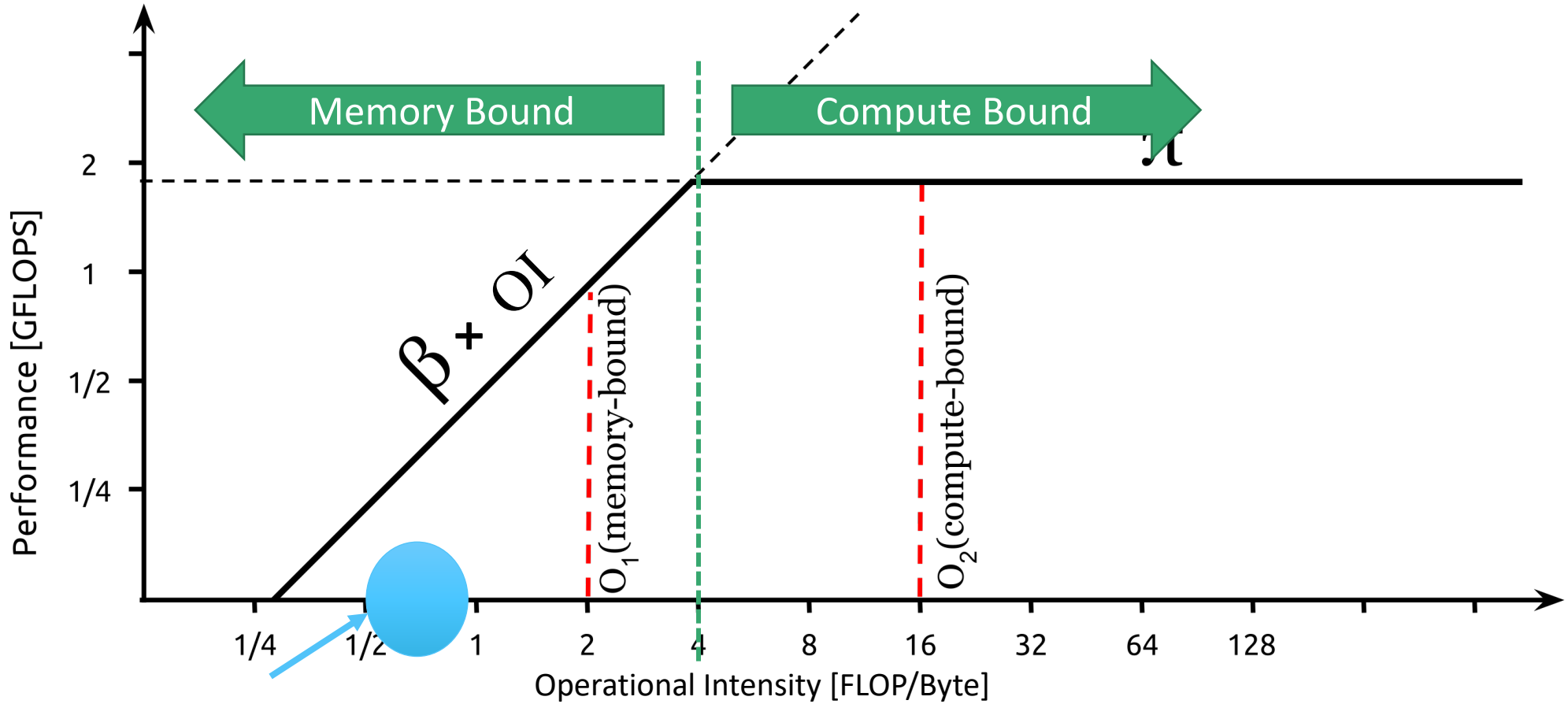


Operational Intensity and the Roofline Model

- **Case 2:** The target computation performs 16 FLOPs for each new loaded byte of data.
- We have OI = O_2 = 16, i.e. the rightmost red line
- We are on the right side of the roofline model, which means that our performance will be limited by compute. Intuitively, we can load all the data needed to keep the ALUs of our system constantly active.



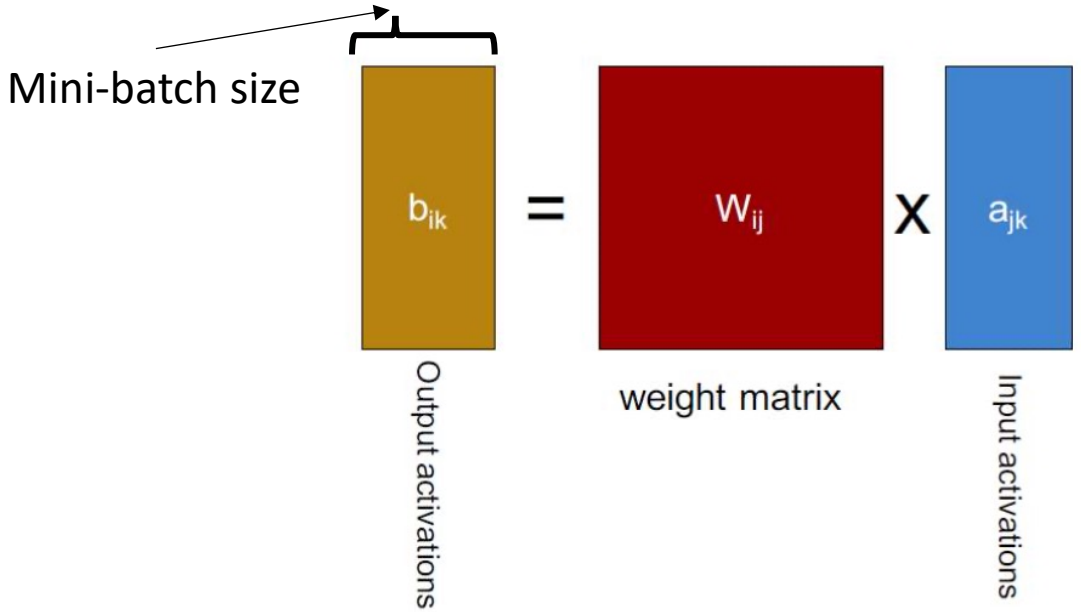
Operational Intensity and the Roofline Model



Fully connected layers inference (typically)

Fully Connected Layer

- Back to batching… let's assume weights and activations are stored as 32-bit floats (4 bytes each).



[Source] Benini, Micropower Deep Learning for Cognitive Cyberphysical Systems

Fully Connected Layer

- Without batching

```
foreach row i of the weight matrix {
        b(i) = 0
        foreach element j in the i-th row {
                Load w(i,j)
                Load a(j)
                b(i) = b(i) +  w(i,j) * a(j)
        }
        Store b(i)
}
```

- 2 loads, 1 MAC: **OI = 1/(2*4) = 0.125**
  - Store operations are less frequent, neglected

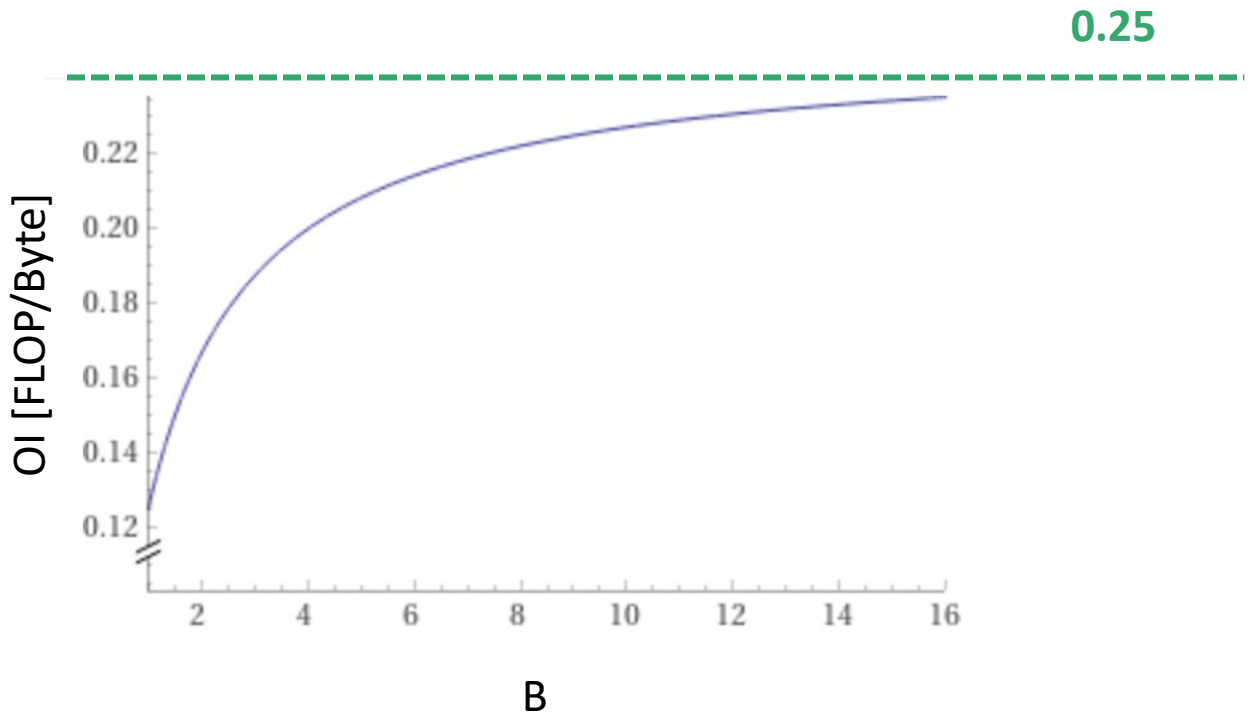
- With a minibatch of 4

```
foreach row i of the weight matrix {
        b(i,0) = b(i,1) = b(i,2) = b(i,3) = 0
        foreach element j in the i-th row {
                Load w(i,j)
                Load a(j,0), a(j,1), a(j, 2), a(j, 3)
                b(i,0) = b(i,0) +  w(i,j) * a(j,0)
                b(i,1) = b(i,1) +  w(i,j) * a(j,1)
                b(i,2) = b(i,2) +  w(i,j) * a(j,2)
                b(i,3) = b(i,3) +  w(i,j) * a(j,3)
        }
        Store b(i,0), b(i,1), b(i, 2), b(i,3) in memory
}
```

- 5 loads, 4 MACs: **OI = 4/(5*4) = 0.2**

Fully Connected Layer

- OI vs minibatch size (B) for this implementation: $OI = B/((B+1)*4)$



Fully Connected Layer

- Some cloud numbers (NVIDIA Titan RTX GPU):
  - 1000s of compute units performing MACs in parallel. Peak performance: 130 TFLOPS
  - Memory bandwidth peak: 670 GB/s
  - Maximum performance due to memory: 670 * 0.25 = 167.5 GFLOPS **(*)**

- Some edge numbers (Raspberry Pi 4, source):
  - Peak performance for a similar linear algebra routine (Linpack): 2GFLOPS
  - Memory bandwidth peak: 4GB/s
  - Maximum performance due to memory: 4 * 0.25 = 1 GFLOPS

- **In both cases, we are memory bound!**

(*) Disclaimer: only true for this particular layer implementation, which is sub-optimal (see next slide). We don't want NVIDIA to sue us!

Fully Connected Layer

- **NB:** The one in the previous slides is a naïve implementation. The OI can be increased beyond 0.25 by extending the data reuse concept further. For example, keeping B=4, we can load 2 weights at the same time and perform 8 MACs with 6 loads

```
…
foreach element j in the i-th row {
    Load w(i,j), w(i+1, j)
    Load a(j,0), a(j,1), a(j, 2), a(j, 3)
    …
}
```

Fully Connected Layer

- The theoretical OI limit tents to B, and would be obtained if we could keep all data in internal registers:

$$OI \leq \frac{Min.\ number\ of\ ops}{Min.\ number\ of\ loads + stores} = \frac{MN * B}{MN + (M + N) * B}$$

- But in practice, at the edge this is **not even remotely possible!!!!**
  - It is (sort-of) possible in high-end devices such as the TPU

Output stores

Input loads

Weight loads

Batching During Inference

- After all this discussion, **can we really do batching during inference?**
- **It depends!!**

- In general, yes, we can gather multiple inputs and process them (e.g. classify them) all together.
  - Especially of interest for cloud devices or edge gateways, which may gather inputs from multiple sources (e.g. sensors).

- However, batching is **not possible in latency-sensitive applications,** where it is important to obtain the results as quickly as possible when the inputs become available.
  - Examples: autonomous driving, real-time translation, etc.

- **Many IoT applications are latency-sensitive, so batching is seldom possible in practice!!**

Data Reuse And Energy

- Until now, we discussed the benefits of data reuse for performance (roofline model, etc.). What about energy?

- Each access to main memory **consumes more than 100x more energy** compared to an instruction using only register operands.

- Data reuse is fundamental also for energy efficiency.

Energy per operation [nJ]

[Source] Yakun Sophia Shao and David Brooks. 2013. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor (*ISLPED '13*).

	Scalar Op
Register	0.45
L1	0.88
L2	7.72
Mem w/ Prefetch	52.14
Mem w/o Prefetch	232.62
Write to Mem	62.14

Data Reuse And Energy

- Another example:

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**

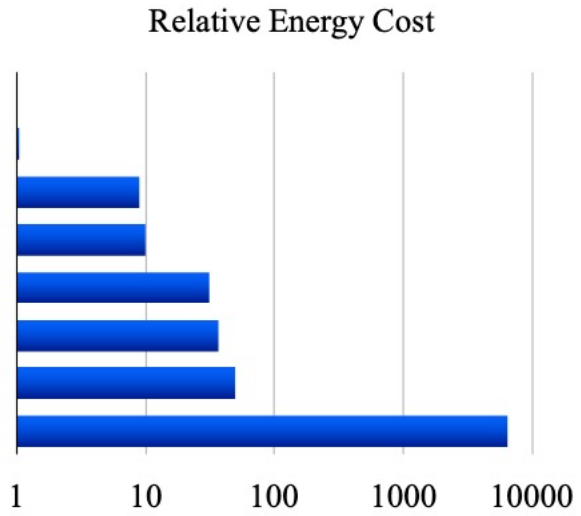


Figure 1: Energy table for 45nm CMOS process [7]. Memory access is 3 orders of magnitude more energy expensive than simple arithmetic.

[Source] Han et al, Learning both Weights and Connections for Efficient Neural Networks

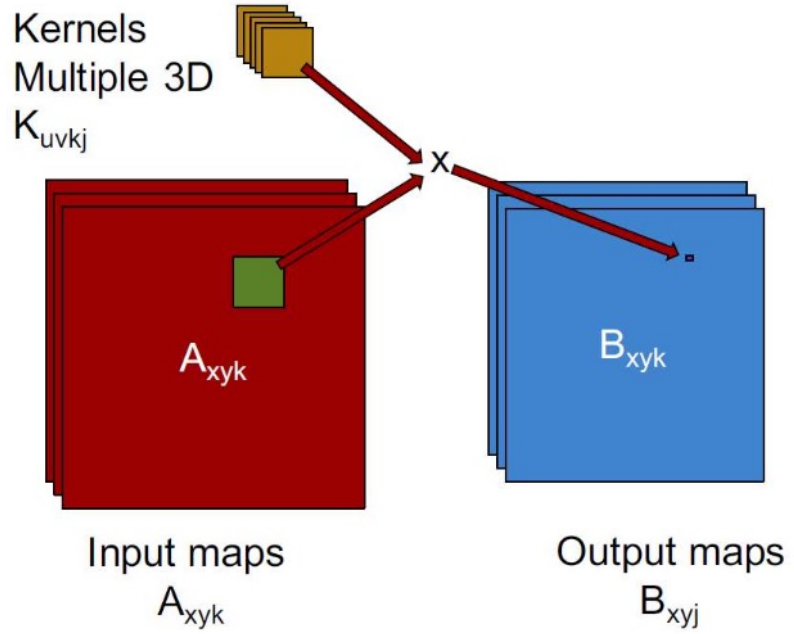
DL Optimizations for IoT Devices

Convolutional Layers

Convolutional Layer

- Convolutional Layer: $B_{xyj} = f(\sum_{u=0}^{K}\sum_{v=0}^{K}\sum_{k=0}^{C_{in}} K_{u,v,k,j} A_{x-u,y-v,k} + b_j)$
- Again, the critical operation is the sum of products between inputs and weights
- Naïve implementation, 6 indented loops:

```
1 For each output map j
1   For each input map k
2     For each pixel x,y
2       For each kernel element u,v
          B_xyj += A_(x-u)(y-v)k x K_uvkj
```



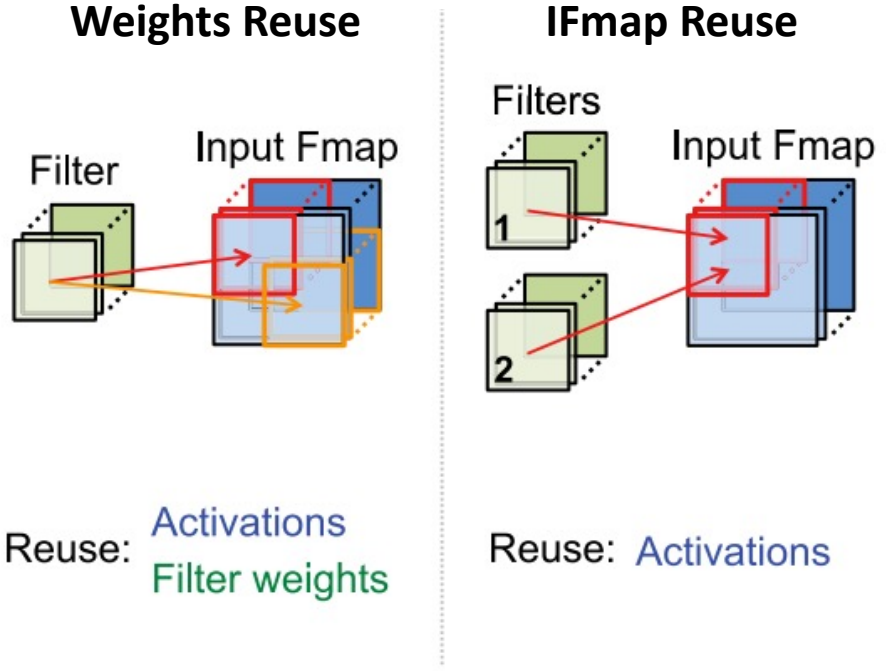
[Source] Benini, Micropower Deep Learning for Cognitive Cyberphysical Systems

Convolutional Layer

- Convolutional Layer: $B_{xyj} = f(\sum_{u=0}^{K}\sum_{v=0}^{K}\sum_{k=0}^{C_{in}} K_{u,v,k,j} A_{x-u,y-v,k} + b_j)$
- Actually, in some platforms (e.g. MCUs) convolutions are converted to MxV with a process called “im2col”.
- Convolution weights are “shared” by all pixels in the input. Higher weight reuse possible.
- With respect to Dense layers, Conv layers tend to be more compute bound, even without batching.

Convolutional Layer

- Conv layers enable multiple data reuse patterns (even without considering batching). These are some basic ones:

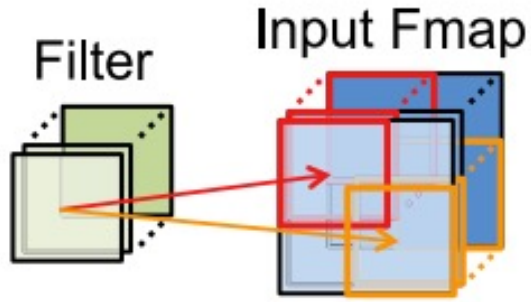


[Source] Sze et al, Efficient Processing of Deep Neural Networks: A Tutorial and Survey

Convolutional Layer

- Weights Reuse keeps the weights stationary and “slides” over the input Feature map (Fmap)

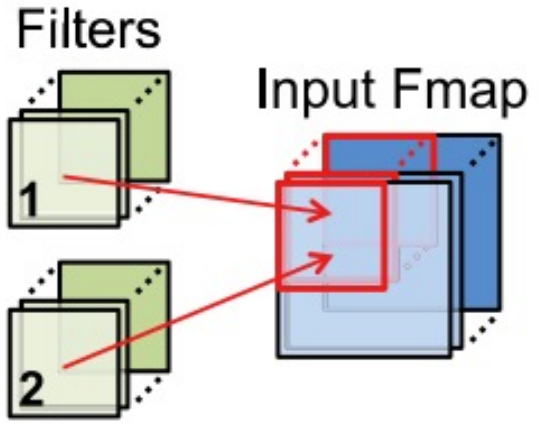
- It reuses both filter weights and part of the input patches
  - Only load one new “partial column” every time



Reuse: Activations
Filter weights

Convolutional Layer

- IFmap Reuse loads each Input Fmap patch only once, and applies all filter weights to it

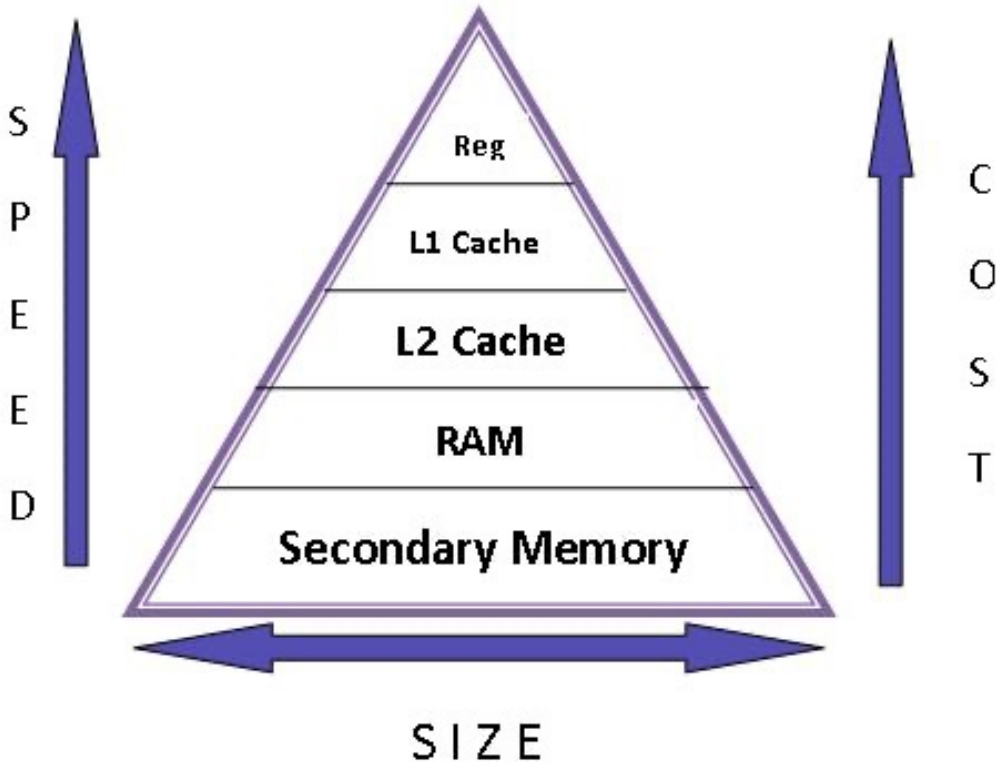


Reuse: Activations

Convolutional Layer

- The best choice between **Weights Reuse** and **Ifmap Reuse** depends on:
  - The features of the underlying hardware
  - The parameters of the layer

- In reality, data reuse is applied **hierarchically** at multiple levels.
  - From main memory to (multiple levels of) caches
  - From caches to internal registers



Convolutional Layer

- At the innermost level (registers) the amount of data that can be kept is very small (e.g. 32/64 registers in a MCU).
- So, reuse patterns become more complex…
- Clearly, when possible, batching remains an option to increase weight reuse.

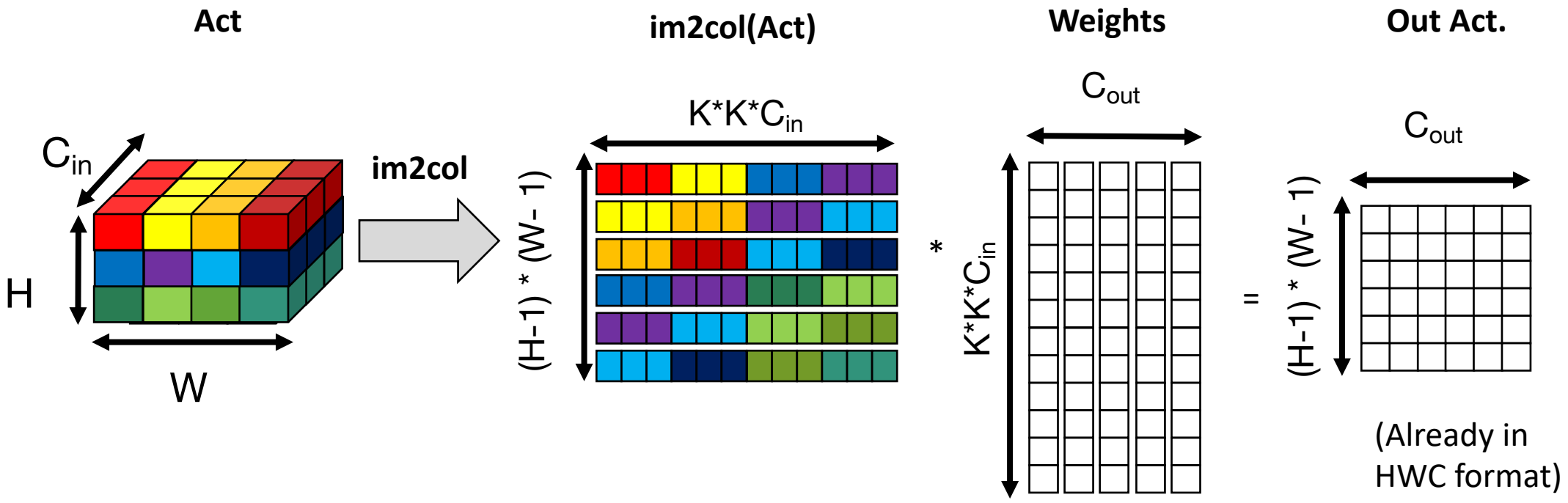
Conv Implementation Example (im2col + GEMM)

- Conv. Layers implementations typically reduce it to a MxM (GEMM)
  - Conversion from 3D tensor to 2D matrix via a process called “im2col” (or “im2row”)
  - Leverage extremely optimized linear algebra routines for GEMM for CPU/GPU (BLAS/cuBLAS)

- Data layout in memory is key. Two choices:
  - **(N)CHW or Channels-first:** faster on GPUs (historically)
  - **(N)HWC or Channels-last:** tipically faster on MCUs
    - Faster im2col memcpy, with SIMD memory instructions

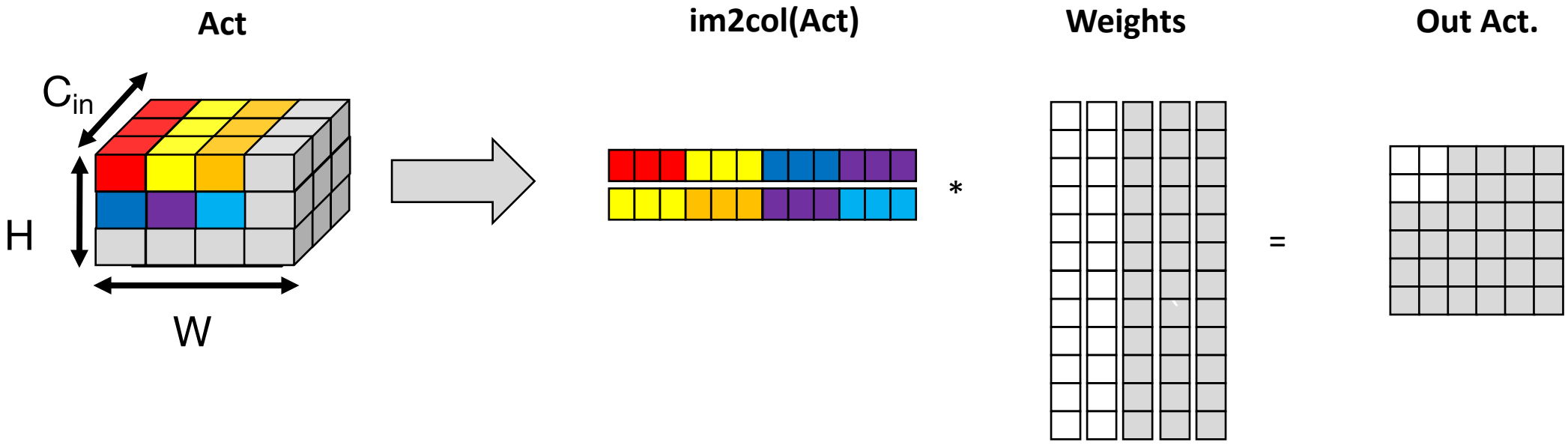
Conv Implementation Example (im2col + GEMM)

- Im2col + GEMM (for HWC format).
  - Example: 2x2 kernel (K=2), C_{in} = 3, C_{out} = 5, stride S=1, dilation D=1, no padding



Conv Implementation Example (im2col + GEMM)

- Practical implementation on MCUs (e.g., CMSIS-NN): **partial im2col**
  - Also called PDOT (panel dot product) microkernel. Used also for FC layers
  - Process a subset of “rows” (e.g., 1, 2 or 4) and a subset of output channels in each iteration
  - Reduce the memory overhead (buffer size)
  - Balance **weights/act reuse** with memory overhead and registers occupation.



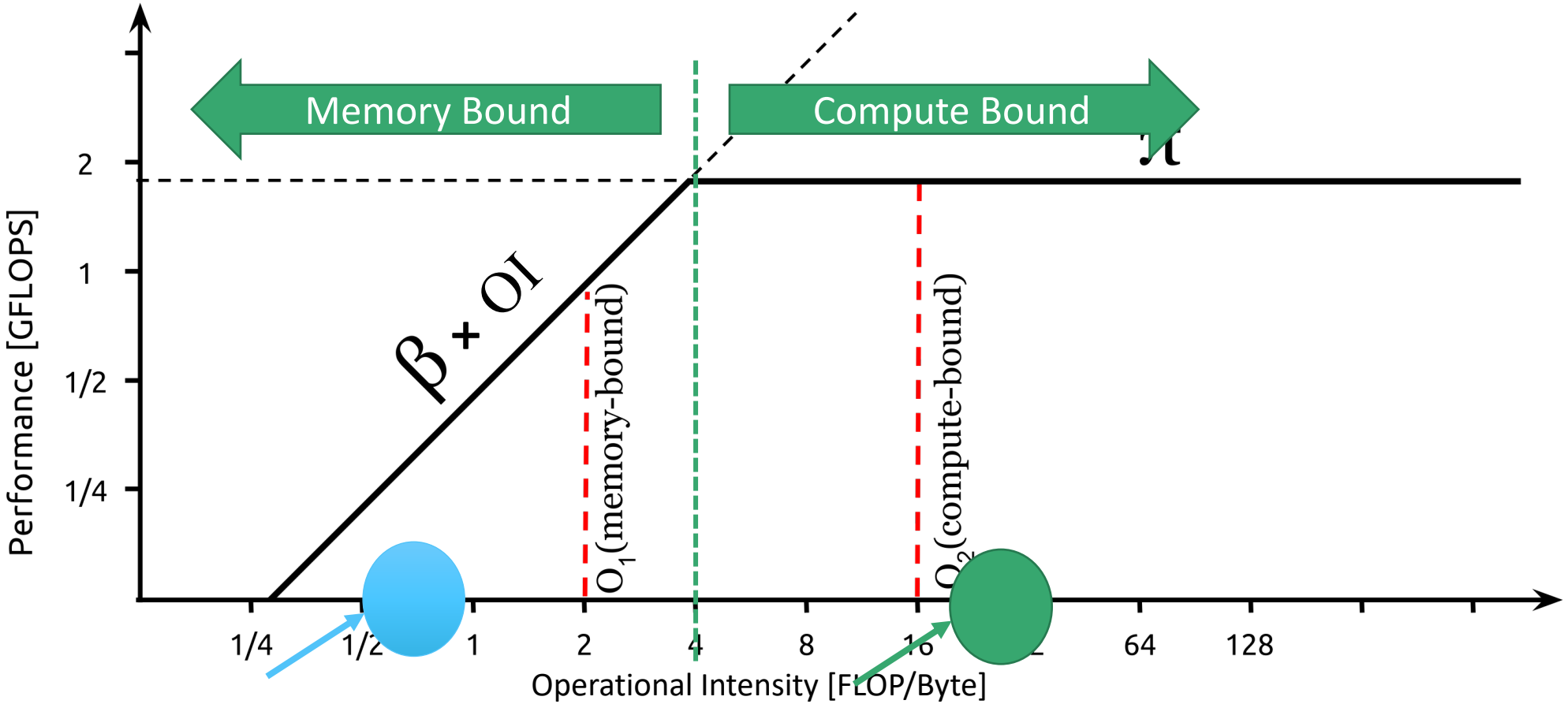
Conv Implementation Example (im2col + GEMM)

- Pseudo-code for a single input location:

```
//Im2col Loop
for ix in [x-K/2, x+K/2]:
  for iy in [y-K/2,  y+K/2]:
    for ci in Cin:
      x_i2c[i] = x[(ix + W*iy)*Cin + ci]
      i++;
//MatMul loop
for co in [0: Cout]:
  for i in [0:K*K*C_in]:
    out[co, i] += w[x, i] * x_i2c[i]
```

- Efficient memory access pattern
- Can use SIMD instructions

Convolutional Layer



DL Optimizations for IoT Devices

Other Layers

Other Layers

- We'll stop here the discussion on computational aspects of DNN layers.

- For sake of time (and to limit your boredom), we didn't mention other layers such as RNN cells (Vanilla RNN, GRU, LSTM, etc.) and Attention. Just know that similar analyses can be applied to those as well.

Other Layers

- In particular, most of the computation of RNN cells can be transformed into a single large matrix multiplication (plus activation functions). So, the considerations done for Dense layers apply here as well.

$$i_t = sigmoid(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$f_t = sigmoid(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

$$o_t = sigmoid(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

$$g_t = tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = tanh(c_t) \odot o_t$$