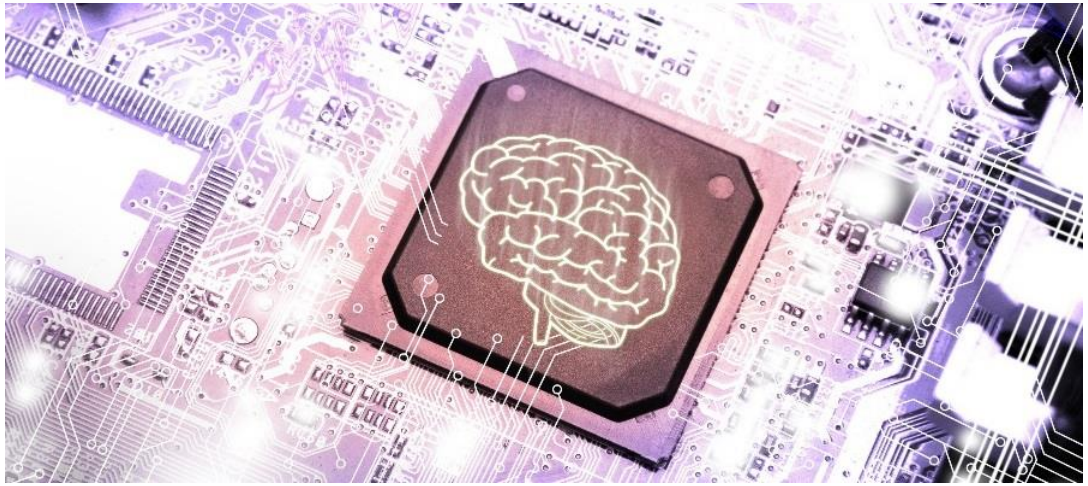


COURS - MICROCONTRÔLEURS ET ARCHITECTURES DÉDIÉES

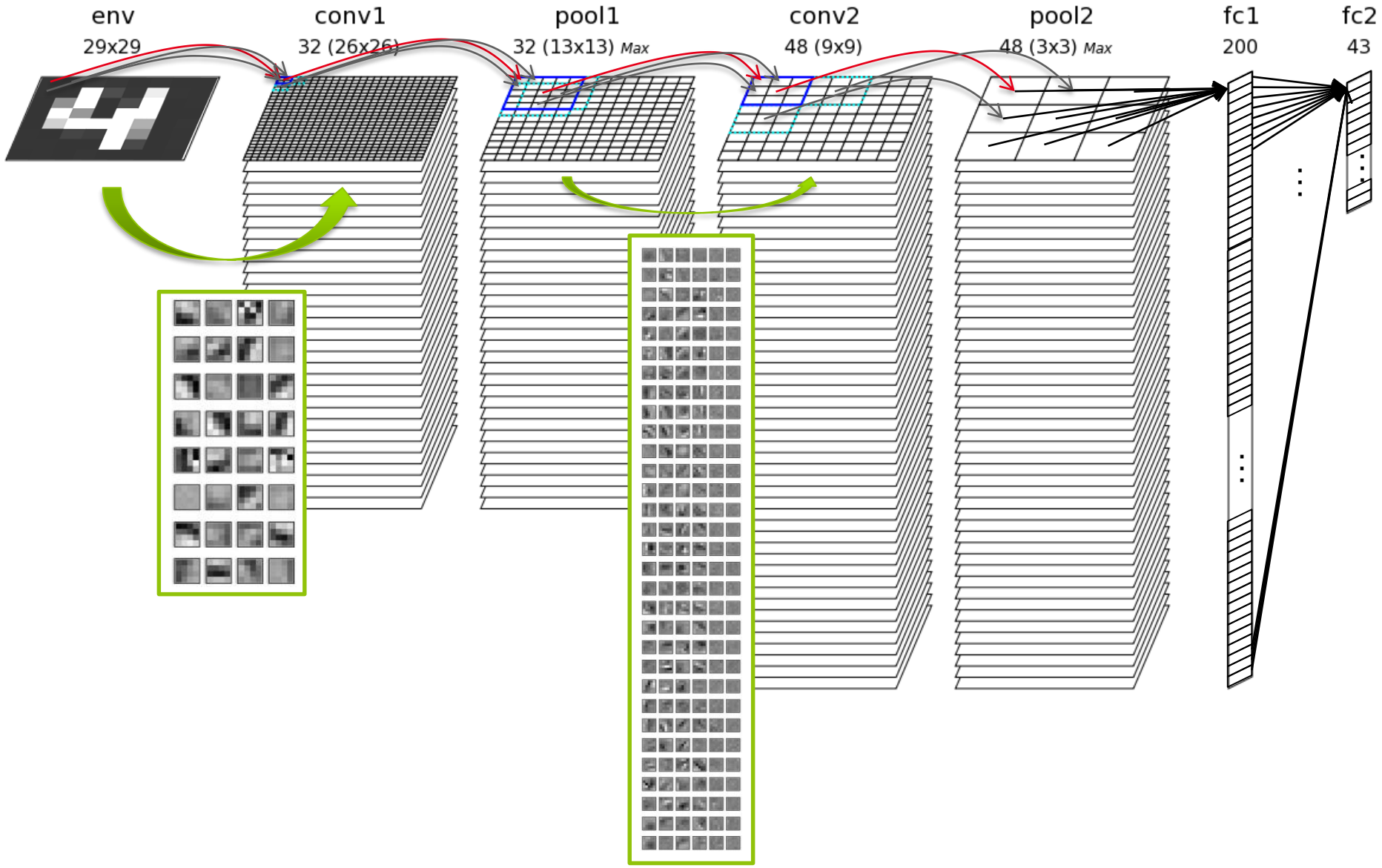
Olivier Bichler
CEA LIST
olivier.bichler@cea.fr

SUMMARY

1. General introduction
2. Memory hierarchy
3. Programming models
4. Architecture examples
5. Advanced concepts

SUMMARY

CONVOLUTIONAL NEURAL NETWORKS OVERVIEW



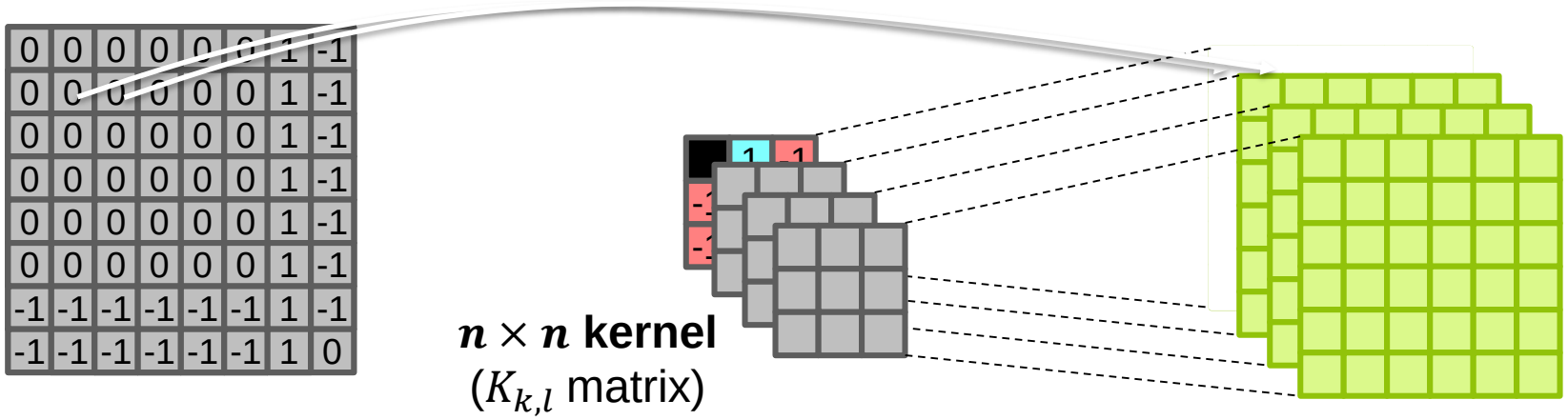
CONVOLUTION OPERATION

- **CNN layer:**

Input map
($I_{i,j}$ matrix)

$$O_{i,j} = \tanh\left(\sum_{k=0}^{n-1}\sum_{l=0}^{n-1} I_{i+k,j+l} . K_{k,l}\right)$$

Output feature map
($O_{i,j}$ matrix)

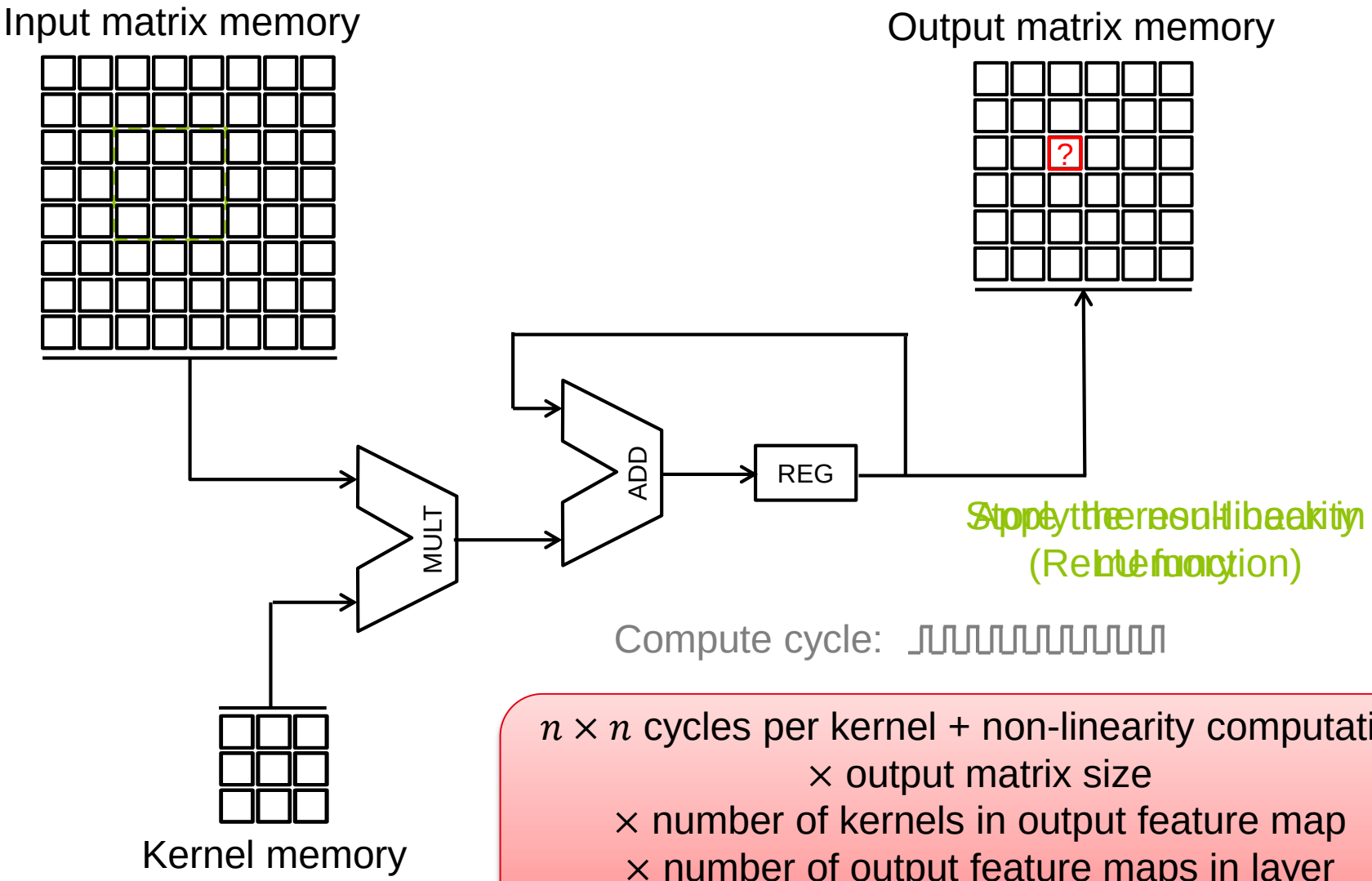


$n \times n$ kernel
($K_{k,l}$ matrix)

Each kernel generates ≠ output feature maps

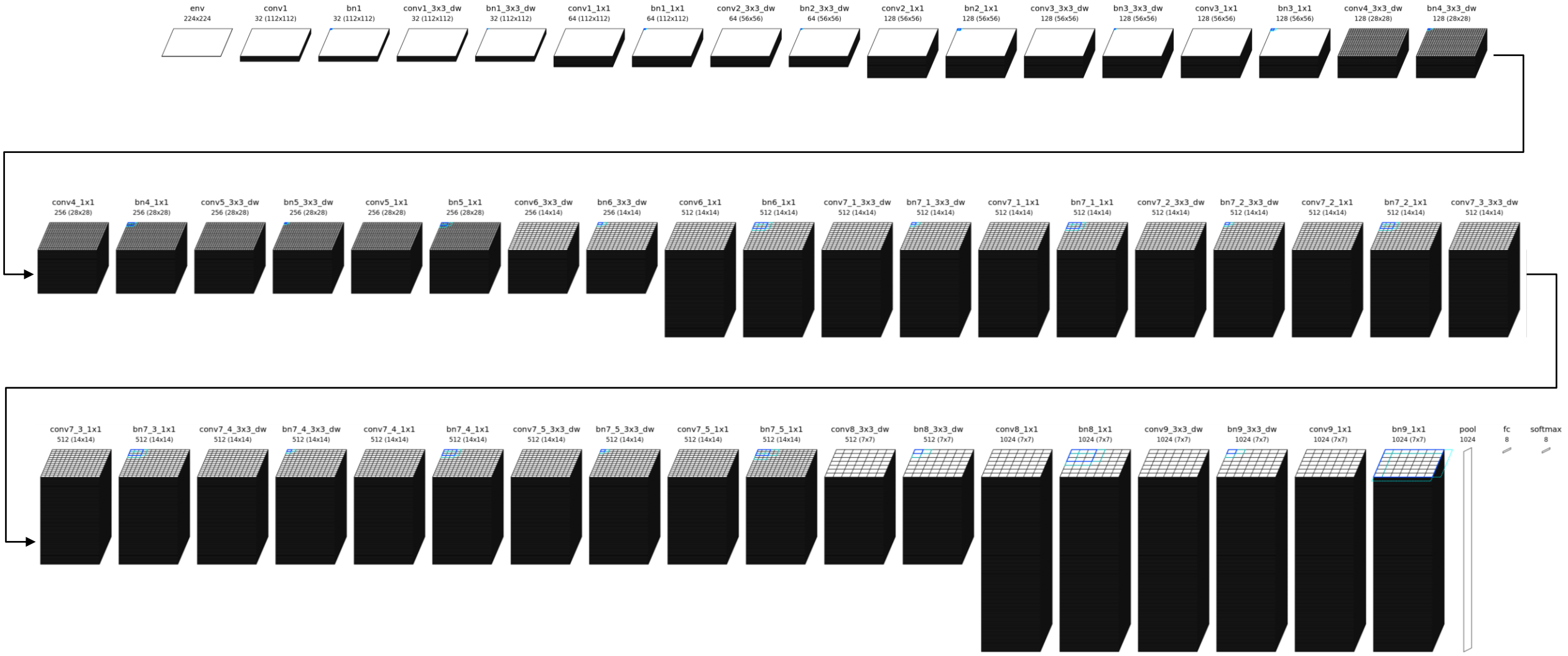
Convolution operation: $O_{i,j} = \tanh\left(\sum_{k=0}^{n-1}\sum_{l=0}^{n-1} I_{i+k,j+l} . K_{k,l}\right)$

MULTIPLY-ACCUMULATE (MAC)



$n \times n$ cycles per kernel + non-linearity computation
× output matrix size
× number of kernels in output feature map
× number of output feature maps in layer
× number of layers

MOBILENET V1 EXAMPLE

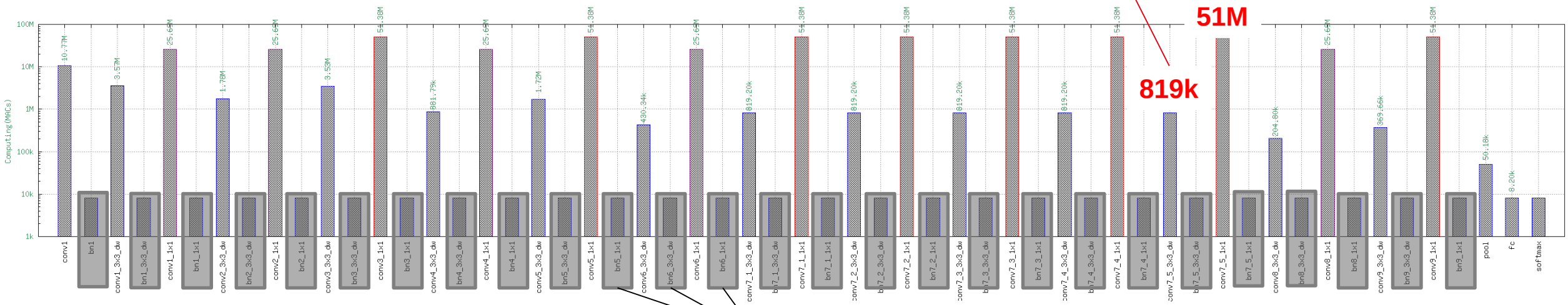


GENERAL INTRODUCTION

MOBILENET V1 EXAMPLE

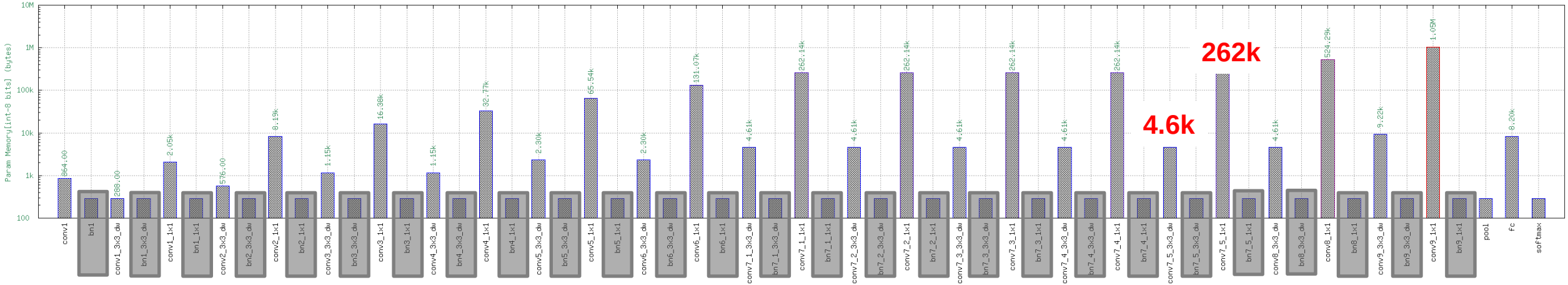
- **Computing requirement (MACs)**

Depthwise convolution
➔ Significant computing and weights reduction



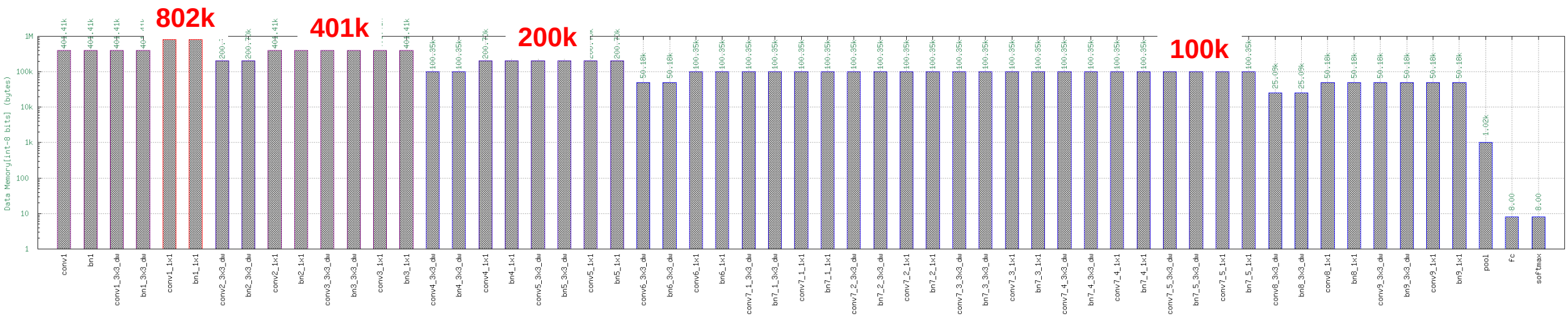
Batch norm.
➔ can be merged with conv.

- **Weights memory requirement (Bytes)**



MOBILENET V1 EXAMPLE

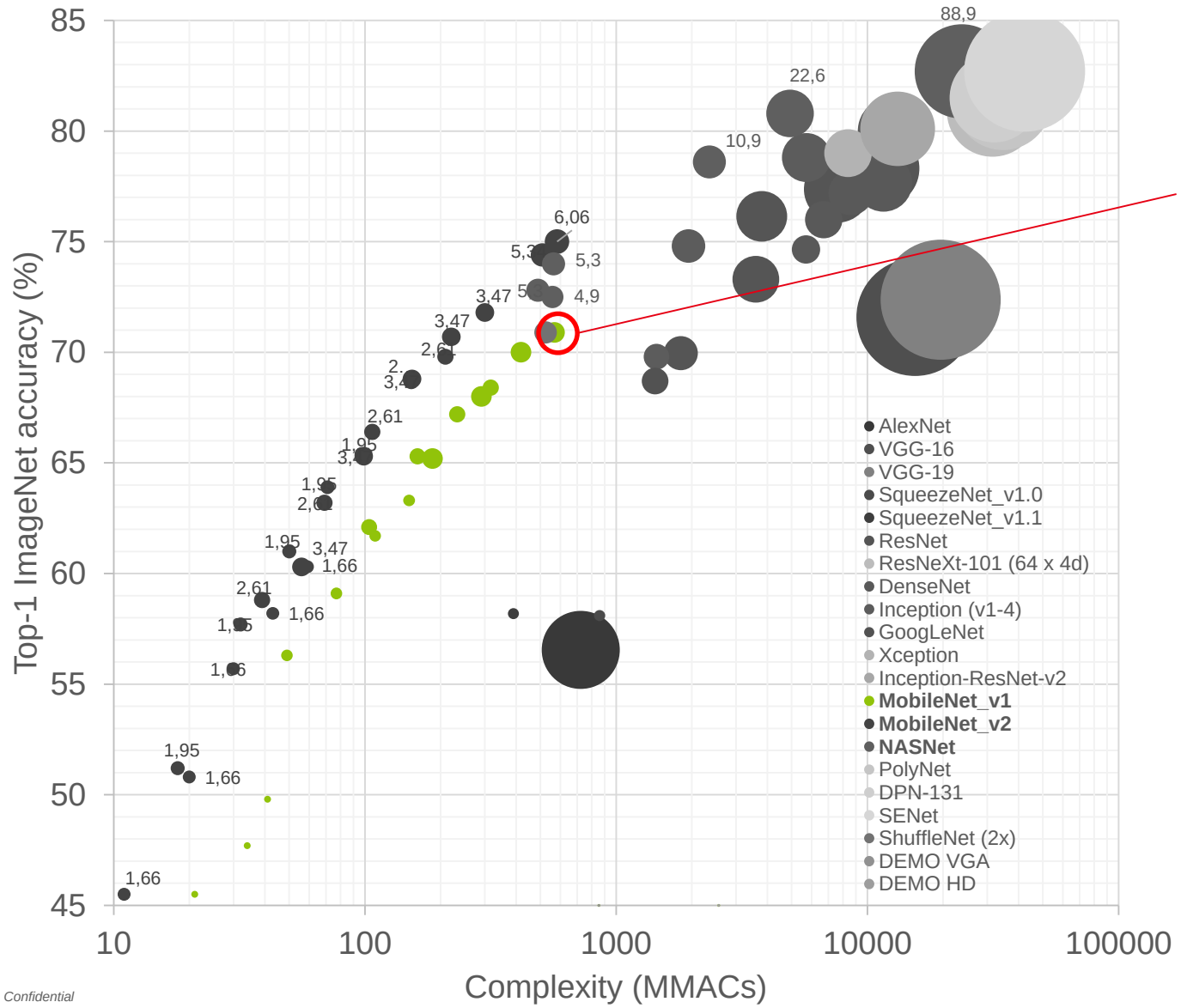
- **Output memory requirement (Bytes)**



- Memory needed to store the full output tensor of each layer
  - Tensor dimension: H (rows), W (columns), C (channels)

- Don’t need to store all the outputs during inference!
  - Re-use the output memory of previous layer for the next one (“ping-pong” buffers)

MOBILENET V1 EXAMPLE SUMMARY




```
[Global stats]
Total number of neurons: 5042704
Total number of nodes: 10086416
Total number of synapses: 3193288
Total number of virtual synapses: 566859944
Total number of connections: 566910120

[Memory]
Input data (int-8 bits): 150.528 kB   (147 KiB)
Input data (float-16 bits): 301.056 kB   (294 KiB)
Input data (float-32 bits): 602.112 kB   (588 KiB)
Free parameters (int-8 bits): 3193.29 kB   (3118.45 KiB)
Free parameters (float-16 bits): 6386.58 kB   (6236.89 KiB)
Free parameters (float-32 bits): 12773.2 kB   (12473.8 KiB)
Layers data (int-8 bits): 10086.4 kB   (9850.02 KiB)
Layers data (float-16 bits): 20172.8 kB   (19700 KiB)
Layers data (float-32 bits): 40345.7 kB   (39400.1 KiB)

[Computing]
MACS / input data: 566.91M
```

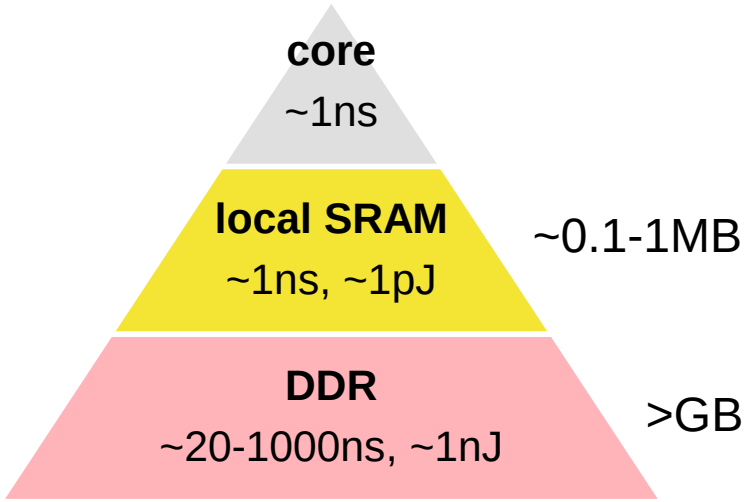
SUMMARY

MEMORY HIERARCHY

THE MEMORY BOTTLENECK

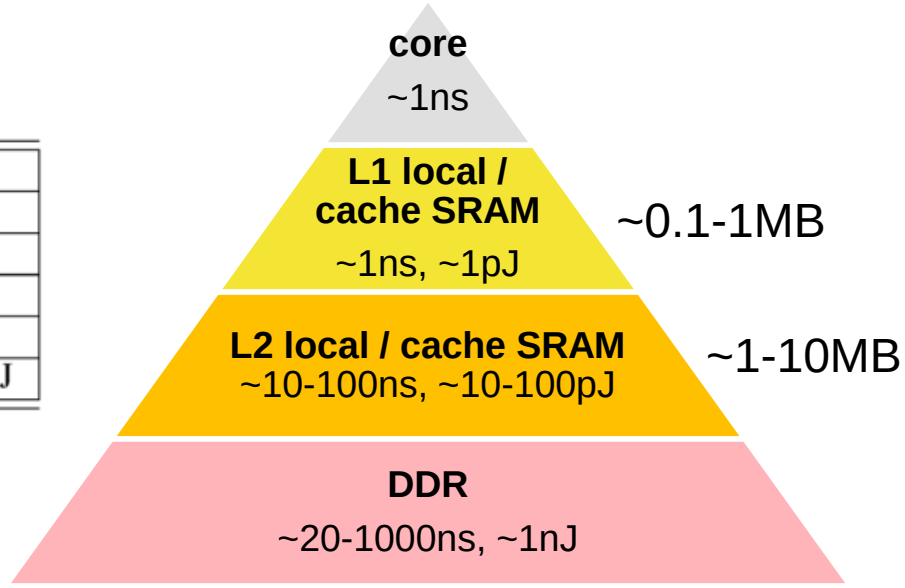
- **Simple memory hierarchy**
  - Example:

Component	Description
CPU Core	Frequency: 1.0 GHz
SPM	SRAM; Size: 1 KB; Access latency: 0.289 ns; Access energy: 0.001 nJ
Main memory	DDR SDRAM; Size: 256MB; Access latency: 17.526 ns; access energy: 0.473 nJ



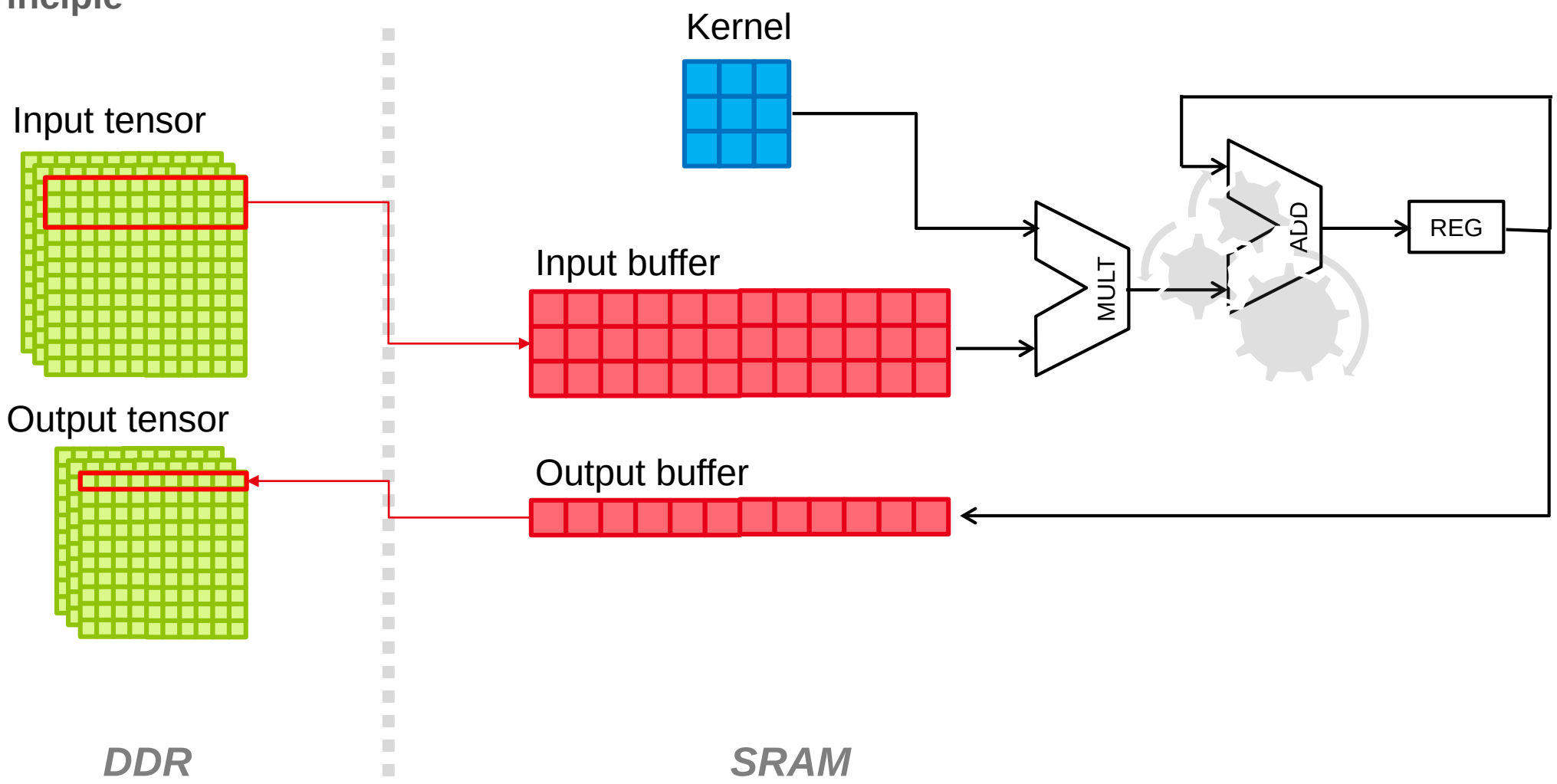
- **Complex memory hierarchy**
  - Example:

Component	Description
CPU Core	Frequency: 1.0 GHz
SPM level 1	SRAM; Size: 1 KB; Access latency: 0.289 ns; Access energy: 0.001 nJ
SPM level 2	SRAM; Size: 2 KB; Access latency: 1.156 ns; Access energy: 0.003 nJ
Main memory 1	SRAM; Size: 4 KB; Access latency: 3.615 ns; Access energy: 0.084 nJ
Main memory 2	DDR SDRAM; Size: 256MB; Access latency: 17.526 ns; Access energy: 0.419 nJ



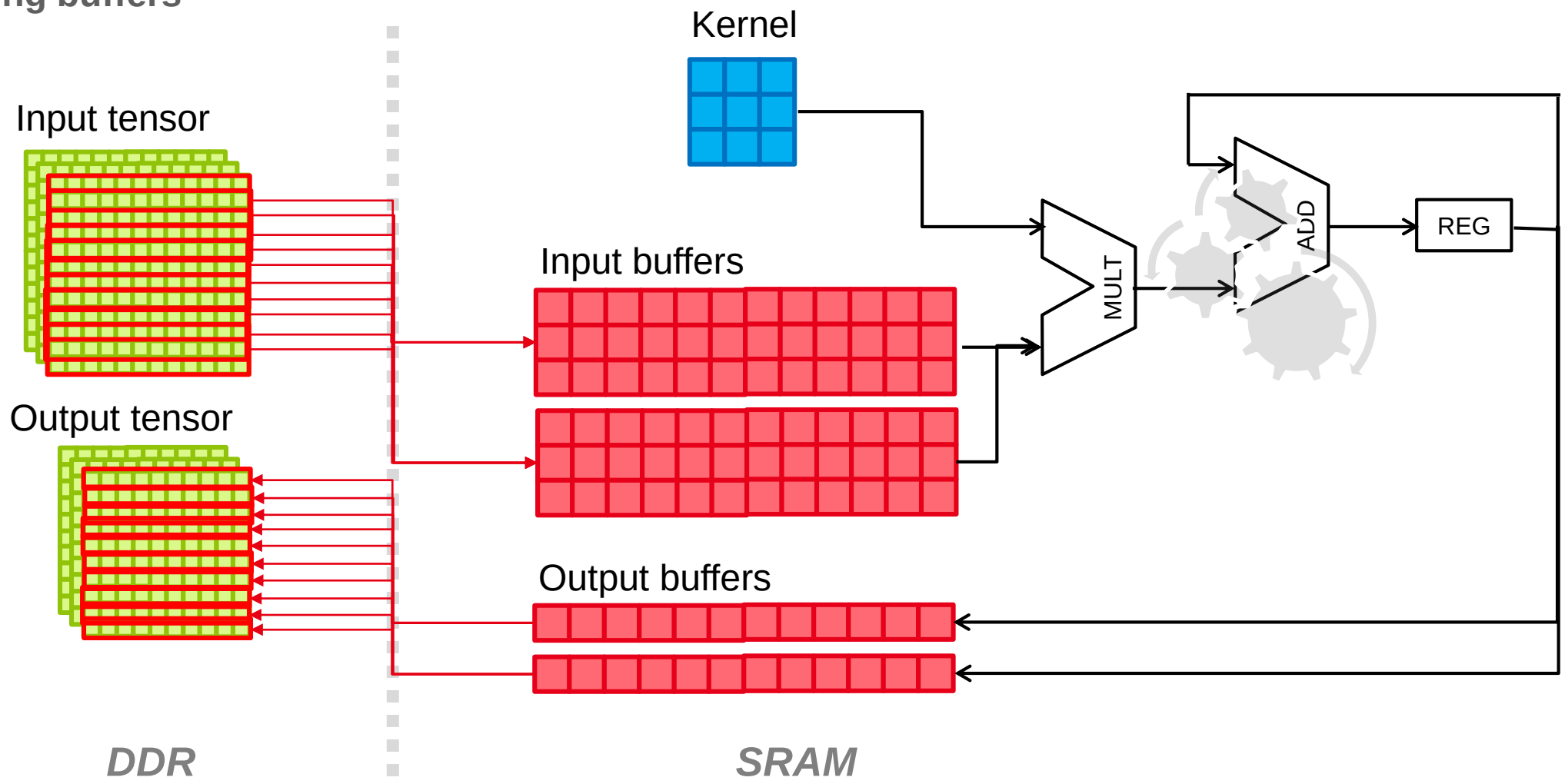
EXPLOITING MEMORY HIERARCHY

- **Basic principle**



EXPLOITING MEMORY HIERARCHY

- **Ping-pong buffers**



EXPLICIT VS IMPLICIT MEMORY TRANSFERS

- **Memory transfers usually handled by a DMA**
- **Can be explicit or implicit**

- **Implicit transfers:**
  - Cache memory
  - Virtual memory

- **Explicit transfers:**
  - Through system call to initiate DMA transfer
  - May be blocking or non-blocking

  - Use non-blocking + synchronization for background memory transfer during computation!

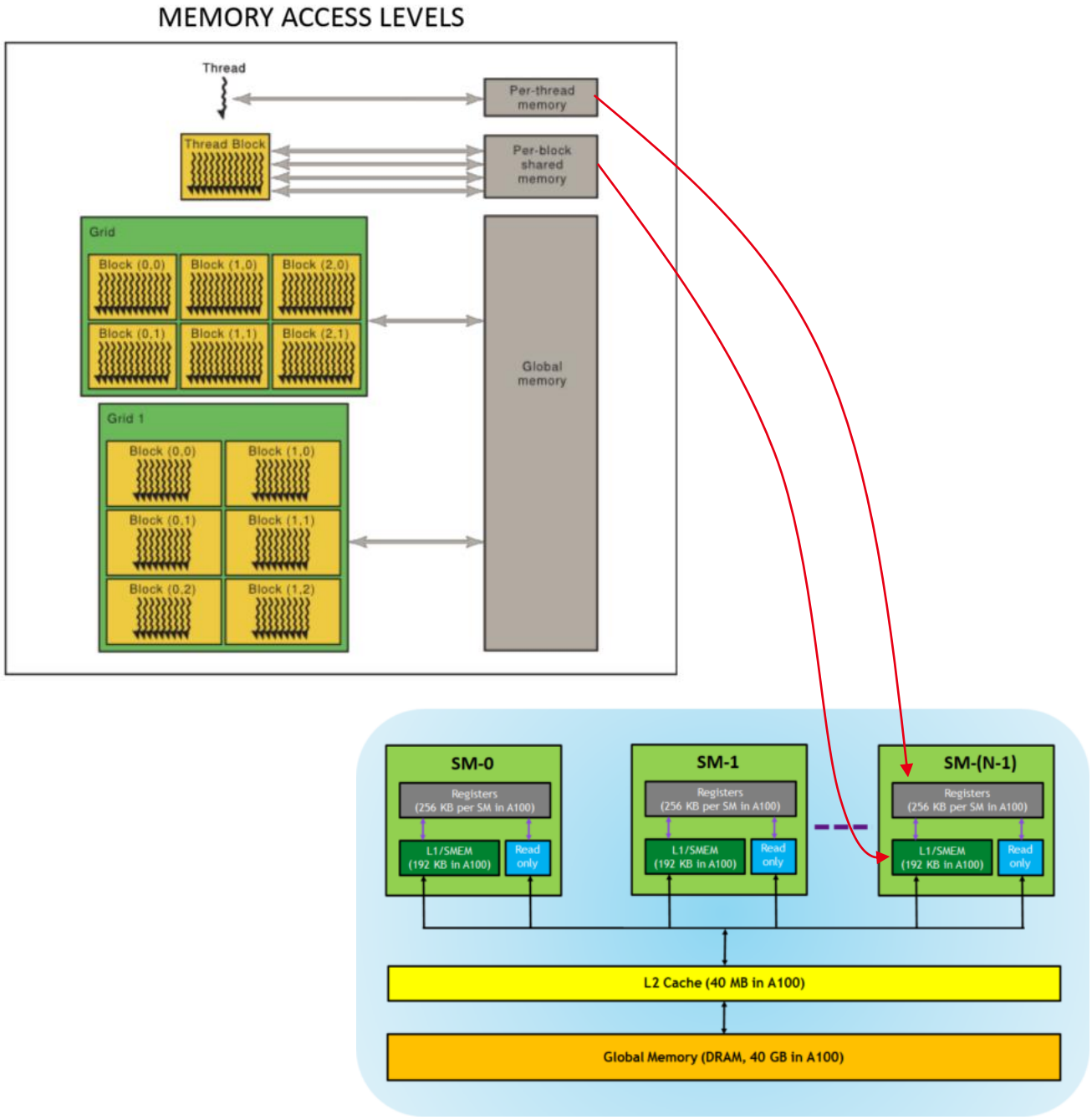
SUMMARY

1. General introduction
2. Memory hierarchy
3. **Programming models**
   - CUDA
   - OpenCL
   - OpenMP
   - Vector instructions (SIMD)
4. Architecture examples
5. Advanced concepts

CUDA

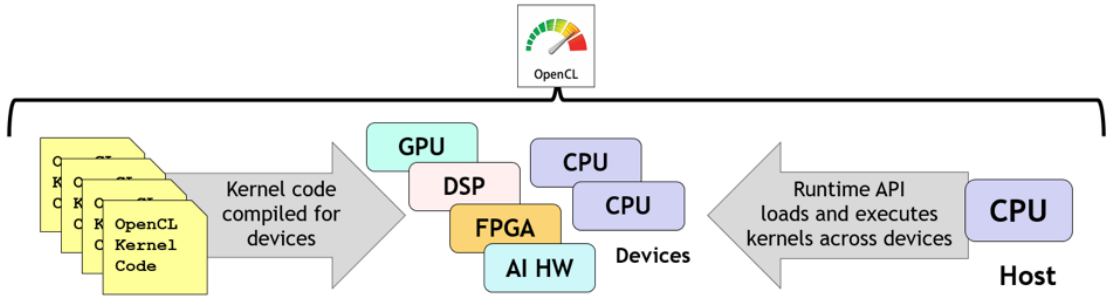
- NVidia GPU only!

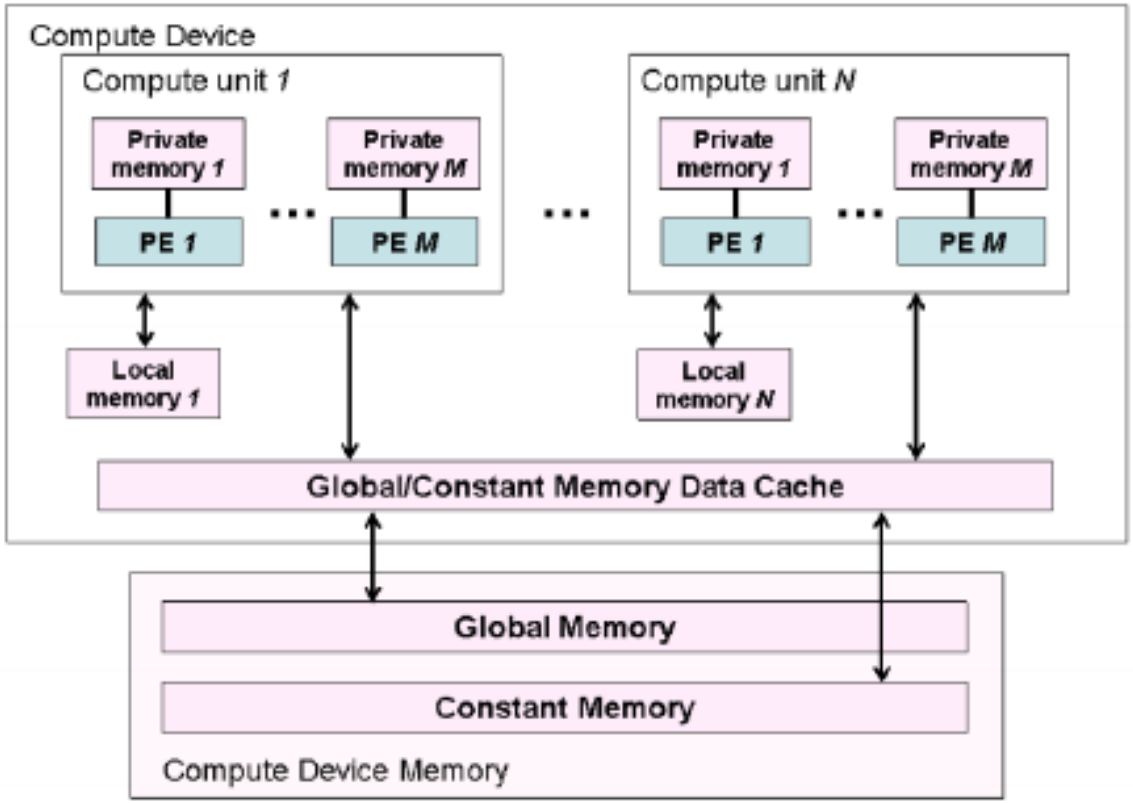
```
// Kernel - Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float
MatB[N][N], float MatC[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        MatC[i][j] = MatA[i][j] + MatB[i][j];
}

int main()
{
    ...
    // Matrix addition kernel launch from host code
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((N + threadsPerBlock.x -1) /
threadsPerBlock.x, (N+threadsPerBlock.y -1) /
threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(MatA,
MatB, MatC);
    ...
}
```



OPENCL

- **Generic, vendor-independent parallel computing model**
  - Very similar to CUDA!
  - Open standard, by the Khronos Group consortium
  - Intel and AMD are major promoters





- **An OpenCL compilation tool must be available for the component…**

CONVOLUTION ALGORITHMS

DIRECT CONVOLUTION

```
for (int oy = 0; oy < OUTPUTS_HEIGHT; ++oy) {
    const int syMin = max(PADDING_Y - (oy * STRIDE_Y), 0);
    const int syMax = clamp(CHANNELS_HEIGHT + PADDING_Y - (oy * STRIDE_Y), 0, KERNEL_HEIGHT);
    const int iy = (oy * STRIDE_Y) - PADDING_Y;

    for (int ox = 0; ox < OUTPUTS_WIDTH; ++ox) {
        const int sxMin = max(PADDING_X - (ox * STRIDE_X), 0);
        const int sxMax = clamp(CHANNELS_WIDTH + PADDING_X - (ox * STRIDE_X), 0, KERNEL_WIDTH);
        const int ix = (ox * STRIDE_X) - PADDING_X;

        for (int output = 0; output < NB_OUTPUTS; ++output) {
            const int oPos = (ox + OUTPUTS_WIDTH * oy);
            const int oOffset = NB_OUTPUTS * oPos;

            SUM_T weightedSum = biasses[output];

            for (int sy = syMin; sy < syMax; ++sy) {
                if (sy >= syMax - syMin)
                    break;

                const int iPos = ((sxMin + ix) + CHANNELS_WIDTH * (iy + syMin + sy));
                const int iOffset = NB_CHANNELS * iPos;
                const int wOffset = NB_CHANNELS * (sxMin + KERNEL_WIDTH * (syMin + sy + KERNEL_HEIGHT * output));

                for (int sx = 0; sx < KERNEL_WIDTH; ++sx) {
                    if (sx >= sxMax - sxMin)
                        break;

                    for (int ch = 0; ch < NB_CHANNELS; ++ch)
                        weightedSum += weights[wOffset + sx * NB_CHANNELS] * inputs[iOffset + sx * NB_CHANNELS];
                }
            }

            outputs[oOffset + output] = ACTIVATION(weightedSum);
        }
    }
}
```

N

H

W

C

These loops can be merged in any order
➔ Parallelization possible

Contiguous data in memory
➔ SIMD instructions possible

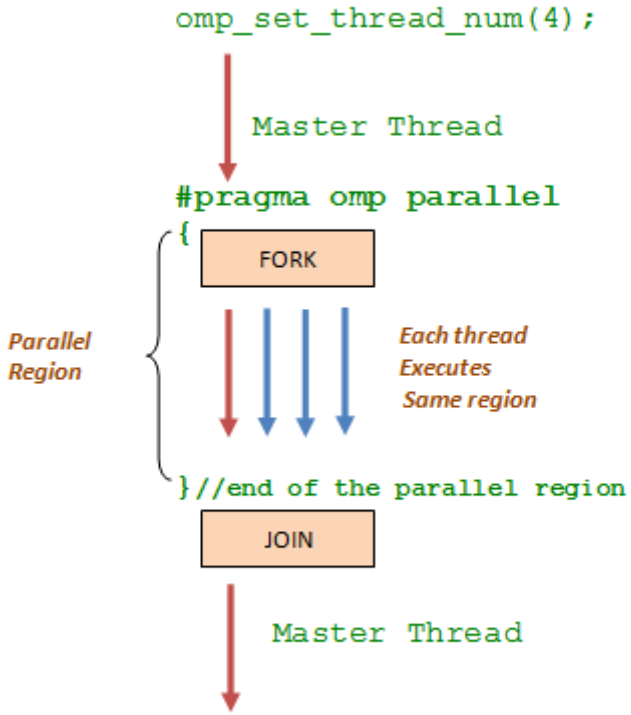
OPENMP

- **Relatively straight forward once the memory hierarchy is correctly handled**

```
for (int oy = 0; oy < OUTPUTS_HEIGHT; ++oy) {
    const int syMin = max(PADDING_Y - (oy * STRIDE_Y), 0);
    const int syMax = clamp(CHANNELS_HEIGHT + PADDING_Y - (oy * STRIDE_Y), 0, KERNEL_HEIGHT);
    const int iy = (oy * STRIDE_Y) - PADDING_Y;

    for (int ox = 0; ox < OUTPUTS_WIDTH; ++ox) {
        const int sxMin = max(PADDING_X - (ox * STRIDE_X), 0);
        const int sxMax = clamp(CHANNELS_WIDTH + PADDING_X - (ox * STRIDE_X), 0, KERNEL_WIDTH);
        const int ix = (ox * STRIDE_X) - PADDING_X;

        for (int output = 0; output < NB_OUTPUTS; ++output) {
            ...
```

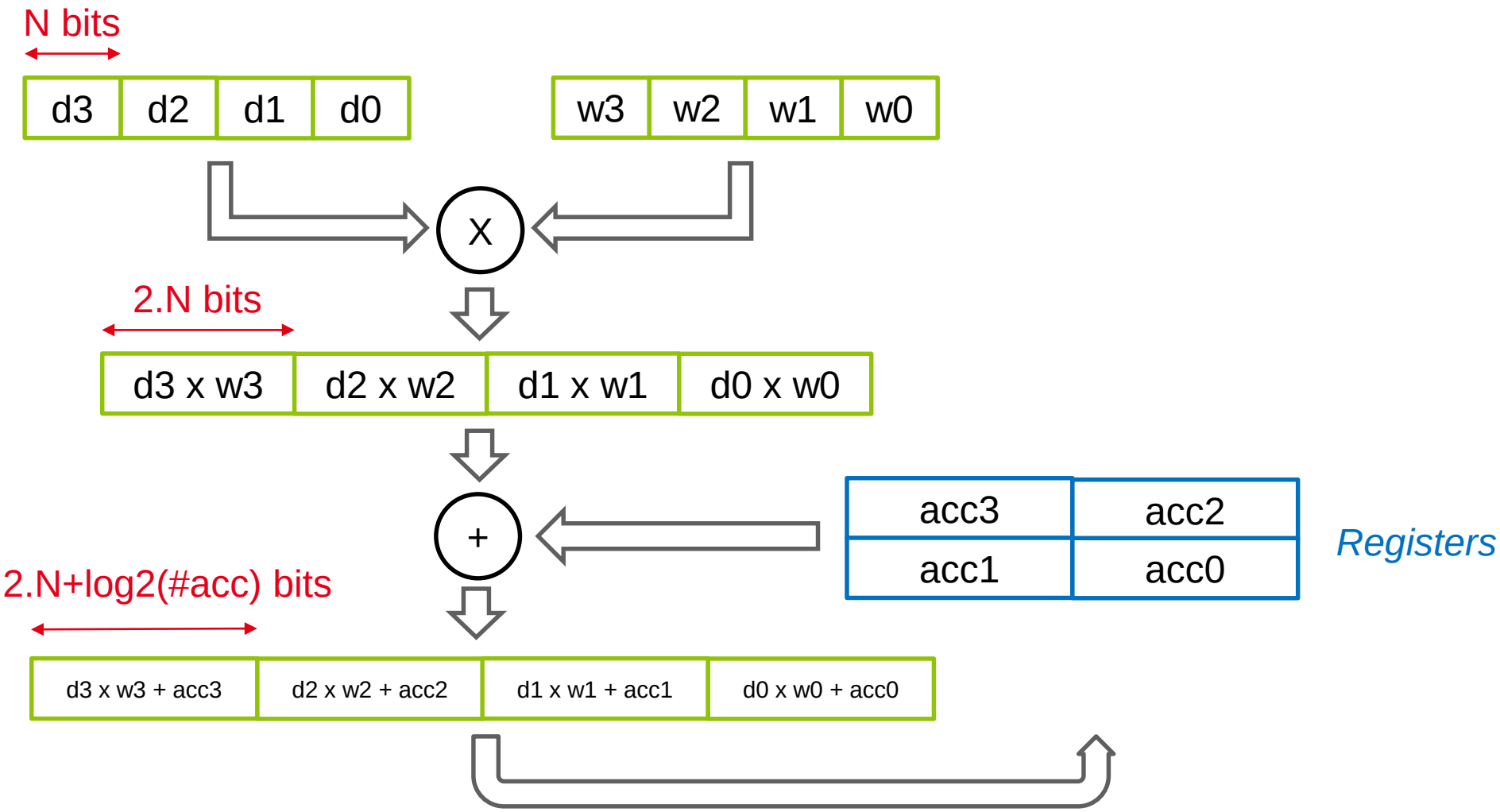


```
for (int oy = 0; oy < OUTPUTS_HEIGHT; ++oy) {
    const int syMin = max(PADDING_Y - (oy * STRIDE_Y), 0);
    const int syMax = clamp(CHANNELS_HEIGHT + PADDING_Y - (oy * STRIDE_Y), 0, KERNEL_HEIGHT);
    const int iy = (oy * STRIDE_Y) - PADDING_Y;

    #pragma omp parallel for collapse(2)
    for (int ox = 0; ox < OUTPUTS_WIDTH; ++ox) {
        for (int output = 0; output < NB_OUTPUTS; ++output) {
            const int sxMin = max(PADDING_X - (ox * STRIDE_X), 0);
            const int sxMax = clamp(CHANNELS_WIDTH + PADDING_X - (ox * STRIDE_X), 0, KERNEL_WIDTH);
            const int ix = (ox * STRIDE_X) - PADDING_X;


            ...
```



VECTOR INSTRUCTIONS (SIMD)

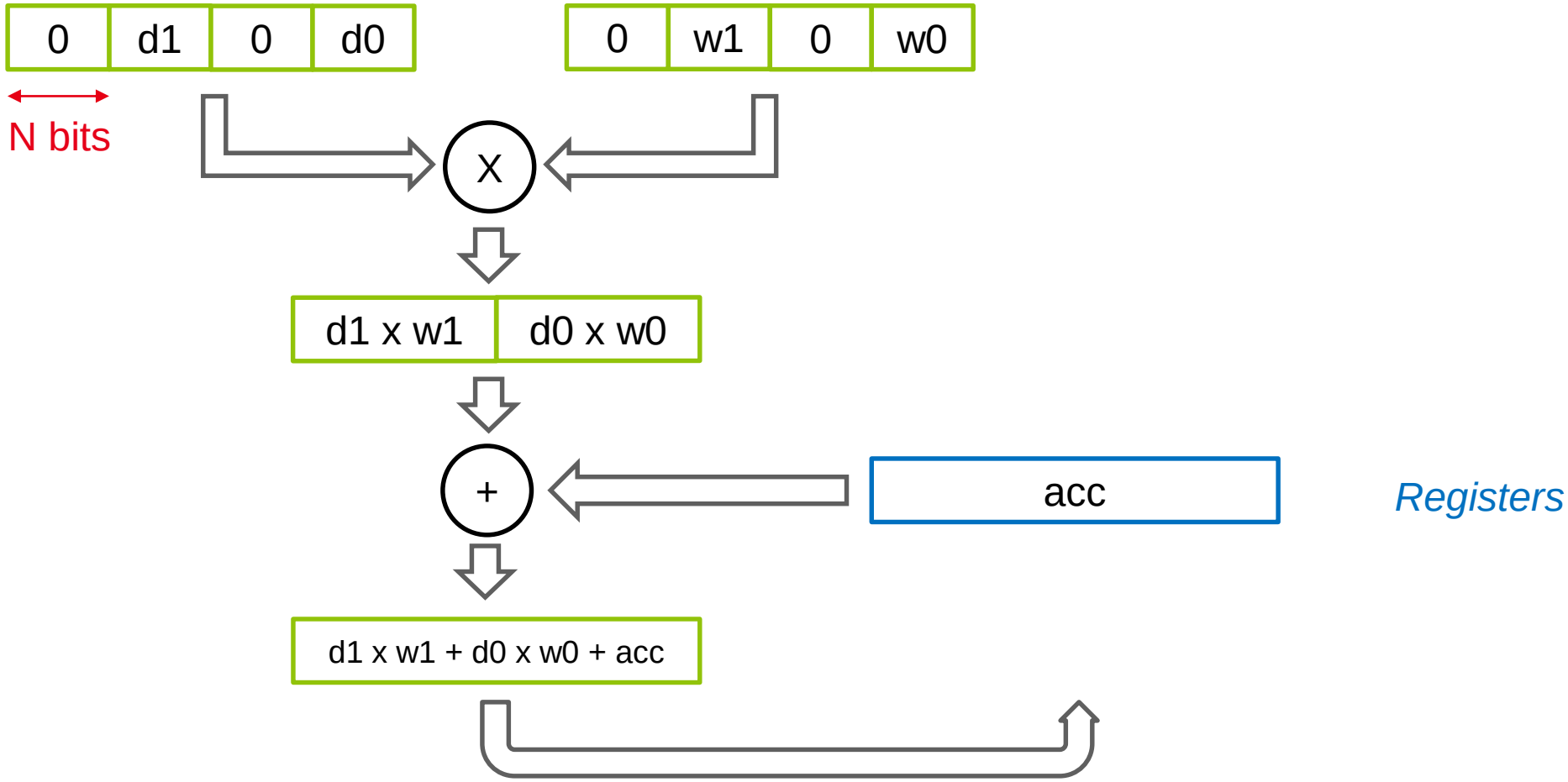
- **Vectorized MAC operation principle (in theory)**



Beware potential overflows!

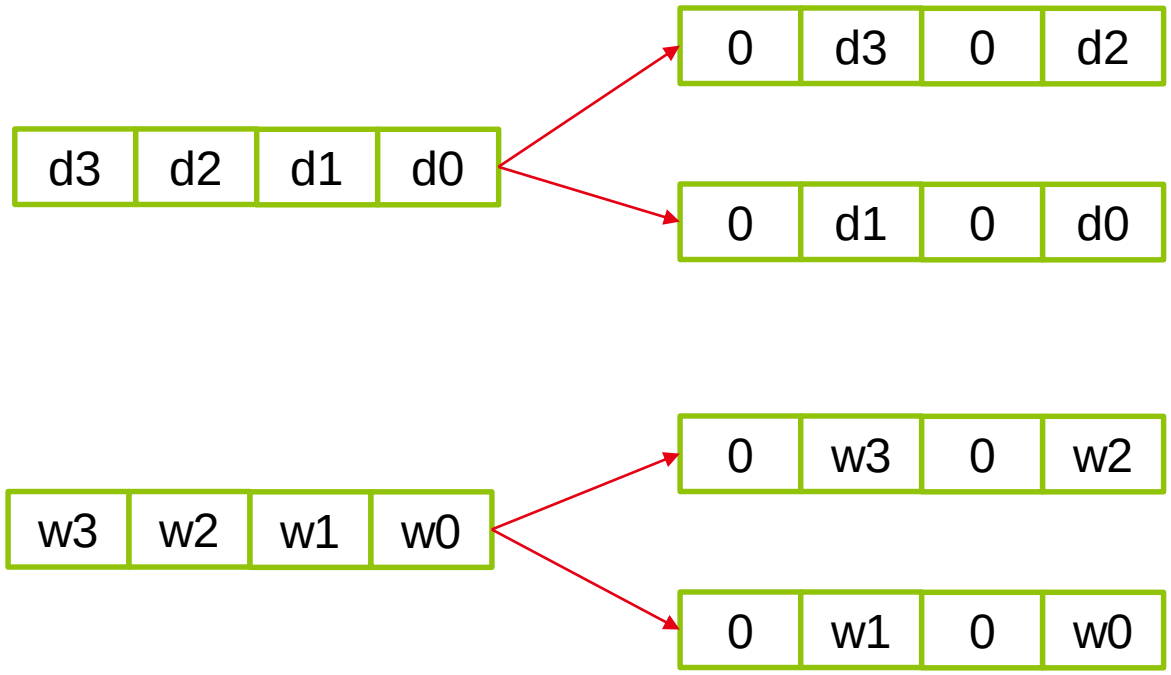
VECTOR INSTRUCTIONS (SIMD)

- **Vectorized MAC operation principle (in practice)**
  - The words have a fixed bit width in hardware



VECTOR INSTRUCTIONS (SIMD)

- **Packing and unpacking**
  - Basic operations generally included in SIMD extension
  - Don’t store 0s in memory!



VECTOR INSTRUCTIONS (SIMD)

- **Example on STM32 (Cortex M4)**
  - Embed DSP instructions
  - Accessible through C/C++ intrinsics

```
SUM_T quadMac(const Input_T* __restrict inputs,
              const WDATA_T* __restrict weights,
              SUM_T weightedSum)
{
          std::uint32_t in;
          std::memcpy((void*) &in, inputs, sizeof(in));
          std::uint32_t in1 = __SXTB16(in);
          std::uint32_t in2 = __SXTB16(in >> 8);

          std::uint32_t wt;
          std::memcpy((void*) &wt, weights, sizeof(wt));
          std::uint32_t wt1 = __SXTB16(wt);
          std::uint32_t wt2 = __SXTB16(wt >> 8);

          weightedSum = __SMLAD(in1, wt1, weightedSum);
          weightedSum = __SMLAD(in2, wt2, weightedSum);
          return weightedSum;
}
```

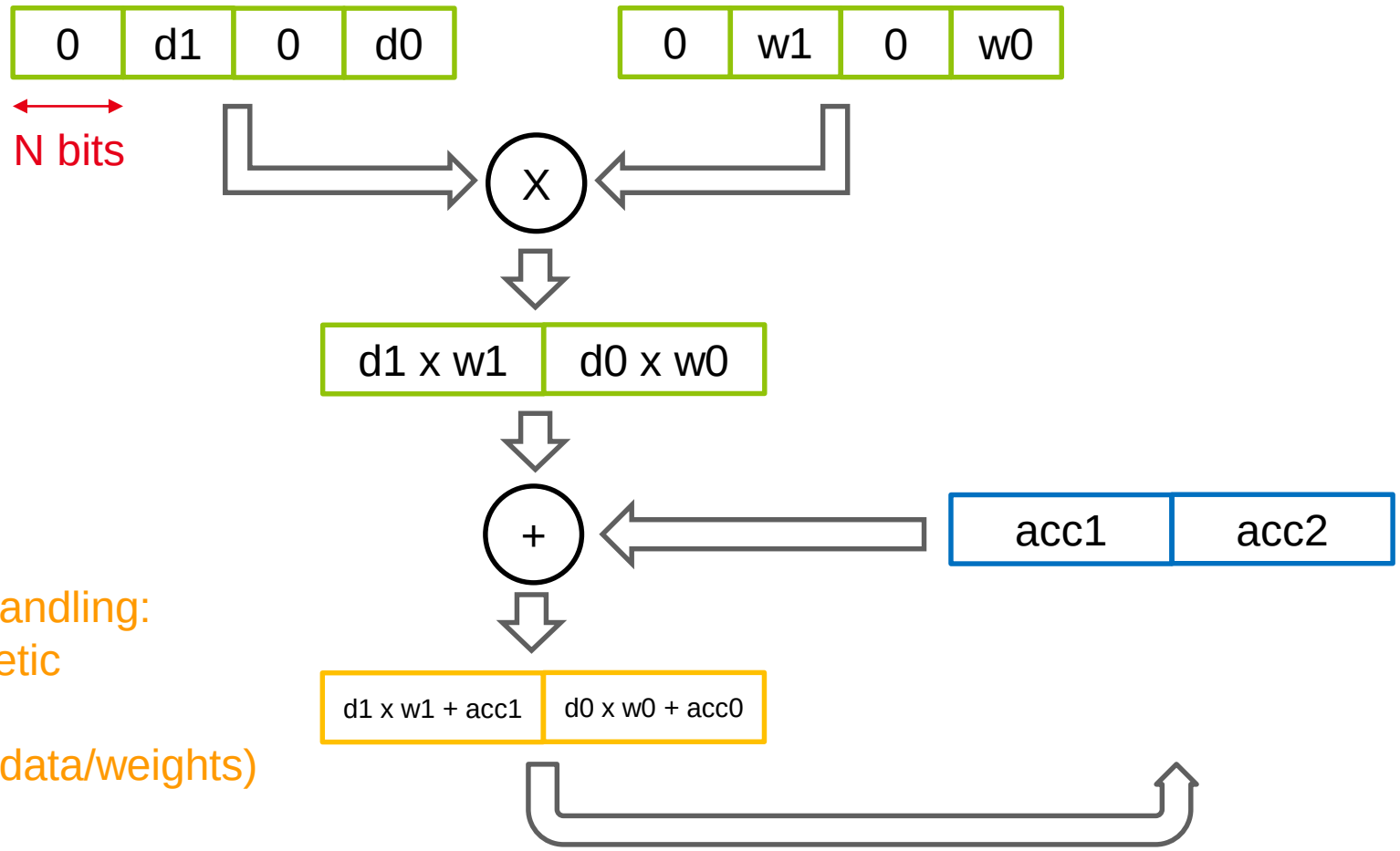
Unpacking

Unpacking

- See https://www.keil.com/pack/doc/CMSIS/Core/html/group__intrinsic__SIMD__gr.html

VECTOR INSTRUCTIONS (SIMD)

- **Possible alternative**
  - Usually require saturated arithmetic



Registers

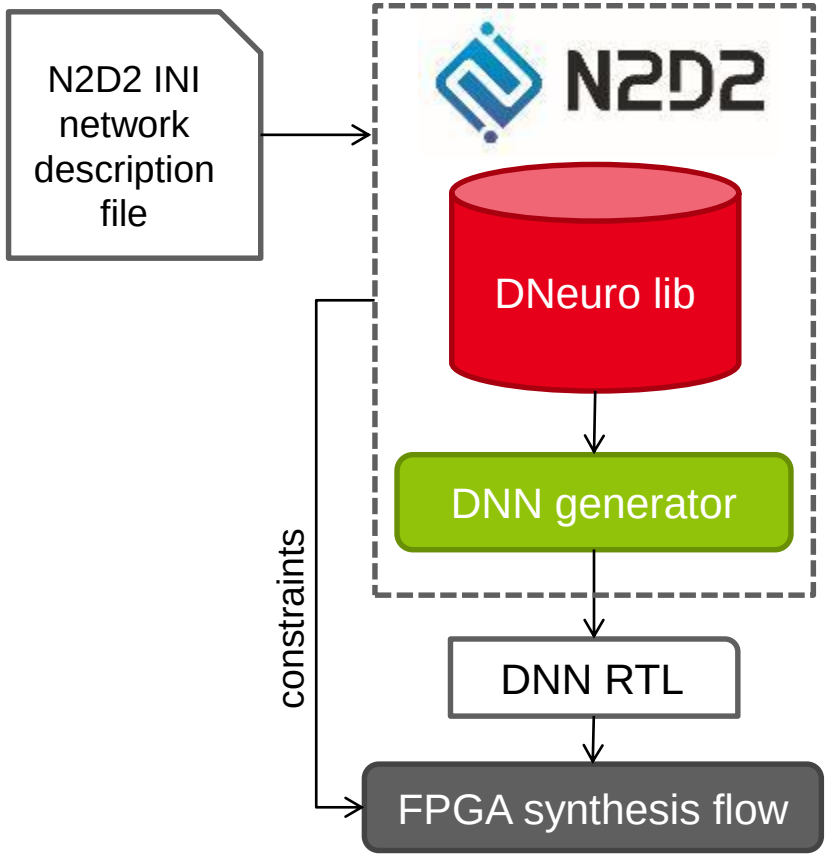
Possible overflow handling:
- saturated arithmetic
- keep margin bits
  (use < N bits for data/weights)

SUMMARY

ARCHITECTURE EXAMPLES

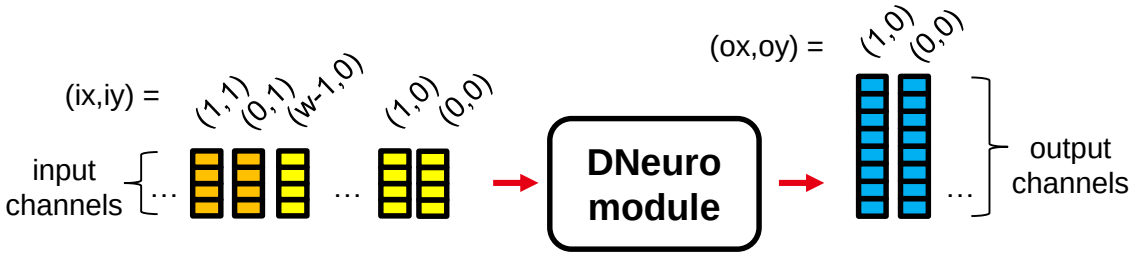
DNEURO: A DATAFLOW RTL IP

- **DNeuro, RTL HW library for FPGA**
  - Complete and independent RTL IP for DNN integration on FPGA
  - Dataflow computation, designed to use the DSP available on FPGA
  - Generated in a few steps from the DNN description and weights
- **Main features**
  - Data flow architecture requiring few memory (potentially **no** DDR)
  - Very high use rate of the DSP per cycle (**> 90%**)
  - Configurable precision (integers from 4 to 16 bits, typically **8 bits**)
  - Up to 4 MAC/DSP operations per cycle
- **Low complexity IP, optimized for Intel and Xilinx FPGA**
- **Support convolutional layers (Fully-CNN)**
  - Convolution and max pooling layers
  - Unit map connectivity and stride support
  - Future work: objects detector layers support (SSD, Yolo, Faster-RCNN…)

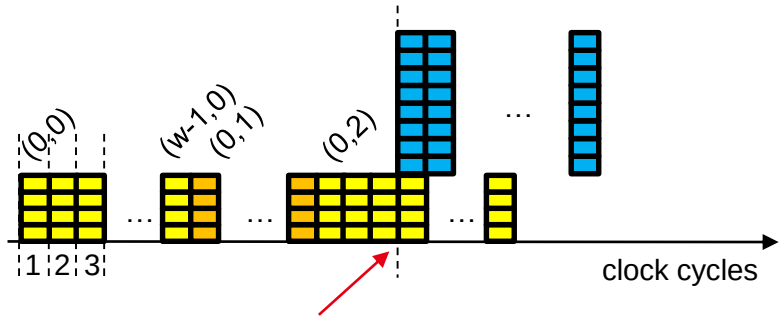


DNEURO WORKING PRINCIPLE

- **Dataflow modules**

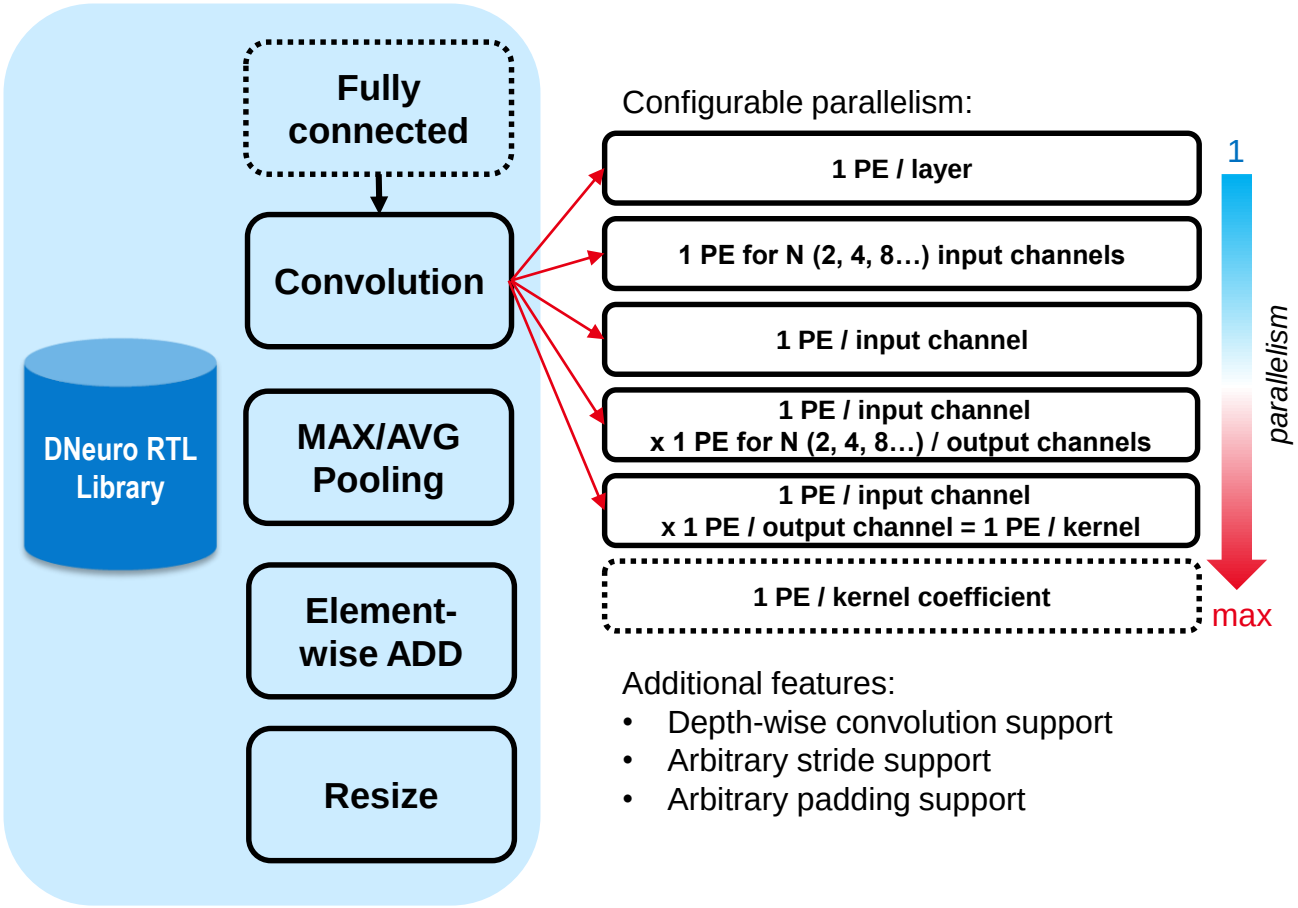


- **Tunable parallelized computation**
  - Down to K_{size} cycle / output (ox,oy)
  - For 32 input x 32 output channels
    ➔ up to 1024 parallel compute units



Output data starts when all inputs data in the first neuron's receptive field is arrived
(e.g. when the 3rd pixel of the 3rd image line is arrived for a 3x3 convolution)

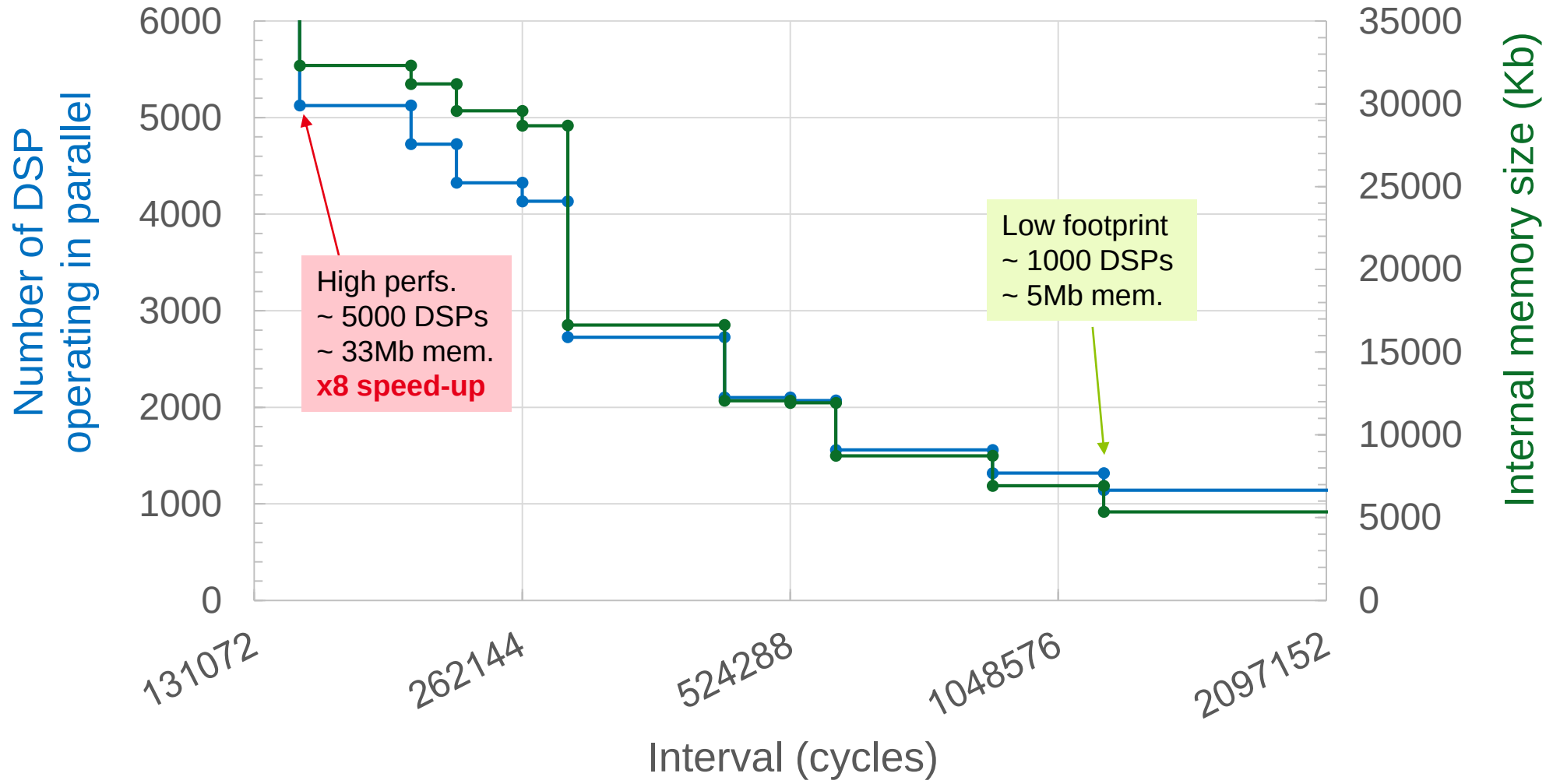
- **DNeuro RTL library modules**



Additional features:
- Depth-wise convolution support
- Arbitrary stride support
- Arbitrary padding support

DNEURO LATENCY VS FOOTPRINT FLEXIBILITY

- **DSP and memory usage can be automatically adjusted to achieve the desired latency**



ARCHITECTURE EXAMPLES

FOOTPRINT ON ARRIA 10 FPGA

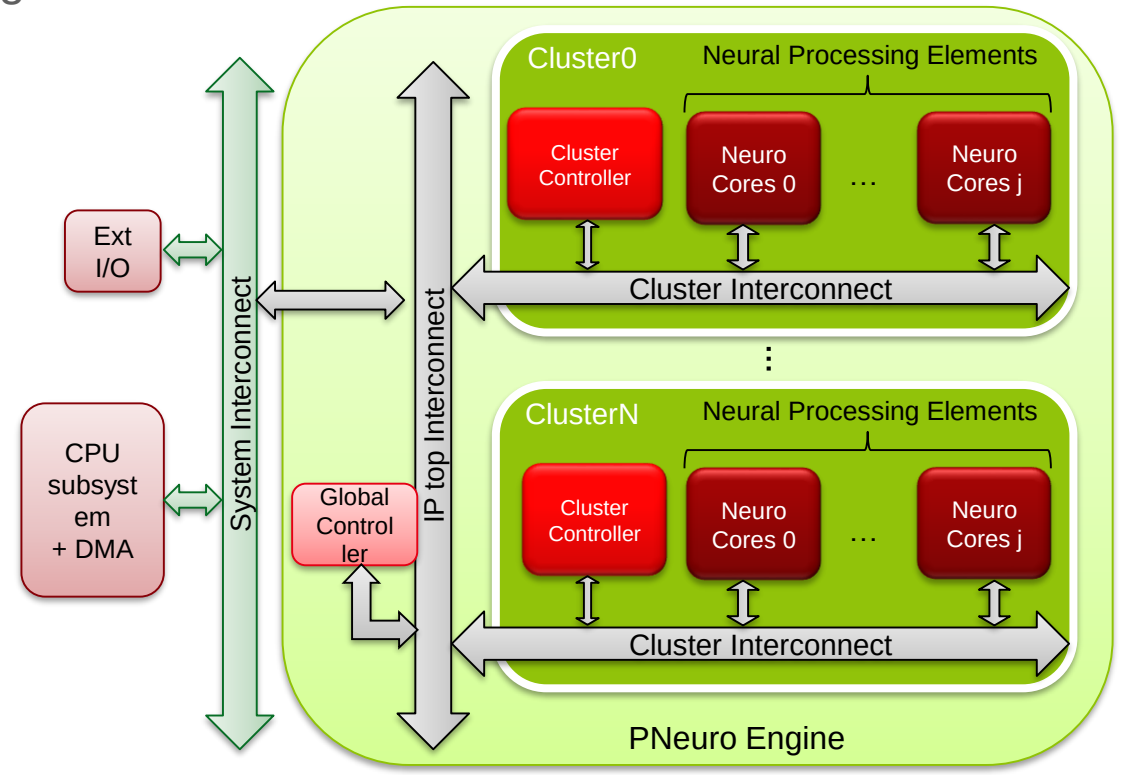
Arria 10	GX/SX 160	GX/SX 220	GX/SX 270	GX/SX 320	GX/SX 480	GX/SX 570	GX/SX 660	GX 900	GX 1150
M20K (MB)	1.12	1.37	1.87	2.12	3.5	4.37	5.25	5.87	6.62
DSP	156	191	830	985	1,368	1,523	1,688	1,518	1,518
Mult. (MAC/c.)	312	382	1,660	1,970	2,736	3,046	3,376	3,036	3,036
MobileNet_v1_0.25	●	●	●	●	●	●	●	●	●
MobileNet_v1_0.5			●	●	●	●	●	●	●
MobileNet_v1_0.75			●		●	●	●	●	●
MobileNet_v1_1.0			●				●	●	●
SqueezeNet_v1.0			● ●	● ●	●	●	●	●	●
SqueezeNet_v1.1			● ●	● ●	●	●	●	●	●
MobileNet_v2_0.35				●	●	●	●	●	●
MobileNet_v2_0.5					●	●	●	●	●
MobileNet_v2_0.75					●	●	●	●	●
MobileNet_v2_1.0						●	●	●	●
MobileNet_v2_1.3								●	●
MobileNet_v2_1.4									●
AlexNet			●	●	●	●	●	●	●
VGG-16						●	● ○	● ○	● ○
GoogLeNet							●	●	●
ResNet-18							●	●	●
ResNet-34								●	●
ResNet-50									○

Table 4: Arria 10 neural networks compatibility table with DNeuro v2, in terms of memory requirement.
Legend:
- ● should be OK for the standard 224x224 input, but depends on the resolution;
- ● M20K memory may be insuffisent depending on the resolution;
- ● there is a better equivalent neural network (see the corresponding arrow);
- ○ using an alternative neural network is possible with a small accuracy loss.

ARCHITECTURE EXAMPLES

PNEURO: A NEURAL DSP PROCESSOR

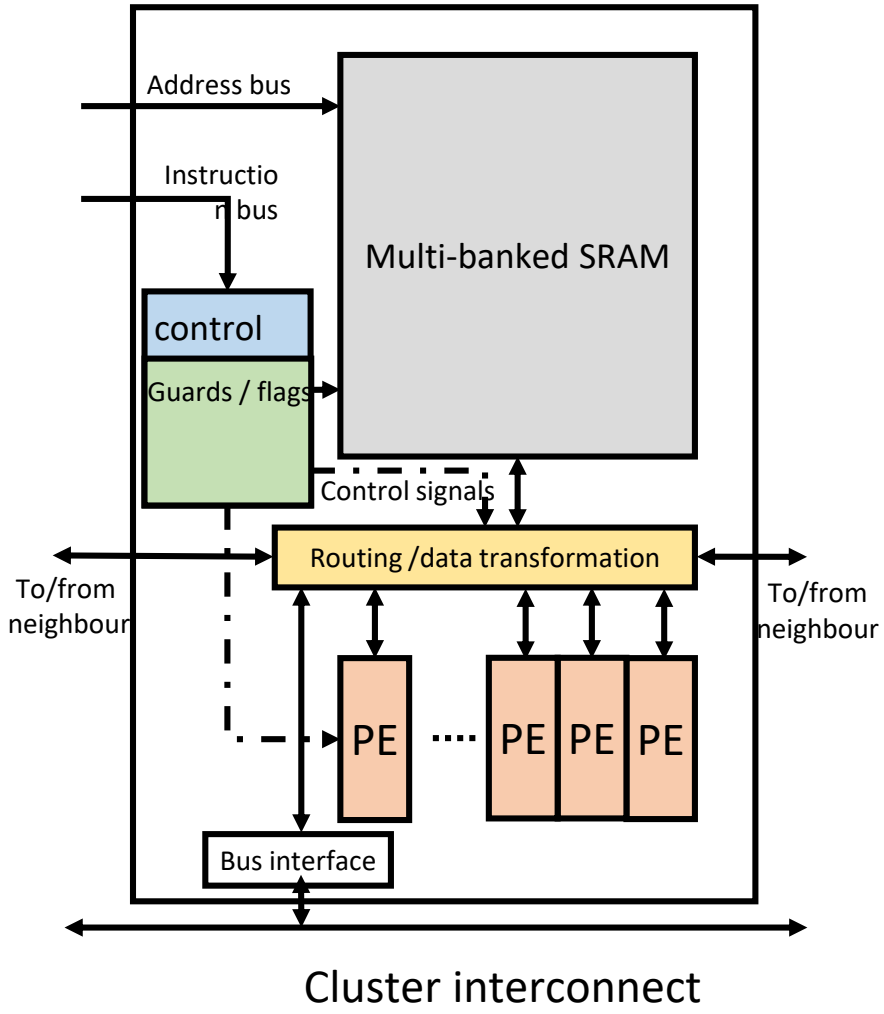
- **Fully-programmable energy efficient hardware accelerator**
  - N2D2 full-development flow
    - Full DNN framework for optimized embedded computing
  - Designed for DNN processing chains
    - Pre/post-processing phases
    - CNN, HMax, RNN (software under development)
  - Supporting traditional image processing operations
    - Filtering, convolution, morpho, etc.
- **Clustered SIMD architecture**
  - Optimized operators for MAC & Non-Linearity approximation
  - Optimized memory accesses to perform efficient data transfers to operators
  - ISA including ~50 instructions (control + computing)



ARCHITECTURE EXAMPLES

PNEURO: NCB ARCHITECTURE

- **Multi-banked SRAM**
  - 2 memories for operands
  - 1 memory for temporary results
  - 1 memory for final results (for next layer / host)
- **Routing / data transformation module**
  - On-the-fly operations with incoming data
    - Padding operations, copy and multicast, bit-shifting…
  - Transfer of multi-precision data (8 and 32 bit paths)
- **SIMD processing elements (PE)**
  - Support image processing and NN operations



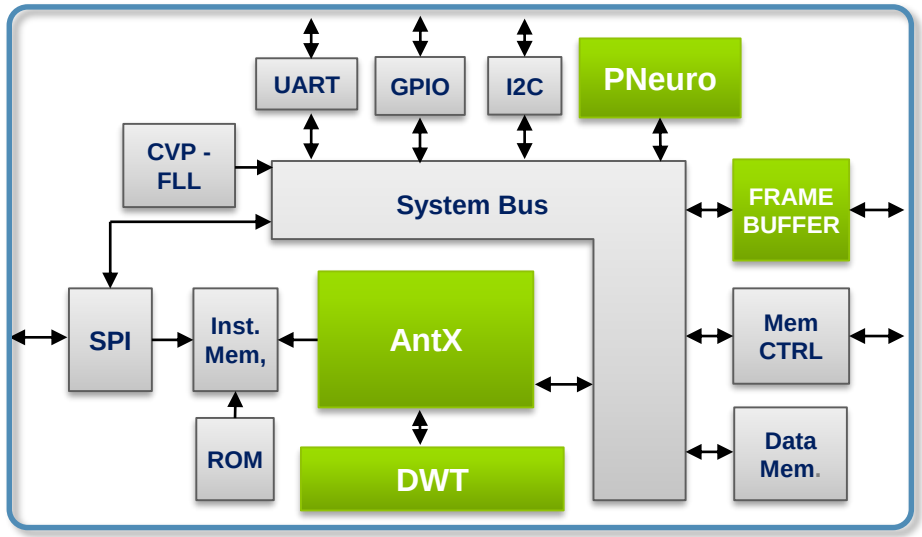
ARCHITECTURE EXAMPLES

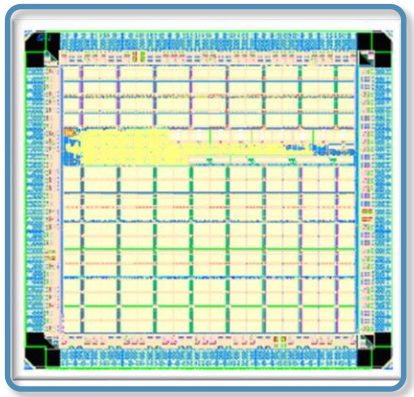
PNEURO FIRST IMPLEMENTATION ON IOT PLATFORM

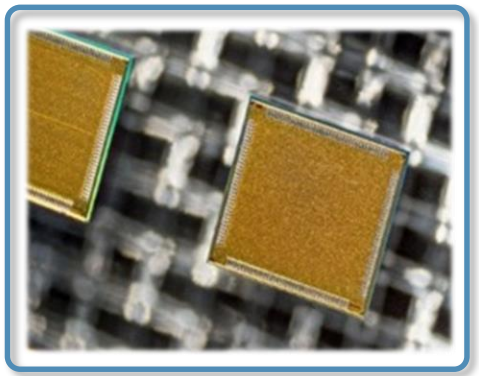
- **Low-power programmable solution for smart IoT**
  - Offers wide range of solutions from image processing and analysis to decision making

- **4 main 100% CEA IPs**
  - AntX low-power control CPU
  - HD Frame Buffer for efficient sensor interface
  - PNeuro neural computing IP (1 cluster)
  - DWT image transformation IP

- **PNEURO 1-core 28 FDSOI**
  - 0,126 mm² | 500 MHz | 3,6mW
    - for a single PNeuro cluster (8 PE) and its control
  - PNeuro performance : 1,1 TMACs/s/W



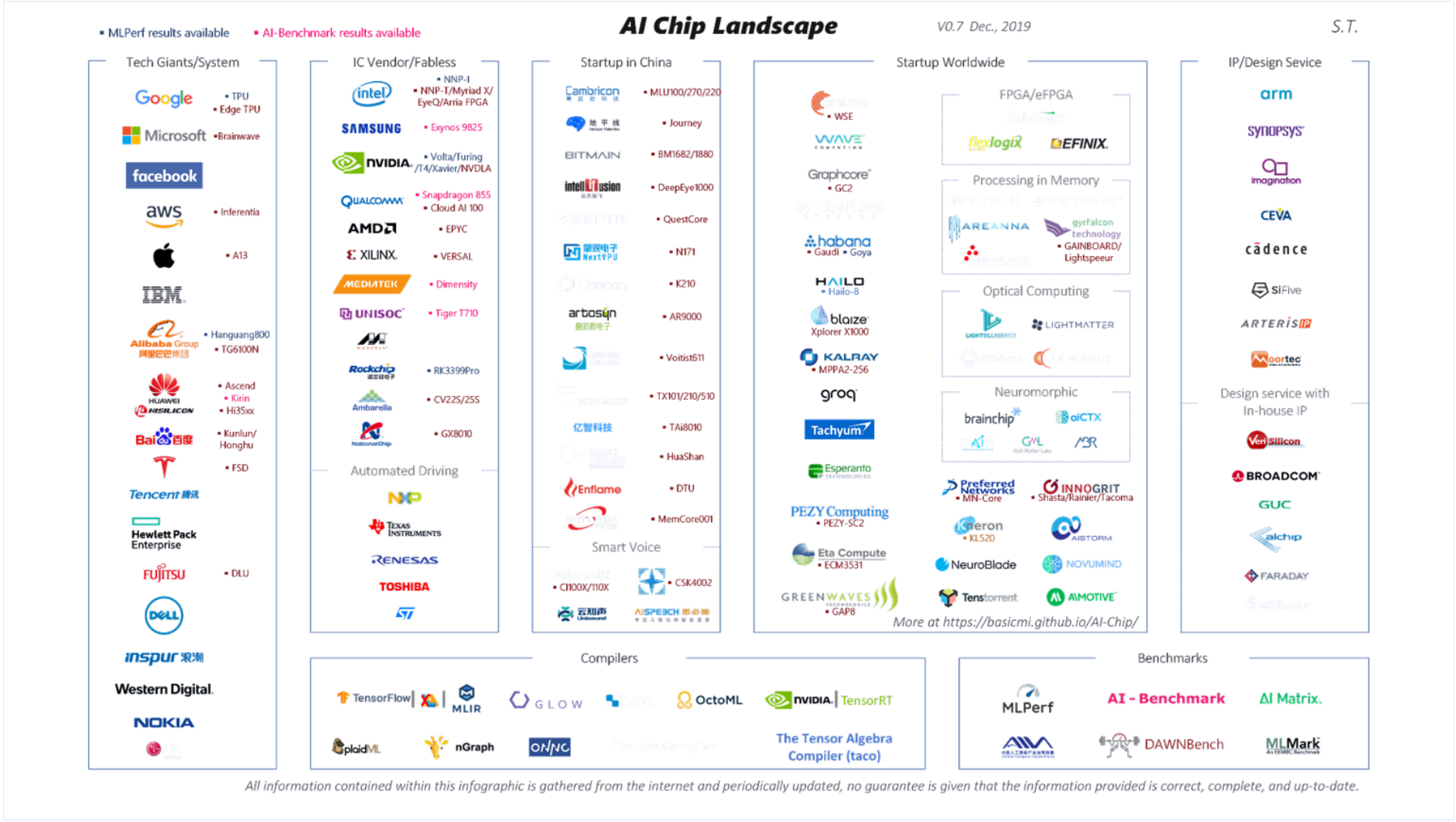




ARCHITECTURE EXAMPLES

MORE ARCHITECTURES

- https://github.com/basicmi/AI-Chip



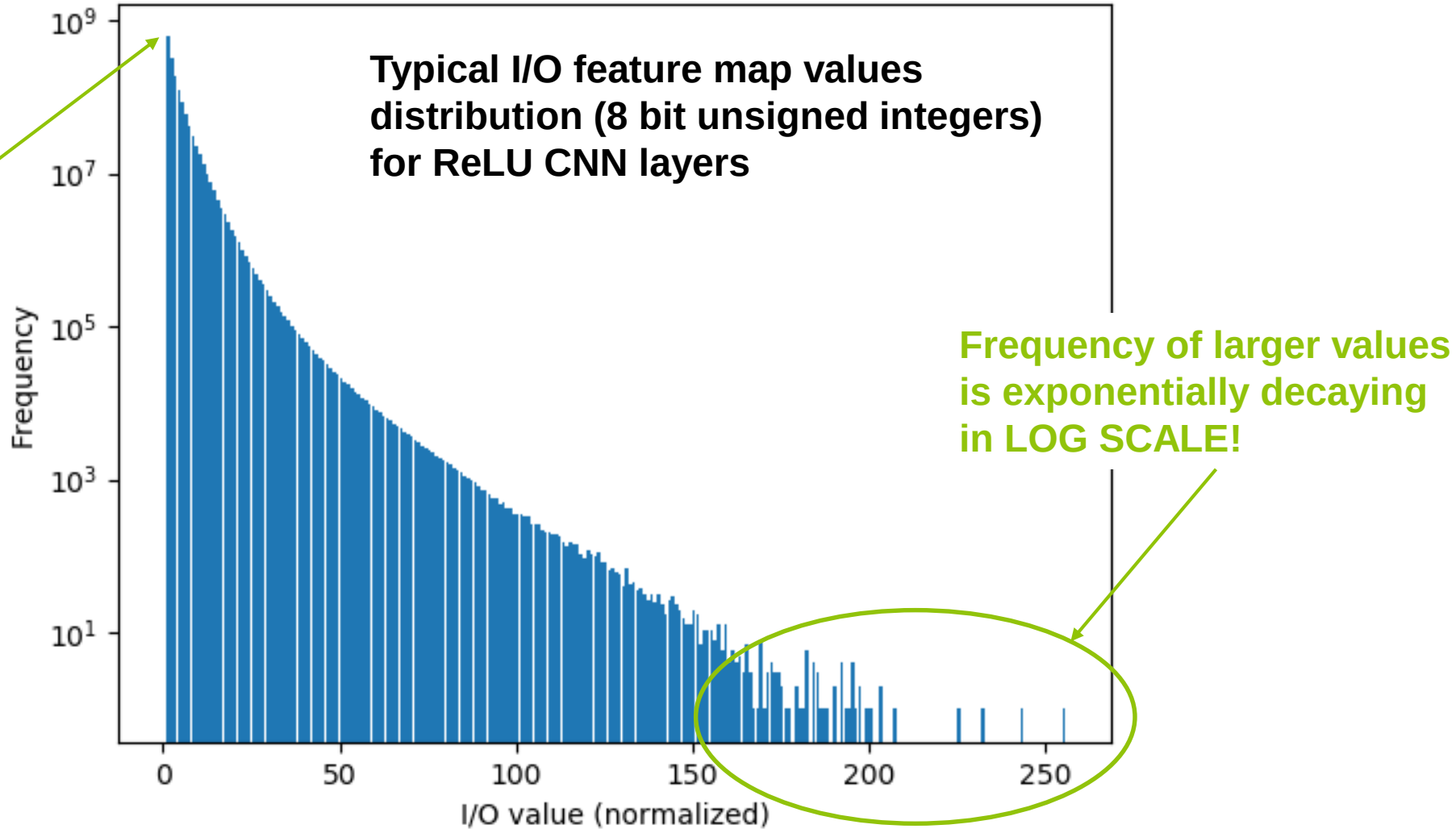
SUMMARY

1. General introduction
2. Memory hierarchy
3. Programming models
4. Architecture examples
5. **Advanced concepts**

TYPICAL ACTIVATION DISTRIBUTION ANALYSIS

Sparse: 80-95% is 0

In this example: 80% (no regularization used during learning)



Frequency of larger values is exponentially decaying in LOG SCALE!

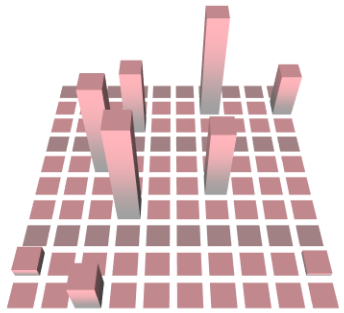
ACTIVATIONS COMPRESSION

- **Sparsity: 0.900450**
- **Non-compressed (8b per value): 125199974400b**
- **Compressed Huffman: 20127883242b**
  - Compression gain Huffman: x6.220226
- **Compressed Non-Zero (1b per 0, 9b per value > 0): 28113628872b**
  - Compression gain Non-Zero: x4.453355
- **Compressed Hop (0b per 0, 8+4b per value > 0): 18695448108b**
  - Compression gain Hop: x6.696816
  - **Used by Eyeriss (MIT): “Run length compression”**
- **Compressed Huffman + Hop (4b): 10709703246b**
  - Compression gain Huffman + Hop (4b): x11.690331

Symbol	Weight	Huffman Code
0	14092042791	1
1	609695202	00
2	313434150	0111
3	189480886	0101
4	124388693	01100
5	85366389	01000
6	59761093	011010
7	42724741	0110111
8	31206737	0100111
9	23179761	0100101
10	17383839	01101101
11	13098285	01001101
12	9926389	01001000
13	7603484	011011000
14	5897847	010011000
15	4604524	010010010
16	3629780	0100110011

ACTIVATIONS CODING

Dense “tensor”

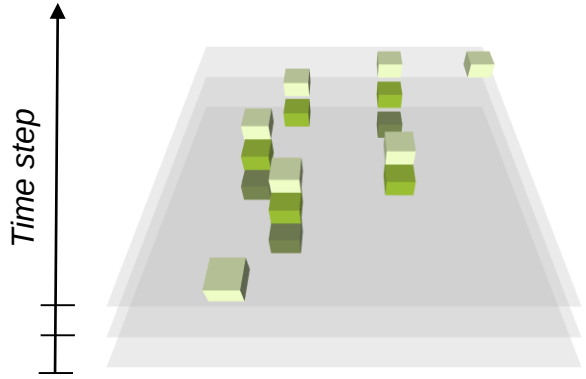


Dense feature map computing

Compressed sparse tensor

Sparse feature map computing

“Spike”-coding



Sparse feature map computing with temporal quantization