High-Level Design of 2D-Convolution Accelerators for AI Leveraging Embedded Scalable Platform (ESP)

Politecnico di Torino

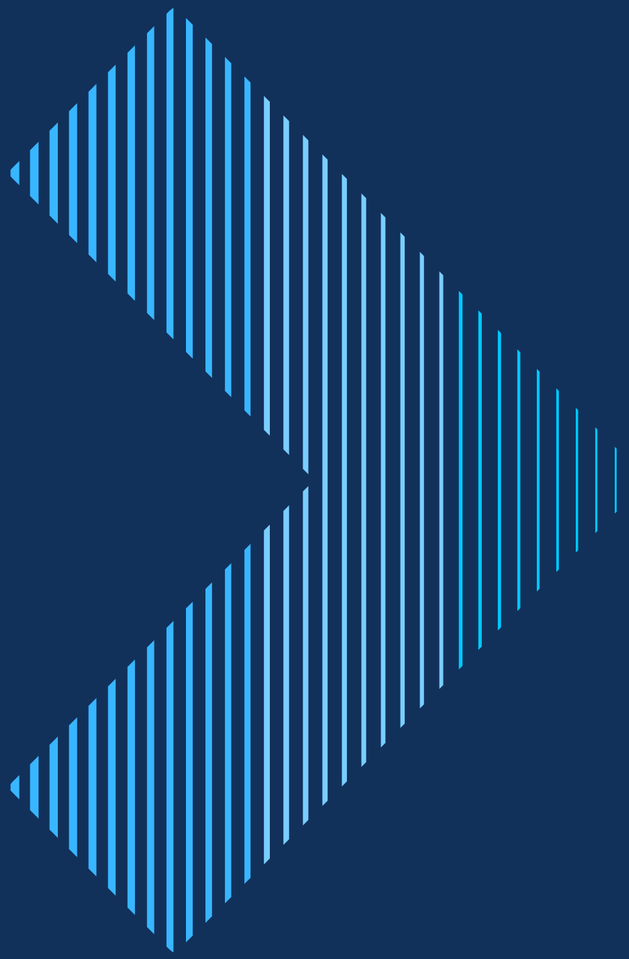
Thesis Defence

Federico Perenno

Master's Degree in Electronic Engineering

Prof. Mario Roberto Casu

Dr. Luca Urbinati



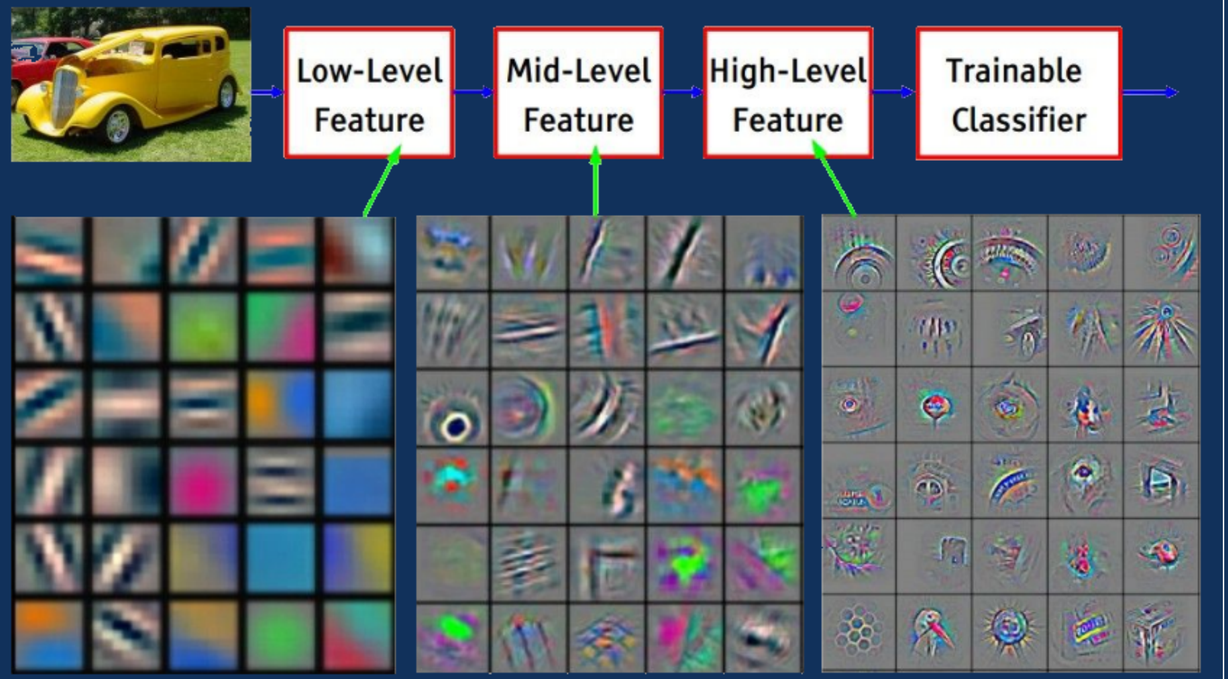
Introduction

- Convolutional Neural Networks (CNNs) are at the foundation of many Artificial Intelligence (AI) applications.

- The goal of this thesis is to **design** accelerators that perform the **2D convolution** algorithm, to **integrate** them on an **SoC** and to **prototype** them on an **FPGA** leveraging High-Level Synthesis (**HLS**) and Embedded Scalable Platform (**ESP**).

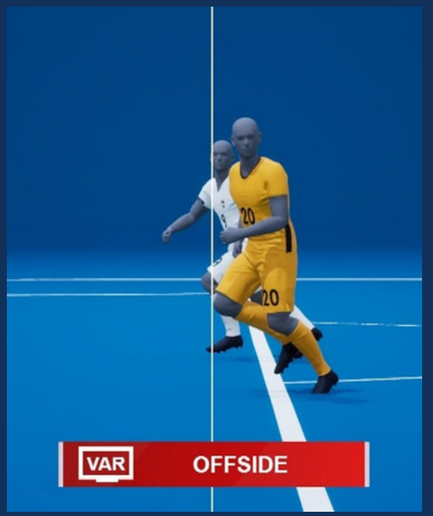
Convolutional Neural Networks

Convolutions are performed at each layer of the neural network to generate progressively higher-level abstractions of input data, called **features**.



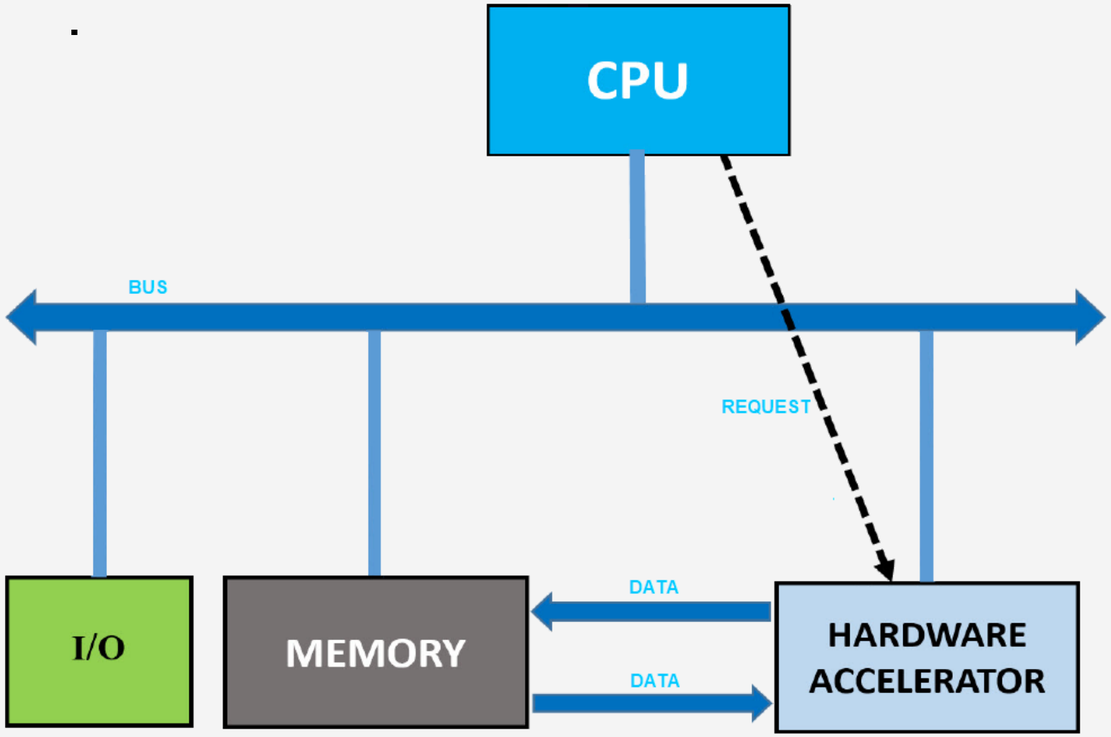
These features are exploited in **real time** for many progressive applications, including:

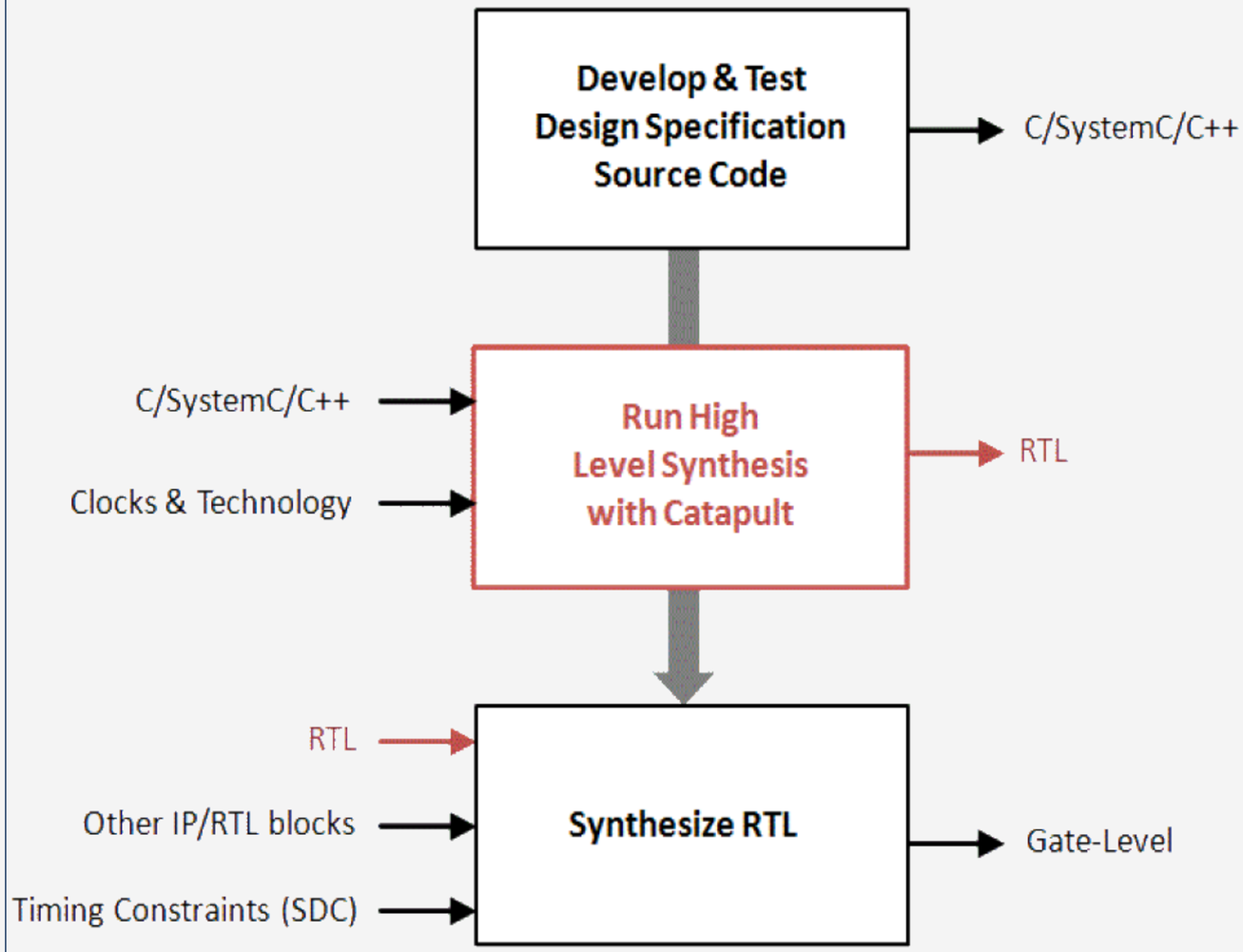
- **Drone Surveillance**.
- **Autonomous Driving**.
- **Offside Technology** in football.



Hardware Accelerators and HLS

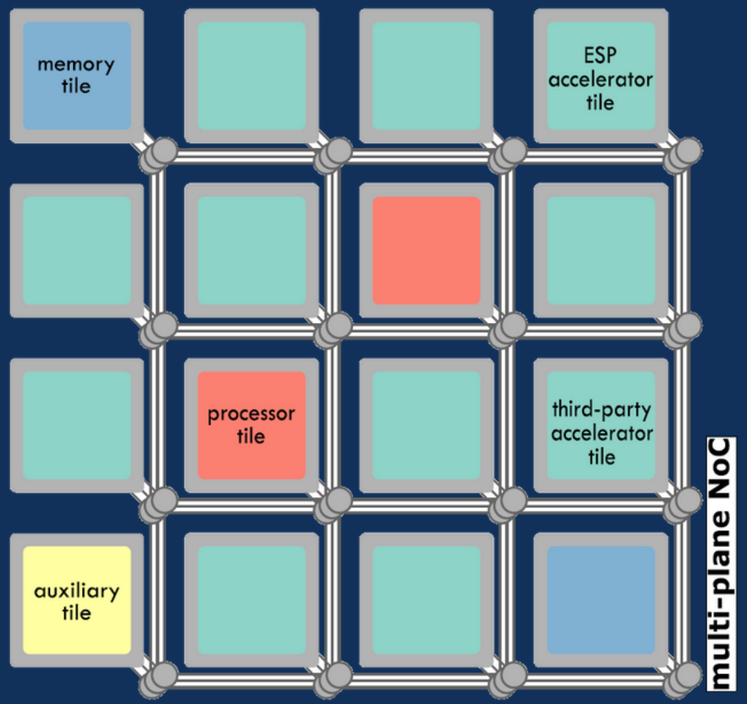
- **Hardware accelerators** speed up the computation of CNNs.
- **HLS** facilitates their design and synthesis.

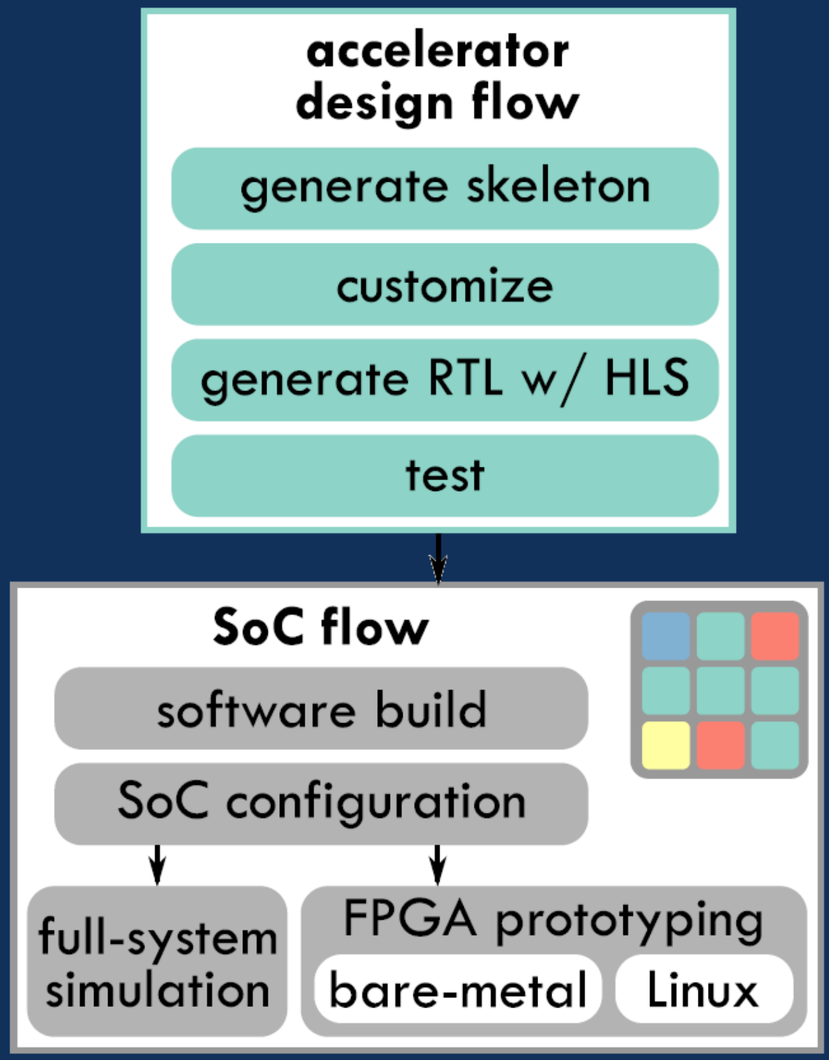




Embedded Scalable Platform

- **ESP** is an **open-source SoC platform**, developed at **Columbia University**.
- ESP facilitates the **design** of a new accelerator, its **integration** in a **tile-based SoC** and **FPGA prototyping**.





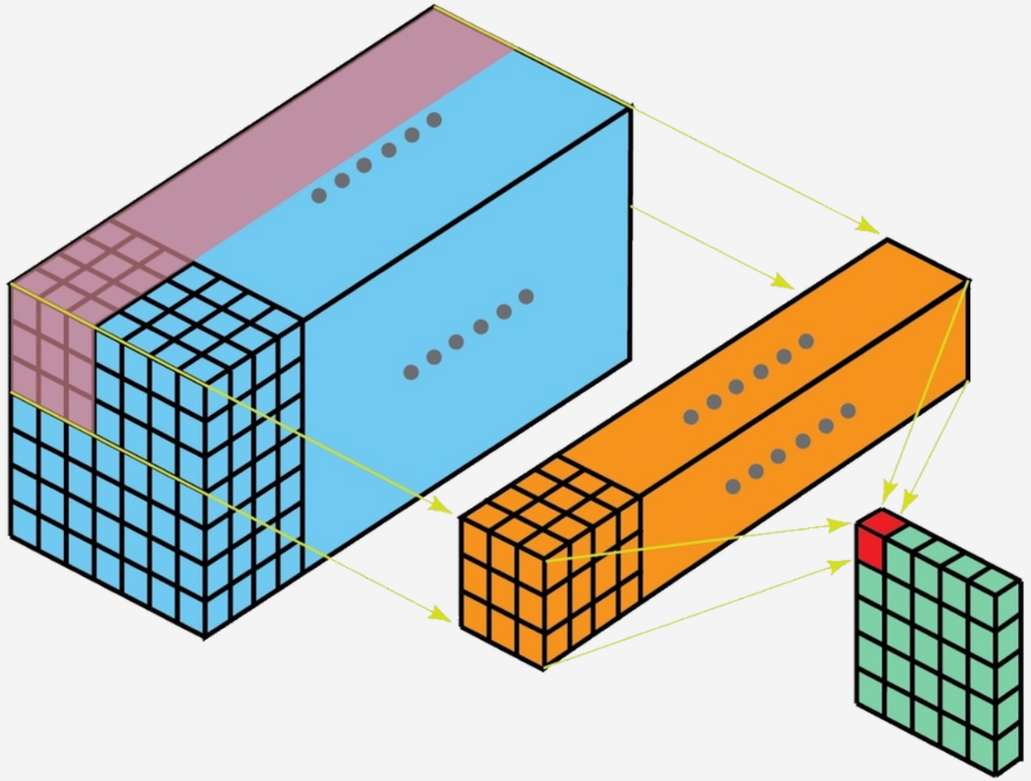
2D-Convolution Accelerator

Our accelerator is able to perform the 2D-convolution between :

- An **18x18x32 input/features tensor**.
- **32** distinct **7x7x32 filters (weights tensor)**.

The accelerator execution includes four distinct phases:

- **Configuration**
- **Load**
- **Compute**
- **Store.**

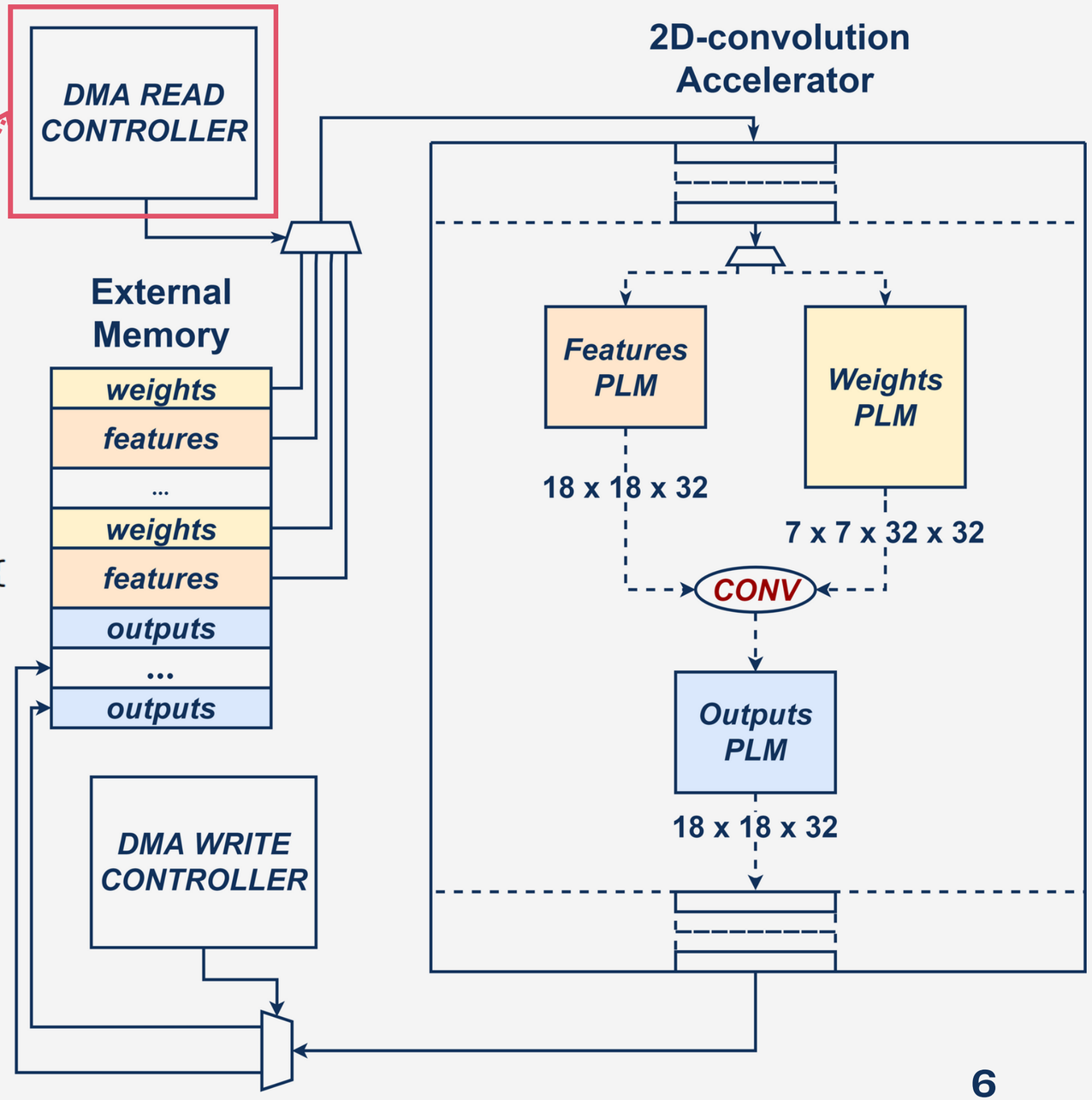


Sequential Architecture

Load Phase

```
dma_read_info = {dma_read_data_index,
                 dma_read_data_length,
                 DMA_SIZE};
dma_read_ctrl.write(dma_read_info);
```

```
LOAD_WEIGHTS_LOOP:
for (uint16_t i = 0; i < FILTERS_SIZE; i++){
    data = dma_read_chnl.read();
    plm_f.data[i] = data;
}
LOAD_FEATURES_LOOP:
for (uint16_t i = 0; i < INPUT_SIZE; i++){
    if (padding_needed) // Zero values
        data = 0;
    else                // Non-zero values
        data = dma_read_chnl.read();
    plm_in.data[i] = data;
}
```

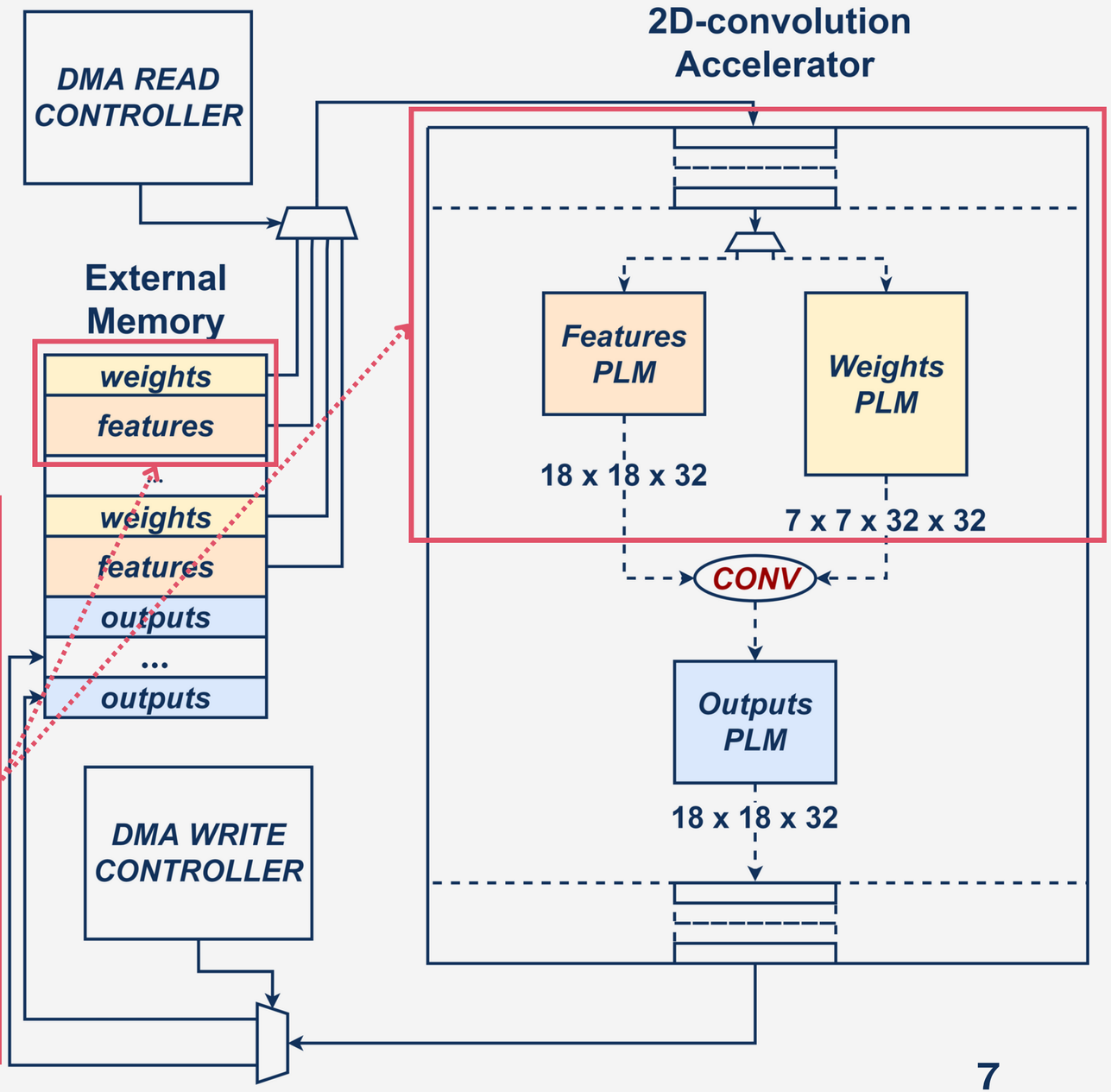


Sequential Architecture

Load Phase

```
dma_read_info = {dma_read_data_index,
                 dma_read_data_length,
                 DMA_SIZE};
dma_read_ctrl.write(dma_read_info);
```

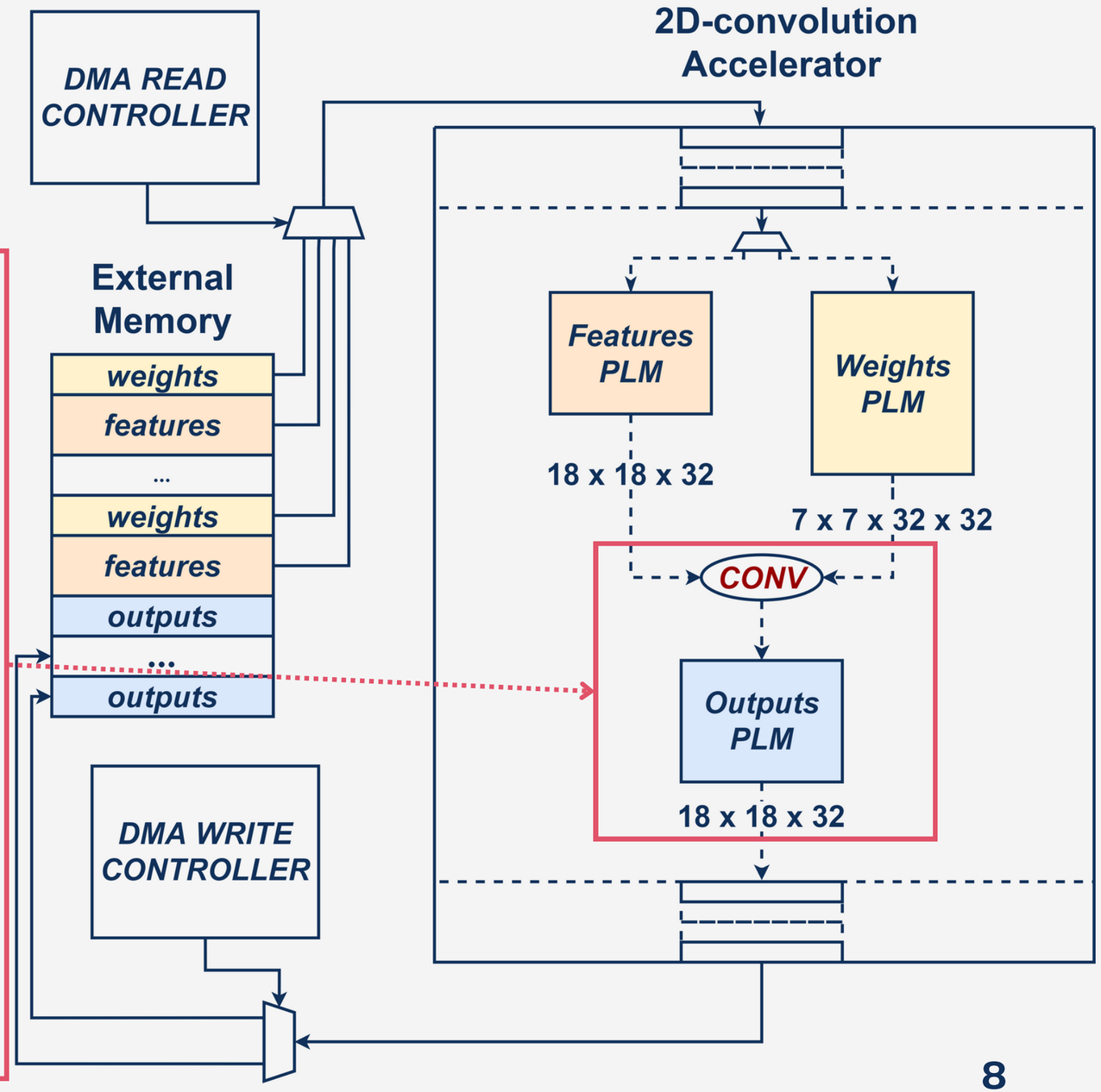
```
LOAD_WEIGHTS_LOOP:
for (uint16_t i = 0; i < FILTERS_SIZE; i++){
    data = dma_read_chnl.read();
    plm_f.data[i] = data;
}
LOAD_FEATURES_LOOP:
for (uint16_t i = 0; i < INPUT_SIZE; i++){
    if (padding_needed) // Zero values
        data = 0;
    else                // Non-zero values
        data = dma_read_chnl.read();
    plm_in.data[i] = data;
}
```



Sequential Architecture

Compute Phase

```
for (fl = 0; fl < FILT; fl++) {
 for (k = 0; k < N_C; k++)      {
  for (i = 0; i < N_W_OUT; i++) {
   for (j = 0; j < N_H_OUT; j++) {
     acc = 0; x = i* stride; y = j* stride;
     for (m = 0; m < KERN; m++) {
       for (n = 0; n < KERN; n++) {
         i_f = fl* kern*kern*n_c + k*kern*
                   kern + m* kern + n;
         i_in = n_w* n_h* k + x* n_w + y;
         acc+= plm_f.data[i_f] *  // MAC
               plm_in.data[i_in];
         y++; }
       x++;
       y = j* stride; }
     i_out = n_w_out* n_h_out* fl
             + i*n_w_out + j;
     buf_acc.data[i][j] += acc;
     if (k == n_c - 1)
       plm_out.data[i_out] = buf_acc.data[i][j];
}}}}
```

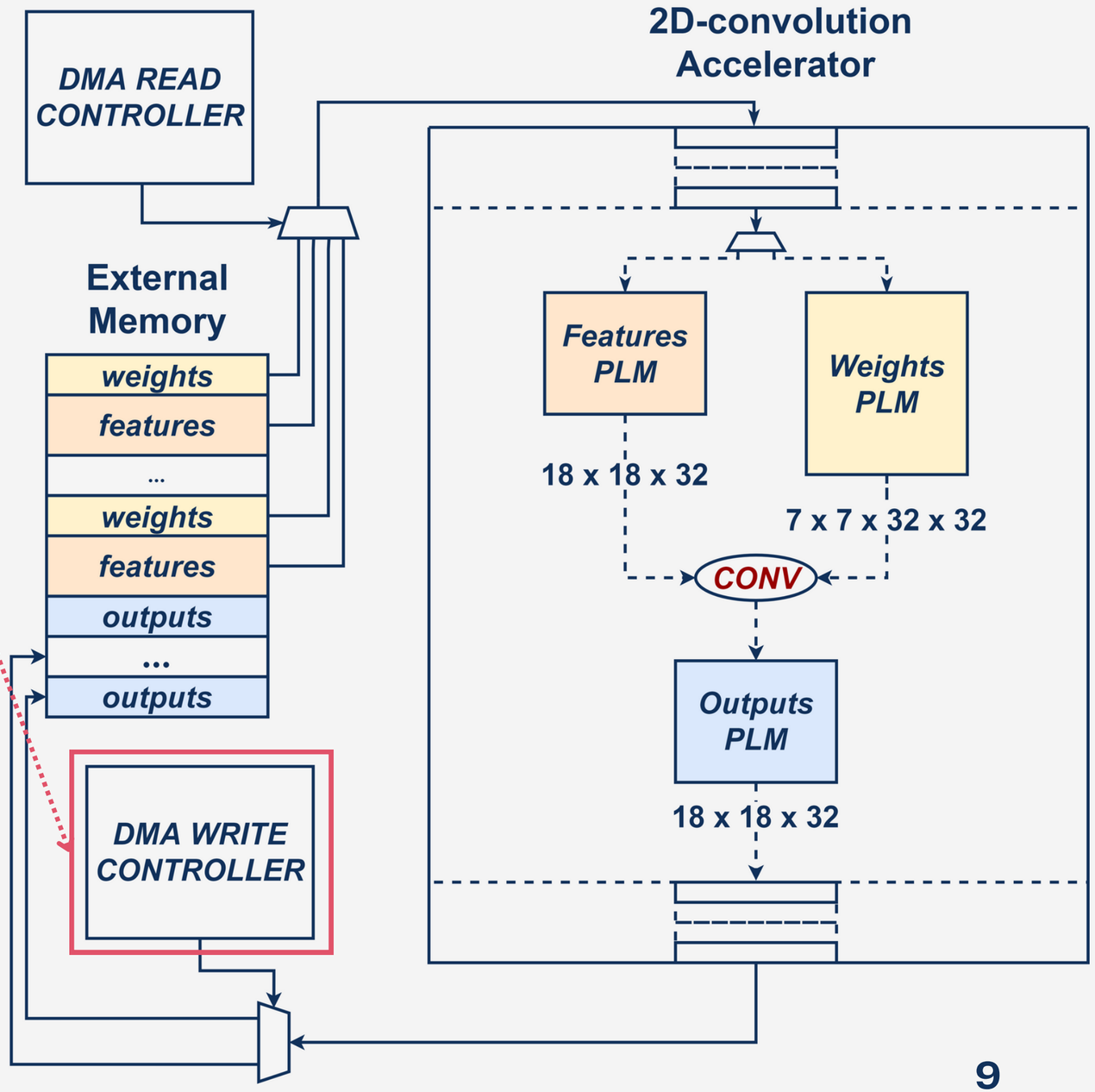


Sequential Architecture

Store Phase

```
dma_write_info = {dma_write_data_index,
                  dma_write_data_length,
                  DMA_SIZE};
dma_write_ctrl.write(dma_write_info);}
```

```
STORE_LOOP:
for (uint16_t i = 0; i < OUTPUTS_SIZE; i++){
    data = plm_out.data[i];
    ac_int<DMA_WIDTH, false> data_ac;
    ac_int<32, false> DEADBEEF = 0xdeadbeef;
    data_ac.set_slc(32, DEADBEEF);
    data_ac.set_slc(0, data);
    dma_write_chnl.write(data_ac);
}
```

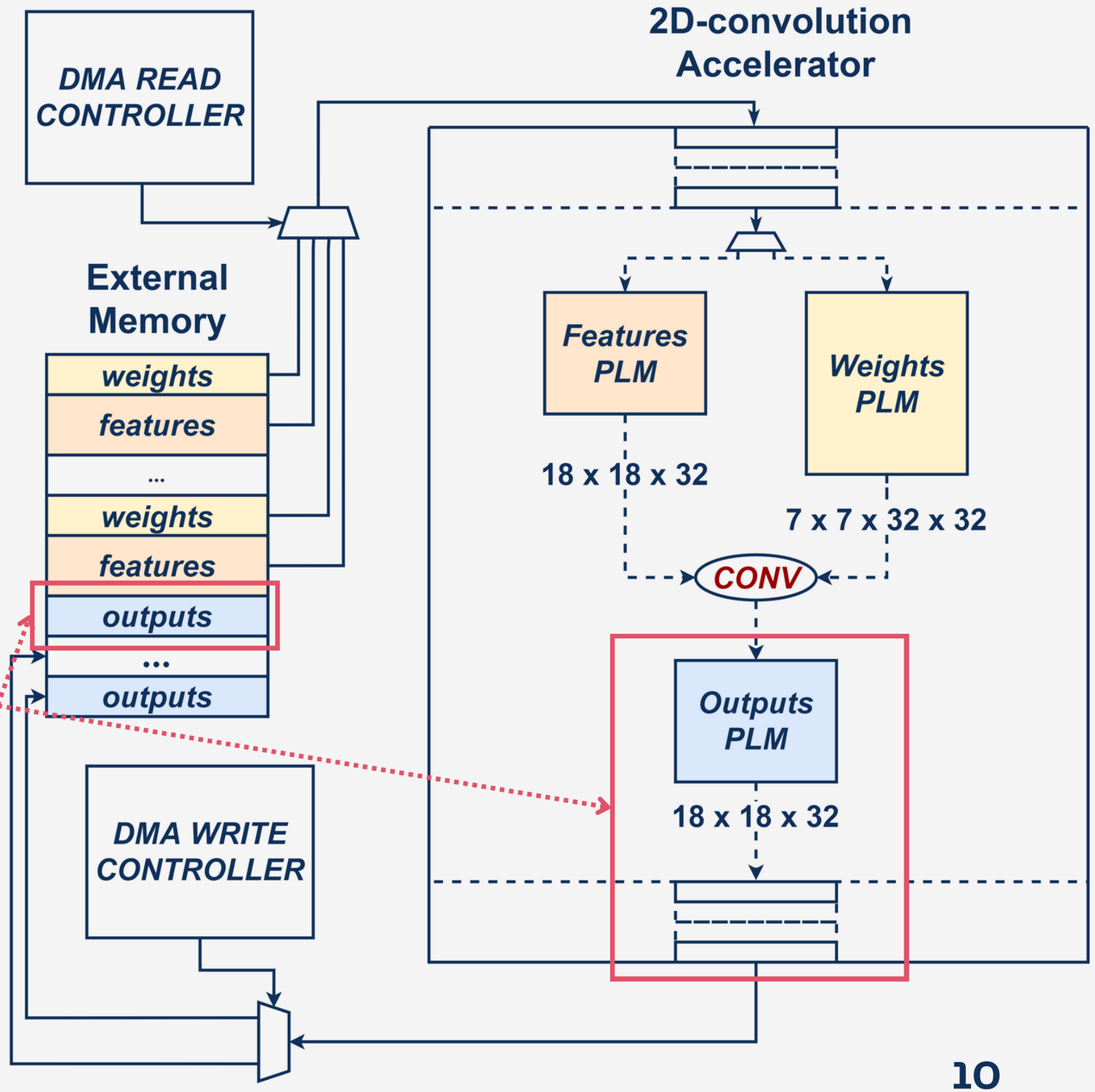


Sequential Architecture

Store Phase

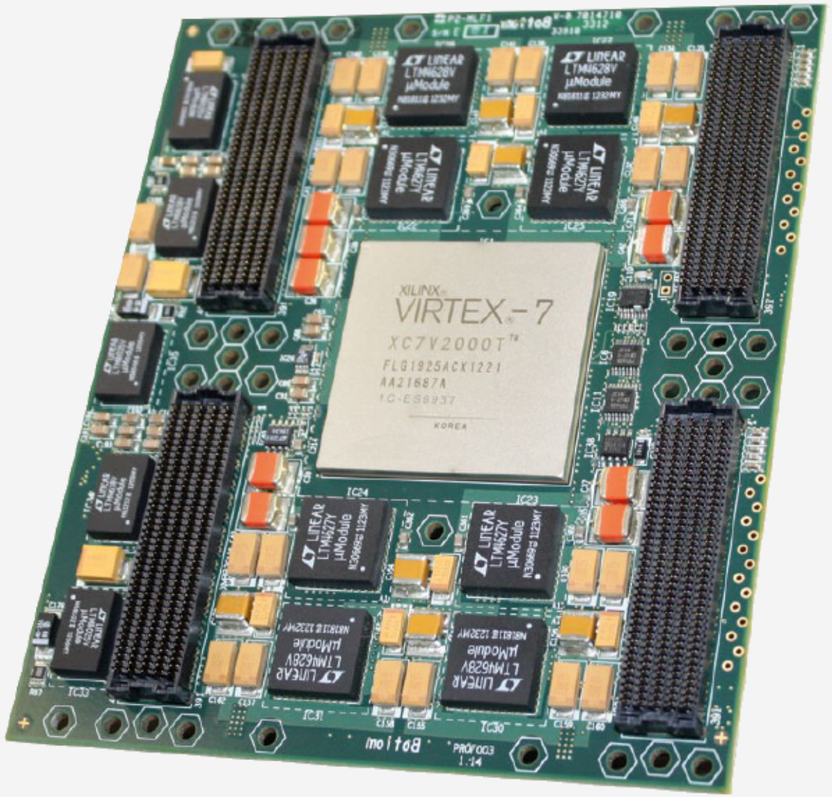
```
dma_write_info = {dma_write_data_index,
                  dma_write_data_length,
                  DMA_SIZE};
dma_write_ctrl.write(dma_write_info);}
```

```
STORE_LOOP:
for (uint16_t i = 0; i < OUTPUTS_SIZE; i++){
    data = plm_out.data[i];
    ac_int<DMA_WIDTH, false> data_ac;
    ac_int<32, false> DEADBEEF = 0xdeadbeef;
    data_ac.set_slc(32, DEADBEEF);
    data_ac.set_slc(0, data);
    dma_write_chnl.write(data_ac);
}
```

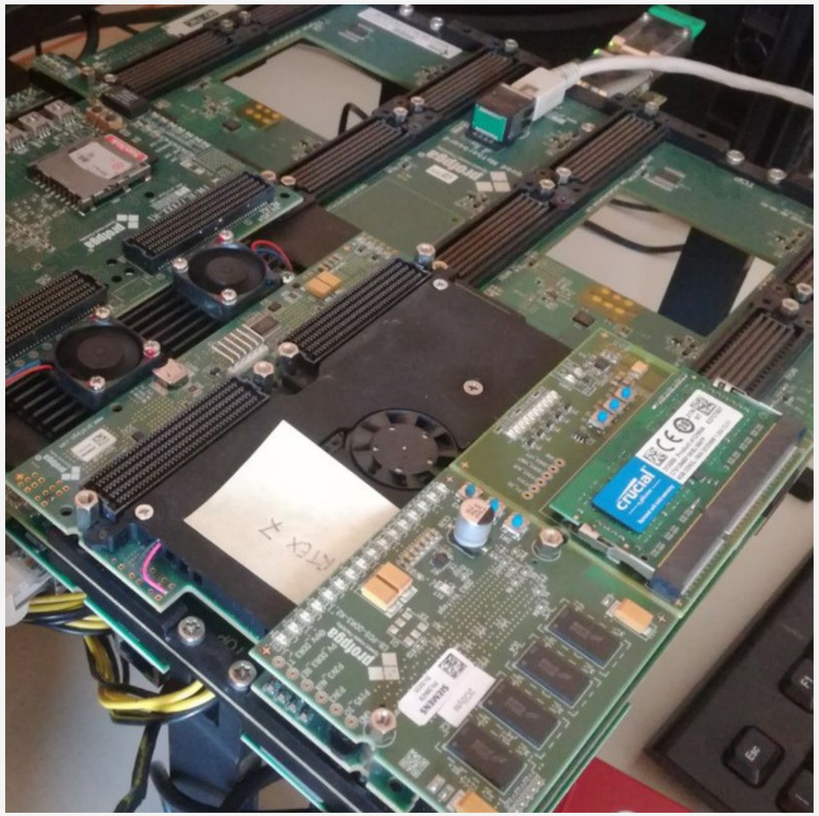


FPGA Setup

- The **Xilinx Virtex 7 XCV2000T proFPGA** module is attached to the **ProFPGA quad Motherboard**.

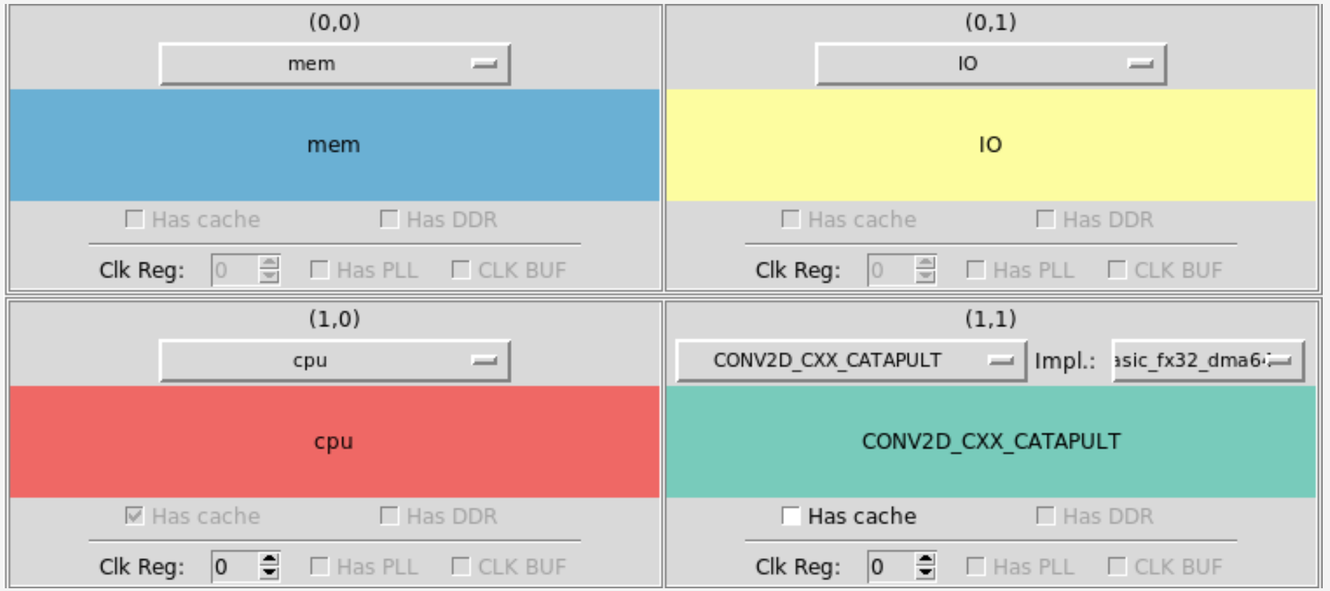


- The FPGA is connected via **ethernet** to a remote server (**host**).



Hardware v. Software

- Our 2D-convolution accelerator is integrated on an SoC with an **Ariane RISC-V CPU**, one **Memory** tile and one **I/O** tile.
- The SoC is synthesized and prototyped on the **proFPGA XC7V2000T** board.

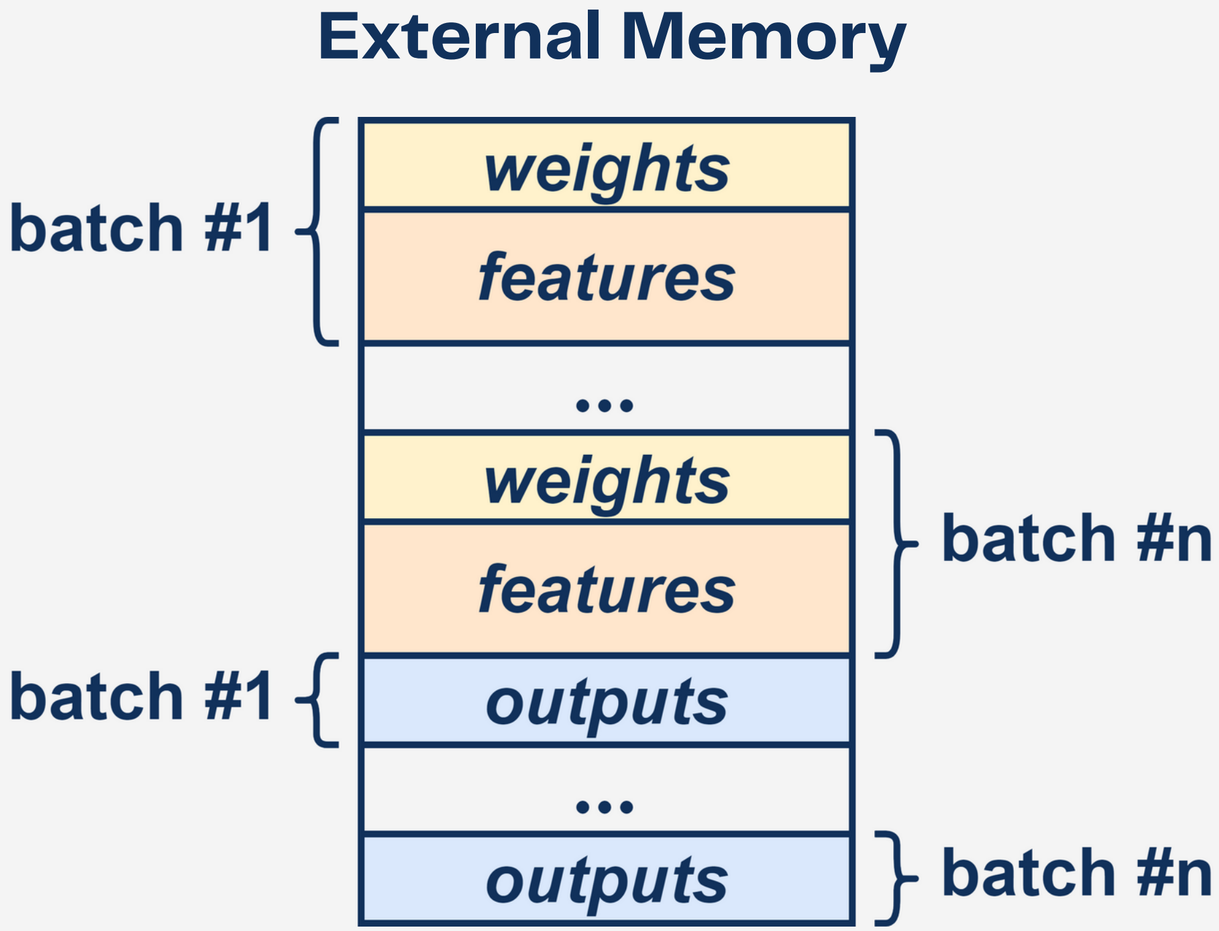


Features	Filters	Proc. Lat. [ms]	Acc. Lat. [ms]	Reduc.
5×5×3	3×3×3×3	0.5205	0.0184	96.47 %
18×18×3	7×7×3×3	3358.21	145.81	95.65 %

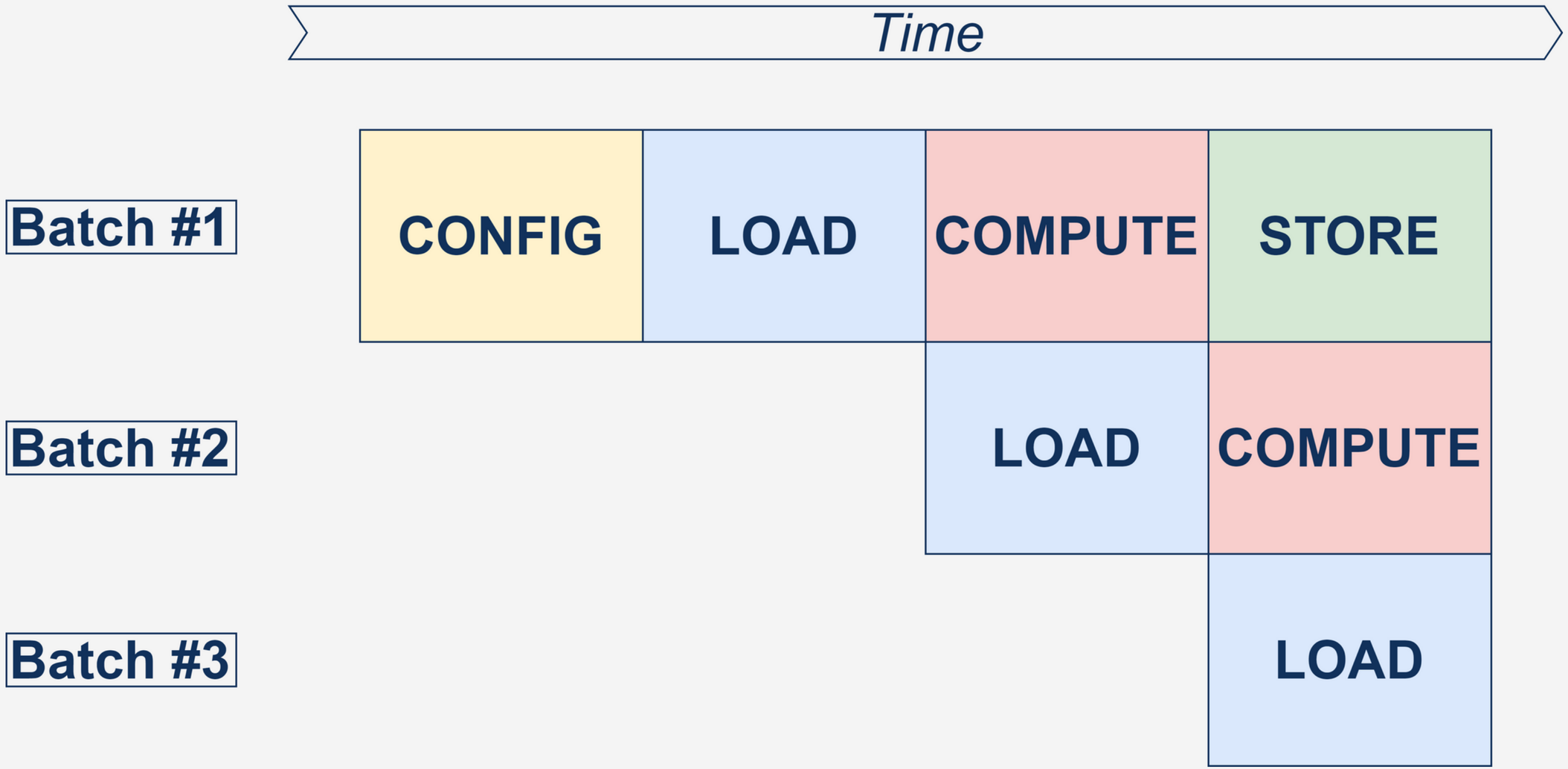
Hierarchical Architecture

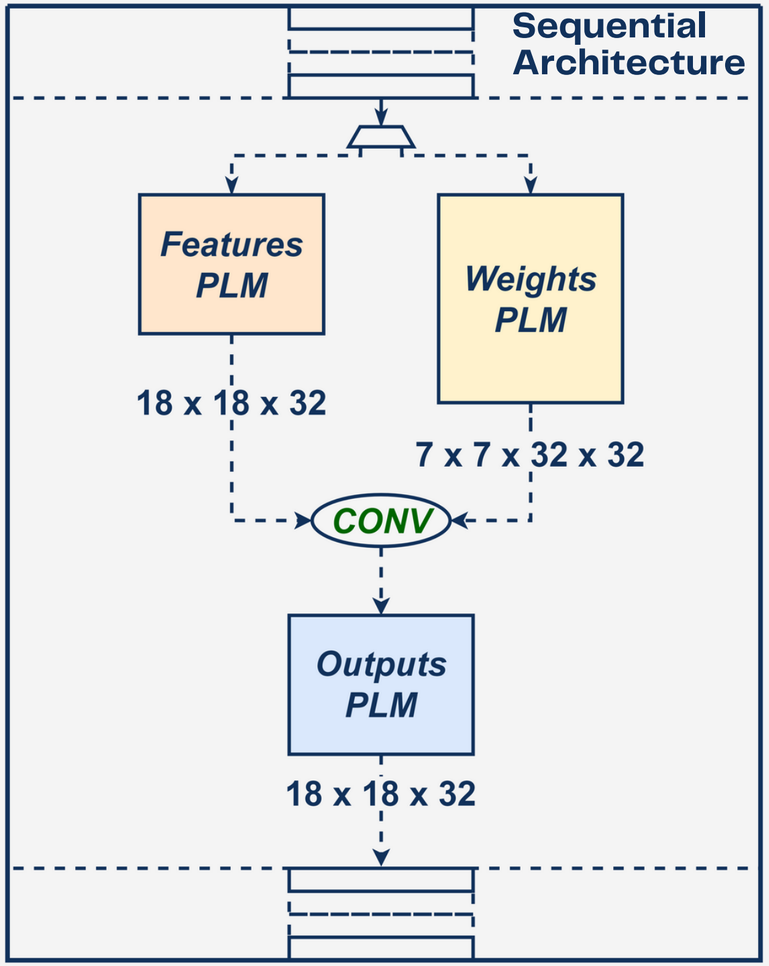
- The accerator phases are described in distinct **functions/blocks** which are able to run **concurrently**, as long as **data dependencies** are respected.

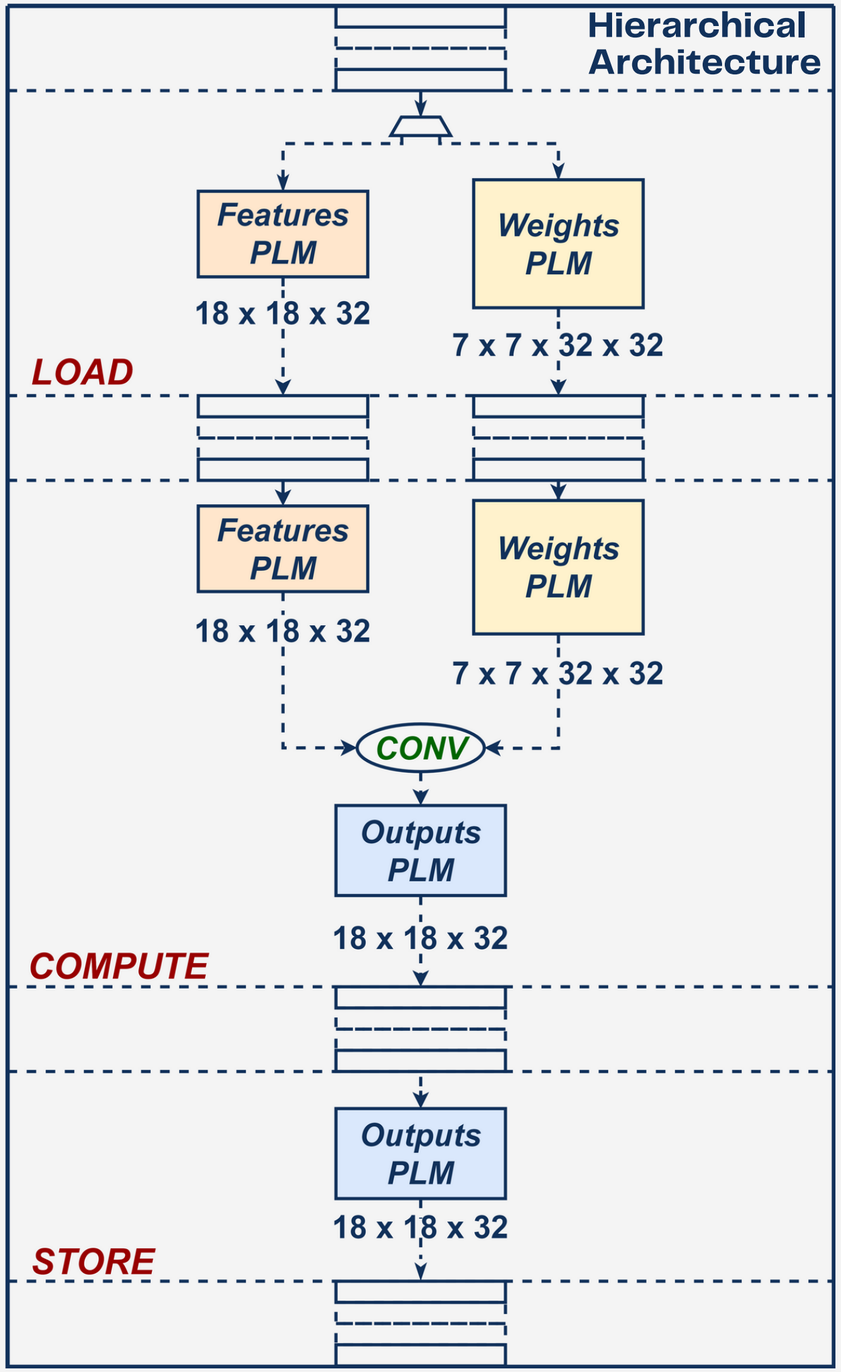
- Blocks can run in parallel only if weights and features are organized in **independent batches** of data.



Hierarchical Architecture




Sequential Architecture
Features PLM
Weights PLM
18 x 18 x 32
7 x 7 x 32 x 32
CONV
Outputs PLM
18 x 18 x 32


Hierarchical Architecture
Features PLM
Weights PLM
18 x 18 x 32
7 x 7 x 32 x 32
LOAD
Features PLM
Weights PLM
18 x 18 x 32
7 x 7 x 32 x 32
CONV
Outputs PLM
18 x 18 x 32
COMPUTE
Outputs PLM
18 x 18 x 32
STORE

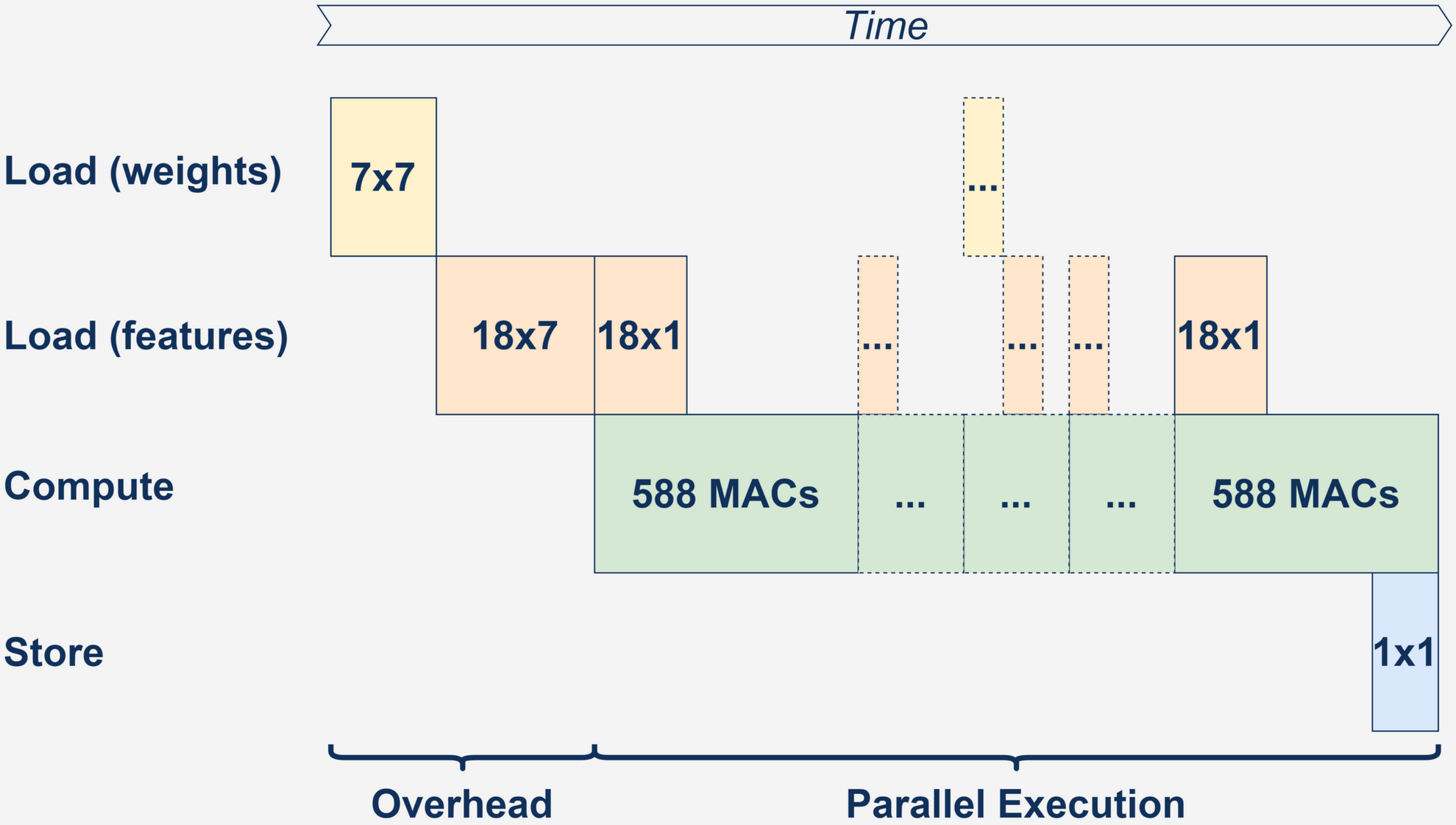
Sequential v. Hierarchical

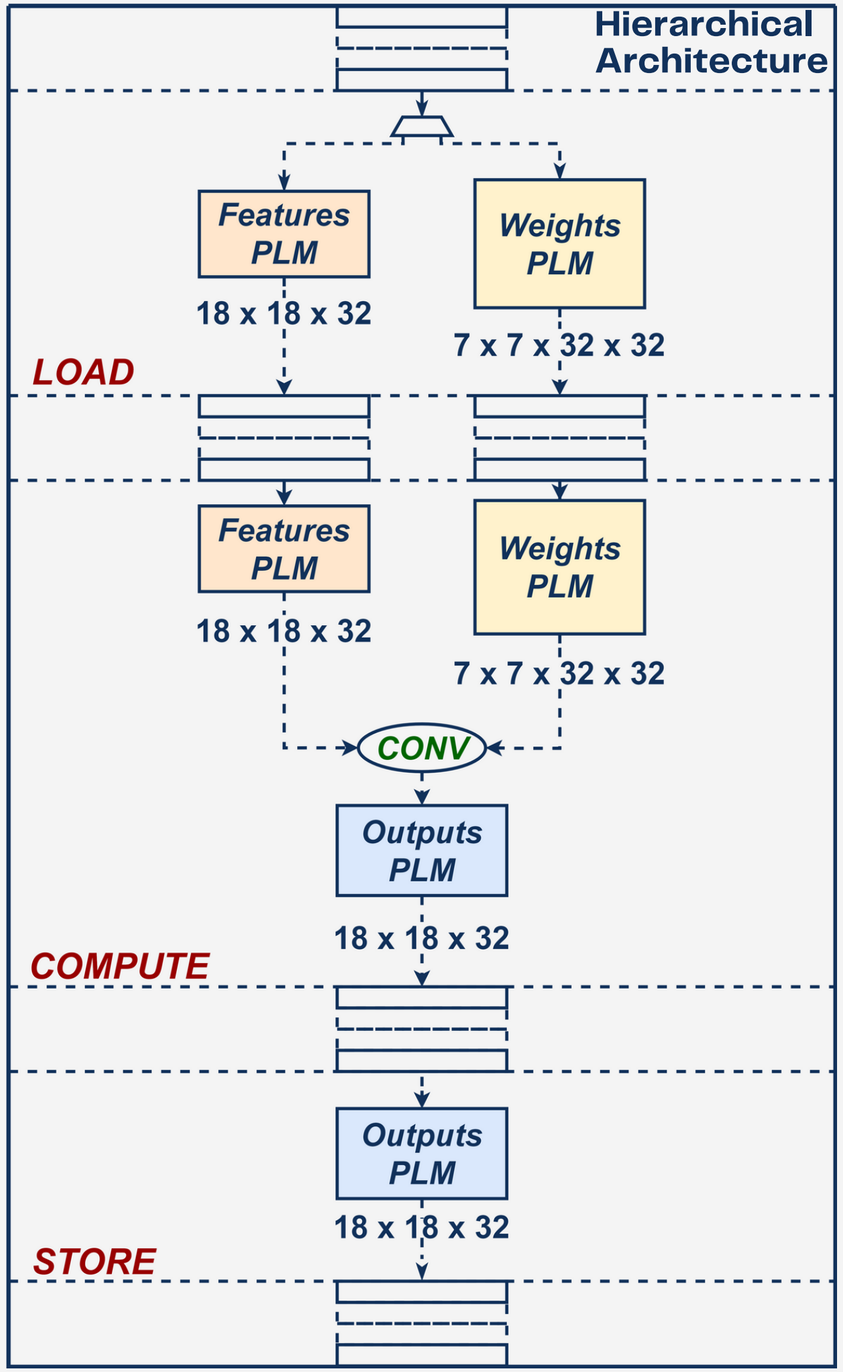
- 2D convolution between **18×18×32 input** tensor and **7×7×32x32 filters**, for different numbers of **batch iterations**.

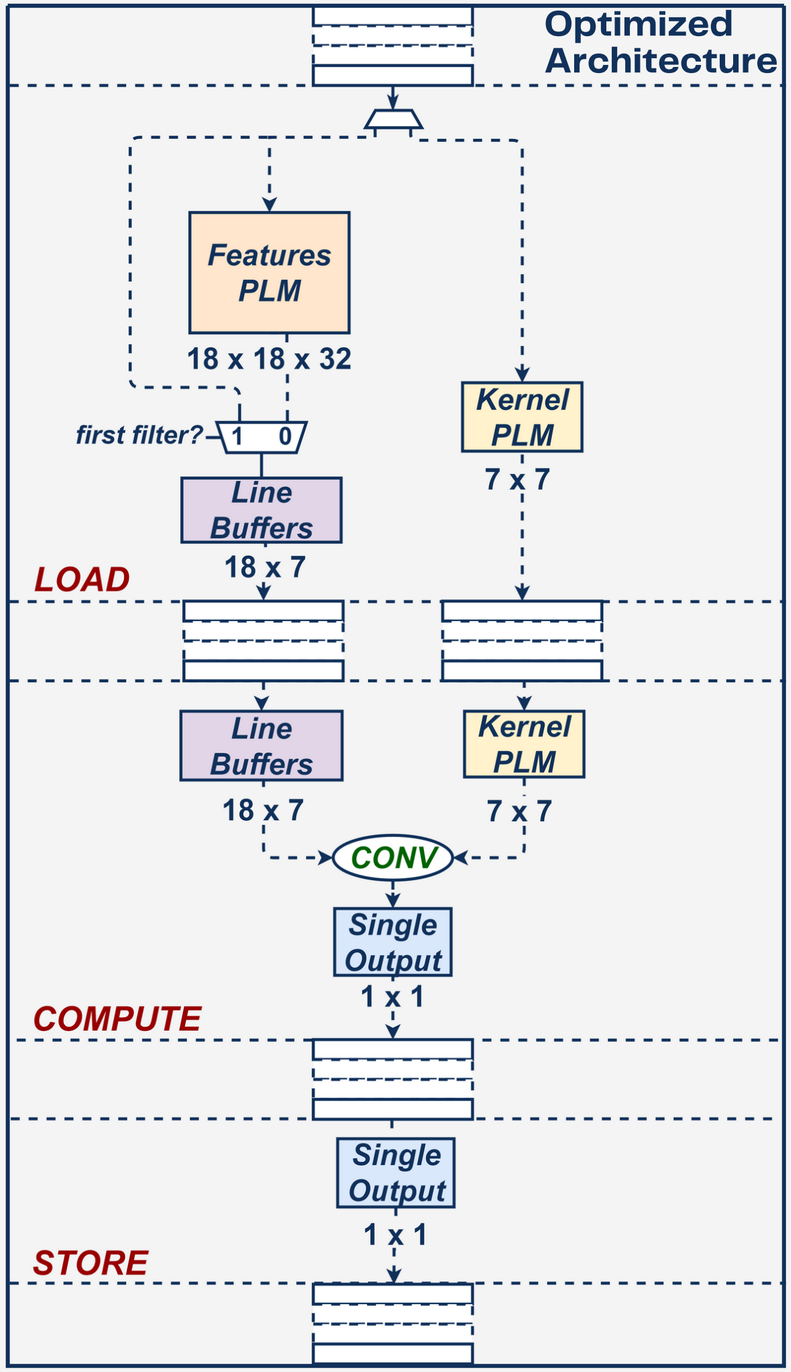
# Batches	Architecture	Estimated Area		Latency $[ms]$	
1	sequential	83268.3	-	145.81	-
1	hierarchical	91576.3	+10%	145.81	0%
8	sequential	83268.3	-	1166.47	-
8	hierarchical	91576.3	+10%	1156.54	-0.85%
16	sequential	83268.3	-	2332.95	-
16	hierarchical	91576.3	+10%	2311.67	-0.91%

- For **multiple batch iterations** the hierarchical architecture reduces **latency** at the cost of **area**.
- For a **single batch** the hierarchical architecture provides **no advantage**.

Optimized Architecture




Hierarchical Architecture
Features PLM
18 x 18 x 32
Weights PLM
7 x 7 x 32 x 32
LOAD
Features PLM
18 x 18 x 32
Weights PLM
7 x 7 x 32 x 32
CONV
Outputs PLM
18 x 18 x 32
COMPUTE
Outputs PLM
18 x 18 x 32
STORE


Optimized Architecture
Features PLM
18 x 18 x 32
first filter?
1
0
Line Buffers
18 x 7
Kernel PLM
7 x 7
LOAD
Line Buffers
18 x 7
Kernel PLM
7 x 7
CONV
Single Output
1 x 1
COMPUTE
Single Output
1 x 1
STORE

FPGA Resource Usage

- All three implementations of our 2D–Convolution accelerator have been integrated in an **SoC**.
- The table shows the FPGA resource usage results obtained by the **Xilinx Vivado** utilization report after **place and route**.
- Line Buffers, Kernel PLM, and Output PLM have all been synthesized as registers in the **optimized architecture**, hence the reduction in **block RAM** usage at the cost of the additional **slice logic**.

Architecture	Slice Logic [%]	Block RAM [%]	DSP [%]
Sequential	20.31	39.32	2.31
Hierarchical	20.33 (+ 0.02)	50.81 (+ 11.49)	2.31 (+ 0.00)
Optimized	22.70 (+ 2.41)	25.70 (- 13.62)	2.13 (- 0.18)

Latency

- Latency comparison between **unoptimized** and **optimized** architectures of our 2D-convolution accelerator with different parameters and a **single batch**.

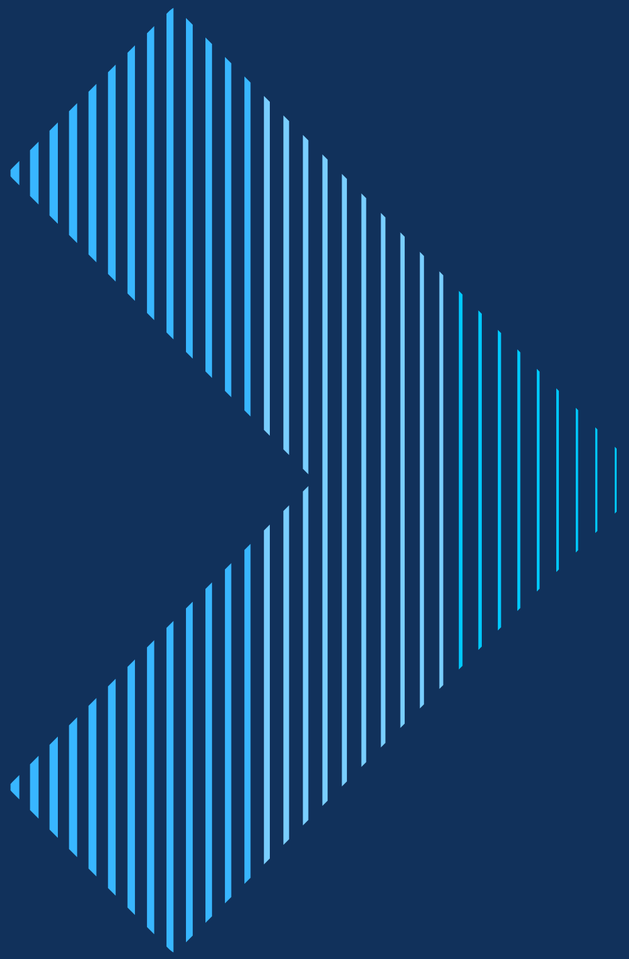
- In the **optimized architecture**, the different accelerator phases are executed in a **transparent** way.

Arch.	Features	Filters	Same	Stride	Latency [*clks*]
Seq./Hier.	5×5×3	3×3×3×3	0	1	918
Opt.	5×5×3	3×3×3×3	0	1	760 (-17.21%)
Seq./Hier.	5×5×3	3×3×3×3	1	1	2335
Opt.	5×5×3	3×3×3×3	1	1	2063 (-11.65%)
Seq./Hier.	18×18×32	7×7×32×32	0	2	1,868,038
Opt.	18×18×32	7×7×32×32	0	2	1,806,518 (-3.29%)
Seq./Hier.	18×18×32	7×7×32×32	0	1	7,290,502
Opt.	18×18×32	7×7×32×32	0	1	7,225,526 (-0.90%)

Future Work

- If we were to **parallelize** the execution of the datapath by **unrolling** completely (32 times) the for-loop on the output channels we would obtain a much more significant latency reduction.

Arch.	Features	Filters	Same	Stride	Latency [*clks*]	
Seq./Hier.	5×5×3	3×3×3×3	0	1	428	-
Opt.	5×5×3	3×3×3×3	0	1	270	-36.9%
Seq./Hier.	5×5×3	3×3×3×3	1	1	981	-
Opt.	5×5×3	3×3×3×3	1	1	709	-27.7%
Seq./Hier.	18×18×32	7×7×32×32	0	2	118,145	-
Opt.	18×18×32	7×7×32×32	0	2	56,625	-52.1%
Seq./Hier.	18×18×32	7×7×32×32	0	1	290,945	-
Opt.	18×18×32	7×7×32×32	0	1	225,969	-22.3%



Contribution

- This thesis contributes to expand the **ESP documentation** with three implementations of a **2D-convolution accelerator**.
- In addition, it can be used as a **detailed tutorial** by future hardware engineers for designing new loosely-coupled accelerators using **ESP**.

Politecnico di Torino

Q&A Session

Thank you for listening!