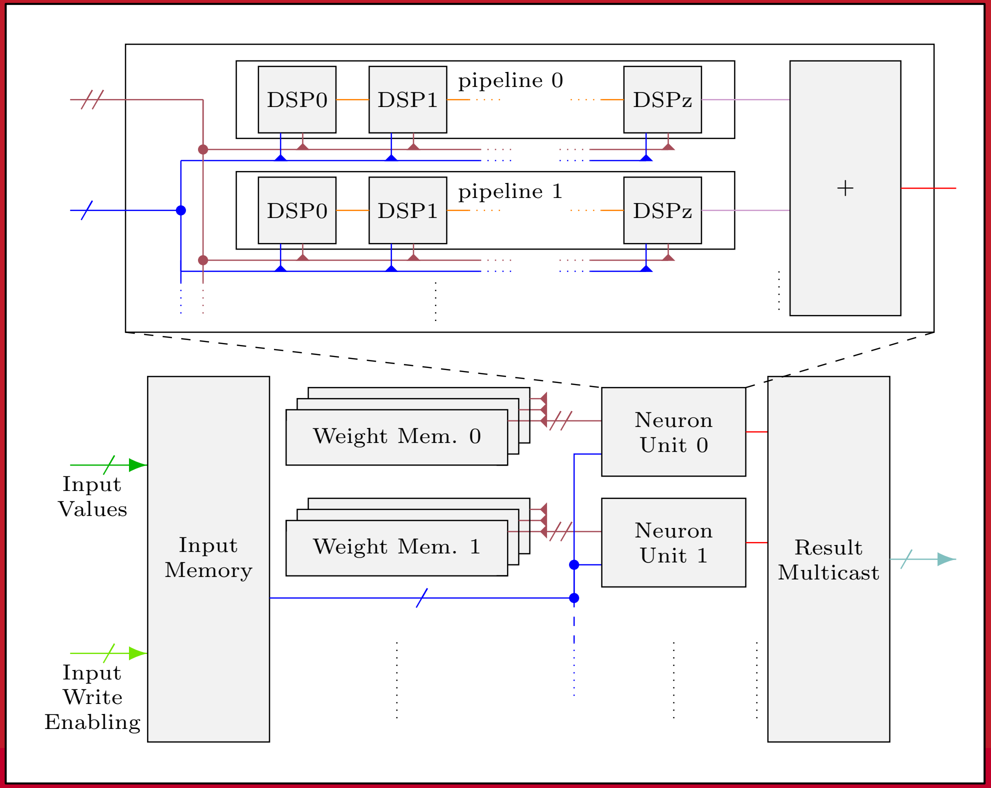
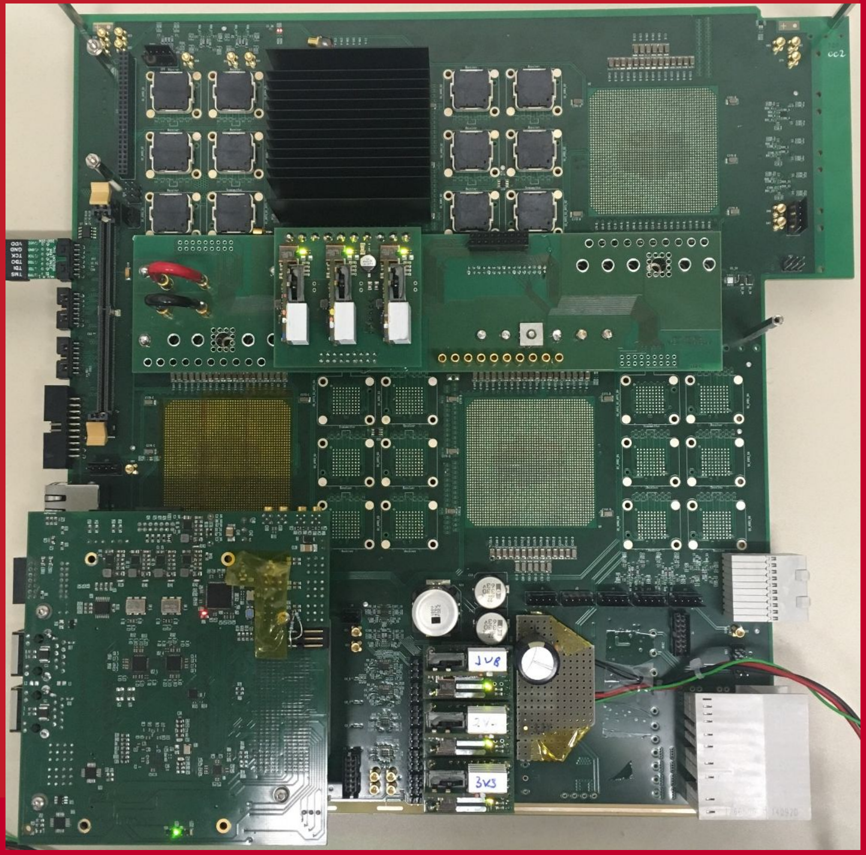
Development and implementation of deep neural networks close to sensors for object reconstruction and identification

Christian Schmitt (Mainz)





Cluster of Excellence PRiSMA+ Precision Physics, Fundamental Interactions and Structure of Matter

Bundesministerium für Bildung und Forschung

JG|U

Aim of the project in Mainz

- **Processing of detector data at extremely high rates**
  - Not possible to store data due to its size
  - Usage of GPUs not possible due to their too high latency
  - Data has to be processed and filtered locally, maybe directly at the corresponding sensors
- **Solution: deep neural networks** as replacement for iterative algorithms, that can be efficiently evaluated on **FPGAs**
- **Test environment: ATLAS L1 Trigger** (40 MHz rate)
- N.B: Complexity of actual networks used in ATLAS start at a few 10^3 multiplications

  (DNN to tag W-Bosons/Top-Quarks, ATL-PHYS-PUB-2017-004)

FPGAs ("Field Programmable Gate Array")

Xilinx US+ XCVU9P-2

- Programmable look-up tables (LUT, 1.2M)
  - Combinational logic
- Registers (FF, 2.4M)
  - Bit storage
- Programmable routing
  - LUT/register wiring
- Specialized units
  - DSPs (6840 'simple ALUs', MULT w/ subsequent ADD)
  - Block memory (~10MB)
  - …
- Lots of IO, computation; **predictable, ns-scale latencies**

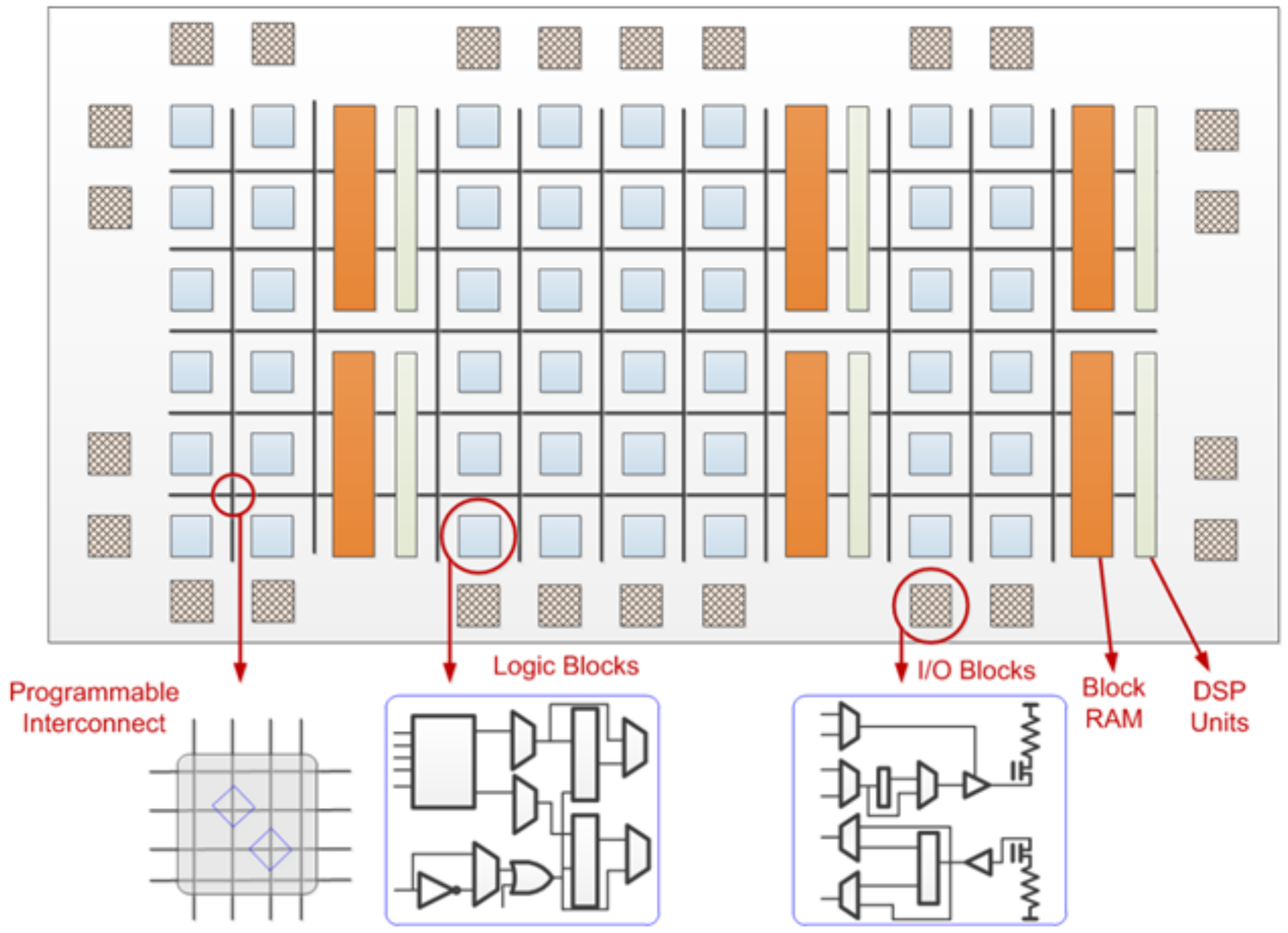
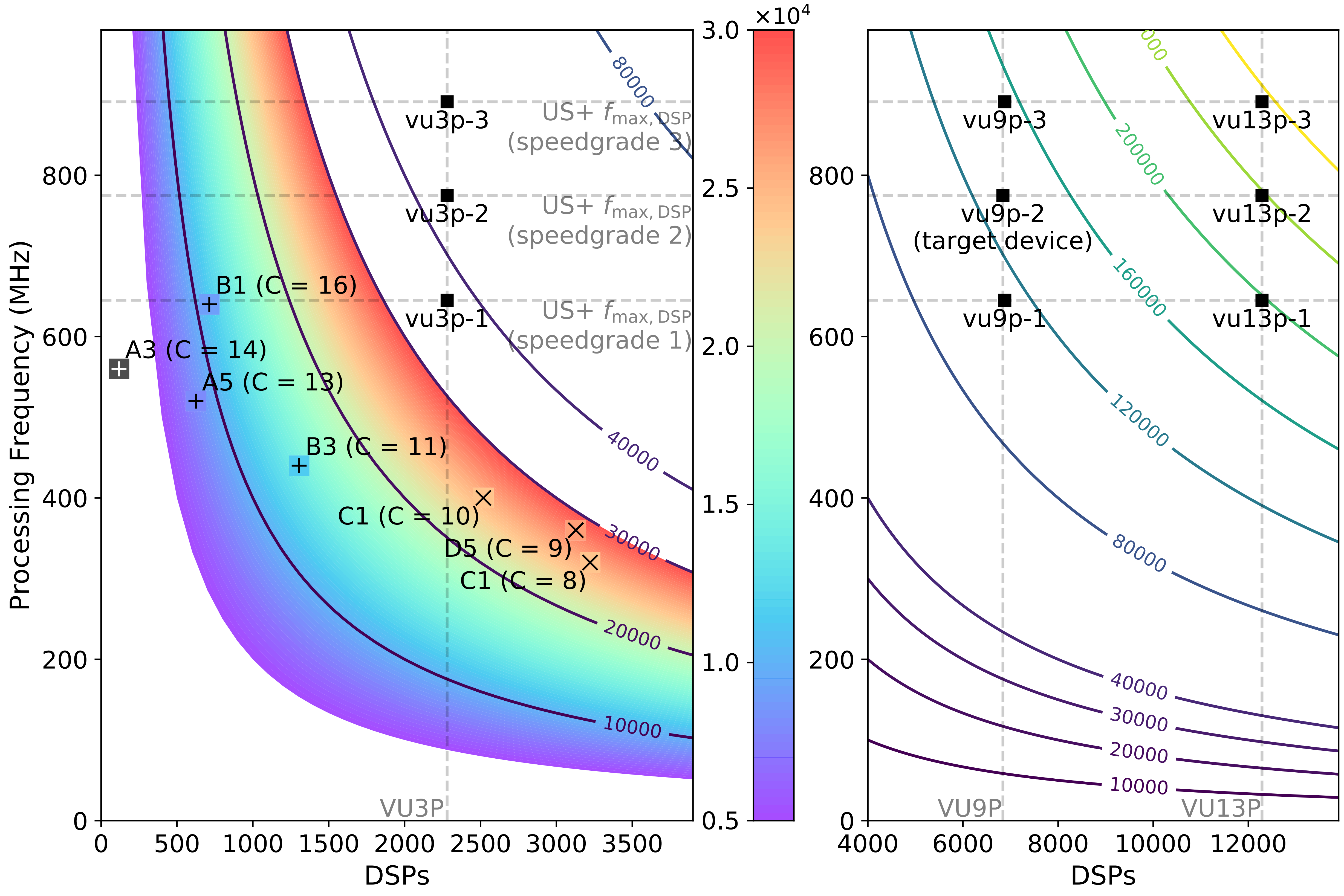


Image: https://medium.com/@ckyrkou/what-are-fpgas-c9121ac2a7ae

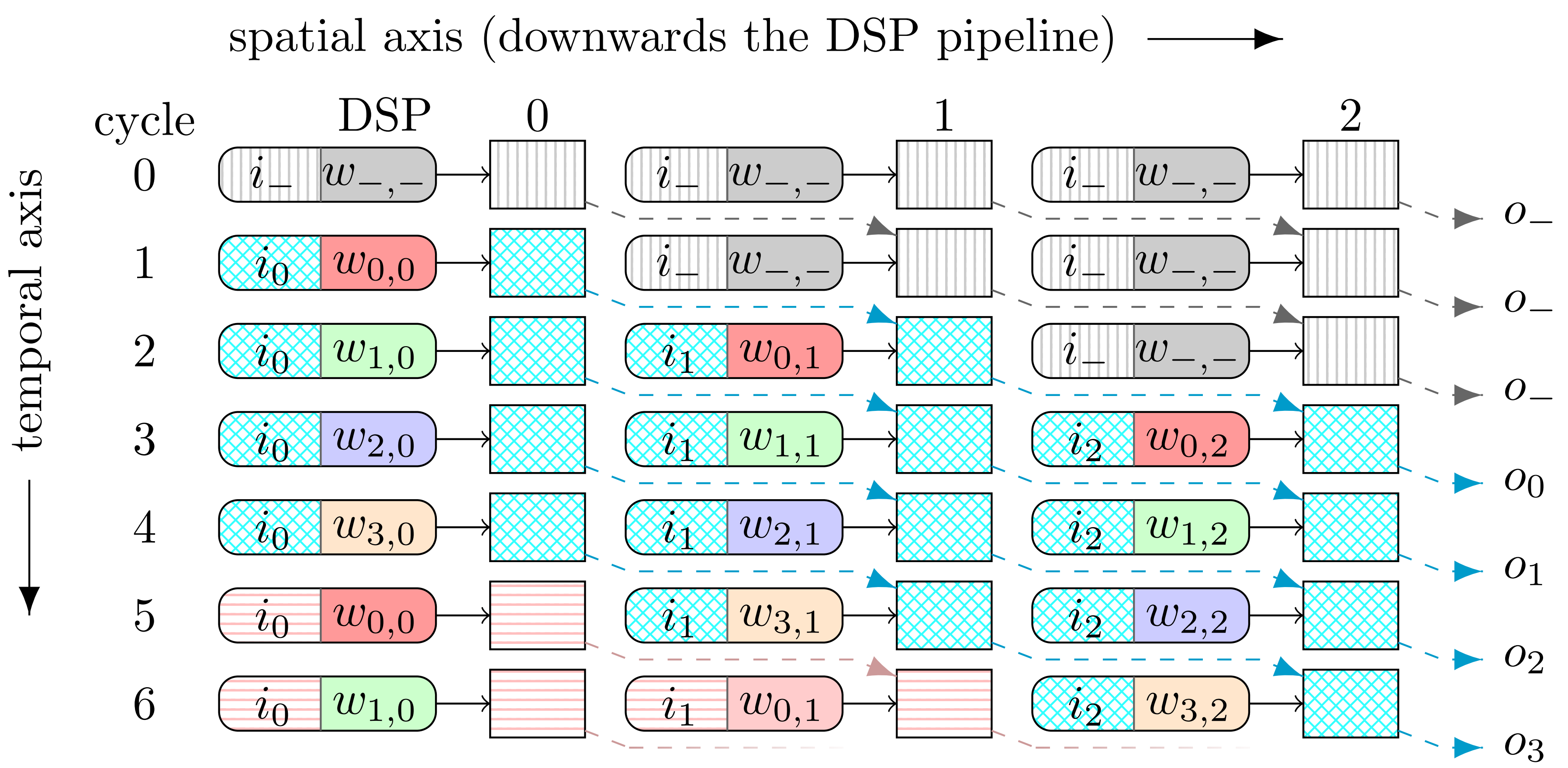
Maximum Network MACs assuming LHC Data Rate of 40MHz

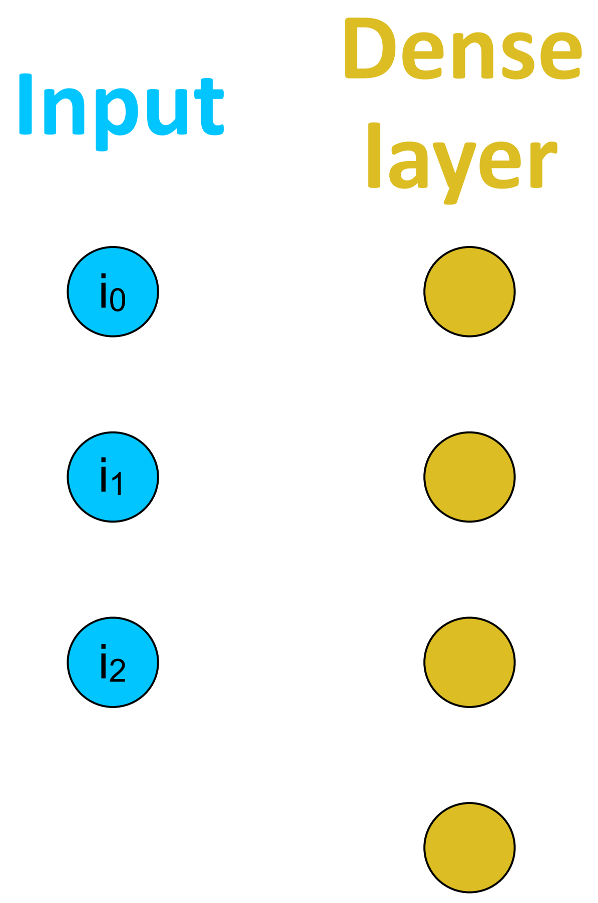


JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Fully connected layer design

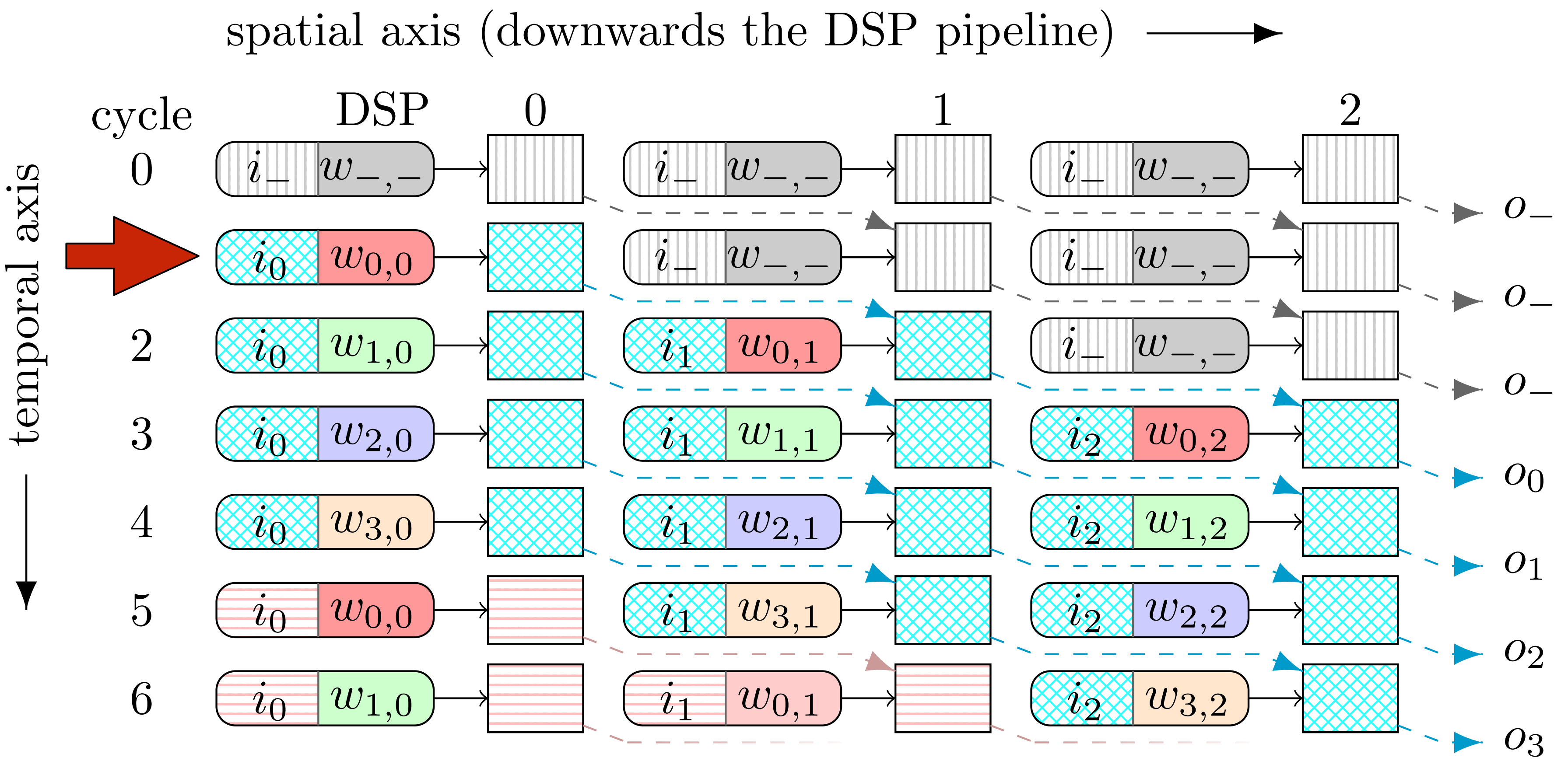
- Exploit: every neuron requires every input
  - Implement neuron processing in **DSP pipelines**
    - Inputs completely reusable
    - Only weight loading / fetching / multiplexing
    - **Simple design with easy parallelisation**

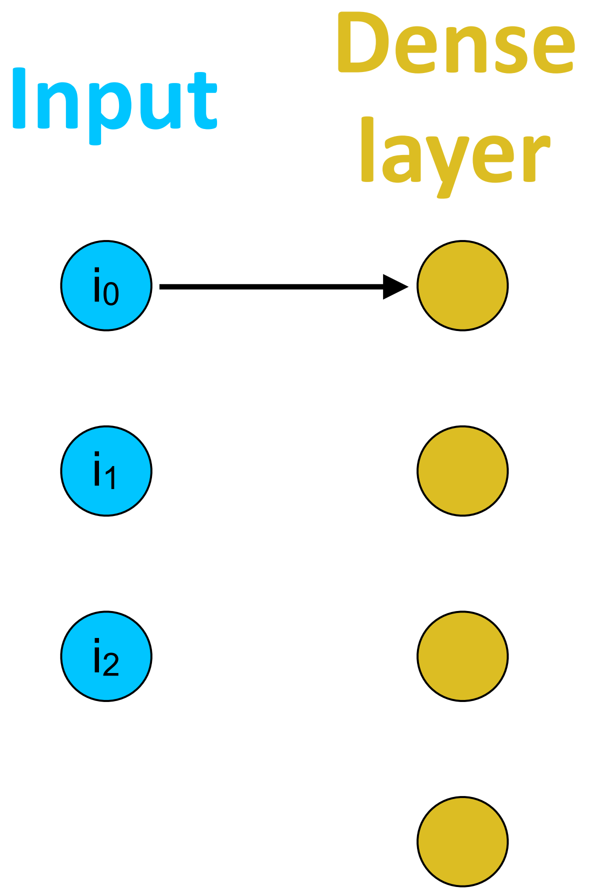




Fully connected layer design

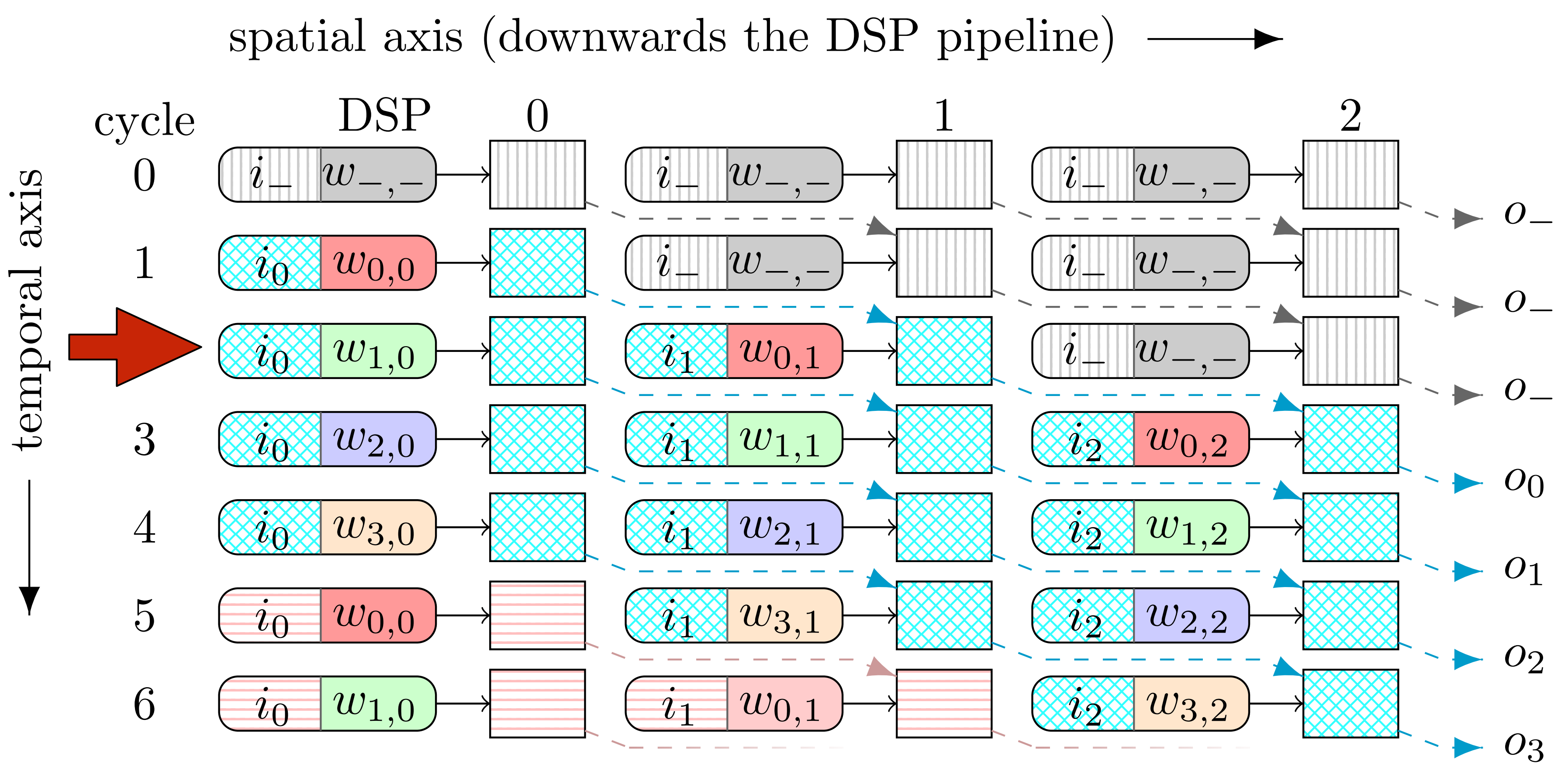
- Exploit: every neuron requires every input
  - Implement neuron processing in **DSP pipelines**
    - Inputs completely reusable
    - Only weight loading / fetching / multiplexing
    - **Simple design with easy parallelisation**

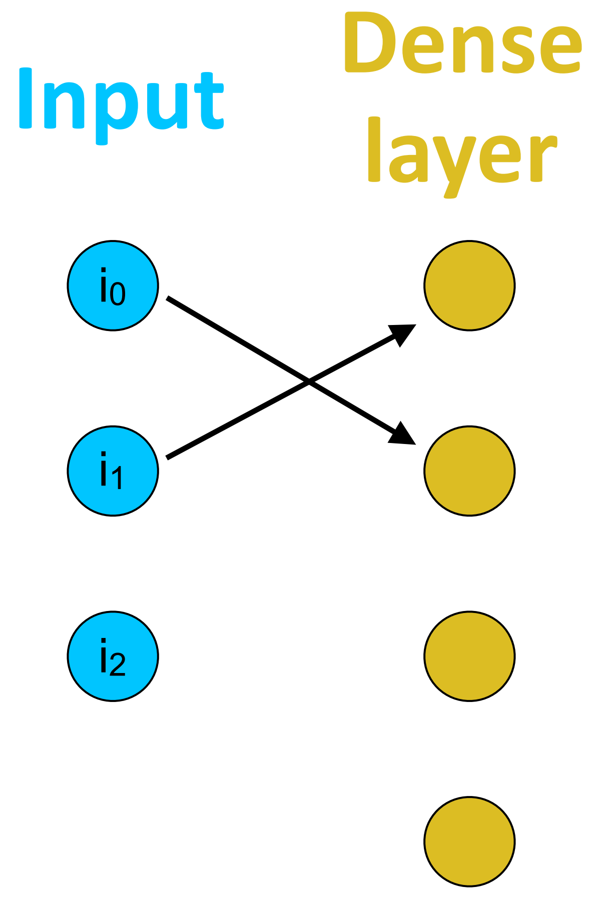




Fully connected layer design

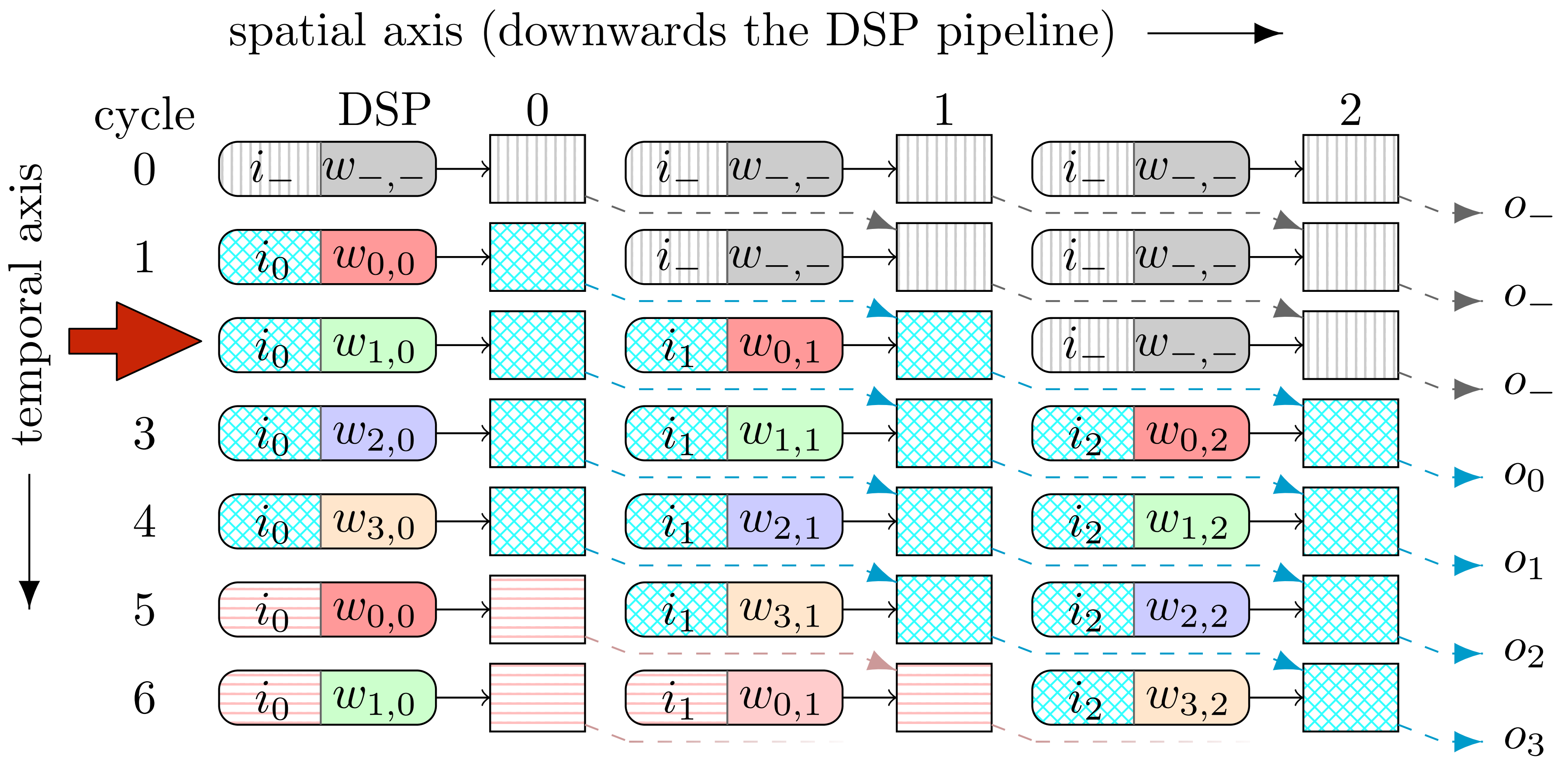
- Exploit: every neuron requires every input
  - Implement neuron processing in **DSP pipelines**
    - Inputs completely reusable
    - Only weight loading / fetching / multiplexing
    - **Simple design with easy parallelisation**

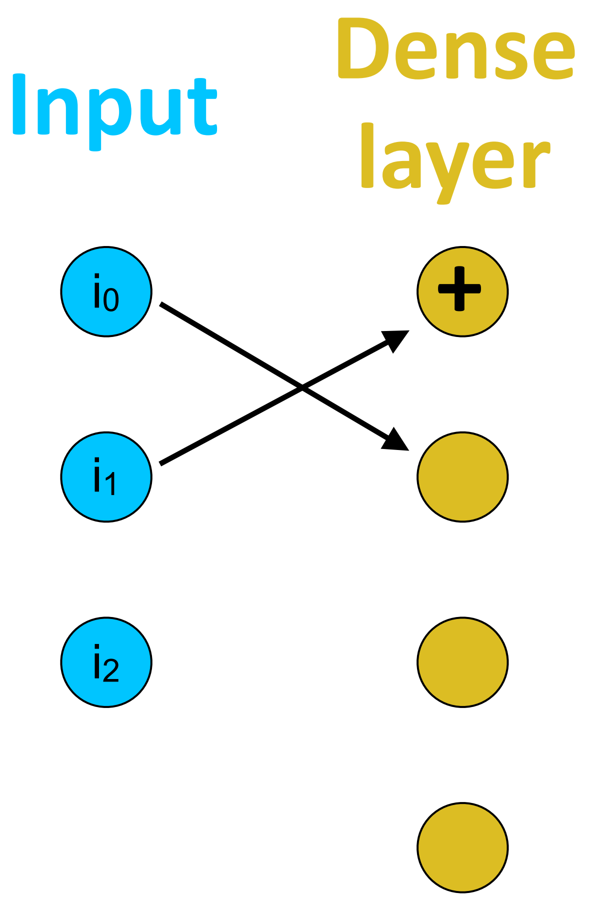




Fully connected layer design

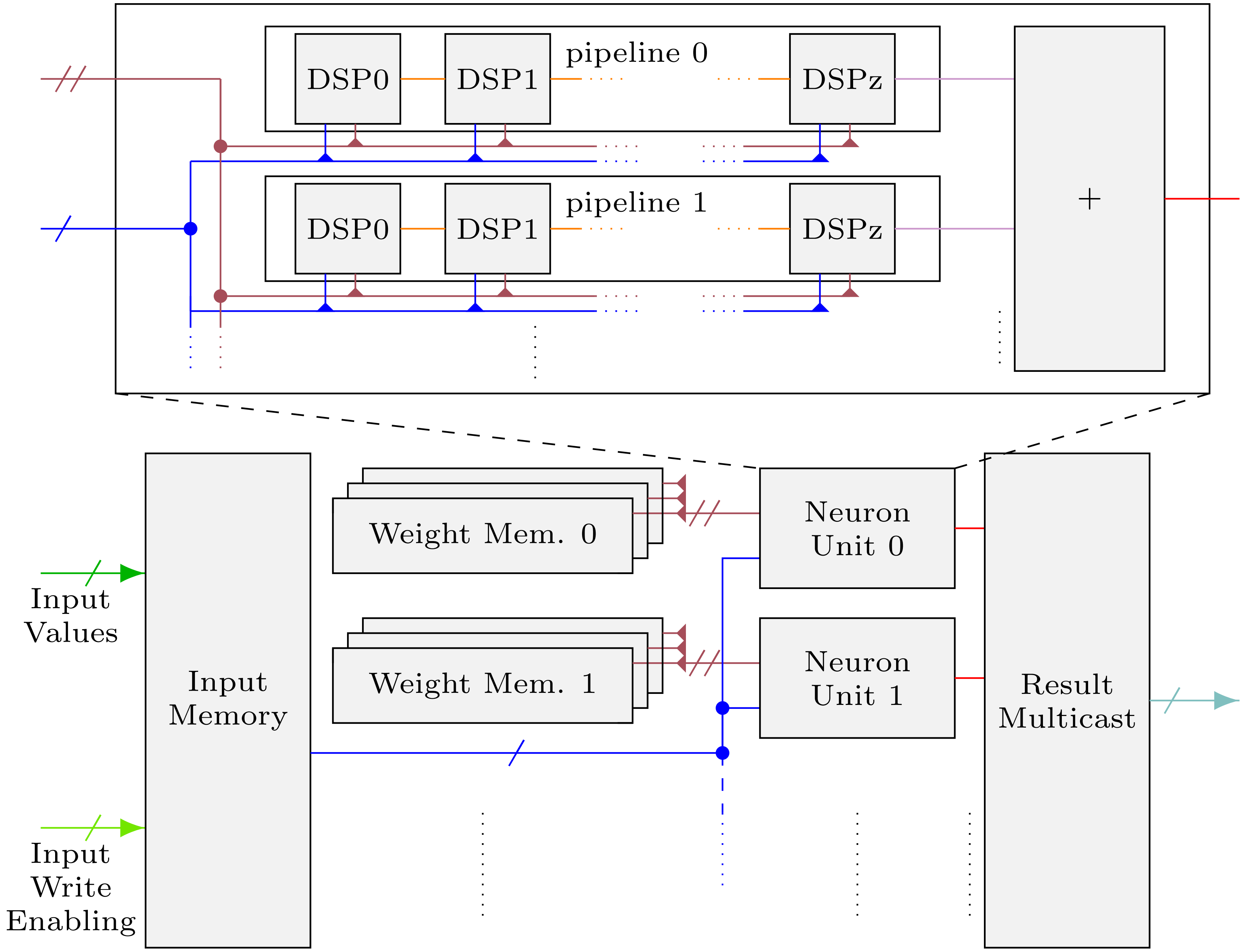
- Exploit: every neuron requires every input
  - Implement neuron processing in **DSP pipelines**
    - Inputs completely reusable
    - Only weight loading / fetching / multiplexing
    - **Simple design with easy parallelisation**





Implementation on the FPGA

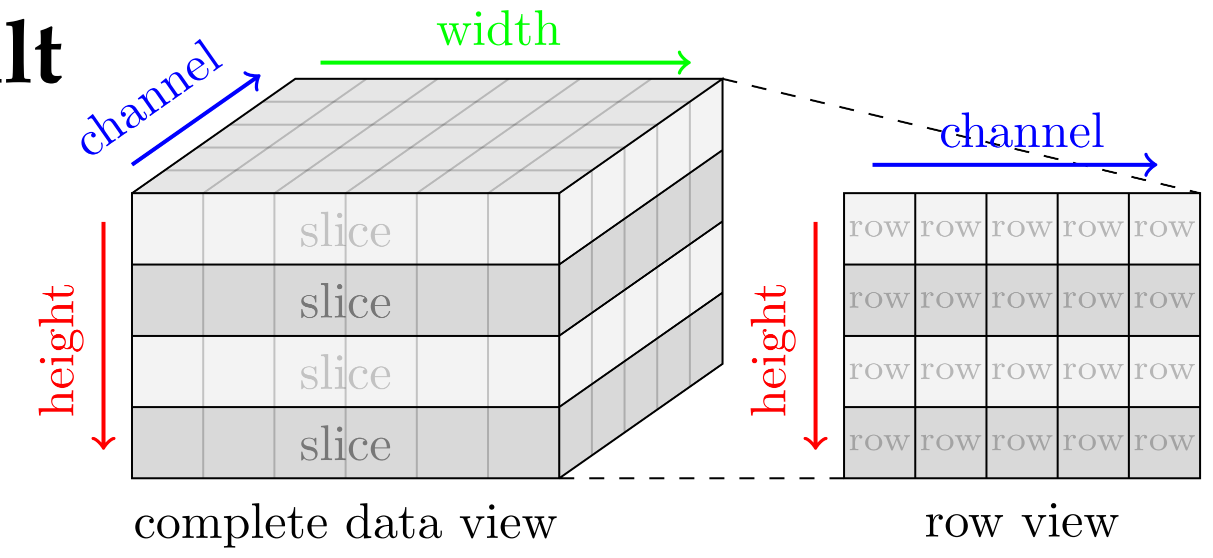
- Use **multiple but shorter pipelines** with additional adder in parallel ("neuron unit") **to reduce latency**

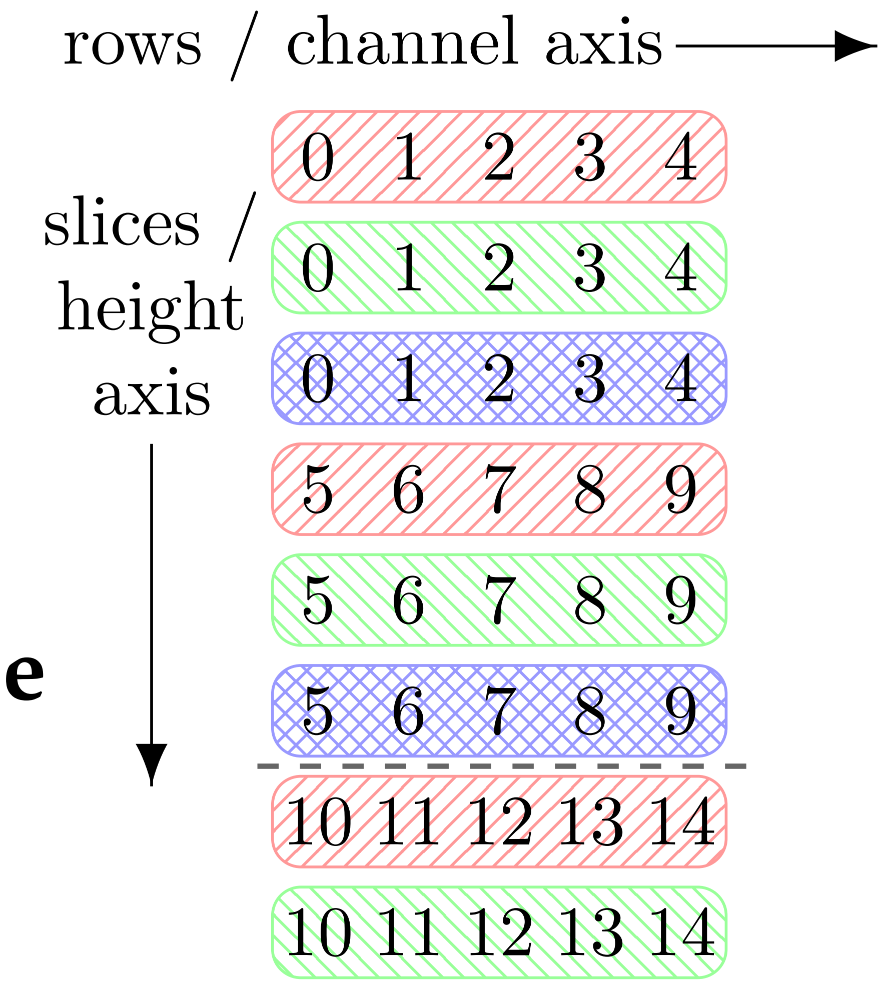


JOHANNES GUTENBERG UNIVERSITÄT MAINZ JG|U

2D Convolution Layer

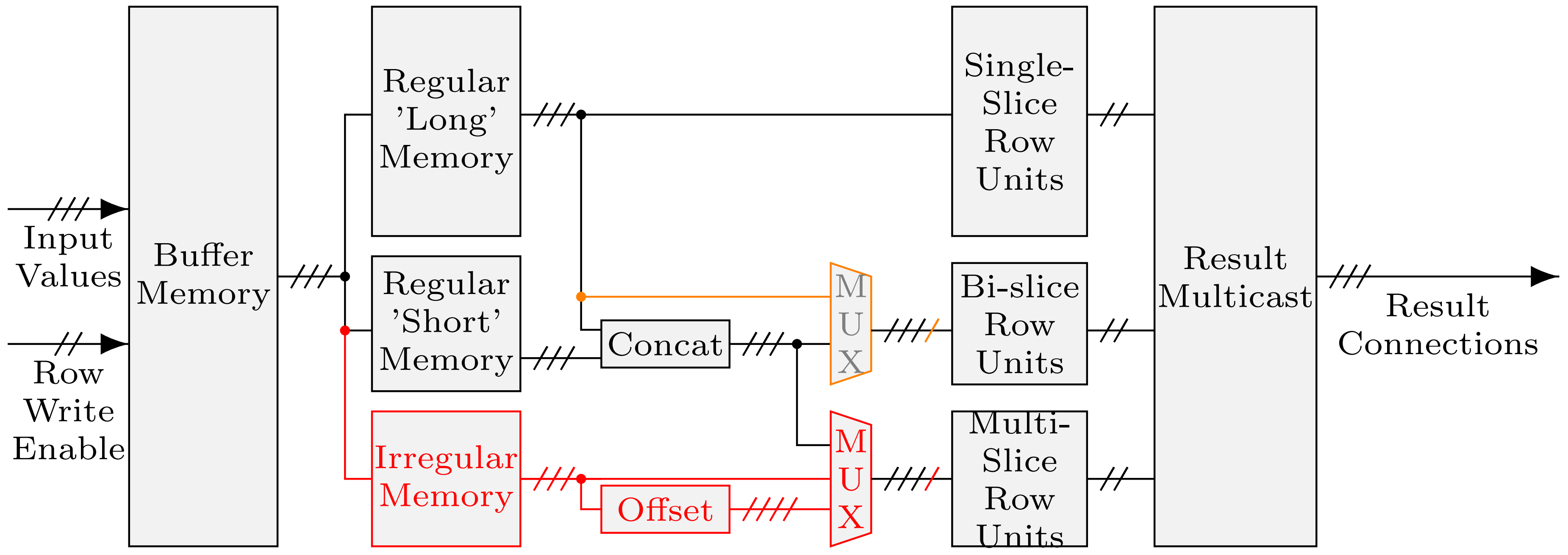
- **2D convolution way more difficult to implement**
  - Naive implementation would need large amount of resources for multiplexing of inputs / weights
- Optimised approach
  - Use “slices” (channel x width) and “rows” (fixed height and channel) as basic quantities
    - **“Row units” yield good compromise of computational efficiency and input/weight reuse**



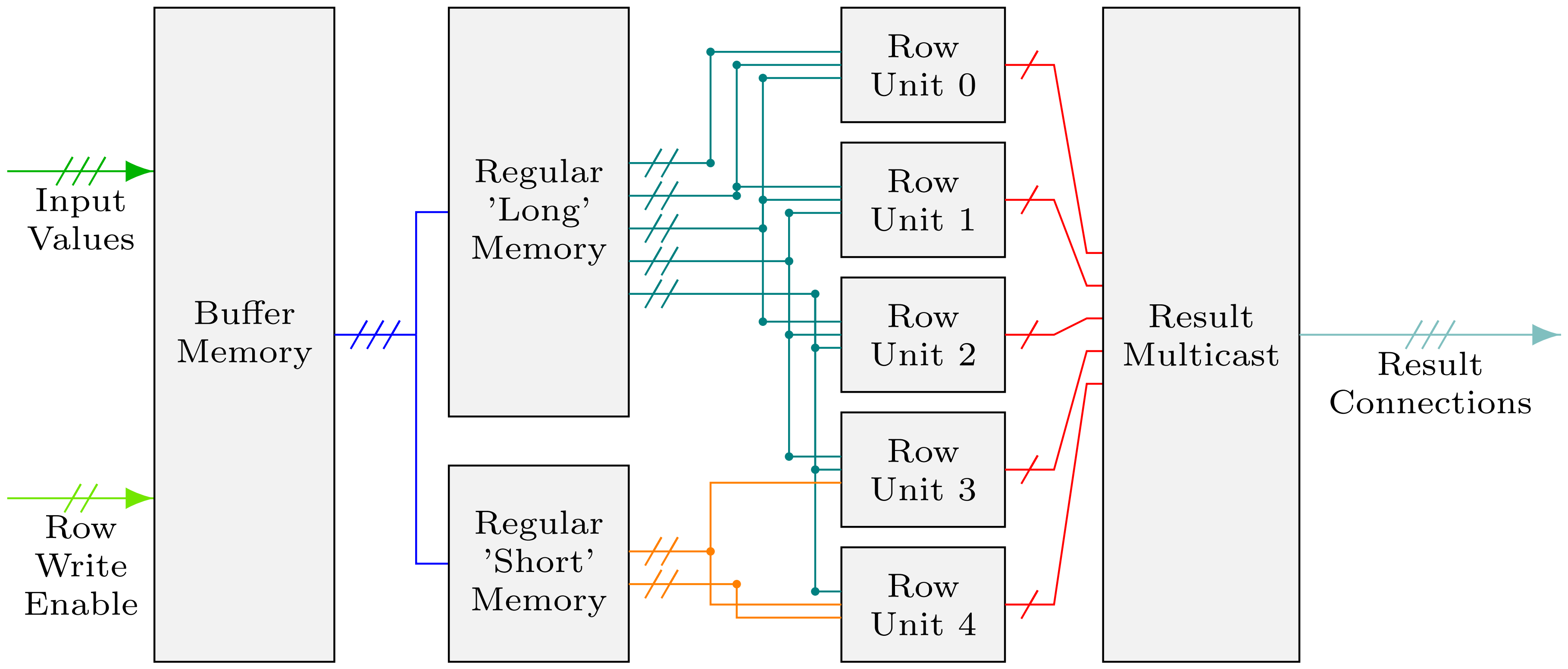


Firmware implementation

All cases:



Regular case:



Implementation results: resource usage

Xilinx US+ XCVU9P-2
(6840 DSPs, 2.4M FF, 1.2M LUT)

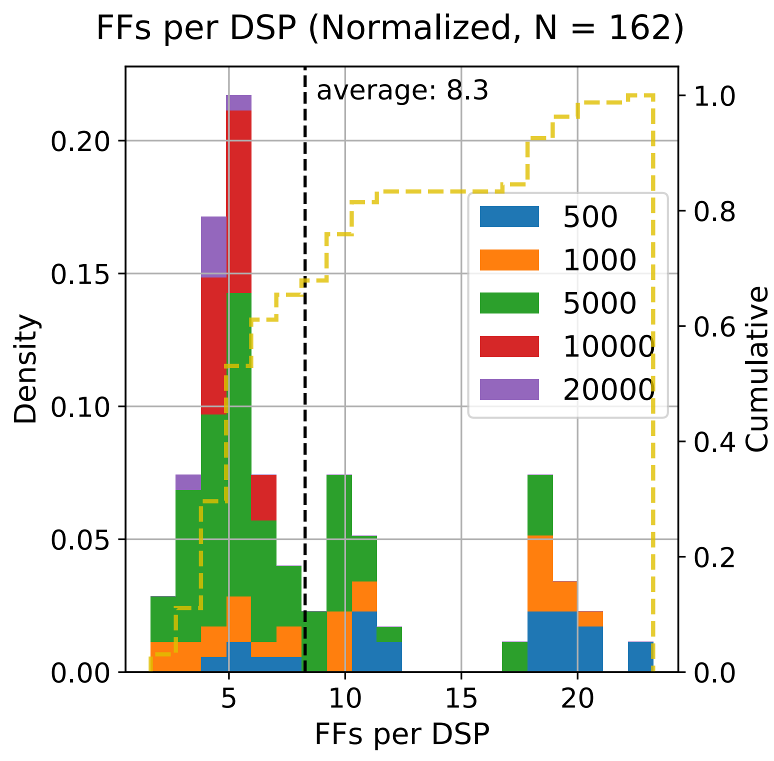
- **Main limitation is number of DSPs**
  - Dense:

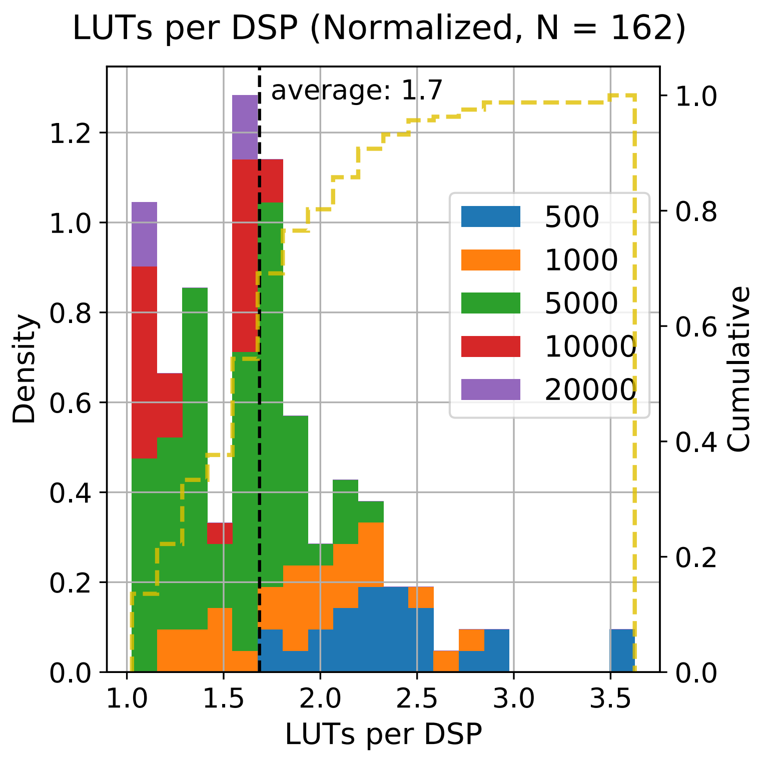
$$N_{\text{DSP}} \approx N_I \cdot N_N \cdot \frac{f_{Data}}{f_{FPGA}}$$

  Conv:

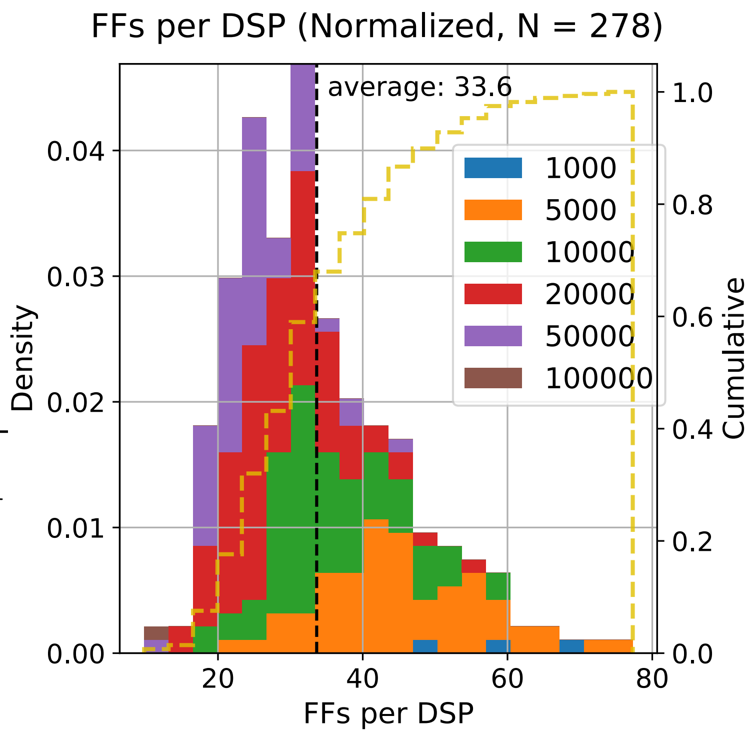
$$N_{\text{DSP}} \approx V_{out} \cdot A_{kernel} \cdot N_{chan,inp} \cdot \frac{f_{Data}}{f_{FPGA}}$$

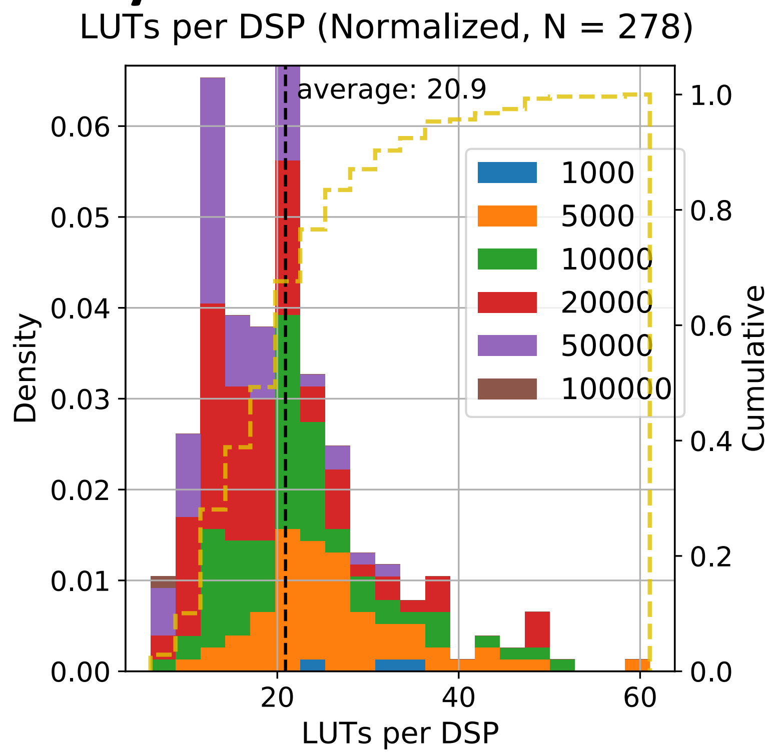
Dense layer





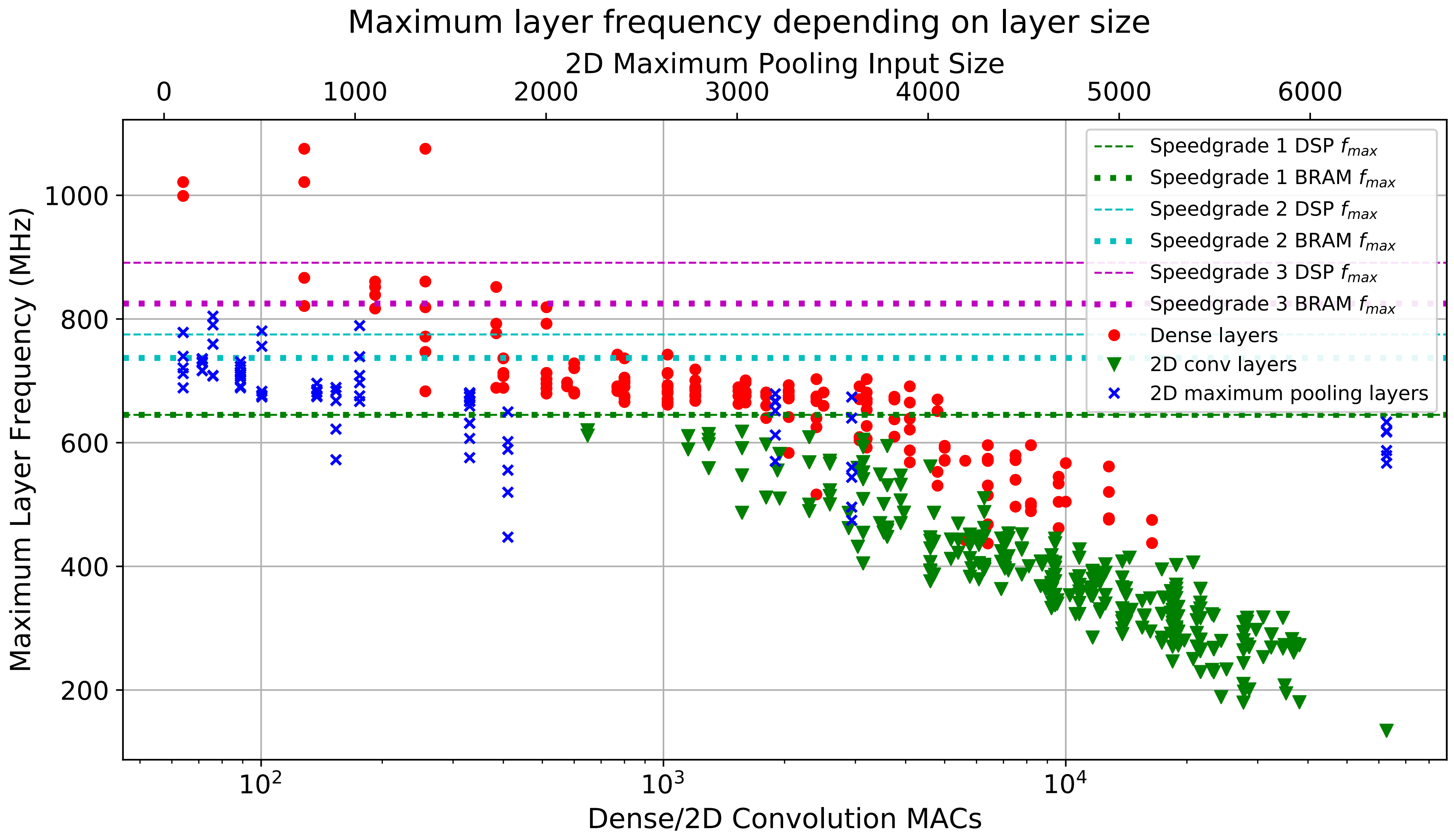
2D conv layer





Implementation results: operating frequency

- Maximum layer frequency depends on resource usage (signal propagation, routing complexity, …)
  - Dense and pooling layers are less complex -> higher frequency
- **Can run at >=400 MHz even for layers with 10k operations**

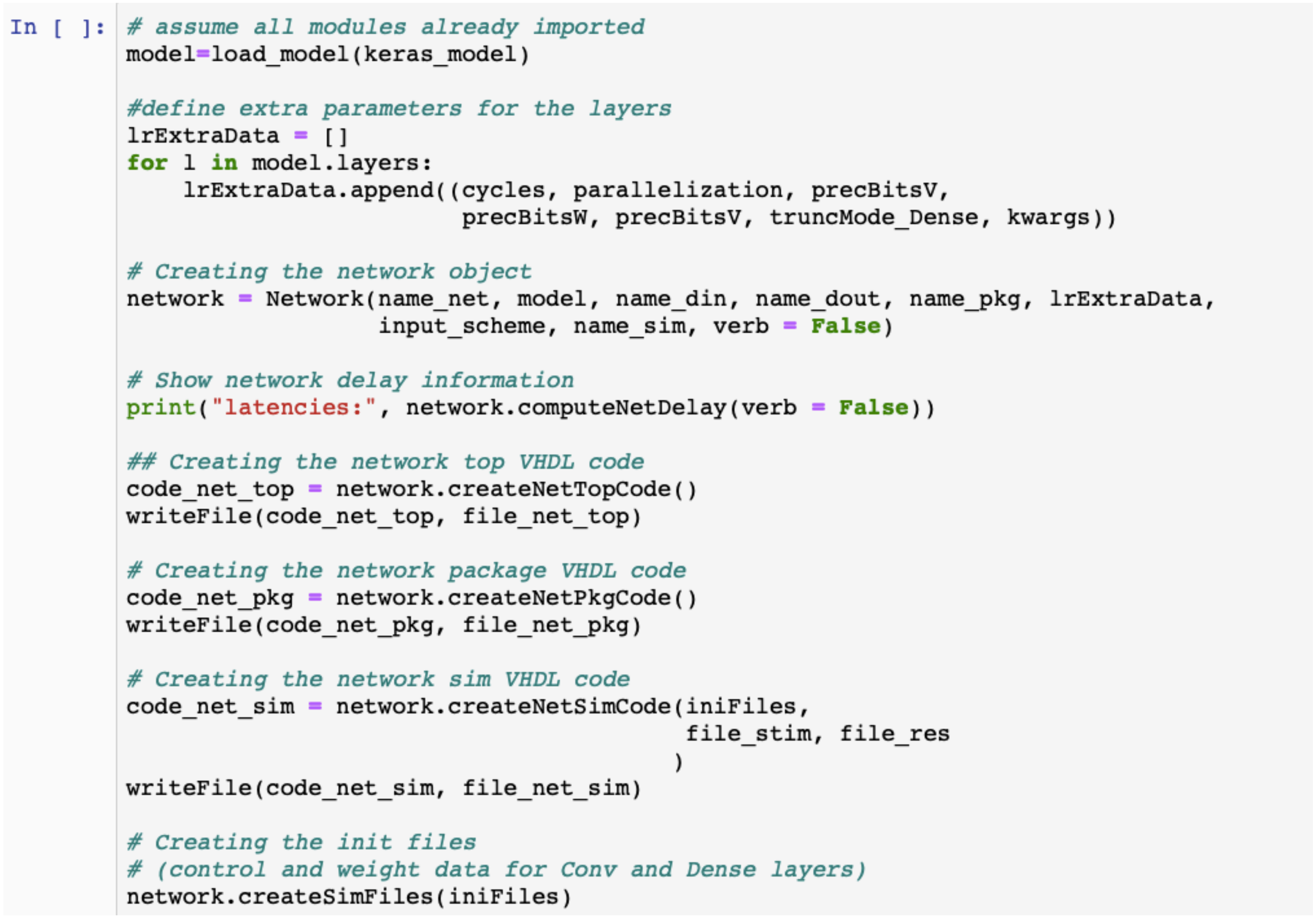


JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Network creation toolkit

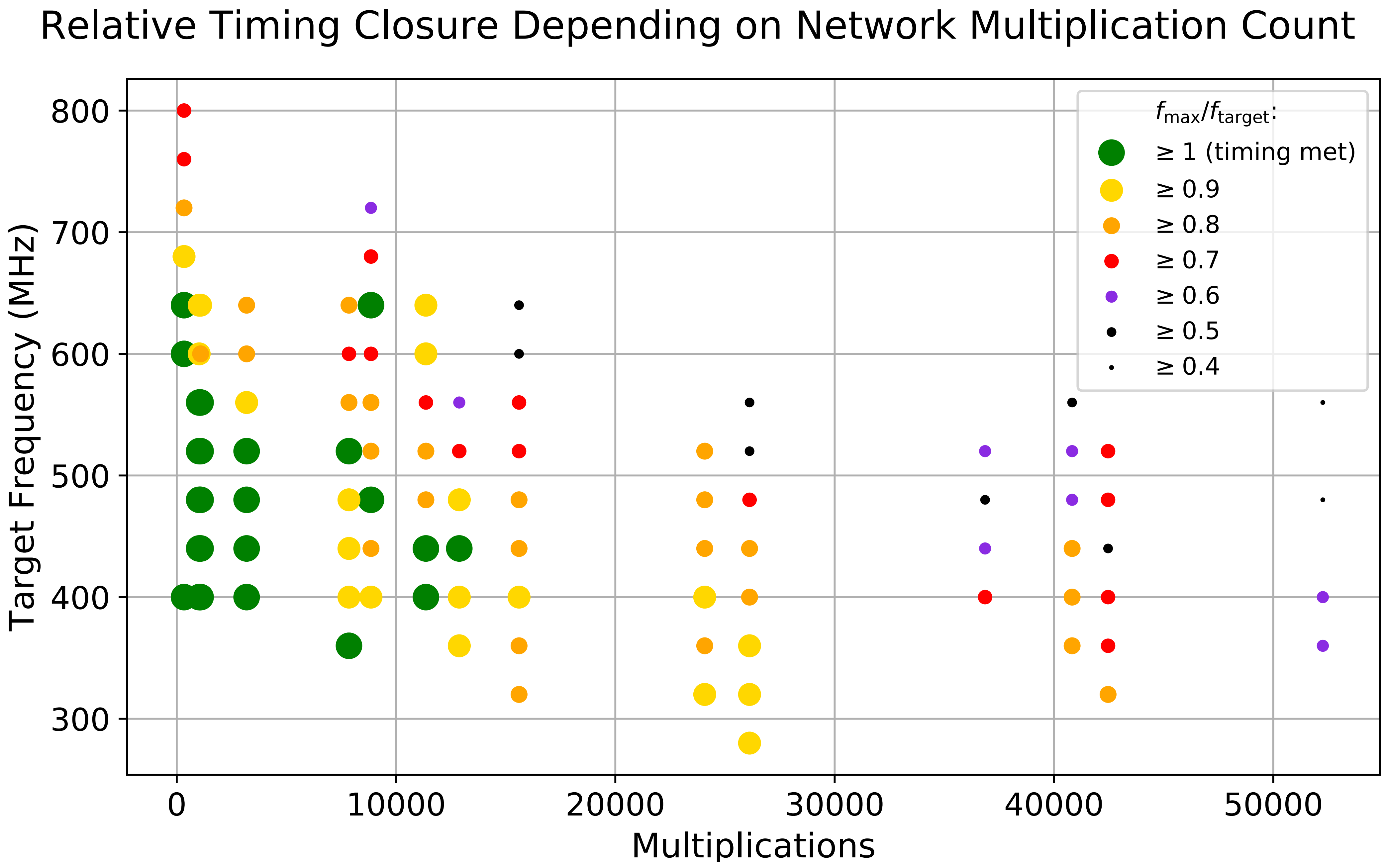
- Python based toolkit for **automated network creation**
- **Starting point: trained Keras network**
  - Supported layers: Dense, 2D-Conv, Maxpool
  - Activation: relu (best for FPGA)
- Additional design parameters can be specified:
  - Precision (integer and fractional bits)
  - Pipelining and routing behaviour
- **Output:**
  - **VHDL code of the corresponding network**

Network creation toolkit: example usage



```
In [ ]: # assume all modules already imported
        model=load_model(keras_model)

        #define extra parameters for the layers
        lrExtraData = []
        for l in model.layers:
            lrExtraData.append((cycles, parallelization, precBitsV,
                                precBitsW, precBitsV, truncMode_Dense, kwargs))

        # Creating the network object
        network = Network(name_net, model, name_din, name_dout, name_pkg, lrExtraData,
                          input_scheme, name_sim, verb = False)

        # Show network delay information
        print("latencies:", network.computeNetDelay(verb = False))

        ## Creating the network top VHDL code
        code_net_top = network.createNetTopCode()
        writeFile(code_net_top, file_net_top)

        # Creating the network package VHDL code
        code_net_pkg = network.createNetPkgCode()
        writeFile(code_net_pkg, file_net_pkg)

        # Creating the network sim VHDL code
        code_net_sim = network.createNetSimCode(iniFiles,
                                                file_stim, file_res
                                               )
        writeFile(code_net_sim, file_net_sim)

        # Creating the init files
        # (control and weight data for Conv and Dense layers)
        network.createSimFiles(iniFiles)
```

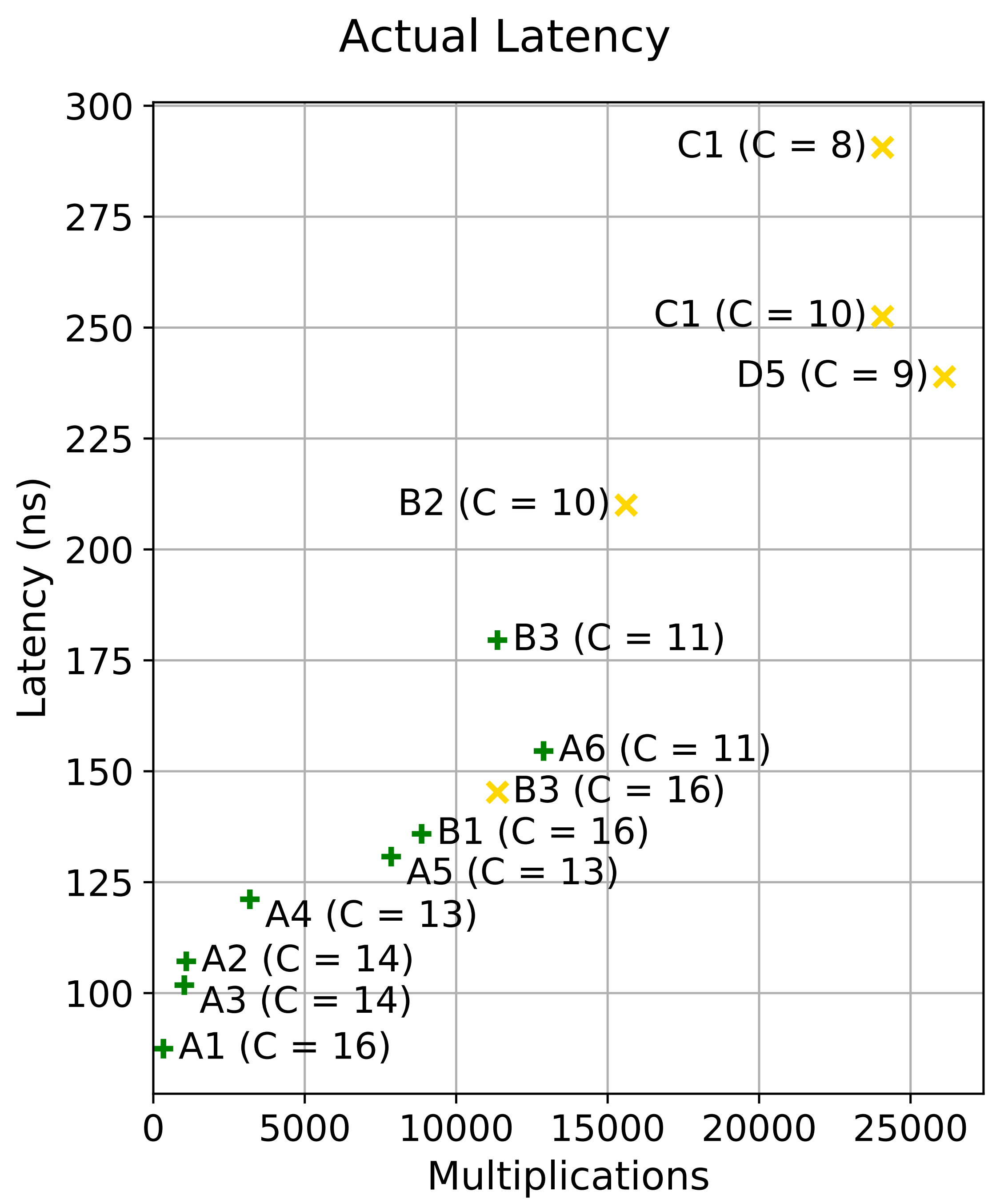
Results: timing closure

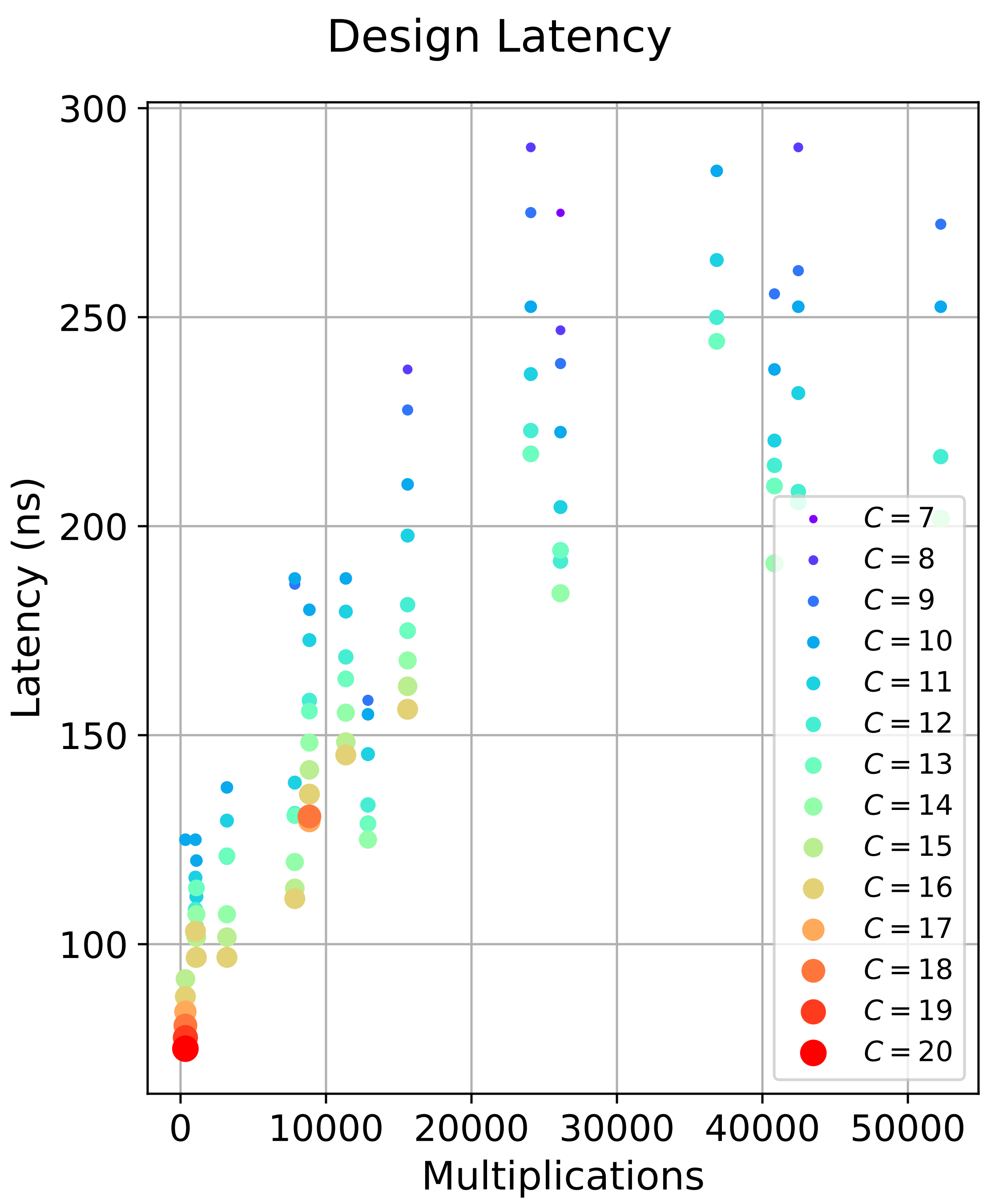


- **Successful network implementations up to 15k multiplications for a data frequency of 40 MHz (e.g. LHC)**

JOHANNES GUTENBERG UNIVERSITÄT MAINZ JG|U

Results: overall latency





- Latency depends on achievable frequency
- **Full network output can be available in ~100ns**

$$C = \frac{f_{FPGA}}{f_{Data}}$$

Summary & Outlook

- **Full networks consisting of 2D-Conv, Maxpooling and Dense layers implemented on FPGAs**
  - Can cope with LHC data frequencies of 40 MHz, full network latencies of O(100ns)
  - **Easy to use** python based **toolkit** for automatic creation of VHDL code from trained Keras model
  - **Publication: 2019 JINST14 P09014**
- Next steps:
  - Implement first physics example network using this toolkit that would fit within ATLAS Run-3 hardware
  - Extend toolkit to support more layer types and further optimisations on layer implementations

Example network architectures

Architecture (see text) (layer information)	MACs (DSP eff.)	T_P (ns)	WNS (ns)	latency (cycles)	N_{LUT} N_{DSP}	N_{FF} N_{BRAM}
Arc$_{A1}$ (C = 16) (input (7×7))	334	1.562	-	56	1793	3571
$(2\times2\times1)$-(2×2)-10	(0.485)				43	10.5
Arc$_{A2}$ (C = 14)	1089	1.786	-	60	5060	9706
$(2\times2\times1)$-(2×2)-7	(0.630)				108	17
Arc$_{A3}$ (C = 14) (input (7×7))	1024	1.786	-	57	3051	5654
$(2\times2\times3)$-(2×2)-16)	(0.620)				118	19
Arc$_{A4}$ (C = 13)	3188	1.923	-	63	8689	16219
$(2\times2\times2)$-(2×2)-17)	(0.774)				317	54.5
Arc$_{A5}$ (C = 13)	7854	1.923	-	68	15567	28450
$(2\times2\times4)$-(2×2)-25	(0.967)				625	93.5
Arc$_{A6}$ (C = 11)	12884	2.273	-	68	20962	34711
$(3\times3\times4)$-(2×2)-50	(0.894)				1310	166
Arc$_{B1}$ (C = 12)	8858	2.083	-	76	18587	32886
$(2\times2\times4)$-(2×2)-$(2\times2\times4)$-25	(0.812)				909	99.5
Arc$_{B1}$ (C = 16)	8858	2.083	-	87	17205	32760
$(2\times2\times4)$-(2×2)-$(2\times2\times4)$-25	(0.812)				713	71.5
Arc$_{B3}$ (C = 11)	11362	2.273	-	79	28383	47140
$(2\times2\times6)$-(2×2)-$(2\times2\times4)$-25	(0.792)				1305	102.5
Arc$_{B2}$ (C = 10)	15610	2.500	-0.134	84	40998	69333
$(3\times3\times6)$-(2×2)-$(3\times3\times6)$-25	(0.855)				1825	68
Arc$_{B3}$ (C = 16)	11362	1.562	-0.014	93	26006	45065
$(2\times2\times6)$-(2×2)-$(2\times2\times4)$-25	(0.825)				861	71.5