DeepLearning on FPGAs

Introduction to FPGAs

Sebastian Buschjäger

Technische Universität Dortmund - Fakultät Informatik - Lehrstuhl 8

October 24, 2017

Recap: Convolution

Observation 1 Even smaller images need a lot of neurons

Our approach Discrete convolution

$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$

170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

image

$*$

1	−0.5
−0.5	1

kernel / weights / filter

$=$

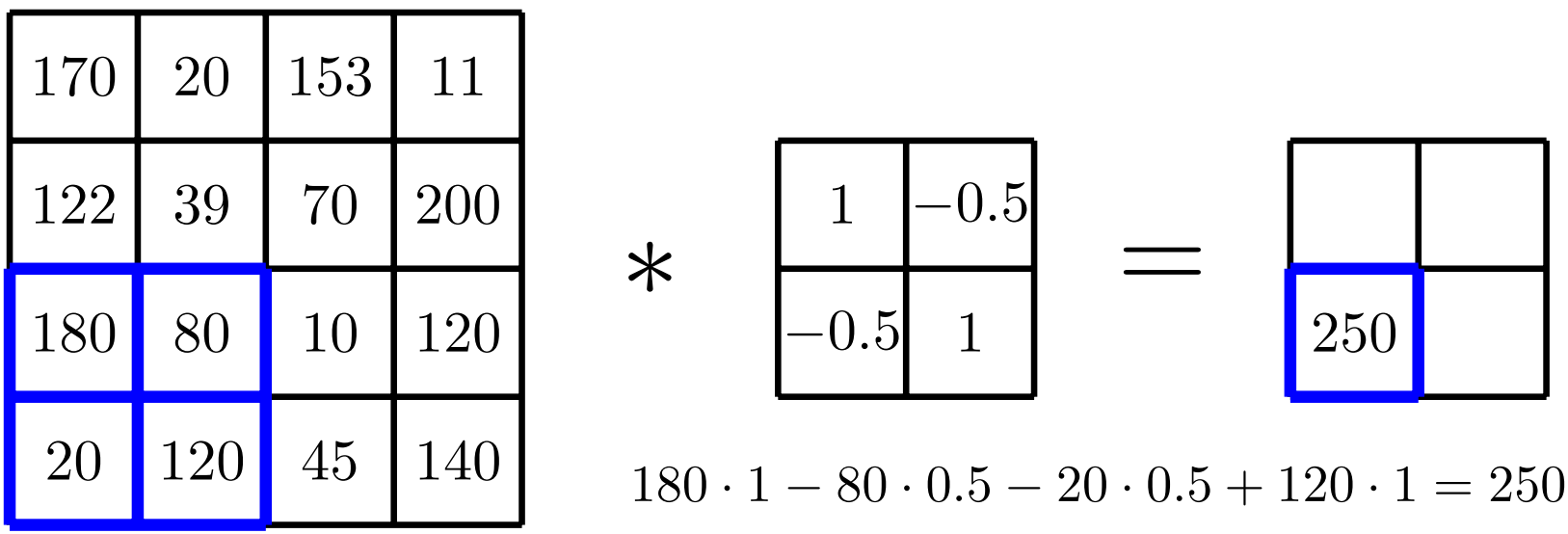
result

Recap: Convolution

Observation 1 Even smaller images need a lot of neurons

Our approach Discrete convolution

$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$



Recap: Convolution

Observation 1 Even smaller images need a lot of neurons

Our approach Discrete convolution

$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$

170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

image

$*$

1	−0.5
−0.5	1

kernel / weights / filter

$=$

250	67

result

$$10 \cdot 1 - 120 \cdot 0.5 - 45 \cdot 0.5 + 140 \cdot 1 = 67$$

Recap: Convolution

Observation 1 Even smaller images need a lot of neurons

Our approach Discrete convolution

$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$

170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

image

$*$

1	−0.5
−0.5	1

kernel / weights / filter

$=$

138	
250	67

result

$$170 \cdot 1 - 20 \cdot 0.5 - 122 \cdot 0.5 + 39 \cdot 1 = 138$$

Recap: Convolution

Observation 1 Even smaller images need a lot of neurons

Our approach Discrete convolution

$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$

170	20	153	11
122	39	70	200
180	80	10	120
20	120	45	140

image

$*$

1	−0.5
−0.5	1

kernel / weights / filter

$=$

138	255
250	67

result

$$153 \cdot 1 - 11 \cdot 0.5 - 70 \cdot 0.5 + 200 \cdot 1 = 255$$

Recap: CNNs and weight sharing

f_{00}	f_{01}	f_{02}
f_{10}	f_{11}	f_{12}
f_{20}	f_{21}	f_{22}

input $\vec{f}$

$*$

w_{00}	w_{01}
w_{10}	w_{11}

weights $\vec{w}$

$=$

$w_{00}f_{00} + w_{01}f_{01} + w_{10}f_{10} + w_{11}f_{11}$	$w_{00}f_{01} + w_{01}f_{02} + w_{10}f_{11} + w_{11}f_{12}$
$w_{00}f_{10} + w_{01}f_{11} + w_{10}f_{20} + w_{11}f_{21}$	$w_{00}f_{11} + w_{01}f_{12} + w_{10}f_{21} + w_{11}f_{22}$

output $\vec{y}$

Mathematically:

$$y_{i,j}^{(l)} = \sum_{i'=0}^{M^{(l)}} \sum_{j'=0}^{M^{(l)}} w_{i,j}^{(l)} \cdot f_{i+i',j+j'}^{(l-1)} + b_{i,j}^{(l)} = w^{(l)} * f^{(l-1)} + b^{(l)}$$

$$f_{i,j}^{(l)} = \sigma(y_{i,j}^{(l)})$$

$M^{(l)} \times M^{(l)}$ bias **matrix**!

Recap: Backpropagation for CNNs

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} * rot180(f)^{(l-1)} f_{i,j}^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

$$\delta^{(l+1)} = \delta^{(l)} * rot180(w^{(l+1)}) \cdot \frac{\partial h(y_{i,j}^l)}{\partial h(y_{i,j}^l)}$$

Recap: Backpropagation for CNNs

Gradient step:

$$\begin{aligned} w_{i,j}^{(l)} &= w_{i,j}^{(l)} - \alpha \cdot \delta^{(l)} * rot180(f)^{(l-1)} f_{i,j}^{(l-1)} \\ b_j^{(l)} &= b_j^{(l)} - \alpha \cdot \delta_j^{(l)} \end{aligned}$$

Recursion:

$$\delta^{(l+1)} = \delta^{(l)} * rot180(w^{(l+1)}) \cdot \frac{\partial h(y_{i,j}^l)}{\partial h(y_{i,j}^l)}$$

$$rot180 \begin{array}{|c|c|} \hline w_{10} & w_{11} \\ \hline w_{00} & w_{01} \\ \hline \end{array} = \begin{array}{|c|c|} \hline w_{01} & w_{00} \\ \hline w_{11} & w_{10} \\ \hline \end{array}$$

Hardware: Current trends

Moore's law: The number of transistors on a chip doubles every $12 - 24$ month $\Rightarrow$ We can double the speed roughly every 2 years

[1] Intel predicts $5nm$ transistors to be available around 2020.

Hardware: Current trends

Moore's law: The number of transistors on a chip doubles every $12 - 24$ month $\Rightarrow$ We can double the speed roughly every 2 years

Fact 1: Engineering is currently producing $11 - 16nm$ transistors[1]
Side-Note: A $4nm$ transistor can be built from only 7 atoms!
Fact 2: The smaller transistors get, the more quantum effects are happening. Moore's law is predicted to expire with $5nm$ transistors

[1] Intel predicts $5nm$ transistors to be available around 2020.

Hardware: Current trends

Moore's law: The number of transistors on a chip doubles every $12 - 24$ month $\Rightarrow$ We can double the speed roughly every 2 years

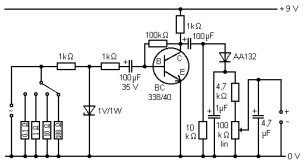
Fact 1: Engineering is currently producing $11 - 16nm$ transistors[1]
Side-Note: A $4nm$ transistor can be built from only 7 atoms!
Fact 2: The smaller transistors get, the more quantum effects are happening. Moore's law is predicted to expire with $5nm$ transistors

How to deal with this problem

- Multi/Many core systems
- Add specialized components in CPU
- Use dedicated hardware for specific tasks

[1] Intel predicts $5nm$ transistors to be available around 2020.

Hardware Overview



ASIC

GPGPU / CPU

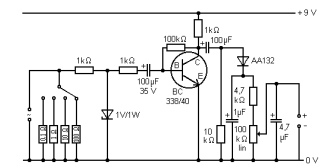
Fact:

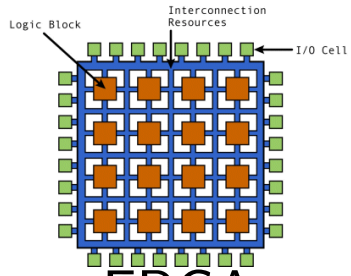
- speed: fastest
- energy: $\sim \mu$W
- application specific
- costs: expensive

Fact:

- speed: fast
- energy: $\sim$ W
- general purpose
- costs: cheap

Hardware Overview





ASIC — FPGA — GPGPU / CPU

Fact:

- speed: fastest
- energy: $\sim \mu$W
- application specific
- costs: expensive

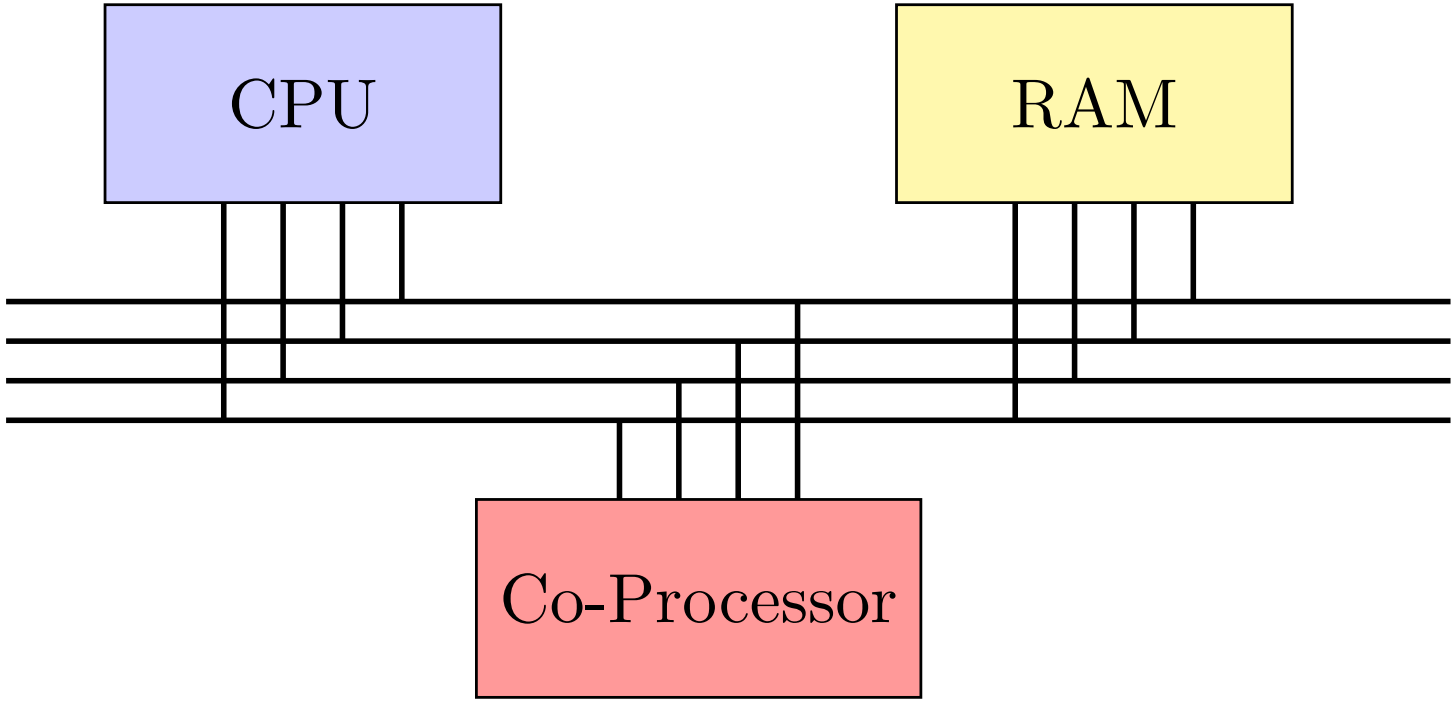
Hope:

- speed: faster
- energy: $\sim$ mW
- general + specific
- costs: cheap

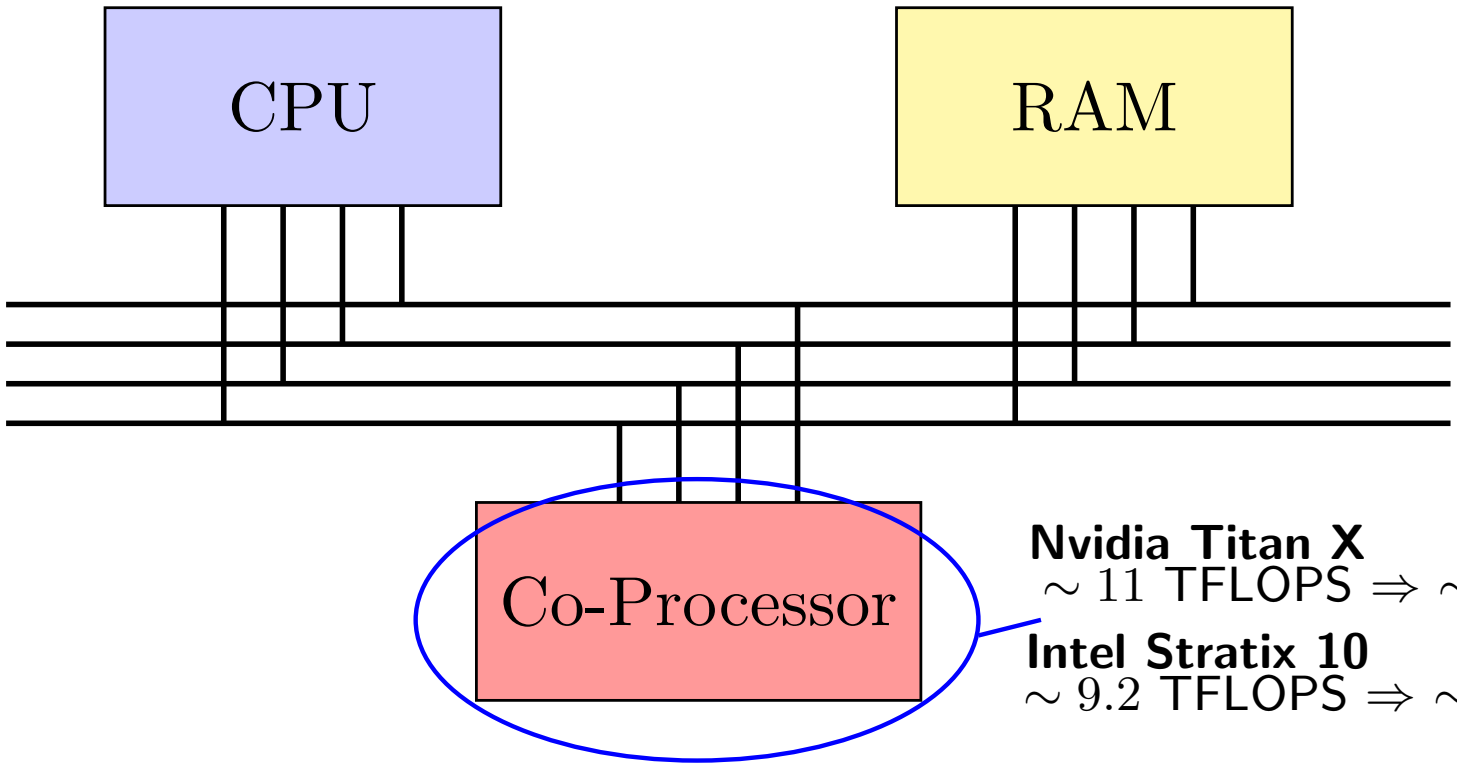
Fact:

- speed: fast
- energy: $\sim$ W
- general purpose
- costs: cheap

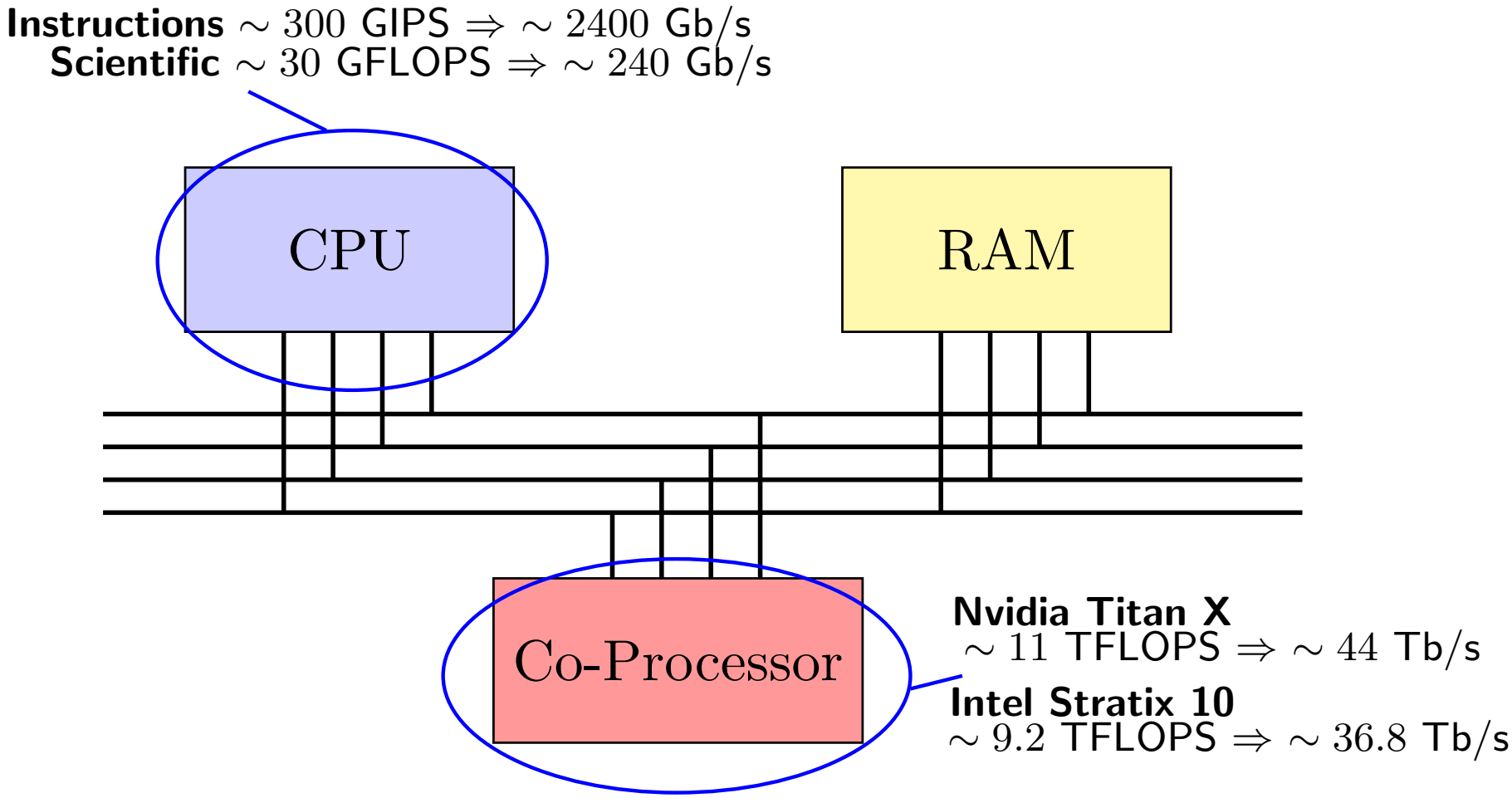
Modular system with common bus



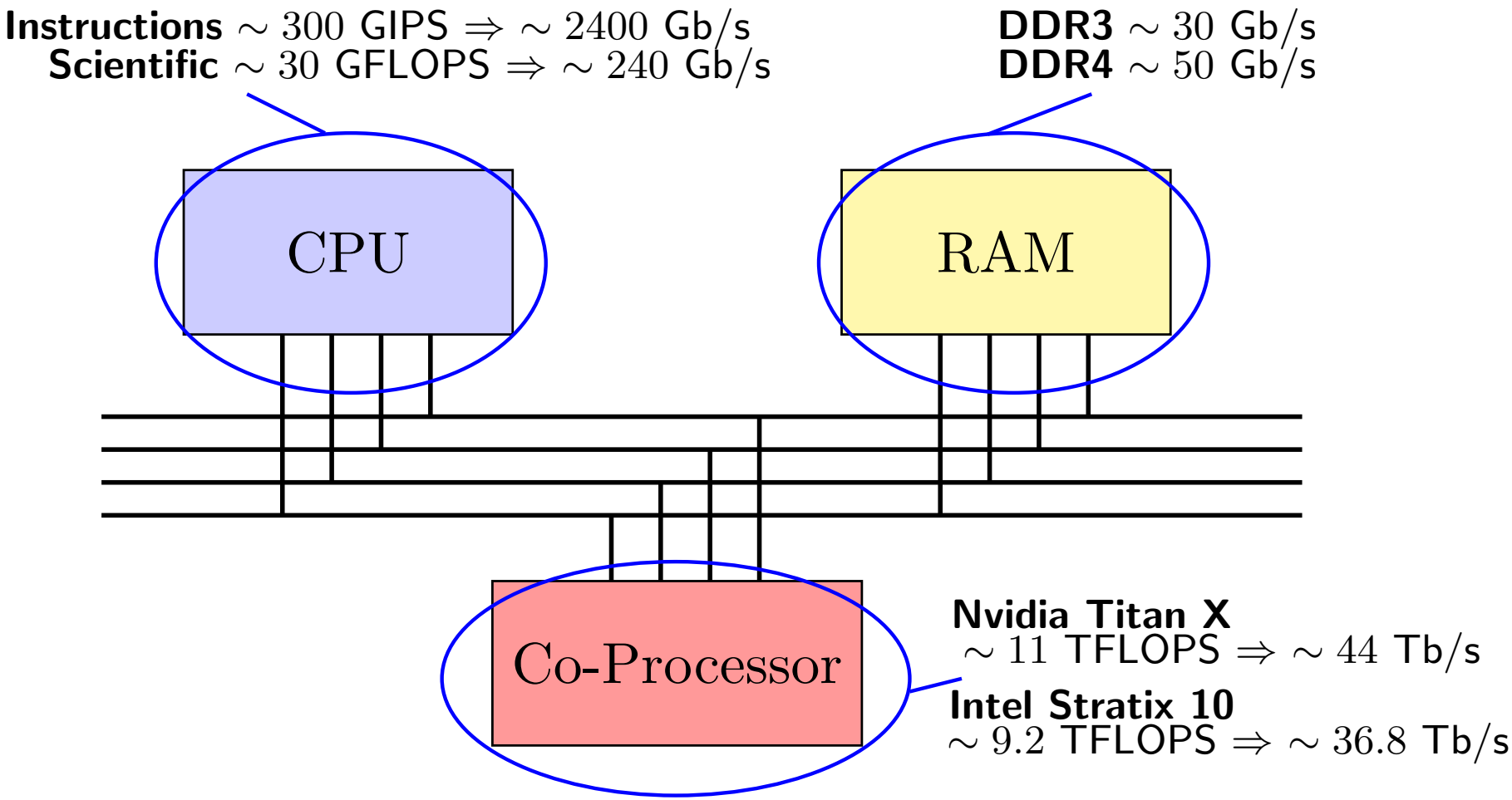
Modular system with common bus



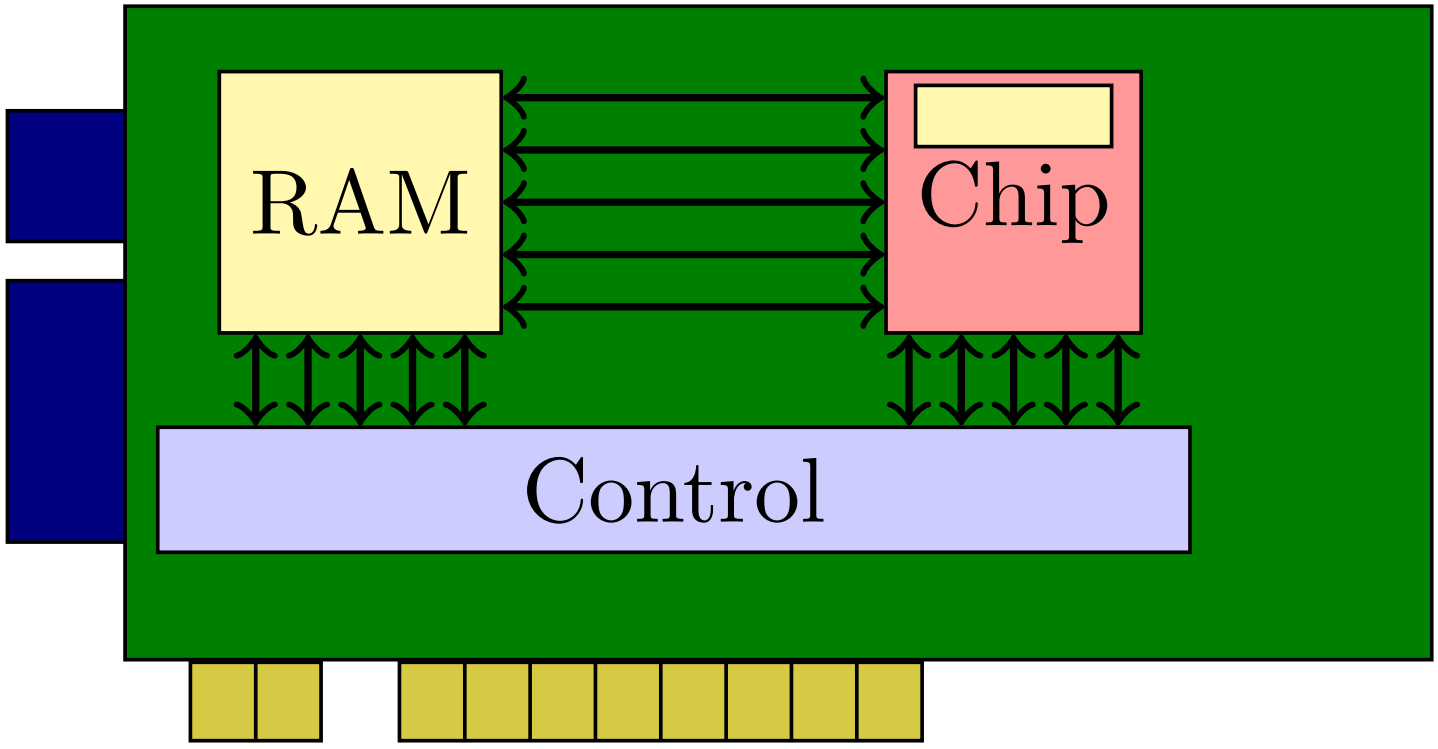
Modular system with common bus



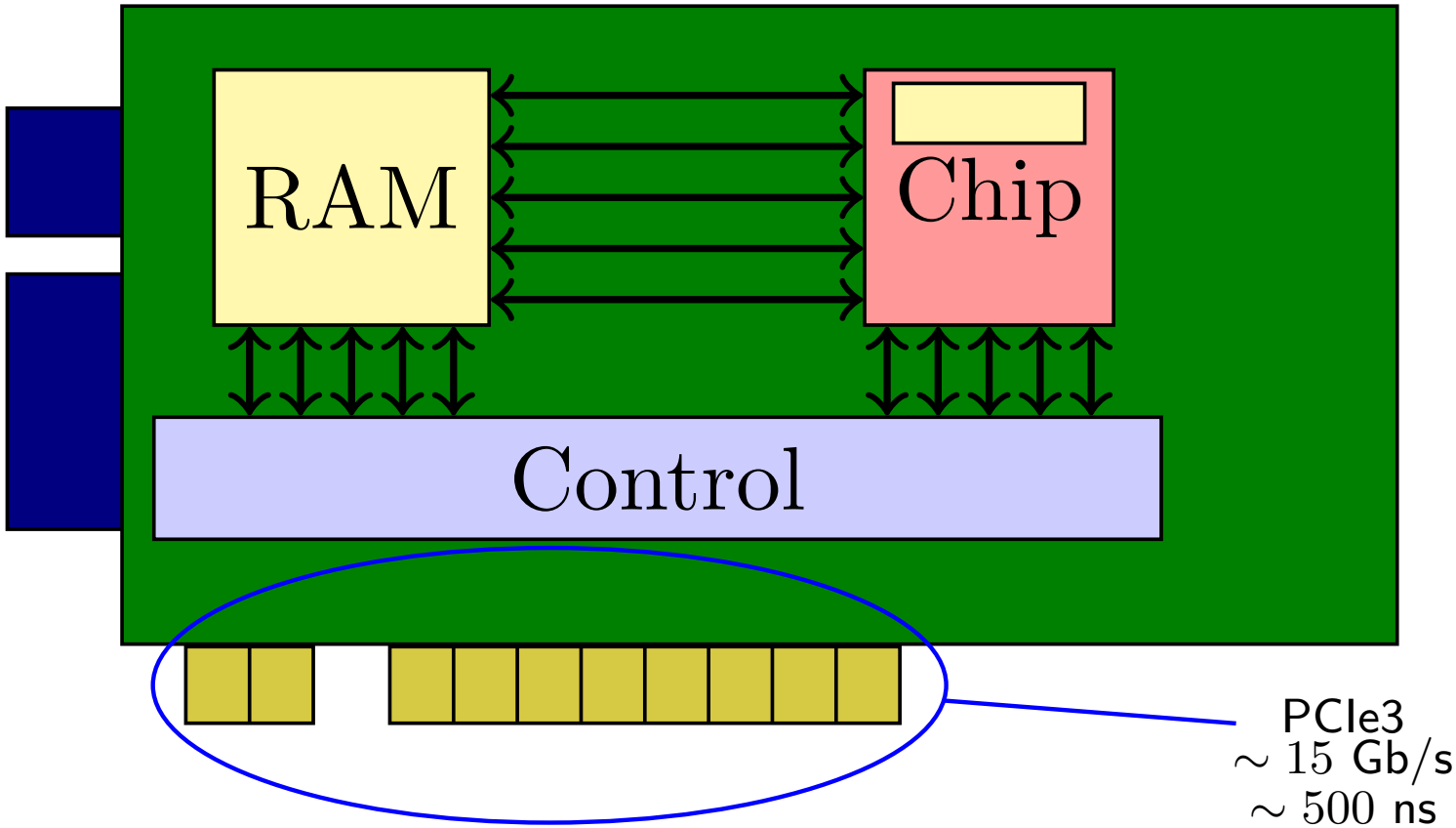
Modular system with common bus



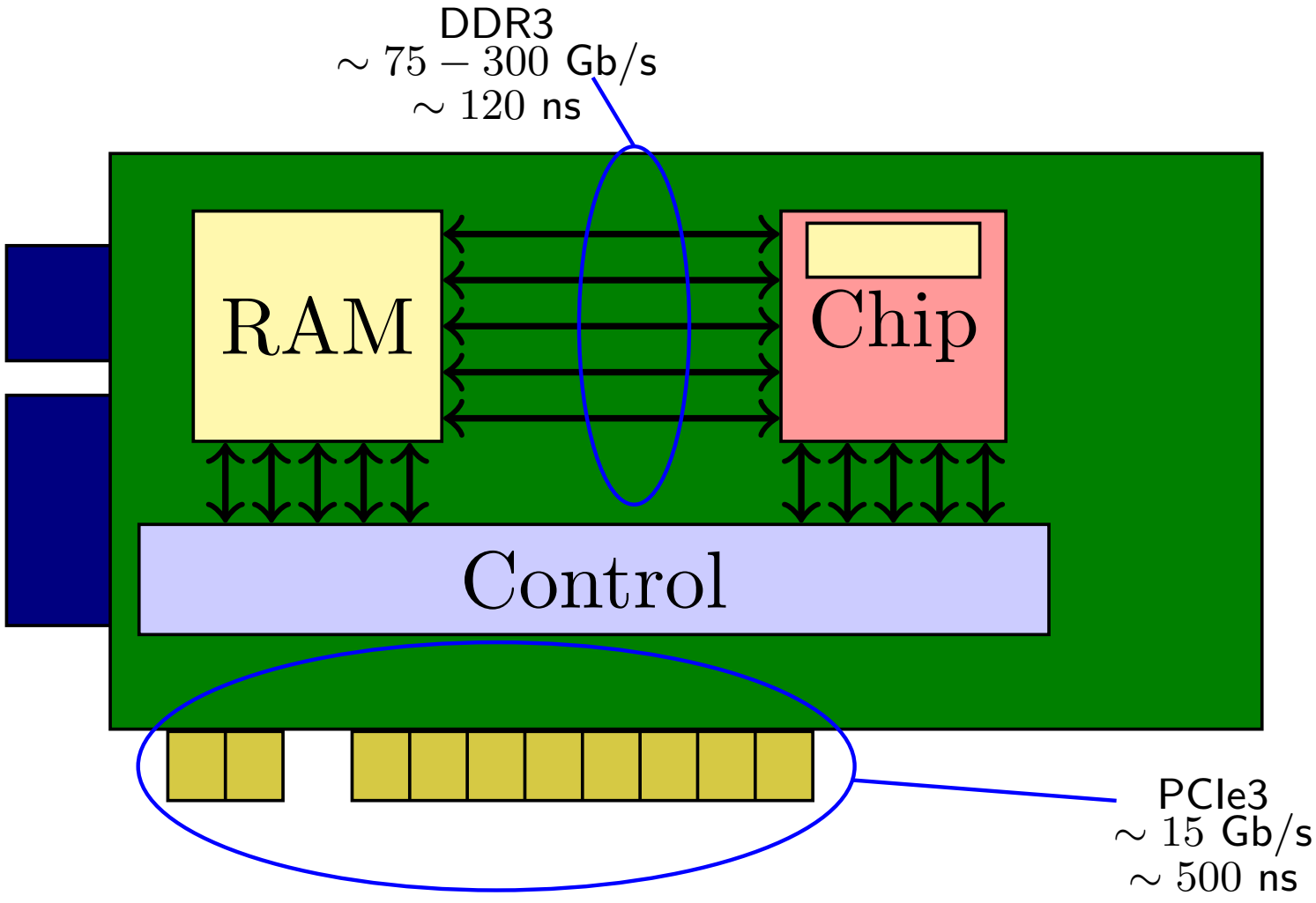
Co-Processor: Actual chip with memory interface



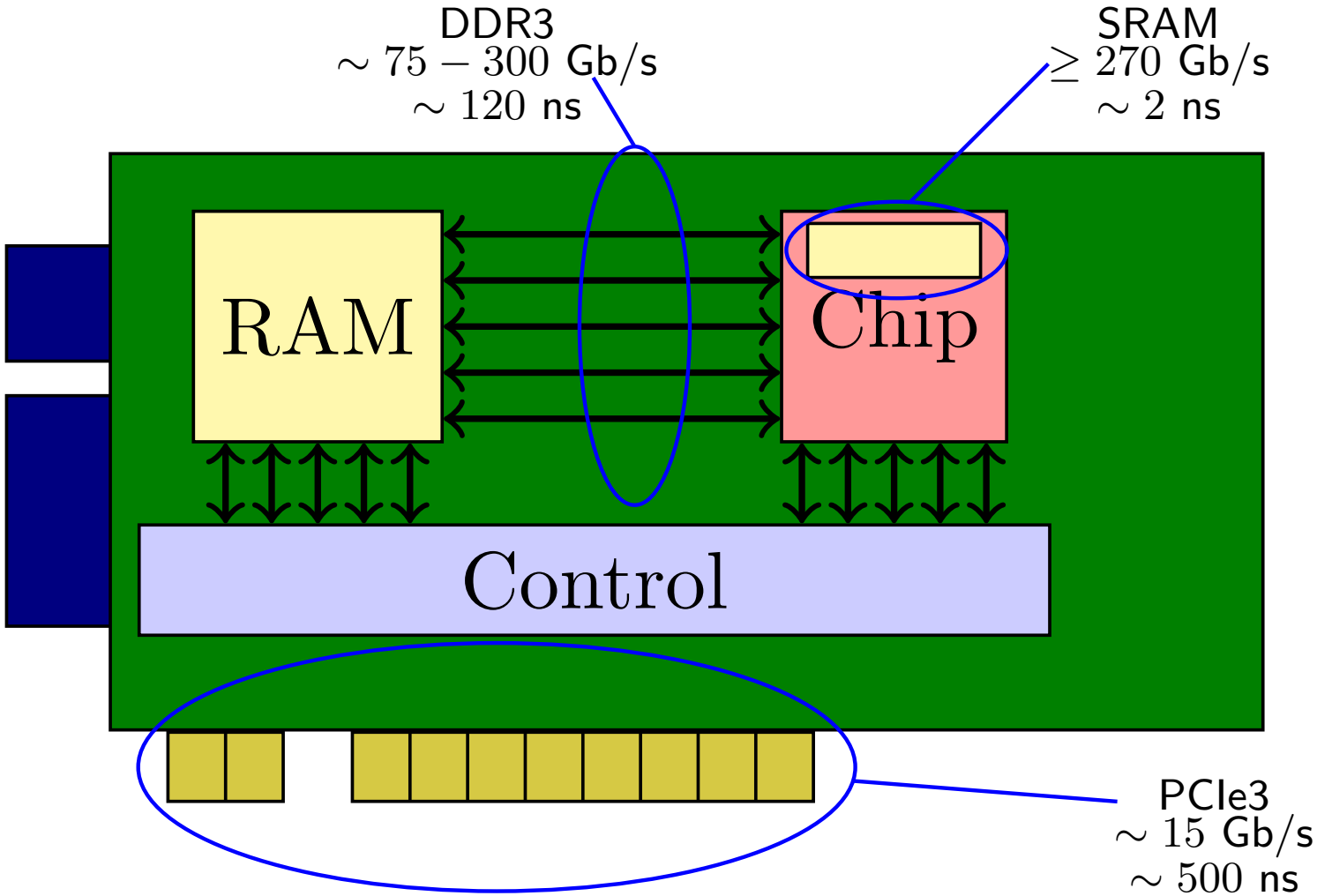
Co-Processor: Actual chip with memory interface



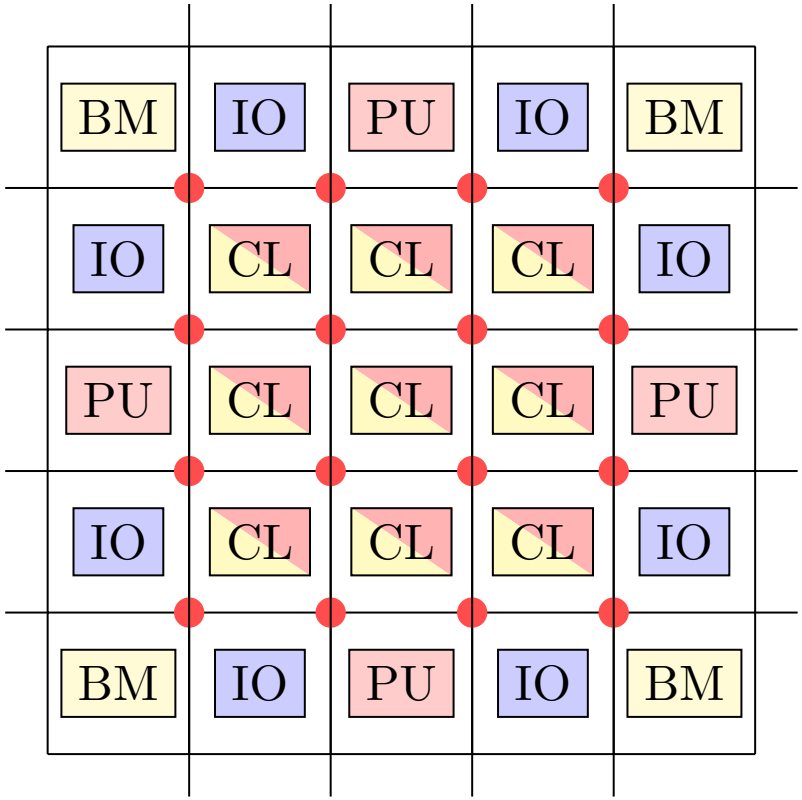
Co-Processor: Actual chip with memory interface



Co-Processor: Actual chip with memory interface

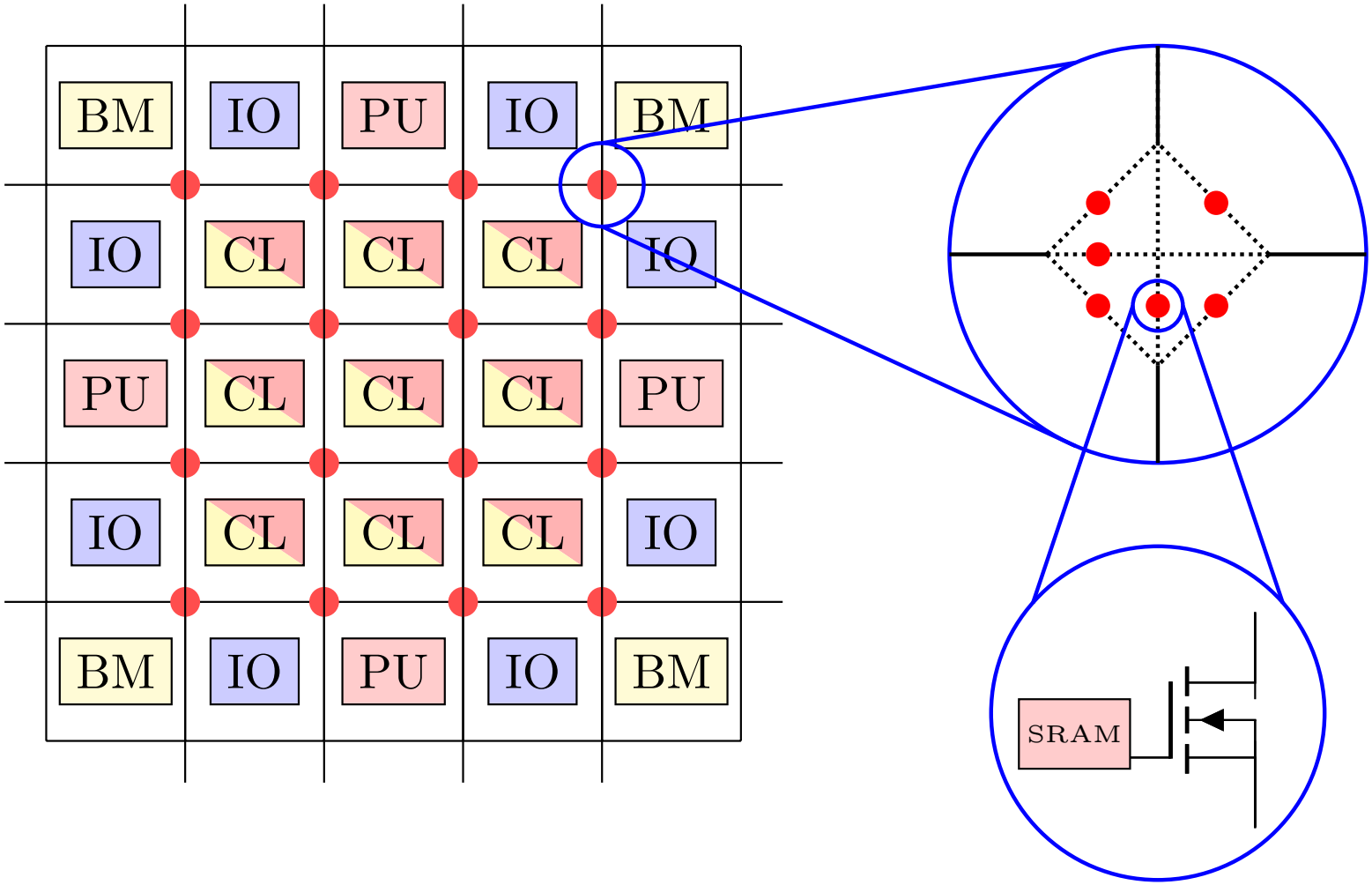


FPGA: How does it work?

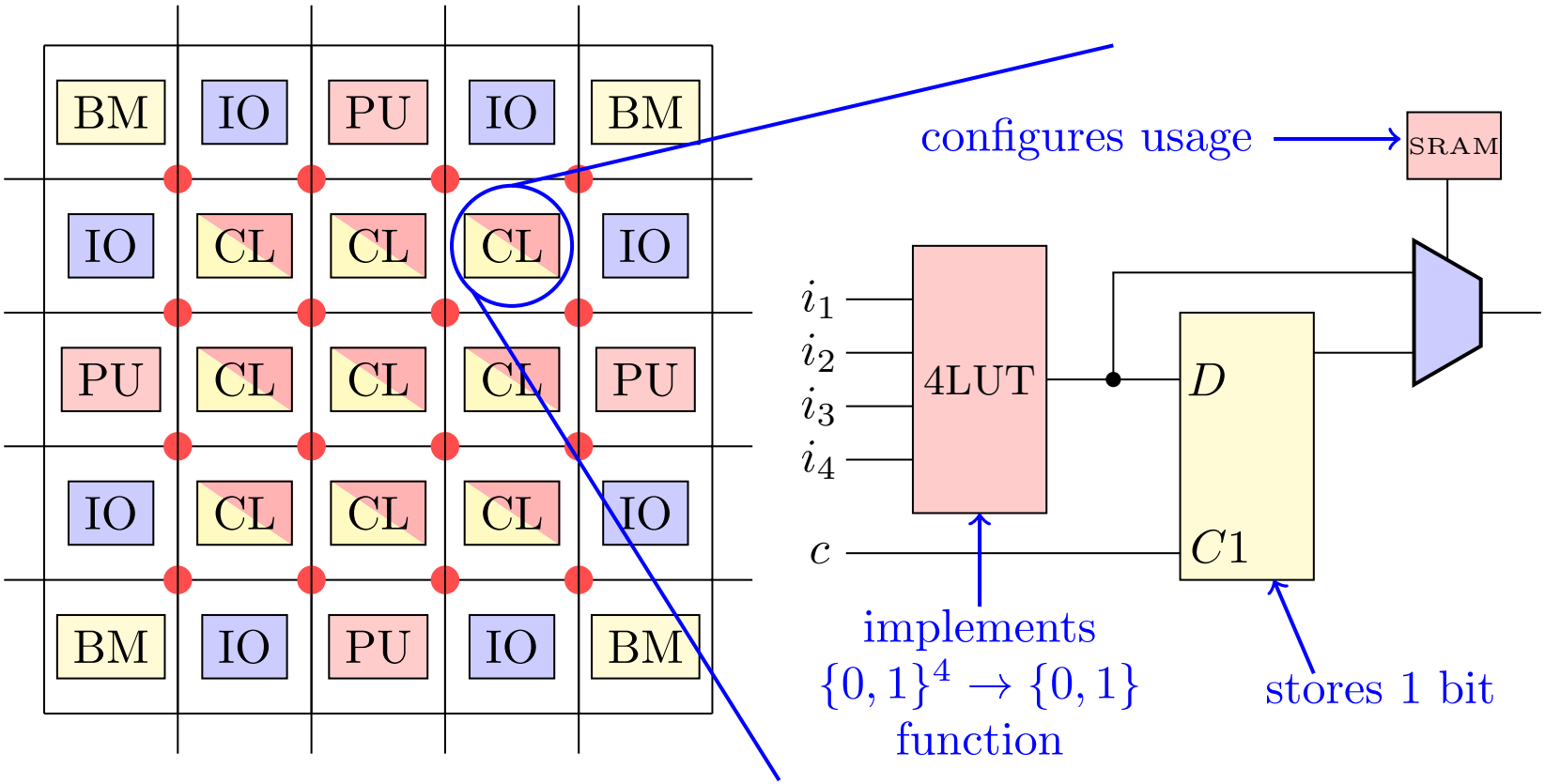


- chip layout 2D grid
- configurable connections between blocks
- configurable logic blocks (CL)
- input/output blocks (IO)
- hard-wired on boards with standard interface
- programmed and flashed with external PC

FPGA: Signal Routing



FPGA: Configurable Logic Block



FPGAs: Strengths

- **Inherent parallelism:** We can perform computations in **real** parallel in any level of granularity

FPGAs: Strengths

- **Inherent parallelism:** We can perform computations in **real** parallel in any level of granularity
- **Large on-chip memory:** Modern CPUs offer caches in the range of ~ 8Mb. Today's largest FPGA chips offer on-chip memory in the range of ~ 64 Mb

FPGAs: Strengths

- **Inherent parallelism:** We can perform computations in **real** parallel in any level of granularity
- **Large on-chip memory:** Modern CPUs offer caches in the range of ~ 8Mb. Today's largest FPGA chips offer on-chip memory in the range of ~ 64 Mb
- **Arbitrary word sizes:** Modern CPUs and GPUs are built and optimized for specific word sizes, e.g. 64 bit. In FPGAs, the word size is arbitrary and can fit the problem given

FPGAs: Strengths

- **Inherent parallelism:** We can perform computations in **real** parallel in any level of granularity
- **Large on-chip memory:** Modern CPUs offer caches in the range of ~ 8Mb. Today's largest FPGA chips offer on-chip memory in the range of ~ 64 Mb
- **Arbitrary word sizes:** Modern CPUs and GPUs are built and optimized for specific word sizes, e.g. 64 bit. In FPGAs, the word size is arbitrary and can fit the problem given
- **Large IO capabilities:** Modern CPUs and GPUs have to use PCIe and direct memory access (DMA) for data IO. FPGAs are free to use what's necessary.

FPGAs: Weaknesses

- **Slow clock rate:** CPUs / GPUs are clocked with $\sim 2 - 3$ GHz, FPGAs with ~ 200 Mhz

FPGAs: Weaknesses

- **Slow clock rate:** CPUs / GPUs are clocked with $\sim 2 - 3$ GHz, FPGAs with ~ 200 Mhz
- **No abstractions:** CPUs / GPUs offer a stack and a heap with data addressing etc. FPGAs just offer raw hardware

FPGAs: Weaknesses

- **Slow clock rate:** CPUs / GPUs are clocked with $\sim 2-3$ GHz, FPGAs with ~ 200 Mhz
- **No abstractions:** CPUs / GPUs offer a stack and a heap with data addressing etc. FPGAs just offer raw hardware
- **No optimizations:** CPUs / GPUs offer a well developed tool-chain support. Additionally, modern CPUs/GPUs often offer specialized hardware instructions

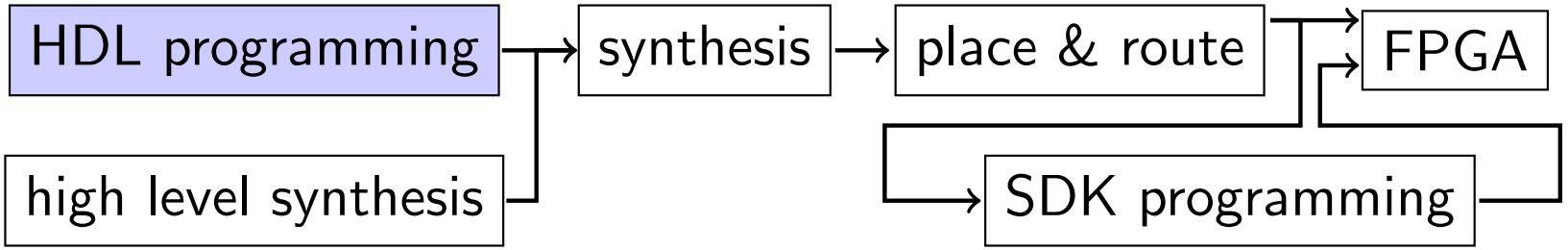
Note 1: High-end FPGAs offer clock rates around 800 Mhz
Note 2: High-end FPGAs also offer specialized hardware blocks, e.g. digital processing units or floating point units

FPGAs: Weaknesses

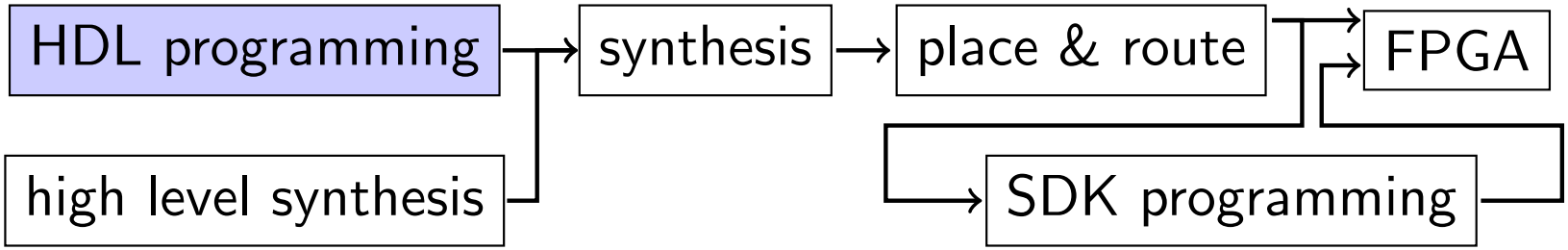
- **Slow clock rate:** CPUs / GPUs are clocked with $\sim 2-3$ GHz, FPGAs with ~ 200 Mhz
- **No abstractions:** CPUs / GPUs offer a stack and a heap with data addressing etc. FPGAs just offer raw hardware
- **No optimizations:** CPUs / GPUs offer a well developed tool-chain support. Additionally, modern CPUs/GPUs often offer specialized hardware instructions

Note 1: High-end FPGAs offer clock rates around 800 Mhz
Note 2: High-end FPGAs also offer specialized hardware blocks, e.g. digital processing units or floating point units
Note 3: Tool support for FPGAs are growing. The so-called 3rd wave of tools finally enables FPGAs for the mass-market

FPGA: Workflow



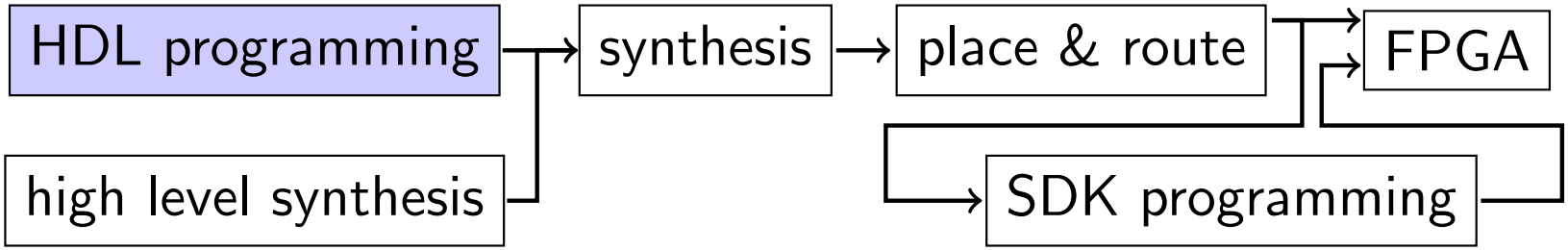
FPGA: Workflow



Hardware Description Languages (HDL):

- describe hardware on transistor and gate level
- modelling real concurrency
- modelling signal flow & timings
- low level bit operations
- high level operations like sums, products, ...
- verified using simulator

FPGA: Workflow

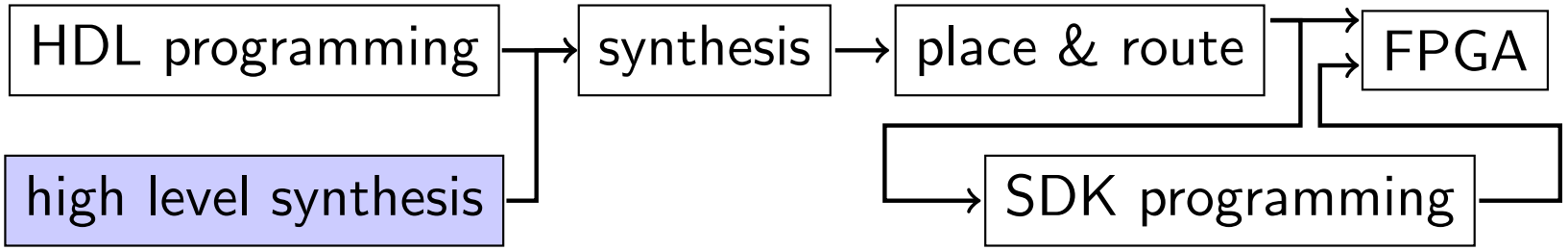


Hardware Description Languages (HDL):

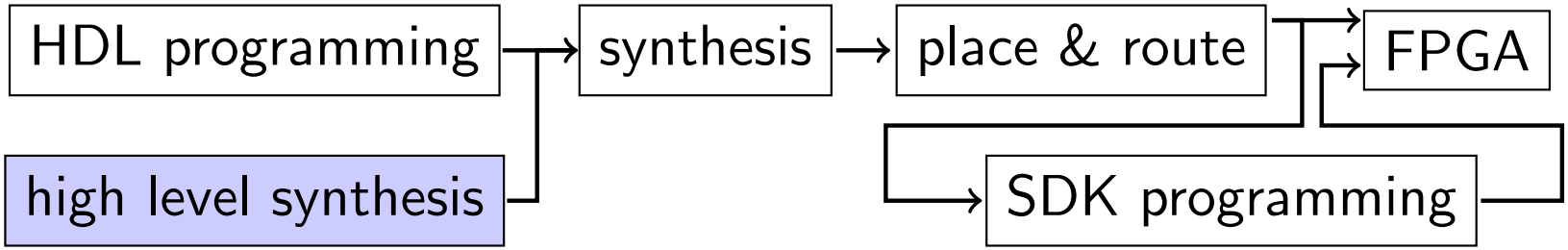
- describe hardware on transistor and gate level
- modelling real concurrency
- modelling signal flow & timings
- low level bit operations
- high level operations like sums, products, ...
- verified using simulator

Note: HDLs are used by hardware designers. HDLs are extremely low-level, but allow ultimate control over your design
But: HDL designs need time and care → We focus on HLS

FPGA: Workflow



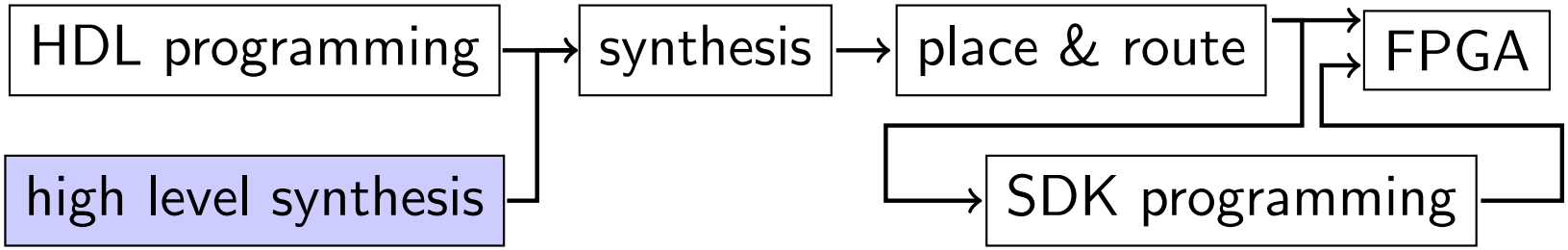
FPGA: Workflow



Basic idea: Automatically translate high level code into HDL

- Automate tedious work
- Compile code specifically for target device
- Lets you explore design space effectively
- Output should be reviewed
- Code must be changed for HLS tool
- Only works on subset of high level language

FPGA: Workflow

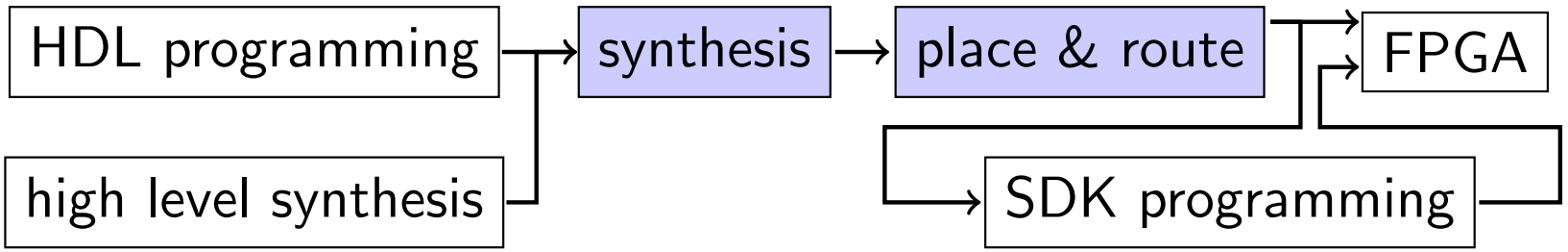


Basic idea: Automatically translate high level code into HDL

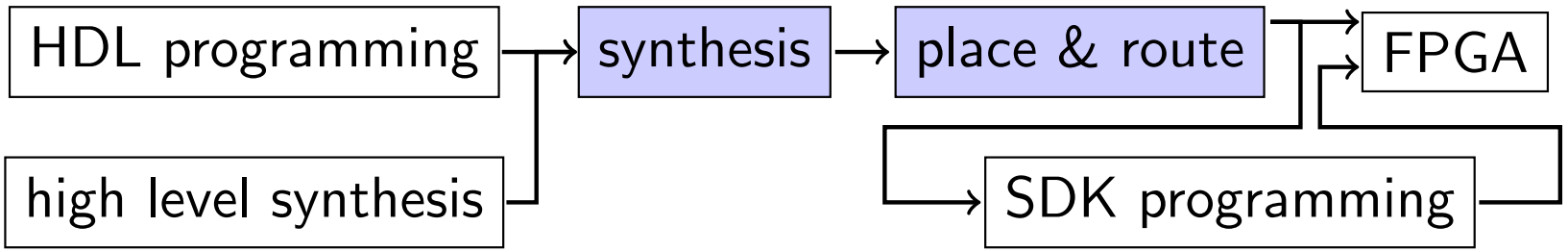
- Automate tedious work
- Compile code specifically for target device
- Lets you explore design space effectively
- Output should be reviewed
- Code must be changed for HLS tool
- Only works on subset of high level language

Note: HLS lets you describe your hardware in C-Code and the HLS tool will try to guess what you code meant and put that on the FPGA (more later)

FPGA: Workflow



FPGA: Workflow



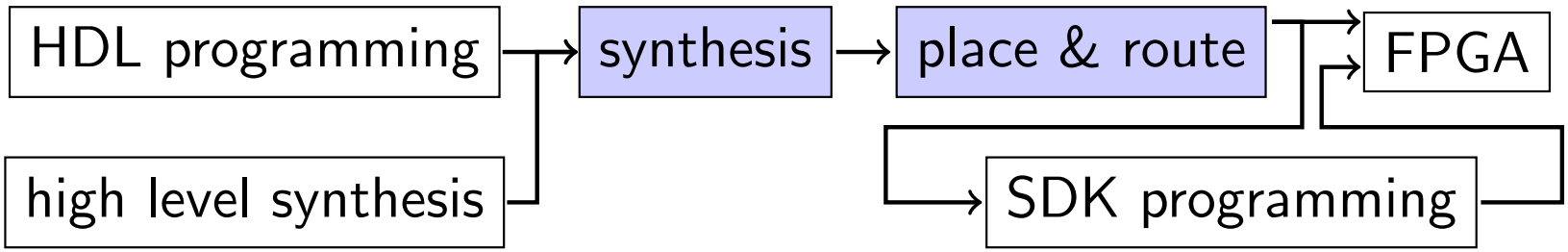
Synthesis: Calculate CL configurations

→ **So far:** HDL contains abstractions, e.g. summation

→ **Thus:** Compile these to a gate description, e.g. half/full-adder

⇒ The netlist contains the functionality of all units of the design

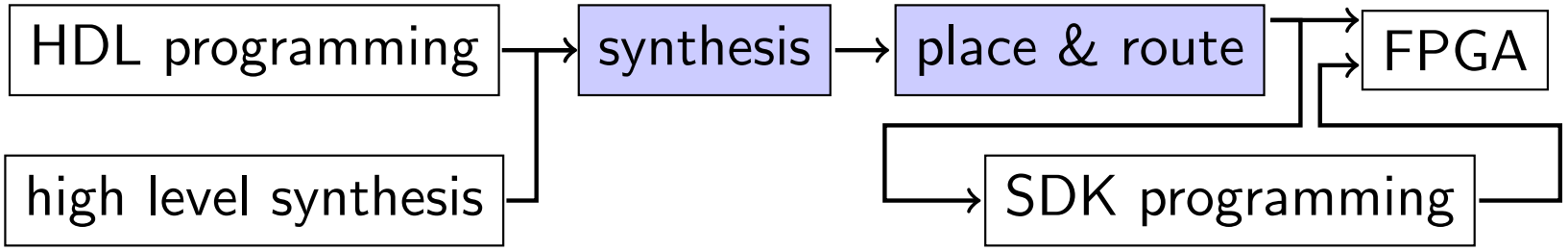
FPGA: Workflow



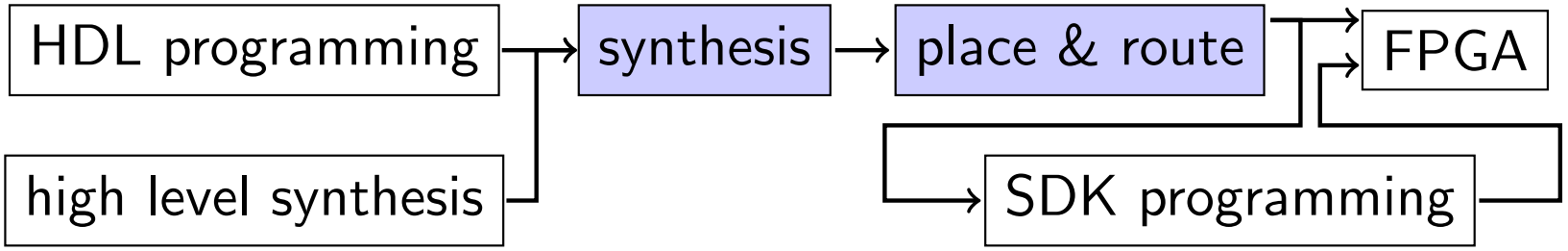
Synthesis: Calculate CL configurations
→ **So far:** HDL contains abstractions, e.g. summation
→ **Thus:** Compile these to a gate description, e.g. half/full-adder
⇒ The netlist contains the functionality of all units of the design

Place & Route: Calculate signal routing
→**So far:** We have netlist with all functional units of our design
⇒ Calculate, which CL implements which functionality and how they are connected

FPGA: Workflow

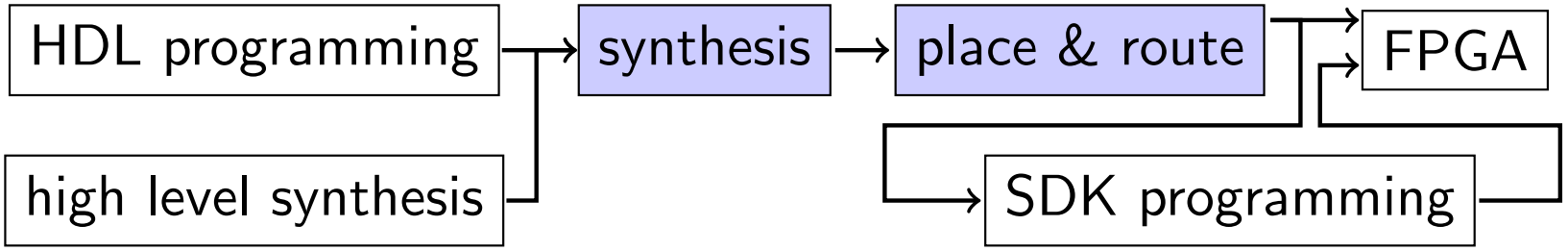


FPGA: Workflow



Important: Synthesis and place & route may fail!

FPGA: Workflow

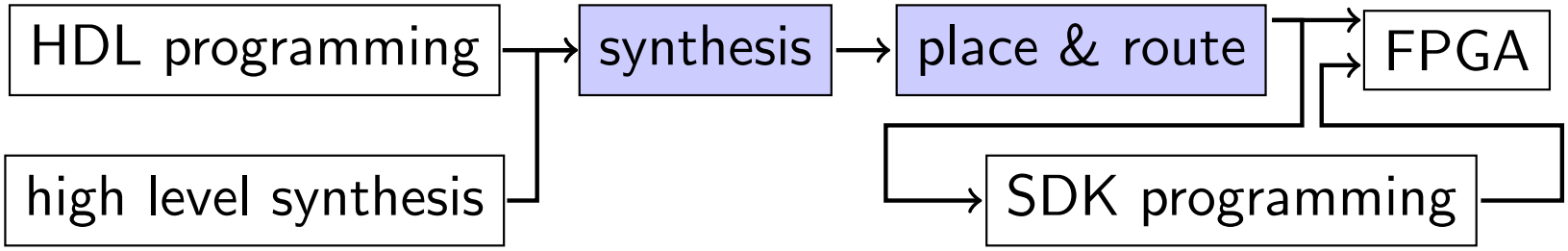


Important: Synthesis and place & route may fail!

Observation 1: HDL and HLS allow us to express things, which are not existent in hardware, e.g. files

Observation 2: Hardware is usually clocked. Place & route may fail to provide the necessary timings to achieve the given clock.

FPGA: Workflow

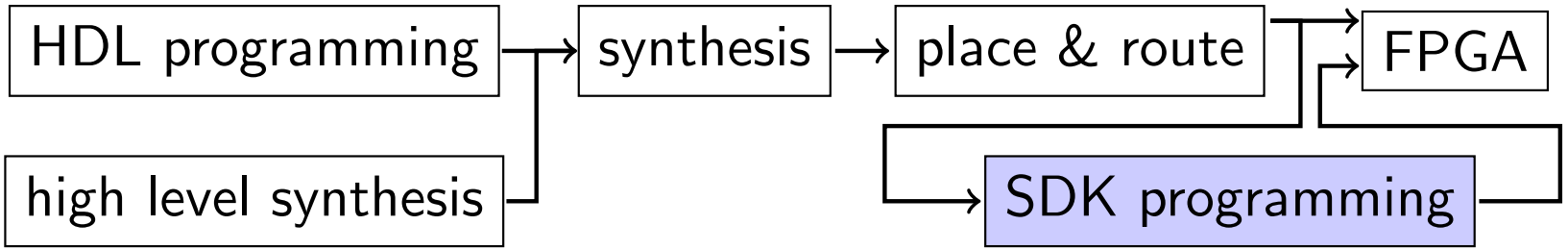


Important: Synthesis and place & route may fail!

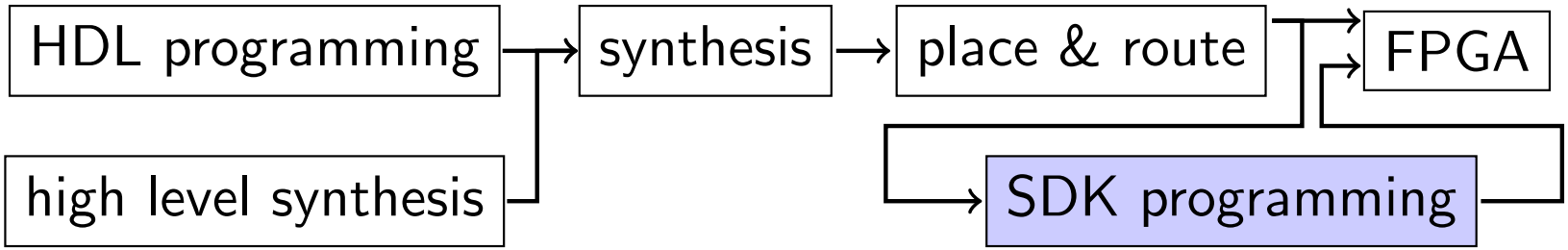
Observation 1: HDL and HLS allow us to express things, which are not existent in hardware, e.g. files
Observation 2: Hardware is usually clocked. Place & route may fail to provide the necessary timings to achieve the given clock.

Note 1: We aim for a clock around $125 - 150$ Mhz.
Note 2: Synthesis and place & route perform a lot of optimizations. Thus this phase is slow (minutes - hours)

FPGA: Workflow



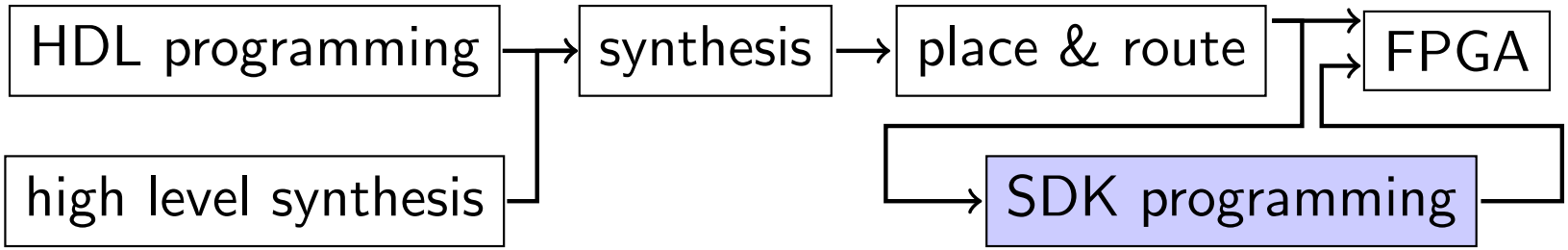
FPGA: Workflow



Observation 1: We can use IP from other programmers[1]

[1] E.g. http://opencores.com/

FPGA: Workflow



Observation 1: We can use IP from other programmers[1]

Observation 2: There are so-called soft processors

- Small processors with own ISA
- Mostly configurable in terms of Caches, Pipelining etc.
- Different optimizations for energy or throughput available
- Usually programmed in C-like language with own compiler

[1] E.g. http://opencores.com/

FPGAs as Co-Processors

So: For what do be use FPGAs?

- CPUs are optimized towards latency
  $\rightarrow$ Execute a single operation as fast as possible
- GPUs are optimized towards throughput
  $\rightarrow$ Process as much data a as fast as possible
- FPGAs are optimized towards ?

FPGAs as Co-Processors

So: For what do be use FPGAs?

- CPUs are optimized towards latency
  $\rightarrow$ Execute a single operation as fast as possible
- GPUs are optimized towards throughput
  $\rightarrow$ Process as much data a as fast as possible
- FPGAs are optimized towards ?

Fact: CPU and GPU designers are smart people!
$\Rightarrow$ It is though to beat a CPU / GPU only with an FPGA

FPGAs as Co-Processors

So: For what do be use FPGAs?

- CPUs are optimized towards latency
  $\rightarrow$ Execute a single operation as fast as possible
- GPUs are optimized towards throughput
  $\rightarrow$ Process as much data a as fast as possible
- FPGAs are optimized towards ?

Fact: CPU and GPU designers are smart people!
$\Rightarrow$ It is though to beat a CPU / GPU only with an FPGA

Rule-of-thumb: CPU is good for control flow
FPGAs / GPUs are good for number crunching
Thus: Combine FPGAs with CPUs

FPGAs as Co-Processors

Either: As PCIe cards in desktop / server systems

- Needs a custom written driver for PCIe
- Usually needs special licenses on FPGA chip or own PCIe protocol implementation
- Requires full desktop system

FPGAs as Co-Processors

Either: As PCIe cards in desktop / server systems

- Needs a custom written driver for PCIe
- Usually needs special licenses on FPGA chip or own PCIe protocol implementation
- Requires full desktop system

Or: fully integrated on development boards

- On-board connections are known, thus 1 driver needed
- Does not require full desktop system $\Rightarrow$ Less energy

FPGAs as Co-Processors

Either: As PCIe cards in desktop / server systems

- Needs a custom written driver for PCIe
- Usually needs special licenses on FPGA chip or own PCIe protocol implementation
- Requires full desktop system

Or: fully integrated on development boards

- On-board connections are known, thus 1 driver needed
- Does not require full desktop system $\Rightarrow$ Less energy

Our focus: Embedded boards with FPGA Co-Processors

Xilinx Zedboard

Board: Xilinx ZedBoard

- **ARM Cortex-A9** Dual Core CPU with 666 Mhz
- **RAM:** 512 Mb DDR RAM
- **Memory:** 512 Kb Cache

FPGA: Xilinx Artix-7 Z-7020

- **LUT:** 53200
- **CLB:** 83000
- **Block-Ram:** 4.9 Mb
- **DSP:** 220

Xilinx Zedboard

Board: Xilinx ZedBoard

- **ARM Cortex-A9** Dual Core CPU with 666 Mhz
- **RAM:** 512 Mb DDR RAM
- **Memory:** 512 Kb Cache

FPGA: Xilinx Artix-7 Z-7020

- **LUT:** 53200
- **CLB:** 83000
- **Block-Ram:** 4.9 Mb
- **DSP:** 220

Idea: Run full blown Linux on CPU and connect with FPGA
Thus: Easy software development for "glue" code + fast energy and efficient computations

Xilinx Zedboard

Board: Xilinx ZedBoard

- **ARM Cortex-A9** Dual Core CPU with 666 Mhz
- **RAM:** 512 Mb `DDR RAM`
- **Memory:** 512 Kb Cache

FPGA: Xilinx Artix-7 `Z-7020`

- **LUT:** 53200
- **CLB:** 83000
- **Block-Ram:** 4.9 Mb
- **DSP:** 220

Idea: Run full blown Linux on CPU and connect with FPGA
Thus: Easy software development for "glue" code + fast energy and efficient computations

CPU: Programmed in `C`
FPGA: Programmed in `C` or `VHDL/Verilog`

Xilinx Zedboard

Board: Xilinx ZedBoard

- **ARM Cortex-A9** Dual Core CPU with 666 Mhz
- **RAM:** 512 Mb `DDR RAM`
- **Memory:** 512 Kb Cache

FPGA: Xilinx Artix-7 `Z-7020`

- **LUT:** 53200
- **CLB:** 83000
- **Block-Ram:** 4.9 Mb
- **DSP:** 220

Idea: Run full blown Linux on CPU and connect with FPGA
Thus: Easy software development for "glue" code + fast energy and efficient computations

CPU: Programmed in `C`
FPGA: Programmed in `C` or `VHDL/Verilog`

Question: How do we combine both?

Software driven System on a Chip development (SDSoC)

Note: FPGA interface might change

Thus: Linux kernel driver needed for every new hardware block

$\rightarrow$ Writing Linux kernel drivers is a though task

Software driven System on a Chip development (SDSoC)

Note: FPGA interface might change

Thus: Linux kernel driver needed for every new hardware block

$\rightarrow$ Writing Linux kernel drivers is a though task

Thus: We use software for that: Xilinx SDSoC

- Standard eclipse GUI for C/C++ programming
- Standard gcc ARM compiler for C/C++ code
- HLS automatically compiles C/C++ code to HDL
- SDSoC generates a kernel driver based on the HLS' output

Software driven System on a Chip development (SDSoC)

Note: FPGA interface might change
Thus: Linux kernel driver needed for every new hardware block
$\rightarrow$ Writing Linux kernel drivers is a though task

Thus: We use software for that: Xilinx SDSoC

- Standard eclipse GUI for C/C++ programming
- Standard gcc ARM compiler for C/C++ code
- HLS automatically compiles C/C++ code to HDL
- SDSoC generates a kernel driver based on the HLS' output

In the end: We get a bootable Linux image for sd card with integrated hardware accelerator

AXI-Interface

Fact 1: The FPGA can support any hardware interface we desire
Fact 2: The ARMs hardware interface is fixed
$\Rightarrow$ The ARM and the FPGA are connected using the `AXI` interface

AXI-Interface

Fact 1: The FPGA can support any hardware interface we desire
Fact 2: The ARMs hardware interface is fixed
$\Rightarrow$ The ARM and the FPGA are connected using the `AXI` interface

`AXI` is part of the `AMBA` protocol stack. It specifies the way how system-on-a-chip components (CPU, RAM, FPGA...) should talk to each other. There are 3 variants:

- `AXI-Lite`: easy, simple communication
- `AXI-Stream`: high throughput in streaming settings
- `AXI`: high speed, low latency

AXI-Interface

Fact 1: The FPGA can support any hardware interface we desire
Fact 2: The ARMs hardware interface is fixed
$\Rightarrow$ The ARM and the FPGA are connected using the `AXI` interface

`AXI` is part of the `AMBA` protocol stack. It specifies the way how system-on-a-chip components (CPU, RAM, FPGA...) should talk to each other. There are 3 variants:

- `AXI-Lite`: easy, simple communication
- `AXI-Stream`: high throughput in streaming settings
- `AXI`: high speed, low latency

Note: HLS generates the desired interface for us

High Level Synthesis: Interface generation

```
1  #define PRAGMA_SUB(x) _Pragma (#x)
2  #define DO_PRAGMA(x) PRAGMA_SUB(x)
3  float diff(float const pX1[dim], float const pX2[dim]) const {
4  DO_PRAGMA(HLS INTERFACE s_axilite port=pX1 depth=dim);
5  DO_PRAGMA(HLS INTERFACE s_axilite port=pX2 depth=dim);
6  #pragma HLS INTERFACE s_axilite port=return
7
8          float sum = 0;
9          for (unsigned int i = 0; i < dim; ++i) {
10             sum += (pX1[i]-pX2[i])*(pX1[i]-pX2[i]);
11         }
12
13         return sum;
14 }
```

Note 1: In standard C "`bool predict(float const pX[dim])`" is the same as "`bool predict(float const *pX)`", but HLS explicitly needs to know the size!

High Level Synthesis: Interface generation

```
1  #define PRAGMA_SUB(x) _Pragma (#x)
2  #define DO_PRAGMA(x) PRAGMA_SUB(x)
3  float diff(float const pX1[dim], float const pX2[dim]) const {
4  DO_PRAGMA(HLS INTERFACE s_axilite port=pX1 depth=dim);
5  DO_PRAGMA(HLS INTERFACE s_axilite port=pX2 depth=dim);
6  #pragma HLS INTERFACE s_axilite port=return
7
8          float sum = 0;
9          for (unsigned int i = 0; i < dim; ++i) {
10             sum += (pX1[i]-pX2[i])*(pX1[i]-pX2[i]);
11         }
12
13         return sum;
14 }
```

Note 1: In standard C "`bool predict(float const pX[dim])`" is the same as "`bool predict(float const *pX)`", but HLS explicitly needs to know the size!

Note 2: We use a special `pragma` if we need to use parameters

High Level Synthesis: Interface generation

```
1  #define PRAGMA_SUB(x) _Pragma (#x)
2  #define DO_PRAGMA(x) PRAGMA_SUB(x)
3  float diff(float const pX1[dim], float const pX2[dim]) const {
4  DO_PRAGMA(HLS INTERFACE s_axilite port=pX1 depth=dim);
5  DO_PRAGMA(HLS INTERFACE s_axilite port=pX2 depth=dim);
6  #pragma HLS INTERFACE s_axilite port=return
7
8          float sum = 0;
9          for (unsigned int i = 0; i < dim; ++i) {
10             sum += (pX1[i]-pX2[i])*(pX1[i]-pX2[i]);
11         }
12
13         return sum;
14 }
```

Note 1: In standard C "`bool predict(float const pX[dim])`" is the same as "`bool predict(float const *pX)`", but HLS explicitly needs to know the size!

Note 2: We use a special `pragma` if we need to use parameters

Note 3: `s_axilite` can be replaced by `axis` for `axi-stream`

Deep Learning: Some considerations

Fact 1: DNNs have a lot of parameters
Fact 2: Many SGD steps are required to get reasonable results

Deep Learning: Some considerations

Fact 1: DNNs have a lot of parameters
Fact 2: Many SGD steps are required to get reasonable results

- We need a lot of data
- We need to learn a lot of parameters
- We need to perform many SGD steps until convergence

Deep Learning: Some considerations

Fact 1: DNNs have a lot of parameters
Fact 2: Many SGD steps are required to get reasonable results

- We need a lot of data
- We need to learn a lot of parameters
- We need to perform many SGD steps until convergence

Additional: We want to use Deep Learning in embedded context's, such as car, robots, etc.

Important: Model inference is different from model training
$\rightarrow$ Optimizations are task specific!

Deep Learning: A hardware perspective

Clear: DeepLearning greatly benefits from new and fast hardware
Note: This is well known. Many publications date back decades ago about specialized Neural-Network hardware

Deep Learning: A hardware perspective

Clear: DeepLearning greatly benefits from new and fast hardware
Note: This is well known. Many publications date back decades ago about specialized Neural-Network hardware

- **Until** 2010**:** Libs for NN mostly CPU based. Research for dedicated hardware available.
- **From** 2010**:** GPUs are widely available in mass-market. NN libs with GPUs backends become popular.

Deep Learning: A hardware perspective

Clear: DeepLearning greatly benefits from new and fast hardware
Note: This is well known. Many publications date back decades ago about specialized Neural-Network hardware

- **Until** 2010**:** Libs for NN mostly CPU based. Research for dedicated hardware available.
- **From** 2010**:** GPUs are widely available in mass-market. NN libs with GPUs backends become popular.
- **Upcoming:** More specialized hardware is being used
  - **Januar 2016:** Nvidias Drive PX2 for autonomous cars
  - **June 2016:** Googles Tensor Processing Unit (TPU)
  - **May 2017:** Googles Tensor Processing Unit 2.0 (TPU)

Currently: GPUs are state-of-the art - Why use FPGAs?

Deep Learning: Better performance per Watt

Ovtcharov et al. 2015 256×256 images

GPU 376 images / sec with 235 W $\Rightarrow$ 0.625 J / image

FPGA 134 images / sec with 25 W $\Rightarrow$ 0.1866 J / image

Deep Learning: Better performance per Watt

Ovtcharov et al. 2015 256×256 images

GPU 376 images / sec with 235 W $\Rightarrow$ 0.625 J / image

FPGA 134 images / sec with 25 W $\Rightarrow$ 0.1866 J / image

Qiu et al. 2016 482×415 images

GPU 57.9 images / sec with 250 W $\Rightarrow$ 4.31 J / image

FPGA 4.45 images / sec with 9.3 W $\Rightarrow$ 2.08 J / image

Deep Learning: Better performance per Watt

Ovtcharov et al. 2015 256×256 images

GPU 376 images / sec with 235 W $\Rightarrow$ 0.625 J / image

FPGA 134 images / sec with 25 W $\Rightarrow$ 0.1866 J / image

Qiu et al. 2016 482×415 images

GPU 57.9 images / sec with 250 W $\Rightarrow$ 4.31 J / image

FPGA 4.45 images / sec with 9.3 W $\Rightarrow$ 2.08 J / image

Thus: FPGAs may offer better performance per watt
Question: How do we bring DNNs to FPGAs?

Deep Learning on FPGAs

Design Goal Use on-chip memory whenever possible

Deep Learning on FPGAs

Design Goal Use on-chip memory whenever possible

Plan
Reduce size of network to ~ 10 Mb

AlexNet
~ 60 M parameter a 4 byte $\Rightarrow 240$ Mb model size

VGG-16
~ 130 M parameter a 4 byte $\Rightarrow 520$ Mb model size

Deep Learning on FPGAs

Design Goal Use on-chip memory whenever possible

Plan
Reduce size of network to ~ 10 Mb

AlexNet
~ 60 M parameter a 4 byte $\Rightarrow 240$ Mb model size

VGG-16
~ 130 M parameter a 4 byte $\Rightarrow 520$ Mb model size

Goal
Need to achieve compression ratio ≥ 20

Solution: Use smaller data types (1)

Gupta et al. 2015, Han et. al 2016, Gysel et. al 2016 ...

Reduction $32 \rightarrow 16$ bit nearly no performance difference in training

Reduction $32 \rightarrow 8$ bit nearly no performance difference in inference

Reduction $32 \rightarrow 2$ bit also possible, but requires special training

Solution: Use smaller data types (1)

Gupta et al. 2015, Han et. al 2016, Gysel et. al 2016 ...

Reduction $32 \rightarrow 16$ bit nearly no performance difference in training

Reduction $32 \rightarrow 8$ bit nearly no performance difference in inference

Reduction $32 \rightarrow 2$ bit also possible, but requires special training

Thus

Compression factor $2 - 4$ for free using fixed point

Solution: Use smaller data types (2)

Fixed point

$$X_{fx} = \overbrace{\underbrace{X_{(1)}X_{(0)}}_{N_l} \cdot \underbrace{X_{(-1)}X_{(-2)}X_{(-3)}X_{(-4)}}_{N_r}}^{N_t}$$

Implementation As scaled integer of size N_t, e.g. `char` for 8 bit

Solution: Use smaller data types (2)

Fixed point

$$X_{fx} = \overbrace{\underbrace{X_{(1)}X_{(0)}}_{N_l} \cdot \underbrace{X_{(-1)}X_{(-2)}X_{(-3)}X_{(-4)}}_{N_r}}^{N_t}$$

Implementation As scaled integer of size N_t, e.g. `char` for 8 bit

Fixed $\rightarrow$ float: $X_{fl} = \sum_{i=0}^{N_l} X_{(i)} 2^{i} + \sum_{i=-1}^{-N_r} X_{(-i)} 2^{-i}$

Float $\rightarrow$ fixed: $X_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor$

Solution: Use smaller data types (2)

Fixed point

$$X_{fx} = \overbrace{\underbrace{X_{(1)}X_{(0)}}_{N_l} \cdot \underbrace{X_{(-1)}X_{(-2)}X_{(-3)}X_{(-4)}}_{N_r}}^{N_t}$$

Implementation As scaled integer of size N_t, e.g. `char` for 8 bit

Fixed $\rightarrow$ float: $X_{fl} = \sum_{i=0}^{N_l} X_{(i)} 2^{i} + \sum_{i=-1}^{-N_r} X_{(-i)} 2^{-i}$

Float $\rightarrow$ fixed: $X_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor$

use shift operations!
$2^i = (1 << i)$
$2^{-i} = (1 >> i)$

Note In SDSoC there is a datatype `ap_fixed<`N_l`,`N_r`>`

Solution: Use smaller data types (3)

Addition/substraction No changes required

$$X''_{fx} = X_{fx} + X'_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor + \lfloor X'_{fl} \cdot 2^{N_r} \rfloor = \lfloor (X_{fl} + X'_{fl}) \cdot 2^{N_r} \rfloor$$

Solution: Use smaller data types (3)

Addition/substraction No changes required

$$X''_{fx} = X_{fx} + X'_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor + \lfloor X'_{fl} \cdot 2^{N_r} \rfloor = \lfloor (X_{fl} + X'_{fl}) \cdot 2^{N_r} \rfloor$$

Multiplication/Division Correct scaling

$$X_{fx} \cdot X'_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor \cdot \lfloor X'_{fl} \cdot 2^{N_r} \rfloor = \lfloor (X_{fl} \cdot X'_{fl}) \cdot (2^{N_r} \cdot 2^{N_r}) \rfloor$$
$$\Rightarrow X''_{fx} = X_{fx} \cdot X'_{fx} \cdot 2^{-N_r}$$

Solution: Use smaller data types (4)

Caution For training

SGD requires unbiased estimate of gradient

Solution: Use smaller data types (4)

Caution For training

SGD requires unbiased estimate of gradient

Fixed point $X_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor$

Gradient estimate is biased towards smaller number

Solution: Use smaller data types (4)

Caution For training

SGD requires unbiased estimate of gradient

Fixed point $X_{fx} = \lfloor X_{fl} \cdot 2^{N_r} \rfloor$

Gradient estimate is biased towards smaller number

Gupta et al. 2015 Stochastic rounding

$$X_{fx} = \begin{cases} \lfloor X_{fl} \cdot 2^{N_r} \rfloor & \text{with prob. p} \sim X_{fl} - \lfloor X_{fl} \rfloor \\ \lfloor X_{fl} \cdot 2^{N_r} + 2^{-N_r} \rfloor & \text{else} \end{cases}$$

Sidenote: The extreme case

Courbariaux et al. 2015, Hubara et al. 2016
Use binary / ternary weights during forward pass

Rastegari et al. 2016 XNOR-Net
Utilize binary operations for forward pass

Sidenote: The extreme case

Courbariaux et al. 2015, Hubara et al. 2016
Use binary / ternary weights during forward pass

Rastegari et al. 2016 XNOR-Net
Utilize binary operations for forward pass

But Train network using (quantized) real values

Sidenote: The extreme case

Courbariaux et al. 2015, Hubara et al. 2016
Use binary / ternary weights during forward pass

Rastegari et al. 2016 XNOR-Net
Utilize binary operations for forward pass

But Train network using (quantized) real values

So far No binary SGD

Sidenote: The extreme case

Courbariaux et al. 2015, Hubara et al. 2016
Use binary / ternary weights during forward pass

Rastegari et al. 2016 XNOR-Net
Utilize binary operations for forward pass

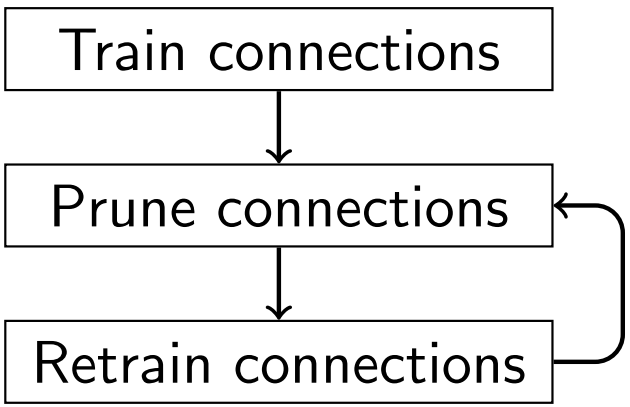
But Train network using (quantized) real values

So far No binary SGD

What about model inference?

For inference: Prune Connections

Han et al. 2015 Prune connections and retrain weights



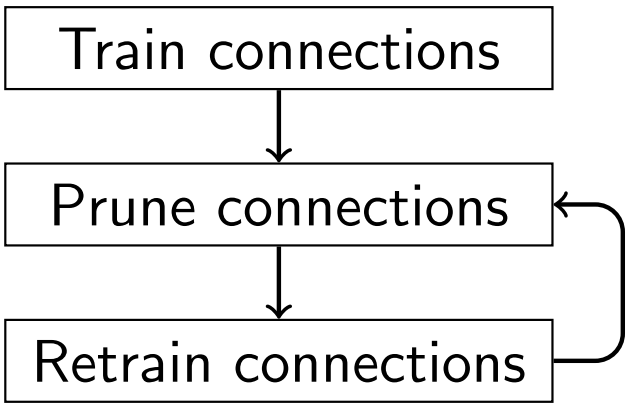
Train network as always

Delete all connections with $w_{i,j}^{l} \leq \tau^{l}$

Retrain network with deleted connections

For inference: Prune Connections

Han et al. 2015 Prune connections and retrain weights



Train network as always

Delete all connections with $w_{i,j}^{l} \leq \tau^{l}$

Retrain network with deleted connections

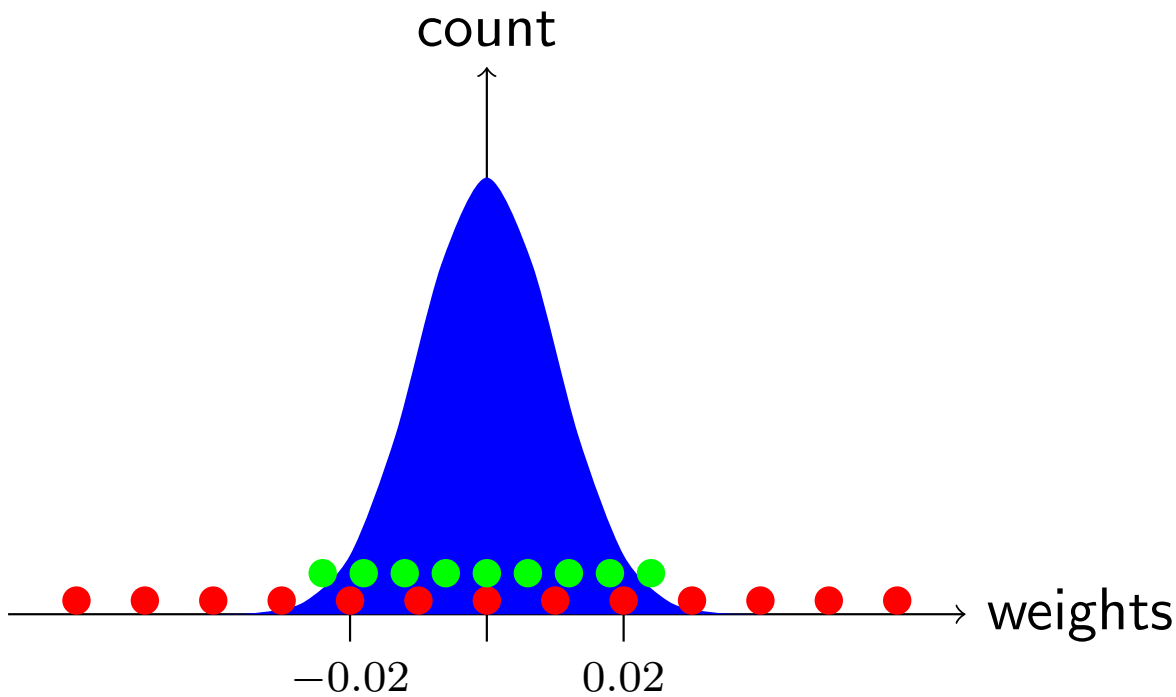
Note Delete neurons if not connected anymore

Results $9-13$x compression in overall size $+$ no loss in accuracy

But Extremely slow $+$ threshold τ^{l} not clear

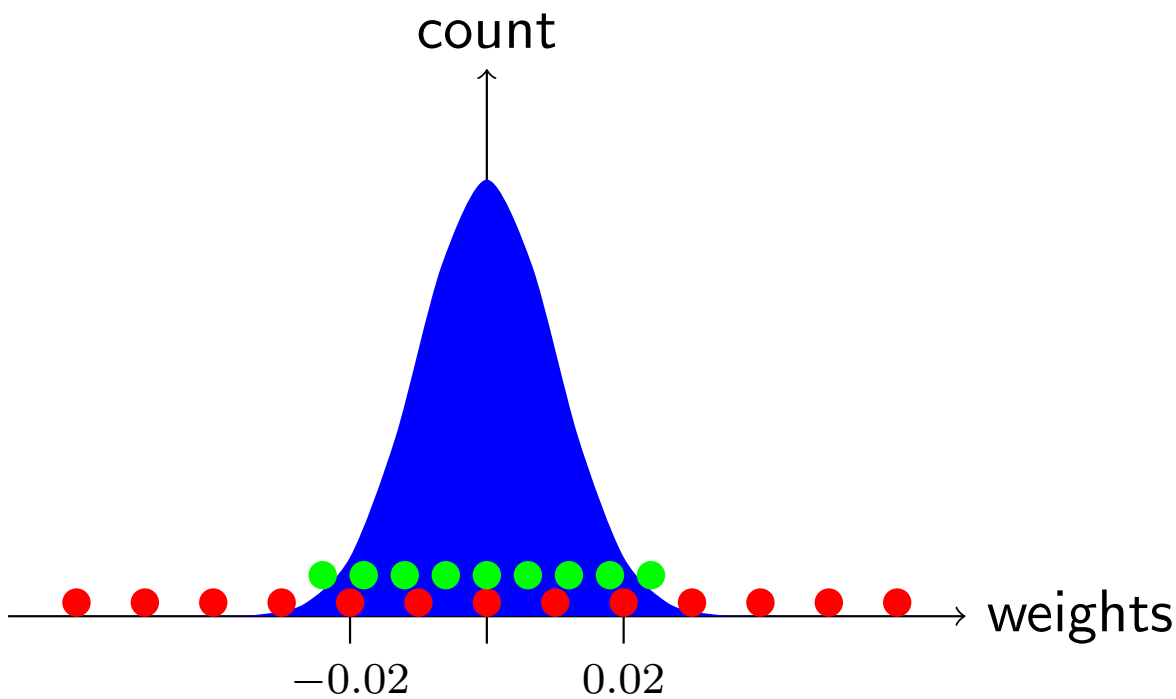
For inference: Distribution of weights

Han et al 2015 Weights are somewhat Gaussian distributed



For inference: Distribution of weights

Han et al 2015 Weights are somewhat Gaussian distributed

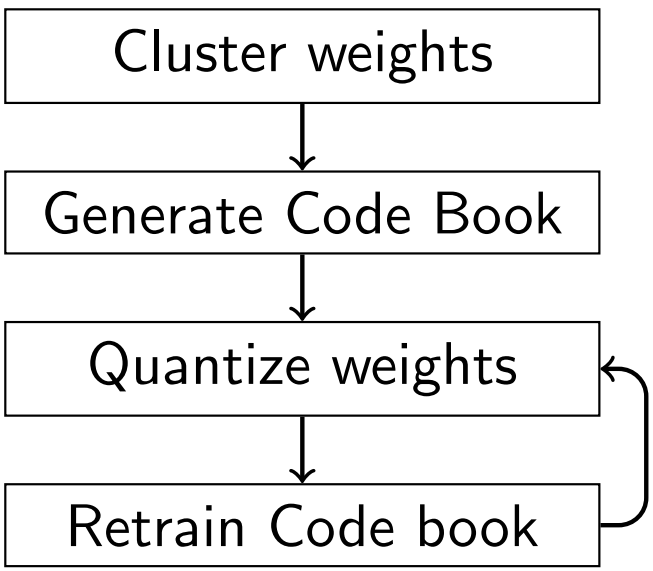


In red Fixed point quantization

In green Dynamic quantization

For inference: Clustering

Han et al 2015 / Han et al 2016 Cluster weights after training



Cluster weights

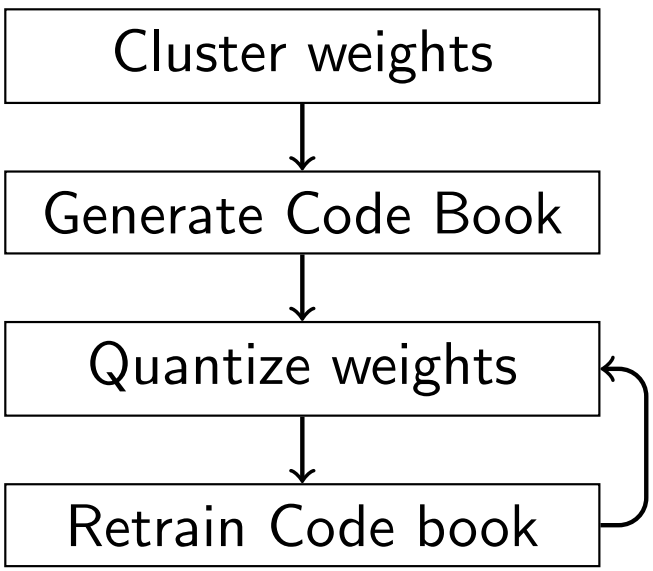
Extract centroids

Assign indexing scheme

Update shared weights jointly

For inference: Clustering

Han et al 2015 / Han et al 2016 Cluster weights after training



Cluster weights

Extract centroids

Assign indexing scheme

Update shared weights jointly

Results

~ 20x compression in overall size

Sidenote: Clustering

Goal: Find K "clusters" $c_1, \ldots, c_K$ in data $x_1, x_2, \ldots, x_N \in \mathbb{R}$
Mathematically: $\arg\min_{c_1,\ldots,c_K} \sum_{k=1}^{K} \sum_{i=1}^{N} (x_i - c_k)^2$

Idea Iterative algorithm

1: **for** $i = 1, 2, ..., K$ **do**
2: $c[i] = \texttt{random\_without\_replacement}(x_1, ..., x_N)$
3: **while** ERROR **do**
4: **for** $i = 1, 2, ..., K$ **do**
5: $cnt[i] = 0; tmp[i] = 0$
6: **for** $i = 1, 2, ..., N$ **do**
7: $m = \arg\min_{k=1,...,K}\{(x[i] - c[k])^2\}$
8: $cnt[m] = cnt[m] + 1; tmp[m] = tmp[m] + x[i]$

Note: Same framework as SGD

For inference: Distribution of clustered weights

Han et al 2016 Clustering changes weight distribution

For inference: Distribution of clustered weights

Han et al 2016 Clustering changes weight distribution

Idea

Use huffman encoding to further reduce size

Results Additional ~ 10x compression

For inference: Combining all approaches

Han et al 2016 Combining all three approaches

$35 - 49$x compression ratio

$3 - 4$x speed-up

$3 - 7$x less energy

No loss in accuracy

But Slow, due to post-processing

For inference: Combining all approaches

Han et al 2016 Combining all three approaches

$35 - 49$x compression ratio

$3 - 4$x speed-up

$3 - 7$x less energy

No loss in accuracy

But Slow, due to post-processing

Nowlan and Hinton 1992 Clustering during backpropagation
Seems to work, but also extremely slow

So far Not been done with todays hardware (?)

Deep Learning on FPGAs

Goal

Need to achieve compression ratio ≥ 20

So far

Compression ratio $2 - 4$ during training
Compression ratio $35 - 49$ after training

Solution: Use different NN structure

Fully Connected Layer needs a lot of parameters

Solution: Use different NN structure

Fully Connected Layer needs a lot of parameters

LeCun 1998

$28 \times 28 \rightarrow 300 \rightarrow 10 = 238200$ parameters

Solution: Use different NN structure

Fully Connected Layer needs a lot of parameters

LeCun 1998
$28 \times 28 \rightarrow 300 \rightarrow 10 = 238200$ parameters

Observation
8bit quantization gives $2^8 = 256$ different weights

Thus
~ 930 weights will be the same, if evenly distributed

Chen et al. 2015 Hashing trick
Randomly group weights together
$2 - 4$x compression without performance loss

Solution: Use different NN structure (2)

Recall

$$\delta_j^{(l-1)} = \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)}$$

Solution: Use different NN structure (2)

Recall

$$\delta_j^{(l-1)} = \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)}$$

For CPUs Use hash function h

$$\delta_j^{(l-1)} = \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{\phi(j,k)}^{(l)}$$

Solution: Use different NN structure (2)

Recall

$$\delta_j^{(l-1)} = \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{j,k}^{(l)}$$

For CPUs Use hash function h

$$\delta_j^{(l-1)} = \frac{\partial h(y_i^{(l-1)})}{\partial y_i^{(l-1)}} \sum_{k=1}^{M^{(l)}} \delta_k^{(l)} w_{\phi(j,k)}^{(l)}$$

For FPGAs

Compute hashing once offline + hard-code memory
Hash function ϕ Fast and easy, e.g. `xxHash`

Deep Learning on FPGAs

Design Goal Fine-tune implementation whenever possible

Deep Learning on FPGAs

Design Goal Fine-tune implementation whenever possible

Jouppi et al. 2016 TPU mostly built from systolic arrays

Deep Learning on FPGAs

Design Goal Fine-tune implementation whenever possible

Jouppi et al. 2016 TPU mostly built from systolic arrays

Idea
Use basic processing elements (PE)
Heavily pipeline computation in two dimensions

Deep Learning on FPGAs

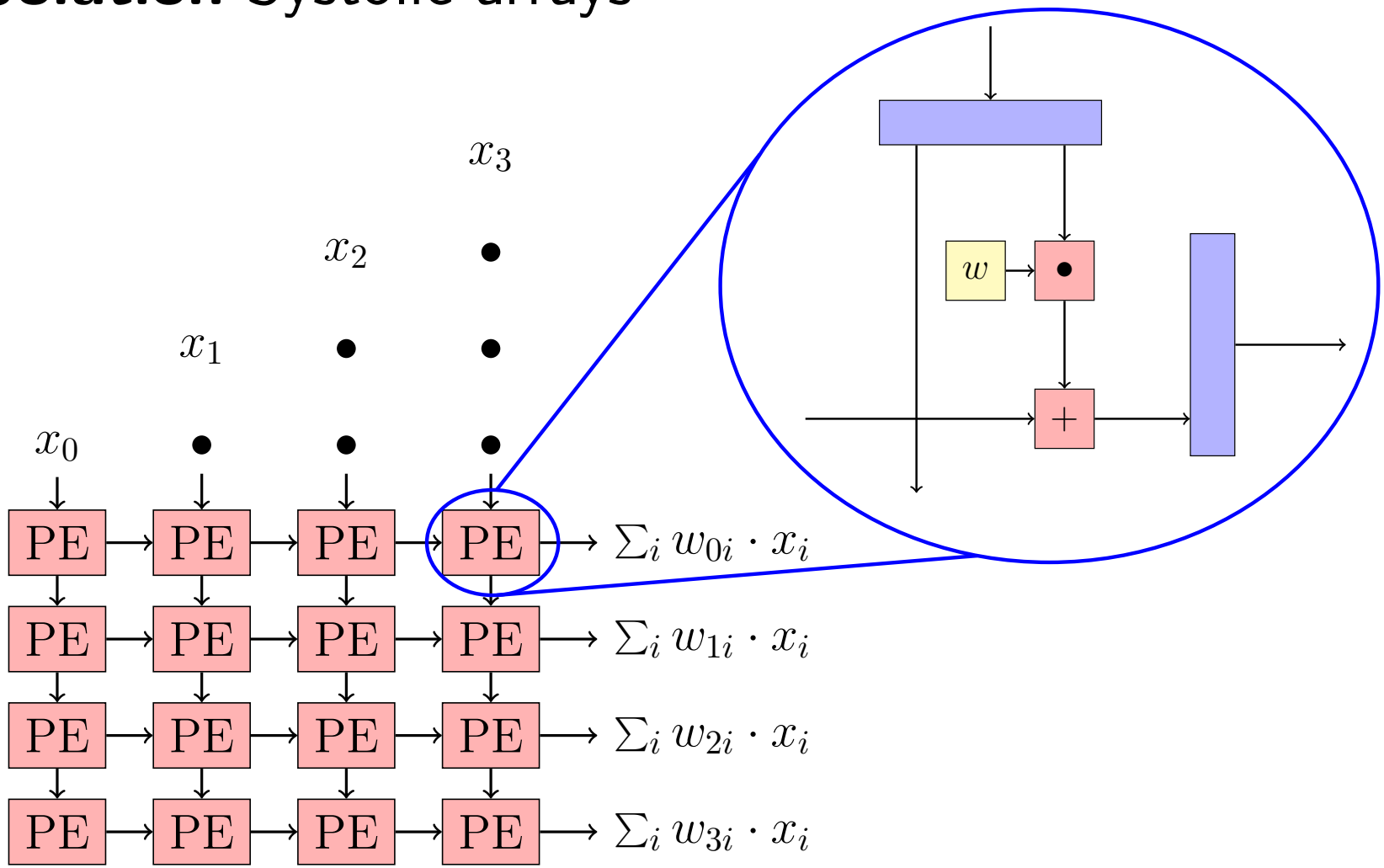
Design Goal Fine-tune implementation whenever possible

Jouppi et al. 2016 TPU mostly built from systolic arrays

Idea
Use basic processing elements (PE)
Heavily pipeline computation in two dimensions

Then
Local communication
Synchronization by design
High throughput

Solution Systolic arrays



Summary

Important concepts:

- **Moore's** law will expire around 2020
- **FPGAs** are programmable hardware circuits
- **FPGAs** work well with parallelism and custom data ranges
- **Use a** combination of CPU and FPGA
- **HLS** helps us to program FPGAs in a timely matter
- **Loop unrolling / Pipelining** are two possible optimizations
- **Reduce** communication between CPU and FPGA
- **Use** fixed floating point operations if possible

Homework (mandatory!!)

Implement backpropagation in C/C++ for the following network:

- Input: 784 (MNIST)
- Hidden: 300, ReLu activation
- Output: 10, sigmoid, MSE

Important 1: You will also need a CSV reader in C/C++
Important 2: Statically allocate all the memory needed - no malloc/new

Note: Expected accuracy is around 92%
This is mandatory: We will put parts of this network on the FPGA next session!

Homework (also mandatory!!)

What are your projects going to be? It should include

- preprocessing of data
- neural network architecture
- model training
- FPGA usage

Homework (also mandatory!!)

What are your projects going to be? It should include

- preprocessing of data
- neural network architecture
- model training
- FPGA usage

Important: You are free to choose your focus

Example 1: Focus on fast implementation for fully connected / convolution on FPGA. Tests mainly on MNIST

Example 2: Focus on Cats-Vs-Dogs data set. Use existing framework and do a lot of pre-processing / model training

Example 3: Focus on integration of FPGA. Use pre-trained model and compute inference on FPGA

Please: Ask me for help / advice!