DeepLearning on FPGAs

Introduction to FPGAs

Sebastian Buschjäger

Technische Universität Dortmund - Fakultät Informatik - Lehrstuhl 8

November 17, 2016

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

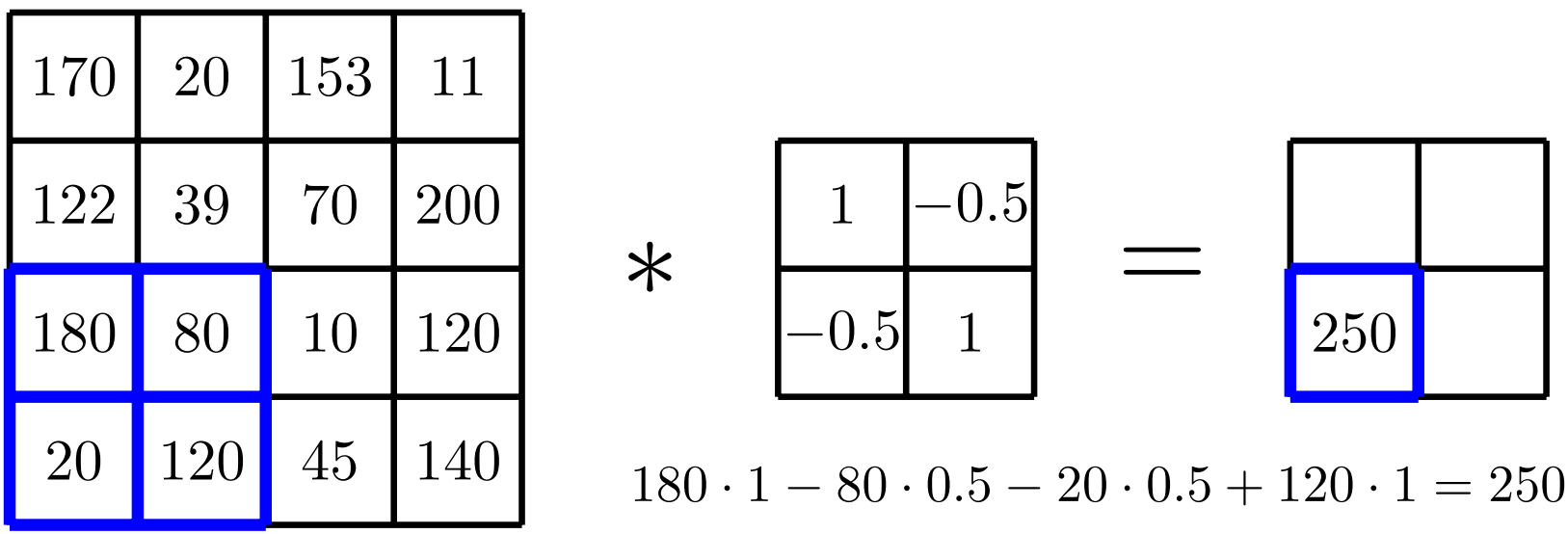
$=$

result

Recap: Convolution

Observation 1: Even smaller images need a lot of neurons
Our approach: Discrete convolution

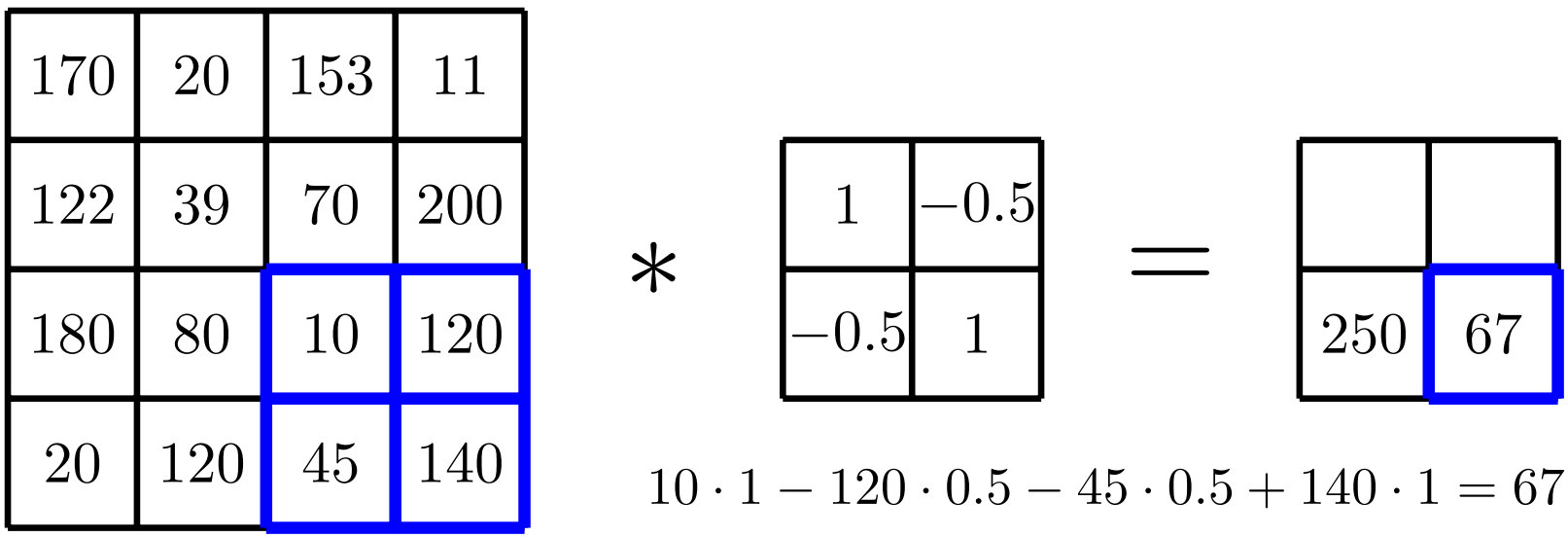
$$k_c = \sum_{i=1}^{r} w_i \cdot c_i = \vec{w} * \vec{c}$$



Recap: Convolution

Observation 1: Even smaller images need a lot of neurons
Our approach: Discrete convolution

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



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

image | kernel / weights / filter | result

Recap: Convolution

Observation 1: Even smaller images need a lot of neurons
Our approach: Discrete convolution

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

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 with sigmoid activation

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:

$$\delta^{(l+1)} = \delta^{(l)} * rot180(w^{(l+1)}) \cdot f_{i,j}^{(l)}(1 - f_{i,j})^l$$

Recap: Backpropagation for CNNs with sigmoid activation

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:

$$
\delta^{(l+1)} = \delta^{(l)} * rot180(w^{(l+1)}) \cdot f_{i,j}^{(l)}(1 - f_{i,j})^l
$$

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

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

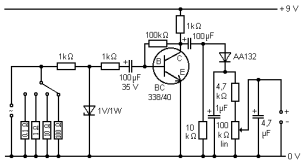
Fact 1: Engineering is currently producing $11 - 16nm$ transistors[1]
Side-Note: A $4nm$ transistor can be built from only 7 atoms!
Fact 2: The smaller transistors get, the more quantum effects are happening. Moore's law is predicted to expire with $5nm$ transistors

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

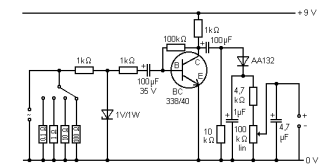
Fact:

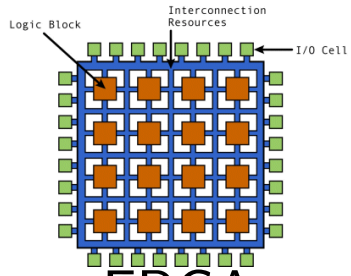
- speed: fastest
- energy: $\sim \mu$W
- application specific
- costs: expensive

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

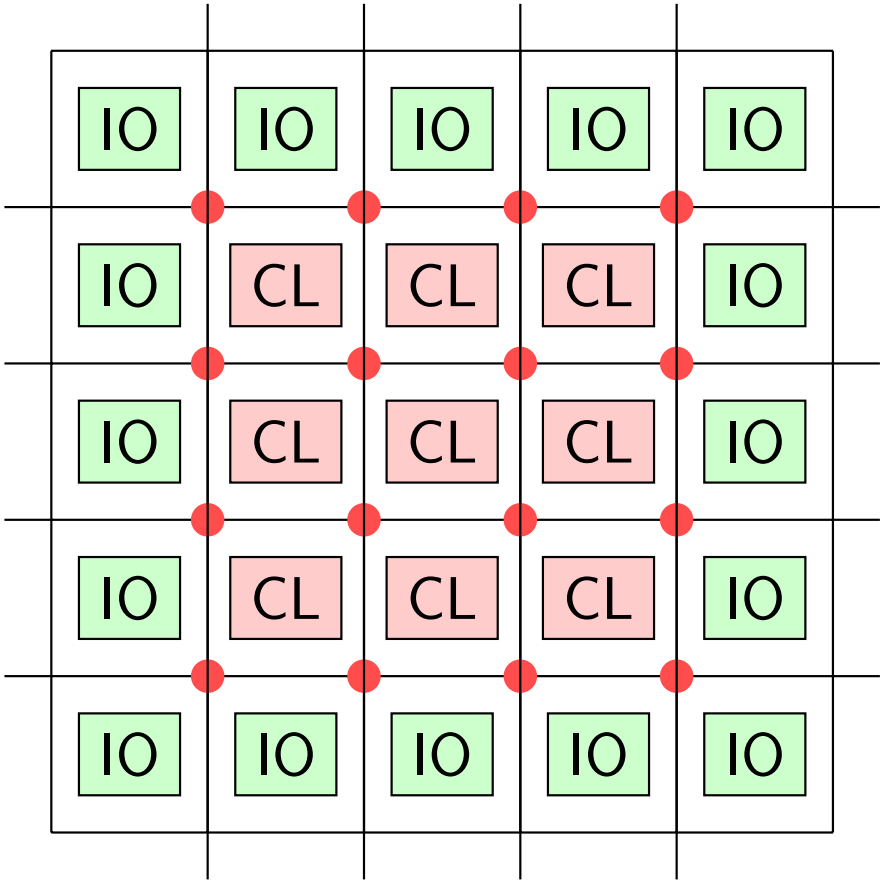
Hope:

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

Fact:

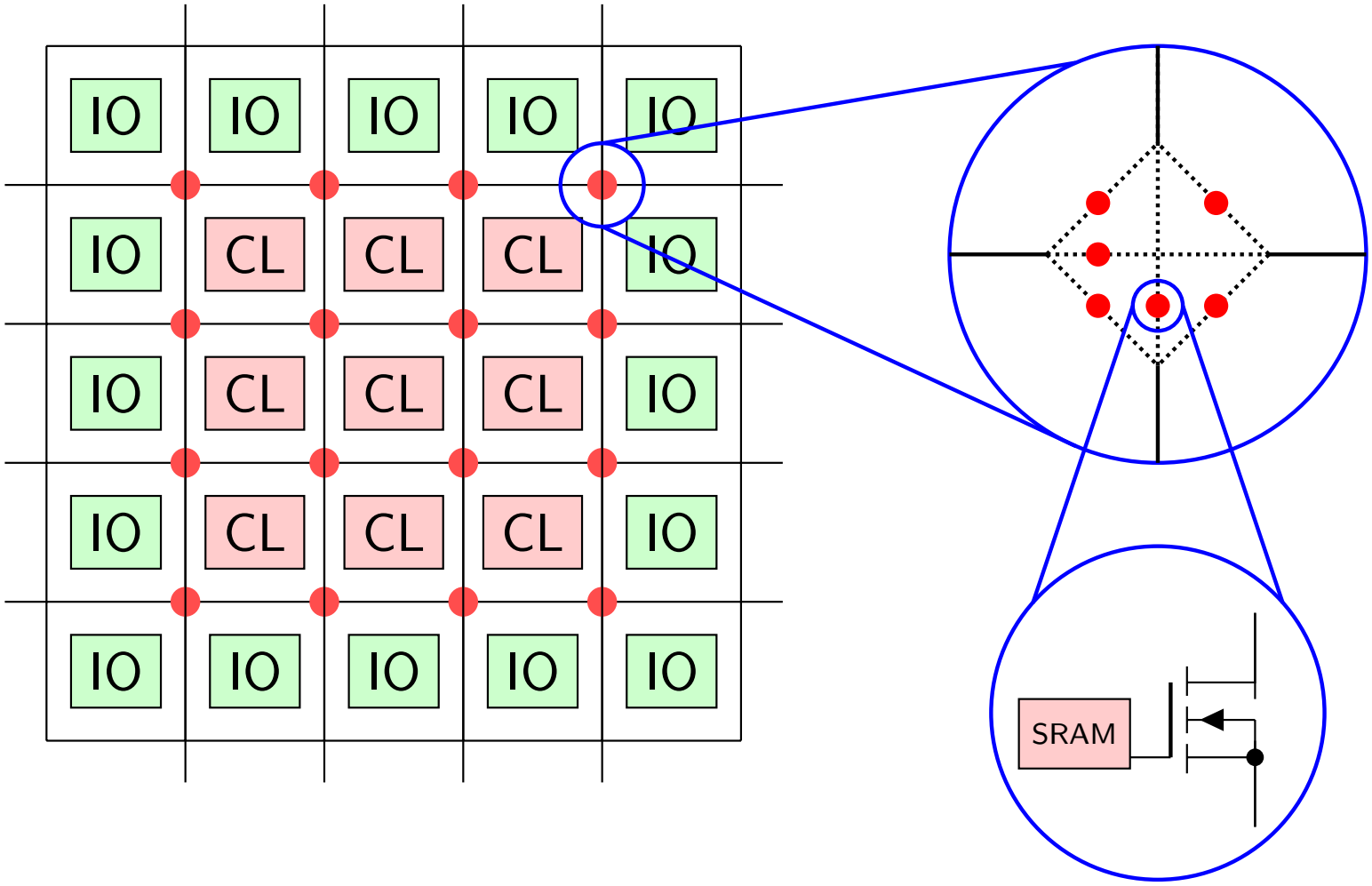
- speed: fast
- energy: $\sim$ W
- general purpose
- costs: cheap

FPGA: How does it work?

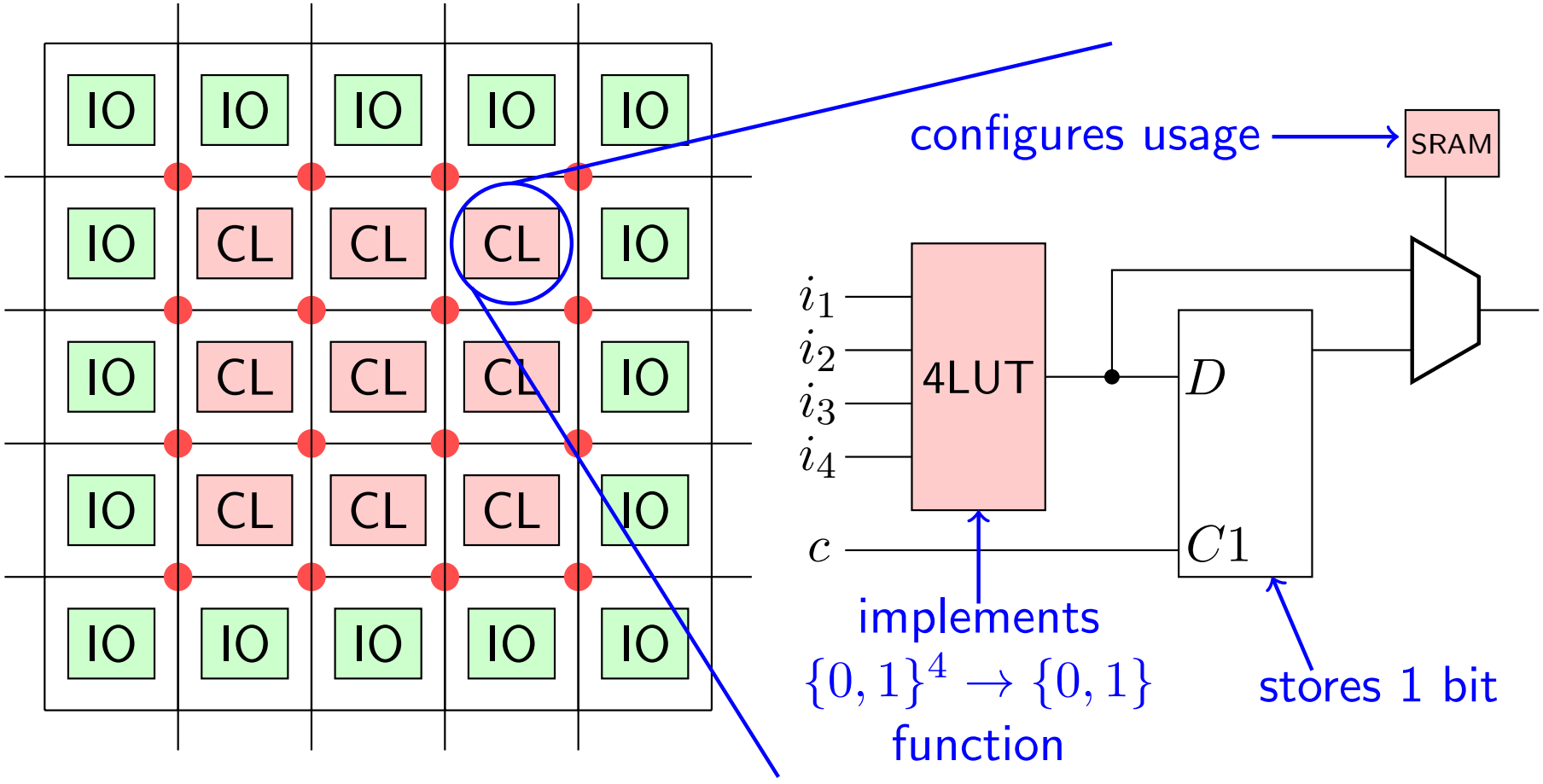


- chip layout 2D grid
- configurable connections between blocks
- configurable logic blocks (CL)
- input/output blocks (IO)
- hard-wired on boards with standard interface
- programmed and flashed with external PC

FPGA: Signal Routing



FPGA: Configurable Logic Block



FPGAs: Strengths

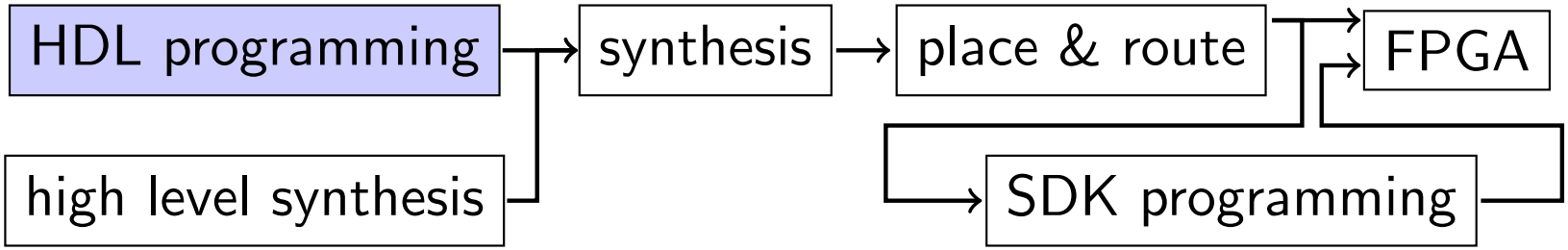
- **Inherent parallelism:** We can perform computations in **real** parallel on any level of granularity.
- **Large on-chip memory:** Modern CPUs offer Caches in the range of ~ 8Mb. Today's largest FPGA chips offer on-chip memory in the range of ~ 64 Mb
- **Arbitrary word sizes:** Modern CPUs and GPUs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

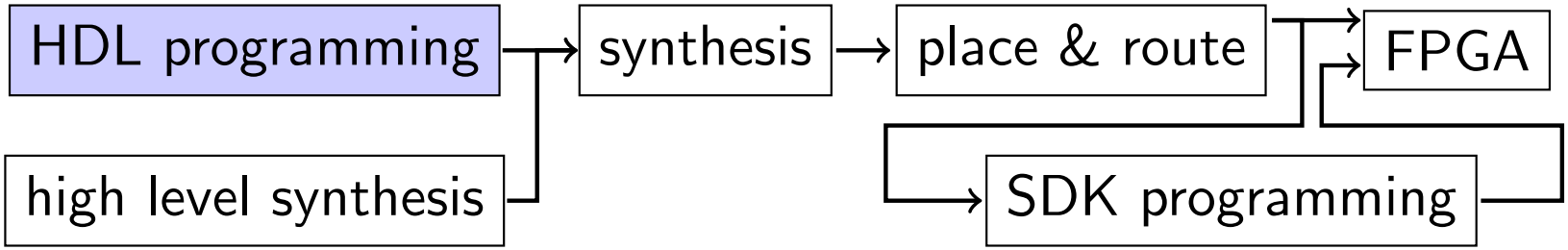
- **Slow clock rate:** CPUs / GPUs are clocked with $\sim 2-3$ GHz, FPGAs with ~ 200 Mhz
- **No abstractions:** CPUs / GPUs offer a stack and a heap with data addressing etc. FPGAs just offer raw hardware
- **No optimizations:** CPUs / GPUs offer a well developed tool-chain support. Additionally, modern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



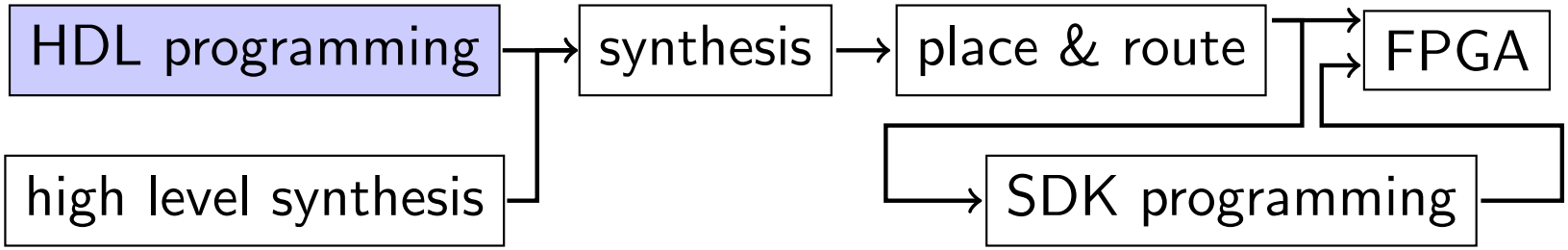
FPGA: Workflow



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

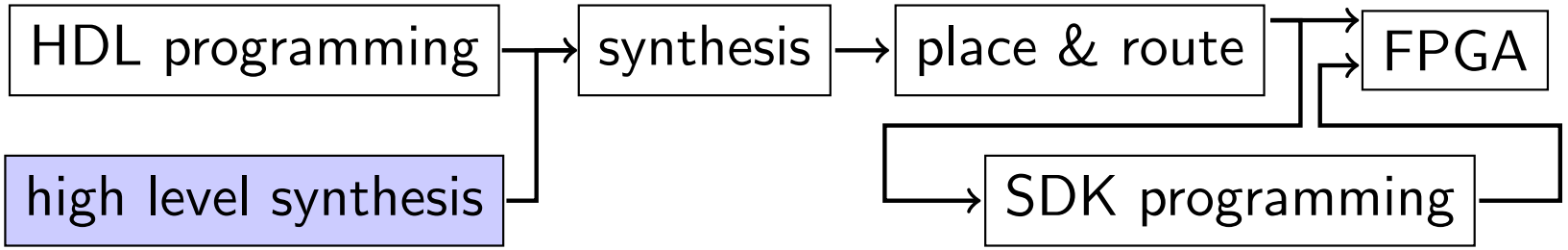


Hardware Description Languages (HDL):

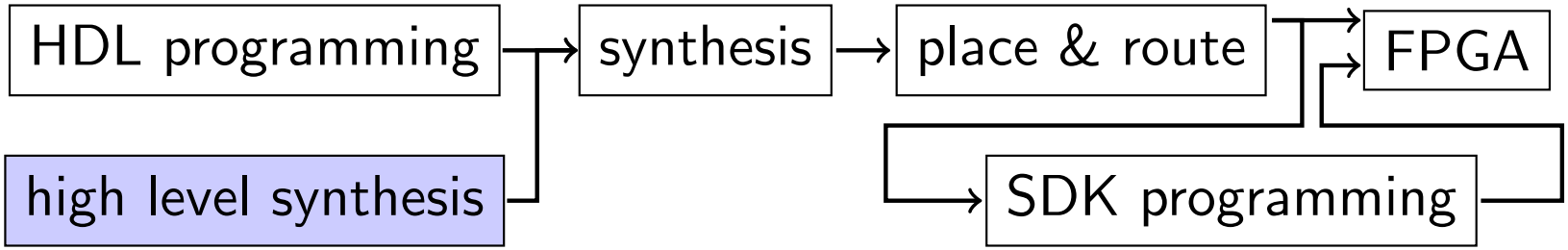
- describe hardware on transistor and gate level
- modelling real concurrency
- modelling signal flow & timings
- low level bit operations
- high level operations like sums, products, ...
- verified using simulator

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



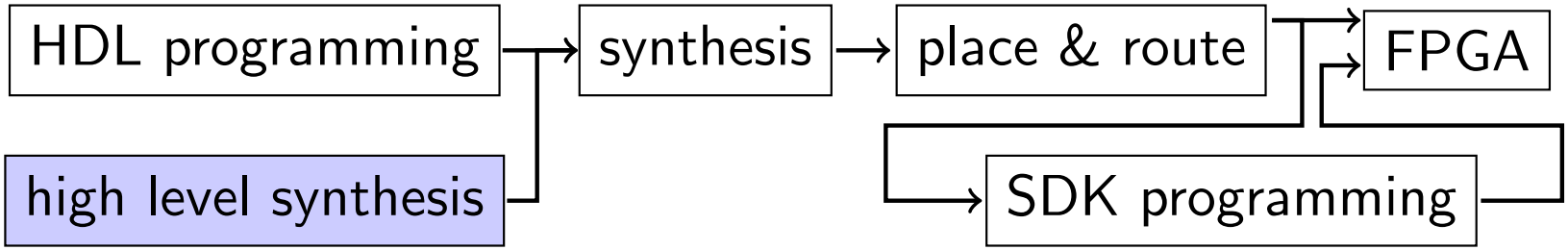
FPGA: Workflow



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

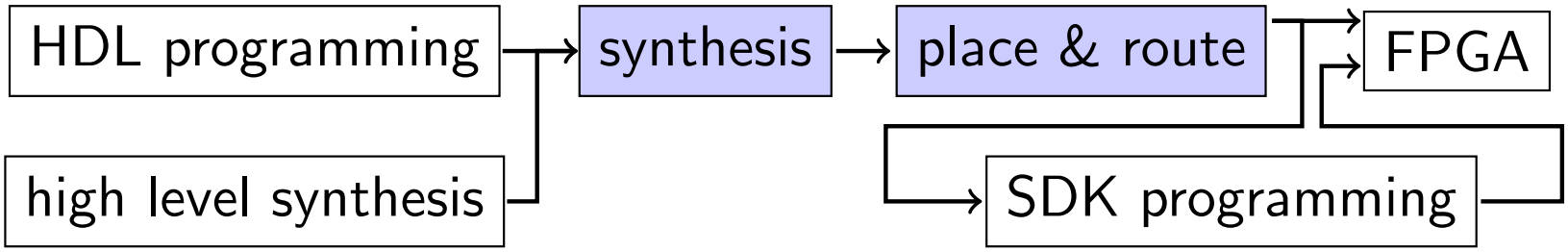


Basic idea: Automatically translate high level code into HDL

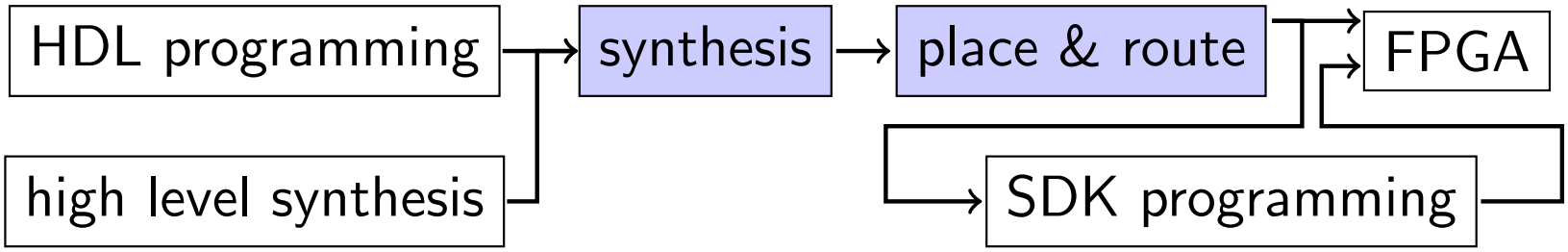
- Automate tedious work
- Compile code specifically for target device
- Lets you explore design space effectively
- Output should be reviewed
- Code must be changed for HLS tool
- Only works on subset of high level language

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



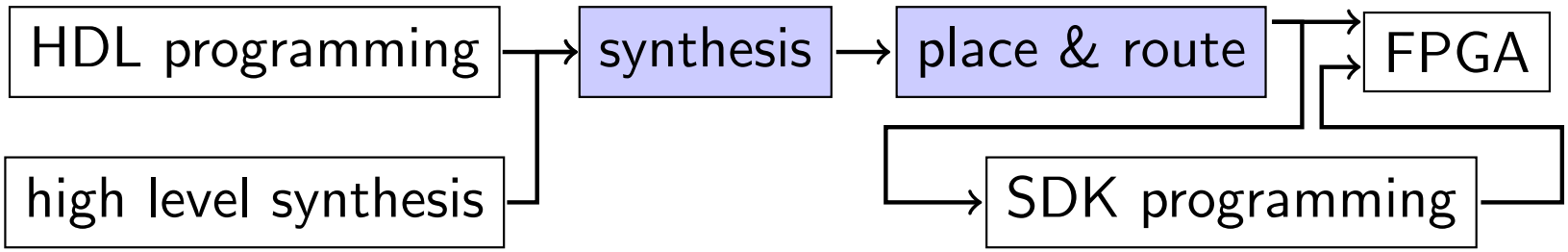
Synthesis: Calculate CL configurations

→ **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

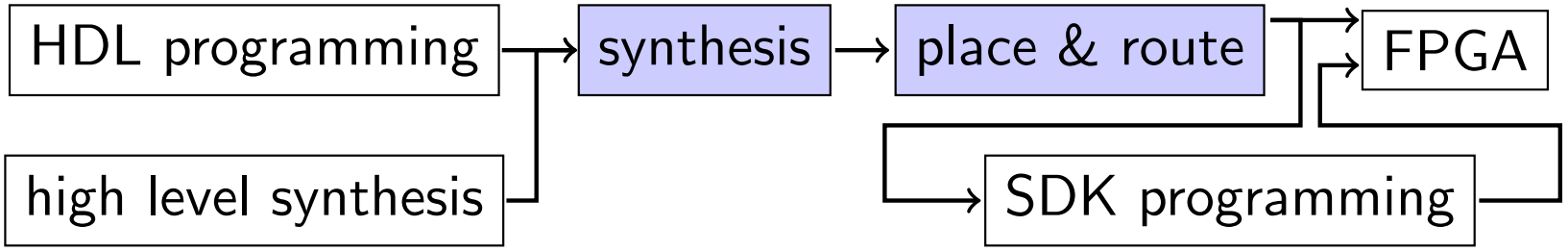
FPGA: Workflow



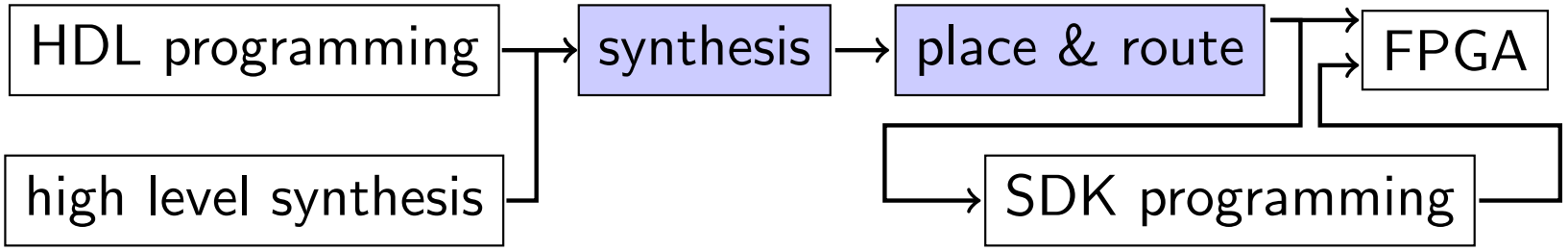
Synthesis: Calculate CL configurations
→ **So far:** HDL contains abstractions, e.g. summation
→ **Thus:** Compile these to a gate description, e.g. half/full-adder
⇒ The netlist contains the functionality of all units of the design

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

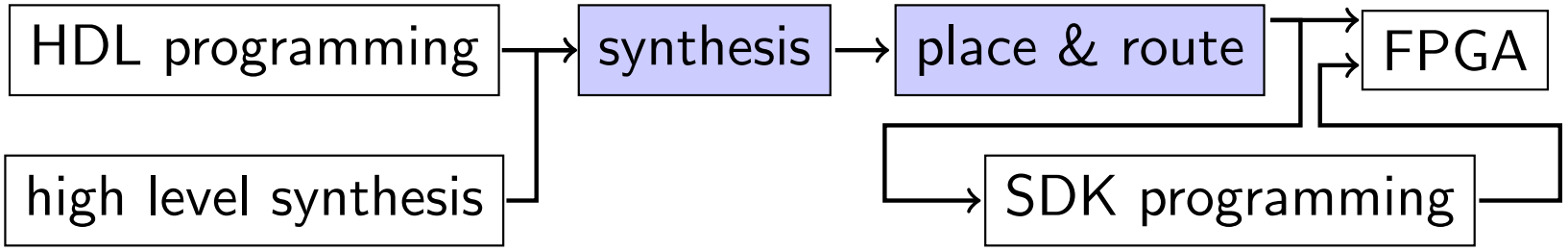


FPGA: Workflow



Important: Synthesis and place & route may fail!

FPGA: Workflow

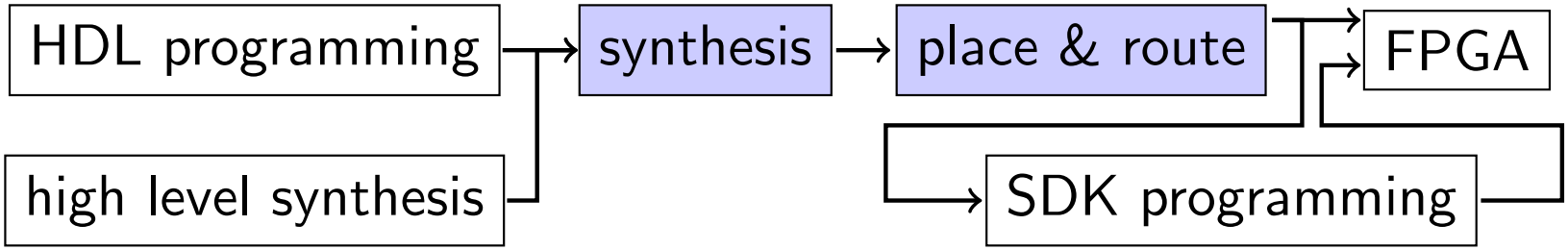


Important: Synthesis and place & route may fail!

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

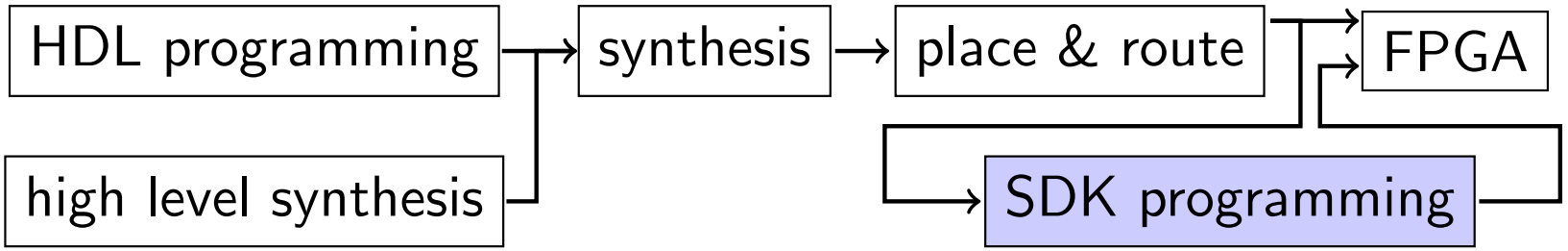


Important: Synthesis and place & route may fail!

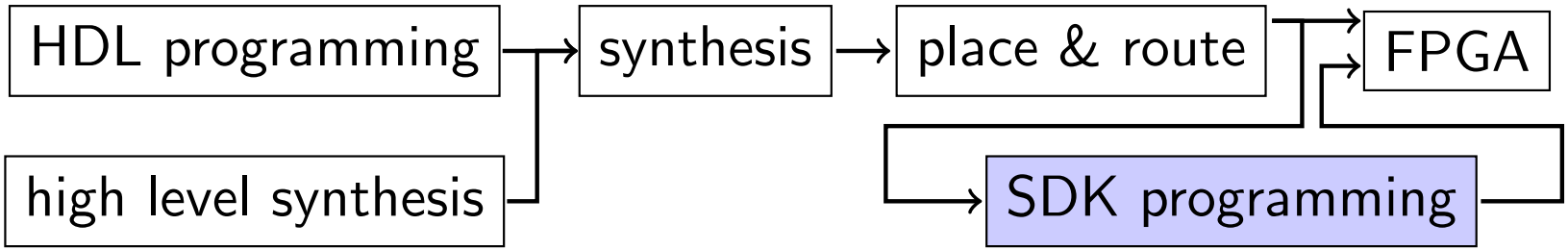
Observation 1: HDL and HLS allow us to express things, which are not existent in hardware, e.g. files
Observation 2: Hardware is usually clocked. Place & route may fail to provide the necessary timings to achieve the given clock.

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



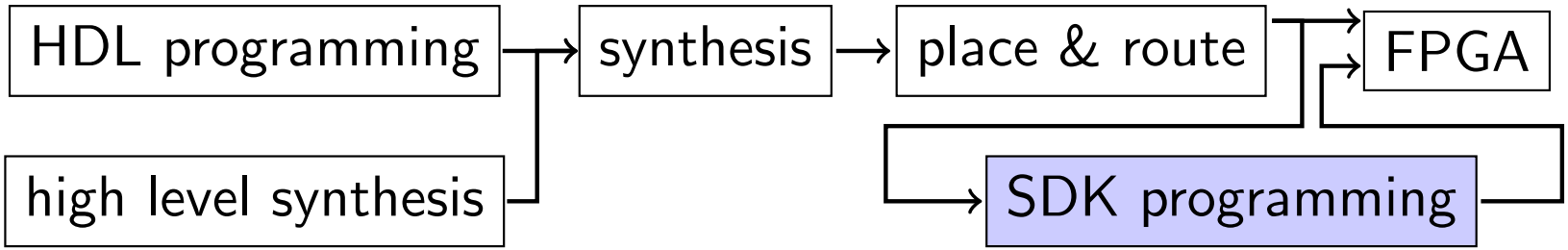
FPGA: Workflow



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, floating point operation etc.
- Different optimizations for energy or throughput available
- Usually programmed in C-like language with own compiler

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

How do we put Deep Learning on FPGAs?

Deep Learning: Some considerations

Why FPGAs for Deep Learning?
Fact: DeepLearning networks still have a lot of parameters
Additional: Many SGD steps are required to get reasonable results

Deep Learning: Some considerations

Why FPGAs for Deep Learning?
Fact: DeepLearning networks still have a lot of parameters
Additional: 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

Why FPGAs for Deep Learning?
Fact: DeepLearning networks still have a lot of parameters
Additional: 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.

$\Rightarrow$ Fast and energy efficient hardware and fast implementations required!

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)

Bottom-Line: Hardware-specific implementations play a great part in DeepLearning!

FPGAs as Co-Processors

Some facts about hardware:

- 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

Some facts about hardware:

- 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

Some facts about hardware:

- 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-thump: 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

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

Usually: CPUs also do not offer a runtime system
Thus: Run full blown Linux on CPU + develop software for CPU + specify hardware accelerator for FPGA
$\Rightarrow$ 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

Usually: CPUs also do not offer a runtime system
Thus: Run full blown Linux on CPU + develop software for CPU + specify hardware accelerator for FPGA
$\Rightarrow$ Easy software development for "glue" code + fast energy and efficient computations

Question: How do we control the FPGA hardware accelerator?

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

Thus: SDSoC compiles C/C++ code, generated HDL code from C/C++ and generated Linux kernel drivers

In the end: We get a bootable Linux image 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`

High Level Synthesis

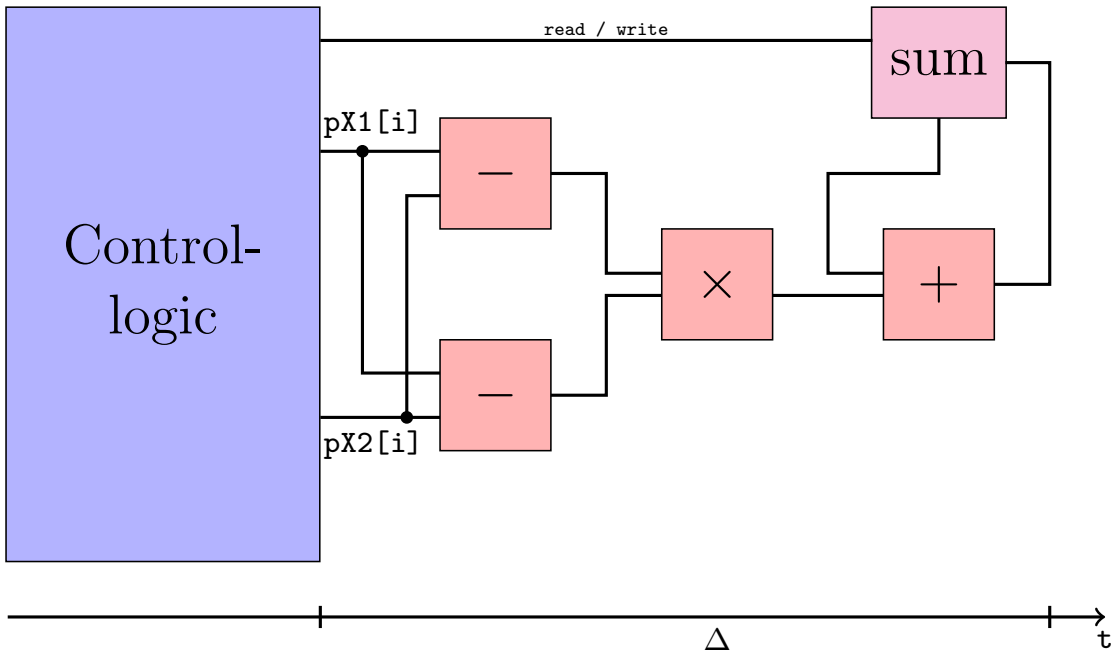
Question: How would we implement this function in hardware?

High Level Synthesis

Question: How would we implement this function in hardware?
Idea: Subtract $\rightarrow$ multiply $\rightarrow$ sum $\rightarrow$ update `sum`:

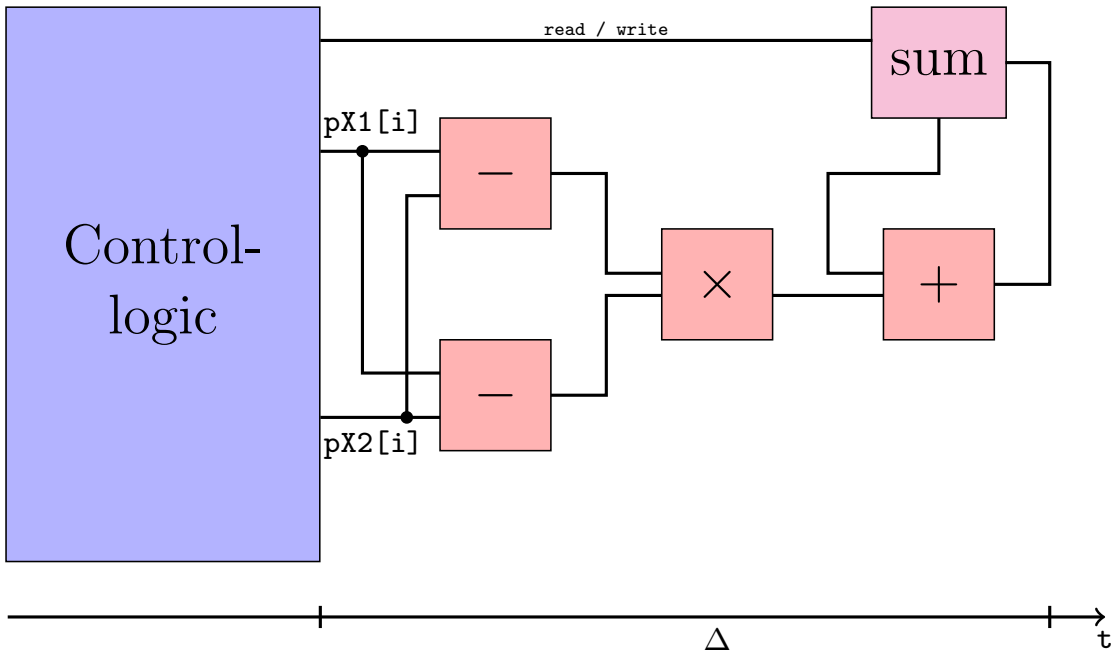
High Level Synthesis

Question: How would we implement this function in hardware?
Idea: Subtract → multiply → sum → update `sum`:



High Level Synthesis

Question: How would we implement this function in hardware?
Idea: Subtract → multiply → sum → update `sum`:



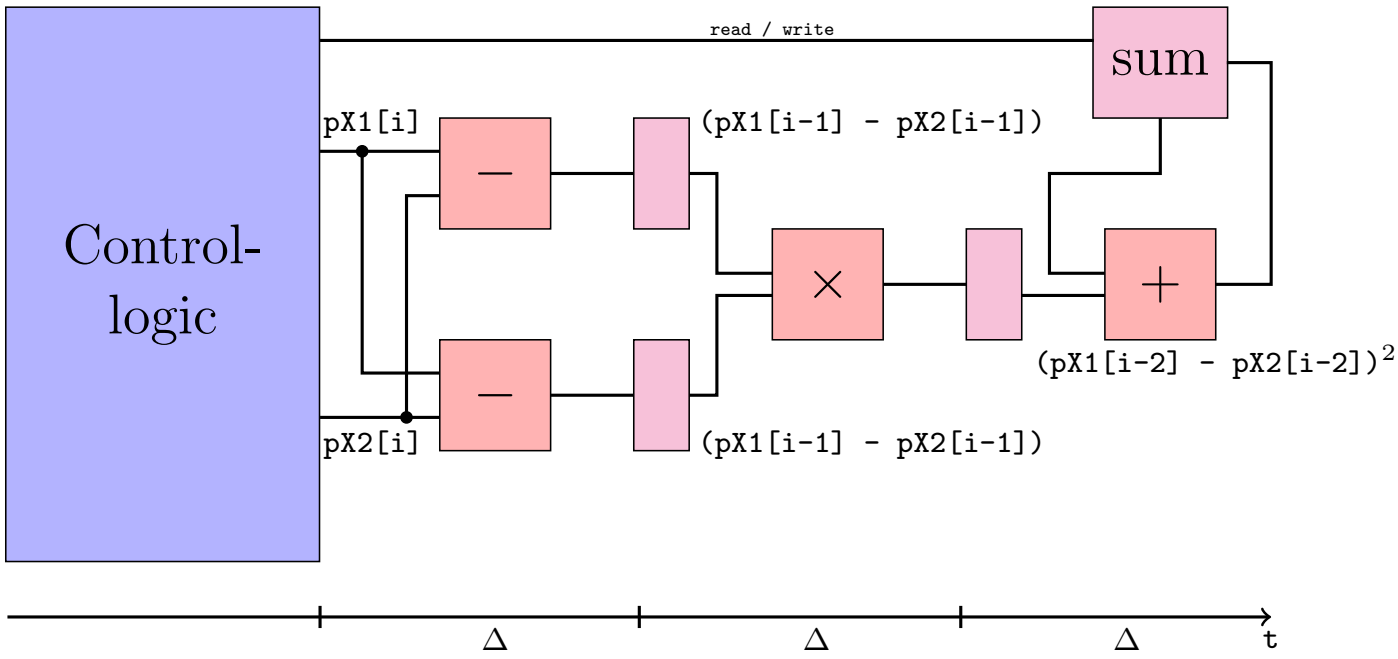
Analysis: Signal delay $\Delta = 4$, needs `dim` clocks
Pragma: This is the HLS default

High Level Synthesis: Pipelining

Observation: Only 1 functional unit active at a time. Pipeline execution to utilize every functional unit

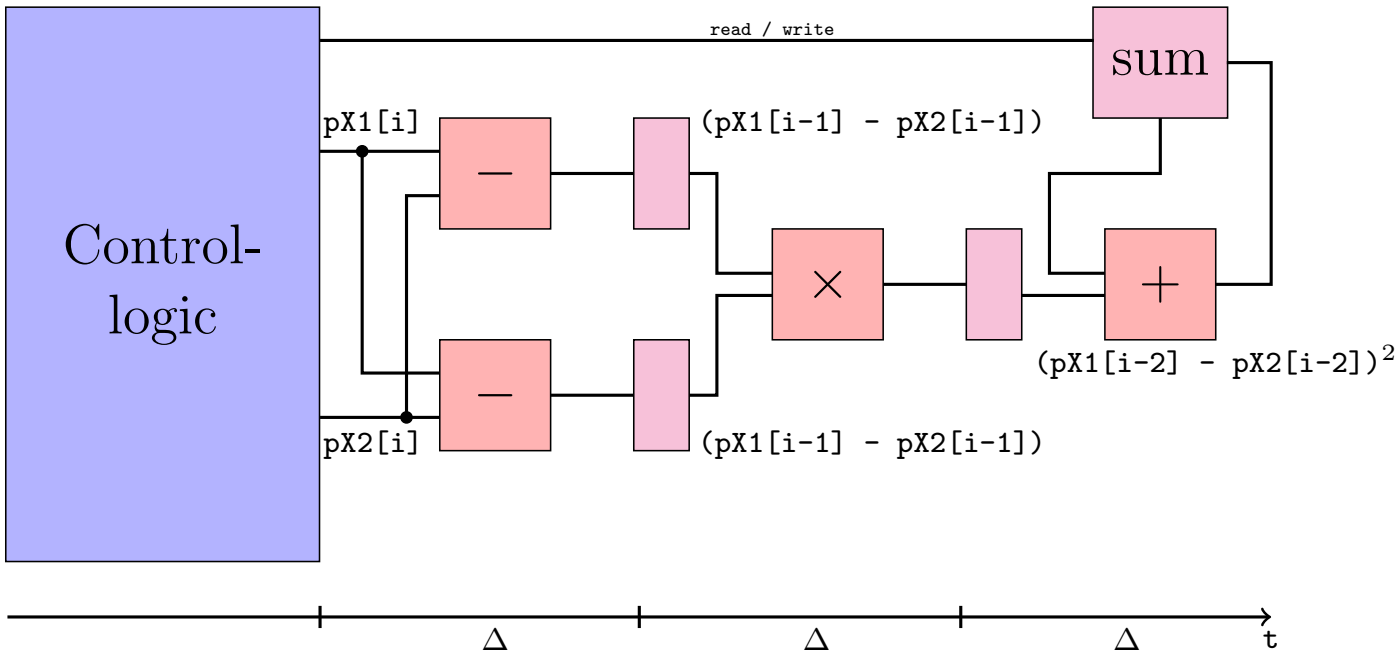
High Level Synthesis: Pipelining

Observation: Only 1 functional unit active at a time. Pipeline execution to utilize every functional unit



High Level Synthesis: Pipelining

Observation: Only 1 functional unit active at a time. Pipeline execution to utilize every functional unit



Analysis: Signal delay $\Delta = 1$, `dim+4` clocks needed

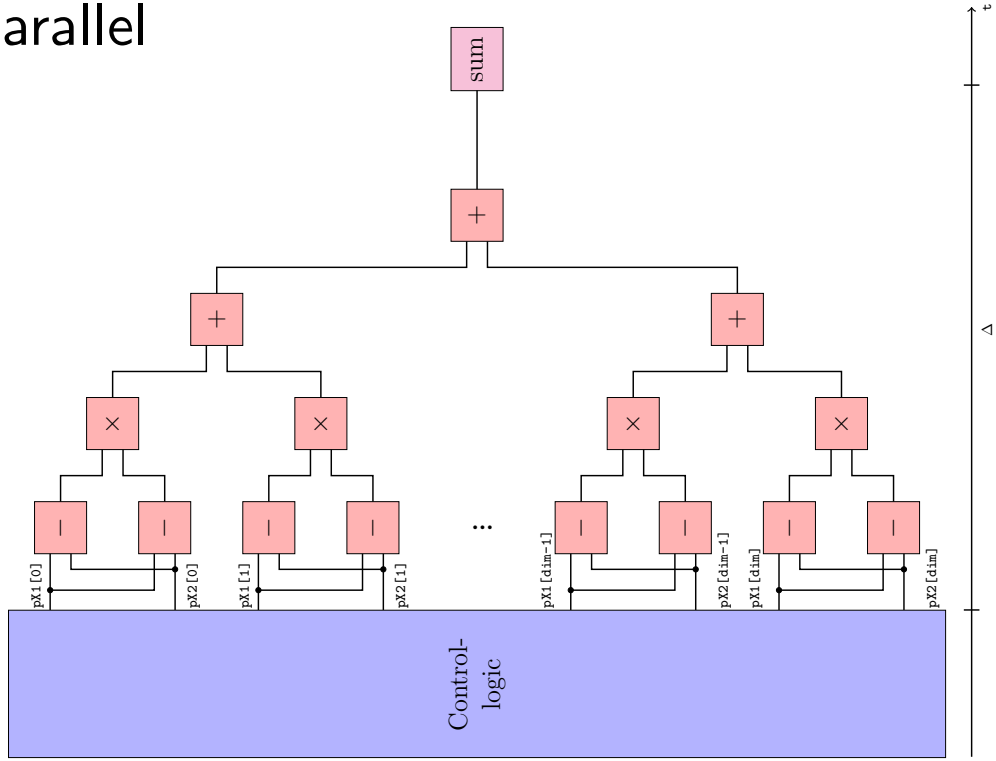
Pragma: `#pragma HLS PIPELINE`

High Level Synthesis: Loop unrolling

Observation: We can compute the subtraction and multiplication in complete parallel

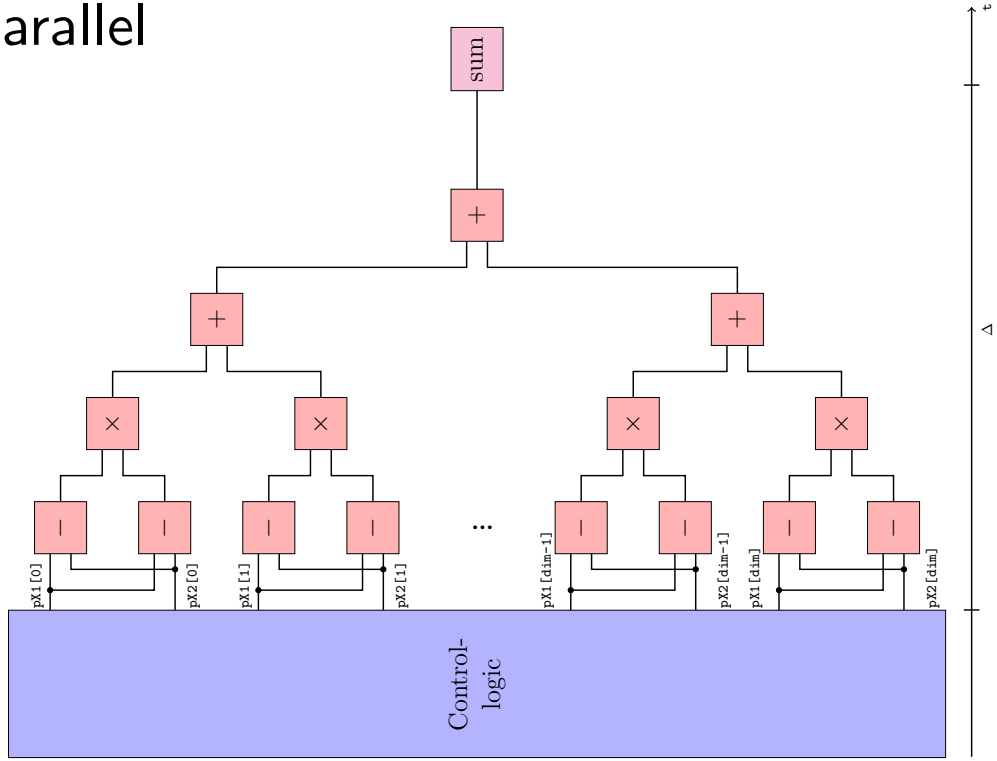
High Level Synthesis: Loop unrolling

Observation: We can compute the subtraction and multiplication in complete parallel



High Level Synthesis: Loop unrolling

Observation: We can compute the subtraction and multiplication in complete parallel



Analysis: Signal delay $\Delta = 4$, 2 clocks needed

Pragma: `#pragma HLS UNROLL`

HLS: Optimizations

Question: So what's best to use? Pipeline? Loop unrolling?

HLS: Optimizations

Question: So what's best to use? Pipeline? Loop unrolling?
Depends on problem, but usually:

- **Loop unrolling:** Needs a lot of space, but offers high parallelism. Clock frequency suffers from large structure.
- **Pipelining:** Good compromise between parallelism and small structure with high clock frequency.

HLS: Optimizations

Question: So what's best to use? Pipeline? Loop unrolling?
Depends on problem, but usually:

- **Loop unrolling:** Needs a lot of space, but offers high parallelism. Clock frequency suffers from large structure.
- **Pipelining:** Good compromise between parallelism and small structure with high clock frequency.

Note 1: Only "perfect" loops can be unrolled!
$\Rightarrow$ If a loop contains branches (`if-clause`), we cannot unroll it

HLS: Optimizations

Question: So what's best to use? Pipeline? Loop unrolling?
Depends on problem, but usually:

- **Loop unrolling:** Needs a lot of space, but offers high parallelism. Clock frequency suffers from large structure.
- **Pipelining:** Good compromise between parallelism and small structure with high clock frequency.

Note 1: Only "perfect" loops can be unrolled!
⇒ If a loop contains branches (`if-clause`), we cannot unroll it

Note 2: Sometimes even pipelining might fail
⇒ Nested loops need to be unrolled

HLS: Optimizations

Question: So what's best to use? Pipeline? Loop unrolling?
Depends on problem, but usually:

- **Loop unrolling:** Needs a lot of space, but offers high parallelism. Clock frequency suffers from large structure.
- **Pipelining:** Good compromise between parallelism and small structure with high clock frequency.

Note 1: Only "perfect" loops can be unrolled!
$\Rightarrow$ If a loop contains branches (`if-clause`), we cannot unroll it

Note 2: Sometimes even pipelining might fail
$\Rightarrow$ Nested loops need to be unrolled

A note on memory: HLS will try to use Block-RAM when possible ($\rightarrow$ use the `static` keyword whenever possible). Otherwise it uses CLB for memory. It will never access `DDR RAM`

FPGA: Custom data ranges

Fact 1: FPGAs offer arbitrary data ranges and data types
Rule-of-thump: The less bits, the better for synthesis

FPGA: Custom data ranges

Fact 1: FPGAs offer arbitrary data ranges and data types
Rule-of-thump: The less bits, the better for synthesis

Fact: For computing, integer operations are the easiest and fastest
But: Sometimes floating point is needed
Fact 2: IEEE-754 floating point operations are slow compared to integer operations (Mantissa needs normalization)

FPGA: Custom data ranges

Fact 1: FPGAs offer arbitrary data ranges and data types
Rule-of-thump: The less bits, the better for synthesis

Fact: For computing, integer operations are the easiest and fastest
But: Sometimes floating point is needed
Fact 2: IEEE-754 floating point operations are slow compared to integer operations (Mantissa needs normalization)

Thus: Use a Fixed point number $N = A.B$ with fixed sized integers A and B
$\Rightarrow$ Fast floating point operations with reduced accuracy possible

Tip: Start to think in terms of bits with integer operations

Deep Learning on FPGAs

Question: So how do we implement Deep Learning on FPGAs?

Deep Learning on FPGAs

Question: So how do we implement Deep Learning on FPGAs?
Some ideas from a hardware perspective:

- **Reduce communication:** Reduce communication between ARM and FPGA to a minimum. Usually, this is your bottleneck.
- **Use on-chip memory:** If your neural networks are small enough, store weights in on-chip memory.
- **Use compile-time constants:** The more the compiler knows at compile time, the better. Use fixed values and upper bounds for loops as often as possible.
- **Use parallelism:** Unroll small structures / parts of your code, e.g. multiplying weights with input values

Deep Learning on FPGAs (2)

Question: So how do we implement Deep Learning on FPGAs?

Deep Learning on FPGAs (2)

Question: So how do we implement Deep Learning on FPGAs?
Some ideas from a the ML perspective:

- **Reduce data range:** Use fixed point whenever possible: Fixed floating point can be implemented efficiently with integer operations. Maybe even reduce the data range.
- **Perform batch SGD:** Load a batch of data points on FPGA and perform gradient with on-chip-memory batches.
- **Change activation function:** Do we really need sigmoid in multiple layers? Maybe one sigmoid layer is enough?
- **Unsynchronized dropout:** Dropout is computed by each neuron individually.

Deep Learning on FPGAs (2)

Question: So how do we implement Deep Learning on FPGAs?
Some ideas from a the ML perspective:

- **Reduce data range:** Use fixed point whenever possible: Fixed floating point can be implemented efficiently with integer operations. Maybe even reduce the data range.
- **Perform batch SGD:** Load a batch of data points on FPGA and perform gradient with on-chip-memory batches.
- **Change activation function:** Do we really need sigmoid in multiple layers? Maybe one sigmoid layer is enough?
- **Unsynchronized dropout:** Dropout is computed by each neuron individually.

Note: Changes must be evaluated with respect to accuracy!

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