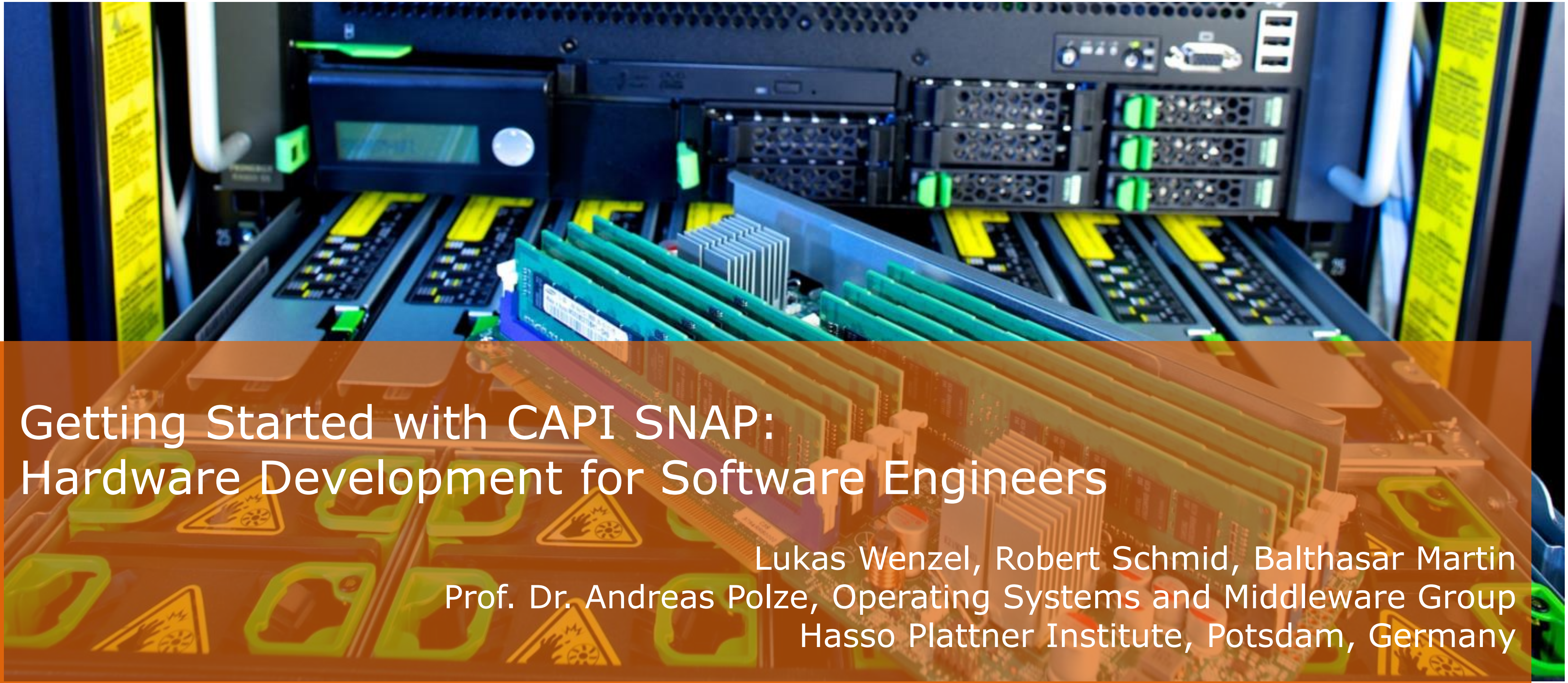


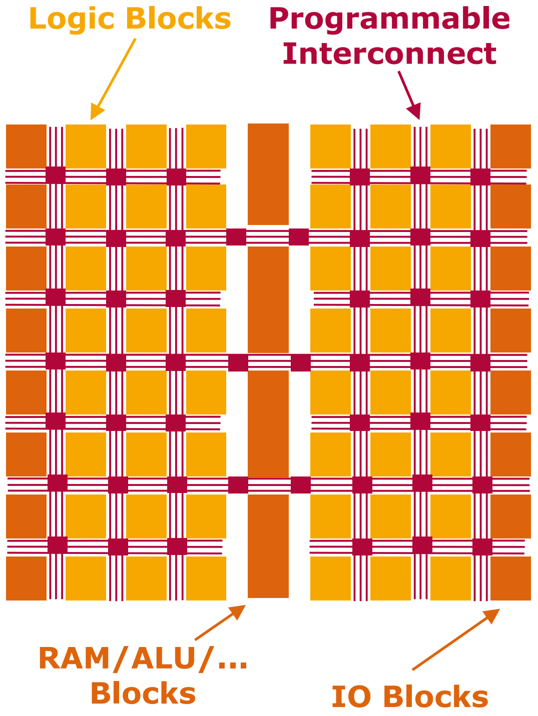
Getting Started with CAPI SNAP: Hardware Development for Software Engineers

Lukas Wenzel, Robert Schmid, Balthasar Martin
Prof. Dr. Andreas Polze, Operating Systems and Middleware Group
Hasso Plattner Institute, Potsdam, Germany

As general purpose computing power stagnates, FPGA acceleration can speed up parallel tasks

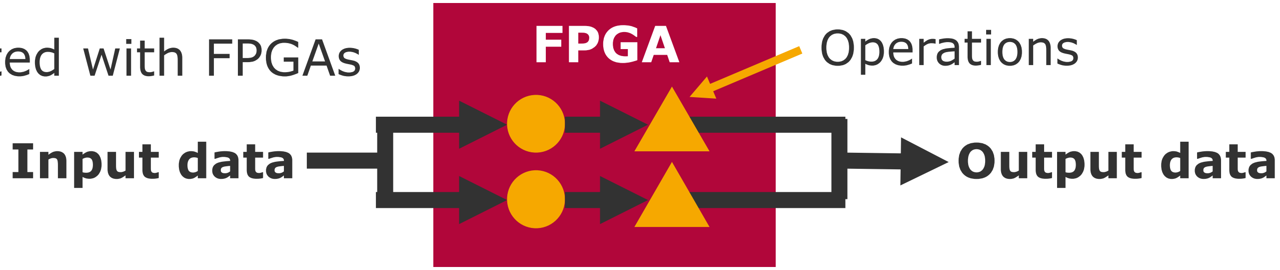
- The exponential development of computing power described in Moore's law starts to stagnate → Fuels development of hardware accelerators
- Many tasks may benefit from specialized hardware, but custom chip manufacturing needs high numbers to be profitable

- **F**ield-**P**rogrammable **G**ate **A**rray: programmable hardware circuit
- Had a peak in interest more than a decade ago and did not live up to the expectations
  - Hard to integrate in software solutions
  - Different development paradigm with high learning effort



The ecosystem and amount of use cases for accelerators changed

- Big data tasks may be efficiently accelerated with FPGAs
  - Parallelizable and pipelined



- FPGAs can now communicate directly with other resources which improves integration
  - E.g. NVMe for direct and parallel access to storage

Google Trends “NVM Express“ 24/08/2018

- Cloud environments give access to hardware accelerators without the need to buy them
- New interfaces and frameworks try to lower the entry barrier to hardware acceleration

We wanted to find out how high the entry barrier is for software engineers

Of course, there is still some learning curve left

How difficult is learning FPGA accelerator design for software engineering students?

3 Students

1 Semester

9 ECTS

Some overhead

The solution we used in our project was was CAPI SNAP with a Nallatech 250S FPGA card

- Open source interfaces and frameworks developed by the Open Power Foundation
- Intended to make use of accelerators on the IBM POWER platform

CAPI

connects host applications with hardware accelerators

- Organizes data exchange between host, FPGA and memory
- Communication interfaces for memory access, job control, …
- Needs to be implemented on FPGA side (PSL) and host side (libcxl)

Builds on

SNAP

introduce abstraction and increase usablitiy

- Build process integrating all tools and supporting different FPGAs
- Integration with C-like Vivado HLS language for hardware design
- Simulation without hardware and debugging

Learning about FPGA acceleration on Power is now easier than before

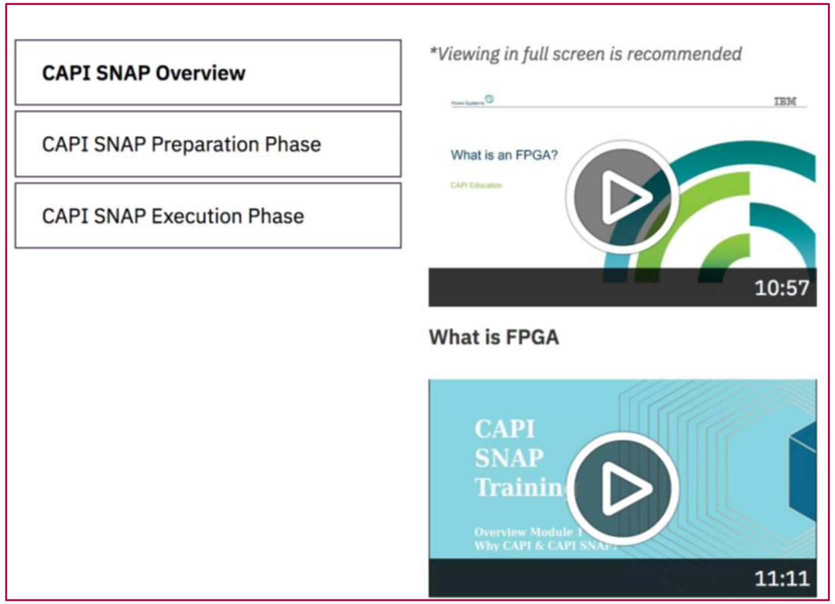
Prerequisites

- Power8 Machine and FPGA Card are optional but recommended
- Xilinx Vivado is required

➔ IBM plans to offer cloud-based development environments providing Vivado and FPGAs

Documentation

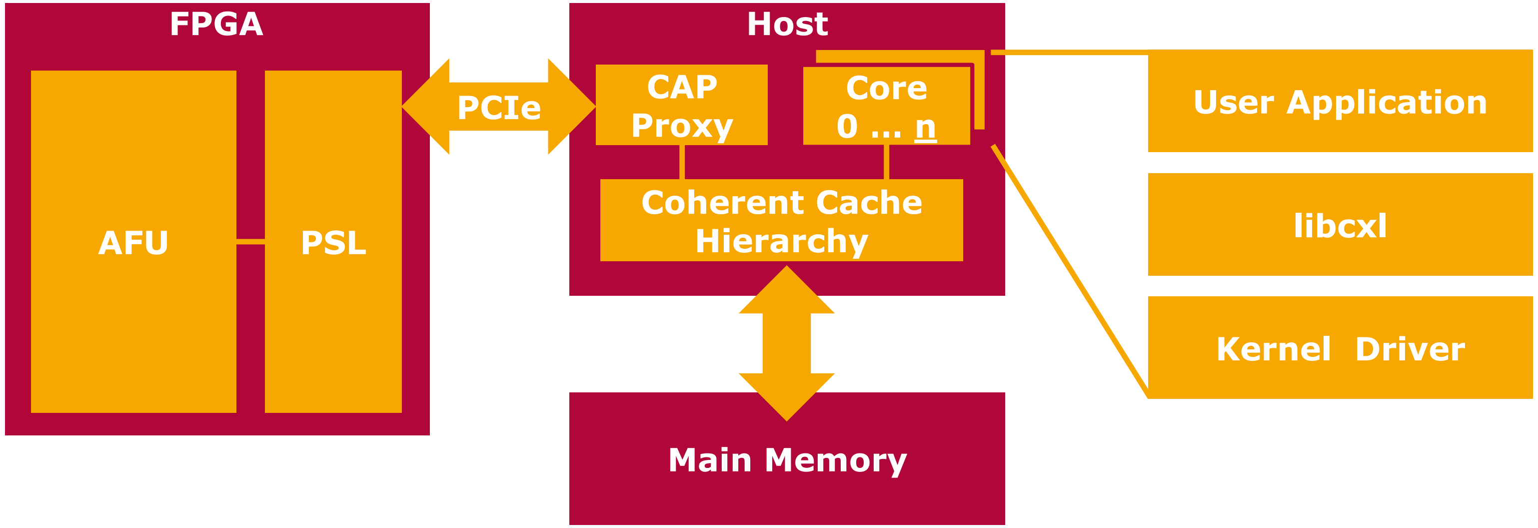
- SNAP Repository with documentation:
  - https://github.com/open-power/snap
- Video introductions and tutorials:
  - https://developer.ibm.com/linuxonpower/capi/education/
- Longer version of our paper as an introduction:
  - https://www.dcl.hpi.uni-potsdam.de/capi-snap



IBM CAPI is an accelerator interface for the communication between host and FPGA

- Acccelerator can coherenctly access host memory

→ No redudant copies or memory access overhead



- Accelerated Function Units (AFUs) are outsourced functionalities
- Communication to FPGA via libcxl and shared memory

What would be desirable from a software developer's perspective

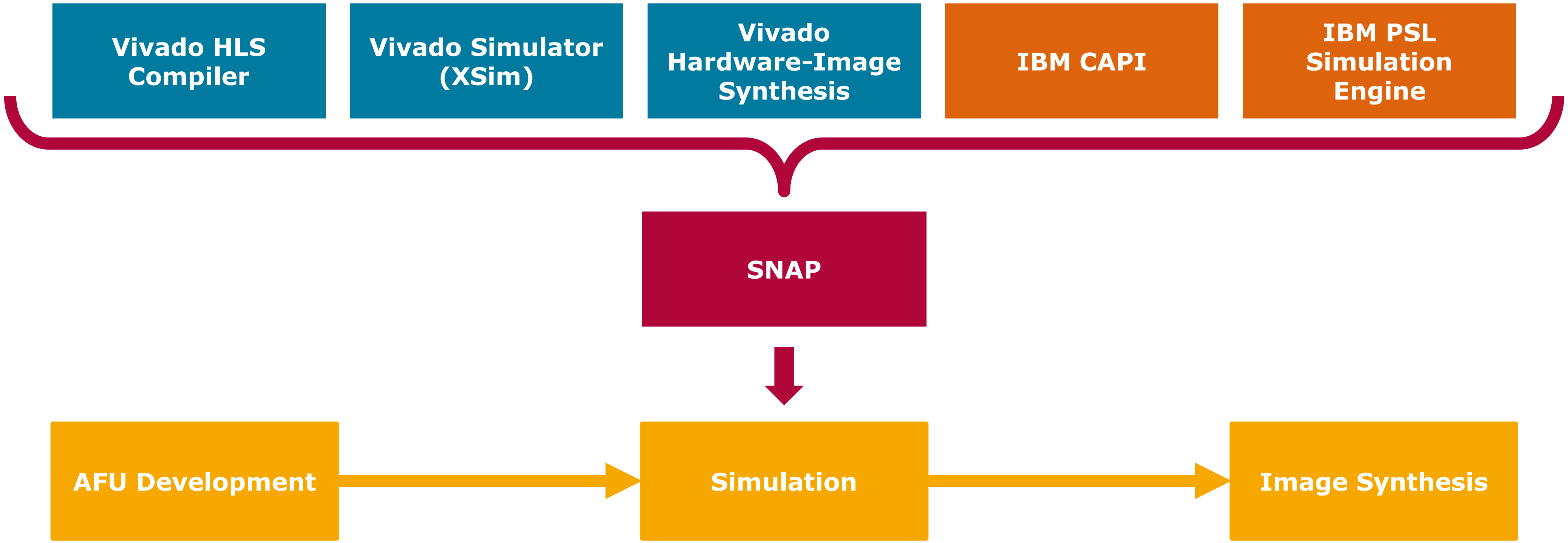
- Programming in a high-level language
  - OpenCL, SDAccel
  - Intel HLS, Vivado HLS

- Portability across different FPGAs

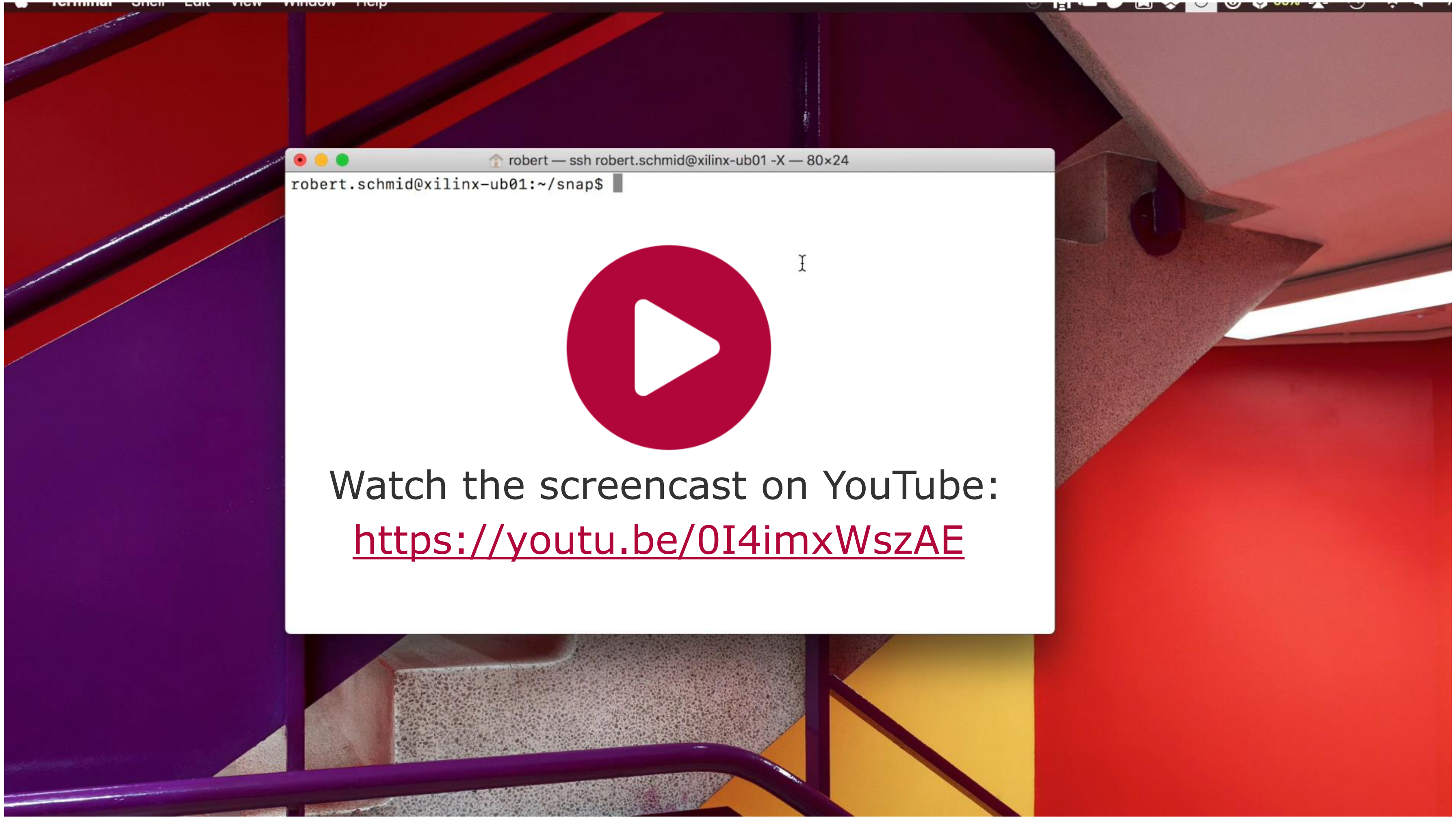
- Access to additional on-card hardware

- Unified workflow covering all development steps
  - Creating the hardware specification
  - Simulation
  - Generating the bitstream

SNAP provides a unified build process and programming model

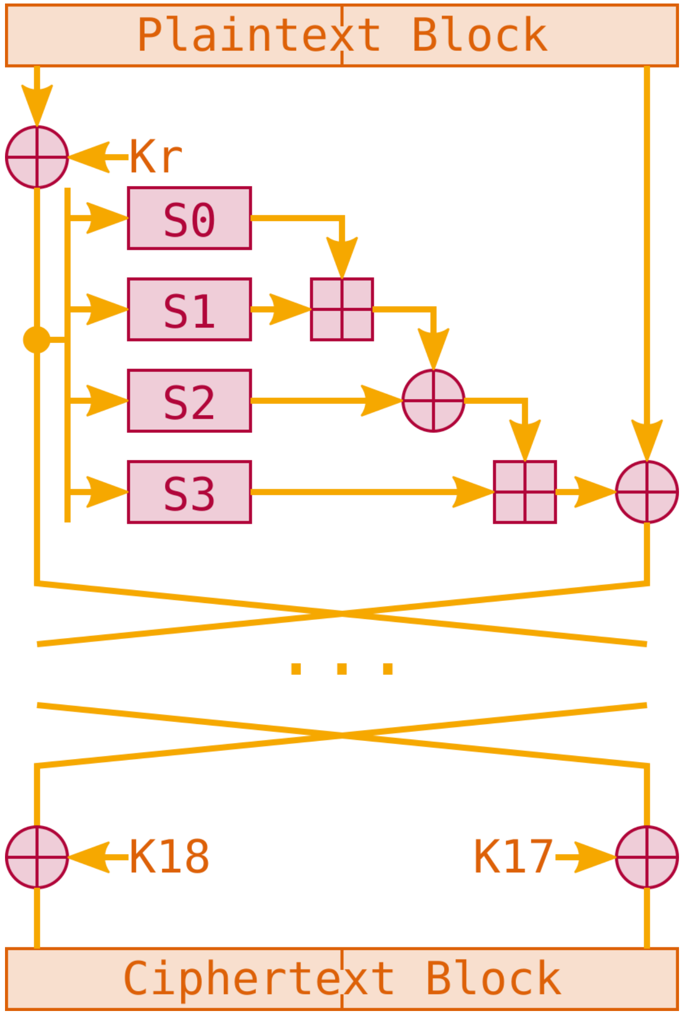


Screencast of the SNAP Action Development workflow



Example
We implemented the block cipher Blowfish in HLS

- Blowfish: symmetric block cipher
  - 64 bit blocks
  - 32 to 448 bit keys
- Free, easy to implement, relatively fast



Example
We implemented the block cipher Blowfish in HLS

- Blowfish-AFU operations:
  - SET_KEY: use byte_count bytes from input buffer to initialize the key for subsequent en-/decrypt operations
  - ENCRYPT: encrypt byte count plaintext bytes in input buffer and store the result in output buffer
  - DECRYPT: decrypt byte count ciphertext bytes in input buffer and store the result in output buffer

```
#define MODE_SET_KEY 0
#define MODE_ENCRYPT 1
#define MODE_DECRYPT 2

#ifndef CACHELINE_BYTES
#define CACHELINE_BYTES 128
#endif

// Blowfish Configuration PATTERN.
// This must match with DATA struc
// Job description should start wi
typedef struct blowfish_job {
    struct snap_addr input_data;
    struct snap_addr output_data;
    uint32_t mode;
    uint32_t data_length;
} blowfish_job_t;
```

Blowfish Job Structure

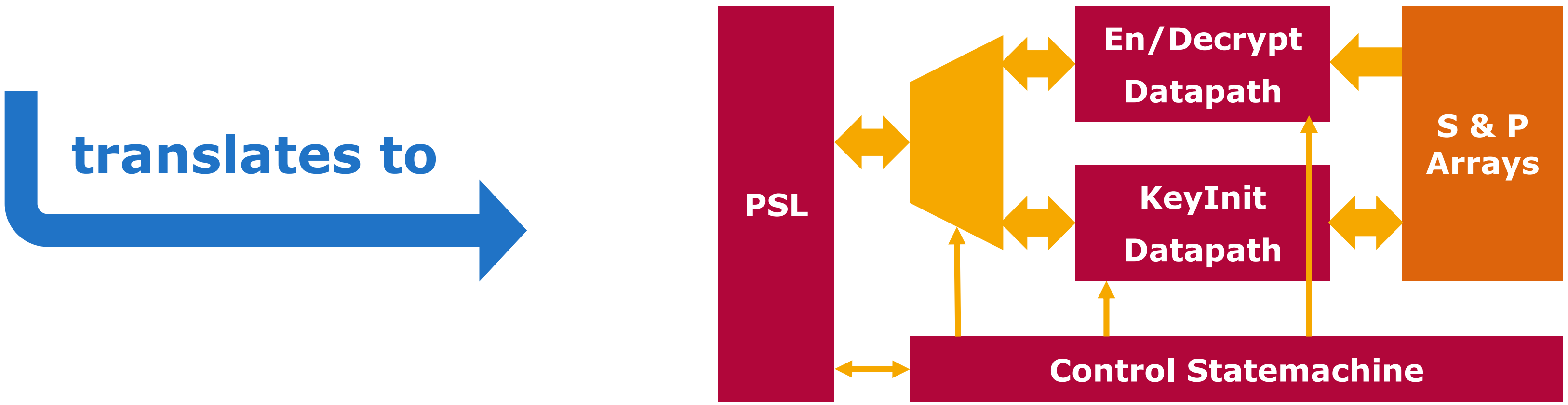
```
void encrypt(void * key, void * plaintext, void * ciphertext) {
    struct snap_job job;
    prepare_blowfish_job(&job, MODE_SET_KEY, strlen(key), key, NULL);
    snap_action_sync_execute_job(action, &job, 60);
    prepare_blowfish_job(&job, MODE_ENCRYPT, strlen(plaintext), plaintext, ciphertext);
    snap_action_sync_execute_job(action, &job, 60);
}
```

Blowfish Job Invocation

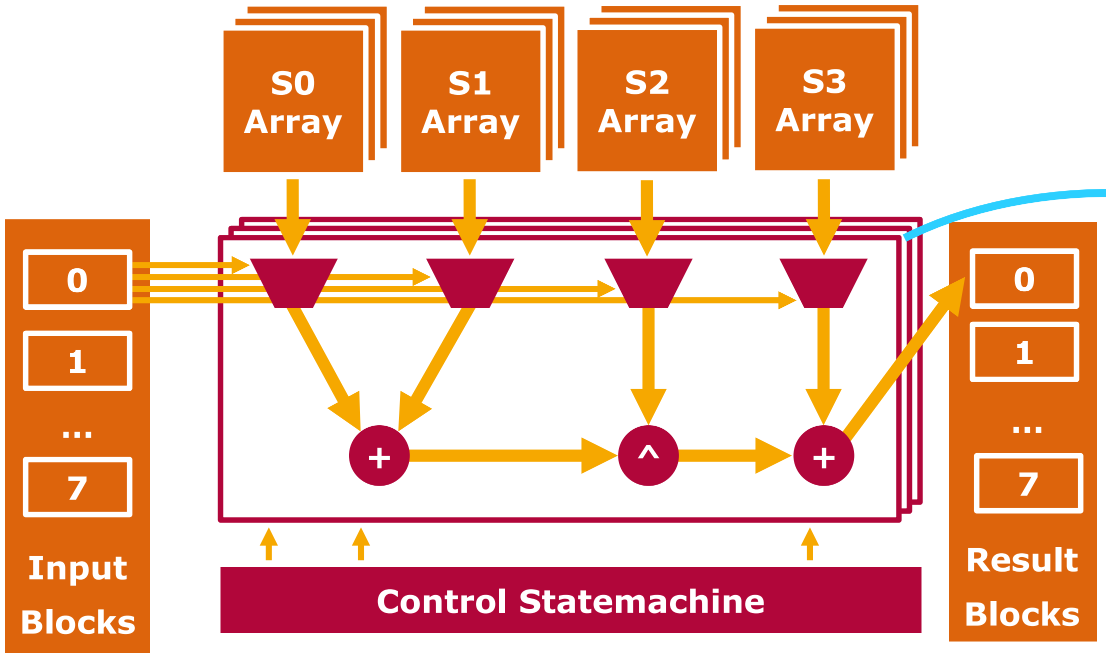
Example
We implemented the block cipher Blowfish in HLS

Hasso Plattner Institut

```
static bf_P_t g_P;
static bf_S_t g_S;
static snapu32_t process_action(snap_membus_t * din_gmem, snap_membus_t * dout_gmem, action_reg * action_reg)
{
    snapu64_t inAddr, outAddr;
    snapu32_t byteCount, mode, retc;
    // initialize arguments from action_reg ...
    switch (mode) {
    case MODE_SET_KEY: retc = action_setkey(din_gmem, inAddr, byteCount); break;
    case MODE_ENCRYPT: retc = action_endecrypt(din_gmem, inAddr, dout_gmem, outAddr, byteCount, 0); break;
    case MODE_DECRYPT: retc = action_endecrypt(din_gmem, inAddr, dout_gmem, outAddr, byteCount, 1);
    }
    return retc;
}
```



Example
We implemented the block cipher Blowfish in HLS




```
#pragma HLS ARRAY_PARTITION variable=g_S complete dim=1

static void bf_fLine(bf_halfBlock_t res[BF_BPL], bf_halfBlock_t h[BF_BPL])
{
    for (bf_uiBpl_t iBlock = 0; iBlock < BF_BPL; ++iBlock)
    {
#pragma HLS UNROLL factor=8 //==BF_BPL
        bf_SiE_t a = (bf_SiE_t)(h[iBlock] >> 24),
                 b = (bf_SiE_t)(h[iBlock] >> 16),
                 c = (bf_SiE_t)(h[iBlock] >> 8),
                 d = (bf_SiE_t) h[iBlock];
        res[iBlock] = ((g_S[iBlock/2][0][a] + g_S[iBlock/2][1][b]) ^
                        g_S[iBlock/2][2][c]) + g_S[iBlock/2][3][d];
    }
}
```

Example
We implemented the block cipher Blowfish in HLS

Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
hls_action	78	0	20868	47824		undef	none
process_action	48	0	16887	41814		undef	none
action_endecrypt	30	0	12281	17069	1~2251	1 ~ 22	none
bf_encryptLine	0	0	3993	7071	129	129	none
bf_fLine	0	0	1318	1841	5	5	none
bf_decryptLine	0	0	3992	7074	121	121	none
bf_splitLine	0	0	0	0	0	0	none
bf_joinLine	0	0	0	0	0	0	none
action_setkey	2	0	4008	24399		undef	none

Resource(blowfish)

Current Module : hls_action > process_action > action_endecrypt >

	Resource\Control Step	C0	C1	C2	C3	C4	C5
1-49	I/O Ports						
50	Memory Ports						
51	g_S_V_6(p0)	read	read	read	read		
52	g_S_V_4(p1)	read	read	read	read		
53	g_S_V_1(p0)	read	read	read	read		
54	g_S_V_3(p0)	read	read	read	read		
55	g_S_V_2(p1)	read	read	read	read		
56	g_S_V_3(p1)	read	read	read	read		
57	g_S_V_7(p1)	read	read	read	read		
58	g_S_V_1(p1)	read	read	read	read		
59	g_S_V_5(p1)	read	read	read	read		
60	g_S_V_2(p0)	read	read	read	read		
61	g_S_V_4(p0)	read	read	read	read		
62	g_S_V_6(p1)	read	read	read	read		
63	g_S_V_7(p0)	read	read	read	read		
64	g_S_V_5(p0)	read	read	read	read		
65	g_S_V_0(p1)	read	read	read	read		
66	g_S_V_0(p0)	read	read	read	read		
67-...	Expressions						

Performance Resource

```
#pragma HLS ARRAY_PARTITION variable=g_S complete dim=1

static void bf_fLine(bf_halfBlock_t res[BF_BPL], bf_halfBlock_t h[BF_BPL])
{
    for (bf_uiBpL_t iBlock = 0; iBlock < BF_BPL; ++iBlock)
    {
#pragma HLS UNROLL factor=8 //==BF_BPL
        bf_SiE_t a = (bf_SiE_t)(h[iBlock] >> 24),
                 b = (bf_SiE_t)(h[iBlock] >> 16),
                 c = (bf_SiE_t)(h[iBlock] >> 8),
                 d = (bf_SiE_t) h[iBlock];
        res[iBlock] = ((g_S[iBlock/2][0][a] + g_S[iBlock/2][1][b]) ^
                       g_S[iBlock/2][2][c]) + g_S[iBlock/2][3][d];
    }
}
```

Conclusion
SNAP Makes Hardware Development Accessible

- Goal: Developing Hardware Accelerators
  - HLS' high level of abstraction is beneficial for beginners, few differences to traditional C/C++
  - SNAP is well documented, contains many examples
- **Within one semester**, students can **implement** common algorithms **and evaluate accelerator solutions** on real world systems (if available)

- Goal: Understanding Hardware Development
  - HLS hides underlying hardware implementation -> Using SNAP with a HDL like Verilog or VHDL makes details more accessible
  - Simulation reveals detailed behavior of HLS or HDL implementations
- **HLS abstracts from hardware details**, but hardware understanding is still beneficial for finer performance optimization

- Questions?

- Further information
  - IBM CAPI SNAP: https://github.com/open-power/snap
  - Our User Guide: https://www.dcl.hpi.uni-potsdam.de/capi-snap

- Lukas Wenzel, Robert Schmid, Balthasar Martin
{firstname}.{lastname}@student.hpi.uni-potsdam.de

- Thank you for your attention!