Hardware Acceleration on IBM Power
First Steps with CAPI SNAP

Balthasar Martin, Robert Schmid, Lukas Wenzel

Heterogeneous Computing Master Project

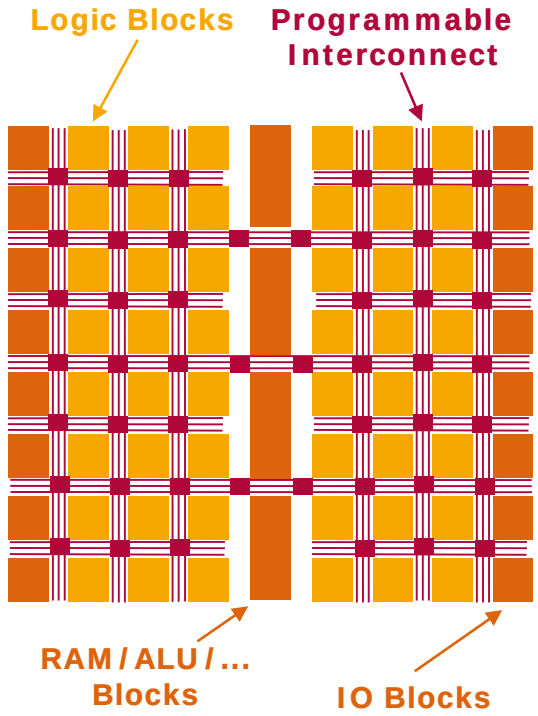
27 September, 2017

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

- Big data operations bring general purpose computers to their limits
  - Handling data streams is often parallelizable and could be pipelined
  - Simulating hardware structures (e.g. for machine learning) is inefficient

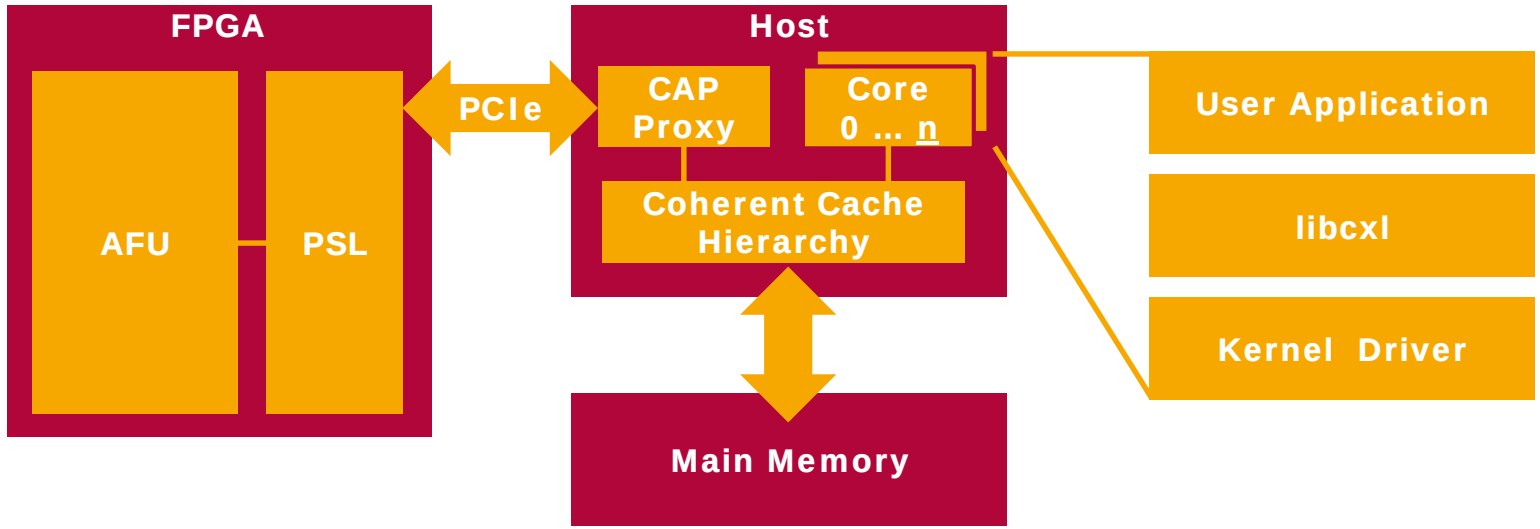
- Would benefit from specialized hardware
- Custom chip manufacturing needs high numbers to be profitable
- **F**ield **P**rogrammable **G**ate **A**rray: programmable hardware circuit

► Outsource computation-intense operations to a FPGA



IBM CAPI is a 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

Having to write hardware code is a barrier for software engineers

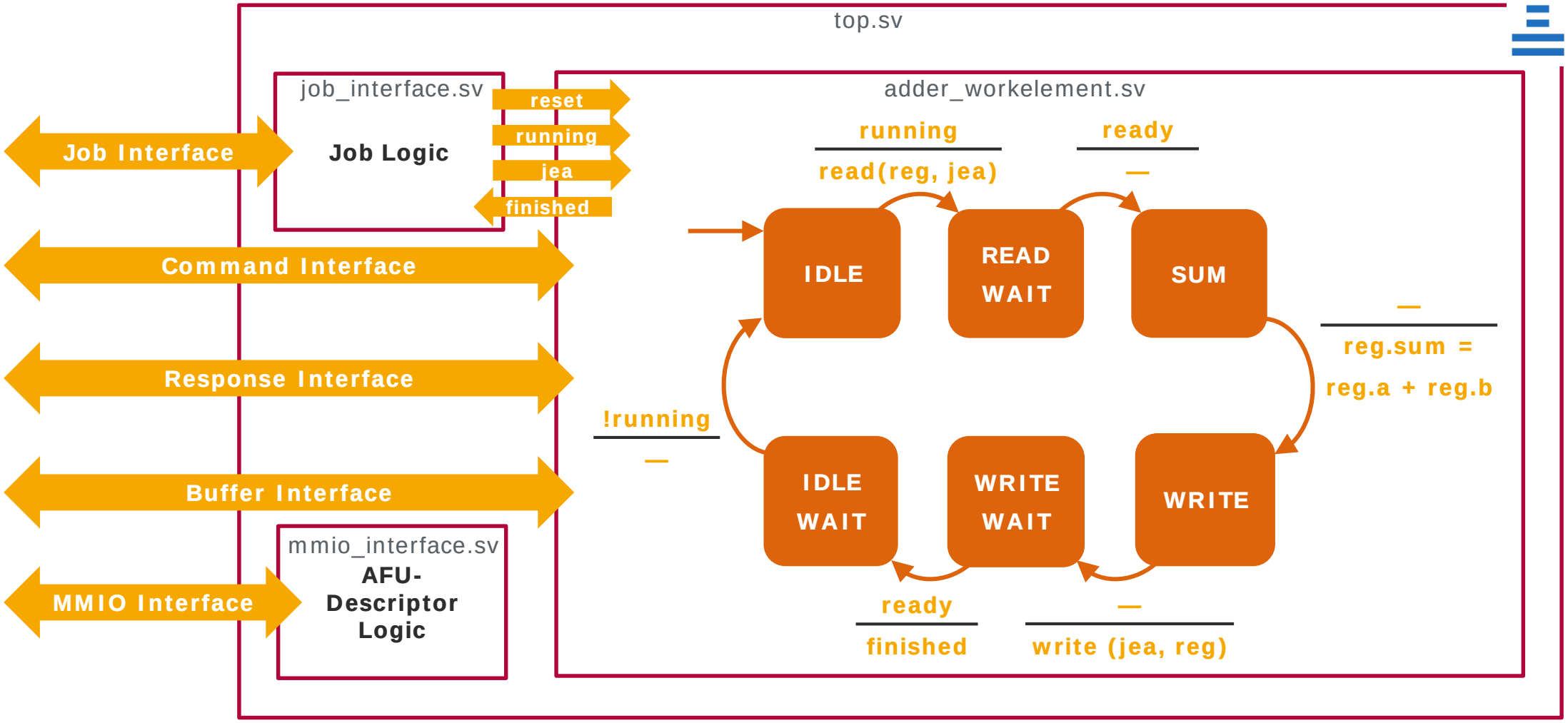
- Creating hardware specifications in Verilog or VHDL is quite different from imperative programming
  - VHDL and Verilog languages
  - Development workflow: Make design changes, synthesize, simulate, generate bitstream, test on device
  - Blocks as units of functional composition
- Detailed knowledge of the underlying hardware architecture is required
  - Communication has to be controlled manually
  - Timing constraints
  - Manage asynchronous command execution
- Each application needs to establish a communication protocol between host and AFU

Implementing a simple Adder-AFU for CAPI with SystemVerilog

- We built on the example presented in Kenneth Wilke's Blog (http://suchprogramming.com)
- Steps
  1. Define job structure for AFU in consumer code
  2. Initialize project with a root module (top.v) and CAPI interface declarations (capi.sv, afu.sv)
  3. Implement modules to encapsulate MMIO communication and Job lifecycle
  4. Implement work element as a state machine (Idle, Read, Sum, Write, WriteWait, IdleWait)
  5. Implement AFU consumer using libcxl

```
 5  typedef struct
 6  {
 7      uint32_t a;
 8      uint32_t b;
 9      uint32_t sum;
10      uint32_t done;
11  } sum_request;
12
13  sum_request *create_sum_req
14  {
15      sum_request *new = alig
16
17      new->done = 0;
18      new->a = 2;
19      new->b = 3;
20
21      return new;
22  }
```

Implementing a simple Adder-AFU for CAPI with SystemVerilog



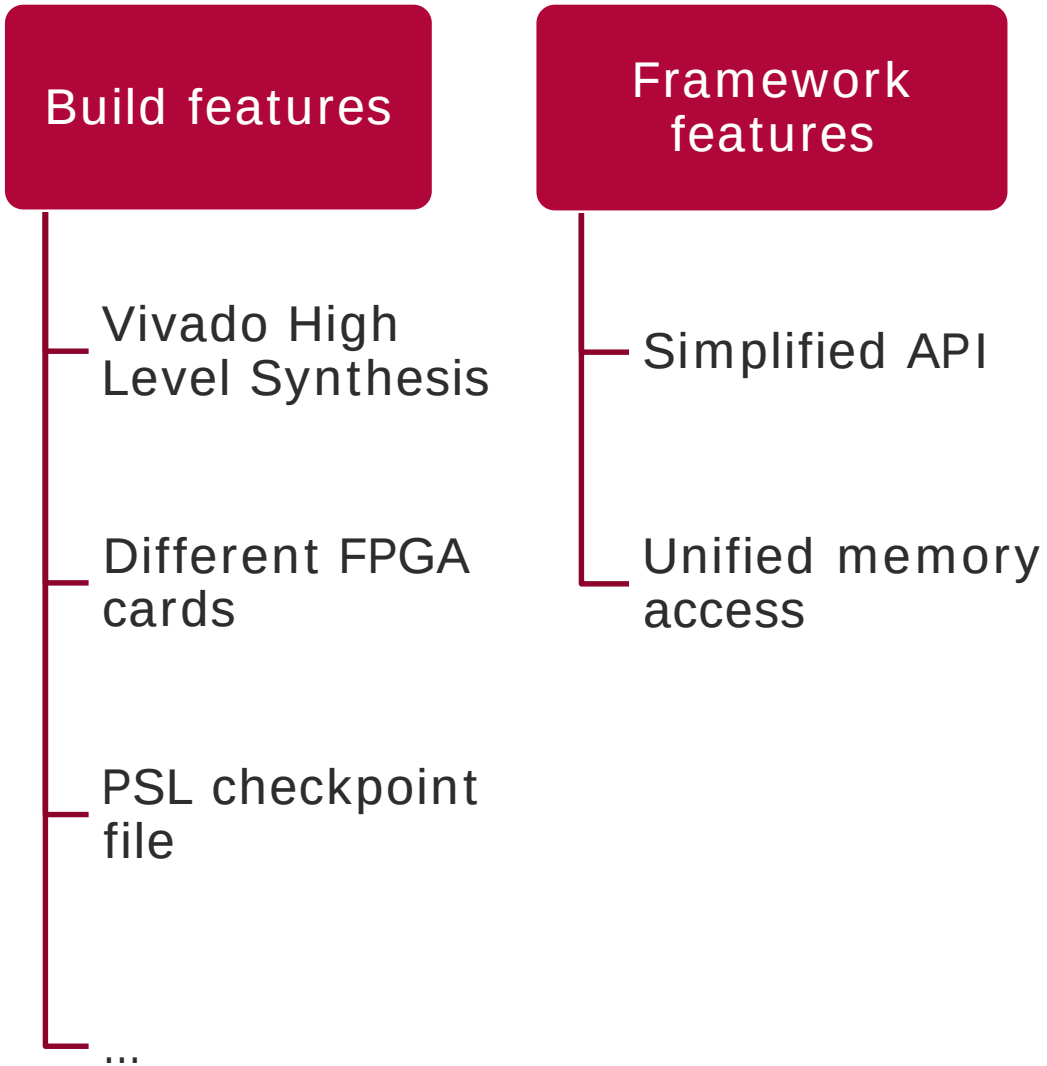
```
balthasar2@plauth-ws:~/repos/afu-hello-world/afu$ make
rm -rf xsim.dir .Xil
rm -f *.jou *.log *.pb libdpi.so
ln -s /home/balthasar2/repos/pslse//afu_driver/src/libdpi.so .
/opt/Xilinx/Vivado/2016.4/bin/xvlog -sourcelibdir . -sourcelibext .v -sourcelibext .
sv -sv top.v
INFO: [VRFC 10-2263] Analyzing SystemVerilog file "/home/balthasar2/repos/afu-hello-
world/afu/top.v" into library work
INFO: [VRFC 10-311] analyzing module top
```

HPI Hasso Plattner Institut

IBM

Demo

CAPI SNAP provides a simple API and a unified build process with support for High Level Synthesis



- Vivado Suite is GUI focused, automating it requires some learning
- A lot of different components are needed for creating or simulating a CAPI FPGA image
- CAPI Developers need to think of job and memory management
- ▶ SNAP for easy building and higher level development
- Provides different ready-to-go examples
  - Breadth-first search, hashjoin, memcopy, ...
- Simulation based on CAPI

CAPI SNAP provides a simple API and a unified build process with support for High Level Synthesis

Source Vivado settings file, export Vivado license file location

```
balthasar2@plauth-ws:~/repos/snap/hardware$ source snap_settings
=========================================================
== SNAP SETUP                                          ==
=========================================================
=====Checking Xilinx Vivado:============================
Path to vivado          is set to: /opt/Xilinx/Vivado/2016.4/bin/vivado
Vivado version          is set to: Vivado v2016.4 (64-bit)
=====CARD variables=====================================
FPGACARD                is set to: "FGT"
FPGACHIP                is set to: "xcku060-ffva1156-2-e"
PSL_DCP                 is set to: "/home/balthasar2/cards/FGT/portal_20170413/b_route_design.dcp"
=====SNAP PATH variables================================
SNAP_ROOT               is set to: "/home/balthasar2/repos/snap"
ACTION_ROOT             is set to: "/home/balthasar2/repos/snap/hardware/action_examples/hdl_example"
=====SNAP simulation variables==========================
PSLSE_ROOT              is set to: "/home/balthasar2/repos/pslse/"
SIMULATOR               is set to: "xsim"
=====SNAP function variables============================
NUM_OF_ACTIONS          is set to: "1"
SDRAM_USED              is set to: "FALSE"
NVME_USED               is set to: "FALSE"
ILA_DEBUG               is set to: "FALSE"
```

We implemented the symmetric block cipher Blowfish in hardware

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

- Blowfish-AFU
  - 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*

```
15  #define MODE_SET_KEY 0
16  #define MODE_ENCRYPT 1
17  #define MODE_DECRYPT 2
18
19  #ifndef CACHELINE_BYTES
20  #define CACHELINE_BYTES 128
21  #endif
22
23  // Blowfish Configuration PATTERN.
24  // This must match with DATA struc
25  // Job description should start wi
26  typedef struct blowfish_job {
27      struct snap_addr input_data;
28      struct snap_addr output_data;
29      uint32_t mode;
30      uint32_t data_length;
31  } blowfish_job_t;
```

```
balthasar2@plauth-ws:~/repos/snap/hardware$ ./snap_settings
==========================================================
== SNAP SETUP                                           ==
==========================================================
=====Checking Xilinx Vivado:============================
Path to vivado          is set to: /opt/Xilinx/Vivado/2016.4/bin/vivado
Vivado version          is set to: Vivado v2016.4 (64-bit)
=====CARD variables=====================================
Setting FPGACARD                to: "FGT"
```

HPI Hasso Plattner Institut

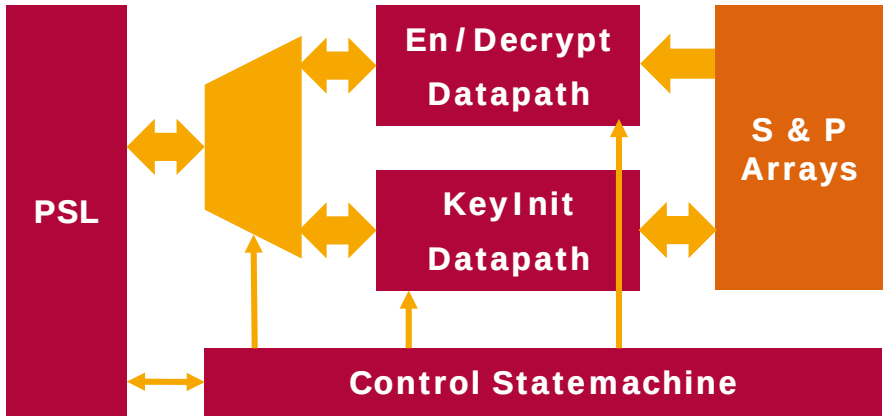
IBM

Demo

We implemented the symmetric block cipher Blowfish in hardware

```
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;
}
```

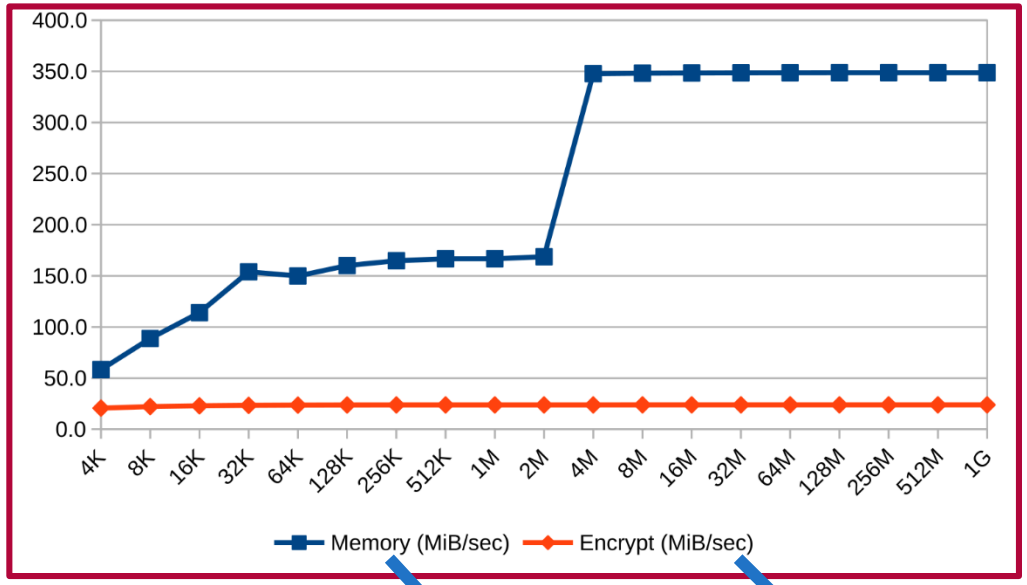
translates to

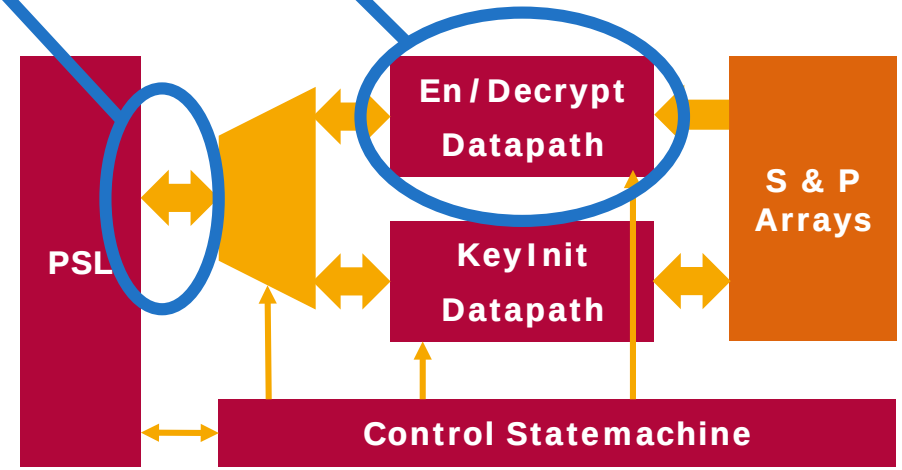


Performance optimization depends on a detailed analysis of different aspects of the AFU design

- Relevant parts of the AFU design must be analyzed to locate bottlenecks

- Blowfish-AFU: Throughput oriented scenario, compare memory and encryption bandwidth
  - ▶ Memory interface suports up to 16 times the encrypt throughput
  - ▶ Multiple instances of encrypt hardware can acheive overall speedup





Performance optimization requires a deeper understanding of the underlying hardware

- To support multiple parallel encrypt, resource conflicts must be eliminated

- Block encrypt uses the `bf_f()` function: four sequential argument dependent read operations
- Multiple instances of `bf_f()` require independent read ports to the S-Array; implementation provides Dual-Port-RAM
- ▶ Solution: More Read-Only Ports can be acheived by providing multiple Copies of the S-Array

```
static bf_halfBlock_t bf_f(bf_halfBlock_t h)
{
    bf_SiE_t a = (bf_SiE_t)(h >> 24),
             b = (bf_SiE_t)(h >> 16),
             c = (bf_SiE_t)(h >> 8),
             d = (bf_SiE_t) h;
    return ((g_S[0][a] + g_S[1][b]) ^ g_S[2][c]) + g_S[3][d];
}
```

Performance optimization requires a deeper understanding of the underlying hardware

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)

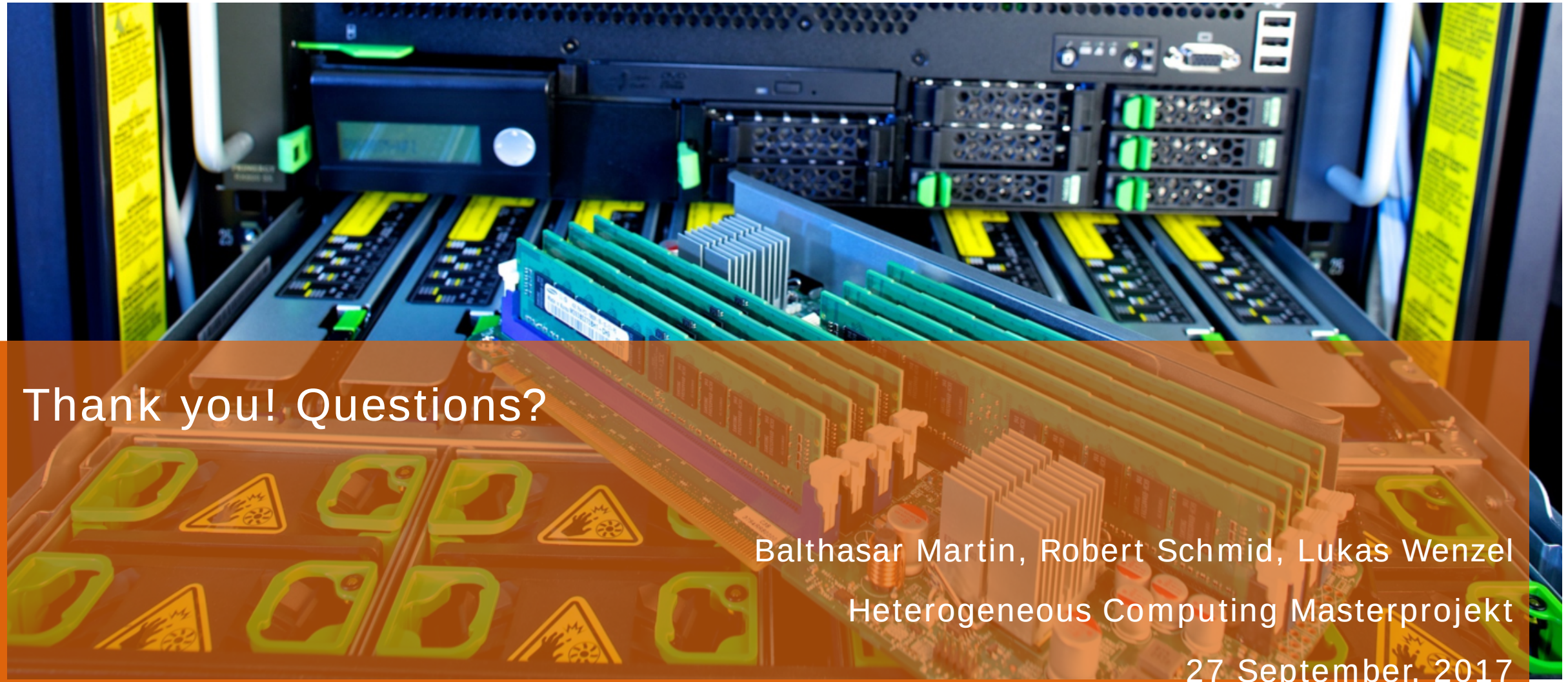
Current Module : hls_action > process_action > action_endecrypt >

	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];
    }
}
```



Thank you! Questions?

Balthasar Martin, Robert Schmid, Lukas Wenzel

Heterogeneous Computing Masterprojekt

27 September, 2017