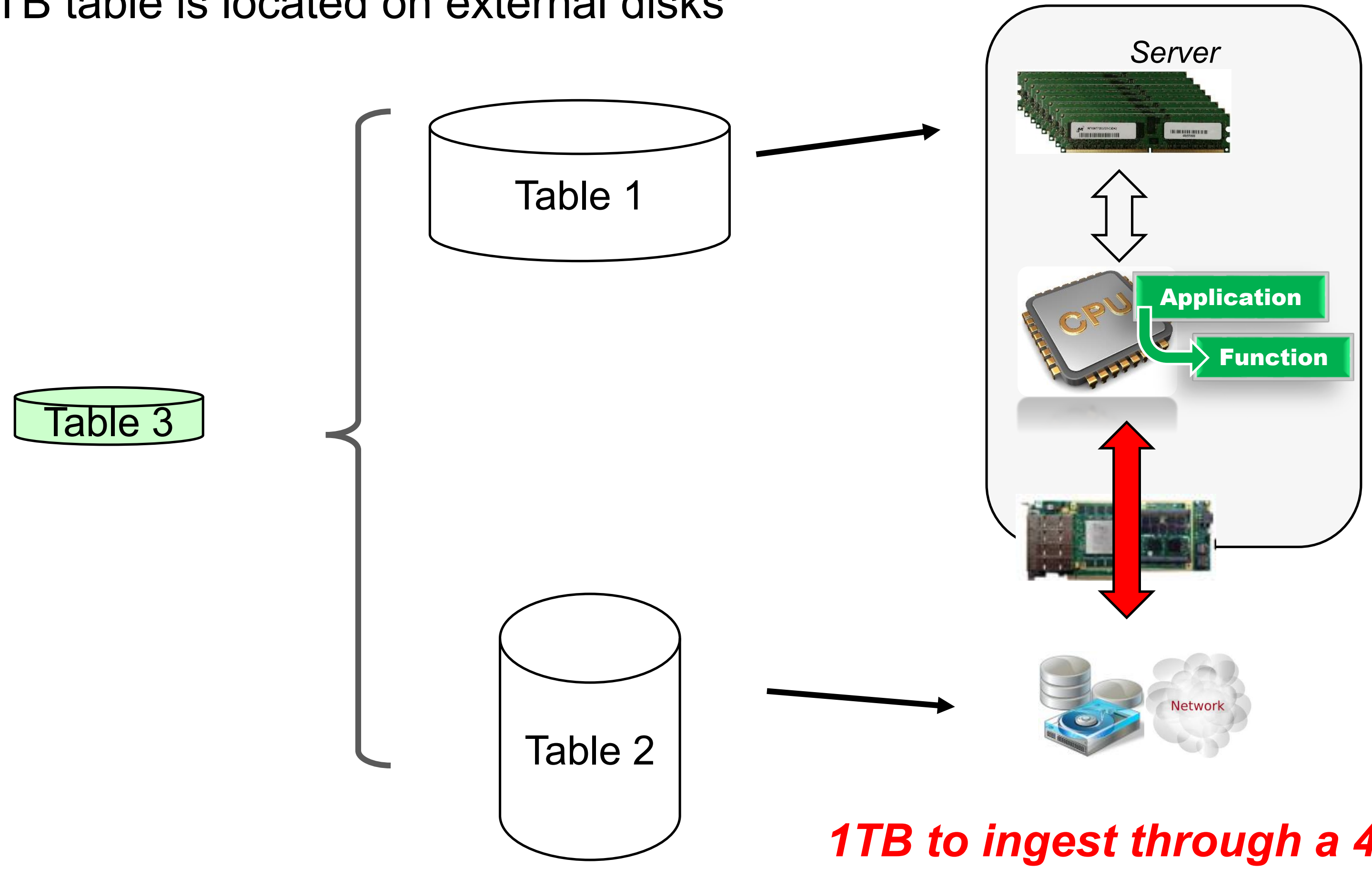
Offload?
State of the art, CAPI, OpenCAPI,…

Understand how to offload a server (1/3)

Use-case: find the **common elements** of 2 tables
- 1 TB table is located in host memory
- 1 TB table is located on external disks

① « Classic » CPU execution



- Direct access to Host memory

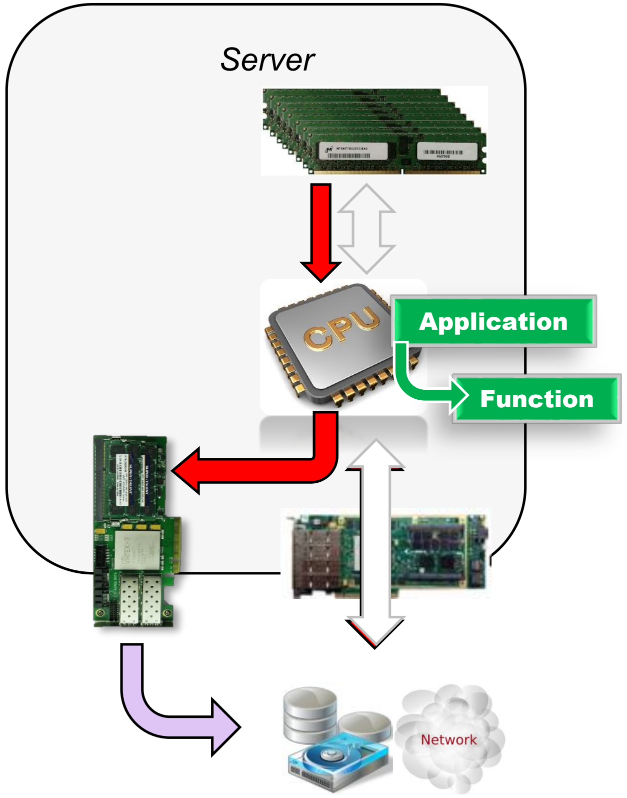
- TB of data to ingest through a 100Gb/s ethernet card
  → **Network + host memory usage**

1TB to ingest through a 4x100Gb/s card takes 20.5 secs!

Where are data located??

Understand how to offload a server (2/3)

② Adding a « *classic* » PCIe FPGA card



☺

- Function is **offloaded / accelerated**
- Server **network** resources **savings**
- Server **memory savings**

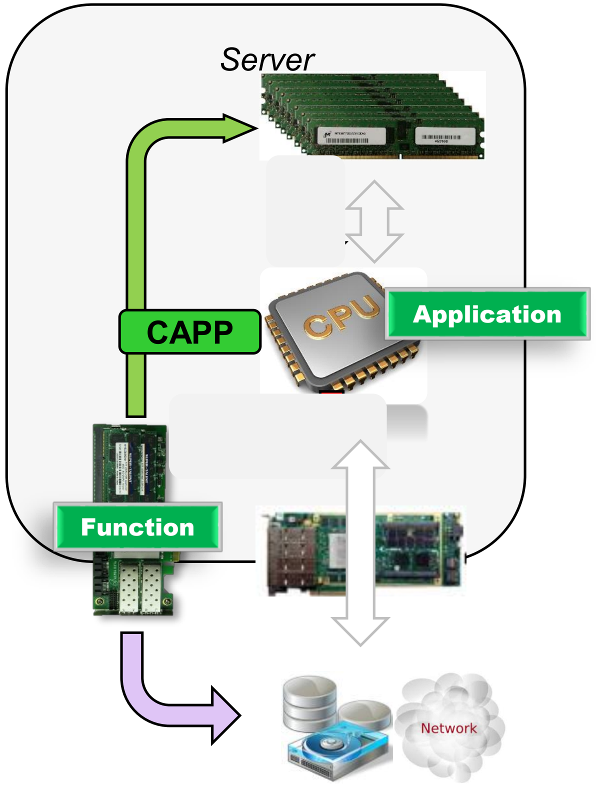
☹

- **Need a software driver**
  - → CPU + memory usage
  - → adding a level of code complexity
  - → losing direct access to Host memory
- **FPGA card is a SLAVE**
  - ➔ ALL data pushed to the FPGA
    - → High utilization of **PCIe BW**
    - → **data coherency lost**
- **1 user / 1 application / 1 function**

Understand how to offload a server (3/3)

- Function is **offloaded / accelerated**
- Server **network** resources **savings**
- Server **memory savings**

③ Adding a « *CAPI-enabled* » FPGA card



- **CAPP = CAPI Hardware driver**
  → CPU + memory savings
- **FPGA card is MASTER**
  → Function accesses **only host data needed**
  → **coherency of data**
  → Address translation (@action=@application)
- **Multiple threads / multiple users** can be associated **to multiple actions**

BONUS:

- **Very small latency**
- **Very high bandwidth**
- **Programming the FPGA** can be done using basic **C/C++**
- **POWER simulation model** to quicker code testing
- **Open-Source SNAP framework** to quicker connections
- **Card manufacturer independent**

CAPI/OpenCAPI evolution: Increase bandwidth and reduce latency

P8 / CAPI1.0

PCIeGen3x8 @8Gb/s

~4GB/s measured

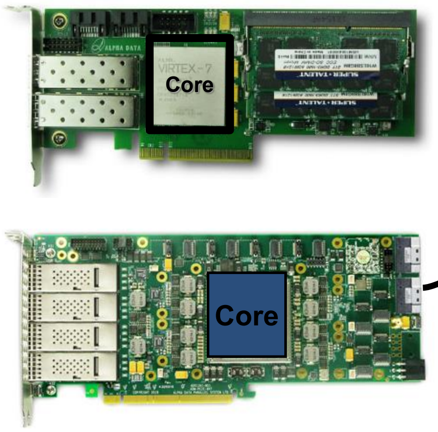
~800ns latency

P9 / CAPI2.0

PCIeGen4x8 @16Gb/s

~14 GB/s measured

est. <555ns total latency



CAPI2.0

OpenCAPI3.0

P9 / OpenCAPI3.0

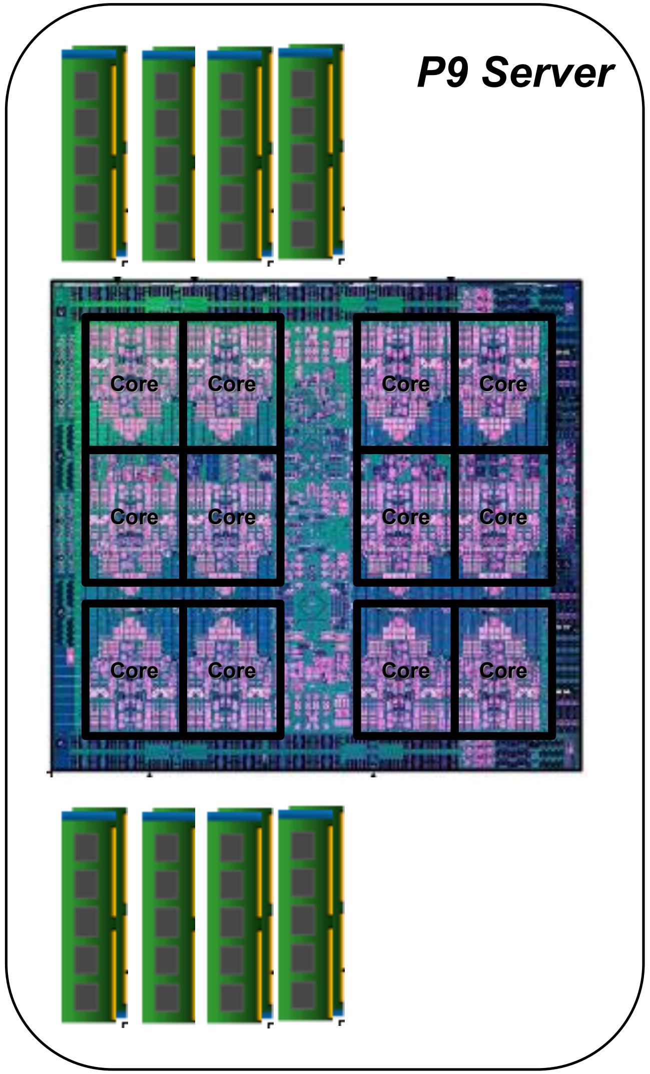
OpenCAPI link 25Gb/s 8 lanes

~22GB/s measured

378ns total latency

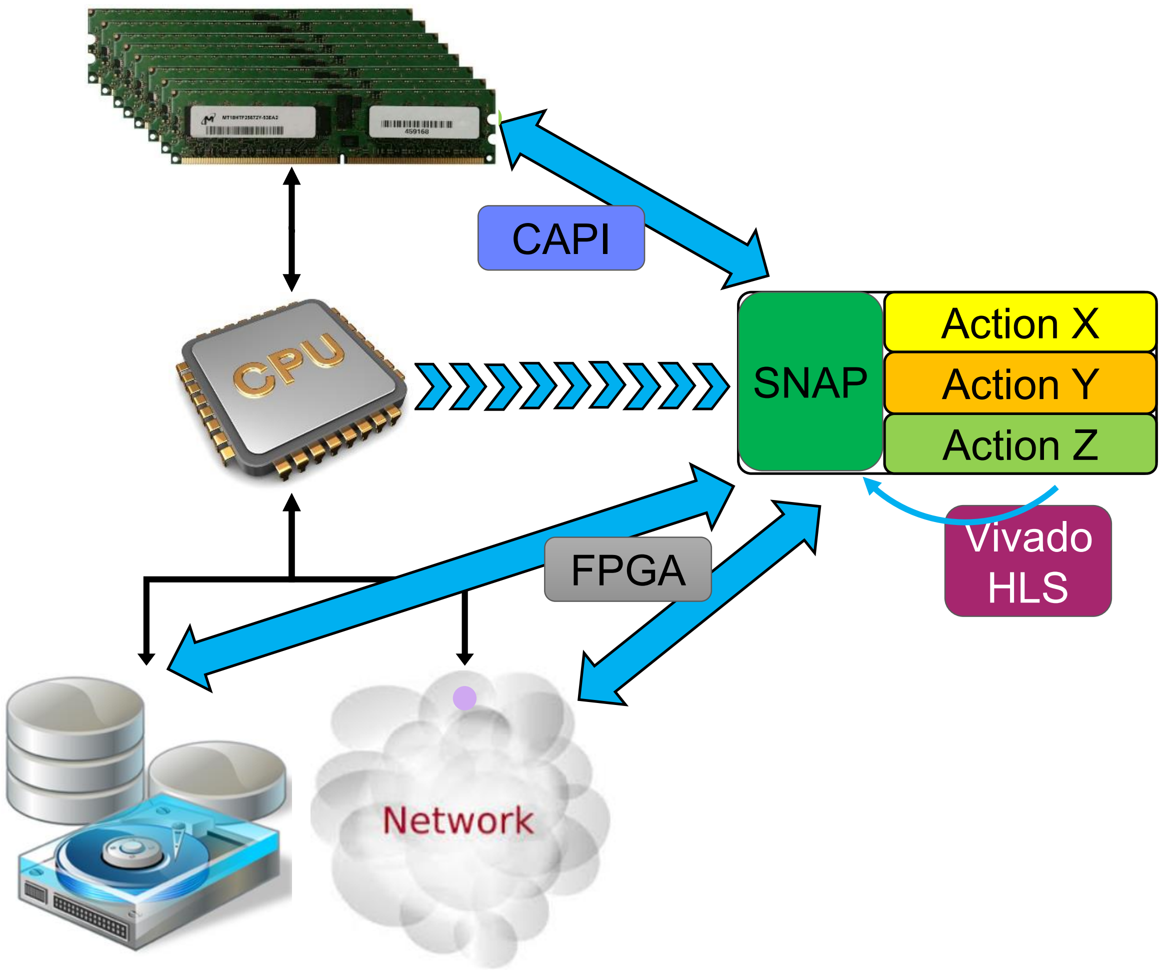
<u>"Total latency" test on OpenCAPI3.0:</u>
Simple workload created to simulate communication between system and attached FPGA

1. *Copy 512B from host send buffer to FPGA*
2. *Host waits for 128 Byte cache injection from FPGA and polls for last 8 bytes*
3. *Reset last 8 bytes*
4. *Repeat Go TO 1.*



Coding: RTL? C/C++? SNAP?

The CAPI – SNAP concept



CAPI FPGA becomes a peer of the CPU
➔ Action **directly** accesses host memory

+

SNAP Manage server threads and actions
Manage access to IOs (memory, network)
➔ Action **easily** accesses resources

+

FPGA Gives on-demand compute capabilities
Gives direct IOs access (storage, network)
➔ Action **directly** accesses external resources

+

Vivado HLS Compile Action written in C/C++ code
Optimize code to get performance
➔ Action code **can be ported efficiently**

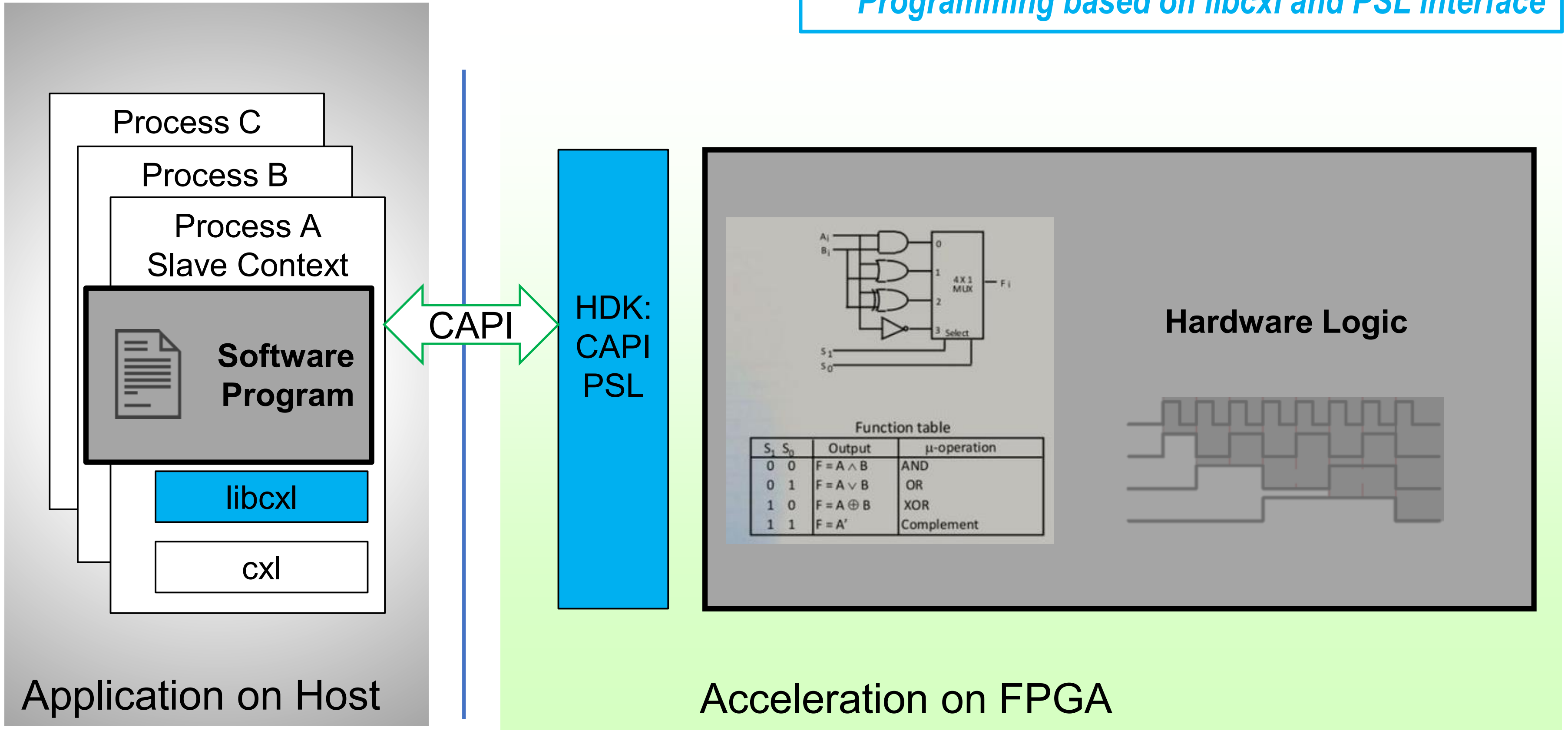
=

Offload/accelerate a C/ C++ code with :
- Quick porting
- Minimum change in code
- Better performance than CPU

FPGA development: Choice1 (air liner cockpit)

Big developing efforts
Extreme performance targeted, full control
Programming based on libcxl and PSL interface

- Develop your code
  - Software side: on libcxl APIs
  - FPGA side: on PSL interface
    - Or TLx for OpenCAPI



Function table

S_1 S_0	Output	µ-operation
0 0	$F = A \wedge B$	AND
0 1	$F = A \vee B$	OR
1 0	$F = A \oplus B$	XOR
1 1	$F = A'$	Complement

FPGA development: Choice2 (Recommended, simplified cockpit)

- **CAPI SNAP** is an environment that makes it easy for programmers to create FPGA accelerators and integrate them into their applications.
  - **Security** based on IBM POWER's technology.
  - **Portable** from CAPI1.0, 2.0 to OpenCAPI
  - **Open-source** (once a driver is available, everyone can make use of it)

https://github.com/open-power/snap

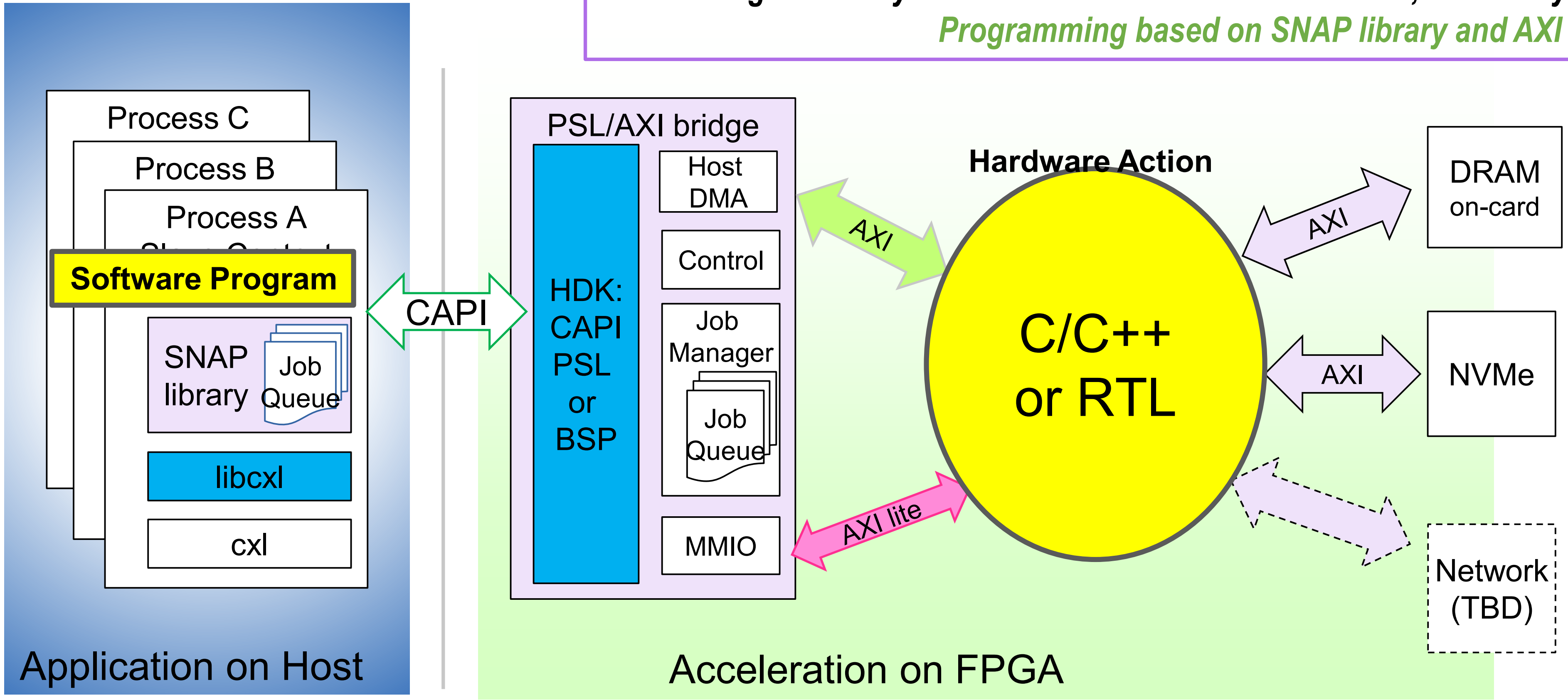
Storage, Networking, Analytics Programming framework

SNAP framework

Quick and easy developing

Use High Level Synthesis tool to convert C/C++ to RTL, or directly use RTL

Programming based on SNAP library and AXI interface

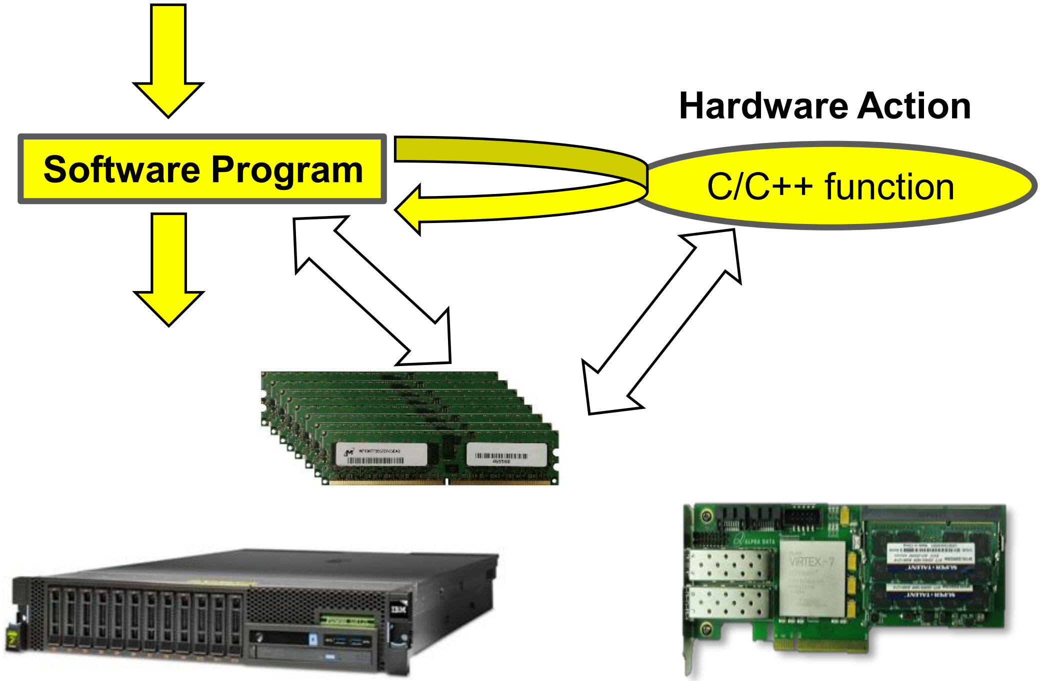


__AXI__ is an industry standard for on-chip interconnection (https://www.arm.com/products/system-ip/amba-specifications)

2 different working modes

The Job-Queue Mode
SERIAL MODE

FPGA-action executes a job and returns after completion



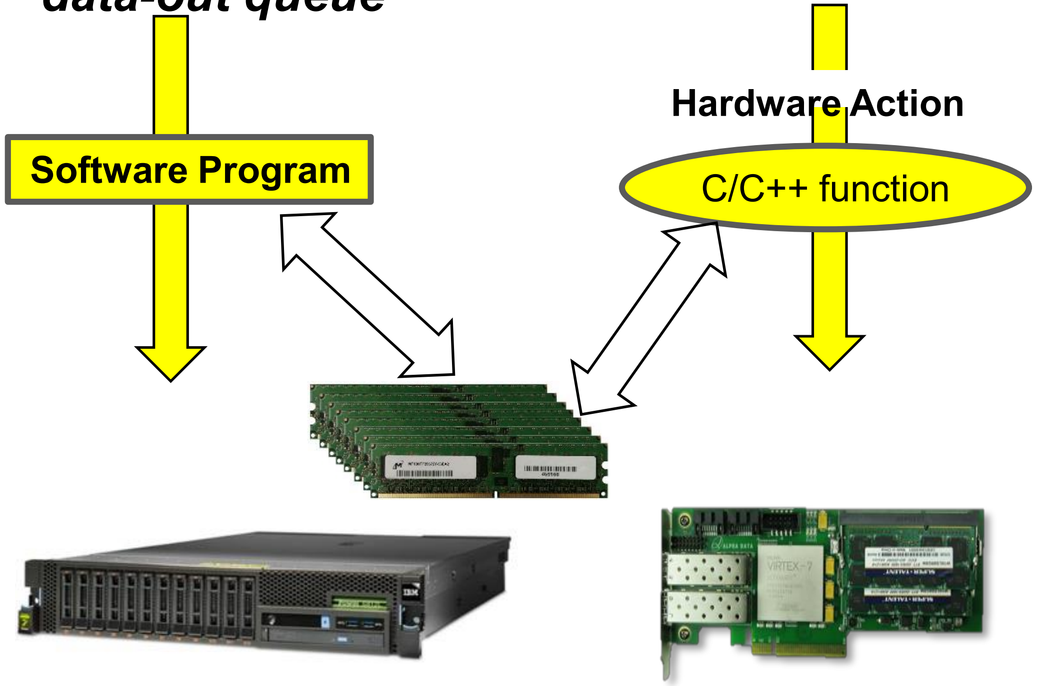
Hardware Action

Software Program

C/C++ function

The Fixed-Action Mode
PARALLEL MODE

FPGA-action is designed to permanently run
Data-streaming approach with data-in and data-out queue



Hardware Action

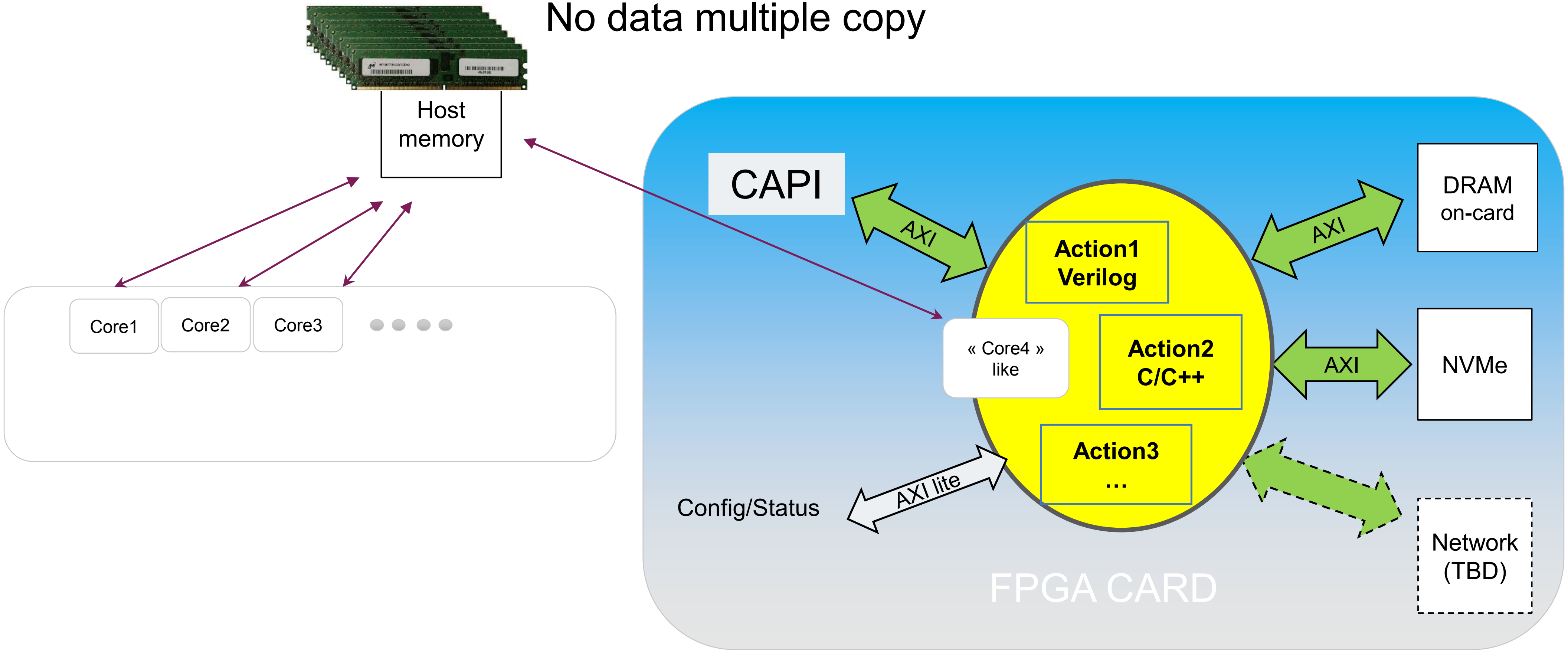
Software Program

C/C++ function

Why CAPI is simpler and faster ? Because of the coherency of memory

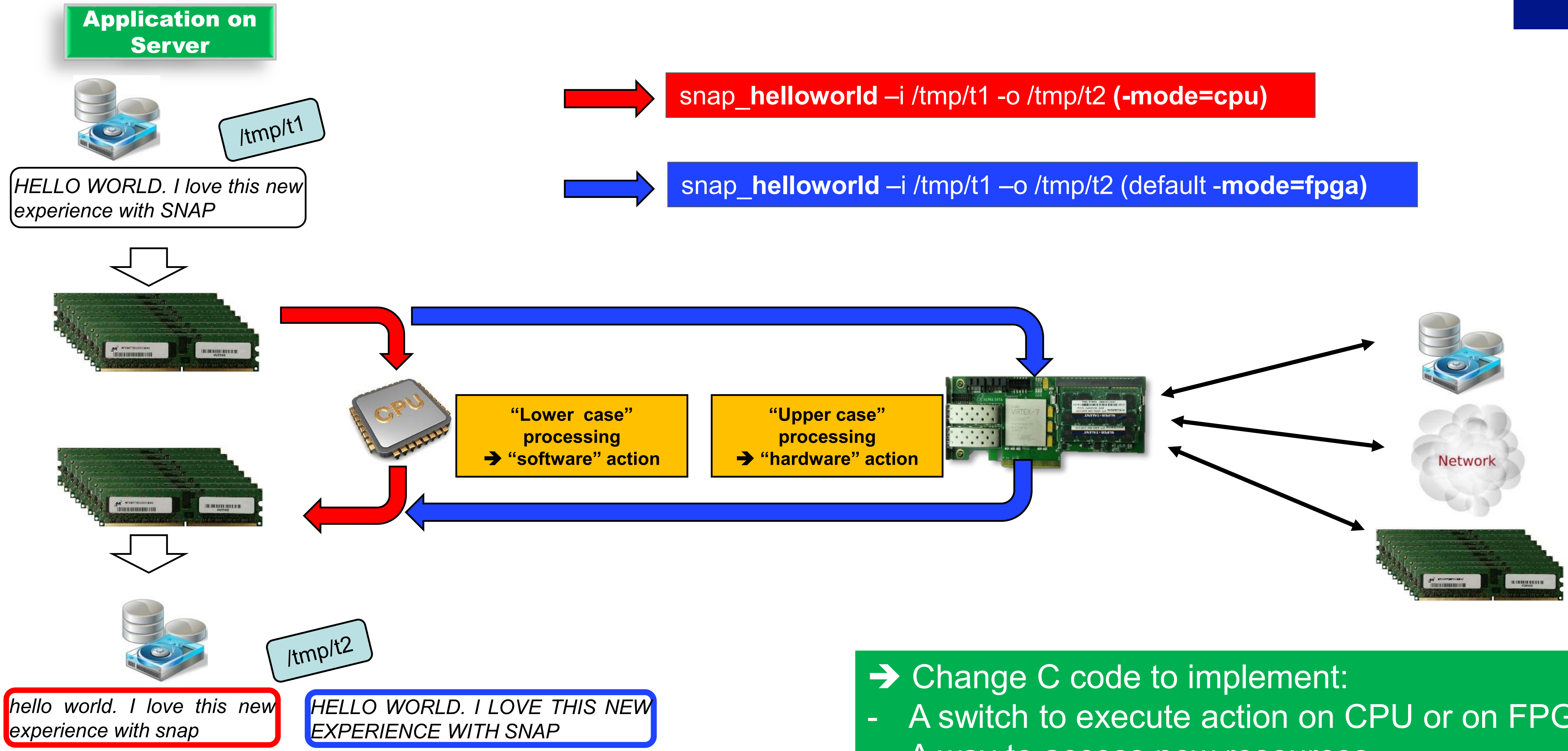
Place computing closer to data

No data multiple copy



From ***CPU-centric*** architecture …. to a …… ***Server memory centric*** architecture

Let’s understand SNAP with a “hello world” example



Application on Server

/tmp/t1

HELLO WORLD. I love this new experience with SNAP

```
snap_helloworld –i /tmp/t1 -o /tmp/t2 (-mode=cpu)
```

```
snap_helloworld –i /tmp/t1 –o /tmp/t2 (default -mode=fpga)
```

“Lower case” processing ➔ “software” action

“Upper case” processing ➔ “hardware” action

/tmp/t2

hello world. I love this new experience with snap

HELLO WORLD. I LOVE THIS NEW EXPERIENCE WITH SNAP

➔ Change C code to implement:

- A switch to execute action on CPU or on FPGA
- A way to access new resources

SNAP solution Flow : prepare the data (hls_helloworld example)

Application | **C/ C++ code used in Application**

① **Fill input data into host server memory**
- Evaluate input file size
- Allocate memory area (64Bytes aligned)
- Read data from input file and fill **ibuff** with data from input file

```
size = __file_size(input);
addr_in = snap_malloc(size)
rc = __file_read(input, addr_in, size)
```

② **Prepare host server memory to store the results:**
- Evaluate output file size (same than input)
- Allocate memory area (64 Bytes aligned)

```
addr_out = snap_malloc(size)
```

③ **Prepare parameters to be written in MMIO registers:**
- **type_in** = SNAP_ADDRTYPE_HOST_DRAM;
- **addr_in** = (unsigned long) **ibuff**;

```
snap_addr_set(&mjob->in, addr_in, size_in, type_in,
              SNAP_ADDRFLAG_ADDR | SNAP_ADDRFLAG_SRC);
```

- **type_out** = SNAP_ADDRTYPE_HOST_DRAM;
- **addr_out** = (unsigned long) **obuff**;

```
snap_addr_set(&mjob->out, addr_out, size_out, type_out,
              SNAP_ADDRFLAG_ADDR | SNAP_ADDRFLAG_DST |
              SNAP_ADDRFLAG_END);
```

- Assign the structure **mjob** containing all parameters we just filled to the job **cjob**

```
snap_job_set(cjob, mjob, sizeof(*mjob), NULL, 0);
```

④ **Allocate the card that will be used**

```
card = snap_card_alloc_dev (device,
                  SNAP_VENDOR_ID_IBM,
SNAP_DEVICE_ID_SNAP);
```

⑤ **Allocate the action that will be used on the allocated card**

```
action = snap_attach_action (card,
                  HELLOWORLD_ACTION_TYPE, action_irq,
timeout);
```

Host System memory

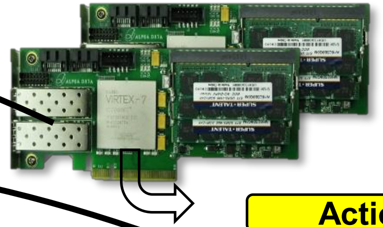
Data memory area

@addr_in
------ Input Text ------

@addr_out
------ Output text area-----

MMIO registers

@mjob
type_in, addr_in, flags_in
type_out, addr_out, flags_out



Action X
Action Y
Action Z

SNAP solution Flow : call + process the action (hls_helloworld example)

Application

C/ C++ code used in Application

Hardware function/action

6

Call the action. This will:
- Write all registers to the action (MMIO)
- Start the action
- Wait for completion (interrupt, MMIO polling, or timeout)
- Read all registers from the action (MMIO)

```
rc = snap_action_sync_execute_job(action, &cjob, timeout);
```

This starts the execution of the *software* or *hardware* function//action code

MMIO registers

@mjob
type_in, addr_in, flags_in
type_out, addr_out, flags_out

```
i_idx = act_reg->Data.in.addr >> ADDR_RIGHT_SHIFT;
o_idx = act_reg->Data.out.addr >> ADDR_RIGHT_SHIFT;
size = act_reg->Data.in.size;

memcpy((char*) text, din_gmem + i_idx, size);

for (i = 0; i < sizeof(text); i++ )
                    if (text[i] >= 'a' && text[i] <= 'z')
                                   text[i] = text[i] - ('a' - 'A');

memcpy(dout_gmem + o_idx, (char*) text, size);

act_reg->Control.Retc = SNAP_RETC_SUCCESS;
```

1. Get and align the **input_data_address**, **input_data _address** and **size to access** (MMIO)
2. Read **data** from **input_data** address directly in host memory server (din_gmem)
3. Process the data (*uppercase conversion*)
4. Write **data** to **output_data** address directly in host memory server (dout_gmem)
5. Fill the return code

The end of the code sends to the application an interrupt (if set)

Host System memory

Data memory area

@addr_in
------ Input Text------

@addr_out
------ Output text area-----

IMPORTANT : The application doesn't need to **wait** for the function completion since function doesn't "return" any data to the application but writes results directly into the host memory

SNAP solution Flow : free the action (hls_helloworld example)

Application	C/ C++ code used in Application
⑦ **Read output data from the host server memory and write them to output file** - Read data from host server (**obuf**) and write data to output file	rc = __file_write(output, **addr_out**, size);

⑧ **Detach action**
Disallocate the card
Free the dynamic allocation of buffers
	snap_detach_action(action);
	snap_card_free(card);
	__free(obuff);
	__free(ibuff);

Host System memory

Data memory area

@addr_in
------ Input Text------

@addr_out
------ Output text ----

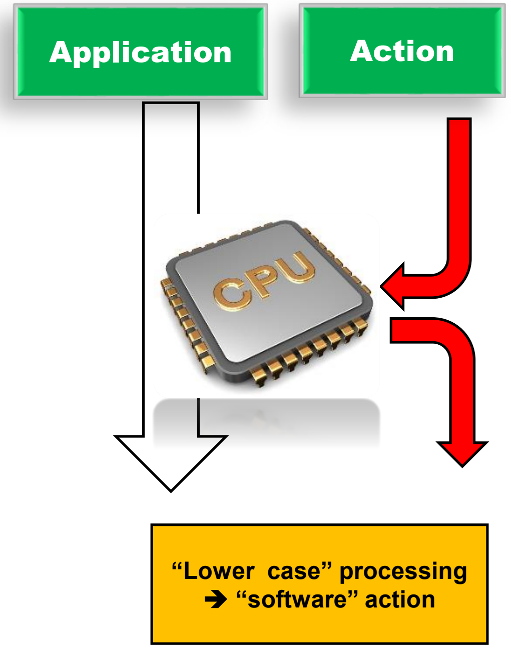
Action X
Action Y
Action Z

A SIMPLE 3 STEPS PROCESS

No specific test bench required
Use your actual application

① ISOLATION

```
SNAP_CONFIG=CPU snap_helloworld –i
/tmp/t1 -o /tmp/t2
```



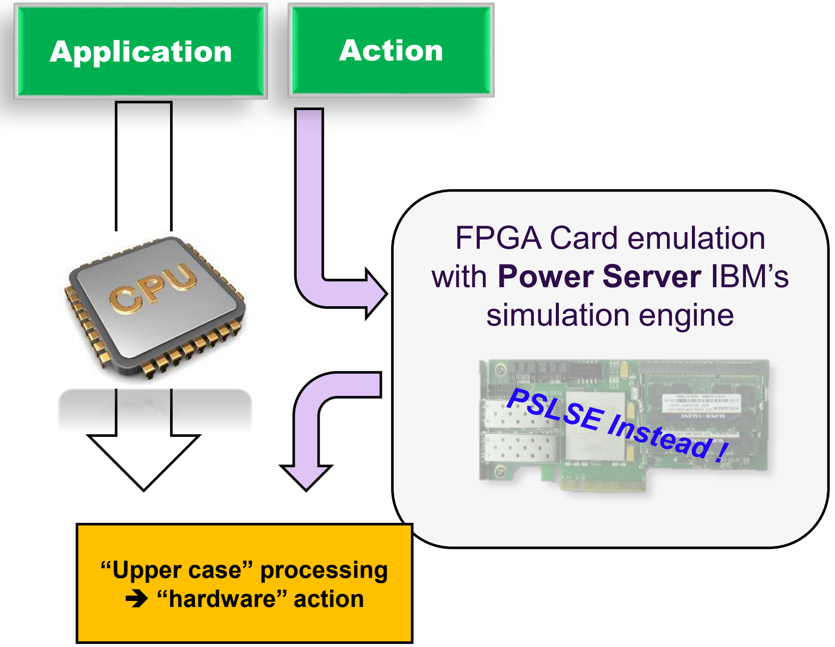
"Lower case" processing
➔ "software" action

x86 server

command: **make**

② SIMULATION

```
SNAP_CONFIG=FPGA
snap_helloworld –i /tmp/t1 –o /tmp/t2
```



FPGA Card emulation with **Power Server** IBM's simulation engine

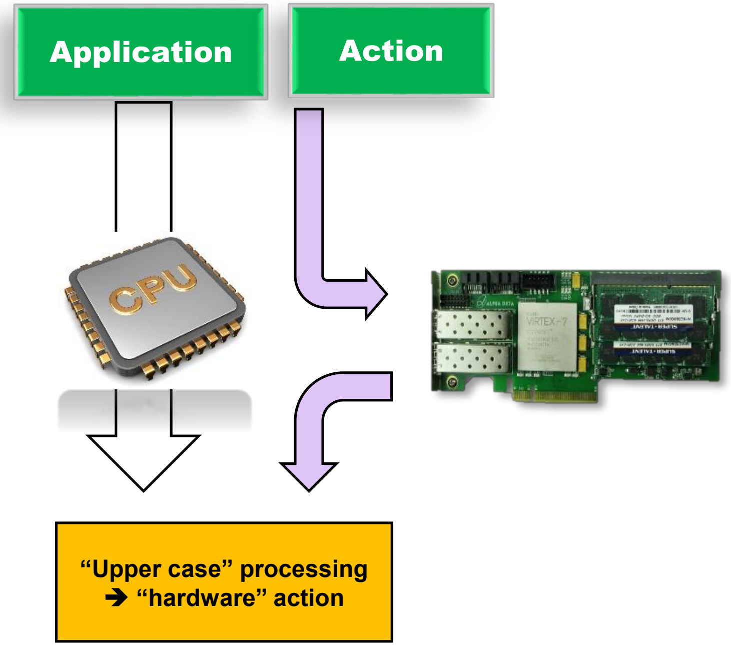
"Upper case" processing
➔ "hardware" action

x86 server

command: **make sim**

③ EXECUTION

```
SNAP_CONFIG=FPGA
snap_helloworld –i /tmp/t1 –o /tmp/t2
```

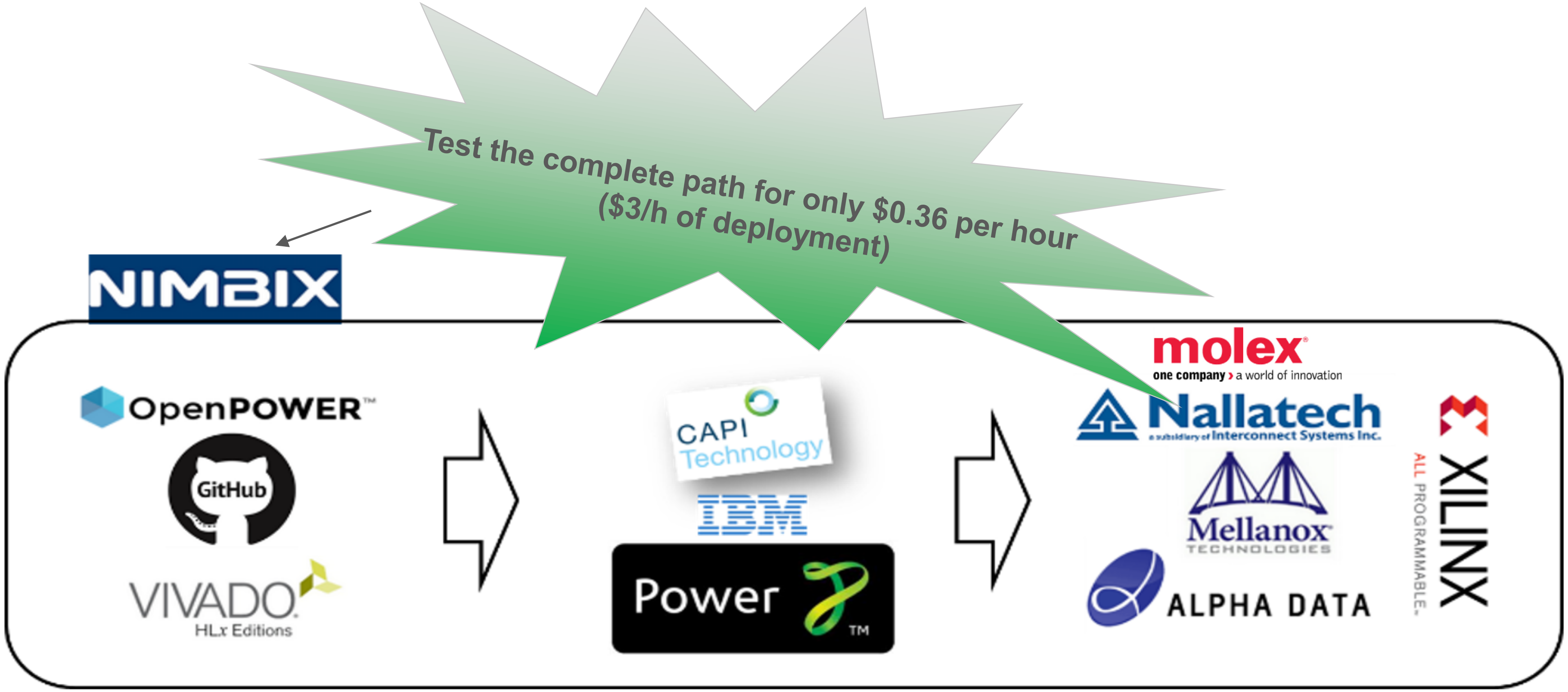


"Upper case" processing
➔ "hardware" action

POWER8/9 server

command: **make image**

SNAP Ecosystem



Test the complete path for only $0.36 per hour ($3/h of deployment)

Software – Hardware – External cards

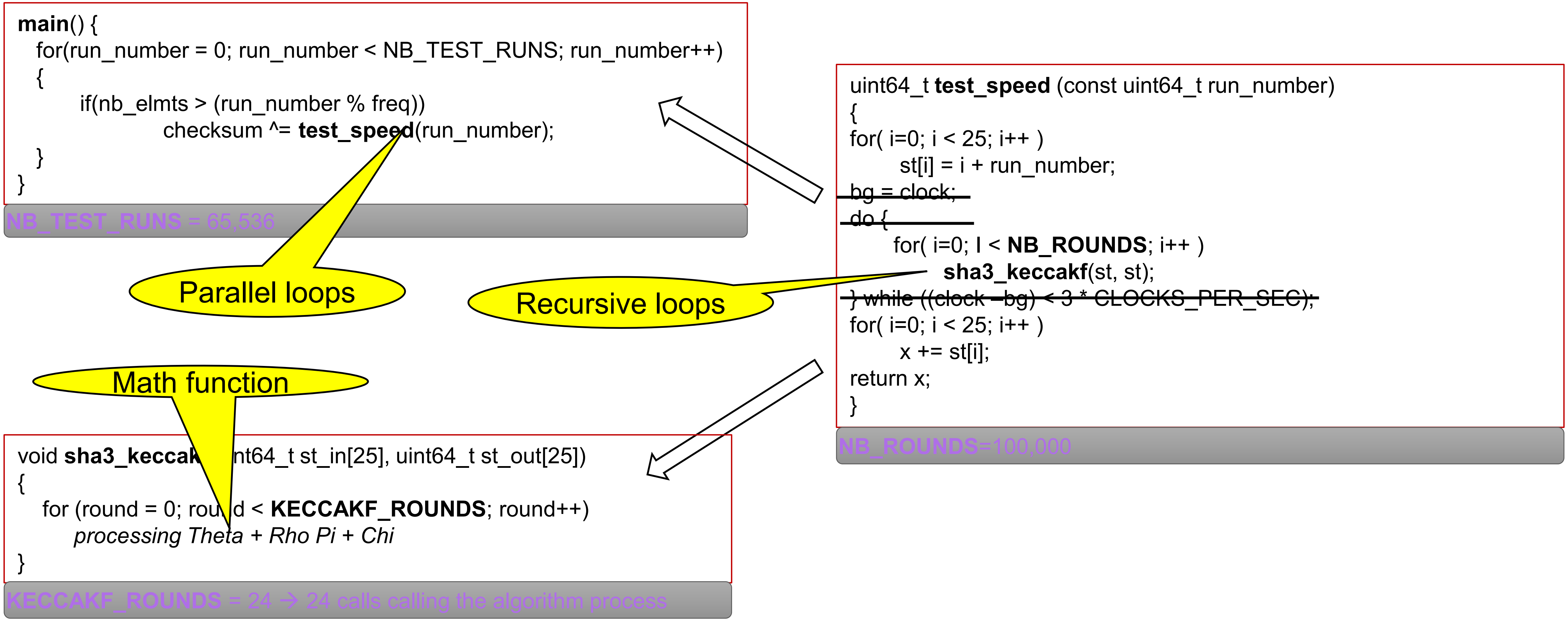
BACKUP SLIDES

SHA3 example

The SHA3 test_speed program structure:
➔ 2 parameters : NB_TEST_RUNS, NB_ROUNDS

As measuring time with HLS is not obvious, the “time” loop was modified so that parallelism could be done. The goal stays to execute the maximum times the keccakf algorithm per second.

Code used was downloaded (with author’s authorization) from: https://github.com/mjosaarinen/tiny_sha3



```
main() {
  for(run_number = 0; run_number < NB_TEST_RUNS; run_number++)
  {
       if(nb_elmts > (run_number % freq))
                 checksum ^= test_speed(run_number);
  }
}
```

NB_TEST_RUNS = 65,536

```
uint64_t test_speed (const uint64_t run_number)
{
for( i=0; i < 25; i++ )
      st[i] = i + run_number;
~~bg = clock;~~
~~do {~~
     for( i=0; I < NB_ROUNDS; i++ )
           sha3_keccakf(st, st);
~~} while ((clock – bg) < 3 * CLOCKS_PER_SEC);~~
for( i=0; i < 25; i++ )
      x += st[i];
return x;
}
```

NB_ROUNDS=100,000

```
void sha3_keccak    nt64_t st_in[25], uint64_t st_out[25])
{
   for (round = 0; rou d < KECCAKF_ROUNDS; round++)
       processing Theta + Rho Pi + Chi
}
```

KECCAKF_ROUNDS = 24 → 24 calls calling the algorithm process

Keccakf function changes

```
void sha3_keccakf(uint64_t st_in[25], uint64_t st_out[25],
{
    int i, j, round;
    uint64_t t, bc[5];
    uint64_t st[25];

     for (i = 0; i < 25; i++)
#pragma HLS UNROLL
                    st[i] = st_in[i];
    for (r = 0; r < KECCAKF_ROUNDS; r++) {
#pragma HLS PIPELINE
        // Theta
        for (i = 0; i < 5; i++)
            bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];

        for (i = 0; i < 5; i++) {
            t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
            for (j = 0; j < 25; j += 5)
                st[j + i] ^= t;
        }

        // Rho Pi
        t = st[1];
        for (i = 0; i < 24; i++) {
            j = keccakf_piln[i];
            bc[0] = st[j];
            st[j] = ROTL64(t, keccakf_rotc[i]);
            t = bc[0];
        }

        //  Chi
        for (j = 0; j < 25; j += 5) {
            for (i = 0; i < 5; i++)
                bc[i] = st[j + i];
            for (i = 0; i < 5; i++)
                st[j + i] ^= (~bc[(i + 1) % 5]) & bc[(i + 2) % 5];
        }

        //  Iota
        st[0] ^= keccakf_rndc[r];

    }
     for (i = 0; i < 25; i++)
#pragma HLS UNROLL
        st_out[i] = st[i];
}
```

Changes done for HLS:

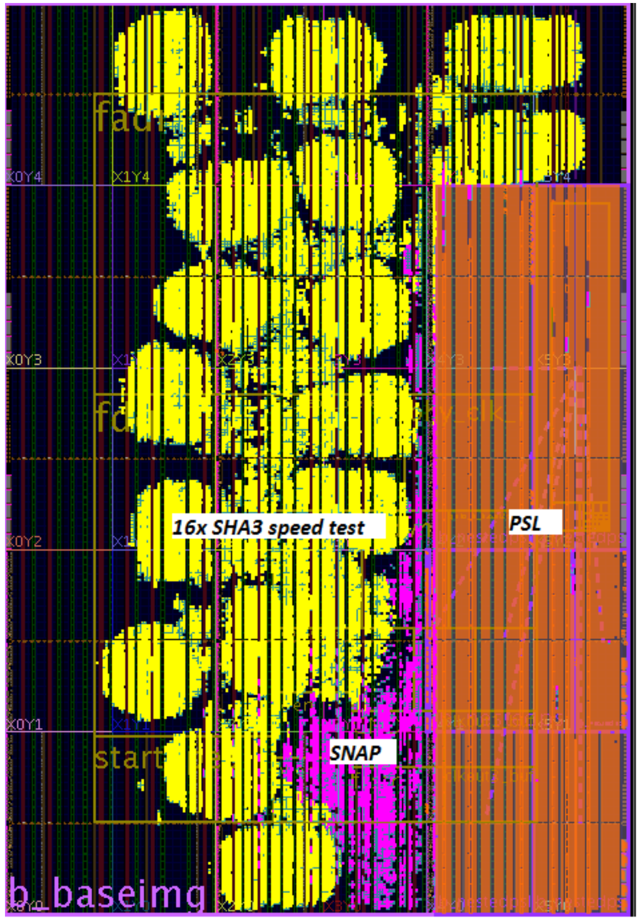
- *splitting in and out ports*
- *adding HLS PIPELINE instruction*

FYI for comparison: changes done to port this code to CUDA

```
…/…
// Rho Pi
    st0 = st00;
    st10 = ROTL643(st01, 1, 63);
    st7 = ROTL643(st010, 3, 61);
    st11 = ROTL643(st07, 6, 58);
    st17 = ROTL643(st011, 10, 54);
    st18 = ROTL643(st017, 15, 49);
    st3 = ROTL643(st018, 21, 43);
    st5 = ROTL643(st03, 28, 36);
    st16 = ROTL643(st05, 36, 28);
    st8 = ROTL643(st016, 45,19);
    st21 = ROTL643(st08, 55, 9);
    st24 = ROTL643(st021, 2, 62);
    st4 = ROTL643(st024, 14, 50);
    st15 = ROTL643(st04, 27, 37);
    st23 = ROTL643(st015, 41, 23);
    st19 = ROTL643(st023, 56, 8);
    st13 = ROTL643(st019, 8, 56);
    st12 = ROTL643(st013, 25, 39);
    st2 = ROTL643(st012, 43, 21);
    st20 = ROTL643(st02, 62, 2);
    st14 = ROTL643(st020, 18, 46);
    st22 = ROTL643(st014, 39, 25);
    st9 = ROTL643(st022, 61, 3);
    st6 = ROTL643(st09, 20, 44);
    st1 = ROTL643(st06, 44, 20);
…/…
```

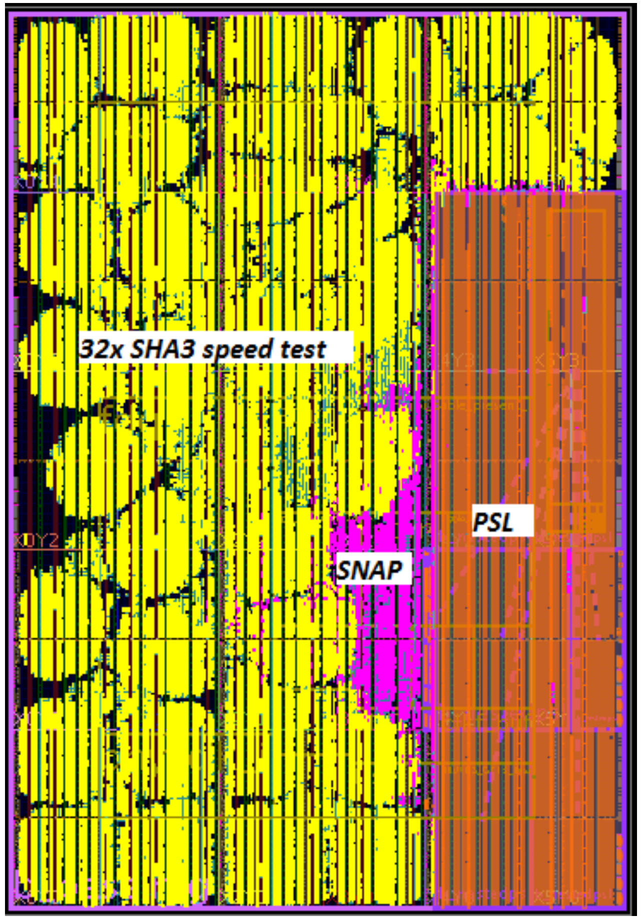
FPGA area used by the design

16 test_speed functions in parallel:



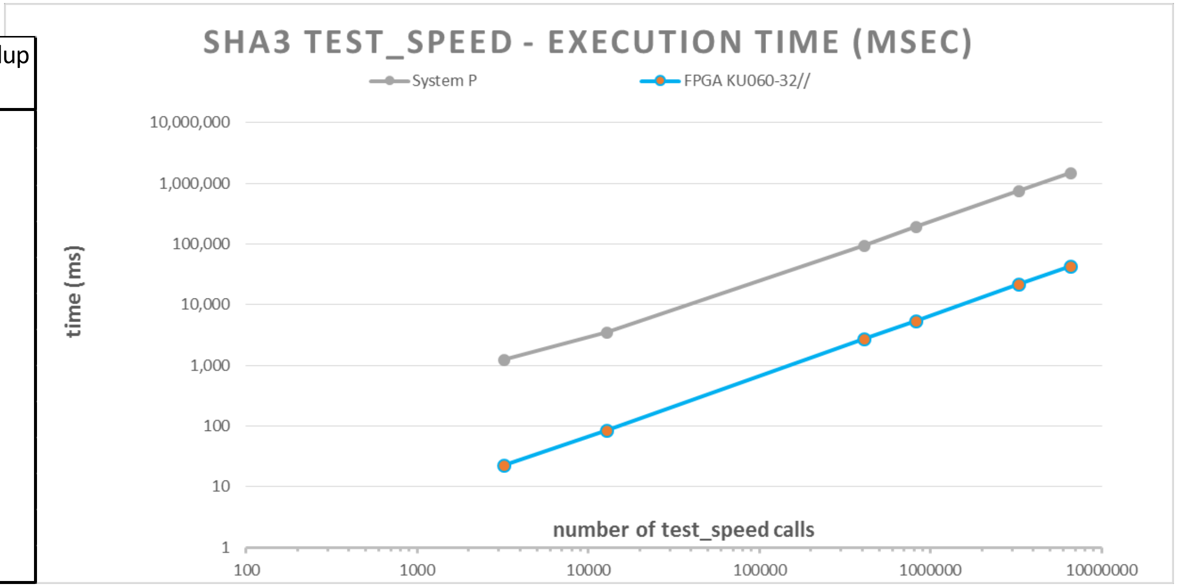
“Hardware view” just to show the place used by the logic in the FPGA

32 test_speed functions in parallel:



Offload Method:
SHA3 speed_test benchmark (on P8): FPGA is >35x faster than CPU

	slices/32	CPU (antipode) 20 cores - 160 threads	slices/32	CPU (antipode) 20 cores - 160 threads	FPGA speedup vs CPU
	FPGA KU060-32//	**System P**	**FPGA KU060-32//**	**System P**	
test_speed calls	**(keccak per sec)**	(keccak per sec)	**(msec)**	(msec)	
100,000	**4,666,573**	**149,575**	21	669	31
200,000	**9,334,453**	**295,786**	21	676	32
400,000	**18,668,036**	**488,441**	21	819	38
800,000	**37,330,845**	**865,289**	21	925	43
1,600,000	**74,672,143**	**1,572,084**	21	1,018	47
3,200,000	**143,568,576**	**2,539,064**	22	1,260	57
12,800,000	**149,900,457**	**3,699,211**	85	3,460	41
409,600,000	**150,837,950**	**4,267,759**	2,715	95,975	**35**
819,200,000	**150,900,077**	**4,303,717**	5,429	190,347	**35**
3,276,700,000	**150,937,573**	**4,344,618**	21,709	754,198	**35**
6,553,600,000	**150,941,821**	**4,352,266**	43,418	1,505,790	**35**



"The lower the better"

https://github.com/open-power/snap/tree/master/actions/hls_sponge

A Truly Heterogeneous Architecture Built on OpenCAPI

Board Level

AMD
Google
IBM
Mellanox Technologies
Micron

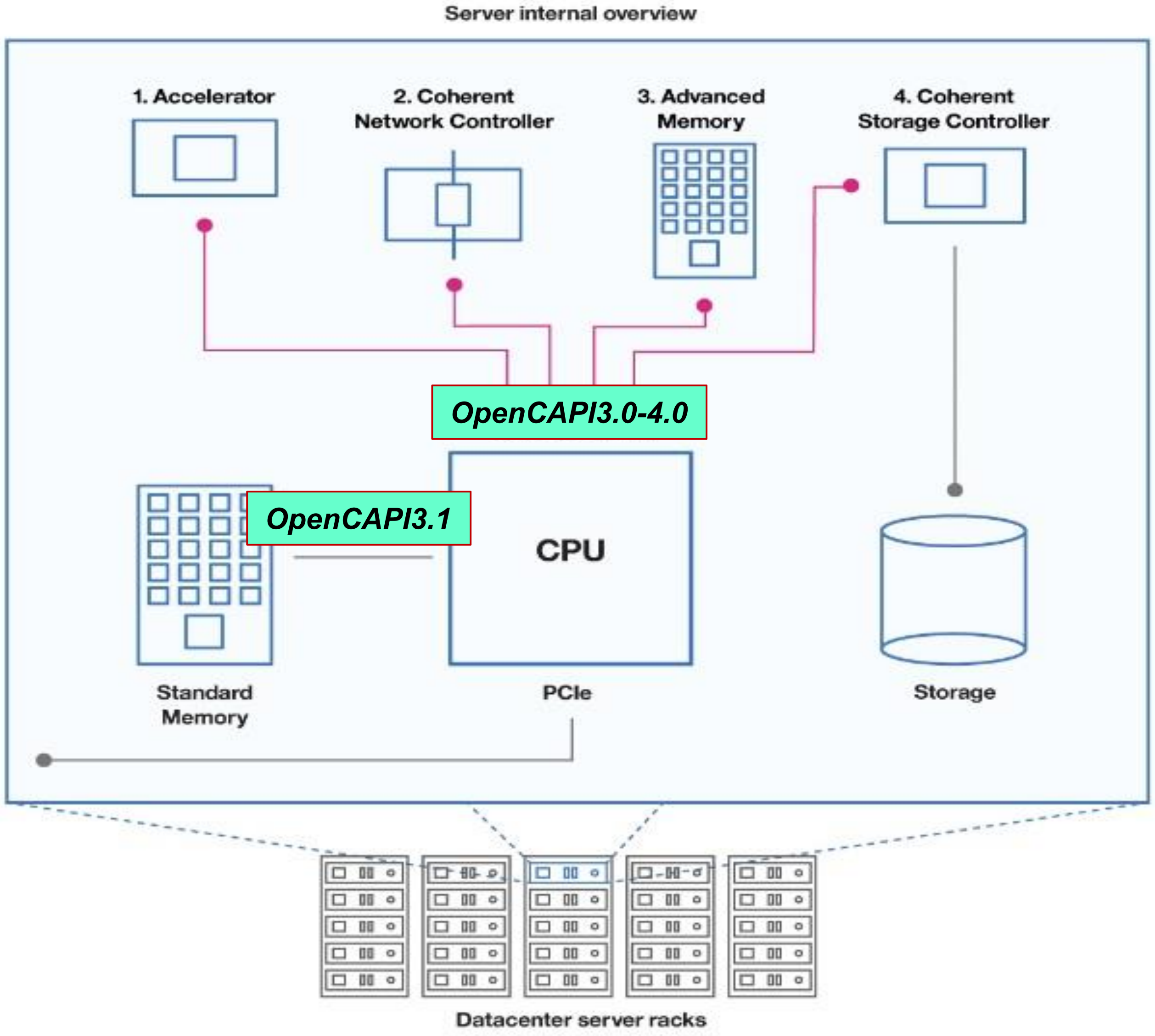
OpenCAPI

Contributor Level

Hewlett Packard Enterprise
NVIDIA
XILINX ALL PROGRAMMABLE

OpenCAPI

1. **Accelerators:** The performance, virtual addressing and coherence capabilities allow FPGA and ASIC accelerators to behave as if they were integrated into a custom microprocessor.
2. **Coherent Network Controller:** OpenCAPI provides the bandwidth that will be needed to support rapidly increasing network speeds Network controllers based on virtual addressing can eliminate software overhead without the programming complexity usually associated with user-level networking protocols.
3. **Advanced Memory:** OpenCAPI allows system designers to take full advantage of emerging memory technologies to change the economics of the datacenter.
4. **Coherent Storage Controller:** OpenCAPI allows storage controllers to bypass kernel software overhead, enabling extreme IOPS performance without wasting valuable CPU cycles.



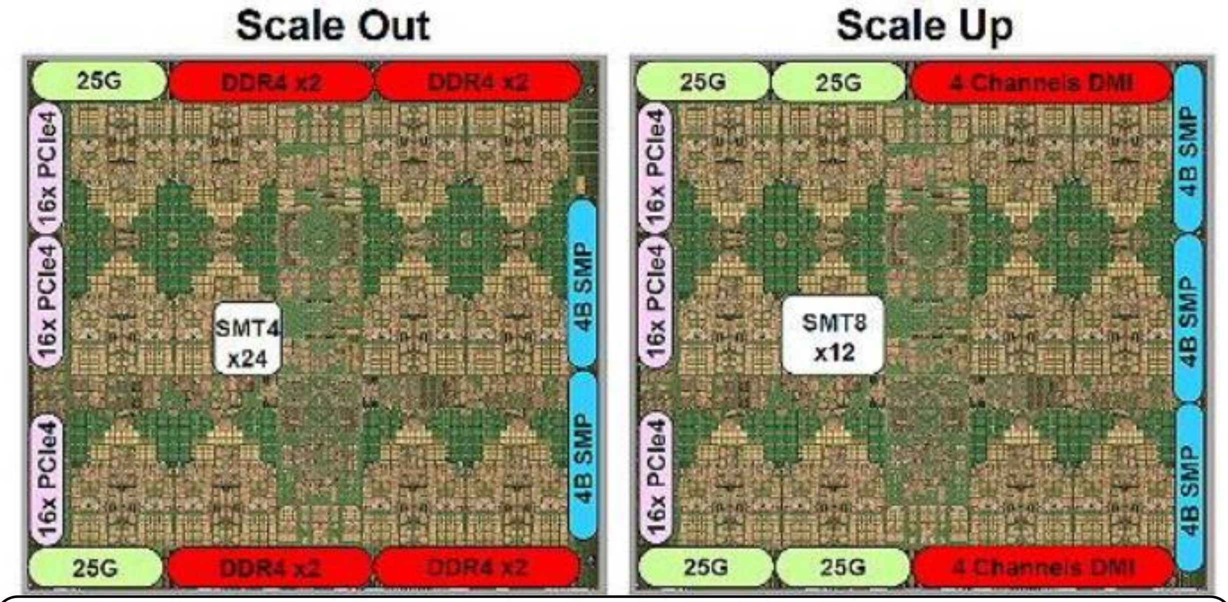
Comparison of IBM CAPI Implementations

TechU

Remove PCIe layers to reduce latency significantly

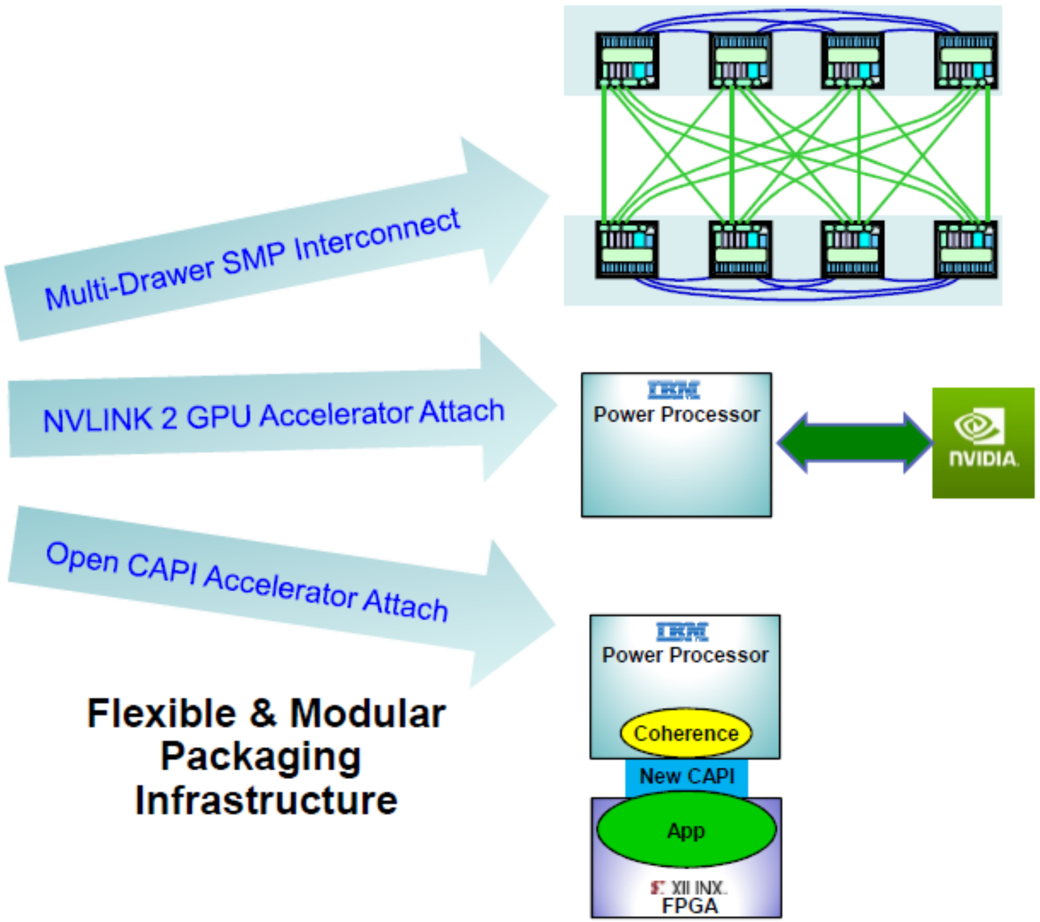
Feature	CAPI 1.0	CAPI 2.0	OpenCAPI 3.0	OpenCAPI 4.0
Processor Generation	POWER8	POWER9	POWER9	Future
CAPI Logic Placement	FPGA/ASIC	FPGA/ASIC	NA DL/TL on Host DLx/TLx on endpoint FPGA/ASIC	NA DL/TL on Host DLx/TLx on endpoint FPGA/ASIC
Interface Lanes per Instance Lane bit rate	PCIe Gen3 x8/x16 8 Gb/s	PCIe Gen4 2 x (Dual x8) 16 Gb/s	→ Direct 25G x8 25 Gb/s	Direct 25G+ x4, x8, x16, x32 25+ Gb/s
Address Translation on CPU	No	Yes	Yes	Yes
Native DMA from Endpoint Accelerator	No	Yes	Yes	Yes
Home Agent Memory on OpenCAPI Endpoint with Load/Store Access	No	No	Yes	Yes
Native Atomic Ops to Host Processor Memory from Accelerator	No	Yes	Yes	Yes
Accelerator -> HW Thread Wake-up	No	Yes	Yes	Yes
Low-latency small message push 128B Writes to Accelerator	MMIO 4/8B only	MMIO 4/8B only	MMIO 4/8B only	Yes
Host Memory Caching Function on Accelerator	Real Address Cache in PSL	Real Address Cache in PSL	No	Effective Address Cache in Accelerator

POWER9: CAPI2.0 and OpenCAPI3.0



48 lanes PCIe G4
Up to 32 lanes CAPI 2.0 & 1.0 enabled

48 lanes over 25Gbps Link
Up to 32 lanes for OpenCAPI 3.0
All can be used for NVLink

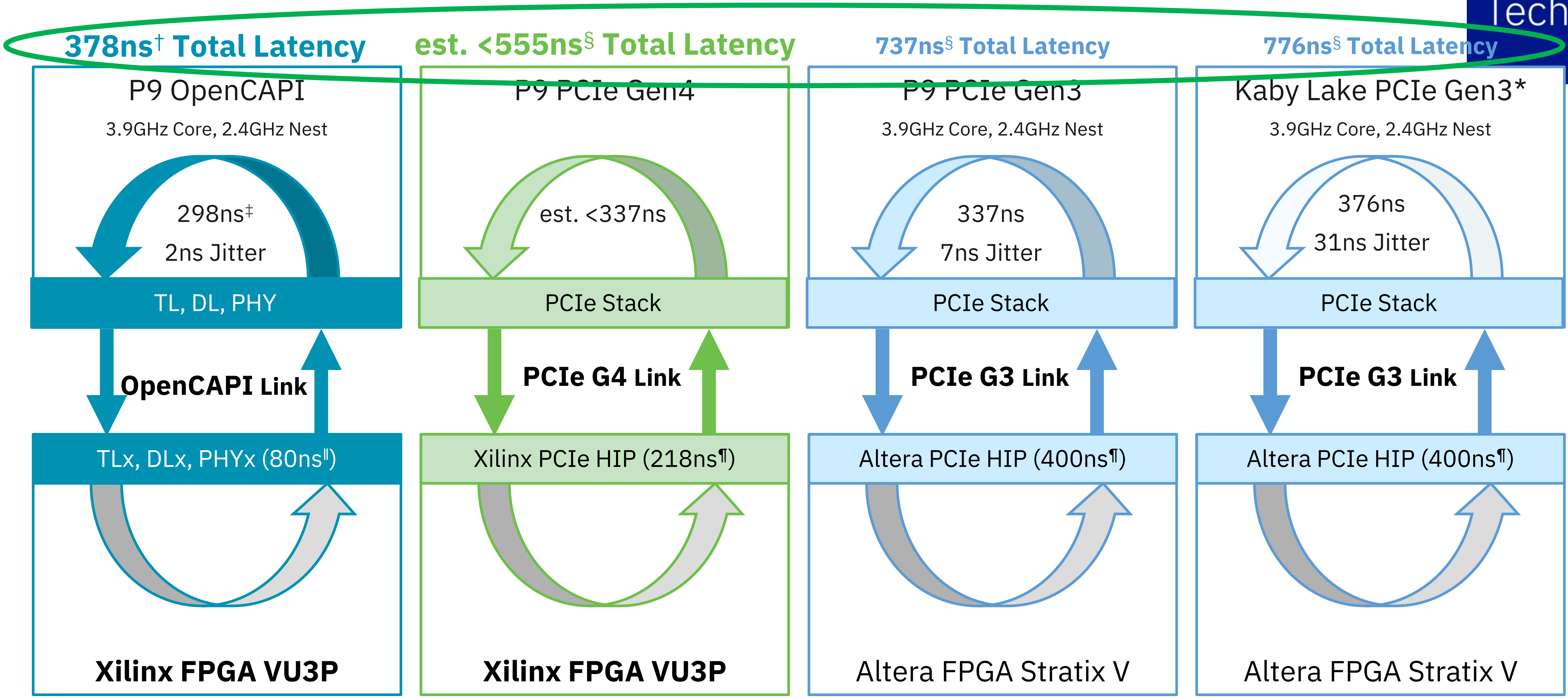

Multi-Drawer SMP Interconnect
NVLINK 2 GPU Accelerator Attach
IBM
Power Processor
NVIDIA
Open CAPI Accelerator Attach
IBM
Power Processor
Coherence
New CAPI
App
XILINX
FPGA

Flexible & Modular Packaging Infrastructure

- *CAPI2.0*
  - *2x PCIe Gen4 x16 lanes CAPI2.0 enabled*
  - *2x faster than PCIe Gen3 x16*

  ➔ *per FPGA card: 1 slot PCIeGen4x8 or PCIeGen3x16 – 16GBps*
- *OpenCAPI3.0*
  - *From 0 to 4x OpenCAPI Link (2x8 lanes) at 25Gbps depending on the P9 chip (i.e. ZZ has 1 OpenCAPI Link per socket)*
  - *100% Open Interface Architecture to connect to user-level accelerators, I/.O devices and advanced memories*

  ➔ *per FPGA card: 1 brick=8 lanes – 25GBps*



378ns[†] Total Latency

P9 OpenCAPI

3.9GHz Core, 2.4GHz Nest

298ns[‡]

2ns Jitter

TL, DL, PHY

OpenCAPI Link

TLx, DLx, PHYx (80ns[‖])

Xilinx FPGA VU3P

est. <555ns[§] Total Latency

P9 PCIe Gen4

est. <337ns

PCIe Stack

PCIe G4 Link

Xilinx PCIe HIP (218ns[¶])

Xilinx FPGA VU3P

737ns[§] Total Latency

P9 PCIe Gen3

3.9GHz Core, 2.4GHz Nest

337ns

7ns Jitter

PCIe Stack

PCIe G3 Link

Altera PCIe HIP (400ns[¶])

Altera FPGA Stratix V

776ns[§] Total Latency

Kaby Lake PCIe Gen3*

3.9GHz Core, 2.4GHz Nest

376ns

31ns Jitter

PCIe Stack

PCIe G3 Link

Altera PCIe HIP (400ns[¶])

Altera FPGA Stratix V

* Intel Core i7 7700 Quad-Core 3.6GHz (4.2GHz Turbo Boost)

[†] Derived from round-trip time minus simulated FPGA app time

[‡] Derived from round-trip time minus simulated FPGA app time and simulated FPGA TLx/DLx/PHYx time

[§] Derived from measured CPU turnaround time plus vendor provided HIP latency

[‖] Derived from simulation

[¶] Vendor provided latency statistic