CAPI SNAP framework, the tool for C/C++ programmers to accelerate by a 2 digit factor using FPGA technology

Bruno MESNET, Power CAPI Enablement
IBM Power Systems
CAPI
CAPI2.0 – OpenCAPI
SNAP framework
Use Cases
CAPI

Coherent Accelerator Processor Interface
IBM Power8 processor and its Coherent Accelerator Processor Interface

- System memory accessed directly by FPGA ➔ Coherent data
- Application in CPU + FPGA code much simpler and quicker ➔ Accelerator
- Specific logic in Power8 and FPGA + standard PCIe Gen3 ➔ Processor Interface

➔ application becomes QUICKER, SIMPLER, SAFER
An application **without CAPI**

- An application calls a device driver to utilize an Accelerator or any device outside the chip.
- The device driver performed a memory mapping operation.

3 versions of the data (not coherent).
1000s of instructions in the device driver.
An application with CAPI

- CPU unloaded since no device driver and accelerator doing the application work
- The FPGA shares memory with the cores

1 coherent version of the data. No device driver call/instructions.
Effect of CAPI hardware vs. PCI-E Device Driver

Typical I/O Model Flow: Total ~13μs for data prep

Flow with a CAPI Model: Total 0.36μs
Power 9
CAPI2.0 - OpenCAPI
POWER9 – Ideal for Acceleration

Extreme CPU/Accelerator Bandwidth

PCIe Gen3 x16
CPU 1x Accelerator
GPU

PCIe Gen4 x16
CPU 2x Accelerator
GPU

POWER8 with NVLink 1.0
5x
NVIDIA GPU

POWER9 with 25G Link
7-10x
NVIDIA GPU

Increased Performance / Features / Acceleration Opportunity

Seamless CPU/Accelerator Interaction
• Coherent memory sharing
• Enhanced virtual address translation
• Data interaction with reduced SW & HW overhead

Broader Application of Heterogeneous Compute
• Designed for efficient programming models
• Accelerate complex analytic / cognitive applications
<table>
<thead>
<tr>
<th>Feature</th>
<th>CAPI 1.0</th>
<th>CAPI 2.0</th>
<th>OpenCAPI 3.0</th>
<th>OpenCAPI 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Generation</td>
<td>POWER8</td>
<td>POWER9</td>
<td>POWER9</td>
<td>Future</td>
</tr>
<tr>
<td>CAPI Logic Placement</td>
<td>FPGA/ASIC</td>
<td>FPGA/ASIC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Interface</td>
<td>PCIe Gen3 x8/x16 8 Gb/s</td>
<td>PCIe Gen4 2 x (Dual x8) 16 Gb/s</td>
<td>Direct 25G x8 25 Gb/s</td>
<td>Direct 25G+ x4, x8, x16, x32 25+ Gb/s</td>
</tr>
<tr>
<td>Address Translation on CPU</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Native DMA from Endpoint</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Home Agent Memory on OpenCAPI Endpoint with Load/Store Access</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Native Atomic Ops to Host Processor Memory from Accelerator</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Accelerator -&gt; HW Thread Wake-up</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Low-latency small message push 128B Writes to Accelerator</td>
<td>MMIO 4/8B only</td>
<td>MMIO 4/8B only</td>
<td>MMIO 4/8B only</td>
<td>Yes</td>
</tr>
<tr>
<td>Host Memory Caching Function on Accelerator</td>
<td>Real Address Cache in PSL</td>
<td>Real Address Cache in PSL</td>
<td>No</td>
<td>Effective Address Cache in Accelerator</td>
</tr>
</tbody>
</table>

Remove PCIe layers to reduce latency significantly.
Recall CAPI technology connections

Proprietary hardware and designs to enable coherent acceleration

Operating system enablement
- little endian linux
- kernel driver (cxl)
- user library (libcxl)

Customer application and accelerator

- PSLSE models the red outlined area
  - Re-implements libcxl api calls
  - Models memory access
  - Provides hardware ports to afu
- Enables co-simulation of AFU and App
- Publicly available on github
OpenCAPI technology connections

Proprietary hardware and reference designs to enable coherent acceleration

Operating system enablement
- little endian linux
- reference kernel driver (ocxl)
- reference user library (libocxl)

Customer application and accelerator

- OCSE models the red outlined area
- OCSE enable AFU and App co-simulation IF reference libocxl and reference TLx/DLx are used.
- OCSE dependencies/assumptions
  - Fixed reference TLx/AFU interface
  - Fixed reference libocxl user API
- Will be available to consortium members
A Truly Heterogeneous Architecture Built on 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.
SNAP Framework
Storage Networking Analytics Processing
What if …
...you could easily program your FPGA using C/C++?
...and get 10x performance* in a few days?
...while operating on data flowing to the server?

<table>
<thead>
<tr>
<th></th>
<th>PCI-E FPGA</th>
<th>CAPI FPGA</th>
<th>CAPI SNAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Customer</td>
<td>Computer Engineers</td>
<td>Computer Engineers</td>
<td>Programmers</td>
</tr>
<tr>
<td>Development time</td>
<td>3-6 Months</td>
<td>3-6 Months</td>
<td>Days</td>
</tr>
<tr>
<td>Software Integration</td>
<td>PCI-E Device Driver</td>
<td>LibCXL</td>
<td>Simple API</td>
</tr>
<tr>
<td>Source Code</td>
<td>VHDL, Verilog, OpenCL</td>
<td>VHDL, Verilog, OpenCL</td>
<td>C/C++, Go</td>
</tr>
<tr>
<td>Coherency, Security</td>
<td>None</td>
<td>POWER + PSL</td>
<td>POWER + PSL</td>
</tr>
</tbody>
</table>

* Compared to running the same C/C++ in software
The CAPI – SNAP concept

- **Action X**
- **Action Y**
- **Action Z**

**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**

Best way to **offload/accelerate** a C/ C++ code with:
- Minimum change in code
- Quick porting
- Better performance than CPU
SNAP : CAPI Framework

Key:
- Base CAPI Components
- CAPI SNAP Components
- User host code and accelerator IP

- PSL/AXI bridge (simplified)
  - Host DMA
  - Control
  - Job Manager
  - Job Queue
  - MMIO

- Accelerated Action
  - Action 1: "VHDL"
  - Action 2: "C/C++"
  - Action 3: "Go"

- DRAM on-card
- NVMe
- Network (TBD)

CAPI SNAP Enabled Card

CAPI SNAP Components
- libcxl
- cxl
- SNAP library
- Job Queue
- Job Context
- Process A

CAPI SNAP Components
- libcxl
- cxl
- SNAP library
- Job Queue
- Job Context
- Process A

HDK: CAPI PSL

User host code and accelerator IP

CAPI

CAPI SNAP Enabled Card

Application

OpenPOWER™

XILINX

VIVADO
SNAP Enabled Card Details

Alpha-Data ADM-PCIE-KU3

- 3.5MB Block Ram on FPGA
- FPGA to Host Memory Access
  Latency to/from FPGA: 0.8us
  Bandwidth to FPGA: ~3.8GB/s reads and writes (CAPI limit)

- 8GB DDR3
  Latency to FPGA: 230ns

- Two 40Gb QSFP+ Ports
  Future Use: Currently no Bridge to SNAP

Choose this card for:
External IO, Offload and DRAM

Nallatech 250S

- 4GB DDR4 (on back of card)
  Latency to FPGA: 184ns Read / 105ns write

- Two 1TB NVMe sticks (1.92TB effective)
  Latency to FPGA: ~0.8us
  Bandwidth to FPGA: Read 1.8GB/s

Choose this card for:
2TB of on-card Flash
Let’s understand SNAP with a “hello world” example

Application on Server

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

HELLO WORLD. I love this new experience with SNAP

“Lower case” processing ➔ “software” action

“Upper case” processing ➔ “hardware” action

snap_helloworld –i /tmp/t1 –o /tmp/t2 -mode=fpga

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

/tmp/t1

/tmp/t2
SNAP 3 steps flow

1. **ISOLATION**
   - SNAP_CONFIG=CPU
   - `snap_helloworld -i /tmp/t1 -o /tmp/t2`
   - “Lower case” processing
     - “software” action
   - x86 server

2. **SIMULATION**
   - SNAP_CONFIG=FPGA
   - `snap_helloworld -i /tmp/t1 -o /tmp/t2`
   - “Upper case” processing
     - “hardware” action
   - x86 server
   - FPGA Card emulation

3. **EXECUTION**
   - SNAP_CONFIG=FPGA
   - `snap_helloworld -i /tmp/t1 -o /tmp/t2`
   - “Upper case” processing
     - “hardware” action
   - Power8 server

---

20
Use cases
Comparison of Ingest + Search

1TB of data to search

<table>
<thead>
<tr>
<th></th>
<th>POWER SW Only</th>
<th>POWER+Accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingest 1TB</td>
<td>20.8 seconds</td>
<td>5 Seconds into FPGA</td>
</tr>
<tr>
<td>Search</td>
<td>&lt;1us</td>
<td>&lt;1us</td>
</tr>
<tr>
<td>Results to core cache</td>
<td>0</td>
<td>&lt;400ns (across PCIE)</td>
</tr>
<tr>
<td>Total</td>
<td>20.8 seconds</td>
<td>5.0000014 seconds</td>
</tr>
</tbody>
</table>

POWER8 SW only

48GB/s

POWER + Accelerator

200GB/s

POWER8 Scale-Out Dual Chip Module

Cards with 4x100Gb/s EN™

PCIe Gen 3 x16

PCIe Gen 3 x16

SMP Interconnect

CAP I

SMP

I

PCIe

SMP

CAP
CAPI SNAP Paradigms

**Memory Transform**
- CPU
- IBM PSL
- FPGA
- Cache
- Actions

Example: Basic off-load

**Ingress, Egress or Bi-Directional Transform**
- CPU
- IBM PSL
- FPGA
- Cache
- Actions

Example: Compression, Crypto, HFT, Database operations

Classic SW Process
- CPU
- Actions
CAPI SNAP: just the best integrated together
Thanks

You need more?

ibm.biz/powercapi_snap
https://github.com/open-power/snap
bruno.mesnet@fr.ibm.com
capi@us.ibm.com
Backup slides
**Sponge benchmark on SNAP : a good story!**

**Purpose** of this benchmark is to:
1. validate how it is **quick** and **simple** to **port** a “computation only” **C coded** program into our SNAP framework
2. **measure** computation performance of this code
3. **compare** these performances with CPU

This porting was done by 1 software designer and 1 hardware designer with very basic Xilinx Vivado HLS tool knowledge.

**5 steps:**
1. Understand the environment
2. Port the code into Vivado HLS
3. Port the code into SNAP
4. Optimize the code
5. Performance results vs CPU
SNAP: CAPI Framework

Process C
- Slave Context
- SNAP library
- libcxl
- cxl

Process B
- Slave Context
- SNAP library
- libcxl
- cxl

Process A
- Slave Context
- SNAP library
- libcxl
- cxl

PSL/AXI bridge (simplified)
- Host DMA
- Control
- Job Manager
- Job Queue
- MMIO

CAPI

Accelerated Action
- Action 1: "Go"
- Action 2: "C/C++"
- Action 3: "VHDL"

AXI

DRAM on-card

NVMe

Network (TBD)

ADM-PCIE-KU3 XCKU060
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.

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

```c
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_keccakf (uint64_t st_in[25], uint64_t st_out[25]) {
    for (round = 0; round < KECCAKF_ROUNDS; round++)
        processing Theta + Rho Pi + Chi
}

KECCAKF_ROUNDS = 24 → 24 calls calling the algorithm process
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++) {
    }

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

Changes done to port code in Cuda

Changes done to port code in Cuda
.../...

// Rho Pi
st0 = st00;
st10 = ROTL64(st01, 1, 63);
st7 = ROTL64(st010, 3, 61);
st11 = ROTL64(st007, 6, 58);
st17 = ROTL64(st011, 10, 54);
st18 = ROTL64(st017, 15, 49);
st3 = ROTL64(st018, 21, 43);
st5 = ROTL64(st03, 29, 36);
st16 = ROTL64(st05, 36, 28);
st8 = ROTL64(st016, 45, 19);
st21 = ROTL64(st08, 55, 9);
st24 = ROTL64(st021, 62, 8);
st4 = ROTL64(st024, 14, 50);
st15 = ROTL64(st024, 27, 37);
st23 = ROTL64(st015, 55, 23);
st19 = ROTL64(st023, 62, 20);
st12 = ROTL64(st012, 8, 56);
st14 = ROTL64(st012, 41, 23);
st9 = ROTL64(st02, 39, 22);
st22 = ROTL64(st014, 56, 3);
st13 = ROTL64(st013, 61, 3);
st6 = ROTL64(st09, 8, 48);
st1 = ROTL64(st06, 44, 20);
.../...
Sha3_keccakf function

1st synthesis – with no code

2nd synthesis – adding #pragma HLS PIPELINE in main loop

⇒ Splitting st argument into st_in and st_out

INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 1, Depth: 2.
SNAP code architecture

sponge application call:

```c
rc = do_checksum(card_no, timeout, threads, 0, 0, 0, 0,
CHECKSUM_SPOONGE, t->nb_elmts, t->freq,
&checksum, &usec, &timer_ticks,
&nb_test_runs, &nb_round, fp);
```
SHA3 test_speed program in SNAP with no optimization

- Analyze 3 main numbers: clock frequency, logic LUT, iteration latency

**Main function (65536 iterations):**
- process_action function duration = 76.57 hours

**Test_speed function (100,000 iterations):**
- Test_speed function duration = 1,051,500,126x4ns = 4.2s

**Keccakf function (24 iterations):**
- iteration time = 438x4ns = 1.75µs
- Keccakf function duration = 10,513x4ns = 42µs
**SHA3 test_speed program in SNAP after optimizations**

**Main function (65536 iterations) / Factor 32:**
- `process_action` function duration = 43 seconds

**Test_speed function (100,000 iterations):**
- `Test_speed` function duration = 5,300,063x4ns = 21.2ms

**Keccakf function (24 iterations):**
- Iteration time = 2x4ns = 8ns
- Keccakf function duration = 51x4ns = 0.216µs

LUT x 25

6410 x faster

Latency / 219
FPGA area used by the design

16 test_speed functions in parallel:
HLS_SYN_CLOCK=2.827000,HLS_SYN_LAT=2713646082,
HLS_SYN_MEM=96,HLS_SYN_DSP=0,HLS_SYN_FF=74689,HLS_SYN_LUT=171,112

32 test_speed functions in parallel:
HLS_SYN_CLOCK=2.827000,HLS_SYN_MEM=192, HLS_SYN_FF=142929,HLS_SYN_LUT=337,640

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

⇒ Vivado HLS estimation is very pessimistic and Vivado doing a very good optimization of resources!
**Offload Method:**

**SHA3 speed_test benchmark:** FPGA is >35x faster than CPU

<table>
<thead>
<tr>
<th>NB_ROUNDS</th>
<th>NB_TEST_RUNS</th>
<th>nb_elmts</th>
<th>freq</th>
<th>test_speed calls (msec)</th>
<th>CPU (antipode) freq</th>
<th>System P freq</th>
<th>FPGA Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>65,536</td>
<td>32</td>
<td>65,536</td>
<td>3,200,000</td>
<td>22</td>
<td>1,260</td>
<td>57</td>
</tr>
<tr>
<td>100,000</td>
<td>65,536</td>
<td>128</td>
<td>65,536</td>
<td>12,800,000</td>
<td>85</td>
<td>3,460</td>
<td>41</td>
</tr>
<tr>
<td>100,000</td>
<td>65,536</td>
<td>4,096</td>
<td>65,536</td>
<td>409,600,000</td>
<td>2,715</td>
<td>95,975</td>
<td>35</td>
</tr>
<tr>
<td>100,000</td>
<td>65,536</td>
<td>8,192</td>
<td>65,536</td>
<td>819,200,000</td>
<td>5,429</td>
<td>190,347</td>
<td>35</td>
</tr>
<tr>
<td>100,000</td>
<td>65,536</td>
<td>32,767</td>
<td>65,536</td>
<td>3,276,700,000</td>
<td>21,709</td>
<td>754,198</td>
<td>35</td>
</tr>
<tr>
<td>100,000</td>
<td>65,536</td>
<td>65,536</td>
<td>65,536</td>
<td>6,553,600,000</td>
<td>43,418</td>
<td>1,505,790</td>
<td>35</td>
</tr>
</tbody>
</table>

![SHA3 TEST_SPEED - EXECUTION TIME (MSEC)](image)

“The lower the better”
Conclusion

This benchmark validates several points:
- Combining CAPI SNAP and Vivado HLS:
  - gives real **good performances** with very little changes needed to the initial C code program
  - gives a very **simple way** to call the **action** on FPGA from the **application** on server
- Code synthesized in a mid-range FPGA is able to beat CPU by a factor **$x_{35}$** on performance side!
- The complete porting + optimizing was done by 2 engineers within..... **11 days**
  - Porting code to HLS: **2 days**
  - Integrate code into SNAP: **2 days**
  - Optimize code to beat CPU: **7 days**
Performances measured

**Write access** (streaming mode: `write_na`)
- Write Bandwidth when data is pushed to system memory: **3.88 GB/s**
- Average **Write Latency** from PSL request to response: **838ns**.

**Read access** (streaming mode: `read_cl_na`)
- Read Bandwidth when data resides in system memory: **3.42 GB/s**
- Average **Read Latency** from PSL request to response: **864ns**.

Read / Write access in cache mode ➔ **Latency** for Read or Writes that hit in the PSL Cache: **120ns**.