POWER CAPI+SNAP+FPGA, the powerful combination to accelerate routines explained through use cases

Bruno MESNET, CAPI / OpenCAPI enablement
IBM Systems
Offload?....CAPI
Use cases
Coding?.....SNAP
SHA3 example......x35

Summary
Some SNAP Use cases

Video / Analytics
- Smart Video surveillance from multiple videos feed
- 3D video stream from multi-angles videos streams
- Image search / Object tracking / Scene recreation
- Multi-jpeg compression

Bank / Finance
- Risk analysis / Faster trading: Monte Carlo libraries
- Credit card fraud detection
- Block chain acceleration

Algorithm acceleration
- Compression on network path or storage
- Encryption on the fly to various memory types
- String match

Machine Learning / Deep learning
- Machine learning inference
- Accelerate frequently used ML / DL algorithm
Thousands of tiny CPU using high parallelization ➔ compute intensive application

Logic + IOs are customized exactly for the application’s needs. ➔ Very low and predictable latency applications

2 options

Fast

Real Time

GPU

FPGA
CAPI SNAP Paradigms

Memory Transform

Example: Basic off-load

Ingress, Egress or Bi-Directional Transform

Example: Compression, Crypto, HFT, Database operations

Classic SW Process

CPU Actions
The CAPI – SNAP concept

- FPGA becomes a peer of the CPU ➔ Action directly accesses host memory
- Manage server threads and actions
  Manage access to IOs (memory, network) ➔ Action easily accesses resources
- Gives on-demand compute capabilities
  Gives direct IOs access (storage, network) ➔ Action directly accesses external resources
- 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
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

1TB to ingest through a 4x100Gb/s card takes 20.8secs!
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
Adding a « CAPI-enabled » FPGA card

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

- 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

CAPI1.0
PCIeGen3x8 @8Gb/s
~4GB/s measured
~800ns latency

CAPI2.0
PCIeGen4x8 @16Gb/s
~14 GB/s measured
Est. <555ns total latency

OpenCAPI3.0
BlueLink 25Gb/s 8 lanes
~22GB/s measured
378ns total latency

"Total latency" test on OpenCAPI3.0:
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.

P9 Server

October 4th, 2018
Use cases in development
Some Use cases in development

**USECASE#07: H.265/HEVC ENCODER**

**The State of the Art**
- Bandwidth required for a 4K broadcast: 
  - H.264: AVC takes 13Mbps
  - H.265/HEVC takes 13Mbps
- Avg. Internet speed in the US is 18Mbps

**The Problem**
- H.265 requires more than a factor more computing complexity
- Real-time software encoding becomes an issue

**The Solution's Implementation**
- Implement the real-time encoding in a CAPI enabled FPGA
  - Big FPGA have the capability to parallelize processing, and support 1080p real-time HEVC encoding with less than 100MHz clock frequency
  - CAPI low latency interface ease the flow to the host memory

**The Key of SUCCESS**
- FPGA parallel compute capabilities
- CAPI low latency + SNAP quick development

**Some Use cases in development**

**USECASE#02: CORAL-2 BENCHMARKS ON CAPI+1**

**The State of the Art**
- The Coral-2 project is part of the Department of Energy (DoE)'s vision to meet the goal of exaccale computing. Coral-2 and/or apex open-source benchmarks are real-world problems: weather, ocean-currents, nuclear energy,...
- This pool of benchmarks include a variety of applications ranging from Machine/Deep Learning to large scientific modeling.

**The Problem**
- Coral-2’s impact is expected to be 5x the current Summit/Coral-1 project

**The Solution's Implementation**
- Explore data structures and communication schemes to speed up fluid dynamics algorithms by leveraging acceleration capabilities of CAPI
  - given a benchmark, split the workload into the FPGA and GPU engines (thus the "+ 1" in the title)
- proof-of-concept apps will be available to external clients

**The Key of SUCCESS**
- FPGA compute capabilities + CAPI low latency

**Important factors of success**
- SNAP quick development

**USECASE#06: REGEX MATCHING**

**The State of the Art**
- Regex used widely in network applications such as network security, bandwidth management, and government surveillance

**The Problem**
- The throughput of regular expression matching can directly influence the network link speed.

**The Solution's Implementation**
- Multi pipelines for multi packets Regex matching
  - Most Regex operators supported + Non-deterministic Finite Automaton(NFA) unit
  - Configurable number of multi Regex patterns in parallel
  - Throughput from 2GB/s to expected 16GB/s on OpenCAPI

**The Key of SUCCESS**
- FPGA parallel compute capabilities
- CAPI low latency + SNAP quick development

**USECASE#19: CAPI INLINE COMPRESSION**

**The State of the Art**
- Add compression to the buffer data
  - Keep buffer chip to keep high speed CPU interface
  - No change to data flow
  - Buffer chip
  - Compresse data when it is written
  - Organizes compressed pages
  - Decompress data when it is read

**The Problem**
- We need more DRAM for less cost and we need it to have little or no overhead

**The Prototype's Implementation**
- Use CAPI to serve up a buffer the host can "read" and "write" into a host address
  - Data will be compressed or decompressed when written or read into/from the the buffer
  - The read and write commands will allow byte offsets and 1-128 byte lengths

**The Key of SUCCESS**
- CAPI/ OpenCAPI Low latency

**Important factors of success**
- Compression in FPGA + FPGA Board DRAM + SNAP ease of programming
Some Use cases in development

**USECASE#03: SMART SURVEILLANCE FOR FALL DETECTION**

*The State of the Art*
- Current methods of fall detection:
  - Medical Alert Devices
  - Life Alert
  - Wearable sensors
  - Accelerometers, gyroscopes

*The Problem*
- How to automatically survey video streams for significant events?

*The Solution’s Implementation*
- A deep convolutional neural network analyzes one image from the video stream at a time.
- The network will output the probability of 4 states (standing, falling, fallen, not moving).
- If a sequence of standing -> falling -> fallen -> not moving is detected, the system declares that the person has fallen and needs assistance.

Importantly:
- FPGA high computing + direct brand video inputs

**USECASE#05: NVIDIA ON FPGA: NVIDIA LEARNING ACCELERATION**

*The State of the Art*
- Analytics on digital data of images, videos, and speech from sources such as social media and the Internet-of-Things use deep learning algorithms to make data understandable and actionable.

*The Problem*
- The inference on a GPU is not an option for Caffe DNN due to NVDIA very specific programming needs required.

*The Solution’s Implementation*
- AccDNN automatically converts the Caffe trained deep neural network to the FPGA RTL level implementation.
- User provides their trained Caffe model + no programming effort.
- Unifies APIs to the users for their recognition task.

**USECASE#20: ACCDNN: ACCELERATE DEEP NEURAL NETWORK**

*The State of the Art*
- Caffe deep neural network is trained on CPUs and GPUs, but inference is complex to port.

**USECASE#09: FACE RECOGNITION**

*The State of the Art*
- Finding persons in photos is a compute-intensive job, which is currently done by CPUs or GPU acceleration

*The Problem*
- Finding persons in emergency camps and identify them.
- Children can’t find their relatives or communicate and vice versa.
- Distributed camps big distances, chaos everywhere.

*The Solution’s Implementation*
- We will find Josef yer you
- Pre-trained Neural Network recognizes equality of persons in a given pair of pictures
  - Emergency Camp streams pictures of rescued victims into our cloud.
  - Relatives send pictures of searched persons, to check if they are safe.
  - For each sent picture, we check each registered victim picture with our SNAP CAPI solution.
  - With such a match we also get information about the identity of prior unidentified victims (e.g., children)

Important factors of success:
- FPGA parallel compute capabilities

**USECASE#14: ACCELERATE FREQUENTLY USED ML/DL ALGORITHMS**

*The State of the Art*
- Current ML/DL algorithms are working only on CPU and GPU

*The Problem*
- How to accelerate frequently used algorithms in ML/DL solution?

*The Solution’s Implementation*
- Identify algorithms suitable for acceleration.
- Each accelerated algorithm becomes an application.
- Multiple applications can be built and used in products such as PowerAI.
- One such algorithm is PICA = Principal Component Analysis.
- Port the PICA algorithm to CAPI connected FPGA.
- Optimize FPGA implementation.

Important factors of success:
- FPGA parallel compute capabilities + CAPI low latency

**USECASE#10: SNAP LOW END**

*The Problem*
- SNAP need of integration

*The Solution’s Implementation*
- How to improve computing capabilities + SNAP quick development

Important features of success:
- FPGA compute capabilities + SNAP quick development

**USECASE#12: SNAP NOS OF INTEGRATION**

*The Problem*
- FPGA low consumption

*The Solution’s Implementation*
- How to automatically survey video streams for significant events?

Important factors of success:
- FPGA high computing + direct brand video inputs

**USECASE#07: SNAP LOW END**

*The Problem*
- Building an open-source SNAP library

*The Solution’s Implementation*
- NVIDIA on FPGA: NVIDIA learning acceleration

Important features of success:
- FPGA compute capabilities + SNAP quick development
Coding: RTL? C/C++? SNAP?
FPGA development: Choice1

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

Big developing efforts
Extreme performance targeted, full control
Programming based on libcxl and PSL interface
FPGA development: Choice2 (Recommended)

- **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**

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

 October 4th, 2018

Fast 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)

Power™ Coherent Acceleration Processor Interface (CAPI)
2 different modes

The Job-Queue Mode
SERIAL MODE

FPGA-action executes a job and returns after completion

For multiple processes N using multiple FPGA-actions M virtually in parallel controlled by built in job-manager

⇒ FPGA acceleration within a Cloud, e.g. using Docker virtualization

The Fixed-Action Mode
PARALLEL MODE

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

For N processes using N FPGA-actions in parallel
⇒ FPGA-action must permanently run
⇒ Networking

#actions depend on size of logic and FPGA
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

 SNAP

“Lower case” processing ➔ “software” action

“Upper case” processing ➔ “hardware” action

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

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

HELLO WORLD. I love this new experience with SNAP

HELLO WORLD. I love this new experience with SNAP
**SNAP solution Flow: prepare the data** (hls_helloworld example)

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

   ```c
   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 as input)
   - Allocate memory area (64 Bytes aligned)

   ```c
   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;`
   - `type_out = SNAP_ADDRTYPE_HOST_DRAM;`
   - `addr_out = (unsigned long) obuff;`
   - Assign the structure `mjob` containing all parameters we just filled to the job `cjob`

   ```c
   snap_addr_set(&mjob->in, addr_in, size_in, type_in,
                 SNAP_ADDRFLAG_ADDR | SNAP_ADDRFLAG_SRC);
   snap_addr_set(&mjob->out, addr_out, size_out, type_out,
                 SNAP_ADDRFLAG_ADDR | SNAP_ADDRFLAG_DST | SNAP_ADDRFLAG_END);
   snap_job_set(cjob, mjob, sizeof(*mjob), NULL, 0);
   ```

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

   ```c
   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**

   ```c
   action = snap_attach_action(card,
                               HELLO_WORLD_ACTION_TYPE, action_irq, timeout);
   ```
SNAP solution Flow: call + process the action (hls_helloworld example)

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

Get and align the input_data_address, input_data_address and size to access (MMIO)

Read data from input_data address directly in host memory server (din_gmem)

Process the data (uppercase conversion)

Write data to output_data address directly in host memory server (dout_gmem)

Fill the return code

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

---

**MMIO registers**

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

---

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 direct to memory
**SNAP solution Flow : free the action** *(hls_helloworld example)*

### 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

   \[
   \text{rc = } \text{file_write}(\text{output}, \text{addr\_out}, \text{size});
   \]

8. **Detach action**
   - Disallocate the card
   - Free the dynamic allocation of buffers

   \[
   \text{snap\_detach\_action(action);}
   \]
   \[
   \text{snap\_card\_free(card);}
   \]
   \[
   \text{__free(obuf);}
   \]
   \[
   \text{__free(ibuff);}
   \]

**Host System memory**

- **Data memory area**
  - `@addr\_in` --- Input Text
  - `@addr\_out` --- Output text

**C/ C++ code used in Application**
A SIMPLE 3 STEPS PROCESS

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

"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

No specific test bench required
Use your actual application

FPGA Card emulation with Power Server IBM’s simulation engine
PSLSE Instead !

Power™ Coherent Acceleration Processor Interface (CAPI)
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.

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

```c
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;
}
```

```c
void sha3_keccakf(uint64_t st_in[25], uint64_t st_out[25])
{
    for (round = 0; round < KECCAKE_ROUNDS; round++)
        processing Theta + Rho Pi + Chi
}
```

Math function

Parallel loops

Recursive loops

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

KECCAKE_ROUNDS = 24 → 24 calls calling the algorithm process

NB_TEST_RUNS = 65,536

NB_ROUNDS = 100,000

October 4th, 2018
The code snippet describes a function for the Keccakf cryptographic hash function. It includes optimizations and changes for HLS (High-Level Synthesis) to improve performance and efficiency. Key changes include:

1. Splitting input and output ports.
2. Adding HLS PIPELINE instructions to enhance parallelism.

The function is parameterized with `st_in` and `st_out` as input and output arrays, respectively, with both being arrays of `uint64_t` of length `25`. The function performs multiple rounds of transformations, including:

- **Theta**
- **Rho Pi**
- **Chi**
- **Iota**

The changes done for HLS are highlighted, focusing on the process of optimizing the function for hardware synthesis. The code snippet also includes comments for future comparison with CUDA implementations to provide context for optimization efforts.

For comparison, CUDA-specific optimizations are mentioned, showing how the function might be further tuned for GPU acceleration.
16 test_speed functions in parallel:

32 test_speed functions in parallel:

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

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

<table>
<thead>
<tr>
<th>test_speed calls (msec)</th>
<th>CPU (antipode) 20 cores - 160 threads</th>
<th>CPU (antipode) 20 cores - 160 threads</th>
<th>FPGA Speedup vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>slices/32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>21</td>
<td>21</td>
<td>149,575</td>
</tr>
<tr>
<td>200,000</td>
<td>21</td>
<td>669</td>
<td>295,786</td>
</tr>
<tr>
<td>400,000</td>
<td>21</td>
<td>819</td>
<td>488,441</td>
</tr>
<tr>
<td>800,000</td>
<td>21</td>
<td>925</td>
<td>865,289</td>
</tr>
<tr>
<td>1,600,000</td>
<td>21</td>
<td>1,018</td>
<td>1,572,084</td>
</tr>
<tr>
<td>3,200,000</td>
<td>22</td>
<td>1,260</td>
<td>2,539,064</td>
</tr>
<tr>
<td>12,800,000</td>
<td>85</td>
<td>3,460</td>
<td>3,699,211</td>
</tr>
<tr>
<td>409,600,000</td>
<td>2,715</td>
<td>95,975</td>
<td>4,267,759</td>
</tr>
<tr>
<td>819,200,000</td>
<td>5,429</td>
<td>190,347</td>
<td>4,303,717</td>
</tr>
<tr>
<td>3,276,700,000</td>
<td>21,709</td>
<td>754,198</td>
<td>4,344,618</td>
</tr>
<tr>
<td>6,553,600,000</td>
<td>43,418</td>
<td>1,505,790</td>
<td>4,352,266</td>
</tr>
</tbody>
</table>

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

“The lower the better”
Summarizing
Power™ Coherent Acceleration Processor Interface (CAPI)

3 good reasons why to try

No experience needed
Just an 1 hour or 2

For everyone
Just know C/C++

No investment
Just 36¢ per hour
3 steps how to implement

Isolate your function

Simulate your function

Execute your function
3 steps to discover and adopt

- Log to jarvice
- Experience the flow
- Boost YOUR function

~1.5 hours
You need to:
- Know more about accelerators?
- See a live demonstration?
- Do a benchmark?
- Get answers to your questions?

Contact us
alexandre.castellane@fr.ibm.com
or bruno.mesnet@fr.ibm.com

More on CAPI and SNAP?
ibm.biz/powercap SNAP
https://github.com/open-power/snap

Visit the IBM Client Center in Montpellier
http://www.ibm.com/ibm/clientcenter/montpellier