POWER CAPI + SNAP + FPGA =

The powerful combination to accelerate routines by factors of more than 30!

OBSPM
Typical Use cases...

Detection

PlantNet
Observation et identification interactive des plantes

Classification

Counting

Cars detected: 0

Lane counter: 0 0 0
“Classic” solution

Import your data
Label what you want to recognize
Choose your algorithm
Choose your model type
Training per iteration
Deploy your model

Use your model

Test it:
Power AI Inference Engine

Our Goal: **PowerAI Inference Engine** is toolset to provide all these capabilities for inference **automatically**

- Analyze DNN model and predict resource requirement, performance
- Convert and generate code packages based on different backend implementation for different HW architecture
Different Use cases... need ....different accelerators

**H.265/HEVC Encoder**

Bandwidth required for a 4K broadcast:
- H.264/AVC takes 32Mbps
- Avg. internet speed in the US is 18Mbps
- H.265/HEVC takes 15Mbps

*The Problem*

H.265 requires more than 4 times more computing complexity → real-time software encoding becomes an issue

**In-line compression for memory**

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

**Key generation for encryption (SHA3...)**

$\times 35$
Thousands of tiny CPUs using high parallelization ➔ compute intensive application

Logic + IOs are customized exactly for the application’s needs. ➔ Very low and predictable latency applications
3 typical cases when you should consider using external accelerators

- Boost a function
- Offload your CPU
- Free network resources
Hardware Acceleration paradigms

**Memory Transform**

- **Classic SW Process**
  - CPU
  - Actions

- **Ingress, Egress or Bi-Directional Transform**
  - CPU
  - Actions

**Example**:
- Basic off-load
- Compression, Crypto, HFT, Database operations

---

**Hardware Acceleration paradigms**

**CPU**
**SNAP**
**FPGA**

---

**October 25th 2018**

*Power™ Coherent Acceleration Processor Interface (CAPI)*
Some FPGA / 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
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

- 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 the data located??
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
3. Adding a « CAPI-enabled » FPGA card

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

- **CAPI** = 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 (round trip)

**P9 / OpenCAPI3.0**  
OpenCAPI link 25Gb/s 8 lanes  
~22GB/s measured  
378ns total latency (round trip)

"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.
A Truly Heterogeneous Architecture Built on OpenCAPI

1. **Accelerators**: The performance, virtual addressing, and coherence capabilities allow FPGA and ARM 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 overheads 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.

---

**OpenCAPI 3.0-4.0**

**OpenCAPI 3.1**

**CAPI 1.0-2.0**
## Comparison of IBM CAPI Implementations

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lanes per Instance</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>Lane bit rate</td>
<td></td>
<td></td>
<td></td>
<td></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 Accelerator</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*
**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†)

**Altera FPGA**
Stratix V

**737ns $ Total Latency**

**P9 PCIe Gen3**

- 337ns
  - 7ns Jitter

PCiE Stack

**PCIe G3 Link**

- Altera PCIe HIP (400ns†)

**Altera FPGA**
Stratix V

**776ns $ Total Latency**

**Kaby Lake PCIe Gen3**

- 378ns
  - 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
¶ Vendor provided latency statistic

October 25th 2018
Coding: RTL? C/C++? SNAP?
FPGA development: Choice 1

- 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 CAPI 1.0, 2.0 to OpenCAPI
  - **Open-source**

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

**S**torage, **N**etworking, **A**nalytics **P**rogramming framework
**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

- **FPGA** gives on-demand compute capabilities
- Gives direct IO 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
**SNAP framework**

- **Software Program**
  - Process C
  - Process B
  - Process A
  - SNAP library
  - libcxl
  - cxl

- **Application on Host**

- **Acceleration on FPGA**
  - PSL/AXI bridge
  - Host DMA
  - Control
  - Job Manager
  - Job Queue
  - MMIO

- **Hardware Action**
  - C/C++ or RTL
  - AXI
  - AXI lite
  - AXI on-card
  - NVMe
  - Network (TBD)

**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](https://www.arm.com/products/system-ip/amba-specifications))

October 25th 2018

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

The Job-Queue Mode
SERIAL MODE

FPGA-action executes a job and returns after completion

Software Program  \arrow{C/C++ function}  Hardware Action

The Fixed-Action Mode
PARALLEL MODE

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

Software Program  \arrow{C/C++ function}  Hardware Action
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
```

```
/tmp/t2
```

```
hello world. I love this new experience with SNAP
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
A SIMPLE 3 STEPS PROCESS

1. **ISOLATION**

   **SNAP_CONFIG=CPU**
   
   SNAP_CONFIG=CPU snap_helloworld -i /tmp/t1 -o /tmp/t2

   Application
   Action

   "Lower case" processing ➔ "software" action

   x86 server
   command: make

2. **SIMULATION**

   **SNAP_CONFIG=FPGA**
   
   SNAP_CONFIG=FPGA snap_helloworld -i /tmp/t1 -o /tmp/t2

   Application
   Action

   "Upper case" processing ➔ "hardware" action

   x86 server
   command: make sim

3. **EXECUTION**

   **SNAP_CONFIG=FPGA**
   
   SNAP_CONFIG=FPGA snap_helloworld -i /tmp/t1 -o /tmp/t2

   Application
   Action

   "Upper case" processing ➔ "hardware" action

   POWER8/9 server
   command: make image

---

No specific test bench required. Use your actual application.
### SNAP solution Flow: *prepare the data* (hls_helloworld example)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Code used in Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fill input data into host server memory</td>
<td>- size = file_size(input); &lt;br&gt;- <code>addr_in = snap_malloc(size)</code> &lt;br&gt;- <code>rc = file_read(input, addr_in, size)</code></td>
</tr>
<tr>
<td>2</td>
<td>Prepare host server memory to store the results</td>
<td>- <code>addr_out = snap_malloc(size)</code></td>
</tr>
<tr>
<td>3</td>
<td>Prepare parameters to be written in MMIO registers</td>
<td>- <code>type_in = SNAP_ADDRTYPE_HOST_DRAM;</code> &lt;br&gt;- <code>addr_in = (unsigned long) ibuff;</code> &lt;br&gt;- <code>type_out = SNAP_ADDRTYPE_HOST_DRAM;</code> &lt;br&gt;- <code>addr_out = (unsigned long) obuff;</code> &lt;br&gt;- `snap_addr_set(&amp;mjob-&gt;in, addr_in, size_in, type_in, SNAP_ADDRFLAG_ADDR</td>
</tr>
<tr>
<td>4</td>
<td>Allocate the card that will be used</td>
<td>- <code>card = snap_card_alloc_dev(device, SNAP_VENDOR_ID_IBM, SNAP_DEVICE_ID_SNAP);</code></td>
</tr>
<tr>
<td>5</td>
<td>Allocate the action that will be used on the allocated card</td>
<td>- <code>action = snap_attach_action(card, HELLOWORLD_ACTION_TYPE, action_irq, timeout);</code></td>
</tr>
</tbody>
</table>
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**

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)

**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 to memory.

C/ C++ code used in Application

```
6
```

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

**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
```

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)
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
   
   ```c
   rc = __file_write(output, addr_out, size);
   ```

8. Detach action
   - Disallocate the card
   - Free the dynamic allocation of buffers
   
   ```c
   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
SNAP Ecosystem

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

Software — Hardware — External cards
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.

The SHA3 test_speed program structure:

```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;
}
```

**Code used was downloaded (with author’s authorization) from:**

https://github.com/mjosaarinen/tiny_sha3

**NB_TEST_RUNS** = 65,536

**Parallel loops**

**Recursive loops**

**Math function**

```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
}
```

**KECCAKE_ROUNDS** = 24 → 24 calls calling the algorithm process

**NB_ROUNDS** = 100,000
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++)
      st[i] = st[i] + t;

  // Rho Pi
  t = st[1];
  for (i = 0; i < 24; i++)
      st[i] = keccakf_pi(st[i]);

  for (i = 0; i < 25; j += 5)
      st[i] ^= t;

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

  // Iota
  st[0] ^= keccakf_iota(st[0]);

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

Keccakf function changes
FYI for comparison: changes done to port this code to CUDA
../...

// Rho Pi
st0 = st00;
st10 = ROTL64(st01, 1, 63);
st7 = ROTL64(st010, 3, 61);
st11 = ROTL64(st07, 6, 58);
st17 = ROTL64(st011, 10, 54);
st18 = ROTL64(st017, 15, 49);
st3 = ROTL64(st018, 21, 43);
st5 = ROTL64(st03, 28, 36);
st16 = ROTL64(st05, 36, 28);
st8 = ROTL64(st016, 45, 19);
st21 = ROTL64(st08, 55, 9);
st24 = ROTL64(st021, 2, 62);
st4 = ROTL64(st024, 14, 50);
st15 = ROTL64(st04, 27, 37);
st23 = ROTL64(st015, 41, 23);
st19 = ROTL64(st023, 56, 8);
st13 = ROTL64(st019, 8, 56);
st12 = ROTL64(st013, 25, 39);
st5 = ROTL64(st012, 43, 21);
st20 = ROTL64(st02, 62, 2);
st14 = ROTL64(st020, 18, 46);
st22 = ROTL64(st014, 39, 25);
st9 = ROTL64(st022, 61, 3);
st6 = ROTL64(st09, 20, 44);
st1 = ROTL64(st06, 44, 20);
FPGA area used by the design

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:**

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

**Plot:** SHA3 TEST_SPEED - EXECUTION TIME (MSEC)

**Note:**

- The lower the better

**Link:**

[https://github.com/open-power/snap/tree/master/actions/hls_sponge](https://github.com/open-power/snap/tree/master/actions/hls_sponge)
Summarizing
No experience needed
Just an 1 hour or 2

For everyone
Just know C/C++

No investment to evaluate
Just 36¢ per hour

3 good reasons
why to try
3 steps to implement

1. Isolate your function
2. Simulate your function
3. Execute your function
3 steps to discover and adopt

Boost YOUR function

Log to jarvice

Experience the flow

~1.5 hours

Experience the flow
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

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

More on CAPI and SNAP?
ibm.biz/powercapi_snap
https://github.com/open-power/snap
Use Cases
• **Example 1:** My data are scattered in memory. This harms performances with a traditional DMA move. *Can CAPI help?*

• **Example 2:** I’m reluctant to use a function in a FPGA since it is often complex to call. *Is there a friendly way to do that?*

• **Example 3:** I have a CPU-intensive processing function. Can FPGA really be **quicker and more power efficient** than a CPU or a GPU?

➔ *All FPGA codes have been written in C or C++ and used SNAP and HLS*

➔ *They are all available in git*
Scatter gather memory access

I want to process a function in a FPGA. Does CAPI help reduce the data moving latency when the data blocks are scattered everywhere in memory?

soon in https://github.com/open-power/snap
Scenario: 2MB data scattered in host memory are processed in a FPGA.

Host memory 64b Virtual address

1024 of 2kB blocks randomly scattered in memory

AddrSet (1k*8 bytes) stores their effective addresses

1 transaction of 8kB for AddrSet from host memory to FPGA

1024 transactions of 2kB from Host memory to FPGA. Directly reads required data at random address.

Traditional DMA way

Gathering data (SW memcpy)
Scatter gather memory access

Results: (Power9 – CAPI2.0 – 2.154GHz, 512MB RAM) (FPGA card: FW609 + S241: VU9P Gen3x16) (SNAP)

<table>
<thead>
<tr>
<th>How scattered</th>
<th>SW gather</th>
<th>DMA</th>
<th>Sum</th>
<th>Verilog</th>
<th>HLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RK1</td>
<td>309.3</td>
<td>183.5</td>
<td>492.8</td>
<td>171.65</td>
<td>173.3</td>
</tr>
<tr>
<td>RK4</td>
<td>319.05</td>
<td>186.05</td>
<td>505.1</td>
<td>180.9</td>
<td>180.9</td>
</tr>
<tr>
<td>RK16</td>
<td>305.1</td>
<td>185.7</td>
<td>490.8</td>
<td>184.6</td>
<td>186.95</td>
</tr>
<tr>
<td>RK64</td>
<td>320.6</td>
<td>186.85</td>
<td>507.45</td>
<td>186.3</td>
<td>187.5</td>
</tr>
<tr>
<td>RK256</td>
<td>318.3</td>
<td>185.65</td>
<td>503.95</td>
<td>218.55</td>
<td>215.35</td>
</tr>
<tr>
<td>RK1024</td>
<td>333</td>
<td>189.15</td>
<td>522.15</td>
<td>236.85</td>
<td>224.95</td>
</tr>
<tr>
<td>RK4096</td>
<td>324.4</td>
<td>189.35</td>
<td>513.75</td>
<td>241.15</td>
<td>225.55</td>
</tr>
<tr>
<td>RK16384</td>
<td>307.4</td>
<td>185.75</td>
<td>493.15</td>
<td>240.9</td>
<td>224.9</td>
</tr>
</tbody>
</table>

More scattered

R = random
K is the dispersion factor of the blocks
Allocate 2MB in a K * 2MB memory area
→ K=1 : all blocks contiguous
→ K=2: 2MB allocated amongst 4MB
→ K=4: 2MB allocated amongst 8MB

190us to transfer 2MiB: speed = 11.04GB/s

- CAPI way saves the time for “SW gather” with relatively small penalty when K grows
- Once tuned (using pragmas), HLS can compete with Verilog coding
Metal FS

A filesystem interface to chain FPGA operators from command line and manage FPGA storage.

```bash
$ echo "Hello World. MetalFS is so simple." | /fpga/op/upper_case
HELLO WORLD. METALFS IS SO SIMPLE.

$ echo "Hello World. MetalFS is awesome." | /fpga/op/encrypt > /fpga/storage/hello.txt
$ /fpga/op/metal_cat /fpga/storage/hello.txt | /fpga/op/decrypt | /fpga/op/upper_case
HELLO WORLD. METALFS IS AWESOME.
```

[https://github.com/osmhpi/metal_fs](https://github.com/osmhpi/metal_fs)
**Scenario:** Use multiple operators (encryption, compression, filtering,...) in command line using data in SSDs while all processing is done in a FPGA

1. **Implement in FPGA a basic FS for the SSD Storage**
   - Precomputation of block count prefix sim enables parallelization
   - FS metadata managed by host system using LMDB (directory structure, filenames, file attributes, free extent lists)

2. **Make the files accessible through the Linux FS:**
   - *libfuse* was used to build the mountable FS driver:
     - easy interface
     - single threaded execution
     - independent from Metal FS core

3. **Stand-in executables which represent operators on the FPGA**
   - Multiple independent executables using the same FPGA delegate processing to the FS driver.
   - FS driver has a list of all operators and handles I/O requests for the virtual stand-in executables files
   - Payload data transfer via memory-mapped files
   - user can add his stand-in executable

4. **Stream switch to orchestrate operators in FPGA**
   - Host
   - FPGA
   - Configuration
   - Input Stream
     - Change case
     - Blowfish Encrypt
     - Blowfish Decrypt
     - Add user block
     - NVMe memory
   - Output Stream

---

```
Metal FS

Scenario: Use multiple operators (encryption, compression, filtering,...) in command line using data in SSDs while all processing is done in a FPGA

2. Make the files accessible through the Linux FS:
   *libfuse* was used to build the mountable FS driver:
   - easy interface
   - single threaded execution
   - independent from Metal FS core

3. Stand-in executables which represent operators on the FPGA
   - Multiple independent executables using the same FPGA delegate processing to the FS driver.
   - FS driver has a list of all operators and handles I/O requests for the virtual stand-in executables files
   - Payload data transfer via memory-mapped files
   - user can add his stand-in executable

1. Implement in FPGA a basic FS for the SSD Storage
   - Precomputation of block count prefix sim enables parallelization
   - FS metadata managed by host system using LMDB (directory structure, filenames, file attributes, free extent lists)

```

---

*OpenPOWER™ Coherent Acceleration Processor Interface (CAPI)*
Results: (Power8 S824L – CAPI1.0) (FPGA card: Nallatech 250S 2NVMe SSDs with 960GB – 4GB DRAM)

- User can define a new operator coding the function in C for the FPGA and coding the configuration at a software level.
- Offload CPU and transparently use compute FPGA capabilities.

```bash
user@antipode$ echo "Hello World. MetalFS is awesome."

| ./operators/blowfish_encrypt -k /tmp/key |
| > ./files/encrypted_message.txt

user@antipode$

user@antipode$ ./operators/blowfish_decrypt -k /tmp/key

| ./operators/change_case -l |
| ./operators/blowfish_encrypt -k /tmp/key |
| > ./files/transformed_message.txt

user@antipode$

user@antipode$ ./operators/blowfish_decrypt -k /tmp/key2

| ./operators/blowfish_encrypt -k /tmp/key

??’???”n?q?b~foot

user@antipode$

user@antipode$ ./operators/blowfish_decrypt -k /tmp/key

hello world. metalfs is awesome.

user@antipode$

Encrypt with “key”
Decrypt with “key”
+ lowercase processing
+ Re-encrypt with “key”
Decrypt with wrong “key2”
Decrypted with right “key”
```
The Bidirectional Long Short-Term Memory (BLSTM) Neural Network case
A transprecision accelerator to solve Optical Character Recognition (OCR) problem.

How tolerating some loss of quality in computed results can greatly improve energy efficiency. Why FPGA helps?

https://github.com/oprecomp/HLS_BLSTM
https://ieeexplore.ieee.org/document/8373077

D. Diamantopoulos, H. Giefers, C. Hagleitner: ecTALK: Energy efficient coherent transprecision accelerators - The bidirectional long short-term memory neural network case. COOL CHIPS 2018
**Implementation:** BLSTM is a neural network algorithm. FPGA implementation allows controlling the use of multiple streaming engines with different precision.

**1. Reducing precision increase parallel feeding of PE**
With BLSTM algorithm, reducing precision down to 5 bits (in some steps) affects final accuracy with only 0.01% ➔ more elements per memory words accessed ➔ parallel feeding of more Processing Elements (PE)

**2. Reducing precision reduce FPGA resources used**
Finding the trade-off between FPGA resources and execution time of a single layer of BLSTM, by tuning the transprecision way of coding.

- Reducing precision increase parallel feeding of PE
- Reducing precision reduce FPGA resources used

---

D. Diamantopoulos, H. Giefer, C. Haagltemer: ecTALK: Energy efficient coherent transprecision accelerators - The bidirectional long short-term memory neural network case. COOL CHIPS 2018
**Results:** (Power8 – 8 cores 3.491GHz - CAPI1.0) (FPGA card: AlphaData KU3 – 250MHz)

4 accelerators on the FPGA compared to 8-P8 SMT4 cores (affinity OpenMP Threads / core=1.1)

- Constant speedup of 4.8-5.6x using the same acceleration scalability step
- Minimal CPU usage of “HW solution” (interrupt-based CAPI API)

---

- **Up to 33x energy efficiency in kPixels/J** compared to CPU (Power8 4.2GHz)
- **Up to 2.4x energy efficiency in kPixels/J** compared to GPU (high-end P100 in float 16)
- **Negligible accuracy loss** compared to software (<0.6% for 3401 images).
Takeaways

- **CAPI** removes the driver latency that a *classic* “FPGA + drivers” adds
- **HLS** can be easily tuned to get performances as good as low level language
- **SNAP** follows the CAPI/OpenCAPI evolution without a change in user’s code
- **Open-source** helps integration with other software (libfuse…) and motivate new IPs/projects coded based on SNAP and CAPI/OpenCAPI
- **Complex C/C++ codes** *(3000 lines)* can be used for FPGA programming
- **CAPI / OpenCAPI Simulation Engine** save huge time for decoding