Bruno MESNET
Alexandre CASTELLANE
Hardware accelerator enablement

IBM France - COGNITIVE

SNAP demonstration
“hello world” example

University of Illinois
June 11th, 2019
Let’s understand SNAP with a “hello world” example

WHAT POWER + FPGA BRING => COHERENCY

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

Application on Server

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

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

Hello World. I love this new experience with SNAP

"Lower case" processing ➔ "software" action

"Upper case" processing ➔ "hardware" action

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

Hello world. I love this new experience with snap
Architecture of the SNAP git files

SNAP

**actions**
- hls_helloworld
  - sw
    - snap_helloworld.c
    - action_lowercase.c
  - hw
    - action_uppercase.cpp
    - action_uppercase.H
  - include
    - action_changecase.h
  - tests
  - doc

**hardware**
- logs
- sim
- capi2-bsp
- build/Images

**software**
- include
  - snap_types.h
- tools
  - snap_maint
  - snap_find_card
helloworld registers

SNAP

`snap_helloworld` application

POWER 8 or 9

Job Manager

DMA

MMIO

`action_uppercase.cpp`

« Action » RTL code generated by HLS

AXI

DRAM on-card

AXI

NVMe

AXI

Network (TBD)

queue_workitem

act/flags | seq | retc | priv_data

Action registers

addr_in

addr_out

typedef struct {
    CONTROL Control; /* 16 bytes */
    action_job_t Data; /* up to 108 bytes */
    uint8_t padding[SNAHLS_JOBSIZE - sizeof(action_job_t)];
} action_reg;

typedef struct snap_addr {
    uint64_t addr;
    uint32_t size;
    snap_addrtype_t type; /* DRAM, NVME, ... */
    snap_addrflag_t flags; /* SRC, DST, EXT, ... */
} snap_addr_t;

$ACTION_ROOT/hw/hw_action.H
SNAP solution execution: Discovery mode (done once)

SNAP starts the inventory of all actions of all cards:
1. Query available actions types
2. Query number of actions
3. Enable actions

SNAP program call

```
snap_maint -v -C<Card#>
```

Each function of each card returns his identification and exit without any processing:

```
Action_Config->action_type = HELLOWORLD_ACTION_TYPE;
Action_Config->release_level = RELEASE_LEVEL;
act_reg->Control.Retc = 0xe00f;
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Action Types</th>
<th>Level</th>
<th>Short</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MEMCOPY</td>
<td>0x21</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>MEMCOPY</td>
<td>0x21</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>HELLOWORLD</td>
<td>0x2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>SEARCH</td>
<td>0x4</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This table allow SNAP to know all available resources and assign actions to the different requests got from applications/threads
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 than 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, HELLOWORLD_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);

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;

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 into the host 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
   
   ```plaintext
   rc = __file_write(output, addr_out, size);
   ```

8. Detach action
   - Disallocate the card
   - Free the dynamic allocation of buffers
   
   ```plaintext
   snap_detach_action(action);
   snap_card_free(card);
   __free(obuff);
   __free(ibuff);
   ```

### C/ C++ code used in Application

---

### Host System memory

#### Data memory area

- @addr_in
- @addr_out

---

### Input Text

---

### Output text

---

### Action X

### Action Y

### Action Z
A SIMPLE 3 STEPS PROCESS

1. **ISOLATION**
   - SNAP_CONFIG=CPU snap_helloworld -i /tmp/t1 -o /tmp/t2
   - x86 server
   - Application: CPU
   - Action: "Lower case" processing ➔ "software" action
   - command: make

2. **SIMULATION**
   - SNAP_CONFIG=FPGA snap_helloworld -i /tmp/t1 -o /tmp/t2
   - x86 server
   - Application: FPGA
   - Action: "Upper case" processing ➔ "hardware" action
   - command: make sim

3. **EXECUTION**
   - SNAP_CONFIG=FPGA snap_helloworld -i /tmp/t1 -o /tmp/t2
   - POWER8/9 server
   - Application: FPGA
   - Action: "Upper case" processing ➔ "hardware" action
   - command: make image

*No specific test bench required
Use your actual application*
Application Code + software action code: what’s in it?

Start

Get input arguments to set action configuration

Allocate card

Attach action

prepare helloworld:
Addr_set(IN)
Addr_set(OUT)
Job_set

snap_action
sync_execute_job

Read data from input_file if defined

write to output_file if output_buffer is in Host memory

Compare data if Verify option and type_in and type_out = Host memory

Get the src@ and len of text

Convert characters to lower case

Write data to dst@

Print results

Detach action

Detach card

Exit

Application: snap_helloworld.c

CPU executed action: action_lowercase.c
Hardware action Code: what's in it?

1. **hls_action**
2. **process_action**

**Used during discovery phase only**

**FPGA executed Action**: action_uppercase.cpp

- **Exit action sending back**: `Action_Config->action_type` and `Action_Config->release_level`

- **Start**
  - Is `Act_reg->Control.flags` set?
  - No
    - Exit action
  - Yes
    - Align `i_idx` and `o_idx` with port width
    - Get size of the input text
    - Read one 64B word at `@i_idx`
    - Cast the 64B word to a char[64]
    - Convert char[64] to uppercase
    - Cast the char[64] to a 64B word
    - Write one 64B word to `@o_idx`
    - Set `ReturnCode` and exit action

- **Check**
  - Increment `i_idx` and `o_idx`
  - Decrement size until size > 0
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/powercapi_snap
https://github.com/open-power/snap

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