Hands-on lab: Port a SHA3 coded in C to a FPGA
SNAP Enabled Card Details: Nallatech 250S (on P8)

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

Choose this card for:
2TB of on-card Flash

FPGA to Host Memory Access
Latency to/from FPGA: 0.8us
Bandwidth to FPGA: ~3.8GB/s reads and writes (CAPI limit)

3.5MB Block Ram on FPGA

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

Nimbix

SNAP Framework built on Power™ CAPI technology
2019, IBM Corporation
SNAP Enabled Card Details: Alpha-Data ADM-PCIE-KU3 (on P8)

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

Choose this card for:
External IO
Offload and DRAM

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

8GB DDR3
Latency to FPGA: 230ns

Nimbix

SNAP Framework built on Power™ CAPI technology
SNAP Enabled Card Details: Nallatech 250S+ (on P9)

Up to 4x1TB NVMe sticks (3.84TB effective)
Latency to FPGA: ~0.8ms
Bandwidth to FPGA: Read 1.8GB/s

University of Illinois

Xilinx KU15P
1.1 million LC
70MB Block Ram
on FPGA

Up to 8GB DDR4 (on back of card)
Latency to FPGA: 184ns Read / 105ns write
Bandwidth to FPGA: ~13 to 14GB/s reads and writes (CAPI limit)

FPGA to Host Memory Access PCIe Gen4x8
Latency to/from FPGA: 0.8us
Bandwidth to FPGA: ~11 to 13GB/s reads and writes (CAPI limit)
Porting and Optimizing code flow

1. Download, compile and execute C code
2. Synthesize the code to port into the FPGA
3. **Evaluate the performance of the action:** Measure/optimize
4. Adapt your code to SNAP
5. Adapt the application to call the action code
6. Measure the action performance
1 - Download, compile and execute C code
Download and evaluate your program performances

• Download source code

```
mkdir education; cd education
git clone https://github.com/mjosaarinen/tiny_sha3
cd tiny_sha3
make
./sha3test
```

$ ./sha3test
FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!
2 - Synthesize the code to port into the FPGA
Porting code example into a new project

• Let’s copy this code into a new directory

   ➢ cd ..
   ➢ mkdir hw; cd hw
   ➢ cp ../*/*.c .
   ➢ cp ../*/*.h .
   ➢ mv main.c main.cpp
   ➢ mv sha3.c sha3.cpp
   ➢ mv sha3.h sha3.H

• Run Vivado HLS

   ➢ which vivado_hls
   ➢ /opt/Xilinx/Vivado/2018.3/bin/vivado_hls

   The only reason why we rename the c files to cpp is to get more functionalities with Vivado HLS

   we use 2018.3 on Nimbix to circumvent a bug in the GUI of HLS2018.1!
Create HLS project:

1. Type a project name under education/hw.
2. Add only the cpp files (no header files) and then Browse to find the “test_speed” function.
Create HLS project:

Then similar window for the testbench – Press “Add files”

Add only the main.cpp file

For this example, set Period 4 (ns) and set part to any KU060 chip

Solution Configuration
Create Vivado HLS solution for selected technology

Solution Name: solution1
Clock
Period: 10
Uncertainty
Part Selection
Part: xqku115-rif1924-2-i
Run simulation

Press simulate button

Correct in the 2 cpp files the call of sha3.h by sha3.H

Press simulate button

This only means that the program is running correctly in the same condition than a basic terminal!
Run simulation

It is key to have a test with check the data in case you break your algorithm
Run first synthesis ➔ work bottom-up

- Start synthesizing a small function such as `sha3_keccakf`
Run first synthesis ➔ work bottom-up

- Then proceed with the program one step upper: test_speed

Press synthesis button

Remove calls to “system time”

Press synthesis button
The SHA3 test_speed program structure:

```c
void test_speed()
{
    int i;
    uint64_t st[25], x, n;
    // clock_t bg, us;

    for (i = 0; i < KECCAKF_ROUNDS; i++)
    {
        st[i] = i;
        // bg = clock();
        n = 0;
        // do {
            for (i = 0; i < 100000; i++)
                sha3_keccakf(st);
            n += i;
            // us = clock() - bg;
            // } while (us < 3 * CLOCK_PER_SEC);

        x = 0;
        for (i = 0; i < 25; i++)
            x += st[i];
        // printf("%016lx,% 3f Keccak-p[1600,24] / Second,\n", 
        //       (unsigned long) x, (CLOCK_PER_SEC * ((double) n)) / ((double) us));
    }
}
```

**INFO:** The `printf` display of the `test_speed` function is now N/A since we removed the system time calls. We will later add a loop to replace it.

**NB_ROUNDS** = 100,000

**KECCAKF_ROUNDS** = 24 → 24 calls calling the algorithm process
Porting conclusion

Let’s summarize where we are at this moment:

• **main** (calling test_speed, test_shake, test_sha3) is executed ok (simulation mode)
• **test_speed** *(will recursively call the key calculation routines)* function was fully synthesized.
  • sha3_keccakf was fully synthesized
• **test_shake, test_sha3** *(one shot key calculations)* and have not been ported / synthesized yet

• Let’s correct the remaining warning

```
ERROR: [SYNCHK 200-72] main.cpp:160: unsupported c/c++ library function 'clock'.
WARNING: [SYNCHK 200-77] The top function 'test_speed' (main.cpp:159) has no outputs.
INFO: [SYNCHK 200-10] 1 error(s), 1 warning(s).
ERROR: [HLS 200-70] Synthesizability check failed.
command 'an source' returned exit code
```
Completing to adapt the program

- To check that result is correct, **test_speed** should send back the result which was displayed with the `printf` so that a testbench could automatically check that correct result is always returned ok.

- In a 1st approach, let’s comment the “if” condition to focus on the **test_speed** function.
Completing to adapt the program

- As all data are generated in the `test_speed` function, let’s just add the timing loop we have removed around this `test_speed` function so that we can parallelize the calls.

```c
188 // main
189 #define NB_TESTS_RUNS 3
190 int main(int argc, char **argv)
191 {
192     unsigned int run_number;
193     int checksum;
194     clock_t bg, ms;
195     // Let's comment the call to these 2 functions in a first approach
196     //if (test_sha3() == 0 && test_shake() == 0)
197     //    printf("FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!\n");
198     // Add the clock measurement
199     bg = clock();
200     for (run_number = 0; run_number < NB_TESTS_RUNS; run_number++)
201         checksum -= test_speed();
202     ms = (clock() - bg) / 1000.0;
203     // Print the result as in the initial program
204     printf("%.16lx $3f Keccak-p[1600.24] / Second.\n",
205         (unsigned long) checksum, ((NB_TESTS_RUNS * NB_ROUNDS * 1000) / ((double) ms)));
206     return 0;
207 }
```
Completing to adapt the program

- Run simulate

In this step, you have synthesized the code of the algorithm that will be placed in the FPGA and put in place the testbench to check it.
Vivado HLS Constraints on C/C++ code

➔ Download the Xilinx HLS User Guide UG902 – Chapter 3 - Unsupported C Constructs

HLS is converting sequential C/C++ code into parallel fixed resource RTL code, but:
- No Operating System ➔ no dynamic memory management (malloc,...)
- C library function supported if basic use ➔ “memset” is supported if no cast

Recommendations
- Keep your code in small functions
  - Add #pragma HLS INLINE off to keep the hierarchy ➔ helpful for debug
- Set fixed bounds to loops
- Keep code simple to help HLS compiler being efficient
- “Switch/case” is often more efficient than “if/then/else”
- Always test your code without #pragma and test after each adding
- Unused logic is removed by Vivado: Remove unused ports to remove the drivers if not needed
- Use SNAP examples to find tips&tricks
3 - Evaluate the performance of the action
Understand the architecture of the code

```c
#define NB_TEST_RUNS 65536

test_runs:
for (run_number = 0;
    run_number < NB_TEST_RUNS;
    run_number++)
    checksum ^= test_speed();
```

```c
#define NB_ROUNDS 100000

uint64_t test_speed ()
{
    for( i=0; i < 25; i++ )
        st[i] = 1;

test_rounds: for (i = 0; i < NB_ROUNDS; i++)
    sha3_keccakf(st);
    for( i=0; i < 25; i++ )
        x += st[i];
    return x;
}
```

```c
#define KECCAKF_ROUNDS 24

void sha3_keccakf(uint64_t st[25])
{
    keccakfronds:
    for (r = 0; r < KECCAKF_ROUNDS; r++) {
        processing Theta + Rho Pi + Chi
    }
}
```
Understand the key numbers to analyze performances

- Let’s synthesize the keccakf function

1. Timing

2. Resources

3. Latency
First run – first numbers – first measurements

- Now look to more details

In summary, this means that the `sha3_keccakf` function:
- Can be synthesized within the clock constraints given (3.14 < 3.5ns)
- Will take 772FF and 1928LUT (whatever it means)
- Will take $354 \times 24 \times 4\text{ns} = 33984\text{ns} = 33.984\mu\text{s}$ to be executed.

➔ we can run this function **29 425 times per second.**

```bash
$ ./sha3test
FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!
```
HLS #pragma (cf HLS User Guide UG902)

24 “Optimizations directives” – 5 major ones are highlighted – only 2 used in this example

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATION</td>
<td>Specify a limit for the number of operations, cores or functions used. This can force the sharing or hardware resources and may increase latency.</td>
</tr>
<tr>
<td>ARRAY_MAP</td>
<td>Combines multiple smaller arrays into a single large array to help reduce block RAM resources.</td>
</tr>
<tr>
<td>ARRAY_PARTITION</td>
<td>Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.</td>
</tr>
<tr>
<td>ARRAY_RESHAPE</td>
<td>Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.</td>
</tr>
<tr>
<td>CLOCK</td>
<td>For SystemC designs multiple named clocks can be specified using the create_clock command and applied to individual SC_MODULES using this directive.</td>
</tr>
<tr>
<td>DATA_PACK</td>
<td>Packs the data fields of a struct into a single scalar with a wider word width.</td>
</tr>
<tr>
<td>DATAFLOW</td>
<td>Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.</td>
</tr>
<tr>
<td>DEPENDENCE</td>
<td>Used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).</td>
</tr>
<tr>
<td>EXPRESSION_BALANCE</td>
<td>Allows automatic expression balancing to be turned off.</td>
</tr>
<tr>
<td>FUNCTION_INSTANTIATE</td>
<td>Allows different instances of the same function to be locally optimized.</td>
</tr>
<tr>
<td>INLINE</td>
<td>Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Specifies how RTL ports are created from the function description.</td>
</tr>
<tr>
<td>LATENCY</td>
<td>Allows a minimum and maximum latency constraint to be specified.</td>
</tr>
<tr>
<td>LOOP_FLATTEN</td>
<td>Allows nested loops to be collapsed into a single loop with improved latency.</td>
</tr>
<tr>
<td>LOOP_MERGE</td>
<td>Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.</td>
</tr>
<tr>
<td>LOOP_TRIPCOUNT</td>
<td>Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.</td>
</tr>
<tr>
<td>OCCURRENCE</td>
<td>Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.</td>
</tr>
<tr>
<td>PIPELINE</td>
<td>Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.</td>
</tr>
<tr>
<td>PROTOCOL</td>
<td>This commands specifies a region of the code to be a protocol region. A protocol region can be used to manually specify an interface protocol.</td>
</tr>
<tr>
<td>RESET</td>
<td>This directive is used to add or remove reset on a specific state variable (global or static).</td>
</tr>
<tr>
<td>RESOURCE</td>
<td>Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.</td>
</tr>
<tr>
<td>STREAM</td>
<td>Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization.</td>
</tr>
<tr>
<td>TOP</td>
<td>The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis. This then allows different solutions within the same project to be specified as the top-level function for synthesis without needing to create a new project.</td>
</tr>
<tr>
<td>UNROLL</td>
<td>Unroll for-loops to create multiple independent operations rather than a single collection of operations.</td>
</tr>
</tbody>
</table>
Increasing performances: 1st major concept

```c
void top(...) {
    ...
    for_mult:for (i=3;i>0;i--) {
        a[i] = b[i] * c[i];
    }
    ...
}
```

#pragma HLS UNROLL
Sha3_keccakf function

1st synthesis – with no code change

In `sha3.cpp` file

- Add the directive
  
  `#pragma HLS UNROLL`

Press synthesis button

![Diagram showing synthesis results and code snippet](image-url)
Increasing performances: 2\textsuperscript{nd} major concept

```c
void func(m, n, o) {
    for (i=2; i>=0; i--) {
        op_Read;
        op_Compute;
        op_Write;
    }
}
```

(A) Without Loop Pipelining

(B) With Loop Pipelining

#pragma HLS PIPELINE
**Sha3_keccakf function**

1\textsuperscript{st} synthesis – with no code change

**Reference**
(function synthesized alone)

---

In `sha3.cpp` file

⇒ Add the directive

```
#pragma HLS PIPELINE
```

Press synthesis button
Synthesis result with PIPELINE

Separate input and output variables

```c
void sha3_keccakf(uint64_t st_in[25], uint64_t st_out[25])
{
    // constants
}
```

Update also sha3_keccakf calls + definition in cpp + H files

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

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

```c
for (r = 0; r < KEECAKF_ROUNDS; r++) {
    #pragma HLS PIPELINE
    // Theta
}
```
Sha3_keccakf function

1\textsuperscript{st} synthesis – with no code change

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP48E</th>
<th>FF</th>
<th>LUT</th>
<th>Pipelined</th>
<th>Latency</th>
<th>Iteration Latency</th>
<th>Initiation Interval</th>
<th>Trip count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td>4.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Estimated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncertainty</td>
<td>2.854</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td>4.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Estimated</td>
<td>2.124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncertainty</td>
<td>0.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td>772</td>
<td>1837</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td>2160</td>
<td>2760</td>
<td></td>
<td>663360</td>
<td>3331680</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRAM / 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRAM / 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2\textsuperscript{nd} synthesis – adding #pragma HLS PIPELINE in main loop

LUT x4.8

Latency / 219

Reference
(function synthesized alone)
Summary with a PIPELINE instruction

In summary, this means that the `sha3_keccakf` function:
- Can be synthesized within the clock constraints given (< 3.5ns)
- Will take 2% of the logic
- Will take 2 x 24 x 4ns = 192ns to be executed.

$ ./sha3test  

Remember the simple SW compilation and execution:
FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!  
Understand the architecture of the code

```c
#define NB_TEST_RUNS 65536

int test_runs:
    for (run_number = 0;
        run_number < NB_TEST_RUNS;
        run_number++)
    {
        checksum ^= test_speed();
    }

#define NB_ROUNDS 100000

uint64_t test_speed ()
{
    for (i=0; i < 25; i++)
        st[i] = 1;

    test_rounds:
        for (i = 0; i < NB_ROUNDS; i++)
            sha3_keccakf(st);
        for (i=0; i < 25; i++)
            x += st[i];
    return x;
}
```

```c
#define KECCAKF_ROUNDS 24

void sha3_keccakf(uint64_t st[25])
{
    keccakfrounds:
        for (r = 0; r < KECCAKF_ROUNDS; r++)
            processing Theta + Rho Pi + Chi
}
```

**Parallel loops**

**Recursive loops**

**Math function**
4 - Adapt your code to SNAP

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

• Read README.txt
• Download source code

➢ git clone https://github.com/open-power/snap
➢ cd snap
➢ make software
Create your SNAP application/function

- Create a new action or copy a from an already coded example

- Choose the example which has the same ports
  ➔ [https://github.com/open-power/snap/tree/master/actions#action-descriptions](https://github.com/open-power/snap/tree/master/actions#action-descriptions)
1. Define configuration ➔ hls_sponge/include

- List parameters that you need to exchange between application and action

```c
typedef struct tinysha3_job {
    uint64_t chk_in;  /* in: checksum input */
    uint64_t chk_out; /* out: checksum output */
    uint32_t chk_type; /* in: SPONGE, CRC32 */
    uint32_t test_choice; /* in: special parameter for sponge */
    uint32_t nb_elmts;  /* in: special parameter for sponge */
    uint32_t freq;      /* in: special parameter for sponge */
    uint32_t nb_test_runs; /* out: special parameter for sponge */
    uint32_t nb_rounds; /* out: special parameter for sponge */
} tinysha3_job_t;
```
2. Define data flow ➔ hls_sponge/hw

- Select the ports you need for your application
void hls_action(snapshot_membus_t *din_gmem,
    snapshot_membus_t *dout_gmem,
    // snapshot_membus_t *d_ddrmem,
    action_reg *Action_Register,
    action_RO_config_reg *Action_Config)
{
    // Host Memory AXI Interface
    #pragma HLS INTERFACE m_axi port=din_gmem bundle=host_mem offset=slave depth=512
    #pragma HLS INTERFACE s_axilite port=din_gmem bundle=ctrl_reg offset=0x030

    #pragma HLS INTERFACE m_axi port=dout_gmem bundle=host_mem offset=slave depth=512
    #pragma HLS INTERFACE s_axilite port=dout_gmem bundle=ctrl_reg offset=0x040

    // DDR memory Interface
    // #pragma HLS INTERFACE m_axi port=d_ddrmem bundle=card_mem0 offset=slave depth=512
    // #pragma HLS INTERFACE s_axilite port=d_ddrmem bundle=ctrl_reg offset=0x050

    // Host Memory AXI Lite Master Interface
    #pragma HLS DATA_PACK variable=Action_Config
    #pragma HLS INTERFACE s_axilite port=Action_Config bundle=ctrl_reg offset=0x010
    #pragma HLS DATA_PACK variable=Action_Register
    #pragma HLS INTERFACE s_axilite port=Action_Register bundle=ctrl_reg offset=0x100
    #pragma HLS INTERFACE s_axilite port=return bundle=ctrl_reg offset=0x105

    /* Hardcoded numbers */
    switch (Action_Register->Control.flags) {
    case 0:
        Action_Config->action_type = (snapu32_t)TINYSHA3_TYPE;
        Action_Config->release_level = (snapu32_t)RELEASE_LEVEL;
        Action_Register->Control.Ind = (snapu32_t)0xe00f;
        return;
        break;
    default:
        Action_Register->Control.Ind = (snapu32_t)0x0;
        process_action(Action_Register);
        break;
    }
void process_action (action_reg *Action_Register) 
{
    int rc = 1;
    uint64_t run_number, j;
    uint64_t checksum = 0;
    // uint64_t checksum_tmp;
    uint32_t nb_elmts, freq;

    switch (Action_Register->Data.test_choice) {
    case (0):
        nb_elmts = Action_Register->Data.nb_elmts;
        freq = Action_Register->Data.freq;

        test_runs:for (run_number = 0; run_number < NB_TEST_RUNS; run_number++)
#pragma HLS UNROLL factor=4  // PARALLELIZATION FACTOR - max is 32
        checksum ^= test_speed(run_number, nb_elmts, freq);

        Action_Register->Data.chk_out = checksum;
        Action_Register->Data.nb_test_runs = NB_TEST_RUNS;
        Action_Register->Data.nb_rounds = NB_ROUNDS;
        rc = 0;
        break;
    }
}

Test_speed call

Return results in registers
3. Read and write data from ports

- **Addressing**

  ```c
  /* byte address received need to be aligned with port width */
  input_address = act_reg->Data.in.addr >> ADDR_RIGHT_SHIFT;
  output_address = act_reg->Data.out.addr >> ADDR_RIGHT_SHIFT;
  ```

- **Data access: MMIO vs DMA**

- **Data formatting**

  ```c
  char text[BPERDW]; /*BPERDW (Byte per Data Word is defined as 64*/
  /* Read in one 64B word */
  memcpy((char*) text, din_gmem + input_address, BPERDW);
  ```
3. Read and write data from ports

- Data access frequency

```c
snap_membus_t buffer[MAX_NB_OF_WORDS_READ]:
    switch (memory_type) {
        case SNAP_ADDRTYPE_HOST_DRAM:
            memcpy((snap_membus_t *) (dout_gmem + input_address), buffer,
                    size_in_bytes_to_transfer);
            break;
        case SNAP_ADDRTYPE_CARD_DRAM:
            memcpy((snap_membus_t *) (d_ddrmem + input_address), buffer
                    size_in_bytes_to_transfer);
            break;
    }

for (int k=0; k<size_in_words; k++)
#pragma HLS PIPELINE
    buffer[k] = (din_gmem + input_address)[k];

buffer[0] = (din_gmem + input_address)[0];
or
(dout_gmem + input_address)[0] = buffer[0];
```
5 - Adapt the application to call the action code
Implement the switch from sw to hw actions ➔ hls_sponge/sw

- card = `snap_card_alloc_dev`(device, SNAP_VENDOR_ID_IBM, SNAP_DEVICE_ID_SNAP);
- action = `snap_attach_action`(card, TINYSHA3_ACTION_TYPE, action_irq, timeout);
- `snap_prepare_tinysha3`(.../...) function.
- rc = `snap_action_sync_execute_job`(action, &cjob, timeout);
- Test the return code and check/display the results
- Clean before exiting detaching the action, freeing the card and freeing the memory allocated
6 – Tools to measure the action performance
+ tools to debug the code
Use the tools to measure & optimize
**Offload Method:**

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

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

*The lower the better*

https://github.com/open-power/snap/tree/master/actions/hls_sponge
Use the tools to debug

vivado_hls

SNAP TRACE=0xF

/software vs hardware action

./snap_trace
Questions?