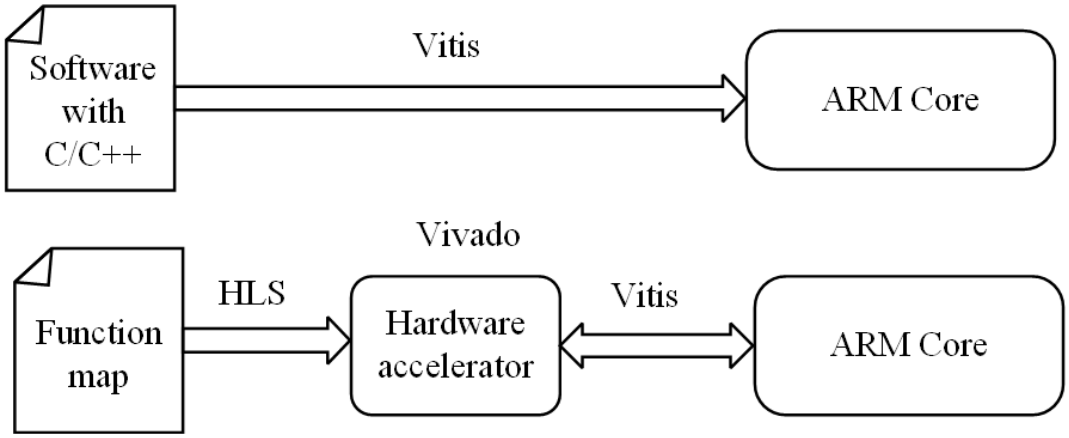
Tutorial 11: Using SoC Framework to accelerate numerical computing

Objectives

After completing this lab, you will be able to:

- Have basic concepts of efficient hardware design with HLS.
- Build a hardware system with AXI Stream protocol.
- Deploy the hardware system on Vitis/PYNQ platform.
- Have concept about hardware acceleration compared with software.

In this exercise, we will test the map function in two experiments: software program using C/C++, and the hardware accelerator. Then we would compare the performance. You will see how hardware design can accelerate computation and why FPGA is important in data center nowadays.



Map Function

The Map function takes a series of values, processes each, and generates zero or more output values. For example, give a list [0, 1, 2, 3, 4], processes with f(x) = x^2, the output will be [0, 1, 4, 9, 16].

Steps

A. Create a Vivado HLS project implementing the map function.

1. In this exercise, our design will receive a series of input data (in stream format), perform a function f(x) = x^3+2*x^2+1, then generate the output for each input element.
2. Create HLS project, select device, **XC7Z020clg400-1**.
3. Create the design file ex_accle.cpp. The full code is:

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"
using namespace hls;

typedef hls::axis<float, 1, 1, 1> data_t;
typedef hls::stream<data_t> mystream;

void readStream(mystream &in, data_t &out);
void doComputing(data_t &in, data_t &out);
void writeStream(data_t &in, mystream &out);

void func_map(mystream &in_data, mystream &out_data){
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=in_data
#pragma HLS INTERFACE axis port=out_data
#pragma HLS DATAFLOW

    data_t tmp1;
    data_t tmp2;

    readStream(in_data, tmp1);
    doComputing(tmp1, tmp2);
    writeStream(tmp2, out_data);
}

void readStream(mystream &in, data_t &out)
{
#pragma HLS PIPELINE II=1
    out = in.read();
}

void writeStream(data_t& in, mystream& out)
{
#pragma HLS PIPELINE II=1
out.write(in);
}

void doComputing(data_t &in, data_t &out)
{
#pragma HLS PIPELINE II=1
    out.data = in.data * in.data * in.data + 2.0 * in.data * in.data + 1.0;
    out.keep = in.keep;
    out.dest = in.dest;
    out.id = in.id;
    out.strb = in.strb;
    out.user = in.user;
    out.last = in.last;
}
```

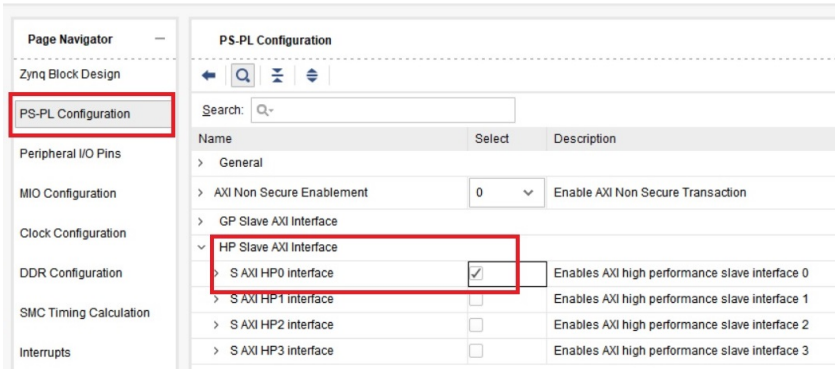
4. We write a struct named **data_t**, the struct describes the AXI4-Stream interface with side-channel signals. Besides, we use HLS Stream library **hls_stream.h** for modeling streaming data structures. The **hls::stream<>** class has **read()** and **write()** methods that can read from and written to sequentially.
5. The top function **func_map** adopts a **DATAFLOW**. It is a kind of task-level parallelism strategy. We have three functions: **readStream**, **doComputing**, and **writeStream**. Each

function executes individually as long as the input is available and output FIFO is not full.

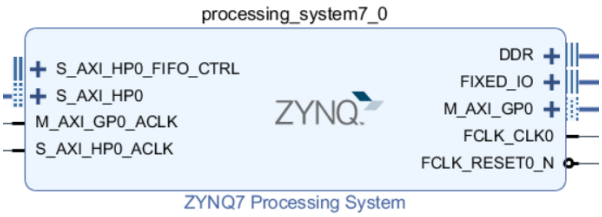
6. For each function, we add **PIPELINE** directive. It is another kind of parallelism strategy. The function is able to receive one data every clock. If you have the concept of pipeline, it is easy to understand.
7. Do **C Synthesis** and **Export RTL** just like tutorial 9. We get our design IP and will build the system in the next step.

B. Build the hardware system in Vivado

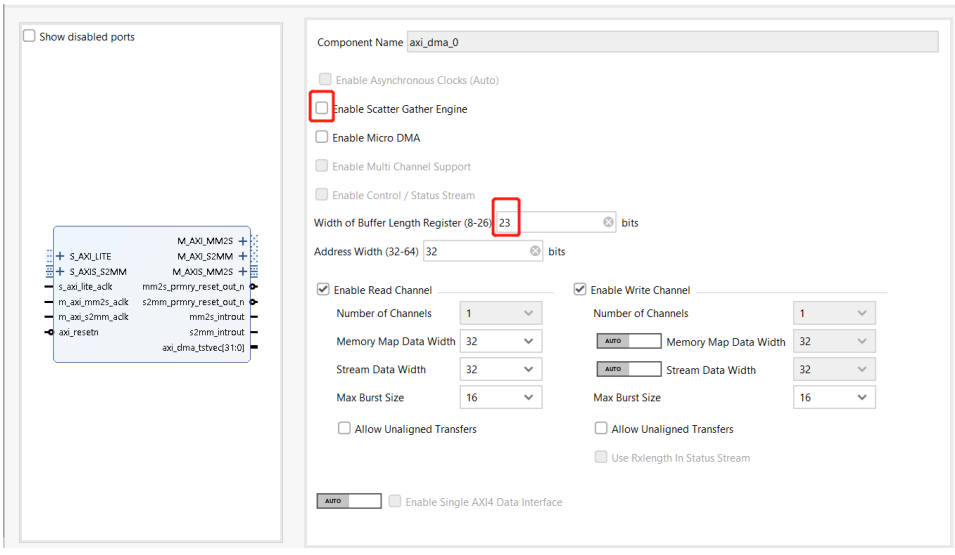
1. Create a new project named **ex_ACCLE** in Vivado. Select the device **XC7Z020clg400-1**. Similar to tutorial 10, add your IP to the repository.
2. Create a block design. Add **ZYNQ7 Processing System**. Click **Preset** and **Apply Configuration**, select **pynq_z2.tcl** file. Then find **PS-PL Configuration->HP Slave AXI Interface**, select HP0 interface.



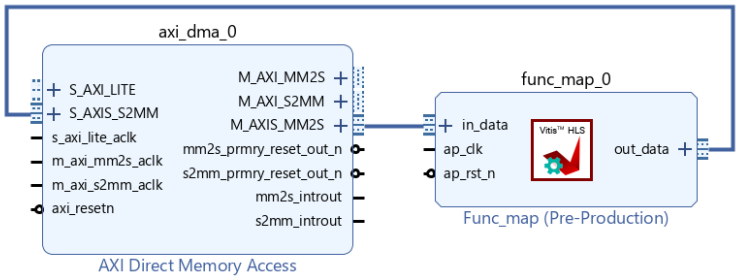
3. Your Zynq has following configuration with an HP port. Then Click **run block automation**.



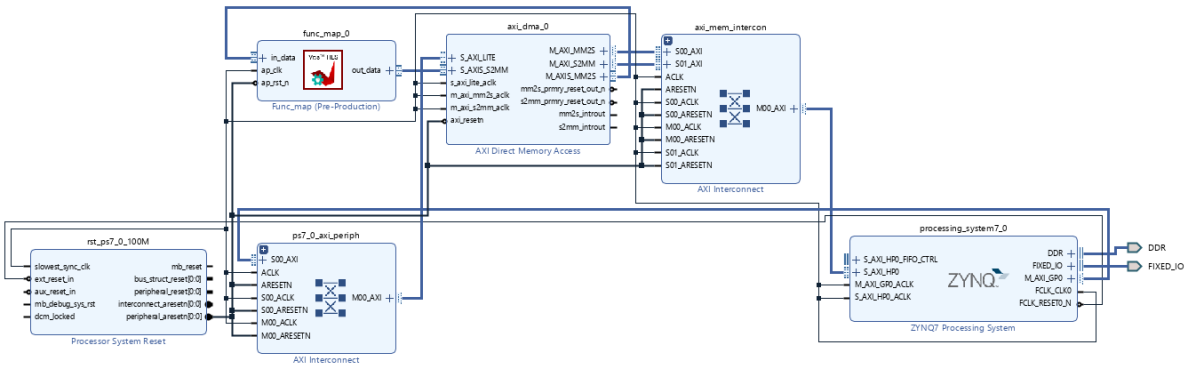
4. Add another IP **AXI Direct Memory Access**, configure the IP.



5. Add custom IP **func_map**, it has two streaming data port. Connect it with **DMA**.



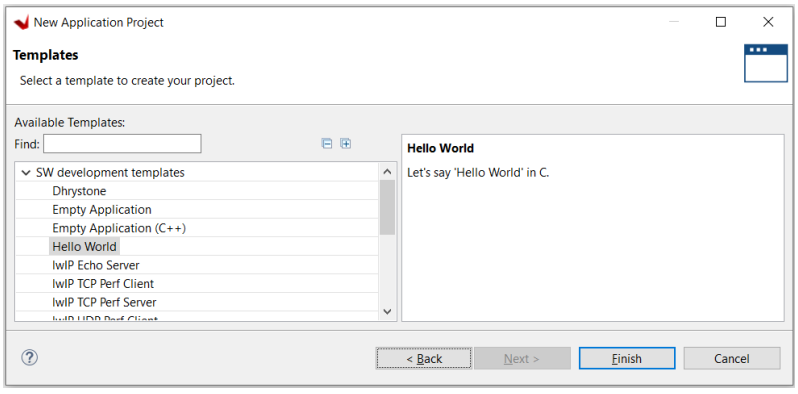
6. Click **run connection Automation** to complete the system design, then regenerate layout and valid design. **AXI_Lite** port is connected to **GP** port, **S2MM** and **MM2S** are connected to **HP** port.



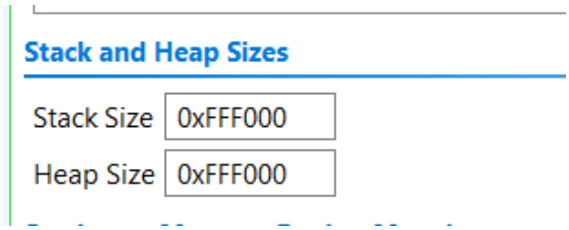
7. Go to **Sources**, right click **design_1**, click **Generate output products**, then click **Create HDL Wrapper**. Generate bitstream file and export hardware similar to tutorial 10.

C. Run program on Vitis

1. Create a Vitis application project from exported hardware, name **ACCLE**, choose the **Hello world** template.



2. Click the **ACCLE -> src** folder, then click the **lscript.ld** file, set the **Stack Size** and **Heap Size** to be **0xFFF000**.



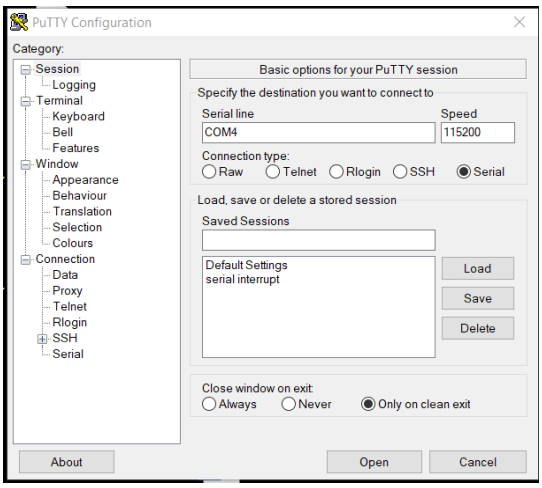
3. Click the **helloword.c** file, replace with the following code:

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xaxidma.h"//DMA
#include "xparameters.h"//address
#include "xil_exception.h"//exception related APIs
#include "xscugic.h"//interrupt controller
#include "xscutimer.h"

XAxiDma axidma;     //XAxiDma Instance
XScuGic intc;       //interrupt controller Instance
volatile int tx_done;      //Send complete flag
volatile int rx_done;      //Receive complete flag
volatile int error;        //Transmission error flag

/*add definitions*/
#define DMA_DEV_ID          XPAR_AXIDMA_0_DEVICE_ID
#define RX_INTR_ID          XPAR_FABRIC_AXIDMA_0_S2MM_INTROUT_VEC_ID
#define TX_INTR_ID          XPAR_FABRIC_AXIDMA_0_MM2S_INTROUT_VEC_ID
#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID

void transmission_initialization()
{
    int status;
    XAxiDma_Config *config;

    config = XAxiDma_LookupConfig(DMA_DEV_ID);
    if (!config){
        xil_printf("No config found for %d\r\n", DMA_DEV_ID);
        return XST_FAILURE;
    }

    status = XAxiDma_CfgInitialize(&axidma, config);
    if (status != XST_SUCCESS) {
        xil_printf("Initialization failed %d\r\n", status);
        return XST_FAILURE; }

    if (XAxiDma_HasSg(&axidma)) {
        xil_printf("Device configured as SG mode \r\n");
        return XST_FAILURE;
    }

    /* Disable interrupts, we use polling mode*/
```

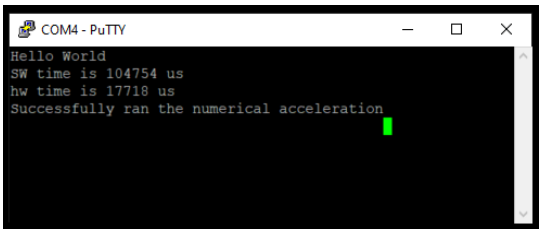
```
    XAxiDma_IntrDisable(&axidma, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(&axidma, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);
}

static XScuTimer_Config *config = NULL;
static XScuTimer            scutimer;
static unsigned int     tt_start;
static unsigned int     tt_end;

void scutimer_init()
{
    tt_start = tt_end = 0;
    XScuTimer_Config *config =
XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);
    XScuTimer_CfgInitialize(&scutimer, config,
XPAR_PS7_SCUTIMER_0_BASEADDR);
    XScuTimer_LoadTimer(&scutimer, 0xFFFFFFFF);
}

void scutimer_start()
{
    if(config == NULL)
        scutimer_init();

    XScuTimer_LoadTimer(&scutimer, 0xFFFFFFFF);
    XScuTimer_RestartTimer(&scutimer);
    XScuTimer_Start(&scutimer);
    tt_start = XScuTimer_GetCounterValue(&scutimer);
}

int scutimer_result()
{
    XScuTimer_Stop(&scutimer);
    tt_end = XScuTimer_GetCounterValue(&scutimer);
    double diff = tt_start - tt_end;
    double t = diff/XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ * 2 * 1000* 1000;
    int totalTime = (int)t;
    return totalTime;
}

#define length 1000000

int main()
{
    init_platform();
    int status;
    print("Hello World\n\r");
    transmission_initialization();
    float input_sw[length];
    float output_sw[length];
    float in_hw[length];
    float out_hw[length];

    //software
    for(int i =0; i<length;i++)
    {
```

```
        input_sw[i]=i;
        in_hw[i]=i;
    }

    scutimer_start();
    for(int i =0; i<length;i++)
    {
        output_sw[i] =
input_sw[i]*input_sw[i]*input_sw[i]+2.0*input_sw[i]*input_sw[i]+1.0;
    }
    printf("SW time is %d us\n",scutimer_result());

    scutimer_start();
    Xil_DCacheFlushRange((UINTPTR)in_hw, length*4);
    Xil_DCacheFlushRange((UINTPTR)output_sw, length*4);
    status = XAxiDma_SimpleTransfer(&axidma, (UINTPTR) in_hw, (length*4),
XAXIDMA_DMA_TO_DEVICE);
    if (status != XST_SUCCESS) { return XST_FAILURE; }
    status = XAxiDma_SimpleTransfer(&axidma, (UINTPTR) out_hw, (length*4),
XAXIDMA_DEVICE_TO_DMA);
    if (status != XST_SUCCESS) { return XST_FAILURE; }
    while ((XAxiDma_Busy(&axidma,XAXIDMA_DEVICE_TO_DMA)) ||
(XAxiDma_Busy(&axidma,XAXIDMA_DMA_TO_DEVICE))){}
    printf("hw time is %d us\n",scutimer_result());

    for(int i =0; i<length;i++)
    {
        //printf("%d,%d,
%f,%f\n",i,out_hw[i]==output_sw[i],output_sw[i],out_hw[i]);
    }

    print("Successfully ran the numerical acceleration.\n");
    cleanup_platform();
    return 0;
}
```

4. The **transmission_initialization** function is to initialize the AXI stream transfer. The **scutimer_** function is to calculate the time based on CPU cycles. We define the parameter **length** to be the length of the data array. We use the **Xil_DCacheFlushRange** function to ensure data cache consistency. We use the **XAxiDma_SimpleTransfer** function to transfer the AXI stream data.
5. Open Putty and set the serial connection. The port number should be same as that in device manager.

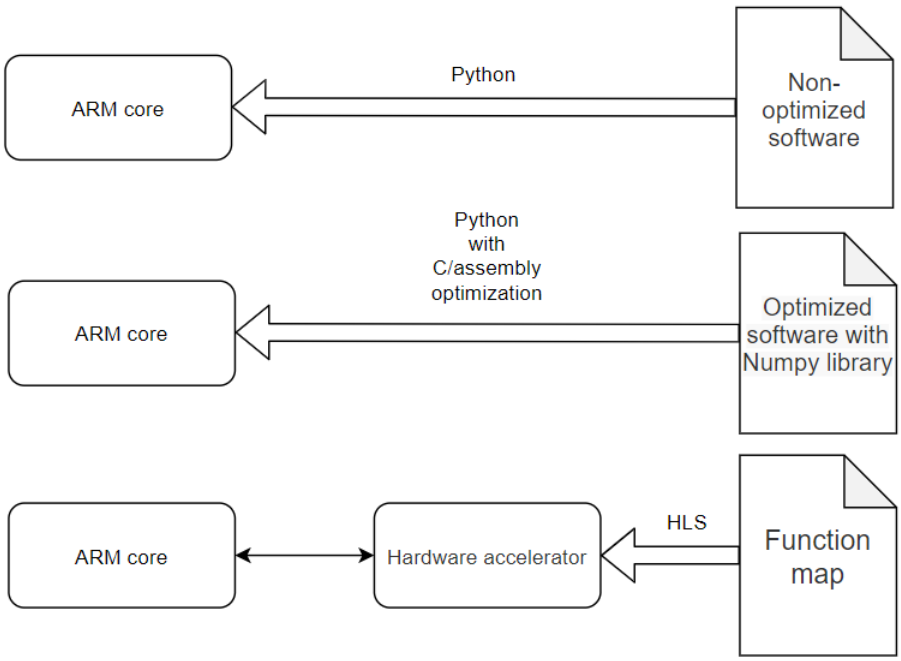


6. Right Click **ACCLE -> build project**, then click **Run as ->launch on hardware**, see the acceleration results from the Putty window.



Experiment	Computing time (second)	Improvement
software (C/C++)	0.104	1
Hardware accelerator	0.017	5.9

Attached: Run numerical acceleration on PYNQ



A. Run program on CPU

1. We first run with **non-optimized software** code:

```
import time

def f(x):
    return x*x*x+2.0*x*x+1.0

input_data = []
for i in range(1000000): # we prepare a list with 1000000 elements
    input_data.append(float(i)) # It's not so fair

start = time.time()
output_data = list(map(f, input_data))
time.time() - start
```

We import **time** library to measure the execution time. Python provide a **map** function, you can pass a function object and a list, it will perform the function on each element in the list and return the result. We prepare a list with 1000000 float elements (very long). The original float type in Python is 64 bits so the comparison is not so fair (but since most time is consumed in for-loop so the result should be similar). The execution time in my test is about **2.385 seconds**.

2. We know that Python language itself is quite slow. Previous code with **map** function on list is equivalent to writing code with for-loop. In Python, **Numpy** library is the fundamental package for scientific computing. It is highly optimized for speed with C++ language with blas. We have the code:

```
import numpy as np
import time
input_data = np.zeros(shape=(1000000,), dtype=np.float32)
output_data = np.zeros(shape=(1000000,), dtype=np.float32)

for i in range(1000000):
    input_data[i] = i

start = time.time()
output_data = np.power(input_data, 3) + 2.0 * np.square(input_data) + 1.0
time.time() - start
```

Firstly, we create input_data and output_data array with 1000000 elements. **numpy.ndarray** is an efficient high-order list structure provided by Numpy for numerical computing (similar to MatLab). Numpy itself provides many element-wise operation functions so the speed is typically faster than the code you write in Python. In my test, the execution time is about **0.352 second**.

B. Test our hardware accelerator

It seems that **0.352 second** is good enough, but actually we can get better results with our simple hardware design with several floating multipliers and floating adders.

```
from pynq import Overlay
import pynq.lib.dma
from pynq import allocate
import numpy as np
import time
```

```
overlay = Overlay('system.bit')
dma = overlay.axi_dma_0 # get dma IP
# prepare input and output buffer, data type is np.float32
in_buffer = allocate(shape=(1000000,), dtype=np.float32)
out_buffer = allocate(shape=(1000000,), dtype=np.float32)

for i in range(1000000):
    in_buffer[i] = i

start = time.time()
dma.sendchannel.transfer(in_buffer) # send data
dma.recvchannel.transfer(out_buffer) # receive data
dma.sendchannel.wait()
dma.recvchannel.wait()
time.time() - start
```

In my test, the processing time is about **0.034 second**, several times faster than Numpy. But this result is still slower than the result obtained in Vitis (**0.017 second**). The calculation runs in a pipelined mode in hardware. The processing frequency is 100MHz, processing 100000000 elements needs **1000000 / 100M = 0.01 second**. Considering there is redundancy in data transmission, the obtained results in Vitis get closer to ideal situation.

Experiment	Computing time (second)	Improvement
Non-optimized software	2.385	1
Optimized software with Numpy library	0.352	6.775
Hardware accelerator	0.034	70.147

Compare with the acceleration results in Vitis(C/C++):

Experiment	Computing time (second)	Improvement
software (C/C++)	0.104	1
Hardware accelerator	0.017	5.9