

Lab 5 Report

Jeon Woongbae, Park Sangbeen

UNIST

EE32601

PYNQ Access : <http://tpugroupd.iptime.org:777>, PW : xilinx

1 Optimization

Below is the optimization approach.

1.1 Implementation Detail

Implemented source code is available at 5.1.

Overall flow of code is listed as below.

- Receive input and output vectors as AXI_T* type and axis port, receive a and size with s_axilite port.
- Read one element at a time from axis port and move the input into hls::stream<float> with load_input function.
- Compute $aX + Y$ with compute_mul and compute_add, passing hls::stream<float>& and reading one element at a time, which is is FIFO stream.
- Store the result, one element at a time using pop_stream function.

1.2 Optimization Detail

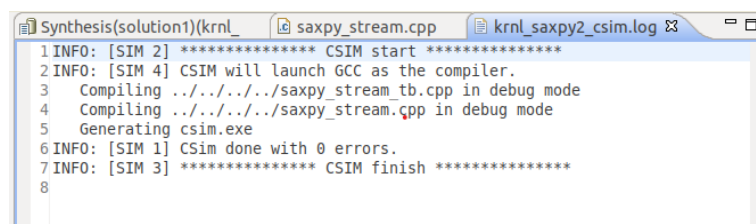
Enabling hls::stream<> is equivalent with a FIFO stream, requires every read and write operations in a sequential order which lead to pipeline approach of read and write.

#pragma HLS INTERFACE PIPELINE II=1 fully utilizes the advantage of hls::stream<> which previously allowed HLS to pipeline all read and write operation sequences.

#pragma HLS INLINE is used in pop_stream and push_stream. These functions are used to convert types between float and AXI_T. By using the inline pragma, it is able to embed the piece of code without generating an additional HW module unlike conventional functions as compute_add.

1.3 Simulation Results

Below is the C simulation result.

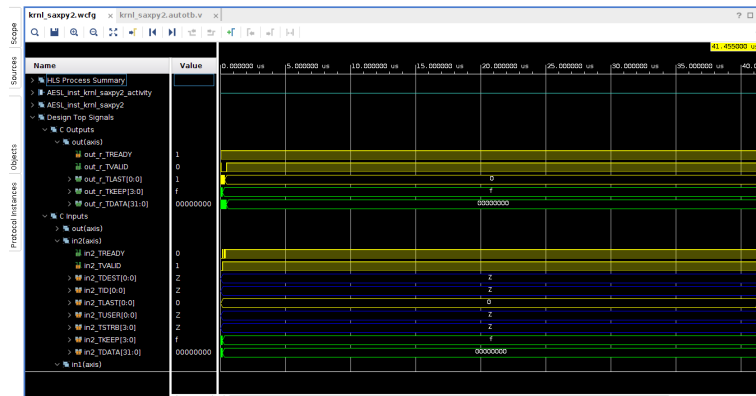


```

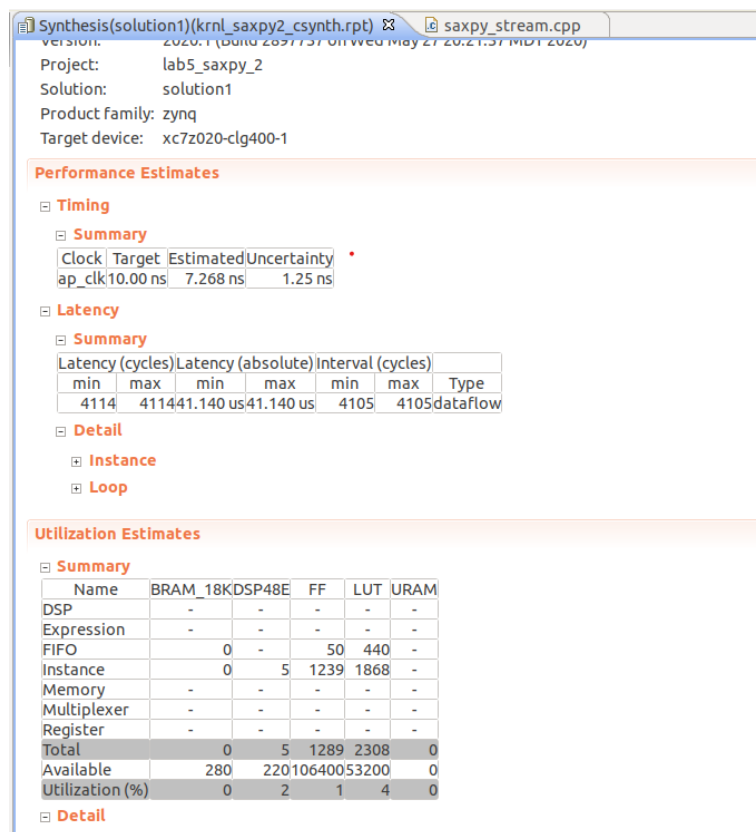
1 INFO: [SIM 2] ***** CSIM start *****
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3 Compiling ../../../../saxpy_stream_tb.cpp in debug mode
4 Compiling ../../../../saxpy_stream.cpp in debug mode
5 Generating csim.exe
6 INFO: [SIM 1] CSim done with 0 errors.
7 INFO: [SIM 3] ***** CSIM finish *****
8

```

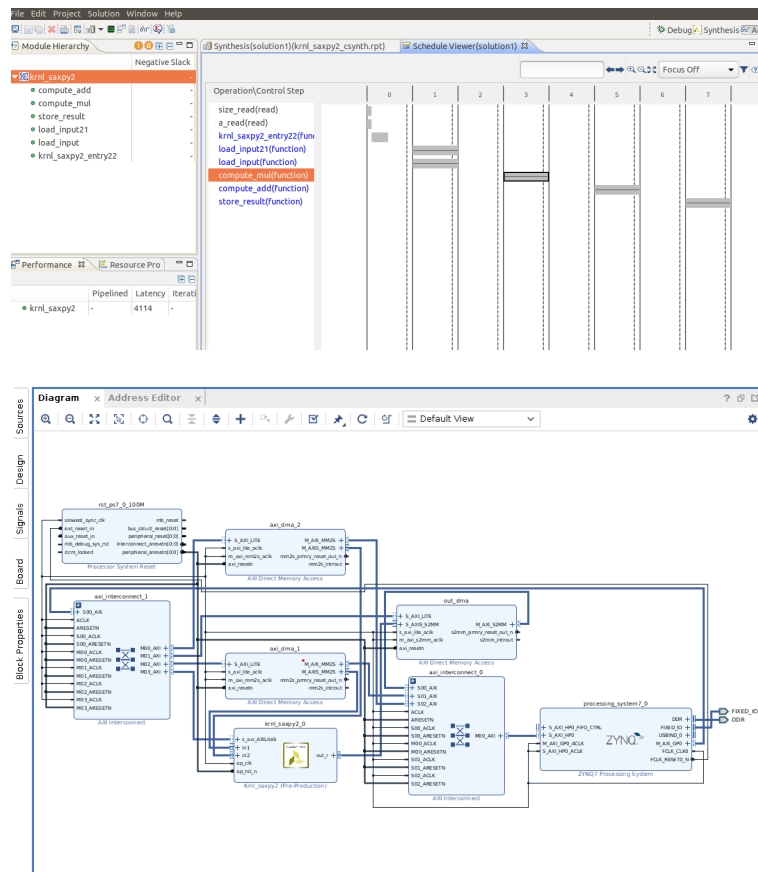
Below is the Cosimulation result.



Below is the C synthesis results.



Below is the top diagram of this SAXPY IP.



2 Python Programming on Pynq

This section briefly explains about key points in Python Pynq programming. Check more details in the submitted Jupyter Notebook, or access our Pynq board via here : <http://tpugroupd.iptime.org:777>.

2.1 Overlay Driver Source Code

Pynq Overlay driver for the SAXPY IP is available at 5.3.

2.2 Use of DMA

SAXPY IP receives its input X and Y from two DMAs and returns its output Z with a single DMA. SAXPY IP transfers its dynamically allocated data array into both DMAS, set an output_dma as a receiving channel, and issue wait() for all 3 DMAs.

2.3 Use of s_axilite port

SIZE and a are sent into SAXPY IP via s_axilite port.

For SIZE and a, the driver reads the address from the register map in the IP itself, and locate those two

registers.

Then, the IP writes the data at the designated addresses, which is equivalent of sending signals to the port.

2.4 Limitations

This driver iterates the DMA issue and wait for 4 times to acquire the correct output, due to the incorrect implementation of floating point operations inside the FPGA board.

Below is the comparison between SW implementation and HW implementation.

Although the latency of SAXPY IP is measured very low at the synthesis phase, the overhead of iterating the DMA issue and wait subroutine for 4 times and accessing the DMA leads to slower actual operation speed of HW.

```
[30]: %%timeit
      SAXPY_HW.run(x,y,a,SIZE)
      516 ms ± 2.34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

[31]: %%timeit
      axpy = np.add(np.multiply(a,x),y)
      227 µs ± 1.58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

3 Bit Stream, Jupyter Notebook, GCD Vivado IP

Bit stream, Jupyter Notebook, GCD Vivado IP are submitted via BB.
Please refer to the files submitted.

4 Contribution

Jeon Woongbae and Park Sangbeen equally contributed.

Hong Junwha and Bu Minsung gave a brief hint about setting 64 bit Zynq HP port settings.

5 Source Codes

5.1 saxpy.cpp

Below is the saxpy_stream.cpp code.

```
#include "saxpy_stream.h"
#define DATA_SIZE 4096

// TRIPCOUNT identifier
const int c_size = DATA_SIZE;

float pop_stream(AXI_T const &e) // type casting AXI_T -> int
{
#pragma HLS INLINE

    union conv {
        float f;
        unsigned int u;
    };

    conv tmp;
    tmp.u = e.data;

    float ret = tmp.f;
    volatile ap_uint<4> keep = e.keep; // -1 : 1111
    volatile ap_uint<1> last = e.last; // only asserted at the last element
    return ret;
}

AXI_T push_stream(float const ret, int last) // type casting AXI_T -> int
{
#pragma HLS INLINE

    union conv {
        float f;
        unsigned int u;
    };

    conv tmp;
    tmp.f = ret;

    AXI_T e;
```

```

    e.data = tmp.u;
    e.keep = -1;           // 1111
    e.last = last ? 1 : 0; // 1 when last
    return e;
}

```

```

static void load_input(AXI_T* in, STREAM_FLOAT& inStream, int size) {
mem_rd:
    for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
#pragma HLS PIPELINE II=1
        inStream.write(pop_stream(in[i]));
    }
}

```

```

static void compute_mul(STREAM_FLOAT& x_stream,
                        STREAM_FLOAT& ax_stream,
                        float a,
                        int size) {
// The kernel is operating with vector of NUM_WORDS integers. The + operator performs
// an element-wise add, resulting in NUM_WORDS parallel additions.
execute:
    for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
#pragma HLS PIPELINE II=1
        ax_stream.write(x_stream.read()*a);
    }
}

```

```

static void compute_add(STREAM_FLOAT & ax_stream,
                        STREAM_FLOAT & y_stream,
                        STREAM_FLOAT & z_stream,
                        int size) {
// The kernel is operating with vector of NUM_WORDS integers. The + operator performs
// an element-wise add, resulting in NUM_WORDS parallel additions.
execute:
    for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
#pragma HLS PIPELINE II = 1
        z_stream.write(ax_stream.read() + y_stream.read());
    }
}

```



```

    }
}

static void store_result(AXI_T* out, STREAM_FLOAT& out_stream, int size) {
mem_wr:
    for (int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_size max = c_size
#pragma HLS PIPELINE II=1
        out[i] = push_stream(out_stream.read(), i == size - 1);
    }
}

```

```

void krnl_saxpy2(AXI_T* in1, AXI_T* in2, AXI_T* out, float a, int size) {
#pragma HLS INTERFACE axis port = in1 depth = 4096
#pragma HLS INTERFACE axis port = in2 depth = 4096
#pragma HLS INTERFACE axis port = out depth = 4096
#pragma HLS INTERFACE s_axilite port = size
#pragma HLS INTERFACE s_axilite port = a
#pragma HLS INTERFACE ap_ctrl_none port=return

```

```

static STREAM_FLOAT x_stream("input_stream_x");
static STREAM_FLOAT y_stream("input_stream_y");
static STREAM_FLOAT z_stream("z_stream");
static STREAM_FLOAT ax_stream("input_stream_ax");

```

```

#pragma HLS dataflow
    // dataflow pragma instruct compiler to run following three APIs in parallel
    load_input(in1, x_stream, size);
    load_input(in2, y_stream, size);
    compute_mul(x_stream, ax_stream, a, size);
    compute_add(ax_stream, y_stream, z_stream, size);
    store_result(out, z_stream, size);
}

```

5.2 saxpy testbench

Below is the saxpy_stream_tb.cpp test bench code.

```

#define SZ 4096

#include <stdio.h>
#include "saxpy_stream.h"

```

```
int main(){
    /*
    float a[SZ];
    float b[SZ];
    float c[SZ];
    float d = 1.9;

    for(int i=0;i<SZ;++i){
        a[i] = 1.0*(i*1.0);
        b[i] = 2.0*(i*2.0);
    }

    krnl_saxpy(a,b,c,int(SZ));

    int ret=0;
    for(int i=0;i<SZ;++i){
        if(c[i]!=(a[i]+b[i])){
            ret=1;
            break;
        }
    }

    return ret;
    */

    AXI_T a[SZ];
    AXI_T b[SZ];
    AXI_T c[SZ];
    float d = 2.0;
    //AXI_T d[SZ];
    float x = 0.15625;
    for(int i=0;i<SZ;++i){
        a[i].data = x;
        a[i].keep = -1;
        a[i].last = (i == SZ - 1 ? 1 : 0);

        b[i].data = x;
        b[i].keep = -1;
        b[i].last = (i == SZ - 1 ? 1 : 0);

        //d[i].data = 1.0;
```

```

        //d[i].keep = -1;
        //d[i].last = (i == SZ - 1 ? 1 : 0);
    }

    krnl_saxpy2(a,b,c,d,int(SZ));

    int ret=0;
    for(int i=0;i<SZ;++i){
        if(c[i].data!=(a[i].data*d + b[i].data)){
            ret=1;
            break;
        }
    }
}

```

5.3 Overlay Python Code

Below is the Overlay for SAXPY IP.

```

class saxpy_overlay(Overlay):
    def __init__(self, bitfile_name, SIZE=4096):
        super().__init__(bitfile_name)
        self.saxpy_hw = self.krnl_saxpy2_0
        self.abuf = allocate((SIZE, ), dtype=np.float32)
        self.bbuf = allocate((SIZE, ), dtype=np.float32)
        self.obuf = allocate((SIZE, ), dtype=np.float32)

    def python_float_to_uint(self, num):
        return ctypes.c_uint.from_buffer(ctypes.c_float(num)).value

    def register_map(self):
        return self.register_map

    def ip_dict(self):
        return self.saxpy_hw.ip_dict()

    def run(self, x, y, a, SIZE):
        np.copyto(self.abuf, x)
        np.copyto(self.bbuf, y)
        self.saxpy_hw.write(saxpy_hw.register_map.size.address, SIZE)
        self.saxpy_hw.write(saxpy_hw.register_map.a.address, self.python_float_to_uint

```

```
#print(self.saxpy_hw.read(saxpy_hw.register_map.size.address))
#print(self.saxpy_hw.read(saxpy_hw.register_map.a.address))

#print(self.abuf, self.bbuf, a, SIZE)

for i in range(4):
    self.axi_dma_1.sendchannel.transfer(self.abuf)
    self.axi_dma_2.sendchannel.transfer(self.bbuf)
    self.out_dma.recvchannel.transfer(self.obuf)
    self.axi_dma_1.sendchannel.wait()
    self.axi_dma_2.sendchannel.wait()
    self.out_dma.recvchannel.wait()

#print(self.obuf)
return self.obuf

def assert_by_sw(self, x, y, a, SIZE):
    ax = np.multiply(a, x)
    axpy = np.add(ax, y)
    axpy_hw = self.run(x, y, a, SIZE)
    comparison = np.isclose(axpy, axpy_hw)
    for i in comparison:
        assert(i == True)

return True
```

6 Bibliography

References