Synthesizing a streaming kernel

Gökçe Aydos

Example multiplier mult_stream.cpp

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32,0,0,0> pkt_t;

void mult_stream(
        hls::stream< pkt_t > &din,
        hls::stream< pkt_t > &dout,
        ap_int<32> multiplier) {
    #pragma HLS INTERFACE axis port=din
    #pragma HLS INTERFACE axis port=dout
    #pragma HLS INTERFACE axi_lite port=multiplier
    #pragma HLS INTERFACE ap_ctrl_none port=return
    pkt_t pkt;
    din.read(pkt);
    pkt.data *= multiplier;
    dout.write(pkt);
}
```

Example multiplier - Explanation

In contrast to a memory-mapped interface, a streaming interface is used between dedicated producers and consumers. We do not need addressing in this case and thus less resources for communication.

A stream of data should be consumed in the same cycle when the data arrives, so it behaves like a first-in first out (FIFO) buffer. To guarantee FIFO semantics in code, Vitis framework defines the datatype `hls::stream<T>` (from `hls_stream.h`) for streaming. [1] states: "Once the data has been read from an `hls::stream` it no longer exists in the stream."

The methods `read()` and `write()` read and write a single AXI packet (`pkt_t`).

[1] Vitis HLS coding styles — Using HLS streams for streaming data

As encapsulated datatype `T` the unsigned `ap_axiu<DATA_WIDTH, WUser, WId, WDest>` or the signed variant `ap_axis<...>` (from `ap_axi_sdata.h`) can be used. Template parameters other than `DATA_WIDTH` are used for tagging the stream packets. These are also called *side channel signals*. For example if our kernel consumes 32 bit unsigned integers without side channels, we use `ap_axiu<32, 0, 0 ,0>` [2].

The `ap_axiu<...>` can be seen as an AXI packet, which contains the data (`pkt_t.data`) and the side-channel signals that we do not use.

An example with `hls::stream<T>` and `ap_axiu<...>` is provided in [3].

[2] How AXI4-stream is implemented

[3] AXI4-Stream interfaces without side-channels

To conveniently change the `multiplier` value through a register we specify:

```
#pragma HLS INTERFACE axi_lite port=multiplier
```

which will make `multiplier` available in one of the control registers of the AXI-lite adapter.

If our kernel should continuously work without any control signals like `ap_start`, then we can get rid of the control signals using `ap_ctrl_none` attribute on the `return` port [4]:

```
#pragma HLS interface ap_ctrl_none port=return
```

[4] Coding guidelines for free-running kernels

The pass-by-value argument `multiplier` will be synthesized using the `ap_none` protocol, which does not use any handshaking signals for data update [5]. `ap_none` ports can typically be updated using a control register connected via AXI-lite.

[5] Port-Level control protocols

Synthesis results of the multiplier:

After the synthesis in Vitis, we see the following hardware description in synthesis summary:

HW interfaces:

- the AXI-lite interface (`S_AXILITE`) `s_axi_control`, because we specified that we would like to interact with the data `multiplier` using the AXI-lite adapter, i.e., control register.
- two AXI stream interfaces `din` and `dout` with 32 bit data width including `TKEEP`, `TREADY`, `TSTRB`, and `TVALID` signals
- one 32 bit register `multiplier` using `ap_none` protocol
- top level control signals including the default `ap_clk` and `ap_rst_n` signals without any control register (`ap_ctrl_none` protocol)

SW I/O shows the datatypes used in the software kernel interface (`top function arguments`) and how these map to the HW:

- `din` and `dout` were mapped with the same name. These are `interfaces` in Vitis nomenclature.
- `multiplier` is a simple port.

A `port` is a signal (in VHDL) or wire (in Verilog) used at the interface of an entity (VHDL) or module (Verilog). An `interface` is a bundle of signals like AXI stream or AXI memory-mapped interface.

Connecting the kernel to the CPU

After we export the hardware description as Vivado IP, we can connect it to the CPU. In Vivado we instantiate a processor and add the kernel. The CPU does not support streaming but only memory-mapping, so we have to use a streaming aware component which can translate between memory-mapped (MM) and streaming interface. `AXI direct memory access` IP is sufficient for our task.

After instantiating the DMA we can change its name to `axi_dma` because we will only have a single DMA IP in our design. To keep our design simple we deactivate the `scatter gatter engine` in the DMA settings.

We connect the data output `dout` of our IP to the AXI stream (AXIS) streaming-to-memory-mapped (S2MM) port, and the data input `din` to the AXIS MM2S port.

The CPU has only an AXI memory-mapped master interface as default. To receive the data from our IP we add additionally an high-performance (HP) slave interface. This data sink allows data transfer to the RAM and will be driven by the AXI interconnect.

The designer assistance can deal with the rest of the connections. In the first use the `connection automation` creates an AXI interconnect and interconnects the high-performance CPU port and DMA related resets, clocks, and AXI interfaces. In the second try an additional AXI interconnect port is created to connect the remaining `M_AXI_S2MM` port of the DMA.

We can optionally connect the fixed ports of the CPU. They are hardwired, and the tool won't bother if we do not connect.

Now we can begin with bitstream generation.

Testing the design

Pay attention that you change the names according to your design!

```
from pynq import Overlay
overlay = Overlay('multiplier.bit')

import pynq.lib.dma
dma = overlay.axi_dma
multiply = overlay.mult_stream

multiply.register_map
```

should output:

```
RegisterMap {
  multiplier = Register(multiplier=0)
}
```

```
from pynq import allocate
import numpy as np

in_buffer = allocate(shape=(6,), dtype=np.uint8)
out_buffer = allocate(shape=(6,), dtype=np.uint8)

for i in range(6):
    in_buffer[i] = i

multiply.register_map.multiplier = 3

dma.sendchannel.transfer(in_buffer)
dma.recvchannel.transfer(out_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```

```
out_buffer
```

should output:

```
PynqBuffer([ 0,  3,  6,  9, 12, 15], dtype=uint8)
```