

ESE532: System-on-a-Chip Architecture

Day 13: October 19, 2020
Vitis OpenCL
Data Transfer Model



Previously

- Value of small, local memories
- Tune local memories in C
- DMA as data-movement threads

Today

Vitis/OpenCL Interface

- Execution Model (Part 1)
- Host-Side Programming (Part 2)
- FPGA-Side Programming (Part 3)

Message

- Vitis uses an explicit DMA transfer interface
- Can build our streaming model on top of primitives offers
- Will need to tune parameters and use carefully to get maximize performance
 - Good and bad → largely exposed

Note

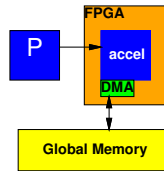
- OpenCL interface in Vitis is new
- In past SDSoC took care of some of this
- Still figuring out how to use interfaces to get the implementations that we know are possible on the FPGA
 - A few things in lecture will be more tentative
 - HW6 experimentation and delay

Execution Model(s)

Part 1

Model

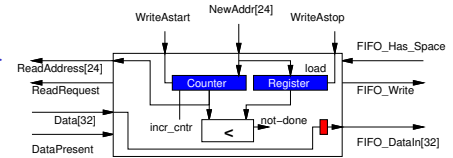
- Host and Accelerator
- Host control
 - Set accelerator arguments
 - Tell it to start
 - DMA bulk data to accelerator
 - Run as independent thread
 - Synchronize on data return



Remember: Preclass 4a

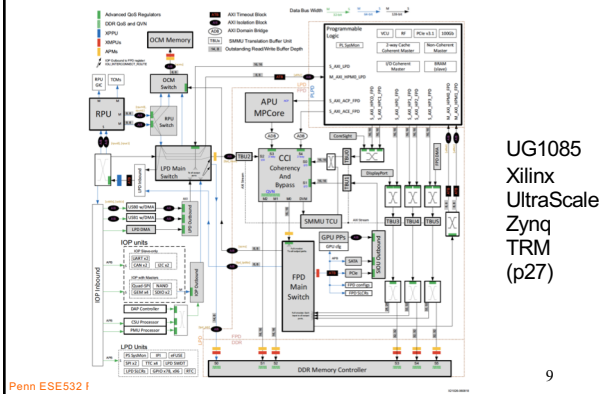
```
P1: for(i=0;i<MAX;i++) Astream.write(a[i]);
Set arguments to accelerator
```

Accelerator Uses to grab inputs



```
int *p;
P1: for(p=&(a[0]);p<&(a[MAX]);p++) Astream.write(*p);
```

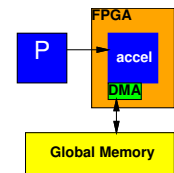
Programmable SoC



UG1085
Xilinx
UltraScale
Zynq
TRM
(p27)

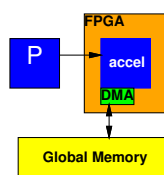
Function Call

- Host and Accelerator
- Host control
 - Set accelerator arguments
 - Tell it to start
 - DMA bulk data to accelerator
 - DMA result back
 - Host program wait on accelerator completion

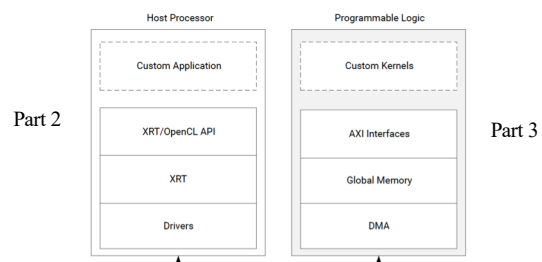


Separate Accelerator Thread

- Host and Accelerator
- Host control
 - Set accelerator arguments
 - Tell it to start
 - DMA bulk data to accelerator
 - DMA result back
 - Host execute something else
 - Host program synchronize before use accelerator result

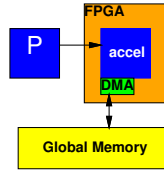


PS<->PL Interface



Minimal

- Simple case one host process, one accelerator

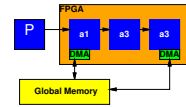


Penn ESE532 Fall 2020 -- DeHon

13

Accelerator Stream

- Dataflow stream accelerators on host side

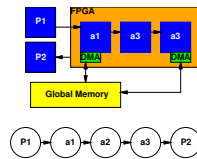


Penn ESE532 Fall 2020 -- DeHon

14

Streaming

- Map general streaming
- Use multiple threads
 - Even processors

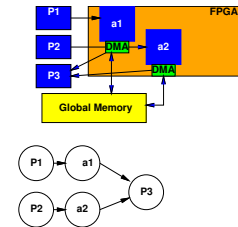


Penn ESE532 Fall 2020 -- DeHon

15

Streaming

- Map general streaming
- Use multiple threads
 - Even processors
- ... and multiple accelerators
- Given space, any streaming case
 - DMA buffers cross PS<->PL

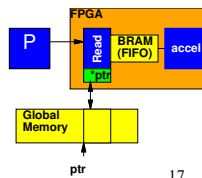
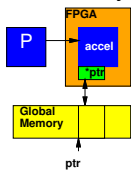


Penn ESE532 Fall 2020 -- DeHon

16

Data Movement (Options)

- Access from global memory
- Configure DMA/accelerator with main-memory pointer
- DMA copy into BRAM memories
 - Built on top of other with DMA copy loop

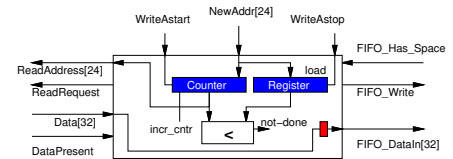


Penn ESE532 Fall 2020 -- DeHon

17

Remember: Preclass 4a

```
P1: for(i=0;i<MAX;i++) Astream.write(a[i]);
```



```
int *p;
P1: for(p=&(a[0]);p<&(a[MAX]);p++) Astream.write(*p);
```

Penn ESE532 Fall 2020 -- DeHon

18

Host-Side Programming

Part 2

Outline

- Platform
- Device
- Context
- Command Queue
- Load FPGA
- Buffers and Transfer
- Kernel Execution
- Synchronize on Result
- Overlap Data and Compute
- Running Multiple Kernels

Platform

```
cl_platform_id platform_id; // platform id
err = clGetPlatformIDs(16, platforms, &platform_count);
// Find Xilinx Platform
for (unsigned int iplat=0; iplat<platform_count; iplat++) {
    err = clGetPlatformInfo(platforms[iplat],
        CL_PLATFORM_VENDOR,
        1000,
        (void *)cl_platform_vendor,
        NULL);
    if (strcmp(cl_platform_vendor, "Xilinx") == 0) {
        // Xilinx Platform found
        platform_id = platforms[iplat];
    }
}
```

- For general case where have multiple OpenCL platforms
- Silly for Ultra96 SoC
 - and, looks like we can partially hide

Device

```
cl_device_id devices[16]; // compute device id
char cl_device_name[100];
err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,
    16, devices, &num_devices);
printf("INFO: Found %d devices\n", num_devices);
//iterate all devices to select the target device.
for (uint i=0; i<num_devices; i++) {
    err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 1024, cl_device_name,
    0);
    printf("CL_DEVICE_NAME %s\n", cl_device_name);
}
```

- For case of multiple FPGAs → which FPGA
- Also somewhat silly for Ultra96 with single FPGA (PL)
- F1.x16large has multiple FPGAs

Device

- Our Utilities.cpp,h – hide Platform

```
std::vector<cl::Device> devices =
get_xilinx_devices();
devices.resize(1);
cl::Device device = devices[0];
```
- Get away with simplification
 - Know what only Xilinx platform
 - Know only one device (F1.x2large, Ultra96)

Context

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

- Context for device
 - Not sure what flexibility/differentiation this is giving
 - Supports some error callback
 - Maybe just for compatibility with OpenCL

Command Queue

```
// Out-of-order Command queue
commands = clCreateCommandQueue(context, device_id,
    CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

// In-order Command Queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

- Setup queue to issue commands
- For “programming” interactions with accelerators
- Use to set them up to happen as separate thread

Load Bitstream on FPGA

```
int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
    &size_var, (const unsigned char **) &kernelbinary,
    &status, &err);
```

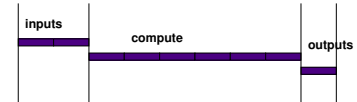
- Where we actually reconfigure FPGA
 - Option to choose which to load
 - Option to reconfigure during operation under host control
 - Probably not one we’ll exercise this term

Kernels

```
kernel1 = clCreateKernel(program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err);
```

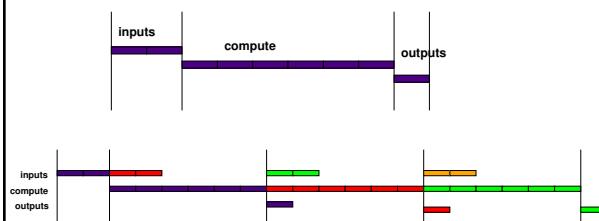
- Setup the actual accelerators to call
 - Extract handle to reference them from the loaded bitstream (program)
 - Just naming them at this point

Preclass 1

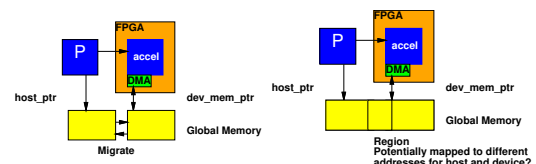


- Kernel with
 - 10ms input transfer
 - 30ms compute
 - 5ms outputs
- Latency?
- Throughput if must serialize?
- Throughput if overlap data transfers and computation?

Preclass 1



Host and Device Memory View



- Host pointers not necessarily meaningful to device
 - Need to map
 - Different memories? (may depend on platform?)
 - Virtual vs. Physical addresses?
 - Same memory, different address map?

Buffers and Transfer

```

// Two cl_mem buffer, for read and write by kernel
cl_mem dev_mem_read_ptr = clCreateBuffer(context,
    CL_MEM_READ_ONLY,
    sizeof(int) * number_of_words, NULL, NULL);

cl_mem dev_mem_write_ptr = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY,
    sizeof(int) * number_of_words, NULL, NULL);

// Setting Arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_read_ptr);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_mem_write_ptr);

// Get Host side pointer of the cl_mem buffer object
auto host_write_ptr =
clEnqueueMapBuffer(queue, dev_mem_read_ptr, true, CL_MAP_WRITE, 0, nullptr,
    &err);
auto host_read_ptr =
clEnqueueMapBuffer(queue, dev_mem_write_ptr, true, CL_MAP_READ, 0, nullptr,
    &err);

// Fill up the host_write_ptr to send the data to the FPGA
for(int i=0; i<MAX; i++) {
    host_write_ptr[i] = <... >
}

// Migrate
clMem mems[2] = {host_write_ptr, host_read_ptr};
clEnqueueMigrateMemObjects(queue, 2, mems, 0, 0, nullptr, &err);

// Schedule the kernel
clEnqueueTask(queue, kernel, 1, &err);

// Migrate data back to host
clEnqueueMigrateMemObjects(queue, 1, &dev_mem_write_ptr,
    CL_MIGRATE_MEM_OBJECT_HOST, 1, &err);

Per
clWaitForEvent(1, &dev_mem_read_ptr);
    
```

Create
Buffers

Set kernel
pointers

Host side pointers

Fill with host data

Launch data transfer
(enqueue)

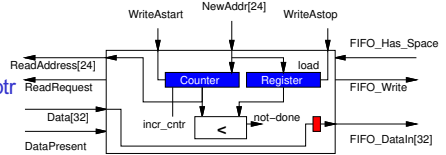
31

Remember: Preclass 4a

```

P1: for(i=0; i<MAX; i++) Astream.write(a[i]);
    clSetKernelArg
    
```

Use
dev_mem_ptr



```

int *p;
P1: for(p=&(a[0]); p<&(a[MAX]); p++) Astream.write(*p);
    
```

Penn ESE532 Fall 2020 -- DeHon

32

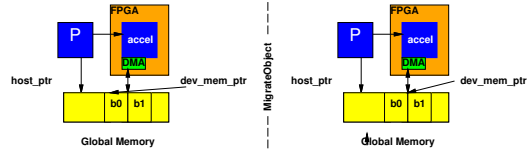
Movement Options

- `clEnqueueReadBuffer` (`WriteBuffer`)
 - Not guarantee timing of transfer
- `clEnqueueMigrateObjects`
 - Supports overlap of compute and communicate
 - recommended

Penn ESE532 Fall 2020 -- DeHon

33

Remap Device Pointer



- Migration remap dev_mem_ptr
- (don't know if this is happening this way)

Penn ESE532 Fall 2020 -- DeHon

34

Kernel Invocation

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

- After
 - Kernel arguments set
 - Buffer transfers have been enqueued
- Causes to actual run on data transferred

Penn ESE532 Fall 2020 -- DeHon

35

Synchronize

```

q.enqueueMigrateMemObjects({in1_buf, in2_buf},
    0, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
q.enqueueTask(krnl_mmult, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
    
```

- Transfers, kernel concurrently
- Must synchronize to know finished

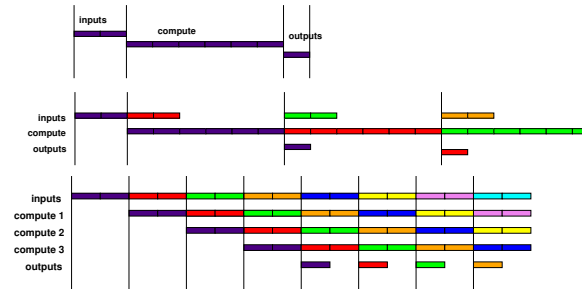
Penn ESE532 Fall 2020 -- DeHon

36

Preclass 2a

- Break kernel into there 10ms pieces and use dataflow between them
 - 10ms input transfer
 - 10ms compute x 3
 - 5ms output transfer
- **Throughput?**

Preclass 1 and 2



Dataflow: Host-to-kernel

- Create a dataflow stream between host process and accelerator
 - Like DATAFLOW pragma
 - Control on FPGA/kernel side (coming up)
- Overlap next computation (setup) on host with FPGA kernel

- **USG1393 says embedded platforms not support?**

Preclass 2b

- How many additional inputs send before get associated output? (pipeline depth)



Kernel Completion/ Synchronization



- Don't have to synchronize right after an operation
- Can have several in flight (in pipeline)
- Synchronize only after sending several (pipeline depth) inputs

Kernel Completion/ Synchronization

Example, slightly simpler

- Don't have to synchronize right after an operation
- Can start several kernel invocations
- Defer wait until later

Run Multiple Tasks

```
for (int i=0; i<num_iter; i++) {
    q.enqueueMigrateMemObjects({subbuf_doc_
words[i]}, 0, &wordWait, &buffDone);
wordWait.push_back(buffDone);
    q.enqueueTask(kernel, &wordWait,
&krnlDone); krnlWait.push_back(krnlDone);
    q.enqueueMigrateMemObjects({subbuf_inh_
flags[i]}, CL_MIGRATE_MEM_OBJECT_HOST,
&krnlWait, &flagDone);
    flagWait.push_back(flagDone);
}
// then synchronize
for (int i=0; i<num_iter; i++) {
    flagWait[i].wait(); }
```

Penn ESE532 Fall 2020 -- DeHon

43

Overlapping Data and Compute

- Use control of when synchronize to overlap data and compute
- Start (queue up) next data transfer and invocation before dataflow pipeline finishes
- Need to reason about (or experiment with) how many to in flight
 - When to synchronize

Penn ESE532 Fall 2020 -- DeHon

44

Running Multiple Kernel Instances

- Can have multiple instances of same kernel on FPGA
- Dispatch from Out-of-Order command queue
- Will distribute among identical kernels

Penn ESE532 Fall 2020 -- DeHon

45

Parallelism

- Pipeline/Dataflow
 - With Host Dataflow
- Task Parallelism
 - Launch multiple accelerator
- Data Parallelism
 - Define multiple instances of one accelerator and enqueue work for

Penn ESE532 Fall 2020 -- DeHon

46

Cleanup

```
clReleaseCommandQueue(Command_Queue);
clReleaseContext(Context);
clReleaseDevice(Target_Device_ID);
clReleaseKernel(Kernel);
clReleaseProgram(Program);
free(Platform_IDs);
free(Device_IDs);
```

- Need to release/free resources setup

Penn ESE532 Fall 2020 -- DeHon

47

FPGA-Side Programming

Part 3

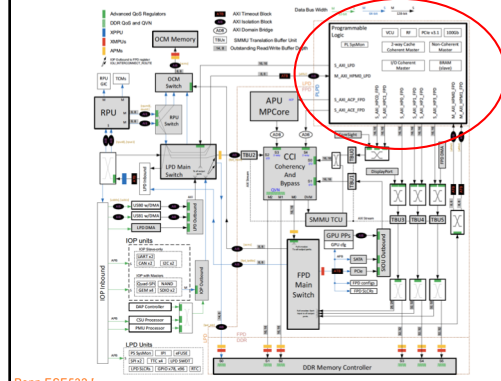
Penn ESE532 Fall 2020 -- DeHon

48

Interface Pragmas

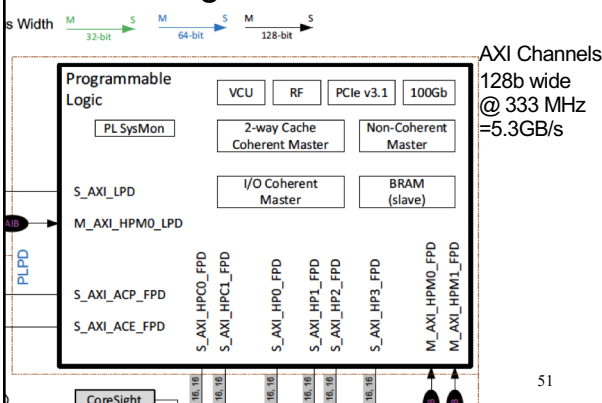
- Assign AXI
 - Bundles
- Host-to-Kernel
- Burst read
- Interface width

Programmable SoC



UG1085
Xilinx
UltraScale
Zynq
TRM
(p27)

Programmable SoC



AXI Channels

```
void cnn( int *pixel, // Input pixel
         int *weights, // Input Weight Matrix
         int *out, // Output pixel
         ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

- Can tell it to use AXI Channels
- By default (as shown above) all share same channel

AXI Channels

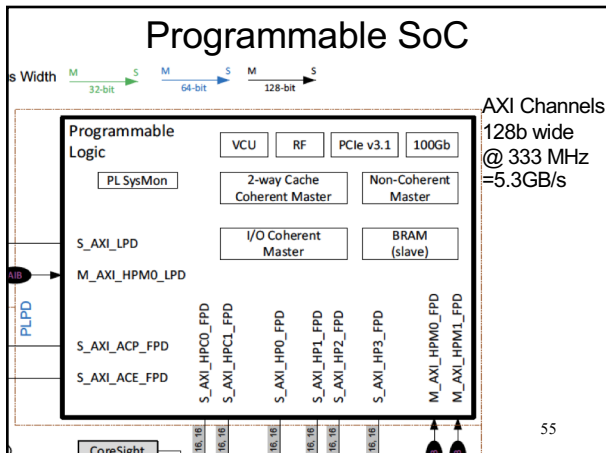
```
void cnn( int *pixel, // Input pixel
         int *weights, // Input Weight Matrix
         int *out, // Output pixel
         ... // Other input or Output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
```

- Separate names to use different
- Note, not naming to match ZCU3EG names
 - Portability across targets

Older and Defaults

```
• 2019.2:
#pragma HLS INTERFACE s_axilite port = in1 bundle = control
#pragma HLS INTERFACE s_axilite port = in2 bundle = control
#pragma HLS INTERFACE s_axilite port = out_r bundle = control
#pragma HLS INTERFACE s_axilite port = size bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control
#pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out_r offset=slave bundle=gmem
• 2020.1:
#pragma HLS INTERFACE m_axi port=in1 bundle=gmem
#pragma HLS INTERFACE m_axi port=in2 bundle=gmem
#pragma HLS INTERFACE m_axi port=out_r bundle=gmem
```



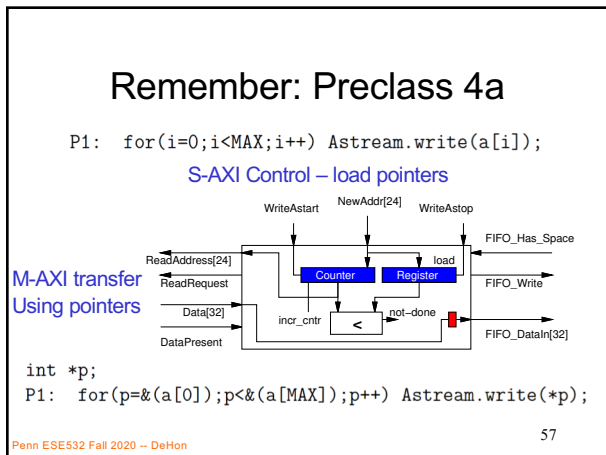
Older and Defaults

- 2019.2:


```
#pragma HLS INTERFACE s_axilite port = in1 bundle = control
#pragma HLS INTERFACE s_axilite port = in2 bundle = control
#pragma HLS INTERFACE s_axilite port = out_r bundle = control
#pragma HLS INTERFACE s_axilite port = size bundle = control
#pragma HLS INTERFACE s_axilite port = return bundle = control
#pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out_r offset=slave bundle=gmem
```
- 2020.1:


```
#pragma HLS INTERFACE m_axi port=in1 bundle=gmem
#pragma HLS INTERFACE m_axi port=in2 bundle=gmem
#pragma HLS INTERFACE m_axi port=out_r bundle=gmem
```

56



Host-to-kernel Dataflow

```
void kernel_name( int *inputs,
    ... )// Other input or Output ports
{
#pragma HLS INTERFACE ..... // Other interface pragmas
#pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
```

- ap_control_chain

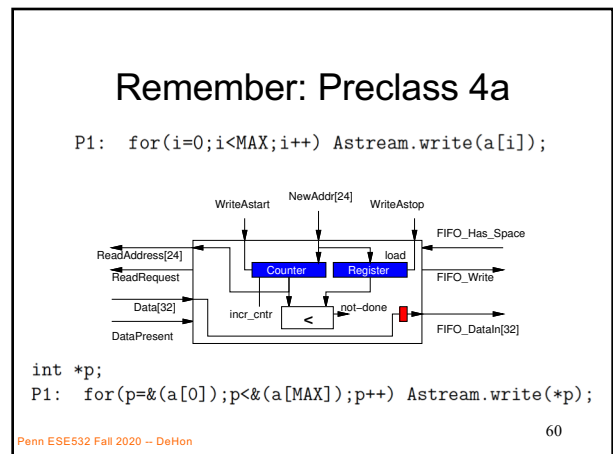
58

Burst Transfer

```
hls::stream<datatype_t> str;
INPUT_READ: for(int i=0; i<INPUT_SIZE; i++) {
#pragma HLS PIPELINE
str.write(inp[i]); // Reading from Input interface
}
```

- Where have we seen this before?

59



Burst Transfer

```

top_function(datatype_t * m_in, // Memory data Input
             datatype_t * m_out, // Memory data Output
             int inp1, // Other Input
             int inp2) { // Other Input
#pragma HLS DATAFLOW

hls::stream<datatype_t> in_var1; // Internal stream to transfer
hls::stream<datatype_t> out_var1; // data through the dataflow region

read_function(m_in, inp1, in_var1); // Read function contains pipelined for
loop // to infer burst

execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

write_function(out_var1, m_out); // Write function contains pipelined for
loop // to infer burst
}
    
```

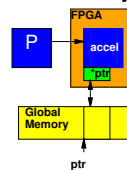
- Use Dataflow to setup as separate threads on accelerator side
- Believe this is way to burst copy into local memory

Penn ESE532 Fall 2020 -- DeHon

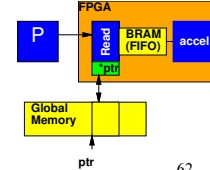
61

Data Movement (Options)

- Access from global memory
- Configure DMA/accelerator with main-memory pointer
- DMA burst copy into BRAM memories
 - Built on top of other with DMA copy loop



Penn ESE532 Fall 2020 -- DeHon



62

Preclass 3, 4

- 333MHz cycle time on bus
- Peak bandwidth if transfer per cycle
 - 32b
 - 64b
 - 128b
- Memory at 4 GB/s
 - Which widths make bus the bottleneck?
 - Memory?

Penn ESE532 Fall 2020 -- DeHon

63

Width

```

void cnn( ap_uint<512> *pixel, // Input pixel
          int *weights, // Input Weight Matrix
          ap_uint<512> *out, // Output pixel
          ... // Other input or output ports

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
    
```

- Will transfer at width of data type give
- May need to pack and unpack data to exploit full width

Penn ESE532 Fall 2020 -- DeHon

64

Big Ideas

- Data between PS and PL managed as DMA
- Can exploit familiar parallelism
 - Thread, Data, Streaming Dataflow
 - Overlap compute and communicate
- Must manage some pieces pretty explicitly

Penn ESE532 Fall 2020 -- DeHon

65

Admin

- Feedback
- Reading for Wednesday on web
- HW6
 - Out soon?
 - Delayed...as mentioned earlier new interface
 - Figuring out what that means about deadlines...
 - Uses Ultra96
 - At least one partner has Ultra96

Penn ESE532 Fall 2020 -- DeHon

66