High Performance
Computing &
Big Data Services

hpc.uni.lu

hpc@uni.lu

@ULHPC

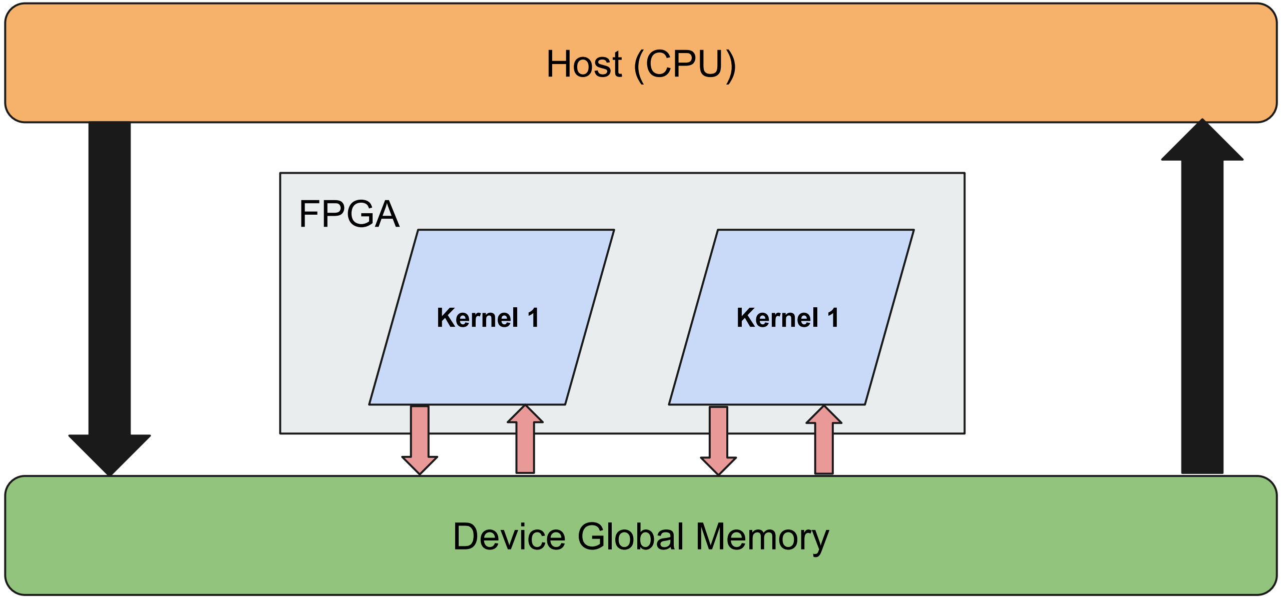
Channels and Pipes with OpenCL

Emmanuel Kieffer

EUMaster4HPC

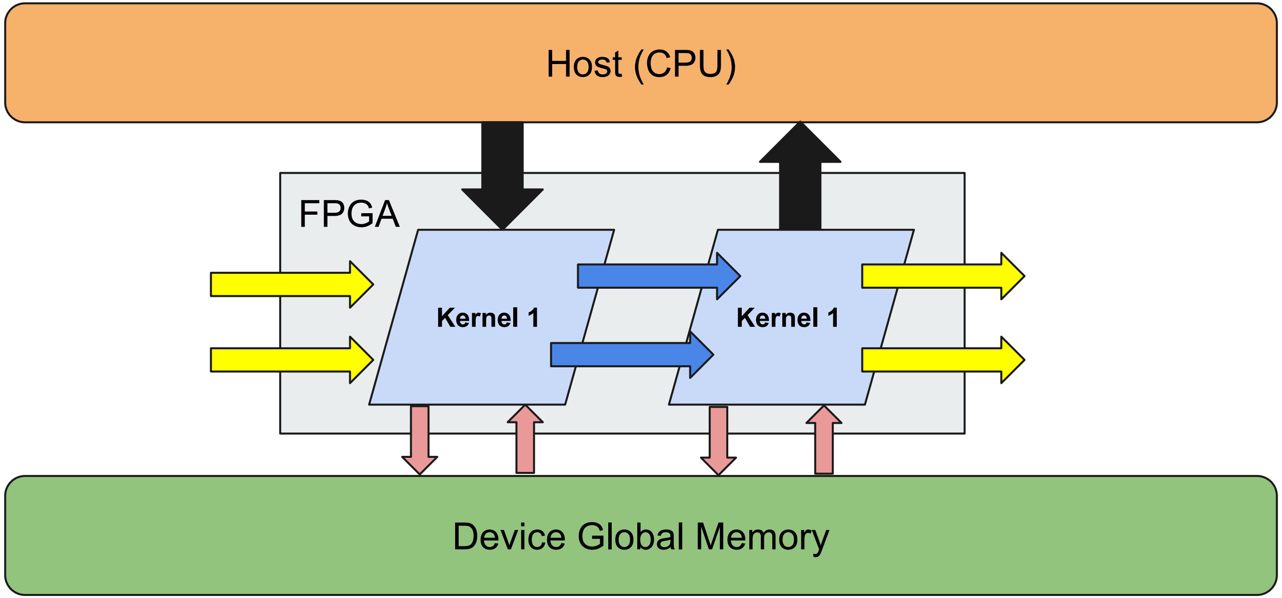
Why using Channels/Pipes ?

Traditional communication scheme



- All communications go through global memory
- Can quickly become a bottleneck

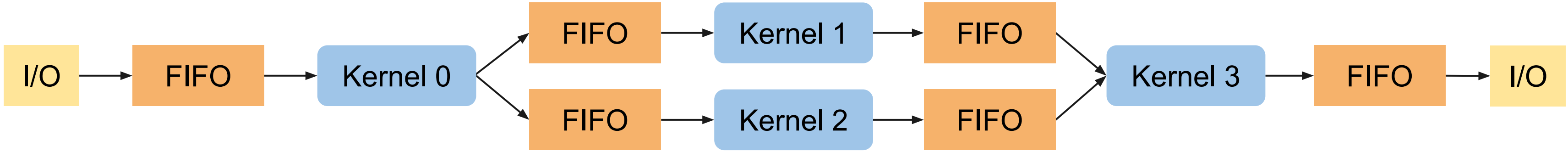
Channels



- Kernel-to-kernel communication
- Io-to-kernel communication
- Host to kernel communication avoiding global memory

} Independent from host

FIFO memory



- FIFO memory allows efficient community
- Bypass Global Memory
- Streaming data application
- Channels/Pipe
    - FIFO-like communication
    - Call site is unidirectionnel (no default duplex communication)
    - Allows BSP-specific I/O communication with kernels compute units

Advantages

- Internal bandwidth of the FPGA hardware
- No bottleneck of using off-chip memory
- Less latency with concurrent kernel execution
- Data is consumed as it is produced which reduced storage requirements
- With Channel/Pipe communication, the hosts can launch kernels in parallel providing Performance Gains

IO Channels

- Classic communication scheme
  - Data need to be written to global memory before kernel get access to it
  - The bandwidth is limited by the PCIe bandwidth and the memory throughput
- With IO channels
  - Kernels send directly data across network interface
  - System is running at speed of network interface

How to use Channels ?

UNIVERSITÉ DU LUXEMBOURG

Channels creation

- Channels are specific to Intel FPGA
- Should be declared in OpenCL files (*.cl)
- Enable Intel FPGA extension of channels : `#pragma OPENCL EXTENSION cl_intel_channels : enable`
- Any built-in OpenCL or user defined type are supported
  - Structs, char, uchar, short, int, uint, long, ulong, float, vector data types
  - Type must be 1024 bits or less
- Channel FIFO depth can be defined

```
channel long a; // unbuffered channel a of type `long`
channel long b __attribute__((depth(8))); // buffered channel b;
channel float4 c[2]; // Channels made up of 2 float4 channels, c[0] and c[1]
```

Channels reads and write

- Read and writes data from channels are blocking functions
- Each write add a single piece of data to the channel
  - `void write_channel_intel(channel <type> channel_id, const <type> data);`
  - Ex: `write_channel_intel(channel a, (float4) vector);`
- Each read remove a single piece of data from the channel
  - `<type> read_channel_intel(channel <type> channel_id);`
  - Ex: `read_channel_intel(channel b);`
- Write blocks when the channel is fully
- Read blocks when the channel is empty

Non-blocking versions

```
bool write_channel_nb_intel(channel <type> channel_id, const <type> data);
<type> read_channel_nb_intel(channel <type> channel_id, (float4) vector);
```

- Same functions except that the functions do not block and the pipeline is not stalled
- The bool value indicates if the operation has been successful
- Non-Blocking should be privileged when it is not guaranteed that the operation is carried out
    - Ex: I/O channels

Kernel concurrency

- To take advantage of kernel concurrency and execute kernels in parallel
- You need to create a separate command queue for each kernel

```
#define NUM_KERNELS
...
// Ex with C++ and std::vector
std::vector<cl::Kernel> kernels;
std::vector<cl::CommandQueue> queues;
```

Buffered channels

- By default, channels are unbuffured, i.e., `__attribute__((depth(20)))`
- The depth attribute specify a **minimum** depth for the channel
- Buffered channels should be considered when the number of reads and writes are not symmetric
- Imbalance between reads/writes can be caused by conditional communication between kernels

I/O channels

- Channels providing data to the board or exporting data outside the board
  - E.g. network interfaces, PCIe interfaces, etc ...
- Board supplier has provided a BSP (board_spec.xml) which contains the I/O channels definitions

```
<channels>
  <interface name="udp_0" port="udp0_out"  type="streamsource" width="256"chan_id="eth0_in"/>
  <interface name="udp_0" port="udp0_in"  type="streamsink" width="256" chan_id="eth0_out"/>
</channels>
```

- I/O channel declaration using the io attribute. Reads and writes usage is the same as other channels

```
channel QUDPWord udp_in_IO __attribute__((io("eth0_in")));
channel QUDPWord udp_ou_IO __attribute__((io("eth0_out")));
```

Channels ordering

- No channel ordering by default
  - Channel calls can be executed out of order
  - Channel calls can be executed in parallel
- Risk of **DEADLOCKS** ...
- Use the mem_fence function to block
  - **CLK_CHANNEL_MEM_FENCE**
  - Enforce ordering

```
__kernel void producer(...){
      for(...){
        write_channel_intel(c0,...);
        mem_fence(CLK_CHANNEL_MEM_FENCE);
        write_channel_intel(c1,...);
}
__kernel void consumer(...){
      for(...){
        val = read_channel_intel(c0);
        mem_fence(CLK_CHANNEL_MEM_FENCE);
        val2 = read_channel_intel(c1);
      }
}
```

How to use Pipes ?

Pipes

- Channels are specific to Intel FPGA
    - Implemented before Khronos defined pipes for OpenCL
- Pipes should be preferred to conform with other SDK
- Intel implements pipes as a wrapper around channels
    - Channels are statically inferred, i.e., can be modified at runtime
    - All that applied to channels are also true for pipes
- AOC does not support the entire pipe specification
    - Not fully OpenCL conformant

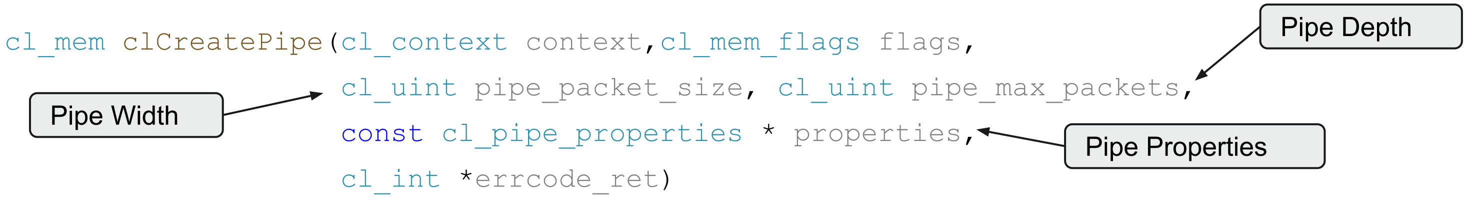
Pipes creation

- Pipes are specified as kernel arguments with the keyword pipe
  - `read_only` or `write_only` qualifiers should be specified
- Read and Write to the pipe with `read_pipe()` and `write_pipe()` calls

```
__kernel void producer(write_only pipe uint p0){
      for(...)
        error = write_pipe(p0, &data);
}
__kernel void consumer(read_only pipe uint p0){
      for(...)
        error = read_pipe(p0, &value);
}
```

Pipes creation -- host side

- Unlike channels, pipes requires host code to work
- Similarly to `clCreateBuffer`, the `clCreatePipe` function returns a cl_mem object representing the pipe object
- `clSetKernelArg` has to be used to map pipe to the correct read and write kernel args to be conform with OpenCL standard



```
cl_mem clCreatePipe(cl_context context,cl_mem_flags flags,
                    cl_uint pipe_packet_size, cl_uint pipe_max_packets,
                    const cl_pipe_properties * properties,
                    cl_int *errcode_ret)
```

Pipe Depth

Pipe Width

Pipe Properties

Pipes attributes

- Contrary to channels, pipe are non-blocking by default
- Add `__attribute__((blocking))` for blocking behavior

```
__kernel void producer(write_only pipe uint __attribute__((blocking)) p0);
__kernel void consumer(read_only pipe uint __attribute__((blocking)) p0);
```

- The depth attribute can be used to specify the minimum depth of the pipe

```
#define SIZE 100
__kernel void producer(write_only pipe uint __attribute__((depth(SIZE))) p0);
__kernel void consumer(read_only pipe uint __attribute__((depth(SIZE))) p0);
```

- I/O pipes used the io attributes, e.g., `__attribute__((io(“eth0_in”)))`

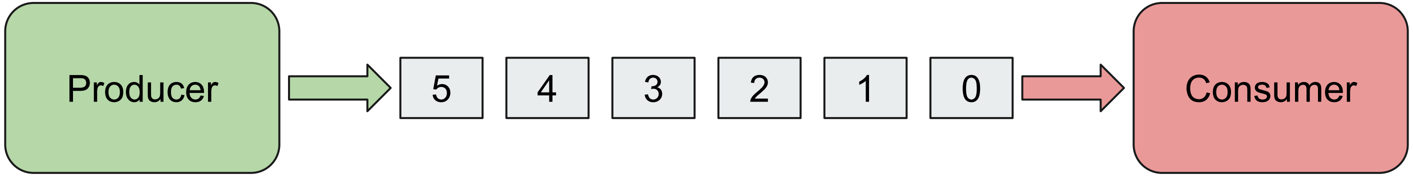
Difference with Standard OpenCL pipes

- Standard OpenCL pipes are created at execution when the host calls `clCreatePipe`
    - For Intel, the pipes are created before the host code is executed
    - Intel pipes implementation is static
    - The standard expect a dynamic implementation
- Standard pipes do not require the same pipe_id on the read and the write sides
    - The standard relies on the `clSetKernelArg` function call to know to match read side and write side
    - The Intel pipes need the same pipe_id to match read side and write side

Using channels and pipes

Channels/Pipes behavior

- Written data into a channel or a pipe remain valid during the kernel program lifetime on FPGA
- What is produced should be consumed or rejected by the consumer
- Ex: 6 elements produced and transmitted to consumer. The consumer read 3 elements at a time
    - There can be 2 read call for each write



Behavior with NDRange kernels

- Work-items have no specific ordering in NDRange kernels
- AOC enforces a work-item ordering with channels.
- These are the rules:
  - Pipelined architecture allows only reads and write once per clock cycle across different work-items
  - Smaller work-item IDs are executed first
  - “Threads” proceed in work-item and work-group order (dimX -> dimY -> dimZ)
  - Thread dependent workflow should be avoided
- For single-work item, consistency and in-order execution are ensured

Global restrictions

- Channels can have multiple read call sites but only a single call site
- Pipes are more restricted. Only a single read call and single write call site
- No loop unrolling when loops contain channels or pipes
- Kernels with channels cannot be vectorized → no num_simd_work_items
- Kernels with channels cannot be replicated → no num_compute_unit
- Static indexing mandatory for arrays of channels because AOC needs the information at compile time

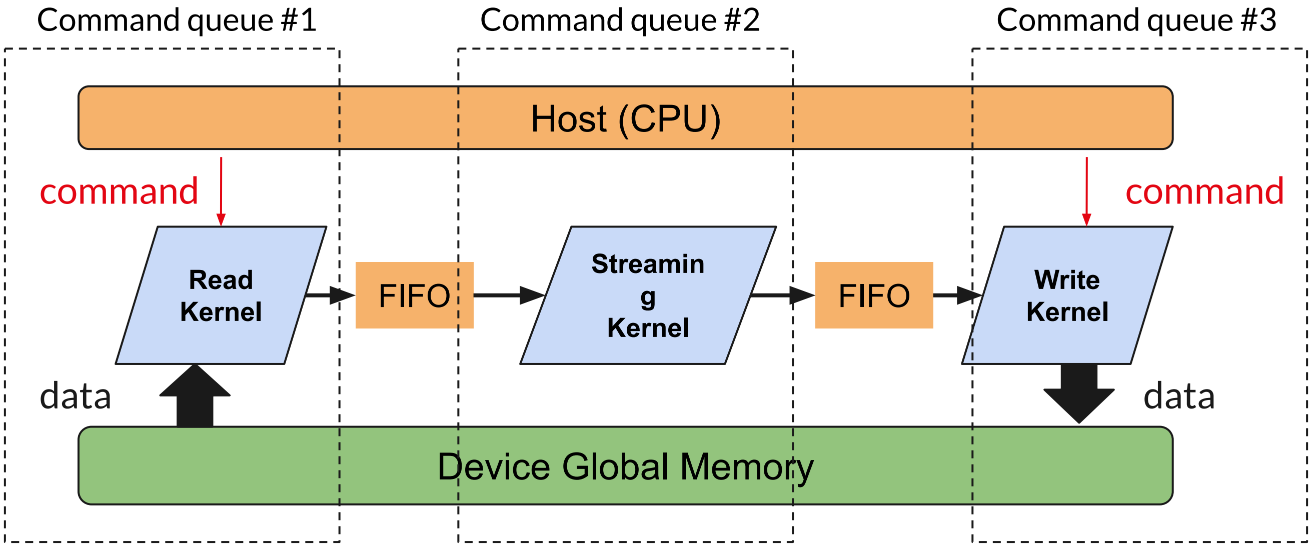
Best practices

- Prefer single work-item kernels over NDRange kernels
- Plan channels/pipes connections by decomposing a single large kernels into multiple small kernels
- Keep the number of channels reasonable → aggregate data when possible else use different channels
- Don’t use non-blocking versions and loop to wait data
  - Replace it with blocking channels/pipes to avoid significant waste
- Channels/Pipes can be emulated
  - Compilation for emulation, default channel depth is different from the default channel depth generated when your kernel is compiled for hardware. See Emulating Channel Depth to change this behavior
- I/O channels emulation more problematic. Refer to Emulating Applications with a Channel That Reads or Writes to an I/O Channel for more details

Channels vs Pipes

- Pipes are partially conformant with OpenCL standard
  - They can be used for host pipes, i.e. bypassing the FPGA global memory
- Channels can be used for autorun kernels (no input arguments needed)
  - No host code, less verbose
  - Should be preferred for FPGA implementation
- Both have same usage and performance

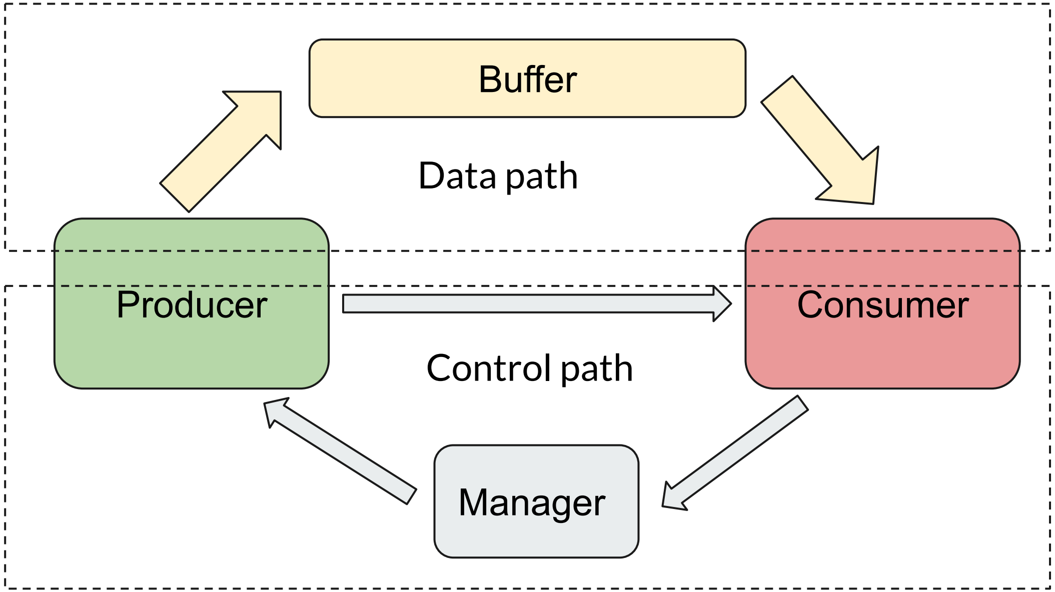
Channels/Pipes use case



- Read kernel transfers data from DDR to channels
- Streaming kernels read input channels, process data and write data to output channel
- Write kernel transfers data to DDR
- **3 different command queues**

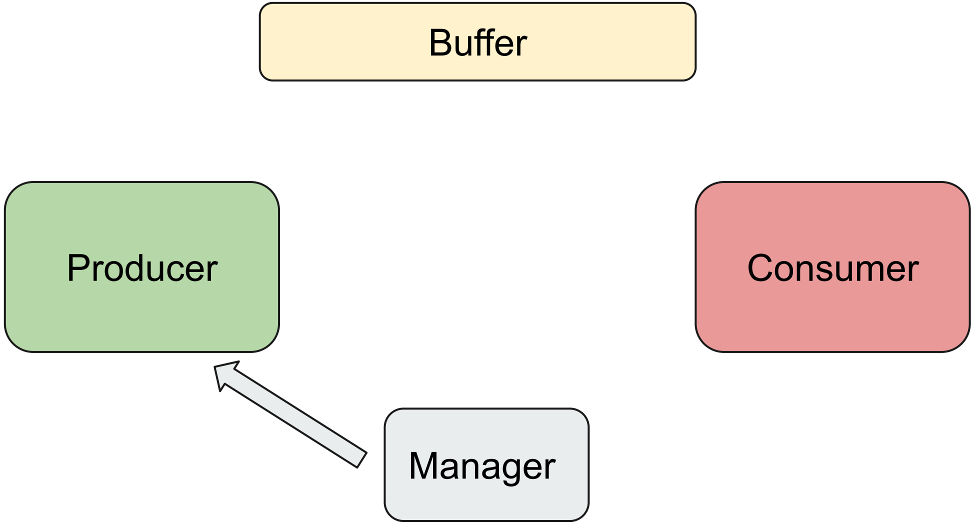
Advised Design Model

- When dealing with a large amount of data -- generalize previous use case



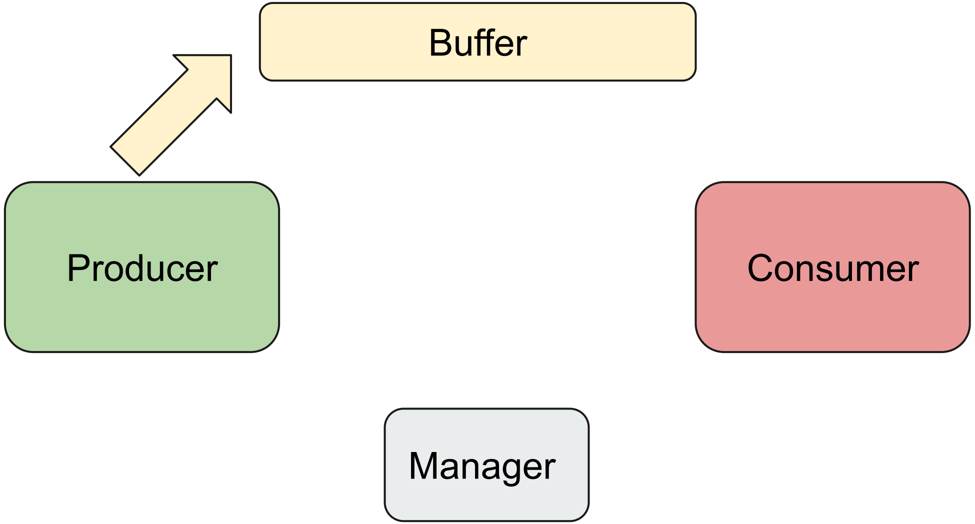
Workflow

- Manager sends a token to the producer



Workflow

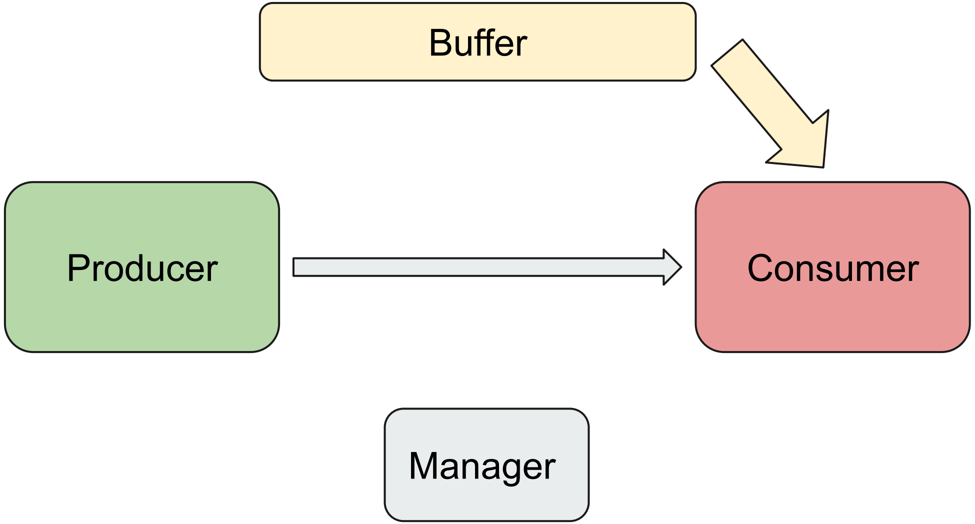
- The producer write data the specific buffer region based on information inside the token



- When dealing with a large amount of data

Workflow

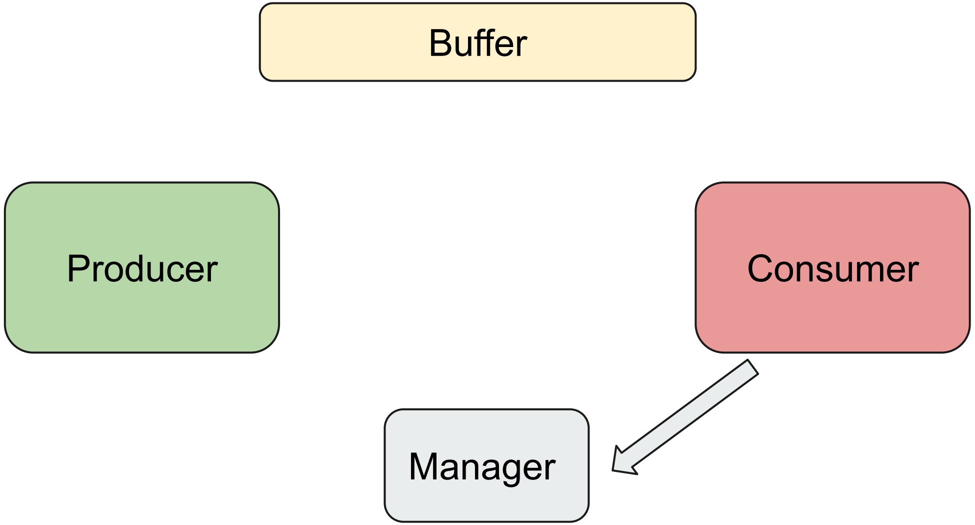
- The producer transmits the token to the consumer which takes over



- When dealing with a large amount of data

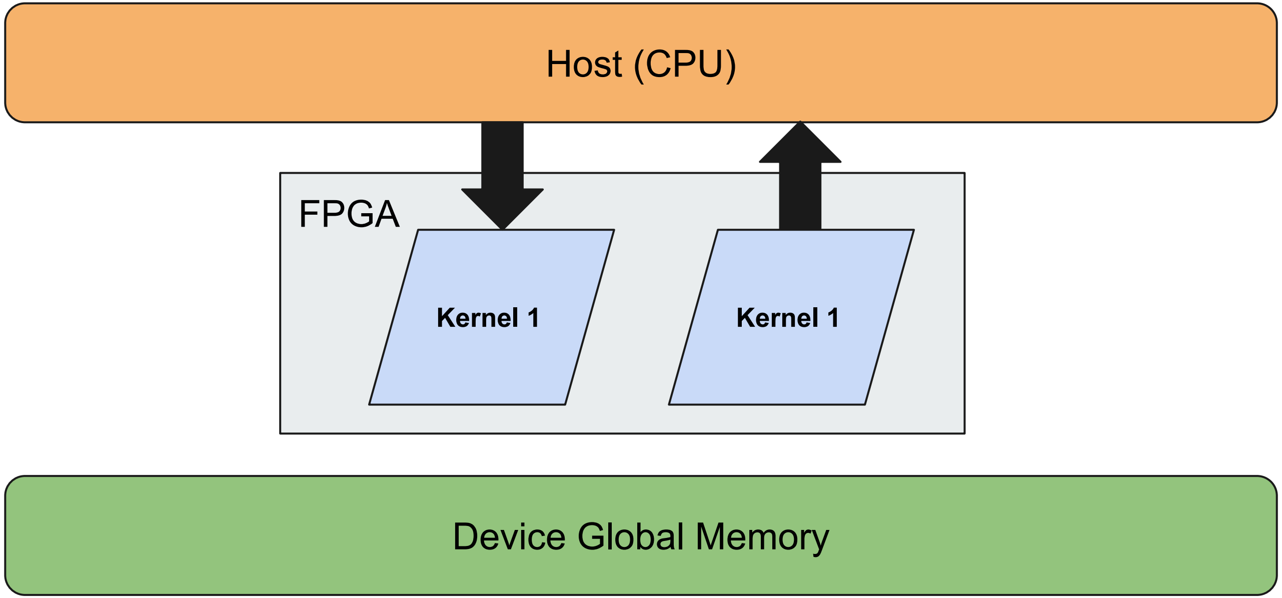
Workflow

- The consumer releases the buffer and send back the token to the manager



Using host pipes

Host pipes



- Host to kernel communication avoiding global memory
- Transfer data to/from the kernel without using global memory
- Only with pipes

Channels creation

- Enable Intel FPGA extension : `#pragma OPENCL EXTENSION cl_intel_fpga_host_pipe : enable`

- Use `__attribute__((intel_host_accessible))` when declaring the host pipe argument in the cl file

- Same usage than classical pipes

```
__kernel void reader(__attribute__((intel_host_accessible)) read_only pipe uint4 host_in){...}
__kernel void writer(__attribute__((intel_host_accessible)) write_only pipe uint4 host_out){...}
```

Host pipe -- host side

- Unlike channels, pipes requires host code to work
- Create the pipe using `clCreatePipe` function
  - Use the flags `CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_WRITE_ONLY`
- `clSetKernelArg` has to be used to map pipe to the correct read and write kernel args

```
cl_mem read_pipe = clCreatePipe(context,CL_MEM_HOST_READ_ONLY, sizeof(uint4),64, NULL, &error);
cl_mem write_pipe = clCreatePipe(context,CL_MEM_HOST_WRITE_ONLY, sizeof(uint4),64, NULL, &error);

clSetKernelArg(kernel,0,sizeof(cl_mem), (void*)&read_pipe);
clSetKernelArg(kernel,1,sizeof(cl_mem), (void*)&write_pipe);
```

Reading from and writing to host pipes

Function	Use case
`cl_int clReadPipeIntelFPGA(cl_mem pipe, gentype *ptr);`	- Read one data from pipe - Non-Blocking. Return 0 if successful.
`cl_int clWritePipeIntelFPGA(cl_mem pipe, gentype *ptr);`	- Read one data from pipe - Non-Blocking. Return 0 if successful.
`cl_int clMapHostPipeIntelFPGA(cl_mem pipe, ...);`	Creates buffer to provide data for reads and writes made up of multiple words
`cl_int clUnmapHostPipeIntelFPGA(cl_mem pipe, ...);`	Frees buffer to provide data for reads and writes made up of multiple words