FPGA

High Performance Computing, Summer 2021

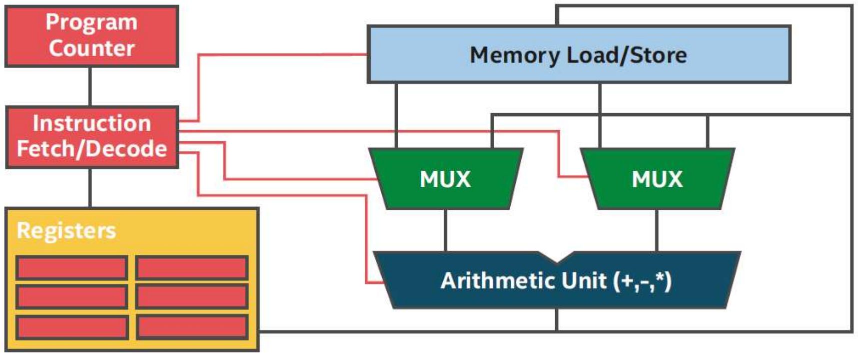
Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- FPGA
  - spatial architecture
  - pipeline
  - data flow
  - pipelining, queues, pipes

Von Neumann Bottleneck

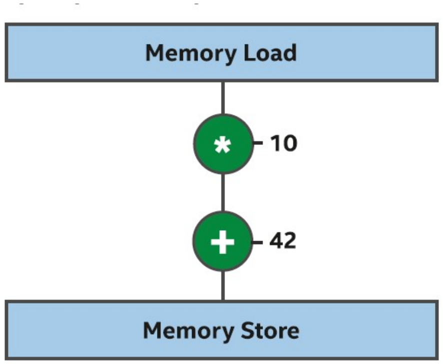
- Load-store “Von Neumann” architecture
  - based on Instruction Set Architecture (ISA)
    - e.g., CPUs and GPUs
  - executes a different instruction from the program in each clock cycle



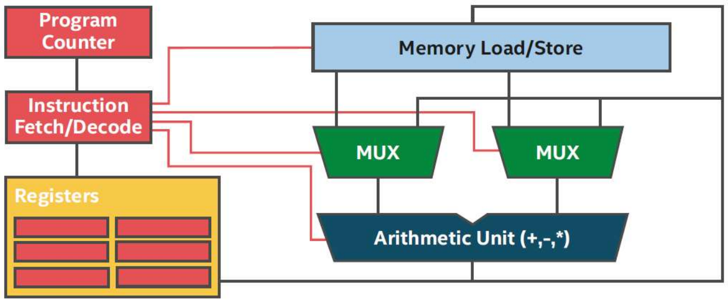
- Hardware reuse
  - hardware architecture that can run different instructions at different times
    - e.g., the memory load unit feeding an addition is probably the same memory load unit used to feed a subtraction
    - e.g., the same arithmetic unit is probably used to execute both the addition and subtraction instructions

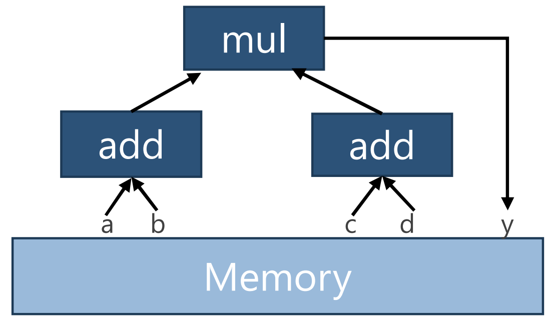
Spatial Architecture

- Spatial implementations of a program conceptually take the entire program as a whole and lay it down at once on the device
  - different regions of the device implement different instructions in the program
  - each instruction receives its own dedicated hardware that can execute simultaneously (same clock cycle) as the hardware implementing the other instructions
  - opposite perspective from sharing hardware between instructions over time (e.g., ISA)
- Static dataflow architecture
  - No longer need: instruction fetch, decode unit, program counter, register file
  - Dataflow: connect the output of one instruction to the input of the next



Von Neumann vs Spatial Architectures





- Von Neumann arch.
- Energy break down for a single add
  - I-Cache access: 25pJ
  - register file access: 6pJ
  - control: 29 pJ
  - total energy per instruction 70pJ

```
x = a + b
```

- Static Dataflow
- Energy per operation: 1-3pJ

```
x = (a + b) * (c + d)
```

Overcoming the Von Neumann Bottleneck

- Control is the bottleneck of Von Neumann architectures
- SIMD vector on CPUs and SIMT on GPUs
  - reduce the control overhead
  - but still need extra energy for registers
- FPGA reduce overhead for registers as well as control

FPGA Fitting

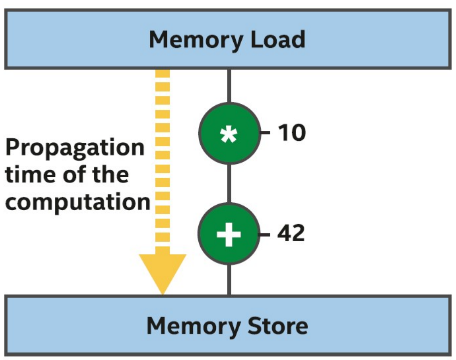
- Each instruction occupies some percentage of the spatial area of the device
  - What happens if the program requires more than 100% of the area?
- Small programs
  - if a program uses most of the area on the FPGA and there is sufficient work to keep all of the hardware busy every clock cycle
  - then executing a program can be incredibly efficient because of the extreme parallelism
- Large programs
  - to be tuned and restructured to fit on a device
  - resource sharing features of compilers can help to address this
    - but usually with some degradation in performance that reduces the benefit of using an FPGA
  - ISA-based architecture are very efficient resource sharing

FPGA Resource Sharing

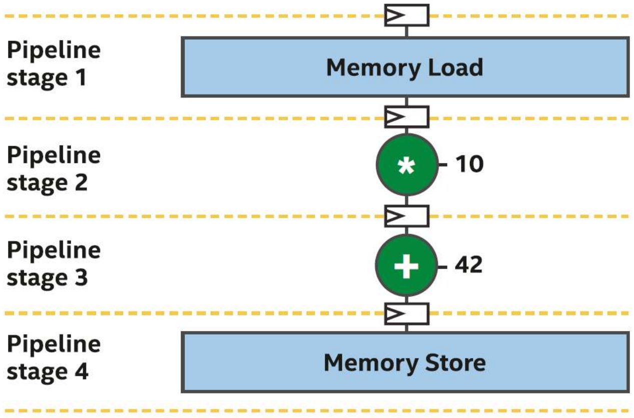
- Taken to the extreme, resource sharing solutions on an FPGA lead to an architecture that looks like an ISA-based accelerator
  - but that is built in reconfigurable logic instead being optimized in fixed silicon
- The reconfigurable logic leads to overhead relative to a fixed silicon design
  - therefore, FPGAs are not typically chosen as ways to implement ISAs
- FPGAs are of prime benefit
  - when an application is able to utilize the resources to implement efficient data flow algorithms

Pipeline Parallelism

- No pipeline
  - quite inefficient
  - most of the time
    - the hw is only doing useful work a small percentage of the time
    - operation such as the multiply is either waiting for new data from the load or holding its output so that operations later in the chain can use its result

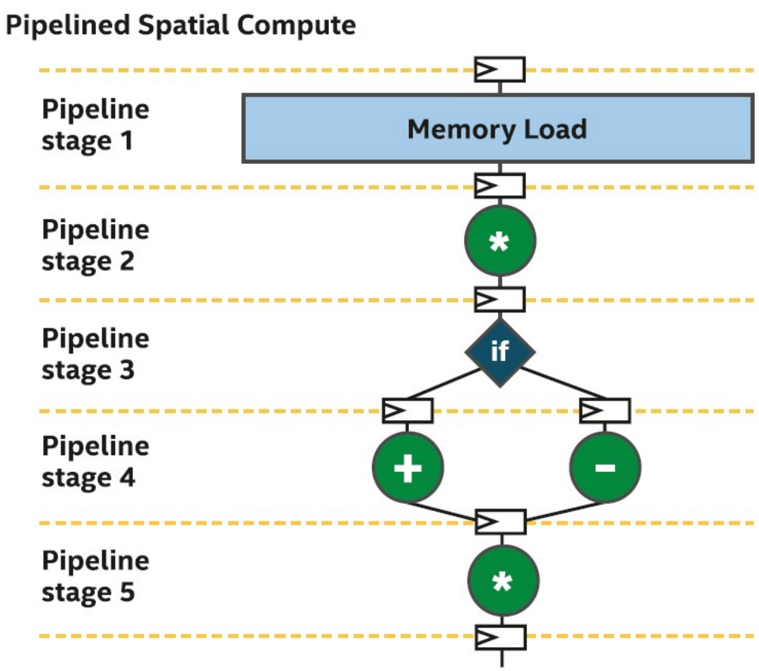


- With pipeline
  - execution of a single program is spread across many cc
  - achieved by inserting registers between some operations
    - each register holds a binary value for the duration of a cc
    - holds the result of an operation's output so that the next operation in the chain can operate on that held value
    - the previous operation is free to work on a different computation without impacting the input to following operations



Pipeline with Control Flow

- On ISA-based architectures
  - control flow (e.g., if/else structures) leaves some parts of the program inactive a certain percentage of the clock cycles
- FPGA compilers typically combine hardware from both sides of a branch, where possible
  - to minimize wasted spatial area and to maximize compute efficiency during control flow divergence
  - flow divergence much less expensive

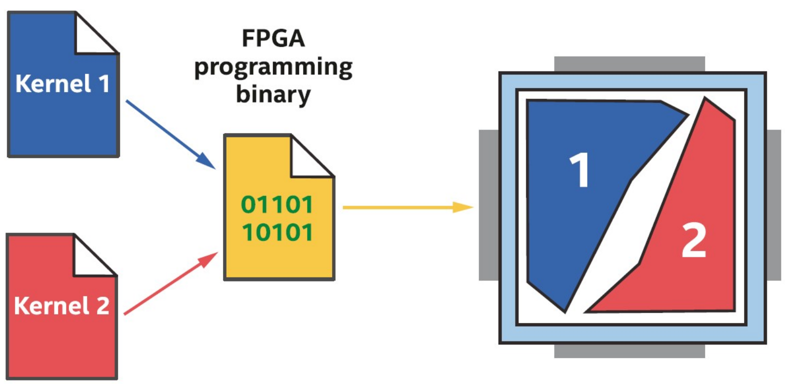


Scalar Data Flow

- Intermediate data between operations not only stays on-chip
  - is not stored to external memory
- Intermediate data between each pipeline stage has dedicated storage registers
  - different from vector architectures where multiple computations are executed as lanes of a shared vector instruction
- FPGAs are good for algorithms where
  - data dependences across units of work (such as work-items) can't be broken
  - fine-grained communication must occur

FPGA Chip Space

- Each kernel consumes chip area or space on the device
- Multiple kernels in the same FPGA binary
  - kernels can run concurrently
  - independent forward progress between kernels



FPGA Occupancy

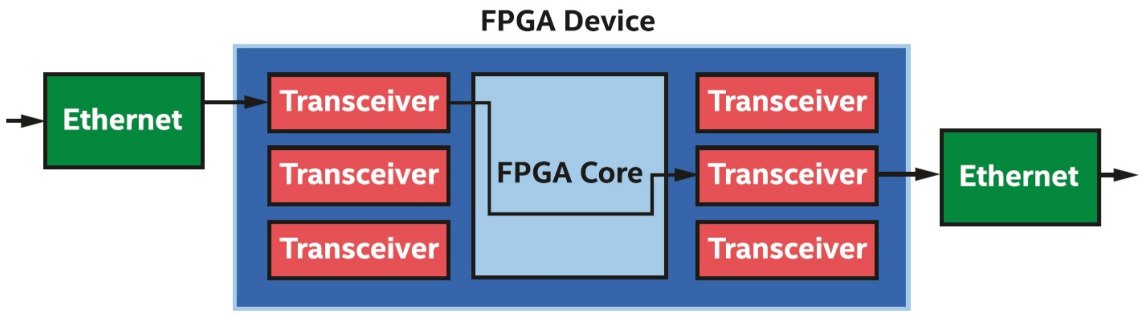
- If there isn't enough work to occupy most of the pipeline stages most of the time, then efficiency will be low
- We'll call the average utilization of pipeline stages over time occupancy of the pipeline
  - different from the definition of occupancy used when optimizing other architectures such as GPUs

FPGA Flexibility with Custom Operations

- FPGAs are very efficient at
  - bitwise operations, integer math operations on custom data widths or types
  - e.g., a 33-bit integer multiplier or a 129-bit adder
    - common in image processing and machine learning

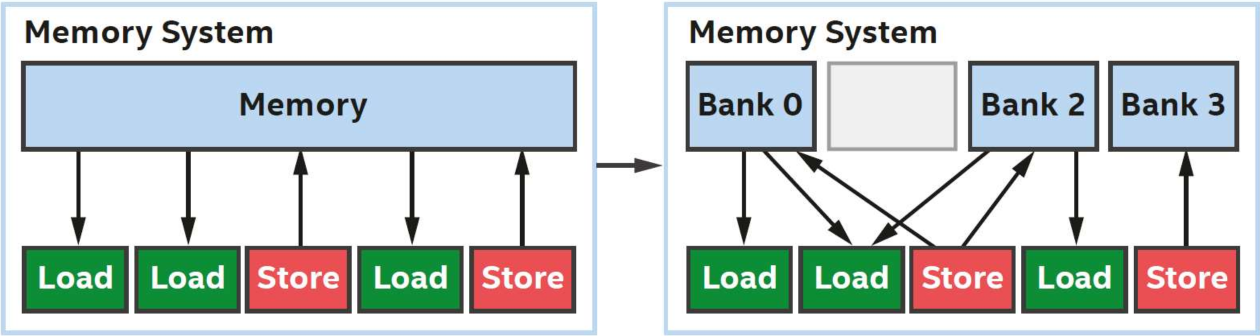
FPGA Low Latency I/O Streaming

- Some FPGA accelerator cards have network interfaces
  - make it possible to stream data directly into the device, process it, and then stream the result directly back to the network
  - good when
    - latency needs to be minimized
    - where processing through operating system network stacks is too slow or needs to be offloaded



FPGA Customized Memory System

- Memory systems built out of small blocks of on-chip memory
  - custom built for the specific portion of an algorithm or kernel using it
  - good for: applications that have atypical memory access patterns and structures
- FPGA memory systems are customized by the compiler for our specific code
  - GPU: avoid bank conflicts
  - FPGA: can change num of access ports per bank or have an unusual number of banks



FPGA: Performance Caveat

- FPGA devices differ from vendor to vendor or even from product generation to product generation
  - best practices for one device may not be best practices for a different device
  - to achieve optimal performance for a particular FPGA, always consult the vendor’s documentation
- Most popular FPGA
  - Intel Stratix 10
    - 14 nm, 1 GhZ, 5.5M logic elements, TDP 125 W
  - Xilinx Virtex Ultrascale+
    - 14 nm, 600 MHz, 2.5 M logic elements, TDP 225W

FPGA Programming Models:

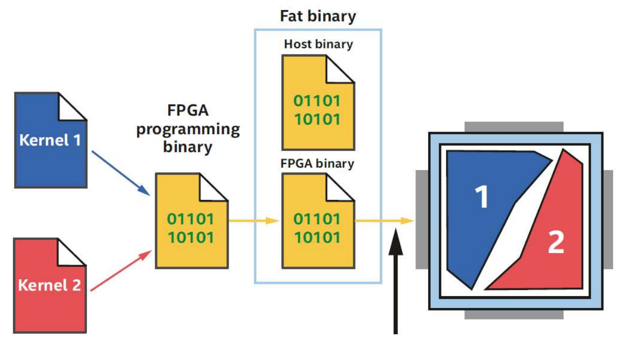
- Hardware Description Languages
  - Example: a counter in Verilog HDL
- High-level programming approaches
  - OpenCL / SYCL

```
module COUNTER #(
pWIDTH=8
) (
input                iCLK,
input                iRESET,
output [pWIDTH-1:0] oCOUNTER
);
reg [pWIDTH-1:0] rCOUNTER;
wire [pWIDTH-1:0] wNEXT_COUNTER;
assign wNEXT_COUNTER = rCOUNTER+1;
assign oCOUNTER = rCOUNTER;
always @(posedge iCLK)
begin
if (iRESET) begin
rCOUNTER<=0;
end else begin
rCOUNTER<= wNEXT_COUNTER;
end
end
endmodule
```

FPGA Programming Models: SYCL/DPC++

- Intel Toolchain
- FPGA binaries are embedded within the compiled DPC++ executable that run on the host
  - FPGA is automatically configured
- When we run a host program and submit the first kernel for execution on an FPGA, there might be a delay before the kernel begins executing, while the FPGA is programmed
  - resubmitting kernels for additional executions won't see the same delay because the kernel is already programmed to the device and ready to run

```
#include <CL/sycl.hpp>
#include <CL/sycl/intel/fpga_extensions.hpp>
using namespace sycl;
int main() {
  queue Q{ INTEL::fpga_selector{} };
  Q.submit([&](handler &h){
    h.parallel_for(1024, [=](auto idx) {
      // ...
    });
  });
  return 0;
}
```

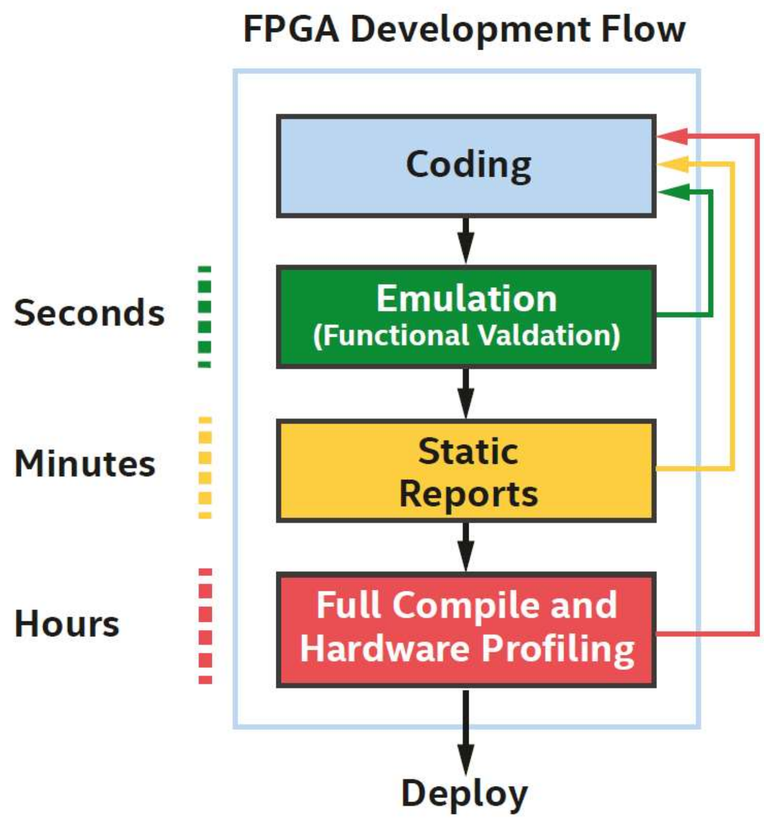


FPGA with SYCL/DPC++

- Emulator: acts as if it was an FPGA, including supporting relevant extensions and emulating the execution model
  - but runs on the host processor
  - not as fast as on actual FPGA

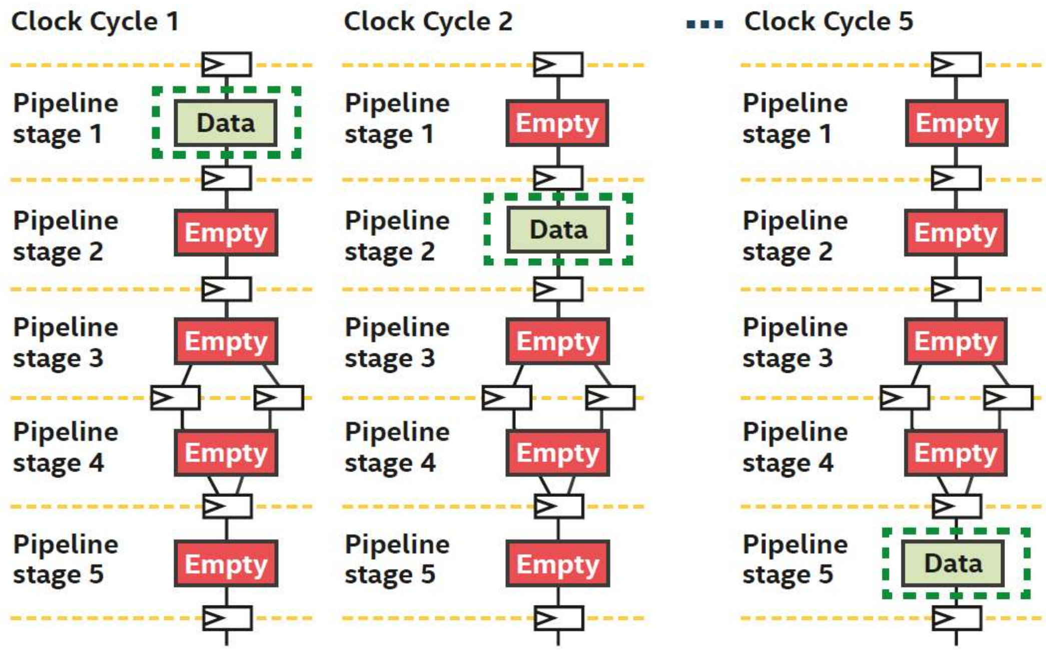
```
queue Q{ INTEL::fpga_emulator_selector{} };
```

- Static reports: report on the FPGA structures created by the compiler and on bottlenecks identified by the compiler



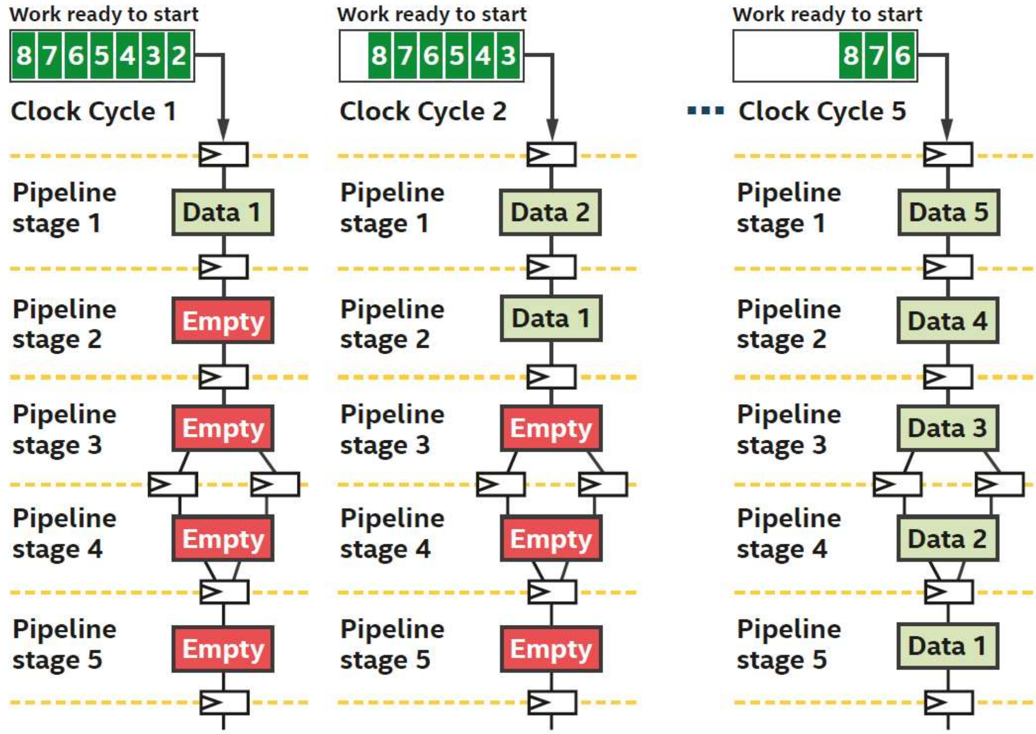
Pipelining and Queues

- Goal of pipelining: enable multiple elements of data to be processed at different stages of the pipeline, simultaneously
- Example of inefficient pipeline
  - pipeline stages are mostly unused if processing only a single element of work



Pipelining and Queues

- Keep the pipeline stages better occupied
  - queue of un-started work waiting before the first stage, which feeds the pipeline
  - each clock cycle, the pipeline can consume and start one more element of work from the queue
  - after some initial startup cycles, each stage of the pipeline is occupied

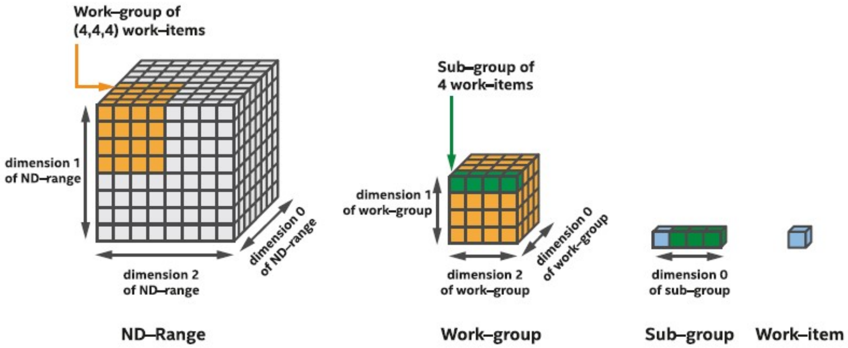


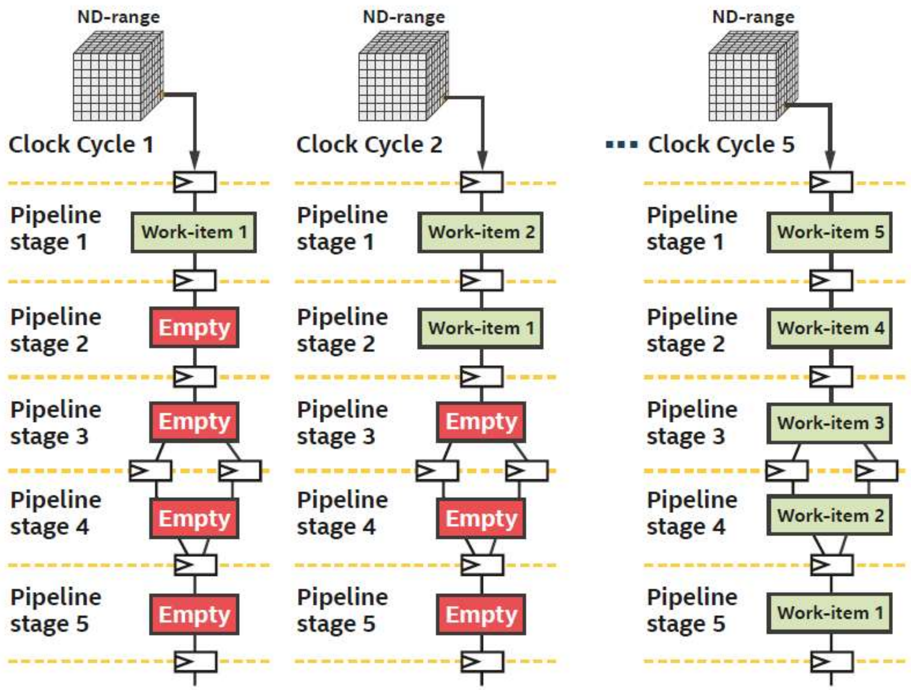
FPGA Queue Feeding in SYCL

- Two methods
  - ND-range kernels
  - Loops
- Optimization must carefully balance between them

FPGA Queue from ND Range

- ND-range execution model
  - work-items do not frequently communicate with each other in most applications



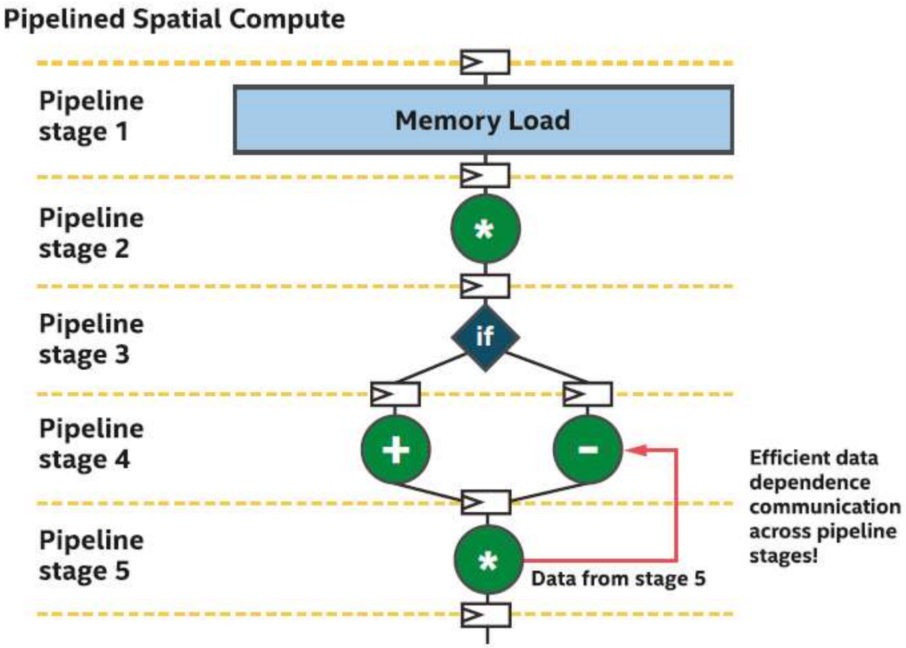


FPGA Backward Communication

- Loop-carried data dependence

```
int state = 0;
for (int i=0; i < size; i++) {
  state = generate_random_number(state);
  output[i] = state;
}
```

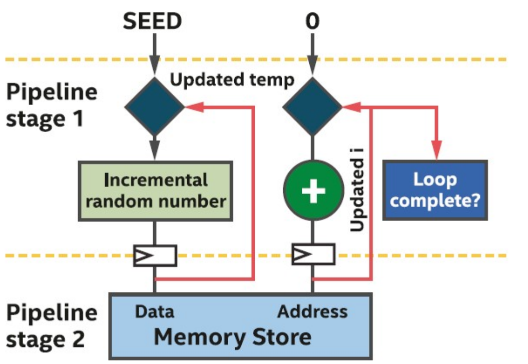
- FPGA can very efficiently communicate results backward in the pipeline
  - to work at an earlier stage in the pipeline
- Approaches
  1. loops
  2. intra-kernel pipes with ND-range kernels

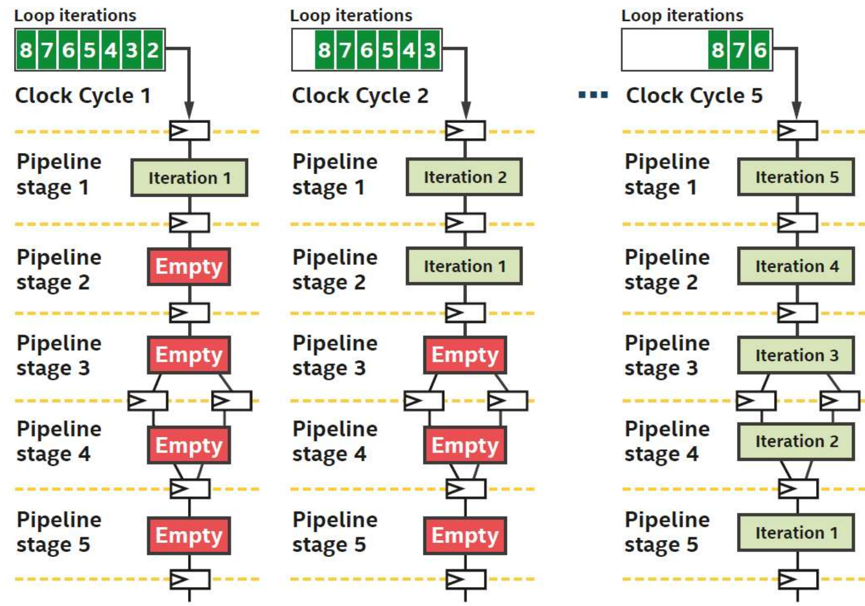


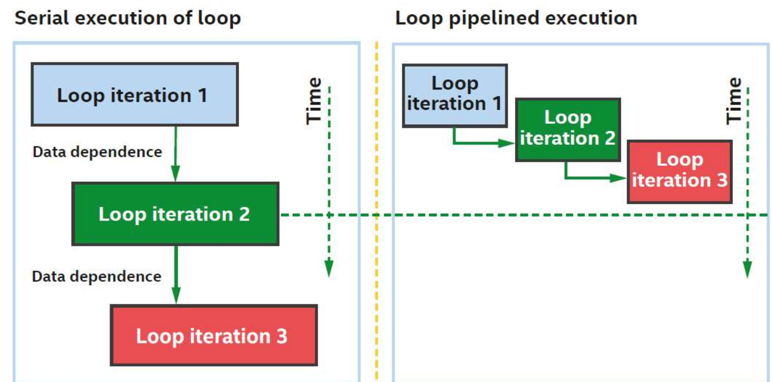
FPGA Queue fed by Loops

- Loop pipelining

```
h.single_task([=]() {
int state = seed;
for (int i=0; i < size; i++) {
  state = generate_incremental_random_number(state);
  output[i] = state;
}
});
```

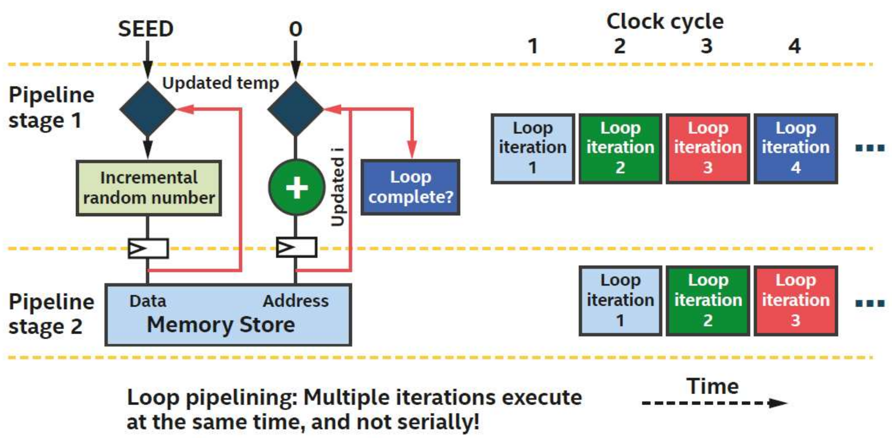


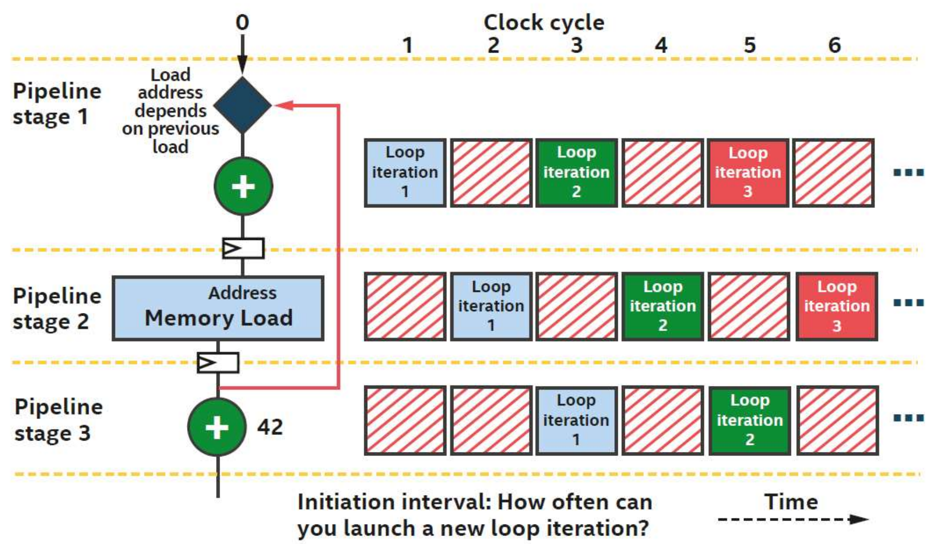




FPGA Loops and Occupancy

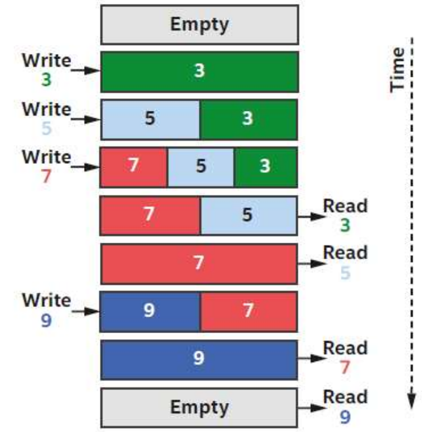
- Pipeline can initiate a new loop iteration over N clock cycles
  - called initiation interval



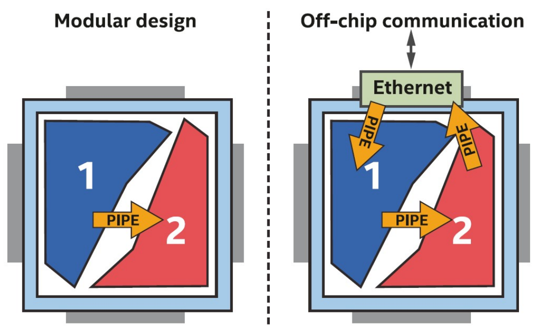


Pipes

- Spatial architectures relies on a first-in first-out (FIFO) buffer
    1. implicit control information carried alongside the data, signals tell us whether the FIFO is empty or full and can be useful when decomposing a problem into independent pieces
    2. FIFOs have storage capacity: easier to achieve performance in the presence of dynamic behaviors such as highly variable latencies when accessing memory

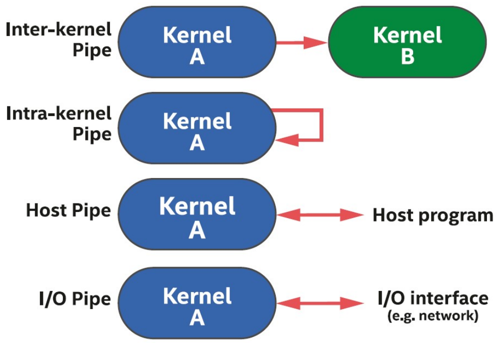


- Used for modular design



Pipes

- Four types of pipe connectivity in SYCL/DPC++



- Example
  1. ND range
  2. Single task with a loop

```
using my_pipe = pipe<class some_pipe, int>;
// ND-range kernel
Q.submit([&](handler& h) {
  auto A = accessor(B_in, h);
  h.parallel_for(count, [=](auto idx) {
  my_pipe::write( A[idx] );
  });
});
// Single_task kernel
Q.submit([&](handler& h) {
  auto A = accessor(B_out, h);
  h.single_task([=]() {
  for (int i=0; i < count; i++) {
    A[i] = my_pipe::read();
  }
  });
});
```

FPGA Custom Memory Systems

- Static coalescing
  - memory accesses into a smaller number of wider accesses, where it can
  - reduces the complexity of a memory system
    - in terms of numbers of load or store units in the pipeline, ports on the memory system, the size and complexity of arbitration networks, and other memory system details
- Memory access style
  - load or store units for memory accesses
  - tailored to the memory technology being accessed
    - e.g., on-chip vs. DDR vs. HBM
  - tailored to the access pattern inferred from the source code
    - e.g., streaming, dynamically coalesced/widened, or likely to benefit from a cache of a specific size
- Memory system structure
  - memory systems (both on- and off-chip) can have banked structures
  - many controls and mode modifications that can be used to control these structures and to tune specific aspects of the spatial implementation

FPGA Summary: Basic Elements

- Lookup table
- Math engine
  - for common math operations such as addition or multiplication of single-precision floating point numbers, FPGAs have specialized hardware to make those operations very efficient
- On-chip memory
  - registers that are used to pipeline between operations and some other purposes
  - block memories that provide small random-access memories spread across the device
- Interfaces to off-chip hardware
- Routing fabric between all of the other elements

Lab

- Project selection, to define
  1. Topic
  2. Target hardware
  3. Baseline
  4. Tasks
- Teaching assistant
  - Alessia Antelmi for projects on hypergraphs
  - Kaijie Fan for projects on energy
  - Majid Salimibeni for GPU
  - I will assist for other topics (vectorization, ect)