SYCL Academy

SYCL™

Can FPGAs accelerate your workload?

Mike Kinsner, Principal Engineer @ Intel

SYCL Academy

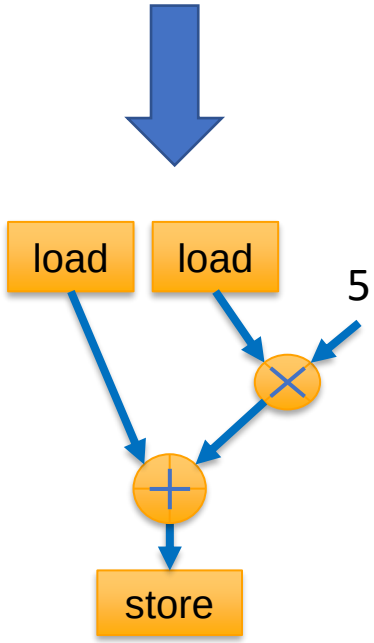
SYCL™

1) Mapping an Algorithm to Hardware

A framework for thinking about Field Programmable Gate Arrays (FPGAs)

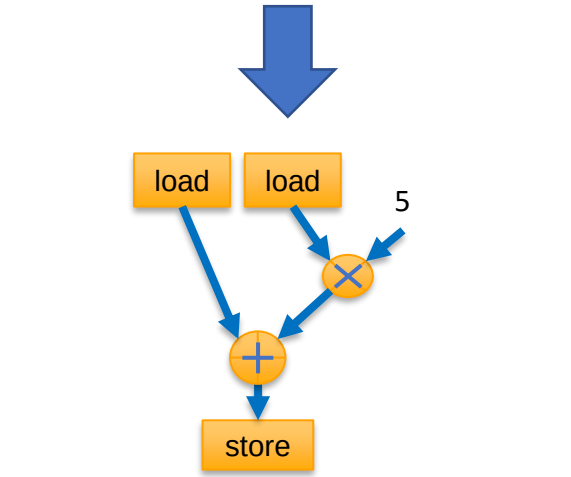
Data Flow Graph: The Core of Modern Compilers

```
for (int i=1; i  < 10; ++i) {
  A[i] += B[i-1] * 5;
}
```



Data Flow Graph: The Core of Modern Compilers

```
for (int i=1; i  < 10; ++i) {
  A[i] += B[i-1] * 5;
}
```

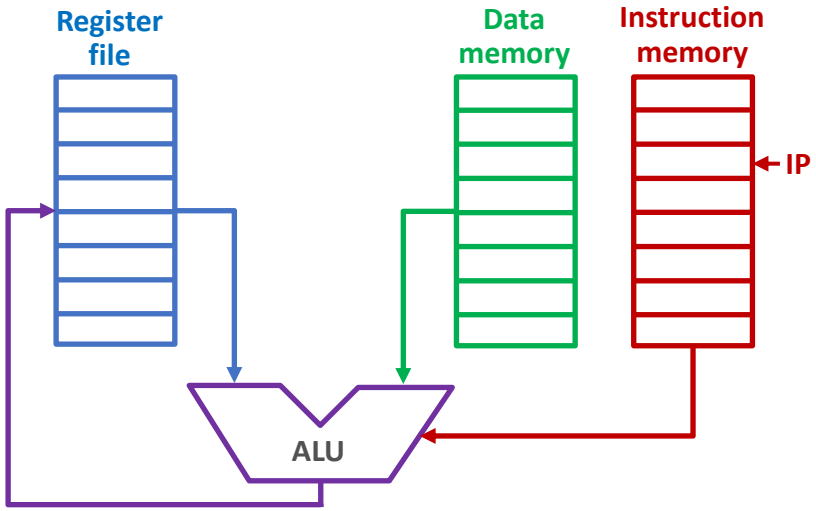


Hardware?

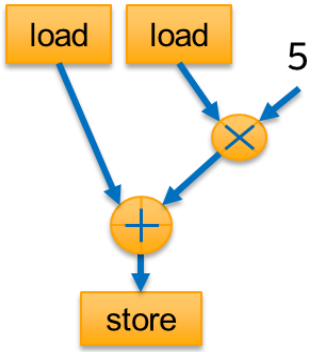
Executing in Silicon: Two key options

Option 1:

Instruction set architecture (ISA) machine



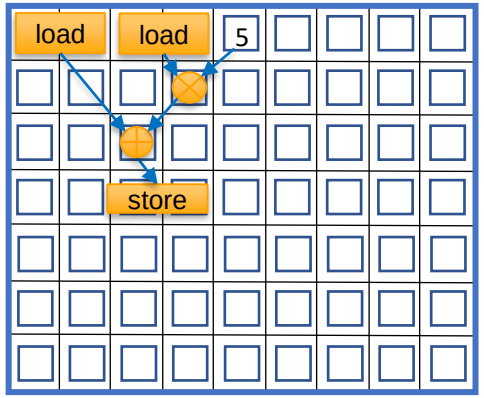

```
for (int i=1; i  < 10; ++i) {
  A[i] += B[i-1] * 5;
}
```



Option 2:

Materialize data flow graph directly on reconfigurable spatial hardware

Hardware with configurable spatial “regions”

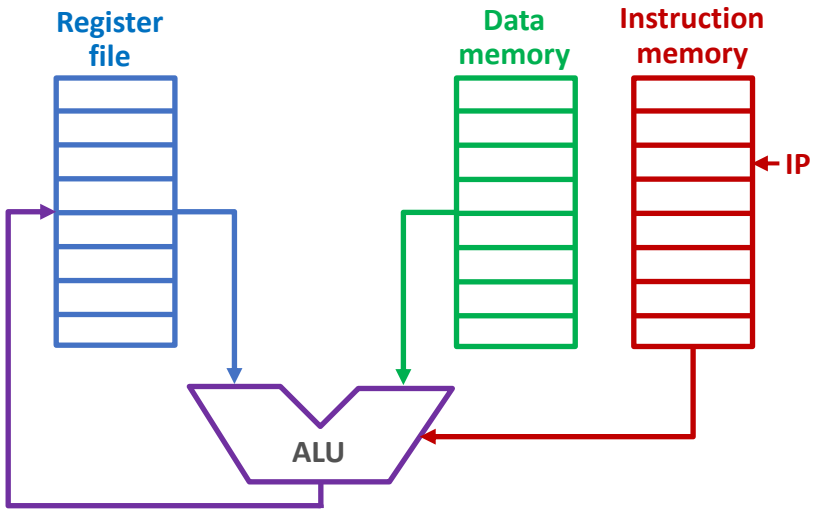


Execute consecutive instructions on **same** hardware via **time multiplexing**

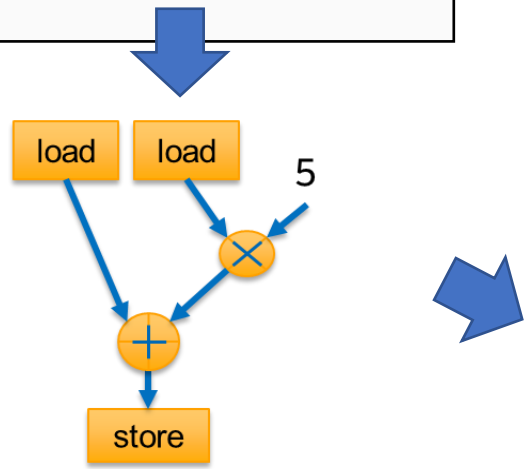
Execute instructions on **different** hardware by **specializing device regions in pipeline**

Many Architectural Tradeoffs

Instruction Set Architecture



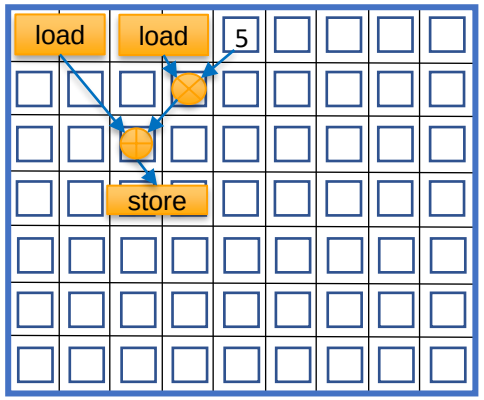

```
for (int i=1; i  < 10; ++i) {
  A[i] += B[i-1] * 5;
}
```



Spatial Architecture

Materialize data flow graph directly on reconfigurable spatial hardware

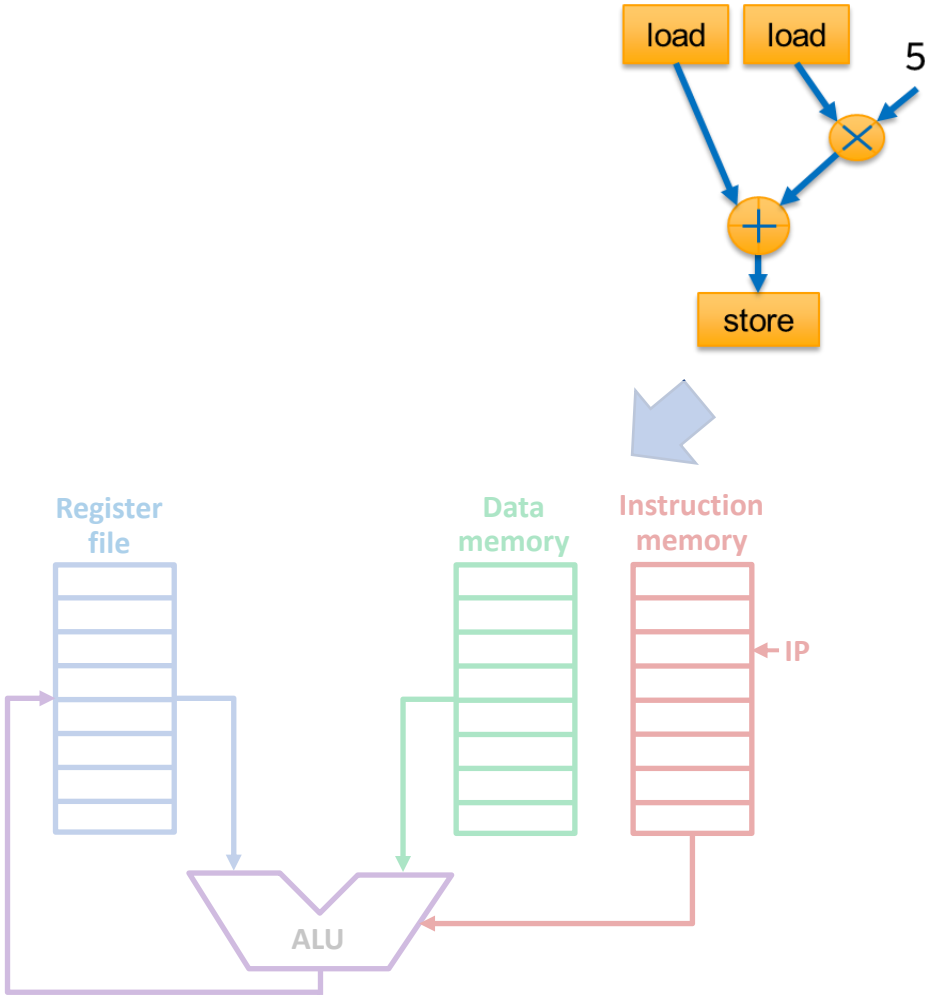
Hardware with configurable spatial "regions"



+ Easy to program
+ General purpose at runtime
- Not all hardware may be used every clock cycle

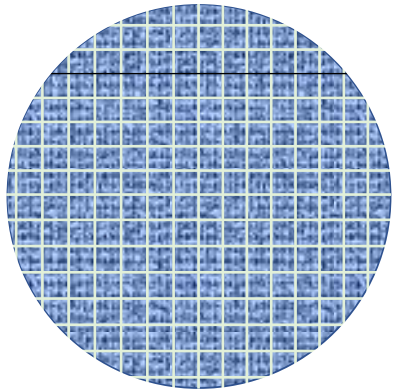
+ Excellent Perf, Perf/Watt from specialization
- Historically inaccessible by SW developers
- Data path not typically runtime context switched

Think of as a Reconfigurable Custom Chip

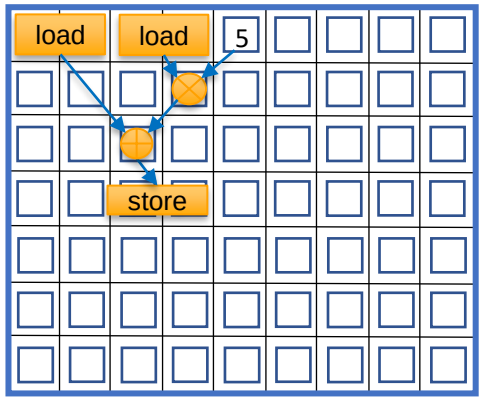


Spatial Architecture

Fixed Spatial (e.g. ASIC)



Reconfigurable Spatial (e.g. FPGA)

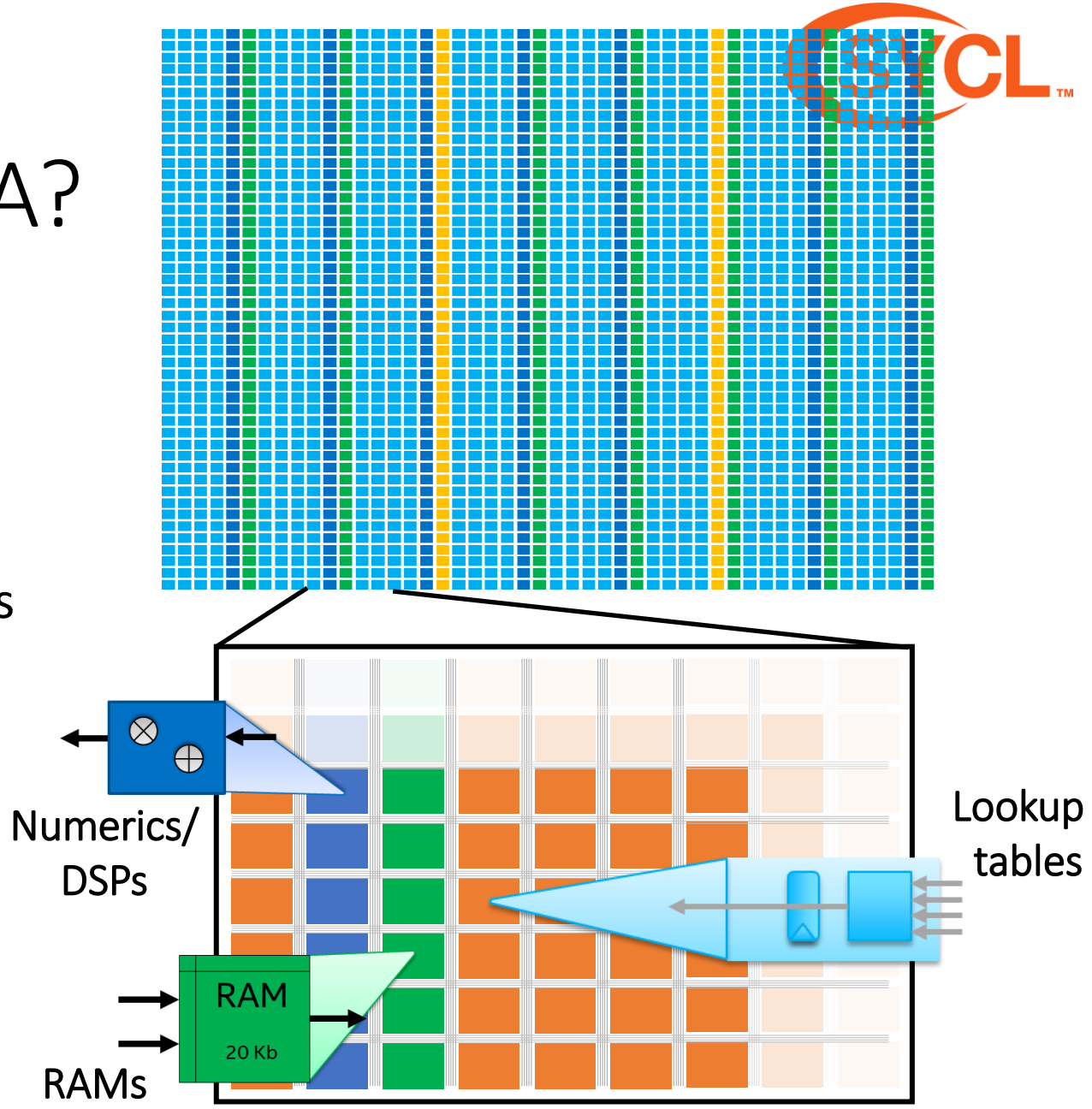


Think of FPGA as able to implement what you would build as a custom data path on a dedicated chip/ASIC!
(there is an overhead cost to FPGA reconfigurability vs ASIC, but much less expensive in low/medium volumes and reconfigurable)

2) Field Programmable Gate Arrays (FPGAs)

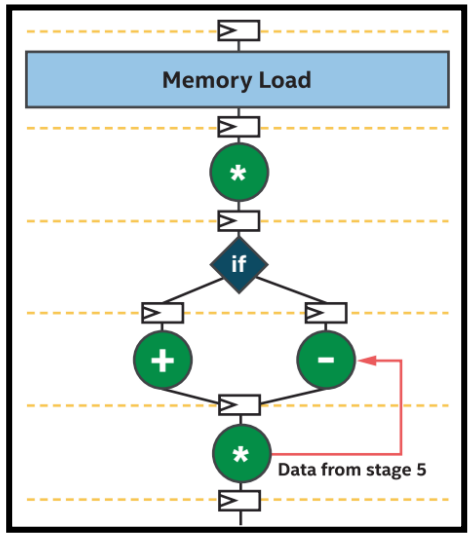
What is a Modern FPGA?

- Massive array of small processing units
  - 1-bit ALUs (~millions)
  - Dual-ported memories (~ten thousand)
  - Floating point MAC (~thousands)
- Connected by mesh of programmable wires
- Sitting inside a ring of high-speed I/O
- With an optional embedded processor
- Common within infrastructure hardware
- Can buy on PCIe cards ready to go today

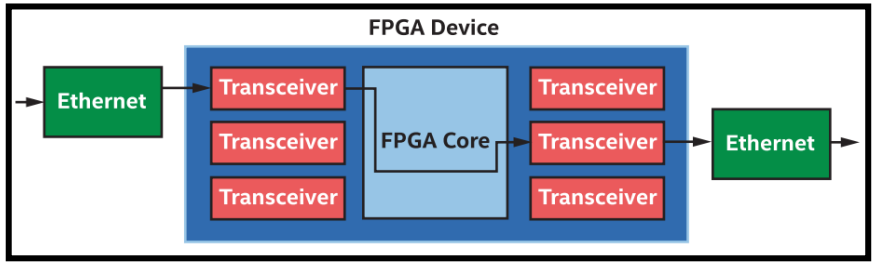


FPGA Value

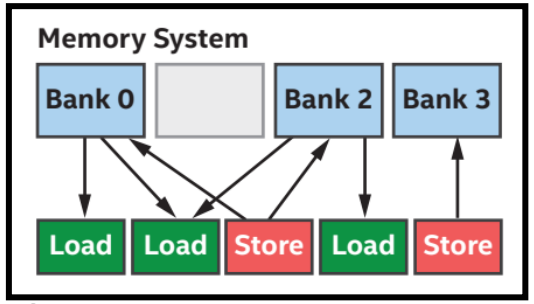
1. **Spatial** architecture – many data flow algorithms map well
   - Data dependences across parallel work
   - Streaming and graph processing patterns
   - Enables some algorithms, makes others tractable to express



2. Rich **I/O**
   - Network, memory, custom interfaces and protocols
   - Very low + deterministic latencies

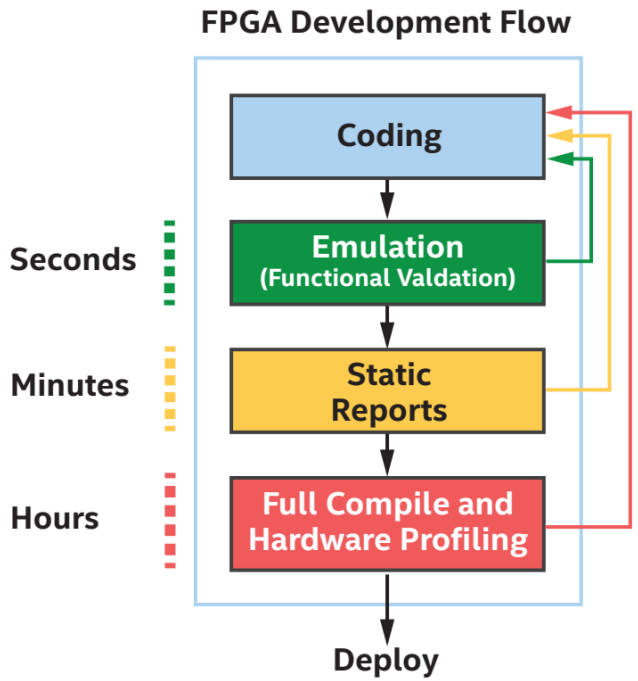


3. Large distributed on-chip **memory** BW
   - Custom topologies tuned to algorithm



Development Flows

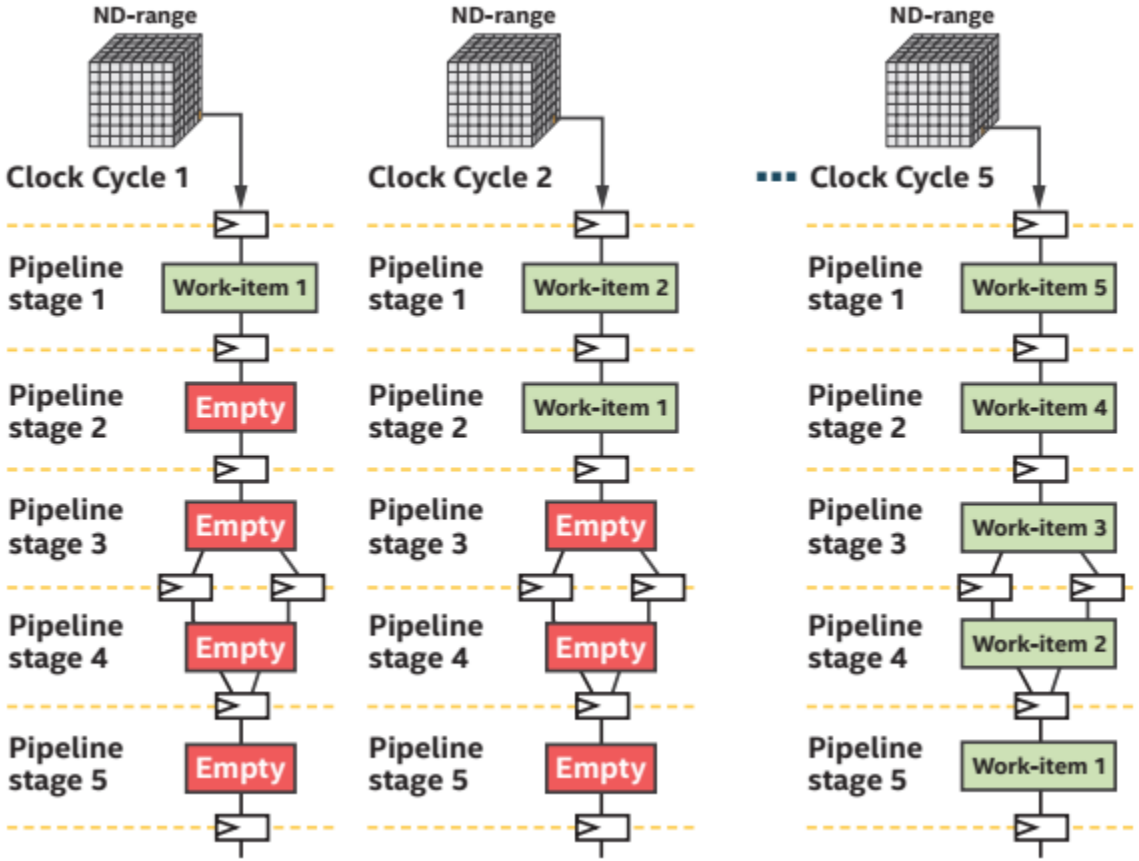
- Compile times to hardware longer than SW developers are used to
  - Tools + reporting allow most development to be done without HW compilation



Deep Spatial Pipelines: Creating Parallelism

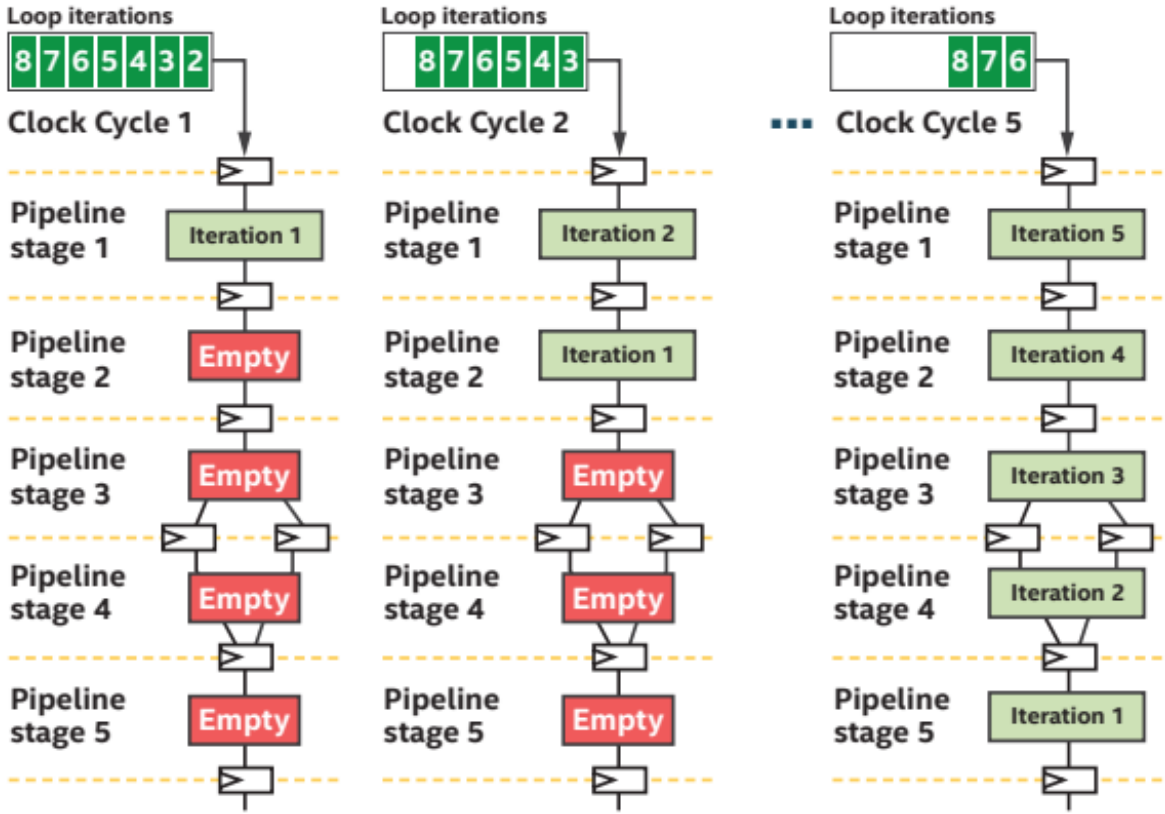
ND-range:

```
h.parallel_for({8,8,8}, [=](auto I) {
  output[I] = in[I] * 5;
});
```



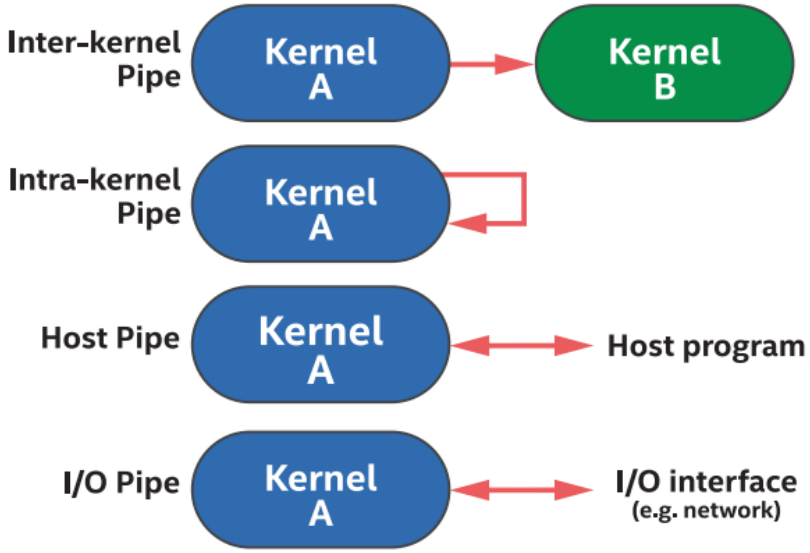
Loop:

```
h.single_task([=] {
  for (int i=2; i < size; i++) {
    A[i] = A[i-1] * foo(A[i-2]); …;
  } });
```



One FPGA-Specific Language Feature (of many)

Data flow pipes:



Combine task graph abstraction with data flow

```
constexpr int count = 1024;
using my_pipe = pipe<class some_pipe, int>;

std::array<int, count> in_array;

for (int i=0; i < count; i++) { in_array[i] = i;} // Init

buffer B_in{ in_array };  // Init from std::array
buffer<int> B_out{ range{count} };
queue Q{ INTEL::fpga_emulator_selector{} };

// 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();
      }
    });  });
```