DATA FLOW PIPES: A SYCL™ EXTENSION FOR SPATIAL ARCHITECTURES

Mike Kinsner and John Freeman, Intel FPGA High Level Design Engineering

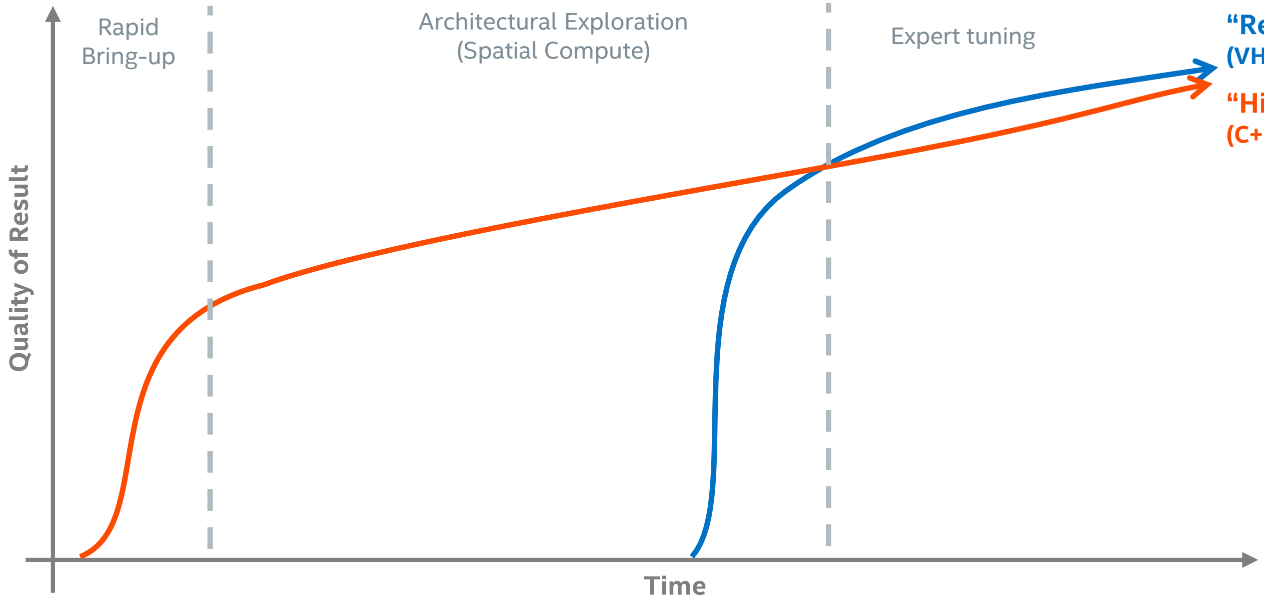
H²RC Workshop @ SC19. Nov 17, 2019

HIGHER LEVEL PROGRAMMING MODELS / TOOLING HAVE EVOLVED FOR FPGA

More Accessible

Easier to Meet Schedules

Easier to Achieve Performance



"Register Transfer Level" Design
(VHDL, Verilog)

"High Level" Design
(C++, OpenCL™, SYCL™)

Compilers

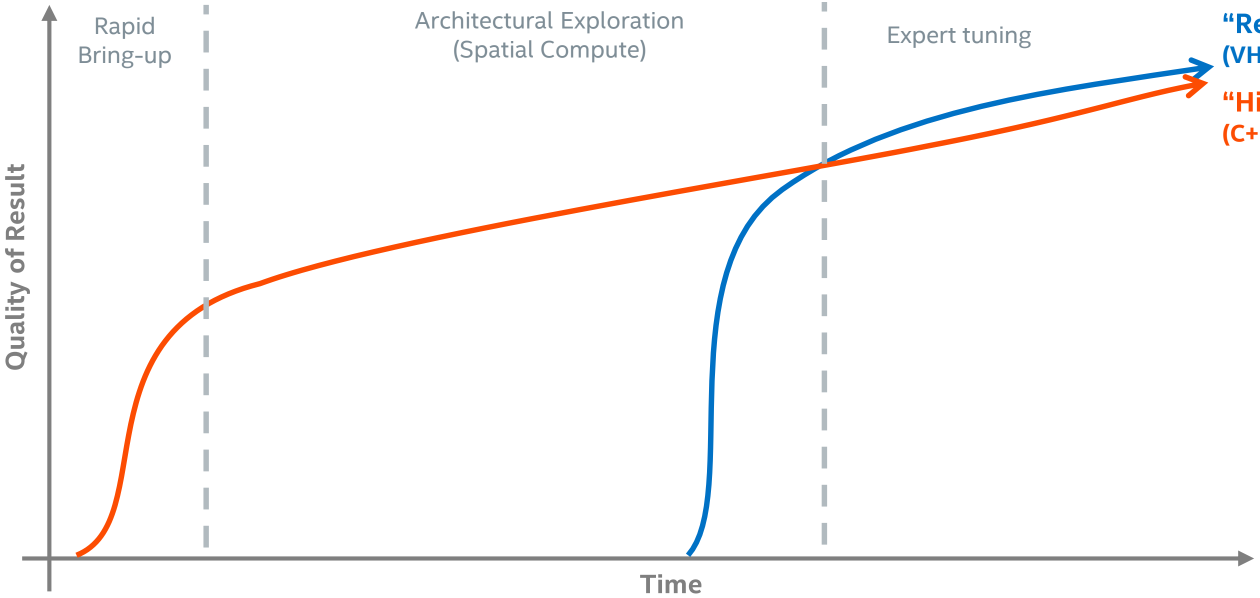
Platforms

Tooling + frameworks

Khronos® is a registered trademark and SYCL and SPIR are trademarks of the Khronos Group, Inc. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

LLVM logo: https://llvm.org/Logo.html

STILL NEED TO CODE TO AN ARCHITECTURE



"Register Transfer Level" Design
(VHDL, Verilog)

"High Level" Design
(C++, OpenCL™, SYCL™)

Compilers

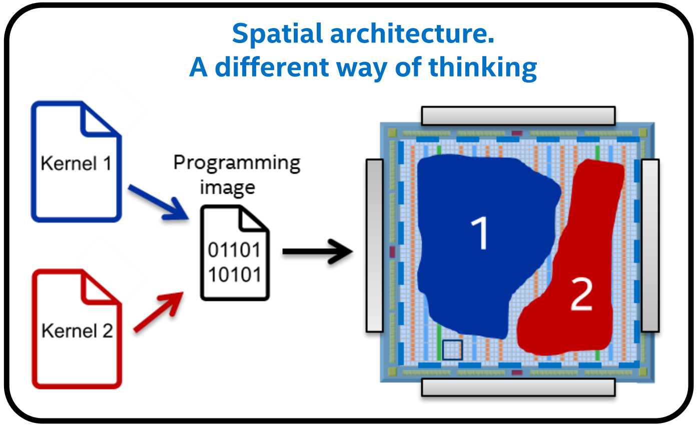
Platforms

Tooling + frameworks

Compilers still don't provide cross-architecture performance portability.

LLVM logo: https://llvm.org/Logo.html

SPATIAL COMPUTE



DIFFERENT PARADIGM

- Logically each operation of the compute is in a different **location** on the device
  - *Operations can execute simultaneously across space*
  - *Operations chained together into a data flow pipeline*

FIFO PRIMITIVE IN SPATIAL COMPUTE

First-In First-Out data storage construct. Control sideband

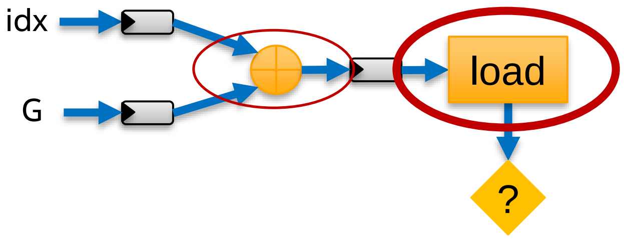
- Checking for data availability is cheap
- Implicit flow control signals (ready/full), low latency
- **Enables:** Producer / consumer can communicate at a very fine granularity

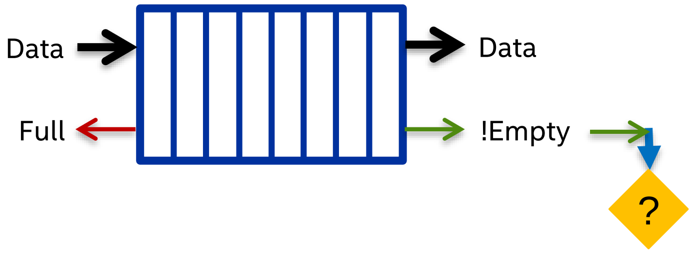
Memory based:

```
h.parallel_for(rng, [=](id<1> idx) {
  if (G[idx]) {...}
})
```

FIFO based (with on-chip implementation)**:**

```
h.parallel_for(rng, [=](id<1> idx) {
  int val; bool success;
  my_pipe::read(val, success)
  if (success) {...}
});
```





A COMPLETE SYCL PROGRAM

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
  auto rng = range<1>(num);
  buffer<int> A{ rng }, B{ rng };
  queue Q;

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(rng, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(rng, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::write>(h);
    h.parallel_for(rng, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  Q.submit([&](handler& h) {
    auto in = A.get_access<access::mode::read>(h);
    auto inout = B.get_access<access::mode::read_write>(h);
    h.parallel_for(rng, [=](id<1> idx) {
      inout[idx] *= in[idx]; }); });
```

Cont'd

```
  auto result = B.get_access<access::mode::read>();
    for (int i=0; i<num; ++i) std::cout << result[i] << "\n";

    return 0;
  }
```

Submit four kernels to a device (e.g. FPGA)!

Output:

```
0
1
4
9
16
25
36
49
64
81
100
121
```

- The SYCL standard is from Khronos
- Intel is building a SYCL implementation in open source, aiming for upstream LLVM
  - https://github.com/intel/llvm

SYCL RUNTIME KERNEL SCHEDULING

```
int main() {
 buffer<int> A{ rng }, B{ rng };
 queue Q;

 Q.submit([&](handler& h) {
  auto out = A.get_access<access::mode::write>(h);
  h.parallel_for(rng, [=](id<1> idx) {
   out[idx] = idx[0]; }); });
```

Kernel 1

```
 Q.submit([&](handler& h) {
  auto out = A.get_access<access::mode::write>(h);
  h.parallel_for(rng, [=](id<1> idx) {
   out[idx] = idx[0]; }); });
```

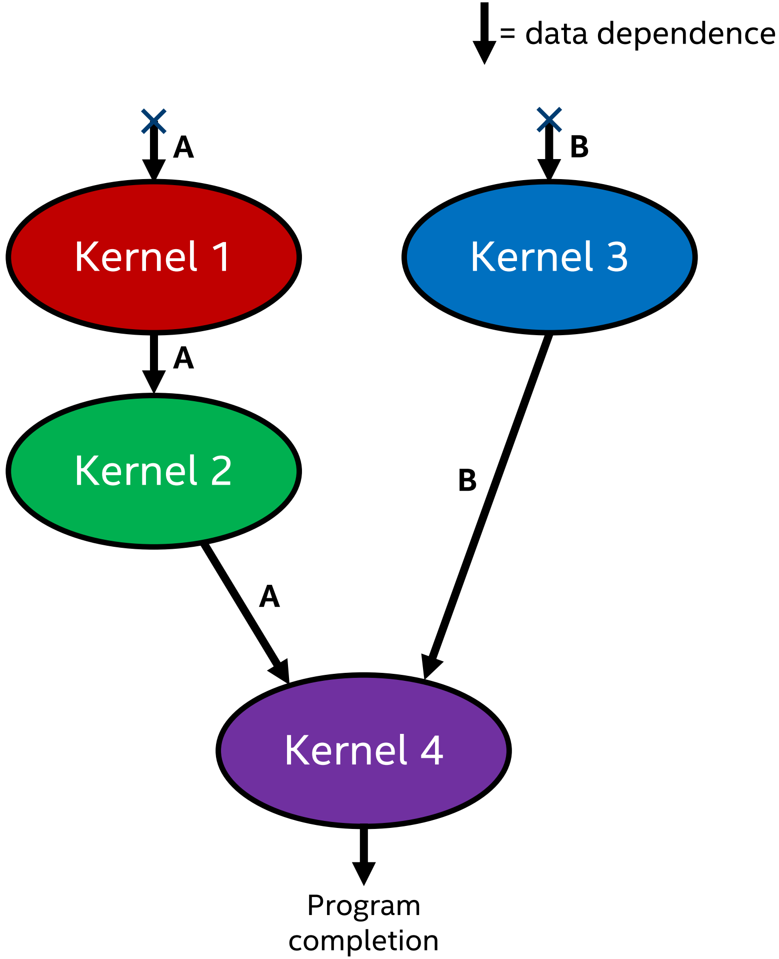
Kernel 2

```
 Q.submit([&](handler& h) {
  auto out = B.get_access<access::mode::write>(h);
  h.parallel_for(rng, [=](id<1> idx) {
   out[idx] = idx[0]; }); });
```

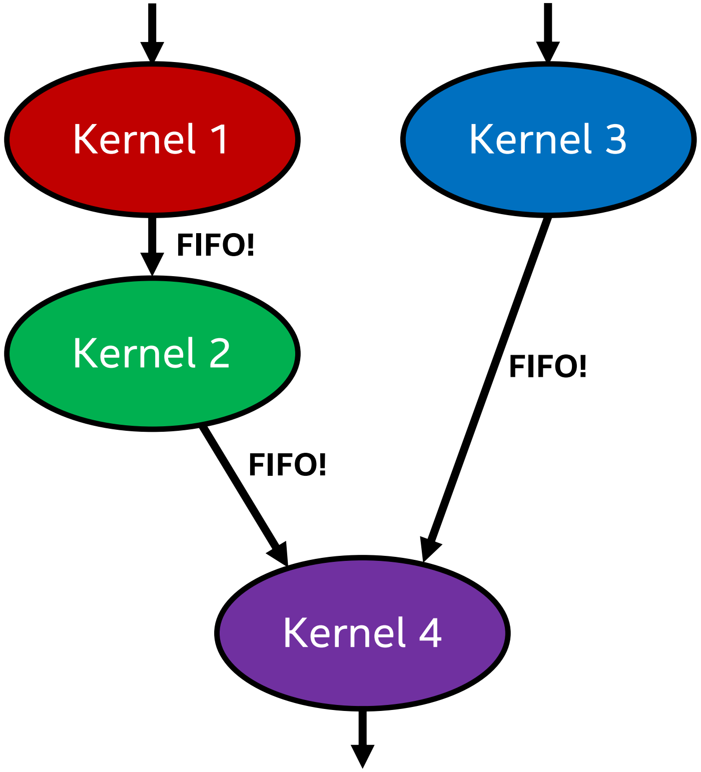
Kernel 3

```
 Q.submit([&](handler& h) {
 auto in = A.get_access<access::mode::read>(h);
 auto inout = B.get_access<access::mode::read_write>(h);
 h.parallel_for(rng, [=](id<1> idx) {
  inout[idx] *= in[idx]; }); });
```

Kernel 4



SYCL RUNTIME KERNEL SCHEDULING



The SYCL runtime graph model

- A data flow graph
- Based on data or control dependencies
- Coarse grained dependencies/sharing

Leverage same model with FIFOs as edges

- Kernels execute concurrently to minimize storage on edges/in FIFOs

Data Flow Pipes

History

- **Intel FPGA:** Channels (static connectivity)
- **OpenCL 2.0 standard:** Pipes (dynamic connectivity at kernel launch time)
- **OpenCL 2.2 standard:** Program pipes (static connectivity at compile time)

New

- **Extension to the SYCL 1.2.1 standard:** Data flow pipe extension
  - https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/DataFlowPipes/data_flow_pipes.asciidoc
  - Guaranteed static connectivity (SYCL programs can be single source!)
  - Type-based approach

Syntax

A pipe is identified by a specialization of:

```
template <class name,
          class dataT,
          size_t min_capacity = 0>
class pipe;
```

Such as:

```
using pipe1 = pipe<class foo, int>;      // Pipe 1
using pipe2 = pipe<class bar, int>;      // Pipe 2
using pipe3 = pipe<class bar, float>;    // Pipe 3
using pipe4 = pipe<class bar, float, 5>; // Pipe 4
```

Syntax (2)

A pipe is identified by a specialization of:

```
template <class name,
          class dataT,
          size_t min_capacity = 0>
class pipe;
```

Such as:

```
using pipe1 = pipe<class foo, int>;      // Pipe 1
using pipe2 = pipe<class bar, int>;      // Pipe 2
using pipe3 = pipe<class bar, float>;    // Pipe 3
using pipe4 = pipe<class bar, float, 5>; // Pipe 4
```

Pipes have blocking and non-blocking members

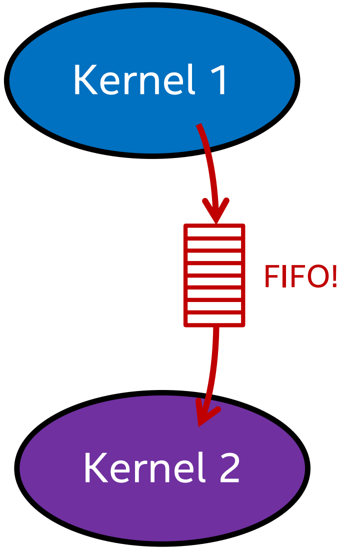
```
template <class name,
          class dataT,
          size_t min_capacity = 0>
class pipe {

  // Blocking
  static dataT read();
  static void write( const dataT &data );

  // Non-blocking
  static dataT read( bool &success_code );
  static void write( const dataT &data,
                     bool &success_code );
}
```

Simple Example

```
// Defining a type alias is the recommended practice
using my_pipe = pipe<class some_pipe, int>;
auto R = range<1>{1024};

myQueue.submit([&](handler& cgh) {
  auto read_acc = readBuf.get_access<access::mode::read>(cgh);

  cgh.parallel_for(R, [=](id<1> idx) {
    my_pipe::write( read_add[idx] );
  });
});

myQueue.submit([&](handler& cgh) {
  auto write_acc = writeBuf.get_access<access::mode::write>(cgh);

  cgh.parallel_for(R, [=](id<1> idx) {
    write_acc[idx] = my_pipe::read();
  });
});
```

} **Kernel 1**

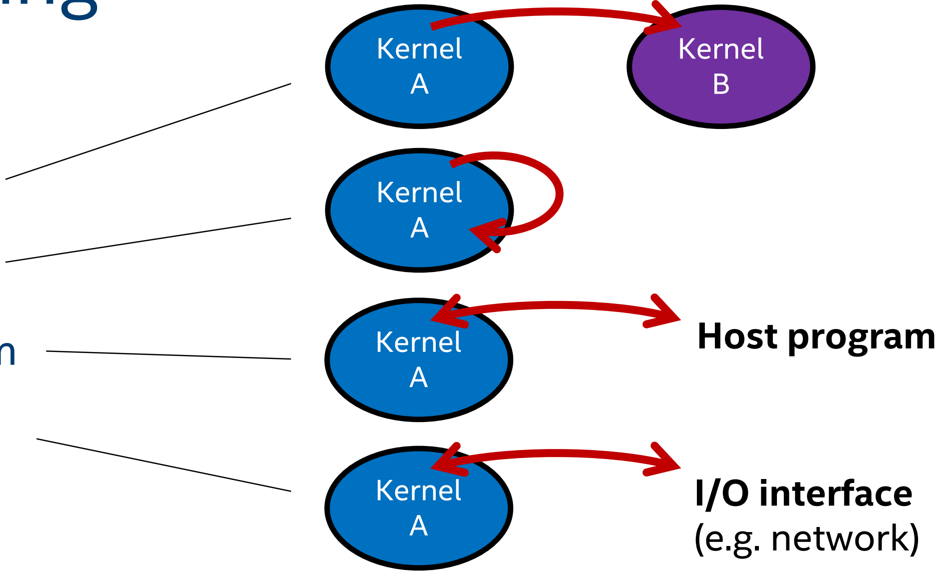
} **Kernel 2**



Connectivity and Lowering

Types of connectivity:

1. **Cross kernel:** Kernel A ⇒ Kernel B
2. **Intra-kernel:** Kernel A ⇒ Kernel A
3. **Host pipe:** Kernel A ⇔ host program
4. **I/O pipe:** Kernel A ⇔ I/O peripheral



Lowering:

- Can lower to OpenCL and SPIR-V representations of OpenCL 2.0 or OpenCL 2.2 pipes
- Can lower to Intel FPGA channels
- Layers on top of significant past investments in optimization

Execution Model

Philosophy

- Don’t incur overhead in the base case, since most applications don’t need it
  - Don’t match Intel FPGA channel cross-work-item loop ordering guarantees

Within a work-item:

1. For **single pipe** (including multiple accesses): Program order applies
2. For **two pipes**: Treated as noalias memory and may be reordered
   - Packetize or add synchronization if required

Across work-items:

- No guarantees. Add synchronization if required

Type Based Approach

Guaranteed static connectivity in device compilers

- Fundamental for performance on FPGAs
- Compiler optimization opportunities

All of C++ applies!

- Metaprogram your own abstractions on top
- Templates and other mechanisms propagate compile-time connectivity
- Use best practices (particularly type aliases)
- Be aware of scoping rules

Future Work

1. Guaranteed concurrent scheduling edges in graph
   - Add SYCL graph edges guaranteeing that two kernels will execute concurrently
2. Extend metaprogramming abstractions on top
   - Publishing array of pipes abstraction. Creating more as templates to extend
3. Type-based approach has some idiosyncrasies
   - Close to global linkage and static storage duration
   - Privatization per invocation requires thought
   - Library interfaces templated on connectivity
   - **Result:** Evaluating abstractions on top, and secondary instance-based interface

Thanks

Intel SYCL implementation open source project

- https://github.com/intel/llvm

Intel extensions for the SYCL standard

- https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions

Feedback

- Issues on the open source project, or email michael.kinsner@intel.com

NOTICES AND DISCLAIMERS

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.
Intel, the Intel logo, and Intel Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

© 2019 Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. Khronos® is a registered trademark and SYCL and SPIR are trademarks of the Khronos Group, Inc. Other names and brands may be claimed as property of others.

intel®