Post-modern C++ abstractions for FPGA & heterogeneous computing with OpenCL SYCL & SPIR-V

ANL REFORM 2016

Ronan Keryell

Xilinx Research Labs

High-level programming for FPGA?

- FPGA $\equiv$ MPSoC with amazing features
- $\exists$ huge libraries waiting for acceleration on them...
- Computation-intensive libraries often written in C/C++ (TernsorFlow (Google...), Caffe (Berkeley, Yahoo...), Torch7 (Facebook...) for DNN...)/Fortran
  - Often with OpenMP or CUDA single-source extensions
  - Often with vector intrinsics or assembly code
- Need fine control of real hardware resources for performance & power efficiency

Can we get higher level-programming in some standard language?

Outline

Outline

C++14

- 2 Open Source compilers available *before* ratification (GCC & Clang/LLVM)
- Strategy: 1 major (C++11) and 1 minor (C++14) version on a 6-year cycle
- Next big version expected in 2017 (C++1z)
  - Already being implemented! ☺
- Well defined memory model for parallel programming!
- Monolithic committee replaced by many smaller *parallel* task forces
  - Parallelism TS (Technical Specification) with Parallel STL
  - Concurrency TS (threads, mutex...)
  - Array TS (multidimensional arrays *à la* Fortran)
  - Transactional Memory TS...

Race to parallelism! Definitely matters for HPC and heterogeneous computing!

C++ is a complete new language

- Forget about C++98, C++03...
- Send your proposals and get involved in C++ committee (pushing heterogeneous computing)!

Modern C++ & HPC *(I)*

- Huge library improvements
  - <thread> library and multithread memory model <atomic> ⇝ HPC
  - Hash-map
  - Algorithms
  - Random numbers
  - ...
- Uniform initialization and range-based for loop

```
std::vector<int> my_vector { 1, 2, 3, 4, 5 };
for (auto &e : my_vector)
  e += 1;
```

Modern C++ & HPC *(II)*

- Easy functional programming style with λ expressions (anonymous functions)

```
std::transform(std::begin(v), std::end(v), [] (int e) { return 2*e; });
```

- Lot of meta-programming improvements to make meta-programming ~~easy~~ easier: variadic templates, type traits <type_traits>...
  - Make simple things simpler to be able to write generic numerical libraries, etc.

Modern C++ & HPC *(III)*

- Automatic type inference for terse programming
  - Python 3.x (interpreted):

```
def add(x, y):
  return x + y
print(add(2, 3))      # 5
print(add("2", "3")) # 23
print(add(2, "Boom")) # Fails at run-time :-(
```

  - Same in C++14 but compiled + static compile-time type-checking:

```
auto add = [] (auto x, auto y) { return x + y; };
std::cout << add(2, 3) << std::endl;        // 5
std::cout << add("2"s, "3"s) << std::endl; // 23
std::cout << add(2, "Boom"s) << std::endl; // Does not compile :-)
```

  Without using templated code! ~~`template <typename >`~~ ☺

Modern C++ & HPC *(IV)*

- R-value references & std::move semantics
  - matrix_A = matrix_B + matrix_C
    - Avoid copying (TB, PB, EB... ☹) when assigning or function return
- Avoid raw pointers, `malloc()`/`free()`/`delete[]`: use references and smart pointers instead

```
// Allocate a double with new() and wrap it in a smart pointer
auto gen() { return std::make_shared<double> { 3.14 }; }
[...]
{
  auto p = gen(), q = p;
  *q = 2.718;
  // Out of scope, no longer use of the memory: deallocation happens here
}
```

Modern C++ & HPC (V)

- C++14 generalizes constexpr to statements

```
constexpr auto fibonacci(int v) {
  long long int u_n_minus_1 = 0;
  auto u_n = u_n_minus_1 + 1;
  for (int i = 1; i < v; ++i) {
    auto tmp = u_n;
    u_n += u_n_minus_1;
    u_n_minus_1 = tmp;
  }
  return u_n;
}
int main() {
  constexpr auto result = fibonacci(80);
  std::cout << result << std::endl;
  return 0;
}
```

Modern C++ & HPC *(VI)*

Compiled to

```
movabsq $23416728348467685, %rsi # imm = 0x533163EF0321E5
movl    $_ZSt4cout, %edi
callq   _ZNSo9_M_insertIxEERSoT_
```

- Lot of other amazing stuff...
- Allow both low-level & high-level programming...
  - Great for heterogeneous computing, embedded & HPC!
- "The C++ Core Guidelines are a set of tried-and-true guidelines, rules, and best practices about coding in C++"
  `http://isocpp.github.io/CppCoreGuidelines`

Outline

C++11 std::thread

- It's all in the standard! http://en.cppreference.com/w/cpp/thread/thread

```
#include <chrono>
#include <iostream>
#include <thread>

void f(int n) {
  std::cout << "Thread " << n << "executing" << std::endl;
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
}

  // Launch f in thread t:
  std::thread t(f, 1);
  // The same with a lambda
  std::thread t2([] {std::cout << "Hello!" << std::endl;});
```

- ∃ higher-level constructs: std::async, std::future/std::promise

C++11 std::async

- Function objects that *may* be executed in parallel by some thread pool

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end) {
  auto len = end - beg;
  if (len < 1000)
    // Small enough for quicker sequential execution
    return std::accumulate(beg, end, 0);

  // Divide and conquer the world!
  RAIter mid = beg + len/2;
  // This returns a std::future
  auto handle = std::async(std::launch::async,
                           parallel_sum<RAIter>, mid, end);
  int sum = parallel_sum(beg, mid);
  // Get the std::future value from the inner std::promise
  return sum + handle.get();
}

int main() {
  // 10000 elements initialized with 1
  std::vector<int> v(10000, 1);
  std::cout << "sum = "
            << parallel_sum(v.begin(), v.end()) << std::endl;
}
```

- std::future/std::promise can transfer exceptions across threads too

http://en.cppreference.com/w/cpp/thread/async

Example of C++ library: TBB

- Open Source library started by Intel
- Threads + work-stealing scheduling
- Containers, algorithms, scalable allocators
- Data-flow graph with message queues

Example of C++ extensions: OpenMP 4.5 *(I)*

- Single source #pragma extensions
- Support C, C++, Fortran with same #pragma
  - Other unofficial standards (Pythran for Python-to-C++ translation...)
- Task-graph oriented programming
- Multithread SMP support
- Loop parallelization
- Relaxed memory consistency model
- SIMD vectorization
- Accelerator offloading with hierarchical parallelism
- Transactional memory (optimistic speculative execution)
- Runtime API (environment queries, locks)

Example of C++ extensions: OpenMP 4.5 *(II)*

```
int fib(int n) {
  int i, j;
  if (n < 2)
    return n;
  else {
#pragma omp task shared(i)
    i = fib(n - 1);
#pragma omp task shared(j)
    j = fib(n - 2);
#pragma omp taskwait
    return i + j;
  }
}
```

Position argument

Which language for unified heterogeneous computing?

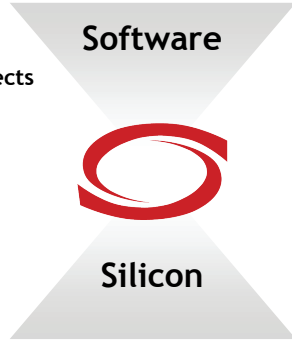
-  Entry cost
- ∃ thousands of dead parallel languages...
  -  Exit cost
- Use standard solutions with open source implementations
- Start with modern C++
  - Very successful & ubiquitous language
  - Very active since C++11 (C++14, C++1z...)
  - Interoperability: seamless interaction with embedded world, libraries, OS...
  - Combine both low-level aspects with high-level programming
    - Pay only for what you need (garbage collector or not...)
  - Classes can be used to define Domain Specific Embedded Language (DSEL)
  - Not directly targeting FPGA...
    - But extensible through classes (⇝ DSEL)
    - Extensible with #pragma (already in Vivado HLS, SDSoC & SDAccel) and attributes

Outline

Khronos Connects Software to Silicon

Open Consortium creating OPEN STANDARD APIs for hardware acceleration
Any company is welcome – many international members – one company one vote

ROYALTY-FREE specifications
State-of-the art IP framework protects members AND the standards



International, non-profit organization
Membership fees cover operating and engineering expenses

Low-level silicon APIs needed on every platform
Graphics, parallel compute, rich media, vision, sensor and camera processing

API Specifications AND Conformance Tests for cross-vendor portability

Strong industry momentum
100s of man years invested by industry experts

Well over a *BILLION* people use Khronos APIs *Every Day…*

Outline

Interoperability nightmare in heterogeneous computing & graphics

- ∃ Many programming languages for heterogeneous computing
  - Writing compiler front-end may not be *the* real value for a hardware vendor...
    - Writing a C++1z compiler from scratch is almost impossible...
- ∃ Many programming languages for writing shaders
- Convergence in computing (Compute Unit) & graphics (Shader) architectures
  - Same front-end & middle-end compiler optimizations
- Need for some non source-readable portable code for IP protection

⇝ Defining common low-level representation !

SPIR-V transforms the language ecosystem

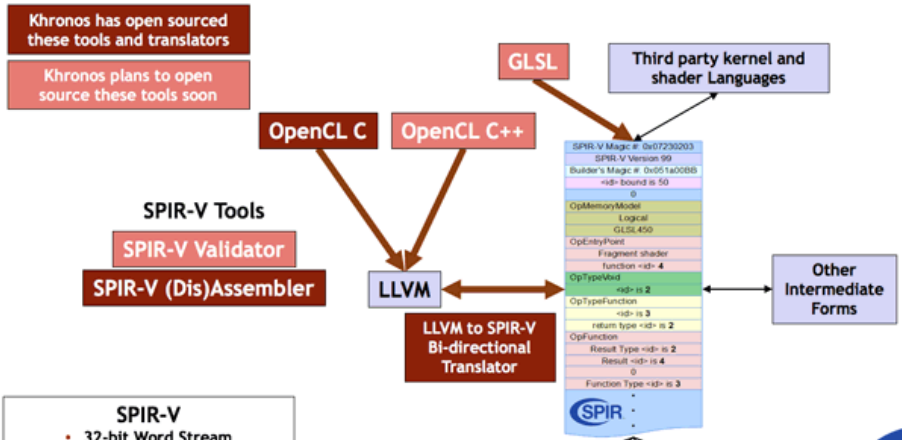
- First multi-API, intermediate language for parallel compute *and* graphics
  - Native representation for Vulkan shader and OpenCL kernel source languages
  - `https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf`
- Cross-vendor intermediate representation
  - Language front-ends can easily access multiple hardware run-times
  - Acceleration hardware can leverage multiple language front-ends
  - Encourages tools for program analysis and optimization in SPIR form

Evolution of SPIR family

SPIR	SPIR 1.2	SPIR 2.0	SPIR-V 1.0
LLVM Interaction	Uses LLVM 3.2	Uses LLVM 3.4	100% Khronos defined Round-trip lossless conversion
Compute Constructs	Metadata/Intrinsics	Metadata/Intrinsics	Native
Graphics Constructs	No	No	Native
Supported Language Feature Sets	OpenCL C 1.2	OpenCL C 1.2 OpenCL C 2.0	OpenCL C 1.2 / 2.0 OpenCL C++ and GLSL
OpenCL Ingestion	OpenCL 1.2 Extension	OpenCL 2.0 Extension	OpenCL 2.1 Core OpenCL 1.2 / 2.0 Extensions
Vulkan Ingestion	-	-	Vulkan 1.0 Core

Not based on LLVM to isolate from LLVM roadmap changes

Driving SPIR-V Open Source ecosystem



SPIR-V 1.0 Resources

- SPIR-V 1.0 specification available in Khronos Registry
  `https://www.khronos.org/registry/spir-v`
- Feedback forum for questions and feedback
  `https://forums.khronos.org/showthread.php/12919-Feedback-SPIR-V`
- Whitepaper “An Introduction to SPIR-V”
  `https://www.khronos.org/registry/spir-v/papers/WhitePaper.html`
- Bug reporting
  `https://www.khronos.org/bugzilla/enter_bug.cgi?product=SPIR-V`
- SPIR-V tools project including an assembler, binary module parser, disassembler & validator for SPIR-V `https://github.com/KhronosGroup/SPIRV-Tools`
- LLVM framework with SPIR-V support including an LLVM↔SPIR-V bi-directional converter `https://github.com/KhronosGroup/SPIRV-LLVM`
- GLSL compiler in development

Outline

Missing link...

- No tool providing
  - Modern C++ environment
  - Heterogeneous computing
  - Single source for programming productivity
  - OpenCL interoperability

What about heterogenous computing???

- C++ std::thread is great...
- ...but supposed shared unified memory (SMP) ☹
  - What if accelerator with own separate memory? Not same address space?
  - What if using distributed memory multi-processor system (MPI...)?
- ⇝ Extend the concepts...
  - Replace raw unified-memory with buffer objects
  - Define with accessor objects which/how buffers are used
  - Since accessors are already here to define dependencies, no longer need for std::future/std::promise! ☺
  - Add concept of queue to express where to run the task
  - Also add all goodies for massively parallel accelerators (OpenCL/Vulkan/SPIR-V) in clean C++

SYCL ≡ pure C++14 DSEL

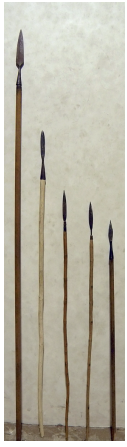
- Implement concepts useful for heterogeneous computing
- Asynchronous task graph
- Buffers to define location-independent storage
- Accessors to express usage for buffers and pipes: read/write/...
- Hierarchical parallelism
- Hierarchical storage
  - Rely on C++ allocator to specify storage
- Single source programming model
  - Take advantage of CUDA on steroids & OpenMP simplicity and power
  - Compiled for host *and* device(s)
  - Enabling the creation of C++ higher level programming models & C++ templated libraries
- Most modern C++ features available for OpenCL
  - Programming interface based on abstraction of OpenCL components (data management, error handling...)
  - Provide OpenCL interoperability
- Host fallback (debug and symmetry for SIMD/multithread on host)
- Directly executable DSEL
- Host emulation for free & no compiler needed for experimenting

Puns and pronunciation explained

OpenCL SYCL

sickle ['si-kəl]

OpenCL SPIR



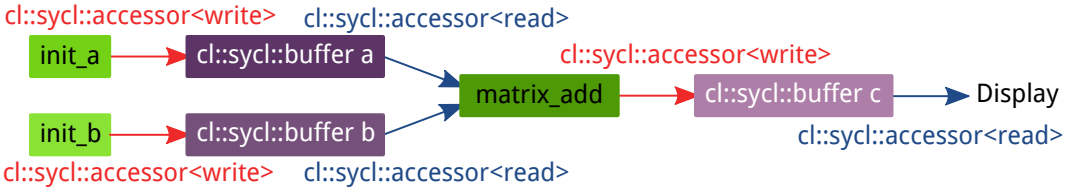
spear ['spir]

Complete example of matrix addition in OpenCL SYCL

```
#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

constexpr size_t N = 2;
constexpr size_t M = 3;
using Matrix = float[N][M];

// Compute sum of matrices a and b into c
int main() {
 Matrix a = { { 1, 2, 3 }, { 4, 5, 6 } };
 Matrix b = { { 2, 3, 4 }, { 5, 6, 7 } };

 Matrix c;

 {// Create a queue to work on default device
  queue q;
  // Wrap some buffers around our data
  buffer<float, 2> A { &a[0][0], range<2> { N, M } };
  buffer<float, 2> B { &b[0][0], range<2> { N, M } };
  buffer<float, 2> C { &c[0][0], range<2> { N, M } };
  // Enqueue some computation kernel task
  q.submit([&](handler& cgh) {
   // Define the data used/produced
   auto ka = A.get_access<access::read>(cgh);
   auto kb = B.get_access<access::read>(cgh);
   auto kc = C.get_access<access::write>(cgh);
   // Create & call kernel named "mat_add"
   cgh.parallel_for<class mat_add>(range<2> { N, M },
      [=](id<2> i) { kc[i] = ka[i] + kb[i]; }
   );
  }); // End of our commands for this queue
 } // End scope, so wait for the queue to complete.
 // Copy back the buffer data with RAII behaviour.
 std::cout << "c[0][2] = " << c[0][2] << std::endl;
 return 0;
}
```

Asynchronous task graph model

- Change example with initialization kernels instead of host
- Theoretical graph of an application described *implicitly* with kernel tasks using buffers through accessors



- Possible schedule by SYCL runtime:

⇝ Automatic overlap of kernels & communications
  - Even better when looping around in an application
  - Assume it will be translated into pure OpenCL event graph
  - Runtime uses as many threads & OpenCL queues as necessary (GPU synchronous queues, AMD compute rings, AMD DMA rings...)

Task graph programming — the code

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl::sycl;
// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;
int main() {
  { // By sticking all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block

    // Create a queue to work on default device
    queue q;
    // Create some 2D buffers of float for our matrices
    buffer<double, 2> a({ N, M });
    buffer<double, 2> b({ N, M });
    buffer<double, 2> c({ N, M });
    // Launch a first asynchronous kernel to initialize a
    q.submit([&](auto &cgh) {
        // The kernel write a, so get a write accessor on it
        auto A = a.get_access<access::write>(cgh);

        // Enqueue parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class init_a>({ N, M },
                                       [=] (auto index) {
                                         A[index] = index[0]*2 + index[1];
                                       });
      });
    // Launch an asynchronous kernel to initialize b
    q.submit([&](auto &cgh) {
        // The kernel write b, so get a write accessor on it
        auto B = b.get_access<access::write>(cgh);
        /* From the access pattern above, the SYCL runtime detect
           this command_group is independant from the first one
           and can be scheduled independently */

        // Enqueue a parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class init_b>({ N, M },
                                       [=] (auto index) {
                                         B[index] = index[0]*2014 + index[1]*42;
                                       });
      });
    // Launch an asynchronous kernel to compute matrix addition c = a + b
    q.submit([&](auto &cgh) {
        // In the kernel a and b are read, but c is written
        auto A = a.get_access<access::read>(cgh);
        auto B = b.get_access<access::read>(cgh);
        auto C = c.get_access<access::write>(cgh);
        // From these accessors, the SYCL runtime will ensure that when
        // this kernel is run, the kernels computing a and b completed

        // Enqueue a parallel kernel on a N*M 2D iteration space
        cgh.parallel_for<class matrix_add>({ N, M },
                                           [=] (auto index) {
                                             C[index] = A[index] + B[index];
                                           });
      });
    /* Request an access to read c from the host-side. The SYCL runtime
       ensures that c is ready when the accessor is returned */
    auto C = c.get_access<access::read, access::host_buffer>();
    std::cout << std::endl << "Result:" << std::endl;
    for(size_t i = 0; i < N; i++)
      for(size_t j = 0; j < M; j++)
        // Compare the result to the analytic value
        if (C[i][j] != i*(2 + 2014) + j*(1 + 42)) {
          std::cout << "Wrong value " << C[i][j] << " on element "
                    << i << ' ' << j << std::endl;
          exit(-1);
        }
  } /* End scope of q, this wait for any remaining operations on the
       queue to complete */
  std::cout << "Good computation!" << std::endl;
  return 0;
}
```

Tasks with host dependencies and host accessors

```
#include <CL/sycl.hpp>
#include <iterator>
#include <numeric>
  cl::sycl::buffer<Type> b { N };
  {
    auto ab = b.get_access<cl::sycl::access::write>();
    // Initialize buffer b starting from the end with increasing
    // integer starting at 42
    std::iota(ab.rbegin(), ab.rend(), 42);
  }
  // A buffer of N Type to get the result
  cl::sycl::buffer<Type> c { N };
  // Launch a kernel to do the summation
  q.submit([&] (cl::sycl::handler &cgh) {
      // Get access to the data
      auto ab = b.get_access<cl::sycl::access::read>(cgh);
      auto ac = c.get_access<cl::sycl::access::write>(cgh);

      cgh.single_task<class sum>([=] {
        [...]
        });
    });
  // Wait for c to be available through this host accessor
  for(auto e : c.get_access<cl::sycl::access::read>())
    BOOST_CHECK(e == N + 42 - 1);
```

From work-groups & work-items to hierarchical parallelism

```
constexpr int size = 10;
int data[size];
constexpr int gsize = 2;
buffer<int> my_buffer { data, size };

my_queue.submit([&](auto &cgh) {
 auto in = my_buffer.get_access<access::read>(cgh);
 auto out = my_buffer.get_access<access::write>(cgh);
 // Iterate on the work-group
 cgh.parallel_for_workgroup<class hierarchical>({ size,
                                                  gsize },
   [=](group<> grp) {
     // Code executed only once per work-group
     std::cerr << "Gid=" << grp[0] << std::endl;
     // Iterate on the work-items of a work-group
     cgh.parallel_for_workitem(grp, [=](item<1> tile) {
       std::cerr << "id␣=" << tile.get_local()[0]
                 << "␣" << tile.get_global()[0]
                 << std::endl;
       out[tile] = in[tile] * 2;
     });
   // Can have other cgh.parallel_for_workitem() here...
   });
});
```

Very close to OpenMP 4 style! ☺

- Easy to understand the concept of work-groups
- Easy to write work-group only code
- Replace code + barriers with several parallel_for_workitem()
  - Performance-portable between CPU and device
  - No need to think about barriers (automatically deduced)
  - Easier to compose components & algorithms
  - Ready for future device with non uniform work-group size

C++11 allocators

- ∃ C++11 `allocator`s to control the way objects are allocated in memory
  - For example to allocate some vectors on some storage
  - Concept of `scoped_allocator` to control storage of nested data structures
  - Example: vector of strings, with vector data and string data allocated in different memory areas (speed, power consumption, caching, read-only...)
- SYCL reuses `allocator` to specify how `buffer` and `image` are allocated on the host side

Outline

Pipes in OpenCL 2.*x*

- Simple FIFO objects
- Useful to create dataflow architectures between kernels without host
- Created on the host with some message size + object number
- `read()`/`write()` functions
- Same behaviour/guaranty as a memory buffer
  - For portability because no hardware FIFO mandatory in OpenCL 2.*x*
  - Can be implemented with a memory buffer
- Non blocking because no independent-forward-progress guaranty in execution model yet
  - No guaranty that a producer can run concurrently with a consumer
  - No guaranty between different work-items when blocking

Pipes on FPGA

- The actual motivation for pipes in OpenCL standard!
- External memory access main cause for power consumption... ☹
- Real FIFO are easy to implement in hardware
  - Simple bus for 1-element FIFO
  - Latches or memory when more elements
- Very energy efficient
- Possible to have full dataflow applications without host control
- ⇝ FPGA vendors provide OpenCL extensions for pipe with stronger guarantees
  - Blocking pipes ⇝ simpler applications
  - Static size possible ⇝ direct synthesis
  - Independent work-groups and kernels for producers/consumers connected with pipes

⇝ Xilinx evaluates pipe extensions for SYCL too

Producer/consumer with blocking pipe

```
#include <CL/sycl.hpp>
#include <iostream>
#include <iterator>

constexpr size_t N = 3;
using Vector = float[N];

int main() {
  Vector va = { 1, 2, 3 };
  Vector vb = { 5, 6, 8 };
  Vector vc;

  {
    // Create buffers from a & b vectors
    cl::sycl::buffer<float> ba { std::begin(va), std::end(va) };
    cl::sycl::buffer<float> bb { std::begin(vb), std::end(vb) };

    // A buffer of N float using the storage of vc
    cl::sycl::buffer<float> bc { vc, N };

    // A pipe of 2 float elements
    cl::sycl::pipe<float> p { 2 };

    // Create a queue to launch the kernels
    cl::sycl::queue q;

    // Launch the producer to stream A to the pipe
    q.submit([&](cl::sycl::handler &cgh) {
      // Get write access to the pipe
      auto kp = p.get_access<cl::sycl::access::write,
                             cl::sycl::access::blocking_pipe>(cgh);
      // Get read access to the data
      auto ka = ba.get_access<cl::sycl::access::read>(cgh);

      cgh.single_task<class producer>([=] {
          for (int i = 0; i != N; i++)
            kp << ka[i];
        });
      });

    // Launch the consumer that adds the pipe stream with B to C
    q.submit([&](cl::sycl::handler &cgh) {
      // Get read access to the pipe
      auto kp = p.get_access<cl::sycl::access::read,
                             cl::sycl::access::blocking_pipe>(cgh);

      // Get access to the input/output buffers
      auto kb = bb.get_access<cl::sycl::access::read>(cgh);
      auto kc = bc.get_access<cl::sycl::access::write>(cgh);

      cgh.single_task<class consumer>([=] {
          for (int i = 0; i != N; i++)
            kc[i] = kp.read() + kb[i];
        });
      });
  } /*< End scope for the queue and the buffers:
       wait for completion q completion & bc copied back to v */

  std::cout << std::endl << "Result:" << std::endl;
  for(auto e : vc)
    std::cout << e << "_";
  std::cout << std::endl;
}
```

Non blocking pipe

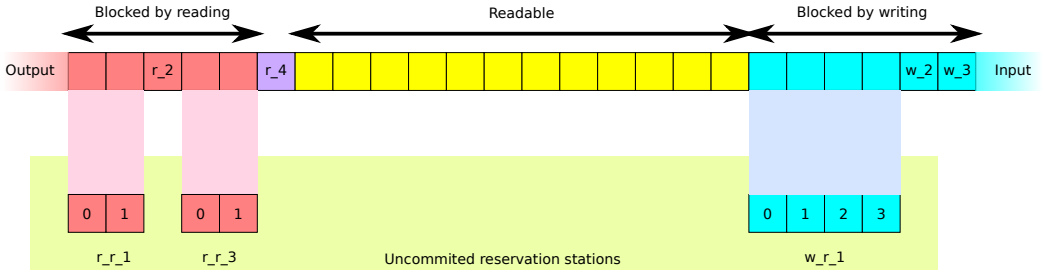
```
// Launch the producer to stream A to the pipe
q.submit([&](cl::sycl::handler &cgh) {
  // Get write access to the pipe
  auto p = P.get_access<cl::sycl::access::write>(cgh);
  // Get read access to the data
  auto ka = A.get_access<cl::sycl::access::read>(cgh);

  cgh.single_task<class producer>([=] {
      for (int i = 0; i != N; i++)
        // Try to write to the pipe up to success
        while (!(p << ka[i]))
          ;
    });
  });
```

pipe access are sequential...

- FIFO: queue with serialized access
- How to implement simultaneous access by several work-item either at input or output?
- How to order access by work-items for deterministic execution?

⇝ Add concept of reservation station

Parallel pipe access with reservation



- Reservation station $\equiv$ array-view reserved in the pipe
- Allow ordered and parallel operation inside pipes
- Accessible up to commit operation
- Several reservation stations alive in parallel
- Can be mixed with on-going simple pipe access

Code example with reservation station

```
// Size of the buffers
constexpr size_t N = 200;
// Number of work-item per work-group
constexpr size_t WI = 20;
// The plumbing with some weird size prime to WI to exercise the system
cl::sycl::pipe<Type> pa { 2*WI + 7 };
// A buffer of N Type to get the result
cl::sycl::buffer<Type> c { N };
q.submit([&] (cl::sycl::handler &cgh) {
    // Get read access to the pipe
    auto apa = pa.get_access<cl::sycl::access::read,
                             cl::sycl::access::blocking_pipe>(cgh);
    // Get write access to the data
    auto ac = c.get_access<cl::sycl::access::write>(cgh);

    /* Create a kernel with WI work-items executed by work-groups of
       size WI, that is only 1 work-group of WI work-items */
    cgh.parallel_for_work_group<class consumer>(
      { WI, WI },
      [=] (auto group) {
        // Use a sequential loop in the work-group to stream chunks in order
        for (int start = 0; start != N; start += WI) {
          auto r = apa.reserve(WI);
          group.parallel_for_work_item([=] (cl::sycl::item<> i) {
              ac[start + i[0]] = r[i[0]];
            });
          // Here the reservation object goes out of scope: commit
        }
      });
  });
```

Co-scheduling

- Pipe connections make sense on FPGA for kernels... running at the same time ☺
  - Take advantage of direct 1-to-1 connections
  - On pageable FPGA with kernel reconfiguration, pipe-connected kernels need to be present at the same time
- SYCL provides a nice asynchronous execution model for distributed-memory system and accelerators
  - No provision for hard pipe-dependencies yet ☹
- ∃ co-scheduling: simultaneously scheduling of interdependent tasks
  - Useful on HPC machine to avoid resource waste
- ⇝ SYCL scheduler needs to consider pipe-based relations too...

Exascale-ready

- Use your own C++ compiler
  - Only kernel outlining needs SYCL compiler
- SYCL as plain single-source C++ can address most of the hierarchy levels
  - MPI
  - OpenMP
  - C++-based PGAS (Partitioned Global Address Space) DSEL (Domain-Specific embedded Language, such as Coarray C++...)
  - Remote accelerators in clusters
  - Use SYCL buffer allocator for
    - RDMA
    - Out-of-core, mapping to a file
    - PiM (Processor in Memory)
    - ...

Using SYCL-like models in other areas

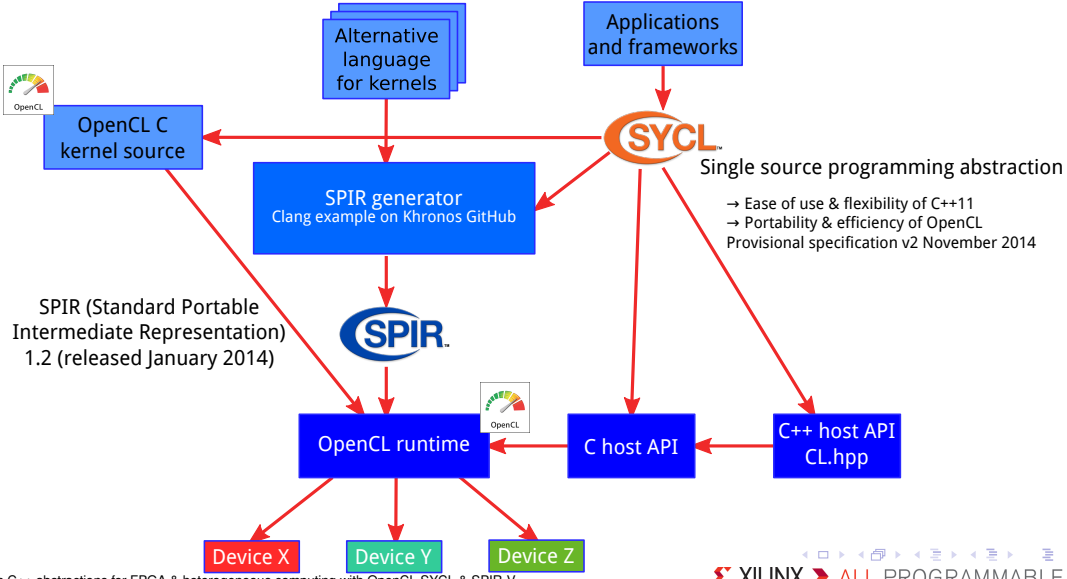
- SYCL ≡ generic heterogeneous computing model beyond OpenCL
  - `device` abstracts the accelerators
  - `queue` allows to launch tasks with computations overlapping communications and pipelining
  - `parallel_for<>` for // computations
  - `accessor` defines the way we access data
  - `buffer` to chose where to store data
  - `allocator` for defining how data are allocated/backed and how pointers work
- Example in PiM (Processor-in-Memory)/Near-Memory Computing world
  - Use `queue` to run on some PiM chips
  - Use `allocator` to distribute data structures or to allocate `buffer` in special memory (memory page, chip...)
  - Use `accessor` to use alternative data access (split address from computation, streaming only, PGAS...)
  - Use `pointer_trait` to use specific way to interact with memory such as bank/transposition or relocation

SYCL C++ for FPGA

- Xilinx FPGA
  - Clocks
  - AXI ports
  - AXI stream ports
  - Interrupt ports
  - I/O devices (Ethernet, Interlaken, ADC, DAC, video...)
  - Reset
  - Dynamic Voltage and Frequency Scaling (DVFS)
  - Dynamic reconfiguration
  - Kernel scheduling
- Use native kernels to access specific I/O & IP
  - Single source C++ ⇝ hidden in “normal” class interface

- Add location/placement in device & sub-device selectors
- Use C++11 allocators to select memory type & location
- Use accessors to define read/write/bus (AXI4 master/slave/...)/pipe/linear/... data access
- Use SystemC-like data types for user-defined size & precision
- C++: normal API to control run-time & OS
- Tool metadata can be moved optionally from XML/TCL/JSON/... into C++ classes
  - Metaprogramming HLS ☺

SYCL in OpenCL ecosystem



Parallel STL towards C++17 proposal

- Current Parallel STL from C++17 proposal N4507 (2015/05/05)
  `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf`

```
// Current C++11: standard sequential sort
std::sort(vec.begin(), vec.end());
// C++17: permitting parallel execution and vectorization as well
sort(std::experimental::parallel::par_vec, vec.begin(), vec.end());
```

- Easy to implement in SYCL
  - Could even be extended to give a kernel name (profile, debug...):
  - Load balancing between CPU and accelerator

```
sycl_policy<class kernelName1> pol;
sort(pol, begin(vec), end(vec));

sycl_policy<class kernelName2> pol2;
// But SYCL allows OpenCL intrinsics in the operation too
for_each(pol2, vec.begin(), vec.end(),
         [](float & ans) { ans += cl::sycl::sin(ans); });
```

Open Source implementation ☺ `https://github.com/KhronosGroup/SyclParallelSTL`

Outline

Known implementations of SYCL

- ComputeCPP by Codeplay `https://www.codeplay.com/products/computecpp`
  - Most advanced SYCL 1.2 implementation
  - Outlining compiler generating SPIR
  - Run on any OpenCL device and CPU
- sycl-gtx `https://github.com/ProGTX/sycl-gtx`
  - Open source
  - No (outlining) compiler ⇝ use some macros with different syntax
- triSYCL `https://github.com/amd/triSYCL`
  - Open Source
  - Some extensions (Xilinx blocking pipes)
  - No (outlining) compiler ⇝ no device support yet

Standard is still moving & no full implementation yet

triSYCL

- Open Source implementation using templated C++1z classes
  - On-going implementation started at AMD and now led by Xilinx
  - `https://github.com/amd/triSYCL`
  - 7 contributors
- Used by Khronos committee to define the standard
  - Languages are now too complex to be defined without implementation
- Pure C++ implementation & CPU-only implementation for now
  - Use OpenMP for computation + `std::thread` for task graph
  - Rely on STL & Boost for zen style
  - Quite useful for debugging
  - CPU emulation for free
- Looking for some interns ☺ to add outlining compiler to generate SPIR-V based on open source Clang/LLVM, etc.

Future

- SYCL 2.*x* is coming to match OpenCL 2.*x* hardware features
  - SVM
  - kernel side enqueue
  - ...
- SYCL DSEL task graph model is pretty generic and not only OpenCL-centric
  - Close to run-time such as StarPU, Nanos++... and can deal with remote nodes, even with lower level API such as MPI, MCAPI...
  - Actually even not restricted to C++ either (SYPyCL, SYJaCL, SYJSCL, SYCaml...). SYFortranCL on top of Fortran 2008?

Conclusion

- Modern FPGA are complex MP-SoC
- Full systems & HPC machines add another complexity level
- Modern C++ can be used for portable single-source DSEL targeting heterogeneous systems
- SYCL C++ Khronos standard provides seamless single-source with OpenCL interoperability
  - Can be used to improve other higher-level frameworks
- SYCL $\equiv$ pure C++ $\rightsquigarrow$ integration with other C/C++ HPC frameworks: OpenCL, OpenMP, libraries (MPI, numerical), C++ DSeL (PGAS...)...
- SYCL interesting as co-design tool for architectural & programming model exploration (FPGA, PiM/Near-Memory Computing, various computing models...)
  - Inspirational to future OpenCL C++ kernel language
  - Built on top of open source projects & HLS tools
    - SPIR-V gives portable execution model
- Modern C++ is not just C program in `.cpp` file ☺ $\rightsquigarrow$ Invest in learning modern C++
  - But can also use SYCL in a old-C style programming... ☺