Beyond SYCL™ 1.2 : SYCL™ 2.2

Maria Rovatsou, SYCL spec editor

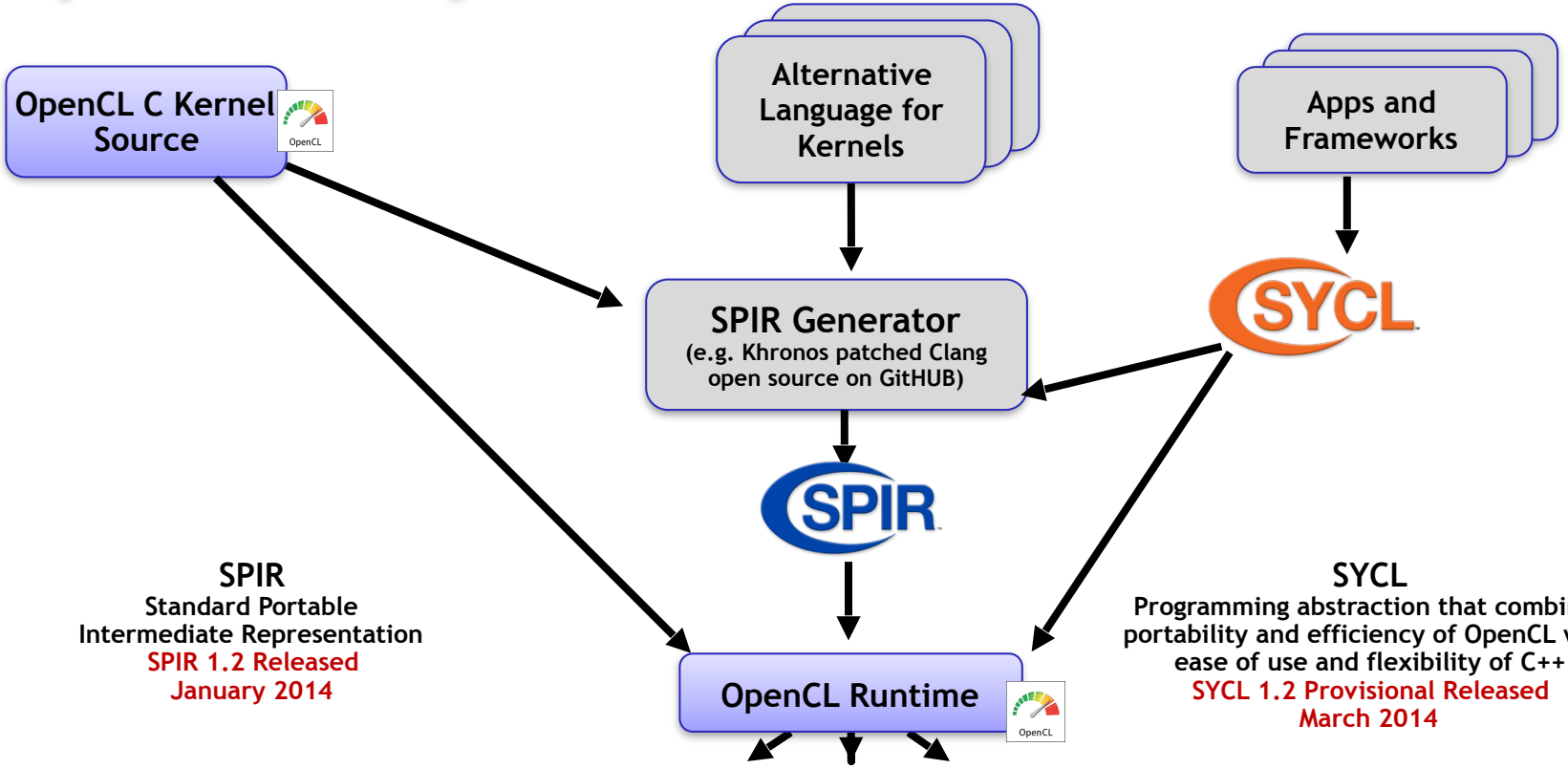
Principal Software Engineer, Codeplay

SYCL 2.2

- SYCL in the OpenCL ecosystem
- Aims of SYCL 2.2 Provisional Specification
- More features, More applications
- Roadmap for SYCL

KHRONOS GROUP

OpenCL Ecosystem



SPIR
Standard Portable Intermediate Representation
SPIR 1.2 Released
January 2014

SYCL
Programming abstraction that combines portability and efficiency of OpenCL with ease of use and flexibility of C++
SYCL 1.2 Provisional Released
March 2014

What we want to achieve

- SYCL single source, modern C++ programming model exposing OpenCL 2.2 feature-set

What we want to achieve

- SYCL single source, modern C++ programming model exposing OpenCL 2.2 feature-set
- Follow current C++ standard developments and enhance integration of OpenCL with developments in the C++ standard

What we want to achieve

- SYCL single source, modern C++ programming model exposing OpenCL 2.2 feature-set
- Follow current C++ standard developments and enhance integration of OpenCL with developments in the C++ standard
- Enhance the Kronos ecosystem and target SPIR-V 1.1 for compute.

What we want to achieve

- SYCL single source, modern C++ programming model exposing OpenCL 2.2 feature-set
- Follow current C++ standard developments and enhance integration of OpenCL with developments in the C++ standardEnhance the Kronos ecosystem and target SPIR-V 1.1 for compute.
- Full and seamless backwards compatibility with SYCL 1.2

What we want to achieve

- SYCL single source, modern C++ programming model exposing OpenCL 2.2 feature-set
- Follow current C++ standard developments and enhance integration of OpenCL with developments in the C++ standardEnhance the Kronos ecosystem and target SPIR-V 1.1 for compute.
- Full and seamless backwards compatibility with SYCL 1.2
- We want to feedback from the OpenCL and C++ communities on this first provisional specification for SYCL 2.2

KHRONOS GROUP

SYCL 2.2:

More features, More applications

SYCL Command Groups

Command Group

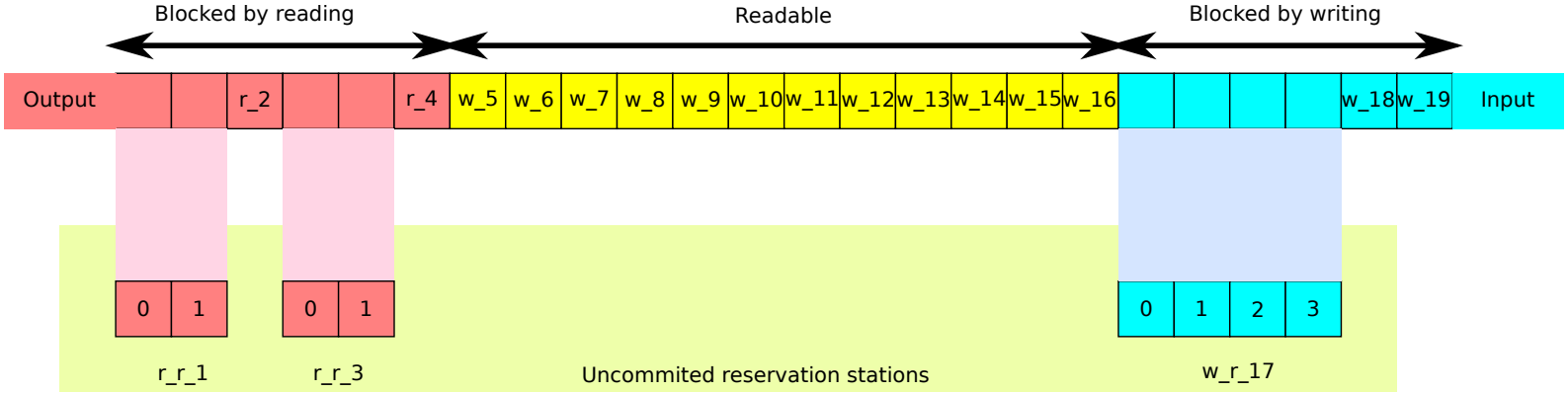
All of the OpenCL commands for memory object creation, copying, mapping and synchronisation operations to correctly execute a kernel on a device are defined in a functor and called a command group.

```
// wrap our data variable in a buffer
buffer<int, 1> resultBuf(data, range<1>(1024));

// create a command_group to issue commands to the queue
myQueue.submit([&](execution_handle<opencl22>& cgh) {
  // request access to the buffer
  auto writeResult = resultBuf.get_access<access::mode::write>(cgh);

  // enqueue a prallel_for task
  cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
    writeResult[idx[0]] = idx[0];
  });  // end of the kernel function
});    // end of our commands for this queue
```

SYCL Pipes

Pipe

Pipes are communication channels between kernels. More conceptually, a pipe memory object is an ordered sequence of data items that stores data organised as a FIFO (first in, first out). A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. Pipe data is not accessible from the host.



SYCL Pipes

static_pipe

Is a pipe with constexpr capacity and is defined for only one target device.

```
constexpr size_t N = 3;

// A static-scoped pipe of 4 float elements
cl::sycl::static_pipe<float, 4> p;
{
  // Launch the producer to stream A to the pipe
    q.submit([&](cl::sycl::execution_handle &cgh) {
      // Get write access to the pipe
      auto kp = p.get_access<cl::sycl::access::write>(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++)
            // Try to write to the pipe up to success
            while (!(kp.write(ka[i])))
              ;
        });
      });
```

```
  // Launch the consumer that adds the pipe stream with B to C
  q.submit([&](cl::sycl::execution_handle &cgh) {
    // Get read access to the pipe
    auto kp = p.get_access<cl::sycl::access::read>(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++) {
          /* Declare a variable of the same type as what the pipe
             can deal (a good example of single source advantage)
          */
          decltype(kp)::value_type e;
          // Try to read from the pipe up to success
          while (!(kp.read(e)))
            ;
          kc[i] = e + kb[i];
        }
      });
    });
} //< End scope for the queue and the buffers, so wait for completion
```

SYCL Nested Parallelism

Nested Parallelism

Nested parallelism is a form of parallelism where more work can be submitted to the device, with the scope of the submission to be the SYCL host command group.

All of the subsequent device command groups that are submitted from the parent command group kernel will be enqueued on a device_queue and executed asynchronously in relation to the parent kernel.

```
q.submit([&](cl::sycl::execution_handle& cgh) {
  auto anAcc = aBuffer.get_access<cl::sycl::access::mode::read_write>(cgh);
  auto anotherAcc =
      anotherBuffer.get_access<cl::sycl::access::mode::read_write>(cgh);

  cgh.single_task<class device_side_enqueue>(
      cl::sycl::range<1>(1), [=](cl::sycl::id<1> index) {
        anAcc[index]++;

        device_queue dq = q.get_default_queue();
        auto event = dq.submit(
            enqueue_policy::wait_kernel, [&](cl::sycl::device_handle& dh) {
              int error = dh.parallel_for(
                  range<1>(numElements2),
                  [=](cl::sycl::id<1> idx) { anotherAcc[idx]--; });
            });
      });
});
```

SYCL Hierarchical Parallelism

Hierarchical Parallelism

The hierarchical parallelism approach models the three levels of parallelism which are available in the OpenCL execution model through three nested parallel for function scopes.

- parallel_for_work_group : once per workgroup
  - parallel_for_sub_group : once per vector of work-items
    - parallel_for_work_item : once per work-item

```
auto flexible_command_group = [&](execution_handle& cgh) {
  cgh.parallel_for_work_group<class example_kernel>(range<3>(2, 2, 2),
                                                    [=](group<3> my_group) {
    parallel_for_sub_group(myGroup, [=](sub_group<3> my_sub_group){
      //[sub-group code]
      parallel_for_work_item(my_sub_group, [=](item<3> my_item) {
        //[work-item code]
      });
    }
    //[workgroup code]
});
```

SYCL Collective Operations

Collective Operations on group and sub_group classes

The group and sub_group classes encapsulate the functionality within a work-group or a sub-group level of parallel execution. Since OpenCL 2.0 collective functions are available
{all, any, broadcast, reduce, scan}

```
q.submit([&](cl::sycl::execution_handle& cgh) {
  auto acc = buffer.get_access<cl::sycl::access::mode::read_write>(cgh);
  cgh.single_task<class svm_sample1>(
      cl::sycl::nd_range<2>(range<2>(16, 16), range<2>(4, 4)),
      [=](cl::sycl::nd_item<1> idx) {
        group my_group = idx.get_group();
        int x = a_function();
        int sum = my_group<work_group_op::add> reduce(x);
        //...
      });
});
```

Shared Virtual Memory (SVM)

Shared Virtual Memory (SVM)

Shared virtual memory, a.k.a. SVM, enables using complex data classes and pointers between host and devices, using atomics and in some cases the lifetime scope of the buffer and accessor classes as synchronisation points. There are different flavours of shared virtual memory, depending on the capabilities of the OpenCL system to share device pointers with the host.

Shared Virtual Memory (SVM)

Coarse grained SVM

A buffer in a context can be allocated as coarse-grained buffer on a device and have the same underlying raw pointer on both host and device. Accessors are needed in this case to ensure data consistency.

```
cl::sycl::capability_selector selectDevice(
    cl::sycl::exec_capabilities::svm_coarse_grain);

cl::sycl::context coarseContext(selectDevice);
cl::sycl::svm_allocator<int> coarseSVMAllocator(coarseContext);

/* Allocate SVM buffer using coarse grained buffer sharing virtual memory
 * address space.
 */
cl::sycl::buffer<
    int, 1,
    cl::sycl::svm_allocator<int, cl::sycl::exec_capabilities::svm_coarse_grain>>
    svmBuffer(cl::sycl::range<1>(numElements), coarseSVMAllocator);
```

Shared Virtual Memory (SVM)

Coarse grained SVM

A buffer in a context can be allocated as coarse-grained buffer on a device and have the same underlying raw pointer on both host and device. Accessors are needed in this case to ensure data consistency.

```
{
  /* Access the SVM buffer on host */
  auto hostSvmCoarsePointer =
      svmBuffer.get_access<cl::sycl::access::mode::write,
                           cl::sycl::access::target::host_buffer>();
  /* Within this block it is safe to use the raw SVM pointer. */
  auto rawSvmCoarsePointer = hostSvmCoarsePointer.get();
  *rawSvmCoarsePointer = 100;
}  // The underlying SVM pointer gets updated and no access on host is
   // possible when the host accessor goes out of scope
```

Shared Virtual Memory (SVM)

Coarse grained SVM

A buffer in a context can be allocated as coarse-grained buffer on a device and have the same underlying raw pointer on both host and device. Accessors are needed in this case to ensure data consistency.

```
q.submit([&](cl::sycl::execution_handle<svm_coarse_grain>& cgh) {
  /* Make the SVM pointer allocation available for updating on the device */
  auto deviceCoarsePointer =
      svmBuffer.get_access<cl::sycl::access::mode::read_write>(cgh);
  cgh.single_task<class svm_sample1>(
      cl::sycl::range<1>(1), [=](cl::sycl::id<1> index) {
        int* rawSvmCoarsePointer = deviceCoarsePointer.get();
        rawSvmCoarsePointer[index] = a + 1;
      });
});
```

Shared Virtual Memory (SVM)

Fine grained buffer sharing SVM

Shared virtual memory with buffer sharing support denotes the capability of being able to share memory allocations between host and device, at the granularity of each allocation. Depending on the availability of atomics loads and stores can be atomic.

```
cl::sycl::svm_allocator<int, cl::sycl::svm_fine_grain> fineGrainAllocator(
    fineGrainedContext);
/* Use the instance of the svm_allocator with any custom container, for
 * example, with std::shared_ptr.
 */
std::shared_ptr<int> svmFineBuffer =
    std::allocate_shared<int>(fineGrainAllocator, numElements);
/* Initialize the pointer on the host */
*svmFineBuffer = 66;

q.submit([&](
    cl::sycl::execution_handler<svm_fine_grain<
        cl::sycl::svm_sharing::buffer, cl::sycl::svm_atomics::none>>& cgh) {
  auto rawSvmFinePointer = svmFineBuffer.get();
  /* Register access on the device in the specific command group with
   * access::mode so that dependency tracking can be possible and the
   * kernel can use the raw pointer. Custom containers are not working
   * on the device side unless they are structs provided by the user
   * and compiled for the device.
   */
  cgh.register_access<cl::sycl::access::mode::read_write>(rawSvmFinePointer);
  cgh.single_task<class svm_fine_grained_kernel>(
      range<1>(1), [=](id<1> index) { rawSvmFinePointer[index]++; });
});
```

Shared Virtual Memory (SVM)

Fine grained system sharing SVM

System sharing virtual memory provides the capability to share the entire host's virtual memory with the device.
In that case, any pointers allocated on the host with the standard C++ API, can be used on the device without any additional synchronisation APIs.

```
cl::sycl::context systemSharingContext(systemSharingSelector);
/* Use the fine grained buffer sharing selector to choose a device that
 * supports this SVM mode and create a queue using the context instance given.*/
cl::sycl::queue q(systemSharingSelector, systemSharingContext);

std::allocator<int> defaultAllocator;  // Default STL allocator

std::shared_ptr<int> svmSystemPtr =
    std::allocate_shared<int>(defaultAllocator, 1);

/* Initialize the pointer on the host */
*svmSystemPtr = 66;

cl::sycl::handler_event he =
    q.submit([&](cl::sycl::execution_handler<
                  svm_fine_grain<svm_sharing::system, svm_atomics::none>>& cgh) {
      cgh.single_task<class svm_system_sharing_kernel>(cl::sycl::range<1>(1),
                                                       [=](cl::sycl::id<1> index) {
                                                         /* In fine grained system
                                                          * sharing the pointer allocated
                                                          * on host can be used
                                                          * on the device. */
                                                         svmSystemPtr[index];
                                                       });
    });

cl::sycl::event complete = he.get_complete();  // get completion event
complete.wait();                                // waits for completion
```

SYCL 2.2 New Features

- Command group versioning and extendability
- Pipes
- Nested Parallelism
- Hierarchical Parallelism with sub_group level
- SYCL collective operations
- Shared virtual memory

Questions?

- SYCL spec and forums:

  http://www.khronos.org/opencl/sycl

- Codeplay's blogs:

  http://www.codeplay.com/portal/