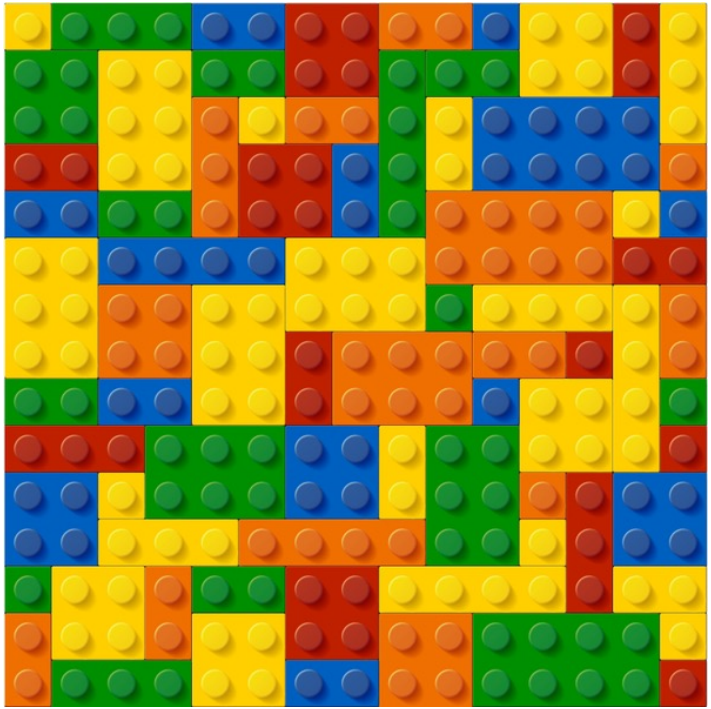
SYCL Building Blocks for C++ libraries

Maria Rovatsou, SYCL specification editor
Principal Software Engineer, Codeplay Software

SYCL Building Blocks for C++ libraries

- Overview
- SYCL standard
- What is the SYCL toolkit for?
- C++ libraries using SYCL: Under construction
- SYCL Building blocks
  - asynchronous execution on multiple SYCL devices
  - command groups for embedding computational kernels in a library
  - buffers and accessors for data storage and access
  - vector classes shared between host and device
  - pipes
  - hierarchical command group execution
  - printing on device using cl::sycl::stream



- SYCL standard

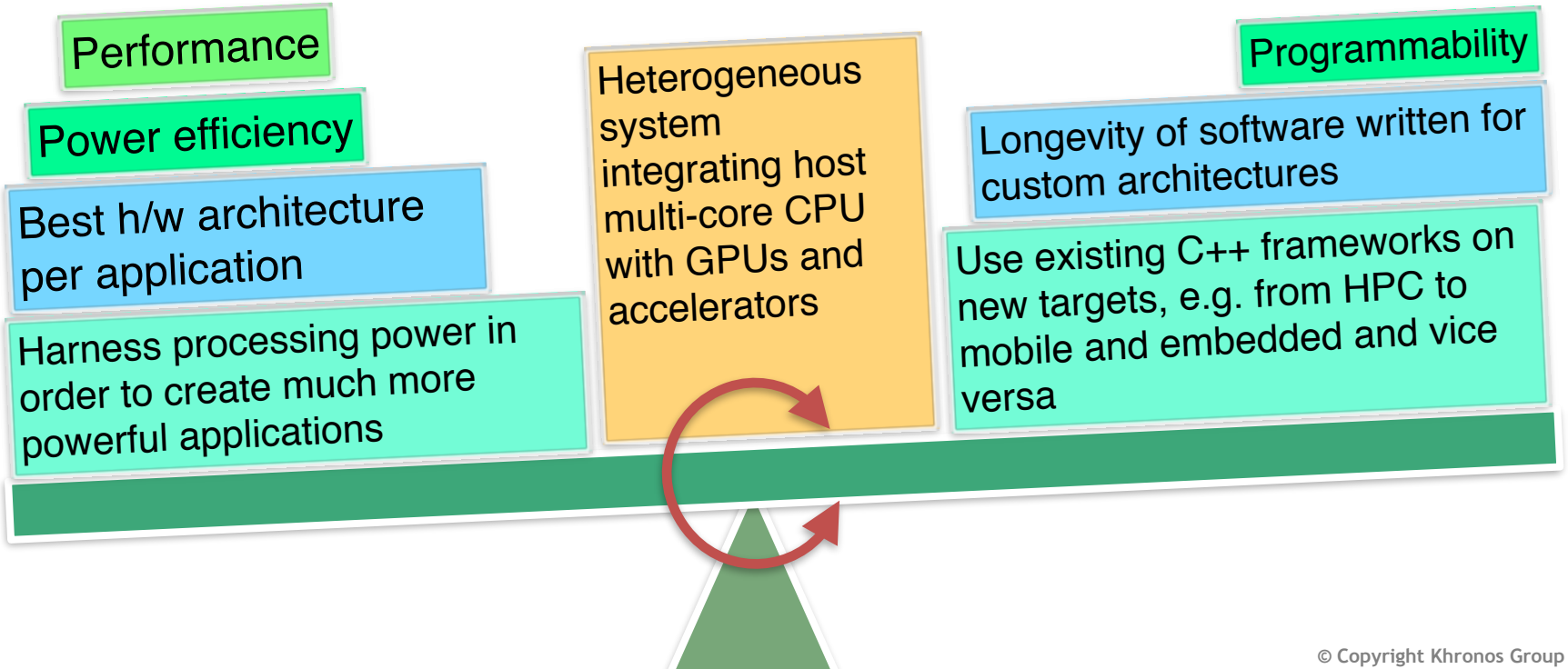
Khronos royalty-free open standard that works with OpenCL™

- SYCL™ for OpenCL™ 1.2 is a published specification that can be found at: https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf
- SYCL™ for OpenCL™ 2.2 is a published provisional specification that the group would like feedback on: https://www.khronos.org/registry/sycl/specs/sycl-2.2.pdf

KHRONOS GROUP

- What is the SYCL toolkit for?

What is the SYCL toolkit for?



- C++ libraries using SYCL building blocks: Under construction

C++ libraries using SYCL building blocks: VisionCpp

VisionCpp is a templated library with a domain specific C++ dialect focused on providing a framework for efficient Machine Vision algorithms

```
auto q = get_queue<gpu_selector>();
cv::VideoCapture cap(0);
cap >> frame;
std::shared_ptr<unsigned char> ptr;
auto a = Node<sBGR, rows,cols,Image>(frame.data);
auto b = Node<sBGR2sRGB>(a);
auto c = Node<sRGB2lRGB>(b);
auto d = Node<lRGB2lHSV>(c);
auto e = Node<DegradeH>(d);
auto f = Node<lHSV2lRGB>(e);
auto g = Node<lRGB2sRGB>(f);
auto h = Node<sRGB2sBGR>(g);
for (;;) {
auto fused_kernel = execute<Fuse>(h, q);
ptr = fused_kernel.getData();
cv::imshow("Colour Degragation", ooo);
if (cv::waitKey(30) >= 0)
break;
cap >> frame; // get a new frame from camera
}
```

C++ libraries using SYCL building blocks: Eigen Tensor Library

Eigen is a C++ templated library for linear algebra.
The Tensor module SYCL support is in progress.

```
// use gpu
cl::sycl::gpu_selector s;
cl::sycl::queue q(s);
Eigen::SyclDevice sycl_device(q);
// create a tensor range
Eigen::array<int, 3> tensorRange = {{100, 100, 100}};
// create tensors
Eigen::Tensor<float, 3> in1(tensorRange);
Eigen::Tensor<float, 3> in2(tensorRange);
Eigen::Tensor<float, 3> in3(tensorRange);
Eigen::Tensor<float, 3> out(tensorRange);
// create random data for each tensor
in1 = in1.random();
in2 = in2.random();
in3 = in3.random();
// create a map veiw for each tensor
Eigen::TensorMap<Eigen::Tensor<float, 3> > gpu_in1(in1.data(), tensorRange);
Eigen::TensorMap<Eigen::Tensor<float, 3> > gpu_in2(in2.data(), tensorRange);
Eigen::TensorMap<Eigen::Tensor<float, 3> > gpu_in3(in3.data(), tensorRange);
Eigen::TensorMap<Eigen::Tensor<float, 3> > gpu_out(out.data(), tensorRange);
// assign operator invokes executor
gpu_out.device(sycl_device) = gpu_in1 * gpu_in2 + gpu_in3;
```

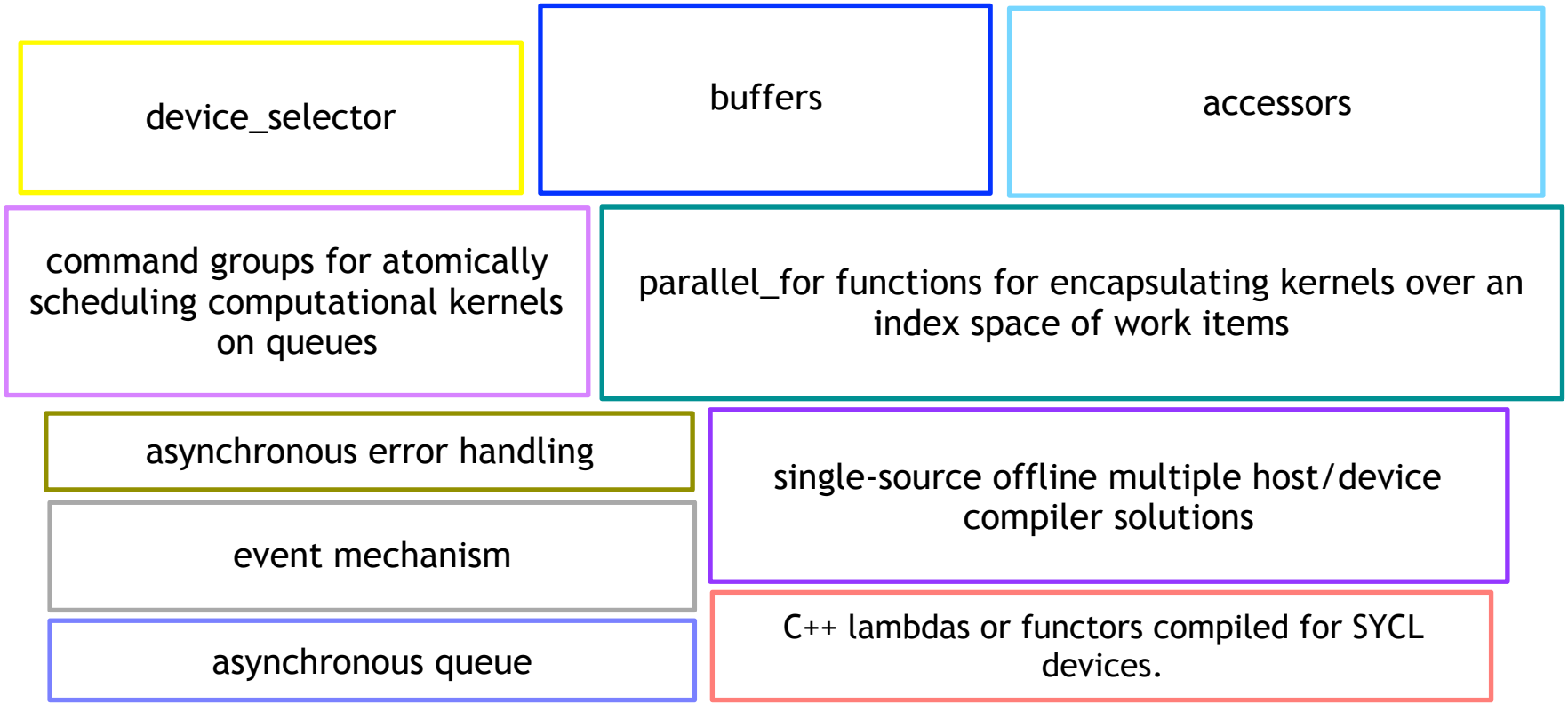
C++ libraries using SYCL building blocks: Parallel STL

Parallel STL is the next version of the C++ standard library which is starting to add support for parallel algorithms.

```
1 vector <int > data = { 8, 9, 1, 4 };
2 std :: sort ( sycl_policy , v. begin () , v. end ());
3 if ( is_sorted ( data )) {
4 cout << " Data is sorted ! " << endl ;
5 }
```

SYCL building blocks

SYCL Building Blocks: asynchronous execution on multiple SYCL devices



SYCL Building Blocks: command groups for embedding computational kernels in a library

command group lambda or functor

The data movement is managed by buffers and accessors.

```
1  auto myCommandGroup = ([&](cl::sycl::execution_handler& cgh) {
2    auto acc = buffer.get_access<cl::sycl::access::mode::read_write>(cgh);
3    cgh.single_task<class lambda_or_functor_kernel>(
4        cl::sycl::nd_range<2>(range<2>(16, 16), range<2>(4, 4)),
5        [=](cl::sycl::nd_item<1> idx) {
6          group my_group = idx.get_group();
7          int x = a_function();
8          int sum = my_group.reduce<work_group_op::add>(x);
9          //..
10       });
11 });
```

On submit the kernel and its access requirements are scheduled for execution on an asynchronous queue.

```
1  queue myQueue;
2  auto myEvent = myQueue.submit(myCommandGroup);
```

SYCL Building Blocks: buffers and accessors for data storage and access

Data is managed across host and device using the buffer class.

```
1 buffer<pos_t, 1> positionsBuf(positions.data(),
2                               range<1>(number_objects));
3 buffer<vel_t, 1> velocitiesBuf(velocities.data(),
4                                range<1>(number_objects));
```

The buffer on destruction waits for all usage of its data by the SYCL runtime to be completed.

The buffer class takes ownership of the data.

SYCL Building Blocks: accessors for data storage and access

Accessor class gives access to a buffer in a command group

The SYCL runtime is effectively constructing an asynchronous runtime graph that schedules all the command groups and their memory transfers.

```
1  auto cg = [&](handler &ch) {
2    auto posAcc = positionsBuf.get_access<access::mode::read>(ch);
3    auto velAcc = velocitiesBuf.get_access<access::mode::read_write>(ch);
4    auto newPosAcc = newposBuf.get_access<access::mode::read_write>(ch);
5    ch.parallel_for<class nbody_kernel>(
6        nd_range<1>(range<1>(number_objects), range<1>(256)),
7        ([=](nd_item<1> item_) {
8          unsigned int i = item_.get_global_linear_id();
9          // Computation and update of velocities[ ... ]
10         // Compute the new positions
11         newPosAcc[i][0] = posAcc[i][0] + dtCube * velAcc[i][0] + 0.5f a[0];
12         newPosAcc[i][1] = posAcc[i][1] + dtCube * velAcc[i][1] + 0.5f * a[1];
13         newPosAcc[i][2] = posAcc[i][2] + dtCube * velAcc[i][2] + 0.5f * a[2];
14       }));
15 };
```

Host accessors force synchronisation of all copies of the buffer on devices with the host.

SYCL Building Blocks: accessors for data storage and access

svm_allocator provides allocation capabilities for SYCL 2.2 devices with fine grained SVM capabilities.

```
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();
  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]++; });
});
```

Registered usage of the SVM allocated pointer.

```
int result = *svmFineBuffer;
```

Direct Usage of SVM allocated fine grained buffer pointer on the host.

SYCL Building Blocks: SIMD vector classes available on host and device

The vec<class T, int S> class can be used on host and device.

```
for (int i = 0; i < 64; i++) {
  dataA[i] = float4(2.0f, 1.0f, 1.0f, static_cast<float>(i));
  dataB[i] = float3(0.0f, 0.0f, 0.0f);
}
```

```
myQueue.submit([&](handler& cgh) {
  auto ptrA = bufA.get_access<access::mode::read_write>(cgh);
  auto ptrB = bufB.get_access<access::mode::read_write>(cgh);

  cgh.parallel_for<class vector_example>(range<3>(4, 4, 4), [=](item<3> item) {
    float4 in = ptrA[item.get_linear_id()];
    float w = in.w();
    float3 swizzle = in.xyz();
    float3 trans = swizzle * w;
    ptrB[item.get_linear_id()] = trans;
  });
});
```

Swizzles are supported as functions, as well as all the common operators for vectors.

SYCL Building Blocks: pipes

```
1  cl::sycl::static_pipe<float, 4> p;
2
3  // Launch the producer to stream A to the pipe
4  q.submit([&](cl::sycl::execution_handle &cgh) {
5    // Get write access to the pipe
6    auto kp = p.get_access<cl::sycl::access::write>(cgh);
7    // Get read access to the data
8    auto ka = ba.get_access<cl::sycl::access::read>(cgh);
9
10   cgh.single_task<class producer_using_pipes>([=] {
11     for (int i = 0; i != N; i++)
12       // Try to write to the pipe up to success
13       while (!(kp.write(ka[i])));
14   });
15 });
```

Pipes are available in SYCL 2.2 for OpenCL 2.2 devices.

Pipes are a memory object with a sequential ordering of data that cannot be accessed from the host.

The static_pipe is a pipe with constexpr capacity that is defined only for one target device. This is the type of pipe which can be useful for compile-time graphs and pipelines.

SYCL Building Blocks: hierarchical command group kernels

parallel_for_work_group() : in this scope the code is executing once per work group and the memory is allocated in the local address space.

```
auto command_group = [&](handler & cgh) {
  // Issue 8 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
      range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {
        int myLocal;
        private_memory<int> myPrivate(myGroup);
        parallel_for_work_item(myGroup, [=](item<3> myItem) {
          //[work-item code]
          myPrivate(myItem) = 0;
        });
        parallel_for_work_item(myGroup, [=](item<3> myItem) {
          //[work-item code]
          output[myGroup.get_local_range() * myGroup.get() + myItem] =
              myPrivate(myItem);
        });
        //[workgroup code]
      });
});
```

In a parallel_for_work_item scope the code is executed once per work-item and the memory is allocated in private memory.

In SYCL 2.2, there is a parallel_for_sub_group which executes once per vector of work items.

SYCL Building Blocks: basic stream class

The stream object is provided to give basic support for printing from the device for debugging or notification reasons.

```
1 myQueue.submit([&](handler& cgh) {
2   /* We create a stream object to print from the device. */
3   stream os(1024, 80, cgh);
4   cgh.single_task<class hello_world>([=]() {
5     /* We use the stream operator on the stream object we created above to
6        * print to stdout from the device. */
7     os << "hello World!" << endl;
8   });
9 });
```

SYCL Building Blocks for C++ libraries

Questions?

- SYCL Building blocks
  - asynchronous execution on multiple SYCL devices
  - command groups for embedding computational kernels in a library
  - buffers and accessors for data storage and access
  - vector classes shared between host and device
  - pipes
  - hierarchical command group execution
  - printing on device using cl::sycl::stream

- SYCL spec and forums:
  www.khronos.org/opencl/sycl

- New website coming!
  sycl.tech