SYCL™ for OpenCL™
in a nutshell

Maria Rovatsou, Codeplay's R&D Product Development Lead & Contributor to SYCL

IWOCL Conference May 2014

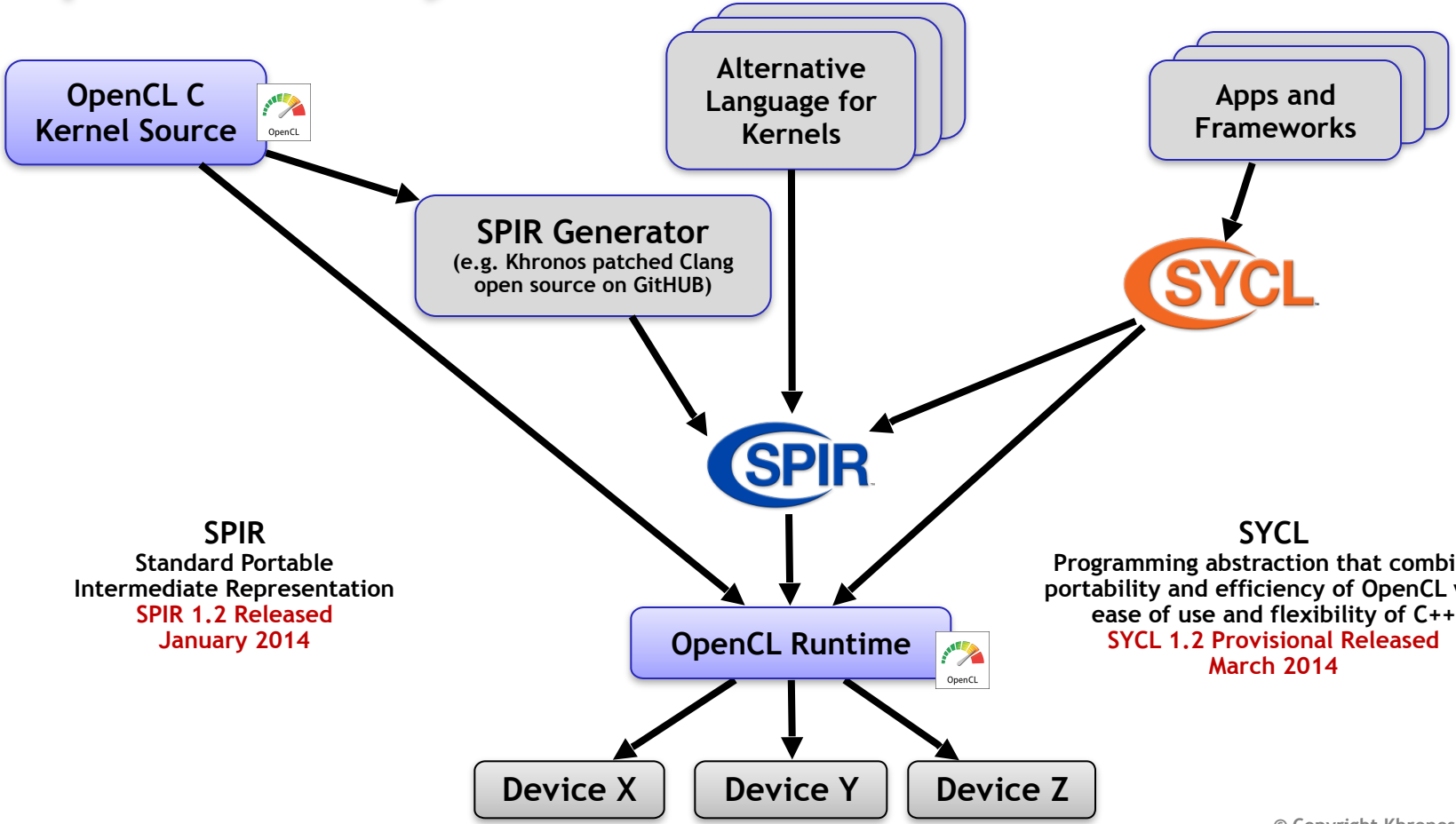
SYCL for OpenCL

in a nutshell

- SYCL in the OpenCL ecosystem
- SYCL aims
- Simple example
- Advanced features overview
- Roadmap for SYCL

KHRONOS GROUP

OpenCL Ecosystem



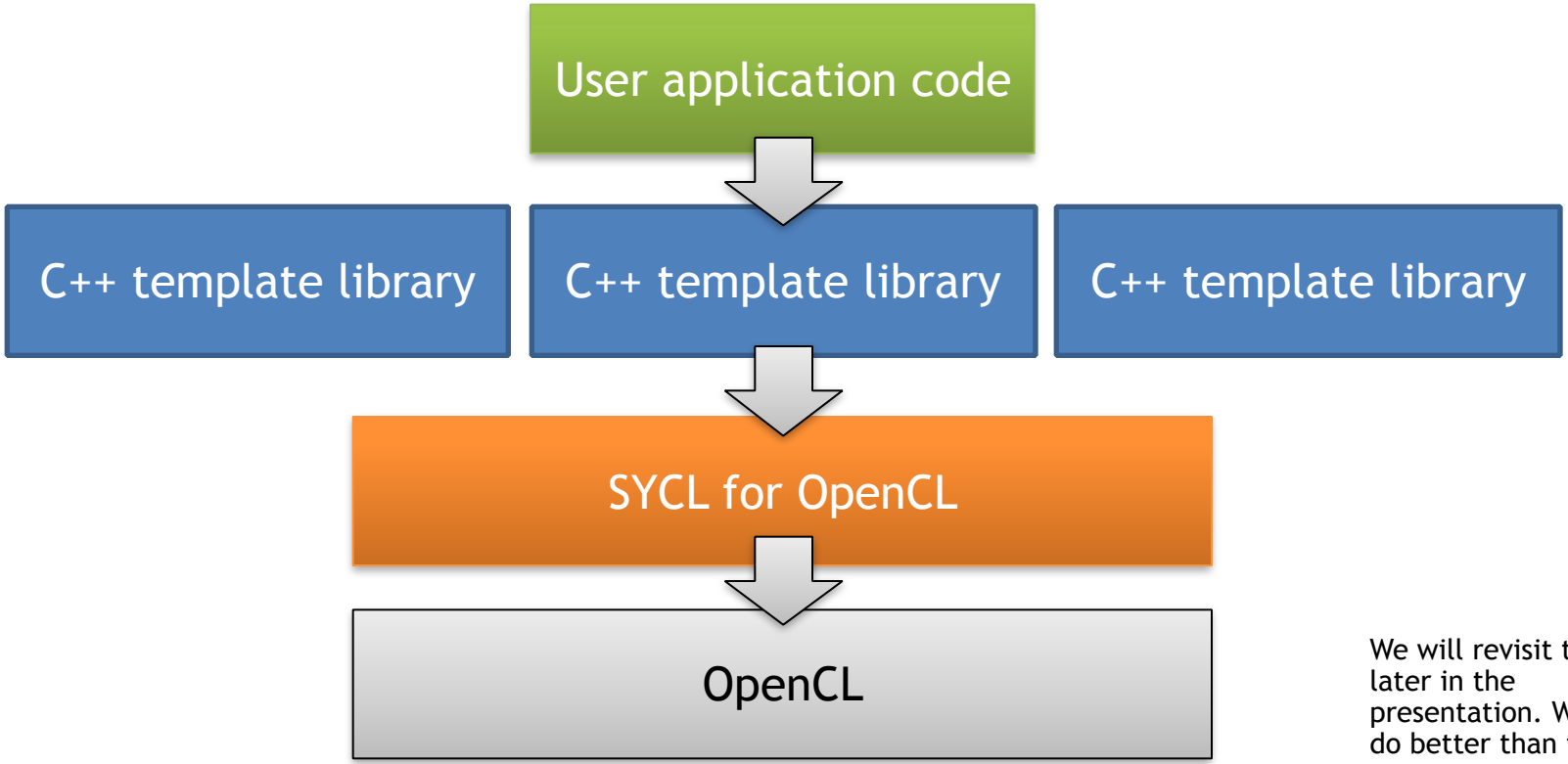
SPIR
Standard Portable Intermediate Representation
SPIR 1.2 Released
January 2014

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

- We want to enable a C++ for OpenCL ecosystem
  - Where more C++ developers can get the benefits of OpenCL
  - With C++ libraries supported on OpenCL platforms
  - C++ tools supported on OpenCL platforms
  - Aim to achieve long-term support for OpenCL features with C++
  - Multiple sources of implementations
  - Reliability by providing host fall-back
  - Enable future innovation

The layering of SYCL: Building an ecosystem



We will revisit this later in the presentation. We can do better than this

SYCL aims for

- Ease of use
  - Single source programming model
  - SYCL source compiled for host and device(s)
  - Development/debugging on host
  - Programming interface that data management and error handling
- C++ features available for OpenCL
- Portability across platforms and compilers
- Providing the full OpenCL feature set and seamless integration with existing OpenCL code
- High performance
- Enabling the creation of higher level programming models and C++ templated libraries based on OpenCL

SYCL Simple Example:

Simple Vector Addition

SYCL Simple Example

Pick your favorite platform and C++ IDE with SYCL support and start:

```
1: #include <CL/sycl.hpp>
2:
3: int main ()
4: {
5:     return 0;
6: }
```

SYCL Simple Example

Pick your favorite platform and C++ IDE with SYCL support and start:

```
1: #include <CL/sycl.hpp>
2: using namespace cl::sycl;
3: int main ()
4: {
5:     return 0;
6: }
```

SYCL Simple Example

Declare and initialize SYCL Buffers:

```
1: #include <CL/sycl.hpp>
2: #include <vector>
3: using namespace cl::sycl;
4:
5: int main()
6: {
7:  int count = 1024; //Number of elements
8:  std::vector<float> a(1024);
9:  std::vector<float> b(1024);
10:  std::vector<float> c(1024);
11: /* Initialization of data */
```

SYCL Simple Example

Declare and initialize SYCL Buffers:

```
16: /** Adding code in main() **/
17: // SYCL buffer that serves as an abstraction to
18: // OpenCL buffers
19: {
20:   buffer<float,1> a_buf(a,count);
21:   buffer<float,1> b_buf(b,count);
22:   buffer<float,1> c_buf(b,count);
23: }
```

synchronization using RAII

SYCL image support
SYCL vec support

KHRONOS GROUP

SYCL Simple Example

Create a SYCL queue for the *default* OpenCL device on your platform:

```
16: /** Adding code in main() **/
17: // SYCL buffer that serves as an abstraction
18: // to OpenCL buffers
19: {
20:   buffer<float,1> a_buf(a,count);
21:   buffer<float,1> b_buf(b,count);
22:   buffer<float,1> c_buf(b,count);
23:  {
24:    queue myQueue;
25:  }
26: }
```

host can serve as a platform and fall-back mechanism

SYCL Simple Example

Alternatively use a **device_selector** that you can customize the selection:

customizable device_selector class

```
16: /** Adding code in main() **/
17: // SYCL buffer that serves as an abstraction
18: // to OpenCL buffers
19: {
20:   buffer<float,1> a_buf(a,count);
21:   buffer<float,1> b_buf(b,count);
22:   buffer<float,1> c_buf(b,count);
23:  {
24:    gpu_selector selection;
25:    queue myQueue(&selection);
26:  }
27: }
```

SYCL context and device available

SYCL Simple Example

Start executing code on the device by encapsulating it in a *command_group*:

```
/** Adding code in main() **/
23:  {
24:    queue myQueue;
25:    command_group(&myQueue,[&]
26:    {
27:      auto a_dev = a_buf.get_access<access::read>();
28:      auto b_dev = b_buf.get_access<access::read>();
29:      auto c_dev = c_buf.get_access<access::write>();
30:
31:      parallel_for(nd_range<1>(range<1>(count)),
32:                   kernel_functor<class vector_add>([=](item item)
33:      {
34:        int index = item.get_global(0);
35:        c_dev[index] = a_dev[index]+b_dev[index];
36:      }));
37:    });
```

atomic operation that encapsulates: memory object creation, copying, mapping, and execution of kernels

asynchronous execution

SYCL Simple Example

Access the data of the buffer on the device by using the *accessor* class:

```
/** Adding code in main() **/
23:  {
24:    queue myQueue;
25:    command_group(&myQueue,[&]
26:    {
27:      auto a_dev = a_buf.get_access<access::read>();
28:      auto b_dev = b_buf.get_access<access::read>();
29:      auto c_dev = c_buf.get_access<access::write>();
30:
31:      parallel_for(nd_range<1>(range<1>(count)),
32:                   kernel_functor<class vector_add>([=](item item)
33:      {
34:        int index = item.get_global(0);
35:        c_dev[index] = a_dev[index]+b_dev[index];
36:      }));
37:    });
```

All OpenCL data types and access modes

SYCL Simple Example

Parallel_for for SYCL and the kernel as a lambda function:

```
/** Adding code in main() **/
23:  {
24:    queue myQueue;
25:    command_group(&myQueue,[&]
26:    {
27:      auto a_dev = a_buf.get_access<access::read>();
28:      auto b_dev = b_buf.get_access<access::read>();
29:      auto c_dev = c_buf.get_access<access::write>();
30:
31:      parallel_for(nd_range<1>(range<1>(count)),
32:                   kernel_functor<class vector_add>([=](item item)
33:      {
34:        int index = item.get_global(0);
35:        c_dev[index] = a_dev[index]+b_dev[index];
36:      }));
37:    });
```

functor, lambda or OpenCL C string

SYCL Simple Example

Parallel_for for SYCL and the kernel as a functor:

```
5:  class vector_add
6:  {
7:  public:
8:
9:   accessor<float,1,access::read,access::global_buffer> a_dev;
10:  accessor<float,1,access::read,access::global_buffer> b_dev;
11:  accessor<float,1,access::write,access::global_buffer> c_dev;
12:
13:  vector_add(accessor<float,1,access::read,access::global_buffer> a,
14:           accessor<float,1,access::read,access::global_buffer> b,
15:           accessor<float,1,access::write,access::global_buffer>c)
16:           :a_dev(a),b_dev(b),c_dev(c){};
17:
18:  void operator()(item item)
19:  {
20:    int index = item.get_global_id(0);
21:    c_dev[index] = a_dev[index] + b_dev[index];
22:  }
23: };
```

SYCL Simple Example

Parallel_for for SYCL and the kernel as a functor:

```
/** Adding code in main() **/
23:  {
24:    queue myQueue;
25:    command_group(&myQueue,[&]
26:    {
27:        accessor<float,1,access::read,access::global_buffer> a_dev(a_buf);
28:        accessor<float,1,access::read,access::global_buffer> b_dev(b_buf);
29:        accessor<float,1,access::write,access::global_buffer> c_dev(c_buf);
30:
31:        parallel_for(nd_range<1>(range<1>(count)),
32:                     kernel_functor(vector_add(a_dev,b_dev,c_dev)));
33:    });
34:  }
```

SYCL Simple Example

Templated parallel_for depending to the data type:

```
5: template<typename DATATYPE>
6: class vector_add
7: {
8: public:
9:
10:  buffer<DATATYPE, 1> * p_a_buf;
11:  buffer<DATATYPE, 1> * p_b_buf;
12:  buffer<DATATYPE, 1> * p_c_buf;
13:  unsigned int count;
14:
15:  vector_add(buffer<DATATYPE, 1> * p_a,
16:     buffer<DATATYPE, 1> * p_b,
17:     buffer<DATATYPE, 1> * p_c,
18:     unsigned int count
19:     ) :p_a_buf(p_a), p_b_buf(p_b), p_c_buf(p_c), count(count) { };
20:  void operator()()
21:  {
21:    auto a_dev = p_a_buf->get_access<access::read>();
22:    auto b_dev = p_b_buf->get_access<access::read>();
23:    auto c_dev = p_c_buf->get_access<access::write)();
24:
25:    parallel_for(nd_range<1>(range<1>(count)),
26:                 kernel_functor<class vector_addition>([=](item item){
27:      int index = item.get_global_id(0);
28:      c_dev[index] = a_dev[index] + b_dev[index];
29:    }));
30: }
31:};
```

SYCL Simple Example

Templated parallel_for depending to the data type:

```
/** Adding code in main() **/
42: {
43:   buffer<float,1> a_buf(a,count);
44:   buffer<float,1> b_buf(b,count);
45:   buffer<float,1> c_buf(c,count);
46:   {
47:     queue myQueue;
48:     command_group(&myQueue,vector_add<float,1>(&a_buf,&b_buf,&c_buf,count));
49:   }
50: }
```

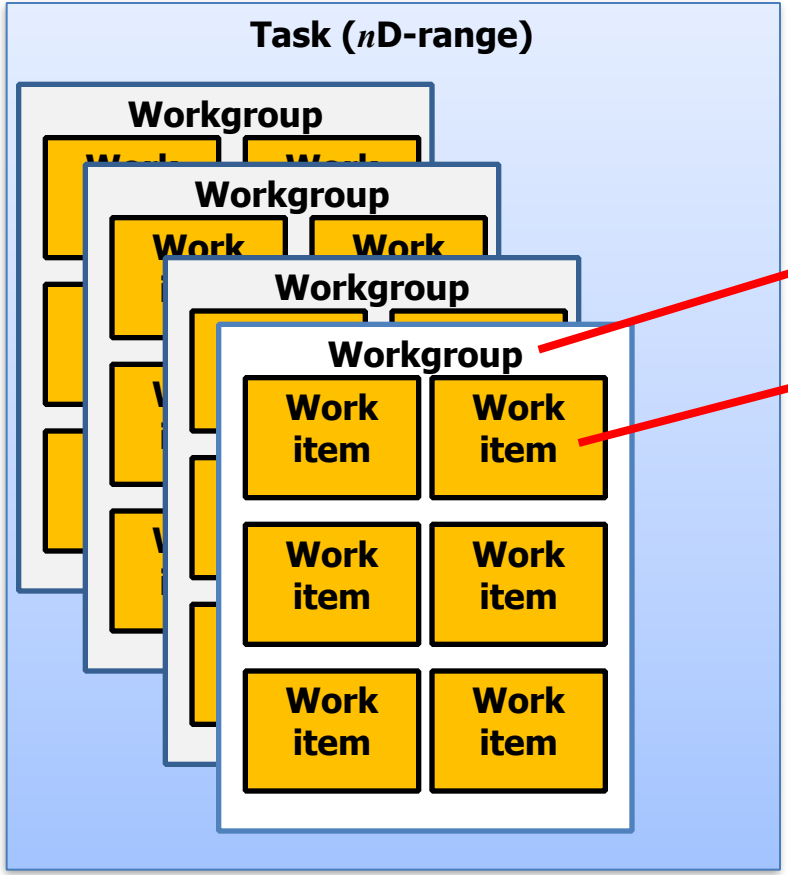
Advanced SYCL features:

Overview

Advanced SYCL features

- More features…
  - Host “fall-back” mode
  - C++ exception handling
  - OpenCL data types and built-in functions available
  - Event handling through *event* class
- Hierarchical Data Parallelism
  - Programming interface introduced in SYCL

Hierarchical Data Parallelism




```
buffer<int> my_buffer(data, 10);

auto in_access = my_buffer.get_access<cl::sycl::access:read>();
auto out_access = my_buffer.access<cl::sycl::access:write>();

command_group(my_queue, [&]()
{
   parallel_for_workgroup(nd_range(range(size), range(groupsize)),
      lambda<class hierarchical>([=](group group)
   {
      parallel_for_workitem(group, [=](item tile)
      {
         out_access[tile] = in_access[tile] * 2;
      });
   }));
});
```

Advantages:

1. Easy to understand the concept of work-groups
2. Performance-portable between CPU and GPU
3. No need to think about barriers (automatically deduced)
4. Easier to compose components & algorithms

Advanced SYCL features

- OpenCL/SYCL interoperability
  - Seamless integration of OpenCL C applications with SYCL applications
- SYCL/OpenGL interoperability
  - Based on OpenCL/OpenGL interoperability extensions

SYCL Roadmap

- GDC, March 2014
  - Released a provisional specification to enable feedback
  - Developers can provide input into standardization process
  - Feedback via Khronos forums
- Next steps
  - Full specification, based on feedback
  - Khronos test suite for implementations
  - Release of implementations

SYCL Useful Links

- SYCL spec and forums:

  http://www.khronos.org/opencl/sycl
- Codeplay's blogs:

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

Questions?

KHRONOS GROUP