codeplay

Implementing Khronos SYCL for OpenCL

Ralph Potter, Research Engineer
University of Bath and Codeplay Software

Visit us at
www.codeplay.com

45 York Place
Edinburgh
EH1 3HP
United Kingdom

Overview

- Motivation
- What is SYCL?
- Where does it fit with OpenCL?
- How do I use it?

Motivation

- To make GPGPU simpler and more accessible.
- To create a C++ for OpenCL™ ecosystem.
  - Combine the ease of use and flexibility of C++ and the portability and efficiency of OpenCL.
- To provide a foundation for constructing complex and reusable template algorithms:
  - parallel_reduce(), parallel_map(), parallel_sort()
- To define an open and portable standard.

What is SYCL?

- Provisional Khronos group specification.
- Cross platform, shared source, C++ programming layer.
- Built on top of OpenCL 1.2 and based on standard C++11.

SYCL Roadmap

- Supercomputing, November 2014
  - Released a second provisional specification to enable feedback.
  - Developers can provide input into standardisation process.
  - Feedback via Khronos forums.
- Next Steps
  - Full specification, based on feedback.
  - Khronos test suite for implementations.
  - Release of implementations.

OpenCL

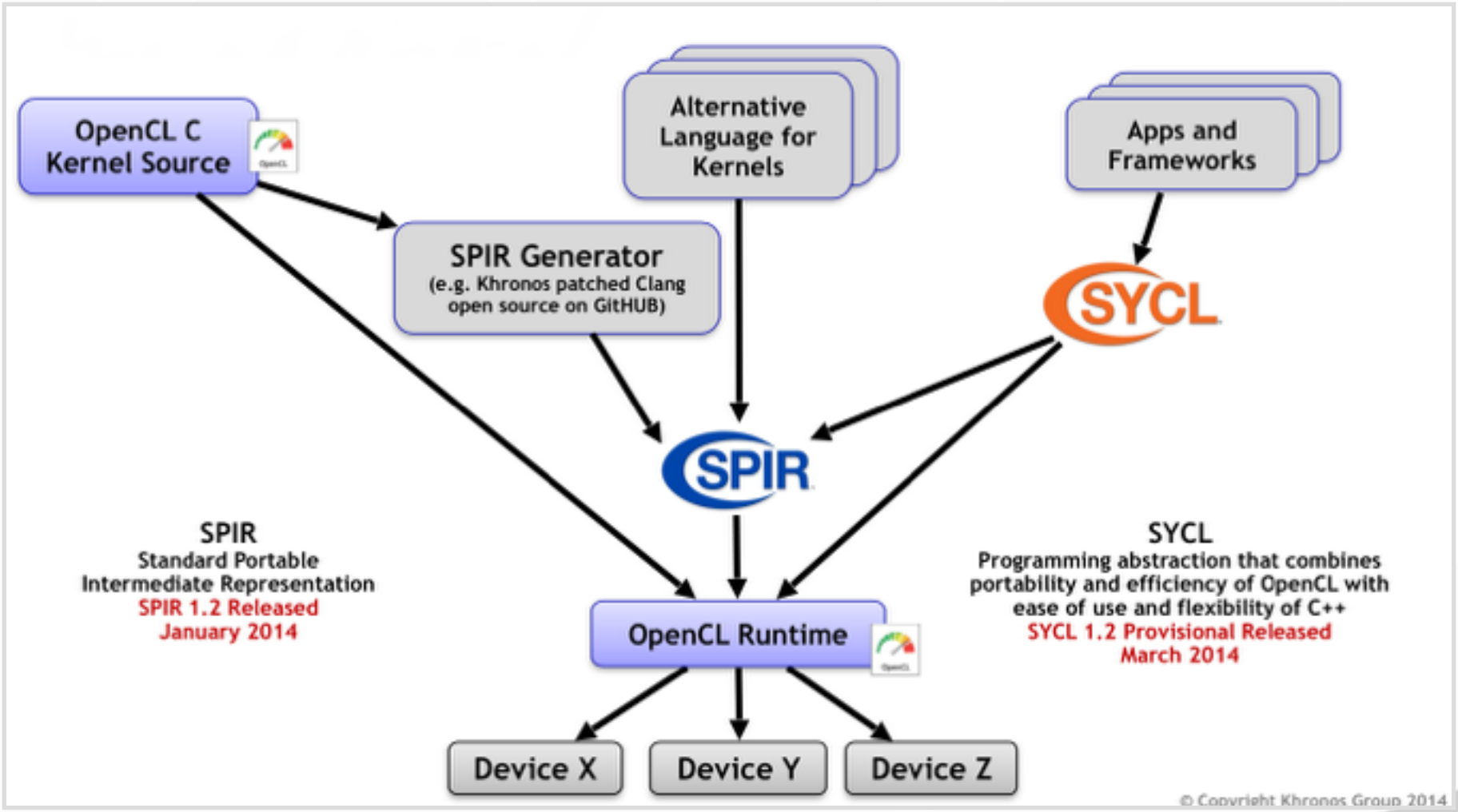
Host
(Sequential)

Device
(Parallel)

```
clEnqueueWriteBuffer(...);
clSetKernelArg(...);
clEnqueueNDRange(..., vectorAdd);


clEnqueueReadBuffer(...);
```

```
__kernel void vectorAdd(__global float* out,
                        __global float* inA,
                        __global float* inB) {
  int i = get_global_id(0);
  out[i] = inA[i] + inB[i];
}
```

OpenCL Ecosystem

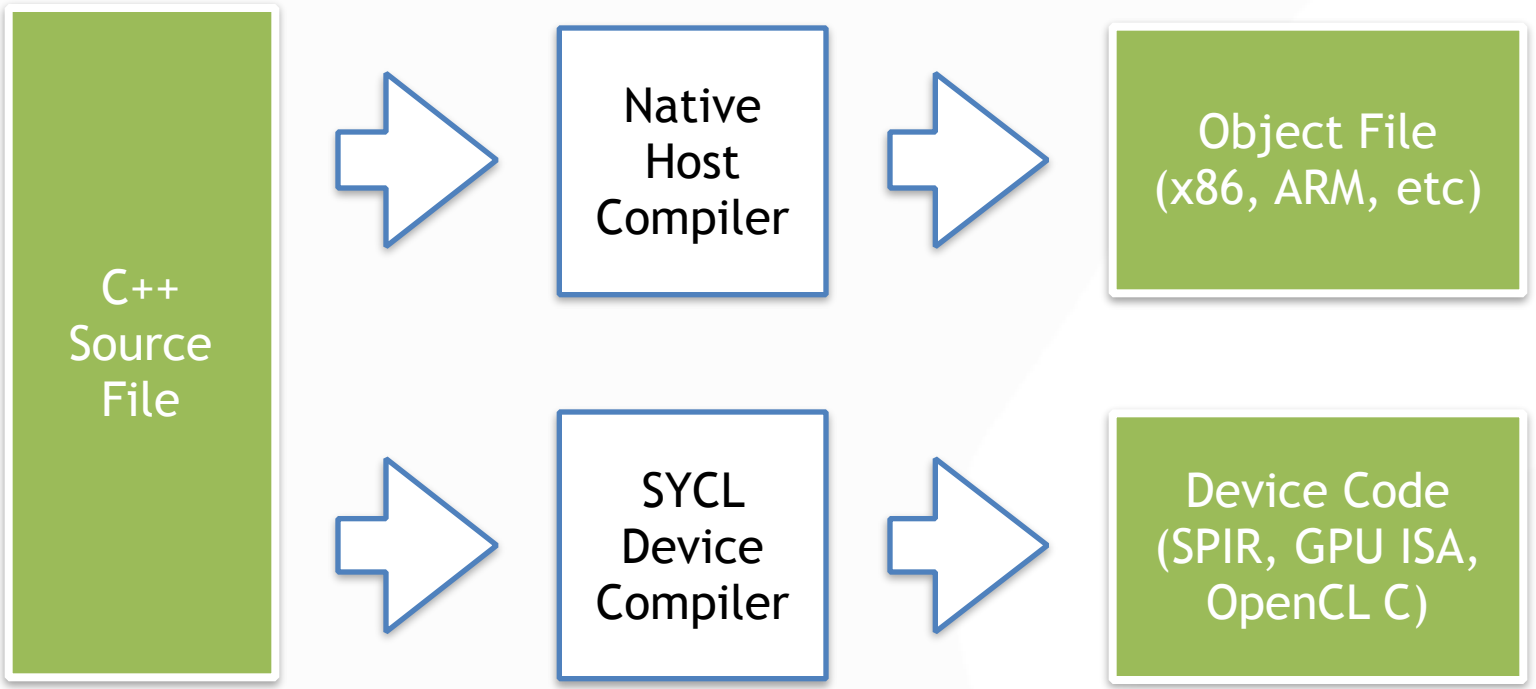


SYCL

- Write your entire program in valid C++11, without having to add additional non-standard keywords to your source.
- Compiler separates accelerator kernels, and generates code to execute them on device.
- Runtime resolves data dependencies and schedules host-device data movement.

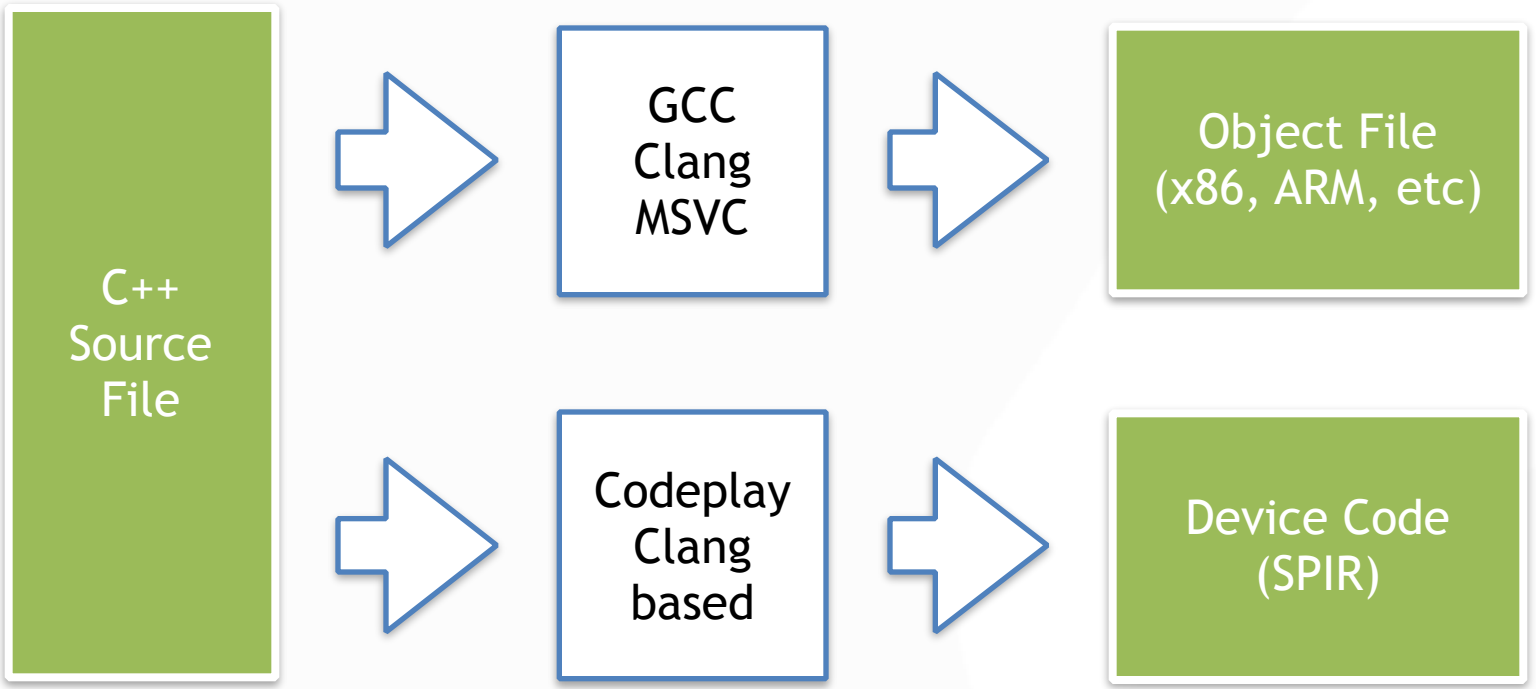
Shared Source Compilation

- Write your C++ code once.
- Use it on both host and device.



Shared Source Compilation

- Write your C++ code once.
- Use it on both host and device.



Example

```
using namespace cl::sycl;

// Create a queue, on the default device.
queue myQueue;

// Create a buffer, wrapping the inArray pointer.
buffer<float, 1> myBuffer(inArray, count);

// Enqueue kernel. Accessor creation also enqueues buffer.
command_group(myQueue, [&]
{
  auto acc = myBuffer.get_access<read_write>();
  parallel_for<class MyKernel>(range<1>(count),
    [=] (id index)
    {
      acc[index] = 42.0f;
    });
});

// Results returned to host on buffer/queue destruction.
```

Example

```
using namespace cl::sycl;

// Create a queue, on the default device.
queue myQueue;

// Create a buffer, wrapping the inArray pointer.
buffer<float, 1> myBuffer(inArray, count);

// Enqueue kernel. Accessor creation also enqueues buffer.
command_group(myQueue, [&]
{
  auto acc = myBuffer.get_access<read_write>();
  parallel_for<class MyKernel>(range<1>(count),
    [=] (id index)
    {
      acc[index] = 42.0f;
    });
});

// Results returned to host on buffer/queue destruction.
```

Kernels

```
float f = 42.0f
auto acc = myBuffer.get_access<read_write>();
cl::sycl::parallel_for<class X>({1024, 0, 0},
  [=](id<1> item){
    // Kernel k
    acc[item] =
  }
```

Gives a uni
Required because lambda functions are anonymous types.

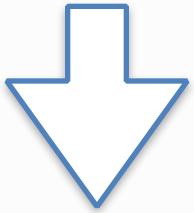
ND-range for grid and workgroup sizes.

The lambda function is converted into an OpenCL kernel.
Variables captured by the lambda are copied automatically.

- function objects
- Transformed into OpenCL kernels
- Entire call graph of the kernel is transformed

Kernel Compilation

```
parallel_for<class vadd>(range<1>(count), [=](id<1> itemID)
{
  outPtr[itemID] = inPtrA[itemID] + inPtrB[itemID];
});
```



```
__kernel sycl_vadd(__global float* outPtr,
                   __global float* inPtrA,
                   __global float* inPtrB,
                   id itemID)
{
  outPtr[itemID] = inPtrA[itemID] + inPtrB[itemID];
}
```

Example: Reduction

```
template <typename N, typename L>
void parallel_for(range<1>, L);
```

```
__kernel void sycl_kernel0(__global float
                           *ptr);
```

```
template <typename T>
T reduce_array(T *data);
```

```
float reduce_array(__global float *data);
```

```
template <typename T>
T reduce(T a, T b);
```

```
float reduce(float a, float b);
```

Example: Lambda Capture

```
// Declare a floating point value on the host.
float scale = 99.0f;

// …
auto acc = myBuffer.get_access<read_write>();
parallel_for<class MyKernel>(range<1>(count), [=] (id index)
{
  acc[index] *= scale;
});
```

```
__kernel sycl_MyKernel(__global float* acc, float scale,
                       id itemID)
{
  acc[itemID] = acc[itemID] * scale;
}
```

Example: Lambda Capture

```
// Declare a large matrix on the host.
matrix<12,12> mat;

// …
auto acc = myBuffer.get_access<read_write>();
parallel_for<class MyKernel>(range<1>(count), [=] (id index)
{
  acc[index] = apply(mat, acc[index]);
});
```

```
__kernel sycl_MyKernel(__global float* acc, matrix_12_12 mat,
                       id itemID)
{
  acc[itemID] = apply(mat, acc[index]);
}
```

Example

```
using namespace cl::sycl;

// Create a queue, on the default device.
queue myQueue;

// Create a buffer, wrapping the inArray pointer.
buffer<float, 1> myBuffer(inArray, count);

// Enqueue kernel. Accessor creation also enqueues buffer.
command_group(myQueue, [&]
{
  auto acc = myBuffer.get_access<read_write>();
  parallel_for<class MyKernel>(range<1>(count),
    [=] (id index)
    {
      acc[index] = 42.0f;
    });
});

// Results returned to host on buffer/queue destruction.
```

Queues

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

int main(int argc, char** argv) {
  cl::sycl::default_selector selector;
  cl::sycl::queue queue(selector);

  return 0;
}
```

- Selectors choose an OpenCL device
- Queues are used to enqueue work
- Host-fallback where no suitable OpenCL device available

Buffers and Accessors

```
float array[1024];
cl::sycl::buffer<1> myBuffer(array, range<1>(1024));
```

- SYCL buffers wrap a pointer to data on host.
- Modifying data wrapped by a buffer is undefined while a buffer is live.
  - Except via accessors.
- Buffer destructors block until dependent kernels complete.

Buffers and Accessors

```
cl::sycl::command_group(myQueue, [&]
{
  // Create accessors at command group scope.
  auto accessor = myBuffer.get_access<read_write>();

  // Create kernel here.
});
```

- Accessors bind a buffer to a command group, and therefore to an OpenCL device.
- They enable SYCL to track data dependencies and correctly order kernels.

Command Groups

```
command_group(myQueue, [&]
{
  auto acc = myBuffer.get_access<read_write>();
  parallel_for<class MyKernel>(range<1>(count),
    [=] (id index)
    {
      acc[index] = 42.0f;
    });
});
```

- Encapsulate enqueuing a kernel and its data dependencies (accessors) to a device.
- Command groups execute asynchronously and potentially out-of-order.
- Synchronisation to host via buffer/queue destructors, host accessors or events.

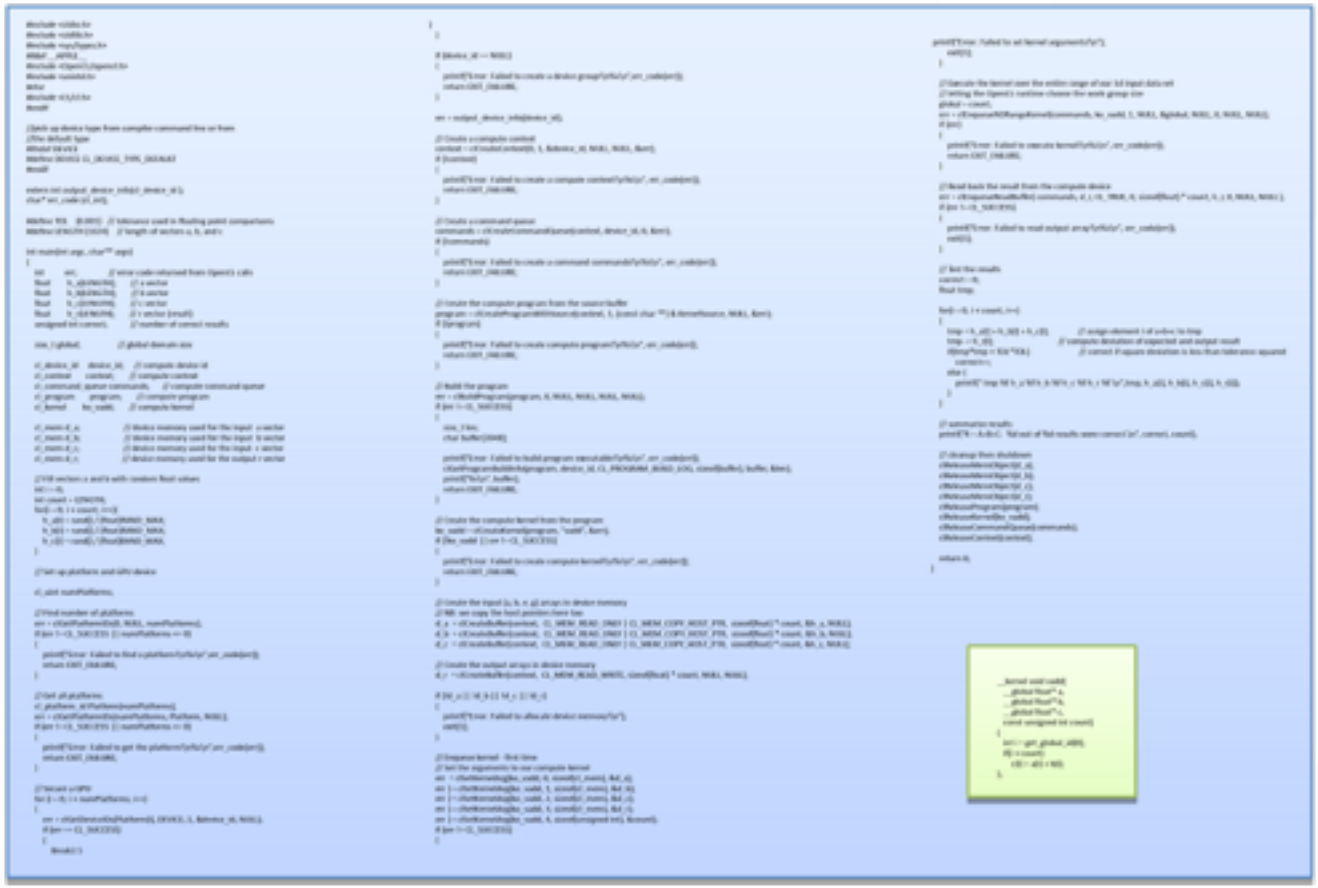
Example

```
using namespace cl::sycl;

// Create a queue, on the default device.
queue myQueue;

// Create a buffer, wrapping the inArray pointer.
buffer<float, 1> myBuffer(inArray, count);

// Enqueue kernel. Accessor creation also enqueues buffer.
command_group(myQueue, [&]
{
  auto acc = myBuffer.get_access<read_write>();
  parallel_for<class MyKernel>(range<1>(count),
    [=] (id index)
    {
      acc[index] = 42.0f;
    });
});

// Results returned to host on buffer/queue destruction.
```

codeplay

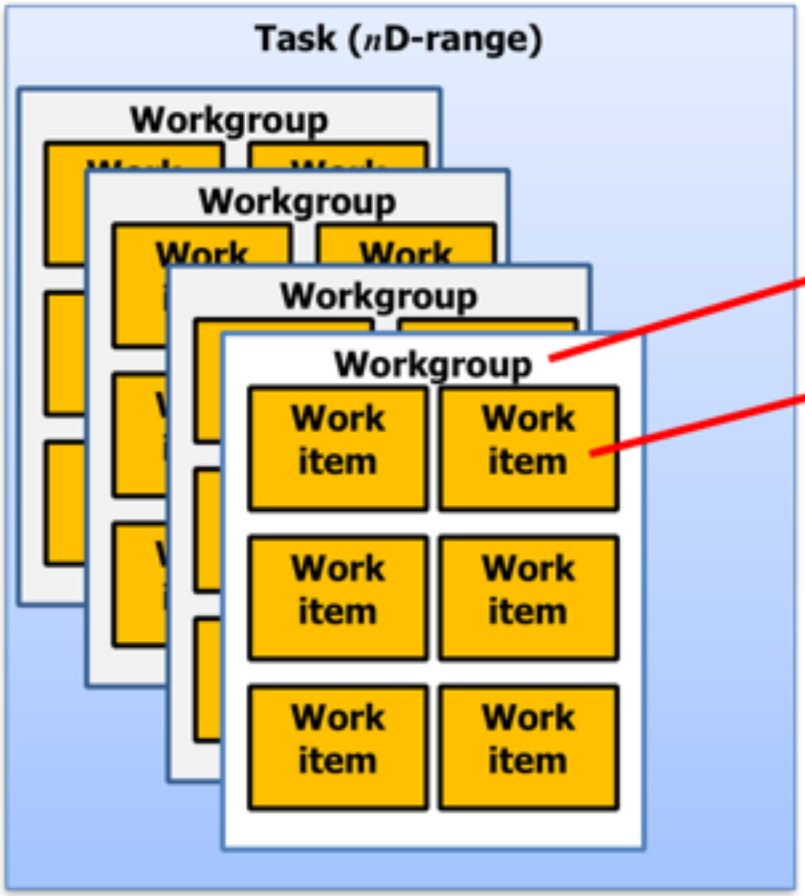
Comparison with OpenCL

OpenCL



SYCL

Hierarchical 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

Interop with OpenCL

- All OpenCL 1.2 core functionality available.
- Can enqueue OpenCL C kernels via parallel_for.
- Can also retrieve a cl_kernel object from a SYCL kernel and enqueue via clEnqueueNDRangeKernel
- Outside SYCL data dependency scheduling. Need to manage lifetimes explicitly.

Limitations

- OpenCL 1.2 hardware platform
- No function pointers, recursion, virtual functions, global variables within kernels

Resources

- Specification and forum
  - https://www.khronos.org/opencl/sycl
- Tutorials
  - http://codeplay.com/portal/blogs/
- Implementations
  - https://github.com/amd/triSYCL
  - Or contact Codeplay

Questions?