The 11th International workshop on OpenCL and SYCL

IWOCL & SYCLCON 2023

Towards Deferred Execution of a SYCL Command Graph

Ewan Crawford, Codeplay

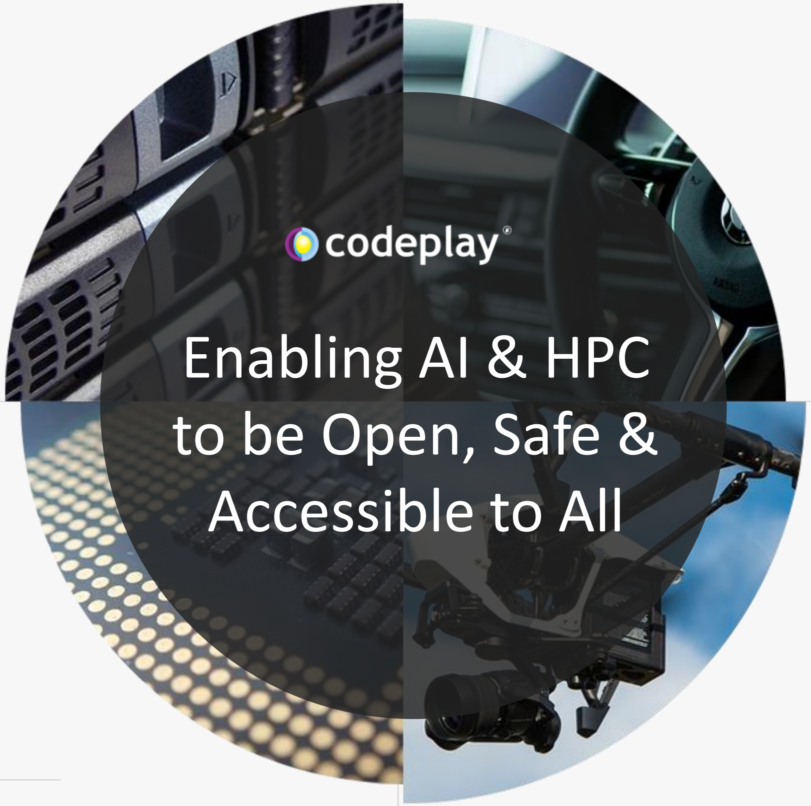
Pablo Reble (Intel), Ben Tracy (Codeplay), and Julian Miller (Intel)

Company

Leaders in enabling high-performance software solutions for new AI processing systems

Enabling the toughest processors with tools and middleware based on open standards

Established 2002 in Scotland, acquired by Intel in 2022 and now ~90 employees.



Collaborations

Synopsys, Broadcom, CEVA, Imagination, Renesas, KMC Kyoto Microcomputer Co., Ltd., NSI-TEXE, Berkeley Lab, Oak Ridge National Laboratory, Argonne National Laboratory

And many more!

Supported Solutions

oneAPI

An open, cross-industry, SYCL based, unified, multiarchitecture, multi-vendor programming model that delivers a common developer experience across accelerator architectures

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

Markets

High Performance Compute (HPC)
Automotive ADAS, IoT, Cloud Compute
Smartphones & Tablets
Medical & Industrial

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

Who we are

- After years of collaboration and contribution to open standards alongside **Intel**, **Codeplay Software** is a subsidiary of **Intel** after an acquisition made last year.

- We will continue to operate as Codeplay Software and will work extensively with **all relevant industries** to **advance the SYCL ecosystem**, especially around **oneAPI.**

- Codeplay is now working jointly with Intel to further advance the **SYCL standard** and the **oneAPI** open ecosystem.

Talk Agenda

oneAPI

- Motivation
- Specification overview
- Implementation details
- Future steps

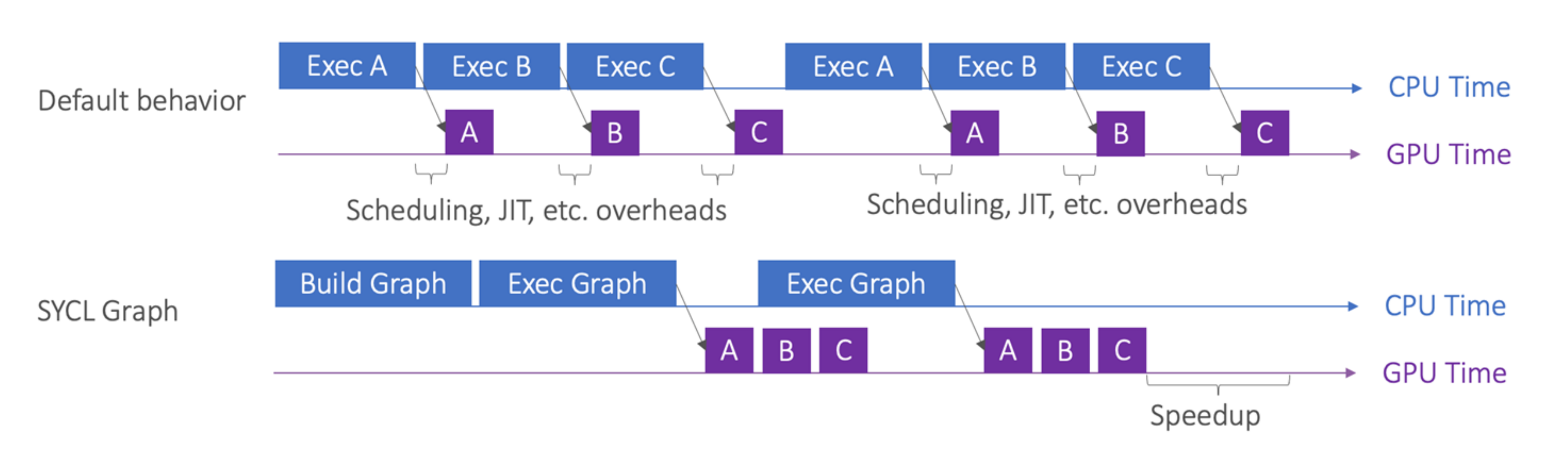
Motivation

- SYCL is already able to define a DAG of execution at runtime.
- Graph is implicit in the code with command creation and submission are tied together.
- Our extension provides a way to give the user control of the dependency graph in a construction step prior to execution.

Benefits of Separating Concerns

- A graph can be defined once and submitted as many times as required.

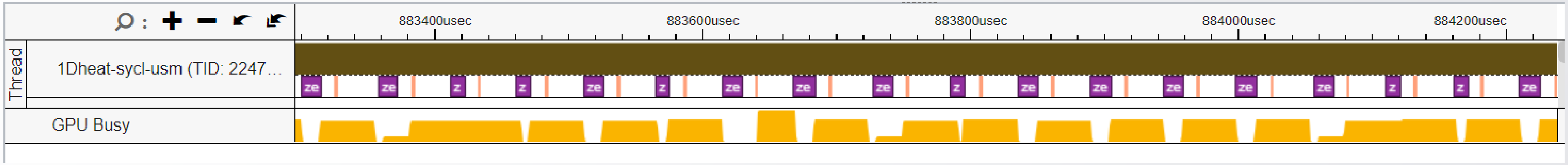
- Reduces latency when submitting commands to the device.



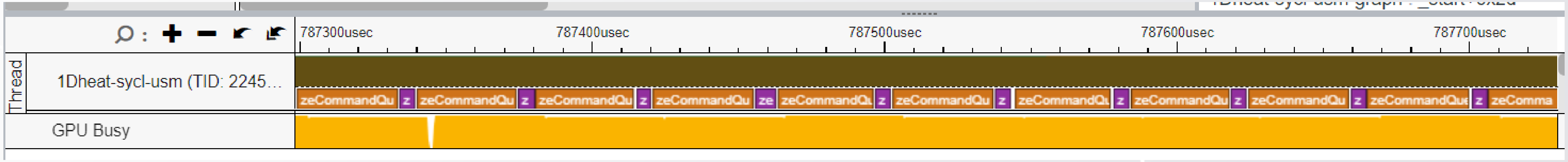
- Optimizations become available across the defined graph.

1Dheat example on GPU comparison

Default SYCL



Modified for SYCL Graph extension



https://github.com/reble/oneAPI-samples/tree/sycl-graph/DirectProgramming/DPC%2B%2B/StructuredGrids/1d_HeatTransfer

Intel® Core™ i7-6770HQ Processor with Intel® Iris® Pro Graphics 580

Related Work

- Splitting command construction from execution is a proven solution.

- Lower-level APIs:
  - Vulkan command-buffer
  - OpenCL cl_khr_command_buffer extension (see IWOCL 2022 talk)
  - Level Zero command-list

- CUDA-Graphs is an analogous feature in CUDA.

Project Goals

1. Extension that integrates well into the SYCL standard.

2. Improve performance by explicit reuse of resources for specific workloads – small kernels with repetitive execution.

3. Support frameworks that can currently target CUDA Graphs:
   - Tensorflow
   - PyTorch
   - GROMACS
   - Kokkos

Extension Specification

sycl_ext_oneapi_graph

- Open development on GitHub https://github.com/reble/llvm
- Spec PR https://github.com/intel/llvm/pull/5626 of first revision
- Experimental extension
  - APIs presented in this talk are subject to change, so any feedback you have is helpful.
  - Additions in ext::oneapi::experimental namespace

Strongly typed graph object

- Strong typing makes the state of the graph clear to the reader.
- Consistent with SYCL kernel bundle design.
- Tied to a single device and context.

```
// State of a graph
enum class graph_state {
  modifiable,
  executable
};

// New object representing graph
template<graph_state State = graph_state::modifiable>
class command_graph {};

template<>
class command_graph<graph_state::modifiable> {
public:
  command_graph(const context& syclContext, const device& syclDevice,
                const property_list& propList = {});

  command_graph<graph_state::executable>
  finalize(const property_list& propList = {}) const;

  // other methods
};

template<>
class command_graph<graph_state::executable> {
public:
  command_graph() = delete;

  // other methods
};
```

State Transition

oneAPI

Modifiable

- Graph is under construction and new nodes may be added to it.

Finalize

- Single point of overheads from optimization and construction of backend representation.
- Many executable state graphs can be created from a single modifiable state graph.

Executable

- Graph topology fixed and is ready for execution.
- Submitted for execution as many times as desired.

Executable Graph Submission

- handler::depends_on can express graph submission dependencies.
- Subgraphs expressed naturally.

```
// New methods added to the sycl::queue class
using namespace ext::oneapi::experimental;
class queue {
public:
  /* -- graph convenience shortcuts -- */

  event ext_oneapi_graph(command_graph<graph_state::executable>& graph);
  event ext_oneapi_graph(command_graph<graph_state::executable>& graph,
                         event depEvent);
  event ext_oneapi_graph(command_graph<graph_state::executable>& graph,
                         const std::vector<event>& depEvents);
};

// New methods added to the sycl::handler class
class handler {
public:
  void ext_oneapi_graph(command_graph<graph_state::executable>& graph);
}
```

Graph Construction Mechanisms

Queue Recording API (Record & Replay)

- Capture command-groups submitted to a queue and recorded them in a graph.

Attributes:

- Easier to use when targeting an existing code base.
- External library calls can be captured to a graph.

Explicit Graph Building API

- User has direct access to graph building interface that adds nodes and edges.

Attributes:

- Working with node objects directly is more expressive.
- Easier to debug and less likely to trigger invalid usage.

Adding Nodes & Edges

oneAPI

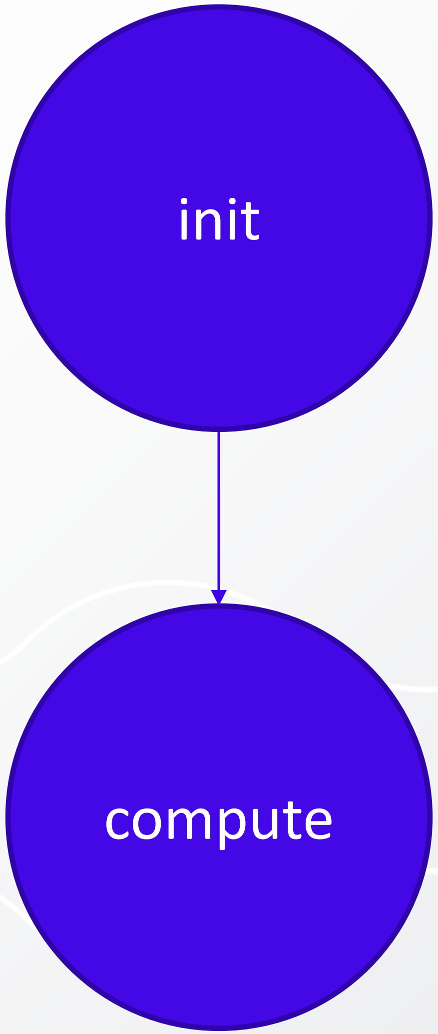
Nodes

- A command-group submission to a queue being recorded by queue recording API.
- A command-group submission to explicit API method for adding nodes.

Edges

- Dependencies defined by sycl::buffer accessors.
- Using handler::depends_on() with an event returned by a queue recording submission.
- Two mechanisms in explicit API:
  - Passing a list of dependent nodes on node creation.
  - make_edge() method

SYCL SAXPY




```
sycl::queue q{sycl::gpu_selector_v};

const size_t n = 1000;
const float a = 3.0f;
float *x = sycl::malloc_device<float>(n, q);
float *y = sycl::malloc_shared<float>(n, q);

auto initEvent = q.submit([&](sycl::handler &h) {
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    x[i] = 1.0f;
    y[i] = 2.0f;
  });
});

auto computeEvent = q.submit([&](sycl::handler &h) {
  h.depends_on(initEvent);
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    y[i] = a * x[i] + y[i];
  });
});

computeEvent.wait();
```

oneAPI

Record and Replay

```
template<>
class command_graph<graph_state::modifiable> {
public:
  // ...

  bool begin_recording(queue& recordingQueue,
                       const property_list& propList = {});
  bool begin_recording(const std::vector<queue>& recordingQueues,
                       const property_list& propList = {});

  bool end_recording();
  bool end_recording(queue& recordingQueue);
  bool end_recording(const std::vector<queue>& recordingQueues);

  // ...
};
```

```
sycl::queue q{sycl::gpu_selector_v};
sycl::ext::oneapi::experimental::command_graph g(q.get_context(), q.get_device());

const size_t n = 1000;
const float a = 3.0f;
float *x = sycl::malloc_device<float>(n, q);
float *y = sycl::malloc_shared<float>(n, q);

g.begin_recording(q);
auto initEvent = q.submit([&](sycl::handler &h) {
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    x[i] = 1.0f;
    y[i] = 2.0f;
  });
});

auto computeEvent = q.submit([&](sycl::handler &h) {
  h.depends_on(initEvent);
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    y[i] = a * x[i] + y[i];
  });
});
g.end_recording(q);

auto executable_graph = g.finalize();
q.submit([&](sycl::handler &h) { h.ext_oneapi_graph(executable_graph); }).wait();
```

Explicit API

```
template<>
class command_graph<graph_state::modifiable> {
public:
  // ...

  node add(const property_list& propList = {});

  template<typename T>
  node add(T cgf, const property_list& propList = {});

  void make_edge(node& src, node& dest);

  // ...
};
```

```
sycl::queue q{sycl::gpu_selector_v};
sycl::ext::oneapi::experimental::command_graph g(q.get_context(), q.get_device());

const size_t n = 1000;
const float a = 3.0f;
float *x = sycl::malloc_device<float>(n, q);
float *y = sycl::malloc_shared<float>(n, q);

auto init = g.add([&](sycl::handler &h) {
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    x[i] = 1.0f;
    y[i] = 2.0f;
  });
});

auto compute = g.add([&](sycl::handler &h) {
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    y[i] = a * x[i] + y[i];
  });
}, {sycl::ext::oneapi::experimental::property::node::depends_on(init)});

auto executable_graph = g.finalize();
q.submit([&](sycl::handler &h) { h.ext_oneapi_graph(executable_graph); }).wait();
```

Explicit API

```
template<>
class command_graph<graph_state::modifiable> {
public:
  // ...

  node add(const property_list& propList = {});

  template<typename T>
  node add(T cgf, const property_list& propList = {});

  void make_edge(node& src, node& dest);

  // ...
};
```

```
sycl::queue q{sycl::gpu_selector_v};
sycl::ext::oneapi::experimental::command_graph g(q.get_context(), q.get_device());

const size_t n = 1000;
const float a = 3.0f;
float *x = sycl::malloc_device<float>(n, q);
float *y = sycl::malloc_shared<float>(n, q);

auto init = g.add([&](sycl::handler &h) {
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    x[i] = 1.0f;
    y[i] = 2.0f;
  });
});

auto compute = g.add([&](sycl::handler &h) {
  h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> idx) {
    size_t i = idx;
    y[i] = a * x[i] + y[i];
  });
});

g.make_edge(init, compute);

auto executable_graph = g.finalize();
q.submit([&](sycl::handler &h) { h.ext_oneapi_graph(executable_graph); }).wait();
```

Implementation Status

oneDNN Example

Implementation today supports:

- Kernel command nodes
- USM
- Level Zero backend
- Both graph construction APIs

Enables oneDNN sycl_interop_usm sample to run using extension with shown changes.

```
-- a/examples/sycl_interop_usm.cpp
+++ b/examples/sycl_interop_usm.cpp
@@ -27,6 +27,8 @@
 #error "Unsupported compiler"
 #endif

+#include <sycl/ext/oneapi/experimental/graph.hpp>
+
 #include <cassert>
 #include <iostream>
 #include <numeric>
@@ -55,6 +57,9 @@ void sycl_usm_tutorial(engine::kind engine_kind) {
             mem_d, eng, sycl_interop::memory_kind::usm, usm_buffer);

     queue q = sycl_interop::get_queue(strm);
+    ext::oneapi::experimental::command_graph g {
+            q.get_context(), q.get_device()};
+    g.begin_recording(q);
     auto fill_e = q.submit([&](handler &cgh) {
         cgh.parallel_for<kernel_tag>(range<1>(N), [=](id<1> i) {
             int idx = (int)i[0];
@@ -70,6 +75,10 @@ void sycl_usm_tutorial(engine::kind engine_kind) {
             relu, strm, {{DNNL_ARG_SRC, mem}, {DNNL_ARG_DST, mem}}, {fill_e});
     relu_e.wait();

+    g.end_recording();
+    auto execGraph = g.finalize();
+    q.ext_oneapi_graph(execGraph);
+
     for (size_t i = 0; i < N; i++) {
         float exp_value = (i % 2) ? 0.0f : i;
         if (usm_buffer[i] != (float)exp_value)
```

https://gist.github.com/Bensuo/5c5eedd703dac8868a01720b0201988f

PI command-buffer

- DPC++ has an intermediate C abstraction API called “PI” that is implemented by SYCL-2020 backends.

- We’ve extended this interface to add a new command-buffer type and entry-points.
  - An extension similar to cl_khr_command_buffer additions to OpenCL.

- Provide an emulation mode to support sycl_ext_oneapi_graph on backends we’ve not yet implemented our PI extension for.

PI Additions

API Addition	Description
pi_ext_command_buffer	New type representing a command-buffer.
piextCommandBufferCreate()	Creates a command-buffer with optional properties.
piextCommandBufferFinalize()	No more commands can be added to command-buffer, and command-buffer is made ready to execute.
piextCommandBufferNDRangeKernel()	Add a kernel command to the command-buffer.
piextEnqueueCommandBuffer()	Submits a command-buffer for execution to a queue.
piextCommandBufferRetain()	Increments reference count.
piextCommandBufferRelease()	Decrements reference count.

PI Backend Mapping

PI API Addition	Intel Level Zero[1]	
	OpenCL cl_khr_command_buffer Extension[2]	CUDA Graphs[2]
pi_ext_command_buffer	ze_command_list_handle_t	
	cl_command_buffer_khr	cudaGraph_t
piextCommandBufferCreate	zeCommandListCreate	
	clCreateCommandBufferKHR	cudaGraphCreate
piextCommandBufferFinalize	zeCommandListClose	
	clFinalizeCommandBufferKHR	cudaGraphInstantiate
piextCommandBufferNDRangeKernel	zeCommandListAppendLaunchKernel	
	clCommandNDRangeKernelKHR	cudaGraphAddKernelNode
piextEnqueueCommandBuffer	zeCommandQueueExecuteCommandLists	
	clEnqueueCommandBufferKHR	cudaGraphLaunch

1. Implemented mapping 2. Intended mapping

Node/Edge Runtime Implementation

oneAPI

Node Implementation

- When a node is created by the graphs runtime code, the details about the command are extracted from the SYCL handler and stored in the node.

- Node is device specific as handler can use device information it normally gets from the queue.

Edge Implementation

- Graph runtime code bypasses existing scheduling to implement edges.

- Edges correspond to either
  a) A new PI sync point type that defines dependencies within a PI command-buffer.
  b) Graph partitioned into multiple command-buffers, synchronized with a PI event.

Future Work

Implementation Development

oneAPI

Goal - Complete implementation of extension revision 1 merged into mainline DPC++.

- Implement executable graph update feature.
- Ensure that buffer accessors correctly form edges.
- Command-group functionality can be captured in a node yet:
  - Host tasks
  - SYCL streams
  - Specialization constants

Specification Development

- Work towards a follow-up specification revision based on feedback:
  - Graph owned memory allocations.
  - A single graph having nodes targeting different devices.
  - More than one submission of the same executable graph in-flight at once.

- Merge with kernel-fusion extension.
  - See next talk “A SYCL Extension for User-Driven Online Kernel Fusion”.

Summary

- oneAPI vendor extension separating command construction from execution as a user accessible command graph.

- Benefits:
  - Remove redundant command construction overheads from repeated submission of the same command sequence.
  - Reduces latency when submitting commands to a device.
  - Provides optimization opportunities across the defined graph.

intel®

+

codeplay®