oneAPI

oneAPI Technical Advisory Board Meeting: SYCL Graph Extensions

Proposal for command graph support to SYCL

9/28/2022

Ewan Crawford and Pablo Reble

Rules of the Road

- DO NOT share any confidential information or trade secrets with the group
- DO keep the discussion at a High Level
  - Focus on the specific Agenda topics
  - We are asking for feedback on features for the oneAPI specification (e.g. requirements for functionality and performance)
  - We are NOT asking for feedback on any implementation details
- Please submit any implementation feedback in writing on Github in accordance with the Contribution Guidelines at spec.oneapi.com. This will allow Intel to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification.

Notices and Disclaimers

The content of this oneAPI Specification is licensed under the Creative Commons Attribution 4.0 International License . Unless stated otherwise, the sample code examples in this document are released to you under the MIT license.

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to The Khronos Group or other standard bodies under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, *you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback in its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.* For complete contribution policies and guidelines, see Contribution Guidelines on www.spec.oneapi.com.

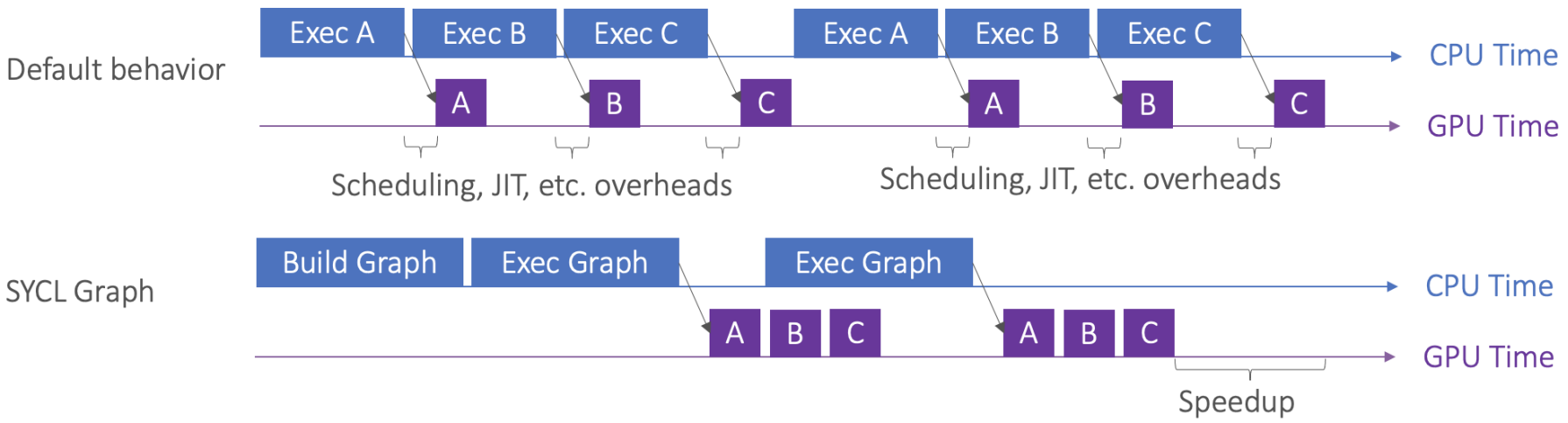
Introduction

What is this extension about?

- Explicit control over a command graph.
- Reduction in runtime overhead by only submitting a single graph object for execution rather than individual kernel commands.
- Enable whole graph optimizations, e.g inter-node memory reuse from memory staying resident on device.

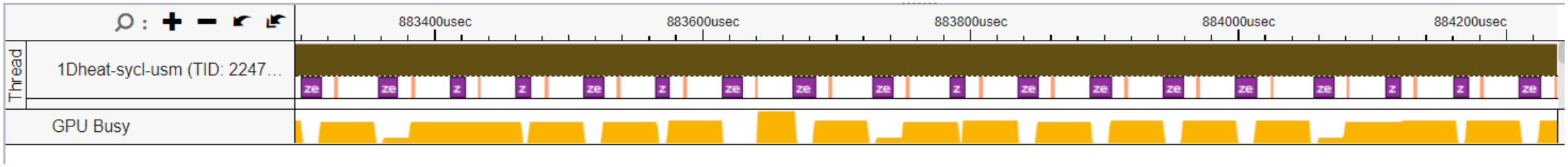
Motivation

- SYCL: single-source, higher-level, multi-platform programming
- Graph extensions: reusable task graph to reduce host overheads
- Especially beneficial for small and repetitive compute kernels

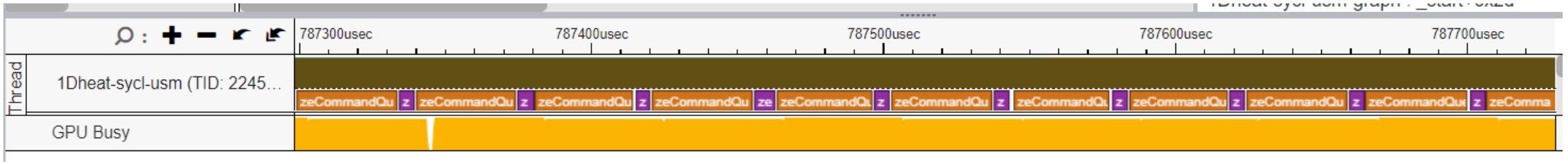


Motivation– 1Dheat example on GPU comparison

SYCL (default)



SYCL Graph POC



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

Comparison of Modes

Explicit API

- Direct Access to Low-Level Graph interface

Attributes:

- Existing code base needs migration
- More expressive
- Can be slightly more verbose

Record & Replay

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

Attributes:

- Easier to use when targeting an existing code base.
- More difficult to identify what memory is internal to graph for optimization.
- Harder to track USM dependencies between commands

Status

Explicit API

- PR https://github.com/intel/llvm/pull/5626
  - Spec only (*proposed)*
  - Stable beta by end of Q4’22 (experimental)
  - Prototype implementation with LevelZero

Record & Replay

- PR https://github.com/codeplaysoftware/standards-proposals/pull/135
  - A first draft and subject to change
  - Implemented in ComputeCpp proof-of-concept

These two approaches are compatible and both mechanisms can co-exist in a single extension

Explicit Mode

Overview and Workflow

- Command Graph

  User explicitly modifies a `command_graph` object
  `sycl::ext::oneapi::experimental::command_graph`

  A command graph can have two states: modifiable (default) and executable

- Node

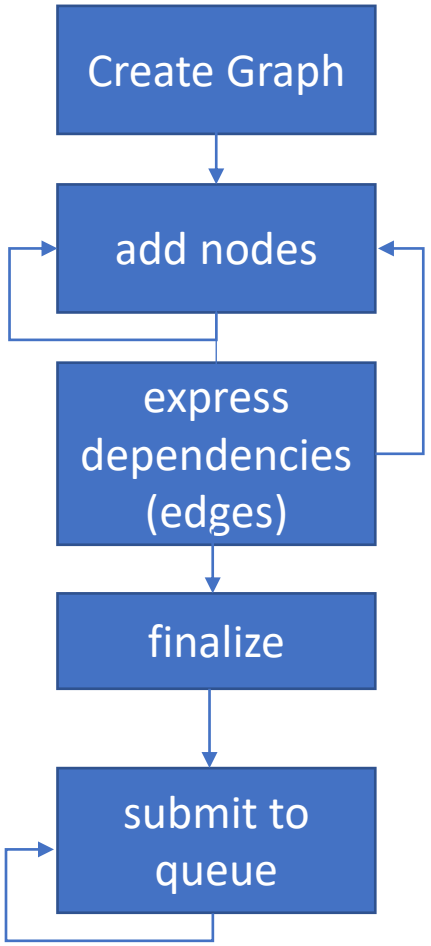
```
template<typename T>
  auto command_graph::add(T cgf); // has overloads
for empty node, subgraph, …
```

- Edge

```
template<typename Sender, typename Receiver>
  void command_graph::make_edge(Sender from, Receiver to);
```

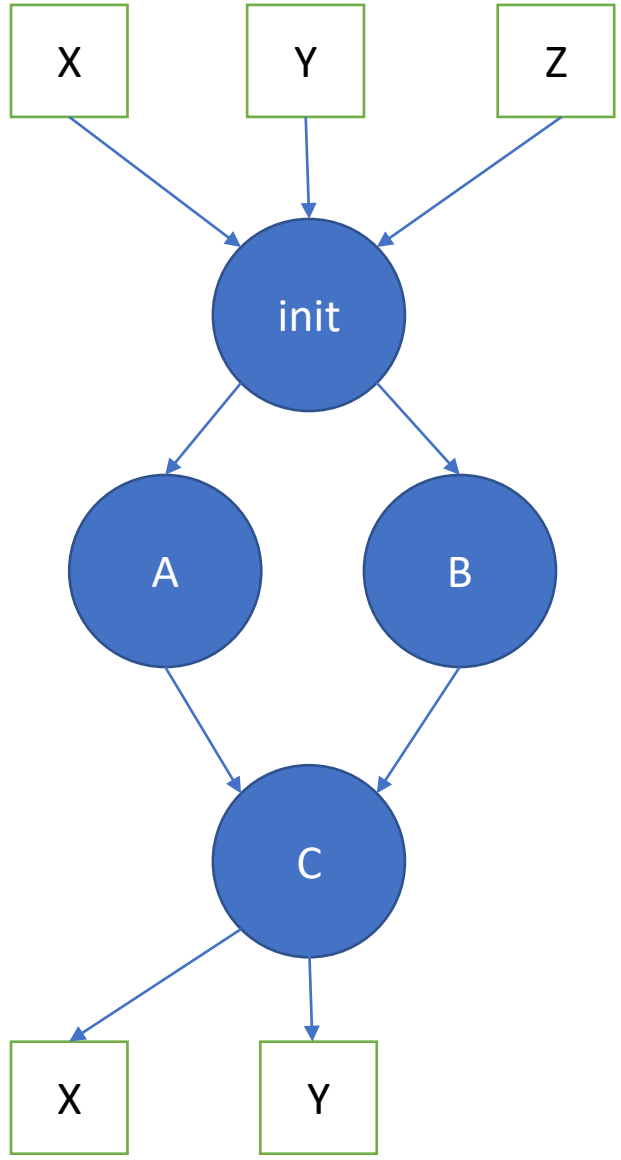
Note: API is experimental and will be subject to change

Workflow:



Code Example

```
26  sycl::ext::oneapi::experimental::command_graph g;
27
28  float *x , *y, *z;
29
30  auto n_x = g.add_malloc_device<float>(x, n);
31  auto n_y = g.add_malloc_device<float>(y, n);
32  auto n_z = g.add_malloc_device<float>(z, n);
33
34  float *dotp = sycl::malloc_shared<float>(1, q);
35  /* init data on device */
36  auto n_i = g.add([&](sycl::handler &h) {
37    h.parallel_for(n, [=](sycl::id<1> it){
38      const size_t i = it[0];
39      x[i] = 1.0f;
40      y[i] = 2.0f;
41      z[i] = 3.0f;
42    });
43  }, {n_x, n_y, n_z});
44  /* compute first input vector */
45  auto node_a = g.add([&](sycl::handler &h) {
46    h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> it) {
47      const size_t i = it[0];
48      x[i] = alpha * x[i] + beta * y[i];
49    });
50  }, {n_i});
51  /* compute second input vector */
52  auto node_b = g.add([&](sycl::handler &h) {
53    h.parallel_for(sycl::range<1>{n}, [=](sycl::id<1> it) {
54      const size_t i = it[0];
55      z[i] = gamma * z[i] + beta * y[i];
56    });
57  }, {n_i});
58
59  auto node_c = g.add([&](sycl::handler &h) {
60    h.parallel_for(sycl::range<1>{n},
61      sycl::reduction(dotp, 0.0f, std::plus()),
62        [=](sycl::id<1> it, auto &sum) {
63          const size_t i = it[0];
64          sum += x[i] * z[i];
65        });
66  }, {node_a, node_b});
67
68  auto node_f1 = g.add_free(x, {node_c});
69  auto node_f2 = g.add_free(y, {node_b});
```



Record & Replay

API Overview

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

// New object representing graph
template<graph_state State = graph_state::modifiable>
class command_graph {
public:
  /* Available only when: (State == graph_state::modifiable) */
  command_graph(const property_list &propList = {});
  /* Available only when: (State == graph_state::modifiable) */
  command_graph<graph_state::executable> finalize(context &syclContext) const;
  /* Available only when: (State == graph_state::executable) */
  void update(const command_graph<graph_state::modifiable> &graph);
};
```

API Overview

```
// State of a queue, returned by info::queue::state
enum class queue_state {
  executing,
  recording
};

// New methods added to the sycl::queue class
class queue {
public:
  bool begin_recording(command_graph<graph_state::modifiable> &graph);
  bool end_recording();
  event submit(command_graph<graph_state::executable> graph);
};
```

Note: API may be subject to change

What is captured as a graph?

- Node is a command-group submission, encompassing either one or both of a) some data movement, b) a single asynchronous kernel launch.
  - Explicit memory operations without kernels such as a memory copy as are still classes as nodes as they can be seen as specialized kernels executing on device.

- An edge is a data dependency between two nodes, expressed through buffer accessors or USM pointers.

Code Example

samples/networks/vgg/vgg.cc

View file @f2c314c1

```
496
                                                                    497
                                                                    498 +  cl::sycl::ext::codeplay::command_graph graph;
                                                                    499 +
                                                                    500 +  q.begin_recording(graph);
                                                                    501 +  network.run();
                                                                    502 +  q.end_recording();
                                                                    503 +
                                                                    504 +  auto exec_graph = graph.finalize(q.get_context());
                                                                    505 +
497     int loops = 10;                                             506     int loops = 10;
498     do {                                                        507     do {
499 -     auto status = network.run();                              508 +     q.submit(exec_graph);
500 -     status.event.wait_and_throw();
501     } while (--loops);                                          509     } while (--loops);
502                                                                 510
503     q.wait_and_throw();                                         511     q.wait_and_throw();
```

Whole Graph Update

- Feature for updating an executable graph with node inputs (buffers/USM) from a topologically identical modifiable graph.

- Enable performant double buffering use-case (submit, update, submit) without mandating requirements for all backends
  - Specified that effects of the update will be visible before next submission of the graph.
  - Specified that an instance of a graph can be submitted before a previous submission has completed execution, but there is no guarantee that submissions won't be serialized.

Implementation Backend

- ComputeCpp proof-of-concept uses OpenCL command-buffer extension cl_khr_command_buffer backend.
  - Same concept as Vulkan command-buffers and Level Zero command-lists

- A single graph can be partitioned into multiple backend command-buffer objects if the SYCL graph contains a command which is not representable in the backend, e.g. a host-task or USM memcpy.

- cl_khr_command_buffer_mutable_dispatch OpenCL extension used for whole graph update, but runtime could fallback to recreating backend object as a less performant alternative.

Future work

- Supporting a multi-device graph, i.e a single graph with sub-graphs marked as running on different devices but whole graph submitted at once.

- Provide a mechanism for the user to identify what memory is internal to graph for optimization.

- Improve USM dependency tracking. Currently is naïve - pointers used as node inputs must be the same to be a data dependency, nodes using different offsets into the same USM allocation won’t be identified as having an edge.

Summary

Summary

- Explicit API gives the user more control to achieve optimal performance, while record & replay approach allows faster porting of existing applications to use extension.

- Both mechanisms can exist in the same extension, and the user decides what approach is most suitable for their application.

- Next Steps
  - Merge Record & Replay functionality into Explicit API as a single oneAPI vendor extension.
  - Continued development of DPC++ prototype for unified extension.

oneAPI

Questions for Technical Advisory Board

- As a user of a graph API do you spot any incompatibilities or limitations with your projects?

- Proposal covers basic functionality, are you missing any features?