SYCL-Graph: A Progress Update

Codeplay: Ewan Crawford, Ben Tracy, Maxime France-Pillois

Intel: Pablo Reble, Julian Miller

oneAPI Language SIG
07/11/2023

Talk Agenda

- Introduction
- Specification Overview
- Implementation Details
- Demo
- Future Work
- Questions

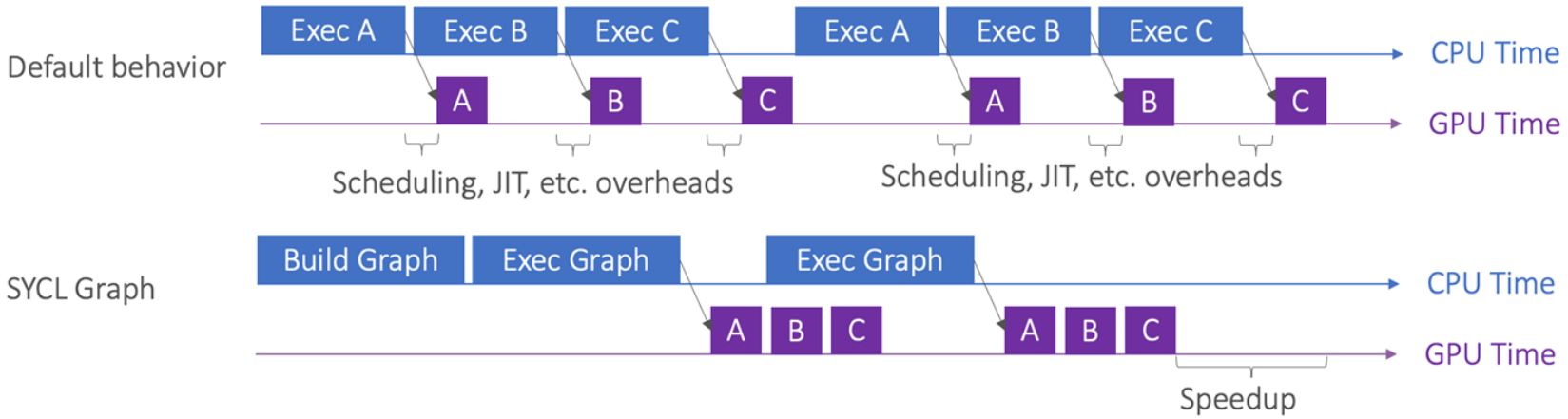
See IWOCL talk & backup slides for more in-depth API details
https://www.youtube.com/watch?v=aOTAmyr04rM

Motivation

- SYCL is already able to define a DAG of execution at runtime.
- Graph is implicit in the code with command creation and submission being tied together.
- Our extension provides a way to give the user control of the dependency graph in a construction step prior to execution.
- Concept already exists in other APIs
  - CUDA-Graph
  - HIP-Graph
  - Vulkan command-buffer
  - OpenCL cl_khr_command_buffer extension
  - Level Zero command-list

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, such as kernel fusion, become available across the defined graph.



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 Graph:
   - Tensorflow
   - PyTorch
   - GROMACS - https://gitlab.com/gromacs/gromacs/-/issues/4906
   - Kokkos

Progress Since Last Presentation to TAB

April 2022

- Two separate vendors extensions, each with their own implementation.
- Each vendor extension had a different mechanism for constructing a graph.
- Plans to unify the vendor extensions.

November 2023

- Unified into single sycl_ext_oneapi_graph extension.
- Consolidated efforts into a single DPC++ implementation in upstream intel/llvm
  - reble/llvm fork used for previous development now stale

Specification Overview

sycl_ext_oneapi_graph

- https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_graph.asciidoc

- First release base level of functionality, without advanced features.

- Experimental Status
  - APIs subject to change
  - Additions in ext::oneapi::experimental namespace

State Transition

Modifiable

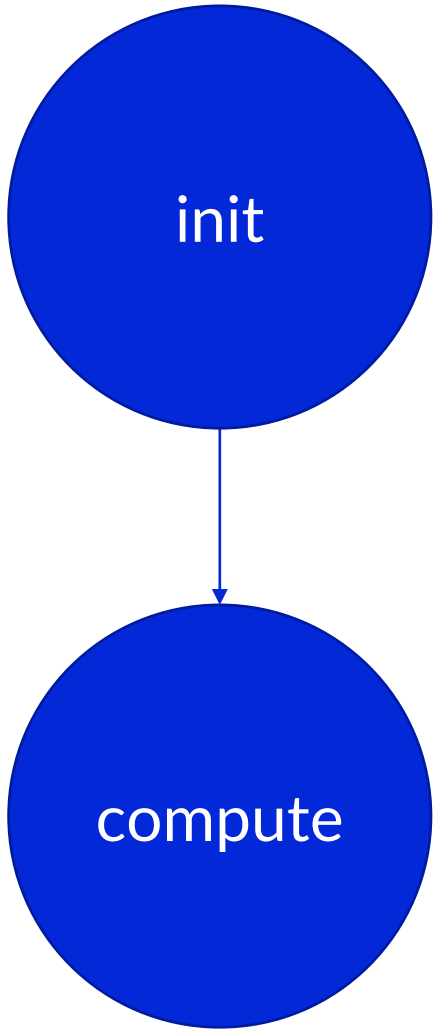
- Graph is under construction and new nodes may be added to it.

- 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.

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();
```

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();
```

Buffer Lifetimes – Current Situation

- Problem – A buffer object can be destroyed during lifetime of command_graph object.

```
void foo(queue q /* queue in recording mode */ ) {
  float data[NUM];
  buffer buf{data, range{NUM}};
  q.submit([&](handler &cgh) {
    accessor acc{buf, cgh, read_only};
    cgh.single_task([] {
       // use "acc"
    });
  });
  // "data" goes out of scope
}
```

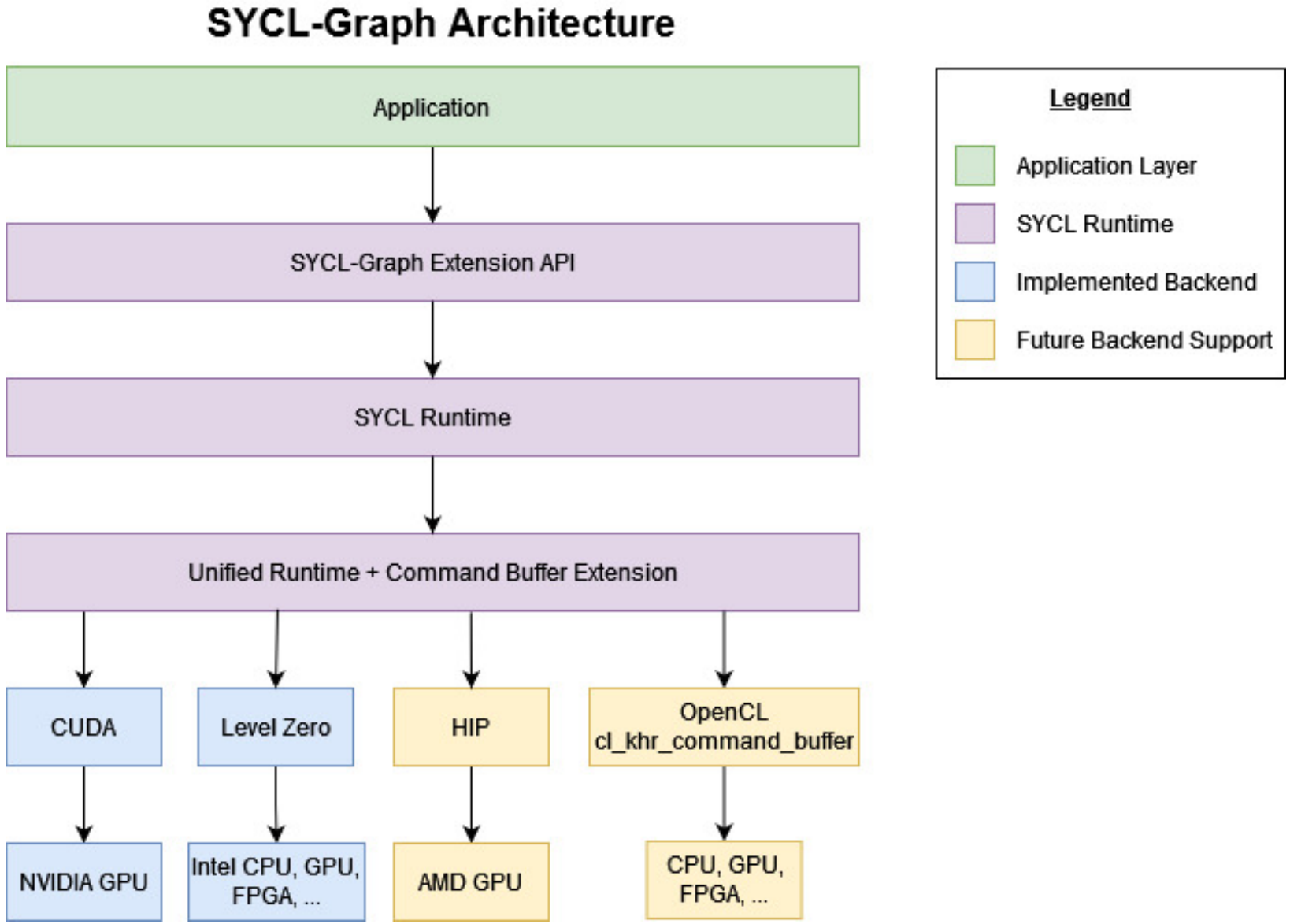
- Current Solution – Property for the user to guarantee buffer lifetime exceeds graph object. Otherwise, an exception is thrown when using a buffer in graph node.

```
sycl_ext::command_graph graph(q.get_context(), q.get_device(),
      {sycl_ext::property::graph::assume_buffer_outlives_graph{}});
```

Buffer Lifetimes – Planned Solution

- We plan for the lifetime of buffers to be extended when used in a graph, but not implemented yet.
- Change assume_buffer_outlives graph to assume_data_outlives_graph for host data going out of scope, and take copy of host data when property not set.
- Restrictions around write back.
  - Can’t submit a command with an accessor on a buffer that would cause write back to happen.
  - Can’t use host_accessor when a buffer is in use by a graph.
  - Must use set_write_back(nullptr) or set_final_data(nullptr)

Implementation Details


SYCL-Graph Architecture
Application
SYCL-Graph Extension API
SYCL Runtime
Unified Runtime + Command Buffer Extension
CUDA
Level Zero
HIP
OpenCL
cl_khr_command_buffer
NVIDIA GPU
Intel CPU, GPU, FPGA, ...
AMD GPU
CPU, GPU, FPGA, ...
Legend
Application Layer
SYCL Runtime
Implemented Backend
Future Backend Support

Unified Runtime command-buffer

- DPC++ has an intermediate abstraction API called Unified Runtime (UR) that is implemented by SYCL-2020 backends.

- We've extended this interface to add a new command-buffer type and entry-points.
  - https://oneapi-src.github.io/unified-runtime/core/EXP-COMMAND-BUFFER.html
  - An experimental feature, not yet implemented for all UR adapters with more API features to come.

UR Backend Implementation Status

- Level Zero
  - Full coverage of UR command-buffer API.
  - The most mature & well tested backend.

- CUDA
  - Full coverage of UR command-buffer API implemented using CUDA-Graph driver API.
  - Merged November 2023.

UR Backend Implementation Status

- OpenCL
  - Partial support when OpenCL backend supports cl_khr_command_buffer
  - https://github.com/intel/llvm/pull/11718 not yet merged.
  - Limitations in type of nodes which can be used. e.g cl_khr_command_buffer doesn't support USM mem fill & memset commands being recorded to a command-buffer.
- HIP
  - Not started.

Node Runtime Implementation

- When a node is created, the details about the command-group are extracted from the SYCL handler and stored in the runtime node class.
- Node is device specific, as handler uses device information it normally gets from the queue.
- Nodes stored as a list in the graph implementation class for iterative searching.

Edge Runtime Implementation

- Each node stores a list of predecessor & successor nodes as weak pointers.
- Edges are automatically created for buffer accessor dependencies between a newly created node and any leaf nodes using the same buffer.
- Graph edges are mapped to UR sync-points, a synchronization primitive that can only represents dependencies inside the same command-buffer.

Finalizing To An Executable Graph

Nodes stored in the runtime are added to UR command-buffer:

- For a node using USM only, the graph runtime adds command directly to UR command-buffer.
- Nodes with buffer accessor dependencies are processed by scheduler as a new type of command for adding node to command-buffer.
  - Scheduler can identify accessor dependencies on scheduler commands outside the user defined graph, e.g. memory allocation for the buffer.

Support Status

Core Command-group Feature Support

- Kernels & memcpy are the two primary types of SYCL command-group our implementation currently supports as nodes in a graph.
  - Kernel bundle
  - Specialization constants - https://github.com/intel/llvm/pull/11556
- Command-group features that can’t currently be used in a graph node.
  - Host tasks
  - Reductions
  - SYCL streams
- Implemented, but not yet merged upstream.
  - Prefetch & mem_advise - https://github.com/intel/llvm/pull/11474
  - Fill & memset - https://github.com/intel/llvm/pull/11472

Extension Support

- Other extensions can define handler methods and queue shortcuts for command-group functionality.
- Right now, we only need to be concerned about compatibility with other oneAPI vendor extensions.
- If SYCL-Graph was a KHR extension, then compatibility with all the extensions in the ecosystem is a harder question.
- Only extension we currently support is sycl_ext_oneapi_enqueue_barrier

oneDNN Demo

oneDNN Demo

- oneDNN is the oneAPI DNN library which has a SYCL backend.
- Release v3.3 contains a commit that enables modified cnn_inference_fp32.cpp example to run using SYCL-Graph API.
  - https://github.com/oneapi-src/oneDNN/tree/rls-v3.3
  - https://github.com/oneapi-src/oneDNN/commit/a96e9b1a6769171e74b0b8e03148930343890 6e5
  - No verification of results in example.
- Not using the extension as part of the oneDNN framework itself.

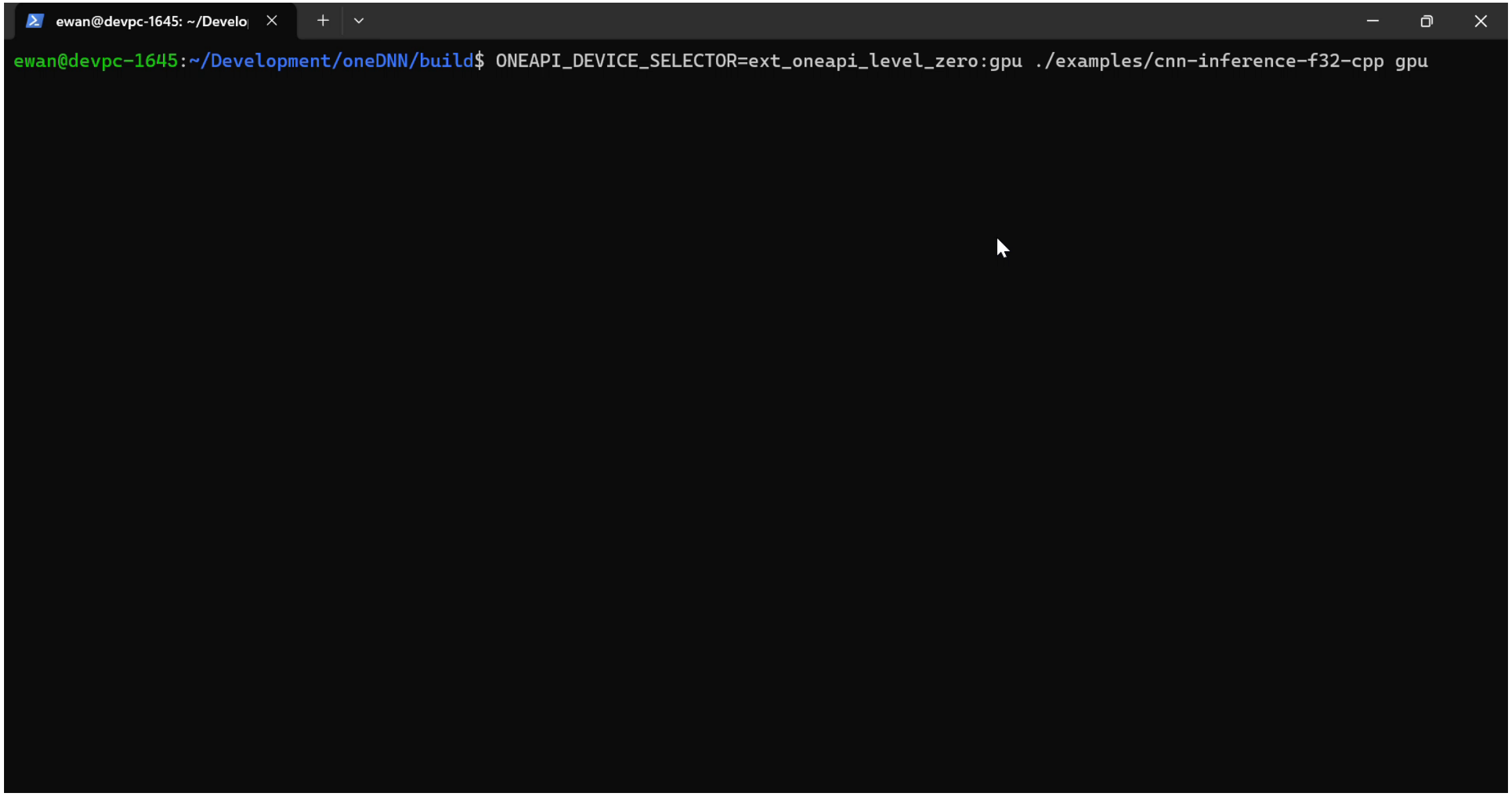
Disclaimer: General support for recording oneAPI library calls is WIP

cnn_inference_f32.cpp

https://gist.github.com/EwanC/70f056e07c4a0d5c43cae200de833b19

```
diff --git a/examples/cnn_inference_f32.cpp b/examples/cnn_inference_f
index 355f0c115..725faa0c3 100644
--- a/examples/cnn_inference_f32.cpp
+++ b/examples/cnn_inference_f32.cpp
@@ -49,6 +49,7 @@
 #include "example_utils.hpp"

 using namespace dnnl;
+using namespace sycl;

 void simple_net(engine::kind engine_kind, int times = 100) {
     using tag = memory::format_tag;
@@ -59,7 +60,10 @@ void simple_net(engine::kind engine_kind, int times
     /// @snippet cnn_inference_f32.cpp Initialize engine and stream
     //[Initialize engine and stream]
     engine eng(engine_kind, 0);
-    stream s(eng);
+    dnnl::stream s(eng);
+
+    queue q = sycl_interop::get_queue(s);
+
     //[Initialize engine and stream]

     /// Create a vector for the primitives and a vector to hold memor
```

```
@@ -751,19 +755,26 @@ void simple_net(engine::kind engine_kind, int times = 100) {
                 {DNNL_ARG_TO, user_dst_memory}});
     }

+    ext::oneapi::experimental::command_graph g {
+            q.get_context(), q.get_device()};
+    g.begin_recording(q);
+
+    assert(net.size() == net_args.size() && "something is missing");
+    for (size_t i = 0; i < net.size(); ++i)
+        net.at(i).execute(s, net_args.at(i));
+    g.end_recording(q);
+    auto execGraph = g.finalize();
+
     /// @page cnn_inference_f32_cpp
     /// Finally, execute the primitives. For this example, the net is executed
     /// multiple times and each execution is timed individually.
     /// @snippet cnn_inference_f32.cpp Execute model
     //[Execute model]
-    for (int j = 0; j < times; ++j) {
-        assert(net.size() == net_args.size() && "something is missing");
-        for (size_t i = 0; i < net.size(); ++i)
-            net.at(i).execute(s, net_args.at(i));
+    for (int j = 0; j < times - 1; ++j) {
+        q.ext_oneapi_graph(execGraph);
+        q.wait_and_throw();
     }
     //[Execute model]
-
-    s.wait();
 }
```


ewan@devpc-1645: ~/Develo
ewan@devpc-1645:~/Development/oneDNN/build$ ONEAPI_DEVICE_SELECTOR=ext_oneapi_level_zero:gpu ./examples/cnn-inference-f32-cpp gpu

Future Work

Filling Implementation Of Current Spec

- Command-group features not yet supported
  - Host-tasks
  - Reductions
  - Sycl streams
  - Support for other extensions
- New Backends
  - Full support on OpenCL cl_khr_command_buffer
  - HIP
- Profiling
  - Can’t currently get the timestamps from the event returned by a graph submission.
- Bug Fixes
  - Our extension will be stressed more as it is used in the real world.

CUDA-Graph Differences

- Transitive stream/queue capture
  - In SYCL-Graph you must specify upfront the queues a graph will record commands from, if one of these queues depends on an event from another queue, this is an error.
  - Using CUDA-Graph transitive stream capture model, the command associated with this event would be implicitly captured.
  - https://pytorch.org/docs/master/notes/cuda.html#usage-with-multiple-streams

- External Event node - cudaEventRecordExternal
  - In CUDA-Graph an event can be explicitly added to the graph as an external event node when performing stream capture.

Missing Features

- Update arguments to graph nodes
  - WIP drafting APIs for whole graph update & individual node update.
  - Can be implemented by cl_khr_command_buffer_mutable_dispatch or cudaGraphExecUpdate()
  - Level Zero doesn't have an API we can use for this yet.

- Graph owned memory allocation
  - Graph to be able to make and use memory allocations that are tied to the lifetime and scope of the graph.
  - Based on our ideas so far, backends would need to be able to make virtual memory allocations, that are only backend up with physical memory at finalize time.

Device Model

Currently a device is associated with the graph on creation.

- Multi-device graph
  - Single command-graph object with nodes targeting different devices.
  - Mixed backend support.
  - How would submission be defined when SYCL queue tied to a single device

- Device agnostic graph
  - Modifiable graph isn't associated with a device.
  - Executable graph is given a device at finalize() time.
  - Technical difficulties in achieving this as handler internals assumes a device when creating the graph.

SYCL Specific Features

- Buffer Lifetimes
  - Implement extending buffer lifetimes when used in a graph.
  - Frameworks can use internal scratchpad buffers that don't exceed the lifetime of the graph without intervention from the runtime.
- User Guided Scheduling
  - Allow users to specify how they want to schedule nodes for execution.
  - Breadth-first, depth-first, based on annotations to command-group.
- Concurrent Graph Execution
  - Allow an executable graph instance to be execute while another instance is already executing.
  - Only adds value if graph update exists so that arguments can be updated in-between submissions.

Graph Fusion

- Proposed layered extension for fusing kernels in graph nodes
  - https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/proposed/sycl_ext_oneapi_graph_fusion.asciidoc
- Keep ideas from Codeplay SYCL Kernel Fusion proposal
  - https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_codeplay_kernel_fusion.asciidoc
  - Updated some APIs to clarify and integrate better with SYCL-Graph.
- Implementation still outstanding
  - JIT compiler from Kernel Fusion can be re-used for Graph Fusion

Summary

- sycl_ext_oneapi_graph is now ready to try in upstream intel/llvm source

- Let us know what future work is a priority for you.

- How to contact us feedback
  - Email: sycl.graph@codeplay.com + pablo.reble@intel.com
  - File a GitHub issue on intel/llvm or reble/llvm

Thank you!

Disclaimers

A wee bit of legal

No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Backup Slides

Nodes & Edges

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

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 & 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
};
```

Executable Graph Submission

- handler::depends_on can express graph submission dependencies.
- Nested graphs 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);
}
```

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();
```

Function	Description
`urCommandBufferCreateExp`	Create a command-buffer.
`urCommandBufferRetainExp`	Incrementing reference count of command-buffer.
`urCommandBufferReleaseExp`	Decrementing reference count of command-buffer.
`urCommandBufferFinalizeExp`	No more commands can be appended, makes command-buffer ready to enqueue on a command-queue.
`urCommandBufferAppendKernelLaunchExp`	Append a kernel execution command to command-buffer.
`urCommandBufferAppendUSMMemcpyExp`	Append a USM memcpy command to the command-buffer.
`urCommandBufferAppendUSMFillExp`	Append a USM fill command to the command-buffer.
`urCommandBufferAppendMemBufferCopyExp`	Append a mem buffer copy command to the command-buffer.
`urCommandBufferAppendMemBufferWriteExp`	Append a memory write command to a command-buffer object.
`urCommandBufferAppendMemBufferReadExp`	Append a memory read command to a command-buffer object.
`urCommandBufferAppendMemBufferCopyRectExp`	Append a rectangular memory copy command to a command-buffer object.
`urCommandBufferAppendMemBufferWriteRectExp`	Append a rectangular memory write command to a command-buffer object.
`urCommandBufferAppendMemBufferReadRectExp`	Append a rectangular memory read command to a command-buffer object.
`urCommandBufferAppendMemBufferFillExp`	Append a memory fill command to a command-buffer object.
`urCommandBufferEnqueueExp`	Submit command-buffer to a command-queue for execution.

UR Backend Mapping

UR API Addition	Intel Level Zero	
	OpenCL cl_khr_command_buffer Extension	CUDA Graphs
ur_exp_command_buffer_handle_t	ze_command_list_handle_t	
	cl_command_buffer_khr	cudaGraph_t
urCommandBufferCreateExp	zeCommandListCreate	
	clCreateCommandBufferKHR	cudaGraphCreate
urCommandBufferFinalizeExp	zeCommandListClose	
	clFinalizeCommandBufferKHR	cudaGraphInstantiate
urCommandBufferAppendKernelLaunchExp	zeCommandListAppendLaunchKernel	
	clCommandNDRangeKernelKHR	cudaGraphAddKernelNode
urCommandBufferFinalizeExp	zeCommandQueueExecuteCommandLists	
	clEnqueueCommandBufferKHR	cudaGraphLaunch