sycl_ext_oneapi_graph (SYCL 拡張 oneAPI グラフ)

この記事は、GitHub にあるインテル社の LLVM レポジトリーで公開されている「sycl_ext_oneapi_graph」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

目次

1. 著作権と商標について

2. コンタクト

この拡張機能に関する問題を報告するには、次のサイトで新しい問題をオープンしてください:
https://github.com/intel/llvm/issues

3. 共著者

Pablo Reble, Intel
Julian Miller, Intel
John Pennycook, Intel
Guo Yejun, Intel
Dan Holmes, Intel
Greg Lueck, Intel
Steffen Larsen, Intel
Jaime Arteaga Molina, Intel
Andrei Elovikov, Intel
Ewan Crawford, Codeplay
Ben Tracy, Codeplay
Duncan McBain, Codeplay
Peter Žužek, Codeplay
Ruyman Reyes, Codeplay
Gordon Brown, Codeplay
Erik Tomusk, Codeplay
Bjoern Knafla, Codeplay
Lukas Sommer, Codeplay
Maxime France-Pillois, Codeplay
Jack Kirk, Codeplay
Ronan Keryell, AMD
Andrey Alekseenko, KTH Royal Institute of Technology
Fábio Mestre, Codeplay
Konrad Kusiak, Codeplay

4. 依存関係

この拡張機能は、SYCL 2020 リビジョン 10 仕様に基づいて作成されています。以下の「コア SYCL 仕様」、または SYCL 仕様のセクション番号のすべての参照はこのリビジョンを指しています。

5. 状況

これは実験的な拡張機能仕様であり、機能への早期アクセスを提供し、コミュニティーからのフィードバックを得ることを目的としています。本仕様で定義されるインターフェイスは DPC++ で実装されていますが、これらは確定されたものではなく、将来の DPC++ バージョンでは事前の通知なく互換性のない変更が行われる可能性があります。**ソフトウェア製品は、本仕様で定義された API に依存すべきではありません。**

6. バックエンド・サポートの状況

この拡張機能は実験的であるため、すべてのデバイスでサポートされるわけではありません。

表 1. デバイスサポートの特徴

デバイス記述子	説明
`aspect::ext_oneapi_graph`	デバイスがこの拡張機能のすべての API をサポートしていることを示します。
`aspect::ext_oneapi_limited_graph`	デバイスが、実行可能グラフ更新セクションで説明されているものを除き、この拡張機能のすべての API をサポートしていることを示します。これは一時的な機能であり、完全なグラフサポートを備えたデバイスが普及したら削除される予定です。

`aspect::ext_oneapi_graph` を報告するデバイスは、`aspect::ext_oneapi_limited_graph` も報告する必要があります。これは、`aspect::ext_oneapi_graph` が限定アスペクトのスーパーセットであるためです。

7. 概要

コマンドグループを使用すると、SYCL は実行時にカーネル実行の暗黙的な依存関係グラフ (有向非巡回グラフの形式) を作成できます。これは、コマンドグループ・オブジェクトが、コマンド (ノード) を実行するために満たす必要のある一連の要件 (エッジ) を定義するためです。ただし、コマンドグループの送信はキューの実行に関連付けられており、実行を開始する前に事前の構築手順がないため、実行前に定義された依存関係グラフがランタイムに認識されず、最適化の可能性が失われます。

ユーザーが実行前に SYCL ランタイムへの依存グラフを定義できれば、次のような利点が得られます:

- 複数の個別のコマンドグループではなく、単一のグラフ・オブジェクトのみを送信することで、実行時のオーバーヘッドを削減できます。
- 事前により多くの作業を実行できるように準備して、実行時のパフォーマンスを向上させます。この初期作業は、グラフを繰り返し実行する前のプログラムのセットアップ段階で行います。あるいは、アプリケーションの実行前に、別のプロセス内のオフライン AOT コンパイラーを実行することもできます。
- ランタイムによるグラフ分析により DMA ハードウェア機能を利用できるようにします。
- グラフ最適化が利用可能になります。これには以下が含まれますが、これらに限定されません:
    - カーネルの融合 (フュージョン) と分裂 (フィッション)
    - デバイス上に常駐するデータのノード間でのメモリーの再利用。
    - より最適なメモリー割り当てに使用される、ピーク中間出力メモリー要件の識別。

SYCL ランタイムの利点に加えて、反復的なワークロードで同じコマンドシーケンスを冗長的に発行する必要がなくなるため、SYCL アプリケーションを開発するユーザーにも利点があります。代わりに、グラフは一度だけ作成され、入力バッファーまたは USM 割り当て内のデータのみを変更しながら、必要な回数だけ実行に向けて送信されます。マシンラーニングなど、異なる入力に対して同じコマンドグループのパターンを繰り返し実行する特定のドメインのアプリケーションでは特に便利です。

7.1. 必要条件

前のセクションで説明した目標を達成するため、次の要件が考慮されました:

1. 全体のグラフ構造を変更せずに、送信間でグラフのパラメーターを更新する機能。
2. 拡張機能を使用するため、既存のアプリケーションを簡単に移植できるようにします。
3. グラフノードの粒度でプロファイル、デバッグ、およびトレースを行う機能。
4. 新しいグラフを構築するときに、サブグラフ (事前に構築されたグラフ) を統合します。
5. USM メモリーモデルとバッファー/アクセサーモデルの両方をサポートします。
6. 他の SYCL 拡張機能や機能 (例:カーネル融合や組み込みカーネル) と互換性があります。
7. 同じコンテキスト内の異なるデバイスに送信されたコマンドを含むグラフを記録する機能。
8. グラフをバイナリー形式にシリアル化し、その後シリアル化を解除して実行する機能。これは、エンドユーザーがグラフ作成のオーバーヘッドを負担することなく、オフラインツールでグラフを作成し、読み込んで実行できる AOT を行う場合に役立ちます。
9. バックエンドの相互運用性: グラフからネイティブ・グラフ・オブジェクトを取得し、それをネイティブ・バックエンド API で使用する機能。

この拡張機能のプロトタイプ実装を迅速に開発して評価できるように、この提案の適用範囲はこれらの要件のサブセットに限定されました。特に、シリアル化機能 (8)、バックエンドの相互運用性 (9)、およびプロファイル/デバッグインターフェイス (3) が省略されました。十分な調査を行うことなく複数のバックエンドに抽象化するのは容易ではありません。また、これらの機能が API への追加変更として公開され、将来の拡張機能バージョンに導入されることも期待されています。

シリアル化/シリアル化の解除 API を延期するもう一つの理由 (8) は、その適用範囲をバイナリー形式でのグラフ出力から、デバイス固有のグラフ最適化を可能にする標準化された中間表現 (IR) 形式の出力まで拡大する可能性があるためです。

マルチデバイスのサポート (7) は、今後の改訂で拡張機能に導入することを検討していますが、API が変更される可能性もあります。グラフノードの定義はデバイス固有となるように計画されていますが、現在はグラフ内のすべてのノードがグラフ・コンストラクターに提供されるのと同じデバイスをターゲットにする必要があります。

7.2. グラフ構築メカニズム

この拡張機能には、コマンドのグラフを構築する 2 つの異なる API メカニズムが含まれています:

1. **明示的なグラフ構築 API** - ユーザーがグラフに追加するノードとエッジを正確に指定できるようにします。
2. **キュー記録 API (別名「記録と再生」)** - コマンドをすぐに実行するようにスケジュールするのではなく、コマンドグループの依存関係からキャプチャーされたエッジを使用して、コマンドがグラフ・オブジェクトに追加されるように、`sycl::queue` に状態を導入します。

グラフを構築するこれらのメカニズムにはそれぞれ独自の利点があるため、両方の API を利用できることで、ユーザーは自分に最も適したものを選択できます。キュー記録 API は既存のアプリケーションの移植を促進し、ライブラリー関数呼び出しなどを通じてキューに送信される外部ワークをキャプチャーできます。明示的な API は、最適化のためグラフ内部のデータをより明確に表現でき、依存関係を推測する必要がありません。

これら 2 つのメカニズムの組み合わせは許されますが、グラフが任意のキューからのコマンドを記録しているときに、明示的な API を使用してグラフを変更することはできません。次に例を示します:

```
graph.begin_recording(queue);
graph.add(/*command group*/);    // グラフがキューを記録しているため無効
graph.end_recording();
```

8. 仕様

8.1. 機能を評価するマクロ

この拡張機能は、コア SYCL 仕様で説明されている機能評価マクロを提供します。この拡張機能をサポートする実装では、以下の表に定義されている値のいずれかに `SYCL_EXT_ONEAPI_GRAPH` マクロを事前定義する必要があります。アプリケーションは、このマクロをテストして、実装がこの機能をサポートしているか判断できます。また、アプリケーションは、マクロの値をテストして、実装がどの拡張機能の API をサポートしているかも判断できます。

表 2. `SYCL_EXT_ONEAPI_GRAPH` マクロの値

値	説明
1	初期の拡張バージョン。基本機能がサポートされています。

8.2. SYCL グラフの用語

表 3. 用語:

コンセプト	説明
グラフ	コマンド (ノード) とその依存関係 (エッジ) の有向非巡回グラフ (DAG)。 `command_graph` クラスによって表されます。
ノード	特定のデバイスを対象とする、さまざまな属性を持つコマンド。
エッジ	ハップンズビフォー関係としてのコマンド間の依存関係。

8.2.1. 明示的なグラフ構築 API

明示的なグラフ構築 API を使用してグラフを構築する場合、ノードとエッジは次のようにキャプチャーされます。

表 4. 明示的なグラフ定義。

コンセプト	説明
ノード	明示的なグラフ構築 API では、ユーザーがコマンドグループ関数 (CGF) を渡して変更可能なグラフ上でメソッドを呼び出すことでノードが作成されます。各ノードは、コマンドグループまたは空の操作のどちらかを表します。
エッジ	明示的なグラフ構築 API では主に、エッジは新しく追加されたインターフェイスを通じてユーザーによって定義されます。これには、`make_edge()` 関数を使用して既存のノード間のエッジを定義するか、グラフに新しいノードを追加するときに `property::node::depends_on` プロパティー・リストを使用します。 依存関係を表現するため既存の SYCL メカニズムを通じてグラフにノードを明示的に追加するときに、エッジを作成することもできます。グラフ内の既存ノードへのアクセサーからのデータ依存関係は、エッジとしてキャプチャーされます。`handler::depends_on()` を使用すると、キューの記録によってキャプチャーされたキューの送信から返されたイベントが同じグラフに渡されたときにもグラフエッジが作成されます。

8.2.2. キューの記録 API

記録と再生 API を使用してキューを記録してグラフを構築する場合、ノードとエッジは次のようにキャプチャーされます。

表 5. 記録されたグラフ定義。

コンセプト	説明
ノード	キュー記録グラフ内のノードは、記録対象のキューに関連付けられたデバイスへのコマンドグループの送信を表します。ノードは、`queue::submit()` に渡されるコマンドグループ関数 (CGF) から、または定義されたハンドラーのコマンドタイプに対するキューのショートカットから構築されます。各送信は、次のいずれか一方または両方を含んでいます: a.) データ移動の一部 b.) 単一の非同期コマンド起動ノードは前方エッジを定義できず、後方エッジのみ定義できます。 つまり、ノードはすでに送信されたコマンドグループに対してのみ依存関係を作成できます。
エッジ	キューの記録グラフのエッジは、コマンドグループの依存関係を通じて次の 3 つの方法のいずれかで表現されます。まず、ノードとしてキャプチャーされた 2 つのコマンドグループ間のデータ依存関係を表すバッファーアクセサーを使用します。次に、ノードとしてキャプチャーされたコマンドグループ内で `handler::depends_on()` メカニズムを使用します。ただし、エッジを作成するため `handler::depends_on()` に渡されるイベントは、同じグラフでキャプチャーされたキュー送信から返されたイベントである必要があります。それ以外の場合、エラーコードが invalid の同期エラーがスローされます。`handler::depends_on()` は、ユーザーが SYCL バッファーではなく USM メモリーを使用してエッジを表現するために使用できます。最後に、順序付きキューで記録されたグラフでは、順序付きキューに送信された 2 つの連続したコマンドグループ間にエッジが自動的に追加されます。

8.2.3. サブグラフ

グラフ内のノードは、ネストされたサブグラフの形式にできます。これは、実行可能なグラフ・オブジェクトを使用して `handler::ext_oneapi_graph()` を呼び出すコマンドグループの送信が、ノードとしてグラフに追加されたときに発生します。子グラフノードは、子グラフのルートノードを親グラフの依存ノードに接続するエッジが作成されたかのように、親グラフにスケジュールされます。

実行可能グラフをサブグラフとして追加しても、既存のノード依存関係には影響しません。そのため、サブグラフとして以前に使用した時の副作用はなく以降に送信できます。

8.3. ノード

```
namespace sycl::ext::oneapi::experimental {
enum class node_type {
  empty,
  subgraph,
  kernel,
  memcpy,
  memset,
  memfill,
  prefetch,
  memadvise,
  ext_oneapi_barrier,
  host_task,
  async_malloc,
  async_free
};

class node {
public:
  node() = delete;

  node_type get_type() const;

  std::vector<node> get_predecessors() const;

  std::vector<node> get_successors() const;

  static node get_node_from_event(event nodeEvent);

  template <int Dimensions>
  void update_nd_range(nd_range<Dimensions> executionRange);

  template <int Dimensions>
  void update_range(range<Dimensions> executionRange);
};

}  // sycl::namespace ext::oneapi::experimental
```

ノードは、SYCL カーネル関数や遅延実行メモリー操作などのタスクをカプセル化するクラスです。まずグラフを作成する必要があり、次にノードとエッジを追加することでグラフの構造が定義されます。

ノードクラスは共通の参照セマンティクスを提供します。

8.3.1. ノードメンバー関数

```
node_type get_type() const;
```

リターン: このノードが表すコマンドのタイプを表す値。

```
std::vector<node> get_predecessors() const;
```

リターン: このノードが直接依存する先行ノードのリスト。

```
std::vector<node> get_successors() const;
```

リターン: このノードに直接依存する後継ノードのリスト。

```
static node get_node_from_event(event nodeEvent);
```

効果: 記録状態のキューへの送信から作成されたイベント `nodeEvent` に関連付けられたノードを検索します。
リターン: `nodeEvent` を返すコマンドが送信されたときに作成されたグラフノード。
スロー: `nodeEvent` がグラフノードに関連付けられていない場合、エラーコードが `invalid` の例外が発生します。

```
template <int Dimensions>
void update_nd_range(nd_range<Dimensions> executionRange);
```

効果: このノードの Nd-range を新しい値 `executionRange` で更新します。この新しい値は、実行可能グラフの更新関数に渡されるまで、このノードが属する実行可能グラフには影響しません。 カーネルノードの更新に関する詳細は、実行可能グラフの更新を参照してください。
条件: `Dimensions` が 1、2、または 3 の場合にのみ使用できます。
スロー:

- `Dimensions` が既存のカーネル実行レンジの次元と一致しない場合、エラーコード `invalid` の例外が発生します。
- ノードのタイプがカーネル実行でない場合、エラーコードが `invalid` の例外が発生します。

```
template <int Dimensions>
void update_range(range<Dimensions> executionRange);
```

効果: このノードの実行レンジを新しい値 `executionRange` で更新します。この新しい値は、実行可能グラフの更新関数に渡されるまで、このノードが属する実行可能グラフには影響しません。カーネルノードの更新に関する詳細は、実行可能グラフの更新を参照してください。
条件: `Dimensions` が 1、2、または 3 の場合にのみ使用できます。
スロー:

- `Dimensions` が既存のカーネル実行レンジの次元と一致しない場合、エラーコード `invalid` の例外が発生します。
- ノードのタイプがカーネル実行でない場合、エラーコードが `invalid` の例外が発生します。

8.3.2. 動的パラメーター

```
namespace ext::oneapi::experimental{
template <typename ValueT>
class dynamic_parameter {
public:
  dynamic_parameter(command_graph<graph_state::modifiable> graph, const ValueT
&initialValue);

  void update(const ValueT& newValue);
};
}
```

動的パラメーターは、ノードがグラフに追加された後にユーザーが更新できるノードのコマンドグループへの引数です。動的パラメーターの値を更新すると、このノードを含む変更可能なグラフに反映されます。更新されたノードは、実行可能なグラフに渡され新しい値で更新されます。

動的パラメーターが表す基になるオブジェクトのタイプは、テンプレート・パラメーターを使用してコンパイル時に設定されます。この基になるタイプは、アクセサー、USM 割り当てへのポインター、値渡しのスカラー、または引数の生のバイト表現になります。生のバイト表現は、`sycl_ext_oneapi_raw_kernel_arg` を使用して設定された引数の更新を可能にすることを目的としています。

動的パラメーターは変更可能なグラフ内のノードに登録され、登録ごとに 1 つ以上のノード引数が動的パラメーター・インスタンスに関連付けられます。登録は、ノードが表すコマンドグループ内で行われ、`handler::set_arg()`/`handler::set_args()` を使用して動的パラメーターがカーネルへのパラメーターとして設定されるときに実行されます。ノード引数を複数の動的パラメーター・インスタンスに登録することもできます。

ノード・パラメーターの更新に関する詳細は、実行可能グラフの更新を参照してください。

`dynamic_parameter` クラスは共通の参照セマンティクスを提供します。

dynamic_parameter クラスのメンバー関数

```
dynamic_parameter(command_graph<graph_state::modifiable> graph,
                  const ValueT &initialValue);
```

効果: 初期値 `initialValue` を使用して、グラフからコマンド・グラフ・ノードに登録できる動的パラメーター・オブジェクトを構築します。

[注:グラフを受け取るコンストラクターは非推奨となり、次の ABI ブレークウィンドウで置き換えられます。新しいコンストラクターは、動的パラメーターを特定のグラフに関連付けません。— 注終了]

```
void update(const ValueT& newValue);
```

効果: この動的パラメーターに登録されているすべてのノードのパラメーターを `newValue` に更新します。新しい値は、登録されたノードを含む変更可能なグラフに直ちに反映されます。変更可能なグラフから作成された実行可能グラフには、変更されたノードを渡して `command_graph::update()` が呼び出されるか、変更可能なグラフから新しい実行可能グラフが確定されるまで、新しい値は反映されません。

登録されたノードのいずれかで `newValue` が現在のパラメーター値に設定されても、エラーにはなりません。

8.3.3. 動的コマンドグループ

```
namespace ext::oneapi::experimental {
class dynamic_command_group {
public:
  dynamic_command_group(
      command_graph<graph_state::modifiable> &graph,
      const std::vector<std::function<void(handler &)>>& cgfList);

  size_t get_active_index() const;
  void set_active_index(size_t cgfIndex);
};
```

動的コマンドグループは、グラフのノードとして追加できます。これらは、グラフがファイナライズされた後にノードのコマンドグループ機能を更新できるメカニズムを提供します。動的コマンドグループには常にアクティブとして設定されるコマンドグループ関数が 1 つあります。これは、グラフが実行可能状態 `command_graph` に確定されたときにノードに対して実行されるコマンドグループであり、`cgfList` 内の他のコマンドグループ関数は無視されます。動的コマンドグループ・オブジェクトの新しいアクティブ・インデックスを選択し、実行可能ファイル `command_graph` で `update(node& node)` メソッドを呼び出すことによって、実行可能ファイル `command_graph` ノードを `cgfList` 内の別のカーネルに更新できます。

dynamic_command_group クラスは共通の参照セマンティクスを提供します。

コマンドグループの更新に関する詳細は、実行可能グラフの更新を参照してください。

制限事項

動的コマンドグループには、次の操作のみを含めることができます:

- カーネル操作
- ホストタスク

他の操作を含む関数を使用して動的コマンドグループを構築しようとすると、エラーが発生します。

動的コマンドグループ内のすべてのコマンドグループ関数は、同じ依存関係を持つ必要があります。アクティブに設定されるとグラフトポロジーが変更されるコマンドグループ関数を、動的コマンドグループに含めることはできません。これは、handler.depends_on() へのすべての呼び出しが、動的コマンドグループ内のすべてのコマンドグループ関数に対して同一でなければならないことを意味します。バッファーアクセサーによって作成された依存関係は、すべてのコマンドグループ関数にわたって同一のノード依存関係も作成しなければなりません。

dynamic_command_group クラスのメンバー関数

```
dynamic_command_group(
command_graph<graph_state::modifiable> &graph,
const std::vector<std::function<void(handler &)>>& cgfList);
```

効果: graph にノードとして追加できる動的コマンドグループ・オブジェクトを構築します。cgfList は、この動的コマンドグループに対してアクティブにできるコマンドグループ関数のリストです。デフォルトでは、インデックス 0 のコマンドグループ関数がアクティブになります。
スロー:

- グラフが property::graph::assume_buffer_outlives_graph プロパティーで作成されておらず、dynamic_command_group がバッファーを使用するコマンドグループ関数で作成されている場合、エラーコードが invalid の同期例外が発生します。詳細は、Assume-Buffer-Outlives-Graph プロパティーを参照してください。
- dynamic_command_group がカーネル実行またはホストタスクではないコマンドグループ関数で作成された場合、エラーコードが invalid の例外が発生します。
- cgfList が空の場合、エラーコードが invalid の例外が発生します。
- cgfList 内のすべてのコマンドグループのタイプが一致しない場合、エラーコードが invalid の例外が発生します。

```
size_t get_active_index() const;
```

リターン: この dynamic_command_group 内の現在アクティブなコマンドグループ関数のインデックス。

```
void set_active_index(size_t cgfIndex);
```

効果: インデックス `cgfIndex` を持つコマンドグループ関数をアクティブに設定します。`dynamic_command_group` 内のコマンドグループ関数のインデックスは、`dynamic_command_group` コンストラクターに渡されたときの `cgfList` ベクトル内のインデックスと同一です。この変更は、この `dynamic_command_group` を含む変更可能なグラフに直ちに反映されます。変更可能なグラフから作成された実行可能グラフには、変更されたノードを渡して `command_graph::update()` が呼び出されるか、変更可能なグラフから新しい実行可能グラフが確定されるまで、新しい値は反映されません。`cgfIndex` を現在アクティブなコマンドグループ関数のインデックスに設定しても何も起こりません。

スロー: `cgfIndex` が有効なインデックスでない場合、エラーコードが `invalid` の例外が発生します。

8.3.4. プロパティーに依存

```
namespace sycl::ext::oneapi::experimental::property::node {
class depends_on {
  public:
    template<typename...NodeTN>
    depends_on(NodeTN... nodes);
};
}
```

`command_graph` にノードを明示的に追加する API には、`property_list` パラメーターが含まれています。この拡張機能は、ここに渡される `depends_on` プロパティーを定義します。`depends_on` は、作成されたノードが依存する `node` オブジェクトを定義し、それによってエッジを形成します。

8.3.5. すべてのリーフに依存するプロパティー

```
namespace sycl::ext::oneapi::experimental::property::node {
class depends_on_all_leaves {
  public:
    depends_on_all_leaves() = default;
};
}
```

`command_graph` にノードを明示的に追加する API には、`property_list` パラメーターが含まれています。この拡張機能は、ここに渡される `depends_on_all_leaves` プロパティーを定義します。`depends_on_all_leaves` は、グラフの現在のすべてのリーフを依存関係として追加するショートカットを提供します。

8.4. グラフ

```
namespace sycl::ext::oneapi::experimental {
// グラフの状態
enum class graph_state {
  modifiable,
  executable
};

// グラフを表す新しいオブジェクト
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(const queue& syclQueue,
                const property_list& propList = {});

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

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

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

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

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

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

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

```
  print_graph(std::string path, bool verbose = false) const;

  std::vector<node> get_nodes() const;
  std::vector<node> get_root_nodes() const;
};

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

    void update(node& node);
    void update(const std::vector<node>& nodes);
    void update(const command_graph<graph_state::modifiable>& graph);

    size_t get_required_mem_size() const noexcept;
};

}  // namespace sycl::ext::oneapi::experimental
```

この拡張機能は、他の SYCL ランタイム・オブジェクトの共通参照セマンティクスに従う新しい `command_graph` オブジェクトを追加します。

`command_graph` はノードの有向非巡回グラフを表します。各ノードは特定のデバイスまたはサブグラフの単一のコマンドを表します。グラフの実行は、そのすべてのノードが完了したときに終了します。

`command_graph` は、キューの送信を記録するか、明示的にノードを追加することによって構築され、ユーザーがグラフが完了したと判断すると、グラフ・インスタンスは実行可能なバリアントに確定され、それ以降ノードを追加できなくなります。ランタイムはグラフ構造に基づいて最適化できるため、ファイナライズは計算コストの高い操作となる可能性があります。ファイナライズ後、グラフはオーバーヘッドを削減しながらキューに 1 回以上送信して実行できます。

`command_graph` は、インオーダー・キューとアウトオブオーダー・キューの両方に送信できます。グラフと同じキューに送信された他のコマンドグループ間の依存関係はすべて反映されます。ただし、キューのインオーダー・プロパティーとアウトオブオーダー・プロパティーは、グラフ内のノードの実行には影響しません (たとえば、依存関係エッジのないグラフノードは、インオーダー・キューを使用している場合でも、アウトオブオーダーで実行される場合があります)。キューのプロパティーがグラフにどのように影響するかは、「キューのプロパティー」のセクションを参照してください。

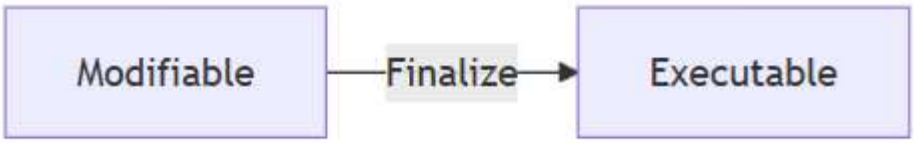
8.4.1. グラフの状態

`command_graph` オブジェクトのインスタンスは、次の 2 つの状態のどちらかになります:

- **変更可能** - グラフは構築中であり、新しいノードが追加される可能性があります。
- **実行可能** - グラフのトポロジーはファイナライズ後に確定され、グラフは実行のため送信する準備が整っています。

`command_graph` オブジェクトは*変更可能*な状態で構築され、ユーザーが `command_graph::finalize()` を呼び出してグラフの新しい*実行可能*なインスタンスを作成することで実行できるようになります。実行可能グラフを変更可能グラフに変換することはできません。変更可能な状態でグラフをファイナライズした後、ユーザーはノードを追加し、再度ファイナライズして後続の実行可能なグラフを作成できます。実行可能なグラフが破棄されると、キューに入れられたグラフの送信が完了した場合にのみ、リソースが解放されます。`command_graph` オブジェクトの状態は、状態のテンプレート化によって明示的に指定され、クラスが強く型付けされます。デフォルトのテンプレート引数は `graph_state::modifiable` であり、構築時のコードの冗長性がなくなります。

グラフの状態ダイアグラム



8.4.2. 実行可能グラフの更新

実行可能状態のグラフでは、グラフ*更新*と呼ばれる概念によってノードの構成を変更できます。これにより、送信間でグラフノードのパラメーターのみが変更される場合、ユーザーが新しい実行可能グラフを再構築してファイナライズする必要がなくなります。

グラフの更新は、実行中の同じグラフの実行後にスケジュールされ、同じグラフの以前の送信には影響しません。ユーザーは、グラフを更新する前に、以前のグラフの送信を待機する必要はありません。

実行可能なグラフを更新するには、ファイナライズ中にグラフが作成される時に `property::graph::updatable` プロパティーが設定されている必要があります。そうでない場合、ユーザーが実行可能グラフを更新しようとすると例外がスローされます。この保証により、バックエンドは可能であればより最適化された実装を提供できるようになります。

サポートされる機能

現在グラフ内で更新できるノードのタイプは次のとおりです:

- カーネル実行
- ホストタスク

グラフの更新に使用できる 2 つの異なる API があります:

- 個別のノード更新により、コマンドグラフの個々のノードを更新できます。
- グラフ全体の更新により、別のグラフを参照として使用することで、グラフ全体を同時に更新できます。

次の表は、更新の実行に使用される API に応じて変更できる、各サポート対象ノードタイプの側面を示しています。

表 6. サポート対象ノードタイプに対するグラフ更新機能

ノードタイプ	個々のノードを更新	グラフ全体の更新
`node_type::kernel`	• カーネル関数 • カーネル・パラメーター • ND-range	• カーネル・パラメーター • ND-range
`node_type::host_task`	• ホストタスク関数	• ホストタスク関数

個々のノードを更新

実行可能グラフの個々のノードを直接更新できます。更新が必要なノードの属性や node_type に応じて、異なる API を使用する必要があります:

パラメーターの更新

サポートされるノードタイプ: カーネル

実行可能状態にあるグラフの個々のノードへのパラメーターは、動的パラメーターを使用してグラフ実行の間に更新できます。変更可能な状態グラフとパラメーターの初期値を持つ `dynamic_parameter` オブジェクトが作成されます。動的パラメーターは、`set_arg()/set_args()` の呼び出しに渡して、そのグラフ内のノードに登録できます。

パラメーターの更新は、`dynamic_parameter` インスタンスを使用して `dynamic_parameter::update()` を呼び出すことで、`dynamic_parameter` が登録されているノードのすべてのパラメーターを更新することによって行われます。更新は、`dynamic_parameter` と同じパラメーター値を使用していても、登録されていないノードには影響しません。

ファイナライズ時にグラフの構造が固定されるため、ノード上のパラメーターを更新しても、ノード間で既に定義されている依存関係は変化しません。この変更によってエッジが自動的に作成または削除されることはないため、ノードのバッファー・パラメーターを更新するときは、この点に注意してください。バッファー・パラメーターを更新しても、実行時にグラフの動作が変わらないように注意する必要があります。

例えば、同じバッファーへの依存関係があるエッジで接続された 2 つのノード (NodeA と NodeB) がある場合、両方のノードでこのバッファー・パラメーターを新しい値に更新する必要があります。これにより、正しいデータ依存関係が維持され、予期しない動作を防止できます。これには、バッファーをパラメーターとして使用するすべてのノードにバッファーの 1 つの動的パラメーターを登録します。その後、単一の `dynamic_parameter::update()` 呼び出しによってグラフデータの依存関係が維持されます。

実行レンジの更新

サポートされるノードタイプ: カーネル

更新可能な別の設定として、カーネルの実行レンジがあります。これは `node::update_nd_range()` または `node::update_range()` を通じて設定でき、事前登録は不要です。

ノードの実行レンジを更新する別の方法は、次のセクションで説明するコマンドグループの更新と同時に実行することです。このメカニズムを使用することで、`node::update_nd_range()` / `node::update_range()` が同じ次元でのみ実行レンジを更新するという制限が解除されます。更新がコマンドグループの変更に関連付けられることは、更新されたカーネルコードが異なる次元で動作することが定義される可能性があることを意味します。

コマンドグループの更新

サポートされるノードタイプ: カーネル、ホストタスク

カーネルノードのコマンドグループは、動的コマンドグループを使用して更新できます。動的コマンドグループにより、カーネルノードのコマンドグループ関数を別の関数に置き換えることが可能になります。これにより、カーネル関数および/またはカーネル実行レンジの更新が可能になります。

コマンドグループの更新は、`dynamic_command_group` クラスのインスタンスを作成することで行われます。動的なコマンドグループは、変更可能な状態グラフと、可能なコマンドグループ関数のリストによって作成されます。動的コマンドグループ内のコマンドグループ関数は、メンバー関数 `dynamic_command_group::set_active_index()` を使用してアクティブに設定できます。

動的コマンドグループは動的パラメーターと互換性があります。つまり、動的コマンドグループの一部であるコマンドグループ関数で動的パラメーターを使用できます。このような動的パラメーターの更新は、アクティブ化されるとコマンドグループ関数に反映されます。

実行レンジはコマンドグループに関連付けられているため、動的コマンドグループを使用するノードのレンジを更新すると、現在アクティブなコマンドグループの実行レンジも更新されることに注意してください。動的コマンドグループが別のノードによって共有されている場合、その動的コマンドグループを共有している他のノードの実行レンジも更新されます。以前に `node::update_range()` または `node::update_nd_range()` で実行レンジを更新したコマンドグループに対し、`set_active_index` を使用してコマンドグループをアクティブ化しても実行レンジは元の値にリセットされず、代わりに最後に更新された値が使用されます。

コミットされた更新

上記の方法でノードを更新すると、変更可能なコマンドグラフ内のノードに対してすぐに反映されます。ただし、実行可能な状態にあるグラフの更新をコミットするには、更新されたノードを `command_graph<graph_state::executable>::update(node& node)` または `command_graph<graph_state::executable>::update(const std::vector<node>& nodes)` に渡す必要があります。

グラフ全体の更新

実行可能状態のグラフでは、`command_graph<graph_state::executable>::update(graph)` メソッドを使用してすべてのノードを更新できます。このメソッドは、変更可能な状態のソースグラフを受け取り、ターゲットの実行可能状態グラフ内のノードを更新して、ソースグラフ内のノードの変更を反映します。更新される特性については、実行可能グラフの更新のセクションで詳しく説明します。

更新のソースグラフとターゲットグラフの両方が次の条件を満たしている必要があります:

- 両方のグラフは同じデバイスとコンテキストで作成されている必要があります。
- どちらのグラフにも、`node_type::async_malloc` または `node_type::async_free` タイプのノードは存在しません。
- 両方のグラフはトポロジー的に同一である必要があります。次の場合、グラフは位相的に同一であるとみなされます:
    - 両方のグラフのノードとエッジの数は同じである必要があります。
    - 内部エッジは各グラフ内の対応するノード間に存在する必要があります。
    - 2 つのグラフでノードは同じ順序で追加する必要があります。ノードは、`command_graph::add` を介して追加できます。また、記録されたキューの場合は、`queue::submit` またはキューのショートカット関数を介して追加できます。
    - 各グラフ内の対応するノードは同じ `node_type` である必要があります。
    - タイプ `node_type::kernel` の対応するノードには、同じタイプのカーネルが必要です:
        - カーネルがラムダとして定義されている場合、ラムダは同一である必要があります。
        - カーネルが名前付き関数オブジェクトとして定義されている場合、カーネルクラスは同一である必要があります。
        - カーネルが単純な関数として定義されている場合、関数は同一である必要があります。
    - 2 つのグラフ内の各ノードに対するエッジの依存関係は、各エッジを作成する際に同じ API 呼び出しを使用し、同じ順序で作成されなければなりません。2 つのグラフ構築 API のそれぞれについて、グラフ内でエッジがどのように定義されるかの詳細な定義については、用語のセクションを参照してください。

トポロジカル同一性の条件を満たさないソースグラフ、またはターゲットグラフで全てのグラフ更新を使用しようとすると、ランタイムがソースグラフとターゲットグラフのノードをペアリングできなくなる可能性があるため、未定義の動作を引き起こします。

グラフ全体の更新で動的パラメーターを含むノードを使用するのは有効です。動的パラメーターを含むノードがグラフ全体の更新 API を通じて更新されると、動的パラメーターに対する以前の更新が新しいグラフに反映されます。

8.4.3. グラフ所有メモリー割り当て

グラフ内のコマンドに関連付けられたメモリーのメモリー割り当てが、グラフが所有および管理することが望ましい場合があります。これは、sycl_ext_oneapi_async_memory_alloc 拡張機能の async_<malloc/malloc_from_pool> コマンドと async_free コマンドを使用することで可能です。これらのコマンドは、キュー記録または明示的なグラフ作成を介してグラフに追加できます。これにより、特定の command_graph によって所有および管理される割り当てが作成され、そのライフタイムはグラフのライフタイムに関連付けられます。

割り当てノードから返されるポインターは、通常の USM ポインターと同じように他のグラフノードで使用できます。

API の使い方

malloc ノードと free ノードは、コマンドグループ内の async_<malloc/malloc_from_pool/free> free 関数を使用して、明示的およびキュー記録グラフ API の両方を介してグラフに追加できます。

```
void* Ptr = nullptr;
size_t AllocSize = 1024;

// 明示的なグラフ作成
Graph.add([&](handler &CGH){
  Ptr = async_malloc(CGH, usm::alloc::device, AllocSize);
});

Graph.add([&](handler &CGH){
  async_free(CGH, Ptr);
});

// キューの記録
Graph.begin_recording(Queue);
Queue.submit([&](handler &CGH){
  Ptr = async_malloc(CGH, usm::alloc::device, AllocSize);
});
Queue.submit([&](handler &CGH){
  async_free(CGH, Ptr);
});
Graph.end_recording(Queue);
```

キューを受け取る `async_*` 関数は、キューの記録でも使用できます。特に、インオーダー・キューを記録する場合は、SYCL イベントが返されないため、依存関係を指定できます。

```
void* Ptr = nullptr;
size_t AllocSize = 1024;
queue Queue {syclContext, syclDevice, {property::queue::in_order{}}};

Graph.begin_recording(Queue);
Ptr = async_malloc(Queue, usm::alloc::device, AllocSize);
async_free(Queue, Ptr);
Graph.end_recording(Queue);
```

サポートされる機能

現在はデバイス割り当てのみがサポートされています。グラフに他のタイプの割り当てを追加しようとすると、エラーコードが `invalid` の同期エラーがスローされます。

制限事項

非同期 `malloc` または `free` ノードを含むグラフには、次の制限が適用されます:

- 特定の変更可能なグラフ (変更可能なグラフをファイナライズすることによって作成される) に対して同時に存在できる実行可能なグラフ・インスタンスは 1 つだけであり、そのインスタンスのすべてのコピー (`command_graph` クラスの共通参照セマンティクスによって作成される) は、グラフを再度ファイナライズする前に破棄する必要があります。
- グラフは別のグラフのサブグラフとして使用できません。
- グラフメモリー割り当てノードは更新できず、これらのノードを含むグラフはグラフ全体の更新によって更新できません。

上記のいずれかの操作を実行しようとすると、エラーコードが `invalid` の同期エラーがスローされます。

ライフタイムの割り当て

グラフ所有の割り当てのライフタイムは、グラフ自体のライフタイムに関連付けられています。

グラフ割り当てノードから返されたポインターは、割り当てられたグラフ内でのみ使用できます。これらの割り当てを使用するノードは、割り当てノードの後、その割り当ての空きノードの前に配置する必要があります。これを行わないと、未定義の動作となります。

これらのポインターを所有するグラフの外で使用することは無効であり、使用すると未定義の動作を引き起こします。

動作

グラフ内の `async_malloc` および `async_free` のセマンティクスは、`sycl_ext_oneapi_async_memory_alloc` 拡張で説明されている非グラフ使用法とは異なります。

- グラフメモリーの割り当ては、デフォルトまたはユーザー指定のメモリープールから直接行われるわけではありません。非同期 `malloc/free` ノードを含む各グラフは、割り当てが行われる独自のメモリープールを維持します。`async_<malloc/malloc_from_pool>` の呼び出しで提供されるデフォルトまたはユーザー指定のメモリープールの以下のプロパティーは、関連するグラフ割り当てで尊重され、その他のプロパティーはすべて無視されます:
  - プールを作成するときに指定された割り当てタイプは `usm::alloc::<host/device/shared>` であり、サポートされている機能のセクションの制限事項に従います。
  - `property::memory_pool::zero_init` - このプロパティーを持つ割り当てを含む変更可能なグラフが実行可能グラフにファイナライズされるたびに、割り当てられたメモリーはゼロで初期化されます (このグラフのファイナライズに関する制限については、このセクションを参照してください)。グラフは、その後の実行前または実行中に、再度ゼロで初期化されることはありません。アプリケーションでそれが必要な場合は、これを実行するのに適切なコマンドをグラフに追加するのはユーザーの責任です。
- グラフ内の `node_type::async_malloc` ノードは、指定されたサイズの割り当てへのポインターを返します。このポインターは、他の USM ポインターと同じように、ノードの後に順序付けられた他のグラフ ノードで使用できます。
- グラフ内の `node_type::async_free` ノードは、特定の割り当てが使用されなくなったことを示します。これらは、関連する割り当てノードの後に配置する必要があります。`async_free` に渡されるポインターは、同じグラフ内の非同期 `malloc` ノードによって割り当てられたメモリー割り当てアドレスである必要があります。これらの前提条件に違反すると、動作は未定義となります。
- 特定のグラフ割り当てを使用する他のノードは、依存関係を通じて順序付けられる必要があります。それらのノードは、特定の割り当てにおいて、割り当てノードの後の空きノードの前に配置されます。依存関係が正しいことを確認するのはユーザーの責任です。`async_free` ノードを介して解放された後に、順序付けられたグラフコマンドでポインターを使用すると未定義の動作が発生します。

実行可能グラフのグラフ割り当てに必要なメモリーの合計量は、`command_graph::get_required_mem_size()` メンバー関数を使用して照会できます。

8.4.4. グラフのプロパティー

No-Cycle-Check プロパティー

```
namespace sycl::ext::oneapi::experimental::property::graph {
class no_cycle_check {
  public:
    no_cycle_check() = default;
};
```

```
}
```

`property::graph::no_cycle_check` プロパティーは、新しく追加された依存関係が特定の `command_graph` でサイクルを引き起こすチェックを無効にし、構築時にプロパティー・ リスト・パラメーターを介して `command_graph` に渡すことができます。その結果、関数が循環依存関係を作成しようとしてもエラーは報告されません。したがって、このプロパティーが設定されている場合、非循環の実行グラフを作成するのはユーザーの責任です。`command_graph` 内でサイクルを作成すると、その `command_graph` は未定義状態になります。この状態の `command_graph` に対してさらに操作を行うと、未定義の動作が発生します。

Assume-Buffer-Outlives-Graph プロパティー

```
namespace sycl::ext::oneapi::experimental::property::graph {
class assume_buffer_outlives_graph {
  public:
    assume_buffer_outlives_graph() = default;
};
}
```

`property::graph::assume_buffer_outlives_graph` プロパティーは、`command_graph` のバッファー使用に関する制限を無効にし、構築時にプロパティー・リスト・パラメーターを介して `command_graph` に渡すことができます。このプロパティーは、グラフ内で使用されるすべてのバッファーが、グラフのライフタイム中はホスト上で存続し続けることをユーザーが確約することを表します。このプロパティーを使用して構築された `command_graph` のライフタイム中にそのバッファーを破棄すると、未定義の動作が発生します。

更新可能なプロパティー

```
namespace sycl::ext::oneapi::experimental::property::graph {
class updatable {
  public:
    updatable() = default;
};
}
```

`property::graph::updatable` プロパティーは、変更可能な `command_graph` の終了時に渡されたときに、`command_graph` を更新できるようにします。詳細については、実行可能グラフの更新に関するセクションを参照してください。

8.4.5. プロファイル有効化プロパティー

```
namespace sycl::ext::oneapi::experimental::property::graph {
class enable_profiling {
  public:
```

```
    enable_profiling() = default;
};
}
```

property::graph::enable_profiling プロパティーは、実行可能グラフの送信から返されるプロファイル・イベントを有効にします。このプロパティーを渡すと、一部の最適化が無効になります。その結果、プロファイルを有効にしてファイナライズしたグラフの実行時間は、プロファイルを行わないグラフの実行時間よりも長くなります。プロパティーなしで作成されたグラフ送信からイベントをプロファイルしようとすると、エラーがスローされます。

8.4.6. グラフメンバー関数

command_graph クラスのコンストラクター

```
command_graph(const context& syclContext,
              const device& syclDevice,
              const property_list& propList = {});
```

効果: コンテキスト syclContext とデバイス syclDevice の変更可能な状態の SYCL command_graph オブジェクトを作成します。構築された SYCL command_graph には、property_list のインスタンスを介して 0 個以上のプロパティーを供給できます。syclContext はグラフの不変の特性です。

条件: このコンストラクターは、command_graph の状態が graph_state::modifiable の場合にのみ使用できます。

前提条件: 有効な command_graph コンストラクターのプロパティーは、グラフ・プロパティーのセクションにリストされています。

スロー:

- syclDevice が syclContext に関連付けられていない場合、エラーコードが invalid の同期例外が発生します。
- syclDevice がこの拡張機能をサポートされていないと報告した場合、エラーコードが invalid の同期例外が発生します。

```
command_graph(const queue& syclQueue,
              const property_list& propList = {});
```

効果: syclQueue がグラフの不変の特性となるデバイスとコンテキストのみを提供する、簡略化されたコンストラクター形式。property_list のインスタンスを介して、構築された SYCL command_graph に 0 個以上のプロパティーを供給できます。キューのその他のプロパティーは、グラフ作成では無視されます。キューのプロパティーが command_graph オブジェクトとどのように相互作用するかは、キューのプロパティーセクションを参照してください。

条件: このコンストラクターは、`command_graph` の状態が `graph_state::modifiable` の場合にのみ使用できます。

前提条件: 有効な `command_graph` コンストラクターのプロパティーは、グラフ・プロパティーのセクションにリストされています。

スロー: `syclQueue` に関連付けられたデバイスが、この拡張機能をサポートされていないと報告した場合、エラーコードが `invalid` の同期例外が発生します。

command_graph クラスのメンバー関数

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

効果: これにより、コマンドを含まない空のノードが作成されます。その目的は、グラフ内のノードグループ間に接続ポイントを作成することで、これによりエッジの数を大幅に削減できます (*O(n)* と *O(n^2)*)。

条件: このメンバー関数は、`command_graph` の状態が `graph_state::modifiable` の場合にのみ使用できます。

リターン: グラフに追加された空のノード。

スロー: キューがグラフにコマンドを記録している場合、エラーコードが `invalid` の同期例外が発生します。

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

効果: `cgf` コマンドグループ関数は、他の拡張機能との相互作用で明示的に記載されていない限り、`queue::submit` に渡されるコマンドグループ関数とほぼ同じように動作します。関数内のコードは、SYCL コマンド (カーネルや明示的なメモリーコピー操作など) を除き、関数が `command_graph::add` に戻る前に同期的に実行されます。これらのコマンドはグラフにキャプチャーされ、グラフがキューに送信されたときに非同期的に実行されます。cgf の要件は、エッジを形成するグラフ内の依存ノードを識別するために使用されます。

条件: このメンバー関数は、`command_graph` の状態が `graph_state::modifiable` の場合にのみ使用できます。

リターン: グラフに追加されたコマンドグループ関数オブジェクト・ノード。

スロー:

- キューがグラフにコマンドを記録している場合、エラーコードが `invalid` の同期例外が発生します。

- グラフが property::graph::assume_buffer_outlives_graph プロパティーで作成されておらず、このコマンドがバッファーを使用している場合、エラーコードが invalid の同期例外が発生します。詳細は、Assume-Buffer-Outlives-Graph プロパティーを参照してください。
- コマンドグループに含まれるコマンドのタイプが async_malloc であり、関連付けられているメモリープールの usm::alloc タイプが usm::alloc::device でない場合、エラーコードが invalid の例外が発生します。
- コマンドグループに含まれるコマンドタイプがカーネル実行ではなく、dynamic_parameter が cgf 内に登録されている場合、エラーコードが invalid の例外が発生します。

```
node add(dynamic_command_group& dynamicCG, const property_list& propList = {});
```

効果: 動的コマンドグループ dynamicCG をノードとしてグラフに追加し、dynamicCG 内のアクティブなコマンドグループ関数を、このグラフノード以降の実行可能ファイルとして設定します。dynamicCG の現在アクティブなコマンドグループ関数は、グラフがキューに送信されたときに非同期に実行されます。このコマンドグループ関数の要件は、エッジを形成するグラフ内の依存ノードを識別するために使用されます。dynamicCG の他のコマンドグループ関数はグラフにキャプチャーされますが、アクティブに設定されていない限りグラフの送信では実行されません。

条件: このメンバー関数は、command_graph の状態が graph_state::modifiable の場合にのみ使用できます。

リターン: グラフに追加された動的コマンドグループのオブジェクト・ノード。

スロー:

- キューがグラフにコマンドを記録している場合、エラーコードが invalid の同期例外が発生します。
- グラフが dynamicCG の構築に使用されたグラフと一致しない場合、エラーコードが invalid の同期例外が発生します。
- cgfList 内のコマンドグループ関数に、相互に互換性のないイベントまたはアクセサー依存関係があり、アクティブに設定すると異なるグラフトポロジーが生成される場合、エラーコードが invalid の例外が発生します。

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

効果: 事前発生関係を表す 2 つのノード間の依存関係を作成します。ノード dest は src に依存します。

条件: このメンバー関数は、command_graph の状態が graph_state::modifiable の場合にのみ使用できます。

スロー:

- キューがグラフ・オブジェクトにコマンドを記録している場合、エラーコードが invalid の同期例外が発生します。

- `src` または `dest` がグラフ・オブジェクトに割り当てられた有効なノードでない場合、エラーコードが `invalid` の同期例外が発生します。
- `src` と `dest` が同じノードの場合、エラーコードが `invalid` の同期例外が発生します。
- 結果として生じる依存関係が循環につながる場合、エラーコードが `invalid` の同期例外が発生します。`property::graph::no_cycle_check` が設定されている場合は、このエラーは省略されます。

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

効果: 固定トポロジーを持つ実行可能状態の新しいグラフを作成する同期操作。このグラフに関連付けられたコンテキストを共有する任意のキューで実行するために送信できます。同じ変更可能なグラフから作成された別の実行可能グラフがすでに存在する場合、そのグラフにグラフ所有のメモリー割り当てが含まれていない限り、このメンバー関数を呼び出して新しい実行可能グラフを作成することは有効です。この関数を呼び出した後も、変更可能なグラフ・インスタンスに新しいノードを追加し続けるのは有効です。コマンドが記録されていない空のグラフ・インスタンスをファイナライズするのは有効です。

条件: このメンバー関数は、`command_graph` の状態が `graph_state::modifiable` の場合にのみ使用できます。

リターン: キューに送信可能な新しい実行可能グラフ・オブジェクト。

スロー:

- グラフにデバイスでサポートされていないコマンドが含まれている場合、エラーコード `feature_not_supported` の同期例外が発生します。
- グラフにグラフ所有のメモリー割り当てが含まれており、この変更可能なグラフから作成された実行可能グラフのインスタンスがまだ存続している場合、エラーコード `invalid` の例外が発生します。

```
void print_graph(std::string path, bool verbose = false) const;
```

効果: 指定された `path` にグラフの DOT 形式を書き込む同期操作。デフォルトでは、グラフトポロジー、ノードタイプ、ノード ID、カーネル名が含まれます。`verbose` を `true` に設定すると、カーネル引数、コピー元、コピー先アドレスなど、各ノード タイプに関する詳細な情報が書き込まれます。現時点では、DOT 形式のみがサポートされています。出力ファイルの名前はこの拡張子と一致する必要があります。つまり "`<filename>.dot`" です。

スロー: パスが無効であるか、ファイル拡張子がサポートされていない、または書き込み操作が失敗した場合は、エラーコード invalid の同期例外が発生します。

```
std::vector<node> get_nodes() const;
```

リターン: グラフ内にあるすべてのノードを追加された順にリストします。

```
std::vector<node> get_root_nodes() const;
```

リターン: グラフ内の依存関係のないすべてのノードのリスト。

```
size_t get_required_mem_size() const noexcept;
```

条件: このメンバー関数は、`command_graph` の状態が `graph_state::executable` の場合にのみ使用できます。

リターン: このグラフ内のグラフ所有メモリー割り当てに必要なメモリーの合計サイズ (バイト単位)。

command_graph クラスのメンバー関数のグラフ更新

```
void update(node& node);
```

効果: `node` に対応する実行可能グラフノードを更新します。ノードはカーネル実行ノードである必要があります。実行可能グラフ内でノードのコマンドグループ関数が更新され、`node` の現在の値が反映されます。これには、カーネル関数、カーネル nd-range、およびカーネル・パラメーターが含まれます。これらの値を更新してもグラフの構造は変更されません。実装は、バックエンドが必要とする場合、この呼び出し中に同じグラフの進行中のすべての実行に対してブロッキング待機を実行する可能性があります。

条件: このメンバー関数は、`command_graph` の状態が `graph_state::executable` の場合にのみ使用できます。

スロー:

- 実行可能グラフの作成時に `property::graph::updatable` が設定されていない場合、エラーコードが `invalid` の同期例外が発生します。
- `node` がグラフに属していない場合、エラーコードが `invalid` の例外が発生します。
- ノードのタイプが `node_type::async_malloc` または `node_type::async_free` のいずれかである場合、エラーコード invalid の例外が発生します。
- その他の例外がスローされた場合、グラフノードの状態は未定義になります。

```
void update(const std::vector<node>& nodes);
```

効果: `nodes` に含まれるノードに対応するすべての実行可能なグラフノードを更新します。すべてのノードはカーネルノードである必要があります。各ノードのコマンドグループ関数は、実行可能グラフ内で更新され、`nodes` の現在の値を反映します。これには、カーネル関数、カーネル nd-range、およびカーネル・パラメーターが含まれます。これらの値を更新してもグラフの構造は変更されません。実装は、バックエンドが必要とする場合、この呼び出し中に同じグラフの進行中のすべての実行に対してブロッキング待機を実行する可能性があります。

条件: このメンバー関数は、`command_graph` の状態が `graph_state::executable` の場合にのみ使用できます。

スロー:

- 実行可能グラフの作成時に `property::graph::updatable` が設定されていない場合、エラーコードが `invalid` の同期例外が発生します。
- ノード内のいずれかのノードがグラフの一部でない場合、エラーコードが `invalid` の例外が発生します。
- `nodes` 内の任意のノードのタイプが `node_type::async_malloc` または `node_type::async_free` のいずれかである場合、エラーコード invalid の例外が発生します。
- その他の例外がスローされた場合、グラフノードの状態は未定義になります。

```
void update(const command_graph<graph_state::modifiable>& source);
```

効果: 変更可能な状態にあるトポロジー的に同一のグラフ `source` からのパラメーターを使用して、ターゲットグラフ内のすべてのノードを更新します。位相的に同一のグラフを構成する完全な定義は、グラフ全体の更新セクションを参照してください。これらのトポロジー要件のいずれかに違反すると、未定義の動作となります。更新される実行可能グラフの特性については、実行可能グラフの更新のセクションで詳しく説明します。`source` 内のノードのすべてのパラメーターが、更新前の実行可能グラフの引数と同じになるように実行可能グラフを更新しても、エラーにはなりません。実装は、バックエンドが必要とする場合、この呼び出し中に同じグラフの進行中のすべての実行に対してブロッキング待機を実行する可能性があります。この関数は、グラフが更新可能なプロパティーで作成された場合にのみ呼び出すことができます。

条件: このメンバー関数は、`command_graph` の状態が `graph_state::executable` の場合にのみ使用できます。

スロー:

- ソースに以下のいずれでもないタイプのノードが含まれている場合、エラーコード `invalid` の同期例外が発生ます:
  - `node_type::empty`
  - `node_type::ext_oneapi_barrier`
  - `node_type::kernel`
- ソースに関連付けられたコンテキストまたはデバイスが、更新される `command_graph` と一致しない場合、エラーコードが `invalid` の同期例外が発生します。
- 実行可能グラフの作成時に `property::graph::updatable` が設定されていない場合、エラーコードが `invalid` の同期例外が発生します。
- グラフにグラフ所有のメモリー割り当てが含まれている場合、エラーコードが `invalid` の同期例外が発生します。
- その他の例外がスローされた場合、グラフノードの状態は未定義になります。

command_graph クラスのメンバー関数のキューを記録

```
void begin_recording(queue& recordingQueue,
                     const property_list& propList = {})
```

効果: 同期的に、recordingQueue の状態を queue_state::recording 状態に変更します。recordingQueue がすでに queue_state::recording 状態にある場合、この操作はエラーになります。
前提条件: propList は、プロパティーを渡すオプションのパラメーターです。command_graph クラスのプロパティーは、グラフ・プロパティーで定義されます。
スロー:

- recordingQueue がすでにグラフに記録中の場合、エラーコードが invalid の同期例外が発生します。
- グラフの作成時に使用されたデバイスおよびコンテキストとは異なるデバイスやコンテキストに recordingQueue が関連付けられている場合、エラーコードが invalid の同期例外が発生します。

```
void begin_recording(const std::vector<queue>& recordingQueues,
                     const property_list& propList = {})
```

効果: recordingQueues 内の各キューの状態を同期的に queue_state::recording 状態に変更し、グラフ・インスタンスへのコマンドの記録を開始します。recordingQueue がすでに queue_state::recording 状態にある場合、この操作はエラーになります。

前提条件: propList は、プロパティーを渡すオプションのパラメーターです。command_graph クラスのプロパティーは、グラフ・プロパティーで定義されます。

スロー:

- recordingQueue のいずれかのキューがすでにグラフに記録中の場合、エラーコードが invalid の同期例外が発生します。
- グラフの作成時に使用されたデバイスおよびコンテキストとは異なるデバイスやコンテキストにいずれかの recordingQueue が関連付けられている場合、エラーコードが invalid の同期例外が発生します。

```
void end_recording()
```

効果: グラフに記録しているすべてのキューの記録を同期的に終了し、状態を queue_state::executing に設定します。この操作は、グラフ内ですでに queue_state::executing 状態にあるキューに対しては何も行いません。

```
void end_recording(queue& recordingQueue)
```

効果: 同期的に、recordingQueue の状態を queue_state::executing 状態に変更します。recordingQueue がすでに queue_state::executing 状態にある場合、この操作は何も行いません。

スロー: recordingQueue が別のグラフに記録している場合、エラーコードが invalid の同期例外が発生します。

```
void end_recording(const std::vector<queue>& recordingQueues)
```

効果: 同期的に、recordingQueue の各キューの状態を queue_state::executing 状態に変更します。この操作は、すでに queue_state::executing 状態にある recordingQueues 内のキューに対しては何も行いません。

スロー: recordingQueue のいずれかのキューが別のグラフに記録中の場合、エラーコードが invalid の同期例外が発生します。

8.5. キュークラスの変更

```
namespace sycl {
namespace ext::oneapi::experimental {
enum class queue_state {
  executing,
  recording
};

} // namespace ext::oneapi::experimental

// sycl::queue クラスに追加された新しいメソッド
using namespace ext::oneapi::experimental;
class queue {
public:

  ext::oneapi::experimental::queue_state
  ext_oneapi_get_state() const;

  ext::oneapi::experimental::command_graph<graph_state::modifiable>
  ext_oneapi_get_graph() const;

  /* -- グラフの便利なショートカット -- */

  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);
};
} // namespace sycl
```

この拡張機能は、SYCL キュークラスを変更してキュー・オブジェクトに状態を導入し、インスタンスを、実行に向けてすぐに送信するのではなく、コマンドグループがグラフに記録されるモードにすることができます。

この拡張機能では、`handler::graph()` のキューのショートカットとして 3 つの新しいメンバー関数も `sycl::queue` クラスに追加されます。

8.5.1. キューの状態

`sycl::queue` オブジェクトは、2 つの状態のいずれかになります。デフォルトの `queue_state::executing` 状態では、キューは送信されたコマンドグループが非同期実行のために直ちにスケジュールされる通常のセマンティクスを持ちます。

代替の `queue_state::recording` 状態はグラフ構築に使用されます。実行に向けてスケジュールされるのではなく、キューに送信されたコマンドグループは、送信ごとに新しいノードとしてグラフ・オブジェクトに記録されます。記録が終了し、キューが実行状態に戻ると、記録されたコマンドは実行されず、後続のキュー操作に対して透過的です。キューの状態は `queue::ext_oneapi_get_state()` で照会できます。

キューの状態ダイアグラム

8.5.2. 推移的キュー記録

実行可能状態のキューにコマンドグループを送信すると、状態が暗黙的に `queue_state::recording` に変更される可能性があります。これは、コマンドグループが、記録状態のキューによって返されたイベントに依存している場合に発生します。状態の変更は、コマンドグループがデバイスに送信される前に発生します (つまり、そのコマンドグループに対して新しいグラフノードが作成されます)。

このメカニズムを使用して状態が `queue_state::recording` に設定されたキューは、`command_graph::begin_recording()` に引数として渡されたように動作します。特に、`command_graph::end_recording()` が呼び出されるまで、その状態は変化しません。

状態の変更をトリガーしたイベントを持つキューの記録プロパティーも継承されます (つまり、`command_graph::begin_recording()` の元の呼び出しに渡されたプロパティーは、状態が遷移するキューに継承されます)。

例

```
// q1 状態は記録に設定されています。
```

```
graph.begin_recording(q1);

// 記録キューに送信することにより、ノードがグラフに追加されます。
auto e1 = q1.single_task(...);

// 記録中のキューによって作成された e1 への依存関係があるため、
// q2 はすぐに記録モードに入り、e1 と e2 の間にエッジを持つ
// 新しいノードが作成されます。
auto e2 = q2.single_task(e1, ...);

// q1 と q2 の記録を終了します。
graph.end_recording();
```

8.5.3. キューのプロパティー

コア SYCL 仕様で定義されている 2 つのプロパティーがあり、構築時にプロパティー・リスト・パラメーターを介して `sycl::queue` に渡すことができます。それらはこの拡張機能と次のように対話します:

1. `property::queue::in_order` - `in-order` プロパティーを使用してキューが作成されると、各操作は前の操作に暗黙的に依存するため、その操作を記録すると直線グラフになります。ただし、順序付きキューに送信されたグラフは既存の構造を維持し、キューに送信された他のコマンドグループに対して完全グラフが順序通り実行されます。SYCL ランタイムは、グラフ実行が順序付きキューに送信された 1 つのコマンドグループのように、グラフ実行の前後に暗黙的な依存関係を自動的に追加します。
2. `property::queue::enable_profiling` - このプロパティーはグラフの記録には影響しません。キューに設定されたグラフが送信されると、グラフ送信によって返されるイベントからプロファイル情報を取得できるようになります。この送信に使用される実行可能グラフは、`enable_profiling` プロパティーを使用して作成される必要があります。詳細については、`Enable-Profiling` を参照してください。実行時に送信されたグラフがスケジュールのためどのように分割されるかは定義されていないため、グラフ実行イベントのプロファイル照会で報告される `uint64_t` タイムスタンプには次のセマンティクスがあり、デバイス上の実行時間は正確でない可能性があります。
    - `info::event_profiling::command_submit` - グラフがキューに送信されたときのタイムスタンプ。
    - `info::event_profiling::command_start` - 最初のコマンドグループ・ノードの実行が開始されたときのタイムスタンプ。
    - `info::event_profiling::command_end` - 最後のコマンドグループ・ノードが実行を完了したときのタイムスタンプ。

8.5.4. 新しいキューのメンバー関数

追加された sycl::queue クラスのメンバー関数

```
queue_state queue::ext_oneapi_get_state() const;
```

効果: キューの記録状態を照会します。

リターン: キューが、コマンドをすぐに実行するようにスケジュールされるデフォルト状態にある場合、`queue_state::executing` が返されます。それ以外の場合は、コマンドが `command_graph` オブジェクトにリダイレクトされる `queue_state::recording` が返されます。

```
command_graph<graph_state::modifiable> queue::ext_oneapi_get_graph() const;
```

効果: 記録時にキューの基になるコマンドグラフを照会します。

リターン: キューがコマンドを記録しているグラフ・オブジェクト。

スロー: キューが `queue_state::recording` 状態にない場合、エラーコードが `invalid` の同期例外が発生します。

```
event queue::ext_oneapi_graph(command_graph<graph_state::executable>& graph)
```

効果: `handler::ext_oneapi_graph(graph)` を含むコマンドグループを送信するのと同等のキュー・ショートカット関数。返されるイベントのコマンド状態は、デバイス上でコマンドグループ・ノードの実行が開始されると `info::event_command_status::running` になり、すべてのノードの実行が終了すると `info::event_command_status::complete` になります。

前提条件: キューは、グラフの作成時に使用されたデバイスおよびコンテキストと同一のデバイスおよびコンテキストに関連付けられる必要があります。

リターン: キューに送信されたコマンドを表すイベントを返します。

スロー: キューがグラフに記録されており、グラフにグラフ所有のメモリー割り当てが含まれている場合、エラーコードが `invalid` の同期例外が発生します。

```
event queue::ext_oneapi_graph(command_graph<graph_state::executable>& graph,
                              event depEvent);
```

効果: `handler::depends_on(depEvent)` と `handler::ext_oneapi_graph(graph)` を含むコマンドグループを送信するのと同等のキュー・ショートカット関数。返されるイベントのコマンド状態は、デバイス上でコマンドグループ・ノードの実行が開始されると `info::event_command_status::running` になり、すべてのノードの実行が終了すると `info::event_command_status::complete` になります。

前提条件: キューは、グラフの作成時に使用されたデバイスおよびコンテキストと同一のデバイスおよびコンテキストに関連付けられる必要があります。

リターン: キューに送信されたコマンドを表すイベントを返します。

スロー: キューがグラフに記録されており、グラフにグラフ所有のメモリー割り当てが含まれている場合、エラーコードが `invalid` の同期例外が発生します。

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

効果: `handler::depends_on(depEvents)` と `handler::ext_oneapi_graph(graph)` を含むコマンドグループを送信するのと同等のキュー・ショートカット関数。返されるイベントのコマンド状態は、デバイス上でコマンドグループ・ノードの実行が開始されると `info::event_command_status::running` になり、すべてのノードの実行が終了すると `info::event_command_status::complete` になります。

前提条件: キューは、グラフの作成時に使用されたデバイスおよびコンテキストと同一のデバイスおよびコンテキストに関連付けられる必要があります。

リターン: キューに送信されたコマンドを表すイベントを返します。

スロー: キューがグラフに記録されており、グラフにグラフ所有のメモリー割り当てが含まれている場合、エラーコードが `invalid` の同期例外が発生します。

8.5.5. 新しいハンドラーのメンバー関数

追加された sycl::handler クラスのメンバー関数

```
void handler::ext_oneapi_graph(command_graph<graph_state::executable>& graph)
```

効果: グラフの実行を呼び出します。一度に実行されるグラフのインスタンスは 1 つだけです。グラフが複数回送信されると、同一グラフの同時実行を防ぐために、ランタイムによって依存関係が自動的に追加されます。

スロー:

- グラフの作成時に使用されたデバイスおよびコンテキストとは異なるデバイスやコンテキストに関連付けられたキューにハンドラーが送信された場合、エラーコードが `invalid` の同期例外が発生します。
- ハンドラーがグラフに記録されているキューに送信され、グラフにグラフ所有のメモリー割り当てが含まれている場合、エラーコードが `invalid` の同期例外が発生します。

```
template <typename DataT, int Dims, access::mode AccMode, access::target
AccTarget, access::placeholder IsPlaceholder> void
handler::require(ext::oneapi::experimental::dynamic_parameter<
                 accessor<DataT, Dims, AccMode, AccTarget, IsPlaceholder>>
```

```
                    dynamicParamAcc)
```

効果: 動的パラメーター `dynamicParamAcc` に含まれるアクセサーに関連付けられたメモリー・オブジェクトへのアクセスが必要です。

スロー:

- この関数が、現在グラフに記録中のキューに送信されたコマンドグループから呼び出された場合、エラーコードが `invalid` の同期例外が発生します。
- この関数が通常の SYCL コマンドグループの送信から呼び出された場合、エラーコードが `invalid` の同期例外が発生します。

```
template <typename T>
void handler::set_arg(int argIndex,
            ext::oneapi::experimental::dynamic_parameter<T> &dynamicParam);
```

効果: 動的パラメーター内の値に基づいて、インデックス `argIndex` を持つ引数をカーネルに設定し、その動的パラメーター `dynamicParam` をこの関数を呼び出すコマンドグループの送信をカプセル化するグラフノードに登録します。

スロー:

- この関数が、現在グラフに記録中のキューに送信されたコマンドグループから呼び出された場合、エラーコードが `invalid` の同期例外が発生します。
- この関数が通常の SYCL コマンドグループの送信から呼び出された場合、エラーコードが `invalid` の同期例外が発生します。

8.6. スレッドの安全性

この拡張機能の新しい関数は、基本 SYCL 仕様のクラスのメンバー関数と同じように、スレッドセーフです。ユーザーコードが同じキューにアクセスする 2 つのスレッドの同期を行わない場合、そのキュー上のイベント間には厳密な順序付けはなく、カーネルによる送信、記録、および終了は不定の順序で発生します。

あるスレッドがキューへの記録を終了しているときに別のスレッドがワークを送信している場合、どのカーネルが後続のグラフの一部になるかは未定義です。ユーザーコードがキューイベントに全順序を強制する場合、動作は明確に定義され、観測可能な全順序と一致します。

`queue::ext_oneapi_get_state()` からの戻り値は、別のスレッドが既にキューの状態を変更している可能性があるため、マルチスレッド環境では古いものであると見なす必要があります。

8.7. 例外の安全性

SYCL 共通参照セマンティクスによって提供される破棄セマンティクスに加えて、変更可能な `command_graph` の最後のコピーが破棄されると、そのグラフに対して記録を行っているすべてのキューでの記録が終了します。これは、`this->end_recording()` を呼び出すのと同等です。

その結果、ユーザーは、例外発生時にキュー記録コードを `try` / `catch` ブロックに手動でラップして、キュー記録の状態を実行状態にリセットする必要がなくなります。代わりに、変更可能なグラフを破棄するキャッチされていない例外がこのアクションを実行します。これは RAII パターンの使用に役立ちます。

8.8. コマンドグループ関数の制限事項

SYCL 仕様では禁止されていませんが、グラフノードの作成に使用される CGF (コマンドグループ関数) 内にある任意の C++ コードをキャプチャーできないことに注意してください。このコードは、`queue::submit()` または `command_graph::add()` の呼び出し中にハンドラー関数の呼び出しとともに 1 回評価されますが、グラフの以降の実行には反映されません。

同様に、`dynamic_command_group` 内のすべてのコマンドグループ関数は、`command_graph::add()` を使用してグラフに送信されるとインデックス順に 1 回評価されます。

そのようなコードは、この拡張機能と互換性を保つため、別のホストタスクに移動し、記録または明示的な API を介してグラフに追加する必要があります。

8.9. ホストタスク

ホストタスクは、ネイティブ C++ 呼び出し可能オブジェクトであり、SYCL の依存関係規則に従ってスケジュールされます。ホストタスクをグラフの一部として記録することは有効ですが、ホストタスクのノードによって SYCL ランタイムが実行可能ファイル `command_graph` 全体をデバイスに一度に送信できなくなることがあるため、グラフのパフォーマンスが低下する可能性があります。

```
auto node = graph.add([&](sycl::handler& cgh){
  // ホストコードは add() の呼び出し中に評価されます
  cgh.host_task([=](){
    // コードはコマンド・グラフ・ノード実行の一部として評価されます
  });
});
```

ホストタスクは、実行可能グラフ更新を使用して更新できます。

8.10. 記録モードでのキューの動作

`command_graph::begin_recording` の呼び出しによってキューが記録モードに設定されると、このモード中はコマンドが実行されないため、キューの一部の機能は使用できなくなります。一般的な考え方は、機能が利用できない時に実行時に例外をスローし、失敗が明確に示されるようにすることです。次のリストは、記録モード中に変化する動作について説明しています。以下にリストされていない機能は、記録モードでも非記録モードと同じように動作します。

8.10.1. イベントの制限事項

変更可能な `command_graph` に記録されているキューの送信では、`handler::depends_on()` へのパラメーター、または `queue::parallel_for()` などのキューのショートカットの依存イベントとして使用できるイベントは、同じ変更可能な `command_graph` に記録されたキューの送信から返されたイベントのみです。

記録状態のキューに送信され、返されるイベントに関するその他の制限は次のとおりです:

- `event::get_info<info::event::command_execution_status>()` または `event::get_profiling_info()` を呼び出すと、同期的にエラーコード `invalid` がスローされます。
- ホスト側でイベントを待機すると、`invalid` エラーコードが同期的にスローされます。
- 記録範囲外でイベントを使用すると、`invalid` エラーコードが同期的にスローされます。

8.10.2. キューの制限事項

記録状態のキューでのホスト側の待機はエラーであり、`invalid` エラーコードが同期的にスローされます。

8.10.3. バッファーの制限事項

ユーザーが `Assume-Buffer-Outlives-Graph` プロパティーを使用してグラフを作成しない限り、`command_graph` 内でのバッファーの使用は制限されます。バッファーのライフタイムは、それが使用される `command_graph` によって延長されないため、ユーザーはバッファーのライフタイムが `command_graph` のライフタイムより長いことを確認する必要があります。このプロパティーを使用せずに `command_graph` でバッファーを使用しようとすると、エラーコードが `invalid` の同期エラーがスローされます。

`command_graph` に記録されたコマンド内のホスト・データ・ポインターで作成されたバッファーの使用にも制限があります。`command_graph` の実行が遅延されるため、これらのバッファーを使用するコマンドがグラフに送信されたときに、データが直ちにデバイスにコピーされない可能性があります。そのため、正しい動作を保証するには、ホストデータもグラフよりも長く存続する必要があります。

記録されたグラフの実行が遅延されるため、バッファーの破棄動作のコピーバックに依存するキャプチャーされたコードをサポートできません。通常、アプリケーションは、コマンドグラフ内では本質的にキャプチャーできないホスト上でのワークを実行するためこの動作に依存します。

- したがって、グラフに記録するときに、ライトバックを引き起こすバッファー上のアクセサーを持つコマンドを送信するとエラーになります。この場合、互換性のないバッファーを使用すると、エラーコードが `invalid` の同期エラーがスローされます。
- ホストストレージが接続されたバッファーのコピーバック・メカニズムは、`buffer::set_final_data(nullptr)` または `buffer::set_write_back(false)` のどちらかを使用して明示的に無効にできます。
- また、コマンドグラフに現在記録中のコマンドで使用されるバッファーへのホストアクセサーを作成することもエラーです。互換性のないバッファーへのホストアクセサーを構築しようとすると、エラーコードが `invalid` の同期エラーがスローされます。

8.10.4. エラー処理

キューが記録モードの場合、デバイスでの実行が行われていないため、非同期例外は発生しません。デフォルトのキュー実行状態でスローされるように指定された同期エラーは、キューが記録状態にあるときにもスローされます。キュー 照会メソッドは、スローとは対照的に、記録モードでは通常どおり動作します。

8.11. 他の拡張機能との相互作用

このセクションでは、`sycl_ext_oneapi_graph` と他の拡張機能との相互作用を定義します。

8.11.1. sycl_ext_oneapi_async_memory_alloc

`sycl_ext_oneapi_async_memory_alloc` で定義された API は、グラフでの使用がサポートされています。詳細については、グラフ所有のメモリー割り当てのセクションを参照してください。

8.11.2. sycl_ext_codeplay_enqueue_native_command

`sycl_ext_codeplay_enqueue_native_command` で定義された新しいメソッドは、グラフノードで使用できます。詳細については、SYCL グラフの相互作用のセクションを参照してください。

8.11.3. sycl_ext_intel_queue_index

`sycl_ext_intel_queue_index` で定義された計算インデックス・キューのプロパティーは、キューの記録中は無視されます。

`sycl_ext_oneapi_graph` の将来の改訂で、この情報の利用が検討される可能性があります。

8.11.4. sycl_ext_oneapi_bindless_images

`sycl_ext_oneapi_bindless_images` で定義された新しいハンドラーメソッドとキューのショートカットは、グラフノードでは使用できません。ユーザーがそれらをグラフに追加しようとすると、エラーコードが `invalid` の同期例外がスローされます。

sycl_ext_oneapi_graph の将来の改訂で、この制限の削除が検討される可能性があります。

8.11.5. sycl_ext_oneapi_device_global

sycl_ext_oneapi_device_global で定義された新しいハンドラーメソッドとキューのショートカットは、グラフノードでは使用できません。ユーザーがそれらをグラフに追加しようとすると、エラーコードが invalid の同期例外がスローされます。

sycl_ext_oneapi_graph の将来の改訂で、この制限の削除が検討される可能性があります。

8.11.6. sycl_ext_oneapi_discard_queue_events

ext::oneapi::property::queue::discard_event プロパティーを使用して作成された sycl::queue を記録する場合、キュー送信から返されたこれらのイベントを使用してグラフエッジを作成するのは無効です。これは、無効なイベントが渡されると handler::depends_on() が例外をスローするという、sycl_ext_oneapi_discard_queue_events 仕様の説明と一致しています。

8.11.7. sycl_ext_oneapi_enqueue_barrier

sycl_ext_oneapi_enqueue_barrier によって定義された新しいハンドラー メソッドとキューのショートカットは、バリアの依存関係強制がイベントに依存するため、記録 & 再生 API を使用して作成されたグラフノードでのみ使用できます。

ユーザーが明示的な API を使用してグラフにバリアコマンドを追加しようとすると、エラーコードが invalid の同期例外がスローされます。ユーザーが明示的な API を使用してグラフを構築する場合、バリアの代わりに node::depends_on_all_leaves プロパティーで作成された空のノードを使用できます。

バリアのセマンティクスは、単一コマンドキューに対し sycl_ext_oneapi_enqueue_barrier で定義され、複数のキューから記録されたノードや明示的な API によって追加されたノードを含む可能性のあるグラフに対し次のように対応します:

- 待機リスト・パラメーターが空のバリアは、バリアコマンドが記録されているキューからグラフに追加されたリーフノードのみに依存します。
- バリアコマンドに暗黙的に依存するコマンドは、バリアコマンドが発行されたのと同じキューから記録されたコマンドのみです。

8.11.8. sycl_ext_oneapi_enqueue_functions

sycl_ext_oneapi_enqueue_functions で定義されているコマンド送信関数は、キュー記録からグラフを作成するときにグラフへのノード追加に使用できます。また、イベントを返さずにキューに直接送信するなど、実行可能なグラフを送信する新しいメソッドも定義されています。

8.11.9. sycl_ext_oneapi_free_function_kernels

sycl_ext_oneapi_free_function_kernels で定義されている
sycl_ext_oneapi_free_function_kernels は、SYCL グラフで使用できます。

8.11.10. sycl_ext_oneapi_kernel_compiler_spirv

sycl_ext_oneapi_kernel_compiler_spirv を使用してロードされたカーネルは、グラフノードで使用すると通常どおり動作します。

8.11.11. sycl_ext_oneapi_kernel_properties

sycl_ext_oneapi_kernel_properties によって定義された新しいハンドラーメソッドとキューのショートカットは、通常のキュー送信と同じ方法でグラフノードで使用できます。

8.11.12. sycl_ext_oneapi_local_memory

sycl::ext::oneapi::group_local_memory() または
sycl::ext::oneapi::group_local_memory_for_overwrite() を使用して、グラフカーネルのノード内でのローカルメモリーを割り当てがサポートされています。これらのメソッドは、sycl_ext_oneapi_local_memory によって定義されます。

8.11.13. sycl_ext_oneapi_memcpy2d

sycl_ext_oneapi_memcpy2d で定義された新しいハンドラーメソッドとキューのショートカットは、グラフノードでは使用できません。ユーザーがそれらをグラフに追加しようとすると、エラーコードが invalid の同期例外がスローされます。

sycl_ext_oneapi_graph の将来の改訂で、この制限の削除が検討される可能性があります。

8.11.14. sycl_ext_oneapi_prod

sycl_ext_oneapi_prod によって追加された新しい sycl::queue::ext_oneapi_prod() メソッドは、キューの記録中は通常どおり動作し、グラフにキャプチャーされません。記録されたコマンドは、操作の目的上、送信済みとしてカウントされません。

8.11.15. sycl_ext_oneapi_queue_empty

sycl_ext_oneapi_queue_empty 拡張機能によって定義された queue::ext_oneapi_empty() クエリーは、キューの記録中は通常どおり動作し、グラフにキャプチャーされません。記録されたコマンドは、このクエリーの目的上、送信済みとしてカウントされません。

8.11.16. sycl_ext_oneapi_queue_priority

sycl_ext_oneapi_queue_priority で定義されたキューの優先度プロパティーは、キューの記録中は無視されます。

8.11.17. sycl_ext_oneapi_work_group_memory

グラフ・カーネル・ノード内の sycl_ext_oneapi_work_group_memory で定義された `work_group_memory` オブジェクトの使用がサポートされています。

8.11.18. sycl_ext_oneapi_work_group_scratch_memory

sycl_ext_oneapi_work_group_scratch_memory で定義された新しいプロパティーは、グラフノードでは使用できません。ユーザーがそれらをグラフに追加しようとすると、エラーコードが `invalid` の同期例外がスローされます。
sycl_ext_oneapi_graph の将来の改訂で、この制限の削除が検討される可能性があります。

9. サンプルと利用ガイド

詳細なコード例とガイドラインは、SYCL グラフ使用ガイドに記載されています。

10. 将来の方向性

このセクションには、開発済みですがまだ実装されていない仕様の機能と、まだ開発中の機能の両方が含まれています。開発済みの機能は、実装されるとすぐにメイン仕様に移動されます。

10.1. 実装待ちの機能

10.1.1. ストレージのライフタイム

コマンドグラフへの送信の一部として記録されたバッファーのライフタイムは、SYCL 仕様の共通参照セマンティクスとバッファーの同期規則に従って延長されます。これは、グラフのライフタイム (変更可能なグラフと、そこから作成された実行可能グラフの両方を含む)、またはグラフでバッファーが不要になるまで (実行可能グラフの更新によって置き換えられた後など) 延長されます。

ホストデータのポインターを使用して作成されたバッファーがコマンドグラフへの送信の一部として記録される場合、バッファー内にそのデータのコピーが作成されることで、そのホストデータのライフタイムも延長されます。次の例を考えてみましょう:

```
void foo(queue q /* 記録モードのキュー */ ) {
  float data[NUM];
```

```
  buffer buf{data, range{NUM}};
  q.submit([&](handler &cgh) {
    accessor acc{buf, cgh, read_only};
    cgh.single_task([] {
      // "acc」を使用
    });
  });
  // "data" は範囲外
}
```

この例では、実装は記録されたグラフで使用されるため、バッファーのライフタイムが延長されます。バッファーはホストのメモリーデータを使用するため、実装ではそのホストデータの内部コピーも作成されます。上に示すように、記録されたグラフが範囲外になる前、またはデータがデバイスにコピーされる前に、ホストメモリーが範囲外になる可能性があります。

このような場合、デフォルトの動作では常にホストデータがコピーされますが、ホストデータのライフタイムが記録されたグラフのライフタイムよりも長くなることが判明している場合、これは必要ありません。ユーザーがこの状況を認識している場合、`graph::assume_data_outlives_buffer` プロパティーを使用して内部コピーを回避できます。プロパティーを `begin_recording()` に渡すと、特定のキューに対して `end_recording()` が呼び出される前に記録されたコマンドにのみホストコピーが防止されます。プロパティーを `command_graph` コンストラクターに渡すと、グラフに記録されたすべてのコマンドのホストコピーが防止されます。

この実装では、このバッファーにアクセスするすべてのコマンドが `access_mode::write` または `no_init` プロパティーを使用する場合、ホストメモリーは必要ないため、ホストメモリーが内部的にコピーされないことが保証されます。ただし、これらのケースでは、バッファーの制限で説明されているように、アプリケーションでコピーバックを無効にする必要があることに注意してください。

10.2. 開発中の機能

10.2.1. デバイス固有のクグラフ

変更可能な状態の `command_graph` は、ファイナライズ時にデバイスにのみ関連付けられたデバイスに依存しない表現ではなく、特定のデバイスを対象とするノードを含みます。これにより、コマンドグループ関数が評価されるときにデバイス情報を必要とするノードを実装が処理できるようになります。例えば、SYCL のリダクション実装では、通常ランタイムがキューに関連付けられたデバイスから収集する work-group/sub-group のサイズが必要となることがあります。

この設計により、将来的にユーザーがさまざまなデバイスを対象とするノードでグラフを構成できるようになり、送信前に実行グラフを定義する利点をマルチデバイス・プラットフォームでも活用できるようになります。この機能がない場合、ユーザーは現在、個別の単一のデバイスグラフを送信し、依存関係のイベントを使用する必要があります。この拡張機能は、この使用モデルを最適化することを目的としています。デバイス間でのコマンドの自動負荷分散は、この拡張機能が解決を目指す課題ではありません。各コマンドが処理されるデバイスを決定するのはユーザーの責任であり、SYCL ランタイムの役割ではありません。

11. 問題

11.1. 多くのコマンドタイプを更新

カーネル実行コマンド以外のノードタイプへの引数の更新をサポートします。

未解決 少なくともメモリー・コピー・ノードとホストタスクに追加する必要があります。ただし、サポートの全範囲を設計して実装する必要があります。

11.2. 更新可能なプロパティー・グラフの再送信

更新可能なグラフを導入することで、同一グラフの以前の送信がまだ実行中である間に、再送信により生成されるグラフ間の依存関係を排除できる可能性があります。ただし、これがユーザーにとって望ましいことであり、意味があることを保証するには、さらなる設計上の議論が必要です。

未解決 さらに議論が必要です。

11.3. 記録 & 再生 API の更新可能なコマンドグループ:

現在、グラフ内のコマンドグループを更新する唯一の方法は、明示的 API を使用することです。一部のバックエンドには制限があり、グラフがファイナライズされる前に更新に使用されるすべてのコマンドグループを指定する必要があります。この制限により、記録 & 再生 API のパフォーマンスを最大化して実装することが難しくなります。

未解決 さらに議論が必要です。

11.4. マルチデバイス・グラフ

実行可能グラフには、異なるデバイスを対象とするノードを含めることができます。

未解決: 解決の可能性があります。この機能は、今後の改訂で拡張機能に導入することを検討しているものです。グラフノードの定義はデバイス固有で、ある程度計画されています。

11.5. デバイスに依存しないグラフ

明示的な API は、キューを介して特定のデバイスに送信可能で、デバイスに依存しないグラフをサポートできます。この問題は、マルチデバイス・グラフに関連しています。

未解決: 解決の可能性はありません。現在のランタイムの制限のため、合理的な労力でこれを実装することはできません。

11.6. 実行プロパティー

現在の提案には、SYCL の既存 API に対する幅広い拡張が含まれています。キューのフラッシュ動作をユーザーが制御し、再実行可能なハンドラーを提供することで、同様の成果を達成できるでしょうか?

未解決: 解決の可能性はありません。設計の再検討と制限の可能性があります。

11.7. ユーザーガイドによるスケジュール

特定のワークロードでは、デバイスにコマンドグラフをスケジュールする方法のヒントをランタイムに提供すると便利な場合があります。この情報は、幅優先や深さ優先、ブロックサイズとの組み合わせなどのスケジュール・ポリシーに影響する可能性があります。

未解決: 解決の可能性があります。基本コマンドグラフの提案を拡張するか、個別の拡張提案として階層化して、新しいプロパティーをファイナライズ呼び出しに追加できます。

11.8. USM ポインターとしてグラフ所有割り当て

現在、グラフ所有のメモリー割り当ては、通常の USM ポインターと同じようにグラフノード内で使用でき、グラフ外での使用は無効であると説明されています。ただし、実際の USM 割り当てでは実装されない可能性があります (仮想メモリーまたはその他のアプローチの可能性があります)。

これらが USM ポインターであると断言できる場合は話が簡単ですが、これらがまったく同じように機能するかどうかは明らかではありません。

未解決: 潜在的な、または未知の問題:

- 仮想アドレスは、`get_pointer_device` や `get_pointer_type` などの照会の USM ポインターとして機能したり、報告されたりしますか?
- `sycl::free` を介して USM ポインターの割り当てを解除できることが期待されますが、これは他の実装アプローチでは有効ではない可能性があります。

12. 未実装の機能と既知の問題

次の機能はまだサポートされていないため、アプリケーション・コードで使用すると例外がスローされます。

1. グラフノードでリダクションを使用する。
2. グラフノードで sycl ストリームを使用する。

13. 改訂履歴

リビジョン	日付	著者	変更
1	2023 年 3 月 23 日	Pablo Reble、Ewan Crawford、Ben Tracy、Julian Miller	最初の公開作業の草案
2	2023 年 8 月 1 日	Pablo Reble、Ewan Crawford、Ben Tracy、Julian Miller、Maxime France-Pillois	状態を実験段階に昇格