SYCL* Graph

この記事は 2024 年 1 月 22 日に Codeplay のウェブサイトで公開された「SYCL™ Graphs」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。

GPU などのアクセラレーターを使用するアプリケーションでは、開発者は**計算カーネル**を作成し、計算カーネルはアクセラレーター上で 1 つずつ実行されます。これは「オフロード」と呼ばれ、開発者が計算を CPU から GPU に移行することを意味します。通常、計算カーネルはホストからデータを受け取り、そのデータに対して何らかの操作を行い、アプリケーションが続行できるようにデータをホスト CPU に戻します。ただし、アクセラレーターにオフロードされるワークロードが増えるにつれて、データを操作する複数のカーネルをデバイスにオフロードすることが一般的になっています。オフロードのプロセスは、たとえ短時間であっても 2 つの異なるプロセッサーを同期させる性質上、一般的にレイテンシー (つまり、カーネルを送信してから実際に実行するまでの時間) や追加の CPU ビジーワーク (CPU スレッドがハンドシェイクやデータの受け渡しのために複数回停止して同期する必要がある) といったオーバーヘッドが発生します。

しかし、アプリケーションによっては、開発者が同じ送信で一緒にスケジューリングできる計算カーネルのグループを特定することが可能です。さらに興味深いことに、アプリケーションのメインループが、入力と出力だけを変えて、同じグループの計算カーネルを何度も再発行するケースもあります。

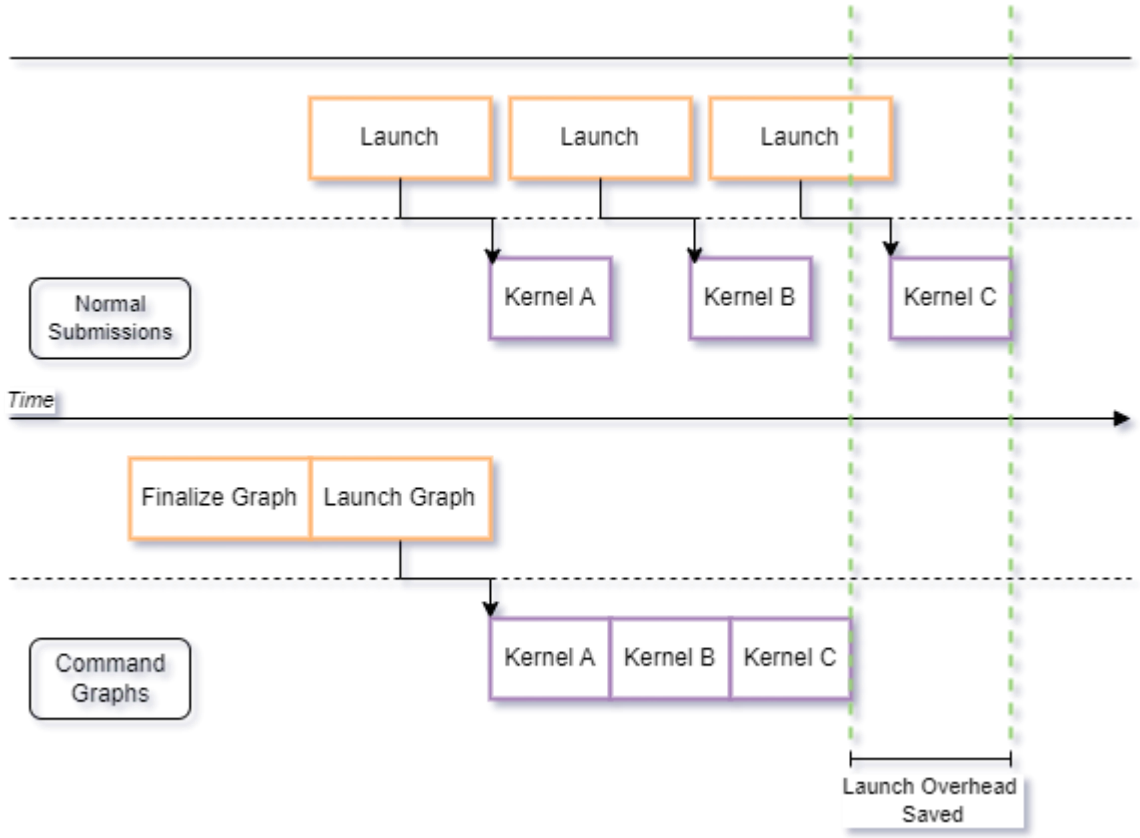
このような状況に対応するため、私たちは「SYCL* Graph」と呼ばれる SYCL* 拡張機能を設計しました。これにより、計算カーネルとメモリー操作間の依存関係の計算グラフを定義し、バッチ化して、何度でも送信することができます。SYCL* はマルチプラットフォームであるため、この機能は、サポートされているさまざまなバックエンドとハードウェア上の抽象化にマッピングされます。例えば、NVIDIA* GPU の CUDA* グラフ、OpenCL* のコマンドバッファー、レベルゼロや Vulkan* のコマンドリストなどです。

この記事では、この実行モデルと、ユーザーがインターフェイスを利用して試行できる API について詳しく説明し、DPC++ 実装でそれを使用する方法を紹介し、最後にロードマップの次のアイデアと計画について触れます。

拡張機能

SYCL* アプリケーションでは、コマンドは実行のためキューに送信され、オプションで依存関係を含めることもできます。これらの送信により、実行時に SYCL* コマンドグループの暗黙的な依存関係グラフが有向非巡回グラフ (DAG) 形式で定義されます。ただし、コマンドの実行はコマンドの送信に関連付けられているため、コマンドは積極的に実行され、ランタイムに送信されたコマンドの完全な DAG は、すべてが実行のため送信されるまで不明です。Codeplay はインテルと共同で、この制約を排除する SYCL* 向けのコマンドグラフ拡張機能「sycl_ext_oneapi_graph」(英語) を開発しました。この拡張機能は、コマンドの送信と実行を分離し、ユーザーが実行したいコマンドの完全な DAG を事前に定義できるようにします。ユーザーは、個々のコマンドを再送信することなく、ユーザー定義グラフを繰り返し送信できます。

これを SYCL* API を介して利用できるようにすることで、SYCL* ランタイムとグラフの両方のレベルで最適化の可能性が生まれ、バッチ化によるコマンドのホスト起動レイテンシーの短縮、カーネル・フュージョンを含むグラフ全体の最適化、コマンドの繰り返し実行のワークフローの簡素化などが可能になります。拡張機能とその実装に関する詳細は、IWOCL & SYCLCON 2023 での講演「SYCL* コマンドグラフの遅延実行に向けて」の録画 (英語) をご覧ください。



主なオブジェクト

command_graph クラス

command_graph クラスは、実行 DAG を形成するコマンドのコレクションを表します。コマンドグラフには、多数のノード (コマンド) とエッジ (ノード間の依存関係を定義) が含まれます。command_graph オブジェクトは、最初は変更可能な状態で作成されます。この状態では、ノードとエッジのどちらもグラフに追加できますが、実行することはできません。変更可能なグラフをファイナライズして、実行可能な状態で新しいグラフを作成できます。このグラフは、新しいノードやエッジを追加して変更することはできませんが、実行のため SYCL* キューに送信することはできます。コマンドグラフは、グラフ作成時に固定される単一のデバイスをターゲットとします。

ノード

前述のように、ノードクラスのオブジェクトは、グラフ内の単一のコマンドを表します。具体的には、SYCL* コマンドグループを表します。これは、カーネル実行、メモリー操作、またはその他の SYCL* コマンドタイプです。

エッジ

グラフ内のエッジは C++ オブジェクトでは表されませんが、ノード間の「事前に発生する」関係を表します。エッジは、使用されている API に応じて、グラフにノードとして送信されたコマンドグループから自動的に推測されるか、手動で追加されます。

API

拡張機能には、コマンドグラフの作成に使用できる 2 つの API が含まれています。これらは、特定のユースケースに応じて個別に使用することも、組み合わせて使用することもできます。

記録と再生 API は、SYCL* キューに送信されたコマンドをグラフ内のノードとして記録し、依存関係を自動的に推測します。

明示的なグラフ作成 API を使用すると、ユーザーはグラフのノードとエッジを明示的に定義して、その構造をより細かく制御できます。

次のセクションでは、実用的な SYCL* の例を用いて、これら 2 つの API の使用法について説明します。

実用的な SYCL* の例

以下は、コマンドグラフを使用するため既存の SYCL* コードを変更する方法を示す基本的な SYCL* プログラムの例です。この例では USM メモリーを使用し、簡潔化のためいくつかの細かいコードを省略しています。

```
using namespace sycl;
using namespace sycl_ext = sycl::ext::oneapi::experimental;

queue Queue{default_selector{}};

float *PtrA = sycl::malloc_device<float>(N, Queue);
float *PtrB = sycl::malloc_device<float>(N, Queue);
float *PtrC = sycl::malloc_device<float>(N, Queue);

// Init device data

auto InitEvent = Queue.submit([&](handler& CGH) {
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrA[id] = 1.0f;
    PtrB[id] = 2.0f;
    PtrC[id] = 3.0f;
  });
});

auto EventA = Queue.submit([&](handler& CGH) {
  CGH.depends_on(InitEvent);
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrA[id] += 1.0f;
  });
});

auto EventB = Queue.submit([&](handler& CGH) {
  CGH.depends_on(InitEvent);
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrB[id] += 1.5f;
  });
});

auto EventC = Queue.submit([&](handler& CGH) {
  CGH.depends_on({EventA, EventB});
```

```
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrC[id] += (PtrA[id] + PtrB[id]);
  });
});

Queue.wait_and_throw();
// Do something with the data
```

コードの「グラフ化」

コマンドグラフを有効にする API に関係なく、このコードにいくつかの修正を加える必要があります。

1. グラフに取り込みたいコマンドの前に、変更可能なグラフを作成します。コマンドグラフの作成には、コンテキストとターゲットデバイスが必要です。

   ```
   sycl_ext::command_graph Graph(Queue.get_context(), Queue.get_device());
   ```

2. 必要なコマンドをすべて追加したら、グラフをファイナライズする必要があります。ファイナライズでは、デバイスで実行するグラフを準備するのに必要な計算コストの高い操作を実行します。通常の SYCL* コマンドの送信では、これらは実行の直前に発生するため、キューの送信から特定のデバイスでのコマンドの実行終了までの総実行時間に影響します。これらの操作を分離することで、アプリケーションはこのコストを事前に処理し、コマンドを一度に実行して、グラフを繰り返し実行することでランタイム・オーバーヘッドを排除できます。変更可能なグラフに対して finalize() を呼び出すと、実行可能なグラフ・オブジェクトが生成されます。次に例を示します。

   ```
   auto ExecGraph = Graph.finalize();
   ```

3. 実行可能なグラフをキューに送信し、必要な回数だけ実行します。ここではキューのショートカットを使用していますが、handler::ext_oneapi_graph() を使用して、通常のコマンドグループ送信内でコマンドグラフを送信することもできます。次に例を示します。

   ```
   Queue.ext_oneapi_graph(ExecGraph);
   ```

記録と再生 API

記録と再生 API を使用すると、最小限の変更でキューにコマンドを送信する既存のコードを簡単にキャプチャーできます。キューがキャプチャー状態にある間、キューに送信されたコマンドはデバイスで実行するために送信されるのではなく、グラフにノードとして追加されます。この API を使用してキューのキャプチャーを有効にするには、キュー送信を囲むように graph:: begin_recording(queue) と graph::end_recording(queue) への呼び出しを追加するだけです。次に例を示します。

```
Graph.begin_recording(Queue);

// Submit commands to the queue here via Queue.submit()...

Graph.end_recording(Queue);

// Finalize and submit graph.
```

現在グラフにコマンドを記録しているキューへの送信から返されるイベントは、コマンドグラフ・コマンドの依存関係を定義する以外には使用できません。また、記録されたコマンドグラフ外からのイベントにグラフコマンドの依存関係を定義することもできません。

この方法で記録されたコマンドには、通常の SYCL* メカニズム (バッファーアクセサーの使用と handler::depends_on() の呼び出し) から推測される依存関係 (エッジ) があります。これは、通常の SYCL* プログラムと同じように、ランタイムによって完全な DAG を構築するために使用されます。

これらの変更を適用した最終的なサンプルコードは次のようになります。

```
using namespace sycl;
using namespace sycl_ext = sycl::ext::oneapi::experimental;

queue Queue{default_selector{}};

float *PtrA = sycl::malloc_device<float>(N, Queue);
float *PtrB = sycl::malloc_device<float>(N, Queue);
float *PtrC = sycl::malloc_device<float>(N, Queue);

// Init device data

sycl_ext::command_graph Graph(Queue.get_context(), Queue.get_device());
Graph.begin_recording(Queue);

auto InitEvent = Queue.submit([&](handler& CGH) {
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrA[id] = 1.0f;
    PtrB[id] = 2.0f;
    PtrC[id] = 3.0f;
  });
});

auto EventA = Queue.submit([&](handler& CGH) {
  CGH.depends_on(InitEvent);
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrA[id] += 1.0f;
  });
});

auto EventB = Queue.submit([&](handler& CGH) {
  CGH.depends_on(InitEvent);
  CGH.parallel_for(range<1>(N),[=](item<1> id) {
    PtrB[id] += 1.5f;
  });
});

auto EventC = Queue.submit([&](handler& CGH) {
  CGH.depends_on(EventA);
  CGH.depends_on(EventB);
  CGH.parallel_for(range<1>(N), [=](item<1> id) {
    PtrC[id] += (PtrA[id] + PtrB[id]);
  });
});

Graph.end_recording(Queue);
auto ExecGraph = Graph.finalize();
```

```
Queue.ext_oneapi_graph(ExecGraph);
Queue.wait_and_throw();

// Do something with the data
```

明示的なグラフ作成 API

明示的なグラフ作成 API は、グラフのノードとエッジの定義をより細かく制御し、コマンドを記録する SYCL* キューを必要としません。ただし、これを使用するには、サンプルコードをさらに変更する必要があります。

コマンドをキューに送信する代わりに、変更可能なグラフで add() を呼び出します。add() を呼び出すと、ノード間の依存関係を定義するのに使用されるノードクラスのインスタンスが返されます。add() を呼び出すときは、キュー送信で使用したのと同じコマンドグループ関数を渡し、オプションで依存関係を定義する任意の数のノードを含むプロパティー node::depends_on を渡します。次に例を示します。

```
auto NodeA = Graph.add(
  [&](handler& CGH) {
    CGH.parallel_for(range<1>(N), [=](item<1> id) {
      PtrA[id] += 1.0f;
    });
  },
  {sycl_ext::property::node::depends_on(NodeInit)});
```

前述の例のように、依存関係を定義するのに handler::depends_on を呼び出している場合、graph::add() からイベントは返されないので、それらを削除する必要があることに注意してください。上記のプロパティーに置き換えるか、依存関係を明示的に定義する graph.make_edge(srcNode, destNode) 呼び出しに置き換えることができます。次に例を示します。

```
Graph.make_edge(NodeInit, NodeA);
```

これらの変更を適用した最終的なサンプルコードは次のようになります。

```
using namespace sycl;
using namespace sycl_ext = sycl::ext::oneapi::experimental;

queue Queue{default_selector{}};

float *PtrA = sycl::malloc_device<float>(N, Queue);
float *PtrB = sycl::malloc_device<float>(N, Queue);
float *PtrC = sycl::malloc_device<float>(N, Queue);

// Init device data

sycl_ext::command_graph Graph(Queue.get_context(), Queue.get_device());

auto InitNode = Graph.add(
  [&](handler& CGH) {
    CGH.parallel_for(range<1>(N), [=](item<1> id) {
      PtrA[id] = 1.0f;
      PtrB[id] = 2.0f;
      PtrC[id] = 3.0f;
    });
});
```

```
auto NodeA = Graph.add(
  [&](handler& CGH) {
    CGH.parallel_for(range<1>(N), [=](item<1> id) {
      PtrA[id] += 1.0f;
    });
  },
  {sycl_ext::property::node::depends_on(InitNode)});

auto NodeB = Graph.add(
  [&](handler& CGH) {
    CGH.parallel_for(range<1>(N), [=](item<1> id) {
      PtrB[id] += 1.5f;
    });
  },
  {sycl_ext::property::node::depends_on(InitNode)});

auto NodeC = Graph.add(
  [&](handler& CGH) {
    CGH.parallel_for(range<1>(N), [=](item<1> id) {
      PtrC[id] += (PtrA[id] + PtrB[id]);
    });
  },
  {sycl_ext::property::node::depends_on(NodeA, NodeB)});

auto ExecGraph = Graph.finalize();
Queue.ext_oneapi_graph(ExecGraph);
Queue.wait_and_throw();

// Do something with the data
```

実装

DPC++ 2024.0 には、コマンドグラフ拡張機能の実験的な実装が含まれています。実験的な実装であるため、すべての DPC++ バックエンド、SYCL* 機能、拡張機能が使用できるわけではありません。サポートを強化するため、引き続き、実装の積極的な開発が行われています。2024.0 でサポートされているバックエンドは、oneAPI レベルゼロのみです。また、実験的な拡張機能は、今後の変更により API の互換性が失われる可能性があることにも注意してください。

この記事の例は、コマンドグラフの基本的な使い方を説明し、コマンドグラフを使用するため既存のコードを変更する方法を示すことを目的としています。API や例についての詳細な情報は、拡張機能の仕様 (英語) を参照してください。

本リリースの機能拡張ではパフォーマンスの向上は期待できませんが、今後のリリースでは潜在的なパフォーマンス向上を実現できるよう取り組んでいきます。コマンドグラフ内のカーネルの実行時間も、ホストの起動レイテンシーの短縮によるパフォーマンス向上に影響します。このレイテンシーはほぼ固定されているため、相対的な効果は実行時間の短いカーネルで最も顕著になります。

2024.0 でサポートされている SYCL* 機能:

- カーネル送信
- メモリーコピー操作
- USM およびバッファーサポート

今後の取り組み

拡張機能の仕様 (英語) では、今後実装が予定されているいくつかの機能の概要を示しています。いくつかの機能はすでに十分に具体化されており、いくつかはより概念的なものであり、変更される可能性があります。仕様の将来のバージョンでは、コマンドグラフの機能を拡張するため、これらの機能を洗練させ、含めることに重点が置かれます。

今後の実装における優先課題は次のとおりです。

- 送信間にグラフの入出力引数を変更する更新機能の追加
- SYCL* の機能/ノードタイプのサポート範囲の拡大
- DPC++ での CUDA* バックエンドのサポート

前述のように、コマンドグラフの性質は、カーネル・フュージョンを含むグラフ全体の最適化を可能にします。現在進行中の既存のカーネル・フュージョン拡張をコマンドグラフで使用できるようにするレイヤード拡張機能の開発については、GitHub* (英語) で確認できます。

関連情報

- GitHub* で公開されている完全な仕様 (英語)
- IWOCL & SYCLCON 2023 の講演「SYCL* コマンドグラフの遅延実行に向けて」の録画 (英語)
- GitHub* で公開されているグラフ・フュージョン拡張に関する提案 (英語)
