ユーザーカーネルの融合

この記事は、Codeplay developer Blogs で公開されている「User-driven Kernel Fusion」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。

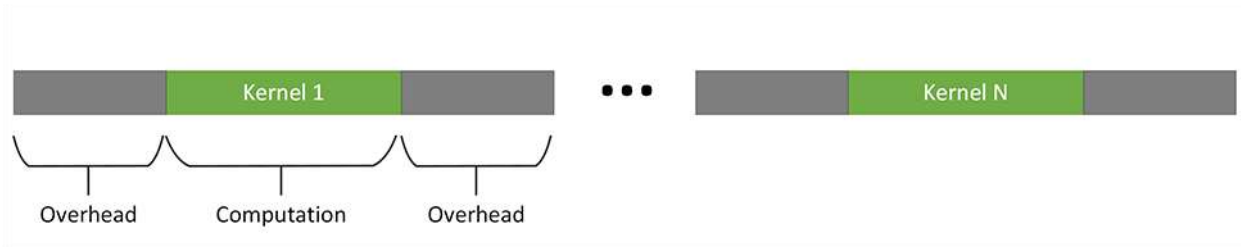
はじめに

アクセラレーター・オフロードに関連するオーバーヘッドは、実行時間の短いデバイスカーネルでは問題になることがあります。複数の小さなカーネルを 1 つに融合することでこの問題を解決できますが、手動で実装する場合、カーネルの組み合わせごとに作業を繰り返す必要があるため面倒です。そこで Codeplay は、ユーザー主導で自動的にカーネルを融合する SYCL* 標準の拡張機能を開発しました。この記事では、この拡張機能について説明し、簡単な例を使って、SYCL* ランタイムにカーネル融合を自動的に行うように指示する方法を示します。

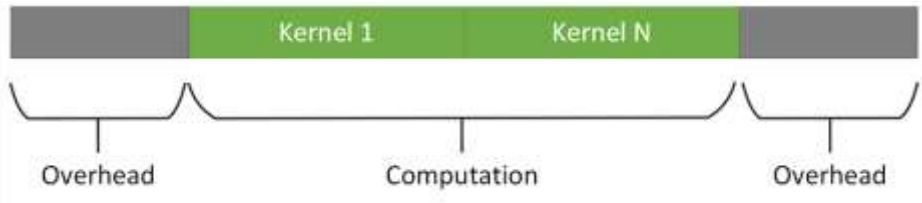
目的

多くの計算タスクは、効率的なソリューションを実現するため、ヘテロジニアス・コンピューティングを必要とします。GPU のようなアクセラレーターは、例えば、タスクを並列に処理して計算時間を大幅に短縮することができ、SYCL* などのプログラミング・モデルは、開発者が GPU のパワーを簡単に利用できるようにします。

しかし、アクセラレーターへのオフロードにはコストが伴います。アクセラレーター上でカーネルを起動するたびに、同期やデータ転送など、タスクの実行に関連したオーバーヘッドが発生します。オーバーヘッドの負荷はシステム構成に依存しますが、アクセラレーターの実行によってもたらされるスピードアップよりも、オーバーヘッドのほうが大きくなる可能性があります。次の図に示すように、実行時間の短いデバイスカーネルは、特にこの影響を受けやすくなります。



これを解決する 1 つの方法は、複数の小さなカーネルを 1 つの大きなカーネルに融合することです。融合カーネルは、次の図に示すように、計算量 / オーバーヘッドの比率が向上するため、デバイスカーネルの起動オーバーヘッドを相殺しやすくなります。



しかし、手動で融合カーネルを実装するのは、面倒でエラーが発生しやすい作業です。また、結果として得られる融合カーネルは、それを構成する個々のカーネルよりも再利用性が低くなる可能性があります。

拡張

この問題に対応し、手動で実装する負担を軽減するため、Codeplay は SYCL* プログラミング・モデルの拡張機能を開発しました。

この拡張機能を利用して、ユーザーがいつ、どのカーネルを融合するかを決定すると、SYCL* ランタイムで自動的にカーネルの融合が行われます。この拡張機能は、シンプルな設計により、最小限のコード変更でアプリケーションのデバイスカーネルの融合を可能にします。

拡張 API の基本的な考え方は非常にシンプルです。SYCL* の `queue` は、新しい API 関数によって「融合モード」と呼ばれる状態になります。`queue` が融合モードの間、`queue` に投入されたカーネルは、実行のためすぐに SYCL* ランタイム・スケジューラーに渡されるのではなく、融合するカーネルのリストに追加されます。融合するカーネルがすべてリストに追加されると、融合モードは終了し、別の新しい API 関数により、融合モードで収集されたカーネルのリストから新しい融合カーネルが作成され、個々のカーネルに渡された引数でそのカーネルが実行されます。

融合カーネルの作成 (コンパイル) は、ジャストインタイム (JIT) コンパイラーにより、SYCL* アプリケーションの**実行時**に (「オンライン」で) 行われます。この拡張機能によって、SYCL* アプリケーションの静的コンパイルステップが変わることはありません。

既存の SYCL* アプリケーションでカーネル融合を行うには、以下の手順を実行する必要があります。

1. SYCL* の `queue` 作成時に、追加プロパティー `sycl::ext::codeplay::experimental::property::queue::enable_fusion` をキューのプロパティー・リストに渡す必要があります。

   ```
   queue q{gpu_selector{},
   {ext::codeplay::experimental::property::queue::enable_fusion()}};
   ```

2. 融合 API を利用するため、`fusion_wrapper` オブジェクトを作成します。

   ```
   ext::codeplay::experimental::fusion_wrapper fw{q};
   ```

3. 融合カーネルに含める最初のカーネルを `queue` に投入する前に、`start_fusion` を呼び出して融合モードに入ります。

   ```
   fw.start_fusion();
   ```

4. 通常と同じようにカーネルを `queue` に投入します。この例では、融合拡張を使用するためコードを変更する必要はありません。
5. 融合カーネルに含めるカーネルをすべて投入したら、融合モードを終了し、`complete_fusion` を呼び出して融合カーネルの作成と実行をランタイムに指示します。

   ```
   fw.complete_fusion();
   ```

融合を行わず、個々のカーネルとして実行する場合は、`complete_fusion` の代わりに `cancel_fusion` を呼び出します。この場合、融合モードは終了し、融合リスト内のカーネルは SYCL* ランタイム・スケジューラーに渡され、1 つずつ実行されます。

この記事で説明する API 拡張は、DPC++ daily 2023-02-04 以降のリリースに含まれています。次のセクションの使用例では、拡張機能をアプリケーションに統合する方法を説明していますが、実際にほとんど変更を加えることなく、既存の SYCL* アプリケーション・コードにカーネル融合を統合できることを示します。

使用例

この記事では、以下の単純な SYCL* アプリケーションの例を使用します。

```
#include <sycl/sycl.hpp>
using namespace sycl;

  constexpr size_t dataSize = 100&amp;apos;000&amp;apos;000;
  int main() {
    // [...] Data initialization

    queue q{ext::codeplay::experimental::property::queue::enable_fusion{}};

    {
      buffer<float> bIn1{in1.data(), range{dataSize}};
      buffer<float> bIn2{in2.data(), range{dataSize}};
      buffer<float> bIn3{in3.data(), range{dataSize}};
      buffer<float> bIn4{in4.data(), range{dataSize}};
      buffer<float> bOut{out.data(), range{dataSize}};
      buffer<float> bTmp1{range{dataSize}};
      buffer<float> bTmp2{range{dataSize}};
      buffer<float> bTmp3{range{dataSize}};

      ext::codeplay::experimental::fusion_wrapper fw{q};
      fw.start_fusion();

      // tmp1 = in1 * in2
      q.submit([&](handler &cgh) {
        auto accIn1 = bIn1.get_access(cgh);
        auto accIn2 = bIn2.get_access(cgh);
        auto accTmp1 = bTmp1.get_access(cgh);
        cgh.parallel_for<class KernelOne>(
          dataSize, [=](id<1> i) { accTmp1[i] = accIn1[i] * accIn2[i]; });
      });

      // tmp2 = in1 - in3
      q.submit([&](handler &cgh) {
        auto accIn1 = bIn1.get_access(cgh);
        auto accIn3 = bIn3.get_access(cgh);
        auto accTmp2 = bTmp2.get_access(cgh);
        cgh.parallel_for<class KernelTwo>(
          dataSize, [=](id<1> i) { accTmp2[i] = accIn1[i] - accIn3[i]; });
      });

      // tmp3 = tmp2 * in4
      q.submit([&](handler &cgh) {
        auto accIn4 = bIn4.get_access(cgh);
        auto accTmp2 = bTmp2.get_access(cgh);
        auto accTmp3 = bTmp3.get_access(cgh);
```

```
        cgh.parallel_for<class KernelThree>(
          dataSize, [=](id<1> i) { accTmp3[i] = accTmp2[i] * accIn4[i]; });
      });

      // out = tmp1 - tmp3
      q.submit([&](handler &cgh) {
        auto accTmp1 = bTmp1.get_access(cgh);
        auto accTmp3 = bTmp3.get_access(cgh);
        auto accOut = bOut.get_access(cgh);
        cgh.parallel_for<class KernelFour>(
          dataSize, [=](id<1> i) { accOut[i] = accTmp1[i] - accTmp3[i]; });
      });

      fw.complete_fusion();

      q.wait();
    }

    process_further(out);

    return 0;
}
```

この例では、個々のカーネルは要素ごとの演算を 1 つだけ実行しており、比較的小さい合計 4 回の呼び出しでグラフを形成し、あるカーネルの出力が別のカーネルの入力として機能します。このような単純な演算カーネルのシーケンスは、GPT などのニューラル・ネットワークのワークロードでよく見られます。

このアプリケーションでカーネル融合を有効にするには、ベース・アプリケーションで前述の 4 行のみを変更し、4 つのカーネルのシーケンスを 1 つのカーネルに融合する必要があります。

カーネルを融合しない場合とした場合のパフォーマンスを比較すると、アプリケーションの実行時間は約 353ms から約 339ms に短縮されます [1]。たった 4 行のコードを変更しただけであることを考えれば、悪くないと言えるでしょう。しかし、カーネル融合はさらに大きなパフォーマンス向上をもたらすことができます。

正当性と収益性に関する考察

これまで拡張機能の**使い方**について説明しましたが、**どのような場合に使うか**についても見ていきましょう。

どのカーネルを融合するかはユーザーに任されているため、ユーザーは次の 2 つの基準を評価する必要があります。

- **正当性:**: 融合カーネルでも正しい結果が得られることを確認します。
- **収益性:**: 融合によってパフォーマンスが向上するか確認します。

正当性に関して最も重要なことは、潜在的なデータ競合と同期の問題です。

融合しない場合、2 つの SYCL* デバイスカーネルのシーケンシャル実行の間には暗黙のグローバルバリアが存在します。つまり、2 つ目のカーネル内のすべてのワークアイテムの実行時に、1 つ目のカーネル内のすべてのワークアイテムによるデータの更新は確定されています。

一般に、JIT コンパイラーが融合カーネルに挿入できるような、デバイスコードをデバイス全体で同期するデバイス関数は存在しないため、融合カーネルではデバイス全体の同期は不可能です。

2 つ目のカーネルのワークアイテムが 1 つ目のカーネルの同じワークグループ内のワークアイテムの更新にのみ依存している限り、データ競合は発生せず、JIT コンパイラーはワークグループ内の同期を保証するため、ワークグループ・バリアを自動的に挿入できます。

ただし、2 つ目のカーネルのワークアイテムが、1 つ目のカーネルの別のワークグループ内のワークアイテムによる更新に依存する場合、融合カーネルで同期の問題が発生し、RAW (リードアフターライト)、WAR (ライトアフターリード)、WAW (ライトアフターライト) の依存関係が発生する可能性があります。ユーザーは、融合リストに追加する前に、デバイスカーネルのアクセスパターンを調査して、カーネル融合に参加するカーネルがそのようなワークグループ間同期を必要としないことを確認する必要があります。

暗黙のデバイス全体の同期に依存する 2 つのカーネルの例を次に示します。

```
q.submit([&](handler &cgh) {
  auto accIn1 = bIn1.get_access(cgh);
  auto accIn2 = bIn2.get_access(cgh);
  auto accTmp = bTmp.get_access(cgh);
  cgh.parallel_for<class KernelOne>(
    nd_range<1>{range<1>{dataSize}, range<1>{8}}, [=](item<1> i) {
      size_t index = (i.get_range(0) - i) - 1;
      accTmp[index] = accIn1[i] + accIn2[i];
  });
});

q.submit([&](handler &cgh) {
  auto accTmp = bTmp.get_access(cgh);
  auto accIn3 = bIn3.get_access(cgh);
  auto accOut = bOut.get_access(cgh);
  cgh.parallel_for<class KernelTwo>(
    nd_range<1>{range<1>{dataSize}, range<1>{8}},
    [=](id<1> i) { accOut[i] = accTmp[i] * accIn3[i]; });
});
```

この例の 2 つ目のカーネルでは、ワークアイテム `i` は 1 つ目のカーネルでインデックス `dataSize - i` を持つワークアイテムの結果を必要とします。ワークグループ・サイズは 8 であるため、ほとんどのワークアイテムでは、これはほかのワークグループのワークアイテムとの同期が必要になります。

しかし、融合が許可されており、融合カーネルが正しい結果をもたらす場合であっても、融合がパフォーマンス上常に**有益**であるとは限りません。例えば、融合カーネルによりレジスター・プレッシャーが増加することで、パフォーマンスが低下する可能性があります。どのような場合に融合が有益であるかを決定する簡単で一般的な経験則はありませんが、拡張 API はシンプルであるため、既存のアプリケーションに統合し、融合した場合としない場合のパフォーマンスを容易に比較できます。

カーネル融合によってアプリケーションのパフォーマンスが向上するかどうかを評価する方法は、次のセクションで追加のガイドラインを示します。

上記の使用例では、カーネル融合によるスピードアップが見られましたが、その幅はかなり小さいため [1]、融合カーネルによりさらにパフォーマンスを向上できるかという疑問が残ります。

データフローと内部化

結論から言うと、可能です。融合カーネルを構成する個々のカーネル間のデータフローを最適化することで、パフォーマンスを大幅に向上できます [1]。

2 つ以上のカーネルが連携して最終結果を生成する場合、あるカーネルが生成し、後続の別のカーネルが消費する中間結果を通信する方法は 1 つしかありません。中間結果を生成するカーネルは、次のカーネルがロードできるように、中間結果をグローバルメモリーに格納する必要があります。カーネルが書き込み可能なほかのすべてのメモリー (レジスター、プライベート・メモリー、ローカルメモリー) は、カーネルの実行中のみ内容が保持されるため、グローバルメモリーよりもかなり高速にアクセスできるものの、カーネル間のデータの受け渡しには使用できません。

しかし、カーネル融合では、2 つのカーネルは 1 つの融合カーネルとして実行されるため、プライベート・メモリーやローカルメモリーを使用することが可能です。そのため、融合カーネルでは、先行する領域 (融合されるカーネルの 1 つ) は結果をグローバルメモリーに書き込む代わりに、はるかに高速なプライベート・メモリーまたはローカルメモリーに格納し、後続の領域 (後続のカーネル) はグローバルメモリーからコストの高い読み取りを行うことなく、そこにアクセスできます。この記事では、データフローを融合カーネルに内部化することから、このプロセスを**内部化**と呼びます。

上記の例に戻ると、内部化できるデータフローを簡単に特定できます。バッファー `bTmp1`、`bTmp2`、`bTmp3` に格納された一時的な結果はすべて内部化できます。融合カーネルの実行が完了した後、アプリケーションがこれらの中間結果にアクセスすることはなく、`out` の最終結果だけがホスト・アプリケーションによって必要とされます。

拡張機能を使用してデータフローを自動的に内部化する方法

カーネル融合の SYCL* 拡張は、融合機能だけでなく、SYCL* ランタイムにデータフローを自動的に内部化するように指示するメカニズムを提供します。

JIT コンパイラーは、次の 3 つの必要条件を満たす場合に内部化を行います。

1. 最初のカーネルが書き込みに使用するアクセサーは、2 つ目のカーネルが読み取りに使用するアクセサーと同じバッファーを参照している。
2. 最初のカーネルによって生成されるデータは、融合に参加しない第 3 のカーネルによって必要とされない。
3. データフローをプライベート・メモリーまたはローカルメモリーに内部化することで、データ競合や不正な動作を引き起こさない。

最初の条件は、SYCL* ランタイムと JIT コンパイラーによって自動的に確定され、ユーザーの判断は必要ありません。

2 つ目の条件は、SYCL* ランタイムによって確定できません。これは、`complete_fusion` の呼び出しが発生した時点では、最初のカーネルからのデータを使用する 3 番目のカーネルがアプリケーションによって送信されていない可能性があるためです。したがって、この条件はユーザーの判断を必要とします。

3 つ目の条件も、将来 JIT コンパイラーがより高度な分析機能を備えるまでは、ユーザーの判断を必要とします。プライベート・メモリーへの内部化を行うには、カーネルのワークアイテムによって書き込まれたメモリー位置は、後続のカーネルの同じワークアイテムによってのみアクセスされなければなりません。ローカルメモリーへの内部化については、この条件を緩和し、カーネルのワークアイテムによって書き込まれたメモリー位置は、後続のカーネルの同じワークグループ内のワークアイテムによってのみアクセスされなければならない、とすることができます。

ユーザーは、アプリケーションとカーネル実装の知識を基に、SYCL* ランタイムに、2 つの内部化のうちどちらが許可されているかを伝えることができます。

SYCL* 拡張は、2 つ目と 3 つ目の必要条件に対する回答を組み合わせて、ユーザーが希望する内部化を簡単に指定できるようにします。`sycl::ext::codeplay::experimental::property::promote_private` プロパティーを指定すると、SYCL* ランタイムと JIT コンパイラーはバッファーのプライベート・メモリーへの内部化を実行できるようになります。`sycl::ext::codeplay::experimental::property::promote_local` プロパティーも同様に動作し、ローカルメモリーへの内部化を可能にします。

どちらのプロパティーも、バッファーの構築時にプロパティー・リストに追加してバッファーを内部化することも、アクセサーで指定してより細かく制御することもできます。その場合は、同じバッファーを参照する**すべて**のアクセサーで指定する必要があります。すべてのアクセサーにこのプロパティーがない場合、JIT コンパイラーは内部化を行いません。

前述のように、この使用例では、`bTmp1`、`bTmp2`、`bTmp3` が内部化に適した候補です。これらは、融合カーネルが実行を完了した後、アプリケーションによってアクセスされることはなく、アプリケーションのワークアイテムは互いに完全に独立しているため、プライベート・メモリーに内部化するためのすべての必要条件を満たしています。

この場合、内部化をランタイムに指示する最も簡単な方法は、プロパティーをバッファーに直接アタッチすることです。

この例では、3 つのバッファーの定義を以下のように変更します。

```
buffer< float> bTmp1{
  range{dataSize},
  {sycl::ext::codeplay::experimental::property::promote_private{}}};
buffer< float> bTmp2{
  range{dataSize},
  {sycl::ext::codeplay::experimental::property::promote_private{}}};
buffer< float> bTmp3{
  range{dataSize},
  {sycl::ext::codeplay::experimental::property::promote_private{}}};
```

この簡単なコード変更により、JIT コンパイラーは、中間結果をバッファーのグローバルメモリーに格納/ロードする代わりに、データフローを内部化します。

再度、パフォーマンス・ベンチマークを実行すると、実行時間は約 353ms (融合なし) から約 272ms になり、22% 以上も短縮されました [1]。

ニューラル・ネットワークなどでは、同じカーネルシーケンスを複数回実行することがよくありますが、これによりパフォーマンスがさらに向上します。この例のカーネルシーケンスをループ内で実行し、2 回目以降の反復に注目すると、1 回の反復あたりの実行時間が 230ms から 88ms に短縮され、2.6 倍もスピードアップします [1]。

なぜでしょうか? 融合しない場合、デバイスドライバーのデバイスバイナリーはキャッシュされるため、2 回目以降の反復では、SPIR-V* からデバイスバイナリーへの変換は発生しません。

融合した場合、JIT コンパイラーで実行時にカーネル融合を行うことでオーバーヘッドが発生しますが、JIT コンパイラーはキャッシュメカニズムを採用しているため、2 回目以降は高コストの JIT コンパイルは必要なく、融合カーネルの SPIR-V* からデバイスバイナリーへの変換もデバイスドライバーにキャッシュされます。

そのため、カーネル融合の拡張は、カーネルシーケンスの繰り返しに適用すると、さらにパフォーマンスを向上します。

今後の展望

この記事では、Codeplay が開発したカーネル融合の SYCL* 拡張機能を紹介しました。この拡張機能を使用することで、既存の SYCL* アプリケーションにカーネル融合を簡単に統合し、ユーザーの指示に基づいてカーネル融合を自動化することで、小さなデバイスカーネルの高いオーバーヘッド・コストを回避できます。

現在はまだ実験的な機能ですが、Codeplay は将来的に API と拡張機能の向上を目指しており、API は将来進化することが予想されます。最新の情報と拡張機能のセマンティクスに関する詳細は、拡張機能の提案 (英語) を参照してください。

この拡張機能により、複数の小さなカーネルから SYCL* アプリケーションを構成することが可能になり、開発者は多くの労力を費やさなくても優れたパフォーマンスを実現できます。将来的には、小さなカーネルのライブラリーを使用してより大きなアプリケーションを構築し、異なるアプリケーションでこれらのモジュール化されたコンポーネント・カーネルを再利用できるようにすることで、新しいスタイルの並列およびヘテロジニアス・プログラミングを促進できるでしょう。