# SYCL* コードのよくある落とし穴を調査

この記事は 2023 年 8 月 31 日に Codeplay のウェブサイトで公開された「My Summer of Bad Code: an Investigation of Common Pitfalls When Writing SYCL Code」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

新しいプログラミング言語を学ぶとき、ほとんどの場合、基本的なプログラムの書き方や実行方法についての入門チュートリアルが用意されています。しかし、このようなチュートリアルでは、してはいけないことや、悪い習慣とそれを見つけて回避する方法についての記載がほとんどありません。Codeplay でのインターンシップの間、私は SYCL* というヘテロジニアス・コンピューティング向けの C++ プログラミング・モデルを扱い、特に SYCL* の失敗分類法 (エラーなしでコンパイルされたにもかかわらず、実行すると期待どおりに動作しないテストのコレクション) の作成に取り組みました。目標の 1 つは、より堅牢でバグのない SYCL* コードの書き方を教えることでしたが、初めて学習する場合、これは必ずしも明確ではありません。

自分が SYCL* を学んだ際の経験だけでなく、インターネットや同僚などからさまざまなアイデアを得て、SYCL* の知識レベルを問わず、誰もが無意識のうちにコードに導入してしまう可能性のあるエラーを正確に表すテストを作成することを目指しました。気付かないうちにエラーにつながるようなミスや、すぐにデバッグできないようなエラーに注目しました。例えば、1 つの wait 文を記述し忘れただけで、データ競合や機能のわずかな誤用が発生しますが、その影響はコードがテストされたデバイスとは異なるデバイスで実行されるまで明らかにならないことがあります。

テストには、インテル® oneAPI DPC++/C++コンパイラー 2023.1.0 と、必要に応じて ComputeCpp CE 2.11.0 (現在は提供終了) を使用しました。利用したハードウェアは、インテル® UHD グラフィックス 770 1.3 GPU とインテル® レベルゼロ・バックエンド、第 12 世代インテル® Core™ i9-12900K プロセッサーとインテルの OpenCL* バックエンドです。

## 目的

なぜ私たちは優れた SYCL* コードを書くことにこだわるのでしょうか? ソフトウェアと自動化があらゆる産業で一般的になるにつれて、自動車や航空電子工学のような安全性が最優先される産業では、高度な機能の自動化にもコードが使用されるようになっています。これらの機能を自動化するアルゴリズムは非常に複雑であることが多く、実装には高レベルの API と十分な物理リソースが必要になるため、従来の組み込みプロセッサーから最新のヘテロジニアス半導体へと移行しています。これより、SYCL* などの高レベルのヘテロジニアス・コンピューティング API は、セーフティー・クリティカルなアプリケーションに適応し、より高い計算能力に対する業界の要求に応えることが求められています。

このような理由から、セーフティー・クリティカル・システム向けの初のマルチプラットフォーム対応の高レベルのコンピューティング API を作成するため、SYCL* API のセーフティー・クリティカル・バージョンである SYCL* SC の開発が始まりました。セーフティー・クリティカルなコードのバグは人命を危険にさらす可能性があるため、慎重に記述されていることを保証することが重要です。セーフティー・クリティカルなコードの開発では、単にバグの数を減らすだけでは不十分です。決定論的な実行、包括的なエラー処理、明確な API などが必要とされ、これらはすべて SYCL* SC の目指すところでもあります。しかし、これは開発者の負担をすべて取り除くわけでは

ありません。完璧に設計されたセーフティー・クリティカルな API であっても、開発者が誤用する可能性は十分にあるからです。

良いコードを書く理由は、セーフティー・クリティカルなアプリケーションに限ったことではありません。堅牢で高品質なコードは、すべてのアプリケーションにおいて重要です。SYCL* の基本概念の 1 つはポータビリティーであり、同じアプリケーションを異なる SYCL* 実装のさまざまなデバイス上で実行できるようにすることです。以下の例に示すように、この SYCL* の価値を損なう不注意なコードを開発することは非常に容易です。

## SYCL* のネガティブテスト

以下は、悪い SYCL* コードの例として私が作成したネガティブテストのコードです。それぞれ予期しない動作を引き起こし、原因と早期に発見しなければどのような結果につながるのかを説明しています。

この 4 つの例は、それぞれ異なる種類のミスに起因するもので、テストを大まかな分類に使用しました。1 つ目は同期の例です。SYCL* では、異なるデバイス間で操作を実行する際に、データの場所を確認することが非常に重要です。同期はさまざまな場所で必要になるため、ミスも発生しやすくなります。2 つ目は実装定義の動作の例です。SYCL* 仕様で定義されていない動作なので、実装によって異なる可能性があります。適切に処理されないと問題を引き起こす可能性があり、さまざまな形で現れます。

3 つ目は機能の誤用、つまり、間違った方法で機能を使用している例です。機能の誤用は、未定義の動作につながることがあります。不正なコードでは、クラッシュしたり、エラーが報告されたり、開発者が意図したとおりに動作することもありますが、最悪の場合、開発者が意図した動作とはわずかに異なるものの長い間誰も気付かなかったりなど、何が起こるか分かりません。最後は、暗黙の動作の例です。これは、コードに明示的に記述されていない振る舞いが、プログラムの実行に影響します。

## 同期

以下のコードでは、同じデータにアクセスする 2 つのカーネルが互いに依存関係を持っています。ユーザーは、これらのカーネルが以下に記述された順序で実行され、devicePtrOut には devicePtrIn の元の値の 2 倍の値が格納されることを期待するでしょう。しかし、この依存関係は SYCL* によって明示的にも暗黙的にも保証されていないため、カーネルの実行順序は実装に依存します。これは簡単に修正できます。各コマンドグループの最後に wait 文を追加するか、あるいはアウトオブオーダー・キューではなくインオーダー・キューを使用します。インオーダー・キューを使用することで、カーネルがサブミットされた順に実行されることが保証されます。ただし、一般にインオーダー・キューを使用すると、パフォーマンスが低下する可能性があります。

この例で示したミスは、SYCL* コードを書くときに非常に起こりやすいものです。SYCL* コードを書く際に考慮すべき重要なことの 1 つは、操作が同期されているかどうかです。このミスは、データ競合が発生するあらゆる場所で起こる可能性があり、SYCL* の非同期の性質上、そのような場所は多く存在します。SYCL* は多くの同期機能を提供していますが、ミスが発生する可能性も高まります。例えば、ワークアイテムの同期にはバリアを使用し、操作の同期には wait 文を使用し、メモリーアクセスの同期にはアトミックを使用します。同期は、SYCL* コードを書くときに最も注意しなければならないことの 1 つです。

CPU 上で実行すると、コードは非決定的に動作し、ほとんどの場合は問題なく動作しますが、場合によっては失敗します (テストしたところ、約 100 回に 1 回の割合で誤動作が観察されました)。対照的に、GPU 上ではこの問題を再現できませんでした。この動作の違いについて考えられる可能性は、CPU と比較して GPU では通信

により多くのレイテンシーが発生することです。これは推測であり、実際の DPC++ 実装を研究して得られたものではありません。また、この動作が DPC++ の将来のバージョンでも同じになるという保証もありません。

```
auto queue = sycl::queue{sycl::cpu_selector_v};

auto devicePtrIn = sycl::malloc_device(dataSize, queue);
auto devicePtrOut = sycl::malloc_device(dataSize, queue);

queue.parallel_for(
    sycl::range{dataSize}, [=](sycl::id<1> idx) {
      auto globalId = idx[0];
      devicePtrIn[globalId] = devicePtrIn[globalId] * 2.0f;
    });

queue.parallel_for(
    sycl::range{dataSize}, [=](sycl::id<1> idx) {
      auto globalId = idx[0];
      devicePtrOut[globalId] = devicePtrIn[globalId];
    });

queue.wait();
```

SYCL* の最大の利点の 1 つは、デバイス・ポータビリティーです。プログラムにデバイス固有の動作 (つまり、実行されるデバイスに応じて実行時の動作が異なる) を追加すると、この利点は失われてしまいます。このようなコードを 1 つのデバイス上で開発してテストすると、そのデバイス上では期待どおりに動作する可能性がありますが、別のデバイスで実行すると、デバイス固有の動作により問題が発生します。デバイス固有のコードは、実装定義の動作によって引き起こされることもあります。以下に示すサブグループ・サイズのように、コードが特定のデバイスの特定の機能に依存する場合、すべてのデバイスがその機能をサポートしているわけではありません。

上記の例は、デバイス固有の動作をよく示していますが、コードは仕様に従っていません。以下の例では、オプションの機能 (SYCL* 実装でサポートが必須ではない) を使用しているため、デバイス固有の動作が発生し、デバイス・ポータビリティーが損なわれます。

```
[=](sycl::nd_item<1> item)
  [[sycl::reqd_sub_group_size(4)]] {
    int idx = item.get_global_id(0);
    auto work_group = item.get_group();

    auto res = sycl::reduce_over_group(
        work_group, accA[idx], sycl::maximum<>());

    if (idx == 0) accB[0] = res;
  };
```

このカーネルでは、カーネル属性を使用して特定のサブグループ・サイズを強制しています。特定のサブグループ・サイズがターゲットデバイスでサポートされるかどうかは実装定義ですが、サポートされていない場合は、例外がスローされます。このようなエラーは、コードをターゲットデバイスと同一ではない単一のデバイスでのみテストすると、見落とす可能性があります。

## 実装定義の動作

SYCL* の実装には、DPC++、HipSYCL、ComputeCpp (現在は非推奨) などがあり、それぞれ異なる点がありますが、すべて同じ仕様に準拠しています。これらはすべて実装レベルで異なるため、特定の状況において実装固有の動作が観察されます。

1 つの例が、ComputeCpp と DPC++ が特定のエラーをどのように処理するかです。以下のコードには、ワークグループ・サイズが無効というエラーが含まれています。

```
try {
  sycl::queue queue(sycl::default_selector{});

  queue.submit([&](sycl::handler& cgh) {
    // invalid work group size causes an exception
    auto range = sycl::nd_range<1>(sycl::range<1>(1), sycl::range<1>(10));
    cgh.parallel_for(range, [=](sycl::nd_item<1>) {});
  });

  queue.wait_and_throw();
} catch (sycl::exception const& e) {
  std::cout << "Caught synchronous SYCL exception:\n"
            << e.what() << std::endl;
}
```

このエラーは、この 2 つの実装では現れ方が異なります。DPC++ では、エラーが同期的にスローされるので、try catch ブロックで処理できます。つまり、プログラムがクラッシュすることなく、例外がうまく処理されます。正確なエラーメッセージを以下に示します。

```
Caught synchronous SYCL exception:
Non-uniform work-groups are not supported by the target device -54
(PI_ERROR_INVALID_WORK_GROUP_SIZE)
```

しかし、同じコードを現在非推奨の ComputeCpp でコンパイルすると、エラーは非同期にスローされるため、以下のコードのエラー処理では処理されません。代わりに、std::terminate を呼び出すデフォルトの非同期エラーハンドラーが呼び出され、プログラムがクラッシュします。ComputeCpp のエラーメッセージを以下に示します。

```
ComputeCpp> Warning: Asynchronous exceptions thrown by the runtime but no
async handler provided
ComputeCpp> -> Triggered at: terminate_async_handler.h (32)
ComputeCpp> Warning: Unhandled exception:
    Error: [ComputeCpp:RT0301] Work-group size is invalid (Local size exceeds
the global work group size )
ComputeCpp> -> Triggered at: terminate_async_handler.h (38)
terminate called without an active exception
```

例外が同期的にスローされるか、非同期的にスローされるかは実装定義であるため、どちらの振る舞いも仕様に準拠しています。

この例は、異なる実装間で動作が同じであるとは限らないことを示しており、同じ実装の異なるバージョンでもそうなる可能性があります。例えば、テストして期待どおりに動作したアプリケーションをデプロイした後、バグを修正するため SYCL* 実装を更新したとします。アプリケーションが実装定義の動作に依存している場

合、実装は仕様に準拠したままこの動作を任意の方法で実装できるため、この更新によりアプリケーションの動作が変わる可能性があります。

注: DPC++ と現在非推奨の ComputeCpp のエラー処理の比較から、DPC++では例外が非同期でスローされると思っていた場合を含め、ほとんどすべての例外が同期でスローされるという興味深い発見がありました。

上記の例は、実装定義の動作が問題を引き起こす可能性を示していますが、このような動作が存在する理由はたくさんあります。SYCL* はオープン・スタンダードであり、SYCL* 仕様は誰でも実装できるように公開されています。仕様では、API の高レベルの抽象化は定義されていますが、実装の詳細は実装者に委ねられており、実装定義の動作を可能にしています。直感に反するように思われるかもしれませんが、この柔軟性があるからこそ、実装者は思いどおりに実装でき、実装レベルでのイノベーションが可能です。

## 機能の誤用

機能の誤用は、初めて SYCL* を学習する際に発生しやすいミスです。大きなミスでなくても、間違ったパラメーターを使用するなど、ささいなミスが生じることがあります。

以下のコマンドグループは、B へのすべてのアクセスが確実に同期されるようにアトミックアクセサーを使用して、A のすべての要素を B に加算する単純なカーネルを示しています。個々に示すエラーは小さなものであり、コードを徹底的にテストしないと簡単に見落とす可能性があります。

```
q.submit([&](sycl::handler& cgh) {
        sycl::accessor accA{bufA, cgh, sycl::read_only};
        sycl::accessor accB{bufB, cgh, sycl::read_write};

        cgh.parallel_for(
            sycl::nd_range<1>{dataSize, groupSize}, [=](sycl::nd_item<1> item)
{
              auto id = item.get_global_id(0);

              sycl::atomic_ref
                  atomic_access(accB[0]);
              atomic_access.fetch_add(accA[id]);
            });
      }).wait();
```

アトミックアクセサーは間違ったアドレス空間をターゲットにしているため、データを見つけることができません。CPU をターゲットデバイスとしてテストすると、正しいアクセサーで期待どおりに動作しますが、GPU では失敗します。これは、CPU には GPU のようなローカルメモリーがないため、アクセサーはデフォルトでグローバルメモリーを使用し、バッファーを見つけるためです。

この例は、安全でない動作を引き起こすエラーは必ずしも複雑であるとは限らず、間違ったアドレス空間を使用するなど単純なものでもあり得ることを示しています。この単純なエラーはデバイス固有の動作を引き起こしますが、適切なテストを行っていない場合 (例えば、GPU でテストしていない場合)、見落とされる可能性があります。

SYCL* 2020 仕様で次のように述べられているように、上記のエラーは未定義の動作の一例です。

オブジェクトの実際のアドレス空間と一致しないアドレス空間を指定すると、未定義の動作が発生します。

未定義の動作は、仕様で要件が定義されていないことを意味するため、何が起きてもおかしくありません。未定義の動作には長所と短所がありますが、定義上、何が起こるかを確実に言うことができず、特定の動作が観察されたとしても、それが一定であることを保証できないため、一般的にはそれに依存することは避けるべきです。未定義の動作は非常に興味深いトピックですが、複雑であるためここでは詳しく説明しません。詳しく知りたい場合は、このトピックをさらに深く掘り下げた優れたリソースがオンラインに多数あります。

## 暗黙的な動作

前出の例では、プログラマーによってエラーが引き起こされましたが、次の例では暗黙的な動作により発生します。つまり、明示的に記述されていないものの、記述されたコードの副作用として発生する動作を意味します。

```
auto bufA = sycl::buffer{a, sycl::range{dataSize}};
auto bufB = sycl::buffer{b, sycl::range{dataSize}};
auto bufR = sycl::buffer{r, sycl::range{dataSize}};

queue.submit([&](sycl::handler& cgh) {
      sycl::accessor accA{bufA, cgh, sycl::read_only};
      sycl::accessor accB{bufB, cgh, sycl::read_only};
      sycl::accessor accR{bufR, cgh, sycl::write_only};
      cgh.parallel_for(
          sycl::range{dataSize},
          [=](sycl::id<1> idx) { accR[idx] = accA[idx] + accB[idx]; });
    })
    .wait();

queue.throw_asynchronous();

for (int i = 0; i < dataSize; ++i) {
  assert(r[i] == static_cast(i) * 2.0f);
}
```

上記のコードは、単純なベクトル加算を実行しますが、バッファー/アクセサーの暗黙的な性質により、コードは期待どおりに動作しません。バッファーがスコープ外にならないため、データがホストにコピーバックされず、テストは失敗します。通常、データをホストにコピーバックするタイミングを明示的に記述することはないため、これは見落とされがちです。

## 課題

この記事では、私が書いたいくつかのテストの例を紹介しましたが、これらのテストを作成する際にいくつかの課題に直面しました。

最初の課題は、探しているニッチな分野のテストを見つけることでした。SYCL* 仕様と実装のバグの検索に多くの時間を費やしました。探しているものが見つからなかったり、見つかったとしても、そこに行きつくまでには非常に長い時間がかかりました。SYCL* コードを作成する際のミスは、結局は同じようなものが多いため、古いテストの再構成ではない新しいテストを作成することは予想よりも困難でした。例えば、作成したテストの多くは、待機やバリアなどの同期をプログラムに追加することで修正できます。単に同期がない新しいテストを作成するだけでは、有用ではなく、新しい洞察も得られません。前者はバグを修正するだけでテストは無効になってしまい、後者は SYCL* コードの記述について何も新しいことを教えてくれません。テストに価値を与え、SYCL* 開発を改善するための教訓を学ぶことができるようにするには、これらの中間を見つけることが重要です。

2 つ目の課題は、現実的なテスト、つまり実際のシナリオで発生する可能性のあるコードを見つけることでした。すべてが間違っているテストを作成し、望ましい振る舞いを生み出すことはできますが、実際のシナリオで現れる可能性は非常に低いでしょう。このようなテストから教訓を得て自分のコードに適用することは非常に困難です。

これらの課題を乗り越えて、良いコードと悪いコードの違いを示し、実際に使用できる適切なテストを見つけるのは、楽しい挑戦でした。

## まとめ

SYCL* を学習している人なら誰でも、いつかはこのようなミスを犯す可能性があります。そのため、SYCL* の危険な動作に関する資料は重要です。堅牢なコードを書くことは、コードが意図したとおりに動作するかどうかで人々の安全が左右されるような環境だけでなく、作成したシステムを信頼できるようにするため、すべてのコードにおいてますます重要になっています。SYCL* で発生するミスを分析することで、ミスを引き起こす可能性がある原因をよく理解し、SYCL* 開発をサポートするより良いツールを開発することも可能になります。不具合のある SYCL* コードを分析すればするほど、堅牢で高品質なコードを作成するための要件が理解できるようになります。

---