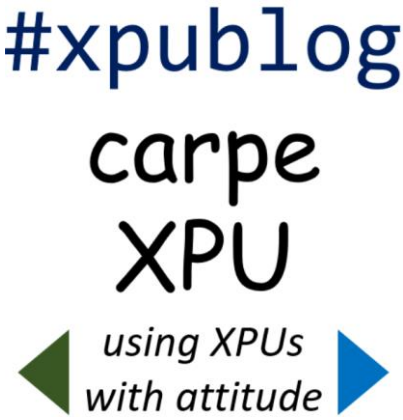
SYCL* キューに関するトピックの続編

この記事は、インテル® デベロッパー・ゾーンに公開されている「SYCL Queues with rest-of-the-story」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原文を優先してください。



キューは、メインプログラムとシステム内の XPU が提供する計算機能との重要なリンクを提供します。キューを使用することで、ワークのスケジュールとデータの移動を調整できます。

今回の #xpublog の記事では、SYCL* キューを確認して、SYCL* キューを最大限に活用する方法 (話の続き) を説明します。

以前の #xpublog の記事 (英語) では、次の 3 つの疑問に答えました。

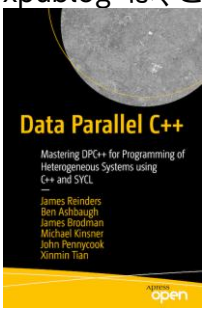
1. XPU とは? XPU が重要な理由は?
2. SYCL* とは? 注目すべき理由は?
3. インテル® DevCloud for oneAPI (無料で簡単に利用可能) を試すには?

記事の最後にあるリンクからフィードバックをお寄せください。ほかの読者のコメントを見ることもできます。フィードバックの内容は、今後の記事に役立てさせていただきます。

キューとは? キューが重要な理由は?

プログラムからデバイス (XPU) を使用するには、キュー・オブジェクトを作成してデバイスへの接続を作成します。キューはシステム内の特定のデバイスに関連付けられます。一度作成されると、キューが接続するデバイスは変更できません。異なるデバイスにコマンドを送る場合は、別のキューを作成する必要があります。必要とする数のキューを作成できます。

『Data Parallel C++』(英語) では、第 2 章「Where Code Executes」でキューの初歩について説明しています。xpublog は、この書籍 (英語) の内容を拡張および補完することを目的としています。この書籍



をお読みになった後は、いくつかの追加項目が参考になります。新しい SYCL* 2020 リファレンス・カード (16 ページ) (英語) は便利なリファレンスです。オンライン DPC++ リファレンス (英語) は、インターフェイスの詳細を知るのに役立ちます。公式 SYCL* 2020 言語仕様 (英語) も参考にしてください。SYCL* 2020 の日本語参考訳はこちらでご覧いただけます。

プログラミングの目的で、実行するデータとコード (カーネル) を入力するものとしてデバイスを扱います。

キューの 6 つのコマンド

キューのタイプは 2 つに分類できます。

1. ワークの「カーネル」を実行するリクエスト
2. カーネルが必要とする、または作成したデータを移動するメモリーコマンド

それぞれ 3 つのアクションがあり、必要なものはキューで利用可能な 6 つのアクションのいずれかで処理されます。

Work Type	Actions (handler class methods)	Summary
Device code execution	single_task	Execute a single instance of a device function.
	parallel_for	Multiple forms are available to launch device code with different combinations of work sizes.
	parallel_for_work_group	Launch a kernel using hierarchical parallelism, described in Chapter 4.
Explicit memory operation	copy	Copy data between locations specified by accessor, pointer, and/or shared_ptr. The copy occurs as part of the DAG, including dependence tracking.
	update_host	Trigger update of host data backing of a buffer object.
	fill	Initialize data in a buffer to a specified value.

Figure 2-21. *Actions that invoke device code or perform explicit memory operations*

キューはデバイスを制御する唯一の方法か?

キューは、ホストとデバイス (XPU) 間の接続機能と考えることができます。

しかし、これは技術的には正確ではないため、「SYCL* キューはデバイスにコマンドを送る唯一の方法である」とは言い切れません。

最初は、「キュー」はデバイスにコードの実行やメモリーの操作を要求するホストからの唯一の接続です。しかし、共有メモリーを設定すると、ホストとデバイスは共有メモリーを通じて対話できるようになります。これは、技術的にはキューがデバイスと通信する唯一の方法ではないことを意味します。

技術的な意味を理解したところで、キューはデバイスを制御する方法であると考えます。

キューはどのデバイスに接続するか?

以前の xpublog (英語) で説明したように、デバイスを指定しないでキューを作成すると、ランタイムは利用可能なデバイスの中から最良と思われるデバイスを選択して割り当てます。SYCL* は、常に利用可能なデバイスが存在することを保証します。システムにアクセラレーターがない場合、デフォルトのデバイスは CPU になります。SYCL* はデバイスが存在することを保証しますが、デバイスドライバーがインストールされていない状況でデバイスにアクセスできないことをランタイムが検出した場合に備えて、実際の製品コードではエラー処理を行うことをお勧めします。

SYCL* を使用すると、ランタイムにデバイスを選択させるだけでなく、キューが接続するデバイスを選択する際に必要なレベルの制御を行うことができます。このプログラムは、実行時に利用可能なリソースを確認し、必要な選択を行うことができます。「GPU で実行」のような一般的なガイダンスを提供することも、必要に応じて具体的なガイダンスを提供することもできます。

制御に関心がない場合のデフォルトデバイス選択

以前の xpublog (英語) で、私の愛猫 (Nermal) の画像を明るくする「Hello SYCL」プログラムを紹介しました。オリジナルコードでは、単純にキューを作成しました。これは、「デフォルトのセレクター」を使用するのと同じです。制御に関心がない場合のアプローチであり、「あるものを使用する」と言っていることになります。

SYCL* は常に利用可能なデバイスが存在することを保証するため、このアプローチは常に動作します。仕様では実装の要件を満たす必要がありますが、ドライバーをインストールしなくてもインストールを完了できるようにすれば、技術的に要件を満たさないようにできます。真に強固なプログラムは、インストールが失敗しない場合でも、キューの作成に関する例外処理を提供します。

制御に少し関心がある場合のデバイス選択

デフォルトのセレクターに加えて、SYCL* では CPU、GPU、またはアクセラレーターを要求できます。アクセラレーターのカテゴリーは、CPU または GPU 以外のデバイスをすべて包括します。DPC++ は、実機の FPGA またはエミュレートされた FPGA を要求するインテルの拡張機能も提供します。FPGA のコンパイル時間は不明であるため、実機の FPGA でコードを確認する前に、エミュレーターを使用して可能な限りデバッグを完了します。

該当するデバイスがシステムに存在しない場合、セレクターはエラーを返します。例えば、GPU セレクターは、GPU が搭載されていないシステムでは失敗します。エラー処理を行うこと (書籍の第 5 章を参照) は、製品レベルのアプリケーションでは重要です。

GPU を要求し、システムに複数の GPU が存在する場合、ランタイムは任意の GPU を選択します。カスタム・デバイス・セレクターを使用しない限り、どの GPU が選択されるか制御することはできません。

制御が必要な場合のデバイス選択

多くの構成が利用可能であることは容易に想像できます。また、実行時に複数の XPU が利用できる場合、デバイス選択の優先順位に非常に具体的なアイデアがあることも容易に想像できます。例えば、USM をサポートしない GPU よりも USM をサポートする GPU を使用する最大のメモリーを搭載した GPU を選択、あるいは GPU よりも FPGA を選択するようにできます。

この「カスタム・デバイス・セレクター」と呼ばれる、特定のデバイスを評価するルーチンを作成すると、ランタイムは評価に基づいてデバイスを選択します。カスタム・デバイス・セレクターについては、書籍の第 12 章で説明しています。ビデオ (英語) の説明もご覧ください。図 12-3 には、この概念を説明する簡単な例が含まれています。メカニズムは単純かつ強力で、完全に制御することができます。

CHAPTER 12 DEVICE INFORMATION

```
class my_selector : public device_selector {
  public:
    int operator()(const device &dev) const {
      int score = -1;

      // We prefer non-Martian GPUs, especially ACME GPUs
      if (dev.is_gpu()) {
        if (dev.get_info<info::device::vendor>().find("ACME")
            != std::string::npos) score += 25;

        if (dev.get_info<info::device::vendor>().find("Martian")
            == std::string::npos) score += 800;
      }

      // Give host device points so it is used if no GPU is available.
      // Without these next two lines, systems with no GPU would select
      // nothing, since we initialize the score to a negative number above.
      if (dev.is_host()) score += 100;

      return score;
    }
};

int main() {
  auto Q = queue{ my_selector{} };

  std::cout << "After checking for a GPU, we are running on:\n "
    << Q.get_device().get_info<info::device::name>() << "\n";

  // Sample output using a system with a GPU:
  // After checking for a GPU, we are running on:
  //  Intel(R) Gen9 HD Graphics NEO.
  //
  // Sample output using a system with an FPGA accelerator, but no GPU:
  // After checking for a GPU, we are running on:
  //  SYCL host device.

  return 0;
}
```

Figure 12-3. *Custom device selector—our preferred solution*

デバイスは互いに通信できるか?

今のところ、デバイスは完全に独立していて、ホストにのみ接続されていると考えます。将来の xpublog では、共有メモリーのようにデバイス間で共有が可能な場合にアクセスできるリソースを管理するのに役立つ SYCL* のコンテキストを説明する予定です。

キューを FPGA エミュレーターに接続する

これまで紹介したことを適用して、以前の xpublog から「Hello SYCL」プログラムに再度アクセスして、FPGA エミュレーターをターゲットにするように変更できます。必要な作業は、FPGA セレクターを使用するようにキューの作成を変更するだけです。

変更前:

```
queue q;
```

変更後:

```
intel::fpga_emulator_selector my_selector;

queue q( my_selector ); // グラフィック・ビデオ・コードが混在していることに注意
```

FPGA 向けのインテルの拡張機能を使用しているため、次の行をインクルードする必要があります。

```
#include <sycl/ext/intel/fpga_extensions.hpp>
```

SYCL* は、ベンダー固有のデバイスタイプを推奨しています。現在の SYCL* 2020 仕様に基づいて FPGA を使用するには、SYCL* の「アクセラレーター」を使用するか (一般的)、FPGA 向けのベンダー拡張機能を使用する必要があります。インテルは、ベンダー拡張機能を作成することを選択し、いくつかの FPGA 固有の機能を提供しています。

オンライン (インテル® DevCloud) で実行してみる

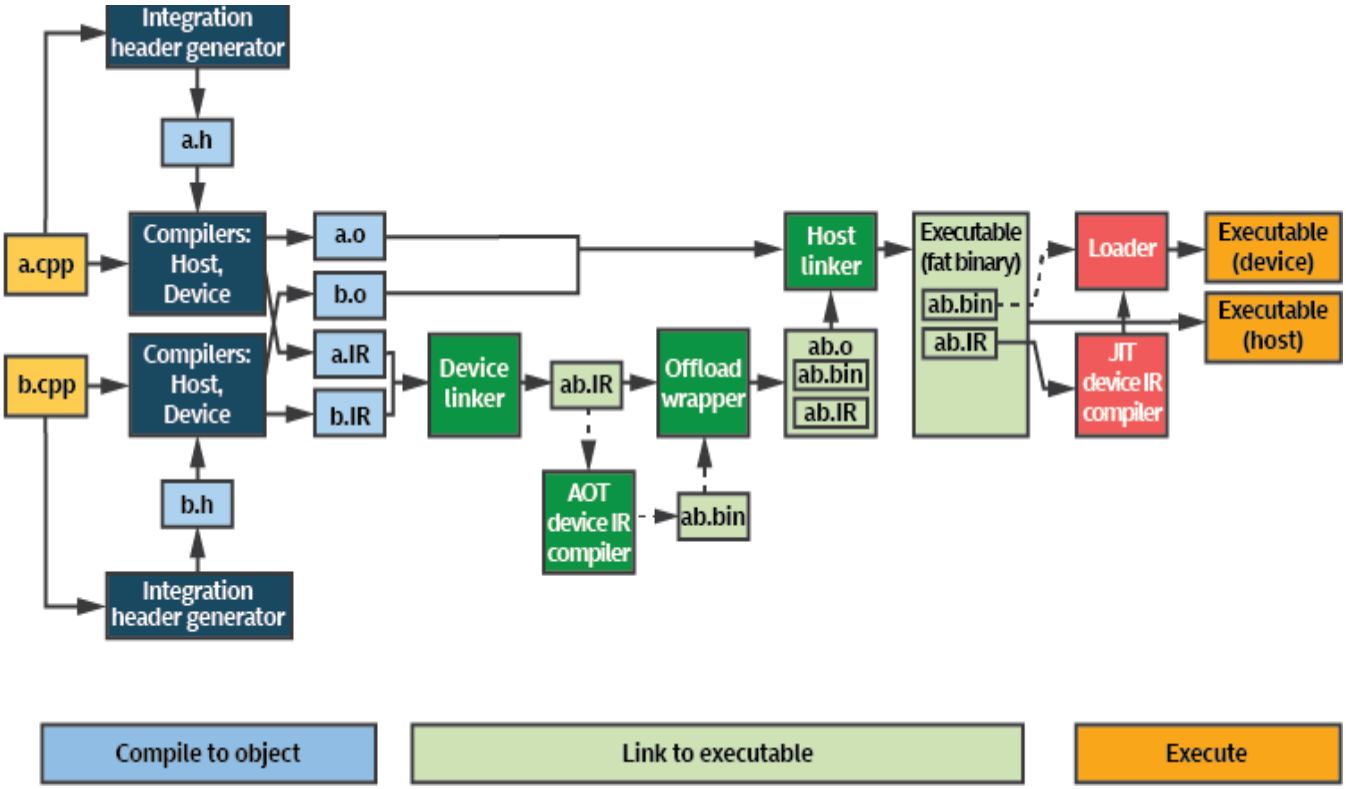
オンラインで試してみることをお勧めします。約 2 分で実際に体験できます。Welcome ノートブックで「xpublog 4」を選択して、Jupyter* Notebook にアクセスします。Welcome ノートブックは、以前の xpublog (英語) のインテル® DevCloud に関する説明の、When returning の手順に従って表示できます。

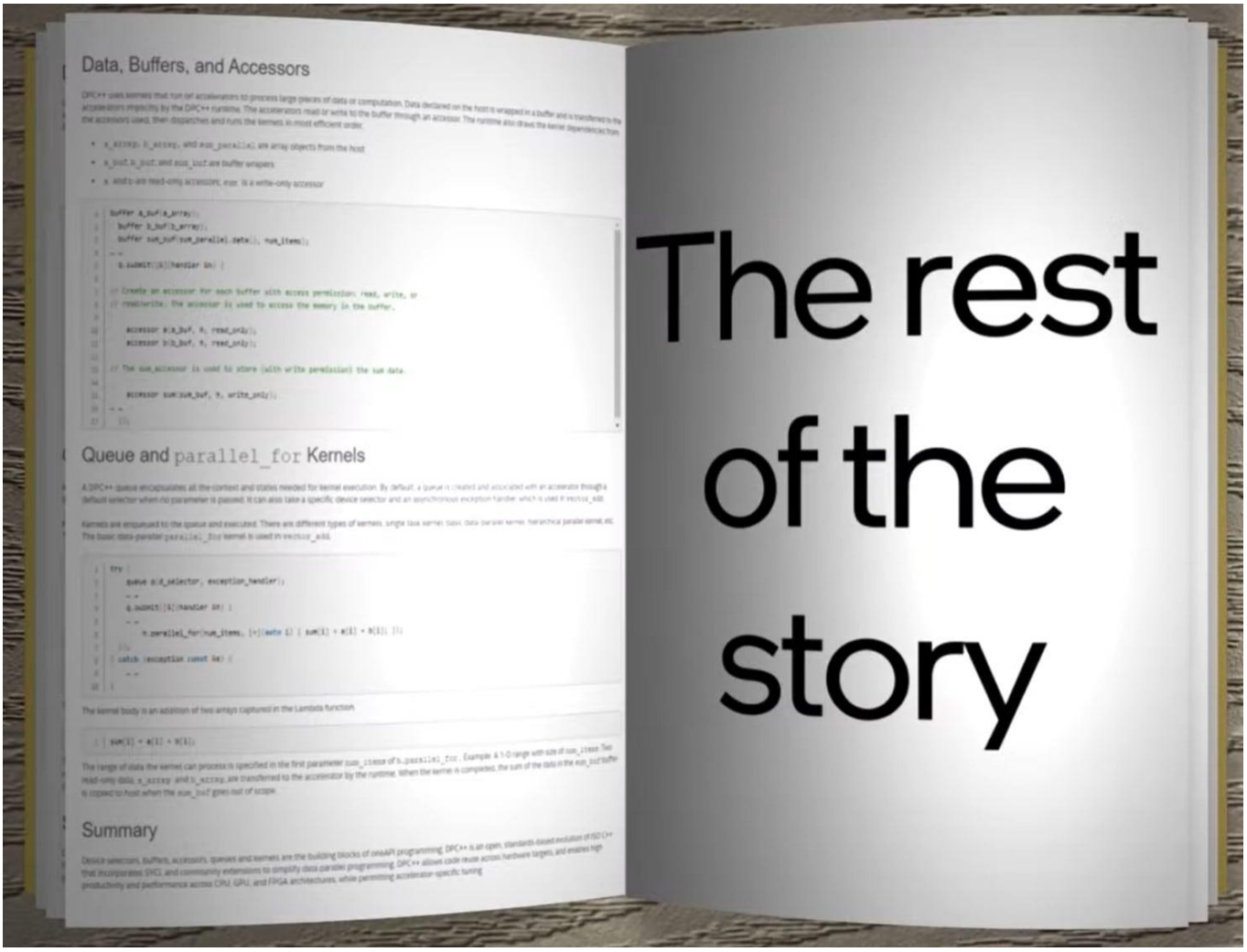
1 つのデバイスに対して静的にプログラムを構築しているため、FPGA をターゲットにしていないときにすばやくコンパイルできます。FPGA をターゲットに含めると、FPGA コンパイルを実行する必要があります。FPGA デバイス向けのコンパイルには 1 時間以上かかります。FPGA エミュレーター・オプションを使用すると、デバッグしながらすばやくコンパイルできます。すぐに確認できるように、このチュートリアルでは、このオプションを使用するように設定しています。

これは、さまざまな XPU に対してワークを指示するプログラムを変更することが容易であるかを示しています。

事前コンパイルに関する注意

ビデオ (英語) では、ほかのデバイスと異なり、FPGA コードは事前にコンパイルする必要があることを説明しています。FPGA のコンパイルは非常に時間がかかる (2 時間以上) ため、プログラムを実行するたびに「ジャストインタイム」で実行できません。
ほかの XPU デバイスに大量のコンパイルプロセスがある可能性もありますが、一般には、ほかの XPU 向けのコードを実行時にデバイス向けにファイナライズします。この処理を、中間形式から利用可能なデバイスで必要なコードへの「ジャストインタイム・コンパイル」と呼んでいます。この過程については、書籍 (英語) の第 13 章で詳しく説明しています。下記の図 13-2 は、高レベルの重要な概念を示しています。





では、話の続きを始めましょう。

キューがどのように動作し、アプリケーションでどのように使用されるか、より明らかにするため、いくつかの質問に対する答えを考えてみましょう。

Q: プログラムが使用するデバイスを強制できますか?

どのデバイスを選択するかはプログラムが制御できますが、プログラムが選択できるデバイスを限定できるのが優れた点です。

これを行う標準的な SYCL* の方法はありませんが、DPC++ 実装から答えが得られます。ほかの SYCL* コンパイラーも同様のメカニズムを備えているかもしれません。

DPC++ では、`SYCL_DEVICE_FILTER` 環境変数を設定して、SYCL* ランタイムが利用可能なデバイスのサブセットのみ使用するように制限することができます。この環境変数は、デバイスクエリー関数 (`platform::get_devices()` および `platform::get_platforms()`) とデバイスセレクターに影響します。

多くのシステムでは、1 つのデバイスが複数のドライバーでサポートされている可能性があります。たとえば、インテルの GPU はレベルゼロと OpenCL* ドライバーをインストールして利用できます。ほかのシステムでは、複数のバージョンのドライバーを同時にインストールできる場合があります。SYCL_DEVICE_FILTER 環境変数は、プログラムが認識するデバイス、およびプログラムが対話するドライバーに影響を与えます。これにより、プログラムがフル・プラットフォームと考えるものを制御できます。

プログラムの実行時に表示されるデバイスを制限していますが、プログラムでデバイスを選択して実行する必要があります。このフィルターで行っているのは、選択肢を制限することだけです。複数の XPU を搭載したシステムでは、プログラムを実行する XPU を制限できます。この方法により、1 つのプログラムをコンパイルした後、特定の XPU で特定のインスタンスを実行できます。

DPC++ LLVM プロジェクトの GitHub* で、この機能の完全なドキュメント (英語) が提供されています。

Q: デバイスの検出および選択プロセスをランタイムで監視できますか?

同じドキュメント (英語) で、デバイスの検出および選択プロセス中のシステムの動作を理解するのに役立つ、非常に優れたトレースオプションが提供されています。予想どおりに動作しない理由が明確でない場合や、プログラムが予想どおりに実行されていることを確認するのに役立ちます。

Q: SYCL* キューとはどのような種類のキューですか?

デフォルトのキュースタイルは FIFO ではありません。代わりに、アイテムをキューに入れると、依存関係が維持されている限り、キューのほかのアイテムに対して任意の順序で実行できます。バッファーを使用すると暗黙的に依存関係が保持されます。ほかの依存関係 (共有メモリーなど) は明示的に管理する必要があります。詳細は、書籍 (英語) の第 8 章で説明しています。

SYCL* は、デフォルトのキューを「インオーダー」プロパティーで拡張した場合、インオーダー・キューをサポートします。

Q: 複数のキューを使用するほうが 1 つのキューを使用するよりも優れていますか?

複数のキューを使用するほうが 1 つのキューを使用するよりも便利で、コードの理解と保守が容易になる場合があります。

同じデバイスに複数のキューを使用するほうが優れている 4 つのケースを次に示します。

1. プログラムに複数のスレッドまたは複数のランクがある場合、単一キューの代わりにプログラミングを簡単に行うため、複数のキューを使用することをお勧めします。
2. 複数のキューにより多くの並列処理を表現できる場合は、複数のキューの使用が適している可能性があります。通常はデフォルトのキュータイプを使用することをお勧めしますが、同時実行の可能性がある場合は、複数の順序付きキューの使用を検討してください。

3. 同期の必要がある場合は、キューで待機することができます。この待機では、キューに送信されたすべてを続行する前に待機する操作を完了する必要があります。実際に待機する操作をより細かく制御したい場合は、複数のキューを使用して、特定のキューに発行された操作のみを待機するようにできます。これは、単一のキューに送信する操作のサブセットを待機できる優れた方法です。
4. デバイスの異なる部分に異なるコマンドを発行する場合は、複数のキューを使用します。SYCL* はサブデバイスの概念をサポートしており、各サブデバイスに 1 つ以上のキューを接続できます。デバイスを分割すると、複数のキューを使用して、デバイス全体のどの部分でワークを実行するか決定できます。書籍 (英語) では、サブデバイスについて説明していません。将来の xpublog および書籍の次回の改訂版で説明します。

同じデバイスに複数のキューを使用するほうが、アプリケーションで有用であるか判断すると良いでしょう。最適なものを選択するのは皆さん自身です。

Q: 複数のキューを使用すると 1 つのキューを使用するより状況が悪化することはありますか?

複数のキューがコンテキストを共有していると仮定すると、複数のキューを使用しても 1 つのキューを使用するより状況が悪化することはありません。コンテキストについては、将来の xpublog で説明します。コードで明示的にコンテキストを使用していなければ、可能であれば、実装はデフォルトでキュー間でコンテキストを共有します。

複数キューの導入により低速であると判断される場合は、次の 2 つを調査します。

1. コンテキストの作成にはコストがかかるため、スレッドごとに過剰な数のコンテキストを作成していないことを確認します。
2. 複数のパスは、理論的には、ソフトウェア・スタックのより複雑な相互作用であるコマンド・バッファリング (「バッチ」) の影響を受ける可能性があります。詳細は、プラットフォーム最適化ガイド (英語) を参照してください。ほかのレベルのアプリケーション・チューニングをまだ完了していないのであれば、この問題に先に取り組む必要はありません。この点も、将来の xpublog で説明する予定です。

Q: キューにデバイスの一部のみを割り当てることはできますか?

はい可能です。SYCL* ではこれをサブデバイスと呼んでいます。

デバイスは、複数のキューで分割できます。デバイス全体にキューを設定することもできます。サブデバイスはデバイスと同じように動作し、さらに分割することができます。将来の xpublog で、コンテキスト、マシンとアプリケーションの抽象化モデルについて説明すると、この使用法は明確になるでしょう。

これは強力なメカニズムであり、非常に便利です。

書籍 (英語) では、サブデバイスについて説明していません。将来の xpublog および書籍の次回の改訂版で説明します。

Q: 複数のアプリケーションでデバイスを共有することはできますか?

1 つのホストで複数の SYCL* アプリケーションを実行する場合、各アプリケーションは XPU (デバイス) に 1 つ以上の接続を確立します。複数のアプリケーションを実行する場合、デバイスを共有することになります。

SYCL* では、デバイスがビジーであることを照会する、特定のポータブルな API は定義されていません。たとえ定義されていたとしても、そのような要求が将来も継続するのか、未来を見たいものです。

前述したように、実装は、フィルターを使用してアプリケーションが「参照」するドライバー (デバイス) を限定することにより、SYCL* プログラムが実行される環境を制御する方法を提供します。これは、どのアプリケーションがどのデバイスを使用するかを制御するために使用でき、指定したレベルが有用であればその後もそれを維持することができます。

Q: キューは複数のデバイスで直接動作しますか?

いいえ。単一の SYCL* キューは、単一のデバイス (またはサブデバイス) にのみディスパッチできます。

キューのように動作してワークを複数のデバイスに分散する抽象化を記述することはできますが、デバイスごとに一意の SYCL* キューを管理する必要があります。このアプローチは、データの配置、移動、アクセスの管理など、複数の課題に直面するでしょう。しかし、独自のアプリケーションのニーズについての詳細な知識があれば、このアプローチを検討することは理にかなっています。

Q: SYCL* 2020 でホストデバイスが削除されたのはなぜですか?

最終的な SYCL* 2020 仕様は、書籍 (英語) の後に登場し、正誤表 (英語) で修正されました。

ホストデバイスの概念は、書籍 (英語) が出版された後に、最終的な SYCL* 2020 仕様で削除されました。これは、書籍に含めたコンテンツで予測しなかった唯一の重要な変更です。

この変更については、GitHub* プロジェクトのコード例とともに管理している書籍の正誤表 (英語) に記載しています。

SYCL* では、常にデバイスが存在する必要があるため、カーネルは常にどこかで実行できます。しかし、SYCL* 2020 では、特定のプロパティーを備えた「ホストデバイス」は削除されました。ターゲットマシンがそのようなデバイスをサポートするとは限らないため、「ホストデバイス」の概念は、すべての組込みプラットフォームに対して適合しませんでした。これは、SYCL* がクロスプラットフォームであることを真剣に考えていることを示す良い例です。

削除されたことは改善であり、ミスではありません。一般に、「ホストデバイス」が必要な場合は、「CPU デバイス」を見つけて使用することをお勧めします。

話の続きはここまでです。

将来の xpublog のトピック

この記事では、SYCL* キューの話の続きと、キューに関するいくつかの質問に答えました。

次の xpublog では、ハードウェアとソフトウェアに関する概念を単純化する方法を検討し、コンテキストについて説明します。この情報は、正当性を判断し、移植可能で保守しやすいアプリケーションを作成するのに役立ちます。SYCL* における明確な考えと、適切にプログラムする方法を知りたい方は必見です。

インテル® DevCloud には、膨大な量の素晴らしいトレーニングが用意されています。詳細は、インテル® oneAPI DevCloud のトレーニング・ページ (英語) を参照してください。最初に、インテル® oneAPI ベース・ツールキットの「View Training Modules」(英語) をクリックして開始することをお勧めします。

コメントをぜひお寄せください

#xpublog の感想、フィードバック、提案などのコメントを community.intel.com James-Reinders-Blog (英語) からぜひ投稿してください。

今後も、経験や開発者同士の共有から生まれる視点や詳細を提供していく予定です。皆さんが興味を持ち、私がお役に立てるトピックを掘り下げていきたいと思っています。

皆さんのご意見、ご感想、そして楽しい議論をお待ちしています。

製品および性能に関する情報

[1] 性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。