インテル® oneAPI プログラミング・ガイド

Intel Corporation
www.intel.com（英語）
著作権と商標について

注意事項:

この日本語マニュアルは、インテル コーポレーションのウェブサイト https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html（英語）で公開されている『Intel® oneAPI Programming Guide』（**バージョン 2022.2、更新日 2022/05/26**）の参考訳です。

インテル社の許可を得て iSUS（IA Software User Society）が翻訳版を作成した iSUS の著作物です。

原文は Intel Corporation の Copyright であり、日本語参考訳版にも適用されます。

目次

著作権と商標について 5

1 はじめに 6
1.1 インテル® oneAPI プログラミングの概要 7
1.2 インテル® oneAPI ツールキットの配布について 8
1.3 関連ドキュメント 9

2 oneAPI プログラミング・モデル 10
2.1 SYCL* を使用したデータ並列 C++ 10
2.1.1 簡単なサンプルコード 10
2.1.2 その他のリソース 12
2.2 C/C++ または Fortran と OpenMP* オフロード・プログラミング・モデル 13
2.2.1 基本的な OpenMP* target 構造 13
2.2.2 Map 変数 14
2.2.3 OMP TARGET を使用するコンパイル 15
2.2.4 OpenMP* オフロードの追加のリソース 16
2.3 デバイスの選択 16
2.3.1 ホストコードでの DPC++ デバイスの選択 17
2.3.2 ホストコードでの OpenMP* デバイスの確認と選択 18

3 oneAPI 開発環境の設定 20
3.1 インストール・ディレクトリー 20
3.2 環境変数 20
3.3 setvars と vars ファイル 21
3.4 モジュールファイル (Linux* のみ) 21
3.4.1 Windows* で `setvars` スクリプトを使用 21
3.4.2 Linux* または macOS* で `setvars` スクリプトを使用 29
3.4.3 Linux* でモジュールファイルを使用 38
3.4.4 oneAPI アプリケーションで CMake* を使用 43

4 oneAPI プログラムのコンパイルと実行 45
4.1 単一ソースのコンパイル 45
4.2 コンパイラーの起動 45
4.3 インテル® oneAPI DPC++/C++ コンパイラーの標準オプション 46
4.4 コンパイル例 46
4.4.1 API ベースのコード 47
4.4.2 ダイレクト・プログラミング 50
4.5 コンパイルの手順 51
4.5.1 伝統的なコンパイル手順 (ホストのみのアプリケーション) 51
4.5.2 SYCL* オフロードコードのコンパイル手順 52
4.5.3 JIT のコンパイル手順 52
4.5.4 AOT のコンパイル手順 53

4.5.5 ファットバイナリー ... 54
4.6 CPU 手順 ... 55
4.6.1 従来の CPU 向け手順 ... 56
4.6.2 CPU オフロード向け手順 ... 56
4.7 GPU 手順 ... 65
4.7.1 GPU オフロードの手順 ... 65
4.7.2 GPU コマンドの例 ... 70
4.7.3 GPU 向けの事前(AOT)コンパイル ... 71
4.8 FPGA 手順 ... 71
4.8.1 FPGA 向けのコンパイルが特殊である理由 ... 72
4.8.2 SYCL* FPGA コンパイルの種別 ... 72
4.8.3 FPGA コンパイルオプション ... 75
4.8.4 設計のエミュレートとデバッグ ... 78
4.8.5 シミュレーションでカーネルを評価 ... 87
4.8.6 FPGA デバイスセレクター ... 93
4.8.7 FPGA 高速再コンパイル ... 94
4.8.8 複数の FPGA イメ―ジを作成(Linux* のみ) ... 99
4.8.9 FPGA BSP とボード ... 102
4.8.10 複数の同種の FPGA デバイスをターゲットにする ... 106
4.8.11 複数のプラットフォームをターゲットにする ... 108
4.8.12 FPGA-CPU インタラクション ... 109
4.8.13 FPGA パフォーマンスの最適化 ... 111
4.8.14 FPGA 向けの RTL ライブラリーを使用する ... 111
4.8.15 サードパーティーのアプリケーションで DPC++ 共有ライブラリーを使用 ... 117
4.8.16 IDE での FPGA ワークフロー ... 123

5 API ベースのプログラミング ... 124
5.1 インテル® oneAPI DPC++ ライブラリー(インテル® oneDPL) ... 125
5.1.1 インテル® oneDPL ライブラリーの使い方 ... 125
5.1.2 インテル® oneDPL サンプルコード ... 125
5.2 インテル® oneAPI マス・カーネル・ライブラリー(インテル® oneMKL) ... 126
5.2.1 インテル® oneMKL の使い方 ... 126
5.2.2 インテル® oneMKL サンプルコード ... 127
5.3 インテル® oneAPI スレッディング・ビルディング・ブロック(インテル® oneTBB) ... 131
5.3.1 インテル® oneTBB の使い方 ... 132
5.3.2 インテル® oneTBB サンプルコード ... 132
5.4 インテル® oneAPI データ・アナリティクス・ライブラリー(インテル® oneDAL) ... 132
5.4.1 インテル® oneDAL の使い方 ... 133
5.4.2 インテル® oneDAL サンプルコード ... 133
5.5 インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー(インテル® oneCCL) ... 134
5.5.1 インテル® oneCCL の使い方 ... 134
5.5.2 インテル® oneCCL サンプルコード ... 135
5.6 インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー(インテル® oneDNN) ... 135
5.6.1 インテル® oneDNN の使い方 ... 136
5.6.2 インテル® oneDNN サンプルコード ... 138
5.7 インテル® oneAPI ビデオ・プロセシング・ライブラリー(インテル® oneVPL) ... 138
5.7.1 インテル® oneVPL の使い方 ... 138
5.7.2 インテル® oneVPL サンプルコード ... 139

5.8 その他のライブラリー 142

6 ソフトウェア開発プロセス 143
6.1 SYCL* と DPC++ へのコードの移行 143
6.1.1 C++ から SYCL* への移行 143
6.1.2 DPC++ コンパイラーを使用した CUDA* から SYCL* への移行 143
6.1.3 OpenCL* コードから SYCL* への移行 144
6.1.4 CPU、GPU、および FPGA 間の移行 144
6.2 コンポーザビリティー 148
6.2.1 C/C++ OpenMP* および SYCL* のコンポーザビリティー 148
6.2.2 OpenCL* コードの相互運用性 151
6.3 DPC++ と OpenMP* オフロード処理のデバッグ 151
6.3.1 oneAPI デバッグツール 152
6.3.2 オフロード処理のトレース 161
6.3.3 オフロード処理のデバッグ 163
6.3.4 オフロードのパフォーマンスを最適化 175
6.4 パフォーマンス・チューニング・サイクル 178
6.4.1 ベースラインの確定 178
6.4.2 オフロードするカーネルの特定 179
6.4.3 カーネルをオフロード 179
6.4.4 最適化 179
6.4.5 再コンパイル、実行、プロファイル、そして繰り返し 181
6.5 oneAPI ライブラリーの互換性 182

用語集 183

著作権と商標について

インテルのテクノロジーを使用するには、対応したハードウェア、特定のソフトウェア、またはサービスの有効化が必要となる場合があります。

絶対的なセキュリティーを提供できるコンピューター・システムはありません。

実際の結果は異なる場合があります。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

最適化に関する注意事項
インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。 注意事項の改訂 #20110804

特に明記されない限り、このドキュメントのサンプルコードは MIT ライセンスの下に次の条件で提供されます。

© 2021 Intel Corporation.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

はじめに

1

現代のコンピューター・アーキテクチャーで高い計算パフォーマンスを達成するには、最適化され、電力効率に優れた、スケーラブルなコードが必要です。従来のハイパフォーマンス・コンピューティング(HPC)を始めとして AI、ビデオ解析、データ解析においてハイパフォーマンスの需要は増え続けています。

現代のワークロードの多様性から、単一のアーキテクチャーですべてのワークロードに対応することは困難になっており、アーキテクチャーも多様化しています。必要とするパフォーマンスを達成するには、CPU、GPU、AI、および FPGA アクセラレーターに配置されたスカラー、ベクトル、行列、および空間(SVMS)アーキテクチャーの組み合わせが求められます。

今日、CPU とアクセラレーター向けのコーディングには、異なる言語、ライブラリー、そしてツールを利用する必要があります。これは、それぞれのハードウェア・プラットフォームは個別のソフトウェア資産を必要とし、異なるターゲット・アーキテクチャー全体ではアプリケーション・コードの再利用が制限されることを意味します。

oneAPI プログラミング・モデルは、データ並列 C++(DPC++)と呼ばれるオープンソース・プログラミング言語と最新の C++ 機能を使用して並列処理を表現することにより、CPU とアクセラレーターのプログラミングを簡素化します。SYCL* 言語は、単一のソース言語でホスト(CPU など)とアクセラレーター(GPU など)のコードの再利用を可能にし、実行とメモリーの依存関係を明確にします。SYCL* コード内のマッピング機能により、ハードウェアまたはハードウェア・セットに最適化されたワークロードを実行するためアプリケーションが移行されます。アクセラレーターを使用できないプラットフォームでも、ホストを利用することでデバイスコードの開発とデバッグを簡素化できます。

oneAPI は、既存の C/C++ または Fortran コードで OpenMP* オフロード機能を使用する CPU およびアクセラレーターのプログラミングもサポートします。

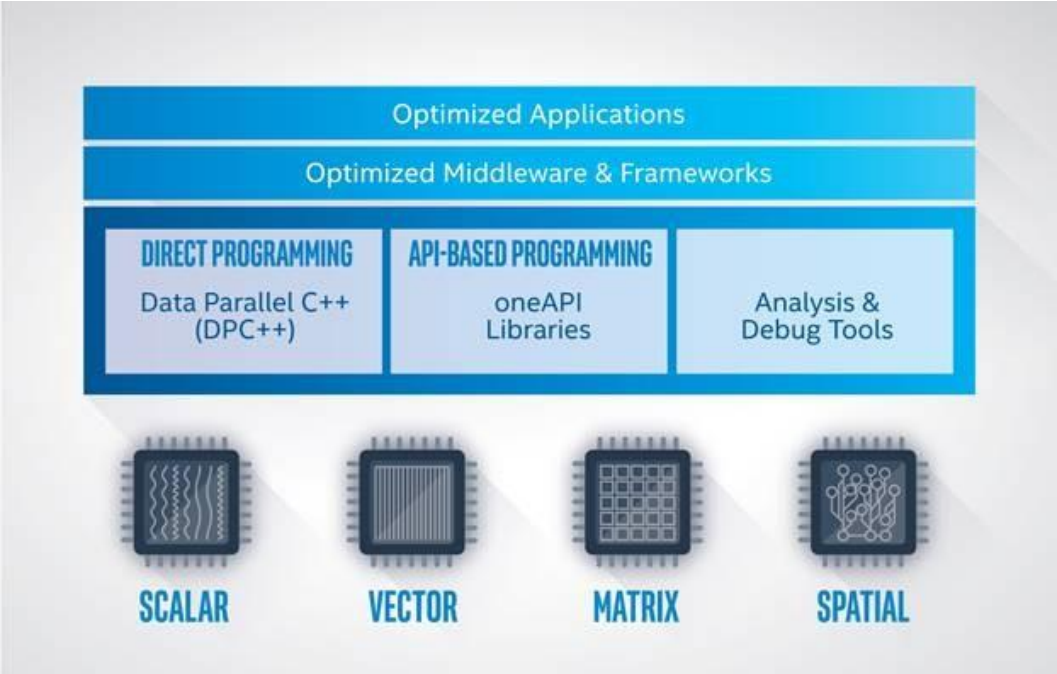
注: すべてのプログラムにおいて、oneAPI が提供する単一プログラミング・モデルの恩恵を得られるわけではありません。プログラムに適しているかどうかは、設計、実装、およびプログラムで使用する oneAPI プログラム・モデルを理解する必要があります。

---

oneapi.com(英語)で oneAPI イニシアチブとプログラミング・モデルの詳細をご覧ください。このサイトでは、oneAPI 仕様、SYCL* 言語ガイドと API リファレンス、その他のリソースが提供されています。

1.1 インテル® oneAPI プログラミングの概要

oneAPI プログラミング・モデルは、複数のワークロード・ドメインにまたがる広範囲のパフォーマンス・ライブラリーを含む、ハードウェア・ターゲット全体で利用できる開発者向けツールの包括的かつ統合された資源を提供します。ライブラリーには、ターゲット・アーキテクチャーごとにカスタマイズされた関数が含まれているため、同じ関数呼び出しを使用して、サポートされるすべてのアーキテクチャーで最適なパフォーマンスを実現できます。



上の図に示すように、oneAPI プログラミング・モデルを利用するアプリケーションは、CPU から FPGA まで複数のターゲット・ハードウェア・プラットフォームで実行できます。インテルは、一連のツールキットの一部としてインテル® oneAPI 製品を提供しています。インテル® oneAPI ベース・ツールキットと、インテル® oneAPI HPC ツールキット、およびその他のツールキットは、特定の開発者のワークロード要件を満たす補完的なツールを備えています。例えば、インテル® oneAPI ベース・ツールキットには、インテル® oneAPI DPC++/C++ コンパイラー、インテル® DPC++ 互換性ツール、ライブラリー、および解析ツールが含まれます。

- 既存の CUDA* コードを DPC++ に移行しようとする開発者は、**インテル® DPC++ 互換性ツール**を使用して既存のプロジェクトを DPC++ を使用した SYCL* に移行できます。
- **インテル® oneAPI DPC++/C++ コンパイラー**は、アクセラレーターをターゲットとするコードのダイレクト・プログラミングをサポートします。ダイレクト・プログラミングは、ユーザーコードで使用されるアルゴリズムで API が利用できない場合にパフォーマンスを向上するコーディング手法です。CPU と GPU ターゲット向けにはオンラインとオフラインコンパイルがサポートされ、FPGA ターゲット向けにはオフラインコンパイルのみがサポートされます。

- API ベースのプログラミングは、最適化済みのライブラリー・セットを介してサポートされます。インテル® oneAPI 製品で提供されるライブラリー関数は、サポートされるターゲット・アーキテクチャー向けに事前チューニングされているため、開発者の介在は必要ありません。例えば、**インテル® oneAPI マス・カーネル・ライブラリー(インテル® oneMKL)** の BLAS ルーチンは、CPU ターゲットだけでなく GPU ターゲット向けにも最適化されています。
- また、コンパイルされた DPC++ アプリケーションは、**インテル® VTune™ プロファイラー**や**インテル® Advisor** などのツールを使用して、パフォーマンス、安定性、エネルギー効率の目標を達成するため、解析およびデバッグできます。

インテル® oneAPI ベース・ツールキットは、インテル® デベロッパー・ゾーン(英語)から無料でダウンロードできます。

インテル® Parallel Studio XE やインテル® System Studio を利用しているユーザーは、インテル® oneAPI HPC ツールキット(英語)やインテル® oneAPI IoT ツールキット(英語)に興味を持つかもしれません。

1.2 インテル® oneAPI ツールキットの配布について

インテル® oneAPI ツールキットは、複数の配布経路から入手できます。

- 製品のローカル・インストール:インテル® デベロッパー・ゾーン(英語)からインテル® oneAPI ツールキットをインストールします。特定のインストールに関する情報は、「インストール・ガイド」(英語)を参照してください。
- コンテナーまたはリポジトリーからインストール:サポートされるコンテナーまたはリポジトリーからインテル® oneAPI ツールキットをインストールします。それぞれの手順は、「インストール・ガイド」(英語)に示されています。
- 事前インストールされたインテル® DevCloud:無料の開発サンドボックスを使用して、最新のインテル® ハードウェアにアクセスし、インテル® oneAPI ツールを選択します。インテル® DevCloud の詳細(英語)を確認し、無料アクセスに登録してください。

1.3 関連ドキュメント

次のドキュメントは、これから oneAPI プロジェクトを導入する開発者向けの入門資料として役立ちます。

- 選択した oneAPI ツールキットの導入ガイド:
  - インテル® oneAPI ベース・ツールキット導入ガイド(Linux* | Windows* | MacOS*(英語))
  - インテル® oneAPI HPC ツールキット導入ガイド(Linux* | Windows* | MacOS*(英語))
  - インテル® oneAPI IoT ツールキット導入ガイド(Linux* | Windows*)(英語)
- oneAPI ツールキットのリリースノート:
  - インテル® oneAPI ベース・ツールキット(英語)
  - インテル® oneAPI HPC ツールキット(英語)
  - インテル® oneAPI IoT ツールキット(英語)
- 言語リファレンス:
  - SYCL* 言語ガイドと API リファレンス(英語)
  - SYCL* 仕様 PDF(バージョン 1.2.1)(英語)
  - SYCL* 仕様 PDF(バージョン 2020)(日本語)
  - データ並列 C++ 書籍:Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL(英語)
  - LLVM/OpenMP* 関連のドキュメント(英語)
  - OpenMP* 仕様(英語)

oneAPI プログラミング・モデル 2

ヘテロジニアス・コンピューティングでは、ホスト・プロセッサーはアクセラレーター・デバイスの利点を活用して、コードをより効率良く実行します。

oneAPI プログラミング・モデルは、データ並列 C++(DPC++)および OpenMP*(Fortran、C、C++)の 2 つのヘテロジニアス・コンピューティング方式をサポートします。

SYCL* は、最新の C++ と Khronos Group の SYCL* 言語に基づいたオープンソース言語であり、インテル社によりいくつかの拡張機能が追加されています。インテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキットに含まれています。

OpenMP* は 20 年以上に渡り標準化されてきたプログラミング言語であり、インテルは OpenMP* 標準のバージョン 5.0 を実装しています。OpenMP* のオフロード機能をサポートするインテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキット、インテル® HPC ツールキット、そしてインテル® oneAPI IoT ツールキットに含まれます。OpenMP* オフロードをサポートするインテル® Fortran コンパイラー・クラシックとインテル® Fortran コンパイラーは、インテル® oneAPI HPC ツールキットで提供されます。

注:OpenMP* では FPGA デバイスをサポートしていません。

次のセクションでは、それぞれの言語について簡単に説明し詳細情報の参照先を示します。

2.1 SYCL* を使用したデータ並列 C++

SYCL* を使用したデータ並列 C++ は、データ並列プログラミングの生産性向けに設計された高レベル言語です。**SYCL*** の主な目的の 1 つは、ベースとなる OpenCL* C コードよりも高レベルのプログラミング言語を提供することです。OpenCL* プログラムに精通している開発者は、OpenCL* コードとの類似点と相違点に気付くでしょう。

2.1.1 簡単なサンプルコード

SYCL* の導入を示す最良の方法は、簡単なサンプルを使用することでしょう。SYCL* は最新の C++ をベースとしているため、この例ではラムダ式や一様初期化など近年 C++ に追加されたいくつかの機能を使用しています。開発者がこれらの機能に精通していなくても、それらの意味と機能はサンプルのコンテ

キストから明らかになります。SYCL* によるプログラミングの経験を積んでいくと、これらの新しい C++ 機能は自然に受け入れられるでしょう。

次のサンプルコードは、a[0] = 0、a[1] = 1、... のように配列の各要素をそのインデックス値に設定します。

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3
4  constexpr int num=16;
5  using namespace sycl;
6
7  int main() {
8   auto r = range{num};
9   buffer<int> a{r};
10  queue{}.submit([&](handler& h) {
11      accessor out{a, h};
12      h.parallel_for(r, [=](item<1> idx) {
13      out[idx] = idx;
14  });
15  });
17
18  host_accessor result{a};
19  for (int i=0; i<num; ++i)
20      std::cout << result[i] << "\n";
21 }
```

最初に気付くことは、ソースファイルが 1 つしかないことです。つまり、ホストコードとオフロードされるアクセラレーター・コードの両方がこの単一のソースファイルから生成されます。次に注目すべき点は、構文が標準の C++ であるということです。並列処理を表現する新しいキーワードやプラグマは使用されていません。代わりに、並列処理は C++ クラスを介して表現されています。例えば、8 行目にある buffer クラスはデバイスにオフロードされるデータを表し、10 行目の queue クラスはホストからアクセラレーターへの接続を表します。

ロジックは次のように動作します。8 行目と 9 行目で、初期値を持たない 16 個の int 要素の buffer を作成します。この buffer は配列のように作用します。11 行目でアクセラレーター・デバイスに接続するキュー(queue)を作成します。この簡単な例では、暗黙的に SYCL* がデフォルトのアクセラレーター・デバイスを選択しますが、アプリケーションによっては、システムのトポロジーを調査して特定のアクセラレーションを選択することもできます。キューが作成されると、この例では submit() メンバー関数を呼び出して、アクセラレーターにワークを送信します。この submit() 関数の引数はラムダ関数であり、ホスト上ですぐに実行されます。ラムダ関数は次の 2 つのことを行います。1 つは、12 行目でアクセサーを作成します。アクセサーはバッファーの要素を書き込むことができます。次に、13 行目で parallel_for() 関数を呼び出してコードをアクセラレーターで実行します。

parallel_for() の呼び出しには 2 つの引数があります。1 つはラムダ関数であり、もう 1 つはバッファー内の要素数を示す範囲オブジェクト "r" です。SYCL* は、ラムダ関数が範囲内のインデックスごとに一度(バッファー要素ごとに 1 回)、アクセラレーターで呼び出されるように調整します。ラムダは、12 行目で作成された out アクセサーを使用してバッファー要素に値を割り当てるだけです。この簡単な例

では、ラムダ呼び出し間に依存関係がないため、SYCL* はアクセラレーターで最も効率良い方法で自由に並行して実行できます。

`parallel_for()` を呼び出した後、ホストのコードはアクセラレーターの完了を待たずに処理を続行します。ホストが次に行うことは、18 行目でバッファーの要素を読み取る `host_accessor` を作成することです。SYCL* は、このバッファーがアクセラレーターによって書き込まれたことを認識するため、`host_accessor` コンストラクター(18 行目)は、`parallel_for()` によって送信されたワークが完了するまでブロックされます。アクセラレーターのワークが完了すると、ホストコードは 18 行目以降を続行し、`out` アクセサーを使用してバッファーから値を読み取ります。

2.1.2 その他のリソース

この SYCL* の概要は、完全なチュートリアルを目指すものではなく、言語機能の一部分を紹介するだけです。ローカルメモリー、バリア、SIMD など一般にアクセラレーター・ハードウェアで使用するため学ぶべきことはほかにもたくさんあります。一度に複数のアクセラレーター・デバイスにワークを送信する機能もあり、1 つのアプリケーションが複数のデバイスで同時にワークを並行して実行することもできます。

以下のリソースは、DPC++ を使用して SYCL* を学習して習得するのに役立ちます。

- 「インテルのサンプルを使用した SYCL* の調査」(英語)では、GitHub* から入手できるサンプル・アプリケーションの紹介とリンクを示しています。
- 「DPC++ 基礎サンプルコードのウォークスルー」(英語)は、最初の一歩である「HelloWorld」アプリケーションに相当する DPC++ ベクトル加算のサンプルコードを詳しく見ていきます。
- oneapi.com(英語)サイトでは、クラスとそれらのインターフェイスの説明が記載された『言語ガイドと API リファレンス』(英語)が公開されています。また、DPC++ と SYCL* の 4 つのプログラミング・モデル(プラットフォーム、実行モデル、メモリーモデル、およびカーネル・プログラミング・モデル)を詳しく説明しています。
- 「SYCL* エッセンシャル・トレーニング・ソース」(英語)は、インテル® DevCloud で Jupyter* Notebook を使用するガイド付きの学習コースです。
- 『データ並列 C++:C++ と SYCL* を使用したヘテロジニアス・システムのプログラミング向けに DPC++ を習得』(英語)は、DPC++ に関連するプログラミングの概念と言語の詳細を紹介する書籍です。

2.2 C/C++ または Fortran と OpenMP* オフロード・プログラミング・モデル

インテル® oneAPI DPC++/C++ コンパイラーおよびベータ版インテル® Fortran コンパイラーを使用すると、OpenMP* ディレクティブを使用してワークをインテルのアクセラレーター・デバイスにオフロードし、アプリケーションのパフォーマンスを向上できます。

このセクションでは、OpenMP* ディレクティブを使用して計算をアクセラレーター・デバイスにオフロードする方法について説明します。OpenMP* ディレクティブに慣れていない開発者は、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドとリファレンス』や『oneAPI 対応インテル® Fortran コンパイラー・デベロッパー・ガイドとリファレンス』(英語)の OpenMP* サポートのセクションで基本的な使い方をご覧いただけます。

注: OpenMP* では FPGA デバイスをサポートしていません。

2.2.1 基本的な OpenMP* target 構造

OpenMP* target 構造は、ホストからターゲットデバイスへ制御を移行するために使用されます。変数はホストとターゲットデバイスでマッピングされます。ホストスレッドは、オフロードされた計算が完了するまで待機します。ほかの OpenMP* タスクは、ホストで非同期に実行できます。それには、`nowait` 節を使用して、スレッドがターゲット領域の完了を待機しないようにします。

C/C++

次の C++ のコードスニペットは、SAXPY 計算をアクセラレーターにオフロードします。

```
#pragma omp target map(tofrom:fa), map(to:fb,a)
#pragma omp parallel for firstprivate(a)
for(k=0; k<FLOPS_ARRAY_SIZE; k++)
        fa[k] = a * fa[k] + fb[k]
```

配列 `fa` は、計算の入力と出力の両方で使用されるため、アクセラレーターの to と from にマップされます。配列 `fb` と変数 `a` は計算の入力であり変更されることがないため、その出力をコピーする必要はありません。変数 `FLOPS_ARRAY_SIZE` はアクセラレーターに暗黙にマップされます。ループ・インデックス k は、OpenMP* に従って暗黙的にプライベートです。

Fortran

次の Fortran のコードのコードスニペットは、行列乗算をアクセラレーターにオフロードします。

```
!$omp target map(to: a, b ) map(tofrom: c )
!$omp parallel do private(j,i,k)
      do j=1,n
         do i=1,n
            do k=1,n
               c(i,j) = c(i,j) + a(i,k) * b(k,j)
            enddo
         enddo
      enddo
!$omp end parallel do
!$omp end target
```

配列 a と b はアクセラレーターの入力にマップされ、配列 c はアクセラレーターの入力と出力にマップされます。変数 n はアクセラレーターに暗黙にマップされます。ループ·インデックスは OpenMP* の仕様に従って自動的に private となるため、private 節はオプションです。

2.2.2 Map 変数

ホストとアクセラレーター間のデータ共有を最適化するため、target data ディレクティブは変数をアクセラレーターにマップし、変数はその領域の範囲内でターゲットのデータ領域に維持されます。この機能は、複数のターゲット領域にまたがって変数をマップするのに役立ちます。

C/C++

```
#pragma omp target data [節 [[,] 節],…]
 構造化ブロック
```

Fortran

```
!$omp target data [節 [[,] 節],…]
構造化ブロック
!$omp end target data
```

節の使用例

節には次の 1 つ以上を指定できます。詳細は、TARGET DATA(英語)を参照してください。

- DEVICE (整数式)
- IF ([TARGET DATA:] スカラー論理式)
- MAP ([[マップタイプ修飾子 [,]] マップタイプ:] リスト)

注: マップタイプには以下を複数指定できます。

  - alloc
  - to
  - from
  - tofrom
  - delete
  - release
- SUBDEVICE ([整数定数,] 整数式[: 整数式[: 整数式]])
- USE_DEVICE_ADDR (リスト) // ifx でのみ利用可能
- USE_DEVICE_PTR (ポインターリスト)

```
DEVICE (整数式)
IF ([TARGET DATA:] スカラー論理式)
MAP ([[マップタイプ修飾子 [,]] マップタイプ: alloc | to | from | tofrom | delete |
release] リスト)
SUBDEVICE ([整数定数,] 整数式[ : 整数式[ : 整数式]])
USE_DEVICE_ADDR (リスト) // ifx でのみ利用可能
USE_DEVICE_PTR (ポインターリスト)
```

target update ディレクティブを使用して、ホストの変数をデバイスの対応する変数と同期することができます。

2.2.3 OMP TARGET を使用するコンパイル

次のコマンドは、OpenMP* target を使用するアプリケーションをコンパイルする例を示します。

C/C++

- Linux*

```
icx -fiopenmp -fopenmp-targets=spir64 code.c
```

- Windows*

```
icx /Qiopenmp /Qopenmp-targets=spir64 code.c
```

Fortran

- Linux*

```
ifx -fiopenmp -fopenmp-targets=spir64 code.f90
```

- Windows*

```
ifx /Qiopenmp /Qopenmp-targets=spir64 code.f90
```

2.2.4 OpenMP* オフロードの追加のリソース

- インテルは、OpenMP* ディレクティブを使用してアクセラレーターをターゲットとする以下の具体的なサンプルを、https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming (英語)で提供しています。
  - 行列乗算(英語)は、2 つの大きな行列を乗算して結果を検証する簡単なプログラムです。このプログラムは、SYCL* または OpenMP* の 2 つの方法で実装されます。
  - ISO3DFD(英語)サンプルは、等方性媒質における 3 次元有限差分波伝搬を参照しています。このサンプルは、3D 等方性媒質を伝搬する波形をシミュレートする 3 次元ステンシルであり、複雑なアプリケーションで OMP アクセラレーター・デバイスをターゲットとして高いパフォーマンスを実現する一般的な課題といくつかの手法を示しています。
  - `openmp_reduction`(英語)は円周率を求める簡単なプログラムです。このプログラムは、インテル® アーキテクチャー・ベースの CPU およびアクセラレーター向けの C++ および OpenMP* により実装されています。
- OpenMP* オフロード機能の導入(英語)では、サポートされるオプションやサンプルコードなど、OpenMP* オフロードでインテル® コンパイラーを使用する方法を説明します。
- LLVM/OpenMP* ランタイム(英語)は、利用可能な各種タイプのランタイムについて説明しており、OpenMP* オフロードをデバッグする際に役立ちます。
- openmp.org では豊富なサンプルコードが提供されています : https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf(英語)。例題ドキュメントの第 4 章では、アクセラレーターと target 構造に焦点を当てています。
- 『Using OpenMP – the Next Step』は、OpenMP* の優れた参考書籍です。第 6 章では、ヘテロジニアス・システムにおける OpenMP* のサポートについて説明しています。この書籍の追加情報については、https://www.openmp.org/tech/using-openmp-next-step(英語)をご覧ください。

2.3 デバイスの選択

デバイス(CPU、GPU または FPGA など)へのコードのオフロードは、DPC++ アプリケーションと OpenMP* アプリケーションの両方で利用できます。

2.3.1 ホストコードでの DPC++ デバイスの選択

ホストコードは明示的にデバイスタイプを選択できます。デバイスを選択するには、キューを選択して次のいずれかのデバイスを初期化します。

- `default_selector`
- `cpu_selector`
- `gpu_selector`
- `accelerator_selector`

`default_selector` が使用されると、カーネルは利用可能な計算デバイス(すべて、または `SYCL_DEVICE_FILTER` 環境変数の値に基づくサブセット)から選択するヒューリスティックに基づいて実行されます。

特定のデバイスタイプ(`cpu_selector` や `gpu_selector`)を使用する場合、指定されたデバイスタイプがプラットフォームで利用可能であるか、`SYCL_DEVICE_FILTER` で指定されるフィルターに含まれていなければなりません。指定したデバイスが利用できない場合、ランタイムシステムはデバイスが利用できないことを示す例外をスローします。このエラーは、事前コンパイル(AOT)したバイナリーが、指定するデバイスタイプを含まないプラットフォームで実行される場合にスローされることがあります。

注: DPC++ アプリケーションは、サポートされる任意のターゲット・ハードウェアで実行できますが、特定のターゲット・ハードウェアで最高のパフォーマンスを引き出すにはチューニングが必要です。例えば、CPU 向けにチューニングされたコードは、変更なしでは GPU アクセラレーターでは高速に実行できない可能性があります。

`SYCL_DEVICE_FILTER` は、DPC++ ランタイムで使用されるランタイム、計算デバイスタイプ、および計算デバイス ID を、利用可能なすべての組み合わせのサブセットに制限することができる複雑な環境変数です。計算デバイス ID は、SYCL* API、`clinfo` または `sycl-ls`(0 から始まる番号)によって返される ID に対応し、その ID を持つデバイスが特定のタイプであるか、特定のランタイムをサポートするかは関係ありません。プログラムによる特定のセレクター(`gpu_selector` など)を使用してフィルターで除外されたデバイスを要求すると、例外がスローされます。使い方と設定可能な値の例については、GitHub* の環境変数の説明をご覧ください。

https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md(英語)

`sycl-ls` ツールを使用して、システムで利用可能なデバイスを確認できます。SYCL* や DPC++ プログラムを実行する前に、このツールでデバイスを確認することを推奨します。`sycl-ls` は、`SYCL_DEVICE_FILTER` に設定されている文字列を各デバイスのプリフィクスとして出力します。`sycl-ls` の出力形式は、[`SYCL_DEVICE_FILTER`] プラットフォーム名、デバイス名、デバイスのバージョン [ドライバーのバージョン] です。次の例で各行の先頭の角かっこ ([ ]) で囲まれた文字列は、プログラムが実行される特定のデバイスを指定する `SYCL_DEVICE_FILTER` 文字列です。

```
$ sycl-ls

[opencl:acc:0] Intel(R) FPGA Emulation Platform for OpenCL(TM), Intel(R)
FPGA Emulation Device 1.2 [2021.12.9.0.24_005321]
[opencl:gpu:1] Intel(R) OpenCL HD Graphics, Intel(R) UHD Graphics 630
[0x3e92] 3.0 [21.37.20939]
[opencl:cpu:2] Intel(R) OpenCL, Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
3.0 [2021.12.9.0.24_005321]
[level_zero:gpu:0] Intel(R) Level-Zero, Intel(R) UHD Graphics 630
[0x3e92] 1.1 [1.2.20939]
[host:host:0] SYCL host platform, SYCL host device 1.2 [1.2]
```

デバイス選択に関する詳しい情報は、『DPC++ 言語ガイドと API リファレンス』(英語)で入手できます。

2.3.2 ホストコードでの OpenMP* デバイスの確認と選択

OpenMP* では、開発者がデバイス上でコードを実行できるか確認および設定する API が用意されています。ホストコードはデバイス番号を明示的に選択および設定できます。開発者は、オフロード領域ごとに **device** 句を使用して、オフロード領域を実行するターゲットデバイスを指定できます。

- `int omp_get_num_procs (void)` API は、デバイスで使用可能なプロセッサー数を返します。
- `void omp_set_default_device(int device_num)` API は、デフォルトのターゲットデバイスを設定します。
- `int omp_get_default_device(void)` API は、デフォルトのターゲットデバイスを返します。
- `int omp_get_num_devices(void)` API は、コードまたはデータをオフロードできるホスト以外のデバイスの数を返します。
- `int omp_get_device_num(void)` API は、呼び出したスレッドが実行されているデバイスのデバイス番号を返します。
- `int omp_is_initial_device(int device_num)` API は、現在のタスクがホストデバイスで実行されている場合は *true* を返し、それ以外は *false* を返します。
- `int omp_get_initial_device(void)` API は、ホストデバイスを表すデバイス番号を返します。

開発者は、環境変数 `LIBOMPTARGET_DEVICETYPE = [ CPU | GPU ]` で実行するデバイスタイプを選択できます。CPU や GPU のように特定のデバイスが指定される場合、そのデバイスがプラットフォームで利用可能であることが求められます。指定するデバイスが利用できない場合、ランタイムシステムは環境変数 `OMP_TARGET_OFFLOAD` に従って動作します。`OMP_TARGET_OFFLOAD=mandatory` の場合、要求されたデバイスが利用できないという意味のメッセージを出力します。それ以外の場合はベースデバイス(通常は `CPU`)でフォールバック実行されます。デバイスの選択に関する追加機能は、OpenMP* 5.1 仕様で確認できます。

環境変数に関する詳細は、以下の GitHub* ページから入手できます。
https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md（英語）

関連情報

FPGA デバイスセレクター

oneAPI 開発環境の設定

3

インテル® oneAPI ツールは、このドキュメントの最初にある「インテル® oneAPI ツールキットの配布について」で説明しているように、いくつかの形式で利用できます。『インテル® oneAPI インストール・ガイド』(英語)の指示に従って、ツールを入手してインストールします。

3.1 インストール・ディレクトリー

Windows* システムでは、インテル® oneAPI 開発ツールは `C:\Program Files (x86)\Intel\oneAPI\` ディレクトリーにインストールされます(デフォルト)。

Linux* と macOS* システムでは、インテル® oneAPI 開発ツールは `/opt/intel/oneapi/` ディレクトリーにインストールされます(デフォルト)。

デフォルトのインストール先はインストール中に変更できます。

oneAPI インストール・ディレクトリー内には、開発システムにインストールされているコンパイラー、ライブラリー、解析ツール、およびそのほかのツールを含むフォルダーが含まれます。正確なファイルは、インストールされるツールキットとインストール中に選択されるオプションによって異なります。oneAPI インストール・ディレクトリー内のほとんどのフォルダーは、コンポーネント名に直結する分かりやすい名前が付いています。例えば、`mkl` フォルダーにはインテル® oneAPI マス・カーネル・ライブラリー(インテル® oneMKL)が含まれ、`ipp` フォルダーにはインテル® インテグレーテッド・パフォーマンス・プリミティブ(インテル® IPP)ライブラリーが含まれます。

3.2 環境変数

インテル® oneAPI ツールキットの一部のツールは、次の環境変数に影響されます。

- コンパイルとリンク処理の制御(PATH、CPATH、INCLUDE など)
- デバッガー、解析ツール、およびローカルヘルプの場所(PATH、MANPATH など)
- ツール固有のパラメーターと動的(共有)リンク・ライブラリーの特定(LD_LIBRARY_PATH、CONDA_* など)

3.3 setvars と vars ファイル

インストールされるすべてのインテル® oneAPI ツールキットには、親スクリプト "setvars" と、ツール固有のスクリプト "vars" が含まれます(setvars.sh と vars.sh は Linux* と macOS*、setvars.bat と vars.bat は Windows*)。これらのスクリプトが実行(または source)されると、各インテル® oneAPI 開発ツールに必要なローカル環境変数が設定されます。

次のセクションでは、インテル® oneAPI の setvars と vars スクリプトを使用して、インテル® oneAPI 開発環境を初期化する方法を詳しく説明します。

- Windows* で setvars スクリプトを使用
- Linux* または macOS* で setvars スクリプトを使用

3.4 モジュールファイル(Linux* のみ)

環境モジュール(英語)を利用するユーザーは、インテル® oneAPI ツールキットのインストール・パッケージに含まれるモジュールファイルを使用して、開発環境を初期化することがあります。インテル® oneAPI モジュール・ファイル・スクリプトは Linux* 環境でのみサポートされており、setvars と vars スクリプトの代わりに使用することができます。モジュールファイルと setvars 環境スクリプトを混在して使用しないでください。

インテル® oneAPI モジュールファイルを使用してインテル® oneAPI 開発環境を初期化する方法の詳細については、「Linux* 環境でモジュールファイルを使用」をご覧ください。

3.4.1 Windows* で setvars スクリプトを使用

ほとんどのインテル® oneAPI コンポーネントのフォルダーには、それぞれのコンポーネントに必要な環境変数を設定する vars.bat スクリプトが含まれています。例えば、デフォルトのインストールでは、Windows* のインテル® インテグレーテッド・パフォーマンス・プリミティブ(インテル® IPP)の vars スクリプトは C:\Program Files (x86)\Intel\oneAPI\ipp\latest\env\vars.bat に配置されます。このパスは、環境変数設定スクリプトを含むすべてのインテル® oneAPI コンポーネントで共有されます。

これらの各コンポーネント向けの vars スクリプトは、直接またはまとめて呼び出すことができます。まとめて呼び出すには oneAPI インストール・ディレクトリーにある setvars.bat スクリプトを使用します。例えば、Windows* マシンのデフォルトのインストールでは、C:\Program Files (x86)\Intel\oneAPI\setvars.bat にあります。

引数なしで setvars.bat スクリプトを実行すると、システムにインストールされているすべての <コンポーネント>\latest\env\vars.bat スクリプトが実行されます。これらの環境変数設定スクリプトを実行した後、Windows* の set コマンドを使用して環境変数を確認できます。

Visual Studio Code* 開発者は、oneAPI 環境拡張機能をインストールして、Visual Studio Code* で setvars.bat を実行できます。詳細については、「Visual Studio Code* でインテル® oneAPI ツールキットを使用する」をご覧ください。

注: setvars.bat スクリプト(または個別の vars.bat スクリプト)により変更された環境は永続的ではありません。これらの変更は、setvars.bat スクリプトが実行された cmd.exe セッションでのみ有効です。

コマンドライン引数

setvars.bat スクリプトはいくつかのコマンドライン引数をサポートしており、--help オプションで引数の一覧を表示できます。

次に例を示します。

```
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" --help
```

--config=file 引数と setvars.bat スクリプトから呼び出される vars.bat スクリプトへの追加引数をインクルードする機能を使用して、環境設定をカスタマイズできます。

--config=file 引数は、特定のインテル® oneAPI コンポーネントの環境の初期化機能を提供するとともに、特定のバージョンの環境を初期化することもできます。例えば、インテル® インテグレーテッド・パフォーマンス・プリミティブ(インテル® IPP)ライブラリーとインテル® oneAPI マス・カーネル・ライブラリー(インテル® oneMKL)の環境のみを設定するには、setvars.bat スクリプトにこれら 2 つのインテル® oneAPI コンポーネントの環境変数のみを設定することを指示する設定ファイルを渡します。詳細と利用例については、「Windows* で setvars.bat の設定ファイルを使用」をご覧ください。

setvars.bat のヘルプメッセージに記載されていないコマンドライン引数は、そのまま vars.bat スクリプトに渡されます。つまり、setvars.bat スクリプトが認識できない引数は、コンポーネントの vars.bat スクリプトで使用されるものとみなし、それらの引数をすべてのコンポーネントの vars.bat スクリプトに渡します。最もよく使用される追加の引数は、ia32 と intel64 です。これらは、インテル® コンパイラー、インテル® IPP、インテル® MKL およびインテル® TBB ライブラリーでアプリケーションのターゲット・アーキテクチャーを指示するために使用されます。

システムに複数バージョンの Microsoft* Visual Studio* がインストールされている場合、vs2017、vs2019 または vs2022 引数を setvars.bat コマンドラインに追加することで、Visual Studio* 環境

のいずれかをインテル® oneAPI 環境の初期化に使用するかを指定できます。デフォルトでは、Visual Studio* の最新バージョンが使用されます。

注: Microsoft Visual Studio* 2017 のサポートはインテル® oneAPI 2022.1 では非推奨となり、将来のリリースで削除される予定です。

個々の vars.bat スクリプトを調べて、受け入れるコマンドライン引数があればそれを決定します。

実行方法

```
<install-dir>\setvars.bat
```

PowerShell ウィンドウで setvars.bat または vars.bat スクリプトを実行するには、以下を使用します。

```
cmd.exe "/K" '"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" && powershell'
```

確認方法

setvars.bat を実行した後、SETVARS_COMPLETED 環境変数を検索して設定の成功を確認できます。setvars.bat が成功すると、SETVARS_COMPLETED には 1 が設定されます。

```
set | find "SETVARS_COMPLETED"
```

戻り値

```
SETVARS_COMPLETED=1
```

SETVARS_COMPLETED=1 以外の場合、setvars.bat は設定に失敗したことを意味します。

複数の実行

各コンポーネントの env\vars.bat スクリプトの多くは、PATH、CPATH、およびそのほかの環境変数に変更を加えるため、最上位の setvars.bat スクリプトは同じセッションで同じ vars.bat を複数回呼び出すことはできません。これは、特に %PATH% 環境変数が原因で環境変数の文字数が長くなりすぎないようにします。設定可能な文字数を超えると、ターミナルセッションで予期しない動作を招くことがあるため回避する必要があります。

これを強制するには、setvars.bat に --force オプションを指定します。この例では、ユーザーが setvars.bat を 2 度実行しています。setvars.bat がすでに実行されているため、2 回目の実行は停止します。

```
> <install-dir>\setvars.bat
:: initializing environment ...
(SNIP: lot of output)
:: oneAPI environment initialized ::
```

```
> <install-dir>\setvars.bat
 :: WARNING: setvars.bat has already been run. Skipping re-execution.
    To force a re-execution of setvars.bat, use the '--force' option.
    Using '--force' can result in excessive use of your environment variables.
```

次は、ユーザーが `<install-dir>\setvars.bat --force` を実行し、初期化が成功した例です。

```
> <install-dir>\setvars.bat --force
 :: initializing environment ...
(SNIP: lot of output)
 :: oneAPI environment initialized ::
```

ONEAPI_ROOT 環境変数

`ONEAPI_ROOT` 環境変数は、スクリプトが実行されるときに最上位の `setvars.bat` によって設定されます。`ONEAPI_ROOT` 環境変数がすでに設定されている場合、`setvars.bat` はスクリプトを実行した `cmd.exe` セッションを一時的に上書きします。この変数は、`oneapi-cli` サンプルブラウザーと Microsoft Visual Studio* および Visual Studio* Code サンプルブラウザーによって使用され、インテル® oneAPI ツールとコンポーネントの検出、および `SETVARS_CONFIG` 機能が有効である場合に `setvars.bat` スクリプトを検出するのに役立ちます。`SETVARS_CONFIG` 機能の詳細については、「Microsoft Visual Studio* で setvars.bat スクリプトを自動化」をご覧ください。

Windows* システムでは、インストーラーは `ONEAPI_ROOT` 変数を環境に追加します。

Windows* で `setvars.bat` の設定ファイルを使用

`setvars.bat` スクリプトは、それぞれの oneAPI ディレクトリーにある `<install-dir>\latest\env\vars.bat` スクリプトを実行することで、oneAPI ツールキットの環境変数を設定します。`setvars.bat` スクリプトを自動実行しないように Windows* システムを設定しない限り、新しいターミナルウィンドウを開くか Visual Studio*、Sublime Text*、またはそのほかの C/C++ エディターを起動するたびに `setvars.bat` スクリプトが実行されます。詳細は、「システムの設定」(英語)を参照してください。

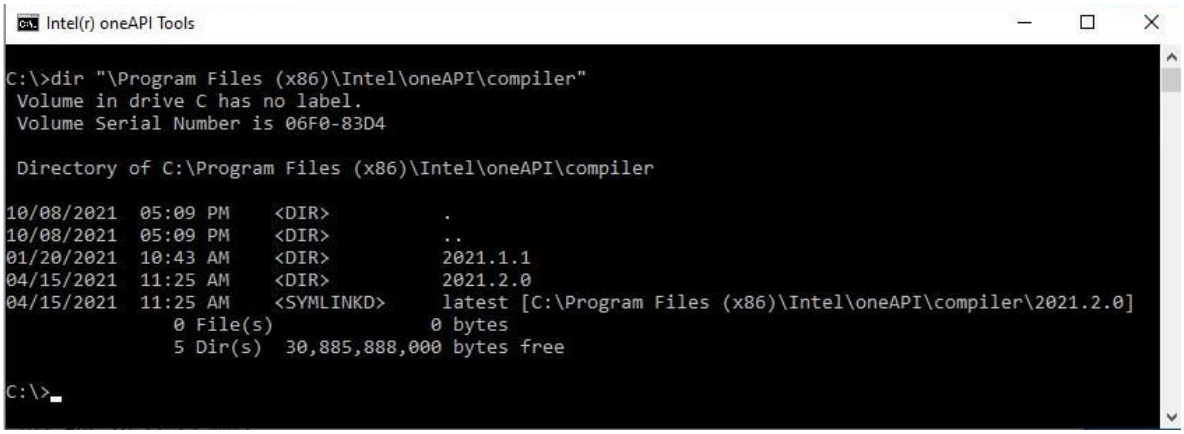
次に設定ファイルを使用して環境変数を管理する方法を説明します。

バージョンと構成

一部のインテル® oneAPI ツールは複数バージョンのインストールをサポートします。複数バージョンをサポートするツールのディレクトリー構造は次のようになります(デフォルトのインストールを想定し、例としてコンパイラーを使用します)。

```
\Program Files (x86)\Intel\oneAPI\compiler\
|--2021.1.1
|--2021.2.0
`--latest -> 2021.2.0
```

次に例を示します。



すべてのツールには、そのコンポーネントの最新バージョンのインストール先を示す `latest` という名前のショートカットがあります。`latest\env\` ディレクトリーにある `vars.bat` スクリプトは、`setvars.bat` によって実行されます(デフォルト)。

必要に応じて、設定ファイルを使用して特定のディレクトリーを示すよう `setvars.bat` をカスタマイズできます。

`--config` パラメーター

最上位の `setvars.bat` スクリプトは、カスタム **config.txt** ファイルを指定する `--config` パラメーターを受け入れます。

```
<install-dir>\setvars.bat --config="path\to\your\config.txt"
```

設定ファイルは任意の名前にすることができます。複数の設定ファイルを作成して、さまざまな開発環境やテスト環境を設定できます。例えば、最新バージョンのライブラリーを古いバージョンのコンパイラーでテストしたいこともあります。そのような場合に、`setvars` 設定ファイルを使用して環境を管理できます。

設定ファイルの例

以下に簡単な設定ファイルの例を示します。

最新のコンポーネントをすべてロード

```
mkl=1.1
dldt=exclude
```

ただし以下のコンポーネントは除外

```
default=exclude
mkl=1.0
ipp=latest
```

設定テキストファイルは次の要件に従う必要があります。

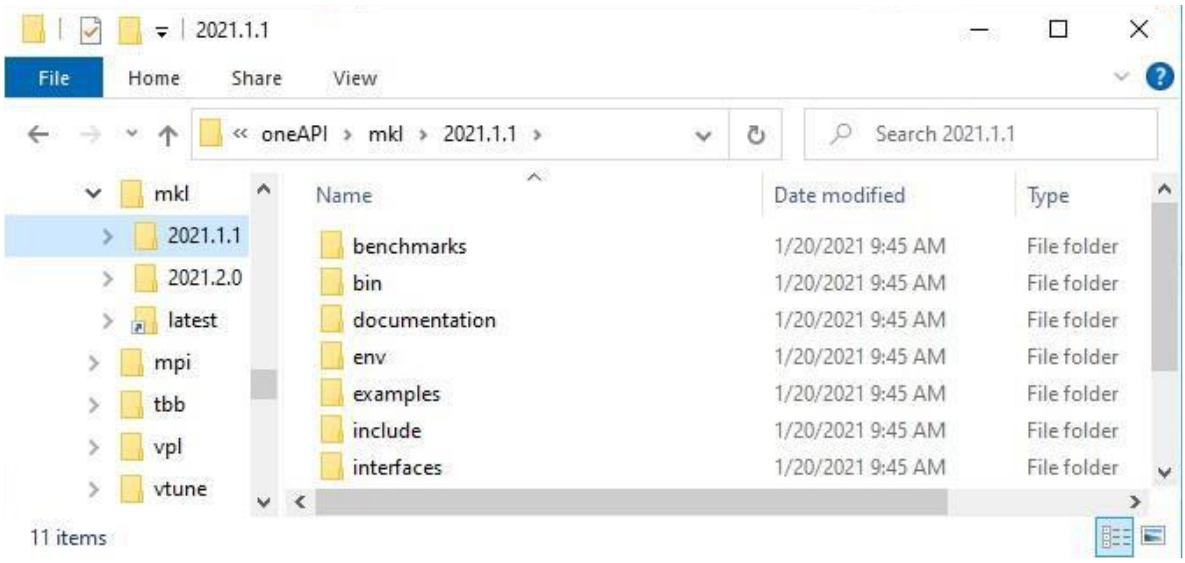
- 改行で区切られたテキストファイル
- 各行は、"key=value" のペアで構成されます。
- "key" には、oneAPI ディレクトリーの最上位(%ONEAPI_ROOT% ディレクトリーにあるフォルダー)のコンポーネント名を指定します。同じ "key" が設定ファイルに複数定義されると、最後の "key" が優先されそれ以外は無視されます。
- "value" には、コンポーネント・ディレクトリーの最上位にあるバージョン・ディレクトリー名を指定します。これには、コンポーネント・ディレクトリーのレベルに存在する可能性があるショートカット(latest など)が含まれます。
- また、"value" は "exclude" にすることもできます。これは、指定された key の vars.bat スクリプトを setvars.bat スクリプトで実行しないことを意味します。

"key=value" を "default=exclude" にすると特別な意味を持ちます。これは、設定ファイルに定義されているものを除き、それ以外のすべての env\vars.bat スクリプトの実行を除外します。以下に例を示します。

設定ファイルのカスタマイズ

設定ファイルを使用して、特定のコンポーネントを除外したり、特定のバージョンを含めたり、特定のコンポーネントのバージョンのみを含めることができます。これには、設定ファイルの "default=exclude" 行を変更します。

デフォルトでは、setvars.bat は最新(latest)のバージョンに対応する env\vars.bat スクリプトを処理します。例えば、2 つのバージョン(2021.1.1 と 2021.2.0)のインテル® oneMKL がインストールされていると仮定します。最新のバージョンを示すショートカットは 2021.2.0 であるため、デフォルトでは setvars.bat は mkl ディレクトリーの 2021.2.0 の vars.bat スクリプトを実行します。



特定のバージョンを指定

setvars.bat に `<install-dir>\mkl\2021.1.1\env\vars.bat` スクリプトを実行するように直接記述するには、設定ファイルに mkl=2021.1.1 を追加します。

これにより、`setvars.bat` は、`mkl` ディレクトリー内の `2021.1.1` フォルダーにある `env\vars.bat` スクリプトを実行するようになります。インストールされている `mkl` 以外のコンポーネントでは、`setvars.bat` は最新バージョンのフォルダーにある `env\vars.bat` スクリプトを実行します。

特定のコンポーネントを除外

コンポーネントを除外する構文は次のようになります。

```
<コンポーネント>=exclude
```

例えば、インテル® IPP を除外して、2021.1.1 のインテル® oneMKL を含めるには次のようにします。

```
mkl=2021.1.1
ipp=exclude
```

この例は次のように作用します。

- `setvars.bat` は、インテル® oneMKL 2021.1.1 の `env\vars.bat` スクリプトを実行します。
- `setvars.bat` は、インテル® IPP の `env\vars.bat` スクリプトを実行しません。
- `setvars.bat` は、そのほかのコンポーネントの最新バージョンの `env\vars.bat` スクリプトを実行します。

特定のコンポーネントを含める

特定のコンポーネントの `env\vars.bat` スクリプトを実行するには、最初にすべてのコンポーネントの `env\vars.bat` スクリプトを除外する必要があります。その後、`setvars.bat` で実行するコンポーネントを追加し直します。次の行を定義して、すべてのコンポーネントの `env\vars.bat` スクリプトを実行から除外します。

```
default=exclude
```

そして、`setvars.bat` にインテル® oneMKL とインテル® IPP の `env\vars.bat` のみを実行させるには、次の行を追加します。

```
default=exclude
mkl=2021.1.1
ipp=latest
```

この例は次のように作用します。

- `setvars.bat` は、インテル® oneMKL 2021.1.1 の `env\vars.bat` スクリプトを実行します。
- `setvars.bat` は、インテル® IPP の最新バージョンの `env\vars.bat` スクリプトを実行します。
- `setvars.bat` は、そのほかのコンポーネントの `env\vars.bat` スクリプトを実行しません。

Microsoft Visual Studio* で `setvars.bat` スクリプトを自動化

注: Microsoft Visual Studio* 2017 のサポートはインテル® oneAPI 2022.1 では非推奨となり、将来のリリースで削除される予定です。

`setvars.bat` スクリプトは、インテル® oneAPI ツールキットを使用するために必要な環境変数を設定します。このスクリプトは、コマンドライン開発向けに新しいターミナルウィンドウを開くたびに実行する必要があります。`setvars.bat` スクリプトはまた、Microsoft Visual Studio* の起動時に自動的に実行することもできます。`SETVARS_CONFIG` 環境変数を使用して `setvars.bat` スクリプトにインテル® oneAPI ツール固有の設定を行うように指示できます。

SETVARS_CONFIG 環境変数の状態

SETVARS_CONFIG 環境変数を使用して、Microsoft Visual Studio* のインスタンスを起動したときにインテル® oneAPI 開発環境を自動的に設定できます。環境変数には 3 つの条件と状態があります。

- 未定義(`SETVARS_CONFIG` 環境変数が存在しない)
- 定義されているが空(値を含まないか空白である)

- setvars.bat 設定ファイルを示すように定義します

SETVARS_CONFIG が定義されていないと、Visual Studio* 起動時に setvars.bat スクリプトは自動実行されません。SETVARS_CONFIG 環境変数は、インテル® oneAPI インストーラーによって定義されないため、これがデフォルト動作です。

SETVARS_CONFIG に値が設定されず空白のみが含まれる場合、Visual Studio* の起動時に setvars.bat スクリプトは自動的に実行されます。この場合、setvars.bat スクリプトはシステムにインストールされているすべての oneAPI ツールの環境を初期化します。setvars.bat スクリプト実行の詳細については、「Visual Studio* コマンドラインを使用したサンプル・プロジェクトのビルドと実行」(英語)を参照してください。

SETVARS_CONFIG に setvars 設定ファイルへの絶対パスが定義されている場合、Visual Studio* の起動時に setvars.bat スクリプトは自動的に実行されます。この場合、setvars.bat スクリプトは、setvars 設定ファイルで定義される oneAPI ツールのみの環境を初期化します。setvars 設定ファイルを作成する方法の詳細は、「setvars.bat で設定ファイルを使用」をご覧ください。

setvars 設定ファイルは任意のファイル名にでき、Visual Studio がその場所とファイルにアクセスして読み取り可能である限り、ハードディスク上の任意の場所に保存できます (Windows* システムに oneAPI ツールをインストールする際に Visual Studio に追加されるプラグインは、SETVARS_CONFIG のアクションを実行します。そのため、Visual Studio は setvars 設定ファイルの場所と内容にアクセスできる必要があります)。

setvars 設定ファイルを空のままにすると、setvars.bat スクリプトはシステムにインストールされているすべてのインテル® oneAPI ツールの環境を初期化します。これは、SETVARS_CONFIG 変数に空の文字列を定義するのと同じです。setvars 設定ファイルの定義の詳細については、「setvars.bat で設定ファイルを使用」を参照してください。

SETVARS_CONFIG 環境変数の定義

SETVARS_CONFIG 環境変数は、インストール中に自動的に定義されないため、Visual Studio* を起動する前に (上記の規則に従って) 手動で環境変数を定義する必要があります)。SETVARS_CONFIG 環境変数は、Windows* の SETX コマンド、または Windows* GUI ツールで「Win + R」ダイアログに「rundll32.exesysdm.cpl,EditEnvironmentVariables」と入力して定義できます (「Win + R」でダイアログを表示できます)。

3.4.2 Linux* または macOS* で setvars スクリプトを使用

ほとんどのインテル® oneAPI コンポーネントのフォルダーには、それぞれのコンポーネントに必要な環境変数を設定する vars.sh スクリプトが含まれています。例えば、デフォルトのインストールでは、Linux* や macOS* のインテル® インテグレーテッド・パフォーマンス・プリミティブ(インテル® IPP)の

vars スクリプトは /opt/intel/ipp/latest/env/vars.sh に配置されます。このパス階層は、環境変数 vars スクリプトを含むすべてのインテル® oneAPI コンポーネントで共有されます。

これらのコンポーネント向けの vars スクリプトは、直接またはまとめて呼び出すことができます。まとめて呼び出すには、oneAPI インストール・ディレクトリーにある setvars.sh スクリプトを使用します。例えば、Linux* や macOS* マシンのデフォルトのインストール先は /opt/intel/setvars.sh になります。

引数なしで setvars.sh スクリプトを source すると、システムにインストールされているすべての <コンポーネント>/latest/env/vars.sh スクリプトが source されます。これらのスクリプトを source した後、env コマンドを使用して環境変数を確認できます。

注: setvars.sh スクリプト(または個別の vars.sh スクリプト)により変更された環境は永続的ではありません。これらの変更は、setvars.sh 環境スクリプトが source されたターミナルセッションでのみ有効です。

コマンドライン引数

setvars.sh スクリプトはいくつかのコマンドライン引数をサポートしており、--help オプションで引数の一覧を表示できます。

次に例を示します。

```
source /opt/intel/oneapi/setvars.sh --help
```

--config=file 引数と setvars.sh スクリプトから呼び出される vars.sh スクリプトへの追加引数をインクルードする機能を使用して、環境設定をカスタマイズできます。

--config=file 引数は、特定のインテル® oneAPI コンポーネントの環境の初期化機能を提供するとともに、特定のバージョンの環境を初期化することもできます。例えば、インテル® インテグレーテッド・パフォーマンス・プリミティブ(インテル® IPP)ライブラリーとインテル® oneAPI マス・カーネル・ライブラリー(インテル® oneMKL)の環境のみを設定するには、setvars.sh スクリプトにこれら 2 つのインテル® oneAPI コンポーネントの環境変数のみを設定することを指示する設定ファイルを渡します。詳細と利用例については、「Linux* や macOS* で setvars.sh の設定ファイルを使用」をご覧ください。

setvars.sh のヘルプメッセージに記載されていないコマンドライン引数は、vars.sh スクリプトに渡されます。つまり、setvars.sh スクリプトが認識できない引数は、コンポーネントの vars.sh スクリプトで使用されるものとみなし、それらの引数をすべてのコンポーネントの vars.sh スクリプトに渡します。最もよく使用される追加の引数は、ia32 と intel64 です。これらは、インテル® コンパイラー、インテル® IPP、インテル® MKL およびインテル® TBB ライブラリーでアプリケーションのターゲット・アーキテクチャーを指示するために使用されます。

個々の vars.sh スクリプトを調べて、受け入れるコマンドライン引数があればそれを決定します。

実行方法

```
source <install-dir>/setvars.sh
```

注:cdh など非 POSIX シェルを使用する場合、次のコマンドを使用します。

```
$ bash -c 'source <install-dir>/setvars.sh; exec csh'
```

または、modulefiles スクリプトを使用して開発環境を設定します。modulefiles スクリプトは、すべての Linux* シェルで機能します。

コンポーネントのリストとコンポーネントのバージョンを調整するには、setvars 設定ファイルを使用して開発環境を設定します。

確認方法

setvars.sh を source した後、SETVARS_COMPLETED 環境変数を検索して設定の成功を確認できます。setvars.sh が成功すると、SETVARS_COMPLETED には 1 が設定されます。

```
env | grep SETVARS_COMPLETED
```

戻り値

```
SETVARS_COMPLETED=1
```

SETVARS_COMPLETED=1 以外の場合、setvars.sh は設定に失敗したことを意味します。

複数の実行

各コンポーネントの env/vars.sh スクリプトの多くは、PATH、CPATH、およびそのほかの環境変数に変更を加えるため、最上位の setvars.sh スクリプトは同じセッションで同じ vars.sh を複数回呼び出すことはできません。これは、特に $PATH 環境変数が原因で環境変数の文字数が長くなりすぎないようにします。

これを強制するには、setvars.bat に --force オプションを指定します。この例では、ユーザーが setvars.sh を 2 回実行しています。setvars.sh がすでに実行されているため、2 回目の実行は停止します。

```
> source <install-dir>/setvars.sh
:: initializing environment ...
(SNIP: lot of output)
:: oneAPI environment initialized ::
```

```
> source <install-dir>/setvars.sh
 :: WARNING: setvars.sh has already been run. Skipping re-execution.
    To force a re-execution of setvars.sh, use the '--force' option.
    Using '--force' can result in excessive use of your environment variables
```

次は、ユーザーが `setvars.sh --force` を実行し、初期化が成功した例です。

```
> source <install-dir>/setvars.sh --force
 :: initializing environment ...
(SNIP: lot of output)
 :: oneAPI environment initialized ::
```

ONEAPI_ROOT 環境変数

`ONEAPI_ROOT` 環境変数は、スクリプトが実行されるときに最上位の `setvars.sh` によって設定されます。`ONEAPI_ROOT` 環境変数がすでに設定されている場合、`setvars.sh` はそれを上書きします。この変数は、`oneapi-cli` サンプルブラウザーと Eclipse* および Visual Studio* Code サンプルブラウザーによって使用され、インテル® oneAPI ツールとコンポーネントの検出、および `SETVARS_CONFIG` 機能が有効である場合に `setvars.sh` スクリプトを検出するのに役立ちます。`SETVARS_CONFIG` 機能の詳細については、「Eclipse* で setvars.sh スクリプトを自動化」をご覧ください。

Linux* と macOS* システムでは、インストーラーは `ONEAPI_ROOT` 環境変数を追加しません。これをデフォルト環境に追加するには、ローカルシェルの初期化ファイル (.bashrc など) または `/etc/environment` ファイルで `ONEAPI_ROOT` 変数を定義します。

Linux* または macOS* で setvars.sh 設定ファイルを使用

Linux* で環境を設定するには次の 2 つの方法があります。

- このページで示すように、`setvars.sh` 設定ファイルを使用します。
- `modulefiles` を使用します。

`setvars.sh` スクリプトは、それぞれの oneAPI ディレクトリーにある `<install-dir>/latest/env/vars.sh` スクリプトを source することで、インテル® oneAPI ツールキットで使用する環境変数を設定します。`setvars.sh` スクリプトを自動的に source するように Linux* システムを設定しない限り、新しいターミナルウィンドウを開くか Eclipse* またはそのほかの C/C++ IDE やエディターを起動する前に source する必要があります。詳細は、「システムの設定」(英語)を参照してください。

次に設定ファイルを使用して環境変数を管理する方法を説明します。

バージョンと構成

一部の oneAPI ツールは複数バージョンのインストールをサポートします。複数バージョンをサポートするツールのディレクトリー構造は次のようになります。

```
intel/oneapi/compiler/
|--2021.1.1
|--2021.2.0
`--latest → 2021.2.0
```

次に例を示します。

```
$ ls -l intel/oneapi/compiler/
total 8
drwxr-xr-x 8 ubuntu ubuntu 4096 Nov  9  2020 2021.1.1/
drwxrwxr-x 8 ubuntu ubuntu 4096 Apr  9 10:06 2021.2.0/
lrwxrwxrwx 1 ubuntu ubuntu    8 Apr  9 10:06 latest -> 2021.2.0/
$
```

すべてのツールには、そのコンポーネントの最新バージョンのインストール先を示す `latest` という名前のシンボリック･リンクがあります。`latest\env\` ディレクトリーにある `vars.sh` スクリプトは、`setvars.sh` によって source されます(デフォルト)。

必要に応じて、設定ファイルを使用して特定のディレクトリーを示すよう `setvars.sh` をカスタマイズできます。

`--config` パラメーター

最上位の `setvars.sh` スクリプトは、カスタム **config.txt** ファイルを指定する `--config` パラメーターを受け入れます。

```
> source <install-dir>/setvars.sh --config="full/path/to/your/config.txt"
```

設定ファイルは任意の名前にすることができます。複数の設定ファイルを作成して、さまざまな開発環境やテスト環境を設定できます。例えば、最新バージョンのライブラリーを古いバージョンのコンパイラーでテストしたいこともあります。そのような場合に、`setvars` 設定ファイルを使用して環境を管理できます。

設定ファイルの例

以下に簡単な設定ファイルの例を示します。

最新のコンポーネントをすべてロード

```
mkl=1.1
dldt=exclude
```

ただし以下のコンポーネントは除外

```
default=exclude
mkl=1.0
ipp=latest
```

設定テキストファイルは次の要件に従う必要があります。

- 改行で区切られたテキストファイル。
- 各行は、"key=value" のペアで構成されます。
- "key" には、oneAPI ディレクトリーの最上位($ONEAPI_ROOT ディレクトリーにあるフォルダー)のコンポーネント名を指定します。同じ "key" が設定ファイルに複数定義されると、最後の "key" が優先されそれ以外は無視されます。
- "value" には、コンポーネント・ディレクトリーの最上位にあるバージョン・ディレクトリー名を指定します。これには、コンポーネント・ディレクトリーのレベルに存在する可能性があるショートカット(latest など)が含まれます。
- また、"value" は "exclude" にすることもできます。これは、指定された "key" の環境変数スクリプトを setvars.sh スクリプトで source しないことを意味します。

"key=value" を "default=exclude" にすると特別な意味を持ちます。これは、設定ファイルに定義されているものを除き、それ以外のすべての env/vars.sh スクリプトの source を除外します。以下に例を示します。

設定ファイルのカスタマイズ

設定ファイルを使用して、特定のコンポーネントを除外したり、特定のバージョンを含めたり、特定のコンポーネントのバージョンのみを含めることができます。これには、設定ファイルの "default=exclude" 行を変更します。

デフォルトでは、setvars.sh は最新(latest)のバージョンに対応する env/vars.sh スクリプトを処理します。

例えば、2 つのバージョン(2021.1.1 と 2021.2.0)のインテル® oneMKL がインストールされていると仮定します。最新のバージョンを示すシンボリック・リンクは 2021.2.0 であるため、デフォルトで setvars.bat は mkl ディレクトリー 2021.2.0 の vars.sh スクリプトを実行します。

```
$ /usr/bin/tree -dL 2 --charset=ascii intel/oneapi/mkl/
intel/oneapi/mkl/
|-- 2021.1.1
|   |-- benchmarks
|   |-- bin
|   |-- documentation
|   |-- env
|   |-- examples
|   |-- include
|   |-- interfaces
|   |-- lib
|   |-- licensing
|   |-- modulefiles
|   `-- tools
|-- 2021.2.0
|   |-- benchmarks
|   |-- bin
|   |-- documentation
|   |-- env
|   |-- examples
|   |-- include
|   |-- interfaces
|   |-- lib
|   |-- licensing
|   |-- modulefiles
|   `-- tools
`-- latest -> 2021.2.0
```

特定のバージョンを指定

`setvars.sh` に `<install-dir>/mkl/2021.1.1/env/vars.sh` スクリプトを source するように直接記述するには、設定ファイルに `mkl=2021.1.1` を追加します。

これにより、`setvars.sh` は、`mkl` ディレクトリー内の `2021.1.1` フォルダーにある `env/vars.sh` スクリプトを source するようになります。インストールされている `mkl` 以外のコンポーネントでは、`setvars.sh` は最新バージョンのフォルダーにある `env/vars.sh` スクリプトを source します。

特定のコンポーネントを除外

コンポーネントを除外する構文は次のようになります。

```
<コンポーネント>=exclude
```

例えば、インテル® IPP を除外して、2021.1.1 のインテル® oneMKL を含めるには次のようにします。

```
mkl=2021.1.1
ipp=exclude
```

この例は次のように作用します。

- setvars.sh は、インテル® oneMKL 2021.1.1 の env/vars.sh スクリプトを source します。
- setvars.sh は、インテル® IPP の env/vars.sh スクリプトを source しません。
- setvars.sh は、そのほかのコンポーネントの最新バージョンの env/vars.sh スクリプトを source します。

特定のコンポーネントを含める

特定のコンポーネントの env/vars.sh スクリプトを source するには、最初にすべてのコンポーネントの env/vars.sh スクリプトを除外する必要があります。その後、setvars.sh で source するコンポーネントを追加し直します。次の行を定義して、すべてのコンポーネントの env/vars.sh スクリプトを source から除外します。

```
default=exclude
```

例えば、setvars.sh がインテル® oneMKL とインテル® IPP コンポーネントの env/vars.sh スクリプトのみを source するようにするには、次の設定ファイルを使用します。

```
default=exclude
mkl=2021.1.1
ipp=latest
```

この例は次のように作用します。

- setvars.sh は、インテル® oneMKL 2021.1.1 の env/vars.sh スクリプトを source します。
- setvars.sh は、インテル® IPP の最新バージョンの env/vars.sh スクリプトを source します。
- setvars.sh は、そのほかのコンポーネントの env/vars.sh スクリプトを source しません。

Eclipse* で servars.sh スクリプトを自動化

setvars.sh スクリプトは、インテル® oneAPI ツールキットを使用するために必要な環境変数を設定します。このスクリプトは、コマンドライン開発向けに新しいターミナルウィンドウを開くたびに実行する必要があります。setvars.sh スクリプトは、Eclipse* の起動時に自動的に実行することもできます。SETVARS_CONFIG 環境変数を使用して、setvars.sh スクリプトにインテル®oneAPI ツール固有の設定を行うように指示できます。

SETVARS_CONFIG 環境変数の状態

SETVARS_CONFIG 環境変数を使用して、C/C++ 開発者向けの Eclipse* IDE インスタンスを起動したときにインテル® oneAPI 開発環境を自動的に設定できます。環境変数には 3 つの条件と状態があります。

- 未定義(SETVARS_CONFIG 環境変数が存在しない)
- 定義されているが空(値を含まないか空白である)
- setvars.sh 設定ファイルを示すように定義します

SETVARS_CONFIG に値が設定されていない(空白のみが含まれる)場合、Eclipse* の起動時に setvars.sh スクリプトは自動的に実行されます。この場合、setvars.sh スクリプトはシステムにインストールされている**すべて**のインテル® oneAPI ツールの環境を初期化します。setvars.sh スクリプト実行の詳細については、「Eclipse* 使用したサンプル・プロジェクトのビルドと実行」(英語)を参照してください。

SETVARS_CONFIG に setvars 設定ファイルへの絶対パスが定義されている場合、Eclipse* の起動時に setvars.sh スクリプトは自動的に実行されます。この場合、setvars.sh スクリプトは、setvars 設定ファイルで定義されるインテル® oneAPI ツールのみの環境を初期化します。setvars 設定ファイルを作成する方法の詳細は、「Linux* と macOS* で setvars.sh 設定ファイルを使用」をご覧ください。

注:Eclipse* でのデフォルトの SETVARS_CONFIG の動作は、Windows* の Visual Studio* で説明されている動作とは異なります。Eclipse* を起動すると、setvars.sh スクリプトは常に自動的に実行されます。Windows* で Visual Studio* を起動すると、SETVARS_CONFIG 環境変数が定義されている場合のみ setvars.bat スクリプトは自動的に実行されます。

---

setvars 設定ファイルは任意の名前で作成でき、そのファイルが Eclipse* からアクセスおよび読み取り可能である限りハードディスク上のどこにでも保存できます (Linux* システムにインテル® oneAPI ツールをインストールしたときに Eclipse* に追加されたプラグインが SETVARS_CONFIG のアクションを実行するため、Eclipse* は setvars 設定ファイルにアクセスできる必要があります)。

setvars 設定ファイルを空のままにすると、setvars.sh スクリプトはシステムにインストールされている**すべて**のインテル® oneAPI ツールの環境を初期化します。これは、SETVARS_CONFIG 変数に空の文字列を定義するのと同じです。setvars 設定ファイルの作成方法については、「Linux* や macOS* で setvars.sh 設定ファイルを使用」を参照してください。

SETVARS_CONFIG 環境変数の定義

SETVARS_CONFIG 環境変数はインストール中に自動的に定義されないため、Eclipse* を起動する前に(前述の方法で)環境変数を追加する必要があります。SETVARS_CONFIG 環境変数には、次のようにさまざまな場所を定義できます。

- /etc/environment
- /etc/profile

- `~/.bashrc`
- など ...

上記の例は、Linux* システムで環境変数を定義する一般的な場所です。`SETVARS_CONFIG` 環境変数に定義する場所は、システムとニーズによって異なります。

3.4.3 Linux* でモジュールファイルを使用

Linux* で環境を設定するには、次の 2 つの方法があります。

- このページで説明されているようにモジュールファイルを使用します。
- `setvars.sh` 設定ファイルを使用します。

ほとんどのインテル® oneAPI コンポーネントのフォルダーには、それぞれのコンポーネントに必要な環境変数を設定するモジュールファイルのスクリプトが含まれています。モジュールファイルを使用して、`setvars.sh` スクリプトに代わって開発環境を設定できます。モジュールファイルでは引数がサポートされていないため、複数の設定(32 ビットや 64 ビットの設定など)をサポートする oneAPI およびライブラリーでは複数のモジュールファイルを使用します。

注: インテル® oneAPI ツールキットで提供されるモジュールファイルは、Tcl 環境モジュール(Tmod)および Lua 環境モジュール(Lmod)と互換性があり、次のバージョンがサポートされます。

- Tmod 3.2.10(コンパイラーのモジュールファイルには 4.1 が必要です。以下を参照してください。)
- Tcl バージョン 8.4
- Lmod バージョン 8.2.10

次のコマンドを使用して、システムにインストールされているバージョンを確認します。

```
module --version
```

各モジュールファイルは、実行時にシステムの Tcl バージョンが適切であるか自動的に確認します。

モジュールファイルのバージョンがサポートされていない場合、回避策があります。詳細については、「インテル開発ツールにおける環境モジュールの利用」(英語)を参照してください。

インテル® oneAPI 2021.4 のリリース以降、バージョン 3.2.10 の Tcl モジュールを使用する場合、icc モジュールファイルを使用して icc および ifort コンパイラーを設定できます。今後の oneAPI リリースでは、コンパイラーのモジュールファイルのサポートが解決される予定です。

インテル® oneAPI のモジュールスクリプトは、それぞれのコンポーネント・ディレクトリーにある modulefiles フォルダーにあります(個々の vars スクリプトの配置方法と同様)。例えば、デフォルトのインストールでは、ipp のモジュールスクリプトは、/opt/intel/ipp/latest/modulefiles/ ディレクトリーにあります。

インテル® oneAPI コンポーネントのフォルダーがどのように展開されているかにより、インテル® openAPI モジュールファイルをインストール先で直接使用することが難しい場合があります。そのため、oneAPI インストール・フォルダーには特殊な modulefiles-setup.sh スクリプトが用意されており、oneAPI モジュールファイルの操作を簡単にできます。デフォルトのインストールでは、設定スクリプトは /opt/intel/oneapi/modulefiles-setup.sh に配置されます。

modulefiles-setup.sh スクリプトは、インテル® oneAPI のインストールに含まれるすべてのモジュールスクリプトを検索し、それらのフォルダーを単一ディレクトリーに保存します。これらのフォルダーには、1 つ以上のモジュールファイルが含まれ、インストールされているインテル® oneAPI コンポーネントごとに名前が付けられます。modulefiles-setup.sh スクリプトの実行時に --ignore-latest オプションを使用すると、module_load コマンドでバージョンが指定されていない場合、最も高い semver バージョンのモジュールファイルがロードされます。

コンポーネントの名前が付けられたそれぞれのフォルダーには、そのインテル® oneAPI コンポーネントが提供するすべてのモジュールファイルを指すシンボリック・リンクが含まれます。シンボリック・リンクは、モジュールファイルのバージョンごとに編成されます。各コンポーネント・フォルダーには、少なくとも "latest"(最新)バージョンのモジュールファイルが含まれ、バージョンを指定しなくてもデフォルトで最新のコンポーネントをロードできます。

モジュールファイルのディレクトリーを作成

modulefiles-setup.sh スクリプトを実行します。

注: デフォルトでは、modulefiles-setup.sh スクリプトは、インテル® oneAPI ツールキットのインストール・フォルダーに modulefiles という名前のフォルダーを作成します。インストール・フォルダーが書き込み可能でない場合、--outputdir=<フォルダーのパス> オプションを使用して、書き込み可能な場所に modulefiles フォルダーを作成します。modulefiles-setup.sh スクリプトの詳細については、modulefiles-setup.sh -help を入力して確認できます。

---

modulefiles-setup.sh スクリプトが実行されると、次のような階層が最上位の modulefiles フォルダーに作成されます(modulefiles の正確なリストはインストールによって異なります)。この例では、インテル® Advisor 環境を設定する 1 つのモジュールファイルと、コンパイラー環境を設定する 2 つのモジュールファイルがあります(コンパイラーのモジュールファイルは、すべてのインテル® コンパイラーの環境を設定します)。最新のシンボリック・リンクをたどると、semver の規則に従って最上位バージョンのモジュールファイルを指します。

```
|-- advisor
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/advisor/2021.2.0/modulefiles/advisor
|   `-- latest -> /home/ubuntu/intel/oneapi/advisor/latest/modulefiles/advisor
|-- ccl
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/ccl/2021.1.1/modulefiles/ccl
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/ccl/2021.2.0/modulefiles/ccl
|   `-- latest -> /home/ubuntu/intel/oneapi/ccl/latest/modulefiles/ccl
|-- clck
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/clck/2021.1.1/modulefiles/clck
|   `-- latest -> /home/ubuntu/intel/oneapi/clck/latest/modulefiles/clck
|-- compiler
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler
|-- compiler-rt
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt
|-- compiler-rt32
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt32
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt32
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt32
|-- compiler32
|   |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler32
|   |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler32
|   `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler32
```

ここで、`MODULEFILESPATH` を更新して、`modulefiles-setup.sh` スクリプトで作成された新しい `modulefiles` フォルダーに含めるか、`moduleuse <folder_name>` コマンドを実行します。

システムに Tcl モジュールファイル環境をインストールする

次の手順は、Ubuntu* で環境モジュール･ユーティリティーを実行する例を示しています。モジュール･ユーティリティーのインストールと設定の詳細については、http://modules.sourceforge.net/ (英語) を参照してください。

```
$ sudo apt update
$ sudo apt install tcl
$ sudo apt install environment-modules
```

`tclsh` のローカルコピーが新しいものであることを確認します (サポートされているバージョンについては、このページの最初を参照してください)。

```
$ echo 'puts [info patchlevel] ; exit 0' | tclsh 8.6.8
```

`module` のインストールをテストするには、`module` エイリアスを初期化します。

```
$ source /usr/share/modules/init/sh
$ module
```

注: POSIX* 互換シェルの初期化は、上記の `source` コマンドで機能するはずです。それ以外のシェル固有の init スクリプトは、`/usr/share/modules/init/` フォルダーにあります。詳細については、各フォルダーと `man module` の初期化セクションをご覧ください。

モジュールのエイリアス `init` スクリプト(`.../modules/init/sh`)でグローバルまたはローカルの起動スクリプトを source して、モジュールコマンドが利用できるようにします。

`modulefiles-setup.sh` スクリプト入門

次のことが前提とされています。

- Linux* 開発システムに `tclsh` がインストールされている
- システムに環境モジュール・ユーティリティー(`module` など)がインストールされている
- init コマンドで `.../modules/init/sh`(または等価なシェル)が source されている
- oneAPI 開発に必要なインテル® oneAPI ツールキットがインストールされている

```
$ cd <oneapi-root-folder>       # cd to the oneapi_root install directory
$ ./modulefiles-setup.sh        # run the modulefiles setup script
$ module use modulefiles        # use the modulefiles folder created above
$ module avail                  # will show tbb/X.Y, etc.
$ module load tbb               # loads tbb/X.Y module
$ module list                   # should list the tbb/X.Y module you just loaded
$ module unload tbb             # removes tbb/X.Y changes from the environment
$ module list                   # should no longer list the tbb/X.Y env var
module
```

アンロードの前に、`env` コマンドを使用して環境を検証し、ロードしたモジュールファイルで変更された部分を探します。例えば、次のコマンドを実行すると、ロードした `tbb modulefile` で変更された環境の部分が表示されます(`modulefile` を調べてすべての変更を確認します)。

```
$ env | grep -i "intel"
```

注: `modulefile` はスクリプトですが、ユーザーがインストールおよび保守する `"module"` インタープリターによってロードおよび解釈されるため、'x'(実行可能)権限を設定する必要はありません。インテル® oneAPI ツールキットのインストールには、`modulefile` インタープリターは含まれていないため、個別にインストールする必要があります。同様に、`modulefiles` には 'w' 権限は必要ありませんが、読み取り可能である必要があります('r' 権限をすべてのユーザーに設定します)。

バージョン管理

インテル® oneAPI ツールキットのインストーラーは、それぞれのツールまたはライブラリーの最上位ディレクトリーでバージョン管理されたサブディレクトリー形式で、個々のインテル® oneAPI ツールおよびライブラリーのバージョンフォルダーを作成します。このようにバージョン管理されたコンポーネント・フォルダーは、モジュールファイルのバージョンを管理するために使用されます。これは基本的に modulefiles-setup.sh スクリプトによって行われます。上位レベルの [$(ONEAPI_ROOT)/modulefiles] フォルダーに作成されるシンボリック・リンクは [<モジュールファイル名>/バージョン] として構成されるため、module コマンドを使用してバージョンごとの modulefiles を参照できます。

```
$ module avail
---------------- modulefiles -----------------
ipp/1.1  ipp/1.2  compiler/1.0  compiler32/1.0
```

複数のモジュールファイル

ツールやライブラリーは、modulefiles フォルダー内に複数の modulefiles を持つ場合があります。それぞれがロード可能モジュールになります。展開されたコンポーネント・フォルダーごとにバージョンが割り当てられます。

oneAPI で使用するモジュールファイルの記述法を理解する

シンボリック・リンクは modulefiles-setup.sh スクリプトによって使用され、利用可能なすべてのモジュールファイルを単一の modulefiles フォルダーに集約します。これは、実際の modulefiles は移動または変更されないことを意味します。その結果、${ModulesCurrentModulefile} 変数は、それぞれのインストール・フォルダーにある実際のモジュールファイルではなく、各モジュールファイルへのシンボリック・リンクを指します。

このため、各モジュールファイルは、次のような形式を使用して、それぞれのインストール・ディレクトリーにある元の modulefile への参照を取得します。

```
[ file readlink ${ModulesCurrentModulefile} ]
```

さらに詳しく理解するには、インストールに含まれる modulefiles を確認してください。ほとんどの場合、実際のファイルへのシンボリック・リンク参照を解決するコメントと、バージョン番号 (およびバージョン・ディレクトリー) の解析が含まれます。また、インストールされた TCL が適切なバージョンであることを確認するチェックと、同じモジュールファイルの異なるバージョンを誤ってロードしないようにするチェックも含まれています。

modulefiles による module load コマンドの使用

modulefiles のいくつかは、module load コマンドを使用して依存関係のあるモジュールもロードできます。依存関係のあるモジュールファイルのバージョンチェックは行われません。これは、依存関係のあるモジュールを必要とするモジュールをロードする前に、その依存関係モジュールを手動で事前

ロードできることを意味します。依存関係があるモジュールを事前ロードしない場合、最新バージョンがロードされます。

これは、環境を柔軟にコントロールできるようにする設計によるものです。例えば、以前のバージョンのコンパイラーで更新されたバージョンのライブラリーをテストするためインストールすると仮定します。更新されたライブラリーにはバグ修正があり、更新されたライブラリー内の他のモジュールには興味がないという可能性もあります。依存するモジュールファイルが、ライブラリーの特定のバージョンを使用するようにハード・コーディングされている場合、このテストは機能しません。

注: 依存関係のあるモジュールのロード条件を満たせない場合、モジュールのロードは終了し環境は変更されません。

関連情報

`modulefiles` の詳細については、以下をご覧ください。

- http://www.admin-magazine.com/HPC/Articles/Environment-Modules(英語)
- https://support.pawsey.org.au/documentation/display/US/Sample+modulefile+for+Environment+Modules(英語)
- https://www.chpc.utah.edu/documentation/software/modules-advanced.php(英語)
- https://modules.readthedocs.io/en/latest/(英語)
- https://lmod.readthedocs.io/en/latest/(英語)

3.4.4 oneAPI アプリケーションで CMake* を使用

インテル® oneAPI 製品で提供される CMake* パッケージを使用すると、CMake* プロジェクトで Windows* または Linux* 上の oneAPI ライブラリーを簡単に利用できます。このパッケージを使用することで、他のシステム・ライブラリーが CMake* プロジェクトと統合する方法と同様の体験ができます。CMake* プロジェクトのターゲットに応じて、依存関係が生じたり、他のビルド変数が必要になることがあります。

次のコンポーネントが CMake* をサポートします。

- インテル® oneAPI DPC++ コンパイラー - Linux*、Windows*
- インテル® インテグレーテッド・パフォーマンス・プリミティブ(インテル® IPP)およびインテル® インテグレーテッド・パフォーマンス・プリミティブ Cryptgraphy(インテル® IPP Cryptgraphy)- Linux*、Windows*
- インテル® MPI ライブラリー - Linux*、Windows*

- インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー(インテル® oneCCL)- Linux*、Windows*
- インテル® oneAPI データ・アナリティクス・ライブラリー(インテル®oneDAL)- Linux*、Windows*
- インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー(インテル® oneDNN)- Linux*、Windows*
- インテル® oneAPI DPC++ ライブラリー(インテル®oneDPL)- Linux*、Windows*
- インテル® oneAPI マス・カーネル・ライブラリー(インテル® oneMKL)- Linux*、Windows*、macOS*
- インテル® oneAPI スレッディング・ビルディング・ブロック(インテル®oneTBB)- Linux*、Windows*、macOS*
- インテル® oneAPI ビデオ・プロセシング・ライブラリー(インテル® oneVPL)- Linux*、Windows*

すべてのインテル® oneAPI コンポーネント製品が、CMake* をサポートするわけではありません。CMake* 設定を提供するライブラリーは、以下で識別できます。

- Linux* または macOS*:
  - システム:`/usr/local/lib/cmake`
  - ユーザー: `~/lib/cmake`
- Windows*: `HKEY_LOCAL_MACHINE\Software\Kitware\CMake\Packages\`

CMake* パッケージを使用するには、他のシステム・ライブラリーと同様にインテル® oneAPI ライブラリーを使用します。例えば、`find_package(tbb)` を使用すると、アプリケーションの CMake* パッケージはインテル® oneTBB パッケージを使用します。

oneAPI プログラムのコンパイルと実行

4

この章では、CPU、GPU、および FPGA を対象とするダイレクト・プログラミングと API ベースのプログラミングにおける oneAPI のコンパイル手順について詳しく説明します。また、コンパイルの各ステージで使用されるツールについても詳しく説明します。

4.1 単一ソースのコンパイル

oneAPI プログラミング・モデルは、単一ソースのコンパイルをサポートします。単一ソースのコンパイルには、ホストとデバイスコードを個別にコンパイルする場合と比較して多くの利点があります。oneAPI プログラミング・モデルは、一部のユーザーの要望に答え、ホストコードとデバイスコードの個別コンパイルもサポートすることを覚えておいてください。単一ソースのコンパイルモデルには次のような利点があります。

- 利便性 - 開発者は作成するファイル数を最小限に抑え、ホストコードの呼び出し元の直後にデバイスコードを定義できます。
- 安全性 - 単一ソースでは、単独のコンパイラーがホストとデバイス間の境界にあるコードを判断することができ、ホスト・コンパイラーによって生成される仮引数がデバイス・コンパイラーによって生成されるカーネルの実引数と一致します。
- 最適化 - デバイス・コンパイラーは、カーネルが起動されるコンテキストを知ることができるため、さらなる最適化を行うことができます。例えば、コンパイラーはいくつかの定数を伝搬したり、関数呼び出し全体でポインターのエイリアシング情報を推測することができます。

4.2 コンパイラーの起動

インテル® oneAPI DPC++/C++ コンパイラーを起動するには、DPC++ アプリケーション向けには `dpcpp` を、OpenMP* オフロードを使用する C++ アプリケーション向けには `icpx -fiopenmp`(Linux*) または `icx /Qiopenmp`(Windows*)を使用します。詳細については、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドとリファレンス』の「コンパイラーの起動」を参照してください。

C++ アプリケーションをコンパイルする際に OpenMP* を有効にするには、次のコマンドでコンパイラーを起動します。

- `icpx -fiopenmp -fopenmp-targets=<arch>`(Linux*)

- `icx /Qiopenmp /Qopenmp-targets:<arch>`(Windows*)

DPC++ アプリケーションをコンパイルする際に OpenMP* を有効にするには、次のコマンドでコンパイラーを起動します。

- `dpcpp -fiopenmp -fopenmp-targets=<arch>`(Linux*)
- `dpcpp-cl /Qiopenmp /Qopenmp-targets:<arch>`(Windows*)

オプションの詳細については、インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドの「コンパイラー・オプション」を参照してください。

dpcpp コンパイラー・ドライバーは、OS ホストごとに互換性が異なります。Linux* では GCC スタイルのコマンドライン・オプションが提供され、Windows* では Microsoft* Visual Studio* の Microsoft Visual C++* 互換の機能が提供されます。

- GCC 形式のコマンドライン・オプション(「-」で開始)を認識し、複数のオペレーティング・システムでビルドシステムを共有するプロジェクトに役立ちます。
- Windows* コマンドライン・オプション(「/」で開始)を認識し、Microsoft Visual Studio* ベースのプロジェクトで使用できます。

4.3 インテル® oneAPI DPC++/C++ コンパイラーの標準オプション

インテル® oneAPI DPC++/C++ コンパイラーの完全なオプションリストは、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドとリファレンス』に記載されています。

- 「オフロード向けのコンパイルオプションと OpenMP* オプションおよび並列処理オプション」には、SYCL* と OpenMP* オフロードに固有のオプションが示されています。
- 利用可能なすべてのオプションと簡単な説明は、「コンパイラー・オプションのリスト(アルファベット順)」にあります。

4.4 コンパイル例

oneAPI アプリケーションは、ダイレクト・プログラミング、利用可能なインテル® oneAPI ライブラリーを使用する API ベース、およびそれらを組み合わせて記述することができます。API ベースのプログラミングでは、ライブラリーの機能を使用してデバイスへのオフロードを行います。これにより、開発者はアプリケーションを開発する時間を節約できます。一般に、API ベースのプログラミングから始め、ニーズに対応できない場合に SYCL* または OpenMP* オフロード機能を導入するのが最も容易であると考えられます。

次のセクションでは、API ベースのコードと SYCL* を使用したダイレクト・プログラミングの例を紹介します。

4.4.1 API ベースのコード

次のコードは、インテル® oneAPI マス・カーネル・ライブラリーの `oneapi::mkl::blas::axpy` 関数を使用して、浮動小数点数のベクトル全体に対して `a` と `x` を乗算して `y` を加算する API 呼び出し (`a * x + y`)の使用法を示しています。このサンプルコードは、oneAPI プログラミング・モデルを利用して、アクセラレーターで加算を実行します。

```
#include <vector> // std::vector()
#include <cstdlib> // std::rand()
#include <CL/sycl.hpp>
#include "oneapi/mkl/blas.hpp"

int main(int argc, char* argv[]) {

    double alpha = 2.0;
    int n_elements = 1024;

    int incx = 1;
    std::vector<double> x;
    x.resize(incx * n_elements);
    for (int i=0; i<n_elements; i++)
        x[i*incx] = 4.0 * double(std::rand()) / RAND_MAX - 2.0;
        // -2.0 から 2.0 の範囲の rand 値

    int incy = 3;
    std::vector<double> y;
    y.resize(incy * n_elements);
    for (int i=0; i<n_elements; i++)
        y[i*incy] = 4.0 * double(std::rand()) / RAND_MAX - 2.0;
        // -2.0 から 2.0 の範囲の rand 値

    cl::sycl::device my_dev;
    try {
        my_dev = cl::sycl::device(cl::sycl::gpu_selector());
    } catch (...) {
        std::cout << "Warning, failed at selecting gpu device. Continuing on
default(host) device.\n";
    }

    // 非同期例外をキャッチ
    auto exception_handler = [] (cl::sycl::exception_list
        exceptions) {
        for (std::exception_ptr const& e : exceptions) {
            try {
                std::rethrow_exception(e);
             } catch(cl::sycl::exception const& e) {
                std::cout << "Caught asynchronous SYCL exception:\n";
                std::cout << e.what() << std::endl;
            }
        }
    };

    cl::sycl::queue my_queue(my_dev, exception_handler);

    cl::sycl::buffer<double, 1> x_buffer(x.data(), x.size());
    cl::sycl::buffer<double, 1> y_buffer(y.data(), y.size());

    // y = alpha*x + y を計算
    try {
        oneapi::mkl::blas::axpy(my_queue, n_elements, alpha, x_buffer,
        incx, y_buffer, incy);
    }
```

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




    std::cout << "The axpy (y = alpha * x + y) computation is complete!" <<
    std::endl;




    // y_buffer をプリント
    auto y_accessor = y_buffer.template
      get_access<cl::sycl::access::mode::read>();
    std::cout << std::endl;
    std::cout << "y" << " = [ " << y_accessor[0] << " ]\n";
    std::cout << "    [ " << y_accessor[1*incy] << " ]\n";
    std::cout << "    [ " << "... ]\n";
    std::cout << std::endl;

    return 0;
}
```

アプリケーション(`axpy.cpp` として保存)をコンパイルするには、次の操作を行います。

1. `MKLROOT` 環境変数が適切に設定されていることを確認します(`echo ${MKLROOT}` - Linux*、`echo %MKLROOT%` - Windows*)。正しく設定されていない場合、`setvars.sh`(Linux*)または `setvars.bat`(Windows*)スクリプトを実行するか、`lib` および `include` サブディレクトリーを含むディレクトリー・パスを変数に設定します。

   `setvars` スクリプトの詳細については、「oneAPI 開発環境の設定」を参照してください。

2. 次のコマンドでアプリケーションをビルドします。

   Linux*:

   ```
   dpcpp -I${MKLROOT}/include -c axpy.cpp -o axpy.o
   ```

   Windows*:

   ```
   dpcpp -I${MKLROOT}/include /EHsc -c axpy.cpp /Foaxpy.obj
   ```

3. 次のコマンドでアプリケーションをリンクします。

Linux*:

```
dpcpp axpy.o -fsycl-device-code-split=per_kernel \
"${MKLROOT}/lib/intel64"/libmkl_sycl.a -Wl,-export-dynamic -Wl,--start-group
\
"${MKLROOT}/lib/intel64"/libmkl_intel_ilp64.a \
"${MKLROOT}/lib/intel64"/libmkl_sequential.a \
"${MKLROOT}/lib/intel64"/libmkl_core.a -Wl,--end-group -lsycl -lOpenCL \
-lpthread -lm -ldl -o axpy.out
```

Windows*:

```
dpcpp axpy.obj -fsycl-device-code-split=per_kernel ^
"${MKLROOT}/lib/intel64"/libmkl_sycl.lib ^
"${MKLROOT}/lib/intel64"/libmkl_intel_ilp64.lib ^
"${MKLROOT}/lib/intel64"/libmkl_sequential.lib ^
"${MKLROOT}/lib/intel64"/libmkl_core.lib ^
sycl.lib OpenCL.lib -o axpy.exe
```

4. 次のコマンドでアプリケーションを実行します。

Linux*:

```
./axpy.out
```

Windows*:

```
axpy.exe
```

4.4.2 ダイレクト・プログラミング

ここでは、「ベクトル加算のサンプルコード」(英語)を使用します。oneAPI プログラミング・モデルを利用して、アクセラレーターで加算を実行します。

次のコマンドで、実行形式をコンパイルしてリンクします。

```
dpcpp vector_add.cpp
```

コマンドとオプションのコンポーネントと関数は、「API ベースのコード」セクションで説明したものと類似しています。

このコマンドを実行すると、実行時にベクトルの加算を実行する実行ファイルが作成されます。

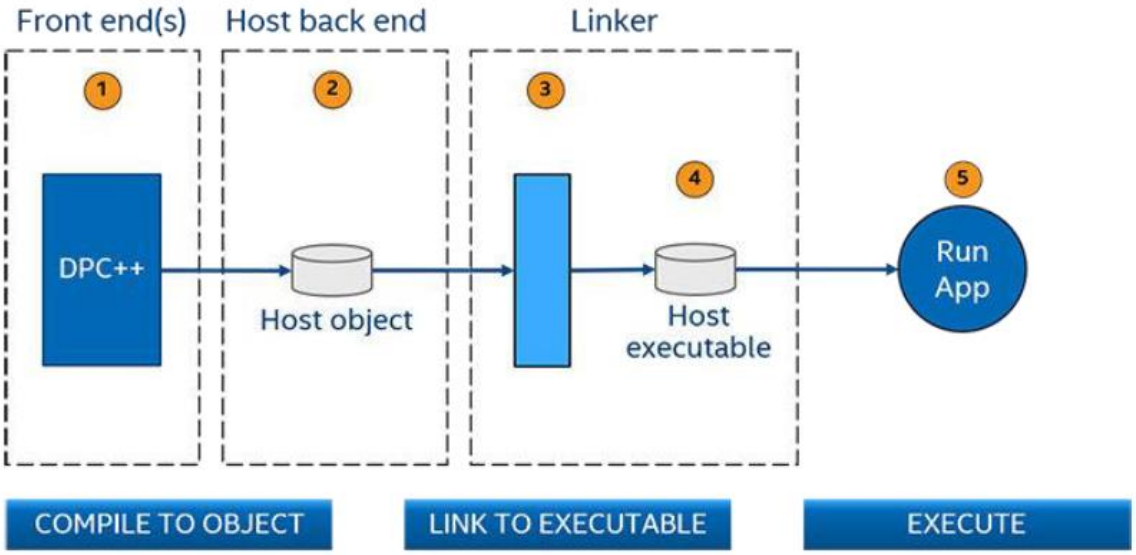
4.5 コンパイルの手順

オフロードを行うプログラムを作成する場合、コンパイラーはホストとデバイス向けの両方のコードを生成する必要があります。oneAPI は、この複雑な作業を開発者から見えないようにします。開発者は、DPC++ コンパイラーを使用して SYCL* アプリケーションをコンパイルするだけで(一度のコンパイルコマンドで)、ホストとデバイス向けのコードを生成できます。

デバイスコードの実行にはジャストインコンパイル(JIT : Just-in-Time)コンパイルと事前(AOT : Ahead-of-Time)コンパイルの 2 つのオプションがありますが、JIT がデフォルトです。このセクションでは、ホストコードのコンパイル方法と、デバイスコードを生成する 2 つの方法を説明します。詳しくは、『データ並列 C++ の書籍』(英語)の 13 章をご覧ください。

4.5.1 伝統的なコンパイル手順(ホストのみのアプリケーション)

従来のコンパイル手順は、C、C++、またはそのほかの言語で利用される標準のコンパイル方法です。デバイスへのオフロードがない場合に使用されます。



コンパイル手順を次の図に示します。

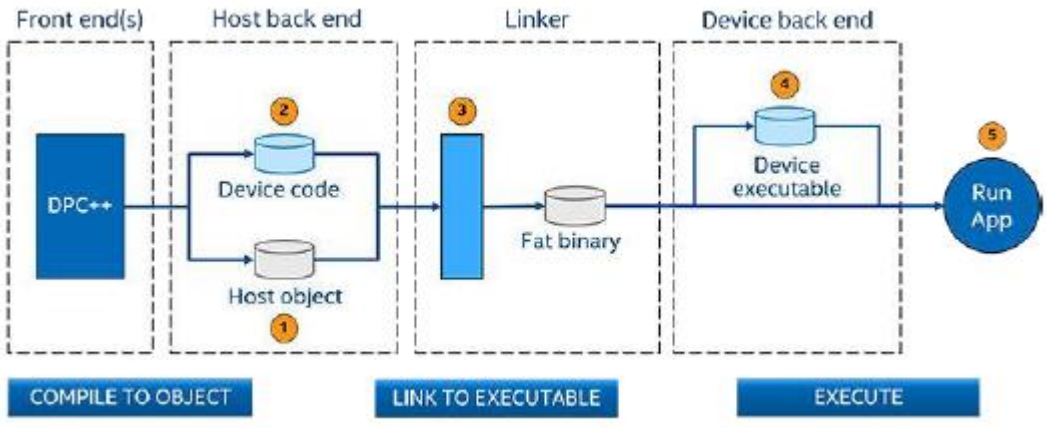
1. フロントエンドは、ソースを中間表現に変換し、バックエンドに渡します。
2. バックエンドは、中間表現をオブジェクト・コードに変換してオブジェクト・ファイル(Windows* では `host.obj`、Linux* では `host.o`)を出力します。
3. 1 つ以上のオブジェクト・ファイルがリンカーに渡されます。
4. リンカーは実行ファイルを生成します。
5. これで、アプリケーションを実行できます。

4.5.2 SYCL* オフロードコードのコンパイル手順

SYCL* オフロードコードのコンパイル手順には、従来のコンパイル手順にデバイスコードの JIT および AOT オプションを追加します。この手順では、開発者は dpcpp を使用して、SYCL* アプリケーションをコンパイルし、出力としてホストコードとデバイスコードの両方を含む実行可能ファイルを作成します。

SYCL* オフロードコードの基本コンパイル手順を次に示します。

コンパイル手順を次の図に示します。



1. ホストコードは、バックエンドでオブジェクト・コードに変換されます。
2. デバイスコードは SPIR-V* 形式またはデバイスバイナリーに変換されます。
3. リンカーは、ホスト・オブジェクト・コードとデバイスコード(SPIR-V* またはデバイスバイナリー)を組み合わせた、(デバイスコードが埋め込まれた)ホスト実行コードを含むファットバイナリーを生成します。
4. 実行されると、オペレーティング・システムはホスト・アプリケーションを起動します。オフロードが行われる場合、SYCL* ランタイムはデバイスコードをロードします(必要に応じて SPIR-V* をデバイスバイナリーに変換します)。
5. アプリケーションは、利用可能なホストとデバイスで実行されます。

4.5.3 JIT のコンパイル手順

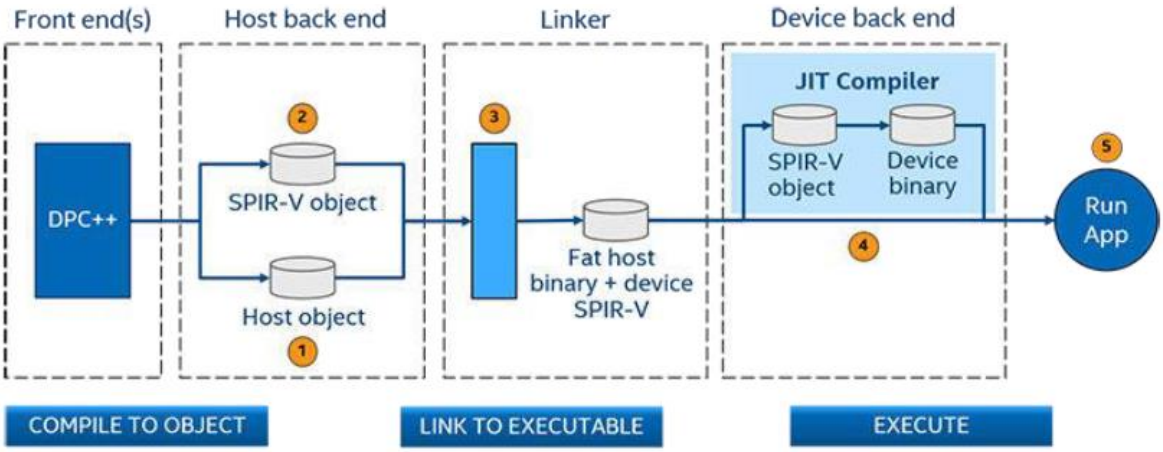
JIT コンパイルの手順では、デバイスコードはバックエンドで SPIR-V* 形式の中間コードに変換され、SPIR-V* としてファットバイナリーに組み込まれ、ランタイムによって SPIR-V* からデバイスバイナリーに変換されます。アプリケーションが起動されると、ランタイムは利用可能なデバイスを判別してそのデバイス固有のコードを生成します。これにより、AOT(事前)コンパイル手順よりも、アプリケーションの実行環境とパフォーマンスの柔軟性が高まります。しかし、アプリケーションの実行時にコンパイル(JIT)が行

われるため、アプリケーションの実行時間が増加する可能性があります。大量のデバイスコードを持つ大規模なアプリケーションでは、パフォーマンスへの影響が顕著に表れることがあります。

ヒント: JIT コンパイル手順は、ターゲットデバイスが不明である場合に役立ちます。

注: JIT コンパイラーは、FPGA デバイスではサポートされません。

コンパイル手順を次の図に示します。



1. ホストコードは、バックエンドでオブジェクト・コードに変換されます。
2. デバイスコードは SPIR-V* 形式に変換されます。
3. リンカーは、ホスト・オブジェクト・コードとデバイス SPIR-V* を組み合わせた、(SPIR-V* が埋め込まれた)ホスト実行コードを含むファットバイナリーを生成します。
4. 実行されると次のように処理されます。

   a. ホスト上のデバイスランタイムは、デバイスの SPIR-V* をデバイスのバイナリーに変換します。
   b. 変換されたデバイスバイナリーはデバイスへロードされます。

5. アプリケーションは、実行時に利用可能なホストとデバイスで実行されます。

4.5.4 AOT のコンパイル手順

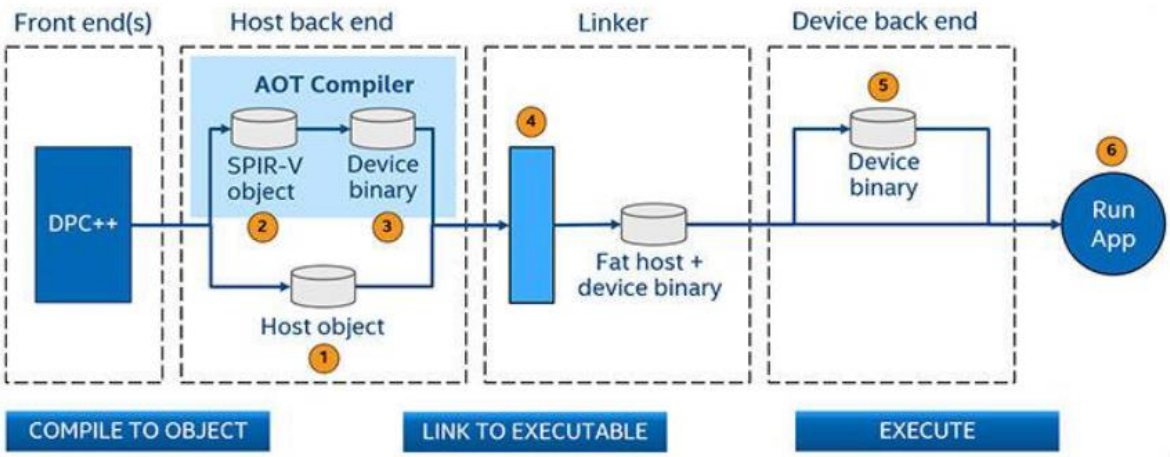
AOT(事前)コンパイルでは、デバイスコードが SPIR-V* に変換されてから、ホスト・バックエンドのデバイスコードに変換され、最終的に生成されたデバイスコードがファットバイナリーに組み込まれます。JIT 手

順とは異なり、AOT 手順では実行ファイルが作成される前にデバイスバイナリーに変換されます。AOT 手順では、コンパイル時にターゲットデバイスを決定する必要があるため、JIT 手順よりも柔軟性は低くなります。しかし、実行ファイルの起動時間は JIT 手順よりも短くなります。

ヒント:

- AOT 手順は、ターゲットとするデバイスが明確に判明している場合に適しています。
- AOT 手順では、デバッグサイクルが高速化されるため、アプリケーションをデバッグする際の利用が推奨されます。

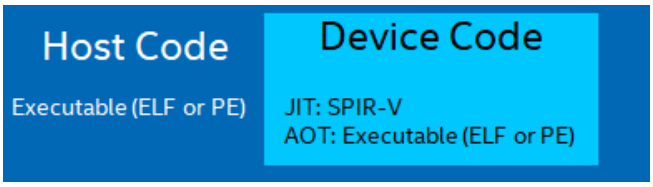
コンパイル手順を次の図に示します。



1. ホストコードは、バックエンドでオブジェクト・コードに変換されます。
2. デバイスコードは SPIR-V* 形式に変換されます。
3. デバイスの SPIR-V* は、コマンドラインでユーザーが指定したデバイス向けのデバイス・コード・オブジェクトに変換されます。
4. リンカーは、ホスト・オブジェクト・コードとデバイス・オブジェクト・コードを組み合わせた、デバイスバイナリーが埋め込まれたホスト実行コードを含むファットバイナリーを生成します。
5. 実行時に、デバイスバイナリーはデバイスへロードされます。
6. アプリケーションは、ホストと指定されたデバイスで実行されます。

4.5.5 ファットバイナリー

ファットバイナリーは、JIT と AOT コンパイル手順から生成されます。デバイスコードが埋め込まれたホストバイナリーです。デバイスコード自体は、コンパイル手順によって異なります。



- ホストコードは、ELF(Linux*)または PE(Windows*)形式の実行ファイルです。
- デバイスコードは、JIT 手順では SPIR-V*、AOT 手順ではデバイスバイナリーです。実行ファイルは次のいずれかの形式です。
  - CPU:ELF(Linux*)、PE(Windows*)
  - GPU:ELF(Windows*、Linux*)
  - FPGA:ELF(Linux*)、PE(Windows*)

4.6 CPU 手順

CPU はコンピューターの頭脳とも呼ばれ、分岐予測、メモリーの仮想化、命令のスケジュールなどを含む複雑な回路/アルゴリズムで構成されています。このような複雑性を考慮し、CPU は幅広いタスクを処理するように設計されています。

SYCL* および OpenMP* オフロードを利用するプログラミング・モデルにより、異種 CPU および GPU システムでのアプリケーションの実装が可能になります。SYCL* および OpenMP* オフロードにおける「デバイス」は、CPU と GPU の両方を指す場合があります。

最新の CPU には、並列計算に利用可能なハイパースレッディングと広い SIMD 幅を備えた複数のコアを搭載しています。ワークロードに計算集約型で並列実行可能な領域が存在する場合、そのような領域は GPU や FPGA などのコプロセッサーではなく CPU へオフロードすることを推奨します。これはまた、データを PCIe* を介してオフロードする必要がないため(コプロセッサーや GPU とは異なり)、データ転送のオーバーヘッドを最小限に抑えてレイテンシーを軽減します。

CPU でアプリケーションを実行するには、CPU で触接実行される従来型の CPU フローと、CPU デバイスで実行される CPU オフロードの 2 つのオプションがあります。CPU オフロードは、SYCL* または OpenMP* オフロード・アプリケーションで使用できます。OpenMP* オフロード・アプリケーションと SYCL* オフロード・アプリケーションは、どちらも OpenCL* ランタイムとインテル® oneAPI スレッディング・ビルディング・ブロック(インテル® oneTBB)を使用して、CPU デバイスで実行できます。

ヒント: ワークロードが CPU、GPU、または FPGA に最適であるか不明な場合は、「各種 oneAPI ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語)を参照してください。

4.6.1 従来の CPU 向け手順

従来の CPU ワークフローは、ランタイムなしで CPU 上で動作します。コンパイルフローは、C、C++ または他の言語で使用されるようなデバイスへのオフロードを行わない標準的なコンパイルです。

従来のワークロードでは、コンパイルフローの概要で説明されているように、従来のコンパイルフロー(ホスト専用アプリケーション)手順を使用して、ホスト上でコンパイルおよび実行されます。

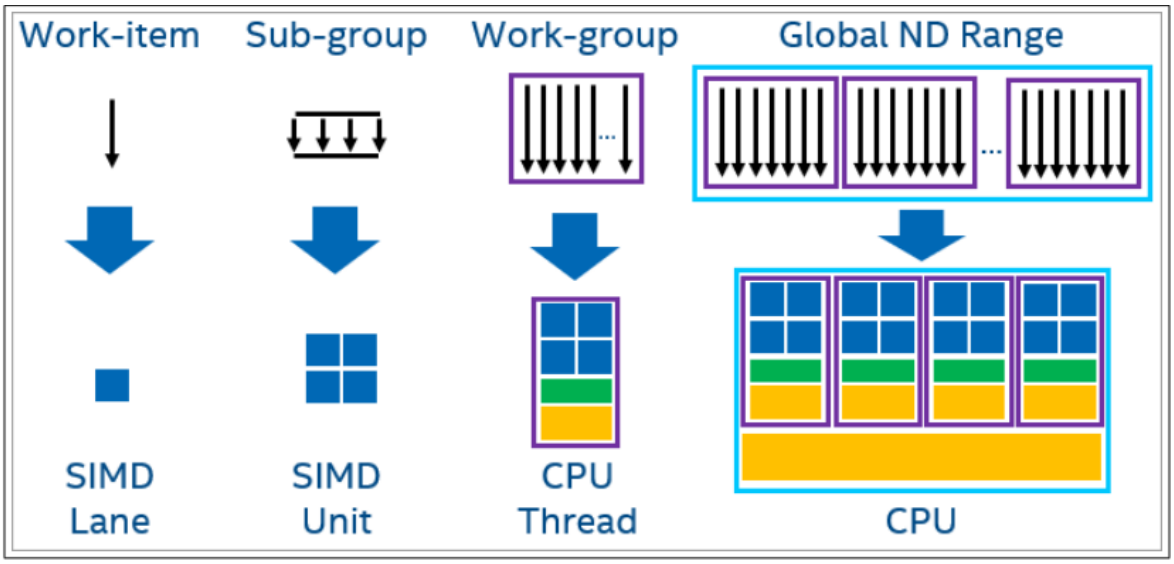
コンパイルコマンドの例。

```
icpx -g -o matrix_mul_omp src/matrix_mul_omp.cpp
```

4.6.2 CPU オフロード向け手順

デフォルトでは、CPU デバイスへオフロードする場合、OpenCL* ランタイムを使用します。OpenCL* ランタイムは、並列処理にインテル® oneAPI スレッディング・ビルディング・ブロック(インテル® oneTBB)も活用します。

CPU にオフロードする場合、ワークグループは異なる論理コアに割り当てられ、これらのワークグループは並列に実行できます。ワークグループ内のワーク項目は、CPU の SIMD レーンにマップできます。ワーク項目(サブグループ)は、SIMD 方式で同時に実行されます。



CPU 実行の詳細については、「各種 oneAPI コンピューティング・ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語)を参照してください。

CPU オフロード向けの設定

1. setvars スクリプトの実行を含む、oneAPI 開発環境の設定セクションのすべての手順を実行したことを確認します。
2. sycl-ls コマンドを実行して、必要な OpenCL* ランタイムが CPU に関連付けられていることを確認します。

```
$ sycl-ls

CPU : OpenCL 2.1 (Build 0)[ 2020.11.12.0.14_160000 ]
GPU : OpenCL 3.0 NEO [ 21.33.20678 ]
GPU : 1.1[ 1.2.20939 ]
```

3. 次のサンプルコードを使用して、コードが CPU で実行されていることを確認します。サンプルコードは、整数の大きなベクトルにスカラーを加算し、結果を検証します。

SYCL*

SYCL* では CPU で実行するための組込みデバイスセレクターが用意されています。これは、device_selector 基本クラスを使用し cpu_selector で CPU デバイスを選択できます。

または、default_selector を使用して、実装定義されたヒューリスティックに従って次の環境変数によって実行時にデバイスを選択することもできます。

```
export SYCL_DEVICE_FILTER=cpu
```

SYCL* サンプルコード:

```
#include <CL/sycl.hpp>
#include <array>
#include <iostream>

using namespace sycl;
using namespace std;
constexpr size_t array_size = 10000;
int main(){
constexpr int value = 100000;
try{
     cpu_selector d_selector;
     queue q(d_selector);
     int *sequential = malloc_shared<int>(array_size, q);
     int *parallel = malloc_shared<int>(array_size, q);
     // シーケンシャル iota
     for (size_t i = 0; i < array_size; i++) sequential[i] = value + i;

     // SYCL* の並列 iota
     auto e = q.parallel_for(range{array_size}, [=](auto i) {
        parallel[i] = value + i;
     });
```

```
      e.wait();
   // 2 つの結果が等しいか検証
   for (size_t I = 0; I < array_size; i++) {
     if (parallel[i] != sequential[i]) {
        cout << "Failed on device.\n";
        return -1;
     }
   }
   free(sequential, q);
   free(parallel, q);
}catch (std::exception const &e) {
   cout << "An exception is caught while computing on device.\n";
   terminate();
}
 cout << "Successfully completed on device.\n";
 return 0;
}
```

次のコマンドでサンプルコードをコンパイルします。

```
dpcpp simple-iota-dp.cpp -o simple-iota
```

追加のコマンドはサンプルの CPU コマンドから取得できます。

生成したバイナリーを実行します。

```
./simple-iota

Running on device: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Successfully completed on device.
```

OpenMP*

OpenMP* のサンプルコードを以下に示します。

```
#include<omp.h>
#define N 1024

int main(){
   float *a = (float *)malloc(sizeof(float)*N);
   for(int i = 0; i < N; i++) a[i] = i;

   #pragma omp target teams distribute \
                           parallel for simd map(tofrom: a[:N])
   for(int i = 0; i < 1024; i++)
      a[i]++; std::cout<<a[100]<<"\n";

   return 0;
}
```

次の環境変数を使用して、CPU での実行向けにサンプルコードをコンパイルします。

```
export LIBOMPTARGET_DEVICETYPE=cpu
```

次のコマンドでサンプルコードをコンパイルします。

```
icpx simple-ompoffload.cpp -fiopenmp -fopenmp-targets=spir64 -o simple-ompoffload
```

生成したバイナリーを実行します。

```
./simple-ompoffload

Successfully completed on device
```

CPU へコードをオフロード

アプリケーションをオフロードする場合、ボトルネックとオフロードの利点が得られるコード領域を特定することが重要です。計算集約型、または高度なデータ並列カーネルコードがある場合、そのコード領域をオフロードすることを検討してください。

オフロードするコード領域を特定するには、インテル® Advisor のオフロードのモデル化が役立ちます。

オフロードされたコードのデバッグ

以下のリストには、オフロードされるコードの基本的なデバッグのヒントが示されています。

- ホストターゲットをチェックしてコードが正しいことを確認します。
- `printf` を使用して、アプリケーションをデバッグします。SYCL* と OpenMP* オフロードでは、どちらもカーネルコードで `printf` がサポートされます。
- 環境変数を設定して詳細なログ情報を取得します。
  - SYCL* では、次のデバッグ環境変数を利用できます。すべての環境変数については GitHub* から入手できます。

環境変数		値	説明
SYCL_DEVICE_FILTER		backend:device_type:device_num	GitHub* の説明を参照してください。
SYCL_PI_TRACE		1|2|-1	1: SYCL*/DPC++ ランタイムプラグインの基本トレースログを出力します 2: SYCL*/DPC++ ランタイムプラグインのすべての API トレースを出力します

			-1: "2" のすべてと追加のデバッグ情報を出力します。

- OpenMP* では、次のデバッグ環境変数が推奨されます。完全なリストは LLVM/OpenMP* のドキュメント(英語)から入手できます。

環境変数	値	説明
`LIBOMPTARGET_DEVICETYPE`	cpu|gpu|host	デバイスを選択します。
`LIBOMPTARGET_DEBUG`	1	詳細なデバッグ情報を出力します。
`LIBOMPTARGET_INFO`	LLVM/OpenMP* のドキュメント(英語)で利用可能な値	ユーザーが `libomptarget` からさまざまなタイプのランタイム情報を要求できるようにします。

- 事前(AOT)コンパイルを使用して、ジャストインタイム(JIT)コンパイルを AOT コンパイルに移行します。詳細については、CPU アーキテクチャー向けの事前コンパイルを参照してください。

oneAPI で利用可能なデバッグ手法とデバッグツールの詳細については、『SYCL* および OpenMP* オフロードプロセスのデバッグ』を参照してください。

CPU コードの最適化

CPU オフロードコードのパフォーマンスに影響する可能性がある多くの要因があります。ワーク項目、ワークグループ、および実行されるワーク量は、CPU コア数によって異なります。

- コアで実行されるワーク量が計算集約型でない場合、パフォーマンスが低下する可能性があります。これは、スケジュールのオーバーヘッドとスレッドのコンテキスト切り替えが原因です。
- CPU では、PCIe* を介したデータ転送が不要であり、オフロード領域がデータを長時間待機する必要がないため、レイテンシーが低くなります。
- アプリケーションの性質に基づいて、スレッド・アフィニティーは CPU のパフォーマンスに影響を与える可能性があります。詳細については、『マルチコアにおけるバイナリーの実行制御』を参照してください。
- JIT コンパイルに代わって AOT コンパイル(オフラインコンパイル)を使用します。オフラインコンパイルでは、コードを特定の CPU アーキテクチャーに固定できます。詳細については、『CPU アーキテクチャー向けの最適化オプション』を参照してください。

『オフロード・パフォーマンスの最適化』から推奨事項を入手できます。

CPU コマンドの例

次のコマンドは、デバイスコードの一部が静的ライブラリーにある実装を想定しています。

注: 動的ライブラリーとのリンクはサポートされていません。

デバイスコードを使用してファット・オブジェクトを生成します。

```
dpcpp -c static_lib.cpp
```

ar ツールを使用して、静的ファット・ライブラリーを作成します。

```
ar cr libstlib.a static_lib.o
```

アプリケーションのソースをコンパイルします。

```
dpcpp -c a.cpp
```

静的ライブラリーとアプリケーションをリンクします。

```
dpcpp -foffload-static-lib=libstlib.a a.o -o a.exe
```

CPU アーキテクチャー向けの事前(AOT)コンパイル

事前(AOT)コンパイル・モードでは、最適化オプションを使用して、特定の CPU アーキテクチャーでの実行を改善するコードを生成できます。

```
dpcpp -fsycl-targets=spir64_x86_64 -Xs "-device <CPU optimization flags>" a.cpp
b.cpp -o app.out
```

次の CPU 最適化オプションがサポートされます。

```
-march=<instruction_set_arch> ターゲットの命令セット・アーキテクチャーを設定:
'sse42' インテル® ストリーミング SIMD 拡張命令 4.2
'avx2' インテル® アドバンスト・ベクトル・エクステンション 2
'avx512' インテル® アドバンスト・ベクトル・エクステンション 512
```

注: サポートされる最適化オプションは、将来のリリースで変更される可能性があります。

複数の CPU コア上でバイナリーの実行をコントロール

環境変数

次の環境変数は、プログラムの実行中に複数の CPU コアでの SYCL* または OpenMP* スレッドの配置をコントロールします。OpenCL* ランタイム CPU デバイスを使用して CPU にオフロードする場合、これらの変数を使用します。

環境変数	説明
DPCPP_CPU_CU_AFFINITY	CPU へスレッド・アフィニティーを設定します。以下を指定できます。 • close - スレッドは利用可能な CPU コアに連続して配置されます。 • spread - スレッドは利用可能なコアに分散されます。 • master - スレッドはマスタースレッドと同じコアに配置されます。DPCPP_CPU_CU_AFFINITY が設定されているとマスタースレッドも固定されます(未設定では固定されません)。 この環境変数は、OpenMP* の OMP_PROC_BIND 環境変数に似ています。 **デフォルト:**未設定
DPCPP_CPU_SCHEDULE	スケジューラーが work-group をスケジュールするアルゴリズムを指定します。現在、DPC++ は TBB のスケジューラーを使用しています。この値は、インテル® TBB スケジューラーが使用するパーティショナーを選択します。以下を指定できます。 • dynamic - インテル® TBB の auto_partitioner。負荷を分散するため適切な分割を行います。 • affinity - インテル® TBB の affinity_partitioner。subrange をワーカースレッドにマッピングすることで、auto_partitioner のキャッシュ・アフィニティーを向上させます。 • static - インテル® TBB の static_partitioner。range の反復をワーカースレッド間でできるだけ均等に分散します。TBB パーティショナーは、grainsize を使用してチャンクの大きさを制御します。デフォルトの grainsize は 1 であり、すべての work-group を独立して実行できることを示します。 **デフォルト:**dynamic
DPCPP_CPU_NUM_CUS	カーネルの実行に使用するスレッド数を設定します。 オーバーサブスクリプション状態を回避するには、DPCPP_CPU_NUM_CUS の最大値をハードウェア・スレッド数にする必要があります。DPCPP_CPU_NUM_CUS が 1 である場合、すべての work-group は単一のスレッドで順番に実行されます。これはデバッグ時に役立ちます。 この環境変数は、OpenMP* の OMP_NUM_THREADS に似ています。 **デフォルト:**未設定。インテル® TBB によって決定されます。

環境変数	説明
DPCPP_CPU_PLACES	アフィニティーを設定する場所を指定します。値は、{ sockets | numa_domains | cores | threads } のいずれかです。 この環境変数は、OpenMP* の OMP_PLACES 環境変数に似ています。 numa_domains が選択されると、インテル® TBB NUMA API が使用されます。これは、OpenMP* 5.1 の OMP_PLACES=numa_domains に似ています。インテル® TBB の task arena は、numa ノードにバインドされ SYCL* nd-range は task arena に均一に分散されます。 DPCPP_CPU_PLACES は、DPCPP_CPU_CU_AFFINITY とともに使用することを推奨します。 **デフォルト:**cores

サポートされる環境変数の詳細については、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドとリファレンス』(英語)をご覧ください。

例 1:ハイパースレッディング有効

2 ソケットで、ソケットごとに 4 つの物理コアがあり、それぞれの物理コアには 2 つのハイパースレッドがあるマシンを想定します。

- S<num> リストで指定される 8 つのコアを持つソケット番号を示します。
- T<num> インテル® oneAPI スレッディング・ビルディング(インテル® oneTBB)のスレッド番号を示します。
- "-" は未使用のコアを意味します。

```
DPCPP_CPU_NUM_CUS=16
   export DPCPP_CPU_PLACES=sockets
   DPCPP_CPU_CU_AFFINITY=close:
S0:[T0 T1 T2 T3 T4 T5 T6 T7]       S1:[T8 T9 T10 T11 T12 T13 T14 T15]
   DPCPP_CPU_CU_AFFINITY=spread:
S0:[T0 T2 T4 T6 T8 T10 T12 T14]    S1:[T1 T3 T5 T7 T9 T11 T13 T15]
   DPCPP_CPU_CU_AFFINITY=master:
S0:[T0 T1 T2 T3 T4 T5 T6 T7]       S1:[T8 T9 T10 T11 T12 T13 T14 T15]

   export DPCPP_CPU_PLACES=cores
   DPCPP_CPU_CU_AFFINITY=close:
S0:[T0 T8 T1 T9 T2 T10 T3 T11]     S1:[T4 T12 T5 T13 T6 T14 T7 T15]
   DPCPP_CPU_CU_AFFINITY=spread:
S0:[T0 T8 T2 T10 T4 T12 T6 T14]    S1:[T1 T9 T3 T11 T5 T13 T7 T15]
   DPCPP_CPU_CU_AFFINITY=master:
S0:[T0 T1 T2 T3 T4 T5 T6 T7]       S1:[T8 T9 T10 T11 T12 T13 T14 T15]

   export DPCPP_CPU_PLACES=threads
   DPCPP_CPU_CU_AFFINITY=close:
S0:[T0 T1 T2 T3 T4 T5 T6 T7]       S1:[T8 T9 T10 T11 T12 T13 T14 T15]
   DPCPP_CPU_CU_AFFINITY=spread:
S0:[T0 T2 T4 T6 T8 T10 T12 T14]    S1:[T1 T3 T5 T7 T9 T11 T13 T15]
   DPCPP_CPU_CU_AFFINITY=master:
S0:[T0 T1 T2 T3 T4 T5 T6 T7]       S1:[T8 T9 T10 T11 T12 T13 T14 T15]

   export DPCPP_CPU_NUM_CUS=8
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close close:
S0:[T0 - T1 - T2 - T3 -]           S1:[T4 - T5 - T6 - T7 -]
   DPCPP_CPU_CU_AFFINITY=close spread:
S0:[T0 - T2 - T4 - T6 -]           S1:[T1- T3 - T5 - T7 -]
   DPCPP_CPU_CU_AFFINITY=close master:
S0:[T0 T1 T2 T3 T4 T5 T6 T7]       S1:[]
```

例 2:ハイパースレッディング無効

2 ソケットで、ソケットごとに 4 つの物理コアがあり、それぞれの物理コアには 2 つのハイパースレッドがあるマシンを想定します。

- S<num> リストで指定される 8 つのコアを持つソケット番号を示します。
- T<num> インテル® oneTBB のスレッド番号を示します。
- "-" は未使用のコアを意味します。

```
export DPCPP_CPU_NUM_CUS=8
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close:    S0:[T0 T1 T2 T3]   S1:[T4 T5 T6 T7]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0 T2 T4 T6]   S1:[T1 T3 T5 T7]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3]   S1:[T4 T5 T6 T7]

export DPCPP_CPU_NUM_CUS=4
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close:    S0:[T0  -  T1 - ]    S1:[T2 - T3 - ]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0  -  T2 - ]    S1:[T1 - T3 - ]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3]     S1:[ -  -  -  - ]
```

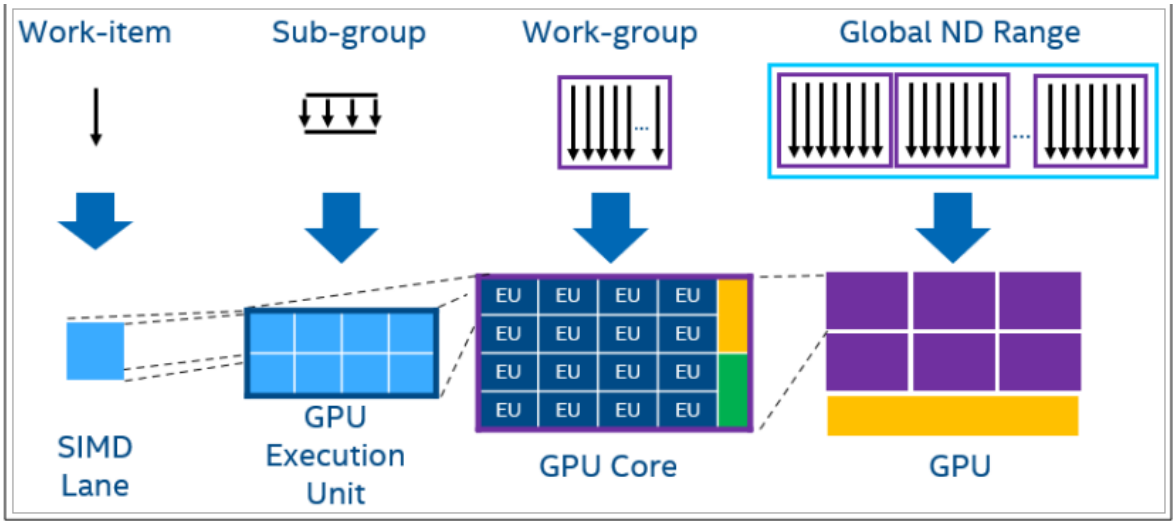
4.7 GPU 手順

GPU は、アプリケーションの計算集約型領域の負荷を軽減する用途で使用できる特殊な計算デバイスです。通常、GPU は多数の小規模なコアで構成され、大量のスループットをもたらします。タスクには、CPU に適したものと GPU に適したものがあります。

注: ワークロードが CPU、GPU、または FPGA に適しているか不明な場合、各種 oneAPI コンピューティング・ワークロードに対する CPU、GPU、および FPGA の利点を比較します。

4.7.1 GPU オフロードの手順

プログラムを GPU にオフロードすると、デフォルトでレベル 0 のランタイムが利用されます。OpenCL* ランタイムに切り替えるオプションも用意されています。SYCL* および OpenMP* オフロードでは、各ワーク項目は SIMD レーンにマップされます。サブグループは並列に実行されるワーク項目で形成される SIMD 幅に分割され、サブグループは GPU の EU スレッドにマップされます。ローカルデータを同期または共有するワーク項目を含むワークグループは、計算ユニット (ストリーミング・マルチプロセッサーまたは Xe コア – サブスライスとも呼ばれます) での実行に割り当てられます。最後に、ワーク項目のグローバル ND-Range 全体が GPU 全体にマップされます。



GPU 実行の詳細については、「各種 oneAPI コンピューティング・ワークロードに対する CPU、GPU、および FPGA の利点の比較」(英語) を参照してください。

GPU オフロードの設定

1. `setvars` スクリプトの実行を含む、oneAPI 開発環境のすべての設定手順を実行したことを確認します。
2. ドライバーをインストールして GPU システムを構成し、ユーザーを video グループに追加します。手順については、導入ガイドを参照してください。

   - インテル® oneAPI ベース・ツールキット導入ガイド (Linux* (英語) | Windows* (英語) | macOS* (英語))
   - インテル® oneAPI HPC ツールキット導入ガイド (Linux* (英語) | Windows* (英語) | macOS* (英語))
   - インテル® oneAPI IoT ツールキット導入ガイド (Linux* (英語) | Windows* (英語))

3. `sycl-ls` コマンドを使用して、サポートされている GPU と必要なドライバーがインストールされていることを確認します。次の例では、OpenCL* およびレベル 0 ドライバーがインストールされている場合、GPU に関連付けられたランタイムごとに 2 つのエントリーが表示されています。

```
CPU : OpenCL 2.1 (Build 0)[ 2020.11.12.0.14_160000 ]
GPU : OpenCL 3.0 NEO [ 21.33.20678 ]
GPU : 1.1[ 1.2.20939 ]
```

4. 次のサンプルコードを使用して、コードが GPU で実行されていることを確認します。サンプルコードは、整数の大きなベクトルにスカラーを加算し、結果を検証しています。

SYCL*

SYCL* では GPU で実行するための組込みデバイスセレクターが用意されています。これは、`device_selector` 基本クラスを使用し `gpu_selector` で GPU デバイスを選択できます。独自のカスタムセレクターを作成することもできます。詳細については、『Data Parallel C++』書籍のデバイス節「C++ および SYCL を使用したヘテロジニアス・プログラミング向けに SYCL をマスターする」を参照してください。

SYCL* サンプルコード:

```
#include <CL/sycl.hpp>
#include <array>
#include <iostream>

using namespace sycl;
using namespace std;
constexpr size_t array_size = 10000;
int main(){
constexpr int value = 100000;
try{
     //
     // デフォルトのデバイスセレクターは、最もパフォーマンスが高いデバイスを選択します。
     default_selector d_selector;
```

```
    queue q(d_selector);

    // USM を使用した共有メモリー割り当て。
    int *sequential = malloc_shared<int>(array_size, q);
    int *parallel = malloc_shared<int>(array_size, q);
    //Sequential iota
    for (size_t i = 0; i < array_size; i++) sequential[i] = value + i;

    // SYCL* の並列 iota
    auto e = q.parallel_for(range{array_size}, [=](auto i) {
       parallel[i] = value + i;
    });
    e.wait();
    // 2 つの結果を検証。
    for (size_t i = 0; i < array_size; i++) {
       if (parallel[i] != sequential[i]) {
          cout << "Failed on device.\n";
          return -1;
       }
    }
    free(sequential, q);
    free(parallel, q);
}catch (std::exception const &e) {
    cout << "An exception is caught while computing on device.\n";
    terminate();
}
    cout << "Successfully completed on device.\n";
    return 0;
}
```

次のコマンドでサンプルコードをコンパイルします。

```
dpcpp simple-iota-dp.cpp -o simple-iota
```

生成されたバイナリーを実行します。

```
./simple-iota
Running on device: Intel® UHD Graphics 630 [0x3e92]
Successfully completed on device.
```

OpenMP*

OpenMP* のサンプルコード:

```
#include <stdlib.h>
#include <omp.h>
#include <iostream>
constexpr size_t array_size = 10000;
#pragma omprequires unified_shared_memory
int main(){
constexpr int value = 100000;
```

```
// デフォルトのターゲットデバイスを返します。
int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device() :
omp_get_initial_device();
int *sequential = (int *)omp_target_alloc_host(array_size, deviceId);
int *parallel = (int *)omp_target_alloc(array_size, deviceId);

     for (size_t i = 0; i < array_size; i++)
        sequential[i] = value + i;

     #pragma omp target parallel for
     for (size_t i = 0; i < array_size; i++)
        parallel[i] = value + i;

     for (size_t i = 0; i < array_size; i++) {
        if (parallel[i] != sequential[i]) {
           std::cout << "Failed on device.\n";
           return -1;
        }
     }

     omp_target_free(sequential, deviceId);
     omp_target_free(parallel, deviceId);

     std::cout << "Successfully completed on device.\n";
     return 0;
}
```

次のコマンドでサンプルコードをコンパイルします。

```
icpx simple-iota-omp.cpp -fiopenmp -fopenmp-targets=spir64 -o simple-iota
```

生成されたバイナリーを実行します。

```
./simple-iota

Successfully completed on device.
```

注： オフロード領域が存在し、アクセラレーターがない場合、`OMP_TARGET_OFFLOAD=mandatory` 環境変数が指定されない限り、カーネルは従来のホストコンパイル（OpenCL* ランタイムなし）にフォールバックします。

GPU へコードをオフロード

どの GPU ハードウェアで、どのコード領域をオフロードするか決定するには、「GPU 最適化ワークフロー・ガイド」（英語）を参照してください。

オフロードするコード領域を特定するには、インテル® Advisor のオフロードのモデル化が役立ちます。

GPU コードのデバッグ

以下のリストには、オフロードされるコードの基本的なデバッグのヒントが示されています。

- CPU またはホスト/ターゲットをチェック、またはランタイムを OpenCL* に切り替えてコードが正しいことを確認します。
- `printf` を使用して、アプリケーションをデバッグします。SYCL* と OpenMP* オフロードでは、どちらもカーネルコードで `printf` がサポートされます。
- 環境変数を設定して詳細なログ情報を取得します。

DPC++ を使用した SYCL* では、次のデバッグ環境変数を利用できます。すべての環境変数については GitHub* から入手できます。

環境変数	値	説明
`SYCL_DEVICE_FILTER`	backend:device_type:device_num	GitHub* の説明を参照してください。
`SYCL_PI_TRACE`	1|2|-1	1: SYCL* ランタイムプラグインの基本トレースログを出力します 2: SYCL* ランタイムプラグインのすべての API トレースを出力します -1: "2" のすべてと追加のデバッグ情報を出力します。
`ZE_DEBUG`	任意の値で定義された変数 – 有効になします。	この変数が SYCL* ランタイムで使用される場合、レベルゼロからのデバッグ出力を有効にします。呼び出されるレベルゼロ API やレベルゼロのイベント情報をレポートします。

- OpenMP* では、次のデバッグ環境変数が推奨されます。完全なリストは LLVM/OpenMP* のドキュメント (英語) から入手できます。

環境変数	値	説明
`LIBOMPTARGET_DEVICETYPE`	cpu|gpu|host	デバイスを選択します。
`LIBOMPTARGET_DEBUG`	1	詳細なデバッグ情報を出力します。
`LIBOMPTARGET_INFO`	LLVM/OpenMP* のドキュメント (英語) で利用可能な値	ユーザーが `libomptarget` からさまざまなタイプのランタイム情報を要求できるようにします。

- 事前 (AOT) コンパイルを使用して、ジャストインタイム (JIT) コンパイルを AOT コンパイルに移行します。詳細については、CPU アーキテクチャー向けの事前コンパイルを参照してください。

oneAPI で利用可能なデバッグ手法とデバッグツールの詳細については、『SYCL* および OpenMP* オフロードプロセスのデバッグ』を参照してください。

GPU コードの最適化

オフロードコードされたコードを最適化するにはいくつかの方法があります。次の表には、最適化のヒントが示されています。詳細については、「oneAPI GPU 最適化ガイド」(英語)を参照してください。

- ホストとデバイス間のメモリー転送のオーバーヘッドを削減します。
- コアをビジー状態に維持し、データ転送のオーバーヘッドのコストを軽減するため十分な量のワークを実行します。
- GPU キャッシュ、共有ローカルメモリーなど GPU メモリー階層を活用して、メモリーアクセスを高速化します。
- JIT コンパイルの代わりに AOT コンパイル(オフラインコンパイル)を使用します。事前コンパイルでは、コードを特定の GPU アーキテクチャーをターゲットにできます。詳細については、GPU 向けの事前(AOT)コンパイルを参照してください。
- インテル® GPU Occupancy Calculator を使用すると、特定のカーネルおよびワークグループのパラメーターに対するインテル® GPU の占有率を計算できます。

オフロード・パフォーマンスの最適化で追加の推奨事項を入手できます。

4.7.2 GPU コマンドの例

次の例は、Linux* でデバイスコードを使用して静的ライブラリーを作成し、使用する方法を示します。

注: 動的ライブラリーとのリンクはサポートされていません。

デバイスコードを使用してファット・オブジェクトを生成します。

```
dpcpp -c static_lib.cpp
```

`ar` ツールを使用して、静的ファット・ライブラリーを作成します。

```
ar cr libstlib.a static_lib.o
```

アプリケーションのソースをコンパイルします。

```
dpcpp -c a.cpp
```

静的ライブラリーとアプリケーションをリンクします。

```
dpcpp -foffload-static-lib=libstlib.a a.o -o a.exe
```

4.7.3 GPU 向けの事前(AOT)コンパイル

次のコマンドは、特定の GPU ターゲット向けに `app.out` を生成します。

DPC++:

```
dpcpp -fsycl-targets=spir64_gen -Xs "-device <device name>" a.cpp b.cpp -o
app.out
```

OpenMP*:

```
icpx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device
<device name>" a.cpp b.cpp -o app.out
```

デバイス名に利用できる値の一覧は、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』(英語)から入手できます。

4.8 FPGA 手順

フィールド・プログラマブル・ゲートアレイ(FPGA)は、任意の機能を実行するように繰り返しプログラム可能な集積回路です。FPGA は、空間コンピューティング・アーキテクチャーとして分類され、CPU や GPU のような固定命令セット・アーキテクチャー(ISA)デバイスとは異なり、従来のアクセラレーター・デバイスとは異なる最適化のトレードオフが求められます。

SYCL* は、CPU、GPU、または FPGA 向けの SYCL* コードをコンパイルできますが、FPGA 開発用のコンパイル手順は、CPU や GPU 開発におけるコンパイル手順とは多少異なります。

次の表に FPGA 手順の説明で使用される用語をまとめています。

FPGA 手順固有の用語

用語	定義
デバイスコード	ホストではなくデバイスで実行される SYCL* ソースコード。デバイスコードは、ラムダ式、ファンクター、またはカーネルクラスを介して指定されます(例、カーネルコード)。
ホストコード	ホスト・コンパイラーによってコンパイルされ、デバイスではなくホストで実行される

	SYCL* ソースコード。
デバイスイメージ	デバイスコードをバイナリー(または中間)表現にコンパイルした結果のファイル。デバイスイメージは、(ファット)オブジェクトまたは実行ファイル内でホスト・オブジェクトと結合されます。「コンパイル手順」を参照してください。
FPGA エミュレーター・イメージ	FPGA エミュレーター向けにコンパイルされたデバイスイメージ。「FPGA エミュレーター」を参照してください。
FPGA 初期イメージ	初期コンパイルステージによるデバイスイメージ。「FPGA 最適化レポート」を参照してください。
FPGA ハードウェア・イメージ	ハードウェア・イメージのコンパイルステージによるデバイスイメージ。「FPGA 最適化レポート」と「FPGA ハードウェア」を参照してください。

注 : FPGA プログラミングの詳細は、https://link.springer.com/chapter/10.1007/978-1-4842-5574-2_17(英語)から入手できるデータ並列 C++ の書籍からも学ぶことができます。

4.8.1 FPGA 向けのコンパイルが特殊である理由

FPGA はいくつかの点で CPU や GPU とは異なります。CPU や GPU と比較した大きな違いは、FPGA ハードウェア専用のデバイスバイナリーが必要なことです。これはほとんどの場合、計算集約型で時間がかかる処理です。FPGA のコンパイルが完了するまで数時間かかるのは通常の動作です。

そのため、FPGA 開発向けに AOT(またはオフライン)カーネル・コンパイル・モードのみがサポートされます。FPGA ハードウェアのコンパイルには時間がかかるため、JIT(またはオンライン)コンパイルは実用的ではありません。

コンパイル時間が長くなるほど、開発者の生産性は悪化します。インテル® oneAPI DPC++/C++ コンパイラーは、FPGA をターゲットとする開発設計をすばやく反復できるメカニズムを提供します。可能な限り完全な FPGA コンパイルにかかる手順を回避することで、CPU および GPU 開発に近い感覚で高速コンパイル時間の恩恵を受けられます。

4.8.2 SYCL* FPGA コンパイルの種別

SYCL* はアクセラレーター全般をサポートします。インテル® oneAPI DPC++/C++ コンパイラーは、FPGA コードの開発を支援するため、FPGA 固有のサポートを提供します。このドキュメントでは、FPGA をターゲットとする SYCL* コード開発向けのコンパイラーと関連するツールを使用した FPGA 向けの開発を紹介します。

以下の表に、FPGA コンパイルの要約を示します。

デバイス・イメージ・タイプ	コンパイル時間	説明
FPGA エミュレーター	数秒	FPGA デバイスコードは CPU 向けにコンパイルされます。OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォームを使用して、SYCL* コードが正しく機能することを確認します。
FPGA シミュレーション	数分	FPGA デバイスコードは CPU にコンパイルされます。Questa*- インテル® シミュレーター FPGA エディション・ソフトウェアを使用して、コードをデバッグします。
最適化レポート	数分	FPGA デバイスコードは部分的にハードウェア向けにコンパイルされます。コンパイラーは、FPGA で生成される構造とパフォーマンスのボトルネックを特定し、リソース利用率を推測して最適化レポートを生成します。
FPGA ハードウェア・イメージ	数時間	ターゲットの FPGA プラットフォームで実行する FPGA ビットストリームを生成します。

一般的な FPGA SYCL* 開発ワークフローでは、これらの各ステージで手順を繰り返し、それぞれのステージからもたらされるフィードバックを反映してコードを改良します。可能な限りエミュレーションと FPGA 最適化レポートを活用することを推奨します。

ヒント: FPGA エミュレーション向けにコンパイルしたり、FPGA 最適化レポートを生成するには、インテル® oneAPI ベース・ツールキットのインテル® oneAPI DPC++/C++ コンパイラーのみが必要です。FPGA ハードウェアのコンパイルには、oneAPI ベース・ツールキット対応のインテル® FPGA アドオン(英語)が必要です。アドオンのインストール手順の詳細は、『インテル® oneAPI ツールキット・インストール・ガイド』(英語)をご覧ください。

FPGA エミュレーター

FPGA エミュレーター(OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォーム)は、コードの正当性を検証する最速の手法です。CPU 上で SYCL* デバイスコードを実行します。FPGA エミュレーターは、SYCL* ホストデバイスに似ていますが、ホストデバイスとは異なり、FPGA パイプや `fpga_reg` など FPGA 拡張機能がサポートされます。詳細については、『インテル® oneAPI DPC++ FPGA 最適化ガイド』の「パイプ拡張」(英語)と「カーネル変数」(英語)を参照してください。

次に FPGA エミュレーターを使用する際に覚えておくべき重要事項を示します。

- **パフォーマンスは代表的なものではない**

FPGA エミュレーターは、FPGA デバイスの動作そのものを再現するものではないため、パフォーマンスの評価に使用しないでください。例えば、FPGA で 100 倍のパフォーマンス向上をもたらす最適化は、エミュレーターのパフォーマンスには影響しないか、多少の増減を示すことがあります。

- **未定義の動作は異なる可能性があります**

FPGA エミュレーターと FPGA ハードウェア向けにコンパイルしたコードの結果が異なる場合、コードに未定義の動作が含まれる可能性があります。未定義の動作は言語仕様では指定されておらず、ターゲットごとに振る舞いが異なる可能性があります。

ヒント: FPGA エミュレーター・デバイスをターゲットにする場合、-O2 コンパイルオプションを使用して最適化を有効にしてエミュレーターを高速化します。(デバッグを容易にするため)最適化をオフにするには -O0 オプションを指定します。

エミュレーションの詳細については、「設計をエミュレート」を参照してください。

FPGA シミュレーター

シミュレーション手順では、Questa*- インテル® FPGA エディション・シミュレーター・ソフトウェアを使用して、合成されたカーネルの正確な動作をシミュレートできます。エミュレーションと同様に、ターゲットの FPGA ボードが装着されていないシステムでシミュレーションを実行できます。シミュレーターは、エミュレーションよりも正確にカーネルをモデル化できますが、エミュレーターよりもかなり低速です。

シミュレーション手順は、サイクル精度とビット精度を目的とし、カーネルのデータパスの動作と浮動小数点データ型の操作結果を正確にモデル化します。ただし、シミュレーションでは、可変レイテンシー・メモリーやその他の外部インターフェイスを正確にモデル化することはできません。シミュレーションは FPGA ハードウェアやエミュレーターよりもかなり低速であるため、小規模な入力データセットで設計をシミュレーションすることを推奨します。

シミュレーション手順をプロファイルと組み合わせて利用することで、設計に関する追加情報を取得できます。プロファイルの詳細については、『インテル® oneAPI DPC++ FPGA 最適化ガイド』の「DPC++ 向けインテル® FPGA ダイナミック・プロファイラー」(英語)を参照してください。

注: GNU Project Debugger(GDB*)、Microsoft* Visual Studio* または通常のソフトウェア・デバッガーを使用して、シミュレーション向けにコンパイルされたカーネルコードをデバッグすることはできません。

シミュレーション手順の詳細については、「シミュレーションによるカーネルの評価(ベータ版)」を参照してください。

FPGA 最適化サポート

完全な FPGA コンパイルは次のステージで行われ、最適化レポートは両方のステージの後に出力されます。

ステージ	説明	最適化レポートの情報
FPGA 初期イメージ (コンパイルには数分かかります)	SYCL* デバイスコードは最適化され、Verilog レジスター転送レベル(RTL)(FPGA の低レベルの設計入力言語)で指定される FPGA 設計に変換されます。これにより、実行形式ファイルではない FPGA 初期イメージが生成されます。 このステージで生成される最適化レポートは、静的レポートと呼ばれることがあります。	これには、コンパイラーが SYCL* デバイスコードから FPGA 設計に変換した手法に関する重要な情報が含まれています。レポートには次の情報が含まれます。 • FPGA で生成された構造の可視化情報 • パフォーマンスと予期されるパフォーマンスのボトルネックに関する情報 • リソースの利用状況を推測
FPGA ハードウェア・イメージ (コンパイルには数時間を要します)	設計回路のトポロジーを指定する Verilog RTL は、インテル® Quartus® Prime 開発ソフトウェアによって FPGA のプリミティブ・ハードウェア・リソースにマッピングされます。コンパイルステージに必要なインテル® Quartus® Prime 開発ソフトウェアは、oneAPI ベース・ツールキット用インテル® FPGA アドオン(英語)に含まれています。結果は、FPGA ビットストリーム(またはバイナリー)を含む FPGA ハードウェア・イメージです。	リソースと f_{max} 数に関する正確な情報が含まれます。レポートの解析に関する詳細は、『インテル® oneAPI DPC++ FPGA 最適化ガイド』の「設計の解析」(英語)を参照してください。

FPGA ハードウェア

これは FPGA ハードウェア・イメージへの完全なコンパイルです。インテル® プログラマブル・アクセラレーション・カード(インテル® Arria® 10 GX FPGA 搭載版)、インテル® FPGA PAC D5005(インテル® FPGA プログラマブル・アクセラレーション・カード D5005)、またはカスタムボードをターゲットにできます。

インテル® PAC またはカスタムボードを使用する詳細については、「FPGA BSP とボード」を参照してください。

4.8.3 FPGA コンパイルオプション

FPGA コンパイルオプションは、インテル® oneAPI DPC++/C++ コンパイラーがターゲットとする FPGA イメージタイプをコントロールします。

```
# FPGA emulator image
dpcpp -fintelfpga -DFPGA_EMULATOR fpga_compile.cpp -o fpga_compile.fpga_emu

# FPGA simulator image
dpcpp -fintelfpga fpga_compile.cpp -Xssimulation -Xsboard=intel_s10sx_pac:pac_s10

# FPGA early image (with optimization report): default board
dpcpp -fintelfpga -Xshardware -fsycl-link=early fpga_compile.cpp -o
fpga_compile_report.a

# FPGA early image (with optimization report): explicit board
dpcpp -fintelfpga -Xshardware -fsycl-link=early -Xsboard=intel_s10sx_pac:pac_s10
fpga_compile.cpp -o fpga_compile_report.a

# FPGA hardware image: default board
dpcpp -fintelfpga -Xshardware fpga_compile.cpp -c -o fpga_compile.fpga

# FPGA hardware image: explicit board
dpcpp -fintelfpga -Xshardware -Xsboard=intel_s10sx_pac:pac_s10 fpga_compile.cpp -
o fpga_compile.fpga
```

インテル次の表は上記のコマンド例で使用するオプションを説明しています。

オプション	説明
`-fintelfpga`	FPGA 向けに AOT(オフライン)コンパイルします。
`-DFPGA_EMULATOR`	プリプロセッサー定義を追加します。
`-Xshardware`	FPGA ハードウェアをターゲットにすることをコンパイラーに指示します。このオプションが省略されると、コンパイラーはデフォルトの FPGA エミュレーターをターゲットにします。 **注:** プリフィクス -X を使用すると、FPGA バックエンドに引数が渡されます。
`-fsycl-link=early`	FPGA 初期イメージ(および関連する最適化レポート)を作成して停止するようコンパイラーに指示します。
`-Xsboard=<bsp:variant>`	[オプションの引数] FPGA ボードバリアントと BSP を指定します。省略されると、コンパイラーは `intel_a10gx_pac` BSP から、デフォルトの `pac_a10` FPGA ボードバリアントを選択します。詳細については、「FPGA BSP とボード」を参照してください。

警告: `dpcpp` コンパイルコマンドの出力は、同じ出力名を持つ以前のコンパイル出力を上書きします。そのため、一意の出力名(`-o` で指定)を使用することを推奨します。失ったハードウェア・イメージの再生成には数時間を要することがあるため、FPGA コンパイルでは注意する必要があります。

上記のコンパイルオプションに加えて、dpcpp コマンドの出力詳細レベル、コンパイルで使用するスレッド数をコントロールするオプションも用意されています。次のセクションでは、これらのオプションについて簡単に説明します。

コンパイラーでサポートされるそのほかの SYCL* FPGA オプション

インテル® oneAPI DPC++/C++ コンパイラーには、カーネルのコンパイルプロセスをカスタマイズするオプションが用意されています。次の表にコンパイラーがサポートするそのほかのオプションをまとめています。

その他の FPGA オプション

オプション名	説明
-fsycl-help=fpga	dpcpp コマンドの FPGA 固有のオプションを出力します。
-fsycl-link=early -fsycl-link=image	• -fsycl-link=early と -fsycl-link は同じ意味です。どちらも、FPGA 初期イメージ(および関連する最適化レポート)を作成して停止するようコンパイラーに指示します。 • -fsycl-link=image は、デバイスリンクのコンパイル手順で使用され、FPGA ハードウェア・イメージを生成するようにコンパイラーに指示します。詳細については、「FPGA 高速再コンパイル」をご覧ください。
-reuse-exe=<exe_file> [Linux* のみ]	既存の実行形式ファイルから FPGA ハードウェア・イメージを抽出して、新しい実行形式ファイルにパッケージするようにコンパイラーに指示します。これにより、デバイスコンパイル時間を節約します。このオプションは、ハードウェア向けにコンパイルする際に役立ちます。詳細については、「FPGA 高速再コンパイル」をご覧ください。
-Xsv	FPGA バックエンドは、コンパイルの進行状況を示すレポートを生成します。
-Xsemulator	エミュレーション・デバイス・イメージを生成します。これは、デフォルトの動作です。
-Xssimulation	シミュレーター・デバイス・イメージを生成します。
-Xsghdl[=<depth>]	シミュレーション手順で信号を Siemens EDA(以前の Mentor Graphics) Questa* 波形ファイルに記録します。オプションの <depth> 属性を使用して、ログに記録される階層レベルを指定できます。<depth> を省略するとすべての信号がログに記録されます。

オプション名	説明
-Xsparallel=<num_threads>	FPGA ビットストリーム・コンパイルで使用される並列レベルを設定します。<num_threads> 値には使用する並列スレッドの数を指定します。推奨される最大値は利用可能なコア数です。このオプションは任意です。デフォルトでは、インテル® Quartus® Prime 開発ソフトウェアは利用可能なすべてのコアを使用して並列コンパイルを行います。
-Xsseed=<value>	FPGA ビットストリームを生成する際にインテル® Quartus® Prime 開発ソフトウェアが使用するシードを設定します。値は符号なし整数で、デフォルト値は 1 です。
-Xsfast-compile	最短時間で FPGA ビットストリームのコンパイルを実行します。このオプションを指定すると、コンパイル時間が短縮されますが、コンパイルされた FPGA ハードウェア・イメージのパフォーマンスは低下します。このオプションは、開発時間を短縮したい場合にのみ使用してください。製品レベルの品質は期待できません。

FPGA の最適化オプションに関する詳細は、『インテル® oneAPI DPC++ FPGA 最適化ガイド』の「最適化オプション」(英語)を参照してください。

4.8.4 設計のエミュレートとデバッグ

OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォーム(エミュレーターまたは FPGA エミュレーターとも呼ばれます)は、カーネルの機能を評価します。エミュレーターは、64 ビット Windows* および Linux* オペレーティング・システムをサポートします。Linux* システムでは、GNU C ライブラリー(glibc)バージョン 2.15 以降が必要です。

注:

- エミュレートされた設計の実行時間から、FPGA での実行時間を予測することはできません。また、エミュレートされた設計を実行しても、機能的に同等な C/C++ 実装を x86-64 ホスト上でネイティブ実行する代わりにはなりません。
- エミュレーションは、ARM® プロセッサーのクロスコンパイルはサポートしていません。ARM SoC デバイスをターゲットとする設計でエミュレーションを実行するには、非 SoC ボード (intel_a10gx_pac または intel_s10sx_pac など) でエミュレーションしてください。エミュレーションの結果を確認できたら、SoC ボード上の設計をターゲットにして、以降の最適化ステップに進むことができます。
- インテル® ディストリビューションの GDB* を使用したデバッグについては以下を参照してください。

  - Linux* ホストでのインテル® ディストリビューションの GDB* によるデバッグ (英語)

- Linux* ホストでのインテル® ディストリビューションの GDB* の導入ガイド (英語)
- Windows* ホストでのインテル® ディストリビューションの GDB* の導入ガイド (英語)

エミュレーターのインストール

OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォームは、インテル® oneAPI ベース・ツールキット (英語) の一部としてインストールされます。ベース・ツールキットのインストール方法については、「インテル® oneAPI ツールキットおよびコンポーネントのインストール・ガイド」(英語) をご覧ください。

追加情報については次のトピックを参照してください。

- エミュレーターの環境変数
- パイプの深度をエミュレート
- I/O パイプの読み取りまたは書き込みを行うパイプを使用してアプリケーションをエミュレート
- 設計のコンパイルとエミュレート
- エミュレーターの制限
- ハードウェアとエミュレーターの結果の不一致
- エミュレーターの既知の問題

エミュレーターの環境変数

次の表に、エミュレーターの動作制御に利用できる環境変数を示します。

環境変数名	説明
CL_CONFIG_CPU_EMULATE_DEVICES	エミュレーター・プラットフォームによって提供される同一エミュレーター・デバイスの数を制御します。省略されると、単一エミュレーター・デバイスを使用できます。この変数は、複数のデバイスをエミュレートする場合にのみ設定します。
OCL_TBB_NUM_WORKERS	エミュレーターが使用できる最大スレッド数を指定します。デフォルトは 32 で、最大値は 255 です。各スレッドは単一のカーネルを実行できます。アプリケーションが複数のカーネルを同時に実行する場合、OCL_TBB_NUM_WORKERS 環境変数を使用するカーネル数以上に設定する必要があります。
CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE	使用可能なローカル・メモリー・サイズをユニット単

環境変数名	説明
	位で指定します。例： 8MB、256KB、または 1024B。
CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE	使用可能なプライベート・メモリー・サイズをユニット単位で指定します。例： 8MB、256KB、または 1024B。 **注：**Windows* では、FPGA エミュレーターがメモリー不足で警告なく失敗することがあります。このエラーをキャッチするため、try-catch 構文を使用してカーネルを記述します。
CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE	カーネルをエミュレーション用にコンパイルする場合、パイプの深度はカーネルをハードウェア向けにコンパイルする際のパイプ深度とは異なります。この動作は、環境変数 CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE を設定して変更できます。詳細については「パイプの深度をエミュレート」を参照してください。

パイプの深度をエミュレート

エミュレーション用にカーネルをコンパイルする場合のデフォルトのパイプの深さは、カーネルをハードウェア向けにコンパイルする際のデフォルトの深度とは異なります。環境変数 CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE を設定して、エミュレーション用にカーネルをコンパイルする際の動作を変更できます。

重要：パイプでは、ホストプログラムを実行する前にCL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE 設定変数を設定する必要があります。

CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE 環境変数には次の値を設定できます。

環境変数の値	説明
ignoredepth	すべてのパイプにカーネル・エミュレーションの実行時間を最速にするパイプの深度が選択されます。明示的に設定されたパイプ深度属性は無視されます。CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE 環境変数が設定されていない場合、これがデフォルトになります。

`default`	明示的な深度属性を持つパイプには深度が指定されます。深度が指定されていないパイプには、カーネル・エミュレーションの実行時間を最速にするデフォルトのパイプ深度が選択されます。
`strict`	エミュレーションのすべてのパイプ深度に、`FPGA` コンパイルで指定される深度と同じ深さが設定されます。

I/O パイプの読み取りまたは書き込みを行うパイプを使用してアプリケーションをエミュレート

OpenCL* ソフトウェア向けインテル® FPGA エミュレーション・プラットフォームは、カーネル間のパイプをエミュレートします。ただし、ターゲットボード上のハードウェア I/O パイプとの直接対話はサポートされていません。次の手順により、I/O パイプの動作をエミュレートできます。

入力 I/O パイプ向け

1. パイプに転送される入力データをパイプの id 特殊化と一致する名前のファイルに保存します。次を考えてみます。

```
// パイプタイプの特殊化
struct read_io_pipe {
    static constexpr unsigned id = 0;
};
using read_iopipe = sycl::ext::intel::kernel_readable_io_pipe<read_io_pipe,
unsigned, 4>;
```

   2. 名前 0 のファイルを作成します。
   3. ファイル 0 にテスト用の入力データを保存します。

出力 I/O パイプ向け

出力データは、出力パイプの id 特殊化と一致する名前のファイルに自動的に書き込まれます。

設計のコンパイルとエミュレート

FPGA のカーネル設計をコンパイルおよびエミュレートするには、次の手順を実行します。

1. プログラムのホスト部分を変更して、`ext::intel::fpga_emulator_selector` デバイスセレクターを宣言します。FPGA デバイスカーネルをエンキューするためデバイスキューをインスタンス化する際に、この `device_selector` を使用します。
   2. dpcpp コマンドに `-fintelfpga` オプションを追加して設計をコンパイルし、実行可能ファイルを生成します。
   3. 生成された実行可能ファイルを実行します。

- Windows* 向け:

  1. 次のコマンドを実行し、エミュレートされるデバイス数を定義します。

     ```
     set CL_CONFIG_CPU_EMULATE_DEVICES=<デバイス数>
     ```

  2. 実行可能ファイルを実行します。
  3. 次のコマンドを実行して設定を解除すます。

     ```
     set CL_CONFIG_CPU_EMULATE_DEVICES=
     ```

- Linux* では次のコマンドを実行します:

  ```
  env CL_CONFIG_CPU_EMULATE_DEVICES=<デバイス数> <実行可能ファイル>
  ```

  このコマンドは、エミュレーターが提供する必要がある同一エミュレーターのかすを指定します。

ヒント: 単一のエミュレーター・デバイスを使用する場合、この設定は必要ありません。`CL_CONFIG_CPU_EMULATE_DEVICES` 環境変数を設定します。

注:

- OpenCL* 向けインテル® FPGA エミュレーター・プラットフォームでは、物理ボードへアクセスすることはできません。エミュレートされるデバイスのみを利用できます。
- エミュレーターは、インテル® oneAPI DPC++/C++ コンパイラーに含まれる GCC 7.4.0 でビルドされます。そのため、エミュレートされる FPGA デバイスの実行ファイルを実行するには、`libstdc++.so` が GCC 7.4.0 のものである必要があります。つまり、`LD_LIBRARY_PATH` 環境変数は、正しいバージョンの `libstdc++.so` の場所を示す必要があります。

正しいバージョンの `libstdc++.so` が見つからない場合、`clGetPlatformIDs` 関数の呼び出しは FPGA エミュレーター・プラットフォームのロードに失敗し、`CL_PLATFORM_NOT_FOUND_KHR` (エラーコード -1001) を返します。検出される `libstdc++.so` のバージョンによっては、`clGetPlatformIDs` 関数の呼び出しが成功することもありますが、後に `clCreateContext` 関数を呼び出すと `CL_DEVICE_NOT_AVAILABLE`(エラーコード-2)で失敗する可能性があります。

`LD_LIBRARY_PATH` 環境変数が互換性のある `libstdc++.so` を指していない場合、次のコマンドでホストプログラムを起動します。

```
env LD_LIBRARY_PATH=<互換性のある libstdc++.so へのパス>:$LD_LIBRARY_PATH <実行可能ファイル> [引数]
```

- カーネルコードのデバッグを有効にするため、デフォルトでは FPGA エミュレーターの最適化は無効になっています。そのため、カーネルコードのエミュレーションで実行スピードが最適ではなくなる可能性があります。`dpcpp` コンパイルコマンドに `-g0` オプションを指定して、デバッグを無効にし、最適化を有効にできます。これにより、エミュレーターの実行が高速になります。
- FPGA エミュレーター・デバイスをターゲットにする場合、`-O2` オプションを指定して最適化を有効にし、エミュレーターを高速化します。最適化を無効にするには (デバッグ用途に)、`-O0` を指定します。

エミュレーターの制限

OpenCL* ソフトウェア向けインテル® FPGA エミュレーター・プラットフォームには、次の制限があります。

- **同時実行**

カーネルの同時実行をモデル化するには制限があります。実行中にエミュレーターが相互作用するワーク項目を並列に実行することは保証されません。したがって、同期のバリアなしでグローバルメモリーにアクセスする異なるカーネルなど、一部の同時実行動作では、実行するごとにエミュレーション結果が異なる可能性があります。

- **同じアドレス空間の実行**

  エミュレーターは、ホストのランタイムとカーネルを同じアドレス空間で実行します。特定のポインターや配列を使用すると、カーネルプログラムが実行に失敗することも成功することもあります。一例として、外部割り当てのメモリー・インデックスの作成やランダムポインターへの書き込みがあります。Valgrind などのメモリーリーク検出ツールを使用して、プログラムの動作を解析できます。カーネルが境界外への書き込みが原因でホストが致命的なエラーを生じる可能性がありますが、その逆も考えられます。

- **条件パイプ操作**

  パイプ動作のエミュレーションには限界があり、特にカーネルの条件付きパイプ操作では、ループの繰り返しごとにパイプ操作を呼び出すとは限りません。この場合、エミュレーターはハードウェアとは異なる順番でパイプ操作を実行する可能性があります。

- **GCC のバージョン**

  エミュレーターのホストプログラムは、GCC 7.4.0 以降の `libstdc++.so` を搭載する Linux* システム上で実行する必要があります。GCC 7.4.0 以降をシステムにインストールするか、`LD_LIBRARY_PATH` 環境変数に互換性のある `libstdc++.so` を識別できるようにパスを設定します。

ハードウェアとエミュレーターの結果の不一致

カーネルをエミュレートすると、カーネルがハードウェア向けにコンパイルされたカーネルとは異なる結果を生成することがあります。ハードウェア向けにコンパイルする前に、カーネルをシミュレーションすることでデバッグを行うことができます。

警告: このような結果の不一致は、多くの場合 OpenCL* 向けインテル® FPGA エミュレーション・プラットフォームがハードウェアの計算を正確にモデル化できない、またはプログラムが未定義の動作に依存する場合に発生します。

エミュレーターとハードウェアの結果が異なる原因は、次のものがあります。

- カーネルコードに `ivdep` 属性が使用されています。`ivdep` 属性によって依存関係が無視されると、エミュレーターはカーネルをモデル化できません。ハードウェア向けの完全なコンパイルでは、誤った結果として現れます。
- カーネルコードが初期化されていないデータに依存しています。初期されていないデータには、初期されていない変数、初期化されていないまたは部分的に初期化されたグローバルバッファー、ローカル配列、プライベート配列などがあります。
- カーネルコードの動作は、浮動小数点演算の結果によって異なります。エミュレーターは CPU の浮動小数点演算ハードウェアを使用しますが、ハードウェア実行は FPGA コアとして実装された浮動小数点コアを使用します。

注: SYCL* 標準では、浮動小数点演算の最下位ビットがプラットフォーム間で異なっていても、それぞれのプラットフォームでは正しいと見なされるように定義されています。

- カーネルコードの動作は、それぞれのカーネルのパイプアクセスの順序によって異なります。チャネル動作のエミュレーションには制限があり、カーネルがループ反復ごとにチャンネル操作を呼び出さない条件付き操作がそれに該当します。この場合、エミュレーターはハードウェアとは異なる順番でチャンネル操作を行う可能性があります。
- カーネルまたはホストコードが範囲外のグローバル・メモリー・バッファーにアクセスしています。

注:

- 初期化されていないメモリーの読み書きの動作は、プラットフォームに依存します。カーネル内のすべてのアドレス空間を使用する場合、グローバル・メモリー・バッファーのサイズを確認してください。
- エミュレートされたカーネルでは、Valgrind などのメモリーリーク検出ツールを使用してメモリーに関連する問題を解析できます。ツールが警告を示さない場合でも問題がないわけではありません。それは、ツールが問題を検出できなかっただけであり、カーネルまたはホストコードを手動で検証することを推奨します。

カーネルコードが範囲外のローカル変数にアクセスしています。このような例として、範囲外のローカル配列にアクセスしたり、スコープから出た場合の変数アクセスが上げられます。

注: 通常、境界外の変数へのアクセスは、ソフトウェア・スタック上でアクセスされる変数に隣接する関連のない変数に影響を与えるため、ソフトウェアにおける用語ではこれはスタック破壊の問題と呼ばれます。エミュレート・カーネルは通常 CPU 関数として実装されるため、スタックが破損する可能性があります。ハードウェアをターゲットとするケースではスタックは存在しません。そのため、スタック破損の問題は、異なる形で明らかになります。スタック破損が疑われる場合、Valgrind のようなメモリー・リーク・アナライザーを使用しますが、スタック関連の問題は特定することが困難です。インテルでは、スタック関連の問題をデバッグする際に、カーネルコードを手動で検証することを推奨しています。

カーネルコードが、シフトされるデータ型よりも大きなシフトを行っています。例えば、64 ビット整数を 65 ビット・シフトしています。SYCL* 仕様のバージョン 1.0 では、このようなシフトの動作は未定義とされています。

エミュレーション向けにカーネルをコンパイルする場合、デフォルトのパイプ深度はハードウェア向けにコンパイルされる際のデフォルトのパイプライン深度とは異なります。このパイプラインの深度の違いにより、カーネル・エミュレーションでは問題なく動作していても、ハードウェアの実行でハングする可能性があります。チャンネル深度の違いを修正する方法は、「パイプの深度をエミュレート」を参照してください。

エミュレーターとハードウェアでは、`cout` ストリーム関数で出力される順番が異なる場合があります。これは、ハードウェアでは `cout` ストリームのデータがグローバル・メモリー・バッファーに格納され、カーネルの実行が完了した、またはバッファーが一杯になったときにのみバッファーからスラッシュされるのに対し、エミュレーターでは `cout` ストリーム関数は x86 の `stdout` を使用するためです。

タイプのアップキャストにより、アライメントされていないロード/ストアを実行すると、ハードウェアとエミュレーターでは異なる結果が生じることがあります。このタイプのロード/ストアは、C99 仕様では未定義です。例えば、次のような操作により、予期しない結果が生成される可能性があります。

```
int tmp = *((int *) (my_ptr + 5));
```

エミュレーターの既知の問題

既知の問題にはエミュレーターの動作に影響するものがあります。これらの問題を確認して、エミュレーターで発生する可能性がある問題を回避してください。

コンパイラーの診断

一部のコンパイラーの診断は、エミュレーターにはまだ実装されていません。

カーネル起動時の CL_OUT_OF_RESOURCES エラー

エミュレーターがデフォルトでサポートするよりも多くのプライベート・メモリーやローカルメモリーをカーネルが使用すると、このエラーが発生する可能性があります。

エミュレーター環境変数の説明に従って、`CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE` または `CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE` 環境変数に大きな値を設定してください。

注: Windows* では、FPGA エミュレーターがメモリー不足により警告なく失敗することがあります。このエラーをキャッチするため、`try-catch` 構文を使用してカーネルコードを記述してください。

FPGA ランタイムとエミュレーション・バイナリーの互換性

oneAPI FPGA ランタイムは、以前のバージョンの oneAPI で構築されたエミュレーション・バイナリーをサポートしていません。現在の oneAPI のリリースでエミュレーション・バイナリーを再コンパイルする必要があります。

4.8.5 シミュレーションでカーネルを評価

Questa* - インテル® FPGA エディション・シミュレーター・ソフトウェアは、カーネルの機能を評価します。

シミュレーター手順 (ベータ) では、ホストで実行されるシミュレーション・バイナリーを生成します。コードのハードウェア部分は RTL シミュレーターで評価され、ホスト部分はプロセッサーでネイティブ実行されます。この機能を使用すると、カーネルをハードウェア向けにコンパイルして FPGA デバイスでその都度実行しなくても、カーネルの機能をシミュレートして設計を繰り返すことができます。

注: シミュレーターのパフォーマンスは、ハードウェアと比較すると非常に低速であるため、小規模なデータセットを使用してテストすることを推奨します。

カーネルの動的なパフォーマンスおよび、エミュレーションやレポートツールが提供する情報よりも、カーネルの機能に関する正確な情報を必要とする場合、シミュレーターを使用します。

シミュレーターはサイクル精度とビット精度が高く、生成されたハードウェアと同じネットリストを備えており、デバッグ向けに完全な波形を提供できます。Siemens * EDA (以前の Mentor Graphics) Questa* ソフトウェアで波形を表示します。

シミュレーションの要件

FPGA シミュレーション手順を実行するには、事前に次のソフトウェアをダウンロードする必要があります。

- インテル® Quartus® ソフトウェア 21.4 パッケージを使用したインテル®カスタム・プラットフォーム用の FPGA アドオン。このパッケージは、インテル® oneAPI ベース・ツールキット (英語) のダウンロード・ページから入手できます。
- 互換性のあるシミュレーション・ソフトウェア。Questa*-インテル® FPGA エディションおよび Questa*-インテル® FPGA スターター・エディションが含まれます。これらは、FPGA のダウンロード・センター (英語) から入手できます。

注:

- Questa*-インテル® FPGA エディションにはライセンスが必要です。Questa*-インテル® FPGA スターター・エディションは無料ですが、無償ライセンスが必要です。詳細については、「インテル® FPGA ソフトウェアのインストールとライセンス」(英語)を参照してください。
- 独自ライセンスの Siemens* EDA ModelSim* SE または Siemens* EDA Questa Advanced Simulator(英語)ソフトウェアを使用することもできます。インテル® Quartus® Prime 開発ソフトウェア・プロ・エディションがサポートするすべての ModelSim* および Questa* ソフトウェアバージョンについては、インテル® Quartus® Prime 開発ソフトウェア・プロ・エディションの「EDA インターフェイス情報」(英語)セクションを参照してください。
- Linux* システムで Questa*-インテル® FPGA エディションおよび Questa*-インテル® FPGA スターター・エディション・ソフトウェアを使用するには、RedHat* 開発ツール(英語)をインストールする必要があります。

Questa*-インテル® FPGA エディション・ソフトウェアのインストール

Questa*-インテル® FPGA エディション・ソフトウェアをインストールするには、次の手順を実行します。

1. FPGA ダウンロード・センター(英語)から、Questa*-インテル® FPGA エディション・ソフトウェアをダウンロードします。
2. 左にある [filter] ペインから、次の手順を実行して検索結果を絞り込みます。

   A) インテル® Quartus® Prime Design Software のオプションを選択します。これにより 3 つの インテル® Quartus® Prime エディション (Pro、Standard、Lite) が表示されます。
   B) 対象のインテル® Quartus® Prime ソフトウェア・エディションを選択します。サポートされるソフトウェアのバージョンリストが表示されます。
   C) 対象のインテル® Quartus® Prime ソフトウェアのバージョンを選択します。
   D) オペレーティング・システム (Linux* または Windows*) を選択します。

3. 絞り込まれたページ一覧から、ダウンロードするページをクリックすると、製品のダウンロード・ページが表示されます。
4. ダウンロード・セクションで個々のファイルタブをクリックします。
5. 各ソフトウェアの下にあるダウンロード <ファイル名> ボタンをクリックして、使用するシステム向けの Questa*- インテル® FPGA エディションをダウンロードします。

注:Questa*-インテル® FPGA エディション(スターター・エディションを含む)と Questa*-インテル® FPGA エディション(スターター・エディションを含む)パート 2 の両方のパッケージをダウンロードする必要があります。

6. **[Accept]** ボタンをクリックして、シミュレーション・ソフトウェアのライセンスに同意します。ファイルのダウンロードは自動的に開始されます。
7. シミュレーション・ソフトウェアのライセンスを取得して設定します。

   システム要件、前提条件、インテル® Quartus® Prime 開発ソフトウェアのインストールに関する情報については、「インテル® FPGA ソフトウェアのインストールとライセンス」(英語)を参照してください。

8. Questa*-インテル® FPGA エディションのインストーラーを実行します。インストールを開始すると、Questa*-インテル® FPGA エディションおよび Questa*-インテル® FPGA スターター・エディション(無料) のどちらをインストールするか選択できます。
9. ライセンスを取得したシミュレーション・ソフトウェアを選択します。インストーラーは、Questa* シミュレーション・ソフトウェアのインストール先を問い合わせてきます。
10. Questa* シミュレーション・ソフトウェアをインストールするディレクトリーを指定します。必須ではありませんが、インテル® Quartus® ソフトウェア 21.2 パッケージを使用して、インテル® カスタム・プラットフォーム用の FPGA アドオンと一緒に Questa* ソフトウェアをインストールすることを推奨します。次の例を考えてみます。

    Linux* システムで、FPGA アドオンパッケージのインテル® Quartus® Prime 開発ソフトウェアが次のディレクトリーにインストールされていると仮定します。

    ```
    /opt/intel/oneapi/intelfpgadpcpp/2021.4.0/QuartusPrimePro/21.2/quartus
    ```

    次に、Quartus ディレクトリーの 1 つ上の階層を選択して、Questaソフトウェアをインストールします。

    ```
    /opt/intel/oneapi/intelfpgadpcpp/2021.4.0/QuartusPrimePro/21.2/
    ```

注: oneAPI 環境で `QUARTUS_ROOTDIR_OVERRIDE` 環境変数を調べると、インテル® Quartus® Prime 開発ソフトウェアのインストール場所を確認できます。

シミュレーション環境の設定

インテル® Quartus® Prime 開発ソフトウェアおよび Questa* シミュレーション・ソフトウェアのバイナリーを含むディレクトリーを `PATH` 環境変数に追加する必要があります。

注: このトピックで紹介するコマンドは、シミュレーションの前提条件に示すよう FPGA アドオンパッケージとともに Questa* シミュレーション・ソフトウェアがインストールされている必要があります。他の場所に Questa* シミュレーション・ソフトウェアをインストールした場合、`PATH` 環境変数 を選択してください。

インテル® Quartus® Prime 開発ソフトウェア向け(シミュレーション手順のみ)

FPGA シミュレーション手順の場合にのみ、インテル® Quartus® Prime 開発ソフトウェアのバイナリーを明示的に追加する必要があります。ディレクトリーを、以下のコマンドで PATH 環境変数に追加します。

Linux*:

```
$ export PATH=$PATH:/quartus/bin
```

Windows*:

```
$ set "PATH=%PATH%;<quartus_installdir>\quartus\bin64"
```

さらに、ロードする ICD (インストール可能なクライアント・ドライバー) を指定するため、OCL_ICD_FILENAMES 環境変数を設定する必要があります。

```
$ set "OCL_ICD_FILENAMES=%OCL_ICD_FILENAMES%;alteracl_icd.dll"
```

Questa*-インテル® FPGA スターター・エディション・ソフトウェア

無償のインテル® FPGA スターター・エディション・ソフトウェアでは、次のコマンドを実行します。

Linux*:

```
$ export PATH=$PATH:/questa_fse/bin
```

Windows*:

```
$ set "PATH=%PATH%;<quartus_installdir>\quartus\bin64"
```

Questa*-インテル® FPGA エディション・ソフトウェア

インテル® FPGA エディション・ソフトウェアでは、次のコマンドを実行します。

Linux*:

```
$ export PATH=$PATH:/questa_fe/bin
```

Windows*:

```
$ set "PATH=%PATH%;<quartus_installdir>\questa_fe\win64"
```

これでシミュレーション用のソフトウェアをコンパイルできます。

シミュレーション向けにカーネルをコンパイル

シミュレーションを行う前に、インテル® Quartus Prime 開発ソフトウェア・プロ・エディションがインストールされていることを確認してください。これには、インテル® Quartus Prime 開発ソフトウェア・プロ・エディションのパッケージで、カスタム・プラットフォーム向けの FPGA アドオンの適切なバージョンをインストールします。

シミュレーション向けにカーネルをコンパイルするには、`dpcpp` コマンドに `-Xssimulation` オプションを追加します。

```
$ dpcpp -fintelfpga -Xssimulation fpga_compile.cpp
```

また、シミュレーション中に波形の収集を有効にするには、`dpcpp` コマンドに `-Xsghdl [=<depth>]` オプションを追加します。オプションの `<depth>` 属性にはログに記録される階層のレベル数を指定します。省略されるとすべての波形信号がログに記録されます。

Windows* システムでシミュレーションを実行する場合、Microsoft* リンカーと追加のライブラリーが必要です。以下の設定を確認します。

- 環境変数 `PATH` には、Microsoft* Visual Studio* の `LINK.EXE` コマンドへのパスを含める必要があります。
- 環境変数 `LIB` には、Microsoft* コンパイル・ライブラリーへのパスを含めます。コンパイル・ライブラリーは、Microsoft* Visual Studio* で利用できます。

カーネルをシミュレート

シミュレーション手順 (ベータ版) でシミュレーション中に収集された波形を表示するには、Siemens* EDA Questa Simulator または ModelSim SE のどちらかがインストールされ、利用可能である必要があります。

シミュレーターを介して DPC++ ライブラリーを実行するには、以下を行います。

1. `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA` 環境変数を設定して、シミュレーションデバイスを有効にします。

   Linux*:

   ```
   $ export CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1
   ```

   Windows*:

   ```
   $ set CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1
   ```

注: 環境変数 `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA` が設定されている場合、シミュレーションデバイスのみが使用可能であり、物理ボードへのアクセスは無効になります。環境変数の設定を解除するには、次のコマンドを実行します。

Linux*

```
$ unset CL_CONTEXT_MPSIM_DEVICE_INTELFPGA
```

Windows*

```
set CL_CONTEXT_MPSIM_DEVICE_INTELFPGA
```

ホストプログラムがシミュレーター・デバイスを検出できない場合、以下を設定する必要があります。

```
$ export CL_CONTEXT_COMPILER_MODE_INTELFPGA=3
```

2. ホストプログラムを実行します。Linux* システムでが、GDB* または Eclipse* を使用してホストをデバッグできます。必要に応じて、カーネルコードのシミュレーション波形を調査して作成されたハードウェアの機能を確認できます。

   `-Xsghdl` オプションを使用してコンパイルしたプログラムを実行すると、波形ファイル `vsim.wlf` が生成されます。このファイルは、ホストコードの実行時に Questa*-インテル® FPGA エディション・ソフトウェアで表示できます。`vsim.wlf` ファイルは、ホストプログラムを実行したディレクトリーに書き込まれます。

シミュレーターの問題のトラブルシューティング

このセクションを確認して、シミュレーションを行う際に発生する可能性があるシミュレーターの問題に対するトラブルシューティングを行います。

socket=-11 エラーが transcript.log に記録される

次のエラーメッセージが表示されると、Questa*-インテル® FPGA エディションや ModelSim* SE など複数のシミュレーターのリソースが競合している可能性があります。

```
Message: "src/hls_cosim_ipc_socket.cpp:202: void IPCSocketMaster::connect():
Assertion `sockfd != -1 && "IPCSocketMaster::connect() call to accept() failed"'
failed."
```

シミュレーターのリソースが競合する例として、ModelSim* SE を使用してデバイスをコンパイルし、Questa*-インテル® FPGA スターター・エディションでホストプログラムを実行する場合が上げられます。

Questa*-インテル® FPGA スターター・エディション・ソフトウェアの互換性

Questa*-インテル® FPGA スターター・エディション・ソフトウェアには、設計の大きさに制限があり、大規模な設計をシミュレートすることができません。Questa*-インテル® FPGA スターター・エディション・ソフトウェアを使用してシミュレーターを起動すると、次のエラーメッセージが表示されることがあります。

```
Error: The simulator's process ended unexpectedly.
```

Questa*-インテル® FPGA エディション、または ModelSim* SE ソフトウェアを使用して設計をシミュレートします。

4.8.6 FPGA デバイスセレクター

ホストコードで正しい SYCL* デバイスセレクターを使用して、FPGA エミュレーターまたは FPGA ハードウェアのターゲットを選択します。次のコード例は、セレクターを使用してコンパイル時にターゲットデバイスを指定する方法を示しています。

```
// FPGA デバイスセレクターは、パイプや fpga_reg などすべての
// FPGA 拡張機能とともにユーティリティー・ヘッダーで定義されます
#include <sycl/ext/fpga_extensions.hpp>

int main() {
  // 次のいずれかを選択:
  //  - FPGA エミュレーター・デバイス (CPU で FPGA をエミュレーション)
  //  - FPGA デバイス (実際の FPGA)
#if defined(FPGA_EMULATOR)
  ext::intel::fpga_emulator_selector device_selector;
#else
  ext::intel::fpga_selector device_selector;
#endif
  queue q(device_selector);
  ...
}
```

注:

- FPGA エミュレーターと FPGA は異なるターゲットデバイスです。`define` プリプロセッサーを使用して、エミュレーターと FPGA セレクターを選択することを推奨します。これにより、ソースを変更することなくコマンドラインで `-D` オプションを使用してターゲットを切り替えることができます。例えば上記のコード例では、`dpcpp` コマンドに `-DFPGA_EMULATOR` オプションを渡すことで FPGA エミュレーター向けにコンパイルできます。
- FPGA は事前(AOT)コンパイルのみをサポートしているため、FPGA をターゲットにする場合、動的セレクター(`default_selector` など)は有用ではありません。

警告:FPGA エミュレーターや FPGA ハードウェアをターゲットにする場合、適切なコンパイルオプションを指定し、ホストコードで正しいデバイスセレクターを使用する必要があります。そうでない場合、ランタイムエラーが発生する可能性があります。FPGA 向けの SYCL* コードのコンパイルに関する詳細は、「インテル® oneAPI サンプルブラウザー」(英語)の「fpga_compile」(英語)チュートリアルを参照してください。

4.8.7 FPGA 高速再コンパイル

インテル® oneAPI DPC++/C++ コンパイラーは、FPGA ハードウェアでは事前(AOT)コンパイルのみをサポートします。FPGA デバイスイメージは、コンパイル時に生成されます。FPGA デバイスイメージの生成には、コンパイルが完了するまで数時間かかる場合があります。ホストコードにのみ影響する変更を行った場合、FPGA デバイスイメージを再利用して、時間のかかるデバイスコンパイルを回避しホストコードのみを再コンパイルします。

インテル® oneAPI DPC++/C++ コンパイラーは、デバイスコードとホストコードのコンパイルを分離する次の機能をサポートします。

- `-reuse-exe=<exe_name>` オプション(Linux* のみ)を指定して、既存の FPGA デバイスイメージを再利用するようコンパイラーに指示します。
- ホストとデバイスのコードを別々のファイルに分離します。コードの変更がホストファイルにのみ影響する場合、FPGA デバイスイメージは生成されません。
- コンパイラー・オプション `-fsycl-device-code-split` を使用して、デバイスコードを分離します。

次のセクションでは、この 3 つの機能について詳しく説明します。

`-reuse-exe` オプションを使用

前回のコンパイル以降、デバイスに影響するデバイスコードとオプションが変更されていない場合、`-reuse-exe=<exe_name>` オプションを指定して FPGA ハードウェア・イメージを既存の実行形式ファイルから抽出して、新しい実行形式ファイルにパッケージするようにコンパイラーに指示します。これにより、デバイスコンパイル時間を節約します。

使用例:

```
# 最初のコンパイル
dpcpp -fintelfpga -Xshardware <files.cpp> -o out.fpga
```

最初のコンパイルで FPGA デバイスイメージを生成しますが、これには数時間かかります。ここで、ホストコードに変更を加えます。

```
# その後のコンパイル
dpcpp <files.cpp> -o out.fpga -reuse-exe=out.fpga -Xshardware -fintelfpga
```

コマンドにより次のいずれかのアクションが実行されます。

- `out.fpga` が存在しない場合、`-reuse-exe` オプションは無視され、FPGA デバイスイメージが再生成されます。これは、プロジェクトを初めてコンパイルする場合、常にあてはまります。
- `out.fpga` が存在すると、コンパイラーは FPGA デバイスコードに影響する変更が前回のコンパイル以降に行われていないか確認します。デバイスコードに影響する変更が行われていない場合、コンパイラーは既存の FPGA デバイスイメージを再利用してホストコードのみを再コンパイルします。再コンパイルには数分かかります。
- `out.fpga` が存在しないか、FPGA デバイスコードが最後のコンパイルと同じ結果を生成することをコンパイラーが証明できない場合、警告が出力され FPGA デバイスコードは完全に再コンパイルされます。コンパイラーのチェックはかなり保守的であるため、`-reuse-exe` オプションを使用すると、必要のない再コンパイルが行われることがあります。

デバイスリンクを使用

プログラムが `main.cpp` と `kernel.cpp` で構成され、`kernel.cpp` ファイルのみがデバイスコードを含むと仮定します。

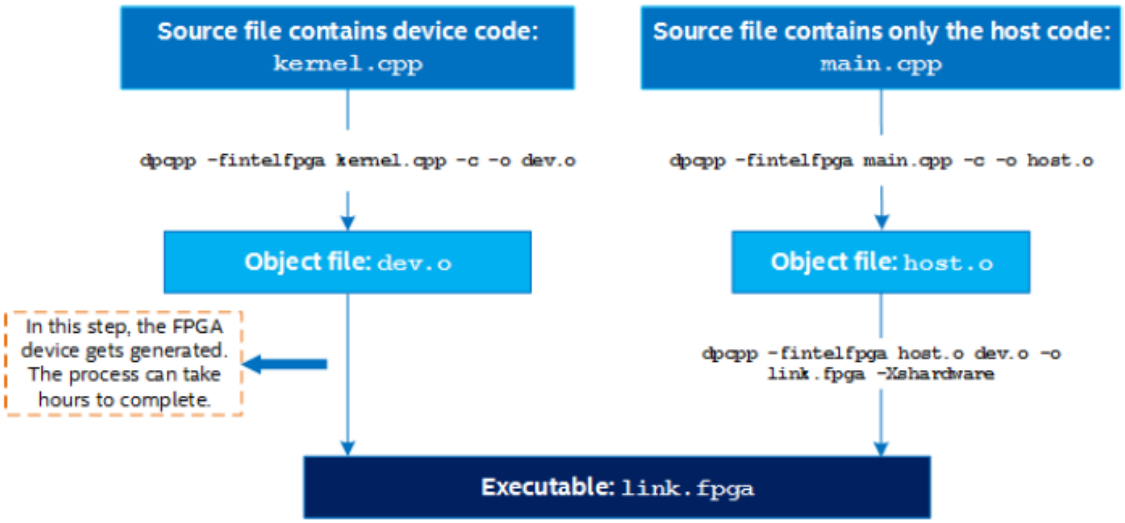
通常のコンパイル手順では、FPGA デバイスイメージの生成はリンク時に行われます。

```
# 通常のコンパイルコマンド
dpcpp -fintelfpga -Xshardware main.cpp kernel.cpp -o link.fpga
```

`main.cpp` または `kernel.cpp` のどちらかを変更すると、FPGA ハードウェア･イメージの再生成がトリガーされます。

次の図はこのコンパイル手順を示します。

コンパイル手順



ホストコードを繰り返し処理し、FPGA デバイスのコンパイル時間が長くならないようにするには、デバイスリンクを使用してデバイスとホストのコンパイルを分離することを検討してください。デバイスリンクは次の 3 つのステップからなります。

```
# デバイス･リンク･コマンド
dpcpp -fintelfpga -fsycl-link=image <入力ファイル>[オプション]
```

コンパイルは次の 3 つの手順から成ります。

1. デバイスコードをコンパイルします。

```
dpcpp -fintelfpga -Xshardware -fsycl-link=image kernel.cpp -o dev_image.a
```

入力ファイルには、デバイス上で実行されるすべてのファイルを含める必要があります。このステップには数時間かかります。

2. ホストコードをコンパイルします。

```
dpcpp -fintelfpga main.cpp -c -o host.o
```

入力ファイルには、ホストコードのみを対象とするすべてのソースファイルを含める必要があります。これには、デバイスで実行されるソースコードを含めることはできませんが、コマンドライン･オプションの解釈や結果レポートなどのセットアップ･コードとティアダウン･コードを含めることができます。このステップには数秒かかります。

3. デバイスリンクを作成します。

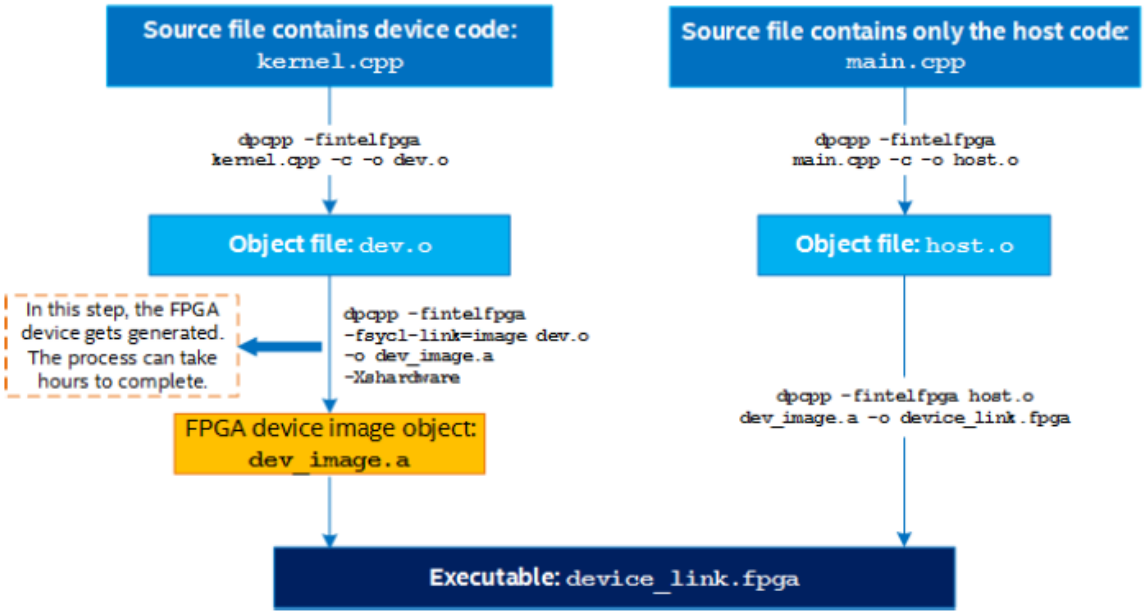
```
dpcpp -fintelfpga host.o dev_image.a -o fast_recompile.fpga
```

このステップには数秒かかります。入力には、1 つ以上のホスト・オブジェクト・ファイル(.o)と 1 つのデバイスファイル(.a)を含める必要があります。静的ライブラリー(.a ファイル)を使用する場合、必ず静的ライブラリーをリンクしてください。これ以外ではライブラリー関数は破棄されます。静的ライブラリーに関する詳細は、https://eli.thegreenplace.net/2013/07/09/library-order-in-static-linking(英語)を参照してください。

注: ホストのみのファイルを変更する場合、手順 2 と 3 を実行します。

次の図はデバイスのリンク手順を示します。

FPGA デバイスリンク



デバイスリンクの使い方については、「インテル® oneAPI サンプルブラウザー」(英語)の「fast_recompile」(英語)チュートリアルを参照してください。

-fsycl-device-code-split[=*value*] オプションを使用

`-fsycl-device-code-split[=value]` オプションを使用すると、コンパイラーは各分割パーティションをそれ自身のデバイスをターゲットにしているようにコンパイルします。このオプションは、次のモードをサポートします。

- **`Auto`**:これはデフォルトモードであり、値なしの `-fsycl-device-code-split` と同等です。コンパイラーはヒューリスティックに基づいてデバイスコードを分割する最良の方法を選択します。
- **`Off`**:すべてのカーネルに対して単一のモジュールを生成します。
- **`per_kernel`**:SYCL* カーネルごとに個別のデバイス・コード・モジュールを生成します。各デバイス・コード・モジュールには、カーネルと呼び出される関数や使用する変数など、すべての依存関係が含まれます。
- **`per_source`**:ソース(翻訳ユニット)ごとに個別のデバイス・コード・モジュールを生成します。各デバイス・コード・モジュールには、ソースごとにグループ化されたカーネルと、他の翻訳ユニットからの SYCL_EXTERNAL マクロマーク付きの関数を含む、使用されたすべての変数など、すべての依存関係が含まれます。

注意:FPGA では、各スプリットは、メモリーなどのデバイスリソースを FPGA 間で共有する必要があります。さらに、カーネルパイプは、同じスプリット内に `source` と `sink` を持つ必要があります。

このオプションの詳細については、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』の「`fsycl-device-code-split`」(英語)のトピックを参照してください。

どの機能を使用するか?

前述の 2 つの機能では、デバイスリンクよりも `-reuse-exe` オプションを使用する方が容易です。このオプションを使用すると、ホストとデバイスのコードを単一ソースとして記述することができます。これは小規模なプログラムに適しています。大規模で複雑なプロジェクトでは、デバイスリンクにはコンパイラーの動作を細かくコントロールできる利点があります。

しかし、個別のファイルをコンパイルするのと比較すると `-reuse-exe` オプションにはいくつかの欠点もあります。以下に `-reuse-exe` オプションを使用する際の注意点を示します。

- コンパイラーは、影響するソースの変更がないことを確認するため、デバイスコードを部分的に再コンパイルする時間が必要です。大規模なプロジェクトでは、これには数分かかります。個別のファイルをコンパイルする場合、このコストは発生しません。
- コンパイラーがデバイスコードの再コンパイルが必要であると誤って判断した場合、偽陽性となることがあります。単一のソースコードでは、ホストとデバイスコードが混在するため、ホストコードに変更を加えるとコンパイラーによるデバイスコードのビューが変わる可能性があります。コンパイラーは常に保守的であり、以前の FPGA バイナリーの再利用が安全であると確認できない場合、完全な再コンパイルをトリガーします。個別にファイルをコンパイルすると、これを排除できます。

4.8.8 複数の FPGA イメージを作成（Linux* のみ）

インテル® oneAPI DPC++/C++ コンパイラーの機能を使用して、FPGA コンパイルを異なる FPGA イメージに分割することができます。この機能は、プロジェクトの設計が単一の FPGA に適合しない場合に役立ちます。これにより大規模な設計を複数の小さなイメージに分割することで、FPGA デバイスを部分的に再構成することができます。

次のいずれかのアプローチで設計を分割できます。それぞれに利点があります。

- 動的リンク
- 動的ロード

動的リンクは動的ロードよりも実装が容易ですが、すべてのデバイスイメージをメモリーにロードする必要があるため、動的リンクではホストデバイスに多くのメモリーが必要になることがあります。動的ロードにはそのような制限はありませんが、ソースレベルで変更が必要になります。次の表はそれぞれの違いを示しています。

	動的リンク	動的ロード
実行時に FPGA イメージを動的に変更できるか?	はい	はい
FPGA イメージのタイプと数の定義	コンパイル時	実行時
ホストプログラムのメモリー容量	すべての FPGA イメージは実行時にメモリーに展開されます。	明示的にロードされた FPGA イメージのみが実行時にメモリーに展開されます。
ホストコードの呼び出し	動的ライブラリー関数を直接呼び出します。	呼び出す動的ライブラリーと関数を明示的にロードします。

動的リンク

この手順では、設計を複数のソースファイルに分割して、それぞれを個別の FPGA イメージにマッピングできます。FPGA イメージ数が少ない設計でこの手順を使用することを推奨します。

この手順を使用するには、次を行います。

1. 必要な FPGA イメージごとにカーネルを送信する個別の `.cpp` ファイルを生成するようにソースコードを分割します。ホストコードを 1 つ以上の `.cpp` ファイルに分割し、カーネルファイル内の関数を参照できるようにします。

   次のような 3 つのファイルがあると想定します。

- ホストコードを含む main.cpp。

```
// host.cpp
int main() {
    queue queueA;
    add(queueA);
    mul(queueA);
}
```

- vector_add カーネルを送信する関数を含む vector_add.cpp。

```
// vector_add.cpp
extern "C"{
    void add(queue queueA){
        queue.submit(
            // Kernel Code
        );
    }
}
```

- vector_mul カーネルを送信する関数を含む vector_mul.cpp。

```
// vector_mul.cpp
extern "C"{
    void mul(queue queueA){
        queue.submit(
            // Kernel Code
        );
    }
}
```

2. 次のコマンドでソースファイルをコンパイルします。

```
dpcpp -fPIC -fintelfpga -c vector_add.cpp -o vector_add.o
dpcpp -fPIC -fintelfpga -c vector_mul.cpp -o vector_mul.o


# FPGA イメージのコンパイルにはかなり時間がかかります
```

```
dpcpp -fPIC -shared -fintelfpga vector_add.o -o vector_add.so -Xshardware -
Xsboard=pac_a10

dpcpp -fPIC -shared -fintelfpga vector_mul.o -o vector_mul.so -Xshardware -
Xsboard=pac_a10



# 最後のリンクステップ

dpcpp -o main.exe main.cpp vector_add.so vector_mul.so
```

この手順では、長い FPGA コンパイル手順を個別のコマンドに分割して、異なるシステムで実行したり、ファイルを変更した場合にのみ実行するようにできます。

動的ロード

異なる FPGA イメージを一括してメモリーにロードしないようにするには、この手順を使用します。動的リンクと同様にこの手順でもコードを分割する必要があります。ただし、ここではホストプログラムに `.so`(共有オブジェクト)ファイルをロードする必要があります。この手順の利点は、コンパイル時にすべてのイメージファイルをリンクすることなく、必要に応じて大きな FPGA イメージファイルを動的にロードできることです。

この手順を使用するには次を行います。

1. 動的リンクのステップ 1 で行ったのと同じ方法でソースコードを分割します。
2. `host.cpp` ファイルを次のように変更します。

```
// host.cpp

#include <dlfcn.h>


int main() {

    queue queueA;

    bool runAdd, runMul;

    // runAdd と runMul が実行時に動的に設定されると想定

    if (runAdd) {

        auto add_lib = dlopen("./add.so", RTLD_NOW);

        auto add = dlsym(add_lib, "add");

        add(queueA);

    }

    if (runMul) {
```

```
        mul_lib = dlopen("./mul.so", RTLD_NOW);

        mul = dlsym(mul_lib, "mul");

        mul(queueA);

    }

}
```

3. 次のコマンドでソースファイルをコンパイルします。

注: .so ファイルは実行時に動的にロードされるため、コンパイル時にリンクする必要はありません。

```
dpcpp -fPIC -fintelfpga -c vector_add.cpp -o vector_add.o

dpcpp -fPIC -fintelfpga -c vector_mul.cpp -o vector_mul.o


# FPGA イメージのコンパイルにはかなり時間がかかります

dpcpp -fPIC -shared -fintelfpga vector_add.o -o vector_add.so -Xshardware -
Xsboard=pac_a10

dpcpp -fPIC -shared -fintelfpga vector_mul.o -o vector_mul.so -Xshardware -
Xsboard=pac_a10


# 最後のリンクステップ。.so ファイルを含むパスを LD_LIBRARY_PATH に追加します

LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH dpcpp -o main.exe main.cpp vector_add.so
vector_mul.so
```

このアプローチでは、必要に応じて実行時に .so ファイルを任意にロードできます。これには、FPGA イメージの大規模なライブラリーがあり、そこからファイルのサブセットを選択する場合に役立ちます。

4.8.9 FPGA BSP とボード

「DPC++ FPGA コンパイルの種別」で説明したように、FPGA ハードウェア・イメージを生成するには、「oneAPI ベース・ツールキット用インテル® FPGA アドオン」(英語)が必要です。これには、RTL から FPGA のプリミティブ・ハードウェア・リソースに設計をマッピングするインテル® Quartus® Prime 開発ソフトウェアが用意されています。さらに、このアドオンパッケージは、FPGA ハードウェアへのコンパイルに必要な基本的なボード・サポート・パッケージ(BSP)を提供します。

ボードとは?

GPU と同様に、FPGA はサーバーまたはデスクトップ・コンピューターに搭載されるカードまたはマウントされる集積回路です。FPGA ボードは、メモリー、電力、熱管理、およびほかのデバイスと通信する物理インターフェイスを提供します。

BSP とは?

BSP は、ソフトウェア・レイヤーと FPGA ハードウェア・スキャホールド・デザインで構成されており、インテル® oneAPI DPC++/C++ コンパイラーを介して FPGA をターゲットにできます。コンパイラーによって生成された FPGA デザインは、BSP で提供されるフレームワークに組込まれています。

ボードバリアントとは?

BSP は、さまざまな機能をサポートする複数のボードバリアントを提供できます。例えば、`intel_s10sx_pac` BSP には、統合共有メモリー(USM)のサポート方法が異なる 2 つのバリアントがあります。USM の詳細については、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドとリファレンス』の「統合共有メモリー(USM)」(英語)を参照してください。

注: ボードが複数の BSP をサポートし、BSP は複数のボードバリアントをサポートする場合があります。

oneAPI ベース・ツールキット用インテル® FPGA アドオンは、2 つのボード BSP を提供し、この BSP によってサポートされるボードバリアントは `dpcpp` コマンドで次のオプションを使用して選択できます。

ボード	BSP	オプション	USM サポート
インテル® プログラマブル・アクセラレーション・カード(インテル® Arria® 10 GX FPGA 搭載版)	`intel_a10gx_pac`	`-Xsboard=intel_a10gx_pac:pac_a10`	明示的な USM
インテル® FPGA PAC D5005(インテル® FPGA プログラマブル・アクセラレーション・カード D5005) (旧名称:インテル® プログラマブル・アクセラレーション・カード(インテル® Stratix® 10 SX FPGA 搭載版))	`intel_s10sx_pac`	`-Xsboard=intel_s10sx_pac:pac_s10`	明示的な USM
		`-Xsboard=intel_s10sx_pac:pac_s10_usm`	明示的な USM 制限付き USM

注:

- インテル® oneAPI DPC++/C++ コンパイラー(インテル® oneAPI ベース・ツールキットに含まれます)は、FPGA の初期イメージを生成するのに十分な部分的な BSP を提供します。対照的に、oneAPI ベース・ツールキット用インテル® FPGA アドオンは、FPGA ハードウェア・イメージの生成に必要な完全な BSP を提供します。
- FPGA ボードで DPC++ の実行形式ファイルを実行する場合、実行形式ファイルがターゲットのボードバリアント向けに FPGA ボードを初期化したことを確認する必要があります。FPGA ボードの初期化については、「FPGA ボードの初期化」を参照してください。
- 制限付き USM に適用可能な FPGA の最適化については、『インテル® oneAPI DPC++ FPGA 最適化ガイド』(英語)の「事前固定(プリピニング)」(英語)と「ゼロコピーのメモリーアクセス」(英語)を参照してください。

カスタム BSP

上記の BSP に加えて、ボードベンダーが提供するカスタム BSP を使用することもできます。カスタム BSP でインテル® oneAPI DPC++/C++ コンパイラーを使用するには、次の手順に従ってください。

1. カスタム BSP に必要なバージョンのインテル® Quartus® Prime 開発ソフトウェアを含む、インテル® カスタム・プラットフォーム・パッケージ向けのインテル® FPGA アドオンをインストールします。詳細については、『インテル® oneAPI ツールキット・インストール・ガイド』(英語)を参照してください。
2. アドオンのインストール先ディレクトリー(`intelfpgadpcpp/latest/board`) に完全なカスタム BSP を配置します。
3. BSP ベンダーが提供するインストール手順に従ってください。

FPGA ボードの初期化

FPGA ハードウェア・イメージを含む DPC++ 実行形式を実行する前に、次のコマンドを使用して FPGA ボードを初期化する必要があります。

```
aoclinitialize<board id> <board variant>
```

説明:

パラメーター	説明
<board_id>	`aocl diagnose` コマンドから取得したボード ID。例:`ac10`、`ac11` など。
<board variant>	DPC++ 実行形式のコンパイル時に `-Xsboard` オプションで指定されたボードバリアントの名前。例:`pac_s10_usm`。

例えば、システムに 1 台のインテル® FPGA PAC D5005（インテル® FPGA プログラマブル・アクセラレーション・カード D5005）（旧名称：インテル® Stratix® 10 SX FPGA 搭載版）が搭載され、次のコマンドで DPC++ 実行形式ファイルをコンパイルすると仮定します。

```
dpcpp -fintelfpga -Xshardware -Xsboard=intel_s10sx_pac:pac_s10_usm kernel.cpp
```

この場合、次のコマンドでボードを初期化する必要があります。

```
aocl initialize acl0 pac_s10_usm
```

初期化が完了すると、次のいずれかを実行する場合を除き、ボードを再び初期化することなく DPC++ 実行形式ファイルを実行できます。

- ホストの電源をオンにした後、初めて DPC++ ワークロードを実行する。
- FPGA で DPC++ ワークロード以外を実行した後に DPC++ ワークロードを実行する。
- `-Xsboard` オプションで別のボードバリアントでコンパイルされた DPC++ ワークロードを実行する。

FPGA ハードウェア・イメージの情報を取得

`aocl binedit` ユーティリティーは、コンパイルされたバイナリーから次のような情報を抽出できます。

- コンパイルした環境の詳細: コンパイラーのバージョン、使用したコンパイルコマンド、インテル® Quartus® Prime ソフトウェアのバージョンなど。
- コンパイルに使用した BSP の board_spec.xml。
- カーネル fMAX (Quartus がコンパイルしたfMAX)。
- コンパイルに使用した BSP およびボード。

構文

`aocl binedit` ユーティリティーは、次のコマンドで使用します。

```
aocl binedit <oneapi_binary> <list/get/print/exists> [<section_name>
[output_file]]
```

以下に利用可能なアクションを示します。

- list: バイナリー中の利用可能な <section_name> をすべてリストします。
- print: バイナリー中のパッケージファイルの指定されたセクションの内容を標準出力に書き出します。
- get: 指定されたセクションの内容を出力ファイルに書き出します。
- exists: セクションがバイナリーのパッケージファイルに存在するか確認します。0 以外の終了コードはセクションが存在しないことを示します。

例えば、シミュレーター・フローでコンパイルされたバイナリーを次のコマンドで SimulatorDevice に出力します。

```
$ aocl binedit <oneapi_binary> print .acl.board
```

また、次のコマンドで BSP のバージョンを確認できます。

```
$ aocl binedit <oneapi_binary> print .board_package
```

4.8.10 複数の同種の FPGA デバイスをターゲットにする

インテル® oneAPI DPC++/C++ コンパイラーは、単一ホスト CPU から複数の同種の FPGA デバイスをターゲットにすることをサポートします。これにより、複数の FPGA でプログラムを並列化して実行することで、設計のスループットを向上できる可能性があります。

複数コンテキストでは、OpenCL* レイヤーのバッファーをコンテキスト間でコピーする必要があり、オーバーヘッドが発生して全体のパフォーマンスに影響を及ぼすため、複数のデバイスキューを使用して単一のコンテキストを作成することを推奨します。ただし、設計が単純でオーバーヘッドが全体のパフォーマンスに影響しない場合は、複数コンテキストを使用できます。

次のいずれかの方法で複数の FPGA デバイスをターゲットにします。

複数のデバイスキューで単一のコンテキストを作成

次の手順を実行して、単一のコンテキストで複数の FPGA デバイスをターゲットにします。

1. 単一の SYCL* コンテキストを作成し、同じプラットフォームの FPGA デバイスのコレクションをカプセル化します。

```
context ctxt(deviceList, &m_exception_handler);
```

2. それぞれの FPGA デバイス向けに SYCL* キューを作成します。

```
std::vector<queue> queueList;
for (unsigned int i = 0; i < ctxt.get_devices().size(); i++) {
  queue newQueue(ctxt, ctxt.get_devices()[i], &m_exception_handler);
  queueList.push_back(newQueue);
}
```

3. 利用可能なすべての FPGA デバイスに、同一または異なるデバイスコードを送信します。すべてのデバイスのサブセットをターゲットにする場合、不要なデバイスを除外するため最初にデバイスの選択を実行する必要があります。

```
for (unsigned int i = 0; i < queueList.size(); i++) {
  queueList[i].submit([&](handler& cgh) {...});
}
```

それぞれのデバイスキュー向けにコンテキストを作成(複数コンテキスト)

次の手順を実行して、複数のコンテキストで複数の FPGA デバイスをターゲットにします。

1. 利用可能な FPGA デバイスのリストを取得します。オプションで、デバイスメンバーまたはデバイス・プロパティーに基づいてデバイスを選択できます。デバイス名などのデバイス・プロパティーは、対象のデバイス・プロパティーとともにメンバー関数 `get_info()const` を使用します。

```
std::vector<device> deviceList = device::get_devices();
```

2. それぞれの FPGA デバイス向けに SYCL* キューを作成します。

```
std::vector<queue> queueList;
for (unsigned int i = 0; i < deviceList.size(); i++) {
  queue newQueue(deviceList[i], &m_exception_handler);
  queueList.push_back(newQueue);
}
```

3. 利用可能なすべての FPGA デバイスに、同一または異なるデバイスコードを送信します。すべてのデバイスのサブセットをターゲットにする場合、不要なデバイスを除外するため最初にデバイスの選択を実行する必要があります。

```
for (unsigned int i = 0; i < queueList.size(); i++) {
  queueList[i].submit([&](handler& cgh) {...});
}
```

制限事項

複数の FPGA デバイスをターゲットにする場合、次の制限事項を考慮してください。

- すべての FPGA デバイスは同じ FPGA ビットストリームを使用します。
- 使用するすべての FPGA デバイスは、同一の FPGA カードである必要があります(同じ `-Xsboard target`)。

4.8.11 複数のプラットフォームをターゲットにする

(異なるデバイスセレクターを使用して)複数のターゲット･デバイス･タイプを対象とする設計をコンパイルするには、次のコマンドを実行します。

エミュレーション･コンパイル

FPGA エミュレーター･ターゲットの SYCL* コードのコンパイルには、次のコマンドを使用します。

```
# Linux* 向け:
dpcpp jit_kernel.cpp -c -o jit_kernel.o

dpcpp -fintelfpga -fsycl-link=image -DFPGA_EMULATOR fpga_kernel.cpp \
-o fpga_kernel.a

dpcpp -fintelfpga main.cpp jit_kernel.o fpga_kernel.a
```

```
# Windows* 向け:
dpcpp jit_kernel.cpp -c -o jit_kernel.o

dpcpp -fintelfpga -fsycl-link=image -DFPGA_EMULATOR fpga_kernel.cpp \
-o fpga_kernel.lib

dpcpp -fintelfpga main.cpp jit_kernel.o fpga_kernel.lib
```

この例は、FPGA カーネル(AOT 手順)と CPU カーネル(JIT 手順)およびライブラリーを使用します。

具体的には、`main.cpp` ファイルに `main` 関数が含まれ、CPU(`jit_kernel.cpp`)と FPGA(`fpga_kernel.cpp`)の両方のカーネルを必要とします。

サンプルの jit_kernel.cpp ファイル:

```
sycl::cpu_selector device_selector;
queue deviceQueue(device_selector);
deviceQueue.submit([&](handler &cgh) {
  // CPU Kernel function
});
```

サンプルの fpga_kernel.cpp ファイル:

```
#ifdef FPGA_EMULATOR
INTEL::fpga_emulator_selector device_selector;
#else
INTEL::fpga_selector device_selector;
#endif
queue deviceQueue(device_selector);
deviceQueue.submit([&](handler &cgh) {
  // FPGA Kernel Function
```

```
});
```

FPGA ハードウェア･コンパイル

FPGA ハードウェア･ターゲット向けにコンパイルするには、-DFPGA_EMULATOR オプションの代わりに -Xshardware オプションを指定します。

```
# Linux* 向け:
dpcpp jit_kernel.cpp -c -o jit_kernel.o

\\ ハードウェアのコンパイルコマンド。時間がかかります。
dpcpp -fintelfpga -fsycl-link=image -Xshardware fpga_kernel.cpp \
-o fpga_kernel.a dpcpp -fintelfpga main.cpp jit_kernel.o fpga_kernel.a
```

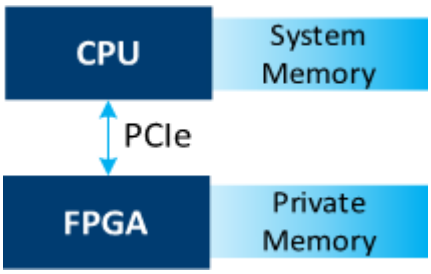
```
# Windows* 向け:
dpcpp jit_kernel.cpp -c -o jit_kernel.o

\\ ハードウェアのコンパイルコマンド。時間がかかります。
dpcpp -fintelfpga -fsycl-link=image -Xshardware fpga_kernel.cpp \
-o fpga_kernel.lib dpcpp -fintelfpga main.cpp jit_kernel.o fpga_kernel.lib
```

4.8.12 FPGA-CPU インタラクション

FPGA の設計におけるパフォーマンスに影響する主な要因の 1 つは、FPGA で実行されるカーネルが CPU ホストとどのような相互作用を持つかです。

ホストとカーネルのインタラクション

FPGA デバイスは PCIe* を介してホスト(CPU)と通信します。



これは、FPGA をターゲットにする SYCL* プログラムのパフォーマンスに影響する重要な要素です。さらに、特定のDPC++ プログラムを初めて実行する場合、FPGA をハードウェア･ビットストリームで構成する必要があり、これには数時間かかります。

データ転送

通常、FPGA ボードには、独自のプライベート DDR(英語)メモリーが搭載されます。CPU は、カーネルが FPGA ローカル DDR メモリーにアクセスして使用するすべてのデータを一括転送するか、ダイナミック・メモリー・アクセス(DMA)を行う必要があります。カーネルは操作を完了すると、結果を DMA 経由で CPU に転送する必要があります。転送速度は PCIe* リンクと DMA の効率によって制限されます。例えば、インテル® プログラマブル・アクセラレーション・カード(インテル® Arria® 10 GX FPGA 搭載版)には、PCIe* Gen 3 x 8 リンクがあり、転送は通常 6 から 7 GB/秒に制限されます。

データ転送時間は次のように管理されます。

- DPC++ では、バッファーに読み取り専用または書き込み専用のタグを付けることができ、不要な転送を排除できます。
- 同時操作数を最大化することでシステム全体の効率を改善できます。PCIe* は逆方向への同時転送をサポートし、PCIe* 転送はカーネルの実行に干渉しないため、ダブル・バッファリングなどの手法を利用できます。これらの手法の詳細については、『インテル® oneAPI DPC++ FPGA 最適化ガイド』の「カーネル呼び出しキューを利用するダブル・バッファリング・ホスト」(英語)と、「double_buffering」(英語)チュートリアルを参照してください。
- 制限付き USM をサポートするボードバリアントにシステムメモリーを事前に固定することで、データ転送スループットを向上させます。詳細については、『インテル® oneAPI DPC++ FPGA 最適化ガイド』の「事前固定(プリピニング)」(英語)を参照してください。

コンフィギュレーション時間

コンフィギュレーションと呼ばれる手順で、FPGA デバイスのハードウェア・ビットストリームをプログラムします。コンフィギュレーションは、FPGA デバイスとの通信に数秒間を要する操作です。SYCL* ランタイムはコンフィギュレーションを自動的に管理します。ランタイムは、コンフィギュレーションが行われるタイミングを決定します。例えば、カーネルが最初に起動されたときにコンフィギュレーションがトリガーされることがありますが、ビットストリームは変更されていないため、同じカーネルを以降に起動する際にはコンフィギュレーションがトリガーされないことがあります。したがって、開発中は、FPGA のコンフィギュレーション後にカーソルの実行時間を計測することを推奨します。例えば、カーネルの実行時間を計測する前に、カーネルをウォームアップ実行します。ウォームアップ・コードは、実際のコードでは削除します。

複数のカーネル呼び出し

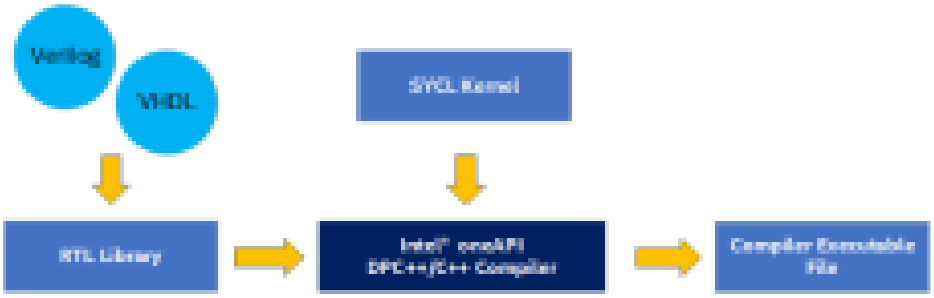
SYCL* プログラムが同じカーネルを SYCL* キューに複数回送信する場合(ループ内での `single_task` 呼び出しなど)、1 つのカーネル呼び出しのみがアクティブになります。それ以降のカーネル呼び出しでは、前のカーネルが実行を完了するのを待機します。

4.8.13 FPGA パフォーマンスの最適化

前述の FPGA 手順では、FPGA 向けの DPC++ コンパイルの基本を説明しましたが、設計時にパフォーマンス向上について習得すべきことはまだたくさんあります。インテル® oneAPI DPC++/C++ コンパイラーは、改善すべき領域を特定するツール、設計およびコンパイラーの動作を制御するさまざまなオプション、属性、そして拡張機能を提供します。これらの情報はすべて、『インテル® oneAPI DPC++ FPGA 最適化ガイド』(英語)に記載されています。設計時の最適化に関する手法を理解したい開発者はこのガイドを参照してください。

4.8.14 FPGA 向けの RTL ライブラリーを使用する

静的ライブラリーは複数の関数を含む単一のファイルです。静的ライブラリー・ファイルは、SYCL*、OpenCL*、高位合成(HLS)ソース、またはレジスター転送レベル(RTL)を使用して作成できます。次に、このライブラリー・ファイルをインクルードして、SYCL* カーネル内で関数を使用します。



SYCL* で使用するライブラリーを生成するには、次のファイルを用意する必要があります。

SYCL* で使用するライブラリーを作成

ファイルまたはコンポーネント	説明
RTL モジュール	
RTL ソースファイル	RTL コンポーネントを定義する、Verilog、System Verilog、または VHDL ファイル。 インテル® Quartus® Prime IP ファイル(`.qip`)、Synopsys Design Constraints ファイル(`.sdc`)、Tcl スクリプトファイル(`.tcl`)などの追加ファイルは許可されません。
eXtensible マークアップ言語ファイル(`.xml`)	RTL コンポーネントのプロパティーを記述します。インテル® oneAPI DPC++/C++ コンパイラーは、これらのプロパティーを使用して RTL コンポーネントを SYCL* パイプラインに統合します。

ファイルまたはコンポーネント	説明
ヘッダーファイル(`.hpp`)	有効な SYCL* カーネル言語を含み、RTL コンポーネントによって実装される関数のシグネチャーを宣言するヘッダーファイルです。
エミュレーション・モデル・ファイル(OpenCL* または HLS ベース)	エミュレーションでのみ使用される RTL コンポーネントの C または C++ モデルを提供します。ハードウェア向けの完全なコンパイルには RTL ソースファイルを使用します。
SYCL* 関数	
SYCL* ソースファイル(`.cpp`)	SYCL* 関数の定義を含みます。これらの関数は、エミュレーションおよび完全なハードウェア・コンパイルで使用されます。
ヘッダーファイル(`.hpp`)	SYCL* 構文で SYCL* から呼び出される関数を記述するヘッダーファイル。

ライブラリー・ファイルの形式は、ソースコードをコンパイルするオペレーティング・システムと同じ形式を使用し、ライブラリー情報を伝達する追加のセクションを持ちます。

- Linux* プラットフォームでは、ライブラリーは `.a` アーカイブファイルで `.o` オブジェクト・ファイルを含みます。
- Windows* プラットフォームでは、ライブラリーは `.lib` アーカイブファイルで `.obj` オブジェクト・ファイルを含みます。

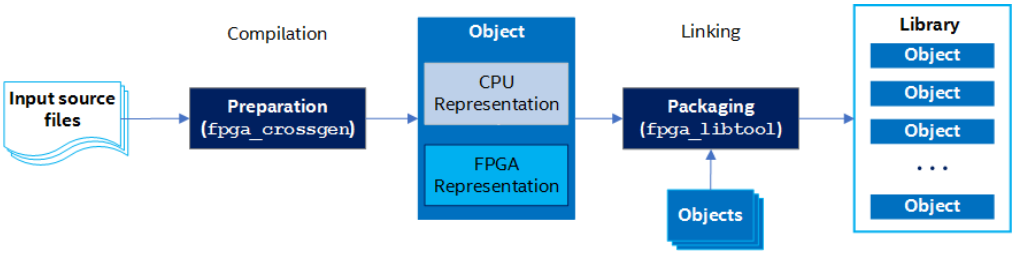
ライブラリー内部の関数のハードウェアの設計や実装の詳細を理解していなくても、カーネルからライブラリーの関数を呼び出すことができます。カーネルをコンパイルする際にライブラリーを `dpcpp` コマンドラインに追加します。

ライブラリーの作成は 2 段階のプロセスからなります。

1. 各オブジェクト・ファイルは、`fpga_crossgen` コマンドにより入力ソースファイルから作成されます。
    - オブジェクト・ファイルは、コードの CPU 形式と FPGA 形式の両方からなるソースコードの中間表現です。
    - オブジェクト・ファイルは、同一のインテル® HLS 設計製品でのみ使用することが想定されています。複数の HLS 設計製品を使用する場合、対象の製品ごとに個別のオブジェクトを作成する必要があります。

2. オブジェクト・ファイルは `fpga_libtool` コマンドでライブラリー・ファイルに結合します。すべてのオブジェクトが同一の HLS 設計製品を対象とする場合、各種タイプのソースコードから生成されたオブジェクトをライブラリーに統合できます。

   ライブラリーにはツールチェーンのバージョン番号が自動的に割り当てられるため、同じバージョンの HLS 設計製品でのみ利用します。



ソースコードからライブラリー・オブジェクトを作成

ソースコードのオブジェクト・ファイルからライブラリーを作成することができます。SYCL* ベースのオブジェクト・ファイルには、ホストで使用する CPU を取得するハードウェア実行とカーネルのエミュレーションで使用する CPU コードが含まれています。

ソースコードからオブジェクト・ファイルを作成

`fpga_crossgen` コマンドを使用して、ソースコードからライブラリー・オブジェクトを作成します。ソースコードから作成されたオブジェクトには、オブジェクト内の関数のエミュレートとオブジェクト関数のハードウェア合成に必要な情報が含まれています。

`fpga_crossgen` コマンドは、1 つの入力ソースファイルから 1 つのオブジェクト・ファイルを作成します。作成されたオブジェクトは、同じインテル® HLS 設計ツールを対象とするライブラリーでのみ利用できます。オブジェクトもまたバージョン管理されます。それぞれのオブジェクトには、コンパイラーのバージョン番号が割り当てられ、同一バージョンのインテル® HLS 設定ツールでのみ使用されます。

次のコマンドを使用してライブラリー・オブジェクトを作成します。

```
fpga_crossgen .xml --emulation_model .cpp --target sycl -o
<object_file>
```

次の表でパラメーターについて説明します。

パラメーター	説明
`<source_file>`	SYCL*、OpenCL*、HLS、または RTL ソースファイルを使用できます。
`--target`	ライブラリー用のインテル® HLS 設計ツール(`sycl`、`ocl`、または `hls`)を対象とします。オブジェクトは、`fpga_libtool` を使用して SYCL* ライブラリー・アーカイブに統合されます。
`-o`	オプション。このオプションは、オブジェクト・ファイル名を指定するのに役立ちます。このオプションを指定しないと、オブジェクト・ファイル名はソースコードのファイル名と同じ名前になりますが、拡張子は `.o` や `.obj` になります。

コマンドの例を示します。

```
fpga_crossgen lib_rtl_spec.xml --emulation_model lib_rtl_model.cpp --source sycl
--target sycl - o lib_rtl.o
```

オブジェクト・ファイルをライブラリー・ファイルにパッケージ化

ほかの開発者がライブラリーをプロジェクトに組込んで、ライブラリー内のオブジェクトに含まれる関数を呼び出すことができるように、オブジェクト・ファイルをライブラリー・ファイルに集約します。オブジェクト・ファイルをライブラリーとしてパッケージ化するには、`fpga_libtool` コマンドを使用します。

オブジェクト・ファイルをライブラリーにパッケージ化する前に、ライブラリーに含めるすべてのオブジェクト・ファイルのパス情報があることを確認してください。

パッケージ含めるすべてのオブジェクトは、同じバージョンでなければなりません。`fpga_libtool` コマンドは、オペレーティング・システム固有のアーカイブファイルのライブラリー(Linux* では `.a`、Windows* では `.lib`)を作成します。使用するオペレーティング・システムとは異なるオペレーティング・システムで実行されているインテル® HLS 設計製品と一緒に使用することはできません。

次のコマンドを使用してライブラリー・ファイルを作成します。

```
fpga_libtool file1 file2 ... fileN --target (sycl|ocl|hls) --create
<library_name>
```

コマンド・パラメーターは次のように定義されます。

パラメーター	説明
file1 file2 … fileN	ライブラリーに含めるオブジェクト・ファイルを指定できます。
`--target (sycl|ocl|hls)`	このライブラリーはカーネル開発を対象とします。`sycl` オプションでは、`--target` は、インテル® oneAPI DPC++/C++ コンパイラーで使用するライブラリーを準備します。
`--create <library_name>`	ライブラリー・アーカイブ・ファイルの名前を指定できます。Linux* プラットフォーム向けのライブラリーでは、ファイル拡張子 `.a` を指定します。

コマンド例:

```
fpga_libtool lib_rtl.o --target sycl --create lib.a
```

このコマンドは、HLS、OpenCL*、RTL、および SYCL* ソースコードから作成されたオブジェクトを `lib.a` という SYCL* ライブラリーにパッケージ化します。

静的ライブラリーの使用

次のコマンドに示すように、コンパイルコマンドにはソースファイルとともに静的ライブラリーを指定することができます。

```
dpcpp -fintelfpga main.cpp lib.a
```

注: RTL で実装した関数を利用するには、コンパイラーが関数を動的にリンクできるよう、ソースコードで宣言する必要があります。

```
SYCL_EXTERNAL extern "C" void foo()
```

RTL サポートの制限と制約

SYCL* カーネル内で使用する RTL モジュールを作成する場合、RTL モジュールが次の制限下で動作することを確認する必要があります。

- RTL モジュールは、単一入力のインテル® Avalon® ストリーミング・インターフェイスを使用しなければなりません。つまり、準備ができた有効なロジックのペアで、すべての入力をコントロールする必要があります。必要とするインテル® Avalon® ストリーミング・インターフェイス・ポートを提供するオプションもありますが、RTL モジュールをストールなしとして宣言する必要があります。この場合、インテル® oneAPI DPC++/C++ コンパイラーがモジュールのラッパーを作成するため、特定のストー

ル動作を実装する必要はありません。詳細については、「RTL モジュールのオブジェクト・マニフェスト・ファイル構文」(英語)を参照してください。

注: RTL モジュールに内部状態がある場合、`ivalid` 信号を適切に処理する必要があります。詳細は、「ストールなしの RTL」(英語)を参照してください。

- RTL モジュールは、カーネルクロック周波数に関係なく適切に動作しなければなりません。
- RTL モジュールは、外部 I/O 信号に接続できません。すべての入力と出力信号は SYCL* カーネルに由来する必要があります。
- RTL モジュールには、クロックポート、リセットポート、およびインテル® Avalon® ストリーミング・インターフェイスの入力ポートと出力ポート(`ivalid`、`ovalid`、`iready`、`oready`)が必要です。ここで示すようにポートに名前を付けます。
- 外部メモリーと通信を行う RTL モジュールには、対応するカスタム・プラットフォーム・パラメーターと一致するインテル® Avalon® メモリー・マップド・インターフェイス・ポートのパラメーターが必要です。インテル® oneAPI DPC++/C++ コンパイラーは、幅やバーストの適用を行うことはありません。
- 外部メモリーと通信を行う RTL モジュールは、次のように動作しなければなりません。
  - バースト境界を越えてバーストすることはできません。
  - クロックサイクルごとに要求を行い、アービトレーション・ロジックを占有してハードウェアをストールさせることはできません。RTL モジュールは、ほかのロードユニットやストアユニットが操作を実行できるよう、要求を定期的にポーズする必要があります。
- RTL モジュールは、スタンドアロンの SYCL* カーネルとしては動作できません。RTL モジュールはヘルパー関数としてのみ使用でき、カーネルのコンパイル中に SYCL* カーネルと統合できます。
- RTL モジュールのインスタンス化に対応するすべての関数呼び出しは、ほかのインスタンスから完全に独立しており、ハードウェアの共有もありません。
- カーネルコードを SYCL* ライブラリー・ファイルに組込んではなりません。カーネルコードを SYCL* ライブラリー・ファイルに組込むと、オフライン・コンパイラーはエラーメッセージを出力します。ヘルパー関数はライブラリー・ファイルに組込んでも問題ありません。
- RTL コンポーネントは、すべての入力を同時に受信する必要があります。単一の無効な入力には有効なデータが含まれていることを意味します。
- RTL モジュール・パラメーターは、`<RTL モジュール記述ファイル名>.xml` 仕様ファイルでのみ設定でき、SYCL* カーネルソースでは設定できません。複数のパラメーターで同じ RTL モジュールを利用するには、パラメーターの組み合わせごとに個別の `FUNCTION` タグを作成します。
- SYCL* カーネルを介してのみ RTL モジュールに入力データを渡すことができます。参照渡し、構造体、またはチャネルを介してデータ入力を RTL モジュールに渡してはなりません。チャネルデータの場合、抽出されたスカラーデータを渡します。

注: 参照渡しまたは構造体を介してデータ入力を RTL モジュールに渡すと、オフラインコンパイルで致命的エラーが発生します。

- ライブラリーがデバッグ情報を持たない場合、デバッガー(Linux* 向け GDB など)はエミュレーション中にライブラリー関数にステップインできません。しかし、ライブラリー・デバッグ情報の有無に関わりなく、最適化レポートとエリアレポートはライブラリー内のそれぞれのコード行にマップされません。
- RTL モジュールのソースファイル名は、インテル® oneAPI DPC++/C++ コンパイラーの IP ファイル名と同じにはできません。RTL モジュールのソースファイルとコンパイラーの IP ファイルは、`<カーネルファイル名>/system/synthesis/submodules` に保存されます。名前が競合すると、ディレクトリー内の既存のコンパイラー IP ファイルが RTL モジュールのソースファイルで上書きされます。
- コンパイラーは `.qip` ファイルをサポートしません。そのため、`.qip` ファイルを自身で解析して、RTL ファイルのフラットリストを作成する必要があります。

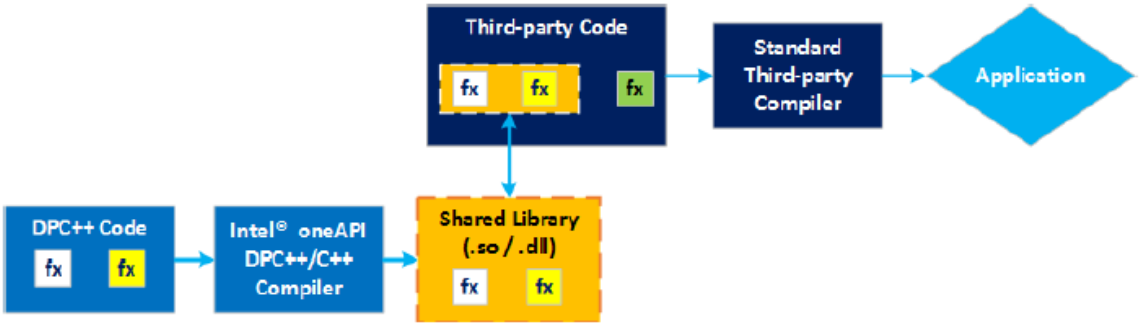
ヒント: 単独では正しく動作するにもかかわらず、SYCL* カーネルの一部として適切に機能しない RTL モジュールをデバッグするのは非常に困難です。`<RTL モジュール記述ファイル名>.xml` ファイルの「ATTRIBUTES エレメント」(英語)下にあるすべてのパラメーターを確認してください。

- すべてのコンパイラー領域推測ツールは、RTL モジュール領域が 0 であると想定します。現在、コンパイラーは RTL モジュール領域モデルを指定する機能をサポートしていません。

4.8.15 サードパーティーのアプリケーションで DPC++ 共有ライブラリーを使用

インテル® oneAPI DPC**/C++ コンパイラーを使用して、DPC++ コードを C 標準に準拠する共有ライブラリー(Linux* では `.so` ファイル、Windows* では `.dll` ファイル)にコンパイルします。次に、ほかのサードパーティーのコードからこのライブラリーを呼び出して、任意のプログラミング言語から高速化された関数にアクセスできます。

サードパーティー・アプリケーションで使用するため、共有ライブラリー・ファイルにパッケージ化された DPC++ 関数



サードパーティー・アプリケーションで DPC++ 共有ライブラリーを使用するには、次の手順を実行します。

1. 共有ライブラリー・インターフェイスを定義します
2. Linux* と Windows* でそれぞれ共有ライブラリー・ファイル を作成します
3. 共有ライブラリーを使用します

共有ライブラリーのインターフェイスを定義

C 標準共有ライブラリーと DPC++ コード間のインターフェイスを定義することを推奨します。インターフェイスにはエクスポートする関数と、DPC++ コードとの関係を含める必要があります。共有ライブラリーをインクルードする関数の前に extern "C" を記述します。

注: `extern "C"` プリフィクスを付けない場合、関数は共有ライブラリーにマングルされた名前で表示されます。

`vector_add` 関数を含む次の例について考えてみます。

```
extern "C" int vector_add(int *a, int *b, int **c, size_t vector_len) {
    // 対象とするデバイスのデバイスセレクターを作成します
    #if FPGA_EMULATOR
    // DPC++ 拡張機能: FPGA カードがないシステム向けの FPGA エミュレーター
    ext::intel::fpga_emulator_selector d_selector;
    #elif FPGA
    // DPC++ 拡張機能: FPGA カードを搭載するシステムの FPGA セレクター
    ext::intel::fpga_selector d_selector;
    #else
    // デフォルトのデバイスセレクターは、最もパフォーマンスが高いデバイスを選択します
```

```
    default_selector d_selector;
    #endif
    try {
        queue q(d_selector, exception_handler);
        // DPC++ コードのインターフェイス:
        // DPC++ のベクトル加算
        VectorAddKernel(q, a, b, c, vector_len);
    } catch (exception const &e) {
        std::cout << "An exception is caught for vector add.\n";
        return -1
    }
    return 0;
}
```

Linux* の共有ライブラリー・ファイルを作成

Linux* システムでは、次の手順で共有ライブラリー・ファイルを作成します。

1. DPC++ デバイスコードを個別にコンパイルします。

```
dpcpp -fPIC -fintelfpga -fsycl-link=image [カーネルソース名] -o <hw イメージ名> -Xshardware
```

各オプションは以下を示します。

fPIC	コンパイラーがデバイスイメージのホスト部分に対し位置独立コードを生成するか決定します。オプション -fPIC は、完全なシンボルのプリエンプションを指定します。グローバルシンボル定義とグローバルシンボル参照は、特に指定されない限りデフォルトの (プリエンプト可能な) 可視性を取得します。共有オブジェクトを作成する場合、このオプションを使用する必要があります。このオプションは -fpic とすることもできます。 **注:** 共有ライブラリー内のポインターがローカルアドレスではなくグローバルアドレスを参照するようにするため PIC が必要です。

fintelfpga	FPGA デバイスをターゲットにします。
fsycl-link=image	インテル® oneAPI DPC++/C++ コンパイラーに、FPGA で使用するデバイスバイナリーの部分リンクを行うように指示します。
Xshardware	エミュレーターではなくハードウェア向けにコンパイルします。

2. DPC++ ホストコードを個別にコンパイルします。

```
dpcpp -fPIC -fintelfpga <ホストのソースファイル> -o <ホストのイメージ名> -c -DFPGA=1
```

オプションは以下を示します。

DFPGA=1	コンパイラーマクロ FPGA を 1 に設定します。これは、ターゲットデバイスを変更するためデバイスセレクターで使用されます (これを可能にするには対応するホストコードが必要です)。デバイスセレクターを FPGA に設定することもできるため、これはオプションです。

3. ホストとデバイスのイメージをリンクしてバイナリーを作成します。

```
dpcpp -fPIC -fintelfpga -shared <ホストイメージ名><hw イメージ名> -o lib<ライブラリー名
>.so
```

オプションは以下を示します。

shared	出力する共有ライブラリー (.so ファイル)
出力ファイル名	GCC タイプのコンパイラーは lib プリフィクス。詳細については、Linux* 上の GCC との共有ライブラリーを参照。以下に例を示します。 gcc -Wall -fPIC -L. -o out.a -l<ライブラリー名>.so

注: 上記の複数ステップの手順の代わりに、単一ステップコンパイルで共有ライブラリーを作成することもできます。ただし、テスト用途で実行ファイル (a.out など) をビルドする場合、または DPC++ コードや C インターフェイスに変更を加える場合は、完全コンパイルを行う必要があります。

Windows* の共有ライブラリー・ファイルを作成

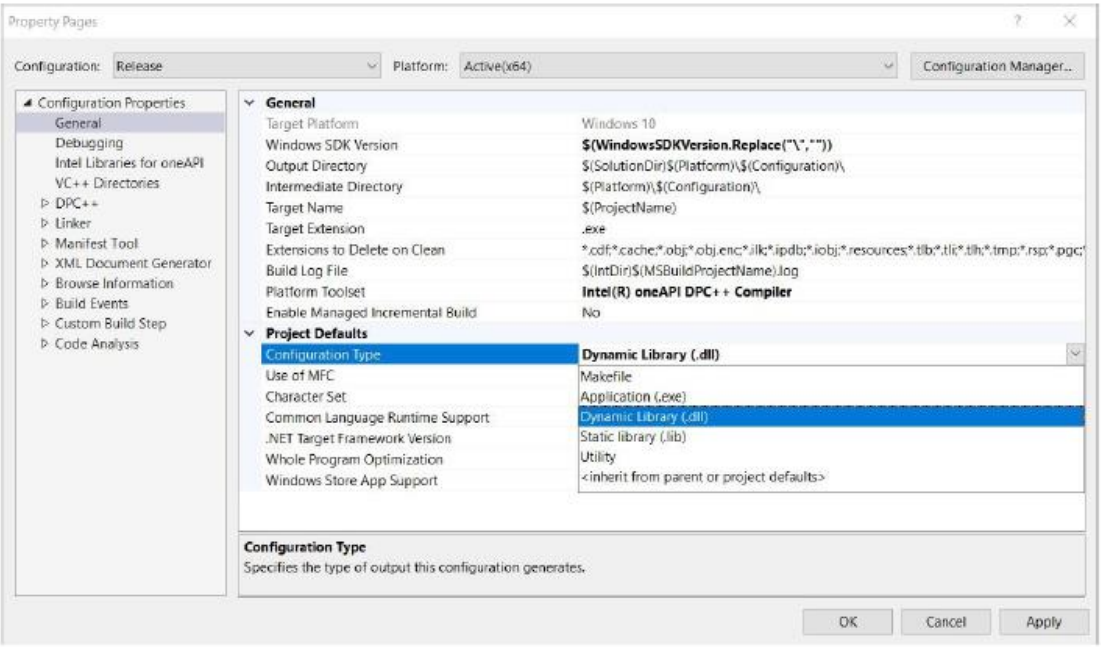
Windows* システムでは、次の手順でライブラリー・ファイルを作成します。

注:

- 同じプロジェクト・プロパティーで新しい構成を作成することを推奨します。これにより、アプリケーションをビルドする際に、プロジェクトの構成タイプを変更しなくても済みます。
- インテル® PAC(インテル® Arria® 10GX FPGA 搭載版)およびインテル® PAC(インテル® Stratix® 10 SX FPGA 搭載版)向けのデフォルトのインテル® oneAPI ベース・ツールキットおよび FPGA アドオンによる Windows* ライブラリーの作成は、FPGA エミュレーションでのみサポートされます。Windows* 向けのカスタム・プラットフォームの FPGA ハードウェア・コンパイルについては、ボードベンダーに問い合わせてください。

1. Microsoft* Visual Studio* で、[プロジェクト] > [プロパティ] に移動します。プロジェクトの [プロパティページ] ダイアログボックスが開きます。
2. [構成のプロパティ] > [全般] > [プロジェクトのデフォルト] > [構成の種類] オプションで、ドロップダウン・リストから [ダイナミック ライブラリー(.dll)] を選択します。

プロジェクトのプロパティ・ダイアログ



3. [OK] をクリックしてダイアログを閉じます。

プロジェクトは自動的にビルドされ、ダイナミック・ライブラリー(.dll)が生成されます。

共有ライブラリーを使用

ここで示す手順は、使用する言語またはコンパイラーによって異なる可能性があります。詳細については、使用する言語の仕様を参照してください。「Linux* 上の GCC と共有ライブラリー」(英語)を参照してください。

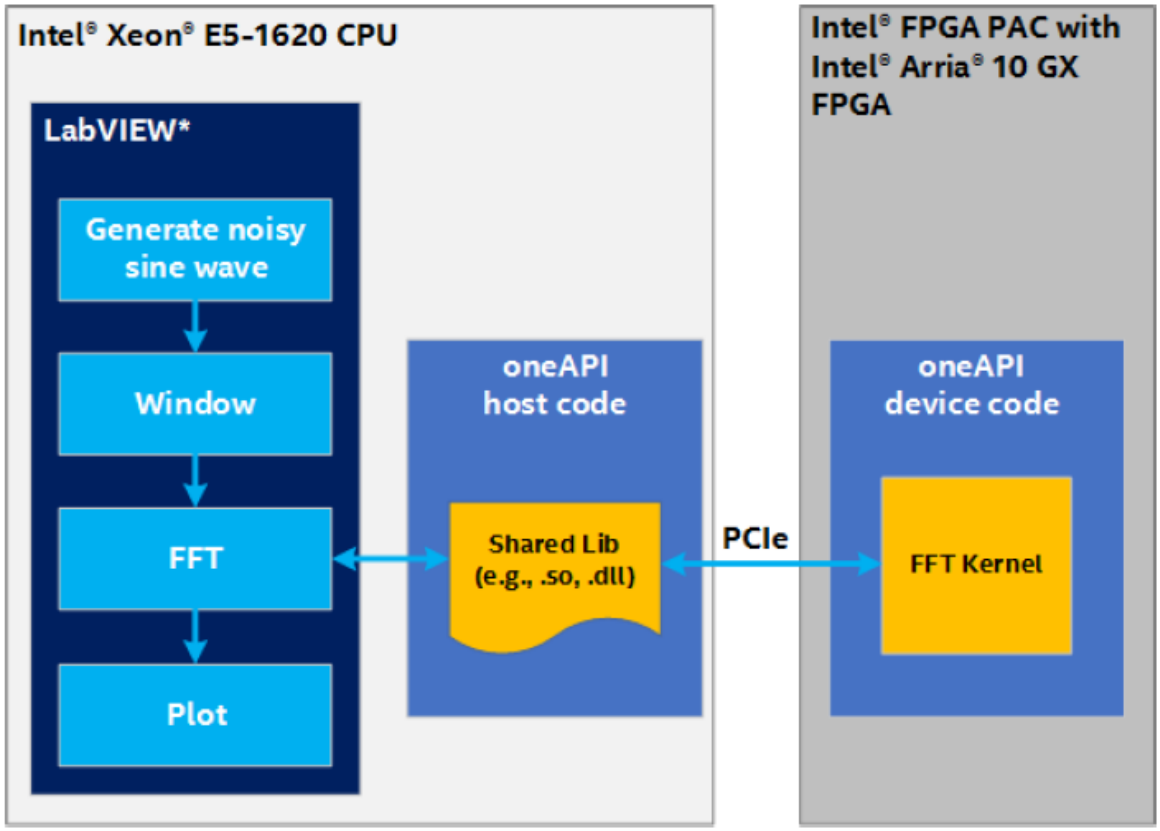
通常、共有ライブラリーを使用するには次の手順を実行します。

1. サードパーティーのホストコードで共有ライブラリー関数を呼び出します。
2. コンパイル中にホストコードを共有ライブラリーにリンクします。
3. ライブラリー・ファイルが検出できることを確認します。次に例を示します。

```
export LD_LIBRARY_PATH=<lib file location>:$LD_LIBRARY_PATH
```

以下に共有ライブラリーの利用例を示します。

共有ライブラリーの利用例



4.8.16 IDE での FPGA ワークフロー

インテル® oneAPI ツールは、サードパーティー統合開発環境(IDE)と統合して、Linux*(Eclipse*)および Windows*(Visual Studio*)でソフトウェア開発にシームレスな GUI 環境を提供します。詳細は、「サードパーティー IDE でのインテル® oneAPI DPC++ FPGA ワークフロー」(英語)を参照してください。

Linux* で Visual Studio* Code を使用した FPGA の開発については、「インテル® oneAPI DPC++ FPGA 開発(Linux* で Visual Studio* Code 向けのサンプルブラウザーを使用)」(英語)を参照してください。

API ベースのプログラミング

5

最適化されたアプリケーション向けの特殊 API を提供することで、プログラミング・プロセスを簡素化するインテル® oneAPI ツールキットのライブラリーを利用できます。この章では、サンプルコードを含むライブラリーの基本的な情報と、特定の利用ケースでどのライブラリーが最も有効であるか判断するのに役立つ情報を提供します。利用可能な API など、それぞれのライブラリーの詳細は、ライブラリーのドキュメントをご覧ください。

インテル® oneAPI ライブラリーの概要

インテル® oneAPI ツールキットには、次のライブラリーが含まれます。

ライブラリー	使用法
インテル® oneAPI DPC++ ライブラリー	ハイパフォーマンスの並列アプリケーションで使用します。
インテル® oneAPI マス・カーネル・ライブラリー	高度に最適化および並列化された数学ルーチンをアプリケーションで利用できます。
インテル® oneAPI スレッディング・ビルディング・ブロック	マルチコア CPU でのインテル® TBB ベースの並列処理と、アプリケーションでの DPC++ デバイス高速並列処理を組み合わせます。
インテル® oneAPI データ・アナリティクス・ライブラリー	ビッグデータ解析アプリケーションと分散計算を高速化します。
インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー	ディープラーニングとマシンラーニングのワークロードを処理するアプリケーションで使用します。
インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー	インテル® アーキテクチャーのプロセッサーおよびインテル® プロセッサー・グラフィックス向けに最適化された、ニューラル・ネットワークを使用するディープラーニング・アプリケーションで使用します。
インテル® oneAPI ビデオ・プロセシング・ライブラリー	アプリケーションのビデオ処理を高速化します。

5.1 インテル® oneAPI DPC++ ライブラリー(インテル® oneDPL)

インテル® oneAPI DPC++ ライブラリー(インテル® oneDPL)は、インテル® oneAPI DPC++.C++ コンパイラーと連携して生産性の高い API を提供します。これらの API を使用して、ハイパフォーマンスな並列アプリケーション向けに、デバイス全体でデータ並列 C++(DPC++)プログラミングの労力を最小限に抑えることができます。

インテル® oneDPL は次のコンポーネントで構成されます。

- Parallel STL
  - Parallel STL の使用手順
  - マクロ
- ライブラリー・クラスと関数の追加セット(このドキュメントでは**拡張 API** と呼びます)
  - 並列アルゴリズム
  - イテレーター
  - 関数オブジェクト・クラス
  - 範囲ベースの API
- テスト済みの標準 C++ API
- 乱数ジェネレーター

5.1.1 インテル® oneDPL ライブラリーの使い方

インテル® oneDPL を使用するには、インテル® oneAPI ベース・ツールキット(英語)をインストールします。

Patallel STL と拡張 API を使用するには、必要なヘッダーファイルをソースコードでインクルードします。すべてのインテル® oneDPL ヘッダーファイルは `oneapi/dpl` ディレクトリーにあります。それらをインクルードするには、`#include <oneapi/dpl/…>` を使用します。インテル® oneDPL には、大部分のクラスと関数対する名前空間 `oneapi::dpl` があります。テスト済みの C++ 標準 API を使用するには、対応する C++ ヘッダーファイルをインクルードし、`std` 名前空間を使用する必要があります。

5.1.2 インテル® oneDPL サンプルコード

インテル® oneDPL のサンプルコードは、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDPL(英語)から入手できます。それぞれのサンプルには、ビルド手順が説明された Readme が含まれています。

5.2 インテル® oneAPI マス･カーネル･ライブラリー(インテル® oneMKL)

インテル® oneAPI マス･カーネル･ライブラリー(インテル® oneMKL)は、最大限のパフォーマンスを必要とするアプリケーション向けに最適化され、広範囲に並列化されたルーチンの数学計算ライブラリーです。インテル® oneMKL には、C/Fortran プログラミング言語インターフェイスを備えた CPU アーキテクチャー向けのインテル® マス･カーネル･ライブラリー(インテル® MKL)のハイパフォーマンスな最適化が含まれており、各種 CPU アーキテクチャーとインテル® グラフィックス･テクノロジーでパフォーマンスを高めるデータ並列 C++(DPC++)プログラミング言語インターフェイスが追加されています。

OpenMP* オフロードを使用して、インテル® GPU で標準のインテル® oneMKL 計算を実行できます。詳細については、「C インターフェイスの OpenMP* オフロード」(英語)および「Fortran インターフェイスの OpenMP* オフロード」(英語)を参照してください。

CPU と GPU アーキテクチャー向けに最適化された新しい DPC++ インターフェイスには、次のような計算機能が追加されています。

- BLAS と LAPACK 密線形代数ルーチン
- スパース BLAS スパース線形代数ルーチン
- 乱数生成器(RNG)
- ベクトルの数学演算用に最適化されたベクトル数学(VM)ルーチン
- 高速フーリエ変換(FFT)

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、インテル® oneAPI マス･カーネル･ライブラリーのウェブサイトをご覧ください。oneMKL を oneAPI ベース･ツールキットの一部として利用する場合、有償オプションとしてプライオリティー･サポートが得られることを考慮してください。インテルのコミュニティー･サポートについては、oneMKL フォーラムを参照してください。コミュニティーがサポートするオープンソース･バージョンについては、oneMKL GitHub* のページを参照してください。

5.2.1 インテル® oneMKL の使い方

DPC++ プログラミング言語インターフェイスを使用する場合、考慮すべきことがあります。

- インテル® oneMKL は、インテル® oneAPI DPC++/C++ コンパイラーおよびインテル® oneAPI DPC++ ライブラリーに依存しています。アプリケーションは、インテル® oneAPI DPC++/C++ コンパイラーでビルドされ、DPC++ ヘッダーを利用しており、DPC++ リンカーを使用してインテル® oneMKL とリンクされている必要があります。

- インテル® oneMKL の DPC++ インターフェイスは、入力データ(ベクトル、行列など)にデバイスのアクセスが可能な統合共有メモリー(USM)ポインターを使用します。
- インテル® oneMKL の一部の DPC++ インターフェイスは、入力データ向けのデバイスへのアクセスが可能な USM(Unified Shared Memory)ポインターのほかに、`sycl::buffer` オブジェクトのアクセスもサポートしています。
- インテル® oneMKL の DPC++ インターフェイスは、浮動小数点型に基づいてオーバーロードされます。標準ライブラリー・タイプ `std::complex<float>`、`std::complex<double>` を使用する、単精度実数引数(float)、倍精度実数引数(double)、半精度実数引数(half)、および異なる精度の複素数引数を受け入れる汎用行列乗算 API があります。
- インテル® oneMKL の 2 レベルの名前空間構造が DPC++ インターフェイスに追加されます。

名前空間	説明
`oneapi::mkl`	oneMKL の各種ドメイン間の共通要素が含まれます。
`oneapi::mkl::blas`	密ベクトル-ベクトル、行列-ベクトル、および行列-行列の低レベル操作が含まれます。
`oneapi::mkl::lapack`	行列因数分解や固有値ソルバーなど高レベルの密行列演算が含まれます。
`oneapi::mkl::rng`	各種確率密度関数の乱数生成器が含まれます。
`oneapi::mkl::stats`	単精度および倍精度の多次元データセットの基本的な統計予測値が含まれます。
`oneapi::mkl::vm`	ベクトル数学ルーチンが含まれます。
`oneapi::mkl::dft`	高速フーリエ変換操作が含まれます。
`oneapi::mkl::sparse`	スパース行列ベクトル乗算、スパース三角ソルバーなどのスパース行列演算が含まれます。

5.2.2 インテル® oneMKL サンプルコード

DPC++ インターフェイスと oneMKL の一般的なワークフローを示すため、次のサンプルコードは、GPU デバイスで倍精度の行列 - 行列乗算を行います。

注: 次のコード例では、行中のコメントで示されるように、コンパイルと実行に追加のコードが必要です。

```
// 標準 SYCL* ヘッダー
#include <CL/sycl.hpp>
// STL クラス
#include <exception>
#include <iostream>
// インテル® oneAPI マス･カーネル･ライブラリーの DPC++ API の宣言
#include "oneapi/mkl_sycl.hpp"
int main(int argc, char *argv[]) {
    //
    // ユーザーは、m、n、k、ldA、ldB、ldC の設定と A、B、C 行列のデータを取得
    //
    // A、B、および C は、data() と size() メンバー関数を含む std::vector などの
    // コンテナーに格納します
    //

    // gpu デバイスを作成
    sycl::device my_device;
    try {
        my_device = sycl::device(sycl::gpu_selector());
    }
    catch (...) {
        std::cout << "Warning: GPU device not found! Using default device
instead." << std::endl;
    }
    // キューにアタッチする非同期例外ハンドラーを作成
    // 必須ではありませんが、システムが適切に構成されていない場合に参照できる
    // 有用な情報が提供されます
    auto my_exception_handler = [](sycl::exception_list exceptions) {
        for (std::exception_ptr const& e : exceptions) {
            try {
                std::rethrow_exception(e);
            }
            catch (sycl::exception const& e) {
                std::cout << "Caught asynchronous SYCL exception:\n"
                    << e.what() << std::endl;
            }
            catch (std::exception const& e) {
                std::cout << "Caught asynchronous STL exception:\n"
                    << e.what() << std::endl;
            }
        }
    };
    // 例外ハンドラーがアタッチされた gpu デバイスで実行キューを作成
    sycl::queue my_queue(my_device, my_exception_handler);
    // デバイスとホスト間のオフロード向けに行列データの SYCL* バッファーを作成
    sycl::buffer<double, 1> A_buffer(A.data(), A.size());
    sycl::buffer<double, 1> B_buffer(B.data(), B.size());
    sycl::buffer<double, 1> C_buffer(C.data(), C.size());
    // add mkl::blas::gemm を実行キューに追加し、同期例外をキャッチ
    try {
      using oneapi::mkl::blas::gemm;
      using oneapi::mkl::transpose;
```

```
        gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha,
            A_buffer, ldA, B_buffer, ldB, beta, C_buffer, ldC);
    }
    catch (sycl::exception const& e) {
        std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    catch (std::exception const& e) {
        std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    // 続行する前に、キャッチされた非同期例外を処理
    my_queue.wait_and_throw();
    //
    // 後処理の結果
    //
    // C バッファーからデータをアクセスし、C 行列の一部を出力
    auto C_accessor = C_buffer.template get_access<sycl::access::mode::read>();
    std::cout << "\t" << C << " = [ " << C_accessor[0] << ", "
        << C_accessor[1] << ", ... ]\n";
    std::cout << "\t  [ " << C_accessor[1 * ldC + 0] << ", "
        << C_accessor[1 * ldC + 1] << ", ... ]\n";
    std::cout << "\t  [ " << "... ]\n";
    std::cout << std::endl;

    return 0;
}
```

倍精度値行列 A(サイズ m x k)、B(サイズ k x n)、C(サイズ m x n)は、ホストマシンの配列(次元 ldA、ldB、ldC)に格納されると想定します。スカラー倍精度 alpha と beta を指定し、行列-行列乗算(`mkl::blas::gemm`)を計算します。

C = alpha * A * B + beta * C

標準 SYCL* ヘッダーと対象の `mkl::blas::gemm` API 宣言を含むインテル® oneMKL DPC++ 固有ヘッダーをインクルードします。

```
// 標準 SYCL* ヘッダー
#include <CL/sycl.hpp>
// STL クラス
#include <exception>
#include <iostream>
// インテル® oneAPI マス･カーネル･ライブラリーの DPC++ API の宣言
#include "oneapi/mkl.hpp"
```

次に、通常のようにホストマシンで行列データをロードまたはインスタンス化し、GPU デバイスを作成して非同期例外ハンドラーを作成し、最後に例外ハンドラーでデバイスキューを作成します。ホストで発生する例外は、標準 C++ 例外メカニズムでキャッチできます。ただし、デバイスで発生する例外は非同期エラーとして見なされ、例外リストに保存されて、ユーザー定義の例外ハンドラーによって処理されます。

```
    // gpu デバイスを作成
    sycl::device my_device;
    try {
        my_device = sycl::device(sycl::gpu_selector());
    }
    catch (...){
        std::cout << "Warning: GPU device not found! Using default device
instead."<< std::endl;
    }
    // キューにアタッチする非同期例外ハンドラーを作成
    // 必須ではありませんが、システムが適切に構成されていない場合に参照できる
    // 有用な情報が提供されます
    auto my_exception_handler = [](sycl::exception_list exceptions) {
        for (std::exception_ptr const& e : exceptions) {
            try {
              std::rethrow_exception(e);
            }
        catch (sycl::exception const& e) {
            std::cout << "Caught asynchronous SYCL exception:\n"
                << e.what() << std::endl;
            }
            catch (std::exception const& e) {
                std::cout << "Caught asynchronous STL exception:\n"
                    << e.what() << std::endl;
            }
        }
    };
```

これで、行列データが DPC++ バッファーにロードされ、ターゲットのデバイスにオフロードして、完了後にホストに戻すことができます。最後に、`mkl::blas::gemm` API がすべてのバッファー、サイズ、および転置操作で呼び出され、行列乗算カーネルとデータをターゲットのキューに追加します。

```
    // 例外ハンドラーがアタッチされた gpu デバイスで実行キューを作成
    sycl::queue my_queue(my_device, my_exception_handler);
    // デバイスとホスト間のオフロード向けに行列データの SYCL* バッファーを作成
    sycl::buffer<double, 1> A_buffer(A.data(), A.size());
    sycl::buffer<double, 1> B_buffer(B.data(), B.size());
    sycl::buffer<double, 1> C_buffer(C.data(), C.size());
    // add mkl::blas::gemm を実行キューに追加し、同期例外をキャッチ
    try {
      using oneapi::mkl::blas::gemm;
      using oneapi::mkl::transpose;
      gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha,
         A_buffer, ldA, B_buffer, ldB, beta, C_buffer, ldC);
    }
    catch (sycl::exception const& e) {
        std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    catch (std::exception const& e) {
        std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
            << e.what() << std::endl;
    }
```

gemm カーネルがキューに登録された後、実行されます。キューは、すべてのカーネルの実行を待機し、キャッチされた非同期例外を例外ハンドラーに渡してスローするように要求します。ランタイムは、ホストと GPU デバイス間のバッファーのデータ転送を処理します。C_buffer のアクセサーが作成されるまでに、バッファーのデータはホストマシンに暗黙的に転送されます。この場合、アクセサーは 2 x 2 の C_buffer のサブ行列を出力するために使用されます。

```
// C バッファーからデータをアクセスし、C 行列の一部を出力
auto C_accessor = C_buffer.template get_access<sycl::access::mode::read>();
std::cout << "\t" << C << " = [ " << C_accessor[0] << ", "
    << C_accessor[1] << ", ...]\n";
std::cout << "\t  [ " << C_accessor[1 * ldC + 0] << ", "
    << C_accessor[1 * ldC + 1] << ", ...]\n";
std::cout << "\t  [ " << "...]\n";
std::cout << std::endl;

return 0;
```

結果データは C_buffer オブジェクトにあり、明示的にどこかにコピーしない限り(元の C コンテナーに戻すなど)、C_buffer がスコープ外になるまでアクセサーを介してのみ利用できます。

5.3 インテル® oneAPI スレッディング・ビルディング・ブロック(インテル® oneTBB)

インテル® oneAPI スレッディング・ビルディング・ブロック(インテル® oneTBB)は、ホスト上でタスクベースの、共有メモリー並列プログラミングを可能にする、広く使用されている C++ ライブラリーです。このライブラリーは、SYCL* および ISO C++ で利用可能な機能のほかに、CPU 上での並列プログラミング向けに次のような機能を提供します。

- 汎用並列アルゴリズム
- コンカレント・コンテナー
- スケラーブル・メモリー・アロケーター
- ワークスチール・タスク・スケジューラー
- 低レベル同期プリミティブ

インテル® oneTBB はコンパイラーに依存せず、さまざまなプロセッサーとオペレーティング・システムで利用できます。CPU 向けのマルチスレッド並列処理を実現するため、ほかのインテル® oneAPI ライブラリー(インテル® oneAPI マス・カーネル・ライブラリー、インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリーなど)で使用されています。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、インテル® oneAPI スレッディング・ビルディング・ブロックのウェブサイトをご覧ください。oneTBB を oneAPI ベース・ツールキットの一部とし

て利用する場合、有償オプションとしてプライオリティー・サポートが得られることを考慮してください。インテルのコミュニティー・サポートについては、oneTBB フォーラムを参照してください。コミュニティーがサポートするオープンソース・バージョンについては、oneTBB GitHub* のページを参照してください。

5.3.1 インテル® oneTBB の使い方

インテル® oneTBB は、ほかの C++ コンパイラーでもインテル® oneAPI DPC++ コンパイラーと同じ方法で使用できます。詳細は、「oneTBB ドキュメント」(英語)をご覧ください。

現在、インテル® oneTBB はアクセラレーターを直接サポートしません。ただし、DPC++ 言語、OpenMP* オフロードおよびほかの インテル® oneAPI ライブラリーを組み合わせて、利用可能なすべてのハードウェア・リソースを効率良く使用するプログラムを構築できます。

5.3.2 インテル® oneTBB サンプルコード

2 つの基本的なインテル® oneTBB サンプルコードが、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneTBB(英語)から入手できます。これらのサンプルコードは、CPU と GPU 向けに記述されています。

- `tbb-async-sycl` は、インテル® oneTBB フローグラフ非同期ノードと機能ノードを使用し、計算カーネルを分割して CPU と GPU 間で実行する方法を示します。フローグラフ非同期ノードは、SYCL* を使用して機能ノードが計算の CPU 部分を実行する間に、GPU で実装された計算を実行します。
- `tbb-task-sycl` は、2 つのインテル® oneTBB タスクが同じ計算カーネルを実行する方法を示します。1 つのタスクが SYCL* コードを実行し、もう一方はインテル® oneTBB コードを実行します。
- `tbb-resumable-tasks-sycl` は、インテル® oneTBB 再開タスクと `parallel_for` を使用して、計算カーネルを分割して CPU と GPU で実行する方法を示します。再開可能なタスクは SYCL* を使用して GPU で計算を実装し、`parallel_for` は計算の CPU 部分を実行します。

5.4 インテル® oneAPI データ・アナリティクス・ライブラリー(インテル® oneDAL)

インテル® oneAPI データ・アナリティクス・ライブラリー(インテル® oneDAL)は、データ解析のすべてのステージ(前処理、変換、解析、モデリング、検証、意思決定)で、バッチ、オンライン、および分散処理モードで計算を実行する、高度に最適化されたアルゴリズムのビルディング・ブロックを提供することで、ビッグデータ解析の高速化を支援するライブラリーです。

アルゴリズムの計算だけでなくデータの取り込みを最適化し、スループットとスケーラビリティーを向上します。C++ および Java* API に加えて、Spark* や Hadoop* などの一般的なデータソースへのコネクターが含まれます。インテル® oneDAL の Python* ラッパーは、インテル® ディストリビューションの Python*(英語)に含まれます。

さらに従来の機能に加えて、インテル® oneDAL は従来の C++ インターフェイスに DPC++ API 拡張機能を提供し、一部のアルゴリズムで GPU も利用できるようにします。

このライブラリーは、分散計算では特に有用です。通信レイヤーから独立した分散アルゴリズムのビルディング・ブロックの完全なセットを提供します。これにより、ユーザーは利用したい通信基盤を使って、高速でスケーラブルな分散アプリケーションを構築できます。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、インテル® oneAPI データ・アナリティクス・ライブラリーの公式ウェブサイト(英語)をご覧ください。oneDAL を oneAPI ベース・ツールキットの一部として利用する場合、コミュニティー・サポート付きの無料のオープンソース・バージョンと、プレミアサポート付きの有料バージョンを利用できます。コミュニティーがサポートするオープンソース・バージョンについては、oneDAL GitHub* ページ(英語)を参照してください。

5.4.1 インテル® oneDAL の使い方

アプリケーションをビルドしてインテル® oneDAL とリンクするのに必要な依存関係に関する情報は、「インテル® oneAPI データ・アナリティクス・ライブラリーのシステム要件」(英語)を参照してください。

インテル® oneDAL ベースのアプリケーションは、適切なデバイスセレクターを選択することで、CPU または GPU でアルゴリズムをシームレスに実行できます。新機能により次のことが可能になります。

- 数値テーブルから DPC++ バッファーを抽出してカスタムカーネルに渡す。
- DPC++ バッファーから数値テーブルを作成する。

アルゴリズムは、DPC++ バッファーを再利用して GPU データを保持し、GPU と CPU 間でデータを繰り返しコピーする過負荷を排除するように最適化されています。

5.4.2 インテル® oneDAL サンプルコード

インテル® oneDAL のサンプルコードは GitHub* リポジトリーから入手できます。次のサンプルコードは、インテル® oneDAL 固有の機能を示す分かりやすい例です。
https://github.com/oneapi-src/oneDAL/tree/master/examples/oneapi/dpc/source/svm(英語)

5.5 インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー (インテル® oneCCL)

インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー(インテル® oneCCL)は、ディープラーニング(DL)、およびマシンラーニング(ML)ワークロード向けのスケーラブルでハイパフォーマンスな通信ライブラリーです。インテル® Machine Learning Scaling Library に由来するアイデアを発展させて、新しい機能と利用ケースを実現するため設計と API を拡張しています。

インテル® oneCCL は次の機能を備えています。

- 低レベルの通信ミドルウェア上に構築された MPI と libfabrics。
- 通信パフォーマンスに対する計算の生産的なトレードオフを可能にすることで、通信パターンのスケーラビリティーを促進する最適化。
- 優先順位、永続的な操作、アウトオブオーダー実行など、一連の DL 固有の最適化。
- CPU や GPU など各種ハードウェア・ターゲットで実行する DPC++ API。
- さまざまなインターコネクトで動作:インテル® Omni-Path アーキテクチャー(インテル® OPA)、InfiniBand*、イーサネット

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、インテル® oneAPI コレクティブ・コミュニケーション・ライブラリー公式ウェブサイト(英語)をご覧ください。コミュニティー・サポート付きの無料のオープンソース・バージョンと、プレミアサポート付きの有料バージョンを利用できます。コミュニティーがサポートするオープンソース・バージョンについては、oneCCL GitHub* ページ(英語)を参照してください。

5.5.1 インテル® oneCCL の使い方

MPI やインテル® oneAPI DPC++/C++ コンパイラーなど、ハードウェアとソフトウェアの依存関係に関する完全なリストについては、「インテル® oneAPI コレクティブ・コミュニケーション・ライブラリーのシステム要件」(英語)を参照してください。

SYCL* 対応 API は、インテル® oneCCL のオプション機能です。インテル® oneCCL ストリーム・オブジェクトを作成する場合、CPU と SYCL* バックエンドのどちらかを選択できます。

- CPU バックエンド:最初の引数に `ccl_stream_host` を指定します。
- SYCL* バックエンド:デバイスタイプに応じて `ccl_stream_cpu` または `ccl_stream_gpu` を指定します。

- SYCL* ストリームで動作する集合操作:
  - C API では、インテル® oneCCL は通信バッファーが `void*` にキャストされた `sycl::buffer` オブジェクトであると想定します。
  - C++ API では、インテル® oneCCL は通信バッファーが参照渡しされることを想定します。

使用法に関する詳しい説明は、https://oneapi-src.github.io/oneCCL/(英語)から入手できます。

5.5.2 インテル® oneCCL サンプルコード

インテル® oneCCL のサンプルコードは、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneCCL(英語)から入手できます。コードをビルドして実行する手順を含む入門サンプルは、同じ GitHub* リポジトリーから入手できます。

5.6 インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (インテル® oneDNN)

インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー(インテル® oneDNN)は、ディープラーニング・アプリケーション向けのオープンソースのパフォーマンス・ライブラリーです。このライブラリーには、インテル® アーキテクチャー・ベースのプロセッサーおよびインテル® プロセッサー・グラフィックス向けに最適化されたニューラル・ネットワークの基本的な構成要素が含まれます。インテル® oneDNN は、インテル® アーキテクチャー・ベースのプロセッサーとインテル® プロセッサー・グラフィックス上でアプリケーションのパフォーマンス向上に注目するディープラーニング・アプリケーションおよびフレームワーク開発者を対象としています。ディープラーニングでは、インテル® oneDNN を利用するアプリケーションを使用すべきです。

インテル® oneDNN は、インテル® oneAPI DL フレームワーク・デベロッパー・ツールキット、インテル® oneAPI ベース・ツールキットの一部として配布され、apt または yum チャネルから入手できます。

インテル® oneDNN は、C および C++ インターフェイス、OpenMP*、インテル® oneAPI スレッディング・ビルディング・ブロック、OpenCL* ランタイムなど、DNNL で現在サポートされる機能を引き続きサポートします。インテル® oneDNN は、oneAPI プログラミング・モデルの DPC++ API とランタイムサポートを導入します。

機能一覧、ドキュメント、サンプルコード、ダウンロードについては、インテル® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー公式ウェブサイト(英語)をご覧ください。コミュニティー・サポート付きの無料のオープンソース・バージョンと、プレミアサポート付きの有料バージョンを利用できます。コミュニティーがサポートするオープンソース・バージョンについては、oneDNN GitHub* ページ(英語)を参照してください。

5.6.1 インテル® oneDNN の使い方

インテル® oneDNN は、インテル® 64 アーキテクチャーまたは互換プロセッサーをベースにするシステムをサポートします。サポートされる CPU およびグラフィックス・ハードウェアのリストは、「インテル® oneAPI Deep Neural Network ライブラリーのシステム要件」を参照してください。

インテル® oneDNN は実行時に命令セット・アーキテクチャー(ISA)を検出し、オンライン生成機能を使用して、サポートされる最新の ISA 向けに最適化されたコードを展開します。

アプリケーションが使用する CPU または GPU ランタイム・ライブラリーとの相互運用性を保証するため、オペレーティング・システムごとにいくつかのパッケージが提供されます。

設定	依存関係
`cpu_dpcpp_gpu_dpcpp`	DPC++ ランタイム
`cpu_iomp`	インテルの OpenMP* ランタイム
`cpu_gomp`	GNU* OpenMP* ランタイム
`cpu_vcomp`	Microsoft Visual C++* OpenMP* ランタイム
`cpu_tbb`	インテル® oneAPI スレッディング・ビルディング・ブロック

パッケージには必要なライブラリーが含まれていないため、ビルド時にインテル® oneAPI ツールキットまたはサードパーティーのツールを使用してアプリケーションで依存関係を解決する必要があります。

DPC++ 環境では、インテル® oneDNN は DPC++ ランタイムを介して CPU または GPU ハードウェアと対話します。インテル® oneDNN は DPC++ を使用するほかのコードを使用することもできます。これを可能にするため、インテル® oneDNN は基盤となる SYCL* オブジェクトと相互作用する API 拡張機能を提供します。

そのような状況の 1 つとして、インテル® oneDNN が提供しないカスタム操作を行うため DPC++ カーネルを実行する場合が考えられます。その場合、インテル® oneDNN はカーネルをシームレスに送信するのに必要な API を提供し、同じデバイスキューを使用して実行コンテキストをインテル® oneDNN と共有します。

相互運用性を提供する API は、次の 2 つのシナリオを想定します。

- 既存の DPC++ オブジェクトをベースとするインテル® oneDNN オブジェクトの構築
- 既存のインテル® oneDNN オブジェクト向けの DPC++ オブジェクトへのアクセス

次の表にインテル® oneDNN オブジェクトと DPC++ オブジェクトのマッピングを示します。

インテル® oneDNN オブジェクト	DPC++ オブジェクト
エンジン	cl::sycl::device and cl::sycl::context
ストリーム	cl::sycl::queue
メモリー	cl::sycl::buffer<uint8_t, 1> または統合共有メモリー(USM)ポインター

注:内部的にライブラリー・メモリー・オブジェクトは、1D uint8_t SYCL* バッファーを使用しますが、異なる型の SYCL* バッファーを介してメモリーを初期化およびアクセスできます。この場合、バッファーは異なる型の `cl::sycl::buffer<uint8_t, 1>` に再解釈されます。

インテル® oneDNN オブジェクト	DPC++ オブジェクトからの再構成
エンジン	`dnnl::sycl_interop::make_engine(sycl_dev, sycl_ctx)`
ストリーム	`dnnl::sycl_interop::make_stream(engine, sycl_queue)`

メモリー	USM ベース:
	`dnnl::memory(memory_desc, engine, usm_ptr)`
	バッファーベース:
	`dnnl::sycl_interop::make_memory(memory_desc, engine, sycl_buf)`

インテル® oneDNN オブジェクト	DPC++ オブジェクトの抽出
エンジン	`dnnl::sycl_interop::get_device(engine)` `dnnl::sycl_interop::get_context(engine)`
ストリーム	`dnnl::sycl_interop::get_queue(stream)`
メモリー	USM ポインター:`dnnl::memory::get_data_handle()` バッファー:`dnnl::sycl_interop::get_buffer(memory)`

注:

- インテル® oneDNN でアプリケーションをビルドするには、コンパイラーが必要です。インテル® oneAPI DPC++/C++ コンパイラーは、インテル® oneAPI ベース・ツールキットに含まれています。

- SYCL* 相互運用性 API を有効にするには、`dnnl_sycl.hpp` をインクルードする必要があります。
- OpenMP* はランタイム・オブジェクトの受け渡しに依存しないため、インテル® oneDNN と連携する相互運用 API は必要ありません。

5.6.2 インテル® oneDNN サンプルコード

インテル® oneDNN のサンプルコードは、https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDNN(英語)から入手できます。導入向けのサンプルは新規ユーザーを対象としており、ビルドコマンドと実行コマンドの例を含む readme ファイルが含まれています。

5.7 インテル® oneAPI ビデオ・プロセシング・ライブラリー(インテル® oneVPL)

インテル® oneAPI ビデオ・プロセシング・ライブラリー(インテル® oneVPL)は、CPU、GPU、およびそのほかのアクセラレーターでポータブルなメディア・パイプラインを構築する、ビデオデコードとエンコード、および処理をおこなうプログラミング・インターフェイスです。インテル® oneVPL API は、インテル® ハードウェア・アクセラレーターを最大限に活用する、高品質で高いパフォーマンスのビデオ・アプリケーションを開発するために使用されます。メディア・セントリックのビデオ解析ワークロードにおけるデバイスの検出と選択、およびゼロコピーバッファー共有向けの API プリミティブを提供します。インテル® oneVPL は、インテル® メディア SDK と下位互換性があり、アーキテクチャー間の互換性も提供するため、ソースコードを変更することなく現在および将来のハードウェアで最適な実行が保証されます。

インテル® oneVPL はオープン仕様の API です。

一般的な情報は、oneVPL GitHub* ページ(英語)でご覧いただけます。完全な機能とドキュメントについては、インテル® oneAPI ビデオ・プロセシング・ライブラリー公式ウェブサイト(英語)で入手できます。コミュニティー・サポート付きの無料のオープンソース・バージョンと、プレミアサポート付きの有料バージョンを利用できます。

5.7.1 インテル® oneVPL の使い方

アプリケーションはインテル® oneVPL を使用して、ビデオのエンコードとデコード、および画像処理コンポーネントをプログラムできます。インテル® oneVPL はアクセラレーターを使用する前に、リファレンス設計として使用できるデフォルトの CPU 実装を提供します。

インテル® oneVPL アプリケーションは、次のプログラミング・モデルの基本的な手順に従います。

1. インテル® oneVPL ディスパッチャーは、実行時に利用可能なすべてのアクセラレーターを自動検索します。
   2. ディスパッチャーは、選択したアクセラレーター・コンテキストのセッションを初期化します。
   3. インテル® oneVPL は、セッションの開始時にビデオ・コンポーネントを構成します。
   4. インテル® oneVPL 処理ループが開始されます。処理ループは非同期で動作します。
   5. アプリケーションがインテル® oneVPL に作業メモリーを管理させることを選択した場合、メモリー割り当ては処理ループ内のビデオ呼び出しによって暗黙的に管理されます。
   6. 作業が完了すると、インテル® oneVPL はクリア呼び出しによりすべてのリソースをクリーンアップします。

インテル® oneVPL API は、クラシックな C スタイルのインターフェイスで定義されており、C++ と DPC++ とも互換性があります。

5.7.2 インテル® oneVPL サンプルコード

インテル® oneVPL には、インテル® oneVPL API の使い方を示す豊富なサンプルコードが用意されています。サンプルコードはリリースパッケージに含まれており、GitHub* リポジトリーの「oneAPI-samples」(英語)からも入手できます。

例えば、「hello-decode」(英語)サンプルでは、HEVC 入力ストリームの簡単なデコード操作とインテル® oneVPL プログラミング・モデルの基本的な手順を説明しています。

サンプルコードは次のような手順に分解できます。

注: 以下の例は最新のサンプルとは異なる場合があります。最新のバージョンについては、リリースパッケージまたはサンプルのリポジトリーを参照してください。

---

1. ディスパッチャーでインテル® oneVPL セッションを初期化します。

```
mfxLoader loader = NULL;
mfxConfig cfg = NULL;

loader = MFXLoad();

cfg = MFXCreateConfig(loader);
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_CODEC_HEVC;
sts = MFXSetConfigFilterProperty(cfg,
(mfxU8*)"mfxImplDescription.mfxDecoderDescription.decoder.CodecID",
ImplValue);

sts = MFXCreateSession(loader, 0, &session);
```

ここで、MFXCreateConfig() はディスパッチャーの内部構成を作成します。ディスパッチャーが構成されると、アプリケーションは MFXSetConfigFilterProperty() を使用して、コーデック ID とアクセラレーター・リファレンスなどの要件を設定します。アプリケーションが要件を設定するとセッションが作成されます。

2. デコードループを開始します。

```
while(is_stillgoing) {
    sts = MFXVideoDECODE_DecodeFrameAsync(session,
             (isdraining) ? NULL : &bitstream,
             NULL,
             &pmfxOutSurface,
             &syncp);
......
}
```

入力ストリームの準備が完了するとストリームには必要なコンテキストが用意され、デコードループはすぐに開始されます。

MFXVideoDECODE_DecodeFrameAsync() は、ビットストリームを第 2 パラメーターとして受け取ります。ビットストリームが NULL に到達すると、インテル® oneVPL は入力から残りのフレームを排出して操作を完了します。3 番目のパラメーターは作業メモリーです。例に示す NULL 入力はアプリケーションがインテル® oneVPL に作業メモリーの管理を要求することを意味します。

3. デコード呼び出しの結果を評価します。

```
while(is_stillgoing) {
    sts = MFXVideoDECODE_DecodeFrameAsync(...);

    switch(sts) {
        case MFX_ERR_MORE_DATA:
            ......
                ReadEncodedStream(bitstream, codec_id, source);
                ......
            }
            break;
        case MFX_ERR_NONE:
            do {
                sts = pmfxOutSurface->FrameInterface-
>Synchronize(pmfxOutSurface, WAIT_100_MILLSECONDS);
                if( MFX_ERR_NONE == sts ) {
                    sts = pmfxOutSurface->FrameInterface-
                    >Map(pmfxOutSurface, MFX_MAP_READ);

                    WriteRawFrame(pmfxOutSurface, sink);

                    sts = pmfxOutSurface->FrameInterface-
                    >Unmap(pmfxOutSurface);

                    sts = pmfxOutSurface->FrameInterface-
                    >Release(pmfxOutSurface);

                    framenum++;
                }
            } while( sts == MFX_WRN_IN_EXECUTION );
            break;

        default:
            break;
    }
```

`MFXVideoDECODE_DecodeFrameAsync()` の呼び出しごとに、アプリケーションはインテル® oneVPL が `MFX_ERR_NONE` で新しいストリームが完了するまで入力ビットストリームの読み取りを続行し、関数が正常に操作を完了したことを示します。新しいフレームごとにアプリケーションは出力メモリー(`surface`) の準備ができるまで待機し、フレームを出力および解放します。

`Map()` は、ディスクリート・グラフィックスのメモリー空間をホストメモリー空間にマップするのに使用されます。

4. 終了してクリーンアップを行います。

```
MFXUnload(loader);
free(bitstream.Data);
    fclose(sink);
    fclose(source);
```

最後に、`MFXUnload()` が呼び出されインテル® oneVPL からリソースが再利用されます。これは、インテル® oneVPL ライブラリーがリソースを再利用するためアプリケーションが行うべき唯一の呼び出しです。

注: この例では、インテル® oneVPL プログラミング・モデルの主な手順について説明しています。入力と出力ユーティリティー関数については説明していません。

5.8 その他のライブラリー

その他のライブラリーは、各種インテル® oneAPI ツールキットに含まれます。各ライブラリーの詳細については、それぞれのライブラリーの公式ドキュメントをご覧ください。

- インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)
- インテル® MPI ライブラリー
- インテル® オープン・ボリューム・カーネル・ライブラリー

ソフトウェア開発プロセス

6

oneAPI プログラミング・モデルを使用したソフトウェア開発プロセスは、標準の開発プロセスをベースにしています。プログラミング・モデルは、アクセラレーターを使用してパフォーマンスを向上するため、この章ではその作業に固有の手順を説明します。これには以下のものがあります。

- パフォーマンス・チューニング・サイクル
- コードのデバッグ
- ほかのアクセラレーター向けのコードの移行
- コードのコンポーザビリティー

6.1 SYCL* と DPC++ へのコードの移行

C++ または OpenCL* などほかのプログラミング言語で記述されたコードは、複数のデバイスで使用するため DPC++ コードへ移行できます。移行手順は、元の言語によって異なります。

6.1.1 C++ から SYCL* への移行

SYCL* は、C++ をベースとする「単一ソース」スタイルのプログラミング・モデルです。C++17 と C++20 の機能を基に構築され、ヘテロジニアス・プログラミングにおけるオープン、マルチベンダー、およびマルチ・アーキテクチャーのソリューションをサポートします。

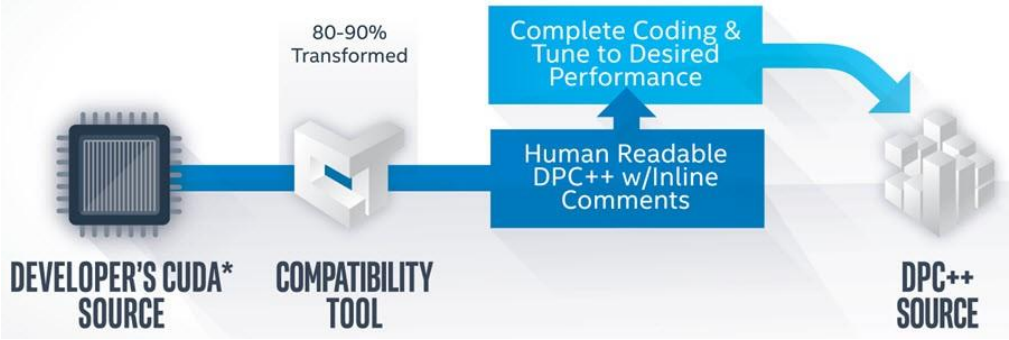
DPC++ コンパイラー・プロジェクトは、SYCL* を LLVM C++ コンパイラーに導入し、複数のベンダーとアーキテクチャーに対応したハイパフォーマンスな実装を実現しています。

既存の C++ アプリケーションを高速化する場合、大部分の C++ コードは変更する必要がないため、SYCL* はシームレスな統合を可能にします。デバイス側のコンパイルを可能にする SYCL* の構造については、「oneAPI プログラミング・モデル」をご覧ください。

6.1.2 DPC++ コンパイラーを使用した CUDA* から SYCL* への移行

インテル® DPC++ 互換性ツールは、インテル® oneAPI ベース・ツールキットに含まれます。このツールの目的は、NVIDIA CUDA* で記述された既存のプログラムから、DPC++ で記述されるプログラムへの移行を支援することです。このツールは、可能な限り DPC++ コードを自動生成し、インテル® oneAPI DPC++

コンパイラーで正しくコンパイルできるようにします。しかし、すべてのコードが自動的に移行されるとは限らず、手動での変更が必要な場合があります。このツールには、IDE プラグインヘルプと、DPC++ への移行を完了するための『ユーザーガイド』(英語)が含まれます。手動での変更が完了したら、DPC++ コンパイラーを使用して実行可能ファイルを作成します。



移行されたコード例、ツールのダウンロード手順などの説明は、インテル® DPC++ 互換性ツール・ウェブサイト(英語)でご覧いただけます。ツールの詳しい使用方法は、『インテル® DPC++ 互換性ツール・ユーザーガイド』(英語)を参照してください。

6.1.3 OpenCL* コードから SYCL* への移行

現在の DPC++ プロジェクトの SYCL* ランタイムは、並列処理を実現するため OpenCL* コードを採用しています。通常、SYCL* ではカーネルを実装するコードの行数と必要な API 関数やメソッドの呼び出しは少なくなります。ホストのソースコード行にデバイスのソースコードを埋め込むことで、OpenCL* プログラムを作成できます。

ほとんどの OpenCL* アプリケーション開発者は、デバイスへのカーネルオフロードに伴うセットアップ・コードの必要性を認識しているでしょう。SYCL* を使用すると、OpenCL* C コードに関連する大部分のセットアップ・コードなしで、シンプルで現代的な C++ ベースのアプリケーションを開発できます。これにより、習得の労力が軽減され、並列化の実装に集中できます。

さらに、OpenCL* アプリケーションの機能は、SYCL* API を介して引き続き利用できます。更新されたコードは、必要に応じて SYCL* インターフェイスを使用できます。

6.1.4 CPU、GPU、および FPGA 間の移行

DPC++ コンパイラーを使用した SYCL* では、プラットフォームは CPU、GPU、FPGA、またはその他のアクセラレーターやプロセッサーなどのデバイス(なくてもかまいません)に接続されたホストデバイスで構成されます。

プラットフォームに複数のデバイスが存在する場合、一部または大部分のワークをデバイスへオフロードするようにアプリケーションを設計します。oneAPI プログラミング・モデルで複数のデバイスにワークを分散するには、いくつかの方法があります。

1. デバイスセレクターの初期化 - SYCL* ではセレクターと呼ばれるクラスが提供され、プラットフォーム内のデバイスを手動で選択するか、oneAPI ランタイムのヒューリスティックがデバイスで利用可能な計算機能を判断してデフォルトデバイスを選択できるようになっています。
2. データセットの分割 - データに依存関係がない高い並列性を持つアプリケーションでは、データセットを明示的に分割して異なるデバイスで利用します。次のサンプルコードは、複数のデバイスにワークロードを分配する例です。`dpcpp snippet.cpp` コマンドでコードをコンパイルします。

```
int main() {
    int data[1024];
    for (int i = 0; i < 1024; i++)
        data[i] = i;
        try {
            cpu_selector cpuSelector;
            queue cpuQueue(cpuSelector);
            gpu_selector gpuSelector;
            queue gpuQueue(gpuSelector);
            buffer<int, 1> buf(data, range<1>(1024));
            cpuQueue.submit([&](handler& cgh) {
                auto ptr =
                buf.get_access<access::mode::read_write>(cgh);
                cgh.parallel_for<class divide>(range<1>(512),
                    [=](id<1> index) {
                    ptr[index] -= 1;
                });
            });
            gpuQueue.submit([&](handler& cgh1) {
                auto ptr =
                buf.get_access<access::mode::read_write>(cgh1);
                cgh1.parallel_for<class offset1>(range<1>(1024),
                    id<1>(512), [=](id<1> index) {
                        ptr[index] += 1;
                });
            });
            cpuQueue.wait();
            gpuQueue.wait();
        }
        catch (exception const& e) {
            std::cout <<
            "SYCL exception caught: " << e.what() << '\n';
            return 2;
        }
        return 0;
}
```

3. デバイス間で複数のカーネルをターゲットにする - アプリケーションに複数の独立したカーネルで並列処理可能なスコープがある場合、ターゲットデバイスごとに異なるキューを使用します。SYCL* がサポートされるプラットフォームとプラットフォームごとのデバイスのリストは、`get_platforms()` と `platform.get_devices()` を呼び出すことで取得できます。すべて

のデバイスが特定されたら、デバイスごとにキューを作成し、異なるカーネルを異なるキューに配布します。次のサンプルコードは、複数の SYCL* デバイスにカーネルを配布する方法を示します。

```
#include <stdio.h>
#include <vector>
#include <CL/sycl.hpp>
using namespace cl::sycl;
using namespace std;
int main()
{
        size_t N = 1024;
        vector<float> a(N, 10.0);
        vector<float> b(N, 10.0);
        vector<float> c_add(N, 0.0);
        vector<float> c_mul(N, 0.0);
    {
        buffer<float, 1> abuffer(a.data(), range<1>(N),
          { property::buffer::use_host_ptr() });
        buffer<float, 1> bbuffer(b.data(), range<1>(N),
          { property::buffer::use_host_ptr() });
        buffer<float, 1> c_addbuffer(c_add.data(), range<1>(N),
          { property::buffer::use_host_ptr() });
        buffer<float, 1> c_mulbuffer(c_mul.data(), range<1>(N),
          { property::buffer::use_host_ptr() });
    try {
               gpu_selector gpuSelector;
               auto queue = cl::sycl::queue(gpuSelector);
               queue.submit([&](cl::sycl::handler& cgh) {
                       auto a_acc = abuffer.template
                         get_access<access::mode::read>(cgh);
                       auto b_acc = bbuffer.template
                         get_access<access::mode::read>(cgh);
                       auto c_acc_add = c_addbuffer.template
                         get_access<access::mode::write>(cgh);
                       cgh.parallel_for<class VectorAdd>
                        (range<1>(N), [=](id<1> it) {
                            //int i = it.get_global();
                                c_acc_add[it] = a_acc[it] + b_acc[it];
                                      });
                                 });
               cpu_selector cpuSelector;
               auto queue1 = cl::sycl::queue(cpuSelector);
               queue1.submit([&](cl::sycl::handler& cgh) {
                       auto a_acc = abuffer.template
                           get_access<access::mode::read>(cgh);
                       auto b_acc = bbuffer.template
                           get_access<access::mode::read>(cgh);
                       auto c_acc_mul = c_mulbuffer.template
                           get_access<access::mode::write>(cgh);
                       cgh.parallel_for<class VectorMul>
                        (range<1>(N), [=](id<1> it) {
                              c_acc_mul[it] = a_acc[it] * b_acc[it];
                                      });
                                 });
               }
               catch (cl::sycl::exception e) {
/* In the case of an exception being throw, print the
error message and return 1. */
                       std::cout << e.what();
                       return 1;
               }
        }
```

```
        for (int i = 0; i < 8; i++) {
                std::cout << c_add[i] << std::endl;
                std::cout << c_mul[i] << std::endl;
        }
        return 0;
}
```

6.2 コンポーザビリティー

oneAPI プログラミング・モデルは、開発ツールチェーン全体をサポートするエコシステムを実現します。これには、CPU、GPU、FPGA など複数のアクセラレーターをサポートするコンパイラーとライブラリー、デバッガーと解析ツールが含まれます。

6.2.1 C/C++ OpenMP* および SYCL* のコンポーザビリティー

oneAPI プログラミング・モデルは、OpenMP* オフロードをサポートする LLVM/Clang ベースの統合コンパイラーを提供します。これにより、OpenMP* 構造を使用してホスト側のアプリケーションを並列化するか、ターゲットデバイスにオフロードする、シームレスな統合が可能になります。インテル® oneAPI ベース・ツールキットで利用可能なインテル® oneAPI DPC++/C++ コンパイラーと、インテル® oneAPI HPC ツールキットやインテル® oneAPI IoT ツールキットで利用可能なインテル® C/C++ コンパイラー・クラシックは、制限付きで OpenMP* と SYCL* の互換性を提供します。単一のアプリケーションでは、OpenMP* target 領域または SYCL* 構造を使用して、さまざまな関数やコードセグメントなどのコード領域の実行を利用可能なデバイスにオフロードできます。

OpenMP* と SYCL* オフロード構造は、別々のファイル、同じファイル、または同じ関数（制限付き）で使用できます。OpenMP* および SYCL* オフロードコードは、実行可能ファイル、静的ライブラリー、またはさまざまな組み合わせで一緒にバンドルできます。

注: SYCL* は CPU でデバイスコードを実行する場合、インテル® TBB のランタイムを使用します。そのため、CPU で OpenMP* と SYCL* の両方を使用すると、スレッドのオーバーサブスクリプションが発生する可能性があります。システムで実行されるワークロードのパフォーマンスを解析することで、この問題が生じているか確認することができます。

制限事項

同じアプリケーションで OpenMP* と SYCL* 構造を混在する場合、いくつかの考慮すべき制限があります。

- OpenMP* ディレクティブは、デバイスで実行される SYCL* カーネル内では使用できません。同様に、SYCL* コードは、OpenMP* target 領域内では使用できません。ただし、ホスト CPU で実行される OpenMP* コード内で SYCL* 構造を使用することはできます。
- プログラム内の OpenMP* および SYCL* デバイス領域は相互依存関係を持つことができません。例えば、デバイスコードの SYCL* 領域で定義された関数は、同じデバイスで実行される OpenMP* コードから呼び出すことはできません(その逆も同様)。OpenMP* デバイス領域と SYCL* デバイス領域は個別にリンクされ、個別のバイナリーとしてコンパイラーによって生成されるファットバイナリーを構成します。
- 現時点では、OpenMP* と SYCL* ランタイム・ライブラリー間での直接的な相互作用はサポートされていません。例えば、OpenMP* API で生成されたデバイス・メモリー・オブジェクトは、SYCL* コードからはアクセスできません。OpenMP* で生成されたデバイス・メモリー・オブジェクトを SYCL* コードで使用すると、未定義の動作となります。

例

次のコードは、SYCL* と OpenMP* オフロード構造を同じアプリケーションで使用する例を示します。

```
#include <CL/sycl.hpp>
#include <array>
#include <iostream>

float computePi(unsigned N) {
  float Pi;
#pragma omp target map(from : Pi)
#pragma omp parallel for reduction(+ : Pi)
  for (unsigned i = 0; i < N; ++i) {
    float T = (i + 0.5f) / N;
    Pi += 4.0f / (1.0 + T * T);
  }
  return Pi / N;
}

void iota(float *A, unsigned N) {
  cl::sycl::range<1> R(N);
  cl::sycl::buffer<float, 1> AB(A, R);
  cl::sycl::queue().submit([&](cl::sycl::handler &cgh) {
    auto AA = AB.template get_access<cl::sycl::access::mode::write>(cgh);
    cgh.parallel_for<class Iota>(R, [=](cl::sycl::id<1> i) {
      AA[i] = i;
    });
  });
}

int main() {
  std::array<float, 1024u> Vec;
  float Pi;

#pragma omp parallel sections
  {
#pragma omp section
    iota(Vec.data(), Vec.size());
#pragma omp section
    Pi = computePi(8192u);
  }

  std::cout << "Vec[512] = " << Vec[512] <<
  std::endl; std::cout << "Pi = " << Pi << std::endl;
  return 0;
}
```

サンプルコードをコンパイルするには次のコマンドを使用します。

```
icpx -fsycl -fiopenmp -fopenmp- targets=spir64 offloadOmp_dpcpp.cpp
```

説明:

- `-fsycl` オプションは DPC++ を有効にします。
- `-fiopenmp -fopenmp-targets=spir64` オプションは OpenMP* オフロードを有効にします。

次はサンプルコードからの出力例となります。

```
./a.out
Vec[512] = 512
Pi = 3.14159
```

注: コードに OpenMP* オフロードが含まれておらず、通常の OpenMP* コードのみが含まれる場合、`-fopenmp-targets` を省いた次のコマンドを使用します。
`icpx -fsycl -fiopenmp omp_dpcpp.cpp`

6.2.2 OpenCL* コードの相互運用性

oneAPI プログラミング・モデルにより、開発者は SYCL* API のさまざまな部分からすべての OpenCL* コードの機能を引き続き利用できます。SYCL* が提供する OpenCL* コードの相互運用性は、SYCL* が持つ高度なプログラミング・モデル・インターフェイスを利用しながら、従来の OpenCL* コードを再利用するのに役立ちます。この相互運用性モードには 2 つの主要機能があります。

1. OpenCL* コードのオブジェクトから SYCL* オブジェクトを生成する。例えば、SYCL* バッファーは、OpenCL* `cl_mem` から、または `cl_command_queue` の SYCL* キューから構築できます。
2. SYCL* オブジェクトから OpenCL* コードのオブジェクトを取得する。例えば、SYCL* アクセサーに関連付けられた暗黙の `cl_mem` を使用する OpenCL* カーネルを起動します。

6.3 DPC++ と OpenMP* オフロード処理のデバッグ

ホスト・プラットフォーム向けにコードを記述、デバッグ、および最適化する場合、コードを改善する手順はシンプルです。ビルド時に言語レベルのエラーに対処し、デバッガーで実行中のクラッシュや不正な結果をキャッチして原因を解決し、プロファイル・ツールを使用してパフォーマンスの問題を特定して修正します。

コードの一部分が DPC++ または OpenMP* オフロードにより別のデバイスで実行されるアプリケーションでは、コードの改善はかなり複雑になる可能性があります。

- DPC++ または OpenMP* オフロードの誤った使い方は、接続されたオフロードデバイスでプログラムが実行される際に、ジャストインタイム(JIT)コンパイルされるまで認識できないことがあります(この問題は、事前コンパイル(AOT)で確認できることがあります)。
- プログラムの論理的なエラーによるクラッシュは、ホスト、オフロードデバイス、または各種計算デバイスを連携させるソフトウェア・スタックで予期しない動作として現れる可能性があります。原因を究明するには以下が必要です。

- インテル® ディストリビューションの GDB などの標準デバッガーを使用して、ホスト上のコードで何が起こっているかデバッグします。
- デバイス固有のデバッガーを使用してオフロードデバイスの問題をデバッグします。ただし、デバイスのアーキテクチャー、計算スレッドを表現する規則、またはアセンブリーがホストとは異なることに注意してください。
- カーネルとデータがデバイスとのやり取りを行う場合にのみ中間ソフトウェア・スタックで現れる問題をデバッグするには、デバイスとホスト間の通信、および処理中に報告されるエラーを監視する必要があります。

- ホストとオフロードデバイスで発生する可能性がある一般的なパフォーマンスの問題に加えて、ホストとオフロードデバイスが連携するパターンは、アプリケーションのパフォーマンスに大きな影響を与えることがあります。これは、ホストとオフロードデバイス間の通信を監視する必要があるもう 1 つのケースです。

ここでは、オフロードプログラムの実行中に利用できる各種デバッグおよびパフォーマンス解析ツールとその用法について説明します。

6.3.1 oneAPI デバッグツール

次に示すツールは、SYCL* および OpenMP* オフロード処理のデバッグに役立ちます。

ツール	使用方法
環境変数	環境変数を使用して、プログラムを変更することなく、プログラムの実行時に OpenMP* および SYCL* ランタイムから診断情報を収集できます。
GPU 向けのプロファイル・ツールである onetrace ツール(GPU 向け PTI)	SYCL* および OpenMP* オフロードで oneAPI レベルゼロ・バックエンドを使用する場合、このツールを利用してバックエンドのエラーをデバッグし、ホストとデバイスの両方でパフォーマンスのプロファイルを実行できます。 関連資料: • onetrace ツールの GitHub*(英語) • GPU 向け PTI のGitHub*(英語)
OpenCL* アプリケーションのインターセプト・レイヤー	SYCL* および OpenMP* オフロードで OpenCL* バックエンドを使用する場合、このライブラリーを利用してバックエンドのエラーをデバッグし、ホストとデバイスの両方でパフォーマンスのプロファイルを実行できます。
インテル® ディストリビューションの GDB	通常ホストおよびデバイス(CPU、GPU、FPGA エミュレーション)で、論理的なバグを調査する際にアプリケーションのソースレベルのデバッグで使用されます。
インテル® Inspector	このツールは、オフロードの失敗につながる問題を含め、メモリーとスレッドの問題を検出してデバッグするのに役立ちます。 **注:** インテル® Inspector は、インテル® oneAPI HPC ツールキットまたはインテル® oneAPI IoT ツールキットに含まれます。
アプリケーション・デバッグ	これらのツールとランタイムベースのアプローチに加えて、開発者は別のアプローチから問題を見つけることもできます。次に例を示します。 • カーネルの出力と期待される出力を比較 • デバッグ用途の変数を作成して中間結果を確認 • カーネル内で結果をプリント **注:** SYCL* と OpenMP* では、どちらもオフロード領域内からの stdout への出力をサポートします。どの SIMD レーンまたはスレッドが出力しているか確認してください。

デバッグ環境変数

OpenMP* と SYCL* のオフロードランタイムは、ホストとオフロードデバイス間の通信を監視する環境変数を提供します。この環境変数を使用して、オフロード計算向けに選択されたランタイムを検出または制御できます。

OpenMP* オフロード環境変数

OpenMP* オフロードがどのように動作するかを理解し、バックエンドの制御に利用できるいくつかの環境変数が用意されています。

注: OpenMP* は、FPGA デバイスではサポートされません。

環境変数	説明
LIBOMPTARGET_DEBUG	この環境設定は、OpenMP* オフロードランタイムからのデバッグ出力を有効にします。以下が報告されます。 • 検出および使用された利用可能なランタイム(1、2) • 選択されたランタイムが開始および停止されたタイミング(1、2) • 使用されるオフロードデバイスの詳細(1、2) • ロードされたサポート・ライブラリー(1、2) • すべてのメモリー割り当てと解放のサイズとアドレス(1、2) • デバイス間とのデータコピーに関する情報、また統合共有メモリーではデバイスマッピング(1、2) • カーネルの起動と起動時の詳細情報(引数、SIMD 幅、グループ情報など)(1、2) • 呼び出されたレベルゼロ/OpenCL* API 関数(関数名、引数/パラメーター)(2) 値:(0、1、2) デフォルト:0
LIBOMPTARGET_PLUGIN_PROFILE	この環境変数は、オフロードされた OpenMP* コードのパフォーマンス・データの表示を有効にします。以下を表示します。 • 合計データ転送時間(リードとライト) • データ割り当て回数 • モジュールのビルド時間(ジャストイン・タイム・コンパイル) • 各カーネルの実行時間。 値: • `F` - 無効化 • `T` - ミリ秒単位のタイミングで有効化 • `T,usec`- マイクロ秒単位のタイミングで有効化 デフォルト:`F` 例:`export LIBOMPTARGET_PLUGIN_PROFILE=T,usec`
LIBOMPTARGET_PLUGIN	この環境変数を使用して OpenMP* オフロードの実行に使用するバックエンドを選択できます。 **注:** レベルゼロ・バックエンドは GPU デバイスでのみサポートされます。 値: • `LEVEL0` - レベルゼロ・バックエンドを使用します • `OPENCL` - OpenCL* バックエンドを使用します デフォルト: • GPU オフロードデバイス:`LEVEL0` • CPU または FPGA オフロードデバイス:`OPENCL`

SYCL* と DPC++ 環境変数

DPC++ プログラムは、すべての標準 SYCL* 環境変数をサポートします。すべての環境変数については GitHub*(英語)をご覧ください。デバッグで注目すべきは、次の SYCL* 環境変数と追加のレベルゼロ環境変数です。

環境変数	説明
SYCL_DEVICE_FILTER	この複雑な環境変数を使用すると、DPC++ ランタイムで使用されるランタイム、計算デバイス ID を利用可能なすべての組み合わせのサブセットに制御できます。 計算デバイス ID は、SYCL* API、clinfo、または sycl-ls(1 から始まる番号)によって返される ID に対応します。その ID を持つデバイスが特定のタイプであったり、特定のランタイムをサポートするかには関連性がありません。プログラムが特定のセレクター(gpu_selector など)を使用して、SYCL_DEVICE_FILTER のフィルターで除外されたデバイスを要求すると、例外がスローされます。 詳細については、GitHub* にある環境変数の説明をご覧ください。 https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md (英語) 次のような値を含みます。 • `opencl:cpu` - 利用可能なすべての CPU デバイスで OpenCL* ランタイムのみを使用します • `opencl:gpu` - 利用可能なすべての GPU デバイスで OpenCL* ランタイムのみを使用します • `opencl:gpu:2` - GPU である 3 番目のデバイスでのみ OpenCL* ランタイムのみを使用します • `level_zero:gpu:1` - GPU である 2 番目のデバイスでのみレベルゼロランタイムのみを使用します • `opencl:cpu,level_zero` - CPU デバイスでは OpenCL* ランタイムのみを使用し、サポートされる計算デバイスではレベルゼロランタイムを使用します デフォルト:利用可能なすべてのランタイムとデバイスを使用します
SYCL_PI_TRACE	この環境設定は、ランタイムからのデバッグ出力を有効にします。 値: • `1` – SYCL* プラグインとデバイスが検出され使用されたことを報告します • `2` - 引数と結果の値を含む SYCL* API 呼び出しを報告します • `-1` - 利用可能なすべてのトレースを報告します デフォルト:無効
ZE_DEBUG	この環境変数は、ランタイムが使用された際にレベルゼロ・バックエンドからのデバッグ出力を有効にします。以下が報告されます。 • レベルゼロ API の呼び出し • レベルゼロ・イベント情報 値:任意の値で定義された変数(有効) デフォルト:無効

サポート向けの診断情報を生成する環境変数

レベルゼロ・バックエンドは、動作を制御して診断を支援するいくつかの環境変数を提供します。

- レベルゼロ仕様、コア・プログラミング・ガイド：https://spec.oneapi.com/level-zero/latest/core/PROG.html#environment-variables（英語）
- レベルゼロ仕様、ツール・プログラミング・ガイド：https://spec.oneapi.com/level-zero/latest/tools/PROG.html#environment-variables（英語）

デバッグ情報の追加リソースは、実行時または事前コンパイル（AOT）時に、レベルゼロまたは OpenCL* バックエンド（OpenMP* オフロードと SYCL*/DPC++ ランタイムで使用される）によって呼び出されるインテル® グラフィックス・コンパイラーから提供されます。インテル® グラフィックス・コンパイラーは、ターゲットのオフロードデバイス向けの実行形式コードを生成します。環境変数の完全なリストは、https://github.com/intel/intel-graphics-compiler/blob/master/documentation/configuration_flags.md（英語）を参照してください。パフォーマンスの問題をデバッグする際に必要となるのは、次の 2 つです。

- `IGC_ShaderDumpEnable=1`（デフォルトは 0）によって、インテル® グラフィックス・コンパイラーが生成するすべての LLVM、アセンブリー、および ISA コードが `/tmp/IntelIGC/<application_name>` に書き込まれます。
- `IGC_DumpToCurrentDir=1`（デフォルトは 0）は、`IGC_ShaderDumpEnable` によって生成されるすべてのファイルを `/tmp/IntelIGC/<application_name>` に書き込みます。多数のファイルが生成される可能性があるため、一時ディレクトリーを作成することを推奨します。

インテル® oneAPI の異なるバージョン間で OpenMP* オフロードや SYCL* オフロード・アプリケーションのパフォーマンスの問題が生じる場合、異なるコンパイラー・オプションを使用したり、デバッガーを使用するなど、`IGC_ShaderDumpEnable` を有効にして結果ファイルを提供する必要があります。互換性の詳細については、「oneAPI ライブラリーの互換性」を参照してください。

オフロード・インターセプト・ツール

オフロード・ソフトウェア自身に組み込まれているデバッガーと診断情報に加えて、オフロード・パイプラインを介して送信される API 呼び出しとデータを監視するのは有用です。OpenCL* では、OpenCL* 呼び出しをインターセプトして報告するライブラリーを `LD_LIBRARY_PATH` に追加して、環境変数を設定しファイルへの診断情報生成を制御できます。レベルゼロでは、アプリケーションは onetrace ツールの引数として実行されます。onetrace の呼び出し方に応じて、レベルゼロ API 呼び出しのさまざまな側面をインターセプトして報告します。

OpenCL* アプリケーションのインターセプト・レイヤー

このライブラリーは、SYCL* または OpenMP* オフロードのバックエンドとして OpenCL* が使用される場合にデバッグおよびパフォーマンス・データを収集します。OpenCL* が SYCL* または OpenMP* オフロードのバックエンドである場合、このツールはバッファーのオーバーライト、メモリーリーク、ポインターの不一致を検出し、ランタイム・エラー・メッセージのより詳しい情報を提供します(CPU、FPGA、または GPU のいずれかが計算デバイスである場合にこれらの問題を診断できます)。OpenCL* バックエンドを使用するプログラムで onetrace を利用する場合や、レベルゼロ・バックエンドを使用するプログラムで OpenCL* アプリケーション・ライブラリーのインターセプト・レイヤーを利用する場合には、あまり役立たないことに注意してください。

追加のリソース:

- OpenCL* アプリケーションのインターセプト・レイヤーのビルドと使用法に関する各種情報は、https://github.com/intel/opencl-intercept-layer(英語)から入手できます。

  注: 最良の結果を得るには、次のフラグを使用して cmake を実行します。
  `-DENABLE_CLIPROF=TRUE - DENABLE_CLILOADER=TRUE`

- 同様のツール(`CLIntercept`)に関する情報は、https://github.com/gmeeker/clintercept(英語)および https://sourceforge.net/p/clintercept/wiki/Home/(英語)で入手できます。
- OpenCL* アプリケーションのインターセプト・レイヤーの制御に関連する情報は、https://github.com/intel/opencl-intercept-layer/blob/master/docs/controls.md(英語)にあります。
- GPU の最適化に関する情報は、『oneAPI GPU 最適化ガイド』から入手できます。

onetrace

OpenCL* アプリケーションのインターセプト・レイヤーと同様に、このツールは、レベルゼロが SYCL* または OpenMP* バックエンドである場合にデバッグおよびパフォーマンス・データを収集します。レベルゼロは、GPU で実行される計算のバックエンドとしてのみ利用できることに注意してください(現時点で、CPU と FPGA 向けのレベルゼロ・バックエンドはありません)。onetrace ツールは、https://github.com/intel/pti-gpu(英語)にある GPU 向けのプロファイル・ツール・インターフェイス(GPU 向け PTI)プロジェクトの一部です。OpenCL* バックエンドを使用するプログラムで onetrace を利用する場合や、レベルゼロ・バックエンドを使用するプログラムで OpenCL* アプリケーション・ライブラリーのインターセプト・レイヤーを利用する場合には、あまり役立たないことに注意してください。

onetrace ツールはソースとして配布されています。ツールのビルドに関する手順は、https://github.com/intel/pti-gpu/tree/master/tools/onetrace(英語)を参照してください。このツールは次の機能を提供します。

- 呼び出しのログ：このモードでは、すべての標準レベルゼロ（L0）API 呼び出しとその引数、およびタイムスタンプ付きの戻り値をトレースできます。これにより、ホストプログラムが接続された計算デバイスを利用する際に発生する障害の補足情報が得られます。
- ホストとデバイスのタイミング：すべての API 呼び出しの存続期間、カーネルの存続期間、およびアプリケーション全体の実行時間を提供します。
- デバイス・タイムライン・モード：各デバイス・アクティビティーのタイムスタンプを提供します。すべてのタイムスタンプは、同じ（CPU）時間スケールで示されます。
- ブラウザーの可視化：呼び出しトレースモードとデバイス・タイムライン・モードの結果をトレースファイル（JSON）に出力し、ブラウザーで視覚的に閲覧できます。

このデータはオフロードの障害やパフォーマンスの問題をデバッグするのに役立ちます。

追加のリソース：

- GPU 向けプロファイル・ツール・インターフェイス（GPU 向け PTI）GitHub* プロジェクト（英語）
- onetrace の GitHub* ページ（英語）

インテル® ディストリビューションの GDB

インテル® ディストリビューションの GDB は、プログラムの状態を調査および変更できるアプリケーション・デバッガーです。アプリケーションのホスト部分とデバイスにオフロードされたカーネルの両方を、同じデバッグセッションでシームレスにデバッグできます。このデバッガーは、CPU、GPU、および FPGA エミュレーション・デバイスをサポートします。主な機能を以下に示します。

- GPU デバイスに自動的に接続してデバッグイベントを監視します
- デバッグ用に JIT コンパイルされた、または動的にロードされたカーネルバイナリーを自動的に検出します
- プログラムの実行を停止するブレークポイントの設定（カーネル内部と外部の両方）
- スレッドのリスト表示、現在のスレッド・コンテキストへの切り替え
- アクティブな SIMD レーンのリスト表示、現在の SIMD レーンのコンテキストをスレッドごとに切り替え
- 複数のスレッドと SIMD レーン・コンテキストの式値の評価と出力
- レジスター値の調査と変更
- マシン命令の逆アセンブル
- 関数呼び出しスタックの表示とナビゲート
- ソースと命令レベルのステップ実行

- 非停止およびすべて停止のデバッグモード
- インテル® プロセッサー・トレースを使用した実行の記録(CPU のみ)

インテル® ディストリビューションの GDB の詳細とドキュメントへのリンクは、「インテル® ディストリビューションの GDB 導入ガイド(Linux* 版)」(英語)または「インテル® ディストリビューションの GDB 導入ガイド(Windows* 版)」(英語)をご覧ください。

オフロード向けインテル® Inspector

インテル® Inspector は、シリアルおよびマルチスレッド・アプリケーションを開発するユーザー向けの動的メモリーおよびスレッドエラーを検出するツールです。動的に生成されたオフロードコードだけでなく、アプリケーションのネイティブコードの正当性を検証するのに利用できます。

前述のツールや手法とは異なり、インテル® Inspector は GPU または FPGA と通信するオフロードコードのエラーを検出することはできません。インテル® Inspector では、CPU をターゲットとしてカーネルを実行するように SYCL* または OpenMP* ランタイムを設定する必要があります。解析を実行する前に、次の環境変数を設定する必要があります。

- CPU デバイスでカーネルを実行するように SYCL* アプリケーションを設定します。

```
Export SYCL_DEVICE_FILTER=opencl:cpu
```

- CPU デバイスでカーネルを実行するように OpenMP* アプリケーションを設定します。

```
export OMP_TARGET_OFFLOAD=MANDATORY
export LIBOMPTARGET_DEVICETYPE=cpu
```

- JIT コンパイラーまたはランタイムでコード解析とトレース有効にします。

```
export CL_CONFIG_USE_VTUNE=True
export CL_CONFIG_USE_VECTORIZER=false
```

次のいずれかのコマンドを使用して、コマンドラインから解析を開始します。インテル® Inspector のグラフィカル・ユーザー・インターフェイスからも開始できます。

- メモリー:inspxe-cl -c mi3 -- <app> [app_args]
- スレッド化:inspxe-cl -c ti3 -- <app> [app_args]

次のコマンドを使用して解析結果を表示します:inspxe-cl -report=problems -report-all

SYCL* や OpenMP* オフロードプログラムが OpenCL* バックエンドに不正なポインターを渡したり、誤ったスレッドからバックエンドに不正なポインターを渡す場合、インテル® Inspector は問題を警告します。これにより、インターセプト・レイヤーやデバッガーを使用して問題を見つけるよりも容易な場合があります。

「インテル® Inspector ユーザーガイド(Linux* 版)」(英語)またはインテル® Inspector ユーザーガイド(Windows* 版)」(英語)で詳細を参照できます。

6.3.2 オフロード処理のトレース

GPU に計算をオフロードするプログラムが開始される場合、プログラムの実行に関連する多くのコンポーネントが動作します。マシン非依存のコードをマシン依存のコードにコンパイルし、データとバイナリーをデバイスにコピーして、結果を戻す必要があります。ここでは、「oneAPI デバッグツール」で説明したツールを使用して、すべてのアクティビティーをトレースする方法を示します。

カーネル設定時間

オフロードコードをデバイスで実行する前に、マシン非依存のカーネルをターゲットデバイス向けにコンパイルしてコードをデバイスにコピーする必要があります。このカーネルのセットアップ時間が考慮されていないと、ベンチマークを複雑にしたり歪めたりする可能性があります。ジャストイン・タイム・コンパイルは、オフロード・アプリケーションのデバッグに遅延を引き起こすことがあります。

OpenMP* オフロードプログラムでは、`LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` を設定すると、オフロードコード "ModuleBuild" のビルドに必要な時間が報告されます。これをプログラムの実行時間全体と比較します。

SYCL* オフロードプログラムでは、カーネルのセットアップ時間を判断するのはさらに困難になります。

- レベルゼロがバックエンドである場合、onetrace によって返されるデバイスのタイミングとタイムラインからカーネルのセットアップ時間を求められます。
- OpenCL* がバックエンドの場合、OpenCL* アプリケーションのインターセプト・レイヤーを使用して情報を取得する際に、BuildLogging、KernelInfoLogging、CallLogging、CallLoggingElapsedTime、KernelInfoLogging、HostPerformanceTiming、HostPerformanceTimeLogging、ChromeCallLogging、または CallLoggingElapsedTime フラグを設定できます。

この方法により、`LIBOMPTARGET_PLUGIN_PROFILE=T` によって戻される情報を補足できます。

バッファーの作成、サイズ、およびコピーの監視

バッファーが作成された時期、作成されたバッファーの数、およびそれらが再利用されるか、また常に作成および破棄されるかを知ることは、オフロード・アプリケーションのパフォーマンスを最適化する上での重要な鍵となります。しかし、それらは OpenMP* や SYCL* などの高レベル・プログラミング言語を使用する場合は、必ずしも明確でない可能性がありユーザーからバッファーの管理は隠匿されます。

高レベルでは、プログラムの実行時に `LIBOMPTARGET_DEBUG` および `SYCL_PI_TRACE` 環境変数を使用して、バッファーに関連するアクティビティーを追跡できます。`LIBOMPTARGET_DEBUG` は、

SYCL_PI_TRACE よりも多くの情報を提供します(作成されたバッファーアドレスとサイズをレポート)。SYCL_PI_TRACE は、API 呼び出しをレポートするだけであり、バッファーアドレスやサイズに関する情報は含まれません。

低レベルでは、レベルゼロのバックエンドを使用する場合、onetrace の呼び出しログモードは引数を含むすべてのレベルゼロ API 呼び出しに関する情報を提供します。例えば、バッファー作成呼び出し(zeMemAllocDevice など)で、デバイスとの間で渡される結果バッファーのサイズを取得できるため役立つことがあります。onetrace は、デバイス・タイムライン・モードですべてのレベルゼロデバイス間とのアクティビティー(メモリー転送を含む)をダンプすることができます。アクティビティーごとに、追加(コマンドリストへ)、送信(キューへ)、開始と終了時間を取得できます。

OpenCL* がバックエンドの場合、OpenCL* アプリケーションのインターセプト・レイヤーを使用する際に、CallLogging、CallLoggingElapsedTime、および ChromeCallLogging フラグを設定して同様の情報を取得できます。

合計転送時間

合計データ転送時間をカーネル実行時間と比較することは、接続されているデバイスへの計算のオフロードが有益であるか判断することが重要になることがあります。

OpenMP* オフロードプログラムでは、LIBOMPTARGET_PLUGIN_PROFILE=T[,usec] を設定すると、ビルド("DataAlloc")、オフロードデバイスの読み取り("DataRead")、および書き込み("DataWrite")に必要な時間がレポートされます(ただし合計として)。

SYCL* プログラムのデータ転送時間を判断するのはさらに困難になります。

- レベルゼロがバックエンドである場合、onetrace によって返されるデバイスのタイミングとタイムラインからデータ転送時間を求められます。
- OpenCL* がバックエンドの場合、OpenCL* アプリケーションのインターセプト・レイヤーを使用して情報を取得する際に、BuildLogging、KernelInfoLogging、CallLogging、CallLoggingElapsedTime、KernelInfoLogging、HostPerformanceTiming、HostPerformanceTimeLogging、ChromeCallLogging、または CallLoggingElapsedTime フラグを設定できます。

カーネル実行時間

OpenMP* オフロードプログラムでは、LIBOMPTARGET_PLUGIN_PROFILE=T[,usec] を設定すると、オフロードされたすべてのカーネル("Kernel#...")の合計実行時間が報告されます。

SYCL* オフロードプログラム向け:

- レベルゼロ・バックエンドでは、onetrace のデバイス・タイミング・モードを使用してすべてのカーネルのデバイスでの実行時間を取得できます。

- OpenCL* がバックエンドの場合、OpenCL* アプリケーションのインターセプト・レイヤーを使用して情報を取得する際に、CallLoggingElapsedTime、DevicePerformanceTiming、DevicePerformanceTimeKernelInfoTracking、DevicePerformanceTimeLWSTracking、DevicePerformanceTimeGWSTracking、ChromePerformanceTiming、ChromePerformanceTimingInStages フラグを設定できます。

デバイスカーネルが呼び出され、スレッドが生成される場合

場合によってには、オフロードカーネルが作成され、実行を開始するかなり前にデバイスへ転送されることがあります(通常は、カーネルの実行に必要なすべてのデータと制御が転送された後にのみ)。

インテル® ディストリビューションの GDB を使用してデバイスカーネルにブレークポイントを設定できます。そして、カーネル引数の照会、スレッドの生成と破棄の監視、現在のスレッドとコード内の位置のリスト表示("`info thread`" を使用)などを行えます。

6.3.3 オフロード処理のデバッグ

異なるランタイムまたは計算デバイスで実行

オフロードプログラムが正常に実行されなかったか、生成された結果が正しくない場合、比較的容易な正当性の確認方法は、OpenMP* アプリケーションでは `LIBOMPTARGET_PLUGIN` と `OMP_TARGET_OFFLOAD` 環境変数を、また SYCL* アプリケーションでは `SYCL_DEVICE_FILTER` 環境変数を使用して、別のランタイム(OpenCL* とレベルゼロ)または計算デバイス(CPU と GPU)でアプリケーションを実行することです。異なるランタイム間で再現されるエラーは、ほとんどの場合、ランタイムの問題として排除できます。そしてデバイス間で再現されるエラーの大部分は、不良ハードウェアの問題を排除できます。

CPU 実行をデバッグ

オフロードコードを CPU で実行するには、"ホスト" 実装と "OpenCL* の CPU バージョン" という 2 つのオプションがあります。"ホスト" 実装は、オフロードコードのネイティブ実装であり、オフロードされないコードと同じようにデバッグできます。"OpenCL* の CPU バージョン" は、OpenCL* ランタイムとコード生成プロセスを通過しますが、最終的にインテル® TBB ランタイムで実行される通常の並列コードになります。繰り返しますが、これにより慣れ親しんだアセンブリーと並列処理メカニズムのデバッグ環境が提供されます。ポインターはスタック全体をアクセスでき、データを直接参照できます。また、オペレーティング・システム・プロセスの通常の上限を超えるメモリー制限はありません。

CPU オフロード実行のエラーを検出して修正すると、GPU オフロード実行で発生するエラーよりもはるかに少ない労力でエラーを解決でき、GPU や他のアクセラレーターが接続されたシステムを利用する必要もなくなります。

OpenMP* アプリケーションで "ホスト" 実装を適用するには、"target" または "device" 構造を削除して、通常のホスト OpenMP* コードに置き換えます。`LIBOMPTARGET_PLUGIN=OPENCL` が設定され、GPU オフロードが無効化されると、オフロードコードは OpenMP* ランタイムで実行され、インテル® TBB が並列処理を行います。

SYCL* アプリケーションで `SYCL_DEVICE_FILTER=host` を設定すると "ホスト" デバイスはシングルスレッドで実行を行います。これは、データ競合やデッドロックなどスレッド化の問題が実行エラーの原因であるか判断するのに役立ちます。`SYCL_DEVICE_FILTER=opencl:cpu` に設定すると、CPU の OpenCL* ランタイムが使用され、インテル® TBB が並列処理を行います。

インテル® ディストリビューションの GDB を使用して GPU 実行をデバッグ

インテル® ディストリビューションの GDB については、「インテル® ディストリビューションの GDB 導入ガイド (Linux* 版)」(英語) または「インテル® ディストリビューションの GDB 導入ガイド (Windows* 版)」(英語) に詳しく記載されています。ただし、GDB を使用して GPU アプリケーションをデバッグする方法は、ホストでの手順とは若干異なるため (一部のコマンドの使い方が異なり、見慣れない出力が表示されることがあります)、それらの違いの一部をここで紹介します。

「インテル® ディストリビューションの GDB を使用したデバッグのチュートリアル (Linux* 版)」(英語) では、SYCL* プログラムのデバッグセッションを開始し、カーネル内にブレークポイントを設定し、プログラムを実行して GPU にオフロードしローカル値を出力し、スレッドの SIMD レーン 5 を切り替えて変数を再度出力するサンプルのデバッグセッションを紹介しています。

通常の GDB と同様に、`command <CMD>` には GDB の `help <CMD>` コマンドを使用して、`<CMD>` の情報テキストを読み取ります。次に例を示します。

```
(gdb) help info threads
Display currently known threads.
Usage: info threads [OPTION]... [ID]...
If ID is given, it is a space-separated list of IDs of threads to display.
Otherwise, all threads are displayed.

Options:
   -gid
     Show global thread IDs.
```

GDB でインフェリアー、スレッド、および SIMD レーンの参照

アプリケーションのスレッドは、デバッガーによって一覧表示できます。表示される情報には、スレッド ID とスレッドが停止している位置が含まれます。GPU スレッドの場合、デバッガーはアクティブな SIMD レーンも表示します。

上記の例では、GDB の "`info threads`" コマンドでスレッドを表示していますが、見慣れない形式で情報が示されることがあります。

```
  Id              Target Id              Frame
  1.1             Thread <id omitted>    <frame omitted>
  1.2             Thread <id omitted>    <frame omitted>
  2.1             Thread 1610612736      (inactive)
* 2.2:1           Thread 1073741824      <frame> at array-transform.cpp:61
  2.2:[3 5 7]     Thread 1073741824      <frame> at array-transform.cpp:61
  2.3:[1 3 5 7]   Thread 1073741888      <frame> at array-transform.cpp:61
  2.4:[1 3 5 7]   Thread 1073742080      <frame> at array-transform.cpp:61
```

GDB は次の形式でスレッドを表示します。

<インフェリアー番号>.<スレッド番号>:<SIMD レーン/s>

例えば、スレッド ID:"2.3:[1 3 5 7]" は、インフェリアー 2 で実行されるスレッド 3 の SIMD レーン1、3、5 および 7 を意味します。

GDB 用語の "inferior(インフェリアー)" は、デバッグされるプロセスを指します。GPU にオフロードを行うプログラムのデバッグセッションには、通常 2 つのインフェリアーがあります。プログラムのホストを示す 1 つの "ネイティブ" インフェリアー(上記のインフェリアー 1)と、GPU デバイスを示すもう 1つの "リモート" インフェリアー(上記のインフェリアー 2)です。インテル® ディストリビューションの GDB は自動的に GPU インフェリアーを生成するため、特に操作は必要ありません。

式の値を出力すると、式は現在のスレッドの現在の SIMD レーンのコンテキストで評価されます。"thread 3:4 "、"thread :6 "、または "thread 7 " などの "thread" コマンドを使用して、スレッドと SIMD レーンを切り替えることができます。最初のコマンドは、スレッド3 と SIMD レーン 4 に切り替えます。2 番目のコマンドは、現在のスレッドで SIMD レーン 6 に切り替えます。3 番目のコマンドは、スレッド 7 に切り替えます。選択されるデフォルトレーンは、以前に選択したレーン(アクティブであれば)、またはスレッド内で最初にアクティブになったレーンのどちらかになります。

"thread apply コマンド" は、同様に広域または集中的である可能性があります(これにより、変数を調査するコマンドからの出力を制限しやすくなります)。SIMD レーンのデバッグの詳細と例については、「インテル® ディストリビューションの GDB を使用したデバッグのチュートリアル(Linux* 版)」(英語)を参照してください。

GDB のスレッドとインフェリアーに関する詳細は、以下をご覧ください。
https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html(英語)
https://sourceware.org/gdb/current/onlinedocs/gdb/Inferiors-Connections-and-Programs.html#Inferiors-Connections-and-Programs(英語)

スケジューラーの制御

デフォルトでは、スレッドがブレークポイントに到達すると、デバッガーはブレーク・ポイント・ヒット・イベントをユーザーに通知する前にすべてのスレッドを停止します。これは GDB のすべて停止モードです。非停止モードでは、他のスレッドが実行される間、スレッドの停止イベントが表示されます。

すべて停止モードでは、スレッドが再開されると(例:`continue` コマンドで通常のように再開する、または `next` コマンドでステップ実行する場合)、他のすべてのスレッドも再開されます。スレッド化されたアプリケーションで複数のブレークポイントが設定されていると、ブレークポイントに到達した次のスレッドが続くスレッドではない可能性があるため、混乱を招く可能性があります。

`set scheduler-locking` コマンドを使用することで、現在のスレッドが再開された時に他のスレッドが再開されないように制御することができます。これは、現在のスレッドのみが命令を実行している時に、他のスレッドの介入を避けるのに有効です。`help set scheduler-locking` と入力すると利用可能なオプションが表示されます。詳細は、https://sourceware.org/gdb/current/onlinedocs/gdb/Thread-Stops.html(英語)をご覧ください。SIMD レーンは個別に再開できないことに注意してください。これは、ベースとなるスレッドと共に再開されます。

非停止モードのデフォルトでは、現在のスレッドのみが再開されます。すべてのスレッドを再開するには、`continue` コマンドで "`-a`" フラグを指定します。

1 つ以上のスレッド/レーンの情報をダンプ(Thread Apply)

プログラム状態を調査するコマンドは、通常、現在のスレッドの現在の SIMD レーンのコンテキストに適用されます。複数のコンテキストの値を調査することが必要なこともあります。そのような場合、`thread apply` コマンドを使用します。例えば、以下はスレッド 2,5 の SIMD レーン 3-5 に対して `print element` コマンドを実行します。

```
(gdb) thread apply 2.5:3-5 print element
```

同様に、以下は、現在のスレッドの SIMD レーン 3、5、および 6 のコンテキストに対し `print element` コマンドを実行します。

```
(gdb) thread apply :3 :5 :6 print element
```

ブレークポイント停止後の GPU コードのステップ実行

GPU にオフロードされたカーネル内で停止するには、カーネル内のソース行にブレークポイントを設定するだけです。GPU スレッドがそのソース行に到達すると、デバッガーは実行を停止してブレークポイントの到達を示します。ソース行単位でスレッドをステップ実行するには、`step` または `next` コマンドを使用します。`step` コマンドは関数にステップインし、`next` コマンドは関数呼び出しをステップオーバーします。ステップ実行する前に、他のスレッドの介入を避けるため `set scheduler-locking step` を設定することを推奨します。

インテル® ディストリビューションの GDB で使用する DPC++ 実行形式をビルド

ホスト・アプリケーションのデバッグと同様に、GPU でデバッグ可能なバイナリーを作成するには、いくつかの追加フラグを指定する必要があります。詳細については、「インテル® ディストリビューションの GDB 導入ガイド(Linux* 版)」(英語)をご覧ください。

JIT コンパイルを行う際にスムーズなデバッグを可能にするには、-g フラグを指定してコンパイラーのデバッグ情報生成を有効にし、アプリケーションのホストと JIT コンパイルカーネルの両方で -O0 フラグを指定して最適化を無効にします。カーネルのフラグはリンク時に適用されます。次に例を示します。

- 次のコマンドでプログラムをコンパイル:`dpcpp -g -O0 -c myprogram.cpp`
- 次のコマンドでプログラムをリンク:`dpcpp -g -O0 myprogram.o`

Cmake を使用してプログラムをビルドする場合、CMAKE_BUILD_TYPE の Debug タイプを使用し、CMAKE_CXX_FLAGS_DEBUG 変数に -O0 を追加します。次に例を示します。

```
set (CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0")
```

デバッグ向けにビルドされたアプリケーションは、通常の「リリース」ビルドで最適化されたアプリケーションよりも起動に時間がかかる場合があります。そのため、デバッガーで起動すると、プログラムの実行速度が遅くなったように感じられることがあります。これにより問題が生じる場合、大規模なアプリケーションの開発者は、プログラムの実行時ではなくビルド時に JIT オフロードコードを事前(AOT)コンパイルすることを推奨します(これには、-g -O0 を使用するとビルドに時間がかかる場合があります)。詳細については、「コンパイル手順の概要」をご覧ください。

GPU 向けの事前コンパイルを行う場合、ターゲットデバイスに対応したデバイスタイプを指定する必要があります。次のコマンドを使用して、現在のマシンで利用可能な GPU デバイスオプションを確認できます。

ocloc compile --help

さらに、カーネルのデバッグモードを有効にします。次の AOT コンパイルの例は、KBL デバイスをターゲットにしています。

```
dpcpp -g -O0 -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
-Xs "-device kbl -internal_options -cl-kernel-debug-enable -
options -cl-opt-disable" myprogram.cpp
```

インテル® ディストリビューションの GDB で使用する OpenMP* 実行形式をビルド

プログラムのコンパイルとリンクに -g -O0 フラグを使用します。次に例を示します。

```
icpx -fiopenmp -O0 -fopenmp-targets=spir64 -c
-g myprogram.cpp icpx -fiopenmp -O0 -fopenmp-
```

次の環境変数を設定して最適化を無効にし、カーネルのデバッグ情報を有効にします。

```
export LIBOMPTARGET_OPENCL_COMPILATION_OPTIONS="-g -
cl-opt-disable" export
```

事前(AOT)コンパイルは現在 OpenMP* をサポートしていません。

オフロード・インターセプト・ツールを使用して GPU 実行をデバッグ

オフロードプログラムの一般的な問題は、実行に失敗することがありますが、追加情報をほとんど持たない OpenCL* エラーが生成されることです。OpenCL* アプリケーションのインターセプト・レイヤーと onetrace を使用して、このエラーに関する詳細情報を取得できます。これは、開発者が問題の原因を特定するのに役立ちます。

OpenCL* アプリケーションのインターセプト・レイヤー

このライブラリーを使用する場合、`Buildlogging`、`ErrorLogging`、および `USMChecking=1` オプションを使用してエラーの原因を特定できます。

1. 次のテキストを含む `clintercept.conf` ファイルをホーム・ディレクトリーに作成します。

```
SimpleDumpProgramSource=1
CallLogging=1
LogToFile=1
//KernelNameHashTracking=1
BuildLogging=1 ErrorLogging=1
USMChecking=1
//ContextCallbackLogging=1
// Profiling knobs
KernelInfoLogging=1
DevicePerformanceTimi
ng=1
DevicePerformanceTime
LWSTracking=1
```

2. 次のように `cliloader` を使用してアプリケーションを実行します。

```
<OCL_Intercept_Install_Dir>/bin/cliloader/cliloader -d ./<app_name> <app_args>
```

3. ~CLIntercept_Dump/<app_name> ディレクトリーで次の結果を確認します。

   - `clintercept_report.txt`:プロファイルの結果
   - `clintercept_log.txt`:OpenCL* の問題をデバッグする際に使用される OpenCL* 呼び出しログ

次のテキストは、CL_INVALID_ARG_VALUE (-50) ランタイムエラーが発生したプログラムで生成されたログファイルの例の一部です。

```
...
<<<< clSetKernelArgMemPointerINTEL -> CL_SUCCESS
>>>>
clGetKernelInfo( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_602
57cS2_S2_S2_S2_S2_S2
_S2_S2_fS2_S2_S2_S2_iENKUlRN2cl4sycl7handlerEE197-
>45clES6_EUlNS4_7nd_itemILi3EEEE225->13 ): param_name = CL_KERNEL_CONTEXT
(1193)
<<<< clGetKernelInfo -> CL_SUCCESS
>>>>
clSetKernelArgMemPointerINTEL( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e35
16dpct_type_60257cS2_
S2_S2_S2_S2_S2_S2_S2_fS2_S2_S2_S2_iENKUlRN2cl4sycl7handlerEE197-
>45clES6_EUlNS4_7nd_itemILi3EEEE225->13 ): kernel = 0xa2d51a0, index =
3, value = 0x41995e0 mem pointer 0x41995e0 is an UNKNOWN pointer and no
device support shared system pointers! ERROR!
clSetKernelArgMemPointerINTEL returned CL_INVALID_ARG_VALUE (-50)
<<<< clSetKernelArgMemPointerINTEL -> CL_INVALID_ARG_VALUE
```

この例は、次の値がエラーのデバッグに役立ちます。

- ZTSZZ10outer_coreiP5mesh
- index = 3, value = 0x41995e0

このデータによりどのカーネルに問題があるか、またどの引数に問題があるかが分かり、その理由を特定できます。

onetrace

OpenCL* アプリケーションのインターセプト・レイヤーと同様に、onetrace ツールはレベルゼロのランタイムエラーの原因を検出するのに役立ちます。

onetrace を使用するには次の操作を行います。

1. 呼び出しログモードでアプリケーションを実行します。ツールの出力をファイルにリダイレクトすることを推奨します。

```
./onetrace -c ./<app_name> <app_args> [2> log.txt]
```

2. 呼び出しトレースを確認してエラーを特定します(log.txt)。次に例を示します。

```
>>>> [102032049] zeKernelCreate: hModule = 0x55a68c762690 desc =
0x7fff865b5570 {29 0 0 GEMM} phKernel = 0x7fff865b5438 (hKernel = 0)
<<<< [102060428] zeKernelCreate [28379 ns] hKernel = 0x55a68c790280 ->
ZE_RESULT_SUCCESS (0)
…
>>>> [102249951] zeKernelSetGroupSize: hKernel = 0x55a68c790280
groupSizeX = 256 groupSizeY = 1 groupSizeZ = 1
<<<< [102264632] zeKernelSetGroupSize [14681 ns] -> ZE_RESULT_SUCCESS
(0)
>>>> [102278558] zeKernelSetArgumentValue: hKernel = 0x55a68c790280
argIndex = 0 argSize = 8 pArgValue = 0x7fff865b5440
<<<< [102294960] zeKernelSetArgumentValue [16402 ns] ->
ZE_RESULT_SUCCESS (0)
>>>> [102308273] zeKernelSetArgumentValue: hKernel = 0x55a68c790280
argIndex = 1 argSize = 8 pArgValue = 0x7fff865b5458
<<<< [102321981] zeKernelSetArgumentValue [13708 ns] ->
ZE_RESULT_ERROR_INVALID_ARGUMENT (2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600
argIndex = 2 argSize = 8 pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] ->
ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600
argIndex = 3 argSize = 4 pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] ->
ZE_RESULT_SUCCESS (0)
...
```

この例のログには次のデータが示されています。

- 問題の原因となるレベルゼロ API 呼び出し(`zeKernelSetArgumentValue`)
- 問題の原因(`ZE_RESULT_ERROR_INVALID_ARGUMENT`)
- 引数インデックス(`argIndex = 1`)
- 不正な値の場所(`pArgValue = 0x7fff865b5458`)
- カーネルハンドル(`hKernel = 0x55a68c790280`)、この問題が検出されたカーネル名を示します(`GEMM`)

[ファイルへのリダイレクト] オプションを省略して、すべての出力(アプリケーションの出力 + ツールの出力)を 1 つのストリームにダンプすることで、より多くの情報を取得できます。単一のストリームにダンプを行うことで、アプリケーションの出力に関連するエラーの原因を特定するのに役立つことがあります(例えば、アプリケーションの初期化と計算の最初のフェーズでエラーが発生しているなどが分かります)。

```
Level Zero Matrix Multiplication (matrix size: 1024 x
1024, repeats 4 times) Target device: Intel(R) Graphics
[0x3ea5]
...
>>>> [104131109] zeKernelCreate: hModule = 0x55af5f39ca10 desc =
0x7ffe289c7f80 {29 0 0 GEMM} phKernel = 0x7ffe289c7e48 (hKernel = 0)
<<<< [104158819] zeKernelCreate [27710 ns] hKernel = 0x55af5f3ca600 ->
ZE_RESULT_SUCCESS (0)
...
>>>> [104345820] zeKernelSetGroupSize: hKernel = 0x55af5f3ca600 groupSizeX
= 256 groupSizeY = 1 groupSizeZ = 1
<<<< [104360082] zeKernelSetGroupSize [14262 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104373679] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600
argIndex = 0 argSize = 8 pArgValue = 0x7ffe289c7e50
<<<< [104389443] zeKernelSetArgumentValue [15764 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104402448] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600
argIndex = 1 argSize = 8 pArgValue = 0x7ffe289c7e68
<<<< [104415871] zeKernelSetArgumentValue [13423 ns] ->
ZE_RESULT_ERROR_INVALID_ARGUMENT (2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600
argIndex = 2 argSize = 8 pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex
= 3 argSize = 4 pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0)
...
Matrix multiplication time:
0.0427564 sec Results are
INCORRECT with accuracy: 1
...
Matrix multiplication time:
0.0430995 sec Results are
INCORRECT with accuracy: 1
```

正当性

オフロードコードは、接続された計算デバイスで大量の情報を効率良く処理するカーネル、または一部の入力パラメーターから大量の情報を生成する際に利用されます。それらのカーネルがクラッシュすることなく実行されると、多くの場合、正しい結果が得られていないことはプログラム実行のかなり後で判明します。

そのような場合、どのカーネルが誤った結果を生成しているか特定するのは困難です。誤った結果を生成するカーネルを特定する方法として、プログラムを 2 回実行することが考えられます。最初はホストベースの実装を実行し、2 回目はオフロード実装を実行してすべてのカーネル(多くは個々のファイル)の入力と出力を取得します。次に、結果を比較して、どのカーネルが予期しない結果を生成しているか確認します(特定のイプシロンで、オフロード・ハードウェアの操作順序やネイティブの精度の違いにより、結果の最後の 1 桁もしくは 2 桁がホストコードと異なる可能性があります)。

誤った結果を生成するカーネルを特定したら、インテル® ディストリビューションの GDB を使用して原因を調査します。基本情報と詳細なドキュメントへのリンクについては、「インテル® ディストリビューションの GDB を使用したデバッグのチュートリアル(Linux* 版)」(英語)を参照してください。

SYCL* と OpenMP* はどちらもオフロードされたカーネル内で標準の print メカニズム(SYCL* と C++ OpenMP* では `printf`、Fortran OpenMP* オフロードでは `print *` など)を利用できます。これを使用して、実行中の動作を確認できます。スレッドと SIMD レーンの出力をプリントし、同期メカニズムを追加して、プリントされた情報がプリント時に一貫性を持つようになることを検討してください。ストリームクラスを使用した DPC++ で同様のことを行う例は、『oneAPI GPU 最適化ガイド』に記載されています。OpenMP* オフロード向けの SYCL* の説明を同様のアプローチを適用できます。

ヒント:SYCL* カーネルでは `printf` は冗長的になる可能性があります。簡単にするため次のマクロを追加します。

```
#ifdef_SYCL_DEVICE_ONLY_
  #define CL_CONSTANT_attribute_
((opencl_constant)) #else
  #define
CL_CONSTANT
#endif
#define PRINTF(format, ...) { \
          static const CL_CONSTANT char _format[] = format;
          \ sycl::ONEAPI::experimental::printf(_format, ##_VA_ARGS_) }
```

使用例:`PRINTF("My integer variable:%d\n, (int) x);`

障害

SYCL* または OpenMP* オフロード言語の誤った用法が原因で JIT コンパイルが失敗すると、プログラムはエラーで終了します。

SYCL* では 事前コンパイルされていることを判定できない場合、OpenCL* バックエンドを選択して OpenCL* アプリケーションのインターセプト・レイヤーを使用すると、構文エラーを持つカーネルを特定できることがあります。

ロジックエラーは、クラッシュや実行中のエラーメッセージを引き起こす場合があります。これには以下が含まれます。

- 誤ったコンテキストに属するバッファーをカーネルに渡す場合
- "`this`" ポインターをクラス要素ではなくカーネルに渡す場合
- デバイスバッファーではなくホストバッファーを渡す場合

- カーネルで使用されなくても、初期化されていないポインターを渡す場合

インテル® ディストリビューションの GDB(またはネイティブ GDB)を使用して注意深く調査することで、生成されたすべてのコンテキストのアドレスを記録してオフロードカーネルに渡されるアドレスが正しいコンテキストに属するか確認できます。同様に、変数のアドレスがそれを含むクラスでなく、変数自身のアドレスと一致するか確認できます。

OpenCL* 割り当て用のインターセプト・レイヤーまたは、onetrace を使用して、適切なバックエンドを選択する方がバッファーとアドレスをトレースするよりも簡単なことがあります。OpenCL* バックエンドを使用する場合、`CallLogging`、`BuildLogging`、`ErrorLogging`、および `USMChecking` を設定してプログラムを実行すると、コード内のどのエラーが OpenCL* エラーを生成したか明らかにする出力が生成されます。onetrace の呼び出しログやデバイス・タイムラインを参照すると、レベルゼロのバックエンドからのエラーの原因を理解するのに役だつ追加のエラー情報が得られます。これは、上記の論理エラーを検出するのに役立ちます。

レベルゼロ・バックエンドを使用してデバイスにオフロードする際にコードでエラーが発生する場合、OpenCL* バックエンドを試してみてください。プログラムが正常に機能する場合、レベルゼロ・バックエンドにエラーをレポートしてください。デバイス向けの OpenCL* バックエンドでもエラーが再現する場合、OpenCL* CPU バックエンドを試します。OpenMP* オフロードでは、OMP_TARGET_OFFLOAD を CPU に設定することで指定できます。SYCL* では、`SYCL_DEVICE_FILTER=opencl:cpu` を設定します。CPU 上でのデバッグはかなり容易になり、データのコピーとプログラムのデバイスへの変換によって生じる複雑性も排除できます。

問題が発生する可能性があるロジックの例として、次の SYCL* コードで `parallel_for` を実装する際に使用されるラムダ関数でキャプチャーされる対象を考えます。

```
class MyClass { private:
  int *data;
  int factor;
  :
void run() {
  :
  auto data2 = data; auto factor2 = factor;
  {
    dpct::get_default_queue_wait().submit([&](cl::sycl::handler &cgh)
    {
       auto dpct_global_range = grid * block; auto dpct_local_range = block;
       cgh.parallel_for<dpct_kernel_name<class
           kernel_855a44>>( cl::sycl::nd_range<1>(
             cl::sycl::range<1> dpct_global_range.get(0)),
             cl::sycl::range<1>( dpct_local_range.get(0))),
             [=](cl::sycl::nd_item<3> item_ct1)
       {
           kernel(data, b, factor, LEN, item_ct1);  // This blows up
       });
     });
}
} // run
} // MyClass
```

上記のコードでは、`[=]` がラムダ内で使用される変数を値でコピーするため、プログラムはクラッシュします。この例では、"`factor`" は実際には "`this->factor`" で、"`data`" は "`this->data`" であるため、"`this`" は "`data`" および "`factor`" を使用するために取得される変数です。OpenCL* またはレベルゼロでは、"`kernel(data, b, factor, LEN, item_ct1)`" 呼び出しで不正な引数エラーが原因でクラッシュします。

この問題を解決するには、ローカル変数 `auto data2` と `auto factor2` を使用します。"`auto factor2 = factor`" は "`int factor2 = this->factor`" になるため、ラムダ内で `[=]` を指定して `factor2` を使用すると、"`int`" が取得されます。内部セクションを "`kernel(data2, b, factor2, LEN, item_ct1);`" に変更します。

注: この問題は、CUDA* カーネルを移行する際によく発生します。同じ CUDA* カーネルの起動シグネチャーを保持し、コマンドグループとラムダをカーネル内に配置することで問題を解決することもできます。

OpenCL* 割り当てのインターセプト・レイヤーや onetrace を使用すると、カーネルが 2 つの同一アドレスで呼び出されることが分かり、拡張エラー情報を見ると、重要なデータ構造をオフロードデバイスにコピーしようとしていることが確認できます。

統合共有メモリー (USM) を使用しており、"`MyClass`" が USM に割り当てられる場合、上記のコードが動作することに注意してください。ただし、USM に "`data`" のみが割り当てられている場合、前述の理由からプログラムはクラッシュします。

この例では、カーネル呼び出しですべての変更する必要がないことから、ローカルスコープ内で同じ名前の変数を再宣言できることも留意してください。

インテル® Inspector は、このような障害の診断に役立ちます。次の環境変数を設定して、CPU デバイスでオフロードコードのメモリー解析を実行すると、インテル® Inspector は上記の問題の多くを通知します。

- OpenMP*
  - `export OMP_TARGET_OFFLOAD=CPU`
  - `export OMP_TARGET_OFFLOAD=MANDATORY`
  - `export LIBOMPTARGET_PLUGIN=OPENCL`
- DPC++
  - `export SYCL_DEVICE_FILTER=opencl:cpu`
  - または、CPU セレクターを使用してキューを初期化し、OpenCL* CPU デバイスの仕様を強制します:`cl::sycl::queue Queue(cl::sycl::cpu_selector{});`
- 両方
  - `export CL_CONFIG_USE_VTUNE=True`
  - `export CL_CONFIG_USE_VECTORIZER=false`

注: コンパイル中に最適化を有効にするとクラッシュする可能性があります。最適化を無効にしてクラッシュが解決される場合、デバッグ向けに -g -O[最適化レベル] を指定します。詳細については、『インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス』を参照してください。

6.3.4 オフロードのパフォーマンスを最適化

オフロード・パフォーマンスの最適化は、3 つのタスクに要約できます。

1. デバイス上のカーネルの実行時間を最大化しつつ、デバイス間とのデータ転送回数とサイズを最小化します。
2. 可能であれば、デバイス上の計算とデバイスとのデータ転送をオーバーラップさせます。
3. デバイス上のカーネルのパフォーマンスを最大化します。

OpenMP* オフロードと SYCL* の両方でデータ転送を明示的に制御することができますが、これを自動的に行うこともできます。また、ホストとオフロードデバイスはほとんど非同期に動作するため、データ転送を制御しようとしても転送が期待通りに行われず、予想よりも時間を要することがあります。デバイスとホストの両方でアクセスされるデータが統合共有メモリー (USM) に格納されている場合、パフォーマンスに影響する透過的な別のレイヤーのデータ転送が行われる可能性があります。

リソース:

- インテル® oneAPI GPU 最適化ガイド
- インテル® oneAPI FPGA 最適化ガイド(英語)

バッファー転送時間と実行時間

オフロードデバイス間とのデータ転送には比較的コストがかかり、ユーザー空間でのメモリー割り当て、システムコール、およびハードウェア・コントローラーとのインターフェイスが必要になります。統合共有メモリー(USM)は、ホストまたはオフロードデバイスのいずれかのメモリーの更新を同期するバックグラウンド・プロセスによるコストがかかります。さらに、オフロードデバイス上のカーネルは、実行に必要なすべての入力および出力バッファーがセットアップされて利用可能になるまで待機する必要があります。

1 回のデータ転送でオフロードデバイスとやり取りする情報量に関わりなく、このオーバーヘッドのコストはほぼ同じです。そのため、1 回に 1 つずつではなく、10 回分の転送をまとめて行うほうがはるかに高効率です。いずれにしても、すべてのデータ転送にはコストが生じるため、転送の総数を最小限にすることが重要です。例えば、複数のカーネルまたは同じカーネルの複数の呼び出しで必要とする定数がある場合、それらはカーネルを呼び出すたびに送信するのではなく、一度オフロードデバイスに転送して再利用するようにします。単一の大量のデータ転送には、単一の少量のデータ転送よりも時間がかかります。

送信されるバッファーの数とサイズは式の一部にすぎません。データがオフロードデバイスに到達したら、カーネルが実行される時間を検討します。オフロードデバイスへのデータ転送よりも実行時間が短い場合、同一操作をホストで実行する時間が、カーネルの実行とデータ転送の合計時間よりも長くない限り、オフロードのメリットはありません。

最後に、あるカーネルの実行と次のカーネルの実行の間に、オフロードデバイスがアイドル状態になっている時間を調査します。長い待機時間は、データ転送やホスト上のアルゴリズムの特性が原因である可能性があります。前者は、データ転送とカーネル実行のオーバーラップを試す価値があります。

ホストでのコード実行、オフロードデバイスでのコード実行、およびデータ転送の関係は複雑です。これらの順番と時間は、単純なコードであっても直感的に理解できるものではありません。すべてのアクティビティーを視覚的に理解し、その情報を参考にしてオフロードコードを最適化するには、次のようなツールが必要になります。

インテル® VTune™ プロファイラー

インテル® VTune™ プロファイラーは、ホスト上の詳しいパフォーマンス情報を提供するだけでなく、接続された GPU のパフォーマンスに関する詳細情報も提供できます。GPU 向けの設定に関する情報は、『インテル® VTune™ プロファイラー・ユーザーガイド』をご覧ください。

インテル® VTune™ プロファイラーの GPU オフロードビューには、それぞれのカーネル間とのデータ転送に費やされた時間など、GPU 上のホットスポットに関するサマリーが表示されます。GPU 計算/メディ

ア・ホットスポット・ビューでは、**動的命令カウント**を使用して GPU カーネルのパフォーマンスのマイクロ解析など、GPU カーネルで何が起こっているか詳しく調査することができます。これらのプロファイル・モードでは、データ転送と計算が時間経過でどのように変化するか観察したり、カーネルを効率良く実行するのに十分なワークがあるか判断したり、カーネルが GPU メモリー階層をどのように利用するか調べたりできます。

これらの解析タイプの詳細については、『インテル® VTune™ プロファイラー・ユーザーガイド』を参照してください。インテル® VTune™ プロファイラーを使用した GPU の最適化に関する詳細は、「インテル® VTune™ プロファイラーでインテル® GPU 向けのアプリケーションの最適化」(英語)をご覧ください。

カーネルの実行時間を計測するには、インテル® VTune™ プロファイラーも使用できます。次のコマンドは、軽量プロファイルの結果を示します。

- `Collect`
  - **レベルゼロ・バックエンド**:
    ```
    vtune -collect-with runss -knob enable-gpu-level-zero=true
    -finalization-mode=none -app-working-dir <app_working_dir> – <app>
    ```
  - **OpenCL* バックエンド**:
    ```
    vtune -collect-with runss -knob collect-programming-api=true
    -finalization-mode=none -app-working-dir <app_working_dir> – <app>
    ```
  - **レポート**:
    ```
    vtune --report hotspots --group-by=source-computing-task
    --sort-desc="Total Time" -r vtune-test
    ```

インテル® Advisor

インテル® Advisor は、計算を GPU にオフロードするパフォーマンスを向上させるのに役立つ 2 つの機能を提供します。

- オフロードのモデル化は、ホストの OpenMP* プログラムを調査して、GPU へのオフロードに適したコード領域を示します。また、さまざまな異なるターゲット向けに GPU をモデル化できるため、ターゲットに適したオフロードコード領域を特定できます。オフロード・アドバイザーは、オフロードのパフォーマンスを制限する可能性がある要因に関する詳細情報を提供します。
- GPU ルーフライン解析は、GPU で実行されるアプリケーションを監視し、各カーネルが GPU のメモリー・サブシステムと計算ユニットをどの程度効率良く使用しているか視覚的に示します。これにより、カーネルが GPU にどれくらい最適化されているか知ることができます。

すでにオフロードを実装するアプリケーションでこのモードを使用するには、CPU 上の OpenCL* デバイスを解析ターゲットにするよう環境を設定する必要があります。手順は、『インテル® Advisor ユーザーズガイド』(英語)をご覧ください。

オフロードのモデル化では、GPU を使用するようにアプリケーションを変更する必要はありません。ホストコードで完全に機能します。

リソース:

- インテル® Advisor クックブック:GPU オフロード
- オフロードのモデル化入門(英語)
- GPU* ルーフライン入門(英語)

OpenCL* アプリケーションのタイムライン向けの onetrace とインターセプト・レイヤー

インテル® VTune™ プロファイラーを使用して、データが GPU にコピーされるタイミング、およびカーネルが実行されるタイミングを調査したくない(またはできない)場合、onetrace と OpenCL* アプリケーションのインターセプト・レイヤーでもこの情報を監視する方法が用意されています(ただし視覚的なランタイム情報が必要な場合は、出力を視覚化するスクリプトを用意する必要があります)。詳細については「トレースとオフロードの手順」をご覧ください。

6.4 パフォーマンス・チューニング・サイクル

パフォーマンス・チューニング・サイクルの目標は、対話型の応答時間やバッチジョブの経過時間など、問題解決までの時間を短縮することです。ヘテロジニアス・プラットフォームの場合、ホストと独立して実行されるデバイスで利用可能な計算サイクルがあります。これらのリソースを利用してパフォーマンスを向上させます。

パフォーマンス・チューニング・サイクルには、次のステップがあります。

1. ベースラインの確定
2. オフロードするカーネルの特定
3. カーネルのオフロード
4. 最適化
5. 目標を達成するまで繰り返す

6.4.1 ベースラインの確定

経過時間、計算カーネル時間、1 秒あたりの浮動小数点演算などのメトリックを含むベースラインを確定します。これは、パフォーマンス向上の測定だけでなく、結果の正当性を検証する手段としても利用できます。

ベースラインを確定する簡単な方法は、C++ のタイム・ライブラリー・ルーチンを使用して、ワークロードの実行前後でタイマー呼び出しを行うことです。

6.4.2 オフロードするカーネルの特定

ヘテロジニアス・プラットフォームのデバイスで利用可能な計算サイクルを最大限に活用するには、計算集約型で並列実行可能なタスクを特定することが重要です。CPU のみで実行されるアプリケーションを調査すると、GPU で実行するのに適したタスクが見つかることがあります。これは、インテル® Advisor(英語)のオフロードのモデル化機能で判別できます。

インテル® Advisor は、アクセラレーターで実行される可能性があるワークロードのパフォーマンス特性を推測できます。ワークロードのプロファイル情報から、パフォーマンスを見積もり、ボトルネックの特性を評価して、さらにオフロードデータ転送を推測します。

一般に、計算主体で、大規模なデータセットを持ち、限られたデータ転送を行うカーネルは、デバイスへのオフロードに適しています。

機能の説明については、『インテル® Advisor ユーザーズガイド』をご覧ください。

6.4.3 カーネルをオフロード

オフロードに適したカーネルを特定したら、DPC++ を使用してカーネルをデバイスにオフロードします。詳細は以前の章をご覧ください。

6.4.4 最適化

oneAPI は、複数のアクセラレーターで実行できるコードを生成します。ただし、コードはすべてのアクセラレーターでは最適ではない可能性があります。パフォーマンスの目標を達成するには 3 段階の最適化を行うことを推奨します。

1. アクセラレーター全体に適用される一般的な最適化を行います。
2. 優先するアクセラレーターに対して積極的に最適化を行います。
3. ステップ 1 と 2 を組み合わせてホストコードを最適化します。

最適化とは、ボトルネック(ほかのコードセクションに比べて実行時間が長いコード領域)を排除する作業です。これらのセクションは、デバイスまたはホストで実行できます。最適化では、インテル® VTune™ プロファイラーなどのプロファイル・ツールを使用して、コード内のボトルネックを特定します。

ここでは、最初のステップであるアクセラレーター全体に適用される一般的な最適化を検討します。デバイス固有の最適化、デバイス固有のベスト・プラクティス(ステップ 2)、およびホストとデバイス間の最適

化(ステップ 3)については、『インテル® oneAPI DPC++ FPGA 最適化ガイド』(英語)などのデバイス固有の最適化ガイドで詳しく説明されています。このセクションでは、アクセラレーターにオフロードするカーネルがすでに決定されていること、および単独のアクセラレーターでワークが完了することを想定しています。このガイドは、ホストとアクセラレーター間、またはホストと複数の/異なるアクセラレーター間のワークの分割については考慮していません。

アクセラレーター全体に適用される一般的な最適化は次の 4 つに分類されます。

1. 高レベルの最適化
2. ループ関連の最適化
3. メモリー関連の最適化
4. DPC++ 固有の最適化

次のセクションでは上記の最適化のみをカバーします。これらの最適化を実際にコードに反映する方法の詳細は、オンラインや一般に入手できるコード最適化関連の資料で見付けることができます。ここでは、DPC++ 固有の最適化に関する詳細を示します。

高レベルの最適化

- 並列ワークの量を増やします。処理要素を十分に活用するには、処理要素よりも多くのワークが必要です。
- カーネルのコードサイズを最小化します。これにより、アクセラレーターの命令キャッシュにカーネルが保持されます。
- カーネルのロードバランスを取ります。実行時間の長いカーネルはボトルネックとなり、ほかのカーネルのスループットに影響する可能性があるため、カーネル間の実行時間は大きく異ならないようにします。
- 高コストの関数は避けてください。実行時間が長い関数は、ボトルネックとなる可能性があるため呼び出さないでください。

ループ関連の最適化

- 適切に構造化および構成された、単純な終了条件のループを使用します。単純な終了条件のループとは、出口が 1 つであり、整数上限値との比較で単一の終了条件を持つループです。
- 線形インデックスと定数上限値を持つループを優先します。例えば、配列への整数インデックスを使用し、コンパイル時に上限値が判明しているループなどです。
- 可能な限り深いスコープで変数を宣言します。これにより、メモリーまたはスタックの使用量を軽減できます。
- ループ伝搬されるデータの依存関係を最小化または緩和します。ループ伝搬の依存関係により、並列化が制限されることがあります。可能な限り依存関係を排除します。排除できない場合は、依存関係の距離を最大化するか、依存関係をローカルメモリー内に留めます。

- pragma unroll でループをアンロールします。

メモリー関連の最適化

- 可能な限り、メモリー使用よりも計算を優先します。メモリーのレイテンシーと帯域幅のほうが計算よりもボトルネックになる可能性があります。
- 可能であれば、グローバルメモリーよりもローカルおよびプライベート・メモリーを使用します。
- ポインターのエイリアシングを避けます。
- メモリーアクセスを結合します。メモリーアクセスをグループ化して個々のメモリー要求の数を減らし、キャッシュラインの利用率を高めます。
- 可能であれば、頻繁に実行されるコード領域の変数と配列をプライベート・メモリーに保持します。
- 同時メモリーアクセスに対するループアンロールの影響に注意してください。
- 別のカーネルが読み取るグローバルメモリーへの書き込みを避けます。代わりにパイプを使用します。
- カーネルに `[[intel::kernel_args_restrict]]` 属性を適用することを検討してください。この属性により、コンパイラーは DPC++ カーネルのアクセサー引数間の依存関係を無視できます。アクセサー引数の依存関係を無視すると、コンパイラーはさらに積極的な最適化を行い、カーネルのパフォーマンスが向上する可能性があります。

SYCL* 固有の最適化

- 可能であれば、work-group サイズを指定します。属性 `[[cl::reqd_work_group_size(X, Y, Z)]]`(X、Y、Z は ND-range の整数次元)を使用して、work-group のサイズを設定できます。コンパイラーはこの情報を利用して積極的な最適化を行うことができます。
- 可能であれば、`-Xsfp-relaxed` オプションの使用を検討してください。このオプションは、算術浮動小数点演算の順序付けを緩和します。
- 可能であれば、`-Xsfpc` オプションの使用を検討してください。このオプションは、可能な場合は常に中間の浮動小数点丸め操作と変換を排除して、精度を維持するため追加ビットをキャリーします。
- `-Xsno-accessor-aliasing` オプションの使用を検討してください。このオプションは、SYCL* カーネルのアクセサー引数間の依存関係を無視します。

6.4.5 再コンパイル、実行、プロファイル、そして繰り返し

コードを最適化したら、パフォーマンスを測定することが重要です。次を確認します。

- メトリックは改善されましたか?
- パフォーマンスの目標は達成できましたか?

- 利用可能な計算サイクルが残されていますか?

結果の正当性を確認します。数値結果を比較すると、コンパイラーの最適化やコード変更により異なる場合があります。差異は許容範囲内ですか? そうでなければ、最適化ステップに戻ります。

6.5 oneAPI ライブラリーの互換性

oneAPI アプリケーションには、インテルのツールのリリースバージョンとの互換性のため、動的ライブラリーが実行時に含まれる場合があります。インテル® oneAPI ツールキットとコンポーネント製品は、互換性を維持するためセマンティック・バージョニングを使用します。

次のポリシーが、インテル® oneAPI ツールキットで提供される API および ABI に適用されます。

注: oneAPI アプリケーションは、64 ビットのターゲットデバイスでサポートされます。

- 新しいインテル® oneAPI デバイスドライバー、インテル® oneAPI 動的ライブラリー、およびインテル® oneAPI コンパイラーは、インテル® oneAPI ツールを使用してビルドされた展開済みのアプリケーションを破壊することはありません。現在の API は、通知とメジャーバージョンの重複なしで削除または変更されることはありません。
- oneAPI アプリケーションの開発者は、ヘッダーファイルとライブラリーのリリースバージョンが同じであることを確認する必要があります。例えば、アプリケーションで、インテル® oneAPI マス・カーネル・ライブラリー 2021.2 のヘッダーファイルをインテル® oneAPI マス・カーネル・ライブラリー 2021.1 で使用してはなりません。
- インテル® コンパイラーで提供される新しい動的ライブラリーは、古いバージョンのコンパイラーで作成されたアプリケーションでも動作します (これは一般に下位互換と呼ばれます)。しかし、その逆は当てはまりません。インテル® oneAPI 動的ライブラリーの新しいバージョンには、以前のバージョンでは提供されていないルーチンが含まれる場合があります。
- oneAPI 対応のインテル® コンパイラーで提供される古い動的ライブラリーは、新しいバージョンの oneAPI コンパイラーでは機能しません。

oneAPI アプリケーションの開発者は、oneAPI アプリケーションが互換性のあるインテル® oneAPI ライブラリーとともに展開されていることを確認するため、完全なアプリケーションのテストを実施する必要があります。

用語集

アクセラレーター(Accelerator)

操作のサブセットを迅速に実行する計算リソースを含む専用コンポーネント。
例:CPU、GPU、または FPGA。

参照:デバイス

アクセサー(Accessor)

アクセスする場所(ホスト、デバイス)とモード(read、write)を通信します。

アプリケーション・スコープ(Application Scope)

ホスト上で実行されるコード。

バッファー(Buffers)

計算を行うためデバイスに送られる項目の数や型を通信するメモリー・オブジェクト。

コマンド・グループ・スコープ(Command Group Scope)

ホストとデバイス間のインターフェイスとして動作するコード。

コマンドキュー(Command Queue)

コマンドグループを同時に発行します。

計算ユニット(Compute Unit)

処理要素間で使用する共有要素を含み、デバイス上のほかの計算ユニットにあるメモリーよりも高速にアクセスするため、処理要素を「コア」にグループ化したもの。

デバイス(Device)

操作のサブセットを迅速に実行する計算リソースを含むアクセラレーターまたは専用コンポーネント。CPU はデバイスとして利用できますが、その場合、CPU はアクセラレーターとして使用されます。
例:CPU、GPU、または FPGA。

参照：アクセラレーター

デバイスコード（Device Code）

ホストではなくデバイスで実行されるコード。デバイスコードは、ラムダ式、ファンクター、またはカーネルクラスを介して指定されます。

ファットバイナリー（Fat Binary）

複数デバイス向けのデバイスコードを含むアプリケーション・バイナリー。バイナリーには、汎用コード（SPIR-V* 形式）とターゲット固有の実行可能コードの両方が含まれます。

ファット・ライブラリー（Fat Library）

複数デバイス向けのオブジェクト・コードを含むアーカイブまたはライブラリー。ファット・ライブラリーには、汎用オブジェクト（SPIR-V* 形式）とターゲット固有のオブジェクト・コードの両方が含まれます。

ファット・オブジェクト（Fat Object）

複数デバイス向けのオブジェクト・コードを含むファイル。ファット・オブジェクトには、汎用オブジェクト（SPIR-V* 形式）とターゲット固有のオブジェクト・コードの両方が含まれます。

ホスト（Host）

プログラムの主要部分、具体的にはアプリケーション・スコープとコマンド・グループ・スコープを実行する CPU ベースのシステム（コンピューター）。

ホストコード（Host Code）

ホスト・コンパイラーによってコンパイルされ、デバイスではなくホストで実行されるコード。

イメージ（Images）

組込み関数を介してアクセスされるフォーマット済みのあいまいなメモリー・オブジェクト。通常、RGB などの形式で保存されピクセルで構成される画像に関係します。

カーネルスコープ（Kernel Scope）

デバイス上で実行されるコード。

ND-range

1 次元、2 次元、または 3 次元の N 次元レンジ、カーネル・インスタンスのグループ、またはワーク項目の略。

処理要素(Processing Element)

計算ユニットを構成する計算向けの個別のエンジン。

単一ソース(Single Source)

ホストとアクセラレーターで実行できる同じファイルのコード。

SPIR-V*

グラフィカル・シェーダー・ステージと計算カーネルを表現するバイナリー中間言語。

サブグループ(Sub-group)

サブグループはインテルによる拡張です。

ワークグループ(Work-group)

計算ユニットで実行するワーク項目の集合。

ワーク項目(Work-item)

oneAPI プログラミング・モデルにおける計算の基本単位。処理要素で実行されるカーネルに関連付けられます。