データ並列 C++ を開始する

この記事は、インテル® デベロッパー・ゾーンに公開されている「Dive Into Data Parallel C++」の日本語参考訳です。

オープンなスタンダードベースのクロスアーキテクチャー・プログラミング・ソリューション

ソフトウェア開発者は、市場でアクセラレーター・アーキテクチャーの数が増加する中、データセントリックな複数のハードウェアにわたるプログラミングの難しさを知っているでしょう。oneAPI 業界イニシアチブ (英語) はこの課題に取り組んでいます。オープンスタンダード・ベースの oneAPI は、誰でも貢献および利用することができます。oneAPI の目標は、異なるアクセラレーター・アーキテクチャーにわたってソフトウェアを記述するのを妨げる開発上の障害を取り除くことです。どこでも利用可能な共通の言語と API を提供することで、どのアーキテクチャーでもコードが素早く機能し、アーキテクチャーごとの解析とチューニングによりピーク・パフォーマンスを達成できるようにします。

oneAPI イニシアチブの中核を成すのは、C++ (英語)と Khronos Group[1] の SYCL*[1] (英語) を拡張するデータ並列 C++ (DPC++) と呼ばれるプログラミング言語です。DPC++ は、ヘテロジニアス・フレームワークの CPU、GPU、FPGA、および AI アクセラレーターにわたる一貫したプログラミング言語を提供し、各アーキテクチャーを個別に、または組み合わせて、プログラムおよび使用できます。共通の言語と API により、開発者は一度習得するだけで異なるアクセラレーターをプログラムできます。アクセラレーターの各クラスではアルゴリズムの適切な定式化とチューニングが必要になりますが、アルゴリズムの開発とチューニングに使用される言語は、ターゲットデバイスに関係なく一貫しています。

DPC++ は、標準化に基づいた言語実装であるとともに、標準化に対する新機能の提案を実証する場でもあると考えるべきです。新しい機能は、標準化される前に評価されるのが一般的です。これにより、実際に機能が実装可能であり、有用であり、十分に設計されているという確信が得られます。DPC++ の構成は、次の式で理解できます。

DPC++ = C++ + SYCL* + 拡張

拡張は、標準化に含まれない追加機能です。DPC++ の目標は、拡張が実証されたら、そのおおもとである SYCL* または C++ 仕様に拡張の実装を提案することです。DPC++ は、SYCL* や C++ から派生するのではなく、実証された新機能でそれらを補完することを目的としています。

DPC++ の言語および API 拡張は、さまざまな開発使用例をもたらし、そのすべてはサポートされ、機能設計が考慮されます。使用例には、新しいオフロード・アクセラレーションの開発、ヘテロジニアス計算アプリケーション、既存の C/C++ コードから SYCL や DPC++ への変換、ほかのアクセラレーター言語やフレームワークからの移行が含まれます。また、DPC++ は上層にヘテロジニアス・フレームワークを配置できるように設計されており、バックエンドとして機能し、ヘテロジニアス・ソリューションで共存できます。

インテルは、DPC++ と SYCL* の両方の開発に深くかかわっており、オープンソース実装を構築し、コミュニティー主導の LLVM プロジェクトに直接貢献しています。DPC++ オープンソース・プロジェクト (英語) は、2019 年 1 月に GitHub* (英語) で公開されて以来、さまざまな企業や貢献者からの多大な支援を含め、アク

ティブに開発を進めています。オープンソース・ツールチェーンは、Clang (英語) と LLVM (英語) プロジェクトをベースにしています。

設計上、DPC++ 言語は複数のベンダーのハードウェアをサポートするように設定されており、開発者は単一のベンダーに縛られず、さまざまなパフォーマンスと価格帯の選択肢から自由に選択できます。標準ベースのアプローチとオープンソース実装により、開発者は幅広いハードウェア・ターゲットを長期間にわたって利用できます。現在、DPC++ はインテルの CPU、GPU、FPGA を含むハードウェアをサポートしています。Codeplay* は、NVIDIA GPU[2] をターゲットとする DPC++ コンパイラーも発表しています。

DPC++ と SYCL* は、OpenCL*[3] を含む下位レベルのインターフェイスの上で、プログラミングの抽象化レイヤーとして動作するように設計されています。この設計は、既存のソフトウェアとドライバースタックに基づく大規模な展開と、必要に応じて下位レベルのスタックとの相互運用性を可能にします。例えば、ベータ版の DPC++ 実装は、SPIR*-V をサポートする任意の OpenCL* 1.2 実装で実行するように設計されています (新しい機能向けのインテルの拡張を利用する場合もあります)。多様なバックエンド環境での改善のため、各種ターゲット上で DPC++ を実行して、oneAPI オープン・プロジェクト (英語) にフィードバックをお寄せください。

DPC++ の詳細

DPC++ 言語は、C++ から生産性向上の利点と使い慣れた構文を継承し、SYCL* からデータ並列処理とヘテロジニアス・プログラミングのクロスアーキテクチャー・サポートを取り込んでいます (**図 1**)。

図 1. DPC++ は C++ をベースに、Khronos Group の SYCL* と XPU アーキテクチャー開発向けのオープン拡張が採用されています。

タスクグラフを使用して XPU 内/間で順序付けされたカーネルベース・モデル

SYCL*、そして DPC++ は、開発者がアクセラレーターで実行するコードと (通常、アクセラレーターによって行われる処理を調整するため) ホスト・プロセッサーで実行するコードを単一のソースに混在させる、シングルソース・プログラミング・モデルを提供します。シングルソース・プログラミングは、より簡単に開発を始めることができ、大規模なアプリケーションには型セーフと最適化の利点をもたらします。

「カーネル」は、アクセラレーターで実行するコードです。通常、このコードは、ND-Range (1、2、または 3 次元の反復空間における work-item の空間) 全体でカーネルが呼び出されるたびに複数回実行されます。各インスタンスは、異なるデータを処理します。カーネルは、コード内で明確に識別できます。開発者は、カーネルをキューに送信して実行する、1 つまたは複数のターゲットデバイスを選択します。SYCL* のいくつかのクラスが、カーネルの実行方法 (例えば parallel_for) を定義します。

カーネルは、以下を含むいくつかの方法で定義できます。

1. C++ ラムダ (通常、関数を呼び出すシンラッパー)
2. ファンクター (関数のように動作する C++ クラス)
3. 特定のバックエンドを使用する相互運用性カーネル (例えば、OpenCL* の cl_kernel オブジェクト)

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R };

  queue{}.submit([&](handler& h) {
    auto out =
      A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  auto result =
    A.get_access<access::mode::read>();
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";

  return 0;
}
```

カーネルを定義

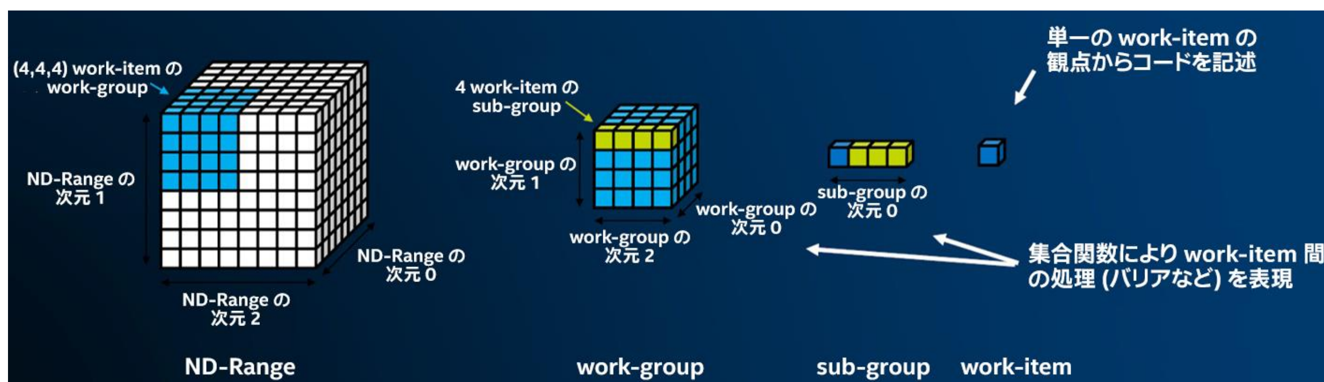

図 2. ボトムアップの階層的な SPMD (Single Program Multiple Data) モデル。

DPC++ プログラムは、次の機能を追加するキュー、バッファー、デバイスなどの抽象化を導入する C++ プログラムです。

- ワークで特定のアクセラレーター・デバイスをターゲットにする
- データを管理する
- 並列処理を制御する

例えば、カーネル形式のワークはキューに送信されます。キューは、SYCL* で定義されたクラスで、特定の GPU や FPGA など、1 つのデバイスのみに関連付けられています。開発者は、キューの作成時に関連付ける単一のデバイスを指定できます。また、ヘテロジニアス・システムでワークをディスパッチするため任意の数のキューを作成できます (**図 3**)。

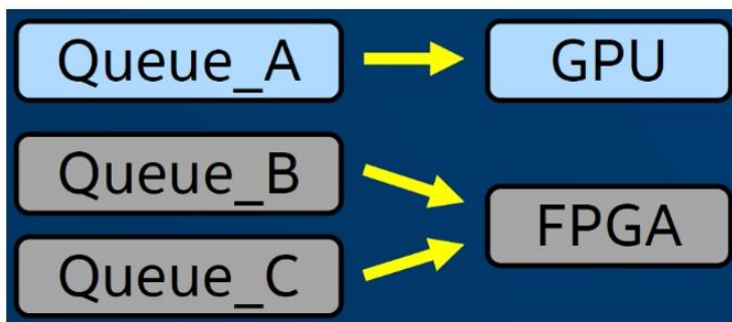

図 3. ヘテロジニアス・システムではキューがワークをディスパッチします。

カーネルはキューに送信され、キューはカーネルを実行する「場所」を定義します。カーネルを安全に実行できる「タイミング」を定義するため、SYCL* には *task graph* 抽象化があります。タスクグラフ (**図 4**) は、データ移動のメカニズムを自動化して、以下を達成します。

- 多くのアプリケーションで記述が必要なコード量を大幅に減らします。
- プログラムを変更せずに、デバイスとインターコネクト・バス間でデータの移動と同期を調整します。
- ユーザーがデータの移動を明示的に管理するまれなケースではオーバーライドすることができます。

カーネル実行の依存関係と順序付けは、通常、「データアクセサー」と呼ばれる抽象化を使用して、カーネル内のデータ使用によって決定されます。

例えば、以下のコードは、カーネル実行の順序付けを決定する依存関係エッジを含む、次のタスク実行グラフになります。

カーネルは、ランタイムシステムによって安全なタイミングで起動されて正しい順序を保証するため、コードや開発者によるアクションは必要ありません。この強力な抽象化を利用することで、アプリケーションを素早く記述し、異なるシステムトポロジーに透過的に適応することができます。

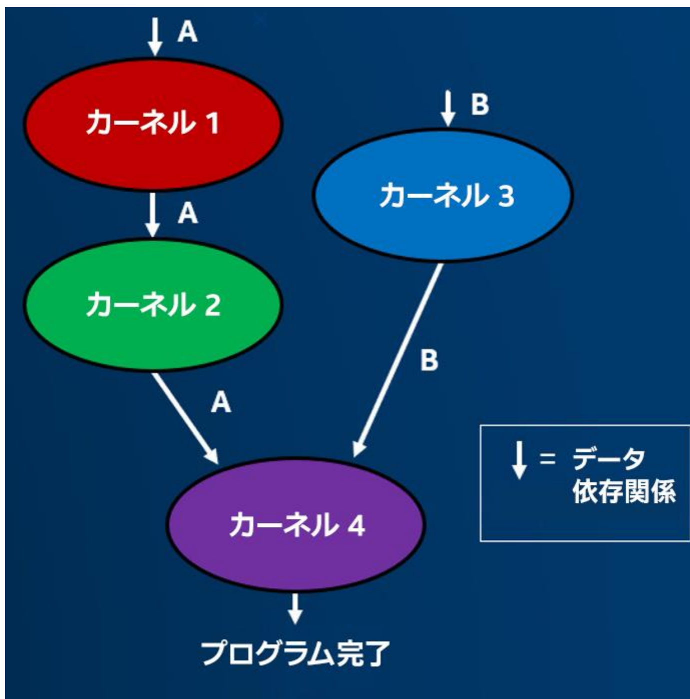

図 4. タスク実行グラフがカーネル実行を順序付けします。

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                  カーネル 1

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                  カーネル 2

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                  カーネル 3

  Q.submit([&](handler& h) {
  auto in = A.get_access<access::mode::read>(h);
  auto inout =
    B.get_access<access::mode::read_write>(h);
  h.parallel_for(R, [=](id<1> idx) {
    inout[idx] *= in[idx]; }); });                カーネル 4
}
```

データ・アクセス・モデル

DPC++ には、データへのアクセスをカプセル化して管理する、次の 2 つのメカニズムがあります。

1. SYCL* バッファーの抽象化は、デバイス全体とホストシステムからアクセス可能なバッファー・オブジェクトにデータをカプセル化します。「アクセサー」は、バッファー内のデータにアクセスするメカニズムを提供して、タスクグラフ内で依存関係エッジを作成し、バッファーのデータを参照/使用するためのハンドラーを提供します。
2. DPC++ 統合共有メモリー (USM) 拡張は、ポインターベースのダイレクト・プログラミングを可能にします。これは、既存の C++ コードを各種アクセラレーターで実行するヘテロジニアス・アプリケーションに変換する場合や、データ構造がホストとデバイスにわたる統合仮想アドレス空間を必要とする場合 (例えば、ホストとデバイスでトラバース可能なリンクリスト) に役立ちます。USM については、今後のブログで詳しく取り上げる予定です。

タスクグラフは非同期

DPC++ プログラミング・モデルで重要なことは、ホストプログラムはカーネルがデバイスで実行を完了 (または開始) するのを待機しません。代わりに、ホストプログラムはキューにワークを送信して、開発者が明示的にデバイス上の何かとの同期を要求しない限り、実行を継続します。例えば、**図 5** のコードでは、デバイスで実行するためカーネルがキューに送信され、ホストプログラムは送信後のコードの実行を継続します。

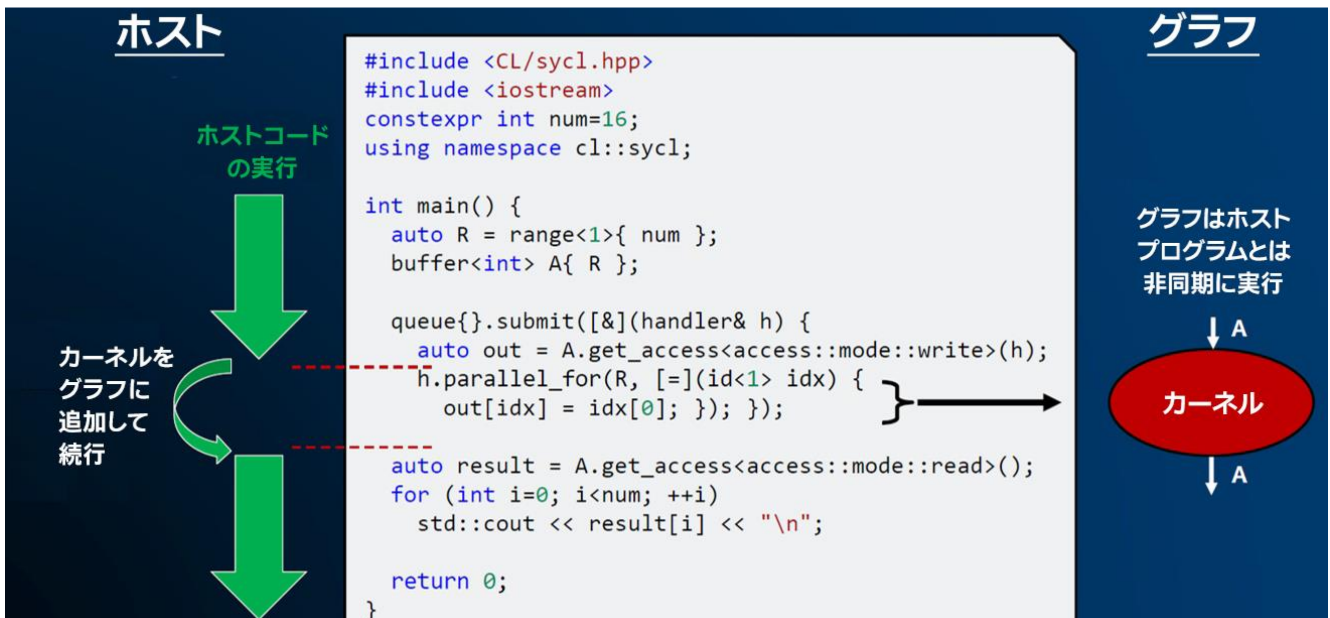

図 5. ホストコードはグラフを構築するワークを送信します。カーネル実行とデータ移動のグラフは、ホストコードとは非同期に動作し、SYCL* ランタイムによって制御されます。

この例では、ホストコードは 2 つ目の get_access 呼び出しで停止し待機します。カーネルがデバイスで生成するデータ (std::cout でストリーミングされる) を使用するため、ホストコードは、カーネルが実行を完了してデータがデバイスからホストに移動されるまで待機する必要があります。get_access 呼び出しや、ホストが待機する必要があるデバイス側のその他の処理がない場合、ホストプログラムはデバイス側の実行と並列に実行を継続します。

SYCL* の DPC++ 拡張

前述のとおり、SYCL* を補完する追加の機能としていくつかの DPC++ 拡張 (英語) が定義されています。以下はその一部です。

- ポインターベース・プログラミング向けの**統合共有メモリー**
- SIMD アーキテクチャーのハードウェアに効率良くマップする**サブグループ**
- 生産性を向上する効率良い**リダクション**と**グループ・アルゴリズム**
- 一般的なパターンを簡素化する**順序付きキュー**
- 空間とデータフロー・アーキテクチャーで効率良く計算するための**パイプ**
- コードの冗長性を軽減するための**オプションのラムダ命名**
- 一般的なコードパターンの表現を容易にする**簡素化**

まとめ

この記事では、DPC++ の概要と C++ や SYCL* との関係を説明し、SYCL* と DPC++ のいくつかの強力な抽象化を紹介しました。DPC++ の言語と API は、次の 2 つの方法で利用できます。

- インテル® DevCloud (英語)
- インテルのリファレンス実装 (インテル® oneAPI ツールキット)

インテルの oneAPI ソフトウェア製品は、DPC++ を含む oneAPI 業界仕様のベータ版実装であり、高度なツールを備えたいくつかのツールキットで構成されています。インテル® DevCloud では、無料でさまざまなインテル® アーキテクチャーでワークロードをビルド、テスト、および実行できます。

oneAPI コミュニティーに参加してクロスアーキテクチャー・アプリケーションを開発するには、こちら (英語) からベータ版インテル® oneAPI ツールキットの詳細を確認してダウンロードしてください。

または、インテル® DevCloud (英語) で、プロジェクトのプロトタイプを生成して、インテルの各種プロセッサーやアクセラレーターでコードとワークロードをテストできます。ダウンロード、ハードウェアの取得、インストール、セットアップと設定は不要です。

DPC++ 開発に関する追加のリソース

oneAPI プログラミング・ガイド日本語版

データ並列 C++: クロスアーキテクチャー開発のオープンな代替手段 (英語) [12:05]

近日出版予定の DPC++ 書籍の上級者向けの章 (英語)

無料の oneAPI と DPC++ ウェビナーとハウツー (英語)

著者

James Brodman
インテル コーポレーションのソフトウェア・エンジニアで、並列プログラミング向けの言語とコンパイラーを研究しています。現在は、oneAPI イニシアチブと DPC++ に取り組んでいます。これまでに、SIMD/ベクトル処理向けのプログラミング・モデル、並列処理向けの言語、分散メモリーの理論と実践、マルチコアシステムのプログラミングなど、幅広く執筆しています。

Mike Kinsner
インテル コーポレーションのソフトウェア・エンジニアで、さまざまなアーキテクチャー向けの並列プログラミング・モデルと空間アーキテクチャー向けの高レベル・コンパイラーに取り組んでいます。また、インテル代表として Khronos Group 内で SYCL* と OpenCL* 業界標準に貢献しており、現在は oneAPI イニシアチブの DPC++ に取り組んでいます。

脚注

[1]Khronos、SYCL、SPIR-V は Khronos Group の商標または登録商標です。

[2]「Codeplay contribution to DPC++ brings SYCL support for NVIDIA GPUs」(英語) 2020 年 2 月 3 日

[3]OpenCL および OpenCL ロゴは、Apple Inc. の商標であり、Khronos Group の使用許諾を受けて使用しています。

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、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製品またはコンポーネントはありません。実際の費用と結果は異なる場合があります。

インテルは、サードパーティーのデータについて管理や監査を行っていません。ほかの情報も参考にして、正確かどうかを評価してください。

インテルは、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、および非侵害性の黙示の保証、ならびに履行の過程、取引の過程、または取引での使用から生じるあらゆる保証を含みますが、これらに限定されるわけではありません。

© Intel Corporation. Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における 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。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。