複数アーキテクチャーにおける DPC++ のデータ管理

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「DPC++ Data Management across Multiple Architectures」の日本語参考訳です。

パート 1: oneAPI プログラミング・フレームワークを使用した効率良いデータの移動と制御

この記事では、oneAPI (英語) プログラミング・モデルにおけるハードウェア・アクセラレーターを含む、クロスアーキテクチャー・システムを対象とするデータ管理について説明します。データ並列 C++ (DPC++) は、oneAPI 向けのクロスアーキテクチャー言語であり、C++ や CUDA*、そして OpenCL* など他の言語でプログラミングを行う開発者に馴染みのある、使いやすいデータ管理方法を提供します (DPC++ の詳細については、言語とカーネルベースのアプローチに関する記事 (英語) をご覧ください)。

デバイスのアーキテクチャーにまたがるデータ管理の問題

オープンで標準化ベースの開発ソリューションでは、機能の移植性は考慮すべき最優先事項です。DPC++ フレームワークは、複数のベンダーから提供されるさまざまなタイプのアクセラレーター上でコードを実行するため、より簡単な開発手順を提供するように設計されており、単一ベンダーへの依存を排除できます。

まず、メモリー関連から始めましょう。図 1 は、従来のシングルソケット CPU ベースのシステムに単一のメモリーが接続されていることを示していますが、アクセラレーター・デバイスには、ホストから直接アクセスできない個別のメモリーが接続されることもあります。個別のデバイスをサポートする並列プログラミング・モデルは、これら複数のメモリーを管理し、メモリー間でデータを移動するメカニズムを提供する必要があります。

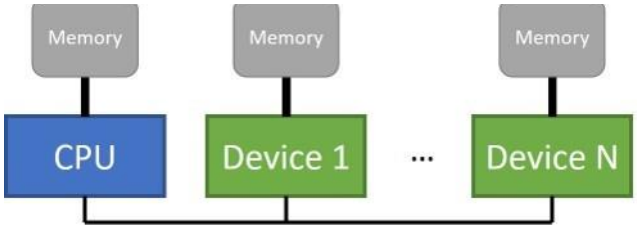

図 1. 複数の分離メモリー

デバイス上で実行される並列プログラムは、デバイスに直接接続されているメモリーのデータを読み書きできるため、ローカル・メモリー・アクセスを優先します。リモート・メモリー・アクセスは、帯域幅が狭くレイテンシーが長いデータリンクを介して移動する必要があるため、低速となる傾向があります。そのため、計算とデータを同一デバイスに配置することには利点がありますが、すべてのデバイスでこれを行うには、何らかの方法でデータを異なるメモリー間でコピーまたは移動して、計算デバイスに近づける必要があります。

複数メモリーの明示的および暗黙的な管理

複数メモリーの管理は、ランタイムによって暗黙的に行うことも、API などを使用して手動で明示的に行うこともできます。それぞれの方法には、長所と短所があります。

並列ランタイムまたはドライバーによって制御される暗黙的なデータ移動により、DPC++ ランタイムはデータを使用する前に適切なメモリーに自動転送できます。このアプローチの利点は、プログラミングの労力が減り、アプリケーションがデバイスのローカルメモリーを利用する際のプログラムエラーを軽減できることです。

ただし、暗黙的なアプローチでは、ランタイムの暗黙的なメカニズムをほとんど (または全く) 制御できないため、プログラムのパフォーマンスに悪影響を及ぼす可能性があります。これは利点であり、また、欠点であるとも言えます。ランタイムは機能的に正しく動作しますが、計算とデータ転送が最大限に重複しない可能性があり、プログラマーほどアプリケーションに関する情報を持ち合わせていません。

明示的なアプローチでは、異なるメモリー間のコードに対し手動で明示的なコピーを追加し、最適化とチューニングを行うことで目的のパフォーマンスを達成できます。どのような状況でこれを行うべきでしょう?

- 明示的なコピー/転送は、統合共有メモリー (USM、これについては後述します) および**デバイス**メモリーに対してのみ必要です。
- バッファーおよび他のタイプの USM も**コピー**機能をサポートしますが、これはアルゴリズムを正しく動作させるためメモリーのコピーを作成する場合にのみ必要となります。
- それぞれのメモリー空間の転送に関連するコピーは、ランタイムによって自動的に行われます。

例えば、ホストメモリーから GPU メモリーにデータを明示的にコピーする専用 GPU (図 2) では、カーネルが新しい結果を計算した後、ホストプログラムがそのデータを参照する前に、データを CPU に戻す (コピーする) 必要があります。明示的にコードを挿入してデータの移動開始を示すことで、異なるメモリー間でのデータ転送のタイミングを完全に制御できます。これにより、計算とデータ転送の重複を最適化して、最高のパフォーマンスを達成することができます。

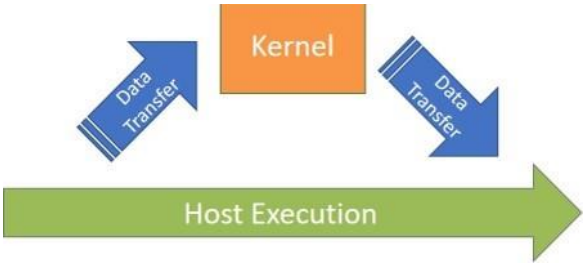

図 2. データ移動とカーネル実行

しかし、このすべてのデータ移動を事前に正しく行うことは、時間を要する退屈な作業です。また、不必要なデータを転送したり、カーネルが計算を開始する前にすべてのデータを完全に転送できない場合、不正な結果が生成される可能性があります。

プログラム内のさまざまなデータに対して、明示的および暗黙的な方法を組み合わせて制御できます。新しいデバイスへのアプリケーションの移植を容易にするため、またはアプリケーションの新規開発を簡単にするため、暗黙的なデータ移動から始めます。その後、アプリケーションが正しく動作することを確認してパフォーマンスのチューニングを開始し、プロファイラーや他の手法で識別されたパフォーマンスが重要なコード領域で暗黙的なデータ移動を明示的なデータ移動に置き換えることができます。

統合共有メモリー (USM)、バッファー、およびイメージ

DPC++ はメモリーを管理する 3 つの抽象化を提供します: USM、バッファー、およびイメージ。

- **USM** はポインターベースのアプローチです。USM を使用する利点は、ポインターを操作する既存の C++ コードとの統合が容易であり、C/C++ 開発者が使い慣れていることです。
- **バッファー**は、1、2、または 3 次元の配列を表現します。これらはメモリーの生の表現であり、ホストまたはデバイスのいずれかでアクセスできるメモリーの抽象的なビューを提供します。
- **イメージ**は、より抽象的で**あいまい**であり、サンプラー・オブジェクトによる画像の読み取りなど、特殊機能の限定されたセットで使用されます。

この記事では、ほとんどのアプリケーションに有効であると考えられる、USM とバッファーを中心に説明します。

データ管理の方針を選択

データ管理の方針を決定する際の最初の考慮点は、明示的または暗黙的なデータ移動のどちらを使用するか決めることです。DPC++ がデータ移動を処理し、開発者は計算の表現に集中できるため、暗黙的なアプローチのほうが一般に容易です。ただし、データ移動を完全に制御したい場合は、USM デバイス割り当てを使用する明示的なアプローチが適しています。ホストとデバイス間に必要なコピーを必ず追加してください。

暗黙的なデータ移動を選択した場合でも、バッファーまたは USM を使用することができます。ポインターを使用する既存の C/C++ プログラムを移植する場合、コードをほとんど変更する必要がないため、USM が適しているかもしれません。

次に決めなければならないことは、カーネル間の依存関係をどのように扱うかです。カーネル間でのデータ依存関係を優先する場合、バッファーを使用してください。ある計算を別の計算の前に実行すると考えられる場合、USM を使用します。計算の順序付けを行うため、DPC++ は、送信された順番でカーネルを逐次実行するインオーダー・キューと、カーネル間での明示的な依存関係制御を必要とし、複数の実行順序を持つ可能性があるアウトオブオーダー・キューをサポートします。

- インオーダー・キューはシンプルで直観的ですが、ランタイムに制約を及ぼし、重複した実行が行われないためパフォーマンスが低下する可能性があります。
- アウトオブオーダー・キューは、実行の依存関係を制御する必要があるため複雑ですが、実行のリオーダーやオーバーラップを自由に行えるため、より効率良く実行できます。

アウトオブオーダー・キューは、複雑な管理を厭わなければ、カーネル間で複雑な依存関係があり、パフォーマンスが重要なコードにおいて適切な選択肢でしょう。プログラムが逐次的に多くのカーネルを実行するならば、インオーダー・キューのほうが適切な選択肢である可能性があります。

次のステップ

クロスアーキテクチャー・プログラミングを多面的に理解するため、他の記事もご覧ください。さらに詳しいガイドについては、DPC++ に関する書籍の事前公開されている 4 つの章 (英語) を参照してください。この書籍は、2020 年後半に Apress から出版される予定です。

DPC++ を導入するには、次の 2 つの方法で言語と API を使用できます。

- インテル® DevCloud (英語): プロジェクトのプロトタイプを作成し、各種インテル® プロセッサーと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 (無料)。
- インテルの oneAPI リファレンス実装 (英語): ベータ版インテル® oneAPI ツールキット

その他の DPC++ 開発リソース

- インテル® oneAPI プログラミング・ガイド
- ビデオ: データ並列 C++: クロスアーキテクチャー開発におけるオープンな代替手段 [12.05] (英語)
- インテル® oneAPI と DPC++ に関連したウェビナーとガイド (英語)

Ben Ashbaugh は、oneAPI や DPC++ を含むインテル® グラフィック・プロセッサー向けの汎用計算の並列プログラミング・モデルに取り組んでいるインテルのソフトウェア・アーキテクトです。多数の OpenCL* 拡張機能を開発しており、Khronos Group におけるインテルの担当者であり、OpenCL*、SPIR*-V、および SYCL* 業界標準に貢献しています。

James Brodman は、言語とイニシアチブ、および DPC++ を研究しているインテルのソフトウェア・エンジニアであり、SIMD/ベクトル処理のプログラミング・モデル、並列処理向け言語、分散メモリー理論と実践、マルチコアシステムにおけるプログラミングなど、幅広く記事を執筆しています。

Mike Kinsner は、各種アーキテクチャーの並列プログラミング・モデル、および空間アーキテクチャー向け高レベル・コンパイラーに取り組んでいるインテルのソフトウェア・エンジニアです。Khronos Group におけるインテルの担当者であり、SYCL* および OpenCL* 業界標準に貢献しており、現在、oneAPI イニシアチブ内の DPC++ に注力しています。