oneAPI の maxloc リダクション

SYCL* と oneDPL でこの一般的な並列パターンを実装

Henry A Gabb インテル コーポレーション シニア主席エンジニア兼 The Parallel Universe 編集長
Alexey Kukanov インテル コーポレーション 主席ミドルウェア・エンジニア
John Pennycook インテル コーポレーション ソフトウェア・イネーブリング & 最適化アーキテクト

maxloc 操作とは

最大値の場所（`maxloc`）を見つけることは、配列で実行される一般的な検索操作です。多くのプログラミング言語とライブラリーが、組込みの `maxloc` 関数を提供しています（NumPy* argmax（英語）、Fortran maxloc（英語）、BLAS amax（英語）、C++ max_element（英語）関数など）。最近の記事、「インテル® AVX-512 命令を使用した Maxloc 操作の最適化」（The Parallel Universe 46 号）では、`maxloc` 検索をベクトル化してパフォーマンスを向上する方法を説明しました。相互相関を含む、多くのアルゴリズムで重要な操作であることは明白です（**図 1**）。

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。

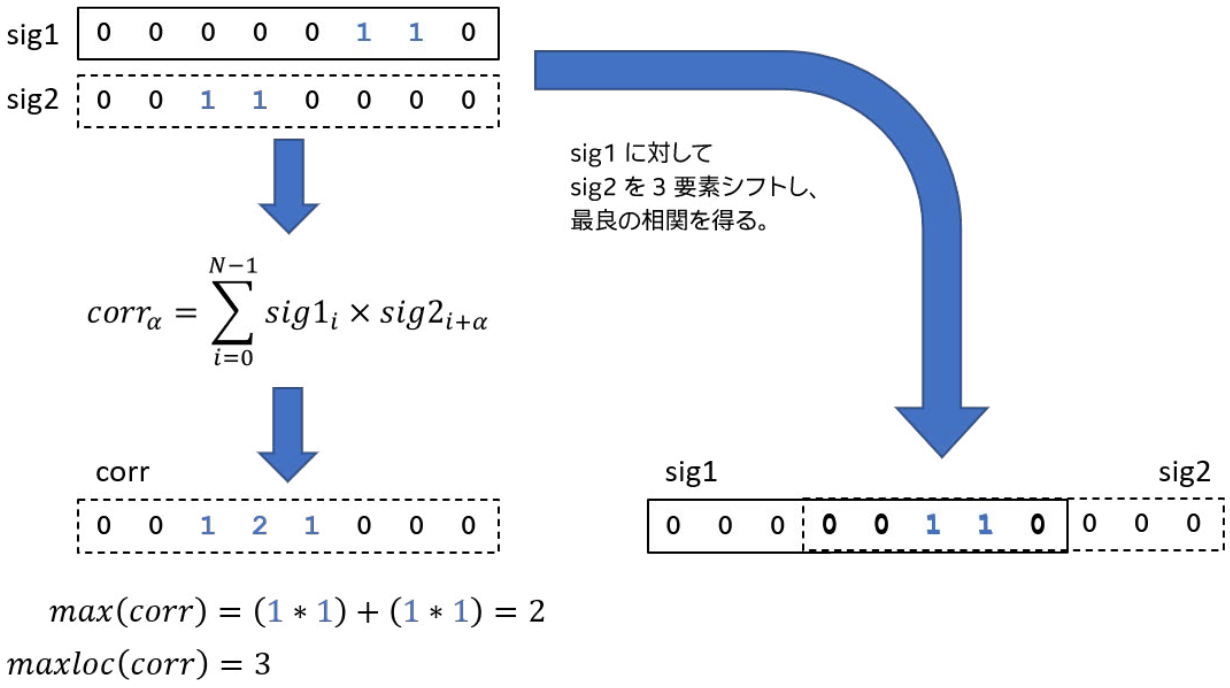

図 1.（2 進数の配列として表された）2 つの離散信号の重なりが最大になる変位を見つける。ここで、α は sig1 に対してシフトされる sig2 の要素の数。相関を計算するとき、2 番目の信号は循環シフトされることに注意。

以前の記事、「oneAPI によるフーリエ相関アルゴリズムの実装」（The Parallel Universe 44 号）では、SYCL* とインテル® oneMKL 関数の組み合わせを使用して 2 つの信号の 1 次元相互相関を計算する方法を紹介しましたが、最後の `maxloc` ステップは省略されていました（**図 2**）。ステップ 1 から 3 はアクセラレーター・デバイスにオフロードされていましたが、ステップ 4 はホスト CPU で計算されていました。つまり、最終的な相関配列をホストに転送する必要があることを意味します。理想的には、信号がデバイスのメモリーにロードされた後に、すべての計算がデバイスで実行されるべきです。ホストで必要なのは、最終的な変位（1 次元相関の 1 つのスカラー）のみです。その他のホストとデバイス間のデータ転送は不要であり、パフォーマンスの低下につながります。そのため、この記事の主題である、デバイス上で `maxloc` を実行するさまざまな方法を実験しました。

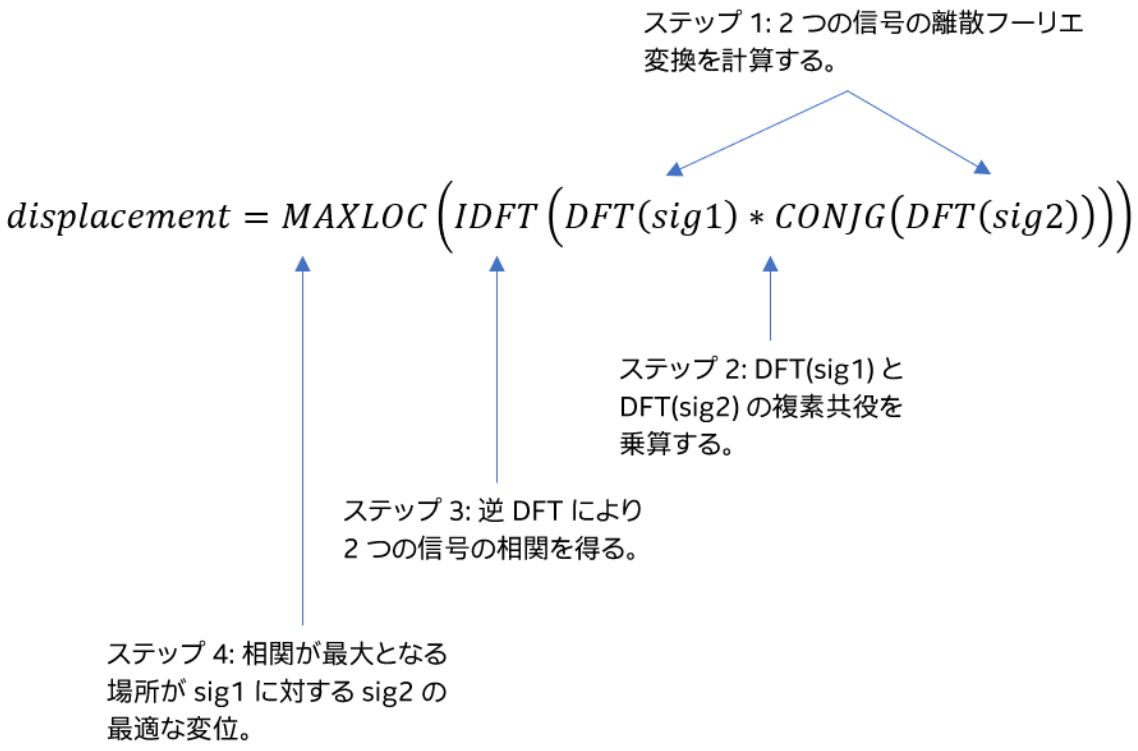

図 2. maxloc リダクションはフーリエ相関アルゴリズムの最後のステップ。DFT は離散フーリエ変換、IDFT は逆 DFT、CONJG は複素共役、MAXLOC は最大相関スコアの場所。

SYCL* のリダクション演算子

リダクションは、複数の値を 1 つの値に減らす一般的な並列パターンです。例えば、総和リダクションは、配列内の値を加算して 1 つの総和を得ます。配列内の最小値や最大値、それらの値の場所を見つけることも、リダクション操作です。SYCL* は、並列カーネルで使用できる組込みのリダクション演算子を提供しています(**図 3**)。

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

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });
    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

図 3. SYCL* の総和リダクションの例。

SYCL* のリダクション・クラスを使用して `maxloc` のようなほかの演算子を実装することもできます（**図 4**）。このコードは、『Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL（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で DPC++ を使いこなす）』（英語）の第 14 章「Common Reduction Patterns（一般的なリダクション・パターン）」（英語）（334 ~ 339 ページ）の `minloc` の例から引用したものです。SYCL* キューは非同期であるため、`wait` 文を使用して続行する前に計算が完了することを保証しています。

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

template <typename T, typename I>
using maxloc = sycl::maximum<std::pair<T, I>>;

constexpr size_t L = 1;

int main(int argc, char **argv)
{
   sycl::queue Q;
   const size_t n = 7;
   float* data = sycl::malloc_shared<float>(n, Q);
   data[0] = 1; data[1] = 1; data[2] = 1; data[3] = 2; data[4] = 1; data[5] = 1; data[6] = 1;

   std::pair<float, int>* max_res = sycl::malloc_shared<std::pair<float, int>>(1, Q);
   std::pair<float, int> max_identity = {
       std::numeric_limits<float>::min(), std::numeric_limits<int>::min()
   };
   *max_res = max_identity;
   auto red_max = sycl::reduction(max_res, max_identity, maxloc<float, int>());

   Q.parallel_for(sycl::nd_range<1>{n, L}, red_max, [=](sycl::nd_item<1> item, auto& max_res) {
       int i = item.get_global_id(0);
       std::pair<float, int> partial = {data[i], i};
       max_res.combine(partial);
   }).wait();

   std::cout << "Maximum value = " << max_res->first << " at element " << max_res->second <<
std::endl;

   sycl::free(data, Q.get_context());
   sycl::free(max_res, Q.get_context());

   return 0;
}
```

図 4. SYCL* のリダクション演算子として maxloc を実装。

これは SYCL* のリダクション･クラスの簡単な例ですが、アーキテクチャーをチューニングする方法はさまざまです。以前の記事、「データ並列 C++ のリダクション操作」(The Parallel Universe 44 号) および「データ並列 C++ のリダクション操作のパフォーマンスを解析する」(The Parallel Universe 45 号) では、さまざまな実装とパフォーマンス特性の例を示しました。組込みのリダクション演算子を使用すれば、最適化の責任を負うのはコンパイラーとライブラリーの開発者です。アプリケーション開発者は、組込みの実装がターゲット CPU やアクセラレーターで優れたパフォーマンスを提供することを期待できます。

oneDPL で同じリダクションを行う

oneDPL は、SYCL* のリダクション・クラスを使用しないで関数を呼び出したいプログラマーに別の方法を提供します。Data Parallel C++（英語）（339 ページ）では次のように説明されています。

> 「C++ 標準テンプレート・ライブラリー（STL）には、並列パターンに対応する複数のアルゴリズムが含まれています…STL のアルゴリズムは通常、イテレーターのペアで指定されたシーケンスに適用されます。C++17 以降は、それらをシーケンシャルに実行するか並列で実行するかを示す実行ポリシー引数をサポートします。oneDPL は、この実行ポリシー引数を活用して、内部の DPC++ で記述されたカーネルを活用する並列プログラミングに生産性の高いアプローチを提供します。STL アルゴリズムの機能のみを使用してアプリケーションを表現できる場合、oneDPL を使用すると、DPC++ カーネルコードを 1 行も記述することなく、システムでアクセラレーターを使用できます。」

この説明には多くのポイントが含まれていますが、2 つのポイントに注目します。

1. **C++ STL :** 関数は C++ プログラマーには馴染み深いものです。
2. **高い生産性 :** コーディングとパフォーマンスの責任は、STL の開発者が負います。アプリケーション開発者は、低レベルの SYCL* カーネルを記述することなくアクセラレーターにアクセスできます。

カーネルベースの `maxloc` コード（**図 4**）と oneDPL 実装（**図 5**）を比較すると、利点が明らかになります。後者は、`max_element` 関数を使用して `maxloc` リダクションを実行します。前の説明で述べたように、SYCL* カーネルは「内部」で活用されます。oneDPL のデフォルトの実行ポリシーは、アクセラレーターが利用可能な場合は、計算をアクセラレーターにオフロードします。利用できない場合は、計算をホスト CPU で実行します。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};

   auto policy = oneapi::dpl::execution::dpcpp _ default;
   auto maxloc = oneapi::dpl::max _ element(policy, data.cbegin(), data.cend());

   std::cout << "Run on "
             << policy.queue().get _ device().template get _ info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data.cbegin(), maxloc) <<
std::endl;

   return 0;
}
```

図 5. oneDPL の max_element と distance 関数、および暗黙的なホストとデバイス間のデータ転送を使用した基本的な maxloc リダクション。

図 5 ではホストとデバイス間のデータ転送が暗黙的に処理されていることに注意してください。計算がアクセラレーターにオフロードされている場合、oneDPL ランタイムは一時バッファーによりデータを自動的にラップします。大規模な oneAPI プログラムで、データがすでに SYCL* バッファーに含まれている場合、oneDPL 関数はバッファーを使用して反復処理を行うことができます（**図 6**）。バッファリングされたデータがデバイスに暗黙的に転送されるか、（SYCL* バッファー経由で）明示的に転送されるかに関係なく、バッファーが変更されない限りホストに転送する必要がないため、不要なオーバーヘッドを回避できます。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   std::vector<int> data{1, 1, 1, 2, 1, 1, 1};
   sycl::buffer<int> data_buf(data);

   auto policy = oneapi::dpl::execution::dpcpp_default;
   auto maxloc = oneapi::dpl::max_element(policy, oneapi::dpl::begin(data_buf), oneapi::dpl::end(data_
buf));

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element "
         << oneapi::dpl::distance(oneapi::dpl::begin(data_buf), maxloc) << std::endl;

   return 0;
}
```

図 6. oneDPL と明示的なホストとデバイス間のデータ転送のために SYCL* バッファリングを使用した maxloc リダクション。

oneDPL 関数は、統合共有メモリー（USM）のポインターを使用してホストとデバイス間のデータ転送を処理することもできます（**図 7**）。この例では、SYCL* の `malloc_shared` 関数と SYCL* キューを使用して、適切な USM に空間を割り当てています。同じキューが oneDPL 実行ポリシーの設定に使用されます。oneDPL アルゴリズムは同期的であるため、**図 5 ~ 7** には明示的な `wait` 文はありません。一方、SYCL* カーネルは非同期です。

```
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/execution>
#include <oneapi/dpl/iterator>
#include <iostream>

int main()
{
   sycl::queue Q(sycl::default_selector{});
   auto policy = oneapi::dpl::execution::make_device_policy(Q);

   const size_t n = 7;
   auto data = sycl::malloc_shared <int>(n, Q);

   data[0] = 1; data[1] = 1; data[2] = 1; data[3] = 2; data[4] = 1; data[5] = 1; data[6] = 1;

   auto maxloc = oneapi::dpl::max_element(policy, data, data + n);

   std::cout << "Run on "
             << policy.queue().get_device().template get_info<sycl::info::device::name>()
             << std::endl;
   std::cout << "Maximum value is at element " << oneapi::dpl::distance(data, maxloc) << std::endl;

   sycl::free(data, Q);
   return 0;
}
```

図 7. oneDPL とホストとデバイス間のデータ転送に USM を使用した maxloc リダクション。

SYCL* リダクションと oneDPL のどちらを使用すべきか

多くのプログラミングの質問と同様に、決定的な答えはありません。どちらのアプローチにも使用する正当な理由があります。どちらを使用するかは要件に依存します。oneDPL が要件を満たすアルゴリズムを提供している場合、標準関数を呼び出すほうが SYCL* カーネルを作成するよりも簡単です。例えば、デバイスメモリーにすでに大きな配列が含まれていて、リダクションからのスカラー出力のみホストで必要な場合は、oneDPL 関数を呼び出すのが最適でしょう。しかし、関数呼び出しは同期的であるため、関数がリターンするまでプログラムがブロックされることに注意する必要があります。SYCL* リダクションは複雑ですが、非同期で柔軟性が高く、より多くのチューニングの機会を提供します。例えば、リダクションを実行しているデータを変換する場合や、複数のリダクションを同時に実行する場合は、SYCL* カーネルを作成することを推奨します。

The Parallel Universe では、アプリケーション開発者とコンパイラー / ライブラリー開発者の間の関心の分離（英語）が繰り返しテーマになっています。後者のグループは、主にパフォーマンスと計算効率に関心があります。前者のグループは、主に問題を可能な限り生産的に解決することに関心があります。このグループに属している場合は、優れたパフォーマンスを発揮する高レベルのプログラミング抽象化を好むでしょう。**図 4** の SYCL* `maxloc` リダクション演算子は複雑で低レベルであり、最高のパフォーマンスを達成するには、アーキテクチャー固有のチューニング（ワークグループのサイズなど）が必要になります。**図 5 ~ 7** の oneDPL の例は単純で、C++ STL ユーザーに馴染み深く、ホストとデバイス間のデータ転送という点で汎用的です。さらに重要なのは、主にパフォーマンスに関心がある oneDPL 製品の開発者がチューニングを行うことです。その結果、oneDPL 関数は、アーキテクチャーに関係なく、優れたパフォーマンスを提供します。これは、ヘテロジニアス・コンピューティングのための oneAPI ソフトウェア抽象化の約束です。

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。