革新的なインテル® oneAPI HPC ツールキットを使用して並列安定ソートのパフォーマンスを最適化する方法

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「How To Optimize A Parallel Stable Sort Performance Using The Revolutionary Intel® oneAPI HPC Toolkit」の日本語参考訳です。

はじめに

この記事では、以前の記事で紹介した並列安定ソートを実装する最新のコードを開発するため、革新的なインテル® oneAPI HPC ツールキットを利用する利点について詳しく解説していきます。

具体的には、SYCL* 実行モデルに基づいて、アプリケーションのワークロード実行をさまざまなハードウェア・アクセラレーターをターゲットにし、CPU、GPU、FPGA などで同じ並列コードを再利用することを可能にする、データ並列 C++ (DPC++) コンパイラーと oneAPI ライブラリーの使い方を習得します。

この記事の目標は、CPU、GPU、そして FPGA などのアクセラレーターを組み合わせて並列計算タスクを実行する (ハードウェア・アーキテクチャーが異なるアクセラレーターで異なるワークロードを実行)、ヘテロジニアス (異種) 計算プラットフォームで並列安定ソートを実装するコードを開発および実行する手法を紹介することです。これにより、ソート処理全体のパフォーマンスが大幅に向上します。

最新の並列コードを示しつつ、oneAPI ライブラリーをインテル® oneAPI ツールキットに同梱されるインテル® oneAPI スレッディング・ビルディング・ブロック (インテル® oneTBB) や Parallel STL などのライブラリーと併用して、ソート処理全体のスケーラビリティーとパフォーマンス・スピードアップを最大化します。OpenMP* のコンパイラー・レベルのプラグマに代わり、インテル® oneTBB ライブラリーを使用して、前述の三元基数クイックソートを実装するコードを現代化する方法を示します。OpenMP* の並列タスクに代わり、tbb::task_group クラスを使用して再帰的な並列サブソートを実行します。CPU で 3 分岐基数クイックソートを実行する並列コードのほかに、oneAPI の SYCL* 実行モデルを使用して配列パーティションを処理するコードを並列化し、CPU だけでなく異なるアーキテクチャーのアクセラレーターで実行する方法についても説明します。

パフォーマンスの最適化とともに、oneAPI の SYCL* 実行モデルの既知の問題と制限、特に安定ソートタスク全体を SYCL* カーネルコードとして実行できない理由を示します。

最後に、この記事で示した並列コードをコンパイルして実行する方法、およびインテル® Xeon® Gold プロセッサー、インテル® HD グラフィックス、またはインテル® プログラマブル・アクセラレーション・カード (インテル® PAC) インテル® Arria® 10 GX FPGA 搭載版など、さまざまなパフォーマンス・アクセラレーション・ハードウェアを使用して、インテル® DevCloud でパフォーマンスを評価する方法を説明します。

インテル® oneTBB ライブラリーを使用する並列 3 分岐基数クイックソート

OpenMP* の並列タスクを使用する並列 3 分岐基数クイックソートの実装については、以前の記事をご覧ください。並列再帰サブソートを実行する特定のコードは、並列同時タスクをサポートする既存のパフォーマンス・ライブラリーやフレームワークのほとんどを使用して実装できるため、同時タスクを採用するアイディアは理想的であると言えます。

しかし、並列同時タスクは、oneAPI ライブラリーと SYCL* 実行モデルでは現在サポートされていません。SYCL* は送信されたカーネルコードごとに 1 つのタスクのみをサポートし、複数の単一タスクが逐次実行されます。また、SYCL* 実行モデルはカーネル内での再帰関数呼び出しをサポートしていないため、結果として、並列再帰サブソートを実行することはできません。

この問題の解決策として、インテル® oneTBB ライブラリーを使用して並列 3 分岐基数クイックソートを実行する別のバージョンのコードを作成します。

```
template<class Container, class _Pred>
void qsort3w(Container& array, std::size_t _First, std::size_t _Last, _Pred compare)
{
    if (_First >= _Last) return;

    std::size_t _Size = array.size(); g_depth++;
    if (_Size > 0)
    {
        std::size_t _Left = _First, _Right = _Last;
        bool is_swapped_left = false, is_swapped_right = false;
        gen::ITEM _Pivot = array[_First];

        std::size_t _Fwd = _First + 1;
        while (_Fwd <= _Right)
        {
            if (compare(array[_Fwd], _Pivot))
            {
                is_swapped_left = true;
                std::swap(array[_Left], array[_Fwd]);
                _Left++; _Fwd++;
            }

            else if (compare(_Pivot, array[_Fwd])) {
                is_swapped_right = true;
                std::swap(array[_Right], array[_Fwd]);
                _Right--;
            }

            else _Fwd++;
        }

        tbb::task_group task_group;
        task_group.run([&]() {
            if (((_Left - _First) > 0) && (is_swapped_left))
                qsort3w(array, _First, _Left - 1, compare);
        });

        task_group.run([&]() {
            if (((_Last - _Right) > 0) && (is_swapped_right))
                qsort3w(array, _Right + 1, _Last, compare);
        });

        task_group.wait();
    }
}
```

このコードでは、OpenMP* のタスク構造の代わりに、tbb::task_group クラスを使用して、並列再帰サブソートを実行する 2 つの同時タスクを生成しています。具体的には、tbb::task_group オブジェクトを宣言し、task_group.run(…) メソッドを呼び出すことで各タスクを生成して、qsort3w(…) 関数の再帰呼び出しを実行するラムダ関数をメソッドの引数として渡します。次のタスクをスポーンした後、task_group.wait() メソッドを呼び出して適切なバリア同期を使用します。

```
tbb::task_group task_group;
task_group.run([&]() {
    if (((_Left - _First) > 0) && (is_swapped_left))
        qsort3w(array, _First, _Left - 1, compare);
});

task_group.run([&]() {
    if (((_Last - _Right) > 0) && (is_swapped_right))
        qsort3w(array, _Right + 1, _Last, compare);
});

task_group.wait();
```

また、tbb::blocked_range<RanIt> などのイテレーターの範囲はインテル® oneTBB ライブラリー API ではサポートされないため、ランダム・アクセス・イテレーター型には可変長テンプレートを使用する必要はありません。代わりに、std::size_t 型のインデックスを使用します。

並列処理を呼び出すには qsort3w(…) 関数を別タスクとして生成し、個別のスレッドでソート処理全体を実行します。

```
    template<class Container, class _Pred >
    void parallel_sort(Container& array, std::size_t _First, std::size_t _Last, _Pred
compare)
    {
        g_depth = 0L;
        tbb::task_group task_group;
        task_group.run([&]() {
            internal::qsort3w(array, _First, _Last - 1, compare);
        });

        task_group.wait();
    }
```

最後に、CPU ハードウェア・ターゲットで次のコードを使用してソートを実行します。

CPU、GPU、および FPGA 上で並列安定ソートを実行

以前の記事で説明したように、並列安定ソートを実行するには、最初に internal::parallel_sort(…) を呼び出してオブジェクトの配列をキーでソートします。この記事で紹介する基本概念では、実際のソートは通常 CPU ターゲットで実行されます。

次に、配列を同じキーを持つオブジェクトのサブセットに分割する処理は、マルチコア CPU ターゲットとは異なる別のハードウェア・ターゲットで並行して実行できます。これには、データ並列 C++ コンパイラーと oneAPI の OpenCL*/SYCL* ラッパー・ライブラリーを使用して、各種ターゲットで実行される間、並列に配列パーティションを実行する特定のカーネルを実装するコードを作成します。

```
    template<class Container, class _CompKey, class _CompVals>
    void parallel_stable_sort(Container& array, sycl::queue device_queue,
        _CompKey comp_key, _CompVals comp_vals)
    {
        std::vector<std::size_t> pv;
        pv.resize(array.size()); pv[0] = 0;

        tbb::task_group task_group;
        task_group.run([&]() {
            internal::parallel_sort(array, 0L, array.size(), comp_key);
        });

        task_group.wait();

        cl::sycl::buffer<std::size_t> pv_buf{ pv.data(), array.size() };
        cl::sycl::buffer<gen::ITEM> array_buf{ array.data(), array.size() };

        device_queue.submit([&](sycl::handler& cgh) {
            auto pv_acc = pv_buf.get_access<cl::sycl::access::mode::write>(cgh);
            auto array_acc = array_buf.get_access<cl::sycl::access::mode::read>(cgh);
            cgh.parallel_for<class partition_kernel>(cl::sycl::range<1>{array.size() - 2},
                [=](cl::sycl::id<1> idx) {
                    if ((comp_key(array_acc[idx[0]], array_acc[idx[0] + 1]) ||
                         (comp_key(array_acc[idx[0] + 1], array_acc[idx[0]])))) {
                        pv_acc[idx[0] + 1] = idx[0] + 1;
                    }
                });
        }).wait();

        pv_buf.get_access<cl::sycl::access::mode::read>();

        auto _LastIt = std::remove_if(dpstd::execution::par_unseq,
            pv.begin() + 1, pv.end(), [&](const std::size_t index) { return index ==
0L; });

        pv.resize(std::distance(pv.begin(), _LastIt));
        pv.push_back(std::distance(array.begin(), array.end()));

        tbb::parallel_for(tbb::blocked_range<std::size_t>(0, pv.size() - 1), \
            [&](const tbb::blocked_range<std::size_t>& r) {
                for (std::size_t index = r.begin(); index != r.end(); index++)
                    internal::parallel_sort(array, pv[index], pv[index + 1], comp_vals);
            });
    }
```

パーティション化のアルゴリズムを実装するコードは、通常、データフローの依存関係やその他の問題が生じることが少ないため、各種ハードウェア・ターゲットで容易に並列化して実行できます。ただし、パーティション化処理のアルゴリズム・レベルには重要な最適化が 1 つあります。通常、SYCL* 実行モデルは、単純な C/C++ の配列、またはより複雑な std::array<> と std::vector<> STL コンテナーなどの各種バッファーに格納されたデータに、アクセサーを使用してアクセスできます。ただし、実行中のカーネル内から std::vector<>::push_back(…) メソッドを使用してバッファーサイズを変更したり、新しい要素を追加する際に動的バッファーの再割り当てを行うことはできません。

そのため、特定のカーネルをアクセラレーターで実行する前に、ホスト CPU で実行されているコードでバッファーを割り当てる std::vector<std::size_t> pv などのパーティション・インデックスのベクトルを宣言する必要があります。

その後、次のコードで示すように、デバイスセレクターとカーネル実行キュー・オブジェクトをインスタンス化する必要があります。

```
    default_selector device_selector;
    queue device_queue(device_selector);
```

SYCL* 実行モデルは、デバイスセレクター数を提供します。それぞれのハードウェア・アクセラレーション・プラットフォームごとに 1 つのデバイスセレクターがあります。

sycl::default_selector {}	インテル® FPGA エミュレーター・アプリケーション
sycl::cpu_selector {}	インテル® CPU
sycl::gpu_selector {}	インテル® GPU
sycl::intel::fpga_selector {}	インテル® FPGA アクセラレーター
sycl::intel::fpga_emulator_selector {}	インテル® FPGA エミュレーター

ここでは、インテル® FPGA エミュレーター・ターゲットで特定のカーネルを実行する default_selector{} オブジェクトを使用します。

パーティション・インデックスのベクトルと、カーネル内で実行されるコードでソートされた配列へのアクセスを提供するため、2 つの cl::sycl::buffer<> クラス・オブジェクトを宣言します。コンストラクターは通常、コンテナー・オブジェクトとバッファーサイズのどちらかの引数を受け入れます。

```
cl::sycl::buffer<std::size_t> pv_buf{ pv.data(), array.size() };
cl::sycl::buffer<gen::ITEM> array_buf{ array.data(), array.size() };
```

次に、特定のカーネルコードを SYCL* 実行キューに送信するコードを実装します。これには、通常、device_queue オブジェクトと device_queue.submit(…) メソッド呼び出しを使用します。このメソッドは、カーネルコードが実装されるラムダ関数の単一引数を受け入れます。ラムダ関数の引数は sycl::handler 型のオブジェクトです。このオブジェクトを使用して、特定のコードを並列実行するメソッドにアクセスしたり、カーネル内からデータバッファーにアクセスできます。また、特定のカーネル実行のバリア同期を行う必要があります。これには、device_queue.submit(...) メソッドの後に呼び出される device_queue.wait() メソッド呼び出しを使用します。

```
device_queue.submit([&](sycl::handler& cgh) {
      // SYCL kernel code...
}).wait();
```

次のラムダ関数のスコープでは、カーネルコードが特定のバッファーにアクセスする際に使用する 2 つのアクセサーをインスタンス化します。

```
auto pv_acc = pv_buf.get_access<cl::sycl::access::mode::write>(cgh);
auto array_acc = array_buf.get_access<cl::sycl::access::mode::read>(cgh);
```

アクセサー・オブジェクトは、cl::sycl::buffer<>::get_access(…) メソッドを使用してインスタンス化され、sycl::handler オブジェクトの単一の引数を受け入れます。

次に、cgh.parallel_for(…) メソッドを呼び出して、パーティション化が行われる反復中に、単一のループを並列に実行します。この方法は、OpenMP* ディレクティブ構造を使用して小さなループを並列化するのによく似ています。ループのそれぞれの反復は、特定のスレッドによって明示的に並列実行されます。

cgh.parallel_for(…) メソッドは、ループの反復中に呼び出される cl::sycl::nd_range<Dims> オブジェクト、またはラムダ関数のいずれかの引数を受け入れます。cl::sycl::nd_range<Dims> オブジェクトは、cgh.parallel_for(…) メソッドによって実行される反復回数を指定します。ラムダ関数は、cl::sycl::id<1> オブジェクトの単一引数を受け入れて、各反復のインデックス値を取得します。また、cl::sycl::nd_range<1> と cl::sycl::id<1> は、実行モデル内の次元数のテンプレート・パラメーターを受け入れます。ここでは、行列やその他の空間データを処理するのではなく、1 次元のベクトルに格納されたデータをソートするため、1 次元の実行モデルを使用します。

```
cgh.parallel_for<class partition_kernel>(cl::sycl::range<1>{array.size() - 2},
    [=](cl::sycl::id<1> idx) {
        if ((comp_key(array_acc[idx[0]], array_acc[idx[0] + 1]) ||
             (comp_key(array_acc[idx[0] + 1], array_acc[idx[0]])))) {
            pv_acc[idx[0] + 1] = idx[0] + 1;
        }
    });
```

この記事で説明するパーティション化アルゴリズムの別形では、並列実行されるパーティション化グループの各反復で、現在のオブジェクトのキー idx[0] を隣接する idx[0] + 1 インデックスのオブジェクトと比較します。キーが等しくない場合、カーネル内部から書き込みアクセスされるベクトル pv の idx[0] + 1 位置に idx[0] + 1 インデックスを割り当てます。それ以外の場合、idx[0] + 1 位置にゼロ値を割り当てます。前述のパーティション化アルゴリズムの別形とは異なり、ソートされる配列サイズに等しい初期容量のインデックスのベクトルを使用するため、通常は O(N) の追加空間が必要です。N は、ソートされる配列内のオブジェクト数です。

カーネル実行の最後で、更新されたベクトル pv をホスト CPU で実行されるコードに返します。これは、次のメソッドを呼び出すことで行われます。

```
pv_buf.get_access<cl::sycl::access::mode::read>();
```

インデックス pv のベクトルがカーネルから正しく返されたら、パーティション化アルゴリズムの相対的順序を維持し、値がゼロに等しい要素をすべて削除してベクトルを正規化する必要があります。

```
auto _LastIt = std::remove_if(dpstd::execution::par_unseq,
     pv.begin() + 1, pv.end(), [&](const std::size_t index)
                 { return index == 0L; });

 pv.resize(std::distance(pv.begin(), _LastIt));
 pv.push_back(std::distance(array.begin(), array.end()));
```

そのため、Parallel STL 実行ポリシー dpstd::execution::par_unseq を適用して並列実行する std::remove_if(…) ルーチンを使用します。

また、パーティション化を行うコードをアルゴリズム・レベルで最適化する必要があります。cgh.parallel_for(…) メソッドで実行される反復は適切に同期されているため、インデックスのベクトル pv をソートしてその順番を維持する必要はありません。そのため、アルゴリズム・レベルでのソート全体のパフォーマンスが向上します。

最後に、オブジェクトの配列を値でソートする必要があります。ベクトル pv のインデックスのそれぞれのペアを反復処理するには、tbb::parallel_for(…) を使用します。これにより、ホスト CPU で各反復を並列に実行できます。このメソッドは、tbb::blocked_range<> オブジェクト、またはループ反復中に実行されるラムダ関数のどちらかの引数を受け入れます。

```
tbb::parallel_for(tbb::blocked_range<std::size_t>(0, pv.size() - 1), \
    [&](const tbb::blocked_range<std::size_t>& r) {
        for (std::size_t index = r.begin(); index != r.end(); index++)
            internal::parallel_sort(array, pv[index], pv[index + 1], comp_vals);
    });
```

tbb::blocked_range<std::size_t> オブジェクトは、std::size_t 型のブロック範囲イテレーターの [0..pv.size() – 1] 範囲を維持するために使用されます。ラムダ関数では、各反復間にループを実装します。このループでは、ブロック p 範囲のイテレーターでアクセスされるベクトル pv からインデックスのペアがフェッチされ、各反復の最後でインクリメントされます。インデックスのペアを使用して [pv[index]..pv[index+1]] の範囲を管理し、internal::parallel_sort() 関数を呼び出して、インデックス範囲に続く配列内のすべてのオブジェクトを個別にソートします。

カーネルが実行するコードでの仮想関数呼び出しには制限があり、SYCL* カーネルコードからインテル® oneTBB API を使用することができないため、次のコードはホスト CPU で実行されます。

インテル® DevCloud でサンプルプログラムをビルドして実行

ヘテロジニアス・プラットフォームで実行可能な最新の並列コードを作成し、さまざまなハードウェア・アクセラレーション・ターゲット (CPU、GPU、FPGA など) で実行してパフォーマンスを評価します。

インテル® DevCloud は、インテル コーポレーションが提供するクラスター・プラットフォーム・ソリューションであり、革新的なインテル® HD グラフィックス、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 ハードウェア・アクセラレーター、インテル® Xeon® プロセッサー・ファミリーなど、最新のインテルのハードウェアとソフトウェアを使用して、さまざまな並列アプリケーションのワークロードを実行、テスト、および評価することができます。

並列安定ソートを実装するコードのパフォーマンスについて説明する前に、oneAPI プロジェクトをインテル® DevCloud でビルドして実行する方法を確認しておきましょう。

最初に行うことは、https://intelsoftwaresites.secure.force.com/devcloud/oneapi (英語) で oneAPI プロジェクト向けのインテル® DevCloud にサインアップすることです。

インテル® DevCloud でのサインアップが完了したら、https://devcloud.intel.com/oneapi/connect (英語) で説明されているように、Jupyter* Notebook を使用して Linux* ターミナルにログインする必要があります。また、この記事の最後にあるプロジェクトのダウンロード・リンクからアーカイブをダウンロードして展開してください。次に、プロジェクトのフォルダーをクラウドにアップロードします。

プロジェクトが正常にアップロードされたら、並列安定ソート・アプリケーションの実行に使用する計算ノードを確保する必要があります。これには、通常、Linux* ターミナルで次のコマンドを実行します。

インテル® Xeon® Gold 6128 CPU	qsub -I (capital "i") -l (lower-case "L") nodes=1:cpu:ppn=2
第 9 世代インテル® グラフィックス NEO	qsub -I (capital "i") -l (lower-case "L") nodes=1:gpu:ppn=2
インテル® PAC プラットフォーム・コンパイル	qsub -I (capital "i") -l (lower-case "L") nodes=1:fpga_compile:ppn=2
インテル® PAC プラットフォーム・ランタイム	qsub -I (capital "i") -l (lower-case "L") nodes=1:fpga_runtime:ppn=2

最後に、プロジェクト・フォルダーに移動して、`make` コマンドを実行してビルドを行います。

```
cd ./parallel_stable_sort_XXXX && make
```

プロジェクトの実行可能ファイルが正常にビルドされたら、次のコマンドのように並列ソート・アプリケーションを実行します。

```
./parallel_stable_sort_XXXX
```

これで、ヘテロジニアス・プラットフォームで並列安定ソートを実行してパフォーマンスを評価できます。

パフォーマンスの評価

以下は、並列コードのパフォーマンス・スピードアップを調査するため使用したハードウェア・アクセラレーション・ターゲットごとの、std::sort() ライブラリー関数のパフォーマンスと比較した結果です。

インテル® Xeon® Gold 6128 CPU @ 3.47GHz/192GB RAM	2x-11x 高速
インテル® FPGA エミュレーター・アプリケーション	4x-6x 高速
第 9 世代インテル® グラフィックス NEO	4x-7x 高速
インテル® PAC プラットフォーム/インテル® Arria® 10 GX	3x-8x 高速

添付ファイル	サイズ
parallel-stable-sort-oneapi.zip	7.9KB
parallel-stable-sort-oneapi-exe.zip	641.1KB
parallel-stable-sort-devcloud.zip	20.7KB

製品とパフォーマンス情報

[1] 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