データ並列 C++ のリダクション操作

一般的なリダクション並列パターンの実装とチューニング

Ramesh Peri インテル・コーポレーション シニア主席エンジニア

リダクション操作の概要

リダクションは、並列プログラミングにおける一般的な操作で、配列の要素を 1 つの結果にレデュースします。例えば、大きな配列の要素の総和を求める場合（**図 1**）、操作を並列に実行するには、部分和を計算し、それを順次組み合わせて最終的な結果を算出する必要があります。リダクション演算子（総和、最小値、最大値、最小位置、最大位置など）は結合的であり、多くの場合、可換的です。リダクションの実装方法は多数あり、そのパフォーマンスはプロセッサー・アーキテクチャーに依存します。この記事では、データ並列 C++（DPC++）でリダクションを表現するいくつかの方法を紹介し、それぞれのパフォーマンスへの影響について考察します。

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。

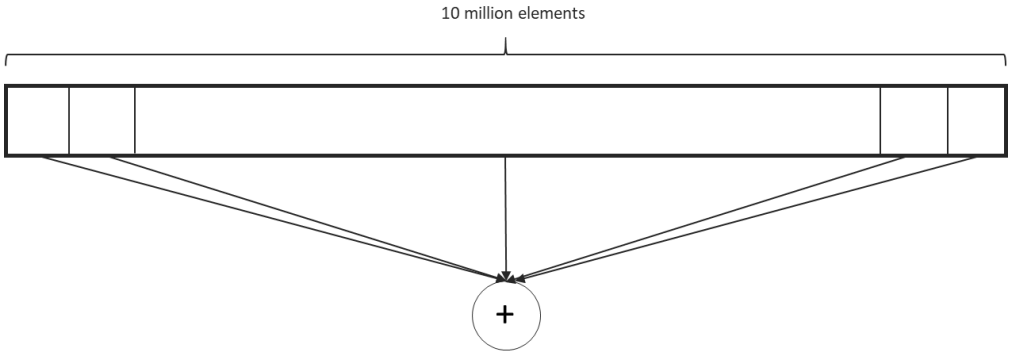

図 1. 総和リダクションの図

グローバルアトミックを使用した DPC++ のリダクション

以下の実装では、DPC++ カーネルの各ワークアイテムが入力配列の要素を処理して、グローバル変数をアトミックに更新します。

```
void reductionAtomics1(sycl::queue &q,
                       sycl::buffer<int> inbuf,
                       int &res,
                       int size) {
   const size_t data_size = inbuf.get_size() / sizeof(int);
   int num_work_items = data_size;
   sycl::buffer<int> sum_buf(&res, 1);
   q.submit([&](auto &h) {
      sycl::accessor buf_acc(inbuf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);

      h.parallel_for(num_work_items, [=](auto index) {
        size_t glob_id = index[0];
        auto v = sycl::ONEAPI::atomic_ref<int,
                  sycl::ONEAPI::memory_order::relaxed,
                  sycl::ONEAPI::memory_scope::device,
                  sycl::access::address_space::global_space>(sum_acc[0]);
        v.fetch_add(buf_acc[glob_id]);
      });
});
```

コンパイラーが生成するスレッドの数（コンパイラーが選択するターゲットデバイスのデフォルトのワークグループとサブグループのサイズに依存）によっては、単一のグローバル変数である **sum_buf** へのアクセス競合が頻繁に発生します。一般的に、このようなソリューションのパフォーマンスはあまり高くありません。

グローバルアトミックの競合の軽減

グローバル変数の更新時の競合を軽減する方法として、この変数にアクセスするスレッドの数を減らすことが挙げられます。これは、各ワークアイテムが配列の複数の要素を処理し、要素のチャンクに対してローカルなリダクションを実行した後、グローバルアトミック更新を実行することで実現できます（**図 2**）。

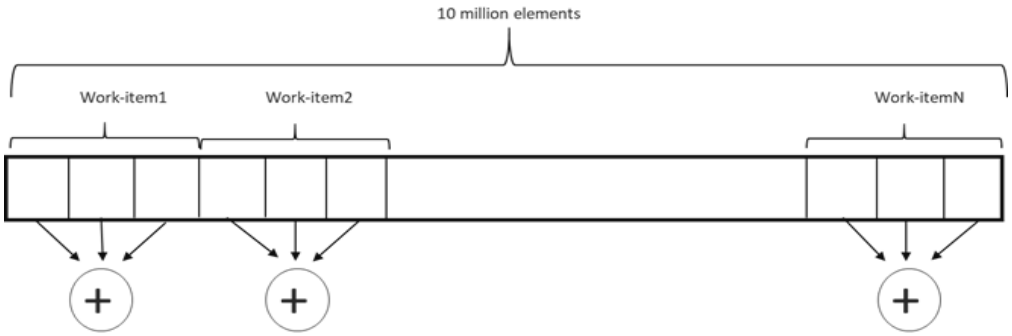

図 2. 各ワークアイテムは部分和を計算

以下は、この実装のコードです。

```
void reductionAtomics2(sycl::queue &q,
                       sycl::buffer<int> inbuf,
                       int &res) {
   const size_t data_size = inbuf.get_size() / sizeof(int);
   sycl::buffer<int> sum_buf(&res, 1);
   int num_work_items =
       q.get_device().get_info<sycl::info::device::max_compute_units>();
   int BATCH = (data_size + num_work_items - 1) / num_work_items;
   q.submit([&](auto &h) {
      sycl::accessor buf_acc(inbuf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);

      h.parallel_for(num_processing_elements, [=](auto index) {
        size_t glob_id = index[0];
        size_t start = glob_id * BATCH;
        size_t end = (glob_id + 1) * BATCH;
        if (end > N)
          end = N;
        int sum = 0;
        for (size_t i = start; i < end; i++)
          sum += buf_acc[i];
          auto v = sycl::ONEAPI::atomic_ref<int,
                    sycl::ONEAPI::memory_order::relaxed,
                    sycl::ONEAPI::memory_scope::device,
                    sycl::access::address_space::global_space>(sum_acc[0]);
        v.fetch_add(sum);
      });
    });
}
```

この実装は、各ワークアイテムが連続するメモリー位置にアクセスするためあまり効率的ではなく、コンパイラーは非効率なコードを生成します。DPC++ コンパイラーは各ワークアイテムをベクトルレーンのように扱うため、連続するメモリー位置にアクセスするワークアイテムは非効率的になります。

ワークアイテムによる効率良いメモリーアクセス

DPC++ コンパイラーは、各ワークアイテムをプロセッサーのベクトルレーンにマップするため、ワークアイテムがプロセッサーのベクトル長よりも大きなストライドでメモリー位置にアクセスする場合、コンパイラーは効率的なコードを生成できます。このアクセスパターンを**図 3** に示します。ラベル w_{i-1} … w_{i-n} は n 個のワークアイテムを示します。

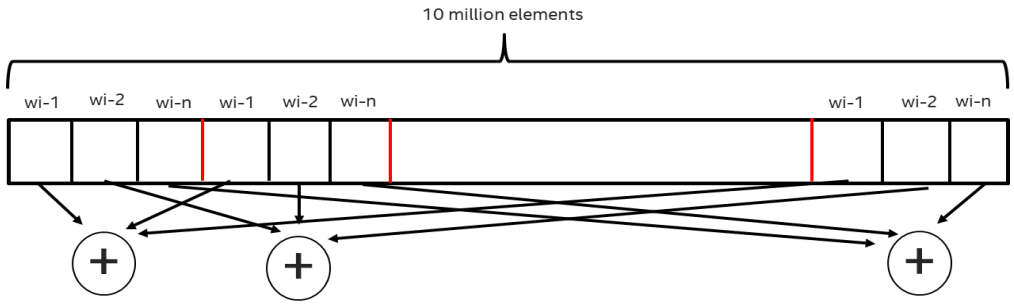

図 3. 各ワークアイテムは連続しないメモリー位置から部分和を計算

以下は、各ワークアイテムが入力ベクトルの複数の要素をインターリーブして処理するカーネルのコードです。

```
void reductionAtomics3(sycl::queue &q,
                       sycl::buffer<int> inbuf,
                       int &res) {
   const size_t data_size = inbuf.get_size() / sizeof(int);
   sycl::buffer<int> sum_buf(&res, 1);
   int num_work_items =
    q.get_device().get_info<sycl::info::device::max_compute_units>() *
    q.get_device().get_info<sycl::info::device::native_vector_width_int>();
   q.submit([&](auto &h) {
      sycl::accessor buf_acc(inbuf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);

      h.parallel_for(num_work_items, [=](auto index) {
        size_t glob_id = index[0];
        int sum = 0;
        for (size_t i = glob_id; i < data_size; i += num_work_items)
          sum += buf_acc[i];
        auto v = sycl::ONEAPI::atomic_ref<int,
                 sycl::ONEAPI::memory_order::relaxed,
                 sycl::ONEAPI::memory_scope::device,
                 sycl::access::address_space::global_space>(sum_acc[0]);
        v.fetch_add(sum);
      });
  });
}
```

ワークアイテム数の選択は重要です。処理要素の数と処理要素に最適なベクトル幅の積になるように選択します。あるプラットフォームでは十分であっても、サポートするスレッド数が処理要素の数よりもはるかに多いプラットフォームでは十分ではない場合があるため、ワークアイテム数はターゲット・プラットフォームに応じて慎重に選択する必要があります。

ツリー・リダクション

ツリー・リダクションは、カーネル内の各ワークアイテムが、隣接する要素にリダクション演算子を適用し、複数のレベルの中間結果を生成するポピュラーな手法です。これは、**図 4** に示すように、スレッドのスケジュールと同期が高効率であることから、ハードウェア・サポートを利用してワークグループ内で適用できます。

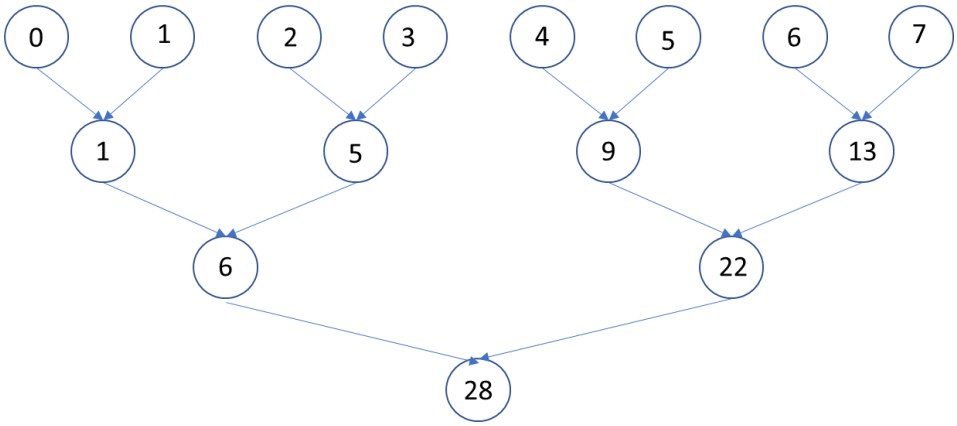

図 4. ツリー・リダクション

ワークグループのサイズはかなり小さく（約 256 または 512）、ハードウェア・デバイスに依存しますが、レデュースする要素の数は、ワークループのサイズよりも 1 桁大きくなります。つまり、各ワークグループで実行されるリダクション操作は、さらにリダクションが必要な中間結果を生成します。これに対応するには、2 つの方法があります。

1. 以下に示すように、各ワークグループにおいて中間結果でグローバルアトミック加算を呼び出します。

```
void buf2finalReduction(sycl::queue &q,
                    sycl::buffer<int> inbuf,
                    int &res) {
   const size_t num_work_items = inbuf.get_size() / sizeof(int);
   int work_group_size =
      q.get_device().get_info<sycl::info::device::max_work_group_size>();
   sycl::buffer<int> sum_buf(&res, sizeof(int));
   q.submit([&](auto &h) {
      sycl::accessor buf_acc(inbuf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::write_only, sycl::noinit);
      sycl::accessor<int, 1, sycl::access::mode::read_write,
              sycl::access::target::local> scratch(work_group_size, h);

      h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
         [=](sycl::nd_item<1> item) {
            size_t global_id = item.get_global_id(0);
            int local_id = item.get_local_id(0);
            int group_id = item.get_group(0);
            int sum = 0;

         if (global_id < data_size)
             scratch[local_id] = buf_acc[global_id];
         else
             scratch[local_id] = 0;
         for (int i = work_group_size / 2; i > 0; i >>= 1)
             item.barrier(sycl::access::fence_space::local_space);
             if (local_id < i)
                scratch[local_id] += scratch[local_id + i];
         }

         if (local_id == 0) {
            auto v = sycl::ONEAPI::atomic_ref<int,
   sycl::ONEAPI::memory_order::relaxed,sycl::ONEAPI::memory_scope::device,
   sycl::access::address_space::global_space>(sum_acc[0]);
            v.fetch_add(scratch[0]);
         }
       });
});
```

2. ワークグループによって生成された中間値のリストに対して、同じリダクション・カーネルを再度呼び出して、別の中間値のセットを生成します。そして、中間結果の数が十分に少なくなった時点で、最終結果を得るためカーネルを呼び出す代わりに、グローバルアトミック更新を使用できます。

   この手法を実装するカーネルを以下に示します。ここでは、カーネルの出力は各ワークグループが出力バッファーに格納する別の値のセットです。

```
void buf2bufReduction(sycl::queue &q,
                      sycl::buffer<int> inbuf,
                      sycl::buffer<int> outbuf) {
   const size_t data_size = inbuf.get_size() / sizeof(int);
   int num_work_items = data_size;
   int work_group_size =
      q.get_device().get_info<sycl::info::device::max_work_group_size>();
   q.submit([&](auto &h) {
      sycl::accessor inbuf_acc(inbuf, h, sycl::read_only);
      sycl::accessor outbuf_acc(outbuf, h, sycl::write_only, sycl::noinit);
      sycl::accessor<int, 1, sycl::access::mode::read_write,
          sycl::access::target::local> scratch(work_group_size, h);

      h.parallel_for(sycl::nd_range<1>(num_work_items, work_group_size),
         [=](sycl::nd_item<1> item) {
            size_t global_id = item.get_global_id(0);
            int local_id = item.get_local_id(0);
            int group_id = item.get_group(0);
            int sum = 0;

            if (global_id < data_size)
                scratch[local_id] = buf_acc[global_id];
            else
                scratch[local_id] = 0;
            for (int i = work_group_size / 2; i > 0; i >>= 1)
               item.barrier(sycl::access::fence_space::local_space);
                if (local_id < i)
                   scratch[local_id] += scratch[local_id + i];
         }
         if (local_id == 0)
            outbuf_acc[group_id] = scratch[0];
       });
   });
}
```

前の 2 つのカーネルを以下のように呼び出すことで、最終的な結果が得られます。ここで、**X** は最終的なアトミックベースのリダクションが呼び出される中間結果のサイズであり、この値はグローバル・アトミック・リダクションの効率に基づいて選択されます。

```
int work_group_size =
   q.get_device().get_info<sycl::info::device::max_work_group_size>();
int ibufsize = (size + work_group_size - 1) / work_group_size;
sycl::buffer<int> *prev = &buf;
while (ibufsize > X) {
   int *bufptr=(int *)malloc(sizeof(int) * ibufsize);
   sycl::buffer<int> *cur = new sycl::buffer<int>(bufptr, ibufsize);
   ibufsize = (ibufsize + work_group_size-1) / work_group_size;
   buf2bufReduction(q, *prev, *cur);
   prev = cur;
}
buf2finalReduction(q, *prev, res);
```

DPC++ のビルトイン・リダクション演算子

DPC++ は、以下のカーネルで使用されている総和リダクション演算子を提供しています。この場合、コンパイラーはプラットフォームにとって最も効率良い実装を選択します。通常は、ビルトイン・リダクション演算子を使用することを推奨します。

```
void builtinReduction(sycl::queue &q,
                      sycl::buffer<int> inbuf,
                      int &res) {
   sycl::buffer<int> sum_buf(&res,1);
   q.submit([&](auto &h) {
      sycl::accessor buf_acc(inbuf, h, sycl::read_only);
      sycl::accessor sum_acc(sum_buf, h, sycl::read_write);
      auto sumr = sycl::ONEAPI::reduction(sum_acc, sycl::ONEAPI::plus<>());
      h.parallel_for(sycl::nd_range<1>{data_size, 256}, sumr,
           [=](sycl::nd_item<1> item, auto &sumr_arg) {
               int glob_id = item.get_global_id(0);
               sumr_arg += buf_acc[glob_id];
      });
  });
}
```

まとめ

リダクションは、多くのアプリケーションで使用される一般的な並列プログラミング・パターンです。この記事では、DPC++ でこの操作を実装するいくつかの方法を紹介しました。これらの実装のパフォーマンスは、コンパイラーやターゲット・プラットフォームによって大きく異なります。次回の記事では、これらのカーネルの CPU および GPU 上でのパフォーマンスを調査する予定です。

関連情報

- **oneAPI** (英語)
- **インテル® oneAPI ツールキット**
- **『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)