OpenMP* ターゲット オフロードの事例

ISO Fortran がヘテロジニアス・コンピューティングに十分でない理由

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
Ron Green インテル コーポレーション コンパイラー・エンジニアリング・マネージャー
Nawal Copty インテル コーポレーション シニア・ソフトウェア・エンジニア

以前の記事では、Fortran プログラムからアクセラレーターへの計算のオフロードについて紹介しました。

- Fortran、oneMKL、OpenMP* を使用して LU 因数分解を高速化
- Fortran と OpenMP* でヘテロジニアス・プログラミングの課題を解決
- oneMKL と OpenMP* ターゲットオフロードで線形システムを解く
- Fortran の DO CONCURRENT を使用したアクセラレーター・オフロード

この記事では、アクセラレーターへのオフロードに関して、Fortran `DO CONCURRENT` 文と OpenMP* `target` 構文の長所と短所を説明します。

`DO CONCURRENT` 構文は ISO Fortran 2008 で追加された機能で、`DO CONCURRENT` ループの反復が独立していて、任意の順序で実行できる（訳注：すなわち、並列に実行できる）ことをコンパイラーに通知またはアサートします。`DO CONCURRENT` ループはシーケンシャルに、あるいは並列に実行でき、OpenMP* バックエンドを使用して `DO CONCURRENT` ループをアクセラレーターにオフロードすることもできます。アクセラレーター・オフロード機能は 2013 年の OpenMP* バージョン 4.0 で追加されました。OpenMP* `target` ディレクティブを使用すると、プログラマーは、アクセラレーターで実行するコードの領域や、ホスト・プロセッサーとアクセラレーター間で転送するデータを指定できます。

アクセラレーター・オフロードのどちらのアプローチも、標準準拠のコンパイラーを備えた任意のシステムに移植できます。`DO CONCURRENT` は、簡潔な ISO Fortran 構文であるという長所があります。ただし、ISO Fortran には ISO C++ と同じいくつかの制限があります（編集者注：ヘテロジニアス並列処理に関する ISO C++ の制限については、「SYCL* の事例」を参照してください）。デバイスや不連続メモリーの概念がないため、制御フローをアクセラレーターに転送したり、ホストとデバイス間のデータ転送を制御する標準的な方法はありません。OpenMP* は、これらの制限に対処します。OpenMP* は ISO 標準ではありませんが、25 年以上にわたり成熟してきたオープンな業界標準です。OpenMP* `target` ディレクティブは冗長であり、プログラマーがコンパイラーに並列処理であることを説明する必要がありますが、次のコード例から分かるように、ホストとデバイス間のデータ転送を細かく制御し、並列領域を集約して効率を向上させることができます。

前号の記事「Fortran の DO CONCURRENT を使用したアクセラレーター・オフロード」では、単純なフィルターをバイナリーイメージに適用したエッジ検出について説明しました。今回は、より現実的なエッジ検出アルゴリズムを比較に使用します。Fortran `DO CONCURRENT` と OpenMP* `target` ディレクティブを使用して Sobel アルゴリズムを実装します。このアルゴリズムは、オリジナルの画像の各ピクセルに水平フィルターと垂直フィルターを適用して、変換された画像内に急激なピクセル強度の変化を抽出します。

−1	0	1
−2	0	2
−1	0	1

水平

−1	−2	−1
0	0	0
1	2	1

垂直

変換前と変換後の例を**図 1** に示します。各ピクセルの演算は独立しているため、アルゴリズムは高度にデータ並列です。

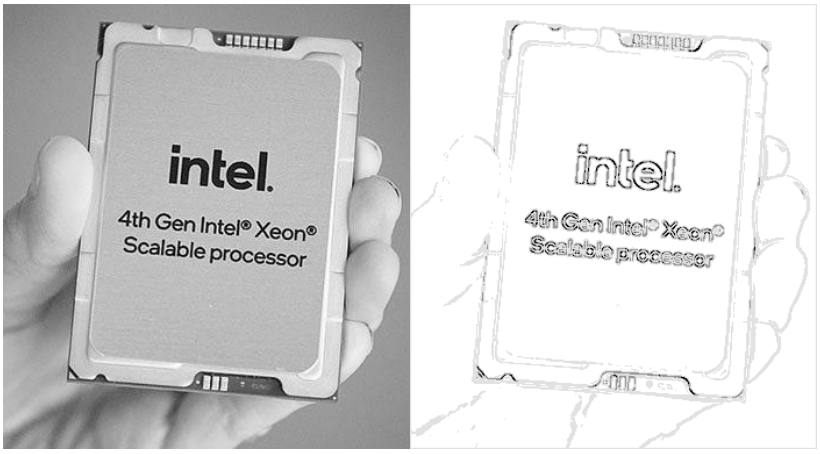

図 1. Sobel エッジ検出

Sobel アルゴリズムは通常、画像のスムージング、画像のエッジ検出、エッジのハイライトの、順に行う 3 つのステップで実装されます（編集者注 : 読者の方々は、以前の記事「ArrayFire と oneAPI による 2 次元フーリエ相関アルゴリズムの高速化」などから、私が単純なアルゴリズムよりも同期とデータの依存関係を強調できるマルチステップのアルゴリズムを支持していることをご存知でしょう）。これらのステップは、3 つの Fortran `DO CONCURRENT` ループを使用して簡単にコーディングできます（**図 2**）。各ループでは、画像のサイズに応じて大幅なデータ並列処理が行われます。この例では 2D 画像を想定していますが、アルゴリズムはボリューム画像にも拡張できます。

`DO CONCURRENT` 構文は、`DO` 構文の異種形式にすぎません。`DO CONCURRENT` を初めて見た場合でも、ほとんどの Fortran プログラマーは、この例が二重に入れ子にされた `DO` ループのように画像の列と行を反復していることを理解できるでしょう。`DO CONCURRENT` ループには、一部の反復をマスクするプレディケートや変数のスコープを定義する追加の節を含めることができます（例えば、**図 2** の 2 つ目のループにはリダクション操作が含まれています）。

ピクセル値を含むデータ構造には赤、緑、青のチャネル（フィールド）がありますが、画像は通常、Sobel エッジ検出の前にグレースケールに変換されます。グレースケール画像ではチャネルが等価であるため、計算量が減ります。そのため、Sobel 実装の各ステップでは 1 つのチャネルのみ操作しています。計算量が減るだけではありません。次に示すように、ホストとデバイスのメモリー間で転送する必要があるデータの量も減ります。

```
gh     = reshape([-1,  0,  1, -2,  0,  2, -1,  0,  1], [3, 3])
gv     = reshape([-1, -2, -1,  0,  0,  0,  1,  2,  1], [3, 3])
smooth = reshape([ 1,  2,  1,  2,  4,  2,  1,  2,  1], [3, 3])

! 画像をスムージングしてノイズを削減
do concurrent (c = 2:img_width - 1, r = 2:img_height - 1)
    image_soa%red(r, c) = sum(image_soa%blue(r-1:r+1, c-1:c+1) * smooth) / 16
enddo

! Sobel エッジ検出を実行
do concurrent (c = 2:img_width - 1, r = 2:img_height - 1) reduce(max: max_gradient)
    image_soa%green(r, c) = abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gh)) + &
                            abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gv))
    max_gradient = max(max_gradient, image_soa%green(r, c))
enddo

! 勾配しきい値に基づいてエッジをハイライト
do concurrent (c = 1:img_width, r = 1:img_height)
    if (image_soa%green(r, c) >= 0.5 * max_gradient) then
        image_soa%green(r, c) = 0
    else
        image_soa%green(r, c) = 255
    endif
    image_soa%red(r, c)  = image_soa%green(r, c)
    image_soa%blue(r, c) = image_soa%green(r, c)
enddo
```

図 2. Fortran `DO CONCURRENT` ループ（青でハイライト表示）を使用して実装した Sobel エッジ検出。オフロードカーネルは緑でハイライト表示しています。完全なコード（`sobel_do_concurrent.F90`）は GitHub*（英語）から入手できます。

インテル® Fortran コンパイラーは、OpenMP* バックエンドを使用して `DO CONCURRENT` ループをアクセラレーターにオフロードできます。次のコマンドを使用して、「pvc」デバイス（インテル® データセンター GPU マックス）向けの事前コンパイルと、OpenMP* バックエンドによるアクセラレーター・オフロードのサンプルプログラムをビルドします。

```
$ ifx ppm_image_io.F90 sobel_do_concurrent.F90 -o sobel_dc_gpu -qopenmp \
>     -fopenmp-targets=spir64_gen -fopenmp-target-do-concurrent \
>     -Xopenmp-target-backend “-device pvc”
```

最初のソースファイル（ppm_image_io.F90（英語））は、画像 I/O を処理するユーティリティー・モジュールです。2 つ目のソースファイル（sobel_do_concurrent.F90（英語））には**図 2** のコードが含まれています。PPM 形式の 8K（7,680 x 8,404）解像度の画像（64,542,720 ピクセル x 4 バイト / ピクセル = 258,170,880 バイト）で実行ファイルを実行します。

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./sobel_dc_gpu -i xeon_4gen_8k.ppm -o xeon_8k_edges.ppm >& edge_detect_do_conc.out
$ grep Moving edge_detect_do_conc.out
Libomptarget --> Moving 258170880 bytes (hst:0x000014c199af22c0) -> (tgt:0xff00000005c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c189af1300) -> (tgt:0xff00000015400000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c179af0340) -> (tgt:0xff00000024c00000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000024c00000) -> (hst:0x000014c179af0340)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000015400000) -> (hst:0x000014c189af1300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000005c00000) -> (hst:0x000014c199af22c0)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c199af22c0) -> (tgt:0xff00000005c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c189af1300) -> (tgt:0xff00000015400000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c179af0340) -> (tgt:0xff00000024c00000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000024c00000) -> (hst:0x000014c179af0340)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000015400000) -> (hst:0x000014c189af1300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000005c00000) -> (hst:0x000014c199af22c0)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c199af22c0) -> (tgt:0xff00000036400000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c189af1300) -> (tgt:0xff00000045c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x000014c179af0340) -> (tgt:0xff00000055400000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000055400000) -> (hst:0x000014c179af0340)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000045c00000) -> (hst:0x000014c189af1300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000036400000) -> (hst:0x000014c199af22c0)
```

ホスト（`hst`）とターゲット（`tgt`）デバイスのメモリー間で転送される赤、緑、青の画像チャネルを色分けしました。**図 2** では、3 つのセクションで 3 つの `DO CONCURRENT` ループが示されています（ターゲットデバイスにマップされる配列の Fortran 配列記述子、またはドープベクトルは示されていません。ドープベクトルは小さいため、このデータ移動は無視できます。同様に、3 x 3 フィルター行列も示されていません）。ホストとデバイス間のデータ転送は暗黙的に処理されます。プログラマーにとっては便利ですが、必ずしも効率が良いとは限りません。例えば、**図 2** の最初の `DO CONCURRENT` ループは、青のチャネルのみ読み取り、赤のチャネルのみ書き込みます。そのため、最初の `DO CONCURRENT` ループで必要なのは、1 つの `hst` → `tgt` 転送と 1 つの `tgt` → `hst` 転送のみです。しかし、実際には、3 つのチャネルがすべてデバイスに転送され、ホストに戻されています。

その結果、大量の不要なデータ移動が行われています。不連続メモリー間のデータ移動には時間と電力がかかるため、ホストとデバイス間のデータ転送を最小限に抑えることは、ヘテロジニアス並列処理のパフォーマンスにとって重要です。残念なことに、ISO Fortran 2018 および 2023 標準では、データ移動を制御したり、データが読み取り専用か書き込み専用かをランタイムに伝える言語構造が提供されていません。

幸いなことに、OpenMP* target オフロード API（英語）では、Sobel 実装の 3 つのステップを 1 つのターゲット・データ・マップ領域に集約する方法が提供されていて、必要な場合にのみデータが転送されます（**図 3**）。

```
!$omp target data map(tofrom: image_soa%blue(1:img_height, 1:img_width),  &
!$omp                         image_soa%green(1:img_height, 1:img_width), &
!$omp                         image_soa%red(1:img_height, 1:img_width))   &
!$omp             map(to: gh(1:3, 1:3), gv(1:3, 1:3), smooth(1:3, 1:3))

!$omp target teams distribute parallel do collapse(2)
do c = 2, img_width - 1
    do r = 2, img_height - 1
        ! 画像をスムージングしてノイズを削減
        image_soa%red(r, c) = sum(image_soa%blue(r-1:r+1, c-1:c+1) * smooth) / 16
    enddo
enddo
!$omp end target teams distribute parallel do

!$omp target teams distribute parallel do reduction(max: max_gradient) collapse(2)
do c = 2, img_width - 1
    do r = 2, img_height - 1
        ! Sobel エッジ検出を実行
        image_soa%green(r, c) = abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gh)) + &
                                abs(sum(image_soa%red(r-1:r+1, c-1:c+1) * gv))
        max_gradient = max(max_gradient, image_soa%green(r, c))
    enddo
enddo
!$omp end target teams distribute parallel do

!$omp target update from(max_gradient)

!$omp target teams distribute parallel do collapse(2)
do c = 1, img_width
    do r = 1, img_height
        ! 勾配しきい値に基づいてエッジをハイライト
        if (image_soa%green(r, c) >= 0.5 * max_gradient) then
            image_soa%green(r, c) = 0
        else
            image_soa%green(r, c) = 255
        endif
        image_soa%red(r, c)  = image_soa%green(r, c)
        image_soa%blue(r, c) = image_soa%green(r, c)
    enddo
enddo
!$omp end target teams distribute parallel do

!$omp end target data
```

図 3. OpenMP* `target` オフロード・ディレクティブ（青でハイライト表示）を使用して実装した Sobel エッジ検出。完全なコード（`sobel_omp_target.F90`）は GitHub*（英語）から入手できます。

図 3 の OpenMP* 実装を次のようにコンパイルして実行しました。

```
$ ifx ppm_image_io.F90 sobel_omp_target.F90 -o sobel_omp_gpu -qopenmp \
>     -fopenmp-targets=spir64_gen -Xopenmp-target-backend “-device pvc”
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./sobel_omp_gpu -i xeon_4gen_8k.ppm -o xeon_8k_edges.ppm >& edge_detect_openmp.out
$ grep Moving edge_detect_openmp.out
Libomptarget --> Moving 258170880 bytes (hst:0x0000150220dfc340) -> (tgt:0xff00000005c00000)
Libomptarget --> Moving 258170880 bytes (hst:0x0000150230dfd300) -> (tgt:0xff00000015400000)
Libomptarget --> Moving 258170880 bytes (hst:0x0000150240dfe2c0) -> (tgt:0xff00000024c00000)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000024c00000) -> (hst:0x0000150240dfe2c0)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000015400000) -> (hst:0x0000150230dfd300)
Libomptarget --> Moving 258170880 bytes (tgt:0xff00000005c00000) -> (hst:0x0000150220dfc340)
```

小さなドープベクトルとフィルター行列を削除して、ホスト（`hst`）とターゲット（`tgt`）デバイスのメモリー間で転送される赤、緑、青の画像チャネルを色分けしました。各画像チャネルは各方向に 1 回のみコピーされるようになり、`DO CONCURRENT` 実装よりもデータ転送が大幅に少なくなりました。ごのように、OpenMP* を使用すると、現在の ISO Fortran よりもデータ移動を細かく制御できます。

OpenMP* `target` オフロードは、大きな画像で Sobel エッジ検出を計算する場合にも、`DO CONCURRENT` よりも優れたパフォーマンスを提供します（**表 1**）。`DO CONCURRENT`（**図 2**）と OpenMP* `target`（**図 3**）の例は、ホスト CPU では同等のパフォーマンスが得られていますが（0.1 秒）、OpenMP* `target` で GPU にオフロードすると最高のパフォーマンスが得られています（0.05 秒）。一方、`DO CONCURRENT` の GPU でのパフォーマンスは、不要なデータ転送のため逆に低下しています（0.18 秒）。

	OpenMP	DO CONCURRENT
シーケンシャル	0.37	0.37
並列(CPU)	0.1	0.1
並列(GPU)	0.05	0.18

表 1. 8K（7,680 x 8,404）解像度の画像の OpenMP* `target` と Fortran `DO CONCURRENT` のパフォーマンスの比較（時間はすべて秒）。シーケンシャル・ベースラインは、OpenMP* を有効にしないでコンパイルされた OpenMP*（sobel_omp_target.F90）および `DO CONCURRENT`（sobel_do_concurrent.F90）の例です。CPU と GPU はそれぞれ、インテル® Xeon® Platinum 8480+ とインテル® データセンター GPU マックス 1100。

`DO CONCURRENT` は暗黙的な並列ループを表現するための便利な構文を提供しますが、ISO Fortran で不連続メモリーの概念、およびホストとデバイス間のデータ転送を制御する構文が利用できるようになるまでは、ヘテロジニアス並列処理には Fortran と OpenMP* を組み合わせることが最善と言えるでしょう。

ソースコードとテストイメージは、GitHub*（英語）から入手できます。最新のインテルのハードウェアとソフトウェアを利用可能な無料のインテル® デベロッパー・クラウド（英語）で、Fortran `DO CONCURRENT` と OpenMP* アクセラレーター・オフロードの実験を行うことができます。