Fortran の DO CONCURRENT を使用したアクセラレーター・オフロード

ヘテロジニアス・コンピューティングに対する標準 Fortran の制限

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

数式（FORmula）をコードに変換（TRANslate）する必要がある場合、FORTRAN は最適なオプションであり、これまで 66 年間にわたり使用されてきました。Fortran こそが数学演算向けのオリジナルのドメイン固有言語であると主張することもできますが、その議論は別の機会に取っておきます。この記事の目的は、Fortran の長所を称賛することではなく、ヘテロジニアス並列処理における Fortran の長所と短所を評価することです。

標準ベースのプログラミング言語は、アルゴリズムを表現する共通の方言を提供しています。「SYCL* の事例 : ISO C++ 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」で説明したように、特殊なハードウェアのサポートには時間がかかる傾向があります。ISO Fortran がヘテロジニアス・コンピューティングをどの程度サポートしているか見てみましょう。おそらく、ロスアラモス国立研究所の最近のレポート「今後 15 年間にミッション・クリティカルなコードを Fortran に依存することのリスクの評価」（英語）によって引き起こされた議論に何かを追加できるでしょう。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
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。

DO CONCURRENT 構文は ISO Fortran 2008 で追加され、最近の ISO 標準で強化されました。この構文は、DO CONCURRENT ループの反復が独立していて、並列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。インテル® Fortran コンパイラーは DO CONCURRENT をサポートしています。DO CONCURRENT ループはシーケンシャルかつ並列に実行でき、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。

単純なイメージ・セグメンテーション・アルゴリズムを使用して、この機能を実証します。このアルゴリズムは、イメージ内のオブジェクトのエッジを検出します (**図 1**)。エッジにはイメージ内のほとんどの情報が含まれるため、このハイパスフィルターは多くのコンピューター・ビジョン・プロセスの最初のステップとなります。**図 1** は、「1」のグループで表される 3 つのオブジェクトを含むバイナリーイメージを示しています。エッジマスクはブール行列で、true(T) は対応する「ピクセル」がオブジェクトのエッジにあることを意味します。

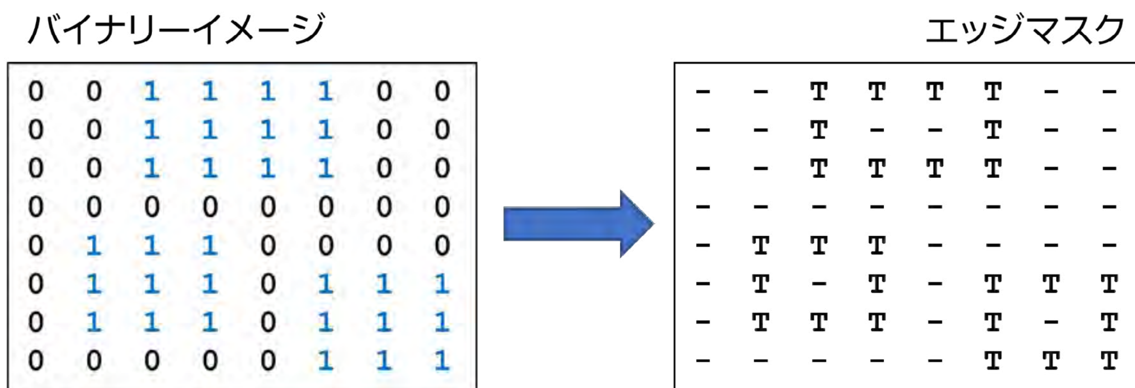

図 1. 単純なバイナリーイメージのエッジ検出

Fortran は、エッジ検出を簡単にコーディングできる、便利な配列表記法と組込みプロシージャーを提供しています (**図 2**)。このアルゴリズムは、9 ポイントのバイナリーフィルターを各ピクセルに適用することで実装できます。DO CONCURRENT ループのプレディケートにより、フィルターはオブジェクトの一部であるピクセルにのみ適用されます。各ピクセルの演算は独立しているため、アルゴリズムはデータ並列性が高く、Fortran DO CONCURRENT ループと数行のコードで簡単に実装できます。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
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。

```
INTEGER, ALLOCATABLE :: image(:,:)
LOGICAL, ALLOCATABLE :: edge_mask(:,:)

! イメージとエッジマスクを割り当て
ALLOCATE (image(n, n), source = 0, stat = allocstat, errmsg = allocmsg)
ALLOCATE (edge_mask(n, n), source = .FALSE., stat = allocstat, errmsg = allocmsg)

! イメージを初期化

! バイナリーイメージのオブジェクトのアウトライン
DO CONCURRENT (j = 1:n, i = 1:n, image(i, j) /= 0)
      IF (i == 1 .OR. i == n .OR. &
          j == 1 .OR. j == n) THEN
         edge_mask(i, j) = .TRUE.
      ELSE
         IF (ANY(image(i-1:i+1, j-1:j+1) == 0)) edge_mask(i, j) = .TRUE.
      ENDIF
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（`img_seg_do_concurrent.F90`）は GitHub*（英語）から入手できます。

`DO CONCURRENT` 構文は、DO 構文の別の形式にすぎません。`DO CONCURRENT` を初めて見た場合でも、ほとんどの Fortran プログラマーには、この例が二重にネストされた `DO` ループのように `i` インデックスと `j` インデックスをループしていることは明らかでしょう。`DO CONCURRENT` 構文の新しい点は、オプションのプレディケートです。これは、`LOGICAL` 型のスカラーマスク式です。プレディケートがある場合、マスク式が `TRUE` の反復のみ実行されます。上記の `DO CONCURRENT` コードは、次の `DO` および `IF` 実装と機能的には同等です。

```
DO j = 1, n
   DO i = 1, n
      IF (image(i, j) /= 0) THEN

         ! 同じループコンテンツ

      ENDIF
   ENDDO
ENDDO
```

主な違いは、`DO CONCURRENT` は依存関係がないことをコンパイラーにアサートするため、反復を任意の順序で実行できることです。

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` ループ内のステートメントを並列化またはオフロードできます。これは、サンプルコードをコンパイルするコマンドを見れば明らかです。

```
$ ifx img_seg_do_concurrent.F90 -o img_seg_do_conc_cpu -qopenmp

$ ifx img_seg_do_concurrent.F90 -o img_seg_do_conc_gpu -qopenmp -fopenmp-targets=spir64
```

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
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。

1 つ目の実行ファイル（img_seg_do_conc_cpu）は、利用可能なすべてのホスト・プロセッサーで DO CONCURRENT ループを並列に実行します。2 つ目の実行ファイル（img_seg_do_conc_gpu）は、計算をアクセラレーター・デバイスにオフロードします。ホストとデバイス間のデータ転送は、OpenMP* ランタイムにより暗黙的に処理されます。ISO C++ と同様に、ISO Fortran 2018 には不連続メモリーの概念がないため、データ転送を制御する言語構造はありません。コーディングが簡素化されるため、プログラマーにとっては朗報です。ランタイムは必要なデータをデバイスにコピーし、DO CONCURRENT ループの実行が終了すると、ホストにコピーバックします。10 個のオブジェクトがランダムに位置する単一の 1,000 x 1,000 のイメージに対してサンプルプログラムを実行した場合の OpenMP* ランタイムからのデバッグ出力を次に示します。

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./img_seg_do_conc_gpu -n 1000 -i 1 -o 10 >& edge_detect_do_conc.out

$ grep Moving edge_detect_do_conc.out

Libomptarget --> Moving 4000000 bytes (hst:0x00007f22a4087200) -> (tgt:0x0000000002e17000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffc6259a60) -> (tgt:0x00000000028d0008)
Libomptarget --> Moving 4000000 bytes (hst:0x00007f228529b240) -> (tgt:0x0000000003217000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffc6259a00) -> (tgt:0x00000000028d0088)
Libomptarget --> Moving 4000000 bytes (tgt:0x0000000003217000) -> (hst:0x00007f228529b240)
Libomptarget --> Moving 4000000 bytes (tgt:0x0000000002e17000) -> (hst:0x00007f22a4087200)
```

イメージとエッジマスクの配列をハイライト表示します。各配列は 1,000 x 1,000 x 4 バイト = 4,000,000 バイトで、hst → tgt と tgt → hst の両方で転送されることが分かります。つまり、合計 16,000,000 バイトがホスト（hst）とターゲット（tgt）デバイス間で転送されます（ハイライトされていない 88 バイトのデータ移動は、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](https://)（英語）です。配列記述子は通常は小さいため、このデータ移動は無視できます）。

暗黙的なホストとデバイス間のデータ転送は便利ですが、常に効率が良いとは限りません。image 変数は DO CONCURRENT ループの本体では変更されていないことに注意してください（**図 2**）。データはデバイスで読み取られるだけであり、ホストへ再度転送する必要はありません。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コンピューティングにおける最優先事項です。残念ながら、ISO Fortran 2018 および 2023 では、データ移動を制御する言語構造が提供されていません。

OpenMP* ターゲットオフロード API（英語）は、ホストとデバイス間のデータ転送を明示的に制御する構文を提供します（**図 3**）。エッジ検出アルゴリズムの OpenMP* 実装は、イメージをデバイスに転送し [MAP(TO:image)]、エッジマスクをホストに転送するだけです [MAP(FROM:edge_mask)]。

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./img_seg_omp_gpu -n 1000 -i 1 -o 10 >& edge_detect_omp.out

$ grep Moving edge_detect_omp.out

Libomptarget --> Moving 88 bytes (hst:0x00007ffd0b0f4638) -> (tgt:0x0000000003d91008)
Libomptarget --> Moving 4000000 bytes (hst:0x00007f4f1a69c200) -> (tgt:0x00000000042cd000)
Libomptarget --> Moving 88 bytes (hst:0x00007ffd0b0f4698) -> (tgt:0x0000000003d91088)
Libomptarget --> Moving 4000000 bytes (tgt:0x0000000003ecd000) -> (hst:0x00007f4f1a2c9240)
```

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
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。

再度、イメージとエッジマスクの配列をハイライト表示します。イメージ（4,000,000 バイト）は hst → tgt で転送され、エッジマスク（4,000,000 バイト）は tgt → hst で転送されるため、合計 8,000,000 バイトのみ転送されることが分かります。DO CONCURRENT コード（**図 2**）は、OpenMP* ターゲット･オフロード･コード（**図 3**）の 2 倍のデータ移動を行います。

```
! バイナリーイメージのオブジェクトのアウトライン
!$OMP TARGET DATA MAP(TO:image) MAP(FROM:edge_mask)
!$OMP TARGET
!$OMP PARALLEL DO
DO j = 1, n
   DO i = 1, n
      edge_mask(i, j) = .FALSE.
      IF (image(i, j) /= 0) THEN
         IF (i == 1 .OR. i == n .OR.&
             j == 1 .OR. j == n) THEN
               edge_mask(i, j) = .TRUE.
         ELSE
            IF (ANY(image(i-1:i+1, j-1:j+1) == 0)) edge_mask(i, j) = .TRUE.
         ENDIF
      ENDIF
   ENDDO
ENDDO
!$OMP END TARGET
!$OMP END TARGET DATA
```

図 3. OpenMP* ターゲットオフロード（青でハイライト表示）を使用して実装したエッジ検出。完全なコード（img_seg_omp_target.F90）は GitHub*（英語）から入手できます。

この例の単純なエッジ検出器は、アクセラレーターへのオフロードの恩恵を受けるほど十分な作業を行いません。バイナリーイメージに単純な 3x3 ポイントフィルターを使用してイメージ・セグメンテーションを実装しました。より複雑なフィルター、実際のイメージ、またはボリュームのあるイメージは、計算やデータの負荷が高く、パフォーマンスとオフロード特性に影響を与えます。次回の記事では、より現実的なエッジ検出器（Sobel フィルターなど）と実際のイメージのベンチマークを行います。ただし、ホストとデバイス間の不要なデータ転送がパフォーマンスを制限することは経験から判明しています。一例は、前号の記事「oneMKL と OpenMP* ターゲットオフロードで線形システムを解く」で示しています。

良いニュースと悪いニュースがあります。良いニュースは、ISO Fortran コード（**図 2**）をアクセラレーターで実行できることです。ランタイムにホストとデバイス間のデータ転送を暗黙的に処理させることは、多くのアルゴリズムでは問題ありません。悪いニュースは、読み取り専用イメージを扱うエッジ検出はその例外にあたることです。データ転送を明示的に制御する方法がないために不要なデータ転送が避けられず､ヘテロジニアス並列処理のパフォーマンスが制限される可能性があります。幸いなことに、OpenMP* ターゲットオフロード API は、必要に応じて明示的な制御を提供します。

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。

© Intel Corporation. Intel、インテル、Intel ロゴ、その他のインテルの名称やロゴは、Intel Corporation またはその子会社の商標です。
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。