Fortran、oneMKL、OpenMP* を使用して LU 因数分解を高速化

一般的な数学演算を高速化する標準ベースのオープンなアプローチ

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

BASIC、Pascal、C は私が最初に学んだプログラミング言語ですが、初めて本格的な（教室外での）コーディングをしたのは Fortran でした。インテルに入社して C と C++ に乗り換えるまでの 12 年間は、ほとんど Fortran しか使いませんでした。この 20 年間、Fortran を 1 行も書いていないことを、私はいくつかの著作で嘆いています。旧友の Jim Cownie が「Fortran は過去の言語なのか？」（英語）において、そうではないという説得力のある答えを出すまで、Fortran がまだ使われているかどうか疑問に思っていたくらいです。そのため、同僚から最新の Fortran コンパイラーの OpenMP* アクセラレーター・オフロード・サポートを試してみないかと誘われたとき、私はそのチャンスに飛びつきました。

私は、Fortran と同じくらい OpenMP* が好きです。OpenMP* は常にコードを並列化する簡単な方法であり、1997 年に並列コンピューティング分野に登場して爆発的に広がって以来、OpenMP* 仕様は長い道のりを歩んできました。現在では、アクセラレーターへのオフロードもサポートしています。そこで、いつも使っている SYCL* ではなく、並列プログラミングの原点に戻って、Fortran、OpenMP*、インテル® oneAPI マス・カーネル・ライブラリー（インテル® oneMKL）を使ってアクセラレーターに計算をオフロードする方法を説明したいと思います。

© 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。

以前の記事（記事（英語）、ウェビナー（英語）、サンプルコード（英語））を読んでいただいた方は、私がフーリエ相関アルゴリズムを気に入っていることをご存じだと思います。私はこれを生体分子のドッキングの研究に使い、最近では 2D 画像と 3D 画像の調整に使用しています。これまで、このアルゴリズムにはうんざりするくらい取り組んできたので、ここでは線形代数計算（特に LU 分解）をアクセラレーターにオフロードすることがいかに簡単か見てみましょう。LU 分解について詳しく説明することがこの記事の目的ではないので、線形方程式を解くために使われるものだと理解いただければ十分です。詳しく知りたい場合は、インテル® MKL クックブック（英語）の「LU 因数分解されたブロック三重対角係数行列を含む連立線形方程式を解く」（英語）と「一般的なブロック三重対角行列の因数分解」（英語）を参照してください。ここでは、Fortran、インテル® oneMKL、および OpenMP* の実装を通して、オフロード構文を説明し、ヘテロジニアス並列処理におけるホストとデバイス間のデータ転送に注目します。まず、インテル® oneMKL に含まれている `dgetri_oop_batch_strided.f90` の例から開始し（「サンプルコードの使用」（英語））、効率を改善するためこれを変更します。

このプログラムは、まず OpenMP* オフロードをサポートする Fortran インターフェイスをインクルードし、次に 32 ビット整数型と 64 ビット整数型のどちらを使用するかを決定します（詳細は、「ILP64 インターフェイスと LP64 インターフェイスの使用」（英語）を参照）（**図 1**）。この判断は、この記事の最後にあるコンパイラー・コマンドで示すように、コンパイル時に行います。セットアップを完了するため、因数分解する行列の数とそのサイズを設定し、インテル® oneMKL サブルーチンが必要とする次元とストライドのパラメーターを設定し、ワーク配列を割り当てます。これで計算を開始する準備が整いました。

```
include "mkl_omp_offload.f90"

program sgetri_oop_batch_strided_example

#if defined(MKL_ILP64)
   use onemkl_lapack_omp_offload_ilp64   ! 64-bit integer type
#else
   use onemkl_lapack_omp_offload_lp64    ! 32-bit integer type
#endif

   integer,   parameter :: n = 10, batch_size = 4
   integer              :: lda, ldainv, stride_a, stride_ainv, stride_ipiv
   real,    allocatable :: a(:,:), ainv(:,:)
   integer, allocatable :: ipiv(:,:), info(:)

   lda         = n
   ldainv      = n
   stride_a    = lda    * n
   stride_ainv = ldainv * n
   stride_ipiv = n

   allocate(a(stride_a, batch_size), ainv(stride_ainv, batch_size), &
            ipiv(stride_ipiv, batch_size), info(batch_size))
```

図 1. サンプルプログラムのセットアップ。OpenMP* ターゲットオフロードをサポートするインテル® oneMKL ヘッダーとモジュールは、青色で表示しています。ハードコードされた行列の次元とバッチサイズのパラメーターは、テストには適していますが、高速化には小さすぎるということに注意してください。

© 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。

手元にある GPU は単精度の計算しかサポートしていないので、Fortran REAL 型と適切な LAPACK（英語）サブルーチンを使用して、LU 因子分解 (sgetrf_batch_strided（英語）と LU 因子分解した行列のアウトオブプレース反転（sgetri_oop_batch_strided（英語））を計算します（倍精度に変換するには、Fortran DOUBLE PRECISION 型に切り替え、LAPACK の dgetrf_batch_strided サブルーチンと dgetri_oop_batch_strided サブルーチンを使用するだけです）。オリジナルのコードでは、2 つの OpenMP* target data 領域を使用しています（**図 2**）。ターゲット構造は、デバイスに制御を転送します。最初の領域は、入力行列をデバイスメモリーに転送し [map(tofrom:a)]、インプレース LU 分解（sgetrf_batch_strided）をデバイスにディスパッチし、LU 分解した行列（a に格納）、ピボット・インデックス [map(from:ipiv)]、ステータス [map(from:info)] をデバイスメモリーから取得します。因数分解が正常に実行されると、プログラムは次の OpenMP* 領域に進みます。LU 因数分解された行列とピボット･インデックスはデバイスメモリーに戻され [map(to:a) と map(to:ipiv)]、アウトオブプレース反転はデバイス上で計算され（sgetri_oop_batch_strided）、逆行列 [map(from:ainv)] とステータス [map(from:info)] はデバイスメモリーから取得されます。

```
! OpenMP* オフロードで LU 因数分解を計算
! 開始時に A は入力行列を格納
! 終了時に A は LU 因数分解された行列を格納
!$omp target data map(tofrom:a) map(from:ipiv) map(from:info)

    !$omp dispatch
    call sgetrf_batch_strided(n, n, a, lda, stride_a, ipiv, stride_ipiv, &
                              batch_size, info)
!$omp end target data

if (any(info .ne. 0)) then
    print *, 'Error: sgetrf_batch_strided returned with errors.'
else
    ! OpenMP* オフロードで行列の反転を計算し、終了時に逆行列は Ainv に格納
    !$omp target data map(to:a) map(to:ipiv) map(from:ainv) map(from:info)

        !$omp dispatch
        call sgetri_oop_batch_strided(n, a, lda, stride_a, ipiv, stride_ipiv, &
                                      ainv, ldainv, stride_ainv, batch_size, info)
    !$omp end target data
endif

if (any(info .ne. 0)) then
    print *, 'Error: sgetri_oop_batch_strided returned with errors.'
else
    print '("The calculations executed successfully.")'
endif
```

図 2. 2 つの LAPACK サブルーチンそれぞれに OpenMP* ターゲット領域を使用して、LAPACK 計算をアクセラレーターにオフロード。OpenMP* ディレクティブは青で表示しています。LAPACK の呼び出しは緑色で表示しています。

このサンプルプログラムの作者は、反転に進む前に因数分解の結果を確認するため、2 つの OpenMP* ターゲット領域を使用しているのでしょう。このプログラマーの注意深さは賞賛に値しますが、追加のホストとデバイス間のデータ転送が発生するため、ヘテロジニアス並列プログラミングでは避けるべきです。**図 3** の矢印はすべて、不連続メモリー間のデータ移動を表しています。例えば、入力行列はデバイスに転送され、因数は

© 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。

ホストに戻されます [map(tofrom:a)]。そして、同じ因数が次の OpenMP* target data 領域でデバイス [map(to:a)] に再度転送されます。同様に、ピボット･インデックスは、sgetrf_batch_strided がリターンした後、デバイス [map(from:ipiv)] から取得され、sgetri_oop_batch_strided への入力としてデバイス [map(to:ipiv)] に戻されます。ヘテロジニアス･コンピューティングを行うということは、これらの行列はおそらく大きいため、ホストからデバイスへ、またはその逆に転送すると、計算と電力のオーバーヘッドが大きくなる可能性があります。1 次元のステータス配列はそれほど大きくないので、map(from:info) の転送にそれほど悩まされることはないです。

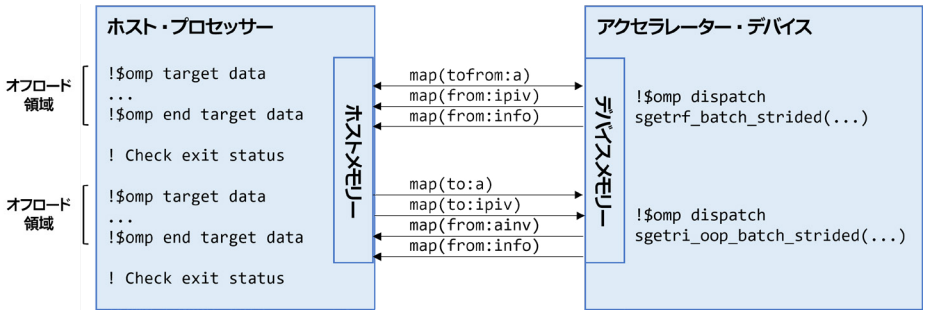

図 3. 図 2 に示した 2 つの OpenMP* ターゲットオフロード領域のホストとデバイス間のデータ転送。各矢印は、ホストとデバイスのメモリー間のデータ転送を示します。

理想的には、OpenMP* ランタイムが、必要なデータがすでにデバイスメモリーにあるかどうかをチェックすることです。しかし、OpenMP* のターゲット領域間でそれらのデータがフラッシュされず、上書きもされないという保証はありません。また、2 つの OpenMP* オフロード領域を作成することによるオーバーヘッドを考慮する必要があります。このため、1 つのオフロード領域ですべての計算を行うほうがよいでしょう (**図 4**)。

```
!$omp target data map(to:a) map(from:ainv) map(from:info_rf) map(from:info_ri)

    !$omp dispatch
    call sgetrf_batch_strided(n, n, a, lda, stride_a, ipiv, stride_ipiv, &
                              batch_size, info_rf)

    !$omp dispatch
    call sgetri_oop_batch_strided(n, a, lda, stride_a, ipiv, stride_ipiv, &
                                  ainv, ldainv, stride_ainv, batch_size, info_ri)
!$omp end target data

if (any(info_rf .ne. 0) then
    print *, 'Error: getrf_batch_strided returned with errors.'
elseif (any(info_ri .ne. 0) then
    print *, 'Error: getri_oop_batch_strided returned with errors.'
else
    print '("Computation executed successfully")'
endif
```

図 4. 1 つの OpenMP* ターゲット領域を使用して、LAPACK 計算をアクセラレーターにオフロード。OpenMP* ディレクティブは青で表示しています。LAPACK の呼び出しは緑色で表示しています。

© 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。

2 つの OpenMP* ターゲット領域を 1 つに融合すると、ホストとデバイス間のデータ転送が少なくなることが分かります（**図 5**）。入力行列（a）はデバイスに転送され、インプレースで因数分解されます。LU 因子とピボット（a、ipiv）はデバイス上で sgetrf_batch_strided によって計算されるので、sgetri_oop_batch_strided が呼び出されたときにはすでにデバイスメモリーにあります。後で連立 1 次方程式を解くために使用する予定がない限り、それらをホストに転送する必要はありません。ただし、その場合は、必要な入力データがデバイスメモリーにある間に、同じ OpenMP* target data 領域で sgetrs_batch_strided（英語）を呼び出すことができます。これは読者の演習として残しておきます。

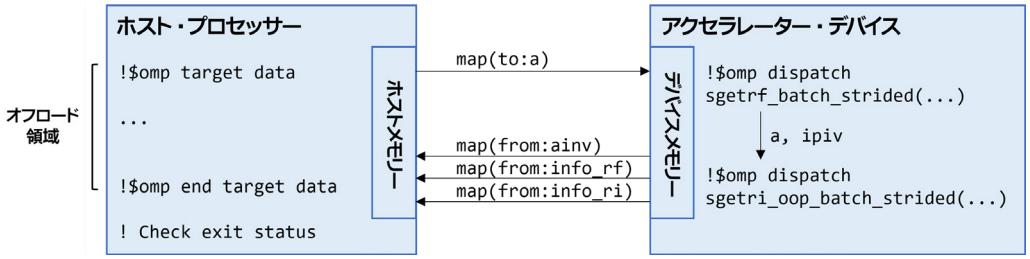

図 5. 図 4 の OpenMP* ターゲットオフロード領域におけるホストとデバイス間のデータ転送。各矢印（アクセラレーター・デバイス内を除く）は、ホストとデバイスのメモリー間のデータ転送を示します。

この記事で説明するサンプルプログラムは、このリポジトリー（英語）から入手できます。サンプルプログラムには、入力行列を初期化し、計算の各ステップの出力を表示するコードが含まれていますが、この記事には含めませんでした。インテル® Fortran コンパイラーとインテル® oneMKL を使用してサンプルプログラムをビルドするコマンドは、OpenMP* を用いる場合とそうでない場合のそれぞれで、次のとおりです。

```
$ ifx -i8 -fpp -DMKL_ILP64 -qopenmp -fopenmp-targets=spir64 -fsycl \
  lu_fact_omp_offload_example.f90 -o lu_fact_omp_offload_example.exe \
  -L${MKLROOT}/lib/intel64 -lmkl_sycl -lmkl_intel_ilp64 -lmkl_intel_thread \
  -lmkl_core -liomp5 -lpthread -ldl

$ ifx -i8 -fpp -DMKL_ILP64 \
  lu_fact_omp_offload_example.f90 -o lu_fact_example.exe \
  -L${MKLROOT}/lib/intel64 -lmkl_intel_ilp64 -lmkl_intel_thread \
  -lmkl_core -liomp5 -lpthread -ldl
```

コードはどちらの場合も同じであり、これは OpenMP* の主な利点の 1 つです。コンパイラーが OpenMP* ディレクティブを無視しても、シリアルコードはそのまま残ります。

インテル® oneMKL 関数は SYCL* メソッドを使用してアクセラレーター・オフロードを実行するため、インテル® Fortran コンパイラーおよびインテル® oneMKL とともに、デバイスドライバーとデータ並列 C++（英語）コンパイラーをインストールして設定する必要があります。最も簡単な方法は、インテル® oneAPI ベース・ツールキットとインテル® oneAPI HPC ツールキットの両方をインストールすることです。インテル® oneAPI を初めて使用する場合、またはアップデートを探している場合は、インテル® oneAPI ツールキットを参照してください。

© 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。

図 1 に示す配列の次元とバッチサイズは、テストには適していますが、アクセラレーションの恩恵を受けるには小さすぎることに注意してください。計算量は、制御フローをターゲットデバイスに転送し、不連続メモリー間でデータを転送する際のオーバーヘッドを相殺できる程度に大きくする必要があります。このサンプルコードのパフォーマンス特性については、今後の記事で取り上げますが、配列の次元やバッチサイズを自由に試すことで、計算をアクセラレーターにオフロードせずにホスト・プロセッサーに残したほうが良い場合について、考察できます。

ここでは、OpenMP* のアクセラレーター・オフロード機能のほんの一部を紹介しました。OpenMP* 仕様（英語）には、ほかにも多くのオフロード構造が記載されています。「OpenMP* API を使用した GPU プログラミングの概要」（英語）で、初心者向けのチュートリアルを提供しています。最新のインテルのハードウェアとソフトウェアがインストールされている無料のインテル® デベロッパー・クラウド（英語）では、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。

ポッドキャスト
災害管理の使命と科学的発見
聴く（英語）

アニメーター、デジタルコンテンツ制作者、建築エンジニア、熟練ゲーマーの皆さん、インテル® oneAPI ベース & レンダリング・ツールキットでビジュアライゼーションの限界に挑戦してみてください。詳細 >
intel. 1 oneAPI RENDERING TOOLKIT

© 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。