oneMKL と OpenMP* ターゲットオフロードで線形システムを解く

標準ベースのオープンなアプローチで線形代数計算を高速化

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
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

前号の記事、「Fortran、oneMKL、OpenMP* を使用して LU 因数分解を高速化」では、LU 因数分解と後続の反転をアクセラレーターにオフロードする方法を紹介しました。OpenMP* オフロード領域を統合して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でオーバーヘッドを削減するヒントをいくつか示しましたが、連立線型方程式を解くには至りませんでした。これは読者の演習として残しておいたものですが、『oneAPI GPU 最適化ガイド』の「データ転送とメモリー割り当てを最小化する」のいくつかのテクニックを説明するためにここで実行します。

OpenMP* を使用してアクセラレーターにオフロードされた oneMKL の LU 因数分解とソルバー関数を使用してバッチ処理された線形システムのグループを解く方法を示します。逆行列の因数は実際にはほとんど使用しないため、このデモではそのステップを省略し、因数分解から解法に進みます。前のサンプルコードは Fortran で記述されていました。ここでも引き続き Fortran を使用します。

© 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。

まず、いくつかの小さな線形システムを読み込んで正しく解いていることを確認します。古い線形代数の教科書に適切な例がいくつか見つかりました（**図 1**）。行列は正方で、LU 因数分解を使用して解くことができます。

$$\begin{bmatrix} 2 & 4 & -4 \\ 1 & -4 & 3 \\ -6 & -9 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ -21 \\ -24 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 3 \\ 2 & 4 & -1 \\ 6 & 5 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 \\ -10 \\ -7 \end{bmatrix}$$

$$(x_1, x_2, x_3) = (-4, 2, -3) \qquad (x_1, x_2, x_3) = (-2, -1, 2)$$

図 1. oneMKL と OpenMP* ターゲットオフロードを使用して、これらの 2 つの線形システムをバッチで解きます。これらの小さな問題はテストには適していますが、高速化の恩恵を受けるには小さすぎます。

このプログラムは、前号の記事のプログラムと似ています。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（**リスト 1**）（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）。この判断はコンパイル時に行われます（次のコンパイラー・コマンドを参照）。**図 1** のテスト問題は、それぞれ 1 つの右辺（RHS）がある 2 つの 3x3 線形システムで構成されています。`batch_size`、`n`、`nrhs`、および `stride` 変数がそれぞれ設定され、`a`、`b`、および `ipiv` 配列が割り当てられて、バッチ処理されたシステムが保持されます。最後の 4 つの文は、2 つの行列とその右辺をそれぞれ `a` と `b` にロードします。

インテル® DPC++ 互換性ツール
CUDA* アプリケーションを標準ベースの SYCL* コードに移行
詳細（英語）

© 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。

```
INCLUDE "mkl_omp_offload.f90"

PROGRAM solve_batched_linear_systems

! 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決定
#IF DEFINED(MKL_ILP64)
    USE onemkl_lapack_omp_offload_ilp64   ! 64 ビット
#ELSE
    USE onemkl_lapack_omp_offload_lp64    ! 32 ビット
#ENDIF

    INTEGER,          PARAMETER :: n = 3, batch_size = 2, nrhs = 1
    INTEGER                     :: lda, stride_a, stride_ipiv
    INTEGER                     :: ldb, stride_b
    REAL (KIND=8), ALLOCATABLE :: a(:,:), b(:,:)
    INTEGER,       ALLOCATABLE :: ipiv(:,:), info(:)

    INTEGER allocstat
    CHARACTER (LEN = 132) :: allocmsg

    lda         = n
    stride_a    = n * lda
    stride_ipiv = n
    ldb         = n
    stride_b    = n * nrhs

    ! 必要なメモリーを割り当て
    ALLOCATE (a(stride_a, batch_size), b(n, batch_size*nrhs), &
              ipiv(stride_ipiv, batch_size),                  &
              info(batch_size), STAT = allocstat, ERRMSG = allocmsg)
    if (allocstat > 0) STOP TRIM(allocmsg)

    ! 行列を初期化。Fortran は列優先言語であることに注意。
    a(:,1) = (/2.0, 1.0, -6.0, 4.0, -4.0, -9.0, -4.0,  3.0, 10.0/)
    a(:,2) = (/0.0, 2.0,  6.0, 0.0,  4.0,  5.0,  3.0, -1.0,  5.0/)
    b(:,1) = (/12.0, -21.0, -24.0/)         ! x = (-4,  2, -3)
    b(:,2) = (/ 6.0, -10.0,  -7.0/)         ! x = (-2, -1,  2)
```

リスト 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 oneMKL ヘッダーとモジュールは、青色で表示しています。

計算を開始する準備が整いました。2 つの OpenMP* ターゲット領域を使用して LU 因数分解と解を線形システムにディスパッチする基本的な実装から始めましょう (**リスト 2**)。サンプルを次のようにコンパイルして実行し、正しい結果が得られることを確認します。

© 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。

```
$ ifx -i8 -DMKL_ILP64 -qopenmp -fopenmp-targets=spir64 -fsycl -free \
> lu_solve_omp_offload_ex1_small.F90 -o lu_solve_ex1_small \
> -L${MKLROOT}/lib/intel64 -lmkl_sycl -lmkl_intel_ilp64 -lmkl_intel_thread \
> -lmkl_core -liomp5 -lpthread -ldl

$ ./lu_solve_ex1_small

=================================
 Solutions to the linear systems
=================================
   -4.0000    2.0000   -3.0000
   -2.0000   -1.0000    2.0000
```

```
    ! OpenMP* オフロードを使用して 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)

        !$OMP DISPATCH
        CALL dgetrf_batch_strided(n, n, a, lda, stride_a, &
                                  ipiv, stride_ipiv, batch_size, info)
    !$OMP END TARGET DATA

    IF (ANY(info .ne. 0)) THEN
        PRINT *, 'Error: getrf_batch_strided returned with errors'
    ELSE
        ! 線形システムを解く。終了時に解を b に格納。
        !$OMP TARGET DATA MAP(TO:a) MAP(TO:ipiv) MAP(TOFROM: b) MAP(FROM:info)

            !$OMP DISPATCH
            CALL dgetrs_batch_strided('N', n, nrhs, a, lda, stride_a, &
                                      ipiv, stride_ipiv, &
                                      b, ldb, stride_b, batch_size, info)
        !$OMP END TARGET DATA

        IF (ANY(info .ne. 0)) THEN
            PRINT *, 'Error: getrs_batch_strided returned with errors'
        ELSE
            PRINT *, 'Computation executed successfully'
        ENDIF
    ENDIF
```

リスト 2. oneMKL と OpenMP* ターゲットオフロードを使用してバッチ処理された線形システムの基本的なソリューション。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。

`TARGET` 構造は、アクセラレーター・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 因数分解をデバイスにディスパッチし (`dgetrf_batch_strided`)、LU 因数分解された行列 (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・インデックスをデバイスメモリーに戻し [`MAP(TO:a)` および `MAP(TO:ipiv)`]、線形システムをデバイスで解きます (`dgetrs_batch_strided`)。RHS は解ベクトルで上書きされ、計算 [`MAP(FROM:info)`] のステータスとともにデバイスメモリー [`MAP(TOFROM:b)`] から取得されます。ホストとデバイス間のデータ転送を図式化したのが**図 2** です。

© 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。

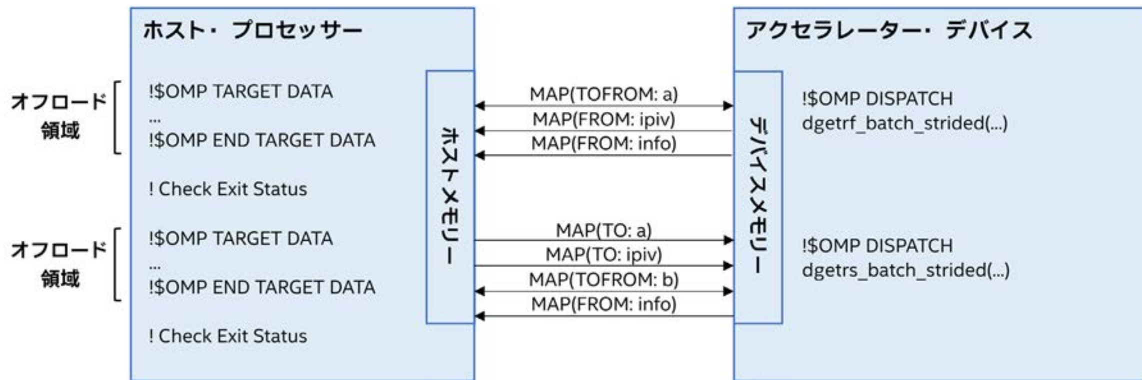

図 2. リスト 2 に示した 2 つの OpenMP* ターゲットオフロード領域のホストとデバイス間のデータ転送。各矢印は、ホストとデバイスのメモリー間のデータ転送を示します。

ただし、ホストとデバイス間のデータ転送は、『oneAPI GPU 最適化ガイド』の「データ転送とメモリー割り当てを最小化する」で説明しているように、ランタイム・ライブラリーからデバッグ情報を要求することで、データの移動を直接確認できます。テスト問題は小さすぎて高速化を証明できないため、問題のサイズを増やします。それぞれ 1 つの RHS がある 8 つの 8,000x8,000 線形システムのバッチを (倍精度で) 解いてみましょう。

```
$ OMP_TARGET_OFFLOAD=MANDATORY ZE_AFFINITY_MASK=0.0 LIBOMPTARGET_DEBUG=1 \
> ./lu_solve_ex1 8000 8 1 1 >& lu_solve_ex1.out

$ grep Moving lu_solve_ex1.out

Libomptarget --> Moving 64 bytes (hst:0x00007fffa0074c28) -> (tgt:0x0000000003871008)
Libomptarget --> Moving 88 bytes (hst:0x00007fffa0074b48) -> (tgt:0x0000000003871088)
Libomptarget --> Moving 4096000000 bytes (hst:0x00007f1d07fff200) -> (tgt:0x00007f1c0dedd000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffa0074cd8) -> (tgt:0x0000000003871108)
Libomptarget --> Moving 4096000000 bytes (tgt:0x00007f1c0dedd000) -> (hst:0x00007f1d07fff200)
Libomptarget --> Moving 512000 bytes (tgt:0x00007f1d8ce05000) -> (hst:0x00007f1da079b280)
Libomptarget --> Moving 64 bytes (tgt:0x0000000003882000) -> (hst:0x00007f1da0a33dc0)
Libomptarget --> Moving 64 bytes (hst:0x00007fffa0074c28) -> (tgt:0x0000000003871008)

Libomptarget --> Moving 512000 bytes (hst:0x00007f1da075b240) -> (tgt:0x000000000576b000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffa0074c78) -> (tgt:0x0000000003871088)
Libomptarget --> Moving 512000 bytes (hst:0x00007f1da079b280) -> (tgt:0x00007f1d8ce05000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffa0074b48) -> (tgt:0x0000000003871108)
Libomptarget --> Moving 4096000000 bytes (hst:0x00007f1d07fff200) -> (tgt:0x00007f1c0dedd000)
Libomptarget --> Moving 88 bytes (hst:0x00007fffa0074cd8) -> (tgt:0x0000000003871188)
Libomptarget --> Moving 512000 bytes (tgt:0x000000000576b000) -> (hst:0x00007f1da075b240)
Libomptarget --> Moving 64 bytes (tgt:0x0000000003882000) -> (hst:0x00007f1da0a33dc0)
```

OpenMP* `MAP` 構文を使用して明示的に転送される配列をハイライトし、2 つの OpenMP* ターゲット領域を区切るスペースを追加しました。ハイライトされていないデータ移動は、ターゲットデバイスにマップされている配列のドープベクトル (英語) です。一般に小さいため、無視できます。

1 つ目のターゲット領域で、配列 `a` (4,096,000,000 バイト) がホストからターゲットへ (`hst → tgt`)、およびターゲットからホストへ (`tgt → hst`) 転送され [`MAP(TOFROM:a)`]、2 つ目のターゲット領域でターゲットデバイスに転送されます [`MAP(TO:a)`]。ピボット (`ipiv`、512,000 バイト) は、1 つ目のターゲット領域のデバイスで計算されて取得され [`MAP(FROM:ipiv)`、`tgt → hst`]、2 つ目のターゲット領域のデバイスに転送されて [`MAP(TO:ipiv)`、`hst → tgt`]、解が計算されます。配列 `b` (512,000 バイト) に格納されている RHS は、デバイスに転送され、ソリューション・ベクトルで上書きされて、ホストに転送されます [`MAP(TOFROM:b)`]。ステータス配列 `info` (64 バイト) は、両方のターゲット領域の最後にデバイスから取得されます [`MAP(FROM:info)`、`tgt → hst`]。

© 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。

不連続メモリー間のデータの移動には時間とエネルギーがかかるため、ハイライトされている移動（合計 12,290,048,128 バイト）に細心の注意を払う必要があります。これを念頭に、初期の実装を改善する方法を見てみましょう。

前号の LU 因数分解の記事では、1 つの OpenMP* ターゲット領域で十分な場合に 2 つのターゲット領域を使用することの短所を説明しました。まず、制御フローをターゲットデバイスに転送すると、オーバーヘッドが発生します。初期の実装では 2 回発生します（**リスト 2**）。次に、冗長なホストとデバイス間のデータ転送が必要です（**図 2**）。線形システムを解くときに必要なのは、入力配列と RHS をデバイスにコピーして、解をデバイスから取得することです。ステータス配列も取得する必要がありますが、これらは比較的小さいものです。ピボットはデバイスでのみ使用されるため、`ipiv` 配列をデバイスメモリーに直接割り当てることにします [`MAP(ALLOC:ipiv(1:stride_ipiv, 1:batch_size))`]。2 つの OpenMP* ターゲット領域を融合すると、クリーンで簡潔なコードになり（**リスト 3**）、データの移動が少なくて済みます（**図 3**）。

```
!$OMP TARGET DATA MAP(TO:a) MAP(TOFROM: b)                    &
                  MAP(ALLOC:ipiv(1:stride_ipiv, 1:batch_size)) &
                  MAP(FROM:info_rf, info_rs)

    !$OMP DISPATCH
    CALL dgetrf_batch_strided(n, n, a, lda, stride_a, &
                              ipiv, stride_ipiv, batch_size, info_rf)

    !$OMP DISPATCH
    CALL dgetrs_batch_strided('N', n, nrhs, a, lda, stride_a, &
                               ipiv, stride_ipiv, &
                               b, ldb, stride_b, batch_size, info_rs)
!$OMP END TARGET DATA

IF (ANY(info_rf .ne. 0)) THEN
    PRINT *, 'Error: getrf_batch_strided returned with errors.'
ELSEIF (ANY(info_rs .ne. 0)) THEN
    PRINT *, 'Error: getrs_batch_strided returned with errors.'
ELSE
    PRINT *, 'Computation executed successfully'
ENDIF
```

リスト 3. 1 つの OpenMP* ターゲット領域を使用して oneMKL でバッチ処理された線形システムを解く。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。

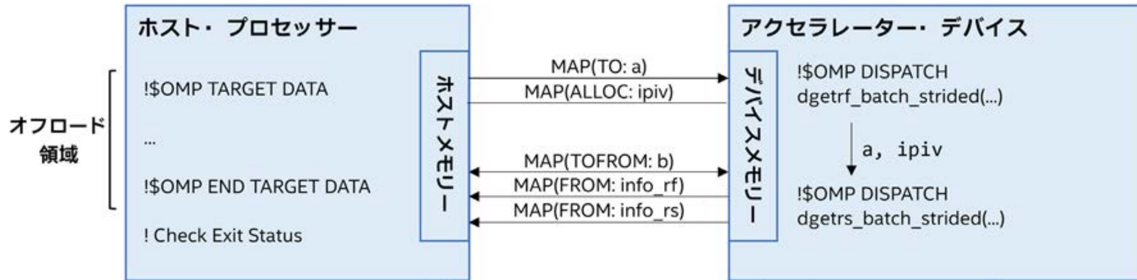

図 3. リスト 3 の OpenMP* ターゲットオフロード領域におけるホストとデバイス間のデータ転送。
各矢印（アクセラレーター・デバイス内を除く）は、ホストとデバイスのメモリー間のデータ転送を示します。

この効果はデバッグ情報で確認できます。再度、OpenMP* MAP 構文を使用して明示的に転送される配列をハイライトして示します。

```
Libomptarget --> Moving 88 bytes (hst:0x00007ffe443ba9c8) -> (tgt:0x0000000001e55008)
Libomptarget --> Moving 64 bytes (hst:0x00007ffe443bad68) -> (tgt:0x0000000001e55088)
Libomptarget --> Moving 64 bytes (hst:0x00007ffe443bad18) -> (tgt:0x0000000001e55108)
Libomptarget --> Moving 512000 bytes (hst:0x00007f39ada86240) -> (tgt:0x00007f399a06d000)
Libomptarget --> Moving 88 bytes (hst:0x00007ffe443bae38) -> (tgt:0x0000000001e55188)
Libomptarget --> Moving 4096000000 bytes (hst:0x00007f3913fff200) -> (tgt:0x00007f3819edd000)
Libomptarget --> Moving 88 bytes (hst:0x00007ffe443bae98) -> (tgt:0x0000000001e55208)
Libomptarget --> Moving 512000 bytes (tgt:0x00007f399a06d000) -> (hst:0x00007f39ada86240)
Libomptarget --> Moving 64 bytes (tgt:0x0000000001e67040) -> (hst:0x00007f39add5fd80)
Libomptarget --> Moving 64 bytes (tgt:0x0000000001e67000) -> (hst:0x00007f39add5fdc0)
```

ドープベクトルを無視すると、**リスト 3** の 1 つの OpenMP* ターゲット領域では、ホストとデバイス間のデータ転送（4,097,024,128 バイト）が初期の実装の 2 つのターゲット領域の転送（12,290,048,128 バイト）（**リスト 2**）よりも大幅に少なくなります。

ホストとデバイス間のデータ転送を最小化することは、ヘテロジニアス・コンピューティングにおける最優先事項です。行列サイズが大きくなるほど、2 つ目のサンプルのコード（**リスト 3**）は、オリジナルのコード（**リスト 2**）よりもパフォーマンスが大幅に向上します（**表 1**）。

© 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。

行列サイズ	CPU 時間	GPU 時間 (サンプル 1)	GPU 時間 (サンプル 2)
1,000x1,000	0.13	1.72	1.70
2,000x2,000	0.60	0.84	0.72
4,000x4,000	3.90	2.51	2.10
8,000x8,000	19.15	5.98	4.55
12,000x12,000	63.11	12.55	9.51
16,000x16,000	139.45	21.33	15.71

表 1. 2 x 第 4 世代インテル® Xeon® Platinum 8480+ プロセッサー（2.0GHz）（CPU）、インテル® データセンター GPU マックス・シリーズ 1550（GPU）、528GB メモリーを搭載した Linux*（Ubuntu* 20.04 x64、5.15.47 カーネル）システムでさまざまな行列サイズのバッチ処理された線形システムを解くのにかかった時間（秒）。GPU テストは 1 つのタイルのみ使用。各線形システムの右辺は 1 つ。各実験は 5 回実行。oneMKL 関数の JIT コンパイルのオーバーヘッドが含まれるため最初の実行は破棄。レポートされている時間は残りの 4 つの実行の合計。

この記事で使用した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。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。

OpenMP* ターゲット構文には、アクセラレーターのオフロードをプログラマーが細かく制御できるようにする、多くのオプションが用意されています。OpenMP* 仕様（英語）とサンプルコード（英語）、および 2 つのチュートリアル「OpenMP* API を使用した GPU プログラミングの概要」（英語）と「データ転送とメモリー割り当てを最小化する」（英語）を確認してみてください。

© 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。