OpenMP* のアクセラレーター・オフロード

ヘテロジニアス・アーキテクチャー全体にわたる移植性

Nitya Hariharan インテル コーポレーション アプリケーション・エンジニア
Rama Kishan Malladi インテル コーポレーション パフォーマンス・モデリング・エンジニア

OpenMP* 標準は、バージョン 4.0 からアクセラレーター・オフロードをサポートしています。これらのプラグマは、エンドユーザーがデータと計算を GPU などのデバイスへオフロードできるようにします。これにより、移植性に優れたヘテロジニアス並列コードを容易に記述できます。この記事では、いくつかの OpenMP* オフロードプラグマと、その使用方法をサンプルコードを用いて説明します。また、OpenACC* から OpenMP* への移行例も紹介します。

OpenACC* から OpenMP* への移行

OpenACC* は、NVIDIA GPU 向けのプラグマベースのプログラミング手法ですが、ほかのベンダーによりサポートされていないため、1 つのプラットフォームに限定されています。一方、OpenMP* オフロードは、oneAPI フレームワーク、NVIDIA の HPC SDK、AMD の ROCm* スタック、IBM の XL コンパイラー・スイートなど、業界で幅広く

サポートされています。OpenACC* プラグマは OpenMP* へほぼ 1 対 1 でマッピングできます (**表 1**)。そのため、通常は既存の OpenACC* コードから OpenMP* へ簡単に移行できます。**表 1** は、一般的な OpenACC* プラグマと同等の OpenMP* プラグマです。

OpenACC* プラグマ	OpenMP* プラグマ	用途
`#pragma acc parallel`	`#pragma omp target teams`	GPU 上でスレッドのチームを作成して、各チームのマスタースレッドが領域を実行します。
`#pragma acc parallel loop gang worker vector collapse(2)`	`#pragma omp target teams distribute parallel for simd collapse(2)`	GPU ハードウェア・スレッドに作業を分散して、計算を並列に実行します。
`#pragma acc kernels loop reduction(+:norm)`	`#pragma omp parallel for reduction(+:norm)`	リダクションを並列に実行します。
`#pragma acc data copy(A[0:Sz])`	`#pragma omp target data map(tofrom: A[0:Sz])`	ターゲットデバイスとの間でデータをコピーします。
`#pragma acc update host(A[0:Sz])`	`#pragma omp target update from(A[0:Sz])`	デバイスからホスト上のデータを更新します。
`#pragma acc data copyin(A[0:Sz])`	`#pragma omp target data map(alloc:A[0:Sz])`	デバイス上にメモリーを割り当てます。
`#pragma acc update device(X)`	`#pragma omp target update to(X[0:Sz])`	ホストからデバイス上のデータを更新します。
`#pragma acc loop vector`	`#pragma omp simd`	ベクトル化します。
`#pragma acc atomic update`	`#pragma omp atomic update`	メモリー位置をアトミックに更新します。

表 1. 一般的な OpenACC* プラグマと同等の OpenMP* プラグマ

図 1a と **1b** は、OpenACC* から OpenMP* へ移行されたコードスニペットです。これは、電波天文パッケージ **tConvolveACC** (英語) のカーネルです。OpenACC* プラグマ **`#pragma acc parallel loop`** は、OpenMP* オフロードプラグマ **`#pragma omp target parallel for`** とターゲットデバイスとの間の明示的なデータ転送プラグマに置き換えられます。OpenACC* 実装では、暗黙のコピーや統合共有メモリー割り当てを使用してデータ転送を管理している可能性があります。

```
degridKernelACC(...)
{
   ...
   #pragma acc parallel loop
   for (dind = 0; dind < d_size; ++dind) {
      ...
   }
}
...
gridKernelACC(...)
{
   ...
   #pragma acc parallel loop
   ...
         #pragma acc atomic update
         gptr_re[0] = gptr_re[0] + cval.real();
   ...
}
```

図 1a. OpenACC* で記述された tConvolveACC 実装からのサンプルカーネル

```
degridKernelOmpOffload(...)
{
   ...
   #pragma omp target parallel for                    \
           map(tofrom:d_data[0:d_size])               \
           map(to:d_grid[:grid.size()])               \
           map(to:d_C[:C.size()])                     ...
   for (dind = 0; dind < d_size; ++dind) {
     ...
   }
}
...
gridKernelOmpOffload(...)
{
   ...
   #pragma omp target teams distribute parallel for      \
           map(tofrom:d_grid[:grid.size()])              ...
   ...
         #pragma omp atomic update
         gptr_re[0] = gptr_re[0] + cval.real();
   ...
}
```

図 1b. OpenMP* で記述された tConvolveACC 実装からのサンプルカーネル

インテル® プラットフォーム上での OpenMP* オフロード

オフロードコードをビルドして実行するために必要な手順を見てみましょう。**インテル® oneAPI ベース・ツールキット** 2021.2.0 で以下のコンパイラー・オプションを使用して、OpenMP* オフロードコードをテストします。

```
-fiopenmp -fopenmp-targets=spir64="-mllvm \
-vpo-paropt-enable-64bit-opencl-atomics=true \
-fp-model=precise"
```

`-fiopenmp` と **`-fopenmp-targets=spir64`** の 2 つの新しいオプションは、GPU 向けのファットバイナリーを生成するようにコンパイラーに指示します。**`-vpo-paropt-enable-64bit-opencl-atomics=true`** コンパイラー・オプションは、アトミック操作とリダクション操作を有効にします。詳細は、**オンライン・ドキュメント**を参照してください。

GPU 上で OpenMP* オフロードコードを実行するため、ユーザーは **`OMP_TARGET_OFFLOAD`** 環境変数を設定する必要があります (GPU が利用できない場合、ランタイムエラーになります)。また、ユーザーは**レベルゼロ** (英語) または OpenCL* バックエンドを選択できます。

```
export OMP_TARGET_OFFLOAD = MANDATORY
export LIBOMPTARGET_PLUGIN = {LEVEL0|OPENCL}
```

LIBOMPTARGET_DEBUG (英語) 環境変数を 1 以上に設定すると、GPU オフロードのデバッグ情報を取得できます。**図 2a** では、**`tConvolveACC`** の OpenMP* オフロードカーネルをレベルゼロプラグインで実行した場合のデバッグ情報をハイライト表示しています。2 つのオフロード領域は、**`Benchmark`** クラスの **`gridKernelACC`** 関数と **`degridKernelACC`** 関数にあります。**図 2b** は、**`map`** 節でターゲットデバイスへ転送される変数を示します。**図 2c** は、ホストからターゲットデバイスへ転送されるデータを示します。計算に必要なすべてのデータがデバイスで利用可能になると、**図 2a** の下部に示すように、カーネルが実行されます。

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。

```
Target LEVEL0 RTL -->   0:
__omp_offloading_35_198bc1a__ZN9Benchmark13gridKernelACCERKSt6vector
Target LEVEL0 RTL -->   1:
__omp_offloading_35_198bc1a__ZN9Benchmark15degridKernelACCERKSt6vector

...

Libomptarget --> Launching target execution
__omp_offloading_35_198bc1a__ZN9Benchmark13gridKernelACCERKSt6vector with pointer
0x0000000000b72070 (index=0)
```

図 2a. tConvolveACC の OpenMP* オフロードカーネルのクラスと関数を赤色でハイライト表示

```
Libomptarget --> Entry  0: Base=0x00007f468ce16010, Begin=0x00007f468ce16010, Size=15310080,
Type=0x21, Name=d_iw[:this->wPlane.size()]

Libomptarget --> Creating new map entry with HstPtrBegin=0x00007f468ce16010,
TgtPtrBegin=0xffffd556aaa00000, Size=15310080, Name=d_iw[:this->wPlane.size()]
```

図 2b. tConvolveACC の OpenMP* オフロードカーネルの変数を青色でハイライト表示

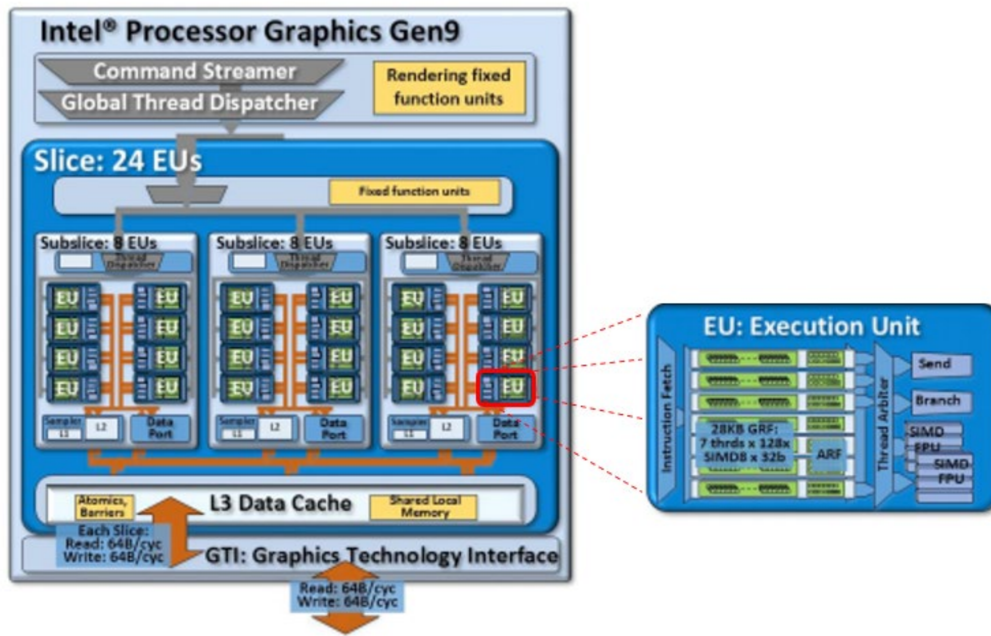
```
Libomptarget --> Moving 15310080 bytes (hst:0x00007f468ce16010) -> (tgt:0xffffd556aaa00000)
Target LEVEL0 RTL --> Copied 15310080 bytes (hst:0x00007f468ce16010) -> (tgt:0xffffd556aaa00000)
```

図 2c. tConvolveACC の OpenMP* オフロードカーネルのデータ転送を緑色でハイライト表示

OpenMP* スレッドをターゲットデバイスへマップ

実行時に、OpenMP* スレッド階層がターゲットデバイスへマップされます。**`#pragma omp teams`** 構文はチームのリーグを作成して、各チームの最初のスレッドが領域を実行します。**`#pragma omp distribute`** 構文は、チームの最初のスレッドにワークを分配して、各チームはサブスライス（インテル® GPU）にスケジュールされます。各チーム内では **`parallel for`** 節によりワークがさらに並列化され、チームのスレッドが実行ユニット(EU)スレッドにマップされます。最後に、**`#pragma omp simd`** 構文は、EU ベクトルレーンを使用してベクトル化されたコードを実行します。チーム内のスレッドは、ワークシェア構文の最後で同期されます。**図 3** は、1 スライス、3 サブスライス、サブスライスごとに 8 EU、EU ごとに 7 スレッド、EU ごとに SIMD ベクトル処理ユニットがある第 9 世代インテル® プロセッサー・グラフィックス（Gen9）の図です。Gen9 の対応するユニットへの OpenMP* オフロードプラグマのマップも示しています。

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。



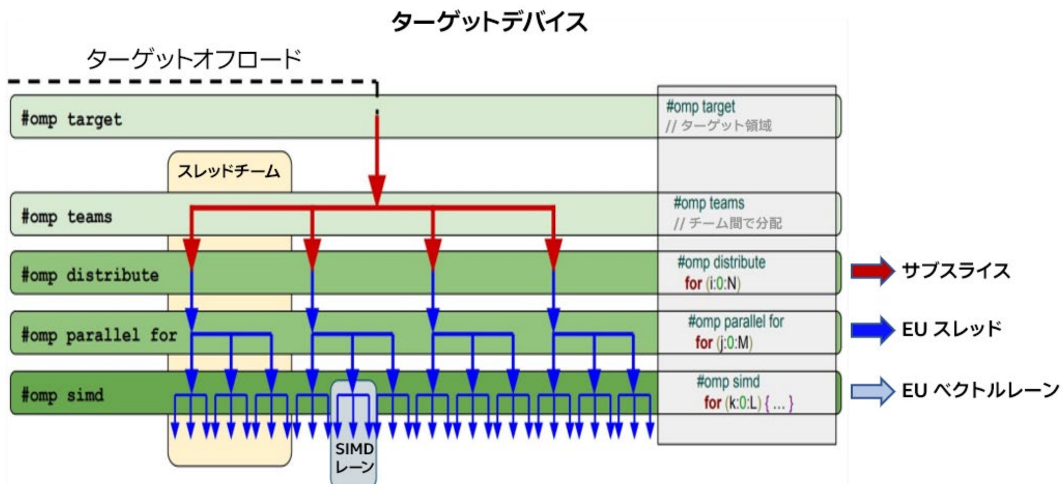


図 3. 第 9 世代インテル® プロセッサー・グラフィックス上のハードウェア機能への OpenMP* オフロードのマップ（「OpenMP* オフロードの確認と検証 : ワークフローと 5.0 への道」（英語）より引用)

ターゲットデバイスとの間で効率良くデータ転送を行う OpenMP* プラグマ

アプリケーションをビルドしていくつかのカーネルをターゲットにオフロードできたら、次のステップはデータ転送などの最適化の可能性を調査することです。OpenMP* には、ホストとターゲット間で効率良いデータ転送を実装するプラグマが用意されています。以下に示す **tHogbomCleanACC** の例では、**HogbomClean** 関数に 2 つのオフロードターゲットがあります。単純な OpenMP* オフロードでは、ターゲットの起動時にデータ転送が行われます。**図 4a** のコードスニペットに示すように、これを **g_niters** のループで繰り返す場合、問題はさらに悪化します。

```
HogbomClean(...)
{
...for (unsigned int i = 0; i < g_niters; ++i)
    {
      findPeakOffload(resdata, absPeakVal, absPeakPos, ressize);...
      subtractPSFOffload(psfdata, psfWidth, psfsize, resdata, ...);...
    }...
}

subtractPSFOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}

findPeakOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}
```

図 4a. 2 つの OpenMP* オフロードカーネルの単純な実装では不要なデータ転送が発生

図 4b は、より効率良くデータ転送を行うように最適化された **HogbomClean** 関数の実装です。**#pragma omp target data map** 構文は、ターゲット上でデータが保持されるスコープを定義します。このスコープ内のカーネルオフロードは、(ハンドルを使用して) データを再利用できます。後続のオフロードカーネルへの **map** 呼び出しでは、データ転送は不要です (ただし、明示的に転送が必要とされているものは除く)。

oneAPI
多様なワークロードには多様なアーキテクチャーが必要
インテル® oneAPI ツールキットを使用して、ヘテロジニアス・アプリケーションを素早く正確に開発
ツールキットの詳細 >

```
HogbomClean(...)
{
  ...
  #pragma omp target data map(tofrom: resdata...
  {
    for (unsigned int i = 0; i < g_niters; ++i)
    {
      findPeakOffload(resdata, absPeakVal, absPeakPos, ressize);...
      subtractPSFOffload(psfdata, psfWidth, psfsize, resdata, ...);...
    }...
  }...
}

subtractPSFOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}

findPeakOffload(...)
{
...#pragma omp target teams distribute parallel for map(to: resdata...
}
```

図 4b. 一度コピーしてデータ転送を再利用する効率良い OpenMP* の例

関数バリアントのディスパッチのサポート強化

OpenMP* オフロード仕様は、基本関数の代わりに条件付きで呼び出し可能な関数バリアントをサポートしています。インテル固有の OpenMP* オフロード関数バリアント API の実装は、**`#pragma omp target variant dispatch`** でサポートされます。このプラグマは、関数呼び出しの周囲に条件付きのディスパッチ・コードを発行するようにコンパイラーに指示します。ターゲットデバイスが利用可能な場合、基本関数の代わりに関数バリアントが呼び出されます。**図 5a**、**5b**、および **5c** は、**`target variant dispatch`** の例です。関数バリアントは基本関数と同じ引数に加えて、**`void *`** 型の最後の引数がなければなりません。

```
findPeakOffload(..., void *p)
{
  ...
  #pragma omp target teams distribute parallel for
          reduction(max:threadAbsMaxVal) map(to:data[0:size])
  for (size_t i = 0; i < size; ++i) {
    if ( abs(data[i]) > threadAbsMaxVal)
      threadAbsMaxVal = abs(data[i]);
  }

  #pragma omp target teams distribute parallel for
          map(to:data[0:size]) map(from:tmpPos)
  for (size_t i = 0; i < size; ++i) {
    if (abs(data[i]) == threadAbsMaxVal)
      tmpPos = i;
  }
  maxVal = data[tmpPos];
  maxPos = tmpPos;
}
```

図 5a. ターゲットデバイスで実行する関数バリアント findPeakOffload

```
#pragma omp declare variant(findPeakOffload)
        match(construct={target variant dispatch})
findPeakBase(...)
{
  #pragma omp parallel
  {
    ...
    #pragma omp for schedule(static)
    for (size_t i = 0; i < size; ++i) {
      if (abs(image[i]) > abs(threadAbsMaxVal)) {
        threadAbsMaxVal = image[i];
        threadAbsMaxPos = i;
      }
    }

    #pragma omp critical
    if (abs(threadAbsMaxVal) > abs(maxVal)) {
      maxVal = threadAbsMaxVal;
      maxPos = threadAbsMaxPos;
    }
  }
}
```

図 5b. ホストで実行する基本関数 findPeak

```
HogbomClean(...)
{
  ...
  #pragma omp target data map(to: psfdata[0:psfsize])
                          map(tofrom: resdata[0:ressize])
  {
    for (unsigned int i = 0; i < g_niters; ++i)
    {
      #pragma omp target
      findPeakBase(resdata, absPeakVal, absPeakPos, ressize);
      ...
    }
  }
}
```

図 5c. 呼び出しはホストバージョンでなければならない—ターゲットデバイスが利用可能な場合はオフロードターゲット関数が実行され、そうでない場合はホストバージョンが実行される

まとめ

OpenMP* 標準のプラットフォームやベンダーに依存しないデバイス・オフロード・サポートにより、ユーザーは同じコードベースで複数のヘテロジニアス・アーキテクチャーに容易に対応できます。そのため、ユーザーやハードウェア / ソフトウェア・ベンダーの間で、OpenMP* のヘテロジニアスな並列処理の採用が進むと予想されます。

関連情報

1. **https://techdecoded.intel.io/resources/using-openmp-accelerator-offload-for-programming-heterogeneous-architectures/** (英語)