インテル® コンパイラーを利用した高度な OpenMP デバイスオフロード

この記事は、インテルのウェブサイトで公開されている「Advanced OpenMP* Device Offload with Intel® Compilers」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

OpenMP (英語) のオフロード機能は、マルチベンダー、マルチプラットフォーム、マルチ言語に対応し、市場の主要なプレーヤーすべてに、包括的で活発に進化するアクセラレーター・プログラミング・フレームワークを提供します。OpenMP のスレッド化と並列オフロードは、Kokkos (英語) や RAJA (英語) など、最近成長しつつあるパフォーマンス・ポータビリティー・エコシステムのバックエンドとしても機能します。ソースコードを変更することなく、数行の OpenMP ディレクティブを追加するだけで、GPU での並列実行を利用しつつ、必要に応じて CPU のみでも実行可能な実行ファイルを生成することができます。

OpenMP コミュニティーは、2021 年 11 月にバージョン 5.2 がリリースされて以来、積極的に仕様を進化させてきました。最新バージョンの OpenMP 6.0 仕様は、2024 年 11 月にリリースされる予定です。新しいリリースでは、GPU へのオフロードサポートが大幅に強化されています。
[訳者注: OpenMP 6.0 は 2024 年 11 月にリリースされました。
https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf (英語)]

2023 年 4 月発行の The Parallel Universe 52 号に掲載された「Fortran と OpenMP でヘテロジニアス・プログラミングの課題を解決」では、OpenMP の GPU オフロード・プログラミング・フレームワークの基礎を紹介しました。

インテルは長年、オープンソース・コミュニティー、特に OpenMP に積極的に貢献してきました。インテル® oneAPI DPC++/C++ コンパイラーとインテル® Fortran コンパイラーの最新リリースでは、OpenMP 5.2 の機能強化と OpenMP 6.0 Technical Report (TR) 12 および TR 13 の提案で導入された多くの新機能を実装しています。これには、最新世代のインテル® Arc™ GPU、インテル® データ センター GPU、統合型インテル® Arc™ Xᵉ2 GPU のサポートが含まれます。

OpenMP の高速コンピューティングとデバイス・オフロード・サポートが大きく進歩したことに伴い、ここでは、このトピックを再検討し、前回の記事を拡張して、OpenMP デバイスオフロードの最近の進化について説明します。

進化を続けるデバイス・オフロード・サポート

本記事では、前回の記事を更新および拡張し、最新のインテル® コンパイラーの高度な機能について説明します。これらのコンパイラーは、インテル® oneAPI ベース・ツールキットおよびインテル® HPC ツールキット 2025.0 の一部として、2024 年 11 月に公開される予定です。
[訳者注: これらのコンパイラーを含むインテル® HPC ツールキット 2025.0 は 2024 年 11 月にリリースされました。]

包括的で完全な GPU プログラミング・フレームワークには通常、次の 5 つの要素が含まれますが、これらに限定されるものではありません。

1. ヘテロジニアス・メモリー・アーキテクチャーでのデータ管理
2. GPU スレッド管理の実行ポリシー/設定
3. GPU 向けに最適化されたライブラリーやその他のコンパイルユニットに既存の API を活用
4. GPU 命令の選択/最適化
5. 同時スレッド実行の制御フロー/分岐制御

前回の記事では、1、2、3 の OpenMP 基本機能 (4 と 5 は記事の対象外) と、明示的なディレクティブを必要としない Fortran 並列処理 (DO PARALLEL) のインテルの OpenMP オフロード実装について説明しました。この記事では、インテル® コンパイラー 2025.0 で追加される 6 つの高度な OpenMP 機能に関する情報を提供します。

以下の機能が追加されます。

1. GPU 共有ローカルメモリー上のデータを管理する GROUPPRIVATE ディレクティブ (OpenMP 6.0)
2. LOOP ディレクティブ
3. TEAMS ディレクティブの REDUCTION 節
4. 実行ポリシーの TARGET ディレクティブ (OpenMP 5.1) の NOWAIT 節
5. 既存の API を利用し、SYCL と OpenMP の相互運用性を実現する DISPATCH ディレクティブ (OpenMP 6.0) の INTEROP 節
6. Fortran 標準の DO CONCURRENT 並列処理の自動オフロードサポートの拡張: レデュース操作、暗黙的なマップ型修飾子 PRESENT、スレッド起動ヒューリスティックの改善

1. GROUPPRIVATE ディレクティブ

OpenMP データ環境ディレクティブと節によって処理されます。「Fortran と OpenMP でヘテロジニアス・プログラミングの課題を解決」で説明したように、データ・マップ・ディレクティブとそのバリアントはホストデバイスのデータ移動を処理します。デバイスメモリーについては、PRIVATE に関連する一連のディレクティブと節があります。

OpenMP 仕様の 5.2 から 6.0 へのアップデートの一環として、競合グループに関して変数をプライベート化することを指定する GROUPPRIVATE ディレクティブが追加されました。GROUPPRIVATE ディレクティブは、各競合グループが独自のコピーを受け取るように変数を複製することを指定します。変数の各コピーは、作成時に初期化されません。GROUPPRIVATE 変数の有効期間は、チームの有効期間に制限されます。あるコンパイルユニット内で GROUPPRIVATE ディレクティブで指定された変数は、その変数が宣言されているすべてのコンパイルユニットにおいて、GROUPPRIVATE ディレクティブで指定されていないければなりません。GROUPPRIVATE 変数は、C/C++ では static、Fortran では SAVE 属性が設定されなければなりません。

次に例を示します。

```
program groupprivate_example
  use omp_lib
  integer, save  :: x
  !$omp groupprivate(x)
  !$omp target teams num_teams(4)
  x = omp_get_team_num()
 print *, x
  !$omp end target teams
end program
```

2. LOOP ディレクティブ

LOOP ディレクティブは OpenMP 5.1 で導入され、collapse された反復がバインド・スレッド・セットのコンテキストで実行されることを指定します。TARGET 領域で LOOP ディレクティブを使用すると、コンパイラーはそれを DISTRIBUTE と PARALLEL DO/PARALLEL FOR に変換します。LOOP は、TARGET、TEAMS、PARALLEL と組み合わせて、さまざまな結合ディレクティブを形成できます。次に例を示します。

最新の OpenMP 仕様で定義されているように、LOOP は実装依存の方法で並列処理生成ディレクティブとペアになるワークシェア・ディレクティブです。LOOP ディレクティブは、collapse された反復がバインド・スレッド・セットのコンテキストで実行されることを指定します。インテルの実装では、LOOP は抽象化されたディレクティブであり、コンパイル中に適切な結合ディレクティブに変換されます。プログラマーは、最適化レポートから実際に使用された結合ディレクティブを確認できます。この単純な入れ子のループの例では、外側の !$OMP LOOP は !$OMP DISTRIBUTE に変換され、内側の !$OMP LOOP は !$OMP PARALLEL DO に変換される必要があります。

```
program loop_directive
        implicit none
        integer :: i,j,bo,bi
        real, allocatable :: A(:,:), B(:,:)
        bo=100 ; bi=200
        allocate(A(bo,bi))
        allocate(B(bo,bi))
        A=1.0d0 ; B=0.0d0
        !$omp target teams map(tofrom:A,B)
        !$omp loop
        DO i=1,bo
          !$omp loop order(concurrent)
          DO j=1,bi
             B(i,j) = A(i,j)
          END DO
          !$omp end loop
        END DO
        !$omp end loop
        !$omp end target teams
        print *,"sum(B)=",sum(B)
end program loop_directive
```

3. TEAMS ディレクティブの REDUCTION 節

REDUCTION 節は、以前はループ構造で使用可能であり、主に PARALLEL FOR/PARALLEL DO ディレクティブで、同じ並列領域内のスレッド間のレデュースに使用されていました。新しい OpenMP 仕様では、TEAMS ディレクティブでも REDUCTION 節を使用できるようになり、複数のチーム間でレデュースを実行できます。

REDUCTION 節には、リダクションが計算される領域を定義する reduction scoping 節とリダクションに参加するタスク/SIMD レーンを定義する reduction participating 節があります。各リスト項目について、暗黙のタスクまたは SIMD レーンごとにプライベート・コピーが作成され、リダクション識別子の初期化値で初期化されます。領域の終了後、リダクション識別子に関連付けられた結合子を使用して、元のリスト項目はプライベート・コピーの値で更新されます。

TEAMS ディレクティブの場合、リーグ内の各チームの最初のタスクに対して各リスト項目の 1 つ以上のプライベート・コピーが作成され、PRIVATE 節が使用された場合と同じ動作になります。

次の例は、REDUCTION 節を PARALLEL ディレクティブと TEAMS ディレクティブの両レベルで使用する方法を示します。libomptarget デバッグ出力は、リダクション変数が適切に処理されていることを示しています。

```
program reduction_clause
  implicit none
  integer ic,ib,ia,i1,i2
  integer :: E2_t, E2, E3
  i2 = 100;  i1 = 16; E2 = 0; E3 = 0
  !$omp target teams distribute reduction(+:E2,E3) private(E2_t) map(tofrom:E2)
  DO ic=1,i1
     E2_t = 0
     !$omp parallel do reduction(+:E2_t) collapse(2)
     DO ic=1,i2
        DO ia=1,i2
           E2_t=E2_t + 1
        ENDDO
     ENDDO
     !$omp end parallel do
     E2 = E2 + E2_t
     E3 = E3 + E2_t
  ENDDO
  !$omp end target teams distribute
  print *,"E2, E3=",E2,E3
end program reduction_clause
```

4. TARGET ディレクティブの NOWAIT 節

TARGET 領域の起動時に NOWAIT 節を使用すると、ホストデバイスから非同期で実行できるようになります。スレッドが TARGET ディレクティブに到達して制御フローをデバイスに渡す場合、プログラマーは OMP TARGET ディレクティブの後に NOWAIT 節を追加して、制御フローをホストに戻すことができます。

その後、ホストとデバイスの両方が、事前に決定された同期ポイントまで個別のワークフローを実行します。同期ポイントを設定する実用的な方法の 1 つは、ホストとデバイスのデータを同期する前に !$OMP TASKWAIT を挿入することです。つまり、ホストとデバイスの両方のワークフロー (タスク) がデータ同期のポイントに到達すると、分岐した 2 つの制御フローはマージしてホストに戻ります。

次の例は、デバイスとホストで非同期にコードを実行する方法を示します。TARGET ディレクティブの NOWAIT 節は、TARGET 領域の実行完了を待機する間、ターゲットタスクのスレッドがほかのワークを実行できるようにします。したがって、TARGET 領域はデバイス上で非同期に実行できます (ターゲットタスクの実行完了を待機する間、ホストのスレッドがアイドル状態になる必要はありません)。

この例では、2 つの配列 A と C が計算されます (計算は何度も繰り返され、ホストデバイスのデータ転送時間に対するデバイス計算の影響が増幅されます)。配列 A はデバイス上で計算され、配列 B はホスト上で同時に計算されます。

明示的なバリア同期により、ホストとデバイスのデータの同期前にターゲットタスク (非同期ターゲット実行) が完了することが保証されます。詳細については、OpenMP 仕様の用語集で「barrier」を参照してください。

```
$ cat nowait_clause.f90

program nowait_clause
        implicit none
        integer :: i,j,k, bo,bi
        real(8), dimension(:,:), allocatable :: A, B, C, D
        bo=1000 ; bi=1000
        allocate(A(bo,bi), B(bo,bi), C(bo,bi), D(bo,bi))
        A=2.0 ; B=1.0 ; C=2.0 ; D=1.0
        !$omp target enter data map(to: A, B)
        !$omp target nowait
        !$omp teams distribute parallel do
        do i=1,bo ; do j=1,bi ; do k=1,300000
            A(i,j)=A(i,j)*B(i,j)
        enddo ;  enddo ; enddo
        !$omp end teams distribute parallel do
        !$omp end target
        do i=1,bo ; do j=1,bi ; do k=1,20000
            C(i,j)=C(i,j)*D(i,j)
        enddo ; enddo ; enddo
        !$omp taskwait
        !$omp target exit data map(from: A, B)
        print *,"sum(A)=",sum(A)," sum(C)=",sum(C)
end program nowait_clause
```

5. 関数バリアントの DISPATCH ディレクティブの INTEROP 節

DISPATCH ディレクティブは、呼び出しサイトに固有の引数の追加をサポートするため INTEROP 節で拡張されました。INTEROP 節は、DISPATCH ディレクティブ内のターゲット呼び出しでバリアントの置換が発生したときに関数バリアントに渡す追加の引数を指定します。INTEROP 変数リスト内の変数は、INTEROP 節で指定された順序で渡されます。INTEROP 節が DISPATCH ディレクティブで指定されている場合、ターゲット呼び出しに対応する DECLARE VARIANT ディレクティブには、INTEROP 節のリスト項目の数以上のリスト項目を持つ APPEND_ARGS 節が必要です。

例: OpenMP と SYCL の相互運用性

次の例は、相互運用オブジェクトを使用してバリアント関数を呼び出す Fortran コードと、OpenMP および SYCL で実装されたバリアント関数で構成されています。

Fortran コードの最初の部分は、基本ルーチン vnxc とバリアントルーチン vnxc_gpu のインターフェイスを含むモジュールを定義します。vnxc は、DISPATCH ディレクティブ内で呼び出されるときに、MATCH 節を含む DECLARE VARIANT ディレクティブで置換ルーチン vnxc_gpu を宣言します。また、このディレクティブは、APPEND_ARGS 節を使用して、DISPATCH ディレクティブで指定された相互運用オブジェクトをバリアントルーチン vnxc_gpu に渡します。

Fortran のメインプログラムは、

- ホスト上のデータ配列 v1 を初期化します。
- データ配列 v1 をデバイスにマップするターゲットデータ領域を作成します。
- SYCL キューアクセスを持つ相互運用オブジェクト iop1 を作成します。

そして、

- INTEROP 節で iop1 を指定した DISPATCH ディレクティブを使用してルーチン vnxc を呼び出します。
- iop1 は、バリアントルーチン vnxc_gpu を呼び出す際に追加の引数として渡されます。

この例の SYCL 部分は、Fortran プログラムによって言語間で呼び出される可能性がある関数を実装します。

- Fortran プログラムは置換ルーチン vnxc_gpu を呼び出すため、ルーチン vnxc は呼び出されるべきではないことを示す print 文とともに定義されています。
- ルーチン vnxc_gpu は、ルーチンの最後の引数として渡された相互運用オブジェクトから取り出された SYCL キューを使用して単純なベクトルスカラー乗算を実行します。このルーチンは SYCL キューにアクセスし、デバイスメモリー上にすでに存在するデータ v1 を計算する SYCL コードを呼び出します。

```
$ cat interop_clause-fortran_interop.f90

module subs
  interface
    subroutine vnxc_gpu(c, v1, n, iop1)  !! variant function
      use iso_c_binding
      integer, intent(in)  :: c, n
      integer, intent(out) :: v1(10)
      type(c_ptr), intent(in):: iop1
    end subroutine vnxc_gpu
    subroutine vnxc(c, v1, n)  !! base function
      import vnxc_gpu          ! Need to add this statement
      integer, intent(in)  :: c, n
      integer, intent(out) :: v1(10)
      !$omp declare variant(vnxc:vnxc_gpu) &
      !$omp& match(construct={dispatch},device={arch(gen)})  &
      !$omp& append_args(interop(targetsync))
    end subroutine vnxc
  end interface
end module subs

program interop_clause
  use subs
  use omp_lib
  integer v1(10)
  integer i, n, d, c
```

```
  integer (kind=omp_interop_kind) :: iop1
  c = 2 ; n = 10 ; do i = 1, n ; v1(i) = i ; enddo
  d = omp_get_default_device()
  !$omp target data map(tofrom: v1(1:10)) use_device_addr(v1)
  !$omp interop init(prefer_type(omp_ifr_sycl), targetsync:iop1) device(d)
  !$omp dispatch device(d) interop(iop1)
  call vnxc(c, v1, n)
  !$omp interop destroy(iop1)
  !$omp end target data
  print *, "v1(1) = ", v1(1), " (2), v1(10) = ", v1(10), " (20)"
end program interop_clause

$ cat interop_clause-sycl_kernel.cpp

#include <omp.h>
#include <stdio.h>
#include <sycl/sycl.hpp>
#define EXTERN_C extern "C"
EXTERN_C void vnxc_(int *c, int *v1, int *n) {
  printf("ERROR: Base function foo should not be called\n");
}
EXTERN_C void vnxc_gpu_(int *c, int *v1, int *n, omp_interop_t obj) {
  int c_val = *c;
  int n_val = *n;
  if (omp_ifr_sycl != omp_get_interop_int(obj, omp_ipr_fr_id, nullptr)) {
    printf("ERROR: Failed to create interop with SYCL queue access\n");
    return;
  }
  auto *q = static_cast<sycl::queue *>(
      omp_get_interop_ptr(obj, omp_ipr_targetsync, nullptr));
  printf("Compute on device\n");
  q->parallel_for(n_val, [=](auto i) { v1[i] = c_val * v1[i]; });
  q->wait();
}
```

この例をコンパイルするには、次のコマンドを使用します。

```
icpx -qopenmp -fopenmp-targets=spir64 -fsycl -c interop_clause-sycl_kernel.cpp

ifx -qopenmp -fopenmp-targets=spir64 -fsycl interop_clause-fortran_interop.f90
interop_clause-sycl_kernel.o
```

生成された実行ファイルを実行します。

```
LIBOMPTARGET_DEBUG=2 OMP_TARGET_OFFLOAD=MANDATORY ./a.out 2>null
```

6. 基本言語の並列処理の自動オフロードサポート

ヘテロジニアス・コンピューティングにおいて C/C++ および Fortran の言語仕様が進化を続ける中、同時実行性、並列処理、メモリー階層など、高速コンピューティングの主要な動作と概念を表現する新しい言語機能が追加されています。

メモリー階層を表現する追加の言語機能も開発中です。現在、C/C++ の std::par や Fortran の DO CONCURRENT など、同時実行性/並列処理を表現するいくつかの主要な言語機能はすでに利用可能です。

std::par と DO CONCURRENT は、同様の概念を表しています。ここでは簡潔にするため、DO CONCURRENT に注目します。

以下は、DO CONCURRENT を使用して入れ子のループをデバイスに自動オフロードする例です。デバイスカーネルの生成は、実装とベンダーに依存します。例えば、NVIDIA CUDA コンパイラーは、独自の PTX 命令組込み関数を使用して自動オフロードカーネルを生成します。インテルは、オープン標準の OpenMP 実装を使用して自動オフロードカーネルを生成します。

```
program doconcurrent_autooffload
    implicit none
    integer :: i, j
    real(8) :: fn
    real(8) , allocatable :: F(:,:), A(:)
    integer :: bo=200, bi=50
    !integer, parameter :: bo=200, bi=50
    allocate(F(bo,bi))
    allocate(A(bo))
    F=0.0001 ; A=0.0
    !$omp target enter data map(to: F, A)
    do concurrent (i=1:bo)
        fn = 0.0
        !$omp parallel loop
        do concurrent(j=1:bi) reduce(+:fn)
            fn = fn + F(i,j)
        enddo
        A(i) = fn
        !$omp parallel loop
        do concurrent(j=1:bi)
            A(i) = A(i) + F(i,j)
        enddo
    enddo
    !$omp target exit data map(from: F, A)
    print *, "sum(A)=",sum(A)
end program doconcurrent_autooffload
```

次のコンパイラー・オプションを使用してデバイスカーネルを生成します。

```
ifx -g -O3 -xHost -fp-model precise -fiopenmp -fopenmp-targets=spir64
-fopenmp-target-do-concurrent -fopenmp-do-concurrent-maptype-modifier=present
-qopt-report=3 doconcurrent_autooffload.f90
```

最適化レポートファイルから分かるように、配列データはデバイスのグローバルメモリーにあると想定され、自動オフロードカーネルはホストデバイスのデータ移動を実行しません。

次のように設定します。

```
LIBOMPTARGET_DEBUG=1 LIBOMPTARGET_INFO=7
```

実行時に libomptarget ランタイム・デバッグ・ユーティリティーは、自動オフロードカーネルの実行ポリシーを示すランタイム情報を出力します。

OpenMP による並列コンピューティングのさらなる向上

OpenMP (英語) は、高速コンピューティング向けの堅牢なマルチベンダー、マルチプラットフォーム、マルチベース言語プログラミング・フレームワークであり続けています。新しい OpenMP 5.x と OpenMP 6.0 には、プログラマーがアプリケーションのヘテロジニアス並列処理を GPU 上でより効率的に表現できるようにする多くの新機能が用意されています。OpenMP のアプローチはオープンで標準ベースであるため、ベンダーは基盤となるハードウェア向けに最適化されたパフォーマンスを備えた独自の実装を提供できます。同時に、ハードウェアの詳細は透過的であり、ランタイムを通じてプログラマーや研究者に公開できます。アプリケーション開発者は、これらの高レベルの OpenMP プログラミングの概念と手法を実装するだけで、GPU ハードウェアの潜在能力を最大限に発揮する、移植性に優れたハイパフォーマンスのコードを作成できます。

OpenMP ユーザー・コミュニティーは非常に活発であり、成長を続けています。インテルは、このような取り組みの最前線に立ち、自社の OpenMP ランタイム・ライブラリー、インテル® Fortran コンパイラー、およびインテル® oneAPI DPC++/C++ コンパイラーにこれらの機能を統合できることを嬉しく思っています。インテルはオープン・スタンダードの精神を尊重し、OpenCL、SYCL、OpenMP などのアクセラレーテッド・コンピューティング・フレームワークの多くのオープンソース・イニシアチブと、LLVM や中間 SPIR-V 抽象化レイヤーなどのコンパイラー・イニシアチブに貢献しています。

コンパイラーのダウンロード

インテル® Fortran コンパイラーとインテル® oneAPI DPC++/C++ コンパイラーは、こちらからダウンロードできます。

インテル® HPC ツールキットやインテル® oneAPI ベース・ツールキットの一部として入手することもできます。これらのツールキットには、基本ツール、ライブラリー、解析ツール、デバッグツール、コード移行ツールが含まれています。

LLVM コンパイラー・プロジェクトへの貢献は、GitHub (英語) を参照してください。

関連情報

- Fortran と OpenMP でヘテロジニアス・プログラミングの課題を解決
- The Parallel Universe
- OpenMP ターゲットオフロード [1:06:07] (英語)
- OpenMP サポート: インテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス (英語)
- OpenMP サポート: インテル® oneAPI DPC++/C++ コンパイラー・デベロッパー・ガイドおよびリファレンス (英語)
- OpenMP 仕様 (英語)

製品および性能に関する情報

[1] 性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。