OpenACC for GPU: an introduction

Olga Abramkina, Rémy Dubois, Thibaut Véry

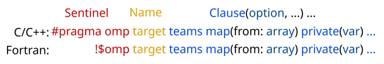
Jun 02, 2023

CONTENTS

Directives

OpenMP since specification 4.5 includes support for offloading to accelerators like GPUs. It uses directives to do so (just like for CPU).

A directive has the following structure:



```
         Sentinel     Name        Clause(option, ...) ...
C/C++:  #pragma omp target teams map(from: array) private(var) ...
Fortran:        !$omp target teams map(from: array) private(var) ...
```

If we break it down, we have these elements:

- The sentinel is special instruction for the compiler. It tells it that what follows has to be interpreted as OpenACC
- The directive is the action to do. In the example, *target* is the way to open a region that will be offloaded to the GPU
- The clauses are "options" of the directive. In the example we want to copy some data on the GPU.
- The clause arguments give more details for the clause. In the example, we give the name of the variables to be copied

Compiling with NVIDIA compiler

To enable OpenMP GPU offloading you need to activate the compilation options `-mp=gpu -gpu=<gpu,opts>`. For example to compile for NVIDIA V100:

```
nvfortran -mp=gpu -gpu=cc70 -o test test.f90
```

GPU offloading

With OpenMP the offloading is realized with the `omp target` directive. By itself, the directive will only offload the computation and do not activate parallelism. It is similar to the `acc serial` compute construct in OpenACC since only one GPU thread is running.

With OpenMP the developer has to activate manually the parallelism.

Here is an example on how to create a GPU kernel:

```
!$omp target
...
!$omp end target
```

Now that we run on the GPU we have to create the threads.

Thread creation on the GPU

Teams

OpenMP `target teams` directive creates several groups of threads that will be able to work in parallel.

With OpenACC it would correspond to the `gang` level of parallelism.

```
!$omp target teams
...
!$omp end target
```

By default the teams will work in replicated mode meaning that they will perform exactly the same things. If you want to share the iterations of a loop between the threads of the teams you have to use the `teams distribute` directive.

```
!$omp target
    !$omp teams distribute
    do i=0, sys_size
        ...
    enddo
!$omp end target
```

This will split the iterations of the loop among the teams. Each team will have a contiguous set of iterations.

It starting to be interesting but we do not yet take advantage of the full power of the GPU.

More threads with `omp parallel`

With the `omp parallel` directive inside a `omp teams` region we create the threads that will be used inside the team.

```
!$omp target teams distribute parallel
do i=1, sys_size
    ...
enddo
```

In this case the threads generated inside the teams will work in replicated mode. If we want to further split the work among those threads we have to add the `omp do` (Fortran) or `omp for` (C/C++) directive.

```
!$omp target teams distribute parallel do
do i=1, sys_size
    ...
enddo
```

With OpenACC it would correspond to the `worker` level of parallelism.

Let's vectorize with `omp simd`

The last level of parallelism we can leverage with OpenMP is the SIMD vectorization. It is done with the `omp simd` directive:

```
!$omp target teams distribute parallel do simd
do i=1, sys_size
    ...
enddo
```

Note for NVIDIA compilers

The `omp simd` construct is not supported for GPU. Currently, the `parallel` directive creates the threads that should be created with `simd`. Since the directive is just ignored, we recommend that you write it for portability reasons.

`collapse` clause

The `collapse` clause enables to merge all the iterations of several associated loops into a single large iteration loop. The number of loops that will be merged is indicated as an integer argument to this clause and should be greater than 1.

```fortran
!$omp target teams distribute parallel do simd collapse(3)
do k = 1, nz
    do j = 1, ny
        do i = 1, nx
            ...
        enddo
    enddo
enddo
```

Up to now, we will recommend you to use the collapse clause as much as you can with OpenMP target in order to achieve good performance.

Example

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/basic_offloading.f90
program basic_offloading
    use iso_fortran_env, only : real64
    implicit none

    integer                                      :: i, sys_size
    real(kind=real64), allocatable, dimension(:) :: array

    sys_size = 100000000

    allocate(array(sys_size))

    !$omp target teams distribute parallel do simd
    do i=1, sys_size
        array(i) = real(i)
    enddo

    print *, "array(42) = ", array(42)
end program basic_offloading
```

Reductions

Reductions should be performed when a memory location is updated by several threads concurrently, and usually prior to its previous value.

This can be performed by using the `reduction` clause of the target construct. This clause will create a private copy of the variables and initialize them as a function of the requested reduction operation. Once you reach the end of the kernel, the original variable will be updated with a combination of all the private copies.

The syntax is:

```
!$omp target parallel do reduction(operation:variable_list)
    ...
```

The available operations are:

- +, -
- _
- &, |, ^, &&, ||

Limitation

The reductions are now only supported for the 2 following combined constructs:

- `omp target parallel for`
- `omp target teams distribute parallel for`

Data management

Implicit behavior

If not specified in a `data map` structure, variables will be mapped implicitly at the entry of one kernel with a default action depending on the type of the variable.

Scalars will be map as `firstprivate`, i.e. every thread will have its own private copy that will be initialized with the value that the scalar have on the CPU before the kernel.

Arrays will be shared in memory between threads and are implicitly mapped as if you specified `map(tofrom:)`.

Pointers will be private by default.

You can see the effect of this implicit behavior with the example below:

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/Implicit_behavior.f90
program Implicit_behavior
    use iso_fortran_env, only : INT32, REAL64
    implicit none

    real   (kind=REAL64), dimension(:)  , allocatable :: Array
    integer(kind=INT32 )                              :: nx, i, scalar

    nx = 10
    allocate(Array(nx))

    scalar = 1000
    !$omp target teams distribute parallel do simd
    do i = 1, nx
```

(continues on next page)

(continued from previous page)

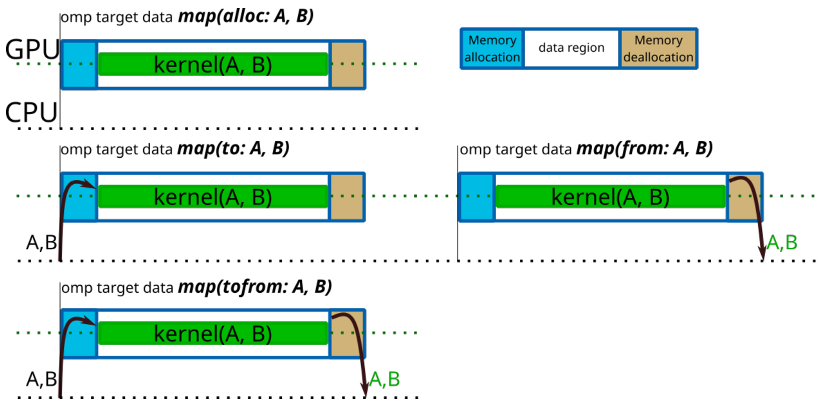
```
        Array(i) = scalar + i
    enddo

    print *, Array

    scalar = -1000
    !$omp target teams distribute parallel do simd
    do i = 1, nx
        Array(i) = scalar + i
    enddo

    print *, Array

    deallocate(Array)
end program Implicit_behavior
```

Relying only on the implicit behavior can lead to performance degradation as data transfers are performed back and forth at every kernels. This should be avoid by using data regions.

You can define a specific action to perform at the entry and/or the exit of a kernel for a variable or a set of variable with the `map` clause of the `target` construct.

The available options are:

- `alloc` to create the memory space of the variables without prior data transfer.
- `to` to create the memory space of the variables and transfer the values from CPU to GPU at the entry of the kernel.
- `from` to create the memory space of the variables and transfer the values from GPU to CPU at the exit of the kernel.
- `tofrom` to create the memory space of the variables and transfer the values from CPU to GPU at the entry of the kernel, then from GPU to CPU at the exit.


omp target data map(alloc: A, B)
GPU
CPU
kernel(A, B)
Memory allocation
data region
Memory deallocation
omp target data map(to: A, B)
kernel(A, B)
A,B
omp target data map(from: A, B)
kernel(A, B)
A,B
omp target data map(tofrom: A, B)
kernel(A, B)
A,B
A,B

The syntax is:

```
!$omp target map(to:variable1, variable2)
    ...
!$omp end target
```

It is also possible to modify the status of the variable manually with the `private` and `firstprivate` clauses of the `target` construct or by setting a default mapping that we will see later.

```
!$omp target private(variable1,variable2) firstprivate(variable3)
    ...
    ! variable1 and variable2 will have independent memory allocations for each␣
↪threads
    ! variable2 will have independent memory allocations for each threads and will be␣
↪initialized with the CPU value
!$omp end target
```

Structured data region

To run the kernels on GPU, the data should be allocated on the device and eventually the original values should be transfered from the CPU to the GPU. You will also have to retrieve some of the data back from the GPU to the CPU in order to store your results. This can be perfomed withing the same program unit by using the `target data` construct.

If you don't use data regions, implicit copies of the variables will be performed at each entry and exit of every kernels. This implies transfers trough the PCIe that could be avoided and thus non-optimal performances.

This construct map the variable to the device, but only for the extent of the region. The `map` clause enables you to decide which action will be performed on the gpu. These actions could be `alloc`, `to`, `from`, `tofrom`.

You can retrieve the values that were stored on the GPU with `from` and `tofrom` clauses

You can inform the GPU of the original CPU values with the clauses `to` and `tofrom`.

If you use the `alloc` or `from` clause, the initial value on the device is undetermined.

The syntax is:

```
real :: A(nx,ny), B(nx,ny)

!$omp target data map(tofrom:A,B)
    ...
!$omp end target data
```

Persistent data (`enter data` / `exit data`)

If you want to allocate the memory of some variables on the device at a given point of your program but it is not possible to free the memory within the same scope of the program, you can then use the `enter data` and `exit data` constructs.

`enter data` will enable you to allocate or allocate and initialize the variables on the GPU with the `map(alloc:variable_list)` and `map(to:variable_list)` clauses respectively.

`exit data` will enable you to free the memory from the device, resp. free the memory after retrieving the data, with the `map(delete:variable_list)`, resp. `map(from:variable_list)`.

These 2 constructs are not tied to each other, such as one `enter data` construct mapping several variables can lead to several `exit data` constructs in different portions of the code as long as 2 `exit data` are not refering to the same variable in this example.

The syntax is:

```
subroutine some_function_somewhere()
    real :: A(nx,ny), B(nx,ny)

    !$omp target enter data map(to:A)
    !$omp target enter data map(alloc:B)
    ...
end subroutine some_function_somewhere

subroutine some_function_elsewhere_or_maybe_the_same_as_before()
    ...
    !$omp target exit data map(delete:A,B)
end subroutine some_function_elsewhere_or_maybe_the_same_as_before
```

Manual data tranfers

When you want to update the values of a given variable, or a set of variables, either on the GPU or on the CPU, you can use the `target update` construct in order to avoid doing it by closing a data structure.

The `to` clause will update the GPU.

The `from` clause will update the CPU.

```
!$omp target update to(variable1,variable2)
```

`defaultmap` clause

You can modify the default mapping for the data transfer upon kernels or data structures with the `defaultmap` clause of the `target` and `target data` constructs.

The new implicit behavior can be specified as `alloc`, `to`, `from`, `tofrom`, `default`, `none`, `firstprivate` or `present` and should be applied to a variable category. Variable categories are:

- scalar
- aggregate (cornresponding to arrays and structures in C/C++ and to derived types in Fortran)
- allocatable (only for Fortran arrays that are dynamically allocated)
- pointers

If you specify the implicit behavior as `none`, you should then map explicitly all variables.

```
real, dimension(:), allocatable :: A
real                            :: B
!$omp target data defaultmap(firstprivate:scalar) defaultmap(tofrom:allocatable)
   ...
!$omp end target data
```

Modular programming

Functions that are call inside a kernel should be executed on the accelerator. You should use the `declare target` construt to inform the compiler that it should produce such an executable. Syntax should be:

```
subroutine my_routine(...)
!$omp declare target
        ...
end subroutine my_routine
```

If the function and the line from which the function is called are not within the same program unit, you should add a named `declare target` construct within the program unit containing the call.

```
subroutine another_routine
!$omp declare target(my_routine)

!$omp target teams
call my_routine()
!$omp end target teams

end subroutine another_routine
```

Exercise

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/Modular_programming_mean_value_exercise.f90
module calcul
    use iso_fortran_env, only : INT32, REAL64
    contains
        subroutine rand_init(array,n)
            real    (kind=REAL64), dimension(1,n), intent(inout) :: array
            integer(kind=INT32 ), intent(in)                     :: n
            real    (kind=REAL64)                                :: rand_val
            integer(kind=INT32)                                  :: i

            call srand(12345900)
            do i = 1, n
               call random_number(rand_val)
               array(1,i) = 2.0_real64*(rand_val-0.5_real64)
            enddo
        end subroutine rand_init

        subroutine iterate(array, array_size, cell_size)
            real    (kind=REAL64), dimension(1:array_size,1), intent(inout) :: array
            integer(kind=INT32 ), intent(in)                               :: array_
 ↪size, cell_size
            real    (kind=REAL64)                                          :: local_
 ↪mean
            integer(kind=INT32 )                                           :: i


            do i = cell_size/2, array_size-cell_size/2
                local_mean = mean_value(array(i+1-cell_size/2:i+cell_size/2,1), cell_
 ↪size)
                if (local_mean .lt. 0.0_real64) then
                    array(i,1) = array(i,1) + 0.1
```

(continues on next page)

(continued from previous page)

```
                else
                    array(i,1) = array(i,1) - 0.1
                endif
            enddo
        end subroutine iterate

        function mean_value(t, n)
            real    (kind=REAL64), dimension(n,1), intent(inout) :: t
            integer(kind=INT32 ), intent(in)                     :: n
            real    (kind=REAL64)                                :: mean_value
            integer(kind=INT32 )                                 :: i
            mean_value = 0.0_real64

            do i = 1, n
                mean_value = mean_value + t(i,1)
            enddo
            mean_value = mean_value / dble(n)
        end function mean_value
end module calcul
program modular_programming
    use calcul
    implicit none

    real    (kind=REAL64), dimension(:,:), allocatable :: table
    real    (kind=REAL64), dimension(:)  , allocatable :: mean_values
    integer(kind=INT32 )                               :: nx, ny, cell_size, i

    nx = 1000000
    ny =    3000
    allocate(table(nx,ny), mean_values(ny))
    table(:,:) = 0.0_real64
    call rand_init(table(1,:),ny)
    cell_size = 32
    do i = 2, ny
        call iterate(table(:,i), nx, cell_size)
    enddo

    do i = 1, ny
        mean_values(i) = mean_value(table(:,i), nx)
    enddo

    do i = 1, 10
        write(0,"(a18,i5,a1,f20.8)") "Mean value of row ",i,"=",mean_values(i)
    enddo

    do i = ny-10, ny
        write(0,"(a18,i5,a1,f20.8)") "Mean value of row ",i,"=",mean_values(i)
    enddo

    deallocate(table, mean_values)
end program modular_programming
```

Solution

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/Modular_programming_mean_value_solution.f90
module calcul
    use iso_fortran_env, only : INT32, REAL64
    contains
        subroutine rand_init(array,n)
            real   (kind=REAL64), dimension(1,n), intent(inout) :: array
            integer(kind=INT32 ), intent(in)                    :: n
            real   (kind=REAL64)                                :: rand_val
            integer(kind=INT32)                                 :: i

            call srand(12345900)
            do i = 1, n
               call random_number(rand_val)
               array(1,i) = 2.0_real64*(rand_val-0.5_real64)
            enddo
        end subroutine rand_init

        subroutine iterate(array, array_size, cell_size)
        !$omp declare target
            real   (kind=REAL64), dimension(1:array_size,1), intent(inout) :: array
            integer(kind=INT32 ), intent(in)                               :: array_
↪size, cell_size
            real   (kind=REAL64)                                           :: local_
↪mean
            integer(kind=INT32 )                                           :: i


            do i = cell_size/2, array_size-cell_size/2
                local_mean = mean_value(array(i+1-cell_size/2:i+cell_size/2,1), cell_
↪size)
                if (local_mean .lt. 0.0_real64) then
                    array(i,1) = array(i,1) + 0.1
                else
                    array(i,1) = array(i,1) - 0.1
                endif
            enddo
        end subroutine iterate

        function mean_value(t, n)
        !$omp declare target
            real   (kind=REAL64), dimension(n,1), intent(inout) :: t
            integer(kind=INT32 ), intent(in)                    :: n
            real   (kind=REAL64)                                :: mean_value
            integer(kind=INT32 )                                :: i
            mean_value = 0.0_real64

            do i = 1, n
                mean_value = mean_value + t(i,1)
            enddo
            mean_value = mean_value / dble(n)
        end function mean_value
end module calcul
program modular_programming
    use calcul
    implicit none
```

(continues on next page)

(continued from previous page)

```
    real    (kind=REAL64), dimension(:,:), allocatable :: table
    real    (kind=REAL64), dimension(:)  , allocatable :: mean_values
    integer(kind=INT32 )                              :: nx, ny, cell_size, i

    nx = 1000000
    ny =    3000
    allocate(table(nx,ny), mean_values(ny))
    table(:,:) = 0.0_real64
    call rand_init(table(1,:),ny)
    !$omp target enter data map(to:table)
    cell_size = 32
    !$omp target teams distribute parallel do simd
    do i = 2, ny
        call iterate(table(:,i), nx, cell_size)
    enddo

    !$omp target teams distribute parallel do simd map(from:mean_values)
    do i = 1, ny
        mean_values(i) = mean_value(table(:,i), nx)
    enddo

    do i = 1, 10
        write(0,"(a18,i5,a1,f20.8)") "Mean value of row ",i,"=",mean_values(i)
    enddo

    do i = ny-10, ny
        write(0,"(a18,i5,a1,f20.8)") "Mean value of row ",i,"=",mean_values(i)
    enddo

    !$omp target exit data map(delete:table)
    deallocate(table, mean_values)
end program modular_programming
```

Using multiple GPUs with OpenMP

If you have multiple accelerators available, you can select the one on which you run the kernels with the `device` clause of the `target` construct. It includes both `target data` constructs and `target teams/parallel` constructs.

You should give an integer that refers to the gpu number (starting from 0) to the `device` clause, such as :

```
    call mpi_comm_rank(MPI_COMM_WORLD,rank,code)
    num_gpus = omp_get_num_devices()
    my_gpu   = mod(my_rank,num_gpus)
    !$omp target data map(...) device(my_gpu)
        ...
    !$omp end target data
```

Exercise

In this exercise, you should bring on the gpu the MPI version of the generation of the Mandelbrot set on the gpu with OpenMP and by using multiple devices.

```
%%idrrun  --cliopts "2000 1000" -m 4 -g 4 --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/mandelbrot_mpi_exercise.f90
program mandelbrot_mpi
    use MPI
    implicit none
    real, parameter               :: min_re = -2.0, max_re = 1.0
    real, parameter               :: min_im = -1.0, max_im = 1.0
    integer                       :: first, last, width, height
    integer                       :: num_elements
    real                          :: step_w, step_h
    integer                       :: numarg, i, length, j, first_elem,last_elem
    integer                       :: rest_eucli,local_height
    integer                       :: rank, nb_procs, code
    character(len=:), allocatable :: arg1, arg2
    integer (kind=1), allocatable :: picture(:)
    real                          :: x, y

    numarg = command_argument_count()
    if (numarg .ne. 2) then
        write(0,*) "Error, you should provide 2 arguments of integer kind : width and␣
↪length"
        stop
    endif
    call get_command_argument(1,LENGTH=length)
    allocate(character(len=length) :: arg1)
    call get_command_argument(1,VALUE=arg1)
    read(arg1,'(i10)') width
    call get_command_argument(2,LENGTH=length)
    allocate(character(len=length) :: arg2)
    call get_command_argument(2,VALUE=arg2)
    read(arg2,'(i10)') height
    step_w = 1.0 / real(width)
    step_h = 1.0 / real(height)

    call mpi_init(code)
    call mpi_comm_rank(MPI_COMM_WORLD,rank,code)
    call mpi_comm_size(MPI_COMM_WORLD,nb_procs,code)

    local_height = height / nb_procs
    first = 0
    last  = local_height
    rest_eucli = mod(height,nb_procs)

    if ((rank .eq. 0) .and. (rank .lt. rest_eucli)) last = last + 1

    if (rank .gt. 0) then
        do i = 1, rank
            first = first + local_height
            last  = last  + local_height
                if (rank .lt. rest_eucli) then
                    first = first + 1
                    last  = last  + 1
                endif
        enddo
    endif

    if (rank .lt. rest_eucli) local_height = local_height + 1
```

(continues on next page)

(continued from previous page)

```
    num_elements = local_height * width

    write(unit=*,fmt="(a9,i3,a18,i8,a3,i8,a5,i10,a9)") "I am rank",rank, &
    " and my range is [",first," ,",last,"[ ie ",num_elements," elements"

    allocate(picture(first*width:last*width))

    do i=first,last-1
        do j=0,width-1
            x =  min_re + j * step_w * (max_re - min_re)
            y =  min_im + i * step_h * (max_im - min_im)
            picture(i*width+j) = mandelbrot_iterations(x,y)
        enddo
    enddo

    call output()
    deallocate(picture)

    call mpi_finalize(code)

    contains
        subroutine output
            integer                          :: fh
            integer(kind=MPI_OFFSET_KIND)    :: woffset

            woffset=first*width
            call MPI_File_open(MPI_COMM_WORLD,"mandel.gray",MPI_MODE_WRONLY+MPI_MODE_
↪CREATE,MPI_INFO_NULL,fh,code)
            call MPI_File_write_at(fh,woffset,picture,num_elements,MPI_INTEGER1,MPI_
↪STATUS_IGNORE,code);
            call MPI_File_close(fh,code)
        end subroutine output
        integer(kind=1) function mandelbrot_iterations(x,y)
            integer, parameter               :: max_iter = 127
            real, intent(in)                 :: x,y
            real                             :: z1,z2,z1_old,z2_old

            z1 = 0.0
            z2 = 0.0
            mandelbrot_iterations = 0
            do while (((z1*z1+z2*z2) .le. 4) .and. (mandelbrot_iterations .lt. max_
↪iter))
                z1_old = z1
                z2_old = z2
                z1 = z1_old*z1_old - z2_old*z2_old  + x
                z2 = 2.0*z1_old*z2_old + y
                mandelbrot_iterations = mandelbrot_iterations + 1
            enddo
        end function mandelbrot_iterations
end program mandelbrot_mpi
```

```python
from idrcomp import show_gray
show_gray("mandel.gray", 2000, 1000)
```

Solution

```
%%idrrun  --cliopts "2000 1000" -m 4 -g 4 --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/mandelbrot_mpi_solution.f90
program mandelbrot_mpi
    use MPI
    implicit none
    real, parameter              :: min_re = -2.0, max_re = 1.0
    real, parameter              :: min_im = -1.0, max_im = 1.0
    integer                      :: first, last, width, height
    integer                      :: num_elements
    real                         :: step_w, step_h
    integer                      :: numarg, i, length, j, first_elem,last_elem
    integer                      :: rest_eucli,local_height
    integer                      :: rank, nb_procs, code
    character(len=:), allocatable :: arg1, arg2
    integer (kind=1), allocatable :: picture(:)
    real                         :: x, y

    numarg = command_argument_count()
    if (numarg .ne. 2) then
        write(0,*) "Error, you should provide 2 arguments of integer kind : width and
 ↪length"
        stop
    endif
    call get_command_argument(1,LENGTH=length)
    allocate(character(len=length) :: arg1)
    call get_command_argument(1,VALUE=arg1)
    read(arg1,'(i10)') width
    call get_command_argument(2,LENGTH=length)
    allocate(character(len=length) :: arg2)
    call get_command_argument(2,VALUE=arg2)
    read(arg2,'(i10)') height
    step_w = 1.0 / real(width)
    step_h = 1.0 / real(height)

    call mpi_init(code)
    call mpi_comm_rank(MPI_COMM_WORLD,rank,code)
    call mpi_comm_size(MPI_COMM_WORLD,nb_procs,code)

    local_height = height / nb_procs
    first = 0
    last  = local_height
    rest_eucli = mod(height,nb_procs)

    if ((rank .eq. 0) .and. (rank .lt. rest_eucli)) last = last + 1

    if (rank .gt. 0) then
        do i = 1, rank
            first = first + local_height
            last  = last  + local_height
                if (rank .lt. rest_eucli) then
                    first = first + 1
                    last  = last  + 1
                endif
        enddo
    endif

    if (rank .lt. rest_eucli) local_height = local_height + 1
```

(continues on next page)

(continued from previous page)

```
    num_elements = local_height * width

    write(unit=*,fmt="(a9,i3,a18,i8,a3,i8,a5,i10,a9)") "I am rank",rank, &
    " and my range is [",first," ,",last,"[ ie ",num_elements," elements"

    allocate(picture(first*width:last*width))
    !$omp target data map(tofrom:picture) device(rank)
    !$omp target teams distribute parallel do simd collapse(2) device(rank)
    do i=first,last-1
        do j=0,width-1
            x =  min_re + j * step_w * (max_re - min_re)
            y =  min_im + i * step_h * (max_im - min_im)
            picture(i*width+j) = mandelbrot_iterations(x,y)
        enddo
    enddo
    !$omp end target data
    call output()
    deallocate(picture)

    call mpi_finalize(code)

    contains
        subroutine output
            integer                          :: fh
            integer(kind=MPI_OFFSET_KIND)    :: woffset

            woffset=first*width
            call MPI_File_open(MPI_COMM_WORLD,"mandel.gray",MPI_MODE_WRONLY+MPI_MODE_
↪CREATE,MPI_INFO_NULL,fh,code)
            call MPI_File_write_at(fh,woffset,picture,num_elements,MPI_INTEGER1,MPI_
↪STATUS_IGNORE,code);
            call MPI_File_close(fh,code)
        end subroutine output
        integer(kind=1) function mandelbrot_iterations(x,y)
            !$omp declare target
            integer, parameter               :: max_iter = 127
            real, intent(in)                 :: x,y
            real                             :: z1,z2,z1_old,z2_old

            z1 = 0.0
            z2 = 0.0
            mandelbrot_iterations = 0
            do while (((z1*z1+z2*z2) .le. 4) .and. (mandelbrot_iterations .lt. max_
↪iter))
                z1_old = z1
                z2_old = z2
                z1 = z1_old*z1_old - z2_old*z2_old  + x
                z2 = 2.0*z1_old*z2_old + y
                mandelbrot_iterations = mandelbrot_iterations + 1
            enddo
        end function mandelbrot_iterations
end program mandelbrot_mpi
```

```
from idrcomp import show_gray
show_gray("mandel.gray", 2000, 1000)
```

Using NV-link with OpenMP target

You can specify to the accelerator the pointer to a given data structure already present on the device that should be used with `use_device_addr` clause of the `data` construct.

Exercise

As an exercise, you can complete the following MPI code that measures the bandwidth between the GPUs:

1. Add directives to create the buffers on the GPU
2. Measure the effective bandwidth between GPUs by adding the directives necessary to transfer data from one GPU to another one in the following cases:

- Not using NVLink
- Using NVLink

We have a bug for MPI in the notebooks and you need to save the file before running the next cell. It is a good way to pratice manual building! Please add the correct extension for the language you are running.

```
%%writefile MultiGPU_mpi_exercise.<extension>
!!  examples_openmp/Fortran/MultiGPU_mpi_exercise.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program MultiGPU_exercice
    use ISO_FORTRAN_ENV, only : INT32, REAL64
    use mpi
    use openacc
    implicit none
    real    (kind=REAL64), dimension(:), allocatable :: send_buffer, receive_buffer
    real    (kind=REAL64)                            :: start, finish , data_volume
    integer(kind=INT32 ), parameter                  :: system_size = 2e8/8
    integer                                          :: comm_size, my_rank, code, reps,
 ↪ i, j, k
    integer                                          :: num_gpus, my_gpu
    integer(kind=acc_device_kind)                    :: device_type
    integer, dimension(MPI_STATUS_SIZE)              :: mpi_stat

    ! Useful for OpenMPI and GPU DIRECT
    call initialisation_openacc()

    ! MPI stuff
    reps = 5
    data_volume = dble(reps*system_size)*8*1024_real64**(-3.0)

    call MPI_Init(code)
    call MPI_Comm_size(MPI_COMM_WORLD, comm_size, code)
    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, code)
    allocate(send_buffer(system_size), receive_buffer(system_size))

    ! OpenACC stuff
    #ifdef _OPENACC
    device_type = acc_get_device_type()
    num_gpus = acc_get_num_devices(device_type)
    my_gpu   = mod(my_rank,num_gpus)
```

(continues on next page)

(continued from previous page)

```
    call acc_set_device_num(my_gpu, device_type)
    #endif

    do j = 0, comm_size - 1
        do i = 0, comm_size - 1
            if ( (my_rank .eq. j) .and. (j .ne. i) ) then
                start = MPI_Wtime()
                do k = 1, reps
                    call MPI_Send(send_buffer,system_size, MPI_DOUBLE, i, 0, MPI_COMM_
↪WORLD, code)
                enddo
            endif
            if ( (my_rank .eq. i) .and. (i .ne. j) ) then
                do k = 1, reps
                    call MPI_Recv(receive_buffer, system_size, MPI_DOUBLE, j, 0, MPI_
↪COMM_WORLD, mpi_stat, code)
                enddo
            endif
            if ( (my_rank .eq. j) .and. (j .ne. i) ) then
                finish = MPI_Wtime()
                write(0,"(a11,i2,a2,i2,a2,f20.8,a5)") "bandwidth ",j,"->",i,": ",data_
↪volume/(finish-start)," GB/s"
            endif
        enddo
    enddo

    deallocate(send_buffer, receive_buffer)

    call MPI_Finalize(code)

    contains
        #ifdef _OPENACC
        subroutine initialisation_openacc
            use openacc
            implicit none
            type accel_info
                integer :: current_devices
                integer :: total_devices
            end type accel_info

            type(accel_info) :: info
            character(len=6) :: local_rank_env
            integer          :: local_rank_env_status, local_rank
        ! Initialisation of OpenACC
            !$acc init

        ! Recovery of the local rank of the process via the environment variable
        ! set by Slurm, as MPI_Comm_rank cannot be used here because this routine
        ! is used BEFORE the initialisation of MPI
            call get_environment_variable(name="SLURM_LOCALID", value=local_rank_env,␣
↪status=local_rank_env_status)
            info%total_devices = acc_get_num_devices(acc_get_device_type())
            if (local_rank_env_status == 0) then
                read(local_rank_env, *) local_rank
                ! Definition of the GPU to be used via OpenACC
                call acc_set_device_num(local_rank, acc_get_device_type())
```

(continues on next page)

(continued from previous page)

```
                info%current_devices = local_rank
            else
                print *, "Error : impossible to determine the local rank of the␣
↪process"
                stop 1
            endif
        end subroutine initialisation_openacc
        #endif

end program MultiGPU_exercice
```

```
module load nvidia-compilers/21.9 cuda/11.2 openmpi/4.0.5-cuda
 # Add compiling here
mpi....
srun -A for@gpu --gpus-per-node=2 --ntasks-per-node=4 --cpus-per-task=5 ./a.out
```

Solution

We have a bug for MPI in the notebooks and you need to save the file before running the next cell. It is a good way to pratice manual building! Please add the correct extension for the language you are running.

```
%%writefile MultiGPU_mpi_exercise.<extension>
!!  examples_openmp/Fortran/MultiGPU_mpi_solution.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program MultiGPU_solution
    use ISO_FORTRAN_ENV, only : INT32, REAL64
    use mpi
    use openacc
    implicit none
    real   (kind=REAL64), dimension(:), allocatable :: send_buffer, receive_buffer
    real   (kind=REAL64)                            :: start, finish , data_volume
    integer(kind=INT32 ), parameter                 :: system_size = 2e8/8
    integer                                         :: comm_size, my_rank, code, reps,
↪ i, j, k
    integer                                         :: num_gpus, my_gpu
    integer(kind=acc_device_kind)                   :: device_type
    integer, dimension(MPI_STATUS_SIZE)             :: mpi_stat

    ! Useful for OpenMPI and GPU DIRECT
    call initialisation_openacc()

    ! MPI stuff
    reps = 5
    data_volume = dble(reps*system_size)*8*1024_real64**(-3.0)

    call MPI_Init(code)
    call MPI_Comm_size(MPI_COMM_WORLD, comm_size, code)
    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, code)
    allocate(send_buffer(system_size), receive_buffer(system_size))
    !$omp target enter data map(alloc: send_buffer(1:system_size), receive_
↪buffer(1:system_size))
```

(continues on next page)

(continued from previous page)

```
    ! OpenMP target stuff
    #ifdef _OPENACC
    device_type = acc_get_device_type()
    num_gpus = acc_get_num_devices(device_type)
    my_gpu   = mod(my_rank,num_gpus)
    call acc_set_device_num(my_gpu, device_type)
    #endif

    do j = 0, comm_size - 1
        do i = 0, comm_size - 1
            if ( (my_rank .eq. j) .and. (j .ne. i) ) then
                start = MPI_Wtime()
                !$omp target data use_device_ptr(send_buffer)
                do k = 1, reps
                    call MPI_Send(send_buffer,system_size, MPI_DOUBLE, i, 0, MPI_COMM_
↪WORLD, code)
                enddo
                !$omp end target data
            endif
            if ( (my_rank .eq. i) .and. (i .ne. j) ) then
                !$omp target data use_device_ptr(send_buffer)
                do k = 1, reps
                    call MPI_Recv(receive_buffer, system_size, MPI_DOUBLE, j, 0, MPI_
↪COMM_WORLD, mpi_stat, code)
                enddo
                !$omp end target data
            endif
            if ( (my_rank .eq. j) .and. (j .ne. i) ) then
                finish = MPI_Wtime()
                write(0,"(a11,i2,a2,i2,a2,f20.8,a5)") "bandwidth ",j,"->",i,": ",data_
↪volume/(finish-start)," GB/s"
            endif
        enddo
    enddo
    !$omp target exit data map(delete: send_buffer, receive_buffer)
    deallocate(send_buffer, receive_buffer)

    call MPI_Finalize(code)

    contains
        #ifdef _OPENACC
        subroutine initialisation_openacc
            use openacc
            implicit none
            type accel_info
                integer :: current_devices
                integer :: total_devices
            end type accel_info

            type(accel_info) :: info
            character(len=6) :: local_rank_env
            integer          :: local_rank_env_status, local_rank
        ! Initialisation of OpenACC
            !$acc init

        ! Recovery of the local rank of the process via the environment variable
```

(continues on next page)

(continued from previous page)

```
        ! set by Slurm, as MPI_Comm_rank cannot be used here because this routine
        ! is used BEFORE the initialisation of MPI
            call get_environment_variable(name="SLURM_LOCALID", value=local_rank_env,␣
↪status=local_rank_env_status)
            info%total_devices = acc_get_num_devices(acc_get_device_type())
            if (local_rank_env_status == 0) then
                read(local_rank_env, *) local_rank
                ! Definition of the GPU to be used via OpenACC
                call acc_set_device_num(local_rank, acc_get_device_type())
                info%current_devices = local_rank
            else
                print *, "Error : impossible to determine the local rank of the␣
↪process"
                stop 1
            endif
        end subroutine initialisation_openacc
        #endif

end program MultiGPU_solution
```

```
%%bash
module load nvidia-compilers/21.9 cuda/11.2 openmpi/4.0.5-cuda
 # Add compiling here
mpi....
srun -A for@gpu --gpus-per-node=4 --ntasks-per-node=8 --cpus-per-task=5 ./a.out
```

Asynchronism

Concurrent executions within the same stream

An implicit barrier is set at the end of each `target` construct to ensure that the parent task (the task on the host) can not move on until the target task has ended. You can disable this implicit behavior and submit several kernels on the GPU by explicitly adding the `nowait` clause to the target construct.

In order to avoid race conditions that could arise from the lack of barrier between kernels, it is possible to specify a scheduling of the kernels based on a dependency mechanism. To do so, you should use the `depend` clause.

Exercise

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/async_async_exercise.f90
program prod_mat
    use iso_fortran_env, only : INT32, REAL64
    implicit none
    integer (kind=INT32)                :: rank=5000
    real    (kind=REAL64), allocatable :: A(:,:), B(:,:), C(:,:)
    integer (kind=INT32)                :: i, j, k
    integer (kind=INT32)                :: streamA, streamB, streamC
```

(continues on next page)

(continued from previous page)

```
    streamA = 1
    streamB = 2
    streamC = 3

    call create_mat(A, rank, streamA)
    call create_mat(B, rank, streamB)
    call create_mat(C, rank, streamC)

    call init_mat(A, rank, 3.0_real64 , streamA)
    call init_mat(B, rank, 14.0_real64, streamB)
    call init_mat(C, rank, 0.0_real64 , streamC)

    do j=1, rank
        do k=1, rank
            do i=1, rank
                C(i,j) = C(i,j) + A(i,k)*B(k,j)
            enddo
        enddo
    enddo
    print *, "Check that this is close to 42.0:", C(12,12)
    deallocate(A, B, C)
    contains
        subroutine create_mat(mat, rank, stream)
            real   (kind=REAL64), intent(inout), allocatable   :: mat(:,:)
            integer(kind=INT32 ), intent(in)                   :: rank, stream
            allocate(mat(rank,rank))
        end subroutine create_mat

        subroutine init_mat(mat, rank, diag, stream)
            real    (kind=REAL64), intent(inout)   :: mat(:,:)
            real    (kind=REAL64), intent(in)      :: diag
            integer (kind=INT32 ), intent(in)      :: rank, stream
            integer (kind=INT32 )                  :: i, j

            do j=1, rank
                do i=1, rank
                   mat(i,j) = 0.0_real64
                enddo
            enddo

            do j=1, rank
                mat(j,j) = diag
            enddo
        end subroutine init_mat
end program prod_mat
```

Solution

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
!!  examples_openmp/Fortran/async_async_solution.f90
program prod_mat
    use iso_fortran_env, only : INT32, REAL64
    implicit none
```

(continues on next page)

(continued from previous page)

```
    integer (kind=INT32)                :: rank=5000
    real    (kind=REAL64), allocatable :: A(:,:), B(:,:), C(:,:)
    integer (kind=INT32)                :: i, j, k
    integer (kind=INT32)                :: streamA, streamB, streamC

    streamA = 1
    streamB = 2
    streamC = 3

    call create_mat(A, rank, streamA)
    call create_mat(B, rank, streamB)
    call create_mat(C, rank, streamC)

    call init_mat(A, rank, 3.0_real64 , streamA)
    call init_mat(B, rank, 14.0_real64, streamB)
    call init_mat(C, rank, 0.0_real64 , streamC)

    !$omp target teams distribute parallel do simd collapse(3)
    do j=1, rank
        do k=1, rank
            do i=1, rank
                C(i,j) = C(i,j) + A(i,k)*B(k,j)
            enddo
        enddo
    enddo
    !$omp target exit data map(delete:A,B)
    !$omp target exit data map(from:C)
    print *, "Check that this is close to 42.0:", C(12,12)
    deallocate(A, B, C)
    contains
        subroutine create_mat(mat, rank, stream)
            real   (kind=REAL64), intent(inout), allocatable   :: mat(:,:)
            integer(kind=INT32 ), intent(in)                   :: rank, stream
            allocate(mat(rank,rank))
            !$omp target enter data map(alloc:mat) nowait depend(out:mat)
        end subroutine create_mat

        subroutine init_mat(mat, rank, diag, stream)
            real    (kind=REAL64), intent(inout)   :: mat(:,:)
            real    (kind=REAL64), intent(in)      :: diag
            integer (kind=INT32 ), intent(in)      :: rank, stream
            integer (kind=INT32 )                  :: i, j

            !$omp target teams distribute parallel do simd collapse(2) nowait␣
 ↪depend(inout:mat)
            do j=1, rank
                do i=1, rank
                   mat(i,j) = 0.0_real64
                enddo
            enddo

            !$omp target teams distribute parallel do simd nowait depend(in:mat)
            do j=1, rank
                mat(j,j) = diag
            enddo
        end subroutine init_mat
end program prod_mat
```

CHAPTER

ONE

OPENMP CHEAT SHEET

1.1 Directive syntax

```
              Sentinel     Name          Clause(option, ...) ...
  C/C++: #pragma omp target teams map(from: array) private(var) ...
Fortran:             !$omp target teams map(from: array) private(var) ...
```

If we break it down, we have those elements:

- The sentinel is a special instruction for the compiler. It tells him that what follows has to be interpreted as OpenMP directives
- The directive is the action to do. In the example, *target* is the way to open a parallel region that will be offloaded to the GPU
- The clauses are “options” of the directive. In the example we want to copy some data from the GPU.
- The clause arguments give more details for the clause. In the example, we give the name of the variables to be copied

1.2 Creating kernels

The way to open kernels on the GPU is to use the `omp target` directive with directive to create threads.

1.2.1 Creating threads

The threads creation is the job of the developper in OpenMP. The standard defines 3 levels of parallelism:

- `omp teams`: Several groups of threads are created but only the master thread is active.
- `omp parallel`: The other threads of the team are activated.
- `omp simd`: SIMD threads are activated

1.2.2 Work Sharing

Creating threads is not enough to have the full power of the GPU. You have to share work among threads:

- `omp teams distribute`: distribute work among teams
- `omp parallel for/do`: distribute work inside a team

1.2.3 *omp target* Clauses

Clause	effect
private(vars, …)	Make *vars* private at *team* level
firstprivate(vars, …)	Make *vars* private at *team* level and copy the value vars had on the host before
device(dev_num)	Set the device on which to run the kernel

Other clauses might be available. Check the specification and the compiler documentation for full list.

1.2.4 *omp teams* Clauses

Clause	effect
num_teams(#teams)	Set the number of teams for the target region
thread_limit(#threads)	Set the maximum number of threads inside a team
private(vars, …)	Make *vars* private at *team* level
firstprivate(vars, …)	Make *vars* private at *team* level and copy the value vars had on the host before
reduction(op:vars, …)	Perform a reduction of the variables *vars* with operation *op*

Other clauses might be available. Check the specification and the compiler documentation for full list.

1.2.5 *omp parallel* Clauses

Clause	effect
private(vars, …)	Make *vars* private at *parallel* level
firstprivate(vars, …)	Make *vars* private at *parallel* level and copy the value vars had on the host before
reduction(op:vars, …)	Perform a reduction of the variables *vars* with operation *op*

Other clauses might be available. Check the specification and the compiler documentation for full list.

1.2.6 *omp simd* Clauses

Clause	effect
private(vars, …)	Make *vars* private at *simd* level
firstprivate(vars, …)	Make *vars* private at *simd* level and copy the value vars had on the host before
reduction(op:vars, …)	Perform a reduction of the variables *vars* with operation *op*
simdlen(vector_size)	Set the length of the vector

Other clauses might be available. Check the specification and the compiler documentation for full list.

1.3 Combined constructs for loops

It is possible to combine the

1.4 Managing data

1.4.1 Data regions

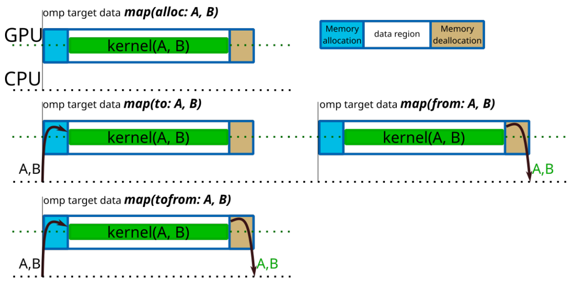
Region	Directive
Program lifetime	`omp target enter data` & `omp target exit data`
Structured	`omp target data`
Kernels	`omp target map(...)`

1.4.2 Data clauses

To choose the right data clause you need to answer the following questions:

- Does the kernel need the values computed on the host (CPU) beforehand? (Before)
- Are the values computed inside the kernel needed on the host (CPU) afterhand? (After)

	Needed after	Not needed after
Needed Before	map(tofrom:var1, …)	map(to:var2, …)
Not needed before	map(from:var3, …)	map(alloc:var4, …)


omp target data map(alloc: A, B)
GPU
CPU
kernel(A, B)
Memory allocation
data region
Memory deallocation
omp target data map(to: A, B)
kernel(A, B)
A,B
omp target data map(from: A, B)
kernel(A, B)
A,B
omp target data map(tofrom: A, B)
kernel(A, B)
A,B
A,B

1.4.3 Updating data already present on the GPU

It is not possible to update data present on the GPU with the data clauses on a data region. To do so you need to use `omp target update`

`omp target update` Clauses

- To update CPU with data computed on GPU: `omp target update from(data, ...)`
- To update GPU with data computer on CPU: `omp target update to(data, ...)`

1.5 GPU routines

A routine called from a kernel needs to be inside a `declare target` region.

```
subroutine my_routine(...)
!$omp declare target
        ...
end subroutine my_routine
```

1.6 Using data on the GPU with GPU aware libraries

To get a pointer to the device memory for a variable you have to use:

- `omp data use_device_ptr(var, ...)` for pointers
- `omp data use_device_addr(var, ...)` for allocatables