OpenMP for GPU: an introduction

Olga Abramkina, Rémy Dubois, Thibaut Véry

Jun 02, 2023

CONTENTS

Structure of the archive

- C: Notebooks in C language
- Fortran: Notebooks in Fortran
- pictures: All figures used in the notebooks
- examples: The source code for the exercises
  - C
  - Fortran
- utils:
  - idrcomp: the source code for the utility to run %%irdrrun cells
  - config: configuration file
  - start_jupyter_acc.py: start the jupyter server

On Jean Zay

You have to execute the following lines to be able to run the notebooks

```
cd $WORK/OpenACC_GPU
module load python/3.7.6
conda activate cours_openacc
 # You have to start once ipython before starting
 # you can exit ipython just after
ipython

./utils/start_jupyter_acc.py
```

A password is printed and will be useful later.

Once it is done you can start a browser and go to https://idrvprox.idris.fr.

- The first identification is with the login and the password you were given.
- The second identification is with the password generated with `./utils/start_jupyter_acc.py`.

List of notebooks

- Get started: Get started with OpenMP target directives for GPU
- Cheat sheet: Summary of the main OpenMP target directives

Notebooks

The training course uses Jupyter notebooks as a support.

We wrote the content so that you should be able to do the training course alone in the case we do not have time to see everything together.

The notebooks are divided into several kinds of cells:

- Markdown cells: those are the text cells. The ones we have written are protected against edition. If you want to take notes inside the notebook you can create new cells.
- Python code cells: A few cells are present with python code inside. You have to execute them to have the intended behavior of subsequent cells
- idrrun code cells: The cells in which the exercises/examples/solutions are written. They are editable directly and when you execute it the code inside is compiled and a job is submitted.

Note about idrrun cells

All idrrun cells with code inside have a comment with the name of the source file associated. You can find all source files inside the folders:

- examples/C
- examples/Fortran

If you do not wish to use the notebooks to edit the exercises, you can always edit the source files directly. Then you will need to proceed manually with the compilation (a makefile is provided) and job submission.

Configuration

Some configuration might be needed in order to have the best experience possible with the training course.

You should have a README.md file shipped with the content, which explains all files that need to be edited.

CHAPTER

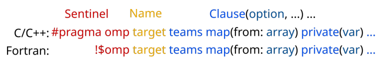
ONE

OPENMP OFFLOADING DIRECTIVES

1.1 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:



```
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

1.2 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:

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

1.3 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:

```
#pragma omp target
{
...
}
```

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

1.4 Thread creation on the GPU

1.4.1 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.

```
#pragma omp target teams
{
...
}
```

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.

```
#pragma omp target
{
    #pragma omp teams distribute
    for (int i=0; i<size; ++u)
    {
        ...
    }
}
```

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.

1.4.2 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.

```
#pragma omp target teams distribute parallel
for (int i=0; i<sys_size; ++i)
{
    ...
}
```

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.

```
#pragma omp target teams distribute parallel do
for (int i=0; i<sys_size; ++i)
{
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

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

1.4.3 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:

```
#pragma omp target teams distribute parallel do simd
for (int i=0; i<sys_size; ++i)
{
    ...
}
```

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.

1.4.4 `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.

```
#pragma omp target teams dsistribute parallel for simd collapse(3)
for (int i=0;i<nx;i++)
    for (int i=0;i<nx;i++)
        for (int i=0;i<nx;i++)
           ...
```

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.

1.4.5 Example

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/basic_offloading.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int size = 100000000;
    double* array = (double*) malloc(size*sizeof(double));

// We need to explicitly manage transfers in C with NVIDIA compilers
```

(continues on next page)

(continued from previous page)

```
// otherwise we get a runtime error (as of nvhpc 21.9)
#pragma omp target teams distribute parallel for simd map(from:array[0:size])
    for (int i=0; i<size; ++i)
        array[i] = (double) i;

    printf("array[42] = %f\n", array[42]);
    free(array);
    return 0;
}
```

1.5 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:

```
#pragma omp target parallel for reduction(operation:variable_list)
{
    ...
}
```

The available operations are:

- +, -
- –
- &, |, ^, &&, ||

1.5.1 Limitation

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

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

1.6 Data management

1.6.1 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:

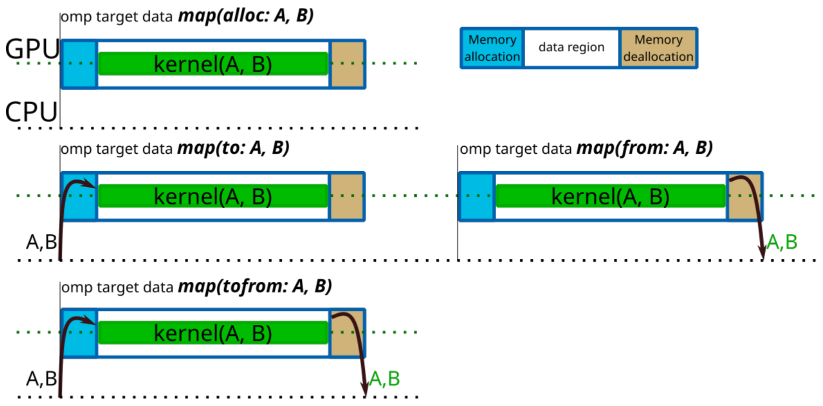
```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/Implicit_behavior.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int size = 10;
    double* array = (double*) malloc(size*sizeof(double));
    double  scalar;

    scalar = 1000.0;
#pragma omp target teams distribute parallel for
    for (int i=0; i<size; ++i)
        array[i] = (double)i + scalar;

    for (int i=0; i<size; ++i)
        printf("%lf\n",array[i]);

    scalar = -1000.1;

#pragma omp target teams distribute parallel for
    for (int i=0; i<size; ++i)
        array[i] = (double)i + scalar;

    for (int i=0; i<size; ++i)
        printf("%lf\n",array[i]);

    free(array);
    return 0;
}
```

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)
A,B
omp target data map(from: A, B)
omp target data map(tofrom: A, B)

The syntax is:

```
#pragma omp target map(from:variable1,variable2)
{
   ...
}
```

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.

```
#pragma 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
}
```

1.6.2 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:

```
    double* A = (double*) malloc(nx*ny*sizeof(double));
    double* B = (double*) malloc(nx*ny*sizeof(double));
    #pragma omp target data map(tofrom:A[0:nx*ny], B[0:nx*ny])
    {
```

(continues on next page)

(continued from previous page)

```
        ...
    }
```

1.6.3 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:

```
void some_function_somewhere(void)
{
    double* A = (double*) malloc(nx*ny*sizeof(double));
    double* B = (double*) malloc(nx*ny*sizeof(double));
    #pragma omp target enter data map(to:A[0:nx*ny])
    #pragma omp target enter data map(alloc:B[0:nx*ny])
        ...
}

void some_function_elsewhere_or_maybe_the_same_as_before(void)
{
    ...
    #pragma omp target exit data map(delete:A, B)
}
```

1.6.4 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.

```
#pragma omp target update to(picture[0:num_elements])
```

1.6.5 `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 (corrensponding 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.

```
int A[N];
int B;

#pragma omp target defaultmape(firstprivate:scalar) defaultmap(tofrom:aggregate)
{
    ...
}
```

1.7 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:

```
#pragma omp declare target
void my_funtion(void)
{
        ...
}
#pragma omp end declare target
```

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.

```
#pragma omp declare target
void my_function(void){
    ...
}
#pragma omp end declare target


int main(void){
    #pragma declare target(my_function)
    ...
    #pragma target teams distribute
    {
    a = my_function();
    }
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

1.7.1 Exercise

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/Modular_programming_mean_value_exercise.c
#include <stdio.h>
#include <stdlib.h>
double mean_value(double* array, size_t array_size){
    double sum = 0.0;
    for(size_t i=0; i<array_size; ++i)
        sum += array[i];
    return sum/array_size;
}

void rand_init(double* array, size_t array_size)
{
     srand((unsigned) 12345900);
     for (size_t i=0; i<array_size; ++i)
         array[i] = 2.*((double)rand()/RAND_MAX -0.5);
}

void iterate(double* array, size_t array_size, size_t cell_size)
{
    double local_mean;
    for (size_t i = cell_size/2; i< array_size-cell_size/2; ++i)
    {
        local_mean = mean_value(&array[i-cell_size/2], cell_size);
        if (local_mean < 0.)
            array[i] += 0.1;
        else if (local_mean > 0.)
            array[i] -= 0.1;
    }
}

int main(void){
    size_t num_cols = 500000;
    size_t num_rows = 3000;

    double* table = (double*) malloc(num_rows*num_cols*sizeof(double));
    double* mean_values = (double*) malloc(num_rows*sizeof(double));
    // We initialize the first row with random values between -1 and 1
    rand_init(table, num_cols);

    for (size_t i=1; i<num_rows; ++i)
       iterate(&table[i*num_cols], num_cols, 32);

    for (size_t i=0; i<num_rows; ++i)
    {
        mean_values[i] = mean_value(&(table[i*num_cols]), num_cols);
    }

    for (size_t i=0; i<10; ++i)
```

(continues on next page)

(continued from previous page)

```
        printf("Mean value of row %6d=%10.5f\n", i, table[i]);
    printf("...\n");
    for (size_t i=num_rows-10; i<num_rows; ++i)
        printf("Mean value of row %6d=%10.5f\n", i, table[i]);
    return 0;
}
```

1.7.2 Solution

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/Modular_programming_mean_value_solution.c
#include <stdio.h>
#include <stdlib.h>
#pragma omp declare target
double mean_value(double* array, size_t array_size){
    double sum = 0.0;
    for(size_t i=0; i<array_size; ++i)
        sum += array[i];
    return sum/array_size;
}
#pragma omp end declare target

void rand_init(double* array, size_t array_size)
{
     srand((unsigned) 12345900);
     for (size_t i=0; i<array_size; ++i)
         array[i] = 2.*((double)rand()/RAND_MAX -0.5);
}

void iterate(double* array, size_t array_size, size_t cell_size)
{
    double local_mean;
    #pragma omp target teams distribute parallel for simd
    for (size_t i = cell_size/2; i< array_size-cell_size/2; ++i)
    {
        local_mean = mean_value(&array[i-cell_size/2], cell_size);
        if (local_mean < 0.)
            array[i] += 0.1;
        else if (local_mean > 0.)
            array[i] -= 0.1;
    }
}

int main(void){
    size_t num_cols = 500000;
    size_t num_rows = 3000;

    double* table = (double*) malloc(num_rows*num_cols*sizeof(double));
    double* mean_values = (double*) malloc(num_rows*sizeof(double));
    // We initialize the first row with random values between -1 and 1
    rand_init(table, num_cols);
    #pragma omp target enter data map(to:table[0:num_rows*num_cols])

    for (size_t i=1; i<num_rows; ++i)
```

(continues on next page)

(continued from previous page)

```
        iterate(&table[i*num_cols], num_cols, 32);

    #pragma omp target teams distribute parallel for simd map(from:mean_values[0:num_
↪rows])
    for (size_t i=0; i<num_rows; ++i)
    {
        mean_values[i] = mean_value(&(table[i*num_cols]), num_cols);
    }

    #pragma omp target exit data map(delete:table)
    for (size_t i=0; i<10; ++i)
        printf("Mean value of row %6d=%10.5f\n", i, table[i]);
    printf("...\n");
    for (size_t i=num_rows-10; i<num_rows; ++i)
        printf("Mean value of row %6d=%10.5f\n", i, table[i]);
    return 0;
}
```

1.8 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 :

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int num_gpus = omp_get_num_devices();
    int my_gpu = my_rank%num_gpus
    #pragma omp target data map(...) device(my_gpu)
    {
        ...
    }
```

1.8.1 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/C/mandelbrot_mpi_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <complex.h>
#include <mpi.h>
#include <omp.h>
void output(unsigned char* picture, unsigned int start, unsigned int num_elements)
{
   MPI_File     fh;
   MPI_Offset   woffset=start;
```

(continues on next page)

(continued from previous page)

```
   if (MPI_File_open(MPI_COMM_WORLD,"mandel.gray",MPI_MODE_WRONLY+MPI_MODE_CREATE,MPI_
↪INFO_NULL,&fh) != MPI_SUCCESS)
   {
        fprintf(stderr,"ERROR in creating output file\n");
        MPI_Abort(MPI_COMM_WORLD,1);
   }

   MPI_File_write_at(fh,woffset,picture,num_elements,MPI_UNSIGNED_CHAR,MPI_STATUS_
↪IGNORE);

   MPI_File_close(&fh);
}

unsigned char mandelbrot_iterations(const float complex c)
{
    unsigned char max_iter = 255;
    unsigned char n = 0;
    float complex z = 0.0 + 0.0 * I;
    while (abs(z*z) <= 2 && n < max_iter)
    {
        z = z*z + c;
        ++n;
    }
    return n;
}

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
    unsigned int width = (unsigned int) atoi(argv[1]);
    float step_w = 1./width;
    unsigned int height = (unsigned int) atoi(argv[2]);
    float step_h = 1./height;

    const float min_re = -2.;
    const float max_re = 1.;
    const float min_im = -1.;
    const float max_im = 1.;

    struct timespec end, start;
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    int i;
    int rank;
    int nb_procs;
    int total_devices;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);

    unsigned int local_height = height / nb_procs;
    unsigned int first = 0;
    unsigned int last = local_height;
    unsigned int rest_eucli = height % nb_procs;

    if ((rank==0) && (rank < rest_eucli))
```

(continues on next page)

(continued from previous page)

```
            ++last;

      for (i=1; i <= rank; ++i)
      {
        first += local_height;
        last  += local_height;
        if (rank < rest_eucli)
            {
                ++first;
                ++last;
            }
      }

      if (rank < rest_eucli)
          ++local_height;

      unsigned int num_elements = width*local_height;
      if (rank == 0) printf("Using MPI\n");
      total_devices = 0
      printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements. Runing on %d␣
   ↪GPUs.\n", rank, first, last, num_elements, total_devices);
      unsigned char* restrict picture = (unsigned char*) malloc(num_
   ↪elements*sizeof(unsigned char));
      for (unsigned int i=0; i<local_height; ++i)
          for (unsigned int j=0; j<width; ++j)
          {
              float complex c;
              c = min_re + j*step_w * (max_re - min_re) + \
                  I * (min_im +  ((i+first) * step_h) * (max_im - min_im));
              picture[width*i+j] = (unsigned char) (255-rank*(255/nb_procs)) -␣
   ↪mandelbrot_iterations(c);
          }
      output(picture, first*width, num_elements);
      MPI_Finalize();

      // Measure time
      clock_gettime(CLOCK_MONOTONIC_RAW, &end);
      unsigned long int delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec␣
   ↪- start.tv_nsec) / 1000;
      printf("The time to generate the mandelbrot picture was %lu us\n", delta_us);
      return EXIT_SUCCESS;
  }
```

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

1.9 Solution

```
%%idrrun  --cliopts "2000 1000" -m 4 -g 4 --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/mandelbrot_mpi_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <complex.h>
#include <mpi.h>
#include <omp.h>
void output(unsigned char* picture, unsigned int start, unsigned int num_elements)
{
   MPI_File     fh;
   MPI_Offset   woffset=start;

   if (MPI_File_open(MPI_COMM_WORLD,"mandel.gray",MPI_MODE_WRONLY+MPI_MODE_CREATE,MPI_
↪INFO_NULL,&fh) != MPI_SUCCESS)
   {
        fprintf(stderr,"ERROR in creating output file\n");
        MPI_Abort(MPI_COMM_WORLD,1);
   }

   MPI_File_write_at(fh,woffset,picture,num_elements,MPI_UNSIGNED_CHAR,MPI_STATUS_
↪IGNORE);

   MPI_File_close(&fh);
}

#pragma omp declare target
unsigned char mandelbrot_iterations(const float complex c)
{
    unsigned char max_iter = 255;
    unsigned char n = 0;
    float complex z = 0.0 + 0.0 * I;
    while (abs(z*z) <= 2 && n < max_iter)
    {
        z = z*z + c;
        ++n;
    }
    return n;
}
#pragma omp end declare target
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
    unsigned int width = (unsigned int) atoi(argv[1]);
    float step_w = 1./width;
    unsigned int height = (unsigned int) atoi(argv[2]);
    float step_h = 1./height;

    const float min_re = -2.;
    const float max_re = 1.;
    const float min_im = -1.;
    const float max_im = 1.;

    struct timespec end, start;
```

(continues on next page)

(continued from previous page)

```
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    int i;
    int rank;
    int nb_procs;
    int total_devices;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);

    unsigned int local_height = height / nb_procs;
    unsigned int first = 0;
    unsigned int last = local_height;
    unsigned int rest_eucli = height % nb_procs;

    if ((rank==0) && (rank < rest_eucli))
          ++last;

    for (i=1; i <= rank; ++i)
    {
      first += local_height;
      last  += local_height;
      if (rank < rest_eucli)
          {
              ++first;
              ++last;
          }
    }

    if (rank < rest_eucli)
        ++local_height;

    unsigned int num_elements = width*local_height;
    if (rank == 0) printf("Using MPI\n");
    total_devices = omp_get_num_devices();
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements. Runing on %d␣
↪GPUs.\n", rank, first, last, num_elements, total_devices);
    unsigned char* restrict picture = (unsigned char*) malloc(num_
↪elements*sizeof(unsigned char));
#pragma omp target data map(tofrom:picture[0:num_elements]) device(rank)
{
#pragma omp target teams distribute parallel for simd collapse(2) device(rank)
    for (unsigned int i=0; i<local_height; ++i)
        for (unsigned int j=0; j<width; ++j)
        {
            float complex c;
            c = min_re + j*step_w * (max_re - min_re) + \
                I * (min_im +  ((i+first) * step_h) * (max_im - min_im));
            picture[width*i+j] = (unsigned char) (255-rank*(255/nb_procs)) -␣
↪mandelbrot_iterations(c);
        }
}
    output(picture, first*width, num_elements);
    MPI_Finalize();

    // Measure time
```

(continues on next page)

(continued from previous page)

```
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
    unsigned long int delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec␣
↪- start.tv_nsec) / 1000;
    printf("The time to generate the mandelbrot picture was %lu us\n", delta_us);
    return EXIT_SUCCESS;
}
```

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

1.9.1 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.

1.9.2 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/C/MultiGPU_mpi_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <openacc.h>
#include <math.h>
#include "../examples/C/init_openacc.h"
int main(int argc, char** argv)
{
    initialisation_openacc();
    MPI_Init(&argc, &argv);
    fflush(stdout);
    double start;
    double end;

    int size = 2e8/8;

    double* send_buffer = (double*)malloc(size*sizeof(double));
    double* receive_buffer = (double*)malloc(size*sizeof(double));
    // MPI Stuff
    int my_rank;
```

(continues on next page)

(continued from previous page)

```
    int comm_size;
    int reps = 5;
    double data_volume = (double)reps*(double)size*sizeof(double)*pow(1024,-3.0);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Status status;

    // OpenACC Stuff
    acc_device_t device_type = acc_get_device_type();
    int num_gpus = acc_get_num_devices(device_type);
    int my_gpu = my_rank%num_gpus;
    acc_set_device_num(my_gpu, device_type);
    for (int i = 0; i<comm_size; ++i)
    {
        for (int j=0; j < comm_size; ++j)
        {
            if (my_rank == i && i != j)
            {
                start = MPI_Wtime();
                for (int k = 0 ; k < reps; ++k)
                    MPI_Ssend(send_buffer, size, MPI_DOUBLE, j, 0, MPI_COMM_WORLD);
            }
            if (my_rank == j && i != j)
            {
                for (int k = 0 ; k < reps; ++k)
                    MPI_Recv(receive_buffer, size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &
↪status);
            }
            if (my_rank == i && i != j)
            {
                end = MPI_Wtime();
                printf("bandwidth %d->%d: %10.5f GB/s\n", i, j, data_volume/(end-
↪start));
            }
        }
    }
    MPI_Finalize();
    return 0;
}
```

```
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/C/MultiGPU_mpi_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <openmp.h>
#include <math.h>
#include "../../examples/init_omp_target.h"

int main(int argc, char** argv)
{
    initialisation_openacc();
    MPI_Init(&argc, &argv);
    fflush(stdout);
    double start;
    double end;

    int size = 200000000/8;

    double* send_buffer = (double*)malloc(size*sizeof(double));
    double* receive_buffer = (double*)malloc(size*sizeof(double));
    #pragma omp targer enter data map(alloc: send_buffer[:size], receive_
↪buffer[:size])
    // MPI Stuff
    int my_rank;
    int comm_size;
    int reps = 5;
    double data_volume = (double)reps*(double)size*sizeof(double)*pow(1024,-3.0);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Status status;

    // OpenMP target Stuff
    int num_gpus = omp_get_num_devices();
    int my_gpu = my_rank%num_gpus;
    acc_set_device_num(my_gpu, device_type);
    for (int i = 0; i<comm_size; ++i)
    {
        for (int j=0; j < comm_size; ++j)
        {
            if (my_rank == i && i != j)
            {
                start = MPI_Wtime();
                #pragma omp target data use_device_ptr(send_buffer)
                {
                    for (int k = 0 ; k < reps; ++k)
                        MPI_Ssend(send_buffer, size, MPI_DOUBLE, j, 0, MPI_COMM_
↪WORLD);
                }
            }
            if (my_rank == j && i != j)
            {
```

(continues on next page)

(continued from previous page)

```
                #pragma omp target data use_device_ptr(receive_buffer)
                {
                    for (int k = 0 ; k < reps; ++k)
                        MPI_Recv(receive_buffer, size, MPI_DOUBLE, i, 0, MPI_COMM_
↪WORLD, &status);
                }
            }
            if (my_rank == i && i != j)
            {
                end = MPI_Wtime();
                printf("bandwidth %d->%d: %10.5f GB/s\n", i, j, data_volume/(end-
↪start));
            }
        }
    }
    #pragma omp targer exit data map(delete: send_buffer[:size], receive_
↪buffer[:size])
    MPI_Finalize();
    return 0;
}
```

```
%%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
```

1.10 Asynchronism

1.10.1 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.

1.10.2 Exercise

```
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/async_async_exercise.c
#include <stdio.h>
#include <stdlib.h>
double* create_mat(int dim, int stream)
{
    double* mat = (double*) malloc(dim*dim*sizeof(double));
    return mat;
}
```

(continues on next page)

(continued from previous page)

```c
void init_mat(double* mat, int dim, double diag, int stream)
{
    for (int i=0; i<dim; ++i)
        for (int j=0; j<dim; ++j)
        {
            mat[i*dim+j] = 0.;
        }
    for (int i=0; i<dim; ++i)
        mat[i*dim+i] = diag;
}

int main(void)
{
    int dim = 5000;

    double* restrict A = create_mat(dim, 1);
    double* restrict B = create_mat(dim, 2);
    double* restrict C = create_mat(dim, 3);

    init_mat(A, dim, 6.0, 1);
    init_mat(B, dim, 7.0, 2);
    init_mat(C, dim, 0.0, 3);

    for (int i=0; i<dim; ++i)
        for (int k=0; k<dim; ++k)
            for (int j=0; j<dim; ++j)
            {
                C[i*dim+j] += A[i*dim+k] * B[k*dim+j];
            }
    }
    printf("Check that value is equal to 42.: %f\n", C[0]);
    return 0;
}
```

1.10.3 Solution

```c
%%idrrun --options "-mp=gpu -gpu=cc70 -Minfo=all"
//  examples_openmp/C/async_async_solution.c
#include <stdio.h>
#include <stdlib.h>
double* create_mat(int dim, int stream)
{
    double* mat = (double*) malloc(dim*dim*sizeof(double));
    #pragma omp enter data map(alloc:mat[0:dim*dim]) nowait depend(out:mat)
    return mat;
}

void init_mat(double* mat, int dim, double diag, int stream)
{
    #pragma acc parallel loop present(mat[0:dim*dim]) async(stream)
    #pragma omp teams distribute parallel for simd collapse(2) nowait␣
 ↪depend(inout:mat)
    for (int i=0; i<dim; ++i)
        for (int j=0; j<dim; ++j)
```

(continues on next page)

(continued from previous page)

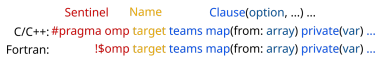
```
        {
            mat[i*dim+j] = 0.;
        }
    #pragma omp teams distribute parallel for simd nowait depend(in:mat)
    for (int i=0; i<dim; ++i)
        mat[i*dim+i] = diag;
}

int main(void)
{
    int dim = 5000;

    double* restrict A = create_mat(dim, 1);
    double* restrict B = create_mat(dim, 2);
    double* restrict C = create_mat(dim, 3);

    init_mat(A, dim, 6.0, 1);
    init_mat(B, dim, 7.0, 2);
    init_mat(C, dim, 0.0, 3);

    #pragma omp target teams distribut parallel for simd collaspe(3)
    for (int i=0; i<dim; ++i)
        for (int k=0; k<dim; ++k)
            for (int j=0; j<dim; ++j)
            {
                C[i*dim+j] += A[i*dim+k] * B[k*dim+j];
            }
    }
    #pragma omp target exit data map(delete:A,B)
    #pragma omp target exit data map(from:C[:dim*dim])
    printf("Check that value is equal to 42.: %f\n", C[0]);
    return 0;
}
```

CHAPTER

TWO

OPENMP CHEAT SHEET

2.1 Directive syntax



```
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

2.2 Creating kernels

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

2.2.1 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

2.2.2 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

2.2.3 *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.

2.2.4 *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.

2.2.5 *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.

2.2.6 *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.

2.3 Combined constructs for loops

It is possible to combine the

2.4 Managing data

2.4.1 Data regions

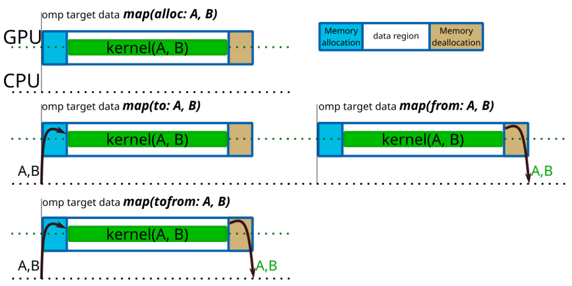
Region	Directive
Program lifetime	`omp target enter data` & `omp target exit data`
Structured	`omp target data`
Kernels	`omp target map(...)`

2.4.2 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
kernel(A, B)
CPU
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

2.4.3 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, ...)`

2.5 GPU routines

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

```
#pragma omp declare target
void my_funtion(void)
{
        ...
}
#pragma omp end declare target
```

2.6 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