OpenACC 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

Day 1

- Introduction: You will find here some information about:
  - Hardware
  - Short history of OpenACC and OpenMP for GPU
  - Programming model

The notebook is purely informational.

- Getting started: You should start here. The notebook presents quickly the main features you need to use OpenACC on the GPU.
- Manual Building: The training uses JupyterLab and you won't need to compile anything by hand. However it is important to know how to do it.
- Data Management: The main bottleneck when porting to GPU will surely be data transfers. Here we see the different way to deal efficiently with this issue.
- Hands-on Game of Life: Conway's Game of Life with OpenACC.

Day 2

- Other Compute Constructs: There are several ways to create kernels with OpenACC. We presented one during the first day (`acc parallel`) and here are the other (`acc kernels` and `acc serial`).
- Variable status: It is important to keep in mind the default variables statuts in the kernels. We also learn how to privatize variables to get correct algorithms.
- Kernels/Loop configuration: By default the compiler and the runtime set the parameters for the kernels and loops. Even though we do not advise to do it manually we will teach you how to do.
- GPU Routines: Functions and subroutines can be called inside kernels only if they were compiled for the GPU.
- The porting process requires that you can profile the code.
- Feeling a bit tight on one GPU? Try using several GPUs.
- MultiGPUs with Mandelbrot either MPI or *OpenMP*.

Day 3

- The GPU might be able to run several kernels at the same time with a feature called asynchronism
- If you need to make sure that only one thread reads/writes a variable at a time then atomic operations are for you
- Are you using C structures or Fortran derived type? Then manual deep copy will interest you.
- You can use CUDA libraries with your OpenACC project
- Your code is reusing data frequently, why not trying to improve data locality with a single clause tiles
- A small Lennard-Jones gas simulator kleineMole

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.

Part I

Day 1

CHAPTER

ONE

INTRODUCTION TO GPU PROGRAMMING WITH DIRECTIVES

1.1 What is a GPU?

Graphical Processing Units (GPU) have been designed to accelerate the processing of graphics and have boomed thanks to video games which require more and more computing power.

For this course we will use the terminology from NVIDIA.

GPUs have a large number of computing core really efficient to process large matrices. For example, the latest generation of NVIDIA GPU (Hopper 100 SXM5) have 132 processors, called Streaming Multiprocessors (SM) with different kinds of specialized cores:

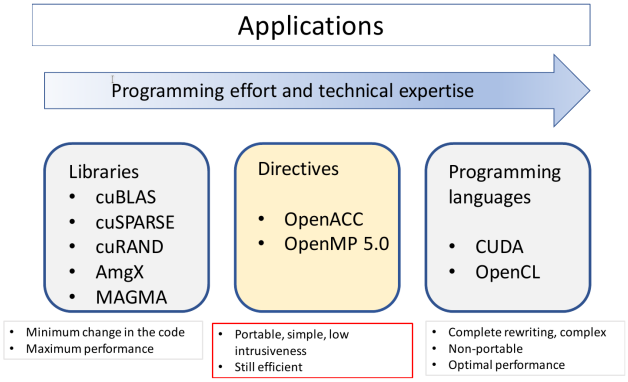
Core Type	Number per SM
FP32	128
FP64	64
INT32	64
TensorCore	4
Total	176

It means that you have roughly 19k cores on one GPU.

At max, CPUs can have a few 10s of cores (AMD Epyc Rome have 64 cores). The comparison with the number of cores on one CPU is not fully relevant since their architecture differs a lot.

This is the scheme for one streaming multiprocessor in an NVIDIA H100 GPU.

1.2 Programming models



You have the choice between several programming models to port your code to GPU:

- Low level programming language (CUDA, OpenCL)
- Programming models (Kokkos)
- GPU libraries (CUDA accelerated libraries, MAGMA, THRUST, AmgX)
- Directives languages (OpenACC, OpenMP target)

Most of the time they are interoperable and you can get the best of each world as long as you take enough time to learn everything :).

In this training course we focus on the directives languages.

1.2.1 Low level programming languages: CUDA, OpenCL

CUDA

Introduction of floating-point processing and programming capabilities on GPU cards at the turn of the century opened the door to general purpose GPU (GPGPU) programming. GPGPU was greatly democratized with the arrival of the CUDA programming language in 2007.

CUDA is a language close to C++ where you have to manage yourself everything that occurs on the GPU:

- Allocation of memory
- Data transfers
- Kernel (piece of code running on the GPU) execution

The kernel configuration has to be explicitly written in your code.

```
my_kernel<<<kernelconfig>>>(arguments);
```

All of this means that if you want to port your code on GPU with CUDA you have to write specialized portions of code. With this you have access to potentially the full processing power of the GPU but you have to learn a new language.

Since it is only available on NVIDIA GPUs you lack the portability to other platforms.

OpenCL

OpenCL have been available since 2009 and it was developed to write code that can run on several kind of architectures (CPU, GPU, FPGA, …).

OpenCL is supported by the major hardware companies so if you choose this option you can alleviate the portability issue. However, you still have to manage by hand everything happening on the GPU.

```
clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size, &
↪local_size, 0, NULL, NULL);
```

1.2.2 Using libraries

Let say that your code is spending a lot of time in only one type of computation (linear algebra, FFTs, etc). Then it is interesting to look for specialized libraries developed for this kind of computation:

- NVIDIA CUDA libraries: FFT, BLAS, Sparse algebra, …
- MAGMA: Dense linear algebra
- etc

The implementation cost is much lower than if you have to write your own kernels and you get (hopefully) very good performance.

1.2.3 Directives

In the general case where the libraries do not fulfill an important part of your code, you can choose to use OpenACC or OpenMP 4.5 and above with the target construct.

With this approach you annotate your code with directives considered as comments if you do not activate the compiler options to use them.

For OpenACC:

```
!$acc parallel loop
do i = 1, size
    ! Code to offload to GPU
enddo
```

For OpenMP target:

```
!omp target teams distribute parallel do
do i = 1, size
    // Code to offload to GPU
```

The implementation cost is much lower than the low level programming languages and usually you can get up to 95% of the performance you would get by writing your own specialized code.

Even though the modifications in your code will be lower than rewriting everything, you have to keep in mind that some changes might be necessary to have the best performance possible. Those changes can be in:

- the algorithms
- the data structures
- etc

1.3 OpenACC

The first version of the OpenACC specification was released in November 2011. It was created by:

- Cray
- NVIDIA
- PGI (now part of NVIDIA)
- CAPS

In November 2022 they released the 3.3 specification.

1.3.1 Compilers

Several compilers are available to produce OpenACC code.

- HPE Cray Programming environment (for HPE/Cray hardware)
- NVIDIA HPC SDK (formerly PGI)
- GCC 10
- AMD Sourcery CodeBench
- etc

You have to be careful since the maturity of each compiler and the specification they respect can change.

Disclaimer

The training course is based on version 2.7 of the specification.

Here we will mainly use the HPC compilers from NVIDIA available on their website which fully respects specification 2.7. You will be able to test the GCC compilers which support specification 2.6

1.4 OpenMP target

The first OpenMP specification which supports GPU offloading is 4.5 released in November 2015. It adds the `target` construct for this purpose.

The newest specification (november 2021) for OpenMP is 5.2.

1.4.1 Compilers

The list of compilers supporting OpenMP is available on the OpenMP website. You have to check if the `target` (or offloading) is supported.

The main compilers which support offloading to GPU are:

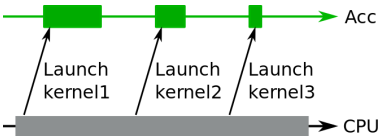
- IBM XL for C/C++ and Fortran
- GCC since version 7
- CLANG
- NVIDIA HPC SDK (formerly PGI)

- Cray Programming environment (for Cray hardware)

1.5 Host driven Language

OpenACC is a host driven programming language. It means that the host (usually a CPU) is in charge of launching everything happening on the device (usually a GPU) including:

- Executing kernels
- Memory allocations
- Data transfers



1.6 Levels of parallelism

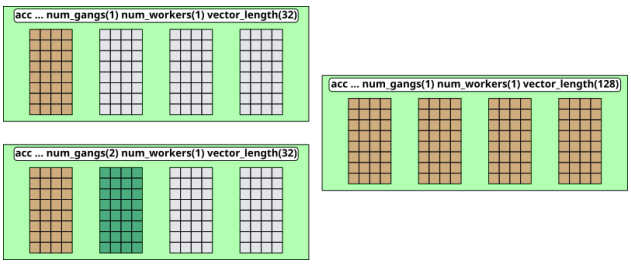
On the GPU you can have 4 different levels of parallelism that can be activated:

- Coarse grain: gang
- Fine grain : worker
- Vectorization : vector
- Sequential : seq

One Gang is made of several Workers which are vectors (with by default a size of one thread). You can increase the number of thread by activating the Vectorization.

Inside a kernel gangs have the same number of threads running. But it can be different from one kernel to another.

So the total number of threads used by a kernel is $(Number\_of\_Gangs) * (Number\_of\_Workers) * (Vector\_Length)$.



1.7 Important notes

- There is no way to synchronize threads between gangs.
- The compiler may decide to add synchronization within the threads in one gang.
- The threads of a worker work in SIMT mode. It means that all threads run the same instruction at the same time. For example on NVIDIA GPUS, groups of 32 threads are formed.
- Usually NVIDIA compilers set the number of workers to one.

1.7.1 Information about NVIDIA devices

The `nvaccelinfo` command gives interesting information about the devices available.

For example, if you run it on Jean Zay A100 partition.

```
$ nvaccelinfo

Device Number:                 7
Device Name:                   NVIDIA A100-SXM4-80GB
Device Revision Number:        8.0
Global Memory Size:            85051572224
Number of Multiprocessors:     108
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Clock Rate:                    1410 MHz
Concurrent Kernels:            Yes
Memory Clock Rate:             1593 MHz
L2 Cache Size:                 41943040 bytes
Max Threads Per SMP:           2048
Async Engines:                 3
Managed Memory:                Yes
Default Target:                cc80
...
```

CHAPTER

TWO

GET STARTED WITH OPENACC

What will you learn here?

1. Open a parallel region with `#pragma acc parallel`
2. Activate loop parallelism with `#pragma acc loop`
3. Open a structured data region with `#pragma acc data`
4. Compile a code with OpenACC support

2.1 OpenACC directives

If you have a CPU code and you want to get some parts on the GPU, you can add OpenACC directives to it.

A directive has the following structure:

```
            Sentinel     Name    Clause(option, ...) ...
C/C++: #pragma acc parallel copyin(array) private(var) ...
Fortran:       !$acc parallel copyin(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, *parallel* 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 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

Some directives need to be opened just before a code block.

```
!$acc parallel
  ! code block opened with `acc parallel` and closed with `acc end parallel`
!$acc end parallel
```

2.1.1 A short example

With this example you can get familiar with how to run code cells during this session. `%%idrrun` has to be present at the top of a code cell to compile and execute the code written inside the cell.

The content has to be a valid piece of code otherwise you will get errors. In Fortran, if you want to run the code, you need to define the `main` function:

```
program your_code
!
!
end program your_code
```

The example initializes an array of integers.

```
%%idrrun
!!  examples/Fortran/Get_started_init_array_exercise.f90
program array_initialisation
    use iso_fortran_env, only : INT32, REAL64
    implicit none

    integer(kind=INT32), parameter :: system_size = 100000
    integer(kind=INT32)            :: array(system_size)
    integer(kind=INT32)            :: i

    do i = 1, system_size
        array(i) = 2*i
    enddo

    write(0,"(a22,i3)") "value at position 21: ", array(21)
end program array_initialisation
```

Now we add the support of OpenACC with `-a` option of idrrun.

To offload the computation on the GPU you have to open a parallel region with the directive `acc parallel` and define a code block which is affected.

Modify the cell below to perform this action. No clause are needed here.

```
%%idrrun -a
!!  examples/Fortran/Get_started_init_array_exercise_acc.f90
program array_initialisation
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32), parameter :: system_size = 100000
    integer(kind=INT32)            :: array(system_size)
    integer(kind=INT32)            :: i

    ! Modifications from here
    do i = 1, system_size
        array(i) = 2*i
    enddo

    write(0,"(a22,i3)") "value at position 21: ", array(21)
end program array_initialisation
```

2.1.2 Solution

```
%%idrrun -a
!!  examples/Fortran/Get_started_init_array_solution_acc.f90
program array_initialisation
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32), parameter :: system_size = 100000
    integer(kind=INT32)            :: array(system_size)
    integer(kind=INT32)            :: i

    !$acc parallel
    do i = 1, system_size
        array(i) = 2*i
    enddo
    !$acc end parallel

    write(0,"(a22,i3)") "value at position 21: ", array(21)
end program array_initialisation
```

Which is equivalent to:

```
%%idrrun -a
!!  examples/Fortran/Get_started_init_array_solution_acc_2.f90
program array_initialisation
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32), parameter :: system_size = 100000
    integer(kind=INT32)            :: array(system_size)
    integer(kind=INT32)            :: i

    !$acc parallel

    !$acc loop
    do i = 1, system_size
        array(i) = 2*i
    enddo

    !$acc end parallel

    write(0,"(a22,i3)") "value at position 21: ", array(21)
end program array_initialisation
```

2.1.3 Let's analyze what happened.

The following steps are printed:

1. the compiler command to generate the executable
2. the output of the command (displayed on red background)
3. the command line to execute the code
4. the output/error of the execution

We activated the verbose mode for the NVIDIA compilers for information about optimizations and OpenACC (compiler option -Minfo=all) and **strongly recommend that you do the same in your developments**.

The compiler found in the `main` function a **kernel** (this is the name of code blocks offloaded to the GPU) and was able to generate code for GPU. The line refers to the directive `acc parallel` included in the code.

By default NVIDIA compilers (formerly PGI) make an analysis of the parallel region and try to find:

- loops that can be parallelized
- data transfers needed
- operations like reductions
- etc

It might result in unexpected behavior since we did not write explicitly the directives to perform those actions. Nevertheless, we decided to keep this feature on during the session since it is the default. This is the reason you can see that a directive `acc loop` (used to activate loop parallelism on the GPU) was added implicitly to our code and a data transfer with `copyout`.

2.2 Loops parallelism

Most of the parallelism in OpenACC (hence performance) comes from the loops in your code and especially from loops with **independent iterations**. Iterations are independent when the results do not depend on the order in which the iterations are done. Some differences due to non-associativity of operations in limited precision are usually OK. You just have to be aware of that problem and decide if it is critical.

Another condition is that the runtime needs to know the number of iterations. So keep incrementing integers!

2.2.1 Directive

The directive to parallelize loops is:

```
!$acc loop
```

2.2.2 Non independent loops

Here are some cases where the iterations are not independent:

- Infinite loops

```
do while(error > tolerance)
    !compute error
enddo
```

- Current iteration reads values computed by previous iterations

```
array(1) = 0
array(2) = 1
do i = 3, system_size
    array(i) = array(i-1) + array(i-2)
enddo
```

- Current iteration reads values that will be changed by subsequent iterations

```
do i = 1, system_size - 1
    array(i) = array(i+1) + 1
enddo
```

- Current iteration writes values that will be read by subsequent iterations

```
do i= 1, system_size - 1
    array(i)   = array(i) + 1
    array(i+1) = array(i) + 2
enddo
```

These kind of loops can be offloaded to the GPU but might not give correct results if not run in sequential mode. You can try to modify the algorithm to transform them into independent loop:

- Use temporary arrays
- Modify the order of the iterations
- etc

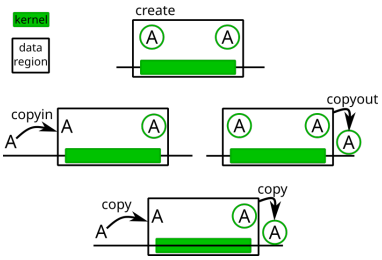
2.3 Managing data in compute regions

During the porting of your code the data on which you work in the *compute regions* might have to go back and forth between the host and the GPU. This is important to minimize the number of data transfers because of the cost of these operations.

For each *compute region* (i.e. `acc parallel` directive or *kernel*) a *data region* is created. OpenACC gives you several clauses to manage efficiently the transfers.

```
!$acc parallel copy(var1(first_index:last_index)) copyin(var2(first_index_i:last_
↪index_i,first_index_j:last_index_j), var3) copyout(var4, var5)
```

clause	effect when entering the region	effect when leaving the region
create	Allocate the memory needed on the GPU	Free the memory on the GPU
copyin	Allocate the memory and initialize the variable with the values it has on CPU	Free the memory on the GPU
copyout	Allocate the memory needed on the GPU	Copy the values from the GPU to the CPU then free the memory on the GPU
copy	Allocate the memory and initialize the variable with the values it has on CPU	Copy the values from the GPU to the CPU then free the memory on the GPU
present	Check if data is present: an error is raised if it is not the case	None



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

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

	Needed after	Not needed after
Needed Before	copy(var1, …)	copyin(var2, …)
Not needed before	copyout(var3, …)	create(var4, …)

Usually it is not mandatory to specify the clauses. The compiler will analyze your code to guess what the best solution and will tell you that one operation was done implicitely. As a good pratice, we recommend to make all implicit operations explicit.

2.4 Exercise: Gaussian blurring filter

In this exercise, we create a picture on the GPU and then we apply a blur filter. For each pixel, the value is computed as the weighted sum of the 24 neighbors and itself with the stencil shown below:

Note: In Fortran the weights are adjusted because we do not have unsigned integers.

Your job is to offload the blur function. Make sure that you use the correct data clauses for “pic” and “blurred” variables.

The original picture is 4000x4000 pixels. We need 1 value for each RGB channel it means that the actual size of the matrix is 4000x12000 (3x4000).

```
%%idrrun -a
!!  examples/Fortran/blur_simple_exercise.f90
MODULE pixels
    USE OPENACC
    IMPLICIT NONE
    CONTAINS
        SUBROUTINE fill(pic,rows,cols)
            INTEGER(kind=1),DIMENSION(0:),INTENT(OUT) :: pic
            INTEGER,INTENT(IN)                        :: rows,cols
            INTEGER                                   :: i,j, val

            DO i=0, rows-1
                DO j=0, 3*cols-1
                     val = i+(MOD(j,3))*j+MOD(i,256)
                     pic(i*3*cols+j) = MOD(val,128)
                END DO
            END DO
```

(continues on next page)

(continued from previous page)

```
        END SUBROUTINE fill

        SUBROUTINE blur(pic, blurred, rows, cols)
            INTEGER(kind=1),DIMENSION(0:),INTENT(IN)  :: pic
            INTEGER(kind=1),DIMENSION(0:),INTENT(OUT) :: blurred
            INTEGER,DIMENSION(0:4,0:4)                :: coefs
            INTEGER,INTENT(IN)                         :: rows,cols
            INTEGER(kind=4)                           :: i,j,i_c,j_c,l,pix
            INTEGER :: my_unit
            coefs(0,:)= (/ 1, 2, 3, 3, 1 /)
            coefs(1,:)= (/ 2, 8, 12, 8, 2 /)
            coefs(2,:)= (/ 3, 12, 14, 12, 3 /)
            coefs(3,:)= (/ 2, 8, 12, 8, 2 /)
            coefs(4,:)= (/ 1, 2, 3, 2, 1 /)
            DO i=2,rows-3
                DO j=2,cols-3
                    DO l=0,2
                        pix = 0
                        DO i_c=0,4
                            DO j_c=0,4
                                pix = pix + pic((i+i_c-2)*3*cols+(j+j_c-
 ↪2)*3+l)*(coefs(i_c,j_c))
                            END DO
                        END DO
                        blurred(i*3*cols+j*3+l)=pix/128
                    END DO
                END DO
            END DO
        END SUBROUTINE blur

        SUBROUTINE out_pic(pic, name)
            INTEGER(kind=1),DIMENSION(0:),INTENT(IN) :: pic
            CHARACTER(len=*),INTENT(IN)              :: name
            INTEGER                                  :: my_unit

            OPEN(NEWUNIT=my_unit, FILE=name, status='replace' ,ACCESS="stream", FORM=
 ↪"unformatted")
            WRITE(my_unit,POS=1) pic
            CLOSE(my_unit)
        END SUBROUTINE out_pic

end module pixels

PROGRAM blur_pix
    use PIXELS
    implicit none

    integer                                        :: rows, cols, i, long, j, check,␣
 ↪numarg
    integer  (kind=1), dimension(:), allocatable :: pic,blurred_pic
    character(len=: ), allocatable               :: arg
    integer, dimension(2)                        :: n

    rows = 4000
    cols = 4000
```

(continues on next page)

(continued from previous page)

```
    write(6,"(a23,i7,a2,i7)") "Size of the picture is ",rows," x",cols
    allocate(pic(0:rows*3*cols), blurred_pic(0:rows*3*cols))

    call fill(pic,rows,cols)
    call blur(pic, blurred_pic, rows, cols)

    call out_pic(pic, "pic.rgb")
    call out_pic(blurred_pic, "blurred.rgb")

    deallocate(pic, blurred_pic)
END PROGRAM blur_pix
```

```
from idrcomp import compare_rgb
compare_rgb("pic.rgb", "blurred.rgb", 4000, 4000)
```

2.4.1 Solution

```
%%idrrun -a
!!  examples/Fortran/blur_simple_solution.f90
MODULE pixels
    USE OPENACC
    IMPLICIT NONE
    CONTAINS
        SUBROUTINE fill(pic,rows,cols)
            INTEGER(kind=1),DIMENSION(0:),INTENT(OUT) :: pic
            INTEGER,INTENT(IN)                        :: rows,cols
            INTEGER                                   :: i,j, my_unit, val

            DO i=0, rows-1
                DO j=0, 3*cols-1
                     val = i+(MOD(j,3))*j+MOD(i,256)
                     pic(i*3*cols+j) = MOD(val,128)
                END DO
            END DO
        END SUBROUTINE fill

        SUBROUTINE blur(pic, blurred, rows, cols)
            INTEGER(kind=1),DIMENSION(0:),INTENT(IN)  :: pic
            INTEGER(kind=1),DIMENSION(0:),INTENT(OUT) :: blurred
            INTEGER,DIMENSION(0:4,0:4)               :: coefs
            INTEGER,INTENT(IN)                        :: rows,cols
            INTEGER(kind=4)                          :: i,j,i_c,j_c,l,pix
            INTEGER :: my_unit
            coefs(0,:)= (/ 1, 2, 3, 3, 1 /)
            coefs(1,:)= (/ 2, 8, 12, 8, 2 /)
            coefs(2,:)= (/ 3, 12, 14, 12, 3 /)
            coefs(3,:)= (/ 2, 8, 12, 8, 2 /)
            coefs(4,:)= (/ 1, 2, 3, 2, 1 /)
            !$acc parallel loop copyin(pic(0:), coefs(0:,0:)) copyout(blurred(0:))
            DO i=2,rows-3
                DO j=2,cols-3
                    DO l=0,2
                        pix = 0
```

(continues on next page)

(continued from previous page)

```
                        DO i_c=0,4
                            DO j_c=0,4
                                pix = pix + pic((i+i_c-2)*3*cols+(j+j_c-
 ↪2)*3+l)*(coefs(i_c,j_c))
                            END DO
                        END DO
                        blurred(i*3*cols+j*3+l)=pix/128
                    END DO
                END DO
            END DO
        END SUBROUTINE blur

        SUBROUTINE out_pic(pic, name)
            INTEGER(kind=1),DIMENSION(0:),INTENT(IN) :: pic
            CHARACTER(len=*),INTENT(IN)              :: name
            INTEGER                                  :: my_unit

            OPEN(NEWUNIT=my_unit, FILE=name, status='replace' ,ACCESS="stream", FORM=
 ↪"unformatted")
            WRITE(my_unit,POS=1) pic
            CLOSE(my_unit)
        END SUBROUTINE out_pic

end module pixels

PROGRAM blur_pix
    use PIXELS
    implicit none

    integer                                        :: rows, cols, i, long, j, check,␣
 ↪numarg
    integer  (kind=1), dimension(:), allocatable :: pic,blurred_pic
    character(len=: ), allocatable               :: arg
    integer, dimension(2)                        :: n

    rows = 4000
    cols = 4000

    write(6,"(a23,i7,a2,i7)") "Size of the picture is ",rows," x",cols
    allocate(pic(0:rows*3*cols), blurred_pic(0:rows*3*cols))

    call fill(pic,rows,cols)
    call blur(pic, blurred_pic, rows, cols)

    call out_pic(pic, "pic.rgb")
    call out_pic(blurred_pic, "blurred.rgb")

    deallocate(pic, blurred_pic)
END PROGRAM blur_pix
```

```
from idrcomp import compare_rgb
compare_rgb("pic.rgb", "blurred.rgb", 4000, 4000)
```

2.5 Reductions with OpenACC

Your code is performing a reduction when a loop is updating at each cycle the same variable:

For example, if you perform the sum of all elements in an array:

```
do i = 1, size_array
    summation = summation + array(i)
enddo
```

If you run your code sequentially no problems occur. However we are here to use a massively parallel device to accelerate the computation.

In this case we have to be careful since simultaneous read/write operations can be performed on the same variable. The result is not sure anymore because we have a race condition.

For some operations, OpenACC offers an efficient mechanism if you use the *reduction(operation:var1,var2,...)* clause which is available for the directives:

- `!$acc loop reduction(op:var1)`
- `!$acc parallel reduction(op:var1)`
- `!$acc kernels reduction(op:var1)`
- `!$acc serial reduction(op:var1)`

Important: Please note that for a lot of cases, the NVIDIA compiler (formerly PGI) is able to detect that a reduction is needed and will add it implicitly. We advise you make explicit all implicit operations for code readability/maintenance.

2.5.1 Available operations

The set of operations is limited. We give here the most common:

Operator	Operation	Syntax
+	sum	`reduction(+:var1, ...)`
*	product	`reduction(*:var2, ...)`
max	find maximum	`reduction(max:var3, ...)`
min	find minimum	`reduction(min:var4,...)`

Other operators are available, please refer to the OpenACC specification for a complete list.

Reduction on several variables

If you perform a reduction with the same operation on several variables then you can give a comma separated list after the colon:

```
!$acc parallel loop reduction(+:var1, var2,...)
```

If you perform reductions with different operators then you have to specify a *reduction* clause for each operator:

```
!$acc parallel reduction(+:var1, var2) reduction(max:var3) reduction(*: var4)
```

2.5.2 Exercise

Let's do some statistics on the exponential function. The goal is to compute

- the integral of the function between 0 and π using the trapezoidal method
- the maximum value
- the minimum value

You have to:

- Run the following example on the CPU. How much time does it take to run?
- Add the directives necessary to create one kernel for the loop that will run on the GPU
- Run the computation on the GPU. How much time does it take?

Your solution is considered correct if no implicit operation is reported by the compiler.

```
%%idrrun -a
!!  examples/Fortran/reduction_exponential_exercise.f90
program reduction_exponential
    use iso_fortran_env, only : INT32, REAL64
    implicit none
    ! current position and values
    real   (kind=REAL64) :: x, y, x_p
    real   (kind=REAL64) :: double_min, double_max, begin, fortran_pi, summation
    real   (kind=REAL64) :: step_l
    ! number of division of the function
    integer(kind=INT32 ) :: nsteps
    integer(kind=INT32 ) :: i

    nsteps     = 1e9
    begin      = 0.0_real64                  ! x min
    fortran_pi = acos(-1.0_real64)           ! x max
    summation  = 0.0_real64                  ! sum of elements
    step_l     = (fortran_pi - begin)/nsteps ! length of the step

    double_min = huge(double_min)
    double_max = tiny(double_max)

    do i = 1, nsteps
        x   =  i * step_l
        x_p = (i+1) * step_l
        y   = (exp(x)+exp(x_p))*0.5_real64
        summation = summation + y
        if (y .lt. double_min) double_min = y
        if (y .gt. double_max) double_max = y
    enddo

    ! print the stats
    write(0,"(a38,f20.8)") "The MINimum value of the function is: ",double_min
    write(0,"(a38,f20.8)") "The MAXimum value of the function is: ",double_max
    write(0,"(a33,f3.1,a1,f8.6,a6,f20.8)") "The integral of the function on [", &
                                           begin,",",fortran_pi,"] is: ",
 ↪summation*step_l
    write(0,"(a18,es20.8)") "   difference is: ",exp(fortran_pi)-exp(begin)-
 ↪summation*step_l
end program reduction_exponential
```

Solution

```
%%idrrun -a
!!  examples/Fortran/reduction_exponential_solution.f90
program reduction_exponential
    use iso_fortran_env, only : INT32, REAL64
    implicit none
    ! current position and values
    real   (kind=REAL64) :: x, y, x_p
    real   (kind=REAL64) :: double_min, double_max, begin, fortran_pi, summation
    real   (kind=REAL64) :: step_l
    ! number of division of the function
    integer(kind=INT32 ) :: nsteps
    integer(kind=INT32 ) :: i

    nsteps     = 1e9
    begin      = 0.0_real64                  ! x min
    fortran_pi = acos(-1.0_real64)           ! x max
    summation  = 0.0_real64                  ! sum of elements
    step_l     = (fortran_pi - begin)/nsteps ! length of the step

    double_min = huge(double_min)
    double_max = tiny(double_max)
    !$acc parallel loop reduction(+:summation) reduction(min:double_min)␣
 ↪reduction(max:double_max)
    do i = 1, nsteps
        x   =  i * step_l
        x_p = (i+1) * step_l
        y   = (exp(x)+exp(x_p))*0.5_real64
        summation = summation + y
        if (y .lt. double_min) double_min = y
        if (y .gt. double_max) double_max = y
    enddo

    ! print the stats
    write(0,"(a38,f20.8)") "The MINimum value of the function is: ",double_min
    write(0,"(a38,f20.8)") "The MAXimum value of the function is: ",double_max
    write(0,"(a33,f3.1,a1,f8.6,a6,f20.8)") "The integral of the function on [",begin,␣
 ↪&
                                          ","fortran_pi,"] is: ",summation*step_l
    write(0,"(a18,es20.8)") "   difference is: ",exp(fortran_pi)-exp(begin)-
 ↪summation*step_l
end program reduction_exponential
```

2.5.3 Important Notes

- A special kernel is created for reduction. With NVIDIA compiler its name is the name of the "parent" kernel with _red appended.
- You may want to use other directives to "emulate" the behavior of a reduction (it is possible by using *atomic* operations). We strongly discourage you from doing this. The *reduction* clause is much more efficient.

Requirements:

- Get started
- Data Management

CHAPTER

THREE

MANUAL BUILDING OF AN OPENACC CODE

During the training course, the building of examples will be done just by executing the code cells. Even though the command line is always printed, we think it is important to practice the building process.

3.1 Build with NVIDIA compilers

The compilers are:

- nvc: C compiler
- nvc++: C++ compiler
- nvfortran: Fortran compiler

3.1.1 Compiler options for OpenACC

- `-acc`: the compiler will recognize the OpenACC directives

  OpenACC is also able to generate code for multicore CPUs (close to OpenMP).

  Some interesting options are:

  - `-acc=gpu`: to build for GPU
  - `-acc=multicore`: to build for CPU (multithreaded)
  - `-acc=host`: to build for CPU (sequential)
  - `-acc=noautopar`: disable the automatic parallelization inside `parallel` regions (the default is `-acc=autopar`)

All options can be found in the documentation.

- `-gpu`: GPU-specific options to be passed to the compiler

  Some interesting options are:

  - `-gpu=ccXX`: specify the compute capability for which the code has to be built

    The list is available at https://developer.nvidia.com/cuda-gpus#compute.

  - `-gpu=managed`: activate NVIDIA Unified Memory (with it you can ignore data transfers, but it might fail sometime)
  - `-gpu=pinned`: activate *pinned* memory. It can help to improve the performance of data transfers
  - `-lineinfo`: generate debugging line information; less overhead than -g

All options can be found in the documentation..

- `-Minfo`: the compiler prints information about the optimizations it uses
  - `-Minfo=accel`: information about OpenACC (Mandatory in this training course!)
  - `-Minfo=all`: all optimizations are printed (OpenACC, vectorization, FMA, …). We recommend to use this option.

3.1.2 Other useful compiler options

- `-o exec_name`: name of the executable
- `-Ox`: level of optimization (0 <= x <=4)
- `-Og`: optimize debugging experience and enables optimizations that do not interfere with debugging.
- `-fast`: equivalent to `-O2 -Munroll=c:1 -Mnoframei -Mlre`
- `-g`: add debugging symbols
- `-gopt`: instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when -g is not specified.

You can specify a comma-separated list of options for each flag.

3.1.3 Examples

For instance to compile a C source code for GPU on NVIDIA V100 (Compute Capability 7.0), the following line should be executed:

```
nvfortran -acc=gpu,noautopar -gpu=cc70,managed -Minfo=all mysource.f90 -o myprog
```

The example below shows how to compile for the following setup:

- OpenACC for GPU `-acc=gpu`
- Compile for Volta architecture `-gpu=cc70`
- Activate optimizations `-fast`
- Print optimizations and OpenACC information `-Minfo=all`

```
ACCFLAGS = -acc=gpu -gpu=cc70
OPTFLAGS = -fast
INFOFLAGS = -Minfo=all

myacc_exec: myacc.f90
    nvfortran -o myacc_exec $(ACCFLAGS) $(OPTFLAGS) $(INFOFLAGS) myacc.f90
```

3.2 Build with GCC compilers

The compilers are:

- gcc: C compiler
- gxx: C++ compiler
- gfortran: Fortran compiler

3.2.1 Compiler options for OpenACC

- `-fopenacc`: the compiler will recognize the OpenACC directives
- `-foffload`: enables the compiler to generate a code for the accelerator. Compilers for host and accelerator are separated
  - `-foffload=nvptx-none`: compile for NVIDIA devices

  It can be used to pass options such as optimization, libraries to link, etc (`-foffload=-O3 -foffload=-lm`). You can enclose options between "" and give it to `-foffload`.

3.2.2 Other useful compiler options

- `-o exec_name`: name of the executable
- `-Ox`: level of optimization (0 <= x <=3)
- `-g`: add debugging symbols

3.2.3 Example

The example shows how to compile for the following setup:

- OpenACC for GPU `-fopenacc`
- Compile for NVIDIA GPU `-foffload=nvptx-none`
- Activate optimizations `-O3 -foffload=-O3`

```
ACCFLAGS = -fopenacc -foffload=nvptx-none
OPTFLAGS = -O3 -foffload=-O3
INFOFLAGS = -fopt-info

myacc_exec: myacc.f90
    gfortran -o myacc_exec $(ACCFLAGS) $(OPTFLAGS) $(INFOFLAGS) myacc.f90
```

3.3 Exercise

- Execute the following cell which produces a file (just add the name you want after `writefile`).
- Open a terminal (File -> New -> Terminal)
- Load the compiler you wish to use (for example: `module load nvidia-compilers/21.7`)
- Use the information above to compile the file, you might need to modify the extension of the file "exercise" to "exercise.c" or "exercise.f90"
- If you want to make sure that the code ran on GPU you can do `export NVCOMPILER_ACC_TIME=1`
- Execute the code with `srun -n 1 --cpus-per-task=10 -A for@v100 --gres=gpu:1 --time=00:03:00 --hint=nomultithread --qos=qos_gpu-dev time <executable_name>`
- Bonus: Compile the code without OpenACC support and compare the elapsed time in both cases.

```
%%writefile exercise
!!  examples/Fortran/Manual_building_exercise.f90
program manual_build
    use iso_fortran_env, only : INT32, REAL64
    implicit none

    real   (kind=REAL64), dimension(:), allocatable :: A, B
    real   (kind=REAL64)                            :: summation, fortran_pi
    integer(kind=INT32 )                            :: system_size
    integer(kind=INT32 )                            :: i

    fortran_pi  = acos(-1.0_real64)
    summation   = 0.0_real64
    system_size = 1e9
    allocate(A(system_size), B(system_size))

    !$acc data create(A(:), B(:))
    !$acc parallel loop present(A(:), B(:))
    do i = 1, system_size
        A(i) = sin(i*fortran_pi/system_size) * sin(i*fortran_pi/system_size)
        B(i) = cos(i*fortran_pi/system_size) * cos(i*fortran_pi/system_size)
    enddo

    call inplace_sum(A, B, system_size)

    !$acc parallel loop present(A(:), B(:)) reduction(+:summation)
    do i = 1, system_size
        summation = summation + A(i)
    enddo
    !$acc end data
    write(0,"(a29,f10.8)") "This should be close to 1.0: ", summation/dble(system_
↪size)
    deallocate(A, B)
    contains
        subroutine inplace_sum(A, B, n)
        real   (kind=REAL64), dimension(:), intent(inout) :: A, B
        integer(kind=INT32 ), intent(in)                  :: n
        !$acc parallel loop present(A(:), B(:))
        do i = 1, n
            A(i) = A(i) + B(i)
```

(continues on next page)

(continued from previous page)

```
        enddo
        end subroutine inplace_sum
end program manual_build
```

CHAPTER

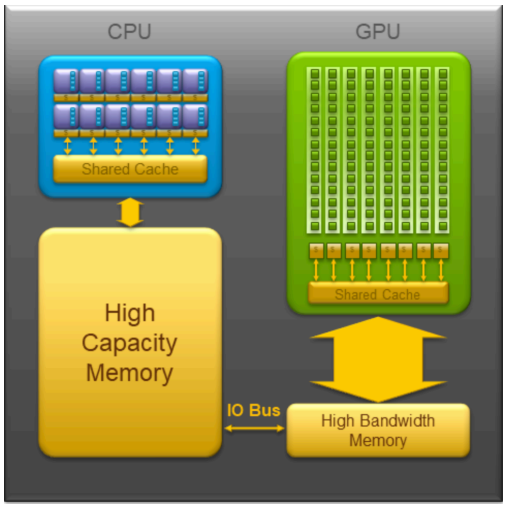
FOUR

DATA MANAGEMENT

4.1 Why do we have to care about data transfers?

The main bottleneck in using GPUs for computing is data transfers between the host and the GPU.

Let's have a look at the bandwidths.



On this picture the size of the arrows represents the bandwidth. To have a better idea here are some numbers:

- GPU to its internal memory (HBM2): 900 GB/s
- GPU to CPU via PCIe: 16 GB/s
- GPU to GPU via NVLink: 25 GB/s
- CPU to RAM (DDR4): 128 GB/s

So if you have to remember only one thing: take care of memory transfers.

4.2 The easy way: NVIDIA managed memory

NVIDIA offers a feature called Unified Memory which allows developers to "forget" about data transfers. The memory space of the host and the GPU are shared so that the normal *page fault* mechanism can be used to manage transfers.

This feature is activated with the compiler options:

- NVIDIA compilers: `-gpu:managed`
- PGI: `-ta=tesla:managed`

This might give good performance results and you might just forget explicit data transfers. However, depending on the complexity of your data structures, you might need to deal explicitly with data transfers. The next section gives an introduction to manual data management.

Unified Memory also allows to increase virtual memory space on GPU (so called GPU memory oversubscription).

4.3 Manual data movement

4.3.1 Data clauses

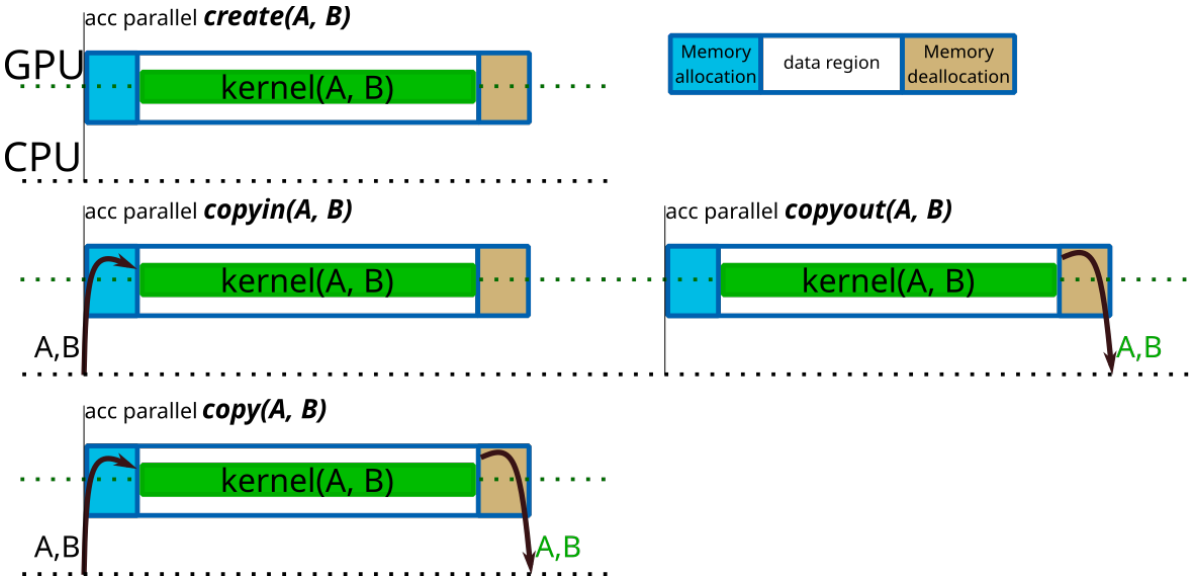
There are multiple data directives that accept the same data clauses. So we start with the data clauses and than continue with data directives.

In order to choose the right data clause for data transfers, you need to answer the following two questions:

- Does the kernel need the values computed beforehand by the CPU?
- Are the values computed inside the kernel needed on the CPU afterhand?

	Needed after	Not needed after
Needed before	`copy(var1, ...)`	`copyin(var2, ...)`
Not needed before	`copyout(var3, ...)`	`create(var4, ...)`

Figure below illustrates transfers, if any, between the CPU and the GPU for these four clauses.



Important: the presence of variables on the GPU is checked at runtime. If some variables are already found on the GPU, these clauses have no effect. It means that you cannot update variables (on the GPU at the region entrance or on the CPU at exit). You have to use the `acc update` directive in this case.

Other data clauses include:

- `present`: check if data is present in the GPU memory; an error is raised if it is not the case
- `deviceptr`: pass the GPU pointer; used for interoperability between other APIs (e.g. CUDA, Thrust) and OpenACC
- `attach`: attach a pointer to memory already allocated in the GPU

Array shapes and partial data transfers

For array transfers, full or partial, one has to follow the language syntax.

In Fortran, you have to specify the range of the array in the format `(first index:last index)`.

```
!$acc data copyout(myarray(1:size))
   ! Some really fast kernels
!$acc end data
```

For partial data transfer, you can specify a subarray. For example:

```
!$acc data copyout(myarray(2:size-1))
```

Before moving on to data directives, some vocabulary needs to be introduced. According to data lifetime on the GPU, two types of data regions can be distinguished: *structured* and *unstructured*. Structured data regions are defined within the same scope (e.g. routine), while unstructured data regions allow data creation and deletion in different scopes.

4.3.2 Implicit structured data regions associated with compute constructs

Any of the three compute constructs – `parallel`, `kernels`, or `serial` – opens an implicit data region. Data transfers will occur just before the kernel starts and just after the kernel ends.

In the Get started notebook, we have already seen that it is possible to specify data clauses in `acc parallel` to manage our variables. The compiler checks what variables (scalar or arrays) are needed in the kernel and will try to add the *data clauses* necessary.

Exercise

- Create a parallel region for each loop.
- For each parallel region, what *data clause* should be added?

```
%%idrrun -a
!!  examples/Fortran/Data_Management_vector_sum_exercise.f90
program vector_sum
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32), parameter              :: system_size  = 10000
    integer(kind=INT32), dimension(system_size) :: a, b, c
```

(continues on next page)

(continued from previous page)

```
    integer(kind=INT32)                          :: i

    ! Insert OpenACC directive
    do i = 1, system_size
        a(i) = i
        b(i) = i * 2
    enddo

    ! Insert OpenACC directive
    do i = 1, system_size
       c(i) = a(i) + b(i)
    enddo

    write(0,"(a22,i3)") "value at position 12: ", c(12)

end program vector_sum
```

Answer

- For each parallel region, what *data clause* should be added?
  - Loop 1: The initialization of a and b is done directly on GPU so we don't need to copy the values from CPU. Variables a and b are used to compute c after execution of the first parallel region. We need to `copyout` a and b.
  - Loop 2: We need the values of a and b to compute c. This computation is the initialization of c. We print the value of one element of c after execution. The values of a and b are not needed anymore. We need to `copyin` a and b. We need to `copyout` c.

```
%%idrrun -a
!!  examples/Fortran/Data_Management_vector_sum_solution.f90
program vector_sum
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32), parameter               :: system_size  = 10000
    integer(kind=INT32), dimension(system_size) :: a, b, c
    integer(kind=INT32)                          :: i

    !$acc parallel loop copyout(a(:), b(:))
    do i = 1, system_size
        a(i) = i
        b(i) = i * 2
    enddo

    !$acc parallel loop copyin(a(:), b(:)) copyout(c(:))
    do i = 1, system_size
       c(i) = a(i) + b(i)
    enddo

    write(0,"(a22,i3)") "value at position 12: ", c(12)

end program vector_sum
```

If you use NVIDIA compilers (formerly PGI), most of the time the right directives will be added *implicitly*.

Our advice is to make explicit all actions performed implicitly by the compiler. It will help you to keep a code understandable and avoid porting problems if you have to change compiler.

All compilers might not choose the same default behavior.

4.3.3 Explicit structured data regions `acc data`

Using the *data regions* associated to kernels is quite convenient and is a good strategy for incremental porting of your code.

However, this results in a large number of data transfers that can be avoided.

If we take a look at the previous example, we count 5 data transfers:

- Loop 1: copyout(a, b)
- Loop 2: copyin(a, b) copyout(c)

If we look closely we can see that we do not need a and b on the CPU between the kernels. It means that data transfers of a and b at the end of kernel1 and at the beginning of kernel2 are useless.

The solution is to encapsulate the two loops in a *structured data region* that you can open we the directive `acc data`. The syntax is:

```
!$acc data <data clauses>
    ! Your code
!$acc end data
```

Exercise

Analyze the code to create a *structured data region* that encompasses both loops. The data clause `present` have been added to the *data region associated with kernels*. You should not remove this part.

How many data transfers occurred?

```
%%idrrun -a
!!  examples/Fortran/Data_Management_structured_data_region_exercise.f90
program vector_addition
    use iso_fortran_env, only : INT32
    use openacc
    implicit none

    integer(kind=INT32), parameter              :: system_size  = 10000
    integer(kind=INT32), dimension(system_size) :: a, b, c
    integer(kind=INT32)                         :: i

!   Structured data region

    !$acc parallel loop present(a(:), b(:))
    do i = 1, system_size
        a(i) = i
        b(i) = i * 2
    enddo

    !$acc parallel loop present(a(:), b(:), c(:))
```

(continues on next page)

(continued from previous page)

```
    do i = 1, system_size
       c(i) = a(i) + b(i)
    enddo

!   End of structured data region

    write(0,"(a22,i3)") "value at position 12: ", c(12)
end program vector_addition
```

Solution

```
%%idrrun -a
!!  examples/Fortran/Data_Management_structured_data_region_solution.f90
program vector_addition
    use iso_fortran_env, only : INT32
    use openacc
    implicit none

    integer(kind=INT32), parameter              :: system_size  = 10000
    integer(kind=INT32), dimension(system_size) :: a, b, c
    integer(kind=INT32)                         :: i

!   Structured data region
    !$acc data create(a, b) copyout(c)

    !$acc parallel loop present(a(:), b(:))
    do i = 1, system_size
        a(i) = i
        b(i) = i * 2
    enddo

    !$acc parallel loop present(a(:), b(:), c(:))
    do i = 1, system_size
       c(i) = a(i) + b(i)
    enddo

!   End of structured data region
    !$acc end data

    write(0,"(a22,i3)") "value at position 12: ", c(12)
end program vector_addition
```

When using *structured data region* we advise to use the `present` data clause which tells that the data should already be in GPU memory.

WRONG example

The example given below doesn't give the right results on the CPU. Why?

```
%%idrrun -a
!!  examples/Fortran/Data_Management_wrong_example.f90
program wrong_usage
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32 ), parameter              :: system_size = 10000
    real   (kind=real64), dimension(system_size) :: a, b, c
    integer(kind=INT32 )                         :: i

!  Structured data region
    !$acc data create(a, b) copyout(c)

        !$acc parallel loop present(a(:), b(:))
        do i = 1, system_size
            a(i) = i
            b(i) = i*2
        enddo

        ! We update an element of the array on the CPU
        a(12) = 42

        !$acc parallel loop present(b(:), c(:)) copyin(a(:))
        do i = 1, system_size
            c(i) = a(i) + b(i)
        enddo
    !$acc end data
    write(0,"(a22,f10.5)") "value at position 12: ", c(12)
end program wrong_usage
```

This example is here to emphasize that you cannot update data with data clauses. It has an unintended behavior.

4.3.4 Updating data

Let's say that all your code is not ported to the GPU. Then it means that you will have some variables (arrays or scalars) for which both, the CPU and the GPU, will perform computation.

To keep the results correct, you will have to update those variables when needed.

`acc update device`

To update the value a variable has on the GPU with what the CPU has you have to use:

```
!$acc update device(var1, var2, ...)
```

Important: The directive cannot be used inside a compute construct.

`acc update self`

Once again if all your code is not ported on GPU the values computed on the GPU may be needed afterwards on the CPU.

The directive to use is:

```
!$acc update self(var1, var2, ...)
```

Correct the previos example in order to obtain correct restuls:

```
%%idrrun -a
!!  examples/Fortran/Data_Management_wrong_example.f90
program wrong_usage
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    integer(kind=INT32 ), parameter              :: system_size = 10000
    real   (kind=real64), dimension(system_size) :: a, b, c
    integer(kind=INT32 )                         :: i

!  Structured data region
    !$acc data create(a, b) copyout(c)

        !$acc parallel loop present(a(:), b(:))
        do i = 1, system_size
            a(i) = i
            b(i) = i*2
        enddo

        ! We update an element of the array on the CPU
        a(12) = 42

        !$acc parallel loop present(b(:), c(:)) copyin(a(:))
        do i = 1, system_size
            c(i) = a(i) + b(i)
        enddo
    !$acc end data
    write(0,"(a22,f10.5)") "value at position 12: ", c(12)
end program wrong_usage
```

4.3.5 Explicit unstructured data regions `acc enter data`

Each time you run a code on the GPU, a data region is created for the lifetime of the program.

There are two directives to manage data inside this region:

- `acc enter data <input data clause>`: to put data inside the region (allocate memory, copy data from the CPU to the GPU)
- `acc exit data <output data clause>`: to remove data (deallocate memory, copy data from the GPU to the CPU)

This feature is helpful when you have your variables declared at one point of your code and used in another one (modular programming). You can allocate memory as soon as the variable is created and just use *present* when you create kernels.

acc enter data

This directive is used to put data on the GPU inside the unstructured data region spanning the lifetime of the program. It will allocate the memory necessary for the variables and, if asked, copy the data present on the CPU to the GPU.

It accepts the clauses:

- *create*: allocate memory on the GPU
- *copyin*: allocate memory on the GPU and initialize it with the values that the variable has on the CPU
- *attach*: attach a pointer to memory already in the GPU

The most common clauses are *create* and *copyin*. The *attach* clause is a bit more advanced and is not covered in this part.

Here is an example of syntax:

```
!$acc enter data copyin(var1(:), ...) create(var2(:))
```

Important: the directive must appear after the allocation of the memory on the CPU.

```
real, dimension(:,:,:) :: var
allocate(var(nx,ny,nz))
!$acc enter data create(var(:,:,:))
```

Otherwise you will have a runtime error.

acc exit data

By default, the memory allocated with `acc enter data` is freed at the end of the program. But usually you do not have access to very large memory on the GPU (it depends on the card but usually you have access to a few tens of GB) and it might be necessary to have a fine control on what is present.

The directive `acc exit data <output data clause>` is used to remove data from the GPU. It accepts the clauses:

- *copyout*: copy to the CPU the values that the variable have on the GPU
- *delete*: free the memory on the GPU
- *detach*: remove the attachment of the pointer to the memory

Important: the directive must appear before memory deallocation on the CPU.

```
!$acc exit data delete(var)
deallocate(var)
```

Otherwise you will have a runtime error.

Exercise

In this exercise you have to add data management directives in order to:

- allocate memory on the GPU for `array`
- perform the initialization on the GPU
- free the memory on the GPU.

```
%%idrrun -a
!!  examples/Fortran/Data_Management_unstructured_exercise.f90
program allocate_array_separately
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    real   (kind=REAL64), dimension(:), allocatable :: array
    integer(kind=INT32 )                            :: system_size
    integer(kind=INT32 )                            :: i

    system_size = 100000

    call init(array, system_size)

    do i = 1, system_size
        array(i) = dble(i)
    enddo

    write(0,*) array(42)

    deallocate(array)

    contains
     subroutine init(array, system_size)

     real   (kind=REAL64), dimension(:), allocatable, intent(inout) :: array
     integer(kind=INT32 ), intent(in)                               :: system_size

     allocate(array(system_size))

     end subroutine init
end program allocate_array_separately
```

Solution

```
%%idrrun -a
!!  examples/Fortran/Data_Management_unstructured_solution.f90
program allocate_array_separately
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    real   (kind=REAL64), dimension(:), allocatable :: array
    integer(kind=INT32 )                            :: system_size
    integer(kind=INT32 )                            :: i
```

(continues on next page)

(continued from previous page)

```
    system_size = 100000

    call init(array, system_size)

    !$acc parallel loop present(array(:))
    do i = 1, system_size
        array(i) = dble(i)
    enddo

    !$acc exit data copyout(array(:))
    write(0,*) array(42)

    deallocate(array)

    contains
     subroutine init(array, system_size)

     real   (kind=REAL64), dimension(:), allocatable, intent(inout) :: array
     integer(kind=INT32 ), intent(in)                               :: system_size

     allocate(array(system_size))
     !$acc enter data create(array(1:system_size))

     end subroutine init
end program allocate_array_separately
```

4.3.6 Implicit data regions `acc declare`

An implicit data region is created for a program and each subprogram. You can manage data inside these data regions using `acc declare` directive.

An implicit data region is created for each function you write. You can manage data inside it with the `acc declare` directive.

```
integer, parameter :: size = 1000000
real               :: array(size)
!$acc declare create(array(1:size))
```

In Fortran this directive can also be used for variables declared inside modules.

In addition to regular data causes, it accepts `device_resident` cause for variables needed only on the GPU.

Example given below illustrates usage of this clause.

Example

In this example we normalize rows (C) or columns (Fortran) of a square matrix. The algorithm uses a temporary array (norms) which is only used on the GPU.

```
%%idrrun -a
!!  examples/Fortran/Data_Management_unstructured_declare_example.f90
module utils
    use iso_fortran_env, only : REAL64, INT32
    contains
        subroutine normalize_cols(mat, mat_size)
            real    (kind=REAL64), allocatable, dimension(:,:), intent(inout) :: mat
            integer (kind=INT32 )                             , intent(in)    :: mat_
 ↪size
            real    (kind=REAL64)                                               :: norm␣
 ↪= 0.0_real64
            integer (kind=INT32 )                                               :: i,j
            real    (kind=REAL64), allocatable, dimension(:) :: norms
            !$acc declare device_resident(norms(:))
            allocate(norms(mat_size))
!! Compute the L1 norm of each column
            !$acc parallel loop present(mat(:,:), norms(:))
            do j = 1, mat_size
                norm = 0
                !$acc loop reduction(+:norm)
                do i = 1, mat_size
                    norm = norm + mat(i,j)
                enddo
                norms(j) = norm
            enddo
!! Divide each element of the columns by the L1 norm
            !$acc parallel loop present(mat(:,:), norms(:))
            do j = 1, mat_size
                do i = 1, mat_size
                    mat(i,j) = mat(i,j)/norms(j)
                enddo
            enddo
        end subroutine normalize_cols
end module utils

program normalize
    use utils
    real    (kind=REAL64), allocatable, dimension(:,:)  :: mat
    real    (kind=REAL64)                               :: mat_sum
    integer (kind=INT32)                                :: mat_size=2000
    integer (kind=INT32)                                :: i, j

    allocate(mat(mat_size, mat_size))
    !$acc enter data create(mat)
    call random_number(mat)
    !$acc update device(mat(:,:))
    call normalize_cols(mat, mat_size)
!! Compute the sum of all elements of the matrix
    !$acc parallel loop present(mat(:,:)) reduction(+:mat_sum)
    do j = 1, mat_size
        do i = 1, mat_size
            mat_sum = mat_sum + mat(i,j)
```

(continues on next page)

(continued from previous page)

```
        enddo
    enddo
    !$acc exit data delete(mat)
    deallocate(mat)
    print *, mat_sum, "=", mat_size, "?"
end program normalize
```

Requirements:

- Get started
- Data management

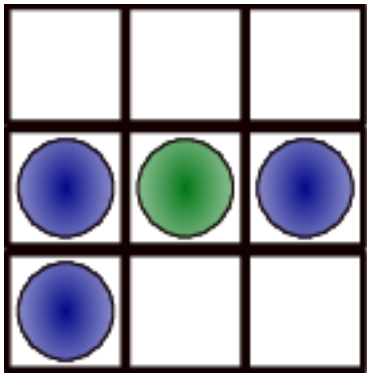
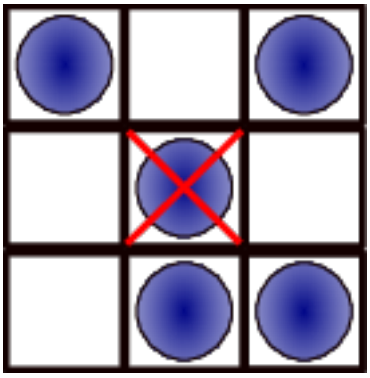
CHAPTER

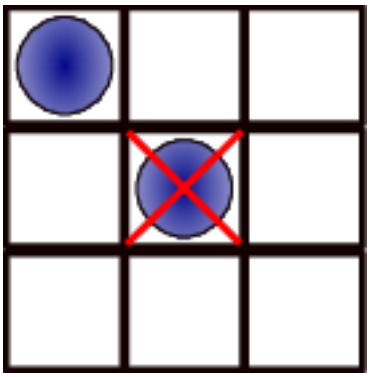
FIVE

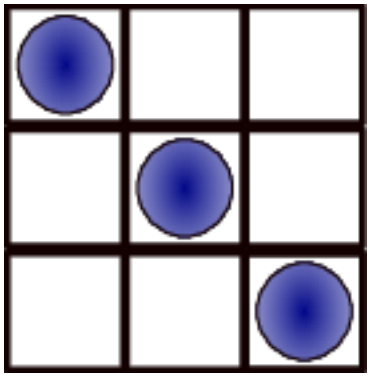
HANDS-ON GAME OF LIFE

The Game of Life is a cellular automaton developed by John Conway in 1970. In this "Game" a grid is filled with an **initial state** of cells having either the status "dead" or "alive". From this initial state, several generations are computed and we can follow the evolution of the cells for each **generation**.

The rules are simple:

- For "dead" cells:
  - If it has exactly 3 neighbors the cell becomes **alive** at the next generation 
- For "alive" cells:
  - If it has more than 3 neighbors the cell becomes **dead** because of overpopulation 
  - If it has less than 2 neighbors the cell becomes **dead** because of underpopulation



For all other situations the state of the cell is kept unchanged. 

5.1 What to do

In this hands-on you have to add the directives to perform the following actions:

- Copy the initial state of the world generated on the CPU to the GPU
- Make sure that the computation of the current generation and the saving of the previous one occur on the GPU
- Compute the number of cells alive for the current generation is done on the GPU
- The memory on the GPU is allocated and freed when the arrays are not needed anymore

```
%%idrrun  -a --cliopts "20000 1000 300"
!!  examples/Fortran/GameOfLife_exercise.f90
program gol
    implicit none
    integer :: rows, cols, generations
    integer, allocatable :: world(:,:), old_world(:,:)
    integer :: g
    integer :: long
    character(len=:), allocatable :: arg
    integer :: i
    integer, dimension(3) :: n
    ! Read cmd line args
    DO i=1, COMMAND_ARGUMENT_COUNT()
       CALL GET_COMMAND_ARGUMENT(NUMBER=i, LENGTH=long)
       ALLOCATE(CHARACTER(len=long) :: arg)
       CALL GET_COMMAND_ARGUMENT(NUMBER=i, VALUE=arg)
       READ(arg,'(i10)') n(i)
       DEALLOCATE(arg)
```

(continues on next page)

(continued from previous page)

```
    END DO
    rows=n(1)
    cols=n(2)
    generations=n(3)
    allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
    old_world(:,:) = 0
    call fill_world
    do g=1,generations
        call save_world(rows,cols, world(:,:), old_world(:,:))
        call next(rows,cols, world(:,:), old_world(:,:))
        print *, "Cells alive at generation ", g, ": ", alive(rows, cols, world)
        call output_world("generation", g, world(:,:), rows, cols)
    enddo
    deallocate(world, old_world)
    contains
    integer function alive(rows, cols, world) result(cells)
        implicit none
        integer, intent(in) :: rows, cols
        integer, intent(in) :: world(0:rows+1,0:cols+1)
        integer :: r,c
        cells = 0
        do r=1, rows
            do c=1, cols
               cells = cells + world(r,c)
            enddo
        enddo
        end function alive
    subroutine save_world(rows,cols,world, old_world)
        implicit none
        integer, intent(in) :: rows,cols
        integer, intent(in) :: world(0:rows+1,0:cols+1)
        integer, intent(out) :: old_world(0:rows+1,0:cols+1)
        integer :: r,c
        do r=1, rows
            do c=1, cols
                old_world(r,c) = world(r,c)
            enddo
        enddo
    end subroutine save_world

    subroutine next(rows,cols,world, old_world)
        implicit none
        integer, intent(in) :: rows,cols
        integer, intent(in) :: old_world(0:rows+1,0:cols+1)
        integer, intent(out) :: world(0:rows+1,0:cols+1)
        integer :: r,c
        integer :: neigh
        do r=1, rows
            do c=1, cols
        neigh = old_world(r-1,c-1)+old_world(r,c-1)+old_world(r+1,c-1)+&
                old_world(r-1,c)+old_world(r+1,c)+&
                old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
                if (old_world(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
                    world(r,c) = 0
                else if (neigh == 3) then
                    world(r,c) = 1
```

(continues on next page)

(continued from previous page)

```
                endif
            enddo
        enddo
    end subroutine next

    subroutine fill_world
        implicit none
        integer :: r,c, temp
        real*8 :: test
        do r=1,rows
            do c=1,cols
                call random_number(test)
                temp = mod(floor(test*100),4)
                if (temp.eq.0) then
                    temp = 1
                else
                    temp = 0
                endif
                world(r,c) = temp
            enddo
        enddo
    end subroutine fill_world

    subroutine output_world(name, g, world, rows, cols)
        implicit none
        character(len=*), intent(in) :: name
        character(len=1024) :: filename
        integer, intent(in) :: g
        integer, intent(in) :: rows,cols
        integer, intent(in) :: world(0:rows+1,0:cols+1)
        integer :: r,c
        integer :: my_unit
        integer(kind=1) :: output(0:rows+1,0:cols+1)
        write(filename,"(A,I5.5,A)") trim(name), g,".gray"
        print *, trim(filename)
        do r=0,rows+1
            do c=0,cols+1
                output(r,c) = mod(world(r,c),2)*127
            enddo
        enddo
        open(newunit=my_unit, file=filename, access="stream", form="unformatted")
        write(my_unit, pos=1) output
        close(my_unit)
    end subroutine output_world
end program gol
```

```
rows = 2000
cols = 1000

from idrcomp import convert_pic
import matplotlib.pyplot as plt
import glob
from PIL import Image
from ipywidgets import interact
```

(continues on next page)

(continued from previous page)

```
files = sorted(glob.glob("*.gray"))
images = [convert_pic(f, cols, rows, "L") for f in files]
print(images[0].size)
def view(i):
    crop = (155,65,360,270)
    plt.figure(figsize=(12,12))
    plt.imshow(images[i].crop(crop), cmap="Greys")
    plt.show()
interact(view, i=(0, len(images)-1))
```

5.2 Solution

```
%%idrrun  -a --cliopts "20000 1000 300"
!!  examples/Fortran/GameOfLife_solution.f90
program gol
    implicit none
    integer :: rows, cols, generations
    integer, allocatable :: world(:,:), old_world(:,:)
    integer :: g
    integer :: long
    character(len=:), allocatable :: arg
    integer :: i
    integer, dimension(3) :: n
    ! Read cmd line args
    DO i=1, COMMAND_ARGUMENT_COUNT()
       CALL GET_COMMAND_ARGUMENT(NUMBER=i, LENGTH=long)
       ALLOCATE(CHARACTER(len=long) :: arg)
       CALL GET_COMMAND_ARGUMENT(NUMBER=i, VALUE=arg)
       READ(arg,'(i10)') n(i)
       DEALLOCATE(arg)
    END DO
    rows=n(1)
    cols=n(2)
    generations=n(3)
    allocate(world(0:rows+1,0:cols+1), old_world(0:rows+1,0:cols+1))
    old_world(:,:) = 0
    call fill_world
    !$ACC enter data copyin(world, old_world)
    do g=1,generations
        call save_world(rows,cols, world(:,:), old_world(:,:))
        call next(rows,cols, world(:,:), old_world(:,:))
        print *, "Cells alive at generation ", g, ": ", alive(rows, cols, world)
        call output_world("generation", g, world(:,:), rows, cols)
    enddo
    !$ACC exit data delete(world, old_world)
    deallocate(world, old_world)
    contains
    integer function alive(rows, cols, world) result(cells)
        implicit none
        integer, intent(in) :: rows, cols
        integer, intent(in) :: world(0:rows+1,0:cols+1)
        integer :: r,c
        cells = 0
```

(continues on next page)

(continued from previous page)

```
        !$ACC parallel loop reduction(+:cells)
        do r=1, rows
            do c=1, cols
               cells = cells + world(r,c)
            enddo
        enddo
        end function alive
    subroutine save_world(rows,cols,world, old_world)
        implicit none
        integer, intent(in) :: rows,cols
        integer, intent(in) :: world(0:rows+1,0:cols+1)
        integer, intent(out) :: old_world(0:rows+1,0:cols+1)
        integer :: r,c
        !$ACC parallel loop
        do r=1, rows
            do c=1, cols
                old_world(r,c) = world(r,c)
            enddo
        enddo
    end subroutine save_world

    subroutine next(rows,cols,world, old_world)
        implicit none
        integer, intent(in) :: rows,cols
        integer, intent(in) :: old_world(0:rows+1,0:cols+1)
        integer, intent(out) :: world(0:rows+1,0:cols+1)
        integer :: r,c
        integer :: neigh
        !$ACC parallel loop
        do r=1, rows
            do c=1, cols
        neigh = old_world(r-1,c-1)+old_world(r,c-1)+old_world(r+1,c-1)+&
                old_world(r-1,c)+old_world(r+1,c)+&
                old_world(r-1,c+1)+old_world(r,c+1)+old_world(r+1,c+1)
                if (old_world(r,c) == 1 .and. (neigh<2.or.neigh>3) ) then
                    world(r,c) = 0
                else if (neigh == 3) then
                    world(r,c) = 1
                endif
            enddo
        enddo
    end subroutine next

    subroutine fill_world
        implicit none
        integer :: r,c, temp
        real*8 :: test
        do r=1,rows
            do c=1,cols
                call random_number(test)
                temp = mod(floor(test*100),4)
                if (temp.eq.0) then
                    temp = 1
                else
                    temp = 0
                endif
```

(continues on next page)

(continued from previous page)

```
                world(r,c) = temp
            enddo
        enddo
    end subroutine fill_world

    subroutine output_world(name, g, world, rows, cols)
        implicit none
        character(len=*), intent(in) :: name
        character(len=1024) :: filename
        integer, intent(in) :: g
        integer, intent(in) :: rows,cols
        integer, intent(in) :: world(0:rows+1,0:cols+1)
        integer :: r,c
        integer :: my_unit
        integer(kind=1) :: output(0:rows+1,0:cols+1)
        write(filename,"(A,I5.5,A)") trim(name), g,".gray"
        print *, trim(filename)
        !$ACC update self(world)
        do r=0,rows+1
            do c=0,cols+1
                output(r,c) = mod(world(r,c),2)*127
            enddo
        enddo
        open(newunit=my_unit, file=filename, access="stream", form="unformatted")
        write(my_unit, pos=1) output
        close(my_unit)
    end subroutine output_world
end program gol
```

```
rows = 2000
cols = 1000

from idrcomp import convert_pic
import matplotlib.pyplot as plt
import glob
from PIL import Image
from ipywidgets import interact

files = sorted(glob.glob("*.gray"))
images = [convert_pic(f, cols, rows, "L") for f in files]
print(images[0].size)
def view(i):
    crop = (155,65,360,270)
    plt.figure(figsize=(12,12))
    plt.imshow(images[i].crop(crop), cmap="Greys")
    plt.show()
interact(view, i=(0, len(images)-1))
```

Clean the pictures:

```
!rm *.gray
```

Part II

Day 2

CHAPTER

SIX

COMPUTE CONSTRUCTS

6.1 Giving more freedom to the compiler: `acc kernels`

We focus the training course on the usage of the `acc parallel` compute construct since it gives almost full control to the developer.

The OpenACC standard offers the possibility to give more freedom to the compiler with the `acc kernels` compute construct. The behavior is different as several kernels might be created from one `acc kernels` region. One kernel is generated for each nest of loops.

6.1.1 Syntax

The following example would generate 2 kernels (if reductions are present more kernels are generated to deal with it):

```
!$acc kernels
        ! 1st kernel generated
    !$acc loop
    do i = 0, system_size_i
        do j = 0, system_size_j
            ! Perform some computation
        enddo
    enddo

        ! 2nd kernel generated
    !$acc loop
    do i = 0, system_size_i
        ! Some more computation
    enddo
!$acc end kernels
```

It is almost equivalent to this example:

```
!$acc data <data clauses>
        ! 1st kernel generated
    !$acc parallel loop
    do i = 0, system_size_i
        do j = 0, system_size_j
            ! Perform some computation
        enddo
    enddo

        ! 2nd kernel generated
```

(continues on next page)

(continued from previous page)

```
    !$acc parallel loop
    do i = 0, system_size_i
        ! Some more computation
    enddo
!$acc end data
```

The main difference is the status of the scalar variables used in the compute construct. With `acc kernels` they are shared whereas with `acc parallel` they are private at the gang level.

The configuration of the kernels (number of gangs, workers and vector length) can be different.

6.1.2 Independent loops

The compiler is a very prudent software. If it detects that parallelizing your loops can cause the results to be wrong it will run them sequentially. Have a look at the compilation report to see if the compiler struggles with some loops.

However it might be a bit too prudent. If you know that parallelizing your loops is safe then you can tell the compiler with the *independent* clause of `acc loop directive`.

```
!$acc kernels
    !$acc loop independent
    do i=1, nelements
        ! A very safe loop
    enddo
!$acc end kernels
```

6.2 Running sequentially on the GPU? The `acc serial` compute construct

The GPUs are not very efficient to run sequential code however there 2 cases where it can be useful:

- Debugging a code
- Avoid some data transfers

The OpenACC standard gives you the `acc serial` directive for this purpose.

It is equivalent to having a parallel kernel which uses only one thread.

6.2.1 Syntax

```
!$acc serial <clauses>
    ! My sequential kernel
!$acc end serial
```

which is equivalent to:

```
!$acc parallel num_gangs(1) num_workers(1) vector_length(1)
    ! My sequential kernel
!$acc end parallel
```

6.3 Data region associated with compute constructs

You can manage your data transfers with data clauses:

clause	effect when entering the region	effect when leaving the region
create	**If the variable is not already present on the GPU**: allocate the memory needed on the GPU	**If the variable is not in another active data region**: free the memory on the GPU
copyin	**If the variable is not already present on the GPU**: allocate the memory and initialize the variable with the values it has on CPU	**If the variable is not in another active data region**: free the memory on the GPU
copyout	**If the variable is not already present on the GPU**: allocate the memory needed on the GPU	**If the variable is not in another active data region**: copy the values from the GPU to the CPU then free the memory on the GPU
copy	**If the variable is not already present on the GPU**: allocate the memory and initialize the variable with the values it has on CPU	**If the variable is not in another active data region**: copy the values from the GPU to the CPU then free the memory on the GPU
present	None	None

IMPORTANT: If your `acc kernels` is included in another data region then you have to be careful because you can not use the data clauses to update data. You need to use `acc update` for data already in another data region.

Requirements:

- Get started

CHAPTER

SEVEN

VARIABLES STATUS (PRIVATE OR SHARED)

7.1 Default status of scalar and arrays

The default status of variables depend on what they are (scalar or array) and the compute construct you use. Here is a summary:

	Scalar	Array
parallel	*gang* firstprivate	shared
kernels	shared	shared

7.2 Private variables

It is possible to make a variable private at *gang*, *worker* or *vector* level of parallelism if you use the `acc loop` clauses *private* or *firstprivate*. The variables will be private at the maximum level of parallelism the loop works.

Here are some examples:

7.2.1 Simple cases

A single loop with variables private at gang level:

```
real(kind=8) :: scalar
real(kind=8), dimension(:) :: array
!$acc parallel
    !$acc loop gang private(scalar, array)
    do i=0, array_size
        !! do some work on scalar and array
    enddo
!$acc end parallel
```

A single loop with variables private at worker level:

```
real(kind=8) :: scalar
real(kind=8), dimension(:) :: array
!$acc parallel
    !$acc loop gang worker private(scalar, array)
    do i=0, array_size
```

(continues on next page)

(continued from previous page)

```
        !! do some work on scalar and array
    enddo
!$acc end parallel
```

A single loop with variables private at vector level:

```
real(kind=8) :: scalar
real(kind=8), dimension(:) :: array
!$acc parallel
    !$acc loop gang vector private(scalar, array)
    do i=0, array_size
        !! do some work on scalar and array
    enddo
!$acc end parallel
```

7.2.2 A bit less straightforward

Nested loops:

```
real(kind=8) :: scalar1
real(kind=8) :: scalar2

scalar1 = 0
!$acc parallel
    !$acc loop gang reduction(+:scalar1) private(scalar2)
    do i=1, size_i
        scalar2 = 0
        !! scalar2 is private at gang level but shared at worker/vector level
        !$acc loop vector reduction(+:scalar2)
        do j=1, size_j
            scalar2 = scalar2 + ...
        enddo
        scalar1 = scalar1 + scalar2
    enddo
!$acc end parallel
```

7.3 Caution

You can make arrays private but in this case the memory requirements might be huge if you want them to be private at *worker* or *vector* level.

Requirements:

- Get started
- Variables_status
- Data management

CHAPTER

EIGHT

ADVANCED LOOP CONFIGURATION

Different levels of parallelism are generated by the *gang*, *worker* and *vector* clauses. The *loop* directive is responsible for sharing the parallelism across the different levels.

The degree of parallelism in a given level is determined by the numbers of gangs, workers and threads. These numbers are defined by the implementation. This default behavior depends on not only the target architecture but also on the portion of code on which the parallelism is applied. No modifications of this default behavior is recommended as it presents good optimization.

It is however possible to specify the numbers of gangs, workers and threads in the parallel construct with the *num_gangs*, *num_workers* and *vector_length* clauses. These clauses are allowed with the *parallel* and *kernel* construct. You might want to use these clauses in order to:

- debug (to restrict the execution on a single gang (without restrictions on the vectors as the *serial* clause will do), to vary the parallelism degree in order to expose a race condition …)
- limit the number of gang to lower the memory occupancy when you have to privatize arrays

8.1 Syntax

Clauses to specify the numbers of gangs, workers and vectors are *num_gangs*, *num_workers* and *vector_length*.

```
!$acc parallel num_gangs(3500) vector_length(128)
!$acc loop gang
do j = 1, size_j
    !$acc loop vector
    do i = 1, size_i
        ! A fabulous calculation
    enddo
enddo
!$acc end parallel

!$acc parallel loop gang num_gangs(size_j/2) vector_length(128)
do j = 1, size_j
    !$acc loop vector
    do i = 1, size_i
        ! A fabulous calculation
    enddo
enddo
!$acc end parallel
```

8.2 Restrictions

The restrictions described here are for NVIDIA architectures.

- The number of gang is limited to 2^{31}-1 (65535 if the compute capability is lower than 3.0).
- The product num_workers x vector_length can not be higher than 1024 (512 if the compute capability is lower than 2.0).
- To achieve performances, it is better to set the vector_length as a multiple of 32 (up to 1024).
- Using routines with a *vector* level of parallelization or higher sets the *vector_length* to 32 (compiler limitation).

This restrictions can vary with the architecture and it is advised to refer to the "Cuda C programming Guide" (Section G "Features and Technical Specifications") for future implementations.

8.3 Example

```
%%idrrun -a
!!  examples/Fortran/Loop_configuration_example.f90
program loop_configuration
    use ISO_FORTRAN_ENV, only : INT32, REAL64
    use openacc
    implicit none
    integer(kind=INT32), parameter        :: n = 500
    integer(kind=INT32), dimension(n,n,n) :: table
    integer(kind=INT32)                   :: ngangs, nworkers, nvectors
    integer(kind=INT32)                   :: i, j, k

    ngangs   = 450
    nworkers = 4
    nvectors = 16

    !$acc parallel loop gang num_gangs(ngangs) num_workers(nworkers) vector_
 ↪length(nvectors) copyout(table(:,:,:))
    do k = 1, n
        !$acc loop worker
        do j = 1, n
            !$acc loop vector
            do i = 1, n
                table(i,j,k) = i + j*1000 + k*1000*1000
            enddo
        enddo
    enddo

    print *, table(1,1,1), table(n,n,n)
end program loop_configuration
```

8.4 Exercise

A simple exercise can be to modify the value of the *num_gang* clause (and add a variation to the vector length) and then compare the execution time.

For a change, we will make an exercise that don't make sense physically. It can however come handy, especially if you try the practical work on HYDRO. In this exercise, you will have to:

- parallelize a few lines of codes
- be sure that it reproduces well the CPU behavior
- manually modify the number of gangs
- observe that the number of gangs will be limited by the system's size in this code

```
%%idrrun --cliopts "500"
!!  examples/Fortran/Loop_configuration_exercise.f90
program Loop_configuration
    use ISO_FORTRAN_ENV, only : INT32, REAL64, INT64
    implicit none
    integer(kind=INT32 ), parameter              :: system_size = 50000
    real   (kind=REAL64), dimension(system_size) :: array
    real   (kind=REAL64), dimension(system_size) :: table
    real   (kind=REAL64)                         :: sum_val, res, norm, time
    integer(kind=INT32 )                         :: i, j, length, ngangs, numarg
    character(len=:)    , allocatable            :: arg1

    numarg = command_argument_count()
    if (numarg .ne. 1) then
        write(0,*) "Error, you should provide an argument of integer kind to specify␣
 ↪the number of gangs that will be used"
        stop
    endif
    call get_command_argument(1,LENGTH=length)
    allocate(character(len=length) :: arg1)
    call get_command_argument(1,VALUE=arg1)
    read(arg1,'(i10)') ngangs

    norm    = 1.0_real64 / (int(system_size, INT64) * int(system_size, INT64))
    res     = 0.0_real64 ! to compare CPU and GPU quickly

    do j = 1, system_size
        sum_val = 0.0_real64
        do i = 1, system_size
            table(i) = (i+j) * norm
        enddo

        do i = 1, system_size
            sum_val  = sum_val + table(i)
        enddo
        array(j) = sum_val
        res = res + sum_val
    enddo

    print *, "result: ",res

!    do i = 1, system_size
```

(continues on next page)

(continued from previous page)

```
!        print *, i,array(i)
!    enddo

end program Loop_configuration
```

8.4.1 Solution

```
%%idrrun -a --cliopts "500"
!!  examples/Fortran/Loop_configuration_solution.f90
program Loop_configuration
    use ISO_FORTRAN_ENV, only : INT32, REAL64, INT64
    implicit none
    integer(kind=INT32 ), parameter              :: system_size = 50000
    real   (kind=REAL64), dimension(system_size) :: array
    real   (kind=REAL64), dimension(system_size) :: table
    real   (kind=REAL64)                         :: sum_val, res, norm, time
    integer(kind=INT32 )                         :: i, j, length, ngangs, numarg
    character(len=:)    , allocatable            :: arg1

    numarg = command_argument_count()
    if (numarg .ne. 1) then
        write(0,*) "Error, you should provide an argument of integer kind to specify␣
 ↪the number of gangs that will be used"
        stop
    endif
    call get_command_argument(1,LENGTH=length)
    allocate(character(len=length) :: arg1)
    call get_command_argument(1,VALUE=arg1)
    read(arg1,'(i10)') ngangs

    norm    = 1.0_real64 / (int(system_size, INT64) * int(system_size, INT64))
    res     = 0.0_real64 ! to compare CPU and GPU quickly

    !$acc parallel num_gangs(ngangs) copyout(array(:)) private(table(:))
    !$acc loop gang reduction(+:res)
    do j = 1, system_size
        sum_val = 0.0_real64
        !$acc loop vector
        do i = 1, system_size
            table(i) = (i+j) * norm
        enddo

        !$acc loop vector reduction(+:sum_val)
        do i = 1, system_size
            sum_val  = sum_val + table(i)
        enddo
        array(j) = sum_val
        res = res + sum_val
    enddo
    !$acc end parallel

    print *, "result: ",res

!    do i = 1, system_size
```

(continues on next page)

(continued from previous page)

```
!        print *, i,array(i)
!    enddo

end program Loop_configuration
```

Requirements:

- Get started
- Data management
- Loop configuration

CHAPTER

NINE

USING OPENACC IN MODULAR PROGRAMMING

Most modern codes use modular programming to make the readability and maintenance easier. You will have to deal with it inside your own code and be careful to make all functions accessible where you need.

If you call a function inside a kernel, then you need to tell the compiler to create a version for the GPU. With OpenACC you have to use the `acc routine` directive for this purpose.

With Fortran you will have to take care of the variables that are declared inside modules and use `acc declare create`.

9.1 acc routine <max_level_of_parallelism>

This directive is used to tell the compiler to create a function for the GPU as well as for the CPU. Since the function is available for the GPU you will be able to call it inside a kernel.

When you use this directive you sign a contract with the compiler (normally no soul selling, but check it twice!) and promise that the function will be called inside a section of code for which work sharing at this level is not yet activated. The clauses available are:

- gang
- worker
- vector
- seq: the function is executed sequentially by one GPU thread

The directive is added before the function definition or declaration:

```
subroutine mean_value(array, array_size)
!$acc routine seq
    ! compute the mean value
end subroutine mean_value
```

9.1.1 Wrong examples

Since it might be a bit tricky here are some wrong examples with an explanation: This example is wrong because `acc parallel loop worker` activates work sharing at the *worker* level of parallelism. The `acc routine worker` indicates that the function can activate *worker* and *vector* level of parallelism and you cannot activate twice the same level.

```
subroutine my_worker_subroutine
    !$acc routine worker
    ...
end subroutine

...
!$acc parallel loop worker
do i=1, sys_size
    call my_worker_subroutine
done
```

For a similar reason this is forbidden:

```
subroutine my_gang_subroutine
    !$acc routine gang
    ...
end subroutine

...
!$acc parallel
!$acc loop gang
do i=1, sys_size
    call my_gang_subroutine
enddo

!$acc end parallel
```

This example is wrong since it breaks the promise you make to the compiler. A vector routine cannot have loops at the *gang* and *worker* levels of parallelism.

```
subroutine my_wrong_subroutine
!$acc routine vector
    ...
    !$acc loop gang worker
    do i=0, sys_size
        // some loop stuff
    enddo
end subroutine
```

9.2 Named `acc routine(name) <max_level_of_parallelism>`

You can declare the `acc routine` directive anywhere a function prototype is allowed within the specification part of the routine.

```
module utils
    use openacc
    interface beautiful
```

(continues on next page)

(continued from previous page)

```
        module procedure beautiful_name
        !$acc routine(beautiful_name) vector
    end interface
    contains
       subroutine beautiful_name(name)
       ! Do something
       end subroutine beautiful_name

end module utils

program another_brick
    use utils
    ...
    !$acc routine(beautiful_name) vector
    ...
    call beautiful(name)
    ! integers are beauty
end program another_brick
```

9.3 Directives inside an `acc routine`

Routines you declare with `acc routine` shall not contain directives to create kernels (*parallel*, *serial*, *kernels*). You have to consider the content of the function already inside a kernel.

```
subroutine init(array, array_size)
!$acc routine vector
integer, dimension(:), intent(inout) :: array
integer, intent(in)                  :: array_size
integer                              :: i
    !$acc loop
    do i = 1, array_size
        array(i) = i
    enddo
end subroutine init
```

9.4 Exercise

In this exercise, you have to compute the mean value of each row of a matrix. The value is computed by a function `mean_value` working on one row at a time. This function can use parallelism.

To have correct results, you will need to make the variable `local_mean` private for each thread. To achieve this you have to use the *private(vars, …)* clause of the `acc loop` directive.

```
%%idrrun -a
!!  examples/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
```

(continues on next page)

(continued from previous page)

```
            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(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)
            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 =  500000
    ny =    3000
    allocate(table(nx,ny), mean_values(ny))
    table(:,:) = 0.0_real64
    call rand_init(table(1,:),ny)
```

(continues on next page)

(continued from previous page)

```
    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
```

9.4.1 Solution

```
%%idrrun -a
!!  examples/Fortran/Modular_programming_mean_value_solution.f90
module calcul
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    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)
        !$acc routine worker
            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

            !$acc loop seq
            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)
```

(continues on next page)

(continued from previous page)

```
                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)
        !$acc routine vector
            real   (kind=REAL64), dimension(n,1), intent(inout) :: t
            integer(kind=INT32 ), intent(in)                    :: n
            real   (kind=REAL64)                                :: mean_value
            integer(kind=INT32 )                                :: i
            mean_value = 0.0_real64
            !$acc loop reduction(+:mean_value)
            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 =  500000
    ny =    3000
    allocate(table(nx,ny), mean_values(ny))
    table(:,:) = 0.0_real64
    call rand_init(table(1,:),ny)
    !$acc enter data copyin(table(:,:))
    cell_size = 32
    !$acc parallel loop
    do i = 2, ny
        call iterate(table(:,i), nx, cell_size)
    enddo

    !$acc parallel loop gang present(table(:,:)) copyout(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

    !$acc exit data delete(table)
```

(continues on next page)

(continued from previous page)

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

CHAPTER

TEN

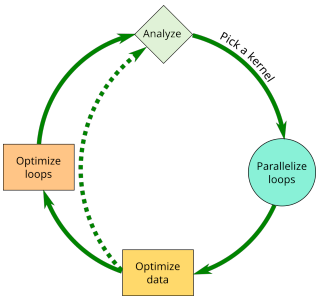
PROFILING YOUR CODE TO FIND WHAT TO OFFLOAD

10.1 Development cycle

When you port your code with OpenACC you have to find the hotspots which can benefit from offloading.

That's the first part of the development cycle (Analyze). This part should be done with a profiler since it helps a lot to find the hotspots.

Once you have found the most time consuming part, you can add the OpenACC directives. Then you find the next hotspot, manage memory transfers and so on.



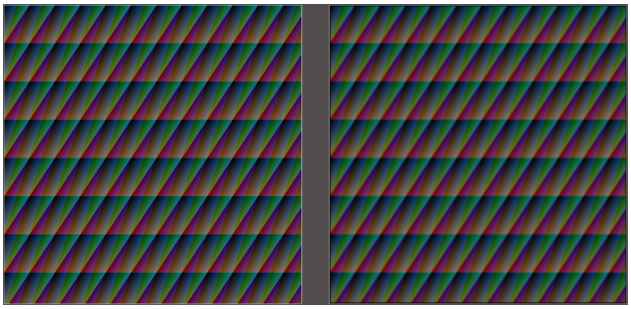
10.2 Quick description of the code

The code used as an example in this chapter generates a picture and then applies a blurring filter.

Each pixel of the blurred picture has a color that is the weighted average of its corresponding pixel on the original picture and its 24 neighbors.

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

It will generate 2 pictures that look like:



10.3 Profiling CPU code

The first task you have to achieve when porting your code with OpenACC is to find the most demanding loops in your CPU code. You can use your favorite profiling tool:

- gprof
- ARM MAP
- Nsight Systems

Here we will use the Nsight Systems.

The first step is to generate the executable file. Run the following cell which will just compile the code inside the blur.c and create 2 files:

- blur.f90 (the content of the cell)
- blur.f90.exe (the executable)

This lets us introduce the command to run an already existing file `%idrrunfile filename`.

```
%idrrunfile --profile ../../examples/Fortran/blur.f90
```

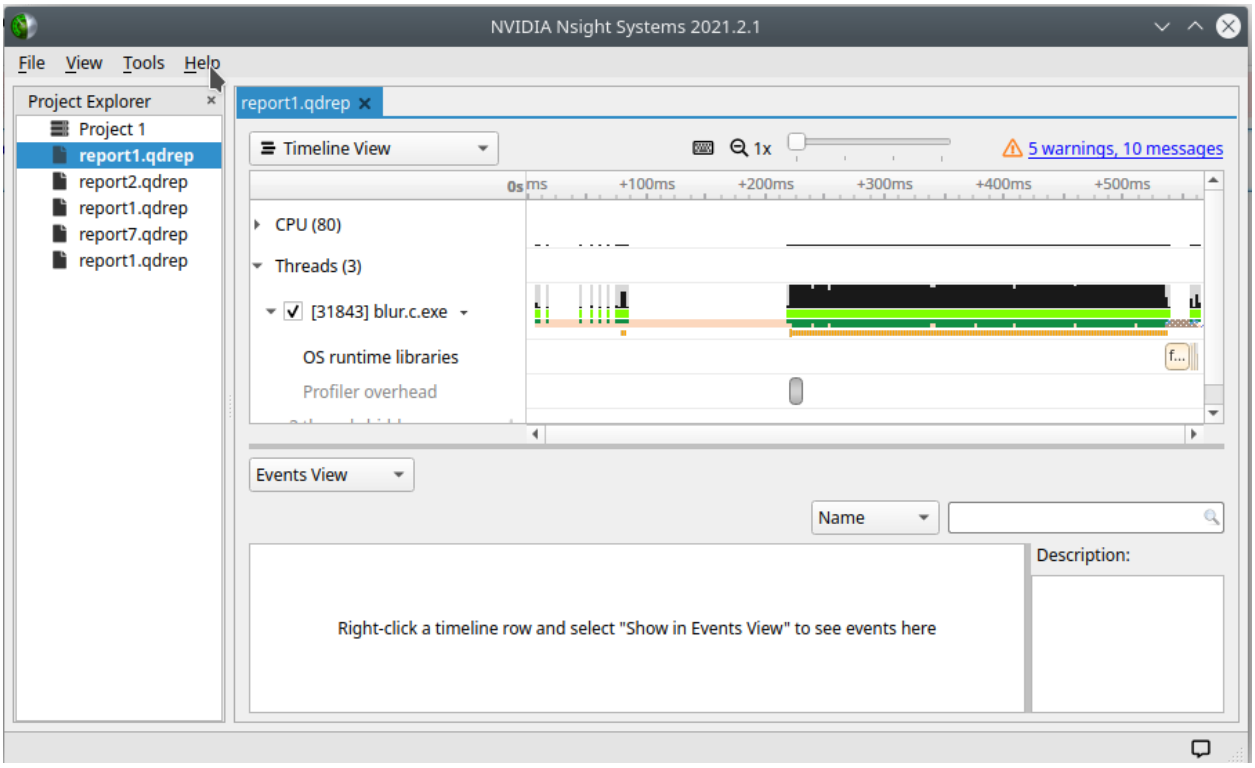
Now you can run the UI by executing the following cell and choosing the right reportxx.qdrep file (here it should be report1.qdrep).

Please also write down the time taken (should be around 0.3 s on 1 Cascade Lake core).

```
%%bash
module load nvidia-nsight-systems/2021.2.1
nsys-ui $PWD/report1.qdrep
```

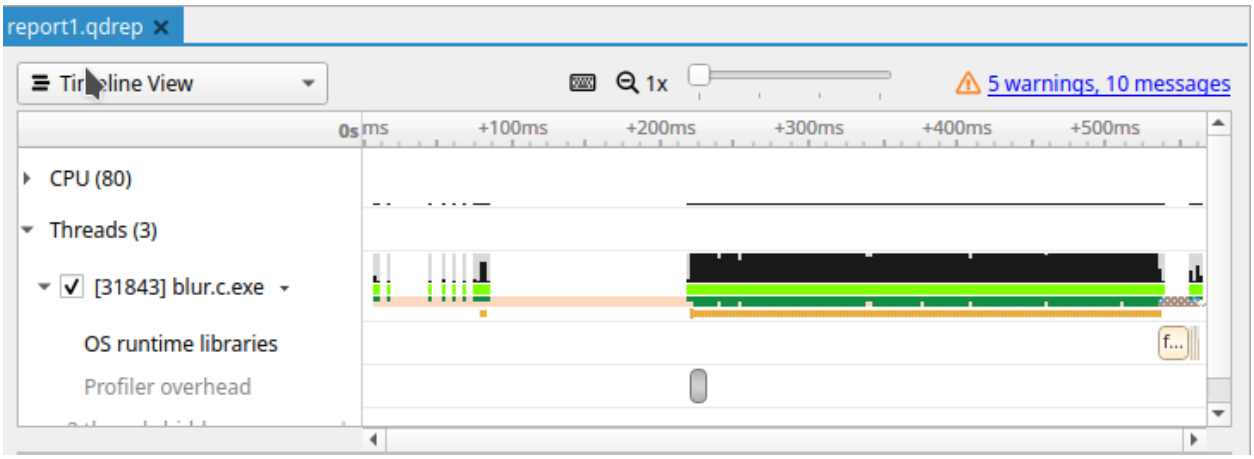
10.4 The graphical profiler

The Graphical user interface for the Nsight Systems (version 2021.2.1) is the following:



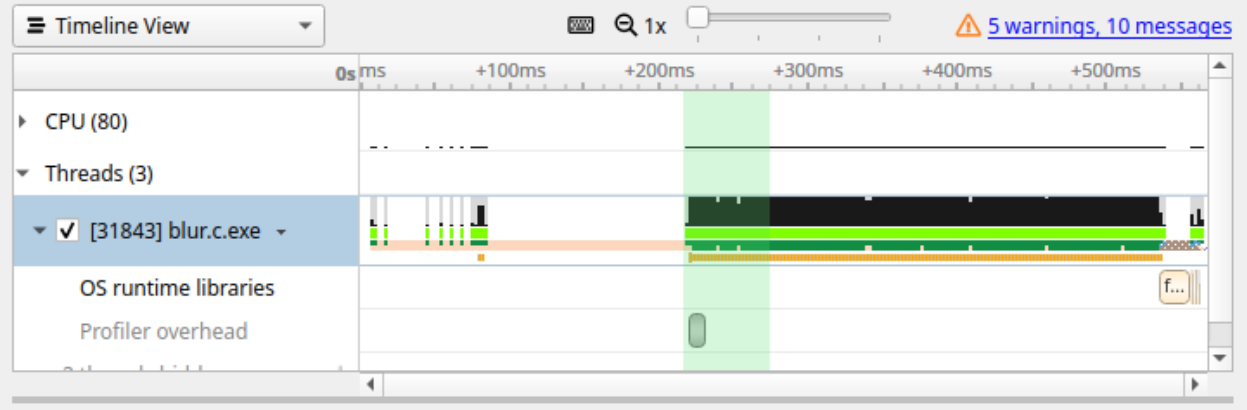
10.4.1 The timeline

Maybe the most important part is the timeline:

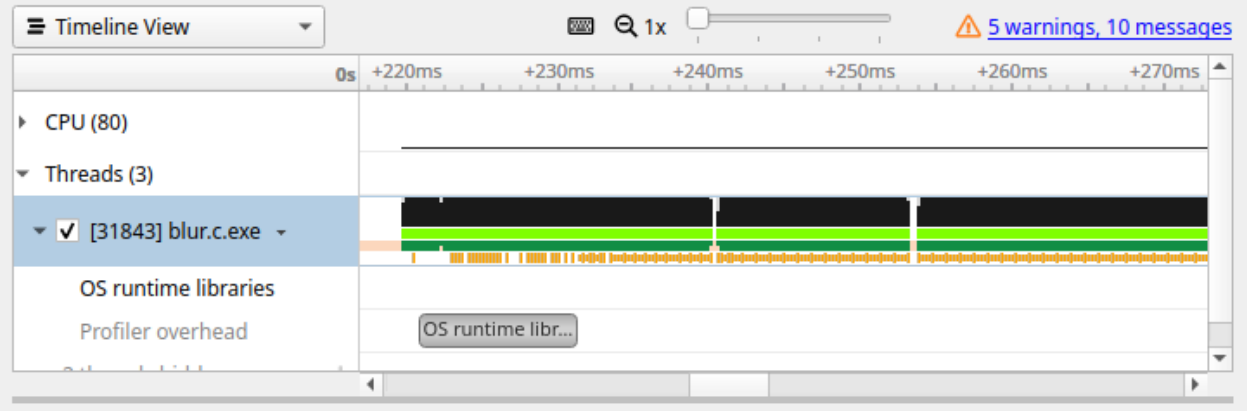


It has the information about what happened during execution of your code with a timeline view.

You can select a portion of the timeline by holding the left button of the mouse (when the mouse is set up for right-handed people) and dragging the cursor.

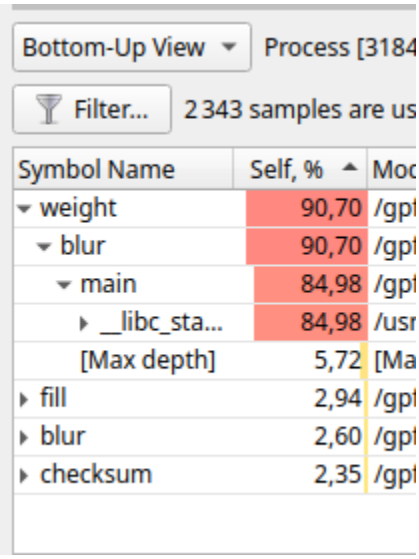


and zoom (maj+z or right-click “Zoom into selection”):



10.4.2 Profile

To see a summary of the time taken by each function you have to select “Bottom-up View” in the part below the timeline. You can unroll the functions to have a complete view.



Analysis

So here we see that most of the time is spent into the weight function. You can open the blur.c file to see what this function does.

The work is done by this double loop which computes the value of the blurred pixel

```
do i = 0, 4
    do j = 0, 4
        pix = pix + pic((x+i-2)*3*cols+y*3+l-2+) * coefs(i,j)
    enddo
enddo
```

Parallelizing this loop will not give us the optimal performance. Why?

The iteration space is 25. So we will launch a lot of kernels (number of pixels in the picture) with a very small number of threads for a GPU.

As a reminder NVIDIA V100 can run up to 5,120 threads at the same time.

You also have to remember that launching a kernel has an overhead.

So the advice is:

- **Give work to the GPU** by having large kernels with a lot of computation
- **Avoid launching too many kernels** to reduce overhead

We have to find another way to parallelize this code! The `weight` function is called by `blur` which is a loop over the pixels.

As an exercise, you can add the directives to offload `blur`. Once you are done you can run the profiler again.

```
%idrrunfile -a ../../examples/Fortran/blur.f90
```

10.5 Profiling GPU code: other tools

Other tools available for profiling GPU codes include:

- ARM MAP
- Environment variables NVCOMPILER_ACC_TIME and NVCOMPILER_ACC_NOTIFY

It is possible to activate profiling by the runtime using two environment variables, NVCOMPILER_ACC_TIME and NVCOMPILER_ACC_NOTIFY. It provides a fast and easy way of profiling without a need of a GUI.

Warning: disable NVCOMPILER_ACC_TIME (`export NVCOMPILER_ACC_TIME 0`) if using another profiler.

10.5.1 NVCOMPILER_ACC_NOTIFY

Additional profiling information can be collected by using the variable NVCOMPILER_ACC_NOTIFY. The values below correspond to activation of profiling data collection depending on a type of GPU operation.

- 1: kernel launches
- 2: data transfers
- 4: region entry/exit
- 8: wait operations or synchronizations
- 16: device memory allocates and deallocates

For example, in order to obtain output including the kernel executions and data transfers, one needs to set NVCOMPILER_ACC_NOTIFY to 3.

Requirements:

- Get started
- Data Management

CHAPTER

ELEVEN

MULTI GPU PROGRAMMING WITH OPENACC

11.1 Disclaimer

This part requires that you have a basic knowledge of OpenMP and/or MPI.

11.2 Introduction

If you wish to have your code run on multiple GPUs, several strategies are available. The most simple ones are to create either several threads or MPI tasks, each one addressing one GPU.

11.3 API description

For this part, the following API functions are needed:

- *acc_get_device_type()*: retrieve the type of accelerator available on the host
- *acc_get_num_device(device_type)*: retrieve the number of accelerators of the given type
- *acc_set_device_num(id, device_type)*: set the id of the device of the given type to use

11.4 MPI strategy

In this strategy, you will follow a classical MPI procedure where several tasks are executed. We will use either the OpenACC directive or API to make each task use 1 GPU.

Have a look at the examples/C/init_openacc.h

```
%%idrrun -m 4 -a --gpus 2 --option "-cpp"
!!  examples/Fortran/MultiGPU_mpi_example.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program multigpu
    use ISO_FORTRAN_ENV, only : INT32
    use mpi
    use openacc
    implicit none
    integer(kind=INT32), dimension(100) :: a
    integer                             :: comm_size, my_rank, code, i
    integer                             :: num_gpus, my_gpu
```

(continues on next page)

(continued from previous page)

```
    integer(kind=acc_device_kind)       :: device_type

    ! Useful for OpenMPI and GPU DIRECT
    call initialisation_openacc()

    ! MPI stuff
    call MPI_Init(code)
    call MPI_Comm_size(MPI_COMM_WORLD, comm_size, code)
    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, code)

    ! 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)
    call acc_set_device_num(my_gpu, device_type)
    my_gpu   = acc_get_device_num(device_type)
    ! Alternatively you can set the GPU number with #pragma acc set device_num(my_gpu)

    !$acc parallel loop
    do i = 1, 100
        a(i) = i
    enddo
    #endif
    write(0,"(a13,i2,a17,i2,a8,i2,a10,i2)") "Here is rank ",my_rank,": I am using GPU
↪",my_gpu, &
                                            " of type ",device_type,". a(42) = ",a(42)
    call MPI_Finalize(code)

    contains
        #ifdef _OPENACC
        subroutine initialisation_openacc
        use openacc

        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())
           info%current_devices = local_rank
```

(continues on next page)

(continued from previous page)

```
       else
           print *, "Error : impossible to determine the local rank of the process"
           stop 1
       endif
       end subroutine initialisation_openacc
       #endif

end program multigpu
```

11.4.1 Remarks

It is possible to have several tasks accessing the same GPU. It can be useful if one task is not enough to keep the GPU busy along the computation.

If you use NVIDIA GPU, you should have a look at the Multi Process Service.

11.5 Multithreading strategy

Another way to use several GPUs is with multiple threads. Each thread will use one GPU and several threads can share 1 GPU.

```
%%idrrun -a -t -g 4 --threads 4 --option "-cpp"
!!  examples/Fortran/MultiGPU_openmp_example.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program MultiGPU_openmp
    use ISO_FORTRAN_ENV, only : INT32
    use OMP_LIB
    use openacc
    implicit none
    integer(kind=INT32)           :: my_rank
    integer                       :: num_gpus, my_gpu
    integer(kind=acc_device_kind) :: device_type

    !$omp parallel private(my_rank, my_gpu, device_type)
        my_rank = omp_get_thread_num()
        ! OpenACC Stuff
        #ifdef _OPENACC
        device_type = acc_get_device_type()
        num_gpus = acc_get_num_devices(device_type)
        my_gpu   = mod(my_rank,num_gpus)
        call acc_set_device_num(my_gpu, device_type)
        ! We check what GPU is really in use
        my_gpu = acc_get_device_num(device_type)
        ! Alternatively you can set the GPU number with #pragma acc set device_num(my_
 ↪gpu)
        write(0,"(a14,i2,a17,i2,a9,i2)") "Here is thread ",my_rank," : I am using GPU
 ↪",my_gpu," of type ",device_type
        #endif
    !$omp end parallel
end program MultiGPU_openmp
```

11.6 Exercise

1. Copy one cell from a previous notebook with a sequential code
2. Modify the code to use several GPUs
3. Check the correctness of the figure

11.7 GPU to GPU data transfers

If you have several GPUs on your machine they are likely interconnected. For NVIDIA GPUs, there are 2 flavors of connections: either PCI express or NVlink. NVLink is a fast interconnect between GPUs. Be careful since it might not be available on your machine. The main difference between the two connections is the bandwidth for CPU/GPU transfers, which is higher for NVlink.

The GPUDirect feature of CUDA-aware MPI libraries allows direct data transfers between GPUs without an intermediate copy to the CPU memory. If you have access to an MPI CUDA-aware implementation with GPUDirect support, you should definitely adapt your code to benefit from this feature.

For information, during this training course we are using OpenMPI which is CUDA-aware. You can find a list of CUDA-aware implementation on NVIDIA website.

By default, the data transfers between GPUs are not direct. The scheme is the following:

1. The **origin** task generates a Device to Host data transfer
2. The **origin** task sends the data to the **destination** task.
3. The **destination** task generates a Host to Device data transfer

Here we can see that 2 transfers between Host and Device are necessary. This is costly and should be avoided if possible.

11.7.1 `acc host_data` directive

To be able to transfer data directly between GPUs, we introduce the **host_data** directive.

```
!$acc host_data use_device(array)
...
!$acc end host_data
```

This directive tells the compiler to assign the address of the variable to its value on the device. You can then use the pointer with your MPI calls. **You have to call the MPI functions on the host.**

Here is a example of a code using GPU to GPU direct transfer.

```
integer, parameter :: system_size = 1000;
integer, dimension(system_size) :: array
!$acc enter_data create(array(1:1000))
! Perform some stuff on the GPU
!$acc parallel present(array(1:1000))
...
!$acc end parallel
! Transfer the data between GPUs
if (my_rank .eq. origin ) then
    !$acc host_data use_device(array)
    MPI_Send(array, size, MPI_INT, destination, tag, MPI_COMM_WORLD, code)
```

(continues on next page)

(continued from previous page)

```
    !$acc end host_data
endif
if (my_rank .eq. destination) then
    !$acc host_data use_device(array)
    MPI_Recv(array, size, MPI_INT, origin, tag, MPI_COMM_WORLD, status, code)
    !$acc end host_data
endif
```

11.7.2 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

```
%%idrrun -a -m 4 -g 4
!!  examples/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
```

Solution

```
%%idrrun -a -m 4 -g 4
!!  examples/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))
    !$acc enter data create(send_buffer(1:system_size), receive_buffer(1: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)
    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
```

(continues on next page)

(continued from previous page)

```
                start = MPI_Wtime()
                !$acc host_data use_device(send_buffer)
                do k = 1, reps
                    call MPI_Send(send_buffer,system_size, MPI_DOUBLE, i, 0, MPI_COMM_
↪WORLD, code)
                enddo
                !$acc end host_data
            endif
            if ( (my_rank .eq. i) .and. (i .ne. j) ) then
                !$acc host_data use_device(receive_buffer)
                do k = 1, reps
                    call MPI_Recv(receive_buffer, system_size, MPI_DOUBLE, j, 0, MPI_
↪COMM_WORLD, mpi_stat, code)
                enddo
                !$acc end host_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
    !$acc exit data 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
        ! 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
```

(continues on next page)

(continued from previous page)

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

end program MultiGPU_solution
```

Requirements:

- Get started
- Data management
- Multi GPU

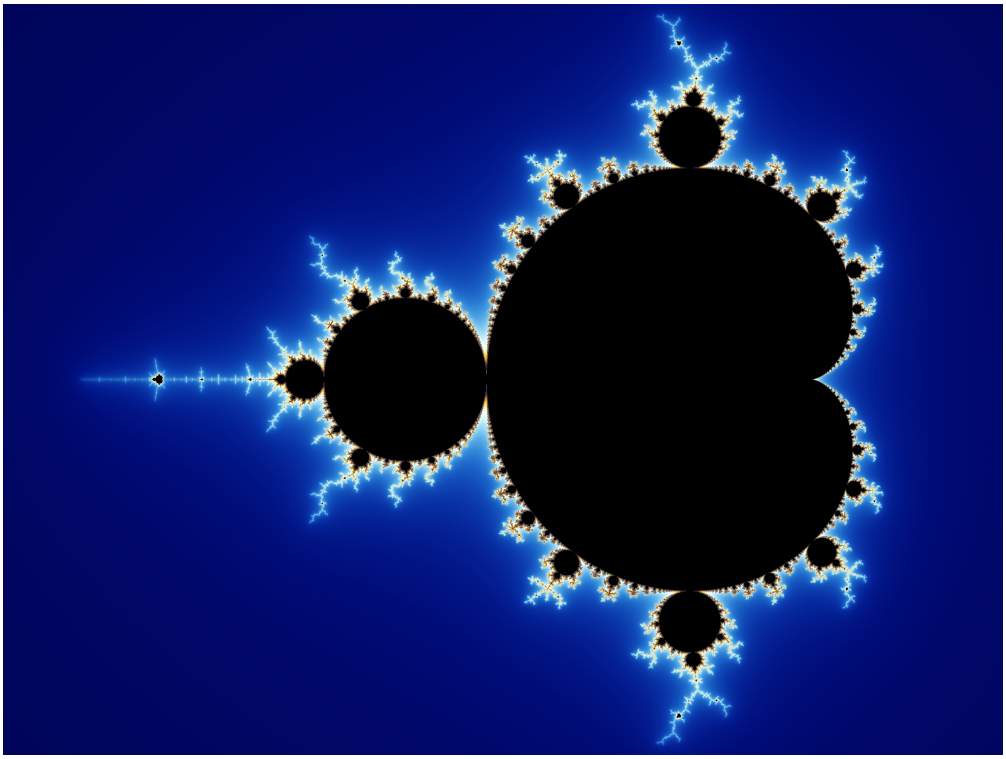
CHAPTER

TWELVE

GENERATE MANDELBROT SET

12.1 Introduction

The Mandelbrot set is the set of complex numbers c for which the function \begin{equation} f_c(z) = z^2+c \end{equation} does not diverge when iterated from z = 0. Wikipedia



By Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, Link

In this hands-on you will generate a picture with the Mandelbrot set using a Multi-GPU version of the code. We use the MPI language to split the work between the GPUs.

12.2 What to do

Add the directives to use several GPUs. Here we do **not** need the GPUs to communicate. Be careful to allocate the memory only for the part of the picture treated by the GPU and not the complete memory.

You can have a look at the file init_openacc.h. It gives the details to associate a rank with a GPU.

The default coordinates show the well known representation of the set. If you want to play around have a look at this webpage giving interesting areas of the set on which you can "zoom".

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.

```
%%idrrun -a -m 4 -g 4
!!  examples/Fortran/mandelbrot_mpi_exercise.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program mandelbrot_mpi
    use MPI
    #ifdef _OPENACC
    use openacc
    #endif
    implicit none
    type accel_info
        integer :: current_devices
        integer :: total_devices
    end type accel_info
    type(accel_info)              :: gpu_info
    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

    #ifdef _OPENACC
    ! add initialisation here as subroutine call or function assignation to a␣
↪type(accel_info) variable
    #endif

    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
```

(continues on next page)

(continued from previous page)

```
    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
    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
        #ifdef _OPENACC
        ! implement function or subroutine here :
        !           function :
        !type(accel_info) function initialisation_openacc
        !end function initialisation_openacc
        !
        !           subroutine :
        !subroutine initialisation_openacc()
        !end subroutine initialisation_openacc
        #endif
```

(continues on next page)

(continued from previous page)

```
        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
```

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

12.3 Solution

```
%%idrrun -a -m 4 -g 4
!!  examples/Fortran/mandelbrot_mpi_solution.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program mandelbrot_mpi
    use MPI
    #ifdef _OPENACC
    use openacc
    #endif
    implicit none
    type accel_info
        integer :: current_devices
        integer :: total_devices
    end type accel_info
    type(accel_info)              :: gpu_info
    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
```

(continues on next page)

(continued from previous page)

```
    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

    #ifdef _OPENACC
    gpu_info = initialisation_openacc()
    #endif

    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
    num_elements = local_height * width

    write(unit=*,fmt="(a9,i3,a18,i8,a3,i8,a5,i10,a9)") "I am rank",rank, &
```

(continues on next page)

(continued from previous page)

```
    " and my range is [",first," ,",last,"[ ie ",num_elements," elements"

    allocate(picture(first*width:last*width))
    !$acc data copyout(picture(first*width:last*width))
    !$acc parallel loop private(x,y)
    do i=first,last-1
        !$acc loop
        do j=0,width-1
            x =  min_re + j * step_w * (max_re - min_re)
            y =  min_im + i * step_h * (max_im - min_im)
            picture(i*width+j) = mandelbrot_iterations(x,y)
        enddo
    enddo
    !$acc end data
    call output()
    deallocate(picture)

    call mpi_finalize(code)

    contains
        #ifdef _OPENACC
        type(accel_info) function initialisation_openacc
           ! use openacc
            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)
            initialisation_openacc%total_devices   = acc_get_num_devices(acc_get_
↪device_type())
            initialisation_openacc%current_devices = -1
            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())
                initialisation_openacc%current_devices = local_rank
            else
                print *, "Error : impossible to determine the local rank of the␣
↪process"
                stop 1
            endif
        end function initialisation_openacc
        #endif
        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)
```

(continues on next page)

(continued from previous page)

```
            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)
            !$acc routine seq
            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)
```

Requirements:

- Get started
- Data management
- Multi GPU

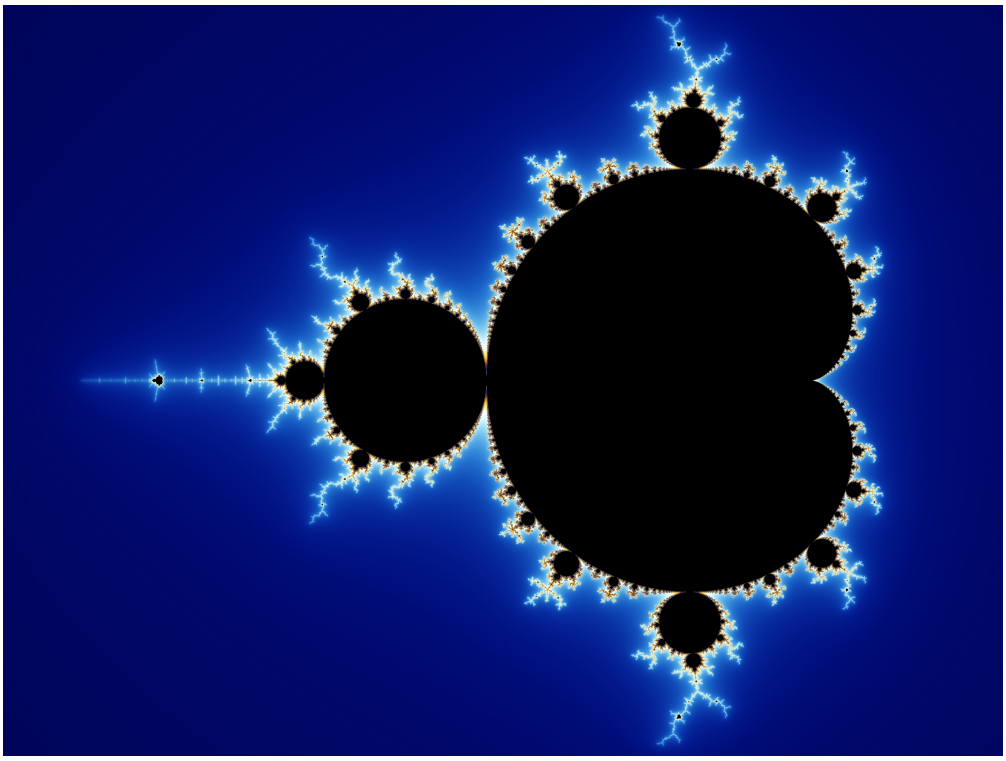
CHAPTER

THIRTEEN

GENERATE MANDELBROT SET

13.1 Introduction

The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from z = 0. Wikipedia



By Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, Link

In this hands-on you will generate a picture with the Mandelbrot set using a Multi-GPU version of the code. We use the OpenMP language to split the work between the GPUs.

13.2 What to do

Add the directives to use several GPUs. Here we do **not** need the GPU to communicate. Be careful to allocate the memory only for the part of the picture treated by the GPU and not the complete memory.

The default coordinates show the well known representation of the set. If you want to play around have a look at this webpage giving interesting areas of the set on which you can "zoom".

```
%%idrrun --cliopts "8000 4000" -t -g 4 --threads 4 --get mandel.gray
!!  examples/Fortran/mandelbrot_openmp_exercise.f90
! you should add ` --option "-cpp" ` as argument to the idrrun command
program mandelbrot
    #ifdef _OPENMP
    use OMP_LIB
    #endif
    #ifdef _OPENACC
    use openacc
    #endif
    implicit none
    integer, parameter              :: max_iter = 127
    real, parameter                 :: min_re = -2.0
    real, parameter                 :: max_re = 1.0
    real, parameter                 :: min_im = -1.0
    real, parameter                 :: max_im = 1.0
    integer                         :: rank
    integer                         :: first,last,width,height
    integer                         :: num_elements,num_threads,num_gpu
    real                            :: step_w,step_h
    character(len=:), allocatable   :: arg1,arg2
    integer (kind=1), allocatable   :: picture(:)
    real                            :: x,y
    integer                         :: len1,len2
    integer                         :: i,j,first_elem,last_elem
    #ifdef _OPENACC
    integer(acc_device_kind)        :: type_d
    #endif

    call GET_COMMAND_ARGUMENT(1,LENGTH=len1)
    allocate(character(len=len1) :: arg1)
    call GET_COMMAND_ARGUMENT(1,VALUE=arg1)
    call GET_COMMAND_ARGUMENT(2,LENGTH=len2)
    allocate(character(len=len2) :: arg2)
    call GET_COMMAND_ARGUMENT(2,VALUE=arg2)
    read(arg1,'(i10)') width
    read(arg2,'(i10)') height
    step_w = 1.0 / width
    step_h = 1.0 / height

    deallocate(arg1,arg2)

    allocate(picture(0:width*height-1))


    !$OMP parallel private(first, last, rank, num_threads, num_elements, num_gpu, x,␣
 ↪y, i, j, last_elem, first_elem) shared(picture) firstprivate(height, width, step_h,␣
 ↪step_w) default(none)
```

(continues on next page)

(continued from previous page)

```
    #ifdef _OPENMP
     rank         = OMP_GET_THREAD_NUM()
     num_threads  = OMP_GET_NUM_THREADS()
     first        =  rank * (height/num_threads)
     last         = (rank + 1) * (height/num_threads) -1
     num_elements = width*height/num_threads
    #endif

    print *, "Using OpenMP"

    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"

    #ifdef _OPENACC
     type_d  = acc_get_device_type()
     num_gpu = acc_get_num_devices(type_d)
     call acc_set_device_num(mod(rank,num_gpu), type_d)
     first_elem   = first*width
     last_elem    = first_elem + num_elements-1
     print *, "I am rank",rank,". I am using GPU #", &
              acc_get_device_num(type_d),first_elem,last_elem
    #endif

    do i = first, last
         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 parallel


    open(unit=10, FILE="mandel.gray", ACCESS="stream", FORM="unformatted")
        write(unit=10,pos=1) picture
    close(10)

    deallocate(picture)
    contains
    integer(kind=1) function mandelbrot_iterations(x,y)
        !$acc routine seq
        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
```

```
from idrcomp import show_gray
show_gray("mandel.gray", 8000, 4000)
```

13.3 Solution

```
%%idrrun --cliopts "8000 4000" -t -a -g 4
!!  examples/Fortran/mandelbrot_openmp_solution.f90
! you should add ` --option "-cpp -Minline" ` as argument to the idrrun command
program mandelbrot
    #ifdef _OPENMP
    use OMP_LIB
    #endif
    #ifdef _OPENACC
    use openacc
    #endif
    implicit none
    integer, parameter               :: max_iter = 127
    !$acc declare create(max_iter)
    real, parameter                  :: min_re = -2.0
    real, parameter                  :: max_re = 1.0
    real, parameter                  :: min_im = -1.0
    real, parameter                  :: max_im = 1.0
    !$acc declare create(min_re,max_re,min_im,max_im)
    integer                          :: rank
    integer                          :: first,last,width,height
    integer                          :: num_elements,num_threads,num_gpu
    real                             :: step_w,step_h
    character(len=:), allocatable    :: arg1,arg2
    integer (kind=1), allocatable    :: picture(:)
    real                             :: x,y
    integer                          :: len1,len2
    integer                          :: i,j,first_elem,last_elem
    #ifdef _OPENACC
    integer(acc_device_kind)         :: type_d
    #endif

    call GET_COMMAND_ARGUMENT(1,LENGTH=len1)
    allocate(character(len=len1) :: arg1)
    call GET_COMMAND_ARGUMENT(1,VALUE=arg1)
    call GET_COMMAND_ARGUMENT(2,LENGTH=len2)
    allocate(character(len=len2) :: arg2)
    call GET_COMMAND_ARGUMENT(2,VALUE=arg2)
    read(arg1,'(i10)') width
    read(arg2,'(i10)') height
    step_w = 1.0 / width
    step_h = 1.0 / height

    deallocate(arg1,arg2)

    allocate(picture(0:width*height-1))

    !$OMP parallel private(first, last, rank, num_threads, num_elements, num_gpu,␣
 ↪type_d, x, y, i, j, last_elem, first_elem) shared(picture) firstprivate(height,␣
 ↪width, step_h, step_w) default(none)
```

(continues on next page)

(continued from previous page)

```
#ifdef _OPENMP
 rank         = OMP_GET_THREAD_NUM()
 num_threads  = OMP_GET_NUM_THREADS()
 first        =  rank * (height/num_threads)
 last         = (rank + 1) * (height/num_threads) -1
 num_elements = width*height/num_threads
#endif

print *, "Using OpenMP"

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"

#ifdef _OPENACC
 type_d  = acc_get_device_type()
 num_gpu = acc_get_num_devices(type_d)
 call acc_set_device_num(mod(rank,num_gpu), type_d)
 first_elem   = first*width
 last_elem    = first_elem + num_elements-1
 print *, "I am rank",rank,". I am using GPU #", &
          acc_get_device_num(type_d),first_elem,last_elem
#endif

!$acc data copyout(picture(first_elem:last_elem))

!$acc parallel loop present(picture(first_elem:last_elem)) private(x,y)
do i = first, last
    !$acc loop
     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
!$acc end data

!$OMP end parallel

open(unit=10, FILE="mandel.gray", ACCESS="stream", FORM="unformatted")
    write(unit=10,pos=1) picture
close(10)

deallocate(picture)
contains
integer(kind=1) function mandelbrot_iterations(x,y)
    !$acc routine seq
    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
```

(continues on next page)

(continued from previous page)

```
            z2 = 2.0*z1_old*z2_old      + y
            mandelbrot_iterations = mandelbrot_iterations + 1
        enddo
    end function mandelbrot_iterations
end program mandelbrot
```

```
from idrcomp import show_gray
show_gray("mandel.gray", 8000, 4000)
```

Part III

Day 3

Requirements:

- Get started
- Atomic operations
- Manual building
- Data management

CHAPTER

FOURTEEN

PERFORMING SEVERAL TASKS AT THE SAME TIME ON THE GPU

This part describes how to overlap several kernels on the GPU and/or how to overlap kernels with data transfers. This feature is called asynchronism and will give you the possibility to get better performance when it is possible to implement it.

On the GPU you can have several execution threads (called *streams* or *activity queue*) running at the same time independently. A *stream* can be viewed as a pipeline that you feed with kernels and data transfers that have to be executed in order.

So as a developer you can decide to activate several streams if your code is able to withstand them. OpenACC gives you the possibility to manage streams with the tools:

- *async* clause
- *wait* clause or directive

By default, only one stream is created.

14.1 *async* clause

Some directives accept the clause *async* to run on another stream than the default one. You can specify an integer (which can be a variable) to have several streams concurrently.

If you omit the optional integer then a "default" extra stream is used.

The directives which accept *async* are:

- the compute constructs: `acc parallel`, `acc kernels`, `acc serial`
- the unstructured data directives: `acc enter data`, `acc exit data`, `acc update`
- the `acc wait` directive

For example we can create 2 streams to allow data transfers and kernel overlap.

```
integer :: stream1=1
integer :: stream2=2

!$acc enter data copyin(array(:)) async(stream1)
! Some stuff
!$acc parallel async(stream2)
    ! A wonderful kernel
!$acc end parallel
```

14.2 *wait* clause

Running fast is important but having correct results is surely more important.

If you have a kernel that needs the result of another kernel or that a data transfer is complete then you have to wait for the operations to finalize. You can add the *wait* clause (with an optional integer) to the directives:

- the compute constructs: `acc parallel`, `acc kernels`, `acc serial`
- the unstructured data directives: `acc enter data`, `acc exit data`, `acc update`

This example implements 2 streams but this time the kernel needs the data transfer on stream1 to complete before being executed.

```
integer stream1=1
integer stream2=2
!$acc enter data copyin(array(:)) async(stream1)
! Some stuff

!$acc parallel async(stream2) wait(stream1)
    ! A wonderful kernel
!$acc end parallel
```

Furthermore you can wait for several streams to complete by giving a comma-separated list of integers as clause arguments

This example implements 2 streams but this time the kernel needs the data transfer on stream1 to complete before being executed.

```
integer stream1=1
integer stream2=2
integer stream3=3
!$acc parallel loop async(stream3)
do i = 1, system_size
    ! Kernel launched on stream3
enddo

!$acc enter data copyin(array(:)) async(stream1)
! Some stuff

!$acc parallel async(stream2) wait(stream1, stream3)
    ! A wonderful kernel
!$acc end parallel
```

If you omit the clause options, then the operations will wait until all asynchronous operations fulfill.

```
!$acc parallel wait
    ! A wonderful kernel
!$acc end parallel
```

14.3 *wait* directive

wait comes also as a standalone directive.

```
integer stream1=1
integer stream2=2
integer stream3=3

!$acc parallel loop async(stream3)
do i = 1, system_size
    ! Kernel launched on stream3
enddo

!$acc enter data copyin(array(:)) async(stream1)
! Some stuff

!$acc wait(stream3)

!$acc parallel async(stream2)
    ! A wonderful kernel
!$acc end parallel
```

14.4 Exercise

In this exercise you have to compute the matrix product $C = A \times B$.

You have to add directives to:

- use the program lifetime unstructured data region to allocate memory on the GPU
- perform the matrix initialization on the GPU
- perform the matrix product on the GPU
- create and analyze a profile (add the option `--profile` to idrrun)
- save the .qdrep file
- check what can be done asynchronously and implement it
- create and analyze a profile (add the option `--profile` to idrrun)
- save the .qdrep file

Your solution is considered correct if no implicit action are done.

```
%%idrrun -a
!!  examples/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

    streamA = 1
```

(continues on next page)

(continued from previous page)

```
    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
```

14.4.1 Solution

```
%%idrrun -a
!!  examples/Fortran/async_async_solution.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

    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)

    !$acc parallel loop &
    !$acc present(A(:rank,:rank), B(:rank,:rank), C(:rank,:rank)) &
    !$acc gang wait(1,2,3)
    do j=1, rank
        do k=1, rank
            !$acc loop vector
            do i=1, rank
                !$acc atomic update
                C(i,j) = C(i,j) + A(i,k)*B(k,j)
            enddo
        enddo
    enddo
    !$acc exit data delete(A(:rank,:rank), B(:rank,:rank)) copyout(C(:rank,:rank))
    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))
            !$acc enter data create(mat(:rank,:rank)) async(stream)
        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

            !$acc parallel loop collapse(2) async(stream)
            do j=1, rank
                do i=1, rank
                   mat(i,j) = 0.0_real64
```

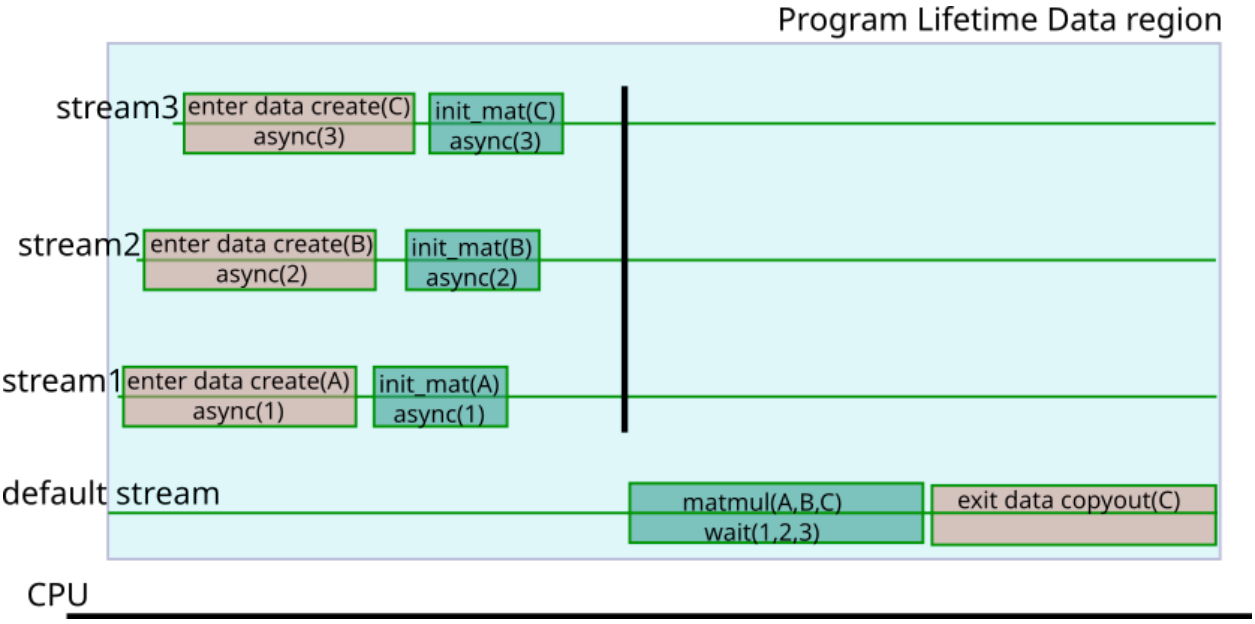
(continues on next page)

(continued from previous page)

```
                enddo
            enddo

            !$acc parallel loop async(stream)
            do j=1, rank
                mat(j,j) = diag
            enddo
        end subroutine init_mat
end program prod_mat
```

In an ideal world, the solution would produce a profile like this one:



14.4.2 Comments

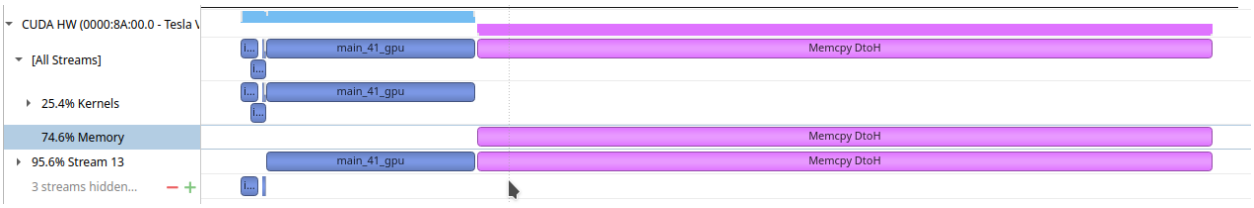
- Several threads will update the same memory location for C so you have to use an `acc atomic update`
- `collapse` is used to fuse the 3 loops. It helps the compiler to generate a more efficient code

14.5 Advanced NVIDIA compiler option to use Pinned Memory: `-gpu=pinned`

If you look at the profiles of your code (at this point "if" should be "when"), you can see that the memory transfers occurs in chunks of more or less constant size. Even though you have a large memory block it will be split into several smaller pieces which have the size of a memory page.

Memory not pinned:

Memory pinned:



Usually the transfer time is reduced when pinned memory is used. It can also cause some segmentation faults. Do your testing!

14.5.1 Bonus

You can launch the exercise with `%%idrrun -a --profile --accopts "cc70,pinned"` to test the effect of pinned memory. You can save a profile to compare the 3 versions.

Requirements:

- Get started
- Data management

CHAPTER

FIFTEEN

ATOMIC OPERATIONS

The `acc atomic` is kind of a generalization of the concept of reduction that we saw in (Get started)[../Get_started.ipynb]. However the mechanism is different and less efficient than the one used for reductions. So if you have the choice, use a *reduction* clause.

The idea is to make sure that only one thread at a time can perform a read and/or write operation on a **shared** variable.

The syntax of the directive depends on the clause you use.

15.1 Syntax

15.1.1 *read, write, update*

```
!$acc atomic <clause>
 ! One atomic operation
!$acc end atomic  ! This statement is optional
```

The clauses *read*, *write* and *update* only apply to the line immediately below the directive.

15.1.2 *capture*

The *capture* clause can work on a block of code:

```
!$acc atomic capture <clause>
 ! Set of atomic operations
!$acc end atomic  ! This statement is optional
```

15.2 Restrictions

The complete list of restrictions is available in the OpenACC specification.

We need the following information to understand the restrictions for each clause:

- **v** and **x** are scalar values
- *binop*: binary operator (for example: +, -, *, /, ++, –, etc)
- *expr* is an expression that reduces to a scalar and must have precedence over *binop*

15.2.1 *read*

The expression must be of the form:

```
!$acc atomic read
v = x
!$acc end atomic   ! This statement is optional
```

15.2.2 *write*

The expression must have the form:

```
!$acc atomic write
x = expr
!$acc end atomic  ! This statement is optional
```

15.2.3 *update*

Several forms are available:

```
!$acc atomic
x = x + (3*10)

!$acc atomic
x = max(x, 3.0, -1.0, 2.0/5.0)  ! The update clause is optional

!$acc atomic update
x = x + (3*10) ! The end atomic statement is optionnal

!$acc atomic
x = x + (3*10)
!$acc end atomic
```

15.2.4 *capture*

A capture is an operation where you set a variable with the value of an updated variable:

```
! x = x operator expr ( update statement)
! v = x               (capture statement)
!$acc atomic capture
x = x + (3*10)
v = x
!$acc end atomic

! x = intinsic_procedure(x,scalar_expr_list) ( update statement)
! v = x                                      (capture statement)
!$acc atomic capture
x = max(x, 3.0, -1.0, 2.0/5.0)
v = x
!$acc end atomic
```

(continues on next page)

(continued from previous page)

```
! v = x               (capture statement)
! x = x operator expr ( update statement)
!$acc atomic capture
v = x
x = x + (3*10)
!$acc end atomic

! v = x                                        (capture statement)
! x = intinsic_procedure(x,scalar_expr_list) ( update statement)
!$acc atomic capture
v = x
x = max(3.0, -1.0, 2.0/5.0, x)
!$acc end atomic


! v = x    (capture statement)
! x = expr (  write statement)
!$acc atomic capture
v = x
x = 3*10
!$acc end atomic
```

15.3 Exercise

Let's check if the default random number generator provided by the standard library gives good results.

In the example we generate an array of integers randomly set from 0 to 9. The purpose is to check if we have a uniform distribution.

We cannot perform the initialization on the GPU since the rand() function is not OpenACC aware.

You have to:

- Create a kernel for the integer counting
- Make sure that the results are correct (you should have around 10% for each number)

```
%%idrrun -a
!!  examples/Fortran/atomic_exercise.f90
program histogram
    use iso_fortran_env, only : REAL64, INT32
    implicit none

    integer(kind=INT32 ), dimension(:) , allocatable :: shots
    integer(kind=INT32 ), dimension(10)              :: histo
    integer(kind=INT32 ), parameter                  :: nshots = 1e9
    real   (kind=REAL64)                             :: random_real
    integer(kind=INT32 )                             :: i

    ! Histogram allocation and initialization
    do i = 1, 10
     histo(i) = 0
    enddo
```

(continues on next page)

(continued from previous page)

```
    ! Allocate memory for the random numbers
    allocate(shots(nshots))

    ! Fill the array on the CPU (rand is not available on GPU with Nvidia Compilers)
    do i = 1, nshots
        call random_number(random_real)
        shots(i) = floor(random_real * 10.0_real64) + 1
    enddo

    ! Count the number of time each number was drawn
    do i = 1, nshots
        histo(shots(i)) = histo(shots(i)) + 1
    enddo

    !  Print results
    do i = 1, 10
        write(0,"(i2,a2,i10,a2,f10.8,a1)") i,": ", histo(i), " (", real(histo(i))/1.
 ↪e9, ")"
    enddo

    deallocate(shots)

end program histogram
```

15.3.1 Solution

```
%%idrrun -a
!!  examples/Fortran/atomic_solution.f90
program histogram
    use iso_fortran_env, only : REAL64, INT32
    use openacc
    implicit none

    integer(kind=INT32 ), dimension(:) , allocatable :: shots
    integer(kind=INT32 ), dimension(10)              :: histo
    integer(kind=INT32 ), parameter                  :: nshots = 1e9
    real   (kind=REAL64)                             :: random_real
    integer(kind=INT32 )                             :: i

    ! Histogram allocation and initialization
    do i = 1, 10
     histo(i) = 0
    enddo

    ! Allocate memory for the random numbers
    allocate(shots(nshots))

    ! Fill the array on the CPU (rand is not available on GPU with Nvidia Compilers)
    do i = 1, nshots
        call random_number(random_real)
        shots(i) = floor(random_real * 10.0_real64) + 1
    enddo

    ! Count the number of time each number was drawn
```

(continues on next page)

(continued from previous page)

```
    !$acc parallel loop copyin(shots(:)) copyout(histo(:))
    do i = 1, nshots
        !$acc atomic
        histo(shots(i)) = histo(shots(i)) + 1
    enddo

    !  Print results
    do i = 1, 10
        write(0,"(i2,a2,i10,a2,f10.8,a1)") i,": ", histo(i), " (", real(histo(i))/1.
↪e9, ")"
    enddo

    deallocate(shots)

end program histogram
```

With compilers supporting it you can replace the atomic operation with a reduction on the array histo.

```
acc parallel loop reduction(+:histo)
```

Requirements:

- Get Started
- Data Management
- Atomic Operations

CHAPTER

SIXTEEN

DEEP COPY

Complex data structures, including struct and classes in C or derived datatypes with pointers and allocatable in Fortran, are frequent. Ways to managed them include:

- using CUDA unified memory with the compilation flag `-gpu:managed`, but the cost of memory allocation will be higher and it will apply to all allocatable variables
- flatten the derived datatypes by using temporary variables and then perform data transfers on the temporary variables
- using deep copy.

Two ways are possible to manage deep copy:

- top-down deep copy with an implicit attach behavior
- bottom-up deep copy with an explicit attach behavior

16.1 Top-down deep copy

In order to implement the top-down deep copy, we should copy to the device the base structure first and then the children structures. For each children transfer, the compiler's implementation will check if the pointers to the children (they are transferred with the parent structure) are present. If they are, an implicit attach behavior is performed and the parents on the device will point toward the children that are newly put on the device.

Please note that it is not mandatory to transfer all the children structure, only the ones that calculations on the device require.

16.1.1 Syntax

```
type velocity
    real, dimension(:), allocatable :: vx, vy, vz
end type

...
type(velocity) :: U

allocate(U%vx(sizeX), U%vy(sizeY), U%vz(sizeZ))
...

!$acc enter data copy(U)
!$acc enter data copy(U.vx(1:sizeX), U.vy(1:sizeY), U.vz(1:sizeZ))

! A humonguous calculation
```

16.1.2 Example

In this example we store 2 arrays in a structure/derived type and use a deep copy to make them available on the GPU.

```
%%idrrun -a
!!  examples/Fortran/Deep_copy_example.f90
program vector_addition
    use iso_fortran_env, only : INT32, REAL64
    use openacc
    implicit none

    type :: vectors
        real(kind=REAL64), dimension(:), allocatable :: s, c
    end type

    type(vectors) :: vec

    integer(kind=INT32 ), parameter              :: system_size = 1e5
    real   (kind=REAL64), dimension(system_size) :: array_sum
    real   (kind=REAL64)                         :: fortran_pi
    integer(kind=INT32 )                         :: i

    fortran_pi = acos(-1.0_real64)
    allocate(vec%s(system_size), vec%c(system_size))

    !$acc enter data create(vec, array_sum(:))
    !$acc enter data create(vec%s(1:system_size), vec%c(1:system_size))


    !$acc parallel
    !$acc loop
    do i = 1, system_size
        vec%s(i) = sin(i*fortran_pi/system_size) * sin(i*fortran_pi/system_size)
        vec%c(i) = cos(i*fortran_pi/system_size) * cos(i*fortran_pi/system_size)
    enddo
    !$acc end parallel

    !$acc parallel
    !$acc loop
    do i = 1, system_size - 1
        array_sum(i) = vec%s(i) + vec%c(system_size - i)
    enddo
    !$acc end parallel

    !$acc exit data delete(vec%s, vec%c)
    !$acc exit data delete(vec) copyout(array_sum(:))

    write(0,"(a10,f10.8)") "sum(42) = ",array_sum(42)
end program vector_addition
```

16.1.3 Exercise

In this exercise, we determine the radial distribution function (RDF) for an ensemble of particles that is read from a file. The position of the particles can be use as a demonstration on the implementation of the deep copy. You can run this example at the end of the next exercise to check the structure of the box at the end of the simulation.

First you need to copy some files:

```
%%bash
cp ../../examples/Fortran/OUTPUT ../../examples/Fortran/CONFIG .
```

You need to add `--cliopts "0.5 15.5 0"` to idrrun.

```
%%idrrun -a --cliopts "0.5 15.5"
!!  examples/Fortran/Deep_copy_exercise.f90
module utils_rdf
    use ISO_FORTRAN_ENV, only : REAL64, INT32
    implicit none

    type :: Location
        real(kind=REAL64), dimension(:), allocatable :: x, y, z
    end type

    type(Location) :: particle
    integer(kind= INT32), dimension(:), allocatable :: hist
    real   (kind=REAL64), dimension(:), allocatable :: gr
    real   (kind=REAL64) :: Rij, deltaR, rcut, nideal, rho, Lx, Ly, Lz
    integer(kind= INT32) :: d, read_restart, Natoms

    contains
    subroutine read_config()
        integer(kind= INT32)              :: i, ierr
        real   (kind=REAL64)              :: yy
        logical                           :: res = .false.
        inquire(file="OUTPUT", exist=res)
        if (res) then
            open(unit=20, file='OUTPUT', iostat=ierr)
            write(0,*) "Reading from OUTPUT file"
        else
            open(unit=20, file='CONFIG', iostat=ierr)
            write(0,*) "Reading from CONFIG file"
        endif
        read(20,*) yy, Natoms, Lx, Ly, Lz

        allocate(particle%x(Natoms), particle%y(Natoms), particle%z(Natoms))

        do i = 1, Natoms
            read(20,*) particle%x(i), particle%y(i), particle%z(i)
            if (read_restart .eq. 1) read(20,*) yy, yy, yy
            if (read_restart .eq. 2) read(20,*) yy, yy, yy
        enddo
        close(20)
        ! Add OpenACC directives here
    end subroutine read_config

    subroutine usage
    ! Only prints how to run the program
```

(continues on next page)

(continued from previous page)

```
    print *, "You should provide three arguments to run rdf : "
    print *, "./rdf deltaR rcut read_restart"
    print *, "- deltaR is the length of each bin, represented by a real*8 "
    print *, "- rcut is the total length on which you determine the rdf ( rcut < box_
↪length / 2 )"
    print *, "- read_restart defined if the file contains :"
    print *, "   - positions and velocities         (read_restart = 1)"
    print *, "   - positions, velocities and forces (read_restart2)"
    print *, "   - position only (input anything that is not 1 or 2)"
    print *, "example : ./rdf 0.5 15.5 0"
    stop
    end subroutine usage

end module utils_rdf

program rdf
    use utils_rdf
    integer(kind= INT32)          :: i, j, nargs, long, max_bin
    character(len=:), allocatable :: arg

    nargs = COMMAND_ARGUMENT_COUNT()
    if (nargs .eq. 3) then
        call GET_COMMAND_ARGUMENT(NUMBER=1, LENGTH=long)
        allocate(CHARACTER(len=long) :: arg)
        call GET_COMMAND_ARGUMENT(NUMBER=1, VALUE=arg)
        read(arg,'(f20.8)') deltaR
        deallocate(arg)
        call GET_COMMAND_ARGUMENT(NUMBER=2, LENGTH=long)
        allocate(CHARACTER(len=long) :: arg)
        call GET_COMMAND_ARGUMENT(NUMBER=2, VALUE=arg)
        read(arg,'(f20.8)') rcut
        deallocate(arg)
        call GET_COMMAND_ARGUMENT(NUMBER=3, LENGTH=long)
        allocate(CHARACTER(len=long) :: arg)
        call GET_COMMAND_ARGUMENT(NUMBER=3, VALUE=arg)
        read(arg,'(i10)') read_restart
        deallocate(arg)
    else
        call usage()
    endif

    call read_config()
    max_bin = int( rcut/deltaR ) + 1
    allocate(hist(max_bin), gr(max_bin))

    ! Add OpenACC directives here

    ! Add OpenACC directives here
    do i = 1, max_bin
        hist(i) = 0
    enddo

    ! Add OpenACC directives here
    do j = 1, Natoms
        ! Add OpenACC directives here
        do i = 1, Natoms
```

(continues on next page)

(continued from previous page)

```
            if (j .ne. i) then
                xij = particle%x(j)-particle%x(i)
                xij = xij - anint(xij/Lx) * Lx
                yij = particle%y(j)-particle%y(i)
                yij = yij - anint(yij/Lz) * Lz
                zij = particle%z(j)-particle%z(i)
                zij = zij - anint(zij/Lz) * Lz

                Rij = xij*xij + yij*yij + zij*zij
                d = int( sqrt(Rij)/deltaR ) + 1

                if (d .le. max_bin) then
                    ! Add OpenACC directives here
                    hist(d) = hist(d) + 1
                endif
            endif
        enddo
    enddo

    rho = dble(Natoms) / (Lx * Ly * Lz)
    rho =  4.0_real64 / 3.0_real64 * acos(-1.0_real64) * rho

    ! Add OpenACC directives here
    do i = 1, max_bin
        nideal  = rho * ( (i*deltaR)**3 - ((i-1)*deltaR)**3)
        gr(i)   = dble(hist(i)) / (nideal * dble(Natoms))
    enddo


    ! Add OpenACC directives here

    open(unit=30, file='RDF')
        do i=1,max_bin
            write(30,'(2f20.8)') (i-1)*deltaR, gr(i)
        enddo
    close(30)

    deallocate(particle%x, particle%y, particle%z)

end program rdf
```

```
import matplotlib.pyplot as plt
import numpy as np
rdf = np.genfromtxt("RDF", delimiter=' ').T
plt.plot(rdf[0], rdf[1])
```

16.1.4 Solution

```
%%idrrun -a --cliopts "0.5 15.5"
!!  examples/Fortran/Deep_copy_solution.f90
module utils_rdf
    use ISO_FORTRAN_ENV, only : REAL64, INT32
    implicit none

    type :: Location
        real(kind=REAL64), dimension(:), allocatable :: x, y, z
    end type

    type(Location) :: particle
    integer(kind= INT32), dimension(:), allocatable :: hist
    real   (kind=REAL64), dimension(:), allocatable :: gr
    real   (kind=REAL64) :: Rij, deltaR, rcut, nideal, rho, Lx, Ly, Lz
    integer(kind= INT32) :: d, read_restart, Natoms

    contains
    subroutine read_config()
        integer(kind= INT32)              :: i, ierr
        real   (kind=REAL64)              :: yy
        logical                           :: res = .false.
        inquire(file="OUTPUT", exist=res)
        if (res) then
            open(unit=20, file='OUTPUT', iostat=ierr)
            write(0,*) "Reading from OUTPUT file"
        else
            open(unit=20, file='CONFIG', iostat=ierr)
            write(0,*) "Reading from CONFIG file"
        endif
        read(20,*) yy, Natoms, Lx, Ly, Lz

        allocate(particle%x(Natoms), particle%y(Natoms), particle%z(Natoms))

        do i = 1, Natoms
            read(20,*) particle%x(i), particle%y(i), particle%z(i)
            if (read_restart .eq. 1) read(20,*) yy, yy, yy
            if (read_restart .eq. 2) read(20,*) yy, yy, yy
        enddo
        close(20)
        !$acc enter data copyin(particle)
        !$acc enter data copyin(particle%x(1:Natoms), particle%y(1:Natoms), particle
↪%z(1:Natoms))
    end subroutine read_config

    subroutine usage
    ! Only prints how to run the program
    print *, "You should provide three arguments to run rdf : "
    print *, "./rdf deltaR rcut read_restart"
    print *, "- deltaR is the length of each bin, represented by a real*8 "
    print *, "- rcut is the total length on which you determine the rdf ( rcut < box_
↪length / 2 )"
    print *, "- read_restart defined if the file contains :"
    print *, "   - positions and velocities         (read_restart = 1)"
    print *, "   - positions, velocities and forces (read_restart2)"
    print *, "   - position only (input anything that is not 1 or 2)"
```

(continues on next page)

(continued from previous page)

```
    print *, "example : ./rdf 0.5 15.5 0"
    stop
    end subroutine usage

end module utils_rdf

program rdf
    use utils_rdf
    integer(kind= INT32)          :: i, j, nargs, long, max_bin
    character(len=:), allocatable :: arg

    nargs = COMMAND_ARGUMENT_COUNT()
    if (nargs .eq. 3) then
        call GET_COMMAND_ARGUMENT(NUMBER=1, LENGTH=long)
        allocate(CHARACTER(len=long) :: arg)
        call GET_COMMAND_ARGUMENT(NUMBER=1, VALUE=arg)
        read(arg,'(f20.8)') deltaR
        deallocate(arg)
        call GET_COMMAND_ARGUMENT(NUMBER=2, LENGTH=long)
        allocate(CHARACTER(len=long) :: arg)
        call GET_COMMAND_ARGUMENT(NUMBER=2, VALUE=arg)
        read(arg,'(f20.8)') rcut
        deallocate(arg)
        call GET_COMMAND_ARGUMENT(NUMBER=3, LENGTH=long)
        allocate(CHARACTER(len=long) :: arg)
        call GET_COMMAND_ARGUMENT(NUMBER=3, VALUE=arg)
        read(arg,'(i10)') read_restart
        deallocate(arg)
    else
        call usage()
    endif

    call read_config()
    max_bin = int( rcut/deltaR ) + 1
    allocate(hist(max_bin), gr(max_bin))

    !$acc data copy(hist(:)) copyout(gr(:)) present(particle, particle%x(1:Natoms),␣
↪particle%y(1:Natoms), particle%z(1:Natoms))

    !$acc parallel loop present(hist(:max_bin))
    do i = 1, max_bin
        hist(i) = 0
    enddo

    !$acc parallel loop reduction(hist(:max_bin))
    do j = 1, Natoms
        !$acc loop reduction(hist(:max_bin)) private(xij, yij, zij, Rij, d)
        do i = 1, Natoms
            if (j .ne. i) then
                xij = particle%x(j)-particle%x(i)
                xij = xij - anint(xij/Lx) * Lx
                yij = particle%y(j)-particle%y(i)
                yij = yij - anint(yij/Lz) * Lz
                zij = particle%z(j)-particle%z(i)
                zij = zij - anint(zij/Lz) * Lz
```

(continues on next page)

(continued from previous page)

```
                Rij = xij*xij + yij*yij + zij*zij
                d = int( sqrt(Rij)/deltaR ) + 1

                if (d .le. max_bin) then
                    hist(d) = hist(d) + 1
                endif
            endif
        enddo
    enddo

    rho = dble(Natoms) / (Lx * Ly * Lz)
    rho =  4.0_real64 / 3.0_real64 * acos(-1.0_real64) * rho

    !$acc parallel loop
    do i = 1, max_bin
        nideal  = rho * ( (i*deltaR)**3 - ((i-1)*deltaR)**3)
        gr(i)   = dble(hist(i)) / (nideal * dble(Natoms))
    enddo

    !$acc end data
    !$acc exit data delete(particle%x, particle%y, particle%z)
    !$acc exit data delete(particle)

    open(unit=30, file='RDF')
        do i=1,max_bin
            write(30,'(2f20.8)') (i-1)*deltaR, gr(i)
        enddo
    close(30)

    deallocate(particle%x, particle%y, particle%z)

end program rdf
```

```
import matplotlib.pyplot as plt
import numpy as np
rdf = np.genfromtxt("RDF", delimiter=' ').T
plt.plot(rdf[0], rdf[1])
```

16.2 Deep copy with manual attachment

It is also possible to proceed to a bottom-up deep copy, in which you can first copy sub-objects on the accelerator and then attach them to existing children. With this procedure you will have to attach explicitly the pointers to the children. This can be easily apprehend with the subsequent code.

```
type vect2D
 real, dimension(:), allocatable :: x, y
 real, pointer                   :: coordinate(:)
end type

type(vect2D) :: v0

!$acc enter data copyin(v0%x, v0%y, v0%coordinate)
!$acc enter data copyin(v0)
```

Here the first copyin will pass the arrays on the device memory and the second will provide the complex datatype. But the pointers of the complex datatype (such as v0.x) will still reference the host structure. We should thus provide to the compiler the information of the datatype as it should be on the device. This is done by adding an `attach` directive.

```
type vect2D
 real, dimension(:), allocatable :: x, y
 real, pointer                   :: coordinate(:)
end type

type(vect2D) :: v0

!$acc enter data copyin(v0%x, v0%y, v0%coordinate)
!$acc enter data copyin(v0) attach(v0%x, v0%y, v0%coordinate)
```

Here, the pointer and its target are present on the device. The directive will inform the compiler to replace the host pointer with the corresponding device pointer in device memory.

The `detach` clause can be use to free the structure memory but is not mandatory.

```
!$acc exit data detach(v0%x, v0%y, v0%coordinate)    ! not required
!$acc exit data copyout(v0%x, v0%y, v0%coordinate)
!$acc exit data copyout(v0)
```

16.2.1 Exercise

In the following exercise we'll resolve the Lotka–Volterra predator–prey equations

for the prey population:

$\frac{dx}{dt} = birth_x x - death_x xy$

and for the predator population:

$\frac{dy}{dt} = birth_y xy - death_y y.$

```
%%idrrun -a
!!  examples/Fortran/Deep_copy_attach_detach_exercise.f90
module utils_lotka
    use ISO_FORTRAN_ENV, only : REAL64, INT32
    implicit none

    type :: population
        real(kind=REAL64),dimension(:),allocatable :: state
    end type

    contains
    subroutine derivee(x, dx, pop)
        type   (population ), dimension(2), intent(in)  :: pop
        real   (kind=REAL64), dimension(2), intent(in)  :: x
        real   (kind=REAL64), dimension(2), intent(out) :: dx
        ! Add openacc directives
        dx(1) =  pop(1)%state(2)*x(1) - pop(1)%state(3)*x(1)*x(2)
        dx(2) = -pop(2)%state(3)*x(2) + pop(2)%state(2)*x(1)*x(2)
    end subroutine derivee

    subroutine rk4(pop, dt)
        type   (population ), dimension(2), intent(inout) :: pop
```

(continues on next page)

(continued from previous page)

```
        real   (kind=REAL64), intent(in)                 :: dt

        real   (kind=REAL64), dimension(2)               :: x_temp, k1, k2, k3, k4
        real   (kind=REAL64)                             :: halfdt
        integer(kind= INT32)                             :: i

        halfdt = dt/2
        ! Add openacc directives

        do i = 1, 2
           x_temp(i) = pop(i)%state(1)
        enddo

        call Derivee(x_temp, k1, pop)
        ! Add openacc directives
        do i = 1, 2
             x_temp(i) = pop(i)%state(1) + k1(i)*halfdt
        enddo

        call Derivee(x_temp, k2, pop)
        ! Add openacc directives
        do i = 1, 2
            x_temp(i) = pop(i)%state(1) + k2(i)*halfdt
        enddo

        call Derivee(x_temp, k3, pop)
        ! Add openacc directives
        do i = 1, 2
            x_temp(i) = pop(i)%state(1) + k3(i)*dt
        enddo

        call Derivee(x_temp, k4, pop)
        ! Add openacc directives
        do i = 1, 2
            pop(i)%state(1) = pop(i)%state(1) + (dt/6.0)*(k1(i) + 2.0*k2(i) + 2.
↪0*k3(i) + k4(i))
        enddo
    end subroutine rk4

end module utils_lotka

program lotka_volterra
    use utils_lotka
    use openacc
    implicit none

    type   (population ), dimension(2)         :: pred_prey
    real   (kind=REAL64)                       :: ti, tf, dt, tmax
    integer                                    :: i

    ti   =   0.00
    dt   =   0.05
    tmax = 100.00

    allocate(pred_prey(2)%state(3), pred_prey(1)%state(3))
```

(continues on next page)

(continued from previous page)

```
    pred_prey(2)%state(1) = 15.00  ! predator count
    pred_prey(2)%state(2) = 0.01   ! predator birth rate
    pred_prey(2)%state(3) = 1.0    ! predator death rate

    pred_prey(1)%state(1) = 100.00 ! prey count
    pred_prey(1)%state(2) = 2.0    ! prey birth rate
    pred_prey(1)%state(3) = 0.02   ! prey death rate

    do i=1,2
    ! Add openacc directives to pass child structure to GPU first
    enddo
    ! Add openacc directives for parent structure and connection to child

    open(unit=42, file="output")
    do while (ti <= tmax)
        tf = ti + dt
        call rk4(pred_prey, dt)
        do i = 1, 2
            ! Add openacc directives
        enddo
        write(42,'(f20.8,a1,f20.8,a1,f20.8)') tf,";", pred_prey(1)%state(1),";", pred_
↪prey(2)%state(1)
        ti = tf
    end do
    close(42)

    do i=1,2
    ! Add openacc directives
    enddo
    ! Add openacc directives
end program lotka_volterra
```

```
from matplotlib import pyplot as plt
import numpy as np

data = np.genfromtxt("output", delimiter=';')
time = data[:, 0]
preys = data[:, 1]
predators = data[:, 2]

plt.plot(time, preys, color = 'blue')
plt.plot(time, predators, color = 'red')
```

16.2.2 Solution

```
%%idrrun -a
!!  examples/Fortran/Deep_copy_attach_detach_solution.f90
module utils_lotka
    use ISO_FORTRAN_ENV, only : REAL64, INT32
    implicit none

    type :: population
        real(kind=REAL64),dimension(:),allocatable :: state
```

(continues on next page)

(continued from previous page)

```
    end type

    contains
    subroutine derivee(x, dx, pop)
        type   (population ), dimension(2), intent(in)  :: pop
        real   (kind=REAL64), dimension(2), intent(in)  :: x
        real   (kind=REAL64), dimension(2), intent(out) :: dx
        !$acc data present(x(:), dx(:), pop(:), pop(1)%state(:), pop(2)%state(:))
        !$acc serial
        dx(1) =  pop(1)%state(2)*x(1) - pop(1)%state(3)*x(1)*x(2)
        dx(2) = -pop(2)%state(3)*x(2) + pop(2)%state(2)*x(1)*x(2)
        !$acc end serial
        !$acc end data
    end subroutine derivee

    subroutine rk4(pop, dt)
        type   (population ), dimension(2), intent(inout) :: pop
        real   (kind=REAL64), intent(in)                  :: dt

        real   (kind=REAL64), dimension(2)                :: x_temp, k1, k2, k3, k4
        real   (kind=REAL64)                              :: halfdt
        integer(kind= INT32)                              :: i

        halfdt = dt/2
        !$acc data create(k1(:), k2(:), k3(:), k4(:), x_temp(:)) present(pop(:),␣
↪pop(1)%state(:), pop(2)%state(:))

        !$acc parallel loop
        do i = 1, 2
           x_temp(i) = pop(i)%state(1)
        enddo

        call Derivee(x_temp, k1, pop)
        !$acc parallel loop
        do i = 1, 2
             x_temp(i) = pop(i)%state(1) + k1(i)*halfdt
        enddo

        call Derivee(x_temp, k2, pop)
        !$acc parallel loop
        do i = 1, 2
            x_temp(i) = pop(i)%state(1) + k2(i)*halfdt
        enddo

        call Derivee(x_temp, k3, pop)
        !$acc parallel loop
        do i = 1, 2
            x_temp(i) = pop(i)%state(1) + k3(i)*dt
        enddo

        call Derivee(x_temp, k4, pop)
        !$acc parallel loop
        do i = 1, 2
            pop(i)%state(1) = pop(i)%state(1) + (dt/6.0)*(k1(i) + 2.0*k2(i) + 2.
↪0*k3(i) + k4(i))
        enddo
```

(continues on next page)

(continued from previous page)

```
       !$acc end data
    end subroutine rk4

end module utils_lotka

program lotka_volterra
    use utils_lotka
    use openacc
    implicit none

    type   (population ), dimension(2)         :: pred_prey
    real   (kind=REAL64)                       :: ti, tf, dt, tmax
    integer                                    :: i

    ti   =   0.00
    dt   =   0.05
    tmax = 100.00

    allocate(pred_prey(2)%state(3), pred_prey(1)%state(3))

    pred_prey(2)%state(1) = 15.00  ! predator count
    pred_prey(2)%state(2) = 0.01   ! predator birth rate
    pred_prey(2)%state(3) = 1.0    ! predator death rate

    pred_prey(1)%state(1) = 100.00 ! prey count
    pred_prey(1)%state(2) = 2.0    ! prey birth rate
    pred_prey(1)%state(3) = 0.02   ! prey death rate

    do i=1,2
       !$acc enter data copyin(pred_prey(i)%state(:))
    enddo
    !$acc enter data copyin(pred_prey(1:2)) attach(pred_prey(1)%state(:), pred_prey(2)
 ↪%state(:))

    open(unit=42, file="output_solution")
    do while (ti <= tmax)
        tf = ti + dt
        call rk4(pred_prey, dt)
        do i = 1, 2
            !$acc update self(pred_prey(i)%state(1))
        enddo
        write(42,'(f20.8,a1,f20.8,a1,f20.8)') tf,";", pred_prey(1)%state(1),";", pred_
 ↪prey(2)%state(1)
        ti = tf
    end do
    close(42)

    do i=1,2
        !$acc exit data detach(pred_prey(i)%state)
        !$acc exit data delete(pred_prey(i)%state)
    enddo
    !$acc exit data delete(pred_prey(:))

end program lotka_volterra
```

```
from matplotlib import pyplot as plt
import numpy as np

data = np.genfromtxt("output_solution", delimiter=';')
time = data[:, 0]
preys = data[:, 1]
predators = data[:, 2]

plt.plot(time, preys, color = 'blue')
plt.plot(time, predators, color = 'red')
```

Requirements:

- Get started
- Atomic operations
- Data Management

CHAPTER

SEVENTEEN

USING CUDA LIBRARIES

OpenACC is interoperable with CUDA and GPU-accelerated libraries. It means that if you create some variables with OpenACC you will be able to use the GPU (device) pointer to call a CUDA function.

17.1 acc host_data use_device

To call a CUDA function, the host needs to retrieve the address of your variable on the GPU. For example:

```
real, dimension(system_size) :: array
!$acc enter data create(array(:))

!$acc host_data use_device(array)
    ! inside the block `array` stores the address on the GPU
    call cuda_function(array)
!$acc end host_data
```

17.2 Example with CURAND

The pseudo-random number generators of the standard libraries are not (as of 2021) available with OpenACC. One solution is to use CURAND from NVIDIA.

In this example we generate a large array of random integer numbers in [0,9] with CURAND. Then a count of each occurrence is performed on the GPU with OpenACC.

The implementation of the generation of the integers list is given but is beyond the scope of the training course.

```
%%idrrun -a --options "-Mcudalib=curand"
!!  examples/Fortran/Using_CUDA_random_example.f90
program using_cuda
    use openacc
    use openacc_curand
    use, intrinsic :: ISO_FORTRAN_ENV , only : REAL32, INT32
    implicit none

    integer(kind=INT32), dimension(:), allocatable :: shots
    integer(kind=INT32)                            :: histo(10)
    integer(kind=INT32)                            :: nshots
    type(curandStateXORWOW)                        :: h
    integer(kind=INT32)                            :: seed,seq,offset
    integer(kind=INT32)                            :: i
```

(continues on next page)

(continued from previous page)

```
    do i = 1, 10
        histo(i) = 0
    enddo

    nshots = 1e9
    !  Allocate memory for the random numbers
    allocate(shots(nshots))

    ! NVIDIA curand will create our initial random data
    !$acc parallel create(shots(:)) copyout(histo(:))
    seed = 1234!5 + j
    seq = 0
    offset = 0
    !$acc loop vector
    do i = 1, 32
        call curand_init(seed, seq, offset, h)
    enddo

    !$acc loop
    do i = 1, nshots
        shots(i) = abs(curand(h))
    enddo

    !  Count the number of time each number was drawn
    !$acc loop
    do i = 1, nshots
        shots(i) = mod(shots(i),10) + 1
        !$acc atomic update
        histo(shots(i)) = histo(shots(i)) + 1
    enddo
    !$acc end parallel
    !  Print results
    do i = 1, 10
        write(0,"(i2,a2,i10,a2,f5.3,a1)") i,": ",histo(i)," (",dble(histo(i))/
 ↪dble(1e9),")"
    enddo
    deallocate(shots)
end program using_cuda
```

It is also possible to create interface to call CUDA functions and CUDA homemade kernels directly. The example below reproduce the above CURAND example by calling it without using the OpenACC API.

```
%%idrrun -n -l cuda --options "-arch=sm_70" --object --keep Using_CUDA_random_
 ↪function.cu
#include <stdio.h>
#include <curand.h>

// Fill d_buffer with num random numbers
extern "C" void fill_rand(float *d_buffer, int num, void *stream)
{
  curandGenerator_t gen;
  int status;

  // Create generator
  status = curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
```

(continues on next page)

(continued from previous page)

```
  // Set CUDA stream
  status |= curandSetStream(gen, (cudaStream_t)stream);

  // Set seed
  status |= curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);

  // Generate num random numbers
  status |= curandGenerateUniform(gen, d_buffer, num);

  // Cleanup generator
  status |= curandDestroyGenerator(gen);

  if (status != CURAND_STATUS_SUCCESS) {
      printf ("curand failure!\n");
      exit (EXIT_FAILURE);
  }
}
```

```
%%idrrun -a -l fortran --options "-Mcudalib=curand Using_CUDA_random_function.cu.o"
program using_cuda
    use ISO_C_BINDING
    use openacc
    use, intrinsic :: ISO_FORTRAN_ENV , only : REAL32, INT32
    implicit none

    interface
        subroutine fill_rand(positions, length, stream) BIND(C,NAME='fill_rand')
            use ISO_C_BINDING
            use openacc
            implicit none
            type (C_PTR)   , value          :: positions
            integer (C_INT), value          :: length
            integer(acc_handle_kind), value :: stream
        end subroutine fill_rand
     end interface


    integer(C_INT), dimension(:), allocatable :: shots
    integer(C_INT)                            :: histo(10)
    integer(C_INT)                            :: nshots
    integer(acc_handle_kind)                  :: stream

    integer(kind=INT32)                       :: i,j

    do i = 1, 10
        histo(i) = 0
    enddo

    nshots = 1e9
    !  Allocate memory for the random numbers
    allocate(shots(nshots))

    ! OpenACC may not use the default CUDA stream so we must query it
    stream = acc_get_cuda_stream(acc_async_sync)
```

(continues on next page)

(continued from previous page)

```
    !$acc data create(shots(:)) copyout(histo(:))
    ! NVIDIA cuRandom will create our initial random data
    !$acc host_data use_device(shots)
    call fill_rand(C_LOC(shots), nshots, stream)
    !$acc end host_data

    !  Count the number of time each number was drawn
    !$acc parallel loop present(shots(:), histo(:))
    do i = 1, nshots
        shots(i) = mod(shots(i),10) + 1
        !$acc atomic update
        histo(shots(i)) = histo(shots(i)) + 1
    enddo
    !$acc end data
    !  Print results
    do i = 1, 10
        write(0,"(i2,a2,i10,a2,f5.3,a1)") i,": ",histo(i)," (",dble(histo(i))/
 ↪dble(1e9),")"
    enddo
    deallocate(shots)
end program using_cuda
```

Requirements:

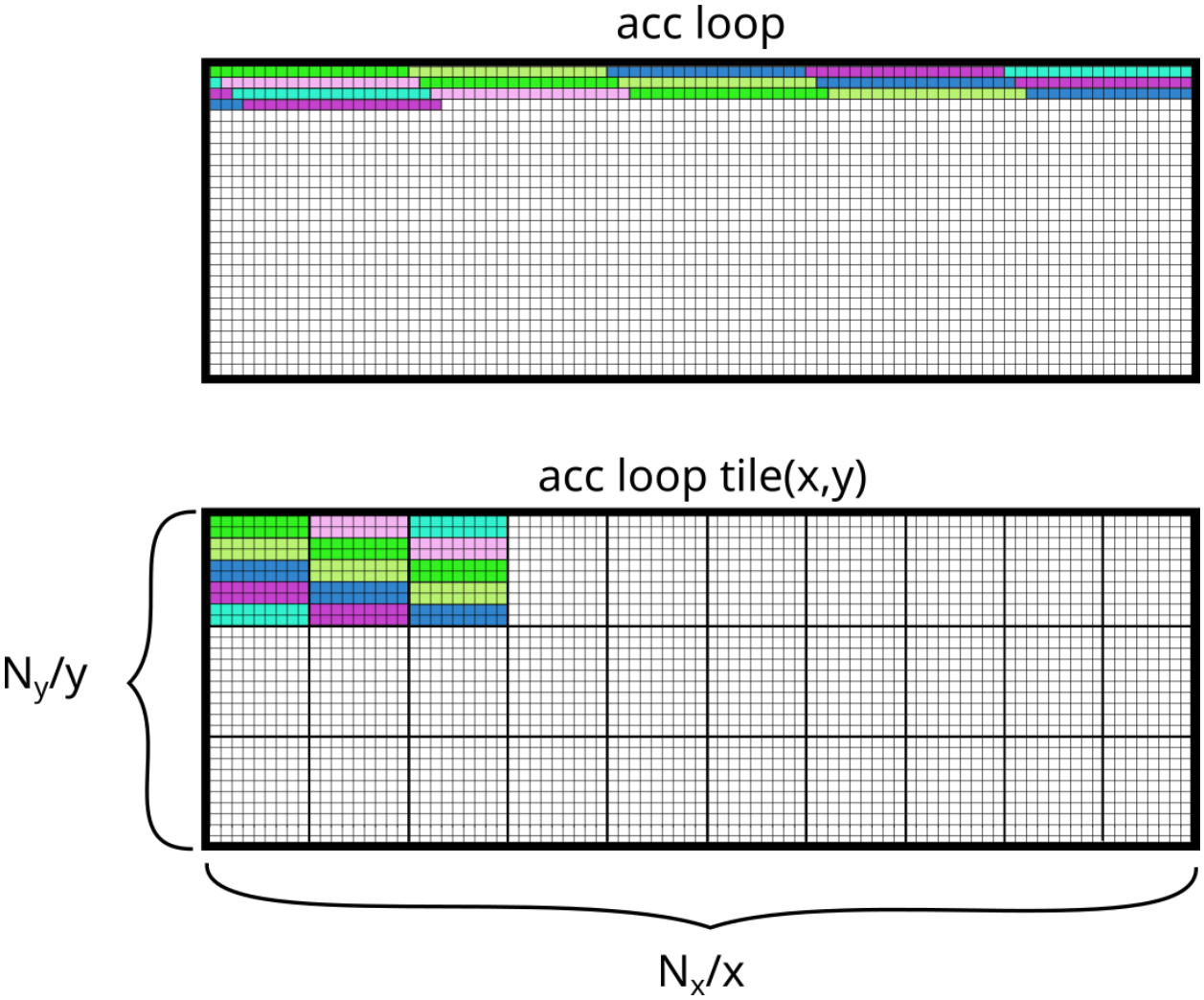
- Get started
- Data Management

CHAPTER

EIGHTEEN

LOOP TILING

Nested loops often reuse the same data across their iterations and keeping the working set inside the caches can improve performance. Tiling is a partitioning method of the loops into blocks. It reorders the loops so that each block will repeatedly hit the cache. A first usage restriction will thus be on the loops' nature itself: not all loops can benefit from tiling, only the ones that will reuse data while showing a poor data locality, thus leading to frequent cache misses.



OpenACC allows to improve data locality inside loops with the dedicated *tile* clause. It specifies the compiler to split each loop in the nest into 2 loops, with an outer set of tile loops and an inner set of element loops.

18.1 Syntax

The tile clause may appear with the *loop* directive for nested loops. For *N* nested loops, the tile clause can take *N* arguments. The first one being the size of the inner loop of the nest, the last one being the size of the outer loop.

```
!$acc loop tile(32,32)
do i = 1, size_i
    do j = 1, size_j
        ! A Fabulous calculation
    enddo
enddo
```

18.2 Restrictions

- the tile size (corresponding to the product of the arguments of the tile clause) can be up to 1024
- for better performance the size for the inner loop is a power of 2 (best with 32 to fit a cuda warp)
- if the vector clause is specified, it is then applied to the element loop
- if the gang clause is specified, it is then applied to the tile loop
- the worker clause is applied to the element loop only if the vector clause is not specified

18.3 Example

In the following example, tiling is used to solve a matrix multiplication followed by an addition. Let us take a look at the performance of the naïve algorithm and the manual tiling on CPU.

```
%%idrrun
!!  examples/Fortran/Loop_tiling_example_cpu.f90
program tiles
    use ISO_Fortran_env, only : INT32, REAL64
    implicit none
    integer(kind=INT32), parameter :: ni=4280, nj=4024, nk=1960, ntimes=30
    real(kind=REAL64)              :: a(ni,nk), b(nk,nj), c(ni,nj), d(ni,nj)
    real(kind=REAL64)              :: summation, t1, t2
    integer(kind=INT32)            :: nt, i, j, k
    integer(kind=INT32)            :: ichunk, l, ii, jj, kk

    call random_number(a)
    call random_number(b)

    a = 4.0_real64*a - 2.0_real64
    b = 8.0_real64*b - 4.0_real64
    c = 2.0_real64
    d = 0.0_real64

    print *, "Start calculation"

    call cpu_time(t1)
    do nt = 1, ntimes
      d = c + matmul(a,b)
```

(continues on next page)

(continued from previous page)

```
end do
call cpu_time(t2)
print *, "CPU matmul"
print *, "elapsed",t2-t1

print *,sum(d)
d = 0.0_real64
print *, " "


call cpu_time(t1)
do nt = 1, ntimes
  do j=1,nj
     do i =1,ni
       summation = 0.0_real64
       do k=1,nk
          summation = summation + a(i,k) * b(k,j)
       enddo
       d(i,j) = summation + c(i,j)
     enddo
  enddo
enddo
call cpu_time(t2)
print *, "CPU naive loop"
print *, "elapsed",t2-t1
print *,sum(d)
d = 0.0_real64
print *, " "


call cpu_time(t1)
l=size(a,dim=2)
ichunk = 256
do nt = 1, ntimes
d = 0.0_real64
do jj = 1, nj, ichunk
    do kk = 1, l, ichunk
       do j=jj,min(jj+ichunk-1,nj)
         do k=kk,min(kk+ichunk-1,l)
          do i=1,ni
            d(i,j) = d(i,j) + a(i,k) * b(k,j)
          enddo
         enddo
        enddo
     enddo
enddo
do j = 1, nj
    do i = 1, ni
        d(i,j) = d(i,j) +c(i,j)
    enddo
enddo
enddo
call cpu_time(t2)
print *, "CPU manual loop tiling 256"
print *, "elapsed",t2-t1
print *,sum(d)
```

(continues on next page)

(continued from previous page)

```
    d = 0.0_real64
    print *, " "


    call cpu_time(t1)
    l=size(a,dim=2)
    ichunk = 512
    do nt = 1, ntimes
    d = 0.0_real64
    do jj = 1, nj, ichunk
        do kk = 1, l, ichunk
           do j=jj,min(jj+ichunk-1,nj)
             do k=kk,min(kk+ichunk-1,l)
              do i=1,ni
                d(i,j) = d(i,j) + a(i,k) * b(k,j)
              enddo
             enddo
            enddo
         enddo
    enddo
    do j = 1, nj
        do i = 1, ni
            d(i,j) = d(i,j) +c(i,j)
        enddo
    enddo
    enddo
    call cpu_time(t2)
    print *, "CPU manual loop tiling 512"
    print *, "elapsed",t2-t1
    print *,sum(d)
    d = 0.0_real64

end program tiles
```

And now it's GPU implementation.

```
%%idrrun -a
!!  examples/Fortran/Loop_tiling_example_gpu.f90
program tiles
    use ISO_Fortran_env, only : INT32, REAL64
    implicit none
    integer(kind=INT32), parameter :: ni=4280, nj=4024, nk=1960, ntimes=30
    real(kind=REAL64)              :: a(ni,nk), b(nk,nj), c(ni,nj), d(ni,nj)
    real(kind=REAL64)              :: summation, t1, t2
    integer(kind=INT32)            :: nt, i, j, k
    integer(kind=INT32)            :: ichunk, l, ii, jj, kk

    call random_number(a)
    call random_number(b)

    a = 4.0_real64*a - 2.0_real64
    b = 8.0_real64*b - 4.0_real64
    c = 2.0_real64
    d = 0.0_real64

    print *, "Start calculation"
```

(continues on next page)

(continued from previous page)

```
!$acc enter data copyin(a,b,c) create(d)

call cpu_time(t1)
l=size(a,dim=2)
ichunk = 256
do nt = 1, ntimes
!$acc parallel loop default(present)
do j = 1, nj
    !$acc loop
    do i = 1, ni
        d(i,j) = 0.0_real64
    enddo
enddo

!$acc parallel loop default(present)
do jj = 1, nj, ichunk
    do kk = 1, l, ichunk
       do j=jj,min(jj+ichunk-1,nj)
         do k=kk,min(kk+ichunk-1,l)
          !$acc loop vector
          do i=1,ni
            d(i,j) = d(i,j) + a(i,k) * b(k,j)
          enddo
         enddo
        enddo
     enddo
enddo
!$acc parallel loop default(present)
do j = 1, nj
    !$acc loop
    do i = 1, ni
        d(i,j) = d(i,j) +c(i,j)
    enddo
enddo
enddo
call cpu_time(t2)
print *, "GPU manual loop tiling 256"
print *, "elapsed",t2-t1
!$acc update self(d(:,:))
print *,sum(d)
!$acc kernels
d(:,:) = 0.0_real64
!$acc end kernels
print *, " "


call cpu_time(t1)
l=size(a,dim=2)
ichunk = 512
do nt = 1, ntimes
!$acc parallel loop default(present)
do j = 1, nj
    !$acc loop
    do i = 1, ni
        d(i,j) = 0.0_real64
```

(continues on next page)

(continued from previous page)

```
    enddo
enddo

!$acc parallel loop gang default(present)
do jj = 1, nj, ichunk
    !$acc loop seq
    do kk = 1, l, ichunk
       !$acc loop seq
       do j=jj,min(jj+ichunk-1,nj)
         do k=kk,min(kk+ichunk-1,l)
          !$acc loop vector
          do i=1,ni
            d(i,j) = d(i,j) + a(i,k) * b(k,j)
          enddo
         enddo
        enddo
     enddo
enddo
!$acc parallel loop default(present)
do j = 1, nj
    !$acc loop
    do i = 1, ni
        d(i,j) = d(i,j) +c(i,j)
    enddo
enddo
enddo
call cpu_time(t2)
print *, "GPU manual loop tiling 512"
print *, "elapsed",t2-t1
!$acc update self(d(:,:))
print *,sum(d)
!$acc kernels
d(:,:) = 0.0_real64
!$acc end kernels
print *, " "

call cpu_time(t1)
l=size(a,dim=2)
ichunk = 16
do nt = 1, ntimes
!$acc parallel loop default(present)
do j = 1, nj
    !$acc loop
    do i = 1, ni
        d(i,j) = 0.0_real64
    enddo
enddo

!$acc parallel loop gang default(present)
do jj = 1, nj, ichunk
    !$acc loop seq
    do kk = 1, l, ichunk
       !$acc loop seq
       do j=jj,min(jj+ichunk-1,nj)
         do k=kk,min(kk+ichunk-1,l)
          !$acc loop vector
```

(continues on next page)

(continued from previous page)

```
            do i=1,ni
              d(i,j) = d(i,j) + a(i,k) * b(k,j)
            enddo
          enddo
        enddo
     enddo
enddo
!$acc parallel loop default(present)
do j = 1, nj
    !$acc loop
    do i = 1, ni
        d(i,j) = d(i,j) +c(i,j)
    enddo
enddo
enddo
call cpu_time(t2)
print *, "GPU manual loop tiling 16"
print *, "elapsed",t2-t1
!$acc update self(d(:,:))
print *,sum(d)
!$acc kernels
d(:,:) = 0.0_real64
!$acc end kernels
print *, " "


call cpu_time(t1)
do nt = 1, ntimes
  !$acc parallel loop default(present)
  do j=1,nj
     !$acc loop
     do i =1,ni
       summation = 0.0_real64
       !$acc loop seq
       do k=1,nk
          summation = summation + a(i,k) * b(k,j)
       enddo
       d(i,j) = summation + c(i,j)
     enddo
  enddo
enddo

call cpu_time(t2)
print *, "GPU naive parallel loop"
print *, "elapsed",t2-t1

!$acc update self(d(:,:))
print *,sum(d)
print *, " "
d(:,:) = 0.0_real64

call cpu_time(t1)
do nt = 1, ntimes
  !$acc parallel loop tile(32,32) default(present)
  do j=1,nj
     do i=1,ni
```

(continues on next page)

(continued from previous page)

```
           summation = 0.0_real64
           !$acc loop seq
           do k=1,nk
              summation = summation + a(i,k) * b(k,j)
           enddo
           d(i,j) = summation + c(i,j)
         enddo
      enddo
    enddo

    call cpu_time(t2)
    print *, "GPU naive parallel loop tiled"
    print *, "elapsed",t2-t1

    !$acc exit data copyout(d(:,:))

    print *,sum(d)
end program tiles
```

18.4 Exercise

In this exercise, you will try to accelerate the numerical resolution of the 2D Laplace's equation with tiles. You can see that tiles parameter should be chosen wisely in order not to deteriorate performance.

```
%%idrrun -a
!!  examples/Fortran/Loop_tiling_exercise.f90
program laplace2d
use ISO_FORTRAN_ENV, only : real64, int32
!use openacc
implicit none

! Calculated solution for (E,B) fields
real   (kind=real64), dimension(:,:), allocatable :: T, T_new
! Dimension of the system
integer(kind=int32 )                              :: nx, ny ! number of points

integer(kind=int32 )                              :: i, j, it
real   (kind=real64)                              :: erreur

nx = 20000 !30000
ny = 10000 !30000

allocate(T(nx,ny),T_new(nx,ny))

! initial conditions
do j=2,ny-1
    do i=2,nx-1
        T(i,j)     = 0.0_real64
        T_new(i,j) = 0.0_real64
    enddo
enddo
!
do i=1,nx
```

(continues on next page)

(continued from previous page)

```
    T(i, 1) = 100.0_real64
    T(i,ny) = 0.0_real64
enddo
!
do i=1,ny
    T(1 ,i) = 0.0_real64
    T(nx,i) = 0.0_real64
enddo

! add acc directive
do it = 1, 10000
  erreur = 0.0_real64
  !add acc directive
  do j= 2,ny-1
      do i= 2,nx-1
         T_new(i,j) =  0.25_real64*(T(i+1,j)+T(i-1,j) + &
                                    T(i,j+1)+T(i,j-1))
         erreur = max(erreur, abs(T_new(i,j) - T(i,j)))
      enddo
  enddo

  if (mod(it,100) .eq. 0) print *, "iteration: ",it," erreur: ",erreur

  !add acc directive
  do j= 2,ny-1
  do i= 2,nx-1
     T(i,j) =  T_new(i,j)
  enddo
  enddo
enddo


deallocate(T, T_new)

end program laplace2d
```

18.5 Solution

```
%%idrrun -a
!!  examples/Fortran/Loop_tiling_solution.f90
program laplace2d
use ISO_FORTRAN_ENV, only : real64, int32
!use openacc
implicit none

! Calculated solution for (E,B) fields
real   (kind=real64), dimension(:,:), allocatable :: T, T_new
! Dimension of the system
integer(kind=int32 )                              :: nx, ny ! number of points

integer(kind=int32 )                              :: i, j, it
real   (kind=real64)                              :: erreur
```

(continues on next page)

(continued from previous page)

```
nx = 20000 !30000
ny = 10000 !30000

allocate(T(nx,ny),T_new(nx,ny))

! initial conditions
do j=2,ny-1
    do i=2,nx-1
        T(i,j)     = 0.0_real64
        T_new(i,j) = 0.0_real64
    enddo
enddo
!
do i=1,nx
    T(i, 1) = 100.0_real64
    T(i,ny) = 0.0_real64
enddo
!
do i=1,ny
    T(1 ,i) = 0.0_real64
    T(nx,i) = 0.0_real64
enddo

!$acc data copy(T) create(T_new)
do it = 1, 10000
  erreur = 0.0_real64
  !$acc parallel loop tile(32,32) reduction(max:erreur)
  do j= 2,ny-1
      do i= 2,nx-1
         T_new(i,j) =  0.25_real64*(T(i+1,j)+T(i-1,j) + &
                                    T(i,j+1)+T(i,j-1))
         erreur = max(erreur, abs(T_new(i,j) - T(i,j)))
      enddo
  enddo

  if (mod(it,100) .eq. 0) print *, "iteration: ",it," erreur: ",erreur

  !$acc parallel loop tile(32,32)
  do j= 2,ny-1
  do i= 2,nx-1
     T(i,j) =  T_new(i,j)
  enddo
  enddo
enddo
!$acc end data

deallocate(T, T_new)

end program laplace2d
```

Requirements:

- Get Started
- Data Management

CHAPTER

NINETEEN

HANDS-ON MD SIMULATION OF LENNARD-JONES SYSTEM

This hands-on simulate a Lennard Jones system with a Berendsen thermostat while the time integration is done using velocity Verlet algorithm.

19.1 What to do

In this hands-on you will have to create the data structures to minimize the data transfers between CPU and GPU and brings all the calculation on the accelerator.

The main part of the program is dm. The configuration parameters and the initial configuration of the box are read from input files, namely INPUT_NML and CONFIG. A temporal loop will, at each step, determine the forces, positions and velocities, then a barostat is applied and ouputs are wrote in the file OUPUT according to the configuration parameter.

```
dm
 |
 |--read_params (read from "INPUT_NML", see comments inside the file)
 |
 |--read_config (read from "CONFIG": start_time, number of atoms,
 |       |                              positions and possibly velocities and forces)
 |       |
 |       |--allocate_variables (allocate size for positions, velocities and forces)
 |
 |
 |- Temporal loop
 |       |
 |       |--forces_from_LJ_potential (determines the values of the force components :
 |       |                              particle%Fext%Fx, particle%Fext%Fy, particle%Fext
↪%Fz)
 |       |
 |       |--velocity_verlet (use velocity verlet alogrithm to update positions and␣
↪velocity)
 |       |          |
 |       |          |--forces_from_LJ_potential
 |       |             (update forces from new positions)
 |       |
 |       |--berendsen_thermostat (determines kinetic energy, temperature,
 |       |                          update velocites)
 |       |
 |       |--write_config (write in "OUTPUT" depending on 'print_config' variable :
 |       |                 simulation_time, number of atoms, positions and possibly␣
↪velocities and forces)
 |       |
```

(continues on next page)

(continued from previous page)

```
 |       |
 |       | write kinetic energy and temperature in "THERMO" file depending on 'print_
↪thermo' variable
 |
 | deallocate variables (deallocate positions, velocities and forces)
```

If you are having difficulties with some part of the code, you can take a look at the following advice:

```
dm
 |
 |--read_params
 |
 |       HINT : nothing to do here
 |
 |--read_config
 |       |
 |       |  HINT : 1/ you can open an unstructured data region here with a deep copy
 |       |          2/ if you use a deep create, don't forget that "CONFIG" is read on␣
↪CPU
 |       |
 |       |--allocate_variables
 |
 |          HINT : 1/ you can open an unstructured data region here with a deep create
 |                 2/ you can fully put the derived type or partially
 |                 3/ scalars should be create on gpu to fully transfer the derived␣
↪type
 |
 |
 |- Temporal loop
 |       |
 |       |  HINT : At the end, no data structures should be opened or closed inside␣
↪the loop
 |       |
 |       |--forces_from_LJ_potential
 |       |
 |       |  HINT : 1/ Most of the time is spent here, it is a good place to start
 |       |          2/ It is preferable to instruct the compiler on the variables␣
↪present on GPU
 |       |
 |       |--velocity_verlet
 |       |           |
 |       |           |  HINT : It is preferable to guide the compiler on the variables␣
↪present on GPU
 |       |           |
 |       |           |--forces_from_LJ_potential
 |       |           |
 |       |           |  HINT : don't forget data transfers if you try to accelerate the␣
↪loops separatly
 |       |
 |       |--berendsen_thermostat
 |       |
 |       |  HINT : 1/ If the results diverge when you accelerate this routine, try to␣
↪determine
 |       |             the loop that makes it diverge
 |       |          2/ You can try to input the value of m0 directly in the loop and␣
↪check if it works
```

(continues on next page)

(continued from previous page)

```
|       |
|       |--write_config
|       |
|       |  HINT : writing is done by CPU
|       |
|       | write kinetic energy and temperature in "THERMO" file
|
|         HINT : nothing to do here
|
| deallocate variables (deallocate positions, velocities and forces)

   HINT : be sure to release GPU memory before freeing the variables on CPU
```

First you need to copy the configuration file.

```
%%bash
cp ../../examples/Fortran/INPUT_NML ../../examples/Fortran/CONFIG .
```

The hands-on starts here :

```
%%idrrun -a
!!  examples/Fortran/Hands_on_LJ_exercise.f90
module utils
    use ISO_FORTRAN_ENV, only : REAL64, INT32
    implicit none

    type :: Force
        real(kind=REAL64), dimension(:), allocatable :: Fx, Fy, Fz
    end type
    type :: velocity
        real(kind=REAL64), dimension(:), allocatable :: vx, vy, vz
    end type
    type :: Pos
        real(kind=REAL64), dimension(:), allocatable :: x, y, z
    end type

    type :: atom
        type(Force)        :: Fext
        type(Velocity)     :: V
        type(Pos)          :: R
        real (kind=REAL64) :: m0
    end type

    type(atom)           :: particle
    ! position parameters
    integer(kind= INT32) :: Natoms
    real   (kind=REAL64) :: Lx, Ly, Lz

    ! potential and force parameters
    real   (kind=REAL64) :: epsilon0, sigma0
    real   (kind=REAL64) :: LJ_tolerance = 1e-9
    real   (kind=REAL64) :: rcut = 15.0_real64

    ! temporal parameters
    integer(kind= INT32) :: it_max
```

(continues on next page)

(continued from previous page)

```
   real   (kind=REAL64) :: dt, end_time, print_config, print_thermo, start_time=0.0_
↪real64
   integer(kind= INT32) :: write_restart = 0, read_restart = 0

   ! Thermodynamic parameters
   real   (kind=REAL64) :: Ek, T, T0 ! kinetic energy & temperature from simulation,␣
↪T0 = targeted temperature
   real   (kind=REAL64), parameter :: kb = 0.831451115

   ! Berendsen thermostat
   real   (kind=REAL64) :: lambda_scaling_factor, tau_temp

   contains
   subroutine forces_from_LJ_potential()
       real   (kind=REAL64) :: rij, xij, yij, zij
       real   (kind=REAL64) :: Fxij, Fyij, Fzij, sigma1, sr2, sr6, sr12
       integer(kind= INT32) :: i, j, divergence = 0
       sigma1   = sigma0*sigma0

       ! Add open acc directives

       do i = 1, Natoms
           Fxij = 0.0_real64
           Fyij = 0.0_real64
           Fzij = 0.0_real64
           ! Add open acc directives
           do j = 1, Natoms ! avoid triangular matrix on gpu

           if (i .ne. j) then
               xij = particle%R%x(j)-particle%R%x(i)
               xij = xij - anint(xij/Lx) * Lx
               yij = particle%R%y(j)-particle%R%y(i)
               yij = yij - anint(yij/Lz) * Lz
               zij = particle%R%z(j)-particle%R%z(i)
               zij = zij - anint(zij/Lz) * Lz

               rij = xij*xij + yij*yij + zij*zij
               if ((rij .gt. LJ_tolerance) .and. (rij .le. rcut)) then
                   sr2  = sigma1 / rij
                   sr6  = sr2 *  sr2 * sr2
                   sr12 = sr6 * sr6
                   sr6  = sr6 / rij
                   sr12 = sr12 / rij
                   Fxij = (sr12 - 24_real64 * sr6) * xij + Fxij
                   Fyij = (sr12 - 24_real64 * sr6) * yij + Fyij
                   Fzij = (sr12 - 24_real64 * sr6) * zij + Fzij
               else
                       if (rij .lt. LJ_tolerance) then
                               divergence = divergence + 1  ! avoid early break on␣
↪GPU
                       endif
               endif

           endif
           enddo
           particle%Fext%Fx(i) = 48.0_real64*epsilon0*Fxij
```

(continues on next page)

(continued from previous page)

```
            particle%Fext%Fy(i) = 48.0_real64*epsilon0*Fyij
            particle%Fext%Fz(i) = 48.0_real64*epsilon0*Fzij
        enddo

        if (divergence .ne. 0) then
             write(0,*) 'Particles are too close'
             stop
        endif

    end subroutine forces_from_LJ_potential

    subroutine velocity_verlet()
        integer(kind= INT32) :: i

        ! Add open acc directives

        do i = 1, Natoms
            particle%V%vx(i) = particle%V%vx(i) + 0.5_real64 * dt * particle%Fext
↪%Fx(i)
            particle%V%vy(i) = particle%V%vy(i) + 0.5_real64 * dt * particle%Fext
↪%Fy(i)
            particle%V%vz(i) = particle%V%vz(i) + 0.5_real64 * dt * particle%Fext
↪%Fz(i)

            particle%R%x(i) = particle%R%x(i) + dt*particle%V%vx(i)
            particle%R%y(i) = particle%R%y(i) + dt*particle%V%vy(i)
            particle%R%z(i) = particle%R%z(i) + dt*particle%V%vz(i)
            particle%R%x(i) = particle%R%x(i) - anint(particle%R%x(i)/Lx) * Lx
            particle%R%y(i) = particle%R%y(i) - anint(particle%R%y(i)/Ly) * Ly
            particle%R%z(i) = particle%R%z(i) - anint(particle%R%z(i)/Lz) * Lz
        enddo

        call forces_from_LJ_potential()

        ! Add open acc directives
        do i = 1, Natoms
            particle%V%vx(i) = particle%V%vx(i) + 0.5_real64 * dt * particle%Fext
↪%Fx(i)
            particle%V%vy(i) = particle%V%vy(i) + 0.5_real64 * dt * particle%Fext
↪%Fy(i)
            particle%V%vz(i) = particle%V%vz(i) + 0.5_real64 * dt * particle%Fext
↪%Fz(i)
        enddo
    end subroutine velocity_verlet

    subroutine berendsen_thermostat()
        integer(kind= INT32) :: i
        ! Add open acc directives
        Ek = 0.0_real64

        ! Add open acc directives
        do i = 1, Natoms
            Ek = Ek + particle%m0 * (particle%V%vx(i)*particle%V%vx(i) + &
                                     particle%V%vy(i)*particle%V%vy(i) + &
                                     particle%V%vz(i)*particle%V%vz(i))
        enddo
```

(continues on next page)

(continued from previous page)

```
        Ek = 0.5_real64 * Ek

        T = 2.0_real64 * kb * Ek / (3.0_real64 * Natoms -3)
        lambda_scaling_factor = sqrt(1 + dt * (-1 + T0/T) / tau_temp)
        ! Add open acc directives
        do i = 1, Natoms
            particle%V%vx(i) = lambda_scaling_factor * particle%V%vx(i)
            particle%V%vy(i) = lambda_scaling_factor * particle%V%vy(i)
            particle%V%vz(i) = lambda_scaling_factor * particle%V%vz(i)
        enddo
        Ek = Ek * lambda_scaling_factor**2
    end subroutine berendsen_thermostat

    subroutine write_config(it)
        real   (kind=REAL64), intent(inout) :: it
        integer(kind= INT32)                :: i
        ! Add open acc directives
        open(unit=20, file='OUTPUT')
        write(20,*) it, Natoms, Lx, Ly, Lz
        ! Add open acc directives
        if (read_restart .eq. 1) then
                ! Add open acc directives
        endif
        if (read_restart .eq. 2) then
                ! Add open acc directives
        endif

        do i = 1, Natoms
            write(20,*) particle%R%x(i), particle%R%y(i), particle%R%z(i)
            if (read_restart .eq. 1) write(20,*) particle%V%vx(i), particle%V%vy(i),␣
↪particle%V%vz(i)
            if (read_restart .eq. 2) write(20,*) particle%Fext%Fx(i), particle%Fext
↪%Fy(i), particle%Fext%Fz(i)
        enddo
        close(20)
    end subroutine write_config

    subroutine read_config(it)
        real   (kind=REAL64), intent(out) :: it
        integer(kind= INT32)              :: i
        open(unit=20, file='CONFIG')
        read(20,*) it, Natoms, Lx, Ly, Lz
        call allocate_variables()
        ! Add open acc directives
        do i = 1, Natoms
            read(20,*) particle%R%x(i), particle%R%y(i), particle%R%z(i)
            if (read_restart .eq. 1) then
                   read(20,*) particle%V%vx(i), particle%V%vy(i), particle%V%vz(i)
            endif
            if (read_restart .eq. 2) then
                   read(20,*) particle%Fext%Fx(i), particle%Fext%Fy(i), particle%Fext
↪%Fz(i)
            endif
        enddo

        ! Add open acc directives
```

(continues on next page)

(continued from previous page)

```
        if (read_restart .eq. 1) then
                ! Add open acc directives
        endif
        if (read_restart .eq. 2) then
                ! Add open acc directives
        endif

        close(20)
        particle%m0 = 1.0
        ! Add open acc directives
    end subroutine read_config

    subroutine allocate_variables
        allocate(particle%R%x(Natoms), particle%R%y(Natoms), particle%R%z(Natoms))
        allocate(particle%V%vx(Natoms), particle%V%vy(Natoms), particle%V%vz(Natoms))
        allocate(particle%Fext%Fx(Natoms), particle%Fext%Fy(Natoms), particle%Fext
 ↪%Fz(Natoms))
        ! Add open acc directives
    end subroutine allocate_variables

    subroutine read_params
        namelist/TIME_CONFIG/ dt, end_time, print_config, print_thermo, write_restart,
 ↪ read_restart
        namelist/LJ_CONFIG/   sigma0, epsilon0
        namelist/BERENDSEN_CONFIG/ T0, tau_temp

        open( 11,file='INPUT_NML',status='old')
        read( 11,NML=TIME_CONFIG)
        read( 11,NML=LJ_CONFIG)
        read( 11,NML=BERENDSEN_CONFIG)
        close(11)
    end subroutine read_params
end module utils

program dm
    use utils

    integer(kind= INT32) :: it, it_print, it_thermo
    real   (kind=REAL64) :: tp

    call read_params()
    call read_config(start_time)
    it_max   = int( (end_time-start_time) /dt)
    it_print = int( print_config/dt)
    it_thermo= int( print_thermo/dt)

    open(unit=42, file="THERMO")
    do it = 1, it_max
        call forces_from_LJ_potential()
        call velocity_verlet()
        call berendsen_thermostat()

        if (mod(it,it_print) .eq. 0) then
            tp = it*dt
            call write_config(tp)
        endif
```

(continues on next page)

(continued from previous page)

```
        if (mod(it,it_thermo) .eq. 0) write(42,*) it*dt, Ek, T
    enddo
    close(42)

    ! Add open acc directives
    deallocate(particle%R%x , particle%R%y , particle%R%z)
    deallocate(particle%V%vx, particle%V%vy, particle%V%vz)
    deallocate(particle%Fext%Fx, particle%Fext%Fy, particle%Fext%Fz)
end program dm
```

19.2 Solution

```
%%idrrun -a
!!  examples/Fortran/Hands_on_LJ_solution.f90
module utils
    use ISO_FORTRAN_ENV, only : REAL64, INT32
    implicit none

    type :: Force
        real(kind=REAL64), dimension(:), allocatable :: Fx, Fy, Fz
    end type
    type :: velocity
        real(kind=REAL64), dimension(:), allocatable :: vx, vy, vz
    end type
    type :: Pos
        real(kind=REAL64), dimension(:), allocatable :: x, y, z
    end type

    type :: atom
        type(Force)        :: Fext
        type(Velocity)     :: V
        type(Pos)          :: R
        real (kind=REAL64) :: m0
    end type

    type(atom)           :: particle
    ! position parameters
    integer(kind= INT32) :: Natoms
    real   (kind=REAL64) :: Lx, Ly, Lz

    ! potential and force parameters
    real   (kind=REAL64) :: epsilon0, sigma0
    real   (kind=REAL64) :: LJ_tolerance = 1e-9
    real   (kind=REAL64) :: rcut = 15.0_real64

    ! temporal parameters
    integer(kind= INT32) :: it_max
    real   (kind=REAL64) :: dt, end_time, print_config, print_thermo, start_time=0.0_
 ↪real64
    integer(kind= INT32) :: write_restart = 0, read_restart = 0

    ! Thermodynamic parameters
    real   (kind=REAL64) :: Ek, T, T0 ! kinetic energy & temperature from simulation,␣
 ↪T0 = targeted temperature
```

(continues on next page)

(continued from previous page)

```
    real   (kind=REAL64), parameter :: kb = 0.831451115

    ! Berendsen thermostat
    real   (kind=REAL64) :: lambda_scaling_factor, tau_temp

    contains
    subroutine forces_from_LJ_potential()
        real   (kind=REAL64) :: rij, xij, yij, zij
        real   (kind=REAL64) :: Fxij, Fyij, Fzij, sigma1, sr2, sr6, sr12
        integer(kind= INT32) :: i, j, divergence = 0
        sigma1   = sigma0*sigma0
        !$acc data present(particle, particle%R, particle%Fext,                    ␣
↪    &
        !$acc              particle%R%x(:) , particle%R%y(:) , particle%R%z(:),    ␣
↪    &
        !$acc              particle%Fext%Fx(:), particle%Fext%Fy(:), particle%Fext
↪%Fz(:) )

        !$acc parallel loop reduction(+:divergence)
        do i = 1, Natoms
            Fxij = 0.0_real64
            Fyij = 0.0_real64
            Fzij = 0.0_real64
            !$acc loop reduction(+:Fxij, Fyij, Fzij, divergence)
            do j = 1, Natoms ! avoid triangular matrix on gpu

            if (i .ne. j) then
                xij = particle%R%x(j)-particle%R%x(i)
                xij = xij - anint(xij/Lx) * Lx
                yij = particle%R%y(j)-particle%R%y(i)
                yij = yij - anint(yij/Lz) * Lz
                zij = particle%R%z(j)-particle%R%z(i)
                zij = zij - anint(zij/Lz) * Lz

                rij = xij*xij + yij*yij + zij*zij
                if ((rij .gt. LJ_tolerance) .and. (rij .le. rcut)) then
                    sr2  = sigma1 / rij
                    sr6  = sr2 *  sr2 * sr2
                    sr12 = sr6 * sr6
                    sr6  = sr6 / rij
                    sr12 = sr12 / rij
                    Fxij = (sr12 - 24_real64 * sr6) * xij + Fxij
                    Fyij = (sr12 - 24_real64 * sr6) * yij + Fyij
                    Fzij = (sr12 - 24_real64 * sr6) * zij + Fzij
                else
                        if (rij .lt. LJ_tolerance) then
                                divergence = divergence + 1  ! avoid early break on␣
↪GPU
                        endif
                endif

            endif
            enddo
            particle%Fext%Fx(i) = 48.0_real64*epsilon0*Fxij
            particle%Fext%Fy(i) = 48.0_real64*epsilon0*Fyij
            particle%Fext%Fz(i) = 48.0_real64*epsilon0*Fzij
```

(continues on next page)

(continued from previous page)

```
        enddo

        if (divergence .ne. 0) then
             write(0,*) 'Particles are too close'
             stop
        endif

        !$acc end data
    end subroutine forces_from_LJ_potential

    subroutine velocity_verlet()
        integer(kind= INT32) :: i
        !$acc data present(particle, particle%R, particle%V, particle%Fext,          ␣
↪    &
        !$acc              particle%R%x(:) , particle%R%y(:) , particle%R%z(:),      ␣
↪    &
        !$acc              particle%V%vx(:), particle%V%vy(:), particle%V%vz(:),     ␣
↪    &
        !$acc              particle%Fext%Fx(:), particle%Fext%Fy(:), particle%Fext
↪%Fz(:) )

        !$acc parallel loop
        do i = 1, Natoms
            particle%V%vx(i) = particle%V%vx(i) + 0.5_real64 * dt * particle%Fext
↪%Fx(i)
            particle%V%vy(i) = particle%V%vy(i) + 0.5_real64 * dt * particle%Fext
↪%Fy(i)
            particle%V%vz(i) = particle%V%vz(i) + 0.5_real64 * dt * particle%Fext
↪%Fz(i)

            particle%R%x(i) = particle%R%x(i) + dt*particle%V%vx(i)
            particle%R%y(i) = particle%R%y(i) + dt*particle%V%vy(i)
            particle%R%z(i) = particle%R%z(i) + dt*particle%V%vz(i)
            particle%R%x(i) = particle%R%x(i) - anint(particle%R%x(i)/Lx) * Lx
            particle%R%y(i) = particle%R%y(i) - anint(particle%R%y(i)/Ly) * Ly
            particle%R%z(i) = particle%R%z(i) - anint(particle%R%z(i)/Lz) * Lz
        enddo

        call forces_from_LJ_potential()

        !$acc parallel loop
        do i = 1, Natoms
            particle%V%vx(i) = particle%V%vx(i) + 0.5_real64 * dt * particle%Fext
↪%Fx(i)
            particle%V%vy(i) = particle%V%vy(i) + 0.5_real64 * dt * particle%Fext
↪%Fy(i)
            particle%V%vz(i) = particle%V%vz(i) + 0.5_real64 * dt * particle%Fext
↪%Fz(i)
        enddo
        !$acc end data
    end subroutine velocity_verlet

    subroutine berendsen_thermostat()
        integer(kind= INT32) :: i
        !$acc data present(particle, particle%V, particle%V%vx(:), particle%V%vy(:),␣
↪particle%V%vz(:))
```

(continues on next page)

(continued from previous page)

```
        Ek = 0.0_real64

        !$acc parallel loop reduction(+:Ek) present(particle%m0)
        do i = 1, Natoms
            Ek = Ek + particle%m0 * (particle%V%vx(i)*particle%V%vx(i) + &
                                     particle%V%vy(i)*particle%V%vy(i) + &
                                     particle%V%vz(i)*particle%V%vz(i))
        enddo
        Ek = 0.5_real64 * Ek

        T = 2.0_real64 * kb * Ek / (3.0_real64 * Natoms -3)
        lambda_scaling_factor = sqrt(1 + dt * (-1 + T0/T) / tau_temp)
        !$acc parallel loop
        do i = 1, Natoms
            particle%V%vx(i) = lambda_scaling_factor * particle%V%vx(i)
            particle%V%vy(i) = lambda_scaling_factor * particle%V%vy(i)
            particle%V%vz(i) = lambda_scaling_factor * particle%V%vz(i)
        enddo
        Ek = Ek * lambda_scaling_factor**2
        !$acc end data
    end subroutine berendsen_thermostat

    subroutine write_config(it)
        real   (kind=REAL64), intent(inout) :: it
        integer(kind= INT32)                :: i
        !$acc data present(particle, particle%R, particle%V, particle%Fext,          ␣
↪    &
        !$acc              particle%R%x(:) , particle%R%y(:) , particle%R%z(:),      ␣
↪    &
        !$acc              particle%V%vx(:), particle%V%vy(:), particle%V%vz(:),     ␣
↪    &
        !$acc              particle%Fext%Fx(:), particle%Fext%Fy(:), particle%Fext
↪%Fz(:) )

        open(unit=20, file='OUTPUT')
        write(20,*) it, Natoms, Lx, Ly, Lz
        !$acc update self(particle%R%x(:) , particle%R%y(:) , particle%R%z(:) )
        if (read_restart .eq. 1) then
            !$acc update self(particle%V%vx(:), particle%V%vy(:), particle%V%vz(:) )
        endif
        if (read_restart .eq. 2) then
            !$acc update self(particle%Fext%Fx(:), particle%Fext%Fy(:), particle%Fext
↪%Fz(:) )
        endif

        do i = 1, Natoms
            write(20,*) particle%R%x(i), particle%R%y(i), particle%R%z(i)
            if (read_restart .eq. 1) write(20,*) particle%V%vx(i), particle%V%vy(i),␣
↪particle%V%vz(i)
            if (read_restart .eq. 2) write(20,*) particle%Fext%Fx(i), particle%Fext
↪%Fy(i), particle%Fext%Fz(i)
        enddo
        close(20)
        !$acc end data
    end subroutine write_config
```

(continues on next page)

(continued from previous page)

```
   subroutine read_config(it)
       real   (kind=REAL64), intent(out) :: it
       integer(kind= INT32)              :: i
       open(unit=20, file='CONFIG')
       read(20,*) it, Natoms, Lx, Ly, Lz
       call allocate_variables()
       !$acc data present(particle, particle%R, particle%V, particle%Fext, particle
↪%m0, &
       !$acc              particle%R%x(:) , particle%R%y(:) , particle%R%z(:),      ␣
↪   &
       !$acc              particle%V%vx(:), particle%V%vy(:), particle%V%vz(:),     ␣
↪   &
       !$acc              particle%Fext%Fx(:), particle%Fext%Fy(:), particle%Fext
↪%Fz(:) )
       do i = 1, Natoms
           read(20,*) particle%R%x(i), particle%R%y(i), particle%R%z(i)
           if (read_restart .eq. 1) then
                  read(20,*) particle%V%vx(i), particle%V%vy(i), particle%V%vz(i)
           endif
           if (read_restart .eq. 2) then
                  read(20,*) particle%Fext%Fx(i), particle%Fext%Fy(i), particle%Fext
↪%Fz(i)
           endif
       enddo

       !$acc update device(particle%R%x, particle%R%y, particle%R%z)
       if (read_restart .eq. 1) then
           !$acc update device(particle%V%vx, particle%V%vy, particle%V%vz)
       endif
       if (read_restart .eq. 2) then
           !$acc update device(particle%Fext%Fx, particle%Fext%Fy, particle%Fext%Fz)
       endif

       close(20)
       particle%m0 = 1.0
       !$acc update device(particle%m0)
       !$acc end data
   end subroutine read_config

   subroutine allocate_variables
       allocate(particle%R%x(Natoms), particle%R%y(Natoms), particle%R%z(Natoms))
       allocate(particle%V%vx(Natoms), particle%V%vy(Natoms), particle%V%vz(Natoms))
       allocate(particle%Fext%Fx(Natoms), particle%Fext%Fy(Natoms), particle%Fext
↪%Fz(Natoms))
       !$acc enter data create(particle)
       !$acc enter data create(particle%R, particle%Fext, particle%V, particle%m0)
       !$acc enter data create(particle%R%x(:)    , particle%R%y(:)    , particle%R
↪%z(:) ,   &
       !$acc                  particle%V%vx(:)   , particle%V%vy(:)   , particle%V
↪%vz(:),   &
       !$acc                  particle%Fext%Fx(:), particle%Fext%Fy(:), particle
↪%Fext%Fz(:) )
   end subroutine allocate_variables

   subroutine read_params
       namelist/TIME_CONFIG/ dt, end_time, print_config, print_thermo, write_restart,
↪ read_restart
```

(continues on next page)

(continued from previous page)

```
        namelist/LJ_CONFIG/   sigma0, epsilon0
        namelist/BERENDSEN_CONFIG/ T0, tau_temp

        open( 11,file='INPUT_NML',status='old')
        read( 11,NML=TIME_CONFIG)
        read( 11,NML=LJ_CONFIG)
        read( 11,NML=BERENDSEN_CONFIG)
        close(11)
    end subroutine read_params
end module utils

program dm
    use utils

    integer(kind= INT32) :: it, it_print, it_thermo
    real   (kind=REAL64) :: tp

    call read_params()
    call read_config(start_time)
    it_max   = int( (end_time-start_time) /dt)
    it_print = int( print_config/dt)
    it_thermo= int( print_thermo/dt)

    open(unit=42, file="THERMO")
    do it = 1, it_max
        call forces_from_LJ_potential()
        call velocity_verlet()
        call berendsen_thermostat()

        if (mod(it,it_print) .eq. 0) then
            tp = it*dt
            call write_config(tp)
        endif
        if (mod(it,it_thermo) .eq. 0) write(42,*) it*dt, Ek, T
    enddo
    close(42)

    !$acc exit data delete(particle%R%x, particle%R%y, particle%R%z,             &
    !$acc                  particle%V%vx, particle%V%vy, particle%V%vz,          &
    !$acc                  particle%Fext%Fx, particle%Fext%Fy, particle%Fext%Fz, &
    !$acc                  particle%R, particle%Fext, particle%V                 )
    !$acc exit data delete(particle)
    deallocate(particle%R%x , particle%R%y , particle%R%z)
    deallocate(particle%V%vx, particle%V%vy, particle%V%vz)
    deallocate(particle%Fext%Fx, particle%Fext%Fy, particle%Fext%Fz)
end program dm
```

Part IV

Resources

CHAPTER

TWENTY

RESOURCES

Most of the resources can be found on the OpenACC website.

20.1 Books

- OpenACC for Programmers: Concepts and Strategies
- Parallel Programming with OpenACC

20.2 Web resources

NVIDIA's training course

Several computing centers offers OpenACC training courses:

- NERSC
- ENCCS
- OpenACC bootcamps training resources

20.3 Porting your code during NVIDIA hackathons

Each year several hackathons and bootcamps are organized by NVIDIA. You can apply for a project and get help from mentors to port your code.

Have a look a this website.

20.4 Contacts (firstname.name@idris.fr)

- Thibaut Véry
- Rémy Dubois
- Olga Abramkina

CHAPTER

TWENTYONE

THE MOST IMPORTANT DIRECTIVES AND CLAUSES

21.1 Directive syntax

```
            Sentinel     Name    Clause(option, ...) ...
C/C++: #pragma acc parallel copyin(array) private(var) ...
Fortran:      !$acc parallel copyin(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 OpenACC directives
- The directive is the action to do. In the example, *parallel* 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 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

21.2 Creating kernels: Compute constructs

Directive	Number of kernels created	Who's in charge?	Comment
`acc parallel`	One for the enclosed region	The developer!	
`acc kernels`	One for each loop nest in the enclosed region	The compiler	
`acc serial`	One for the enclosed region	The developer	Only one thread is used. It is mainly for debug purpose

21.2.1 Clauses

Clause	Available for	Effect
num_gangs(#gangs)	`parallel`, `kernels`	Set the **number of gangs** used by the kernel(s)
num_workers(#workers)	`parallel`, `kernels`	Set the **number of workers** used by the kernel(s)
vector_length(#length)	`parallel`, `kernels`	Set the number of threads in a worker
reduction(op:vars, …)	`parallel`, `kernels`, `serial`	Perform a reduction of *op* kind on *vars*
private(vars, …)	`parallel`, `serial`	Make *vars* private at *gang* level
firstprivate(vars, …)	`parallel`, `serial`	Make *vars* private at *gang* level and initialize the copies with the value that variable originally has on the host

21.3 Managing data

21.3.1 Data regions

Region	Directive
Program lifetime	`acc enter data` & `acc exit data`
Function/Subroutine	`acc declare`
Structured	`acc data`
Kernels	Compute constructs directives

21.3.2 Data clauses

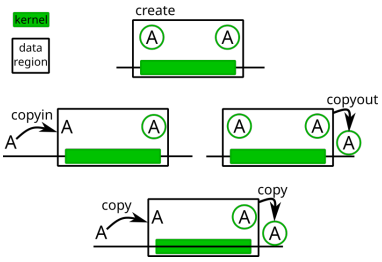
To choose the right data clause you need to answer the following questions:

- 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	copy(var1, …)	copyin(var2, …)
Not needed before	copyout(var3, …)	create(var4, …)

Effects

clause	effect when entering the region	effect when leaving the region
create	**If not already present on the GPU**: allocate the memory needed on the GPU	**If not in another active data region**: free the memory on the GPU
copyin	**If not already present on the GPU**: allocate the memory and initialize the variable with the values it has on CPU	**If not in another active data region**: free the memory
copyout	**If not already present on the GPU**: allocate the memory needed on the GPU	**If not in another active data region**: copy the values from the GPU to the CPU then free the
copy	**If not already present on the GPU**: allocate the memory and initialize the variable with the values it has on CPU	**If not in another active data region**: copy the value
present	Check if data is present: an error is raised if it is not the case	None



21.3.3 Updating data

What to update	Directive
The host (CPU)	`acc update self(vars, …)
The device (GPU)	`acc update device(vars)

21.4 Managing loops

21.4.1 Combined constructs

The `acc loop` directive can be combined with the compute construct directives if there is only one loop nest in the parallel region:

- `acc parallel loop <union of clauses>`
- `acc kernels loop <union of clauses>`
- `acc serial loop <union of clauses>`

21.4.2 Loop clauses

Here are some clauses for the `acc loop` directive:

Clause	Effect
gang	The loop activates work distribution over gangs
worker	The loop activates work distribution over workers
vector	The loop activates work distribution over the threads of the workers
seq	The loop is run sequentially
auto	Let the compiler decide what to do (default)
independent	For `acc kernels`: tell the compiler the loop iterations are independent
collapse(n)	The *n* tightly nested loop are fused in one iteration space
reduction(op:vars, …)	Perform a reduction of *op* kind on *vars*
tile(sizes …)	Create tiles in the iteration space

21.5 GPU routines

You can write a device routine with the `acc routine <max level>` directive: **max_level** is the maximum parallelism level inside the routine including the function calls inside. It can be *gang*, *worker*, *vector*.

21.6 Asynchronous behavior

You can run several streams at the same time on the device using *async(queue)* and *wait* clauses or `acc wait` directive.

Directive	*async(queue)*	*wait(queues,…)*
`acc parallel`	X	X
`acc kernels`	X	X
`acc serial`	X	X
`acc enter data`	X	X
`acc exit data`	X	X
`acc wait`	X	

For the *async* clause, *queue* is an integer specifying the stream on which you enqueue the directive. If omitted a default stream is used.

21.7 Using data on the GPU with GPU aware libraries

To get a pointer to the device memory for a variable you have to use `acc host_data use_device(data)`. Useful for:

- Using GPU libraries (ex. CUDA)
- MPI CUDA-Aware to avoid spurious data transfers

21.8 Atomic construct

To make sure that only one thread performs a read/write on a variable you have to use the `acc atomic <operation>` directive.

operation is one of the following:

- read
- write
- update (read + write)
- capture (update + saving to another variable)