OpenACC for GPU: an introduction

Olga Abramkina, Rémy Dubois, Thibaut Véry

Jun 02, 2023

CONTENTS

I Day 1 **5**

1 Introduction to GPU programming with directives **7**
- 1.1 What is a GPU? . . . 7
- 1.2 Programming models . . . 8
- 1.3 OpenACC . . . 10
- 1.4 OpenMP target . . . 10
- 1.5 Host driven Language . . . 11
- 1.6 Levels of parallelism . . . 11
- 1.7 Important notes . . . 12

2 Get started with OpenACC **13**
- 2.1 OpenACC directives . . . 13
- 2.2 Loops parallelism . . . 16
- 2.3 Managing data in compute regions . . . 17
- 2.4 Exercise: Gaussian blurring filter . . . 18
- 2.5 Reductions with OpenACC . . . 23

3 Manual building of an OpenACC code **27**
- 3.1 Build with NVIDIA compilers . . . 27
- 3.2 Build with GCC compilers . . . 29
- 3.3 Exercise . . . 30

4 Data management **31**
- 4.1 Why do we have to care about data transfers? . . . 31
- 4.2 The easy way: NVIDIA managed memory . . . 32
- 4.3 Manual data movement . . . 32

5 Hands-on Game Of Life **43**
- 5.1 What to do . . . 44
- 5.2 Solution . . . 48

II Day 2 **53**

6 Compute constructs **55**
- 6.1 Giving more freedom to the compiler: `acc kernels` . . . 55
- 6.2 Running sequentially on the GPU? The `acc serial` compute construct . . . 56
- 6.3 Data region associated with compute constructs . . . 57

7 Variables status (private or shared) **59**
- 7.1 Default status of scalar and arrays . . . 59

7.2 Private variables . . . 59
7.3 Caution . . . 61

8 Advanced loop configuration **63**
8.1 Syntax . . . 63
8.2 Restrictions . . . 64
8.3 Example . . . 64
8.4 Exercise . . . 65

9 Using OpenACC in modular programming **67**
9.1 `acc routine <max_level_of_parallelism>` . . . 67
9.2 Named `acc routine(name) <max_level_of_parallelism>` . . . 68
9.3 Directives inside an `acc routine` . . . 69
9.4 Exercise . . . 69

10 Profiling your code to find what to offload **73**
10.1 Development cycle . . . 73
10.2 Quick description of the code . . . 73
10.3 Profiling CPU code . . . 74
10.4 The graphical profiler . . . 74
10.5 Profiling GPU code: other tools . . . 77

11 Multi GPU programming with OpenACC **79**
11.1 Disclaimer . . . 79
11.2 Introduction . . . 79
11.3 API description . . . 79
11.4 MPI strategy . . . 79
11.5 Multithreading strategy . . . 80
11.6 Exercise . . . 81
11.7 GPU to GPU data transfers . . . 81

12 Generate Mandelbrot set **87**
12.1 Introduction . . . 87
12.2 What to do . . . 88
12.3 Solution . . . 90

13 Generate Mandelbrot set **93**
13.1 Introduction . . . 93
13.2 What to do . . . 94
13.3 Solution . . . 96

III Day 3 **99**

14 Performing several tasks at the same time on the GPU **103**
14.1 *async* clause . . . 103
14.2 *wait* clause . . . 104
14.3 *wait* directive . . . 105
14.4 Exercise . . . 105
14.5 Advanced NVIDIA compiler option to use Pinned Memory: `-gpu=pinned` . . . 108

15 Atomic operations **109**
15.1 Syntax . . . 109
15.2 Restrictions . . . 110
15.3 Exercise . . . 111

16 Deep copy **115**
16.1 Top-down deep copy . . . 115
16.2 Deep copy with manual attachment . . . 124

17 Using CUDA libraries **131**
17.1 `acc host_data use_device` . . . 131
17.2 Example with CURAND . . . 131

18 Loop tiling **135**
18.1 Syntax . . . 136
18.2 Restrictions . . . 136
18.3 Example . . . 136
18.4 Exercise . . . 142
18.5 Solution . . . 143

19 Hands-on MD simulation of Lennard-Jones system **145**
19.1 What to do . . . 145
19.2 Solution . . . 154

IV Resources **165**

20 Resources **167**
20.1 Books . . . 167
20.2 Web resources . . . 167
20.3 Porting your code during NVIDIA hackathons . . . 167
20.4 Contacts (firstname.name@idris.fr) . . . 167

21 The most important directives and clauses **169**
21.1 Directive syntax . . . 169
21.2 Creating kernels: Compute constructs . . . 169
21.3 Managing data . . . 170
21.4 Managing loops . . . 171
21.5 GPU routines . . . 172
21.6 Asynchronous behavior . . . 172
21.7 Using data on the GPU with GPU aware libraries . . . 172
21.8 Atomic construct . . . 173

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:

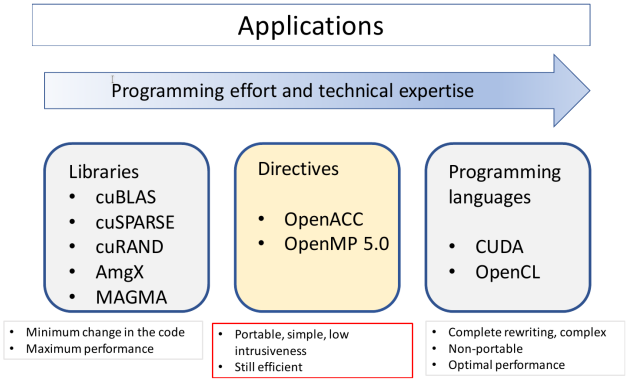
Core Type	Number per SM
FP32	128
FP64	64
INT32	64
TensorCore	4
Total	176

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:

```
#pragma acc parallel loop
for (int i=0; i<size; ++i)
{
    // Code to offload to GPU
}
```

For OpenMP target:

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

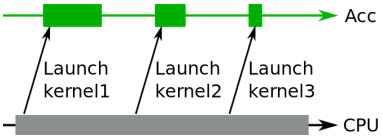
The main compilers which support offloading to GPU are:

- 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

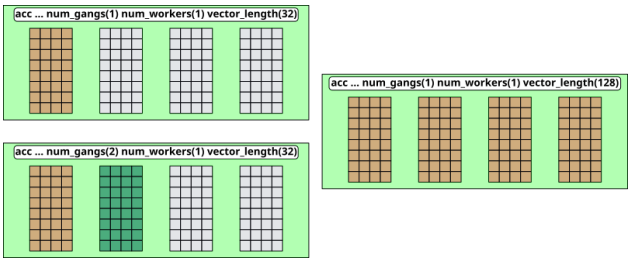
On the GPU you can have 4 different levels of parallelism that can be activated:

- 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.

```
#pragma acc parallel
{
    // code block opened with '{' and closed by '}'
}
```

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 C, if you want to run the code, you need to define the `main` function:

```
int main(void)
{
//
}
```

or:

```
int main(int argc, char** argv)
{
//
}
```

The example initializes an array of integers.

```
%%idrrun
//  examples/C/Get_started_init_array_exercise.c
#include <stdio.h>
int main(void)
{
    int size = 100000;
    int array[size];
    for (int i=0; i<size; ++i)
        array[i] = 2 * i;
    printf("%d", array[21]);
}
```

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/C/Get_started_init_array_exercise_acc.c
#include <stdio.h>
int main(void)
{
    int size = 100000;
    int array[size];
    // Modifications from here
    for (int i=0; i<size; ++i)
        array[i] = 2 * i;
    printf("%d", array[12]);
}
```

2.1.2 Solution

```
%%idrrun -a
//  examples/C/Get_started_init_array_solution_acc.c
#include <stdio.h>
int main(void)
{
    int size = 100000;
    int array[size];
    #pragma acc parallel
    {
    for (int i=0; i<size; ++i)
        array[i] = 2 * i;
    }
    printf("%d", array[12]);
}
```

Which is equivalent to:

```
%%idrrun -a
//  examples/C/Get_started_init_array_solution_acc_2.c
#include <stdio.h>
int main(void)
{
    int size = 100000;
    int array[size];
    #pragma acc parallel
    {
    #pragma acc loop
    for (int i=0; i<size; ++i)
        array[i] = 2 * i;
    }
    printf("%d", array[12]);
}
```

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:

```
#pragma acc loop
```

2.2.2 Non independent loops

Here are some cases where the iterations are not independent:

- Infinite loops

```
while(error > tolerance)
{
    //compute error
}
```

- Current iteration reads values computed by previous iterations

```
array[0] = 0;
array[1] = 1;
for (int i = 2; i<size; ++i)
    array[i] = array[i-1]+array[i-2];
```

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

```
for (int i=0; i< size-1; ++i)
    array[i] = array[i+1] + 1
```

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

```
for (int i = 0; i<size-1; ++i)
{
    array[i]++;
    array[i+1] = array[i]+2;
}
```

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

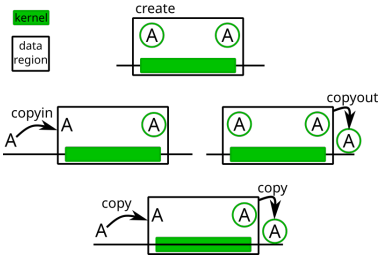
2.3 Managing data in compute regions

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.

```
#pragma acc parallel copy(var1[first_index:num_elements]) copyin(var2[first_index_
↪i:num_elements_i][first_index_j:num_elements_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/C/blur_simple_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

(continues on next page)

(continued from previous page)

```
#include <time.h>
/**
 * Apply a gaussian blurring filter to a picture generated on the fly
 *
 * List of functions:
 *   - void blur(unsigned char* pic,  unsigned char* blurred, size_t rows, size_t␣
↪cols)
 *     the actual filter
 *   - void fill(unsigned char* pic, size_t rows, size_t cols)
 *     generate the original picture
 *   - void out_pic(unsigned char* pic, char* name, size_t rows, size_t cols)
 *     create a .rgb file
 */

void blur(unsigned char* pic,  unsigned char* blurred, size_t rows, size_t cols)
{
    /**
     * Perform the blurring of the picture
     * @ param pic(in): a pointer to the original picture
     * @ param blurred(out): a pointer to the blurred picture
     */
   size_t i, j, l, i_c, j_c;
   unsigned int pix;
   unsigned char coefs[5][5] = { {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}};

   for (i=2; i<rows-2; ++i)
      for (j=2; j<cols-2; ++j)
         for (l=0; l<3; ++l)
         {
            pix = 0;
            for (i_c=0; i_c<5; ++i_c)
                for (j_c=0; j_c<5; ++j_c)
                   pix += (pic[(i+i_c-2)*3*cols+(j+j_c-2)*3+l]
                           *coefs[i_c][j_c]);

             blurred[i*3*cols+j*3+l] = (unsigned char)(pix/256);
         }
}

void fill(unsigned char* pic, size_t rows, size_t cols)
{
    /**
     * Fill the picture with data
     * @param pic(out): a pointer to the pixel to be blurred
     * @param rows(in) the number of rows in the picture
     * @param cols(in) the number of columns in the picture
     *
     */

    size_t i, j;
   for (i=0; i < rows; ++i)
      for (j=0; j < 3*cols; ++j)
```

(continues on next page)

(continued from previous page)

```
          pic[i*3*cols+j] = (unsigned char) (i+(j%3)*j+i%256)%256;


}

void out_pic(unsigned char* pic, char* name, size_t rows, size_t cols)
{
    /**
     * Output of the picture into a sequence of pixel
     * Use show_rgb(filepath, rows, cols) to display
     * @param rows(in) the number of rows in the picture
     * @param cols(in) the number of columns in the picture
     */
   FILE* f = fopen(name, "wb");
   fwrite(pic, sizeof(unsigned char), rows*3*cols, f);
   fclose(f);
}

int main(void)
{
   size_t rows,cols;

   rows = 4000;
   cols = 4000;

   printf("Size of picture is %d x %d\n", rows, cols);
   unsigned char* pic = (unsigned char*) malloc(rows*3*cols*sizeof(unsigned char));
   unsigned char* blurred_pic = (unsigned char*) malloc(rows*3*cols*sizeof(unsigned␣
 ↪char));

   // Create the original picture
   fill(pic, rows, cols);

   // Apply the blurring filter
   blur(pic, blurred_pic, rows, cols);

   out_pic(pic, "pic.rgb", rows, cols);
   out_pic(blurred_pic, "blurred.rgb", rows, cols);

   free(pic);
   free(blurred_pic);

   return 0;
}
```

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

2.4.1 Solution

```
%%idrrun -a
//  examples/C/blur_simple_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
/**
 * Apply a gaussian blurring filter to a picture generated on the fly
 *
 * List of functions:
 *   - void blur(unsigned char* pic,  unsigned char* blurred, size_t rows, size_t␣
↪cols)
 *     the actual filter
 *   - void fill(unsigned char* pic, size_t rows, size_t cols)
 *     generate the original picture
 *   - void out_pic(unsigned char* pic, char* name, size_t rows, size_t cols)
 *     create a .rgb file
 */

void blur(unsigned char* pic,  unsigned char* blurred, size_t rows, size_t cols)
{
    /**
     * Perform the blurring of the picture
     * @ param pic(in): a pointer to the original picture
     * @ param blurred(out): a pointer to the blurred picture
     */
   size_t i, j, l, i_c, j_c;
   unsigned int pix;
   unsigned char coefs[5][5] = { {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}};

#pragma acc parallel loop copyin(coefs[:5][:5],pic[:3*rows*cols])␣
↪copyout(blurred[:3*rows*cols])
   for (i=2; i<rows-2; ++i)
      for (j=2; j<cols-2; ++j)
         for (l=0; l<3; ++l)
         {
            pix = 0;
            for (i_c=0; i_c<5; ++i_c)
                for (j_c=0; j_c<5; ++j_c)
                   pix += (pic[(i+i_c-2)*3*cols+(j+j_c-2)*3+l]
                           *coefs[i_c][j_c]);

             blurred[i*3*cols+j*3+l] = (unsigned char)(pix/256);
         }
}
void fill(unsigned char* pic, size_t rows, size_t cols)
{
    /**
     * Fill the picture with data
     * @param pic(out): a pointer to the pixel to be blurred
     * @param rows(in) the number of rows in the picture
```

(continues on next page)

(continued from previous page)

```
     * @param cols(in) the number of columns in the picture
     *
     */

    size_t i, j;
#pragma acc parallel loop copyout(pic[0:3*rows*cols])
   for (i=0; i < rows; ++i)
      for (j=0; j < 3*cols; ++j)
         pic[i*3*cols+j] = (unsigned char) (i+(j%3)*j+i%256)%256;


}

void out_pic(unsigned char* pic, char* name, size_t rows, size_t cols)
{
    /**
     * Output of the picture into a sequence of pixel
     * Use show_rgb(filepath, rows, cols) to display
     * @param rows(in) the number of rows in the picture
     * @param cols(in) the number of columns in the picture
     */
   FILE* f = fopen(name, "wb");
   fwrite(pic, sizeof(unsigned char), rows*3*cols, f);
   fclose(f);
}

int main(void)
{
   size_t rows,cols;

   rows = 4000;
   cols = 4000;

   printf("Size of picture is %d x %d\n", rows, cols);
   unsigned char* pic = (unsigned char*) malloc(rows*3*cols*sizeof(unsigned char));
   unsigned char* blurred_pic = (unsigned char*) malloc(rows*3*cols*sizeof(unsigned␣
↪char));

   // Create the original picture
   fill(pic, rows, cols);

   // Apply the blurring filter
   blur(pic, blurred_pic, rows, cols);

   out_pic(pic, "pic.rgb", rows, cols);
   out_pic(blurred_pic, "blurred.rgb", rows, cols);

   free(pic);
   free(blurred_pic);

   return 0;
}
```

```
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:

```
for (int i=0; i<size_array; ++i)
    sum += array[i];
```

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:

- `#pragma acc loop reduction(op:var1)`
- `#pragma acc parallel reduction(op:var1)`
- `#pragma acc kernels reduction(op:var1)`
- `#pragma 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:

```
#pragma acc parallel loop reduction(+:var1, var2,...)
```

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

```
#pragma 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/C/reduction_exponential_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
int main(void)
{
    // current position and value
    double x,y,x_p;
    // Number of divisions of the function
    int nsteps = 1e9;
    // x min
    double begin = 0.;
    // x max
    double end = M_PI;
    // Sum of elements
    double sum = 0.;
    // Length of the step
    double step_l = (end-begin)/nsteps;

    double dmin = DBL_MAX;
    double dmax = DBL_MIN;
    for (int i=0 ; i < nsteps ; ++i )
    {
        x = i*step_l;
        x_p = (i+1)*step_l;
        y = (exp(x)+exp(x_p))/2;
        sum += y;
        if (y < dmin)
            dmin = y;
        if (y > dmax)
            dmax = y;
    }
    // Print the stats
    printf("The MINimum value of the function is: %f\n",dmin);
    printf("The MAXimum value of the function is: %f\n",dmax);
    printf("The integral of the function on [%f,%f] is: %f\n",begin,end,sum*step_l);
    printf("   difference is: %5.2e",exp(end)-exp(begin)-sum*step_l);
}
```

Solution

```
%%idrrun -a
//  examples/C/reduction_exponential_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
int main(void)
{
    // current position and value
    double x,y,x_p;
    // Number of divisions of the function
    int nsteps = 1e9;
    // x min
    double begin = 0.;
    // x max
    double end = M_PI;
    // Sum of elements
    double sum = 0.;
    // Length of the step
    double step_l = (end-begin)/nsteps;

    double dmin = DBL_MAX;
    double dmax = DBL_MIN;
#pragma acc parallel loop reduction(+:sum) reduction(min:dmin) reduction(max:dmax)
    for (int i=0 ; i < nsteps ; ++i )
    {
        x = i*step_l;
        x_p = (i+1)*step_l;
        y = (exp(x)+exp(x_p))/2;
        sum += y;
        if (y < dmin)
            dmin = y;
        if (y > dmax)
            dmax = y;
    }
    // Print the stats
    printf("The MINimum value of the function is: %f\n",dmin);
    printf("The MAXimum value of the function is: %f\n",dmax);
    printf("The integral of the function on [%f,%f] is: %f\n",begin,end,sum*step_l);
    printf("   difference is: %5.2e",exp(end)-exp(begin)-sum*step_l);
}
```

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:

```
nvc -acc=gpu,noautopar -gpu=cc70,managed -Minfo=all mysource.c -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
    nvc -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.c
    gcc -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/C/Manual_building_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void inplace_sum(double* A, double* B, size_t size)
{
    #pragma acc parallel loop present(A[0:size], B[0:size])
    for (size_t i=0; i<size; ++i)
        A[i] += B[i];
}

int main(void)
{
    size_t size = (size_t) 1e9;
    double* A = (double*) malloc(size*sizeof(double));
    double* B = (double*) malloc(size*sizeof(double));
    double sum = 0.0;

    #pragma acc data create(A[0:size], B[0:size])
    {
        #pragma acc parallel loop present(A[0:size], B[0:size])
        for (size_t i=0; i<size; ++i)
        {
            A[i] = sin(M_PI*(double)i/(double)size)*sin(M_PI*(double)i/(double)size);
            B[i] = cos(M_PI*(double)i/(double)size)*cos(M_PI*(double)i/(double)size);
        }

        inplace_sum(A, B, size);

        #pragma acc parallel loop present(A[0:size], B[0:size]) reduction(+:sum)
        for (size_t i=0; i<size; ++i)
            sum += A[i];
    }
    printf("This should be close to 1.0: %f\n", sum/(double) size);
    return 0;
}
```

CHAPTER

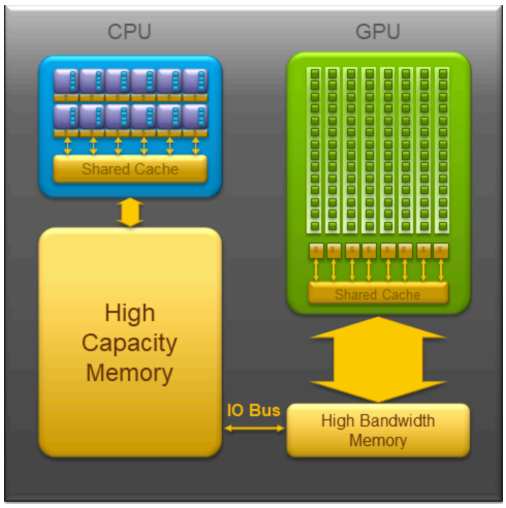
FOUR

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

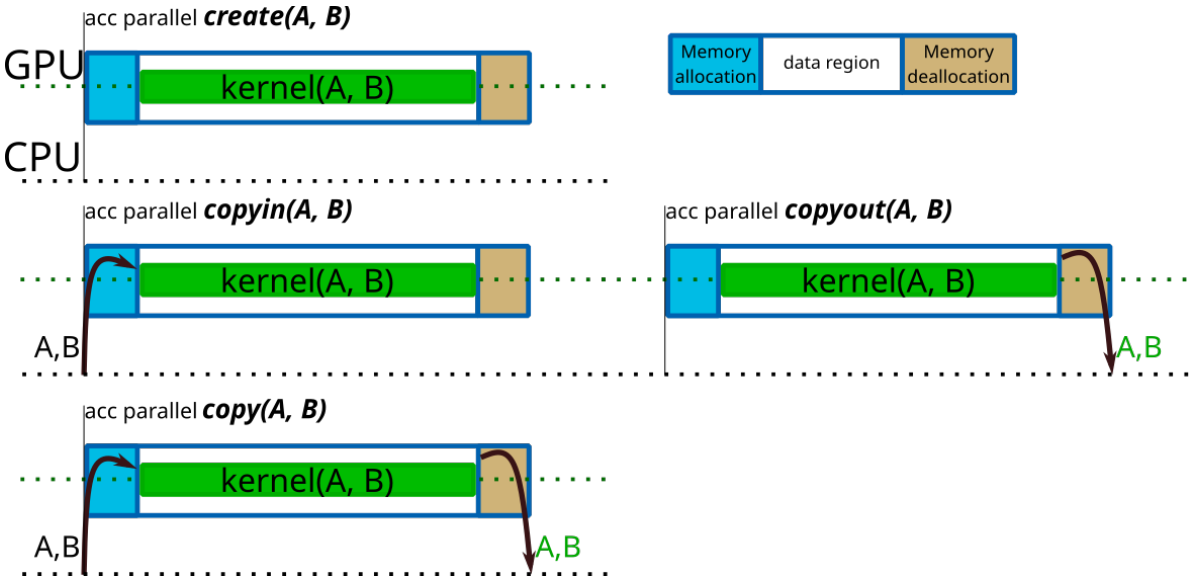
There are multiple data directives that accept the same data clauses. So we start with the data clauses and than continue with data directives.

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 C, you have to specify the range of the array in the format `[first index:number of elements]`.

```
#pragma acc data copyout(myarray[0:size])
// In C, 0 is the first index by default and can be omitted
// the previous line is equivalent to
// #pragma acc data copyout(myarray[:size])
{
   // Some really fast kernels
}
```

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

```
#pragma acc data copyout(myarray[2:size-2])
```

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/C/Data_Management_vector_sum_exercise.c
#include <stdio.h>
#include <stdlib.h>
int main(void){
```

(continues on next page)

(continued from previous page)

```
    int size = 10000;
    int a[size], b[size], c[size];

    // Insert OpenACC directive
    for(int i=0; i<size; ++i){
        a[i] = i;
        b[i] = 2*i;
    }

    // Insert OpenACC directive
    for (int i=0; i<size; ++i){
        c[i] = a[i]+b[i];
    }

    printf("value at position 14: %d\n", c[14]);
}
```

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/C/Data_Management_vector_sum_solution.c
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int size = 10000;
    int a[size], b[size], c[size];

    #pragma acc parallel loop copyout(a, b)
    for(int i=0; i<size; ++i){
        a[i] = i;
        b[i] = 2*i;
    }

    #pragma acc parallel loop copyin(a, b) copyout(c)
    for (int i=0; i<size; ++i){
        c[i] = a[i]+b[i];
    }

    printf("value at position 14: %d\n", c[14]);
}
```

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:

```
#pragma acc data <data clauses>
{
    // Your code
}
```

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/C/Data_Management_structured_data_region_exercise.c
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int size = 10000;
    int a[size], b[size], c[size];

    {
        #pragma acc parallel loop present(a, b)
        for(int i=0; i<size; ++i){
            a[i] = i;
            b[i] = 2*i;
        }

        #pragma acc parallel loop present(a, b, c)
        for (int i=0; i<size; ++i){
            c[i] = a[i]+b[i];
        }
    }
    printf("value at position 14: %d\n", c[14]);
}
```

Solution

```
%%idrrun -a
//  examples/C/Data_Management_structured_data_region_solution.c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int size = 10000;
    int a[size], b[size], c[size];

// Structured data region
    #pragma acc data create(a, b) copyout(c)
    {
        #pragma acc parallel loop present(a, b)
        for(int i=0; i<size; ++i){
            a[i] = i;
            b[i] = 2*i;
        }

        #pragma acc parallel loop present(a, b, c)
        for (int i=0; i<size; ++i){
            c[i] = a[i]+b[i];
        }
    } // End of structured data region
    printf("value at position 14: %d\n", c[14]);
}
```

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/C/Data_Management_wrong_example.c
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int size = 10000;
    int a[size], b[size], c[size];

// Structured data region
    #pragma acc data create(a, b) copyout(c)
    {
        #pragma acc parallel loop present(a, b)
        for(int i=0; i<size; ++i){
            a[i] = i;
            b[i] = 2*i;
        }
        // We update an element of the array on the CPU
        a[14] = 78324;

        #pragma acc parallel loop present(b, c) copyin(a)
```

(continues on next page)

(continued from previous page)

```
        for (int i=0; i<size; ++i){
            c[i] = a[i]+b[i];
        }
    }
    printf("value at position 14: %d\n", c[14]);
}
```

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:

```
#pragma 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:

```
#pragma acc update self(var1, var2, ...)
```

Correct the previos example in order to obtain correct restuls:

```
%%idrrun -a
//  examples/C/Data_Management_wrong_example.c
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int size = 10000;
    int a[size], b[size], c[size];

// Structured data region
    #pragma acc data create(a, b) copyout(c)
    {
        #pragma acc parallel loop present(a, b)
        for(int i=0; i<size; ++i){
            a[i] = i;
            b[i] = 2*i;
        }
        // We update an element of the array on the CPU
```

(continues on next page)

(continued from previous page)

```
        a[14] = 78324;

        #pragma acc parallel loop present(b, c) copyin(a)
        for (int i=0; i<size; ++i){
            c[i] = a[i]+b[i];
        }
    }
    printf("value at position 14: %d\n", c[14]);
}
```

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:

```
#pragma acc enter data copyin(var1[:size_var1], ...) create(var2[0:size_var2])
```

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

```
double* var = (double*) malloc(size_var*sizeof(double));
#pragma acc enter data create(var[0:size_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.

```
#pragma acc exit data delete(var[0:size_var])
free(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/C/Data_Management_unstructured_exercise.c
#include <stdio.h>
#include <stdlib.h>
double* init(size_t size){
    double* array = (double*) malloc(size*sizeof(double));
    return array;
}

int main(void){
    size_t size = 100000;
    double* array = init(size);
    for (size_t i=0; i<size; ++i)
        array[i] = (double)i;
    printf("This should be 42: %f\n", array[42]);
    free(array);
}
```

Solution

```
%%idrrun -a
//  examples/C/Data_Management_unstructured_solution.c
#include <stdio.h>
#include <stdlib.h>
double* init(size_t size){
    double* array = (double*) malloc(size*sizeof(double));
    #pragma acc enter data create(array[0:size])
    return array;
}

int main(void){
    size_t size = 100000;
    double* array = init(size);
    #pragma acc parallel loop present(array[0:size])
    for (size_t i=0; i<size; ++i)
        array[i] = i;
    #pragma acc exit data delete(array[0:size])
    printf("%f\n", array[42]);
    free(array);
}
```

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.

```
int size = 1000000;
double* array;
#pragma acc declare create(array[0: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/C/Data_Management_unstructured_declare_example.c
#include <stdio.h>
#include <stdlib.h>

void normalize_rows(double* mat, size_t size)
{
    double norms[size];
```

(continues on next page)

(continued from previous page)

```
    #pragma acc declare device_resident(norms)
    double norm;
    // Compute the L1 norm of each row
    #pragma acc parallel loop present(mat[0:size*size])
    for (size_t i=0; i<size; ++i)
    {
        norm = 0.;
        #pragma acc loop reduction(+:norm)
        for (size_t j=0; j<size; ++j)
            norm += mat[i*size+j];
        norms[i] = norm;
    }
    // Divide each row element by the L1 norm
    #pragma acc parallel loop present(mat[0:size*size])
    for (size_t i=0; i<size; ++i)
        for (size_t j=0; j<size; ++j)
            mat[i*size+j] /= norms[i];
}

int main(void)
{
    size_t size = 2000;
    double* mat = malloc(size*size*sizeof(double));
    double sum = 0.;
    srand((unsigned) 12345900);
    for (size_t i=0; i<size; ++i)
        for (size_t j=0; j<size; ++j)
            mat[i*size+j] = (double)rand() / (double) RAND_MAX;
    #pragma acc enter data copyin(mat[0:size*size])

    normalize_rows(mat, size);

    // Compute the sum of all elements in the matrix
    #pragma acc parallel loop present(mat[0:size*size]) reduction(+:sum)
    for (size_t i=0; i<size; ++i)
        for (size_t j=0; j<size; ++j)
            sum += mat[i*size+j];
    #pragma acc exit data delete(mat[0:size*size])
    free(mat);

    printf("%f == %d?\n", sum, size);

    return 0;
}
```

Requirements:

- Get started
- Data management

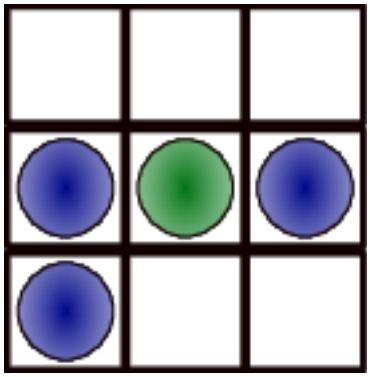
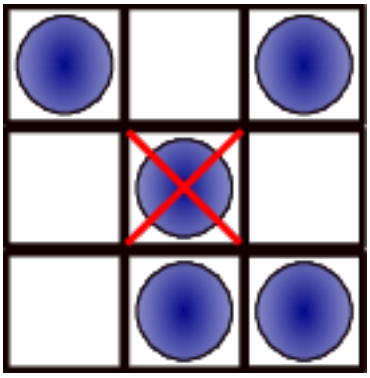
CHAPTER

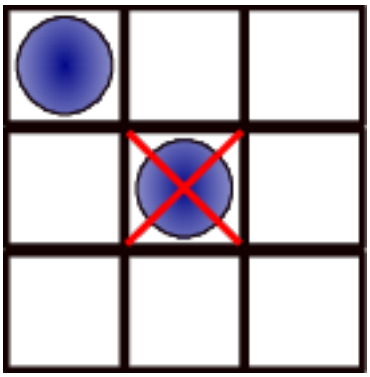
FIVE

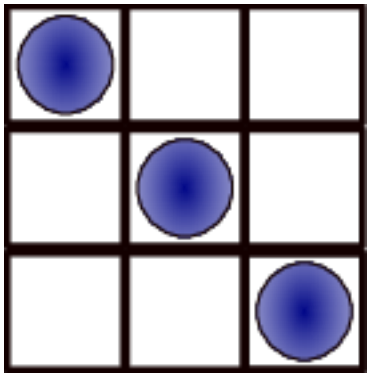
HANDS-ON GAME OF LIFE

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/C/GameOfLife_exercise.c
#include <stdio.h>
#include <stdlib.h>

void output_world(int* restrict world, int rows, int cols, int generation)
{
    /**
     * Write a file with the world inside
     * @param world: a pointer to the storage for the current step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    char* path = (char*) malloc(sizeof(char)*80);
    sprintf(path, "generation%05d.gray", generation);
    FILE* f = fopen(path, "wb");
    unsigned char* mat = (unsigned char*) malloc(sizeof(unsigned␣
 ↪char)*(rows+2)*(cols+2));
```

(continues on next page)

(continued from previous page)

```
    for (int i=0; i<rows+2; ++i)
        for (int j=0; j<cols+2; ++j)
            mat[i*cols+j] = (unsigned char) world[i*cols+j] * 255;
    fwrite(world, sizeof(unsigned char), (rows+2)*(cols+2), f);
    fclose(f);
}

void next(int* restrict world, int* restrict  oworld, int rows, int cols)
{
    /**
     * Apply the rules and compute the next generation
     * @param world: a pointer to the storage for the current step
     * @param oworld: a pointer to the storage for the previous step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    int neigh = 0;
    int row_current = 0;
    int row_above = 0;
    int row_below = 0;
    for (int r=1; r<=rows; ++r)
        for (int c=1; c<=cols; ++c)
        {
            row_current = r*(cols+2);
            row_above = (r-1)*(cols+2);
            row_below = (r+1)*(cols+2);
            neigh = oworld[row_above + c-1] + oworld[row_above + c]   + oworld[row_
 ↪above + c+1] +
                    oworld[row_current + c+1]+                        oworld[row_
 ↪current + c-1] +
                    oworld[row_below + c-1] + oworld[row_below + c] + oworld[row_
 ↪below + c+1];
            if (oworld[r*(cols+2)+c] == 1 && (neigh<2||neigh>3))
                world[r*(cols+2)+c] = 0;
            else if (neigh==3)
                world[r*(cols+2)+c] = 1;
        }
}

void save(int* restrict world, int* restrict oworld, int rows, int cols)
{
    /**
     * Save the current world to oworld
     * @param world: a pointer to the storage for the current step
     * @param oworld: a pointer to the storage for the previous step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    for (int r=1; r<=rows; ++r)
        for (int c=1; c <= cols; ++c)
            oworld[r*(cols+2) + c] = world[r*(cols+2) + c];
}

int alive(int* restrict world, int rows, int cols)
{
    /**
```

(continues on next page)

(continued from previous page)

```
     * Compute the number of cells alive at the current generation
     * @param world: a pointer to the storage for the current step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    int cells = 0;
    for (int r=1; r <= rows; ++r)
        for (int c=1; c <= cols; ++c)
            cells += world[r*(cols+2) + c];
    return cells;
}

void fill_world(int* restrict world, int rows, int cols)
{
    /**
     *  Set the initial state of the world
     * @param world: a pointer to the storage for the current step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    for (int r=1; r <= rows; ++r)
        for (int c=1; c <= cols; ++c)
            world[r*(cols+2) + c] = rand()%4==0 ?1 : 0;
    // The border of the world is a dead zone
    for (int i=0;i<=rows;++i)
    {
        world[i*(cols+2)] = 0;
        world[i*(cols+2)+cols+1] = 0;
    }
    for (int j=0; j<cols; ++j)
    {
        world[j] = 0;
        world[(rows+1)*(cols+2)+j] = 0;
    }
}

int* allocate(int rows, int cols)
{
    /**
     * Allocate memory for a 2D array
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     * @return a pointer to the matrix
     */

    int* mat = (int*) malloc((rows+2)*(cols+2)*sizeof(int));

    return mat;
}

void destroy(int* mat, int rows, int cols)
{
    /**
     * Free memory for a 2D array
     * @param mat: a pointer to the matrix to free
     * @param rows: the number of rows without the border
```

(continues on next page)

(continued from previous page)

```
     * @param cols: the number of columns without the border
     */
    free(mat);
}

int main(int argc, char** argv)
{
    int rows, cols, generations;
    int* world;
    int* oworld;

    if (argc < 4)
    {
        printf("Wrong number of arguments: Please give rows cols and generations\n");
        return 1;
    }
    rows = strtol(argv[1], NULL, 10);
    cols = strtol(argv[2], NULL, 10);
    generations = strtol(argv[3], NULL, 10);

    world = allocate(rows, cols);
    oworld = allocate(rows, cols);
    fill_world(world, rows, cols);
    printf("Initial state set\n");
    printf("Cells alive at generation %d: %d\n", 0, alive(world, rows, cols));
    for (int g=1; g <= generations; ++g)
    {
        save(world, oworld, rows, cols);
        next(world, oworld, rows, cols);
        output_world(world, rows, cols, g);
        printf("Cells alive at generation %4d: %d\n", g, alive(world, rows, cols));
    }

    destroy(world, rows, cols);
    destroy(oworld, rows, cols);

    return 0;
}
```

```
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")
```

(continues on next page)

(continued from previous page)

```
    plt.show()
interact(view, i=(0, len(images)-1))
```

5.2 Solution

```
%%idrrun  -a --cliopts "20000 1000 300"
//  examples/C/GameOfLife_solution.c
#include <stdio.h>
#include <stdlib.h>

void output_world(int* restrict world, int rows, int cols, int generation)
{
    /**
     * Write a file with the world inside
     * @param world: a pointer to the storage for the current step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    char* path = (char*) malloc(sizeof(char)*80);
    sprintf(path, "generation%05d.gray", generation);
    FILE* f = fopen(path, "wb");
    unsigned char* mat = (unsigned char*) malloc(sizeof(unsigned␣
 ↪char)*(rows+2)*(cols+2));
    #pragma acc parallel loop copyout(mat[(rows+2)*(cols+2)])␣
 ↪present(world[(rows+2)*(cols+2)])
    for (int i=0; i<rows+2; ++i)
        for (int j=0; j<cols+2; ++j)
            mat[i*cols+j] = (unsigned char) world[i*cols+j] * 255;
    fwrite(world, sizeof(unsigned char), (rows+2)*(cols+2), f);
    fclose(f);
}
void next(int* restrict world, int* restrict  oworld, int rows, int cols)
{
    /**
     * Apply the rules and compute the next generation
     * @param world: a pointer to the storage for the current step
     * @param oworld: a pointer to the storage for the previous step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    int neigh = 0;
    int row_current = 0;
    int row_above = 0;
    int row_below = 0;
#pragma acc parallel loop present(world[:(rows+2)*(cols+2)],␣
 ↪oworld[:(rows+2)*(cols+2)])
    for (int r=1; r<=rows; ++r)
        for (int c=1; c<=cols; ++c)
        {
            row_current = r*(cols+2);
            row_above = (r-1)*(cols+2);
            row_below = (r+1)*(cols+2);
            neigh = oworld[row_above + c-1] + oworld[row_above + c]   + oworld[row_
 ↪above + c+1] +
```

(continues on next page)

(continued from previous page)

```
                    oworld[row_current + c+1]+                          oworld[row_
 ↪current + c-1] +
                    oworld[row_below + c-1] + oworld[row_below + c] + oworld[row_
 ↪below + c+1];
            if (oworld[r*(cols+2)+c] == 1 && (neigh<2||neigh>3))
                world[r*(cols+2)+c] = 0;
            else if (neigh==3)
                world[r*(cols+2)+c] = 1;
        }
}

void save(int* restrict world, int* restrict oworld, int rows, int cols)
{
    /**
     * Save the current world to oworld
     * @param world: a pointer to the storage for the current step
     * @param oworld: a pointer to the storage for the previous step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
#pragma acc parallel loop collapse(2) present(world[:(rows+2)*(cols+2)],␣
 ↪oworld[:(rows+2)*(cols+2)])
    for (int r=1; r<=rows; ++r)
        for (int c=1; c <= cols; ++c)
            oworld[r*(cols+2) + c] = world[r*(cols+2) + c];
}

int alive(int* restrict world, int rows, int cols)
{
    /**
     * Compute the number of cells alive at the current generation
     * @param world: a pointer to the storage for the current step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    int cells = 0;
#pragma acc parallel loop collapse(2) reduction(+:cells)␣
 ↪present(world[:(rows+2)*(cols+2)])
    for (int r=1; r <= rows; ++r)
        for (int c=1; c <= cols; ++c)
            cells += world[r*(cols+2) + c];
    return cells;
}

void fill_world(int* restrict world, int rows, int cols)
{
    /**
     *  Set the initial state of the world
     * @param world: a pointer to the storage for the current step
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
    for (int r=1; r <= rows; ++r)
        for (int c=1; c <= cols; ++c)
            world[r*(cols+2) + c] = rand()%4==0 ?1 : 0;
    // The border of the world is a dead zone
```

(continues on next page)

(continued from previous page)

```
    for (int i=0;i<=rows;++i)
    {
        world[i*(cols+2)] = 0;
        world[i*(cols+2)+cols+1] = 0;
    }
    for (int j=0; j<cols; ++j)
    {
        world[j] = 0;
        world[(rows+1)*(cols+2)+j] = 0;
    }
}

int* allocate(int rows, int cols)
{
    /**
     * Allocate memory for a 2D array
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     * @return a pointer to the matrix
     */

    int* mat = (int*) malloc((rows+2)*(cols+2)*sizeof(int));
#pragma acc enter data create(mat[0:(rows+2)*(cols+2)])

    return mat;
}

void destroy(int* mat, int rows, int cols)
{
    /**
     * Free memory for a 2D array
     * @param mat: a pointer to the matrix to free
     * @param rows: the number of rows without the border
     * @param cols: the number of columns without the border
     */
#pragma acc exit data delete(mat[0:(rows+2)*(cols+2)])
    free(mat);
}

int main(int argc, char** argv)
{
    int rows, cols, generations;
    int* world;
    int* oworld;

    if (argc < 4)
    {
        printf("Wrong number of arguments: Please give rows cols and generations\n");
        return 1;
    }
    rows = strtol(argv[1], NULL, 10);
    cols = strtol(argv[2], NULL, 10);
    generations = strtol(argv[3], NULL, 10);

    world = allocate(rows, cols);
    oworld = allocate(rows, cols);
```

(continues on next page)

(continued from previous page)

```
    fill_world(world, rows, cols);
    printf("Initial state set\n");
#pragma acc update device(world[0:(rows+2)*(cols+2)])
    printf("Cells alive at generation %d: %d\n", 0, alive(world, rows, cols));
    for (int g=1; g <= generations; ++g)
    {
        save(world, oworld, rows, cols);
        next(world, oworld, rows, cols);
        output_world(world, rows, cols, g);
        printf("Cells alive at generation %4d: %d\n", g, alive(world, rows, cols));
    }

    destroy(world, rows, cols);
    destroy(oworld, rows, cols);

    return 0;
}
```

```
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):

```
#pragma acc kernels
{
    // 1st kernel generated
    #pragma acc loop
    for(int i=0; i<size_i; ++i)
    {
        for(int j=0; j<size_j; ++j)
        {
            // Perform some computation
        }
    }

    // 2nd kernel generated
    #pragma acc loop
    for(int i=0; i<size_i; ++i)
    {
        // Some more computation
    }
}
```

It is almost equivalent to this example:

```
#pragma acc data <data clauses>
{
    // 1st kernel generated
    #pragma acc parallel loop
    for(int i=0; i<size_i; ++i)
    {
```

(continues on next page)

(continued from previous page)

```
        for(int j=0; j<size_j; ++j)
        {
            // Perform some computation
        }
    }

    // 2nd kernel generated
    #pragma acc parallel loop
    for(int i=0; i<size_i; ++i)
    {
        // Some more computation
    }

}
```

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`.

```
#pragma acc kernels
{
    #pragma acc loop independent
    for (int i=0; i<size; ++i)
    {
        // A very safe loop
    }
}
```

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

```
#pragma acc serial <clauses>
{
    // My sequential kernel
}
```

which is equivalent to:

```
#pragma acc parallel num_gangs(1) num_workers(1) vector_length(1)
{
    // My sequential kernel
}
```

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:

```
double scalar = 42.;
double* array = (double) malloc(size*sizeof(double));
#pragma acc parallel
{
    // scalar would have been private in any case because we use 'acc parallel'␣
 ↪compute construct
    #pragma acc loop gang private(scalar, array)
    for (int i=0; i<size; ++i)
    {
        // do some work on scalar and array
    }
}
```

A single loop with variables private at worker level:

```
double scalar = 42.;
double* array = (double) malloc(size*sizeof(double));
#pragma acc parallel
{
    #pragma acc loop gang worker private(scalar, array)
    for (int i=0; i<size; ++i)
    {
        // do some work on scalar and array
    }
}
```

A single loop with variables private at vector level:

```
double scalar = 42.;
double* array = (double) malloc(size*sizeof(double));
#pragma acc parallel
{
    // the number of workers here is 1
    #pragma acc loop gang vector private(scalar, array)
    for (int i=0; i<size; ++i)
    {
        // do some work on scalar and array
    }
}
```

7.2.2 A bit less straightforward

Nested loops:

```
double scalar1 = 0.;
double scalar2;
#pragma acc parallel
{
    #pragma acc loop gang reduction(+:scalar1) private(scalar2)
    for (int i=0; i<size_i; ++i)
    {
        scalar2 = 0.;
        // scalar2 is private at gang level but shared at worker/vector level
        #pragma acc loop vector reduction(+:scalar2)
        for (int j=0; i<size_j; ++j)
        {
            scalar2 += ... ;
        }
        scalar1 += scalar2;
    }
}
```

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*.

```
#pragma acc parallel num_gangs(3500) vector_length(256)
{
    #pragma acc loop gang
    for(int i = 0 ; i < size_i ; ++i)
    {
        #pragma acc loop vector
        for(int j = 0 ; j < size_j ; ++j)
        {
            // A Fabulous calculation
        }
    }
}

#pragma acc parallel loop gang num_gangs(size_i/2) vector_length(256)
for(int i = 0 ; i < size_i ; ++i)
{
    #pragma acc loop vector
    for(int j = 0 ; j < size_j ; ++j)
    {
        // A Fabulous calculation
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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/C/Loop_configuration_example.c
#include <stdio.h>
#include <stdlib.h>
#include <openacc.h>
int main(void)
{
    int n = 200;
    int ngangs = 1, nworkers = 2, nvectors = 32;
    size_t table[n*n*n];

#pragma acc parallel loop gang num_gangs(ngangs) num_workers(nworkers) vector_
 ↪length(nvectors) copyout(table[0:n*n*n])
    for (int i=0; i<n; ++i)
    {
#pragma acc loop worker
        for (int j=0; j<n; ++j)
        {
            #pragma acc loop vector
            for (int k=0; k<n; ++k) table[i*n*n + j*n + k] = k + 1000*j + 1000*1000*i;
        }
    }
    printf("%d %d\n",table[0],table[n*n*n-1]);
}
```

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/C/Loop_configuration_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <openacc.h>

int main(int argc, char** argv)
{
    size_t size=50000;
    double table[size];
    double sum_val;
    double res;

    unsigned int ngangs = (unsigned int) atoi(argv[1]);

    res = 0.0;
    double norm = 1./((double)size*(double)size);
    for (size_t i=0; i<size; ++i)
    {
        for(size_t j=0; j<size; ++j)
        {
            table[j] = (i+j)*norm;
        }
        sum_val = 0.0;
        for(size_t j=0; j<size; ++j)
        {
            sum_val += table[j];
        }
        res += sum_val;
    }
    printf("result: %lf \n",res);
    return 0;
}
```

8.4.1 Solution

```
%%idrrun -a --cliopts "500"
//  examples/C/Loop_configuration_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <openacc.h>

int main(int argc, char** argv)
{
    size_t size=50000;
    double array[size];
    double table[size];
    double sum_val;
    double res;

    unsigned int ngangs = (unsigned int) atoi(argv[1]);

    res = 0.0;
    double norm = 1./((double)size*(double)size);
#pragma acc parallel num_gangs(ngangs) copyout(array[0:size]) private(table[0:size])
{
    #pragma acc loop gang reduction(+:res)
    for (size_t i=0; i<size; ++i)
    {
        #pragma acc loop vector
        for(size_t j=0; j<size; ++j)
        {
           table[j] = (i+j)*norm;
        }
        sum_val = 0.0;
        #pragma acc loop vector reduction(+:sum_val)
        for(size_t j=0; j<size; ++j)
        {
            sum_val += table[j];
        }
        array[i] = sum_val;
        res += sum_val;
    }
}

    printf("result: %lf\n",res);
    return 0;
}
```

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:

```
#pragma acc routine seq
double mean_value(double* array, size_t array_size)
{
    // compute the 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.

```
#pragma acc routine worker
void my_worker_func(){...}

...
#pragma acc parallel loop worker
for (int i=0; i<size; ++i)
    my_worker_func();
```

For a similar reason this is forbidden:

```
#pragma acc routine gang
void my_gang_func(){...}

...
#pragma acc parallel
{
    #pragma acc loop gang
    for (int i=0; i<size; ++i)
        my_gang_func();
}
```

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

```
#pragma acc routine vector
void my_wrong_routine()
{
    #pragma acc loop gang worker
    for (int i=0; i<size; ++i)
        // some loop stuff
}
```

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

You can declare the `acc routine` directive anywhere a function prototype is allowed. It has to be done *before* the definition of the function or its usage in that scope.

```
#pragma acc routine(beautiful_name) seq
...

char* beautiful_name(char* name)
{
    // Do something
}
```

or:

```
#pragma acc routine(beautiful_name) seq
...
#pragma acc routine
int another_brick(char* name)
{
    char* beauty = beautiful_name(name);
    // Integers are beauty
    ...
    return int_beauty;
}
```

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.

```
#pragma acc routine vector
void init(int* array, size_t size){
    #pragma acc loop
    for (int i=0; i<size; ++i)
        array[i] = i;
}
```

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/C/Modular_programming_mean_value_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double mean_value(double* array, size_t array_size){
    double sum = 0.0;
    for(size_t i=0; i<array_size; ++i)
        sum += array[i];
    return sum/array_size;
}

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

9.4.1 Solution

```
%%idrrun -a
//  examples/C/Modular_programming_mean_value_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#pragma acc routine vector
double mean_value(double* array, size_t array_size){
    double sum = 0.0;
    #pragma acc loop vector reduction(+:sum)
    for(size_t i=0; i<array_size; ++i)
        sum += array[i];
    return sum/array_size;
}

void rand_init(double* array, size_t array_size)
{
     srand((unsigned) 12345900);
```

(continues on next page)

(continued from previous page)

```
     for (size_t i=0; i<array_size; ++i)
         array[i] = 2.*((double)rand()/RAND_MAX -0.5);
}

void iterate(double* array, size_t array_size, size_t cell_size)
{
    double local_mean;
    #pragma acc parallel loop private(local_mean) present(array[:array_size])
    for (size_t i = cell_size/2; i< array_size-cell_size/2; ++i)
    {
        local_mean = mean_value(&array[i-cell_size/2], cell_size);
        array[i] += signbit(local_mean) * 0.1;
    }
}

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

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

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

    #pragma acc parallel loop gang present(table[0:num_rows*num_cols]) copyout(mean_
↪values[0:num_rows])
    for (size_t i=0; i<num_rows; ++i)
    {
        mean_values[i] = mean_value(&(table[i*num_cols]), num_cols);
    }

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

CHAPTER

TEN

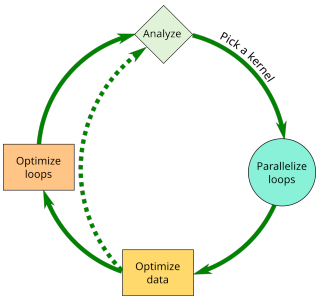
PROFILING YOUR CODE TO FIND WHAT TO OFFLOAD

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.

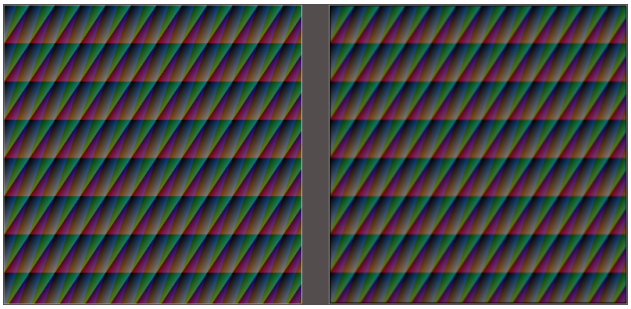


10.2 Quick description of the code

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.

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

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

```
%idrrunfile --profile ../../examples/C/blur.c
```

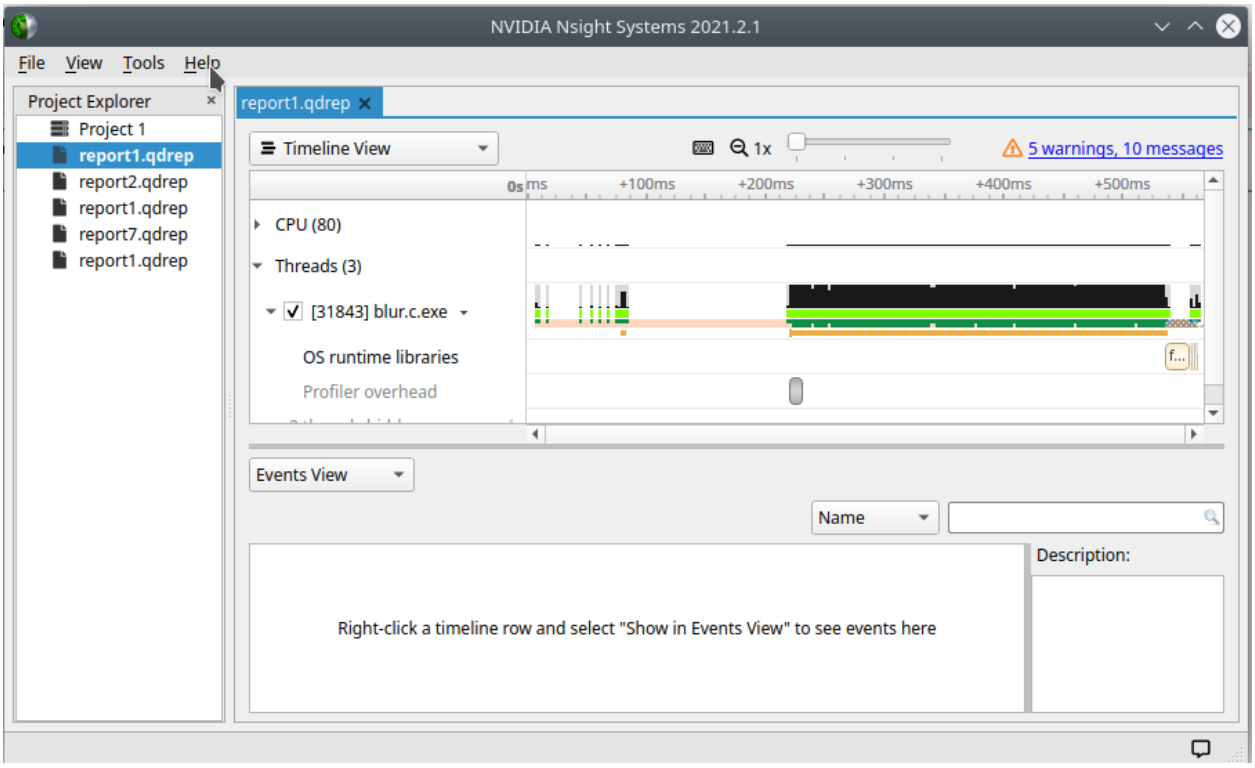
Now you can run the UI by executing the following cell and choosing the right reportxx.qdrep file (here it should be report1.qdrep).

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
```

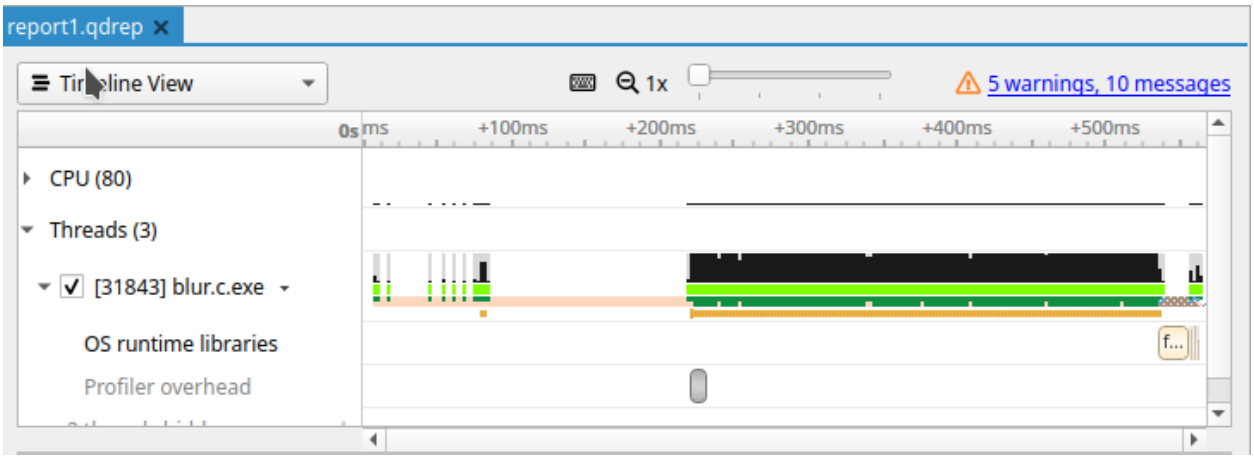
10.4 The graphical profiler

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



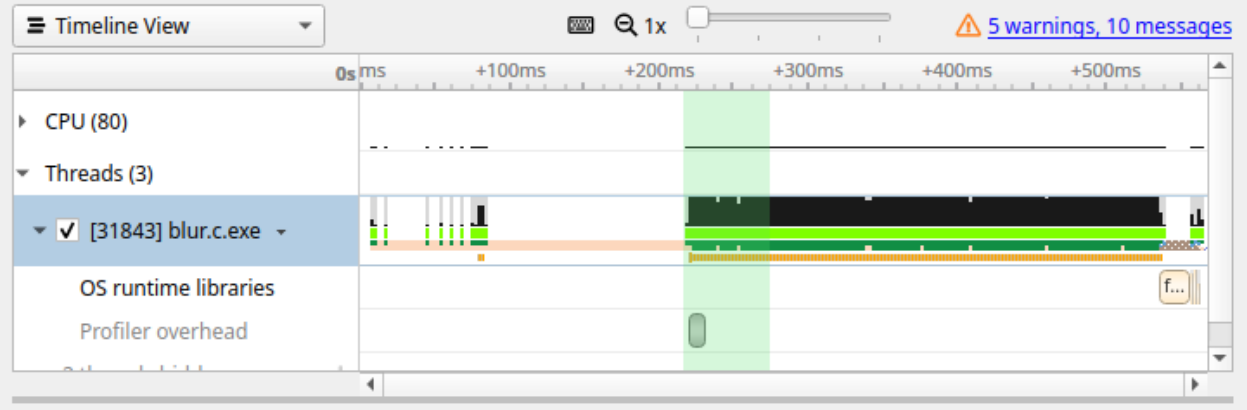
10.4.1 The timeline

Maybe the most important part is the timeline:

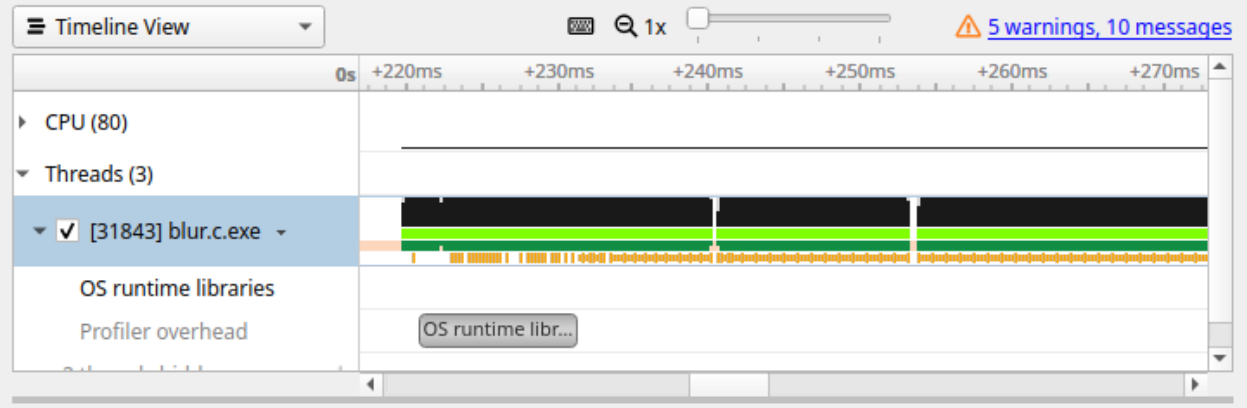


It has the information about what happened during execution of your code with a timeline view.

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.

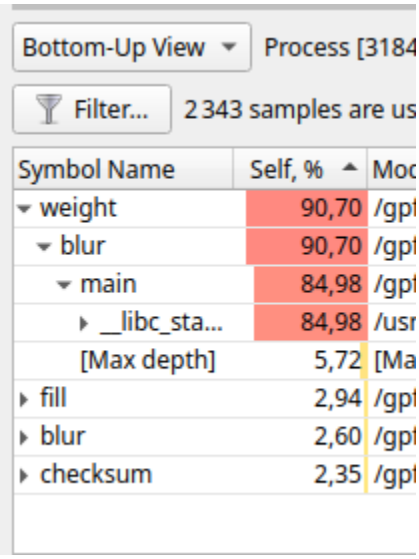


and zoom (maj+z or right-click “Zoom into selection”):



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

```
for (i=0; i<5; ++i)
   for (j=0; j<5; ++j)
      pix += pic[(x+i-2)*3*cols+y*3+l-2+j]*coefs[i][j];
```

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/C/blur.c
```

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/C/MultiGPU_mpi_example.c
#include <stdio.h>
#include <mpi.h>
#include <openacc.h>
#include "../../examples/C/init_openacc.h"
int main(int argc, char** argv)
{
    // Useful for OpenMPI and GPU DIRECT
    initialisation_openacc();
    MPI_Init(&argc, &argv);
```

(continues on next page)

(continued from previous page)

```
    // MPI Stuff
    int my_rank;
    int comm_size;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    int a[100];

    // OpenACC Stuff
    #ifdef _OPENACC
    acc_device_t device_type = acc_get_device_type();
    int num_gpus = acc_get_num_devices(device_type);
    int my_gpu = my_rank%num_gpus;
    acc_set_device_num(my_gpu, device_type);
    my_gpu = acc_get_device_num(device_type);
    // Alternatively you can set the GPU number with #pragma acc set device_num(my_
↪gpu)

    #pragma acc parallel
    {
        #pragma acc loop
        for(int i = 0; i< 100; ++i)
            a[i] = i;
    }
    #endif
    printf("Here is rank %d: I am using GPU %d of type %d. a[42] = %d\n", my_rank,␣
↪my_gpu, device_type, a[42]);
    MPI_Finalize();
    return 0;
}
```

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/C/MultiGPU_openmp_example.c
#include <stdio.h>
#include <openacc.h>
#include <omp.h>
int main(int argc,char** argv)
{
    #pragma omp parallel
    {
```

(continues on next page)

(continued from previous page)

```
        int my_rank = omp_get_thread_num();
        // OpenACC Stuff
        #ifdef _OPENACC
        acc_device_t dev_type = acc_get_device_type();
        int num_gpus = acc_get_num_devices(dev_type);
        int my_gpu = my_rank%num_gpus;
        acc_set_device_num(my_gpu, dev_type);
        // We check what GPU is really in use
        my_gpu = acc_get_device_num(dev_type);
        // Alternatively you can set the GPU number with #pragma acc set device_
↪num(my_gpu)
        printf("Here is thread %d: I am using GPU %d of type %d.\n", my_rank, my_gpu,
↪ dev_type);
        #endif
    }
}
```

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.

```
#pragma acc host_data use_device(array)
```

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.

```
int size = 1000;
int* array = (int*) malloc(size*sizeof(int));
#pragma acc enter_data create(array[0:1000])
// Perform some stuff on the GPU
#pragma acc parallel present(array[0:1000])
{
...
}
// Transfer the data between GPUs
if (my_rank == origin )
{
    #pragma acc host_data use_device(array)
    MPI_Send(array, size, MPI_INT, destination, tag, MPI_COMM_WORLD);
}
else if (my_rank == destination)
{
    #pragma acc host_data use_device(array)
    MPI_Recv(array, size, MPI_INT, origin, tag, MPI_COMM_WORLD, &status);
}
```

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/C/MultiGPU_mpi_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <openacc.h>
#include <math.h>
#include "../../examples/C/init_openacc.h"
int main(int argc, char** argv)
{
    initialisation_openacc();
```

(continues on next page)

(continued from previous page)

```
    MPI_Init(&argc, &argv);
    fflush(stdout);
    double start;
    double end;

    int size = 2e8/8;

    double* send_buffer = (double*)malloc(size*sizeof(double));
    double* receive_buffer = (double*)malloc(size*sizeof(double));
    // MPI Stuff
    int my_rank;
    int comm_size;
    int reps = 5;
    double data_volume = (double)reps*(double)size*sizeof(double)*pow(1024,-3.0);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Status status;

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

Solution

```
%%idrrun -a -m 4 -g 4
//  examples/C/MultiGPU_mpi_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <openacc.h>
#include <math.h>
#include "../../examples/C/init_openacc.h"
int main(int argc, char** argv)
{
    initialisation_openacc();
    MPI_Init(&argc, &argv);
    fflush(stdout);
    double start;
    double end;

    int size = 2e8/8;

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

    // OpenACC Stuff
    acc_device_t device_type = acc_get_device_type();
    int num_gpus = acc_get_num_devices(device_type);
    int my_gpu = my_rank%num_gpus;
    acc_set_device_num(my_gpu, device_type);
    for (int i = 0; i<comm_size; ++i)
    {
        for (int j=0; j < comm_size; ++j)
        {
            if (my_rank == i && i != j)
            {
                start = MPI_Wtime();
                #pragma acc host_data use_device(send_buffer)
                for (int k = 0 ; k < reps; ++k)
                    MPI_Ssend(send_buffer, size, MPI_DOUBLE, j, 0, MPI_COMM_WORLD);
            }
            if (my_rank == j && i != j)
            {
                #pragma acc host_data use_device(receive_buffer)
                for (int k = 0 ; k < reps; ++k)
                    MPI_Recv(receive_buffer, size, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &
↪status);
            }
            if (my_rank == i && i != j)
            {
```

(continues on next page)

(continued from previous page)

```
                end = MPI_Wtime();
                printf("bandwidth %d->%d: %10.5f GB/s\n", i, j, data_volume/(end-
↪start));
            }
        }
    }
    MPI_Finalize();
    return 0;
}
```

Requirements:

- Get started
- Data management
- Multi GPU

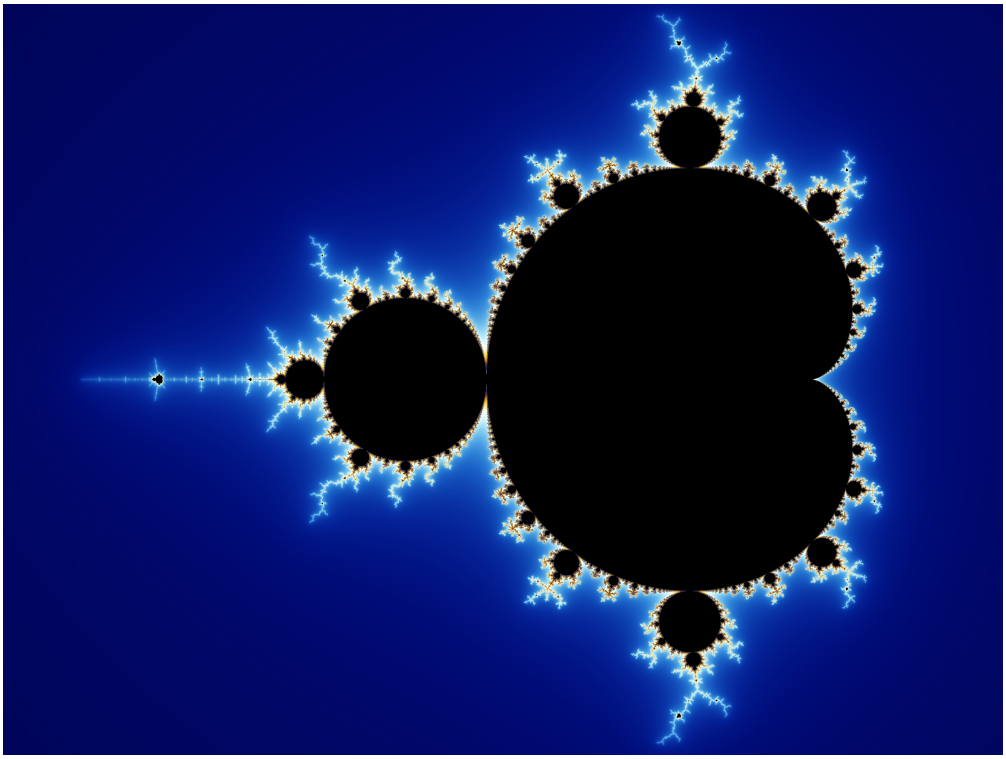
CHAPTER

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/C/mandelbrot_mpi_exercise.c
// you can use ` --option "-DMULTIGPU" `  to print the device info after filling the␣
 ↪openacc initialisation
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef _OPENACC
  #include <openacc.h>
#endif
#include <complex.h>
#include <mpi.h>
// add openacc initialisation
void output(unsigned char* picture, unsigned int start, unsigned int num_elements)
{
   MPI_File     fh;
   MPI_Offset   woffset=start;

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

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

   MPI_File_close(&fh);
}

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

(continues on next page)

(continued from previous page)

```
    #ifdef _OPENACC
    // add initilisation openacc
    #endif
    MPI_Init(&argc, &argv);
    unsigned int width = (unsigned int) atoi(argv[1]);
    float step_w = 1./width;
    unsigned int height = (unsigned int) atoi(argv[2]);
    float step_h = 1./height;

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

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

    int i;
    int rank;
    int nb_procs;

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

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

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

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

    if (rank < rest_eucli)
        ++local_height;

    unsigned int num_elements = width*local_height;
    if (rank == 0) printf("Using MPI\n");
    #if defined(_OPENACC) && defined(MULTIGPU)
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements. I use GPU %d␣
↪over %d devices.\n", rank, first, last, num_elements,info.current_device, info.
↪total_devices);
    #else
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements.", rank, first,␣
↪last, num_elements);
    #endif
```

(continues on next page)

(continued from previous page)

```
    unsigned char* restrict picture = (unsigned char*) malloc(num_
 ↪elements*sizeof(unsigned char));

    for (unsigned int i=0; i<local_height; ++i)
        for (unsigned int j=0; j<width; ++j)
        {
            float complex c;
            c = min_re + j*step_w * (max_re - min_re) + \
                I * (min_im +  ((i+first) * step_h) * (max_im - min_im));
            picture[width*i+j] = (unsigned char)255 - mandelbrot_iterations(c);
        }
    output(picture, first*width, num_elements);
    MPI_Finalize();

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

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

12.3 Solution

```
%%idrrun -a -m 4 -g 4
//  examples/C/mandelbrot_mpi_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef _OPENACC
  #include <openacc.h>
#endif
#include <complex.h>
#include <mpi.h>
#include "../../examples/C/init_openacc.h"
void output(unsigned char* picture, unsigned int start, unsigned int num_elements)
{
   MPI_File     fh;
   MPI_Offset   woffset=start;

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

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

(continues on next page)

(continued from previous page)

```
   MPI_File_close(&fh);
}

#pragma acc routine seq
unsigned char mandelbrot_iterations(const float complex c)
{
    unsigned char max_iter = 255;
    unsigned char n = 0;
    float complex z = 0.0 + 0.0 * I;
    while (abs(z*z) <= 2 && n < max_iter)
    {
        z = z*z + c;
        ++n;
    }
    return n;
}
int main(int argc, char** argv)
{
    #ifdef _OPENACC
    acc_info info = initialisation_openacc();
    #endif
    MPI_Init(&argc, &argv);
    unsigned int width = (unsigned int) atoi(argv[1]);
    float step_w = 1./width;
    unsigned int height = (unsigned int) atoi(argv[2]);
    float step_h = 1./height;

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

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

    int i;
    int rank;
    int nb_procs;

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

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

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

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

(continues on next page)

(continued from previous page)

```
            {
                ++first;
                ++last;
            }
    }

    if (rank < rest_eucli)
        ++local_height;

    unsigned int num_elements = width*local_height;
    if (rank == 0) printf("Using MPI\n");
    #ifdef _OPENACC
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements. I use GPU %d␣
↪over %d devices.\n", rank, first, last, num_elements,info.current_device, info.
↪total_devices);
    #else
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements.", rank, first,
↪ last, num_elements);
    #endif

    unsigned char* restrict picture = (unsigned char*) malloc(num_
↪elements*sizeof(unsigned char));

#pragma acc data copyout(picture[0:num_elements])
{
#pragma acc parallel loop
    for (unsigned int i=0; i<local_height; ++i)
        for (unsigned int j=0; j<width; ++j)
        {
            float complex c;
            c = min_re + j*step_w * (max_re - min_re) + \
                I * (min_im +  ((i+first) * step_h) * (max_im - min_im));
            picture[width*i+j] = (unsigned char)255 - mandelbrot_iterations(c);
        }
}
    output(picture, first*width, num_elements);
    MPI_Finalize();

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

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

Requirements:

- Get started
- Data management
- Multi GPU

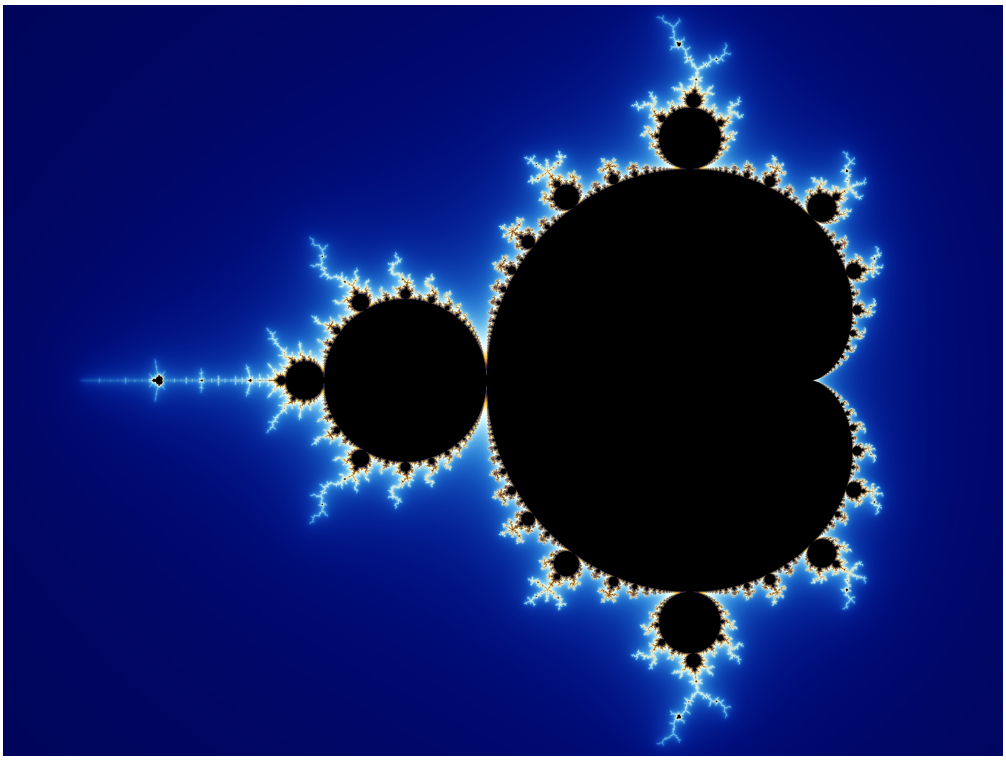
CHAPTER

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/C/mandelbrot_openmp_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>
#include <time.h>
#ifdef _OPENMP
   #include <omp.h>
#endif
#ifdef _OPENACC
   #include <openacc.h>
#endif
void output(unsigned char* picture, unsigned int width, unsigned int height)
{
   FILE* f = fopen("mandel.gray", "wb");
   fwrite(picture, sizeof(unsigned char), width*height, f);
   fclose(f);
}

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

int main(int argc, char** argv)
{
    if (argc < 3)
    {
        printf("Please give width and height of the world.");
        return 1;
    }
    unsigned int width = (unsigned int) atoi(argv[1]);
    float step_w = 1./width;
    unsigned int height = (unsigned int) atoi(argv[2]);
    float step_h = 1./height;
    unsigned char* restrict picture = (unsigned char*)␣
 ↪malloc(width*height*sizeof(unsigned char));
    // Here we set the bonds of the coordinates of the picture.
    const float min_re = -2;
    const float max_re = 1;
```

(continues on next page)

(continued from previous page)

```
    const float min_im = -1;
    const float max_im = 1;

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

    int rank = 0;
    unsigned int first = 0;
    unsigned int last = height;
    int num_elements = width*height;
#pragma omp parallel private(first, last, rank) shared(picture) firstprivate(height,␣
 ↪width, min_re, max_re, min_im, max_im, step_h, step_w, num_elements) default(none)
{
#ifdef _OPENMP
    rank = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    first = rank * (height/num_threads);
    last  = (rank + 1) * (height/num_threads);
    num_elements = width*height/num_threads;
#pragma omp master
{
    printf("Using OpenMP\n");
}
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements\n", rank,␣
 ↪first, last, num_elements);
#endif

#ifdef _OPENACC
    acc_device_t type = acc_get_device_type();
    int num_gpu = acc_get_num_devices(type);
    acc_set_device_num(rank%num_gpu, type);
    printf("I am rank %2d. I am using GPU %d\n", rank, acc_get_device_num(type));
#endif

    for (unsigned int i=first; i<last; ++i)
        for (unsigned int j=0; j<width; ++j)
        {
            float complex c;
            c = min_re + j * step_w * (max_re - min_re) + \
                I * (min_im + ( i * step_h) * (max_im - min_im));
            picture[width*i+j] = (unsigned char) 255 - mandelbrot_iterations(c);
        }
}
    // Measure time
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
    unsigned long int delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec␣
 ↪- start.tv_nsec) / 1000;
    printf("The time to generate the mandelbrot picture was %10.5e s\n", delta_us/1.
 ↪e6);
    output(picture, width, height);
}
```

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

13.3 Solution

```
%%idrrun --cliopts "8000 4000" -t -a -g 4
//  examples/C/mandelbrot_openmp_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>
#include <time.h>
#ifdef _OPENMP
   #include <omp.h>
#endif
#ifdef _OPENACC
   #include <openacc.h>
#endif
void output(unsigned char* picture, unsigned int width, unsigned int height)
{
   FILE* f = fopen("mandel.gray", "wb");
   fwrite(picture, sizeof(unsigned char), width*height, f);
   fclose(f);
}

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

int main(int argc, char** argv)
{
    if (argc < 3)
    {
        printf("Please give width and height of the world.");
        return 1;
    }
    unsigned int width = (unsigned int) atoi(argv[1]);
    float step_w = 1./width;
    unsigned int height = (unsigned int) atoi(argv[2]);
    float step_h = 1./height;
    unsigned char* restrict picture = (unsigned char*)␣
 ↪malloc(width*height*sizeof(unsigned char));
    // Here we set the bonds of the coordinates of the picture.
    const float min_re = -2;
    const float max_re = 1;
    const float min_im = -1;
    const float max_im = 1;

    struct timespec end, start;
```

(continues on next page)

(continued from previous page)

```
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    int rank = 0;
    unsigned int first = 0;
    unsigned int last = height;
    int num_elements = width*height;
#pragma omp parallel private(first, last, rank) shared(picture) firstprivate(height,␣
 ↪width, min_re, max_re, min_im, max_im, step_h, step_w, num_elements) default(none)
{
#ifdef _OPENMP
    rank = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    first = rank * (height/num_threads);
    last  = (rank + 1) * (height/num_threads);
    num_elements = width*height/num_threads;
#pragma omp master
{
    printf("Using OpenMP\n");
}
    printf("I am rank %2d and my range is [%5d, %5d[ ie %10d elements\n", rank,␣
 ↪first, last, num_elements);
#endif

#ifdef _OPENACC
    acc_device_t type = acc_get_device_type();
    int num_gpu = acc_get_num_devices(type);
    acc_set_device_num(rank%num_gpu, type);
    printf("I am rank %2d. I am using GPU %d\n", rank, acc_get_device_num(type));
#endif

#pragma acc parallel copyout(picture[first*width:num_elements])
    {
    #pragma acc loop
    for (unsigned int i=first; i<last; ++i)
        for (unsigned int j=0; j<width; ++j)
        {
            float complex c;
            c = min_re + j * step_w * (max_re - min_re) + \
                I * (min_im + ( i * step_h) * (max_im - min_im));
            picture[width*i+j] = (unsigned char) 255 - mandelbrot_iterations(c);
        }
    }
}
    // Measure time
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
    unsigned long int delta_us = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec␣
 ↪- start.tv_nsec) / 1000;
    printf("The time to generate the mandelbrot picture was %10.5e s\n", delta_us/1.
 ↪e6);
    output(picture, width, height);
}
```

```
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.

```
int stream1=1;
int stream2=2;
#pragma acc enter data copyin(array[:size]) async(stream1)
// Some stuff
#pragma acc parallel async(stream2)
{
    // A wonderful kernel
}
```

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.

```
int stream1=1;
int stream2=2;
#pragma acc enter data copyin(array[:size]) async(stream1)
// Some stuff
#pragma acc parallel async(stream2) wait(stream1)
{
    // A wonderful kernel
}
```

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.

```
int stream1=1;
int stream2=2;
int stream3=3;
#pragma parallel loop async(stream3)
for (int i=0; i <size; ++i)
{
    // Kernel launched on stream3
}
#pragma acc enter data copyin(array[:size]) async(stream1)
// Some stuff
#pragma acc parallel async(stream2) wait(stream1, stream3)
{
    // A wonderful kernel
}
```

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

```
#pragma acc parallel wait
{
    // A wonderful kernel
}
```

14.3 *wait* directive

wait comes also as a standalone directive.

```
int stream1=1;
int stream2=2;
int stream3=3;
#pragma parallel loop async(stream3)
for (int i=0; i <size; ++i)
{
    // Kernel launched on stream3
}
#pragma acc enter data copyin(array[:size]) async(stream1)
// Some stuff

#pragma acc wait(stream3)

#pragma acc parallel async(stream2)
{
    // A wonderful kernel
}
```

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/C/async_async_exercise.c
#include <stdio.h>
#include <stdlib.h>
double* create_mat(int dim, int stream)
{
    double* mat = (double*) malloc(dim*dim*sizeof(double));
    return mat;
}
```

(continues on next page)

(continued from previous page)

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

int main(void)
{
    int dim = 5000;

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

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

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

14.4.1 Solution

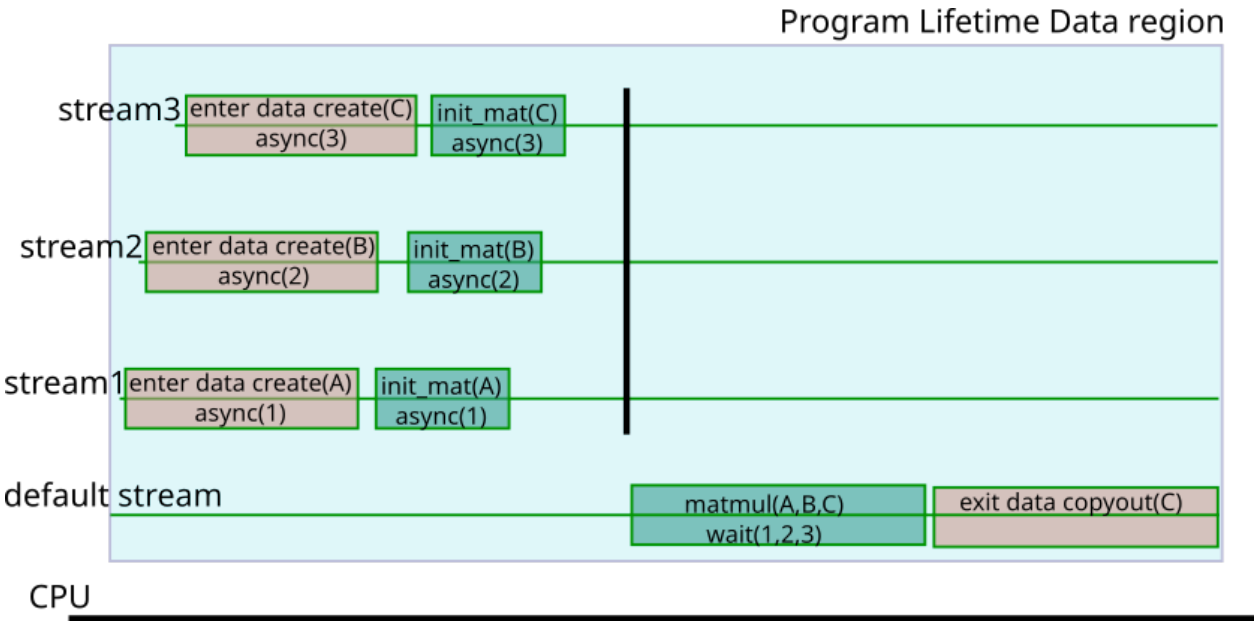
```
%%idrrun -a
//  examples/C/async_async_solution.c
#include <stdio.h>
#include <stdlib.h>
double* create_mat(int dim, int stream)
{
    double* mat = (double*) malloc(dim*dim*sizeof(double));
    #pragma acc enter data create(mat[0:dim*dim]) async(stream)
    return mat;
}

void init_mat(double* mat, int dim, double diag, int stream)
{
    #pragma acc parallel loop present(mat[0:dim*dim]) async(stream)
    for (int i=0; i<dim; ++i)
        #pragma acc loop
        for (int j=0; j<dim; ++j)
        {
            mat[i*dim+j] = 0.;
```

(continues on next page)

(continued from previous page)

```
        }
    #pragma acc parallel loop present(mat[0:dim*dim]) async(stream)
    for (int i=0; i<dim; ++i)
        mat[i*dim+i] = diag;
}

int main(void)
{
    int dim = 5000;

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

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

    #pragma acc parallel present(A[:dim*dim], B[:dim*dim], C[:dim*dim]) wait(1,2,3)
    {
    #pragma acc loop gang vector collapse(3)
    for (int i=0; i<dim; ++i)
        for (int k=0; k<dim; ++k)
            for (int j=0; j<dim; ++j)
            {
                #pragma acc atomic update
                C[i*dim+j] += A[i*dim+k] * B[k*dim+j];
            }
    }
    #pragma acc exit data delete(A[:dim*dim], B[:dim*dim]) copyout(C[:dim*dim])
    printf("Check that value is equal to 42.: %f\n", C[0]);
    return 0;
}
```

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



14.4.2 Comments

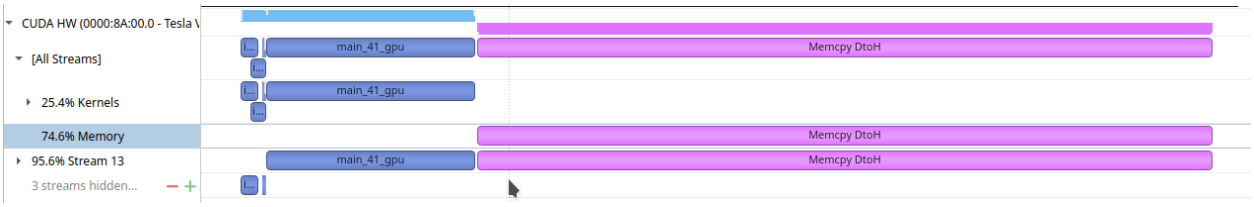
- Several threads will update the same memory location for C so you have to use an `acc atomic update`
- `collapse` is used to fuse the 3 loops. It helps the compiler to generate a more efficient code

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*

```
#pragma acc atomic <clause>
// One atomic operation
```

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:

```
#pragma acc atomic capture
{
//Several atomic operations
}
```

In C it can also work on the capture operation just below.

```
#pragma acc atomic capture
// One capture operation
```

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:

```
#pragma acc atomic read
v = x;
```

15.2.2 *write*

The expression must have the form:

```
#pragma acc atomic write
x = expr;
```

15.2.3 *update*

Several forms are available:

```
//x = x _binop_ expr;
#pragma acc update
x = x + (3*10);

//x_binop_;
#pragma acc update
x++;

//_binop_x
#pragma acc update
--x;

//x _binop_= expr
#pragma acc update
x += 30;
```

15.2.4 *capture*

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

```
//v = x = x _binop_ expr;
#pragma acc capture
v = x = x + (3*10);

//v = x_binop_;
#pragma acc capture
v = x++;

//v = _binop_x
#pragma acc capture
v = --x;

//v = x _binop_= expr
#pragma acc capture
v = x += 30;
```

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/C/atomic_exercise.c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    // Histogram allocation and initialization
    int histo[10];
    for (int i=0; i<10; ++i)
        histo[i] = 0;
    size_t nshots = (size_t) 1e9;

    // Allocate memory for the random numbers
    int* shots = (int*) malloc(nshots*sizeof(int));

    srand((unsigned) 12345900);

    // Fill the array on the CPU (rand is not available on GPU with Nvidia Compilers)
    for (size_t i=0; i< nshots; ++i)
    {
        shots[i] = (int) rand() % 10;
```

(continues on next page)

(continued from previous page)

```
    }

    // Count the number of time each number was drawn
    for (size_t i=0; i<nshots; ++i)
    {
        histo[shots[i]]++;
    }

    // Print results

    for (int i=0; i<10; ++i)
        printf("%3d: %10d (%5.3f)\n", i, histo[i], (double) histo[i]/1.e9);

    return 0;
}
```

15.3.1 Solution

```
%%idrrun -a
//  examples/C/atomic_solution.c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    // Histogram allocation and initialization
    int histo[10];
    for (int i=0; i<10; ++i)
        histo[i] = 0;
    size_t nshots = (size_t) 1e9;

    // Allocate memory for the random numbers
    int* shots = (int*) malloc(nshots*sizeof(int));

    srand((unsigned) 12345900);

    // Fill the array on the CPU (rand is not available on GPU with Nvidia Compilers)
    for (size_t i=0; i< nshots; ++i)
    {
        shots[i] = (int) rand()%10;
    }

    // Count the number of time each number was drawn
    #pragma acc parallel loop  copyin(shots[:nshots]) copyout(histo[0:10])
    for (size_t i=0; i<nshots; ++i)
    {
        #pragma acc atomic update
        histo[shots[i]]++;
    }
    // Print results
    for (int i=0; i<10; ++i)
        printf("%3d: %10d (%5.3f)\n", i, histo[i], (double) histo[i]/1.e9);

    return 0;
}
```

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

```
typedef class{
    float *vx, *vy, *vz;
}velocity;

...
velocity U;
U.vx = (float*) malloc(sizeX*sizeof(float));;
U.vy = (float*) malloc(sizeY*sizeof(float));
U.vz = (float*) malloc(sizeZ*sizeof(float));
...

#pragma acc enter data copy(U)
#pragma acc enter data copy(U.vx[0:sizeX], U.vy[0:sizeY], U.vz[0:sizeZ])
```

(continues on next page)

(continued from previous page)

```
// 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/C/Deep_copy_example.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct
{
    double* s;
    double* c;
} Array;

Array* allocate_array(size_t size);

Array* allocate_array(size_t size)
{
    Array* arr = (Array*) malloc(sizeof(Array));
    arr->s = (double*) malloc(size*sizeof(double));
    arr->c = (double*) malloc(size*sizeof(double));
    return arr;
}

int main(void)
{
    int size=100000;
    double sum[size];

    Array* vec;
    vec = allocate_array(size);

    #pragma acc data create(vec, vec->s[:size], vec->c[:size]) copyout(sum)
    {
    #pragma acc parallel
    {
        #pragma acc loop
        for (int i=0; i<size;++i)
        {
            vec->s[i] = sin(i*M_PI/size)*sin(i*M_PI/size);
            vec->c[i] = cos(i*M_PI/size)*cos(i*M_PI/size);
        }
    }
    #pragma acc parallel
    {
        #pragma acc loop
        for (int i=1; i<size ; ++i)
            sum[i] = vec->s[i] + vec->c[size - i];
    }
```

(continues on next page)

(continued from previous page)

```
    }// end of structured data region
    printf("sum[42] = %f\n", sum[42]);
}
```

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/dyn.xyz .
```

You need to pass `--cliopts "0.5 15.5"` to idrrun.

```
%%idrrun -a --cliopts "0.5 15.5"
//  examples/C/Deep_copy_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
# include <math.h>
#include <float.h>

typedef struct
{
    size_t size;
    double* data;
} Array;

typedef struct
{
    Array* x;
    Array* y;
    Array* z;
} Coordinates;

typedef struct
{
    size_t natoms;    // number of atoms
    double lx;        // box length in each dimension
    double ly;
    double lz;
} Config;

Array* allocate_array(size_t size);
void free_array(Array* arr);
Coordinates* read_coords(char* filepath, Config *config);
void free_coords(Coordinates* coords);

Array* allocate_array(size_t size)
{
    Array* arr = (Array*) malloc(sizeof(Array));
```

(continues on next page)

(continued from previous page)

```
    arr->size = size;
    arr->data = (double*) malloc(size*sizeof(double));
    // OpenACC here
    return arr;
}

Coordinates* allocate_coords(size_t size)
{
    Coordinates* coords = (Coordinates*) malloc(sizeof(Coordinates));
    // OpenACC here
    coords->x  = allocate_array(size);
    coords->y  = allocate_array(size);
    coords->z  = allocate_array(size);
    return coords;
 }

void free_array(Array* arr)
{
    // OpenACC here
    free(arr->data);
    free(arr);
}

void free_coords(Coordinates* coords)
{
    // OpenACC here
    free_array(coords->x);
    free_array(coords->y);
    free_array(coords->z);
    free(coords);
}

Coordinates* read_coords(char* filepath, Config* conf)
{
    FILE* fptr = fopen(filepath, "r");
    char* line = NULL;
    size_t len = 0;
    size_t i = 0;
    char n[20], x[20], y[20], z[20], vx[20], vy[20], vz[20];

    if (fptr == NULL)
        exit(EXIT_FAILURE);

    getline(&line, &len, fptr);
    sscanf(line, "%s", n);
    conf->natoms = atoi(n);
    getline(&line, &len, fptr);
    sscanf(line, "%s", n);
    conf->lx = atof(n);
    conf->ly = atof(n);
    conf->lz = atof(n);
    // OpenACC here
    printf("Number of atoms in %s file: %d\n", filepath, conf->natoms);

    Coordinates* coords = allocate_coords(conf->natoms);
```

(continues on next page)

(continued from previous page)

```
    while (i < conf->natoms)
    {
        getline(&line, &len, fptr);
        sscanf(line, "%s %s %s %s %s %s %s", n, x, y, z, vx, vy, vz);
        coords->x->data[i] = atof(x);
        coords->y->data[i] = atof(y);
        coords->z->data[i] = atof(z);
        ++i;
    }
    // OpenACC here
    fclose(fptr);
    return coords;
}

int main(int argc, char** argv)
{
    double deltaR, rCutOff;
    FILE* fPtr;
    char* input;
    double xij, yij, zij, rij;
    int d;

    if (argc < 3 || argc > 4)
    {
        fprintf(stderr, "%s", "ERROR: Wrong number of parameters.\n");
        fprintf(stderr, "%s", "ERROR: The program requires at least two parameters:\n
↪");
        fprintf(stderr, "%s", "ERROR: deltaR, the length of each bin, and \n");
        fprintf(stderr, "%s", "ERROR: rCutoff, the total length (rcut < box_length/2).
↪\n");
        fprintf(stderr, "%s", "ERROR: Usage example: ./rdf 0.5 15.5 [input]\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        deltaR = atof(argv[1]);
        rCutOff = atof(argv[2]);
        if (argc == 4)
            input = argv[3];
        else
            input = "./dyn.xyz";
    }

    int maxbin = rCutOff/deltaR + 1;
    int* hist = (int*) malloc(maxbin*sizeof(int));
    double* gr = (double*) malloc(maxbin*sizeof(double));
    // OpenACC here

    for (size_t i=0; i<maxbin; ++i)
        hist[i] = 0;

    Config* conf = (Config*) malloc(sizeof(Config));
    Coordinates* coords = read_coords(input, conf);

    // OpenACC here
    for (int j = 0; j < conf->natoms; ++j)
```

(continues on next page)

(continued from previous page)

```
    // OpenACC here
        for (int i = 0; i < conf->natoms; ++i)
            if (i != j)
            {
                xij = coords->x->data[j]-coords->x->data[i];
                yij = coords->y->data[j]-coords->y->data[i];
                zij = coords->z->data[j]-coords->z->data[i];
                xij -= floor(xij/conf->lx + 0.5) *conf->lx;
                yij -= floor(yij/conf->ly + 0.5) *conf->ly;
                zij -= floor(zij/conf->lz + 0.5) *conf->lz;
                rij = xij*xij + yij*yij + zij*zij;
                d = (int) (sqrt(rij)/deltaR);
                if (d < maxbin)
    // OpenACC here
                    ++hist[d];
        }

    double rho = ((double) conf->natoms) / ((double)(conf->lx * conf->ly * conf->lz))␣
↪* 4.0 / 3.0 * acos(-1.0);
    // OpenACC here
    for (int i = 0; i < maxbin; ++i)
    {
        double nideal  = rho * ( pow((i+1)*deltaR,3) - pow(i*deltaR,3) );
        gr[i] = ((double) hist[i]) / (nideal*conf->natoms);
    }
    // OpenACC here

    fPtr = fopen("RDF","w");
    for (int i = 0; i < maxbin; ++i)
      fprintf(fPtr,"%lf %lf\n", i*deltaR, gr[i]);
    fclose(fPtr);
    // OpenACC here
    free(hist);
    free(gr);
    // OpenACC here
    free(conf);
    free_coords(coords);
    return 0;
}
```

```
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/C/Deep_copy_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>

typedef struct
{
    size_t size;
    double* data;
} Array;

typedef struct
{
    Array* x;
    Array* y;
    Array* z;
} Coordinates;

typedef struct
{
    size_t natoms;    // number of atoms
    double lx;        // box length in each dimension
    double ly;
    double lz;
} Config;

Array* allocate_array(size_t size);
void free_array(Array* arr);
Coordinates* read_coords(char* filepath, Config *config);
void free_coords(Coordinates* coords);

Array* allocate_array(size_t size)
{
    Array* arr = (Array*) malloc(sizeof(Array));
    arr->size = size;
    arr->data = (double*) malloc(size*sizeof(double));
    #pragma acc enter data create(arr, arr->data[:size]) copyin(arr->size)
    return arr;
}

Coordinates* allocate_coords(size_t size)
{
    Coordinates* coords = (Coordinates*) malloc(sizeof(Coordinates));
    #pragma acc enter data create(coords)
    coords->x  = allocate_array(size);
    coords->y  = allocate_array(size);
    coords->z  = allocate_array(size);
    return coords;
 }

void free_array(Array* arr)
{
```

(continues on next page)

(continued from previous page)

```
    #pragma acc exit data delete(arr->data,arr)
    free(arr->data);
    free(arr);
}

void free_coords(Coordinates* coords)
{
    #pragma acc exit data delete(coords)
    free_array(coords->x);
    free_array(coords->y);
    free_array(coords->z);
    free(coords);
}

Coordinates* read_coords(char* filepath, Config* conf)
{
    FILE* fptr = fopen(filepath, "r");
    char* line = NULL;
    size_t len = 0;
    size_t i = 0;
    char n[20], x[20], y[20], z[20], vx[20], vy[20], vz[20];

    if (fptr == NULL)
        exit(EXIT_FAILURE);

    getline(&line, &len, fptr);
    sscanf(line, "%s", n);
    conf->natoms = atoi(n);
    getline(&line, &len, fptr);
    sscanf(line, "%s", n);
    conf->lx = atof(n);
    conf->ly = atof(n);
    conf->lz = atof(n);
    #pragma acc enter data copyin(conf)
    printf("Number of atoms in %s file: %d\n", filepath, conf->natoms);

    Coordinates* coords = allocate_coords(conf->natoms);

    while (i < conf->natoms)
    {
        getline(&line, &len, fptr);
        sscanf(line, "%s %s %s %s %s %s %s", n, x, y, z, vx, vy, vz);
        coords->x->data[i] = atof(x);
        coords->y->data[i] = atof(y);
        coords->z->data[i] = atof(z);
        ++i;
    }
    #pragma acc update device(coords->x->data[:conf->natoms],coords->y->data[:conf->
 ↪natoms],coords->z->data[:conf->natoms])
    fclose(fptr);
    return coords;
}

int main(int argc, char** argv)
{
    double deltaR, rCutOff;
```

(continues on next page)

(continued from previous page)

```
    FILE* fPtr;
    char* input;
    double xij, yij, zij, rij;
    int d;

    if (argc < 3 || argc > 4)
    {
        fprintf(stderr, "%s", "ERROR: Wrong number of parameters.\n");
        fprintf(stderr, "%s", "ERROR: The program requires at least two parameters:\n
↪");
        fprintf(stderr, "%s", "ERROR: deltaR, the length of each bin, and \n");
        fprintf(stderr, "%s", "ERROR: rCutoff, the total length (rcut < box_length/2).
↪\n");
        fprintf(stderr, "%s", "ERROR: Usage example: ./rdf 0.5 15.5 [input]\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        deltaR = atof(argv[1]);
        rCutOff = atof(argv[2]);
        if (argc == 4)
            input = argv[3];
        else
            input = "./dyn.xyz";
    }

    int maxbin = rCutOff/deltaR + 1;
    int* hist = (int*) malloc(maxbin*sizeof(int));
    double* gr = (double*) malloc(maxbin*sizeof(double));
    #pragma acc enter data create(hist[:maxbin],gr[:maxbin])

    #pragma acc parallel loop present(hist[:maxbin])
    for (size_t i=0; i<maxbin; ++i)
        hist[i] = 0;

    Config* conf = (Config*) malloc(sizeof(Config));
    Coordinates* coords = read_coords(input, conf);

    #pragma acc parallel loop present(conf,coords,coords->x,coords->y,coords->z)\
                              present(coords->x->data[:conf->natoms],coords->y->
↪data[:conf->natoms],coords->z->data[:conf->natoms])
    for (int j = 0; j < conf->natoms; ++j)
        #pragma acc loop private(xij,yij,zij,rij,d)
        for (int i = 0; i < conf->natoms; ++i)
            if (i != j)
            {
                xij = coords->x->data[j]-coords->x->data[i];
                yij = coords->y->data[j]-coords->y->data[i];
                zij = coords->z->data[j]-coords->z->data[i];
                xij -= floor(xij/conf->lx + 0.5) *conf->lx;
                yij -= floor(yij/conf->ly + 0.5) *conf->ly;
                zij -= floor(zij/conf->lz + 0.5) *conf->lz;
                rij = xij*xij + yij*yij + zij*zij;
                d = (int) (sqrt(rij)/deltaR);
                if (d < maxbin)
                    #pragma acc atomic update
```

(continues on next page)

(continued from previous page)

```
                    ++hist[d];
        }

    double rho = ((double) conf->natoms) / ((double)(conf->lx * conf->ly * conf->lz))␣
↪* 4.0 / 3.0 * acos(-1.0);
    #pragma acc parallel loop present (hist[:maxbin],gr[:maxbin])
    for (int i = 0; i < maxbin; ++i)
    {
        double nideal  = rho * ( pow((i+1)*deltaR,3) - pow(i*deltaR,3) );
        gr[i] = ((double) hist[i]) / (nideal*conf->natoms);
    }
    #pragma acc update self(gr[:maxbin])

    fPtr = fopen("RDF","w");
    for (int i = 0; i < maxbin; ++i)
      fprintf(fPtr,"%lf %lf\n", i*deltaR, gr[i]);
    fclose(fPtr);
    #pragma acc exit data delete(hist,gr)
    free(hist);
    free(gr);
    #pragma acc exit data delete(conf)
    free(conf);
    free_coords(coords);
    return 0;
}
```

```
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.

```
typedef class{
    float *vx, *vy, *vz;
}velocity;

...
velocity U;
...

#pragma acc enter data copy(U.vx[0:size], U.vy[0:size], U.vz[0:size])
#pragma acc enter data copy(U)
```

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.

```
typedef class{
    float *vx, *vy, *vz;
}velocity;

...
velocity U;
...

#pragma acc enter data copy(U.vx[0:size], U.vy[0:size], U.vz[0:size])
#pragma acc enter data copy(U) attach(U.vx, U.vy, U.vz)
```

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.

```
#pragma acc exit data detach (U.vx, U.vy, U.vz)    // not required
#pragma acc exit data copyout(U.vx, U.vy, U.vz)
#pragma acc exit data copyout(U)
```

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/C/Deep_copy_attach_detach_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>

typedef struct
{
    double* count;
    double* birth_rate;
    double* death_rate;
} Population;

void free_pop(Population* pop)
{
    // Add openacc directives
    free(pop->count);
    free(pop->birth_rate);
    free(pop->death_rate);
    free(pop);
}
```

(continues on next page)

(continued from previous page)

```
void derivee(double* x, Population* pop, double* dx)
{
    // Add openacc directives
    dx[0] =  pop->birth_rate[0]*x[0] - pop->death_rate[0]*x[0]*x[1];
    dx[1] = -pop->death_rate[1]*x[1] + pop->birth_rate[1]*x[0]*x[1];
}

void rk4(Population* pop, double dt)
{
        double* x_temp = (double*) malloc(2*sizeof(double));
        double* k1     = (double*) malloc(2*sizeof(double));
        double* k2     = (double*) malloc(2*sizeof(double));
        double* k3     = (double*) malloc(2*sizeof(double));
        double* k4     = (double*) malloc(2*sizeof(double));
        double halfdt  = dt / 2.0;

        // Add openacc directives
          for(int i=0; i<2; i++)
             x_temp[i] = pop->count[i];

          derivee(x_temp, pop, k1);
          for(int i=0; i<2; i++)
            x_temp[i] = pop->count[i] + k1[i]*halfdt;

          derivee(x_temp, pop, k2);
          for(int i=0; i<2; i++)
            x_temp[i] = pop->count[i] + k2[i]*halfdt;

          derivee(x_temp, pop, k3);
          for(int i=0; i<2; i++)
            x_temp[i] = pop->count[i] + k3[i]*dt;

          derivee(x_temp, pop, k4);
          for(int i=0; i<2; i++)
            pop->count[i] = pop->count[i] + (dt/6.0)*(k1[i] + 2.0*k2[i] + 2.0*k3[i] +␣
 ↪k4[i]);
}


int main(void)
{
    double ti   =   0.00;
    double dt   =   0.05;
    double tmax = 100.00;

    Population *pred_prey = (Population *) malloc(sizeof(Population));
    pred_prey->count      = (double*) malloc(2*sizeof(double));
    pred_prey->birth_rate = (double*) malloc(2*sizeof(double));
    pred_prey->death_rate = (double*) malloc(2*sizeof(double));

    pred_prey->count[1]      = 15.00;  // predator count
    pred_prey->birth_rate[1] =  0.01;  // predator birth rate
    pred_prey->death_rate[1] =  1.00;  // predator death rate

    pred_prey->count[0]      = 100.00; // prey count
    pred_prey->birth_rate[0] =   2.00; // prey birth rate
```

(continues on next page)

(continued from previous page)

```
    pred_prey->death_rate[0] =   0.02; // prey death rate

    FILE* fichier = fopen("output", "w");
    // Add openacc directives
    while (ti < tmax)
    {
        ti += dt;
        rk4(pred_prey, dt);
        for(int i=0; i<2; i++)
        {
            // Add openacc directives
        }
        fprintf(fichier, "%lf %s %lf %s %lf\n", ti,";", pred_prey->count[0],";",pred_
↪prey->count[1]);
    }
    fclose(fichier);
    free_pop(pred_prey);
    return 0;
}
```

```
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/C/Deep_copy_attach_detach_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>

typedef struct
{
    double* count;
    double* birth_rate;
    double* death_rate;
} Population;

void free_pop(Population* pop)
{
    #pragma acc exit data detach(pop->count, pop->birth_rate, pop->death_rate)
    #pragma acc exit data delete(pop->count, pop->birth_rate, pop->death_rate)
    #pragma acc exit data delete(pop)
```

(continues on next page)

(continued from previous page)

```
    free(pop->count);
    free(pop->birth_rate);
    free(pop->death_rate);
    free(pop);
}

void derivee(double* x, Population* pop, double* dx)
{
    #pragma acc serial present(pop, pop->birth_rate[0:2], pop->death_rate[0:2],␣
↪x[0:2], dx[0:2])
    {
    dx[0] =  pop->birth_rate[0]*x[0] - pop->death_rate[0]*x[0]*x[1];
    dx[1] = -pop->death_rate[1]*x[1] + pop->birth_rate[1]*x[0]*x[1];
    }
}

void rk4(Population* pop, double dt)
{
        double* x_temp = (double*) malloc(2*sizeof(double));
        double* k1     = (double*) malloc(2*sizeof(double));
        double* k2     = (double*) malloc(2*sizeof(double));
        double* k3     = (double*) malloc(2*sizeof(double));
        double* k4     = (double*) malloc(2*sizeof(double));
        double halfdt  = dt / 2.0;

        # pragma acc data create(k1[0:2], k2[0:2], k3[0:2], k4[0:2], x_temp[0:2])␣
↪present(pop, pop->birth_rate[0:2], pop->death_rate[0:2], pop->count[0:2])
        {

          #pragma acc parallel loop
          for(int i=0; i<2; i++)
             x_temp[i] = pop->count[i];

          derivee(x_temp, pop, k1);
          #pragma acc parallel loop
          for(int i=0; i<2; i++)
            x_temp[i] = pop->count[i] + k1[i]*halfdt;

          derivee(x_temp, pop, k2);
          #pragma acc parallel loop
          for(int i=0; i<2; i++)
            x_temp[i] = pop->count[i] + k2[i]*halfdt;

          derivee(x_temp, pop, k3);
          #pragma acc parallel loop
          for(int i=0; i<2; i++)
            x_temp[i] = pop->count[i] + k3[i]*dt;

          derivee(x_temp, pop, k4);
          #pragma acc parallel loop
          for(int i=0; i<2; i++)
            pop->count[i] = pop->count[i] + (dt/6.0)*(k1[i] + 2.0*k2[i] + 2.0*k3[i] +␣
↪k4[i]);
        }
}
```

(continues on next page)

(continued from previous page)

```
int main(void)
{
    double ti    =   0.00;
    double dt    =   0.05;
    double tmax = 100.00;

    Population *pred_prey = (Population *) malloc(sizeof(Population));
    pred_prey->count      = (double*) malloc(2*sizeof(double));
    pred_prey->birth_rate = (double*) malloc(2*sizeof(double));
    pred_prey->death_rate = (double*) malloc(2*sizeof(double));

    pred_prey->count[1]      = 15.00;  // predator count
    pred_prey->birth_rate[1] =  0.01;  // predator birth rate
    pred_prey->death_rate[1] =   1.0;  // predator death rate

    pred_prey->count[0]      = 100.00; //prey count
    pred_prey->birth_rate[0] =   2.00; // prey birth rate
    pred_prey->death_rate[0] =   0.02; // prey death rate

    #pragma acc enter data copyin(pred_prey->count[0:2], pred_prey->birth_rate[0:2],␣
↪pred_prey->death_rate[0:2])
    #pragma acc enter data copyin(pred_prey) attach(pred_prey->count, pred_prey->
↪birth_rate, pred_prey->death_rate)

    FILE* fichier = fopen("output_solution", "w");
    while (ti < tmax)
    {
        ti += dt;
        rk4(pred_prey, dt);
        for(int i=0; i<2; i++)
        {
            #pragma acc update self(pred_prey->count[i:1])
        }
        fprintf(fichier, "%lf %s %lf %s %lf\n", ti, ";", pred_prey->count[0], ";",␣
↪pred_prey->count[1]);
    }
    fclose(fichier);
    free_pop(pred_prey);
    return 0;
}
```

```
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:

```
double* array = (double*) malloc(size*sizeof(double));
#pragma acc enter data create(array[:size])

#pragma acc host_data use_device(array)
{
    // inside the block `array` stores the address on the GPU
    cuda_function(array);
}
```

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/C/Using_CUDA_random_example.c
#include <stdio.h>
#include <stdlib.h>
#include <curand.h>
#include <openacc.h>

// Fill d_buffer with num random numbers
void fill_rand(unsigned int *d_buffer, size_t num, cudaStream_t stream)
{
    curandGenerator_t gen;
    int status;
    // Create generator
```

(continues on next page)

(continued from previous page)

```
    status = curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
    // Set CUDA stream
    status |= curandSetStream(gen, stream);
    // Set seed
    status |= curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
    // Generate num random numbers
    status |= curandGenerate(gen, d_buffer, num);
//  Peut essayer curandStatus_tcurandGeneratePoisson(curandGenerator_t generator,␣
↪unsigned int *outputPtr, size_t n, double lambda)
    // Cleanup generator
    status |= curandDestroyGenerator(gen);

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

int main(void) {
    // Histogram allocation and initialization
    int histo[10];
    for (int i=0; i<10; ++i)
        histo[i] = 0;

    size_t nshots = (size_t) 1e9;
    cudaStream_t stream ;

    // Allocate memory for the random numbers
    unsigned int* shots = (unsigned int*) malloc(nshots*sizeof(unsigned int));
    #pragma acc data create(shots[:nshots]) copyout(histo[:10])
    {
        #pragma acc host_data use_device(shots)
        {
             stream = (cudaStream_t) acc_get_cuda_stream(acc_async_sync);
             fill_rand(shots, nshots, stream);
        }

        // Count the number of time each number was drawn
        #pragma acc parallel loop present(shots[:nshots])
        for (size_t i=0; i<nshots; ++i)
        {
            shots[i] = shots[i] % 10;
            #pragma acc atomic update
            histo[shots[i]]++;
        }
    }// End acc data

    // Print results
    for (int i=0; i<10; ++i)
        printf("%3d: %10d (%5.3f)\n", i, histo[i], (double) histo[i]/1.e9);

     return 0;
}
```

Requirements:

- Get started

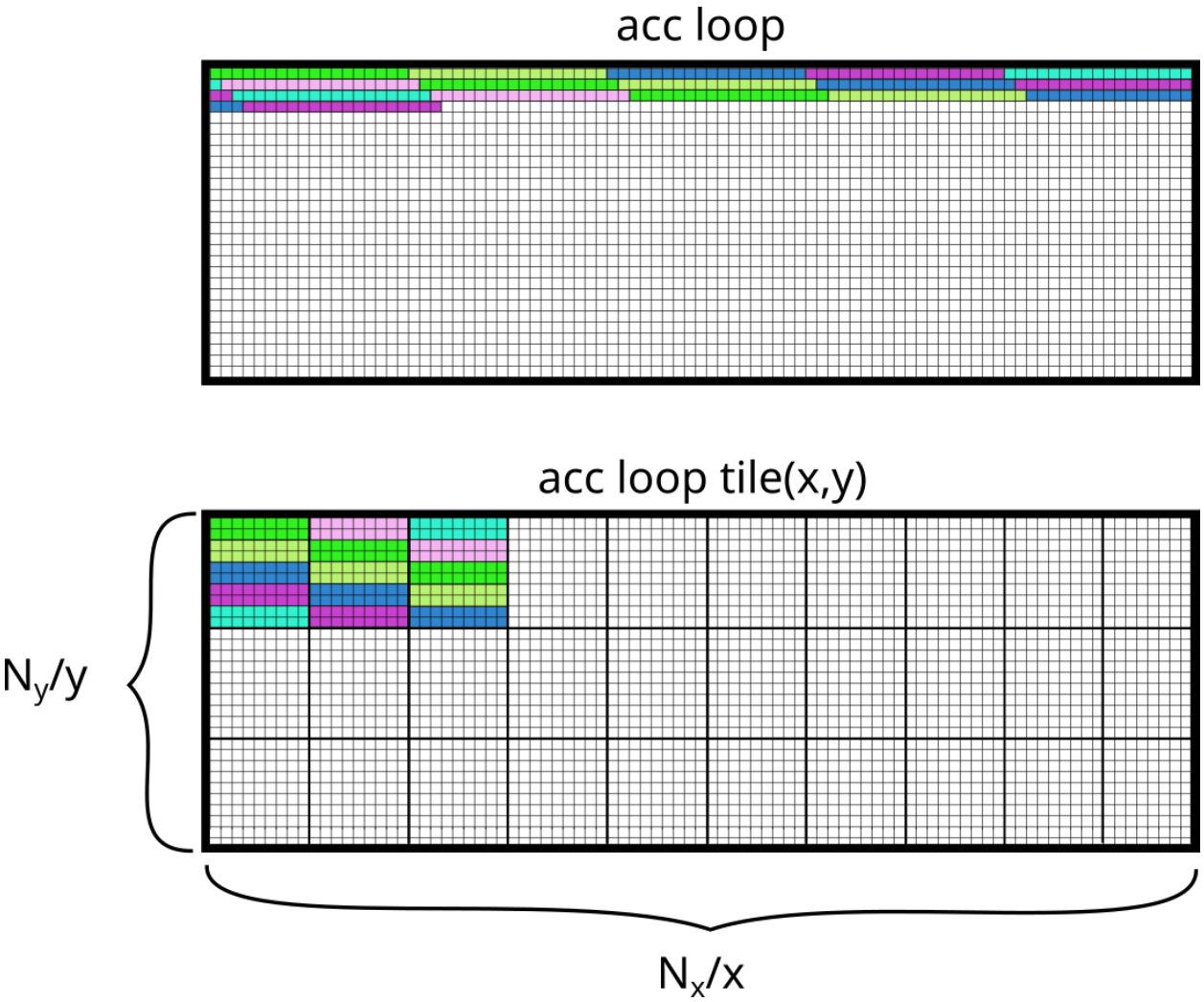
- Data Management

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.

```
#pragma acc loop tile(32,32)
for(int i = 0 ; i < size_i ; ++i)
{
    for(int j = 0 ; j < size_j ; ++j)
    {
        // A Fabulous calculation
    }
}
```

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/C/Loop_tiling_example_cpu.c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>

#define MIN(a,b) ( ((a)<(b))?(a):(b) )

double double_random(){
    return (double) (rand()) / RAND_MAX;
}

void nullify(int ni, int nj, double* d){
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           d[j+i*nj] = 0.0;
       }
    }
}

double checksum(int ni, int nj, double* d){
```

(continues on next page)

(continued from previous page)

```
    double dsum = 0.0;
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           dsum = dsum + d[j+i*nj];
       }
    }
    return dsum;
}

void naive_matmul(int ni, int nj, int nk, double* a, double* b, double* c, double* d){
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           for (int k=0; k<nk; k++){
                d[i*nj +j] = d[i*nj +j] + a[k+i*nk] * b[j+k*nj];
           }
           d[j+i*nj]= d[j+i*nj] + c[j+i*nj];
       }
    }
}

void tiled_matmul(int tile, int ni, int nj, int nk, double* a, double* b, double* c,␣
 ↪double* d){
     for (int i=0; i<ni; i+=tile){
       for (int j=0; j<nj; j+=tile){
          for (int ii=i; ii< MIN(i+tile,ni); ii++){
            for (int jj=j; jj<MIN(j+tile,nj); jj++){
                 for (int k=0; k<nk; k++){
                       d[ii*nj +jj] = d[ii*nj +jj] + a[k+ii*nk] * b[jj+k*nj];
                 }
            }
          }
       }
    }
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           d[j+i*nj]= d[j+i*nj] + c[j+i*nj];
       }
    }

}

int main(void)
{
    int ni=4280, nj=4024, nk=1960;

    clock_t t1, t2;

    double* a = (double*) malloc(ni*nk*sizeof(double));
    double* b = (double*) malloc(nk*nj*sizeof(double));
    double* c = (double*) malloc(ni*nj*sizeof(double));
    double* d = (double*) malloc(ni*nj*sizeof(double));
    double test;

    unsigned int seed = 1234;
    srand(seed);
```

(continues on next page)

(continued from previous page)

```
    for (int i=0; i<ni; i++){
       for (int k=0; k<nk; k++){
           a[k+i*nk] = double_random();
       }
    }

    for (int k=0; k<nk; k++){
       for (int j=0; j<nj; j++){
           b[j+k*nj] = double_random();
       }
    }

    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           c[j+i*nj] = 2.0;
       }
    }

    nullify(ni, nj, d);

    t1 = clock();
    naive_matmul(ni, nj, nk, a, b, c, d);
    t2 = clock();
    test = checksum(ni, nj, d);
    fprintf(stderr, "CPU naive Elapsed: %lf\n", (double) (t2-t1) /CLOCKS_PER_SEC);
    fprintf(stderr, "\tchecksum=%lf\n\n", test);
    nullify(ni, nj, d);

    int tile = 512;
    t1 = clock();
    tiled_matmul(tile, ni, nj, nk, a, b, c, d);
    t2 = clock();
    test = checksum(ni, nj, d);
    fprintf(stderr, "CPU Manually tiled Elapsed: %lf\n", (double) (t2-t1) /CLOCKS_
↪PER_SEC);
    fprintf(stderr, "\tchecsum=%lf\n\n", test);
    nullify(ni, nj, d);

    free(a);
    free(b);
    free(c);
    free(d);

    return 0;
}
```

And now it's GPU implementation.

```
%%idrrun -a
//  examples/C/Loop_tiling_example_gpu.c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>

#define MIN(a,b) ( ((a)<(b))?(a):(b) )
```

(continues on next page)

(continued from previous page)

```
double double_random(){
    return (double) (rand()) / RAND_MAX;
}

void nullify(int ni, int nj, double* d){
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           d[j+i*nj] = 0.0;
       }
    }
}

double checksum(int ni, int nj, double* d){
    double dsum = 0.0;
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           dsum = dsum + d[j+i*nj];
       }
    }
    return dsum;
}

void naive_matmul(int ni, int nj, int nk, double* a, double* b, double* c, double* d){
    #pragma acc parallel loop default(present)
    for (int i=0; i<ni; i++){
       #pragma acc loop
       for (int j=0; j<nj; j++){
           for (int k=0; k<nk; k++){
                d[i*nj +j] = d[i*nj +j] + a[k+i*nk] * b[j+k*nj];
           }
           d[j+i*nj]= d[j+i*nj] + c[j+i*nj];
       }
    }
}

void naive_matmul_acc_tiled(int ni, int nj, int nk, double* a, double* b, double* c,␣
↪double* d){
    #pragma acc parallel loop tile(32,32) default(present)
    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           for (int k=0; k<nk; k++){
                d[i*nj +j] = d[i*nj +j] + a[k+i*nk] * b[j+k*nj];
           }
           d[j+i*nj]= d[j+i*nj] + c[j+i*nj];
       }
    }
}


void tiled_matmul(int tile, int ni, int nj, int nk, double* a, double* b, double* c,␣
↪double* d){
     #pragma acc parallel loop default(present) num_workers(8) vector_length(128)
     #pragma acc loop gang collapse(2)
     for (int i=0; i<ni; i+=tile){
       for (int j=0; j<nj; j+=tile){
```

(continues on next page)

(continued from previous page)

```
          #pragma acc loop worker
          for (int ii=i; ii< MIN(i+tile,ni); ii++){
            #pragma acc loop vector
            for (int jj=j; jj<MIN(j+tile,nj); jj++){
                 #pragma acc loop seq
                 for (int k=0; k<nk; k++){
                       d[ii*nj +jj] = d[ii*nj +jj] + a[k+ii*nk] * b[jj+k*nj];
                 }
             }
          }
       }
    }
    #pragma acc parallel loop default(present)
    for (int i=0; i<ni; i++){
       #pragma acc loop
       for (int j=0; j<nj; j++){
           d[j+i*nj]= d[j+i*nj] + c[j+i*nj];
       }
    }

}

int main(void)
{
    int ni=4280, nj=4024, nk=1960;

    clock_t t1, t2;

    double* a = (double*) malloc(ni*nk*sizeof(double));
    double* b = (double*) malloc(nk*nj*sizeof(double));
    double* c = (double*) malloc(ni*nj*sizeof(double));
    double* d = (double*) malloc(ni*nj*sizeof(double));
    double test;

    unsigned int seed = 1234;
    srand(seed);

    for (int i=0; i<ni; i++){
       for (int k=0; k<nk; k++){
           a[k+i*nk] = double_random();
       }
    }

    for (int k=0; k<nk; k++){
       for (int j=0; j<nj; j++){
           b[j+k*nj] = double_random();
       }
    }

    for (int i=0; i<ni; i++){
       for (int j=0; j<nj; j++){
           c[j+i*nj] = 2.0;
       }
    }

    nullify(ni, nj, d);
```

(continues on next page)

(continued from previous page)

```
    #pragma acc data copyin(a[0:ni*nk], b[0:nk*nj], c[0:ni*nj]) create(d[0:ni*nj])
    {
    t1 = clock();
    naive_matmul(ni, nj, nk, a, b, c, d);
    t2 = clock();
    #pragma acc update self(d[0:ni*nj])
    test = checksum(ni, nj, d);
    fprintf(stderr, "GPU naive Elapsed: %lf\n", (double) (t2-t1) /CLOCKS_PER_SEC);
    fprintf(stderr, "\tchecksum=%lf\n\n", test);
    nullify(ni, nj, d);
    #pragma acc update device(d[0:ni*nj])

    t1 = clock();
    naive_matmul_acc_tiled(ni, nj, nk, a, b, c, d);
    t2 = clock();
    #pragma acc update self(d[0:ni*nj])
    test = checksum(ni, nj, d);
    fprintf(stderr, "GPU OpenACC tiled Elapsed: %lf\n", (double) (t2-t1) /CLOCKS_PER_
↪SEC);
    fprintf(stderr, "\tchecksum=%lf\n\n", test);
    nullify(ni, nj, d);
    #pragma acc update device(d[0:ni*nj])

    int tile = 512;
    t1 = clock();
    tiled_matmul(tile, ni, nj, nk, a, b, c, d);
    t2 = clock();
    #pragma acc update self(d[0:ni*nj])
    test = checksum(ni, nj, d);
    fprintf(stderr, "GPU Manually tiled Elapsed: %lf\n", (double) (t2-t1) /CLOCKS_
↪PER_SEC);
    fprintf(stderr, "\tchecksum=%lf\n\n", test);


    }

    free(a);
    free(b);
    free(c);
    free(d);

    return 0;
}
```

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/C/Loop_tiling_exercise.c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<openacc.h>

int main(void) {
    int    nx = 20000;
    int    ny = 10000;
    int    idx;
    double T[nx*ny], T_new[nx*ny];
    double erreur;

    for(int i=1; i<nx-1; ++i) {
        for(int j=1; j<ny-1; ++j) {
        T[i*ny+j]     = 0.0;
        T_new[i*ny+j] = 0.0;
        }
    }

    for(int i=0; i<nx; ++i){
        T[i*ny     ] = 100.0;
        T[i*ny+ny-1] = 0.0;
    }

    for(int j=0; j<ny; ++j){
        T[j]            = 0.0;
        T[(nx-1)*ny+j] = 0.0;
    }

    // add acc directive
    for (int it = 0; it<10000; ++it){
        erreur = 0.0;
        // add acc directive
        for (int i=1; i<nx-1; ++i) {
            for (int j=1; j<ny-1; ++j) {
                idx = i*(ny)+j;
                T_new[idx] = 0.25*(T[idx+ny]+T[idx-ny] + T[idx+1]+T[idx-1]);
                erreur = fmax(erreur, fabs(T_new[idx]-T[idx]));
            }
         }
        if(it%100 == 0) fprintf(stderr,"it: %d, erreur: %e\n",it,erreur);

        // add acc directive
        for (int i=1; i<nx-1; ++i) {
            for (int j=1; j<ny-1; ++j) {
                T[i*(ny)+j] =  T_new[i*(ny)+j];
            }
        }
    }
    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

18.5 Solution

```
%%idrrun -a
//  examples/C/Loop_tiling_solution.c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<openacc.h>

int main(void) {
    int    nx = 20000;
    int    ny = 10000;
    int    idx;
    double T[nx*ny], T_new[nx*ny];
    double erreur;

    for(int i=1; i<nx-1; ++i) {
        for(int j=1; j<ny-1; ++j) {
        T[i*ny+j]     = 0.0;
        T_new[i*ny+j] = 0.0;
        }
    }

    for(int i=0; i<nx; ++i){
        T[i*ny     ] = 100.0;
        T[i*ny+ny-1] = 0.0;
    }

    for(int j=0; j<ny; ++j){
        T[j]           = 0.0;
        T[(nx-1)*ny+j] = 0.0;
    }

#pragma acc data copy(T) create(T_new)
{
    for (int it = 0; it<10000; ++it){
        erreur = 0.0;
        #pragma acc parallel loop tile (32,32) reduction(max:erreur)
        for (int i=1; i<nx-1; ++i) {
            for (int j=1; j<ny-1; ++j) {
                idx = i*(ny)+j;
                T_new[idx] = 0.25*(T[idx+ny]+T[idx-ny] + T[idx+1]+T[idx-1]);
                erreur = fmax(erreur, fabs(T_new[idx]-T[idx]));
            }
        }
        if(it%100 == 0) fprintf(stderr,"it: %d, erreur: %e\n",it,erreur);

        #pragma acc parallel loop
        for (int i=1; i<nx-1; ++i) {
            for (int j=1; j<ny-1; ++j) {
                T[i*(ny)+j] =  T_new[i*(ny)+j];
```

(continues on next page)

(continued from previous page)

```
                }
            }
        }
}
    return 0;
}
```

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.

If you are having difficulties with some part of the code, you can take a look at the following advice:

First you need to copy the configuration file.

```
%%bash
cp ../../examples/C/conf.dat .
```

The hands-on starts here :

```
%%idrrun -a
//  examples/C/Hands_on_LJ_exercise.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
# include <math.h>
#include <float.h>
// Mass of the atoms (only 1 kind and normalized)
const double mass = 1.0;
// Boltzmann constant
const double kb = 0.831451115;

typedef struct
{
    size_t size;
    double* data;
} array;

typedef struct
{
    array* Fx;
    array* Fy;
    array* Fz;
} Forces;
```

(continues on next page)

(continued from previous page)

```
typedef struct
{
    array* vx;
    array* vy;
    array* vz;
    array* x;
    array* y;
    array* z;
} dynamic;

typedef struct
{
    size_t nsteps; // number of steps
    size_t dump_dyn; // number of steps between dumps
    char dump_file[20]; // number of steps between dumps
    double dt; // time step
    double lattice_length; // length of the box
    double Berendsen_T; // Temperature of the thermostat
    double Berendsen_coupling; // Temperature of the thermostat
    size_t NAtoms; // Number of atoms
    double LJ_sigma; // sigma parameter of the Lennard-Jones potential
    double LJ_epsilon; // epsilon parameter of the Lennard-Jones potential
    double LJ_cutoff; // cutoff of the Lennard-Jones potential
    double LJ_tolerance; // cutoff of the Lennard-Jones potential
} Config;

/**
 * Management of the dynamic struct
 */
dynamic* initialize_dyn(Config* conf, int random); //done
void free_dyn(dynamic* dyn); //done
array* allocate_array(size_t size); //done
void free_array(array* ar); //done

/**
 * Dynamic
 */
void velocity_verlet(dynamic* dyn, Forces* forces, Config* conf); //done
double LJ_pot(double rij2, double sigma2, double epsilon2);// done
void forces_from_LJ(dynamic* dyn, Forces* forces, Config* conf); //done
double berendsen_thermostat(dynamic* dyn, Config*);
void sd(dynamic* dyn,
        Forces* forces,
        Config* conf,
        double step_length,
        double threshold,
        size_t max_steps);
double stat_forces(Forces* forces, Config* conf);

#pragma acc routine seq
inline double LJ_pot(double rij2, double sigma2, double epsilon)
{
    return epsilon * pow((2.0 * (sigma2/rij2)), 6) - pow((2.0 * (sigma2/rij2)), 3);
}
```

(continues on next page)

(continued from previous page)

```
void forces_from_LJ(dynamic* dyn, Forces* forces, Config* conf)
{
    double sigma2 = conf->LJ_sigma*conf->LJ_sigma;
    double rij2, xij, yij, zij = 0.0;
    double potential_energy=0.0;

    for (size_t i=0; i<conf->NAtoms; ++i)
    {
        forces->Fx->data[i] = 0.;
        forces->Fy->data[i] = 0.;
        forces->Fz->data[i] = 0.;
    }

    for (size_t i=0; i<conf->NAtoms; ++i)
        for (size_t j=0; j<conf->NAtoms; ++j)
        {
            xij =  (dyn->x->data[j] - dyn->x->data[i]);
            yij =  (dyn->y->data[j] - dyn->y->data[i]);
            zij =  (dyn->z->data[j] - dyn->z->data[i]);
            // Apply Periodic Boundary Conditions
            xij -= floor(xij/conf->lattice_length + 0.5) *conf->lattice_length;
            yij -= floor(yij/conf->lattice_length + 0.5) *conf->lattice_length;
            zij -= floor(zij/conf->lattice_length + 0.5) *conf->lattice_length;
            rij2 = xij*xij + yij*yij + zij*zij;

            if ((rij2 > conf->LJ_tolerance) && (rij2 < conf->LJ_cutoff * conf->LJ_
 ↪cutoff))
            {
                potential_energy += LJ_pot(rij2, sigma2, conf->LJ_epsilon);
                forces->Fx->data[i] += 24.0*LJ_pot(rij2, sigma2, conf->LJ_
 ↪epsilon)*xij/sqrt(rij2);
                forces->Fy->data[i] += 24.0*LJ_pot(rij2, sigma2, conf->LJ_
 ↪epsilon)*yij/sqrt(rij2);
                forces->Fz->data[i] += 24.0*LJ_pot(rij2, sigma2, conf->LJ_
 ↪epsilon)*zij/sqrt(rij2);
                forces->Fx->data[j] -= 24.0*LJ_pot(rij2, sigma2, conf->LJ_
 ↪epsilon)*xij/sqrt(rij2);
                forces->Fy->data[j] -= 24.0*LJ_pot(rij2, sigma2, conf->LJ_
 ↪epsilon)*yij/sqrt(rij2);
                forces->Fz->data[j] -= 24.0*LJ_pot(rij2, sigma2, conf->LJ_
 ↪epsilon)*zij/sqrt(rij2);
            }
        }
    printf("Epot= %15.5e ", potential_energy);
}


double stat_forces(Forces* forces, Config* conf)
{
    double Fmax = 0.;
    double Fmin = DBL_MAX;
    double Fnorm = 0.;
    double F = 0.;
    for (int i =0; i< conf->NAtoms; ++i)
    {
        F = forces->Fx->data[i]*forces->Fx->data[i]+
```

(continues on next page)

(continued from previous page)

```
            forces->Fy->data[i]*forces->Fy->data[i]+
            forces->Fz->data[i]*forces->Fz->data[i];
        if (F < Fmin) Fmin = F;
        if (F > Fmax) Fmax = F;
        Fnorm += F;
    }
    printf("<F>= %10.3e min(F)= %10.3e max(F)= %10.3e ", sqrt(Fnorm)/conf->NAtoms,␣
↪sqrt(Fmin), sqrt(Fmax));
    return sqrt(Fnorm)/conf->NAtoms;
}

void velocity_verlet(dynamic* dyn, Forces* forces, Config* conf)
{
    for (size_t i=0; i < conf->NAtoms; ++i)
    {
        dyn->vx->data[i] += 0.5 * conf->dt * forces->Fx->data[i];
        dyn->vy->data[i] += 0.5 * conf->dt * forces->Fy->data[i];
        dyn->vz->data[i] += 0.5 * conf->dt * forces->Fz->data[i];

        dyn->x->data[i] += conf->dt*dyn->vx->data[i];
        dyn->y->data[i] += conf->dt*dyn->vy->data[i];
        dyn->z->data[i] += conf->dt*dyn->vz->data[i];

        // Apply the Periodic Boundary Conditions
        dyn->x->data[i] -= floor(dyn->x->data[i]/conf->lattice_length + 0.5) * conf->
↪lattice_length;
        dyn->y->data[i] -= floor(dyn->y->data[i]/conf->lattice_length + 0.5) * conf->
↪lattice_length;
        dyn->z->data[i] -= floor(dyn->z->data[i]/conf->lattice_length + 0.5) * conf->
↪lattice_length;
    }

    forces_from_LJ(dyn, forces, conf);

    for (size_t i=0; i < conf->NAtoms; ++i)
    {

        dyn->vx->data[i] += 0.5 * conf->dt * forces->Fx->data[i];
        dyn->vy->data[i] += 0.5 * conf->dt * forces->Fy->data[i];
        dyn->vz->data[i] += 0.5 * conf->dt * forces->Fz->data[i];
    }
}

/**
 * Read/write configuration
 */
Config* read_params(char* filepath); //done
void read_initial(char* filepath, dynamic* dyn);
void write_step(char* filepath, dynamic* dyn);

void free_array(array* ar)
{
    free(ar->data);
    free(ar);
}
```

(continues on next page)

(continued from previous page)

```
array* allocate_array(size_t size)
{
    array* ar = (array*) malloc(sizeof(array));
    ar->size = size;
    ar->data = (double*) malloc(size*sizeof(double));

    return ar;
}

void free_forces(Forces* forces)
{
    free_array(forces->Fx);
    free_array(forces->Fy);
    free_array(forces->Fz);

    free(forces);
}

void free_dyn(dynamic* dyn)
{
    free_array(dyn->x);
    free_array(dyn->y);
    free_array(dyn->z);
    free_array(dyn->vx);
    free_array(dyn->vy);
    free_array(dyn->vz);

    free(dyn);
}

void update_array(array* ar, size_t size, int gpu)
{
    if (gpu)
    {

    }
    else
    {

    }
}

void update_dyn(dynamic* dyn, Config* conf, int gpu)
{
    update_array(dyn->x, conf->NAtoms, gpu);
    update_array(dyn->y, conf->NAtoms, gpu);
    update_array(dyn->z, conf->NAtoms, gpu);
    update_array(dyn->vx, conf->NAtoms, gpu);
    update_array(dyn->vy, conf->NAtoms, gpu);
    update_array(dyn->vz, conf->NAtoms, gpu);
}

/**
 * Initialize the structures for the dynamic
 * If random is >0 we generate a grid on which we place the atoms
 */
```

(continues on next page)

(continued from previous page)

```
dynamic* initialize_dyn(Config* conf, int random)
{
    size_t id = 0;
    size_t n = floor(pow(conf->NAtoms,1./3.))+1;
    size_t leftover = conf->NAtoms - n*n*n;
    printf("%d %d %d\n", leftover, leftover/n/n, leftover%(n*n));
    double s = conf->lattice_length/(double) n;
    dynamic* dyn = (dynamic*) malloc(sizeof(dynamic));

    dyn->x  = allocate_array(conf->NAtoms);
    dyn->y  = allocate_array(conf->NAtoms);
    dyn->z  = allocate_array(conf->NAtoms);
    dyn->vx = allocate_array(conf->NAtoms);
    dyn->vy = allocate_array(conf->NAtoms);
    dyn->vz = allocate_array(conf->NAtoms);
    if (random > 0)
    {
        srand(47329);
        for (size_t i=0; i<n; ++i)
        {
            for (size_t j=0; j<n; ++j)
            {
                for (size_t k=0; k<n; ++k)
                {
                    id = i*n*n + j*n + k;
                    if (id >= conf->NAtoms) break;
                    dyn->x->data[id] = s*((double)i + 0.5) + (double)rand()/RAND_MAX␣
↪* 0.3*s;
                    dyn->y->data[id] = s*((double)j + 0.5) + (double)rand()/RAND_MAX␣
↪* 0.3*s;
                    dyn->z->data[id] = s*((double)k + 0.5) + (double)rand()/RAND_MAX␣
↪* 0.3*s;
                    dyn->vx->data[id] = 0.;//(double)rand()/RAND_MAX * 5.0 - 2.5;
                    dyn->vy->data[id] = 0.;//(double)rand()/RAND_MAX * 5.0 - 2.5;
                    dyn->vz->data[id] = 0.;//(double)rand()/RAND_MAX * 5.0 - 2.5;
                }
                if (id >= conf->NAtoms) break;
            }
            if (id >= conf->NAtoms) break;
        }
        // Apply PBC
        for (int i=0; i<conf->NAtoms; ++i)
        {
            dyn->x->data[i] -= floor(dyn->x->data[i]/conf->lattice_length + 0.5) *␣
↪conf->lattice_length;
            dyn->y->data[i] -= floor(dyn->y->data[i]/conf->lattice_length + 0.5) *␣
↪conf->lattice_length;
            dyn->z->data[i] -= floor(dyn->z->data[i]/conf->lattice_length + 0.5) *␣
↪conf->lattice_length;
        }
    }

    int gpu=1;
    update_dyn(dyn, conf, gpu);
    return dyn;
}
```

(continues on next page)

(continued from previous page)

```
/**
 * Initialize Forces
 */
Forces* initialize_forces(Config* conf)
{
    Forces* forces = (Forces*) malloc(sizeof(Forces));

    forces->Fx = allocate_array(conf->NAtoms);
    forces->Fy = allocate_array(conf->NAtoms);
    forces->Fz = allocate_array(conf->NAtoms);

    for (size_t i=0; i<conf->NAtoms; ++i)
    {
        forces->Fx->data[i] = 0.;
        forces->Fy->data[i] = 0.;
        forces->Fz->data[i] = 0.;
    }
    return forces;
}

/**
 * Read the configuration
 */
Config* read_params(char* filepath)
{
    FILE* fp = fopen(filepath, "r");
    char* line = NULL;
    size_t len = 0;
    char key[20], val[20];

    Config* conf = (Config*) malloc(sizeof(Config));
    if (fp == NULL)
        exit(EXIT_FAILURE);

    while ((getline(&line, &len, fp)) != -1)
    {
        sscanf(line, "%s %s", key, val);
        if (strcmp(key, "T") == 0)
        {
            conf->Berendsen_T = atof(val);
        } else if (strcmp(key, "nsteps") == 0){
            conf->nsteps = atoi(val);
        } else if (strcmp(key, "dump_dyn") == 0){
            conf->dump_dyn = atoi(val);
        } else if (strcmp(key, "dump_file") == 0){
             strcpy(conf->dump_file, val);
        } else if (strcmp(key, "dt") == 0){
            conf->dt = atof(val);
        } else if (strcmp(key, "tau") == 0){
            conf->Berendsen_coupling = atof(val);
        } else if (strcmp(key, "lattice") == 0){
            conf->lattice_length = atof(val);
        } else if (strcmp(key, "LJ_sigma") == 0){
            conf->LJ_sigma = atof(val);
        } else if (strcmp(key, "LJ_epsilon") == 0){
```

(continues on next page)

(continued from previous page)

```
            conf->LJ_epsilon = atof(val);
        } else if (strcmp(key, "LJ_cutoff") == 0){
            conf->LJ_cutoff = atof(val);
        } else if (strcmp(key, "LJ_tolerance") == 0){
            conf->LJ_tolerance = atof(val);
        } else if (strcmp(key, "natoms") == 0){
            conf->NAtoms = atoi(val);
        }
    }
    fclose(fp);

    return conf;
}

void print_conf(Config* conf)
{
    printf("natoms %d\n", conf->NAtoms);
    printf("dt %f\n", conf->dt);
    printf("nsteps %d\n", conf->nsteps);
    printf("tau %f\n", conf->Berendsen_coupling);
    printf("T %f\n", conf->Berendsen_T);
    printf("lattice %f\n", conf->lattice_length);
    printf("LJ_sigma %f\n", conf->LJ_sigma);
    printf("LJ_epsilon %f\n", conf->LJ_epsilon);
    printf("LJ_cutoff %f\n", conf->LJ_cutoff);
}

void dump_dyn(dynamic* dyn, Config* conf, char* mode)
{
    FILE* fp = fopen(conf->dump_file, mode);
    fprintf(fp, "%d\n", conf->NAtoms);
    fprintf(fp, "%10.5f\n", conf->lattice_length);
    for (int i=0; i<conf->NAtoms; ++i)
    {
        fprintf(fp, "Ne %15.10f %15.10f %15.10f %15.8e %15.8e %15.8e\n",
                dyn->x->data[i], dyn->y->data[i], dyn->z->data[i],
                dyn->vx->data[i], dyn->vy->data[i], dyn->vz->data[i]);
    }
    fclose(fp);
}

double berendsen_thermostat(dynamic* dyn, Config* conf)
{
    double kinetic_E = 0.0;

    for (size_t i=0; i<conf->NAtoms; ++i)
        kinetic_E += 1.0 * (dyn->vx->data[i]*dyn->vx->data[i])
                         + (dyn->vy->data[i]*dyn->vy->data[i])
                         + (dyn->vz->data[i]*dyn->vz->data[i]);

    kinetic_E *= 0.5;
    double T = 2.0 * kb * kinetic_E/(3.0 * conf->NAtoms -3);
    double lambda_scaling = sqrt(1 + (conf->dt/conf->Berendsen_coupling) * (conf->
 ↪Berendsen_T/T-1));
    printf("T= %15.6e l= %10.3e ", T, lambda_scaling);
```

(continues on next page)

(continued from previous page)

```
    for (size_t i=0; i < conf->NAtoms; ++i)
    {
        dyn->vx->data[i] *= lambda_scaling;
        dyn->vy->data[i] *= lambda_scaling;
        dyn->vz->data[i] *= lambda_scaling;
    }

    return T;
}

int main(int argc, char** argv)
{
    double T;
    int cpu=0;
    Config* conf = read_params("conf.dat");
    dynamic* dyn = initialize_dyn(conf, 1);
    Forces* forces = initialize_forces(conf);
    forces_from_LJ(dyn, forces, conf);
    dump_dyn(dyn, conf, "w");
//    sd(dyn, forces, conf, 0.0001, 0.0001, 2000);
    for (int i=0; i<conf->nsteps; ++i)
    {
        printf("Step %6d ",i);
        velocity_verlet(dyn, forces, conf);
        stat_forces(forces, conf);
        T = berendsen_thermostat(dyn, conf);
        if (i > 100 && T > conf->Berendsen_T*1000)
        {
            fprintf(stderr, "Oups something went wrong with T\n");
            break;
        }
        if (i%100 == 0)
        {
            update_dyn(dyn, conf, cpu);
            dump_dyn(dyn, conf, "a");
        }
        printf("\n");
    }
    update_dyn(dyn, conf, cpu);
    dump_dyn(dyn, conf, "a");
    free_dyn(dyn);
    free_forces(forces);
    return 0;
}
```

19.2 Solution

```
%%idrrun -a
//  examples/C/Hands_on_LJ_solution.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
# include <math.h>
#include <float.h>
// Mass of the atoms (only 1 kind and normalized)
const double mass = 1.0;
// Boltzmann constant
const double kb = 0.831451115;

typedef struct
{
    size_t size;
    double* data;
} array;

typedef struct
{
    array* Fx;
    array* Fy;
    array* Fz;
} Forces;

typedef struct
{
    array* vx;
    array* vy;
    array* vz;
    array* x;
    array* y;
    array* z;
} dynamic;

typedef struct
{
    size_t nsteps; // number of steps
    size_t dump_dyn; // number of steps between dumps
    char dump_file[20]; // number of steps between dumps
    double dt; // time step
    double lattice_length; // length of the box
    double Berendsen_T; // Temperature of the thermostat
    double Berendsen_coupling; // Temperature of the thermostat
    size_t NAtoms; // Number of atoms
    double LJ_sigma; // sigma parameter of the Lennard-Jones potential
    double LJ_epsilon; // epsilon parameter of the Lennard-Jones potential
    double LJ_cutoff; // cutoff of the Lennard-Jones potential
    double LJ_tolerance; // cutoff of the Lennard-Jones potential
} Config;

/**
 * Management of the dynamic struct
 */
```

(continues on next page)

(continued from previous page)

```
dynamic* initialize_dyn(Config* conf, int random); //done
void free_dyn(dynamic* dyn); //done
array* allocate_array(size_t size); //done
void free_array(array* ar); //done

/**
 * Dynamic
 */
void velocity_verlet(dynamic* dyn, Forces* forces, Config* conf); //done
double LJ_pot(double rij2, double sigma2, double epsilon2);// done
void forces_from_LJ(dynamic* dyn, Forces* forces, Config* conf); //done
double berendsen_thermostat(dynamic* dyn, Config*);
void sd(dynamic* dyn,
        Forces* forces,
        Config* conf,
        double step_length,
        double threshold,
        size_t max_steps);
double stat_forces(Forces* forces, Config* conf);

#pragma acc routine seq
inline double LJ_pot(double rij2, double sigma2, double epsilon)
{
    return epsilon * pow((2.0 * (sigma2/rij2)), 6) - pow((2.0 * (sigma2/rij2)), 3);
}

void forces_from_LJ(dynamic* dyn, Forces* forces, Config* conf)
{
    double sigma2 = conf->LJ_sigma*conf->LJ_sigma;
    double rij2, xij, yij, zij = 0.0;
    double potential_energy=0.0;

    #pragma acc parallel loop present(forces, dyn, conf)\
                              present(forces->Fx,forces->Fx->data[:conf->NAtoms])\
                              present(forces->Fy,forces->Fy->data[:conf->NAtoms])\
                              present(forces->Fz,forces->Fz->data[:conf->NAtoms])
    for (size_t i=0; i<conf->NAtoms; ++i)
    {
        forces->Fx->data[i] = 0.;
        forces->Fy->data[i] = 0.;
        forces->Fz->data[i] = 0.;
    }

    #pragma acc parallel loop present(forces, dyn, conf)\
                              copy(potential_energy) reduction(+:potential_energy)\
                              present(forces->Fx,forces->Fx->data[:conf->NAtoms])\
                              present(forces->Fy,forces->Fy->data[:conf->NAtoms])\
                              present(forces->Fz,forces->Fz->data[:conf->NAtoms])\
                              present(dyn->x,dyn->x->data[:conf->NAtoms])\
                              present(dyn->y,dyn->y->data[:conf->NAtoms])\
                              present(dyn->z,dyn->z->data[:conf->NAtoms])
    for (size_t i=0; i<conf->NAtoms; ++i)
        #pragma acc loop private(xij, yij, zij, rij2)
        for (size_t j=0; j<conf->NAtoms; ++j)
        {
            xij =  (dyn->x->data[j] - dyn->x->data[i]);
```

(continues on next page)

(continued from previous page)

```
            yij =  (dyn->y->data[j] - dyn->y->data[i]);
            zij =  (dyn->z->data[j] - dyn->z->data[i]);
            // Apply Periodic Boundary Conditions
            xij -= floor(xij/conf->lattice_length + 0.5) *conf->lattice_length;
            yij -= floor(yij/conf->lattice_length + 0.5) *conf->lattice_length;
            zij -= floor(zij/conf->lattice_length + 0.5) *conf->lattice_length;
            rij2 = xij*xij + yij*yij + zij*zij;

            if ((rij2 > conf->LJ_tolerance) && (rij2 < conf->LJ_cutoff * conf->LJ_
↪cutoff))
            {
                potential_energy += LJ_pot(rij2, sigma2, conf->LJ_epsilon);
                forces->Fx->data[i] += 24.0*LJ_pot(rij2, sigma2, conf->LJ_
↪epsilon)*xij/sqrt(rij2);
                forces->Fy->data[i] += 24.0*LJ_pot(rij2, sigma2, conf->LJ_
↪epsilon)*yij/sqrt(rij2);
                forces->Fz->data[i] += 24.0*LJ_pot(rij2, sigma2, conf->LJ_
↪epsilon)*zij/sqrt(rij2);
                forces->Fx->data[j] -= 24.0*LJ_pot(rij2, sigma2, conf->LJ_
↪epsilon)*xij/sqrt(rij2);
                forces->Fy->data[j] -= 24.0*LJ_pot(rij2, sigma2, conf->LJ_
↪epsilon)*yij/sqrt(rij2);
                forces->Fz->data[j] -= 24.0*LJ_pot(rij2, sigma2, conf->LJ_
↪epsilon)*zij/sqrt(rij2);
            }
        }
    printf("Epot= %15.5e ", potential_energy);
}


double stat_forces(Forces* forces, Config* conf)
{
    double Fmax = 0.;
    double Fmin = DBL_MAX;
    double Fnorm = 0.;
    double F = 0.;
    #pragma acc parallel loop reduction(min:Fmin) reduction(max:Fmax)␣
↪reduction(+:Fnorm)\
                              private(F) copy(Fmin, Fmax, Fnorm)\
                              present(forces, forces->Fx, forces->Fy, forces->Fz)\
                              present(forces->Fx->data[:forces->Fx->size])\
                              present(forces->Fy->data[:forces->Fy->size])\
                              present(forces->Fz->data[:forces->Fz->size])
    for (int i =0; i< conf->NAtoms; ++i)
    {
        F = forces->Fx->data[i]*forces->Fx->data[i]+
            forces->Fy->data[i]*forces->Fy->data[i]+
            forces->Fz->data[i]*forces->Fz->data[i];
        if (F < Fmin) Fmin = F;
        if (F > Fmax) Fmax = F;
        Fnorm += F;
    }
    printf("<F>= %10.3e min(F)= %10.3e max(F)= %10.3e ", sqrt(Fnorm)/conf->NAtoms,␣
↪sqrt(Fmin), sqrt(Fmax));
    return sqrt(Fnorm)/conf->NAtoms;
}
```

(continues on next page)

(continued from previous page)

```
void velocity_verlet(dynamic* dyn, Forces* forces, Config* conf)
{
    #pragma acc parallel loop present(conf, dyn, forces, dyn->vx, dyn->vx->data[:conf-
 ↪>NAtoms])\
                              present(dyn->vy,dyn->vy->data[:conf->NAtoms])\
                              present(dyn->vz,dyn->vz->data[:conf->NAtoms])\
                              present(dyn->x,dyn->x->data[:conf->NAtoms])\
                              present(dyn->y,dyn->y->data[:conf->NAtoms])\
                              present(dyn->z,dyn->z->data[:conf->NAtoms])\
                              present(forces->Fx,forces->Fx->data[:conf->NAtoms])\
                              present(forces->Fy,forces->Fy->data[:conf->NAtoms])\
                              present(forces->Fz,forces->Fz->data[:conf->NAtoms])
    for (size_t i=0; i < conf->NAtoms; ++i)
    {
        dyn->vx->data[i] += 0.5 * conf->dt * forces->Fx->data[i];
        dyn->vy->data[i] += 0.5 * conf->dt * forces->Fy->data[i];
        dyn->vz->data[i] += 0.5 * conf->dt * forces->Fz->data[i];

        dyn->x->data[i] += conf->dt*dyn->vx->data[i];
        dyn->y->data[i] += conf->dt*dyn->vy->data[i];
        dyn->z->data[i] += conf->dt*dyn->vz->data[i];

        // Apply the Periodic Boundary Conditions
        dyn->x->data[i] -= floor(dyn->x->data[i]/conf->lattice_length + 0.5) * conf->
 ↪lattice_length;
        dyn->y->data[i] -= floor(dyn->y->data[i]/conf->lattice_length + 0.5) * conf->
 ↪lattice_length;
        dyn->z->data[i] -= floor(dyn->z->data[i]/conf->lattice_length + 0.5) * conf->
 ↪lattice_length;
    }

    forces_from_LJ(dyn, forces, conf);

    #pragma acc parallel loop present(conf, forces, dyn)\
                              present(dyn->vx,dyn->vx->data[:conf->NAtoms])\
                              present(dyn->vy,dyn->vy->data[:conf->NAtoms])\
                              present(dyn->vz,dyn->vz->data[:conf->NAtoms])\
                              present(forces->Fx,forces->Fx->data[:conf->NAtoms])\
                              present(forces->Fy,forces->Fy->data[:conf->NAtoms])\
                              present(forces->Fz,forces->Fz->data[:conf->NAtoms])
    for (size_t i=0; i < conf->NAtoms; ++i)
    {

        dyn->vx->data[i] += 0.5 * conf->dt * forces->Fx->data[i];
        dyn->vy->data[i] += 0.5 * conf->dt * forces->Fy->data[i];
        dyn->vz->data[i] += 0.5 * conf->dt * forces->Fz->data[i];
    }
}

/**
 * Read/write configuration
 */
Config* read_params(char* filepath); //done
void read_initial(char* filepath, dynamic* dyn);
void write_step(char* filepath, dynamic* dyn);
```

(continues on next page)

(continued from previous page)

```
void free_array(array* ar)
{
    free(ar->data);
    free(ar);
}

array* allocate_array(size_t size)
{
    array* ar = (array*) malloc(sizeof(array));
    ar->size = size;
    ar->data = (double*) malloc(size*sizeof(double));
    #pragma acc enter data create(ar, ar->data[:size]) copyin(ar->size)
    return ar;
}

void free_forces(Forces* forces)
{
    free_array(forces->Fx);
    free_array(forces->Fy);
    free_array(forces->Fz);
    #pragma acc exit data delete(forces)
    free(forces);
}

void free_dyn(dynamic* dyn)
{
    free_array(dyn->x);
    free_array(dyn->y);
    free_array(dyn->z);
    free_array(dyn->vx);
    free_array(dyn->vy);
    free_array(dyn->vz);
    #pragma acc exit data delete(dyn)
    free(dyn);
}

void update_array(array* ar, size_t size, int gpu)
{
    if (gpu)
    {
        #pragma acc update device(ar->data[:size])
    }
    else
    {
        #pragma acc update self(ar->data[:size])
    }
}

void update_dyn(dynamic* dyn, Config* conf, int gpu)
{
    update_array(dyn->x, conf->NAtoms, gpu);
    update_array(dyn->y, conf->NAtoms, gpu);
    update_array(dyn->z, conf->NAtoms, gpu);
    update_array(dyn->vx, conf->NAtoms, gpu);
    update_array(dyn->vy, conf->NAtoms, gpu);
```

(continues on next page)

(continued from previous page)

```
    update_array(dyn->vz, conf->NAtoms, gpu);
}

/**
 * Initialize the structures for the dynamic
 * If random is >0 we generate a grid on which we place the atoms
 */
dynamic* initialize_dyn(Config* conf, int random)
{
    size_t id = 0;
    size_t n = floor(pow(conf->NAtoms,1./3.))+1;
    size_t leftover = conf->NAtoms - n*n*n;
    printf("%d %d %d\n", leftover, leftover/n/n, leftover%(n*n));
    double s = conf->lattice_length/(double) n;
    dynamic* dyn = (dynamic*) malloc(sizeof(dynamic));
    #pragma acc enter data create(dyn)
    dyn->x  = allocate_array(conf->NAtoms);
    dyn->y  = allocate_array(conf->NAtoms);
    dyn->z  = allocate_array(conf->NAtoms);
    dyn->vx = allocate_array(conf->NAtoms);
    dyn->vy = allocate_array(conf->NAtoms);
    dyn->vz = allocate_array(conf->NAtoms);
    if (random > 0)
    {
        srand(47329);
        for (size_t i=0; i<n; ++i)
        {
            for (size_t j=0; j<n; ++j)
            {
                for (size_t k=0; k<n; ++k)
                {
                    id = i*n*n + j*n + k;
                    if (id >= conf->NAtoms) break;
                    dyn->x->data[id] = s*((double)i + 0.5) + (double)rand()/RAND_MAX␣
↪* 0.3*s;
                    dyn->y->data[id] = s*((double)j + 0.5) + (double)rand()/RAND_MAX␣
↪* 0.3*s;
                    dyn->z->data[id] = s*((double)k + 0.5) + (double)rand()/RAND_MAX␣
↪* 0.3*s;
                    dyn->vx->data[id] = 0.;//(double)rand()/RAND_MAX * 5.0 - 2.5;
                    dyn->vy->data[id] = 0.;//(double)rand()/RAND_MAX * 5.0 - 2.5;
                    dyn->vz->data[id] = 0.;//(double)rand()/RAND_MAX * 5.0 - 2.5;
                }
                if (id >= conf->NAtoms) break;
            }
            if (id >= conf->NAtoms) break;
        }
        // Apply PBC
        for (int i=0; i<conf->NAtoms; ++i)
        {
            dyn->x->data[i] -= floor(dyn->x->data[i]/conf->lattice_length + 0.5) *␣
↪conf->lattice_length;
            dyn->y->data[i] -= floor(dyn->y->data[i]/conf->lattice_length + 0.5) *␣
↪conf->lattice_length;
            dyn->z->data[i] -= floor(dyn->z->data[i]/conf->lattice_length + 0.5) *␣
↪conf->lattice_length;
```

(continues on next page)

(continued from previous page)

```
        }
    }

    int gpu=1;
    update_dyn(dyn, conf, gpu);
    return dyn;
}

/**
 * Initialize Forces
 */
Forces* initialize_forces(Config* conf)
{
    Forces* forces = (Forces*) malloc(sizeof(Forces));
    #pragma acc enter data create(forces)
    forces->Fx = allocate_array(conf->NAtoms);
    forces->Fy = allocate_array(conf->NAtoms);
    forces->Fz = allocate_array(conf->NAtoms);
    #pragma acc parallel loop present(forces, conf,forces->Fx, forces->Fy, forces->Fz,
 ↪ forces->Fx->data[:conf->NAtoms])\
                              present(forces->Fy->data[:conf->NAtoms])\
                              present(forces->Fz->data[:conf->NAtoms])
    for (size_t i=0; i<conf->NAtoms; ++i)
    {
        forces->Fx->data[i] = 0.;
        forces->Fy->data[i] = 0.;
        forces->Fz->data[i] = 0.;
    }
    return forces;
}

/**
 * Read the configuration
 */
Config* read_params(char* filepath)
{
    FILE* fp = fopen(filepath, "r");
    char* line = NULL;
    size_t len = 0;
    char key[20], val[20];

    Config* conf = (Config*) malloc(sizeof(Config));
    if (fp == NULL)
        exit(EXIT_FAILURE);

    while ((getline(&line, &len, fp)) != -1)
    {
        sscanf(line, "%s %s", key, val);
        if (strcmp(key, "T") == 0)
        {
            conf->Berendsen_T = atof(val);
        } else if (strcmp(key, "nsteps") == 0){
            conf->nsteps = atoi(val);
        } else if (strcmp(key, "dump_dyn") == 0){
            conf->dump_dyn = atoi(val);
        } else if (strcmp(key, "dump_file") == 0){
```

(continues on next page)

(continued from previous page)

```
            strcpy(conf->dump_file, val);
        } else if (strcmp(key, "dt") == 0){
            conf->dt = atof(val);
        } else if (strcmp(key, "tau") == 0){
            conf->Berendsen_coupling = atof(val);
        } else if (strcmp(key, "lattice") == 0){
            conf->lattice_length = atof(val);
        } else if (strcmp(key, "LJ_sigma") == 0){
            conf->LJ_sigma = atof(val);
        } else if (strcmp(key, "LJ_epsilon") == 0){
            conf->LJ_epsilon = atof(val);
        } else if (strcmp(key, "LJ_cutoff") == 0){
            conf->LJ_cutoff = atof(val);
        } else if (strcmp(key, "LJ_tolerance") == 0){
            conf->LJ_tolerance = atof(val);
        } else if (strcmp(key, "natoms") == 0){
            conf->NAtoms = atoi(val);
        }
    }
    fclose(fp);
    #pragma acc enter data copyin(conf)
    return conf;
}

void print_conf(Config* conf)
{
    printf("natoms %d\n", conf->NAtoms);
    printf("dt %f\n", conf->dt);
    printf("nsteps %d\n", conf->nsteps);
    printf("tau %f\n", conf->Berendsen_coupling);
    printf("T %f\n", conf->Berendsen_T);
    printf("lattice %f\n", conf->lattice_length);
    printf("LJ_sigma %f\n", conf->LJ_sigma);
    printf("LJ_epsilon %f\n", conf->LJ_epsilon);
    printf("LJ_cutoff %f\n", conf->LJ_cutoff);
}

void dump_dyn(dynamic* dyn, Config* conf, char* mode)
{
    FILE* fp = fopen(conf->dump_file, mode);
    fprintf(fp, "%d\n", conf->NAtoms);
    fprintf(fp, "%10.5f\n", conf->lattice_length);
    for (int i=0; i<conf->NAtoms; ++i)
    {
        fprintf(fp, "Ne %15.10f %15.10f %15.10f %15.8e %15.8e %15.8e\n",
                dyn->x->data[i], dyn->y->data[i], dyn->z->data[i],
                dyn->vx->data[i], dyn->vy->data[i], dyn->vz->data[i]);
    }
    fclose(fp);
}

double berendsen_thermostat(dynamic* dyn, Config* conf)
{
    double kinetic_E = 0.0;
    #pragma acc parallel loop present(dyn) reduction(+:kinetic_E) \
                            copy(kinetic_E)\
```

(continues on next page)

(continued from previous page)

```
                            present(dyn->vx, dyn->vx->data[:conf->NAtoms])\
                            present(dyn->vy, dyn->vy->data[:conf->NAtoms])\
                            present(dyn->vz, dyn->vz->data[:conf->NAtoms])
    for (size_t i=0; i<conf->NAtoms; ++i)
        kinetic_E += 1.0 * (dyn->vx->data[i]*dyn->vx->data[i])
                         + (dyn->vy->data[i]*dyn->vy->data[i])
                         + (dyn->vz->data[i]*dyn->vz->data[i]);

    kinetic_E *= 0.5;
    double T = 2.0 * kb * kinetic_E/(3.0 * conf->NAtoms -3);
    double lambda_scaling = sqrt(1 + (conf->dt/conf->Berendsen_coupling) * (conf->
↪Berendsen_T/T-1));
    printf("T= %15.6e l= %10.3e ", T, lambda_scaling);

    #pragma acc parallel loop present(dyn)\
                            copyin(lambda_scaling)\
                            present(dyn->vx, dyn->vx->data[:conf->NAtoms])\
                            present(dyn->vy, dyn->vy->data[:conf->NAtoms])\
                            present(dyn->vz, dyn->vz->data[:conf->NAtoms])
    for (size_t i=0; i < conf->NAtoms; ++i)
    {
        dyn->vx->data[i] *= lambda_scaling;
        dyn->vy->data[i] *= lambda_scaling;
        dyn->vz->data[i] *= lambda_scaling;
    }

    return T;
}

int main(int argc, char** argv)
{
    double T;
    int cpu=0;
    Config* conf = read_params("conf.dat");
    dynamic* dyn = initialize_dyn(conf, 1);
    Forces* forces = initialize_forces(conf);
    forces_from_LJ(dyn, forces, conf);
    dump_dyn(dyn, conf, "w");
//    sd(dyn, forces, conf, 0.0001, 0.0001, 2000);
    for (int i=0; i<conf->nsteps; ++i)
    {
        printf("Step %6d ",i);
        velocity_verlet(dyn, forces, conf);
        stat_forces(forces, conf);
        T = berendsen_thermostat(dyn, conf);
        if (i > 100 && T > conf->Berendsen_T*1000)
        {
            fprintf(stderr, "Oups something went wrong with T\n");
            break;
        }
        if (i%100 == 0)
        {
            update_dyn(dyn, conf, cpu);
            dump_dyn(dyn, conf, "a");
        }
        printf("\n");
```

(continues on next page)

(continued from previous page)

```
    }
    update_dyn(dyn, conf, cpu);
    dump_dyn(dyn, conf, "a");
    free_dyn(dyn);
    free_forces(forces);
    return 0;
}
```

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

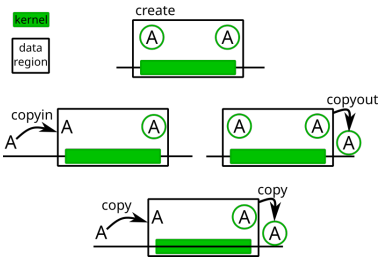
To choose the right data clause you need to answer the following questions:

- 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)