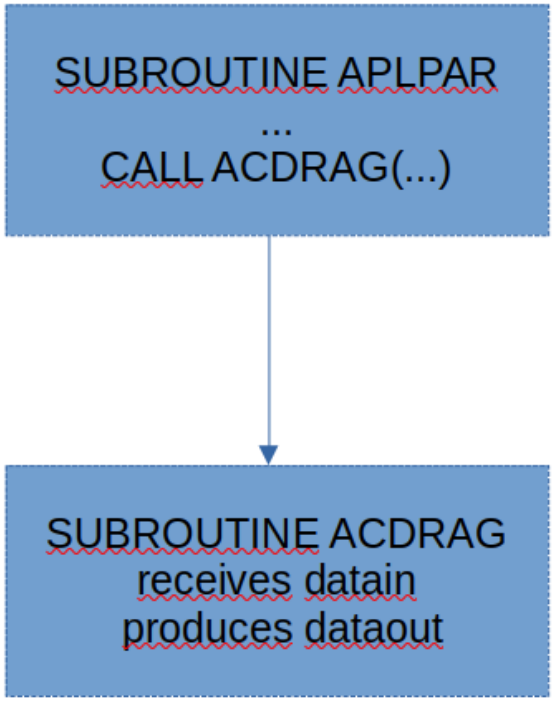
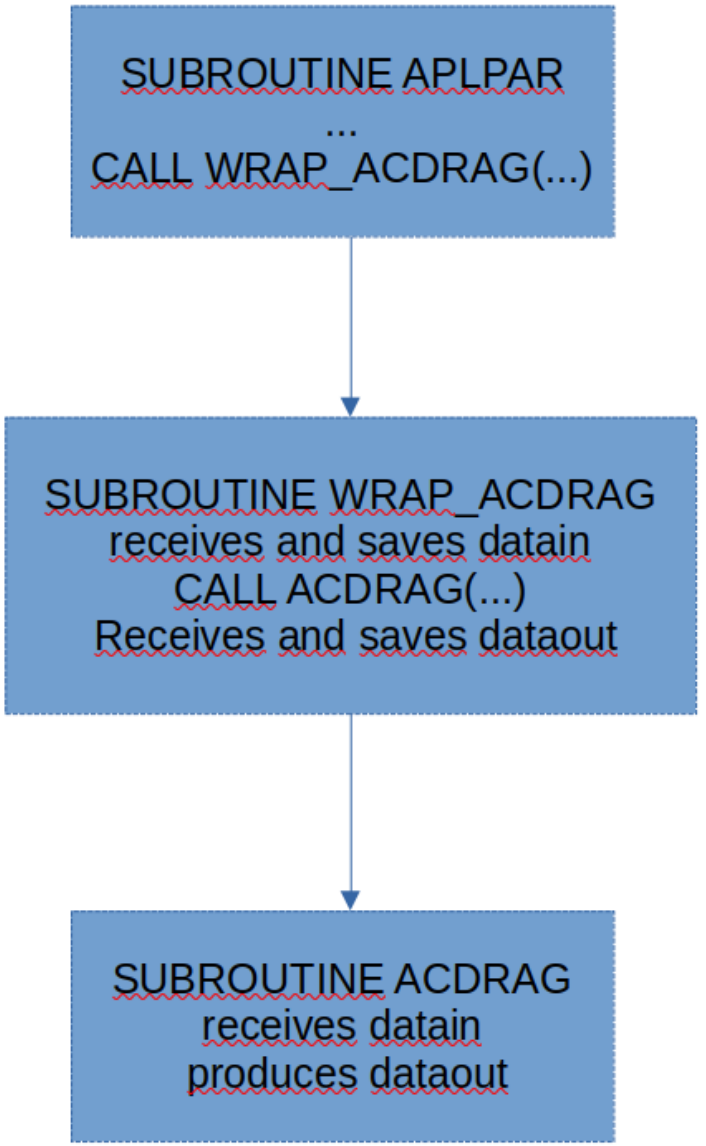
Extracting a test case from a subroutine in IAL

Cossevin Erwan, Penigaud Nicolas

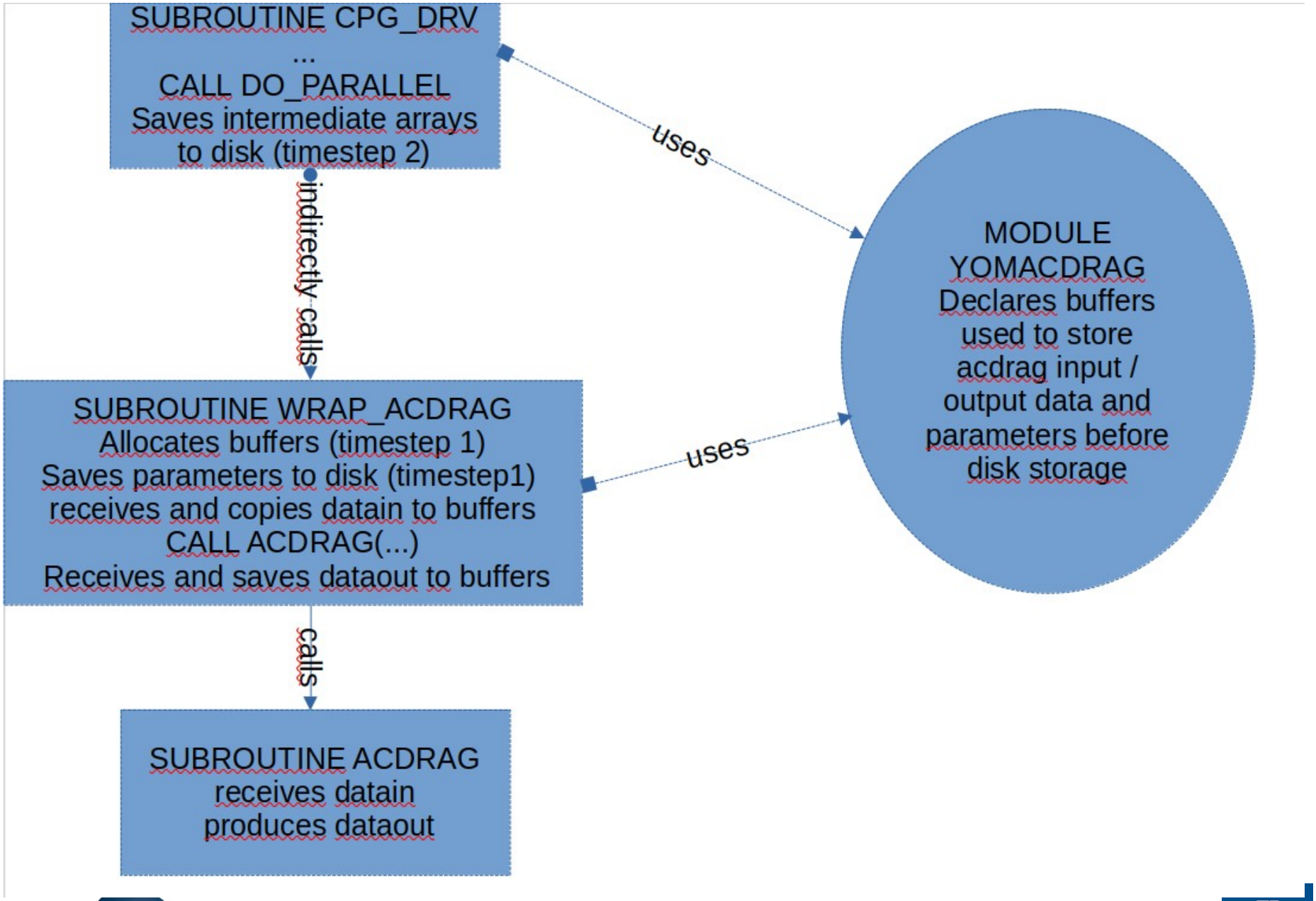
Summary

Organisation (1/2)





Organisation (2/2)



General presentation

- Extract a small test case from the code => run a subroutine outside of the whole code
- When you have a tricky bugs, it can be useful to extract just one routine from the code and run the code just for this routine
- It’s easier to work on a routine rather than all the code
- There are two parts (each with several steps detailled below) :

  part A) Extract the data needed with a wrapper

  part B) Create the test case by moving out all the routines you need

Environment

- This lab session is run on ecmwf hpc
- To compile the code, you must use ecinteractive : https://confluence.ecmwf.int/display/UDOC/HPC2020%3A+Persistent+interactive+job+with+ecinteractive

  Connexion : ecinteractive -c 32 -m 32G -s 60G

  Restore SSD: ec_restore_local_ssd -r

  go to SSD: cd $LOCALSSD

- The code is compiled with IAL-bundle :

  Source: https://github.com/dhaumont/IAL-bundle/tree/CY50T1

  Presentation: https://www.accord-nwp.org/IMG/pdf/cmake_compilation_in_ial-bundle.pdf

  Compilation: https://github.com/dhaumont/IAL-bundle/blob/CY50T1/doc/cmake_compilation.md#quick-start

Environment

- Use this branch of IAL_bundle :

  https://github.com/pmarguinaud/IAL-bundle/tree/CY50T1-ecmwf-nhvpc25.7%2Boptwrap

- Compile the bundle with this command :

  ./ial-bundle build --arch arch/ecmwf/hpc2020/intel/2023.2.0/hpcx-openmpi/2.9.0 --install-dir=WHERE_YOU_WANT &> out.log

- The install-dir is the place from which the code will be run, you can for example put it in $PERM

Solution code

- Solution to part A : the branch in which the wrapper is coded is : https://github.com/ecossevin/IAL/tree/create_wrapper_acdrag

- Solution to part B : the branch with the extracted test case is :

  https://github.com/ecossevin/acdrag/tree/master

- Some files to help (starting point for each step mentionned):

  to extract the data : https://github.com/ecossevin/acdrag/tree/extract_test_case

  to generate util methods:

  https://github.com/ecossevin/acdrag/tree/generate_util

  to run the test_case: https://github.com/ecossevin/acdrag/tree/run_test_case

- You can find all that at : /perm/rma1/test_case_working_week

  on ecmwf cluster

A) Extract the data

- The idea is to replace the call to the subroutine we want to extract by a call to a wrapper ; the wrapper copies the data for every NPROMA block, saves the the parameters to disk, and calls the original subroutine

- A module containing arrays with an extra block dimension is created, whole fields are stored in this module :

  example of a variable of this module :

```
REAL(KIND=JPRB), ALLOCATABLE, DIMENSION(:, :, :) :: ZZAPRS
```

A) Extract the data

- Data extraction can be separated in 5 steps :

  1) write a module to store the arrays in which data will be copied

  2) write the wrapper

  3) changes in cpg_drv

  4) generate the util_xxx_mod files

  5) run the code to generate the data

- For each step, the code is available :
  /perm/rma1/test_case_working_week

  PATH_TO_THE_CODE or/and GIT_BRANCH

A) Extract the data :
1) write the yomacdrag module

You need to create a module containing arrays with an extra block dimension. The input and output field of your subroutine are going to be stored in this array.

The module also contains other variable :

LLSAVE : to save the data only for the first process

IGBLKS : total number of blocks

JBLK : Index of the current block

LLALLOCATED : true when arrays of the module are allocated, the allocation has to be done just once, for the first block

A) Extract the data :
1) write the yomacdrag module

```
USE PARKIND1

IMPLICIT NONE

LOGICAL :: LLSAVE = .FALSE.

INTEGER :: IGPBLKS
INTEGER :: JBLK

!$OMP THREADPRIVATE (JBLK)

LOGICAL :: LLALLOCATED = .FALSE.

REAL(KIND=JPRB), ALLOCATABLE, DIMENSION(:, :, :) :: ZZAPRS
```

The blueprint of this module is available here :
/perm/rma1/test_case_working_week/extract_test_case/yomacdrag_blueprint.F90

A) Extract the data
2) write the wrapper

The goal of the wrapper subroutine is to save input data, call the original data and then save the output data

The wrapper is called in the original code instead of the subroutine

1) The blueprint of the wrapper is available here : /perm/rma1/test_case_working_week/extract_test_case/wrapper_blueprint.F90

A) Extract the data
2) write the wrapper

Different steps :

1) Fields are allocated for all blocks once

```
IF (.NOT. LLALLOCATED) THEN
    ALLOCATE(ZZAPRS(KLON, 0:KLEV, IGPBLKS))
```

2) Save derived types and integers only for first block

```
IF (LLSAVE .AND. (JBLK==1)) THEN
    ILUN = 77
    OPEN (ILUN, FILE='ACDRAG.CONST.dat', FORM='UNFORMATTED')

    SAVE(YDCST, ILUN)
    SAVE(KLEV, ILUN)
```

A) Extract the data
2) write the wrapper

3) Copy the input data of the current block

```
ZZAPRS(:, :, JBLK) = PAPRS(:, :)
```

4) Call the original subroutine

```
PSTRDU_OUT(:, :, JBLK) = PSTRDU(: ,:)
```

5) Copy the ouput data of the current block

A) Extract the data
3) Changes in cpg_drv

- cpg_drv.F90 is the subroutine that calls all the grid point calculation of the model, there is the upper loop on the blocks that calls the whole model's physic.

- In the loop over the block, the wrapper is called for each block, and data is saved for each block.

- After the loop over the blocks, the data can be saved for all the blocks

```
IF (LLSAVE) THEN
    OPEN (ILUN, FILE='ACDRAG.IN.dat', FORM='UNFORMATTED')
    CALL SAVE(ZZAPRS(:, :, 1:IGPBLKS - 1), ILUN)
    CLOSE(ILUN)

    OPEN (ILUN, FILE='ACDRAG.OUT.dat', FORM='UNFORMATTED')
    CALL SAVE(PSTRDU_OUT(:, :, 1:IGPBLKS - 1), ILUN)
    CLOSE(ILUN)
ENDIF
```

A) Extract the data
3) Changes in cpg_drv

- The data is saved just for the first process, and not for the first time step

```
LLSAVE = (MYPROC==1) .AND. (NSTEP==2)
```

You can find cpg_drv blue_print here :

- /perm/rma1/test_case_working_week/extract_test_case/cpg_drv_blueprint.F90

A) Extract the data
4) Generate the util_xxx_mod

- To run the test case we need 4 special types of methods : SAVE, LOAD, COPY, WIPE
- SAVE : to save derived types in a file
- LOAD : to load derived types from a file
- COPY : to copy derived types from the host to the device
- WIPE : to remove derived types from the device

A) Extract the data
4) Generate the util_xxx_mod

- The SAVE_ACDC method used below is generated by a script :

```
USE MODEL_PHYSICS_MF_MOD , ONLY : MODEL_PHYSICS_MF_TYPE
IMPLICIT NONE
TYPE(MODEL_PHYSICS_MF_TYPE),INTENT(IN):: YDML_PHY_MF

...

CALL SAVE_ACDC(YDML_PHY_MF, ILUN)
```

- They are many types in this module, only three are used in acdrag :

```
MODULE MODEL_PHYSICS_MF_MOD

!$ACDC methods

 USE YOMPHY    , ONLY : TPHY
 USE YOMPHY0   , ONLY : TPHY0
 USE YOMPHY1   , ONLY : TPHY1
 USE YOMPHY2   , ONLY : TPHY2
 USE YOMPHY3   , ONLY : TPHY3
 USE YOMAERO   , ONLY : TAERO
 USE TYPE_AERO, ONLY : TAERO_AVG
 ...
```

A) Extract the data
4) Generate the util_xxx_mod

- A new module is created outside of the code, containing only the types needed

```
MODULE MODEL_PHYSICS_MF_MOD

  USE YOMPHY   , ONLY : TPHY
  USE YOMPHY0  , ONLY : TPHY0
  USE YOMPHY2  , ONLY : TPHY2
```

- The script is going to be run on this new reduced module

- You must do the same for TPHY, TPHY0 and TPHY2 types

A) Extract the data
4) Generate the util_xxx_mod

- Add /home/sor/fxtran/bin to your PATH

- To generate the util_xxx_mod run this command :

  /home/rma1/fxtran-acdc/bin/fxtran-f90 --dryrun --method methods --methods save,load,copy,wipe -- f90 --dir tmp -c ./tmp/yomphy0.F90

  yomphy0.F90 must be in the ./tmp directory, all the generated files will be in the current directory

- The util_xxx_files can be found :
  /perm/rma1/test_case_working_week/generate_util

  see fxtran.sh script to generate the files

A) Extract the data
5) Run the code to generate the data

- You need a bash a script to run the model, we want to extract big and small data. To extract the big data, run the model with a smaller number of MPI task, each task will compute more points (recall that we save the data only for the first MPI task).

- You can find a script to run the model here :

  /perm/rma1/test_case_working_week/acdrag/run_small.sh

B) Creating the test case

- Now that the data is extracted, the test case can be created

- Steps of the test case creation :

  1) create the main_acdrag.F90 file

  2) move all the files to a new directory (new git repo if you want to save your test case on git)

  3) compile the code

  4) run the code

B) Creating the test case
1) create the main_acdrag.F90 file

- The main programs :

  1) loads the data

  2) initialize all the variables

  3) calls the subroutine of the test case from a loop on the blocks wrapped inside a loop on the time steps

- You can find the blueprint of the main.F90 file here:

  /perm/rma1/test_case_working_week/run_test_case/main_blueprint.F90

B) Creating the test case
1) create the main_acdrag.F90 file

1) Initialization of the variables with the GETOPTION method

```
CALL INITOPTIONS

NPROMA  = 0; CALL GETOPTION ("--nproma",  NPROMA)
```

2) Load the arrays with a block dimension

```
SUBROUTINE LOADALL

ILUNCI = 77
ILUNFI = 78

OPEN (ILUNCI, NAME=TRIM (CLCASE_IN)//'/ACDRAG.CONST.dat', FORM='UNFORMATTED')
OPEN (ILUNFI, NAME=TRIM (CLCASE_IN)//'/ACDRAG.IN.dat',    FORM='UNFORMATTED')
CALL LOAD(ILUNFI, ZZAPRS, YDD=YLD)
```

B) Creating the test case
1) create the main_acdrag.F90 file

3) Create the loop over the time and the loop over the blocks, from wich the routine is called

```
CALL GET_TIME (TSC)

DO ITIME = 1, NTIME

  IF (TRIM (CLMETHOD) == 'openmp') THEN

!$OMP PARALLEL DO PRIVATE (JBLK)
    DO JBLK = 1, NGPBLKS

      CALL ACDRAG(...
```

4) Compute the differences with the OUTPUT fields from the model

```
IF (LLDIFF) THEN
  CALL DIFFALL
ENDIF
```

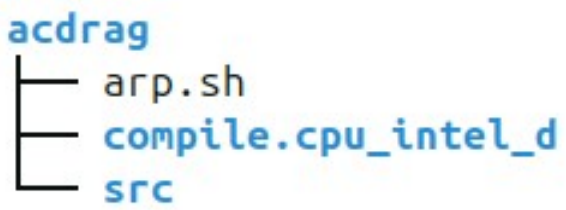
B) Creating the test case
1) create the main_acdrag.F90 file

To get the number of blocks inside are yomacdrag module :

```
IGPBLKS = YDGEO%YRDIM%NGPBLKS
```

B) Creating the test case
3) Compiling the code

- To compile, your test case directory must have this structure :



- See /perm/rma1/test_case_working_week/acdrag

- Then you go in the compile.cpu_intel_d directory and run this command : fxtran-makemaker –SRC=../src to create the Makefile

  if you have trouble with this command, you can have a look at /perm/rma1/test_case_working_week/acdrag/make.sh to have the right env.

- When compiling, you must simplify the files, only with what is needed, in order

B) Creating the test case
3) Compiling the code

- You will need many modules from IAL
- To lower the number of files you copy in the test case, reduce code only to what is needed. The test case must remain as simple as possible. See /perm/rma1/test_case_working_week/acdrag/src/yomlun.F90 as an example.
- You will need the get_time function : see /perm/rma1/test_case_working_week/run_test_case/get_time.c

B) Creating the test case
4) Run the code : arp.sh script to run the code

```
#!/bin/bash
#SBATCH  ...

set -x

ulimit -s unlimited
export OMP_STACKSIZE=4G
export OMP_NUM_THREADS=8

SUBMIT_DIR=./turb.$$
mkdir $SUBMIT_DIR
cd $SUBMIT_DIR

arch=cpu_intel_d

for method in openmp
do
../compile.${arch}/main_acdrag.x \
  --case-in PATH_TO_YOUR_DATA \
  --verbose  --diff  \
  --nproma 32        \
  --method $method > $method.txt 2>&1
done
```

Depends on the machine you are using

You can find this script there : /perm/rma1/test_case_working_week/acdrag/run_small.sh

Other methods can be added, for example openaccsinglecolumn or openaccmanyblocks

C) Using fxtran-acdc to compile for GPU

Within the acdrag directory, a new directory, compile.gpu_nvhpc_d, is created with the same structure as compile.cpu_intel_d.

The compilation option for CPU or GPU are located in Makefile.inc, in each compilation directory.

Before compiling for GPU, env.gpu_nvhpc_d can be sourced to set the right environment. Likewise, env.cpu_intel.d for CPU.

This Makefile.inc also defines boot, which enables creation of an fxtran-acdc subdirectory inside compile.cpu_intel_d and compile.gpu_nvhpc_d.

compile.gpu_nvhpc_d contains headers and library libFxtranACDC.a containing auxiliary files compiled from acdrag/fxtran-acdc/src

C) Using fxtran-acdc to compile for GPU

Within each compilation directory, compile.cpu_intel.d and compile.gpu_nvhpc_d, a directory user-out is present, with the generated code.

It is possible to add a user-in directory, which can be used, for example, for debugging : if a routine from user-out is modified and copied to user-in, the modified routine will be used during the next compilation process initiated by make in this compilation directory.

When generating bit-reproducible routines, it is necessary to export BITREPCPP=1, and to recompile compile.cpu_intel_d/fxtran-acdc and compile.gpu_nvhpc_d/fxtran-acdc if necessary.

To do so, rm the previous compile.cpu_intel_d directory and make boot in compile.cpu_intel_d ; likewise in compile.gpu_nvhpc_d.

C) Using fxtran-acdc to compile for GPU

The scripts to convert CPU code to GPU code can be parameterized, in each routine in src with ACDC directives, and in the configuration file src/fxtran.conf with options.

For this practical on acdrag, there in only one routine in which to add ACDC directives, ACDRAG.F90. In the screen capture below, directives are inserted to produce an openaccsinglecolumn version of ACDRAG.F90, and an openaccmanyblocks version of ACDRAG.F90.

```
!OPTIONS XOPT(NOEVAL)
SUBROUTINE ACDRAG (YDCST, YDML_PHY_MF,KIDIA,KFDIA,KLON,KTDIA,KLEV,&
 !---------------------------------------------------------------------
 ! - INPUT  2D .
 & PAPRS,PAPRSF,PDELP,PNBVNO,PRDELP,PU,PV,&
 ! - INPUT  1D .
 & PRCORI,PGETRL,PGWDCS,PVRLAN,PVRLDI,&
 ! - OUTPUT 2D .
 & PSTRDU,PSTRDV,PRAPTRAJ)
!$ACDC singlecolumn
!$ACDC manyblocks --max-statements-per-parallel=20
```

C) Using fxtran-acdc to compile for GPU

The calls to the generated routines need to be added separately to main_acdrag.F90, as seen below for openaccsingleblock ; the rest of the code is generated by the scripts.

```
  IF (TRIM (CLMETHOD) == 'openmp') THEN

!$OMP PARALLEL DO PRIVATE (JBLK)
    DO JBLK = 1, NGPBLKS
      CALL ACDRAG(YDCST, YDML_PHY_MF,KIDIA,KFDIA,KLON,KTDIA,KLEV,&
      & ZZAPRS(:, :, JBLK), ZZAPRSF(:, :, JBLK),  &
      & ZZDELP(:, :, JBLK), ZZNBVNO(:, :, JBLK),  &
      & ZZRDELP(:, :, JBLK), ZZU(:, :, JBLK), ZZV(:, :, JBLK),  &
      & ZZRCORI(:, JBLK), ZZGETRL(:, JBLK), ZZGWDCS(:, JBLK),  &
      & ZZVRLAN(:, JBLK), ZZVRLDI(:, JBLK), ZZSTRDU(:, :, JBLK),  &
      & ZZSTRDV(:, :, JBLK), ZZRAPTRAJ(:, :, JBLK))
    ENDDO !jblk
!$OMP END PARALLEL DO

  ELSEIF (TRIM (CLMETHOD) == 'openaccsinglecolumn') THEN

    !$ACC PARALLEL LOOP GANG PRIVATE(JBLK) PRESENT(YFXTRAN_ACDC_STACK) VECTOR_LENGTH(NPROMA)
    DO JBLK=1,NGPBLKS
      !$ACC LOOP VECTOR PRIVATE(JLON,YLSTACK)
      DO JLON=1,NPROMA
        YLSTACK%L8=fxtran_acdc_stack_l8(YFXTRAN_ACDC_STACK,JBLK,NGPBLKS)
        YLSTACK%U8=fxtran_acdc_stack_u8(YFXTRAN_ACDC_STACK,JBLK,NGPBLKS)
        YLSTACK%L4=fxtran_acdc_stack_l4(YFXTRAN_ACDC_STACK,JBLK,NGPBLKS)
        YLSTACK%U4=fxtran_acdc_stack_u4(YFXTRAN_ACDC_STACK,JBLK,NGPBLKS)

        CALL ACDRAG_OPENACC(YDCST, YDML_PHY_MF,JLON,JLON,KLON,KTDIA,KLEV,&
        & ZZAPRS(:, :, JBLK), ZZAPRSF(:, :, JBLK),  &
        & ZZDELP(:, :, JBLK), ZZNBVNO(:, :, JBLK),  &
        & ZZRDELP(:, :, JBLK), ZZU(:, :, JBLK), ZZV(:, :, JBLK),  &
        & ZZRCORI(:, JBLK), ZZGETRL(:, JBLK), ZZGWDCS(:, JBLK),  &
        & ZZVRLAN(:, JBLK), ZZVRLDI(:, JBLK), ZZSTRDU(:, :, JBLK),  &
        & ZZSTRDV(:, :, JBLK), ZZRAPTRAJ(:, :, JBLK),YDSTACK=YLSTACK)

      ENDDO
    ENDDO
```

C) Using fxtran-acdc to compile for GPU

Here are example of fxtran configuration files. In this practical on acdrag, we will only change the two options for bit reproducibility (on the right) :
1) --use-bit-repro-intrinsics replaces, for instance, a call to COS by a call to FXTRAN_ACDC_BR_COS, which is bit-reproducible on CPU and GPU
2) --use-bit-repro-parens adds parentheses on the sums, for bit reproducibility between NVHPC and other compilers.

```
[
  '--object-merge-method', 'concatenate',
  '--stack84',
  '--methods-list', 'save,load,copy,wipe',
  '--type-bound-methods',
  '--merge-interfaces',
  '--parallelmethod-section',
  '--numbered-submodules',
  '--use-stack-manyblocks',
  '--stack-method',
  '--tmp', ]
```

```
[
  '--object-merge-method', 'concatenate',
  '--stack84',
  '--methods-list', 'save,load,copy,wipe',
  '--type-bound-methods',
  '--merge-interfaces',
  '--parallelmethod-section',
  '--numbered-submodules',
  '--use-bit-repro-intrinsics',
  '--use-bit-repro-parens',
  '--use-stack-manyblocks',
  '--stack-method',
  '--tmp', '1' ]
```

C) Using fxtran-acdc to compile for GPU

To generate bit-reproducible routines, it is also necessary to include ACDC directives in the ACDRAG.F90 code, as seen below :
The second ACDC directive specifies the name of the generated routine (to be added in MAIN_ACDRAG.F90), the fourth ACDC directive mandates bit-reproducibility.
The call to the generated bit-reproducible routine, ACDRAG_OPENACC_BITREPRO needs to be added separately to MAIN_ACDRAG.F90.

```
!OPTIONS XOPT(NOEVAL)
SUBROUTINE ACDRAG (YDCST, YDML_PHY_MF,KIDIA,KFDIA,KLON,KTDIA,KLEV,&
 !-----------------------------------------------------------------------
 ! - INPUT  2D .
 & PAPRS,PAPRSF,PDELP,PNBVNO,PRDELP,PU,PV,&
 ! - INPUT  1D .
 & PRCORI,PGETRL,PGWDCS,PVRLAN,PVRLDI,&
 ! - OUTPUT 2D .
 & PSTRDU,PSTRDV,PRAPTRAJ)
!$ACDC singlecolumn
!$ACDC singlecolumn --suffix-singlecolumn _OPENACC_BITREPRO --use-bit-repro-intrinsics
!$ACDC manyblocks --max-statements-per-parallel=20
!$ACDC bitrepro
```

C) Using fxtran-acdc to compile for GPU

```
#!/bin/bash
#SBATCH -p gpu
#SBATCH --job-name=arp_acdrag
#SBATCH --nodes=1
#SBATCH --gres=gpu:4
#SBATCH --time 00:15:00



set -x

module load nvidia/24.5

SUBMIT_DIR=./acdrag_gpu.$$
mkdir $SUBMIT_DIR
cd $SUBMIT_DIR
arch=gpu_nvhpc_d


for method in openaccsinglecolumn openaccmanyblocks
do
../compile.${arch}/main_acdrag.x \
  --case-in /perm/soa1/data/data_big \
  --verbose  --diff  \
  --nproma 32 \
  --ngpblks 2000 \
  --times 100 \
  --method $method > $method.txt 2>&1
done
```

In the launch script, it is possible to specify the routines called by the main (--method) and the input data (--case-in).

Optionally, it is also possible to specify the number of times the routine is called (default : 1), the nproma and number of blocks (default : input data).

A case-out directory can also optionally be specified, in which the data computed by the routine is stored. This can be used, for example, to generate output data which is bit-reproducible for a given set of compiler / compilation option.

Questions ?