natESM QUINCY Porting Guide

A Quick Guide to Porting QUINCY Subroutines (queued tasks) to GPU

Sergey Sukov[1]

[1] Forschungszentrum Jülich GmbH, Jülich, Germany

Contact: s.sukov@fz-juelich.de; info@nat-esm.de

Published on 07.01.2025 on https://www.nat-esm.de/services/accepted-sprints

1 Generating a reference data file

Using the CPU executable, create a simulation results file **resCPU.nc** (preferably a restart file that contains the values of all variables). In case of sequential (one after another) porting of tasks, it is advisable to compare the current data file with the previous one [**cdo diffn resCPU.nc prev_resCPU.nc**]. This way, we can verify that adding a new task to the queue results in noticeable deviations of variable values.

2 Building a local (within the portable subroutine) call tree and marking "frozen" code branches

At this stage, statements

```
#ifdef _OPENACC
CALL finish(routine, 'Code block not ported to GPU, yet. Stop.')
#endif
```

are added to inactive code branches.

3 Simplifying the task: dividing the subroutine into code blocks

Sometimes it makes sense to first divide logically complex subroutines into code blocks, which can then be ported independently of each other. Before and after such code blocks, CPU/GPU data synchronization directives are added:

```
!$ACC UPDATE HOST(a(:), b(:), c(:))
CALL SUBROUTINE_S1(a, b, c)
!$ACC UPDATE DEVICE(a(:), b(:), c(:))
#ifdef _OPENACC
a(:) = -1000.0; b(:) = -2000.0; c(:) = -3000.0
#endif

!$ACC UPDATE HOST(c(:), d(:))
CALL SUBROUTINE_S2(c, d)
!$ACC UPDATE DEVICE(c(:), d(:))
#ifdef _OPENACC
c(:) = -8000.0; d(:) = -9000.0
#endif
```

The last (optional) statements fill the data arrays on the CPU with "garbage". This helps to avoid situations where part of the calculations are mistakenly left on the host.

After subroutine decomposition into code blocks, you need to compile, build and run the GPU executable. The obtained simulation results (**restGPU.nc**) should match **restCPU.nc**.

4 Transforming the coding style to a form suitable for the GPU porting

Typically, this involves replacing calls to structure members with direct pointers and transforming operations on array slices into element-based loops. For example, the statement

```
data%arrayA(:) = data%arrayB(:)
```

will be expanded into a code block

```
ptrA => data%arrayA(:); ptrB => data%arrayB(:)
DO ic = 1, nc
      ptrA(ic) = ptrB(ic)
ENDDO
```

Here to check the correctness of the new code, you need to compile and run the CPU executable.

5 Offloading computations to GPU

To offload computations to the GPU, either a combination of the **!$ACC PARALLEL** + **!$ACC LOOP** directives or a combined **!$ACC PARALLEL LOOP** directive is used:

```
ptrA => data%arrayA(:); ptrB => data%arrayB(:)
!$ACC PARALLEL LOOP GANG VECTOR DEFAULT(PRESENT)
DO ic = 1, nc
      ptrA(ic) = ptrB(ic)
ENDDO
!$ACC END PARALLEL LOOP
```

It is assumed that the memory for the data arrays required for the computations has been allocated on the GPU and all variables have been updated to the correct values. Therefore, the **DEFAULT(PRESENT)** clause is mandatory in the **PARALLEL** and **PARALLEL LOOP** directives. If any data is missing on the device, the program will terminate with an error message:

```
0: FATAL ERROR: data in PRESENT clause was not found on device 1:
   name=l_growing_season_cl host:0x15496d3b93b0
0: file:/externals/jsbach/src/q_radiation/mo_q_rad_interface.f90
   update_time_average_q_radiation line:357
```

This usually means that the memory for local data arrays (that are used only within the subroutine in question) has not been allocated on the GPU. In this case, the **DATA** directive is added to allocate/deallocate temporary memory:

```
SUBROUTINE subroutineName(arg1, arg2)
...
REAL(8), ALLOCATABLE, DIMENSION(:,:) :: tmp
...
ALLOCATE(tmp(nc, ncanopy))
!$ACC DATA CREATE(tmp(:,:))
...
!$ACC PARALLEL LOOP GANG VECTOR COLLAPSE(2) DEFAULT(PRESENT)
DO icanopy = 1, ncanopy
DO ic = 1, nc
...
! Some offloaded computations using tmp
...
ENDDO
ENDDO
```

```
!ACC END PARALLEL LOOP
...
!$ACC END DATA
DEALLOCATE(tmp)
...
END SUBROUTINE subroutineName
```

6 Adding OpenACC directives to subroutine definitions

According to OpenACC rules, the **!$ACC ROUTINE** directive must be added to prototypes and/or definitions of procedures (subroutines and functions) that are called from GPU-ported compute regions. This directive tells the compiler to compile a procedure for the device and gives the execution context for calls to the procedure. In the case of ICON/JSBACH/QUINCY all procedures called from GPU compute regions are executed sequentially by the thread making the call and are specified using **!$ACC ROUTINE SEQ**. That is, the procedure is called in an element-based loop and processes only one data element:

```
!$ACC PARALLEL LOOP GANG VECTOR DEFAULT(PRESENT)
DO ic = 1, nc
CALL SUBROUTINE_S(a(ic), b(ic))
ENDDO
!$ACC END PARALLEL LOOP
...
...
SUBROUTINE SUBROUTINE_S(a, b)
!$ACC ROUTINE SEQ
...
END SUBROUTINE SUBROUTINE_S
```

If the subroutine's code and the offloaded loop within which it is called are in the same Fortran source file or module, the compiler may detect and specify the subroutine's type implicitly (although this is incorrect). Otherwise, the code compilation will be interrupted with an error message:

```
FC externals/jsbach/src/q_vegetation/mo_q_veg_update_pools.pp-jsb.o
NVFORTRAN-S-1061-Procedures called in a compute region must have acc routine
information - calc_mixing_ratio_c14c
(externals/jsbach/src/q_vegetation/mo_q_veg_update_pools.pp-jsb.f90: 742
```

7 Allocating memory and updating global variable values on the GPU

If global variable values are used within ported computational regions (or sequential subroutines/functions called from offloaded regions), memory for them must be explicitly allocated on the GPU for the program duration time. Otherwise, compilation will fail with the error message:

```
NVFORTRAN-W-1054-Module variables used in acc routine need to be in acc declare
create() - dwv2co2_turb
(externals/jsbach/src/q_assimilation/mo_q_assimi_process.pp-jsb.f90: 787)
```

OpenACC directives **!$ACC DECLARE** and **!$ACC UPDATE** are used to allocate memory for global variables and update their values. In the case where the initialization of the variable value occurs at the same point with its declaration, only the directive **!$ACC DECLARE** is used:

```
MODULE mo_q_assimi_constants
...
IMPLICIT NONE
PUBLIC
...
REAL(wp), SAVE :: Dwv2co2_air = Dwv / Dco2
...
```

```
!$ACC DECLARE COPYIN(Dwv2co2_air)
END MODULE mo_q_assimi_constants
```

If a special subroutine is called to set the value of a variable, then the **!$ACC DECLARE** directive is added at the variable declaration point, and the **!$ACC UPDATE** directive is used inside the data initialization subroutine:

```
MODULE mo_q_assimi_parameters
...
REAL(wp), SAVE :: discr_ps_a_C13 = def_parameters
...
!$ACC DECLARE CREATE(discr_ps_a_C13)
CONTAINS
...
SUBROUTINE init_q_assimi_parameters
...
discr_ps_a_C13 = 4.4_wp
...
!$ACC UPDATE DEVICE(discr_ps_a_C13)
END SUBROUTINE init_q_assimi_parameters
...
END MODULE mo_q_assimi_parameters
```

Note: In the last example, at the same time as the variable **discr_ps_a_C13** is declared, it is assigned the value **def_parameters**. Therefore, from a formal point of view, the **!$ACC DECLARE COPYIN(discr_ps_a_C13)** directive should be used here. However, the **def_parameters** parameter is declared as

```
REAL(wp), PARAMETER :: def_parameters = -9999.0_wp !< the default value of model
parameters (not constants) prior to their init
```

When the variable **discr_ps_a_C13** is declared, it is assigned some dummy value, which is then changed during the call to the **init_q_assimi_parameters** subroutine. That is, copying the initial value is not practical and the **!$ACC DECLARE CREATE(discr_ps_a_C13)** directive can be used.

8 Note A: Using the KERNELS directive

The **KERNELS** directive is effective (reasonable) to use in two cases:

- Porting simple array element value initialization statements.
- Preliminarily code porting when the main goal is to quickly offload computations to the GPU without regard to low performance.

Otherwise, there is a high probability that the compiler will detect non-existent data dependencies and force the code block to be executed sequentially. For such serialized code blocks, the compiler typically gives the following output:

```
334, Generating default present(ptr5d(i_part,i_elem,ics:ice,:,iblk),
                                ptr3d(ics:ice,:ptr3d$sd494+ptr5d$sd-1,iblk))
335, Complex loop carried dependence of ptr3d,ptr5d prevents parallelization
     Accelerator serial kernel generated
     Generating NVIDIA GPU code
335, !$acc loop seq
```

9 Note B: Asynchronous kernels execution

The ICON programming standard requires the developer to add the **ASYNC(1)** clause to loop parallelization and data transfer directives. Executing compute regions asynchronously using only one stream provides a small performance benefit by reducing the overhead of launching kernels. However, this option works correctly only if there are no dependencies between calculations on the CPU and GPU. Such implicit dependencies can arise, for example, when just

one CPU variable is used to select multiple offloaded code regions. Finding asynchronous execution bugs is a non-trivial and time-consuming task. Therefore, the **ASYNC** clause is not recommended to be used during the preliminarily porting stage, especially when the CPU source code is still under development.

10 Note C: CPU vs GPU calculation results comparison

Based on the experiments performed, it can be said that the calculation results (restart files) created by one executable should be completely identical regardless of the MPI process number and the block size (**nproma**). For files generated by different executables (Nvidia CPU/GPU executables or CPU executables build by different compilers), three types of variable value deviations can be considered as correct program execution:

- minor maxima of absolute (**cdo diffn ⇒ Max_Absdiff**) and relative (**cdo diffn ⇒ Max_Reldiff**) differences:

```
Max_Absdiff Max_Reldiff : Parameter name
  1.3399e-09 6.2395e-12 : a2l_drag_srf_box
```

- minor absolute difference and arbitrary relative difference:

```
Max_Absdiff Max_Reldiff : Parameter name
     5.2404e-16 0.99882 : spq_evaporation_veg
```

- minor relative difference for large absolute values of variables:

```
Max_Absdiff Max_Reldiff : Parameter name
   0.0013580 1.6476e-13 : spq_temp_srf_eff_4_pft09
```