Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Eidgenössisches Departement des Innern EDI
Bundesamt für Meteorologie und Klimatologie MeteoSchweiz

Accelerating climate models with GPUs using OpenACC

Xavier Lapillonne, Mikael Stellio, Daniel Hupp

Goals of the training

- Provide an introduction on key aspects and challenges related to GPU computing.

- Present minimal set of OpenACC directives to enable climate scientist to start porting a model to GPU/understand OpenACC directives in an existing code.

Content

- Introduction, GPU computing and directives
- Generating GPU kernels with OpenACC
- Data management with OpenACC
- Other OpenACC features
- Validation and performance

Material available at
https://github.com/C2SM-RCM/OpenACC_Training

Motivation: why using new hardware?

They are already there!

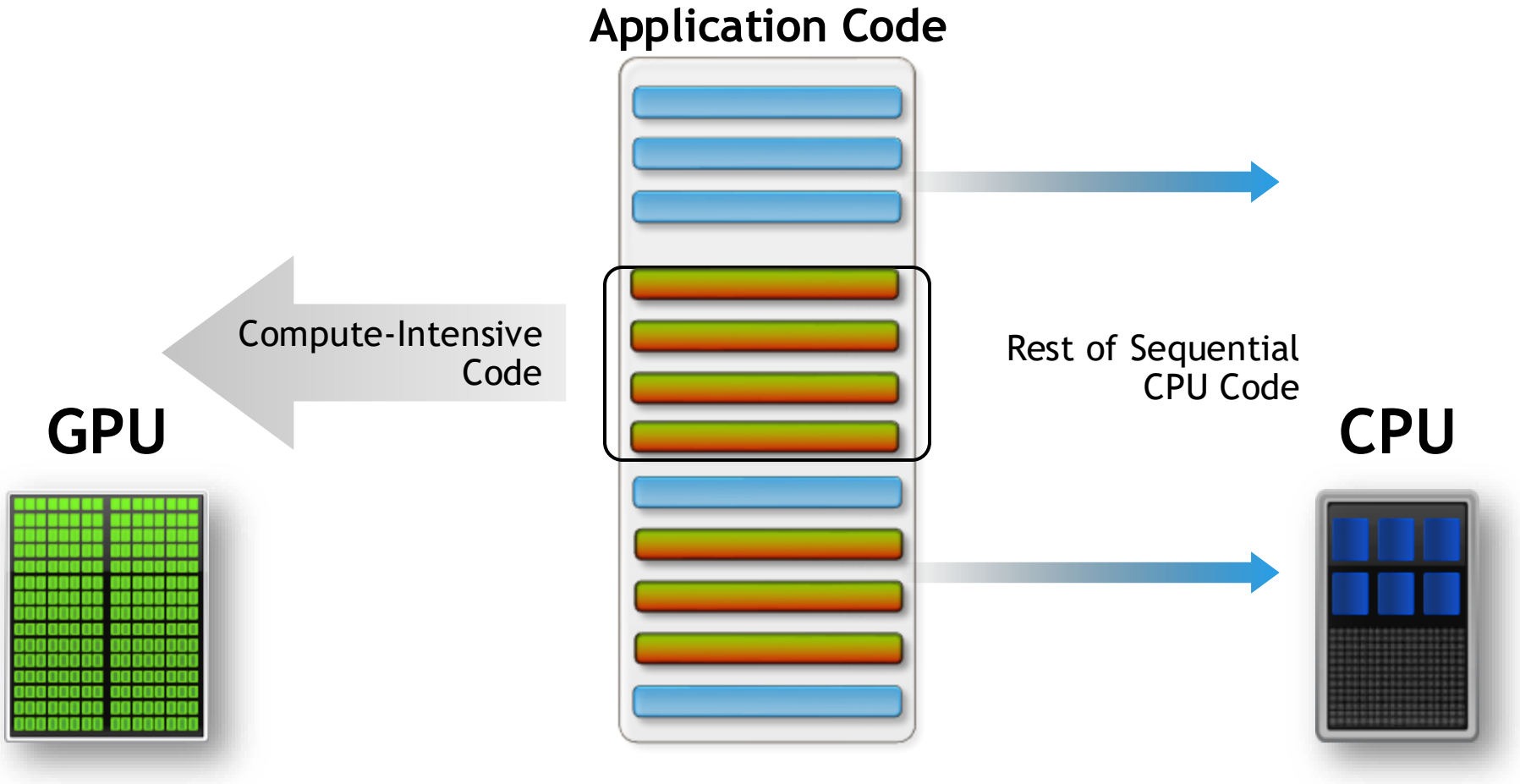
Most of the largest supercomputers are based on GPU accelerator

Easier to get allocation time on GPU based system when effectively using the GPUs

What is GPU computing

- GPU = Graphical Processing Units
- GPU computing: use of a GPU to offload (intensive) parts of an application, while the remainder is computed on the CPU

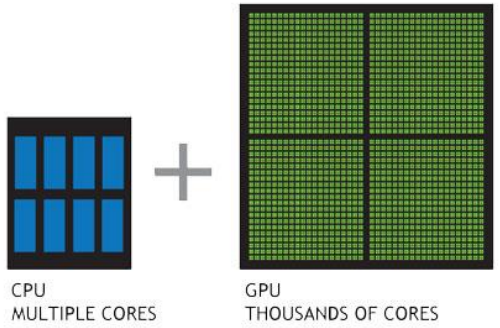
GPU computing: hybrid approach



G

GPU computing: key aspects

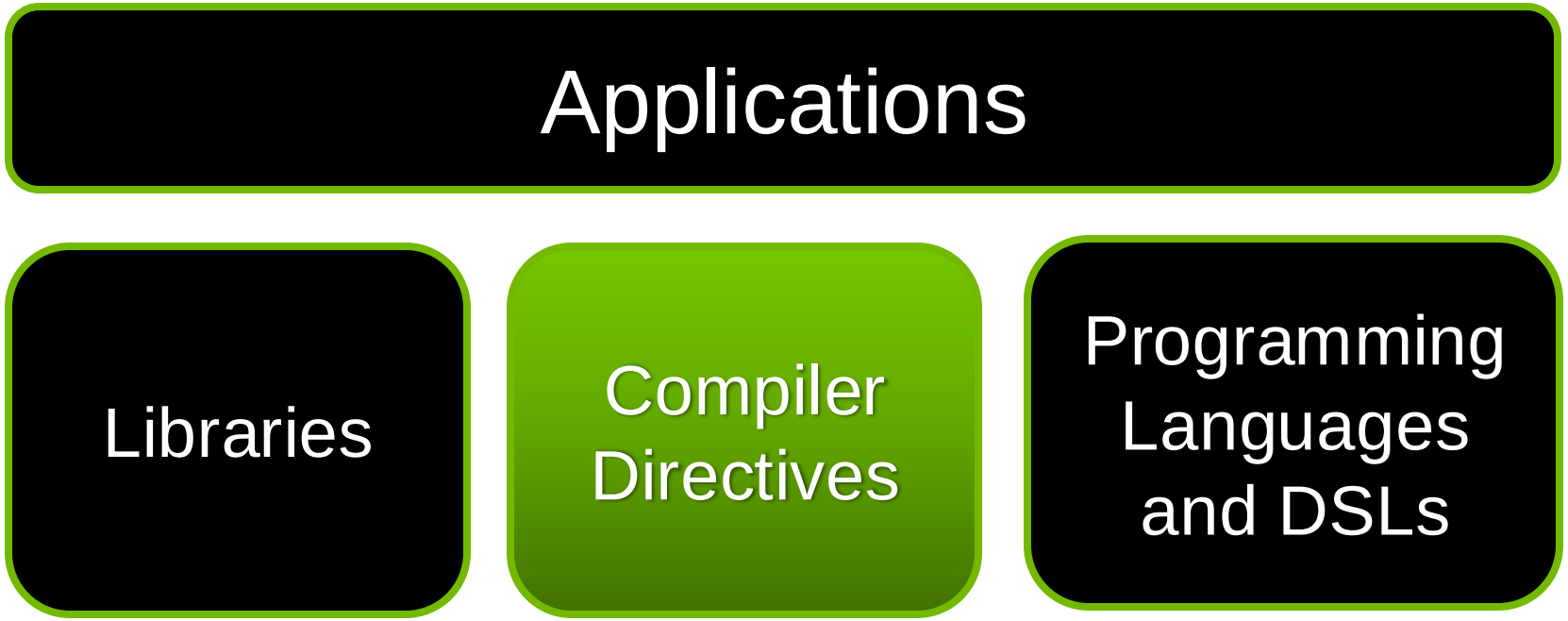
GPUs have thousands of compute cores: need to express fine-grain parallelism



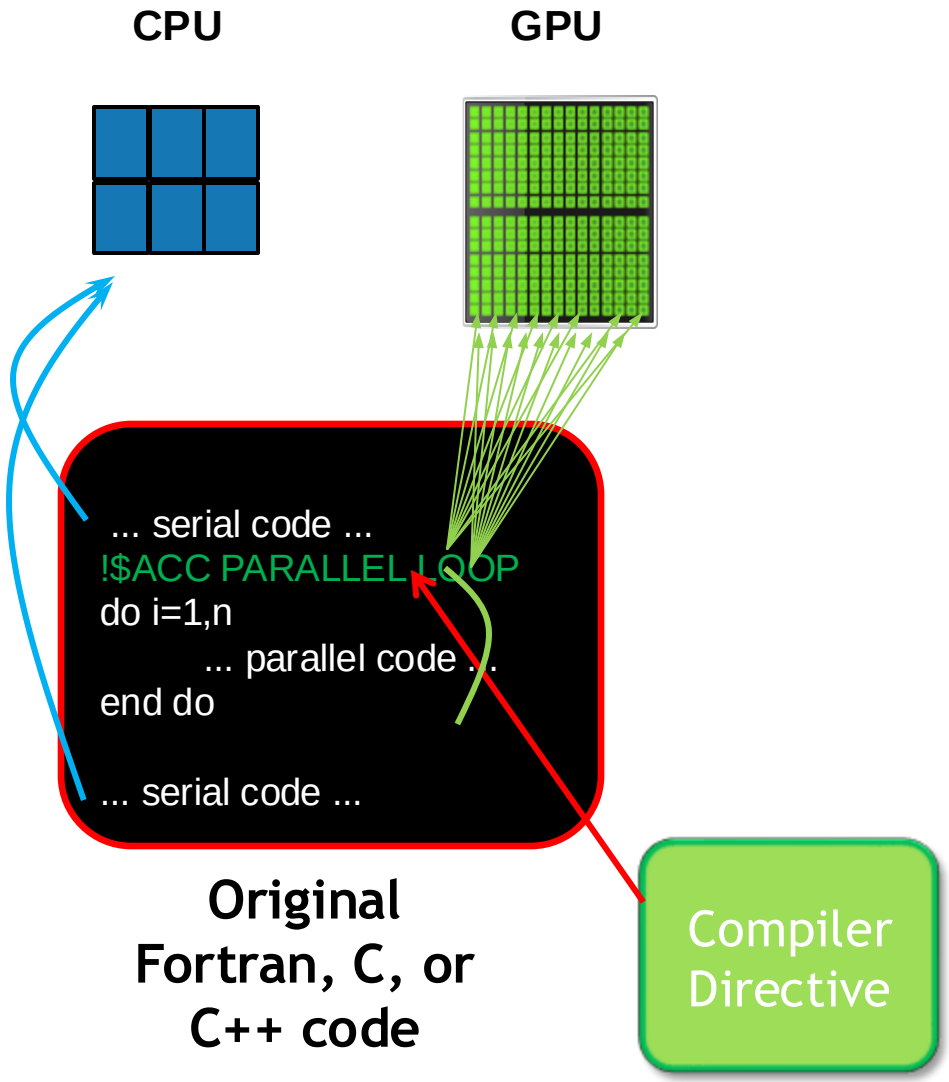
- GPU and CPU have (currently) separate physical memory
  - requires specific data management
  - data transfer may be a performance issue (slow transfer via PCI bus)

- As compared to typical multicore CPUs, GPUs have:
  - 5-10x higher peak performance (double precision)
  - 3-5x higher memory bandwidth

3 Ways to use GPUs



What are compiler directives?



- Compiler directives are comments or pragmas which can be inserted into existing code
- When a compiler directive is encountered the compiler/runtime will…

1. Generate parallel code for GPU
2. Allocate GPU memory and copy input data
3. Execute parallel code on GPU
4. Copy output data to CPU and deallocate GPU memory

- Directives are ignored on architecture without GPU

Content

- Introduction, GPU computing and directives
- Generating GPU kernels with OpenACC
- Data management with OpenACC
- Other OpenACC features
- Validation and performance

OpenACC

- OpenACC is a specification for high-level, compiler directives for expressing parallelism for accelerators in Fortran/C and C++.
- Aims to be performance portable to a wide range of accelerators.
- Multiple Vendors, Multiple Devices, One Specification
- Latest official release Official web site: http://www.openacc.org

OpenACC Directive Syntax

```
!$ACC DIRECTIVE [CLAUSE [,] CLAUSE] …]
```

...often paired with a matching end directive surrounding a structured code block:

```
!$ACC END DIRECTIVE
```

Fortran is capitalization agnostic.
In ICON capitalization was the style choice
(Style guide: https://gitlab.dkrz.de/icon/wiki/-/wikis/GPU-development/ICON-OpenAcc-style-and-implementation-guide)

Parallel and loop construct

!$ACC PARALLEL Starts parallel execution of the following section until the **!$ACC END PARALLEL**.

```
!$ACC PARALLEL
structured block
!$ACC END PARALLEL
```

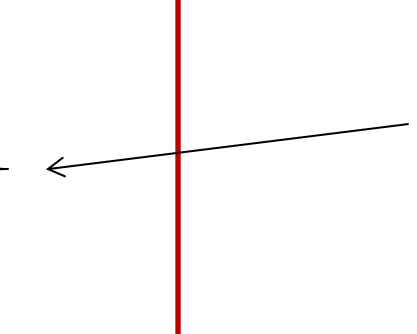
!$ACC LOOP [*clause*] applies to the immediately following loop, and describes the type of accelerator parallelism (vector, worker, gang) to use to execute the iterations of the loop. As a general rule all loop within a parallel region should have an !$ACC LOOP

!$ACC LOOP SEQ executes the loop sequentially

!$ACC LOOP with no clause the compiler is free to choose the type of parallelism

Example: SAXPY (y = a*x + y)

```
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

  !$ACC PARALLEL
  !$ACC LOOP
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$ACC END PARALLEL

end subroutine saxpy
```



This code region (kernel) will be executed in parallel on the GPU

Data management:

- Arrays x and y are automatically allocated and copied to the GPU
- y is automatically copied back to the CPU after the kernel execution
- scalars are automatically copied to the GPU

What does “parallel” mean?

Sequential (CPU)

```
Load x(1)
Load y(1)
Compute a*x(1) + y(1)
Store y(1)
Load x(2)
Load y(2)
Compute a*x(2) + y(2)
Store y(2)
Load x(3)
Load y(3)
Compute a*x(3) + y(3)
Store y(3)
Load x(4)
Load y(4)
Compute a*x(4) + y(4)
Store y(4)
…
```

Parallel (GPU)

```
Load x(1), x(2), x(3), …
Load y(1), y(2), y(3), …
Compute a*x(:) + y(:)
Store y(1), y(2), y(3)
```

xavier.lapillonne@meteoswiss.ch

Toy model porting example

Available at:
https://github.com/C2SM-RCM/OpenACC_Training/archive/master.zip

Mimics the structure of a simplified atmospheric model having only physical parametrizations and output.

The code is structured as follows:

main.f90	main driver, calls init, time loop, and output
m_config.f90	configuration information domain size, number of steps
m_fields.f90	global fields
m_io.f90	output routine
m_parametrization.f90	physical parametrizations doing the actual computation
m_physics.f90	driver for the physical paramtrization
m_setup.f90	code initialization and clean up
m_timing.f90	timing routines

Overview

main() in main.f90

```
CALL initialize()

!---------------------------------------
! time loop

WRITE(*,"(A)") "Start of time loop"

CALL start_timer(itimloop, "Time loop")

DO ntstep = 1, nstop

  ! call the physical parameterizations
  CALL physics()

  ! call outputs
  CALL write_output( ntstep )

END DO

CALL end_timer( itimloop )

WRITE(*,"(A)") "End of time loop"
```

physics() in:m_physics

```
! call a first physical parametrization
CALL saturation_adjustment(nx, ny, nz, t, qc, qv)

! call a second physical parametrization
CALL microphysics(nx, ny, nz, t, qc, qv)
```

saturation_adjustment(npx,npy,nlev,t,qc,qv) in m_parametrizations.f90

```
DO k = 1, nlev
  DO j = 1, npy
    DO i = 1, npx
      qv(i,j,k) = qv(i,j,k) + cs1*EXP(cs2*( t  ...
      qc(i,j,k) = cs4*qv(i,j,k)
    END DO
  END DO
END DO
```

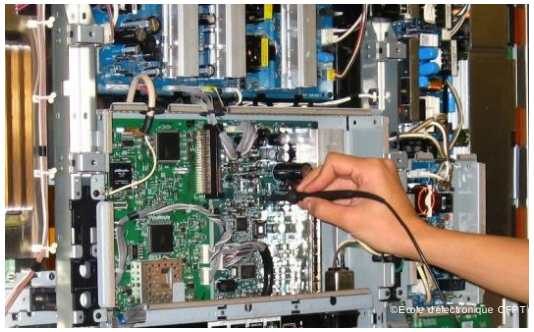
microphysics(npx,npy,nlev,t,qc,qv) in M_parametrizations.f90

```
DO k = 2, nlev
  DO j = 1, npy
    DO i = 1, npx
      qv(i, j, k) = qv(i,j,k-1) + cm1*  ...
      t(i, j, k)  = t(i, j, k)*( 1.0D0  ...
    END DO
  END DO
END DO
```

Building and running OpenACC code on Balfrin

```
# log on balfrin
ssh -Y balfrin

# change to scratch directory
cd $SCRATCH

# get code from ftp server
git clone https://github.com/C2SM-RCM/OpenACC_Training.git
cd OpenACC_Training

# setup environment
module use $USER_ENV_ROOT/modules
module load nvhpc
module load cuda

cd handsOn
make COMPILER=nvidia TARGET=gpu handsOn1
srun -n 1 --partition=debug --nodes=1 --time=0:05:00  ./handsOn1/handsOn1
```

Hands-on 1



In directory *handsOn1/*

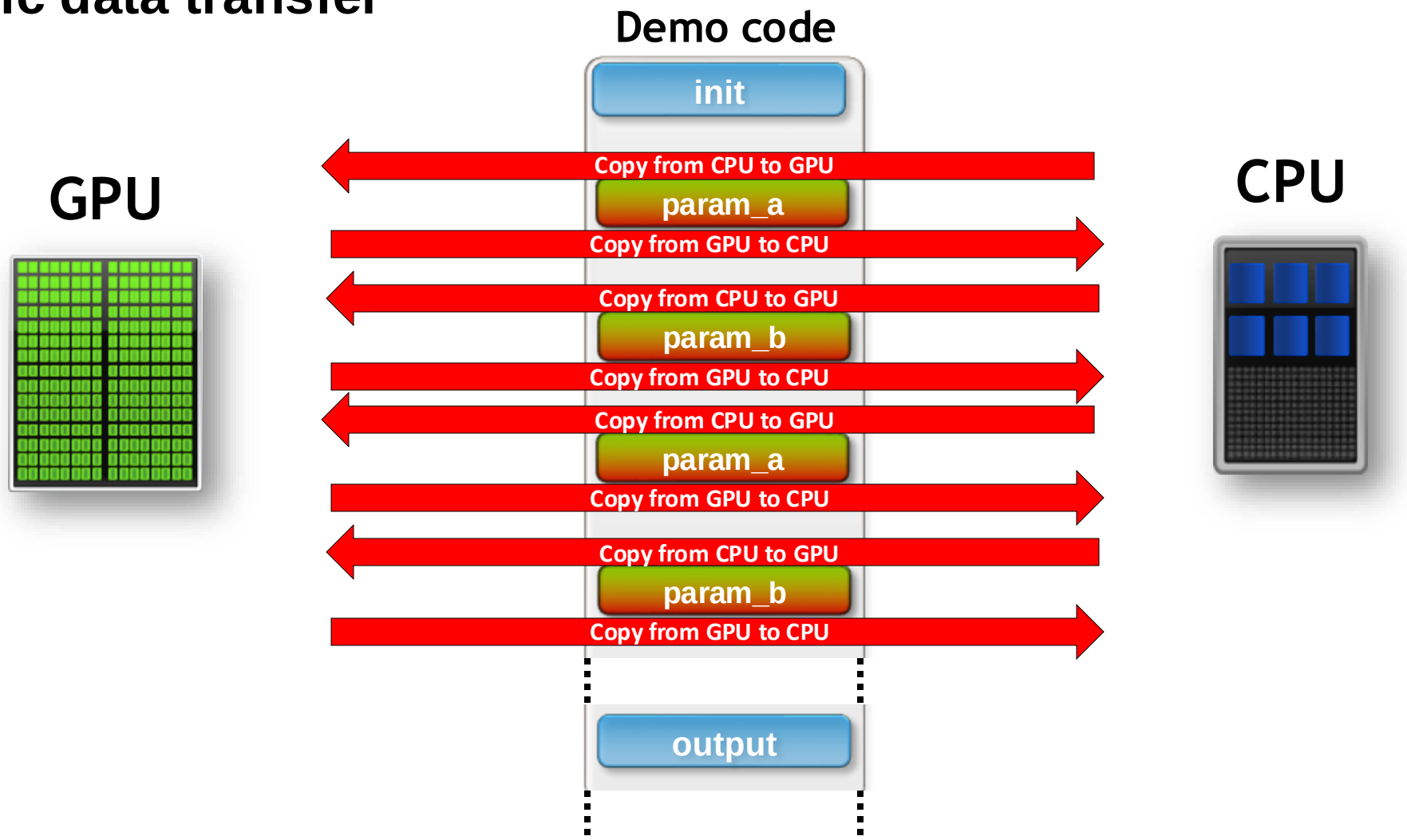
1. Compile and run *handsOn1* without modification to get the CPU reference.
2. Save standard output in *out_ref.txt.*
3. Port subroutines *saturation_adjustment* and *microphysics* to GPU using the OpenACC PARALLEL and LOOP constructs.
4. Run and check results against CPU.
5. (profile it and look at the profile)

How fast does your code run? Are you happy with the GPU acceleration?

Content

- Introduction, GPU computing and directives
- Generating GPU kernels with OpenACC
- Data management with OpenACC
- Other OpenACC features
- Validation and performance

Automatic data transfer



Data constructs

In order to avoid excessive data movement between the CPU and the GPU the programmer can manually allocate and transfer data on the GPU. Data can then be kept on the GPU and reused between different kernels

!$ACC ENTER DATA CREATE(var): allocates *var* on the GPU. Should be called after the allocation of *var* on the CPU.

!$ACC EXIT DATA DELETE(var): deallocates *var* on the GPU. Should be called before the deallocation of *var* on the CPU.

!$ACC UPDATE DEVICE(var): copies data from CPU to GPU.

!$ACC UPDATE HOST(var): copies data from GPU to CPU.

!$ACC DATA CREATE(var): allocates *var* on the GPU for a scope, *var* will be deallocated at the when the END DATA directive is met. Used for local subroutine arrays.

!$ACC END DATA: closes the scope of an open DATA region.

Present clause

!$ACC DATA PRESENT(var): tell the compiler that *var* is already allocated on the GPU. Valid until matching **!$ACC END DATA**.

The compiler will not do any automatic data transfer for this variable between the **!$ACC DATA PRESENT** and **!$ACC END DATA** statements.

```
!$ACC DATA PRESENT(var)
…
!$ACC END DATA
```

Example: SAXPY (y = a*x + y)

```
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i
  !$ACC DATA PRESENT(x,y)
  !$ACC PARALLEL DEFAULT(NONE)
  !$ACC LOOP
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$ACC END PARALLEL
  !$ACC END DATA
end subroutine saxpy
```

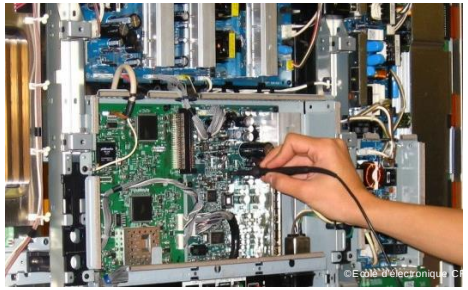
```
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i
  !$ACC PARALLEL DEFAULT(PRESENT)
  !$ACC LOOP
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
  !$ACC END PARALLEL
end subroutine saxpy
```

```
!allocate and copy to GPU
!$ACC ENTER DATA CREATE(arr1,arr2)
!$ACC UPDATE DEVICE(arr1,arr2)

!first call to saxpy
call saxpy(n, a, arr1, arr2)

!second call to saxpy
call saxpy(n, a, arr1, arr2)

!copy back to CPU and deallocate GPU memory
!$ACC UPDATE HOST(arr2)
!$ACC EXIT DATA DELETE(arr1,arr2)
```

Data is kept on the GPU between the two calls to *saxpy*

Scalars are still automatically copied

Hands-on 2



Minimize data transfers using data constructs. Keep initialization on the CPU.

1. Locate which array should be retained resident on the GPU.
2. Allocate/deallocate on the GPU.
3. Insert explicit data transfer.
4. Avoid automatic copies for parallel regions.
5. Check output against reference (*out_ref.txt*).
6. (profile it and look at the profile)

How fast does your code run? How much time was spent in copying data?

Part 2: explicit data placement 1/2

initialize() in m_setup.f90:

```
    ! print info
#ifdef _OPENACC
    WRITE(*,"(A)") "Running with OpenACC"
#else
    WRITE(*,"(A)") "Running without OpenACC"
#endif

    WRITE(*,"(A)") "Initialize"

    CALL init_timers()

    CALL start_timer( itiminit, "Initialization" )

    ! allocate memory
    ALLOCATE( t(nx,ny,nz), qv(nx,ny,nz) )

    ! allocate on the GPU
    !$acc enter data create(t,qv)

    ! initialize global fields
    DO k =1, nz
      DO j = 1, ny
        DO i = 1, nx
          t(i,j,k)  = 293.0D0 * (1.2D0 + 0.07D0 *
          qv(i,j,k) = 1.0D-6 * (1.1D0 + 0.13D0 * C
        END DO
      END DO
    END DO

    ! initialize fields on the GPU
    !$acc update device(t,qv)

#ifdef  OPENACC
```

Preprocessor macro defined when compiling with OpenACC

Allocation on GPU

Copy data from CPU to GPU

Part 2: 2/2

saturation_adjustment

```
!$acc data present(t,qv,qc)
! do the computation
!$acc parallel
!$acc loop
DO k = 1, nlev
  !$acc loop
  DO j = 1, npy
    !$acc loop
    DO i = 1, npx
      qv(i,j,k) = qv(i,j,k) +
```

Present directive ensures no GPU-CPU transfer

microphysics

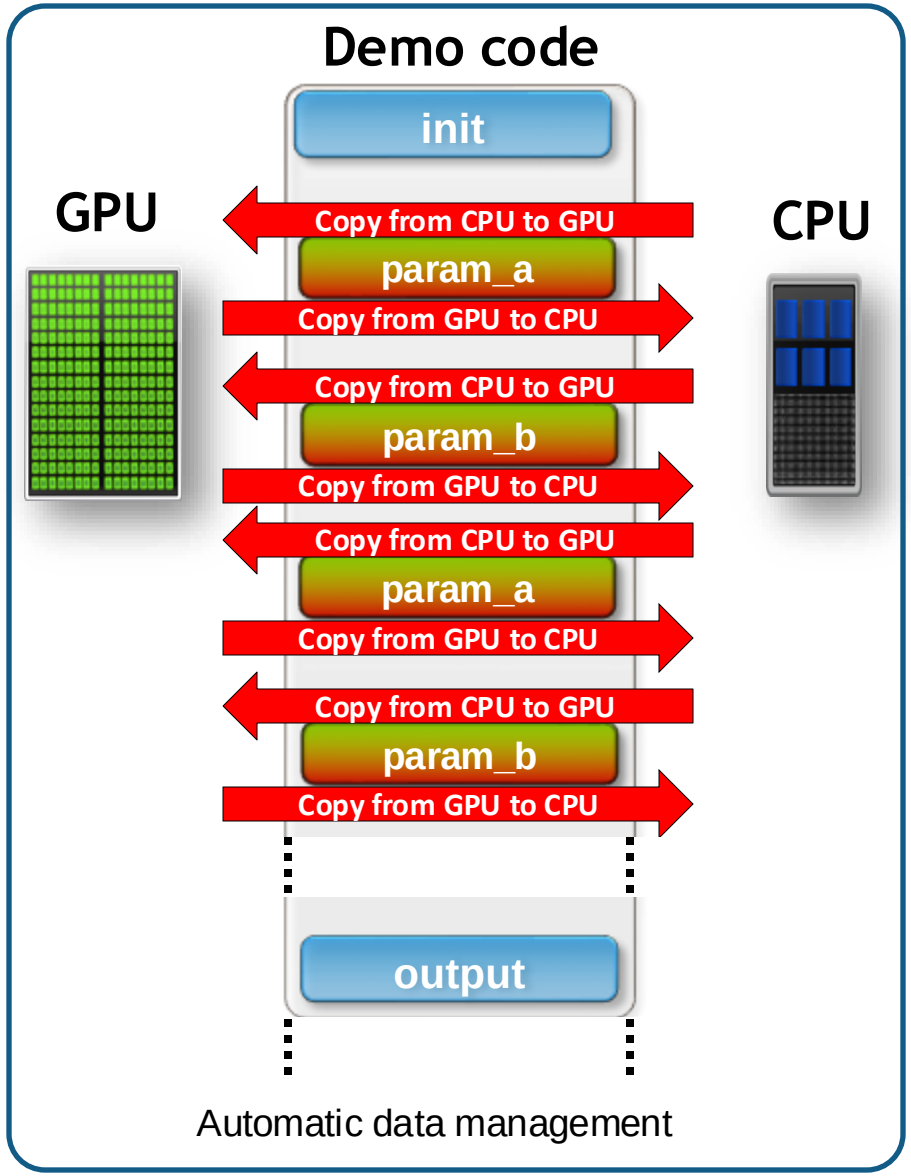
```
!$acc data present(t,qv,qc)
! do the computation
!$acc parallel
!$acc loop seq
DO k = 2, nlev
  !$acc loop
  DO j = 1, npy
    !$acc loop
    DO i = 1, npx
      qv(i, j, k) = qv(i,j,k-1) +
```

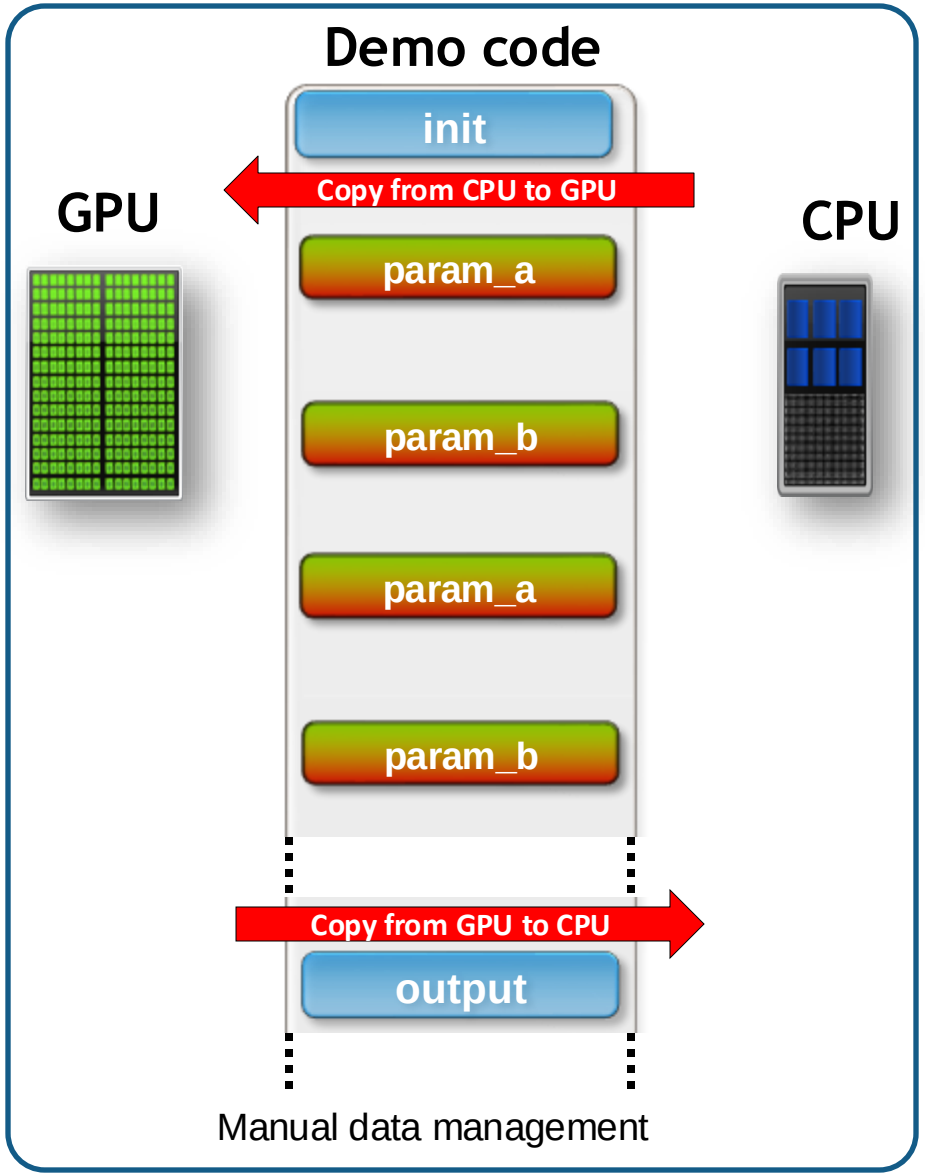
write_output in *m_io.f90*

```
! skip if this is not a output timestep
IF (MOD(ntstep, nout) /= 0) RETURN
!$acc update host(qv)
```

Copy data from GPU to CPU only for output steps

Automatic vs. manual data management





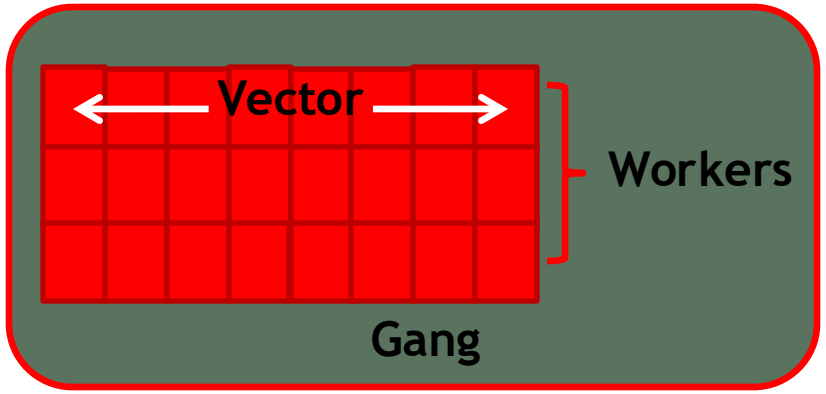
Content

- Introduction, GPU computing and directives
- Generating GPU kernels with OpenACC
- Data management with OpenACC
- Other OpenACC features
- Validation and performance

OpenACC: 3 Levels of Parallelism

Additional keyword to address how parallelism is mapped to hardware:

- *Vector* threads work in lockstep (SIMD/SIMT parallelism).
- *Workers* have 1 or more vectors.
- *Gangs* have 1 or more workers and share resources (such as cache, streaming multiprocessor, etc.).
- Multiple gangs work independently of each other.



For best performance the *vector* parallelism should be associated with the stride one index loop. This corresponds to the left most index in Fortran.

Example using GANG and VECTOR

```
subroutine saxpy2d(n, a, x, y)
  real :: x(n,n), y(n,n), a
  integer :: n, i
  !$ACC PARALLEL DEFAULT(PRESENT)
  !$ACC LOOP GANG
  do j=1,n
   !$ACC LOOP VECTOR
   do i=1,n
    y(i,j) = a*x(i,j)+y(i,j)
   end
  enddo
  !$ACC END PARALLEL
end subroutine saxpy2d
```

The **COLLAPSE** directive can be used to apply the same parallelism to tightly nested loops

```
subroutine saxpy2d(n, a, x, y)
  real :: x(n,n), y(n,n), a
  integer :: n, i
  !$ACC PARALLEL DEFAULT(PRESENT)
  !$ACC LOOP GANG VECTOR COLLAPSE(2)
  do j=1,n
    do i=1,n
    y(i,j) = a*x(i,j)+y(i,j)
   end
  enddo
  !$ACC END PARALLEL
end subroutine saxpy2d
```

Private variables

Scalar variables are by default *first-private* over the gangs not the threads.

This means that the same memory storage *could* be used by all vector threads.

Scalars which need to be thread private (i.e. scalar on the left side of the "=" within the vector loop) need to be made **PRIVATE** on the vector loop.

Scalar which are parameters for the considered kernel should be left with their default, i.e. no OpenACC statement. Note that loop variables are by default thread private and do not need a PRIVATE statement.

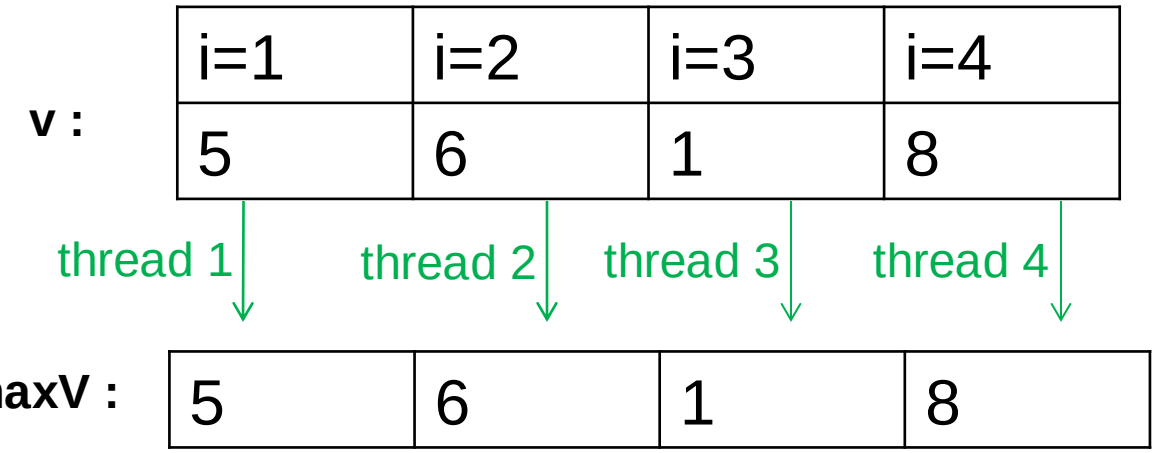
```
!$ACC PARALLEL DEFAULT(NONE) ASYNC(1)
!$ACC LOOP SEQ
DO ic = 1, i_count
  jj = idx_lst(ic, jb)
  !$ACC LOOP GANG VECTOR PRIVATE(xx)
  DO jc = i_startidx, i_endidx
    xx = A(jc) ** 2.
    p_prog%B(jj, jb) = xx
  ENDDO
ENDDO
!$ACC END PARALLEL
```

Other OpenACC features: reduction

```
!assumes v >= 0
maxV=0
do i=1,n
  maxV=max(maxV,v(i))
end do
```

If the *i*-loop is run in parallel all threads will write in *maxV* => **race condition**.

Ex:



maxV will have the value of the last thread writing at this location (i.e. any of 5,6,1,8)
=> Undefined behaviour, non-reproducible.

Reduction and atomic clause

REDUCTION(op:var): At the end of the loop, the values for each thread are combined using the specified reduction operator *op*, and the result stored in the original variable *var* at the end of the PARALLEL or KERNELS region. *Note: it should appear on each parallel loop (in case of nesting).*

```
!assumes v >= 0
maxV=0
!$ACC PARALLEL DEFAULT(PRESENT)
!$ACC LOOP GANG VECTOR REDUCTION(MAX:maxV)
do i=1,n
  maxV=max(maxV,v(i))
end do
!$ACC END PARALLEL
```

Valid Fortran operators are **+, *, MAX, MIN, IAND, IOR, IEOR, .AND., .OR., .EQV., .NEQV.**

See also: **ATOMIC** clause to prevent simultaneous access to a memory location.

OpenACC Routine Directive

The routine directive specifies that the compiler should generate a device copy of the function/subroutine in addition to the host copy.

```
!$ACC ROUTINE [GANG/WORKER/VECTOR/SEQ]
```

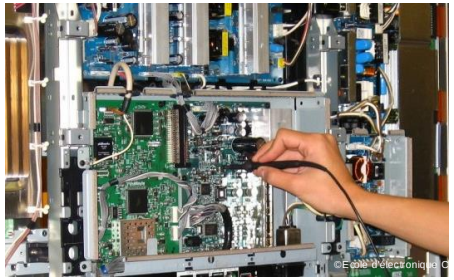
```
subroutine foo(v,i,j)
  !$ACC ROUTINE SEQ
  integer i,j
  real :: v

  v = 1.0/(i*j)
end subroutine

!$ACC PARALLEL DEFAULT(PRESENT)
!$ACC LOOP GANG
do j=1,n
   !$ACC LOOP VECTOR
   do i=1,n
    call foo(v(i,j),i,i)
  enddo
enddo
!$ACC END PARALLEL
```

The `routine` directive may appear in a fortran function or subroutine definition, or in an interface block.

Nested acc routines require the routine directive within each nested routine.

The save attribute is not supported.

Hands-on 3



Add explicit GANG and VECTOR clauses. Try what happens when only adding the GANG clause on the *i*-loop. Compute output results (sum of the elements of *a*) on the GPU.

```
sum_a=0
do k=1,nz
      do j=1,ny
         do i=1,nx
            sum_a=sum_a+a(i,j,k)
          end do
       end do
  end do
```

Bonus: adapt physics to call *saturation_adjustment* and *microphysics* in the same *i,j*-loops, and use the ACC ROUTINE directive.

Derived type

OpenACC supports derived types only since Version 2.6.

Derived types with allocatable or pointer components need special care.

Pointer/allocatable vs. statically allocated array

- Static components (scalar and fixed-size arrays) are copied with the derived types.
- Allocatable components have to be copied individually.
- If the components are already on the device, they need to be attached explicitly to the derived type.

Update only individual components of a derived type. An update directive applied on a derived type updates also the pointer references, which leads to invalid memory references.

```
type :: t_container
  integer           :: scal
  real, allocatable :: allo(:)
  real, pointer     :: poin(:)
  real              :: stat(500)
end type t_container

type(t_container) :: con
[…]
allocate(con% allo(500), con% poin(500))
```

```
!$ACC ENTER DATA COPYIN(con)
!$ACC ENTER DATA COPYIN(con% allo, con% poin)
[…]
!$ACC EXIT DATA COPYOUT(con% allo, con% poin)
!$ACC EXIT DATA COPYOUT(con)
```

p_derived_type_0_enter_A.f90

```
!$ACC ENTER DATA COPYIN(con% allo, con% poin)
!$ACC ENTER DATA COPYIN(con)
!$ACC ENTER DATA ATTACH(con% allo, con% poin)
[…]
!$ACC EXIT DATA COPYOUT(con% allo, con% poin)
!$ACC EXIT DATA COPYOUT(con)
```

p_derived_type_0_enter_B.f90

Slide from Marek Jacob, DWD, Intro part 1

Derived type

- Safest way is to create base derived type and components on device first and update needed components individually.
- If a variable is already present on device a COPY/COPYIN/COPYOUT does not update device/host.
- By updating individual components we can restrict memory transfer to those components, which are needed on device/host.
- The attach counter keeps track whether the pointers are attached or not.

```
type :: t_container
  integer           :: scal
  real, allocatable :: allo(:)
  real, pointer     :: poin(:)
  real              :: stat(500)
end type t_container

type(t_container) :: con
[…]
allocate(con% allo(500), con% poin(500))
```

```
!$ACC ENTER DATA CREATE(con)
!$ACC ENTER DATA CREATE(con% allo, con% poin)

!$ACC UPDATE DEVICE(con% scal, con% allo)&
!$ACC   DEVICE(con% poin, con% stat)

[…]

!$ACC UPDATE SELF(con% scal, con% allo) &
!$ACC   SELF(con% poin, con% stat)

!$ACC EXIT DATA DELETE(con% allo, con% poin)
!$ACC EXIT DATA DELETE(con)
```

p_derived_type_0_enter_C.f90

Slide from Marek Jacob, DWD, Intro part 1

Kernels vs parallel construct

- ACC KERNELS are similar to PARALLEL regions.
- A KERNELS region gives the compiler maximum freedom of the parallel implementation.
- Might be better or much worse than manual parallelization, may be compiler dependent.
- Use of kernels is not recommended.
- Implicit Fortran loops need to be replaced by explicit loops in a parallel region.

```
!$ACC KERNELS
do i = 1, n
  a(i) = 2.0 * b(i)
end do
!$ACC END KERNELS
```

```
!$ACC KERNELS
a(:) = 0.0
!$ACC END KERNELS

!$ACC PARALLEL LOOP
do i = 1, n
  a(i) = 0.0
end do
!$ACC END PARALLEL
```

Slide from Marek Jacob, DWD, Intro part 1

Asynchronous Operations

- GPU Kernels run asynchronous to CPU
- Schedule different tasks in parallel
- Synchronize tasks with queue number
- Accelerate acc kernel scheduling
- Optimize host device data transfer by overlapping with other operations
- Example: Kernels 1 and 3 may run parallel to kernel 2

- in ICON: everything should be `ASYNC(1)`, except regions with cuda-graphs
- `!$ACC WAIT` is added when needed

```
! Kernel 1
!$ACC PARALLEL LOOP ASYNC(1)
...
!$ACC END PARALLEL LOOP

! Kernel 2
!$ACC PARALLEL LOOP ASYNC(2)
...
!$ACC END PARALLEL LOOP

! Kernel 3
!$ACC PARALLEL LOOP ASYNC(3)
...
!$ACC END PARALLEL LOOP

!$ACC WAIT(1,2) ! Explicit queues
!$ACC WAIT      ! All queues
```

Slide adapted from Marek Jacob, DWD, Intro part 1

Asynchronous Operations

Special places where we always need to put **!$ACC WAIT**:

- Before **!$ACC UPDATE**
- Before ending a **CREATE** data region, if it is at the end of a subroutine.
  - (DATA region has no ASYNC option before OpenACC 3.2)
- Before ending a **COPY[IN|OUT]** data region, if it is at the end of a subroutine.
- After a compute region with **REDUCTION** before the reduce variable is used on device or host.

ENTER DATA/EXIT DATA may have ASYNC

```
allocate(a(N))
!$ACC DATA CREATE(a)

! Kernel 1
!$ACC PARALLEL LOOP ASYNC(1)
...
!$ACC END PARALLEL LOOP
!$ACC WAIT(1)
!$ACC END DATA
deallocate(a(N))
```

```
allocate(a(N))
!$ACC ENTER DATA CREATE(a) ASYNC(1)

! Kernel 1
!$ACC PARALLEL LOOP ASYNC(1)
...
!$ACC END PARALLEL LOOP
!$ACC EXIT DATA DELETE(a) ASYNC(1)
!$ACC WAIT(1)
deallocate(a(N))
```


Slide adapted from Marek Jacob, DWD, Intro part 1

Content

- Introduction, GPU computing and directives
- Generating GPU kernels with OpenACC
- Data management with OpenACC
- Other OpenACC features
- Validation and performance

Validation

- Both CPU and GPU implement IEEE-754 compliant floating point operations
  a+b or a*b will give the same results = bitwise identical

However:

- Transcendental functions are not defined, compiler dependent
- Compiler may (or not) use so called: fused add multipy

⇒ In general CPU and GPU results are not bitwise identical

Note: CPU codes compiled with two compilers are in general not bitwise identical

- Validations: requires to define acceptable thresholds, e.g. by perturbing reference CPU results to the last bit

Perfomance

Great care is required when evaluating performance, what is being compared . What is the expected performance?

We often use socket to socket comparison, expectation based on hardware characteristic, compare multicore CPU vs GPU

- Use profiler, nsys, … : memory bandwidth, Flops …