Status ICON NWP GPU port

D. Hupp[1], J. Jucker[2], R. Dietlicher[1], X. Lapillonne[1], F. Gessler[1],
V. Cherkas[1], C. Osuna[1], M. Zhigun[1], R. Svoboda[1], C. L. Lo[1]
[1]MeteoSwiss, [2]C2SM

September 7, 2021

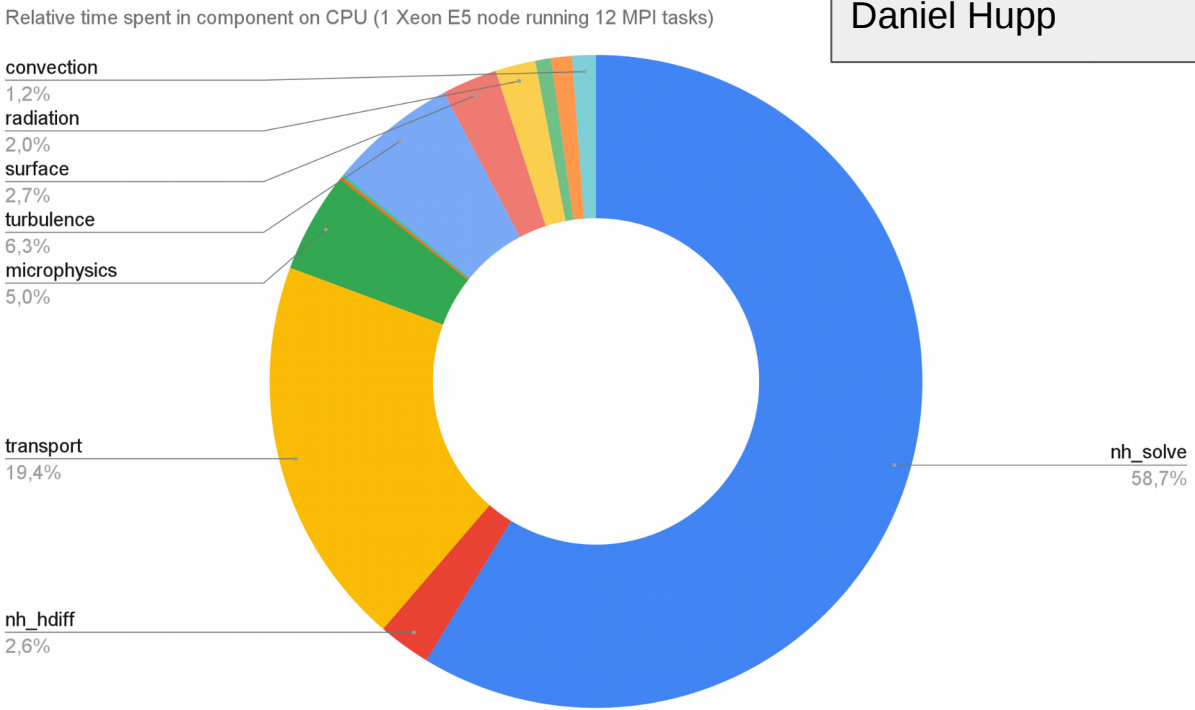
Porting strategy

- Avoid GPU-CPU transfer: all components of the time loop need to be ported to GPU
  - Exception: Data assimilation runs on CPU (see Fabian Gesslers Talk)
- Design : Main parallelization along nproma (not at the block level like OpenMP for CPU)
  - Use large nproma when running on GPU (ideally 1 single block per GPU)
  - Compatible with COSMO parameterizations already ported to GPU
- Test...
  - Individual components with Serialbox and PCAST (Nvidia compiler tool) during development
  - ICON output on buildbot under tolerance (CPU/GPU not bit-identical)

Status of the OpenACC port

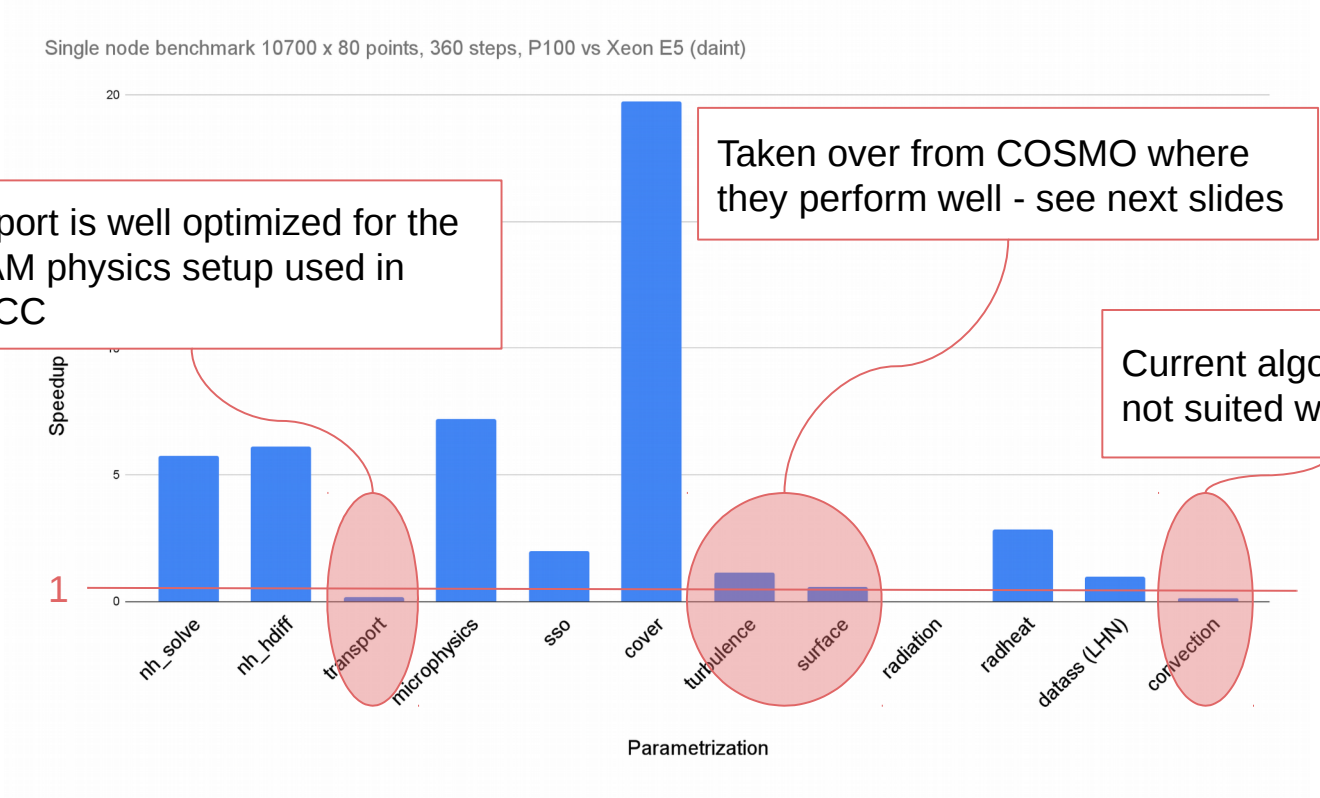
Not the operational configuration yet

More on this later by Daniel Hupp



	ported	optimized
nh_solve		
nh_hdiff		
transport		
convection		
LHN		
radiation		
radheat		
microphysics		
cover		
turbulence		
surface		
sso		

Status of OpenACC performance



Surface

Optimizations applied so far

- Use routines from mo_index_list to generate index lists
- Initialize arrays on GPU to prevent expensive copyin to GPU

Open questions

- Kernel launch time bigger than effective runtime on GPU
- Unnecessary (?) HtoD memory copies use some resources

Source: Jonas Jucker

Turbulence

Optimizations applied so far

- Apply async clause as much as possible
- Collapse purely nested k- and i-loops (identical to COSMO)
- Fuse i-loops inside k-loop (in COSMO done by CLAW)
- Replace !$acc data present with default(present) at kernel-level (identical to COSMO)

Open questions

- Almost identical code as for COSMO, but performance still very bad!
- Unnecessary (?) HtoD memory copies use some resources
- Why profile for ICON so interrupted

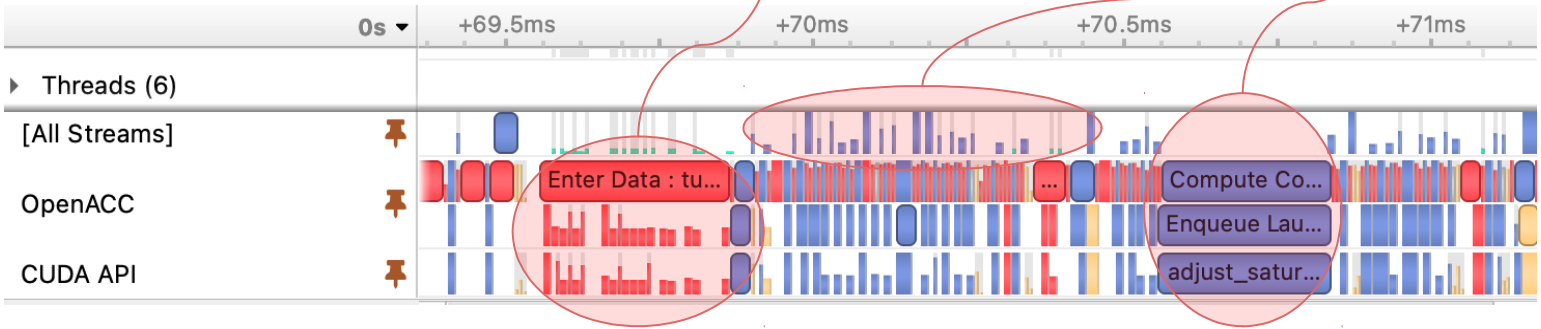
Source: Jonas Jucker

Turbulence

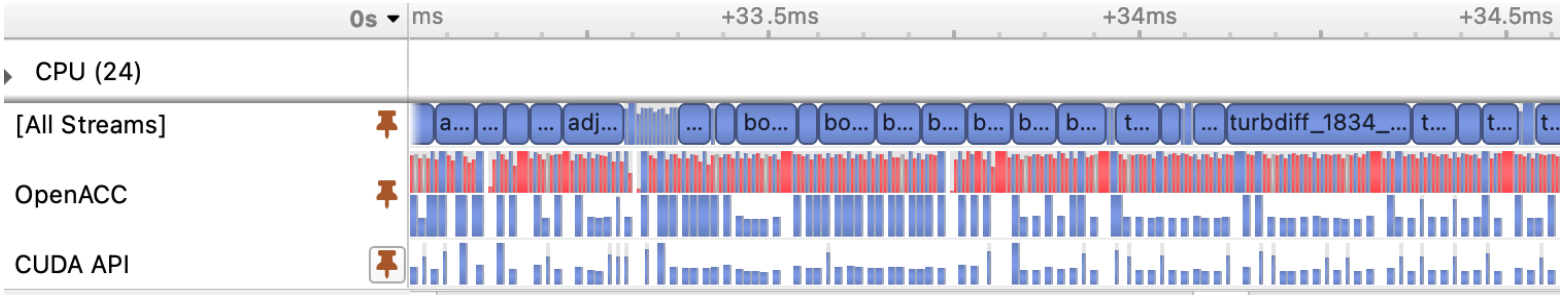
Fortran Automatic arrays should only be allocated once

Significant launch overhead

ICON profile



COSMO profile



Source: Jonas Jucker

ECRAD:
Porting to GPU with OpenACC

R. Dietlicher, X. Lapillonne, F. Gessler,
V. Cherkas, D. Hupp, C. Osuna, M. Zhigun
MeteoSwiss

September 7, 2021

Porting process

- **Stage 0**: Evaluate and determine porting strategy:
  - Study different data layouts, loop nesting and OpenACC techniques.
  - Determine parallel regions
- **Stage 1**: Refactor automatic array allocations and increase dimension of arrays that will be processed in parallel
- **Stage 2**: Introduce OpenACC to the code, with minimal refactoring impact
- **Stage 3:** Optimize performance

Main components in ecRad

ecRad component	Stage 0	Stage 1	Stage 2	Stage 3
gas_optics			Work in progress	
cloud_optics			Work in progress	
add_aerosol_optics				
solver_mcica_lw				
solver_mcica_sw				

Stage 0 → Stage 1 → Stage 2 → Stage 3
Code refactoring

- Increase dimension of variables that will be processed in parallel
- Automatic array allocation inside OpenACC parallel regions are spoiling the performance, affected subroutines:
  - `cloud_generator`
  - `generate_column_exp_ran`
  - `adding_ica_sw`
  - `fast_adding_ica_lw`
  - `INITIALIZE_RANDOM_NUMBERS`

⇒ Encapsulated in git branches :

- https://gitlab.dkrz.de/dwd-sw/libecrad/-/tree/pre_acc_port (ecRad standalone)
- https://gitlab.dkrz.de/dwd-sw/libecrad/-/tree/port_aerosol_optics_submodule (icon-submodule)

Stage 0 → Stage 1 → Stage 2 → Stage 3

Refactoring example 1

```
subroutine solver_mcica_lw(…)
  …
  real(jprb), dimension(ng) :: od_total
  …
  do jcol = istartcol,iendcol
    …
    do jg = 1,ng
      …
      od_total(jg) = ...
      …
    end do
    …
  end do
end subroutine solver_mcica_lw
```

@master

```
subroutine solver_mcica_lw(…)
  …
  real(jprb), dimension(ng,istartcol:iendcol) :: od_total
  …
  do jcol = istartcol,iendcol
    …
    do jg = 1,ng
      …
      od_total(jg,jcol) = ...
      …
    end do
    …
  end do
end subroutine solver_mcica_lw
```

@pre_acc

Stage 0 → Stage 1 → Stage 2 → Stage 3

Refactoring example 2

```
subroutine solver_mcica_lw(…)
  …
  do jcol = istartcol,iendcol
    …
    call cloud_generator(…)
    …
  end do
end subroutine solver_mcica_lw


subroutine cloud_generator(…)
  …
 real(jprb) :: rand_top(ng)
  …
end subroutine cloud_generator
```

@master

```
subroutine solver_mcica_lw(…)
  …
  real(jprb), dimension(ng, istartcol:iendcol) :: tmp_work_ng


  …
  do jcol = jstartcol,jendcol
    …
    call cloud_generator(…, &
       & rand_top=tmp_work_ng(:,jcol),…)
    …
  end do
end subroutine solver_mcica_lw


subroutine cloud_generator(…, rand_top, …)
  …
  real(jprb), intent(out) :: rand_top(ng)
  …
end subroutine cloud_generator
```

@pre_acc

Code refactoring

- Add optional flag `use_acc` subroutines in call tree
  - Add warnings to subroutines that are about to be ported
  - Add error messages and terminating run for subroutines that are not covered by our experiments.
- Manual inlinling subroutines
- Breakup derived types
- Splitting loop to encapsulate `cloud_generator` subroutine
- Swapping if-statements with loop-statements
- Precomputing random numbers
- Replacing return statements with else condition
- Writing loops explicitly
- ⇒ Encapsulated in git branches :
  - https://gitlab.dkrz.de/dwd-sw/libecrad/-/tree/acc_port (as ecRad standalone)
  - https://gitlab.dkrz.de/dwd-sw/libecrad/-/tree/acc_port_submodule (as icon-submodule)

Refactoring example 1

```
subroutine some_subroutine(…, use_acc)
  …
  logical, intent(in), optional :: use_acc
  …
  logical :: lacc
  …
  if(present(use_acc)) then
    lacc = use_acc
  else
    lacc = .FALSE.
  end if
  …
end subroutine some_subroutine
```

```
#ifdef _OPENACC
     write(nulerr,'(a)') '*** Error: some_subroutine not
ported to GPU'
     call radiation_abort()
   endif
#endif
```

```
#ifdef _OPENACC
     write(nulerr,'(a)') '*** WARNING: some_subroutine port me'
   endif
#endif
```

- 27/07/21: 67 times *port me* in 54 files
- 01/09/21: 44 times *port me* in 41 file

⇒ 0.62 subroutines per day ⇒70 days left

Stage 0 → Stage 1 → Stage 2 → Stage 3

Code refactoring

- Replace random number generator in solver_mcica
  → will require scientific revalidation
- Potentially merge jg-loops over function calls
- …

Constraints:

- Balancing with CPU and NEC performance
- Balancing with readability and maintainability

Timings (ecRad standalone)

ecRad component	CPU @master [ms]	CPU Stage 1 @pre_acc [ms]	CPU Stage 2 @acc [ms]	GPU Stage 2 @acc [ms]
gas_optics	20.0	20.0	20.9	470.5
cloud_optics	0.598	0.688	0.791	21.5
add_aerosol_optics	0.587	0.748	0.0278	1.31
solver_mcica_lw	12.9	17.8	18.0	22.8
solver_mcica_sw	20.6	20.5	21.0	14.9

Set-up

- nblocksize: 1 for CPU ncol for GPU
- Ncol: 360
- pgf90 20.4-0 -O3 -Mipa=fast

Hardware-node:

- 8 x NVIDIA® V100 GPUs (with 32 GB HBM memory each)
- 2 x Intel® Xeon® Skylake 6134 CPUs (8-core, 3.2 GHz)
- 384 GB DDR4 memory

Goal: evaluate feasibility of a single source code (ECWFM github) that performs on CPU, GPU and NEC

Study different data layouts, loop nesting and OpenACC techniques.

Disclaimer: work in progress, slides only contain mostly code, small experiments (and a summary)

Sketch of loop operations in ecRad, part 2

- First part of any all-sky solver
  - do **column, level**
    - if **cloud_present(column, level)** *Clouds are rare…*
      - do **spectrum** Add cloud optics to gas+aerosol and compute layer reflectance & transmittance
    - else
      - do **spectrum** Copy over clear-sky reflectance & transmittance
- Tripleclouds solver
  - do **column** if sun above horizon
    - do **level** *Longwave optimizations may only loop from cloud top*
      - if **cloud_present(column, level)**
        - do **spectrum** Propagate fluxes in clear and cloudy regions
      - else
        - do **spectrum** Propagate fluxes in clear regions
- McICA solver
  - do **column** if sun above horizon
    - do **level** } *Recently swapped to*
      - do **spectrum** } *this order for DWD*
        - **scaling(spectrum,level)** depends on **cloud_fraction(column,level)** and **scaling(spectrum,level-1)**
    - do **level** *One level depends on the previous*
      - do **spectrum** Radiative transfer

@Robin Hogan

Sketch of loop operations in ecRad, part 2

- First part of any all-sky solver
  - do **column, level**
    - if **cloud_present(column, level)** *Clouds are rare…*
      - do **spectrum** Add cloud optics to gas+aerosol and compute layer reflectance & transmittance
    - else
      - do **spectrum** Copy over clear-sky reflectance & transmittance
- Tripleclouds solver
  - do **column** if sun above horizon
    - do **level** *Longwave optimizations may only loop from cloud top*
      - if **cloud_present(column, level)**
        - do **spectrum** Propagate fluxes in clear and cloudy regions
      - else
        - do **spectrum** Propagate fluxes in clear regions

Focus on the LW solver of McICA

- McICA solver
  - do **column** if sun above horizon
    - do **level** } *Recently swapped to*
      - do **spectrum** } *this order for DWD*
        - **scaling(spectrum,level)** depends on **cloud_fraction(column,level)** and **scaling(spectrum,level-1)**
    - do **level** *One level depends on the previous*
      - do **spectrum** Radiative transfer

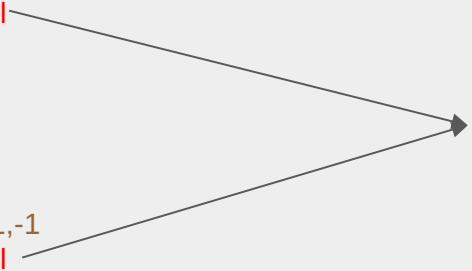
Loop structure in master of ECRAD

LW solver of McICA

```
! each ecrad component contains a single top level jcol loop
subroutine solver_lw(...)
  do jcol=jstart,jend
    ! jcol loop is not perfectly nested,
    !... a single jcol loop contains many jlev,jg loops
    do jlev=1,nlev
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```

Issues for performance: GPU

```
! each ecrad component contains a single top level jcol loop
subroutine solver_lw(...)
  do jcol=jstart,jend
    ! jcol loop is not perfectly nested,
    !... a single jcol loop contains many jlev,jg loops
    do jlev=1,nlev
      !$acc parallel
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      !$acc parallel
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```



Not enough parallelism on the wavelength dimension

Issues for performance: GPU

```
! each ecrad component contains a single top level jcol loop
subroutine solver_lw(...)
  !$acc parallel
  do jcol=jstart,jend
    ! jcol loop is not perfectly nested,
    !... a single jcol loop contains many jlev,jg loops
    do jlev=1,nlev
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```

Memory footprint of wavelength dimension, ng, does not fit into GPU memory

Issues for performance: NEC

```
! each ecrad component contains a single top level jcol loop
subroutine solver_lw(...)
  do jcol=jstart,jend
    ! jcol loop is not perfectly nested,
    !... a single jcol loop contains many jlev,jg loops
    do jlev=1,nlev
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```

Vector length not large enough, specially if reduced to ~40

Variant 1: ncol - ng collapse

```
subroutine solver_lw(...)
  do jcol=jstart,jend
    do jlev=1,nlev
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```

@master

```
subroutine solver_lw(...)
    do jlev=1,nlev
      !$acc parallel loop collapse(2)
      do jcol=jstart,jend
        do jg = 1, ng
          !...
        enddo
      enddo
      !...
      do jlev=nlev,1,-1
        !$acc parallel loop collapse(2)
        do jcol=jstart,jend
          do jg = 1, ng
            !...
          enddo
        enddo
      enddo
end subroutine
```

@Variant 1

Advantages:

Easy refactoring

Allows to gain parallelism using collapse (GPU) and swap jcol <-> jg (NEC)

Variant 1: ncol - ng collapse

```
subroutine solver_lw(...)
  do jcol=jstart,jend
   do jlev=1,nlev
    do jg = 1, ng
     !...
    enddo
   enddo
   !...
   do jlev=nlev,1,-1
    do jg = 1, ng
     !...
    enddo
   enddo
  enddo
end subroutine
```

@master

```
subroutine solver_lw(...)
   do jlev=1,nlev
    !$acc parallel loop collapse(2)
    do jcol=jstart,jend
     do jg = 1, ng
      !...
     enddo
    enddo
   !...
   do jlev=nlev,1,-1
    !$acc parallel loop collapse(2)
    do jcol=jstart,jend
     do jg = 1, ng
      !...
     enddo
    enddo
  enddo
end subroutine
```

@Variant 1

Issues:

Vertical field accesses planck_hl(jcol,jg,jlev+1) triggers page faulting (CPU)
Less cache friendly since jcol is added inside the jlev solver
Observed impact in CPU performance ~2x (slower)

Variant 2: ncol-ng swap

```
subroutine solver_lw(...)
 do jcol=jstart,jend
  do jlev=1,nlev
   do jg = 1, ng
    !...
   enddo
  enddo
  !...
  do jlev=nlev,1,-1
   do jg = 1, ng
    !...
   enddo
  enddo
 enddo
end subroutine
```

@master

```
subroutine solver_lw(...)
 do jg = 1, ng
  do jlev=1,nlev
   do jcol=jstart,jend
    !...
   enddo
  enddo
  !...
  do jlev=nlev,1,-1
   do jcol=jstart,jend
    !...
   enddo
  enddo
 enddo
end subroutine
```

@Variant 2

Advantages:

Size of jcol is adjustable for best vectorization (NEC & GPU) & parallelization

On CPU, better or at par performance vs master

Issues:

Without inlining, introduces an extra overhead for function calls

In order to eliminate the memory footprint issue, do jg needs to to moved out to the driver -> doable but not trivial refactoring

Variant 2: ncol-ng swap (GPU)

```
subroutine solver_lw(...)
  do jcol=jstart,jend
    do jlev=1,nlev
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```

@master

```
subroutine solver_lw(...)
  do jg = 1, ng
    do jlev=1,nlev
    !$acc parallel loop
      do jcol=jstart,jend
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      !$acc parallel loop
      do jcol=jstart,jend
        !...
      enddo
    enddo
  enddo
end subroutine
```

@Variant 2 GPU

Advantages:

Size of jcol is adjustable for best vectorization & parallelization

Issues:

Not best data locality for GPU performance [k offset values field(jg, jlev+1, jcol)]

Proposal for ECRAD on GPU

➔ Combination of block parallelism (acc gang) with vector parallelism, allows to combine jcol and jg to achieve enough occupancy on device.

```
subroutine solver_lw(...)
  !$acc loop independent gang
  do jcol=jstart,jend
    !$acc loop seq
    do jlev=1,nlev
      !$acc vector
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
  enddo
end subroutine
```

@Variant g0

Issues: what is OpenACC behaviour with calls to subroutines using automatic arrays allocation?

```
subroutine calc_reflectance_transmittance_lw(ng, &
    &    od, gamma1, gamma2, planck_top, planck_bot, &
    &    reflectance, transmittance, source_up, source_dn)

  real(jprb), intent(in), dimension(ng) :: od
  real(jprb), intent(in), dimension(ng) :: gamma1, gamma2
  real(jprb), intent(in), dimension(ng) :: planck_top, planck_bot
  ...
```

Variant g2: ncol-ng swap (GPU)

```
subroutine solver_lw(...)
  do jcol=jstart,jend
    do jlev=1,nlev
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
      do jg = 1, ng
        !...
      enddo
    enddo
  enddo
end subroutine
```

@master

```
subroutine solver_lw(...)
  !$acc loop independent gang
  do jg = 1, ng
    do jlev=1,nlev
    !$acc vector
      do jcol=jstart,jend
        !...
      enddo
    enddo
    !...
    do jlev=nlev,1,-1
    !$acc vector
      do jcol=jstart,jend
        !...
      enddo
    enddo
  enddo
end subroutine
```

@Variant g2

Advantages:

- Size of jcol is adjustable for best vectorization & parallelization
- Best data locality and parallelism

Variant g0: master GPU

➔ Combination of block parallelism (acc gang) with vector parallelism, allows to combine jcol and jg to achieve enough occupancy on device.

```
subroutine solver_lw(...)
  !$acc loop independent gang
  do jcol=jstart,jend
    !$acc loop seq
    do jlev=1,nlev
      !$acc vector
      do jg = 1, ng
        !...
      enddo
    enddo
    !...
  enddo
end subroutine
```

@Variant 0 GPU

```
subroutine solver_lw(...)
  !$acc loop independent gang
  do jg = 1, ng
    !$acc loop seq
    do jlev=1,nlev
      !$acc vector
      do jcol=jstart,jend
        !...
      enddo
    enddo
  enddo
end subroutine
```

@Variant 2 GPU

First summary

- Studied multiple strategies and variants for achieving portability of ECRAD
- Promising options, but still need to resolve issues (and avoid blockers)
- **Variant g0**: Use of acc gang + acc vector would provide enough parallelism and solve the memory. No refactoring required
  - Experiments required (see next slides)
- **Variant 1** (ncol-ng collapse): would provide good performance on GPU, but cash unfriendly on CPU
  - Not considered for experiments
- **Variant g2** (ncol-ng swap): would provide best flexibility to match the size of innermost dimension to vector length -> NEC & GPU. Loop refactorings required
  - Experiments required (see next slides)

ECRAD experiment: Variant g0

- Focusing on small part of solver_mcica_lw

```
subroutine solver_mcica_lw( … )

  tstart = omp_get_wtime()

  !$acc loop independent gang
  do jcol=jstart,jend
    !$acc loop seq
    do jlev=1,nlev
      call  calc_no_scattering_transmittance_lw(...)
    end do
    call  calc_fluxes_no_scattering_lw(...)
    end do
  end do

  tend = omp_get_wtime()

  do jcol=jstart,jend
    !...
  end do
end subroutine
```

```
subroutine calc_no_scattering_transmittance_lw( … )
  !$acc routine worker

  !$acc loop independent
   do jg =1,ng
     ...
   end do
end subroutine

subroutine calc_fluxes_no_scattering_lw( … )
  !$acc routine worker
  flux_dn(1:ng,1) = 0.0_jprb

   !$acc loop seq
   do jlev = 1,nlev
         flux_dn(1:ng,jlev+1) = flux_dn(1:ng,jlev) * …
   end do
   …
```

ECRAD experiment: Variant g2

- Focusing on small part of solver_mcica_lw

```
subroutine solver_mcica_lw( … )
  ! transpose data

  tstart = omp_get_wtime()

  !$acc loop independent gang
  do jg=1,ng
    !$acc loop seq
    do jlev=1,nlev
      call  calc_no_scattering_transmittance_lw(...)
    end do
    call  calc_fluxes_no_scattering_lw(...)
    end do
  end do

  tend = omp_get_wtime()

  ! transpose data back
  do jcol=jstart,jend
    !...
  end do
end subroutine
```

```
subroutine calc_no_scattering_transmittance_lw( … )
  !$acc routine worker

  !$acc loop independent
   do jcol =jstart,jend
     ...
   end do
end subroutine

subroutine calc_fluxes_no_scattering_lw( … )
  !$acc routine worker
  flux_dn(jstart:jend,1) = 0.0_jprb

   !$acc loop seq
   do jlev = 1,nlev
         flux_dn(jstart:jend,jlev+1) = flux_dn(:,jlev) * …
   end do

   …
```

ECRAD experiment: solver_mcica_lw

Set-up

- ng: 140, 40
- nblocksize: 16 for CPU ncol for GPU
- Ncol: 2500, 6000, 10000
- Number of workers on GPU: 5
- Vector length on GPU: 32
- Inline (calc_no_scattering_transmittance_lw, calc_fluxes_no_scattering_lw) using compiler flags: -Mextract/-Minline
- pgf90 20.4-0 -O3 -Mipa=fast

Hardware-node:

- 8 x NVIDIA® V100 GPUs (with 32 GB HBM memory each)
- 2 x Intel® Xeon® Skylake 6134 CPUs (8-core, 3.2 GHz)
- 384 GB DDR4 memory

Comparing variant g0 and g2 on CPU and GPU

	ncol	CPU [us/col]	CPU inlined [us/col]	GPU [us/col]	GPU inlined [us/col]	Speed-up CPU/GPU
Var. g0	2500	11	12	7.3	3.1	3.8
	6000	21	21	7.1	2.9	7.4
	10000	21	21	7.1	2.9	7.2
Var. g2	2500	14	14	3.8	3.6	3.9
	6000	48	46	3.3	3.6	12.8
	10000	45		46	3.3	3.6

Comparing Variant g0 and g2, inlined, nq=40

	ncol	CPU [us/col]	GPU (w=2) [us/col]	GPU (w=5) [us/col]	GPU (w=32) [us/col]	Speed-up CPU/GPU
Var. g0	2500	3.1	1.0	-	-	3.0
	6000	3.5	0.99	-	-	3.5
	10000	3.2	0.99	-	-	3.2
Var. g2	2500	3.1	-	2.4	0.94	3.3
	6000	4.2	-	2.4	0.87	4.8
	10000	3.2		-	2.4	0.94

Automatic Fortran Arrays in inner ACC function

Observations:

- Local variable (1:ng): prohibits inlining, roughly 10x slower
- Local variable (1:ng,1:nlev): prohibits inlining, at runtime: FATAL ERROR: FORTRAN AUTO ALLOCATION FAILED

Conclusion:

- Necessary refactor: moving allocation out of acc kernel regions
  - Code becomes slightly more complex
  - No negative performance impact on the CPU
  - Chance to reduce memory footprint

Final Conclusions for GPU port

- Parallelism on multiple levels:
  - Is manageable
- Many small device function calls:
  - Can be handled by compile-time inlining
- Allocation inside device functions:
  - Needs to be refactored, preallocation on kernel level
- Memory consumption:
  - (Introduce subblocking in ICON interface to ecRad, topic of another meeting)

Adding subblocking to ecRad interface

```
subroutine nwp_ecrad_radiation( … )

  ! allocate ecRad data structures (nproma, nlev)

  do jb=i_startblk, i_endblk
    ! copy variables from ICON to ecRad
    ! call ecRad
    ! copy variable back from ecRad to ICON
  end do

  ! deallocate ecRad data structures

end subroutine
```

```
subroutine nwp_ecrad_radiation( … )

  ! allocate ecRad data structures (nproma/nsublock, nlev)

  do jb=i_startblk, i_endblk
    do jsub=1, nsublock
      ! copy variables from ICON to ecRad          ⚠ careful with indexing
      ! call ecRad
      ! copy variable back from ecRad to ICON      ⚠ careful with indexing
    end do
  end do

  ! deallocate ecRad data structures

end subroutine
```

- nsubblock>1 can be used to reduce memory footprint (necessary for GPUs)
- nsubbock=1 is equivalent to current version

Comparing Variant g0 and g2 on CPU and GPU

	ncol	CPU [ms]	CPU inlined [ms]	GPU [ms]	GPU inlined [ms]	Speed-up CPU/GPU
Var. g0	160	*9.1*	*9.1*	1.11	0.691	*13.1*
	2500	29	30	18.3	7.72	3.8
	6000	130	130	42.6	17.6	7.4
	10000	210	210	70.5	29.2	7.2
Var. g2	160	*8.1*	*8.0*	0.935	0.643	*12.5*
	2500	36	35	9.47	8.98	3.9
	6000	290	280	19.9	21.8	12.8
	10000	450		460	32.6	36.3

Comparing Variant g0 and g2, inlined, ngp=40

	ncol	CPU [ms]	GPU (w=2) [ms]	GPU (w=5) [ms]	GPU (w=32) [ms]	Speed-up CPU/GPU
Var. g0	160	*1.98*	*0.494*	-	-	*4.0*
	2500	7.63	2.51	-	-	3.0
	6000	21.1	5.95	-	-	3.5
	10000	31.9	9.91	-	-	3.2
Var. g2	160	2.49	-	*0.497*	*0.617*	4.0
	2500	7.75	-	5.88	2.34	3.3
	6000	25.2	-	14.3	5.29	4.8
	10000	31.8		-	24.1	9.41