Fast computation of cloud 3D radiative effects in dynamical models by optimizing the ecRad scheme

Peter Ukkonen[1] and Robin J Hogan[2]

[1]Danish Meteorological Institute
[2]ECMWF

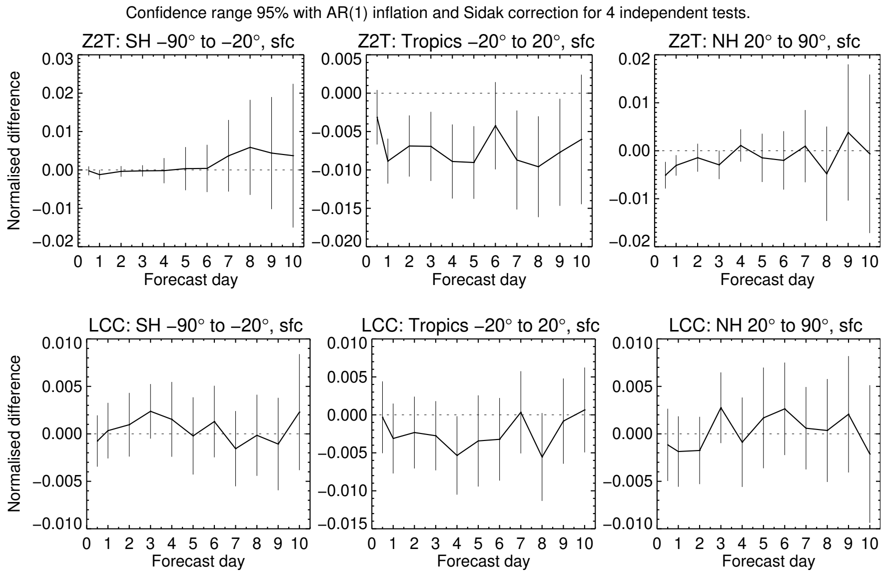
May 2, 2023

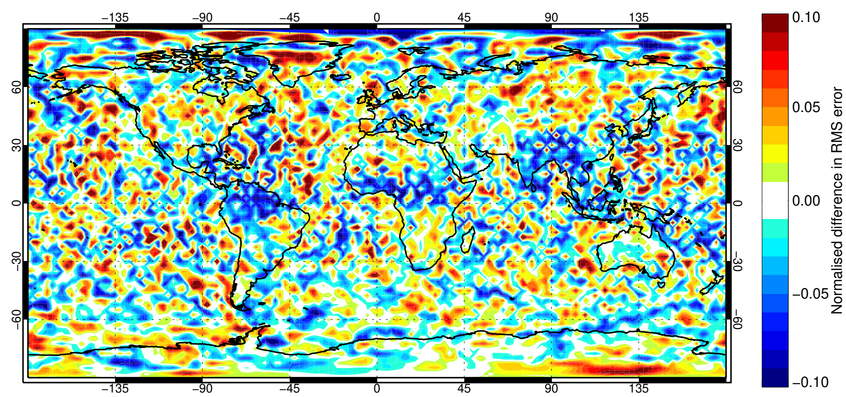
Abstract

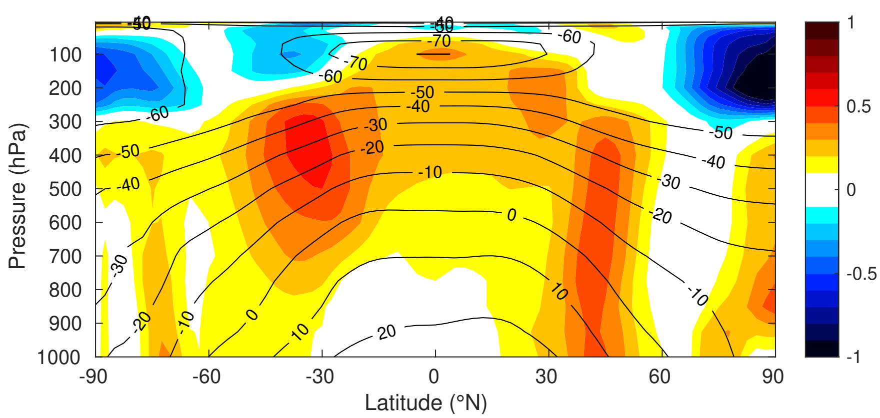
Radiation schemes are fundamental components of weather and climate models that need to be both efficient and accurate. In this work we refactor ecRad, a flexible radiation scheme developed at the European Centre for Medium-Range Weather Forecasts (ECMWF). The goal was to improve performance especially with ecCKD, a new gas optics scheme that requires only 32 spectral intervals in the longwave and shortwave to be accurate. This speeds up ecRad considerably, but also reduces performance due to short inner loops.

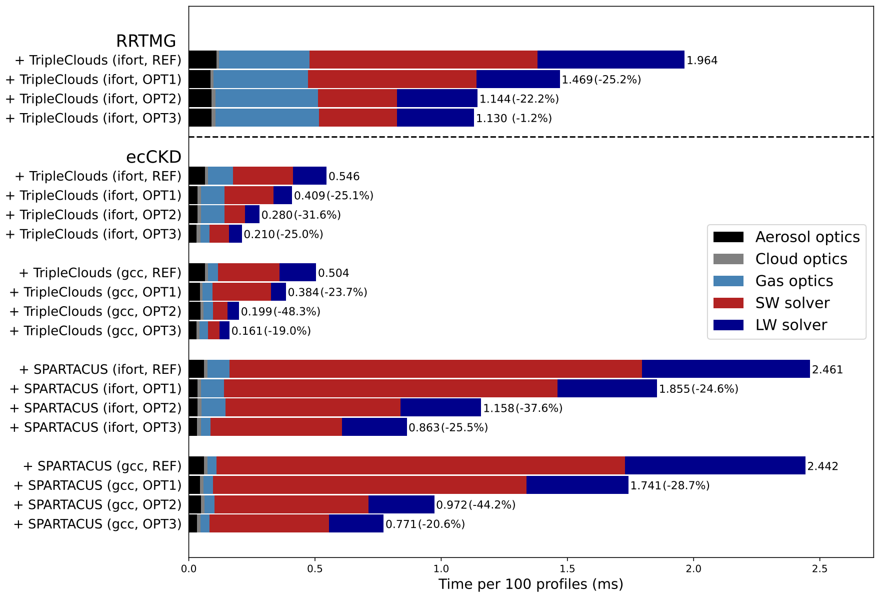
We therefore carry out both higher-level code restructuring and kernel-level optimizations for the radiative transfer solvers TripleClouds and SPARTACUS. SPARTACUS computes cloud 3D radiative effects, which have so far been neglected in large-scale models. We exploit the lack of vertical loop dependencies in key computations by merging the spectral and vertical dimensions, improving vectorization and instruction-level parallelism.

On the new AMD Rome-based ECMWF supercomputer, we obtain a 3-fold speedup for both solvers when using 32-term ecCKD models. Combining ecCKD with optimized code results in very fast yet accurate radiation computations: with TripleClouds we achieve 1.7 TFLOPs and a throughput of 621 columns/ms on a 128-core node. This is 11.5 times faster than ecRad in Integrated Forecasting System cycle 47r3, which uses a more noisy solver (McICA) and less accurate gas optics (RRTMG). SPARTACUS with ecCKD is now 2.4 times faster than CY47r3-ecRad, making cloud 3D radiative effects affordable to compute within large-scale models. Preliminary results show that SPARTACUS slightly improves forecasts of 2-metre temperature and low clouds in the tropics.

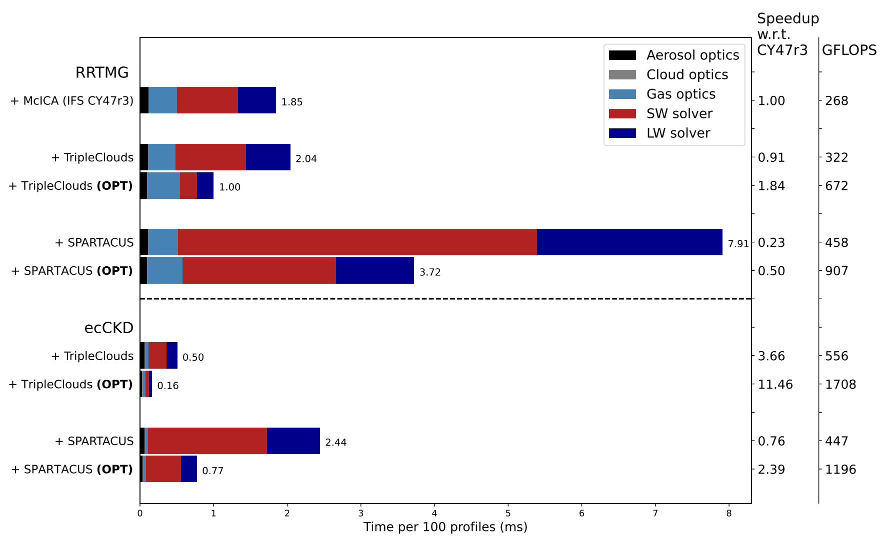

Confidence range 95% with AR(1) inflation and Sidak correction for 4 independent tests.
Z2T: SH –90° to –20°, sfc
Z2T: Tropics –20° to 20°, sfc
Z2T: NH 20° to 90°, sfc
LCC: SH –90° to –20°, sfc
LCC: Tropics –20° to 20°, sfc
LCC: NH 20° to 90°, sfc
Normalised difference
Forecast day


Normalised difference in RMS error
0.10
0.05
0.00
–0.05
–0.10

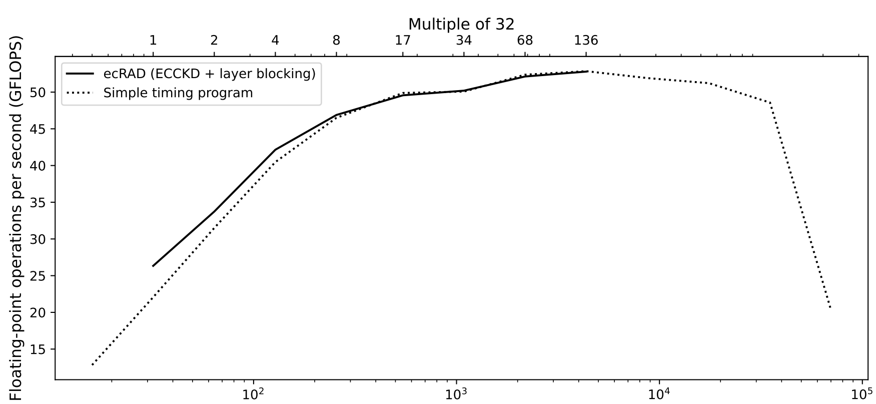

Pressure (hPa)
Latitude (°N)



RRTMG
+ TripleClouds (ifort, REF) 1.964
+ TripleClouds (ifort, OPT1) 1.469(-25.2%)
+ TripleClouds (ifort, OPT2) 1.144(-22.2%)
+ TripleClouds (ifort, OPT3) 1.130 (-1.2%)
ecCKD
+ TripleClouds (ifort, REF) 0.546
+ TripleClouds (ifort, OPT1) 0.409(-25.1%)
+ TripleClouds (ifort, OPT2) 0.280(-31.6%)
+ TripleClouds (ifort, OPT3) 0.210(-25.0%)
+ TripleClouds (gcc, REF) 0.504
+ TripleClouds (gcc, OPT1) 0.384(-23.7%)
+ TripleClouds (gcc, OPT2) 0.199(-48.3%)
+ TripleClouds (gcc, OPT3) 0.161(-19.0%)
+ SPARTACUS (ifort, REF) 2.461
+ SPARTACUS (ifort, OPT1) 1.855(-24.6%)
+ SPARTACUS (ifort, OPT2) 1.158(-37.6%)
+ SPARTACUS (ifort, OPT3) 0.863(-25.5%)
+ SPARTACUS (gcc, REF) 2.442
+ SPARTACUS (gcc, OPT1) 1.741(-28.7%)
+ SPARTACUS (gcc, OPT2) 0.972(-44.2%)
+ SPARTACUS (gcc, OPT3) 0.771(-20.6%)
Aerosol optics
Cloud optics
Gas optics
SW solver
LW solver
0.0
0.5
1.0
1.5
2.0
2.5
Time per 100 profiles (ms)



Speedup w.r.t. CY47r3
GFLOPS
RRTMG
+ McICA (IFS CY47r3) 1.85 1.00 268
+ TripleClouds 2.04 0.91 322
+ TripleClouds (OPT) 1.00 1.84 672
+ SPARTACUS 7.91 0.23 458
+ SPARTACUS (OPT) 3.72 0.50 907
ecCKD
+ TripleClouds 0.50 3.66 556
+ TripleClouds (OPT) 0.16 11.46 1708
+ SPARTACUS 2.44 0.76 447
+ SPARTACUS (OPT) 0.77 2.39 1196
Aerosol optics
Cloud optics
Gas optics
SW solver
LW solver
0
1
2
3
4
5
6
7
8
Time per 100 profiles (ms)



Multiple of 32
1
2
4
8
17
34
68
136
ecRAD (ECCKD + layer blocking)
Simple timing program
Floating-point operations per second (GFLOPS)
15
20
25
30
35
40
45
50
10^2
10^3
10^4
10^5

```fortran
do jlev = 1,nlay ! Start at top-of-atmosphere
 nreg = nregions
 if (is_clear_sky_layer(jlev) nreg = 1

 do jreg = 1,nreg ! Loop over relevant regions (only 1 if layer is clear-sky)
  if (jreg == 1) then ! optical properties are equal to clear-sky values
   optical_depth_tot  =  optical_depth(:,jlev,jcol)
   ssa_tot = ssa(:,jlev,jcol)
   g_tot   =   g(:,jlev,jcol)
  else
   do jg = 1,ng ! loop over g-points
    ! Cloudy-sky optical properties from band-wise cloud values and g-point-wise clear-sky values
    optical_depth_tot(jg) =  optical_depth(jg,jlev,jcol) + ...
    ...
   end do
  end if

  call calc_two_stream_gammas_sw(ng,  mu0, ssa_tot, g_tot,  gamma1, gamma2, gamma3)
  call calc_reftrans_sw(ng, mu0, optical_depth_tot, ssa_tot, gamma1, gamma2, gamma3, &
   &  reflectance(:,jreg,jlev), transmittance(:,jreg,jlev), & ! outputs
   &  ref_dir(:,jreg,jlev), trans_dir_diff(:,jreg,jlev), trans_dir_dir(:,jreg,jlev)) ! outputs
 end do
end do
```

⇓

```fortran
! Computations for clear-sky region as a separate step: collapse the two inner dimensions
call calc_reftrans_sw_opt(ng*nlay, mu0, optical_depth(:,:,jcol), ssa(:,:,jcol), g(:,:,jcol), &
 &  reflectance_clear, transmittance_clear, ref_dir_clear, trans_dir_diff_clear, trans_dir_dir_clear)

! Cloudy computations: start at top-of-atmosphere and find first cloudy layer, if one exists
any_clouds_below = .false.
jtop = findloc(is_clear_sky_layer(1:nlay), .false., dim=1)
if (jtop>0) any_clouds_below = .true.

do while (any_clouds_below)
 ! Find the bottom of this cloud
 jbot = ...
 nlay_cloud = jbot - jtop + 1
 allocate(optical_depth_tot_cloudy(ng,2:nreg,jtop:jbot), ssa_tot_cloudy(ng,2:nreg,jtop:jbot), &
    & g_tot_cloudy(ng,2:nreg,jtop:jbot))

 do jlev = jtop, jbot
  do jreg = 2, nregions ! = 3
   do jg = 1,ng
    ! Spectral cloudy-sky optical properties from band-wise cloud values and spectral clear-sky values
    optical_depth_tot_cloudy(jg,jreg,jlev) = ...
    ...
   end do
  end do
 end do

 call calc_reftrans_sw_opt(ng*2*nlay_cloud, & ! g-points * cloudy regions * adjacent cloudy layers
  &  mu0, optical_depth_tot_cloudy, ssa_tot_cloudy, g_tot_cloudy, &
  &  reflectance(:,:,jtop:jbot), transmittance(:,:,jtop:jbot), & ! outputs
  &  ref_dir(:,:,jtop:jbot), trans_dir_diff(:,:,jtop:jbot), trans_dir_dir(:,:,jtop:jbot)) ! outputs

 deallocate(optical_depth_tot_cloudy, ssa_tot_cloudy, g_tot_cloudy)

 ! Does another cloudy layer exist? If not, set logical to false to exit "while"
 if (jbot== nlay) any_clouds_below=.false. ! surface reached

 if (any(.not. is_clear_sky_layer(jbot+1:nlay))) then
  ! find the top of the new cloud
  jtop = ...
 else
  any_clouds_below=.false.
 end if
end do
```

Figure 1: Refactoring of TripleClouds-SW. In addition to optimizing and fusing kernels, in the new code (bottom) the reflectance-transmittance computations are performed in a batched manner for multiple layers by collapsing the spectral and vertical dimensions.

```
! Treat A and B each as n m-by-m square matrices (with the n dimension
! varying fastest) and perform matrix multiplications on all n matrix pairs
mat_x_mat = 0.0_jprb ! Array-wise assignment
mblock = m/3
m2block = 2*mblock
if (i_actual_matrix_pattern == IMatrixPatternShortwave) then
  ! Matrix has a sparsity pattern
  !     (C D E)
  !     (F G H)
  !     (0 0 I)
  ! Do the top-left (C, D, F, G)
  do j2 = 1,m2block ! 1,6
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m2block ! 1,6
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                  + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
  do j2 = m2block+1,m ! 7,9
    ! Do the top-right (E & H)
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                  + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
    ! Do the bottom-right (I)
    do j1 = m2block+1,m ! 7,9
      do j3 = m2block+1,m ! 7,9
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                  + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
else
...
```

⇓

```
pure subroutine mat_x_mat_sw_repeats(ng_sw_in, nlev_b, A, B, C)
  integer,    intent(in)                             :: ng_sw_in, nlev_b
  real(jprb), intent(in), dimension(ng_sw*nlev_b,9,9) :: A, B
  real(jprb), intent(out),dimension(ng_sw*nlev_b,9,9) :: C
  integer    :: j1, j2, j22
  !dir$ assume_aligned A:64,B:64,C:64
  ! Input matrices have pattern:
  !   (C          D          E)
  !   (F=-D       G=-C       H)
  !   (0          0          I), where each element is a 3-by-3 matrix
  ! As a result, output matrices have pattern:
  !   (C          D          E)
  !   (F=D        G=C        H)
  !   (0          0          I)
  do j2 = 1,3
    j22 = j2 + 6
    do j1 = 1,6
      ! Do the top-left (C, F)
      ! Unroll innermost matmul loop: more work for each iteration of SIMD loop
      C(:,j1,j2) = A(:,j1,1)*B(:,1,j2) + A(:,j1,2)*B(:,2,j2) + A(:,j1,3)*B(:,3,j2) &
      &          + A(:,j1,4)*B(:,4,j2) + A(:,j1,6)*B(:,6,j2)
      ! Do the top-right (E & H)
      C(:,j1,j22) = A(:,j1,1)*B(:,1,j22) + A(:,j1,2)*B(:,2,j22) + A(:,j1,3)*B(:,3,j22) &
      &      + A(:,j1,4)*B(:,4,j22) + A(:,j1,5)*B(:,5,j22) + A(:,j1,6)*B(:,6,j22) &
      &      + A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
    do j1 = 7,9  ! Do the bottom-right (I)
      C(:,j1,j22) = A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
  end do
  C(:,1:3,4:6) = C(:,4:6,1:3) ! D = F
  C(:,4:6,4:6) = C(:,1:3,1:3) ! G = C
  C(:,7:9,1:6) = 0.0_jprb     ! Lower left corner
```

Figure 1: Reference (top) and optimized (bottom) versions of the matrix-matrix multiplication kernel used in the shortwave matrix exponential computations. The latter unrolls loops and reduces work by exploiting that some matrix elements are repeated. For this performance-critical code, further speedup was gained by data alignment. The Intel compiler reported aligned data access only after declaring **ng_sw** at compile-time.

1

```fortran
! Initialize the derivatives at the surface; the surface is treated as a
 single
! clear-sky layer so we only need to put values in region 1.
lw_derivatives_g_reg = 0.0_jprb
lw_derivatives_g_reg(:,1) = flux_up_surf / sum(flux_up_surf)
lw_derivatives(icol, nlev+1) = 1.0_jprb

! Move up through the atmosphere computing the derivatives at each half-level
do jlev = nlev,1,-1
  ! Compute effect of overlap at half-level jlev+1, yielding
  ! derivatives just above that half-level
  lw_derivatives_g_reg = singlemat_x_vec(ng,ng,nreg,u_matrix(:,:,jlev+1),
 lw_derivatives_g_reg)

  ! Compute effect of transmittance of layer jlev, yielding
  ! derivatives just below the half-level above (jlev)
  lw_derivatives_g_reg = transmittance(:,:,jlev) * lw_derivatives_g_reg

  lw_derivatives(icol, jlev) = sum(lw_derivatives_g_reg)
end do
```

⇓

```fortran
...

! Move up through the atmosphere computing the derivatives at each half-level
do jlev = nlev,1,-1
  ! Inline everything in one loop over g-points
  lw_deriv_old  = lw_derivatives_g_reg
  sum_tmp       = 0.0_jprb
  associate(A=>u_matrix(:,:,jlev+1), b=>lw_deriv_old)
    !$omp simd reduction(+:sum_tmp)
    do jg = 1, ng
      ! Compute effect of overlap at half-level jlev+1, yielding derivatives just above that
      ! half-level (matrix-vector multiply)
      ! both inner and outer loop of the matrix loops j1 and j2 unrolled
      ! inner loop:                 j2=1              j2=2              j2=3
      lw_derivatives_g_reg(jg,1) = A(1,1)*b(jg,1) + A(1,2)*b(jg,2) + A(1,3)*b(jg,3)
      lw_derivatives_g_reg(jg,2) = A(2,1)*b(jg,1) + A(2,2)*b(jg,2) + A(2,3)*b(jg,3)
      lw_derivatives_g_reg(jg,3) = A(3,1)*b(jg,1) + A(3,2)*b(jg,2) + A(3,3)*b(jg,3)

      ! Compute effect of transmittance of layer jlev, yielding
      ! derivatives just below the half-level above (jlev)
      lw_derivatives_g_reg(jg,1) = lw_derivatives_g_reg(jg,1) * transmittance(jg,1,jlev)
      lw_derivatives_g_reg(jg,2) = lw_derivatives_g_reg(jg,2) * transmittance(jg,2,jlev)
      lw_derivatives_g_reg(jg,3) = lw_derivatives_g_reg(jg,3) * transmittance(jg,3,jlev)

      sum_tmp = sum_tmp + lw_derivatives_g_reg(jg,1) + lw_derivatives_g_reg(jg,2) + &
            & + lw_derivatives_g_reg(jg,3)
    end do
  end associate

  lw_derivatives(icol, jlev) = sum_tmp
end do
```

Figure 1: Reference (top) and optimized (bottom) version of the longwave derivatives kernel used by TripleClouds.

1

Fast computation of cloud 3D radiative effects in dynamical models by optimizing the ecRad scheme

Peter Ukkonen [1], Robin J. Hogan [2,3]

[1]Danish Meteorological Institute
[2]European Centre for Medium-Range Weather Forecasts, Reading, UK
[3]Department of Meteorology, University of Reading, Reading, UK

Key Points:

- The ecRad radiation scheme was sped up threefold by using code optimization
- Combining the optimized TripleClouds solver with new gas optics reduces the runtime of IFS radiation 11-fold
- Cloud 3D radiative effects can now be computed twice as fast as the operational scheme

Corresponding author: Peter Ukkonen, peterukk@gmail.com

Abstract

Radiation schemes are fundamental components of weather and climate models that need to be both efficient and accurate. In this work we refactor ecRad, a flexible radiation scheme developed at the European Centre for Medium-Range Weather Forecasts (ECMWF). The goal was to improve performance especially with ecCKD, a new gas optics scheme that requires only 32 spectral intervals in the longwave and shortwave to be accurate. This speeds up ecRad considerably, but also reduces performance due to short inner loops.

We therefore carry out both higher-level code restructuring and kernel-level optimizations for the radiative transfer solvers TripleClouds and SPARTACUS. SPARTACUS computes cloud 3D radiative effects, which have so far been neglected in large-scale models. We exploit the lack of vertical loop dependencies in key computations by merging the spectral and vertical dimensions, improving vectorization and instruction-level parallelism.

On the new AMD Rome-based ECMWF supercomputer, we obtain a 3-fold speedup for both solvers when using 32-term ecCKD models. Combining ecCKD with optimized code results in very fast yet accurate radiation computations: with TripleClouds we achieve 1.7 TFLOPs and a throughput of 621 columns/ms on a 128-core node. This is 11.5 times faster than ecRad in Integrated Forecasting System cycle 47r3, which uses a more noisy solver (McICA) and less accurate gas optics (RRTMG). SPARTACUS with ecCKD is now 2.4 times faster than CY47r3-ecRad, making cloud 3D radiative effects affordable to compute within large-scale models. Preliminary results show that SPARTACUS slightly improves forecasts of 2-metre temperature and low clouds in the tropics.

Plain Language Summary

A crucial step in simulating weather and climate is calculating how atmospheric radiation (shortwave radiation from the sun and terrestrial longwave radiation) interacts with Earth's atmosphere and surface. The complexity of the underlying physics has necessitated making approximations in how radiative transfer is treated, such as considering it only in upwards and downwards directions, thereby ignoring 3D effects. Even so, radiative transfer has historically been a computationally expensive component of weather and climate simulations.

Here we show that a state-of-the-art radiation code can be sped up threefold by using code optimization techniques that seek to maximise performance on modern processors. Combining this with a recent innovation that reduces the number of spectral computations required for accurate solutions, an order-of-magnitude increase in speed is obtained compared to the existing radiation scheme in a global weather model. Crucially, these improvements also make a radiation scheme that accounts for 3D radiative effects by clouds fast enough to be used operationally. When included in global simulations, these 3D effects act to warm the lower atmosphere substantially.

1 Introduction

Atmospheric radiation is well-understood, but too complex to be solved in an exact manner in weather and climate models. That is, with the exception of the treatment of sub-grid cloud structure (which can easily become a dominant source of error), highly accurate solutions to atmospheric radiative transfer are available but too costly for dynamical models. This leaves the parameterization of radiation as an exercise in how to obtain as accurate broadband longwave and shortwave fluxes as possible at the least possible computational cost. For the spectral integration, the correlated-k-distribution method (CKD, e.g. Goody et al., 1989) has emerged as a leading solution. CKD is based on reordering the highly detailed absorption spectra of atmospheric gases by its optical properties into a cumulative probability function. Accurate spectral integration then becomes possible with only O(10^2 - 10^3) quadrature points - in CKD schemes these pseudo-monochromatic spectral intervals are referred to as k-terms or g-points - compared with O(10^6 - 10^7) for line-by-line methods which resolve individual spectral lines.

Despite the use of CKD, and considering the transfer of diffuse radiation only in the upward and downward directions ('two streams'), radiation computations are expensive enough that their temporal and/or spatial frequency is often limited. In high-resolution forecasts based on the IFS, a global numerical weather prediction (NWP) model developed at ECMWF, radiation is called every hour on a grid with roughly 10 times fewer columns than the rest of the model (Hogan & Bozzo, 2018). Such approximations are a source of uncertainty in large-scale models. In particular, 3D radiative effects by clouds are routinely ignored in weather and climate simulations, yet were estimated by Schafer (2017) to be similar in magnitude to anthropogenic greenhouse gas forcing (this does not imply they are as important for climate projections, as 3D effects are not *changing* and biases associated with missing processes are generally offset by model tuning). Due to the spatial and temporal coarsening, ecRad is only a few percent of the total IFS runtime (Hogan & Bozzo, 2018), but radiation becomes more expensive for larger-scale simulations where it must be called at a higher frequency relative to the model time step. For instance, in a coarse-resolution setup of the ECHAM climate model, radiation accounted for half of the runtime of the atmospheric model (Cotronei & Slawig, 2020).

The perceived expense of radiation schemes has led to attempts to replace them with a faster and approximative neural network (NN) emulator (Chevallier et al., 1998; Krasnopolsky et al., 2008; Pal et al., 2019; Liu et al., 2020; Roh & Song, 2020; Song & Roh, 2021; Kim & Song, 2022), avoiding explicit spectral computations and typically predicting heating rates directly. While large speed-ups of 1-2 orders of magnitude have been achieved, this approach can suffer from not only worse accuracy but also a lack of energy conservation, generalization and flexibility. For example, emulators are almost always tied to a specific vertical grid, and are less interpretable and configurable than modern radiation schemes which use different modules to compute the optical properties of gases, aerosols and clouds, and combine these in a radiative transfer solver. The advantages of flexibility, also with regards to vertical grids, were retained in Ukkonen et al. (2020) by only replacing the gas optics component with NNs. Radiative forcings with respect to individual greenhouse gases, important for climate applications, may also not be well represented by top-down emulators (we are not aware of any full-emulation paper evaluating these). Although ML emulators may yet prove useful, for instance by being able to run on graphics processing units (GPUs), a recent study (Ukkonen, 2022a) indicates that they suffer from similar speed-accuracy trade-offs as radiation schemes: a recurrent NN approach which structurally mimics radiative transfer computations gave much better accuracy than dense networks, but also a much smaller speed-up.

Fortunately, the reliable radiative transfer equations need not be sacrificed at the altar of efficiency. Algorithmic developments can, for instance, substantially reduce the number of spectral terms required for a given level of accuracy (Hogan & Matricardi, 2022). It may also be argued that the use of code restructuring to better exploit modern CPU's represents an underutilized potential for many physics codes. In one case, a modern radiation scheme was made roughly 3 times faster by combining a refactoring of the radiative transfer solver with replacing the gas optics module with a NN version (Ukkonen et al., 2020). In another, code restructuring of the RRTMG radiation scheme also improved speed threefold on targeted Intel hardware (Michalakes et al., 2016). In many legacy codes, the baseline performance may be much worse (Michalakes et al., 2016). While the independent column framework used in sub-grid parameterizations enables straightforward parallelization across multiple cores, exploiting other types of parallelism offered by modern CPUs, namely SIMD (single instruction, multiple data) vectorization, or instruction-level parallelism, may be considerably more challenging. Similarly, efficient use of complex cache memory hierarchies is anything but guaranteed. For any potentially expensive physics routine that is likely called within an OpenMP loop in a NWP or climate model, it follows that knowledge of basic optimization techniques of serial code becomes important, especially so as simulations are being performed at increasingly high resolution, with ever higher energy costs (Fuhrer et al., 2018).

Related to this, the move towards heterogeneous supercomputing platforms which incorporate accelerators presents a great challenge and necessitates re-thinking how we write and maintain code (Lawrence et al., 2018). An example of how this can be tackled at the parameterization level is found in the RTE+RRTMGP radiation code (Pincus et al., 2019), which makes use of isolated computational objects that can be adapted to new hardware platforms. Whether CPU or GPU, hardware is evolving towards higher levels of parallelism, as simply increasing clock counts is no longer feasible. Allowing this to influence not only algorithm design and implementation, but also the choice of algorithm, may therefore be prudent. For radiation, schemes based on CKD have traditionally been expensive enough to have kept less accurate broadband schemes relevant, as they have allowed spatially or temporally more frequent radiation computations. However, CKD has a higher level of parallelism owing to the independent spectral computations, and so hardware trends may favour it over band-based approaches. In total a CKD-based radiation scheme has two "embarassingly parallel" dimensions (columns and g-points), and a partially parallelizable vertical dimension (as not all computations have vertical loop dependencies). If the code is organized in a way where this parallelism can be fully exploited by the hardware, high performance can be achieved.

With this in mind, we describe various optimizations for ecRad, a flexible and open-source CKD-based radiation scheme developed at ECMWF. Our main goal was to improve the performance with ecCKD, a new gas optics scheme which uses relatively few k-terms (only 32 for the candidate SW and LW models). This improves speed but also reduces efficiency of the vectorized code by shortening vectorized loops. To address this we restructure the longwave (LW) and shortwave (SW) versions of the TripleClouds and SPARTACUS solvers (Hogan et al., 2016). While targeting ECMWF's new HPC platform based on AMD Zen 2 ('Rome') microarchitecture, expressing more parallelism should also help prepare ecRad for GPUs. In addition we optimize many kernels, e.g. to avoid the use of double precision in numerically sensitive two-stream calculations, which requires tuning some coefficients and introducing physical or numerical securities in order to avoid substantial errors in fluxes. We note that thorough refactoring of SPARTACUS is a laboursome undertaking; being a more sophisticated solver, the shortwave alone contained more than 1500 lines of code (excluding subroutines). We follow a simple strategy based on manually instrumenting ecRad code to get a profile of the runtimes and estimates of floating point operations per second (FLOPS) for different code sections. Although this is not always a useful metric, radiation codes are computationally intensive, and code sections with significant runtimes and low FLOPS indicated optimization potential. Unfortunately, the code contained relatively few hotspots and in total, many person months were spent on the refactoring. However, the effort should be well spent as SPARTACUS is the only radiation scheme that is capable of representing 3D radiative effects at a relatively low cost, having previously been 5.8 times slower than the McICA solver used in the IFS (Hogan & Bozzo, 2018). This difference is reduced by the use of ecCKD. A major goal was to eliminate the remaining gap and make SPARTACUS fast enough to be considered for operational use in weather and climate models.

The bulk of the paper concerns optimizations to ecRad; following a brief overview of the radiation scheme and its relevant components (Section 2) we describe the high-level code restructuring to improve performance (Section 3). In Section 4, we list some other optimizations that were used, while kernel-specific changes are detailed in Appendix A. We then evaluate runtimes and performance in Section 5. Given that global simulations with SPARTACUS have not yet been published, some preliminary results of the impact of 3D cloud radiative effects in the IFS are presented in Section 6, followed by concluding remarks (Section 7).

2 The ECMWF radiation scheme 'ecRad'

The ecRad radiation scheme was developed at ECMWF and has been used operationally in the IFS since 2017 (Hogan & Bozzo, 2018) and by the German Weather Service (DWD) since 2021, as well as being available for anyone to use under an open-source license. It is highly configurable, with the capability for the four main components (the radiative transfer solver and the calculation of the optical properties of gases, aerosols and clouds) to be changed independently of each other. Two of these components offer opportunities for a significant trade-off between accuracy and efficiency: the solver (discussed in section 2.1) and the treatment of gas optics (section 2.2).

2.1 Radiative transfer solvers

The solver takes as input the optical properties of the atmosphere in different spectral regions, and computes profiles of broadband fluxes from which heating rates may be computed. The main challenge is to represent sub-grid cloud structure. The McICA solver (Monte Carlo Independent Column Approximation) is used operationally by ECMWF and DWD, and feeds each spectral interval of the radiative transfer calculation with a different stochastic realization of the cloud profile. The McICA implementation described by Hogan and Bozzo (2018) exactly respects the total cloud cover prescribed by the model's overlap assumptions, as well as the fraction of clouds exposed to space at each level. However, the model's assumption on sub-grid heterogeneity of cloud water content is only respected in a statistical sense, so there is a modest amount of noise in instantaneous radiative fluxes.

The TripleClouds solver (Shonk & Hogan, 2008) takes a quite different approach: each layer containing cloud is divided horizontally into three 'regions', one clear and two cloudy, with the water contents of the two cloudy regions chosen to best approximate the radiative impact of the full probability distribution of cloud water assumed by the model. The model's overlap assumptions are used to pass the fluxes between adjacent layers in a way that reproduces exactly the same total cloud cover as used by McICA, but the fluxes are free from stochastic noise.

The SPARTACUS (Speedy Algorithm for Radiative Transfer through Cloud Sides) solver of Hogan et al. (2016) describes the sub-grid cloud field in the same way as TripleClouds, but terms are added to the equations to allow radiation to flow laterally between regions at a rate proportional to the assumed length of the interface between them, flows that are neglected in all operational radiation schemes worldwide. In the shortwave, this approach to representing 3D radiative transfer has been found to perform well against reference Monte Carlo radiation calculations for a wide range of cloud types (Hogan et al., 2019), capturing differences with traditional 1D radiative transfer of as much as 40 W m^{-2}. In the longwave, emission from cloud sides acts to increase the cloud radiative effect, but preliminary evaluation against Monte Carlo calculations suggests that the SPARTACUS somewhat overestimates this 3D effect; work is ongoing to improve the physical assumptions made in the longwave. It was reported by Hogan and Bozzo (2018) that compared to TripleClouds, SPARTACUS makes ecRad 3.3 times slower, while compared to McICA, SPARTACUS makes ecRad 5.8 times slower. Thus, SPARTACUS is a good example of a parameterization that offers a more accurate representation of the real world but is too expensive to deploy operationally, and therefore with code optimization could become affordable for operational use.

2.2 The RRTMG and ecCKD gas-optics scheme

The gas-optics component dictates the spectral resolution of the entire radiative transfer scheme, and scales its overall computational cost. Like the radiation schemes of many weather and climate models worldwide, ecRad by default computes the spectral absorption of gases using the Rapid Radiative Transfer Model for General Circulation Models, RRTMG (Mlawer et al., 1997), which uses a total of 140 spectral intervals in the longwave and 112 in the shortwave.

Hogan and Matricardi (2022) recently developed the ECMWF Correlated k-Distribution tool 'ecCKD', which generates gas-optics models in the form of look-up tables that can be stored in a single configuration file. Since version 1.4, ecRad has the capability to use ecCKD gas-optics models. Hogan and Matricardi (2022) used three techniques to reduce the number of spectral intervals while retaining accuracy: the full-spectrum correlated-k method, the hypercube partition method for treating the spectral overlap of gases, and the optimization of look-up table coefficients against a set of training profiles. We use their models with 32 spectral intervals in each of the longwave and shortwave; since this is several times fewer than used by RRTMG, we expect a speed-up of the entire radiation scheme.

3 High-level code restructuring to expose more parallelism

3.1 Motivation

In both TripleClouds and SPARTACUS, the computation of layer reflectances, transmittances and source functions take a large share of the total runtime. In the reference code, these kernels are called within a vertical loop, and contain SIMD-vectorized loops over g-points, the innermost dimension in ecRad. This is problematic for ecCKD as it results in loops that are too short (e.g. 32 iterations) to efficiently utilize modern CPU's. Similarly to a car assembly line which can produce cars at a rate that is much faster than the time taken to produce an individual car, microprocessors have a level of parallelism that comes from *instruction pipelining*. Because pipelined instructions include a wind-up and wind-down phase where microprocessor units are idling for a given number of cycles - the number of overlapped instructions, known as latency or *depth* - the throughput (number of operations per cycle) when executing N independent operations with a pipeline of depth m is given by $p = \frac{1}{1+\frac{m-1}{N}}$ (Hager & Wellein, 2010).

In the reference code, the reflectance-transmittance kernels are called inside a vertical loop and N is equal to the number of g-points. With ecCKD, $N = 32$, and to obtain a decent efficiency of e.g. $p = 0.64$ results per cycle, we arrive at $m = 19$. However, complex calculations can have much longer latencies than this, with the exponential function alone having a longer latency. The computations of reflectance and transmittance using a two-stream approximation are very involved and include many high-latency operations such as floating point division. This can easily lead to the instruction stream being stalled ('pipeline bubble'). Vector or superscalar parallelism makes the situation even worse as multiple identical pipelines operating in parallel decreases the loop length of each pipe (Hager & Wellein, 2010).

Knowing that the exponential function (used in the two-stream kernels to compute transmittance from optical depth) alone has a long latency, simply moving it outside of the long SIMD-vectorized loop with other complex arithmetic improved performance by alleviating such a pipeline stall. However, even after the separately vectorized exponential it is useful, if possible, to increase N. Luckily, this can be done by exploiting the lack of vertical dependencies in the underlying computations. Specifically, collapsing the vertical and g-point dimension together prior to the kernel calls acts to increase the length of SIMD-vectorized loops (improving vectorization and instruction-level parallelism) and also reduces overhead from procedure calls.

3.2 Batched clear-sky computations

Beginning with the most trivial change, in both TripleClouds and SPARTACUS the computation of clear-sky reflectance and transmittance is performed for all layers (regardless of whether they contain clouds) and so the subroutine call can simply be moved outside a vertical loop and the two inner dimensions collapsed, e.g. `call calc_reftrans_opt (ng*nlev, od(:,:,jcol), ..., reflectance_clear, ...)`. Here the first argument gives the length of the SIMD-vectorized dimension i.e. number of g-points (`ng`) times number of layers, or levels as they are called in ecRad (fluxes, meanwhile, are defined at `nlev+1` 'half-levels'). The performance of the shortwave reflectance-transmittance kernel (which includes optimizations described in Appendix A) as a function of the vectorized dimension N is shown in Figure 1. Optimal performance with ecCKD is achieved when the vertical dimension is fully collapsed with the spectral dimension, without the need for blocking, with roughly doubled performance compared to the previous code layout where the length of the vectorized loop equals `ng`=32. The new structure is efficient also when using other gas optics schemes, as considerably larger spectral and/or vertical dimensions can be accommodated before a performance drop-off occurs when the arrays can no longer fit in faster cache. The trade-off is a small increase in code complexity, as it requires the reflectances and transmittances to be split into separate arrays for clear-sky and cloudy regions: `reflectance_clear(ng, nlev)` and `reflectance_cloudy(ng, 2:nregions , nlev)` instead of `reflectance(ng, nregions, nlev)`, but in practice other code sections are hardly affected as flux computations depend on the presence of clouds anyway. Another benefit is that overhead from subroutine calls is much reduced.

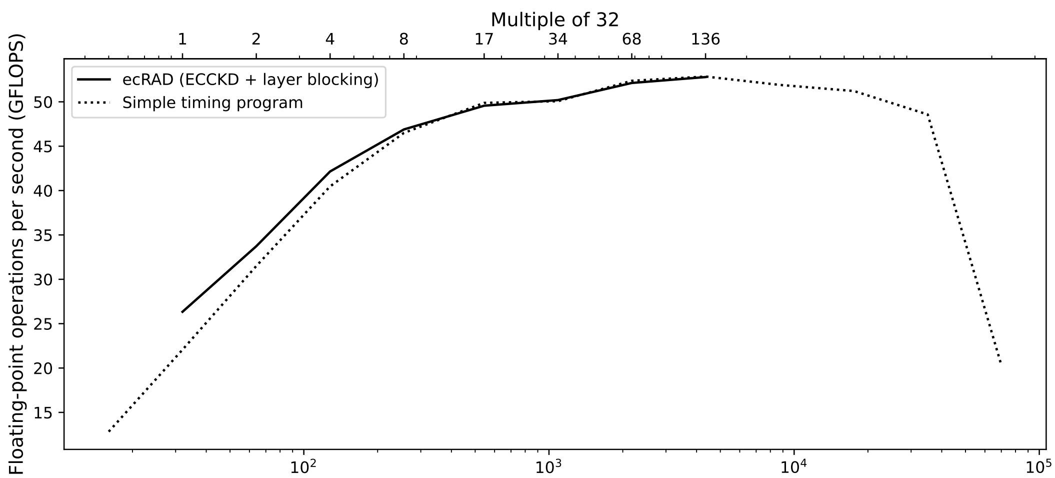


Figure 1: Serial single-precision performance of the optimized shortwave two-stream kernel (y-axis) versus loop length N (x-axis). The solid black line shows the performance as measured within a realistic program running the full radiation code for 7320 columns using a column block size of 8, ecCKD gas optics, the TripleClouds solver, and blocking also in the vertical dimension with different block sizes (top x-axis) to test the impact of varying N. Conveniently, the performance peaks around N corresponding to the number of g-points in ecCKD (32) times the number of vertical levels in the IFS high-resolution model (137), meaning that collapsing the g-point dimension with the vertical dimension results in optimal performance on this platform (AMD Ryzen 9 3900, GNU Fortran 9.3). The dotted line was obtained using a simple timing program that calls the kernel with synthetic data in order to test a wider range of N.

3.3 Batched cloudy computations

The lack of loop dependencies in the vertical dimension can likewise be exploited in the more demanding reflectance-transmittance computations for cloudy layers and regions, but this requires batching together the two cloudy regions and/or adjacent cloudy layers. The best way to do this depends on the particular solver.

3.3.1 TripleClouds-SW

In shortwave TripleClouds, we collapse the g-point, region and vertical dimensions by grouping together adjacent cloudy layers. This was implemented with a `do while` loop which checks if any cloudy layers still exists and finds the top and bottom of this extended cloudy layer, as illustrated in Fig. 2. The new code leads to a vectorized dimension of $2 \times ng \times nlay_{cloud-depth}$ in the cloudy reflectance-transmittance computations.

```fortran
do jlev = 1,nlay ! Start at top-of-atmosphere
 nreg = nregions
 if (is_clear_sky_layer(jlev) nreg = 1

 do jreg = 1,nreg ! Loop over relevant regions (only 1 if layer is clear-sky)
  if (jreg == 1) then ! optical properties are equal to clear-sky values
   optical_depth_tot  =  optical_depth(:,jlev,jcol)
   ssa_tot = ssa(:,jlev,jcol)
   g_tot   =   g(:,jlev,jcol)
  else
   do jg = 1,ng ! loop over g-points
    ! Cloudy-sky optical properties from band-wise cloud values and g-point-wise clear-sky values
    optical_depth_tot(jg) =  optical_depth(jg,jlev,jcol) + ...
    ...
   end do
  end if

  call calc_two_stream_gammas_sw(ng,  mu0, ssa_tot, g_tot,  gamma1, gamma2, gamma3)
  call calc_reftrans_sw(ng, mu0, optical_depth_tot, ssa_tot, gamma1, gamma2, gamma3, &
   &  reflectance(:,jreg,jlev), transmittance(:,jreg,jlev), & ! outputs
   &  ref_dir(:,jreg,jlev), trans_dir_diff(:,jreg,jlev), trans_dir_dir(:,jreg,jlev)) ! outputs
 end do
end do
```

$\Downarrow$

```fortran
! Computations for clear-sky region as a separate step: collapse the two inner dimensions
call calc_reftrans_sw_opt(ng*nlay, mu0, optical_depth(:,:,jcol), ssa(:,:,jcol), g(:,:,jcol), &
 &  reflectance_clear, transmittance_clear, ref_dir_clear, trans_dir_diff_clear, trans_dir_dir_clear)

! Cloudy computations: start at top-of-atmosphere and find first cloudy layer, if one exists
any_clouds_below = .false.
jtop = findloc(is_clear_sky_layer(1:nlay), .false., dim=1)
if (jtop>0) any_clouds_below = .true.

do while (any_clouds_below)
 ! Find the bottom of this cloud
 jbot = ...
 nlay_cloud = jbot - jtop + 1
 allocate(optical_depth_tot_cloudy(ng,2:nreg,jtop:jbot), ssa_tot_cloudy(ng,2:nreg,jtop:jbot), &
    & g_tot_cloudy(ng,2:nreg,jtop:jbot))

 do jlev = jtop, jbot
  do jreg = 2, nregions ! = 3
   do jg = 1,ng
    ! Spectral cloudy-sky optical properties from band-wise cloud values and spectral clear-sky values
    optical_depth_tot_cloudy(jg,jreg,jlev) = ...
    ...
   end do
  end do
 end do

 call calc_reftrans_sw_opt(ng*2*nlay_cloud, & ! g-points * cloudy regions * adjacent cloudy layers
  &  mu0, optical_depth_tot_cloudy, ssa_tot_cloudy, g_tot_cloudy, &
  &  reflectance(:,:,jtop:jbot), transmittance(:,:,jtop:jbot), & ! outputs
  &  ref_dir(:,:,jtop:jbot), trans_dir_diff(:,:,jtop:jbot), trans_dir_dir(:,:,jtop:jbot)) ! outputs

 deallocate(optical_depth_tot_cloudy, ssa_tot_cloudy, g_tot_cloudy)

 ! Does another cloudy layer exist? If not, set logical to false to exit "while"
 if (jbot== nlay) any_clouds_below=.false. ! surface reached

 if (any(.not. is_clear_sky_layer(jbot+1:nlay))) then
  ! find the top of the new cloud
  jtop = ...
 else
  any_clouds_below=.false.
 end if
end do
```

Figure 2: Refactoring of TripleClouds-SW. In addition to optimizing and fusing kernels, in the new code (bottom) the reflectance-transmittance computations are performed in a batched manner for multiple layers by collapsing the spectral and vertical dimensions.

3.3.2 TripleClouds-LW

In longwave TripleClouds we decided to batch the reflectance-transmittance computations only over g-points and the two cloudy regions, but not layers, as this was slightly faster on the tested platform. To achieve better performance on platforms with longer vector lengths it would likely be worth the increase in memory footprint to batch over the vertical dimension as well, but we did not wish to sacrifice performance on the targeted hardware or write more complex code to allow both options at this point.

3.3.3 SPARTACUS-SW

SPARTACUS represents cloud 3-D radiative effects by adding extra terms to the two-stream equations. The coupled system of equations can be solved by a method based on the matrix exponential. In both LW and SW solvers, these matrix exponentials are a computational hotspot, with the shared `expm` kernel accounting for almost 50% of the total runtime of the reference code. The matrix exponential is performed for each '3D' g-point in cloudy layers, where 3D effects are not considered for g-points which have very large optical depths that exceed a threshold. Because the individual matrices for which the matrix exponential is computed have small sizes corresponding to the total number of clear and cloudy regions, (`nreg` × 3, `nreg` × 3), they are placed non-contiguously in memory and the g-point dimension is vectorized instead. To vectorize over '3D' g-points, it is assumed that prior to the solver the g-points have been reordered in approximate order of gas optical depth which is in practice is implemented using a hard-coded mapping. Clear-sky optical depths are then searched for the cut-off index `ng3D` used in `expm`, which is dominated by matrix-matrix multiplications implemented as C(1:`ng3D`,j1,j2) = C(1:`ng3D`,j1,j2) + A(1:`ng3D`,j1,j3) × B(1:`ng3D`,j3,j2). Kernel-level optimization of the shortwave kernel `expm_sw` is described in Appendix A.

Efficiency can again be improved by collapsing the spectral and vertical dimensions. We exploit the same principle as in TripleClouds-SW, batching multiple cloudy layers using a `do while` loop. Recognizing that in the shortwave, `ng3D` is typically close to `ng`, 3D computations can be performed for all g-points without much redundancy (capping the optical depths to the threshold value), and flattening the two dimensions. This results in a vectorized dimension of `ng` × $\texttt{nlay}_{\text{cloud-depth}}$ instead of `ng3D` ≈ `ng`. As an added benefit, SPARTACUS-SW no longer requires g-points to be ordered by optical depth, eliminating any errors associated with assuming a constant reordering. Because the computation of reflectances, transmittances and source functions (longwave only) in SPARTACUS involves many intermediate arrays which are quite large, to use cache memory efficiently it is useful to ensure that the number of batched layers does not become too large. In the refactored code we set this threshold using a simple expression that depends on `ng` and working precision, and a constant tuned to result in 6 layers when using 32 g-points and single precision (if using double precision, only 3 layers would be batched). This gave good performance on the AMD platform; for optimal performance the user may wish to tune the maximum batch size to the hardware at hand.

Finally, after reflectances and transmittances have been determined, the solver works its way up from the surface to the top-of-atmosphere computing the total albedos (the albedo of the entire atmosphere below a layer). In the shortwave, this includes the computation of entrapment (Hogan et al., 2019) where the rate of exchange between the subregions in a given layer and the subregions in the layer above is computed via a coupled differential equation written in terms of a singular exchange matrix. This is once again solved using the matrix-exponential method. The simpler structure of these matrices enables using a faster method described in the appendix of Hogan et al. (2018). Nonetheless, these computations represent a small hotspot. While the loop-carried dependencies prevent batching across the vertical dimension as for `expm`, it is possible to batch the `fast_expm` computations across the three subregions times two (being performed for both diffuse and direct albedo), increasing the vectorized dimension by a factor of 6.

3.3.4 SPARTACUS-LW

In the longwave, the fraction of g-points which have optical depths small enough for 3D effects to matter is typically much lower than in the shortwave, and doing them for all g-points would result in a great deal of redundancy. Therefore, the code was restructured to collect all the '3D' g-points from adjacent cloudy layers, where `ng3D` varies by layer, into larger arrays with the inner dimension $\texttt{ng3D}_{\mathrm{tot}}$. This increases code complexity and introduces overhead but is worth it as the time spent in `expm` was more than halved (when using ecCKD and optimized kernel) due to avoiding very inefficient calls with small loop lengths. This change made SPARTACUS-LW faster by roughly a third.

4 Other optimizations

Many other optimization techniques were applied across the radiation scheme, including loop unrolling, loop fusion (often made possible by inlining functions), and avoiding temporary arrays. Here we list some general optimizations - employed in different modules of the radiation scheme - below, and refer the reader to Appendix A for an account of kernel-specific optimizations, which included important but painstaking work of porting code fully to single precision. (We only discuss this aspect for the two-stream kernel but note that SPARTACUS had issues with numerical instability that were especially difficult to solve as they were not immediately reproducible offline).

- Declaring `ng` at compile time. In ecRad, the spectral dimension, whose length is given by the number of g-points `ng`, is the leading dimension and in many sections cannot be collapsed with the vertical dimension. Simply declaring $\texttt{ng}_{\mathrm{SW}}$ and $\texttt{ng}_{\mathrm{LW}}$ at compile time can improve performance of ecRad with ecCKD by up to 25% (Table 1) by allowing the compiler to optimize many such short loops in the solvers, aerosol optics and gas optics. This was implemented using a preprocessing directive `#ifdef ng_sw` which sets the leading dimension to a parameter if it is passed to the compiler, and to a procedure argument `ng_sw_in` if it is not.
- Removing conditionals. Conditional branches to prevent division by zero, e.g. in sections where optical properties from gases, clouds and aerosols are combined within a spectral loop, were replaced with the use of max(*value*, *some number*) in the denominator by recognizing that if the denominator was zero the numerator was also zero. In the LW two-stream kernel moving a necessary conditional to a separate loop also improved performance by vectorizing the more compute-intensive parts.
- Merged broadband flux computations. The last step in the solver is to compute broadband fluxes by summing the fluxes defined at g-points and three regions. In the shortwave, this reduction over two dimensions is performed for three variables: upwelling, downwelling, and direct downwelling flux. By doing all three sums in a single loop over g-points with the *SIMD reduction* clause in OpenMP, and manually unrolling the sum over regions, the arithmetic intensity can be greatly improved compared to having separate calls to the *sum* intrinsic function:

```
! Store the broadband fluxes
! flux%sw_up(jcol,jlev+1) = sum(sum(flux_up,1))
! flux%sw_dn(jcol,jlev+1) = mu0 * sum(sum(direct_dn,1)) + sum(sum(flux_dn,1))
sums_up = 0.0_jprb; sums_dn = 0.0_jprb; sums_dn_dir = 0.0_jprb
!$omp simd reduction(+:sums_up, sums_dn, sums_dn_dir)
do jg = 1, ng_sw
  sums_up = sums_up + flux_up(jg,1) + flux_up(jg,2) + flux_up(jg,3)
  sums_dn = sums_dn + flux_dn(jg,1) + flux_dn(jg,2) + flux_dn(jg,3)
  sums_dn_dir = sums_dn_dir + direct_dn(jg,1) + direct_dn(jg,2) + direct_dn(jg,3)
end do
flux%sw_up(jcol,jlev+1) = sums_up
flux%sw_dn(jcol,jlev+1) = mu0*sums_dn_dir + sums_dn
```

  For cloud-free layers, summing over cloudy regions (regions 2-3) can be skipped.
- Avoiding temporary arrays. In many sections, one or more temporary arrays were removed by using the output array(s) of a subroutine for intermediate computations and/or by reusing temporary/local arrays. Code clarity was retained by the use of Fortran's *associate* construct.

5 Timing results

We evaluate performance by running an offline version of ecRad, which can be compiled with both reference and optimized code, on a single node of ECMWF's new AMD-based supercomputer. The results were obtained using a test case of 10,000 columns randomly sampled from a global snapshot from a high-resolution IFS simulation (00 UTC 2020/04/30), repeated 4 times for a total of 40,000 profiles with 137 vertical levels. Fig. 3 shows ecRad runtimes with a breakdown into components, as well as the overall single-precision floating-point performance, as obtained by instrumentation with the GPTL library. The dynamically scheduled OpenMP parallelization was over blocks of columns (block size was set to 8) in an outer loop, in which the ecRad derived type arguments, and not their array components, are blocked in order to avoid inefficient striding over all columns (unlike ecRad's internal variables, its input/outputs use columns innermost). This reflects IFS use, except that the offline setup does not include preparation of derived types and interpolation to the coarser grid. Computations were repeated 10 times in an outermost loop, and the program was run 5 times, with the fastest result shown.

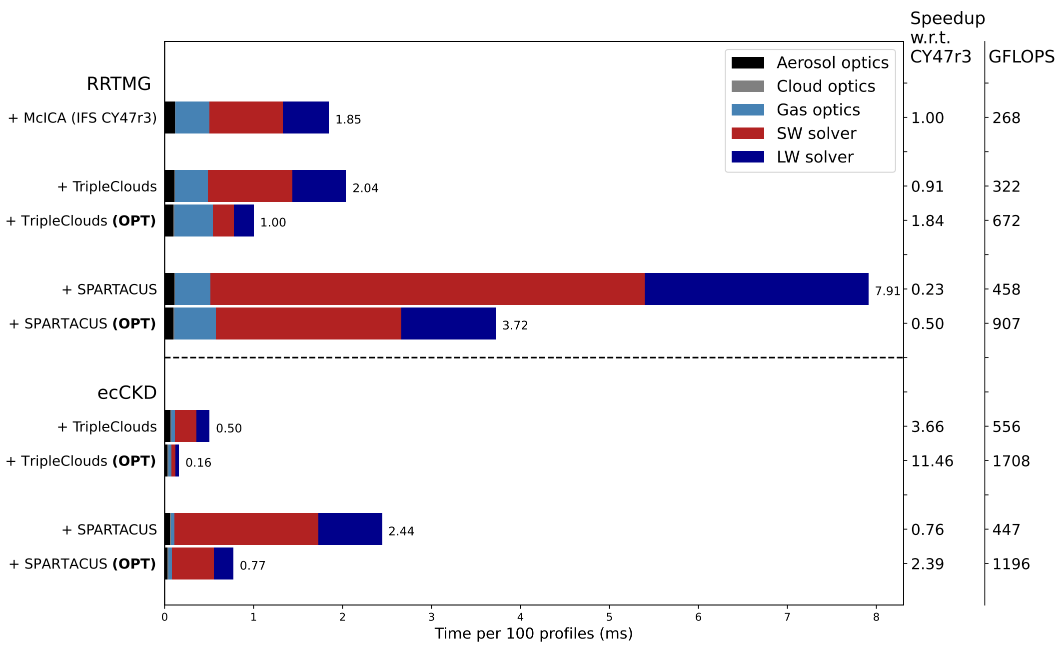


Figure 3: Time per 100 profiles (x-axis) for different configurations of ecRad (y-axis), with colors indicating different components of the radiation scheme. The results are grouped firstly by the choice of gas optics, as this determines the number of g-points. Then, the results are grouped by solver (McICA, TripleClouds and SPARTACUS), and finally (for TripleClouds and SPARTACUS only) by different versions of code, where the runtime profile of the optimized code (OPT) is plotted below the reference. To the right, speedup w.r.t. the configuration of ecRad in IFS cy47r3 (RRTMG+McICA) is shown, followed by an estimate of floating-point performance. The component runtimes are means of per-thread values reported by GPTL, but normalized so they add up to the total time spent in the OpenMP loop (annotated values). Platform: AMD EPYC 7H12, GNU Fortran compiler version 9.3 ('-O3 march=native'), 128 threads=cores.

The optimizations give roughly a three-fold speed-up in the total runtime of ecRad configured with ecCKD and either TripleClouds and SPARTACUS. Optimized TripleClouds with ecCKD is blazingly fast: 100 atmospheric profiles takes only 0.16 milliseconds to compute on the 128-core AMD node, or roughly 20 ms per core. This is nearly 11.5 times faster than the operational IFS radiation (reference ecRad using McICA and RRTMG), achieved mainly by the reduction in spectral resolution (64 versus 252 g-points in total) combined with a much higher floating point performance (1708 GFLOPS versus 268), as opposed to fundamental differences between the solvers (their reference versions have similar runtimes and FLOPS). For SPARTACUS, we find that the optimized code with ecCKD runs more than twice as fast as operational ecRad, and ten times faster than reference SPARTACUS with RRTMG, making cloud 3D effects truly affordable for large-scale dynamical models. Importantly, performance is improved also when using other gas optics schemes, as ecRad configured with RRTMG and either TripleClouds or SPARTACUS is roughly 2 times faster than before.

These speed-ups are a result of a large number changes. To assess the relative impact of different optimizations, our version of offline ecRad can be compiled with three levels of increased refactoring. The runtimes using different versions of the code and two different compilers (including Intel's compiler, which is used for the operational forecast model at ECMWF) are shown in Fig. 4. It can be seen that both high-level refactoring and kernel-level optimizations are important, but the latter are decisive in achieving high performance and getting the full benefit of layer batching, as switching to the new reflectance-transmittance and `expm` kernels (the main hotspots) gives the largest percentage reduction in runtime relative to the previous level of code optimization. Finally, making `ng` a compile time constant in the aerosol optics, gas optics and solvers speeds up radiation computations with 32-term ecCKD models by a further 19-25%, having a larger impact for TripleClouds compiled with the Intel compiler.

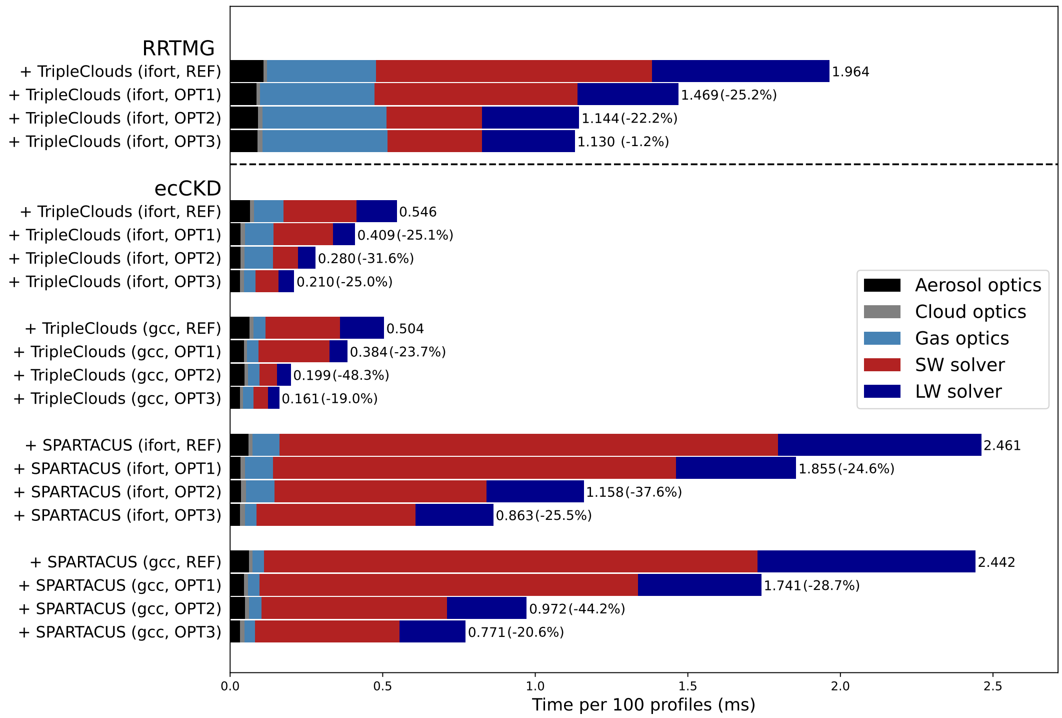


Figure 4: As in Fig. 3, but using increasing levels of code optimization and both the GNU Fortran (labeled "gcc") and Intel Fortran compiler ("ifort", with compiler options '-O2 -march=avx2 -align array64byte -fast-transcedentals -finline-functions ...' reflecting IFS use) included in Intel OneAPI version 2021.4. Annotations again give the total runtime, with the percentage change relative to the previous level shown in brackets. OPT1 = all changes except using the original reflectance-transmittance and matrix exponential kernels, and without declaring `ng` at compile time. OPT2 = OPT1 + optimized main kernels. OPT3 = OPT2 + declaring `ng` at compile time (full optimizations, corresponding to 'OPT' in Fig. 3).

6 Preliminary IFS results with SPARTACUS

We now briefly describe the impact of cloud 3D radiative effects in the IFS by comparing simulations using SPARTACUS and TripleClouds, which is otherwise similar to the SPARTACUS solver but does not compute 3D effects. Firstly, to estimate climate impacts, eight 13-month long (first month is spin-up) coupled atmosphere-ocean simulations using a horizontal grid spacing of around 60 km ($\mathrm{T_{Co}199}$) were performed. These simulations are long enough to capture fast atmospheric and land-surface processes that respond to changes in the radiation scheme, but short enough that the response is not significantly affected by the longer-term changes to ocean circulation. We note that while 3D effects have an overall warming effect on larger scales, they include several processes such as shortwave cloud side interception whose cooling effect can dominate at low solar zenith angles; this could be seen if looking at instantaneous and local 3D effects as opposed to long-term averages (Schafer, 2017), which is our focus here.

Fig. 5 shows a latitude-pressure cross-section of zonal mean temperature differences between the SPARTACUS and TripleClouds runs. In year-long simulations, 3D effects warm almost the entire troposphere by up 0.5 K, the warming being strongest at mid-latitudes, while impacts are neutral below 700 hPa near the equator. We stress that these simulations are too short to capture the ocean response, and 3D effects are likely to have a stronger impact in longer simulations. Interestingly, a visual comparison with Figure 2 of Tian et al. (2013), depicting CMIP5 tropospheric temperature biases against the MERRA reanalysis and a satellite infrared product, suggests a decent match between the SPARTACUS warming pattern and CMIP5 cold biases. Comparing our IFS simulations to ERA5, some existing mid-latitude cold biases were indeed reduced, but SPARTACUS also introduced a warm bias in low latitudes between 200 and 700 hPa, and exacerbated existing IFS stratospheric cold biases near the poles (not shown), where 3D effects have a cooling effect that reaches 1 K over the North Pole. Because operational models are carefully tuned to produce a realistic climate, and contain numerous compensating errors, tuning or revision of other model components is likely required to compensate for the temperature changes caused by SPARTACUS.

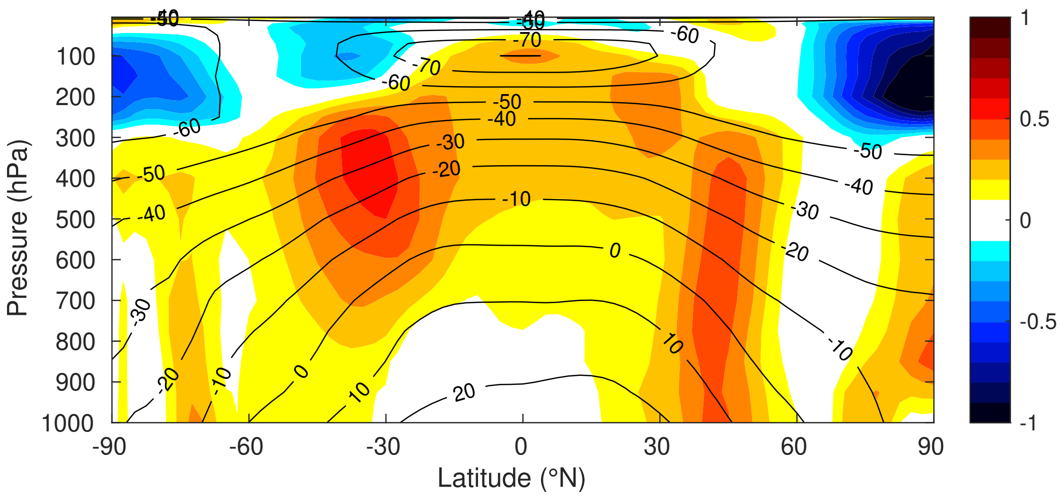


Figure 5: Height-latitude cross section of the zonal mean of the temperature difference between SPARTACUS and TripleClouds runs (where the former includes cloud 3D radiative effects).

Finally, we briefly evaluate the impact on forecast skill using a suite of high-resolution (TCo1279; roughly 9 km horizontal grid spacing) 10-day simulations initialized at consecutive days between 1. June and 31. August 2021 (a total of 92 runs using both TripleClouds and SPARTACUS). It should again be noted that these results are without any model tuning to counteract the tropospheric warming and stratospheric cooling by SPARTACUS. It is therefore not surprising that the SPARTACUS runs exhibit higher root-mean-square-error (RMSE) in temperature aloft due to increased bias, with significant skill degradation in the low latitudes between 100 and 900 hPa (due to warming), and in the northern hemisphere between 10 and 100 hPa (due to cooling). This is not shown, instead we focus on the areas where we find improvement. Most notably, RMSE of 2-metre temperature is reduced by up to 10% in the tropics (Fig. 6). The decrease in RMSE over tropical land was mostly due to reduced cold bias. But encouragingly, the standard deviation of 2-metre temperature is also significantly reduced in the tropics overall (Fig. 7, top row), nearly 1% on average. The inclusion of 3D cloud effects also slightly reduces random errors of low cloud cover in the tropics (Fig. 7, bottom row).

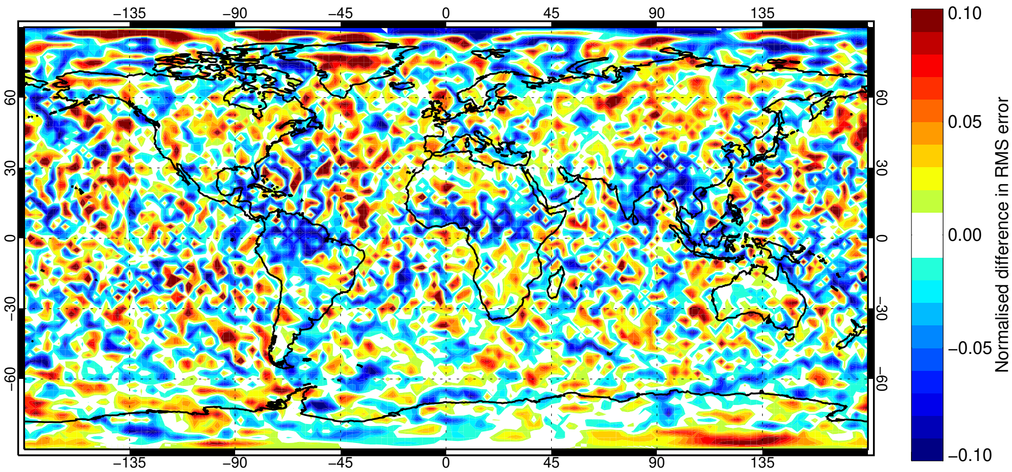


Figure 6: Normalised difference in root-mean-square error in the 7-day forecast of 2-metre temperature between high-resolution simulations using SPARTACUS and Triple-Clouds. The plot shows the average impact on forecast skill across a suite of TCo1279 IFS simulations in June-July-August 2019 (82 samples). Negative numbers (blue colors) indicate improved skill from incorporating 3D effects, up to 10% as shown in dark blue.

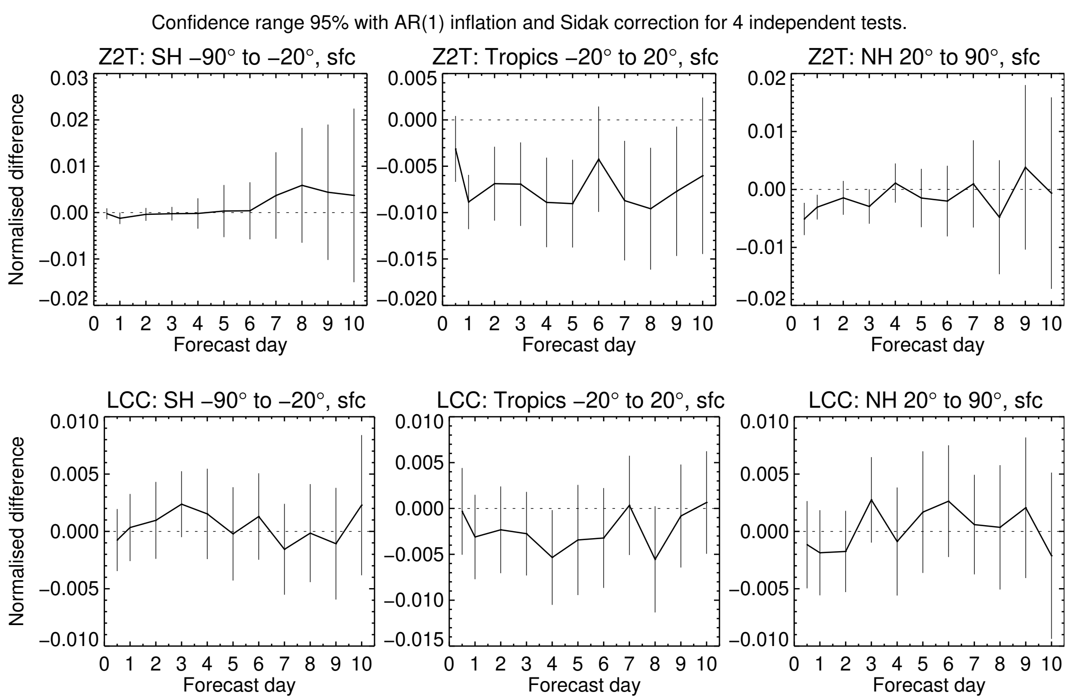


Figure 7: As in Fig. 6, but showing the normalised difference in standard deviation of 2-metre temperature (top row) and low cloud cover (bottom row) by forecast day (x-axis) and region (Southern Hemisphere, Tropics, and Northern Hemisphere). Error bars give the 95% confidence range computed from 82-92 samples.

7 Conclusions

In this work we have refactored the ecRad radiation scheme by using both kernel-level optimizations and higher-level code restructuring to improve performance. Our goal was to capitalize on recent developments in gas optics schemes, namely the new ecCKD tool, which allows the spectral dimension to be reduced considerably (to e.g. 32 g-points in the LW and SW; 64 in total) while retaining accuracy. While speeding up all ecRad solvers, it also decreases floating-point performance due to shortening the innermost vectorized loops over g-points. To address this we restructured the TripleClouds and SPARTACUS solvers to collapse the spectral and vertical dimensions where possible. We also performed many kernel-level optimizations, for instance to improve the efficiency of matrix computations in SPARTACUS, a solver that can compute cloud 3D radiative effects at a relatively low cost. In an effort to make it truly affordable for operational use, we ended up carrying out a thorough performance refactoring of the entire SPARTACUS code. Taken together, our optimizations increase the performance of ecRad configured with ecCKD and either TripleClouds or SPARTACUS by factor of three, and the optimized code is also much faster when using older gas optics schemes with more g-points.

While targeting ECMWF's new supercomputer equipped with AMD Zen 2 CPUs, the high-level code restructuring to expose more parallelism should be useful for any future code porting on GPU, and benefit CPU's with longer vector lengths (via AVX-512 instructions) even more. It should also be applicable to other correlated-k radiation codes, or possibly even other physics parameterizations which include demanding computations conditional to the presence of clouds. The memory layout of ecRad with the spectral dimension innermost, combined with code restructuring to group together cloudy layers in a column and collapsing with the spectral dimension, is likely ideal for performance for 1D radiation schemes, as it allows for sufficiently long vectorized loops (even for spectrally reduced gas optics) to achieve high performance. A memory layout with columns innermost would not allow the compute-intensive computations specific to cloudy layers to be batched in a similar way, and the column batch size may have to be kept small due to memory constraints, reducing SIMD and instruction-level parallelism.

Combining optimized TripleClouds with ecCKD, we obtain a speed-up factor larger than ten relative to the operational radiation scheme in IFS cy47r3 that is based on McICA and RRTMG. This may have implications for emulation studies, which attempt to replace physical schemes with a cheap NN emulator: considering that a low-complexity recurrent NN (which, unlike a faster dense NN, could produce both fluxes and heating rates accurately) was only 4 times faster than a shortwave radiation scheme using 7 times more g-points than ecCKD (Ukkonen, 2022a), the value of using ML for radiation can be questioned - at least for emulation of 1D radiation schemes seeking a speed-up on CPUs. Future studies on this topic should strive to compare NNs to a state-of-the-art radiation scheme, as older codes may be orders-of-magnitudes slower.

With SPARTACUS, we find that the optimized code coupled with ecCKD is more than twice as fast as the operational IFS radiation. To our knowledge, cloud 3D radiative effects have until now been neglected in all weather and climate models due to computational reasons, so this represents a major development. In year-long coupled IFS simulations, SPARTACUS significantly warms the troposphere compared to its fully-1D counterpart (TripleClouds), and these effects are likely to be more pronounced in longer climate simulations, which we leave for future studies to explore. We also performed high-resolution simulations and find that SPARTACUS improves medium-range forecasts of 2-metre temperature and low cloud cover in the tropics. SPARTACUS is still under development to improve some physical assumptions made in the longwave, and we also foresee other opportunities to further increase realism, such as using high-resolution cloud fields to determine SPARTACUS inputs related to cloud sub-grid variability (instead of using a constant value) when running radiation on a coarser grid as is currently done in the IFS.

Appendix A: Kernel-level optimizations

Two-stream kernels

The reference version of ecRad computes the two-stream solutions of reflectance and transmittance (Meador & Weaver, 1980) in double precision, as the underlying equations are numerically sensitive. This issue was also noted by Cotronei and Slawig (2020), who left this kernel in double precision when converting ECHAM radiation to single precision. We found that the code can be made mostly accurate in single precision simply by using a different minimum value for the variable k (Eq. 18 in Meador & Weaver, 1980) in the single precision case (10^{-4} instead of 10^{-12}), but that very rare combinations of the inputs (single-scattering albedo, optical depth and asymmetry factor) could still cause unphysical results in the shortwave computations. This issue was solved by constraining the output variables to prevent that energy could be spuriously created, recognising that the direct beam can either be reflected (`ref_dir`), penetrate unscattered to the base of a layer (`trans__dir`), or penetrate through but be scattered on the way (`trans_dir_diff`) - the rest must be absorbed. This was coded as:

```
ref_dir(jg)        = max(0, min(ref_dir(jg), mu0*(1-trans_dir_dir(jg))))
trans_dir_diff(jg) = max(0, min(trans_dir_diff(jg), mu0*(1-trans_dir_dir(jg)) - ref_dir(jg)))
```

Here, the cosine of the solar zenith angle (`mu0`) is present because ecRad uses a convention that the direct flux is into a plane perpendicular to the sun's direction while diffuse fluxes are into a horizontal plane. After implementing the adjusted threshold and security, the mean absolute difference in SW and LW net fluxes between double and single precision computations with TripleClouds was around 0.001 Wm^{-2} for 10000 columns saved from a high-resolution IFS simulation, and heating rate biases were close to zero.

Both the longwave and shortwave kernels were also sped up by vectorizing the transmittance computation separately by calling the exponential function with an array argument, and conditionals to ensure accurate source functions when the optical depth is low were also placed in a separate post-processing loop, improving performance despite some redundant computations. In the shortwave kernel, conditionals could be removed altogether by borrowing a security to avoid division by zero from RTE+RRTMGP.

SPARTACUS matrix operations

Loop unrolling is a common optimization strategy that compilers can in some cases perform automatically, but if the loop bounds are not known at compile time, the compiler may not know it is advantageous. More involved code patterns may also prevent the compiler from doing this. SPARTACUS uses a matrix exponential solver based on a single precision variant of an optimal scaling and squaring algorithm utilizing Padé approximants (Higham, 2005). The scaling and squaring method involves performing many matrix-matrix multiplications. Because the matrices operated by SPARTACUS are very small, $(\texttt{nreg} \times 3, \texttt{nreg} \times 3) = (9, 9)$ in the shortwave and $(\texttt{nreg} \times 2, \texttt{nreg} \times 2) = (6, 6)$ in the longwave, for performance reasons the matrix-exponential kernel `expm` stores them in the two outer dimensions of 3D arrays, and the fastest-varying spectral dimension is vectorized instead. We found that manually unrolling the innermost of the matrix multiplication loops improved performance on the tested compilers. Redundant computations in `expm` were also identified and removed: in the shortwave (only), many of the matrix-matrix multiplications can exploit not only the sparsity but also some repeated elements in the input matrices, which result in the output matrices also having repeated elements. Given this and the different matrix dimensions, separate LW and SW versions were written for `expm`. The refactoring of the shortwave matrix multiplication kernel is illustrated in Fig. A1.

Similar optimizations were also employed in the many other matrix operations performed by SPARTACUS, such as matrix-vector multiplication, and solving linear systems of equations for a matrix or vector using LU decomposition. For most of these, separate longwave and shortwave kernels were made to allow declaring the inner dimension ($\texttt{ng}_{\mathrm{SW}}$ or $\texttt{ng}_{\mathrm{LW}}$) at compile time, even if other dimensions were identical.

```fortran
! Treat A and B each as n m-by-m square matrices (with the n dimension
! varying fastest) and perform matrix multiplications on all n matrix pairs
mat_x_mat = 0.0_jprb ! Array-wise assignment
mblock = m/3
m2block = 2*mblock
if (i_actual_matrix_pattern == IMatrixPatternShortwave) then
  ! Matrix has a sparsity pattern
  !     (C D E)
  !     (F G H)
  !     (0 0 I)
  ! Do the top-left (C, D, F, G)
  do j2 = 1,m2block ! 1,6
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m2block ! 1,6
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                   + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
  do j2 = m2block+1,m ! 7,9
    ! Do the top-right (E & H)
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                   + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
    ! Do the bottom-right (I)
    do j1 = m2block+1,m ! 7,9
      do j3 = m2block+1,m ! 7,9
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                   + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
else
...
```

$\Downarrow$

```fortran
pure subroutine mat_x_mat_sw_repeats(ng_sw_in, nlev_b, A, B, C)
  integer,    intent(in)                            :: ng_sw_in, nlev_b
  real(jprb), intent(in), dimension(ng_sw*nlev_b,9,9) :: A, B
  real(jprb), intent(out),dimension(ng_sw*nlev_b,9,9) :: C
  integer    :: j1, j2, j22
  !dir$ assume_aligned A:64,B:64,C:64
  ! Input matrices have pattern:
  !   (C          D          E)
  !   (F=-D       G=-C       H)
  !   (0          0          I), where each element is a 3-by-3 matrix
  ! As a result, output matrices have pattern:
  !   (C          D          E)
  !   (F=D        G=C        H)
  !   (0          0          I)
  do j2 = 1,3
    j22 = j2 + 6
    do j1 = 1,6
      ! Do the top-left (C, F)
      ! Unroll innermost matmul loop: more work for each iteration of SIMD loop
      C(:,j1,j2) = A(:,j1,1)*B(:,1,j2) + A(:,j1,2)*B(:,2,j2) + A(:,j1,3)*B(:,3,j2) &
      &          + A(:,j1,4)*B(:,4,j2) + A(:,j1,6)*B(:,6,j2)
      ! Do the top-right (E & H)
      C(:,j1,j22) = A(:,j1,1)*B(:,1,j22) + A(:,j1,2)*B(:,2,j22) + A(:,j1,3)*B(:,3,j22) &
      &      + A(:,j1,4)*B(:,4,j22) + A(:,j1,5)*B(:,5,j22) + A(:,j1,6)*B(:,6,j22) &
      &      + A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
    do j1 = 7,9  ! Do the bottom-right (I)
      C(:,j1,j22) = A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
  end do
  C(:,1:3,4:6) = C(:,4:6,1:3) ! D = F
  C(:,4:6,4:6) = C(:,1:3,1:3) ! G = C
  C(:,7:9,1:6) = 0.0_jprb     ! Lower left corner
```

Figure A1: Reference (top) and optimized (bottom) versions of the matrix-matrix multiplication kernel used in the shortwave matrix exponential computations. The latter unrolls loops and reduces work by exploiting that some matrix elements are repeated. For this performance-critical code, further speedup was gained by data alignment. The Intel compiler reported aligned data access only after declaring `ng_sw` at compile-time.

The other main optimization for `expm` was in the last step of the algorithm, where the matrices across different g-points are individually squared. This section has poor performance because the nature of the scaling and squaring method means that the number of squarings (stored in the N-sized integer array `expo`) varies by g-point, resulting in many temporary copies of small arrays and lack of vectorization. Efficiency was improved by first squaring all the matrices by the minimum `expo`, ensuring vectorization. In the shortwave, performance was also increased (at the cost of code complexity) by squaring groups of matrices, based on array indexing of memory-contiguous matrices that still need to be squared after the first step.

Longwave derivatives

The final step in the longwave solvers is the computation of longwave derivatives, the rate of change of layer broadband upwelling longwave fluxes with respect to surface broadband upwelling flux, which is used for approximate radiation updates in every model column at every model time step (Hogan & Bozzo, 2015). This kernel was relatively expensive for TripleClouds, as it consists of doing `ng` multiplications of very small matrices and vectors (`m`=`nreg`), followed by a multiplication with transmittance (`ng`,`nreg`) at each g-point, and finally a sum over `ng` and `nreg`, at each level. In the expected case of `nreg=3`, the matrix-vector computations, multiplication with transmittance and sum over `nreg` and `ng` were all combined in a single vectorized loop over g-points by inlining the matrix-vector computation and unrolling the three regions (Fig. A2). A similar optimization was done for SPARTACUS where transmittances are 3-D arrays. When combined `ng` being made a compile-time constant, the kernels were sped up by a factor of 5-7, decreasing their share of the total runtime from almost a fifth to only a few percent when using optimized TripleClouds and ecCKD.

```fortran
    ! Initialize the derivatives at the surface; the surface is treated as a
      single
    ! clear-sky layer so we only need to put values in region 1.
    lw_derivatives_g_reg = 0.0_jprb
    lw_derivatives_g_reg(:,1) = flux_up_surf / sum(flux_up_surf)
    lw_derivatives(icol, nlev+1) = 1.0_jprb

    ! Move up through the atmosphere computing the derivatives at each half-level
    do jlev = nlev,1,-1
      ! Compute effect of overlap at half-level jlev+1, yielding
      ! derivatives just above that half-level
      lw_derivatives_g_reg = singlemat_x_vec(ng,ng,nreg,u_matrix(:,:,jlev+1),
     lw_derivatives_g_reg)

      ! Compute effect of transmittance of layer jlev, yielding
      ! derivatives just below the half-level above (jlev)
      lw_derivatives_g_reg = transmittance(:,:,jlev) * lw_derivatives_g_reg

      lw_derivatives(icol, jlev) = sum(lw_derivatives_g_reg)
    end do
```

$\Downarrow$

```fortran
    ...

    ! Move up through the atmosphere computing the derivatives at each half-level
    do jlev = nlev,1,-1
      ! Inline everything in one loop over g-points
      lw_deriv_old  = lw_derivatives_g_reg
      sum_tmp       = 0.0_jprb
      associate(A=>u_matrix(:,:,jlev+1), b=>lw_deriv_old)
        !$omp simd reduction(+:sum_tmp)
        do jg = 1, ng
          ! Compute effect of overlap at half-level jlev+1, yielding derivatives just above that
          ! half-level (matrix-vector multiply)
          ! both inner and outer loop of the matrix loops j1 and j2 unrolled
          ! inner loop:                   j2=1             j2=2             j2=3
          lw_derivatives_g_reg(jg,1) = A(1,1)*b(jg,1) + A(1,2)*b(jg,2) + A(1,3)*b(jg,3)
          lw_derivatives_g_reg(jg,2) = A(2,1)*b(jg,1) + A(2,2)*b(jg,2) + A(2,3)*b(jg,3)
          lw_derivatives_g_reg(jg,3) = A(3,1)*b(jg,1) + A(3,2)*b(jg,2) + A(3,3)*b(jg,3)

          ! Compute effect of transmittance of layer jlev, yielding
          ! derivatives just below the half-level above (jlev)
          lw_derivatives_g_reg(jg,1) = lw_derivatives_g_reg(jg,1) * transmittance(jg,1,jlev)
          lw_derivatives_g_reg(jg,2) = lw_derivatives_g_reg(jg,2) * transmittance(jg,2,jlev)
          lw_derivatives_g_reg(jg,3) = lw_derivatives_g_reg(jg,3) * transmittance(jg,3,jlev)

          sum_tmp = sum_tmp + lw_derivatives_g_reg(jg,1) + lw_derivatives_g_reg(jg,2) + &
                & + lw_derivatives_g_reg(jg,3)
        end do
      end associate

      lw_derivatives(icol, jlev) = sum_tmp
    end do
```

Figure A2: Reference (top) and optimized (bottom) version of the longwave derivatives kernel used by TripleClouds.

Open Research Section

The development version of ecRad 1.6, which includes our configurable optimizations and new gas optics schemes (ecCKD, RRTMGP and RRTMGP-NN), has been uploaded to Zenodo (`https://doi.org/10.5281/zenodo.7148329`) (Ukkonen, 2022b). We expect most of the optimizations to feature in a future official version of ecRad (`https://github.com/ecmwf-ifs/ecrad`).

References

Chevallier, F., Chéruy, F., Scott, N., & Chédin, A. (1998). A neural network approach for a fast and accurate computation of a longwave radiative budget. *Journal of applied meteorology*, *37*(11), 1385–1397.

Cotronei, A., & Slawig, T. (2020). Single-precision arithmetic in echam radiation reduces runtime and energy consumption. *Geoscientific Model Development*, *13*(6), 2783–2804.

Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., Lapillonne, X., Leutwyler, D., … Vogt, H. (2018, May). Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. , *11*(4), 1665–1681. Retrieved from `https://doi.org/10.5194/gmd-11-1665-2018` doi: 10.5194/gmd-11-1665-2018

Goody, R., West, R., Chen, L., & Crisp, D. (1989). The correlated-k method for radiation calculations in nonhomogeneous atmospheres. *Journal of Quantitative Spectroscopy and Radiative Transfer*, *42*(6), 539–550.

Hager, G., & Wellein, G. (2010). *Introduction to high performance computing for scientists and engineers.* CRC Press.

Higham, N. J. (2005). The scaling and squaring method for the matrix exponential revisited. *SIAM Journal on Matrix Analysis and Applications*, *26*(4), 1179–1193.

Hogan, R. J., & Bozzo, A. (2015). Mitigating errors in surface temperature forecasts using approximate radiation updates. *Journal of Advances in Modeling Earth Systems*, *7*(2), 836–853.

Hogan, R. J., & Bozzo, A. (2018). A flexible and efficient radiation scheme for the ecmwf model. *Journal of Advances in Modeling Earth Systems*, *10*(8), 1990–2008. doi: https://doi.org/10.1029/2018MS001364

Hogan, R. J., Fielding, M. D., Barker, H. W., Villefranque, N., & Schäfer, S. A. (2019). Entrapment: An important mechanism to explain the shortwave 3d radiative effect of clouds. *Journal of the Atmospheric Sciences*, *76*(7), 2123–2141.

Hogan, R. J., & Matricardi, M. (2022). A tool for generating fast k-distribution gas-optics models for weather and climate applications. *Journal of Advances in Modeling Earth Systems*, *14*(10), e2022MS003033. Retrieved from `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2022MS003033` (e2022MS003033 2022MS003033) doi: https://doi.org/10.1029/2022MS003033

Hogan, R. J., Quaife, T., & Braghiere, R. (2018). Fast matrix treatment of 3-d radiative transfer in vegetation canopies: Spartacus-vegetation 1.1. *Geoscientific Model Development*, *11*(1), 339–350.

Hogan, R. J., Schäfer, S. A., Klinger, C., Chiu, J. C., & Mayer, B. (2016). Representing 3-d cloud radiation effects in two-stream schemes: 2. matrix formulation and broadband evaluation. *Journal of Geophysical Research: Atmospheres*, *121*(14), 8583–8599.

Kim, P. S., & Song, H.-J. (2022). Usefulness of automatic hyperparameter optimization in developing radiation emulator in a numerical weather prediction model. *Atmosphere*, *13*(5), 721.

Krasnopolsky, V. M., Fox-Rabinovitz, M. S., & Belochitski, A. A. (2008). Decadal climate simulations using accurate and fast neural network emulation of full, longwave and shortwave, radiation. *Monthly Weather Review*, *136*(10), 3683–3695.

Lawrence, B. N., Rezny, M., Budich, R., Bauer, P., Behrens, J., Carter, M., … others (2018). Crossing the chasm: how to develop weather and climate models for next generation computers? *Geoscientific Model Development*, *11*(5), 1799–1821.

Liu, Y., Caballero, R., & Monteiro, J. M. (2020). Radnet 1.0: Exploring deep learning architectures for longwave radiative transfer. *Geoscientific Model Development*, *13*(9), 4399–4412.

Meador, W., & Weaver, W. (1980). Two-stream approximations to radiative transfer in planetary atmospheres: A unified description of existing methods and a new improvement. *Journal of Atmospheric Sciences*, *37*(3), 630–643.

Michalakes, J., Iacono, M. J., & Jessup, E. R. (2016). Optimizing weather model radiative transfer physics for intel's many integrated core (mic) architecture.

Parallel Processing Letters, *26*(04), 1650019.

Mlawer, E. J., Taubman, S. J., Brown, P. D., Iacono, M. J., & Clough, S. A. (1997). Radiative transfer for inhomogeneous atmospheres: Rrtm, a validated correlated-k model for the longwave. *Journal of Geophysical Research: Atmospheres*, *102*(D14), 16663–16682.

Pal, A., Mahajan, S., & Norman, M. R. (2019). Using deep neural networks as cost-effective surrogate models for super-parameterized e3sm radiative transfer. *Geophysical Research Letters*, *46*(11), 6069–6079. doi: https://doi.org/10.1029/2018GL081646

Pincus, R., Mlawer, E. J., & Delamere, J. S. (2019). Balancing accuracy, efficiency, and flexibility in radiation calculations for dynamical models. *Journal of Advances in Modeling Earth Systems*, *11*(10), 3074–3089.

Roh, S., & Song, H.-J. (2020). Evaluation of neural network emulations for radiation parameterization in cloud resolving model. *Geophysical Research Letters*, *47*(21), e2020GL089444. doi: https://doi.org/10.1029/2020GL089444

Schafer, S. A. (2017). *What is the global impact of 3d cloud-radiation interactions?* (Unpublished doctoral dissertation). University of Reading.

Shonk, J. K., & Hogan, R. J. (2008). Tripleclouds: An efficient method for representing horizontal cloud inhomogeneity in 1d radiation schemes by using three regions at each height. *Journal of Climate*, *21*(11), 2352–2370.

Song, H.-J., & Roh, S. (2021). Improved weather forecasting using neural network emulation for radiation parameterization. *Journal of Advances in Modeling Earth Systems*, *13*(10), e2021MS002609. doi: https://doi.org/10.1029/2021MS002609

Tian, B., Fetzer, E. J., Kahn, B. H., Teixeira, J., Manning, E., & Hearty, T. (2013). Evaluating cmip5 models using airs tropospheric air temperature and specific humidity climatology. *Journal of Geophysical Research: Atmospheres*, *118*(1), 114–134.

Ukkonen, P. (2022a). Exploring pathways to more accurate machine learning emulation of atmospheric radiative transfer. *Journal of Advances in Modeling Earth Systems*, e2021MS002875. doi: 10.1029/2021MS002875

Ukkonen, P. (2022b, October). *Optimized version of the ecRad radiation scheme with new RRTMGP-NN gas optics [Dataset].* Zenodo. Retrieved from `https://doi.org/10.5281/zenodo.7852526` doi: 10.5281/zenodo.7852526

Ukkonen, P., Pincus, R., Hogan, R. J., Nielsen, K. P., & Kaas, E. (2020). Accelerating radiation computations for dynamical models with targeted machine learning and code optimization. *Journal of Advances in Modeling Earth Systems*, *12*(12), e2020MS002226. doi: https://doi.org/10.1029/2020MS002226

Figure 1.

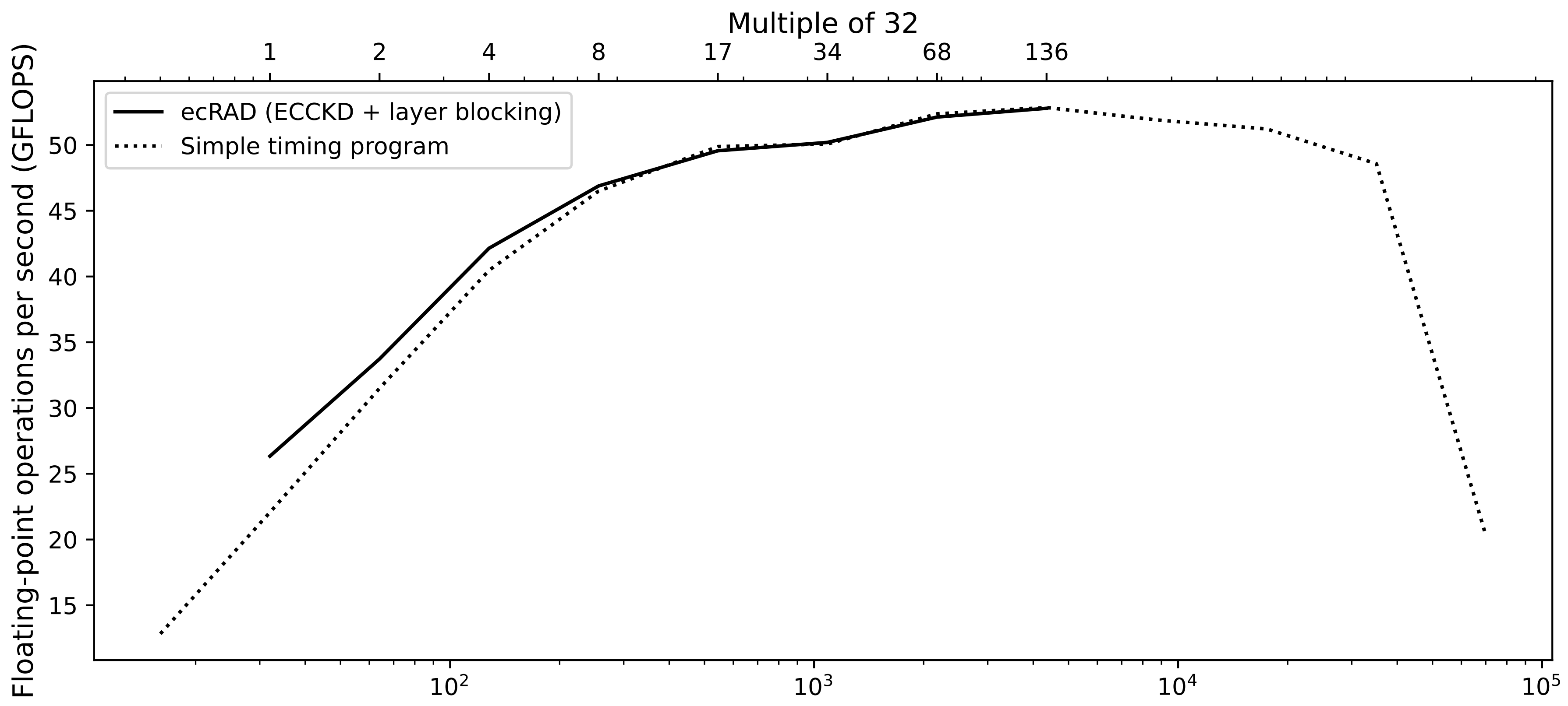

Multiple of 32
1
2
4
8
17
34
68
136
ecRAD (ECCKD + layer blocking)
Simple timing program
Floating-point operations per second (GFLOPS)
50
45
40
35
30
25
20
15
10^2
10^3
10^4
10^5

Figure 2 (Latex-generated code listing).

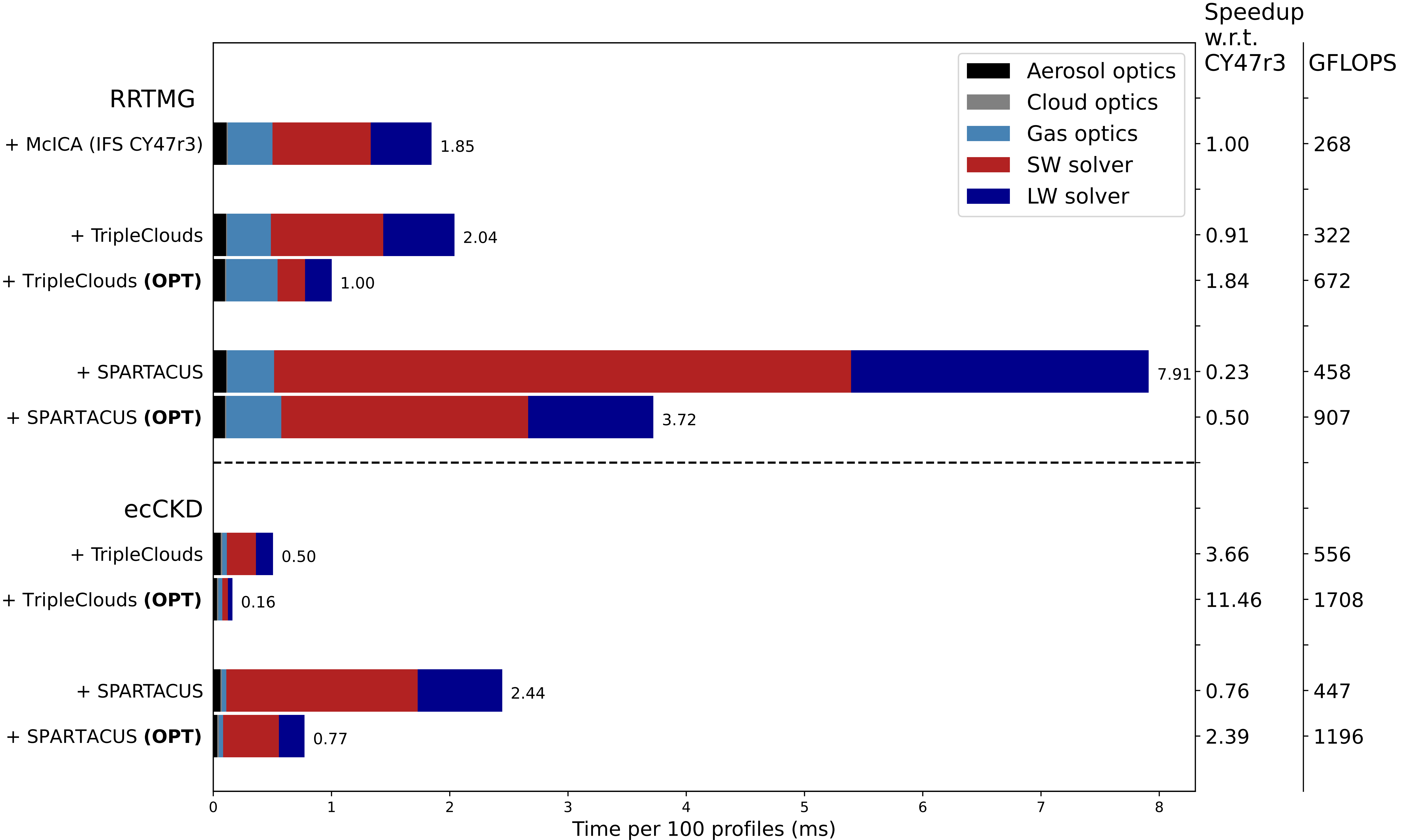
```fortran
do jlev = 1,nlay ! Start at top-of-atmosphere
 nreg = nregions
 if (is_clear_sky_layer(jlev) nreg = 1

 do jreg = 1,nreg ! Loop over relevant regions (only 1 if layer is clear-sky)
  if (jreg == 1) then ! optical properties are equal to clear-sky values
   optical_depth_tot  =  optical_depth(:,jlev,jcol)
   ssa_tot = ssa(:,jlev,jcol)
   g_tot   =   g(:,jlev,jcol)
  else
   do jg = 1,ng ! loop over g-points
    ! Cloudy-sky optical properties from band-wise cloud values and g-point-wise clear-sky values
    optical_depth_tot(jg) =  optical_depth(jg,jlev,jcol) + ...
    ...
   end do
  end if

  call calc_two_stream_gammas_sw(ng,  mu0, ssa_tot, g_tot,  gamma1, gamma2, gamma3)
  call calc_reftrans_sw(ng, mu0, optical_depth_tot, ssa_tot, gamma1, gamma2, gamma3, &
   &  reflectance(:,jreg,jlev), transmittance(:,jreg,jlev), & ! outputs
   &  ref_dir(:,jreg,jlev), trans_dir_diff(:,jreg,jlev), trans_dir_dir(:,jreg,jlev)) ! outputs
 end do
end do
```

⇓

```fortran
! Computations for clear-sky region as a separate step: collapse the two inner dimensions
call calc_reftrans_sw_opt(ng*nlay, mu0, optical_depth(:,:,jcol), ssa(:,:,jcol), g(:,:,jcol), &
 &  reflectance_clear, transmittance_clear, ref_dir_clear, trans_dir_diff_clear, trans_dir_dir_clear)

! Cloudy computations: start at top-of-atmosphere and find first cloudy layer, if one exists
any_clouds_below = .false.
jtop = findloc(is_clear_sky_layer(1:nlay), .false., dim=1)
if (jtop>0) any_clouds_below = .true.

do while (any_clouds_below)
 ! Find the bottom of this cloud
 jbot = ...
 nlay_cloud = jbot - jtop + 1
 allocate(optical_depth_tot_cloudy(ng,2:nreg,jtop:jbot), ssa_tot_cloudy(ng,2:nreg,jtop:jbot), &
    & g_tot_cloudy(ng,2:nreg,jtop:jbot))

 do jlev = jtop, jbot
  do jreg = 2, nregions ! = 3
   do jg = 1,ng
    ! Spectral cloudy-sky optical properties from band-wise cloud values and spectral clear-sky values
    optical_depth_tot_cloudy(jg,jreg,jlev) = ...
    ...
   end do
  end do
 end do

 call calc_reftrans_sw_opt(ng*2*nlay_cloud, & ! g-points * cloudy regions * adjacent cloudy layers
  &  mu0, optical_depth_tot_cloudy, ssa_tot_cloudy, g_tot_cloudy, &
  &  reflectance(:,:,jtop:jbot), transmittance(:,:,jtop:jbot), & ! outputs
  &  ref_dir(:,:,jtop:jbot), trans_dir_diff(:,:,jtop:jbot), trans_dir_dir(:,:,jtop:jbot)) ! outputs

 deallocate(optical_depth_tot_cloudy, ssa_tot_cloudy, g_tot_cloudy)

 ! Does another cloudy layer exist? If not, set logical to false to exit "while"
 if (jbot== nlay) any_clouds_below=.false. ! surface reached

 if (any(.not. is_clear_sky_layer(jbot+1:nlay))) then
  ! find the top of the new cloud
  jtop = ...
 else
  any_clouds_below=.false.
 end if
end do
```

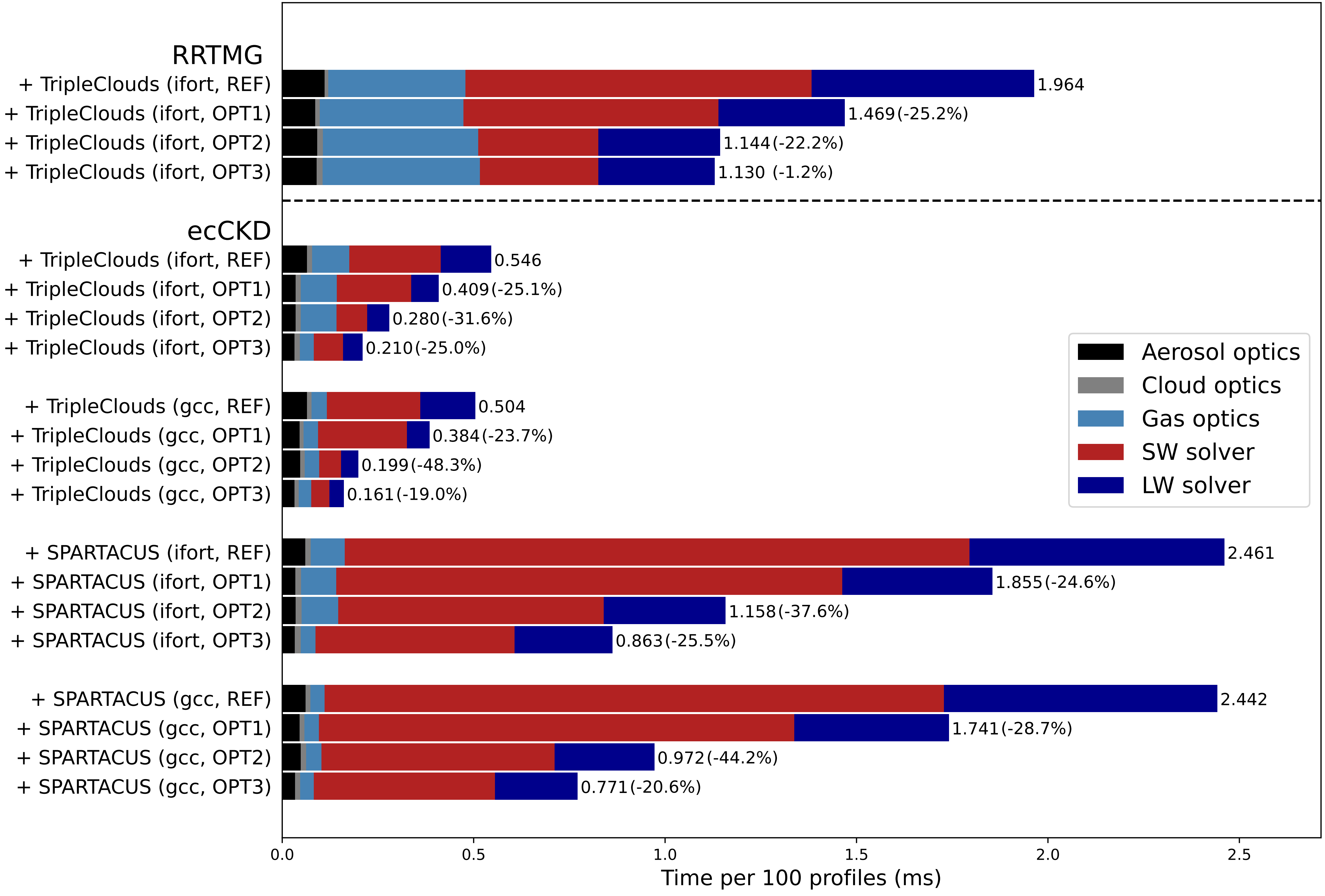
Figure 1: Refactoring of TripleClouds-SW. In addition to optimizing and fusing kernels, in the new code (bottom) the reflectance-transmittance computations are performed in a batched manner for multiple layers by collapsing the spectral and vertical dimensions.

Figure 3.



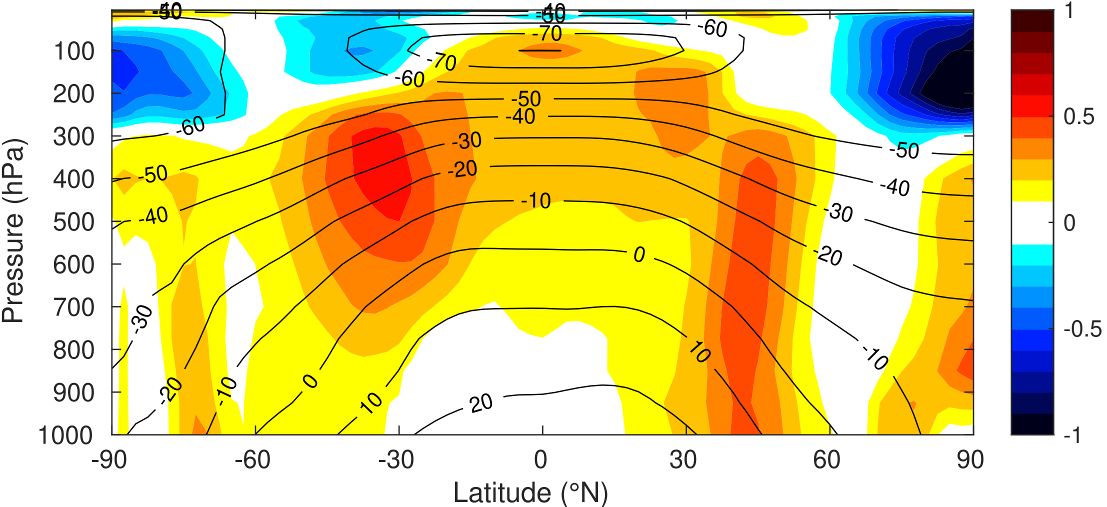
Speedup w.r.t. CY47r3
GFLOPS
Aerosol optics
Cloud optics
Gas optics
SW solver
LW solver
RRTMG
+ McICA (IFS CY47r3)
1.85
1.00
268
+ TripleClouds
2.04
0.91
322
+ TripleClouds **(OPT)**
1.00
1.84
672
+ SPARTACUS
7.91
0.23
458
+ SPARTACUS **(OPT)**
3.72
0.50
907
ecCKD
+ TripleClouds
0.50
3.66
556
+ TripleClouds **(OPT)**
0.16
11.46
1708
+ SPARTACUS
2.44
0.76
447
+ SPARTACUS **(OPT)**
0.77
2.39
1196
0
1
2
3
4
5
6
7
8
Time per 100 profiles (ms)

Figure 4.



RRTMG
+ TripleClouds (ifort, REF)
+ TripleClouds (ifort, OPT1)
+ TripleClouds (ifort, OPT2)
+ TripleClouds (ifort, OPT3)
1.964
1.469 (-25.2%)
1.144 (-22.2%)
1.130 (-1.2%)
ecCKD
+ TripleClouds (ifort, REF)
+ TripleClouds (ifort, OPT1)
+ TripleClouds (ifort, OPT2)
+ TripleClouds (ifort, OPT3)
0.546
0.409 (-25.1%)
0.280 (-31.6%)
0.210 (-25.0%)
+ TripleClouds (gcc, REF)
+ TripleClouds (gcc, OPT1)
+ TripleClouds (gcc, OPT2)
+ TripleClouds (gcc, OPT3)
0.504
0.384 (-23.7%)
0.199 (-48.3%)
0.161 (-19.0%)
+ SPARTACUS (ifort, REF)
+ SPARTACUS (ifort, OPT1)
+ SPARTACUS (ifort, OPT2)
+ SPARTACUS (ifort, OPT3)
2.461
1.855 (-24.6%)
1.158 (-37.6%)
0.863 (-25.5%)
+ SPARTACUS (gcc, REF)
+ SPARTACUS (gcc, OPT1)
+ SPARTACUS (gcc, OPT2)
+ SPARTACUS (gcc, OPT3)
2.442
1.741 (-28.7%)
0.972 (-44.2%)
0.771 (-20.6%)
Aerosol optics
Cloud optics
Gas optics
SW solver
LW solver
0.0
0.5
1.0
1.5
2.0
2.5
Time per 100 profiles (ms)

Figure 5.



Pressure (hPa)
100
200
300
400
500
600
700
800
900
1000
Latitude (°N)
-90
-60
-30
0
30
60
90
1
0.5
0
-0.5
-1
-70
-60
-50
-40
-30
-20
-10
0
10
20

Figure 6.

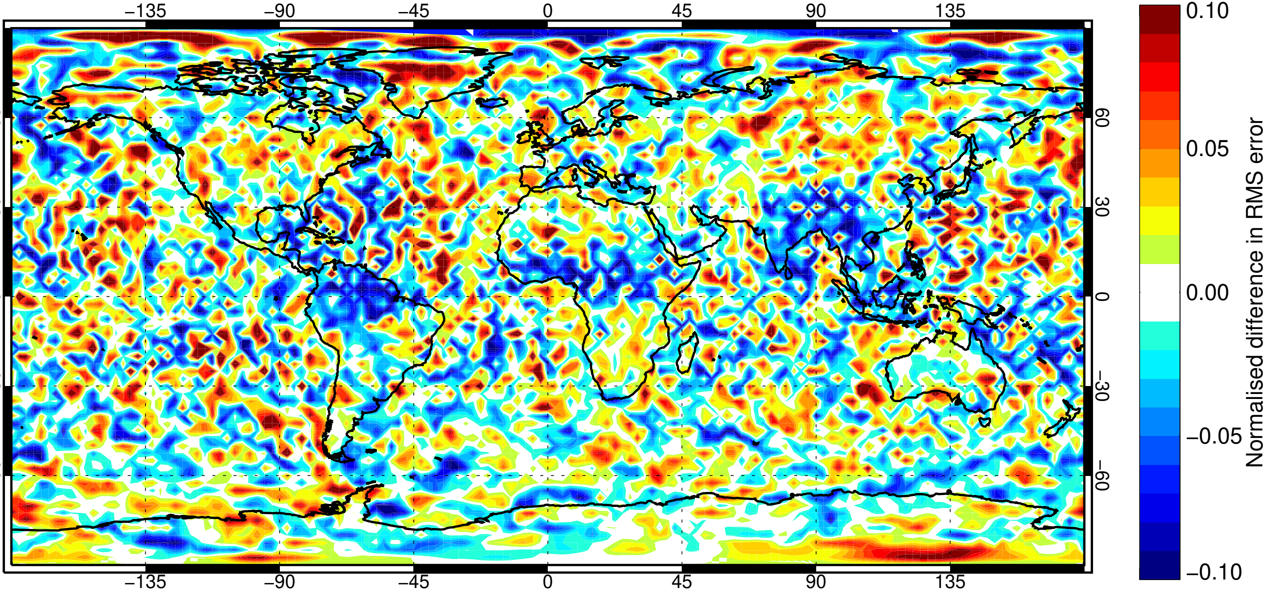
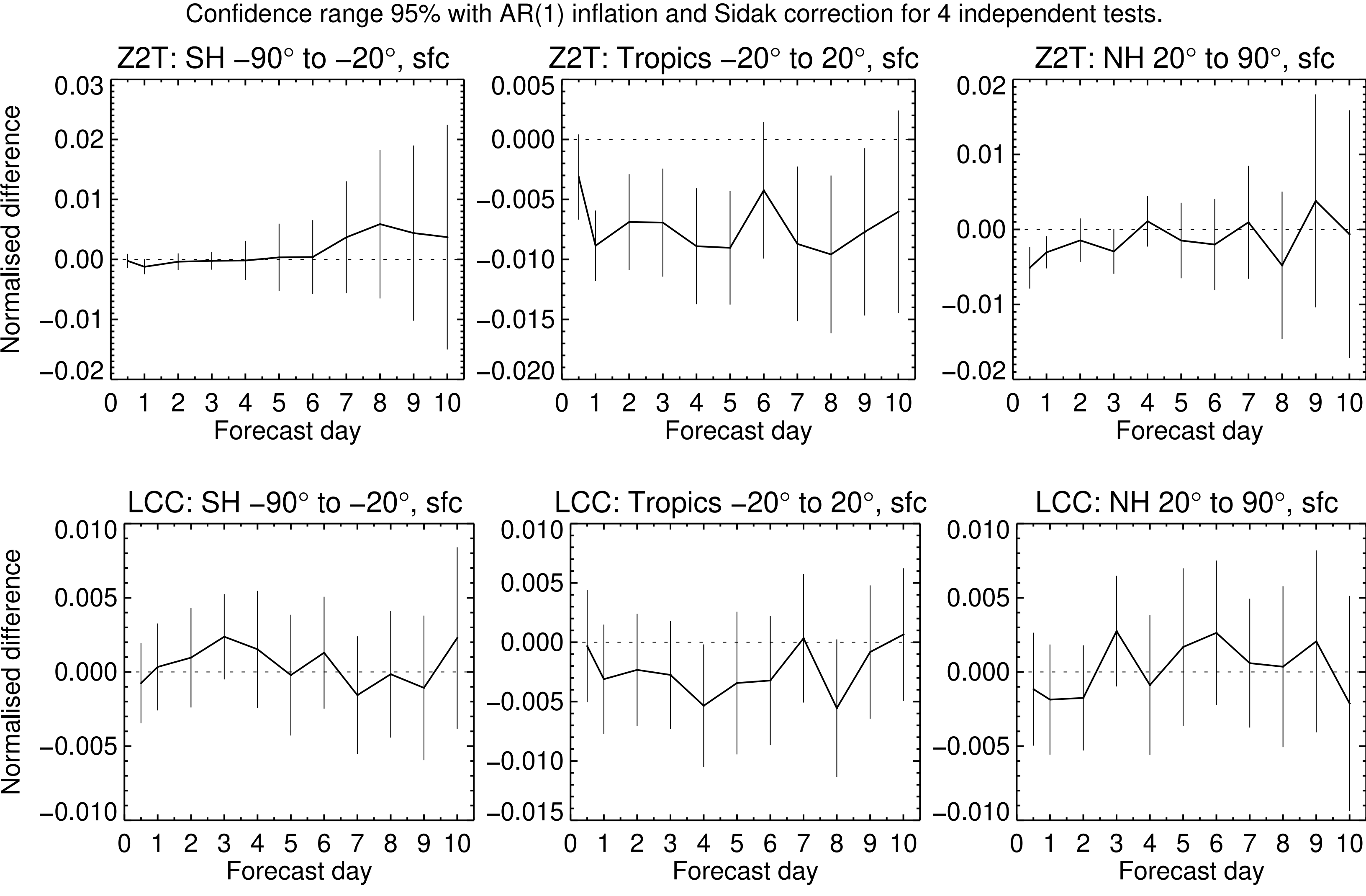

−135
−90
−45
0
45
90
135
60
30
0
−30
−60
0.10
0.05
0.00
−0.05
−0.10
Normalised difference in RMS error

Figure 7.



Confidence range 95% with AR(1) inflation and Sidak correction for 4 independent tests.
Z2T: SH –90° to –20°, sfc
Z2T: Tropics –20° to 20°, sfc
Z2T: NH 20° to 90°, sfc
LCC: SH –90° to –20°, sfc
LCC: Tropics –20° to 20°, sfc
LCC: NH 20° to 90°, sfc
Normalised difference
Forecast day
0.03
0.02
0.01
0.00
–0.01
–0.02
0.005
0.000
–0.005
–0.010
–0.015
–0.020
0.010
0 1 2 3 4 5 6 7 8 9 10

Figure A1 (Latex-generated code listing).

```fortran
! Treat A and B each as n m-by-m square matrices (with the n dimension
! varying fastest) and perform matrix multiplications on all n matrix pairs
mat_x_mat = 0.0_jprb ! Array-wise assignment
mblock = m/3
m2block = 2*mblock
if (i_actual_matrix_pattern == IMatrixPatternShortwave) then
  ! Matrix has a sparsity pattern
  !     (C D E)
  !     (F G H)
  !     (0 0 I)
  ! Do the top-left (C, D, F, G)
  do j2 = 1,m2block ! 1,6
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m2block ! 1,6
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                   + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
  do j2 = m2block+1,m ! 7,9
    ! Do the top-right (E & H)
    do j1 = 1,m2block ! 1,6
      do j3 = 1,m
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                   + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
    ! Do the bottom-right (I)
    do j1 = m2block+1,m ! 7,9
      do j3 = m2block+1,m ! 7,9
        mat_x_mat(1:ng3D,j1,j2) = mat_x_mat(1:ng3D,j1,j2) &
             &                   + A(1:ng3D,j1,j3)*B(1:ng3D,j3,j2)
      end do
    end do
  end do
else
...
```

⇓

```fortran
pure subroutine mat_x_mat_sw_repeats(ng_sw_in, nlev_b, A, B, C)
  integer,    intent(in)                           :: ng_sw_in, nlev_b
  real(jprb), intent(in), dimension(ng_sw*nlev_b,9,9) :: A, B
  real(jprb), intent(out),dimension(ng_sw*nlev_b,9,9) :: C
  integer    :: j1, j2, j22
  !dir$ assume_aligned A:64,B:64,C:64
  ! Input matrices have pattern:
  !   (C          D          E)
  !   (F=-D       G=-C       H)
  !   (0          0          I), where each element is a 3-by-3 matrix
  ! As a result, output matrices have pattern:
  !   (C          D          E)
  !   (F=D        G=C        H)
  !   (0          0          I)
  do j2 = 1,3
    j22 = j2 + 6
    do j1 = 1,6
      ! Do the top-left (C, F)
      ! Unroll innermost matmul loop: more work for each iteration of SIMD loop
      C(:,j1,j2) = A(:,j1,1)*B(:,1,j2) + A(:,j1,2)*B(:,2,j2) + A(:,j1,3)*B(:,3,j2) &
      &          + A(:,j1,4)*B(:,4,j2) + A(:,j1,6)*B(:,6,j2)
      ! Do the top-right (E & H)
      C(:,j1,j22) = A(:,j1,1)*B(:,1,j22) + A(:,j1,2)*B(:,2,j22) + A(:,j1,3)*B(:,3,j22) &
      &      + A(:,j1,4)*B(:,4,j22) + A(:,j1,5)*B(:,5,j22) + A(:,j1,6)*B(:,6,j22) &
      &      + A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
    do j1 = 7,9  ! Do the bottom-right (I)
      C(:,j1,j22) = A(:,j1,7)*B(:,7,j22) + A(:,j1,8)*B(:,8,j22) + A(:,j1,9)*B(:,9,j22)
    end do
  end do
  C(:,1:3,4:6) = C(:,4:6,1:3) ! D = F
  C(:,4:6,4:6) = C(:,1:3,1:3) ! G = C
  C(:,7:9,1:6) = 0.0_jprb     ! Lower left corner
```

Figure 1: Reference (top) and optimized (bottom) versions of the matrix-matrix multiplication kernel used in the shortwave matrix exponential computations. The latter unrolls loops and reduces work by exploiting that some matrix elements are repeated. For this performance-critical code, further speedup was gained by data alignment. The Intel compiler reported aligned data access only after declaring `ng_sw` at compile-time.

Figure A2 (latex-generated code listing).

```fortran
! Initialize the derivatives at the surface; the surface is treated as a
 single
! clear-sky layer so we only need to put values in region 1.
lw_derivatives_g_reg = 0.0_jprb
lw_derivatives_g_reg(:,1) = flux_up_surf / sum(flux_up_surf)
lw_derivatives(icol, nlev+1) = 1.0_jprb

! Move up through the atmosphere computing the derivatives at each half-level
do jlev = nlev,1,-1
  ! Compute effect of overlap at half-level jlev+1, yielding
  ! derivatives just above that half-level
  lw_derivatives_g_reg = singlemat_x_vec(ng,ng,nreg,u_matrix(:,:,jlev+1),
 lw_derivatives_g_reg)

  ! Compute effect of transmittance of layer jlev, yielding
  ! derivatives just below the half-level above (jlev)
  lw_derivatives_g_reg = transmittance(:,:,jlev) * lw_derivatives_g_reg

  lw_derivatives(icol, jlev) = sum(lw_derivatives_g_reg)
end do
```

⇓

```fortran
...

! Move up through the atmosphere computing the derivatives at each half-level
do jlev = nlev,1,-1
  ! Inline everything in one loop over g-points
  lw_deriv_old  = lw_derivatives_g_reg
  sum_tmp       = 0.0_jprb
  associate(A=>u_matrix(:,:,jlev+1), b=>lw_deriv_old)
    !$omp simd reduction(+:sum_tmp)
    do jg = 1, ng
      ! Compute effect of overlap at half-level jlev+1, yielding derivatives just above that
      ! half-level (matrix-vector multiply)
      ! both inner and outer loop of the matrix loops j1 and j2 unrolled
      ! inner loop:                   j2=1               j2=2               j2=3
      lw_derivatives_g_reg(jg,1) = A(1,1)*b(jg,1) + A(1,2)*b(jg,2) + A(1,3)*b(jg,3)
      lw_derivatives_g_reg(jg,2) = A(2,1)*b(jg,1) + A(2,2)*b(jg,2) + A(2,3)*b(jg,3)
      lw_derivatives_g_reg(jg,3) = A(3,1)*b(jg,1) + A(3,2)*b(jg,2) + A(3,3)*b(jg,3)

      ! Compute effect of transmittance of layer jlev, yielding
      ! derivatives just below the half-level above (jlev)
      lw_derivatives_g_reg(jg,1) = lw_derivatives_g_reg(jg,1) * transmittance(jg,1,jlev)
      lw_derivatives_g_reg(jg,2) = lw_derivatives_g_reg(jg,2) * transmittance(jg,2,jlev)
      lw_derivatives_g_reg(jg,3) = lw_derivatives_g_reg(jg,3) * transmittance(jg,3,jlev)

      sum_tmp = sum_tmp + lw_derivatives_g_reg(jg,1) + lw_derivatives_g_reg(jg,2) + &
          & + lw_derivatives_g_reg(jg,3)
    end do
  end associate

  lw_derivatives(icol, jlev) = sum_tmp
end do
```

Figure 1: Reference (top) and optimized (bottom) version of the longwave derivatives kernel used by TripleClouds.