Implementation of an Eulerian-Eulerian Optimized Emission Based Reciprocity Monte Carlo, for multiphase participating media, on GPU

S. Silvestri

source code

- `main_comp.f` → main fortran routine with parallelization strategy
- `mc_gpu.cu` → main c routine with the communication function and all CUDA functions
- `read_tables.cpp` → handling the input of radiative properties coming from narrow-band cK (fluid) and mie theory (particles)
- `spline.cpp` → interpolation of quantities for adaptive mesh coarsening
- `memory_copy.cu` → c program handling all memory copy from CPU to GPU

header files

- `definitions.h` → definition of all needed objects for CPU and GPU (grid, properties, emissions, rays, etc)
- `device_functions.h` → definition of GPU CUDA functions
- `functions.h` → definition of CPU C++ functions
- `memory.h` → definition of CPU CUDA funtions
- `shared.h` → definition of shared extern variables
- `NarrowBand.h` → details of the properties data from NB-cK and MIE
- `param.h` → simulation parameters and all macro definitions

Input files

- `../distr/tables/` → folder with the location of input files for fluid
- `NarrowBand%d.txt` → with `%d = 0 - (nB-1)`, files containing absorption coefficient of fluid
- `prob_new2.txt` → contains the probability of emission at a certain wavelength for all bands for all quadrature points for all temperatures for the fluid
- `planck-mean.txt` → planck mean absorption coefficient of the fluid vs temperature
- `../distr/tables/particles` → folder with the location of input files for particles
- `NarrowBand%d.txt` → with `%d = 0 - (nB-1)`, absorption coefficient of particles, for single band and all temperatures
- `prob.txt` → contains the probability of emission at a certain wavelength for all bands for all temperatures for particles (particles have no quadrature)
- `planck-mean.txt` → planck mean absorption coefficient of the particles vs temperature

description of files with functions

`main_comp.f`
This file containes the communication between the fortran piece of the program and the C piece of the program. The parallelization is handled with the creation of a new communicator `MPI_COMM_NODE`. The new communicator groups all of the cores that belong to a node, based on `p_row`. The first core of each node `node_rank=0` gathers all the relevant input for the MC routine (Temperature, particle temperature, particle concentration) from the whole domain and calls `MC_GPU` (that in `C` language is called `mc_gpu_`). This routine gives back the radiative power source for fluid and particles (`Qr, Qrp`) and the associated variances (`Vr, Vrp`). These variables are known only by `node_rank=0` of `MPI_COMM_NODE` in node domain `(imax,jmax/p_row,kmax)`.
They are then redistributed to each core `(imax,jmax/p_row,kmax/p_col)`. Note that `p_col` division is not seen be `MC_GPU` since the GPU computes the

whole `k` direction (if there are x GPUs in one node, they can compute `kmax/x` each, this is defined in the `C` side, see `param.h`).

`mc_gpu.cu`
This is the most important file of the program. it contains:

- `mc_gpu_` this function is the "main" of the `C` side, it takes the input from `main_comp.f` and calls interpolation functions to create the adaptive mesh. Then it calls the read table routines to read the radiative properties, and sets the GPU memory with the memory copy routines. Finally it calls the GPU kernels to perform the MC calculations and returns the results to `main_comp.f`.

- `kernel_fluid` the "main" routine on the GPU side. The routine is divided in blocks and threads, found in `<<<>>>`, and is parallelized in streams, as well as in number of GPUs per node (all found in `param.h`). While blocks and threads are intrinsic in one kernel call, parallelization in number of GPU and streams simply consists in sequential asynchronous kernel calls in two nested for loops. Each thread handles one cell at the time. The first loop in `kernel_fluid` is a grid stride loop that spans the whole calculated domain `(imax*jmax/p_row*kmax/num_gpu)`. Each thread, therefore, calculates sequentially the latter amount of cells divided by `blockDim.x*gridDim.x` (set in `param.h`). The indices of the cell are retrieved based on the thread index and stored in `ray.i,j,k`. Here also the ray temperature `ray.Ti[0,1]` and the ray power `Ibmax[0,1]` are calculated.
  The second nested loop is the variance loop: within each cell the radiative power will be calculated `nVar` times, to calculate the associated statistical variance.
  The third nested loop is the photon loop, where each bundle's (`Beam ray`) parameters (wavenumber `wvc[0,1]` and direction `ray.sx,sy,sz`) are set. To the ray, it is associated an initial transmissivity of 1. The ray will stop when, either `ray.tra[0&1]<toll` or the ray hits a boundary. In the particle+fluid version, each ray contains an array of 2 for each variable (except for direction and position that are the same). The index `[0]` stands for a variable associated with the fluid radiative source, while in `[1]`, the variables associated with particles radiative source are contained.
  Random numbers for random number relations are drawn from the curand

library (functions `curand_init` and `curand_uniform`). To reduce register pressure, the variable containing the state of the random sequence `state`, has been located in the shared memory. Finally the nested while loop calls the marching routing and, based on the possible events occurring (summarized in the output variable `sca_flag`), it performes the required action. 4 scenarios are possible:

  - The ray reached a boundary or `ray.tra[0&1]<toll` (`ray.tra=0` was set in the marching routing, the while loop is exited, one radiative source calculation is finished)
  - A scattering event took place (scatter the ray and refresh the scattering probability)
  - The mesh has to be coarsened (coarsen mesh)
  - A scattering event took place and the mesh has to be coarsened

- `kernel_find` runs only if the sorting of narrowbands is active (i.e., if `srt==1` in `param.h`). It picks the bands associated with all the photons that will be launched in one variance calculation and counts the occurrence of each band. The results are stored in `count->nb_cnt[nb]` and `count->g_cnt[g][nb]` (not beneficial when fluid+particle). Reduces heavily the efficiency of the code when a large number of samples are required (i.e., large `nVar`).

- `march_ray` Marching algorithm that effectively calculates the radiative power `De_OERMc[0,1]` (`[0]` for fluid and `[1]` for particles). Initially a check is performed to verify that, after a scattering event, a ray is not located on a cell boundary while moving in a concorde direction (it would create an error). If this is the case, the index is adjusted accordingly. If, during this check, the ray is sitting on a black boundary it is terminated. If it is sitting on a periodic boundary, the `ray.xp,yp,zp` locations are adjusted accordingly. If at least one of the transmissivities is `>0` then the ray is marched, the distances to the boundaries in `i,j,k` directions (`dsx`,`dsy`,`dsz`) are calculated, and the smaller between these and `sca_tot-ray->sca` is chosen (`sca_tot` is the total distance until scattering, calculated from random number relations and the scattering coefficient, while `ray->sca` is the cumulative distance travelled by the ray). The `xp,yp,zp` locations of the ray are adjusted based on the chosen `ds`. Temperatures are fetched from the texture

memory as well as absorption coefficient of particle and fluid (vector of 2 due to the different wavelength of particle and fluid ray).
The two different radiative sources are calculated based on the reciprocity principle, transmissivity is updated, if it is `<toll` it is set to zero and the remaining energy is descharged all in the present cell.
If a scattering event is not occurring, the indices of the ray `ic,jc,kc` are modified according to the direction taken by the ray. If a boundary is reached, boundary conditions are applied (`ray->tra` is set to `0` in case of a black boundary). At the end of the while loop the counter that counts the number of steps in one grid (the maximum for each grid is `maxs[]`, see `param.h`) is increased. Out of the loop the occurred event (absorption by boundary, depletion of the ray, end of the grid, scattering event) is flagged in order to send the correct instructions to `kernel_fluid`.

- `emiss_ang` calculates the angle of emission and the associated direction vector, based on isotropic emission (all directions same probability)
- `scatt_ang` calculates the scattering angles and the associated directions, based on the scattering phase function probability calculated from MIE theory and contained in `prob_A`
- `find_band` finds the band of emission and quadrature point of the fluid based on random number relations and binary search
- `find_band_p` finds the band of emission of the particles based on random number relations and binary search
- `wave_find` runs only if the narrowband sorting is active (i.e, `srt==1`). This routine finds the associated band based on the reordering of the wavenumbers and on the count performed by `kernel_find`
- `I_black` given temperature and wavenumber (in cm^{-1}), calculates the balckbody intensity
- all routines and variables finishing with capital `S`, in `mc_gpu.cu`, refer to solar monte carlo and are not yet used (still requires a parallelization strategy)

`read_tables.cpp`

This file contains the routines that read in the input files which contain the spectral data of fluid and particles, calculated from NB-cK and MIE routines (separate developed programs).

- `sort_idx` sorting of the absorption index form lower to higher value, storing the result in `narrBand[i].idx`
- `readT` reading in all spectral properties of fluid (non-scattering), comprising of: `kP` Planck absorption coefficient, `narrBand[i].kq[j][g]` absorption coefficient, `prob[i][j]` band probability and `probg[i][j][g]` quadrature probability. For the last 3 variables `[i]` is the band `[j]` is temperature and `[g]` is quadrature point dependency, respectively.
- `readP` reading in all spectral properties of particles (scattering), comprising of: `kPp` Planck absorption coefficient, `narrBand[i].Cabs` absorption cross section and `narrBand[i].Csca` scattering cross section, where `[i]` is the band. Again the variable `probp[i][j]` contains the band probability for particles where `[j]` is temperature.
- `readH` reads the probability of scattering angle `phi->cuprob[j][i]` based on the phase function calculated by MIE theory. `[j]` is the angle while `[i]` is the band (phase function is independent of temperature).

`spline.cpp`
This file contain routines that take as an input the temperature of the fluid, the temperature of the particles and the concentration of the particles on the whole grid and interpolate them on coarser grids based on `grid_num` and `maxi`, `maxj` and `maxk` (see the header `param.h`). The interpolation is done with a three dimensional spline for the temperatures, while for the concentrations, the number of particles in the finer mesh are added on the coarser meshes and divided by the mesh volume $\#/m^3$.

- `sum3D` calculates the "summed" particle number on the coarser mesh and returns the concentration
- `interpolate3D` interpolates fluid temperature on coarser meshes
- `interpolate3DP` interpolates particle temperature on coarser meshes
- `spline` calculates spline coefficients (numerical recipes)
- `splint` calculates spline interpolant (numerical recipes)

`memory_copy.cu`
This file handles all the communication `HostToDevice` setting up all the GPU variables and all texture memories (3D variables with random access, namely, temperature, particle temperature, concentration, absorption coefficient, quadrature probability)

- `grid_copy` copying the grid details to the GPU
- `temp_fluid_copy` copying the fluid temperature (for all grids) in GPU with an array of texture memory objects
- `temp_part_copy` copying the particle temperature (for all grids) in GPU with an array of texture memory objects
- `concentration_copy` copying the concentration (for all grids) in GPU with an array of texture memory objects
- `narrowband_copy` copying all other relevant variables in GPU. Note that, if `srt==1`, the variables dependent on the narrow bands are copied following the sorting operated performed by `sort_idx`.