# Auto GPUification

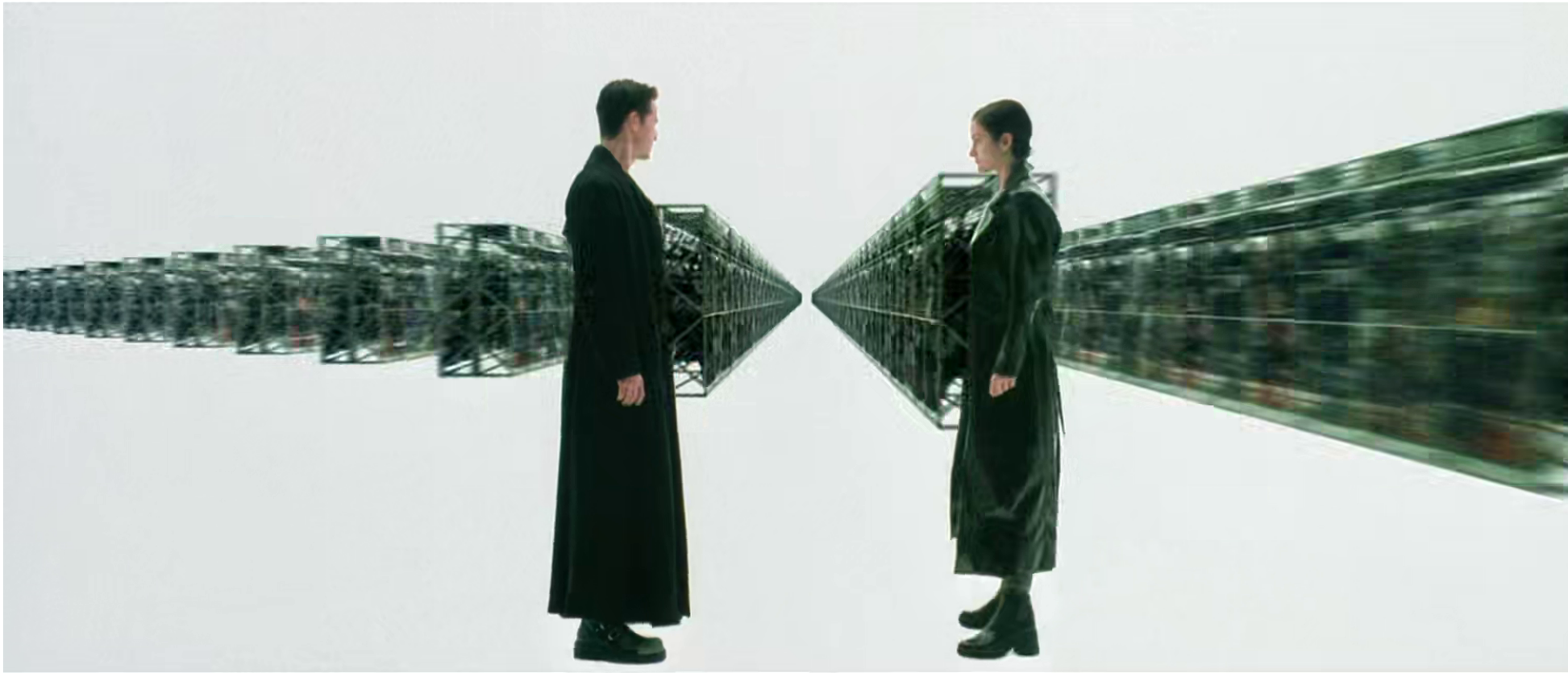
## Comment accélérer son modèle d'effet de serre sans suer ?

Vincent Lafage

IJCLab, Laboratoire de Physique des 2 Infinis Irène Joliot-Curie
CNRS/IN2P3 & Université Paris-Saclay, Orsay, France


27 mai 2026

# GPU... *Lots of threads*



- OpenCL
  + passe-partout (y compris hors NVIDIA)
  – Lourd (+ Binding C)
- (Vulkan)
  + passe-partout (y compris hors NVIDIA)
  – Lourd (+ Binding C)
- CUDAFortran
  + bibliothèques cuBLAS, cuFFT, cuRAND
  – passe seulement sur NVIDIA
- OpenACC (directives !$acc)
  – gestion explicite de la mémoire
- OpenMP (directives !$omp)
  – modèle d'*offloading* distinct
- nvc++ & nvfortran
  – passe seulement sur NVIDIA
  + std::for_each / std::transform / do concurrrent ⇒ rien à faire (vraiment ???)

# Échelles du parallélisme

- Processus (dans des environnements bien séparés) :
  - problèmes dits *« embarassingly parallel »*
    — traitement distribué par événement (physique des particules) sur une grille
  - communication inter-processus *à la Unix*
    — pipe
    — mémoire partagée
    — sémaphore
  - avec des ordinateurs séparés et passage de messages à travers le réseau (systèmes à mémoire distribuée) :
    — `PVM` *« Parallel Virtual Machine »* 1989
    — `MPI` *« Message Passing Interface »* 1994

- Threads (*alias « processus légers »*)
  - overhead réduits : *par rapport à un processus* — `C/C++ pthread` *« POSIX thread »* 1995
    — tâche `Ada` / objets protégés
  - Shared Memory sur SMP *« Symmetric Multi Processing »* machines :
    — `OpenMP` *« Open Multi Processing »* 1997.

- Vectorisation :
  - même les threads ont trop de surcharge lorsque les tâches concurrentes sont élémentaires :
    ⇒ nous ne pouvons utiliser aucun thread CPU pour la somme individuelle ou les produits de nombreuses données ⇒ we can use no CPU thread for individual sum or products of many data
  - ILP *« Instruction Level Parallelism »*
    — processeur vectoriel *« à la Cray »* : SIMD *« Single Instruction, Multiple Data »*

    — processeur moderne avec SIMD à longueur fixe

      `MMX` *« MultiMedia eXtension »* 1997 Pentium P5
      `SSE` *« Streaming Extension »* 1999 Pentium ///
      `AVX` *« Advanced Vector eXtension »* 2008-2011
      `AVX512` 2013-2017

# Comment vectoriser ? ranger les données

Nous connaissons *le parallélisme au niveau des bits* :
nous pouvons AND ou OR 8, 16, 32, 64 bits en une seule instruction
De la même manière, pour que notre instruction vectorielle fonctionne de manière fluide, nous devons aligner notre vecteur de données sur les limites de la mémoire.

Nous avons intérêt à ordonner les données de même nature comme un tableau (aligné) (contigu)
$\Rightarrow$ pour la contiguïté des données nous préférons une

**structure de tableaux**

à la

**collection (tableau) d'objets (structure)**

orientée objet classique.

# Comment vectoriser ?

La vectorisation, un processus d'industrialisation :
considèrez votre calcul comme une ligne de production

Prise en tenaillle de la performance :

- paralléliser d'en haut,
- vectoriser d'en bas

... + garder le cache chaud !

# Comment vectoriser ?

1. créer manuellement le code vectoriel avec des instructions de bas niveau dédiées (intrinsèques)
2. utiliser des bibliothèques vectorielles
3. laisser la main au compilateur, à supposer qu'on puisse
   1. exprimer le parallélisme des données (aspect de haut niveau)
      $\Rightarrow$ mettre à profit la puissance des tableaux (*à la Matlab / Fortran 90*)
   2. donner des indices aux compilateur quant à la contiguité des données & leur alignement (aspect de bas niveau)

Et puis, comment vérifier qu'elle a bien été vectorisée ?

- flagS du compilateur pour signaler la vectorisation
- vérifier l'assemblage résultant (yuk !)
- `maqao`
- ...indirectement, `perf`

# Comment vectoriser ?

Les nombres complexes : une vectorisation symbolique
... qui fait obstacle à la vectorisation pratique !

$$(a+ib)\times(c+id) = (ac-bd)+i(ad+bc)$$

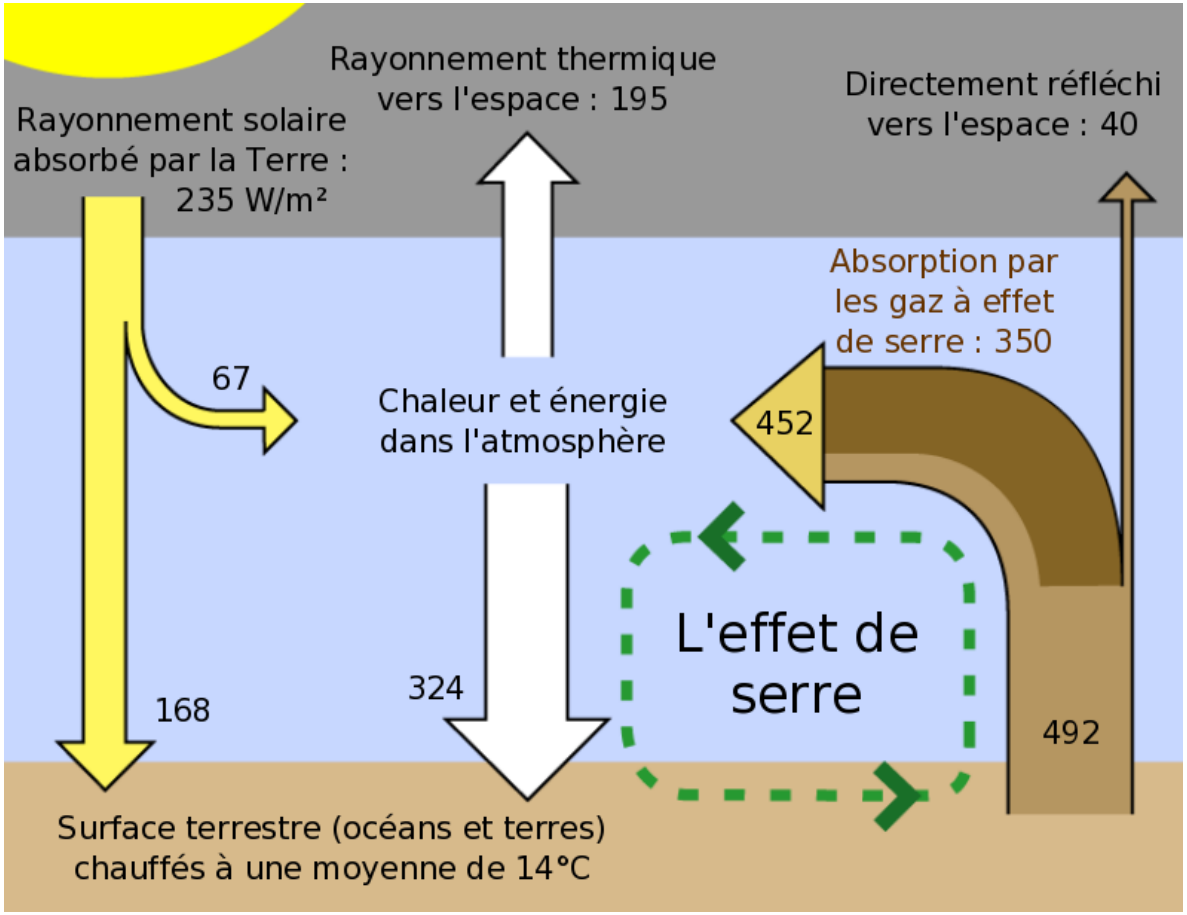
On veut un complexe de vecteurs plutôt qu'un vecteur de complexes.

$$\begin{aligned}
(a_1+ib_1)\times(c_1+id_1) &= (a_1c_1-b_1d_1)+i(a_1d_1+b_1c_1)\\
(a_2+ib_2)\times(c_2+id_2) &= (a_2c_2-b_2d_2)+i(a_2d_2+b_2c_2)\\
(a_3+ib_3)\times(c_3+id_3) &= (a_3c_3-b_3d_3)+i(a_3d_3+b_3c_3)\\
(a_4+ib_4)\times(c_4+id_4) &= (a_4c_4-b_4d_4)+i(a_4d_4+b_4c_4)
\end{aligned}$$

# Effet de serre

https://github.com/scienceetonnante/RadiativeForcing

https://www.youtube.com/watch?v=ewc8FBtEKPs



# Saturation de l'effet de serre ?

un argument climato-sceptique

https://www.randombio.com/co2.html

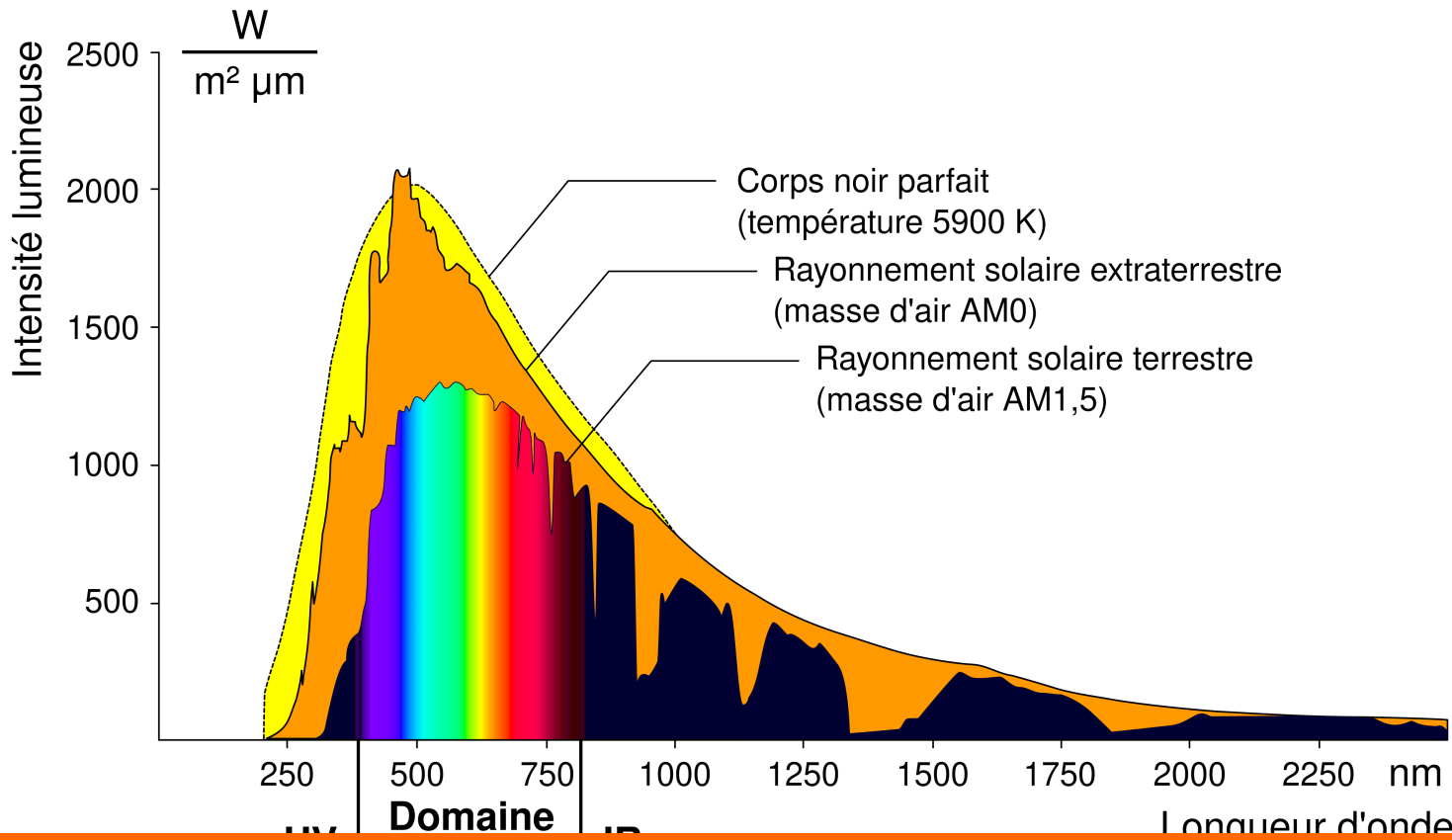
Solar irradiance $(1{,}3608 \pm 0{,}0005)\,\mathrm{kW/m^2}$

arrivent perpendiculaire au disque terrestre (superficie : $\pi R^2$) et sont réparties sur la sphère terrestre (superficie : $4\pi R^2$)

$\Longrightarrow (340{,}2 \pm 0{,}1)\,\mathrm{W/m^2}$

Une partie est réfléchie (albédo terrestre moyenne $0{,}3$ )

$\Longrightarrow (238{,}14 \pm 0{,}09)\,\mathrm{W/m^2}$



# Effet de serre

$$F_{\text{sortant}} = F_{\text{entrant}} - \kappa \Delta z F_{\text{entrant}} + \kappa \Delta z \pi B(\lambda, T)$$

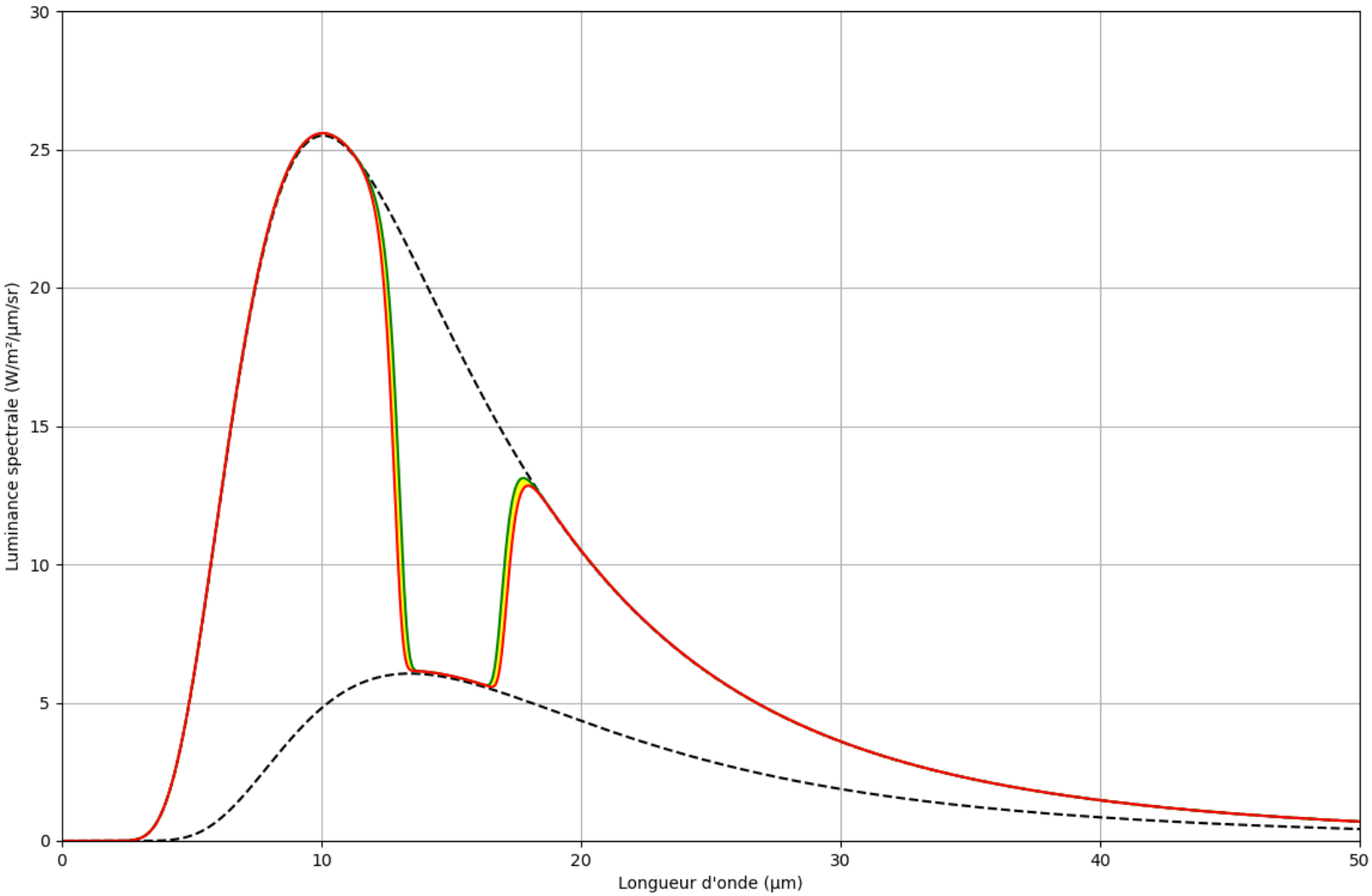
où :

- $F_{\text{sortant}}$ est le flux sortant de la couche,
- $F_{\text{entrant}}$ est le flux entrant dans la couche,
- $\kappa$ est le coefficient d'absorption,
- $\Delta z$ est l'épaisseur de la couche,
- $B(\lambda, T)$ est la fonction de Planck pour la luminance spectrale d'un corps noir à une température $T$ et une longueur d'onde $\lambda$.

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \frac{1}{e^{\frac{hc}{\lambda k_B T}} - 1}$$

- $h$ est la constante de Planck ($6{,}626\,070\,15 \times 10^{-34}$ J · s),
- $c$ est la vitesse de la lumière ($2{,}997\,924\,58 \times 10^{8}$ m · s$^{-1}$),
- $k_B$ est la constante de Boltzmann ($1{,}380\,649 \times 10^{-23}$ J · K$^{-1}$ ).

# Saturation de l'effet de serre ?



# Effet de serre

## On part de la version Python (102 lignes de code) https://github.com/scienceetonnante/RadiativeForcing

```python
def simulate_radiative_transfer(CO2_fraction, z_max = 80000.0, delta_z = 10.0, lambda_min = 0.1e-6, lambda_max = 100e-6, delta_lambda = 0.01e-6):

    # Altitude and wavelength grids
    z_range = np.arange(0, z_max, delta_z)
    lambda_range = np.arange(lambda_min, lambda_max, delta_lambda)

    # Initialize arrays
    upward_flux = np.zeros((len(z_range), len(lambda_range)))
    optical_thickness = np.zeros((len(z_range), len(lambda_range)))

    # Boundary condition : Compute the outward vertical flux emitted by the Earth's surface for all wavelengths
    earth_flux = np.pi * planck_function(lambda_range, temperature(0.0)) * delta_lambda
    print(f"Total earth surface flux in wavelength range: {earth_flux.sum():.2f} W/m^2")
    print(earth_flux.sum())

    flux_in = earth_flux
    for i, z in enumerate(z_range):

        # Number density of CO2 molecules and absorption coefficient
        n_CO2 = air_number_density(z) * CO2_fraction
        kappa = cross_section_CO2(lambda_range) * n_CO2

        # Compute fluxes within the layer
        optical_thickness[i,:] = kappa * delta_z
        absorbed_flux = np.minimum(kappa * delta_z * flux_in , flux_in)
        emitted_flux = optical_thickness[i,:] * np.pi * planck_function(lambda_range, temperature(z)) * delta_lambda
        upward_flux[i, :] = flux_in - absorbed_flux + emitted_flux

        # The flux leaving the layer becomes the flux entering the next layer
        flux_in = upward_flux[i, :]

    print(f"Total outgoing flux at the top of the atmosphere: {upward_flux[-1,:].sum():.2f} W/m^2")
    print(upward_flux[-1,:].sum())

    return lambda_range, z_range, upward_flux, optical_thickness

# ------------------------------------------------------------------------------------------------------------------------

# MAIN

CO2_fraction = 280.0e-6
lambda_range, z_range, upward_flux, optical_thickness = simulate_radiative_transfer(CO2_fraction)

CO2_fraction *= 2
```

# Effet de serre

et on produit une version `Fortran` (148 lignes de code) https://gitlab.in2p3.fr/lafage/radiative_forcing.git

```
  impure subroutine simulate_radiative_transfer (CO2_fraction, lambda_range, z_range, upward_flux, optical_thickness)
    real (pr), intent (in) :: CO2_fraction
    real (pr), dimension (:), intent (in) :: lambda_range, z_range
    real (pr), dimension (:, :), allocatable, intent (out) :: upward_flux, optical_thickness ! épaisseur optique dans chaque couche d'altitude i et
    real (pr), dimension (:), allocatable :: earth_flux, flux_in
    real (pr), dimension (size (lambda_range), size (z_range)) :: kappa, absorbed_flux, emitted_flux
    real (pr), dimension (size (z_range)) :: n_CO2
    integer :: i, j

    if (.not. allocated (optical_thickness)) allocate (optical_thickness (size (lambda_range), size (z_range)))
    if (.not. allocated (upward_flux)) allocate (upward_flux (size (lambda_range), size (z_range)))

    ! Boundary condition: Compute the outward vertical flux emitted by the Earth's surface for all wavelengths
    earth_flux = pi * planck_function (lambda_range, temperature (0.0_pr)) * delta_lambda
    print *, "Total earth surface flux in wavelength range:", sum (earth_flux), "W/m^2"

    flux_in = earth_flux

    do i = 1, size (z_range)
       n_CO2 (i) = air_number_density (z_range (i)) * CO2_fraction
       do concurrent (j=1:size (lambda_range))
          kappa (j, i) = cross_section_CO2 (lambda_range (j)) * n_CO2 (i)
          ! Compute fluxes within the layer
          optical_thickness (j, i) = kappa (j, i) * delta_z
          absorbed_flux (j, i) = min (kappa (j, i) * delta_z * flux_in (j), flux_in (j))
          emitted_flux (j, i) = optical_thickness (j, i) * pi * &
               planck_function (lambda_range (j), temperature (z_range (i))) * delta_lambda
          upward_flux (j, i) = flux_in (j) - absorbed_flux (j, i) + emitted_flux (j, i)
          ! The flux leaving the layer becomes the flux entering the next layer
          flux_in (j) = upward_flux (j, i)
       end do
    end do

    print *, "Total outgoing flux at the top of the atmosphere:", sum (upward_flux (:, size (z_range))), "W/m^2"

    deallocate (earth_flux, flux_in)
  end subroutine simulate_radiative_transfer
end module radiation_transfer

program main
  use constants
  use radiation_transfer
  implicit none
  real (pr), parameter :: CO2_fraction_before_industrial_era = 280.0e-6_pr
```

# Effet de serre

## ainsi qu'une version C++ (138 lignes de code) https://gitlab.in2p3.fr/lafage/radiative_forcing.git

```cpp
void simulate_radiative_transfer (double CO2_fraction, const std::vector<double>& lambda_range, const std::vector<double>& z_range,
                                  std::vector<std::vector<double>>& upward_flux, std::vector<std::vector<double>>& optical_thickness) {
    const auto num_lambda = lambda_range.size ();
    const auto num_z = z_range.size ();

    // Allocate and initialize arrays
    upward_flux.resize (num_z, std::vector<double> (num_lambda, 0.0));
    optical_thickness.resize (num_z, std::vector<double> (num_lambda, 0.0));

    // Boundary condition: Earth's surface flux
    std::vector<double> earth_flux (num_lambda);
    for (unsigned j = 0U; j < num_lambda; ++j) {
        earth_flux[j] = M_PI * constants::delta_lambda * planck_function (lambda_range[j], temperature(0.0));
    }

    double total_earth_flux = 0.0;
    for (double flux : earth_flux) {
        total_earth_flux += flux;
    }
    std::cout << "Total earth surface flux in wavelength range: "
              << total_earth_flux << " W/m^2" << std::endl;

    std::vector<double> flux_in = earth_flux;
    std::vector<double> n_CO2 (num_z);
    for (unsigned i = 0U; i < num_z; ++i) {
        n_CO2[i] = air_number_density (z_range[i]) * CO2_fraction;
    }

    std::vector<double> absorbed_flux(num_lambda);
    std::vector<double> emitted_flux(num_lambda);
    for (unsigned i = 0U; i < num_z; ++i) {
        for (unsigned j = 0U; j < num_lambda; ++j) {
            upward_flux[i][j] = cross_section_CO2 (lambda_range[j]) * n_CO2[i];
            optical_thickness[i][j] = upward_flux[i][j] * constants::delta_z;
        }
        for (unsigned j = 0U; j < num_lambda; ++j) {
            absorbed_flux[j] = std::min(upward_flux[i][j] * constants::delta_z * flux_in[j], flux_in[j]);
            emitted_flux[j] = optical_thickness[i][j] * M_PI * planck_function(lambda_range[j], temperature(z_range[i])) * constants::delta_lambda;
            upward_flux[i][j] = flux_in[j] - absorbed_flux[j] + emitted_flux[j];
        }
        flux_in = upward_flux[i];
    }

    double total_outgoing_flux = 0.0;
```

# Effet de serre

Ça compile,

```
 g++ --param=ssp-buffer-size=8 -ftree-vectorize -fopt-info -ftree-vectorizer-verbose=1
   -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx512vp
     -fopenmp -fopenmp-simd -Wall -Wextra -Wpedantic -Wconversion -Wshadow -Wpedantic
     -O3 -g -march=native -mtune=native -funroll-loops -ftree-parallelize-loops=8 -std=c++17
     -o radiative_forcing_no_algo radiative_forcing.cpp


gfortran --param=ssp-buffer-size=8 -ftree-vectorize -fopt-info -ftree-vectorizer-verbose=1
      -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx512
         -fopenmp -fopenmp-simd -fimplicit-none -Wno-unused-dummy-argument -Wall -Wextra -Wpedantic -Wconversion -Wshadow
         -O3 -g -march=native -mtune=native -funroll-loops -fmax-array-constructor=80000
         -o radiative_forcing_gf radiative_forcing.f90
```

# Effet de serre

ça s'exécute

```
time ./radiative_forcing_no_algo        # C++
Total earth surface flux in wavelength range: 389.127 W/m^2
Total outgoing flux at the top of the atmosphere: 336.335 W/m^2
Total earth surface flux in wavelength range: 389.127 W/m^2
Total outgoing flux at the top of the atmosphere: 331.535 W/m^2

real 0m2,284s
user 0m2,206s
sys 0m0,149s
```

```
time ./radiative_forcing_gf             # Fortran
 Total earth surface flux in wavelength range:   389.12793040773630      W/m^2
 Total outgoing flux at the top of the atmosphere:   336.33535609257359      W/m^2
 Total earth surface flux in wavelength range:   389.12793040773630      W/m^2
 Total outgoing flux at the top of the atmosphere:   331.53568078124198      W/m^2

real 0m2,742s
user 0m2,286s
sys 0m0,456s
```

(les temps d'exécution sont proches)
Et les résultats sont différents ($0{,}3\,\%$ à $0{,}4\,\%$) de `Python`...

```
python RadiativeForcing/RadiativeForcing.py
Total earth surface flux in wavelength range: 388.03 W/m^2
Total outgoing flux at the top of the atmosphere: 335.03 W/m^2
Total earth surface flux in wavelength range: 388.03 W/m^2
Total outgoing flux at the top of the atmosphere: 330.22 W/m^2
```

...

# Effet de serre

## on débuggue le Python

Avant

```
python RadiativeForcing/RadiativeForcing.py
Total earth surface flux in wavelength range: 388.03 W/m^2
Total outgoing flux at the top of the atmosphere: 335.03 W/m^2
Total earth surface flux in wavelength range: 388.03 W/m^2
Total outgoing flux at the top of the atmosphere: 330.22 W/m^2
```

Après

```
$ python3 --version
Python 3.11.2
$ time python3 radiative_forcing.py
Total earth surface flux in wavelength range: 389.13 W/m^2
389.12793044282944
Total outgoing flux at the top of the atmosphere: 336.34 W/m^2
336.33535612766707
Total earth surface flux in wavelength range: 389.13 W/m^2
389.12793044282944
Total outgoing flux at the top of the atmosphere: 331.54 W/m^2
331.5356808163359

real 0m3,208s
user 0m3,155s
sys 0m1,550s
```

Le temps d'exécution n'est pas ridicule.
et les résultats sont désormais identiques à Fortran et à 1 ppm de C++

⇒ tout ça à cause de l'absence de typage fort en Python :

des paramètres numériques passés sans point décimal ont transformé des expression moralement flottantes en entier, d'où troncatures et mélanges

# Effet de serre

Pour confirmer, on produit une version Ada (147 lignes de code) https://gitlab.in2p3.fr/lafage/radiative_forcing.git

```
procedure Simulate_Radiative_Transfer
          (CO2_Fraction : Real;
           Lambda_Range, Z_Range : Real_Array_Access;
           Upward_Flux, Optical_Thickness : out Real_2d_Array_Access) is
   Earth_Flux     : Real_Array_Access
       := new Real_Array (1 .. Lambda_Count);
   Flux_In        : Real_Array_Access
       := new Real_Array (1 .. Lambda_Count);
   Kappa          : Real_2d_Array_Access
       := new Real_2d_Array (1 .. Lambda_Count, 1 .. Z_Size);
   Absorbed_Flux : Real_2d_Array_Access
       := new Real_2d_Array (1 .. Lambda_Count, 1 .. Z_Size);
   Emitted_Flux  : Real_2d_Array_Access
       := new Real_2d_Array (1 .. Lambda_Count, 1 .. Z_Size);
   N_CO2          : Real_Array_Access
       := new Real_Array (1 .. Z_Size);
   Total_Earth_Flux : Real := 0.0;
   Total_Outgoing_Flux : Real := 0.0;
begin
   for I in 1 .. Lambda_Count loop
      Earth_Flux (I) := Pi * Delta_Lambda *
                        Planck_Function (Lambda_Range (I), Temperature (0.0));
      Total_Earth_Flux := Total_Earth_Flux + Earth_Flux (I);
   end loop;

   Put_Line ("Total earth surface flux in wavelength range: " &
             Real'Image (Total_Earth_Flux) & " W/m^2");

   Flux_In := Earth_Flux;

   for I in 1 .. Z_Size loop
      N_CO2 (I) := Air_Number_Density (Z_Range (I)) * CO2_Fraction;

      for J in 1 .. Lambda_Count loop
         Kappa (J, I) := Cross_Section_CO2 (Lambda_Range (J)) * N_CO2 (I);
         Optical_Thickness (J, I) := Kappa (J, I) * Delta_Z;
         Absorbed_Flux (J, I) :=
              Real'Min (Kappa (J, I) * Delta_Z * Flux_In (J), Flux_In (J));
         Emitted_Flux (J, I) := Optical_Thickness (J, I) * Pi
             * Planck_Function (Lambda_Range (J), Temperature (Z_Range (I)))
             * Delta_Lambda;
         Upward_Flux (J, I) :=
                  Flux_In (J) - Absorbed_Flux (J, I) + Emitted_Flux (J, I);
         Flux_In (J) := Upward_Flux (J, I);
```

# Effet de serre

Ça s'exécute,

```
Total earth surface flux in wavelength range:  3.89127930442830E+02 W/m^2
Total outgoing flux at the top of the atmosphere:  3.36335356127668E+02 W/m^2
Total earth surface flux in wavelength range:  3.89127930442830E+02 W/m^2
Total outgoing flux at the top of the atmosphere:  3.31535680816336E+02 W/m^2

real 0m16,034s
user 0m15,684s
sys 0m0,348s
```

et les résultats sont identiques au nouveau `Python` et à `Fortran` et à $1$ ppm ou $2$ ppm de `C++`

# Parallélisme

Et si on exprimait le parallélisme de ce calcul ? ⇒ en `Fortran`, c'est déjà fait

```
    do i = 1, size (z_range)
       n_CO2 (i) = air_number_density (z_range (i)) * CO2_fraction
       do concurrent (j=1:size (lambda_range))
          kappa (j, i) = cross_section_CO2 (lambda_range (j)) * n_CO2 (i)
          ! Compute fluxes within the layer
          optical_thickness (j, i) = kappa (j, i) * delta_z
          absorbed_flux (j, i) = min (kappa (j, i) * delta_z * flux_in (j), flux_in (j))
          emitted_flux (j, i) = optical_thickness (j, i) * pi * &
               planck_function (lambda_range (j), temperature (z_range (i))) * delta_lambda
          upward_flux (j, i) = flux_in (j) - absorbed_flux (j, i) + emitted_flux (j, i)
          ! The flux leaving the layer becomes the flux entering the next layer
          flux_in (j) = upward_flux (j, i)
       end do
    end do
```

Ça compile,

```
$ gfortran --param=ssp-buffer-size=8 -ftree-vectorize -fopt-info -ftree-vectorizer-verbose=1
       -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx51
           -fopenmp -fopenmp-simd -fimplicit-none -Wno-unused-dummy-argument -Wall -Wextra -Wpedantic -Wconversion -Wshadow
           -O3 -g -march=native -mtune=native -funroll-loops -fmax-array-constructor=80000
           -ftree-parallelize-loops=8 -o radiative_forcing_gf radiative_forcing.f90


$ time ./radiative_forcing_gf
 Total earth surface flux in wavelength range:   389.12793040773630      W/m^2
 Total outgoing flux at the top of the atmosphere:   336.33535609257359      W/m^2
 Total earth surface flux in wavelength range:   389.12793040773630      W/m^2
 Total outgoing flux at the top of the atmosphere:   331.53568078124198      W/m^2

real 0m1,269s
user 0m5,890s
sys 0m4,246s
```

⇒ Moins de temps chrono mais plus de temps CPU (overhead pour trop de petits threads)

```
$ nvfortran --version

nvfortran 26.3-0 64-bit target on x86-64 Linux -tp tigerlake
NVIDIA Compilers and Tools
Copyright (c) 2026, NVIDIA CORPORATION & AFFILIATES.  All rights reserved.
$ # double precision
$ nvfortran -g -O4 -Minfo=accel -fast -Wall
        -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx5
            -Mdclchk radiative_forcing.f90 -o radiative_forcing_nv && time ./radiative_forcing_nv
 Total earth surface flux in wavelength range:    389.1279304077366      W/m^2
 Total outgoing flux at the top of the atmosphere:    336.3353560925743       W/m^2
 Total earth surface flux in wavelength range:    389.1279304077366      W/m^2
 Total outgoing flux at the top of the atmosphere:    331.5356807812431       W/m^2

real 0m1,875s
user 0m1,398s
sys 0m0,476s

$ # simple precision
$ nvfortran -g -O4 -Minfo=accel -fast -Wall
        -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx5
            -Mdclchk radiative_forcing.f90 -o radiative_forcing_nv && time ./radiative_forcing_nv
 Total earth surface flux in wavelength range:    389.1281     W/m^2
 Total outgoing flux at the top of the atmosphere:    336.3358     W/m^2
 Total earth surface flux in wavelength range:    389.1281     W/m^2
 Total outgoing flux at the top of the atmosphere:    331.5361     W/m^2

real 0m0,860s
user 0m0,655s
sys 0m0,202s
```

⇒ ça optimise bien !

```
$ nvfortran -g -O4 -stdpar=multicore -Minfo=accel -fast -Wall
        -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx5
            -Mdclchk radiative_forcing.f90 -o radiative_forcing_nv_multicore && time ./radiative_forcing_nv_multicore
simulate_radiative_transfer:
    167, Generating Multicore code
        167, Loop parallelized across CPU threads

 Total earth surface flux in wavelength range:    389.1281     W/m^2
 Total outgoing flux at the top of the atmosphere:    336.3358     W/m^2
 Total earth surface flux in wavelength range:    389.1281     W/m^2
 Total outgoing flux at the top of the atmosphere:    331.5361     W/m^2
...
real 0m1,709s
user 0m13,360s
sys 0m13,072s
```

$\Rightarrow$ trop d'overhead pour des petits threads

# Parallélisme C++

## version C++ std::for_each (142 lignes de code)

```
void simulate_radiative_transfer (double CO2_fraction, const std::vector<double>& lambda_range, const std::vector<double>& z_range,
                                  std::vector<std::vector<double>>& upward_flux, std::vector<std::vector<double>>& optical_thickness) {
    const auto num_lambda = lambda_range.size ();
    const auto num_z = z_range.size ();

    // Allocate and initialize arrays
    upward_flux.resize (num_z, std::vector<double> (num_lambda, 0.0));
    optical_thickness.resize (num_z, std::vector<double> (num_lambda, 0.0));

    // Boundary condition: Earth's surface flux
    std::vector<double> earth_flux (num_lambda);
    std::vector<unsigned> indices (num_lambda);
    std::iota (indices.begin (), indices.end (), 0U);   // besoin de #include <numeric>
    std::transform (std::execution::par_unseq, lambda_range.begin (), lambda_range.end (), earth_flux.begin (),
                    [](double lambda) {
                        return M_PI * constants::delta_lambda * planck_function (lambda, temperature (0.0));
                    });

    //double total_earth_flux = std::reduce (std::execution::par_unseq, earth_flux.begin (), earth_flux.end ());
    double total_earth_flux = std::transform_reduce (std::execution::par_unseq, earth_flux.begin (), earth_flux.end (), 0.0, std::plus<>{}, [](doub
    std::cout << "Total earth surface flux in wavelength range: "
              << total_earth_flux << " W/m^2" << std::endl;

    std::vector<double> flux_in = earth_flux;
    std::vector<double> n_CO2 (num_z);
    std::transform (std::execution::par_unseq, z_range.begin (), z_range.end (), n_CO2.begin (),
                    [CO2_fraction](double z) {
                        return air_number_density (z) * CO2_fraction;
                    });

    std::vector<double> absorbed_flux(num_lambda);
    std::vector<double> emitted_flux(num_lambda);
    for (unsigned i = 0U; i < num_z; ++i) {
        const auto temp = temperature (z_range[i]);  // calculé une seule fois

        std::for_each (std::execution::par_unseq, indices.begin (), indices.end (),
                       [&](unsigned j) {
            upward_flux[i][j] = cross_section_CO2 (lambda_range[j]) * n_CO2[i];
            optical_thickness[i][j] = upward_flux[i][j] * constants::delta_z;
            absorbed_flux[j] = std::min(upward_flux[i][j] * constants::delta_z * flux_in[j], flux_in[j]);
            emitted_flux[j] = optical_thickness[i][j] * M_PI
                            * planck_function(lambda_range[j], temp)
                            * constants::delta_lambda;
```

# Parallélisme C++

## version C++ std::transform (155 lignes de code)

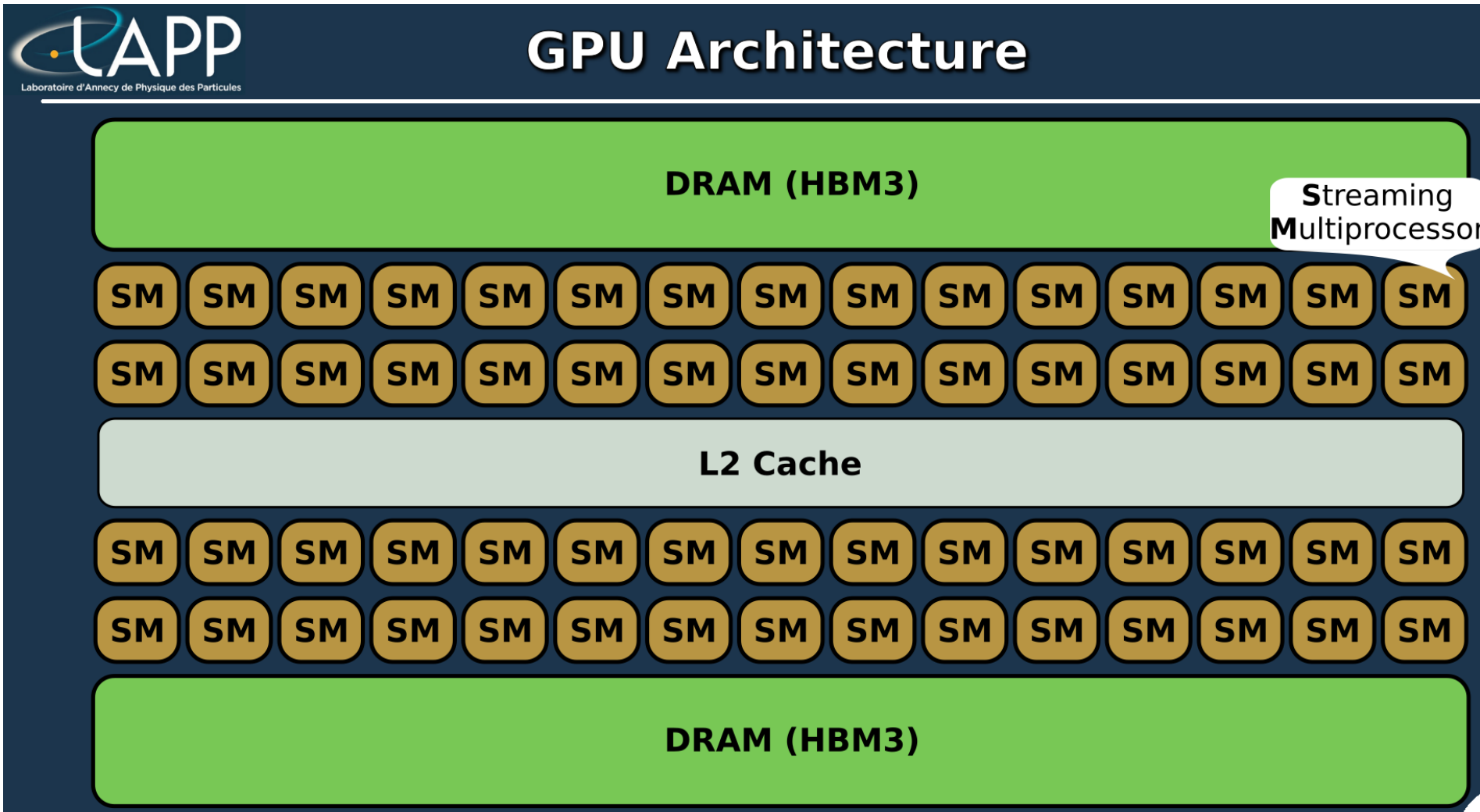
```
void simulate_radiative_transfer (double CO2_fraction, const std::vector<double>& lambda_range, const std::vector<double>& z_range,
                                  std::vector<std::vector<double>>& upward_flux, std::vector<std::vector<double>>& optical_thickness) {
    const auto num_lambda = lambda_range.size ();
    const auto num_z = z_range.size ();

    // Allocate and initialize arrays
    upward_flux.resize (num_z, std::vector<double> (num_lambda, 0.0));
    optical_thickness.resize (num_z, std::vector<double> (num_lambda, 0.0));

    // Boundary condition: Earth's surface flux
    std::vector<double> earth_flux (num_lambda);
    std::vector<unsigned> indices (num_lambda);
    std::iota (indices.begin (), indices.end (), 0U);   // besoin de #include <numeric>
    std::transform (std::execution::par_unseq, lambda_range.begin (), lambda_range.end (), earth_flux.begin (),
                    [](double lambda) {
                        return M_PI * constants::delta_lambda * planck_function (lambda, temperature (0.0));
                    });

    //double total_earth_flux = std::reduce (std::execution::par_unseq, earth_flux.begin (), earth_flux.end ());
    double total_earth_flux = std::transform_reduce (std::execution::par_unseq, earth_flux.begin (), earth_flux.end (), 0.0, std::plus<>{}, [](doub
    std::cout << "Total earth surface flux in wavelength range: "
              << total_earth_flux << " W/m^2" << std::endl;

    std::vector<double> flux_in = earth_flux;
    std::vector<double> n_CO2 (num_z);
    std::transform (std::execution::par_unseq, z_range.begin (), z_range.end (), n_CO2.begin (),
                    [CO2_fraction](double z) {
                        return air_number_density (z) * CO2_fraction;
                    });

    std::vector<double> absorbed_flux(num_lambda);
    std::vector<double> emitted_flux(num_lambda);
    for (unsigned i = 0U; i < num_z; ++i) {
        std::transform(std::execution::par_unseq, lambda_range.begin(), lambda_range.end(), upward_flux[i].begin(),
                       [&n_CO2, i](double lambda) {
                           return cross_section_CO2(lambda) * n_CO2[i];
                       });
        std::transform(std::execution::par_unseq, upward_flux[i].begin(), upward_flux[i].end(), optical_thickness[i].begin(),
                       [](double flux) {
                           return flux * constants::delta_z;
                       });
        std::transform(std::execution::par_unseq, upward_flux[i].begin(), upward_flux[i].end(), flux_in.begin(), absorbed_flux.begin(),
                       [](double flux, double flux_in_val) {
```

```
$  nvc++ -g -std=c++20 -O4 -stdpar=multicore -Minfo=accel -fast -Wall
     -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx512v
        radiative_forcing.TRANSFORM.cpp -ltbb -o radiative_forcing_nv++ && time ./radiative_forcing_nv++
…
Total earth surface flux in wavelength range: 389.127 W/m^2
Total outgoing flux at the top of the atmosphere: 336.335 W/m^2
Total earth surface flux in wavelength range: 389.127 W/m^2
Total outgoing flux at the top of the atmosphere: 331.535 W/m^2

real 0m1,491s
user 0m16,354s
sys 0m2,940s
```

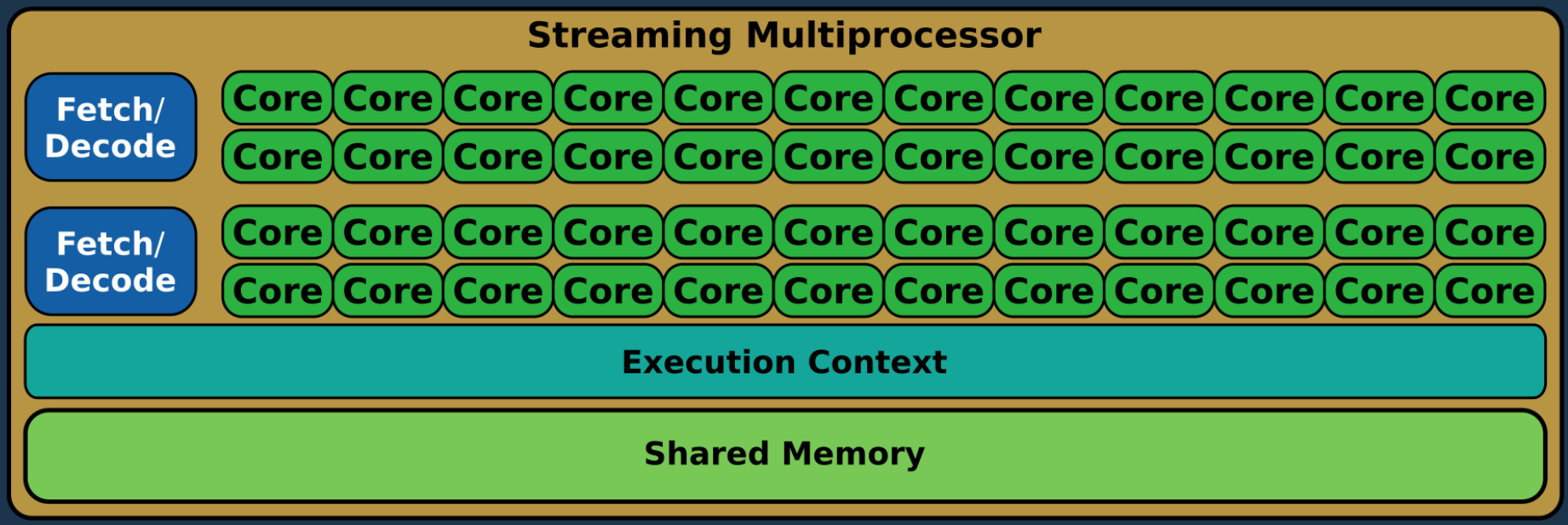
⇒ trop d'overhead pour des petits threads

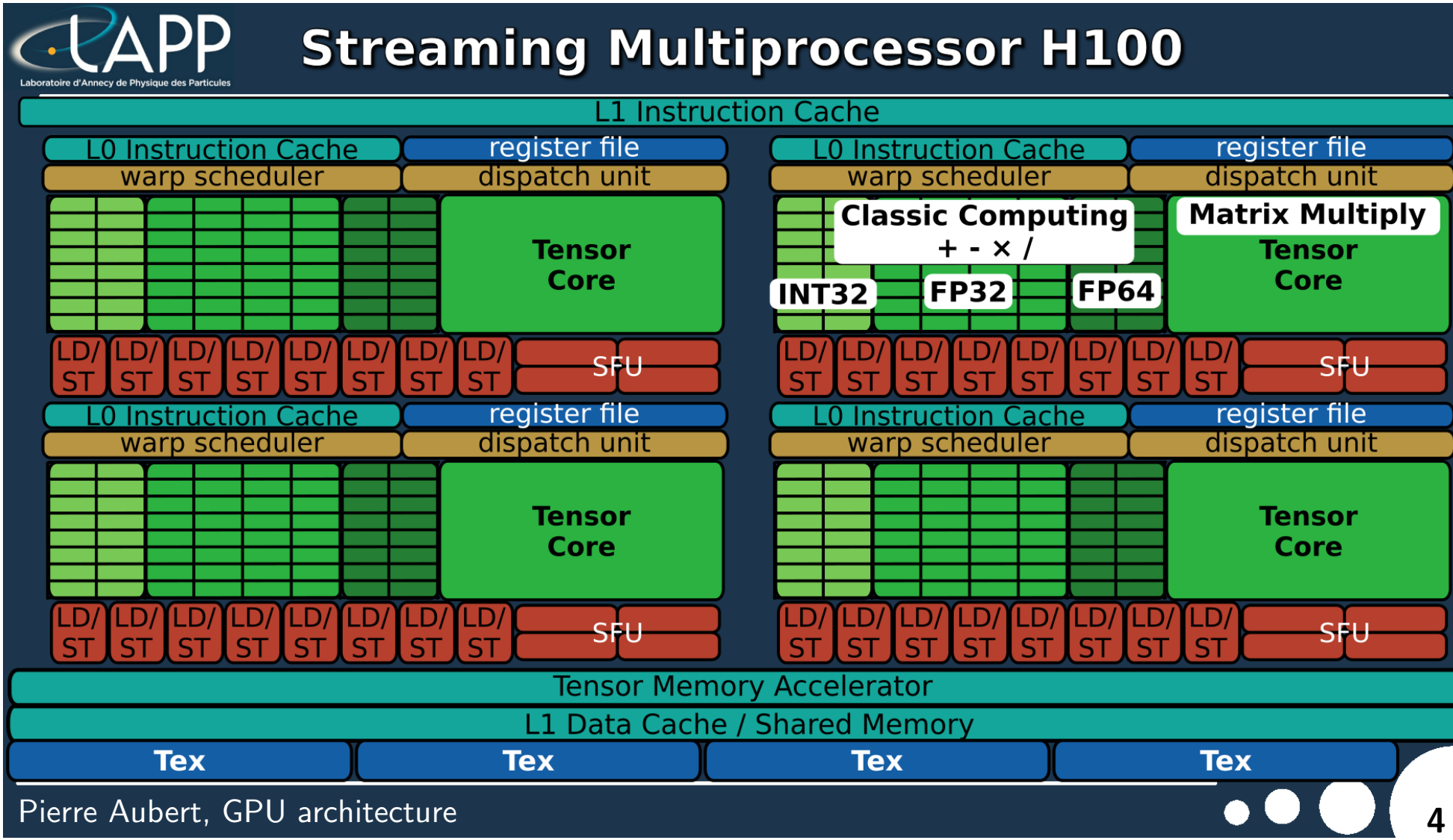
```
$ nvc++ -g -std=c++20 -O4 -Minfo=accel -fast -Wall
     -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx512v
        radiative_forcing.FOREACH.cpp -ltbb -o radiative_forcing_nv++ && time ./radiative_forcing_nv++
…
real 0m10,681s
user 0m10,331s
sys 0m0,343s
Total earth surface flux in wavelength range: 389.127 W/m^2
Total outgoing flux at the top of the atmosphere: 336.335 W/m^2
Total earth surface flux in wavelength range: 389.127 W/m^2
Total outgoing flux at the top of the atmosphere: 331.535 W/m^2

real 0m0,693s
user 0m7,617s
sys 0m0,215s
```

⇒ trop d'overhead pour des petits threads, mais mieux avec FOREACH que TRANSFORM


LAPP
Laboratoire d'Annecy de Physique des Particules
GPU Architecture
DRAM (HBM3)
Streaming Multiprocessor
SM
L2 Cache
DRAM (HBM3)
Pierre Aubert, GPU architecture
2

# Streaming Multiprocessor




LAPP
Laboratoire d'Annecy de Physique des Particules
Streaming Multiprocessor H100
L1 Instruction Cache
L0 Instruction Cache
register file
warp scheduler
dispatch unit
Tensor Core
Classic Computing
+ - × /
INT32
FP32
FP64
Matrix Multiply
Tensor Core
LD/ ST
SFU
Tensor Memory Accelerator
L1 Data Cache / Shared Memory
Tex

## Compute terminology translation guide

| CPU (with an x86 bias) | NVidia CUDA | Khronos (OpenCL, SYCL, Vulkan…) |
|---|---|---|
| SIMD lane | CUDA core | Processing Element |
| SIMD vector | Warp | Subgroup |
| Simultaneous Multi-Threading / Hyper-Threading | Hardware multithreading | *Not named, hidden inside PE/CU model* |
| Advanced Matrix eXtensions, Neural Processing Unit | Tensor core | *No standard name (only usable via Vulkan vendor extensions)* |
| *No equivalent, ray tracing is not special* | RTX core | *Not named, exposed via VK_KHR extensions* |
| Intel Data Streaming Accelerator | Tensor Memory Accelerator (kind of [1]) | *Not named, exposed via OpenCL async copy API* |
| Core | Streaming Multiprocessor | Compute Unit |
| Core complex / Sub-NUMA cluster | Thread block cluster (kind of [2]) | *Not exposed yet* |
| NUMA node | Device | Device |
| SIMD registers, with manual [3] spill to caches/RAM | Local memory, with automatic spill to global memory | Private memory, with automatic spill to global memory |
| L1/L2 cache, with automatic spill to lower caches/RAM | Shared memory | Local memory |
| *No equivalent, all memory is writable* | Constant memory | Constant memory |
| *No equivalent, images are not special* | Texture memory | *Not named, exposed via image API* |
| Cache hierarchy + RAM | Global memory | Global memory |
| SIMD lane | Threads within the same warp | Work-items within the same subgroup |
| SIMD vector | Warp | Subgroup |
| Thread | Threads within the same thread block | Work-items within the same work-group |
| *No equivalent, all threads can synchronize with each other* | Thread block | Work-group |
| *No equivalent, all threads can synchronize with each other* | Thread block cluster | *Not exposed yet* |
| *No equivalent, threads are independent from each other* | Grid | NDrange |
| *No equivalent, can spawn threads anytime* | Stream | Command queue |
| *No equivalent, can spawn threads anytime* | Graph | Command buffer *(only exposed in Vulkan)* |
| Buffer | Buffer | Buffer |
| *No equivalent, images are not special* | Texture | Image |
| *No equivalent, images are not special* | Texture | Sampler |
| RAM | Memory | Device memory |
| *No equivalent, all useful memory is reachable* | Unified memory | Shared memory |
| *No equivalent, all useful memory is reachable* | Host/pinned memory | Host memory |

[1] Both TMA and DSA allow offloading memory copy work from the compute cores, but Intel DSA is more focused on device DMA and Nvidia TMA on VRAM-cache copies.
[2] On a hardware level, it's the same idea, but CUDA only allows some forms of synchronization when threads belong to the same thread block cluster.
[3] Speaking from the perspective of machine code here. Of course, the compiler of most programming langages will automate it for you.

# GPU setup : when developpers get their hands dirty

```
$ nvidia-smi
Fri Jun 13 12:35:50 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.35.05              Driver Version: 560.35.05      CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA RTX A4000 Laptop GPU    On  |   00000000:01:00.0 Off |                  N/A |
| N/A   61C    P8             17W /   90W |      18MiB /   8192MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A      3899      G   /usr/lib/xorg/Xorg                             4MiB |
+-----------------------------------------------------------------------------------------+
```

- good kernel module : `/usr/sbin/modinfo nvidia-current`
- ...avec le bon noyau : `uname -a`
- ...compatible avec la version CUDA : `nvidia-smi`
- ...compatible avec la *compute capability* : `nvidia-smi --query-gpu=name,compute_cap --format=csv`

# nvfortran do concurrent

```
subroutine evolve
  use hdf5_utils, only: write_slice
  integer :: i, j, iter
  real (pr) :: lap_u, lap_v
  integer :: h5error

  do iter = 1,nsteps/2
  ! Apply the stencil kernel for the laplacian computation
     do concurrent(i=lbound (u, 1) + 1 : ubound (u, 1) - 1, j=lbound (u, 2) + 1 : ubound (u, 2)-1)
        !lap_u = 0
        !do sj = -1, 1
        !   do si = -1, 1
        !      lap_u = lap_u + stencil (si, sj) * U0 (i+si, j+sj)
        !   end do
        !end do
        lap_u = sum (stencil * U0 (i-1:i+1, j-1:j+1))
        U1(i, j) = U0 (i, j) + dt * (Diffusivity_u * lap_u - U0 (i, j) * V0 (i, j)**2 + Feed_Rate * (1.0_pr - U0 (i,
        lap_v = sum (stencil * V0 (i-1:i+1, j-1:j+1))
        V1(i, j) = V0 (i, j) + dt * (Diffusivity_v * lap_v + U0 (i, j) * V0 (i, j)**2 - (Feed_Rate + Kill_Rate) * V0
     end do
```

```
$ nvidia-smi --query-gpu=name,compute_cap --format=csv
name, compute_cap
NVIDIA RTX A4000 Laptop GPU, 8.6
$ nvfortran -g -O4 -stdpar=gpu -Minfo=accel -fast -Wall -Mdclchk
                radiative_forcing.f90 -o radiative_forcing_nv_gpu && time ./radiative_forcing_nv_gpu
 Total earth surface flux in wavelength range:    389.1279304077366      W/m^2
 Total outgoing flux at the top of the atmosphere:    336.3353560925743       W/m^2
 Total earth surface flux in wavelength range:    389.1279304077366      W/m^2
 Total outgoing flux at the top of the atmosphere:    331.5356807812431       W/m^2

real 0m1,480s
user 0m0,960s
sys 0m0,387s
```

⇒ On n'a pas gagné grand'chose, mais la taille du calcul ne s'y prête pas

```
$ time nvc++ -g -std=c++20 -O4 -stdpar=gpu -Minfo=accel -fast -Wall
          -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx
               radiative_forcing.TRANSFORM.cpp -ltbb -o radiative_forcing_nv++ && time ./radiative_forcing_nv++
"radiative_forcing.TRANSFORM.cpp", line 159: warning: function "lambda []" captures local object "emitted_flux" by reference, will likely cause an illegal memor
                         [&](unsigned j) {
                            ^

"radiative_forcing.TRANSFORM.cpp", line 137: warning: function "lambda []" captures local object "n_CO2" by reference, will likely cause an illegal memory acces
                          [&n_CO2, i](double lambda) {
                                     ^

"radiative_forcing.TRANSFORM.cpp", line 154: warning: function "lambda []" captures local object "temp" by reference, will likely cause an illegal memory access
                          [&](double lambda, double tau) -> double {
                             ^


real 1m13,468s
user 1m12,410s
sys 0m0,999s
Total earth surface flux in wavelength range: 389.127 W/m^2
terminate called after throwing an instance of 'thrust::_V_300201_SM_86_NVHPC::system::system_error'
  what():  transform: failed to synchronize: cudaErrorIllegalAddress: an illegal memory access was encountered
Abandon

real 0m0,398s
user 0m0,072s
sys 0m0,323s
```

⇒ Ouch !

```
$ time nvc++ -g -std=c++20 -O4 -stdpar=gpu -Minfo=accel -fast -Wall
          -mavx512bitalg -mavx512bw -mavx512cd -mavx512dq -mavx512f -mavx512ifma -mavx512vbmi -mavx512vbmi2 -mavx512vl -mavx512vnni -mavx512vp2intersect -mavx
               radiative_forcing.FOREACH.cpp -ltbb -o radiative_forcing_nv++ && time ./radiative_forcing_nv++
"radiative_forcing.FOREACH.cpp", line 139: warning: function "lambda []" captures local object "temp" by reference, will likely cause an illegal memory access wh
                          [&](unsigned j) {
                             ^


real 1m3,399s
user 1m2,409s
sys 0m0,945s
Total earth surface flux in wavelength range: 389.127 W/m^2
terminate called after throwing an instance of 'thrust::_V_300201_SM_86_NVHPC::system::system_error'
  what():  parallel_for: failed to synchronize: cudaErrorIllegalAddress: an illegal memory access was encountered
Abandon

real 0m0,358s
user 0m0,091s
sys 0m0,261s
```

⇒ Ouch !

# Effet de serre

$$F_{\text{sortant}} = F_{\text{entrant}} - \kappa \Delta z F_{\text{entrant}} + \kappa \Delta z \pi B(\lambda, T)$$

où :

- $F_{\text{sortant}}$ est le flux sortant de la couche,
- $F_{\text{entrant}}$ est le flux entrant dans la couche,
- $\kappa$ est le coefficient d'absorption,
- $\Delta z$ est l'épaisseur de la couche,
- $B(\lambda, T)$ est la fonction de Planck pour la luminance spectrale d'un corps noir à une température $T$ et une longueur d'onde $\lambda$.

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \frac{1}{e^{\frac{hc}{\lambda k_B T}} - 1}$$

$$\frac{dF}{dz} = -\kappa F + \kappa \pi B(\lambda, T)$$

où $F$ est le flux radiatif.
⇒ Équa. Diff.
⇒ À ne jamais résoudre par une méthode d'Euler !

⇒ Modèle d'atmosphère minimaliste...

- le plus dur reste d'exprimer le parallélisme
- ... assez technique pour C++
- ⇒ commencez avec des exemples élémentaires, voire triviaux

  Produits de HADAMARD *cf* Gray Scott School
- même quand on a bien exprimé le parallélisme
  l'expression peut déborder de la mémoire GPU (pas aujourd'hui)
- ⇒ compromis entre la performance, la portabilité et la lisibilité

BILAN : c'est chouette, mais...