The GPU code FARGO3D: presentation and implementation strategies

Frédéric Masset

Universidad Nacional Autónoma de México (UNAM)

Pablo Benítez-Llambay (UC, Argentina & NBI Copenhagen),
David Velasco (UNAM & ICS Zürich)

Overview

FARGO3D is a hydrodynamics/MHD code based on upwind methods on a staggered mesh.

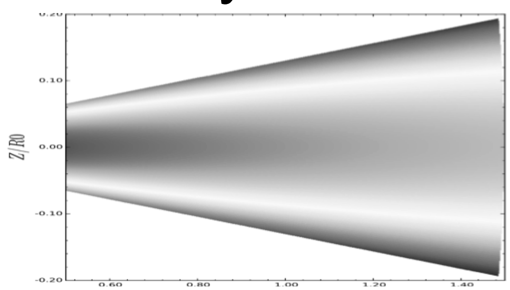
It is a complete rewrite (from scratch) of the FARGO code.

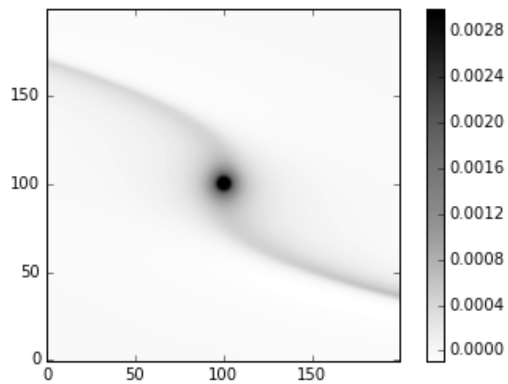
Its primary focus are protoplanetary discs and planet-disc interactions.

It can run on (clusters of) GPUs.

Wish list of properties of a code dedicated to pp discs and planet-disc interactions

Correctly account for the disc's hydrostatic equilibrium and rotational equilibrium.





Typical density response near an embedded planet. Corotates with the planet.

Must correctly describe steady flows with source terms

Wish list of properties of a code dedicated to pp discs and planet-disc interactions

Other important aspects:

➔ Advection of potential vorticity

➔ Advection of entropy

➔ Shocks should be properly handled as well, but they do not have a direct impact on the tidal force on the planet.

We also need speed, whereas memory is not too much a concern.

Why GPUs ?

Simulations of planet-disc interactions are generally compute bound rather than memory bound.

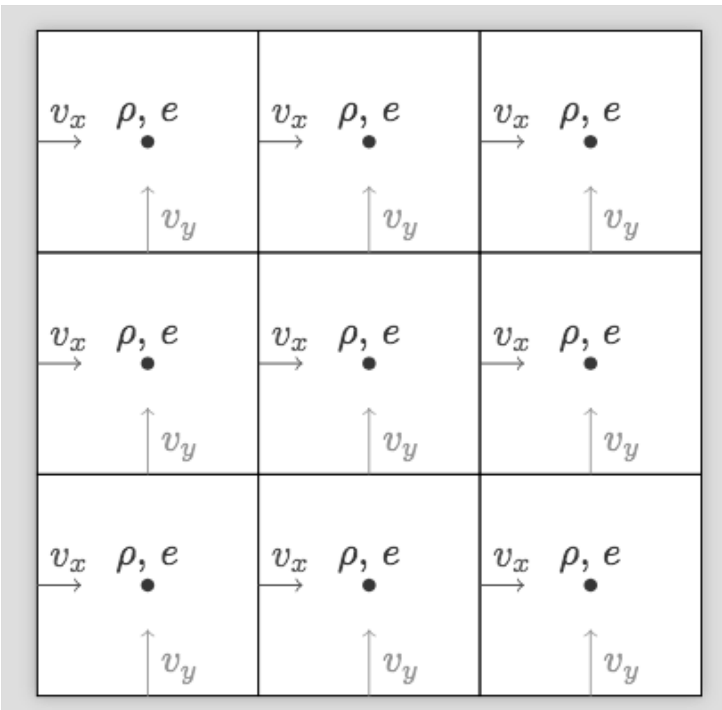
Most of the HD or MHD substeps are local (the updated value of a cell depends on itself and its neighbors).

➔ Ideally suited for GPUs

Many planet-disc interaction studies imply explorations of parameter space

➔ Very well suited to clusters of GPUs.

Main features of the code



Solves the HD equations on a staggered mesh.

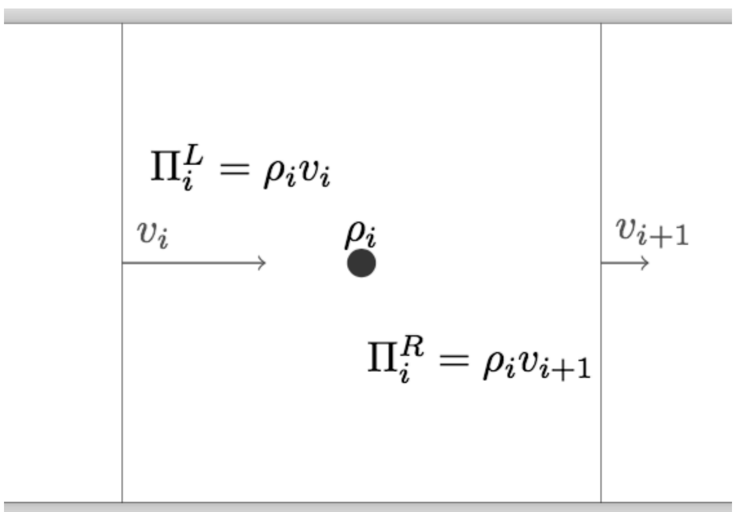
The mesh can be either Cartesian,
cylindrical or spherical

Possibly in a rotating frame with variable rotation speed

Main features of the code

Most of the code structure is similar to that of the ZEUS code (Stone & Norman 92): upwind method using van Leer's slopes, dimensionally split solver)

The momenta advection is different.



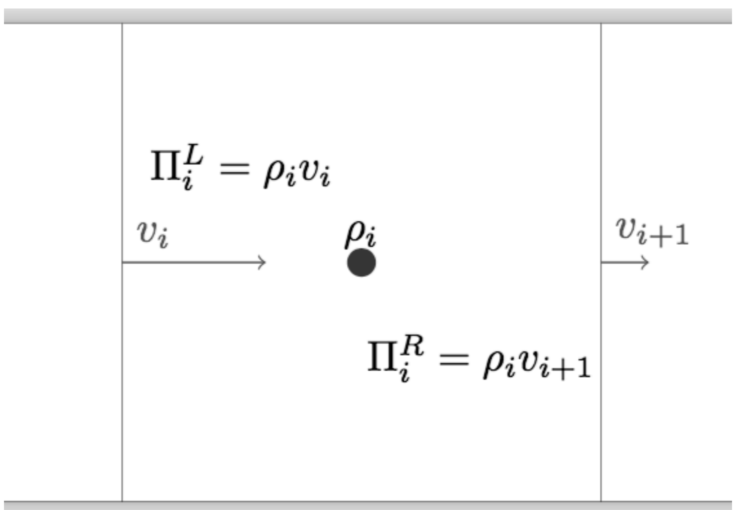
For each dimension we define two momenta (left & right momentum) transported like any cell-centred quantity.

$$v_i = \frac{\Pi_i^L + \Pi_{i-1}^R}{\rho_{i-1} + \rho_i}$$

Main features of the code

This technique allows to have the same control volume (the cell boundaries) for all quantities.

➔ Orbital advection (aka FARGO: *Fast Advection In Rotating Gaseous Objects*) is trivial to implement.



For each dimension we define two momenta (left & right momentum) transported like any cell-centred quantity.

Conservation properties

Mass is conserved

Momentum (defined as arithmetic average of $\prod^L$ and $\prod^R$) is conserved.

Unlike in codes based on Riemann solvers, the momenta fluxes do not include the pressure, which is dealt with in a separate source term. The way it is implemented, however, implies that momentum is conserved.

The implementation is « as conservative as possible », so as to leave as few terms as possible to source substeps.
➔ Conservation of angular momentum even when the frame is rotating.

Conservation properties

For isothermal setups (*i.e.* setups with a fixed temperature, possibly depending on the position), mass and momentum conservation imply that :

➔ Shocks are correctly described (they fulfill Rankine-Hugoniot relationships).

➔ The correct amount of vortensity is produced across the shocks.

Conservation properties

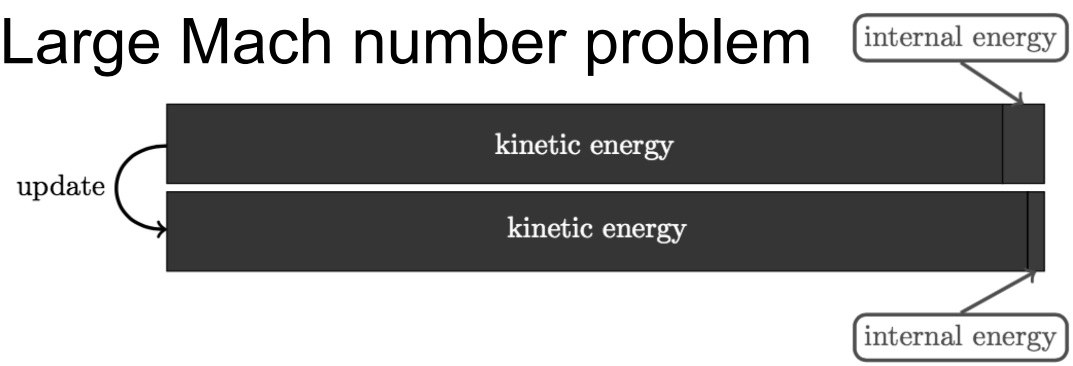
When solving the energy equation (*i.e.* in non-isothermal setups), we use an equation on the internal energy.

➔ No enforcement of the conservation of total energy.

Note: pp discs are thin: H/r ~ a few percent.

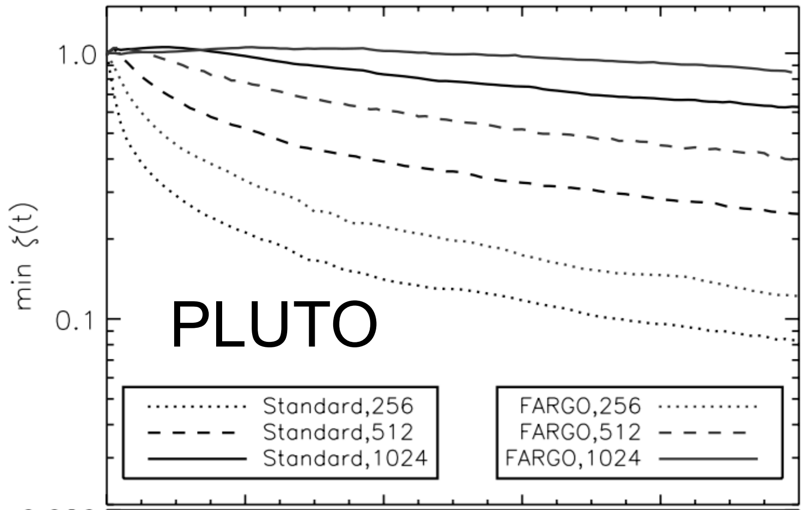
$$\frac{c_s}{v_k} = \frac{H}{r}$$ ➔ Internal energy much smaller than kinetic energy

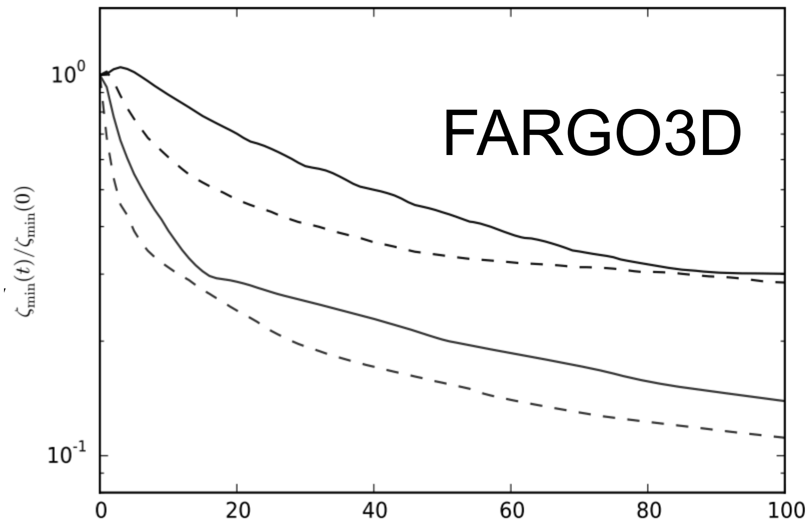
Large Mach number problem



Any truncation error on the kinetic energy is forcibly transferred to the internal energy, which compounds the relative error

About the wish list





No comparison with other schemes available on planet-disc interactions setups, so far.

Roughly similar behaviour between PLUTO and FARGO3D.

The flow on a staggered mesh can converge to a numerical steady state solution with source terms.

GPU implementation

Antecedent: GFARGO, a GPU-avatar of the former FARGO code. Entirely coded by hand in CUDA in 2009-2010. Made explicitly use of the shared memory of the SMPs, for each kernel.

➔ Large speed up wrt to CPU cores, comparable to those quoted at the time by NVIDIA

➔ Never again. Direct programming in CUDA is tedious, error prone, hardly maintainable. Non-editable black box for non CUDA proficient users.

GPU implementation strategy

- We developped a Python script that automatically translates our C functions to CUDA kernels (part of PhD of Pablo Benítez-Llambay).
- User only has to code in C (with a few helper, html like comments) ➔ code runs on GPU
- CPU-GPU (D2H/H2D) transfers dealt with automatically (INPUT/OUTPUT directives…)
- Side effect: low memory footprint (21 fields in 3D HD, 27 fields in 3D MHD).

Translation example

C file (preamble)

```
void VanLeerX_a_cpu(Field *Q){

//<USER_DEFINED>
  INPUT(Q);
  OUTPUT(Slope);
//<\USER_DEFINED>

//<EXTERNAL>
  real* slope = Slope->field_cpu;
  real* q = Q->field_cpu;
  int pitch  = Pitch_cpu;
  int stride = Stride_cpu;
  int size_x = XIP;
  int size_y = Ny+2*NGHY;
  int size_z = Nz+2*NGHZ;
  real dx = Dx;
//<\EXTERNAL>

//<INTERNAL>
  int i;
  int j;
  int k;
  int ll;
  int llxm;
  int llxp;
  real dqm;
  real dqp;
//<\INTERNAL>
```

Management of D ⇔ H transfers

C++ wrapper

```
extern "C" void VanLeerX_a_gpu(Field *Q) {

  INPUT(Q);
  OUTPUT(Slope);

  dim3 block (BLOCK_X, BLOCK_Y, BLOCK_Z);
  dim3 grid ((Nx+2*NGHX+block.x-1)/block.x,
             ((Ny+2*NGHY)+block.y-1)/block.y,
             ((Nz+2*NGHZ)+block.z-1)/block.z);

#ifdef BIGMEM
#define xmin_d &Xmin_d
#define ymin_d &Ymin_d
#define zmin_d &Zmin_d
#define Sxj_d &Sxj_d
#define Syj_d &Syj_d
#define Szj_d &Szj_d
#define Sxk_d &Sxk_d
#define Syk_d &Syk_d
#define Szk_d &Szk_d
#define InvVj_d &InvVj_d
#endif

  CUDAMEMCPY(xmin_s, xmin_d, sizeof(real)*(Nx+1), 0, cudaMemcpyDeviceToD
  CUDAMEMCPY(ymin_s, ymin_d, sizeof(real)*(Ny+2*NGHY+1), 0, cudaMemcpyDe
  CUDAMEMCPY(zmin_s, zmin_d, sizeof(real)*(Nz+2*NGHZ+1), 0, cudaMemcpyDe


  cudaFuncSetCacheConfig(VanLeerX_a_kernel, cudaFuncCachePreferL1 );
  VanLeerX_a_kernel<<<grid,block>>>(Slope->field_gpu,
                                    Q->field_gpu,
                                    Pitch_gpu,
                                    Stride_gpu,
                                    XIP,
                                    Ny+2*NGHY,
                                    Nz+2*NGHZ,
                                    Dx);

  check_errors("VanLeerX_a_kernel");

}
```

Translation example

C file (main loop)

```
#ifdef Z
  for (k=0; k<size_z; k++) {
#endif
#ifdef Y
    for (j=0; j<size_y; j++) {
#endif
#ifdef X
      for (i=XIM; i<size_x; i++) {
#endif
//<#>
        ll = l;
        llxm = lxm;
        llxp = lxp;

        dqm = (q[ll]-q[llxm]);
        dqp = (q[llxp]-q[ll]);
        if(dqp*dqm<=0.0)  slope[ll] = 0.0;
#ifndef DONOR
        else  slope[ll] = (2.*dqp*dqm) /
                ((dqm+dqp)*(zone_size_x(j,k)));
#else
        else  slope[ll] = 0.0;
#endif
//<\#>
#ifdef X
      }
```

Loops are converted to thread arithmetics and limits check.

Cell indices are macrocommands which expand differently in the CPU and GPU versions (for alignment and coalesced transactions)

CUDA file (main part)

```
#ifdef X
  i = threadIdx.x + blockIdx.x * blockDim.x;
#else
  i = 0;
#endif
#ifdef Y
  j = threadIdx.y + blockIdx.y * blockDim.y;
#else
  j = 0;
#endif
#ifdef Z
  k = threadIdx.z + blockIdx.z * blockDim.z;
#else
  k = 0;
#endif

#ifdef Z
  if(k>=0 && k<size_z) {
#endif
#ifdef Y
    if(j>=0 && j<size_y) {
#endif
#ifdef X
      if(i<size_x) {
#endif
        ll = l;
        llxm = lxm;
        llxp = lxp;

        dqm = (q[ll]-q[llxm]);
        dqp = (q[llxp]-q[ll]);
        if(dqp*dqm<=0.0)  slope[ll] = 0.0;
#ifndef DONOR
        else  slope[ll] = (2.*dqp*dqm) /
                ((dqm+dqp)*(zone_size_x(j,k)));
#else
        else  slope[ll] = 0.0;
#endif
#ifdef X
```

Technical notes on CUDA implementation

We have striven for the best efficiency of the CUDA code generated:

- *Automatic CUDA block size optimization for each kernel*
- *Direct GPU-GPU communications if built with a CUDA aware version of MPI (e.g. OpenMPI >= 1.7, MVAPICH2)*
- ***ALL** the calculations are performed on the GPU (even boundary conditions)*

A brief timeline

April 2014: CUDA 6.0 release (unified memory).

June 2013: openacc 2.0 release

Nov 2011 to Feb 2014: FARGO3D development

Advantages of automatic translation

Only reduction kernels (CFL, diagnostics) have been implemented manually once for all. All other kernels are produced automatically at build time. No need for third party libraries (for the modules of the public version).

In principle, translating the code to OpenCL instead of CUDA is possible by simply changing the translation script (+ need to write a reduction function manually).

We have a full control of transactions between the host and device (CPU & GPU). No « under the hood » transactions.

Drawbacks of our implementation

Our naming conventions much obey strict rules, and are specific to our project.

Our setups must fit in the GPU memory (they cannot be transferred by chunks).

It works well for routines (kernels) for which the output is a local function of other fields (most of HD/MHD routines). For other cases one has to program in CUDA or resort to libraries.

Contributors

Public version

Main module (publicly available at http:fargo.in2p3.fr):
P. Benítez-Llambay (main developper), F. Masset

Git repository

Multifluid capability
P. Benítez-Llambay, L. Krapp

Non-ideal MHD
L. Krapp

Nested meshes
D. Velasco

Simplified RT (FLD)
F. Masset (w/ input from J. Szulágyi)

Planetary heating
H. Eklund

Enhancements specific to planet-disc interactions
P. Benítez-Llambay

Some numbers

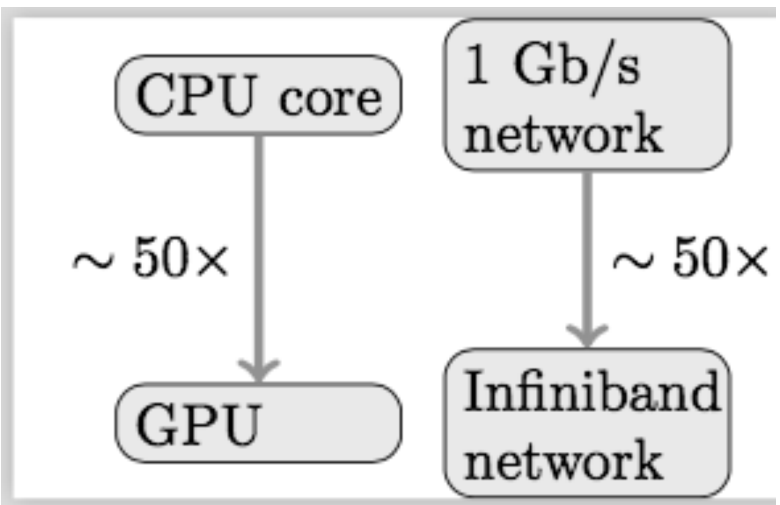
Throughputs on K20, with ECC on

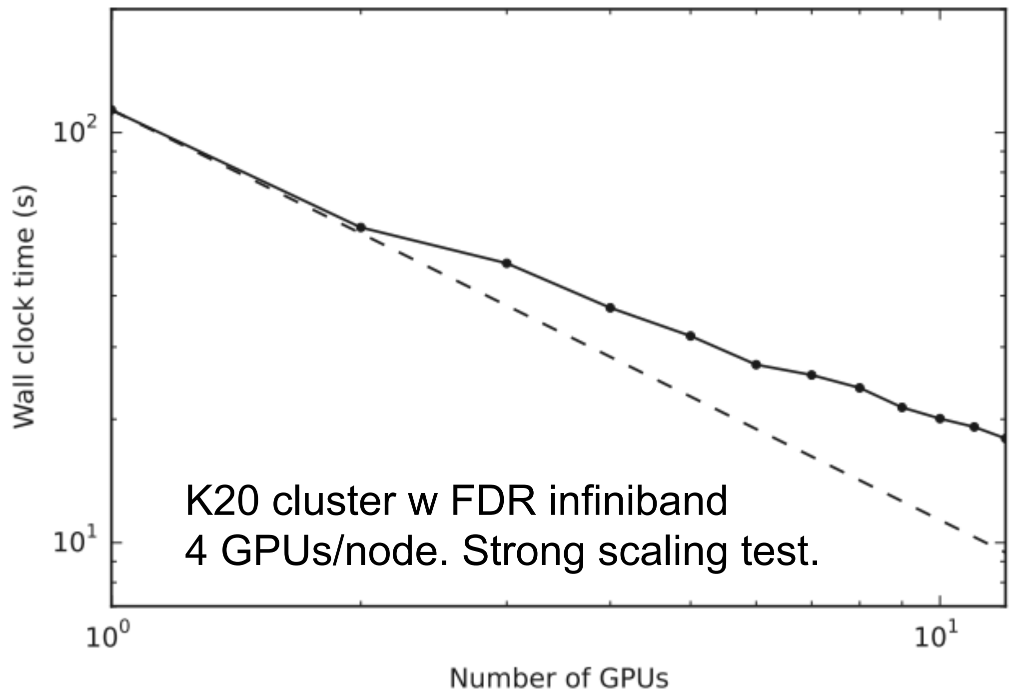
2D polar setup, isothermal	36 Mcell/s, 138 bytes/cell
3D spherical setup, isothermal	20.4 Mcell/s, 168 bytes/cell
3D spherical setup, adiabatic	17 Mcell/s, 168 bytes/cell

Throughputs on P100 are 4x larger

Scaling with MPI

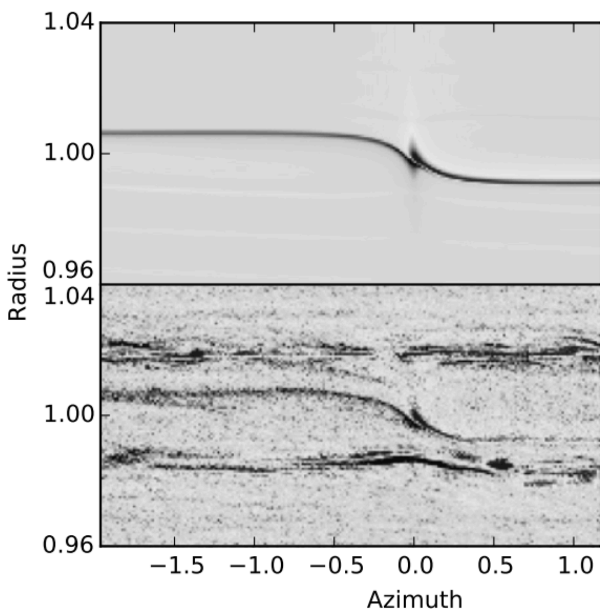
Scaling efficiency depends on the computational throughput to bandwidth ratio.

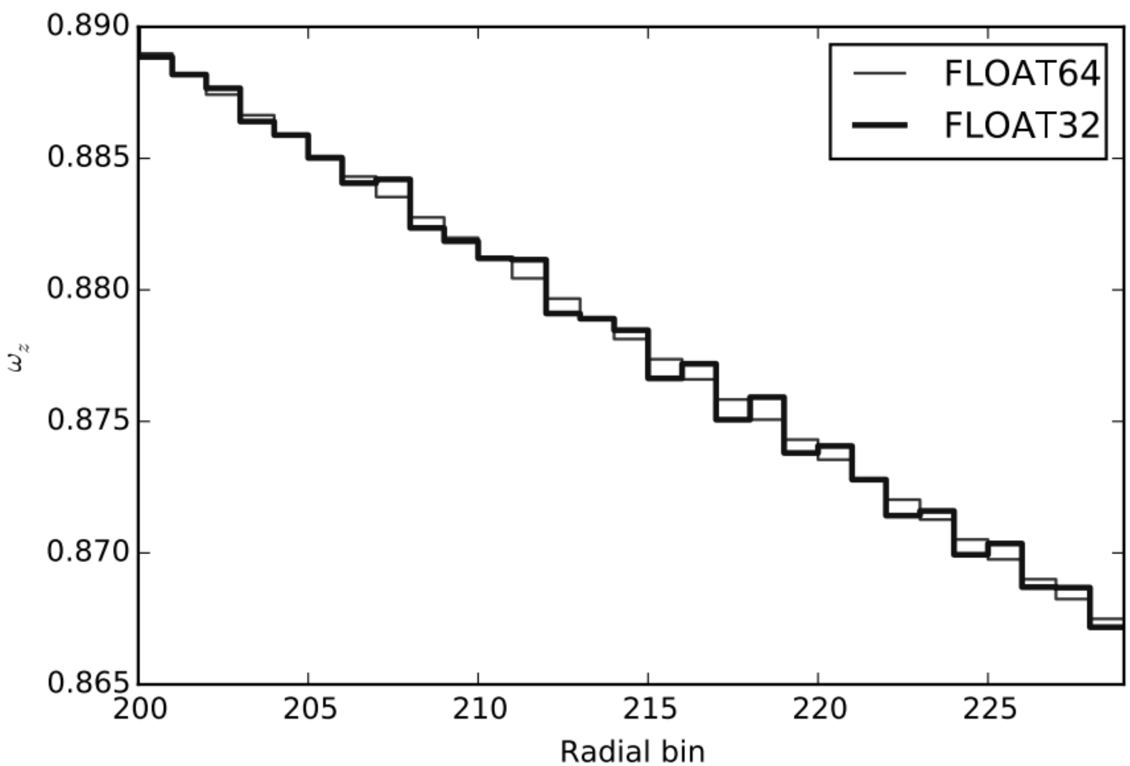




About single precision

Most low-end GPUs have very limited double precision compared to their HPC counterpart (*e.g.* GTX1080 vs P100). Is single precision suitable for simulations of pp discs ?





Low accuracy: many local extrema of PV, subject to RWI