



gPLUTO

GPUs for the PLUTO code

M. Rossazza¹, V. Berta¹, A. Mignone¹, M. Bugli^{1,2}, G. Mattia³, S. Truzzi¹, A. Suriano¹

¹Physics Department, University of Torino

²IAP-CNRS, Paris

³Max Planck Institute for Astronomy (MPIA), Heidelberg

gPLUTO is the new GPU-enabled version of the PLUTO code.

We are going to discuss:

- The rise and principles of GPU computing;
- Features of OpenACC;
- The GPU-porting process of gPLUTO;
- Recent performance results.

Still me...
But
gBETTER!

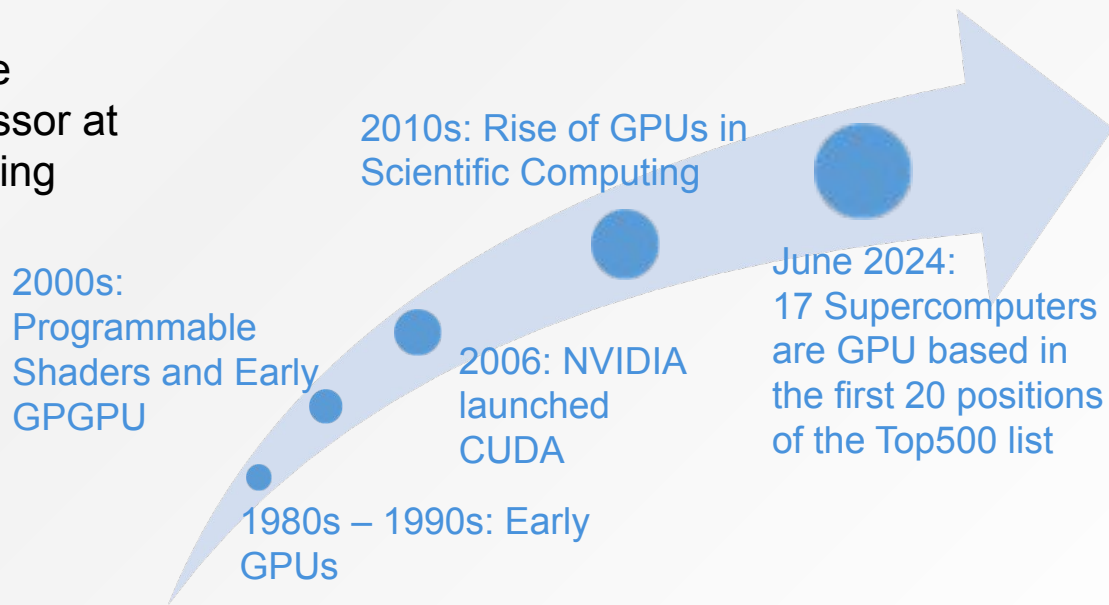


Rise of GPUs



Parallel scaling has largely replaced frequency as the primary way to improve computational performance

Multi-core architecture perform more operations than a single-core processor at the same clock frequency by executing tasks concurrently.



GPU architecture



A modern GPU architecture is designed to handle massive parallel computations by leveraging thousands of small, efficient **cores**.

These cores are organized into **Streaming Multiprocessors (SMs)**, which can execute thousands of threads simultaneously.

NVIDIA Ampere architecture:
GA100 Full GPU with 128 SMs.

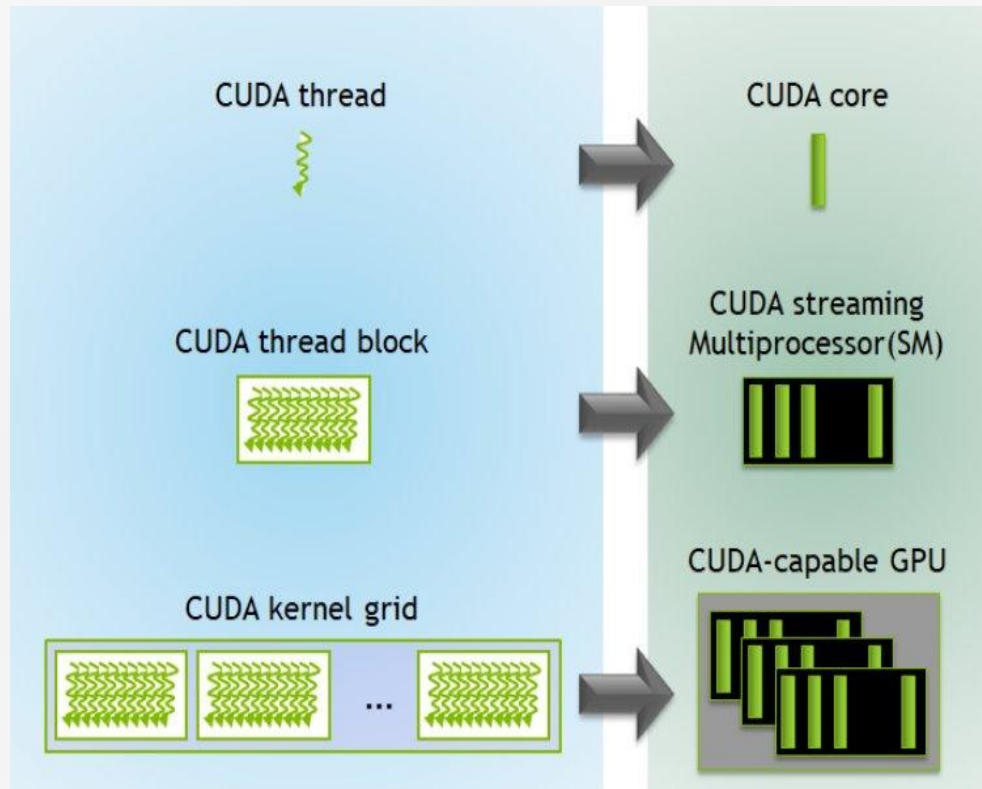


GPU keywords

A **kernel** is a portion of the code that runs on the GPU.

A **thread** is a single stream of program instructions, defined in a kernel, that runs in parallel with other threads and are mapped to cores.

Occupancy is determined by the amount of memory required by each thread.



OpenACC is the programming model chosen for the GPU-porting.

- High-level;
- Requires few changes to the code;
- Directive-based.

The pragma acc parallel loop is a **compute directive** used to execute a loop on the GPU

```
#pragma acc parallel loop
for (int i = 0; i < N; i++) {
    // ... Loop body
}
```

```
#pragma acc enter data copyin (A, B)
// ... A and B are used in one or more kernels
#pragma acc exit data delete (A, B)
```

The pragma enter data copyin directive is a **data directive** used to explicitly transfer data from the CPU memory to the GPU memory.

GPU computing keypoints



Data Locality: reduce data movement between CPU and GPU memory as much as possible.
all computational part of the program should reside in GPU memory !

Private Variables: GPU threads should perform identical operations but on different memory addresses.

Without precautions, simultaneous operations are performed at the same memory address leading to incorrect results.

```
int V[8];
int A[NX][NY][NZ];

#pragma acc parallel loop collapse(3) private(V[:8])
for (i = 0; i < NX; i++){
  for (j = 0; j < NY; j++){
    for (k = 0; k < NZ; k++){

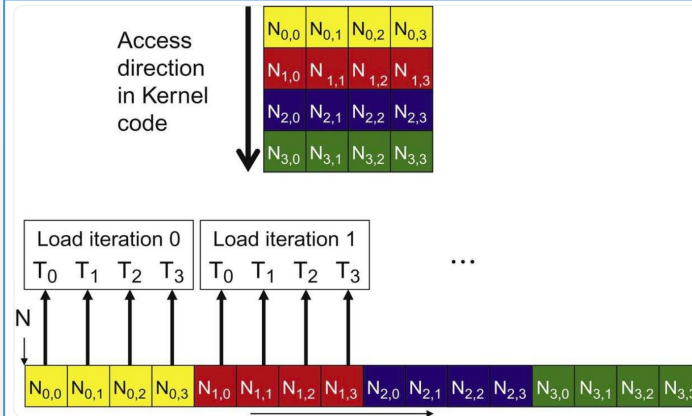
      A[i][j][k] *= 2.0;

      V[0] = ...;
      V[1] = ...;
      ...
    }
  }
}
```


GPU computing keypoints

Coalesced Memory Access: consecutive threads access consecutive memory addresses. Memory coalescing is a technique which allows optimal usage of the global memory bandwidth.

Requires different array ordering so that the inner loop we're accelerating should be also the fastest index of the multidimensional array as in this example.



Access direction in Kernel code

Load iteration 0: T_0, T_1, T_2, T_3

Load iteration 1: T_0, T_1, T_2, T_3

...

```
#pragma acc parallel loop vector
for (i = ibeg; i <= iend; i++){
#pragma acc loop seq
for (nv = 0; nv < NVAR; nv++) {

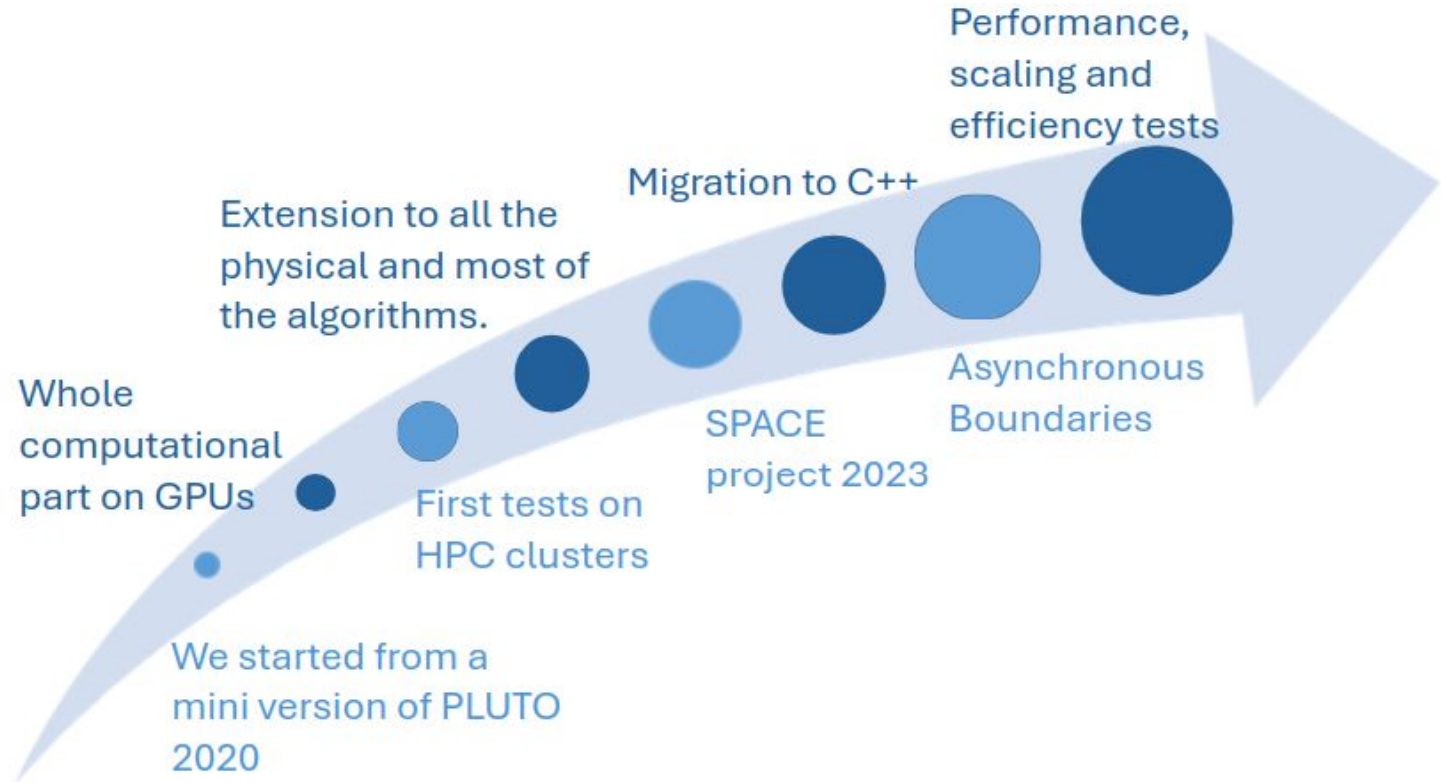
    // MUST REVERSE INDICES HERE:

    v[i][nv] *= 2; ➡ v[nv][i] *= 2;
}}

Using C++ templates:    v[nv][i] ➡ v(i,nv)
```



gPLUTO development



An example kernel



```
#pragma acc parallel loop collapse(2)
for (k = KBEG; k <= KEND; k++){
for (j = JBEG; j <= JEND; j++){
    #pragma acc loop
    for (i = 0; i < NX1_TOT; i++){
    for (nv = 0; nv < NVAR; nv++){
        vc[i][nv] = Vc[nv][k][j][i];
    }}

Reconstruct (...);
Riemann (...);
Powell (...);

#pragma acc loop
for (i = IBEG; i <= IEND; i++){
#pragma acc loop
for (nv = 0; nv < NVAR; nv++){
    dU[k][j][i][nv] = -dtdx*(flux[i][nv] - flux[i-1][nv]) + dt*src[i][nv];
}
Cdt[k][j][i] = 0.5*(cmax[i-1] + cmax[i])/grid->dx1;
}
}}
```



```
#pragma acc parallel loop collapse(2)
for (k = KBEG; k <= KEND; k++){
for (j = JBEG; j <= JEND; j++){
    long int offset1 = NVAR*offset;
    Ary2D vc(&data->sweep.vC[offset1],ni,NVAR);

    #pragma acc loop vector
    for (i = 0; i < NX1_TOT; i++){
        ...
        #pragma acc loop seq
        for (nv = 0; nv < NVAR; nv++) vc(i,nv) = Vc(i,j,k,nv);
    }
}}

Reconstruct3D<IDIR> (...);

Riemann3D (...);

RightHandSide3D (...);

RightHandSideSource3D (...);

#pragma acc parallel loop collapse(3)
for (nv = 0; nv < NVAR; nv++){
for (k = KBEG; k <= KEND; k++){
for (j = JBEG; j <= JEND; j++){
    long int offset1 = NVAR*offset;
    Ary2D rhs(&data->sweep.rhs[offset1],ni,NVAR);

    #pragma acc loop vector
    for (i = IBEG; i <= IEND; i++) dUc(i,j,k,nv) = rhs(i,nv);
}}}
```

C++ features - Classes



From pointer-to-pointer
approach...

```
data->Uc = Array4D(ntot[KDIR], ntot[JDIR], ntot[IDIR], NVAR, double);

char ****Array4D (int nx, int ny, int nz, int nv, size_t dsize)
{

m = (char ****) malloc ((size_t) nx*sizeof (char ***));
m[0] = (char ***) malloc ((size_t) nx*ny*sizeof (char **));
m[0][0] = (char **) malloc ((size_t) nx*ny*nz*sizeof (char *));
m[0][0][0] = (char *) malloc ((size_t) nx*ny*nz*nv*dsize);
...
}

Uc[k][j][i][nv];
```



...To classes with **single memory block, single pointer allocation**.
Index order can be easily modified
for optimal memory access.

```
data->Uc.create(ntot[IDIR],ntot[JDIR],ntot[KDIR],NVAR);

void create(int n0, int n1, int n2, int n3) {

    n0_=n0;n1_=n1;n2_=n2;n3_=n3;

    stride0_ = 1;
    stride1_ = n0_;
    stride2_ = n1_*n0_;
    stride3_ = n2_*n1_*n0_;

    total_size_ = n0_*n1_*n2_*n3_;
    data_ = new T[total_size_];
}

Uc(i,j,k,nv);
```

C++ features - Templated functions



templated functions allow for code specialization based on template parameters, in our case “dir”.

Templated parameters can facilitate optimal memory access for arrays that depend on these parameters.

```
template<int dir>
void HLLD_Solver (Data *d, timeStep *Dts, Grid *grid, RBox *box);

void HLLD_Solver_T (Data *d, timeStep *Dts, Grid *grid, RBox *box){
    if(g_dir==IDIR) HLLD_Solver<IDIR>(d, Dts, grid, box);
    else if(g_dir==JDIR) HLLD_Solver<JDIR>(d, Dts, grid, box);
    else if(g_dir==KDIR) HLLD_Solver<KDIR>(d, Dts, grid, box);
}

template<int dir>
void HLLD_Solver (Data *d, timeStep *Dts, Grid *grid, RBox *box)
{
    GEN_INDEX_CYCLE (dir, VX1, VXn, VXt, VXb);

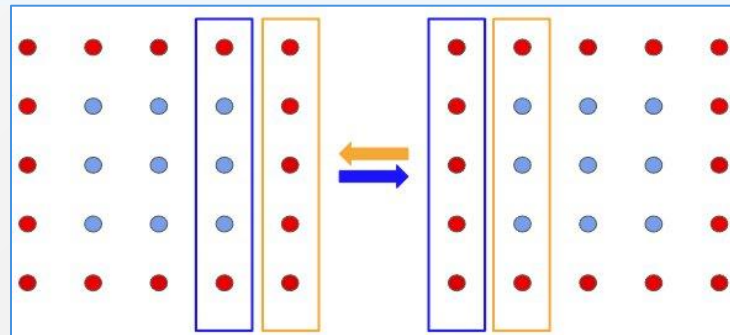
    #pragma acc parallel loop collapse(2)
    for (k = kbeg; k <= kend; k++){
        for (j = jbeg; j <= jend; j++){
            #pragma acc loop
            for (i = ibeg; i <= iend; i++){

                int v[NVAR];
                v[VXn] = ...;
            }
        }
    }
}
```

Multi-process implementation

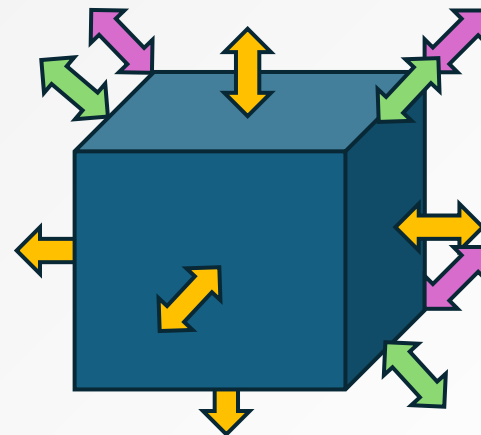


We implemented a new multi-process implementation routine based on **asynchronous** MPI calls, kernels and GPUmemory only arrays.



It requires up to 26 directions of data exchange per process:

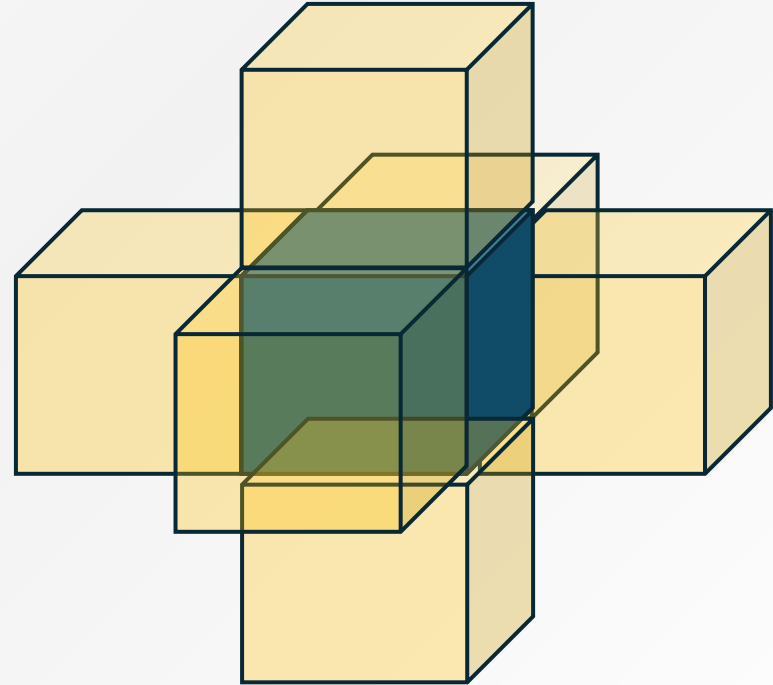
- 6 **faces**;
- 12 **edges**;
- 8 **corners**.



Multi-process implementation



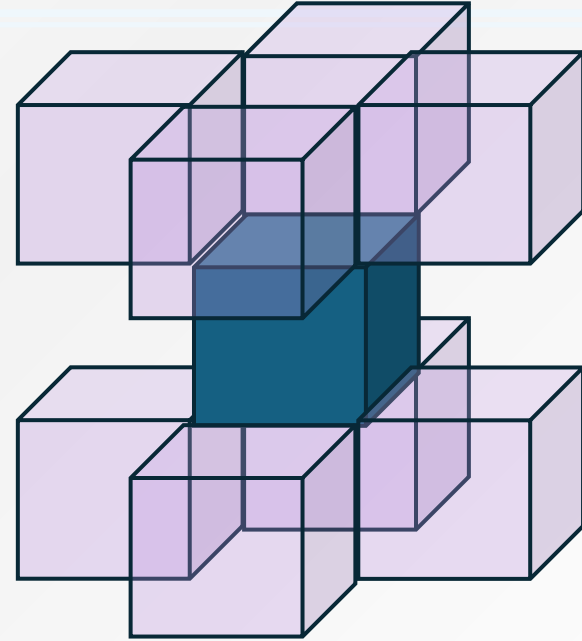
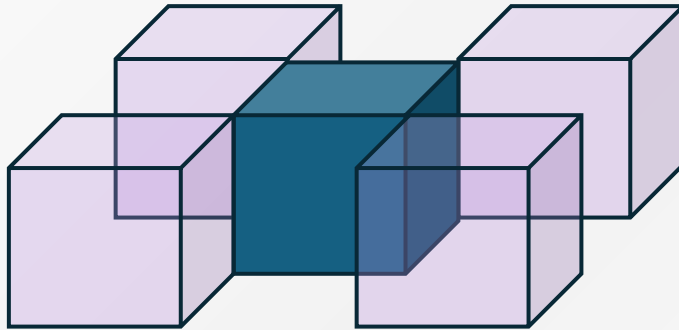
6 potential neighbour to
exchange points on **faces**
with



Multi-process implementation



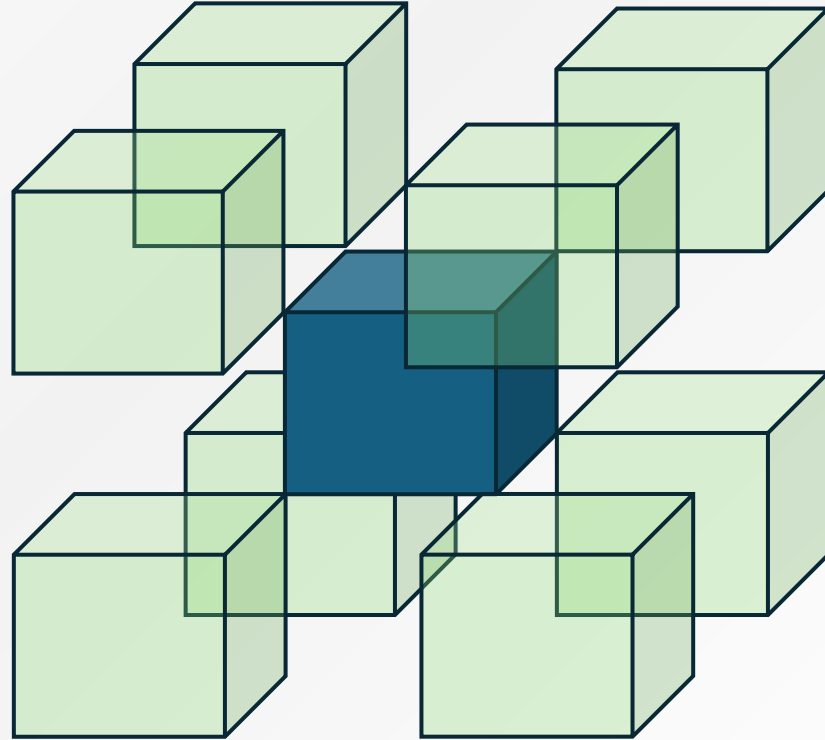
12 potential neighbour to
exchange points on **edges**
with



Multi-process implementation



8 potential neighbour to
exchange points on **corners**
with



3D Orszag-Tang test problem



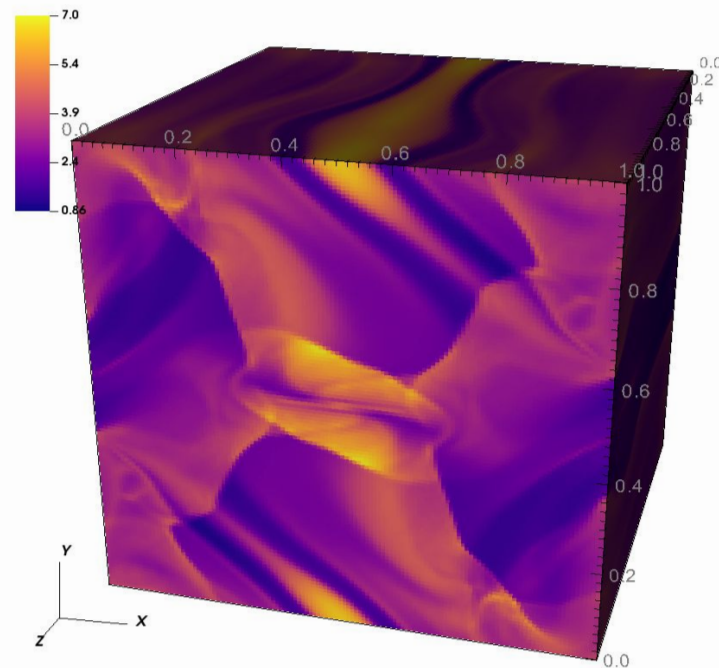
Unit box domain with $P = 5/3$, $\rho = 25/9$.

$$\mathbf{v} = -\varepsilon(z)\sin(2\pi y)\mathbf{e}_x + \varepsilon(z)\sin(2\pi x)\mathbf{e}_y + 0.2\sin(2\pi z)\mathbf{e}_z$$

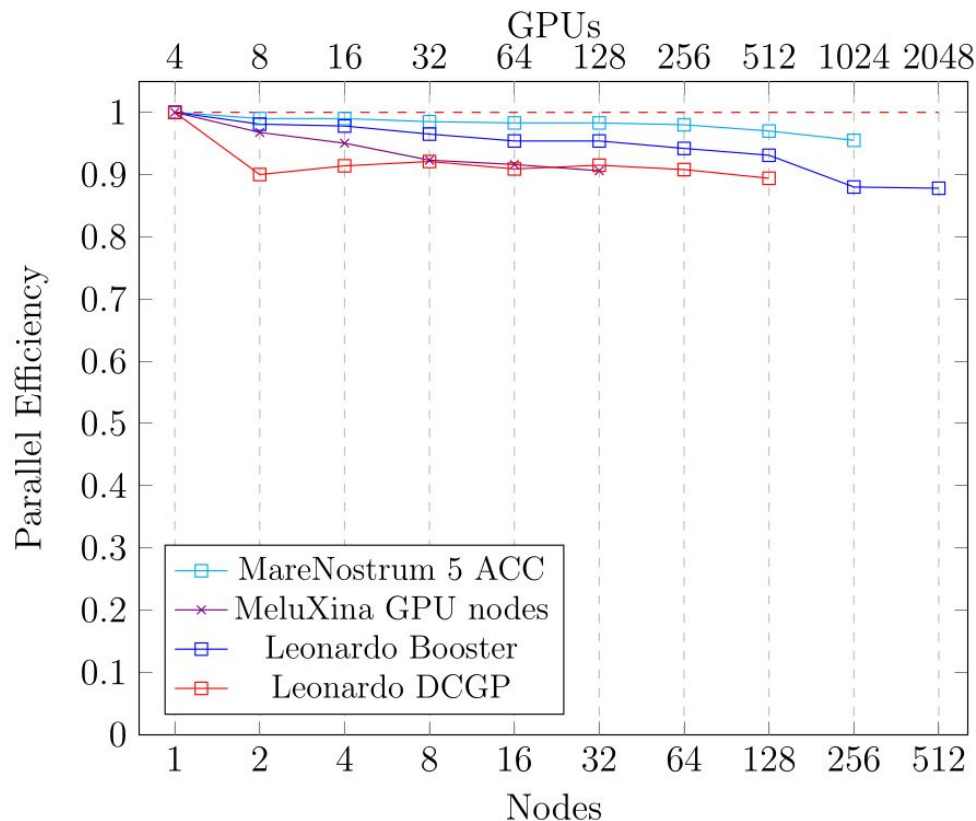
where: $\varepsilon(z) = 1 + \sin(2\pi z)/5$

$$\mathbf{B} = -B_0 [\sin(2\pi y)\mathbf{e}_x + \sin(4\pi x)\mathbf{e}_y], B_0 = 1/\sqrt{4\pi}$$

- Method: RK3 / WENOZ / HLLD Riemann solver;
- Final integration time $t_s = 1$.
- Constrained Transport for $\nabla \cdot \mathbf{B} = 0$ control

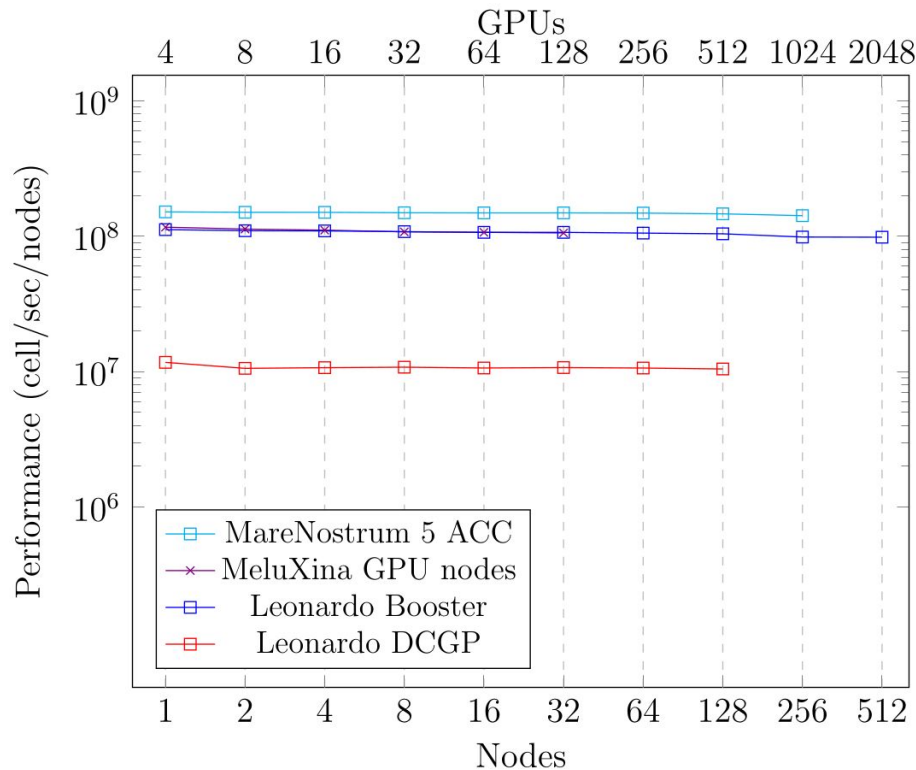


Weak Scaling on HPC clusters



System	Resolution per Node
Leonardo Booster and DCGP	704 x 704 x 352
MareNostrum 5 ACC	704 x 704 x 352
MeluXina GPU nodes	512 x 512 x 512


Speed-up and Performances



Nodes	T_{GPU_s} (sec)	T_{CPU_s} (sec)	Acceleration ($T_{\text{CPU}_s}/T_{\text{GPU}_s}$)
1	312	2982	9.55
2	318	3300	10.38
4	319	3263	10.23
8	323	3236	10.02
16	327	3281	10.03
32	327	3257	9.96
64	331	3283	9.92
128	335	3336	9.96

Future work



- Full OpenMP integration. 
- Adaptive Mesh Refinement implementation.
- Test the scaling on higher number of nodes



Thank you!

