CERFACS

CENTRE EUROPÉEN DE RECHERCHE ET DE FORMATION AVANCÉE EN CALCUL SCIENTIFIQUE

Porting a legacy Fortran CFD HPC code to GPU using OpenACC

Joeffrey Legaux

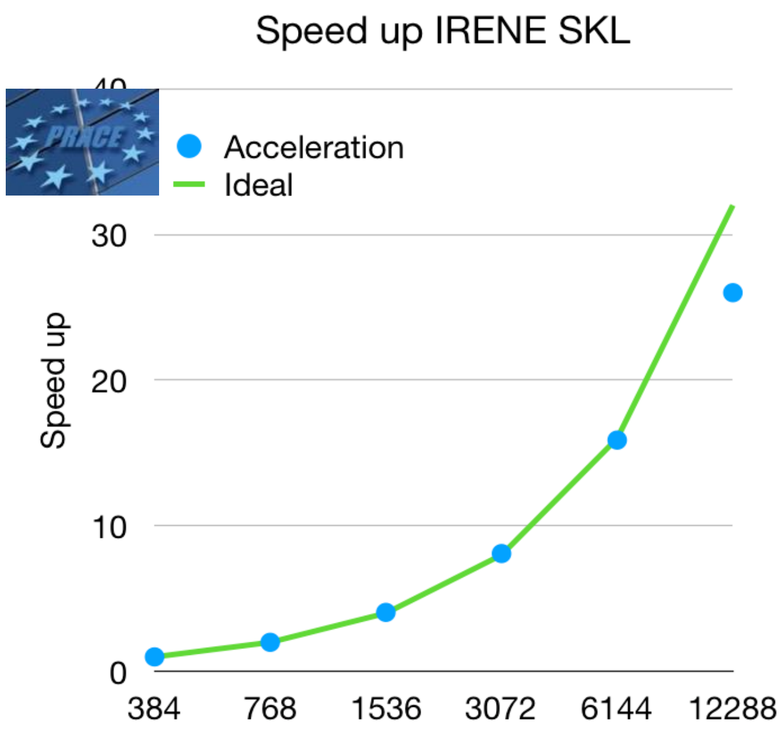
Journées NumPEx « retour d'expérience de portage de code sur GPU »

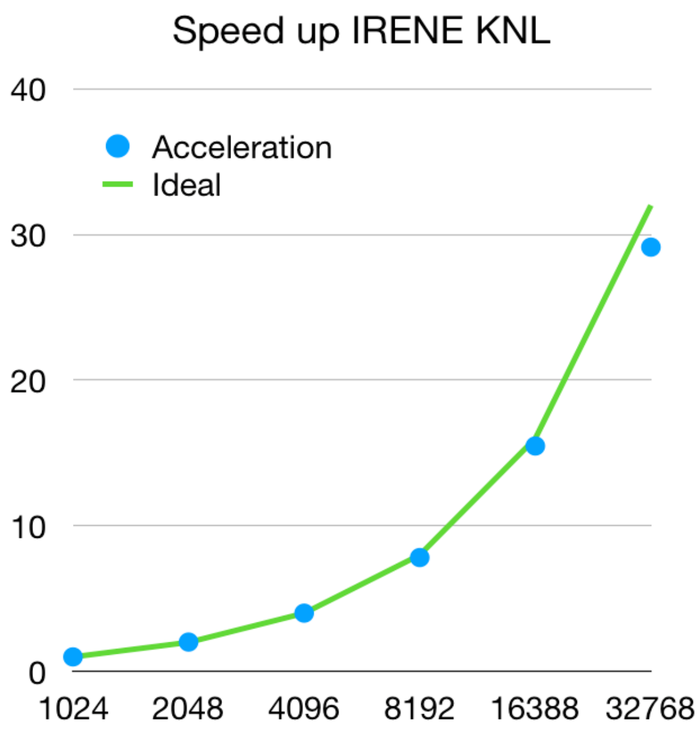
13th June 2024

joeffrey.legaux@cerfacs.fr

AVBP : a HPC code

- AVBP : compressive reactive LES on unstructured grids





- Excellent strong scaling response on CPU systems with full MPI
- Current record 200k cores 90% scaling

- However, most supercomputers have GPU-accelerated partitions nowadays

Extending AVBP with OpenACC

- Legacy fortran code (*1997*), explicit computations (no libraries)

- Code needs to **remain "simple"**
  - Very large code, active development and used in production
  - Fast new feature cycle (6 months)
  - **Limited HPC developers** and needs to be compatible with "CFD" students
  - Single CPU / CPU + GPU version of the code
  - Code needs to remain portable
    - ✓ Rewriting for GPU-specific languages or libraries is out of the window

- OpenACC was the most sensible choice at the time
  - Nvidia close to monopoly in HPC GPUs and provided support
  - OpenMP target implementations were (still are) lacking

How does AVBP work ?

- Fortran + MPI + Parallel HDF5 I/O

Input/Read

Decomposition and distribution of data

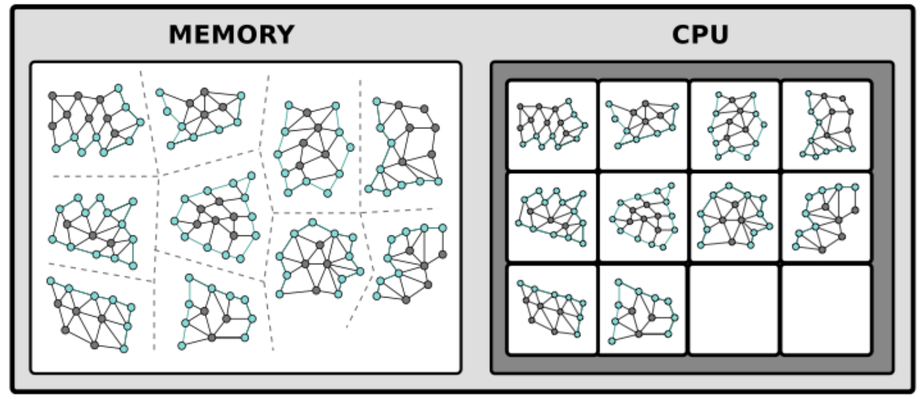
Numerical Scheme

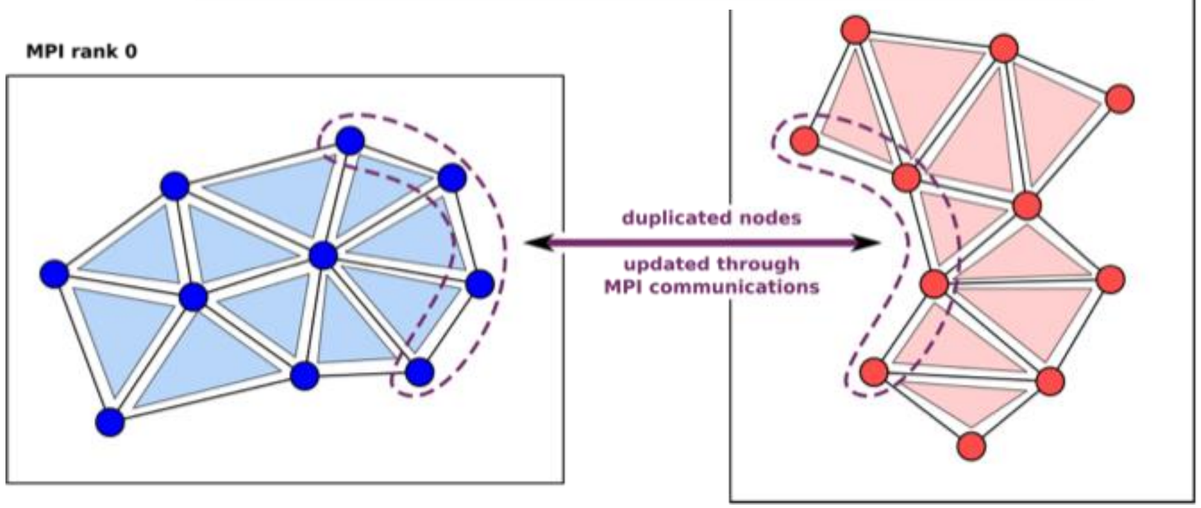
Main loop

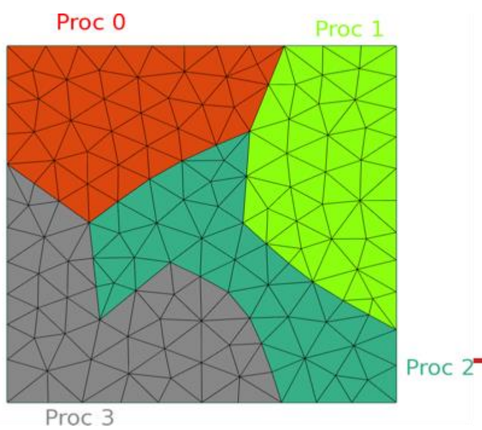
- Convective Scheme
- Diffusion Scheme
- Boundary treatment
- Physical models

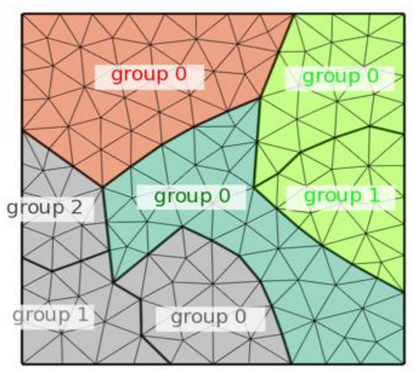
Postprocessing and I/O

END ? — NO / YES



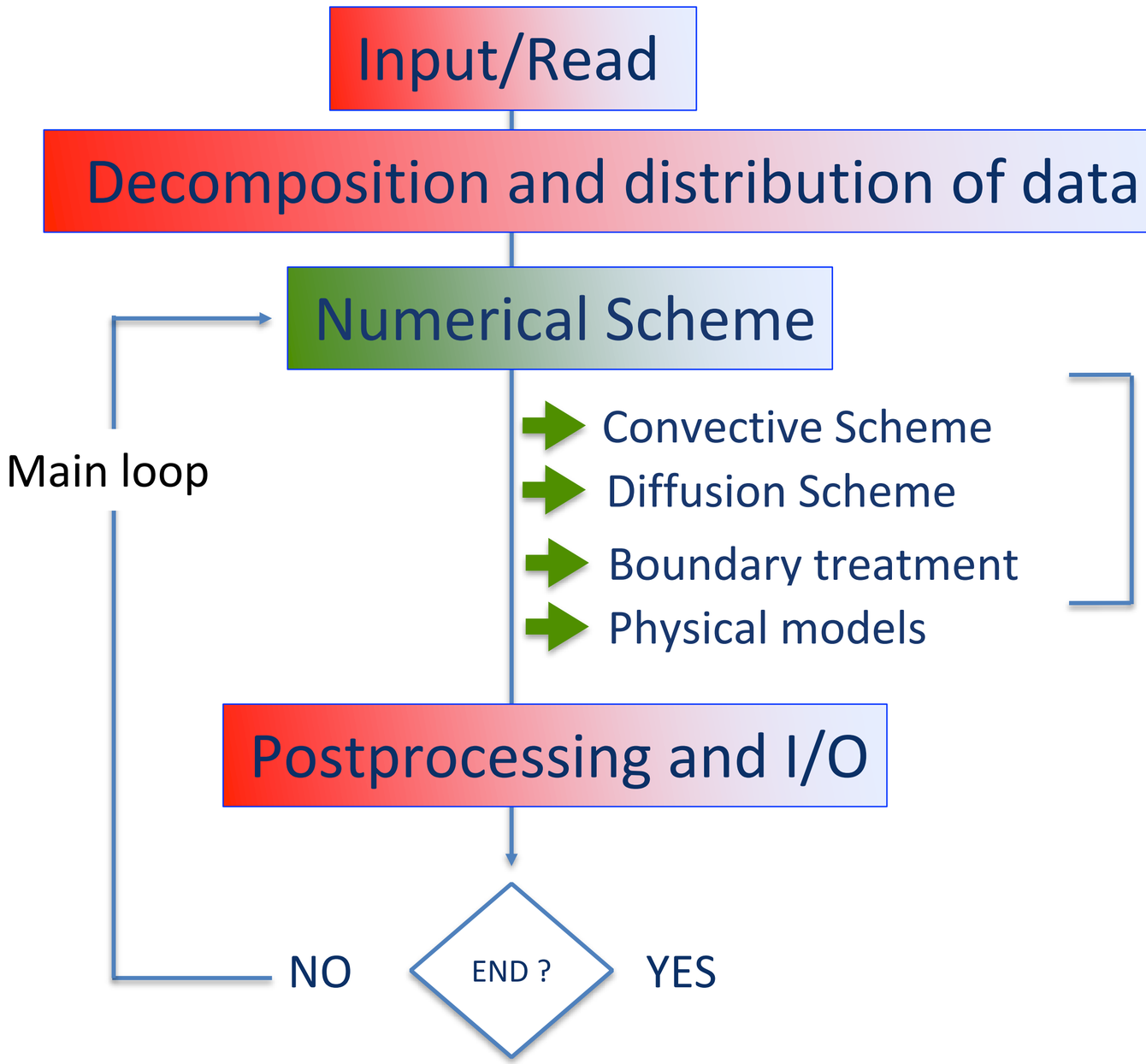






Extending AVBP for GPUs

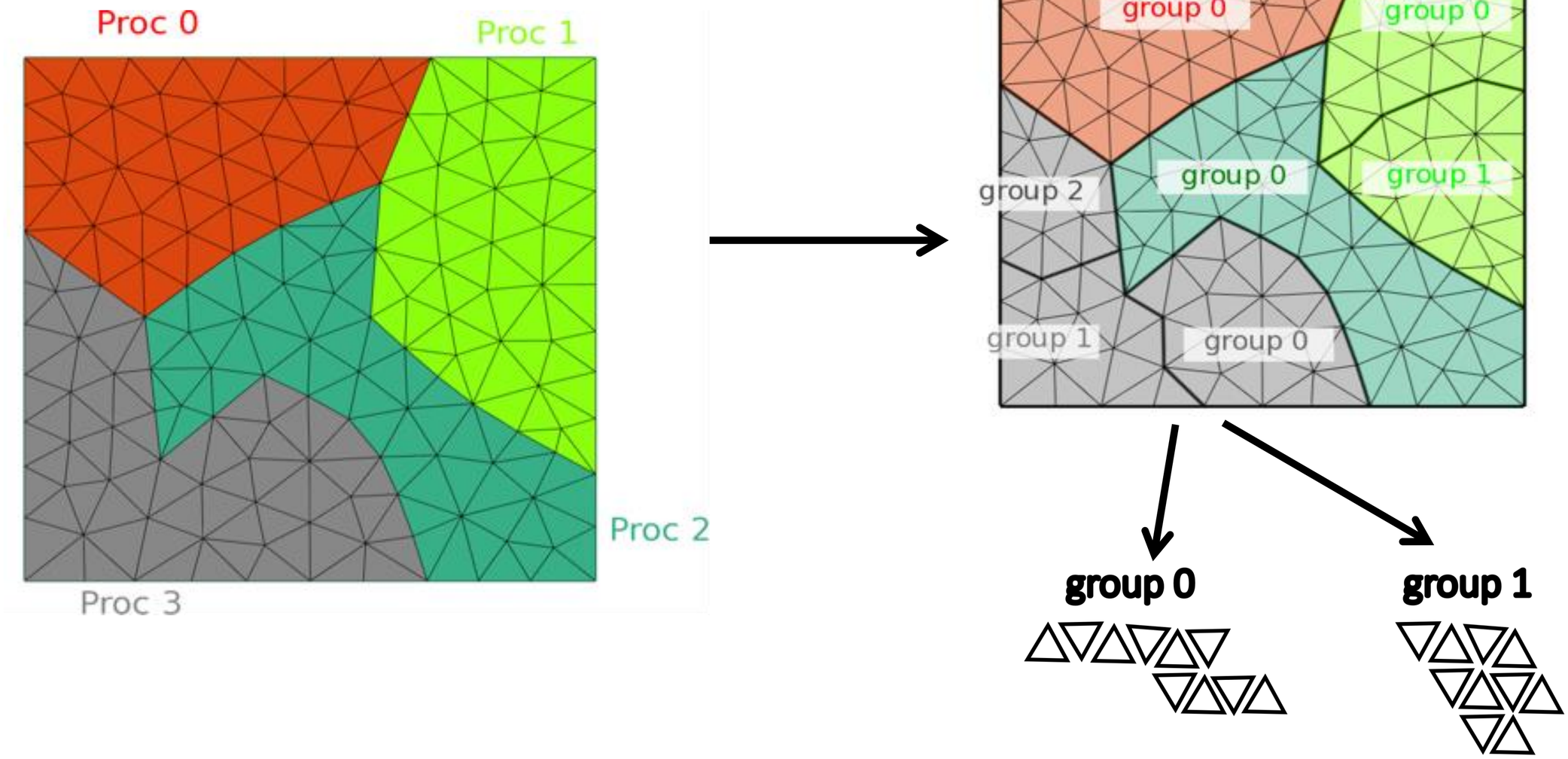
- Fortran + MPI (+ OpenMP) + Parallel HDF5 I/O



- **High I/O sections incompatible with GPU**
- **Compute intensive kernels compatible with GPUs**

Data and loop structure

- Unstructured mesh
  - decomposed into structured groups of independent « cells »



Data and loop structure

MAIN LOOP

```
DO n = 1, ngroup
  Call scheme (global R data, global RW data)
END DO
```

COARSE GRAIN

```
USE module only scheme_data

CALL function1(global_R_data,global_RW_data,
scheme_data)
..
CALL function…(global_R_data,global_RW_data,
scheme_data)
```

```
USE module only internal_data
DO i=1,ncells
  X[i] = B* X[i] +. A*Y[i]
END DO
```

FINE GRAIN

Porting approach

- Coarse grain : out of the way
  - ✓ Would require complete revamp of the data structure
  - ✓ Very long, divergent threads

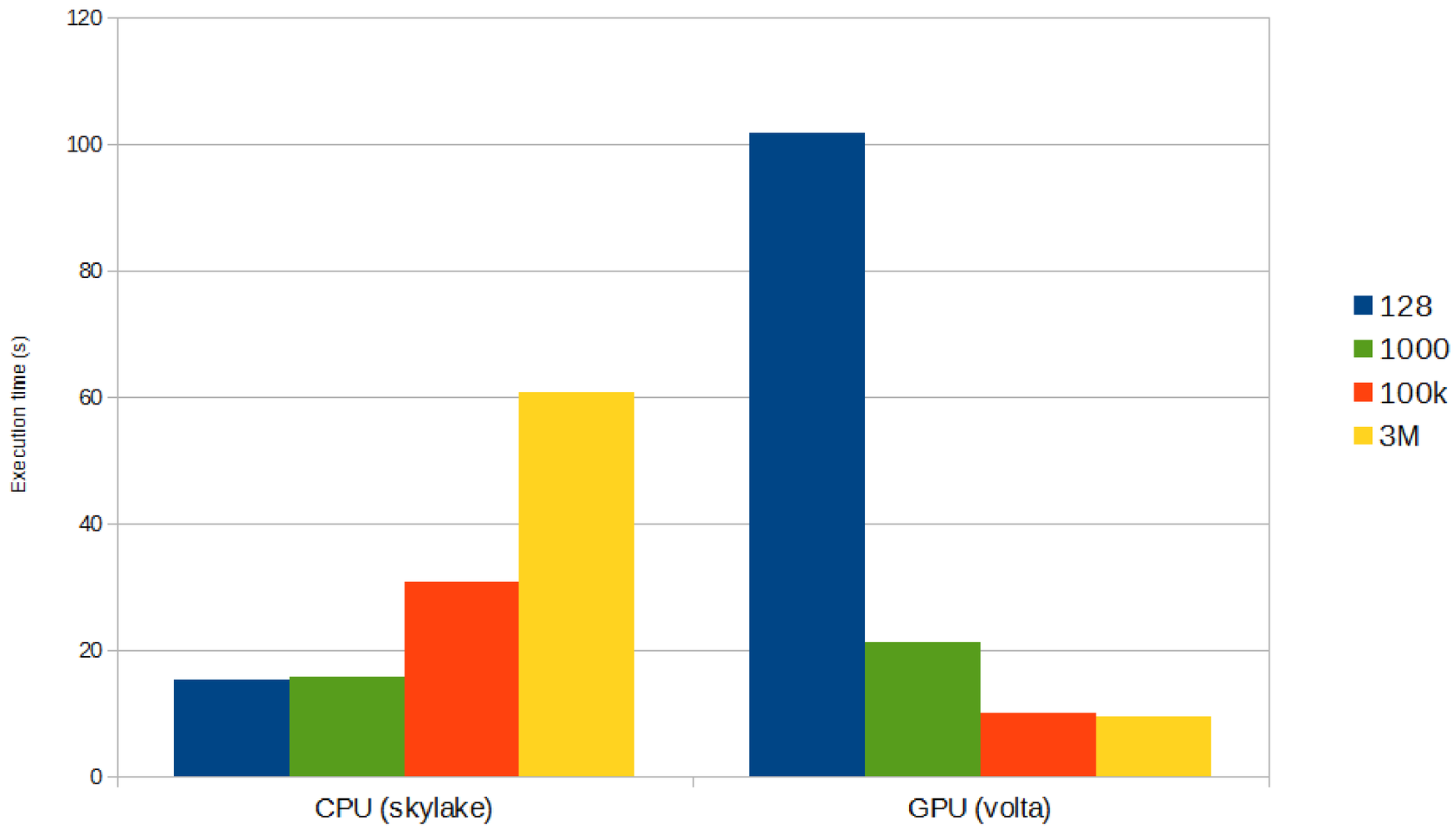
- Small kernels at fine grain : retained approach
  - ✓ Parallelise loops over cells only
  - ✓ Local loops : easy to track memory usage

- A tedious, **step-by-step work, but easy to check**
  - ✓ Each loop can easily be isolated, ported one at a time for debugging, optimisation or precision evaluation

Fine grain approach

- Originally : small groups of cells for CPU cache
- On GPU : large groups for fewer kernel launches and better occupancy

Gradient computation kernel



Fine grain approach

- First prototype : focus on 2 majors kernels (60~80% of iterative loop time)

  - Correct results, interesting individual speedups
  - Memory exchanges around kernels are too expensive
  - Copy costs increase with mesh size while GPU kernels performance improve
    - negative balance quickly reached

Data-driven approach

- Broadening of the previous strategy
  - Still based on compute-intensive loop porting
  - Port every operation involving large arrays
    - avoid reccurent memory exchanges
  - Quickly, it comes down to porting all of the temporal loop
- Collaboration with IDRIS (*Contrat de Progrès)*
  - 2 FTE during a year + ~1 FTE from Cerfacs

Technical insights

- Explicit memory management
  - Only copy the relevant parts of the deep data structures
  - Prevent costly implicit operations
    - ✓ DEFAULT(PRESENT) policy enforced on every GPU kernel
  - The most difficult task on this port

Technical insights

- Compute-intensive loops
  - Parallelise the outer loop on cells (embarassingly parallel)
  - Collapse whenever possible
  - Small internal loops
    - ✓ Few iterations (*# of equations, # of vertices in a cell, …*) over runtime variables
      - Would not efficiently fill a warp of threads
    - ✓ !$ACC LOOP SEQ

Technical insights

- Pattern easily reproducible by other developers

```
!$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT) PRIVATE(summ) IF ( using_acc )
DO n=1,nlcell
  DO k=1,neq

    summ = zero
    !$ACC LOOP SEQ
    DO nv = 1,nvert

      summ = summ - v_c_nv(k,1,nv,n) * snc(1,nv,n)&
                  - v_c_nv(k,2,nv,n) * snc(2,nv,n)

    END DO
    div_c(k,n) = half * summ

  END DO
END DO
```

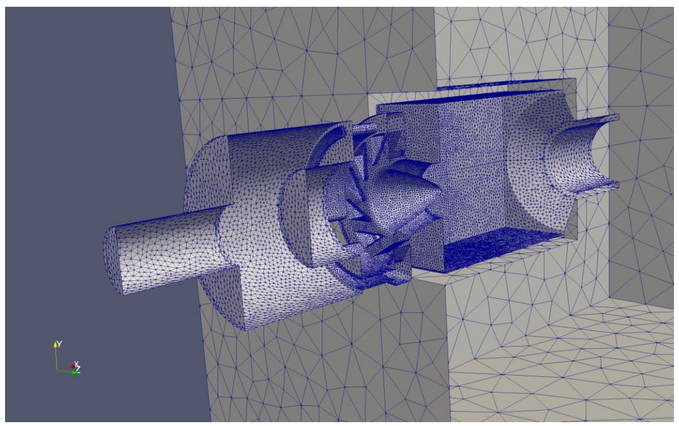
Technical insights

- Routines that manipulate the data structures are tough to port

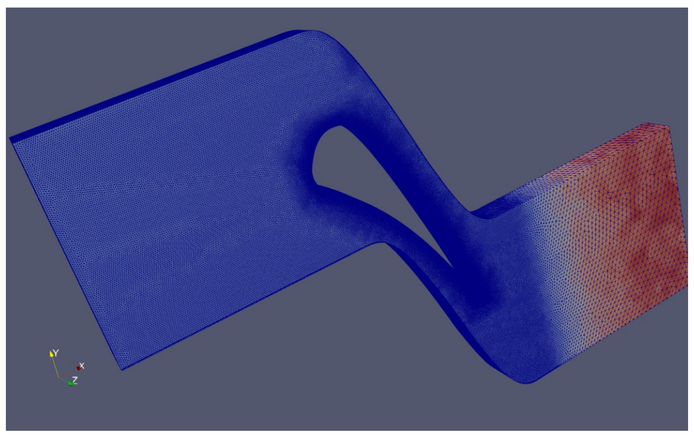
  - Most complex treatments (mesh loading, domain decomposition, …) before the temporal loop
    - leave them on CPU, data copy only once

  - There remains routines inside the loop (MPI buffers, post-proc for outputs…)
    - Some could not be efficiently ported (sequential by nature) and had to be rewritten specifically for GPU
    - « Expert » routines rarely touched by the CFD developers

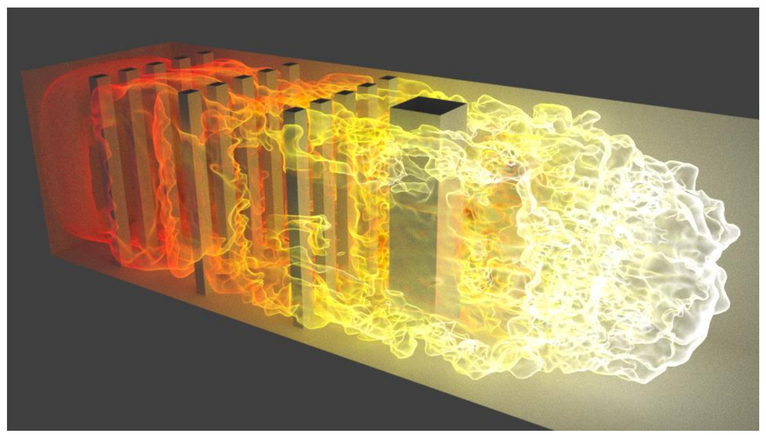
Performances



Lab scale burner : Preccinsta



Academic aerodynamics : NASAC3X



Multi-scale experiment : MASRI explosion

Combustion chamber

Improved acceleration with new optimisations

Acceleration on one node of JEANZAY (2x20core Intel Xeon 6248+ 4 **V100**)

CPU vs GPU (full nodes)	AVBP V7.7	AVBP V7.8	AVBP V7.9
Acceleration Preccinsta	1.82	3.9	4.7
Acceleration NASAC3X	2.81	3.4	3.8
Acceleration Explosion	1.84	4.2	4.8
Acceleration Industrial C.C	Not supported	3.5	5

Nvhpc 21.5

Portable performance between architecture generations

Acceleration on one node of JUWELS BOOSTER (2x24core AMD Epyc 7402 + 4 **A100**)

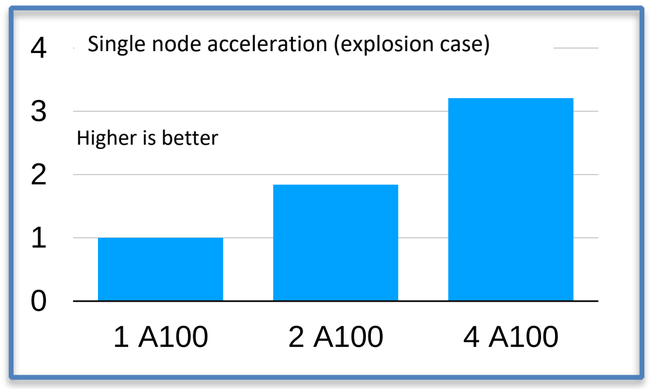
Higher is better

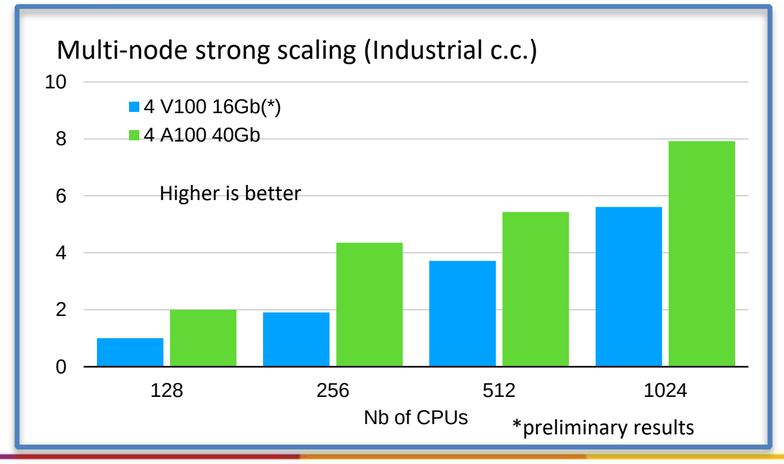
CPU vs GPU (full nodes)	AVBP V7.7	AVBP V7.8	AVBP V7.9
Acceleration Preccinsta	1.9	1.9	2.9
Acceleration NASAC3X	2.9	2.9	3.4
Acceleration Explosion	2.2	2.3	3.65
Acceleration Industrial C.C	Not supported	Not supported	4.8

Nvhpc 20.9

This work has been supported by the EXCELLERAT project (which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823691) and the French National Technology Watch Group.

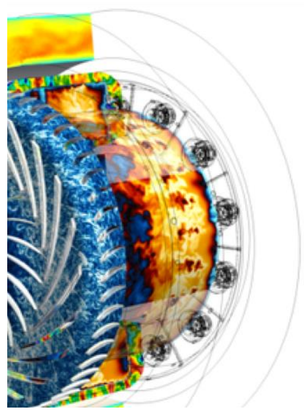
Excellent single node and multi-node strong scaling



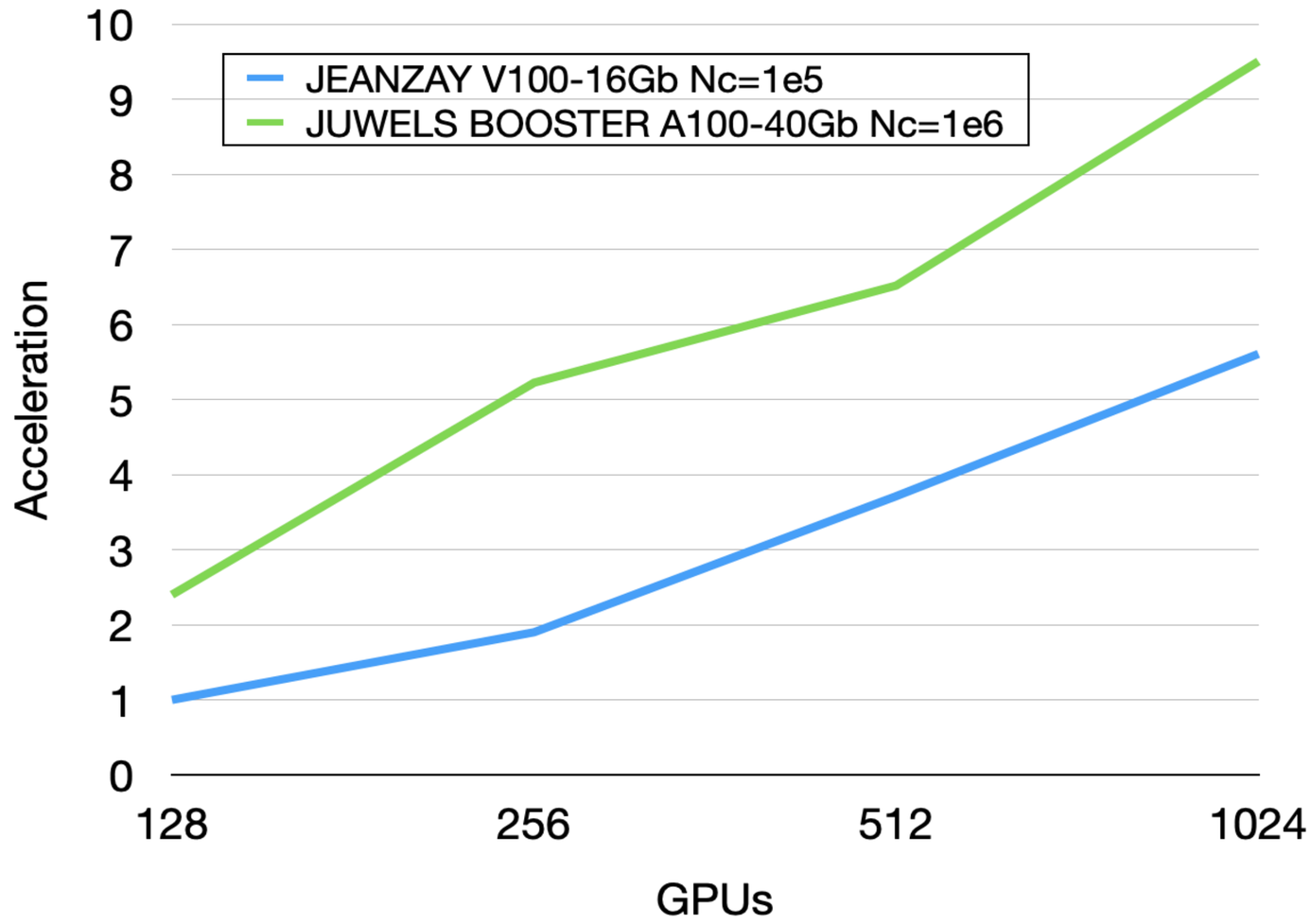


Performances

- Strong scaling over multiple GPU nodes
  - Using a heavier version of the CC case (full chamber)
  - Using full nodes on Jean-Zay based on Volta cards
  - Relative speed-up on Juwels BOOSTER, based on Ampere cards



CC Full chamber



Going forward

- Fully integrated to the official release
- Coverage of around 70% of use cases and growing
- Relatively efficient port with minimal changes to the codebase

- 1 A100 GPU ~= 50 Icelake cores

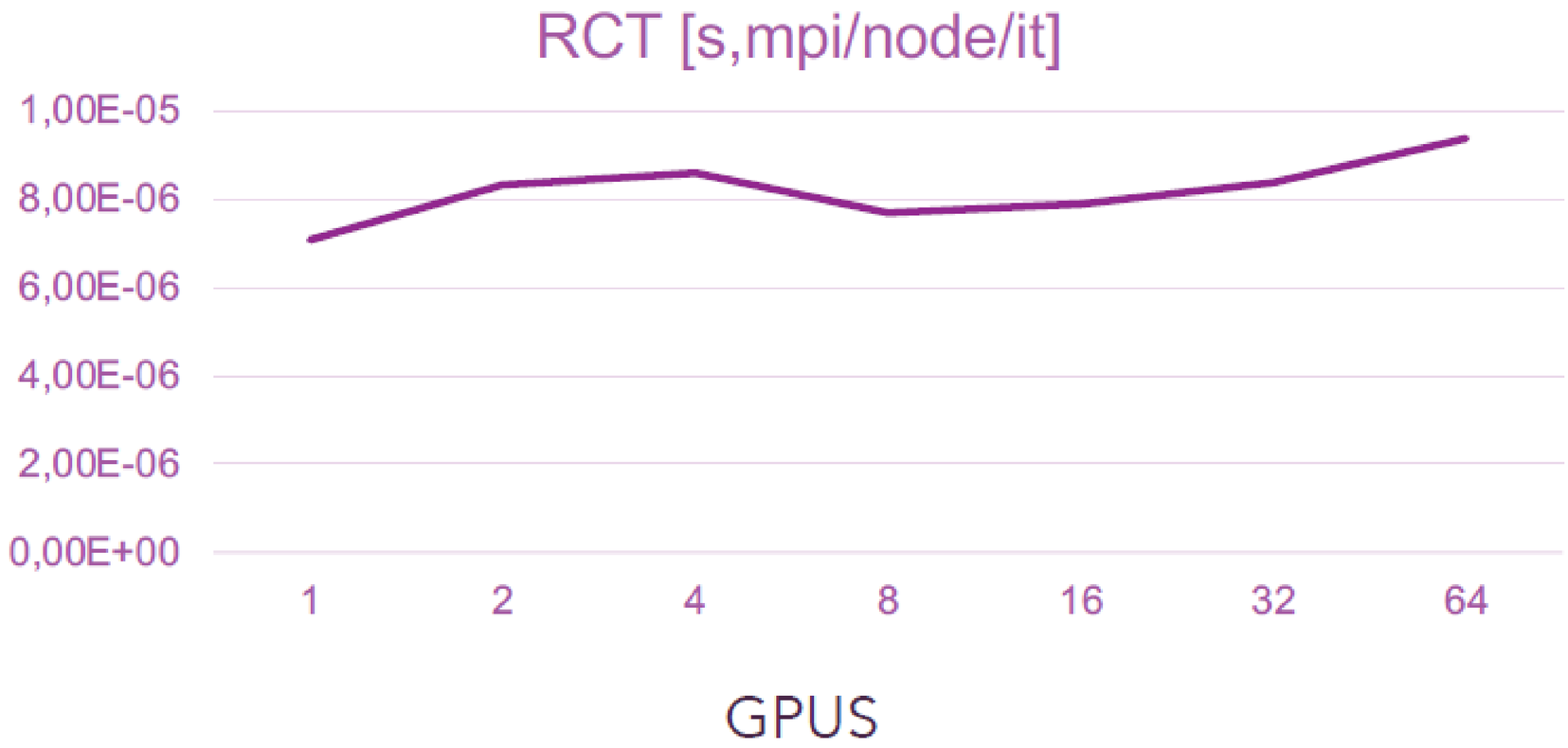
- Many kernels with few operations on large arrays
  - We are essentially bound by global memory bandwidth
  - Investigation on how to improve locality, but this might require substantial changes on the original code.

Going forward

- Porting to others GPU vendors
- We take advantage of Cray compiler for OpenACC support on AMD GPUs (*Lumi, Adastra…*)
- It does not handle some things as well as nvhpc
  - !$ACC KERNELS are turned into sequential kernels
    - Rewrite array-syntax operations as explicit loops
  - Module variables inside !$ACC ROUTINE not supported
    - Change some of the structure of the code
  - Atomic operations seem costly

Going forward

- Prototype on Adastra



- Good scalability
- However base performance is still bad
  - *A30 card is around 1,00 e-06 !*

Perspectives

- OpenACC port requires improvements for AMD

- OpenMP offloading assessed to target Intel GPUs
  - Limitations of OpenMP / OneAPI (deep derived types structures, …)
  - Potential workarounds, but too invasive on the code

- Exploring possibilities to improve the code structure for better GPU performance (kernel fusion, layered parallelism, memory layout…)

- *Extending and maintaining the code !*
  - Keeping the code usable for many various supercomputers

THANK YOU