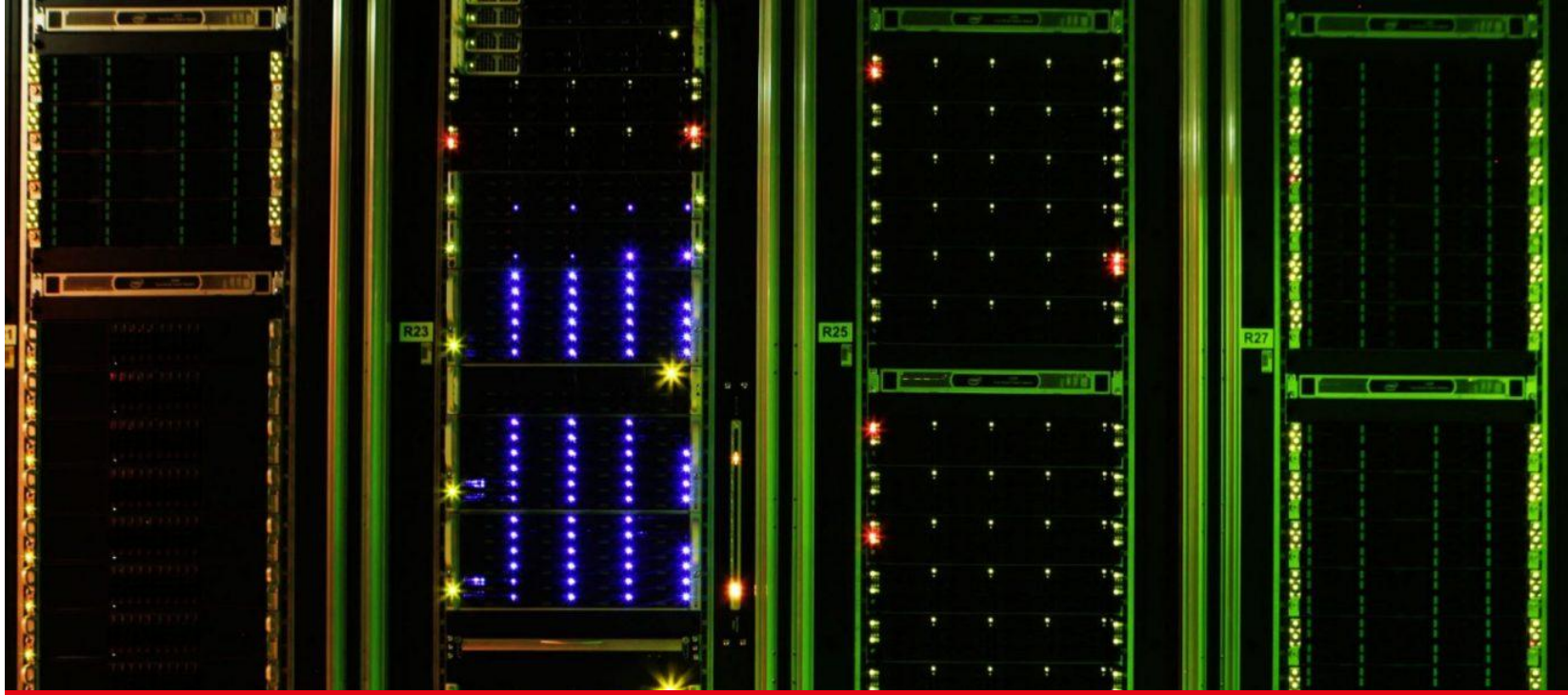
EPFL



GPU Porting of CAS3D and SOLEDGE3X codes using OpenACC/OpenMP offload

SWISS PLASMA CENTER

INFORMATIQUE SCIENTIFIQUE & SUPPORT APPLICATIF
SCITAS
SCIENTIFIC IT AND APPLICATION SUPPORT

E. Bourne, G. Fourestey, M. Peybernes
Ecole Polytechnique Fédérale de Lausanne (EPFL), SCITAS

Transition towards GPU codes

There are 3 main approaches:

Pragma directives (OpenMP offload / OpenACC)

Cuda

Library encapsulation (e.g. Kokkos)

- ASCOT5 (Aalto University)
- CAS3D (IPP)
- FELTOR (DTU)
- GBS (EPFL)
- GRILLIX (IPP)
- GyselaX (cea irfm)
- ORB5 (EPFL)
- Soledge3X (cea irfm)

Transition towards GPU codes

There are 3 main approaches:

- Pragma directives (OpenMP offload / OpenACC)
- Cuda / Hip / …
- Library encapsulation (e.g. Kokkos)

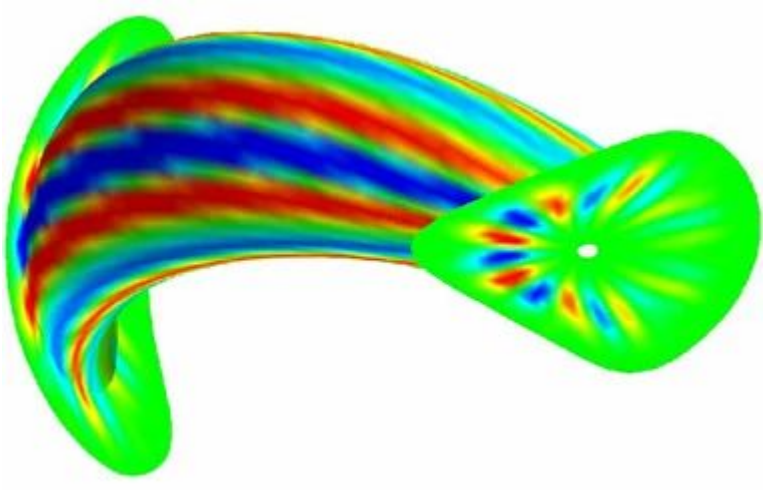
- ASCOT5
- CAS3D
- FELTOR
- GBS

- GRILLIX
- GyselaX
- ORB5
- Soledge3X

CAS3D

G. Fourestey, M. Peybernes (SCITAS),
C. Nuehrenberg (Max Planck Institute for Plasma Physics)

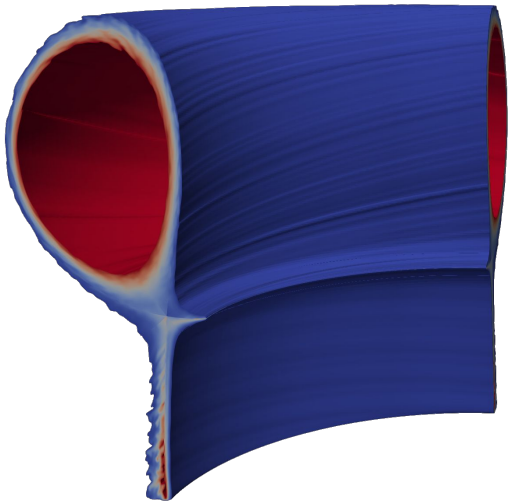
- CAS3D is used to study the ideal magnetohydrodynamic (MHD) properties of fusion toroidal plasmas
- The numerical scheme uses:
  - **Finite elements** along the radial direction
  - **Fourier decomposition** along the poloidal and toroidal directions
  - Inverse iteration algorithm to solve **eigenvalues problem** for a sparse symmetric matrix.



Soledge3X

E. Bourne M. Peybernes (SCITAS)
H. Bufferand, P. Tamain (Soledge group - CEA)

- Soledge3x is used to study tokamak edge plasma
- The numerical scheme uses:
  - WENO methods for the advection
  - 1D, 2D, and 3D Sparse matrix problems
  - linear solvers
- 5 implicit solvers in SOLEDGE3X (typical share of computing time):
  - Parallel viscosity terms (~12% / 7% with FN)
  - Parallel heat conduction terms (~17% / 9% with FN)
  - Vorticity equation (~22% / 12% with FN)
  - (optional) fluid neutrals (~48% with FN!!)
  - (optional) potential filter (negligible)



Porting codes to GPU with pragmas

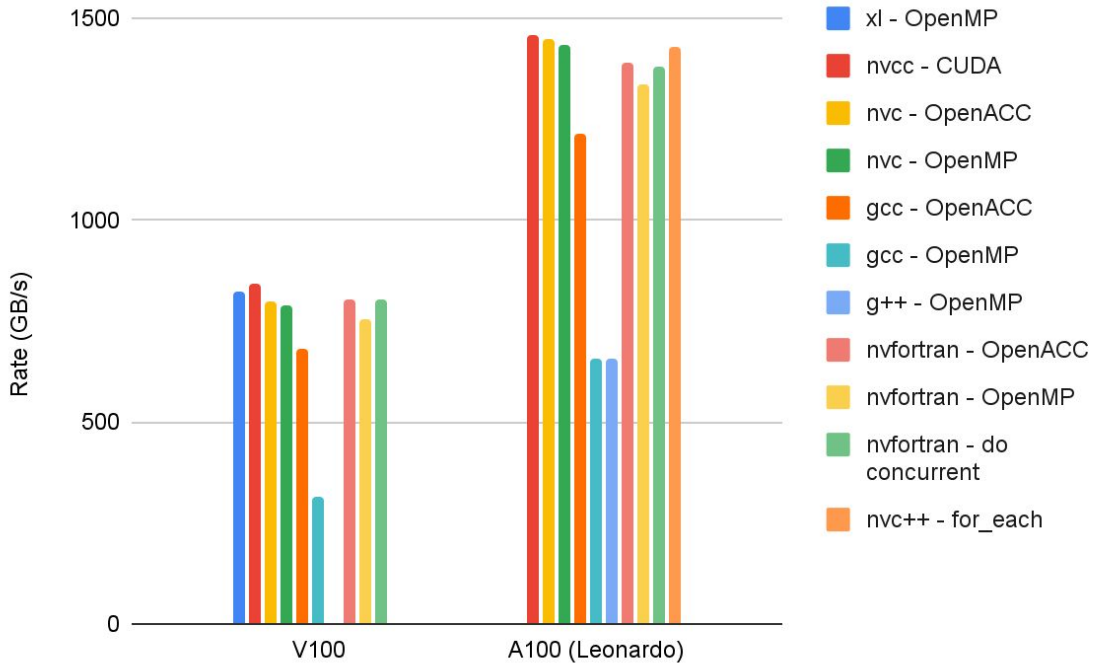
General strategy to port CAS3D and SOLEDGE3X codes to GPU:

- Porting by directives with OpenMP offload and/or OpenACC
- This choice allows us
  - to keep only one version of the code with readability and durability
  - to perform tests on NVIDIA and AMD GPUs (and INTEL) for multiple compilers:

    xl (M100-Nvidia) ; gnu (Leonardo-Nvidia) ; cce (Piz-Daint-Nvidia, then ALPS & Adastra-AMD & LUMI-AMD) ; nvhpc (Leonardo-Nvidia)
- This strategy can show good performance (not always!), in particular on Marconi100 with the IBM XL compiler with OpenMP offload
- However, other compilers, like GCC, present weak performances with OpenMP offload while OpenACC appears to be more efficient
- This lack of performance portability led to the introduction of generic pragmas to use OpenMP or OpenACC

Porting codes to GPU with pragmas

STREAM benchmark:



https://gitlab.com/ursache/stream-openmp-offload

Porting codes to GPU with pragmas

```
!$omp target teams distribute collapse(2)
do ichunk = 1, split%Nchunks
  do ispec = 0, Nspecies
      … some work …
      $omp parallel do simd collapse(3)
      do ipsi = ipsimin, ipsimax
        do itheta= ithetamin, ithetamax
          do iphi = iphimin, iphimax
```

OpenMP

```
!$acc loop gang collapse(2)
do ichunk = 1, split%Nchunks
  do ispec = 0, Nspecies
      … some work …
      !$acc loop worker vector collapse(3)
      do ipsi = ipsimin, ipsimax
        do itheta= ithetamin, ithetamax
          do iphi = iphimin, iphimax
```

OpenACC

```
GPU_LOOP_L1 collapse(2)
do ichunk = 1, split%Nchunks
  do ispec = 0, Nspecies
      … some work …
      GPU_LOOP_L2_L3 collapse(3)
      do ipsi = ipsimin, ipsimax
        do itheta= ithetamin, ithetamax
          do iphi = iphimin, iphimax
```

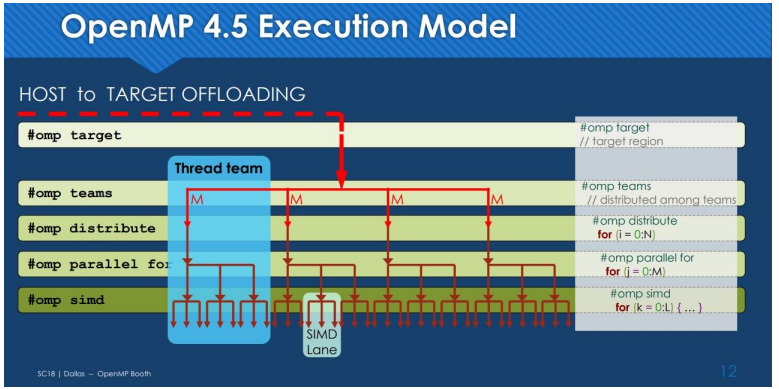
Pragmas

```
#ifndef gpu_commands
#define gpu_commands
#ifdef _OPENMP

#define GPU_MAP_TO_DEVICE !$omp target enter data map(to:
#define GPU_MAP_FROM_DEVICE !$omp target exit data map(from:

#define GPU_LOOP_L1 !$omp target teams distribute
#define GPU_END_LOOP_L1 !$omp end target teams distribute

#define GPU_LOOP_L2_L3 $omp parallel do simd

… etc …
#elif _OPENACC

#define GPU_MAP_TO_DEVICE !$acc enter data copyin(
#define GPU_MAP_FROM_DEVICE !$acc exit data copyout(

#define GPU_LOOP_L1 !$acc loop gang
#define GPU_END_LOOP_LEVEL_1 !$acc end loop

#define GPU_LOOP_L2_L3 !$acc loop worker vector

… etc …
#endif
```

gpu_transfers.inc

Levels of Parallelism

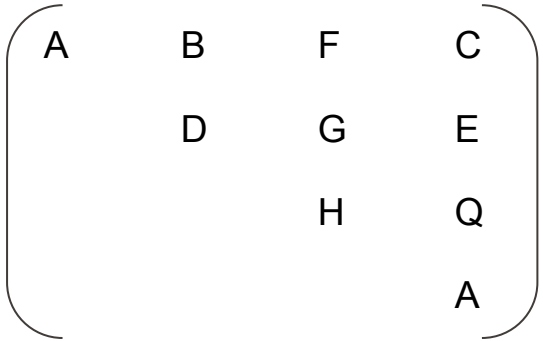
- OpenMP and OpenACC have similar coding paradigms under the hood
- In theory (according to the standards) the implementation of the levels adapts to the hardware, even if in reality some compilers struggle with certain parallelisation levels



OpenMP	OpenACC	HIP	CUDA	Mapping AMD GPU	Mapping NVIDIA GPU
Parallel	Kernel/parallel	Kernel	Kernel	GPU/GCD	GPU
Team	gang	Work group	Thread bloc	Comput Unit	SM (Symmetric Multiprocessor)
Thread	worker	Work item	thread	Scalar Unit Part of Vector (SIMD) Unit	Comput Unit
SIMD	Vector	Wavefront (64 work items)	Warp (32 threads)	64-wide work item	32-wide thread

CAS3D - CPU Algorithm

- The expensive part is the computation of matrices (memory latency bound)
- Each Matrix block has the following dimensions
  - A-block: *modx · modx*
  - B-block: *modx · mody*
  - F-block: *modx · modz*
  - C-block: *modx · modx*
  - D-block: *mody · mody*
  - G-block: *mody · modz*
  - E-block: *mody · modx*
  - H-block: *modz · modz*
  - Q-block: *modz · modx*



where *modx, mody, modz* depend on the number of Fourier coefficients used during the discretisation

—> typically, modx,y,z in the order of 10^2

CAS3D - GPU Multi-level parallelism

- Each GPU kernel computes a matrix block (N ~ 10^4) :
  - each team/gang calls *matrixa* subroutine
  - each team/gang spawns its threads to parallelize the loop in *matrixa* subroutine
  - need to refactor initial code to collapse 2 loops on Fourier coefficients to extract more parallelism over the teams/gangs

Parallelization over gangs/teams

```
if (irun>=1 … then
GPU_LOOP_L1 COLLAPSE(2) PRIVATE(Tab1(N), Tab2(N), …)
      do jmod = 1, mody
          do imod = 1, modx
              call matrixa (wpot, wkin, imod, jmod)
              Tab1(k) = …
              a    (imod,jmod,igrid) = fachalf*wpot
              arhs(imod,jmod,igrid) = fachalf*wki
```

Parallelization over workers/threads

```
GPU_LOOP_L2_L3
      do jt = 1, N
            wpot = …
            wki  = …
```

CAS3D - Gang-private array workaround

- On some machines/compilers (nvfortran < 23.1) private arrays are not correctly handled
- A workaround is to allocate shared arrays over the number of gangs

  —> avoids also dynamical allocation and improves performance

- This requires accessing a gang/team id.
  - In OpenMP this is available
  - In OpenACC with **nvhpc** we can use pragmas exposed via C (not compiler-portable):

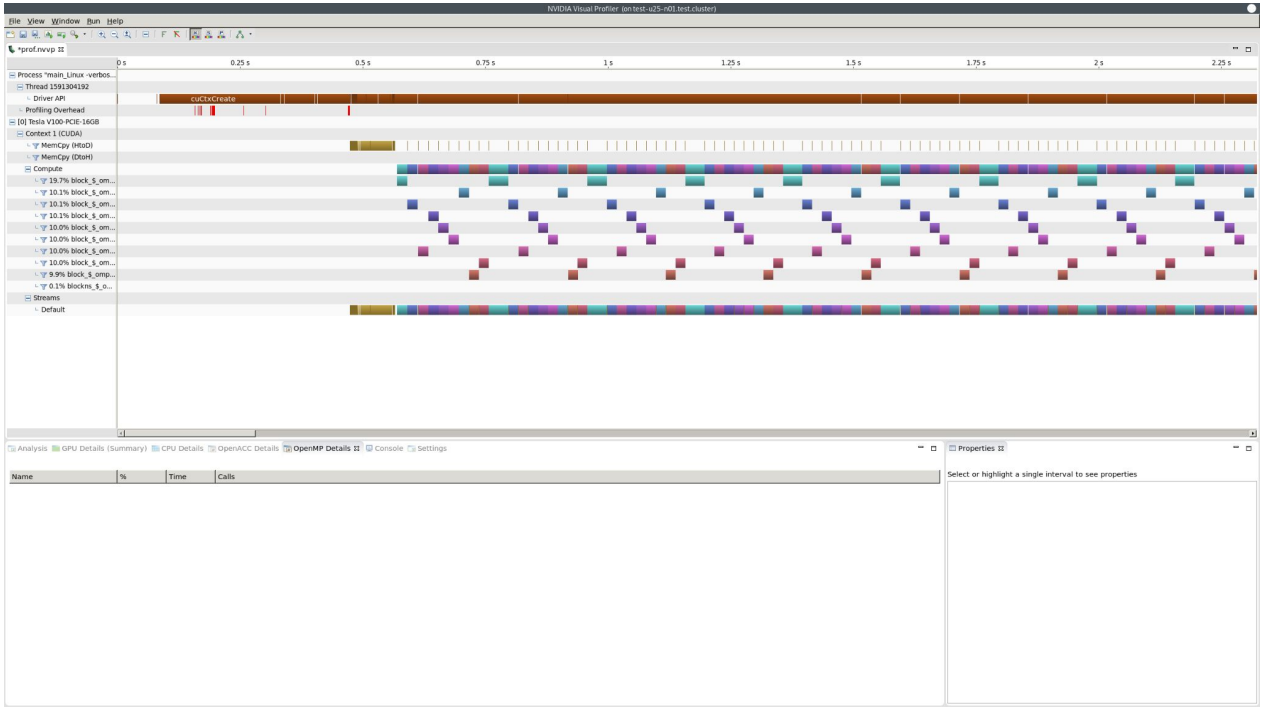
```
#ifdef __PGI
#include "openacc.h"

#pragma acc routine worker
int get_gang_id() {
        return __pgi_gangidx() + 1;
}

#pragma acc routine vector
int get_worker_id() {
        return __pgi_workeridx() + 1;
}

#pragma acc routine seq
int get_vector_id() {
        return __pgi_vectoridx() + 1;
}

#endif
```

```
allocate(Tab1(N,nbgangs))
allocate(Tab2(N,nbgangs))

if (irun>=1 … then
GPU_LOOP_L1 COLLAPSE(2) PRIVATE(Tab1(N), Tab2(N), …)
      do jmod = 1, mody
          do imod = 1, modx
             call matrixa (wpot, wkin, imod, jmod)
             Tab1(k,gangId) = …
             a    (imod,jmod,igrid) = fachalf*wpot
             arhs(imod,jmod,igrid) = fachalf*wki
```

```
GPU_LOOP_L2_L3
      do jt = 1, N
           wpot = …
           wki  = …
```

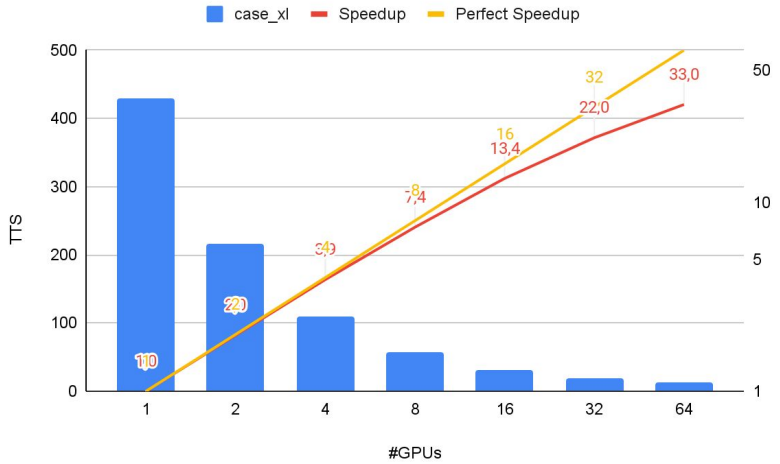
CAS3D - GPU Profiling

- GPU
  - Profiling with Nvidia Nsight Systems
  - Transfers optimized between kernels



CAS3D - MPI+GPUs Results

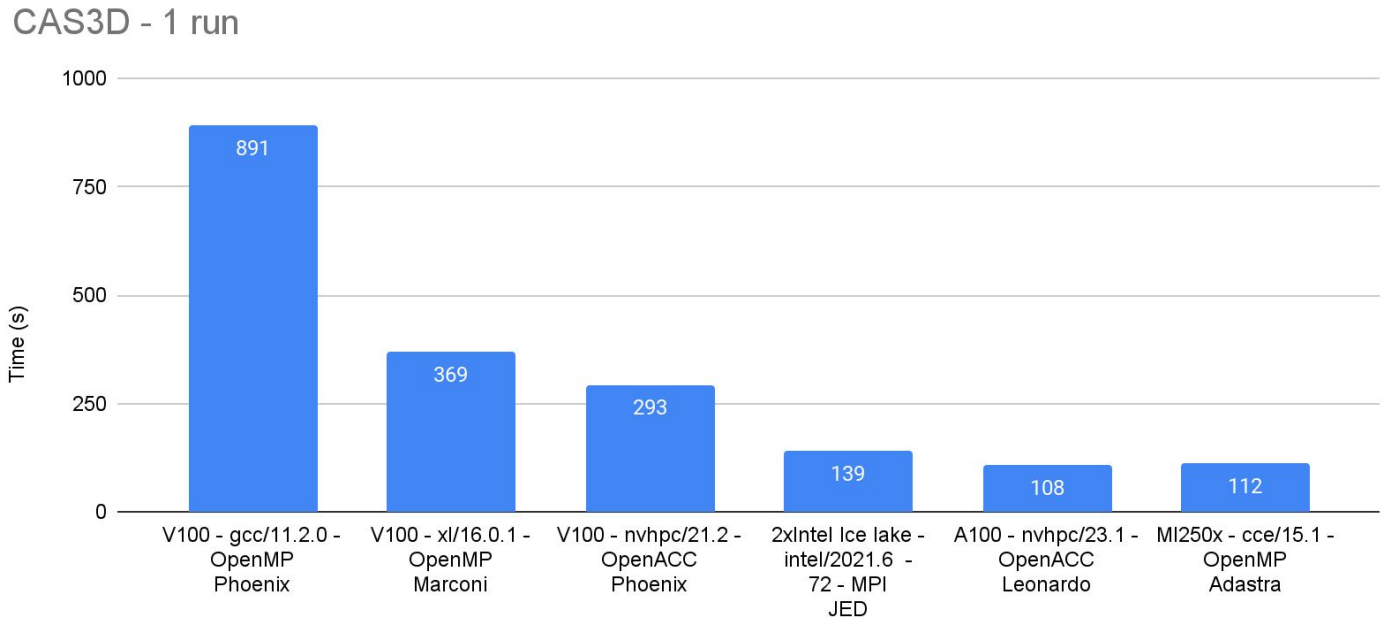
- Results on Marconi 100 (P9 + V100 GPUs)
  - Good scaling
  - Performance increases compared to the CPU version for large test cases



XL-compiler on M100	1 node (4GPUs)	2 nodes	4 nodes	8 nodes	16 nodes	GPU-serial (640 teams - 128 threads)	CPU: 1node (32-cores)
#GPUs	4	8	16	32	64	1	0
case_mm	4 s	3,2 s	3 s	3,1 s	3,3 s	6 s	5 s
case_ll	78 s	42 s	24 s	15 s	14,8 s	299 s	190 s
case_xl	110 s	58 s	32 s	19,5 s	13 s	410 s	418 s

CAS3D - GPU Results

- GPU - OpenMP-Offload vs OpenACC

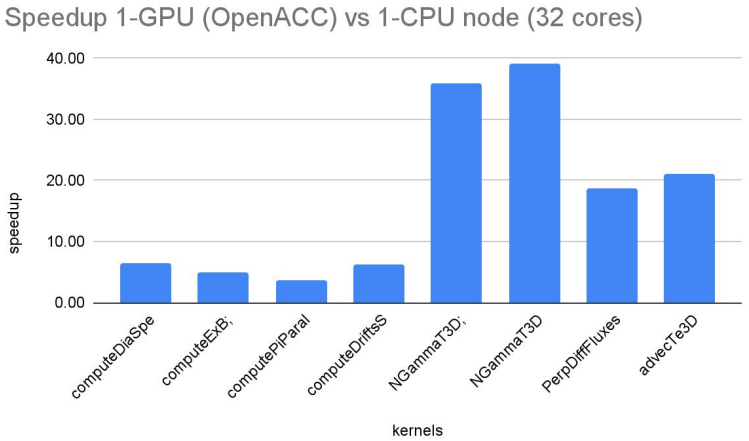


Soledge3X - Porting strategy

- 3 level domain decomposition:
  - Structured zones for magnetic topology
  - MPI blocks: prioritized by flux surface across zones
    - If NMPI≤NFS each MPI process in charge of a set of FS
    - If NMPI>NFS largest flux surfaces will be shared by a team MPI processes
  - Thread chunks: no direction priority, aiming at load balance between chunks ; CPU OpenMP loops are on chunks and species
- GPU porting
  - Open ACC/MP for advection and matrix construction
  - PETSC (with CUDA/HIP) for linear solvers

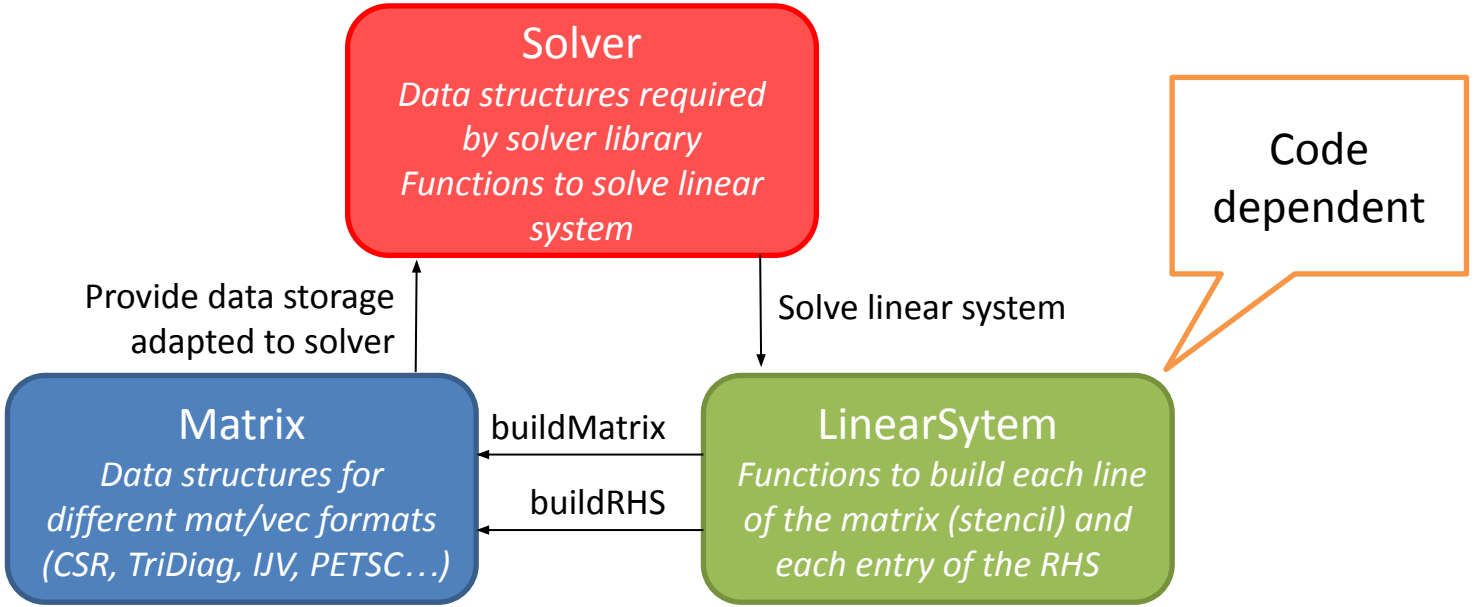
```
!$OMP DO SCHEDULE(RUNTIME) COLLAPSE(2)
GPU_PARALLEL_LOOP_L1 collapse(2)
do ichunk = 1, split%Nchunks
  do ispec = 0, Nspecies
     … some work …
      GPU_LOOP_L2_L3 collapse(3)
      do ipsi = ipsimin, ipsimax
        do itheta= ithetamin, ithetamax
          do iphi = iphimin, iphimax
```

Each kernel profiled with nsys-nvtx on CPU/GPU



Soledge3X - solvers

- Initially strong duplication of effort with risk of bugs
  - implementation of each solver library needs to be done separately for each implicit linear system, including matrix construction (bugs!)
- Solution recast of solvers management based on 3 Fortran classes

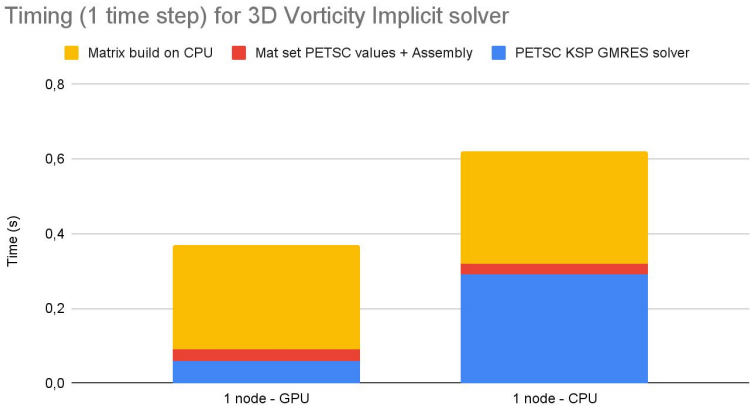


Soledge3X - solvers

- OpenACC Interoperability with CUDA

allocated via *cudaMallocManaged*

```
!$acc parallel loop collapse(3) deviceptr(p_vorticity_stencVal_coo)
do ipsi = ipsimin, ipsimax
  do itheta = ithetamin, ithetamax
    do iphi = iphimin, iphimax
      do ifield = 1, self%NdofPerPoint
        ! Get local row index and carry on only if it is non-zero
        ! (otherwise means that this point is not part of the linear system
        irowLoc = self%getMatLocalIndex(ichunk,ipsi,itheta,iphi,ifield)
        SOME WORKS…
        p_vorticity_oor(cnt:cnt+stencsize-1) = irowGlobList(1)
        p_vorticity_ooc(cnt:cnt+stencsize-1) = icolGlobList(1:stencSize)
        p_vorticity_stencVal_coo(cnt:cnt+stencsize-1) = stencVal(1:stencSize)
        cnt = cnt + stencSize
      enddo ! ifield
    enddo ! iphi
  enddo ! itheta
enddo ! ipsi
cnt = cnt - 1
petsc_cnt = cnt
call MatSetPreallocationCOO(mat_ptr%PETSCmat, &
                    petsc_cnt,p_vorticity_oor(1:cnt),p_vorticity_ooc(1:cnt),ierrPETSC)
call MatSetValuesCOO(mat_ptr%PETSCmat,p_vorticity_stencVal_coo(1:cnt),INSERT_VALUES,ierrPETSC)
```



Preliminary results on Leonardo

Conclusions

- Porting by directives with OpenMP offload and/or OpenACC
- Pragmas allow for a single code base with different solutions
- (Performance) portability is strongly dependent on hardware, compiler etc.
- For the moment, OpenACC is more mature than OpenMP offload
- Depending on the compiler, we can get quite good performance for CAS3D and Soledge3X codes
- Leonardo (Nvidia A100 - OpenACC) is ~3x faster than Marconi 100 (Nvidia V100 - OpenMP offload) for CAS3D
- Most CPU algorithms have to be modified further to get better performance on GPU
- GPU porting of Soledge3x needs some workarounds due to use of “recent” Fortran features like function pointer or class not handled by the most recent nvhpc compilers (23.9)

  —> ongoing work …