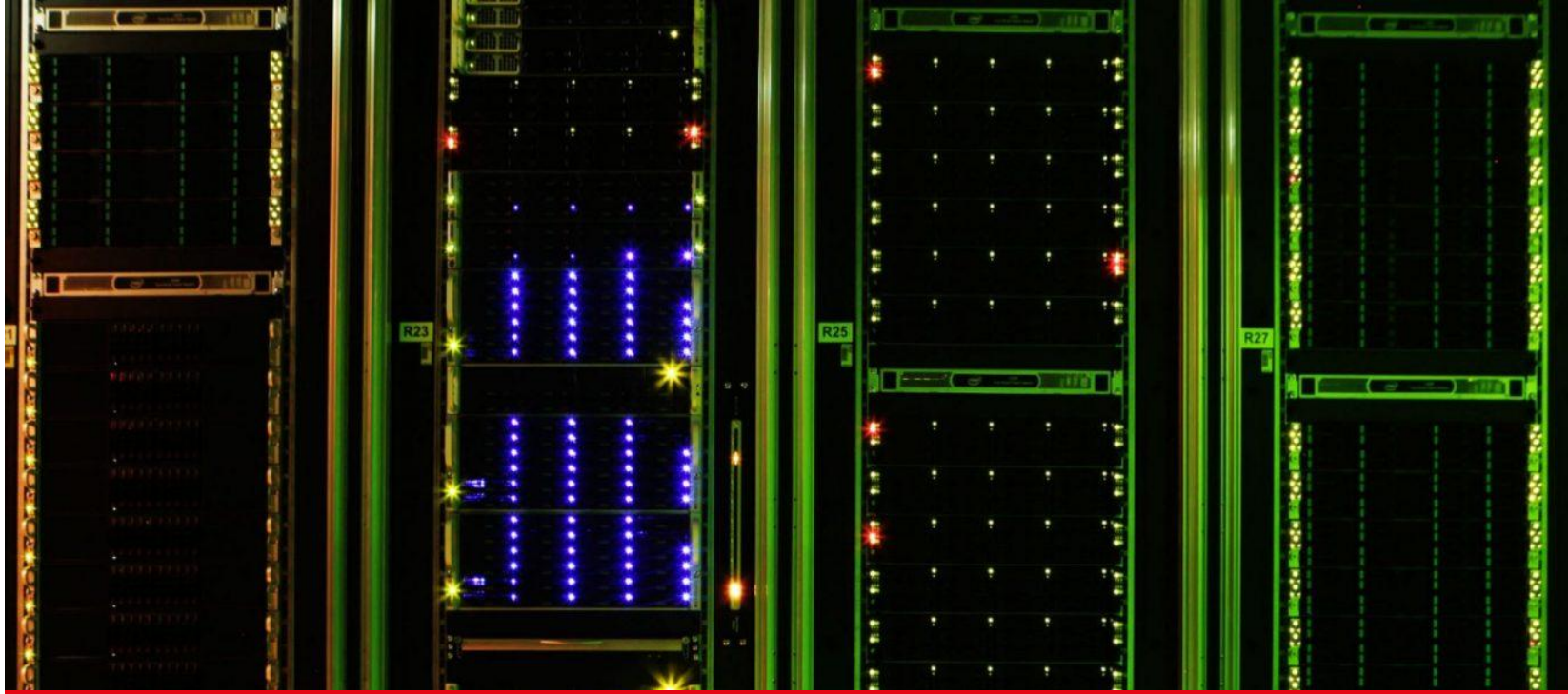


Portably Targeting GPUs using Pragma Directives

E. Bourne[1], G. Fourestey[1], E. Lanti[1], M. Peybernes[1], P. Ricci[2], N. Varini[1]
[1]Ecole Polytechnique Fédérale de Lausanne (EPFL), SCITAS
[2]Ecole Polytechnique Fédérale de Lausanne (EPFL), Swiss Plasma Center (SPC)

IFERC - Jun '24

Plasma Physics Codes at our hub

Context:

- EUROfusion codes are research codes mostly written in C and Fortran, using MPI and under active development by physicists, mathematicians, etc.

In particular, the EPFL Hub has a high demand on porting codes to GPUs.

- ASCOT5
- CAS3D
- FELTOR
- GBS
- GENE

- GRILLIX
- GyselaX
- ORB5
- Soledge3X

How to 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)
- GENE (IPP)

- GRILLIX (IPP)
- GyselaX (cea irfm)
- ORB5 (EPFL)
- Soledge3X (cea irfm)

How to transition towards GPU codes

There are 3 main approaches:

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

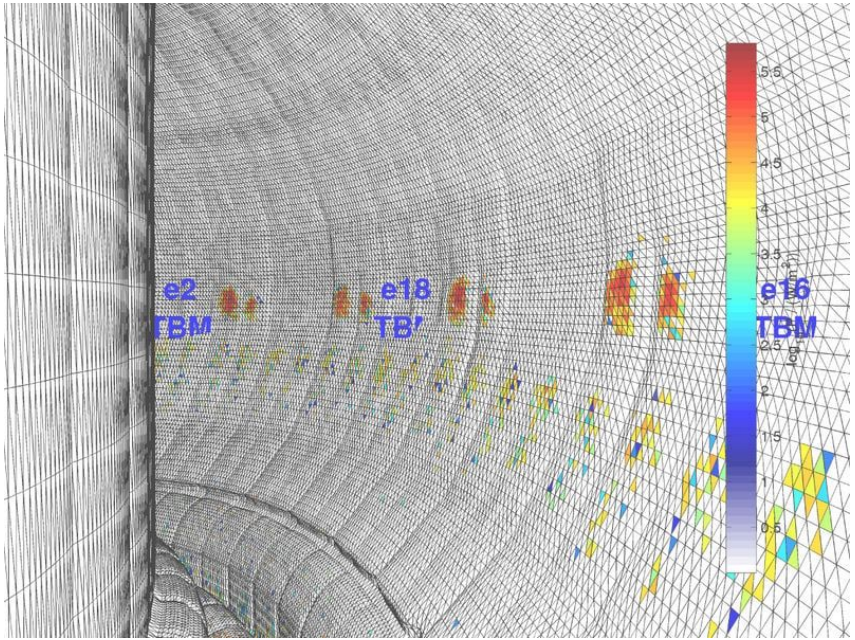
- ASCOT5 (Aalto University)
- CAS3D (IPP)
- FELTOR (DTU)
- GBS (EPFL)
- GENE (IPP)

- GRILLIX (IPP)
- GyselaX (cea irfm)
- ORB5 (EPFL)
- Soledge3X (cea irfm)

ASCOT5

G. Fourestey M. Peybernes (SCITAS)
J. Varje, S. Äkäslompolo, C. Gheller (ASCOT group - Aalto University)

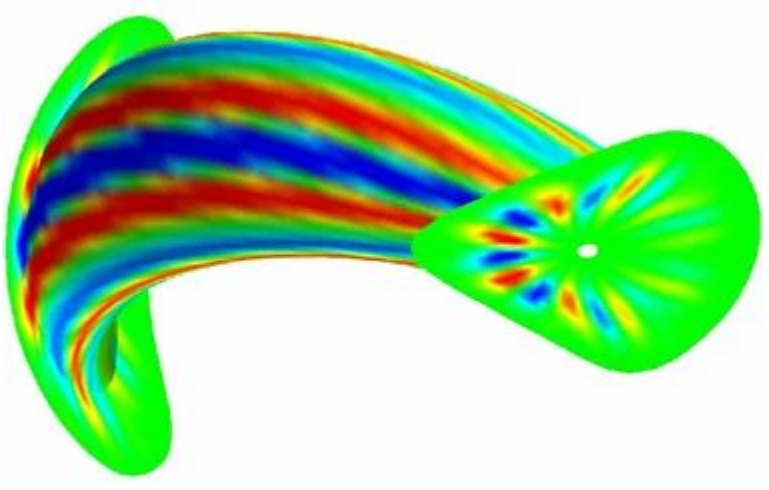
- ASCOT5 is used to study neoclassical transport
- The detailed magnetic fields and the first wall can be fully three-dimensional
- **Monte-Carlo** particle-following algorithm
- Particles are followed until thermalised/absorbed by a wall
- Collisions occur with a static Maxwellian background plasma
- No inter-particle collisions ⇒ Embarrassingly parallel
- Particles have different trajectories ⇒ different lifetime ⇒ load balancing problems



CAS3D

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

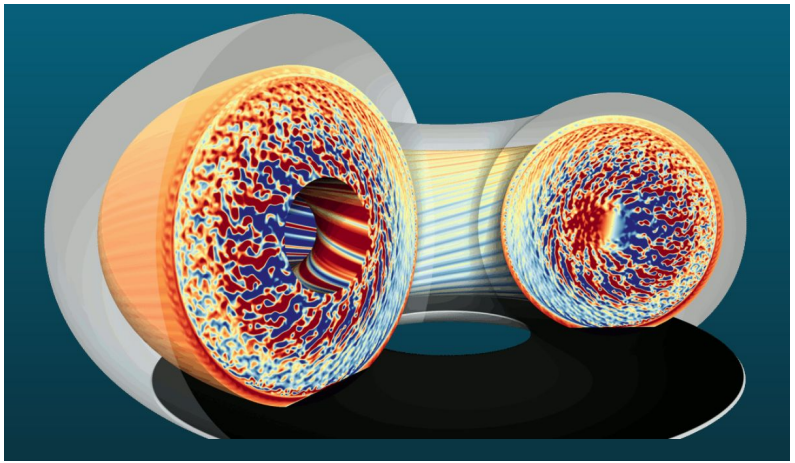
- CAS3D is used to study the ideal magnetohydrodynamic (MHD) properties of fusion toroidal plasmas
- The numerical scheme uses:
  - **Finite elements** along the radial direction
  - **Fourier decomposition** along the poloidal and toroidal directions
  - Inverse iteration algorithm to solve **eigenvalues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



GyselaX

Collaboration with M. Peybernes (SCITAS), J. Dechard (Eolen), V. Grandgirard, K. Obrejan, P. Donnel, D. Midou (CEA), P. Vezolle, P.E. Bernard (HPE), J. Noudohouenou (AMD), E. Malaboeuf, G. Gil (CINES)

- GYSELA-X is a global full-f nonlinear gyrokinetic code that simulates electrostatic plasma turbulence and transport in the core of Tokamak devices
- The numerical scheme uses:
  - **Semi-Lagrangian advection** scheme
  - **2d Finite elements** on the polar plane
  - **Fourier decomposition** along the toroidal direction
  - Iteration algorithm to solve sparse symmetric matrix problem.



Porting codes to GPU with pragmas

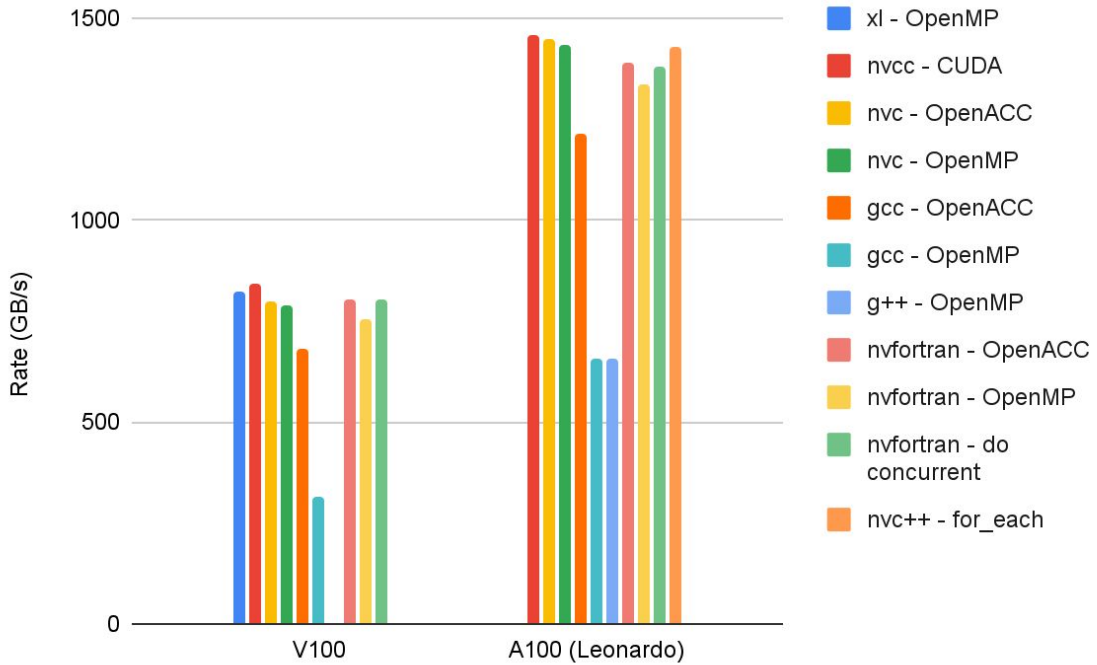
General strategy to port EUROfusion codes to GPU:

- Porting by directives with OpenMP offload and/or OpenACC
- 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
- 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)

Porting codes to GPU with pragmas

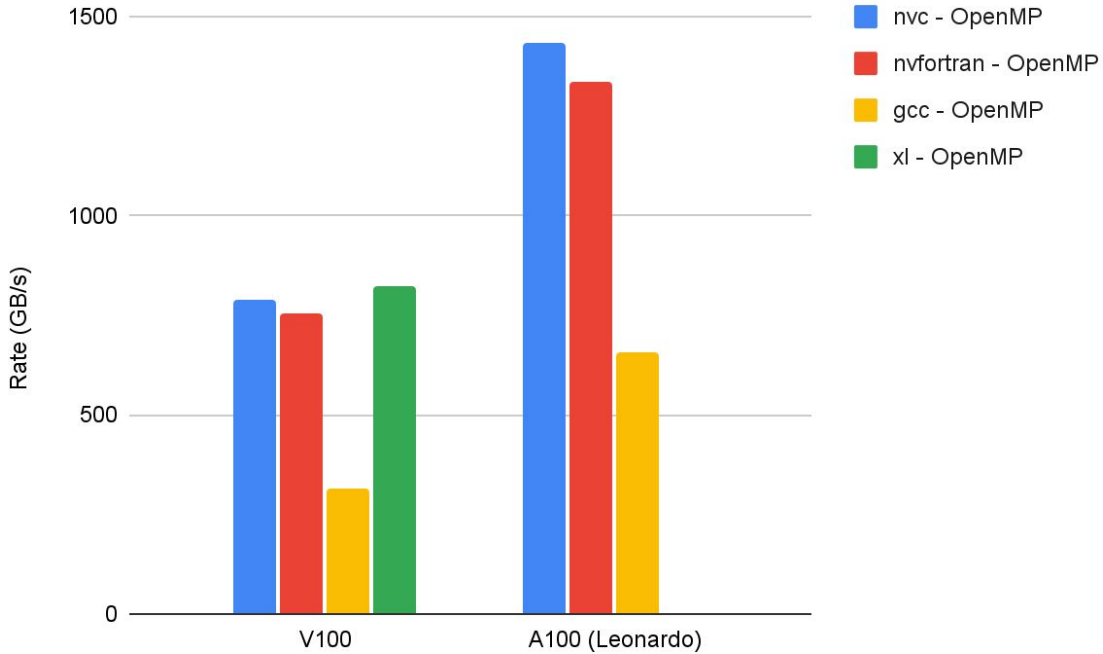
STREAM benchmark:



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

Porting codes to GPU with pragmas

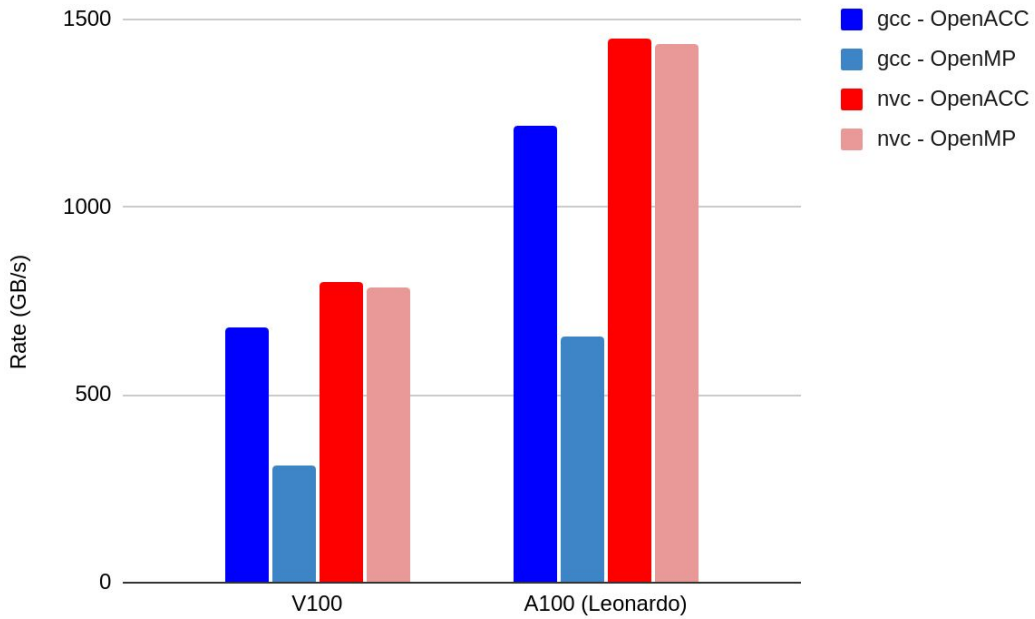
STREAM benchmark - OpenMP:



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

Porting codes to GPU with pragmas

STREAM benchmark - OpenMP vs OpenACC:



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

Porting codes to GPU with pragmas

OpenMP

```
#pragma omp target teams distribute private(iprt)
for(int iprt = 0; iprt < NbParticules ; iprt += NSIMD) {
        …some work…
        particle_simd_fo p; //new set of NSIMD particles
        #pragma omp parallel do simd
        for(int i=0; i< NSIMD; i++) {
            p.running[i] = 0;
            …some work…
        #pragma omp parallel do simd
        for(int i=0; i< NSIMD; i++) {
            …some work…
```

OpenACC

```
#pragma acc parallel loop gang private(iprt)
for(int iprt = 0; iprt < NbParticules ; iprt += NSIMD) {
        …some work…
        particle_simd_fo p; //new set of NSIMD particles
        #pragma acc loop worker vector
        for(int i=0; i< NSIMD; i++) {
            p.running[i] = 0;
            …some work…
        #pragma acc loop worker vector
        for(int i=0; i< NSIMD; i++) {
            …some work…
```

Pragmas

```
GPU_PARALLEL_LOOP_L0 private(iprt)
for(int iprt = 0; iprt < NbParticules ; iprt += NSIMD) {
        …some work…
        particle_simd_fo p; //new set of NSIMD particles
        GPU_LOOP_L1_L2
        for(int i=0; i< NSIMD; i++) {
            p.running[i] = 0;
            …some work…
        GPU_LOOP_L1_L2
        for(int i=0; i< NSIMD; i++) {
            …some work…
```

Typical problems when porting Fortran code

NGammaT3D - Original CPU version

```
if (any(b1_ptr(:,:,:).NE.0._dp).OR.any(bEM1_ptr(:,:,:).NE.0._dp).OR.any(vperp1(:,:,:).NE.0._dp)) then
   do itheta = ithetamin, ithetamax
      do iphi = iphimin, iphimax
         call NGammaT1D(mass, n_ptr(iphi,itheta,:), G_ptr(iphi,itheta,:), T_ptr(iphi,itheta,:), vperp1(iphi,itheta,:), &
              b1_ptr(iphi,itheta,:), bEM1_ptr(iphi,itheta,:), J_ptr(iphi,itheta,:), fluxN_psi(iphi,itheta,:), fluxG_psi(iphi,itheta,:), fluxE_psi(iphi,itheta,:), chi_ptr(iphi,itheta,:))
      enddo
   enddo
endif
```

Loop calling 1D function (limits parallelism)

Non-contiguous slices passed to function

NGammaT1D - Original CPU version

```
! Recovers the size of the arrays
Nz = size(n)

! Allocate memory for temporary arrays
allocate(v(1:Nz))
allocate(nl(1:Nz), vl(1:Nz), Tl(1:Nz))
allocate(nr(1:Nz), vr(1:Nz), Tr(1:Nz))

! Computes the parallel velocity
v = G / n

! Proceed to the WENO extrapolation of all the fields
call weno1D(n,nl,nr,2,chi)
call weno1D(v,vl,vr,2,chi)
call weno1D(T,Tl,Tr,2,chi)

! Applies thresholds on the extrapolated fields to avoid non
! physical values (negative densities, temperatures...)
nl = max(nl,NthreshMin)
nr = max(nr,NthreshMin)
Tl = max(Tl,TthreshMin)
Tr = max(Tr,TthreshMin)
```

Local allocations in loops

Vectorised operations

```
 ! Loop on cell faces and apply the Marquina's scheme
do iz = 2, Nz-2
   call NGammaTMarquina(mass,nl(iz),vl(iz),Tl(iz),vperpl(iz),bl(iz),Jl(iz), &
        nr(iz),vr(iz),Tr(iz),vperpr(iz),br(iz),Jr(iz),FluxM)
   fluxN(iz+1) = fluxN(iz+1) + FluxM(1)
   fluxG(iz+1) = fluxG(iz+1) + FluxM(2)
   fluxE(iz+1) = fluxE(iz+1) + FluxM(3)
enddo

! Clears memory
deallocate(v)
deallocate(nl, vl, Tl)
deallocate(nr, vr, Tr)
```

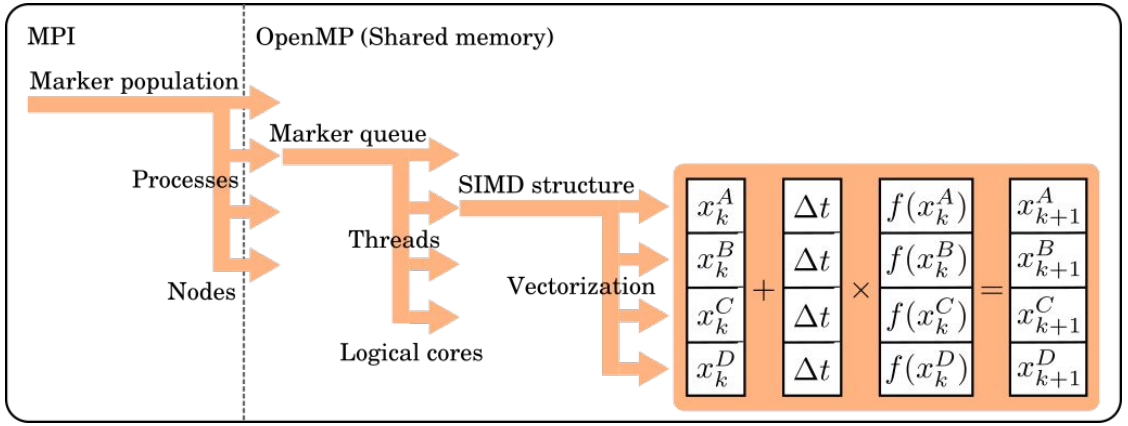
Loops in function called from a loop

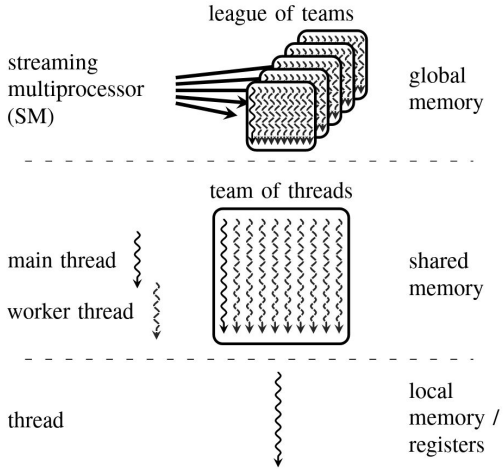
Fixed in OpenMP 6.0 (out Nov. 2024)

!$omp target teams workshare

Levels of Parallelism

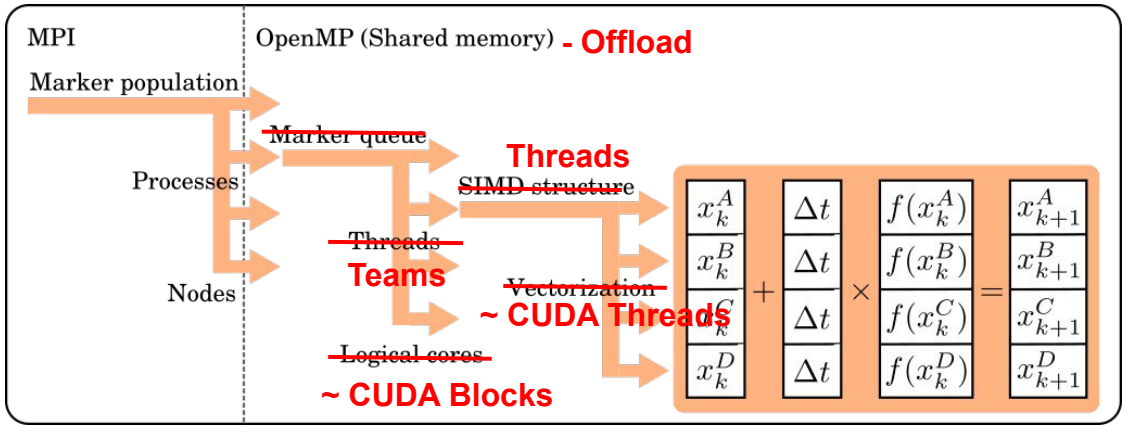
- The codes have 2 levels of CPU parallelism : MPI + OpenMP
- For GPU we aim to use **One MPI + two levels of parallelism**: MPI + OpenMP teams + threads/OpenACC gangs + workers
  - This is achieved via blocking and mimics the loops in the code (operations are not usually applied on all dimensions)
  - 1 block per streaming multiprocessor (SM)
- For ASCOT5 GPU blocking was based on preexisting CPU blocking designed for vectorisation

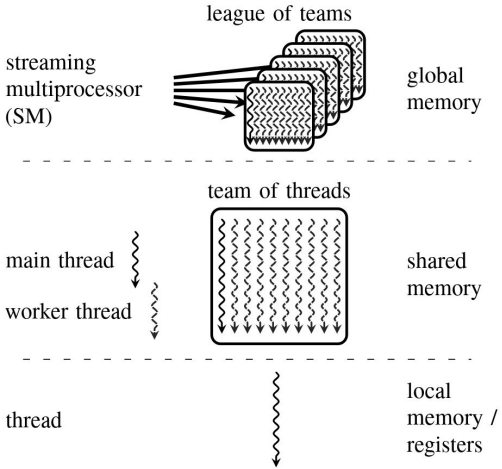




Levels of Parallelism

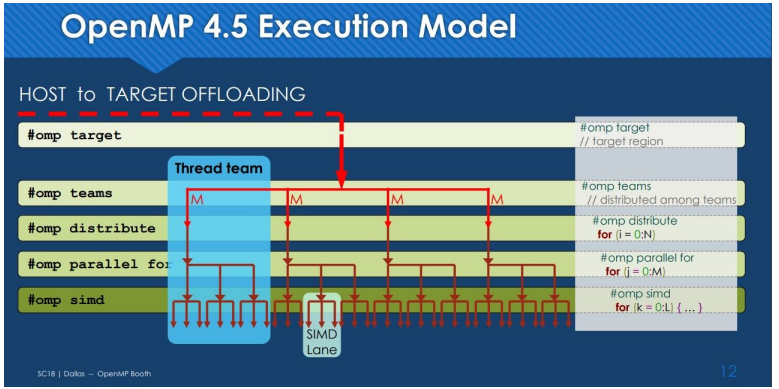
- The codes have 2 levels of CPU parallelism : MPI + OpenMP
- For GPU we aim to use **One MPI + two levels of parallelism**: MPI + OpenMP teams + threads/OpenACC gangs + workers
  - This is achieved via blocking and mimics the loops in the code (operations are not usually applied on all dimensions)
  - 1 block per streaming multiprocessor (SM)
- For ASCOT5 GPU blocking was based on preexisting CPU blocking designed for vectorisation





Levels of Parallelism

- OpenMP and OpenACC have similar coding paradigms under the hood
- In theory (according to the standards) the implementation of the levels adapt to the hardware, but 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

ASCOT5 Benchmarks

"May2022" Benchmark, comparison with different compilers/platforms

- gcc11 on x86 + v100 (Phoenix@EPFL)
- XL compilers + v100 (m100@Cineca)
- intel compilers on skylake and icelake (Jed@EPFL, ASCOT5 cpu-only)
- gcc11 with OpenACC on x86 + v100 (Phoenix@EPFL)

ASCOT5	TTS [s]	may2022_2dwall_go_analyticB			
	markers:	10000	100000		
				Platform	Compiler
m100@CINECA	**OMP Offload**	**46**	**473**	Power9 + v100	XL compilers
Phoenix@EPFL	**OMP Offload**	**232**	**2143**	6138 gold + v100	**gcc 11**
Phoenix@EPFL	**OpenACC**	**48**	**261**	6138 gold + v100	**gcc 11**
Helvetios@EPFL	**OMP (cpu-only)**	87	860	2x Gold 6140	intel compilers
Jed@EPFL	**OMP (cpu-only)**	31	318	2x Platinum 8360Y	intel compilers

New algorithmic approach

- The original implementation is not GPU-friendly:
  - **one very large kernel**
  - events depend on the previous event

- Implement a new version by **splitting the initial kernel**:
  - **Parallelize over events** instead of execute all particles
  - small kernels independent of each other

```
#pragma acc loop
for each particle
   while particle still running
      step_forward();
      end_condition_time();
      diag();
      …
   end while
end for
```

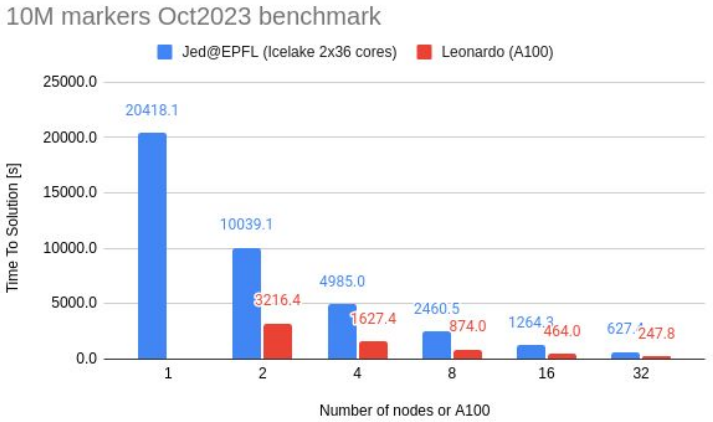
Initial method

```
while particles are still running
   #pragma acc loop
   for each particle still running
      step_forward();
   #pragma acc loop
   for each particle still running
      end_condition_time();
   #pragma acc loop
   for each particle still running
      diag();
      …
end while
```

Event-Based

Benchmarks

- "Sept2023" Benchmark:
  - **Jed**: 2x Platinum 8360Y, intel/2021.6.0, -Ofast -qopt-zmm-usage=high -march=native
  - **Leonardo**: A100, nvhpc/23.1, -O3 -acc -gpu=managed
  - **Time-to-Solution** in seconds



Memory consumption with team blocking

- Multiple levels of GPU parallelism requires blocking
- Independent memory required for each thread ⇒ potentially large memory requirements (3D/5D)
- Allocating over the number of teams limits memory but requires accessing a 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_team_id() {
        return __pgi_gangidx() + 1;
}

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

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

#endif
```

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
```

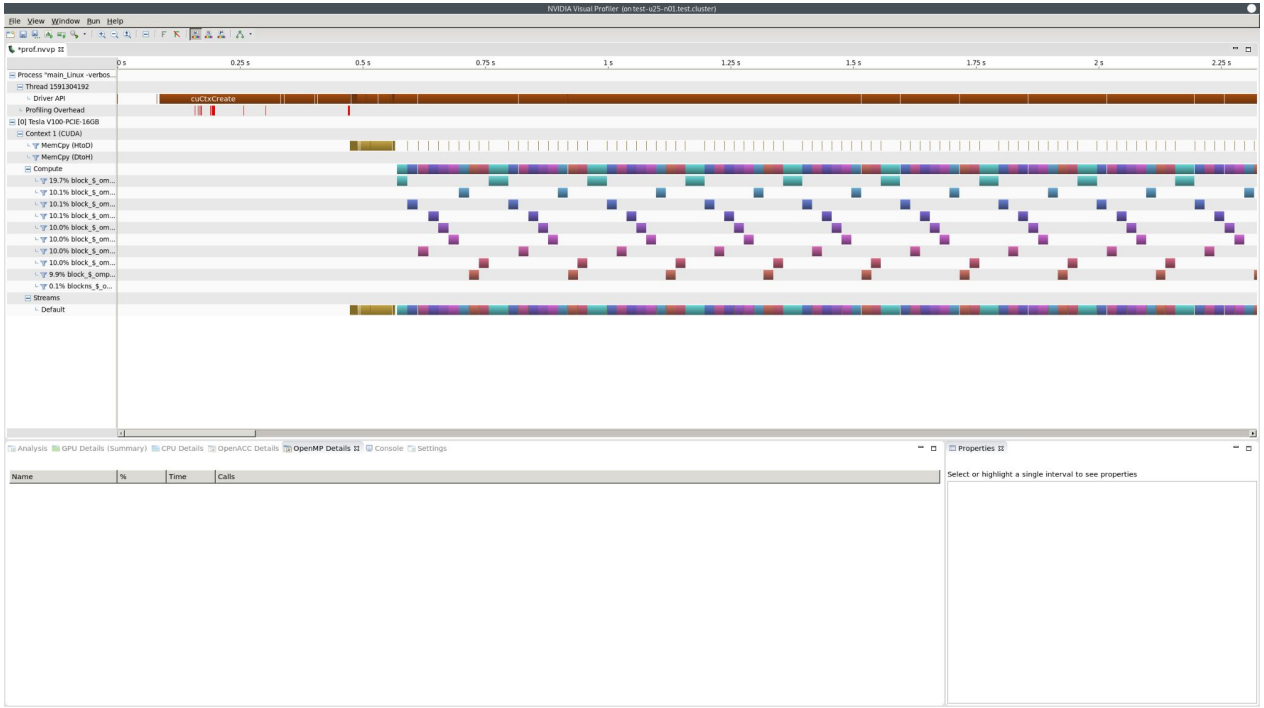
```
allocate(Tab1(N,nbgangs))
allocate(Tab2(N,nbgangs))

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  = …
```

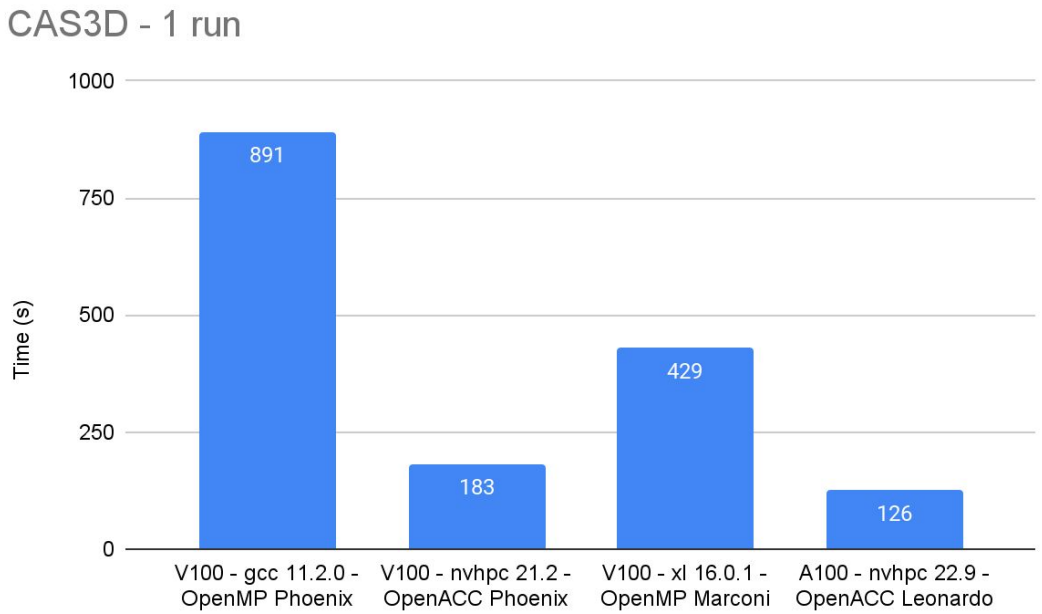
CAS3D - GPU Profiling

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



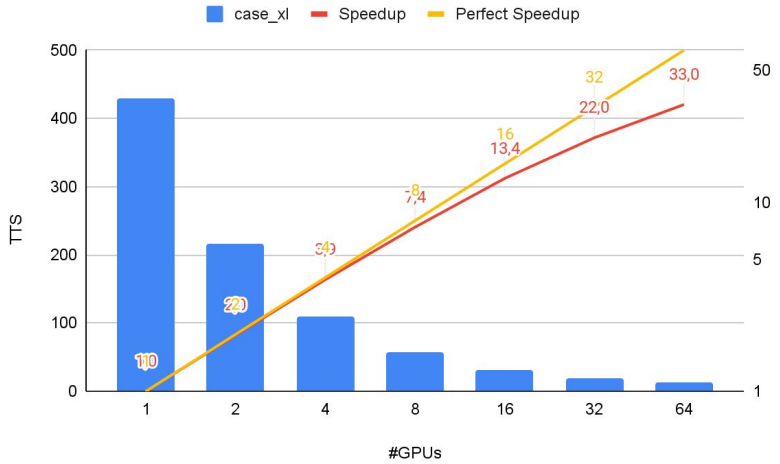
CAS3D - GPU Results

- GPU - OpenMP-Offload vs OpenACC



CAS3D - MPI+GPUs Results

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



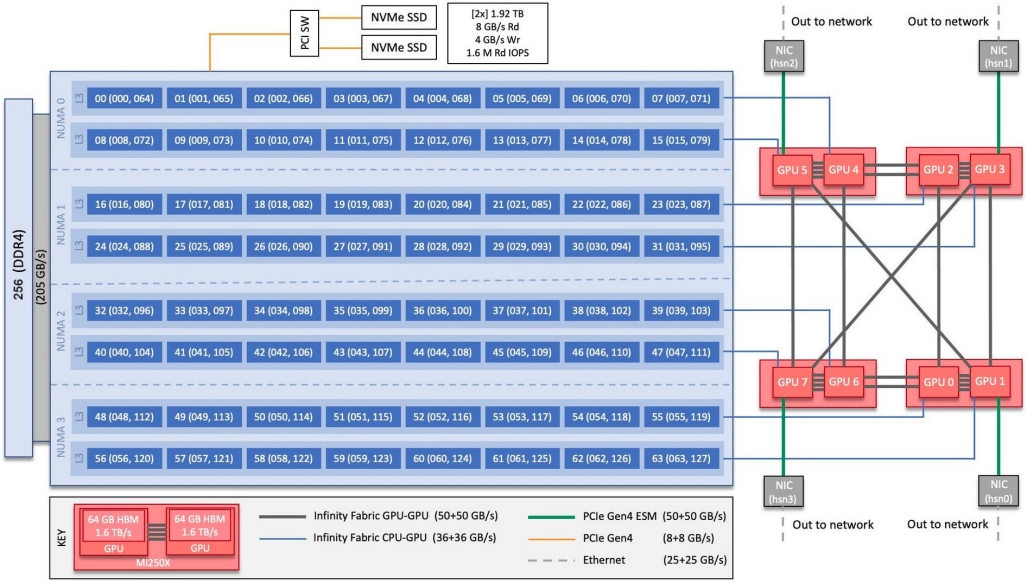
XL-compiler on M100	1 node (4GPUs)	2 nodes	4 nodes	8 nodes	16 nodes	GPU-serial (640 teams - 128 threads)	CPU: 1node (32-cores)
#GPUs	4	8	16	32	64	1	0
case_mm	4 s	3,2 s	3 s	3,1 s	3,3 s	6 s	5 s
case_ll	78 s	42 s	24 s	15 s	14,8 s	299 s	190 s
case_xl	110 s	58 s	32 s	19,5 s	13 s	410 s	418 s

GPU-direct for MPI communications

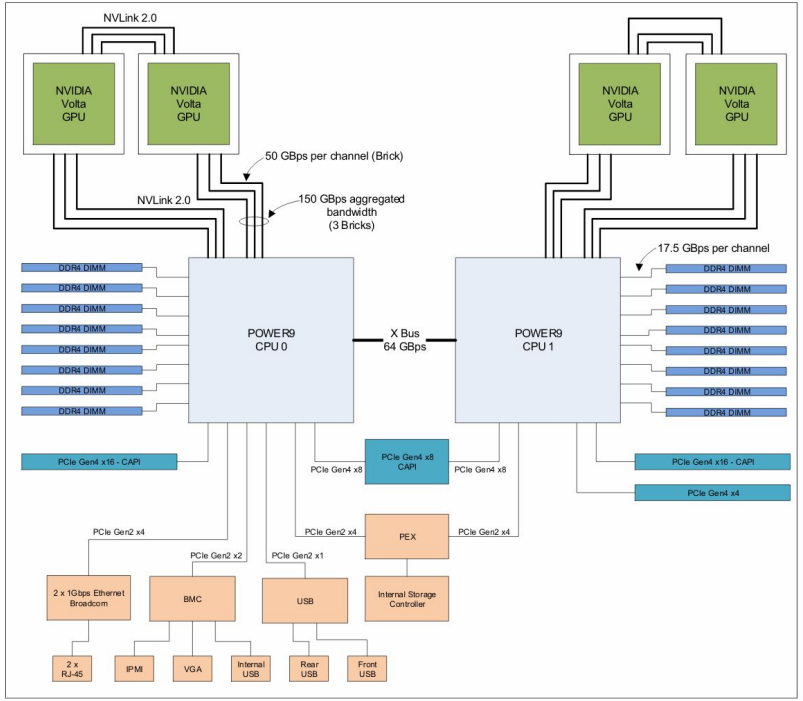
- Some compute systems provide functionalities for GPU-GPU direct communications
- GPU-direct MPI communications can exploit this
- The effectiveness of these communications can depend on the node configuration
  - Each Adastra accelerated compute node consists of one 64-core AMD Trento Optimized 3rd Gen EPYC CPU and four AMD Instinct MI250X accelerators.
  - Each M100 accelerated compute node consists of two 16-core IBM POWER9 AC922 CPU and four NVIDIA Volta V100 accelerators with NVLink.

Node configurations

ADASTRA

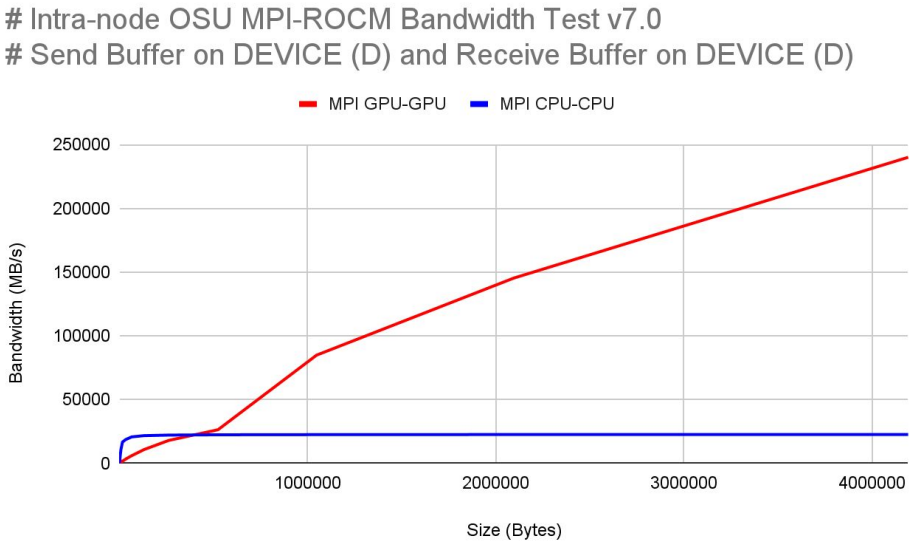


M100

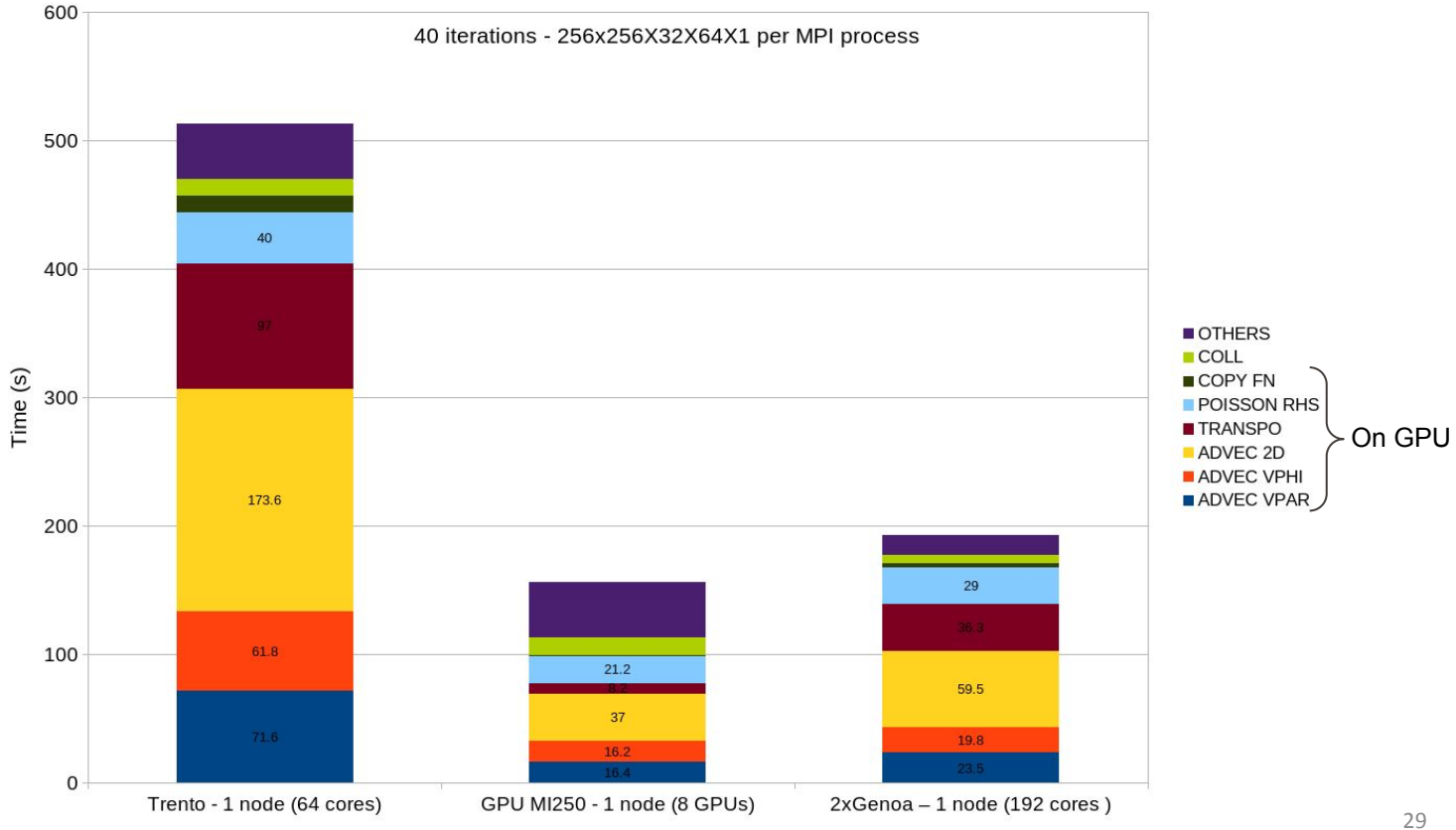


GPU-direct for MPI communications

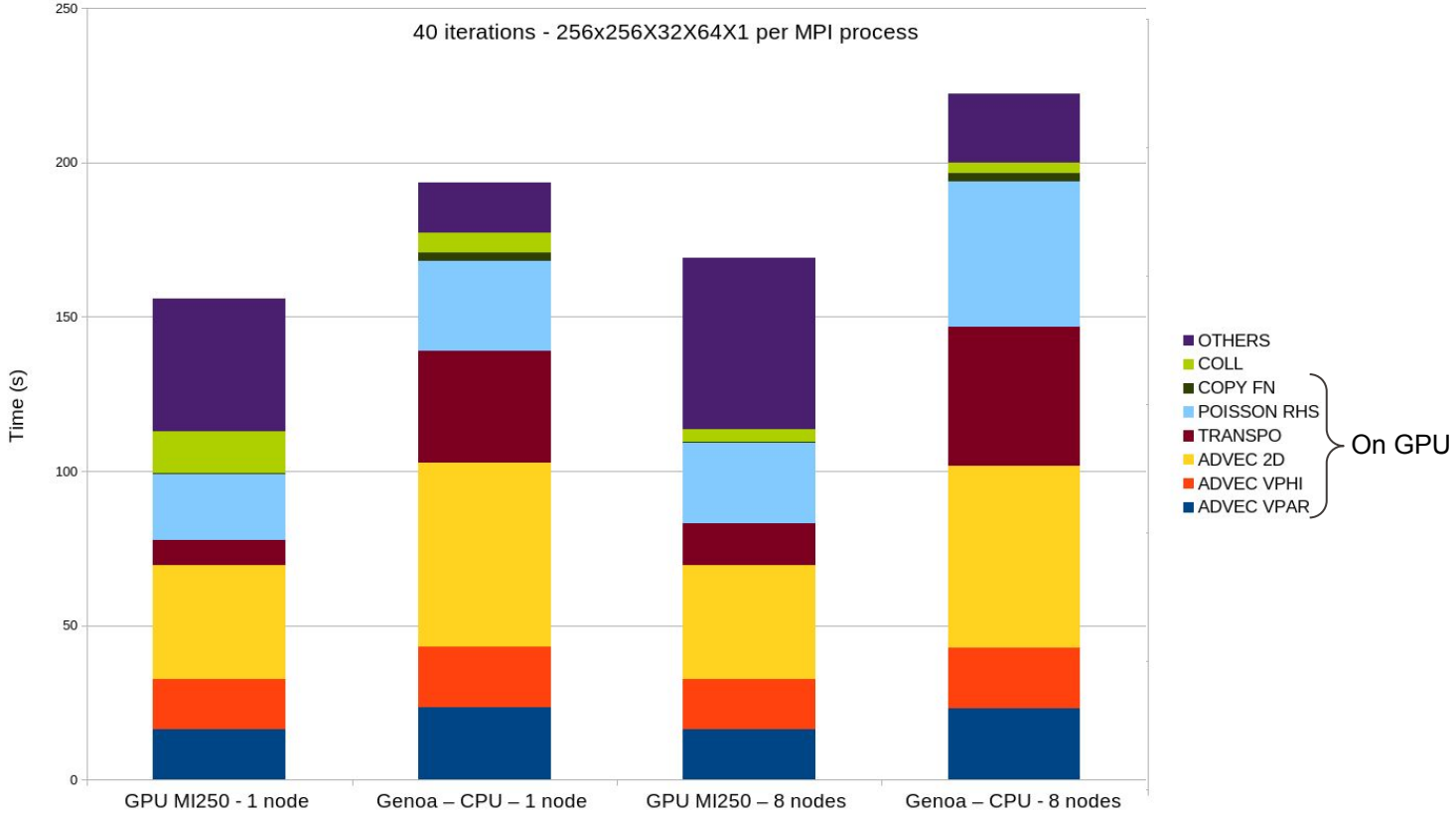
- GPU-direct is very beneficial for large data transfers
- Small data transfers do not benefit from GPU-direct on ADASTRA
- 0.5MB cutoff ⇒ grid of 256x256 floats
- GPU-direct should only be used for 3D data or relatively large 2D cases on ADASTRA
- Useful for Gysela with 5D All-to-All communications



Performance

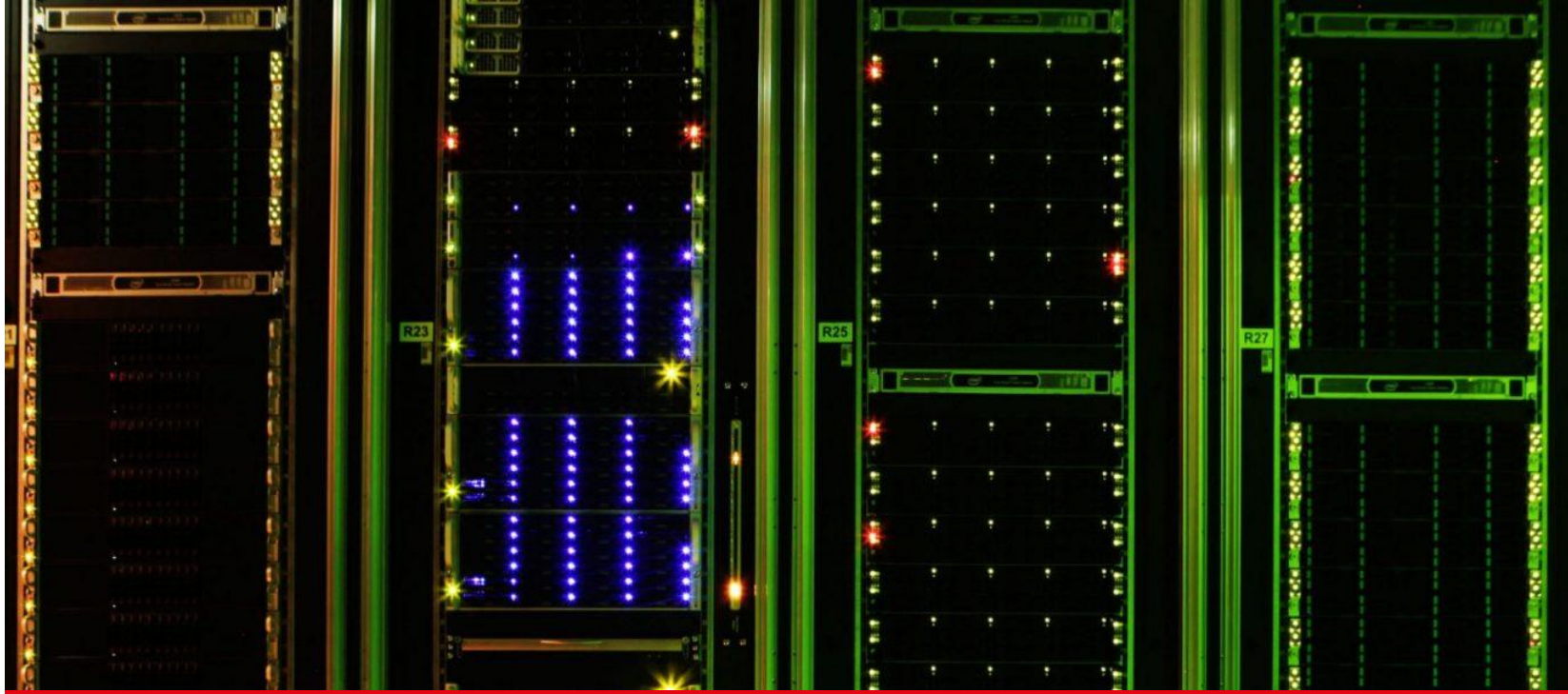


Performance - Weak Scaling



Conclusions

- Porting by directives with OpenMP offload and/or OpenAC
- 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 Eurofusion CAS3D, ASCOT5 and GYSELAX codes
- Leonardo (Nvidia A100) is ~3x faster than Marconi 100 (Nvidia V100) for CAS3D, ASCOT5
- Most of CPU algorithms have to be modified further to get better performance on GPU
- Other techniques are currently used in Eurofusion codes such as Cuda (GBS code), Kokkos (GYSELAX code)


R23
R25
R27