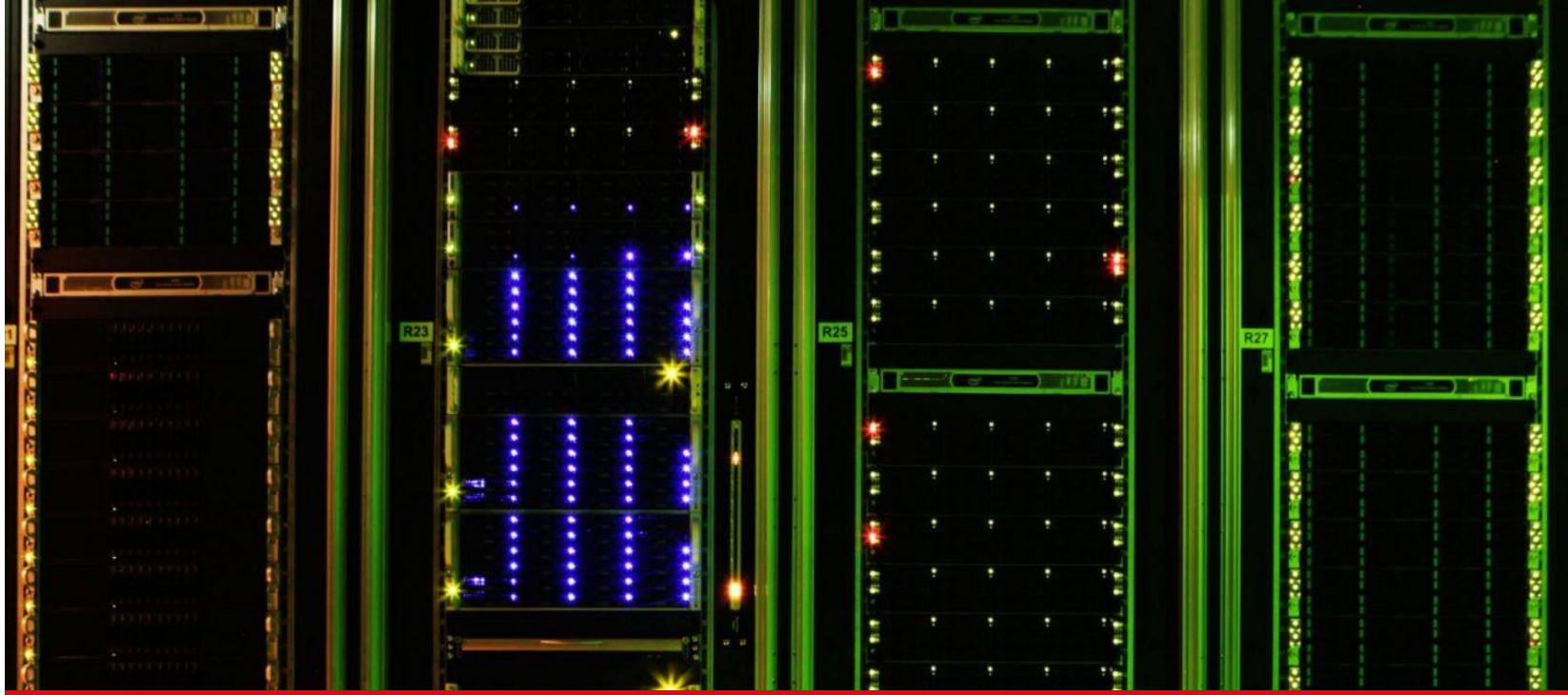


GPU Porting Paradigms in GyselaX

E.Bourne[1], M. Peybernes[1]

[1]Ecole Polytechnique Fédérale de Lausanne (EPFL), SCITAS

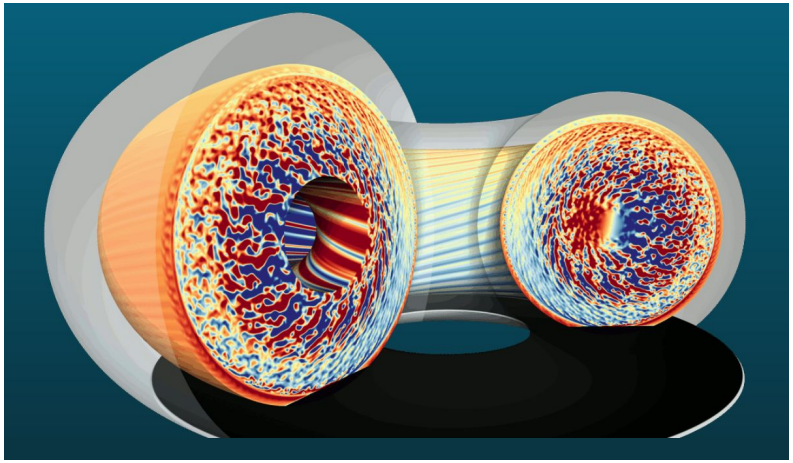
ACH - November 2023

Overview

- Gysela - State of the Art
  - Understanding the Gysela(X) code
  - CPU parallelism
- Part 1 : Porting Gysela to GPU in Fortran
  - GPU porting strategy in Fortran
  - GPU Aware MPI communications
  - Performance Results
- Part 2 : GyselaX: Leveraging C++
  - Why C++?
  - Choosing a GPU Porting Paradigm
  - DDC : Type Casting Discrete Computations

Gysela(X) - State of the Art

- Non-linear 5D simulations (3D in space + 2D in velocity)
- Multi-scale problem in space and time
- Unique gyrokinetic code based on a semi-Lagrangian scheme modelling both core & edge plasmas
- Intensive use of petascale resources: ~ 150 Mhours / year
- Exascale needs for ITER plasma turbulence simulation with electromagnetic effects

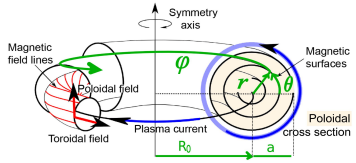


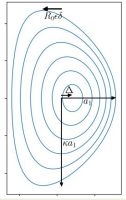
Physical Model

Geometry

mesh
(equidistant in (r,θ,φ))

magnetic configuration
(circular cross-section or D-shape)





Vlasov

5D Vlasov Solver for D + W
(semi-lagrangian scheme)

+

Kinetic electrons

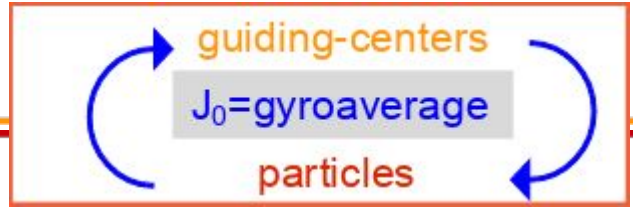
$$B^*_{\|s}\frac{\partial \bar{F}_s}{\partial t} + \nabla\cdot\left(\frac{d\mathbf{x_G}}{dt}B^*_{\|s}\bar{F}_s\right) + \frac{\partial}{\partial v_{G\|}}\left(\frac{dv_{G\|}}{dt}B^*_{\|s}\bar{F}_s\right) = C(\bar{F}_s) + S + \mathcal{K}_{\text{buff}}(\bar{F}_s) + \mathcal{D}_{\text{buff}}(\bar{F}_s)$$

with the equations of motion:

$$B^*_{\|s}d_t\mathbf{x}_G = v_{G\|}\mathbf{B}^* + \frac{1}{e}\mathbf{b}\times\nabla\Lambda$$

$$B^*_{\|s}m_s d_t v_{G\|} = -\mathbf{B}^*\cdot\nabla\Lambda$$

where $\mathbf{B}^* = \mathbf{B} + (m_s v_{G\|}/e)\nabla\times\mathbf{b}$ and $\Lambda = eJ_0\phi + \mu B$;



Poisson

3D Poisson Solver (Finite Elements in (r,θ) + Fourier in φ)

$$\frac{e}{T_{e,\text{eq}}}\left(\phi - \langle\phi\rangle\right) - \frac{1}{n_{e_0}}\sum_s Z_s\nabla_\perp\cdot\left(\frac{n_{s,\text{eq}}}{B\Omega_s}\nabla_\perp\phi\right) = \frac{1}{n_{e_0}}\sum_s Z_s\int J_0\cdot\left(\bar{F}_s - \bar{F}_{s,\text{eq}}\right)d^3v$$

Existing CPU Parallelism

- Hybrid parallelization MPI/OpenMP
- Evolves the distribution function Fs on 5 dimensions (r, theta, phi, vpar, μ) :
- MPI decomposition along (r, theta) + 1 MPI communicator for each value of μ
- MPI Transposition MPI (r, theta) <-> (phi, vpar) necessary for advection steps

for *time step* $n \geq 0$ **do**

Field solver, Derivatives' computation, Diagnostics

Vlasov solver:

- 1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
- 1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
- **Transpose** of f
- **2D Advection** in (r, θ) $(\forall(\mu, \varphi, v_\parallel) = [local], \forall(r, \theta) = [*])$
- **Transpose** of f
- 1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
- 1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$

GPU porting strategy

- Exploit 2 levels of parallelism Teams/Threads to map algorithms on GPU architectures :
  - OpenMP teams mapped on Compute Unit (CU)
  - OpenMP team threads mapped on Scalar Unit / Part of Vector (SIMD) Unit
- Exploit parallelism using loop blocking:
  - distribute the work among teams and threads with a new parameter setting blocks size

Common characteristics among routines:
- Nested subroutine calls in each single kernel
- BLAS/LAPACK calls for small matrices matrices
- Several loop levels (code 5D)

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

GPU porting strategy

The initial collision module, for instance, has the following OpenMP-**CPU** layout:

```
!$OMP PARALLEL default(none) &
!$OMP private(iphi, itheta, ir,  RHS_vec) &
!$OMP shared(Nvpar,Npolmax_,index_max_,collstart,collend, &
!$OMP  jstart,jend,istart,iend)

allocate(RHS_vec(Npolmax_,Nvpar))

do iphi = collstart,collend
  !$OMP DO COLLAPSE(2)  SCHEDULE(GYS_OMP_DYNAMIC,1)
  do itheta = jstart,jend
    do ir = istart,iend

      call collvparmu_general_CvCd ( self, Npolmax_, &
                  ir, itheta, iphi, dt, &
                  RHS_vec, A_mat, B_mat, C_mat, Id_mat, D_mat, &
                  DL_mat, DU1_mat, DU2_mat, IPIV_mat, &
                  alpha_ivpar, beta_ivpar)
```

CPU

- TEAMS are used with a blocking along (r,theta,phi) and dynamical allocation
- THREADS are used in subroutines (e.g. collvparmu_general_CvCd)

GPU porting strategy

- Blocking allows the distribution of a well balanced workload over teams and threads
- The dimension *nspace* depend on blocks size
- Allow the algorithm to be set up according to a given hardware architecture
- Avoid having, for instance, not enough work for threads in a team

```
allocate( RHS_vec_batched (Nspace,Npolmax_,Nvpar, NbParallel ))

#ifdef GPU
  !$OMP TARGET
  !$OMP TEAMS DISTRIBUTE COLLAPSE(3) num_teams(NbParallel)
#else
  !$OMP PARALLEL default(none) &
  !$OMP private(iphi, phi_start, itheta, itheta_start, &
  !$OMP  ir, ir_start) &
  !$OMP shared(Nvpar,Npolmax_,index_max_,collstart,collend, &
  !$OMP  jstart,jend,istart,iend, &
  !$OMP  iphi_block_size,itheta_block_size,ir_block_size, &
  !$OMP   RHS_vec_batched )
  !$OMP DO COLLAPSE(3) SCHEDULE(GUIDED)
#endif
     do iphi_start = collstart,collend,iphi_block_size
       do itheta_start = jstart,jend,itheta_block_size
         do ir_start = istart,iend,ir_block_size
           ir_end     = ir_start     + ir_block_size     - 1
           itheta_end = itheta_start + itheta_block_size - 1
           iphi_end   = iphi_start   + iphi_block_size   - 1

GPU        call collvparmu_general_CvCd( RHS_vec_batched )
```

work distributed over teams

NbParallel = NbTeams on GPU
NbParallel = NbThreads on CPU

Arrays allocated according to the number of teams to avoid dynamical allocations

```
  do ivpar = 1, Nvpar
     !DIR$ INLINE
     call collvparmu_general_matrix_blocks( Npol, Nspace, ivpar,
dvpar, deltat, A_mat_dt(0:,0:,0:, tid), B_mat_dt(0:,0:,0:, tid),
C_mat_dt(0:,0:,0:, tid) )
!$OMP OMP PARALLEL DO
     do ispace = 0, Nspace
        ..........
```

Nspace = iphi_block_size,itheta_block_size ,ir_block_size

work distributed among threads of a team

GPU-aware MPI communications

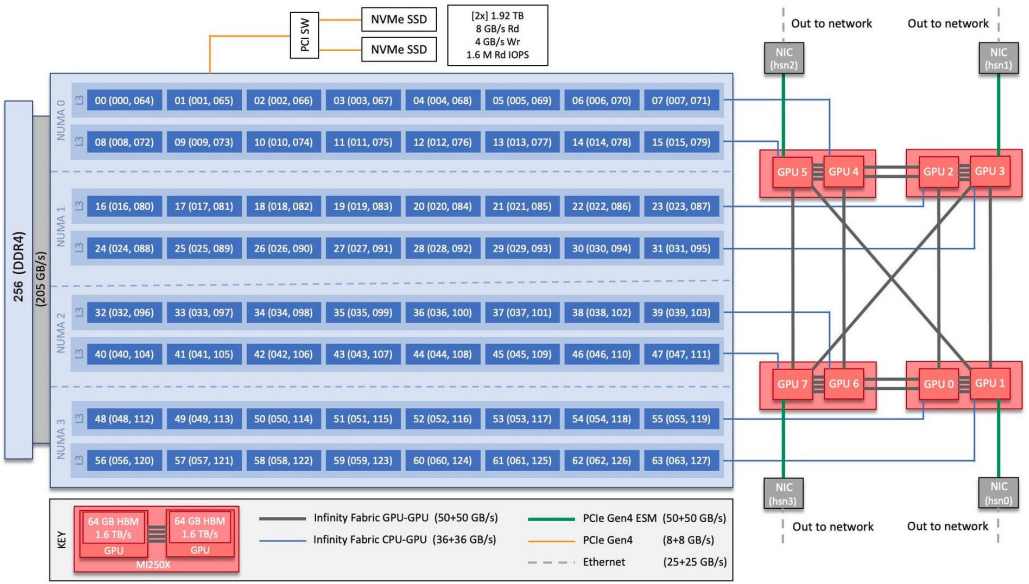
- Some compute systems provide functionalities for GPU-GPU direct communications
- GPU-direct MPI communications can exploit this
- The effectiveness of these communications can depend on the node configuration
  - Each Adastra accelerated compute node consists of one 64-core AMD Trento Optimized 3rd 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.

```
!$OMP TARGET ENTER DATA MAP (ALLOC:f_send,f_recv)
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO COLLAPSE(2)
do vpar_id = 0,Nbproc_vpar-1
  do ivpar = 0,dom_vpar-1
    do phi_id = 0,Nbproc_phi-1
      do iphi = 0,dom_phi-1
        do itheta = jstart, jend
         do ir = istart, iend
           offset = vpar_id*Nbproc_phi+phi_id
           phi_offset  = phi_id * dom_phi
           vpar_offset  = (vpar_id * (Nvpar+1)) / Nbproc_vpar
           f_send(ir, itheta, iphi, ivpar, offset) = &
                       fval(ir, itheta, phi_offset+iphi, vpar_offset+ivpar)
         end do
        end do
      end do
    end do
  end do
end do
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
!$omp target data use_device_addr(f_send,f_recv)
call MPI_Alltoall( f_send, sendsize, MPI_REAL8, f_recv, sendsize,
MPI_REAL8, mpi_comm_mu, ierr )
!$omp end target data
```

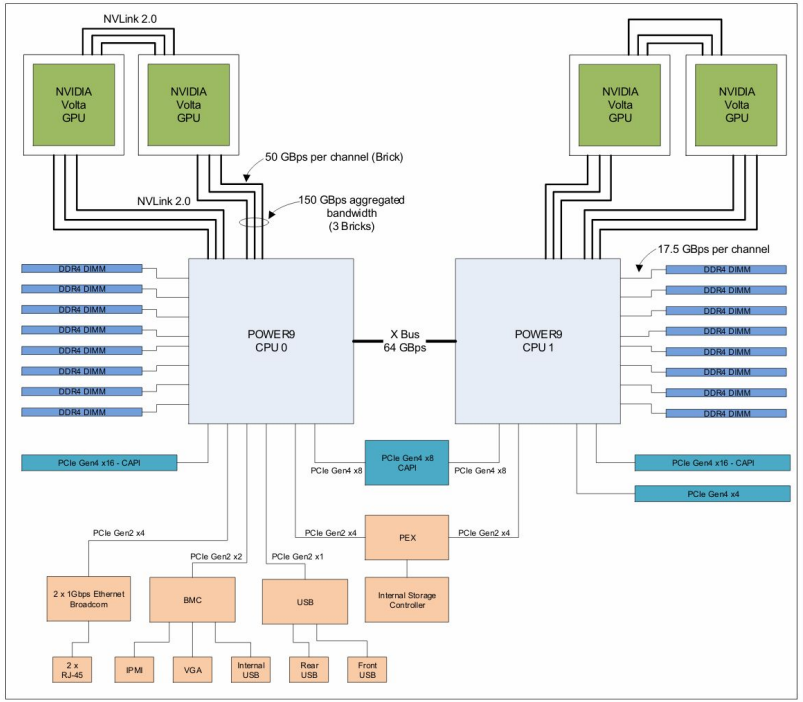
GPU

Node configurations

ADASTRA

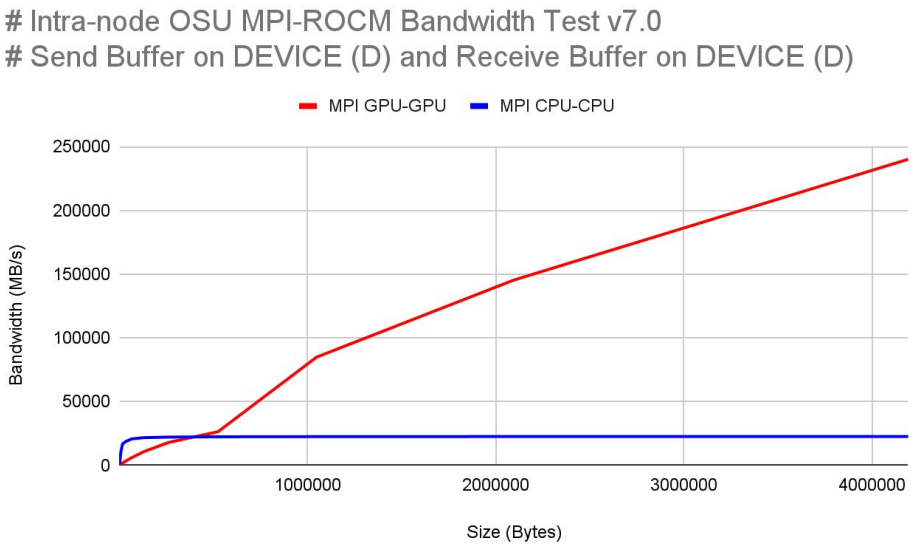


M100

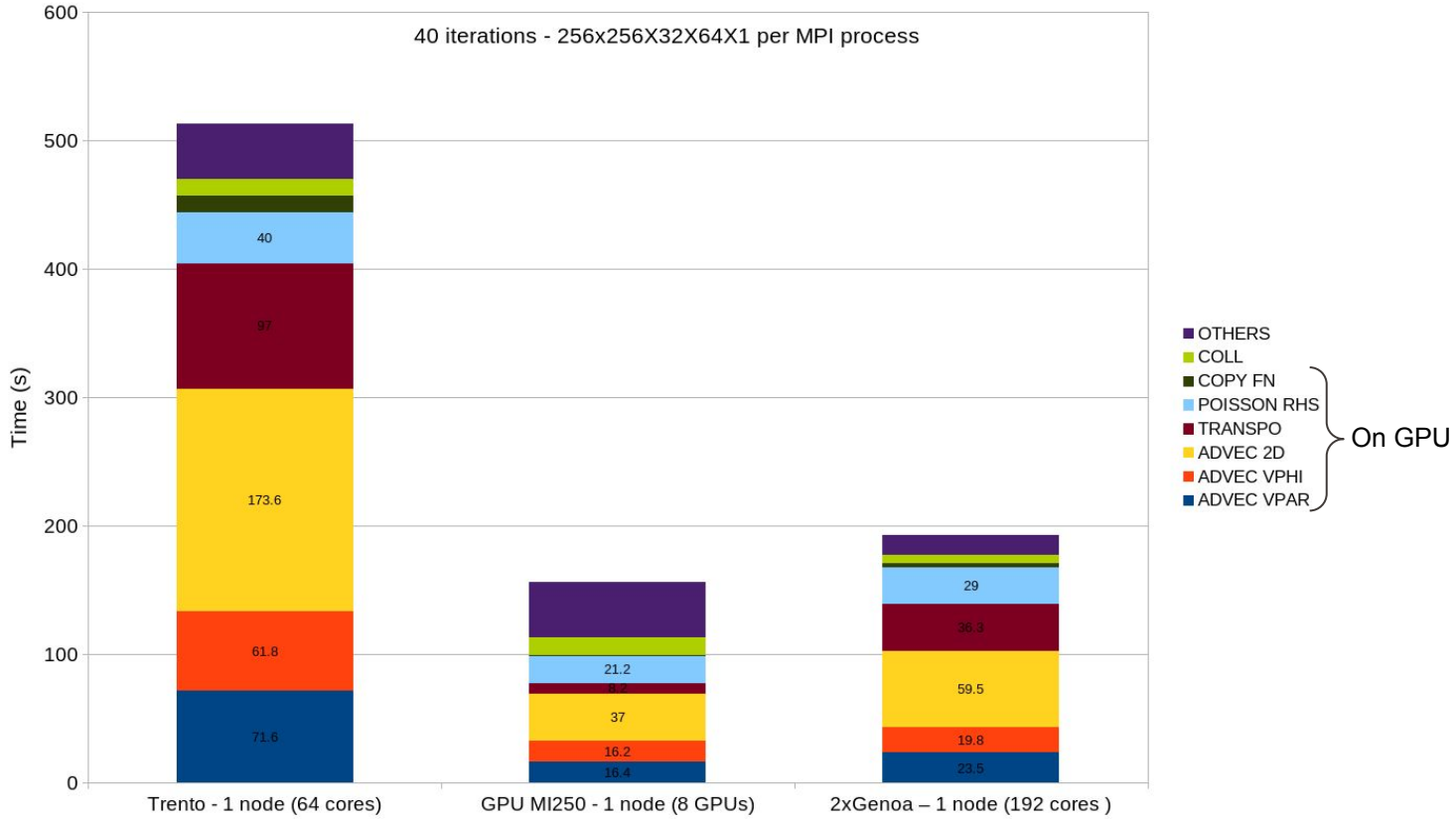


GPU-aware MPI communications

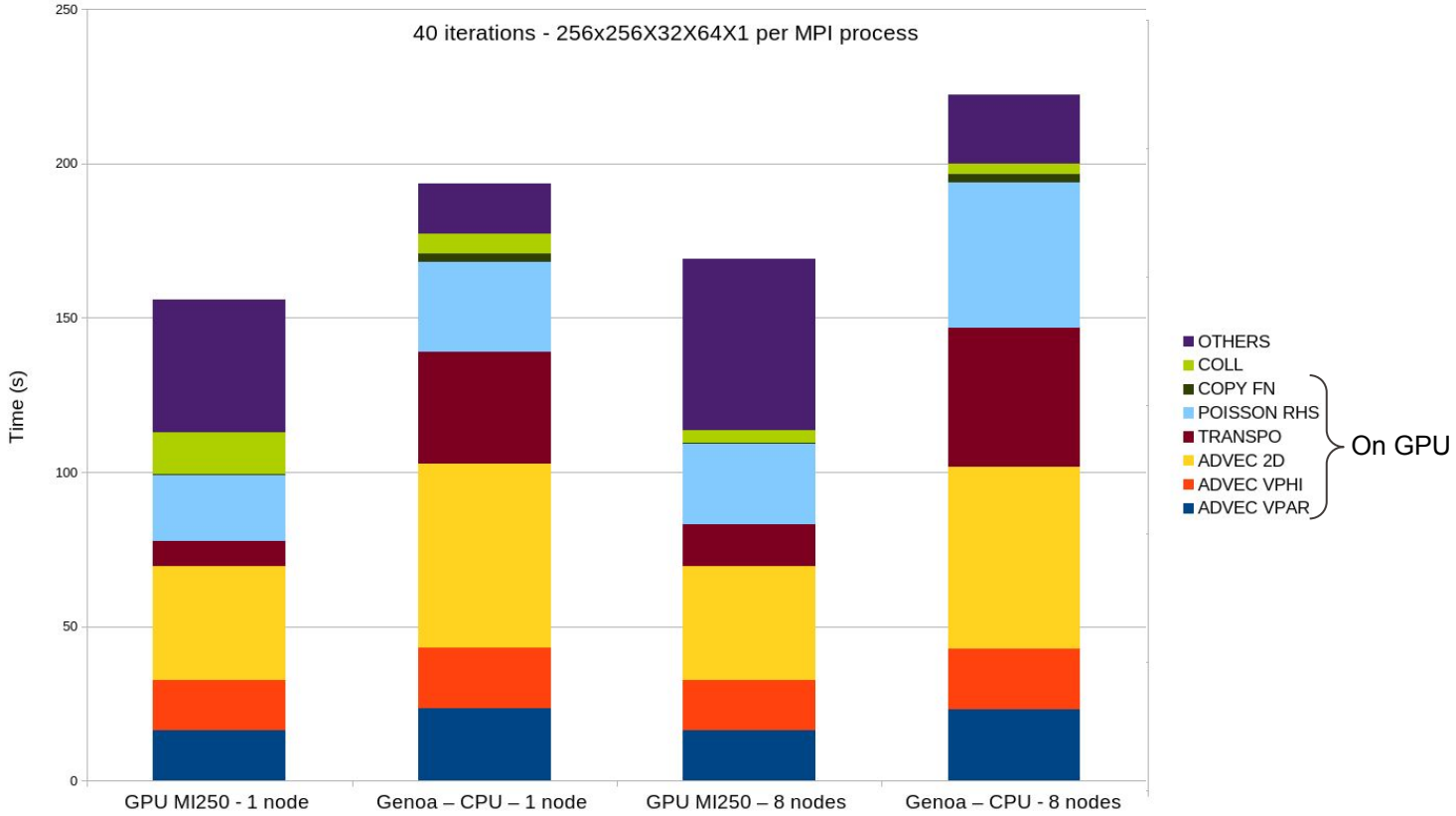
- GPU-direct is very beneficial for large data transfers
- Small data transfers do not benefit from GPU-direct on ADASTRA
- 0.5MB cutoff ⇒ grid of 256x256 floats
- GPU-direct should only be used for 3D data or relatively large 2D cases on ADASTRA
- Useful for Gysela with 5D All-to-All communications



Performance



Performance - Weak Scaling

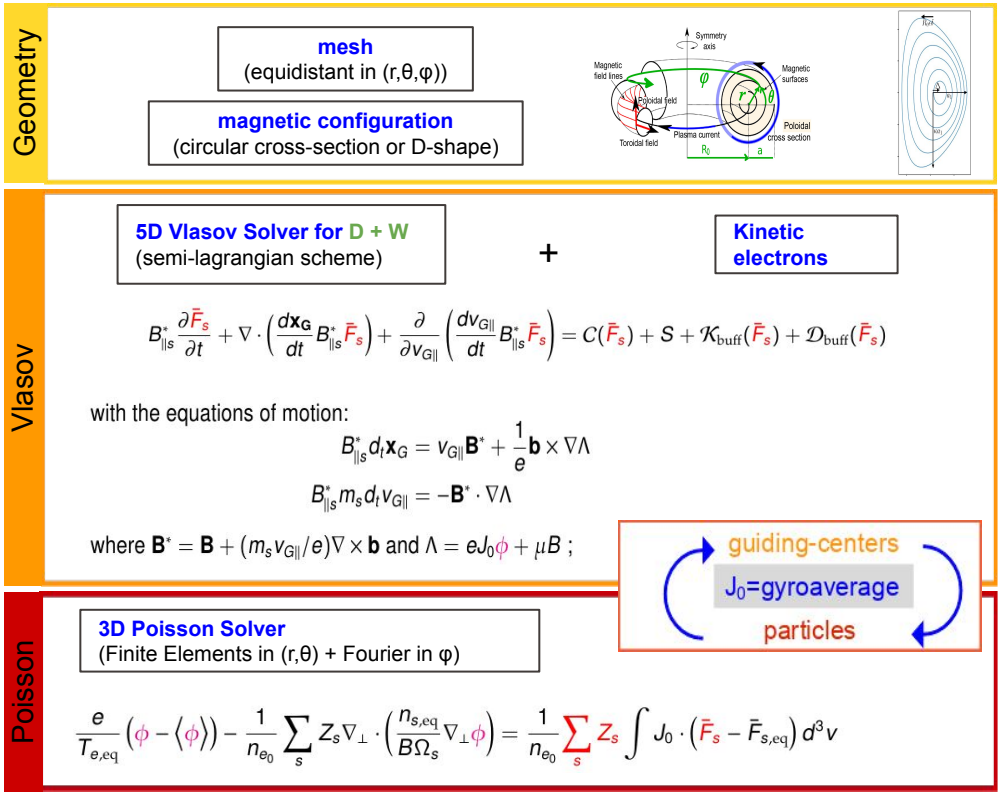


Part 2 : GyselaX: Leveraging C++

GyselaX: Leveraging C++

Why C++?

- Non-equidistant meshes and multi-patches to be introduced
  - Splines rewritten
  - Needed for Vlasov
  - Needed for Poisson
  - => near total re-write
- The original Fortran code is old (2001)
  - Global variables
  - Unwieldy structures
- C++ has larger community
  - More linters
  - More libraries
  - More compiler support



GyselaX: Leveraging C++

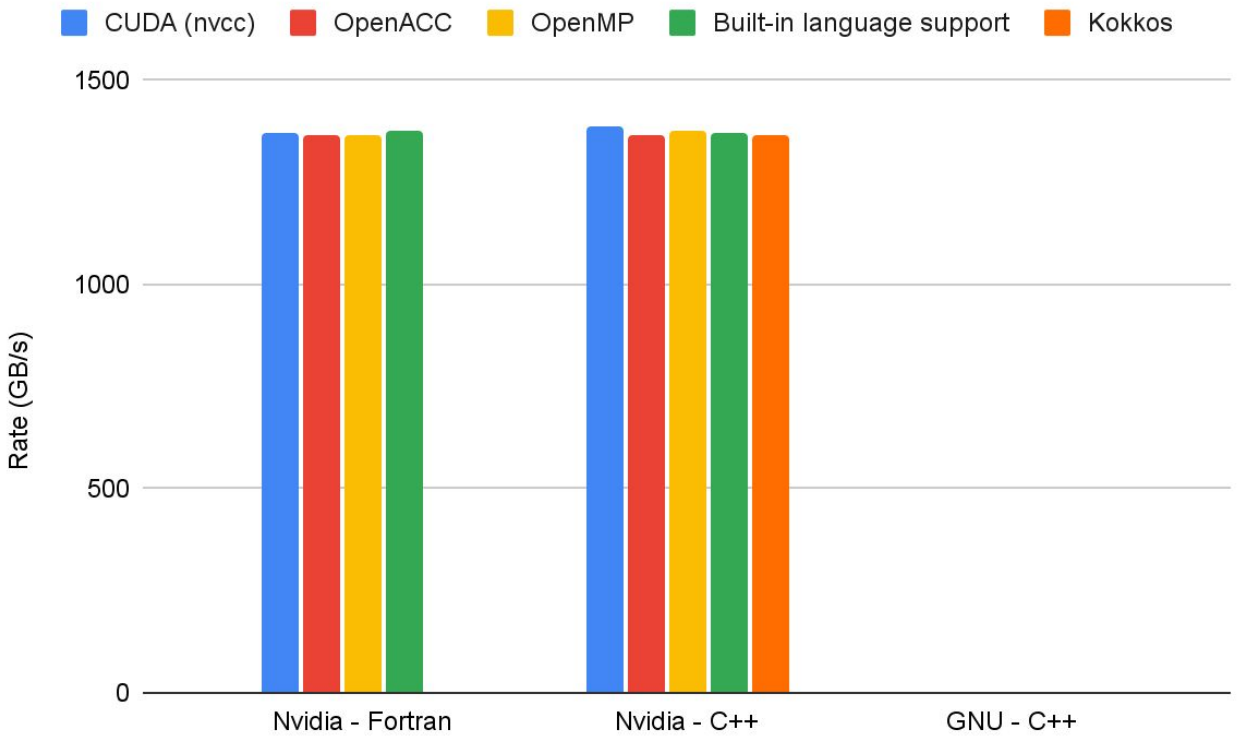
GPU Porting Paradigms

There are three main approaches:

- Pragma Directives
  - \+ Simple commands
  - \+ (Mostly) Portable
  - \- OpenMP vs OpenACC
  - \- CPU vs GPU
- Library Encapsulation
  - \+ Simple commands
  - \+ Portable
  - \- Compiler dependent results
- Cuda/HIP/…
  - \+ Fine-grain control
  - \- No portability
  - \- Harder to read for non-specialists

GyselaX: Leveraging C++

GPU Porting Paradigms - Stream Benchmark



https://github.com/UoB-HPC/BabelStream

GyselaX: Leveraging C++

GPU Porting Paradigms - Stream Benchmark

```
c = a;
```

```
std::transform(std::execution::par_unseq, c.begin(), c.end(), b.begin(), [scalar](auto cj) {return scalar*cj;});
```

```
std::transform(std::execution::par_unseq, a.begin(), a.end(), b.begin(), c.begin(), [scalar](auto cj) {return aj + bj;});
```

```
std::transform(std::execution::par_unseq, b.begin(), b.end(), c.begin(), a.begin(), [scalar](auto cj) {return bj + scalar*cj;});
```

Serial	Parallel
22426	4837
13297	22477
14475	28596
14631	1427595

DDC : Type Casting Discrete Computations

- C++ allows us to make mathematical concepts type safe
- DDC is a wrapper around kokkos which provides mathematical types templated by dimension
  - `Coordinate<Dim>`
  - `DiscreteDomain<PointSampling<Dim>>`
  - `DiscreteElement<PointSampling<Dim>>`
  - `Chunk<double, DiscreteDomain<PointSampling<Dim>>>`
- Tagging indexes removes the need for knowing the ordering. This can help with generic loops.
- E.g. a quadrature over $v_{/\!/}$:

```
void integrate_in_v(ddc::ChunkSpan<double, ddc::DiscreteDomain<IDimR, IDimP, IDimT, IDimMu>> result,
                    ddc::ChunkSpan<const double, ddc::DiscreteDomain<IDimR, IDimP, IDimT, IDimMu, IDimV>> fdistrib) {
    ddc::for_each(result.domain(), [&](auto i) {
        result = integrate(fdistrib[i]);
    });
}
```

DDC : Type Casting Discrete Computations

Advantages and Difficulties

Advantages

- Compilation errors for mathematical errors
- Loops over elements of multiple dimensions are reduced to one loop which is easier to distribute
- Index order is unimportant which makes it easier to reorder between MPI calls

Difficulties

- Maturity of DDC
  - Points
- Balancing reusability and readability
  - Most physicists are not familiar with templated code
  - Avoiding templates leads to code that can only be used in one dimension (e.g. a quadrature in r only)

Conclusions

The Gysela Fortran code was ported using OpenMP

- 2 levels of OpenMP parallelism were used
- Blocking allows both CPU and GPU parallelism to co-exist
- GPU-aware MPI works well but is not always the best option for smaller arrays

The GyselaX C++ code will be ported using Kokkos

- Kokkos hides the choice of acceleration from non-experts
- Code must be written with the correct paradigms
- Tools such as DDC can help parallelism to be written in mathematical terms