GPU porting of GBS with CUDA

Nicola Varini
nicola.varini@epfl.ch

Overview

- GBS computational patterns
  - RHS(plasma) - CUDA implementation
  - Solver - PETSc implementation
- Performance analysis - Leonardo vs LUMI-C
- Conclusion and future work

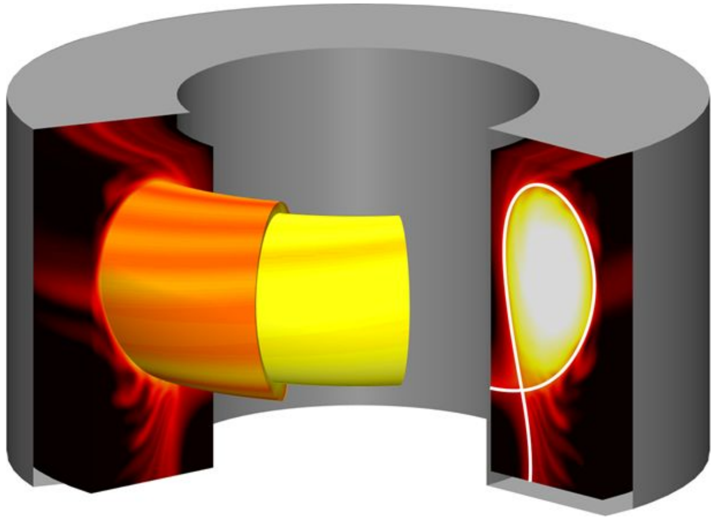
GBS - Global Braginskii Solver

GBS is used to study **plasma turbulence** in the tokamak boundary

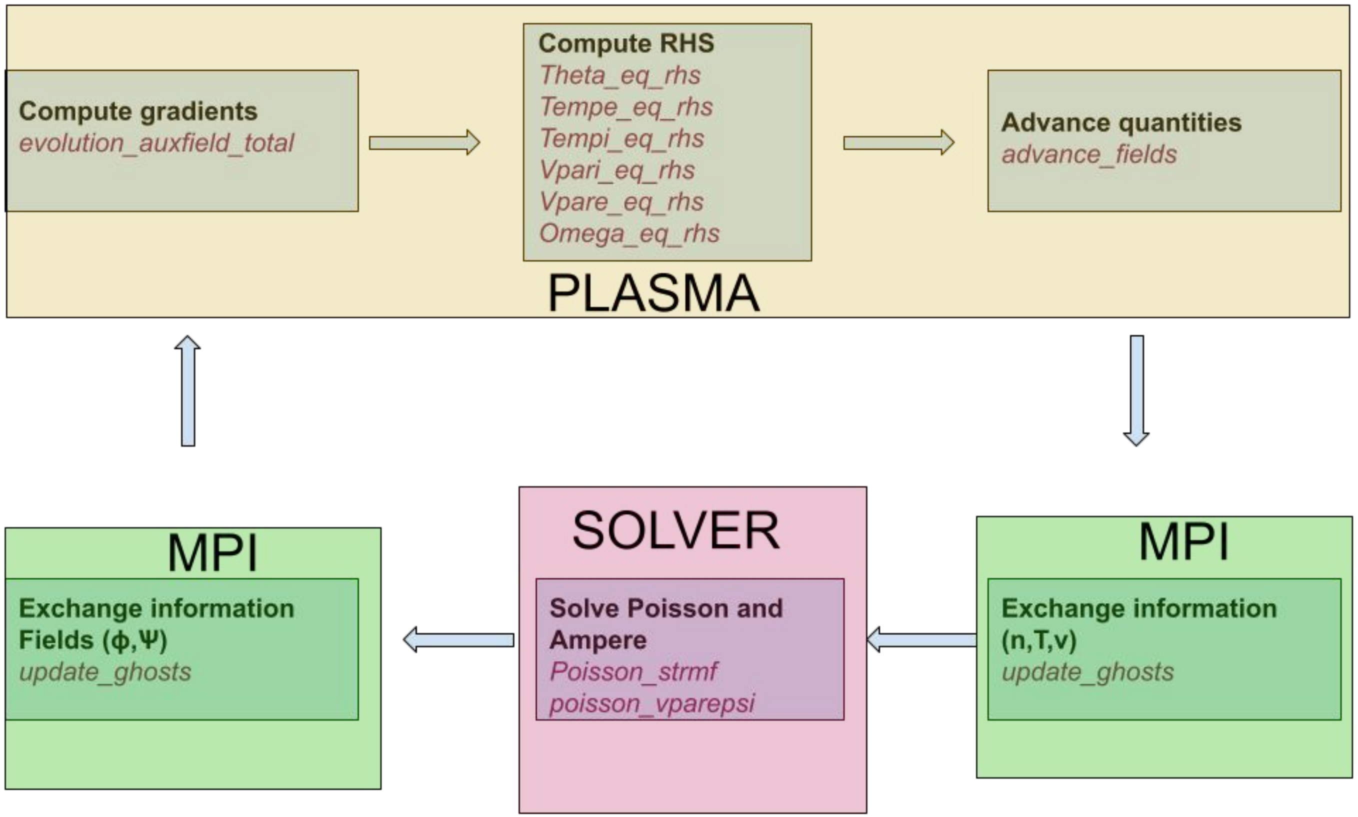
- Plasma model based on drift-reduced Braginskii equations
- Single species kinetic neutral model
- Time evolution: 4th order Runge-Kutta algorithm
- Spatial discretisation: 4th order finite centered differences

HPC in GBS:

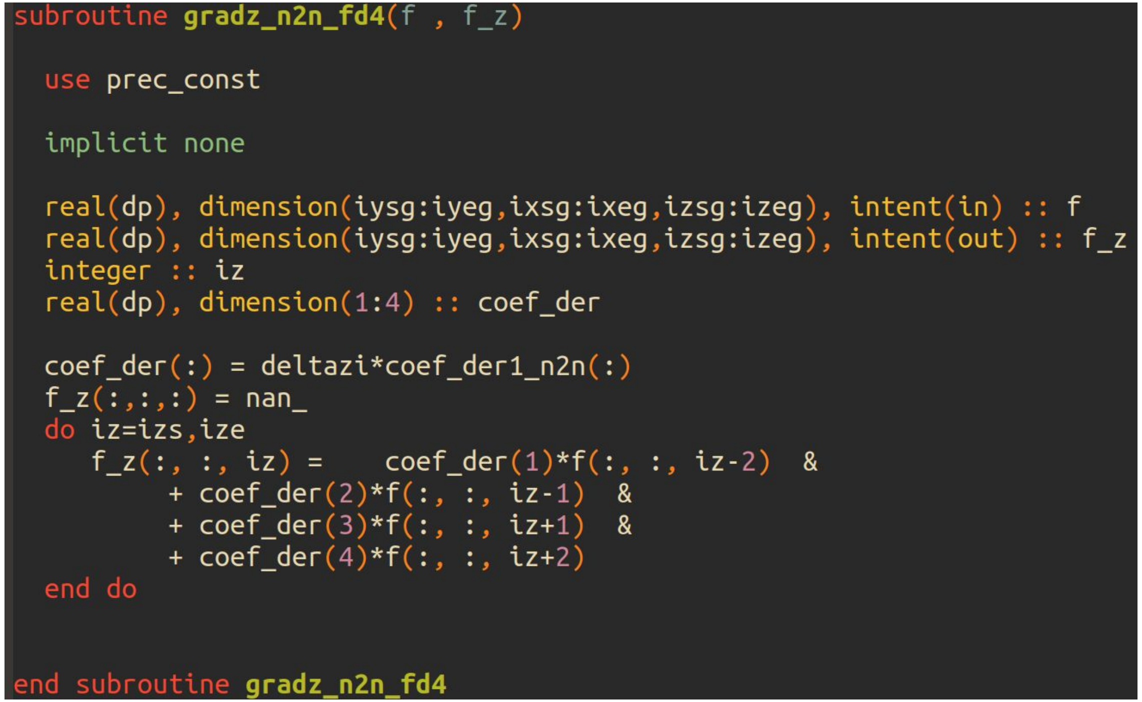
- It can run efficiently on Tier-0 systems.
- Written in **Fortran90 + MPI, CUDA for NVIDIA GPU**
- Dependencies: MPI, HDF5, PETSc, CUDA
- Main bottlenecks:
    - RHS computation(**stencil operations**)
    - Poisson and Ampere solvers - **PETSc**



GBS TIMESTEP



GBS computational pattern - RHS



```fortran
subroutine gradz_n2n_fd4(f , f_z)

  use prec_const

  implicit none

  real(dp), dimension(iysg:iyeg,ixsg:ixeg,izsg:izeg), intent(in) :: f
  real(dp), dimension(iysg:iyeg,ixsg:ixeg,izsg:izeg), intent(out) :: f_z
  integer :: iz
  real(dp), dimension(1:4) :: coef_der

  coef_der(:) = deltazi*coef_der1_n2n(:)
  f_z(:,:,:) = nan_
  do iz=izs,ize
     f_z(:, :, iz) =    coef_der(1)*f(:, :, iz-2)  &
          + coef_der(2)*f(:, :, iz-1)  &
          + coef_der(3)*f(:, :, iz+1)  &
          + coef_der(4)*f(:, :, iz+2)
  end do


end subroutine gradz_n2n_fd4
```

Derivative of a field along z

- The RHS is composed of a series of **stencil** operations.
- Idea: CUDA C implementation.
- These routines are not modified by the developers.

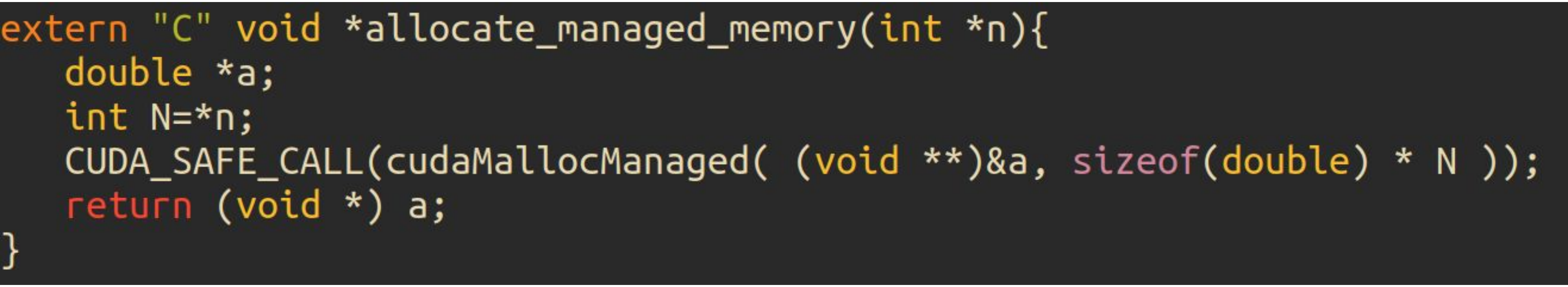
CUDA C++

- Ingredients:
  - Memory management, ideally accessible from Fortran and C++
  - C++ CUDA kernels
  - Call C++ CUDA kernels from fortran

- Pros:
  - Native CUDA compiler (reliable)
  - “Easy” to debug
  - “Easy” to tune
  - Portability to NVIDIA architectures.

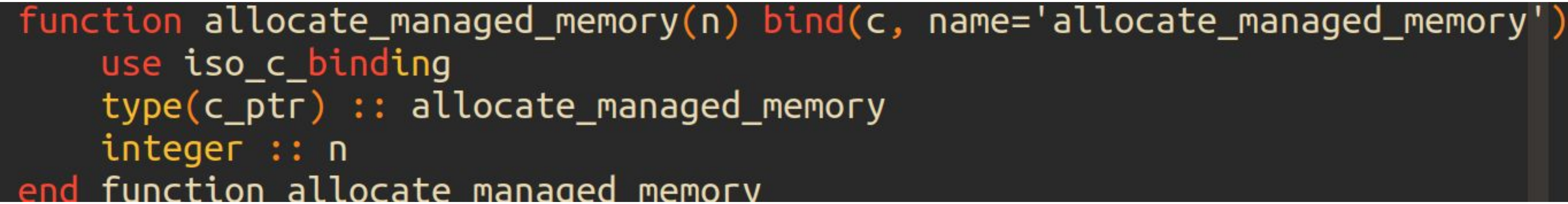
- Cons:
  - Requires code duplication.
  - Fortran/C++ interface
    - row vs column
    - Single vs multidimensional array

CUDA Managed memory



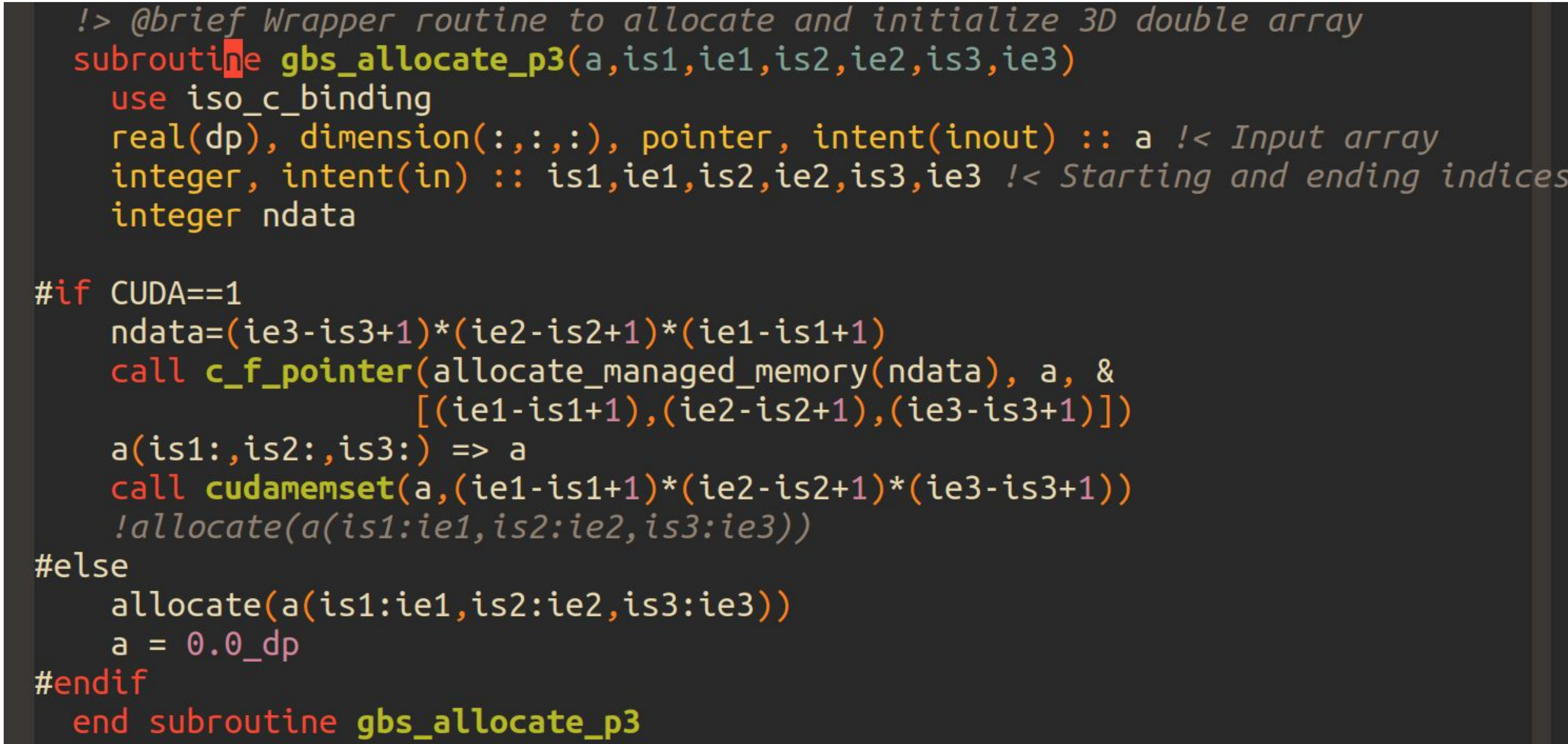
```c
extern "C" void *allocate_managed_memory(int *n){
   double *a;
   int N=*n;
   CUDA_SAFE_CALL(cudaMallocManaged( (void **)&a, sizeof(double) * N ));
   return (void *) a;
}
```

The arrays are accessible from both CPU and GPU



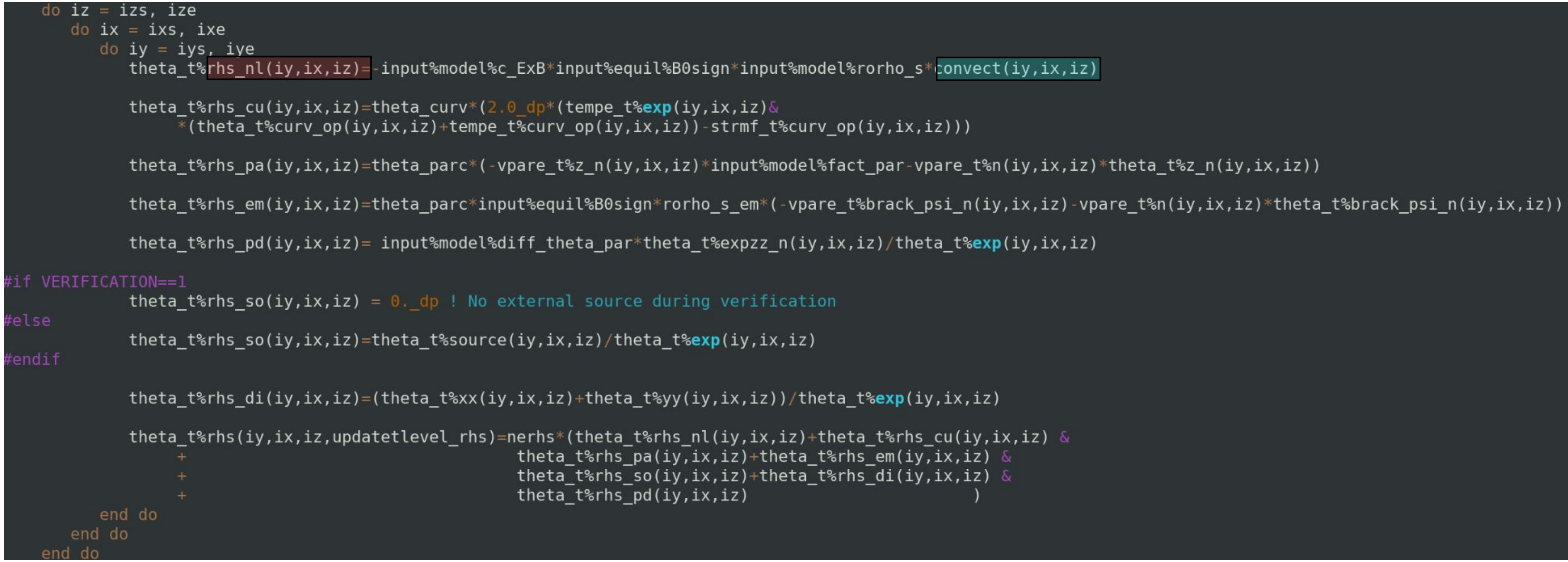
```fortran
 function allocate_managed_memory(n) bind(c, name='allocate_managed_memory')
     use iso_c_binding
     type(c_ptr) :: allocate_managed_memory
     integer :: n
 end function allocate_managed_memory
```

Managed memory - API interoperability



```
  !> @brief Wrapper routine to allocate and initialize 3D double array
  subroutine gbs_allocate_p3(a,is1,ie1,is2,ie2,is3,ie3)
    use iso_c_binding
    real(dp), dimension(:,:,:), pointer, intent(inout) :: a !< Input array
    integer, intent(in) :: is1,ie1,is2,ie2,is3,ie3 !< Starting and ending indices
    integer ndata

#if CUDA==1
    ndata=(ie3-is3+1)*(ie2-is2+1)*(ie1-is1+1)
    call c_f_pointer(allocate_managed_memory(ndata), a, &
                    [(ie1-is1+1),(ie2-is2+1),(ie3-is3+1)])
    a(is1:,is2:,is3:) => a
    call cudamemset(a,(ie1-is1+1)*(ie2-is2+1)*(ie3-is3+1))
    !allocate(a(is1:ie1,is2:ie2,is3:ie3))
#else
    allocate(a(is1:ie1,is2:ie2,is3:ie3))
    a = 0.0_dp
#endif
  end subroutine gbs_allocate_p3
```

Example - Fortran implementation



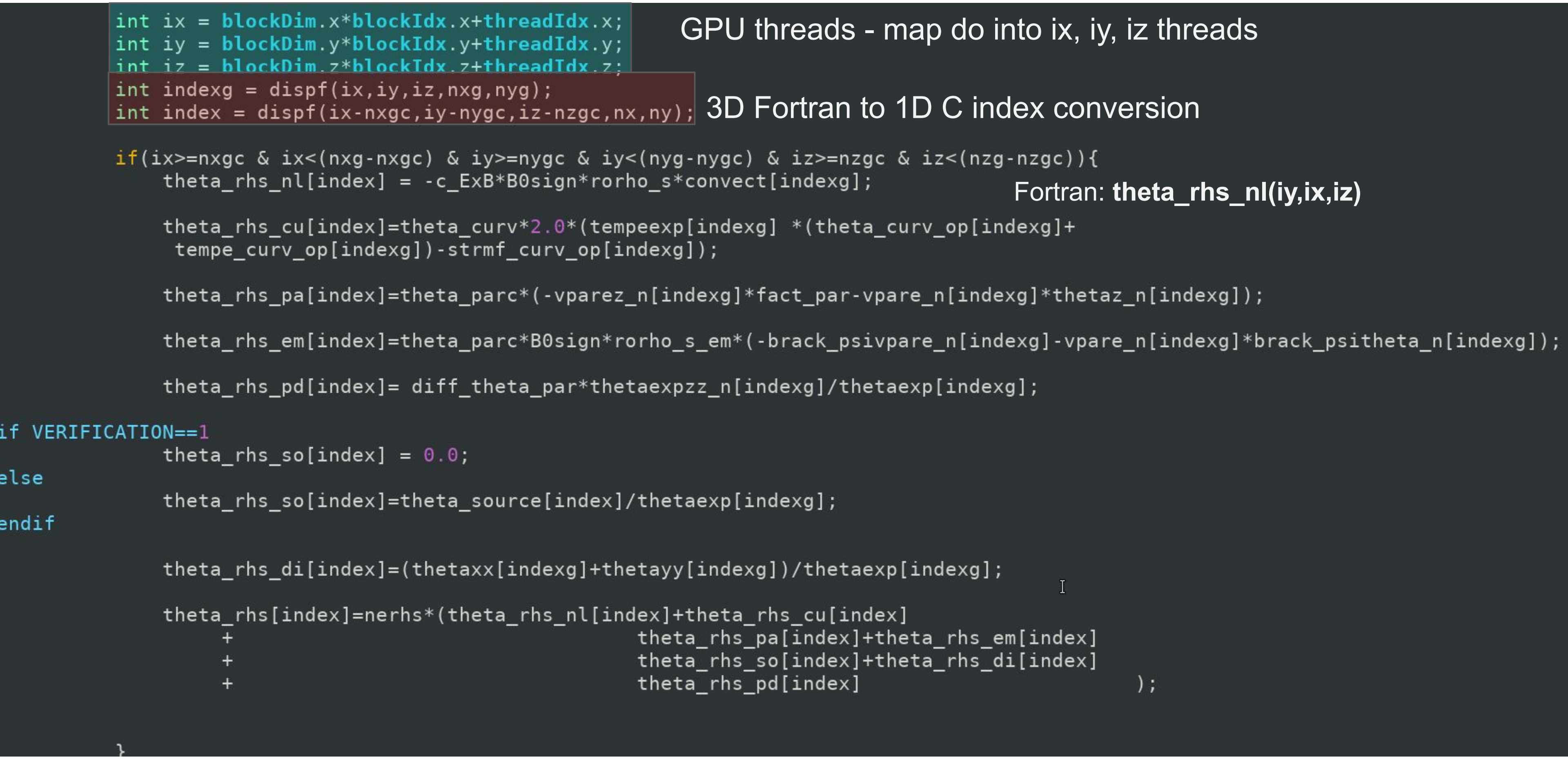
```fortran
    do iz = izs, ize
       do ix = ixs, ixe
          do iy = iys, iye
             theta_t%rhs_nl(iy,ix,iz)=-input%model%c_ExB*input%equil%B0sign*input%model%rorho_s*convect(iy,ix,iz)

             theta_t%rhs_cu(iy,ix,iz)=theta_curv*(2.0_dp*(tempe_t%exp(iy,ix,iz)&
                  *(theta_t%curv_op(iy,ix,iz)+tempe_t%curv_op(iy,ix,iz))-strmf_t%curv_op(iy,ix,iz)))

             theta_t%rhs_pa(iy,ix,iz)=theta_parc*(-vpare_t%z_n(iy,ix,iz)*input%model%fact_par-vpare_t%n(iy,ix,iz)*theta_t%z_n(iy,ix,iz))

             theta_t%rhs_em(iy,ix,iz)=theta_parc*input%equil%B0sign*rorho_s_em*(-vpare_t%brack_psi_n(iy,ix,iz)-vpare_t%n(iy,ix,iz)*theta_t%brack_psi_n(iy,ix,iz))

             theta_t%rhs_pd(iy,ix,iz)= input%model%diff_theta_par*theta_t%expzz_n(iy,ix,iz)/theta_t%exp(iy,ix,iz)

#if VERIFICATION==1
             theta_t%rhs_so(iy,ix,iz) = 0._dp ! No external source during verification
#else
             theta_t%rhs_so(iy,ix,iz)=theta_t%source(iy,ix,iz)/theta_t%exp(iy,ix,iz)
#endif

             theta_t%rhs_di(iy,ix,iz)=(theta_t%xx(iy,ix,iz)+theta_t%yy(iy,ix,iz))/theta_t%exp(iy,ix,iz)

             theta_t%rhs(iy,ix,iz,updatetlevel_rhs)=nerhs*(theta_t%rhs_nl(iy,ix,iz)+theta_t%rhs_cu(iy,ix,iz) &
                  +                                  theta_t%rhs_pa(iy,ix,iz)+theta_t%rhs_em(iy,ix,iz) &
                  +                                  theta_t%rhs_so(iy,ix,iz)+theta_t%rhs_di(iy,ix,iz) &
                  +                                  theta_t%rhs_pd(iy,ix,iz)                       )
          end do
       end do
    end do
```

MEMORY ALLOCATION

convect(iysg:iyeg,ixsg:ixeg,izsg:izeg) - **with** ghost cells

theta%rhs_nl(iys:iye,ixs:ixe,izs:ize) - **without** ghost cells

C++ CUDA implementation



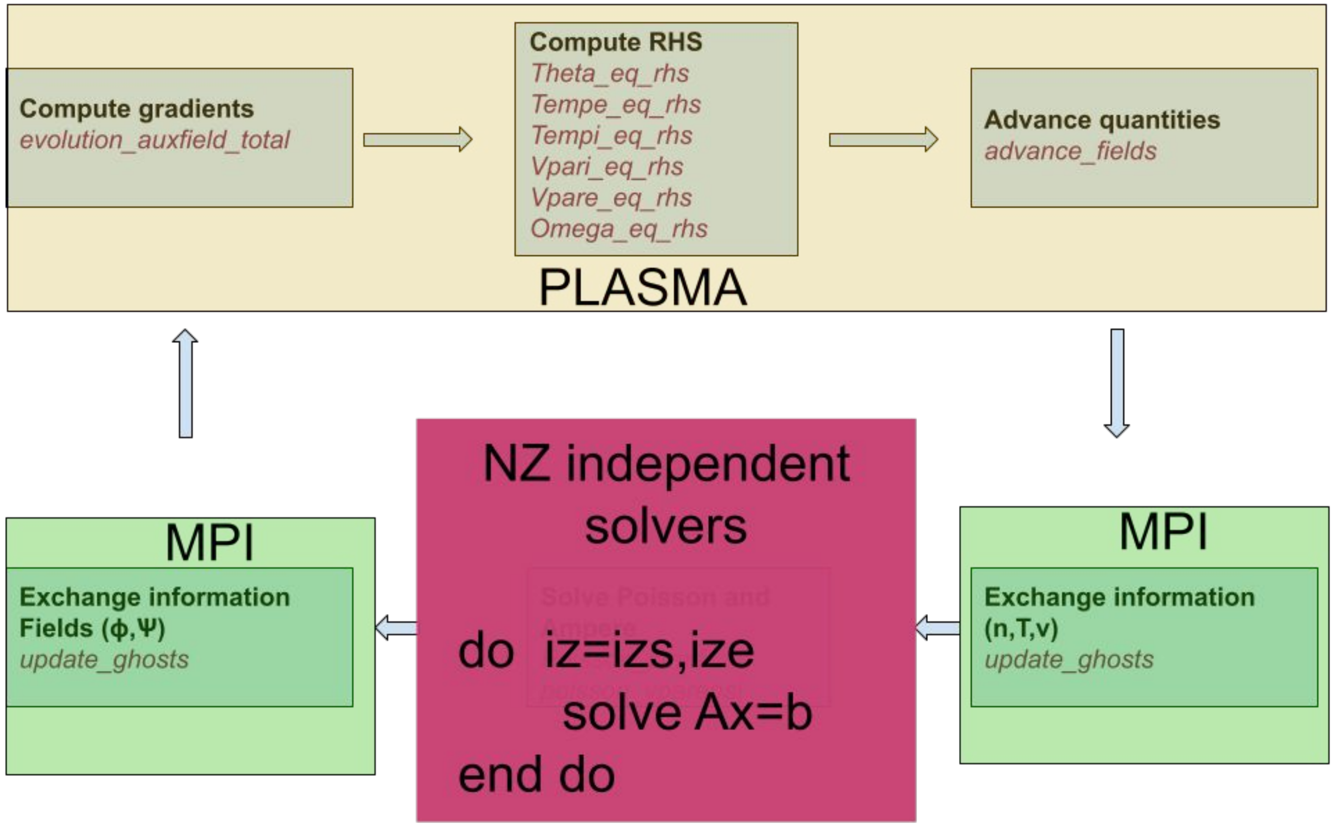
GPU threads - map do into ix, iy, iz threads

3D Fortran to 1D C index conversion

Fortran: **theta_rhs_nl(iy,ix,iz)**

```
        int ix = blockDim.x*blockIdx.x+threadIdx.x;
        int iy = blockDim.y*blockIdx.y+threadIdx.y;
        int iz = blockDim.z*blockIdx.z+threadIdx.z;
        int indexg = dispf(ix,iy,iz,nxg,nyg);
        int index = dispf(ix-nxgc,iy-nygc,iz-nzgc,nx,ny);

        if(ix>=nxgc & ix<(nxg-nxgc) & iy>=nygc & iy<(nyg-nygc) & iz>=nzgc & iz<(nzg-nzgc)){
            theta_rhs_nl[index] = -c_ExB*B0sign*rorho_s*convect[indexg];

            theta_rhs_cu[index]=theta_curv*2.0*(tempeexp[indexg] *(theta_curv_op[indexg]+
             tempe_curv_op[indexg])-strmf_curv_op[indexg]);

            theta_rhs_pa[index]=theta_parc*(-vparez_n[indexg]*fact_par-vpare_n[indexg]*thetaz_n[indexg]);

            theta_rhs_em[index]=theta_parc*B0sign*rorho_s_em*(-brack_psivpare_n[indexg]-vpare_n[indexg]*brack_psitheta_n[indexg]);

            theta_rhs_pd[index]= diff_theta_par*thetaexpzz_n[indexg]/thetaexp[indexg];

if VERIFICATION==1
            theta_rhs_so[index] = 0.0;
else
            theta_rhs_so[index]=theta_source[index]/thetaexp[indexg];
endif

            theta_rhs_di[index]=(thetaxx[indexg]+thetayy[indexg])/thetaexp[indexg];

            theta_rhs[index]=nerhs*(theta_rhs_nl[index]+theta_rhs_cu[index]
                 +                                 theta_rhs_pa[index]+theta_rhs_em[index]
                 +                                 theta_rhs_so[index]+theta_rhs_di[index]
                 +                                 theta_rhs_pd[index]                    );


        }
```

The solver



```
do  iz=izs,ize
      solve Ax=b
end do
```

The solver is performed in the **poloidal** xy plane

Many **independent** linear systems

This task is performed through **PETSc** (CSR API)

Performance comparison : Leonardo vs LUMI-C

System	Cores/node	Mem/node	GPU/node	#nodes
Leonardo(GPU)	32 Intel Icy Lake	512(DDR4)	4 A100	3456
LUMI-C	128 AMD Epyc	256(DDR4)	-	2048

GBS on Leonardo

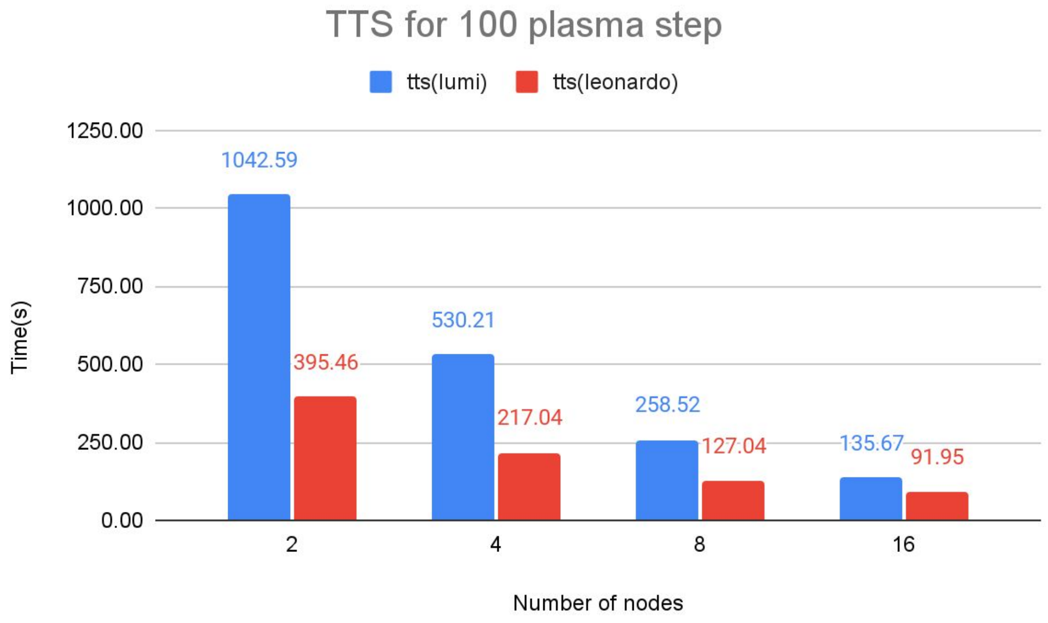
- Software stack: *gcc-11.3.0 cuda-11.8.0 hdf-1.12.2*
- RHS -> nvcc compiler
- Solver: PETSc
- Configuration:
  - Gcc stack with openmpi and cuda
  - *./configure --prefix=../petsc-install* ***--with-cuda****=1* ***--download-hypre*** *--download-hypre-configure-arguments="--enable-unified-memory" --with-fc=mpif90 --with-cc=mpicc --with-cxx=mpicxx --with-cuda-arch=80* ***--download-amgx***
  - In this way we can use HYPRE and AMGX as algebraic preconditioners in PETSc

petscrc

```
-ksp_type dgmres
-pc_type hypre

-mat_type mpiaijcusparse
-vec_type cuda
```

- PETSc read a configuration file
- Easy to change parameters

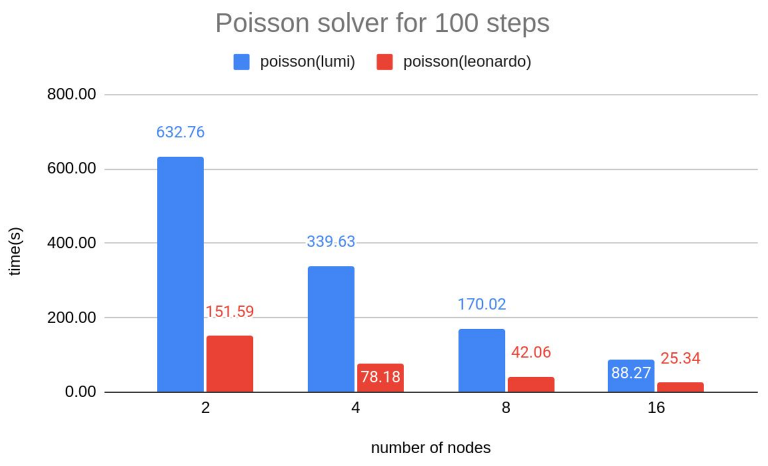
Leonardo vs LUMI - TCV@0.9T

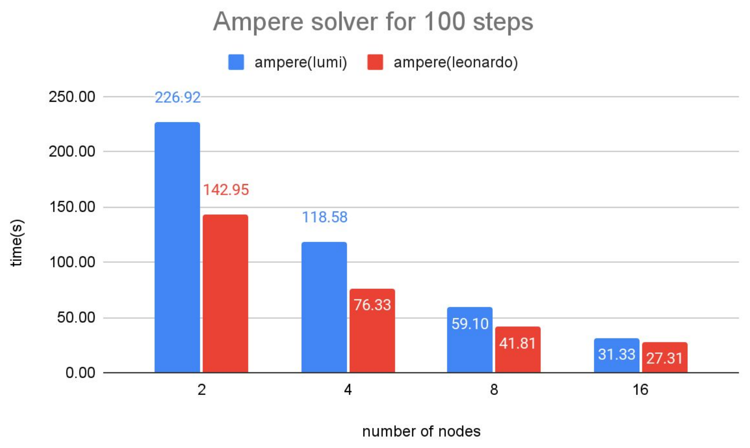


- Grid size:
  - Nx=300, Ny=600, Nz=128
- 100 plasma steps
- No neutrals
- Scaling lumi leonardo

nodes	Px		Py		Pz
2	8	1	16	1	2
4	8	1	16	1	4
8	8	1	16	1	8
16	8	1	16	1	16

Solver

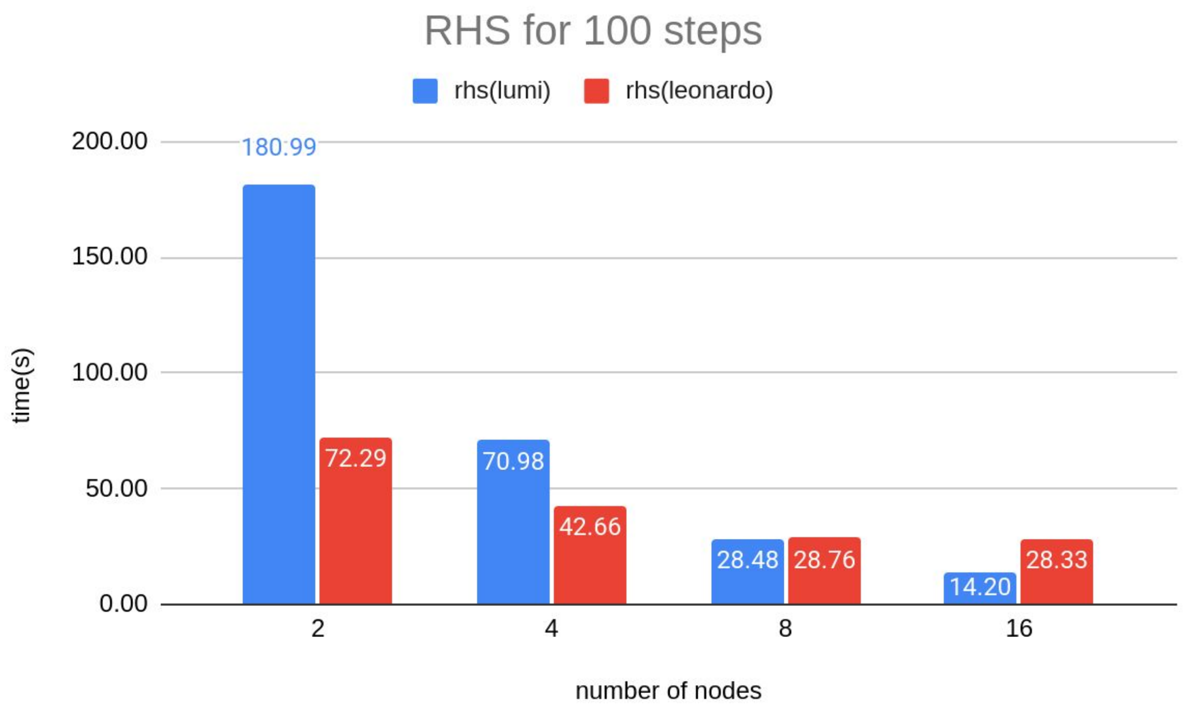




- Solver: dgmres
- Preconditioner: HYPRE/BoomerAMG
  - V cycle
  - Aggressive coarsening

```
-poisson_ksp_type dgmres
-poisson_ksp_rtol 1e-7
-poisson_ksp_reuse_preconditioner yes
-poisson_ksp_initial_guess_nonzero yes
-poisson_pc_type hypre
-poisson_pc_hypre_type boomeramg
-poisson_pc_hypre_boomeramg_strong_threshold 0.25
-poisson_pc_hypre_boomeramg_max_levels 30
-poisson_pc_hypre_boomeramg_agg_nl 1
-poisson_pc_hypre_boomeramg_agg_num_paths 1
-poisson_pc_hypre_boomeramg_truncfactor 0.2
-poisson_pc_hypre_boomeramg_interp_type ext+i
-mat_type mpiaijcusparse
-vec_type cuda
```

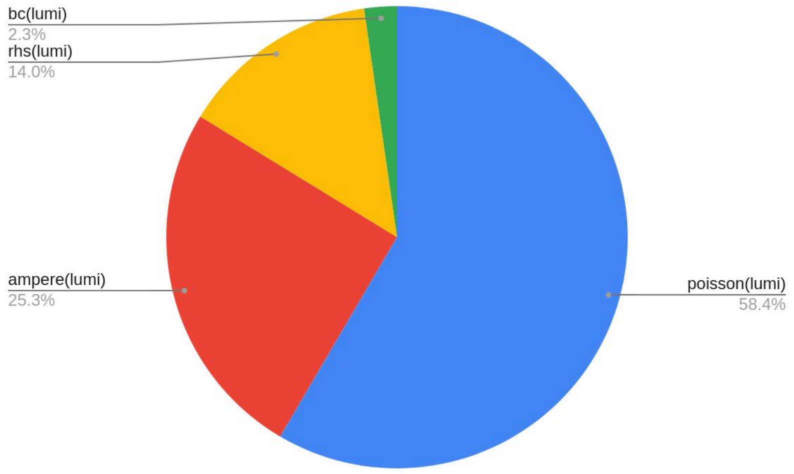
RHS

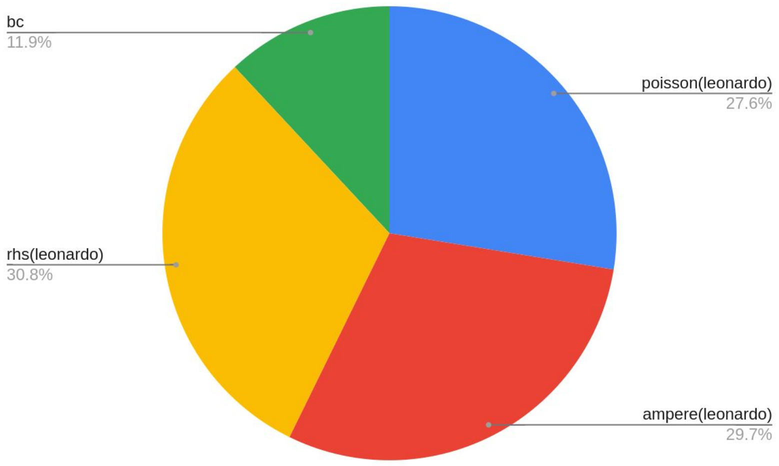


- Good GPU performance at small number of nodes
-

Timers at 16 nodes

- Poisson + Ampere is the major bottleneck.
  - 83% on LUMI
  - 58% on Leonardo
- The boundary conditions are not ported on GPU, ~ 10% of the tts.





Conclusion and future work

- Porting the RHS to GPU with CUDA show good performance with a basic implementation.
    - No specific GPU optimization
- CUDA is reliable but cumbersome to maintain:
    - Need code duplication and Fortran/C++ interface
    - In 2024 we plan to implement the RHS with **OpenMP/OpenACC**
- Solver improvement:
    - Algebraic multigrid preconditioners work well on TCV.
    - **Guess improvement** to reduce the number of iterations.