January 30, 2024
ABINIT 2024 school
Toward ab initio simulations on exascale computers

ABINIT on Graphics Processing Units

Marc Torrent, Pierre Kestener

CEA, DAM, DIF. F-91297 Arpajon, France

Matter under Extreme Conditions Laboratory (LMCE)
High Performance Computing Laboratory for Calculation and Simulation (LIHPC)
Paris-Saclay university, France

ABINIT on GPU – Outline

- Porting strategy
- Programming model
- Performances
- How to use

First ABINIT+GPU version – 2013 – cuda

Programming model

- MPI + **cuda** kernels + **cublas** + **cuFFT** + **magma**
- Replace MPI FFT level by batched cuFFT calls
- Write specific *cuda* kernels for non-local H
- Use *cublas* for linear algebra in iterative solver
- Use *magma* v1.x for *Rayleigh-Ritz* procedure

Some drawbacks

- Use of block CG: lot of **host-device transfers**
- GPU-*Rayleigh-Ritz* not really profitable
- Cuda code to be **regularly updated**
- **Nvidia only**
- **Limited** in system size

1 CPU ⟷ 1 GPU

1 CPU [+ 1 GPU]

Time proc 0 (s)	Titane			Curie		
	CPU	**CPU+GPU**	**Speedup**	**CPU**	**CPU+GPU**	**Speedup**
107 gold atoms	4230	1857	2,3	5154	621	8.3
31 copper atoms	153	102	1,5	235	64	3,7
5 BaTiO3 atoms	85	98	0,9	86	73	1,2

Strongly dependent on architecture

8 CPU [+ 8 GPU]

Time proc 0 (s)	Curie		
	CPU	**CPU+GPU**	**Speedup**
215 silicon atoms (1 iter.)	16160	3680	4,4
107 gold atoms	866	621	4,7
31 copper atoms	49	64	1,5

200 CPU [+ 200 GPU]

Time proc 0 (s)	Curie		
	CPU	**CPU+GPU**	**Speedup**
215 silicon atoms (45 iter.)	25856	6309	4,1

ABINIT on GPU 2022

Porting strategy

ABINIT+GPU : motivations for a modern version

- Legacy version = 2013 – Still running but obsolete – Tests no more relevant
- Constraints for a new version:
  - Has to run on the 25 bots of the ABINIT testfarm:
    gfortran, nag fc, xlf, flang, llvm, ifort, ifx, nvfortran, …
  - NOMAD CoE objectives and deliverables
  - Institutions platforms or computing time grants
- « Standard » physicists should be able to continue their developments
  = be as invisible as possible
- Abinit used on (almost) all computing centers
  Has to run on as many plateforms as possible
  gcc, *nvcc*, *aocc*, *cray*, *intel oneapi*, …
- Be as persistent as possible, minimize the software maintenance

ABINIT+GPU implementation – Strategy

- Compare programming models
- Select which parts of the code should be ported (GS first)
- Profile the code
- Choose algorithm
- Rewrite some parts of the algorithm
- Rewrite **H** application (”`xg`” type)
- Create « low level » wrappers
  - Be invisible
  - Be « hardware- agnostic »
  - Use several versions of library (`cuda`, `HIP`, `ELPA`, …)
- Program

ABINIT+GPU implementation – Strategy

- Use more « **batch processing** » on the GPU
  Increase FLOPS
- **Minimize** host-devices **transfers**
  Coarse grained programming model?
- **Less** rely on **Rayleigh-Ritz** calls (limiting factor)
  Choose relevant algorithm
- Don’t be be directly dependent on cuda code/kernels
- Run on non-Nvidia GPU
- Compare « [CPU] node to [CPU+GPU] node »

➠ *Run larger systems on MPI+GPU than on MPI+openMP*

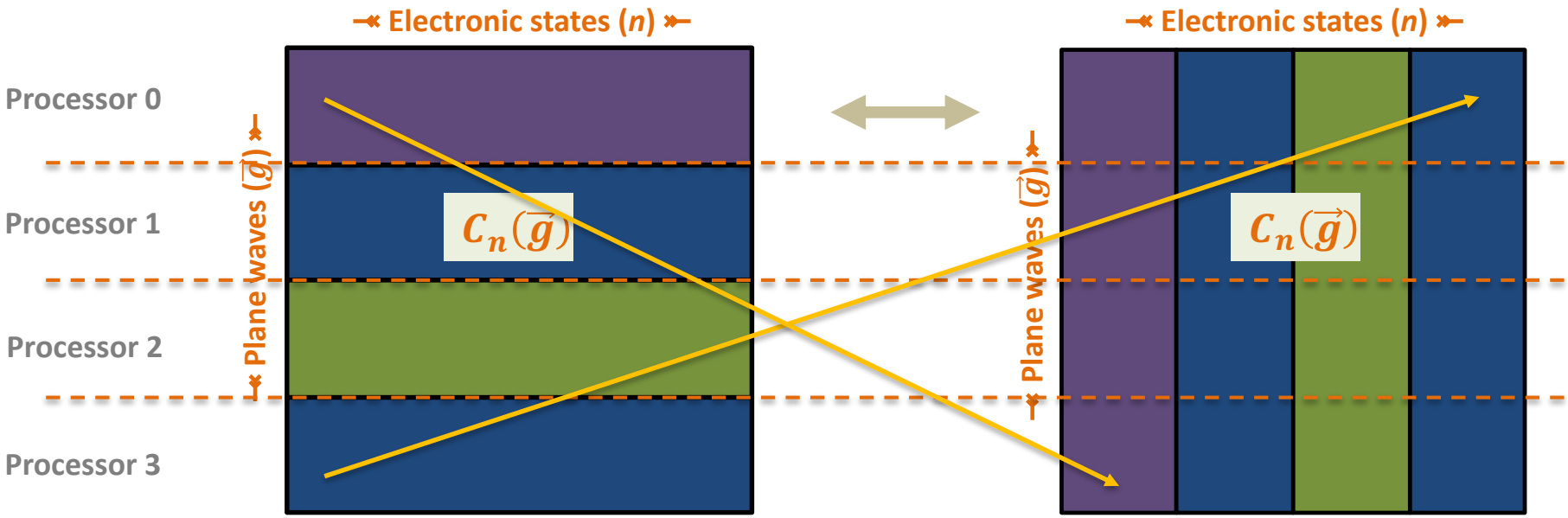
Plane-wave DFT on GPU

- The **application of the Hamiltonian** is an operation well-suited for GPUs. It can efficiently take advantage from "**batch processing**" mode.
- The application of the Hamiltonian is performed on **continuously refreshed data** (wave functions). These data should stay on the GPU.
- The ***Rayleigh-Ritz* procedure** requires **communication** between GPUs. Diagonalization libraries are not efficient for the considered system sizes. We have to minimize the calls to the *Rayleigh-Ritz* procedure (use Chebyshev filtering algorithm?).

Internal representation of wave-functions

$$\psi_n(\vec{r}) = \sum_{\vec{g}} C_n(\vec{g})\, e^{i\vec{g}\cdot\vec{r}}$$

All processors communicate with all processors!
Do it as few times as possible!



Linear algebra

➠ **Need all some plane-waves ($\vec{g}$)for all ψ**

Hamiltonian application

➠ **Need all plane-waves ($\vec{g}$) for some ψ**

Application of H in plane-wave basis

Perfectly adapted to GPU

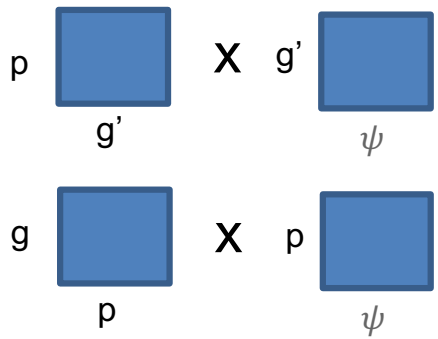
$$H = \frac{1}{2}\Delta + V_{loc} + P\, D_V[\rho]\, P^T$$

$$\langle \vec{g} | \tilde{v}_{eff} | \psi \rangle = \underbrace{\int d\vec{r} \cdot e^{-i\cdot\vec{g}\cdot\vec{r}} \cdot \left(\tilde{v}_{eff}(\vec{r}) \cdot \underbrace{\left[\sum_{\vec{g}'} \left(c_{\vec{g}'} e^{i\cdot\vec{g}\cdot\vec{r}} \right) \right]}_{\text{FFT}} \right)}_{\text{FTT}}$$

⇢ FFT + Batch processing

$$\langle \tilde{p}_i | \psi \rangle = \sum_{\vec{g}'} \left\langle \tilde{p}_i \middle| \vec{g}' \right\rangle \left\langle \vec{g}' \middle| \psi \right\rangle$$

$$\langle \vec{g} | V_{NL} | \psi \rangle = \sum_{\tilde{p}_i} \langle \vec{g} | \tilde{p}_i \rangle \langle \textstyle\sum_j D_V \tilde{p}_j \, | \psi \rangle$$

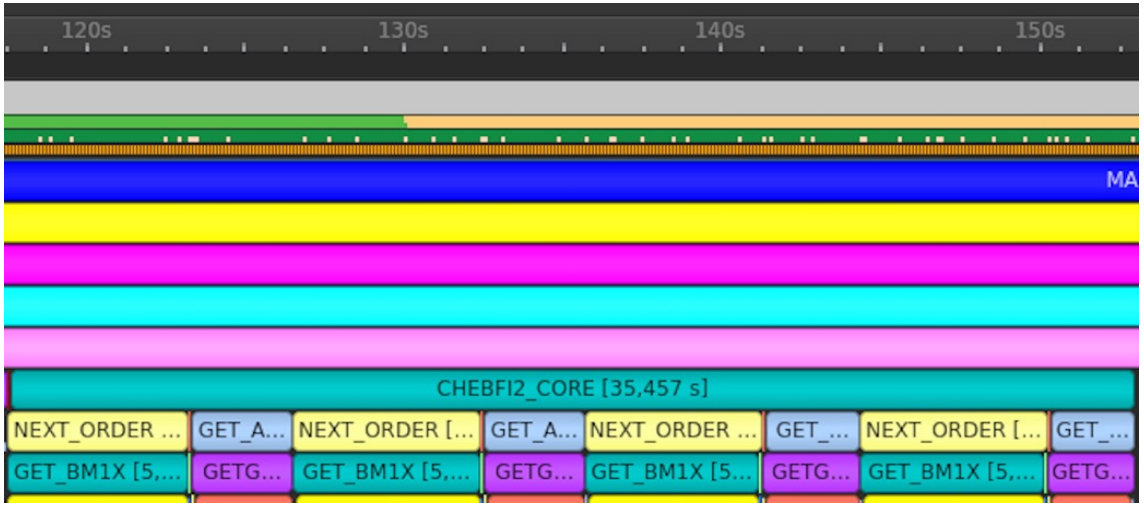


⇢ GEMM = Batch processing

Application of H on GPU

Titanium 25 atoms
4096 bands

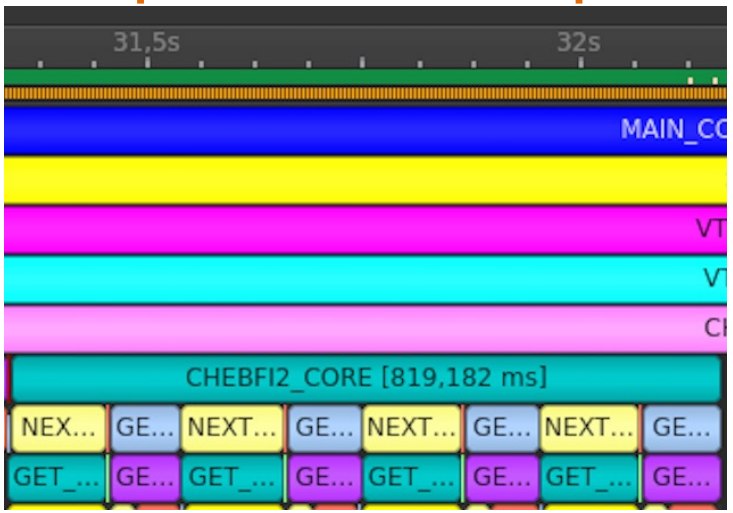
10”



*512 **CPU** cores*
AMD Milan EPYC 7543

4096 x
H applications
3,1”

0,5”



*4 **GPU** cards*
Nvidia A100 80Go

4096 x
H applications
0,09”

LOBPCG vs Chebyshev filtering

Locally Optimal Block Preconditioned CG algo

Eigenvectors are grouped in blocks. Each block is updated with a parallel procedure.

For n=1 to N_{blocks} ➡ This loop is done sequentially

Let ψ^0 be a block vector, T a preconditioner and $P^0 = 0$

For j=1 to N_{iter}

MPI transposition of eigenvectors (one per block)

Transfer of eigenvectors ψ from CPU to GPU

- Compute Rayleigh quotients $\rho^j(\psi^j)$
- Compute preconditioned residuals $W^j = T(H - \rho^j S\psi^j)$

Section executed on independent GPU

Transfer of eigenvectors ψ from CPU to GPU

MPI transposition of eigenvectors (one per block)

- Apply Rayleigh-Ritz method on $\{P^j_1 \dots P^j_m, \psi^j_1 \dots \psi^j_m, W^j_1 \dots W^j_m\}$ to find $\psi^{j+1} = \psi^j \Delta^j + W^j \Lambda^j + P^j \Gamma^j$
- Compute $P^{j+1} = W^j \Lambda^j + P^j \Gamma^j$

Section executed on collaborating GPU

End For

End For

Apply Rayleigh-Ritz on the entire $\{\psi\}$ space which is a union of block subspaces

Chebyshev filtering algo

Filtering phase is independently done on each vector.

MPI transposition of eigenvectors

Transfer of eigenvectors ψ from CPU to GPU

Set ψ^0 and ψ^1

For j=2 to degree_of_P_{cheb}

For i=1 to $N_{vectors}$

- Compute

$$\psi^{j+1} = \frac{2}{r}\left(S^{-1}H\psi^{j-1} - c\psi^{j-1}\right) - \psi^{j-2}$$

End For

End For

Section executed on independent GPU (each GPU handles a subset of ψ)

Transfer of eigenvectors ψ from CPU to GPU

MPI transposition of eigenvectors

Apply Rayleigh-Ritz on the entire $\{P_{cheb}(S^{-1}H)\psi\}$ space has "filtered" eigen-components

Section executed on collaborating GPU

ABINIT on GPU

Programming model

Dual strategy for porting on GPU

Cuda + Kokkos + YAKL

- Nvidia only
- Direct use of cuBLAS and cuFFT
- Systematic use of batch processing
  Use GEMM favored over to GEMV
- Use of *Kokkos* to allow for easy correpondance between CPU (Fortran) and accelerator (C++) code
- Use of *YAKL* to unify memory management

openMP target + HIP

➠ Port entire block/filtering step on GPU

➠ Use external libs for RR

- Tested on Nvidia (A100) and AMD (Mi250)
- Cuda ➠ cuBLAS + cuFFT
  Rocm ➠ rocBLAS + rocFFT
- Systematic use of batch processing mode
  COLLAPSE directive on loops
- Easy porting of non-expensive Fortran code by addition of directives
- Easy management of « persistent » memory on GPU

Kokkos & YAKL performance libraries

Kokkos

https://github.com/kokkos/kokkos

- Allow for automatic use of C++ code on GPU device
- Need to rewrite some Fortran parts in C++

```fortran
!Accumulate gxfac related to non-local operator
!-----------------------------------------------------------------
if (paw_opt==0) then
  !Enl is E(Kleinman-Bylander)
  ABI_CHECK(cplex_enl/=2,"BUG: invalid cplex_enl=2!")
  ABI_CHECK(cplex_fac==cplex,"BU
  do idat = 1,ndat
    do ispinor = 1,nspinor
      do ia = 1,nincat
        do ilmn = 1,nlmn
          ispinor_index = ispino
          idat_ispinor = nspinor
          iln = indlmn(5,ilmn)
          enl_(1) = enl_ptr(iln,
          gxfac_(1:cplex,ilmn,ia
        end do ! ilmn
      end do  ! ia
    end do ! ispinor
  end do ! idat
end if ! paw_opt == 0
```

```kokkos
//Accumulate gxfac related to non-local operator
//-----------------------------------------------------------------
if (paw_opt==0)
{

  // Enl is E(Kleinman-Bylander)
  ABI_CHECK(cplex_enl!=2, "BUG: invalid value; cplex_enl=2!");
  ABI_CHECK(cplex_fac==cplex,"BUG: invalid value; cplex_fac/=cplex!");
  {
    using policy = Kokkos::MDRangePolicy< Kokkos::Rank<4> >;
    Kokkos::parallel_for(
      "accum_gxfac_norm_conserving",
      policy({0,0,0,0},{nlmn,nincat,nspinor,ndat}),
      KOKKOS_LAMBDA(const int ilmn, const int ia, const int ispinor, const int idat)
      {
        int ispinor_index = ispinor + shift_spinor;
        int idat_ispinor = ispinor + idat*nspinor;
        int iln = indlmn(5-1, ilmn, itypat);
        int iproj = shift_proj + ilmn + ia*nlmn;

        double enl1 = enl(iln, itypat, ispinor_index, iphase);

        gxfac (0, iproj, idat_ispinor) = enl1 * gx(0, iproj, idat_ispinor);
        if (cplex>1)
          gxfac (1, iproj, idat_ispinor) = enl1 * gx(1, iproj, idat_ispinor);
      });
  }
```

Yet Another Kernel Launcher (YAKL)

https://github.com/mrnorman/YAKL

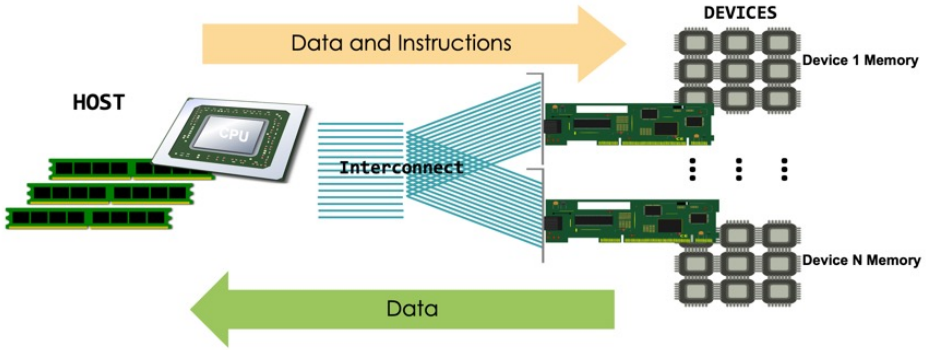
- Allow to allocate Unified Virtual Memory (UVM)
- Visible as a multi-dimensional array in `Fortran`
  Visible as a GPU pointer

```fortran
function gator_allocate_c( bytes ) result(ptr) bind(C, name="gatorAllocate")
  use iso_c_binding
  type(c_ptr)              :: ptr
  integer(c_size_t), value :: bytes
end function gator_allocate_c

interface gator_allocate
  module procedure :: gator_allocate_int4_1d
end interface gator_allocate

subroutine gator_allocate_int4_1d( arr , dims , lbounds_in )
  integer, parameter :: ndims = 1
  integer, pointer , intent(  out) :: arr       (:)
  integer           , intent(in   ) :: dims      (ndims)
  integer, optional, intent(in   ) :: lbounds_in(ndims)
  integer :: lbounds(ndims)
  type(c_ptr) :: data_ptr
  if (present(lbounds_in)) then
    lbounds = lbounds_in
  else
    lbounds = 1
  endif
  data_ptr = gator_allocate_c( int(product(dims)*sizeof(i4),c_size_t) )
  call c_f_pointer( data_ptr , arr , dims )
  arr(lbounds(1):) => arr
end subroutine gator_allocate_int4_1d
```

OpenMP offload

- Directive-based programming
- Extension of OpenMP
- Since version 4.0 but full support in version 5.0
- Supported by several compilers: gcc, nvhpc, aocc, intel, cray, …
- Allow to address several hardware: Nvidia, AMD, intel, …

- OpenMP vs openACC:
  - Same philosophy
  - openACC more selected
  - openACC supported by Nvidia



OpenMP offload in ABINIT

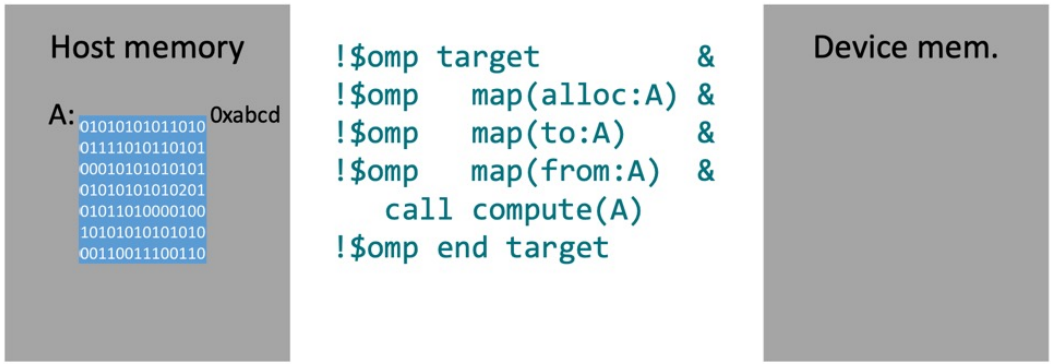
Most used directives

- `!OMP TARGET ENTER DATA`: allocation and association of a CPU array on GPU
- `!OMP TARGET UPDATE TO`: transfer of CPU array to GPU
- `!OMP TARGET UPDATE FROM`: transfer of GPU array to CPU
- `!OMP TARGET EXIT DATA`: deallocation of GPU array et end of association with CPU

OpenMP in ABINIT – Memory allocation

```
ABI_MALLOC(proj_ompgpu,        (cplx,nprojs,nspinor*ndat))
ABI_MALLOC(sm1proj_ompgpu,     (cplx,nprojs,nspinor*ndat))
ABI_MALLOC(PtPsm1proj_ompgpu,  (cplx,nprojs,nspinor*ndat))
!$OMP TARGET ENTER DATA MAP(alloc:proj_ompgpu,sm1proj_ompgpu,PtPsm1proj_ompgpu)
```

- When unified memory is in use these directives are ignored



OpenMP in ABINIT – Loops

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO COLLAPSE(2) &
!$OMP& PRIVATE(idat,ipw,i1,i2,i3) MAP(to:work_gpu,kg_kin,fofgin)
do idat = 1, ndat
  do ipw = 1, npwin
    i1=kg_kin(1,ipw); if(i1<0)i1=i1+n1; i1=i1+1
    i2=kg_kin(2,ipw); if(i2<0)i2=i2+n2; i2=i2+1
    i3=kg_kin(3,ipw); if(i3<0)i3=i3+n3; i3=i3+1
    work_gpu(1, i1, i2, i3+n3*(idat-1)) = fofgin(1, (ipw + npwin*(idat-1)))

    work_gpu(2, i1, i2, i3+n3*(idat-1)) = fofgin(2, (ipw + npwin*(idat-1)))

  end do
end do
```

- Loop collapse
- Workload is managed by the GPU

OpenMP in ABINIT – Call to libraries

```
!$OMP TARGET DATA USE_DEVICE_ADDR(work_gpu) MAP(to:fofr)
fft_state = cuFFTEXECZ2Z(plan_fft, fofr, work_gpu, CUFFT_FORWARD)
!$OMP END TARGET DATA
if(fft_state /= CUFFT_SUCCESS) then
  write(msg,"(a,i3)") "ERROR while fft fofr ==> work_gpu:\n",fft_state
  ABI_ERROR(msg)
end if
```

Call to cuFFT

Kokkos in ABINIT – Call to libraries

FORTRAN code

```fortran
!Accumulate gxfac related to non-local operator
!-----------------------------------------------------------------------
if (paw_opt==0) then
  !Enl is E(Kleinman-Bylander)
  ABI_CHECK(cplex_enl/=2,"BUG: invalid cplex_enl=2!")
  ABI_CHECK(cplex_fac==cplex,"BUG: invalid cplex_fac/=cplex!")
  do idat = 1,ndat
    do ispinor = 1,nspinor
      do ia = 1,nincat
        do ilmn = 1,nlmn
          ispinor_index = ispinor+shift
          idat_ispinor = nspinor*(idat-1) + ispinor
          iln = indlmn(5,ilmn)
          enl_(1) = enl_ptr(iln,itypat,ispinor_index)
          gxfac_(1:cplex,ilmn,ia,idat_ispinor) = enl_(1)*gx(1:cplex,ilmn,ia,idat_ispinor)
        end do ! ilmn
      end do  ! ia
    end do ! ispinor
  end do ! idat
end if ! paw_opt == 0
```

C++ code

```cpp
//Accumulate gxfac related to non-local operator
//-----------------------------------------------------------------------
if (paw_opt==0)
{

  // Enl is E(Kleinman-Bylander)
  ABI_CHECK(cplex_enl!=2, "BUG: invalid value; cplex_enl=2!");
  ABI_CHECK(cplex_fac==cplex,"BUG: invalid value; cplex_fac/=cplex!");
  {
    using policy = Kokkos::MDRangePolicy< Kokkos::Rank<4> >;
    Kokkos::parallel_for(
      "accum_gxfac_norm_conserving",
      policy({0,0,0,0},{nlmn,nincat,nspinor,ndat}),
      KOKKOS_LAMBDA(const int ilmn, const int ia, const int ispinor, const int idat)
      {
        int ispinor_index = ispinor + shift_spinor;
        int idat_ispinor = ispinor + idat*nspinor;
        int iln = indlmn(5-1, ilmn, itypat);
        int iproj = shift_proj + ilmn + ia*nlmn;

        double enl1 = enl(iln, itypat, ispinor_index, iphase);

        gxfac (0, iproj, idat_ispinor) = enl1 * gx(0, iproj, idat_ispinor);
        if (cplex>1)
          gxfac (1, iproj, idat_ispinor) = enl1 * gx(1, iproj, idat_ispinor);
      });
  }
```

Optimized and offloaded code produced by Kokkos

YAKL in ABINIT

Wrappers to YAKL functions

```fortran
function gator_allocate_c( bytes ) result(ptr) bind(C, name="gatorAllocate")
  use iso_c_binding
  type(c_ptr)               :: ptr
  integer(c_size_t), value :: bytes
end function gator_allocate_c

interface gator_allocate
  module procedure :: gator_allocate_int4_1d
end interface gator_allocate

subroutine gator_allocate_int4_1d( arr , dims , lbounds_in )
  integer, parameter :: ndims = 1
  integer, pointer , intent(  out) :: arr       (:)
  integer          , intent(in   ) :: dims      (ndims)
  integer, optional, intent(in   ) :: lbounds_in(ndims)
  integer :: lbounds(ndims)
  type(c_ptr) :: data_ptr
  if (present(lbounds_in)) then
    lbounds = lbounds_in
  else
    lbounds = 1
  endif
  data_ptr = gator_allocate_c( int(product(dims)*sizeof(i4),c_size_t) )
  call c_f_pointer( data_ptr , arr , dims )
  arr(lbounds(1):) => arr
end subroutine gator_allocate_int4_1d
```

Macros for data allocation

```fortran
#define ABI_MALLOC_MANAGED(array,size) call gator_allocate(array,size)
#define ABI_MALLOC_MANAGED_BOUNDS(array,size,lbounds) call gator_allocate(array,s

#define ABI_FREE_MANAGED(array) call gator_deallocate(array)
#define ABI_SFREE_MANAGED(array) if (associated(array)) then NEWLINE ABI_FREE_MAN
```

Use of the macros

```fortran
if (l_use_gpu==ABI_GPU_KOKKOS) then

#if defined HAVE_GPU && defined HAVE_YAKL
  if ( associated(xg%vecR) ) then
    ABI_FREE_MANAGED(xg%vecR)
  end if
  ABI_MALLOC_MANAGED_BOUNDS(xg%vecR,(/rows,cols/), (/1,1/))
  size_bytes = rows*cols*dp
  call gpu_memset(c_loc(xg%vecR), izero, size_bytes)
  xg%trans = 't'
#endif
```

GPU in ABINIT – Safe coding

- Protected with directives
- Differentiate implementations in low-level routines

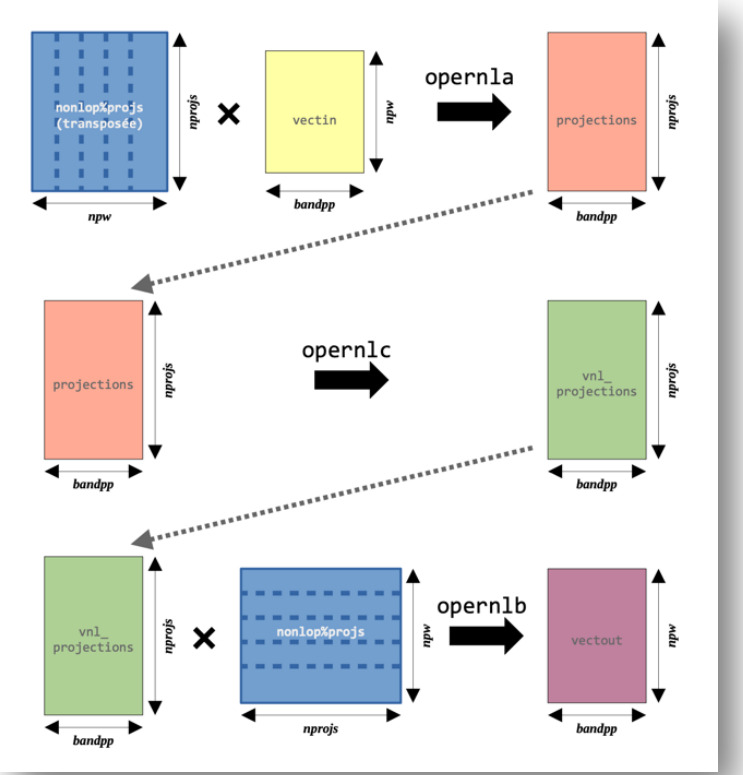
```
#ifdef HAVE_OPENMP_OFFLOAD
 !$OMP TARGET ENTER DATA MAP(to:cg,eig,resid) IF(gs_haml
#endif
```

```
if (l_use_gpu==ABI_GPU_KOKKOS) then

#if defined HAVE_GPU && defined HAVE_YAKL
  if ( associated(xg%vecR) ) then
    ABI_FREE_MANAGED(xg%vecR)
  end if
  ABI_MALLOC_MANAGED_BOUNDS(xg%vecR,(/rows,cols/), (/1,1/))
  size_bytes = rows*cols*dp
  call gpu_memset(c_loc(xg%vecR), izero, size_bytes)
  xg%trans = 't'
#endif

else if (l_use_gpu==ABI_GPU_OPENMP) then

#if defined HAVE_GPU && defined HAVE_OPENMP_OFFLOAD
  if ( allocated(xg%vecR) ) then
    !$OMP TARGET EXIT DATA MAP(release:xg%vecR)
    ABI_FREE(xg%vecR)
  end if
  ABI_MALLOC(xg%vecR,(1:rows,1:cols))
  !$OMP TARGET ENTER DATA MAP(alloc:xg%vecR)
  size_bytes = sizeof(xg%vecR)
  !$OMP TARGET DATA USE_DEVICE_PTR(xg%vecR)
  call gpu_memset(c_loc(xg%vecR), izero, size_bytes)
  !$OMP END TARGET DATA
  xg%trans = 't'
#endif

end if
```

OpenMP in ABINIT – Distribution over GPUs

For large sytems

- Non-local projectors $\langle \vec{g} | \tilde{p}_i \rangle$ fill the GPU memory
- Need them to compute $\langle \tilde{p}_i | \psi \rangle = \sum_{\vec{g'}} \left\langle \tilde{p}_i \middle| \vec{g'} \right\rangle \left\langle \vec{g'} \middle| \psi \right\rangle$
- Distribute them among GPUs
- Need to communicate from GPU to GPU
  By default via the CPU
- ***Nvidia GPU direct*** greatly improves the efficiency
- Do not need to distribute among all GPUs
  Distribution is automatically determined
  Can be forced with `gpu_nl_splitsize`



ABINIT on GPU

Performances

Test cases

- UO2 - 54 atoms, 256 bands

  Could be affected by load unbalancing

- Au - 107 atoms, 640 bands

  Our 2008 test case for: Comp. Mat. Sc. ***42****, 329 (2008)*

- Ti - 255 atoms, 4096 bands

  Our 2015 test case for: Comp. Phys. Comm. ***187****, 98 (2015)*

- Ga2O3, 1960 atoms, 8640 bands

  NOMAD test

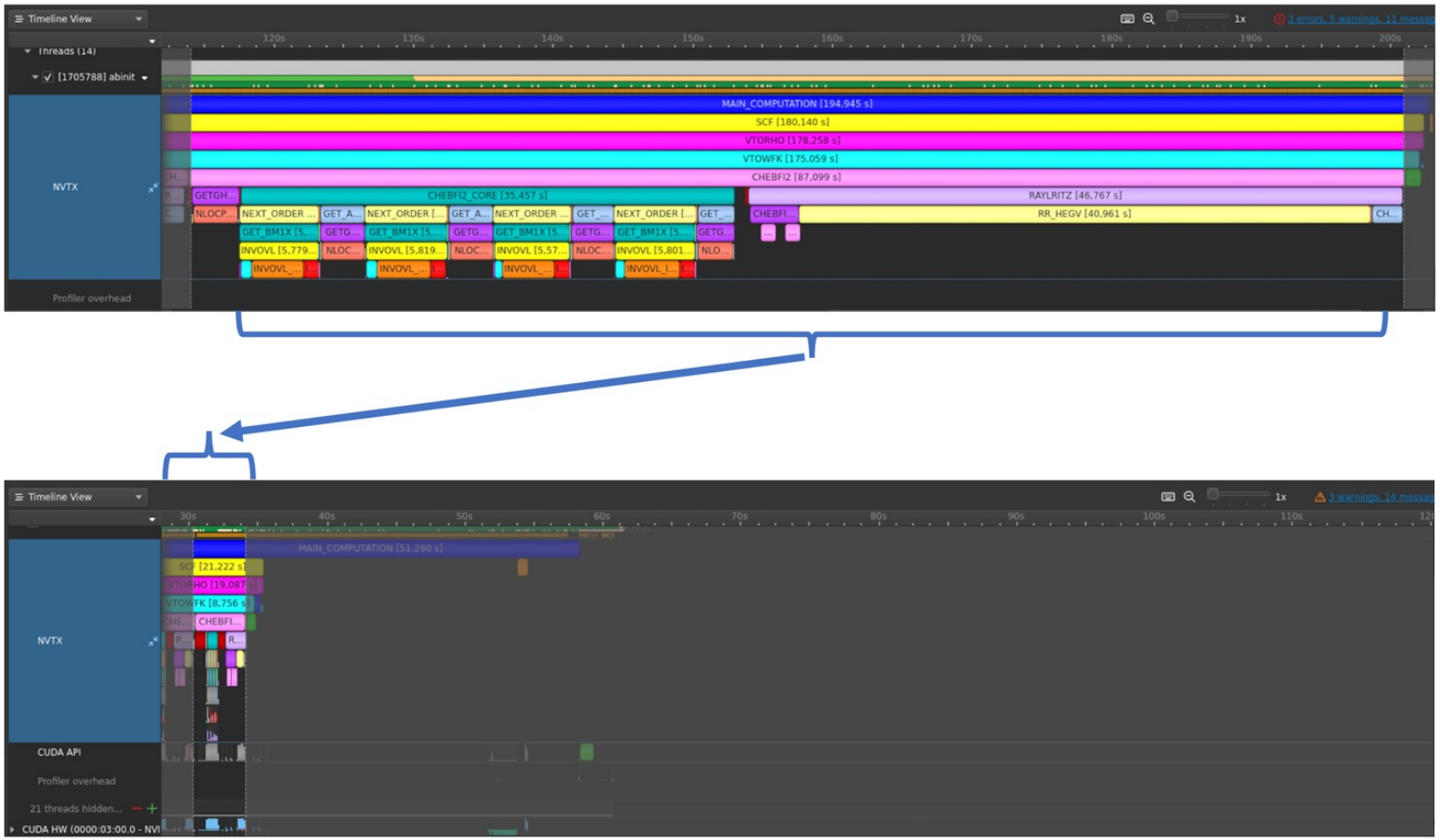
Test platforms

Type de noeud	Adastra MI250X	Topaze a100
CPU	AMD EPYC « Trento » 7A53s 1×64 coeurs	AMD EPYC « Milan » 7763 2×64 coeurs
GPU	AMD Instinct MI250X 8×64 GB	NVIDIA A100-SXM 4×80 GB
Interconnect	Slingshot-11, 200 Gbit/s	Infiniband HDR 200 Gbit/s
Environnement de compilation	Cray 16.0, ROCM 5.5.1	NVHPC 22.11, CUDA 11.8, MKL 22.3
ABINIT	V9.11.x, branche adastra-support,	V9.11.x, branche gpu-develop,

Global execution

Ga_2O_3 – 8640 electronic bands
Chebyshev filtering

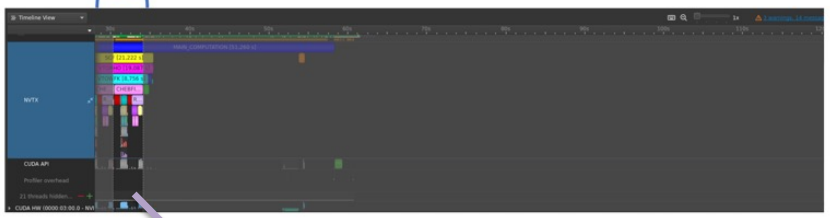
Topaze (Nvidia A100)



Global execution

Ga_2O_3 – 8640 electronic bands
Chebyshev filtering

Topaze (Nvidia A100)



Filering steps

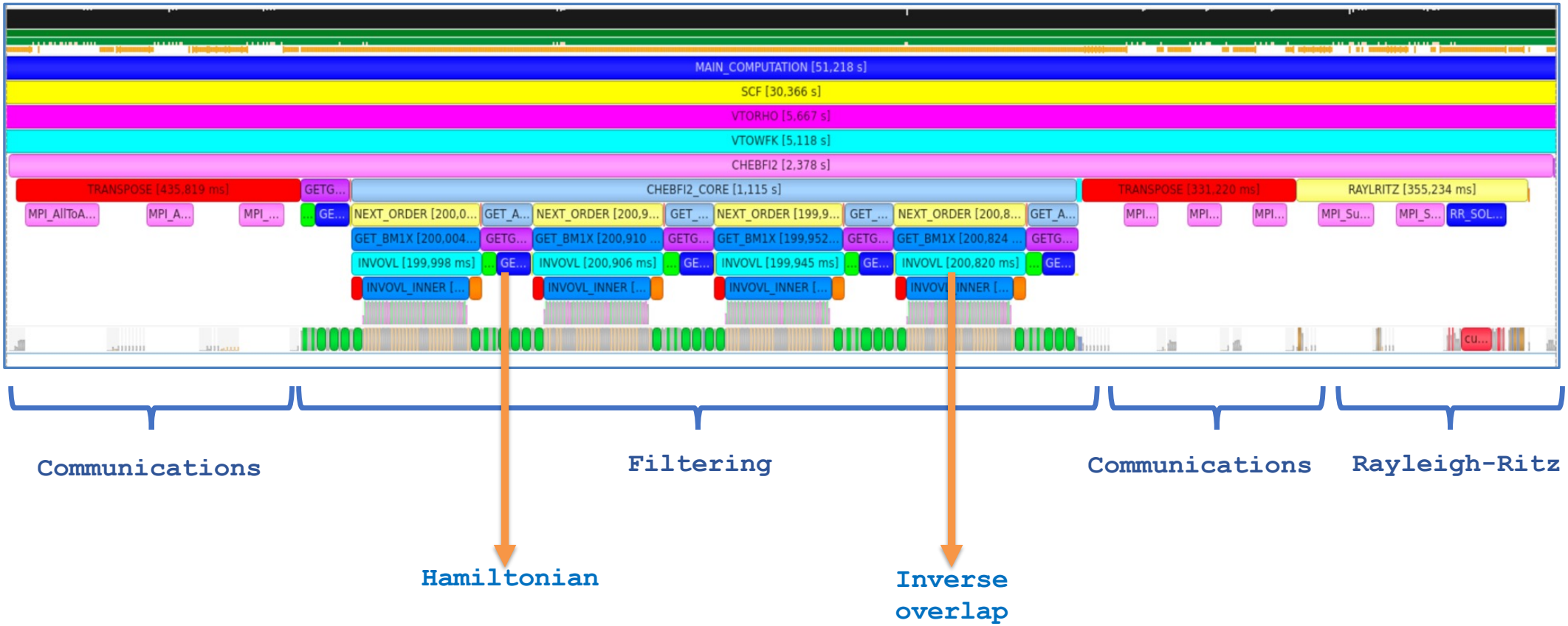
MPI comm

Rayleigh-Ritz procedure
Matrix size = 8640x8640

Zoom: Chebyshev filtering

Ti – 4096 electronic bands
Chebyshev filtering

Topaze (Nvidia A100)



Communications

Filtering

Communications

Rayleigh-Ritz

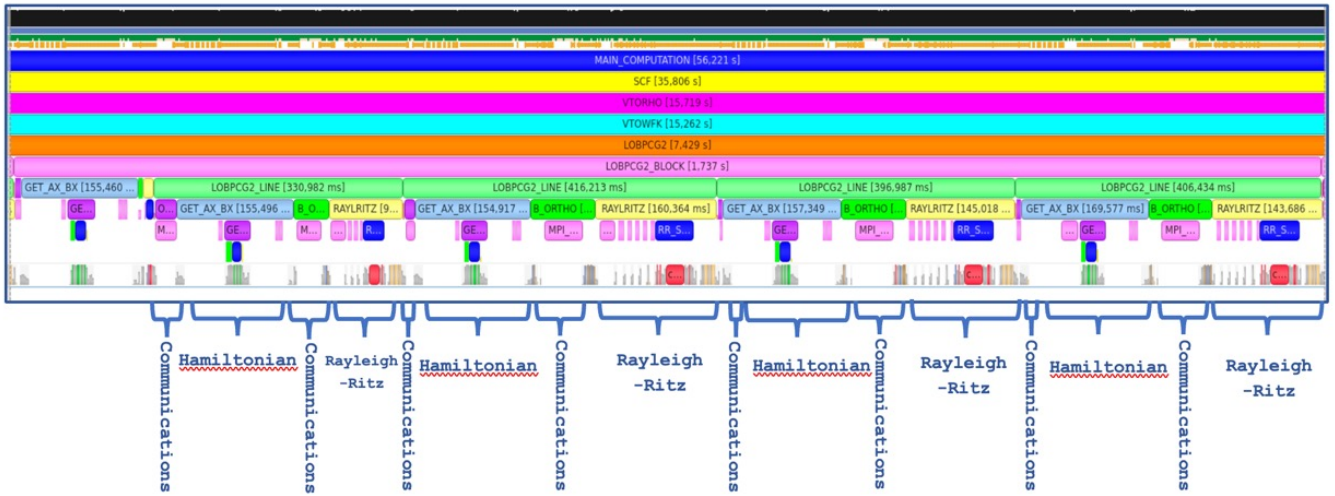
Hamiltonian

Inverse overlap

Chebyshev filtering vs LOBPCG

Ti – 4096 electronic bands
Chebyshev filtering

Topaze (Nvidia A100)



ABINIT+GPU – Cuda/Kokkos - Performance

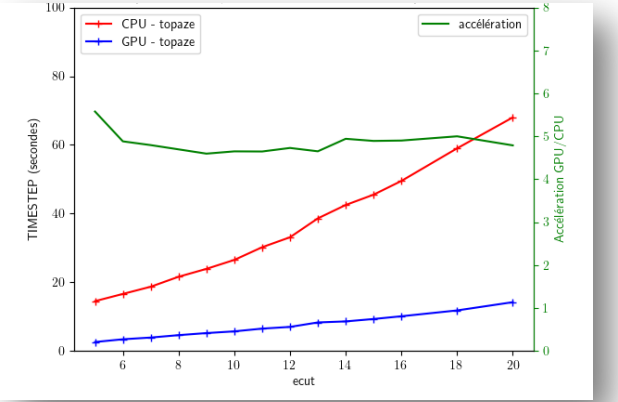
Ti – 255 atoms
Chebyshev filtering

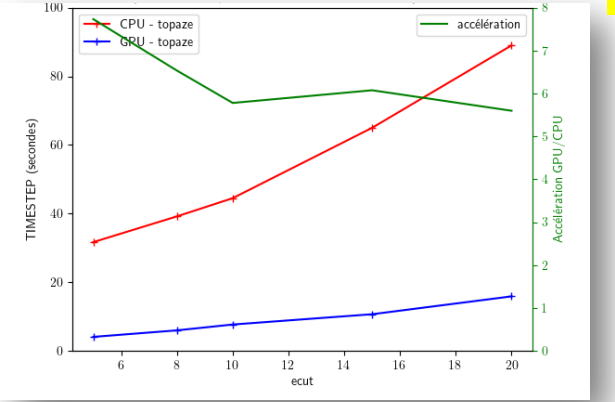
Topaze (Nvidia A100)

Nband= 2048/ecut=5

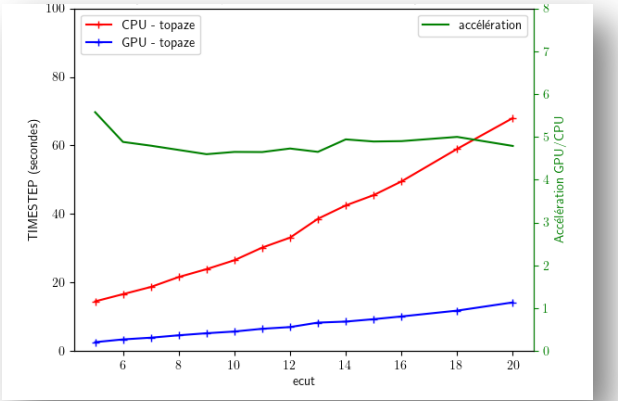
Nband= 4096/ecut=8

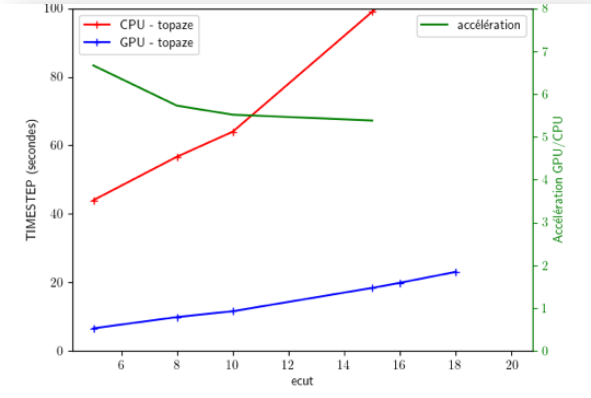
8 GPU
Nvidia A100
vs
256 CPU AMD





16 GPU
Nvidia A100
vs
512 CPU AMD



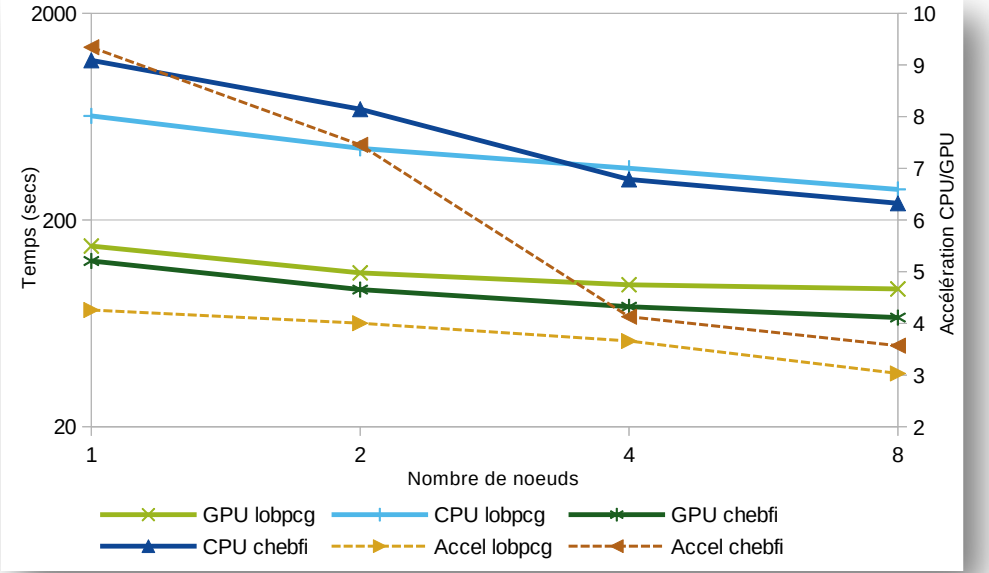


ABINIT+GPU – openMP - Performance

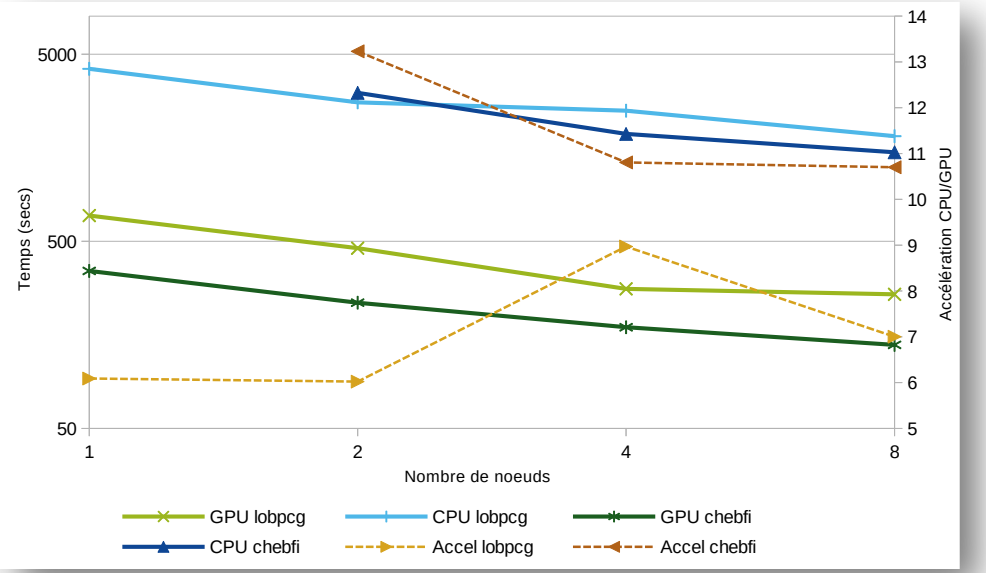
Ti – 255 atoms
Chebyshev filtering

Topaze (Nvidia A100)

Nband= 2048/ecut = 5



Nband= 4096/ecut = 8



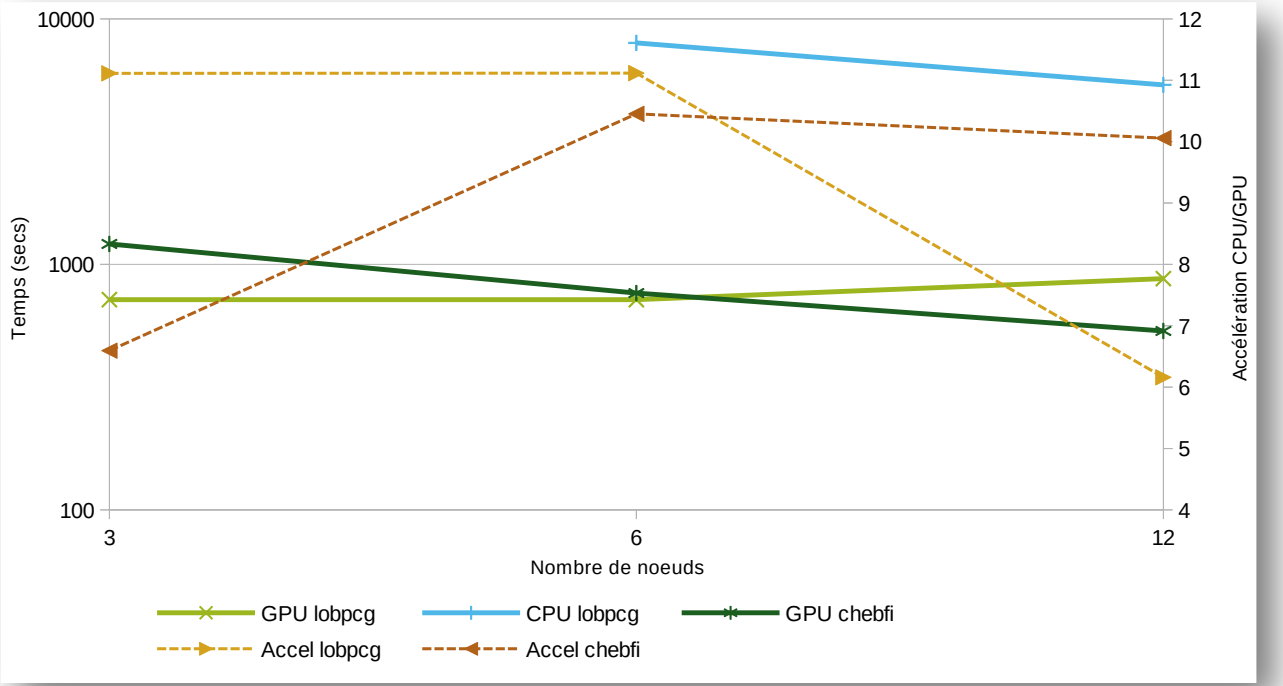
4 GPU	8 GPU	16 GPU	32 GPU
VS	*VS*	*VS*	*VS*
128 CPU	256 CPU	512 CPU	1024 CPU

ABINIT+GPU – openMP - Performance

Ga_2O_3 – 1960 atoms
Chebyshev filtering

Topaze (Nvidia A100)

Nband= 8640 / ecut = 5



12 GPU	24 GPU	48 GPU
VS	*VS*	*VS*
384 CPU	768 CPU	1536 CPU

ABINIT+GPU – Kokkos vs OMP

Ti – 255 atoms
Chebyshev filtering

Topaze (Nvidia A100)

src	host	testcase	algo	getghc_impl	#nodes	#GPU	elapsed	SCFstep
gcc	a100	ti255-A	chebfi2	GPU Kokkos	1	4	144.4	5.34
nvhpc	a100	ti255-A	chebfi2	GPU OpenMP	1	4	139.22	4.81

ABINIT+GPU – Nvidia vs AMD

Ti – 255 atoms
Chebyshev filtering

Topaze (Nvidia A100)
vs
Adastra (AMD Mi250)

Impl.	#nodes	#PPN	#thr	Hôte	bandpp	#SCFstep	elapsed	SCFstep#3	diago
GPU OpenMP	1	4	8	Topaze-a100	1024	22	347.35	11.45	192.513
GPU OpenMP	2	4	8	Topaze-a100	512	22	235	7.06	116.178
GPU OpenMP	4	4	8	Topaze-a100	256	22	173.63	4.82	75.288
GPU OpenMP	8	4	8	Topaze-a100	128	22	139.83	3.51	54.115
GPU OpenMP	1	8	8	Mi250X	512	19	240.49	10.16	178.188
GPU OpenMP	2	8	8	Mi250X	256	19	190.23	7.73	134.804
GPU OpenMP	4	8	8	Mi250X	128	19	175.16	6.85	118.301
GPU OpenMP	8	8	8	Mi250X	64	18	159.65	6.46	105.024

Impl.	#nodes	#PPN	#thr	Hôte	bandpp	#SCFstep	diago	getAX_BX	getBm1X	Rayl-Ritz	gTranspose
GPU OpenMP	1	4	8	Topaze-a100	1024	22	192.5[illegible]3	39.25	41.67[illegible]	39.163	65.657
GPU OpenMP	2	4	8	Topaze-a100	512	22	116.1[illegible]8	20.132	21.52[illegible]	35.325	35.044
GPU OpenMP	4	4	8	Topaze-a100	256	22	75.2[illegible]8	10.386	11.20[illegible]	32.693	18.468
GPU OpenMP	8	4	8	Topaze-a100	128	22	54.1[illegible]5	5.498	6.3[illegible]	31.124	9.76
GPU OpenMP	1	8	8	Mi250X	512	19	178.1[illegible]8	26.025	21.98[illegible]	94.568	20.905
GPU OpenMP	2	8	8	Mi250X	256	19	134.8[illegible]4	12.956	10.59[illegible]	94.147	7.998
GPU OpenMP	4	8	8	Mi250X	128	19	118.3[illegible]1	7.603	6.61[illegible]	94.789	4.536
GPU OpenMP	8	8	8	Mi250X	64	18	105.0[illegible]4	3.973	3.29[illegible]	91.961	2.357

ABINIT+GPU – cuSolver vs ELPA

Ga_2O_3 – 1960 atoms
Chebyshev filtering

Topaze (Nvidia A100)

Impl.	#nodes	#PPN	#thr	Hôte	bandpp	#SCFstep	elapsed	SCFstep#3	diago
GPU OpenMP	3	4	8	Topaze-a100	720	4	590.09	63	322.151
GPU OpenMP	6	4	8	Topaze-a100	360	4	430.04	36.15	182.164
GPU OpenMP	12	4	8	Topaze-a100	180	4	339.72	19.78	99.636
GPU OpenMP ELPA	3	4	1	Topaze-a100	720	4	786.50	95	517.414
GPU OpenMP ELPA	6	4	1	Topaze-a100	360	4	541.89	53.76	290.433
GPU OpenMP ELPA	12	4	1	Topaze-a100	180	4	407.82	31.19	164.966

With cuSolverMP

With ELPA (Nvidia)

Impl.	#nodes	#PPN	#thr	Hôte	bandpp	#SCFstep	diago	getAX_BX	getBm1X	Rayl-Ritz	gTranspose
GPU OpenMP	3	4	8	Topaze-a100	720	4	322.151	57.617	194.65	41.348	15.969
GPU OpenMP	6	4	8	Topaze-a100	360	4	182.164	30.894	96.862	37.948	9.283
GPU OpenMP	12	4	8	Topaze-a100	180	4	99.636	16.421	37.862	37.215	4.753
GPU OpenMP ELPA	3	4	1	Topaze-a100	720	4	517.414	57.321	193.769	227.388	25.457
GPU OpenMP ELPA	6	4	1	Topaze-a100	360	4	290.433	30.864	84.429	141.136	13.791
GPU OpenMP ELPA	12	4	1	Topaze-a100	180	4	164.966	16.352	37.611	99.468	7.193

With cuSolverMP

With ELPA (Nvidia)

ABINIT+GPU – Distributed projections

Ga_2O_3 – 1960 atoms
Chebyshev filtering

Topaze (Nvidia A100)

Impl.	#nodes	#PPN	#thr	Hôte	bandpp	#SCFstep	elapsed	SCFstep#3	diago
GPU OpenMP	3	4	8	Topaze-a100	720	4	590.09	63	322.151
GPU OpenMP	6	4	8	Topaze-a100	360	4	430.04	36.15	182.164
GPU OpenMP	12	4	8	Topaze-a100	180	4	339.72	19.78	99.636
GPU OpenMP distrib. 6 MPI/bloc	6	4	8	Inti-a100	360	4	1111.65	148	816.451
GPU OpenMP distrib. 4 MPI/bloc	6	4	8	Inti-a100	360	4	580.85	59.41	302.864

Impl.	#nodes	#PPN	#thr	Hôte	bandpp	#SCFstep	diago	getAX_BX	getBm1X	Rayl-Ritz	xgTranspose
GPU OpenMP	3	4	8	Topaze-a100	720	4	322.151	57.617	194.66	41.348	15.969
GPU OpenMP	6	4	8	Topaze-a100	360	4	182.164	30.894	96.862	37.948	9.283
GPU OpenMP	12	4	8	Topaze-a100	180	4	99.636	16.421	37.862	37.215	4.753
GPU OpenMP distrib. 6 blocs	6	4	8	Inti-a100	360	4	816.451	421.779	312.048	47.57	10.52
GPU OpenMP distrib. 4 blocs	6	4	8	Inti-a100	360	4	302.864	64.24	153.812	47.643	10.055

ABINIT

How to use

ABINIT+GPU – How to use

- Compile with GPU features (tested with *nvhpc, aocc, cray)*
- Automatic activation of GPU offload if GPU device(s) detected
- `gpu_option` keyword
- `gpu_nl_distrib` keyword if large system
- One MPI process per GPU
- CPU threads can be used
- Ground-state calculations only (feb 2024)

ABINIT+GPU – Activate GPU

gpu_option ¶

Mnemonics: GPU: OPTION to choose the implementation
Mentioned in topic(s): topic_parallelism
Variable type: integer or string
Dimensions: scalar
Default value: 2 if OPENMP_OFFLOAD, 3 if KOKKOS, 1 if CUDA, 0 otherwise.

Added in version: v9.12

Test list (click to open). Rarely used, [7/1148] in all abinit tests, [0/158] in abinit tutorials

Only relevant for Ground-State calculations (`optdriver` == 0). This option is only available if ABINIT executable has been compiled for the purpose of being used with GPU accelerators. It allows to choose between the different GPU programming models available in ABINIT:

- **gpu_option** = "GPU_DISABLED" or **gpu_option** = 0: no use of GPU (even if compiled for GPU).
- **gpu_option** = "GPU_LEGACY" or **gpu_option** = 1: use the "legacy" 2013 implementation of GPU. This is a partial CUDA implementation, using the `nvcc` CUDA compiler. The old LOBPCG algorithm is automatically used to compute the eigenstates (`wfoptalg`=14). The external linear algebra library `MAGMA can also be linked to ABINIT to improve performances on large systems (see `gpu_linalg_limit`).
- **gpu_option** = "GPU_OPENMP" or **gpu_option** = 2: use of the OPENMP_OFFLOAD programming model to execute time consuming parts of the code on GPU. This implementation works on NVidia accelerators, if ABINIT has been compiled with a CUDA compatible compiler and linked with NVidia FFT/linear algebra libraries (cuFFT, cuBLAS and cuSOLVER). It also works on `AMD accelerators (EXPERIMENTAL), if ABINIT has been compiled with a AMD compatible compiler and linked with NVidia FFT/linear algebra libraries (ROCm or HIP).
- **gpu_option** = "GPU_KOKKOS" or **gpu_option** = 3: use of the KOKKOS+CUDA programming model to execute time consuming parts of the code on GPU. This implementation – at present – is only compatible with NVidia accelerators. It required that ABINIT has been linked to the Kokkos and YAKL performance libraries. It also uses NVidia FFT/linear algebra libraries (cuFFT, cuBLAS). The KOKKOS GPU implementation can be used in conjuction with openMP threads on CPU (see `gpu_kokkos_nthrd`).

ABINIT+GPU – Distribution among GPU

gpu_nl_distrib ¶

Mnemonics: GPU: Non-Local operator, DISTRIBute projections
Mentioned in topic(s): topic_parallelism
Variable type: integer
Dimensions: scalar
Default value: 0
Only relevant if: `use_gemm_nonlop` == 1 and `gpu_option` == 2 (OPENMP_OFFLOAD)
Added in version: 9.12

Test list (click to open). Rarely used, [0/1148] in all abinit tests, [0/158] in abinit tutorials

When using GPU acceleration, the wave-function projections ($<\tilde{p}*i*|*Psi*> (used in the non-local operator) are all stored on all GPU devices. **gpu_nl_distrib** forces the distribution of these projections on several GPU devices. This uses less memory per GPU but requires communications beween GPU devices. These communication penalize the execution time. `gpu_nl_splitsize` defines the number of blocks used to split the projections. In standard executions, the distribution is only needed on large use cases to address the high memory needs. Therefore ABINIT uses the “distributed mode” automatically when needed.

gpu_nl_splitsize ¶

Mnemonics: GPU: Non-Local operator SPLITing SIZE
Mentioned in topic(s): topic_parallelism
Variable type: integer
Dimensions: scalar
Default value: 1
Only relevant if: `use_gemm_nonlop` == 1 and `gpu_option` == 2 (OPENMP_OFFLOAD)
Added in version: 9.12

Test list (click to open). Rarely used, [0/1148] in all abinit tests, [0/158] in abinit tutorials

Only relevant when `gpu_nl_distrib` = 1. When using GPU acceleration, the wave-function projections ($<\tilde{p}*i*|*Psi*> (used in the non-local operator) are all stored on all GPU devices. **gpu_nl_splitsize** defines the number of blocks used to split these projections on several GPU devices. In standard executions, the distribution is only needed on large use cases to address the high memory needs. Therefore ABINIT uses the “distributed mode” automatically when needed.

ABINIT+GPU – How to compile

- Not yet documented in official documentation
- Start from a example of `???.ac9` file
  See `~abinit/config/specs/testfarm.conf`
- Use one of the compatible compiler suite
  `nvhpc`, `aocc`, `cray` *tested* (`gnu` *OK for Kokkos implementation)*
- `Nvidia`: use a "**cuda-aware**" MPI library
- Set compiler flags according to your computing architecture
  This section is "tricky" in `???.ac9` *file*

ABINIT+GPU – How to compile – Nvidia + nvhpc

```
# Library location
# ======================================
CUDA_PREFIX=${CUDA_HOME}

# Compilers + flags
# ======================================
FC=mpifort
CC=mpicc
CXX=mpicxx
FCFLAGS_EXTRA="-tp=zen3 -cudalib=cufft -L${CUDA_PREFIX}/math_libs/lib64 -fPIC"
CFLAGS_EXTRA="-tp=zen3"
CXXFLAGS_EXTRA="-tp=zen3"
#enable_zdot_bugfix="yes"
with_optim_flavor="standard"
fcflags_opt_80_tdep="-O1 -g"

# Parallel compilation
# ======================================
with_mpi="yes"
enable_mpi_io="yes"
enable_mpi_inplace="yes"
enable_openmp="yes"
enable_openmp_offload="yes"

# CUDA GPU
# ======================================
with_gpu="${CUDA_PREFIX}"
with_gpu_flavor="cuda-double"
NVCC="${CUDA_PREFIX}/bin/nvcc"
NVCC_CFLAGS="-O3 -arch=sm_80 -Xptxas=-v -g --use_fast_math --compiler-options
-O3,-fPIC,-DHAVE_CONFIG_H  -I${MPI_ROOT}/include"
GPU_CFLAGS=" -I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_CPPFLAGS="-I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_CXXFLAGS="-I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_FCFLAGS="-I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_LIBS="-L${CUDA_PREFIX}/math_libs/lib64 -L${CUDA_PREFIX}/lib64 -lcusolver -lcublas
-lcublasLt -lcufft -lcudart -lstdc++"
GPU_ARCH=80

# Linear Algebra library
# ======================================
with_linalg_flavor="mkl"
LINALG_FCFLAGS="${MKL_FFLAGS}"
LINALG_LIBS="-L${MKLROOT}/lib/intel64 -lmkl_gf_lp64 -lmkl_scalapack_lp64
-lmkl_blacs_openmpi_lp64 -lmkl_core -lmkl_pgi_thread -lpthread -lm"
LINALG_LDFLAGS="${LINALG_LIBS}"

# FFTW3
# ==============================
with_fft_flavor="dfti"

# NetCDF library
# ======================================
with_hdf5="${HDF5_ROOT}"
with_netcdf="${NETCDFC_ROOT}"
with_netcdf_fortran="${NETCDFFORTRAN_ROOT}"

# Additional plugins
# ======================================
with_libxc="${CCCWORKDIR}/INSTALL/libxc-6.1.0"
```

```
# Compilers + flags
# ==========================================
FC=mpifort
CC=mpicc
CXX=mpicxx
FCFLAGS_EXTRA="-tp=zen3 -cudalib=cufft -L${CUDA_PREFIX}/math_libs/lib64 -fPIC"
CFLAGS_EXTRA="-tp=zen3"
CXXFLAGS_EXTRA="-tp=zen3"
#enable_zdot_bugfix="yes"
with_optim_flavor="standard"
fcflags_opt_80_tdep="-O1 -g"
```

```
# CUDA GPU
# ==========================================
with_gpu="${CUDA_PREFIX}"
with_gpu_flavor="cuda-double"
NVCC="${CUDA_PREFIX}/bin/nvcc"
NVCC_CFLAGS="-O3 -arch=sm_80 -Xptxas=-v -g --use_fast_math --compiler-options
-O3,-fPIC,-DHAVE_CONFIG_H  -I${MPI_ROOT}/include"
GPU_CFLAGS=" -I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_CPPFLAGS="-I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_CXXFLAGS="-I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_FCFLAGS="-I${CUDA_PREFIX}/include -I${CUDA_PREFIX}/math_libs/include"
GPU_LIBS="-L${CUDA_PREFIX}/math_libs/lib64 -L${CUDA_PREFIX}/lib64 -lcusolver -lcublas
-lcublasLt -lcufft -lcudart -lstdc++"
GPU_ARCH=80
```

ABINIT+GPU – How to compile – AMD + cray

```
# Compilers + flags
# ========================================
GPU_ARCH=gfx90a
OMPGPU_FLAGS="-D__HIP_PLATFORM_AMD__  -h
acc_model=auto_async_none:no_fast_addr:no_deep_copy"
FCFLAGS_EXTRA="${OMPGPU_FLAGS} -h nofortran_ptr_alias -h nofortran_ptr_overlap -h list=a
-O ipa0 -O thread1 -h noautoprefetch"
CFLAGS_EXTRA="-D__HIP_PLATFORM_AMD__"
CXXFLAGS_EXTRA="-D__HIP_PLATFORM_AMD__"

# Ugly workaround (specific to GPU?)
FCLIBS="-L/opt/rh/gcc-toolset-10/root/usr/lib64
-L/opt/cray/pe/libsci/23.05.1.4/CRAY/12.0/x86_64/lib
-L/opt/cray/pe/hdf5/1.12.2.5/crayclang/16.0/lib
-L/opt/cray/pe/mpich/8.1.26/ofi/cray/16.0/lib -L/opt/cray/pe/mpich/8.1.26/gtl/lib
-L/opt/cray/pe/dsmml/0.2.2/dsmml/lib -lmpi_gtl_hsa -ldl
-L/opt/cray/pe/fftw/3.3.10.4/x86_milan/lib
-L/opt/cray/pe/netcdf/4.9.0.5/crayclang/16.0/lib
-L/opt/cray/pe/cce/16.0.0/cce/x86_64/lib/pkgconfig/../ -lfi -lquadmath -lcraymp
-lcrayacc_amdgpu -lopenacc -lmodules -lcraymath -lf -lu -lcsup -lstdc++ -lpthread
-L/opt/cray/pe/cce/16.0.0/cce-clang/x86_64/lib/clang/16/lib/linux
-lclang_rt.builtins-x86_64 -lhdf5_hl -lhdf5 -lhdf5hl_fortran -lhdf5_fortran
-lsci_cray_mpi_mp -lsci_cray_mp -lfftw3f_mpi -lfftw3f_omp -lfftw3f -lfftw3_mpi -lfftw3_omp
-lfftw3 -lnetcdf -lnetcdff -lmpifort_cray -lmpi_cray -ldsmml -lpgas-shmem -latomic
-lgfortran -lm -lclang_rt.craypgo-x86_64 -lclang_rt.builtins-x86_64
-L/opt/rh/gcc-toolset-10/root/usr/lib/gcc/x86_64-redhat-linux/10
-L/opt/cray/pe/cce/16.0.0/binutils/x86_64/x86_64-pc-linux-gnu/..//x86_64-unknown-linux-gnu
/lib
-L/opt/rocm-5.5.1/llvm/lib:/lus/home/CT9/cin4563/msarraute/work/spack_install/23.02/0.20.0
/linux-rhel8-zen3/clang-16.0.0/libxc-6.1.0-hidp/lib"

# Parallel compilation
# ========================================
with_mpi="yes"
with_debug_flavor="none"
enable_mpi_io="yes"
enable_mpi_inplace="yes"
enable_openmp="yes"
enable_openmp_offload="yes"

# AMD GPU
# ========================================
ROCM_PREFIX=${ROCM_PATH}
with_gpu="${ROCM_PREFIX}"
with_gpu_flavor="hip-double"
GPU_CFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include        "
GPU_CXXFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include       "
GPU_CPPFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include       "
GPU_FCFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include
-I${HIPFORT_ROOT}/include/amdgcn       "
GPU_LIBS="-L${ROCM_PREFIX}/lib -lrocfft -lamdhip64 -lhipfft -lrocblas -lhipblas
-lrocsolver -lhipsolver -lroctx64  -lstdc++ ${OPENMPGPU_FLAGS} "
```

```
# Linear Algebra library
# ========================================
BLAS_PREFIX="$CRAY_LIBSCI_PREFIX_DIR"
with_linalg="$BLAS_PREFIX"
LINALG_FCFLAGS="-I${BLAS_PREFIX}/include"
LINALG_LIBS="-L${BLAS_PREFIX}/lib -lsci_cray"

# FFTW3
# ================================
with_fft_flavor="fftw3"
with_fftw3="${FFTW_ROOT}"
FFTW3_CFLAGS="-I${FFTW_ROOT}/include"
FFTW3_FCFLAGS="-I${FFTW_ROOT}/include"
FFTW3_CPPFLAGS="-I${FFTW_ROOT}/include"
FFTW3_LIBS="-L${FFTW_ROOT}/lib -lfftw3 -lfftw3f -lfftw3_threads "

# NetCDF library
# ========================================
with_hdf5="${HDF5_DIR}"
with_netcdf="${NETCDF_DIR}"
with_netcdf_fortran="${NETCDF_DIR}"

# Additional plugins
# ========================================
 with_libxc="${BENCH_ROOT}/install/GPUB/libxc"

# Miscellaneous options
# ========================================
 enable_zdot_bugfix="no"
 enable_gw_dpc="yes"
```


```
# Compilers + flags
# ========================================
GPU_ARCH=gfx90a
OMPGPU_FLAGS="-D__HIP_PLATFORM_AMD__  -h
acc_model=auto_async_none:no_fast_addr:no_deep_copy"
FCFLAGS_EXTRA="${OMPGPU_FLAGS} -h nofortran_ptr_alias -h nofortran_ptr_overlap -h list=a
-O ipa0 -O thread1 -h noautoprefetch"
CFLAGS_EXTRA="-D__HIP_PLATFORM_AMD__"
CXXFLAGS_EXTRA="-D__HIP_PLATFORM_AMD__"

# Ugly workaround (specific to GPU?)
FCLIBS="-L/opt/rh/gcc-toolset-10/root/usr/lib64
-L/opt/cray/pe/libsci/23.05.1.4/CRAY/12.0/x86_64/lib
-L/opt/cray/pe/hdf5/1.12.2.5/crayclang/16.0/lib
-L/opt/cray/pe/mpich/8.1.26/ofi/cray/16.0/lib -L/opt/cray/pe/mpich/8.1.26/gtl/lib
-L/opt/cray/pe/dsmml/0.2.2/dsmml/lib -lmpi_gtl_hsa -ldl
-L/opt/cray/pe/fftw/3.3.10.4/x86_milan/lib
-L/opt/cray/pe/netcdf/4.9.0.5/crayclang/16.0/lib
-L/opt/cray/pe/cce/16.0.0/cce/x86_64/lib/pkgconfig/../ -lfi -lquadmath -lcraymp
-lcrayacc_amdgpu -lopenacc -lmodules -lcraymath -lf -lu -lcsup -lstdc++ -lpthread
```



```
# AMD GPU
# ========================================
ROCM_PREFIX=${ROCM_PATH}
with_gpu="${ROCM_PREFIX}"
with_gpu_flavor="hip-double"
GPU_CFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include        "
GPU_CXXFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include       "
GPU_CPPFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include       "
GPU_FCFLAGS="-D__HIP_PLATFORM_AMD__ -I${ROCM_PREFIX}/include
-I${HIPFORT_ROOT}/include/amdgcn       "
GPU_LIBS="-L${ROCM_PREFIX}/lib -lrocfft -lamdhip64 -lhipfft -lrocblas -lhipblas
-lrocsolver -lhipsolver -lroctx64  -lstdc++ ${OPENMPGPU_FLAGS} "
```

Conclusion

ABINIT+GPU – Current status

- Ground-state calculations only
- Two programming models
  - *Cuda + Kokkos + YAKL – The development of this branch is (temporary) stopped*
  - *openMP offload – Development still active*
  - *Choice of programming model: compilation + runtime*
- Nvidia + AMD
- Unified memory OK
- LOBPCG + Chebyshev filtering algorithms
- Legacy version (2013) still running for testing purposes
- Everything will be released in next official ABINIT version

ABINIT+GPU – Next improvements

- Better use of MPI (NCLL?, …)
- Improve AMD implementation (ROCm?...)
- Wait for ELPA future versions
- Implement next iterative diagonalization algorithm generation:
  ”Spectrum slicing” algorithm
  Goal : decrease size of Rayleigh-Ritz subspace

ABINIT+GPU – On-going developments

- Density-Functional Perturbation Theory (DFPT)
  Currently under process
- Hartree-Fock operator ➠ Hybrid XC functionals
- Dynamical Mean-Field Theory (DMFT)
- GWr ?

Abinit