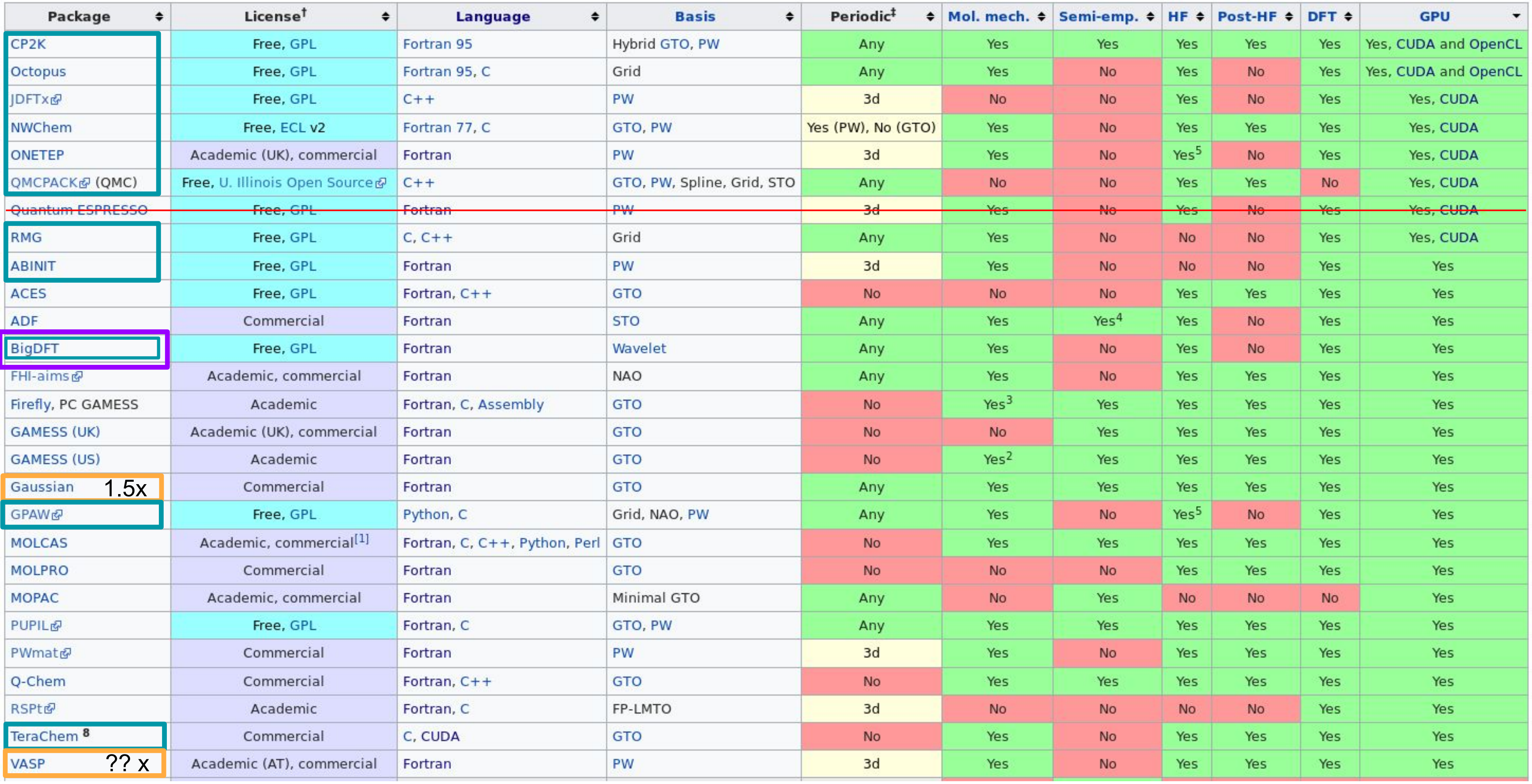
QuantumESPRESSO on accelerators, rebooted

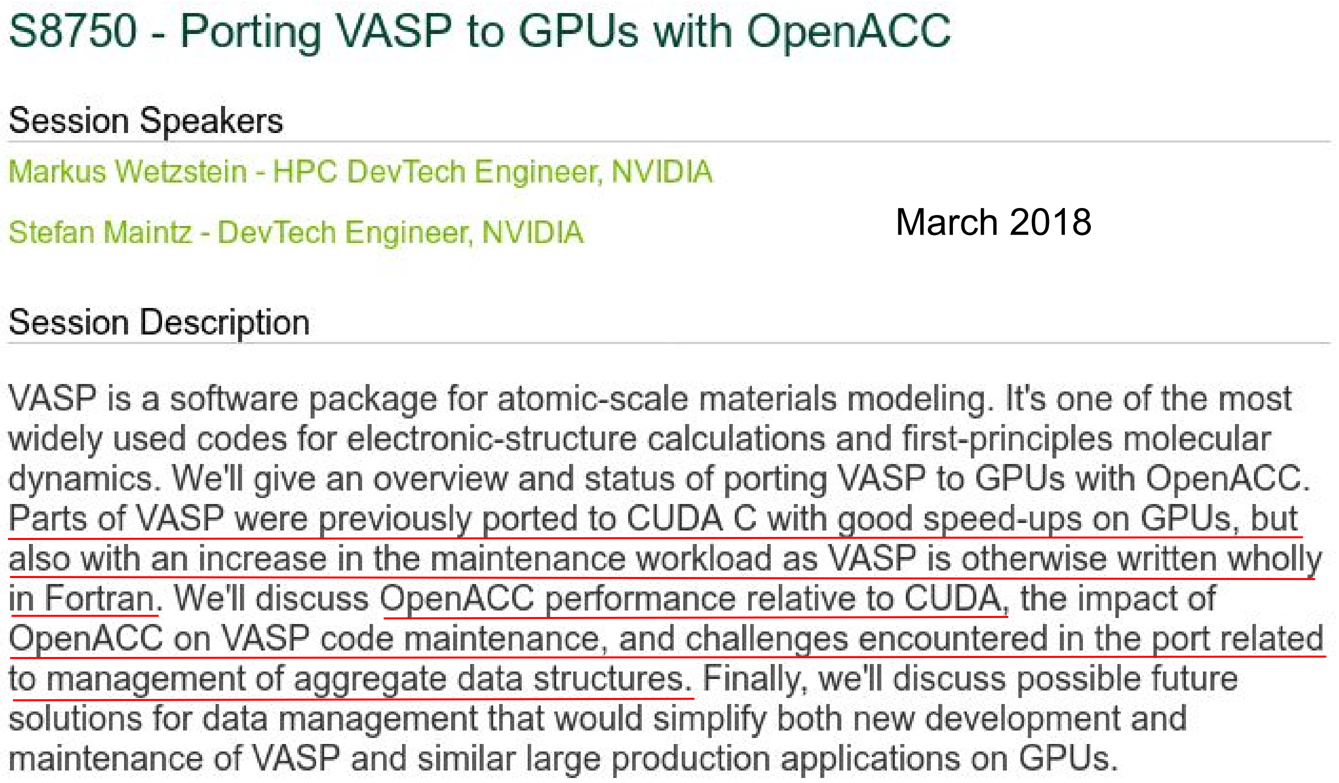
Pietro Bonfà
CINECA

QuantumESPRESSO Developers Meeting, 1st February 2018, SISSA openacc cuda opencl



Package	License†	Language	Basis	Periodic‡	Mol. mech.	Semi-emp.	HF	Post-HF	DFT	GPU
CP2K	Free, GPL	Fortran 95	Hybrid GTO, PW	Any	Yes	Yes	Yes	Yes	Yes	Yes, CUDA and OpenCL
Octopus	Free, GPL	Fortran 95, C	Grid	Any	Yes	No	Yes	No	Yes	Yes, CUDA and OpenCL
JDFTx	Free, GPL	C++	PW	3d	No	No	Yes	No	Yes	Yes, CUDA
NWChem	Free, ECL v2	Fortran 77, C	GTO, PW	Yes (PW), No (GTO)	Yes	No	Yes	Yes	Yes	Yes, CUDA
ONETEP	Academic (UK), commercial	Fortran	PW	3d	Yes	No	Yes[5]	No	Yes	Yes, CUDA
QMCPACK (QMC)	Free, U. Illinois Open Source	C++	GTO, PW, Spline, Grid, STO	Any	No	No	Yes	Yes	No	Yes, CUDA
~~Quantum ESPRESSO~~	~~Free, GPL~~	~~Fortran~~	~~PW~~	~~3d~~	~~Yes~~	~~No~~	~~Yes~~	~~No~~	~~Yes~~	~~Yes, CUDA~~
RMG	Free, GPL	C, C++	Grid	Any	Yes	No	No	No	Yes	Yes, CUDA
ABINIT	Free, GPL	Fortran	PW	3d	Yes	No	No	No	Yes	Yes
ACES	Free, GPL	Fortran, C++	GTO	No	No	No	Yes	Yes	Yes	Yes
ADF	Commercial	Fortran	STO	Any	Yes	Yes[4]	Yes	No	Yes	Yes
BigDFT	Free, GPL	Fortran	Wavelet	Any	Yes	No	Yes	No	Yes	Yes
FHI-aims	Academic, commercial	Fortran	NAO	Any	Yes	No	Yes	Yes	Yes	Yes
Firefly, PC GAMESS	Academic	Fortran, C, Assembly	GTO	No	Yes[3]	Yes	Yes	Yes	Yes	Yes
GAMESS (UK)	Academic (UK), commercial	Fortran	GTO	No	No	Yes	Yes	Yes	Yes	Yes
GAMESS (US)	Academic	Fortran	GTO	No	Yes[2]	Yes	Yes	Yes	Yes	Yes
Gaussian 1.5x	Commercial	Fortran	GTO	Any	Yes	Yes	Yes	Yes	Yes	Yes
GPAW	Free, GPL	Python, C	Grid, NAO, PW	Any	Yes	No	Yes[5]	No	Yes	Yes
MOLCAS	Academic, commercial[1]	Fortran, C, C++, Python, Perl	GTO	No	Yes	Yes	Yes	Yes	Yes	Yes
MOLPRO	Commercial	Fortran	GTO	No	No	No	Yes	Yes	Yes	Yes
MOPAC	Academic, commercial	Fortran	Minimal GTO	Any	No	Yes	No	No	No	Yes
PUPIL	Free, GPL	Fortran, C	GTO, PW	Any	Yes	Yes	Yes	Yes	Yes	Yes
PWmat	Commercial	Fortran	PW	3d	Yes	No	Yes	Yes	Yes	Yes
Q-Chem	Commercial	Fortran, C++	GTO	No	Yes	Yes	Yes	Yes	Yes	Yes
RSPt	Academic	Fortran, C	FP-LMTO	3d	No	No	No	No	Yes	Yes
TeraChem [8]	Commercial	C, CUDA	GTO	No	Yes	No	Yes	Yes	Yes	Yes
VASP ?? x	Academic (AT), commercial	Fortran	PW	3d	Yes	No	Yes	Yes	Yes	Yes

Separate source codes: did it work? Is it needed?



S8750 - Porting VASP to GPUs with OpenACC

Session Speakers

Markus Wetzstein - HPC DevTech Engineer, NVIDIA

Stefan Maintz - DevTech Engineer, NVIDIA

Session Description

VASP is a software package for atomic-scale materials modeling. It's one of the most widely used codes for electronic-structure calculations and first-principles molecular dynamics. We'll give an overview and status of porting VASP to GPUs with OpenACC. Parts of VASP were previously ported to CUDA C with good speed-ups on GPUs, but also with an increase in the maintenance workload as VASP is otherwise written wholly in Fortran. We'll discuss OpenACC performance relative to CUDA, the impact of OpenACC on VASP code maintenance, and challenges encountered in the port related to management of aggregate data structures. Finally, we'll discuss possible future solutions for data management that would simplify both new development and maintenance of VASP and similar large production applications on GPUs.

March 2018

Decision depends on

Source code:

C → CUDA, OpenACC, OpenCL(?)

C++ → CUDA, OpenACC, OpenCL(?)

Fortran → **OpenACC, CudaFortran**

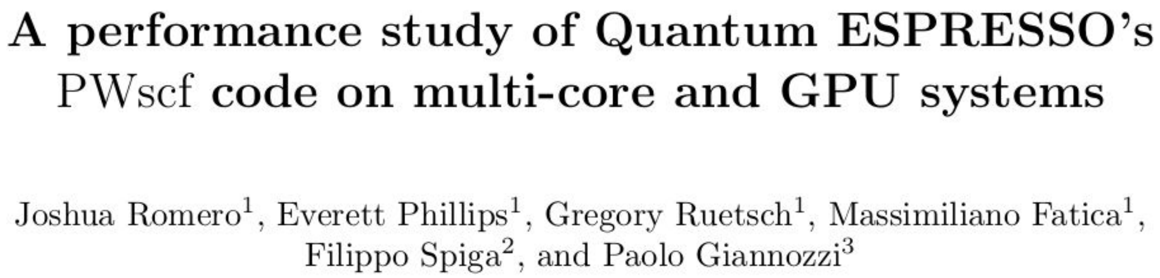
Impact on source:

OpenACC < CudaFortran

Performance:

CudaFortran > OpenACC

What’s in the new *pure fortran* GPU code



A performance study of Quantum ESPRESSO’s PWscf code on multi-core and GPU systems

Joshua Romero[1], Everett Phillips[1], Gregory Ruetsch[1], Massimiliano Fatica[1], Filippo Spiga[2], and Paolo Giannozzi[3]

http://www.dcs.warwick.ac.uk/pmbs/pmbs/PMBS/papers/paper3.pdf

“This paper presented development details and performance of PWscf on CPU and GPU systems. The new GPU version produces accurate results and can reduce the time-to-solution by an average factor of <u>2−3 relative to a reference CPU system</u>.”

What’s in the new *pure fortran* GPU code

Parallel multi-node (MPI+OMP+CUDA Fortran)	✓ (1 MPI per GPU card)
K-points	✓
Gamma point	×
Non-magnetic and collinear magnetic	✓
Non collinear magnetic	×
DFT+U	×
Task groups	×
Norm-conserving, GTH pseudo, 1/r, real space augmentation	×
...	×

Tested v1.0 with test-suite

Changed all gamma points with 1 1 1 0 0 0, still **limited implemented features** lead to

```
All done. ERROR: only 77 out of 180 tests passed.
```

True failures:

- 2 wrong values (bug in stress in v1.0, fixed in develop branch)
- Missing features not advertised
- 1 segfault (spinorbit, should report feature not implemented)

Keeping a single (fortran) source for each subroutine

Obstacles in current PGI compiler for CudaFortran and OpenACC:

- (bug) types are not supported. You have to use pointers.
- GPU programming is different/limited
    - Loops unrolled / reorganized
    - (sometime) kernels needed
    - Different/additional modules and routines (eg. gemm)
- GPU memory is limited (see cegterg)
- Data movement/allocation should be avoided (see cegterg)
- Difficult coexistence with OpenMP (OpenACC)

Keeping a single (fortran) source for each subroutine

Pure fortran without templating

(i.e. what is finally compiled is a portion, and *only* a portion, of the source code)

Openacc (+ cudafortran)

Fortran + templating

(i.e. compiled source code is different from source file content)

Preprocessor directives or PGI's @CUF

Modern templating engine

OpenACC

Leave memory management and acceleration to OpenACC, use cudafortran for "difficult" kernels

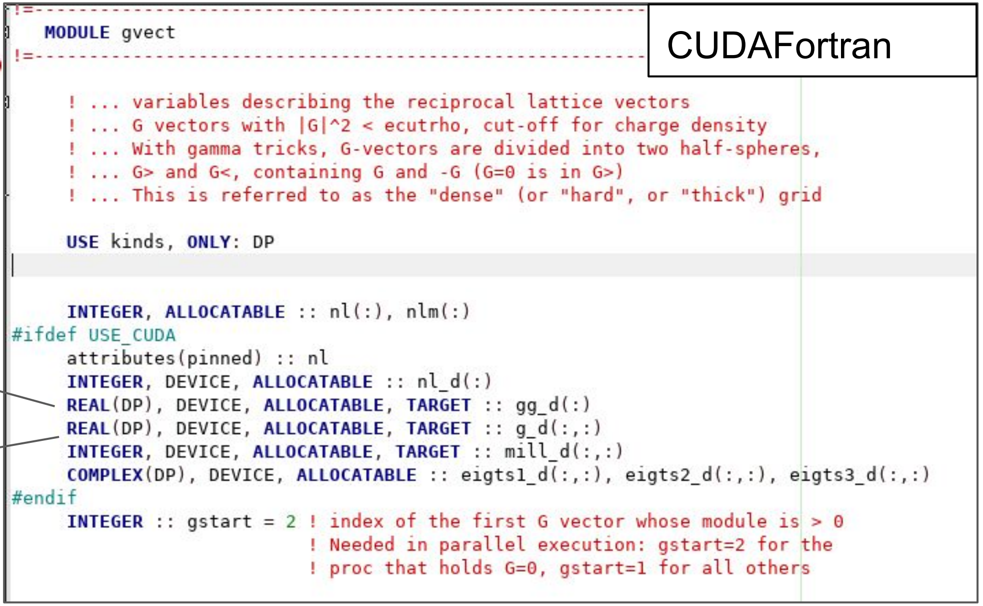
Pros:

- #ifdef -> $!acc
- Less directives
- Open standard
  (partially implemented in gcc7, performances?)
- Reasonable performances (in 3 days ~0.25-0.5x w.r.t. Cudafortran implementation in addusdens)
- GPU directives guarded by logical input parameter

Cons:

- Poorer performances
- Opaque memory allocation and usage
- Openacc/CUDAFortran interoperability is still uncertain/buggy:
  - Compiler fails to compile original code with additional -acc flag
  - Problems with interfaces
  - APIs for CUDA streams
- Difficult coexistence with openmp

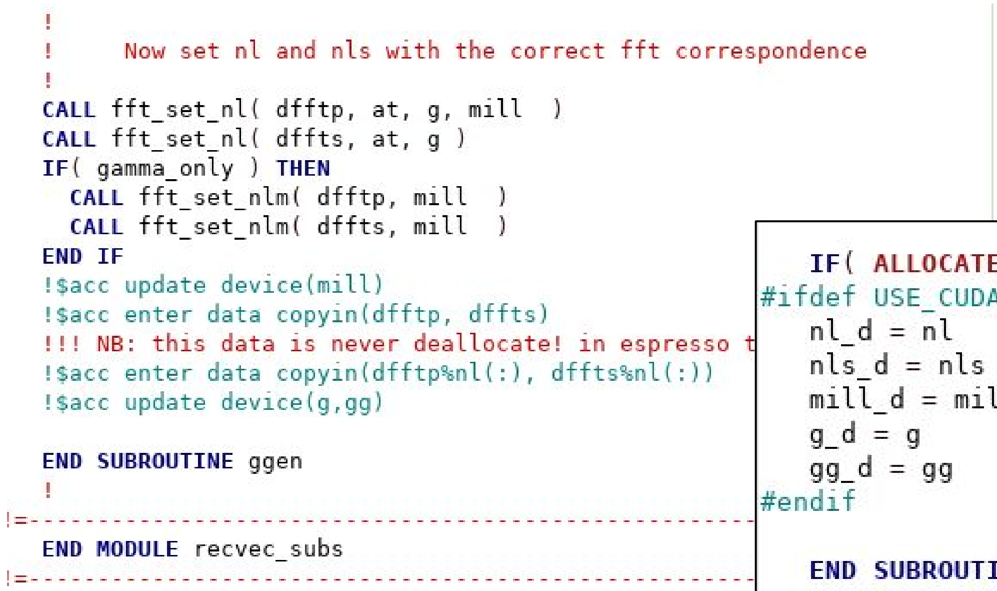
```fortran
INTEGER :: ngm  = 0  ! local  number of G vectors (on this processor)
                     ! with gamma tricks, only vectors in G>
INTEGER :: ngm_g= 0  ! global number of G vectors (summed on all procs)
                     ! in serial execution, ngm_g = ngm
INTEGER :: ngl = 0   ! number of G-vector shells
INTEGER :: ngmx = 0  ! local number of G vectors, maximum across all procs

REAL(DP) :: ecutrho = 0.0_DP ! energy cut-off for charge density
REAL(DP) :: gcutm = 0.0_DP   ! ecutrho/(2 pi/a)^2, cut-off for |G|^2

INTEGER :: gstart = 2 ! index of the first G vector whose module is > 0
                      ! Needed in parallel execution: gstart=2 for the
                      ! proc that holds G=0, gstart=1 for all others

!     G^2 in increasing order (in units of tpiba2=(2pi/a)^2)
!
REAL(DP), ALLOCATABLE, TARGET :: gg(:)
!$acc declare create (gg(:))

!     gl(i) = i-th shell of G^2 (in units of tpiba2)
!     igtongl(n) = shell index for n-th G-vector
!
REAL(DP), POINTER :: gl(:)
INTEGER, ALLOCATABLE, TARGET :: igtongl(:)
!
!     G-vectors cartesian components ( in units tpiba =(2pi/a)  )
!
REAL(DP), ALLOCATABLE, TARGET :: g(:,:)
!$acc declare create (g(:,:))

!     mill = miller index of G vectors (local to each processor)
!            G(:) = mill(1)*bg(:,1)+mill(2)*bg(:,2)+mill(3)*bg(:,3)
!            where bg are the reciprocal lattice basis vectors
!
INTEGER, ALLOCATABLE, TARGET :: mill(:,:)
!$acc declare create (mill(:,:))
```



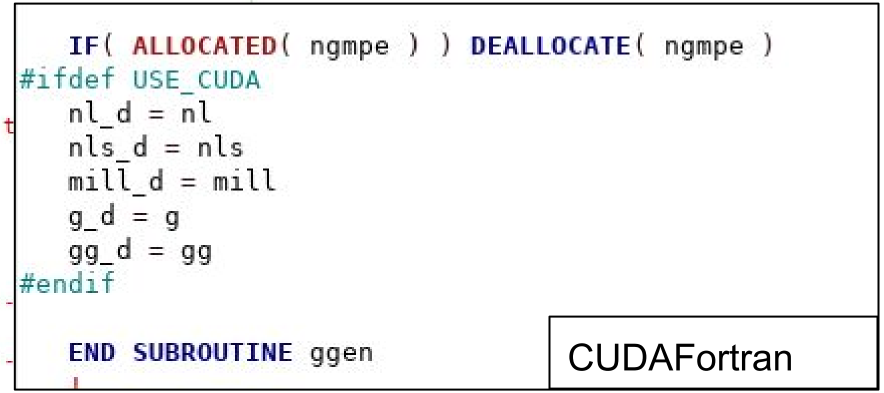
CUDAFortran

```fortran
!=----------------------------------------------------------------------------
   MODULE gvect
!=----------------------------------------------------------------------------

     ! ... variables describing the reciprocal lattice vectors
     ! ... G vectors with |G|^2 < ecutrho, cut-off for charge density
     ! ... With gamma tricks, G-vectors are divided into two half-spheres,
     ! ... G> and G<, containing G and -G (G=0 is in G>)
     ! ... This is referred to as the "dense" (or "hard", or "thick") grid

     USE kinds, ONLY: DP


     INTEGER, ALLOCATABLE :: nl(:), nlm(:)
#ifdef USE_CUDA
     attributes(pinned) :: nl
     INTEGER, DEVICE, ALLOCATABLE :: nl_d(:)
     REAL(DP), DEVICE, ALLOCATABLE, TARGET :: gg_d(:)
     REAL(DP), DEVICE, ALLOCATABLE, TARGET :: g_d(:,:)
     INTEGER, DEVICE, ALLOCATABLE, TARGET :: mill_d(:,:)
     COMPLEX(DP), DEVICE, ALLOCATABLE :: eigts1_d(:,:), eigts2_d(:,:), eigts3_d(:,:)
#endif
     INTEGER :: gstart = 2 ! index of the first G vector whose module is > 0
                           ! Needed in parallel execution: gstart=2 for the
                           ! proc that holds G=0, gstart=1 for all others
```

Is it that different?

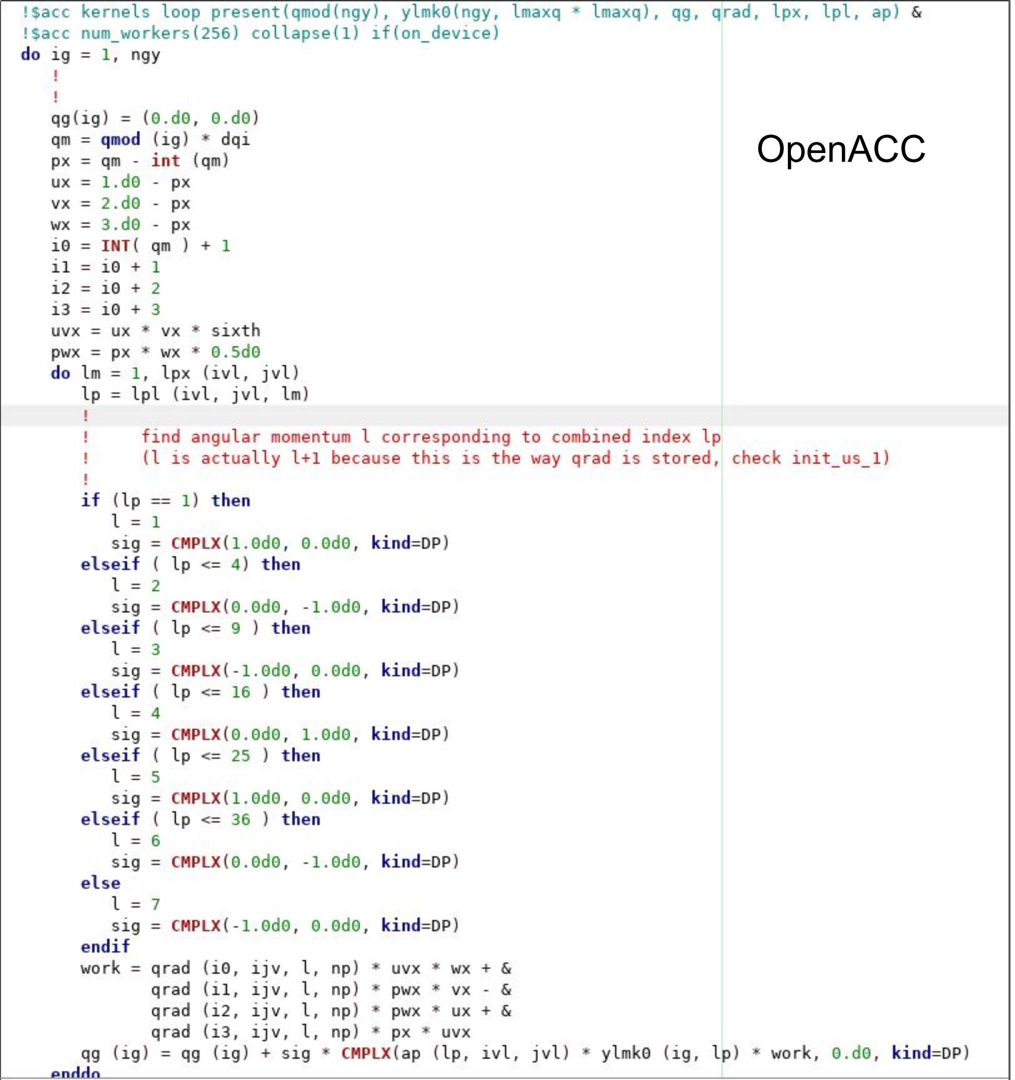


```
  !
  !     Now set nl and nls with the correct fft correspondence
  !
  CALL fft_set_nl( dfftp, at, g, mill  )
  CALL fft_set_nl( dffts, at, g )
  IF( gamma_only ) THEN
    CALL fft_set_nlm( dfftp, mill  )
    CALL fft_set_nlm( dffts, mill  )
  END IF
  !$acc update device(mill)
  !$acc enter data copyin(dfftp, dffts)
  !!! NB: this data is never deallocate! in espresso t
  !$acc enter data copyin(dfftp%nl(:), dffts%nl(:))
  !$acc update device(g,gg)

  END SUBROUTINE ggen
  !
!=----------------------------------------------------------

  END MODULE recvec_subs
!=----------------------------------------------------------
```



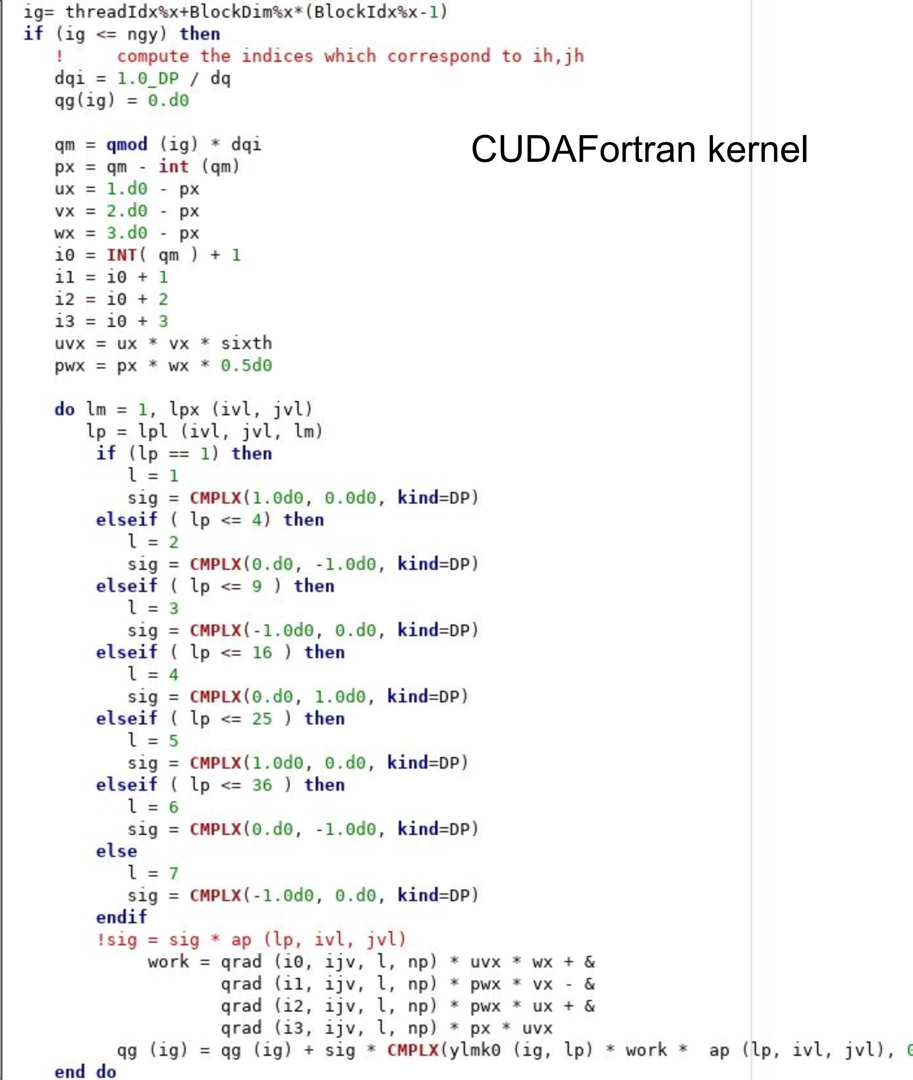
```
   IF( ALLOCATED( ngmpe ) ) DEALLOCATE( ngmpe )
#ifdef USE_CUDA
   nl_d = nl
   nls_d = nls
   mill_d = mill
   g_d = g
   gg_d = gg
#endif

   END SUBROUTINE ggen
```

CUDAFortran



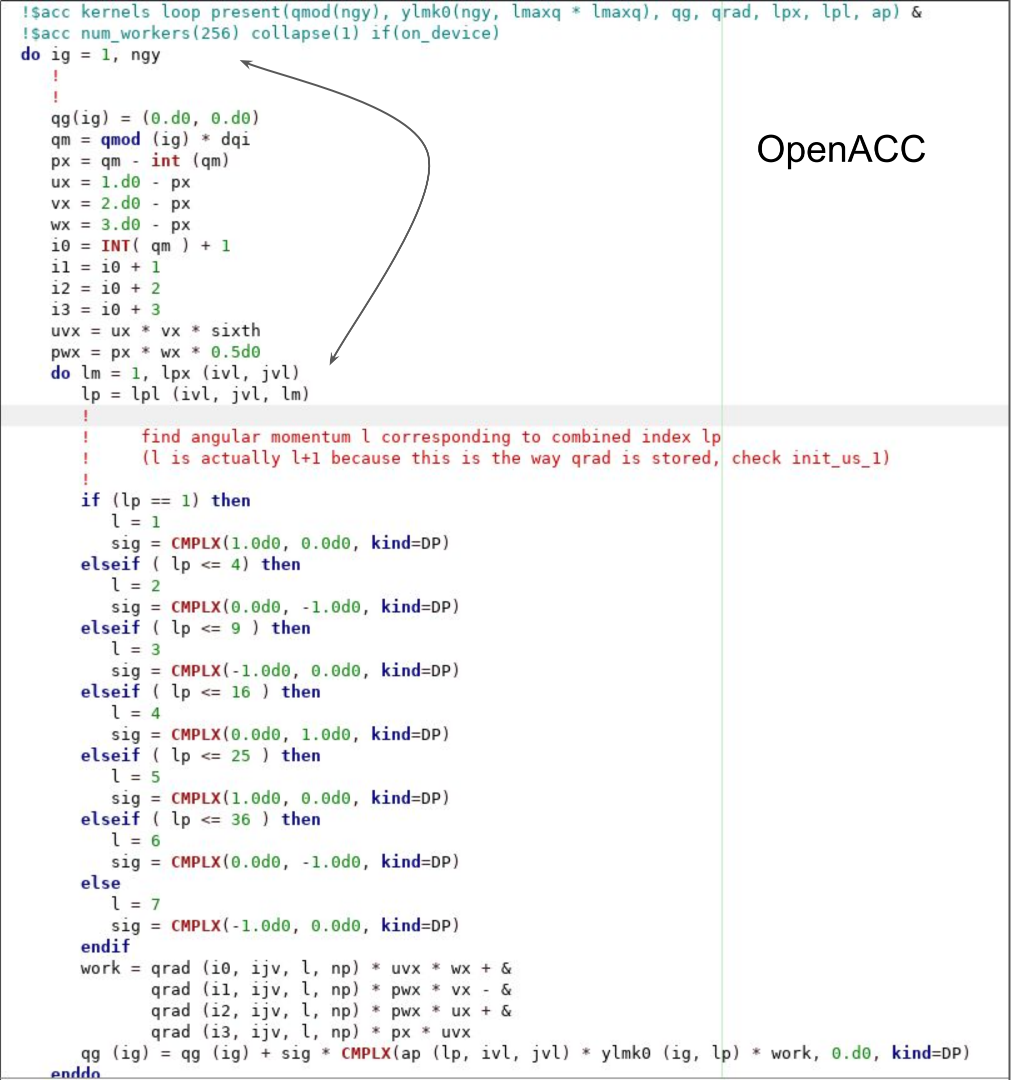
OpenACC

```fortran
!$acc kernels loop present(qmod(ngy), ylmk0(ngy, lmaxq * lmaxq), qg, qrad, lpx, lpl, ap) &
!$acc num_workers(256) collapse(1) if(on_device)
do ig = 1, ngy
   !
   !
   qg(ig) = (0.d0, 0.d0)
   qm = qmod (ig) * dqi
   px = qm - int (qm)
   ux = 1.d0 - px
   vx = 2.d0 - px
   wx = 3.d0 - px
   i0 = INT( qm ) + 1
   i1 = i0 + 1
   i2 = i0 + 2
   i3 = i0 + 3
   uvx = ux * vx * sixth
   pwx = px * wx * 0.5d0
   do lm = 1, lpx (ivl, jvl)
      lp = lpl (ivl, jvl, lm)
      !
      !     find angular momentum l corresponding to combined index lp
      !     (l is actually l+1 because this is the way qrad is stored, check init_us_1)
      !
      if (lp == 1) then
         l = 1
         sig = CMPLX(1.0d0, 0.0d0, kind=DP)
      elseif ( lp <= 4) then
         l = 2
         sig = CMPLX(0.0d0, -1.0d0, kind=DP)
      elseif ( lp <= 9 ) then
         l = 3
         sig = CMPLX(-1.0d0, 0.0d0, kind=DP)
      elseif ( lp <= 16 ) then
         l = 4
         sig = CMPLX(0.0d0, 1.0d0, kind=DP)
      elseif ( lp <= 25 ) then
         l = 5
         sig = CMPLX(1.0d0, 0.0d0, kind=DP)
      elseif ( lp <= 36 ) then
         l = 6
         sig = CMPLX(0.0d0, -1.0d0, kind=DP)
      else
         l = 7
         sig = CMPLX(-1.0d0, 0.0d0, kind=DP)
      endif
      work = qrad (i0, ijv, l, np) * uvx * wx + &
             qrad (i1, ijv, l, np) * pwx * vx - &
             qrad (i2, ijv, l, np) * pwx * ux + &
             qrad (i3, ijv, l, np) * px * uvx
      qg (ig) = qg (ig) + sig * CMPLX(ap (lp, ivl, jvl) * ylmk0 (ig, lp) * work, 0.d0, kind=DP)
   enddo
```



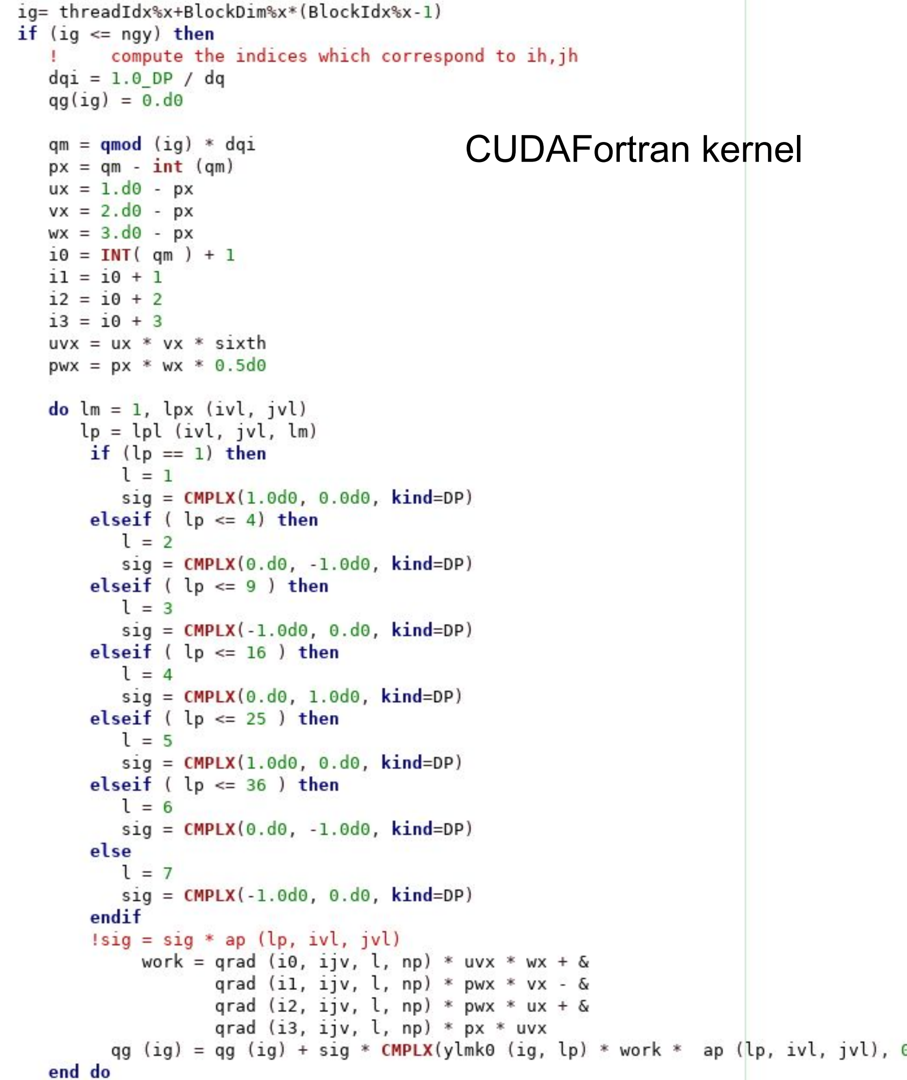
CUDAFortran kernel

```fortran
ig= threadIdx%x+BlockDim%x*(BlockIdx%x-1)
if (ig <= ngy) then
   !     compute the indices which correspond to ih,jh
   dqi = 1.0_DP / dq
   qg(ig) = 0.d0

   qm = qmod (ig) * dqi
   px = qm - int (qm)
   ux = 1.d0 - px
   vx = 2.d0 - px
   wx = 3.d0 - px
   i0 = INT( qm ) + 1
   i1 = i0 + 1
   i2 = i0 + 2
   i3 = i0 + 3
   uvx = ux * vx * sixth
   pwx = px * wx * 0.5d0

   do lm = 1, lpx (ivl, jvl)
      lp = lpl (ivl, jvl, lm)
       if (lp == 1) then
          l = 1
          sig = CMPLX(1.0d0, 0.0d0, kind=DP)
       elseif ( lp <= 4) then
          l = 2
          sig = CMPLX(0.d0, -1.0d0, kind=DP)
       elseif ( lp <= 9 ) then
          l = 3
          sig = CMPLX(-1.0d0, 0.d0, kind=DP)
       elseif ( lp <= 16 ) then
          l = 4
          sig = CMPLX(0.d0, 1.0d0, kind=DP)
       elseif ( lp <= 25 ) then
          l = 5
          sig = CMPLX(1.0d0, 0.d0, kind=DP)
       elseif ( lp <= 36 ) then
          l = 6
          sig = CMPLX(0.d0, -1.0d0, kind=DP)
       else
          l = 7
          sig = CMPLX(-1.0d0, 0.d0, kind=DP)
       endif
       !sig = sig * ap (lp, ivl, jvl)
            work = qrad (i0, ijv, l, np) * uvx * wx + &
                   qrad (i1, ijv, l, np) * pwx * vx - &
                   qrad (i2, ijv, l, np) * pwx * ux + &
                   qrad (i3, ijv, l, np) * px * uvx
      qg (ig) = qg (ig) + sig * CMPLX(ylmk0 (ig, lp) * work *  ap (lp, ivl, jvl),
   end do
```



OpenACC

```
!$acc kernels loop present(qmod(ngy), ylmk0(ngy, lmaxq * lmaxq), qg, qrad, lpx, lpl, ap) &
!$acc num_workers(256) collapse(1) if(on_device)
do ig = 1, ngy
   !
   !
   qg(ig) = (0.d0, 0.d0)
   qm = qmod (ig) * dqi
   px = qm - int (qm)
   ux = 1.d0 - px
   vx = 2.d0 - px
   wx = 3.d0 - px
   i0 = INT( qm ) + 1
   i1 = i0 + 1
   i2 = i0 + 2
   i3 = i0 + 3
   uvx = ux * vx * sixth
   pwx = px * wx * 0.5d0
   do lm = 1, lpx (ivl, jvl)
      lp = lpl (ivl, jvl, lm)
      !
      !     find angular momentum l corresponding to combined index lp
      !     (l is actually l+1 because this is the way qrad is stored, check init_us_1)
      !
      if (lp == 1) then
         l = 1
         sig = CMPLX(1.0d0, 0.0d0, kind=DP)
      elseif ( lp <= 4) then
         l = 2
         sig = CMPLX(0.0d0, -1.0d0, kind=DP)
      elseif ( lp <= 9 ) then
         l = 3
         sig = CMPLX(-1.0d0, 0.0d0, kind=DP)
      elseif ( lp <= 16 ) then
         l = 4
         sig = CMPLX(0.0d0, 1.0d0, kind=DP)
      elseif ( lp <= 25 ) then
         l = 5
         sig = CMPLX(1.0d0, 0.0d0, kind=DP)
      elseif ( lp <= 36 ) then
         l = 6
         sig = CMPLX(0.0d0, -1.0d0, kind=DP)
      else
         l = 7
         sig = CMPLX(-1.0d0, 0.0d0, kind=DP)
      endif
      work = qrad (i0, ijv, l, np) * uvx * wx + &
             qrad (i1, ijv, l, np) * pwx * vx - &
             qrad (i2, ijv, l, np) * pwx * ux + &
             qrad (i3, ijv, l, np) * px * uvx
      qg (ig) = qg (ig) + sig * CMPLX(ap (lp, ivl, jvl) * ylmk0 (ig, lp) * work, 0.d0, kind=DP)
   enddo
```

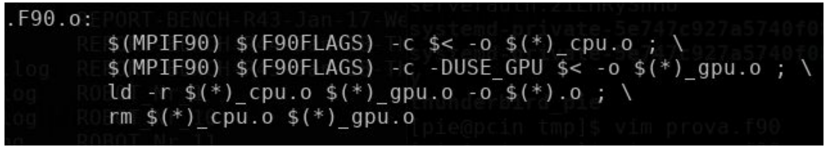


CUDAFortran kernel

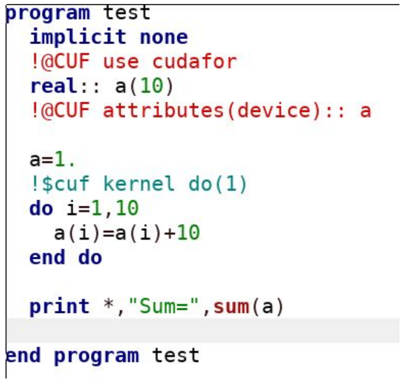
```
ig= threadIdx%x+BlockDim%x*(BlockIdx%x-1)
if (ig <= ngy) then
   !     compute the indices which correspond to ih,jh
   dqi = 1.0_DP / dq
   qg(ig) = 0.d0

   qm = qmod (ig) * dqi
   px = qm - int (qm)
   ux = 1.d0 - px
   vx = 2.d0 - px
   wx = 3.d0 - px
   i0 = INT( qm ) + 1
   i1 = i0 + 1
   i2 = i0 + 2
   i3 = i0 + 3
   uvx = ux * vx * sixth
   pwx = px * wx * 0.5d0

   do lm = 1, lpx (ivl, jvl)
      lp = lpl (ivl, jvl, lm)
       if (lp == 1) then
          l = 1
          sig = CMPLX(1.0d0, 0.0d0, kind=DP)
       elseif ( lp <= 4) then
          l = 2
          sig = CMPLX(0.d0, -1.0d0, kind=DP)
       elseif ( lp <= 9 ) then
          l = 3
          sig = CMPLX(-1.0d0, 0.d0, kind=DP)
       elseif ( lp <= 16 ) then
          l = 4
          sig = CMPLX(0.d0, 1.0d0, kind=DP)
       elseif ( lp <= 25 ) then
          l = 5
          sig = CMPLX(1.0d0, 0.d0, kind=DP)
       elseif ( lp <= 36 ) then
          l = 6
          sig = CMPLX(0.d0, -1.0d0, kind=DP)
       else
          l = 7
          sig = CMPLX(-1.0d0, 0.d0, kind=DP)
       endif
       !sig = sig * ap (lp, ivl, jvl)
            work = qrad (i0, ijv, l, np) * uvx * wx + &
                   qrad (i1, ijv, l, np) * pwx * vx - &
                   qrad (i2, ijv, l, np) * pwx * ux + &
                   qrad (i3, ijv, l, np) * px * uvx
     qg (ig) = qg (ig) + sig * CMPLX(ylmk0 (ig, lp) * work *  ap (lp, ivl, jvl), 0
   end do
```

Templating

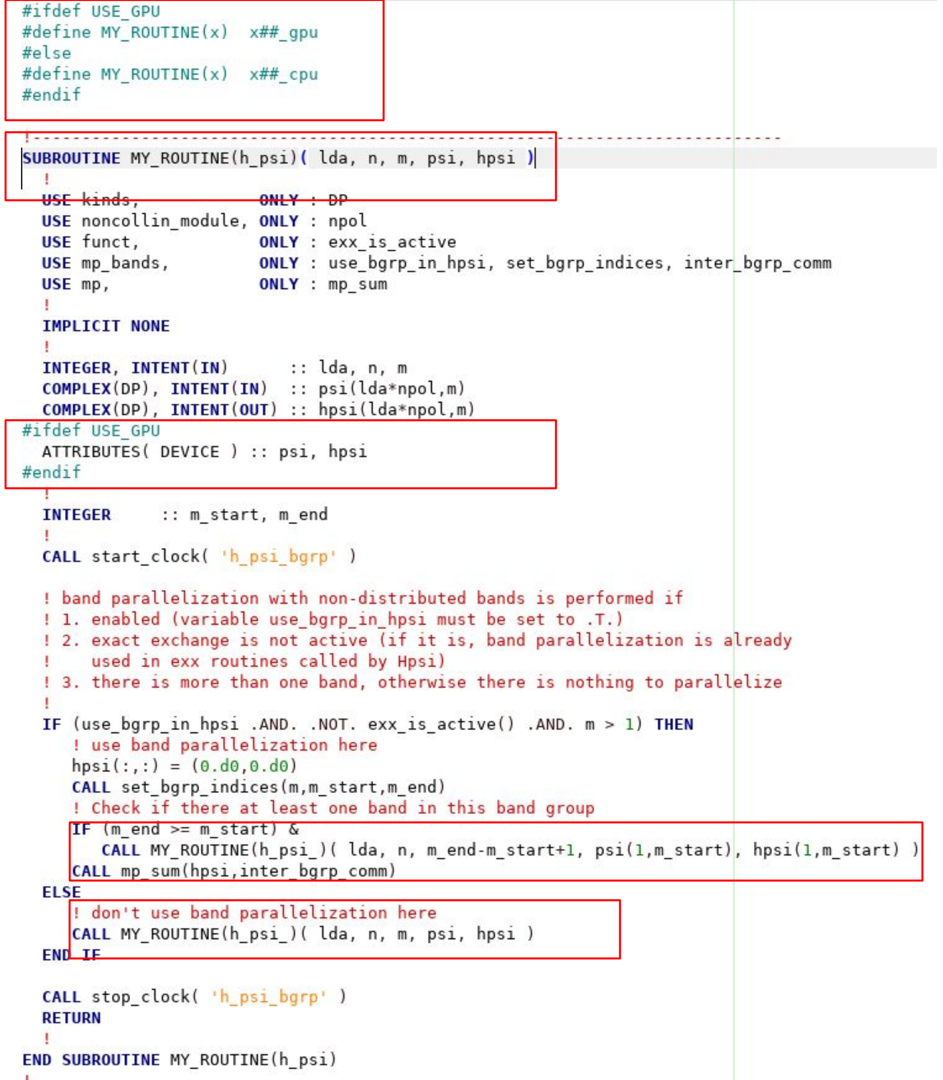
use preprocessor directives as a templating engine to change small portions of the code at/before compile time.



```
.F90.o:
        $(MPIF90) $(F90FLAGS) -c $< -o $(*)_cpu.o ; \
        $(MPIF90) $(F90FLAGS) -c -DUSE_GPU $< -o $(*)_gpu.o ; \
        ld -r $(*)_cpu.o $(*)_gpu.o -o $(*).o ; \
        rm $(*)_cpu.o $(*)_gpu.o
```



```
program test
  implicit none
  !@CUF use cudafor
  real:: a(10)
  !@CUF attributes(device):: a

  a=1.
  !$cuf kernel do(1)
  do i=1,10
    a(i)=a(i)+10
  end do

  print *,"Sum=",sum(a)

end program test
```

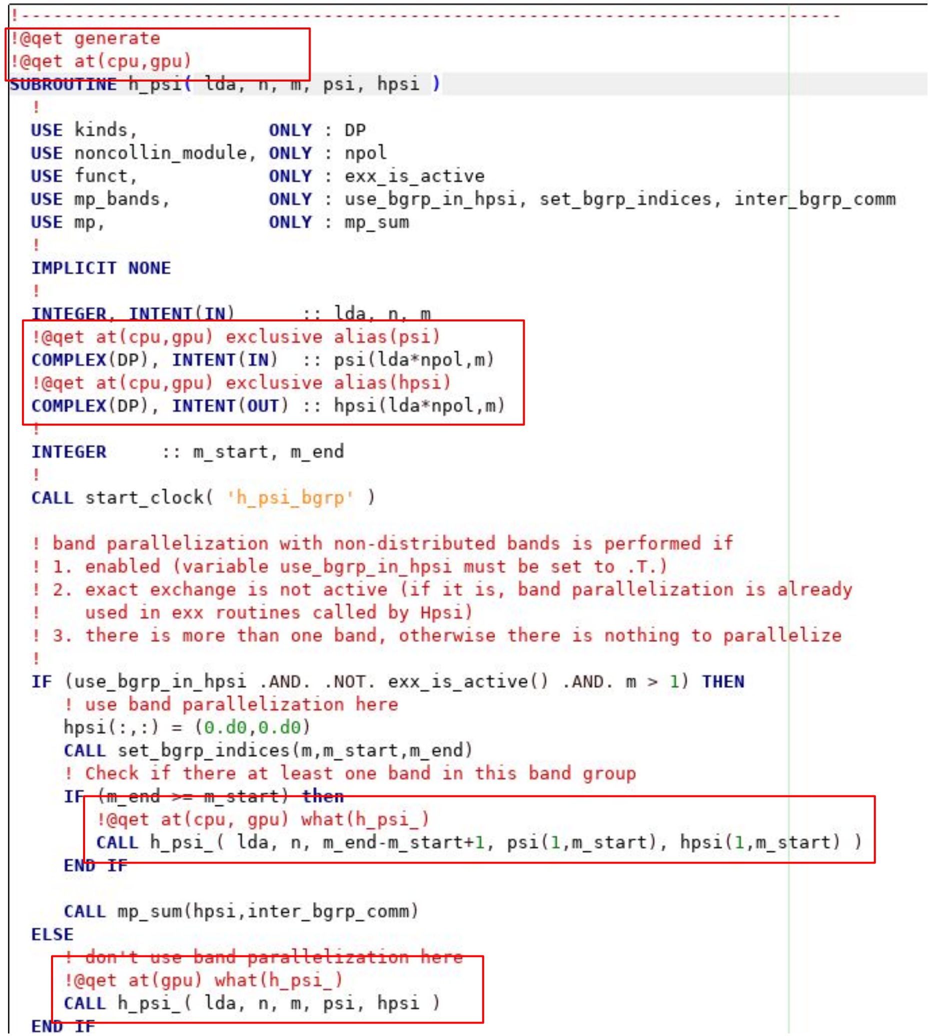


```
#ifdef USE_GPU
#define MY_ROUTINE(x)  x##_gpu
#else
#define MY_ROUTINE(x)  x##_cpu
#endif

!-----------------------------------------------------------------------------------
SUBROUTINE MY_ROUTINE(h_psi)( lda, n, m, psi, hpsi )
  !
  USE kinds,             ONLY : DP
  USE noncollin_module,  ONLY : npol
  USE funct,             ONLY : exx_is_active
  USE mp_bands,          ONLY : use_bgrp_in_hpsi, set_bgrp_indices, inter_bgrp_comm
  USE mp,                ONLY : mp_sum
  !
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN)      :: lda, n, m
  COMPLEX(DP), INTENT(IN)  :: psi(lda*npol,m)
  COMPLEX(DP), INTENT(OUT) :: hpsi(lda*npol,m)
#ifdef USE_GPU
  ATTRIBUTES( DEVICE ) :: psi, hpsi
#endif
  !
  INTEGER     :: m_start, m_end
  !
  CALL start_clock( 'h_psi_bgrp' )

  ! band parallelization with non-distributed bands is performed if
  ! 1. enabled (variable use_bgrp_in_hpsi must be set to .T.)
  ! 2. exact exchange is not active (if it is, band parallelization is already
  !    used in exx routines called by Hpsi)
  ! 3. there is more than one band, otherwise there is nothing to parallelize
  !
  IF (use_bgrp_in_hpsi .AND. .NOT. exx_is_active() .AND. m > 1) THEN
     ! use band parallelization here
     hpsi(:,:) = (0.d0,0.d0)
     CALL set_bgrp_indices(m,m_start,m_end)
     ! Check if there at least one band in this band group
     IF (m_end >= m_start) &
        CALL MY_ROUTINE(h_psi_)( lda, n, m_end-m_start+1, psi(1,m_start), hpsi(1,m_start) )
     CALL mp_sum(hpsi,inter_bgrp_comm)
  ELSE
     ! don't use band parallelization here
     CALL MY_ROUTINE(h_psi_)( lda, n, m, psi, hpsi )
  END IF

  CALL stop_clock( 'h_psi_bgrp' )
  RETURN
  !
END SUBROUTINE MY_ROUTINE(h_psi)
```

Templating

Use a real templating engine (Jinja2, used by Mozzilla, Sourceforge, Instagram,…)

Replace ugly jinja syntax with custom commands that are Fortran aware.



```
!----------------------------------------------------------------------------
!@qet generate
!@qet at(cpu,gpu)
SUBROUTINE h_psi( lda, n, m, psi, hpsi )
  !
  USE kinds,              ONLY : DP
  USE noncollin_module,   ONLY : npol
  USE funct,              ONLY : exx_is_active
  USE mp_bands,           ONLY : use_bgrp_in_hpsi, set_bgrp_indices, inter_bgrp_comm
  USE mp,                 ONLY : mp_sum
  !
  IMPLICIT NONE
  !
  INTEGER, INTENT(IN)      :: lda, n, m
  !@qet at(cpu,gpu) exclusive alias(psi)
  COMPLEX(DP), INTENT(IN)  :: psi(lda*npol,m)
  !@qet at(cpu,gpu) exclusive alias(hpsi)
  COMPLEX(DP), INTENT(OUT) :: hpsi(lda*npol,m)
  !
  INTEGER     :: m_start, m_end
  !
  CALL start_clock( 'h_psi_bgrp' )

  ! band parallelization with non-distributed bands is performed if
  ! 1. enabled (variable use_bgrp_in_hpsi must be set to .T.)
  ! 2. exact exchange is not active (if it is, band parallelization is already
  !    used in exx routines called by Hpsi)
  ! 3. there is more than one band, otherwise there is nothing to parallelize
  !
  IF (use_bgrp_in_hpsi .AND. .NOT. exx_is_active() .AND. m > 1) THEN
     ! use band parallelization here
     hpsi(:,:) = (0.d0,0.d0)
     CALL set_bgrp_indices(m,m_start,m_end)
     ! Check if there at least one band in this band group
     IF (m_end >= m_start) then
        !@qet at(cpu, gpu) what(h_psi_)
        CALL h_psi_( lda, n, m_end-m_start+1, psi(1,m_start), hpsi(1,m_start) )
     END IF

     CALL mp_sum(hpsi,inter_bgrp_comm)
  ELSE
     ! don't use band parallelization here
     !@qet at(gpu) what(h_psi_)
     CALL h_psi_( lda, n, m, psi, hpsi )
  END IF
```

Plan for the integration of the GPU code

1. Create a gpu branch and keep it aligned with master branch
2. Work on libraries with CUDAFortran
    a. Diagonalization
       using https://github.com/NVIDIA/Eigensolver_gpu
    b. MPI interfaces for data hosted on the device
    c. FFTXlib
    d. KS_Solvers (?)

    $\left.\begin{array}{l} \text{a, b, c} \end{array}\right\}$ Also needed for OpenACC
3. Get all tests working using the GPU version of the libraries
4. Merge CUDAFortran enabled libraries into master.
5. Work on the rest of the code, avoid duplication as much as possible and check the impact of the changes done for the GPU on the CPU performances.