CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

GPU acceleration of electronic structure calculations

Trieste, May 29th 2015

Anton Kozhevnikov, CSCS

Introduction

Swiss National Supercomputing Center (CSCS)

Piz Daint @CSCS

Cray XC30: 5’272 nodes of 8-core SandyBridge@2.6GHz + NVIDIA K20X
with Dragonfly network topology

Theoretical peak performance: 7.787 Petaflops

General GPU porting strategy

Scientific community applications are typically:

- monolithic Fortran90
- MPI with poor OpenMP implementation
- ignorant of GPU

Usual steps of porting such applications

- cleanup and refactor the code
- (probably) change the data layout
- fully utilize CPU threads (this helps t
  the compute-intensive kernels of the
- move compute-intensive kernels to G

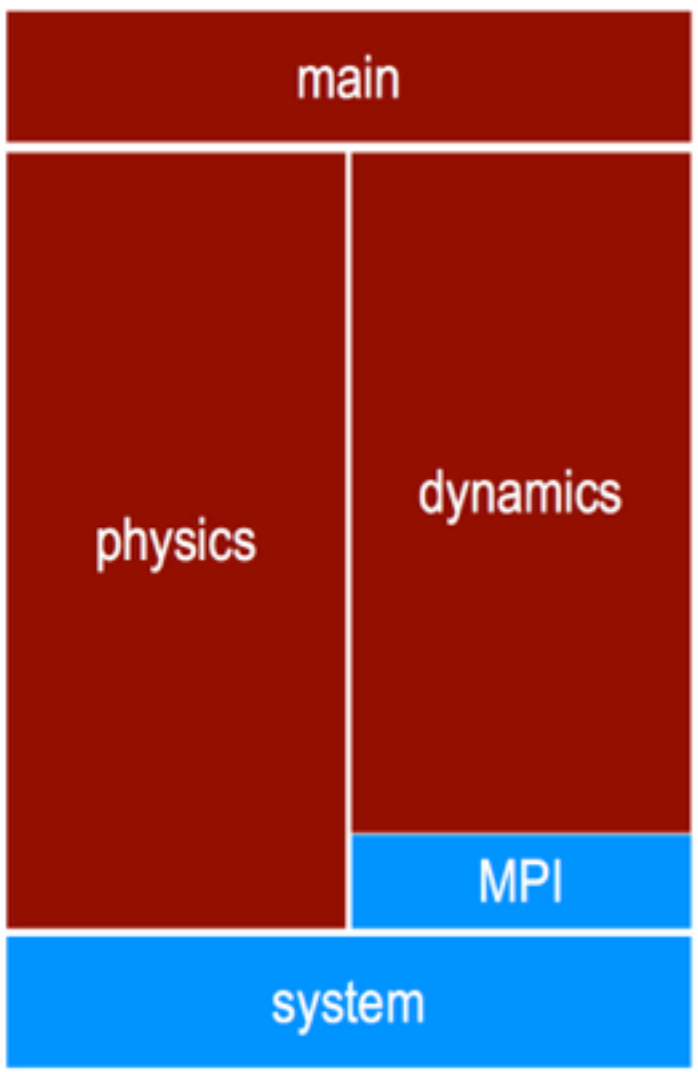
General GPU porting strategy

Separation of concerns is a key to success!

Scientist do a scientific-related coding and prototyping of new features. Low level platform- and hardware-specific functionality is “outsourced” to HPC specialists and wrapped in some kind of domain-specific library.

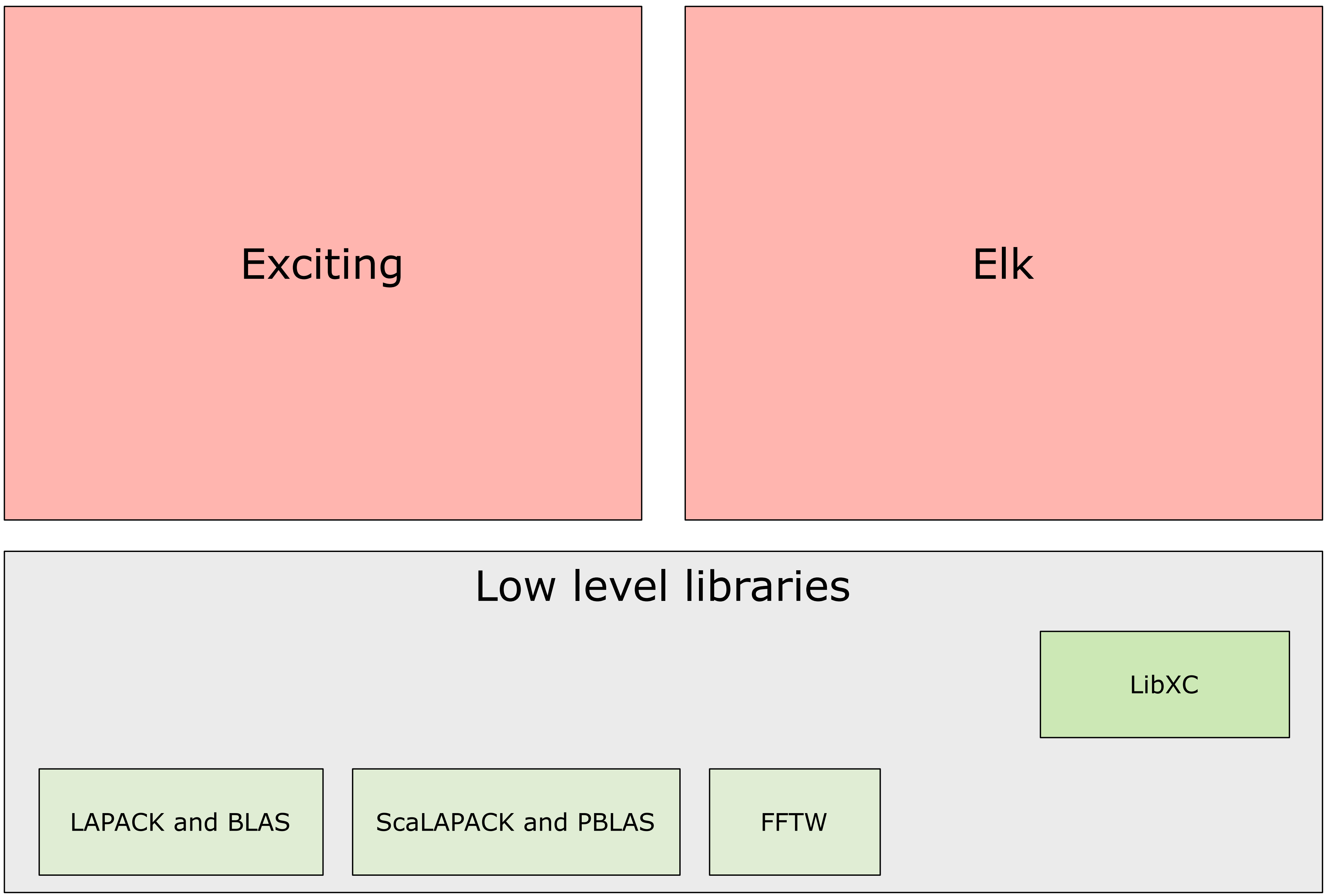
General GPU porting strategy

Climate code COSMO



Electronic structure codes

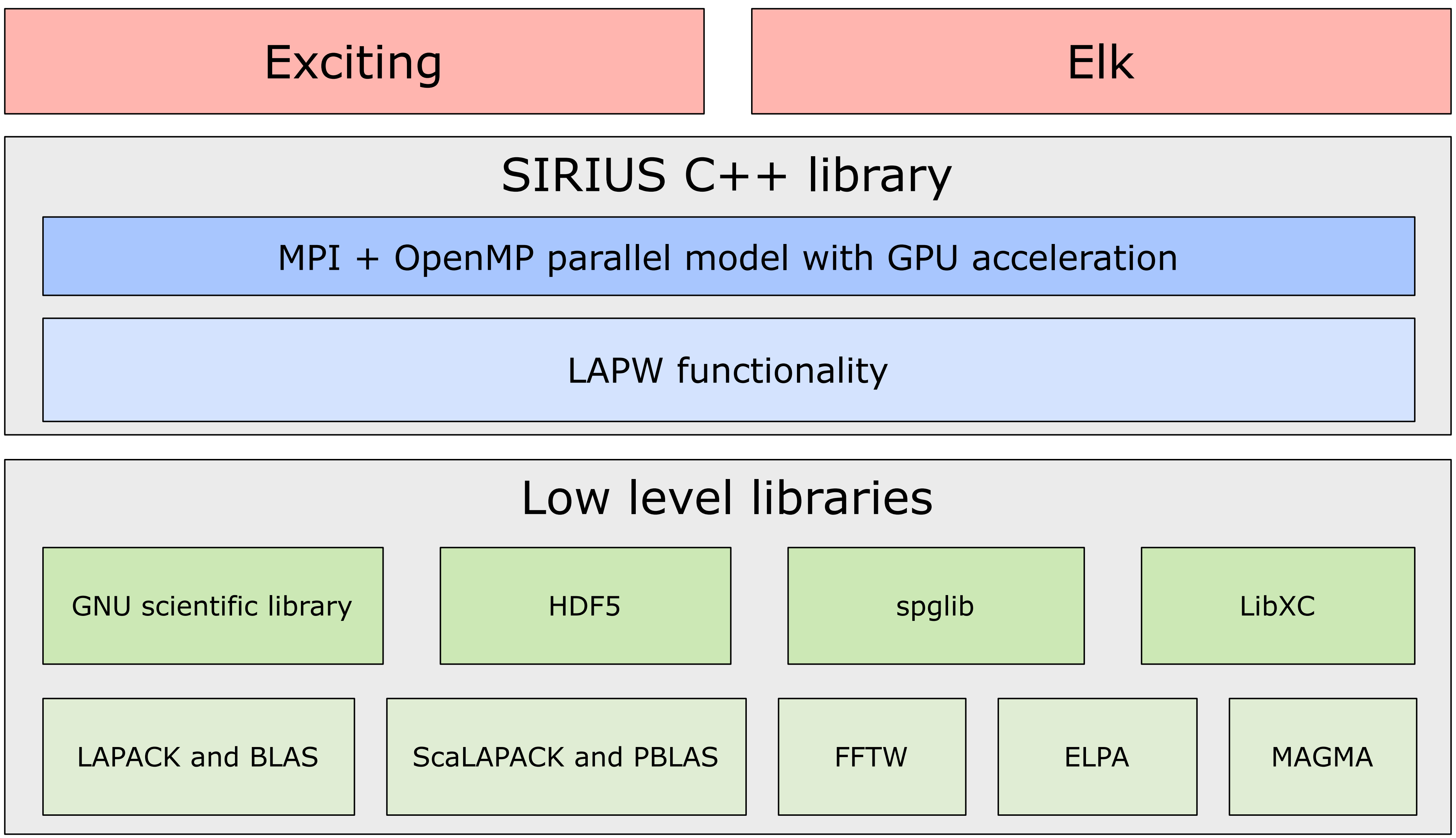
PRACE-2IP WP8: how to approach the problem of community code refactoring in a more systematic way?



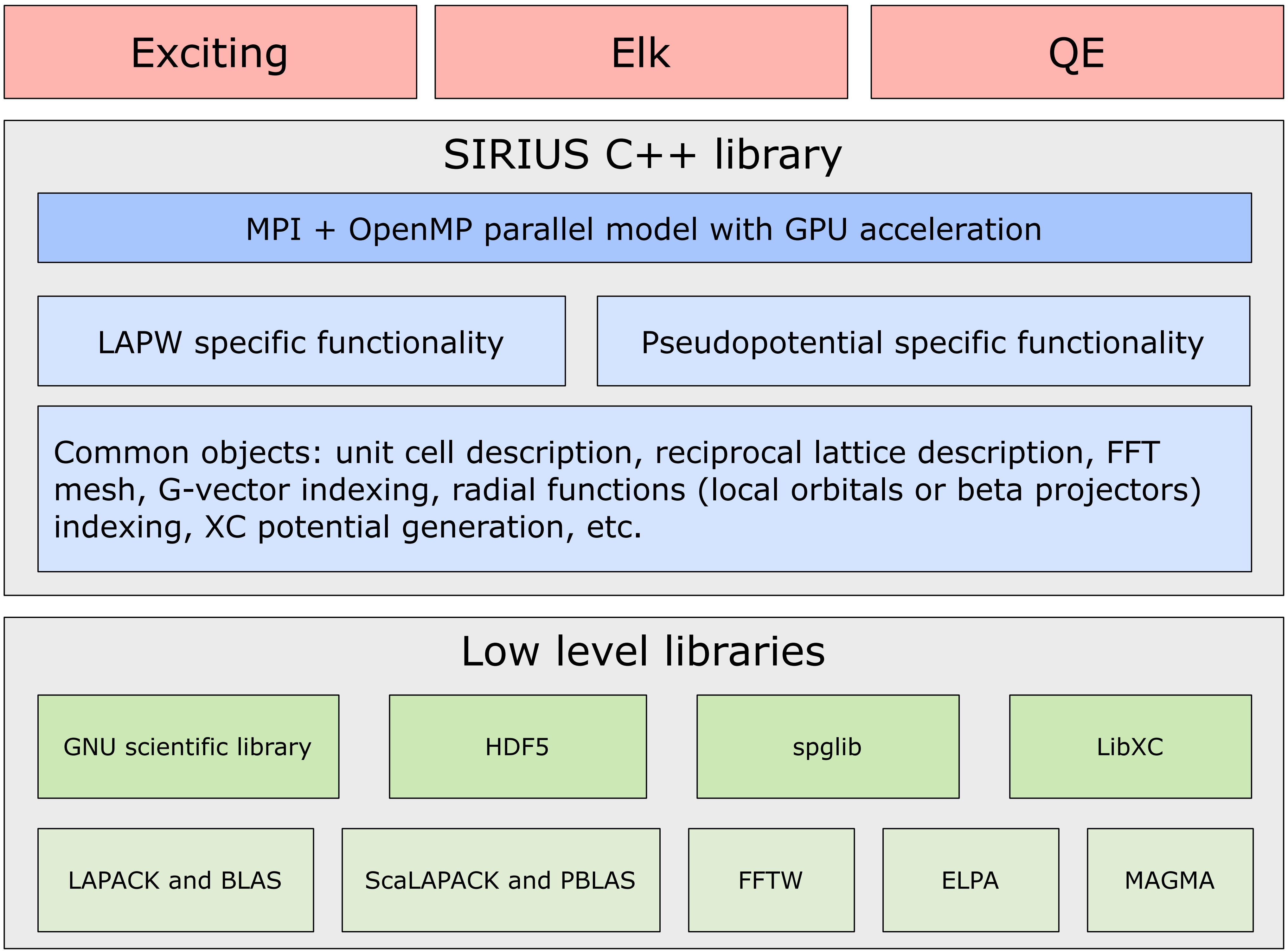
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Electronic structure codes

SIRIUS is a prototype low level library developed under PRACE-2IP work package 8 (community codes refactoring and optimization) as a general-purpose solution for optimization and scaling of both Exciting and Elk full-potential LAPW codes.



Electronic structure codes



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Kohn-Sham equations of DFT

Eigen-value problem

$$\left(-\frac{1}{2}\Delta + v_{eff}(\mathbf{r})\right)\psi_j(\mathbf{r}) = \varepsilon_j \psi_j(\mathbf{r})$$

Density generation

$$\rho^{new}(\mathbf{r}) = \sum_j |\psi_j(\mathbf{r})|^2$$

Density mixing

$$\rho(\mathbf{r}) = \alpha \rho^{new}(\mathbf{r}) + (1-\alpha)\rho^{old}(\mathbf{r})$$

Effective potential construction

$$v_{eff}(\mathbf{r}) = \int \frac{\rho(\mathbf{r}')}{|\mathbf{r}'-\mathbf{r}|} d\mathbf{r}' + v_{XC}[\rho](\mathbf{r}) + v_{ext}(\mathbf{r})$$

Charge density construction

Wave-functions are represented by a set of plane-wave expansion coefficients:

$$\psi_j(\mathbf{r}) = \sum_{\mathbf{G}} e^{i\mathbf{G}\mathbf{r}} \tilde{\psi}_j(\mathbf{G})$$

Charge density is a sum of squared absolute values of wave-functions:

$$\tilde{\psi}_j(\mathbf{G}) \xrightarrow{FFT^{-1}} \psi_j(\mathbf{r}) \rightarrow \rho(\mathbf{r}) = \sum_j |\psi_j(\mathbf{r})|^2$$

Charge density construction

$$\tilde{\psi}_j(\mathbf{G}) \xrightarrow{FFT^{-1}} \psi_j(\mathbf{r}) \rightarrow \rho(\mathbf{r}) = \sum_j |\psi_j(\mathbf{r})|^2$$

version #1

```
for j in [1:num_psi]
  ! transform psi to real space
  inverse_fft(psi(:,j), psi_of_r(:))
  ! accumulate in density
  rho(:) += abs(psi_of_r(:))**2
end
```

Charge density construction

$$\tilde{\psi}_j(\mathbf{G}) \xrightarrow{FFT^{-1}} \psi_j(\mathbf{r}) \rightarrow \rho(\mathbf{r}) = \sum_j |\psi_j(\mathbf{r})|^2$$

version #2

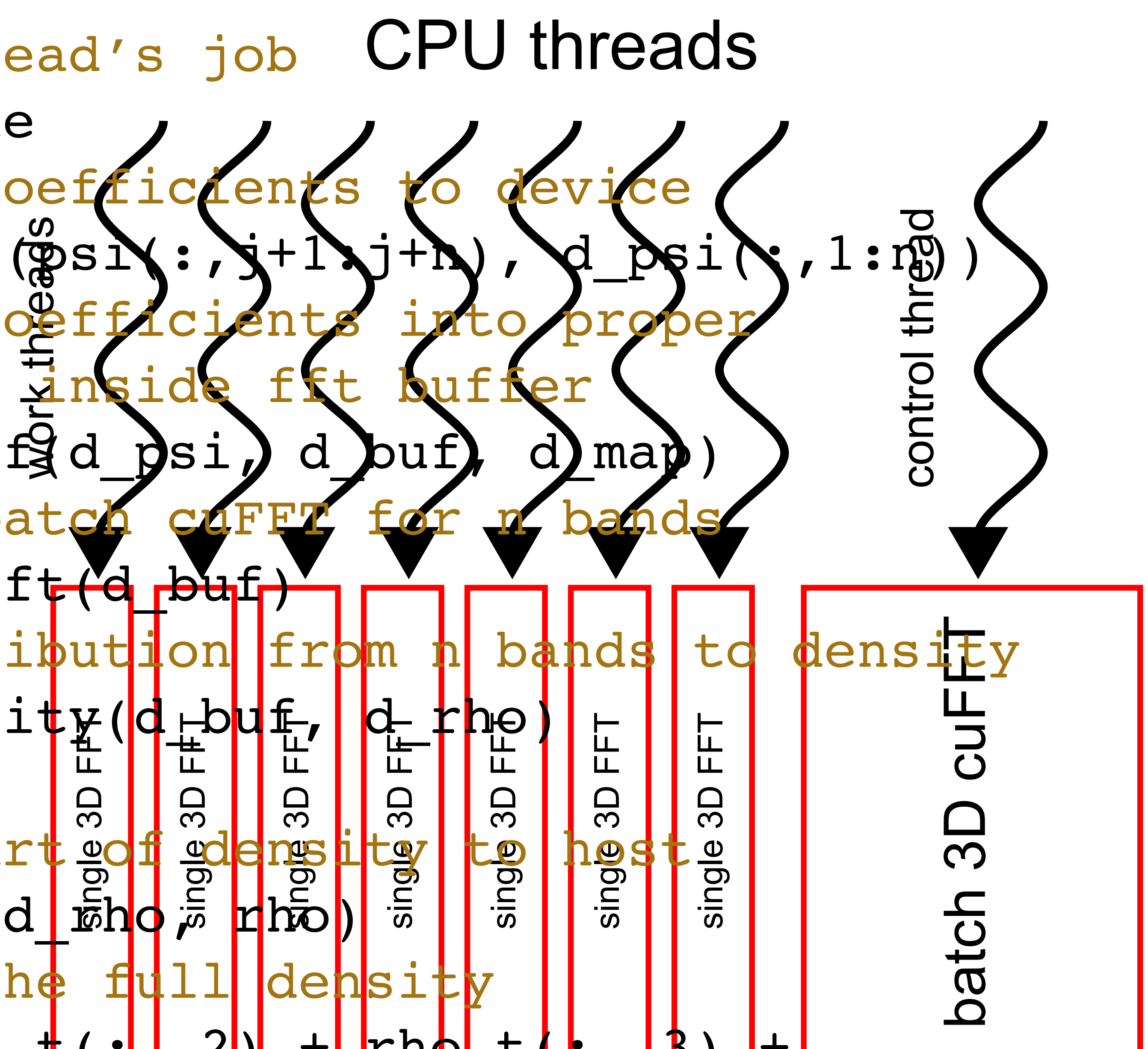
```
$omp parallel
tid = thread_id()
$omp for
for j in [1:num_psi]
  ! transform psi to real space
  inverse_fft(psi(:,j), psi_of_r(:, tid))
  ! accumulate in density
  rho_t(:, tid) += abs(psi_of_r(:, tid))**2
end
$end omp for
$end omp parallel
! calculate the full density
rho(:) = rho_t(:, 1) + rho_t(:, 2) + ...
```

Charge density construction

$$\tilde{\psi}_j(\mathbf{G}) \xrightarrow{FFT^{-1}} \psi_j(\mathbf{r}) \rightarrow \rho(\mathbf{r}) = \sum_j |\psi_j(\mathbf{r})|^2$$

version #3 (with GPU)

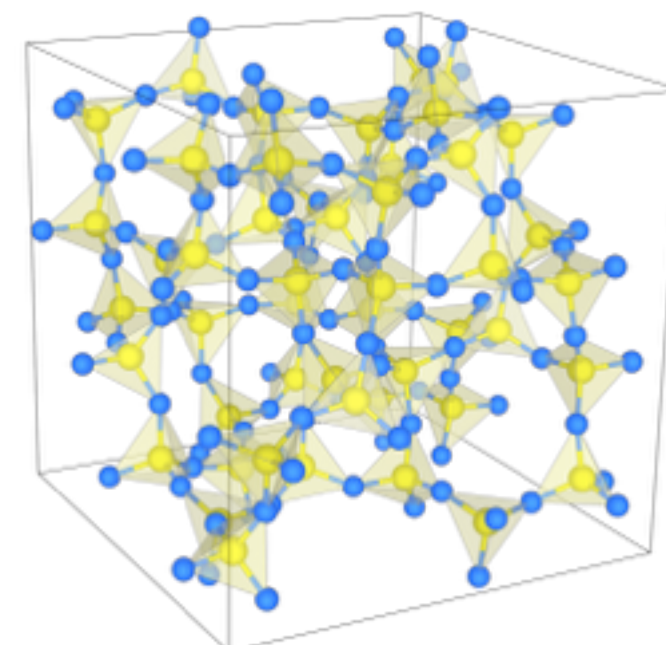
```
! control thread's job
while not done
   ! copy pw coefficients to device
   copy_to_gpu(psi(:,j+1:j+n), d_psi(:,1:n))
   ! load pw coefficients into proper
   ! positions inside fft buffer
   load_fft_buf(d_psi, d_buf, d_map)
   ! execute batch cuFFT for n bands
   inverse_cufft(d_buf)
   ! add contribution from n bands to density
   add_to_density(d_buf, d_rho)
end
! copy gpu part of density to host
copy_to_host(d_rho, rho)
! calculate the full density
rho(:) += rho_t(:, 2) + rho_t(:, 3) + ...
```



Charge density augmentation

Valence charge density has to be augmented with:

$$\tilde{\rho}(\mathbf{G}) = \sum_{\alpha} \sum_{\xi\xi'} q^{\alpha}_{\xi\xi'} Q^{\alpha}_{\xi'\xi}(\mathbf{G})$$



Relation between plane-wave coefficients of the Q-operator for a given atom α and the corresponding coefficients of the Q-operator for a given atom type A:

$$Q^{\alpha(A)}_{\xi'\xi}(\mathbf{G}) = e^{-i\mathbf{G}\tau_{\alpha(A)}} Q^{A}_{\xi'\xi}(\mathbf{G})$$

$$\tilde{\rho}(\mathbf{G}) = \sum_{A} \sum_{\xi\xi'} Q^{A}_{\xi'\xi}(\mathbf{G}) \sum_{\alpha(A)} \underbrace{\boxed{q^{\alpha(A)}_{\xi\xi'}} \boxed{e^{-i\mathbf{G}\tau_{\alpha(A)}}}}_{\textbf{zgemm}} = \sum_{A} \sum_{\xi\xi'} Q^{A}_{\xi'\xi}(\mathbf{G}) d^{A}_{\xi\xi'}(\mathbf{G})$$

Charge density augmentation

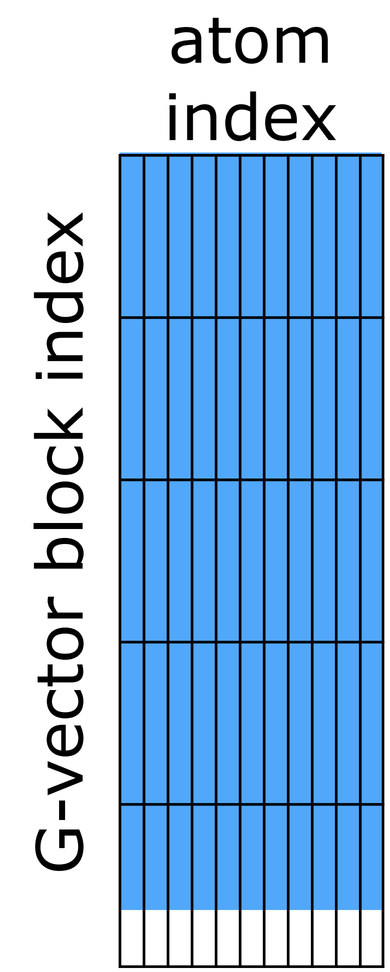
$$\tilde{\rho}(\mathbf{G}) = \sum_{A} \sum_{\xi\xi'} Q^{A}_{\xi'\xi}(\mathbf{G}) \sum_{\alpha(A)} \boxed{q^{\alpha(A)}_{\xi\xi'}} \boxed{e^{-i\mathbf{G}\tau_{\alpha(A)}}} = \sum_{A} \sum_{\xi\xi'} Q^{A}_{\xi'\xi}(\mathbf{G}) d^{A}_{\xi\xi'}(\mathbf{G})$$

```
! copy G-vectors to device
copy_to_gpu(gvec, d_gvec)
! loop over atom types
for iat in [1:natom_types]
  ! number of atoms of this type
  na = num_atoms(iat)
  ! copy Q-coeffs to device
  copy_to_gpu(Q(:, iat), d_Q)
  ! copy density matrix do device
  copy_to_gpu(q(:, 1:na, iat), d_q)
  ! copy atomic positions to device
  copy_to_gpu(atom_pos(:, 1:na, iat), d_atom_pos)
  ! generate d-matrix
  generate_dmtrx_pw_gpu(d_gvec, d_atom_pos, d_q, d_dmtrx)
  ! accumulate augmentation charge
  sum_q_pw_dmtrx_pw_gpu(d_dmtrx, d_Q, d_rho)
end
```

Charge density augmentation

$$d^{A}_{\xi\xi'}(\mathbf{G}) = \sum_{\alpha(A)} q^{\alpha(A)}_{\xi\xi'} e^{-i\mathbf{G}\tau_{\alpha(A)}}$$

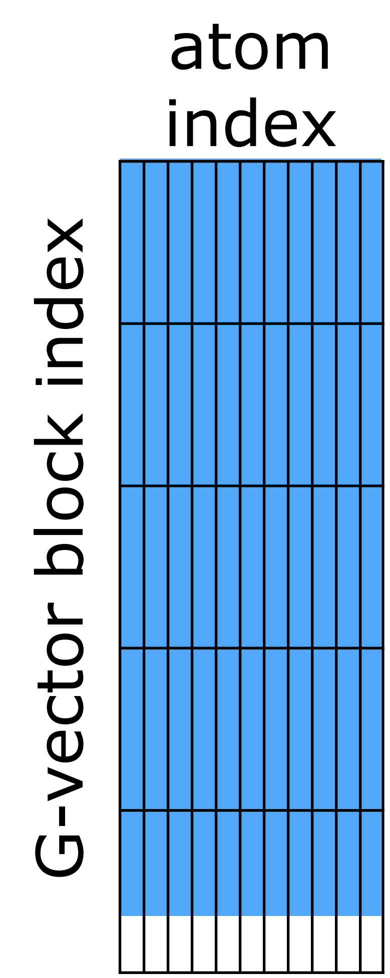
```
extern "C" void generate_dmtrx_pw_gpu(int num_atoms,
                                      int num_gvec_loc,
                                      int num_beta,
                                      double* atom_pos,
                                      int* gvec,
                                      cuDoubleComplex* qmtrx,
                                      cuDoubleComplex* dmtrx_pw)
{
    cuDoubleComplex* phase_factors;
    phase_factors = (cuDoubleComplex*)cuda_malloc(num_gvec_loc * num_atoms * sizeof (cuDoubleComplex));

    dim3 grid_t(64);
    dim3 grid_b(num_blocks(num_gvec_loc, grid_t.x), num_atoms);

    generate_phase_factors_conj_gpu_kernel <<<grid_b, grid_t>>>
    (
        num_gvec_loc,
        num_atoms,
        atom_pos,
        gvec,
        phase_factors
    );

    cuDoubleComplex zone = make_cuDoubleComplex(1.0, 0.0);
    cuDoubleComplex zzero = make_cuDoubleComplex(0.0, 0.0);

    cublas_zgemm(0, 1, num_gvec_loc, num_beta * num_beta, num_atoms, &zone,
                 phase_factors, num_gvec_loc, qmtrx, num_beta * num_beta, &zzero,
                 dmtrx_pw, num_gvec_loc, -1);

    cuda_free(phase_factors);
}
```



Charge density augmentation

$$e^{-i\mathbf{G}\tau_{\alpha(A)}}$$

```
__global__ void generate_phase_factors_conj_gpu_kernel
(
    int num_gvec_loc,
    int num_atoms,
    double* atom_pos,
    int* gvec,
    cuDoubleComplex* phase_factors
)
{
    int ia = blockIdx.y;
    int igloc = blockIdx.x * blockDim.x + threadIdx.x;

    if (igloc < num_gvec_loc)
    {
        int gvx = gvec[array2D_offset(0, igloc, 3)];
        int gvy = gvec[array2D_offset(1, igloc, 3)];
        int gvz = gvec[array2D_offset(2, igloc, 3)];

        double ax = atom_pos[array2D_offset(ia, 0, num_atoms)];
        double ay = atom_pos[array2D_offset(ia, 1, num_atoms)];
        double az = atom_pos[array2D_offset(ia, 2, num_atoms)];

        double p = twopi * (ax * gvx + ay * gvy + az * gvz);

        double sinp = sin(p);
        double cosp = cos(p);

        phase_factors[array2D_offset(igloc, ia, num_gvec_loc)] = make_cuDoubleComplex(cosp, -sinp);
    }
}
```



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Charge density augmentation

$$\tilde{\rho}(\mathbf{G}) = \tilde{\rho}(\mathbf{G}) + \sum_{\xi\xi'} Q^A_{\xi\xi'}(\mathbf{G}) d^A_{\xi\xi'}(\mathbf{G})$$

```
__global__ void sum_q_pw_d_mtrx_pw_gpu_kernel
(
    int num_gvec_loc,
    int num_beta,
    cuDoubleComplex* q_pw_t,
    cuDoubleComplex* d_mtrx_pw,
    cuDoubleComplex* rho_pw
)
{
    int igloc = blockIdx.x * blockDim.x + threadIdx.x;
    if (igloc < num_gvec_loc)
    {
        cuDoubleComplex zval = make_cuDoubleComplex(0.0, 0.0);

        // \sum_{xi1, xi2} D_{xi2,xi1} * Q(G)_{xi1, xi2}
        for (int xi2 = 0; xi2 < num_beta; xi2++)
        {
            int idx12 = xi2 * (xi2 + 1) / 2;

            // add diagonal term
            zval = cuCadd(zval, cuCmul(d_mtrx_pw[array2D_offset(igloc, xi2 * num_beta + xi2, num_gvec_loc)],
                                       q_pw_t[array2D_offset(igloc, idx12 + xi2, num_gvec_loc)]));

            // add non-diagonal terms
            for (int xi1 = 0; xi1 < xi2; xi1++, idx12++)
            {
                cuDoubleComplex q = q_pw_t[array2D_offset(igloc, idx12, num_gvec_loc)];
                cuDoubleComplex d1 = d_mtrx_pw[array2D_offset(igloc, xi2 * num_beta + xi1, num_gvec_loc)];
                cuDoubleComplex d2 = d_mtrx_pw[array2D_offset(igloc, xi1 * num_beta + xi2, num_gvec_loc)];

                zval = cuCadd(zval, cuCmul(q, d1));
                zval = cuCadd(zval, cuCmul(cuConj(q), d2));
            }
        }
        rho_pw[igloc] = cuCadd(rho_pw[igloc], zval);
    }
}
```

D-operator evaluation

We need to calculate new D-operator from a new potential:

$$D_{\xi\xi'}^{\alpha} = \sum_{\mathbf{G}} Q_{\xi\xi'}^{\alpha}(\mathbf{G}) V(\mathbf{G})$$

Use definition of Q-operator for an atom:

$$D_{\xi\xi'}^{\alpha} = \sum_{\mathbf{G}} \boxed{Q_{\xi\xi'}^{A}(\mathbf{G})} \boxed{e^{-i\mathbf{G}\tau_{\alpha(A)}} V(\mathbf{G})}$$

zgemm

D-operator evaluation

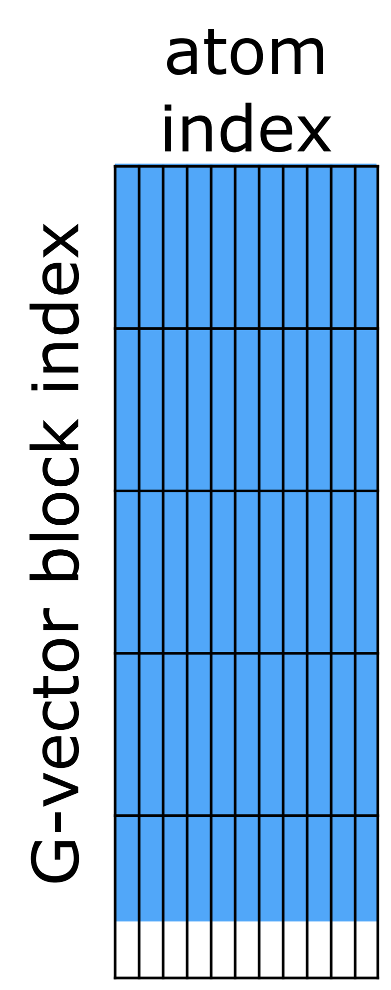
$$D^{\alpha}_{\xi\xi'} = \sum_{\mathbf{G}} Q^{A}_{\xi\xi'}(\mathbf{G}) e^{-i\mathbf{G}\tau_{\alpha(A)}} V(\mathbf{G})$$

```
! copy G-vectors to device
copy_to_gpu(gvec, d_gvec)
! copy effective potential to device
copy_to_gpu(veff, d_veff)
! loop over atom types
for iat in [1:natom_types]
  ! number of atoms of this type
  na = num_atoms(iat)
  ! copy atomic positions to device
  copy_to_gpu(atom_pos(:, 1:na, iat), d_atom_pos)
  ! copy Q-coeffs to device
  copy_to_gpu_async(Q(:, iat), d_Q)
  ! create a matrix from phase-factors and veff(G)
  mul_veff_with_phase_factors_gpu(d_gvec, d_atom_pos, d_veff, d_veff_pw)
  ! do a matrix-matrix multiplication on a GPU
  cuda_zgemm(d_Q, d_veff_pw, d_D_op)
  ! copy result to the host
  copy_to_host(d_D_op, D_op(:, 1:na, iat))
end
```

D-operator evaluation

$$e^{-i\mathbf{G}\tau_{\alpha(A)}} V(\mathbf{G})$$

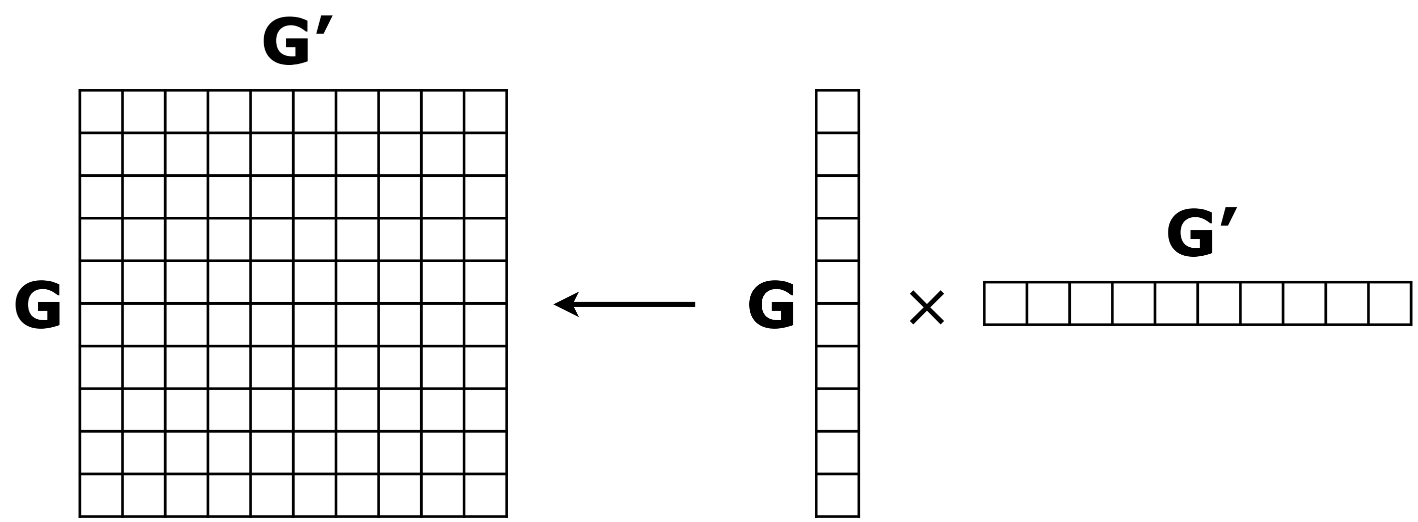
```
__global__ void mul_veff_with_phase_factors_gpu_kernel(int num_gvec_loc__,
                                                       cuDoubleComplex const* veff__,
                                                       int const* gvec__,
                                                       double const* atom_pos__,
                                                       cuDoubleComplex* veff_pw__)
{
    int igloc = blockDim.x * blockIdx.x + threadIdx.x;
    int ia = blockIdx.y;

    if (igloc < num_gvec_loc__)
    {
        int gvx = gvec__[array2D_offset(0, igloc, 3)];
        int gvy = gvec__[array2D_offset(1, igloc, 3)];
        int gvz = gvec__[array2D_offset(2, igloc, 3)];
        double ax = atom_pos__[array2D_offset(0, ia, 3)];
        double ay = atom_pos__[array2D_offset(1, ia, 3)];
        double az = atom_pos__[array2D_offset(2, ia, 3)];

        double p = twopi * (ax * gvx + ay * gvy + az * gvz);

        veff_pw__[array2D_offset(igloc, ia, num_gvec_loc__)] =
            cuCmul(veff__[igloc], make_cuDoubleComplex(cos(p), -sin(p)));
    }
}
```



Overlap matrix construction

Overlap matrix of the LAPW basis:

$$O^{\mathbf{k}}_{\mathbf{G}\mathbf{G}'} = \sum_{\alpha} \sum_{\ell m} \sum_{\nu=1}^{\kappa_\ell^\alpha} A^{\mathbf{k}*}_{\alpha\ell m\nu}(\mathbf{G}) A^{\mathbf{k}}_{\alpha\ell m\nu}(\mathbf{G}') + \Theta(\mathbf{G}-\mathbf{G}')$$



version #1

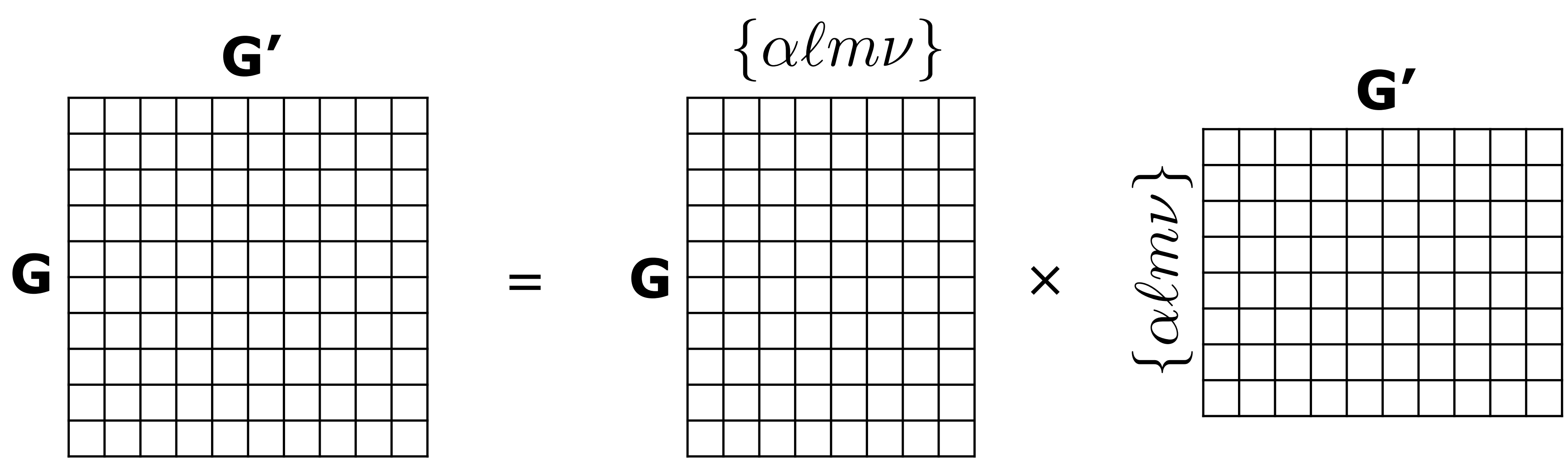
```
for ia in [1:num_atoms]
  for l in [0:lmax]
    for m in [-l:l]
      for io in [1:apw_ord(l, ia)]
        $omp parallel for
        for ig in [1:num_gvec]
          O(:, ig) += conj(A(:, io, m, l, ia)) * A(ig, io, m, l, ia)
        end
      end
    end
  end
end
```

Overlap matrix construction

$$\sum_{\alpha} \sum_{\ell m} \sum_{\nu=1}^{\kappa_\ell^\alpha} A^{\mathbf{k}*}_{\alpha\ell m\nu}(\mathbf{G}) A^{\mathbf{k}}_{\alpha\ell m\nu}(\mathbf{G}') \rightarrow \sum_{\mu} A^{\mathbf{k}*}_{\mu}(\mathbf{G}) A^{\mathbf{k}}_{\mu}(\mathbf{G}') \text{ where } \mu = \{\alpha\ell m\nu\}$$

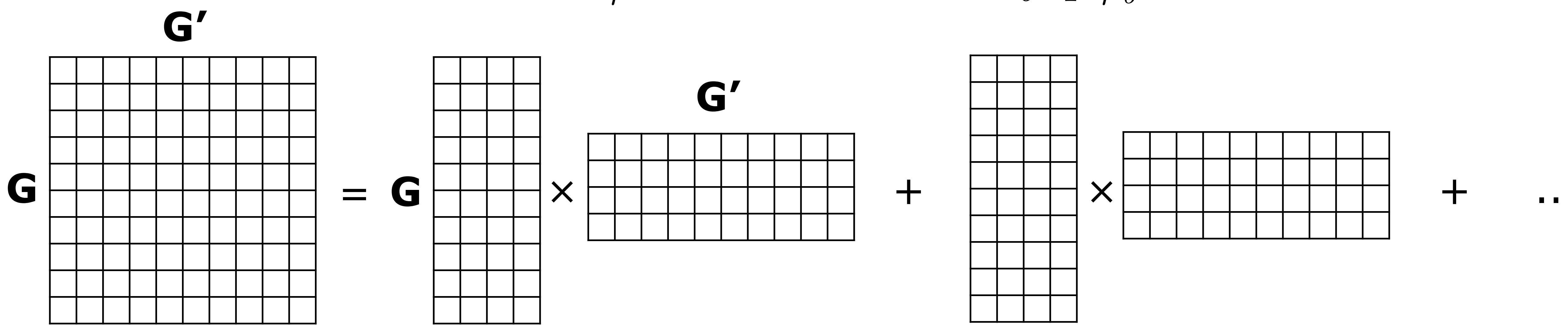
version #2

```
! use matrix-matrix multiplication to setup O
zgemm(conj(A), A, O)
```

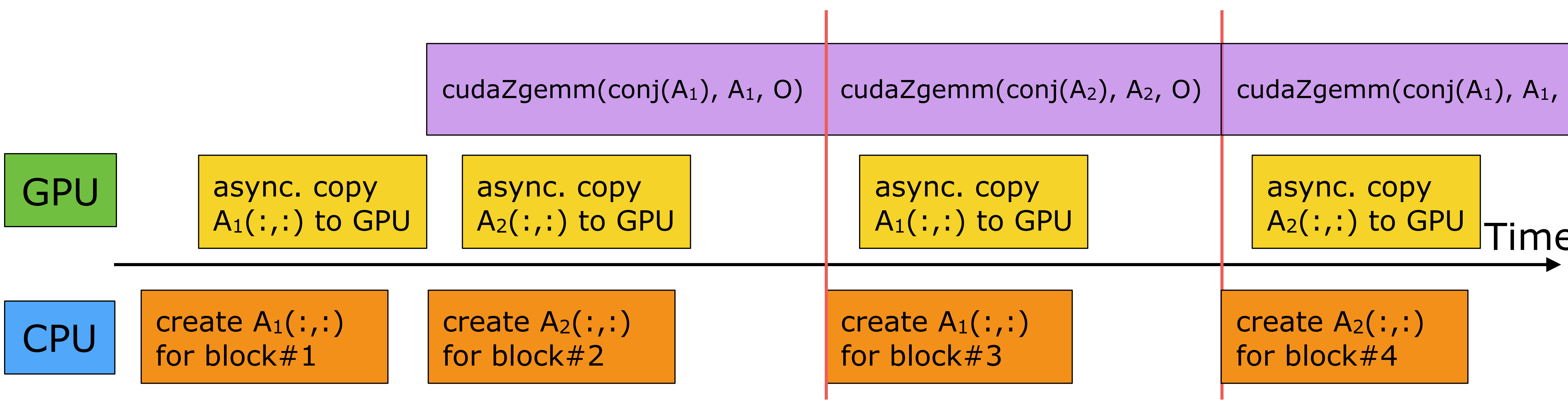


Overlap matrix construction

Block zgemm over atoms $\sum_{\mu} A_{\mu}^{\mathbf{k}*}(\mathbf{G}) A_{\mu}^{\mathbf{k}}(\mathbf{G}') \rightarrow \sum_{b=1}^{N_b} \sum_{\mu_b} A_{\mu_b}^{\mathbf{k}*}(\mathbf{G}) A_{\mu_b}^{\mathbf{k}}(\mathbf{G}')$



version #3 (with GPU)



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Thank you for your attention.