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 to understand the compute-intensive kernels of the application)
- move compute-intensive kernels to GPU
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?

Low level libraries

Exciting

Elk

LAPACK and BLAS
ScalAPACK and PBLAS
FFT

LibXC
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**

### SIRIUS C++ library
- MPI + OpenMP parallel model with GPU acceleration
- LAPW specific functionality
- Pseudopotential specific functionality
- Common objects: unit cell description, reciprocal lattice description, FFT mesh, G-vector indexing, radial functions (local orbitals or beta projectors) indexing, XC potential generation, etc.

### Low level libraries
- GNU scientific library
- HDF5
- spglib
- LibXC
- LAPACK and BLAS
- ScaLAPACK and PBLAS
- FFTW
- ELPA
- MAGMA
Kohn-Sham equations of DFT

Eigen-value problem

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

Effective potential construction

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

Density generation

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

Density mixing

\[
\rho(\mathbf{r}) = \alpha \rho^{new}(\mathbf{r}) + (1 - \alpha) \rho^{old}(\mathbf{r})
\]
Charge density construction

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

\[
\psi_j(r) = \sum_G e^{iG \cdot r} \tilde{\psi}_j(G)
\]

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

\[
\tilde{\psi}_j(G) \xrightarrow{FFT^{-1}} \psi_j(r) \rightarrow \rho(r) = \sum_j |\psi_j(r)|^2
\]
Charge density construction

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

version #1

\texttt{for } j \texttt{ in } [1:\text{num}_\text{psi}]
\hspace{1em} ! \text{transform psi to real space}
\hspace{1em} \text{inverse}_\text{fft}(\text{psi}(:,j), \text{psi}_\text{of}_r(:))
\hspace{1em} ! \text{accumulate in density}
\hspace{1em} \text{rho}(:) += \text{abs}(\text{psi}_\text{of}_r(:))**2
\texttt{end}
Charge density construction

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

version #2

```cpp
$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

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

version #3 (with GPU)

! control thread’s job

while not done

! copy pw coefficients to device

\[
\text{copy_to_gpu(psi(:,j+1:j+n), d_psi(:,1:n))}
\]

! load pw coefficients into proper positions inside fft buffer

\[
\text{load_fft_buf(d_psi, d_buf, d_map)}
\]

! execute batch cuFFT for n bands

\[
\text{inverse_cufft(d_buf)}
\]

! add contribution from n bands to density

\[
\text{add_to_density(d_buf, d_rho)}
\]

end

! copy gpu part of density to host

\[
\text{copy_to_host(d_rho, rho)}
\]

! calculate the full density

\[
rho(:) += \text{rho}_t(:, 2) + \text{rho}_t(:, 3) + \ldots
\]
Charge density augmentation

Valence charge density has to be augmented with:

$$\tilde{\rho}(G) = \sum_{\alpha} \sum_{\xi \xi'} q_{\xi \xi'}^\alpha Q_{\xi \xi'}^\alpha (G)$$

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

$$Q_{\xi' \xi}^{\alpha(A)} (G) = e^{-iG_{\tau(\alpha)}(A)} Q_{\xi' \xi}^A (G)$$

$$\tilde{\rho}(G) = \sum_A \sum_{\xi \xi'} Q_{\xi' \xi}^A (G) \sum_{\alpha(A)} q_{\xi \xi'}^{\alpha(A)} e^{-iG_{\tau(\alpha)}(A)} = \sum_A \sum_{\xi \xi'} Q_{\xi' \xi}^A (G) d_{\xi' \xi}^A (G)$$

zgemm
Charge density augmentation

\[ \tilde{\rho}(G) = \sum_A \sum_{\xi \xi'} Q^A_{\xi \xi'}(G) \sum_\alpha(A) q^\alpha(A) e^{-iG\tau_\alpha(A)} = \sum_A \sum_{\xi \xi'} Q^A_{\xi \xi'}(G) d^A_{\xi \xi'}(G) \]

! 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
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

\[ e^{-iG\tau_\alpha(A)} \]

```c
__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);
    }
}
```
Charge density augmentation

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

```c
__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 + xi1, 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(cuConj(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:

\[ D_{\xi \xi'}^\alpha = \sum_G Q_{\xi \xi'}^\alpha(G) V(G) \]

Use definition of Q-operator for an atom:

\[ D_{\xi \xi'}^\alpha = \sum_G Q_{\xi \xi'}^A(G) e^{-iG\tau_{\alpha}(A)} V(G) \]

*zgemm*
D-operator evaluation

\[ D_{\xi \xi'}^\alpha = \sum_G Q^A_{\xi \xi'}(G) e^{-iG \tau_\alpha(A)} V(G) \]

! 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

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

```c
__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:

\[
O_{GG'}^k = \sum_\alpha \sum_\ell \sum_m \sum_\nu=1 A_{\alpha \ell m \nu}^k(G) A_{\alpha \ell m \nu}^k(G') + \Theta(G - G')
\]

version #1

```python
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
```
Overlap matrix construction

\[
\sum_{\alpha} \sum_{\ell m} \sum_{\nu=1}^{K^{\alpha}_{\ell}} A^k_{\alpha \ell m \nu}(G) A^k_{\alpha \ell m \nu}(G') \rightarrow \sum_{\mu} A^k_{\mu} (G) A^k_{\mu} (G') \text{ where } \mu = \{\alpha \ell m \nu\}
\]

version #2

use matrix-matrix multiplication to setup O
\[
zgemm(\text{conj}(A), A, O)
\]
Overlap matrix construction

Block zgemm over atoms

\[\sum_{\mu} A_{\mu}^{k^*}(G)A_{\mu}^{k}(G') \rightarrow \sum_{b=1}^{N_b} \sum_{\mu_b} A_{\mu_b}^{k^*}(G)A_{\mu_b}^{k}(G')\]

version #3 (with GPU)

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>create A(_1)(::,:) for block#1</td>
<td>async. copy A(_1)(::,:) to GPU</td>
</tr>
<tr>
<td>create A(_2)(::,:) for block#2</td>
<td>async. copy A(_2)(::,:) to GPU</td>
</tr>
<tr>
<td>create A(_1)(::,:) for block#3</td>
<td>async. copy A(_1)(::,:) to GPU</td>
</tr>
<tr>
<td>create A(_2)(::,:) for block#4</td>
<td>async. copy A(_2)(::,:) to GPU</td>
</tr>
</tbody>
</table>

Time
Thank you for your attention.