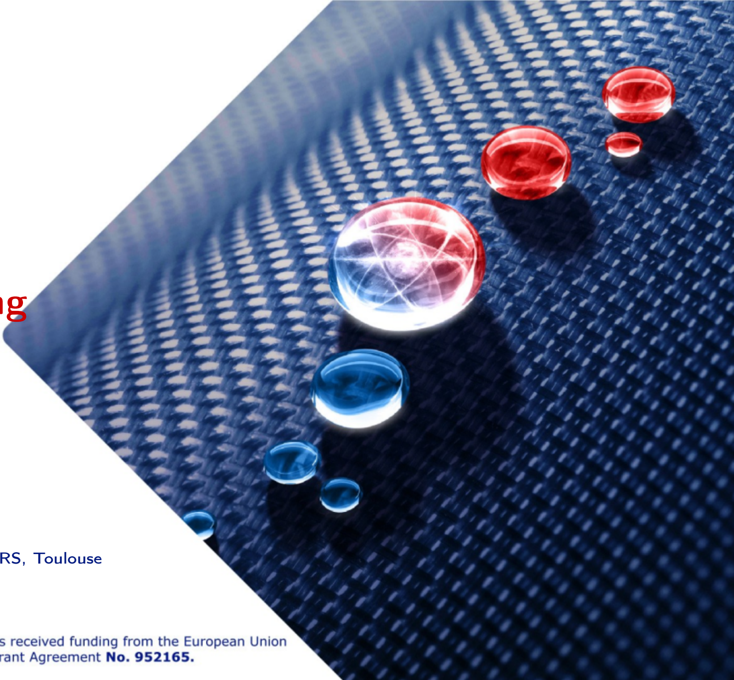
Targeting Real chemical accuracy at the EXascale

QMCkl: A Unified Approach to Accelerating Quantum Monte Carlo Codes

Anthony Scemama

5/02/2024

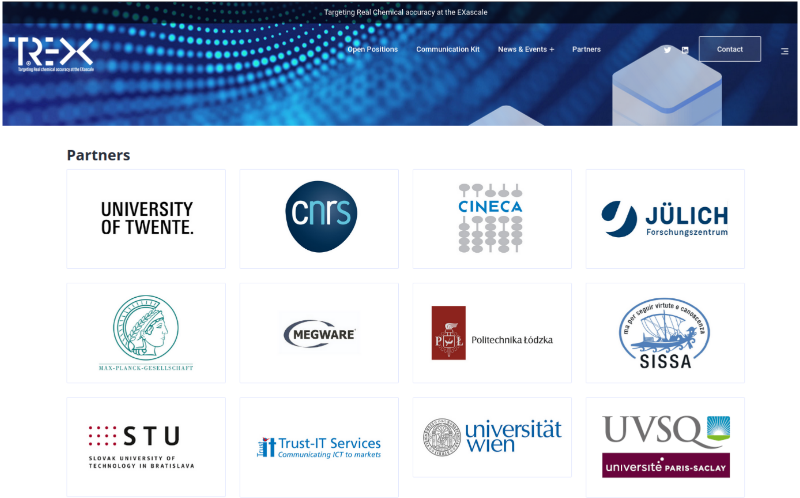
Lab. Chimie et Physique Quantiques, FERMI, UPS/CNRS, Toulouse (France)



Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizoon 2020 research and innovation programme under Grant Agreement **No. 952165.**

The TREX European Center of Excellence

The TREX CoE



Codes

- CHAMP
- QMC=Chem
- TurboRVB
- NECI
- Quantum Package
- GammCor

CHAMP (Claudia Filippi)

- Wave function optimization: Jastrow, CI, MOs
- Ground/Excited states
- Geometry optimization

CHAMP (Claudia Filippi)

- Wave function optimization: Jastrow, CI, MOs
- Ground/Excited states
- Geometry optimization

TurboRVB (Sandro Sorella + Michele Casula)

- Molecular and Periodic systems
- JAGP, Pfaffian, . . .
- LRDMC

CHAMP (Claudia Filippi)

- Wave function optimization: Jastrow, CI, MOs
- Ground/Excited states
- Geometry optimization

TurboRVB (Sandro Sorella + Michele Casula)

- Molecular and Periodic systems
- JAGP, Pfaffian, ...
- LRDMC

QMC=Chem (Michel Caffarel + Me!)

- DMC as "Post-Full-CI" energy calculations (CIPSI)
- Very large CI expansions (millions of determinants)
- Designed with HPC in mind
- Highly optimized with W. Jalby's group (UVSQ) in 2011-2013

- TREX CoE: Targeting REal chemical accuracy at the eXascale
- Started in Oct. 2020
- Objective: Make codes ready for exascale systems

- TREX CoE: Targeting REal chemical accuracy at the eXascale
- Started in Oct. 2020
- Objective: Make codes ready for exascale systems
- How: Instead of re-writing codes, provide libraries
  - One library for high-performance (QMCkl)
  - One library for exchanging information between codes (TREXIO)

The QMC kernel library (QMCkl)

- Progress in quantum chemistry requires codes with new ideas/algorithms
- New ideas/algorithms are implemented by physicists/chemists
- Different scientists have different programming language knowledge/preference
- Exascale machines are horribly complex to program

- Progress in quantum chemistry requires codes with new ideas/algorithms
- New ideas/algorithms are implemented by physicists/chemists
- Different scientists have different programming language knowledge/preference
- Exascale machines are horribly complex to program

Question

Is it reasonable to ask physicists/chemists to write codes for exascale machines?

No: Proof

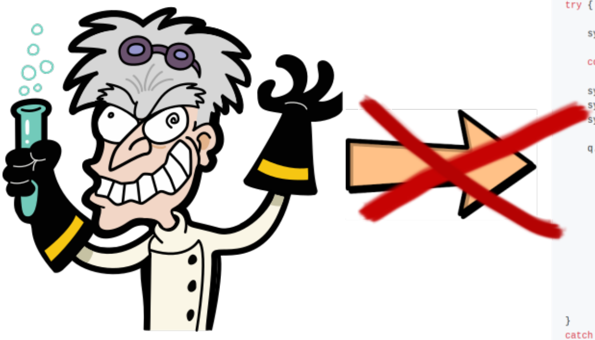
(from https://github.com/jeffhammond/dpcpp-tutorial)

$$Z_{n+1} = Z_n + aX_n + Y_n$$

```
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

```
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

No



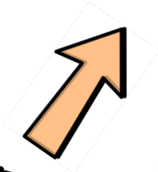
```cpp
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

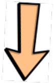
A compiler[1] that can read an average researcher's code and transform it into highly efficient code on an exascale machine.

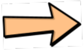
[1] Wikipedia: A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language)

Artificial Intelligence was not ready in 2021 when we started the project . . .



```fortran
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

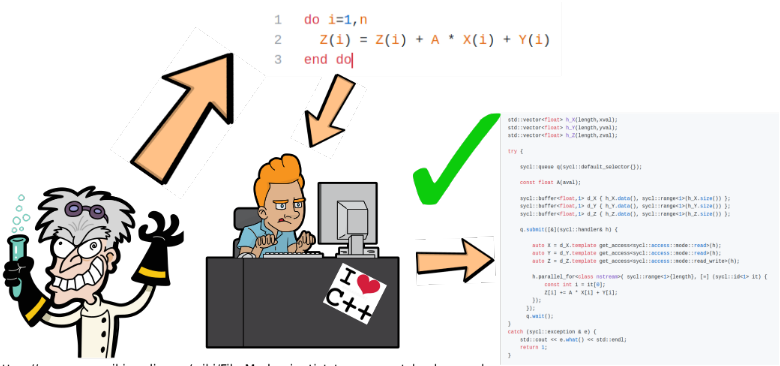




```cpp
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

try {

    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
    sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
    sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };

    q.submit([&](sycl::handler& h) {

        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>( sycl::range<1>{length}, [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception & e) {
    std::cout << e.what() << std::endl;
    return 1;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

. . . so we decided to use *Natural Intelligence*, and add a human layer between the machine and the researchers : a biological compiler



https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

- Identify the common computational kernels of QMC
- Implement these kernels in a human-readable library (QMC experts)
- *Bio-compile* the human-readable library in a HPC-library (HPC experts)
- Scientists can link either library with their codes

For scientists

- The choice of the programming language is not imposed to the scientist
- The code can stay easy to understand by the physicists/chemists
  Performance-related aspects are delegated to the library
- Codes will not die with a change in hardware
- Scientific code development does not break the performance
- Scientists don't lose control on their codes

For scientists

- The choice of the programming language is not imposed to the scientist
- The code can stay easy to understand by the physicists/chemists
  Performance-related aspects are delegated to the library
- Codes will not die with a change in hardware
- Scientific code development does not break the performance
- Scientists don't lose control on their codes

Separation of concerns

- Scientists will never have to manipulate low-level HPC code
- HPC experts will not be required to be experts in theoretical physics
- Better re-use of the optimization effort among the community

The QMCkl Documentation library

QMCkl documentation library

- The API is C-compatible: QMCkl appears to scientists like a C library $\Longrightarrow$ can be used in all other languages
- System functions in programmed C (memory allocation, thread safety, *etc*)
- Computational kernels programmed in simple Fortran for readability
- A lot of documentation (remember: the HPC compiler is a human!)

Literate programming is a programming paradigm introduced by Donald Knuth in which a computer program is given an explanation of its logic in a natural language, such as English, interspersed with snippets of macros and traditional source code, from which compilable source code can be generated. (Wikipedia)

Literate programming with *org-mode*:

- Here, comments are more important than code
- Can add graphics, LaTeX formulas, tables, *etc*
- Documentation always synchronized with the code
- Some functions can be generated by embedded scripts
- Web site auto-generated when code is pushed

Instead of writing comments documenting code, we write code illustrating documentation.

Literate programming with *org-mode*

File Edit Options Buffers Tools Table Org Text Help

```
#+TITLE: Atomic Orbitals
#+SETUPFILE: ../docs/theme.setup
#+INCLUDE: ../tools/lib.org
```

The atomic basis set is defined as a list of shells. Each shell s is centered on a nucleus A, possesses a given angular momentum l and a radial function R_s. The radial function is a linear combination of \emph{primitive} functions that can be of type Slater ($p = 1$) or Gaussian ($p = 2$):

$$R_s(\mathbf{r}) = \mathcal{N}_s |\mathbf{r} - \mathbf{R}_A|^{n_s} \sum_{k=1}^{N_{\text{prim}}} a_{ks} \exp\left(-\gamma_{ks} |\mathbf{r} - \mathbf{R}_A|^p\right).$$

In the case of Gaussian functions, n_s is always zero. The normalization factor $\mathcal{N}_s$ ensures that all the functions of the shell are normalized to unity. As this normalization requires the ability to compute overlap integrals, it should be written in the file to ensure that the file is self-contained and does not require the client program to have the ability to compute such integrals.

Atomic orbitals (AOs) are defined as

$$\chi_i(\mathbf{r}) = P_{\eta(i)}(\mathbf{r})\, R_{\theta(i)}(\mathbf{r})$$

where $\theta(i)$ returns the shell on which the AO is expanded, and $\eta(i)$ denotes which angular function is chosen.

In this section we describe the kernels used to compute the values, gradients and Laplacian of the atomic basis functions.

- Headers
- Context...
- Polynomial part...
- Radial part
  - Gaussian basis functions

~qmckl_ao_gaussian_vgl~ computes the values, gradients and Laplacians at a given point of ~n~ Gaussian functions centered at the same point:

U:@--- qmckl_ao.org Top (1459,0) <N> Git:context (Org ARev ² Undo-Tree Fill) Mail [1]

~context~	input	Global state
~X(3)~	input	Array containing the coordinates of the points
~R(3)~	input	Array containing the x,y,z coordinates of the center
~n~	input	Number of computed Gaussians
~A(n)~	input	Exponents of the Gaussians
~VGL(ldv,5)~	output	Value, gradients and Laplacian of the Gaussians
~ldv~	input	Leading dimension of array ~VGL~

Requirements :

- ~context~ is not 0
- ~n~ > 0
- ~ldv~ >= 5
- ~A(i)~ > 0 for all ~i~
- ~X~ is allocated with at least 3×8 bytes
- ~R~ is allocated with at least 3×8 bytes
- ~A~ is allocated with at least $n \times 8$ bytes
- ~VGL~ is allocated with at least $n \times 5 \times 8$ bytes

```c
#+begin_src c :tangle (eval h_func)
qmckl_exit_code
qmckl_ao_gaussian_vgl(const qmckl_context context,
                      const double *X,
                      const double *R,
                      const int64_t *n,
                      const int64_t *A,
                      const double *VGL,
                      const int64_t ldv);
#+end_src
```

```fortran
#+begin_src f90 :tangle (eval f)
integer function qmckl_ao_gaussian_vgl_f(context, X, R, n, A, VGL, ldv) result(info)
  use qmckl
  implicit none
  integer*8 , intent(in)  :: context
  real*8    , intent(in)  :: X(3), R(3)
  integer*8 , intent(in)  :: n
  real*8    , intent(in)  :: A(n)
  real*8    , intent(out) :: VGL(ldv,5)
  integer*8 , intent(in)  :: ldv

  integer*8        :: i,j
  real*8           :: Y(3), r2, t, u, v
```

U:@--- qmckl_ao.org 85% (1493,0) <N> Git:context (Org ARev ² Undo-Tree Fill) Mail [1]

Generated code

```
qmckl_ao_f.f90                  qmckl_numprec_fh_func.f90
qmckl_ao_fh_func.f90            qmckl_numprec_func.h
qmckl_ao_func.h                 qmckl_numprec.org
qmckl_ao.org                    qmckl_numprec_private_type.h
qmckl_ao_private_func.h         qmckl_numprec_type.h
qmckl_ao_private_type.h         qmckl.org
qmckl_context.c                 README.org
qmckl_context_fh_func.f90       table_of_contents
qmckl_context_fh_type.f90       test_qmckl
qmckl_context_func.h            test_qmckl_ao.c
qmckl_context.org               test_qmckl_ao_f.f90
qmckl_context_private_type.h    test_qmckl.c
qmckl_context_type.h            test_qmckl_context.c
qmckl_distance_f.f90            test_qmckl_distance.c
qmckl_distance_fh_func.f90      test_qmckl_distance_f.f90
qmckl_distance_func.h           test_qmckl_error.c
qmckl_distance.org              test_qmckl_memory.c
qmckl_error.c                   test_qmckl_numprec.c
qmckl_error_fh_func.f90         test_qmckl.org
(base) scemama@lpqdh82:~/TREX/qmckl/src$
```

```c
char* qmckl_get_ao_basis_shell_ang_mom (const qmckl_context context) {
  if (qmckl_context_check(context) == QMCKL_NULL_CONTEXT) {
    return NULL;
  }

  qmckl_context_struct* const ctx = (qmckl_context_struct* const) context;
  assert (ctx != NULL);

  int32_t mask = 1 << 4;

  if ( (ctx->ao_basis.uninitialized & mask) != 0) {
    return NULL;
  }

  assert (ctx->ao_basis.shell_ang_mom != NULL);
  return ctx->ao_basis.shell_ang_mom;
}
~/TREX/qmckl/src/qmckl_ao.c [unix] [C]  [ 15% ]  (104/674,16)
```

```c
#define  QMCKL_INVALID_CONTEXT          ((qmckl_exit_code) 103)
#define  QMCKL_ALLOCATION_FAILED        ((qmckl_exit_code) 104)
#define  QMCKL_DEALLOCATION_FAILED      ((qmckl_exit_code) 105)
#define  QMCKL_INVALID_EXIT_CODE        ((qmckl_exit_code) 106)
/* Context handling */

/*  The context variable is a handle for the state of the library, */
/*  and is stored in a data structure which can't be seen outside of */
/*  the library. To simplify compatibility with other languages, the */
/*  pointer to the internal data structure is converted into a 64-bit */
/*  signed integer, defined in the ~qmckl_context~ type. */
/*  A value of ~QMCKL_NULL_CONTEXT~ for the context is equivalent to a */
/*  ~NULL~ pointer. */

/*  #+NAME: qmckl_context */

typedef int64_t qmckl_context ;
#define QMCKL_NULL_CONTEXT (qmckl_context) 0
/* Decoding errors */

/*   To decode the error messages, ~qmckl_string_of_error~ converts an */
/*   error code into a string.  */

/*   #+NAME: MAX_STRING_LENGTH */
/*   : 128 */


const char* qmckl_string_of_error(const qmckl_exit_code error);
void qmckl_string_of_error_f(const qmckl_exit_code error,
                             char result[128]);

/* Updating errors in the context */

/*   The error is updated in the context using ~qmckl_set_error~. */
/*   When the error is set in the context, it is mandatory to specify */
/*   from which function the error is triggered, and a message */
/*   explaining the error. The exit code can't be ~QMCKL_SUCCESS~. */

/*   # Header */
~/TREX/qmckl/include/qmckl.h [unix] [CPP]  [ 31% ]  (85/269,1)
```

```
1 | 2 | 3 | Biblio | OrgWork | Web | eMail     us   +6°C   1.1K   565B   LinksysRouter   02%   49°C   /: 21G   100%   0   82%   04/19 10:21   19
```

Generated web site

Atomic Orbitals

The atomic basis set is defined as a list of shells. Each shell s is centered on a nucleus A, possesses a given angular momentum l and a radial function R_s. The radial function is a linear combination of \emph{primitive} functions that can be of type Slater ($p=1$) or Gaussian ($p=2$):

$$R_s(\mathbf{r}) = \mathcal{N}_s |\mathbf{r} - \mathbf{R}_A|^{n_s} \sum_{k=1}^{N_{\text{prim}}} a_{ks} \exp(-\gamma_{ks} |\mathbf{r} - \mathbf{R}_A|^p).$$

In the case of Gaussian functions, n_s is always zero. The normalization factor $\mathcal{N}_s$ ensures that all the functions of the shell are normalized to unity. As this normalization requires the ability to compute overlap integrals, it should be written in the file to ensure that the file is self-contained and does not require the client program to have the ability to compute such integrals.

Atomic orbitals (AOs) are defined as

$$\chi_i(\mathbf{r}) = P_{\eta(i)}(\mathbf{r})\, R_{\theta(i)}(\mathbf{r})$$

where $\theta(i)$ returns the shell on which the AO is expanded, and $\eta(i)$ denotes which angular function is chosen.

In this section we describe the kernels used to compute the values, gradients and Laplacian of the atomic basis functions.

1 Polynomial part

1.1 Powers of $x - X_i$

The `qmckl_ao_power` function computes all the powers of the `n` input data up to the given maximum value given in input for each of the n points:

$$\nabla_z v_i = -2 a_i (X_z - R_z) v_i$$

$$\Delta v_i = a_i (4 |X - R|^2 a_i - 6) v_i$$

`context`	input	Global state
`X(3)`	input	Array containing the coordinates of the points
`R(3)`	input	Array containing the x,y,z coordinates of the center
`n`	input	Number of computed Gaussians
`A(n)`	input	Exponents of the Gaussians
`VGL(ldv,5)`	output	Value, gradients and Laplacian of the Gaussians
`ldv`	input	Leading dimension of array `VGL`

Requirements :

- `context` is not 0
- `n` > 0
- `ldv` >= 5
- `A(i)` > 0 for all `i`
- `X` is allocated with at least 3×8 bytes
- `R` is allocated with at least 3×8 bytes
- `A` is allocated with at least $n \times 8$ bytes
- `VGL` is allocated with at least $n \times 5 \times 8$ bytes

```
qmckl_exit_code
qmckl_ao_gaussian_vgl(const qmckl_context context,
                      const double *X,
                      const double *R,
                      const int64_t *n,
                      const int64_t *A,
                      const double *VGL,
                      const int64_t ldv);
```

At each QMC step, we need to evaluate $E_{\text{loc}}(r_1, \ldots, r_N) = \frac{\hat{H}\Psi(r_1,\ldots,r_N)}{\Psi(r_1,\ldots,r_N)}$:

- $\Psi(r_1, \ldots, r_N)$
- $\Delta_i \Psi(r_1, \ldots, r_i, \ldots, r_N)$: kinetic energy
- $\vec{\nabla}_i \Psi(r_1, \ldots, r_i, \ldots, r_N)$: drift in the stochastic process

At each QMC step, we need to evaluate $E_{\text{loc}}(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \frac{\hat{H}\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N)}{\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N)}$:

- $\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)$
- $\Delta_i \Psi(\mathbf{r}_1, \ldots, \mathbf{r}_i, \ldots, \mathbf{r}_N)$: kinetic energy
- $\vec{\nabla}_i \Psi(\mathbf{r}_1, \ldots, \mathbf{r}_i, \ldots, \mathbf{r}_N)$: drift in the stochastic process

Kernels implemented and well tested today

- AOs: $\chi(\mathbf{r}), \vec{\nabla}\chi(\mathbf{r}), \Delta\chi(\mathbf{r})$
- MOs: $\phi(\mathbf{r}), \vec{\nabla}\phi(\mathbf{r}), \Delta\phi(\mathbf{r})$
- Jastrow correlation factor (eN, ee, eeN)
- Inverses of small matrices

At each QMC step, we need to evaluate $E_{\text{loc}}(\mathbf{r}_1, \dots, \mathbf{r}_N) = \frac{\hat{H}\Psi(\mathbf{r}_1,\dots,\mathbf{r}_N)}{\Psi(\mathbf{r}_1,\dots,\mathbf{r}_N)}$:

- $\Psi(\mathbf{r}_1, \dots, \mathbf{r}_N)$
- $\Delta_i \Psi(\mathbf{r}_1, \dots, \mathbf{r}_i, \dots, \mathbf{r}_N)$: kinetic energy
- $\vec{\nabla}_i \Psi(\mathbf{r}_1, \dots, \mathbf{r}_i, \dots, \mathbf{r}_N)$: drift in the stochastic process

Kernels implemented and well tested today

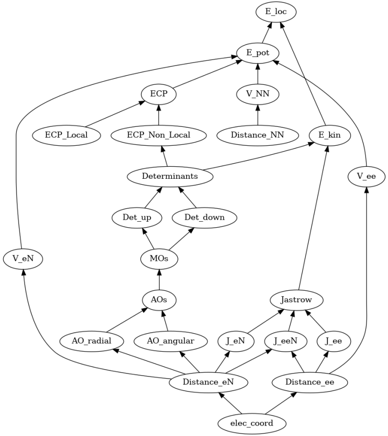
- AOs: $\chi(\mathbf{r}), \vec{\nabla}\chi(\mathbf{r}), \Delta\chi(\mathbf{r})$
- MOs: $\phi(\mathbf{r}), \vec{\nabla}\phi(\mathbf{r}), \Delta\phi(\mathbf{r})$
- Jastrow correlation factor (eN, ee, eeN)
- Inverses of small matrices

Work in progress

- Everything else required to compute Ψ, $\nabla\Psi$ and $\Delta\Psi$.

Dependencies between kernels

$$
\begin{aligned}
E_{\text{loc}}(\mathrm{R}) &= E_{\text{pot}}(\mathrm{R}) + E_{\text{kin}}(\mathrm{R}) \\
E_{\text{pot}}(\mathrm{R}) &= V_{ee}(\mathrm{R}) + V_{eN}(\mathrm{R}) + \\
& \quad V_{NN}(\mathrm{R}) + V_{\text{ECP}}(\mathrm{R}) \\
E_{\text{kin}}(\mathrm{R}) &= -\frac{1}{2}\frac{\Delta\Psi(\mathrm{R})}{\Psi(\mathrm{R})} \\
\Psi(\mathrm{R}) &= \Phi(\mathrm{R})J(\mathrm{R}) \\
\dots
\end{aligned}
$$



All the graph is invalidated updated when the electron coordinates are changed.

Algorithms

Before computing anything, QMCkl needs to be given a trial wave function.

Setting wave function parameters

- Wave function exchange between codes is a major difficulty
- Our solution:
  - Define a standard format for wavefunction parameters
  - TREXIO: TREX Input/Output library (see Evgeny Posenitskiy's presentation)

Initialization of QMCkl

Two ways:

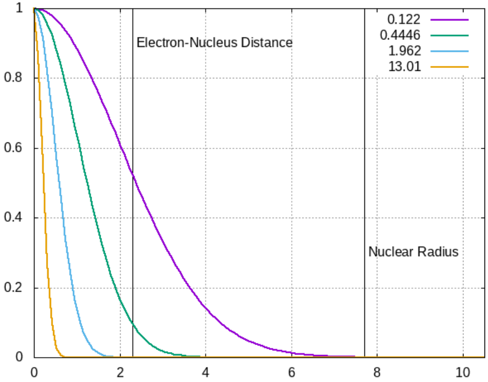
1. Control : Each array can be set by hand
2. Simplicity : Read all the wave function parameters from a TREXIO file

Atomic Orbitals

$$R_s(\mathbf{r}) = \mathcal{N}_s |\mathbf{r} - \mathbf{R}_A|^{n_s} \sum_{k=1}^{N_\text{prim}} a_{ks}\, f_{ks} \exp\left(-\gamma_{ks} |\mathbf{r} - \mathbf{R}_A|^p\right).$$

Flexible

- Software like GAMESS use different normalization factors for *d* orbitals
- Implementing Slater-type orbitals is a minor modification (in the very long to-do list)
- Contribution from the FHI-AIMS group for the evaluation of numerical AOs
- Separation of the radial and angular components packed in shells
- Efficient computation of powers of x, y, z to maximize data re-use



- Definition of an atomic radius for each nucleus beyond which all AOs are zero (VGL[a]).
- Primitives are sorted in ascending order of the exponents.
- Only non-zero elements are computed

[a] VGL: value, gradients, Laplacian

Molecular Orbitals

$$\phi_i(\mathrm{r}_j) = \sum_k A_{ik}\chi_k(\mathrm{r}_j) \qquad B_1 = A \cdot C_1$$

$$\nabla_x\phi_i(\mathrm{r}_j) = \sum_k A_{ik}\nabla_x\chi_k(\mathrm{r}_j) \qquad B_2 = A \cdot C_2$$

$$\nabla_y\phi_i(\mathrm{r}_j) = \sum_k A_{ik}\nabla_y\chi_k(\mathrm{r}_j) \qquad B_3 = A \cdot C_3$$

$$\nabla_z\phi_i(\mathrm{r}_j) = \sum_k A_{ik}\nabla_z\chi_k(\mathrm{r}_j) \qquad B_4 = A \cdot C_4$$

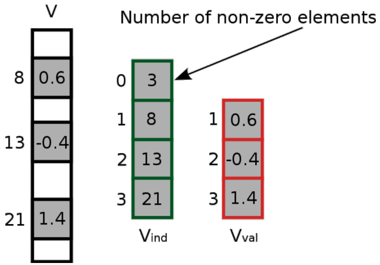
$$\Delta\phi_i(\mathrm{r}_j) = \sum_k A_{ik}\Delta\chi_k(\mathrm{r}_j) \qquad B_5 = A \cdot C_5$$

- QMC=Chem (2013): `https://doi.org/10.1002/jcc.23216`
- Exploits the common sparse character of the AO matrices:
  - When $\chi(\mathrm{r}) = 0$ because r is too far, all the derivatives are also zero
  - Quadratic scaling
- Can be fully vectorized
  - >60% of peak performance on Sandy-Bridge CPUs

	Smallest system	**β-Strand**	**β-Strand TZ**	**1ZE7**	**1AMB**
N	158	434	434	1056	1731
N_{basis}	404	963	2934	2370	3892
% of non-zero[a] MO coefficients a_{ij} ($A_{ij} \neq 0$)	81.3% (99.4%)	48.4% (76.0%)	73.4% (81.9%)	49.4% (72.0%)	37.1% (66.1%)
Average % of non-zero basis functions $\chi_i(\mathbf{r}_j)$ ($B_{1ij} \neq 0$)	36.2%	14.8%	8.2%	5.7%	3.9%
Average number of non-zero elements per column of B_{1ij}	146	142	241	135	152

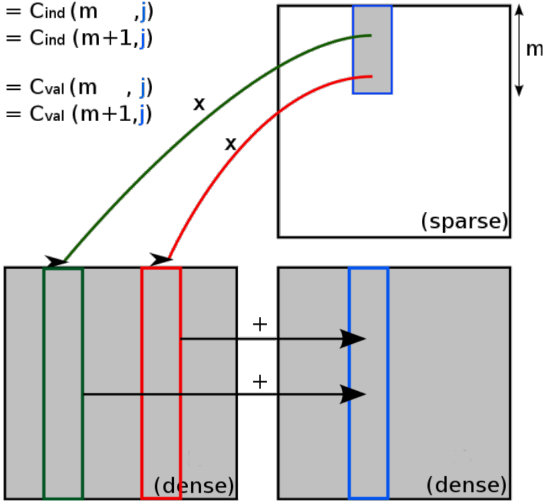
```
do j=1,point_num
   mo_vgl(:,:,j) = 0.d0
   do k=1,ao_num
      if (ao_vgl(k,1,j) /= 0.d0) then
         c1 = ao_vgl(k,1,j)
         c2 = ao_vgl(k,2,j)
         c3 = ao_vgl(k,3,j)
         c4 = ao_vgl(k,4,j)
         c5 = ao_vgl(k,5,j)
         do i=1,mo_num
            mo_vgl(i,1,j) = mo_vgl(i,1,j) + coefficient_t(i,k) * c1
            mo_vgl(i,2,j) = mo_vgl(i,2,j) + coefficient_t(i,k) * c2
            mo_vgl(i,3,j) = mo_vgl(i,3,j) + coefficient_t(i,k) * c3
            mo_vgl(i,4,j) = mo_vgl(i,4,j) + coefficient_t(i,k) * c4
            mo_vgl(i,5,j) = mo_vgl(i,5,j) + coefficient_t(i,k) * c5
         end do
      end if
   end do
end do
```

Sparse / dense matrix multiplication



$k_1 = C_{ind}(m, j)$
$k_2 = C_{ind}(m+1, j)$

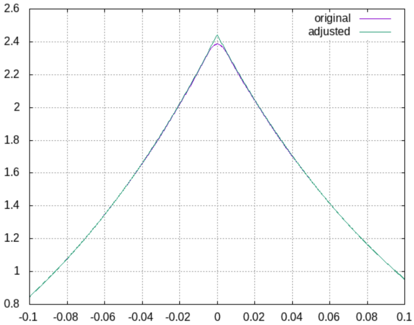
$c_1 = C_{val}(m, j)$
$c_2 = C_{val}(m+1, j)$



Electron-Nucleus Cusp fitting

$$\phi_{\text{cusp}\, i}(\mathrm{r}) = \phi_i(\mathrm{r}) - \phi_{s_A i}(\mathrm{r}) + \sum_{k=0}^{3} f_k \left|\mathrm{r} - \mathrm{R}_A\right|^k, \quad \text{where } \left|\mathrm{r} - \mathrm{R}_A\right| < r_{\text{cusp},A}$$

- $\phi_{s_A i}$: contributions of the *s* AOs centered at *A* to MO ϕ_i.
- 3 conditions:
  - Electron-nucleus cusp at $|\mathrm{r} - \mathrm{R}_A| = 0$
  - Continuity of the MO: $\phi_{\text{cusp}\, i} = \phi_i$ when $|\mathrm{r} - \mathrm{R}_A| = r_{\text{cusp},A}$
  - Continuity of the gradient: $\nabla\phi_{\text{cusp}\, i}(\mathrm{r}) = \nabla\phi_i(\mathrm{r})$ when $|\mathrm{r} - \mathrm{R}_A| = r_{\text{cusp},A}$



Jastrow factor

$$J_{\text{een}}(\text{r},\text{R}) = \sum_{\alpha=1}^{N_{\text{nucl}}} \sum_{i=1}^{N_{\text{elec}}} \sum_{j=1}^{i-1} \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} c_{lkp\alpha} \left(r_{ij}\right)^k \left[\left(R_{i\alpha}\right)^l + \left(R_{j\alpha}\right)^l\right] \left(R_{i\,\alpha}\, R_{j\alpha}\right)^{(p-k-l)/2}$$

can be rewritten as

$$J_{\text{een}}(\text{r},\text{R}) = \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} \sum_{\alpha=1}^{N_{\text{nucl}}} c_{lkp\alpha} \sum_{i=1}^{N_{\text{elec}}} \bar{\mathtt{R}}_{i,\alpha,(p-k-l)/2}\, \bar{\mathtt{P}}_{i,k,\alpha,(p-k+l)/2}$$ (↓ complexity)

with

$$\bar{\mathtt{P}}_{i,k,\alpha,l} = \sum_{j=1}^{N_{\text{elec}}} \bar{\mathtt{r}}_{i,k,j}\, \bar{\mathtt{R}}_{j,\alpha,l}.$$ (GEMM)

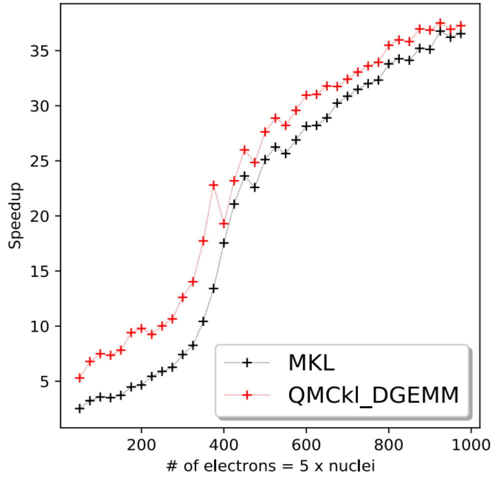
3-body component of the Jastrow factor

$$\begin{aligned}
\nabla_{im} J_{\text{een}}(\text{r}, \text{R}) = & \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} \sum_{\alpha=1}^{N_{\text{nucl}}} c_{lkp\alpha} \sum_{i=1}^{N_{\text{elec}}} \bar{\mathtt{G}}_{i,m,\alpha,(p-k-l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k+l)/2} + \\
& \bar{\mathtt{G}}_{i,m,\alpha,(p-k+l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k-l)/2} + \bar{\mathtt{R}}_{i,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,m,\alpha,k,(p-k+l)/2} + \\
& \bar{\mathtt{R}}_{i,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,m,\alpha,k,(p-k-l)/2} + \delta_{m,4} \Big(\\
& \bar{\mathtt{G}}_{i,1,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,1,\alpha,k,(p-k-l)/2} + \bar{\mathtt{G}}_{i,2,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,2,\alpha,k,(p-k-l)/2} + \\
& \bar{\mathtt{G}}_{i,3,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,3,\alpha,k,(p-k-l)/2} + \bar{\mathtt{G}}_{i,1,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,1,\alpha,k,(p-k+l)/2} + \\
& \bar{\mathtt{G}}_{i,2,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,2,\alpha,k,(p-k+l)/2} + \bar{\mathtt{G}}_{i,3,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,3,\alpha,k,(p-k+l)/2} \Big)
\end{aligned}$$

with

$$\bar{\mathtt{G}}_{i,m,\alpha,l} = \frac{\partial \left(R_{i\alpha}\right)^l}{\partial r_i}, \qquad \bar{\mathtt{g}}_{i,m,j,k} = \frac{\partial \left(r_{ij}\right)^k}{\partial r_i}, \qquad \text{and} \quad \bar{\mathtt{Q}}_{i,m,\alpha,k,l} = \sum_{j=1}^{N_{\text{elec}}} \bar{\mathtt{g}}_{i,m,j,k}\, \bar{\mathtt{R}}_{j,\alpha,l}$$

Speedup



Speedup
35
30
25
20
15
10
5
200
400
600
800
1000
# of electrons = 5 x nuclei
MKL
QMCkl_DGEMM

HPC implementations

- MAQAO, developed by the UVSQ team, is used to help us optimize the CPU code
  - Loop-level diagnostics
  - Vectorization ratio
  - Hints to improve efficiency
- Algorithms rewritten in C:
  - C compilers are usually more mature than Fortran on new hardware
  - Access to more low-level features than Fortran (pinned memory, alignment, inline assembly, etc)
- Precision can be changed on-the-fly: switch to single-precision if possible
- Specialization:
  - Specialization for s, p and d AOs
  - Inverse of small matrices hard-coded for 2×2 to 5×5
  - Small matrix multiplication
  - ...

Cofactor 4×4

```
subroutine cofactor4(a,LDA,b,LDB,na,det_l)
  implicit none
  double precision, intent(in)    :: A (LDA,na)
  double precision, intent(out)   :: B (LDA,na)
  integer*8, intent(in)           :: LDA, LDB
  integer*8, intent(in)           :: na
  double precision, intent(inout) :: det_l
  integer :: i,j
  det_l =  a(1,1)*(a(2,2)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))  &
                  -a(2,3)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))  &
                  +a(2,4)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))) &
          -a(1,2)*(a(2,1)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))  &
                  -a(2,3)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))  &
                  +a(2,4)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))) &
          +a(1,3)*(a(2,1)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))  &
                  -a(2,2)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))  &
                  +a(2,4)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))) &
          -a(1,4)*(a(2,1)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))  &
                  -a(2,2)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))  &
                  +a(2,3)*(a(3,1)*a(4,2)-a(3,2)*a(4,1)))

  b(1,1) =  a(2,2)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))-a(2,3)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))+a(2,4)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))
  b(2,1) = -a(2,1)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))+a(2,3)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))-a(2,4)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))
  b(3,1) =  a(2,1)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))-a(2,2)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))+a(2,4)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))
  b(4,1) = -a(2,1)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))+a(2,2)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))-a(2,3)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))

  b(1,2) = -a(1,2)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))+a(1,3)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))-a(1,4)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))
  b(2,2) =  a(1,1)*(a(3,3)*a(4,4)-a(3,4)*a(4,3))-a(1,3)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))+a(1,4)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))
  b(3,2) = -a(1,1)*(a(3,2)*a(4,4)-a(3,4)*a(4,2))+a(1,2)*(a(3,1)*a(4,4)-a(3,4)*a(4,1))-a(1,4)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))
  b(4,2) =  a(1,1)*(a(3,2)*a(4,3)-a(3,3)*a(4,2))-a(1,2)*(a(3,1)*a(4,3)-a(3,3)*a(4,1))+a(1,3)*(a(3,1)*a(4,2)-a(3,2)*a(4,1))

  b(1,3) =  a(1,2)*(a(2,3)*a(4,4)-a(2,4)*a(4,3))-a(1,3)*(a(2,2)*a(4,4)-a(2,4)*a(4,2))+a(1,4)*(a(2,2)*a(4,3)-a(2,3)*a(4,2))
  b(2,3) = -a(1,1)*(a(2,3)*a(4,4)-a(2,4)*a(4,3))+a(1,3)*(a(2,1)*a(4,4)-a(2,4)*a(4,1))-a(1,4)*(a(2,1)*a(4,3)-a(2,3)*a(4,1))
  b(3,3) =  a(1,1)*(a(2,2)*a(4,4)-a(2,4)*a(4,2))-a(1,2)*(a(2,1)*a(4,4)-a(2,4)*a(4,1))+a(1,4)*(a(2,1)*a(4,2)-a(2,2)*a(4,1))
  b(4,3) = -a(1,1)*(a(2,2)*a(4,3)-a(2,3)*a(4,2))+a(1,2)*(a(2,1)*a(4,3)-a(2,3)*a(4,1))-a(1,3)*(a(2,1)*a(4,2)-a(2,2)*a(4,1))

  b(1,4) = -a(1,2)*(a(2,3)*a(3,4)-a(2,4)*a(3,3))+a(1,3)*(a(2,2)*a(3,4)-a(2,4)*a(3,2))-a(1,4)*(a(2,2)*a(3,3)-a(2,3)*a(3,2))
  b(2,4) =  a(1,1)*(a(2,3)*a(3,4)-a(2,4)*a(3,3))-a(1,3)*(a(2,1)*a(3,4)-a(2,4)*a(3,1))+a(1,4)*(a(2,1)*a(3,3)-a(2,3)*a(3,1))
  b(3,4) = -a(1,1)*(a(2,2)*a(3,4)-a(2,4)*a(3,2))+a(1,2)*(a(2,1)*a(3,4)-a(2,4)*a(3,1))-a(1,4)*(a(2,1)*a(3,2)-a(2,2)*a(3,1))
  b(4,4) =  a(1,1)*(a(2,2)*a(3,3)-a(2,3)*a(3,2))-a(1,2)*(a(2,1)*a(3,3)-a(2,3)*a(3,1))+a(1,3)*(a(2,1)*a(3,2)-a(2,2)*a(3,1))

end subroutine cofactor4
```

GPU

- GPU library has the same functions, suffixed with `_device`
- Two different flavours: OpenMP or OpenACC
- Possibility to use CPU and GPU library together in the same code
- In early development, not fully integrated to our codes yet (work in progress)
- Although the kernels are fast on Nvidia GPUs, GPU acceleration is not clear because of data transfer
  - Maybe efficient on next generation of hardware
- On GPU, brute-force CuBLAS DGEMM is faster than sparse AO-MO transformation. Energy efficiency?

GPU: major difficulties with OpenMP

- Tensor core instructions are not generated in OpenMP kernels $\Longrightarrow \leq$ 50% peak DP
- Conflict between OpenMP runtime of the code and of QMCkl-GPU $\Longrightarrow$
  - Need to compile the code with GPU compiler (Nvfortran)
  - May not compile, or with low CPU efficiency
  - Our solution: decouple QMCkl-CPU and QMCkl-GPU and recover CPU performance with QMCkl-CPU
- RocBLAS $\sim$ CuBLAS, but some OpenMP kernels have $10\times$ lower performance on AMD GPUs than Nvidia (under investigation...)
- Unreliable software stack: $\Longrightarrow$ Compared to CPU, very inefficient in human resources
- Open Question:
  - Should we have opted instead for vendor-specific implementations? (Cuda, HIP)

Technical aspects

```
$ tar -zxvf qmckl.tar.gz
$ cd qmckl
$ ./configure --enable-hpc
$ make -j 32
$ make check
$ make install
```

- QMCkl has been used in
  - C / C++
  - Fortran
  - Python
  - Julia
  - Rust

- Very few dependencies:
  - BLAS/Lapack (CPU)
  - TREXIO (optional) with HDF5 (optional)
- BSD license: very permissive. You can distribute the `tar.gz` with your code
- Hosted on GitHub: `https://github.com/trex-coe/qmckl`

Integration into TREX codes

- Single-core benchmark: C_{60}, Hartree-Fock/cc-pVQZ/ECP(BFD)
  - Time for a single MC step (all-electrons)
  - 4140 AOs, 120 MOs, 240 electrons

CPU	Compiler	QMCkl	milliseconds	Speedup
Intel(R) Core(TM) i7	ifort/mkl	-	24.58	
(8-core Laptop, 2.8GHz)	ifort/mkl	gcc12	24.06	1.02x
	ifort/mkl	icx	23.85	1.03x
	gfortran/openblas	-	30.58	
	gfortran/openblas	gcc12	26.04	1.17x
ARM Neoverse V1	gfortran/armpl	-	41.24	
(80 cores, 3GHz)	gfortran/armpl	gcc12	31.91	1.29x

- Single-core benchmark: C_{60}, Hartree-Fock/cc-pVXZ/ECP(BFD)
  - Short VMC run
  - 4140 AOs, 120 MOs, 240 electrons

Basis	# AOs	Compiler	QMCkl	seconds	Speedup
cc-pVDZ	840	ifort/mkl	-	315.45	
			gcc12	218.29	1.45x
			icx	212.35	1.49x
cc-pVTZ	2040	ifort/mkl	-	565.67	
			gcc12	287.32	1.97x
			icx	271.68	2.08x
cc-pVQZ	4140	ifort/mkl	-	993.42	
			gcc12	462.74	2.15x
			icx	441.32	2.25x

Other possible applications beyond accelerating QMC

- Reproducibility of QMC calculations (Jastrow factors)
- 3D visualization software:
  - AO or MO visualization
  - Interpretative methods like AIM or ELF
- Numerical integration
  - Computation of density grids for DFT with gradients
  - Jastrow factor in transcorrelated methods (Quantum Package)
- Teaching QMC algorithms in Jupyter notebooks
- Implementation of QMC methods in traditional quantum chemistry software

Example : Evaluate MOs on a grid

```python
import qmckl
import numpy as np

def main(trexio_filename):
  context = qmckl.context_create()                    # Create a QMCkl context
  qmckl.trexio_read(context, trexio_filename)         # Read the TREXIO file into the context

  nucl_num   = qmckl.get_nucleus_num(context)                    # Get the number of nuclei
  nucl_coord = qmckl.get_nucleus_coord(context, 'N', nucl_num*3) # Get the nuclear coordinates
  nucl_coord = np.reshape(nucl_coord, (3, nucl_num))
  mo_num     = qmckl.get_mo_basis_mo_num(context)                # Get the number of MOs

  point     = setup_grid_points(nucl_coord)
  point_num = len(point)

  qmckl.set_point(context, 'N', point_num, np.reshape(point, (point_num*3))) # Give points to QMCkl

  mo_value = qmckl.get_mo_basis_mo_value(context, point_num*mo_num)  # Get the values of the MOs

  qmckl.context_destroy(context)                    # Free QMCkl resources
```

Contributors

CNRS
- Vijay Gopal Chilkuri
- Evgeny Posenitskiy
- Anthony Scemama

U-Twente
- Ravindra Shinde
- Edgar Landinez Borda
- Ramon Lorenzo Panades-Barrueta
- Claudia Filippi

SISSA
- Oto Kohulak
- Sandro Sorella

CINECA
- Tommaso Gorni
- Gianfranco Abrusci

UVSQ
- François Coppens
- Kevin Camus
- Aurelien Delval
- Max Hoffer
- Pablo Heitor De Oliveira Castro Herrero
- Cedric Valensi
- William Jalby