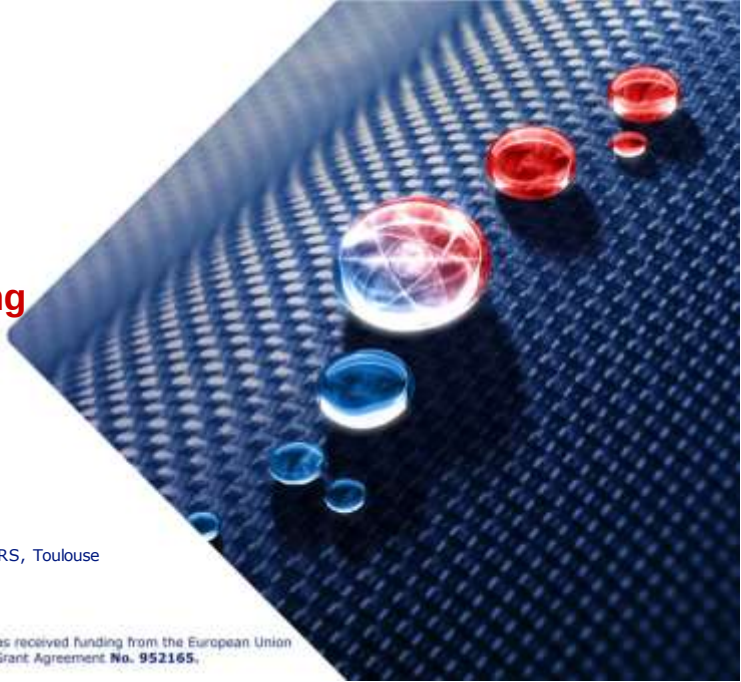
Targeting Real chemical accuracy at the EXascale

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

Partners

UNIVERSITY OF TWENTE.
cnrs
CINECA
JÜLICH
MEGWARE
SISSA
STU
Trust-IT Services
universität wien
UVSQ

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?

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

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

```
1   do i=1,n
2     Z(i) = Z(i) + A * X(i) + Y(i)
3   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 ostream>( 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;
}
```

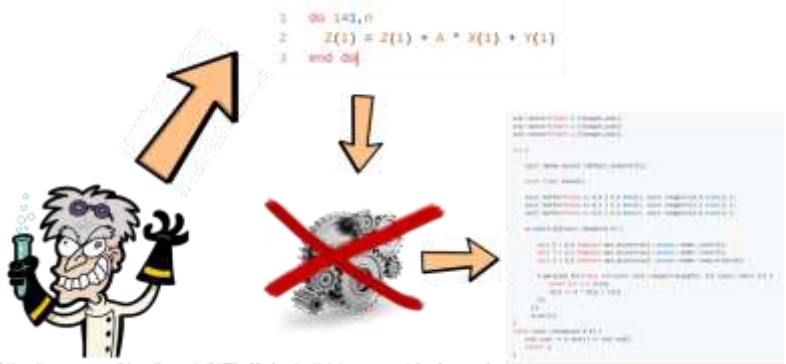
```
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

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 saxpyone>( sycl::range<1>{length}, [=] (sycl::id<1> i) {
            const int i = i[0];
            Z[i] += A * X[i] + Y[i];
        });
      });
      q.wait();
}
catch (sycl::exception A 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.

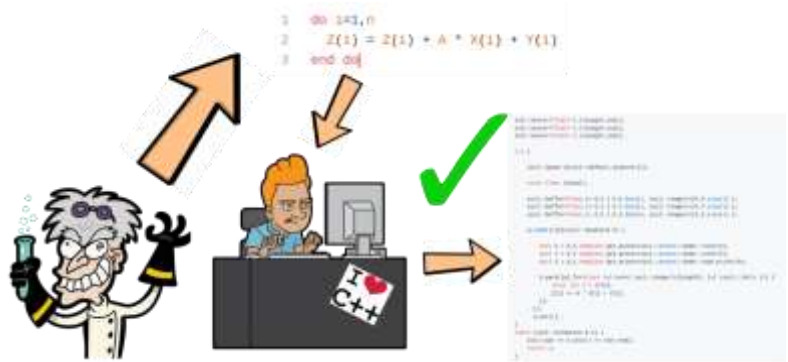
[1]Wikipedia: A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language)

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



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

Benefits of this model

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

- 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!)

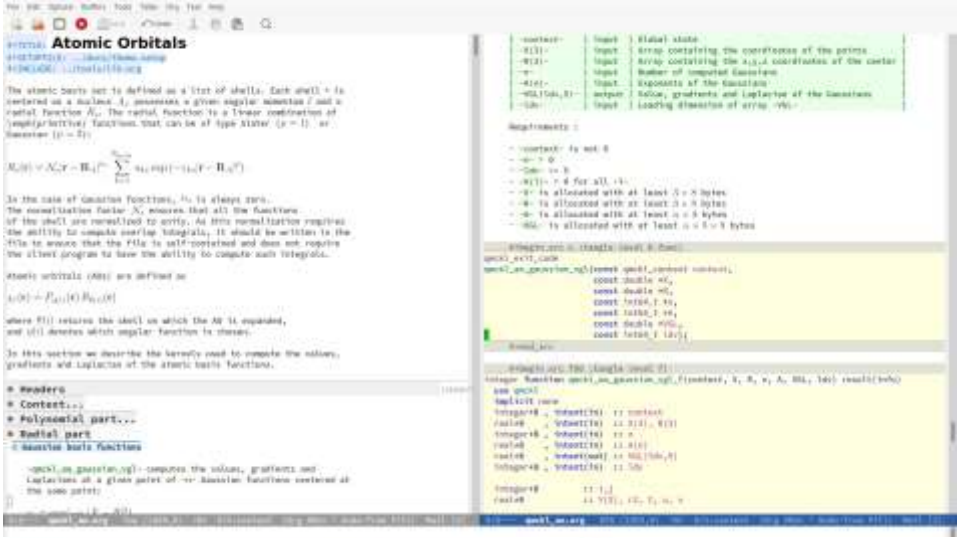
Literate programming is a programming paradigm introduced by Donald Knuth in which a computer program is given an explanation of its logic in a natural language, such as English, interspersed with snippets of macros and traditional source code, from which compilable source code can be generated. (Wikipedia)

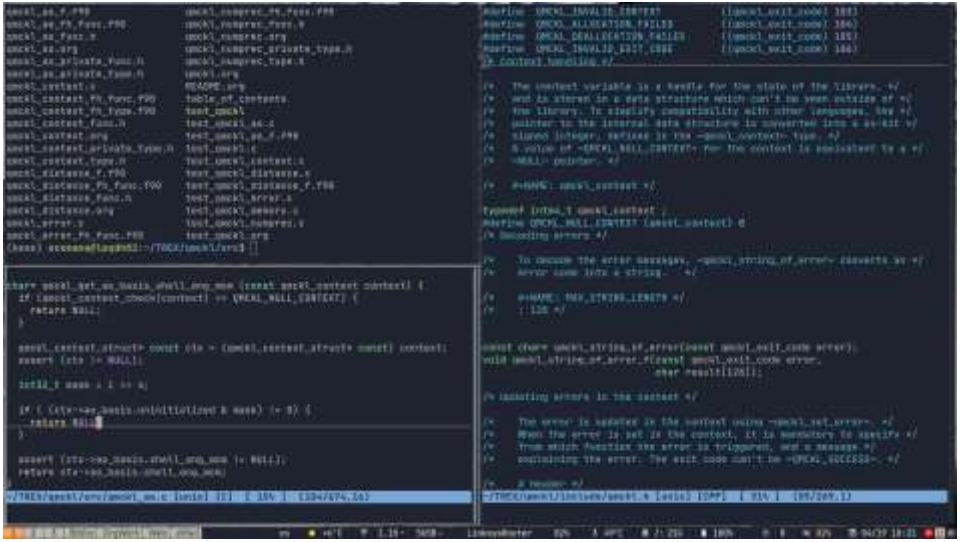
Literate programming with *org-mode*:

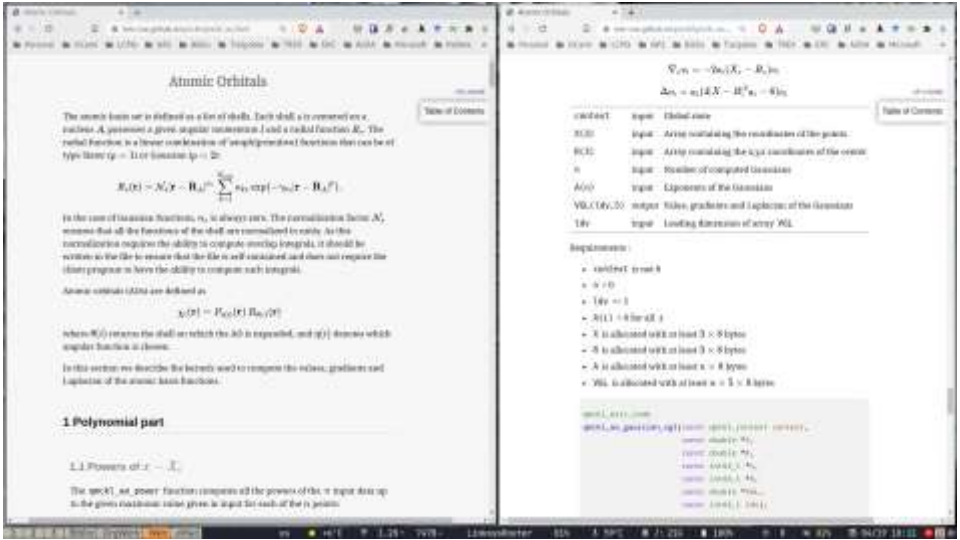
- Here, comments are more important than code
- Can add graphics, LaTeX formulas, tables, *etc*
- Documentation always synchronized with the code
- Some functions can be generated by embedded scripts
- Web site auto-generated when code is pushed

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

Literate programming with *org-mode*




qmckl_ao_f.F90 qmckl_nuclear_f4_func.F90
qmckl_ao_f4_func.F90 qmckl_nuclear_func.h
qmckl_ao_func.h qmckl_nuclear.org
qmckl_ao.org qmckl_nuclear_private_type.h
qmckl_ao_private_func.h qmckl_nuclear_type.h
qmckl_ao_private_type.h qmckl.org
qmckl_context.c README.org
qmckl_context_f4_func.F90 table_of_contents
qmckl_context_f4_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_f4_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_f4_func.F90 test_qmckl.org
(kess) xxxxxx@Laptop01:~/TREX/qmckl/src$
#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. Hereafter is 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~. */
/* # Version */
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] 72: [35%] 1334/3774,16)
~/TREX/qmckl/include/qmckl.h [unix] [INF] [81%] [85/105,1)
vs +6°C 1.15 5458 LinksysMaster 82% 40°C 2: 25% 100% 82% 04/29 10:31


Atomic Orbitals
Table of Contents
1 Polynomial part
Table of Contents

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 $E_{\text{loc}}(r_1, \ldots, r_N) = \frac{\hat{H}\Psi(r_1,\ldots,r_N)}{\Psi(r_1,\ldots,r_N)}$:

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

Kernels implemented and well tested today

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

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

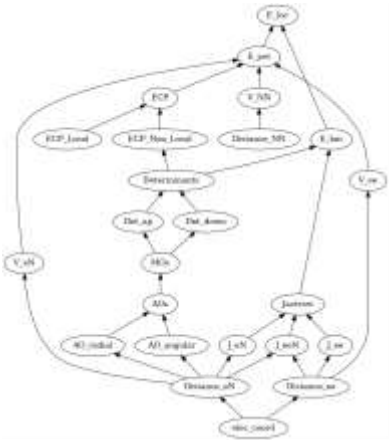
Kernels implemented and well tested today

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

Work in progress

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

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



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

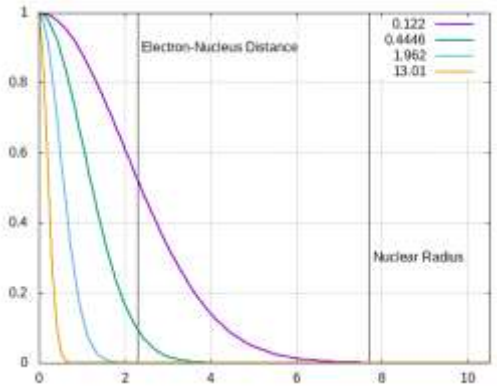
Atomic Orbitals

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

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

Atomic Orbitals: Linear scaling



- 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

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

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

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

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

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

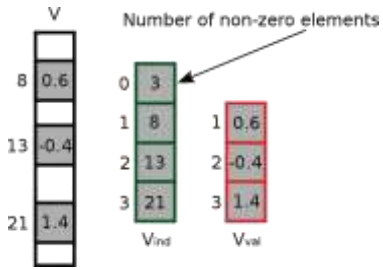
- QMC=Chem (2013): https://doi.org/10.1002/jcc.23216
- Exploits the common sparse character of the AO matrices:
  - When $\chi(r) = 0$ because r is too far, all the derivatives are also zero
  - Quadratic scaling
- Can be fully vectorized
  - >60% of peak performance on Sandy-Bridge CPUs

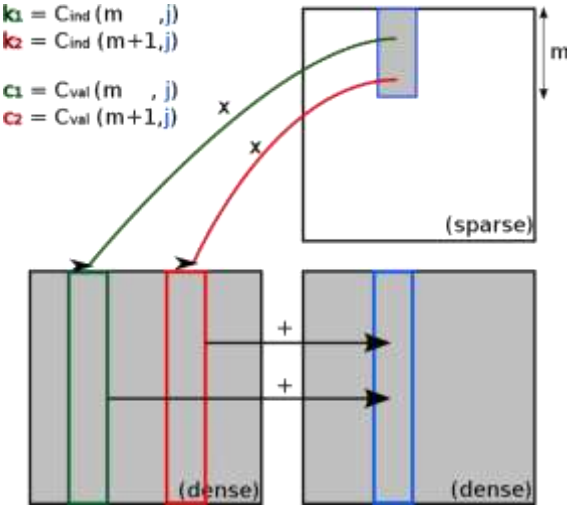
	Smallest system	β-Strand	β-Strand TZ	1ZE7	1AMB
N	158	434	434	1056	1731
N_{basis}	404	963	2934	2370	3892
% of non-zero[a] MO coefficients a_{ij} ($A_{ij} \neq 0$)	81.3% (99.4%)	48.4% (76.0%)	73.4% (81.9%)	49.4% (72.0%)	37.1% (66.1%)
Average % of non-zero basis functions $\chi_i(\mathbf{r}_j)$ ($B_{ij}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

Sparse / dense matrix multiplication

```
1   do j=1,point_num
2      mo_vgl(:,:,j) = 0.d0
3      do k=1,ao_num
4         if (ao_vgl(k,1,j) /= 0.d0) then
5            c1 = ao_vgl(k,1,j)
6            c2 = ao_vgl(k,2,j)
7            c3 = ao_vgl(k,3,j)
8            c4 = ao_vgl(k,4,j)
9            c5 = ao_vgl(k,5,j)
10           do i=1,mo_num
11              mo_vgl(i,1,j) = mo_vgl(i,1,j) + coefficient_t(i,k) * c1
12              mo_vgl(i,2,j) = mo_vgl(i,2,j) + coefficient_t(i,k) * c2
13              mo_vgl(i,3,j) = mo_vgl(i,3,j) + coefficient_t(i,k) * c3
14              mo_vgl(i,4,j) = mo_vgl(i,4,j) + coefficient_t(i,k) * c4
15              mo_vgl(i,5,j) = mo_vgl(i,5,j) + coefficient_t(i,k) * c5
16           end do
17        end if
18     end do
19  end do
```

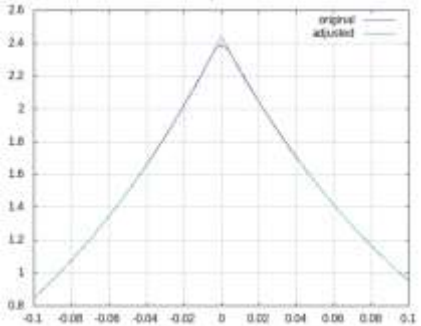
Sparse / dense matrix multiplication





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

- $\varphi_{s_A i}$: 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: $\varphi_{\text{cusp}\,i} = \varphi_i$ when $|\mathrm{r} - \mathrm{R}_A| = r_{\text{cusp},A}$
  - Continuity of the gradient: $\nabla\varphi_{\text{cusp}\,i}(\mathrm{r}) = \nabla\varphi_i(\mathrm{r})$ when $|\mathrm{r} - \mathrm{R}_A| = r_{\text{cusp},A}$



Jastrow factor

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

can be rewritten as

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

with

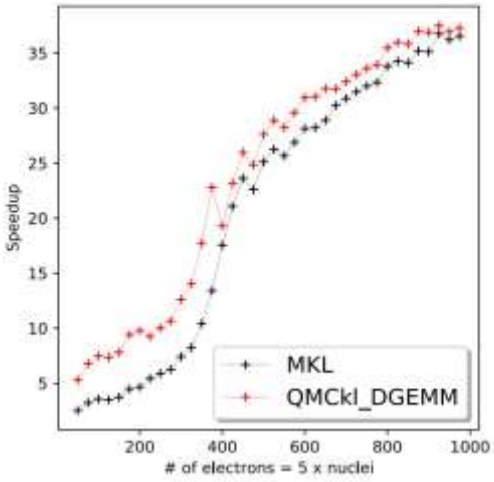
$$\bar{P}_{i,k,a,l} = \sum_{j=1}^{N_{elec}} \bar{r}_{i,k,j} \, \bar{R}_{j,a,l}.$$ (GEMM)

3-body component of the Jastrow factor

$$\nabla_{im} J_{een}(r, R) = \sum_{p=2}^{N_{ord}} \sum_{k=0}^{p-1} \sum_{l=0}^{2\delta_{k,0}} \sum_{a=1}^{N_{ucl}} \sum_{i=1}^{N_{elec}} c_{lkpa} \bar{G}_{i,m,a,(p-k-l)/2} \bar{P}_{i,a,k,(p-k+l)/2} +$$
$$\bar{G}_{i,m,a,(p-k+l)/2} \bar{P}_{i,a,k,(p-k-l)/2} + \bar{R}_{i,a,(p-k+l)/2} \bar{Q}_{i,m,a,k,(p-k-l)/2} +$$
$$\bar{R}_{i,a,(p-k+l)/2} \bar{Q}_{i,m,a,k,(p-k-l)/2} + \delta_{m,4}$$
$$\bar{G}_{i,1,a,(p-k+l)/2} \bar{Q}_{i,1,a,k,(p-k-l)/2} + \bar{G}_{i,2,a,(p-k+l)/2} \bar{Q}_{i,2,a,k,(p-k-l)/2} +$$
$$\bar{G}_{i,3,a,(p-k+l)/2} \bar{Q}_{i,3,a,k,(p-k-l)/2} + \bar{G}_{i,1,a,(p-k-l)/2} \bar{Q}_{i,1,a,k,(p-k+l)/2} +$$
$$\bar{G}_{i,2,a,(p-k-l)/2} \bar{Q}_{i,2,a,k,(p-k+l)/2} + \bar{G}_{i,3,a,(p-k-l)/2} \bar{Q}_{i,3,a,k,(p-k+l)/2}$$

with

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


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

- 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 $\Rightarrow \leq$ 50% peak DP
- Conflict between OpenMP runtime of the code and of QMCkl-GPU $\Rightarrow$
  - 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: $\Rightarrow$ Compared to CPU, very inefficient in human resources
- Open Question:
  - Should we have opted instead for vendor-specific implementations? (Cuda, HIP)

```
$ 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