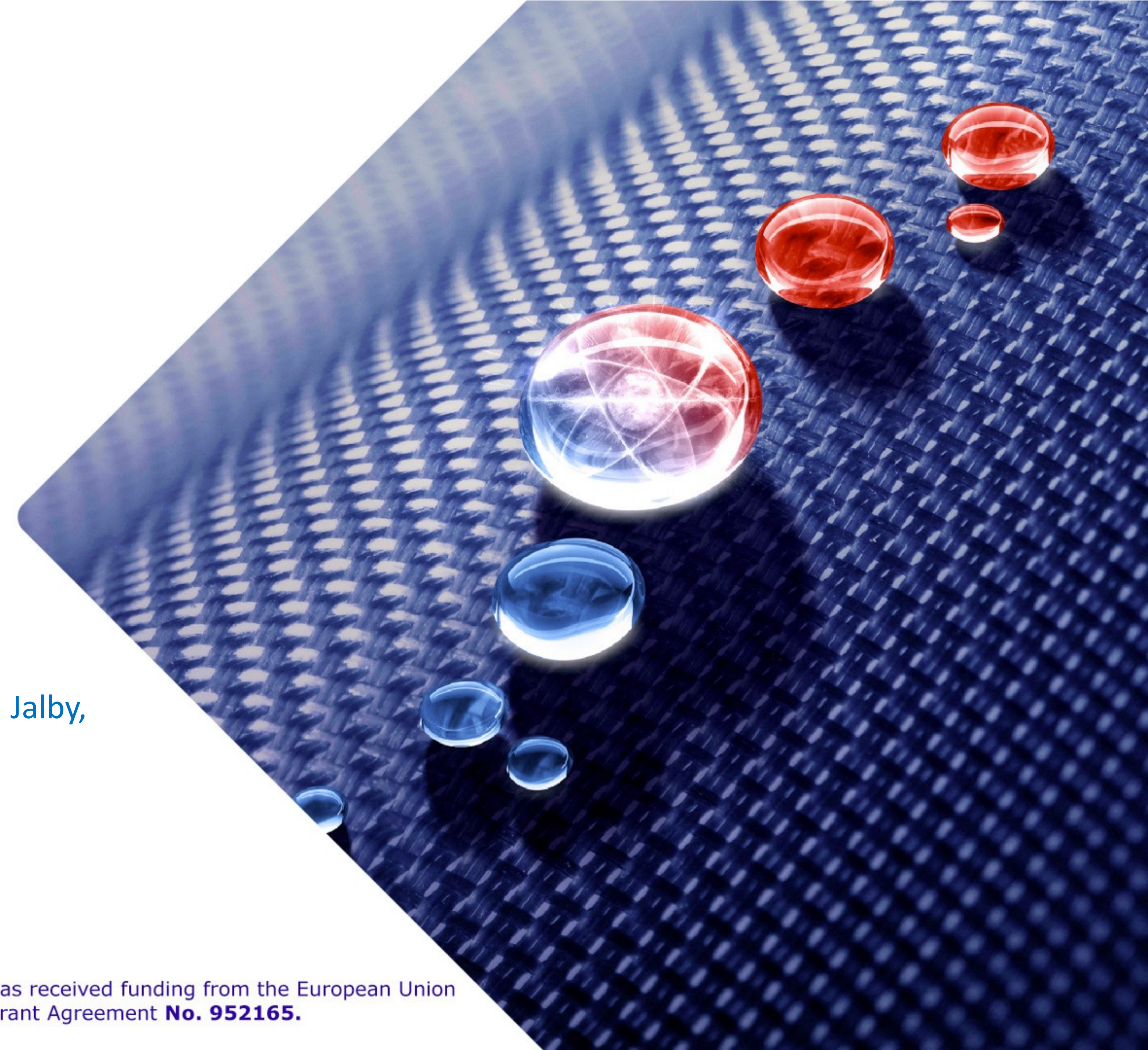
TREX

Targeting Real chemical accuracy at the EXascale



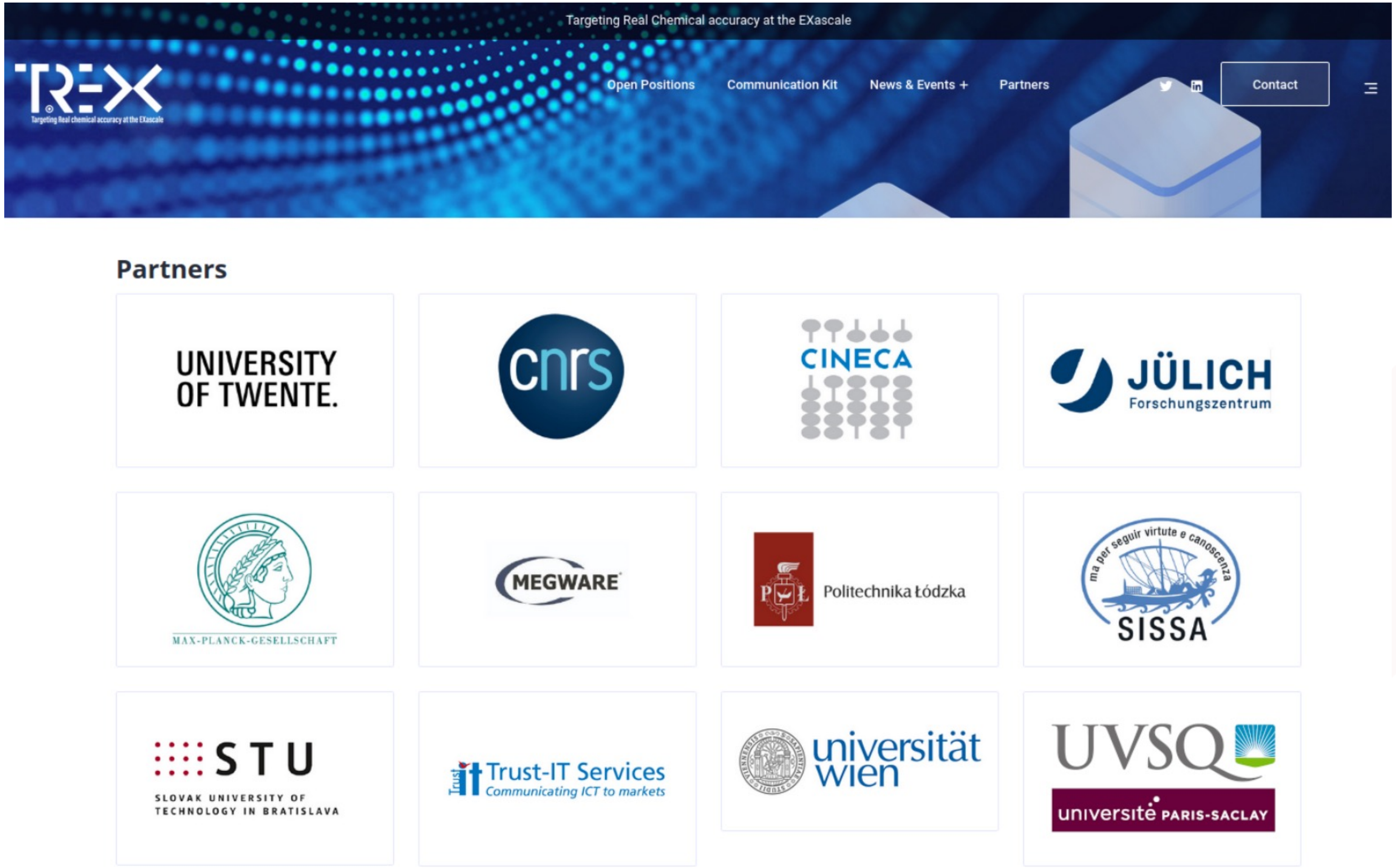
HPC Training Series: HPC Quantum Monte Carlo library (QMCkl)

Vijay Gopal Chilkuri, Evgeny Posenitskiy, William Jalby, Anthony Scemama

09-12-2024

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

Introduction – TREX


Targeting Real Chemical accuracy at the EXascale
TREX
Targeting Real chemical accuracy at the EXascale
Open Positions
Communication Kit
News & Events +
Partners
Contact
UNIVERSITY OF TWENTE.
cnrs
CINECA
JÜLICH Forschungszentrum
MAX-PLANCK-GESELLSCHAFT
MEGWARE
Politechnika Łódzka
ma per seguir virtute e canoscenza
SISSA
STU SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
Trust-IT Services Communicating ICT to markets
universität wien
UVSQ
université PARIS-SACLAY

Partners

Codes

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

Introduction – TREX

- TREX CoE: Targeting REal chemical accuracy at the eXascale
- Started in October 2020, ended in March 2024
- Objective: making codes ready for exascale systems
- How ? – Instead of rewriting codes, provide libraries
  - One library for high-performance (QMCkl)
  - One library for exchanging information (TREXIO)

Outline

Introduction to QMCkl

- QMCkl: High-performance Quantum Monte Carlo library

Development of Accurate and Efficient Algorithms

- Fast and Accurate Calculations: Jastrow Factor and DGEMM
- Conclusion

Introduction to QMCkl

- QMCkl: High-performance Quantum Monte Carlo library

Development of Accurate and Efficient Algorithms

- Fast and Accurate Calculations: Jastrow Factor and DGEMM
- Conclusion

QMCkl: A unified approach to accelerating Quantum Monte Carlo Codes

Quantum Monte Carlo kernel library (QMCkl)

QMCkl: Algorithms and APIs implemented

Quantum Monte Carlo method

$$\hat{H}|\Psi\rangle = \left(-\frac{1}{2}\nabla^2 + V\right)|\Psi\rangle = E|\Psi\rangle$$

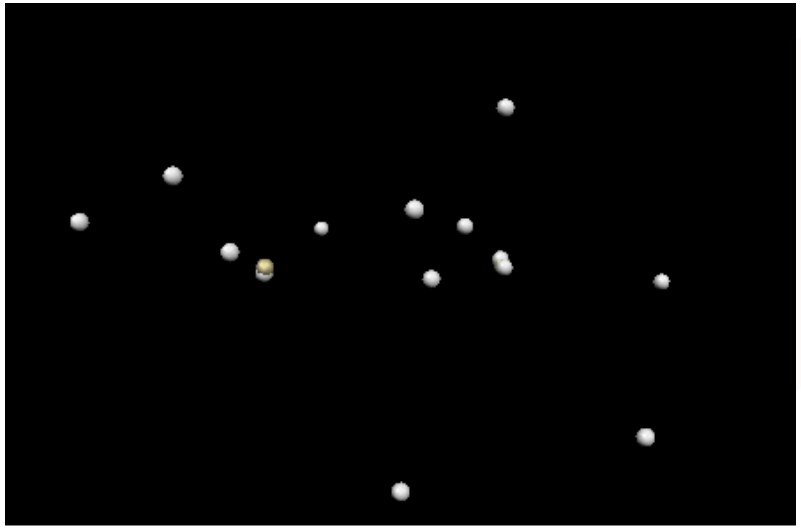
$$E = \int_R \Psi(R)(H\Psi)(R)\mathrm{d}R = \int_R E_L(R)\Psi^2\mathrm{d}R, \text{ where, } R = (r_1, \ldots, r_i, \ldots, r_n)$$

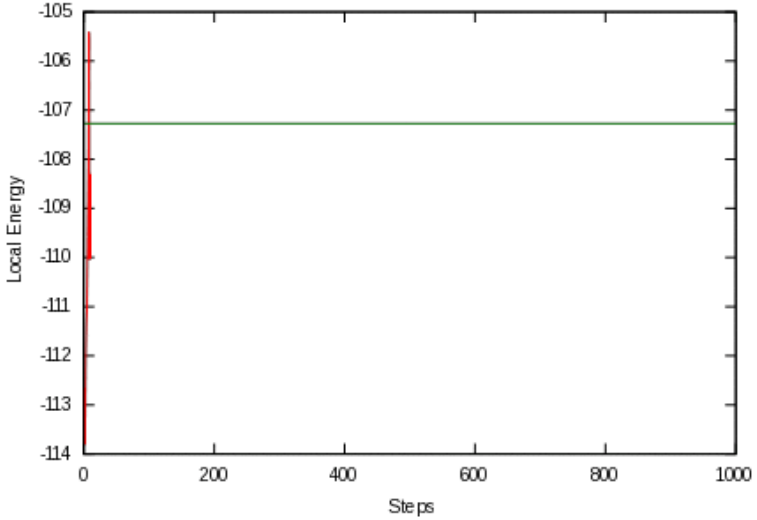
$$\langle E\rangle_{\Psi^2} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} E_L(R_i) \quad \text{with, } E_L(R) = \frac{(\hat{H}\Psi)(R)}{\Psi(R)}$$

Quantum Monte Carlo method

Simulation: N_2 molecule – 14e, 2 Nuclei

$$\langle E \rangle_{\Psi^2} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} E_L(R_i)$$





QMCkl: Algorithms and APIs implemented

Kernels Needed

$\Psi(r_1,\ldots,r_n)$: Wavefunction

$\vec{\nabla}\Psi(r_1,\ldots,r_n)$: Drift Vector

$\nabla^2\Psi(r_1,\ldots,r_n)$: Kinetic Energy

Kernels well Implemented and Tested

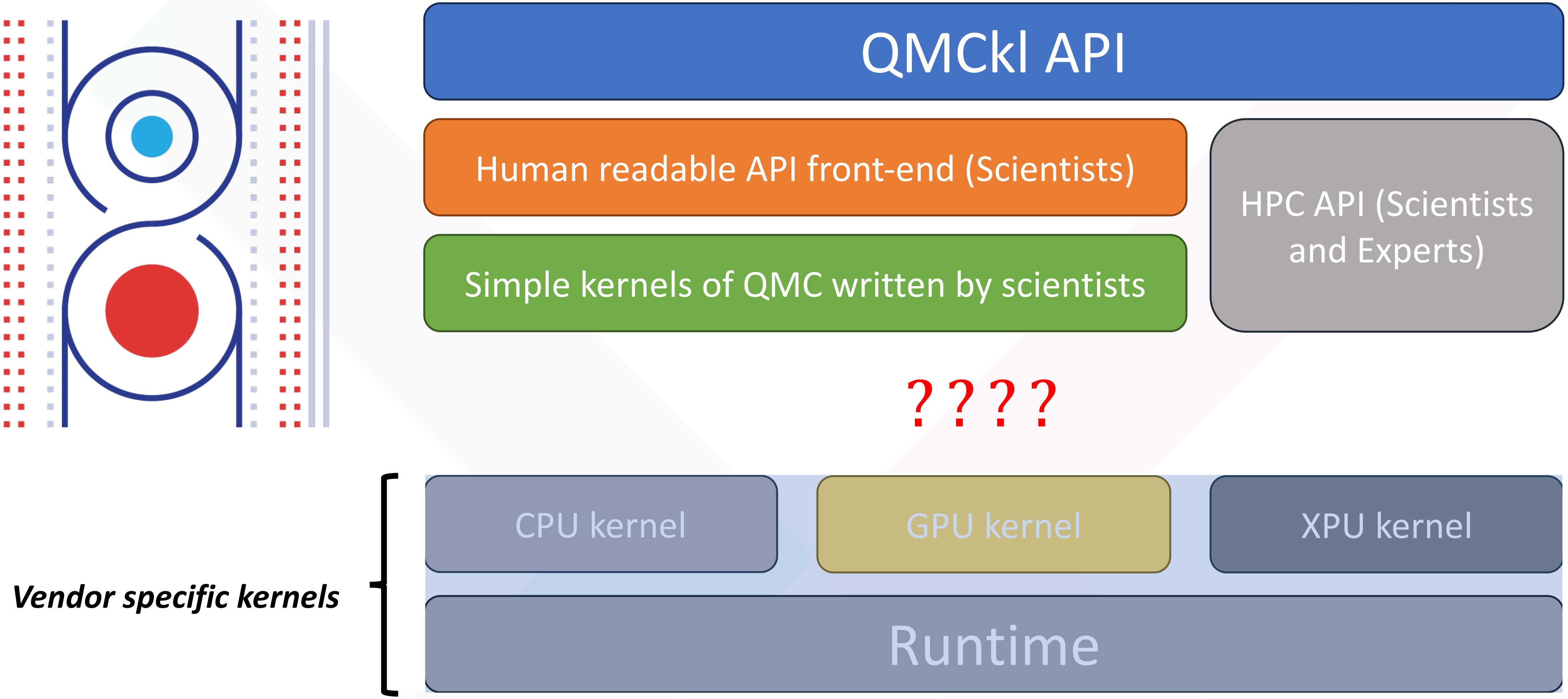
AOs: $\chi(r_i), \vec{\nabla}\chi(r_i), \nabla^2\chi(r_i)$

MOs: $\phi(r_i), \vec{\nabla}\phi(r_i), \nabla^2\phi(r_i)$

Inverse of small matrices

Jastrow correlation factor (eN, ee, eeN)

Quantum Monte Carlo kernel library (QMCkl)



QMCkl API

Human readable API front-end (Scientists)

Simple kernels of QMC written by scientists

HPC API (Scientists and Experts)

????

CPU kernel

GPU kernel

XPU kernel

Runtime

Vendor specific kernels

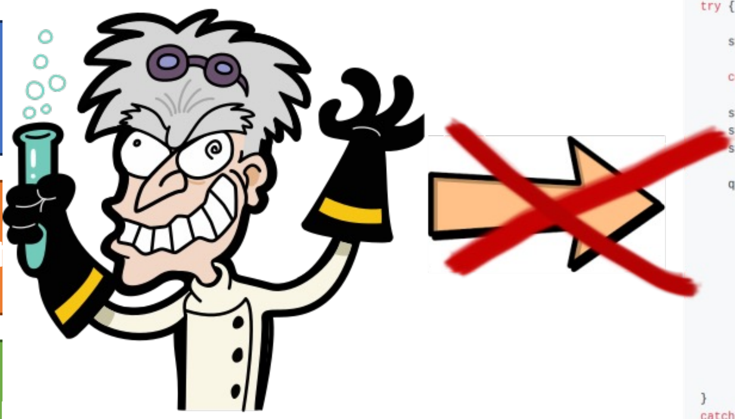
Introduction – QMCkl

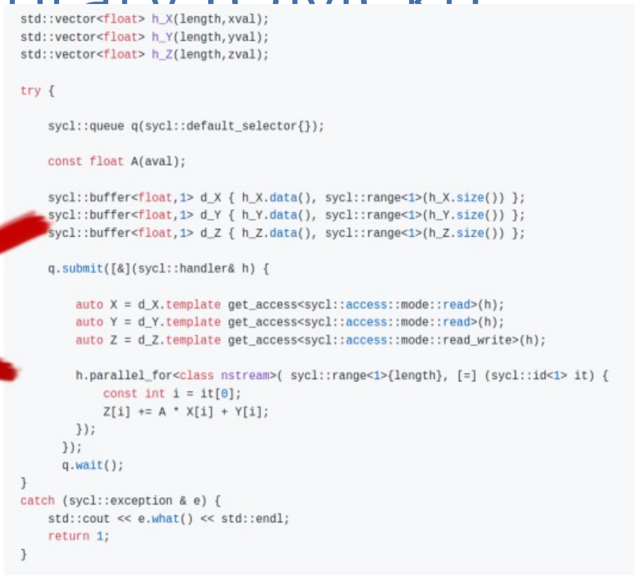
Quantum Monte Carlo kernel library (QMCkl)

Hu

Sim

ists



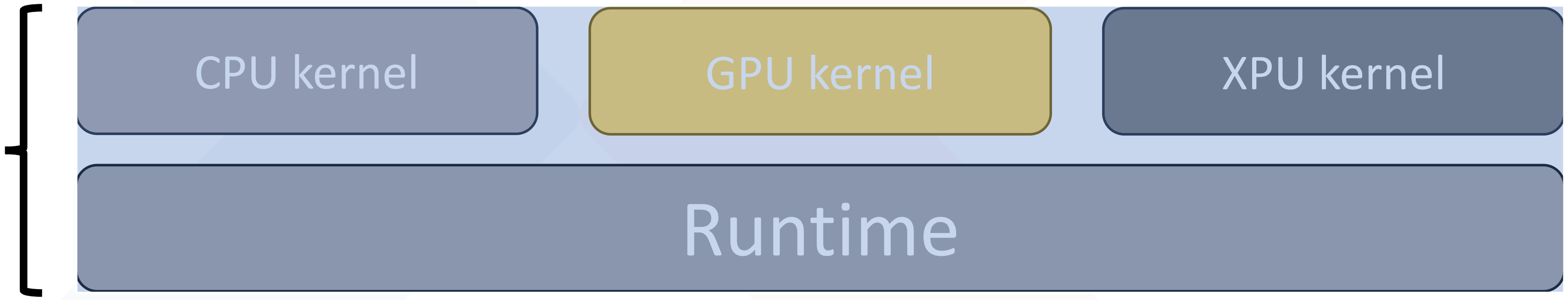


```
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

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

? ? ? ?

Vendor specific kernels



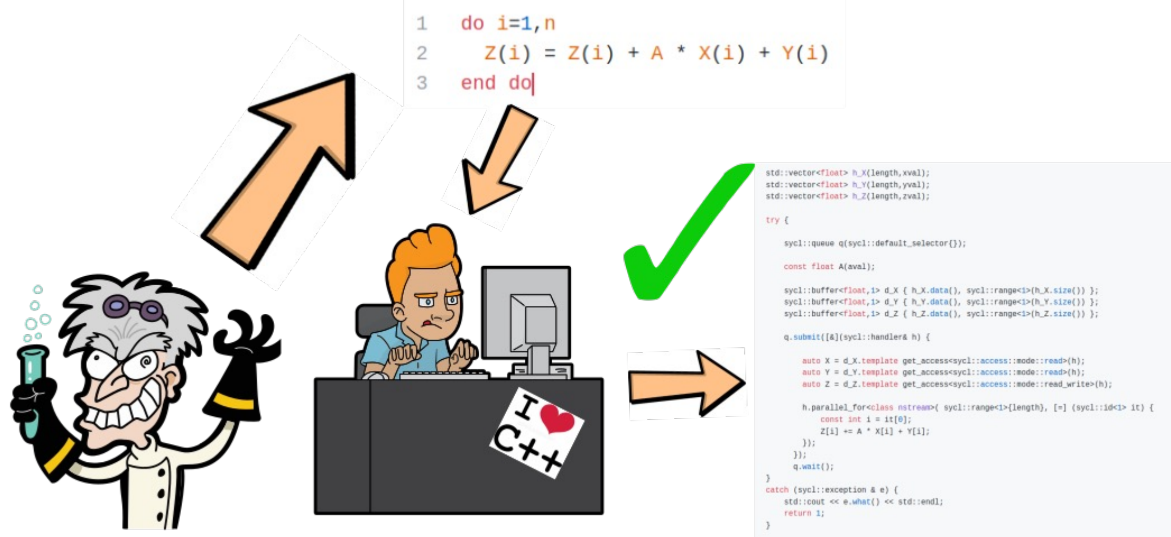
Quantum Monte Carlo kernel library (QMCkl)

Human

Simple k

ntists

ts)



```
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;
}
```

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

Bio-compiled kernels to obtain HPC library

CPU kernel

GPU kernel

XPU kernel

Runtime

Vendor specific kernels

Outline

Introduction to QMCkl

- QMCkl: High-performance Quantum Monte Carlo library

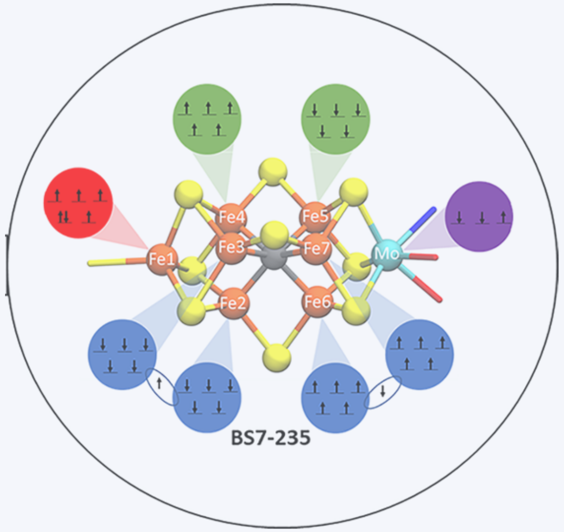
Development of Accurate and Efficient Algorithms

- Fast and Accurate Calculations: Jastrow Factor and DGEMM
- Conclusion

High-Performance Implementation: Jastrow Factor

Accurate *ab initio* calculations ⇒ Accurate models

Mono-determinantal *ansätze*
(DFT/HF/SR-CC)



$$\Psi_T = D_0$$

Single slater determinant:
no static and part of dynamic correlation

Multi-determinantal Fully parallel
Quantum Monte Carlo *ansätze*

$$\Psi_T = \sum_{i=1}^{N_{det}} C_i D_i$$

Multi-determinant Ψ_T:
static and part of dynamic correlation

3-body Jastrow factor

Slater determinants

$$\Psi_T = \mathcal{J}(e,e,n) \sum_{i=1}^{N_{det}} C_i D_i$$

Jastrow slater multi-determinant Ψ_T :
static and dynamic correlation

Chilkuri, Vijay Gopal, and Frank Neese, *J. Comput. Chem.* **2021,** 42.14, 982-1005.

Schautz, Friedemann, and Claudia Filippi, *J. Chem. Phys.* **2004,** 120.23, 10931-10941.

High-Performance Implementation: Jastrow Factor

Jastrow 3-body Correlation Factor

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

electron-electron distances

electron-nucleus distances

Anthony Scemama, Vijay Gopal Chilkuri and Claudia Filippi, *in preparation*

High-Performance Implementation: Jastrow Factor

Jastrow 3-body Correlation Factor

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

Scaling: $\mathcal{O}(\mathrm{N_{ord} N_{nuc} N_{elec}^2})$

Anthony Scemama, Vijay Gopal Chilkuri and Claudia Filippi, *in preparation*

High-Performance Implementation: Jastrow Factor

Jastrow 3-body Correlation Factor

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

can be rewritten as

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

with

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

Anthony Scemama, Vijay Gopal Chilkuri and Claudia Filippi, *in preparation*

High-Performance Implementation: Jastrow Factor

Jastrow 3-body Correlation Factor

Scaling: $\mathcal{O}(\mathrm{N_{ord}N_{nuc}N_{elec}^2})$

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

can be rewritten as

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

with

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

Scaling: $\mathcal{O}(\mathrm{N_{nuc}N_{elec}^2})$

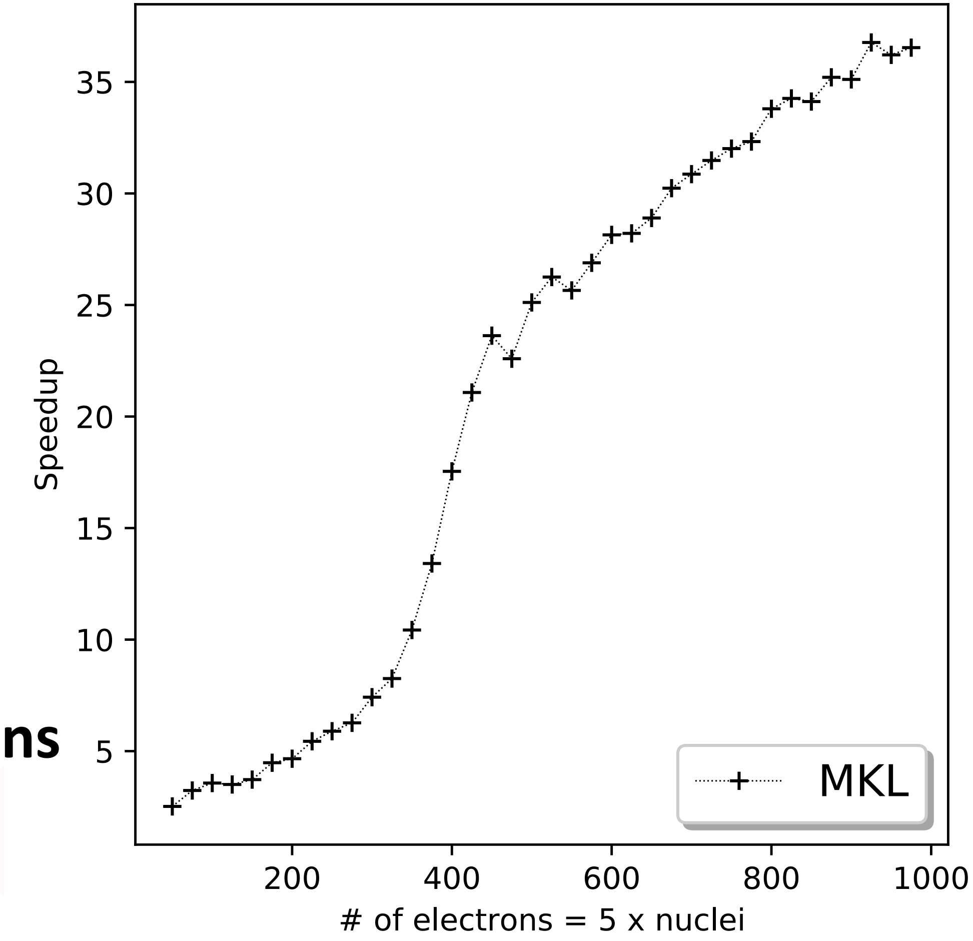
Anthony Scemama, Vijay Gopal Chilkuri and Claudia Filippi, *in preparation*

High-Performance Implementation: Jastrow Factor

Speedup for Jastrow Factor

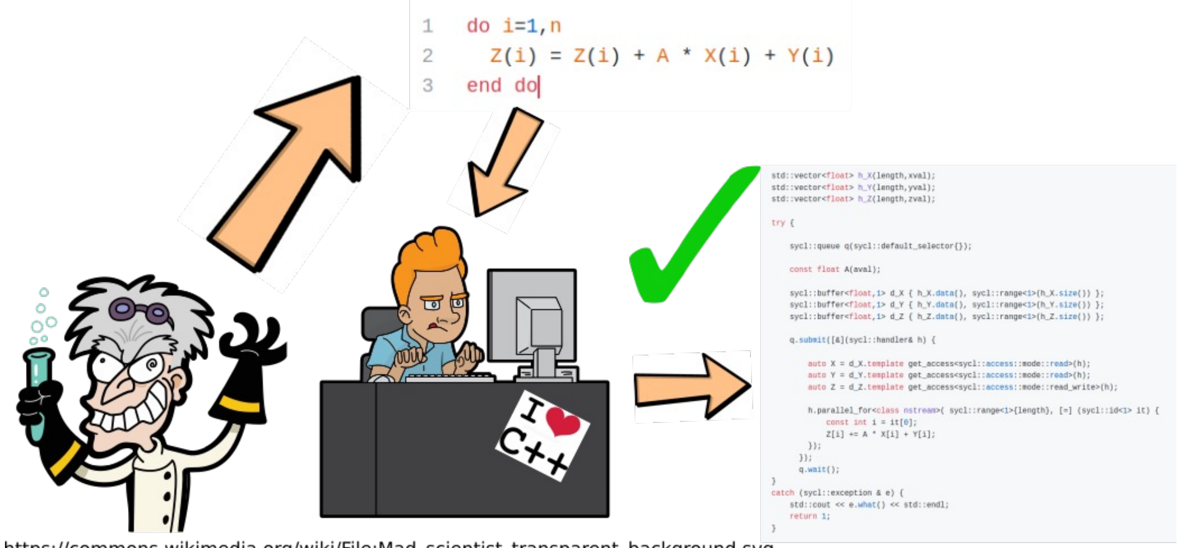
- DGEMM based algorithm shows large speedup over naïve algorithm
- Automatic OpenMP based intra-node parallelization

Final Speedup (vs Doc) → 35× for 1000 electrons



High-Performance Implementation: Jastrow Factor

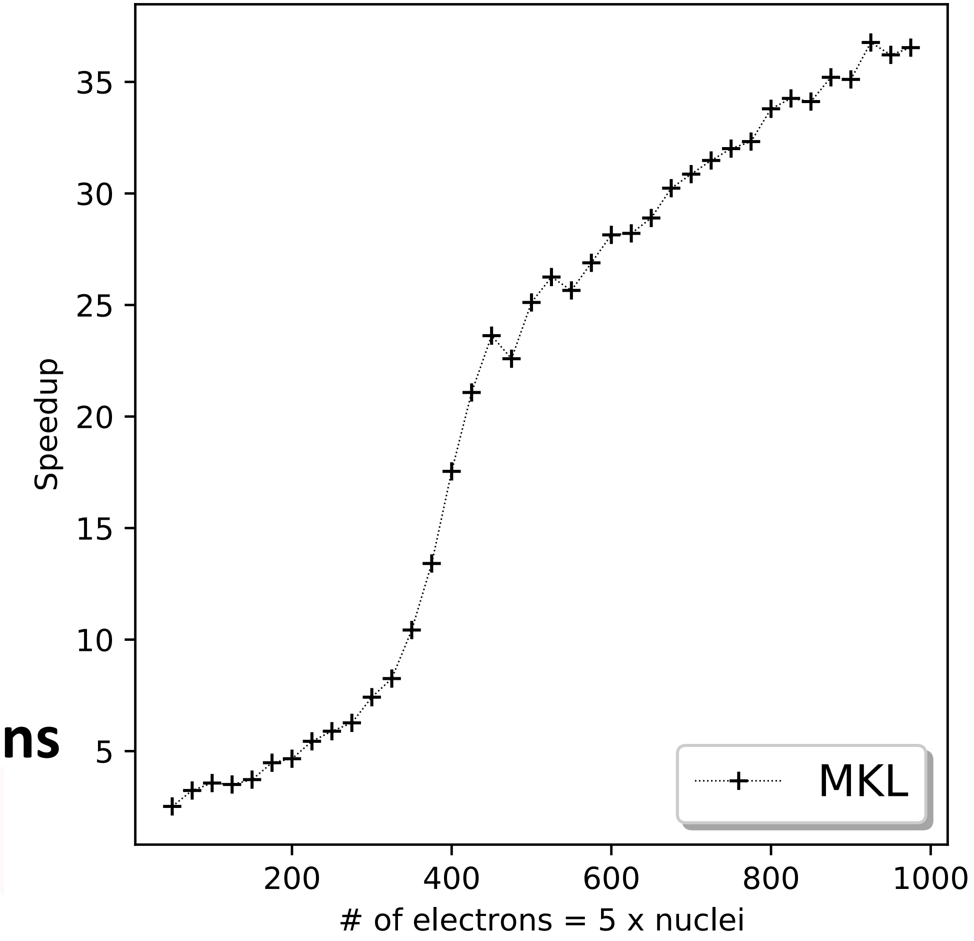
Speedup for Jastrow Factor



```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



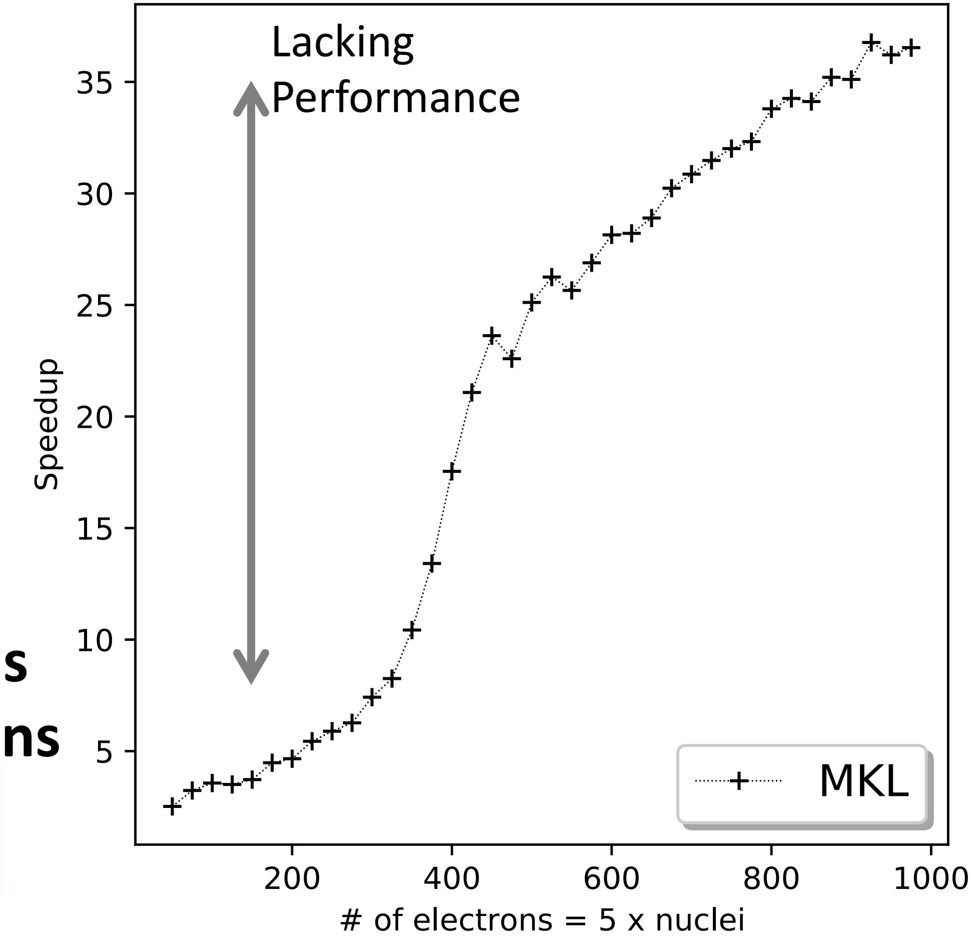
Final Speedup (vs Doc) → 35× for 1000 electrons

High-Performance Implementation: Jastrow Factor

Speedup for Jastrow Factor

- DGEMM based algorithm shows large speedup over naïve algorithm
- Automatic OpenMP based intra-node parallelization

Final Speedup (vs Doc) → 2× for 100 electrons
Final Speedup (vs Doc) → 35× for 1000 electrons



Outline

Introduction and Motivation

- Introduction: Biological N_2 reduction
- Phenomenological models

Development of Accurate and Efficient Algorithms

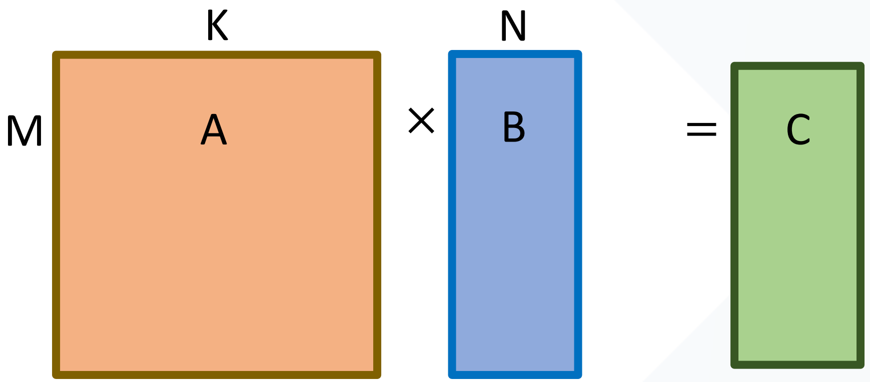
- Fast and Accurate Calculations: Jastrow Factor and DGEMM
- Conclusion

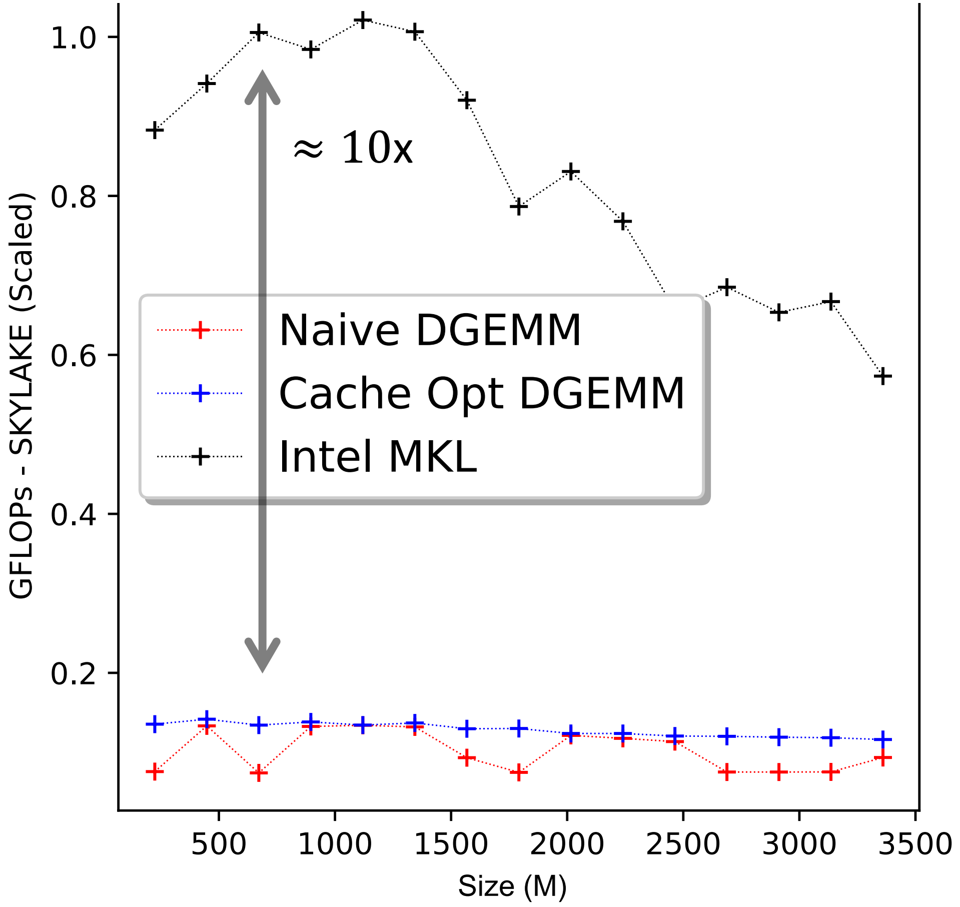
Jastrow Factor: Algorithmic Development

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

Naïve DGEMM vs Intel MKL

- Naïve DGEMM is 10x slower
- Performance worse especially for small sizes
- State of the art – Intel MKL

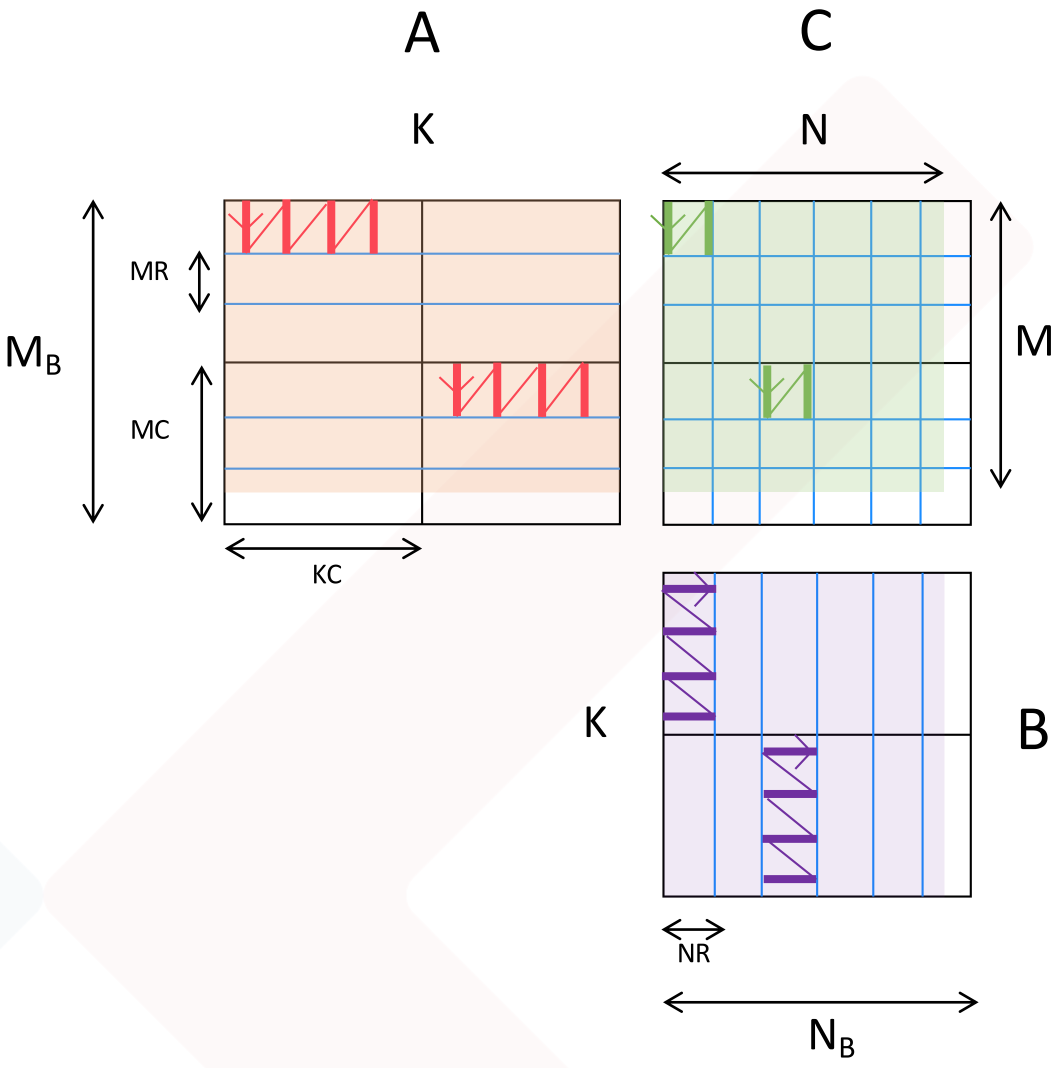




Jastrow Factor: Algorithmic Development

Hierarchical Data Layout

- Blocking of data
- Tiling based on Hardware Caches
- Highly efficient memory access
- Almost zero cache miss (prefetch)
- Aligned allocation of blocks and tiles
    - More information than MKL

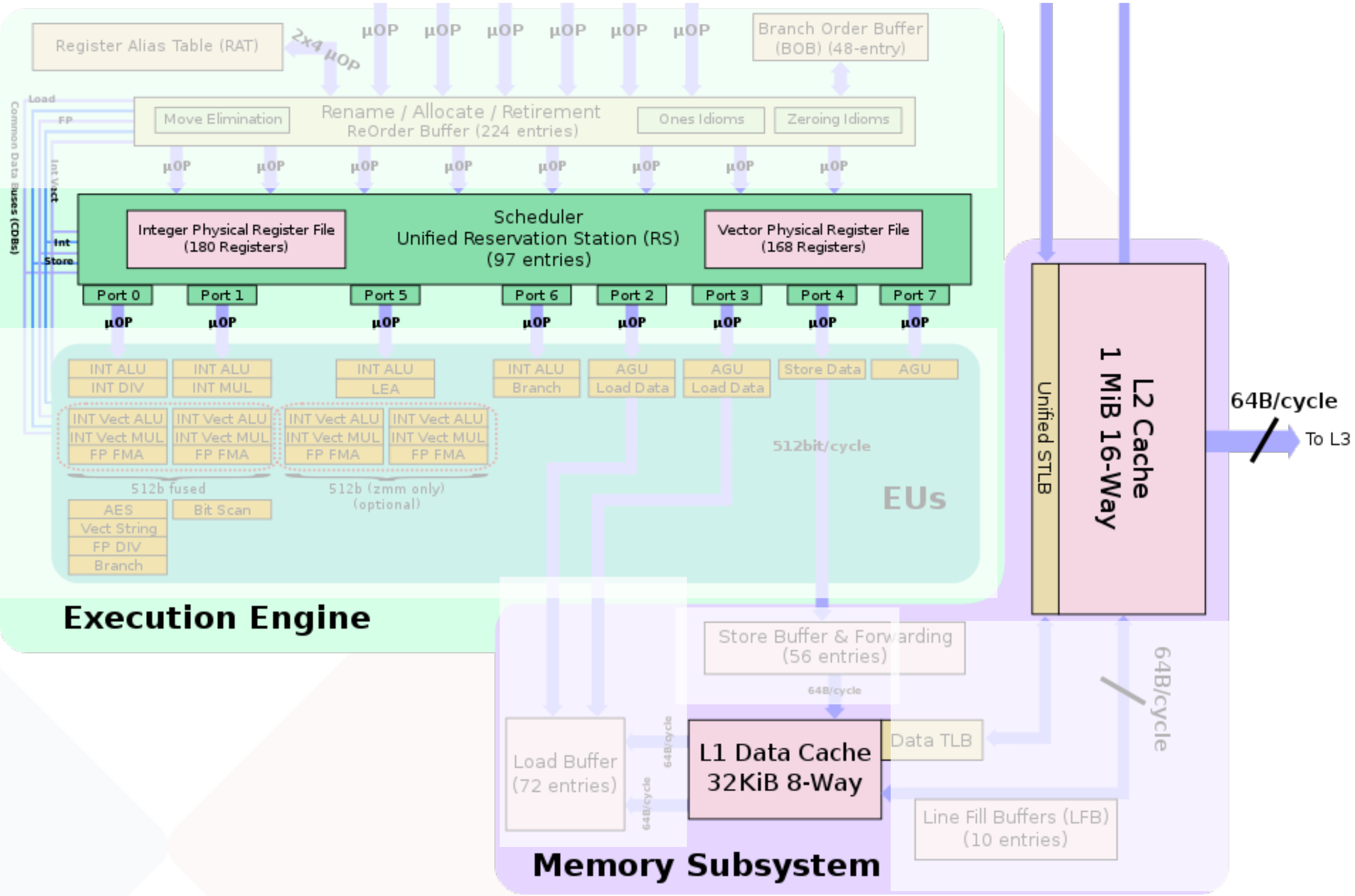


Goto, Kazushige, and Robert A. van de Geijn. *ACM Transactions on Mathematical Software (TOMS)* **2008,** 34.3, 1-25

Jastrow Factor: Algorithmic Development

Core Block Layout

- μArchitecture – Skylake
- Fast memory buffers
  - Cache Layout
  - L2 and L1 cache
  - Register file
- Port Layout



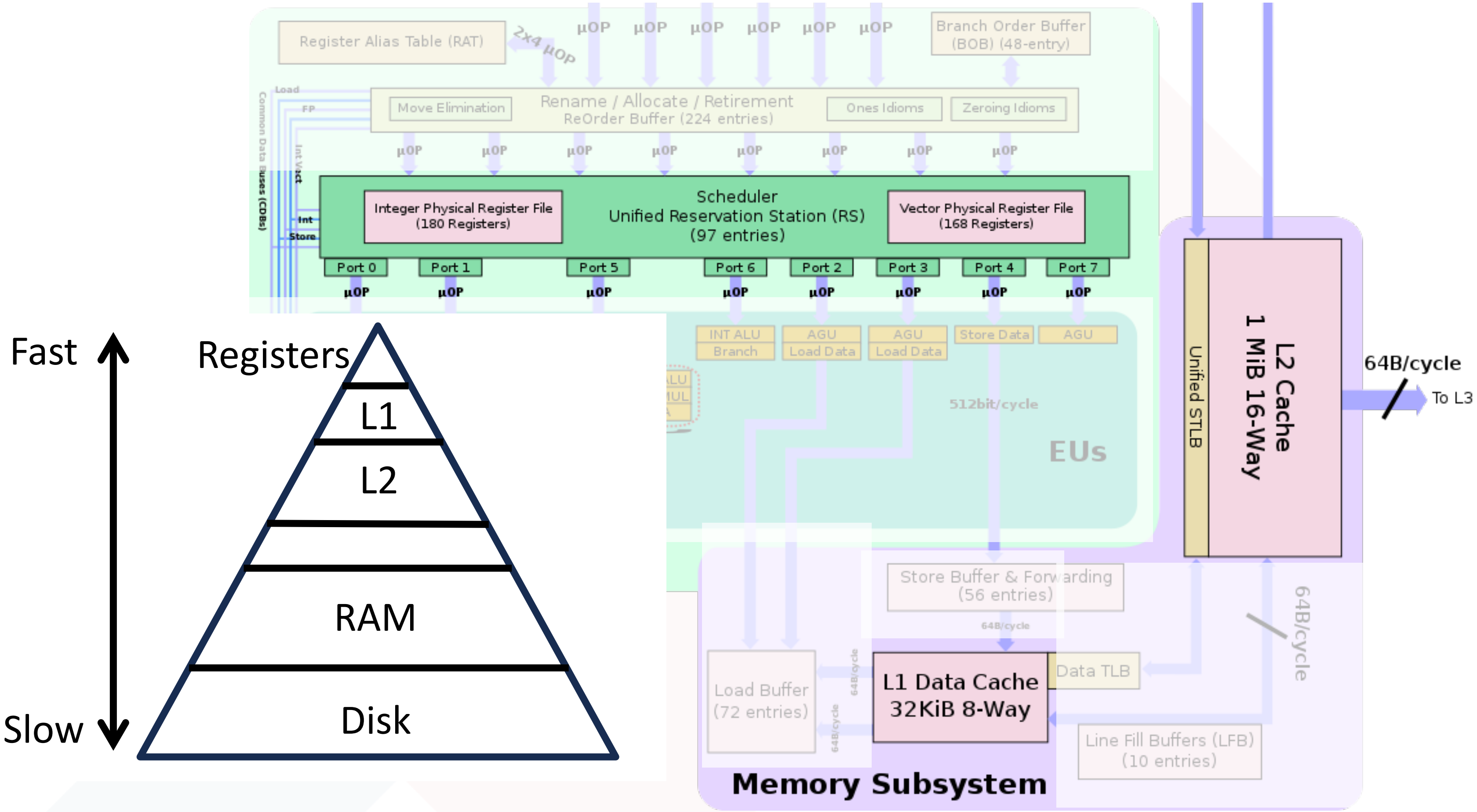
Goto, Kazushige, and Robert A. van de Geijn. *ACM Transactions on Mathematical Software (TOMS)* **2008,** 34.3, 1-25

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

Jastrow Factor: Algorithmic Development

Core Block Layout

- μArchitecture – Skylake
- Fast memory buffers
  - Cache Layout
  - L2 and L1 cache
  - Register file
- Port Layout



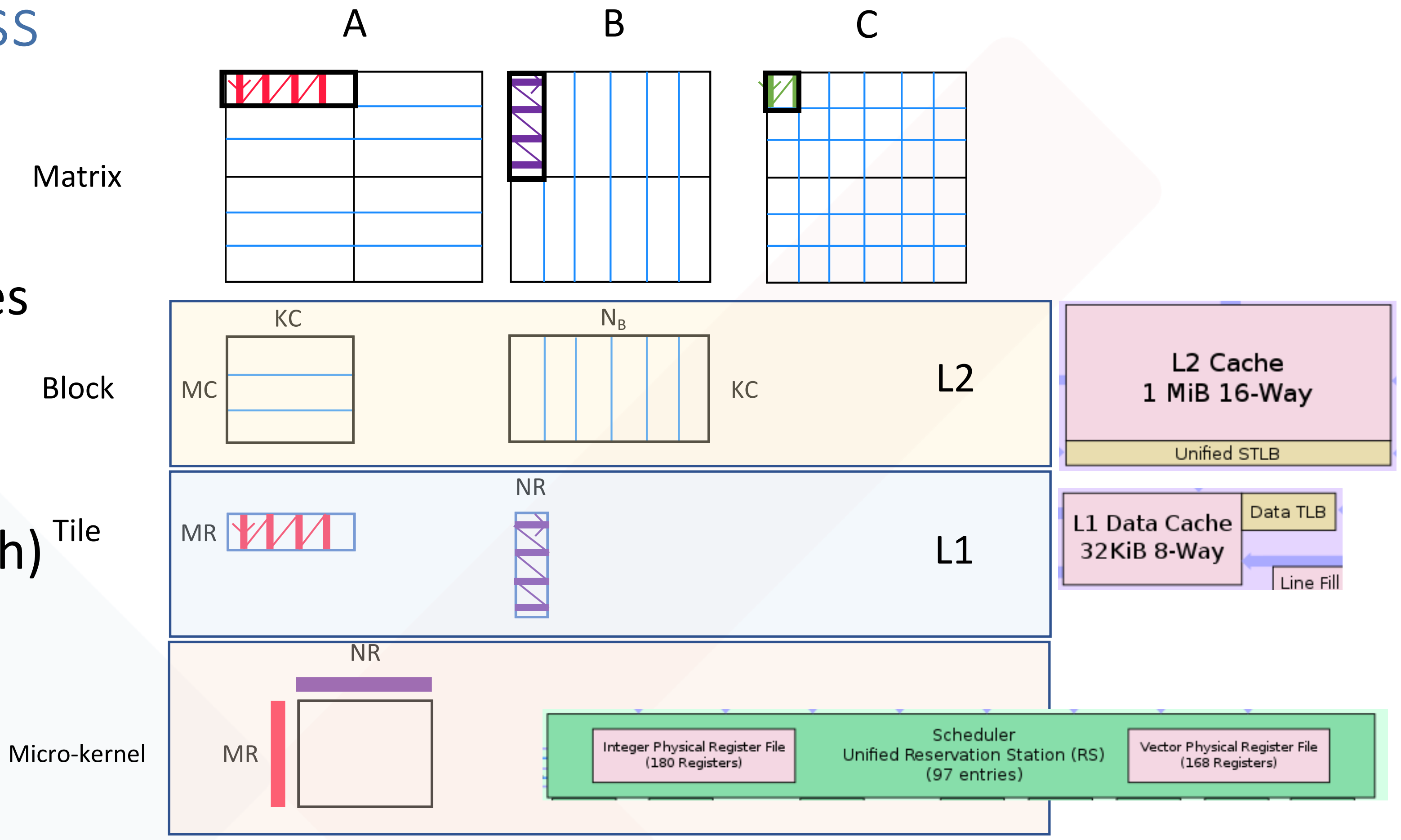
Goto, Kazushige, and Robert A. van de Geijn. *ACM Transactions on Mathematical Software (TOMS)* **2008,** 34.3, 1-25

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

Jastrow Factor: Algorithmic Development

Optimized Cache Access

- Blocking of data
- Tiling based on Hardware Caches
- Highly efficient memory access
- Almost zero cache miss (prefetch)

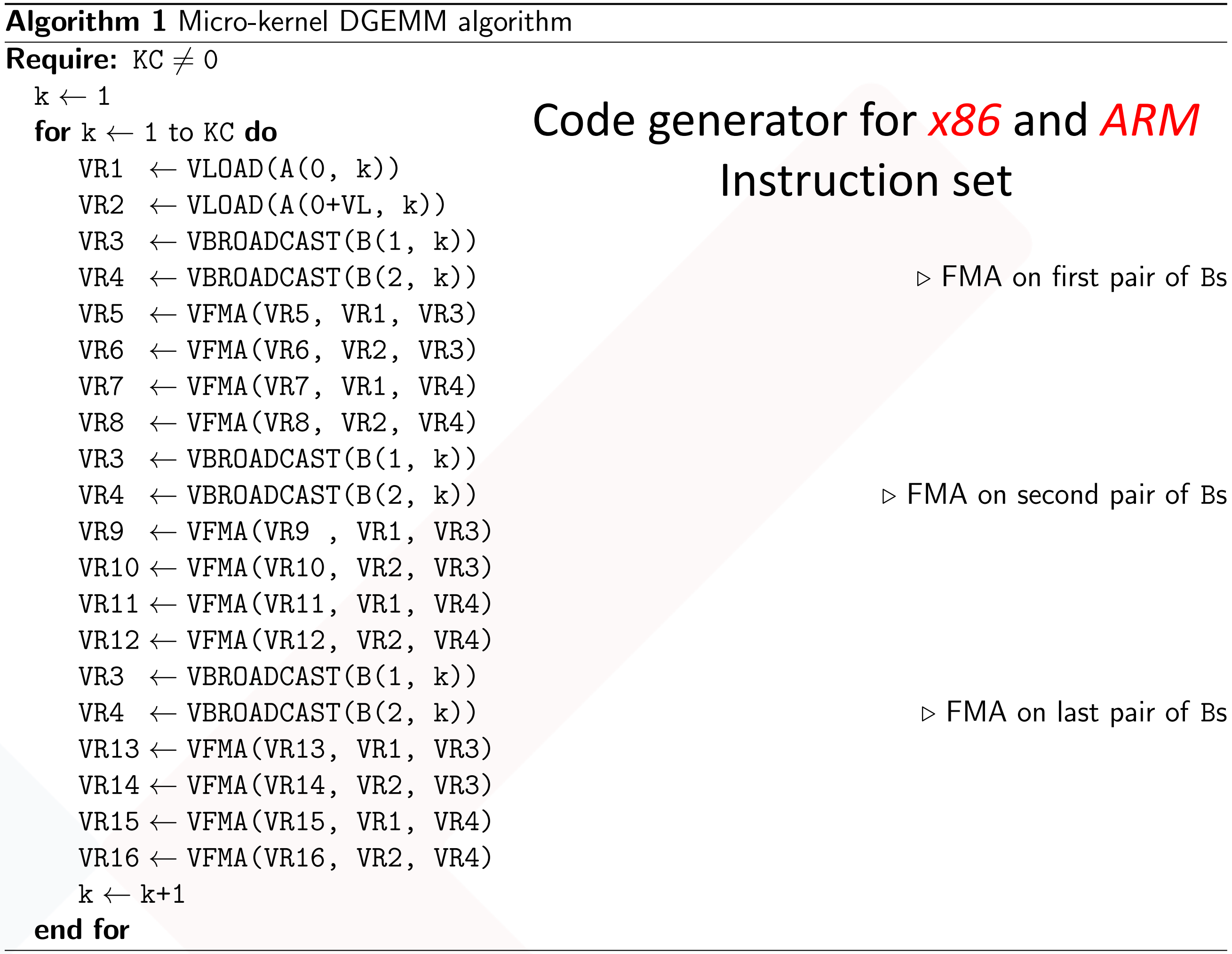


Goto, Kazushige, and Robert A. van de Geijn. *ACM Transactions on Mathematical Software (TOMS)* **2008,** 34.3, 1-25

Jastrow Factor: Algorithmic Development

Micro-Kernel : ASM Volatile

- ASM allows complete control over register allocation
- Portability ensured by code-generator *x86* and *RISC-V*
- Bypassing the compiler
- Compiler independent performance



Algorithm 1 Micro-kernel DGEMM algorithm

```
Require: KC ≠ 0
  k ← 1
  for k ← 1 to KC do
      VR1  ← VLOAD(A(0, k))
      VR2  ← VLOAD(A(0+VL, k))
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                    ▷ FMA on first pair of Bs
      VR5  ← VFMA(VR5, VR1, VR3)
      VR6  ← VFMA(VR6, VR2, VR3)
      VR7  ← VFMA(VR7, VR1, VR4)
      VR8  ← VFMA(VR8, VR2, VR4)
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                    ▷ FMA on second pair of Bs
      VR9  ← VFMA(VR9 , VR1, VR3)
      VR10 ← VFMA(VR10, VR2, VR3)
      VR11 ← VFMA(VR11, VR1, VR4)
      VR12 ← VFMA(VR12, VR2, VR4)
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                    ▷ FMA on last pair of Bs
      VR13 ← VFMA(VR13, VR1, VR3)
      VR14 ← VFMA(VR14, VR2, VR3)
      VR15 ← VFMA(VR15, VR1, VR4)
      VR16 ← VFMA(VR16, VR2, VR4)
      k ← k+1
  end for
```

Code generator for *x86* and *ARM* Instruction set

Jastrow Factor: Algorithmic Development

Micro-Kernel : ASM Volatile

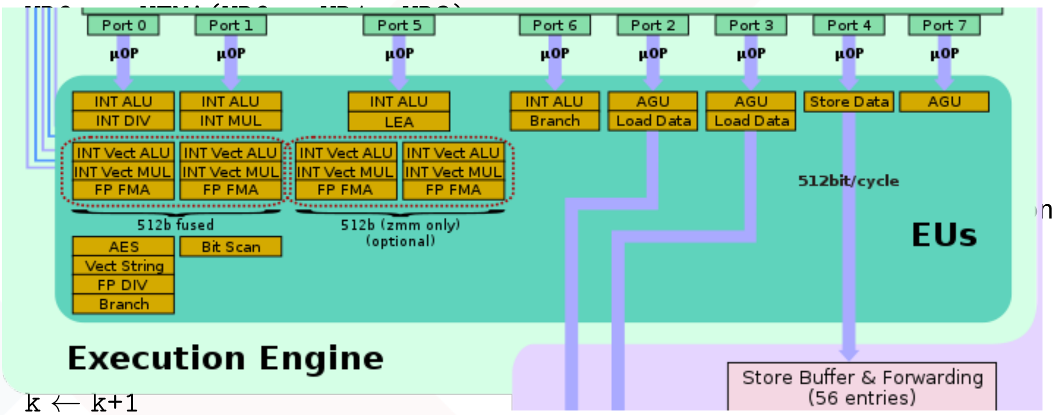
- ASM allows complete control over register allocation
- Portability ensured by code-generator *x86* and *ARM*
- Bypassing the compiler
- Compiler independent performance

Algorithm 1 Micro-kernel DGEMM algorithm

```
Require: KC ≠ 0
  k ← 1
  for k ← 1 to KC do
      VR1  ← VLOAD(A(0, k))
      VR2  ← VLOAD(A(0+VL, k))
      VR3  ← VBROADCAST(B(1, k))                                  ▷ FMA on first pair of Bs
      VR4  ← VBROADCAST(B(2, k))
      VR5  ← VFMA(VR5, VR1, VR3)
      VR6  ← VFMA(VR6, VR2, VR3)
      VR7  ← VFMA(VR7, VR1, VR4)
      VR8  ← VFMA(VR8, VR2, VR4)
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                                  ▷ FMA on second pair of Bs
                                                                   n last pair of Bs
      k ← k+1
  end for
```

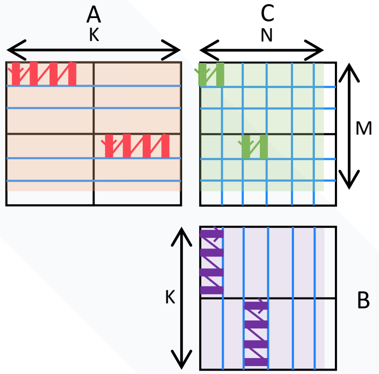
Ports – 2 and 3 (3/2)

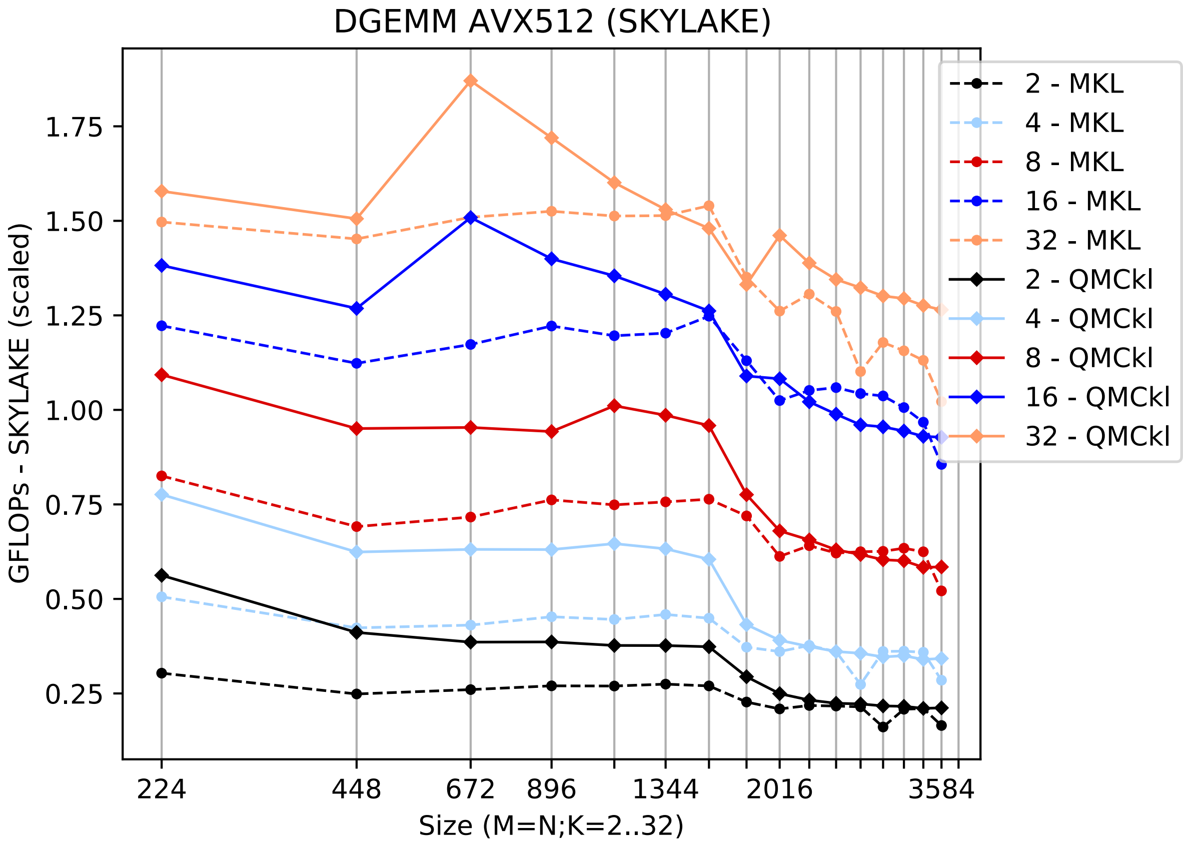
Ports – 0+1, 5 (4/2)



Comparison with MKL (Skylake) AVX512

- 2x Speedup for $200 < M = N < 500$
- Portability and Productivity
- Modular code
  - DGEMV $(u.A, A.v)$
  - Dot product





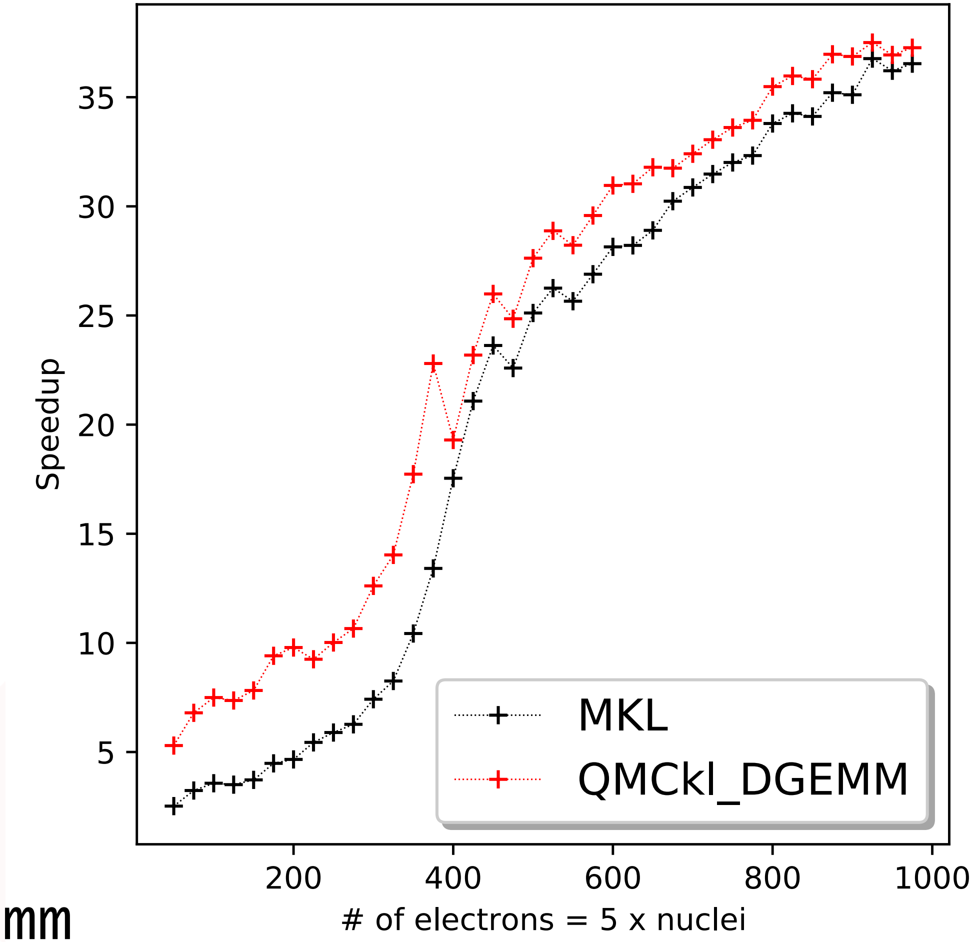
https://github.com/TREX-CoE/qmckl

Application to the Jastrow Factor

- DGEMM based algorithm shows large speedup over naïve algorithm
- QMCkl DGEMM gives further speedup for small # of electrons

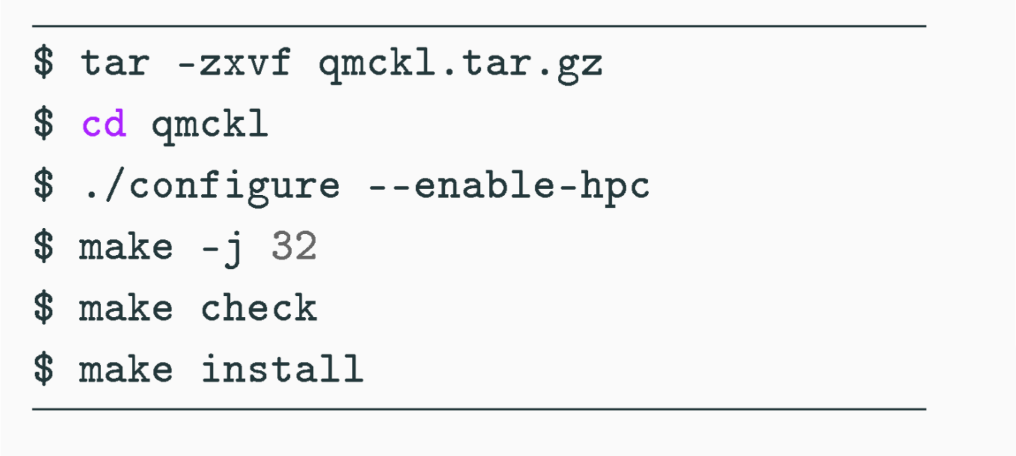
Final Speedup (vs Doc) → 5× for 100 electrons
Final Speedup (vs Doc) → 35× for 1000 electrons

```
https://github.com/TREX-CoE/qmckl
https://github.com/TREX-CoE/qmckl_dgemm
```



Quantum Monte Carlo kernel library (QMCkl)

python setup.py install

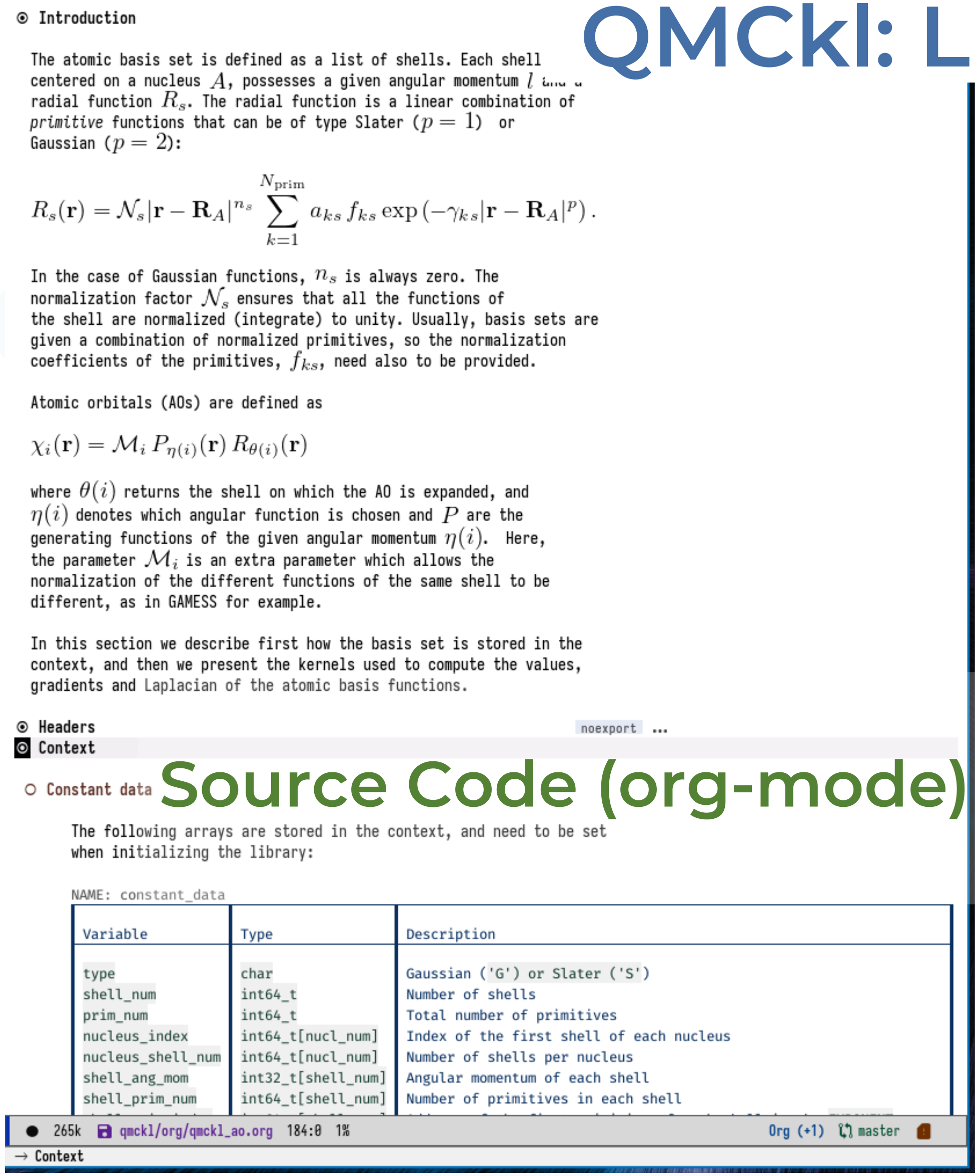


```
$ tar -zxvf qmckl.tar.gz
$ cd qmckl
$ ./configure --enable-hpc
$ make -j 32
$ make check
$ make install
```

- Very few dependencies
  - BLAS/LAPACK (CPU)
  - TREXIO (optional) and HDF5 (optional)
- BSD license: very permissive, you can distribute the `.tar.gz` with your code.
- Hosted on GitHub:

`https://github.com/trex-coe/qmckl`

QMCkl: Literate Programming



⦿ Introduction

The atomic basis set is defined as a list of shells. Each shell 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 *primitive* functions that can be of type Slater ($p = 1$) or Gaussian ($p = 2$):

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

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 (integrate) to unity. Usually, basis sets are given a combination of normalized primitives, so the normalization coefficients of the primitives, f_{ks}, need also to be provided.

Atomic orbitals (AOs) are defined as

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

where $\theta(i)$ returns the shell on which the AO is expanded, and $\eta(i)$ denotes which angular function is chosen and P are the generating functions of the given angular momentum $\eta(i)$. Here, the parameter $\mathcal{M}_i$ is an extra parameter which allows the normalization of the different functions of the same shell to be different, as in GAMESS for example.

In this section we describe first how the basis set is stored in the context, and then we present the kernels used to compute the values, gradients and Laplacian of the atomic basis functions.

⦿ Headers noexport ...

⦿ Context

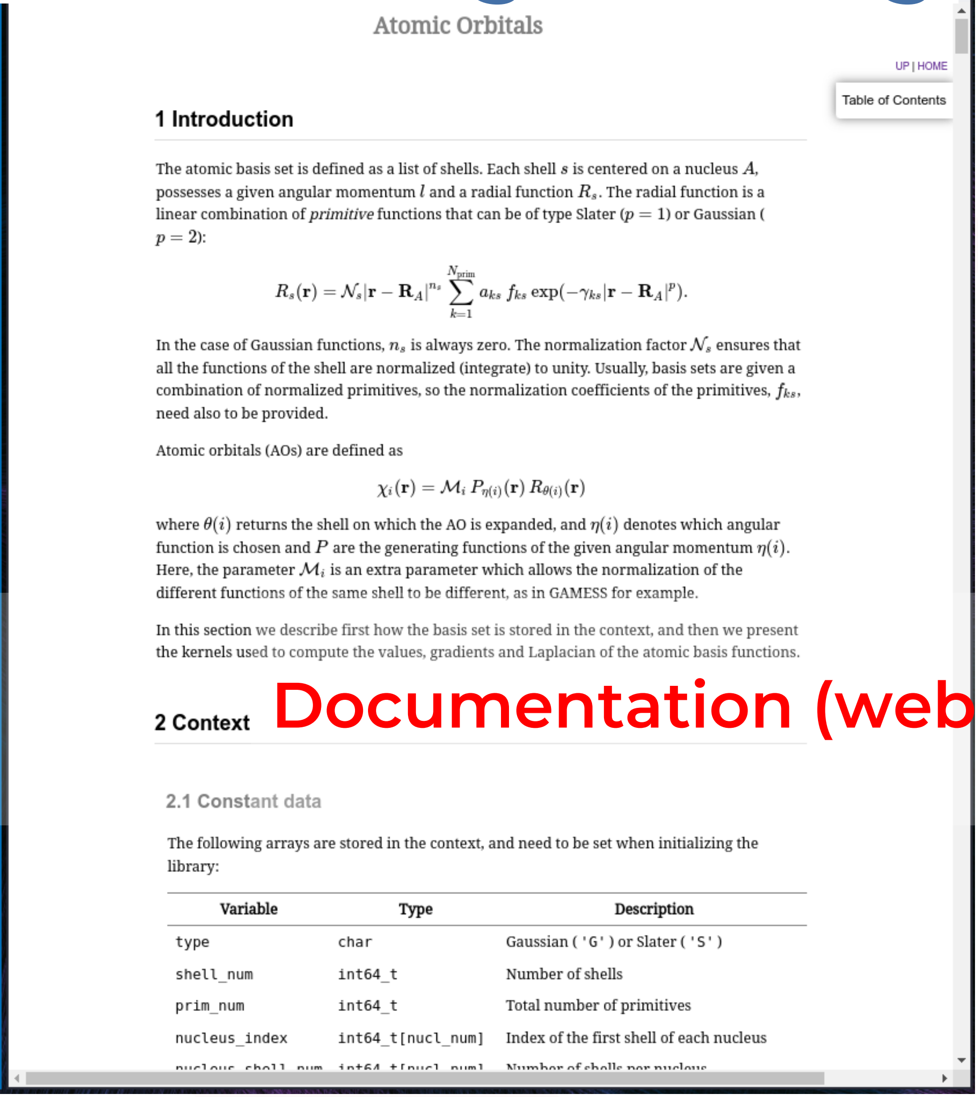
○ Constant data

The following arrays are stored in the context, and need to be set when initializing the library:

NAME: constant_data

Variable	Type	Description
type	char	Gaussian ('G') or Slater ('S')
shell_num	int64_t	Number of shells
prim_num	int64_t	Total number of primitives
nucleus_index	int64_t[nucl_num]	Index of the first shell of each nucleus
nucleus_shell_num	int64_t[nucl_num]	Number of shells per nucleus
shell_ang_mom	int32_t[shell_num]	Angular momentum of each shell
shell_prim_num	int64_t[shell_num]	Number of primitives in each shell

Source Code (org-mode)



Atomic Orbitals

1 Introduction

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 *primitive* functions that can be of type Slater ($p = 1$) or Gaussian ($p = 2$):

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

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 (integrate) to unity. Usually, basis sets are given a combination of normalized primitives, so the normalization coefficients of the primitives, f_{ks}, need also to be provided.

Atomic orbitals (AOs) are defined as

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

where $\theta(i)$ returns the shell on which the AO is expanded, and $\eta(i)$ denotes which angular function is chosen and P are the generating functions of the given angular momentum $\eta(i)$. Here, the parameter $\mathcal{M}_i$ is an extra parameter which allows the normalization of the different functions of the same shell to be different, as in GAMESS for example.

In this section we describe first how the basis set is stored in the context, and then we present the kernels used to compute the values, gradients and Laplacian of the atomic basis functions.

2 Context

2.1 Constant data

The following arrays are stored in the context, and need to be set when initializing the library:

Variable	Type	Description
type	char	Gaussian ('G') or Slater ('S')
shell_num	int64_t	Number of shells
prim_num	int64_t	Total number of primitives
nucleus_index	int64_t[nucl_num]	Index of the first shell of each nucleus
nucleus_shell_num	int64_t[nucl_num]	Number of shells per nucleus

Documentation (website)

Conclusions

Scientists Reference Code

```
do ii = 1, 4
   do j = 1, elec_num
      factor_een_gl(j,ii,nw) = factor_een_gl(j,ii,nw) + (          &
           tmp_c(j,a,m,k,nw)        * een_rescaled_n_gl(j,ii,a,m+l,nw) + &
           (dtmp_c(j,ii,a,m,k,nw))   * een_rescaled_n(j,a,m+l,nw)          + &
           (dtmp_c(j,ii,a,m+l,k,nw)) * een_rescaled_n(j,a,m  ,nw)          + &
           tmp_c(j,a,m+l,k,nw)      * een_rescaled_n_gl(j,ii,a,m,nw)    &
           ) * cn
   end do
end do
```

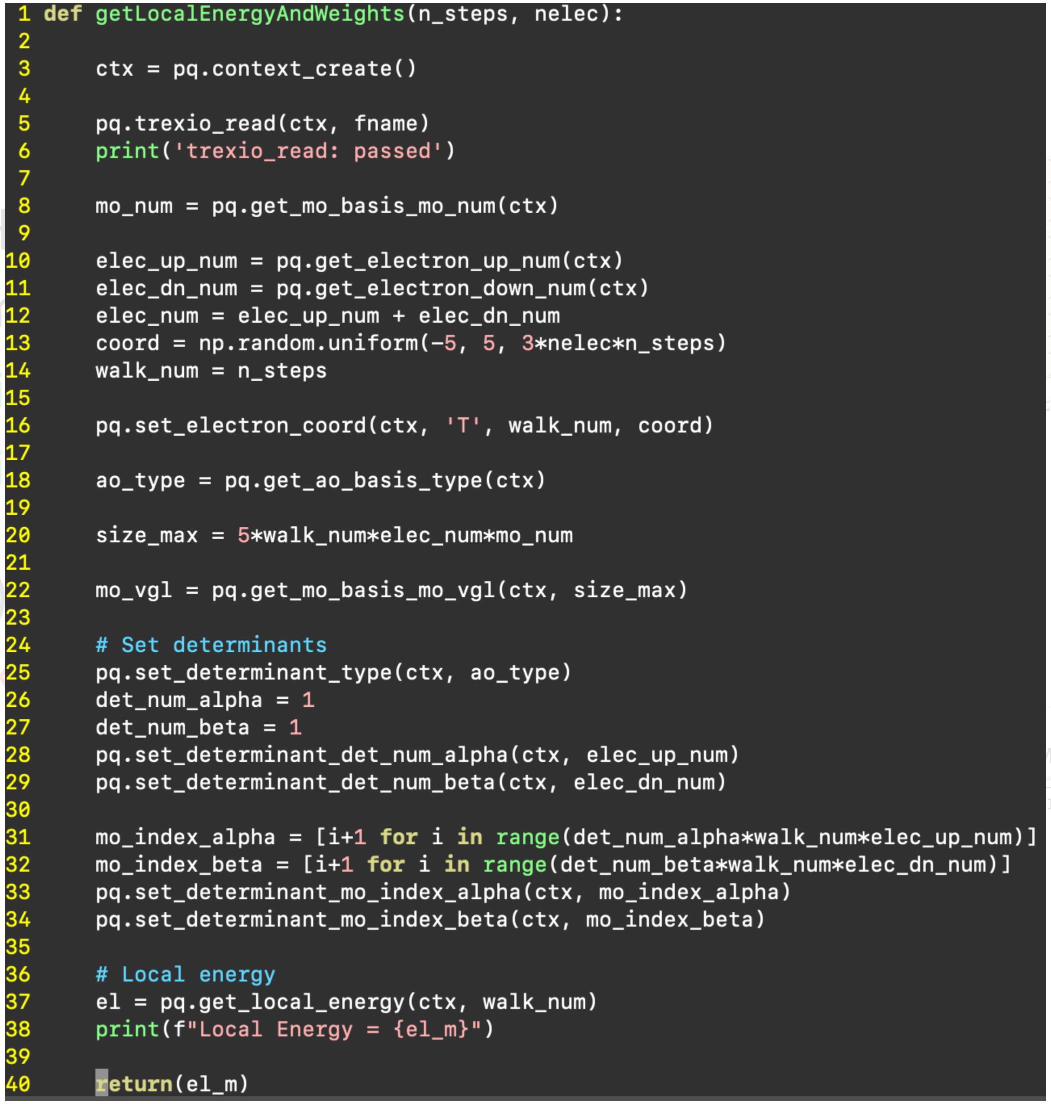
$$\langle E\rangle_{\Psi^2} = \sum_{R_i} E_L(R_i) \quad \text{with, } E_L(R) = \frac{(\hat{H}\Psi)(R)}{\Psi(R)}$$

```
do j = 1, elec_num
   factor_een_gl(j,4,nw) = factor_een_gl(j,4,nw) +  (               &
        (dtmp_c(j,1,a,m  ,k,nw)) * een_rescaled_n_gl(j,1,a,m+l,nw)  + &
        (dtmp_c(j,2,a,m  ,k,nw)) * een_rescaled_n_gl(j,2,a,m+l,nw)  + &
        (dtmp_c(j,3,a,m  ,k,nw)) * een_rescaled_n_gl(j,3,a,m+l,nw)  + &
        (dtmp_c(j,1,a,m+l,k,nw)) * een_rescaled_n_gl(j,1,a,m  ,nw)  + &
        (dtmp_c(j,2,a,m+l,k,nw)) * een_rescaled_n_gl(j,2,a,m  ,nw)  + &
        (dtmp_c(j,3,a,m+l,k,nw)) * een_rescaled_n_gl(j,3,a,m  ,nw)    &
        ) * cn
end do
```

$$\Psi_T = \mathcal{J}(e,e,n)\sum_{i=1}^{N_{det}} C_i D_i$$

- QMCkl: H
  developm
  - Scienti
  - Scienti
    algorit
    work")
- HPC Ex

(↓ complexity)



```python
def getLocalEnergyAndWeights(n_steps, nelec):

    ctx = pq.context_create()

    pq.trexio_read(ctx, fname)
    print('trexio_read: passed')

    mo_num = pq.get_mo_basis_mo_num(ctx)

    elec_up_num = pq.get_electron_up_num(ctx)
    elec_dn_num = pq.get_electron_down_num(ctx)
    elec_num = elec_up_num + elec_dn_num
    coord = np.random.uniform(-5, 5, 3*nelec*n_steps)
    walk_num = n_steps

    pq.set_electron_coord(ctx, 'T', walk_num, coord)

    ao_type = pq.get_ao_basis_type(ctx)

    size_max = 5*walk_num*elec_num*mo_num

    mo_vgl = pq.get_mo_basis_mo_vgl(ctx, size_max)

    # Set determinants
    pq.set_determinant_type(ctx, ao_type)
    det_num_alpha = 1
    det_num_beta = 1
    pq.set_determinant_det_num_alpha(ctx, elec_up_num)
    pq.set_determinant_det_num_beta(ctx, elec_dn_num)

    mo_index_alpha = [i+1 for i in range(det_num_alpha*walk_num*elec_up_num)]
    mo_index_beta = [i+1 for i in range(det_num_beta*walk_num*elec_dn_num)]
    pq.set_determinant_mo_index_alpha(ctx, mo_index_alpha)
    pq.set_determinant_mo_index_beta(ctx, mo_index_beta)

    # Local energy
    el = pq.get_local_energy(ctx, walk_num)
    print(f"Local Energy = {el_m}")

    return(el_m)
```

Thank you!

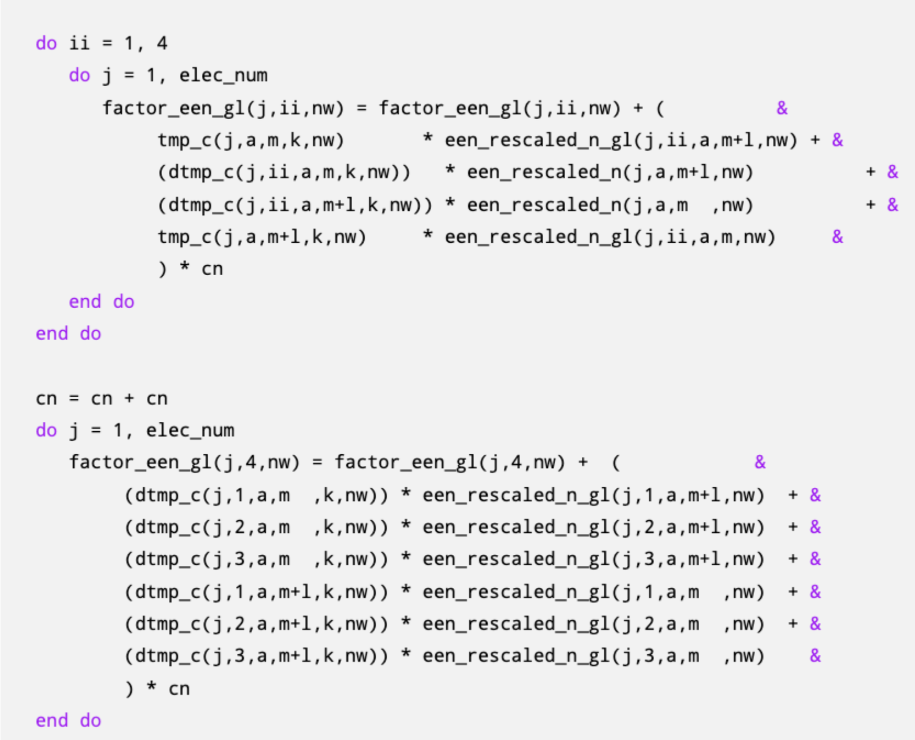
Follow us

in company/trex-eu

@trex_eu

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

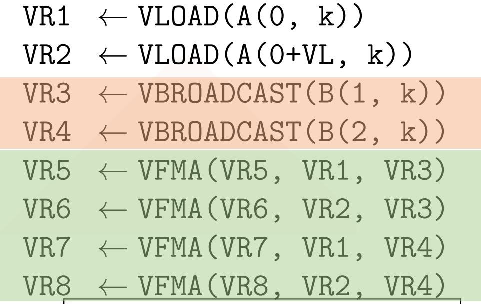
Scientists Reference Code



```
do ii = 1, 4
   do j = 1, elec_num
      factor_een_gl(j,ii,nw) = factor_een_gl(j,ii,nw) + (          &
           tmp_c(j,a,m,k,nw)       * een_rescaled_n_gl(j,ii,a,m+1,nw) + &
           (dtmp_c(j,ii,a,m,k,nw))  * een_rescaled_n(j,a,m+1,nw)        + &
           (dtmp_c(j,ii,a,m+1,k,nw)) * een_rescaled_n(j,a,m  ,nw)        + &
           tmp_c(j,a,m+1,k,nw)     * een_rescaled_n_gl(j,ii,a,m,nw)     &
           ) * cn
   end do
end do

cn = cn + cn
do j = 1, elec_num
   factor_een_gl(j,4,nw) = factor_een_gl(j,4,nw) +  (            &
        (dtmp_c(j,1,a,m  ,k,nw)) * een_rescaled_n_gl(j,1,a,m+1,nw)  + &
        (dtmp_c(j,2,a,m  ,k,nw)) * een_rescaled_n_gl(j,2,a,m+1,nw)  + &
        (dtmp_c(j,3,a,m  ,k,nw)) * een_rescaled_n_gl(j,3,a,m+1,nw)  + &
        (dtmp_c(j,1,a,m+1,k,nw)) * een_rescaled_n_gl(j,1,a,m  ,nw)  + &
        (dtmp_c(j,2,a,m+1,k,nw)) * een_rescaled_n_gl(j,2,a,m  ,nw)  + &
        (dtmp_c(j,3,a,m+1,k,nw)) * een_rescaled_n_gl(j,3,a,m  ,nw)    &
        ) * cn
end do
```

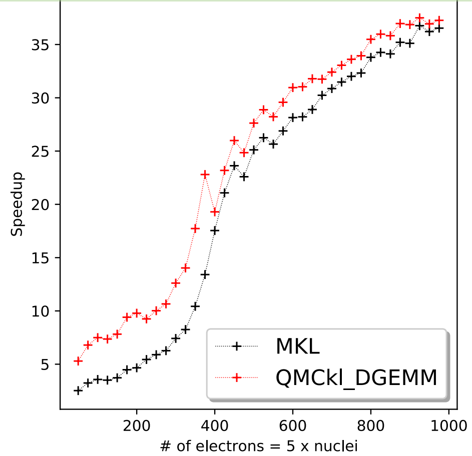
$$\Psi_T = \mathcal{J}(e,e,n) \sum_{i=1}^{N_{det}} C_i D_i$$

HPC Kernel



```
VR1  ← VLOAD(A(0, k))
VR2  ← VLOAD(A(0+VL, k))
VR3  ← VBROADCAST(B(1, k))
VR4  ← VBROADCAST(B(2, k))
VR5  ← VFMA(VR5, VR1, VR3)
VR6  ← VFMA(VR6, VR2, VR3)
VR7  ← VFMA(VR7, VR1, VR4)
VR8  ← VFMA(VR8, VR2, VR4)
```

- QMCkl: High Performance code development
  - Scientists: Documentation (code)
  - Scientists + HPC: Rewriting algorithm/equations (“LaTeX work”) for HPC
  - HPC Experts: Optimization



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

(↓ complexity)

with

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

(GEMM)

Further work and Perspectives

- Work in Progress for GPU based Jastrow factor calculation
- Collaborations for further work on GPUs:
  - Runtimes – StarPU with INRIA – Bordeaux
  - Performance Analysis MAQAO – UVSQ (William Jalby)
  - Blocking/Linear Algebra for GPU – Chameleon with INRIA – Bordeaux

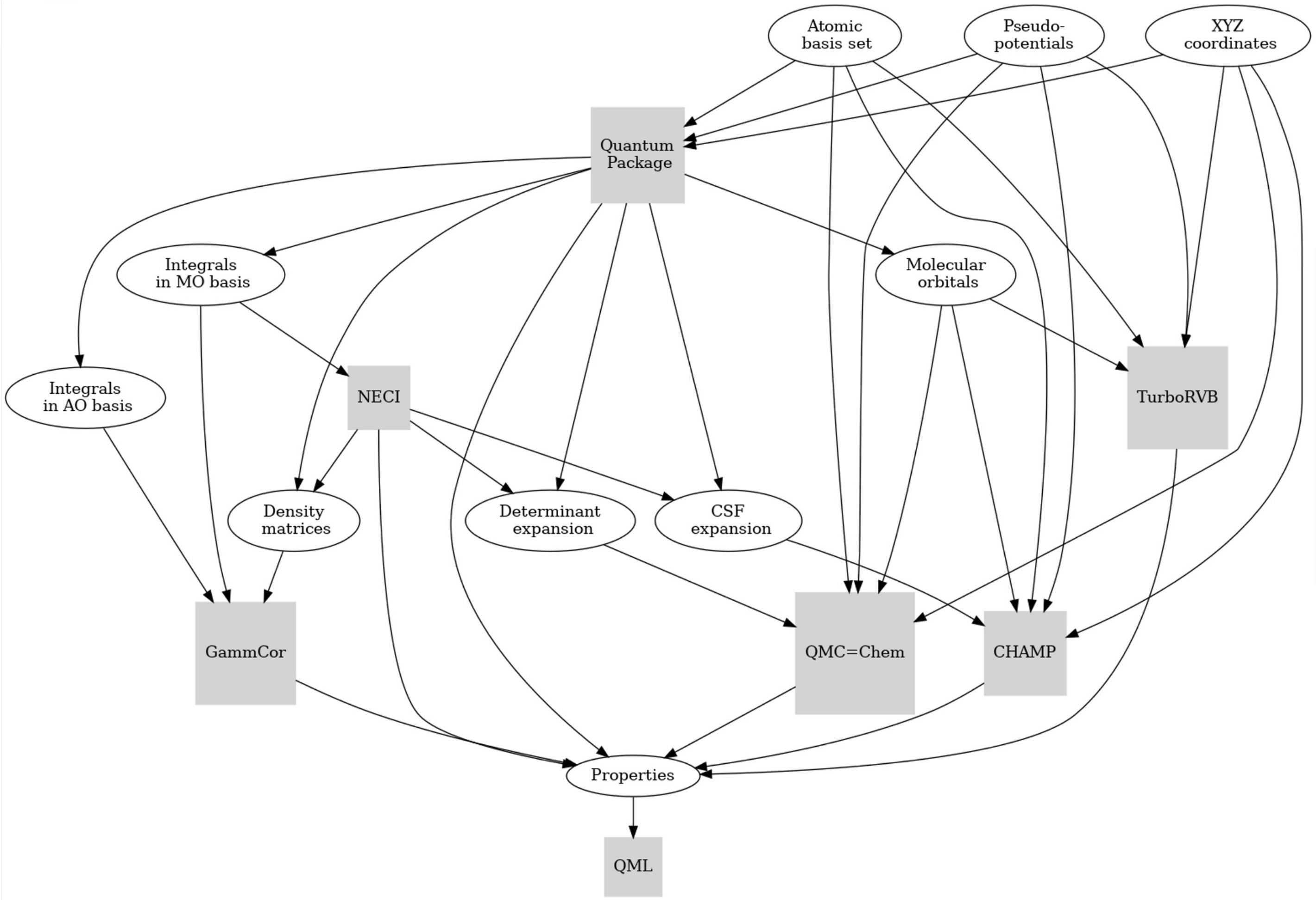
Introduction and Motivation

- Introduction: Biological N_2 reduction
- Phenomenological models

Conclusion and Perspectives

- Fast and Accurate Calculations: Jastrow Factor and DGEMM
- Conclusions

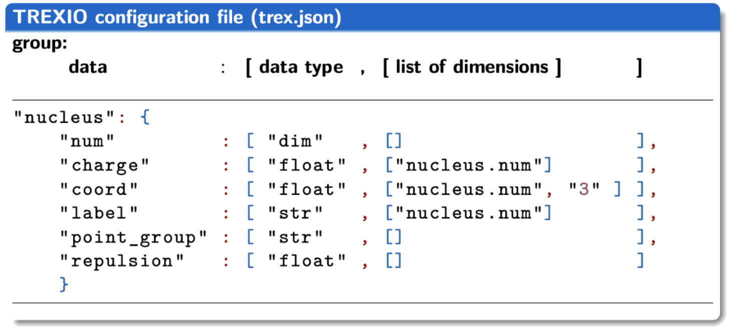
TREXIO for Quantum Chemistry data



⬭ - Data

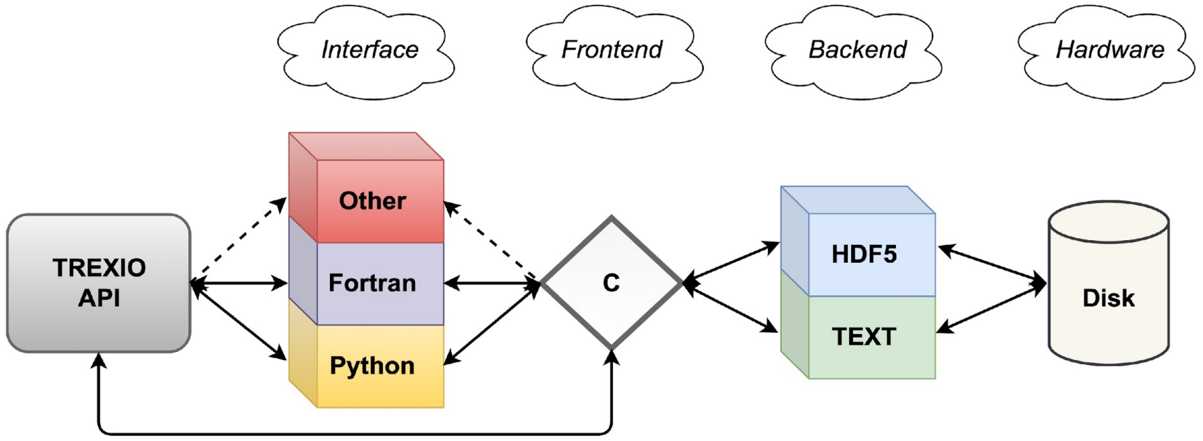
■ - Application

TREXIO for Quantum Chemistry data



TREXIO configuration file (trex.json)

```
group:
      data            :  [ data type  ,  [ list of dimensions ]          ]

"nucleus": {
    "num"         : [ "dim"   , []                           ],
    "charge"      : [ "float" , ["nucleus.num"]              ],
    "coord"       : [ "float" , ["nucleus.num", "3" ]        ],
    "label"       : [ "str"   , ["nucleus.num"]              ],
    "point_group" : [ "str"   , []                           ],
    "repulsion"   : [ "float" , []                           ]
    }
```



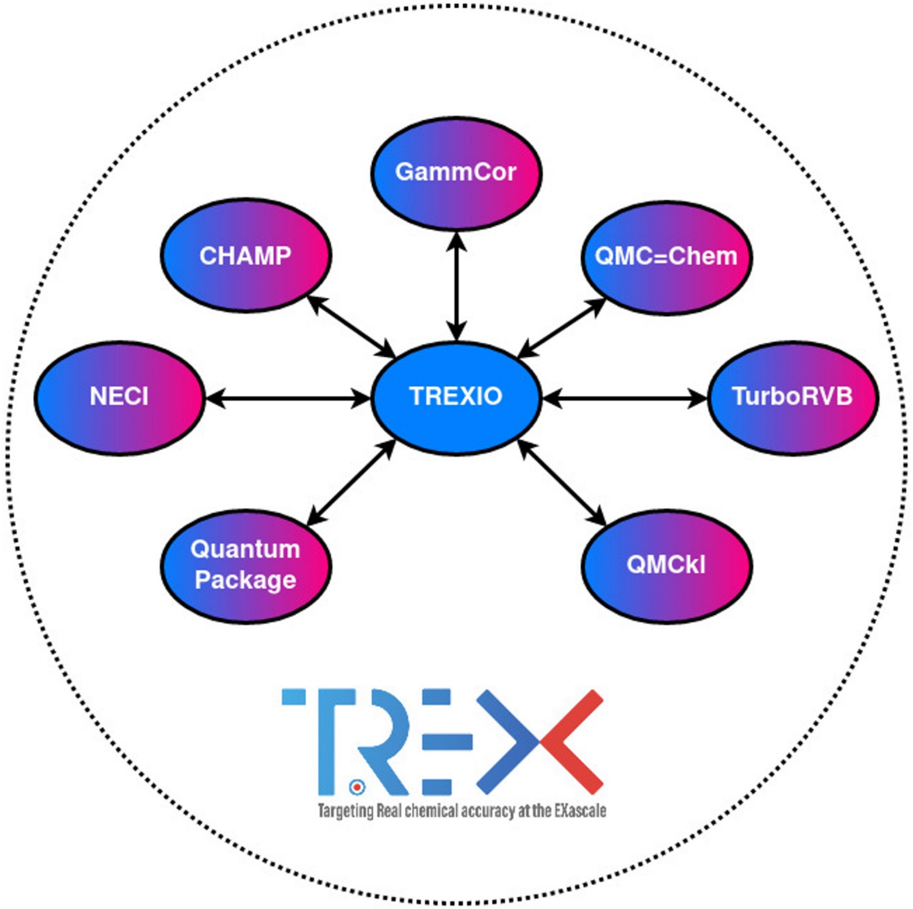
- **Self-Consistent**: Self-contained No external knowledge required
- AOs: **Cartesian**, **Spherical**, **Numerical**, etc…
- Compact Storage: **2e Integrals, CI coefficients (Det, CSFs)**
- Source code in pure **C** (C99): Best performance/ portability
- Performant HDF5 backends for **parallel I/O**
- Interfaces: **Fortran, Python, Ocaml, Julia, Rust**

More details in the TREXIO documentation*

Posenitskiy, E., Chilkuri, V. G., Ammar, A., Hapka, M., Pernal, K., Shinde, R., ... & Scemama, A. (**2023**). *The Journal of chemical physics*, *158*(17).

* *https://trex-coe.github.io/trexio/trex.html*

TREXIO Today


GammCor
CHAMP
QMC=Chem
NECI
TREXIO
TurboRVB
Quantum Package
QMCkl
TREX
Targeting Real chemical accuracy at the EXascale