Using BLIS for tensor computations in Q-Chem

Evgeny Epifanovsky
Q-Chem

BLIS Retreat, September 19–20, 2016

Q-Chem is an integrated software suite for modeling the properties of molecular systems from first principles.

Density functional theory

- Pure and hybrid functionals
- Range-separated functionals
- Double hybrid functionals
- Time-dependent DFT

Wavefunction theory

- Hartree–Fock theory
- Møller–Plesset perturbation theory
- Coupled-cluster theory
- Methods for excited states: CIS, ADC, EOM, ...

Sustainability model:

- Commercial licenses for research in academia and industry (includes technical support)
- Free developer licenses for using Q-Chem as a development platform (200+ contributors)

Outline

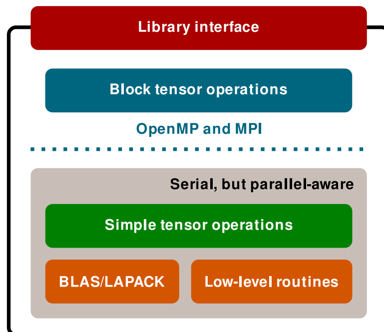
- Libtensor as a programming tool for tensor-based electronic structure methods
- Block tensor format for storing and manipulating tensors
- Block tensor contraction algorithm
- Interface with BLIS and results

Coupled cluster theory methods in Q-Chem

	Ground state	Excited state	Properties
2010–2012	MP2 QCISD CCD, CCSD CCSD(T), (dT), (fT)	CISD EOM-CCSD (EA, EE, IP, SF, DIP, DSF) IP-CISD, EA-CISD	OPDM, TPDM Properties (all methods) Gradient (CC, EOM)
2013	RI-CCSD	RI-EOM-CCSD (EA, EE, IP, SF)	RI-OPDM, RI-TPDM RI properties
2013–2015	CS/CX-MP2 CS/CX-CCSD	CS/CX-CISD CS/CX-EOM-CCSD (EA, EE, IP, SF)	CS/CX-OPDM Real, complex Dyson orbitals Two-photon absorption Spin-orbit coupling
2016	CCVB-SD	(RI-)EOM-MP2 (EA, EE, IP, SF)	Polarizabilities for CC and EOM wavefunctions

- Over 1000 programmable expressions implemented
- 17 contributors to date

Overview of libtensor



- Declarative internal domain-specific language for programming tensor expressions in C++
- Multiple back-ends for tensor computations:
  - built-in disk/memory-based
  - disk-based contraction algorithm via libxm (I. Kaliman)
  - distributed memory via CTF library (E. Solomonik)

Coupled cluster programmable expressions

$$T_{ij}^{ab} D_{ij}^{ab} = \langle ij || ab \rangle + \mathcal{P}_{-}(ab) \left(\sum_{c} f_{bc} t_{ij}^{ac} - \frac{1}{2} \sum_{klcd} \langle kl || cd \rangle t_{kl}^{bd} t_{ij}^{ac} \right) - \mathcal{P}_{-}(ij) \left(\sum_{k} f_{jk} t_{ik}^{ab} + \frac{1}{2} \sum_{klcd} \langle kl || cd \rangle t_{jl}^{cd} t_{ik}^{ab} \right)$$
$$+ \frac{1}{2} \sum_{kl} \langle ij || kl \rangle t_{kl}^{ab} + \frac{1}{4} \sum_{klcd} \langle kl || cd \rangle t_{ij}^{cd} t_{kl}^{ab} + \frac{1}{2} \sum_{cd} \langle ab || cd \rangle t_{ij}^{cd} - \mathcal{P}_{-}(ij) \mathcal{P}_{-}(ab) \left(\sum_{kc} \langle kb || jc \rangle t_{ik}^{ac} - \frac{1}{2} \sum_{klcd} \langle kl || cd \rangle t_{lj}^{db} t_{ik}^{ac} \right)$$

$$\mathcal{P}_{-}(ij) A_{ij} = A_{ij} - A_{ji} \qquad D_{ij}^{ab} = \epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b$$

```
f1_oo(i|j) = f_oo(i|j) + 0.5 * contract(k|a|b, i_oovv(j|k|a|b), t2(i|k|a|b));
f1_vv(b|c) = f_vv(b|c) - 0.5 * contract(k|l|d, i_oovv(k|l|c|d), t2(k|l|b|d));
ii_oooo(i|j|k|l) = i_oooo(i|j|k|l) + 0.5 * contract(a|b, i_oovv(k|l|a|b), t2(i|j|a|b));
ii_ovov(i|a|j|b) = i_ovov(i|a|j|b) - 0.5 * contract(k|c, i_oovv(i|k|b|c), t2(k|j|c|a));

t2new(i|j|a|b) = div(
      i_oovv(i|j|a|b)
    + asymm(a, b, contract(c, t2(i|j|a|c), f1_vv(b|c)))
    - asymm(i, j, contract(k, t2(i|k|a|b), f1_oo(j|k)))
    + 0.5 * contract(k|l, ii_oooo(i|j|k|l), t2(k|l|a|b))
    + 0.5 * contract(c|d, i_vvvv(a|b|c|d), t2(i|j|c|d))
    - asymm(a, b, asymm(i, j, contract(k|c, ii_ovov(k|b|j|c), t2(i|k|a|c)))),
      d_oovv(i|j|a|b));
```

Evaluation of expressions

$$t_{ij}^{ab} = \mathcal{P}(ij)\mathcal{P}(ab)\sum_{kc} I_{kbic}^{(1)} t_{jk}^{ac} + \mathcal{P}(ij)\sum_{c}\langle jc||ba\rangle t_i^c$$

$$\downarrow$$

```
t2(i|j|a|b) =
     asymm(i, j, asymm(a, b, contract(k|c, i1(k|b|i|c), t2(j|k|a|c))))
   + asymm(i, j, contract(c, ovvv(j|c|b|a), t2(i|c)));
```

$$\downarrow$$

```
{ = t2(i,j,a,b) { +
    { asym(i,j; a,b) { * i1(k,b,i,c) t2(j,k,a,c) } }
    { asym(i,j) { * ovvv(j,c,b,a) t1(i,c) } }
} }
```

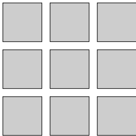
$$\downarrow$$

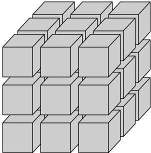
Factorization and optimization

$$\downarrow$$

Rendering as basic tensor operations and execution

Block tensor format

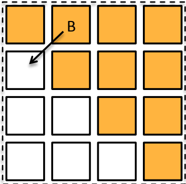




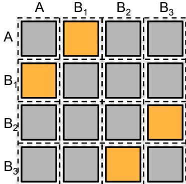
Three components:

1. Block tensor space: dimensions + tiling pattern.
2. Symmetry relations between blocks.
3. Non-zero canonical data blocks.

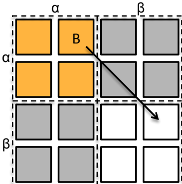
- Natural for block-sparse tensors.
- Imbalance in diagonal blocks with permutational symmetry.



Permutational



Point group



Spin

Block tensor contraction algorithm

$$C_{ijba} = \sum_{kc} A_{kbic} B_{jkac}$$

$$\downarrow$$

$$\tilde{C}_{ibja} = \sum_{kc} \tilde{A}_{ibkc} \tilde{B}_{jakc}$$

$$\downarrow$$

$$\tilde{C}_{IJ} = \sum_{K} \tilde{A}_{IK} \tilde{B}_{JK}$$

1. Compute the symmetry of C analytically
2. Compute unfolding transformations for A, B, C
3. Perform block tensor contraction of $\tilde{A}$, $\tilde{B}$ into $\tilde{C}$

Block tensor contraction algorithm

$$C_{IJ} = \sum_K A_{IK} B_{JK}$$

1. Compute the list of non-zero canonical blocks in C:
   - For each K, take sparse "column"-"row" outer products symbolically, reduce into a list of non-zero block indices
   - Filter out non-canonical block indices
2. Compute non-zero canonical blocks of C in parallel (one task per block):
   - Given (I, J), compute list of K that give non-zero products
   - Using symmetry of A and B, convert (I, K) and (J, K) into canonical (IK', Ta), (JK', Tb)
   - Optimize list of blockwise contractions to avoid duplication
   - Evaluate $C_{IJ} = \sum (T_a A_{IK'})(T_b B_{JK'})$ using the dense tensor contraction algorithm

Dense tensor contraction algorithm

Dense tensor contractions via loop merge followed by loop-over-GEMM:

$$C_{[ib][ja]} = \sum_{[kc]} A_{[ib][kc]} B_{[ja][kc]}$$

1. Reduce the number of nested loops by merging tensor indices
2. Match inner loops against available kernels (BLAS primitives)
3. Evaluate loops-over-kernel

Integration with BLIS:
enable use of BLIS routines as linear algebra primitives

libtensor+BLIS vs. libtensor+MKL

Timings (CPU sec.) for a UCCSD T amplitude update in water clusters, cc-pVDZ.
NERSC Cori: 2x16-core 2.3 GHz Haswell Xeon E5-2698, 128 GB DDR4 2133 MHz

	W8		W10		W12	
	MKL	BLIS	MKL	BLIS	MKL	BLIS
BS=16						
dgemm	436	534	1415	1681	3596	4203
- pack		130		470		1109
- kernel		373		1115		2865
dcopy	46	48	105	107	207	227
BS=32						
dgemm	404	461	1272	1401	3434	4167
- pack		54		143		423
- kernel		397		1235		3674
dcopy	61	67	140	168	319	321

Note on BLAS primitives

Given two vectors, it is possible to perform:

- Inner product (dot): $c = \sum_i a_i b_i$
- Outer product (ger): $c_{ij} = a_i b_j$
- Pointwise product: $c_i = a_i b_i$

axpy = inner product + translation: $c_i = \sum_k a_{ik} b_k$

gemm = inner product + outer product $c_{ij} = \sum_k a_{ik} b_{jk}$

inner product + pointwise product? $c_i = \sum_k a_{ik} b_{ik}$

all three? $c_{ijk} = \sum_m a_{ikm} b_{jkm}$