JCTC Journal of Chemical Theory and Computation

Article

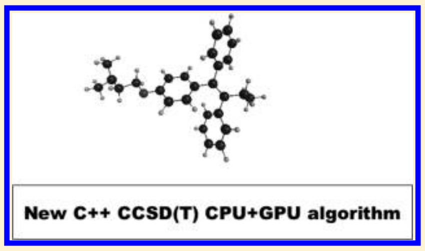
pubs.acs.org/JCTC

Fast and Flexible Coupled Cluster Implementation

Andrey Asadchev and Mark S. Gordon*

Department of Chemistry, Iowa State University, Spedding Hall, Ames, Iowa 50011-2030, United States

ABSTRACT: A new coupled cluster singles and doubles with triples correction, CCSD(T), algorithm is presented. The new algorithm is implemented in object oriented C++, has a low memory footprint, fast execution time, low I/O overhead, and a flexible storage backend with the ability to use either distributed memory or a file system for storage. The algorithm is demonstrated to work well on single workstations, a small cluster, and a high-end Cray computer. With the new implementation, a CCSD(T) calculation with several hundred basis functions and a few dozen occupied orbitals can run in under a day on a single workstation. The algorithm has also been implemented for graphical processing unit (GPU) architecture, giving a modest improvement. Benchmarks are provided for both CPU and GPU hardware.



1. INTRODUCTION

As a rule of thumb, the electronic energy obtained with the Hartree−Fock method accounts for ~99% of the energy. However, many chemical properties of interest are dependent on the remaining 1%, frequently called the electron correlation energy, or simply the correlation energy. The correlation energy is defined as the difference between the reference Hartree−Fock energy and the true energy,

$$E_{\text{corr}} = E_{\text{HF}} - E \tag{1}$$

Of the many electron correlation methods,[1−3] the coupled cluster (CC) method is one of the most successful. The coupled cluster method was first developed by nuclear physicists,[4] adapted to quantum chemistry by Cizek, Paldus, Shavitt, Mukherjee, Schaefer, and others[5−9] and especially popularized by Bartlett.[10]

The iterative singles and doubles coupled cluster (CCSD), plus triples that are included perturbatively,[11] CCSD(T), method is the most popular approach. The CCSD(T) method is often referred to as the gold standard of computational chemistry, among the several other higher-order methods.[12]

The coupled cluster method is usually introduced in the exponential *ansatz* form

$$\Psi = e^{T}\Psi_0 = e^{(T_1 + T_2 + \ldots T_n)}\Psi_0 \tag{2}$$

where $T_1 \ldots T_n$ are the *n*-particle cluster operators and Ψ_0 is the reference wave function, typically the Hartree−Fock reference Ψ_{HF}.

The excitation operator applied to a reference wave function is written in terms of cluster excitation amplitudes *t* from hole states *i*, *j*, *k*, ... (occupied orbitals in chemistry parlance) to particle states (or virtual orbitals), *a*, *b*, *c*, ...

$$T_n\Psi_0 = \sum_{ijk\ldots}\sum_{abc\ldots} t_{ijk\ldots}^{abc\ldots}\Psi_{ijk\ldots}^{abc\ldots} \tag{3}$$

Truncating the expansion at doubles leads to the approximate coupled cluster singles and doubles method, CCSD,[10]

$$T \approx T_1 + T_2 \tag{4}$$

The singles t_i^a and doubles t_{ij}^{ab} amplitudes are found by solving a system of nonlinear equations

$$\langle \Phi_i^a | (H_N e^{T_1 + T_2}) | \Phi \rangle = 0 \tag{5}$$

$$\langle \Phi_{ij}^{ab} | (H_N e^{T_1 + T_2}) | \Phi \rangle = 0 \tag{6}$$

where Φ, Φ_i^a, and Φ_{ij}^{ab} are, respectively, the reference determinant, and the singly and doubly excited determinants, and $H_N = H - \langle \Phi | H | \Phi \rangle$ is the normal order Hamiltonian, constructed so that its reference energy is zero.

The final algebraic CC equations, derived using a diagrammatic approach, result in a number of integral terms, *V*, contracted with *T* amplitudes. For example, VT_1^2 signifies integral terms contracted with $t_i^a t_j^b$. For the purposes of this work, the spin-free equations by Piecuch and co-workers[13] are used. One goal of this work is to develop a CCSD(T) implementation that is sufficiently flexible to run efficiently on a single node, a modest cluster, or a supercomputer.

The algebraic CC equations are presented in Einstein summation terminology, in which repeated co- and contra-variant indices; e.g., the index *s* in $X_s^r X_t^s$ or the index *r* in $X_s^r Y_r^t$ imply summation. For the following discussion, the one-electron integrals are defined as $f_q^p = \langle p|f|q \rangle$, the two-electron molecular integrals $v_{rs}^{pq} = \langle pq|v|rs \rangle$, and the many-body denominators $D_{qs\ldots}^{pr\ldots} = f_q^q + f_s^s + \ldots - f_p^p - f_r^r - \ldots$ for an arbitrary number of orbitals. Now, the CCSD nonlinear equations may be expressed as follows:

$$\begin{aligned} D_i^a t_i^a &= f_i^a + I_e^a t_i^e - I'^m_i t_m^a + I_e^m(2t_{mi}^{ea} - t_{im}^{ea}) \\ &\quad + t_m^e(2v_{ei}^{ma} - v_{ei}^{am}) - v_{ei}^{mn}(2t_{mn}^{ea} - t_{mn}^{ae}) \\ &\quad + v_{ef}^{ma}(2t_{mi}^{ef} - t_{im}^{ef}) \end{aligned} \tag{7}$$

Received: January 21, 2013
Published: July 12, 2013

$$D_{ij}^{ab} t_{ij}^{ab} = v_{ij}^{ab} + P(ia/jb)\Big[t_{ij}^{ae} I_e^b - t_{im}^{ab} I_j^m + \frac{1}{2} v_{ef}^{ab} c_{ij}^{ef} + \frac{1}{2} c_{mn}^{ab} I_{ij}^{mn} - t_{mj}^{ae} I_{ie}^{mb} - I_{ie}^{ma} t_{mj}^{eb} + (2t_{mi}^{ea} - t_{im}^{ea}) I_{ej}^{mb} + t_i^e I'^{ab}_{ej} - t_m^a I'^{mb}_{ij} \Big] \tag{8}$$

In eqs 7 and 8, the intermediates c, I, and I' are defined as

$$I_a^i = f_a^i + 2v_{ae}^{im} t_m^e - v_{ea}^{im} t_m^e \tag{9}$$

$$I_b^a = (1 - \delta_a^b) f_b^a + (2v_{be}^{am} t_m^e - v_{be}^{ma} t_m^e) - (2v_{eb}^{mn} c_{mn}^{ea} - v_{be}^{mn} c_{mn}^{ea}) - t_m^a f_b^m \tag{10}$$

$$I_j^i = I'^i_j + I_e^i t_j^e \tag{11}$$

$$I'^i_j = (1 - \delta_j^i) f_j^i + (2v_{je}^{im} t_m^e - v_{ej}^{im} t_m^e) + (2v_{ef}^{mi} t_{mj}^{ef} - v_{ef}^{im} t_{mj}^{ef}) \tag{12}$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik/jl) t_k^e v_{el}^{ij} \tag{13}$$

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} c_{jm}^{ea} - v_{jb}^{im} t_m^a + v_{eb}^{ia} t_j^e \tag{14}$$

$$I'^{ab}_{ci} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - t_m^a v_{ci}^{mb} \tag{15}$$

$$I'^{ia}_{jk} = v_{jk}^{ia} + v_{ef}^{ia} t_{jk}^{ef} + t_j^e t_k^f v_{ef}^{ia} \tag{16}$$

$$c_{ab}^{ij} = t_a^i t_b^j + t_{ab}^{ij}$$

In the foregoing, the permutation operator P

$$P(ia/jb) u_{ab}^{ij} = u_{ab}^{ij} + u_{ba}^{ji} \tag{17}$$

has the effect of symmetrizing an arbitrary operand u, such that

$$P(ia/jb) u_{ab}^{ij} = P(ia/jb) u_{ba}^{ji} \tag{18}$$

The integrals over molecular orbitals are obtained from the integrals over the atomic orbital (AO) basis via the 4-index transformation

$$v_{cd}^{ab} = C_p^a C_r^b C_c^q C_d^s \left\langle pq \left| \frac{1}{r} \right| rs \right\rangle \tag{19}$$

The coefficients C in eq 19 are obtained from the iterative Hartree–Fock procedure. The transformed integrals have the following general symmetries,

$$v_{ar}^{bs} = v_{br}^{as}$$

$$v_{ab}^{qs} = v_{ba}^{sq}$$

The (T) correction is given as

$$E^{[T]} = \bar{t}_{abc}^{ijk} t_{ijk}^{abc} D_{ijk}^{abc} \tag{20}$$

$$E^{(T)} = E^{[T]} + t_{ijk}^{abc} D_{ijk}^{abc} \bar{z}_{abc}^{ijk} \tag{21}$$

An arbitrary quantity $\bar{x}_{abc}^{ijk}$ is defined as

$$\bar{x}_{abc}^{ijk} = \frac{4}{3} x_{abc}^{ijk} - 2x_{acb}^{ijk} + \frac{2}{3} x_{bca}^{ijk} \tag{22}$$

and

$$z_{abc}^{ijk} = (t_a^i v_{bc}^{jk} + t_b^j v_{ac}^{ik} + t_c^k v_{ab}^{ij}) / D_{ijk}^{abc} \tag{23}$$

The T_3 amplitudes are

$$D_{ijk}^{abc} t_{ijk}^{abc} = P(ia/jb/kc)[t_{ij}^{ae} v_{ek}^{bc} - t_{im}^{ab} v_{jk}^{mc}] \tag{24}$$

where the symmetrizer $P(ia/jb/kc)$ is

$$P(ia/jb/kc) u_{abc}^{ijk} = u_{abc}^{ijk} + u_{acb}^{ikj} + u_{bac}^{jik} + u_{bca}^{jki} + u_{cba}^{kji} + u_{cab}^{kij}$$

2. COMPUTATIONAL DETAILS

The CCSD equations are nonlinear and must be solved to self-consistency via an iterative procedure, usually with the help of an acceleration method.[14] The CCSD method is dominated by its most expensive term, $v_{ef}^{ab} c_{ij}^{ef}$, which scales as $v^4 o^2$, where v, o are the number of virtual and occupied molecular orbitals, respectively. Formally, the method is expensive in terms of memory and storage as well, with amplitude storage on the order of $v^2 o^2$ and integral storage on the order of v^4,$v^3 o$, ... and so on. The amount of in-core memory depends on the specific algorithm used; most algorithms require $v^2 o^2$ storage per node. This amount of memory is not scalable. For example, a problem with 100 occupied and 1000 virtual orbitals would require 80 GB of memory per node, which is not commonly available.

The noniterative (T) correction requires $v^3 o$ storage and scales as $v^4 o^3$. A naive (T) algorithm is trivial to implement, but an algorithm that has a small memory requirement and scalable input/output (I/O) is more challenging.

There is a CCSD(T) method in nearly every quantum chemistry package. The ACES[15] and NWChem[16] implementations can handle very large computations, provided that a supercomputer is available.[17] The MOLPRO[18] algorithm has an $o^2 v^2$ memory requirement, which limits its utility, but it is perhaps the fastest algorithm for smaller calculations. The GAMESS[19] implementation runs in parallel but is similarly limited by an $o^2 v^2$ memory requirement. The Janowski–Ford–Pulay disk array CC implementation[20–22] can handle large computations of the order of a thousand basis functions on a commodity cluster by utilizing a filesystem for storage, but the performance of their algorithm may be limited by disk I/O.

3. DESIGN OF A SCALABLE AND EFFICIENT ALGORITHM

In our previous work, we developed an MP2 energy algorithm,[23] which has a small memory footprint, good performance, a flexible storage implementation, and is able to run on workstations and clusters equally well. In the same spirit, a coupled cluster algorithm can be designed, such that it is efficient, has a small memory footprint, is able to flexibly utilize a filesystem and memory for storage, and as a result can run on machines with very different capabilities.

For coupled cluster algorithms (and other many-body methods), it is the memory that is most likely to limit the application of the algorithm. Memory is a limited resource, unlike the time. For example, the time to completion for calculations can be decreased by providing more computational hardware, whereas the amount of physical memory per node cannot be increased by adding another node.

Some very large arrays can (and need to) be distributed across the nodes (distributed memory) or stored on the filesystem. Disks are inexpensive and offer terabytes of storage, but filesystem I/O can be very slow if not done right.

Nevertheless, a considerable amount of memory must be present to carry out local calculations.

What are the memory limitations of current hardware? A "*typical*" workstation or a cluster node in most research groups has between 1 and 8 GB of memory per core, with 2 GB of RAM probably being the most common. For an entire *node*, the amount of memory can be as much as 64 GB or more, depending on the number of cores/node. That number will increase in the future but possibly at a slower rate than the increase in the number of cores.

To draw a connection between memory and the dimensions present in CC calculations, several generic arrays of varying dimensions, corresponding to 100 occupied orbitals and 1000 and 2000 basis functions, are listed in Table 1. The dimensions of these arrays may correspond, for example, to an entire integral array or to the first three indices. The algorithm design is then guided by what arrays are small enough to be stored per node or per core. It should be kept in mind that the sizes listed are not for the entire calculation, but for one of the several arrays needed. Some of the arrays can be shared, but some must be allocated per thread/core.

Table 1. Array Sizes for o = 100

array	size (GB), n_{basis} = 1000	size (GB), n_{basis} = 2000
o^4	0.8	0.8
o^2n	0.08	0.16
on^2	0.8	3.2
o^3n	8.0	16.0
n^3	8.0	64.0
o^2n^2	80.0	320.0
on^3	800.0	6400.0
n^4	8000.0	128000.0

Storing an o^2n^2 array per node (let alone per core) is too expensive: A node with 80 GB of RAM is rare and one with 320 GB is even more rare. The same is true for the quartic arrays other than o^4 and arrays involving an n^2 factor. Storing, for example, several 3 GB arrays would preclude most systems from being able to handle more than a thousand basis functions.

While this was recognized years ago, c.f. work by Rendell et al.,[24,25] many programs continue to operate on much more lax assumption about memory. For example the GAMESS CCSD(T) algorithm assumes that there is enough memory to store n^3 quantities per node and MOLPRO assumes o^2n^2 storage per node.

The choice is then to restrict memory requirements to o^2n (or smaller) arrays, whose size only increases linearly with the basis set. Trying to limit memory further than o^2n, to say on, will come at a very high cost of increased I/O.

Some arrays, notably n^4, are too great to store even in secondary storage. The terms involving such an array must be evaluated directly, i.e. on the fly, at the modest cost of recomputing atomic integrals, cf. Olson et al.[26−28] However, to push the ability of the algorithm beyond a thousand basis functions, on^3 storage also must be eliminated in the CCSD algorithm. To ensure that I/O overhead is low even on filesystems, transfers to and from secondary storage must be contiguous and in large chunks. There are three basic remote memory *transfer* operations: `put`, `get`, `accumulate`. The last of these cannot be implemented efficiently via the filesystem I/O, and the algorithm must not rely on it.

Finally, to achieve computational efficiency, all of the expensive tensor contractions that must be carried out using `dgemm` and tensor permutations must not exceed two adjacent indices to ensure data locality; e.g., $A(j, i, k) = A(i, j, k)$ is okay, but $A(k, j, i) = A(i, j, k)$ is not, because the latter has poor memory performance. The work distribution between the nodes must be over the virtual index rather than the (usually) much smaller occupied index, to ensure that the algorithm can scale to hundreds of nodes. The work within the node is easily parallelized using threads. This multilevel parallelization guarantees that the algorithm will scale to thousands of cores.

In the following discussion, the primary focus is on memory, then on secondary storage and I/O, and only then on the computational aspect. The consequent performance is illustrated below with benchmarks.

4. IMPLEMENTATION

This section is broken into three subsections that address the direct CCSD terms, the nondirect CCSD terms, and the triples correction, respectively. The CCSD component of the CCSD(T) algorithm is by far the most complex due to the number of terms.

Before proceeding to the respective sections, consider I/O optimization via loop blocking. In Algorithm 1, B is a blocking factor. If B = 1, then it is just a regular loop: the innermost (most expensive) load operation is executed N^3 times, the total I/O overhead is K^2N^3, and the local buffer size is K^2. If B is greater than 1, the innermost load operation is called $(N/B)^3$ times, the I/O overhead is $K^2B(N/B)^3 = K^2N^3/B^2$, and the local buffer size is K^2B. So, at the cost of increasing the local buffer size, the I/O overhead can be reduced by a factor of B^2. In general, loop blocking decreases I/O by $B^{(L-1)}$ where L is the number of nested loops.

The loop blocking will be used where I/O might pose a problem. Since blocking also requires an increase in memory overhead, the blocking factor can be determined by setting a runtime memory limit.

```
for i = 0:N,B { // iterate to N in steps of B
  for j = 0:N,B {
    for k = 0:N,B {
      // the innermost load operation
      buffer(K,K,B) = load A(K,K,k:k+B)
      ...
    }
  }
}
```

Algorithm 1: loop blocking.

4.1. Direct Terms.

The evaluation of the most expensive, four-virtual, ν_{cd}^{ab} term is almost always carried out in AO basis.[28] As mentioned already, ν_{cd}^{ab} has to be evaluated directly due to storage constraints. The same approach can be extended to evaluate terms ν_{bc}^{ia} directly as well at little additional cost.

To make notation simpler, the conventional VT notation is used, where the general single and double amplitudes contractions are referred to as VT_1, VT_2, $VT_1{}^2$, the latter implying contraction with two single amplitudes.

The integral ν_{cd}^{ab} is contracted with $(T_1{}^2)_{ij}^{cd} = (T_1T_1)_{ij}^{cd} = t_i^c t_j^d$ and $(T_2)_{ij}^{cd} = t_{ij}^{cd}$ amplitudes,

$$V(T_2)_{ij}^{ab} = t_{ij}^{cd} C_d^s C_c^q V_{qs}^{pr} C_r^b C_p^a \quad (25)$$

$$V(T_1^2)_{ij}^{ab} = t_i^c t_j^d C_d^s C_c^q V_{qs}^{pr} C_r^b C_p^a \tag{26}$$

Half-transforming the amplitudes to the AO basis and factoring out half-contracted terms yields expressions in terms of half-transformed intermediates U, with subscripts referring to the T contraction (Recall that p, q, r, and s are AO indices.).

$$(U_2)_{ij}^{pr} = (t_{ij}^{cd} C_d^s C_c^q) V_{qs}^{pr} \tag{27}$$

$$(U_1^2)_{ij}^{pr} = (t_i^c C_c^q)(t_j^d C_d^s) V_{qs}^{pr} \tag{28}$$

$$V(T_2)_{ij}^{ab} = (U_2)_{ij}^{pr} C_r^b C_p^a \tag{29}$$

$$V(T_1^2)_{ij}^{ab} = (U_1^2)_{ij}^{pr} C_r^b C_p^a \tag{30}$$

All similar VT terms can be obtained from U at virtually no cost by having the last two AO indices transformed to occupied and virtual indices. For example, the ν_{bc}^{ia} terms in eq 7 are just

$$v_{ef}^{ma}(2t_{mi}^{ef} - t_{im}^{ef}) = 2U_{mi}^{qs} C_q^m C_s^a - U_{im}^{qs} C_q^m C_s^a \tag{31}$$

The ν_{bc}^{ia} also enter the VT_1 diagrams

$$V(T)_{1\,ab}^{ij} = t_i^c C_j^s C_c^q V_{qs}^{pr} C_r^b C_p^a \tag{32}$$

$$V(T)_{1\,jb}^{ia} = t_i^c C_d^s C_j^q V_{qs}^{pr} C_r^b C_p^a \tag{33}$$

and two more intermediates are needed

$$(U_1)_{qs}^{ij} = (t_a^i C_p^a) C_r^j V_{qs}^{pr} \tag{34}$$

$$(U_1)_{js}^{ir} = (t_a^i C_p^a) C_j^q V_{qs}^{pr} \tag{35}$$

which can then be transformed into appropriate VT_1 diagrams.

Now, if all four U intermediates are available, neither ν_{cd}^{ab} nor ν_{bc}^{ia} need to be stored for the CCSD iterations; they can be replaced with much smaller $4o^2n^2$ storage.

The overall computational cost is dominated by the n^4o^2 step of computing half-transformed intermediates. That cost can be halved by noticing that the intermediates are symmetric under simultaneous exchange of transformed and atomic orbitals, so that only unique atomic index pairs need to be evaluated. While the above only utilizes one of the several permutation symmetries, the memory is guaranteed to not exceed the N^2M^2 quantity and I/O is guaranteed to be contiguous.

Half-transformed T_2 amplitudes, eq 27, also provide a way to devise a direct contraction algorithm with very little memory requirement. Since the contraction is in the AO basis, atomic indices can be contracted without having to construct V_{qs}^{ab} which would require all atomic basis p, r indices and thus N^2M^2 memory, where M is the size of the largest shell. Algorithm 2 only needs NM^3 memory.

The important points of Algorithm 2 are the following:

- The integral symmetry is exploited to halve the number of integral calculations *and* transformations.
- The loop over Q, S can be distributed over nodes.
- The loop over R can be parallelized over threads. In this case, the U storage can be shared, provided the updates to shared memory are synchronized.
- The innermost $t2$ loads can be reduced by blocking the Q, S loops (cf. the discussion on loop blocking).
- Per thread storage is NM^3, which is 16 MB for a basis set of size 2000 with f shells (M = 10). The local U storage is likewise small, only 8 MB for o = 100. This tiny memory footprint allows for a very large Q, S blocking factor, and consequently, the I/O can be dramatically reduced.

```
for S in Shells {
  for Q ≤ S {
    for R in Shells {
      for P in Shells {
        // skip insignificant ints
        if (!screen(P,Q,R,S)) continue;
        // evaluate 2-e integrals(PQ|RS)
        V(P,R,Q,S) = eri(P,Q,R,S);
      }
      // i and j are unrestricted
      // loops over all P functions are implied
      // loops over shells Q,S are implied
      for r in R {
        U1(i,j,q,s) = ...
        U12(i,j,q,s) = ...
        load t(o,o,n,r)
        U2(i,j,q,s) += t(i,j,p,r)*V(p,r,q,s)
      }
    }
    store U1(i,j,Q,S), U1(j,i,S,Q)
    store U12(i,j,Q,S), U12(j,i,S,Q)
    store U2(i,j,Q,S), U2(j,i,S,Q)
  }
}
```

Algorithm 2: direct CCSD intermediates.

Note that both UT_1 terms cannot be evaluated simultaneously using the above algorithm, as they correspond to two different integrals, $\langle pq|rs\rangle$ and $\langle pr|qs\rangle$. However, one of them can easily be evaluated by applying the algorithm a second time to compute just a *single* UT_1 term at a very modest $n^4/2$ computational cost of re-evaluating $2-e$ integrals, compared with an overall cost of on^4 to compute VT_1 diagrams.

In many respects the algorithm implementation is similar to the algorithm described by Janowski and Pulay,[21] although the authors only recently became aware of this symmetric/antisymmetric approach that can reduce the cost by an additional factor of 2. Another important difference is that the current implementation is fully threaded, allowing for an overall smaller memory footprint.

4.2. Nondirect CCSD Terms. Because the singles amplitudes storage is negligible, on, the singles part of the CCSD code is easy to implement and parallelize. By making a virtual index the outermost index, the local memory is guaranteed not to exceed o^2n since all of the diagrams with three and four virtual indices have already been evaluated above.

The doubles amplitudes calculation requires the most effort to implement, primarily due to the number of contractions and the terms that require significant I/O. Recall that all ν_{cd}^{ab} and ν_{bc}^{ia} terms have been evaluated, as have many similar VT terms.

The first step toward deriving a scalable algorithm for Dt_{ab}^{ij} (See eq 8) is to fix the outermost loop at the outermost virtual index b, since the b index can be evaluated across nodes independently. For each b iteration an o^2n Dt_{ab}^{ij} block is evaluated and stored.

The quantities with a b index are loaded once, guaranteed not to exceed size o^2n. The tensors without a b index imply that the tensor is needed in its entirety for each b iteration. To ensure that no ν or t memory exceeds o^2n, those tensors without a b index must be loaded into memory o^2n tiles at a time for each b index inside a loop over a dummy virtual orbital index, u. This increases the I/O cost to o^2n^2 per b index, or o^2n^3 overall, which is still below the o^3n^3 computational cost.

There are three tensors that must be contracted fully for any given b index (i.e., they have no external label b): v_{ia}^{jb}, v_{ij}^{ab}, v_{ij}^{ab}, cf. equation 8 and the relevant intermediates. The loop corresponding to v_{ia}^{jb} can be eliminated right away, it is only needed in its entirety to evaluate $I_{ie}^{ma}t_{mj}^{eb}$ in eq 8. Since, this term appears inside the symmetrizer P,

$$P(v_{je}^{mb}t_{mi}^{ea}) = P(v_{ie}^{ma}t_{mj}^{eb})$$

I_{ie}^{ma} can be replaced by an equivalent I_{je}^{mb}. This leads to Algorithm 3.

```
for b in v {  //  loop  over  virtual  b
index
  Dt(i,j,a) = 0

  load t(o,o,v,b)
  load V(o,o,v,b)
  load V(o,v,o,b)
  load V(o,o,o,b)

  Dt += Vt

  // terms with t
  for u in v {
    load t'(o,o,v,u)
    // evaluate terms with t'
    Dt += Vt'
  }

  // terms with v
  for u in v {
    load v'(o,o,v,u)
    // evaluate terms with v'
    Dt += V't
  }

  store Dt(o,o,v,b)
}
```

Algorithm 3: CCSD.

The important points about Algorithm 3:

- The loop over the b index is easy to make parallel.
- The local memory is on the order $4o^2n$ plus o^2n per innermost v'/t' temporary storage, corresponding to loading all of the v_{ij}^{ab}, v_{ij}^{ab} quantities, one virtual index at a time.
- The b loop can be easily blocked to reduce the I/O by a blocking factor B at the expense of increasing the memory by a factor of B.
- Since the memory footprint is low, B can be fairly large. For example, for $O = 100$, $V = 2000$, $B = 4$ and $B = 8$, the required memory is 2.6 and 5.2 GB per *node*, respectively.
- The operations outside the u loop can be parallelized inside the node by using a threaded math library.
- The operations inside the u loop can be explicitly parallelized inside the node via threads, with the added benefit of overlapping I/O and computations.

4.3. (T).

The (T) correction, eq 14, only involves t_{ab}^{ij}, v_{ka}^{ij}, v_{ab}^{ij}, and v_{bc}^{ia}. The unused CCSD arrays previously allocated can be freed to make space for v_{bc}^{ia}. Since v_{bc}^{ia} was never constructed, another integral transformation needs to be carried out at a small on^4 cost.

The Piecuch (T) correction[13] equations were given in a way that requires keeping an occupied index fixed and permuting the virtual indexes. In other words the local memory required for t_{abc}^{ijk} would have been v^3. Since the triples amplitudes are symmetric with respect to the exchange of index "columns",

$$t_{abc}^{ijk} = t_{bac}^{jik} = t_{acb}^{ikj} = ...$$

all terms with t_{bac}^{ijk} can be written with a virtual index fixed, e.g., $t_{bac}^{ijk} = t_{abc}^{jik}$, $t_{cab}^{ijk} = t_{abc}^{jki}$, etc.

This leads to an algorithm that is similar to that of Rendell et al.[25] or Jankowski and Pulay,[21] which require o^2n storage. With the correction terms permuted as above, the memory scales as $B^3o^3 + B^2o^2n$, where B is the blocking factor. This implies that even for large computations, the memory overhead will be modest. Since the implementation is multithreaded, only one set of memory buffers need to be allocated to drive multiple cores.

Now the T_3 amplitudes can be implemented as a series of 12 `dgemms` and 6 index permutations, as illustrated in Algorithm 4. The important points about Algorithm 4 are:

- The symmetry in a, b, c indices is utilized.
- The loop over a, b, c indices is easily parallelizable.
- Only the loads with an a index are innermost.
- The loops can be easily blocked to reduce the I/O by a factor of B^2 where B is the blocking factor.
- The local storage required is $3o^2vB + 3o^3B + 6ovB^2 + o^3B^3$.
- If $B > 1$, the actual `dgemms` are carried out inside another B^3 loop, which can be parallelized *within* a node by using threads.
- Since the memory footprint is low, the blocking factor can be large. For example, for $O = 100$, $V = 1000$, $B = 4$, and $B = 8$, the required memory is 1.6 and 6.4 G per *node* respectively.

4.4. Overall Picture.

The algorithm is implemented entirely in object oriented C++, as a part of stand-alone library (LIBCCHEM) which includes previously reported ERI (electron repulsion integrals),[29] Fock,[30] and MP2 methods.

```
for c in V {
  for b in c {
    for a in b {

      load t(o,o,a,b)
      load t(o,o,a,c)
      load t(o,o,b,c)

      load V(o,o,o,a)
      load V(o,o,o,b)
      load V(o,o,o,c)

      load V(o,o,v,a)
      load V(o,o,v,b)
      load V(o,o,v,c)

      load V(o,v,b,c)
      load V(o,v,c,b)
      load V(o,v,a,c)
      load V(o,v,c,a)
      load V(o,v,a,b)
      load V(o,v,b,a)

      // t(i,j,e,a)*V(e,k,b,c) corresponds to
      // dgemm(t(ij,e), V(e,k)), etc
      t(i,j,k) = t(i,j,e,a)*V(e,k,b,c) - t(i,m,a,b)*V(j,k,m,c)
      t(i,k,j) = t(i,k,e,a)*V(e,j,c,b) - t(i,m,a,c)*V(k,j,m,b)
      t(k,i,j) = t(k,i,e,c)*V(e,j,a,b) - t(k,m,c,a)*V(i,j,m,b)
      t(k,j,i) = t(k,j,e,c)*V(e,i,b,a) - t(k,m,c,b)*V(j,i,m,a)
      t(j,k,i) = t(j,k,e,b)*V(e,i,a,c) - t(j,m,b,c)*V(k,i,m,a)
      t(j,i,k) = t(j,i,e,b)*V(e,k,c,a) - t(j,m,b,a)*V(i,k,m,c)
      ...
    }
  }
}
```

Algorithm 4: (T).

The code uses C++ templates and various Boost components to concisely express data load, store, and BLAS operations through tensor objects, as well as to handle the distribution of work among nodes and between threads. The outcome is a code that is easier to understand and *modify* than comparable Fortran routines, provided the programmer is familiar with advanced features of C++. The library requires only minimal input from the host program and can be connected to a variety of packages.

The storage is implemented using global arrays (GA)[31] for distributed memory and HDF5[32] for file storage. GAMESS's own distributed data interface[33] (DDI) currently has no support for arrays of more than two dimensions. But with a small addition, the DDI calls can be translated directly into GA equivalents, so that GAMESS can run via GA without modifications while at the same time providing 3D and 4D array functionality via direct calls to GA. The arrays are first allocated in faster GA memory until the limit is reached, and then on the filesystem. The arrays that would require the most I/O need to be allocated first to ensure that they reside in distributed memory.

The overall algorithm may be outlined as follows:

- The CCSD arrays are allocated, with t and ν_{ij}^{ab} first to ensure that these arrays are in fast storage. Overall, storage is needed for t, ν_{ij}^{ab}, ν_{ij}^{ka}, ν_{ia}^{jb}, ν_{ij}^{kl}, Dt, and four U intermediates.
- The allocated arrays are evaluated using the regular 4-index transformation.
- The initial $T2$ amplitudes are taken to be the MP2 amplitudes, $\nu_{ij}^{ab}/D_{ij}^{ab}$, and the $T1$ amplitudes are set to zero.
- The intermediate U storage is allocated.
- The CCSD equations are repeated until an acceptable threshold is reached, either the energy difference or the amplitude difference.
- The CCSD step is optionally accelerated using DIIS.[16]
- Once converged, all but the first three arrays are freed and ν_{ij}^{bc} array is allocated and evaluated.
- The noniterative (T) method is performed.

5. PERFORMANCE

To assess the performance and applicability of the algorithm, three scenarios are considered here: single node performance, performance on a cluster of modest size, and high-end cluster performance. The inputs are selected to reflect a range of basis functions and occupied orbitals.

The modest cluster, Exalted, is composed of nodes connected by InfiniBand. Each node has one Intel X5550 2.66 GHz 6-core processor, 24 GB of RAM, two local disk drives, and an NVIDIA Fermi C2050 GPU card. The filesystem is composed of two drives, connected in RAID0 configuration, for a total throughput of just over 100 MB/s.

GAMESS treats spherical basis sets in Cartesian coordinates internally, as do many other codes. That is, the transformation from Cartesian integrals is done via the coefficient matrix rather than directly. Therefore, the calculations presented here also had to account for this: for example d and f shells have 6 and 10 functions rather than 5 and 7. This leads to higher computation and storage cost over purely spherical implementations.

First, consider the performance of the CCSD(T) algorithm when all arrays are allocated on the filesystem (Table 2), rather than in the distributed memory, with a direct I/O driver, bypassing operating system I/O caching. The algorithm can handle a fairly large CCSD(T) $C_8H_{10}N_4O_2$/cc-PVTZ (640 virtual and 37 occupied orbitals) calculation with a very low memory footprint, as low as 128 MB shared among *all* six threads. Second, the CCSD iterations are not particularly sensitive to I/O; increasing the available memory (and hence the blocking factor) only modestly changes the running time. However, the (T) calculations are sensitive to memory: limiting memory to 128 MB increases the runtime by 16%. Nevertheless, owing to the fact that the algorithm was designed with memory in mind, even a small amount of available memory gives a large enough blocking factor to hide the I/O penalty.

Table 2. Exalted CCSD I/O Performance, $C_8H_{10}N_4O_2$/cc-PVTZ, 6 Cores

total available memory, MB	CCSD[a]	(T)	(T) I/O
128	189	4640	688
256	185	4055	288
512	180	3911	224
1024	179	3905	182

[a]All times are in minutes.

Next, consider the ability of the algorithm to run on a single node with reasonable amount of memory (1 GB per core, 6 GB total) and to use a filesystem in case not enough memory is available to store all data, Table 3. As can be seen, even on a single node fairly large CCSD(T) jobs can still run in a reasonable time frame (which in the opinion of the authors is under a week). Despite falling back to disk in all (T) calculations, across the board the I/O time as a fraction of total time is very small, below 5%.

Table 3. Exalted Single Node Performance, 6 Cores

input	no. AO/Occ[a]	CCSD[b]	(T)	(T) Mem/Disk[c]	(T) I/O
$C_4N_3H_5$/aug-cc-PVTZ	565/18	42 min	8 h	2.1/19.5 GB	13 min
$C_8H_{10}N_4O_2$/aug-cc-PVDZ	440/37	50 min	17 h	5.5/17.0 GB	13 min
$SiH_4B_2H_6$/aug-cc-PVQZ	875/11	141 min	18 h	3.4/53.4 GB	49 min
$C_8H_{10}N_4O_2$/cc-PVTZ	640/37	180 min	64 h	12.2/49.0 GB	42 min

[a]Number of atomic/valence occupied orbitals. [b]Single CCSD iteration time. [c]Memory/disk used to evaluate (T).

The cluster performance is assessed on the basis of the time larger jobs take to run, Table 4, and the scalability of a medium-size job, Table 5. First, all of the inputs used for single node benchmarking can run in under a day on the cluster. Second, a

Table 4. Exalted Cluster Performance[a]

input	no. AO/Occ[b]	no. cores	CCSD[c]	(T)
$C_4N_3H_5$/aug-ccPVTZ	565/18	24	12	61
$SiH_4B_2H_6$/aug-ccPVQZ	875/11	48	20	133
$C_8H_{10}N_4O_2$/ccPVTZ	640/37	48	26	482
$C_{26}H_{29}BO$/aug-ccPVQZ	961/72	96	211	N/A[d]

[a]All times are in minutes. [b]Number of atomic/occupied valence orbitals. [c]Single CCSD iteration time. [d]Job requires 0.5 TB of storage: Exalted does not have sufficient memory or parallel FS.

Table 5. Exalted Cluster Scaling, $C_8H_{10}N_4O_2$/ccPVTZ[a]

cores/nodes	CCSD[b]/speed-up	(T)/speed-up
24/4	28/1.00	971/1.00
48/8	15/1.87	482/2.01
96/16	11/2.55	240/4.04

[a]All times are in minutes. [b]Single iteration time.

large CCSD modified Tamoxifen (T-B) calculation, $C_{26}H_{29}BO$, can run on this relatively small (Exalted) cluster, 3 h per iteration.

As expected, the (T) algorithm scales well, as shown in Table 4, since it is very easy to parallelize to a large number of nodes. However, the scalability of the CCSD algorithm is not perfect. This is especially noticeable when running on a large cluster, such Cray XE6 system, which has thousands of cores across the 32-core AMD Bulldozer 64 GB nodes connected by a fast network. The performance gain from increasing the number of nodes, Table 6, is below the linear scaling but the longer B-T calculation scales reasonably well to 1024 cores, reducing the runtime by a factor of 3.3 relative to 256 core run. The Cray system on which the CCSD benchmarks were executed was a test system, with runtime limited to 3 h. Therefore, it was not possible to execute longer CCSD(T) calculations. However, considering the algorithm design and scalability shown in Table 5, the (T) calculations can be expected to scale nearly linearly.

Table 6. Cray XE6 CCSD Performance[a]

no. cores	256 cores	512 cores/ speed-up	1024 cores/ speed-up
$SiH_4B_2H_6$ (T)/aug-ccPVQZ	130	76/1.71	42/3.10
$C_8H_{10}N_4O_2$ CCSD/ ccPVTZ	15	9/1.67	6/2.5
$C_{26}H_{29}BO$ CCSD/aug-ccPVQZ	253	134/1.89	76/3.33

[a]All times are in minutes per single iteration.

Each XE6 node has two chips, 16 cores each. The benchmarks in Table 6 were obtained running 32 threads over the entire node created from a single MPI process. The better option, especially in the case of (T) is to run one MPI process per *chip* rather than per *node*; see Table 7. If each MPI process runs (and creates threads) within a single chip only, the threads do not need to communicate over the slower bridge connecting two chips. Generally, there is a large penalty for sharing data across the *chips*, which must be avoided by having a flexible approach to launch jobs.

Table 7. Cray XE6 Intranode Configuration, $SiH_4B_2H_6$ (T)/ aug-ccPVQZ[a]

no. cores	32 × 1 threads/MPI	16 × 2 threads/MPI
256	130	101
512	76	49
1024	42	27

[a]All times are in minutes.

5.1. GPU. As expected, the direct terms account for the most time in CCSD iterations. In the present implementation, most of that work is concentrated in a continuous application of just one `dgemm` operation. Adding a graphical processor (GPU) `dgemm` to handle matrix multiplication, while keeping the integral evaluation on the host, is fairly easy. In a multithreaded environment, several threads must be assigned to a GPU device to avoid work imbalance.

Augmented with GPU BLAS, via CUBLAS,[34] the CCSD calculations on a single Exalted node get a noticeable speed up, shown in Table 8, if the direct term (See section 4.1) dominates the entire iteration (this is the case if the number of occupied orbitals is very small relative to the size of the basis set). If the number of occupied orbitals is relatively high, the direct term accounts for a smaller fraction of the total iteration time, and consequently, the GPU benefit is less noticeable overall.

Table 8. Exalted Single Node + GPU CCSD Performance[a]

input	$C_8H_{10}N_4O_2$/ ccPVTZ	$SiH_4B_2H_6$/aug-ccPVQZ	$C_4N_3H_5$/aug-ccPVTZ
direct	124	131	36
direct + GPU[b]	53	65	26
CCSD	163	142	42
CCSD + GPU[b]	95	75	33
CCSD speed-up[c]	1.4×	1.9×	1.3×

[a]All times are in minutes per iteration. [b]GPU enabled. [c]Overall CCSD speed-up.

The major challenge with using GPUs is to supply data fast enough from the host to keep the device busy. This was accomplished by using multiple streams to execute kernels and fetch data from the secondary storage while at the same time executing CPU code on multiple cores. Each thread on a node can execute a particular set of transformations on a GPU; however, that may oversubscribe the GPU. Therefore, each thread queries whenever there is a backlog of previously scheduled streams and uses CPU matrix routines if the device is busy.

Provided that there is enough data to hide transfer latency, the GPU can almost double the performance of a single Exalted node, roughly 6× speed-up over a single core. For reference, the maximum speed-up achieved by measuring just the performance of a CUBLAS `dgemm` over a *single-core* Intel `dgemm` was on the order of 12× for the given GPU device for a general square matrix. The difference in performance is attributable to how efficiently the GPU can be supplied with data.

6. CONCLUSIONS

The algorithm presented in this paper is able to handle fairly large jobs on a single node, a small cluster, and high-end Cray system. The algorithm has a small adjustable memory footprint and is able to optionally use the filesystem if the data exceeds distributed memory storage. The algorithm can also optionally use GPUs to speed up certain CCSD computations. When running on the multicore node with multiple processor packages (chips), the algorithm benefits from limiting thread communication to within a chip.

The algorithm is implemented entirely in object oriented C++, as a part of a stand-alone library which includes previously reported ERI, Fock, and MP2 methods.

The geometries used in these benchmarks calculations as well as further details or source code are available from the authors upon request.

AUTHOR INFORMATION

Corresponding Author

*E-mail: mark@si.msg.chem.iastate.edu.

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the Nvidia Corporation and in part by a National Science Foundation Software Infrastructure for Sustained Innovation grant. Many help discussions with Drs. Ryan Olson, Sarom Leang, and Professor Theresa Windus are gratefully acknowledged.

REFERENCES

(1) Pople, J. A.; Seeger, R.; Krishnan, R. Variational configuration interaction methods and comparison with perturbation theory. *Int. J. Quantum Chem.* **1977**, *12*, 149−163.
(2) Pople, J. A.; Head-Gordon, M.; Raghavachari, K. Quadratic configuration interaction. a general technique for determining electron correlation energies. *J. Chem. Phys.* **1987**, *87*, 5968−5975.
(3) Pople, J. A.; Gill, P. M. W.; Johnson, B. G. Kohn-Sham density-functional theory within a finite basis set. *Chem. Phys. Lett.* **1992**, *199*, 557−560.
(4) Coester, F.; Kümmel, H. Short-range correlations in nuclear wave functions. *Nucl. Phys.* **1960**, *17*, 477−485.
(5) Cizek, J. On the correlation problem in atomic and molecular systems. Calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *J. Chem. Phys.* **1966**, *45*, 4256−4266.
(6) Paldus, J.; Cizek, J.; Shavitt, I. Correlation problems in atomic and molecular systems. IV. Extended coupled-pair many-electron theory and its application to the BH3 molecule. *Phys. Rev. A.* **1972**, *5*, 50.
(7) Mukherjee, D.; Moitra, R. K.; Mukhopadhyay, A. Applications of a nonperturbative many-body formalism to general open-shell atomic and molecular problems: calculation of the ground and the lowest pi-pi* singlet and triplet energies and the first ionization potential of trans-butadiene. *Mol. Phys.* **1977**, *33*, 955−969.
(8) Lindgren, I. A coupled-cluster approach to the many-body perturbation theory for open-shell systems. *Int. J. Quantum Chem.* **1978**, *14*, 33−58.
(9) Scuseria, G. E.; Janssen, C. L.; Schaefer, H. F. An efficient reformulation of the closed-shell coupled cluster single and double excitation (CCSD) equations. *J. Chem. Phys.* **1988**, *89*, 7382−7387.
(10) Purvis, G. D., III; Bartlett, R. J. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *J. Chem. Phys.* **1982**, *76*, 1910.
(11) Raghavachari, K.; Trucks, G. W.; Pople, J. A.; Head-Gordon, M. A fifth-order perturbation comparison of electron correlation theories. *Chem. Phys. Lett.* **1989**, *157*, 479−483.
(12) Piecuch, P.; Kucharski, S. A.; Bartlett, R. J. Coupled-cluster methods with internal and semi-internal triply and quadruply excited clusters: CCSDT and CCSDTQ approaches. *J. Chem. Phys.* **1999**, *110*, 6103.
(13) Piecuch, P.; Kucharski, S. A.; Kowalski, K.; Musiał, M. Efficient computer implementation of the renormalized coupled-cluster methods: The R-CCSD [T], R-CCSD (T), CR-CCSD [T], and CR-CCSD (T) approaches. *Comput. Phys. Commun.* **2002**, *149*, 71−96.
(14) Scuseria, G. E.; Lee, T. J.; Schaefer, H. F. Accelerating the convergence of the coupled-cluster approach: The use of the DIIS method. *Chem. Phys. Lett.* **1986**, *130*, 236−239.
(15) Lotrich, V.; Flocke, N.; Ponton, M.; Yau, A. D.; Perera, A.; Deumens, E.; Bartlett, R. J. Parallel implementation of electronic structure energy, gradient, and hessian calculations. *J. Chem. Phys.* **2008**, *128*, 194104.
(16) Bernholdt, D. E.; Apra, E.; Früchtl, H. A.; Guest, M. F.; Harrison, R. J.; Kendall, R. A.; Kutteh, R. A.; Long, X.; Nicholas, J. B.; Nichols, J. A.; et al. Parallel computational chemistry made easier: The development of NWChem. *Int. J. Quantum Chem.* **1995**, *56* (S29), 475−483.
(17) PCC benchmarks. http://www.qtp.ufl.edu/PCCworkshop/PCCbenchmarks.html.
(18) Werner, H. J.; Knowles, P. J.; Lindh, R.; Manby, F. R.; Schütz, M.; Celani, P.; Korona, T.; Rauhut, G.; Amos, R. D.; Bernhardsson, A. et al. *Molpro*, version 2006.1, a package of Ab Initio programs. 2006.
(19) Gordon, M. S. and Schmidt, M. W. *Advances in electronic structure theory: GAMESS a decade later*; Elsevier: Amsterdam, 2005; pp 1167−1189.
(20) Janowski, T.; Ford, A. R.; Pulay, P. Parallel calculation of coupled cluster singles and doubles wave functions using array files. *J. Chem. Theory Comput.* **2007**, *3*, 1368−1377.
(21) Janowski, T.; Pulay, P. Efficient Parallel Implementation of the CCSD External Exchange Operator and the Perturbative Triples (T) Energy Calculation. *J. Chem. Theory Comput.* **2008**, *4*, 1585−1592.
(22) Baker, J.; Janowski, T.; Wolinski, K.; Pulay, P. Recent developments in the PQS program. *Comput. Mol. Sci.* **2012**, *2*, 63−72.
(23) Asadchev, A.; Gordon, M. S., unpublished work.
(24) Rendell, A. P.; Lee, T. J.; Lindh, R. Quantum chemistry on parallel computer architectures: coupled-cluster theory applied to the bending potential of fulminic acid. *Chem. Phys. Lett.* **1992**, *194*, 84−94.
(25) Rendell, A. P.; Lee, T. J.; Komornicki, A.; Wilson, S. Evaluation of the contribution from triply excited intermediates to the fourth-order perturbation theory energy on Intel distributed memory supercomputers. *Theor. Chim. Acta.* **1993**, *84*, 271−287.
(26) Kobayashi, R.; Rendell, A. P. A direct coupled cluster algorithm for massively parallel computers. *Chem. Phys. Lett.* **1997**, *265*, 1−11.
(27) Olson, R. M.; Bentz, J. L.; Kendall, R. A.; Schmidt, M. W.; Gordon, M. S. A novel approach to parallel coupled cluster calculations: Combining distributed and shared memory techniques for modern cluster based systems. *J. Chem. Theory Comput.* **2007**, *3*, 1312−1328.
(28) Meyer, W. Theory of self-consistent electron pairs. An iterative method for correlated many-electron wavefunctions. *J. Chem. Phys.* **1976**, *64*, 2901−2908.
(29) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. Uncontracted Rys Quadrature implementation of up to g functions on graphical processing units. *J. Chem. Theory Comput.* **2010**, *6*, 696−704.
(30) Asadchev, A.; Gordon, M. S. New multithreaded hybrid CPU/GPU approach to Hartree-Fock. *J. Chem. Theory Comput.* **2012**, *8*, 4166−4176.
(31) Nieplocha, J.; Palmer, B.; Tipparaju, V.; Krishnan, M.; Trease, H.; Aprà, E. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.* **2006**, *20*, 203−231.
(32) The HDF Group. Hierarchical data format, version 5. http://www.hdfgroup.org/HDF5.
(33) Olson, R. M.; Schmidt, M. W.; Gordon, M. S.; Rendell, A. P. Enabling the efficient use of SMP clusters: the GAMESS/DDI model. *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Phoenix, AZ, Nov 15−21, 2003; p 41.
(34) Nath, R.; Tomov, S.; Dongarra, J. Accelerating GPU kernels for dense linear algebra. *High Performance Computing for Computational Science−VECPAR 2010*, Berkeley, CA, June 22−25, 2010; pp 83−92.