JCTC Journal of Chemical Theory and Computation
ARTICLE
pubs.acs.org/JCTC

Coupled Cluster Theory on Graphics Processing Units I. The Coupled Cluster Doubles Method

A. Eugene DePrince, III[*,†] and Jeff R. Hammond[‡]

[†]Center for Nanoscale Materials and [‡]Leadership Computing Facility, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, United States

ABSTRACT: The coupled cluster (CC) ansatz is generally recognized as providing one of the best wave function-based descriptions of electronic correlation in small- and medium-sized molecules. The fact that the CC equations with double excitations (CCD) may be expressed as a handful of dense matrix−matrix multiplications makes it an ideal method to be ported to graphics processing units (GPUs). We present our implementation of the spin-free CCD equations in which the entire iterative procedure is evaluated on the GPU. The GPU-accelerated algorithm readily achieves a factor of 4−5 speedup relative to the multithreaded CPU algorithm on same-generation hardware. The GPU-accelerated algorithm is approximately 8−12 times faster than Molpro, 17−22 times faster than NWChem, and 21−29 times faster than GAMESS for each CC iteration. Single-precision GPU-accelerated computations are also performed, leading to an additional doubling of performance. Single-precision errors in the energy are typically on the order of 10^{-6} hartrees and can be improved by about an order of magnitude by performing one additional iteration in double precision.

1. INTRODUCTION

The accuracy and extensibility of computational chemistry methods, particularly those which approximately solve the Schrödinger equation, are ultimately limited by the speed at which computer processors can execute floating point and memory operations. Due to fundamental limitations in processor technology, clock speeds are not increasing, and all future increases in computational capability are expected to come from parallelism, which now more than ever can be found within a single processor. Graphics processing units (GPUs) are a type of massively parallel processor in which hundreds of cores can execute many instructions at once, provided they are sufficiently regular. Recently, many groups have demonstrated the incredible power of GPUs for scientific applications when sufficient effort is devoted to programming them to exploit their high degree of instruction-level parallelism.[1] The programmability of GPUs has increased dramatically with the NVIDIA CUDA API[2] and associated SDK including CUBLAS and CUFFT, although these tools as well as the vendor-independent alternative, OpenCL,[3] require more programming effort to realize the same relative performance as CPUs, especially for irregular algorithms.

To date, many computational chemistry methods have been implemented on GPUs (or other accelerators), including classical molecular dynamics,[4−7] atomic integrals,[8−10] density functional[11−14] and Hartree−Fock theory,[15,16] low-order perturbation theory,[17−19] and quantum Monte Carlo (QMC) for both fermions[20,21] and bosons.[22] Related efforts include the development of fast multipole methods for biomolecular electrostatics.[23] Notably lacking is an implementation of a high-accuracy many-body method, such as coupled cluster (CC) or configuration-interaction (CI), for GPUs. Both CC and CI have high floating point cost (N^6 or greater, where N is the number of electrons correlated) and are memory intensive, hence they are ideally suited for GPUs, which have significantly greater floating point capability and memory bandwidth than equivalently priced CPUs.

In this paper, we report the first demonstration of CC executed entirely on GPUs. Specifically, the coupled cluster doubles method (CCD) has been implemented using CUDA and the associated dense linear algebra routines (BLAS). The utility of BLAS, specifically dense matrix−matrix multiplication (MMM) kernels, to achieve high performance in coupled cluster methods is well-known, having been central to the implementation of CC for vector processors in the 1980s[24,25] followed by scalar and superscalar processors in the 1990s.[26] However, simply moving BLAS calls from the CPU to the GPU is not sufficient to achieve good performance. The modest amount of memory available on the GPU relative to the CPU requires the programmer to move data back and forth between the two devices across the PCI bus, which has limited bandwidth compared to that within each device. Simply inlining GPU BLAS calls would generate significantly more data motion than necessary and inhibit performance significantly. Our implementation of CCD minimizes memory motion both by organizing BLAS calls on the GPU in an optimal way and by performing all other computations (e.g., tensor permutations) on the GPU. We utilized both the independence of the different terms in CCD to decompose the calculation as well as by splitting large arrays into tiles when the total input and output required for a particular BLAS call could not fit on the GPU. These two strategies allow us to realize a significant fraction of the theoretical performance possible for CCD and are extensible to more complicated CC theories or other methods relying heavily upon tensor contractions. While the cost of tensor permutations is not significant in CCD, it will become a greater portion of the run time for CCSD

Received: October 13, 2010
Published: April 15, 2011

and higher-order CC methods, hence our code addresses an important issue for implementing more complex theories.

CCD is quite reasonable as a first demonstration of iterative CC on GPUs. The addition of singles provides a method (CCSD) which is not significantly more expensive,[27] but CCD contains all of the same computational bottlenecks, and its simplicity facilitates the design of an optimal algorithm. In addition, accurate energies require an approximate treatment of triples, e.g., CCSD(T), and it is well-known that the non-iterative triples contribution beyond CCSD is a series of very large MMMs even more amenable to GPUs than CCD.[28−30] Hence, the implementation strategy and performance analysis in this paper are immediately applicable to CCSD(T), which is the ultimate goal of our ongoing efforts in this area. Very recently, Kowalski and co-workers demonstrated an 8-fold speedup using GPUs for the computation of perturbative triples corrections.[30]

The performance of our implementation of spin-free CCD is analyzed in two ways. First, the same code is executed on both CPU and GPU; the code runs 4.0−5.2 times faster on a single NVIDIA C2050 GPU than it does on two Intel Xeon CPUs. In both cases, the hardware is fully utilized with vendor-optimized BLAS routines (CUBLAS and Intel MKL), and other routines are parallelized for the GPU using CUDA. Second, we compare to several implementations of CCD found in well-known electronic structure packages, including GAMESS,[31] NWChem,[32] and Molpro.[33] Running on the C2050, our CCD implementation outperforms all other CCD implementations by at least a factor of eight when using double precision throughout. An additional factor of two speed-up can be achieved by performing GPU computations in single precision. Single precision computations give errors on the order of 10^{-6} hartrees; a single iteration in double precision following convergence in single precision reduces the numerical error to 10^{-7} hartrees. While far from the "magic" 100-fold speed-up observed in other applications, both the CPU and GPU implementations use tuned BLAS libraries, which precludes a defective comparison between CPU and GPU implementations of vastly different quality.[34] In fact, the comparison of algorithms dominated by BLAS routines favors the CPU since we find that tuned BLAS libraries for CPUs achieve a higher percentage of theoretical peak than their GPU counterparts, although we find that the gap in implementation quality of BLAS has decreased significantly in CUDA 3.2.

This paper is organized as follows: In Section 2, the equations of CCD are presented, followed by a description of their implementation (Section 3). Performance results and analysis of numerical precision for hydrocarbons with as many as 20 carbon atoms can be found in Section 4. Section 5 contains our conclusions and a discussion of future work.

2. THEORY

A detailed perspective of coupled cluster theory is available in the literature,[26,35,36] so we describe only the equations necessary to understand the specifics of our algorithm. The cluster amplitudes, t_{ij}^{ab}, for the spin-free CC equations with double excitations are determined by the solution of the set of nonlinear equations

$$D_{ij}^{ab}t_{ij}^{ab} = v_{ij}^{ab} + P(ia,jb)\Big[t_{ij}^{ae}I_e^b - t_{im}^{ab}I_j^m + \frac{1}{2}v_{ef}^{ab}t_{ij}^{ef} + \frac{1}{2}t_{mn}^{ab}I_{ij}^{mn} - t_{mj}^{ae}I_{ie}^{mb} - I_{ie}^{ma}t_{mj}^{eb} + (2t_{mi}^{ea} - t_{im}^{ea})I_{ej}^{mb}\Big] \tag{1}$$

Table 1. Dimensions of the Matrix−Matrix Multiplications That Comprise the Spin-Free CCD Equations[a]

	dimension		
no. of occurrences	M	N	K
1	o^2	o^2	v^2
1	o^2	v^2	v^2
1	o^2	v^2	o^2
6	ov	ov	ov
1	v	v	o^2v
1	v	o^2v	v
1	o	o	ov^2
1	o	ov^2	o

[a] The dimensions correspond to C(M × N) = A(M × K)·B(K × N). The symbols o and v refer to the number of occupied and unoccupied orbitals in the reference function.

where we have slightly modified the tensors given by Piecuch et al.:[26]

$$I_b^a = (-2v_{eb}^{mn} + v_{be}^{mn})t_{mn}^{ea} \tag{2}$$

$$I_j^i = (2v_{ef}^{mi} - v_{ef}^{im})t_{mj}^{ef} \tag{3}$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij}t_{kl}^{ef} \tag{4}$$

$$I_{jb}^{ia} = v_{jb}^{ia} - \frac{1}{2}v_{eb}^{im}t_{jm}^{ea} \tag{5}$$

$$I_{bj}^{ia} = v_{bj}^{ia} + v_{be}^{im}\left(t_{mj}^{ea} - \frac{1}{2}t_{mj}^{ae}\right) - \frac{1}{2}v_{be}^{mi}t_{mj}^{ae} \tag{6}$$

$$D_{ij}^{ab} = f_{ii} + f_{jj} - f_{aa} - f_{bb} \tag{7}$$

Here the indices i, j, k, l, m, and n (a, b, c, d, e, and f) represent those orbitals that are occupied (unoccupied) in the reference function. We have used the Einstein summation convention in which repeated upper and lower indices are summed; note, however, that the left-hand side of eq 1 involves no sum. The permutation operator, $P(ia,jb)$, involves a sum of two terms: $P(ia,jb)v_{ij}^{ab} = v_{ij}^{ab} + v_{ji}^{ba}$. The Fock matrix elements are denoted by f_{pq}, and the two-electron integrals are given by

$$v_{ij}^{ab} = \int\int \varphi_a^*(1)\varphi_b^*(2)\frac{1}{r_{12}}\varphi_i(1)\varphi_j(2) \tag{8}$$

where ϕ represents a canonical molecular orbital. In this spin-free representation, eqs 1−6 require 9 MMMs that scale as the sixth power of system size and 4 that scale as the fifth power of system size. The dimensions for these multiplications are given in Table 1.

Equation 1 is solved iteratively, beginning with an MP2 guess for t_{ij}^{ab}. Evaluating the right-hand side of eq 1 yields the updated amplitudes. The algorithm proceeds by simple substitution until the convergence criterion is satisfied, when the norm of the change in t_{ij}^{ab} between iterations falls below 1×10^{-7}. If all integrals and amplitudes are stored on the device, the evaluation of this norm represents the only communication between device and host following the initial copy of the Fock matrix and the two-electron integrals to the GPU. However, if the v_{cd}^{ab} block of integrals is prohibitively large, the associated MMM may be blocked, and the integrals copied to the device as needed.

3. COMPUTATIONAL DETAILS

The CC equations presented in eqs 1−7 were implemented in both single (SP) and double precision (DP) for computations with CPU and GPU hardware. The CPU implementation evaluates the tensor contractions with dense matrix multiplication routines (SGEMM or DGEMM) provided by Intel MKL 10.2. The CPU hardware utilized was a dual socket quad-core 2.67 GHz Intel Xeon X5550 processor with 36 GB of available memory. Both NVIDIA C1060 Tesla and NVIDIA C2050 Tesla graphics processors with total memories of 4 and 2.6 GB (with ECC enabled), respectively, were used to perform GPU computations. See Table 2 for important hardware parameters for both the CPU and GPU processors used in this study. The SP and DP GPU implementations were achieved using an identical algorithm with the MKL BLAS routines substituted with the corresponding routines of the CUBLAS 3.2 library. The required one- and two-electron integrals were generated by the GAMESS electronic structure package on a CPU. For SP GPU computations, the Fock matrix was computed on the host in DP before demoting its elements to SP for use on the device.

Table 3 presents pseudocode for three different algorithms to evaluate one of the more complicated diagrams of CCD using either a CPU or GPU. The naive GPU implementation, labeled GPU 1, copies data to and from the GPU for nearly every MMM and performs all tensor permutations and amplitude updates on the CPU. These operations represent wasted opportunity for acceleration by the enhanced memory bandwidth of the GPU.

Table 2. Summary of Hardware Details for the Processors Used[a]

	CPU	GPU	
	X5550[b]	C1060[c]	C2050[d]
processor speed (MHz)	2660	1300	1150
memory bandwidth (GB/s)	32	102	144
memory speed (MHz)	1066	800	1500
ECC available	yes	no	yes
SP peak (GF)	85.1	933	1030
DP peak (GF)	42.6	78	515
power usage (W)	95[e]	188	238

[a] Note that details are given for a single Intel X5550 processor, but two processors were used for all CPU calculations. [b] http://ark.intel.com/Product.aspx?id=37106. [c] http://www.nvidia.com/object/product_tesla_c1060_us.html. [d] http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html. [e] CPU power usage does not include CPU DRAM, whereas GPU power usage includes GPU DRAM.

Table 3. CPU and GPU Algorithms for Evaluating One Diagram of CCD[a]

CPU	GPU 1	GPU 2
	$I^{ia}_{jb} = v^{ia}_{jb} - 1/2 v^{im}_{eb} t^{ea}_{jm}$	
		cudaMemcpy$(t_{\text{gpu}} \leftarrow t(e,a,j,m))$
		cudaMemcpy$(v^{1}_{\text{gpu}} \leftarrow v(i,m,e,b))$
		cudaMemcpy$(v^{2}_{\text{gpu}} \leftarrow v(i,a,j,b))$
$t'(a,j,m,e) \leftarrow t(e,a,j,m)$	$t'(a,j,m,e) \leftarrow t(e,a,j,m)$	$t'_{\text{gpu}}(a,j,m,e) \leftarrow t_{\text{gpu}}(e,a,j,m)$
$v^{1}(m,e,i,b) \leftarrow v(i,m,e,b)$	$v^{1}(m,e,i,b) \leftarrow v(i,m,e,b)$	$v'_{\text{gpu}}(m,e,i,b) \leftarrow v^{1}_{\text{gpu}}(i,m,e,b)$
$I(a,j,i,b) = v(i,a,j,b)$	$v^{2}(a,i,j,b) \leftarrow v(i,a,j,b)$	$I_{\text{gpu}}(a,j,i,b) = v^{2}_{\text{gpu}}(i,a,j,b)$
	cudaMemcpy$(v'_{\text{gpu}} \leftarrow v^{1})$	
	cudaMemcpy$(t'_{\text{gpu}} \leftarrow t')$	
	cudaMemcpy$(I_{\text{gpu}} \leftarrow v^{2})$	
$I(a,j,i,b) - = 1/2t'(a,j,m,e)$- $v^{1}(m,e,i,b)$ (**DGEMM**)	$I_{\text{gpu}}(a,j,i,b) - = 1/2t'_{\text{gpu}}(a,j,m,e)$- $v'_{\text{gpu}}(m,e,i,b)$ (**cublasDgemm**)	$I_{\text{gpu}}(a,j,i,b) - = 1/2t'_{\text{gpu}}(a,j,m,e)$- $v'_{\text{gpu}}(m,e,i,b)$ (**cublasDgemm**)
	$R^{ab}_{ij} + = -t^{ae}_{mj} I^{mb}_{ie} - I^{ma}_{ie} t^{eb}_{mj}$	
$R'(b,i,a,j) = I(b,i,m,e)[t'(a,j,m,e)]^{T}$ (**DGEMM**)	$v'_{\text{gpu}}(b,i,a,j) = I_{\text{gpu}}(b,i,m,e)[t'_{\text{gpu}}(a,j,m,e)]^{T}$ (**cublasDgemm**)	$v'_{\text{gpu}}(b,i,a,j) = I_{\text{gpu}}(b,i,m,e)[t'_{\text{gpu}}(a,j,m,e)]^{T}$ (**cublasDgemm**)
	cudaMemcpy$(R' \leftarrow v'_{\text{gpu}})$	
$R(a,b,i,j) - = R'(b,i,a,j) + R'(a,j,b,i)$	$R(a,b,i,j) - = R'(b,i,a,j) + R'(a,j,b,i)$	$R_{\text{gpu}}(a,b,i,j) - = v'_{\text{gpu}}(b,i,a,j) + v'_{\text{gpu}}(a,j,b,i)$
$t'(m,e,b,j) \leftarrow t(e,b,m,j)$	$t'(m,e,b,j) \leftarrow t(e,b,m,j)$	$t'_{\text{gpu}}(m,e,b,j) \leftarrow t_{\text{gpu}}(e,b,m,j)$
	cudaMemcpy$(t'_{\text{gpu}} \leftarrow t')$	
$R'(a,i,b,j) = I(a,i,m,e)t'(m,e,b,j)$ (**DGEMM**)	$v'_{\text{gpu}}(a,i,b,j) = I_{\text{gpu}}(a,i,m,e)t'_{\text{gpu}}(m,e,b,j)$ (**cublasDgemm**)	$v'_{\text{gpu}}(a,i,b,j) = I_{\text{gpu}}(a,i,m,e)t'_{\text{gpu}}(m,e,b,j)$ (**cublasDgemm**)
	cudaMemcpy$(R' \leftarrow v'_{\text{gpu}})$	
$R(a,b,i,j) - = R'(a,i,b,j) + R'(b,j,a,i)$	$R(a,b,i,j) - = R'(a,i,b,j) + R'(b,j,a,i)$	$R_{\text{gpu}}(a,b,i,j) - = v'_{\text{gpu}}(a,i,b,j) + v'_{\text{gpu}}(b,j,a,i)$
update t with R	update t with R	update t_{gpu} with R_{gpu}
	GPU storage requirements	
0	$3o^{2}v^{2}$	$5o^{2}v^{2}$

[a] Note that for both GPU implementations, the temporary array v'_{gpu} can be reused. The naive GPU implementation, GPU 1, performs many unnecessary memory transfers and fails to exploit the high memory bandwidth of the GPU for additions and tensor permutations. For GPU implementation 2, all memory transfers can occur before CCD iterations begin, and the amplitudes can be directly updated on the GPU. R denotes the residual of the CCD equations and is defined by the right-hand side of eq 1.

The algorithm labeled GPU 2 is a better implementation that moves data between host and device as seldom as possible and performs all tensor permutations and updates to the CC amplitudes on the GPU. Algorithm GPU 2 is designed such that all memory transfers occur before the CC iterations begin. By monitoring convergence on the device, the amplitudes need never be copied from the GPU. Algorithm GPU 2 has greater memory requirements than GPU 1 for the evaluation of this diagram, but the temporary arrays used therein are useful in the evaulation of the remainder of the CCD equations, and the difference in storage requirements is insignificant when considering the storage of the v_{cd}^{ab} block of integrals. The present GPU implementation is most similar to GPU 2.

For systems with hundreds of active orbitals, the limited memory of GPUs necessitates an algorithm which repeatedly copies data from CPU to GPU memory. In single precision, the global memory of the C1060 GPU can accommodate the v_{cd}^{ab} block of integrals (the largest array required by CCD or CCSD) for no more than 181 virtual orbitals, and the storage requirements for the CC amplitudes and all other blocks of integrals further limit the size of systems which can be treated before such repeated memory transfers become necessary. In our algorithm, all two-electron integrals and CC amplitudes are stored in GPU memory whenever possible. In addition, we allocate several arrays to contain convenient permutations of the amplitudes and smaller blocks of integrals. While on-device storage of the two-electron integrals and CC amplitudes will ultimately limit the size of the applications that may be treated with our code, the lack of significant host/device communication will result in a best case for performance acceleration. To extend the applicability of this implementation, large MMMs can be blocked, and integrals copied to the GPU on-demand. Such tiling of the matrix multiplication involving the v_{cd}^{ab} block of two-electron integrals will alleviate some of these memory limitations but will result in the new overhead of transferring the integrals to the device every iteration. A different approach to large matrix multiplications is the streaming approach employed in LINPACK, wherein the matrix dimensions are much greater than 10 000. We are currently exploring whether this approach is suitable for the evaluation of the CC equations. The two-electron integral memory bottleneck may be entirely avoided through integral-direct techniques, provided one has an atomic integral code that can run on the GPU.

3.1. Comparison to Other Codes. The performance of the multicore CPU and GPU implementations of CCD is compared to a number of implementations in well-known packages. Only the GAMESS package implements the same equations as ours does, but we have also compared to the Molpro and NWChem implementation of CCD. Timings for CCSD are reported for these three packages as well as PSI3[37] to estimate the performance of a GPU CCSD code. It is important to point out if and how each CPU implementation is parallelized, given the essential role of fine-grain parallelism in our implementation. None of the codes tested have threading in their CC codes. In principle, all can take advantage of threading in the BLAS library, but this only improves performance for large matrices (dimension $\gtrsim$500). Both NWChem and Molpro are parallelized using global arrays. It was previously determined that NWChem performance is no worse, and in some cases significantly better, when one core is dedicated to communication, which is an artifact of optimizing interprocess communication within the node.[38] Hence, all NWChem and Molpro jobs used only seven cores for computation. For consistency, all computations are performed in C_1 symmetry.

3.1.1. GAMESS. The CCD algorithm contained in the GAMESS package is detailed in ref 26 and is essentially identical to that presented herein. The algorithm makes heavy use of BLAS DGEMM calls, but these are constrained by design to only use one thread, significantly restricting performance. We note that the present algorithm, when executing on a single CPU core, performs nearly identically to the GAMESS algorithm.

3.1.2. NWChem. NWChem implements CC in two different modules: the first is spin-free and AO-direct in the three- and four-virtual integrals,[39] while the second module (TCE) uses the more expensive spin−orbital CC equations.[40] Due to the much larger memory footprint of the two-electron integrals in spin−orbital form, a comparison could be made to this procedure only for small systems. However, the TCE module also permits of the use spin-free integrals with spin−orbital amplitudes, which provides the generality of the spin−orbital representation with a much-reduced memory footprint.[41]

Neither the TCE implementation of CCD and CCSD nor the partially direct spin-free implementation of CCSD in NWChem are directly comparable to our CCD code. However, as NWChem is parallel throughout, it is capable of utilizing a large number of CPU cores per node, unlike GAMESS.

3.1.3. Molpro. Molpro[33] implements CCD as a special case of spin-free CCSD where single excitations are set to zero. The CCD algorithm is thus not optimal and performs identically to the CCSD algorithm. The spin-free implementation is described in ref 42 .

4. RESULTS

Three different methods were used to evaluate the performance of CCD on CPU and GPU processors. First, we compare both our CPU and GPU implementations of CCD to those found in GAMESS, Molpro, and NWChem. Second, the performance benefit of using single-precision is considered due to the large gap in single- and double-precision performance of some GPUs. Finally, the performance of MMM was measured in both single and double precision on the CPU and GPU to establish an approximate upper bound on the performance of a CC code implemented using BLAS for tensor contractions. The performance of the present implementations is compared to that of the underlying BLAS routines.

The polyacetylene series, C_nH_{n+2}, with n ranging from 8 to 18, the acene series for 2-, 3-, and 4-fused benzene rings, and the smallest fullerene, C_{20}, were used to evaluate the performance of the CC implementations. The 6-31G basis set used throughout was not selected on the basis of chemical considerations but rather to allow us to consider a wide range of system sizes. By using a relatively small basis set, more emphasis is placed upon the performance of tensor contractions involving more than two occupied indices. A very large basis set places almost all the computational work in the evaluation of a single diagram involving four virtual indices. While this may provide a very high flop rate due to the presence of a very large MMM call, it is not particularly useful for evaluating implementation quality.

4.1. Comparison of CPU and GPU Implementations. The per iteration computational costs of our double-precision GPU and CPU algorithms are compared to implementations found in well-known electronic structure packages in Table 4. For our CPU and GPU implementations, the timings correspond to

Table 4. Comparison of CPU and GPU Implementations of CCD[a]

			present implementation			X5550		
molecule	o	v	C1060	C2050	X5550	GAMESS	Molpro	NWChem (TCE)[b]
C_8H_{10}	21	63	0.8	0.3	1.3	6.2	2.3	5.1 (4.7)
$C_{10}H_8$	24	72	1.5	0.5	2.5	12.7	4.8	10.6 (10.1)
$C_{10}H_{12}$	26	78	2.5	0.8	3.5	19.7	7.1	16.2 (15.1)
$C_{12}H_{14}$	31	93	7.1	2.0	10.0	57.7	17.6	42.0
$C_{14}H_{10}$	33	99	10.2	2.7	13.9	78.5	29.9	59.5
$C_{14}H_{16}$	36	108	16.7	4.5	21.6	129.3	41.5	90.2
C_{20}	40	120	29.9	8.8[c]	40.3	238.9	103.0	166.3
$C_{16}H_{18}$	41	123	35.9	10.5[c]	50.2	279.5	83.3	190.8
$C_{18}H_{12}$	42	126	42.2[c]	12.7[c,d]	50.3	329.4	111.8	218.4
$C_{18}H_{20}$	46	138	73.0[c]	20.1[c,d]	86.6	555.5	157.4	372.1

[a] Timings per CC iteration are given in seconds. The symbols o and v represent the number of doubly occupied and virtual orbitals in each system, respectively. [b] Numbers in parentheses are for uncompacted (spin−orbital) two-electron integrals, which runs out of memory for even medium-sized jobs. [c] The MMM involving v_{cd}^{ab} was tiled. [d] Some two-electron integrals were pushed to the GPU every iteration.

Table 5. Comparison of GAMESS, Molpro, NWChem, and PSI3 Implementations of CCD and CCSD[a]

	GAMESS		Molpro		NWChem			PSI3
molecule	CCD	CCSD	CCD	CCSD	CCSD[b]	CCD[c]	CCSD[c]	CCSD
C_8H_{10}	6.2	7.2	2.3	2.4	9.6/3.6	5.1	8.4	7.9
$C_{10}H_8$	12.7	15.3	4.8	5.1	22.8/8.2	10.6	16.8	17.9
$C_{10}H_{12}$	19.7	23.6	7.1	7.2	20.5/11.3	16.2	25.2	23.6
$C_{12}H_{14}$	57.7	65.1	17.6	19.0	53.6/29.4	42.0	64.4	54.2
$C_{14}H_{10}$	78.5	92.9	29.9	31.0	92.7/49.1	59.5	90.7	61.4
$C_{14}H_{16}$	129.3	163.7	41.5	43.1	103.2/65.0	90.2	129.2	103.4
C_{20}	238.9	277.5	103.0	102.0	294.6/175.7	166.3	233.9	162.6
$C_{16}H_{18}$	279.5	345.8	83.3	84.1	169.1/117.5	190.8	267.9	192.4
$C_{18}H_{12}$	329.4	380.0	111.8	116.2	274.2/178.6	218.4	304.5	216.4
$C_{18}H_{20}$	555.5	641.3	157.4	161.4	278.1/216.3	372.1	512.0	306.9

[b] The spin-free CCSD code in NWChem is integral direct for the terms with integrals having 3 or 4 virtual indices. The first number is the first iteration when the stored integrals (those with 0−2 virtual indices) are computed and written to disk; the second number is for subsequent iterations when the stored integrals are read from disk. [c] TCE implementation with spin−orbital CC equations but using compacted (spin-free) integrals. [a] The data given are seconds per CC iteration.

those required to evaluate all tensors given by eqs 2−6 and to update the CC amplitudes according to eq 1. Timings for the GPU version exclude the initial integral push to the device. For standard packages, timings correspond to only the iterative portions of the CCD algorithms; integral generation and sorting were excluded. The four-index transformation and the related I/O and processing are excluded for three reasons: (1) the implementation of these procedures varies greatly between different packages, (2) for larger calculations the time required to generate the integrals is insignificant, as it scales as N^5 while many CC diagrams scale as N^6, and (3) the transfer time from the CPU to GPU will disappear when it is possible to compute all integrals directly on the GPU.

The C2050 GPU-accelerated CCD algorithm outperforms all other implementations on a per iteration basis. As compared to the threaded CPU implementation, we observe application accelerations of 4.0−5.2 for all systems considered. In addition to the acceleration of MMMs, all required transposes are performed on the GPU, which has significantly higher memory bandwidth than the CPU, making these operations much faster. The C1060 GPU-accelerated algorithm performs anywhere from 2.6 to 3.7 times worse than the C2050 algorithm. This generation of hardware provides relatively poor double-precision performance and is thus not optimal for general-purpose scientific computing.

Aside from our CPU algorithm, the best CPU CCD implementation presented here is that found in Molpro. Because CCD is implemented in Molpro as a special case of CCSD, the addition of singles results in a marginal increases in computational cost, and the comparison to the GPU-accelerated CCD code is thus slightly biased in favor of the GPU. Regardless of this deficiency, the Molpro CCD implementation is still the best available, and the GPU-accelerated algorithm consistently outperforms it by a factor of 8−12. For the largest system studied, $C_{18}H_{20}$, a single iteration in Molpro requires about 2.5 min, while a C2050 iteration requires only 20 s.

For comparison, we present in Table 5 the costs for both CCD and CCSD in GAMESS, Molpro, and NWChem. We have also included the CCSD timings for PSI3, which does not implement the CCD method. The addition of single excitations does not significantly increase the cost of CCD, as supported by the relative costs of CCD and CCSD given here. The GAMESS CCD and CCSD timings suggest that the addition of single excitations in our algorithm will increase the cost of the serial CPU algorithm by around 25%. The cost of the threaded CCSD code may be slightly worse than expected due to a less-than-optimal performance of threaded MKL BLAS for some of the very small MMMs that arise in CCSD. Nonetheless, we predict that the CCSD algorithm given in ref 26 would be several times faster than the best available CPU-based CCSD algorithms if implemented properly for GPU hardware.

As stated above, the limited global memory associated with GPUs limits the size of the applications that may be directly treated by eq 1, especially in double precision. For larger systems, the v_{cd}^{ab} block of integrals will not fit into global memory, and it is necessary to break up the associated MMM into multiple calls for subblocks of the input arrays (known as tiling). Those instances in which tiling was necessary are noted accordingly in Table 4. In

Table 6. SP and DP GPU and CPU Timings in Seconds[a]

	time (s)							
	C1060		C2050		X5550		error (μE_h)	
molecule	SP	DP	SP	DP	SP	DP	SP	mixed
C_8H_{10}	0.2	0.8	0.2	0.3	0.7	1.3	0.05	0.01
$C_{10}H_8$	0.4	1.5	0.2	0.5	1.3	2.5	−0.42	−0.04
$C_{10}H_{12}$	0.7	2.5	0.4	0.8	2.0	3.5	−0.13	−0.02
$C_{12}H_{14}$	1.8	7.1	1.0	2.0	5.6	10.0	−0.30	−0.04
$C_{14}H_{10}$	2.6	10.2	1.5	2.7	8.4	13.9	−3.74	−0.35
$C_{14}H_{16}$	4.1	16.7	2.4	4.5	12.1	21.6	−1.00	−0.16
C_{20}	6.7	29.9	4.1	8.8[b]	22.3	40.3	−1.43	0.09
$C_{16}H_{18}$	9.0	35.9	5.0	10.5[b]	28.8	50.2	−2.66	−0.44
$C_{18}H_{12}$	10.1	42.2[b]	5.6	12.7[b,c]	29.4	50.3	−15.03	−1.30
$C_{18}H_{20}$	17.2	73.0[b]	10.1[b]	20.1[b,c]	47.0	86.6	−5.72	−0.91

[a] Errors in the energy for single- and mixed-precision algorithms are presented. The mixed-precision algorithm converges in SP and performs one iteration in DP. All errors are given in units of μE_h (10^{-6} hartrees). [b] The MMM involving v_{cd}^{ab} was tiled. [c] Some two-electron integrals were pushed to the GPU every iteration.

addition, some permutations of integrals that scale as o^2v^2 were, in some cases, pushed to the GPU as needed. The timings presented in Table 4 include all memory transfers occurring during the iterative portions of the algorithm, and the overhead associated with these transfers has minimal implications for performance. In the case of $C_{18}H_{20}$, the v_{cd}^{ab} block of integrals represents 2.7 GB of data. These data, as well as several hundreds of MB of o^2v^2 integrals, are pushed to the GPU every iteration; regardless, the GPU-accelerated algorithm is more than four times faster per iteration than the corresponding CPU implementation and almost eight times faster than the Molpro package. This result suggests that larger systems can be treated with this algorithm provided that the larger MMMs are appropriately tiled. The notion of tiling leads naturally to a framework for many GPU CC; scalable parallelization can be realized by distributing comparably sized tiles among many GPUs.

4.2. Impact of Numerical Precision upon Accuracy and Performance. Many implementations of scientific algorithms on GPU hardware utilize single or mixed precision due to the markedly reduced DP performance of older graphics cards. As accelerator software and hardware mature for HPC, GPUs are becoming increasingly efficient at performing DP operations. On the other hand, most commodity graphics processors cannot support DP. It is important to understand the performance advantages of low-precision computing as weighed against the disadvantages. We have implemented our CCD algorithm in SP on the same CPU and GPU hardware discussed above, and the SP timings are presented in Table 6. In general, we observe the same GPU/CPU accelerations for SP computations as we did for strictly DP computations. On the C2050 processor, SP per iteration costs are 4.2−5.8 times less than that of the SP costs of the CPU-based threaded MKL algorithm, and C2050 SP operations are roughly half the cost of DP operations. The advantages of SP computations are more evident for the older C1060 card, where DP is 3.9−4.4 times more expensive than SP.

Table 6 also lists the energy errors associated with the SP algorithm. For all systems investigated, the observed SP errors are at worst on the order of 10^{-5} E_h. Chemical accuracy is considered to be kcal/mol, which corresponds to roughly 1.6×10^{-3} E_h. Clearly, SP CCD on the GPU yields more than acceptable accuracy for single-point energy evaluations. Should the need for higher accuracy arise, it is not difficult to design an algorithm in which we converge to a SP solution, promote the amplitudes to DP, and perform a single iteration in DP. The results in Table 6 labeled "mixed" represent such an algorithm. A single CC iteration in DP on the C2050 graphics card costs about two times as much as a single iteration in SP and can increase the accuracy of the computation by an order of magnitude. Five of the mixed precision errors in Table 6 are below 1×10^{-7} E_h; the largest error decreased from -15.03 μE_h to only -1.30 μE_h. For larger systems, the SP energy errors may be larger, necessitating further DP iterations. Total application speedup is only marginally affected by the final DP iteration.

4.3. Matrix−Matrix Multiplication Performance. The performance of SGEMM and DGEMM with the X5550 using MKL and C2050 using CUBLAS 3.2 are shown in Figures 1 and 2, respectively. MKL SGEMM and DGEMM implementations are threaded and utilize eight threads. The GPU is superior in performance to the CPU for larger matrices (dimension greater than ~600 for SGEMM), and the performance of the GPU is

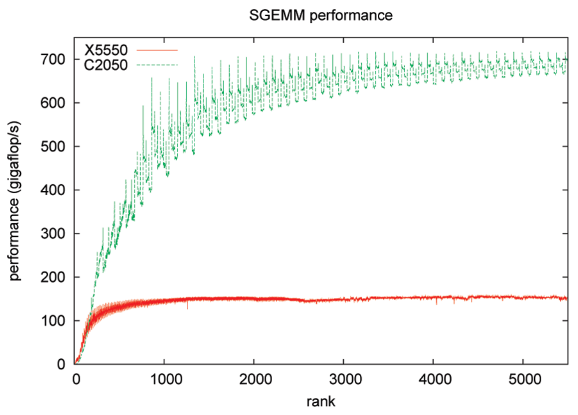

Figure 1. Performance in gigaflop/s (10^9 floating point operations per second) for SGEMM on CPU and GPU devices. The CPU SGEMM implementation utilizes eight threads. The maximum gigaflop/s for the CPU and GPU are 156.2 and 717.6, respectively.

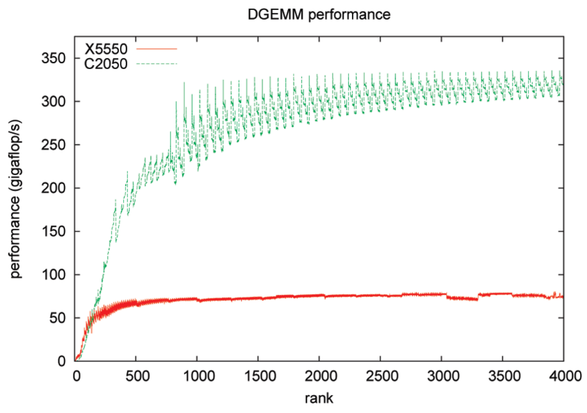

Figure 2. Performance in gigaflop/s (10^9 floating point operations per second) for DGEMM on CPU and GPU devices. The CPU DGEMM implementation utilizes eight threads. The maximum gigaflop/s for the CPU and GPU are 79.2 and 335.6, respectively.

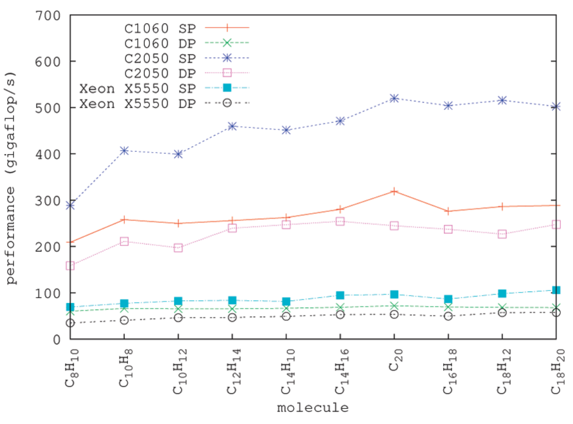

Figure 3. Performance in gigaflop/s (10^9 floating point operations per second) for different implementations of spin-free CCD. Results are given for both CPU and GPU hardware, and CPU BLAS routines utilize eight threads. S SP and DP performance data are given.

more than four times better than that of the CPU in the limit of very large matrices. The best-case performance of the GPU for SGEMM is greater than 700 gigaflop/s (GF), which is approximately 70% of the theoretical peak performance. On the C2050, the performance of DGEMM is approximately half of that of SGEMM, which again is approximately 70% of peak. The 2:1 ratio of single- and double-precision performance is new as of the Fermi GPU architecture; a much larger ratio of single to double precision performance was observed with the Tesla GPU architecture (C1060). The variation in the performance as a function of matrix dimension on the C2050 is more than on the CPU but significantly less than previous generations of NVIDIA hardware and software. The absolute performance of SGEMM and DGEMM also changed significantly upon the release of CUDA 3.2. Prior to this release, algorithm performance was tightly coupled to system size and could be maximized by padding matrix dimensions in an effort to match warp sizes. However, as of CUDA 3.2, padding appears to have been integrated into the implementation of BLAS, and our manual implementation of padding no longer improves performance. Finally, previous generations of CUBLAS achieve only ~50% of peak for very large matrices, whereas the latest version achieves ~70% of peak.

Assuming no overhead for computing transposes and data transfers, the CPU and GPU CCD algorithms should achieve the flop rates of the underlying DGEMM and SGEMM kernels depicted in Figures 1 and 2. Figure 3 depicts the performance of the CPU and GPU implementations of spin-free CCD for all of the systems studied here. The SP CPU implementation utilizes threaded MKL SGEMM calls and achieves 69−105 GF on the Intel Xeon X5550 CPU. The DP implementation achieves only half of that flop rate, 35−57 GF. The C2050 is capable of delivering at or near 500 GF of SP performance for systems larger than and including C_{20}, while DP performance approaches 250 GF. The release of CUDA 3.2 provides enhanced DP capabilities on the C2050 graphics card that is well beyond those of older Tesla products. For the C1060, we observe considerably lower flop rates of only 209−319 (60−72) GF for SP (DP).

5. CONCLUSIONS

We have reported the first implementation of the iterative procedures of any coupled cluster method running entirely on a GPU. We find that the NVIDIA C2050 graphics processor can achieve approximately 500 gigaflop/s performance in single precision (SP) and 250 gigaflop/s in double precision (DP). This performance translates to per iteration accelerations of 4.2−5.8 for SP and 4.0−5.2 for DP as compared to the multithreaded CPU implementation. The quality of both the CPU and GPU implementations are as similar as possible, as both employ the vendor-optimized BLAS libraries provided by Intel and NVIDIA, respectively. To the best of our knowledge, this is the first time such a direct evaluation of hardware performance has been undertaken for any quantum chemistry kernel; previous papers compare an optimized GPU implementation to a standard package written for CPUs or evaluated the impact of using GPUs for a limited set of procedures. In contrast, the entire iterative CC procedure is computed using the GPU, and the overhead of transferring integrals from CPU memory is demonstrated to be nominal. As most CPU packages are neither multithreaded (e.g., with OpenMP) nor optimized for vector floating point instructions (e.g., SSE3 for x86 processors), such a comparison is intrinsically unfair to the CPU. Unless special effort has been devoted to architecture-specific optimization of CPU code, as is done for BLAS calls, it is impossible to make a fair comparison of CPU and GPU hardware.

Our DP implementation running on the C2050 GPU processor is more than an order of magnitude faster than several well-known electronic structure packages for the CC iterative procedure, which dominates the total wall time of a CC calculation (when neglecting the perturbative triples correction). Specifically, the DP algorithm was shown to be 8−12 times faster than Molpro and 17−22 times faster than NWChem when each is executed on two quad-core CPUs. Our DP implementation is 21−29 times faster than the serial implementation of CCD implmented in GAMESS executing on a single CPU core. It is important to point out that none of the software packages tested make efficient use of multicore CPUs using threads. GAMESS is constrained by design to run on a single CPU core and using multiple threads in BLAS did not improve the performance of PSI3 as much as it did ours. For example, our CCD iteration timings for $C_{12}H_{14}$ reduced from 51 to 10 s when the number of threads increased from 1 to 8, whereas we found that the performance of PSI3 improved by less than 50% for the same 8-fold increase in thread utilization. While it was possible for NWChem and Molpro to utilize two quad-core CPUs using global arrays, this process-based parallelism necessarily divides the data into smaller chunks, reducing the efficiency of BLAS calls. In the end, the best predictor of the performance improvement of CC codes using GPUs instead of CPUs is our own CPU code, for which the performance improvement of four to five times is in good agreement with the relative performance of SGEMM and DGEMM we measured.

Because most GPUs have modest DP performance relative to what can be done in SP, computing in SP or some mixture of SP and DP can improve performance, provided the results are still numerically accurate. We demonstrate that CC is amenable to a very simple multiprecision algorithm due to its iterative nature and that we can converge to a standard DP threshold while performing all but one iteration in SP. While the state-of-the art NVIDIA C2050 (Fermi) processor has a similar ratio of SP to DP performance, as is found on CPU hardware, the older NVIDIA C1060 (Tesla) architecture and noncompute-oriented commodity GPUs have a much larger discrepancy between SP and DP. While the Fermi architecture may be more relevant to computational chemists using dedicated high-performance computing resources, which are likely to be equipped with more expensive hardware that is more suitable for scientific computation, the commodity GPU hardware found in laptop and desktop computers is likely to continue to provide substantially more performance in SP than DP. Thus, our mixed precision algorithm will still be relevant in the future. We also note that the mixed precision approach can be used on the CPU as well, but the performance gain will not be more than two times, as all modern CPU architectures we are aware of are optimized for DP floating point computation.

The present implementation of GPU-accelerated CCD attempts to store all two-electron integrals and CC amplitudes in global memory on the GPU device and is thus limited in its applicability to systems with less than 200 spatial orbitals. We have experimented with tiling the multiplication involving the v_{cd}^{ab} block of integrals to allow us to treat systems as large as $C_{18}H_{20}$ in a 6-31G basis in full double precision on the C2050 card, which has only 2.6 GB of global memory. It was shown that this system could be treated roughly four times more efficiently by the C2050 GPU than with the X5550 CPU, despite the fact that 3 GB of integrals were transferred to the GPU every iteration. An algorithm dominated by tiled DGEMM calls is also naturally amenable to parallelization. In the extension of this algorithm to multiple GPUs, the cost of the required MPI collectives will eventually dominate the integral push−pull time. Regardless of these arguments for tiled matrix multiplications, the memory limitation is completely artificial in the sense that it is coupled to current hardware limitations and will therefore change as GPU hardware matures for scientific applications; the NVIDIA C2070 card, which was not used in these experiments, has 6 GB of global memory.

The tremendous performance increase observed for these moderate systems can have profound implications for computations that require multiple energy evaluations. An obvious target of fast CC calculations on GPUs is ab initio molecular dynamics of small molecules. Such calculations might require higher than SP for accurate results, but as has been shown herein, it is trivial to design a mixed precision algorithm that can yield DP accuracy in SP time. Additionally, computations on clusters of GPUs would be ideal for local correlation approximations such as the clusters-in-molecule (CIM) approximation.[43,44] The CIM-CC methods are embarrassingly parallel, and the 1×10^{-6} E_h error for SP GPU algorithms is negligible compared to the corresponding CIM errors, which can be three orders of magnitude larger. Thus, it would be straightforward to utilize many GPUs by performing each cluster simulation on a single GPU, since no data needs to be communicated between subsystem calculations after the original partitioning of the molecule into clusters.

Future work will include the implementation of the full CCSD equations for similar studies in GPU acceleration. Based upon our current results and the performance of the GAMESS implementations of CCD and CCSD, we expect GPU-accelerated CCSD to be several times more efficient than any existing CPU implementation. Additionally, preliminary tests of our implementation of CC for multiple GPUs suggest that distributing independent diagrams allows for the utilization of 5−10 GPUs. Scaling to more than 10 GPUs requires breaking up a single diagram computation across multiple GPUs, which is more difficult due to increased communication but certainly possible for larger calculations. Because CCSD has more diagrams and because these vary greatly in computational cost, more care is required to load balance these calculations to fully utilize all available processor resources. However, we believe that significant speed-ups can be obtained by overlapping CPU and GPU computations in a hybrid CPU-GPU implementation of CCSD.

AUTHOR INFORMATION

Corresponding Author
*E-mail: adeprince@anl.gov.

ACKNOWLEDGMENT

A.E.D. is supported by the Computational Postdoctoral Fellowship program at Argonne National Laboratory. J.R.H. is supported by the Director's Postdoctoral Fellowship program at Argonne National Laboratory and thanks Alán Aspuru-Guzik for interesting discussions and encouragement.

This research used the "Dirac" GPU test bed system of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC02-05CH11231.

We gratefully acknowledge use of "Breadboard," a development computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC02-06CH11357.

We gratefully acknowledge use of "Fusion," a computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC02-06CH11357.

REFERENCES

(1) The line between instruction- and thread-level parallelism in a GPU is blurred relative a CPU, but we consider them to be effectively single-instruction multiple-data (SIMD) processors.

(2) *CUDA Programming Guide*; NVIDIA: Santa Clara, CA; http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf. Accessed March 10, 2011).

(3) Stone, J. E.; Gohara, D.; Shi, G. *Comput. Sci. Eng.* **2010**, *12*, 66.

(4) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *J. Comput. Chem.* **2007**, *28*, 2618.

(5) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comput. Phys.* **2008**, *227*, 5342.

(6) Liu, W.; Schmidt, B.; Voss, G.; Møller-Wittig, W. *Comput. Phys. Commun.* **2008**, *179*, 634.

(7) Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. *J. Comput. Chem.* **2009**, *30*, 864.

(8) Ufimtsev, I. S.; Martnez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222.

(9) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. *J. Chem. Theory Comput.* **2010**, *6*, 696.

(10) Titov, A. V.; Kindratenko, V. V.; Ufimtsev, I. S.; Martínez, T. J. In Proceedings of Symposium on Application Accelerators in High-Performance Computing (SAAHPC) ,Knoxville, TN, July 13−15, 2010; National Center for Supercomputing Applications, University of Illinois: Urbana-Champaign, IL, 2010.

(11) Yasuda, K. *J. Chem. Theory Comput.* **2008**, *4*, 1230.

(12) Woods, C. J.; Brown, P.; Manby, F. R. *J. Chem. Theory Comput.* **2009**, *5*, 1776.

(13) Genovese, L.; Ospici, M.; Deutsch, T.; Mehaut, J.-F.; Neelov, A.; Goedecker, S. *J. Chem. Phys.* **2009**, *131*, 034103.
(14) Brown, P.; Woods, C. J.; McIntosh-Smith, S.; Manby, F. R. *J. Comput. Chem.* **2010**, *31*, 2008.
(15) Ufimtsev, I. S.; Martnez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004.
(16) Ufimtsev, I. S.; Martnez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619.
(17) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049.
(18) Olivares-Amaya, R.; Watson, M. A.; Edgar, R. G.; Vogt, L.; Shao, Y.; Aspuru-Guzik, A. *J. Chem. Theory Comput.* **2010**, *6*, 135.
(19) Koniges, A.; Preissl, R.; Kim, J.; Eder, D.; Fisher, A.; Masters, N.; Mlaker, V.; Ethier, S.; Wang, W.; Head-Gordon, M. In Proceedings of Cray User Group (CUG), Edinburgh, Scotland, May 24−27, 2010; Cray User Group, Inc.: Corvallis, Oregon, 2010.
(20) Anderson, A. G.; Goddard, W. A., III; Schröder, P. *Comput. Phys. Commun.* **2007**, *177*, 298.
(21) Esler, K.; Kim, J.; Shulenburger, L.; Ceperley, D. Fully accelerating quantum Monte Carlo simulations of real materials on GPU clusters. *Comput. Sci. Eng.*; http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.122. Accessed March 10, **2011**).
(22) Gothandaraman, A.; Peterson, G. D.; Warren, G. L.; Hinde, R. J.; Harrison, R. J. *Parallel Comput.* **2008**, *34*, 278.
(23) Yokota, R.; Hamada, T.; Bardhan, J. P.; Knepley, M. G.; Barba, L. A. Biomolecular electrostatics using a fast multipole BEM on up to 512 GPUs and a billion unknowns. 2011, arXiv:1007.4591v3. arXiv.org ePrint archive. http://arxiv.org/abs/1007.4591. Accessed March 10, 2011).
(24) Kucharski, S. A.; Bartlett, R. J. *Theor. Chim. Acta* **1991**, *80*, 387.
(25) Stanton, J. F.; Gauss, J.; Watts, J. D.; Lauderdale, W. J.; Bartlett, R. J. *Int. J. Quantum Chem.* **1992**, *44*, 879.
(26) Piecuch, P.; Kucharski, S. A.; Kowalski, K; Musial, M. *Comput. Phys. Commun.* **2002**, *149*, 71.
(27) Scuseria, G. E.; Schaefer, H. F., III *J. Chem. Phys.* **1989**, *90*, 3700.
(28) Melicherčk, M.; Demovič, L.; Pitoňák, M.; Neogrády, P. In *Proceedings of the 9th Central European Symposium on Theoretical Chemistry*, 2010.
(29) Ma, W., Krishnamoorthy, S.; Villa, O.; Kowalski, K. In Proceedings of Cluster Computing and the Grid (CCGRID), IEEE/ACM International Symposium, Melbourne, Australia, May 17−20, 2010; IEEE: Piscataway, NJ, 2010.
(30) Ma, W.; Krishnamoorthy, S.; Villa, O.; Kowalski, K. *J. Chem. Theory Comput.* **2011,** in press.
(31) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S.; Windus, T. L.; Dupuis, M.; Montgomery, J. A., Jr. *J. Comput. Chem.* **1993**, *14*, 1347.
(32) Bylaska, E. J.; de Jong, W. A.; Govind, N.; Kowalski, K.; Straatsma, T. P.; Valiev, M.; Wang, D.; Aprà, E.; Windus, T. L.; Hammond, J.; Autschbach, J.; Nichols, P.; Hirata, S.; Hackler, M. T.; Zhao, Y.; Fan, P.-D.; Harrison, R. J.; Dupuis, M.; Smith, D. M. A.; Nieplocha, J. Tipparaju, V.; Krishnan, M.; Vazquez-Mayagoitia, A.; Wu, Q.; Van Voorhis, T.; Auer, A. A.; Nooijen, M.; Crosby, L. D.; Brown, E.; Cisneros, G.; Fann, G. I.; Früchtl, H.; Garza, J.; Hirao, K.; Kendall, R.; Nichols, J. A.; Tsemekhman, K.; Wolinski, K.; Anchell, J.; Bernholdt, D.; Borowski, P.; Clark, T.; Clerc, D.; Dachsel, H.; Deegan, M.; Dyall, K.; Elwood, D.; Glendening, E.; Gutowski, M.; Hess, A.; Jaffe, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; Pollack, L.; Rosing, M.; Sandrone, G.; Stave, M.; Taylor, H.; Thomas, G.; van Lenthe, J.; Wong, A.; Zhang, Z. *NWChem, A Computational Chemistry Package for Parallel Computers*, version 5.1.1; Environmental Molecular Sciences Laboratory (EMSL): Richland, WA, 2009.
(33) Werner,H.-J.; Knowles, P. J.; Manby, F. R.; Schütz, M.; Celani, P.; Knizia, G.; Korona, T.; Lindh, R.; Mitrushenkov, A.; Rauhut, G.; Adler, T. B.; Amos, R. D.; Bernhardsson, A.; Berning, A.; Cooper, D. L.; Deegan, M. J. O.; Dobbyn, A. J.; Eckert, F.; Goll, E.; Hampel, C.; Hesselmann, A.; Hetzer, G.; Hrenar, T.; Jansen, G.; Köppl, C.; Liu, Y.; Lloyd, A. W.; Mata, R. A.; May, A. J.; McNicholas, S. J.; Meyer, W.; Mura, M. E.; Nicklass, A.; Palmieri, P.; Pflüger, K.; Pitzer, R.; Reiher, M; Shiozaki, T.; Stoll, H.; Stone, A. J.; Tarroni, R.; Thorsteinsson, T.; Wang, M.; Wolf, A. *MOLPRO*, version 2010.1; University College Cardiff Consultants Limited: Wales, U.K., 2010; http://www.molpro.net.
(34) Lee, V. W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A. D.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P.; Singhal, R.; Dubey, P. *SIGARCH Comput. Archit. News* **2010**, *38*, 451.
(35) Crawford, T. D.; Schaefer, H. F., III *Rev. Comput. Chem.* **2000**, *14*, 33.
(36) Bartlett, R. J.; Musial, M. *Rev. Modern Phys.* **2007**, *79*, 291.
(37) Crawford, T. D.; Sherrill, C. D.; Valeev, E. F.; Fermann, J. T.; King, R. A.; Leininger, M. L.; Brown, S. T.; Janssen, C. L.; Seidl, E. T.; Kenny, J. P.; Allen, W. D. *J. Comput. Chem.* **2007**, *28*, 1610.
(38) Hammond, J. R.; Krishnamoorthy, S.; Shende, S.; Romero, N. A.; Malony, A. D. *Performance Characterization of Global Address Space Applications: A Case Study with NWChem* **2011**.
(39) Kobayashi, R.; Rendell, A. P. *Chem. Phys. Lett.* **1997**, *265*, 1.
(40) Hirata, S. *J. Phys. Chem. A* **2003**, *107*, 9887.
(41) Kowalski, K.; Hammond, J. R.; de Jong, W. A.; Fan, P.-D.; Valiev, M.; Wang, D.; Govind, N. Coupled Cluster Calculations for Large Molecular and Extended Systems. In *Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology*; Reimers, J. R., Ed.; Wiley: Hoboken, NJ, 2011.
(42) Hampel, C.; Peterson, K. A.; Werner, H.-J. *Chem. Phys. Lett.* **1992**, *190*, 1.
(43) Li, S.; Ma, J.; Jiang, Y. *J. Comput. Chem.* **2002**, *23*, 237.
(44) Li, S.; Shen, J.; Li, W. *J. Chem. Phys.* **2006**, *125*, 074109.