JCTC Journal of Chemical Theory and Computation

Multicore Parallelization of Kohn−Sham Theory

Christopher J. Woods, Philip Brown, and Frederick R. Manby*

Centre for Computational Chemistry, School of Chemistry, University of Bristol, Bristol BS8 1TS, U.K.

Received March 23, 2009

Abstract: A multicore parallelization of Kohn−Sham theory is described, using standard commodity multisocket and multisocket/multicore shared-memory processors. Near-linear scaling of the parallel parts of the code was observed up to the maximum of sixteen cores that were available for benchmarking, and an order of magnitude reduction in run time was achieved running using sixteen threads on a quad-socket quad-core Xeon system. The speed-ups achieved using multisocket/multicore processors were competitive with those achieved using numerical accelerator cards.

Introduction

Recent advances in method development, computing technology, and algorithm design have allowed electronic structure theory methods to be applied routinely to large biological molecules.[1−3] However, *ab initio* electronic structure methods are not yet widely used in biomolecular simulations for drug design or computational enzymology. The computational expense of *ab initio* calculations of biomolecular systems is such that access to large high performance computing (HPC) facilities is typically required. Drug design or computational enzymology simulations are commonly performed using commodity computing resources (e.g., desktop computers or Beowulf clusters), and so most calculations rely on inexpensive semiempirical methods, such as AM1,[4] PM3,[5] or tight binding.[6]

Recent changes in the way that commodity processors are designed have made it increasingly difficult for programmers to extract the full double-precision computing power that is available. In this paper we describe a parallel implementation of Kohn−Sham density functional theory (DFT)[7,8] optimized for modern commodity processors and fully benchmark this port to assess whether it is practical to use widely available computer hardware to run DFT simulations on biomolecular systems routinely.

Numerical Accelerators and Multicore Processors

Moore's law—devised in 1965—states that the number of transistors that can be placed inexpensively into an integrated circuit doubles every two years.[9] This observation has remained true for over forty years, and for most of this time, as the number of transistors in the processor doubled, so too did the clock speed. As clock speeds increased, each generation of processor became capable of performing more floating point operations per second, and so scientific applications automatically ran more quickly.

Recently, there has been a sea change. While transistor counts are still doubling, problems of managing power consumption and heat dissipation mean that the processor clock speed of commodity processors has remained static since around 2004 or indeed has even been falling. The extra transistors have been used to provide larger on-chip memory caches or to add extra processor units (called cores). Thus commodity dual-core processors (which can run two independent threads of execution simultaneously per processor) became available from 2005 and quad-core from 2007. Oct-core processors are likely to become available in 2009[10] (note that this follows the lead of the development of HPC processors, where dual-core chips, such as the IBM POWER4, have been available since 2001 and multisocket symmetric multiprocessor [SMP] nodes are common). Many existing scientific applications that were developed for commodity processors were designed to use only a single thread of execution. They can thus only use a single core of the processor and cannot automatically take advantage of any additional cores that are available. Thus, the automatic and dramatic reduction in run-times for calculations on commodity hardware is no longer assured.

* Corresponding author e-mail: fred.manby@bris.ac.uk.

One solution to this problem is provided by numerical accelerators. In these processor chips the large numbers of available transistors are arranged to create hundreds of minicores, such that large numbers of floating point operations can be performed in parallel. For example, general purpose graphic processing units (GPGPUs) evolved from 3D graphics processors and can perform hundreds of floating point operations simultaneously within each clock cycle. Dedicated numerical accelerators have also been developed, such as the ClearSpeed CSX600 and CSX700 chips, each of which can also perform hundreds of floating point operations in parallel.[11] While these accelerators provide large amounts of computing power, algorithms must be redesigned to fit the paradigm of performing hundreds or thousands of independent parallel computations. This requires significant effort, but the reward can be dramatic reductions in the run time of the calculation.

There has been significant interest and success in porting *ab initio* quantum chemistry programs to numerical accelerators.[12−18] Recently, we demonstrated a port of the DFT code from the Molpro quantum chemistry package[19] that was parallelized for ClearSpeed numerical accelerator cards.[13] In this work the Coulomb problem was reformulated to be dominated by numerical quadrature, and, as a result, good scaling was found over the 2304 processor elements available in a ClearSpeed CATs system. This port was capable of speeding up DFT calculations of medium-sized (30−50 atom) molecules by over an order of magnitude.[13] Calculations on molecules of about this size are needed for QM/MM calculations on biomolecular systems.[1]

The algorithms developed to port the DFT calculation to ClearSpeed are generally applicable to any processing platform that is capable of operating on multiple double-precision values in parallel. Commodity processors are now capable of this, both via vector instruction sets (such as SSE for Intel or AMD processors[20]) and by providing access to multiple cores. Dual-core or quad-core desktop processors are now readily available, and modern Beowulf computer clusters are now being built from large numbers of multi-socket/multicore processors. For example, a dual-socket/quad-core machine, here called a dual-quad, contains two quad-core processors that both share the same memory address space; on such a system eight threads of execution can run efficiently in parallel. If SSE2 is available, then a dual-quad system can perform sixteen double-precision operations at a time. Comomodity oct-core processors are likely to become readily available some time over the next year,[10] and it is not unreasonable to expect that commodity quad-oct (which could operate on 64 doubles at once) or even oct-oct platforms (128 doubles at once) will be marketed.

An advantage of these platforms over accelerators, in addition to their wide availability, is that they can be programmed using long-established and portable languages, such as OpenMP and MPI. This makes maintenance and porting of the code as processors evolve, and the number of cores increase, more straightforward. Indeed, there is a long history of parallelizing quantum chemical programs over HPC multisocket and multicore supercomputers. Through a combination of MPI and OpenMP, efficient scaling over hundreds or thousands of processor cores is readily achievable.[21−28]

One method of taking advantage of multicore/multisocket systems is to completely design the quantum chemical program from the ground up to use multiple threads of execution. For example, this is the approach taken by the PQS quantum chemical program,[29] that was developed since 1998 to use the multiple processors available in commodity Beowulf clusters. The ONETEP[30,31] DFT program also takes this approach, being designed from the start to run in parallel on a range of different architectures, including commodity clusters. An alternative approach, and the one we adopt in this paper, is to identify the computationally demanding bottlenecks of the calculation and to adapt just those to run over multiple cores. Parallelization of the bottlenecks of the calculation can be an effective strategy; for example, Kleinschmidt et al.[32] parallelized the matrix-vector multiplication which was the most time-consuming part of their direct multireference configuration interaction (MRCI) code. They achieved good scaling over commodity multicore processors but noted that the memory bandwidth to the processor became a significant bottleneck, particularly as the number of cores per processor increased.

Theory

DFT has two main bottlenecks when applied to 30−50 atom systems: the evaluation of the Coulomb matrix

$$J_{\alpha\beta} = \sum_{\gamma\delta} \gamma_{\gamma\delta}(\alpha\beta|\gamma\delta) \tag{1}$$

and the numerical quadrature used to evaluate the exchange-correlation contribution to the Fock matrix

$$V^{\mathrm{xc}}_{\alpha\beta} = \int \mathrm{d}\mathbf{r}\, v^{\mathrm{xc}}(\mathbf{r})\chi_\alpha(\mathbf{r})\chi_\beta(\mathbf{r}) \approx \sum_\lambda w_\lambda v^{\mathrm{xc}}_\lambda \chi_{\alpha\lambda}\chi_{\beta\lambda} \tag{2}$$

Here, and throughout, we use the notation $(\cdot|\cdot)$ to denote a two-electron repulsion integral (ERI), so for example

$$(\alpha\beta|\gamma\delta) = \int \mathrm{d}\mathbf{r}_1 \int \mathrm{d}\mathbf{r}_2 \frac{\chi_\alpha(\mathbf{r}_1)\chi_\beta(\mathbf{r}_1)\chi_\gamma(\mathbf{r}_2)\chi_\delta(\mathbf{r}_2)}{r_{12}} \tag{3}$$

The numerical quadrature runs over the points $\mathbf{r}_\lambda$, with weights w_λ, and $v^{\mathrm{xc}}_\lambda = v^{\mathbf{xc}}(\mathbf{r}_\lambda)$, and $\chi_{\alpha\lambda} = \chi_\alpha(\mathbf{r}_\lambda)$.

It is straightforward to parallelize numerical quadrature by distributing batches of integration points between processing cores. The Coulomb term is more problematic. Direct calculation of the Coulomb contribution requires four-index ERIs $(\alpha\beta|\gamma\delta)$. Density fitting[33,34] can be used to treat the Coulomb contribution using just two- and three-index integrals. The conventional Kohn−Sham density

$$\rho(\mathbf{r}) = \sum_{\alpha\beta} \gamma_{\alpha\beta}\chi_\alpha(\mathbf{r})\chi_\beta(\mathbf{r}) \tag{4}$$

is approximated using an auxiliary basis of fitting functions, Ξ_A

$$\tilde{\rho}(\mathbf{r}) = \sum_B d_B \Xi_B(\mathbf{r}) \tag{5}$$

Here d_B are the density-fitting coefficients, calculated by minimizing the Coulomb self-energy of the fitting residual[34]

$$\Delta = \frac{1}{2}(\rho - \tilde{\rho}|\rho - \tilde{\rho}) \tag{6}$$

The Coulomb Fock contribution becomes

$$J_{\alpha\beta} = (\alpha\beta|\rho) \approx (\alpha\beta|\tilde{\rho}) = \sum_B d_B(\alpha\beta|B) \tag{7}$$

However, this requires the evaluation of a large number of three-index integrals. To achieve the very fine-grained parallelism required for the port to the ClearSpeed accelerator, we introduced a quadrature-based Coulomb method.[13] The details can be found elsewhere,[13] but for convenience the key points of the so-called grid-based density fitting Poisson method (GDFP) for the Coulomb problem are as follows:

1) The density is expanded in a basis that contains a small number of standard, atom-centered basis functions Ξ_A, and many Poisson functions of the form $\nabla^2\Xi_B$.[35,36]

2) Three-index Coulomb integrals involving Poisson functions become short-ranged overlap-like integrals, which are evaluated by quadrature.

3) The remaining, small number of conventional Coulomb integrals are evaluated analytically.

By using quadrature, both the grid-based Coulomb method and the treatment of exchange-correlation fall into the class of embarrassingly parallel problems, and it is expected that a parallel implementation for multisocket/multicore commodity processors will result in perfect linear scaling for these steps.

Bottlenecks

There are three bottlenecks in GDFP Kohn−Sham theory for our target system sizes. These, in order of decreasing time, are

1) The evaluation of the conventional Kohn−Sham density at each grid point, ρ_λ.

2) The quadrature-based evaluation of the exchange-correlation contributions to the Fock matrix.

3) The quadrature-based GDFP evaluation of the Coulomb contribution.

The first bottleneck is the calculation of the density on the grid, ρ_λ, which is evaluated during each Kohn−Sham iteration based on density matrix, $\gamma_{\alpha\beta}$, and the representation of each orbital at each grid point, $\chi_{\alpha\lambda}$

$$\rho_\lambda = \sum_{\alpha\beta} \gamma_{\alpha\beta}\chi_{\alpha\lambda}\chi_{\beta\lambda} \tag{8}$$

The density on the grid is used to evaluate the exchange-correlation potential on each grid point, v_λ^{xc}, which is used to calculate the exchange-correlation contributions to the Fock matrix

$$V_{\alpha\beta}^{\mathrm{xc}} = \sum_\lambda w_\lambda v_\lambda^{\mathrm{xc}} \chi_{\alpha\lambda}\chi_{\beta\lambda} \tag{9}$$

(Similar terms arise for gradient-corrected functionals).

The density on the grid is also used in the GDFP-based Coulomb numerical quadrature. The quadrature is used to evaluate the contribution to the Coulomb Fock matrix that arises from the Poisson part of the fitted density

$$J_{\alpha\beta} \leftarrow \sum_\lambda w_\lambda \chi_{\alpha\lambda}\chi_{\beta\lambda}\left[\sum_B d_B \Xi_{B\lambda}\right] \tag{10}$$

Program Design

The three bottlenecks in the calculation each involve evaluation of values at all of the large number of independent quadrature grid points. It is therefore natural to design the program so that the outer loop is over grid points, and thus parallelization is achieved by using vectorization to process multiple grid points per processor cycle and by dividing batches of grid points between processor cores. Vectorization of double-precision operations on x86 and x86-64 processors such as Intel Xeon or AMD Opteron is achieved by using version 2 of the SSE instruction set (SSE2),[20] which can be generated automatically by a good compiler, or which are available directly to the programmer in the C and C++ languages via intrinsics (in the emmintrin.h header file).

Dividing batches of grid points between processor cores can be achieved using OpenMP. OpenMP provides a set of compiler pragmas in C, C++, or Fortran, which the programmer can use to delineate parallel and serial parts of the code. OpenMP was first published in 1997, at which time commodity multisocket or multicore platforms were not available. Ports of code to OpenMP were thus limited to specialist hardware, such as the Silicon Graphics Origin 2000, on which a parallel port of part of the Gaussian 98 quantum chemical program using OpenMP was demonstrated.[37]

OpenMP is easy to use, but efficient parallelization requires that the programmer ensures that data synchronization between the serial and parallel parts of the code occurs infrequently and that each parallel batch computes as much as possible using thread-local data. Thread-local data are data which exist in memory that is owned and accessed by only a single thread of execution, and so no synchronization is needed between threads when reading or writing that data. This requires tight control of memory read/write operations, and of which data are thread-local and which are global. This is best achieved using scoping, where variables that represent data that are thread-local exist only within the parallel regions of the code.

C++ is a computer language that provides scoped variable declarations (e.g., a variable can exist only within a loop, and is not accessible outside the loop). Because of this, and because C++ provides direct access to SSE2, it was decided to rewrite the computationally expensive kernels of the Molpro DFT implementation in C++.

Vectorization Using SSE2. Vectorization of the DFT calculation was achieved by batching together grid points into vectors. A C++ class, called MultiDouble, was created to represent a vector of doubles within the code. An ancillary class, called MultiPoint, was then created to represent a vector of grid points. SSE2 intrinsics were then added to the member functions of MultiDouble to allow hardware vectorization to be used where it was available. The SSE2 code was hidden behind conditional statements, so the code is portable to platforms where SSE2 is not available.

Parallelization Using OpenMP. Each of the three bottlenecks is written as a loop over MultiDouble batches of grid points. All application-global read-only data are accessed via read-only pointers, thereby ensuring that synchronization between threads is not necessary to access that data. Each thread, when entering a parallel region, creates a local workspace in which to assemble the intermediates during the calculation. This local workspace is accessed via variables local to the scope of the parallel region and is cleared at the end of each parallel section. Synchronization between threads occurs only when thread-local contributions to the exchange-correlation and Coulomb parts of the Fock matrix are summed.

Screening Orbitals and Caching Data. All three bottlenecks of the calculation involve loops over all pairs of atomic orbitals and over each grid point. For large molecules, screening small contributions of orbitals on remote grid points reduces the scaling of computational work from cubic to linear with molecular size.

The representation of each orbital at each grid point, and the decision of whether or not an orbital should be screened, is a constant throughout the DFT calculation and does not need to be recalculated at each Kohn−Sham iteration. However, the memory required to store the representation of the orbitals on the grid scales with the number of orbitals times the number of grid points and can take several gigabytes for 30−50 atom molecules and commonly used basis sets. This memory requirement is naturally reduced by the application of screening, as only the non-negligible orbitals at each grid point need to be saved. The memory requirement then scales linearly with molecule size for sufficiently large cases.

Screening has been implemented on the basis of MultiDouble batches of grid points. Each MultiDouble batch is assigned a unique identification number, and an array of MultiDouble objects is then used to store the representation of only the non-negligible orbitals for this batch of grid points. This is then cached using a dictionary container, indexed using the unique identification number. A dictionary cache is used to allow the program to gracefully cope with cases where there is insufficient memory to store all of the orbitals on the grid. In these cases, as many MultiDouble batches that can fit into the cache are saved, while the remainder are recomputed for each Kohn−Sham iteration as needed.

Results

Benchmarking System and Platforms. The OpenMP DFT code was benchmarked by calculating the DFT B-LYP single point energy of an analogue of the neuraminidase inhibitor, DANA (Figure 1). This molecule is typical of those proposed during rational drug design, and it consists of 34 atoms, of which 17 are hydrogen.

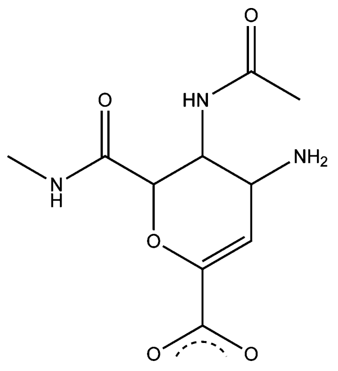

Figure 1. DANA, the test molecule used to benchmark the OpenMP DFT code.

Table 1. Four Multisocket/Multicore Platforms Used To Benchmark the OpenMP DFT Code

vendor	model	speed/ GHz	platform	memory/ GB	cores	peak[a]
AMD	Opteron 2218	2.59	dual−dual	8	4	20.7
Intel	Xeon E5472	2.80	dual-quad	8	8	44.8
AMD	Opteron 8220	2.80	oct-dual	32	16	89.6
Intel	Xeon X7350	2.93	quad−quad	32	16	93.8

[a] Peak double-precision performance in GFLOPS.

The DFT energy was calculated using three different basis sets, 6-31G*, cc-pVDZ, and aug-cc-pVDZ, to investigate how well the code scaled as the computational cost for each grid point increased. Density fitting was performed using a fitting set optimized for the cc-pVTZ atomic orbital basis.[38] For a carbon atom, the set consists of 1s1p1d standard Gaussian functions and 10s7p5d2f1g Poisson functions.

The code was benchmarked on four different platforms (see Table 1), chosen to represent a balance between AMD and Intel processors, and spread the range from two-socket to eight-socket platforms and dual-core to quad-core processors. To ensure that comparisons between the platforms were not biased by different compiler or OpenMP implementations, the C++ DFT code was compiled on all platforms using GCC 4.2,[39] with the libgomp[40] library provided with GCC.

Selecting a Vector Size. First, the sensitivity of performance to the size of the MultiDouble vector was investigated. While SSE2 hardware vectors are currently limited to just two doubles (as 128-bit registers are used), the MultiDouble class provides a software vector, the size of which can be controlled using a compile-time constant. Using a large vector is likely to improve the speed of the code, as it should allow for efficient pipelining. However, large vectors require more memory, which can have a negative impact on cache management and performance. To investigate this, the MultiDouble vector size was varied in powers of two from 8 to 128 doubles, and the impact on the speed of each of the three bottlenecks was measured (Figure 2).

While the exchange-correlation and numerical Coulomb bottlenecks benefit from larger vector sizes, the density bottleneck performs best using a vector size of 16 or 32. It is also clear that while the performance of the three bottlenecks is roughly equal using both the 6-31G* and cc-pVDZ basis sets, the performance of the exchange-correlation

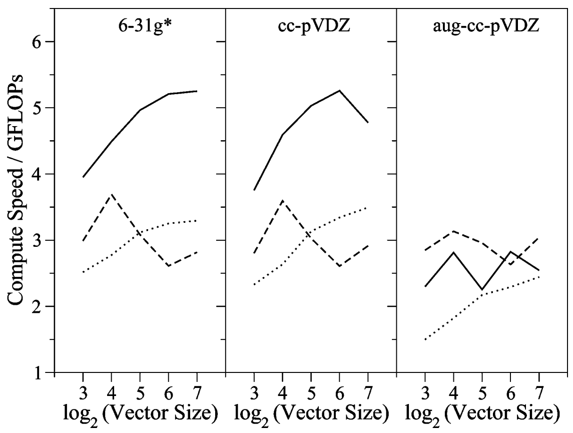

Figure 2. Average speed of calculation for the three main bottlenecks from each Kohn−Sham iteration for different MultiDouble vector sizes, for three different basis sets. Calculated using four threads on a dual−dual Opteron 2218 (solid: exchange-correlation contributions, dashed: density, dotted: numerical Coulomb).

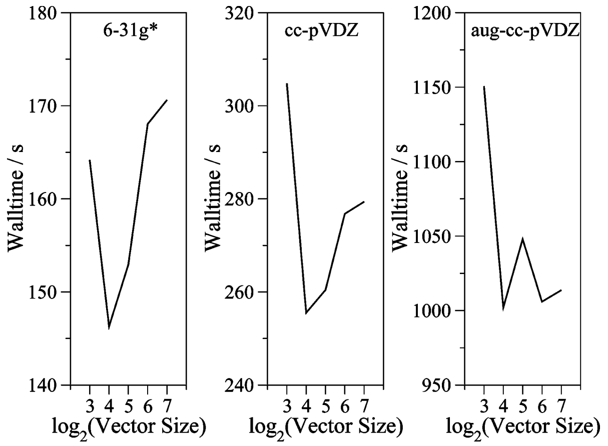

Figure 3. Total run time (wallclock) for different MultiDouble vector sizes, for three different basis sets. Calculated using four threads on a dual−dual Opteron 2218.

and density code is significantly reduced using aug-cc-pVDZ. This suggests that the increased memory-transfer requirements of this larger basis set may be stalling the calculation.

In addition to looking at the speed of the bottlenecks, the total run time of the calculation was also investigated as a function of vector size (Figure 3). From this it is clear that the preference of the density calculation for smaller vectors out-competes the preference for the other bottlenecks for larger vectors, and so the best performance is obtained using a vector size of 16 or 32. This observation was checked and found to be true on all of the other benchmark platforms, and so a vector size of 16 was chosen for all further tests.

Screening and Caching Orbitals on the Grid. The performance gains of orbital caching and screening were assessed. First, the total run time of the DFT calculation for each of the three basis sets was measured on the dual−dual Opteron 2218, both with the cache enabled and with the cache disabled. Using a dictionary-based cache significantly improves the run time, as shown in Table 2. The run time is reduced by 40−50% for the 6-31G* and cc-pVDZ calculations, but there is a comparatively smaller reduction for aug-cc-pVDZ. This is because the time required to perform each iteration (63 s) is much larger than the time required to build the orbitals on the grid (7 s). In addition, while look-up of the orbitals from the cache remains fast (taking just 0.1 s per iteration), the actual transport of the data for the orbitals on the grid from main memory appears to be hitting the bandwidth limit. This is evidenced by a reduction in computational efficiency from 3.3 GFLOPs to 2.8 GLOPs for the density calculation when the cache is used, compared to when the cache is disabled. Despite this, the use of the cache is still beneficial. Building the orbitals on the grid takes between 5−7 s, and, without the cache, this must be repeated for each Kohn−Sham iteration.

Table 2. Required Size of the Orbital Cache in Megabytes for Each of the Three Basis Sets, Together with the Run Time (Wallclock), in s, of the DFT Calculation with Both the Cache Enabled and Cache Disabled[a]

basis set	cache size	cache enabled	cache disabled
6-31G*	1819	146	296
cc-pVDZ	2128	256	409
aug-cc-pVDZ	3060	1002	1066

[a] Calculated using four threads on a dual−dual Opteron 2218.

Table 3. Time Required To Build the Orbital Cache, in s, Compared to the Total Run Time (Wallclock), in s, for the DFT Calculation, for Three Basis Sets[a]

basis set	threshold	% screened	build	run
6-31G*	10^{-15}	49	5.2	146
	0	9	5.9	306
cc-pVDZ	10^{-15}	47	5.5	256
	0	9	6.2	407
aug-cc-pVDZ	10^{-15}	43	7.3	1066
	0	8	7.6	1722

[a] Calculated using four threads on a dual−dual Opteron 2218. Times with screening (threshold = 10^{-15}) and without screening (threshold = 0) are shown, together with the percentage of orbital/grid batches that are less than or equal to the screening threshold.

The cost of evaluating orbitals on the grid can be reduced through a per-orbital/per-grid point screening algorithm. At the time of building the orbitals, a screening test is performed which compares the absolute value of each orbital at each grid point with a screening threshold (10^{-15}). The screening threshold was chosen to ensure that there was no detectable numerical difference between the screened and unscreened energies, but obviously greater savings could be made at the expense of some numerical precision. If the density is less than the screening threshold for each grid point for a MultiDouble batch (16 grid points), then this orbital is discarded for this batch of grid points. This slightly reduces the cost of building the orbitals on the grid, but, more importantly, this test absolutely removes all insignificant orbitals at each batch of grid points. As the subsequent parts of the calculation scale with the square of the number of significant orbitals times the number of grid points, the cost of building all orbitals at all grid points is more than recovered by the savings in the later stages. This is demonstrated in Table 3, which shows that the total time paid to screen and build the orbitals on the grid is significantly less than the time saved when performing screening.

Table 4. Average Speed, in Gigaflops, of the Three Main Bottlenecks over Each Kohn−Sham Iteration for Three Different Basis Sets, Both with Enabled and Disabled Manual SSE2 Code[a]

bottleneck	SSE enabled	6-31G*	cc-pVDZ	aug-cc-pVDZ
exchange-	yes	13.8	14.0	11.0
correlation	no	13.9	14.0	11.2
density	yes	11.7	10.9	9.6
	no	8.6	8.3	7.7
numerical	yes	8.7	9.6	8.3
Coulomb	no	8.3	9.0	7.9

[a] Calculations performed using eight threads of a dual-quad Xeon E5472.

Table 5. Total Run Time (Wallclock), in s, for the DFT Calculation Using Three Different Basis Sets, Both with Enabled and Disabled Manual SSE2 Code[a]

basis set	SSE enabled	1 core	8 cores
6-31G*	yes	308	59
	no	357	62
cc-pVDZ	yes	550	91
	no	639	101
aug-cc-pVDZ	yes	1606	290
	no	1854	314

[a] Calculations performed using either a single thread or eight threads of a dual-quad Xeon E5472.

Benchmarking Manual SSE. All of the benchmark processors support the use of SSE2 vector instructions. Manual SSE2 code can be easily enabled or disabled using a preprocessor flag within the implementation of the MultiDouble vector class. Manual SSE2 code is used extensively in the density calculation, while it is used sparingly when calculating the exchange-correlation contributions (for the reason, see the Appendix). The impact on enabling manual SSE2 on computational efficiency is shown in Table 4.

While manual SSE2 has little effect on the raw speed of the exchange-correlation and numerical Coulomb bottlenecks, the calculation of the density is the major bottleneck that dominates the total run time (approximately 40%). The extra 2−3 GFLOPs that is obtained by enabling manual SSE2 therefore has a large impact on the run time, particular when using a small number of threads (see Table 5). Because the use of manual SSE2 always reduced the run time, it was enabled for all further tests.

Scaling of the Bottlenecks. The previous benchmarks had validated the benefits of using a vector class, caching the representation of the orbitals on the grid, performing grid-based screening, and using manually coded SSE. The final part of the design to test was the use of OpenMP to parallelize the loops over MultiDouble batches of grid points. First, the scaling of the computational speed of each bottleneck was measured as a function of the number of threads. The speed was measured during each Kohn−Sham iteration of a full calculation and then averaged. The dependence of speed on the number of threads is shown in Figures 4, 5, and 6 for exchange-correlation contributions, calculation of the density, and numerical Coulomb respectively.

These results show that, as expected, the use of quadrature has resulted in linear scaling for each of the bottlenecks, even up to 16 threads. However, while this is seen for the 6-31G*

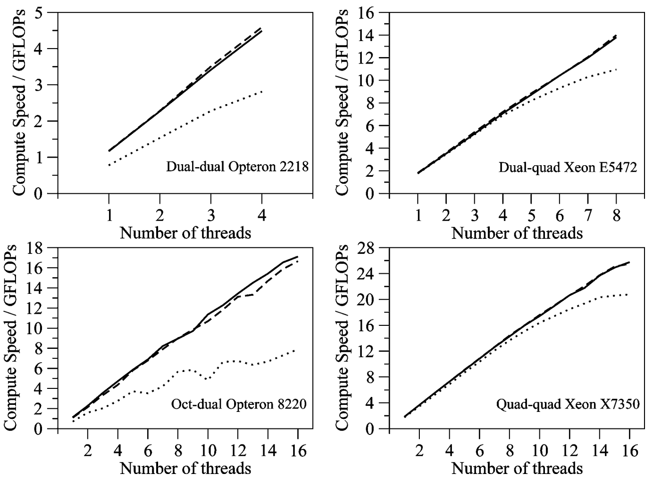

Figure 4. Average speed of the calculation of the exchange-correlation contributions to the Fock matrix, for three different basis sets (solid: 6-31G*, dashed: cc-pVDZ, dotted: aug-cc-pVDZ) as a function of the number of threads of execution on each of the four benchmark platforms.

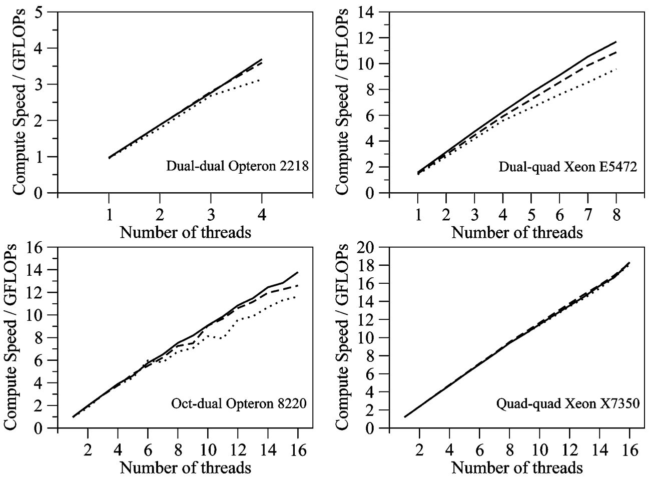

Figure 5. Average speed of the construction of the density, for three different basis sets (solid: 6-31G*, dashed: cc-pVDZ, dotted: aug-cc-pVDZ) as a function of the number of threads of execution on each of the four benchmark platforms.

and cc-pVDZ calculations, the limits of memory bandwidth mean that the scaling of the bottlenecks of the aug-cc-pVDZ calculations quickly tops out. Here, the quad−quad Xeon platform performs best, with linear scaling up to 8 threads for the exchange-correlation and numerical Coulomb bottlenecks and with linear scaling maintained up to all 16 threads when evaluating the density. This contrasts with the dual−dual and oct-dual Opteron platforms, which both suffer from substantially degraded performance of the exchange-correlation and numerical Coulomb bottlenecks (only 8% of the peak performance of the oct-dual Opteron is used for the 16 threads aug-cc-pVDZ exchange-correlation calculation, compared to 22% using 16 threads on the quad−quad Xeon). When making this comparison, it must be remembered that the Opteron processors used in these benchmarks are older designs than the Xeons, and it is likely that the latest AMD processors are likely to perform better in this regard.

In general these results show that the code performs well across each of the benchmark platforms. The evaluation of

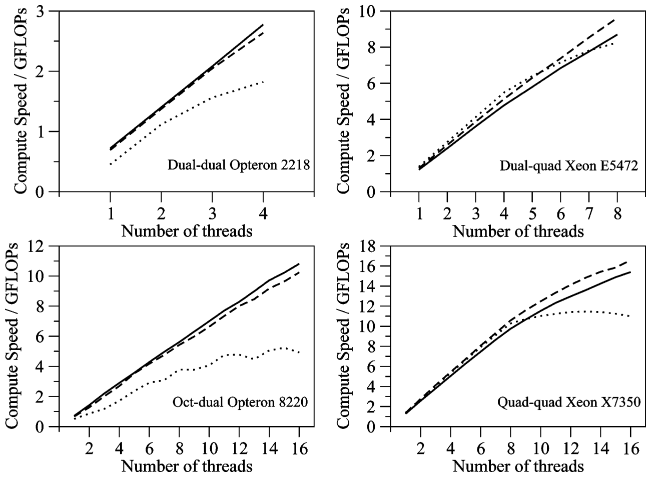

Figure 6. Average speed of the calculation of the Coulomb numerical quadrature, for three different basis sets (solid: 6-31G*, dashed: cc-pVDZ, dotted: aug-cc-pVDZ) as a function of the number of threads of execution on each of the four benchmark platforms.

the exchange-correlation components is particularly efficient on the Xeon platforms, while the dual-quad Xeon E5472 is consistently the most efficient platform tested. There does, however, seem to be some room for improvement in the efficiency of the code used to evaluate the Coulomb numeric quadrature contributions, with this region of the code having consistently the lowest performance of the three bottlenecks and having the poorest scaling and greatest performance degradation when using the large aug-cc-pVDZ basis set. One of the reasons for the comparatively poor performance of this bottleneck is that it is essentially just a product of all pairs of orbitals with all Poisson orbitals, at each grid point (see eq 10). This requires large amounts of data (all orbitals and Poisson orbitals), but very little actual computation (just three multiplies on four values loaded from four different regions of memory, in a naive implementation). It is thus not surprising that this memory-access-heavy but computationally light bottleneck has the poorest scaling and the biggest performance degradation with increasing basis size.

Discussion

Benchmarking of the OpenMP DFT implementation has shown that the code performs well up to the sixteen cores that were available for testing. Ultimately though, the user of the code is not really interested in the performance characteristics of parts of the code. Instead, their primary interest is likely to be the total run time of the calculation. This is shown in Table 6, where the total run time of the DFT calculations on each of the benchmark platforms is reported using both one thread and using the maximum number of threads available on each platform.

The original aim of this work was to reduce the run time to the extent that DFT calculations on commodity processors are sufficiently fast to allow practical applications in QM/MM free energy simulations. This has been achieved, with calculations using 6-31G* or cc-pVDZ basis sets reduced to less than 100 s for the eight and sixteen core machines. Indeed, a run time of just 46 s is required using the 6-31G* basis set on the quad−quad Xeon. QM/MM free energy calculations based on a recently developed Monte Carlo method[41] require of the order of 1000−2000 QM energy evaluations run in sequence to calculate a converged QM/MM relative free energy. This would take approximately 13−26 h using the quad−quad Xeon, using a 6-31G* basis set, or 1−2 days using a cc-pVDZ basis set on the more readily available dual-quad Xeon. This is well within the time scale necessary to make these calculations practical for drug-discovery type protein−ligand relative binding free energy simulations or for use in computational enzymology.

Table 6. Total Run Time (Wallclock), in s, for the DFT Calculation Using Three Different Basis Sets, on the Four Different Benchmark Platforms, As Calculated Using Just One Core, and Using All Available Cores[a]

platform	basis set	1 core	all cores
Opteron	6-31G*	493	146
2218	cc-pVDZ	894	256
dual−dual	aug-cc-pVDZ	3371	1002
Opteron	6-31G*	495	59
8220	cc-pVDZ	916	94
oct-dual	aug-cc-pVDZ	3364	369
Xeon	6-31G*	308	59
E5472	cc-pVDZ	550	91
dual-quad	aug-cc-pVDZ	1606	290
Xeon	6-31G*	345	46
X7350	cc-pVDZ	599	63
quad−quad	aug-cc-pVDZ	1751	187

[a] E.g. all eight cores of the dual-quad Xeon.

Table 7. Comparison of the Run Time (Wallclock), in s, for the Different Versions of Molpro on a Dual−Dual Opteron 2218 with Attached ClearSpeed CATs[a]

version	6-31G*	cc-pVDZ	aug-cc-pVDZ
Fortran	519	801	2070
OpenMP1	493	894	3371
OpenMP4	146	256	1002
ClearSpeed1	283	326	429
ClearSpeed12	39	46	86

[a] The original serial (Fortran) DFT implementation is compared to the OpenMP implementation on one core (OpenMP1), the OpenMP implementation on all four cores (OpenMP4), the ClearSpeed implementation using one ClearSpeed card (ClearSpeed1), and the ClearSpeed implementation using all twelve ClearSpeed cards (ClearSpeed12).

Finally, the performance of this OpenMP DFT code must be compared against the other implementations. Such a comparison is not entirely fair, due to differences in the implementation of screening, the use or not of numerical quadrature for the Coulomb calculation, a different reliance on and efficiency of the BLAS library, etc. To minimize the effect of these difference, this comparison was made using only the dual−dual Opteron (as it was this system that was connected to the ClearSpeed CATs). Table 7 shows the comparison of the total run time for the DFT calculations of the OpenMP code using one thread and four threads against the run time using an unmodified serial Molpro executable and against the ClearSpeed-enabled Molpro using one ClearSpeed dual socket CSX600 card ($2 \times 96 = 192$ cores) and using twelve ClearSpeed dual socket CSX600 cards (2304 cores).

Table 8. Total Run Time of the DFT Calculations, in s, Measured Using Different Numbers of ClearSpeed Cards Available via a ClearSpeed CATs Attached to a Dual−Dual Opteron 2218

number of cards	6-31G*	cc-pVDZ	aug-cc-pVDZ
1	283	326	429
2	150	173	231
3	107	122	177
4	89	102	149
5	49	59	130
6	46	55	122
7	43	51	90
8	41	49	88
9	40	47	87
10	39	46	86
11	39	47	86
12	39	46	86

This comparison raises two interesting points. First, the speed of the OpenMP implementation using one thread is competitive with standard Molpro, at least for the smaller basis sets. For the aug-cc-pVDZ basis set, standard Molpro is significantly faster than the OpenMP implementation, probably because in serial the numerical Coulomb calculation is not competitive with analytic integral evaluation. The second interesting point is that using four threads in the OpenMP code is quicker than using one ClearSpeed card, despite the higher peak double-precision performance (66 GFLOPs for one card versus 41 GFLOPs for the dual−dual Opteron). Again, this is only observed for the smaller basis sets. The ClearSpeed port is most efficient for larger basis sets, hence why the performance degradation is significantly lower when moving up to aug-cc-pVDZ compared to any of the other implementations.

The observation that the four-thread OpenMP implementation is faster than using a ClearSpeed card does raise an interesting question. For smaller systems it may be more effective to use a multiprocessor/multicore platform as opposed to using a single numerical accelerator. This can be quantified, again by using a slightly unfair comparison, by asking how many ClearSpeed accelerator cards are necessary to reduce the run time so that it is less than the minimum run time on each of the multiprocessor/multicore platforms. For example, the run time for sixteen threads for the cc-pVDZ calculation on the quad−quad Xeon was 63 s (Table 6). The run time using four ClearSpeed cards on the dual−dual Opteron was 102 s (see Table 8). Thus, in this case, a user would be better served by using the quad−quad Xeon. However, the run time using five ClearSpeed cards is 59 s, and so a user would be better served by using the accelerators. The number of ClearSpeed cards necessary to outperform each benchmark platform for each basis set has been calculated and is shown in Table 9. Again, it must be stressed that this is not a completely fair comparison, as the serial parts of the calculation are computed using an Opteron 2218, which was the slowest of the four processors used in this study. Despite this, it is clear that it is only the larger basis sets that benefit from the use of small numbers of accelerator cards and that large numbers of accelerator cards must be used in parallel in order to compete against a

Table 9. Number of ClearSpeed Cards That Must Be Used To Exceed the Speed of the OpenMP Implementation with the Maximum Number of Threads Available on Each of the Four Benchmark Platforms[a]

platform	6-31G*	cc-pVDZ	aug-cc-pVDZ
Opteron 2218	3	2	1
Opteron 8220	5	5	2
Xeon E5472	5	5	2
Xeon X7350	7	5	3

[a] The ClearSpeed calculations ran on a dual−dual Opteron 2218 attached to a ClearSpeed CATs, and this comparison does not account for the slower speed of the serial parts of the code on this system compared to the others.

multiprocessor/multicore system for drug-sized molecules with commonly used basis sets.

Given that three of the benchmark systems are faster using 8 or 16 cores than using four ClearSpeed cards (with 768 cores) for the 6-31G* and cc-pVDZ basis sets, questions should be asked as to whether or not more effort should be spent adapting algorithms for existing multiprocessor/multicore architectures. These questions are particularly pertinent now, as the standards for writing code for numerical accelerators and the underlying hardware model of these accelerators have yet to mature. On the other hand, multicore platforms are ubiquitous, and the languages and methods used to target them are well-established and performance-portable. Scientific applications are written over time scales of years to decades and contain large amounts of accumulated knowledge. Rewriting them so that they run efficiently in parallel represents a significant undertaking. Targeting multiprocessor/multicore machines leads to code that can be deployed efficiently across a range of platforms and which is easier to maintain and port over the coming years.

Acknowledgment. We thank the EPSRC for funding this work (EP/F010516/1) and for the Universities of Bristol, Cardiff, and Manchester for providing computer resources. In particular, the authors thank Ian Stewart, Christine Kitchen, and Simon Hood for providing access to, and for their help getting us set up on, the benchmark systems.

Appendix

The use of manual SSE2 in the MultiDouble vectors is only possible if the iteration over grid points occurs in an inner loop (as this MultiDouble is being used to vectorize over grid points). The calculation of the density is a sum over all pairs of orbitals, at all grid points (see eq 8), and so it is most efficient to use the iteration over grid points as the inner of the three loops. In contrast, the calculation of the exchange-correlation contributions is a sum over all grid points of products of all pairs of orbitals. In this case, it is most efficient to place the iteration over grid points as the outer loop, as otherwise poor cache performance is encountered during the iteration over all pairs of orbitals. Poor cache performance arises because the representation of orbitals in memory is arranged in contiguous blocks of 16 doubles (one for each grid point represented by the MultiDouble batch). This means that the distance in memory between adjacent orbitals, commonly called the stride, is 16 doubles. Loops over pairs of orbitals are therefore inefficient, as the stride between each iteration is larger than a cache line, and thus it is likely that different orbitals will exist on different cache lines. In the original implementation, this lead to poor performance. This was resolved by preceding the loop over pairs of orbitals with a loop over the 16 grid points within a MultiDouble batch, that copied the value of the orbital for that grid point into a temporary array of doubles. This packed the orbitals on a single grid point together into a single contiguous block in memory. In effect, this transposed the memory layout from using grid point as the inner index, to using orbital as the inner index. The cost of transposing memory was found to be negligible (formally scaling linearly with the number of orbitals), while the time saved was significant (as the cost of looping over pairs of orbitals scales quadratically with the number of orbitals). However, because the iteration over grid points was now the outer, not the inner, loop, manual SSE2 could no longer be used.

References

(1) Claeyssens, F.; Harvey, J. N.; Manby, F. R.; Mata, R. A.; Mulholland, A. J.; Ranaghan, K. E.; Schütz, M.; Thiel, S.; Thiel, W.; Werner, H. J. *Angew. Chem. Int. Ed.* **2006**, *45*, 6856–6859.

(2) Scuseria, G. *J. Phys. Chem. A* **1999**, *103*, 4782.

(3) Gogenea, V.; Suárez, D.; van der Vaart, A.; Jr., K. W. M. *Curr. Opin. Struct. Biol.* **2001**, *11*, 217.

(4) Dewar, M. J. S.; Zoebisch, E. G.; Healy, E. F.; Stewart, J. J. P. *J. Am. Chem. Soc.* **1985**, *107*, 3902.

(5) Stewart, J. J. P. *J. Comput.-Aided Mol. Des.* **1990**, *4*, 1.

(6) Frauenheim, T.; Seifert, G.; Elstner, M.; Hajnal, Z.; Jungnickel, G.; Porezag, D.; Suhai, S.; Scholz, R. *Phys. Stat. Sol. (B)* **2000**, *217*, 41.

(7) Kohn, W.; Sham, L. J. *Phys. Rev. A* **1965**, *140*, 1133.

(8) Kohn, W.; Becke, A. D.; Parr, R. G. *J. Phys. Chem.* **1996**, *100*, 12974.

(9) Moore, G. E. *Electronics* **1965**, *38*, 8.

(10) Intel announced an oct-core Xeon in February 2009 at the International Solid State Circuits Conference.

(11) Advance e620 Product Brief. http://www.clearspeed.com (accessed March 11, 2008).

(12) Anderson, A. G.; Goddard, W. A.; Schröder, P. *Comput. Phys. Commun.* **2007**, *177*, 298–306.

(13) Brown, P.; Woods, C. J.; McIntosh-Smith, S.; Manby, F. R. *J. Chem. Theory Comput.* **2008**, *4*, 1620–1626.

(14) Yasuda, K. *J. Comput. Chem.* **2008**, *29*, 334–342.

(15) Yasuda, K. *J. Chem. Theory Comput.* **2008**, *4*, 1230–1236.

(16) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049–2057.

(17) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222–231.

(18) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004–1015.

(19) Werner, H.-J.; Knowles, P. J.; Lindh, R.; Manby, F. R.; Schütz, M. et al. *MOLPRO, version 2008.1, a package of ab initio programs*; 2008. See http://www.molpro.net (accessed Feb 2009).

(20) Intel C++ Compiler for Linux - 9.X: Intrinsics Reference. http://software.intel.com/en-us/articles/intel-c-compiler-for-linux-9x-manuals (accessed May 28, 2009).

(21) Smith, L.; Kent, P. *Concurrency: Pract. Exper.* **2000**, *12*, 1121–1129.

(22) Shellman, S. D.; Lewis, J. P.; Glaesemann, K. R.; Sikorski, K.; Voth, G. A. *J. Comput. Phys.* **2003**, *188*, 1–15.

(23) Medvedev, D. M.; Goldfield, E. M.; Gray, S. K. *Comput. Phys. Commun.* **2005**, *166*, 94–108.

(24) Hutter, J.; Curioni, A. *Parallel Comput.* **2005**, *31*, 1–17.

(25) Almasi, G.; Bhanot, G.; Chen, D.; Eleftheriou, M.; Fitch, B.; Gara, A.; Germain, R.; Gunnels, J.; Gupta, M.; Heidelberg, P.; Pitman, M.; Rayshubskiy, A.; Sexton, J.; Suits, F.; Vranas, P.; Walkup, B.; Ward, C.; Zhestkov, Y.; Curioni, A.; Andreoni, W.; Archer, C.; Moreira, J.; Loft, R.; Tufo, H.; Voran, T.; Riley, K.; Cunha, J. C. *Euro-Par 2005 Parallel Process.* **2005**, *3648*, 560–570.

(26) Bottin, F.; Leroux, S.; Knyazev, A.; Zérah, G. *Comput. Mater. Sci.* **2008**, *42*, 329–336.

(27) Fan, P.-D.; Valiev, M.; Kowalski, K. *Chem. Phys. Lett.* **2008**, *458*, 205–209.

(28) Rayson, M.; Briddon, P. *Comput. Phys. Commun.* **2008**, *178*, 128–134.

(29) Baker, J.; Wolinski, K.; Malagoli, M.; Kinghorn, D.; Wolinski, P.; Magyarfalvi, G.; Saebo, S.; Janowski, T.; Pulay, P. *J. Comput. Chem.* **2009**, *30*, 317–335.

(30) Mostofi, A.; Haynes, P.; Skylaris, C.-K.; Payne, M. *Mol. Simul.* **2007**, *33*, 551–555.

(31) Skylaris, C.-K.; Haynes, P.; Mostofi, A.; Payne, M. *J. Phys.: Condens. Matter* **2008**, *20*, 064209.

(32) Kleinschmidt, M.; Marian, C.; Waletzke, M.; Grimme, S. *J. Chem. Phys.* **2009**, *130*, 044708.

(33) Baerends, E. J.; Ellis, D. E.; Ros, P. *Chem. Phys.* **1973**, *2*, 41.

(34) Dunlap, B. I.; Connolly, J. W. D.; Sabin, J. R. *J. Chem. Phys.* **1979**, *71*, 3396.

(35) Manby, F. R.; Knowles, P. J.; Lloyd, A. W. *J. Chem. Phys.* **2001**, *115*, 9144.

(36) Manby, F. R.; Knowles, P. J. *Phys. Rev. Lett.* **2001**, *87*, 163001.

(37) Sosa, C. P.; Scalmani, G.; Gomperts, R.; Frisch, M. J. *Parallel Comput.* **2000**, *26*, 843–856.

(38) Polly, R.; Werner, H.-J.; Manby, F. R.; Knowles, P. J. *Mol. Phys.* **2004**, *102*, 2311–2321.

(39) GNU Compiler Collection 4.2.4. http://gcc.gnu.org (accessed March 21, 2009).

(40) libgomp: The GNU implementation of OpenMP. http://gcc.gnu.org/onlinedocs/libgomp (accessed March 21, 2009).

(41) Woods, C. J.; Manby, F. R.; Mulholland, A. J. *J. Chem. Phys.* **2008**, *128*, 014109.

CT900138J