Porting ONETEP to Graphical Processing Unit-Based Coprocessors. 1. FFT Box Operations

Karl Wilkinson and Chris-Kriton Skylaris*

We present the first graphical processing unit (GPU) coprocessor-enabled version of the Order-N Electronic Total Energy Package (ONETEP) code for linear-scaling first principles quantum mechanical calculations on materials. This work focuses on porting to the GPU the parts of the code that involve atom-localized fast Fourier transform (FFT) operations. These are among the most computationally intensive parts of the code and are used in core algorithms such as the calculation of the charge density, the local potential integrals, the kinetic energy integrals, and the nonorthogonal generalized Wannier function gradient. We have found that direct porting of the isolated FFT operations did not provide any benefit. Instead, it was necessary to tailor the port to each of the aforementioned algorithms to optimize data transfer to and from the GPU. A detailed discussion of the methods used and tests of the resulting performance are presented, which show that individual steps in the relevant algorithms are accelerated by a significant amount. However, the transfer of data between the GPU and host machine is a significant bottleneck in the reported version of the code. In addition, an initial investigation into a dynamic precision scheme for the ONETEP energy calculation has been performed to take advantage of the enhanced single precision capabilities of GPUs. The methods used here result in no disruption to the existing code base. Furthermore, as the developments reported here concern the core algorithms, they will benefit the full range of ONETEP functionality. Our use of a directive-based programming model ensures portability to other forms of coprocessors and will allow this work to form the basis of future developments to the code designed to support emerging high-performance computing platforms. © 2013 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23410

Introduction

Current hardware trends show machines with heterogeneous architectures are emerging as the computing platform of choice within high-performance computing (HPC). Heterogeneous machines feature multiple distinct types of processors, typically standard central processing units (CPUs) and coprocessor boards connected through the peripheral component interconnect (PCI) express bus. Heterogeneous architectures are particularly appealing as a typical software package will contain a wide range of algorithms and they offer the opportunity to execute a given algorithm on the most suitable processing unit. For example, serial algorithms that can take advantage of out-of-order operations are well-suited to CPUs whereas the parallel, data intensive operations inherent to a fast Fourier transform (FFT) are well-suited to a coprocessor board with a highly parallel architecture. The coprocessor boards most commonly used in heterogeneous machines are based on graphical processing units (GPUs). As the name suggests, GPUs process data to produce the graphical output shown on a computer's display. As a result of the independence of each pixel on a display, GPUs, a type of stream processor,[1] are highly parallel by design: often containing large numbers ($O(1000)$) of relatively simple cores per socket. The development of GPU technology has progressed very quickly, initially driven by the insatiable demands of the computer gaming industry. However, with the emergence of general purpose computing on GPUs[2] (GPGPU), the utilization of GPUs within computational tasks has increased dramatically. Indeed, the recently deployed Titan supercomputer at the Oak Ridge National Laboratory in the contains 18,688 AMD Opteron CPUs and the same number of NVIDIA Kepler K20 GPUs, giving a peak performance of over 20 petaflops and currently occupying the second position in the Top 500 list of the most powerful supercomputers in the world.

GPU accelerator boards are typically connected to a "host" machine through the PCI express bus, a single board usually contains a single-GPU processor, and several gigabytes of global memory. NVIDIA GPUs are comprised of a number of streaming multiprocessors, each divided into a collection of relatively simple compute unified device architecture (CUDA) cores. Each CUDA core has access to memory shared with other cores within the same multiprocessor as well as the global GPU memory. Further technical details of the parallel structure of GPU-based coprocessors and the manner in which code is executed on them are available elsewhere within the computational chemistry literature.[3,4] In addition, a huge

K. Wilkinson, C.-K. Skylaris
School of Chemistry, University of Southampton, Highfield, Southampton, SO17 1BJ, United Kingdom
E-mail: c.skylaris@soton.ac.uk

Contract grant sponsor: Engineering and Physical Sciences Research Council (K.W.); Contract grant number: EP/I006613/1; Contract grant sponsor: Royal Society for a University Research Fellowship (C.K.S.)

number of resources are available online.[2,5] Further, as one of the strengths of the programming methods used in this article is that these details are transparent to the developer, a detailed discussion of these technicalities is not presented here.

GPGPU reached its current level of popularity within the HPC field due to the development of programming models specifically targeted at GPU architectures. The most well-known example of such a language is the C-based CUDA[5] language developed by NVIDIA. CUDA Fortran[6] is the Fortran counterpart to standard CUDA, a significant difference between CUDA and CUDA Fortran is that CUDA Fortran may interact with the GPU indirectly through language constructs, whereas CUDA only interacts directly through application programming interface calls. Also available is the open computing language[7] (OpenCL) from the Khronos Group which allows a single-code base to be compiled for execution on multiple architectures while, theoretically at least, maintaining performance. Before the emergence of these models, developers of nongraphics software hoping to utilize GPUs were forced to use graphics languages such as OpenGL[8] and interface them with their codes.[9,10] CUDA and OpenCL represent the most commonly used programming models for the development of GPU code. However, for many researchers porting existing packages to these models represents a significant investment of time and can pose a considerable learning curve to developers unfamiliar with the models. A further issue is the maintenance of a single-code base when a given routine has versions written in multiple languages. Alternative, less programming intensive approaches are available in the form of PGI Accelerate[11] and OpenACC.[12] These methods allow developers to write code using standard Fortran or C while utilizing a pragma paradigm similar to OpenMP[13] to label areas of the code that may be executed in parallel. A particular advantage arising from the use of these models is the compiler's ability to generate a thread scheduling scheme, removing the onus from the developer. Despite the ease with which suitably parallel code may be adapted for execution on GPUs with these methods, it is typically necessary to make structural changes to the code to achieve optimum performance. In particular, care must be taken when considering the communications expense resulting from the transfer of data across the PCI express bus. Another significant advantage to these models is the recently announced support for other coprocessor technologies such as GPUs from AMD and the Xeon Phi from Intel.[12] The development of GPU optimized code is further enhanced by the availability of highly optimized mathematical libraries. For example, CUBLAS[14] and CUFFT[15] for linear algebra and FFTs, respectively.

The high computational requirements of the computational chemistry field mean that accelerated computing is of great relevance. A number of computational chemistry software packages are available that may be executed on GPUs, these are discussed in recent reviews.[16–18] Classical molecular dynamics (MD) codes are highly compute intensive, there is a constant demand for improved performance, be it to enable calculations on larger systems, for longer periods of time or to allow greater sampling of conformational space. Many of the algorithms used within classical MD are well-suited to GPU acceleration. A number of MD packages such as NAMD[19,20] and AMBER[21,22] have ported these algorithms to GPUs. Other packages such as HOOMD[23] and ACEMD[24,25] have been designed from the ground up to use GPUs. Quantum chemistry packages are also computationally very demanding and a number of codes such as GAMESS-UK,[4] GAMESS-US,[26,27] and Terachem[3,28,29] contain sections of code that may be executed on GPUs. Within these codes and others Hartree–Fock, density functional theory (DFT) and even post Hartree–Fock methods such as MP2[30] and coupled cluster[31–33] have been adapted for execution on a GPU. The initial focus of these codes has typically been the evaluation of the electron repulsion integrals (ERIs). However, the number of ERIs within Gaussian basis set calculations scales formally to the fourth power with chemical system size, and may reach the order of many billions of integrals. However, when integral screening is applied this number, and the scaling, may be reduced to lower than the fourth power. This means that even with the performance gains reported for these codes the cost of this analysis becomes prohibitive, especially when a more accurate basis set incorporating diffuse and polarization functions are to be used. Another commonly used basis set is the plane-wave basis which provides high accuracy and the ability to systematically improve accuracy. However, it is very demanding to port to a coprocessor as it is uniform in space and has high-memory demands. An example of a plane-wave code that has been ported to GPUs is the Vienna *ab initio* simulation package.[34] Another example of a high-resolution basis set code that has been ported to GPUs is the BigDFT code, which uses a wavelet basis set.[35]

In this article, we present and discuss the first GPU-enabled version of the Order-N Electronic Total Energy Package (ONETEP) software package.[36] ONETEP is a quantum chemistry package capable of simulating large numbers of atoms[37] based on linear-scaling electronic structure theory.[38,39]. A unique feature of this code is that it can achieve linear-scaling while maintaining the high-basis set accuracy of conventional plane-wave codes. In this initial implementation, we focus on porting the so-called FFT box operations which take up a significant fraction of the calculation time while maintaining the current architecture of the code relating to communications and load balancing. In typical GPGPU terminology, a section of code that is executed on a GPU is referred to as a "kernel." However, this terminology conflicts with the term "density kernel," which is a critical feature of the methods used within ONETEP. To prevent confusion, these terms will be referred to as "GPU kernel" and "density kernel" accordingly.

The structure of this article is as follows; First, the ONETEP linear-scaling quantum chemistry software package is briefly discussed in the section titled, ONETEP. Technical Details, discusses the hardware and test systems used within this study. Section, Profiling, illustrates the bottlenecks within the ONETEP code and Methods and Results presents the theory, algorithms, and technical details relating our developments and testing of our GPU algorithms. This section also contains detailed results showing the performance of the code on a variety of GPU-accelerated platforms. These are followed by a discussion of the scaling of the ported code with both the size of the chemical system and the number of compute nodes. We finish with some conclusions.

ONETEP

ONETEP[36] is a linear-scaling quantum chemistry software package for DFT calculations. The linear-scaling computational cost with respect to the number of atoms within ONETEP is achieved through the exploitation of the "nearsightedness of electronic matter" principle.[40,41] The theoretical details of the ONETEP methodology are discussed in detail elsewhere [36] and are only summarized here: The ONETEP program is based on a reformulation of DFT in terms of the one-particle density matrix, $\rho(\mathbf{r},\mathbf{r}')$,

$$\rho(\mathbf{r},\mathbf{r}')=\sum_{i=1}^{N} f_i\psi_i(\mathbf{r})\psi_i(\mathbf{r}'), \tag{1}$$

where N is the total number of Kohn–Sham molecular orbitals $\{\phi_i(\mathbf{r})\}_{i=1}^{N}$ within the chosen basis and f_i are their occupancies. The one particle density matrix is the basis of many linear-scaling DFT approaches[38] where the memory and processing requirements increase linearly with N. This is achieved by taking advantage of the exponential decay of the density matrix in systems with a band gap.

In ONETEP, the density matrix is expressed in the following form:

$$\rho(\mathbf{r},\mathbf{r}')=\sum_{\alpha}\sum_{\beta}\phi_\alpha(\mathbf{r})\mathrm{K}^{\alpha\beta}\phi_\beta(\mathbf{r}'), \tag{2}$$

where the "density kernel" $\mathbf{K}$ is the density matrix expressed in the duals of the set of nonorthogonal generalized Wannier functions (NGWFs)[42] $\{\phi_\alpha(\mathbf{r})\}$. The NGWFs are constrained to be strictly localized within spherical regions centered on atoms and their shape is optimized self-consistently by expressing them in a psinc basis set,[43] which is equivalent to a plane-wave basis set. As a result ONETEP is able to achieve linear-scaling computational cost while retaining the large basis set accuracy characteristics of plane-wave codes. The calculation of the electronic energy within ONETEP takes the form of two nested loops, the density kernel, $\mathbf{K}$, and NGWFs $\{\phi_\alpha(\mathbf{r})\}$ are optimized within the inner and outer loops, respectively. ONETEP also exhibits parallel scaling as the number of processors used to perform a calculation is increased using the message passing interface (MPI) paradigm. Again, the techniques used to achieve this are discussed in detail elsewhere.[44,45]

Data abstractions

ONETEP is the only computational chemistry software package that uses the psinc basis set.[46] Psinc functions are centered on the points of a regular real-space grid and are related to a plane-wave basis through Fourier transforms. The use of psinc functions enables the strict localization of the NGWFs, which is crucial to ONETEPs linear-scaling behavior. To perform operations involving NGWFs, the FFT box technique is used. An FFT box is a box of grid points centered on the atom associated with an NGWF and large enough to contain any overlapping NGWF localization spheres in their entirety. This representation permits the use of plane-wave methodology to perform momentum space operations with a computational cost that is independent of the size of the simulation cell.

Load balancing within ONETEP is achieved through the distribution of the workloads associated with atomic data (operations performed on NGWFs), the simulation cell (simulation cell FFTs) and sparse matrix data. The schemes used to achieve a balanced load are discussed in detail elsewhere.[44] Due to the fact that the different workloads are distributed in this manner, it is necessary to perform communication between compute nodes. To do this efficiently, it is necessary to use a memory efficient data abstraction, the PPD (parallelpiped) representation (bottom panel of Fig. 1). The size of the PPDs in a calculation is constant and is chosen such that the simulation cell is built from an integer number of PPDs.[44] An important point when considering the GPU code is that in the FFT box representation the data associated with a single PPD is noncontiguous in memory while in the PPD representation it is. This is illustrated using the psinc functions x, y, and z highlighted in the lower two panels of Figure 1.

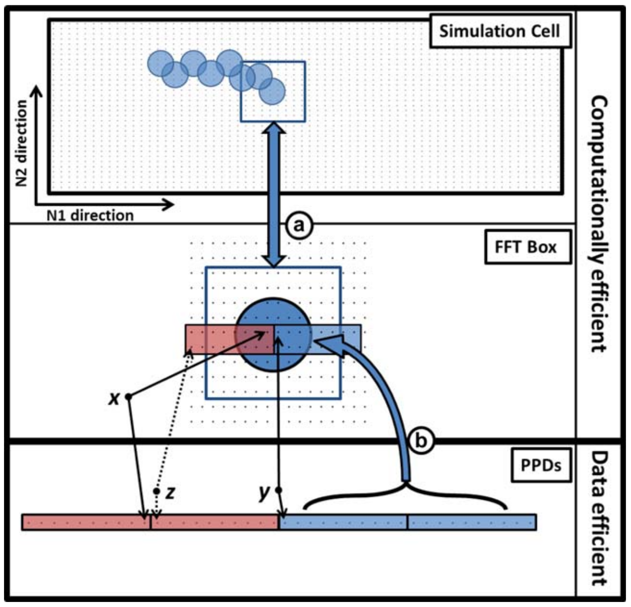

Figure 1. Data abstractions within ONETEP. a): represents the processes of extracting and depositing FFT boxes from the simulation cell, (b) represents the interconversion of data between the FFT box and PPD representations as required for efficient communication of data between compute nodes. In the simulation cell and FFT boxes, data points in the N1 direction (x and y) are contiguous in memory. In the PPD representation, this is not always the case (x and z). An FFT box is not forced to consist of an integer number of PPDs. The simulation cell and FFT box are three-dimensional (3D) arrays, whereas the PPD storage array is 1D. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Technical Details

Chemical systems

To illustrate the performance of the ported code, a series of chemical systems were selected. As shown in Figure 2, these structures are the "tennis ball" dimer with methane guest (181 atoms) and two cellulose fibril systems with 495 and 957 atoms, respectively. Data shown in Profiling are limited to the "tennis ball" dimer whilst Methods and Results contains calculations performed with all three systems.

Calculation details

Calculations performed on the "tennis ball" dimer system detailed in Profiling and on the cellulose systems in Methods and Results are limited to single iterations of the inner and outer loops of the ONETEP process. To maintain focus for this initial study, the calculations discussed will solely employ the Li–Nunes–Vanderbilt algorithm[47,48] method for the optimization of the density kernel.

Hardware specifications

A variety of machines were used to perform the calculations presented here,

- The GPU nodes of the Iridis 3 supercomputer (30 M2050 Tesla GPUs with 2.40-GHz quad-core Intel Xeon E5620 CPUs) at the University of Southampton,
- The Emerald GPU cluster (372 M2090 Tesla GPUs with 2.53-GHz hexa-core Intel Xeon E5649 CPUs) at Rutherford Appleton Laboratory in Oxford, and
- The NVIDIA PSG cluster (A wide variety of GPUs running on Sandy Bridge CPUs) at NVIDIA in Santa Clara. Calculations performed on the PSG cluster were limited to nodes containing Kepler K20 GPUs.

All these machines utilize the PCIe 2 standard. Communications between host and GPU were observed to perform data transfers at similar speeds. The profiling calculations performed in the Profiling section were performed on a single node of the Iridis3 machine, calculations on all machines were performed using multiple GPUs as detailed in the relevant subsections of Methods and Results.

In all cases, the use of error correcting-code (ECC) memory was enabled, as this is the standard setting on the Emerald and Iridis3 nodes. Although disabling ECC would result in enhanced performance, the timings presented here represent those that would be experienced by a typical user executing code on these machines. This choice also ensures the reliability of the results obtained when executing the code on the GPUs. A further GPU-specific modification to the code is the use of page locked memory. This is a region of host memory that is visible to the GPU as a part of its own memory space. To move data from the host to the device, it is first necessary to move it to this region. The consistent use of pinned memory means that arrays containing data to be moved between the host and GPU are specified to be within the page-locked region of memory on initialization.

The Emerald compute nodes contain multiple CPU sockets. Therefore, it is necessary to control the processor and GPU affinity of the MPI threads to ensure that QuickPath interconnects are not crossed and optimal performance is obtained. This is achieved through the use of a script that ensures threads are associated with CPU sockets and GPUs that are on the same PCI express bus

Timings

Timings in the following sections are presented for the specified number of MPI threads. Each MPI thread is associated with either a single-CPU core or a single GPU coupled with a CPU core. We made this choice as it is the current execution model used within ONETEP. A more reasonable comparison is that of a CPU processor (all available cores) against a GPU card, this comparison is presented in the subsection Multiple GPU performance. It should be highlighted that the developments presented here are an initial step toward a truly heterogeneous code that is efficiently executed on both CPUs and coprocessors simultaneously, thus utilizing hardware as efficiently as possible. However, as highlighted by this work, a number of developments are required before this goal can be achieved.

Profiling

Optimization of both the NGWFs and the density kernel will initially be limited to a single iteration of each to produce

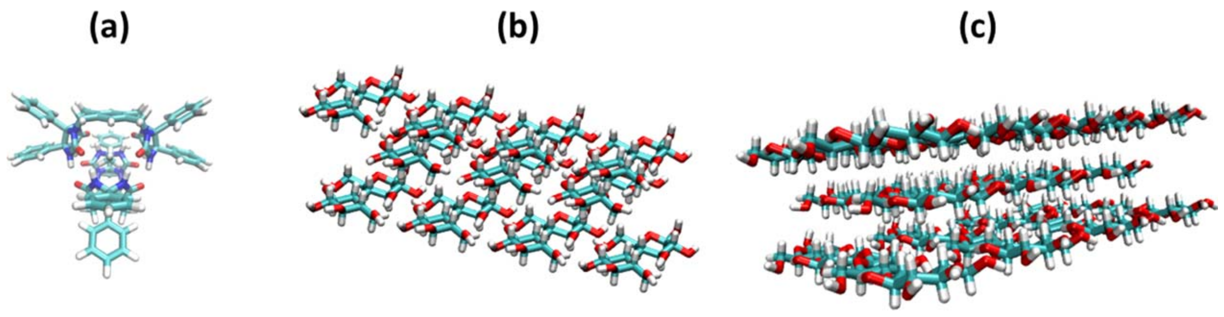

Figure 2. Chemical systems used in profiling and benchmarking calculations. a): 181 atom "tennis ball" dimer with methane guest, (b) 495 atom cellulose fibril, and c) 957 atom cellulose fibril. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

representative data as this is the most widely used energy optimization algorithm within the ONETEP code. To identify initial targets for GPU acceleration calculations with a psinc basis set, kinetic energy cutoff of 1200 eV was performed. This calculation used the "tennis ball" dimer system shown in Figure 2, and the timings are reported for a short calculation consisting of single iterations of the inner and outer loops of the ONETEP scheme. A breakdown of bottlenecks of the code that are suitable for GPU acceleration is shown in Table 1.

FFTs on FFT boxes and the processes involved in the movement and conversion of data between the abstractions introduced above feature heavily within each of the FFT box operations highlighted in Table 1. Together, these processes represent 59.0 and 14.1% of the entire calculation run time, respectively. Although not shown in Table 1, simulation cell FFTs (distinct from FFT box FFTs) take up to 1.5% of the calculations run time. As detailed above, this calculation was performed using a single iteration of the density kernel and NGWF optimization loops. As a result, the relative cost of the NGWF gradient is overstated in this example. The relative cost of the NGWF gradient is heavily dependent on the actual calculation but is typically around 5–10% of the total runtime, justifying its inclusion here.

Method and Results

Fast Fourier transforms

FFTs performed on FFT boxes[49] contribute significantly to the bottlenecks identified above, representing over half of the computational runtime in the calculation profiled in Profiling. Initial attempts at porting ONETEP to GPU-based accelerators simply involved supplementing the calls to the CPU FFT libraries with calls to a GPU-based equivalent: CUFFT.[15] However, porting the FFTs alone to the GPU resulted in a performance gain of only 1.2× over the CPU implementation (1869 s on GPU compared with 2180 s on the CPU). This result reflects the most common problem faced when porting code to GPUs: the expense of transferring data from the host to the GPU. To overcome this issue, it is necessary to improve the ratio of operations to data transfer. In the case of ONETEP, the ratio of data transfer to execution can be improved through porting additional code surrounding the FFTs for GPU execution. In the following sections, we describe how this is achieved.

Charge density

As shown in Table 1, a significant bottleneck (36% of the runtime for the profiled calculation) within ONETEP is the calculation of the charge density, $n(\mathbf{r})$. The charge density is given by the diagonal elements of the one-particle density matrix from eq. (2) using $n(\mathbf{r})=\rho(\mathbf{r},\mathbf{r})$. As such, the porting of this section of the code and the analysis of its performance will be discussed in detail. Many of the techniques used in the optimization of this part of the code are also used in other areas of the ported code and are discussed later. The charge density, $n(\mathbf{r})$, may be represented as

$$n(\mathbf{r})=\sum_{\alpha} n(\mathbf{r};\alpha)=\sum_{\alpha}\phi_{\alpha}(\mathbf{r})\sum_{\beta}K^{\alpha\beta}\phi_{\beta}(\mathbf{r}), \tag{3}$$

Therefore, the calculation of the charge density may be considered as the calculation of the series of $n(\mathbf{r};\alpha)$ components. These are the product of the FFT box for $\phi_{\alpha}(\mathbf{r})$and the FFT box of the sum of all $\phi_{\beta}(\mathbf{r})$'s with which $\phi_{\alpha}(\mathbf{r})$ overlaps, including itself. The following stages, as shown in Figure 3 are required for the calculation of an NGWFs contribution to the charge density:

1. Gather data

a. Extract the $\phi_{\alpha}(\mathbf{r})$ NGWF from the PPD representation to an FFT box.

b. Gather the sum of overlapping ϕ_{β} NGWFs $\sum_{\beta}K^{\alpha\beta}\phi_{\beta}(\mathbf{r})$ for NGWF $\phi_{\alpha}(\mathbf{r})$ to an FFT box.

2. Populate the coarse grid FFT box with $\phi_{\alpha}(\mathbf{r})$ and $\sum_{\beta}K^{\alpha\beta}\phi_{\beta}(\mathbf{r})$.
3. Forward FFT of the $\phi_{\alpha}(\mathbf{r})$ and $\sum_{\beta}K^{\alpha\beta}\phi_{\beta}(\mathbf{r})$ coarse grid FFT boxes to obtain their Fourier transforms $\tilde{\phi}_{\alpha}(\mathbf{g})$ and $\sum_{\beta}K^{\alpha\beta}\tilde{\phi}_{\beta}(\mathbf{g})$.
4. Up sample, the $\tilde{\phi}_{\alpha}(\mathbf{g})$ and $\sum_{\beta}K^{\alpha\beta}\tilde{\phi}_{\beta}(\mathbf{g})$ coarse grid FFT boxes to a fine grid FFT box using zero padding.
5. Inverse FFT of the $\tilde{\phi}_{\alpha}(\mathbf{g})$ and $\sum_{\beta}K^{\alpha\beta}\tilde{\phi}_{\beta}(\mathbf{g})$ fine grid FFT box into real space.
6. Calculate product of $\phi_{\alpha}(\mathbf{r})$ and $\sum_{\beta}K^{\alpha\beta}\phi_{\beta}(\mathbf{r})$ and sum over α if required (This occurs if the current NGWF is on the same atom (A) as the previous NGWF).
7. Deposit the charge density FFT box for the $\phi_{\alpha}(\mathbf{r})$ NGWFs on an atom (A, $\sum_{\alpha\in A}n(\mathbf{r};\alpha)$) into the simulation cell ($n(\mathbf{r})$).

Stage 1, the extraction of data stored from the PPD representation and conversion to an FFT box representation is currently performed on the host. As shown in Figure 3, the calculation of the $\sum_{\beta}K^{\alpha\beta}\phi_{\beta}(\mathbf{r})$FFT box is currently performed on the CPU. Although a GPU equivalent of this section of the code would be quicker, the data transfer associated with the numerous $\phi_{\beta}(\mathbf{r})$ FFT boxes makes such implementation significantly slower than the CPU implementation. Stage 2, the population of the coarse work array, includes the data transfer of the FFT boxes from the host to the GPU and the construction of the coarse grid FFT boxes. Stages 3–6 of Figure 3 have been ported to be performed entirely on the GPU. Stage 7, the deposition of the charge density in the simulation cell includes the movement of data from the GPU to the host followed by any necessary communication to transfer the charge density FFT box to the nodes that store the relevant slices of the simulation cell.

Table 1. Computational expense of FFT box and related operations within a ONETEP calculation.

Property/process	Time (%)
Charge density	36.0
Local potential integrals	20.2
Kinetic integrals	7.2
NGWF gradient	30.3

Values are given as a percentage of total run time.

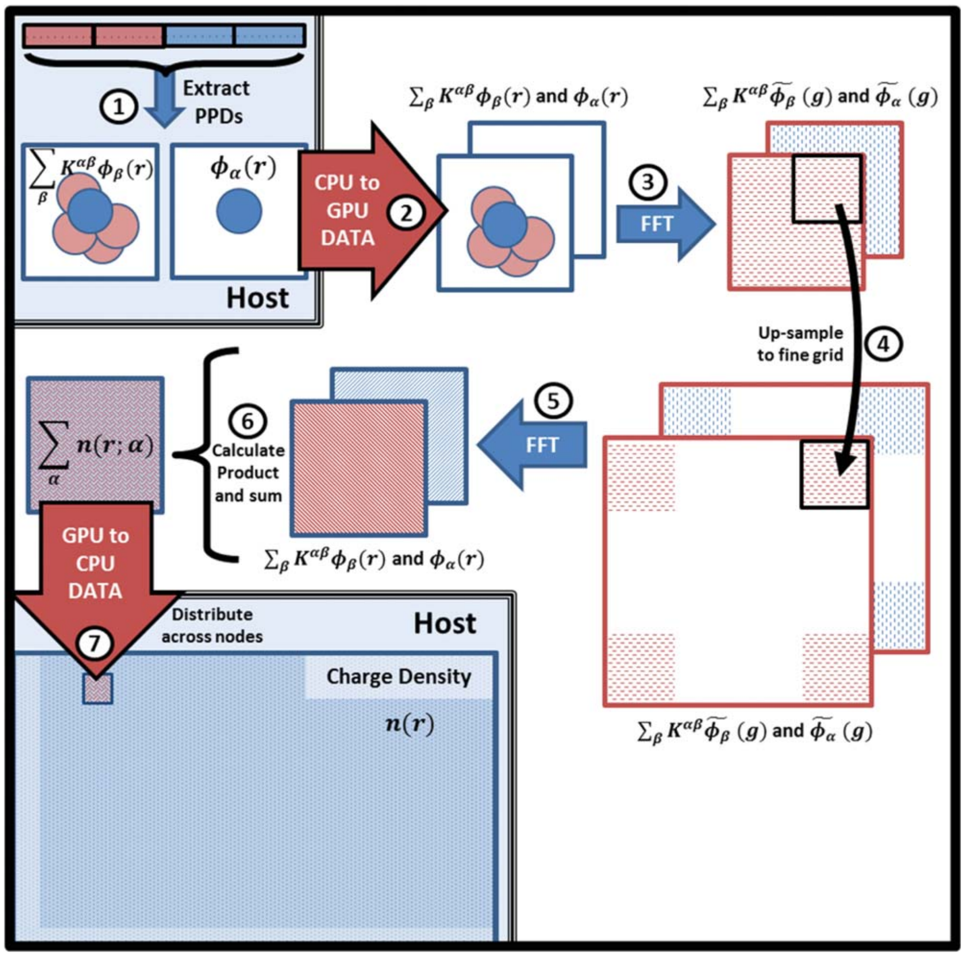
Extract PPDs
1
$\sum_\beta K^{\alpha\beta}\phi_\beta(r)$
$\phi_\alpha(r)$
Host
CPU to GPU DATA
2
$\Sigma_\beta K^{\alpha\beta}\phi_\beta(r)$ and $\phi_\alpha(r)$
3
FFT
$\Sigma_\beta K^{\alpha\beta}\widetilde{\phi_\beta}(g)$ and $\widetilde{\phi_\alpha}(g)$
Up-sample to fine grid
4
5
FFT
6
Calculate Product and sum
$\sum_\alpha n(r;\alpha)$
$\Sigma_\beta K^{\alpha\beta}\phi_\beta(r)$ and $\phi_\alpha(r)$
GPU to CPU DATA
7
Distribute across nodes
Host
Charge Density
$n(r)$
$\Sigma_\beta K^{\alpha\beta}\widetilde{\phi_\beta}(g)$ and $\widetilde{\phi_\alpha}(g)$

Figure 3. Calculation of the charge density FFT box, $\sum_{\alpha\in A} n(\mathbf{r};\alpha)$, for the NGWFs ($\varphi_\alpha(\mathbf{r})$) on atom A and the deposition of this FFT box data into the simulation cell ($n(\mathbf{r})$). Real space data are denoted by the ($\mathbf{r}$) variable and reciprocal space data by the ($\mathbf{g}$) variable. Host to GPU data transfers are highlighted with the DATA label.

The code has been ported for execution on the GPU through a combination of PGI accelerator pragmas to define code which is executed on the GPU, calls to CUDA Fortran functions for the management of data transfers and the CUFFT library to perform FFTs. This approach was chosen as it ensured clarity within the code and facilitated future developments. In some cases, this is a very straightforward process requiring only the addition of the !$acc region and !$acc end region pragmas. However, more complex changes are necessary to control data movement to achieve the performance benefits discussed below.

Table 2 shows the computational cost of each of the stages outlined in Figure 3 for the "tennis ball" dimer system described above. The complexity of the scheduling of GPU

Table 2. Timings for the stages of the calculation of the charge density.

	CPU		GPU								
	Xeon E5620		Tesla M2050			Tesla M2090			Kepler K20		
Stage	s	%	s	%	Acc	s	%	Acc	s	%	Acc
1. Extract PPDs	4.5	0.4	17.0	3.1	0.3	4.6	1.5	1.0	3.9	1.7	1.1
2. Populate coarse	1.7	1.1	71.5	13.0	0.2	39.5	13.1	0.3	43.0	19.1	0.3
3. Forward coarse FFT	110.5	9.0	12.3	2.2	9.0	9.4	3.1	11.8	4.2	1.9	26.1
4. Populate fine	77.2	6.3	16.3	3.0	4.7	17.7	5.9	4.4	10.8	4.8	7.2
5. Inverse fine FFT	830.5	68.0	126.2	22.9	6.6	97.1	32.3	8.6	51.0	22.6	16.3
6. Calculate product	76.9	6.3	12.9	2.4	5.9	14.2	4.7	5.4	10.3	4.6	7.4
7. Deposit charge density	107.8	8.8	294.4	53.5	0.4	118.1	39.3	0.9	102.2	45.3	1.1
Blocked total			550.6		2.2	300.6		4.1	225.4		5.4
Unblocked total	1221.1		340.5		3.6	271.1		4.5	225.4		5.4

CPU and GPU calculations were performed on 1 CPU core and 1 CPU core with 1 GPU respectively. Percentage times are relative to the total time taken for the calculation of the charge density. The stages are labeled as in Figure 3. Acc is the acceleration relative to the CPU timings.

kernels means that it is difficult to separate the costs of the individual stages in Figure 3. To show the cost of these stages individually, it is necessary to include calls to blocking CUDA functions that force these operations to occur in serial and thus permit each stage to be timed accurately. It must be stressed that these blocking functions may significantly reduce the performance of the code. As such, Table 2 shows both detailed timings for calculations in which the code is forced to run in serial due to the blocking routines and the total time for the calculation of the charge density when these functions are removed.

These results show that the CPU bottlenecks, the coarse and fine grid FFTs, are performed up to 26× faster on the GPU. The other compute intensive operations such as the population of the fine grid and the calculation of the product of the charge density for each atom also benefit from a 6–9× speed-up when executed on the GPU. The cost of the stages containing data transfers represents the majority (55–65%) of the computational cost of the GPU port of the code, this is a significant shift in the performance bottleneck when compared to the CPU implementation of the algorithm.

The observation that the copy on of the data has a larger effect on performance relative to the CPU code than the copy off (approximately 0.3× for the copy on and 0.4–0.9× for the copy off) may be explained by the fact that the copy on is performed for every single call to the GPU version of the code while the copy off is not. This is a result of the summation of the charge density for NGWFs that belong to the same atom. Indeed, an early version of the code did not perform this summation on the GPU resulting in particularly poor performance.

The performance benefit of the Kepler architecture is seen to be significant, particularly for the FFTs in stages 3 and 5 and the data transfer in stage 7 which are seen to take less than half the time of the same operations on the other GPU models shown here. The FFTs benefit significantly as the algorithms are able to utilize the increased amount of parallelism afforded by the increased number of streaming multiprocessors in the Kepler chipset. The observed variation for stage 7, the deposition of the charge density, is thought to be a result of the hardware configuration.

Local potential integrals

The local potential integrals are needed to construct the Hamiltonian matrix (the representation of the Hamiltonian operator in the set of NGWFs) and have the form shown in eq. (4). Where $\hat{V}_{\text{loc}}(\mathbf{r})$ is the sum of the local pseudo potential $\hat{V}_{\text{ps,loc}}(\mathbf{r})$, the Hartree potential $\hat{V}_H(\mathbf{r})$, and the exchange-correlation potential $\hat{V}_{XC}(\mathbf{r})$.

$$V_{\alpha\beta}=\int \phi^*_\alpha(\mathbf{r})\hat{V}_{\text{loc}}(\mathbf{r})\phi_\beta(\mathbf{r})d\mathbf{r} \qquad (4)$$

The calculation of the local potential integrals proceeds through the following stages, which are also noted in Figure 4:

1. The $\phi_\beta(\mathbf{r})$ NGWF is extracted from the PPD representation and stored in a coarse grid FFT box.

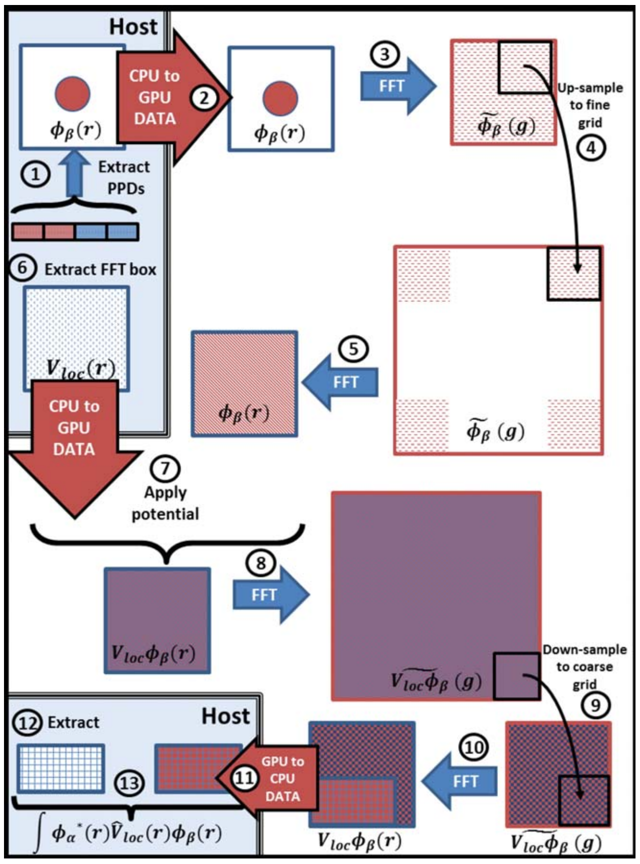

Figure 4. Calculation of local potential integrals $\int \varphi^*_\alpha(\mathbf{r})\hat{V}_{loc}(\mathbf{r})\varphi_\beta(\mathbf{r})d\mathbf{r}$. Real space data are denoted by the $(\mathbf{r})$ variable and reciprocal space data by the $(\mathbf{g})$ variable. Host to GPU data transfers are highlighted with the DATA label.

2. The $\phi_\beta(\mathbf{r})$ FFT box is moved to the coarse grid FFT box.
3. The $\phi_\beta(\mathbf{r})$ FFT box is transformed to reciprocal space with a forward FFT to obtain $\tilde{\phi}_\beta(\mathbf{g})$.
4. The reciprocal space $\tilde{\phi}_\beta(\mathbf{g})$ FFT box is up sampled to the fine grid.
5. The fine grid, reciprocal space $\tilde{\phi}_\beta(\mathbf{g})$ FFT box is transformed to real space with an inverse FFT.
6. The region of the local potential coinciding with the $\phi_\beta(\mathbf{r})$ FFT box is extracted from its simulation cell representation.
7. The local potential is applied to the fine grid $\phi_\beta(\mathbf{r})$ work array FFT box through a multiplication of the FFT boxes, giving the $\hat{V}_{\text{loc}}(\mathbf{r})\phi_\beta(\mathbf{r})$ FFT box on a fine grid.
8. The $\hat{V}_{\text{loc}}(\mathbf{r})\phi_\beta(\mathbf{r})$ FFT box is transformed to reciprocal space with a forward FFT to give $\widetilde{\hat{V}_{\text{loc}}\phi_\beta}(\mathbf{g})$.
9. The $\widetilde{\hat{V}_{\text{loc}}\phi_\beta}(\mathbf{g})$ FFT box is down sampled to a coarse grid FFT box.
10. The $\widetilde{\hat{V}_{\text{loc}}\phi_\beta}(\mathbf{g})$ FFT box is transformed to real space with an inverse FFT.
11. PPDs from $\hat{V}_{\text{loc}}(\mathbf{r})\phi_\beta(\mathbf{r})$ that overlap with $\phi_\alpha(\mathbf{r})$ are extracted in their entirety from the FFT box and stored in the PPD representation.
12. PPDs containing points from $\phi_\alpha(\mathbf{r})$ that overlap with $\phi_\beta(\mathbf{r})$ are extracted in their entirety from the PPD representation.

13. The $\int \phi_\alpha^*(\mathbf{r})\hat{V}_{\text{loc}}(\mathbf{r})\phi_\beta(\mathbf{r})d\mathbf{r}$ integral is calculated as the dot product over grid points of the common PPDs between $\phi_\alpha(\mathbf{r})$ and $\phi_\beta(\mathbf{r})$.

Stages 1, 12, and 13 are performed on the CPU in all cases and stages 2, 6, and 13 involve the transfer of data across the PCIe bus. Table 3 shows the cost of each of the stages outlined in Figure 4 using the "tennis ball" dimer calculation described in the section Chemical system. These results show that the FFTs and most of the compute intensive stages are performed between six and 26 times faster when executed on the GPU. Similar to the charge density code, the most significant bottlenecks in the GPU port of the code are stages that involve data transfers.

The acceleration of stage 7, the application of the potential ($\hat{V}_{loc}(\mathbf{r})\phi_\beta(\mathbf{r})$), which is the multiplication of $\hat{V}_{loc}(\mathbf{r})$ with $\phi_\beta(\mathbf{r})$ has a speedup of 2.4–4.9×. This is notably lower than that observed for stage 6 in the calculation of the charge density (5.4–7.4×), despite these stages representing mathematically equivalent operations. Although the FFT boxes are on the fine grid in both cases, there are subtle differences in the execution of these algorithms arising from the relative locations of the data; in the charge density version, the two factors are adjacent in memory, within the real and imaginary components of a complex array. In contrast, the factors for the local potential version are located in completely different arrays. This results in more efficient reads from global memory in the case of the charge density version.

The cost of the stages containing data transfers make up a large percentage of the total cost (66–75%), higher than observed for the calculation of the charge density. This is due to the fact that there are more data transfers in this process.

The performance benefits observed for the K20 GPU are even more significant here, the entire process takes half the time of the same code executed on a M2050 GPU. This is a result of the increased number of operations that are performed on the fine grid which, as aforementioned, benefit significantly from the increased hardware parallelism of the Kepler architecture. Also significant is the reduction in the cost of the data transfer in stage 6, the population of the potential $\hat{V}_{loc}$ FFT box on a fine grid.

Kinetic integrals

The kinetic energy, E_{kin}, is calculated through the evaluation of the kinetic integrals using eq. (5).[49]

$$E_{\text{kin}}=\sum_{\alpha\beta} K^{\beta\alpha}\int \phi_\alpha^*(\mathbf{r})\hat{T}(\mathbf{r})\phi_\beta(\mathbf{r})d\mathbf{r} \tag{5}$$

Where $\hat{T}$ is the kinetic energy operator $\hat{T}=-(1/2)\nabla^2$. The calculation of $\int \phi_\alpha^*(\mathbf{r})\hat{T}(\mathbf{r})\phi_\beta(\mathbf{r})d\mathbf{r}$ follows a process similar to that used for the local potential integrals, with stages 1–3 and stages similar to 10–13 performed in the same manner as detailed in Figure 4. The significant differences are that the application of the kinetic operator is performed on the coarse grid. The nature of the kinetic operator further simplifies matters as, unlike the local potential operator, it is not necessary to extract data representing the kinetic operator from the simulation cell and the same data may be reused for each $\int \phi_\alpha^*(\mathbf{r})\hat{T}(\mathbf{r})\phi_\beta(\mathbf{r})d\mathbf{r}$.

The timings for the kinetic integrals shown in Table 4 show that once again the bottlenecks within this section of the GPU port of the code are the transfer of data between the GPU and host and the conversions between the PPD and FFT box representations. These stages take up over 90% of the runtime. The application of the kinetic integral operator shows a higher level of acceleration (9.1–14.5×) than that of the local

Table 3. Timings for the stages of the calculation of the local potential integrals.

	CPU		GPUs								
	Xeon E5620		Tesla M2050			Tesla M2090			Kepler K20		
Stage	s	%	s	%	Acc	s	%	Acc	s	%	Acc
1. Extract	2.8	0.3	8.5	1.5	0.3	2.5	0.7	1.1	1.8	0.8	1.5
2. Populate	30.2	3.5	25.4	4.5	1.2	15.4	4.4	2	11.5	5.3	2.6
3. Forward Coarse FFT	27.7	3.2	3.1	0.5	9	2.4	0.7	11.8	1.1	0.5	25.9
4. Populate fine	19.4	2.3	4.1	0.7	4.8	4.5	1.3	4.3	2.7	1.2	7.2
5. Forward fine FFT	207.1	24	31.6	5.6	6.6	24.3	7	8.5	12.8	5.8	16.2
6. Populate potential	59	6.9	226.9	40.3	0.3	90.7	26.1	0.7	54.8	25	1.1
7. Apply potential	25	2.9	10.4	1.8	2.4	5.4	1.5	4.7	5.1	2.3	4.9
8. Forward fine FFT	206.9	24	31.6	5.6	6.6	24.3	7	8.5	12.8	5.8	16.2
9. Populate coarse	4.9	0.6	0.7	0.1	6.9	1.3	0.4	3.7	0.8	0.4	6.4
10. Inverse coarse FFT	27.2	3.2	3.1	0.5	8.9	2.3	0.7	11.7	1.1	0.5	25.7
11. Separate	6.3	0.7	0.8	0.1	8.3	1.4	0.4	4.6	0.6	0.3	11.1
12. Extract alpha	141.7	16.5	160.5	28.5	0.9	150.6	43.3	0.9	72	32.9	2
13. Integrate	103.1	12	57.1	10.1	1.8	22.8	6.5	4.5	42.2	19.3	2.4
Blocked total			563.6		1.5	347.9		2.5	219.1		3.9
Unblocked total	861.3		301.2		2.9	242.2		3.6	219.1		3.9

CPU and GPU calculations were performed on 1 CPU core and 1 CPU core with 1 GPU, respectively. Percentage times are relative to the time taken for the total time taken for the calculation of the local potential integrals. The stages are labeled as in Figure 4. Acc is the acceleration relative to the CPU timings.

Table 4. Timings for the stages of the calculation of the kinetic integrals.

	CPU		GPUs								
	Xeon E5620		Tesla M2050			Tesla M2090			Kepler K20		
	s	%	s	%	Acc	s	%	Acc	s	%	Acc
1. Populate	5.3	2	47.2	16.9	0.1	12.9	11.1	0.4	17.5	13.4	0.3
2. Forward coarse FFT	63.4	23.6	6.1	2.2	10.4	3.1	2.6	20.7	2.1	1.6	29.8
3. Apply operator	14.4	5.4	1.6	0.6	9.1	1	0.8	14.5	1.4	1	10.5
4. Inverse coarse FFT	59.7	22.2	6.1	2.2	9.9	3.1	2.6	19.6	2.1	1.6	28.6
5. Deposit	9.6	3.6	30.9	11	0.3	9.2	7.9	1	14.3	10.9	0.7
6. Extract and integrate	116.5	43.3	188.3	67.2	0.6	87.6	75	1.3	93.6	71.5	1.2
Blocked total			280.1		1	116.9		2.3	131		2.1
Unblocked total	268.9		127		2.1	112.4		2.4	131		2.1

CPU and GPU calculations were performed on 1 CPU core and 1 CPU core with 1 GPU, respectively. Percentage times are relative to the time taken for the total time taken for the calculation of the kinetic integrals. Acc is the acceleration relative to the CPU timings.

potential operator (2.4–4.8×). Presumably as this operation is applied on a coarse grid, in this case which permits more efficient data accesses. The performance of the nonaccelerated sections of the code (stages 1, 5, and 6 in Fig. 4) result in a slower total time for the Kepler card then for either Fermi card. However, as the performance of the isolated FFTs is significantly improved on the K20 card (29× vs. 10–20×), this is purely a result of the areas of the code executed on the CPU and results from the different models of CPU installed in the respective machines. The issues seen in the performance of the kinetic integrals code are of relatively low priority as they take up typically only 5–10% of the total calculation time.

NGWF gradient

As described in detail elsewhere,[43] the gradient of the energy with respect to the NGWF expansion coefficients is calculated using:

$$\left.\frac{\delta E}{\delta \phi_\alpha}\right|_{\mathbf{r}} = \sum_\beta [\hat{H}\phi_\beta](\mathbf{r})A^\beta + \sum_\gamma \phi_\gamma(\mathbf{r})B^\gamma. \quad (6)$$

Where the method being used to evaluate the gradient controls the nature of matrices A and B. The gradient is built up of contributions from the kinetic, local potential, and nonlocal potential operators before being subjected to kinetic energy preconditioning.[43] Finally, the gradient is extracted from the FFT box representation in the PPD format and "pruned" such that any points within the PPDs of the gradient that fall outside of the NGWF localization region are set to zero.

Currently, only the kinetic and local potential sections of the NGWF gradient code have been ported to GPUs as these closely follow the processes detailed in the section Local potential integrals. However, as these sections contribute around 60% of the runtime for the NGWF gradient calculation, the effect of the port of these stages is expected to be beneficial. The mechanism by which the local potential is applied is algorithmically identical to that described for stages 1–9 for the calculation of the local potential integrals in Figure 4. The algorithm for the kinetic operator differs in that the stages relating to the generation of the fine grid FFT box are excluded in the same manner as described for the kinetic integrals.

Table 5 shows the timings for the NGWF gradient calculation. The overall acceleration of the NGWF gradient code is low (1.6–2.2×) compared to that of the other ported regions (2.9–5.4×). This is a result of the relatively high expense of unaccelerated code in stages 4–7 that make up over 25% of the run time for this section of the code. The application of

Table 5. Timings for the stages of the calculation of the NGWF gradient.

	CPU		GPUs								
	Xeon E5620		Tesla M2050			Tesla M2090			Kepler K20		
Stage	s	%	s	%	Acc	s	%	Acc	s	%	Acc
1 Setup	124.7	11.1	156.4	22.1	0.8	120.8	23.4	1	121.4	23.8	1
2 Kinetic operator	77	6.9	10.5	1.5	7.3	6.1	1.2	12.6	4.3	0.9	17.7
3 Local potential	604.2	53.9	177.2	25	3.4	149.7	29.1	4	130.1	25.6	4.6
4 Non-local potential	275.6	24.6	309.8	43.8	0.9	210.2	40.8	1.3	223.8	44	1.2
5 Extract PPDs	0.3	0	0.6	0.1	0.5	0.3	0.1	1.1	0.3	0.1	1.1
6 Shave PPDs	4.8	0.4	4.8	0.7	1	3.8	0.7	1.3	4	0.8	1.2
7 Preconditioning	34.2	3.1	48.6	6.9	0.7	24.2	4.7	1.4	25.2	5	1.4
Blocked total			707.9		1.6	515.1		2.2	509.1		2.2
Unblocked total	1120.8		636.1		1.8	521.5		2.1	509.1		2.2

CPU and GPU calculations were performed on 1 CPU core and 1 CPU core with 1 GPU respectively. Percentage times are relative to the time taken for the total time taken for the calculation of the NGWF gradient. Acc is the acceleration relative to the CPU timings.

Table 6. Effect of the hardware on the accuracy of the electronic energy obtained from a ONETEP calculation on the "tennis ball" dimer using a kinetic energy cutoff of 1200 eV.

Hardware	Energy (Eh)
E5620, 1 core	−807.777563**121909**
E5620, 4 cores	−807.777563**127638**
1 M2050	−807.777563**134168**
4 M2050	−807.777563**132823**
1 M2090	−807.777563**134166**
4 M2090	−807.777563**127636**
1 K20	−807.777563**110516**
4 K20	−807.777563**127598**

Decimal places that vary depending on hardware are highlighted for clarity.

the local potential to the gradient is observed to have a significantly lower level of acceleration than the application of the kinetic operator due to the current use of a sequential data transfer to move the local potential to the GPU. Surprisingly, it can be seen that the blocked code is faster than the unblocked code in the case of the M2090 timings. This was also found to be the case in additional tests.

Further improvements are possible through porting stages 4–7 of the NGWF gradient code. However, these stages are fundamentally different from the types of operations we have been concerned with in the work reported here.

Precision

Table 6 shows the numerical precision achieved by the GPU port of the code for the total energy of the "tennis ball" dimer in atomic units (E_h). This is the total energy that results from combining all the stages described so far and calculated according to the theory utilized in the ONETEP code.[50]

We observe that the ported code achieves greater than μE_h levels of accuracy in all cases. This is within the norm of the precision achieved by parallel quantum chemistry codes and significantly better than the m E_h limit of the chemical processes that we expect to be simulated in such codes to reach the level of "chemical accuracy."

Although GPUs are capable of performing calculations at double precision (DP), the nature of the hardware means that there can be a significant disparity in the performance of GPU kernels executed at single precision (SP) and DP. This is a result of the higher number of SP cores compared to DP cores. A further advantage of performing calculations at the SP level is the reduced cost of the data transfer resulting from the reduced number of bytes required to store the data. It would be advantageous if we could exploit the increased performance available when using SP operations. This has been implemented in other GPU-accelerated quantum chemistry packages such as GAMESS-UK and Terachem,[51] which contain techniques aimed at utilizing the higher SP performance during the evaluation of the ERIs. These techniques involve performing the evaluation of the angular components of the integrals that provide a relatively small contribution to the Fock matrix at SP ("thresholding") or performing early iterations of the self consistent field (SCF) process at the SP level of accuracy and then switching to the DP level once certain convergence criterion have been reached ("switching"). This switching point is carefully chosen so as to prevent an increase in the number of SCF iterations.

As an initial investigation toward such approaches, we have implemented a dynamic precision calculation within the charge density code using a switching scheme analogous to the one described above. The switch between SP and DP is triggered once a certain convergence criterion has been reached. Currently, the energy difference between the results of the two previous inner loop iterations of the ONETEP scheme is used as the convergence criterion. The implementation of this scheme is relatively simple: two additional CPU stages are added to the scheme shown in Figure 3. These are a simple conversion of the $\phi_\alpha(\mathbf{r})$and $\sum_\beta K^{\alpha\beta}\phi_\beta(\mathbf{r})$ FFT boxes from DP to SP after stage 1 and the reverse for the $\sum_{\alpha\in A} n(\mathbf{r};\alpha)$ FFT box during stage 7 of this process (once the data transfer from the GPU to the host has completed). In this way, stages 2–6 of Figure 3, including the data transfers to and from the GPU, take place entirely in SP.

Initial results are shown in Figure 5 and Table 7. Figure 5 shows that the effect of using this technique on the final

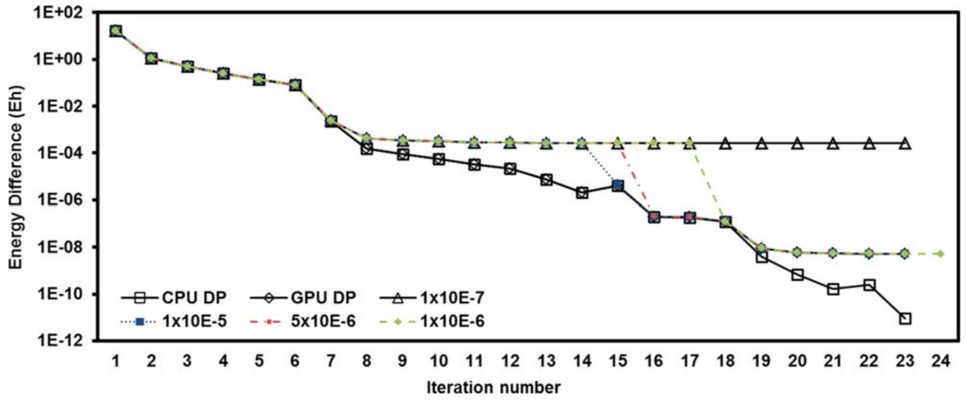

Figure 5. Difference in energy relative to the converged total energy of a reference calculation performed at full double precision on eight CPUs. The thresholds at which accuracy switches from SP to DP are shown in the legend. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Table 7. Effect of the value used for the dynamic precision switch on the timings for the calculation of the charge density.

	Times				
	Charge density		Total	Iterations	Total energy
Switch	(s)	(%)	(s)	energy	(Eh)
CPU reference					−808.46142717**8460**
CPU, full DP	1197	33	3610	23	−808.46142717**9391**
GPU, full DP	378	21	1787	23	−808.46142717**4129**
1×10^{-3}	322	19	1736	23 (9)	−808.46142717**4149**
1×10^{-4}	322	18	1748	23 (9)	−808.46142717**4149**
1×10^{-5}	291	17	1725	23 (14)	−808.46142717**4090**
5×10^{-6}	285	17	1723	23 (15)	−808.46142717**4096**
1×10^{-6}	289	16	1777	24 (17)	−808.46142717**4128**
1×10^{-7}	228	14	1626	23 (23)	−808.461**167046019**

Decimal places in the total energy that vary from the reference value obtained using CPU cores are highlighted as bold text. Numbers in parenthesis show the number of iterations performed at single precision.

energy is dependent on the value of the switching criterion used:

- A value of $1\times10^{-7}\,E_h$ or lower results in convergence to an erroneous energy that differs in the fourth decimal place. This is a consequence of the fact that the entire calculation is performed with the charge density calculated in SP.
- A value of $1\times10^{-6}\,E_h$ results in the correct energy but takes an additional iteration to reach convergence with a resulting increase in computational time of 4 s relative to the calculation using a value of $5\times10^{-6}\,E_h$.
- Switches at higher energy differences all converge to the same result as the full DP GPU code but with differing numbers of operations performed at SP as detailed in Table 7.

As shown in Table 7, the benefit of the SP code is small but consistent, with speedups relative to the CPU code ranging from 3.7× to 4.1× compared to 3.2× for the full-DP calculation on the GPU. However, these initial tests use a relatively small molecule and tests with larger systems are required. It would appear that fine tuning of the switch value is unlikely to give a major advantage but rather a conservative value should be used to avoid the danger of converging to an erroneous final energy. It should be emphasized, however, that with the exception of the extreme case of the $1\times10^{-7}\,E_h$ switch threshold, all other thresholds produce energies of the same precision as the full-DP code.

Multiple GPU performance

Figures 6a–6c show the results for calculations performed on the 181 atom "tennis ball" dimer, 495 atom cellulose, and 954 atom cellulose systems, respectively. Calculations are performed using 1, 2, and 4 GPUs of various types, except for the CPU timings where all available cores were used.

These results show that the GPU-enabled port of the code is consistently faster than the CPU implementation when the equivalent number of CPU cores and GPUs are used. As would be expected, the performance of the ported code on different GPUs follows a trend associated with the number of CUDA cores and clock speed of the hardware. It can be seen that an increase in the number of GPUs does result in an increased level of performance and this is consistent for calculations of different sizes, with a single GPU outperforming four CPU cores in some cases.

Figure 7 shows that the strong scaling performance of the GPU port of the code is excellent at GPU counts up to 4, the largest number to which we have access across all types of GPU tested. As a CPU code ONETEP achieves good scaling on 100's of cores, scaling to such numbers of GPUs will need to be explored in the future. Currently, these tests cannot be performed within a reasonable timescale as they require a significant amount of the total resources on the available machines and are limited by the queuing rate of these machines.

Initial investigations have shown that load balancing is complicated by factors such as the cost of the data transfer across threads. This may vary significantly between MPI processes depending on variables such as the use of the summation in stage 6 of the calculation of the charge density (Fig. 3), which depends on the atomic environment of the NGWFs. Other issues relating to hardware configuration have also been identified but require further investigation.

Conclusions

We have presented the first version of the ONETEP code for GPU-based coprocessors. Our work has focused on porting to the GPU the parts that involve FFT box-based operations which are among the most computationally intensive parts of the code. The performance of some stages of the code is observed to be accelerated by a factor of up to 30×. However, the transfer of data between the GPU and host machine is a significant bottleneck in the reported version of the code. As such, the performance of the ported code on a single GPU is similar to that of standard ONETEP on a single multicore CPU. For further performance improvement, extension of the existing port will be required, such as the development of GPU-specific algorithms for the interconversion between the FFT

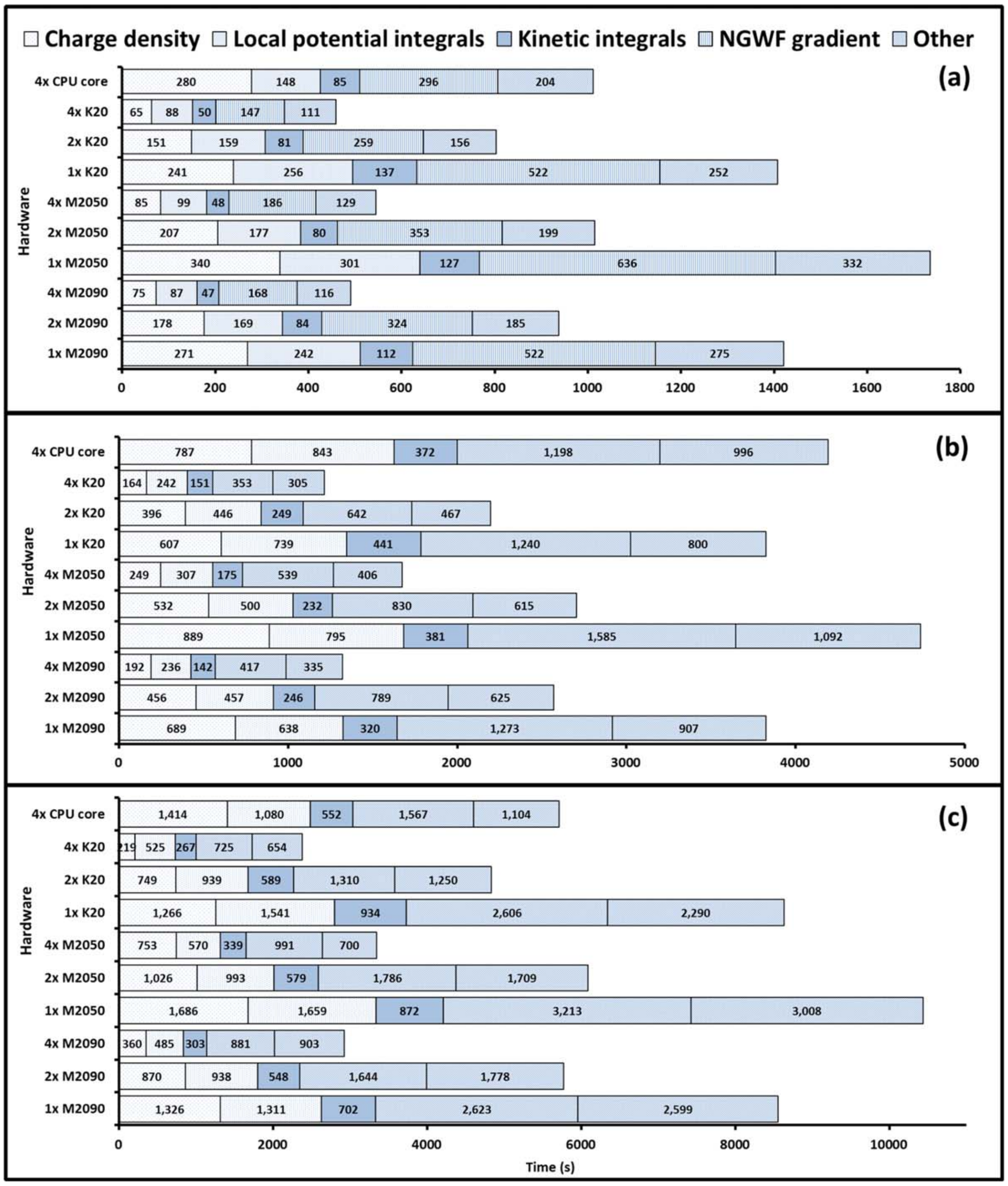

Figure 6. Scaling performance of the GPU-enabled version of the ONETEP code. Panels (a), (b), and (c) show the results for calculations performed on the "tennis ball" dimer, 495 atom cellulose, and 954 atom cellulose systems, respectively. From left to right, the bars denote the timing for the calculation of the charge density, the local potential integrals, kinetic integrals, NGWF gradient, and unaccelerated code, respectively. An equal number of GPUs and CPU cores are used in all cases. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

box and PPD data structures. This will reduce the current data-transfer bottlenecks in the GPU code, as will the explicit use of asynchronous data transfers. Other possibilities lie in refactoring the code to exploit the currently untouched layer of parallelism available through the simultaneous execution of multiple GPU kernels. A related development would be the use of batched FFTs, when permitted by the memory requirements of the algorithms. This is not currently possible within the ported ONETEP code as the requisite changes involve code relating to the communication between MPI threads. A significant number of processes within ONETEP are not yet ported to GPUs. These include the simulation cell FFTs, the calculation of the XC energy, and calls to sparse and dense matrix algebra subroutines. The utilization of both CPU cores and GPUs simultaneously represents a further challenge, currently the CPU cores simply pass the computationally intensive operations to the GPU and then stand idle until the computation is complete.

The development of the ported code using the PGI accelerator model has resulted in minimal disruption to the existing

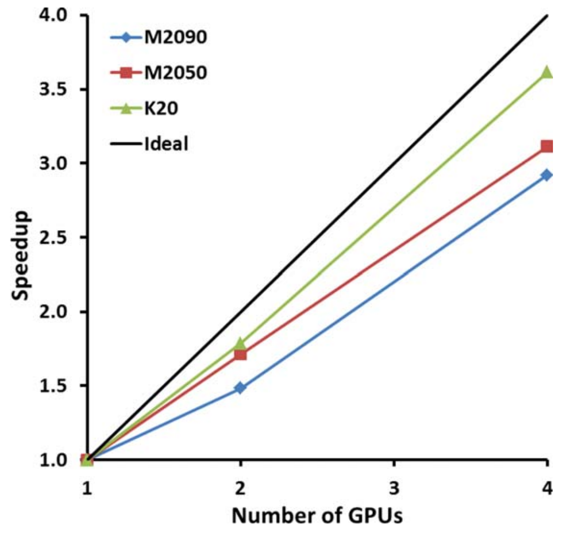

Figure 7. Strong scaling performance of the GPU enabled port of the ONETEP code for calculations on the 957 atom cellulose system relative to ideal performance. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

ONETEP code base. As this model will soon support accelerator platforms such as the Intel Xeon Phi and GPUs from AMD, it will be possible to compile the current code base for execution on these architectures with minimal effort. Also, the conversion of this code to OpenACC should be a straightforward process which will allow compilation of the code using products from other vendors.

The initial investigation into a dynamic precision scheme for the ONETEP energy calculation has shown it is possible to take advantage of the enhanced SP capabilities of GPUs. Further investigations into this issue are justified as it is shown that many operations may be performed at SP. It is also worth noting that execution of SP operations on a CPU will also result in a performance benefit. Accordingly, more extensive validation of the dynamic precision approach and other techniques such as thresholding are planned.

The developments reported here should be applicable to more extensive refactoring of the core algorithms of the code toward emerging massively parallel HPC platforms with many hundreds of thousands of CPU cores and coprocessors. In that context, our current MPI-based code will need to be extended to mixed-mode OpenMP-MPI parallelism and work flow models of communication should be utilized to improve load balancing.

In summary, the GPU port of ONETEP presented here enables ONETEP users to utilize the many GPU-enabled machines currently available. Although the version of the code described here does not provide a significant performance advantage relative to a multicore CPUs, it is clear that forthcoming developments will enhance the performance of the code through the removal of the current data-transfer bottleneck and allow simulations to be performed in less time than possible on CPU-only platforms.

Acknowledgments

The authors would especially like to thank Mark Berger and Timothy Lanfear of NVIDIA for discussions and support. We would also like to thank PGI for their support, in particular Matthew Colgrove and Pat Brooks for their hospitality and helpful discussions. We are also grateful to the e-Infrastructure South consortium for access to the Emerald GPU cluster at Rutherford Appleton Laboratory, in particular Istvan Reguly for useful discussions. We are grateful to iSolutions unit of the University of Southampton for access to the GPU nodes of the Iridis3 supercomputer and administrative support.

Keywords: linear scaling · graphical processing units · heterogeneous computing · quantum chemistry

How to cite this article: K. Wilkinson, C.-K. Skylaris. *J. Comput. Chem.* **2013**, *34*, 2446–2459. DOI: 10.1002/jcc.23410

[1] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, J. D. Owens, *Computer* **2003**, *36*, 54.
[2] General-Purpose Computation on Graphics Hardware, Available at: http://www.gpgpu.org (accessed on: January, 2013).
[3] I. S. Ufimtsev, T. J. Martinez, *J. Chem. Theory Comput.* **2008**, *4*, 222.
[4] K. A. Wilkinson, P. Sherwood, M. F. Guest, K. J. Naidoo, *J. Comp. Chem.* **2011**, *32*, 2313.
[5] J. Nickolls, I. Buck, M. Garland, K. Skadron, *Queue* **2008**, *6*, 40.
[6] PGI CUDA Fortran Compiler, Available at: http://www.pgroup.com/resources/cudafortran.htm
[7] J. E. Stone, D. Gohara, G. Shi, *IEEE Des. Test Comput.* **2010**, *12*, 66.
[8] M. Woo, J. Neider, T. Davis, D. Shreiner, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd ed.; Addison-Wesley Longman Publishing: Boston, MA, **1999**.
[9] J. Bolz, I. Farmer, E. Grinspun, P. Schröoder, *ACM Trans. Graph.* **2003**, *22*, 917.
[10] K. Fatahalian, J. Sugerman, P. Hanrahan, In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '04; ACM: New York, **2004**; pp. 133–137.
[11] M. Wolfe, Implementing the PGI accelerator model. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10; ACM: New York, **2010**; pp. 43–50.
[12] OpenACC. Directives for Accelerators, Available at: http://www.openacc-standard.org (accessed on: January, 2013).
[13] L. Dagum, R. Menon, *IEEE Comput. Sci. Eng.* **1998**, *5*, 46–55.
[14] CUBLAS User Guide, Available at: http://docs.nvidia.com/cuda/cublas/index.html (accessed on: January 2013).
[15] CUFFT User Guide, Available at: http://docs.nvidia.com/cuda/cufft/index.html (accessed on: January 2013).
[16] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, K. Schulten, *J. Mol. Graph. Model.* **2010**, *29*, 116.
[17] M. J. Harvey, G. De Fabritiis, *Wiley Interdiscip. Rev. Comput. Mol. Sci.* **2012**, *2*, 734.
[18] J. A. Baker, J. D. Hirst, *Mol. Inform.* **2011**, *30*, 498.
[19] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, K. Schulten, *J. Comput. Chem.* **2007**, *28*, 2618.
[20] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, L. G. Pedersen, *J. Chem. Phys.* **1995**, *103*, 8577.
[21] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, R. C. Walker, *J. Chem. Theory Comput.* **2012**, *8*, 1542.
[22] S. Le Grand, A. W. Götz, R. C. Walker, *Comput. Phys. Commun.* **2013**, *184*, 374.
[23] J. A. Anderson, C. D. Lorenz, A. Travesset, *J. Comput. Phys.* **2008**, *227*, 5342.
[24] M. J. Harvey, G. Giupponi, G. De Fabritiis, *J. Chem. Theory Comput.* **2009**, *5*, 1632.
[25] M. J. Harvey, G. De Fabritiis, *J. Chem. Theory Comput.* **2009**, *5*, 2371.
[26] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, T. L. Windus, *J. Chem. Theory Comput.* **2010**, *6*, 696.
[27] A. Asadchev, M. S. Gordon, *J. Chem. Theory Comput.* **2012**, *8*, 4166.

[28] I. S. Ufimtsev, T. J. Martinez, *J. Chem. Theory Comput.* **2009**, *5*, 1004.
[29] I. S. Ufimtsev, T. J. Martinez, *J. Chem. Theory Comput.* **2009**, *5*, 2619.
[30] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, A. Aspuru-Guzik, *J. Phys. Chem. A* **2008**, *112*, 2049.
[31] A. E. DePrince, J. R. Hammond, *J. Chem. Theory Comput.* **2011**, *7*, 1287.
[32] W. Ma, S. Krishnamoorthy, O. Villa, K. Kowalski, *J. Chem. Theory Comput.* **2011**, *7*, 1316.
[33] K. Bhaskaran-Nair, W. Ma, S. Krishnamoorthy, O. Villa, H. J. J. van Dam, E. Apr, K. Kowalski, *J. Chem. Theory Comput.* **2013**, *9*, 1949.
[34] M. Hutchinson, M. Widom, *Comput. Phys. Commun.* **2012**, *183*, 1422.
[35] L. Genovese, M. Ospici, T. Deutsch, J. -F. Mehaut, A. Neelov, S. Goedecker, *J. Chem. Phys.* **2009**, *131*, 034103.
[36] C. -K. Skylaris, P. D. Haynes, A. A. Mostofi, M. C. Payne, *J. Chem. Phys.* **2005**, *122*, 084119.
[37] S. J. Fox, C. Pittock, T. Fox, C. S. Tautermann, N. Malcolm, C. -K. Skylaris, *J. Chem. Phys.* **2011**, *135*, 224107.
[38] S. Goedecker, *Rev. Mod. Phys.* **1999**, *71*, 1085.
[39] D. R. Bowler, T. Miyazaki, *Rep. Prog. Phys.* **2012**, *75*, 036503.
[40] W. Kohn, *Phys. Rev. Lett.* **1996**, *76*, 3168.
[41] E. Prodan, W. Kohn, *Proc. Natl. Acad. Sci. USA* **2005**, *102*, 11635.
[42] C. -K. Skylaris, A. A. Mostofi, P. D. Haynes, O. Diéguez, M. C. Payne, *Phys. Rev. B* **2002**, *66*, 035119.
[43] A. A. Mostofi, P. D. Haynes, C. -K. Skylaris, M. C. Payne, *J. Chem. Phys.* **2003**, *119*, 8842.
[44] C. -K. Skylaris, P. D. Haynes, A. A. Mostofi, M. C. Payne, *Phys. Stat. Sol. (b)* **2006**, *243*, 973.
[45] N. D. M. Hine, P. D. Haynes, A. A. Mostofi, C. -K. Skylaris, M. C. Payne, *Comput. Phys. Commun.* **2009**, *180*, 1041.
[46] D. Baye, P. H. Heenen, *J. Phys. A Math. Gen.* **1986**, *19*, 2041–2060.
[47] X. -P. Li, R. W. Nunes, D. Vanderbilt, *Phys. Rev. B* **1993**, *47*, 10891.
[48] P. D. Haynes, C. -K. Skylaris, A. A. Mostofi, M. C. Payne, *J. Phys. Condens. Matter* **2008**, *20*, 294207
[49] C. -K. Skylaris, A. A. Mostofi, P. D. Haynes, C. J. Pickard, M. C. Payne, *Comput. Phys. Commun.* **2001**,*3*, 315–322.
[50] A. A. Mostofi, C. -K. Skylaris, P. D. Haynes, M. C. Payne, *Comput. Phys. Commun.* **2002**, *147*, 788.
[51] N. Luehr, I. S. Ufimtsev, T. J. Martinez, *J. Chem. Theory Comput.* **2011**, *2011*, 949.

Received: 28 April 2013
Revised: 17 June 2013
Accepted: 19 July 2013
Published online on 26 August 2013