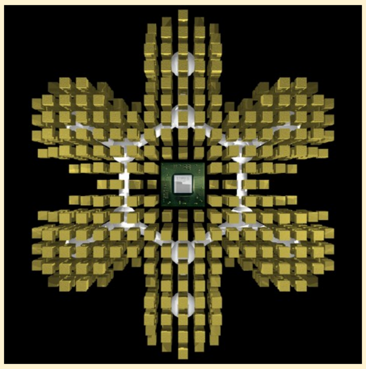
Real-Space Density Functional Theory on Graphical Processing Units: Computational Approach and Comparison to Gaussian Basis Set Methods

Xavier Andrade and Alán Aspuru-Guzik*

Department of Chemistry and Chemical Biology, Harvard University, 12 Oxford Street, Cambridge, Massachusetts 02138, United States

S *Supporting Information*

ABSTRACT: We discuss the application of graphical processing units (GPUs) to accelerate real-space density functional theory (DFT) calculations. To make our implementation efficient, we have developed a scheme to expose the data parallelism available in the DFT approach; this is applied to the different procedures required for a real-space DFT calculation. We present results for current-generation GPUs from AMD and Nvidia, which show that our scheme, implemented in the free code Octopus, can reach a sustained performance of up to 90 GFlops for a single GPU, representing a significant speed-up when compared to the CPU version of the code. Moreover, for some systems, our implementation can outperform a GPU Gaussian basis set code, showing that the real-space approach is a competitive alternative for DFT simulations on GPUs.



1. INTRODUCTION

For many years, the constant reduction in the size of the transistors, as described by Moore's law,[1] has been translated into an increment of the processing capacity of central processing units (CPUs). However, due to the limitations in efficiency and power consumption related to the breakdown of Dennard scaling,[2] CPU designers moved toward parallel processing to profit from the increasing number of transistors. This trend toward parallelism can be seen in current CPUs, which have multiple cores, with each core capable of executing multiple threads and containing vectorial processing units that operate simultaneously on several sets of values.

Simultaneously, a more parallel kind of processor appeared: the graphical processing unit (GPU). Originally designed for real-time rendering of images, a computationally intensive and highly parallel task, modern GPUs are also suitable for general purpose computing, in particular, for high-performance numerical simulations. They typically have thousands of execution units, which give them approximately 1 order of magnitude higher processing power than a CPU. This difference is explained by different design strategies: while a single instruction may be executed faster on a CPU, GPUs can execute thousands of them in parallel.

In the last years, there has been a considerable interest in applying GPUs to computational science. Whereas, in some areas of atomistic simulations, GPUs are becoming a standard tool,[3] in the electronic structure domain and, in particular, in density functional theory (DFT),[4] the adoption of GPUs has been slower. The first full electronic-structure implementation on GPUs was TeraChem, presented by Ufimtsev and Martínez in 2008.[5] Currently, several electronic structure codes have also incorporated some degree of GPU acceleration.[6−21]

Still, how to get the most out of a GPU for modeling electronic systems is an active area of research,[15,22,23] as simulation approaches that are efficient on a CPU might not be as efficient on a GPU. These approaches can be improved or replaced by other methods that are better suited to massively parallel architectures. In this respect, the large diversity of methods used for electronic structure methods by chemists and physicists offers an interesting starting point to explore the application of GPUs to the simulation of electronic systems.

In this work, we focus on one particular approach for electronic structure, real-space DFT, and how it can be adapted to GPUs. Although not as widely used as basis-set methods, the real-space grid discretization is a popular alternative for DFT simulations.[24−37] Its main features are the flexibility to model different types of electronic systems, the systematic control of the discretization error, and its potential for parallelization in distributed memory systems with thousands of processors.[12,38−40]

The development of an efficient GPU implementation does not only involve rewriting and optimizing low-level routines for the GPU. For complex scientific software, choosing an appropriate design strategy for the entirety of the code can be fundamental for optimal GPU performance. This work is mainly focused on this issue: we have developed a scheme to

Received: June 19, 2013
Published: August 15, 2013

apply DFT efficiently on GPUs by exposing the available parallelism to the low-level routines.

Our approach was developed for the implementation of GPU support in the Octopus code[12,30,40] and is freely available under an open source license.[42] Octopus is used by several research groups for theoretical development[43−50] and applications in different fields of chemistry and physics.[51−60] In this article, we describe in detail our general strategy and its application to the different procedures required for real-space DFT, extending previous results for real-time time-dependent DFT.[11,12] Our GPU implementation is based on OpenCL,[61] a standard and portable framework for writing code for parallel processors, so it can run on GPUs, CPUs, and other processing devices, from different vendors.

To assess the efficiency of our implementation, we perform a series of tests involving top-end GPUs from Nvidia and AMD, and a set of molecular systems of different sizes. We provide different indicators that illustrate the performance of our implementation: numerical throughput (number of floating point operations executed per unit of time), total calculation times, and comparisons with the CPU version of the code and a different GPU-DFT implementation. These results show that real-space DFT is an interesting and competitive approach for GPU-accelerated electronic structure calculations.

2. REAL-SPACE DENSITY FUNCTIONAL THEORY

In the Kohn−Sham (KS) formulation of DFT, the electronic density of an interacting electronic system, $n(\boldsymbol{r})$, is generated by a set of single-particle orbitals, or states, $\varphi_k(\boldsymbol{r})$. These orbitals are generated by the KS equations[4]

$$H[n]\varphi_k(\boldsymbol{r}) = \varepsilon_k\varphi_k(\boldsymbol{r}) \tag{1a}$$

$$n(\boldsymbol{r}) = \sum_{k=1}^{N} \varphi_k^*(\boldsymbol{r})\varphi_k(\boldsymbol{r}) \tag{1b}$$

where the H operator is the KS effective single-particle Hamiltonian (atomic units are used throughout)

$$H[n] = -\frac{1}{2}\nabla^2 + v_{\mathrm{ext}}(\boldsymbol{r}) + v_{\mathrm{hxc}}[n](\boldsymbol{r}, t) \tag{2}$$

The external potential v_{ext} contains the nuclear potential and other external fields that may be present; v_{hxc} represents the electron−electron interaction and is usually divided in the Hartree term, which contains the classical electrostatic interaction between electrons, and the exchange and correlation (XC) potential.

To solve the KS equations numerically, the orbitals, the density, and other fields need to be represented as a finite set of numbers. The selection of the discretization scheme is probably the most important aspect in the numerical solution of the electronic structure problem. Traditionally, a basis set expansion is used: atomic orbitals for molecules, and plane waves for crystalline systems. In the real-space approach, instead of a basis, fields are discretized in a grid. This provides a simple and flexible scheme that is suitable to model both finite and periodic systems.[62] The electron ion interaction is modeled by the pseudo-potential approximation, or the projector-augmented-wave method,[33] that removes the problem of representing the hard Coulomb potential, so uniform grids can be used.

One of the main advantages of the real-space grid approach is that the discretization error can be controlled systematically by reducing the spacing and increasing the size of the box. Of course, this increases the number of points and, proportionally, the time and memory cost of the calculation. To keep the number of grid points to a minimum, our implementation uses arbitrarily shaped grids. This choice makes the code more complex but allows for an important reduction in grid size compared with a simpler cubic grid. For molecular systems, we use a uniform grid whose shape is given by the union of spheres around each atom, as shown in Figure 1. This strategy avoids placing points in regions where the value of the density is not significant for the desired accuracy.

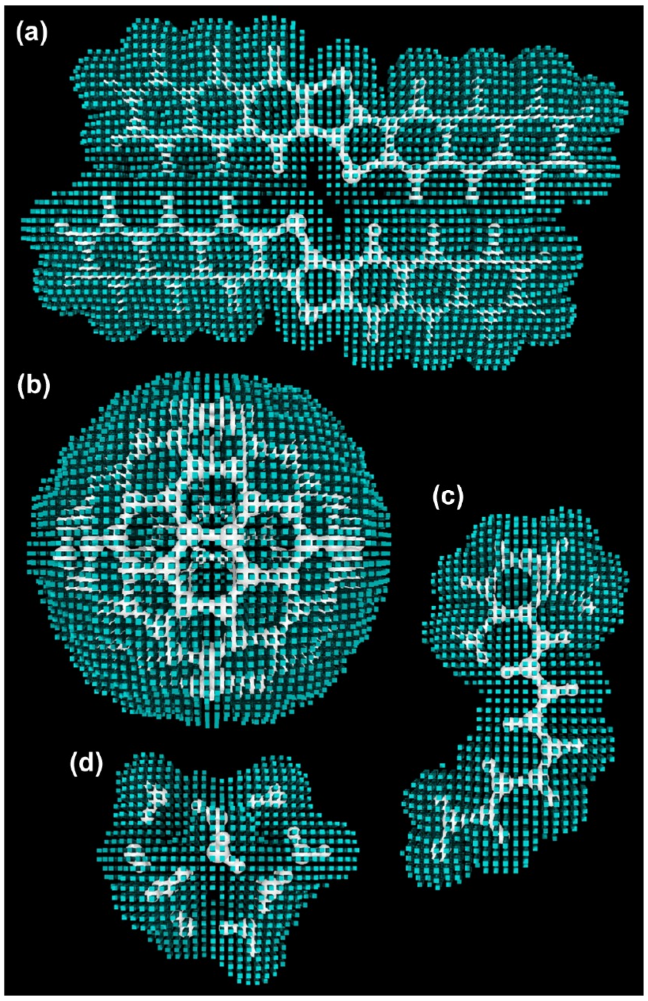


Figure 1. Examples of real-space grids adapted to the shape of different molecular systems: (a) DAT−thiophane dimer, (b) C_{180} fullerene, (c) *cis*-retinal, and (d) water cluster. The (cyan) cubes mark the position of the grid points. For visualization purposes, we represent smaller and coarser grids than the ones used for actual calculations.

3. NUMERICAL SOLUTION OF REAL-SPACE DENSITY FUNCTIONAL THEORY

We now describe the numerical procedure to solve the KS equations in real space. As it is standard in Hartree−Fock (HF) and DFT, in order to account for the nonlinearity introduced by the density dependence in eq (1a), a self-consistent field (SCF) iterative scheme is used. A new set of orbitals and density are generated with each SCF iteration; this involves several numerical procedures, which are shown in Figure 2.

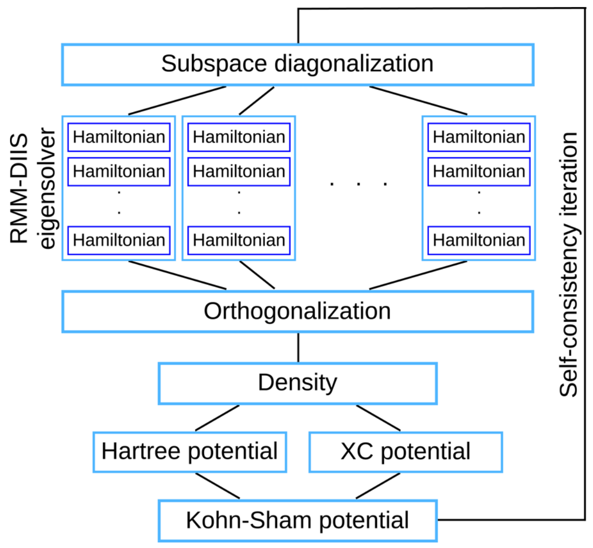


Figure 2. Schematic of a density functional theory calculation in real space using a self-consistency scheme and the residual minimization-direct inversion in the iterative subspace (RMM-DIIS) eigensolver. The boxes represent the different numerical procedures that need to be performed.

In every SCF step, we need to find the lower eigenvectors and eigenvalues of the KS Hamiltonian for a fixed density. In real space, the discretization of the KS Hamiltonian, eq 2, is done using a high-order finite-differences representation.[25] As this results in a sparse operator, the diagonalization is done using iterative methods that do not require the Hamiltonian matrix to be built explicitly, only to be applied as an operator. In this work, we use the efficient residual minimization-direct inversion in the iterative subspace (RMM-DIIS) eigensolver[63,64] (not to be confused with the DIIS SCF scheme[65]). To precondition the eigensolver, we use the filter operator proposed by Saad et al.[66]

In practice, it is not worth it to find a converged solution of the eigenvalue problem at each SCF iteration: instead, we do a fixed number of eigensolver iterations per step. In this manner, the eigenproblem convergence is achieved toward the end of the SCF cycle.

The RMM-DIIS scheme requires the application of the KS Hamiltonian and two additional procedures that act over the whole set of orbitals: orthogonalization and subspace diagonalization. Given a set of orbitals, the orthogonalization procedure performs a linear transformation that generates a new orthogonal set. Similarly, subspace diagonalization is an effective method to remove contamination between orbitals. It calculates the representation of the KS Hamiltoninan in the subspace spanned by a set of orbitals, and generates a new set where the subspace Hamiltonian is diagonal.

After the eigensolver steps and the posterior orthogonalization, a new set of orbitals and a new density are obtained; this density is mixed with the densities from previous steps to generate a new guess density according to the Broyden scheme.[67,68]

From the new guess density, the new KS effective potential is calculated. Numerically, the most expensive part of this step is obtaining the Hartree potential, V_H, that requires the solution of the Poisson equation

$$\nabla^2 V_H(\mathbf{r}) = -4\pi n(\mathbf{r}) \tag{3}$$

In our implementation, we use a Poisson solver based on fast Fourier transforms. The XC potential, v_{xc}, also needs to be recalculated. This is approximated by a local or semilocal expression that is evaluated directly on the grid.

4. GENERAL GPU OPTIMIZATION STRATEGY

In this section, we discuss the general scheme that we have developed to solve efficiently the real-space DFT equation on GPUs. This strategy was designed taking into account the strengths and weaknesses of the current generation of GPUs, but is also effective for CPUs with vectorial floating point units.

For optimum efficiency, GPUs need to operate simultaneously over large amounts of data so that the numerous independent operations fill the execution units and hide operation and memory latency (the time it takes for the result of an instruction to be available to other instructions). A way to fulfill this requirement is to expose data-parallelism to the low-level routines. For example, if the same operation needs to be performed over certain data objects, the routines should receive as an argument a group of those objects, instead of operating over one object per call.

To expose parallelism in the DFT case, our GPU optimization strategy is based on the concept of blocks of KS orbitals. Instead of acting over a single KS orbital at a time, performance critical routines receive a group of orbitals as argument. By operating simultaneously over several orbitals, the amount of parallelism exposed to the processor is increased considerably. In Figure 3, we show a scheme of how this concept works.

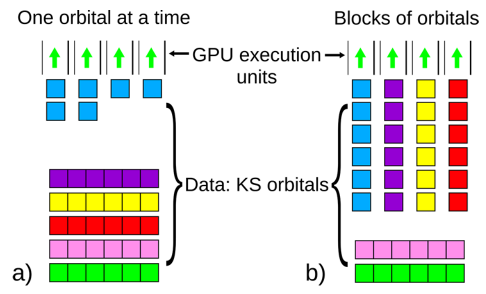


Figure 3. Scheme illustrating the blocks-of-orbitals strategy for DFT on GPUs. (a) Operating on a single orbital might not provide enough parallelism for the GPU to perform efficiently. (b) By operating simultaneously over several orbitals, there is a larger degree of data parallelism and there is less divergence among GPU threads.

The blocks-of-orbitals strategy has an additional advantage: in a GPU, threads are divided in groups of 32 (Nvidia) or 64 (AMD), called warps or wavefronts; for efficient execution, all threads in a warp must execute exactly the same instruction sequence. Since the same operation has to be performed over each orbital, we can assign operations corresponding to different orbitals to different threads in a warp. This ensures that the execution within each warp is regular, without divergences in the instruction sequence. In a CPU, vectorial floating point units play a similar role as warps.

A possible drawback of the block-of-orbitals approach is that memory access issues might appear, as working with a larger amount of data can saturate caches and reduce their ability to speed-up memory access. This is especially true for CPUs, which rely more on caches than GPUs. Larger blocks can also increase the amount of memory required for temporary data, so using blocks that are too large might be detrimental for performance.

In our implementation, the number of orbitals in a block, or block size, is variable and controlled at execution time. Ideally, the block size should be an integer multiple of the warp size. This might not be possible if not enough orbitals are available; in such a case, the block size should be a divisor of the warp size. Following these considerations, we restrict our block size to be a small power of two. (This has the additional advantage that the integer multiplication by the block size, required for array address calculations, can be done using the cheaper bit-shift instructions.)

The way blocks of orbitals are stored in memory is also fundamental for optimal performance. A natural scheme would be to store the coefficients for each orbital (Figure 4a)

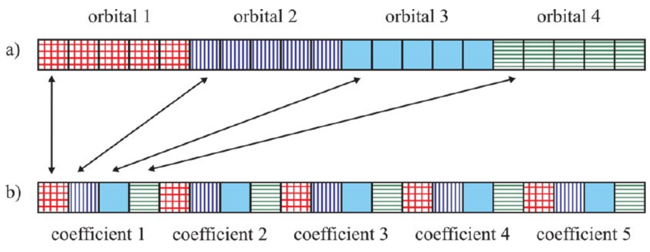


Figure 4. Example of memory layout for a block of four orbitals with five coefficients each: (a) Standard memory layout where each orbital is contiguous in memory. (b) Optimal memory layout where all the coefficients in a block are contiguous. The arrows indicate the relation of the position of the first coefficient in both schemes.

contiguously in memory so that each orbital in a block can be easily accessed. However, memory access is usually more efficient when threads access adjacent memory locations as loads or stores go to the same cache lines. Because, in our approach, consecutive threads are assigned to different orbitals, we order blocks by the orbital index first and then by the discretized r-index, ensuring that adjacent threads will access adjacent memory locations (Figure 4b).

In the following sections, we show how these general strategies are applied to the different numerical procedures that were introduced in section 3. For each operation, we show the numerical performance that our implementation obtains for a test system, β-cyclodextrin, on an Intel Core i7 3820 CPU and two GPUs, an AMD Radeon HD 7970 and a Nvidia Tesla K20 (shown in Figure 5). Details about the platforms and the calculations can be found in section 12.

Figure 5. AMD Radeon HD 7970 and Nvidia Tesla K20 GPU cards used for the numerical tests.

5. KOHN–SHAM HAMILTONIAN

The application of the KS Hamiltonian, eq 2, is the basic operation of the real-space DFT approach; as such, it is the first target for efficient GPU execution. Moreover, the KS Hamiltonian application is also used in other DFT-based simulations, such as on-the-fly molecular dynamics,[69] and response calculations in time[70] and frequency domains.[71]

As a matrix, the real-space KS Hamiltonian operator is sparse, with the number of coefficients proportional to the number of grid points. While the matrix could be stored in a sparse form, it is not convenient to do so. It is more efficient to use it in operator form, with three different terms that are applied independently: the kinetic energy operator, the local potential, and the nonlocal potential.

5.1. Kinetic Energy Operator.

In real space, the kinetic-energy operator corresponds to the Laplacian differential operator. Whereas, in a basis-set approach, this term is calculated exactly, in real-space, the Laplacian is approximated using high-order finite differences.[25] Numerically, this is a stencil calculation, where the value at each point is calculated as a linear combination of the neighboring-point values. The stencil represents the grid points used in the calculation of the differential operator; see Figure 6 for an example. In the simulations presented in this paper, we use a fourth-order approximation that, in 3D, results in a stencil size of 25.

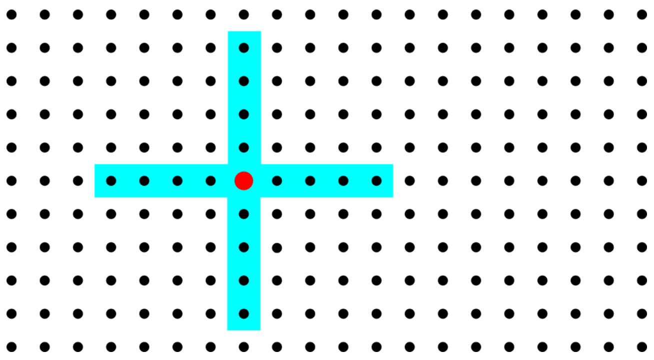


Figure 6. Example of a stencil for the fourth-order Laplacian in a 2D grid. The values from points in the colored region are used to calculate the Laplacian in the central (red) point.

Since stencil calculations are common in scientific and engineering applications, their optimization in CPU and GPU architectures has received considerable interest.[72−78] In our approach, the stencil is applied over several orbitals at once, avoiding some of the performance issues that appear in the application of a stencil to a single data set, in particular, with respect to vectorization.[77]

On the GPU, to perform the application of the Laplacian over a block of orbitals, the threads are arranged in a two-dimensional grid: the first dimension corresponds to the orbital index and the second to the point index. The first task of each group of threads is to find the location of the neighboring points in the input array. Because the grid has an arbitrary shape, this location cannot be easily calculated and we need to use a table of neighbors. (To avoid working with a full table of neighbors, we use a compact form.[79]) Once the neighbor addresses are obtained, each thread iterates over the stencil position, loading the neighbor value, multiplying it by the corresponding weight, and accumulating the result.

Memory access is usually the limiting factor for the performance of the finite-difference operators,[73] as, for each point, we need to iterate over the stencil loading values that are only used for one multiplication and one addition. As the values of the neighbors are scattered, memory access is not regular. This part of the problem is addressed by using blocks of orbitals: since the Laplacian is calculated over a group of orbitals at a time, for each point of the stencil, we load several values, one per orbital in the block, that are contiguous in memory. This makes memory access more regular and hence more efficient for both GPUs and CPUs.

Still, a potential problem with memory access persists. As each input value of the stencil has to be loaded several times, ideally, it should be loaded from main memory once and kept in the cache for subsequent uses. Unfortunately, as the stencil operation has poor memory locality, this is not always the case.

We devised an approach to improve cache utilization by controlling how grid points are ordered in memory, that is, how the three-dimensional grid is mapped to a linear array. The standard approach is to use a row-major or column-major order that leads to some neighboring points being allocated in distant memory locations. Our approach is to enumerate the grid points based on a sequence of small parallelepipedic grids, as shown in the example of Figure 7. This approach permits close

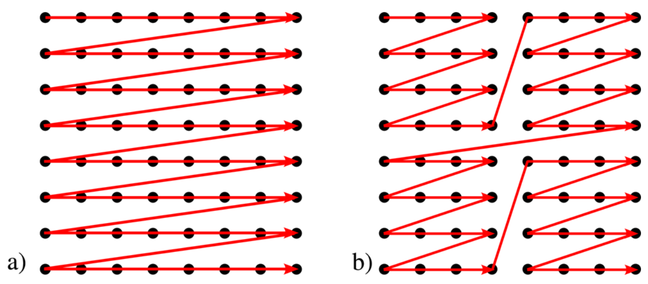


Figure 7. Examples of different grid orders in 2D: (a) standard order and (b) grid ordered by small parallelepipedic blocks.

spatial regions to be stored closer in memory, improving memory locality for the Laplacian operator. The effect of this optimization can be seen in Figure 8, where we compare the throughput of the Laplacian operator, as a function of the block size, for the optimized grid order with respect to the standard one. For the CPU with the standard ordering of points, there is only a small gain performance from using blocks of orbitals, while, by optimizing the grid order, the parallelism exposed by a larger block size allows a considerable performance gain. For the GPU, the effect of the optimization is less dramatic, but still significant.

An area for further improvement is that the optimal size of the parallelepipedic subgrids depends on the processor and the shape and size of the grid, which change for each molecule. Because it is not practical to optimize these parameters for each case, we use a fixed set that does not always yield the best possible performance. This can be seen in Figure 9, where we show a comparison of the numerical throughput of the GPU and CPU implementations of the Laplacian operator for a β-cyclodextrin molecule: the performance obtained is not as high as that in Figure 8. We plan to study the applicability of more sophisticated space-filling curves[80] to address this issue.

It is clear from Figure 9 that, for all processors, the use of blocks of KS states represents a significant numerical performance gain with respect to working with one state at a

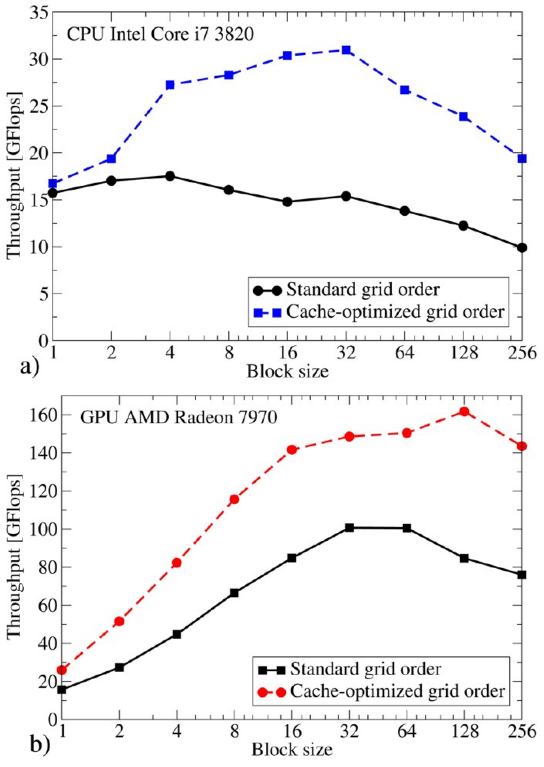


Figure 8. Effect of the optimization of the grid mapping for data locality in the numerical throughput of the Laplacian operator as a function of the size of the orbital block. Spherical grid with 500k points. (a) Computations with an Intel Core i7 3820 (8 threads). (b) Computations with an AMD Radeon 7970.

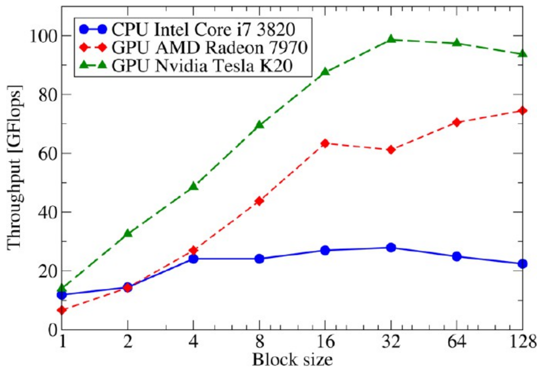


Figure 9. Numerical throughput of the calculation of the finite-difference fourth-order Laplacian as a function of the size of the block of orbitals (block size) for different processors. Calculation for β-cyclodextrin with 256 orbitals and 260k grid points.

time. This is particularly important for GPUs, where performance with a single state is similar to the CPU, but it is more than 5 times larger with blocks of size 32 or 64.

5.2. Local Potential.

The second term of the Hamiltonian is the local potential, which includes contributions from the external potential, including the local parts of the pseudopotentials and the Hartree, exchange, and correlation potentials. All of these terms are summed into a single potential, so we only need to multiply each orbital by this potential and store the result.

Because there are only two arithmetic operations per element, the application of the local potential is heavily limited by memory access. Using blocks of orbitals has two beneficial effects: the larger number of simultaneous operations can hide the memory latency, and the values of the potential are reused, reducing the number of memory accesses. In Figure 10, we compare the numerical performance of the application of the local potential for different processors. As expected, the GPU has a considerable performance advantage caused by the higher memory bandwidth. Still, the numerical throughput is significantly below the values we obtain for other parts of the calculation.

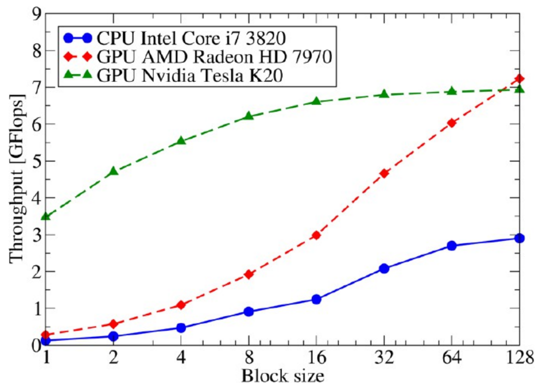


Figure 10. Numerical throughput of the application of the local potential as a function of size of the block of orbitals (block size) for different processors. Calculation for β-cyclodextrin with 256 orbitals and 260k grid points.

5.3. Nonlocal Potential. The final term required for the application of the Hamiltonian is the nonlocal potential that comes from the norm-conserving pseudo-potentials.[81] The nonlocality comes from the fact that each angular momentum component of the orbital sees a different potential. In practice, we calculate

$$V_{\mathrm{nl}}\varphi_k(\boldsymbol{r}) = \sum_A \sum_{lm} \gamma^A_{lm}(\boldsymbol{r} - \boldsymbol{R}) \int_{\boldsymbol{r}'<r_c} \mathrm{d}\boldsymbol{r}' \gamma^A_{lm}(\boldsymbol{r}' - \boldsymbol{R}_A)\varphi_k(\boldsymbol{r}') \tag{4}$$

where γ^A_{lm} corresponds to the pseudo-potential projectors for atom A, and l and m are the angular momentum components that go from 0 to a certain $l_{\max}$, usually 3, and from $-l$ to l, respectively. The projector functions are localized over a sphere, such that $\gamma^A_{lm}(\boldsymbol{r}) = 0$ for $|\boldsymbol{r}| > r_c$.

In our implementation, eq 4 is calculated in two parts that are parallelized differently on the GPU. The first part is to calculate the integrals over $\boldsymbol{r}'$ and store the results. This calculation is parallelized for a block of orbitals, angular-momentum components, and all atoms, with each GPU thread calculating an integral.

The second part of the application of the nonlocal potential is to multiply the stored integrals by the radial functions and sum over angular-momentum components. In this case, the calculation can be parallelized over orbitals, and, if the pseudo-potential spheres associated with each atom do not overlap, it can also be parallelized over the $\boldsymbol{r}$-index and atoms. Usually, the spheres do not overlap, but if they do, race conditions would appear as several threads would try to update the same point. To do the calculations in parallel, we divide the atoms in groups whose spheres do not overlap. We then parallelize over all atoms in each group. In Figure 11, we show an example of the division of atoms for the C_{60} molecule in nonoverlapping groups.

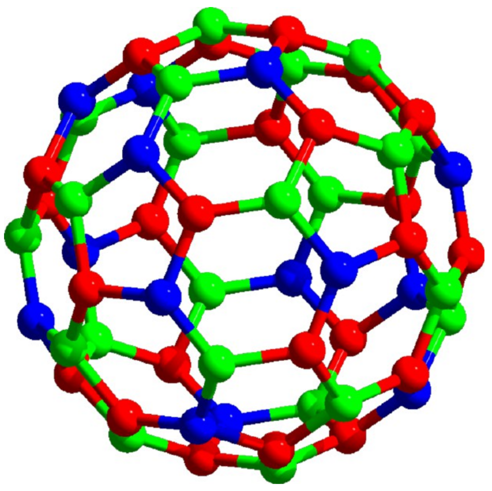


Figure 11. Division of the atoms of a C_{60} molecule in groups (represented by different colors) whose pseudo-potential spheres do not overlap.

In Figure 12, we plot the throughput obtained by the nonlocal potential implementation for a β-cyclodextrin molecule. The Nvidia card shows a good performance, 46 GFlops, only when large blocks of orbitals are used. The AMD card has a similar behavior, but the performance is much lower, with a maximum of 11 GFlops. This is a clear example of how our approach is an effective way of increasing the performance that can be obtained from the GPU. As this is a complex routine, and our current implementation is very basic, we suspect that a more sophisticated and optimized version could significantly increase the numerical performance of this part of the application of the KS Hamiltonian, in particular, for the AMD GPU.

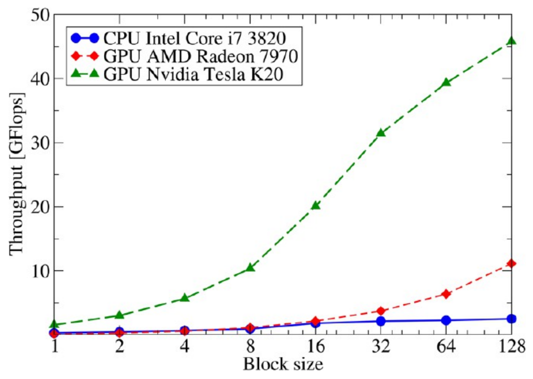


Figure 12. Numerical throughput of the application of the nonlocal potential as a function of the size of the block of orbitals (block size). Calculation for β-cyclodextrin with 256 orbitals and 260k grid points.

6. ORTHOGONALIZATION AND SUBSPACE DIAGONALIZATION

Given a set of orbitals, $\{\varphi_k\}$, the orthogonalization process generates a new set of orthogonal orbitals, $\{\bar{\varphi}_k\}$, as a linear combination of the original ones. Our implementation of the orthogonalization procedure is based on the Cholesky decomposition and other matrix linear algebra operations.[64] For CPUs, BLAS and LAPACK provide an efficient and portable set of routines to perform these operations. For GPUs, we use the OpenCL BLAS implementation provided by AMD as part of the Accelerated Parallel Processing Math Libraries (APPML).

The first step of the orthogonalization is to calculate the overlap between orbitals

$$S_{jk} = \langle \varphi_j | \varphi_k \rangle \tag{5}$$

Our first approximation to this problem was to use the orbitals-block approach to calculate the matrix, S, by dividing it into submatrices, where each submatrix corresponds to the dot product between all the elements of two blocks of orbitals; however, this scheme is not efficient as it reduces the amount of data reuse in the matrix multiplication.[82]

We have found that a much more efficient approach is to first copy the data to an array where all the coefficients corresponding to different orbitals are contiguous in memory, then we can use BLAS to calculate S as a *rank k* operation. To avoid allocating a full copy of all the orbitals, we perform the operation for a set of points at a time. Effectively, we are switching from a block-of-orbitals representation to a block-of-points approach. Once S is calculated, we need to factorize it into a $U^{\dagger}U$ form using a Cholesky decomposition.[83] In our implementation, this operation is done on the CPU using LAPACK. (The MAGMA project[84] implements some of the LAPACK calls on OpenCL, including the Cholesky decomposition and dense matrix diagonalization. We expect to support this library in the future.) However, this is not an issue in our current implementation, since the cost of the decomposition is much smaller than other operations.

From the upper-triangular matrix, U, given by the Cholesky decomposition, we can obtain the new set of orthogonal orbitals from the linear equation

$$\sum_k U_{jk} \bar{\varphi}_k(\boldsymbol{r}) = \varphi_j(\boldsymbol{r}) \tag{6}$$

Because U is triangular, the solution of the linear problem is a simple operation that is done by BLAS. As this procedure mixes all states, we cannot use the blocks-of-orbitals approach; instead, we switch again to the blocks-of-points representation.

In Figure 13, we show the performance obtained for our implementation of the orthogonalization procedure. The GPU speed-up is not very large with respect to the CPU. As this operation is based on linear algebra operations, we attribute the poor speed-up to the difference in the linear algebra libraries. Whereas, for CPUs, BLAS implementations are quite mature,

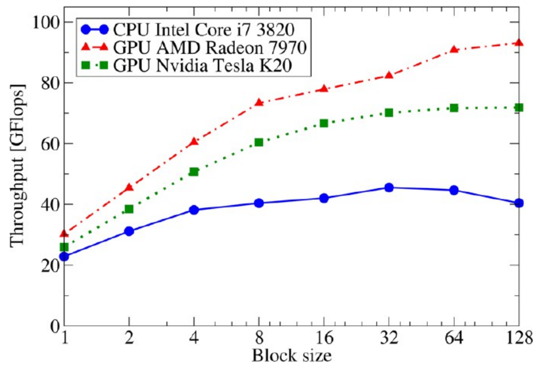


Figure 13. Numerical throughput of the orthogonalization procedure as a function of the size of the block of orbitals (block size) for different processors. Calculation for β-cyclodextrin with 256 orbitals and 260k grid points.

the implementation of linear algebra operations on a GPU is still a field of active study, in particular, for the solution of triangular systems,[85] like eq 6.

The procedure for subspace diagonalization is very similar in form to the orthogonalization. It is used by diagonalization algorithms for sparse matrices to resolve between eigenvectors that have close eigenvalues. The first step in subspace diagonalization is to generate the representation of the Hamiltonian in the subspace of the approximated orbitals, $\{\varphi_k\}$

$$h_{jk} = \langle \varphi_j | H | \varphi_k \rangle \tag{7}$$

As in the case of the matrix S for the orthogonalization, we perform this operation by blocks of points. This time, we need to apply the Hamiltonian to the orbitals first, and then calculate the dot products as a matrix multiplication.

Once the subspace Hamiltonian is calculated, it is diagonalized to obtain the matrix of its eigenvectors, ξ_{jk}. As in the case of the Cholesky decomposition, this dense-matrix diagonalization is done by the CPU. This is not a performance issue for the systems studied in this article, but for larger systems, the dense eigensolver, which scales as $O(n^3)$, could consume a considerable part of the computation time.

Once the subspace Hamiltonian is diagonalized, the new set of orbitals, $\{\bar{\varphi}_k\}$, is generated by rotating the old set by the eigenvector matrix

$$\bar{\varphi}_k(\boldsymbol{r}) = \sum_j \xi_{jk} \varphi_j(\boldsymbol{r}) \tag{8}$$

Because this rotation mixes all orbitals, we follow a similar procedure as we do in eq 6 for the orthogonalization. The only difference is that, in this case, we directly multiply by the matrix instead of its inverse.

Figure 14 shows the performance obtained for the subspace diagonalization. In this case, the GPU speed-up is larger than

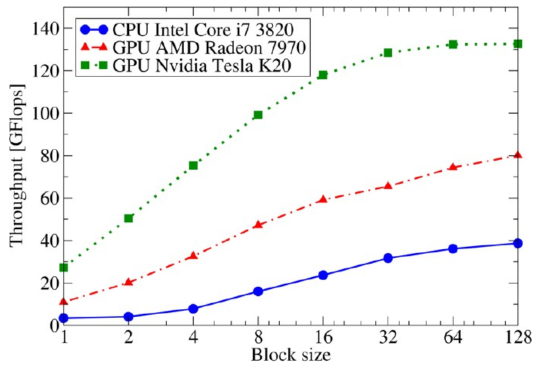


Figure 14. Numerical throughput of the subspace-diagonalization procedure as a function of the size of the block of orbitals (block size) for different processors. Calculation for β-cyclodextrin with 256 orbitals and 260k grid points.

that for the orthogonalization case, probably because this routine is based on our implementation of the KS Hamiltonian, and on matrix–matrix multiplications, that, in general, are simpler to optimize and parallelize than other linear algebra operations.

7. THE HARTREE POTENTIAL

The other operation that we execute on the GPU is the calculation of the Hartree potential by solving the Poisson problem (eq 3). This equation also appears in other contexts in electronic structure simulations, for example, in the calculation of approximations to the exchange term,[86] in the calculation of integrals that appear in Hartree−Fock or Casida theories,[87] or to impose electrostatic boundary conditions.[88−90]

The Poisson equation can be solved by different methods in linear or quasi-linear time.[91−94] In our GPU implementation, we use an approach based on fast Fourier transforms (FFTs), as it is quite efficient and simple to implement. By using FFTs, in principle, we are imposing periodic boundary conditions to the electrostatic potential. We can, however, find the free-boundary solution by enlarging the FFT grid and using a modified interaction kernel.[95]

The solution process involves several steps. The first one is to copy the density from the arbitrarily shaped grid to a cubic grid, where we perform the forward FFT. The result is the density in Fourier space, which is multiplied by the Coulomb-interaction kernel. After an inverse FFT, we obtain the Hartree potential, which is copied back to the arbitrarily shaped grid. Because we only need to solve a single Poisson equation, independently of the size of the system, we cannot use the block approach in this case. The essential component of this solver is an FFT implementation; for GPUs, we use the clAMDFft library provided by AMD. For CPUs, we use the multithreaded FFTW library.[96]

In Figure 15, we show the performance of our GPU-based Poisson solver for different system sizes. For the AMD card, the GPU version outperforms the CPU version, in some cases, by a factor of 7. For the Nvidia GPU, the speed-up is smaller, probably because the library has not been explicitly optimized for this GPU. The step structure seen on the plot is caused by the fact that FFTs cannot be performed efficiently over grids of any size: the grid dimension in each direction must be a product of certain values, or radices, that are determined by the implementation. If a grid dimension is not valid, the size of the grid is increased. Because the CPU implementation is more mature and supports more radices, the steps are smaller than the GPU implementation that only supports radices 2 and 3. (The clAMDFft library also supports radix-5, but we could not use it due to execution and performance issues.) Therefore, it is reasonable to expect that, as the GPU-accelerated FFT implementations improve, the numerical performance of the calculation of the Hartree potential will increase.

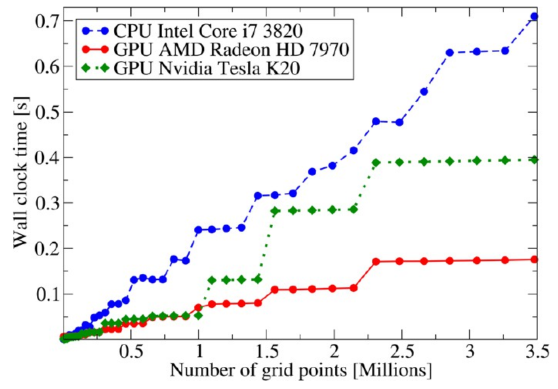


Figure 15. Comparison of the computational time required for solving the Poisson equation using FFTs as a function of the number of grid points. The data are originally on the main memory, so the time required to copy the input data to the GPU and copy back the result is included. The number of points corresponds to the grid used by Octopus; the FFT grid has a larger number of points.

8. OTHER OPERATIONS

In the previous sections, we have described the main operations that we have implemented on the GPU. There are several simpler operations that also need to be performed on the GPU. These operations include basic operations between orbitals, such as copies, linear combinations, and dot products. All of them are implemented on the GPU using the block-of-orbitals approach to improve performance. In fact, we have found that it is necessary to pay attention to the parallelization of most of the operations performed on the GPU, as a single routine that is not properly parallelized can spoil the numerical performance of the entire code.

In our current implementation, there are two procedures that are still done by the CPU, as they would require a considerable effort to implement on the GPU, but have a minor impact in numerical performance. The first one is the evaluation of the XC potential. This is a local operation that is straightforward to parallelize and should perform well on the GPU. The problem is that there is large number of XC approximations, each one involving complex formulas[97] that would need to be implemented on the GPU. The second procedure that is executed on the CPU is the initialization of the molecular orbitals by a linear combination of the atomic orbitals obtained from the pseudo-potentials. The reason is that we use a spline interpolation to transfer the orbitals to the grid, which depends on the GSL library[98] that is not available on the GPU.

9. ACCURACY

The strategy presented in this article does not imply a reduction in the precision of the calculations with respect to the original real-space DFT implementation. There are, however, some factors that could produce some numerical differences in the results.

In a sparse eigensolver, usually the eigenvectors are only converged until their error goes below a certain threshold, to avoid wasting computational time in overconverging some eigenvectors. In our implementation, iterations are only stopped when a whole block of eigenvectors is below the threshold. This makes the code simpler and avoids thread divergence but introduces a dependency of the results on the block size.

Another source of differences in the results is the calculation of the Hartree potential. Because the number of prime factors supported by the GPU FFT library is smaller than the CPU implementation, the size of the FFT grid can be larger for the GPU. However, as, in both cases, the grid is large enough to eliminate periodicity effects, the change in the results due to this difference is minimum.

Finally, there might be some differences in the numerical operations. Although we use double precision for all operations and both GPUs used for the tests are IEEE-754 compliant, there might be differences in the finite precision arithmetic from fused multiply addition (FMA) operations, which are not available in the tested CPU, and due to different ordering of operations.

In our tests with different molecules, we observe that the difference in the total energy between CPU and GPU calculations is on average 0.1 millihartree with a maximum of 0.5 millihartree . This difference is caused mainly by the different sizes of FFT grids used by the CPU and GPU implementations of the Poisson solver . The difference between the energy computed with the Nvidia GPU with respect to the AMD GPU is on average 0.008 millihartree with a maximum of 0.08 millihartree . The variation of the total energy with the block size is well below this values.

10. NUMERICAL PERFORMANCE

In this section, we evaluate the numerical performance of our implementation and how it depends on the size of the blocks of orbitals or the size of the molecular systems. For this analysis, we use several parameters: the throughput, the total calculation time for a single-point energy calculation, the speed-up with respect to the CPU implementation, and the comparison with a second GPU implementation.

10.1. Block Size. We start our performance analysis by studying how the block size influences execution performance. In Figure 16, we plot, for the β-cyclodextrin molecule, both the numerical throughput obtained for the SCF loop and the total execution time as a function of the size of the blocks of orbitals. For the CPU, the optimal block size is 16, with a second local optimum for block size 256. For GPUs, increasing the block size always improves performance up to size 128, which is the limit imposed by the GPU memory. This shows how the block approach produces a significant improvement with respect to working with a single orbital at a time (the block size 1 case).

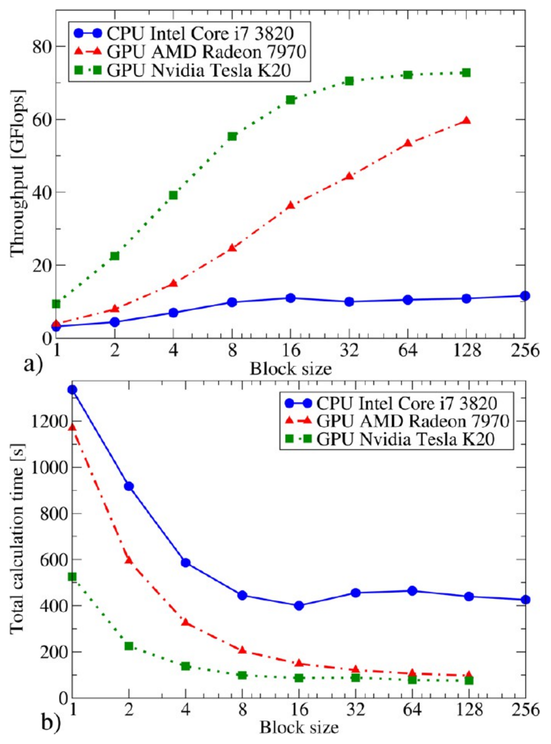


Figure 16. Performance of our CPU and GPU implementations as a function of the size of the block of orbitals (block size). (a) Numerical throughput of the self-consistency cycle. (b) Total execution time for a single-point energy calculation. Simulation for β-cyclodextrin with 256 orbitals and 260k grid points.

10.2. Molecule Size. We now focus our attention on how our GPU implementation performs for molecules of different sizes. For this test, we have selected a set of 40 molecules, listed in Table 1. In this respect, we would like to assert that we did *not* select the set of molecules based on any performance-related criterion, we just aimed to have a set of molecules composed mainly of first- and second-row elements with different numbers of valence electrons and that could fit in the memory of our GPUs.

In Figure 17, we show, for the molecules in our set, the performance measured as throughput of the SCF cycle and total computational time as a function of the number of electrons. As expected, the computational time tends to increase with the number of electrons, but there is a strong variation from system to system. This variation is mainly explained by the physical size of each molecule, which determines the size of the grid that is used in the simulation. The number of iterations required for eigensolver and self-consistency convergence can also change from one system to the other, affecting the total calculation time. From Figure 17a, it is clear that, as the size of the system increases, the GPU becomes more efficient, with a maximum throughput of 90 GFlops for the largest molecule tested, C_{180}.

We now measure the speed-up of the GPUs with respect to the CPU version. In Figure 18a, we plot the speed-up measured using the total computational time. The maximum value we get is 5.2× for the Nvidia GPU and 4.2× for the AMD GPU. If we only consider the time spent in the SCF cycle and ignore the initialization time, the speed-up is 8.0× for the Nvidia GPU and 5.2× for the AMD GPU; the curve also becomes more regular, hinting that much of the variation in the computational time for systems with similar numbers of electrons comes from initialization routines.

While the speed-ups are not as large as some that have been reported in the literature, there are several factors to consider when analyzing GPU speed-ups. First of all, the maximum speed-up we could obtain is given by the peak-performance ratio between the GPU and the CPU, which is approximately 8× for the AMD GPU and 10× for the Nvidia card. If performance is limited by the memory bandwidth, then the maximum speed-up is reduced to 5× (AMD) or 6× (Nvidia). The CPU code taken as reference is also important. In this case, we are comparing code that uses the similar optimization strategies on the CPU and the GPU, and in both the cases, it has been parallelized to use all the execution units available on each processor. This is not the case, for example, when a full GPU is compared against a single core of a CPU.

10.3. Comparison with TeraChem. To make an exhaustive evaluation of the performance of our approach, we compare it with another GPU-accelerated DFT implementation, the TeraChem code.[5,100] TeraChem uses Gaussian-type orbitals (GTOs) as a basis for the expansion of the molecular orbitals: the traditional approach used in quantum chemistry. TeraChem has been extended to perform different types of simulations, such as excited states[101] or ab initio molecular dynamics,[102] and thanks to the computational power offered by GPUs, it has been used to study challenging systems, such as large proteins.[103]

Since Octopus and TeraChem use very different simulation techniques, we take great care in making a significant comparison. The main issue is to select discretization parameters that produce a similar level of approximation. We take as reference the caffeine molecule, $C_8H_{10}N_4O_2$, in the Becke−Lee−Yang−Parr (BLYP) XC approximation.[104] In TeraChem, we select the 6-311g* basis that has an error in the total energy of 5 millihartree per atom, with respect to a calculation with the *aug-cc-pvqz* basis. We then look for grid

Table 1. List of Systems Used for the Performance Studies Done in This Article[a]

system		calculation size		single-point calculation time [s]			
stoichiometry	description	electrons	points [1/1000]	CPU	AMD	Nvidia	TeraChem
C_6H_6	benzene	30	37.3	4.6	13.0	5.9	2.0
$C_{10}H_{18}$	*cis*-decalin	58	62.5	12.8	16.9	10.3	6.8
$C_{14}H_{10}$	anthracene	66	63.0	16.3	15.7	9.2	7.6
$C_8H_{10}N_4O_2$	caffeine	74	63.1	15.9	16.1	10.0	8.7
$C_{16}H_{24}O_2$	palmitoyl	100	93.2	31.3	22.8	15.7	15.1
$C_{18}H_{24}$	*cis*-retinal	102	96.5	31.0	22.7	15.6	17.4
$(H_2O)_{13}$	water cluster	104	83.7	27.7	20.9	13.5	8.2
$C_{20}H_{24}O_2$	ethinyl estradiol	116	99.1	33.6	25.6	17.8	24.1
$C_{18}H_{32}O_2$	linoleic acid	116	122.7	47.5	28.5	20.5	14.4
$C_{22}H_{28}O_2$	etonogestrel	128	107.3	43.1	27.3	19.5	30.0
$C_{26}H_{16}O_3S$	molecule from CEP[b]	142	110.0	62.7	29.3	21.1	24.6
$C_{29}H_{20}N_2$	molecule from CEP[b]	146	119.9	69.5	34.5	25.0	30.4
$C_{34}H_{22}$	diphenylpentacene	158	131.2	99.0	34.1	24.8	33.8
$C_{22}H_{30}N_6O_4S$	sildenafil citrate	178	137.8	90.6	45.3	36.6	41.3
$CH_4(H_2O)_{24}$	methane + water	200	132.6	88.2	35.5	26.4	27.8
$C_{40}H_{52}$	carotene	212	206.0	163.9	65.8	56.2	42.1
$C_{48}H_{24}$	kekulene	216	147.6	97.0	41.9	27.2	49.2
$C_{44}H_{54}Si_2$	TIPS-pentacene	238	182.8	158.6	59.6	55.5	68.6
C_{60}	fullerene	240	102.4	66.6	27.0	18.9	76.6
C_{70}	fullerene	280	113.1	97.2	38.2	24.9	128.3
$C_{51}H_{33}N_5O_3$	porphyrin	280	209.8	196.7	79.5	61.8	105.4
$C_{58}H_{32}S_3$	molecule from CEP[b]	282	214.9	192.0	64.8	49.5	82.0
$C_{41}H_{40}N_8O_8$	carbazole complex	292	192.3	203.5	80.4	64.7	126.2
$C_{60}H_{32}S_4$	DAT−thiophane dimer	296	213.7	211.6	70.4	54.5	78.9
$C_{42}H_{83}NO_8P$	phosphatidylcholine	308	283.9	340.8	109.7	96.4	95.7
$C_{45}H_{51}NO_{15}$	taxol	324	219.0	222.6	71.2	57.6	141.1
$C_{50}H_{238}MgN_4O_5$	chlorophyll	340	269.8	303.9	107.8	94.9	174.5
$C_{58}H_{48}N_8O_{12}$	methotrexate complex	376	238.1	337.4	110.1	94.3	135.4
$C_{36}H_{60}O_{30}$	α-cyclodextrin	384	222.9	354.9	86.6	69.2	89.6
C_{100}	fullerene	400	160.8	272.2	64.9	53.1	194.2
$C_{60}(H_2O)_{20}$	fullerene + water	400	200.5	319.2	83.7	66.3	225.2
$C_{54}H_{90}N_6O_{18}$	valinomycin	444	293.6	494.3	152.5	124.5	185.6
$C_{42}H_{70}O_{35}$	β-cyclodextrin	448	259.5	401.0	99.5	81.6	100.6
$C_{62}H_{63}N_{15}O_{12}$	methotrexate complex	458	265.4	461.6	163.1	145.0	331.1
$C_{122}H_4$	fullerene dimer	492	198.3	331.4	87.2	69.6	344.0
$C_{114}H_{48}$	graphite cluster	504	277.8	684.1	162.6	132.5	672.4
$C_{48}H_{80}O_{40}$	γ-cyclodextrin	512	290.9	543.3	216.8	109.5	131.0
$C_{68}H_{76}N_{13}O_{16}P$	cAMP complex	514	327.4	734.6	273.0	206.7	334.1
$C_{68}H_{318}Na_2O_{20}P_2$	phospholipid	540	404.7	925.8	291.3	177.8	808.8
C_{180}	fullerene	720	267.8	699.2	188.9	141.8	461.1

[a]For each molecule, we include the number of valence electrons; the number of grid points used in the simulation; the computational times for Octopus with three different processors, CPU Intel Core i7 3820 (CPU), GPU AMD Radeon HD 7970 (AMD), and GPU Nvidia Tesla K20 (Nvidia); and the calculation time for TeraChem with the Nvidia Tesla K20 GPU (TeraChem). The geometry for each molecule can be found in the Supporting Information. [b]These molecules were obtained from the Harvard Clean Energy Project (CEP).[99]

parameters that give a similar error, this time taking as reference the converged real-space result. The selected grid is a union of spheres of radius 5.5 bohr around each atom and a spacing of 0.41 bohr. However, the real-space approach has an additional approximation, as it requires pseudo-potentials so that the ionic potential is smooth enough to be represented in a uniform grid. To minimize the effect of this difference in computation time and to compare the actual implementation, we test molecules composed mainly of first- and second-row elements.

In Figure 19, we compare the timings for both codes for the same set of systems used in section 10.2 (Table 1). We show the comparison between absolute times and also the relative performance between the two DFT implementations. We can see that TeraChem tends to be faster for smaller systems, whereas Octopus has an advantage for systems with more than 100 electrons. It is difficult to generalize these results due to the different simulation approaches and their different strengths and weaknesses. For example, our current implementation will certainly be much slower than TeraChem for hybrid HF-DFT XC approximations[105] due to the cost of applying the exact-exchange operator in real-space. However, we can conclude that, for pure DFT calculations, the real-space method can compete with the Gaussian approach, and can outperform it for some systems.

11. CONCLUSIONS

We have presented an approach for the implementation of real-space density functional theory on GPUs. What we have shown

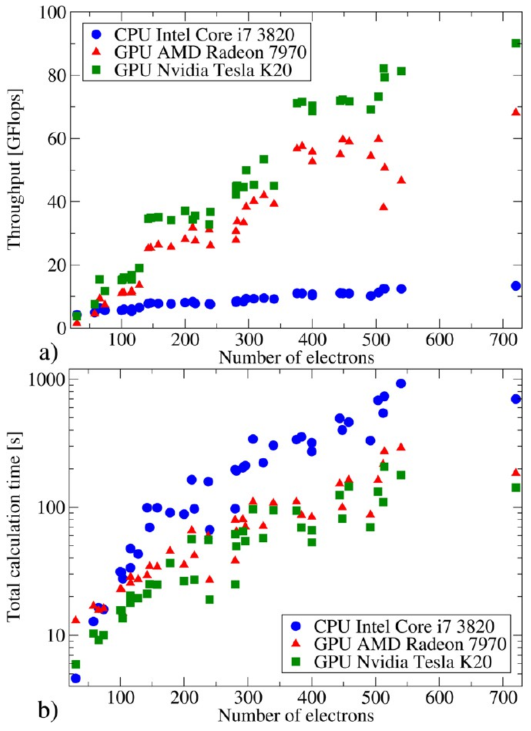


Figure 17. Performance of our CPU and GPU implementations for a set of 40 molecules of different sizes. (a) Numerical throughput of the self-consistency cycle. (b) Total execution time for a single-point energy calculation. The list of molecules and the calculation times are given in Table 1.

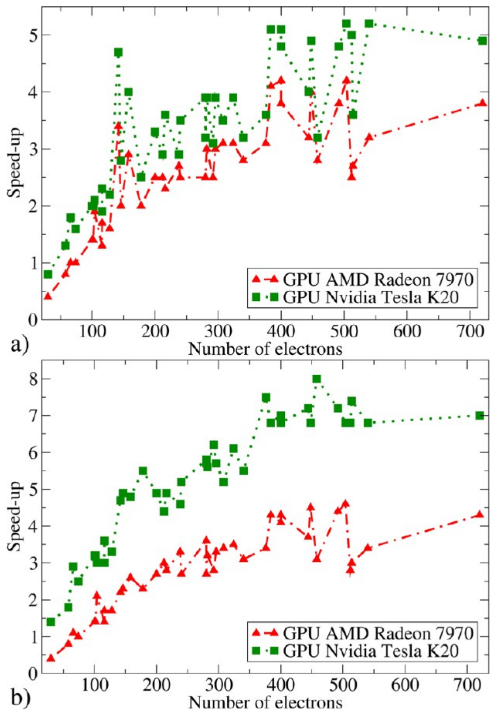


Figure 18. Speed-up of the GPU calculation with respect to the CPU for different molecules as a function of the number of valence electrons. (a) Speed-up calculated from the total calculation time. (b) Speed-up computed from the time spent in the SCF cycle (without considering initializations). The reference CPU is an Intel Core i7 3820 using eight threads.

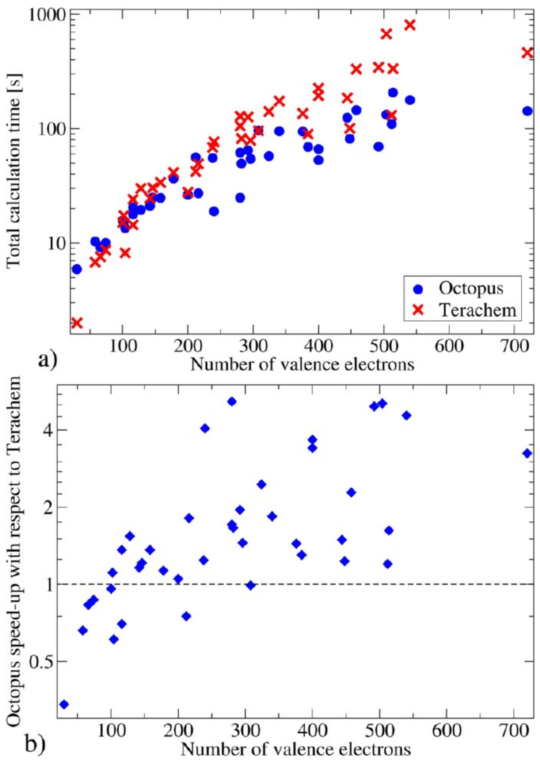


Figure 19. Numerical performance comparison between our GPU implementation (Octopus) and the TeraChem code. (a) Comparison of the total calculation time as a function of the number of valence electrons. (b) Speed-up of our implementation with respect to TeraChem (run time of TeraChem divided by the run time of Octopus). The calculations are single-point energy evaluations performed on a set of 40 molecules, running on a Nvidia Tesla K20 GPU. The list of molecules and the calculation times are given in Table 1.

is much more than a reimplementation of the code in GPU language, but a scheme designed to perform DFT calculations efficiently on massively parallel processors.

Our approach is based on using blocks of KS orbitals as the basic data object. This provides the GPU with enough data to perform efficiently, something that would be harder to achieve by working on single orbitals at a time. However, this approach is not applicable or does not work efficiently for all operations, so in other cases, a block-of-points strategy is used. Many of these techniques are applicable to other DFT discretization approaches, especially those based on sparse representations, such as plane waves or wavelets.

The efficiency of our approach is analyzed by examining several parameters. We achieve a considerable throughput and speed-up with respect to the CPU version of Octopus. More importantly, in comparison to a GPU-accelerated implementation of DFT based on Gaussian basis sets, we find that calculation times are similar, with our code being faster for several of the systems that were tested. This is not to be taken lightly, as the GTO approach has been designed and constantly improved with the specific purpose of efficiently modeling molecular systems. The real-space method, on the other hand, is a more general approach used to study different types of partial differential equations.

We can conclude that the real-space formulation provides a good framework for the implementation of DFT on GPUs, making real-space DFT an interesting alternative for electronic structure calculations, as it offers good performance, systematic control of the discretization, and the flexibility to study many classes of systems, including both periodic and finite systems.

A particular advantage of real-space DFT is its potential for large-scale parallelization in distributed memory systems with tens of thousands of processors.[12,39,40] This is something we want to apply in future work, by exploring the combination of in-processor (OpenCL) and distributed memory (MPI) parallelization for DFT calculations on GPU-based supercomputers.

12. COMPUTATIONAL METHODS

Our numerical implementation is included in the Octopus code,[12,30,41] and it is publicly available under the GPL free-software license.[42] The calculations were performed with the development version (Octopus Superciliosus, svn revision 10562). GPU support is also available in the 4.1 release of Octopus.

Since Octopus is written in Fortran 95, we wrote a wrapper library to call OpenCL from that language. This library is called FortranCL, and it is available as a stand-alone package under a free-software license.[106]

All calculations were performed using the default pseudo-potentials of Octopus, filtered to remove high-frequency components.[107] The grid for all simulation is a union of spheres of radius 5.5 bohr around each atom with a uniform spacing of 0.41 bohr.

The GTO calculations were done with TeraChem (version v1.5K) with the 6-311g* basis and dftgrid = 1. All other simulation parameters were kept in their default values. For all calculations, we used the BLYP XC functional.[104]

The system used for the tests has an Intel Core i7 3820 CPU, which has four cores running at 3.6 GHz that can execute two threads each. The CPU has a quad-channel memory subsystem with 16 GiB of RAM running at 1600 MHz. The GPUs are an AMD Radeon HD 7970 with 3 GiB of RAM and a Nvidia Tesla K20c with 5 GiB (ECC is disabled, as the other processors do not support ECC). Both GPUs are connected to a PCIe 16x slot; the AMD card supports the PCIe 3 protocol, whereas the Nvidia card is limited to PCIe 2. Octopus was compiled with the GNU compiler (gcc and gFortran, version 4.7.2) with AVX vectorization enabled. For finite-difference operations, CPU vectorization is implemented explicitly using compiler directives. We use the Intel MKL (version 10.3.6) implementation of BLAS and LAPACK that is optimized for AVX. We use the OpenCL implementation from the respective GPU vendor: the AMD OpenCL version is 1084.4 (VM), and the Nvidia one is 310.32 (OpenCL is not used for the CPU calculations). All tests are executed with eight OpenMP threads.

Total and partial execution times were measured using the gettimeofday call. The throughput is defined as the number of floating point additions and multiplications per unit of time. The number of operations for each procedure is counted by inspection of the code. For TeraChem, the total execution time is obtained from the program output.

ASSOCIATED CONTENT

Ⓢ Supporting Information

Structures in xyz format for all the molecules in the test set. The geometries are for benchmarking purposes; they do not necessarily represent the experimental or theoretical structure of the system. This material is available free of charge via the Internet at http://pubs.acs.org.

AUTHOR INFORMATION

Corresponding Author

*E-mail: aspuru@chemistry.harvard.edu.

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

We thank N. Suberviola, J. Muguerza, and A. Arruabarrena for useful discussions, and D. Strubbe, J. Alberdi, M. A. L. Marques, and all the Octopus development team for their effort in developing and maintaining the code (in particular, for detecting and fixing many of the bugs introduced while implementing the GPU support). We would like to acknowledge Nvidia for support via the Harvard CUDA Center of Excellence, and both Nvidia and Advanced Micro Devices (AMD) for providing the GPUs used in this work. This work was supported by the Defense Threat Reduction Agency under Contract No HDTRA1-10-1-0046.

REFERENCES

(1) Moore, G. E. *Electronics* **1965**, *38*, 4.
(2) (a) Dennard, R.; Gaensslen, F.; Rideout, V.; Bassous, E.; LeBlanc, A. *IEEE J. Solid-State Circuits* **1974**, *9*, 256−268. (b) Bohr, M. The new era of scaling in an SoC world. In *Proceedings of the IEEE International Solid-State Circuits Conference: ISSCC 2009*, San Francisco, CA, Feb 8−12, 2009.
(3) Harju, A.; Siro, T.; Canova, F.; Hakala, S.; Rantalaiho, T. In *Applied Parallel and Scientific Computing*; Manninen, P., Öster, P., Eds.; Lecture Notes in Computer Science; Springer: Berlin, 2013; Vol. 7782, pp 3−26.
(4) (a) Hohenberg, P.; Kohn, W. *Phys. Rev.* **1964**, *136*, B864−B871. (b) Kohn, W.; Sham, L. *J. Phys. Rev.* **1965**, *140*, A1133−A1138.
(5) Ufimtsev, I.; Martínez, T. *Comput. Sci. Eng.* **2008**, *10*, 26−34.
(6) Yasuda, K. *J. Chem. Theory Comput.* **2008**, *4*, 1230−1236.
(7) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. *J. Phys. Chem. A* **2008**, *112*, 2049−2057.
(8) Genovese, L.; Ospici, M.; Deutsch, T.; Méhaut, J.-F.; Neelov, A.; Goedecker, S. *J. Chem. Phys.* **2009**, *131*, 034103.
(9) Watson, M.; Olivares-Amaya, R.; Edgar, R. G.; Aspuru-Guzik, A. *Comput. Sci. Eng.* **2010**, *12*, 40−51.
(10) Tomono, H.; Aoki, M.; Iitaka, T.; Tsumuraya, K. *J. Phys.: Conf. Ser.* **2010**, *215*, 012121.
(11) Andrade, X.; Genovese, L. In *Fundamentals of Time-Dependent Density Functional Theory*; Marques, M. A., Maitra, N. T., Nogueira, F. M., Gross, E., Rubio, A., Eds.; Lecture Notes in Physics; Springer: Berlin, 2012; Vol. *837*, pp 401−413.
(12) Andrade, X.; Alberdi-Rodriguez, J.; Strubbe, D. A.; Oliveira, M. J.; Nogueira, F.; Castro, A.; Muguerza, J.; Arruabarrena, A.; Louie, S. G.; Aspuru- Guzik, A.; Rubio, A.; Marques, M. A. L. *J. Phys.: Condens. Matter* **2012**, *24*, 233202.
(13) Maintz, S.; Eck, B.; Dronskowski, R. *Comput. Phys. Commun.* **2011**, *182*, 1421−1427.
(14) DePrince, A. E.; Hammond, J. R. *J. Chem. Theory Comput.* **2011**, *7*, 1287−1295.
(15) Spiga, F.; Girotto, I. phiGEMM: A CPU-GPU Library for Porting Quantum ESPRESSO on Hybrid Systems. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Garching, Germany, Feb 15−17, 2012; Stotzka, R., Schiffers, M., Cotronis, Y., Eds.; The Institute of Electrical and Electronics Engineers, Inc.: New York, 2012.
(16) Maia, J. D. C.; Urquiza Carvalho, G. A.; Mangueira, C. P.; Santana, S. R.; Cabral, L. A. F.; Rocha, G. B. *J. Chem. Theory Comput.* **2012**, *8*, 3072−3081.
(17) Hacene, M.; Anciaux-Sedrakian, A.; Rozanska, X.; Klahr, D.; Guignon, T.; Fleurat-Lessard, P. *J. Comput. Chem.* **2012**, *33*, 2581−2589.

(18) Esler, K.; Kim, J.; Ceperley, D. M.; Shulenburger, L. *Comput. Sci. Eng.* **2012**, *14*, 40−51.
(19) Hakala, S.; Havu, V.; Enkovaara, J.; Nieminen, R. In *Applied Parallel and Scientific Computing*; Manninen, P., Öster, P., Eds.; Lecture Notes in Computer Science; Springer: Berlin, 2013; Vol. *7782*, pp 63−76.
(20) (a) Jia, W.; Cao, Z.; Wang, L.; Fu, J.; Chi, X.; Gao, W.; Wang, L.-W. *Comput. Phys. Commun.* **2013**, *184*, 9−18. (b) Jia, W.; Fu, J.; Cao, Z.; Wang, L.; Chi, X.; Gao, W.; Wang, L.-W. *J. Comput. Phys.* **2013**, *251*, 102−115.
(21) Hutter, J.; Iannuzzi, M.; Schiffmann, F.; VandeVondele, J. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2013**, DOI: 10.1002/wcms.1159.
(22) Bhaskaran-Nair, K.; Ma, W.; Krishnamoorthy, S.; Villa, O.; van Dam, H. J. J.; Aprà, E.; Kowalski, K. *J. Chem. Theory Comput.* **2013**, *9*, 1949−1957.
(23) Titov, A. V.; Ufimtsev, I. S.; Luehr, N.; Martínez, T. J. *J. Chem. Theory Comput.* **2013**, *9*, 213−221.
(24) Becke, A. D. *Int. J. Quantum Chem.* **1989**, *36*, 599−609.
(25) Chelikowsky, J. R.; Troullier, N.; Saad, Y. *Phys. Rev. Lett.* **1994**, *72*, 1240−1243.
(26) Briggs, E. L.; Sullivan, D. J.; Bernholc, J. *Phys. Rev. B* **1995**, *52*, R5471−R5474.
(27) Fattebert, J.-L.; Bernholc, J. *Phys. Rev. B* **2000**, *62*, 1713−1722.
(28) Fattebert, J.-L.; Nardelli, M. B. In *Special Volume: Computational Chemistry*; Bris, C. L., Ed.; Handbook of Numerical Analysis; Elsevier: Amsterdam, 2003; Vol. *10*; pp 571−612.
(29) Beck, T. L. *Rev. Mod. Phys.* **2000**, *72*, 1041−1080.
(30) Marques, M. A.; Castro, A.; Bertsch, G. F.; Rubio, A. *Comput. Phys. Commun.* **2003**, *151*, 60−78.
(31) Torsti, T.; Heiskanen, M.; Puska, M. J.; Nieminen, R. M. *Int. J. Quantum Chem.* **2003**, *91*, 171−176.
(32) Hirose, K. *First-Principles Calculations In Real-Space Formalism: Electronic Configurations and Transport Properties of Nanostructures*; Imperial College Press: London, 2005.
(33) Mortensen, J. J.; Hansen, L. B.; Jacobsen, K. W. *Phys. Rev. B* **2005**, *71*, 035109.
(34) Kronik, L.; Makmal, A.; Tiago, M. L.; Alemany, M. M. G.; Jain, M.; Huang, X.; Saad, Y.; Chelikowsky, J. R. *Phys. Status Solidi B* **2006**, *243*, 1063−1079.
(35) Yabana, K.; Nakatsukasa, T.; Iwata, J.- I.; Bertsch, G. F. *Phys. Status Solidi B* **2006**, *243*, 1121−1138.
(36) Hernández, E. R.; Janecek, S.; Kaczmarski, M.; Krotscheck, E. *Phys. Rev. B* **2007**, *75*, 075108.
(37) Iwata, J.-I.; Takahashi, D.; Oshiyama, A.; Boku, T.; Shiraishi, K.; Okada, S.; Yabana, K. *J. Comput. Phys.* **2010**, *229*, 2339−2363.
(38) Bernholc, J.; Hodak, M.; Lu, W. *J. Phys.: Condens. Matter* **2008**, *20*, 294205.
(39) Enkovaara, J.; Rostgaard, C.; Mortensen, J. J.; Chen, J.; Dulak, M.; Ferrighi, L.; Gavnholt, J.; Glinsvad, C.; Haikola, V.; Hansen, H. A.; Kristoffersen, H. H.; Kuisma, M.; Larsen, A. H.; Lehtovaara, L.; Ljungberg, M.; Lopez- Acevedo, O.; Moses, P. G.; Ojanen, J.; Olsen, T.; Petzold, V.; Romero, N. A.; Stausholm-Møller, J.; Strange, M.; Tritsaris, G. A.; Vanin, M.; Walter, M.; Hammer, B.; Häkkinen, H.; Madsen, G. K. H.; Nieminen, R. M.; Nørskov, J. K.; Puska, M.; Rantala, T. T.; Schiøtz, J.; Thygesen, K. S.; Jacobsen, K. W. *J. Phys.: Condens. Matter* **2010**, *22*, 253202.
(40) Hasegawa, Y.; Iwata, J.-I.; Tsuji, M.; Takahashi, D.; Oshiyama, A.; Minami, K.; Boku, T.; Shoji, F.; Uno, A.; Kurokawa, M.; Inoue, H.; Miyoshi, I.; Yokokawa, M. First-Principles Calculations of Electron States of a Silicon Nanowire with 100,000 Atoms on the K Computer. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seatle, WA, Nov 12−18, 2011; ACM: New York, 2011.
(41) Castro, A.; Appel, H.; Oliveira, M.; Rozzi, C. A.; Andrade, X.; Lorenzen, F.; Marques, M. A. L.; Gross, E. K. U.; Rubio, A. *Phys. Status Solidi B* **2006**, *243*, 2465−2488.
(42) The Octopus source code can be obtained from: http://tddft.org/program/octopus/.
(43) Burnus, T.; Marques, M. A. L.; Gross, E. K. U. *Phys. Rev. A* **2005**, *71*, 010501.
(44) Botti, S.; Castro, A.; Andrade, X.; Rubio, A.; Marques, M. A. L. *Phys. Rev. B* **2008**, *78*, 035333.
(45) Andrade, X.; Castro, A.; Zueco, D.; Alonso, J. L.; Echenique, P.; Falceto, F.; Rubio, A. *J. Chem. Theory Comput.* **2009**, *5*, 728−742.
(46) Räsänen, E.; Pittalis, S.; Proetto, C. R. *J. Chem. Phys.* **2010**, *132*, 044112.
(47) Helbig, N.; Fuks, J. I.; Casula, M.; Verstraete, M. J.; Marques, M. A. L.; Tokatly, I. V.; Rubio, A. *Phys. Rev. A* **2011**, *83*, 032503.
(48) De Giovannini, U.; Varsano, D.; Marques, M. A. L.; Appel, H.; Gross, E. K. U.; Rubio, A. *Phys. Rev. A* **2012**, *85*, 062515.
(49) Elliott, P.; Fuks, J. I.; Rubio, A.; Maitra, N. T. *Phys. Rev. Lett.* **2012**, *109*, 266404.
(50) Andrade, X.; Sanders, J. N.; Aspuru-Guzik, A. *Proc. Natl. Acad. Sci. U.S.A.* **2012**, *109*, 13928−13933.
(51) Wasserman, A.; Maitra, N. T.; Heller, E. J. *Phys. Rev. A* **2008**, *77*, 042503.
(52) Malloci, G.; Mulas, G.; Cecchi-Pestellini, C.; Joblin, C. *Astron. Astrophys.* **2008**, *489*, 1183−1187.
(53) Botti, S.; Castro, A.; Lathiotakis, N. N.; Andrade, X.; Marques, M. A. L. *Phys. Chem. Chem. Phys.* **2009**, *11*, 4523−4527.
(54) Vila, F. D.; Strubbe, D. A.; Takimoto, Y.; Andrade, X.; Rubio, A.; Louie, S. G.; Rehr, J. J. *J. Chem. Phys.* **2010**, *133*, 034111.
(55) Zhang, G. P.; Strubbe, D. A.; Louie, S. G.; George, T. F. *Phys. Rev. A* **2011**, *84*, 023837.
(56) Bonaca, A.; Bilalbegović, G. *Mon. Not. R. Astron. Soc.* **2011**, *416*, 1509−1513.
(57) Avendaño Franco, G.; Piraux, B.; Grüning, M.; Gonze, X. *Theor. Chem. Acc.* **2012**, *131*, 1−10.
(58) Castro, A. *ChemPhysChem* **2013**, *14*, 1488−1495.
(59) Andrea Rozzi, C.; Maria Falke, S.; Spallanzani, N.; Rubio, A.; Molinari, E.; Brida, D.; Maiuri, M.; Cerullo, G.; Schramm, H.; Christoffers, J.; Lienau, C. *Nat. Commun.* **2013**, *4*, 1602.
(60) Räsänen, E.; Heller, E. J. *Eur. Phys. J. B.* **2013**, *86*, 1−5.
(61) Munshi, A., Ed. *The OpenCL Specification*; Khronos Group: Philadelphia, PA, 2009.
(62) Natan, A.; Benjamini, A.; Naveh, D.; Kronik, L.; Tiago, M. L.; Beckman, S. P.; Chelikowsky, J. R. *Phys. Rev. B* **2008**, *78*, 075109.
(63) Wood, D. M.; Zunger, A. *J. Phys. A: Math. Gen.* **1985**, *18*, 1343.
(64) Kresse, G.; Furthmüller, J. *Phys. Rev. B* **1996**, *54*, 11169−11186.
(65) Pulay, P. *Chem. Phys. Lett.* **1980**, *73*, 393−398.
(66) Saad, Y.; Stathopoulos, A.; Chelikowsky, J.; Wu, K.; Ôĝût, S. *BIT Numer. Math.* **1996**, *36*, 563−578.
(67) Broyden, C. G. *Math. Comput.* **1965**, *19*, 577−593.
(68) Srivastava, G. P. *J. Phys. A: Math. Gen.* **1984**, *17*, L317.
(69) (a) Tuckerman, M. E.; Parrinello, M. *J. Chem. Phys.* **1994**, *101*, 1302−1315. (b) Alonso, J. L.; Andrade, X.; Echenique, P.; Falceto, F.; Prada- Gracia, D.; Rubio, A. *Phys. Rev. Lett.* **2008**, *101*, 096403.
(70) (a) Yabana, K.; Bertsch, G. F. *Phys. Rev. B* **1996**, *54*, 4484−4487. (b) Castro, A.; Marques, M. A. L.; Rubio, A. *J. Chem. Phys.* **2004**, *121*, 3425−3433.
(71) (a) Baroni, S.; deGironcoli, S.; Dal Corso, A.; Giannozzi, P. *Rev. Mod. Phys.* **2001**, *73*, 515−562. (b) Andrade, X.; Botti, S.; Marques, M. A. L.; Rubio, A. *J. Chem. Phys.* **2007**, *126*, 184106.
(72) Peng, L.; Seymour, R.; Nomura, K.-i.; Kalia, R. K.; Nakano, A.; Vashishta, P.; Loddoch, A.; Netzband, M.; Volz, W.; Wong, C. High-order stencil computations on multicore clusters. In *Proceedings of the 2009 IEEE International Parallel & Distributed Processing Symposium: IPDPS 2009*, Rome, Italy, May 25−29, 2009.
(73) Datta, K.; Kamil, S.; Williams, S.; Oliker, L.; Shalf, J.; Yelick, K. *SIAM Rev.* **2009**, *51*, 129−159.
(74) Dursun, H.; Nomura, K.-i.; Peng, L.; Seymour, R.; Wang, W.; Kalia, R.; Nakano, A.; Vashishta, P. In *Euro-Par 2009: Parallel Processing*; Sips, H., Epema, D., Lin, H.-X., Eds.; Lecture Notes in Computer Science; Springer: Berlin, 2009; Vol. *5704*, pp 642−653.
(75) Treibig, J.; Wellein, G.; Hager, G. *J. Comput. Sci.* **2011**, *2*, 130−137.

(76) de la Cruz, R.; Araya-Polo, M. *Procedia Comput. Sci.* **2011**, *4*, 2146−2155.
(77) Henretty, T.; Stock, K.; Pouchet, L.-N.; Franchetti, F.; Ramanujam, J.; Sadayappan, P. In *Compiler Construction*; Knoop, J., Ed.; Lecture Notes in Computer Science; Springer: Berlin, 2011; Vol. *6601*, pp 225−245.
(78) Holewinski, J.; Pouchet, L.-N.; Sadayappan, P. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing: ICS '12*, Venice, Italy, June 25−29, 2012; ACM: New York, 2012.
(79) Andrade, X. Linear and Non-Linear Response Phenomena of Molecular Systems within Time-Dependent Density Functional Theory. Ph.D. thesis, University of the Basque Country, UPV/EHU, Donostia, Spain, 2010.
(80) (a) Peano, G. *Math. Ann.* **1890**, *36*, 157−160. (b) Sagan, H. *Space-Filling Curves*; Springer-Verlag: New York, 1994. (c) Günther, F.; Mehl, M.; Pögl, M.; Zenger, C. *SIAM J. Sci. Comput.* **2006**, *28*, 1634−1650.
(81) (a) Kleinman, L.; Bylander, D. M. *Phys. Rev. Lett.* **1982**, *48*, 1425−1428. (b) Troullier, N.; Martins, J. L. *Phys. Rev. B* **1991**, *43*, 1993−2006.
(82) Wadleigh, K.; Crawford, I. *Software Optimization for High-Performance Computers*; HP Professional Books Series; Prentice Hall: Upper Saddle River, NJ, 2000.
(83) Benoit, C. *Bull. Geod.* **1924**, *2*, 67−77.
(84) Agullo, E.; Demmel, J.; Dongarra, J.; Hadri, B.; Kurzak, J.; Langou, J.; Ltaief, H.; Luszczek, P.; Tomov, S. *J. Phys.: Conf. Ser.* **2009**, *180*, 012037.
(85) Ries, F.; De Marco, T.; Guerrieri, R. *IEEE Trans. Parallel Distrib. Syst.* **2012**, *23*, 177−184.
(86) (a) Perdew, J. P.; Zunger, A. *Phys. Rev. B* **1981**, *23*, 5048−5079. (b) Umezawa, N. *Phys. Rev. A* **2006**, *74*, 032505. (c) Andrade, X.; Aspuru-Guzik, A. *Phys. Rev. Lett.* **2011**, *107*, 183002.
(87) Shang, H.; Li, Z.; Yang, J. *J. Phys. Chem. A* **2010**, *114*, 1039−1043.
(88) (a) Tan, I.-H.; Snider, G. L.; Chang, L. D.; Hu, E. L. *J. Appl. Phys.* **1990**, *68*, 4071−4076. (b) Luscombe, J. H.; Bouchard, A. M.; Luban, M. *Phys. Rev. B* **1992**, *46*, 10262−10268.
(89) (a) Klamt, A.; Schuurmann, G. *J. Chem. Soc., Perkin Trans. 2* **1993**, 799−805. (b) Tomasi, J.; Persico, M. *Chem. Rev.* **1994**, *94*, 2027−2094.
(90) (a) Olivares-Amaya, R.; Stopa, M.; Andrade, X.; Watson, M. A.; Aspuru- Guzik, A. *J. Phys. Chem. Lett.* **2011**, *2*, 682−688. (b) Watson, M. A.; Rappoport, D.; Lee, E. M. Y.; Olivares- Amaya, R.; Aspuru-Guzik, A. *J. Chem. Phys.* **2012**, *136*, 024101.
(91) (a) Greengard, L. F.; Rokhlin, V. *Acta Numer.* **1997**, *6*, 229. (b) Kutteh, R.; Apra, E.; Nichols, J. *Chem. Phys. Lett.* **1995**, *238*, 173−179.
(92) (a) Briggs, W. L. *A Multigrid Tutorial*; Wiley: New York, 1987. (b) Beck, T. L. *Int. J. Quantum Chem.* **1997**, *65*, 477−486.
(93) Cerioni, A.; Genovese, L.; Mirone, A.; Sole, V. A. *J. Chem. Phys.* **2012**, *137*, 134108.
(94) Garcia-Risueno, P.; Alberdi-Rodriguez, J.; Oliveira, M. J. T.; Andrade, X.; Pippig, M.; Muguerza, J.; Arruabarrena, A.; Rubio, A. arXiv:1211.2092. [physics.comp-ph], 2012.
(95) Rozzi, C. A.; Varsano, D.; Marini, A.; Gross, E. K. U.; Rubio, A. *Phys. Rev. B* **2006**, *73*, 205119.
(96) Frigo, M.; Johnson, S. G. *Proc. IEEE* **2005**, *93*, 216−231.
(97) Marques, M. A.; Oliveira, M. J.; Burnus, T. *Comput. Phys. Commun.* **2012**, *183*, 2272−2281.
(98) Galassi, M.; Davies, J.; Theiler, J.; Gough, B.; Jungman, G.; Booth, M.; Rossi, F. *GNU Scientific Library: Reference Manual*; Network Theory Ltd., 2003. http://www.gnu.org/software/gsl.
(99) Hachmann, J.; Olivares-Amaya, R.; Atahan-Evrenk, S.; Amador-Bedolla, C.; Sánchez-Carrera, R. S.; Gold-Parker, A.; Vogt, L.; Brockway, A. M.; Aspuru- Guzik, A. *J. Phys. Chem. Lett.* **2011**, *2*, 2241−2251.
(100) (a) Ufimtsev, I. S.; Martínez, T. J. *J. Chem. Theory Comput.* **2008**, *4*, 222−231. (b) Ufimtsev, I. S.; Martínez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 1004−1015. (c) Ufimtsev, I. S.; Martínez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619−2628. (d) Luehr, N.; Ufimtsev, I. S.; Martínez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 949−954.
(101) Isborn, C. M.; Luehr, N.; Ufimtsev, I. S.; Martínez, T. J. *J. Chem. Theory Comput.* **2011**, *7*, 1814−1823.
(102) Ufimtsev, I. S.; Luehr, N.; Martínez, T. J. *J. Phys. Chem. Lett.* **2011**, *2*, 1789−1793.
(103) (a) Kulik, H. J.; Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. *J. Phys. Chem. B* **2012**, *116*, 12501−12509. (b) Isborn, C. M.; Götz, A. W.; Clark, M. A.; Walker, R. C.; Martínez, T. J. *J. Chem. Theory Comput.* **2012**, *8*, 5092−5106.
(104) (a) Becke, A. D. *Phys. Rev. A* **1988**, *38*, 3098−3100. (b) Lee, C.; Yang, W.; Parr, R. G. *Phys. Rev. B* **1988**, *37*, 785−789. (c) Miehlich, B.; Savin, A.; Stoll, H.; Preuss, H. *Chem. Phys. Lett.* **1989**, *157*, 200−206.
(105) Becke, A. D. *J. Chem. Phys.* **1993**, *98*, 1372−1377.
(106) Andrade, X. *FortranCL: A Fortran/OpenCL interface*; 2011. http://fortrancl.googlecode.com.
(107) Tafipolsky, M.; Schmid, R. J. *Chem. Phys.* **2006**, *124*, 174102.