C. R. Mecanique 339 (2011) 149–164

Contents lists available at ScienceDirect

Comptes Rendus Mecanique

www.sciencedirect.com

High Performance Computing / Le Calcul Intensif

Daubechies wavelets for high performance electronic structure calculations: The BigDFT project

Ondelettes de Daubechies pour le calculs de structure élecronique à haute performance: Le code BigDFT

Luigi Genovese[a,*], Brice Videau[d], Matthieu Ospici[b,c,d], Thierry Deutsch[d], Stefan Goedecker[e], Jean-François Méhaut[b]

[a] *European Synchrotron Radiation Facility, 6, rue Horowitz, BP 220, 38043 Grenoble cedex, France*
[b] *Université Joseph-Fourier, laboratoire d'informatique de Grenoble – INRIA, BP 53, 38041 Grenoble cedex 9, France*
[c] *Bull SAS, 1, rue de Provence, 38130 Echirolles, France*
[d] *Laboratoire de simulation atomistique (L_Sim), SP2M/INAC/CEA, 17, avenue des Martyrs, 38054 Grenoble cedex, France*
[e] *Institut für Physik, Universität Basel, Klingelbergstr. 82, CH-4056 Basel, Switzerland*

A R T I C L E   I N F O

Article history:
Available online 30 December 2010

Keywords:
Computer science
Density Functional Theory
Daubechies wavelets
BigDFT

Mots-clés :
Informatique, algorithmique
Théorie de la Fonctionnelle de la Densité
Ondelettes de Daubechies
BigDFT

A B S T R A C T

In this contribution we will describe in detail a Density Functional Theory method based on a Daubechies wavelets basis set, named BigDFT. We will see that, thanks to wavelet properties, this code shows high systematic convergence properties, very good performances and an excellent efficiency for parallel calculations. BigDFT code operation are also well-suited for GPU acceleration. We will discuss how the problematic of fruitfully benefit of this new technology can be match with the needs of robustness and flexibility of a complex code like BigDFT. This work may be of interest not only for expert in electronic structure calculations, but may also provide feedback to the wider community of high performance scientific computing.

R É S U M É

Dans cet article nous allons décrire en détail BigDFT, une mise en oeuvre de la Théorie de la Fonctionnelle de la Densité basée sur les ondelettes de Daubechies. Nous verrons que, grâce aux propriétés des ondelettes, ce code présente une bonne convergence systématique, de très bonnes performances et un excellent passage à l'échelle lors de calculs distribués. Les opérations constituantes de BigDFT sont également bien adaptée aux accélérateurs de type GPU. Nous analyserons comment bénéficier efficacement de cette nouvelle technologie tout en respectant les contraintes de robustesse et de flexibilité d'un programme de la complexité de BigDFT. Ce travail peut intéresser non seulement les experts en calcul des structures électroniques, mais également constituer un retour d'expérience pour la communauté plus large du calcul scientifique à hautes performances.

* Corresponding author.
E-mail address: luigi.genovese@esrf.fr (L. Genovese).

1631-0721/$ – see front matter © 2010 Académie des sciences. Published by Elsevier Masson SAS. All rights reserved.
doi:10.1016/j.crme.2010.12.003

1. Introduction

In the recent years, the development of efficient and reliable methods for studying matter at atomistic level has become an asset for important advancements in a wide range of disciplines (material science, biology, chemistry, etc.). Both modern technological evolution and the need for new conception of materials and nanoscaled devices require a deep understanding of the properties of systems of many atoms from a fundamental viewpoint. To this aim, the support of computer simulation can be of great importance. Indeed, via computer simulation scientists try to model systems with many degrees of freedom by giving a set of "rules" of general validity (under some assumptions). Once these "rules" come from first-principles laws, these simulation have the ambition to model system properties from a fundamental viewpoint. With such a tool, the properties of existing materials can be studied in depth, and new materials and molecules can be conceived, with potentially enormous scientific and technological impact.

In this context, the advent of modern supercomputers represents an important resource in view of advancements in this field. In other terms, the physical properties which can be analysed via such methods are tightly connected to the computational power which can be exploited for calculation. It is thus important to provide reliable solutions to benefit from the enhancements of computational power in order to use these tools in more challenging systems.

The Kohn–Sham (KS) formalism of the Density Functional Theory (DFT) approach [1] is the most widely used first-principles method for investigating properties of atomistic systems. The increasing of the computational power of modern supercomputers has further stimulated the interest of the community for electronic structure calculations of systems with many electrons. Systems which were untractable only few years ago become now accessible with the advent of modern machines. However, despite the approximate nature of the approach, the computational demand becomes huge already for systems with few hundreds atoms. For the most common DFT codes, the number of computational operations scales cubically with respect to the number of atoms in the system. As a result, the computational overhead for treating systems with large number of atoms now represents a serious limitation for the maximum size of the system considered. A high performance computing (HPC) electronic structure program will make the analysis of more complex systems and environments possible, thus opening a path towards new discoveries.

1.1. The importance of the basis set for HPC DFT codes

The computational machinery of DFT calculations has been widely developed in the last decade, giving rise to a plethora of DFT codes. The usage of DFT calculation has thus become more and more common, and its domain of application comprises solid state physics, chemistry, materials science, biology and geology.

From a computational point of view, one of the most important characteristics of a DFT code is the set of basis functions used for expressing the KS orbitals. The domain of applicability of a code is tightly connected to this choice. For example, a non-localised basis set like plane waves is highly suitable for electronic structure calculations of periodic and/or homogeneous systems, while it is much less efficient in expanding localised information, which has a wider range of components in the reciprocal space. For these reasons DFT codes based on plane waves are not convenient for simulating inhomogeneous or isolated systems like molecules, due to the high memory requirements for such kind of simulations.

A remarkable difference should be also made between codes which use systematic and non-systematic basis sets. A systematic basis set allows us to calculate the exact solution of the KS equations with arbitrarily high precision as the number of basis functions is increased. In other terms, the numerical precision of the results is related to the number of basis functions used to expand the KS orbitals. With such a basis set it is thus possible to obtain results that are free of errors related to the choice of the basis, eliminating a source of uncertainty. A systematic basis set allows us thus to really calculate the solution of a particular exchange correlation functional. On the other hand, an example of a non-systematic set is provided by Gaussian type basis, for which over-completeness may be achieved before convergence. Such basis sets are more difficult to use, since the basis set must be carefully tuned by hand by the user, which will sometimes require some preliminary knowledge of the system under investigation. This is the most important weakness of this popular basis set.

Another property which has a role in the performances of a DFT code is the orthogonality of the basis set. The use of non-orthogonal basis sets requires the calculation of the overlap matrix of the basis function and performing various operations with this overlap matrix such as inverting the matrix. This makes methods based on non-orthogonal basis functions not only more complicated but also slower.

In this contribution we will describe in detail a DFT method based on a Daubechies wavelets basis set [4]. This method is implemented in a DFT code called BigDFT, distributed under GNU-GPL license and integrated in the ABINIT [2] software package. A separate, standalone version of this code is also available and distributed under GNU-GPL license [3]. We will see that, thanks to wavelet properties, this code shows high systematic convergence properties, very good performances and an excellent efficiency for parallel calculations. BigDFT code operation are also well-suited for GPU acceleration. We will see in the following how the problematic of fruitfully benefit of this new technology can be match with the needs of robustness and flexibility of a complex code like BigDFT.

In the next few paragraphs we will discuss the importance of the properties of Daubechies wavelets in the context of electronic structure calculations.

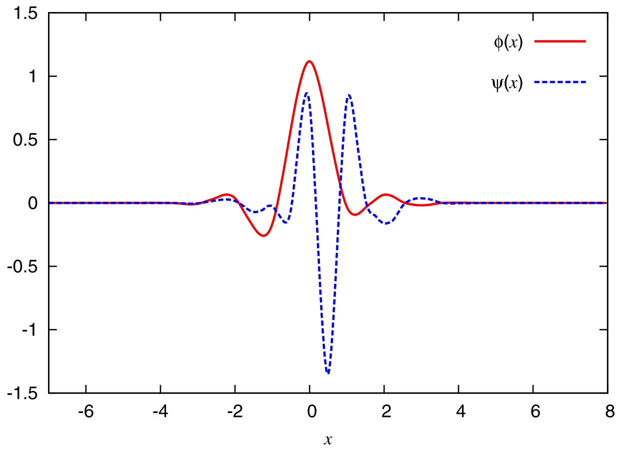


Fig. 1. Daubechies scaling function ϕ and wavelet ψ of order 16. Both are different from zero only in the interval from -7 to 8.

2. Overview of the BigDFT code

We describe in this section how the wavefunctions are expressed in the Daubechies basis. Though a more complete description can be found in [3], it is useful to revisit the principal points in view of a HPC implementation.

2.1. The basis set: Daubechies wavelets

There are two fundamental functions in Daubechies family, the scaling function $\phi(x)$ and the wavelet $\psi(x)$, see Fig. 1. The full basis set can be obtained from all translations by a certain grid spacing h of the scaling and wavelet functions centred at the origin. Both functions are localised, with compact support. All the properties of these functions can be obtained from the relations

$$\phi(x) = \sqrt{2}\sum_{j=1-m}^{m} h_j \phi(2x-j) \tag{1}$$

$$\psi(x) = \sqrt{2}\sum_{j=1-m}^{m} g_j \phi(2x-j)$$

which relate the basis functions on a grid with spacing h and another one with spacing $h/2$. h_j and $g_j = (-1)^j h_{-j+1}$ are the elements of a filter that characterises the wavelet family, and m is the order of the family. From Eq. (1), every scaling function and wavelet on a coarse grid of spacing h can be expressed as a linear combination of scaling functions at the finer grid level $h/2$. For this reason, wavelet functions complete the information which is lacking for refining the resolution level. In our implementation we use only two resolution levels, namely one level of adaptivity. We can thus classify the coarse and the fine degrees of freedom by the information expanded by the scaling and the wavelet functions respectively.

For a three-dimensional description, the simplest basis set is obtained by a tensor product of one-dimensional basis functions. For a two resolution level description, in each grid point the coarse degrees of freedom are expanded by a single three-dimensional function $\phi^0_{i_1,i_2,i_3}(\mathbf{r})$, while the fine degrees of freedom can be expressed by adding other seven basis functions, $\phi^\nu_{j_1,j_2,j_3}(\mathbf{r})$, which include tensor products with one-dimensional wavelet functions.

A wavefunction $\Psi(\mathbf{r})$ can thus be expanded in this basis:

$$\Psi(\mathbf{r}) = \sum_{i_1,i_2,i_3} c^0_{i_1,i_2,i_3}\phi^0_{i_1,i_2,i_3}(\mathbf{r}) + \sum_{j_1,j_2,j_3}\sum_{\nu=1}^{7} c^\nu_{j_1,j_2,j_3}\phi^\nu_{j_1,j_2,j_3}(\mathbf{r}) \tag{2}$$

The sum over i_1, i_2, i_3 runs over all the grid points contained in the low resolution region and the sum over j_1, j_2, j_3 over all the points contained in the (generally smaller) high resolution region. Such points belong to a uniform mesh of grid spacing h. Each wavefunction is then associated to a set of coefficients $\{c^\mu_{j_1,j_2,j_3}\}$, $\mu = 0,\ldots,7$. The wavefunctions are stored in a compressed form where only the non-zero scaling function and wavelets coefficients are stored. The basis set being orthogonal, several operations such as scalar products among different orbitals and between orbitals and the projectors of the non-local pseudopotential can directly be done in this compressed form.

The wavefunction can be also expanded in the pure fine scaling function representation (a basis set which contains only scaling functions $\phi_{i'_1,i'_2,i'_3}$ centred on a finer grid of spacing $h' = h/2$):

$$\Psi(\mathbf{r}) = \sum_{i'_1,i'_2,i'_3} s_{i'_1,i'_2,i'_3} \phi_{i'_1,i'_2,i'_3}(\mathbf{r}) \tag{3}$$

The transformation between a mixed coarse scaling function/wavelet representation and a pure fine scaling function representation is done by the fast wavelet transformation [8] which is a three-dimensional, separable convolution, that can be obtained from the filters h_j and g_j of Eq. (1).

In the following sections we will illustrate the main operations which must be performed in the context of a DFT calculation.

2.2. The Kohn–Sham formalism

In the KS formulation of DFT, the KS wavefunctions $|\Psi_i\rangle$ are eigenfunctions of the KS hamiltonian, with pseudopotential V_{psp}:

$$\left(-\frac{1}{2}\nabla^2 + V_{\text{KS}}[\rho] + V_{\text{psp}}\right)|\Psi_i\rangle = \epsilon_i|\Psi_i\rangle \tag{4}$$

The KS potential $V_{\text{KS}}[\rho]$ is a functional the electronic density of the system:

$$\rho(\mathbf{r}) = \sum_{i=1}^{\mathcal{N}_{\text{orbitals}}} n_{\text{occ}}^{(i)} \left|\Psi_i(\mathbf{r})\right|^2 \tag{5}$$

where $n_{\text{occ}}^{(i)}$ is the occupation of orbital i.

The KS potential $V_{\text{KS}}[\rho] = V_H[\rho] + V_{\text{xc}}[\rho] + V_{\text{ext}}$ contains the Hartree potential V_H, solution of the Poisson's equation $\nabla^2 V_H = -4\pi\rho$, the exchange-correlation potential V_{xc} and the external ionic potential V_{ext} acting on the electrons. In BigDFT code the pseudopotential term V_{psp} is of the form of norm-conserving GTH-HGH pseudopotentials [5–7], which have a local and a non-local term, $V_{\text{psp}} = V_{\text{local}} + V_{\text{non-local}}$. The KS hamiltonian can then be written as the action of three operators on the wavefunction:

$$\left(-\frac{1}{2}\nabla^2 + V_L + V_{\text{non-local}}\right)|\Psi_i\rangle = \epsilon_i|\Psi_i\rangle \tag{6}$$

where $V_L = V_H + V_{\text{xc}} + V_{\text{ext}} + V_{\text{local}}$ is a real-space based (local) potential, and V_{nonlocal} comes from the pseudopotentials.

As usual in a KS DFT calculation, the application of the hamiltonian is a part of a self-consistent cycle, needed for minimising the total energy. In addition to the usual orthogonalisation routine, in which scalar products $\langle\Psi_i|\Psi_j\rangle$ should be calculated, another operation which is performed on wavefunctions in BigDFT code is the preconditioning. This is calculated by solving the Helmholtz equation

$$\left(-\frac{1}{2}\nabla^2 - \epsilon_i\right)|\tilde{g}_i\rangle = |g_i\rangle \tag{7}$$

where $|g_i\rangle$ is the gradient of the total energy with respect to the wavefunction $|\Psi_i\rangle$, of energy ϵ_i. The preconditioned gradient $|\tilde{g}_i\rangle$ is found by solving Eq. (7) by a preconditioned conjugate gradient method.

2.3. The kinetic operator

The matrix elements of the kinetic energy operator on the basis functions can be calculated analytically [9,3]. For the pure fine scaling function representation described in Eq. (3), the result of the application of the kinetic energy operator on this wavefunction, has the expansion coefficients $\hat{s}_{i'_1,i'_2,i'_3}$, which are related to the original coefficients $s_{i'_1,i'_2,i'_3}$ by a convolution

$$\hat{s}_{i'_1,i'_2,i'_3} = \frac{1}{2}\sum_{j'_1,j'_2,j'_3} K_{i'_1-j'_1,i'_2-j'_2,i'_3-j'_3} s_{j'_1,j'_2,j'_3} \tag{8}$$

where

$$K_{i_1,i_2,i_3} = T_{i_1}\delta_{i_2}\delta_{i_3} + \delta_{i_1}T_{i_2}\delta_{i_3} + \delta_{i_1}\delta_{i_2}T_{i_3} \tag{9}$$

and T_i are the filters of the one-dimensional second derivative in Daubechies scaling functions basis, which can be computed analytically.

2.4. *Application of the local potential*

The potential V_L is defined in real space, in particular on the points of the finer grid of spacing h'. The application of the local potential in Daubechies basis consists of the basis decomposition of the function product $V_L(\mathbf{r})\Psi(\mathbf{r})$. As explained in [10,3], the simple evaluation of this product in terms of the point values of the basis functions is not precise enough. A better result may be achieved by performing a transformation to the wavefunction coefficients, which allows to calculate the values of the wavefunctions on the fine grid, via a smoothed version of the basis functions. This is the so-called "magic filter" transformation, which can be expressed as follows:

$$\Psi(\mathbf{r}_{i'_1,i'_2,i'_3}) = \sum_{j'_1,j'_2,j'_3} \omega_{i'_1-j'_1}\omega_{i'_2-j'_2}\omega_{i'_3-j'_3} s_{j'_1,j'_2,j'_3} \tag{10}$$

and allows to express with better accuracy the potential application. In other terms, the point values of a given wavefunction $|\Psi\rangle$ are expressed as if $\Psi(\mathbf{r})$ would be the smoothest function which has the same Daubechies expansion coefficients of $|\Psi\rangle$. This procedure guarantees the highest precision ($\mathcal{O}(h^{16})$ in the potential energy) and can be computationally expressed by a three-dimensional separable convolution in terms of the filters ω_i. After application of the local potential (pointwise product), a transposed magic filter transformation can be applied to obtain Daubechies expansion coefficients of $V_L|\Psi\rangle$.

2.5. *Local density calculation*

The density of the electronic system is derived from the square of the point values of the wavefunctions (see Eq. (5)). As described is Section 2.4, a convenient way to express the point values of the wavefunctions is to apply the magic filter transformation to the wavefunctions expressed in Daubechies basis. The operations needed for calculating the local density would then be identical to the operation needed to evaluated the real-point values of the wavefunctions.

2.6. *Other operations for hamiltonian application*

The local potential V_L can be obtained from the local density ρ by solving the Poisson's equation and by calculating the exchange-correlation potential $V_{\text{xc}}[\rho]$. These operations are performed via a Poisson solver based on interpolating scaling functions [11], which is a basis set tightly connected with Daubechies functions, optimal for electrostatic problems, and which allows for mixed boundary conditions. A description of this Poisson solver can be found in Refs. [12,13].

The complete hamiltonian contains also the non-local part of the pseudopotential which, thanks to the orthogonality of Daubechies wavelets, can directly be applied in the compressed form. The schematic of all these operations is depicted in Fig. 2.

2.7. *The code structure: preliminary CPU investigation*

Applying the hamiltonian operator on the KS wavefunction is only one of the operations which are performed in the BigDFT code. During the self-consistent cycle, wavefunctions have to be updated and orthogonalised. Since all the basis functions are orthogonal with each other, the overlap matrices have to be calculated via suitable calls to BLAS routines, and then processed with LAPACK subprograms.

An optimisation iteration of a KS wavefunction is organised as follows:

(I) local hamiltonian, construction and application,
(II) non-local hamiltonian,
(III) overlap matrix,
(IV) preconditioning,
(V) wavefunction update,
(VI) orthogonalisation (Cholesky factorisation).

The steps I and II have been described in Fig. 2. The preconditioner (step IV) can also be expressed via a kinetic convolution, as described above. Steps III, V and VI are performed via BLAS/LAPACK calls. In the current hybrid implementation, we can execute on the GPU the steps I and IV and also all BLAS routines performed in steps III (DGEMM), V and VI (DSYRK, DTRMM). The step IV is performed via a call to the DPOTRF LAPACK routine.

3. Parallelisation

Two data distribution schemes are used in the parallel version of our program. In the orbital distribution scheme, each processor works on one or a few orbitals for which it holds all its scaling function and wavelet coefficients. In the coefficient distribution scheme (see Fig. 3) each processor holds a certain subset of the coefficients of all the orbitals. Most of

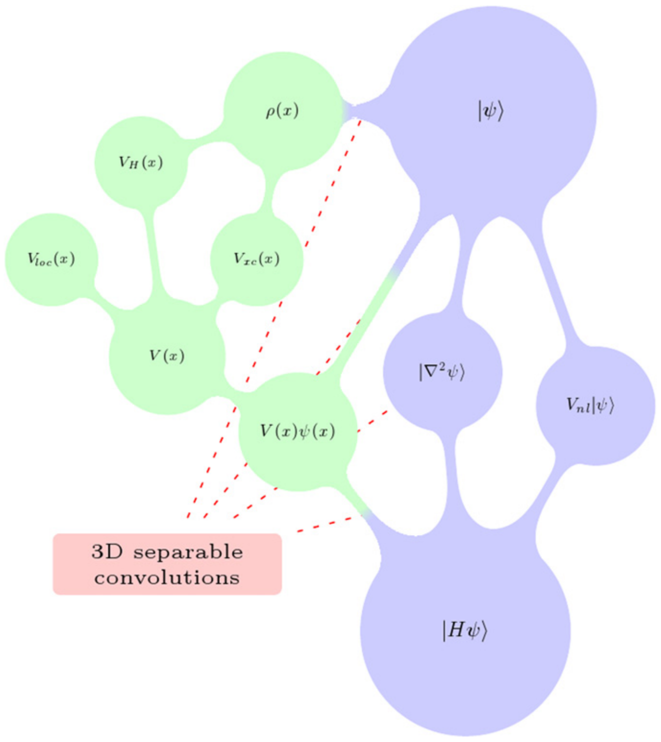


Fig. 2. Schematic of the application of the hamiltonian in the BigDFT formalism. The operator $H = -\frac{1}{2}\nabla^2 + V + V_{\mathrm{nl}}$ is applied on the wavefunction $|\psi\rangle$. The local potential V is derived from charge density ρ, which is built from the wavefunctions. In the BigDFT basis, most of the operations which are needed can be expressed via the action of 3d convolutions.

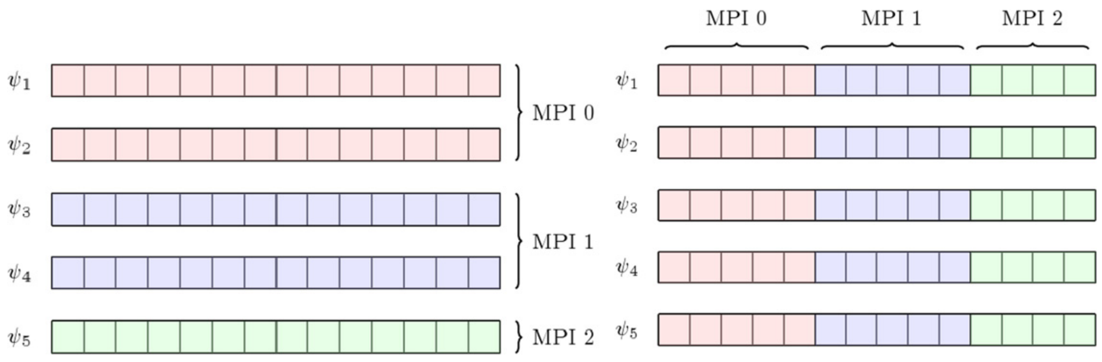


Fig. 3. Orbital (left) and coefficient (right) distribution schemes.

the operations such as applying the hamiltonian on the orbitals, and the preconditioning is done in the orbital distribution scheme. This has the advantage that we do not have to parallelise these routines and we therefore achieve almost perfect parallel speedup. The calculation of the Lagrange multipliers that enforce the orthogonality constraints onto the gradient as well as the orthogonalisation of the orbitals is done in the coefficient distribution scheme (Fig. 3). For the orthogonalisation we have to calculate the matrix $\langle \Psi_j | \Psi_i \rangle$ and for the Lagrange multipliers the matrix $\langle \Psi_j | H | \Psi_i \rangle$. So each matrix element is a scalar product and each processor is calculating the contribution to this scalar product from the coefficients it is holding. A global reduction sum is then used to sum the contributions to obtain the correct matrix. Such sums can easily be performed with the very well optimised BLAS-LAPACK libraries. Switch back and forth between the orbital distribution scheme and the coefficient distribution scheme is done by the MPI global transposition routine MPI_ALLTOALL(V). For parallel computers where the cross sectional bandwidth [14] scales well with the number of processors this global transposition does not require a lot of CPU time. The most time consuming communication is the global reduction sum required to obtain the total charge distribution from the partial charge distribution of the individual orbital.

Table 1
Computational time in seconds for a single minimisation iteration for different runs of a 44-atoms cinchonidine molecule. The values for different cutoff energies E_c for the plane waves run are shown. The input parameters for the wavelet run are chosen such as to obtain the same absolute precision of the plane wave calculations. The plane wave runs are performed with the ABINIT code, which uses iterative diagonalisation and with CPMD code [15] in direct minimisation. These timings are taken for a serial run on a 2.4 GHz AMD Opteron CPU.

E_c (Ha)	ABINIT (s)	CPMD (s)	Abs. precision	Wavelets (s)
40	403	173	3.7×10^{-1}	30
50	570	207	1.6×10^{-1}	45
75	1123	422	2.5×10^{-2}	94
90	1659	538	9.3×10^{-3}	129
145	4109		2×10^{-4}	474

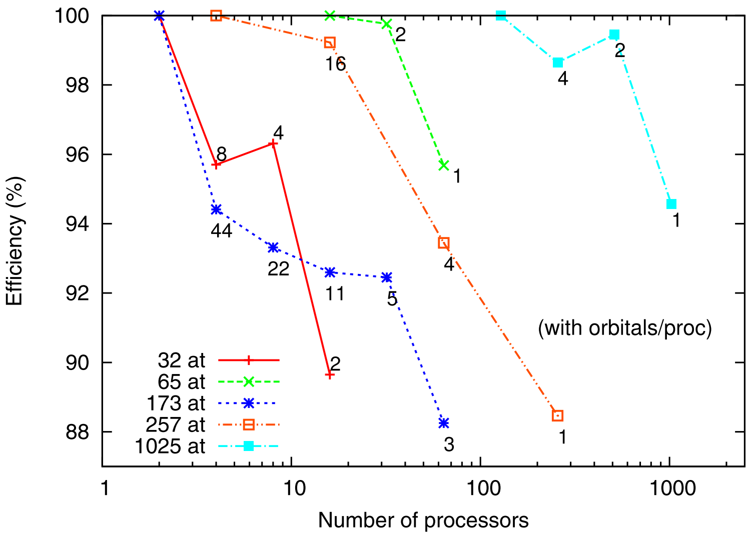


Fig. 4. Efficiency of the parallel implementation of the code for several runs with different number of atoms. The number close to each point indicates the number of orbitals treated by each processors, in the orbital distribution scheme.

3.1. OpenMP parallelisation

In the parallelisation scheme of the BigDFT code another level of parallelisation was added via OpenMP directive. In particular, all the convolutions and the linear algebra part can be executed in multi-threaded mode. This add further flexibility on the parallelisation scheme. At present, several strategies are under analysis for systems with different sizes to understand the best repartition of the data between nodes such as to minimise the computational overhead.

4. BigDFT performances. Optimisation strategies

We have evaluated the amount of time spent for a given operation on a typical run. To do this we have profiled the different sections of the BigDFT code for a parallel calculation. In Fig. 10 below we show the percent of time which is dedicated to any of the above described operation, for runs with different architectures.

4.1. Performance results

We have applied our method on different molecular systems in order to test its performances. As expected, the localisation of the basis set allows us to reduce considerably the number of degrees of freedom (i.e. the number of basis function which must be used) to attain a given absolute precision with respect to a plane-wave code. This fact reduces the memory requirements and the number of floating point operations.

The parallelisation scheme of the code is tested and has given the efficiency detailed in Fig. 4. The overall efficiency is always higher than 88%, also for large systems with a big number of processors.

It is also interesting to see which is the computational share of the different sections of the code with respect to the total execution time. Figs. 5 and 10 represent the percentage of the computational time for the different section of the code. The code operations were regrouped following their nature: the linear algebra operations (scalar products and orthogonalisation), the convolution-based routines (density construction and local hamiltonian application), and the rest of the operations (Poisson solver, non-local potential application). All the categories may benefit from MPI parallelisation, and some of them (linear algebra and convolutions operation) also from the lower level OpenMP parallelisation.

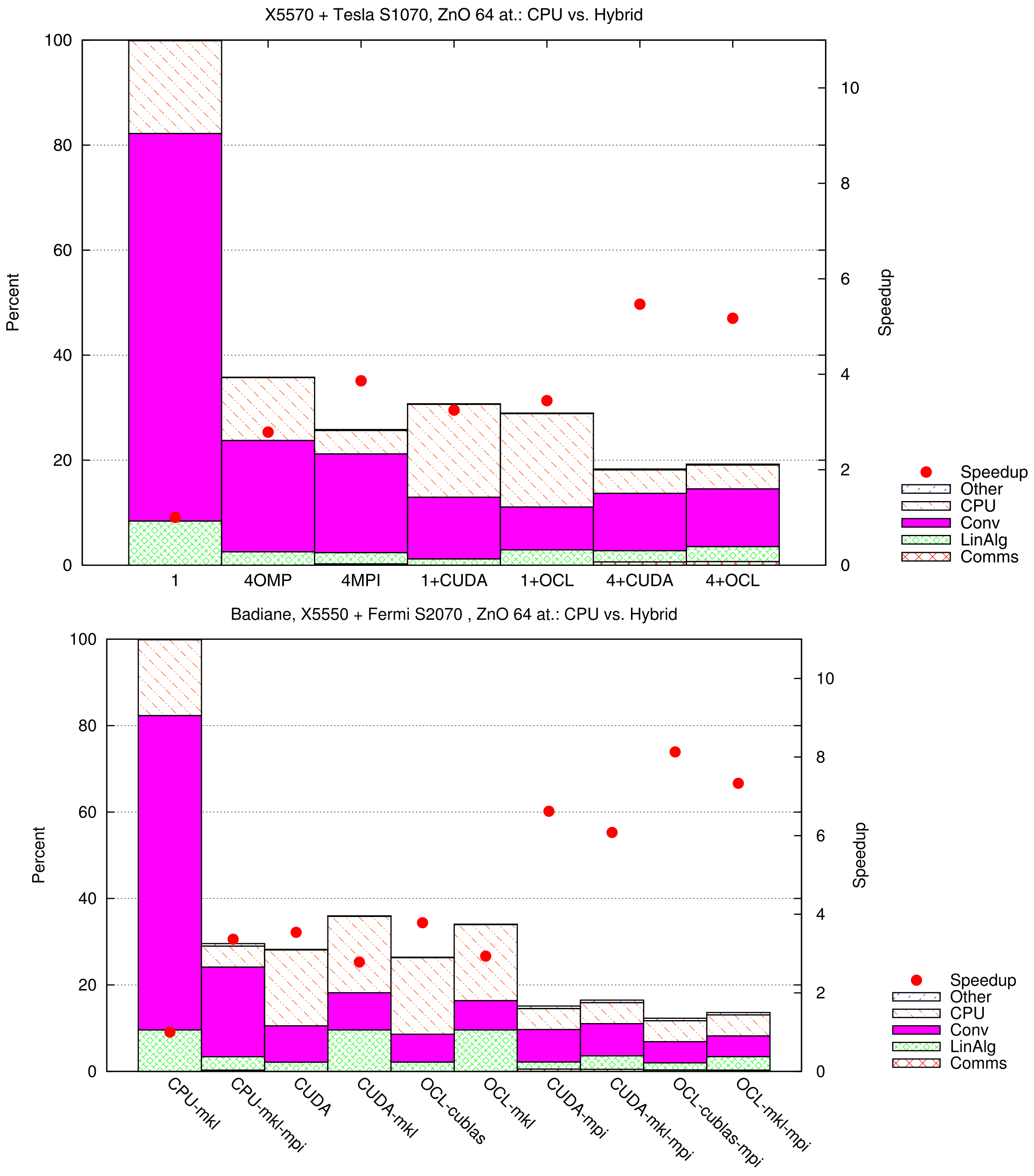


Fig. 5. Performance of a run of the BigDFT code for a 64 atoms ZnO system. Different acceleration strategies are compared with respect to the time spent for a sequential pure CPU run. Two different platforms were used for this test; in the first run (top panel), a Nehalem quad-core is related to a Tesla S1070 GPU (CUDA 3.0 driver, with OpenCL support). In the second run (bottom panel), a Nehalem processor is associated to a Fermi S2070 card, with CUDA 3.2.

While for relatively small systems the most time-dominating part of the code is related to the convolutions and Poisson solver operations, for large systems the most expensive section is by far the calculation of the linear algebra operations. The operations performed in this section scales cubically with respect to the number of atoms. Apart from the Cholesky factorisation, which has a scaling of $\mathcal{O}(n_{\mathrm{orb}}^3)$, where n_{orb} is the number of orbitals, the cubic terms are of the form

$$\mathcal{O}(n \cdot n_{\mathrm{orb}}^2) \tag{11}$$

where n is the number of degrees of freedom, i.e. the number of scaling function and wavelet expansion coefficients. Both the calculation of the overlap matrices and the orthogonality transformation of the orbitals lead to this scaling. For a code based on a systematic basis set, the number of the coefficients n is typically much bigger than the number of orbitals. For this reason, the optimisation of the operations which depend of the number of coefficients is of primary importance to reduce the computational overhead.

4.2. Convolutions optimisation

Convolutions are among the basic processing blocks of BigDFT. Special care has to be taken regarding their performances. The CPU convolutions of BigDFT have thus been thoroughly optimised. The convolutions can be expressed with three nested loops.

Despite their simplicity, optimising convolutions is a tedious work. The unusual memory access patterns combined with the transposition of the result defeat the compiler optimiser. For instance, an unoptimised Magic Filter convolution on an *Intel(R) Xeon(R) CPU X5550 @ 2.67 GHz* yields 0.55 GFLOPS which barely represents 5% of peak performances (10.7 GFLOPS).

A first optimisation pass has been applied on the Fortran code directly. Unrolling the outer loop allows reuse of the filter value and index calculations for outer elements. Using standard optimisation, tis code runs at 2.4 GFLOPS, which represents 22% of peak performances. But this code can also be compiled using automatic vectorisation and reaches 3.4 GFLOPS corresponding to 32% of peak performances.

Optimisations granted an acceleration of a factor 6. Nonetheless, this result is unsatisfactory regarding peak attainable performances. The code was thus hand-tuned in order to make better use of the available vector engine. Several dozens of memory access patterns and vectorising schemes were designed and tested in order to find the most efficient. The resulting Magic Filter convolution runs at 7.7 GFLOPS yielding 72% of peak performances. This convolution is 13 times faster than the unoptimised one.

This optimisation process is difficult and tedious, but the regularity in tested patterns may allow automated pattern generation and algorithm tuning. If this proves successful every convolution of BigDFT will be optimised accordingly in the future.

5. GPU acceleration

In the past few years, the possibility of using Graphic Processing Units (GPU) for scientific calculations has raised a lot of interest. A technology initially developed for home PC hardware has rapidly evolved in the direction of programmable parallel streaming processor. The features of these devices, in particular the very low price performance ratio, together with the relatively low energy consumption, make them actractive platforms for intensive scientific computations. A lot of scientific applications have been recently ported on GPU, including for example molecular dynamics [16], quantum Monte-Carlo [17], Finite Element Methods [18].

The operation of the BigDFT code are well suited for GPU acceleration. Indeed, on one hand the computational nature of 3D separable convolutions may allow to write efficient routines which may benefit of GPU computational power. On the other hand, the parallelisation scheme of BigDFT code is optimal in this sense: GPU can be used without affecting the nature of the communications between the different MPI process.

In the following sections, after a brief overview of a typical GPU architecture, we will discuss the effect of GPU porting of some code sections on the overall code performances.

5.1. GPU architecture

The GPUs (Graphics Processing Units) were initially devoted to computer games, where a set of processors was used to render, in parallel, a part of the graphical scene. Today, GPUs are very general since they can render a graphical scene but also perform general purpose computations (GPGPU: General-Purpose computation on Graphics Processing Units).

The GPUs can be viewed as a co-processor controlled exclusively by the CPU (Fig. 6), where GPU and CPU can run asynchronously that allow concurrent execution and memory transfer. One GPU cannot be used alone and must be connected to the CPU. For now, the only way to use a GPU is to connect it through the PCI-express bus which has a theoretical bandwidth of 8 GB/s. Both ATI/AMD [19] and NVIDIA [20] provides GPU designed to be used for GPUGPU. We focus in this section on NVIDIA GPU.

GPUs are multicore processors, but the contrary to CPU multicore, GPU have very simple processors with limited cache memory. These characteristics allow to put on one chip thousands of processors in order to get high calculation power. The most common GPUs used in high performance computing are based on the NVIDIA GT200 architecture consisting of a global memory (up to 4 GB) associated with a set of multiprocessors containing height processors. The most powerful NVIDIA GPU based on the GT200 architecture is called C1060 and holds 4 GB of memory and 32 multiprocessors, thus 240 cores (32×8). The theoretical peak performance achieved for this GPUs is near to the teraflop for single precision and close to 80 GFLOP for double precision computations

Recently, NVIDIA has released a new generation of GPU based on the so-called FERMI architecture. The double precision performance has improved dramatically: now there is only a factor two between double and single precision performance.

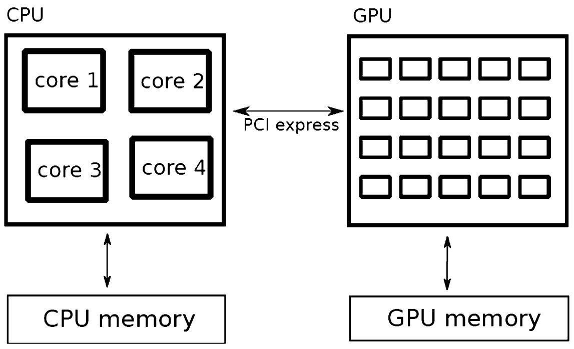


Fig. 6. A GPU associated with a CPU.

The C2070 card is, for now, the fastest GPU based on the FERMI architecture and consists of 6 GB of memory with 512 processors for a theoretical peak performance near to the teraflop for single precision and close to 500 GFLOPS for double precision computations.

5.2. Programming on GPU: OpenCL language

A major issue which concern GPU programming is that the programming paradigms are sensitive to the different architectures. In other terms, a code should be reprogrammed to run on a different GPU. Different vendors are about to provide a programming language suitable to the architecture they can provide in the market, and CUDA programming language of NVidia [20] is undoubtedly the most advanced in terms of functionalities and maturity.

Our development of GPU acceleration have been done in this language. When the OpenCL specification came out, and we ported our code in this language. The OpenCL BigDFT code is more optimised and complete than the older CUDA version. OpenCL is an open standard defined by the *Khronos Group* [21]. It is aimed at cross-platform parallel computing. Among the several types of devices defined by the OpenCL standard, the GPU one corresponds to modern Graphical Processing Units. These accelerators present a high performance over cost ratio, and a similarly high performance over power consumption ration. These accelerators have become increasingly available in computing clusters, and thus are target of choice for numerical experimenters.

The GPU device in OpenCL is made of several address spaces and a set of multiprocessors. In order to be cross-platform, OpenCL describes the work to be done in term of work-groups that are composed of work-items, and is aimed at data parallel tasks. When executing an OpenCL function (called *kernel*), work-items execute the same code. The difference between work-items from different work-groups is the visibility of address spaces. The four address spaces are *global*, *local*, *private* and *constant*. Each of these address spaces corresponds to a specific usage and has distinct characteristics. The *global* address space is shared among every work-group. This address space is usually large as it corresponds to the device on-board RAM. Accessing this address space is expensive, as latency is high, and should be done linearly rather than randomly as contiguous accesses can be coalesced into a single access. Synchronisation using *global* memory can be done through the use of atomic operations but is expansive and should be avoided. The *local* address space is shared among all work-items of a work group. *Local* memory is very fast (compared to *global*), and is organised in banks. Read an writes to local memory are simultaneous as long as they are done in different banks. Read are also simultaneous if work-items read from the same address. Multiprocessors have a reduced amount of shared memory, so it should be used wisely. Access to *private* memory is restricted to one work-item, and is used for local variables and parameters. *Constant* memory is visible by all work-groups, and is optimised for simultaneous reading from all work-items. Content of *constant* memory cannot be modified past initialisation.

5.3. Evaluation of benefits: performance with complex codes

We have seen that the operations which have to be explicitly ported on GPUs is a set of separable three-dimensional convolutions. From the GPU parallelism point of view, there is a set of different independent one-dimensional convolutions to be computed. Details of the implementation can be found in [3].

For a code with the complexity of BigDFT, the evaluation of the benefits of using a GPU-accelerated code must be performed at three different levels:

- Firstly, one has to evaluate the effective speedup provided by the GPU kernels with respect to the corresponding CPU routines which perform the same operations. This is the "bare" speedup, which of course, for a given implementation, depends of the computational power which the device can provide us. For the BigDFT code, these results are obtained by analysing the performances of the GPU kernels which performs the convolutions and the linear algebra (BLAS) operations, as provided by the CUBLAS library. Such results can be found in Fig. 8.

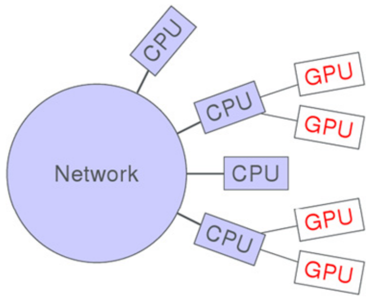


Fig. 7. Schematic of the organisation of a hybrid supercomputer: GPU act as a coprocessor. The access to the network of GPU should be mediated by the CPU.

- At the second level, the "complete" speedup has to be evaluated; the performances of the whole hybrid CPU/GPU code should be analysed with respect to the pure CPU executions. Clearly, this result depends of the actual importance of the ported routines in the context of the whole code (i.e. following the Amdahl's law). This is the first reliable result of the actual performance enhancements of the GPU porting of the code. For a hybrid code which originates from a monocore CPU program, this is the last level of evaluation.
- For a parallel code, there is still another step which has to be evaluated. This is the behaviour of the hybrid code in a parallel (or massively parallel) environment. Indeed, for parallel runs the picture is complicated by two things. The first one is the management of the extra level of communication which is introduced by the PCI-express bus, which may interact negatively with the underlying code communication scheduling (MPI or OpenMP par example), see Fig. 7. The second is the behaviour of the code for a number of GPU devices which is lower than the number of CPU processes which are running. In this case the GPU resource is not homogeneously distributed and the management of this fact adds an extra level of complexity. The evaluation of the code at this stage contributes at the "user level" speedup, which is the actual time-to-solution speedup.

5.4. GPU convolution routines and CUBLAS linear algebra

We have evaluated the performances of the GPU port of the 1D convolutions needed for the wavelet implementation of the local hamiltonian operators, and their three-dimensional counterpart. For these evaluations, we used a computer with an Intel Xeon Processor X5472 (3 GHz) and a NVidia Tesla S1070 card. The CPU version of BigDFT is deeply optimised with optimal loop unrolling and compiler options. The GPU code is compiled with the Intel Fortran Compiler (10.1.011) and the most aggressive compiler options (`-O2 -xT`). With these options the magic filter convolutions run at about 3.4 GFLOPS, similarly as we have shown above. All benchmarks are performed with double precision floating point numbers.

The GPU versions of the one-dimensional convolutions are about one order of magnitude faster than their CPU counterparts. We can then achieve an effective performance rate of the GPU convolutions of about 40 GFLOPs, by also considering the data transfers in the card. We are not close to peak performance since, on GPU, a considerable fraction of time is still spent in data transfers rather than in calculations. This appears since data should be transposed between input and output array, and the arithmetics needed to perform convolutions is not heavy enough to hide the latency of all the memory transfers. However, we will later show that these results are really satisfying for our purposes.

The performance graphs for the above mentioned convolutions, together with the compression–decompression operator, are indicated in Fig. 8 as a function of the size of the corresponding three-dimensional array.

Also the linear algebra operation can be executed on the card thanks to the CUBLAS routines. In Fig. 8 we present the speedups we obtain for double precision calls to CUBLAS routines for a typical wavefunction size of a BigDFT run as a function of the number of orbitals. These results take into account the amount of time needed for transfer the data of the card.

5.5. Three-dimensional operators

To build a three-dimensional operation one must chain three times the corresponding one-dimensional GPU kernels. We obtain in this way the three-dimensional wavelet transformations as well as the kinetic operator and the magic filter transformation (direct and transposed). The multiplication with the potential and the calculation of the square of the wavefunction are performed via the application of some special GPU kernels, based on the same guidelines described earlier. The GPU speedup of the local density construction as well as the local hamiltonian application and of the preconditioning operation is represented in Fig. 9 as a function of the compressed wavefunction size.

From these tests, we can see that both GPU-ported sections are order of magnitude (or more) faster than the corresponding CPU counterpart. We will now discuss the performance of the complete code, BigDFT.

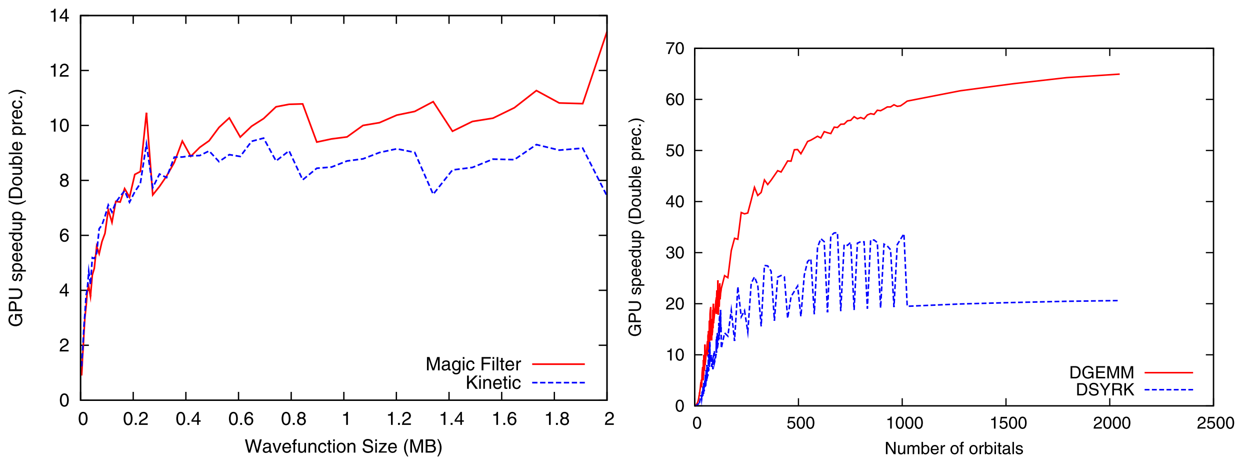


Fig. 8. Left panel: Double precision speedup for the GPU version of the fundamental operations on the wavefunctions as a function of the single wavefunction size. Right panel: Double precision speedup for the CUBLAS operations used in the code for a typical wavefunction size (300 KB) as a function of the number of orbitals.

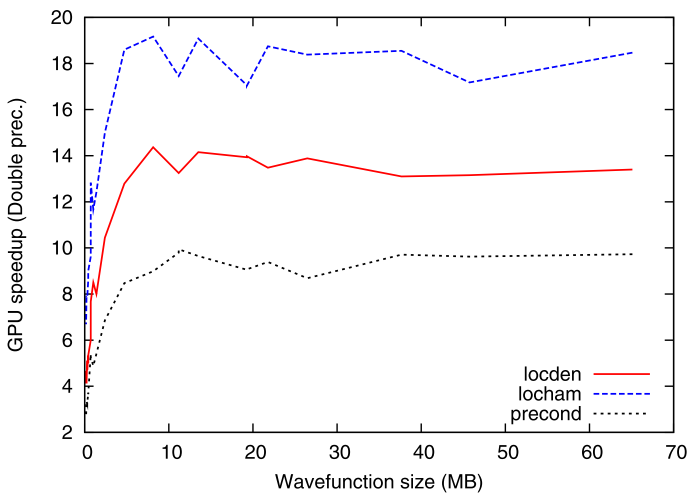


Fig. 9. Double precision speedup for the GPU version of the three-dimensional operators used in the BigDFT code as a function of the single wavefunction size.

5.6. Performance evaluation of hybrid code

As a test system, we used the ZnO crystal, which has a wurtzite bulk-like structure. Such system has a relatively high density of valence electrons so that the number of orbitals is rather large even for a moderate number of atoms.

We performed two kinds of tests. The first is related to the behaviour of the GPU-accelerated code on a single machine. Results can be found in Fig. 5. It can be seen that GPU acceleration contribute to a significant reduction of the overhead of linear algebra operations and convolutions. Both CUDA and the mode recent OpenCL implementation of these convolutions were tested. By combining these accelerations with MPI parallelisation on the machine, we may achieve a speedup up to one order of magnitude faster than the monocore CPU run.

The second test compares the behaviour of the Hybrid code in a multinode machine. To check the behaviour of the code for systems of increasing size, we performed a set of calculations for different supercells with increasing number of processors, such that the number of orbital per MPI process is kept constant. We performed a comparison for the same runs in which all the CPU cores have a GPU associated. The hybrid code is around 5.5 times faster than its pure CPU counterpart, regardless of the system size. We have also used the hybrid section of the CCRT Titane machine, with Intel X5570 (Nehalem) CPUs and Tesla S1070 cards. In this test we kept fixed the size of the system and increased the number of MPI process such as to decrease the number of orbitals per core. We then controlled the speedup of each run with the hybrid code. The parallel efficiency of the code is not particularly affected by the presence of the GPU. For this machine, due to the better CPU technology, the time-to-solution speedup is around 3.

Table 2
Energy consumed (in kJ) for different BigDFT runs.

	GPU	CPU	TOT
8 MPIs, 0 GPU (kJ)	60	**59**	**119**
8 MPIs, 1 GPUs (kJ)	38	35	**73**
8 MPIs, 2 GPUs (kJ)	28	19	**47**
16 MPIs, 4 GPUs (kJ)	28	23	**52**

Results of the two test are depicted in Fig. 10. Due to the fact that there is no hot-spot operation, the actual time-to-solution speedup of the complete code is influenced by the features of the code. In other words, keeping Amdhal's law in mind, a performance analysis is of great importance to forecast the final benefit of the accelerated parts.

These results are interesting and sounds very promising for a number of reasons. First of all, as already discussed, not all the routines of the code were ported on GPU. We focus our efforts to the operators which can be written via a convolution. Also the application of the non-local part of the hamiltonian can be performed on the GPU, and we are planning to do this in further developments.

Also the linear algebra operations can be further optimised. For the moment, only the calls to the BLAS routines were accelerated on the GPU, via suitable calls to the corresponding CUBLAS routines. Also the LAPACK routines, which are needed to perform the orthogonalisation process, can be ported on GPU, with a considerable gain. Indeed the linear algebra operations represent the most expensive part of the code for very large systems (see [3]). An optimisation of this section is then crucial for future improvements of the hybrid code.

5.7. Energy consumption

The emerging concept of "Green computing" deals with the energy consumption of modern supercomputers. To this aim, a particularly interesting quantity is the energy per flops, which should state how energetically efficient architectures are. the advent of GPU is trying to open pathways in this directions. However, such kind of quantities are always provided by using the peak performances declared by the vendors. As a matter of fact, since GPUs are specialised architectures, for some applications it is difficult to fully exploit the power. In other terms, in some situations it may be more difficult to reach performances close to peak for GPU than for CPUs. In order to provide results for a production application, we have tried to measure the energy consumption of some hybrid CPU GPU run with the BigDFT code.

The hardware on which our energy consumption experiments are done consists of 2 nodes with 16 GB of memory and two processors Intel Xeon L5420 (4 cores, Harpertown technology) cadenced at 2.5 GHz. On this machine, we have one NVIDIA C1070 rack that holds 4 GPUs with 4 GB of memory. Each node is connected to 2 GPUs NVIDIA held by our NVIDIA C1070, so, we can use only 2 GPUs by node. We have a full control of this machine and we can separately measure the power consumption of the GPUs and nodes. To measure the power consumption, we use a device named "APC Metered PDU" [22] designed to provide in real time the amount of electric current $i(t)$, in Ampere (A) consumed. The device is connected to both nodes and GPUs. The value measured for the nodes take into account all the devices held by the node (CPU cores, network card, hard disks, etc.).

For a BigDFT run time from t_{beg} to t_{end}, the total energy E consumed by the run in Joule (J) is

$$E = U\cos(\phi)\int_{t_{beg}}^{t_{end}} i(t)\,\mathrm{d}t \tag{12}$$

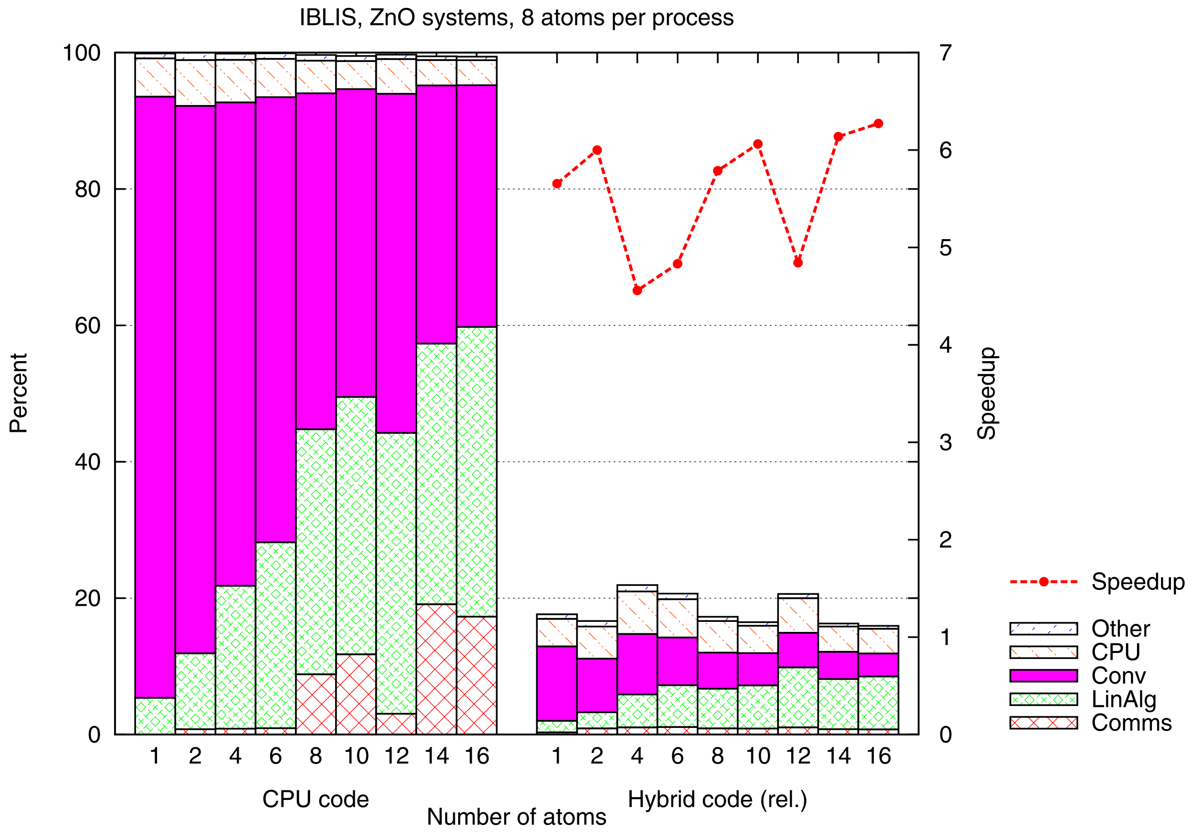
where $U = 230$ V and $\cos(\phi) = 0.98$.

Table 2 shows that when an application use the GPUs with efficiency (BigDFT in our example), GPUs can help to reduce energy consumption (59 kJ without GPU and 47 kJ with 2 GPUs). When 2 nodes and 4 GPUs are used, the total simulation time decrease, but the overhead of the additional CPUs and GPUs increase the total energy consumption.

The GPU idle consumption must also be taken into consideration. It is obvious but if the GPUs are not used, they consume a power between 300 W and 400 W (depending the number of cards activated) for nothing. So, on a machine where the GPUs cannot be disconnected, it is very important to use them with efficiency.

5.8. The time-to-solution problem

Another point which is of interest in the evaluation of the performances is the behaviour of the GPU acceleration with respect to the OpenMP parallelisation which is present in the BigDFT code. To give an example of that, we have performed ab initio molecular dynamic (AIMD) simulation of a 32 water molecules system with combined MPI–OpenMP parallelisation.



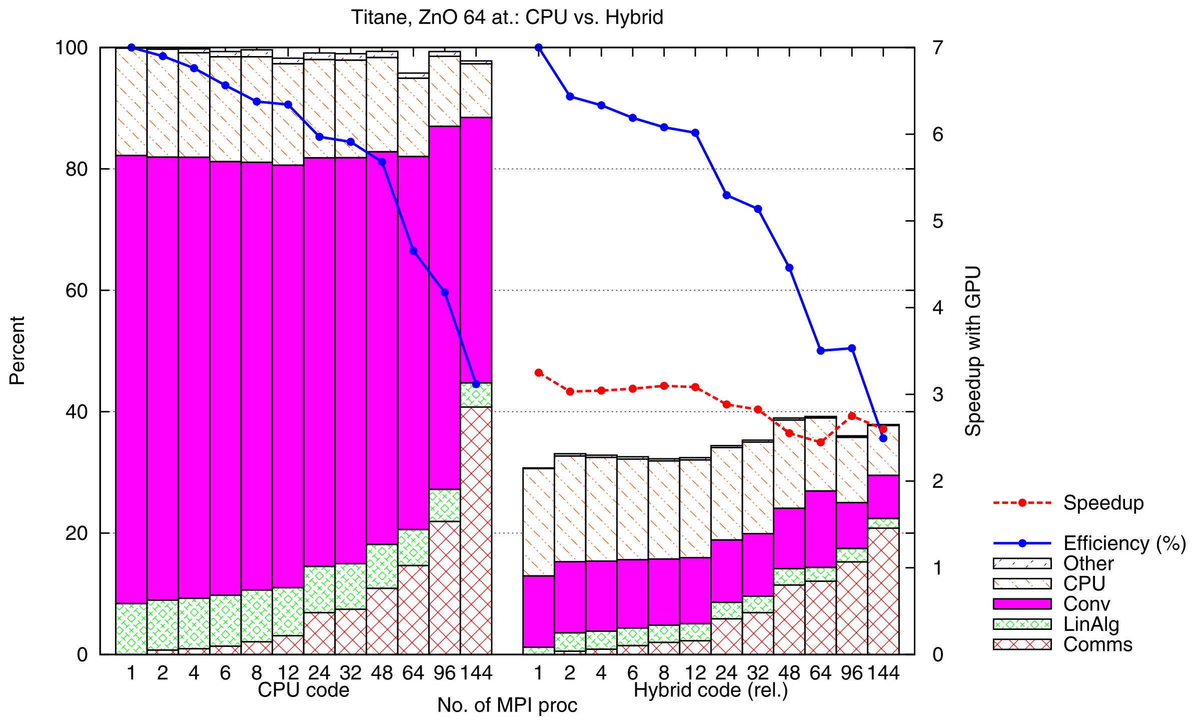


Fig. 10. Relative speedup of the hybrid DFT code wrt the equivalent pure CPU run. In the top panel, different runs for systems of increasing size have been done on an Intel X5472 3 GHz (Harpertown) machine. In the bottom panel, a given system have been tested with increasing number of processors on an Intel X5570 2.93 GHz (Nehalem) machine. The scaling efficiency of the calculation is also indicated. In the right side of each panel, the same calculation have been done by accelerating the code via one Tesla S1070 card per CPU core used, for both architectures. The speedup is around a value of six for a Harpertown, and around 3.5 for a Nehalem based calculation.

Table 3
Timings on Rosa (Cray XT) for one wavefunction optimisation step for a system of 32 water molecules with periodic boundary conditions. The number of cores is written as a product of the number of MPI processes and the number of OpenMP threads.

Number of cores	Serial	32 × 1	64 × 1	128 × 1	32 × 6 = 192	128 × 6 = 768	128 MPI + 128 GPU
SCF iteration (s)	170	7.2	3.8	2.0	1.5	0.44	0.30
Force evaluation (s)	2210	93.6	49.4	26.0	19.5	5.72	3.92
AIMD steps/day	40	923	1749	3323	4430	15 104	22 040

6. Conclusions and future directions

The problems that BigDFT aims to tackle are computationally intensive. The use of hybrid architectures is an efficient way to harness high amounts of computing power. Nonetheless, the scalability of the program is only one aspect of a high performance distributed code. The basic building blocks that compose the program should be thoroughly optimised. In this paper, we have identified several optimisations opportunities (communications, CPU convolutions, CPU/GPU overlap, communication/computation overlap, etc.) that can bring important performance improvements. There are certainly many more opportunities left to discover. But this process of optimisation is tedious and dependent on the target architecture. Automation of the optimising process for certain kind of operations (convolutions for instance) should be considered. Previous approach of this kind have proven successful [23,24], and their viability should be evaluated in BigDFT's context.

The GPU implementation of the code we propose fully respects these properties. We use double precision calculations, and we may achieve considerable speedup for the converted routines (up to a factor of 20 for some operations). Our developments are fully compatible with the existing parallelisation of the code, and the communication between CPU and GPU does not affect the efficiency of the existing implementation.

At present, developments in several directions are under way to further increase the efficiency of the usage of GPU in the context of BigDFT. Different developments directions are active at present:

- With the advent of Nehalem processor, we have seen that processors have had a big increase in performances in last years, which have not been followed by similar increase in network performances (especially latency). So, even tough the parallel efficiency is lower that in the past, the overall walltime has decreased significantly, which is not bad. For last generation architectures (and compact systems which have a lower components per orbital ratio) it is reasonable to expect an efficiency around 70%, instead of the 85–90% value we had until Harpertown architectures. We are at present implementing a new communication scheme of BigDFT which will use non-blocking communication that may allow to detect node failures and overlap communication and calculations at the same time. We believe that this new method will be competitive for architectures which have a large spread between network and CPU performances.
- The strategy of the repartition of the GPU resource should be reworked. In particular, the possibility of the concurrent kernel execution should be allowed whenever possible, and the strategies of GPU repartition should be adapted accordingly.

The hybrid BigDFT code, like its pure CPU counterpart, is available under GNU-GPL license and can be downloaded form the site in Ref. [3].

References

[1] W. Kohn, L.J. Sham, Phys. Rev. 140 (1965) A1133.
[2] X. Gonze, B. Amadon, P.-M. Anglade, J.-M. Beuken, F. Bottin, P. Boulanger, F. Bruneval, D. Caliste, R. Caracas, M. Coté, T. Deutsch, L. Genovese, Ph. Ghosez, M. Giantomassi, S. Goedecker, D. Hamann, P. Hermet, F. Jollet, G. Jomard, S. Leroux, M. Mancini, S. Mazevet, M. Oliveira, T. Rangel, Y. Pouillon, G.-M. Rignanese, D. Sangalli, R. Shaltaf, M. Torrent, M. Verstraete, G. Zerah, J. Zwanziger, ABINIT: first-principles approach to material and nanosystem properties, Comput. Phys. Comm. 180 (2009) 2582–2615.
[3] L. Genovese, et al., J. Chem. Phys 129 (2008) 014109;
L. Genovese, et al., J. Chem. Phys. 131 (2009) 034103, http://inac.cea.fr/sp2m/L_Sim/BigDFT.
[4] I. Daubechies, Ten Lectures on Wavelets, SIAM, Philadelphia, 1992.
[5] S. Goedecker, M. Teter, J. Hutter, Phys. Rev. B 54 (1996) 1703.
[6] C. Hartwigsen, S. Goedecker, J. Hutter, Phys. Rev. B 58 (1998) 3641.
[7] M. Krack, Theor. Chem. Acc. 114 (2005) 145.
[8] S. Goedecker, Wavelets and Their Application for the Solution of Partial Differential Equations, Presses Polytechniques Universitaires Romandes, Lausanne, Switzerland, ISBN 2-88074-398-2, 1998.
[9] G. Beylkin, SIAM J. Numer. Anal. 6 (1992) 1716.
[10] A.I. Neelov, S. Goedecker, J. Comput. Phys. 217 (2006) 312–339.
[11] G. Deslauriers, S. Dubuc, Constr. Approx. 5 (1989) 49.
[12] L. Genovese, T. Deutsch, A. Neelov, S. Goedecker, G. Beylkin, Efficient solution of Poisson's equation with free boundary conditions, J. Chem. Phys. 125 (2006) 074105.
[13] L. Genovese, T. Deutsch, S. Goedecker, Efficient and accurate three-dimensional Poisson solver for surface problems, J. Chem. Phys. 127 (2007) 054704.
[14] S. Goedecker, A. Hoisie, Performance Optimization of Numerically Intensive Codes, SIAM Publishing Company, Philadelphia, USA, ISBN 0-89871-484-2, 2001.
[15] CPMD Version 3.8: developed by J. Hutter, A. Alavi, T. Deutsch, M. Bernasconi, S. Goedecker, D. Marx, M. Tuckerman, M. Parrinello, Max-Planck-Institut für Festkörperforschung and IBM Zürich Research Laboratory, 1995–1999.

[16] J. Yang, et al., J. Comput. Phys. 221 (2007) 779.
[17] A. Anderson, et al., Comput. Phys. Comm. 177 (2007) 298.
[18] D. Göddeke, et al., Parallel Comput. 33 (2007) 10685.
[19] ATI stream technology, www.amd.com/stream.
[20] NVidia CUDA Programming Guide, version 3.1, see http://www.nvidia.com/object/cuda_home.html.
[21] Khronos Group, The OpenCL Standard, http://www.khronos.org/opencl/, 2009.
[22] APC–Metered Rack PDU, www.apc.com/.
[23] Markus Püschel, José M.F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, Nick Rizzolo, SPIRAL: code generation for DSP transforms, Proc. IEEE (special issue on Program Generation, Optimization, and Adaptation) 93 (2) (2005) 232–275.
[24] Matteo Frigo, Steven G. Johnson, The design and implementation of FFTW3, Proc. IEEE (special issue on Program Generation, Optimization, and Platform Adaptation) 93 (2) (2005) 216–231. Invited paper.