# Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures

Luigi Genovese,[1,a)] Matthieu Ospici,[2,3,4] Thierry Deutsch,[4] Jean-François Méhaut,[2] Alexey Neelov,[5,6] and Stefan Goedecker[5]
[1]*European Synchrotron Radiation Facility, 6 rue Horowitz, BP 220, 38043 Grenoble, France*
[2]*Laboratoire d'Informatique de Grenoble, Université Joseph Fourier, INRIA, 51, av. Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France*
[3]*Bull SAS, 1 rue de Provence, 38130 Echirolles, France*
[4]*Laboratoire de Simulation Atomistique (L_Sim), SP2M, INAC, CEA-UJF, 38054 Grenoble Cedex 9, France*
[5]*Institut für Physik, Universität Basel, Klingelbergstr. 82, 4056 Basel, Switzerland*
[6]*Institute for Computational Physics, Universität Stuttgart, Pfaffenwaldring 27, 70569 Stuttgart, Germany*

(Received 10 April 2009; accepted 10 June 2009; published online 16 July 2009)

We present the implementation of a full electronic structure calculation code on a hybrid parallel architecture with graphic processing units (GPUs). This implementation is performed on a free software code based on Daubechies wavelets. Such code shows very good performances, systematic convergence properties, and an excellent efficiency on parallel computers. Our GPU-based acceleration fully preserves all these properties. In particular, the code is able to run on many cores which may or may not have a GPU associated, and thus on parallel and massive parallel hybrid machines. With double precision calculations, we may achieve considerable speedup, between a factor of 20 for some operations and a factor of 6 for the whole density functional theory code. © *2009 American Institute of Physics*. [DOI: 10.1063/1.3166140]

## I. INTRODUCTION

The Kohn–Sham (KS) formalism of the density functional theory (DFT) approach[1] is a well-established first-principles method to investigate properties of atomistic systems. In recent years, the raising of the computational power of modern supercomputers further stimulated the interest of the community for electronic structure calculations of systems with many electrons. Systems, which were untractable only few years ago, now became accessible with the advent of modern machines. However, despite the approximate nature of the approach, the computational demand already becomes huge for systems with few hundreds of atoms. For the most distributed DFT codes, the number of computational operations increases with the third power of the number of atoms (cubic scaling). 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 possible strategy to circumvent this problem can be provided by the development of linear scaling algorithms for electronic structure calculations, which have been widely developed in recent years (see, e.g., Ref. 2). Such kind of approach has a crossover point with the traditional cubic codes, which is placed right around few hundreds atoms and, at present, they represent one of the most promising strategies for *ab initio* simulation of large systems that exhibit quantum mechanical behavior.

In the past few years, the possibility of using graphic processing units (GPUs) for scientific calculations raised a lot of interest. A technology initially developed for home personal computer hardware 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 attractive platforms for intensive scientific computations. A lot of scientific applications have been recently ported on GPU, including, for example, molecular dynamics,[3] quantum Monte Carlo,[4] and finite element methods.[5] In the domain of electronic structure calculation, the calculation of the exchange-correlation term in a Gaussian based DFT code has been implemented on GPU,[6] as well as the evaluation of the Coulomb potential[7] and the resolution-of-the-identity MP2 technique.[8] The self-consistent field calculation of the Gaussian-based GAMESS code was also ported on GPU.[9,10] Most of these implementation are performed on single precision calculation units and with a single CPU core connected to a single GPU card.

In this paper, we will present an implementation of a full DFT code which can run on massive parallel hybrid CPU-GPU clusters. Our implementation is based on the architecture of the most recent NVidia GPU cards (compute capability of type 1.3), which supports double precision floating point numbers. The routines that define the GPU kernels are thus coded within the specification of the CUDA language.[11] Our GPU port is performed over a DFT code based on Daubechies wavelets.[12] The latter is a systematic, orthogonal, multiresolution real-space basis set that presents optimal properties for expanding localized information. The properties of this basis set are well suited for an extension on a GPU-accelerated environment. This DFT code, named BIGDFT,[13] is delivered within the GNU-GPL license either in a standalone version or integrated in the ABINIT (Ref. 14) software package.

[a)]Electronic mail: luigi.genovese@esrf.fr. Tel.: +33 4 76 88 25 54.

The paper is organized as follows. First we present the main features of the BIGDFT code to explain how the operators of the KS Hamiltonian can be written in Daubechies wavelets basis. We will then discuss the implementation of these operators in the GPU architectures and inspect their performances separately from the complete DFT code. Afterwards we will inspect the problem of the transfer of the data on the card and we will finally discuss the performances of the full DFT code on parallel environments.

## II. OVERVIEW OF THE BIGDFT CODE

In the KS formulation of DFT, the KS wave functions $|\Psi_i\rangle$ are eigenfunctions of the KS Hamiltonian, with pseudopotential $V_{\mathrm{psp}}$,

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

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

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

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

The KS potential $V_{\mathrm{KS}}[\rho]=V_H[\rho]+V_{\mathrm{xc}}[\rho]+V_{\mathrm{ext}}$ contains the Hartree potential $V_H$, the solution of the Poisson's equation $\nabla^2 V_H=-4\pi\rho$, the exchange-correlation potential $V_{\mathrm{xc}}$ and the external ionic potential $V_{\mathrm{ext}}$ acting on the electrons. In the BIGDFT code the pseudopotential term $V_{\mathrm{psp}}$ is under the form of norm-conserving GTH-HGH pseudopotentials,[15–17] which have a local and a nonlocal term, $V_{\mathrm{psp}}=V_{\mathrm{local}}+V_{\mathrm{nonlocal}}$. The KS Hamiltonian can then be written as the action of three operators on the wave function[18–21]

$$\left(-\tfrac{1}{2}\nabla^2 + V_L + V_{\mathrm{nonlocal}}\right)|\Psi_i\rangle = \epsilon_i|\Psi_i\rangle, \tag{3}$$

where $V_L=V_H+V_{\mathrm{xc}}+V_{\mathrm{ext}}+V_{\mathrm{local}}$ is a real-space based (local) potential and $V_{\mathrm{nonlocal}}$ 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 minimizing the total energy. In addition to the usual orthogonalization routine, in which scalar products $\langle\Psi_i|\Psi_j\rangle$ should be calculated, another operation performed on wave functions in BIGDFT code is preconditioning. This is calculated by solving the Helmholtz equation

$$\left(-\tfrac{1}{2}\nabla^2 - \epsilon_i\right)|\tilde{g}_i\rangle = |g_i\rangle, \tag{4}$$

where $|g_i\rangle$ is the gradient of the total energy with respect to the wave function $|\Psi_i\rangle$ of energy $\epsilon_i$. The preconditioned gradient $|\tilde{g}_i\rangle$ is found by solving Eq. (4) by a preconditioned conjugate gradient method.

### A. Daubechies basis and convolutions

The set of basis functions used to express the KS orbitals is of key importance for the nature of the computational operations, which have to be performed. In the case of the BIGDFT code, the locality of the basis functions allows us to store the expansion coefficients of a KS orbital in a compressed form where only the nonzero values are stored. The basis set being orthogonal, several operations such as scalar products among different orbitals and between orbitals and the projectors of the nonlocal pseudopotential can directly be carried out in this compressed form. The application of kinetic and local potential operators can be written in terms of three-dimensional convolutions with *short*, *separable* filters. Such convolutions belong to three different classes; the kinetic one expresses *analytically* the expansion coefficients of the Laplacian of the KS wave function in the basis set. The second type of convolutions, the wavelet transformations, switches back and forth between a two-resolution level description and a uniform, high resolution description. The third class is the so-called "magic filter" transformations, through which the values of the KS wave functions on the points of the simulation box can be expressed with optimal accuracy. A more complete description of these operations can be found in the BIGDFT reference paper.[13]

### B. Local Hamiltonian operator

The described above operations must be combined for the application of the local Hamiltonian $(-(1/2)\nabla^2+V_L(\mathbf{r}))$. The order of the operation must be the following:

(1) Decompression of the data.
(2) Wavelet transformation.
(3) Local potential application:

(a) Magic filter transformation.
(b) Potential multiplication.
(c) Transposed magic filter transformation.

(4) Kinetic operator from the output of (2).
(5) Sum with potential application.
(6) Inverse wavelet transformation.
(7) Data compression.

In the CPU version of BIGDFT some of these operations are merged, e.g., (2) and (3a), as well as (3c) and (6), are combined into a set of common filters. However, in our GPU implementation, we chose to apply these operations separately, since it is simpler and accounts for better modularity. The wave function compression-decompression can also be calculated in the card.

### C. Local density calculation, Poisson solver

The density of the electronic system is derived from the square of the point values of the wave functions, [see Eq. (2)]. As described in Sec. II A, a convenient way to express the point values of the wave functions is to apply the magic filter transformation to the Daubechies basis expansion coefficients. The operations needed to calculate the local density would then be identical to points (1), (2), and (3a) of the list at Sec. II B, followed by an accumulation of the squares of the wave functions in the same array.

The local potential $V_L$ can be obtained from the local density $\rho$ by solving the Poisson's equation and by calculating the exchange-correlation potential $V_{\mathrm{xc}}[\rho]$. These operations are performed via a Poisson solver based on interpolating scaling functions,[22] a basis set tightly connected with Daubechies functions. The properties of this basis are optimal for electrostatic problems and mixed boundary conditions (BCs) can be treated explicitly. A description of this Poisson solver can be found in papers.[23,24]

### D. Parallelization for homogeneous computing clusters (CPU code)

Two data distribution schemes are used to parallelize the code. In the orbital distribution scheme (used for Hamiltonian application, preconditioning), each processor works on one or a few orbitals for which it holds all its expansion coefficients. The other operations are performed in the coefficient distribution scheme, in which each processor holds a certain subset of the coefficients of all the orbitals. The MPI global transposition routine MPI_ALLTOALL is used to switch back and forth between the orbital distribution scheme and the coefficient distribution scheme. Further flexibility has been added by allowing each processor to store variable number of orbitals and/or wave function coefficients. This is particularly useful for a hybrid architecture, when only some of the cores can be in relation with some hardware accelerator, like a GPU card. In the case of this variable repartition case, the communications are performed with suitable calls to MPI_ALLTOALLV routines. For parallel computers in which the cross sectional bandwidth[25] scales well with the number of processors, this global transposition does not require a lot of CPU time. The BIGDFT code shows very good computational performances and an excellent efficiency (above 90%) on such kind of architectures.

### E. Wavelet transformation and magic filters convolutions

In order to present the computational implementation, let us analyze in detail the magic filter transformation. A three-dimensional array $s$ (input array) of dimension $n_1,n_2,n_3$ is transformed into the output array $\Psi_r$ given by

$$\Psi_r(I_1,I_2,I_3) = \sum_{j_1,j_2,j_3=-L}^{U} \omega_{j_1}\omega_{j_2}\omega_{j_3} s(I_1-j_1,I_2-j_2,I_3-j_3). \tag{5}$$

With a lowercase index $i_p$ we indicate the elements of the input array, while with a capital index $I_p$ we indicate the indices after application of the magic filters $\omega_i$, which have extension from $-L$ to $U$. In BIGDFT, different BCs can be applied at the border of the simulation region, which affects the values of the array $s$ in Eq. (5) when the indices are outside bounds. For our GPU implementation we use periodic BC in the three dimensions, i.e., data are wrapped periodically, though the original CPU version of these convolutions also admits isolated and slablike BC. However, our GPU implementation can be straightforwardly extended to such BCs.

The most convenient way to calculate a three-dimensional convolution of this kind is by combining one-dimensional (1D) convolutions and array transpositions, as explained in Ref. 26. In fact, the calculation (5) can be cut in three steps:

(1) $F_3(I_3,i_1,i_2)=\Sigma_j\omega_j s(i_1,i_2,I_3-j) \quad \forall i_1,i_2,$

(2) $F_2(I_2,I_3,i_1)=\Sigma_j\omega_j F_3(I_3,i_1,I_2-j) \quad \forall I_3,i_1,$

(3) $\Psi_r(I_1,I_2,I_3)=\Sigma_j\omega_j F_2(I_2,I_3,I_1-j) \quad \forall I_2,I_3.$

The final result is thus obtained by a successive application of the same operation:

$$F(I,a) = \sum_{j=-L}^{U} \omega_j G(a,I-j) \quad \forall\, a=1,\ldots,N, I=1,\ldots,n. \tag{6}$$

The lowest level routine, which will be ported on GPU, is then a set of $N$ independent, 1D (periodic), convolutions of arrays of size $n$. The number $N$ equals $n_1n_2$, $n_1n_3$, and $n_2n_3$, respectively, for each step of the three-dimensional construction, while $n$ equals to the dimension, which is going to be transformed. The output of the first step is then taken as the input of the second, and so on and so forth.

This procedure can also be applied to the wavelet transformation by suitably changing the values of the filters and their extensions. This can be done thanks to the fact that the latter is a three-dimensional separable convolution as well, i.e., it has the same formal expression as Eq. (5).

### F. Kinetic convolution

A little, but substantial, difference should be stressed for the kinetic operator application. In this case, the three-dimensional filter is the sum of three different filters. This implies that the kinetic filter operation must be cut differently from the other separable convolutions:

(1) $K_3(I_3,i_1,i_2)=\Sigma_j T_j s(i_1,i_2,I_3-j) \quad \forall i_1,i_2,$

(2) $K_2(I_2,I_3,i_1)=\Sigma_j T_j s(i_1,I_2-j,I_3)+K_3(I_3,i_1,I_2) \quad \forall I_3,i_1,$

(3) $\hat{s}(I_1,I_2,I_3)=\Sigma_j T_j s(I_1-j,I_2,I_3)+K_2(I_2,I_3,I_1) \quad \forall I_2,I_3.$

Here $T$ indicates the one-dimensional kinetic filter. In this case, the three-dimensional kinetic operator can be also seen as a result of a successive application of the same operation,

$$K_p(I,a) = \sum_j T_j G_{p-1}(a,I-j) + K_{p-1}(a,I) \quad \text{and}$$
$$G_p(I,a) = G_{p-1}(a,I) \quad \forall\, a,I. \tag{7}$$

In other terms, the 1D kernel of the kinetic energy has two input arrays, $G_{p-1}$ and $K_{p-1}$, and returns two output arrays, $K_p$ and $G_p$, which are the transpositions of $G_{p-1}$. At the first step ($p=1$) we put $G_0=s$ and $K_0=0$. Eventually, for $p=3$, we have $K_3=\hat{s}$ and $G_3=s$. If $K_0\neq 0$, then $K_3=\hat{s}+K_0$. This permits the merge of steps (4) and (5) in the list of Sec. II B by putting $K_0$ equal to the output of step (3c). This algorithm can also be used for the Helmholtz operator of the preconditioner by putting $K_0=-\epsilon_i s$.

## III. IMPLEMENTATION OF CONVOLUTIONS ON GPU

### A. NVidia GPU architecture

A NVidia GPU is composed of a global memory and a set of multiprocessors.[11] Each multiprocessor includes eight computing units (cores) and a "private" shared memory. To obtain optimal performance, it is necessary to store the data to be processed in the private memory of multiprocessors. Fine-grained threads are the basic activities of parallel execution in CUDA. Indeed, a CUDA program is mapped on a set ("grid") of execution blocks running on multiprocessors. A block contains several threads executed by the processors of a multiprocessor. The processors execute the threads in a synchronous way (data parallel threads), in groups of 32 parallel threads called *warps*. The memory bandwidth of a GPU is used more efficiently when each half of a warp accesses a set of contiguous elements in the global memory. The shared memory is divided in 16 banks, each one containing a 32-bit word. Two contiguous words belong to different banks. Any shared memory request made of addresses in different banks can be performed simultaneously. Optimal performance will then be achieved for shared memory access if each thread of a half-warp accesses a different memory bank (no bank conflict). In this case, accessing the shared memory is as fast as accessing a register. The GPU can dialog with the CPU via the PCI-Express bus, which has rather high latency time (around 20 $\mu$s). It is thus convenient to reduce as much as possible the number of data transfers between host memory and the GPU. In the following section, we will show the design and implementation of the magic filter convolution described in Sec. II E on NVidia GPUs.

## B. Implementation details

From the GPU parallelism point of view, there is a set of $N$ independent convolutions to be computed. Each of the lines of $n$ elements to be transformed is split in chunks of size $N_e$. Each multiprocessor of the graphic card computes a group of $N_\ell$ different chunks and parallelizes the calculation on its computing units. After the calculation of the convolution values, these $N_e N_\ell$ elements are copied in the corresponding part of the output array, which is transposed with respect to the input. The size of the data fed to each block is identical (such as to avoid block-dependent treatment), hence when $N$ and $n$ are not multiples of $N_\ell$ and $N_e$, some data treated by different blocks may overlap. This fact has no double-counting effect since the overlap is reproduced also in the output array. Figure 1 shows the data distribution on the grid of blocks during the transposition.

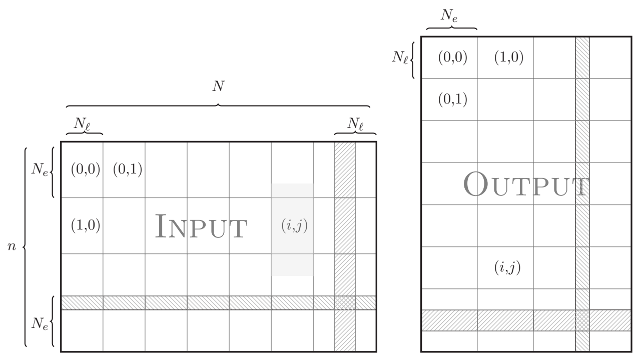

FIG. 1. Data distribution for 1D convolution+transposition on the GPU. Input data (left panel) are ordered along the $N$-axis, while output (right panel) is ordered in the $n$-axis direction, see Eq. (6). When executing GPU convolution kernel, each block $(i,j)$ of the execution grid is associated with a set of $N_\ell$ ($N$-axis) times $N_e$ ($n$-axis) elements. The filled patterns in the figure indicate the overlap region, i.e., data which are associated to more than a block. Behind the $(i,j)$ label, in light gray, it is indicated the portion of data which should be copied to the shared memory to treat the data in the block, which also contains the buffers needed for computing the convolution.

In order to perform the convolution for $N_e$ elements, $N_e+N_{\mathrm{buf}}$ elements must be sent to the shared memory for the calculation, where $N_{\mathrm{buf}}$ depends on the size of the filter. $N_{\mathrm{buf}}$ is equal to $L+U$ for the convolution described in Sec. II E. The shared memory must thus contain buffers to store the data needed for the convolution computations. The desired BCs (periodic in our case) are implemented in the shared memory buffers during the data transfer.

To avoid bank conflicts, the half warp size (16 for modern cards) must be a multiple of $N_\ell$. Each half warp thus computes at least $16/N_\ell$ values and $N_e$ is a multiple of that number, chosen in such a way that the total number of elements $N_\ell(N_e+N_{\mathrm{buf}})$ fits in the shared memory (which has 16 kbyte maximum capacity).

## IV. PERFORMANCE EVALUATION OF GPU CONVOLUTION ROUTINES

In this section, we will evaluate the performance of 1D convolution routines described in Eqs. (7) and (6), together with the analogous operation for the wavelet transformation, by comparing the execution times on CPU and GPU. 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 optimized with optimal loop unrolling and compiler options. The GPU code is compiled with the Intel Fortran compiler (10.1.011) and optimal compiler options (which turned out to be –O2 –xT for our case). With these options, the magic filter convolutions on CPU run at about 3.4 Gflops. All benchmarks are performed with double precision floating point values.

The GPU versions of the 1D convolutions are about one order of magnitude faster than their CPU counterparts. For the magic filter convolutions as well as for the kinetic operator we have a factor of roughly 10, while the GPU wavelet transformation is 20–25 times faster. This fact is mainly related to the CPU implementation of the wavelet transformation, which has a different memory access pattern. Indeed, the values to be convolved for a wavelet transformation belong to two different arrays instead of being contiguous. This fact has no effect on the GPU implementation since the shared memory access does not exhibit cache-miss behavior. 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. These values are lower than the peak performances declared by the vendor, which, for double precision calculations, are of the order of 200 Gflops. A more precise inspection shows that the reason for this fact is that, 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 arithmetic needed to perform convolutions is not heavy enough for hiding the latency of all the memory transfers. This behavior is much more evident for the compression-decompression operations, since it has no arithmetic at all

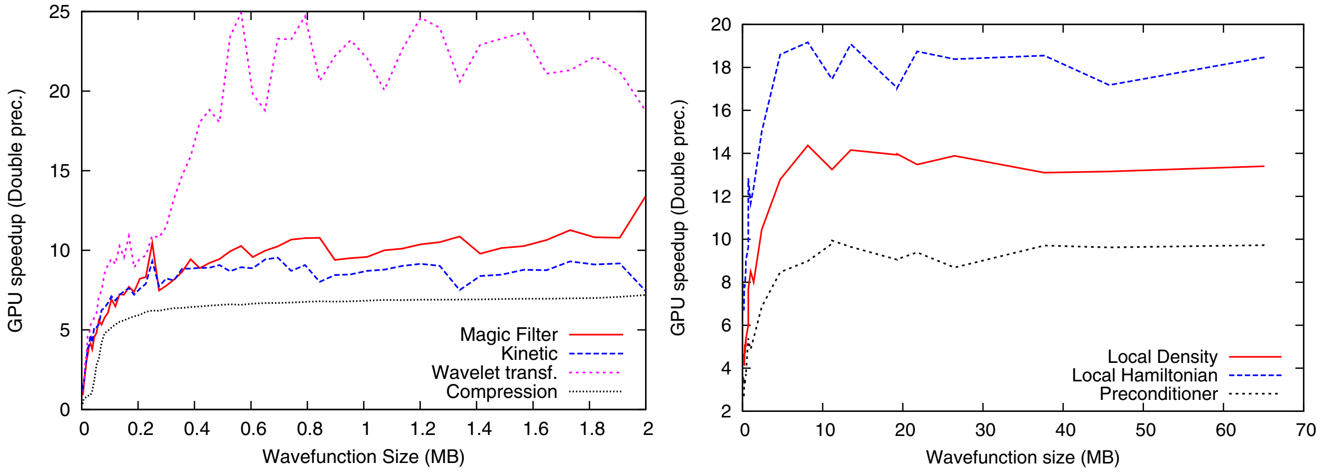

FIG. 2. Double precision speedup for the GPU version of the fundamental operations (left panel) and for the three-dimensional operators used in the BIGDFT code (right panel) as a function of the single KS orbital size.

and is just five to seven times faster on GPU. We will later show that these results are really satisfying for our purposes.

These building blocks must be combined to build the three-dimensional operators (see the list of Sec. II B). The multiplication with the potential and the calculation of the square of the wave function are performed via the application of some special GPU kernels, based on the same guidelines as the others. The reduction operations, such as the calculation of the potential and kinetic energy, can be seen as linear algebra operations and can thus be performed with suitable calls to the corresponding CUBLAS routines.

The performance of the three-dimensional operations, which are based on convolutions, is rather high. We obtain speedups between a factor slightly below 20 for the application of the local Hamiltonian and a factor of 10 for the preconditioner operator. This is related to the fact that, in the latter, the relative importance of the compression-decompression operations is higher, while this becomes almost negligible for the former. Intermediate performances are obtained for the local density construction, which exhibits an overall speedup of about 15. The results for the GPU speedups of the 1D building blocks and for the three-dimensional operators of the BIGDFT code are represented in Fig. 2 as a function of the compressed wave function size.

## V. TOWARD A COMPLETE HYBRID CODE

The memory transfers between memories of CPU and GPU using the PCI-Express bus are well known to be potentially a bottleneck and, consequently, a major obstacle to high performance. Hence, we have to adapt the algorithm to reduce the memory transfers as much as possible. In our case, this can be carried out thanks to the scheduling of operations in the BIGDFT code. Indeed, an optimization iteration of a single wave function is organized as follows:

(I) Density construction.
(II) Poisson solver→local potential $(V_H+V_{\mathrm{xc}})$.
(III) Local Hamiltonian.
(IV) Nonlocal Hamiltonian.
(V) Wave function residue.
(VI) Residue preconditioning.
(VII) Wave function update.
(VIII) Orthogonalization (Cholesky factorization).

The wave functions do not evolve between steps (I) and (III). Therefore, they can be transferred on the GPU (global memory) before computing the density. After step (III), they can be sent back to the host memory (CPU), thus saving two memory transfers. Moreover, in this way, computation time is saved since the operations needed for the density calculation coincide with the first part of the operations for the local Hamiltonian (see Sec. II C). Given the above scheduling of operations, the full CPU-GPU implementation of BIGDFT code can be efficiently implemented. In the current hybrid implementation, we can execute on the GPU steps (I), (III), and (VI) and also all BLAS routines performed in steps (V) (DGEMM) and (VIII) (DSYRK). However, all other operations, such as LAPACK routines [step (VIII)] or the multiplication with the nonlocal pseudopotentials [step (IV)] are still executed on the CPU and can be ported on GPU. We left these implementations to future versions of the hybrid code.

### A. Parallel distribution

Hybrid architectures are becoming more and more popular with configurations of several CPU and GPU. Typically, a configuration may be composed of two quadcore processors (Intel Xeon or Nehalem) and two NVidia GPUs. Hence, in this case, the two GPUs have to be shared between the eight CPU cores. The problem of data distribution and load balancing is a major issue to achieve optimal performance. The operators implemented on GPU are parallelized within the orbital distribution scheme (see Sec. II D). This means that each core may host a subset of the orbitals and apply the operators of Sec. IV only to these wave functions.

A possible solution to the GPU sharing is to dedicate statically one GPU to one CPU core. Two CPU cores are thus more powerful because they have access to GPU. The six other CPU cores do not interact with the GPU. Since the number of orbitals, which may be assigned to each core, can be adjusted, a possible way to handle the inhomogeneity CPU/GPU would be to assign more orbitals to the cores that

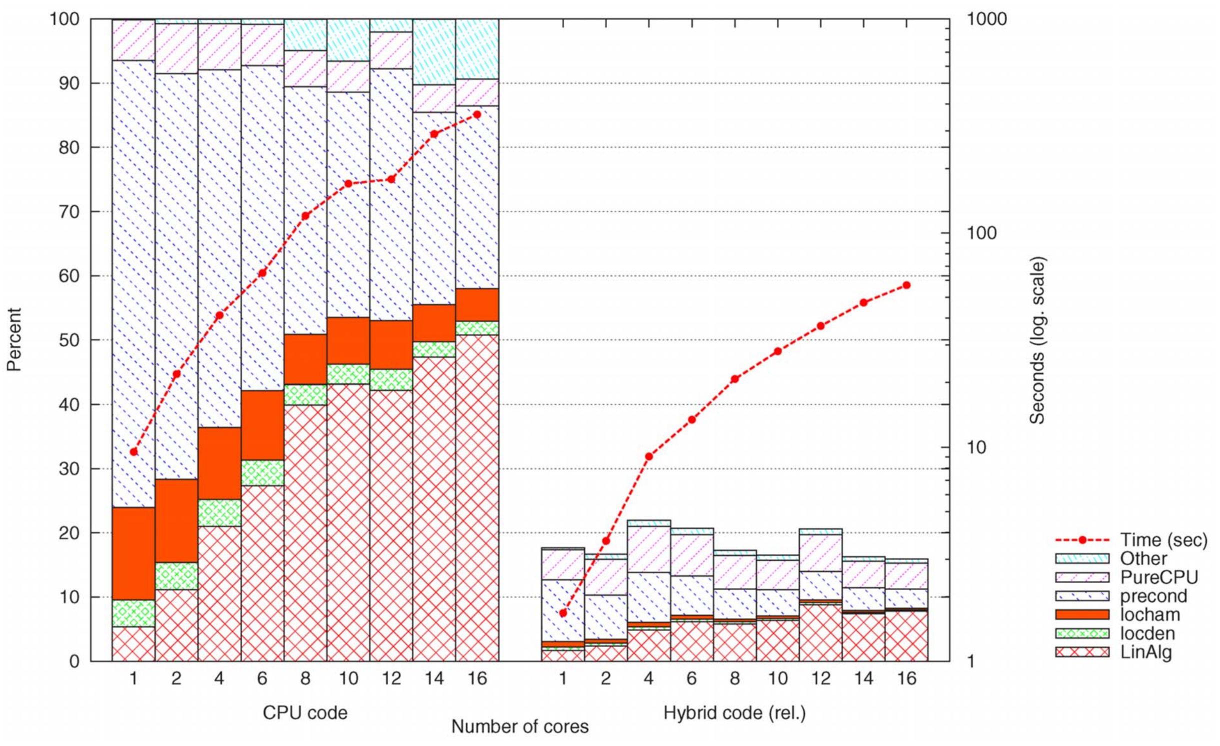

FIG. 3. Relative speedup of the hybrid DFT code with respect to the equivalent pure CPU run, as a function of the number of orbitals. The calculation is performed in parallel such that each CPU core holds the same number of orbitals (36 in this figure). The number of atoms of each system is eight times the number of CPU cores considered. Also the time in seconds for a single minimization iteration is indicated, showing a speedup of a factor of around 6 with the hybrid CPU-GPU architecture, in double precision computation.

have a GPU associated. Such kind of approach can be realized thanks to the flexibility of the data distribution scheme of BIGDFT (see Sec. II D), but it may be difficult for the end user to optimally define a repartition of the orbitals between the different cores for a given system.

For this reason, we designed an alternative approach where the GPU are completely shared by all CPU cores. The major consequence is that the set of orbitals is equally distributed to CPU cores. In brief, in a given node each of the CPU cores is allowed to use one of the GPU cards, so that a card is associated with a group of CPU cores. A GPU is then associated to two semaphores, which control the memory transfers and the calculations. In this way the memory transfers of the data between the different cores and the card can be executed asynchronously and overlapped with the calculations. This is optimal since each orbital is processed independently. The technical details of this implementation will be described in a more technical paper. In the next section, we will focus on the performance of the hybrid BIGDFT code.

## VI. PERFORMANCE EVALUATION OF HYBRID CODE

As a test system for the full DFT code, we used the ZnO crystal, which has a wurtzite bulklike structure. Such a 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. Our calculations are performed on the hybrid part of the CINES IBLIS machine, which has 12 nodes, connected to an Infiniband 4X DDR connectX network, each node (2 Xeon X5472 quadricore) connected with 2 GPU of a tesla S1070 card.

We performed two different tests. In the first one, we performed a set of calculations for different supercells with increasing number of MPI processes, so that the number of orbital per MPI process is kept constant. We performed a comparison for the same runs in which the total number of GPU used equals the number of MPI processes, thus with a 1/1 ratio GPU/CPU cores. The aim of this test is to verify the overall speedup of the hybrid code, considering that not all the computational operations of the code are ported on the card.

The second test is conceived to verify the behavior of the hybrid code where the GPU/CPU ratio is lower than one. In this test, for a given system size, we controlled the behavior for 1/2 and a 1/4 ratios by using the shared GPU repartition described at the end of Sec. V A. In these runs, a GPU can be viewed as a coprocessor, which is shared between two and four CPU cores, respectively. This could be carried out thanks to the data transfer overlap, which allows for multiple cores to share the same card.

### A. One-to-one CPU/GPU binding

Figure 3 shows the results of the first test. Our comparison shows an overall speedup of the whole code by a factor of around 6. As expected, we can see in Fig. 4 that the speedup of the different GPU-ported parts of the code is in agreement with the performances of the separate three-dimensional operators presented in Sec. IV.

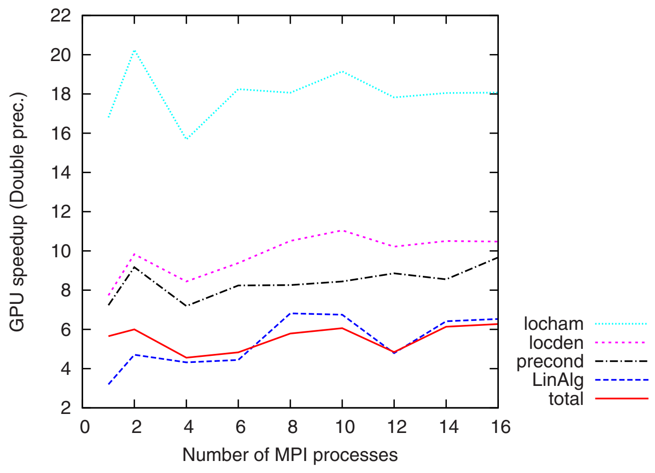

FIG. 4. Speedup of the GPU-related sections of the code for the systems of Fig. 3. The linear algebra operations indicated by the "LinAlg" curve are only partially ported on GPU (no LAPACK routines).

These results are interesting and sound very promising for a number of reasons. First of all, as discussed before, not all the routines of the code were ported on GPU. We focus our efforts in particular on the operators that can be written via a convolution. The application of the nonlocal part of the Hamiltonian can also be performed on the GPU and we are planning to do so in further developments. Moreover, the actual implementation of the GPU convolutions can be further optimized. For example, in the preconditioner the wavelet transformation can be combined with the kinetic operator (as it is done in the CPU version) in order to define a faster GPU kernel. This is not negligible since the preconditioner still represents a considerable fraction of the overall time in the GPU version as well. The linear algebra operations can also be further optimized. Currently, only the calls to the BLAS routines were accelerated on the GPU, via suitable calls to the corresponding CUBLAS routines. The LAPACK routines, which are needed to perform the orthogonalization process, can also 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 Ref. 13). Figure 4 shows that for large systems the overall speedup is dominated by the linear algebra operations; an optimization of this section is then crucial for future improvements of the hybrid code.

## B. Many-to-one CPU/GPU binding

For the second test, we performed runs with two different systems, with 64 and 128 atoms, respectively. For each system, we collected the overall speedups of the full DFT code when one GPU card is associated with two or four CPU cores. In Fig. 5, these results are represented as a function of the total number of MPI processes. The speedup is about 3.5 for a 1/4 repartition, while for a 1/2 repartition it tends to be of the same order as the homogeneous case (see Fig. 4). There is thus no need to reserve an entire GPU per MPI process, since the same speedup can be achieved by sharing the card between two cores. In other terms, we can say that during the calculation only the 50% of the GPU is used. According to this, the speedup of a four-to-one repartition is

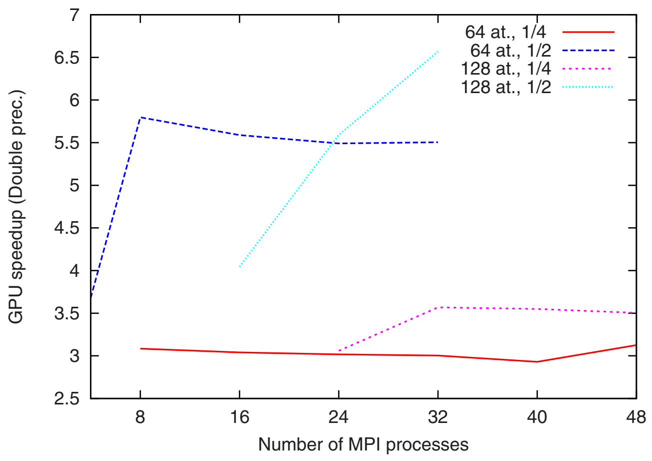

FIG. 5. Speedup of the full DFT code as a function of the number of CPU cores (i.e., MPI processes), when the number of CPU cores associated with the same GPU is two (1/2 curves) or four (1/4 curves). Systems with 64 and 128 atoms are analyzed, showing no significant differences in the behaviors.

thus roughly halved. In particular such a repartition is the more common in modern hybrid CPU-GPU supercomputers (where each node contains eight CPU cores and two GPUs). These results are particularly encouraging since at present only the convolutions operators are desynchronized by the semaphores (see Sec. V A) and the BLAS routines are executed at the same time on the card. Future improvements in this direction may allow us to better optimize the load on the cards in order to further increase the efficiency.

## VII. CONCLUSIONS AND PERSPECTIVES

The port of the principal sections of a electronic structure code over GPUs has been shown. Such GPU sections have been inserted in the complete code in order to have a production DFT code which is able to run in a multi-GPU environment. The DFT code we used, named BIGDFT, is based on Daubechies wavelets and has high systematic convergence properties, excellent performances, and optimal efficiency on parallel computation. The GPU implementation of the code we presented fully respects these properties. With double precision calculations, considerable speedup is achieved for the converted routines (up to a factor of 20 for some operations). Our developments are fully compatible with the existing parallelization of the code and the communication between CPU and GPU does not affect the efficiency of the existing implementation. The data transfers between the CPU and the GPU can be optimized in such a way that more than one CPU core communicates with the same card. This is optimal for modern hybrid supercomputer architectures in which the number of GPU cards is generally smaller (normally a 25% ratio) than the number of CPU cores. We tested our implementation by running systems of variable number of atoms on a 12-node hybrid machine. These developments produce an overall speedup on the whole code of a factor of around 6 for a fully parallel run as well. It should be stressed that, for these runs, our code has no hot-spot operations and all the sections which are ported on GPU contribute to the overall speedup. Moreover, given the nature of the parallelization of the BIGDFT code, we expect these results to also hold on bigger systems in massive parallel hybrid environments, such as the machine installed in France at the Centre de Calcul Recherche et Technologie.

The GPU port of the routines was performed for fully periodic BC, but different BCs can also be implemented without altering the nature of the developments. This is particularly interesting since it means that all these developments are totally compatible with the addition of new functionalities, like a linear scaling approach of the BIGDFT code, which is under way.

Such results can be further improved by optimizing the present GPU routines and by accelerating other sections of the code. The hybrid BIGDFT code, like its pure CPU counterpart, is available under GNU-GPL license and can be downloaded from the site in Ref. 13. To our knowledge, it is the first time that a systematic electronic structure code has been able to run on hybrid (super-) computers in (massively) parallel environment. To summarize, our results open a path toward the use of GPU for double precision DFT calculations and encourage us to proceed further in this direction.

[1] W. Kohn and L. J. Sham, Phys. Rev. **140**, A1133 (1965).
[2] S. Goedecker, Rev. Mod. Phys. **71**, 1085 (1999).
[3] J. Yang, Y. Wang, and Y. Chen, J. Comput. Phys. **221**, 799 (2007).
[4] A. Anderson, W. A. Goddard III, and P. Schröder, Comput. Phys. Commun. **177**, 298 (2007).
[5] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek, Parallel Comput. **33**, 10685 (2007).
[6] K. Yasuda, J. Chem. Theory Comput. **4**, 1230 (2008).
[7] K. Yasuda, J. Comput. Chem. **29**, 334 (2008).
[8] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Beolia, and A. Aspuru-Guzik, J. Phys. Chem. A **112**, 2049 (2008).
[9] I. S. Ufimtsev and T. J. Martinez, J. Chem. Theory Comput. **5**, 1004 (2009).
[10] I. S. Ufimtsev and T. J. Martinez, J. Chem. Theory Comput. **4**, 222 (2008).
[11] NVidia CUDA programming guide, Version 2.1, see http://www.nvidia.com/object/cuda_home.html.
[12] I. Daubechies, *Ten Lectures on Wavelets* (SIAM, Philadelphia, 1992).
[13] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. Alireza Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, A. Bergman, and R. Schneider, J. Chem. Phys. **129**, 014109 (2008); http://inac.cea.fr/sp2m/L_Sim/BigDFT.
[14] X. Gonze, J.-M. Beuken, R. Caracas, F. Detraux, M. Fuchs, G.-M. Rignanese, L. Sindic, M. Verstraete, G. Zerah, F. Jollet, M. Torrent, A. Roy, M. Mikami, Ph. Ghosez, J.-Y. Raty, and D. C. Allan, Comput. Mater. Sci. **25**, 478 (2002); http://www.abinit.org.
[15] S. Goedecker, M. Teter, and J. Hutter, Phys. Rev. B **54**, 1703 (1996).
[16] C. Hartwigsen, S. Goedecker, and J. Hutter, Phys. Rev. B **58**, 3641 (1998).
[17] M. Krack, Theor. Chem. Acc. **114**, 145 (2005).
[18] S. Goedecker, *Wavelets and Their Application for the Solution of Partial Differential Equations* (Presses Polytechniques Universitaires Romandes, Lausanne, Switzerland, 1998).
[19] G. Beylkin, SIAM (Soc. Ind. Appl. Math.) J. Numer. Anal. **29**, 1716 (1992).
[20] A. I. Neelov and S. Goedecker, J. Comput. Phys. **217**, 312 (2006).
[21] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method without the Agonizing Pain" (unpublished) (http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf).
[22] G. Deslauriers and S. Dubuc, Constructive Approx. **5**, 49 (1989).
[23] L. Genovese, T. Deutsch, A. Neelov, S. Goedecker, and G. Beylkin, J. Chem. Phys. **125**, 074105 (2006).
[24] L. Genovese, T. Deutsch, and S. Goedecker, J. Chem. Phys. **127**, 054704 (2007).
[25] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically Intensive Codes* (SIAM, Philadelphia, 2001).
[26] S. Goedecker, Comput. Phys. Commun. **76**, 294 (1993).