The analysis of a plane wave pseudopotential density functional theory code on a GPU machine

Weile Jia\(^a,b\), Zongyan Cao\(^a\), Long Wang\(^a\), Jiyun Fu\(^a,b\), Xuebin Chi\(^a\), Weiguo Gao\(^c,d\), Lin-Wang Wang\(^e,\ast\)

\(^a\) Supercomputing Center, Computer Network Information Center, Chinese Academy of Sciences, Beijing, China
\(^b\) Graduate University of Chinese Academy of Sciences, Beijing, China
\(^c\) School of Mathematical Sciences, Fudan University, Shanghai, China
\(^d\) MOE Key Laboratory of Computational Physical Sciences, Shanghai, China
\(^e\) Material Science Division, Lawrence Berkeley National Laboratory, Berkeley, USA

**Abstract**

Planewave pseudopotential (PWP) density functional theory (DFT) calculation is the most widely used material science simulation, and the PWP DF T codes are arguably the most important material science codes. We have implemented a PWP DFT code PEt on a multi-node GPU machine. Starting from a previous work, we have further improved the speed of the code, and achieved x13-x22 speedups over the CPU calculations for a typical 512 atom system. Such speedups are much higher than other similar works for this important class of material simulation codes on GPU clusters. The current achievement is obtained by (1) moving the calculation fully into the GPU; (2) adopting a new algorithm to reduce the data amount for MPI communication; and (3) using new GPU and CPU numerical libraries. We have also provided a detailed quantitative analysis of the computational times for different physical systems and number of GPU units, which helps one to understand the challenges and bottlenecks of the PWP DFT simulations on GPU machines. Based on the analysis, we listed the machine and library requirements in order to further improve the performances of the PWP DFT calculations.

© 2012 Elsevier B.V. All rights reserved.

**1. Introduction**

This is a time for rapid changes in high performance computing. In order to realize the exascale computation, it would be impractical to simply scale the current ∼1 million cores CPU parallelization to ∼1 billion cores parallelization both from the reliability consideration and the energy usage point of views. It has been realized that some kind of accelerator is necessary. Among the different accelerators, the graphic processing unit (GPU) has gained significant momentum in recent years [1]. Nevertheless, GPU calculation is still in its infancy, both for the computer architectures and for application software. In terms of building up the computer, it is still not clear what the optimal deployment is between CPU and GPU for important scientific applications. For example, what is the requirement for the communication speed, what is the best ratio between the number of CPU and GPU on a node, and what is the requirement for the memory copying speed between CPU and GPU in terms of the application software, the new architecture demands rethinking of the implementation strategy for a given application. The optimal strategies for most major applications on massively CPU parallel architectures have been worked out over the past decades. With the arrival of the CPU/GPU machine, new parallel and multi-thread algorithms have to be tested. All these make the code adaptations to the GPU machines extremely timely and important.

Among the major applications in high performance computing (HPC), the density functional theory (DFT) is the most important one in material simulations [2]. It takes more than 70% of the computer time for material simulations [3], and has contributed to the recent upsurge in \textit{ab initio} simulations. There are different ways to numerically carry out the DFT formalism, and the most popular one is the plane wave pseudopotential (PWP) method [4,5]. In the PWP method, the electron wave functions are described by a linear summation of plane wave basis functions, and the application of the wave function to the Hamiltonian involves Fast Fourier Transformation (FFT), and nonlocal potential projectors [6]. One of the most widely used PWP DFT codes is VASP. An x7 speedup has been achieved for VASP on a single NVIDIA Tesla C2050 GPU for a small physical system [7]. However, no multiple GPU parallelization has been done in that work, and other preliminary works by other groups for VASP and other PWP codes on multiple GPU have so far achieved less impressive...
results (e.g., with speed up less than x2). On the other hand, some quantum chemistry codes have achieved higher speedups on GPU machines with the benefit of using localized basis sets [8]. For example, TeraChem has claimed more than 100× speedups using GPU [9]. BigDFT, which is a DFT code based on wavelet theory, has also reached a x6 speedup on a multi-GPU cluster [10]. However, no PWP DFT code has reached high speedup on GPU platforms, especially for 100–1000 atom systems. This raises a question of whether PWP DFT code can be significantly improved using GPUs, especially because the FFT is essential for such calculations.

In the current work, we have implemented the PWP code PEtot [11] on a GPU machine based on a Fortran/MPI/CUDA programming scheme. Compared to the CPU version of the PEtot code, we have achieved x13–x22 speedups. This is a major improvement on our previous work [6] which has x6–x10 speedups over the CPU calculations. This improvement is achieved by (1) moving the calculation fully into the GPU; (2) adopting a new algorithm to reduce the amount of data in MPI communication; and (3) using new GPU and CPU numerical libraries. More importantly, in this work, we will provide a quantitative performance analysis for the total computational time for any given physical problems and any number of CPU/GPU units. This can help people to identify the remaining bottlenecks of the calculations, and provide strategies and hardware requirements for further improvements of the calculations.

Our GPU implementation of the PEtot code is based on a G-space and band index hybrid parallelization scheme. In this scheme, the wave functions \( \{ \Psi(G) \} \) are parallelized along the band index “i” when they are applied to the Hamiltonian \( H : H \Psi_i \) thus the FFT and nonlocal potential projection can be evaluated on a single CPU/GPU computing unit. On the other hand, they are parallelized among the reciprocal G-space coefficients (plane wave coefficients) when the overlap matrix like \( \langle \Psi_i | \Psi_j \rangle \) needs to be calculated. The detail of the algorithm is described in Ref. [6]. The focus of the calculations is for systems with up to a few thousand atoms. The argument is that, for systems much larger than a few thousand atoms, some kind of linear scaling methods should take over. Actually, in the divide-and-conquer linear scaling 3 dimensional fragment (LS3DF) method [12], the large system (>1000 atoms) calculation is built on top of the smaller fragment system calculations, thus the GPU code developed here can be directly used in LS3DF calculations.

While in our previous work [6], the main point was to demonstrate the feasibility of using GPU to significantly speed up the PWP DFT calculations, in the current work, we have three focuses. The first is to demonstrate the possibility to further speed up the PWP DFT GPU calculation to the regime of x20 times compared with the CPU codes. This is significant since the GPU card is often about 10 times more expensive than the CPU core. Thus achieving this x20 fold increase means it is economically cheaper to run on the GPU machine, notwithstanding that the absolute speed is perhaps a more important achievement. The second focus is to perform a quantitative analysis for the total computational time of the GPU code, so other people can use this model to analyze their problems. The third focus is to list the architecture and software library requirements in order to further speed up such calculations. This is important for the hardware engineers because this is the time the architectures of GPU machines are shaping up.

Through our analysis, we come up with the following observations and machine requirements: (1) The MPI communication becomes the bottleneck, which does not scale with the increasing number of CPU/GPU tasks (units). Thus, it will be of utmost importance to speed up the MPI communications for both inter and inner nodes; this can be achieved by different interconnect topology (e.g., an all-to-all connection between nodes), or by software library improvement (e.g., faster MPI communication inside one shared memory node). (2) The memory copy between CPU and GPU becomes expensive. This raises the issue of using the GPU as a mere accelerator, where inevitably the data need to be copied frequently in and out of the GPU, while the book-keeping is done inside the CPU. In our implementation, we have moved almost all the calculations inside the GPU, which reduces the CPU/GPU communications. But this also treats the GPU as our main computing and book keeping resource, not merely as an accelerator. It will thus be helpful for direct access of MPI communication inside the GPU, much like what is planned by NVIDIA Inc. (3) It is essential to have a large GPU global memory. As we move the calculation into GPU, it also becomes necessary to keep all the data inside GPU. (4) It is necessary to have a better CPU/GPU library for linear algebra, including matrix diagonalization and decomposition. This becomes exceedingly difficult when the matrix size is small, e.g., around 1000 as in our case. Nevertheless, such a small size matrix diagonalization has taken a large portion of the time in our calculation. If further improvement in this part is not possible, some other algorithm changes might be used to speed up this part. For example, one might want to use iterative methods to diagonalize or decompose the matrix [13]. (5) It might be necessary to test the MPI/OpenMP/CUDA parallelization scheme. In our current implementation, we have used a MPI/CUDA scheme. This is ideal for our machine where each GPU will have one CPU. But for many other machines, there might be many more CPU cores than GPU cards within a shared memory node. In such cases, it might be necessary to have a MPI/OpenMP/CUDA programming scheme. But there could also be different ways to implement such schemes, especially to improve the inner node communications. (6) Finally, to maximize the speed for PWP DFT calculations, one should have as many GPU cards as possible within a node, although as we will discuss later, there could be issues of CPU–GPU bus contentsions.

Overall, we found that the GPU clusters can be powerful machines for PWP DFT calculations. One problem of the PWP DFT calculation on the purely CPU machine is its inability to scale to large number of CPUs, especially for the type of problems we are considering here. This makes it difficult to study problems which require long time molecular dynamics simulations. How to speed up such simulations in temporal scale using supercomputers is one of the most urgent problems facing the material science simulation community. Using the CPU, the fastest MD dynamics that can be achieved is about 2–3 min per MD steps for a few hundred to one thousand atom system. This is regardless of how many CPU processors one can use. Using GPU, this time can be reduced by 20 folds as demonstrated in the current paper. Further improvement on the computer architecture and software library might make it possible to have a 50–100 fold speedup, thus to carry out one MD simulation step within a few seconds. That will certainly have a major impact on ab initio material science simulations.

2. DFT algorithm and parallelization schemes on CPU and GPU clusters

2.1. DFT calculation

A modern DFT calculation solves a system with a self-consistent iteration [4]. It begins with the wave function improvement based on the Kohn–Sham equation \( H \Psi_i = \varepsilon_i \Psi_i \), and \( H \) is the single particle Hamiltonian which depends on \( \{ \Psi_i \} \) via a self-consistent step. After the wave functions \( \{ \Psi_i \} \) are solved from \( H \Psi_i = \varepsilon_i \Psi_i \) for a fixed \( H \), the charge density \( p(r) \) is calculated by occupying the wave functions. Then Coulomb potential \( V_{\text{Coul}} \) is calculated from \( p(r) \) using a Poisson solver and exchange correlation potential \( V_{\text{xc}} \) is calculated using the local density approximation (LDA) or general gradient approximation (GGA). This leads to a new potential: \( V(r) = V_{\text{Coul}}(r) + V_{\text{xc}}(p(r)) \). This output potential \( V(r) \) will be compared with the input potential for its self-consistency. If they do not agree, a potential mixing scheme is used to generate a new
in the plane wave calculation, the wave function $\Psi_i$ is expanded by plane wave basis set. Each plane wave function corresponds to one reciprocal lattice point of the real space periodic box, and these lattice points are within a G-space sphere in the FFT grid. On the other hand, the real space FFT grid is a full 3D box with $n_1, n_2, n_3$ dimensions. The G-space parallelization scheme does the FFT within the $n$-nodes processors. The FFT is optimized for the spherical G-space data by load balancing and minimizing the communication between processors [14]. The nonlocal potential projection operator $\sum_{i} \langle \Phi_i | \Phi_j \rangle$ will be calculated in real space within a sphere for each atom [15]. In the PEtot CPU code, all three levels of parallelizations are implemented, and they can be used simultaneously.

2.3. Previous GPU PWP DFT parallelization scheme

In our previous work, a G-space and band-index hybrid interchange parallelization scheme has been adopted in a GPU PEtot code [6]. A G-space parallelization is used in overlap matrix calculation parts like Sub_diag, Projection and Orth shown in Fig. 1, while a band-index parallelization is used to calculate $H\psi$, thus the FFT and nonlocal potential projection can be evaluated in a single CPU/GPU unit. The wave function book-keeping is done in the CPU. As a result, each time an overlapping matrix needs to be calculated, we copy the corresponding wave functions $|\Psi_i\rangle, |P_i\rangle$, or $|\epsilon_i \Psi_i\rangle$ from the CPU to the GPU. Matrix multiplication is then done by calling CUBLAS-3 routine CUBLAS_ZGEMM. The calculated matrix with the size of $M_x \times M_y$ (where $M_y = 1025$ in our test case is the total number of the wave functions), will be copied from the GPU to the CPU for an MPI_allreduce (G-space parallelization makes the partial products of $\langle \Psi_i | \Psi_j \rangle$ distributed among the CPU/GPU computing units). Diagonalization zheev or Cholesky decomposition zpotrf is performed on the resulting matrix on the CPU, and then results will be copied to the GPU for wave function rotations or projection for $P_i$. In the calculation of Hpsi, the G-space distributed wave functions $\{P_i\}$ need to be transposed to band-index parallelization using an MPI_alltoall call. This ensures each CPU/GPU computing unit holds a few full wave functions. In this way, FFT is calculated using a standard three-dimensional CUFFT call (by mapping the G-space coefficients within the sphere into a full 3D grid and padding the outside of the sphere with zeros). Real space nonlocal potential projection $\langle \Phi_i | \Psi_j \rangle$ (to be called Nonlocal) is done after the inverse FFT on $\Psi_i$. The FFT and Nonlocal calculations are done in a band-by-band manner. According to our test, MPI_alltoall consumes more than 50% of the Hpsi time when large numbers of CPU/GPU computing units (128 or 256) are used. We note that the overall parallelization scheme used here is similar to that used in the BigDFT code [16].

3. The improvements on the DFT GPU implementations

The details of the PWP DFT GPU implementation have been described in the Ref. [6]. Here we will focus on the three main changes in the current work: (1) moving AB–CG into GPU; (2) reducing MPI_alltoall communication; (3) using the best math libraries for matrix diagonalization and decomposition.

GPU computation power is enormous, but the CPU–GPU PCI-Express connection is rather slow. This means that we should put more computation into the GPU and reduce the CPU–GPU memory copy. In our current implementation, we have moved the AB–CG fully into GPU by keeping the wave functions inside the GPU memory as shown in Fig. 2. Each Tesla C2050 GPU card has 3 GB global memory and it is enough to hold several wave functions as discussed in the Ref. [6]. As a matter of fact, we can calculate up to ~4000 atom systems in this way with enough CPU/GPU units. Updates on wave functions in Sub_diag,
Projection and Orth are directly performed inside the GPU with data from the GPU memory. The only part that still requires wave function CPU–GPU copy is before and after $H\Psi$ the operation for the MPI_Alltoall operations. The wave functions are book kept in $G$-space parallelization during the CG iteration, and the $H\Psi$ operation is done in band index parallelization. Thus, we need to do a wave function transposition between $G$-space and band-index parallelization using MPI_Alltoall as shown in Fig. 2. Book-keeping of wave functions on the CPU is no longer needed. Precondition and line minimization are also moved into the GPU by hand-coded CUDA code, which also gives us a gain in the speed. Overall, keeping wave functions inside the GPU throughout AB–CG gives us improvements in both computation and CPU–GPU memory copy.

MPI_Alltoall is a bottleneck in the GPU PETot code, yet it is unavoidable to the hybrid $G$-space and band-index parallelization scheme. To reduce the communication between CPUs, a compression algorithm is introduced for the Hpsi calculations. Although not explicit in Fig. 1, the $H\Psi$ in the line “do loop” in Fig. 1 is actually carried out through $H_p$, where the $P$ is the wave function residuals $P = H\Psi - s\Psi$ from the last iteration (or say, $P$ is related to the change of $\Psi$ for each “do loop” iteration through the “line minimization” step shown in Fig. 1). Note that, $P$ is always very small, and the purpose of the AB–CG is to reduce $P$ further (e.g., by a factor of 10). Thus, it is not necessary to describe $P$ with high precision. That means we don’t need to keep as many bits as double precision complex for both the exponent part and fraction part. In the new implementation, $P$ is converted from 16 bytes double complex to 4 bytes compressed digit, which reduces the MPI_Alltoall data to 25% of its original amount. Furthermore, the precision reduction on $P$ makes it possible to calculate the FFT and Nonlocal potential in single precision floating point operations, which further speeds up the calculation in the GPU. The compression does not affect the precision of the final result, nor does it affect the convergence rate as shown in Fig. 3. The detailed behavior and the underlying mathematical theory of this new algorithm will be published somewhere else. It is sufficient to say that our algorithm is a mixed precisions algorithm, and its final result is in double precision and our convergence rate is not affected by the data compression.

Diagonalization of matrix $h(i, j)$ using zheev and Cholesky decomposition on the overlap matrix $(\Psi_i | \Psi_j)$ using zpotrf consumes a significant amount of time in the PETot code. It is critical to reduce these times because they do not scale well with the number of processors. Different new CPU and GPU libraries are tested for this purpose as shown in Fig. 4. Based on these tests, we have chosen ELPA_SOLVE_EVP for matrix diagonalization using up to 64 CPUs; single GPU MAGMA_ZPOTRF and CUBLAS_ZTRSM for Cholesky decomposition and wave function rotations respectively.

4. Testing results

4.1. GPU testing machine

The test platform is the Mole-8.5 GPU cluster in the Institute of Process Engineering of Chinese Academy of Sciences, which has 1012.65Tflops peak performance and ranks 37th in the June 2012 Top500 list. This is the same machine as the one used in the work of Ref. [6], but with upgrades in its operation systems and some of the software libraries. NVIDIA CUDA 4.0 toolkit with 270.41.19 NVIDIA drivers is installed. We used Intel C/Fortran compiler with the version 12.1.2 as the basic compiler in our tests. For MPI communication routines, we used OpenMPI 1.4.4 implementation binding with the Intel compiler. We also selected and installed some high performance linear algebra libraries for testing; they are Intel MKL 10.3.8, CULA R11, MAGMA 1.0.0 and ELPA (git version) [17,18]. Note that due to these updates, our current test times for both CPU and GPU results are slightly faster than the original results reported in Ref. [6] even for the same codes.

Hardware configuration of the Mole-8.5 machine is as follows: it has 320 computing nodes. Each node has 2 quad-core CPUs and 6 GPU cards. The CPU is Intel Xeon L5520, based on Nehalem microarchitecture, running at 2.26 GHz, with 9.06Gflops theoretical speed. In each computing node, 8 CPU cores share 48 GB DDR3 system memory through the QuickPath Interconnect (QPI). The GPU card in the computing node is NVIDIA Tesla C2050. One card has 448 stream processors and provides a peak performance of 1.03Tflops for single precision operation and 515Gflops for double precision operation. The 6 GPU cards are connected with the 8 CPU cores by PCI-Express bus lanes. Each GPU card has its own 3 GB global memory. Computing nodes are connected with QDR Infiniband provided by Mellanox.
In our test runs, each MPI process was bound to one CPU core and attached to a single GPU card. So we counted one CPU core plus one GPU card as one CPU/GPU computing unit. Two CPU cores in one node will be idle during the operation. We have access to 44 computing nodes, not the whole Mole-8.5 machine, thus our tests are limited to 256 CPU/GPU units.

4.2. The calculated system

Our main testing system is a 512 atom GaAs bulk with one As atom being replaced by one N atom forming one isovalent impurity [19]. This is the same system as the one in the Ref. [6]. As discussed in the Ref. [6], using our parallelization scheme, the maximum system which can be calculated in this way (given a sufficiently large machine) is likely to be around 4000 atoms due to GPU memory limitation. Thus the current system is a relatively small one, which makes the speedup more challenging as the matrix–matrix multiplication plays a relatively small role compared with the MPI communications. Note that if the size of a matrix is $M$, the multiplication scales as $M^3$, while the communication scales as $M^2$. Nevertheless, the size of the current system represents a big class of problems typically calculated in present day material science simulations.

The plane wave kinetic energy cutoff is $E_c = 40$ Ryd, and the real space grid is $128^3$. There are 1024 occupied electron states, and we have used $M_e = 1025$ as the number of calculated wave functions. The total number of plane waves $N_G$ within the kinetic energy cutoff sphere is $3.3 \times 10^5$. Norm conserving pseudopotentials are used, and the nonlocal potential projections are carried out in real space. For Ga and N atoms, there are $8\Phi_l$ nonlocal projectors for each atom, while for As, there are $4\Phi_l$ nonlocal projectors.

4.3. The total computational time

We will concentrate on the all band conjugated gradient (AB–CG) routine as depicted in Figs. 1 and 2 for solving the Schrodinger’s equation $H\Psi_i = \varepsilon_i \Psi_i$, as this part can take 98% of the computational time for large systems. Note that the AB–CG is not the only algorithm to solve this equation, but it is a typical one used in such PWP DFT code. The total computational time for the AB–CG with 4 CG steps (line minimizations) (4 lines are the typical number of steps used in real calculations) are listed in Table 1 using from 16 CPU/GPU to 256 CPU/GPU computing units (again, one computing unit is one CPU core with one GPU card). From Table 1, we can see that the current implementation of the PEtot_GPU code has a speedup over the PEtot_CPU code by a factor of 13 to 22. It increases the speedup over the PEtot_GPU0 reported in Ref. [6] by a factor of 2 due to the improvements discussed above.

4.4. The decomposition of the computation time

The total computation time can be divided into three parts: the numerical operation time, the MPI communication time, the CPU–GPU data copy time. The numerical operation time includes: the FFT using CUFFT in a single GPU, the nonlocal potential projection implemented by in-house CUDA code in a single GPU, the matrix–matrix multiplication using CUBLAS_ZGEMM, the linear equation solution using CUBLAS_ZTRSM, the data compression to shorten the digits of residual $P$ by an in-house GPU code, and some miscellaneous operations which will be summarized under the name of “Precond and line minimization”. Besides, we also have matrix diagonalization (zheev) and Cholesky decomposition.
<table>
<thead>
<tr>
<th>Number of cores</th>
<th>System</th>
<th>PEtot (CPU)</th>
<th>PEtot_GPU0 (GPU)</th>
<th>PEtot_GPU (GPU)</th>
<th>Speed-up (with PEtot CPU)</th>
<th>Total flops (Tflops)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>512-GaAs</td>
<td>735</td>
<td>87.3</td>
<td>34.06</td>
<td>221</td>
<td>1.51</td>
<td>17.14%</td>
</tr>
<tr>
<td>32</td>
<td>512-GaAs</td>
<td>596</td>
<td>55.9</td>
<td>19.48</td>
<td>118.3</td>
<td>2.94</td>
<td>16.69%</td>
</tr>
<tr>
<td>64</td>
<td>512-GaAs</td>
<td>396</td>
<td>33.3</td>
<td>7.21</td>
<td>7.13</td>
<td>4.47</td>
<td>12.69%</td>
</tr>
<tr>
<td>128</td>
<td>512-GaAs</td>
<td>221</td>
<td>23.4</td>
<td>6.79</td>
<td>7.15</td>
<td>18.69</td>
<td>10.15%</td>
</tr>
<tr>
<td>256</td>
<td>512-GaAs</td>
<td>128</td>
<td>19.8</td>
<td>6.79</td>
<td>7.39</td>
<td>87.3</td>
<td>5.39%</td>
</tr>
</tbody>
</table>

The coefficients in the formulas in Table 2 are directly related to machine parameters (the GPU flops, the communication bandwidth and latency, the GPU–CPU connection bandwidth). More specifically, the coefficients $\alpha_1$, $\alpha_2$, $\alpha_7$, $\alpha_8$, and $\alpha_9$ should be roughly proportional to the peak performance of the GPU since they correspond to floating point operations. The coefficients $\alpha_3$ and $\alpha_4$ depend on the bandwidth between two nodes while $\beta_3$ and $\beta_4$ reflect the communication latencies and overheads. We have to keep in mind that in real calculation the actual communication performance not only depends on the theoretical intra-node or inter-node bandwidths and latencies, but is also influenced by the communication pattern and the potential network contentions. Thus, among these formulas, their derivations and meanings are obvious. Some of the meanings of the communication formulas will be discussed in detail in Section 6.

The coefficients in the formulas for different physical systems and numbers of computing units, we break up the total AB–CG time into different kernels, as shown in Fig. 5. Then we write down the formula for the computational time for each kernel depending on the system sizes and number of computing units $N_p$. The formulas are shown in Table 2. For most of these formulas, their derivations and meanings are obvious. Some of the meanings of the communication formulas will be discussed in detail in Section 6.

The comparison of the tested computing time for all the CUBLAS routines and the estimated time according to the $T_{\text{CUBLAS}}$ formula in Table 2. The line is a visual guide for $y = x$.

The comparison of the tested communication times for all the MPI_Alltoall calls and the estimated times according to the $T_{\text{Alltoall}}$ formula in Table 2. Times, representing numerical operation and communication respectively.

As we can see, the formulas with fitted coefficients can describe the actual computing times within about a 20% error. In Fig. 8, we also show the total computational times predicted from our times, representing numerical operation and communication respectively.
The computational times for different kernels of the calculations. The total AB–CG time is the sum of all these times (rows). The units for the parameters are seconds. $N = n_1 + n_2 + n_3$, where $n_1$, $n_2$, $n_3$ are the real space FFT size, $M_x$ is the number of wave functions calculated and $N_p$ is the number of CPU/GPU computing units. $N_{\text{atoms}}$ is the total number of atoms. $N_G$ is the number of plane wave coefficients in the whole system, which equals $(E_r^{1/2} Q/8\pi^3)$, and $E_r$ is the kinetic energy cutoff. $Q$ is the volume of the supercell (all in atomic units). $N_n$ is the number of grid points within each real space nonlocal projector, which equals: $4\pi r_t^2 N/(32)$, and $r_t$ is the real space cutoff radius for the nonlocal projector. Note all the coefficients correspond to 4 line minimizations in one AB–CG step.

<table>
<thead>
<tr>
<th>Time</th>
<th>Formula</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{FFT}}$</td>
<td>$\alpha_1 \cdot (\log N \cdot M_x) / N_p$</td>
<td>$\alpha_1 = 3.0e-9$</td>
</tr>
<tr>
<td>$T_{\text{Scal}}$</td>
<td>$\alpha_2 \cdot \frac{M_x}{N_p}$</td>
<td>$\alpha_2 = 3.8e-8$</td>
</tr>
<tr>
<td>$T_{\text{Allscal}}$</td>
<td>$\alpha_3 \cdot (N_p - 1) \cdot \frac{M_x}{N_p} + \alpha_3 \cdot (N_p - 1)$</td>
<td>$\alpha_3 = 1.7e-7$</td>
</tr>
<tr>
<td>$T_{\text{Allreduce}}$</td>
<td>$\alpha_4 \cdot (2N_p - 1) \cdot \frac{M_x^2}{N_p} + \alpha_4 \cdot (2N_p - 1)$</td>
<td>$\beta_4 = 6.7e-4$</td>
</tr>
<tr>
<td>$T_{\text{Matr_copy}}$</td>
<td>$\alpha_5 \cdot \frac{M_x}{N_p}$</td>
<td>$\alpha_5 = 1.5e-7$</td>
</tr>
<tr>
<td>$T_{\text{CUBLAS}}$</td>
<td>$\alpha_6 \cdot \frac{M_x^2}{N_p}$</td>
<td>$\alpha_6 = 3.3e-7$</td>
</tr>
<tr>
<td>$T_{\text{Precond and line}}$</td>
<td>$\alpha_7 \cdot \frac{N_G}{N_p}$</td>
<td>$\alpha_7 = 3.8e-10$</td>
</tr>
<tr>
<td>$T_{\text{CUBLAS}}$</td>
<td>$\alpha_8 \cdot \frac{N_G}{N_p}$</td>
<td>$\alpha_8 = 7.3e-9$</td>
</tr>
<tr>
<td>$T_{\text{Compress}}$</td>
<td>$\alpha_9 \cdot \frac{M_x}{N_p}$</td>
<td>$\alpha_9 = 5.9e-8$</td>
</tr>
<tr>
<td>$T_{\text{Mollic and Free}}$</td>
<td>$\alpha_{10} \cdot \frac{N_G}{N_p} + \beta_{10}$</td>
<td>$\beta_{10} = 2.7e-8$</td>
</tr>
<tr>
<td>$T_{\text{Zheev}}$</td>
<td>$\alpha_{11} \cdot \frac{M_x^3}{N_p^2}$</td>
<td>$\alpha_{11} = 7.6e-10$</td>
</tr>
</tbody>
</table>

Fig. 8. The comparison of the tested total computing times for the three tested systems (512, 256, 128 atoms) with a different number of GPU/CPU processors and the estimated total time by summing up all the terms in Table 2. It shows that AB–CG times calculated from the fitted formulas match the real test results well.

Fig. 9 shows the predicted computational times when the system size increases from 512 atoms to 1024 atoms, and to 2048 atoms. We see that the minimum time with $N_p$ happens when the number of processors is roughly equal to the number of atoms in the system. We have calculated a 1024 atom system to test the prediction accuracy. The predicted times agree well with the cases where we have enough CPU/GPU units to calculate it. For the 2048 atom system, we do not have enough CPU/GPU units to calculate this system. Nevertheless, our formulas predict that it will take about 100 s to finish the AB–CG calculations. Thus further improvement is needed, especially to reduce the $\beta$ related terms in Table 2 which makes the calculation nonscalable.

5.2. The scaling of the computation times

In Fig. 10, we show the scalings of the computational times for various numerical operation kernels. We see that some parts scale well with the number of computation units $N_p$, while other parts do not scale. The scaling parts refer to the calculations proportional to the number of wave functions $M_x \cdot n = M_x / N_p$ in one CPU/GPU unit (FFT, nonlocal, data compression) or number of plane wave coefficients $n_G \cdot n = N_G / N_p$ in one CPU/GPU unit (CUBLAS_ZGEMM, CUBLAS_ZTRSM, preccond and line min). The nonscaling parts include the diagonalization (zheev) and Cholesky decomposition (zpotrf) of the $S(M_x, M_x)$ matrix. All these scalings can also be seen from Table 2 from the formula. As shown in Table 2, most numerical operation kernels scale with $N_p$, except the zheev and zpotrf. As we can see in Fig. 5, for the total computational time, the nonscaling zheev and zpotrf have become a significant
part of the total time, almost one fifth of the total value, and the situation will become worse if \( N_p \) is increased further as predicted by the formulas in Table 2.

The MPI communication time includes the MPI_Alltoall time and the MPI_Allreduce time. As shown in Table 2, their computational times have both the scaling part and nonscaling part. The MPI_Alltoall formula is derived based on the pairwise exchange algorithm and the MPI_Allreduce formula is derived based on the ring algorithm. The MPI_Alltoall time model accuracy is shown in Fig. 7. Among all the formulas in Table 2, the MPI_Alltoall and MPI_Allreduce are the two most inaccurate formulas, because their times might depend on the details of the underlying implementation of the MPI library, the hardware architecture of the machine, and the contents during the data communication. According to our formula, for the 512 atom system, when \( N_p \) is larger than 256, the nonscaling part (\( \beta \) related term) becomes dominant for MPI_Alltoall. Beyond that, the times for these parts increase with \( N_p \), which is observed in our actual tests. We see in Fig. 5, for large \( N_p \) counts, the MPI communication takes about almost one half of the total computational time, and the MPI_Alltoall time is slightly increasing with \( N_p \). Such behavior highlights the importance of further improvement on the communication (reduce the \( \beta \) coefficients) by improving the MPI lib or the hardware architecture.

One major improvement of the current code is the adoption of the precision reduction and data compression algorithm on the wave function residual \( P \) as discussed before. With 4 line minimizations, there are 2+8 MPI_Alltoall calls (each \( \Psi \) corresponds to two MPI_Alltoalls, one before and one after the operation as shown in Fig. 2). With the \( P \) digit reduction algorithm, only the first 2 MPI_Alltoall (before the do loop, in Fig. 1) need to use the full 16 bytes double complex numbers, and the remaining 8 MPI_Alltoall can use 4 bytes numbers. This has reduced the total MPI_Alltoall time by a factor of 2.5. Note the data compression from a 16 bytes number to a 4 bytes number is done in the GPU by an in-house developed CUDA code. The computational time for this compression is relatively small. For example, in the 256 CPU case, the MPI_Alltoall time for the case of 4 bytes number is 0.07 s, while each corresponding compression or decompression will take 0.016 s.

As one can see from Fig. 5, another major computational time comes from the data copy (memcpy) between the CPU and GPU. These data copies are shown schematically in Fig. 2, and their times are summarized in Table 2. There are two major types of memory copy operations, one involves the wave functions, which scales with \( N_p \), another is related to the overlap matrix \( S(M_e, M_i) \) which does not scale with \( N_p \). Note, in our case of \( M_e = 1025 \), each memcpy for the matrix takes about 0.03 s, while each wave function memory copy also takes about 0.03 s for 256 CPUs, but 0.45 s for 16 CPUs. Thus, for a relatively small number of CPU/GPU units, it is important to reduce the memory copy of the wave functions. That is another major motivation for moving the calculation and wave function book keeping from CPU to GPU. Thus, the only necessary wave function memcpy operations are for the wave function transpose.

6. Future improvements and machine requirements

We now discuss the possibility of future improvements and the corresponding architecture requirements for the machine. The goal is to see whether the code can be further improved by a factor of 3–5, so it can be 50–100 times faster than the CPU version, and the \textit{ab initio} MD can be calculated with a few seconds per step for a 512 atom system.

It is clear that the time of the scalable part of the numerical operation as discussed in Section 5 and Fig. 10 can be further reduced when we increase the number of CPU/GPU units. This part takes about 2.5 s out of the 6.8 s total time in the 256 CPU/GPU unit calculations. Theoretically we can increase to 1025 computing units until there is only one wave function per unit. That will be likely to reduce the time for this part to 0.6 s. Thus the challenge is at the nonscalable parts.

The first nonscaling part is the MPI_Alltoall communications. In the current work, we have significantly reduced the time of this part by using a reduced precision of the wave function residual \( P \). In Table 2, we used the formula with coefficients \( \alpha_1 \) and \( \beta_1 \) to measure the all-to-all time. In this formula, \( \alpha_1 \) is mainly determined by the bandwidth of the Infiniband, \( \beta_1 \) reflects the latency of the network and the overhead of the communication. We made the analysis of the communication time and give the formula following Ref. [20] and the LogGP model [21]. The formula was built based on the pairwise algorithm in all-to-all implementation. In OpenMPI and other major MPI implementations, pairwise is commonly used for large message all-to-all communication, while some other algorithms such as the Bruck algorithm [22] should be used for short messages. In OpenMPI 1.4.4, MPI_Alltoall will choose a pairwise algorithm while the data size is larger than 3000 bytes. In our calculations, \( N_p * M_e / N_d \) data size is definitely larger than 3000 bytes. In AB–CG calculation, we have 2 times full and 8 times compressed wave function all-to-all communications. One complex number has 16 bytes. Since the in Table 2 includes all the 10 MPI_Alltoall communications (with 2 full, and 8 reduced as 4 bytes number), we can get \( \alpha_1 \) which corresponds to one MPI_Alltoall (and in the unit of bytes): \( \alpha_1 = \alpha_1 / (2 + 0.25 \times 8) = 2.70 \) e/\( \text{Byte} \). Now \( \alpha_1 \) reflects the Infiniband bandwidth and the communication efficiency. It indicates that one processor got \( 1/2.70 = 0.37 \) GB/s communication bandwidth in our code. We run 6 processes on one node with the bandwidth for the node as 4 GB/s, so if network performance is balanced for all processes in each node, one process could theoretically get 4/6 = 0.67 GB/s bandwidth, thus we have 55% utilization. We can also calculate that \( \beta_1 = \beta_1 / (2 + 8) = 2.92 \) e – 4 s as the latency time for each MPI_Alltoall operation for each process pair. This number is much larger than the point-to-point latency of Infiniband. So there may be a lot of overhead in all-to-all communication. This also indicates that faster interconnects, especially with a physical all- to-all connection with lower latency and overhead can speed this up significantly. Note, currently, it is this overhead and latency time which makes the MPI_Alltoall nonscalable.
The next nonscaling part is the MPI_Allreduce. Our formula in Table 2 is based on the ring algorithm for large messages. In OpenMPI 1.4.4, the ring algorithm is used while the message is larger than 10,000 bytes. So while $M_k$ is small (if the test system became smaller) and $N_g$ is relatively large, the formula may not work well. But in our tests, $M_k^2/N_g$ is always large enough except for the case of a 128 atom system with 256 CPU/GPU units (we have ignored this case while fitting the coefficients $\alpha_4$ and $\beta_4$). Similar to the all-to-all case, $\alpha_4$ is also related with bandwidth and $\beta_4$ is dominated by latency and overhead. An immediate potential speedup is to take advantage of the shared memory within one node. This can be improved by either having a better MPI_Allreduce implementation inside a node (using share memory), or by using other programming models like OpenMP for CPUs within a node. There might also be the potential to reduce the data amount by taking advantage of the fact that many of the matrices are heavily diagonally dominant. It might thus be possible to use some compression algorithm to reduce the data amount by reducing the precision of the off diagonal elements, much like the case we have reduced the precision of the wave function residual $P$. In turns of hardware, increasing the inter node bandwidth can of course also reduce this time. Thus, besides faster interconnect, possible improvement in communication should be efficient MPI library with faster collective communication or advanced MPI tuning according to the actual computer architecture, or lastly algorithm improvements based on data compression.

The third nonscaling part is the data copying of the matrix $S(M_k, M_g)$ between CPU and GPU. This is used before and after the MPI_Allreduce. The aforementioned data compression on $S$ (for MPI_Allreduce) can also reduce this data copying time. Other than that, the only hope will be on the GPU/CPU connection bandwidth. PCI-Express for Mole-8.5 theoretical limit is 4 GB/s. The latency is not an issue here, as we see the memory copy time reduces as the data amount decreases. Our measured memory copy time for one 16 MB matrix $S(M_k, M_g)$ is about 0.03 s. So the PCI-Express works at 16 MB/0.03 s = 533 MB/s. This is 13.3% of the theoretical limit. This is common for GPU clusters like Mole 8.5 which has multiple GPU cards in one node. The memory copy operations are usually quite time-consuming. This is partly because of PCI-Express contention. The test on Mole 8.5 shows that memory copy operations could reach 3 GB/s for paged memory if using a single GPU per node. Another potential reason lies in the architecture of the node. In Mole 8.5, GPU cards 0, 1, 2 are connected to one chipset, while GPU cards 3, 4 and 5 are connected to another chipset. A simultaneous memory copy operation could cause the competition for the QPI bandwidth and slow down the memory copy rate between the CPU and GPU. The same problem occurs on the GPU cluster at the Supercomputing Center of Chinese Academy of Sciences (SCCAS). If using multiple GPUs per node, the PCI-Express will be limited by the CPU front BUS inside the node for the SCCAS machine. This shows that we need to consider the interconnection between the GPU and the CPU in the architecture design of GPU nodes. Note, all these data copies between CPU and GPU are needed in order to perform the MPI_Alltoall and MPI_Allreduce on the CPU. Thus, if direct MPI access from GPU becomes available in the future, all these data copies can be removed.

The last nonscaling part is the library calls to diagonalize and decompose the matrix $S(M_k, M_g)$. The tests on different libraries have been shown in Fig. 4. On a single CPU/GPU computing unit, we see the significant improvement of using the GPU library over the CPU library. However, we are using parallel CPU libraries like ELPAA, which is still faster than the single card GPU library. It is thus quite hopeful that future development for a parallel GPU library over multiple CPU/GPU units will significantly reduce the computation times in these parts.

Thus, overall, we do believe it is possible to reduce the total computational time by another factor of $\sim 3$, through both software development and hardware improvement. The hardware improvement should come from the faster interconnect between nodes, faster CPU–GPU connections, or altogether with direct GPU access to MPI communication. Lastly, in terms of hardware, it will be beneficial to have larger GPU global memory as we move all the computation and book keeping into the GPU.

Note that, in our current machine, we have used one CPU fixed with one GPU. This makes the MPI/CUDA programming pattern very natural. Our success indicates that such a large number of GPU cards on a single CPU node might be a good deployment for scientific computing; it can thus be used as one requirement for future architecture design. Nevertheless, there are many current GPU machines which have many more GPUs than GPUs within a single node. In those cases, it makes sense to have a MPI/OpenMP/CUDA paradigm. However, there could be different ways to implement such a paradigm. A simple way is to have the number of MPI tasks equal to the number of GPUs. Then, a few CPU cores will be grouped inside one MPI task via OpenMP. Another way is to use a single MPI task for one node, while the MPI CPU core controls multiple GPUs, and other CPU cores are controlled through OpenMP. One advantage of this is to have a faster summation of the overlap matrix $S(M_k, M_g)$ within a single node. In both these approaches, it will be a challenge to find the computation to be done for the CPUs. Currently, in our scheme, we found that it will be better to move almost all the computation into the GPU, simply to avoid the data communication between the CPU and the GPU. However, if the physical problem becomes larger, as shown in our quantitative formulas in Table 2, the data communication (which is proportional to data size) will become less of a bottleneck compared to the numerical operation (which can have a higher power dependence on the data size). In those cases, it might make sense to keep the wave functions inside the CPU.

Finally, we would like to mention that further improvement might come from the overall mathematical algorithm. For example, instead of using the AB–CG, the DIIS [23] algorithm can be used. There, within the iterations, there is no need for interactions between different wave functions. Thus, all the MPI communications within those steps can be avoided, as well as the memory copies between the CPU and the GPU. However, AB–CG is more stable than the DIIS, and can be used for more cases. Nevertheless, our preliminary test indicates that the DIIS can be twice as fast as the AB–CG for the same number of iterations. The details of that ongoing study will be reported in the future.

7. Conclusions

In this work, we have further improved (by a factor of 2 in speed) our previous GPU implementation of the PWP DFT calculation introduced in Ref. [6]. The improvement comes from:

1. further moving all the calculations from the CPU to the GPU, hence increasing the numerical operation speed, and to reduce CPU/GPU memory copy;
2. a precision reduction for the wave function residual $P$, which reduces the MPI_Alltoall time, while not affecting the final converged results and the convergence rate;
3. using better CPU and GPU libraries for matrix diagonalization and decomposition.

We have carried out a detailed analysis and come up with a serial of quantitative formulas to predict the computation times for different kernels in the calculation and the total computational time. Thus we can in the future predict the computational time within about 20% of the actual tested times.

Based on the quantitative analysis, and the actual tested times, we have discussed the scaling of the calculation to the number of...
GPU/CPU units, and we have identified the remaining bottlenecks of the calculation.

We have compared our performances, especially for the MPI communication and CPU–GPU data copies to the machine communication bandwidth, discussed the difference between the actual achieved performance and the theoretical limits. We have also discussed what is needed for further improvement of the computation based on both hardware and software improvements.

More specifically, we have the following technical conclusions: (1) Our current PEtot_GPU code is x20 faster than the PEtot_CPU code; (2) This speedup is achieved by using a hybrid parallelization scheme, where the FFT and \( H \psi \) for a full wave function \( \psi \) are carried out within a single GPU, while it is \( G \)-space parallelized (distributed) when the wave function to wave function overlaps are calculated. The interchange between these two ways of parallelization is achieved via a MPI_Alltoall transpose operation; (3) In our scheme, we find that it is beneficial to move almost all the calculations into the GPU. This will not only increase the computation speed, it will also reduce the data copy between CPU and GPU; (4) As a result of (3), the best architecture for the PWP DFT calculation is a machine with as many GPUs as the number of CPU cores, and with large GPU global memory; (5) We have a quantitative analytical model which can be used to predict the computational speed of an arbitrary system for an arbitrary number of CPU/GPU processors with a 80% accuracy; (6) For our 256 CPU/GPU unit calculation of the 512 atom system, the scalable numerical operation takes about 30% of the time, the non scalable numerical operation (\( \psi \), ztoperf) takes about 15% of time, the MPI communication takes about 40% of the time, and the CPU–GPU data copy takes about 15% of the time; (7) To further speed up the calculation, it is essential to have a faster MPI_Alltoall and MPI_Allreduce. For example, to speed up our calculation by another factor of 3–5, it needs to be able to carry out an MPI_Alltoall (for 5 MB as the total amount of data inside one CPU) within 0.01 s, and finish an MPI_Allreduce for 16 MB of data also within 0.01 s. It is also important to have a faster CPU/GPU connection (e.g., each memory copy for a 16 MB data should take less than 0.005 s). All these are required in order to have a 50–100 fold increase over the current CPU calculation, and to have a MD simulation with a few seconds per MD step; (8) Finally, the GPU libraries to diagonalize and decompose a matrix need to be further improved.

Acknowledgments

The numerical calculation is carried out on the GPU cluster Mole-8.5 at the Institute of Processing Engineering, CAS. We would like to thank J. Dongarra and S. Tomov for providing MAGMA and V. Blum for providing ELPA. We would like to thank P. Kent for showing us his test results for the ELPA library. We also want to thank our team member Lijie Wang for help during data collection. Thanks also to Haifeng Lu and Han Wang for advice. The work of the first 5 authors is supported by National Basic Research Program of China 2010CB832702; NSF of China 61202054, 10972215 and 60873113; Knowledge Innovation Program of CAS CINIC_ZR_201202; 863 Program 2010AA012301 and 2010AA012402. The work of W. Gao is supported by NSF of China 11071047 and the 111 Project. The work of L.W. Wang is supported by the Director, Office of Science, Office of Basic Energy Sciences, Materials Science and Engineering Division, of the US Department of Energy (DOE) under Contract No, DE-AC02-05CH11231.

References

[5] There are many DFT PWP codes, here is a partial list: VASP, CASTEP, CPMD, ABINIT, PWSCF, PEtot, DACEPO, DFT++, PARATEC, DOD–PW, CP2K, SPHINX, and QBOX.