Accelerating DFT calculations using the Graphical Processor Unit

July 31, 2008

Matías A. Nitsche[1], Darío A. Estrín[1] and Mariano C. González Lebrero[1,2]

[1]*Departamento de Química Inorgánica, Analítica y Química Física, INQUIMAE, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Ciudad Universitaria, Pab. 2 (C1428EHA) Buenos Aires, Argentina*

[2]*Departamento de Química Biológica-IQUIFIB (UBA-CONICET), Facultad de Farmacia y Bioquímica, Universidad de Buenos Aires, Junín 956, C1113AAD, Buenos Aires, Argentina*

Abstract

In this work we show how GPGPU, or General Processing on the Graphical Processor Unit, can be useful on scientific computing. In particular, we applied this technology to electronic structure calculations performed at the density functional theory (DFT) level. We started from an existing chemistry simulation program, and selectively replaced portions of it with GPU-oriented code. In energy calculations over fixed geometries, we achieved speedups of up to approximately five times, compared to the CPU version.

1 Introduction

Modern video cards have dedicated processors (Graphical Processing Units or *GPU*s) that perform the heavy computations necessary to display 2D and 3D objects on the screen. These computations are generally specific operations that are applied to a big set of data (pixels, 3D vertexes, etc.). Because of this, GPU's are specially designed as SIMD (Single Instruction Multiple Data) processors, where parallelism is used to significantly improve performance.

Current GPUs now have the ability to execute user-written general purpose code. This is specially useful in simulation programs, where arithmetic intensity is generally high. In cases where there is high data-parallelism, significant speedups could be achieved.

Also, these video cards have the advantage of costing considerably less than a group of *CPUs* (in a HPC cluster or multicore computer) that scale as high a single GPU. The processors and memory behind these pieces of hardware have evolved with time and are now incredibly fast.

In this case, we have applied this technology to chemistry-simulation computations, as others previously did [1][2][3][4]. Specifically, we took an existing program [5] and implemented its heavier parts of some computations (in terms of computational cost) for GPU.

2 The GPU

While multiple options regarding graphical board models and programming environments exist, we have chosen a nVidia GPU (GeForce 8800 GTX, with 768MB of RAM) and the programming environment called CUDA[6], developed by the same company. Therefore, the following section will refer to this particular combination of hardware and software. In any case, similarities exist with the other available choices.

The reference CPU hardware used for comparisons is an AMD Athlon 64 X2 3600+, dual-core processor.

To generate the original program, the Intel Fortran compiler was used (with flags -ip -O3 to enhance performance, and -mp1 to improve precision). In the case of CUDA code, the -O3 optimization flag was used.

2.1 Architecture

In the case of the 8*x*-series of *nVidia*'s GPUs, the general architecture is comprised of a series of multiprocessors or *stream-processors*, according to the manufacturer, that share access to a *global* memory segment (this is actually the graphical board's device memory). Each multiprocessor also has its own *shared* memory, which can be used to intercommunicate the actual processors contained in it. The *shared* memory can be as fast as accessing a processor register, while a *global* memory access involves hundreds of cycles. Memory accesses in general are not cached, unless specific memory segments (the *constant* and the so-called *texture* segments) are used.

An important issue with this model (which was present in all models to the time of writing) is that it supports single-precision floating-point operations only. In the particular application we used, precision is a crucial issue.

2.2 Programming Model

While there were originally low-level approaches to programming on GPUs, currently we have alternative options. As mentioned previously, we used CUDA (Compute Unified Device Architecture). This programming environment consists of a *C*-based language with some extensions that allows us to define specific *methods* or *functions* as GPU code (which are called *kernels* in general). The CUDA compiler will then generate GPU assembly out of these portions, together with interface code between regular *C* and CUDA.

The programming model is similar to a multithreaded application (it generally consists of a pipe-and-filter layout). A group of threads are spawned and the GPU method or *kernel* (written by the CUDA user) is launched. Each thread will execute the *kernel* code on its own. Generally, the *kernel* is written in such way that threads cooperate by exchanging intermediate results (using each of the multiprocessor's shared memory).

This group of threads needs to be defined as a bi-dimensional grid, subdivided in bi-dimensional *blocks* (see figure 1). Each *block* of threads can intercommunicate through a multiprocessor's shared memory, as *blocks* are guaranteed to run on a single multiprocessor (not all threads in it will be run concurrently, but in batches called *warps*). To synchronize access to this shared medium, a *barrier* method is used. This holds all threads until they reach the *barrier*, after which thread execution continues independent of each other again. This type of synchronization is only possible between threads of the same *block*, and not between threads of different *blocks* (what we could call, *global synchronization*).

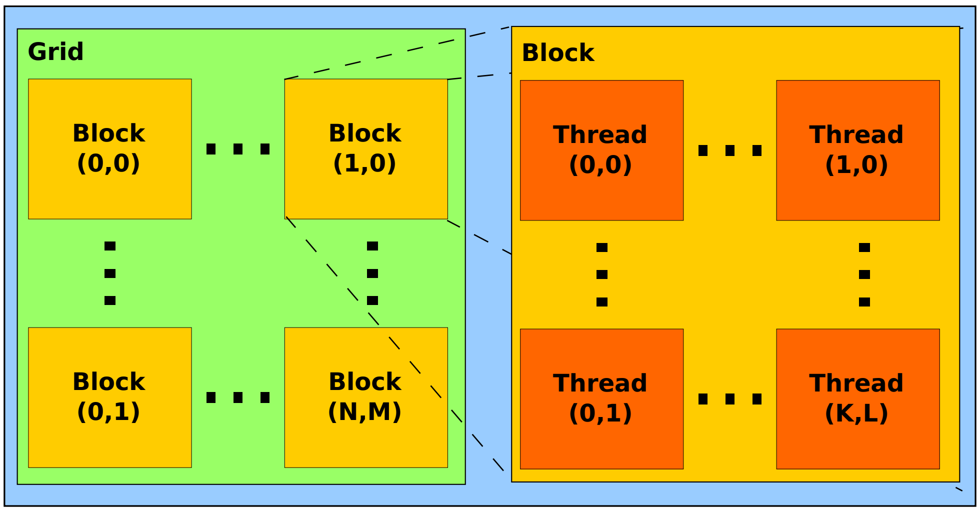

Figure 1. Grouping of threads in a *grid* of *blocks*

3 A Concrete Application

The code developed consists of a reimplementation of specific parts of a chemistry-simulation program, based on the quantum mechanics approach called Density Functional Theory (DFT)[7]. The goal of this calculation is to compute the electronic structure and energetics of a system consisting of a set of nuclei and a group of electrons. The energy is calculated as:

$$E[\rho] = T_s[\rho] + V_{\text{ne}}[\rho] + \frac{1}{2}\int\int \frac{\rho(\vec{r}_1)\rho(\vec{r}_2)}{R_{12}}\, d\vec{r}_1\, d\vec{r}_2 + E_{\text{xc}}[\rho] \tag{1}$$

where T_s is the kinetic energy, V_{ne} is the electronic-nuclei interaction, the third term represents the classical electron-electron interaction ($\rho(\vec{r}\,)$ corresponds to the electronic density at point $\vec{r}\,$) and the fourth is the so-called exchange-correlation energy. The major computational cost resides in this last term, and it's computed as follows:

$$E_{\text{xc}} = \oint \rho(\vec{r}\,)\,\varepsilon_{\text{xc}}(\rho(\vec{r}\,))\, d\vec{r}\,, \vec{r} \in \mathbb{R}^3 \tag{2}$$

where ε_{xc} is the local exchange-correlation energy functional of the density. This is the simplest method of approximation because it uses exclusively the local density. Because of this, this method is referred to as Local Density Approach or LDA[8].

The previous integral is implemented as an approximation over a grid[9] $G \subseteq \mathbb{R}^3$ of points $\vec{p}$. This grid is actually a sum of various others that result from centering and scaling a grid $g \subseteq \mathbb{R}^3$, over each atom of the system. This grid g consists of a set of points distributed in concentric shells around the origin. The approximation is then:

$$E_{\text{xc}} \cong \sum_{\vec{p} \in G} \rho(\vec{p}\,)\,\varepsilon_{\text{xc}}(\rho(\vec{p}\,)) \tag{3}$$

where

$$\rho(\vec{p}\,) = \sum_i |\psi_i\,(\vec{p}\,)|^2, \forall \vec{p} \in G \tag{4}$$

and the functions ψ_i (referred to as orbitals) are defined as:

$$\psi_i(x,y,z) = \sum_{k=1}^{n} c_i^k \chi_k(x,y,z) \tag{5}$$

over the basis functions χ_k:

$$\chi_k(x,y,z) = (x-x_0)^{n_x^k}(y-y_0)^{n_y^k}(z-z_0)^{n_z^k} \sum_j d_j^k e^{-\alpha_j \vec{s}} \tag{6}$$

$$\vec{s} = (x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2 \tag{7}$$

where c_i^k and d_j^k are constants.

3.1 The original implementation (serial version)

The approximation of the energy is achieved by using an iterative algorithm, that seeks to minimize the total energy (to which the exchange-correlation energy contributes) of the input system. At each step, for each point of the grid G, ρ is first calculated. Then, this value is used to update the *Fock-Matrix*, which is necessary for the rest of the simulation. This matrix depends on the basis functions and the density. At the final step (once convergence is assumed to be reached), the energy is computed, this time using a denser grid g (and therefore, a denser G). The existing program implements this with several nested loops, as seen in code listing of figure 2. This method iteration_step, is responsible for computing the *Fock-Matrix* during the iterative approach, and the energy at the last step.

```
def iteration_step(input/output: fock_matrix, input/output: e, last_step)
  for each i in atoms do
    for each j in shells(i) do
      for each k in grid $g$ do
        p = compute_grid_point(i, j, k)   // $p \in G$
        Fs = compute_functions(p, fock_matrix)
        d = compute_density(p, Fs, fock_matrix)

        if (last_step) then e += energy(Fs, d, fock_matrix)
        else update_fock_matrix(Fs, d, fock_matrix) end
      end
    end
  end
end
```

Figure 2. Pseudo-code for the algorithm to be replaced for GPU

```
def update_fock_matrix(Fs, d, fock_matrix)
  for each Fi in Fs do
    for each Fj in Fs do
      fock_matrix[i][j] += compute_fock_matrix_element(Fi, Fj, d)
    end
  end
end
```

Figure 3. Pseudo-code for the update_fock_matrix method

The temporal complexity of this algorithm is $O(n^2m)$, where n is the number of basis-functions (the size of the `Fs` array) and m the number of grid-points in G. The dimension of the *Fock-Matrix* is n^2. The grid g used during convergence has 116 points, and 194 at the last step. The number of layers is generally 35/40 (depending on the atomic number of each atom).

The code that will be reimplemented on the GPU corresponds to the `iteration_step` method (although it will be implemented by two separate *kernels*, as described in the next section). The original programs spends the majority of the total run-time executing this portion of code (figure 4).

operation	% of total run-time
Exchange-Correlation	97.3
Coulomb Integral	0.8
Other	1.9

Figure 4. Percentage of time spent by the execution of the most significant portions of the computation (measured during a 12 H_2O water cluster LDA computation)

3.2 Reimplementation on the GPU

In a naïve approach, we could just write a kernel that implements the actual code inside the innermost `for` of the `iteration_step` method, and launch this over a *grid* of threads defined for every (i, j, k). That is, each thread would be computing a single iteration of the three loops. Afterwards, all threads would need to add their results together. This idea poses various problems.

The first problem is a consequence of the lack of global synchronization on this architecture. In other words, results from arbitrary threads can't be added together to get a global final result. The only possibility is to get partial intermediate results for every block of threads, since these can intercommunicate and add their results together. Afterwards, this group of intermediate results can be again reduced in the same manner, until a final value is computed.

Another issue involved in a GPU accumulation-scheme is that all of the operations (in this case, addition) would be performed on single-precision. As explained before, in this type of application, precision is something that needs to be taken into account.

The third problem which arises is the need of each thread to have each own memory space to store its partial results (for example, we would need n^2 `floats` for every *Fock-*

Matrix of every thread). In other words, we would be translating the temporal complexity into spatial complexity.

Because of these reasons, we decided to: (a) simply store the intermediate results of each thread, and then add them on the CPU; (b) separate the code into two distinct kernels. With (a) we now have double-precision addition and we also avoid the need of global synchronization or an iterative reduction algorithm as described before. Decision (b), as we will see, avoids requiring massive amounts of memory.

```
def energy_kernel(input/output: e, input/output: Fs, last_step)
  i,k = threadId.x,threadIdx.y
  for each j in shells(i) do
    p = compute_grid_point(i,j,k)
    Fs[i][j][k] = compute_functions(p, fock_matrix)
    d[i][j][k] = compute_density(p, Fs[i][j][k], fock_matrix)
    if (last_step) then e[i,j,k] = energy(Fs[i][j][k], d, fock_matrix) end
  end
end
```

Figure 5. First *kernel*, responsible of computing and storing the basis-functions and energy, for each grid point

```
def fock_matrix_kernel(fock_matrix, Fs, d)
  fi,fj = threadId.x, threadId.y
  shared_Fi[blockSize], shared_Fj[blockSize]
  fock_matrix[fi][fj] = 0
  for each i in atoms do
    for each j in shells(i) do
      for each k in grid do
        p = compute_grid_point(i,j,k)
        syncthreads()
        if (threadIdx.y == 0) then shared_Fi[fi] = Fs[i][j][k][fi] end
        if (threadIdx.x == 0) then shared_Fj[fj] = Fs[i][j][k][fj] end
        syncthreads()
        fock_matrix[fi][fj] += compute_fock_matrix_element(shared_Fi[fi],
                                                          shared_Fj[fj],
                                                          d[i][j][k])
      end
    end
  end
end
```

Figure 6. *Fock-Matrix* updating *kernel*

So, the first *kernel* (figure 5) computes the basis functions and density (or energy) for each valid (i, j, k) combination. The second kernel (figure 6) is in charge of computing the *Fock-Matrix*. The main difference compared to the serial version, is that the loops are *inverted*, to avoid the memory issue mentioned before. This inversion consists of writing this second kernel so that each thread computes an element of the *Fock-Matrix*, by going trough every (i, j, p) and reads the necessary parameters computed by the first kernel. This approach eliminates the need of requiring one intermediate *Fock-Matrix* per thread (to be later accumulated).

Because of decision (a), we now have a tri-dimensional array (or matrix) corresponding to `e` (the energy computed at each point of the grid). This matrix's elements will be accumulated to a single final value on the CPU. This way of storing per-thread values is also needed to communicate partial values between kernels (`Fs` and `d` arrays).

In fock_matrix_kernel it can be seen how shared memory was used to load the function values that two threads would be reading in common. To compute $\text{Fock}_{i,j}$, function values F_i and F_j are required. This means that to compute $\text{Fock}_{i,k}$ and $\text{Fock}_{l,j}$, functions F_i and F_j will be required, respectively, for every k, l. Therefore, the shared arrays `shared_Fi` and `shared_Fj` are first loaded with all function values that are to be used by all the threads in this block.

The grid of threads for the first kernel was defined in blocks of 128 threads, and the second one, in blocks of 256 threads.

Determining how to parallelize (ie: how to define the *grid* and *blocks*) is not trivial, since it involves thinking kernel code from a different perspective. Also, to parallelize more implies less work per-kernel and more memory used for intermediate results between successive kernel calls (since threads can't communicate globally). The per-kernel work was a factor considered when defining the first kernel, since completely parallelizing all loops was slower than just doing two of them. Therefore, the kernel actually includes one of the for loops in its code, while the other two are replaced by the bi-dimensional grid of threads that execute it.

Also, since the original program was programmed completely using double-precision due to the importance of the quality of intermediate results, the use of single-precision was a factor to take into consideration. The main problem is that these results span a wide range of values, which introduces accumulation errors for example. In any case, CUDA is almost in complete compliance with the IEEE754 floating point standard. The biggest difference with the floating point processor (or FPU) of the CPU, is that the GPU uses strictly 32bit data-types for single-precision operations instead of having intermediate registers with higher precision.

4 Results

We've tested the implementation by comparing both the GPU and CPU versions of the program, applying the computation to different input systems. We present the total run-time of the simulation and the differences in the final energy value computed between both program versions (absolute and relative error) in tables 7 and 8. On table 10, we show the time spent by the GPU and CPU on the portion of the code in question, next to the factor of acceleration.

The absolute ($\triangle E$) and relative error ($|\triangle E/E_{\text{cpu}}|$) of the final energy values (equation 1), obtained by comparing the values of the CPU and GPU versions of the code, are a measure of the quality of the results in terms of simulation accuracy. The total run-time is measured from simulation start to end.

Input System	$\triangle E$ [kcal/mol]	$\|\triangle E/E_{\text{cpu}}\|$	T_{gpu} [s]	T_{cpu} [s]	$\frac{T_{\text{cpu}}}{T_{\text{gpu}}}$
CH_4	0.0010	4.03×10^{-8}	3.45	6.43	1.81
C_2H_8	0.0017	3.52×10^{-8}	6.32	21.48	3.40
C_3H_8	0.0027	3.66×10^{-8}	15.74	52.86	3.29
C_4H_{10}	0.0035	3.65×10^{-8}	24.21	106.16	4.38
C_5H_{12}	0.0055	4.56×10^{-8}	47.55	182.55	3.84
C_7H_{16}	0.0124	7.33×10^{-8}	72.90	289.12	3.97
C_9H_{20}	0.0101	4.65×10^{-8}	172.99	745.83	4.31
$C_{13}H_{28}$	0.0191	6.07×10^{-8}	1206.15	5840.91	4.84

Figure 7. Results for alcane series (total simulation time and quality of results)

Input System	$\triangle E$ [kcal/mol]	$\|\triangle E/E_{\text{cpu}}\|$	T_{gpu} [s]	T_{cpu} [s]	$\frac{T_{\text{cpu}}}{T_{\text{gpu}}}$
H_2O	0.0028	5.89×10^{-8}	2.35	2.21	0.94
$(H_2O)_2$	0.0024	2.56×10^{-8}	4.52	9.21	2.04
$(H_2O)_3$	0.0048	3.38×10^{-8}	8.3	25.23	3.04
$(H_2O)_4$	0.0041	2.18×10^{-8}	17.77	54.53	3.07
$(H_2O)_5$	0.0019	8.18×10^{-9}	26.2	98.42	3.76
$(H_2O)_6$	0.0039	1.38×10^{-8}	39.5	162.84	4.12
$(H_2O)_7$	0.0034	1.03×10^{-8}	69.04	252.46	3.66
$(H_2O)_8$	0.0042	1.12×10^{-8}	90.33	371.40	4.11
$(H_2O)_9$	0.0032	7.45×10^{-9}	123.94	529.25	4.27
$(H_2O)_{10}$	0.0053	1.12×10^{-8}	181.08	735.70	4.06
$(H_2O)_{11}$	0.0119	2.30×10^{-8}	230.45	957.27	4.15
$(H_2O)_{12}$	0.0023	4.15×10^{-9}	298.32	1352.62	4.53

Figure 8. Results of water cluster series (total simulation time and quality of results)

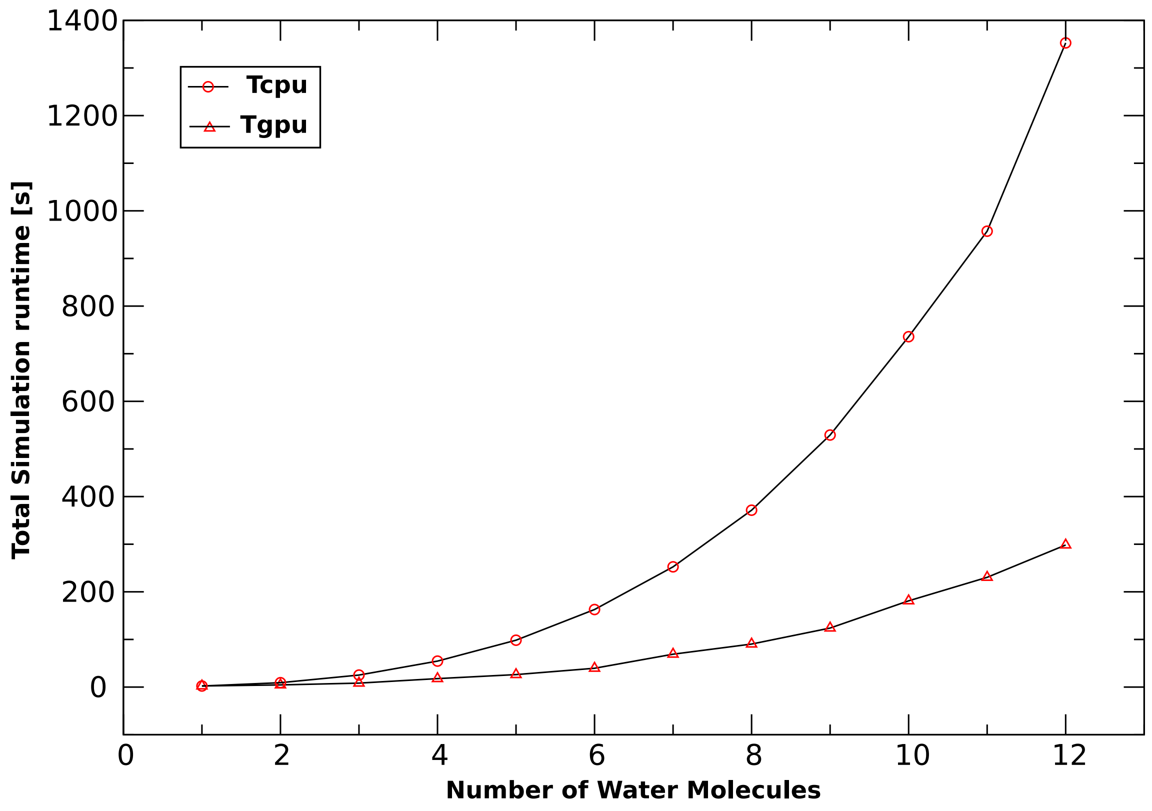

Figure 9. Results of water cluster series (total runtime of simulation)

	T_{gpu} [ms]	T_{cpu} [ms]	$\frac{T_{\text{cpu}}}{T_{\text{gpu}}}$
H_2O	33.71	40.39	1.2
$(H_2O)_2$	76.1	173.66	2.28
$(H_2O)_3$	145.95	499.45	3.42
$(H_2O)_4$	327.2	1061.32	3.24
$(H_2O)_5$	484.6	1958.75	4.04
$(H_2O)_6$	726.9	3236.98	4.45
$(H_2O)_7$	1325.18	5148.9	3.89
$(H_2O)_8$	1730.16	7594.89	4.69
$(H_2O)_9$	2335.16	10824.81	4.64
$(H_2O)_{10}$	3388.5	16097.94	4.75
$(H_2O)_{11}$	4448.49	20492.17	4.61
$(H_2O)_{12}$	5589.94	26696.37	4.78

Figure 10. Time spent on the exchange-correlation subroutine

We've also run *geometry optimizations*, which seek to find a better geometry of the atoms in space, by minimizing energy at each geometry. Results are seen in figure 11. RMSD corresponds to the Root Mean Square Deviation computed between the last geometries returned by GPU and CPU (the difference of the final geometries determined, measured in Angstrom). While this value is proportional to the input system size, it serves to show the quality of the results thrown by the GPU version of the code.

	RMSD $[\dot{A}]$	$T_{\text{gpu}}\,[s]$	$T_{\text{cpu}}\,[s]$	$\frac{T_{\text{cpu}}}{T_{\text{gpu}}}$
C_5H_{12}	3.99×10^{-5}	190.05	712.52	3.75
C_9H_{20}	1.33×10^{-2}	466.59	1632.95	3.49
$C_{13}H_{28}$	6.48×10^{-3}	1714.97	5331.87	3.10
$(H_2O)_9$	2.3×10^{-4}	889.22	3577.13	4.02
$(H_2O)_{12}$	1.8×10^{-4}	1405.82	5526.45	3.93
Fe(CN)$_5N_2^{-3}$	2.1×10^{-3}	1179.17	3459.00	2.93
Fe(CN)$_5N_2C_4H_9^{-2}$	1.25×10^{-2}	3612.94	14633.18	4.05

Figure 11. Geometry optimization results

5 Conclusion

We can see that in the case of the code implemented on the GPU, the total runtime of the simulation achieved speedups of up to approximately five times, compared to the CPU version. While this speedup is considerable (specially taking into account the long runtime of these simulations), the code could possibly be optimized. Memory usage is quite high, which limits the size of the input system.

The quality of the results is more than acceptable, since what matters is the absolute error, in terms of simulation accuracy. In all cases, this value is less that 0.02 kcal/mol. This ensures that values computed by the simulation can actually be used as a replacement of the results returned by the CPU version. It is possible that significant improvements could only be reached by using double-precision, already present in the next-generation GPUs.

In terms of concrete applications, an important conclusion of this work, is that the quality of the results is a motivating factor to port further parts of the simulation program and perform molecular dynamics methods describing at least a part of the system at the quantum level. In general, we have showed how useful GPUs can be in computer simulation, since these systems tend to have the necessary properties, like high data-parallelism, to be successfully parallelized in hardware.

6 Acknowledgment

This work was partially supported by the University of Buenos Aires, Agencia Nacional de Promoción Científica y Tecnológica (project PICT 25667), CONICET. We thank A. Roitberg for useful discussions and A. Colubri for a great idea.

Bibliography

[1] Amos G. Anderson, William A. Goddard III, and Peter Schröder. Quantum monte carlo on graphical processing units. *Computer Physics Communications*, 177(3):298–306, 2007.

[2] I.S. Ufimtsev and T.J. Martinez. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation*, 4(2):222–231, 2008.

[3] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, and A. Aspuru-Guzik. Accelerating resolution-of-the-identity second-order möller-plesset quantum chemistry calculations with graphical processing units. *Journal of Physical Chemistry A*, 112(10):2049–2057, 2008.

[4] Koji Yasuda. Two-electron integral evaluation on the graphics processor unit. *Journal of Computational Chemistry*, 29(3):334–342, 2008.

[5] Estrin D. A., Corongiu G., and Clementi E. *Methods and Techniques in Computational Chemistry*. METECC, 1993.

[6] NVIDIA. *CUDA Programming Guide 1.1*, 2007.

[7] Ira N. Levine. *Quantum Chemistry*, chapter 15.20. 5th edition, 2001.

[8] S. H. Vosko, L. Wilk, and M. Nusair. *Can. J. Phys.*, 58:1200, 1980.

[9] A. D. Becke. *Chem. Phys.*, 88:1053, 1988.