Quantum Monte Carlo on Graphical Processing Units

Amos Anderson William A. Goddard III

Materials and Process Simulation Center, Division of Chemistry and Chemical Engineering, California Institute of Technology (MC 139-74), Pasadena, California 91125

Peter Schröder

Department of Computer Science, California Institute of Technology, Pasadena, California 91125

Abstract

Quantum Monte Carlo (QMC) is among the most accurate methods for solving the time independent Schrödinger equation. Unfortunately, the method is very expensive and requires a vast array of computing resources in order to obtain results of a reasonable convergence level. On the other hand, the method is not only easily parallelizable across CPU clusters, but as we report here, it also has a high degree of *data parallelism.* This facilitates the use of recent technological advances in Graphical Processing Units (GPUs), a powerful type of processor well known to computer gamers. In this paper we report on an end-to-end QMC application with core elements of the algorithm running on a GPU. With individual kernels achieving as much as 30x speed up, the overall application performs at up to 6x relative to an optimized CPU implementation, yet requires only a modest increase in hardware cost. This demonstrates the speedup improvements possible for QMC in running on advanced hardware, thus exploring a path toward providing QMC level accuracy as a more standard tool. The major current challenge in running codes of this type on the GPU arises from the lack of fully compliant IEEE floating point implementations. To achieve better accuracy we propose the use of the Kahan summation formula in matrix multiplications. While this drops overall performance, we demonstrate that the proposed new algorithm can match CPU single precision.

Key words: Graphical Processing Units, Quantum Monte Carlo, matrix multiplication, floating point error, Kahan Summation Formula, de-normals
PACS: 07.05.Bx, 02.70.Ss, 02.60.Dc, 89.20.Ff

Email addresses: `amosa@caltech.edu` (Amos Anderson),
`wag@wag.caltech.edu` (William A. Goddard III),
`ps@cs.caltech.edu` (Peter Schröder).

1. Introduction

The rapid increase in GPU floating point performance and their excellent flops/$ characteristics, suggests that they may provide cost effective solutions for scientific computation problems. Given that the GPU computing model is (1) quite different from standard CPU models, (2) lacks a fully compliant IEEE floating point implementation, and (3) is optimized for very specific graphics type computational kernels, it is not clear *a priori* which scientific computing tasks are cost effective on GPUs.

A number of scientific computing algorithms have been pursued on the GPU, *e.g.*, fluid simulations[1,2], elasticity[3], and general finite element methods[4]. At the level of computational mathematics kernels, we have seen work on LU decomposition[5], matrix/vector products[6–14], iterative solvers[10,15], and transforms such as Fourier and Wavelet[7,16–18]. In some cases the results can be disappointing relative to highly tuned CPU implementations, in particular when high precision answers are required, or when problem sizes do not hit a particular sweet spot (*i.e.*, large matrices, or power-of-2 sized data structures, *etc.*). With continuing hardware development these performance barriers are being ameliorated, and with the recent announcement by nVidia of double precision availability on the GPU in 2007, computational precision is a fading problem as well.

In this paper we consider quantum chemistry computations, the heart of which is the computation of the electronic structure of a given molecule using the quantum mechanical equations of motion. This information is critical for, among other tasks, finding optimized geometric structures for the molecule, reaction pathways, obtaining vibrational information, and providing a basis for developing higher level approximation methods including molecular dynamics simulations. Accurate results have application in catalysis, nanotechnology, drug design, and fuel cells, among many others.

Due to the large state space ($3N$ for N electrons) and the non linear nature of the time independent Schrödinger equation, exact results are all but impossible. Consequently a variety of approximation algorithms have been developed. One such approach, Quantum Monte Carlo (QMC)[19], is based on the stochastic evaluation of the underlying integrals and is guaranteed to produce accurate answers in the limit of infinite state space sampling. Even though a very large number of samples are typically required, QMC is easily parallelizable and scales as $O(N^3)$ (albeit with a very large constant). This motivates a search for computational augmentation.

We report on our implementation of QMC on the nVidia 7800 GTX and compare it against a 3.0 gHz Intel P4, considered to be representative of similar levels of development. These technologies are improving very fast, both for CPUs and for GPUs. Currently however, the time to doubled performance on GPUs is noticably shorter than for CPUs, leading to increasing performance advantages for GPUs *if a computation maps well enough onto the GPU.* Since CPUs are beginning to follow the same multicore technology trend, the notion that precision issues are temporal is reenforced.

In the present paper, scientific results as well as underlying formalisms were simplified for purposes of presentation and to focus on the essential computational aspects. We admit that it is unclear how single precision results might be useful, especially for an algorithm designed to produce highly accurate results. In the mean time, our single precision implementation is presented. Aside from the performance of individual kernels we consider (1) precision issues arising from the noticeable differences to single precision IEEE floating point arithmetic, (2) performance issues arising from the specific sizes of matrices we must use, and (3) the overall performance of an end to end application when compared against a heavily tuned CPU based version.

2. Intro to Graphical Processing Units

GPUs have received much interest outside the graphics world recently due to their immense processing power even though they are actually devices designed for very specialized tasks. Many reviews of GPU adaptability and compatibility are already available[1,20,21], and we do not attempt to improve upon them. In addition, there has been the development of specialized programming environments[6,22,23] for GPUs specifically designed to smooth the porting of non-graphics applications, and GPU vendors themselves have recently released general purpose GPU programming environments.

Our approach was to start from the ground up in hopes of squeezing the best performance we can from the device. To describe our techniques a truncated description of the technology is required. The motivating principle for GPU design is that simple calculations do not need general processors, so the addition of an auxiliary processor could both speed up graphics related calculations as well as free the CPU to complete other tasks. Since graphical calculations most typically involve drawing 2D images of colors ultimately intended for a screen, GPUs start with pixels (more generally referred to as fragments or texels) as the atomistic unit of data. Fragments are manifested here as 4 single precision floats, aliased as `xyzw` channels. A 2D array of fragments is called a *texture*, and is the fundamental storage class. A GPU will stream a region of a texture through an array of simple *fragment processors* (our nVidia 7800 GTX has 24), where each of these will produce one fragment as output. A programmer can utilize this process by designating a kernel for the fragment processors to use, resulting in the evaluation of data for a specified region in a texture. This entire procedure is commonly referred to as a *pass.* A kernel is a small program which in the graphics context would typically perform some shading calculation. There is nothing in principle preventing the user from writing a "shader" which performs some scientifically relevant computation using the broad class of functions available at the programmable shader level.

In practice, many considerations are necessary in order to maximize efficiency. Graphics processing can be thought of as a sophisticated queuing system were a CPU sends a list of tasks to one (or more) connected GPUs and collects the results when the calculations are complete. This means that there are also processor communication factors that need to be included. As far as the GPU itself is concerned, we mention here the considerations

- padding empty slots in texture data with 0 whenever data dimensions do not match dimensions on the GPU,
- running as many passes with a kernel before swapping it for another since the GPU can only have one kernel loaded at a time,
- careful data arrangement,
- a tuning of how much of the computation as a whole should be assigned to each kernel
- and in general, keeping the GPU busy at all times.

Before discussing how these concerns play out in our setting we give a brief high level introduction to Quantum Monte Carlo computations to understand the needed computational components which we seek to map to the GPU.

3. Intro to Quantum Monte Carlo

The most important information about a molecule is its ground state energy, calculated by means of the time independent Schrödinger equation

$$\langle E\rangle = \frac{\int \Psi(\bar{\boldsymbol{r}})\hat{H}\Psi(\bar{\boldsymbol{r}})d\bar{\boldsymbol{r}}}{\int \Psi^2(\bar{\boldsymbol{r}})d\bar{\boldsymbol{r}}} \tag{1}$$

where $\Psi(\bar{\boldsymbol{r}}) : \mathbb{R}^{3N} \to \mathbb{R}$ is the wave function, mapping the $3N$ Cartesian coordinates of N electrons into a probability amplitude related to the probability density in Eq. (4). (Equation (1) includes the common restriction that $\Psi(\bar{\boldsymbol{r}})$ is a real valued function.) The Hamiltonian operator $\hat{H}$ is given by

$$\hat{H} = -\frac{1}{2}\nabla^2 + V(\bar{\boldsymbol{r}}) \tag{2}$$

where the Laplacian is over all $3N$ electronic coordinates and calculates the kinetic energy (in the unitless Hartree measure) of the electrons in the molecule. The $V(\bar{\boldsymbol{r}})$ term represents the potential energy due to Coulomb interactions between all pairs of electrons and nuclei. The energy E is the eigen value of $\hat{H}$ operating on the eigen function $\Psi(\bar{\boldsymbol{r}})$. The ground state energy is the lowest such eigen value, and is of primary interest here.

There are many methods to calculate Eq. (1) with varying degrees of accuracy and computational complexity. The highly accurate QMC family of algorithms[24] uses Metropolis[25] integration to fine tune the result provided by a cheaper method. It uses the *local energy*

$$E_L(\bar{\boldsymbol{r}}) = \frac{\hat{H}\Psi(\bar{\boldsymbol{r}})}{\Psi(\bar{\boldsymbol{r}})} = -\frac{1}{2}\frac{\nabla^2\Psi(\bar{\boldsymbol{r}})}{\Psi(\bar{\boldsymbol{r}})} + V(\bar{\boldsymbol{r}}) \tag{3}$$

which represents an evaluation of the energy for a set of electronic coordinates. In terms of the stationary probability distribution of electrons

$$\rho(\bar{\boldsymbol{r}}) = \frac{\Psi^2(\bar{\boldsymbol{r}})}{\int \Psi^2(\bar{\boldsymbol{r}})d\bar{\boldsymbol{r}}} \tag{4}$$

we can transform Eq. (1) into the Monte Carlo integration form

$$\langle E\rangle = \int \rho(\bar{\boldsymbol{r}})E_L(\bar{\boldsymbol{r}})d\bar{\boldsymbol{r}} = \lim_{N_t\to\infty}\frac{1}{N_t}\sum_{t=1}^{N_t} E_L(\bar{\boldsymbol{r}}_t). \tag{5}$$

Here $\bar{\boldsymbol{r}}_t$ are a series of electronic coordinates generated with respect to $\rho(\bar{\boldsymbol{r}})$ by some importance sampling scheme[26]. Since error scales as $1/\sqrt{N_t}$ in Monte Carlo methods a rather large number of samples is required to achieve useful accuracies. Additionally, it is common to run several independent series, called *walkers*, in order to minimize the error due to serial correlation between the N_t data points.

In terms of computational complexity, the difficulty for QMC lies in the evaluation of $\nabla^2\Psi(\bar{\boldsymbol{r}}_t)$ for each $E_L(\bar{\boldsymbol{r}}_t)$ as well as the evaluation of $\Psi(\bar{\boldsymbol{r}}_t)$ and $\nabla\Psi(\bar{\boldsymbol{r}}_t)$ which are used for importance sampling. The most common functional form for $\Psi(\bar{\boldsymbol{r}})$ has at least three nested stages of evaluation. At the first stage, we place a collection of N_{bf} basis functions centered at the nuclei in the $3D$ coordinate space. Typically a given nucleus is associated with multiple basis functions. The basis function takes as argument the local coordinates of a given electron (i) relative to the nucleus (j), $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{R}_j$. The best results are achieved with the following functional form

$$\chi_j(x_{ij}, y_{ij}, z_{ij}) = x_{ij}^{k_j} y_{ij}^{l_j} z_{ij}^{m_j} \sum_{n_j} a_{n_j} e^{-b_{n_j} r_{ij}^2}. \quad (6)$$

For each basisfunction, $R_j, k_j, l_j, m_j, n_j, a_{n_j}$ and b_{n_j} are parameters given as input to the QMC program. The $k_j, l_j, m_j \in \mathbb{N}$ parameters give the basisfunction the required symmetry, and $n_j \in \mathbb{N}^+$ helps select the quality of fit. The other parameters are all real numbers.

The second stage of evaluation takes linear combinations of basisfunctions to create *molecular orbitals*. The k^{th} orbital is given by $\phi_k(\boldsymbol{r}_i) = \sum_j \chi_j(r_{ij})c_{jk}$, where $c_{jk} \in \mathbb{R}$ are coefficients input to QMC. These orbitals represent the spread of the electron across the entire molecule.

Finally, the third stage of evaluation relevant to this study is the *Slater determinant*, chosen for its antisymmetric properties. For the N_s electrons of a given quantum spin ($N = N_\alpha + N_\beta \sim 2N_\alpha$) the determinant is a function of the ϕ_k (which in turn are functions of the $\chi_j(r_{ij})$)

$$D_s(\bar{\boldsymbol{r}}_s) = |M_s(\bar{\boldsymbol{r}}_s)| = \begin{vmatrix} \phi_1(\boldsymbol{r}_1) & \phi_2(\boldsymbol{r}_1) & \cdots & \phi_{N_s}(\boldsymbol{r}_1) \\ \phi_1(\boldsymbol{r}_2) & \phi_2(\boldsymbol{r}_2) & & \\ \vdots & & \ddots & \\ \phi_1(\boldsymbol{r}_{N_s}) & & & \phi_{N_s}(\boldsymbol{r}_{N_s}) \end{vmatrix} \quad (7)$$

(here we partition $\bar{\boldsymbol{r}}$ into $\bar{\boldsymbol{r}}_\alpha$ and $\bar{\boldsymbol{r}}_\beta$) and the wavefunction is

$$\Psi(\bar{\boldsymbol{r}}) = D_\alpha(\bar{\boldsymbol{r}}_\alpha)D_\beta(\bar{\boldsymbol{r}}_\beta).$$

To calculate the kinetic energy, we first obtain $\nabla_i^2\phi_k(\boldsymbol{r}_i) = \sum_j \nabla_i^2\chi_j(\boldsymbol{r}_{ij})c_{jk}$, and then sum the contributions from all the electrons in all the orbitals

$$\frac{\nabla^2\Psi(\bar{\boldsymbol{r}})}{\Psi(\bar{\boldsymbol{r}})} = \sum_{s\in\{\alpha,\beta\}} \sum_{i,k\in N_s} \left[M_s^{-1}(\bar{\boldsymbol{r}}_s)\right]_{ki} \nabla_i^2\phi_k(\boldsymbol{r}_i). \quad (8)$$

A similar procedure is followed for calculating the gradient of the wavefunction for each electron with the exception that the final summation results in a vector of gradients.

To summarize the algorithm, we are given a set of nuclear coordinates, basis function parameters, and the c_{jk}, which describe the wavefunction as fit by some other (more approximate and cheaper) method. Additionally, we choose some parameters including the number of steps N_t, the number of walkers W, an initial guess scheme for positions $\bar{\boldsymbol{r}}$ of all the electrons, as well as several parameters relating to the importance sampling. Although specific choices are often related to the computational resources available and to the importance sampling method used, W is usually $O(10)$ to $O(10^3)$, N_t is $O(10^4)$ to $O(10^8)$, and the dimensions of c_{jk} are usually between $O(10)$ and $O(10^3)$, depending upon the molecule. With these in hand, the algorithm can be stated as shown in Algorithm 1 (the $\otimes$ represents matrix multiplication), where simplifications have been included based on assumptions about the importance sampling.

Algorithm 1 The QMC algorithm

```
$E_{sum} \leftarrow 0$
**for** $w = 1$ to $W$ **do**
  $\boldsymbol{r}_{ij} \leftarrow$ initialize()
  **for** $t = 1$ to $N_t$ **do**
    **for** $s = \alpha$ and $s = \beta$ **do**
      $M_s \leftarrow \chi_j(\boldsymbol{r}_{ij}) \otimes c_{jk}$
      $X_s \leftarrow \frac{\partial}{\partial x_i}\chi_j(\boldsymbol{r}_{ij}) \otimes c_{jk}$
      $Y_s \leftarrow \frac{\partial}{\partial y_i}\chi_j(\boldsymbol{r}_{ij}) \otimes c_{jk}$
      $Z_s \leftarrow \frac{\partial}{\partial z_i}\chi_j(\boldsymbol{r}_{ij}) \otimes c_{jk}$
      $L_s \leftarrow \nabla_i^2\chi_j(\boldsymbol{r}_{ij}) \otimes c_{jk}$
    **end for**
    Jastrow $\leftarrow J(\bar{\boldsymbol{r}})$
    $\Psi \leftarrow \det M_\alpha * \det M_\beta *$ Jastrow
    $E_{sum} \leftarrow E_{sum} +$
      $E_L(M_s, \text{Jastrow}, \{\textit{derivatives}\}\ldots)$
    $\boldsymbol{r}_{ij} \leftarrow$ sampling($\Psi, \boldsymbol{r}_{ij}, X_s, Y_s, Z_s, L_s$)
  **end for**
**end for**
$E_{avg} \leftarrow E_{sum}/(N_t * W)$
```

The high degree of parallelism is evident since each processor can calculate all the linear algebra for its walkers and only needs to produce a single value; the energy. *

One big advantage of QMC relative to alternative methods is the freedom one has in choosing the functional form of $\Psi(\bar{\boldsymbol{r}})$. This is exploited by multiplying the Slater determinant wave function with a set of pairwise interaction terms which explicitly model electron correlation by employing inter-electronic coordinates. The only condition is that these terms, called *Jastrow* functions, preserve the antisymmetry of the wave function. To satisfy this condition, we use the functional form

$$J(\bar{\boldsymbol{r}}) = \prod_{q<p} e^{u_{pq}(r_{pq})} \tag{9}$$

which provides a term for each particle-particle interaction, where

$$u_{pq}(r_{pq}) = \frac{\sum_{\kappa=1}^{\Gamma} a_{pq\kappa} r_{pq}^{\kappa}}{1 + \sum_{\kappa=1}^{\Lambda} b_{pq\kappa} r_{pq}^{\kappa}} \tag{10}$$

and p and q index all electrons and nuclei, and r_{pq} is the distance separating the two particles. The number of terms (Γ and Λ) is arbitrary, and depends on the quality of fit. These parameters, along with $a_{pq\kappa}, b_{pq\kappa} \in \mathbb{R}$, are input to the QMC algorithm. With this modification, our wave function is now $\Psi_{QMC}(\bar{\boldsymbol{r}}) = D_\alpha(\bar{\boldsymbol{r}}_\alpha) D_\beta(\bar{\boldsymbol{r}}_\beta) J(\bar{\boldsymbol{r}})$, and there are chain rule effects for the gradient and Laplacian. The rationale for these additional terms is the improved convergence if the wave function is a better approximation of the eigen function of $\hat{H}$ to begin with. Jastrow functions involving 3 particles were not considered here.

Within the family of QMC algorithms, there are two popular varieties. The first is called Variational Monte Carlo (VMC) in which the procedure described in this section is employed to provide an exact integration for the given wavefunction. The method is termed variational since it is commonly coupled with a wavefunction optimization step. Diffusion Monte Carlo (DMC) uses the wavefunction only as a guide. Instead of a direct integration, it has a mechanism to project out a (mostly) correct wavefunction, and thus provide exact energies for the system. That said, a DMC calculation will converge better for higher quality wavefunctions. The subject matter considered here is agnostic to this choice except that DMC includes slightly more computational effort than VMC.

* While some QMC algorithms only update one electron per Monte Carlo step, our method updates all at once[26].

4. Implementation on the GPU

The QMcBeaver[27] code, under development in our group to perform QMC calculations, was used as the CPU implementation on which to base our study of a GPU implementation. In order to locate the computationally expensive components in the code, we minimize file I/O, ignore localization procedures which lead to sparser matrices[28,29], and we only consider single determinant, restricted Hartree-Fock wavefunctions. Moving all electrons at once allows us to use the highly optimized matrix multiplication routines available in the ATLAS 3.7.11[30,31] BLAS library and use the LAPACK extension to ATLAS to perform the necessary matrix inversions. Using this representation of QMC as our starting point, we find that the computational effort on the CPU for N electrons is approximately 11% focused on the 10 dense matrix multiplications at $O(N^3)$ each, 73% on the 10 basis function set evaluations at $O(N^2)$ each, and 4% on the (electron - electron) pairwise Jastrow function evaluations at $O(N^2)$. These fractional estimates are relatively stationary for molecules with as many as 150 electrons. The leading components not yet ported to the GPU include matrix inversion and electron-nuclear Jastrow functions as well as other processes specific to DMC.

For the molecule sizes we are targeting the matrices are small and rectangular; specializations currently overlooked in GPU code. Combined with the fact that the c_{jk} matrix can be reused for all matrix multiplications, we pursued several optimization strategies in detail. In particular, all of our kernels were designed to evaluate as many walkers simultaneously as GPU hardware limitations permit.

4.1. *Walker Batch Scheme*

The GPU pipeline is very deep, so there is a substantial overhead cost for any calculation we wish to perform. This is in terms of work the GPU has to do to prepare for a given calculation, effort needed to move the GPU into full production efficiency, and any costs incurred by traversing the CPU/GPU boundary. This can be amortized by processing as many fragments simultaneously on the GPU as possible. For Monte Carlo type algorithms we can accomplish this by increasing the number of walkers processed per GPU pass. This has allowed us to tune both the size of the problem and the texture aspect ratio to the GPU. For example, we can arrange our data in GPU memory according to an empirically optimized pattern such as 4 rows by 4 columns so that each pass amounts to 16 walker evaluations in parallel.

4.2. *Basis Function Evaluation*

The number of basis functions as well as their controlling parameters are chosen according to chemical considerations. Typical are 5 basis functions for each Hydrogen and 15 basis functions for each atom Lithium to Neon, leading to a matrix aspect ratio of between 4 and 8. The choice of basis set and all associated parameters are held fixed during a run and evaluation only depends on the $3N$ electronic coordinates, producing value, gradient, and Laplacian.

4.2.1. *Kernel 1: Data Generation*

The major choice regarding basis function evaluation (Eq. (6)) concerns the organization of the output data: different regions of one output texture or separation by channel (`xyzw`) resulting in two output textures. We opted for keeping the output in different regions so as to allow specialization (*i.e.*derivatives) of the kernels. As regards input data re-use, we opted for evaluating a single basis function for 4 electrons. This choice minimizes texture lookups and increases instruction parallelism since only one n_j from Eq. (6) is used in the same fragment.

4.2.2. *Kernel 2: Layout Conversion*

Most matrix multiplication approaches on the GPU pack 2x2 submatrices into a single `xyzw` memory slot and we employed this layout as well. The basis function evaluation output is in 4x1 layout, necessitating a conversion which we used to filter out any bad values as well. Due to the batching (Section 4.1) texture layout, fences between rows and columns of walkers required special maintenance at this stage.

4.3. *Matrix Multiplication*

For purposes of performance comparison we used the ATLAS 3.7.11[30,31] library's single precision matrix multiplication on our 3 GHz Pentium 4 as a CPU benchmark. For the GPU several studies of matrix multiplication performance have been performed[7–9,11,13,14] so our main focus is on the performance for the (relatively) small rectangular matrices we encounter in our application, as well as the fact that we use the same multiplicand for all multiplications.

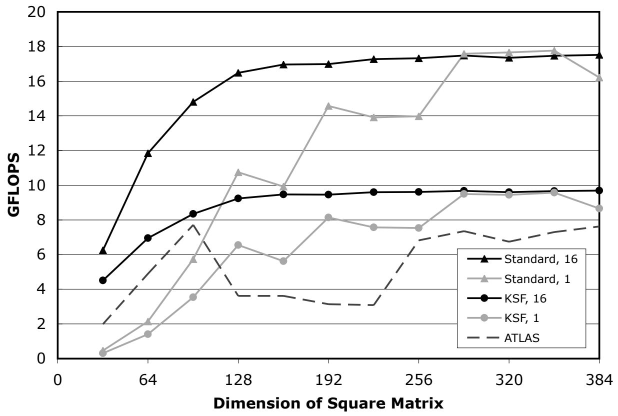

Fig. 1. The cost of correcting for the summation error in multiplication of square matrices. Indicated is the number of multiplications performed simultaneously, reusing the multiplicand.

For the 2x2 layout the inner product for the pixel at `C[i,j]` becomes the series of pixel products

```
for(k=0; k<N; k++){
 C[i,j].xyzw += A[i,k].xxzz*B[k,j].xyxy
             +  A[i,k].yyww*B[k,j].zwzw;
}
```

with `N` representing the number of pixels used in the inner product. In the GPU vector notation above, the $C[i,j].x$ data written separately is

```
C[i,j].x += A[i,k].x*B[k,j].x
         +  A[i,k].y*B[k,j].z}.
```

The values are stored in row-major format across the `xyzw` channels. This method can be modified to take advantage of multiple render target (MRT)[8] functionality on the GPU. Essentially, MRTs can take advantage of up to 4 related data structures on the GPU with which to arrange and facilitate re-use of data.

The results shown in Figures 1 and 2 both show the matrix performance speedups for a variety of matrix sizes and parameter choices. The effect of multiplying several matrices simultaneously is to raise the performance level (in terms of GFLOPS) for smaller matrices. When performing calculations using rectangular matrices, the set up costs can be quenched almost entirely. It is also apparent that for some domains, the GPU has significant performance gains relative to the CPU when CPU cache peculiarities play a role. Although the KSF error correcting

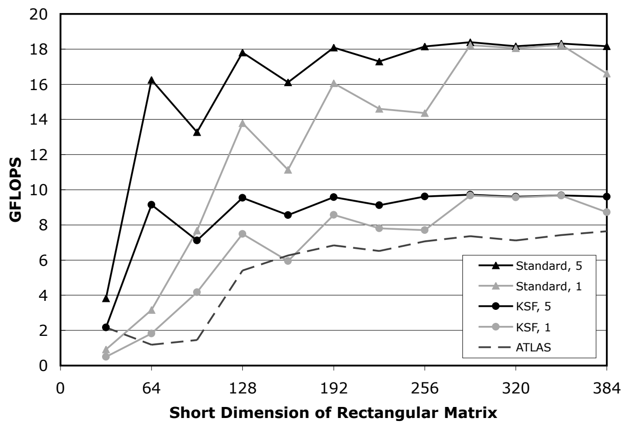

Fig. 2. The dimension of the inner product is 6 times that as the short dimension shown. The multiplicand is reused for all 5 multiplications.

algorithm (described in Section 5.2) negates most speedup gains for the particular technologies compared here, the hidden advantage remaining is that the calculation is performed on the GPU, minimizing GPU/CPU communication.

4.4. *Jastrow Functions*

The third most computationally demanding component of our QMC algorithm is the evaluation of the pairwise Jastrow function in Eq. (9). For the GPU implementation, we focused on porting the electron-electron terms (electron-nuclei terms are substantially fewer). We need to evaluate N choose 2 polynomials (one for each electron-electron pair) which are then summed. Since parameters in Eq. (10) differ between same/opposite spin electron pairs, texture data is partitioned in order to allow kernel specialization.

We proceed in 3 steps:

Kernel 1 evaluates the magnitude and normalized vector between all pairs of electrons for a total of 4 values per fragment.

Kernel 2 finds the value, Laplacian, and gradient of Eq. (9), writing the first two to one texture and the latter three to another.

Kernel 3 computes the sums, maintaining the electron indices for the gradient summands.

5. GPU Floating Point Error

One of the goals of quantum chemistry is the calculation of the electronic energy of a molecule with sufficient accuracy, stated as 1 to 2 kcal/mol. To this end absolute error of the final result must not be worse than 1×10^{-3} Hartrees. An appropriately parameterized QMC calculation can meet this criterion given enough Monte Carlo iterations. For this study, we want to consider whether single precision is satisfactory. To test this, three simple DMC calculations were performed on a large CPU cluster to compare numerically a result calculated in double precision with exactly the same calculation in single precision. First, a calculation is performed on a Helium atom using a 17s basis set[32] and a 2 determinant expansion in natural orbitals obtained using GAMESS[33]. Figure 3 shows that the single and double precision results are very similar, where the exact answer is approximately -2.903724[34] hartrees. Second, the torsional barrier in ethane was studied using the cc-pCVTZ[35] basis set with CCSD(T) optimized Eclipsed and Gauche configurations[36]. Figure 4 again shows similar results between single and double precision, where the experimental value is 2.73 kcal/mol[36]. While these results are by no means conclusive, especially since the quality of the result is dependent upon the quality of the wavefunction, they provide evidence that single precision is not altogether unreasonable. This is can be seen since the iterates are decoupled to some degree from each other by random numbers, and since the Monte Carlo statistics itself happens in double precision. Furthermore, if a pathological electronic configuration is identified, it can always be more delicately handled on the CPU in double precision. Lastly, single precision QMC calculations might be useful in an independent VMC wavefunction optimization calculation. Since DMC only employs the wavefunction as a guide, variationally optimized parameters are far less restrictive in terms of precision.

As far as our nVidia 7800 GTX GPU is concerned, we studied the floating point error to obtain a best estimate for single point evaluations. We considered two principal sources of error relevant to our problem as compared to the level of error available on a CPU: underflow and effects of rounding. The evaluation of basis functions (Eq. (6)), for example, can easily underflow if the b_{n_j} are too negative. We investigated whether the lack of de-normals on GPUs was a problem since this means a GPU will underflow faster than a CPU. As regards rounding, the IEEE floating point standard calls for a relative error of $\pm 0.5 \times 10^{-7}$ in the basic arithmetic operations for single precision. On current GPUs the relative error in these operations appears to be[37] at least $\pm 0.5 \times 10^{-7}$ and $\pm 1.0 \times 10^{-7}$. For dense linear al-

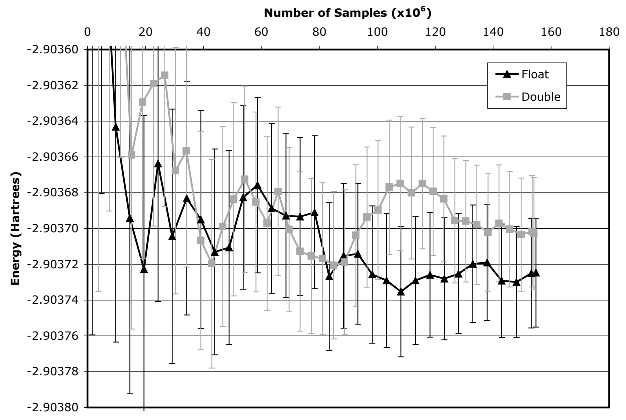

Fig. 3. Helium calculation showing the average and the error as the calculation progresses. The calculation was done at dt = 0.001, with 200 walkers each on 128 CPU processors.

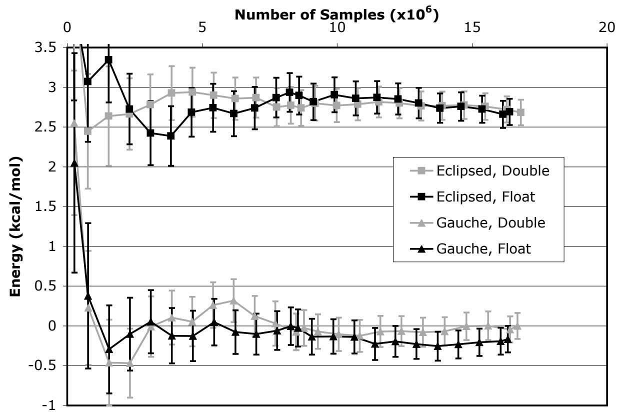

Fig. 4. Ethane calculation showing the average and the error as the calculation progresses. The calculation was done at dt = 0.005, with 200 walkers each on 128 CPU processors.

gebra this yields a difference in error between CPU and GPU computed results.

5.1. *Underflow Corrections*

To begin with, it is questionable whether one would permit de-normals to be included in calculations even on some CPUs. Many processor manufacturers elect software implementations of de-normals, which severely penalize the processing speed. Since we were unable to get decent timing results in matrix multiplication on the CPU unless de-normals were flushed to zero before multiplication, our performance comparisons actually already represent a lack of de-normals on both processors.

Basis function evaluation involves exponentials with arguments negative enough to cause underflow, an effect we do not want to ignore. To avoid underflow error one may simply scale relevant variables to avoid the de-normal range, but must do so carefully to avoid the worse problem of overflow. The effect of this type of error depends heavily on the distribution of parameters, which is highly specific to our application. Thus we measured the effect of these shifts on the final calculated $E_L(\bar{\boldsymbol{r}})$ for each iteration, compared to the same calculation as performed on the CPU in double precision.

The effect of shifting the exponential turns out to be relatively small for the set of parameters we considered. We conclude that shifting helps, but the lack of de-normals on the GPU turned out not to be a significant source of error. For parameter sets which consistently produce de-normals, single precision should probably be avoided entirely.

5.2. *Kahan Method*

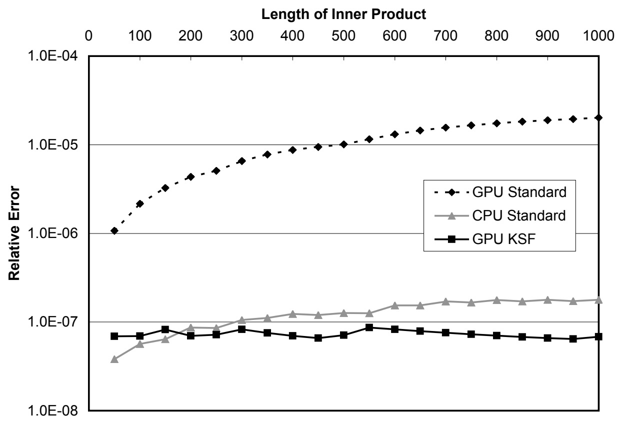

Fig. 5. KSF corrects for rounding error in matrix multiplication. The resultant matrix is 1000x1000, and the operand data is sampled from a uniform distribution [0,1].

Dense matrix multiplication is the most significant source of error in our computations when run on the GPU. Figure 5 shows the roundoff error inherent in matrix multiplication, as estimated by multiplying two matrices created with a uniform distribution of data. As a function of the dimension of the inner product, we calculate the relative error averaged over all the elements in the resultant 1000x1000 matrix using CPU double precision as our reference data. The problem is due to the propagation of errors, which scales approximately linearly with the length of the inner products. A CPU typically mini-

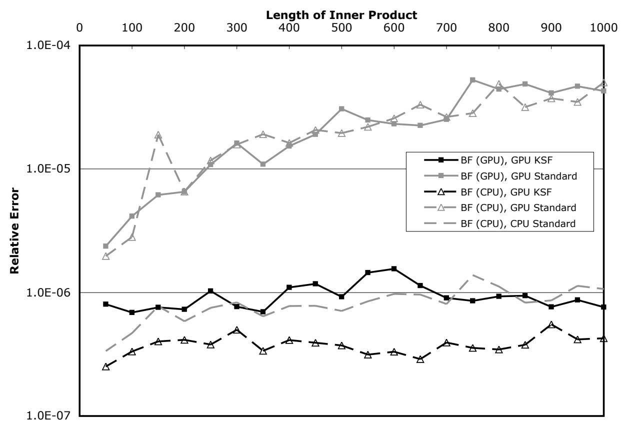

Fig. 6. The "QMC-Distributed" data for the multipliers was generated either on the CPU or on the GPU, and the matrix multiplication was either corrected using KSF or left as the standard method

mizes this by performing the calculations at a higher precision than the data type.

When summing a sequence of floating point numbers using the basic formula $\sum x_j$, the floating point result is $\sum x_j(1+\delta_j)$, where the perturbation error is defined as $|\delta_j| < (n-j)\epsilon$ and ϵ is the machine error. To compensate for the propagation of errors we use the Kahan summation formula (KSF)[38,39] in the context of matrix multiplication. This alternative method for summing a sequence of n numbers is shown below:

```
S = x[1];
C = 0;
for(j=2; j<=n; j++){
  Y = x[j] - C;
  T = S + Y;
  C = (T - S) - Y;
  S = T;
}
```

This method is algebraically equivalent, but if these steps are preserved during compilation, the algorithm has the power to produce the result $\sum x_j(1+\delta_j) + O(n\epsilon^2)\sum |x_j|$ where $|\delta_j| \leq 2\epsilon$[40]. To explain this algorithm, one first observes that the low order bits of Y are lost when adding it to S. These bits can be recovered with the correction term C. The value for C is found by subtracting Y from the part of Y which is properly accounted for in the sum (the parenthesis are critical). This is not the only summation improvement available although it does compete well[41].

A simple modification makes the KSF suitable for use in matrix multiplications as shown in Algorithm 2. Here (i, j) represents the coordinates of the element in the product matrix we are working on. It is important to note that the propagation error in addition is corrected for, but not any error due to multiplication, even though such corrections are possible[42]. However, as Figure 5 shows, the improvement is enough to even beat single precision on the CPU for long enough inner products.

Algorithm 2 KSF-corrected GPU Matrix Multiplication

```
float4 T = 0, C = 0, Y = 0, S=0;
int j = 0;
while(j < N){
  Y = A[i,k].xxzz*B[k,j].xyxy - C;
  T = S + Y;
  C = (T - S) - Y;
  S = T;
  Y = A[i,k].yyww*B[k,j].zwzw - C;
  T = S + Y;
  C = (T - S) - Y;
  S = T;
  j++;
}
return S;
```

To estimate the improvement that KSF provides for our QMC methods, we move to a "QMC distribution" of data for our multiplier matrices while keeping the multiplicand (representing c_{jk}) as uniformly random matrix. The distribution was formed by generating a representative set of basisfunction parameters and a psuedo-random configuration of electrons. This distribution was evaluated either on the GPU or on the CPU and then sent to the GPU for multiplication. The relative error was again estimated against double precision on the CPU. Although the results in Figure 6 have a higher variance, it shows that using the KSF method, we are able to approximately obtain equivalent results as CPU single precision.

6. Results

To test the GPU port of our code, we sample 7 arbitrary molecules spanning the range over which we wish to measure performance. We present speedup estimates for the calculation time spent on equivalent tasks performed on both our 7800 GTX GPU and our 3GHz Pentium 4, as well as compare the final cost of incorporating the KSF correction. We ran the calculations long enough to converge the speedup ratio.

It is evident that for the range of molecules considered, the speed penalty incurred with KSF rose as the matrix multiplication cost became more prominent. The KSF formula served to keep the relative error in the calculated $E_L(\bar{\boldsymbol{r}})$ to a constant across all molecules at approximately $1x10^{-6}$. It is worth noting that KSF did not make a significant difference in either speed nor correction for many of the smaller molecules.

		Number of	Number of	Total Speedup		Basisfunction	Jastrow
Name	Formula	Electrons	Basisfunctions	Standard	KSF	Speedup	Speedup
Acetic acid	CH_3COOH	32	80	3.2	3.1	18.2	0.7
Benzaldehyde	C_6H_5CHO	56	150	4.4	4.1	25.9	2.1
[10]Annulene	$C_{10}H_{10}$	70	200	6.3	5.6	30.2	3.4
Diazobenzene	$C_{12}H_{10}N_2$	96	326	5.3	4.5	31.6	6.4
Lysine	$C_6H_{14}N_2O_2$	102	280	4.5	3.9	29.2	7.2
Arginine	$C_6H_{14}N_4O_2$	116	387	4.9	4.1	28.5	9.3
HMX	$C_4H_8N_8O_8$	152	516	6.6	5.3	33.3	14.0

Fig. 7. QMC performance results on arbitrary molecules picked to represent varying problem sizes. Speedup is defined as the time spend processing on the CPU divided by the time spend processing on the GPU.

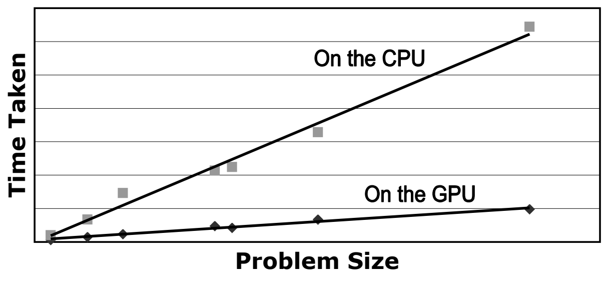

Fig. 8. Problem size is defined as the number of basisfunctions × the number of electrons. The data points are from the arbitrary molecules listed in Figure 7

To provide an estimate for the impact of these speedup factors, we point out that for HMX, the calculation is now 5 to 7 times faster. This means that the new fractions of evaluation cost are that matrix multiplication, which formerly composed 15% of the cost is now only 4% (non-KSF) of the original total cost, the basis function cost went from 73% to 2.2%, and the electron-electron Jastrow evaluations which used to cost 3.5% of the effort are now 0.3%. If we approximate the effect of improving GPU technology over CPU technology as well as the possibility of multiple GPUs per CPU by setting the residual percentages at 0%, the original unaccounted for 8% suggests a theoretical factor of 13 speedup. A recent calculation[43] on free-base porphyrin which has 162 electrons and 938 basis functions in the cc-pVDZ basis set cost 40,000 CPU hours on an IBM SP POWER3+ cluster. Thus, ignoring the precision issue, we speculate that this calculation could theoretically cost 3,000 processor hours.

Although some of the performance numbers for the individual kernels are very good, the code suffers from Amdahl's Law type inefficiencies because of diminishing returns discovered during porting. This is for several reasons. A few of the elements of the computation, like the Monte Carlo statistical manipulations, can not be permitted to be run in single precision. Furthermore, there are several portions of the code for which a GPU port is currently unsuitable due to a lack of sufficient data parallelism either as $O(N)$ components or as problems with GPU-unfriendly data interdependencies. With increasing capability on the GPU, more of the code will be available to porting considerations.

It is obvious however that there is a GPU kind of Gustafson's Law[44] advantage available. Specifically, if basis function and Jastrow function evaluations can be considered as essentially free, then one is encouraged to employ whatever functional form is deemed best, regardless of computational complexity. This is likely to increase both the quality of individual iterates as well as improve the overall convergence characteristics of a Monte Carlo calculation. Of course this assumes that these advantages are not washed out by precision errors stemming from other parts of the code.

7. Conclusion

QMC type algorithms for first principles chemistry calculations are simple to parallelize and capable of exploiting the *data parallel* aspects of GPU based computing. While the matrix sizes needed in actual application practice are on the small side, recent generation GPUs coupled with a few tricks have become significantly better in achieving high performance at these sizes. The overall result is a 3x to 6x speedup in the end to end simulation application with a modest increase in hardware cost, making this a very cost effective solution. The lack of full IEEE floating point support is perhaps the most critical issue for QMC. We were able to correct for the error propagation, albeit only with a performance penalty due to the more complex evaluation cost of the Kahan summation formula. Clearly a more complete IEEE floating point treatment would be an excellent improvement, and forthcoming improvements will be welcomed.

Beyond that, we note that due to the rapid evolution of GPU hardware (and the associated driver software), attaining a sweet spot in the performance landscape is a never ending quest of parameter and algorithm tweaking. We speculate that adoption of the GPU as a computational engine will be greatly facilitated if approaches such as ATLAS [8,30] and application specific libraries can be further brought to the GPU arena.

8. Acknowledgments

We would like to thank nVidia for providing the 7800 GTX used in this study. This research has been supported in part by the DOE (W-7405-ENG-48/B341492 and DE-FG02-04ER25657) and the Caltech Center for Mathematics of Information.

References

[1] Matt Pharr, editor. *GPU Gems 2*. Addison-Wesley, 2005.

[2] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proc. of ACM/IEEE Superc. Conf.*, page 47, 2004.

[3] J. Georgii and R. Westermann. A multigrid framework for real-time simulation of deformable bodies. *Computers & Graphics*, 30(3), 2006.

[4] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision fem simulations with gpus. In *Proc. of ASIM 2005*, pages 139–144, 2005.

[5] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. of ACM/IEEE Superc. Conf.*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[6] nVidia. Compute unified device architecture. http://developer.nvidia.com/cuda.

[7] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *Proc. of ACM/IEEE Superc. Conf.*, Washington, DC, USA, 2006. IEEE Computer Society.

[8] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. *PACT'05*, pages 185–196, 2005.

[9] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proc. of ACM/EG Conf. on Graph. HW*, pages 133–137, 2004.

[10] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[11] J. D. Hall, N. A. Carr, and J. C. Hart. Cache and Bandwidth Aware Matrix Multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois, 2003.

[12] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[13] Á. Moravánszky. Dense matrix algebra on the gpu, 2003. NovodeX AG.

[14] S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware, 2001.

[15] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proc. of ACM/EG Conf. on Graph. HW*, pages 102–111, 2003.

[16] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. In *Proc. of ACM/IEEE Superc. Conf.*, volume 52, pages 654–663, 2005.

[17] P. A. Heng J. Q. Wang, T. T. Wong and C. S. Leung. Discrete wavelet transform on gpu. In *ACM Workshop on GPGPU*, 2004.

[18] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proc. of ACM/EG Conf. on Graph. HW*, pages 112–119, 2003.

[19] D. M. Ceperley and B. J. Alder. Quantum Monte Carlo. *Science*, 231(4738):555–560, 1986.

[20] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[21] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *Course Notes*, 2004.

[22] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hard-

ware. In *Proc. of ACM SIGGRAPH*, 2004.

[23] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh.* AK Peters, Ltd., 2004.

[24] P. J. Reynolds, D. M. Ceperley, B. J. Alder, and W. A. Lester. Fixed-node Quantum Monte Carlo for molecules. *Journal of Chemical Physics*, 77(11):5593–5603, 1982.

[25] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087, 1953.

[26] C. J. Umrigar, M. P. Nightingale, and K. J. Runge. A diffusion Monte Carlo algorithm with very small time-step errors. *Journal of Chemical Physics*, 99(4):2865–2890, 1993.

[27] D.R. Kent IV, M.T. Feldmann, D. R. Fisher, and A.G. Anderson. QM^{C}Beaver v20051021 ©, 2001-2007.

[28] AJ Williamson, R.Q. Hood, and JC Grossman. Linear-Scaling Quantum Monte Carlo Calculations. *Physical Review Letters*, 87(24):246406, 2001.

[29] A. Aspuru-Guzik, R. Salomon-Ferrer, B. Austin, and W.A. Lester. A sparse algorithm for the evaluation of the local energy in quantum Monte Carlo. *Journal of Computational Chemistry*, 26(7):708–715, 2005.

[30] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[31] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proc. of ACM/IEEE Superc. Conf.*, 1998.

[32] S. Huzinaga, E. Miyoshi, and M. Sekiya. A method of generating an effective orbital set for configuration interaction calculations. *The Journal of Chemical Physics*, 100(2):1435–1439, 1994.

[33] Michael W. Schmidt, Kim K. Baldridge, Jerry A. Boatz, Steven T. Elbert, Mark S. Gordon, Jan H. Jensen, Shiro Koseki, Nikita Matsunaga, Kiet A. Nguyen, Shujun Su, Theresa L. Windus, Michel Dupuis, and Jr. John A. Montgomery. General atomic and molecular electronic structure system. *J. Comput. Chem.*, 14(11):1347–1363, 1993.

[34] C. Schwartz. Experiment and theory in computations of the he atom ground state. *Int. J. of Mod. Phys.*, 15(4):877–888, 2006.

[35] David E. Woon and Jr. Thom H. Dunning. Gaussian basis sets for use in correlated molecular calculations. v. core-valence basis sets for boron through neon. *The Journal of Chemical Physics*, 103(11):4572–4585, 1995.

[36] Attila G. Csaszar, Wesley D. Allen, and Henry F. Schaefer III. In pursuit of the ab initio limit for conformational energy prototypes. *Journal of Chemical Physics*, 108(23):9751–9764, 1998.

[37] K. E. Hillesland and A. Lastra. Gpu floating-point paranoia. In *GP*^{2}, pages C–8, 2004.

[38] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Com. of the ACM*, 8(1):40–40, 1965.

[39] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[40] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms.* Addison-Wesley, 1997.

[41] John Michael McNamee. A comparison of methods for accurate summation. *SIGSAM Bull.*, 38(1):1–7, 2004.

[42] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[43] A. Aspuru-Guzik, O. El Akramine, J.C. Grossman, and W.A. Lester Jr. Quantum Monte Carlo for electronic excitations of free-base porphyrin. *Journal of Chemical Physics*, 120(7):3049–3050, 2004.

[44] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.