Accelerating Correlated Quantum Chemistry Calculations Using Graphical Processing Units

Accelerating correlated quantum chemistry calculations using graphical processing units

Mark A. Watson, Roberto Olivares-Amaya, Richard G. Edgar, and Alán Aspuru-Guzik*
Department of Chemistry and Chemical Biology, Harvard University, Cambridge, Massachusetts 02138.

Tomás Arias
Laboratory of Atomic and Solid State Physics, Cornell University, Ithaca, New York 14853.

Graphical processing units are now being used with dramatic effect to accelerate quantum chemistry applications. The authors give a brief introduction to electronic structure methods and describe their efforts to accelerate a correlated quantum chemistry code. They propose and analyze two new tools for accelerating matrix-multiplications where single-precision accuracy is insufficient.

I. INTRODUCTION

With the advent of modern quantum theory a century ago, scientists quickly realized that quantum mechanics could offer a predictive theory of chemistry, revolutionizing the subject in the same way that Newton's laws had transformed the study of classical mechanics. Over the intervening decades, we have witnessed an exponential increase in available computing power. Coupled with tremendous advances in theory and algorithmic methods, as well as the painstaking development of sophisticated program suites often consisting of millions of lines of code, the scope of phenomena now amenable to the predictive techniques of quantum chemistry is large and continually growing.

Modern computational chemistry has an important role to play in many practical applications, such as the discovery of new drugs or industrial catalysts, and the development of new materials or technologies to meet global energy and environmental challenges. Its widespread application has resulted in major efforts to reduce its computational cost: accurate quantum chemistry methods consume a significant fraction of computing resources at national laboratories.

As a result, we are witnessing a new era in the optimization of quantum chemistry codes following an explosion of interest in the utilization of coprocessors such as graphical processing units (GPUs). This interest in GPUs and other accelerators is largely driven by their combination of formidable performance and relatively low cost. But another key reason for their emergence in scientific fields was the release of NVIDIA's CUDA (compute unified device architecture) toolkit that dramatically simplified the process of developing code for GPUs.

Already, GPUs have started to be used by computational chemists to treat a wide range of problems. These include molecular dynamics and quantum Monte Carlo simulations, density-functional theory and self-consistent field calculations [1, 2] as well as correlated quantum chemistry applications [3, 4]. Efficiency gains of between one and three orders of magnitude have been reported compared to conventional implementations on a CPU. Thus new domains of scientific application are now possible where, previously, extremely expensive and rare supercomputing facilities would have been required.

A very important area for many scientific applications is the acceleration of linear algebra operations, which are quite well-suited for GPU architectures. Included in the CUDA software development toolkit is an implementation of the BLAS linear algebra library, named CUBLAS. By simply replacing the BLAS `*GEMM` routines with corresponding CUBLAS `SGEMM` calls to accelerate key matrix multiplications, our group [3] was able to achieve a speedup of 4.3x when calculating the RI-MP2 (resolution-of-the-identity second-order Møller-Plesset perturbation theory [5, 6]) correlation energy of doeicosane ($C_{22}H_{46}$).

This initial effort was one of the first quantum chemistry applications to leverage GPUs, but it revealed several issues for future work. For example, while modern GPU cards designed for research can have up to 4 GiB of RAM, consumer level cards may have as little as 256 MiB. Without some means to overcome the memory bottleneck, our first attempts to use GPUs to accelerate RI-MP2 calculations were limited to systems with less than 600 basis functions.

Another issue is numerical precision. The vast majority of GPU cards currently in use worldwide support only single-precision arithmetic. Single precision is generally insufficient to achieve 'chemical accuracy' of 1 kcal/mol in calculations on anything but the smallest and simplest systems, since the errors quickly accumulate for larger molecules. An interesting topic in the computer science [7, 8] community has been the development of libraries which achieve precision beyond the hardware specification through algorithmic techniques. In this work, we consider this problem in detail for the special case of single-precision GPU devices and applications to quantum chemistry.

In summary, we describe our efforts to develop tools for the GPU acceleration of correlated quantum chem-

*Electronic address: `aspuru@chemistry.harvard.edu`

istry calculations [3, 4]. We address the issues of limited GPU device memory, as well as achieving higher accuracy using only single-precision GPU operations. We begin in section II with an overview of basic quantum chemistry theory, followed by the algorithms behind our new libraries. Next we report the efficiency of our methods for general matrix multiplications and in the context of accelerating quantum chemistry calculations. We end the article with a brief conclusion and our future perspective.

II. QUANTUM CHEMISTRY THEORY

Traditional quantum chemistry strives to solve the time-independent Schrödinger equation,

$$\hat{H}(\mathbf{r},\mathbf{R})\psi(\mathbf{r},\mathbf{R}) = E(\mathbf{R})\psi(\mathbf{r},\mathbf{R}) \tag{1}$$

where $\psi(\mathbf{r},\mathbf{R})$ is the electronic wavefunction for a molecular system defined by the Hamiltonian $\hat{H}(\mathbf{r},\mathbf{R})$ in terms of a set of coordinates for the electrons, $\mathbf{r}$, and nuclei, $\mathbf{R}$. The total molecular energy is given by the eigenvalue $E(\mathbf{R})$ and is often referred to as the *potential energy surface*. It is parameterized in terms of the nuclear coordinates since we have assumed the Born-Oppenheimer approximation, and is fundamental to the *ab initio* theory of chemical reaction dynamics.

The solution of eqn. (1) yields the electronic wavefunction for a given nuclear configuration, and in turn the probability density for finding an electron at a given point in space. Exact solution is intractable, even numerically, for all but the simplest systems due to the formidable nature of the many-body problem inherent in the form of $\hat{H}(\mathbf{r},\mathbf{R})$, which describes not only the electronic kinetic energy, but also the complex Coulomb interactions between the electrons and nuclei. Only for one-electron systems, where there is no electron-electron coupling, are exact solutions readily available.

Nevertheless, it is a hallmark of quantum chemistry that there is a well-defined hierarchy of methods for solving eqn. (1) approximately. Indeed, given sufficient computational power, a solution may be improved systematically to yield a wavefunction of arbitrary numerical accuracy. The simplest method in the hierarchy is a mean-field approach known as Hartree-Fock (HF) theory.

In HF theory, we write the electronic wavefunction for an N-electron system as an antisymmetrized product of *molecular orbitals*, $\phi(\mathbf{r}_i)$, which are wavefunctions for a single electron under a one-electron Hamiltonian known as the *Fock operator*,

$$\hat{f}(\mathbf{r},\mathbf{R}) = \hat{h}(\mathbf{r},\mathbf{R}) + \sum_{j=1}^{N/2}\left[2\hat{J}_j(\mathbf{r}) - \hat{K}_j(\mathbf{r})\right] \tag{2}$$

The Coulomb, $\hat{J}_j(\mathbf{r})$, exchange, $\hat{K}_j(\mathbf{r})$, and one-electron, $\hat{h}(\mathbf{r})$, operators are given by

$$\hat{h}(\mathbf{r},\mathbf{R}) = -\frac{1}{2}\nabla^2 + v(\mathbf{r},\mathbf{R}) \tag{3}$$

$$\hat{J}_j(\mathbf{r}) = \int \frac{\phi_j^*(\mathbf{r}')\phi_j(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\mathrm{d}\mathbf{r}' \tag{4}$$

$$\hat{K}_j(\mathbf{r})\phi_i(\mathbf{r}) = \int \frac{\phi_j^*(\mathbf{r}')\phi_i(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\mathrm{d}\mathbf{r}'\phi_j(\mathbf{r}) \tag{5}$$

where $v(\mathbf{r},\mathbf{R})$ is the nuclear potential. The associated Hartree-Fock equations,

$$\hat{f}\phi_i(\mathbf{r}) = \epsilon_i\phi_i(\mathbf{r}) \tag{6}$$

permit a complete set of eigenfunctions. In the simplest case where N is even, each of the $N/2$ orbitals corresponding to the lowest eigenvalues is associated with two electrons of opposite spin. These *occupied* orbitals are used to construct the many-electron wavefunction; they are labelled by the subscripts $i, j, \ldots$ The remaining functions form the *virtual* orbitals and are indicated by the subscripts $a, b, \ldots$.

Although in some implementations, the Hartree-Fock equations are solved numerically, the traditional approach is to expand the molecular orbitals (MOs) in a basis of *atomic orbitals*, $\{\eta_\alpha\}$,

$$\phi_p(\mathbf{r}) = \sum_{\alpha}^{N_b} c_{\alpha p}\eta_\alpha(\mathbf{r}) \tag{7}$$

The optimized coefficients, $c_{\alpha p}$, and orbital energies, ϵ_p, are determined from the secular equations, which in matrix notation can be written as

$$\mathbf{FC} = \mathbf{SC}\epsilon \tag{8}$$

where $\mathbf{S}$ is the atomic orbital (AO) overlap matrix, $\langle\eta_\alpha|\eta_\beta\rangle$, $\mathbf{F}$ is the AO Fock matrix, $\langle\eta_\alpha|\hat{f}|\eta_\beta\rangle$, $\mathbf{C}$ is the matrix of MO coefficients and ϵ is the diagonal matrix of MO energies. These are the celebrated Roothaan-Hall self-consistent field equations.

The Hartree-Fock solution (in a complete AO basis) generally recovers more than 99% of the exact energy, which is remarkable. The neglected energy is known as the *correlation energy*, and its accurate and efficient recovery is the central challenge in quantum chemistry. Indeed, for the purposes of predictive chemistry, we are largely interested in energy differences which are of similar magnitude to the correlation energy: about 0.04 E_h (100 kJ mol^{-1}) for two electrons in a doubly occupied orbital.

Currently the most popular approach for solving eqn. (1) including electron correlation, is density functional theory (DFT). DFT, which bypasses explicit construction of the many-body wavefunction and focuses only on the much simpler 3-dimensional electron density as the basic variable, is extremely useful due its favorable balance between accuracy and efficiency.

Instead, we examine in this article another approach: a widely-used and computationally efficient correlated wavefunction-based method, known as second-order Møller-Plesset perturbation theory (MP2)[5]. MP2 is known to produce equilibrium geometries of comparable accuracy to density functional theory (DFT) and in particular is able to capture long-range correlation effects such as the dispersion interaction. For many weakly bound systems where DFT results are often questionable, MP2 is essentially the least expensive and most reliable alternative.

If HF theory can be thought of as a first-order solution to eqn. (1), then MP2 theory is the second-order solution within the framework of perturbation theory, where the many-electron Hamiltonian is partitioned as

$$\hat{H} = \sum_i \hat{f}(\mathbf{r}_i) + \lambda \hat{H}^{(1)} \tag{9}$$

We have introduced an order parameter, λ, to expand the energy and wavefunction,

$$E = E^{(0)} + \lambda E^{(1)} + \lambda^2 E^{(2)} + \cdots \tag{10}$$

$$\Psi = \Psi_{\mathrm{HF}} + \lambda \Psi^{(1)} + \lambda^2 \Psi^{(2)} + \cdots \tag{11}$$

where Ψ_{HF} is the zero-order (Hartree-Fock) wavefunction, and the zero and first-order energies are given by

$$E^{(0)} = \sum_i \epsilon_i \tag{12}$$

$$E^{(0)} + \lambda E^{(1)} = \langle \Psi_{\mathrm{HF}} | \hat{H} | \Psi_{\mathrm{HF}} \rangle \tag{13}$$

The second-order (MP2) correlation energy takes the form

$$E^{(2)} = \sum_{ijab} \frac{(ia|jb)^2 + \frac{1}{2}\left[(ia|jb) - (ib|ja)\right]^2}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{14}$$

where the MO integrals,

$$(ij|ab) = \sum_{\mu\nu\lambda\sigma} C_{\mu i} C_{\nu j} C_{\lambda a} C_{\sigma b} (\mu\nu|\lambda\sigma) \tag{15}$$

are obtained by transforming the AO integrals,

$$(\mu\nu|\lambda\sigma) = \int\int \frac{\eta_\mu(\mathbf{r}_1)\eta_\nu(\mathbf{r}_1)\eta_\lambda(\mathbf{r}_2)\eta_\sigma(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|} \mathrm{d}\mathbf{r}_1 \mathrm{d}\mathbf{r}_2 \tag{16}$$

One way to considerably reduce the computational cost of MP2 calculations is to exploit the linear dependence in the product space of atomic orbitals. This allows us to expand products of AOs as linear combinations of atom-centered auxiliary basis functions, P,

$$\rho_{\mu\nu}(\mathbf{r}) = \eta_\mu(\mathbf{r})\eta_\nu(\mathbf{r}) \approx \tilde{\rho}_{\mu\nu}(\mathbf{r}) = \sum C_{\mu\nu,P} P(\mathbf{r}) \tag{17}$$

and to therefore approximate eqn.(16) in terms of only 2 and 3-index quantities as

$$\widetilde{(\mu\nu|\lambda\sigma)} = \sum_{P,Q} (\mu\nu|P)(P|Q)^{-1}(Q|\lambda\sigma) \tag{18}$$

The idea is equivalent to an approximate insertion of the resolution-of-the-identity (RI),

$$I = \sum_m |m)(m| \approx \sum_{P,Q} |P)(P|Q)^{-1}(Q| \tag{19}$$

from which the name RI-MP2 is derived.

Our work is implemented in a development version of Q-Chem 3.1 [9], where the RI-MP2 correlation energy is evaluated in five steps, as described elsewhere [3]. Previously we showed that steps 3 and 4, the formation of the approximate MO integrals, were by far the most expensive operations for medium to large-sized systems, and require the matrix multiplications

$$\widetilde{(ia|jb)} \approx \sum_Q B_{ia,Q} B_{jb,Q} \tag{20}$$

and

$$B_{ia,Q} = \sum_P (ia|P)\,(P|Q)^{-1/2} \tag{21}$$

These are the two operations we will concentrate on accelerating in this work.

III. GPU ACCELERATION OF GEMM

In this section we first describe our cleaving algorithm to allow matrix multiplications of arbitrary size to be accelerated on the GPU (assuming sufficient CPU memory). Next, we propose two different algorithms for the MGEMM ('mixed-precision general matrix multiply') library, using two different schemes to partition the matrices into simpler components.

A. Cleaving GEMMs

Consider the matrix multiplication

$$\mathbf{C} = \mathbf{A}\mathbf{B} \tag{22}$$

where $\mathbf{A}$ is an $(m \times k)$ matrix, $\mathbf{B}$ is an $(k \times n)$ matrix, and $\mathbf{C}$ an $(m \times n)$ matrix. We can divide $\mathbf{A}$ into a column vector of $r+1$ matrices

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_r \end{pmatrix} \tag{23}$$

where each entry $\mathbf{A}_i$ is a $(p_i \times k)$ matrix, and $\sum_i^r p_i = m$. In practice, all the p_i will be the same, with the possible exception of p_r, which will be an edge case. In a similar manner, we can divide $\mathbf{B}$ into a row vector of $s+1$ matrices

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_0 & \mathbf{B}_1 & \cdots & \mathbf{B}_s \end{pmatrix} \tag{24}$$

where each $\mathbf{B}_j$ is an $(k \times q_j)$ matrix and $\sum_j^s q_j = n$. Again all the q_j will be the same, with the possible exception of q_s. We then form the outer product of these two vectors

$$\mathbf{C} = \begin{pmatrix} \mathbf{A}_0 \\ \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_r \end{pmatrix} \begin{pmatrix} \mathbf{B}_0 & \mathbf{B}_1 & \cdots & \mathbf{B}_s \end{pmatrix} \quad (25)$$

$$= \begin{pmatrix} \mathbf{A}_0\mathbf{B}_0 & \mathbf{A}_0\mathbf{B}_1 & \cdots & \mathbf{A}_0\mathbf{B}_s \\ \mathbf{A}_1\mathbf{B}_0 & \mathbf{A}_1\mathbf{B}_1 & & \mathbf{A}_1\mathbf{B}_s \\ \vdots & & \ddots & \\ \mathbf{A}_r\mathbf{B}_0 & & & \mathbf{A}_r\mathbf{B}_s \end{pmatrix} \quad (26)$$

Each individual $\mathbf{C}_{ij} = \mathbf{A}_i\mathbf{B}_j$ is an $(p_i \times q_j)$ matrix, and can be computed independently of all the others. Generalizing this to a full `*GEMM` implementation, which includes the possibility of transposes being taken, is tedious but straightforward.

We have implemented this approach for the GPU as a complete replacement for `*GEMM`. The p_i and q_j values are chosen such that each sub-multiplication fits within the currently available GPU memory. Each multiplication is staged through the GPU, and the results assembled on the CPU. This process is hidden from the user code, which simply sees a standard `*GEMM` call. In this way, we are able to multiply matrices of arbitrary size using `MGEMM` regardless of the available GPU memory.

B. Bitwise `MGEMM` algorithm

Consider the partitioning of a double-precision (DP) floating point number, $A = m * 2^k$,

$$A \approx A^{\mathrm{u}} + A^{\mathrm{l}} \quad (27)$$

where A^{u} and A^{l} are single-precision (SP) numbers storing the uppermost n_u, and the next lowest n_l, significant bits of m, respectively. Consider next the multiplication of two scalars, A, B. Applying the bitwise partitioning, we can approximate the full DP multiplication as four SP multiplications,

$$AB \approx A^{\mathrm{u}}B^{\mathrm{u}} + A^{\mathrm{u}}B^{\mathrm{l}} + A^{\mathrm{l}}B^{\mathrm{u}} + A^{\mathrm{l}}B^{\mathrm{l}} \quad (28)$$

where the result of each SP multiplication is accumulated in DP. For eqn. (28) to be exact, n_u and n_l must now be sufficiently small such that no round-off error occurs when storing each of the four multiplications in SP.

Anticipating the performance of eqn. (28), we can introduce an alternative approximation, involving only three SP multiplications,

$$AB \approx A^{\mathrm{u}}B^{\mathrm{u}} + A^{\mathrm{u}}B^{\mathrm{l}} + A^{\mathrm{l}}(B^{\mathrm{u}} + B^{\mathrm{l}}) \quad (29)$$

where $B^{\mathrm{u}}+B^{\mathrm{l}}$ is replaced by the SP cast of B. In general, we can expect the round-off error associated with $A^{\mathrm{l}}B^{\mathrm{u}}$ to be of the same order of magnitude, on average, as the round-off error associated with $A^{\mathrm{l}}(B^{\mathrm{u}} + B^{\mathrm{l}})$.

Finally, we can generalize eqn. (29) for the matrix multiplication,

$$\mathbf{AB} \approx \mathbf{A}^{\mathrm{u}}\mathbf{B}^{\mathrm{u}} + \mathbf{A}^{\mathrm{u}}\mathbf{B}^{\mathrm{l}} + \mathbf{A}^{\mathrm{l}}(\mathbf{B}^{\mathrm{u}} + \mathbf{B}^{\mathrm{l}}) \quad (30)$$

where, for each element of $\mathbf{X} \in \{\mathbf{A}, \mathbf{B}\}$,

$$X_{ij} \approx X^{\mathrm{u}}_{ij} + X^{\mathrm{l}}_{ij} \quad (31)$$

As above, the final term may be considered as either two separate multiplications, or as a single multiplication where $\mathbf{B}^{\mathrm{u}} + \mathbf{B}^{\mathrm{l}}$ is pre-summed. All the multiplications may be evaluated efficiently using the CUBLAS `SGEMM` library routines on the GPU. The results may then be accumulated in DP on the CPU to yield the final approximation for $\mathbf{AB}$.

For eqn. (30) to be exact, in addition to the issues for scalar multiplication, we have the additional round-off errors arising from the multiply-add operations. Specifically, if $\mathbf{A}$ is a $M \times K$ matrix and $\mathbf{B}$ is a $K \times N$ matrix, $\mathbf{AB}$ effectively consists of MN dot products of length K. As K increases, or as the range of magnitudes of X_{ij} becomes wider, more round-off error will occur due to the accumulation in SP. We explore these issues more in the benchmarks below.

C. Heterogeneous `MGEMM` algorithm

A different way to improve the precision is to consider the magnitudes of the matrix elements from the outset. That is, we decompose the matrix multiplication, $\mathbf{C} = \mathbf{AB}$, by splitting $\mathbf{A}$ and $\mathbf{B}$ into 'large' and 'small' components, giving

$$\begin{aligned} \mathbf{C} &= \left(\mathbf{A}^{\mathrm{large}} + \mathbf{A}^{\mathrm{small}}\right)\left(\mathbf{B}^{\mathrm{large}} + \mathbf{B}^{\mathrm{small}}\right) \\ &= \mathbf{AB}^{\mathrm{large}} + \mathbf{A}^{\mathrm{large}}\mathbf{B}^{\mathrm{small}} + \mathbf{A}^{\mathrm{small}}\mathbf{B}^{\mathrm{small}} \end{aligned} \quad (32)$$

where we have taken the simple approach of introducing a cutoff value, δ, to define the split. i.e. if $|X_{ij}| > \delta$, the element is considered 'large,' otherwise it is considered 'small.'

The $\mathbf{A}^{\mathrm{small}}\mathbf{B}^{\mathrm{small}}$ term consists entirely of 'small' numbers, and can be run with reasonable accuracy in single precision on the GPU (using the cleaving approach described above, if needed). The other two terms contain 'large' numbers, and need to be run in double precision to achieve greater accuracy. However, since each of the 'large' matrices will often be sparse, these terms each consist of a dense-sparse multiplication.

We only store the nonzero terms of the $\mathbf{A}^{\mathrm{large}}$ and $\mathbf{B}^{\mathrm{large}}$ matrices, cutting the computational complexity significantly. Consider

$$C'_{ik} = A_{ij}B^{\mathrm{large}}_{jk} \quad (33)$$

Only a few B^{large}_{jk} will be nonzero, and we consider each in turn. For a particular scalar B^{large}_{jk}, only the kth column

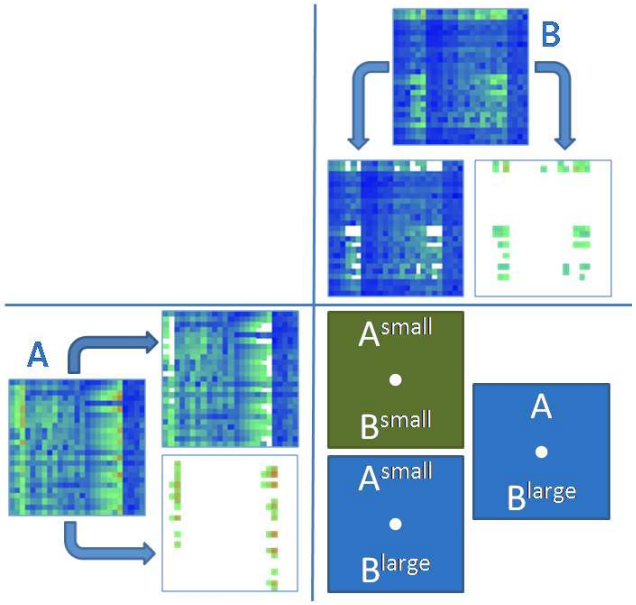

FIG. 1: Pictorial representation of the heterogeneous MGEMM algorithm. The bottom-left and top-right panels show the **A** and **B** matrices and their separation into matrices with large and small elements, given a cutoff parameter δ. The right-bottom panel shows the components of the final **AB** matrix, where the green component involves a dense matrix multiplication computed with cublas-SGEMM, while the blue components involve sparse matrix multiplications computed with a daxpy-like algorithm (explained in the text). The blocks follow the nomenclature shown on eqn. 32.

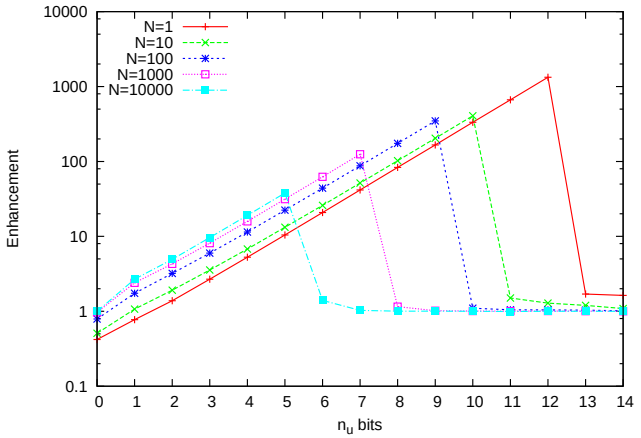

FIG. 2: MGEMM enhancement versus the number of upper bits, n_u, (and $n_l = 23$) for $N \times N$ matrices where $N \in \{1, 10, 100, 10^3, 10^4\}$. The matrices were initialized with uniform random numbers on the range $[1, 2]$.

of $\mathbf{C}'$ will be nonzero and is equal to the product of $B^{\rm large}_{jk}$ and the jth column vector of **A**. This nonzero column vector C'_{ik} can be added to the final result, **C**, and the next $B^{\rm large}_{jk}$ considered. A similar process can be applied to the $\mathbf{A}^{\rm large}\mathbf{B}^{\rm small}$ term (producing *row* vectors of **C**). Again, this approach can be generalized to a full *GEMM implementation including transposes. Figure 1 shows a cartoon of the heterogeneous MGEMM algorithm described above.

IV. MGEMM BENCHMARKS

We now explore the accuracy and efficiency of the two MGEMM algorithms for various matrix structures. Clearly, the aim is to achieve greater accuracy than a simple SGEMM call on the GPU, but with minimal extra computational cost. Throughout this section, calculations were made using an Intel Xeon E5472 (Harpertown) CPU clocked at 3.0 GHz attached to an NVIDIA Tesla C1060.

A. Bitwise MGEMM benchmarks

First we examine the MGEMM algorithm described in Sec. III B. In the following, we chose to only benchmark the implementation using 3 multiplications, as in eqn. (29). Numerous test calculations showed that using 4 explicit multiplications gave no significant improvement in accuracy over the scheme with 3 multiplications. Since the latter is faster, it should obviously be favoured.

To quantify the accuracy, we found it useful to introduce a new metric which we call the *enhancement*, χ. Using DGEMM as the reference, if s and m are the root mean square (RMS) errors in a matrix element for SGEMM and MGEMM, respectively, then we define $\chi = s/m$. Note that when multiplying two $N \times N$ matrices, the RMS errors are evaluated from N^2 data points. For the small matrices, therefore, the number of data points was insufficient to obtain reliable statistics. To remedy this, we repeated the smaller matrix multiplications (with different random numbers) to get more data points. The number of repeats was chosen to ensure that the standard error of the mean was at least two orders of magnitude smaller than the RMS error.

Figure 2 shows the enhancement for the multiplication of random $N \times N$ matrices with varying number of upper bits, n_u, (and $n_l = 23$). Five curves are shown for $N \in \{1, 10, 100, 10^3, 10^4\}$. The matrices were initialized with numbers uniform in the range $[1, 2]$.

The enhancement is dominated by round-off errors in the $\mathbf{A}^{\rm u}\mathbf{B}^{\rm u}$ terms. On average, these errors are of the same order of magnitude as the SGEMM errors, thus MGEMM will only show an improvement when these errors vanish. We can achieve this by first remembering that a single-precision float can only store 24 significant bits in the mantissa. Therefore, if we only multiply floats with less than 12 significant bits, there will be no round-off error associated with the operation.

For $N = 1$, the effect is clear in Figure 2: the enhancement suddenly decreases for $n_u > 12$. For $N > 1$, the number of bits, n_u, must be sufficiently small to also prevent round-off errors when accumulating results from

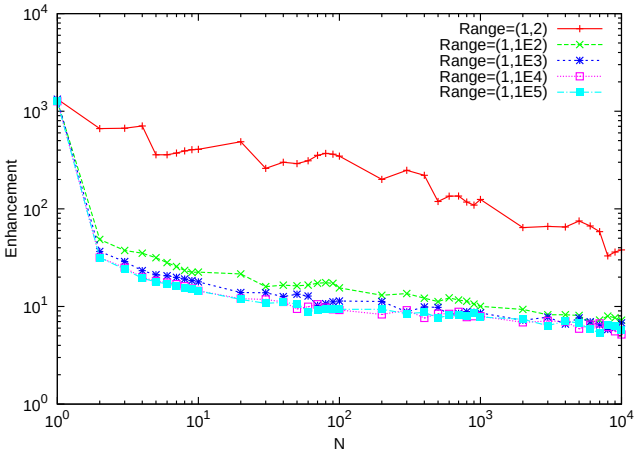

FIG. 3: `MGEMM` peak enhancement versus matrix size using uniform random numbers on 5 ranges: $[1,2]$, $[1,100]$, $[1,10^3]$, $[1,10^4]$, and $[1,10^5]$. Optimal values of n_u upper bits (and $n_l = 23$) were used for all data points.

many multiply-add operations. As N becomes larger, this effect is exacerbated, so the optimal n_u decreases.

Decreasing n_u, however, introduces larger errors into the other terms, such as $\mathbf{A}^{\mathrm{u}}\mathbf{B}^{\mathrm{l}}$. These errors are smaller than the $\mathbf{A}^{\mathrm{u}}\mathbf{B}^{\mathrm{u}}$ errors, but increase exponentially as n_u decreases. Decreasing n_u is therefore favourable until the $\mathbf{A}^{\mathrm{u}}\mathbf{B}^{\mathrm{u}}$ errors vanish, but there is no advantage to decreasing n_u further. In general, the combination of these effects means that the peak enhancement decreases with N.

Figure 3 explores the peak enhancement in more detail. Each curve uses the optimal n_u values and initializes $\mathbf{A}$ and $\mathbf{B}$ with uniform random numbers on one of 5 intervals: $[1,W]$ where $W \in \{2, 100, 10^3, 10^4, 10^5\}$.

For $W = 2$, the enhancement decreases from $O(10^3)$ to $O(10^1)$ as N increases, for reasons discussed above. For errors corresponding to a classical random walk, we'd expect χ to decrease as $\sqrt{N}$, which implies a gradient of $1/2$ in the log-log plot; this is approximately true for $W = 2$. For $W > 2$, however, the gradient in the log-log plot is reduced, and the enhancements are almost 2 orders of magnitude smaller for all $N > 1$. In summary, bitwise `MGEMM` is clearly most effective in a few special cases, such as for very small matrices, or when all the matrix elements are the same order of magnitude.

Table I highlights the issues regarding speedup when running bitwise `MGEMM` on the GPU compared to `DGEMM` on the CPU for different N. In addition, we catalogue the optimal values of n_u, as a function of N and W (the range of random numbers $[1,W]$.) As expected, the speedup increases as N increases, but unfortunately, bitwise `MGEMM` is only faster for the very largest matrices. The overheads of using the GPU, such as data transfer costs and memory access latencies, are well known; for $N < 1000$ the acceleration can be marginal (see also Figure 4). In addition, in our simple implementation of eqn. (29), there is also a significant overhead from splitting and processing the $\mathbf{A}$ and $\mathbf{B}$ matrices. Eqn. (30) implies three matrix multiplications, which are independent of each other. Therefore, it has not escaped our attention that this scheme is parallelizable. In summary, we see up to 3 times speedup over CPU `DGEMM` for large N, but a slowdown for $N \leq 1000$.

N	Speedup	Optimal n_u $W=2$	$W=100$	$W=10^3$	$W=10^4$	$W=10^5$
1	0.01	12	12	12	12	12
10	0.01	10	6	6	6	6
100	0.07	9	5	4	4	4
1000	0.52	7	4	4	4	4
10000	2.87	5	3	3	3	3

TABLE I: Speedups relative to CPU `DGEMM` and optimal number of upper bits, n_u, when using bitwise `MGEMM` for various matrix sizes with random elements on five intervals $[1,W]$.

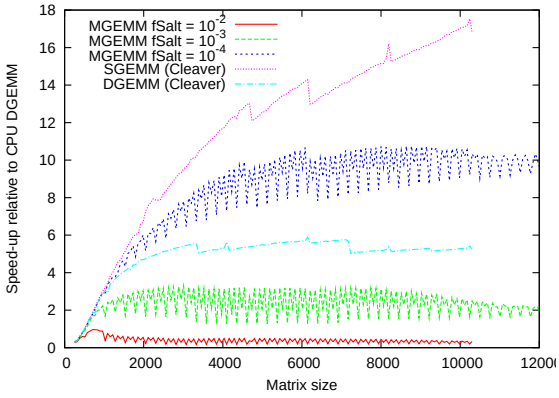

FIG. 4: Speedup for various `*GEMM` calls as a function of matrix size. Most elements were in the range $[-1,1]$, with the 'salt' values in the range $[90,110]$. Times are scaled relative to running `DGEMM` on the CPU.

B. Heterogeneous `MGEMM` benchmarks

We now benchmark the second `MGEMM` algorithm, described in Sec. III C. Figure 4 shows the speedup for a variety of `*GEMM` calls on the GPU with respect to the size, N, of a $N \times N$ matrix, relative to the time taken for the corresponding `DGEMM` call on the CPU.

The input matrices were initialized with uniform random values in the range $[-1,1]$ and then 'salted' with a fraction f_{salt} of random larger values in the range $[90,110]$. Both `SGEMM` and `DGEMM` were timed using the GPU (the latter being possible for modern cards, and included for reference). Three `MGEMM` runs were tested, for $f_{\mathrm{salt}} = 10^{-2}$, 10^{-3} and 10^{-4}. The size of the cutoff parameter δ was chosen such that all the salted elements were considered 'large'. All timings were averaged over ten runs.

Running CUBLAS `SGEMM` is approximately 17.1 times

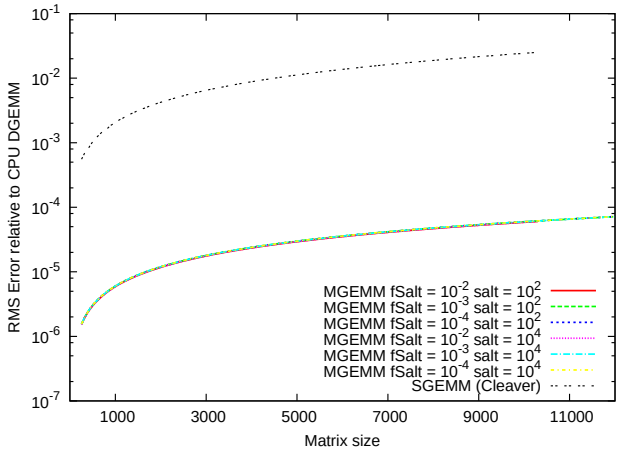

FIG. 5: RMS error in a single element for various GEMM calls as a function of matrix size compared to CPU DGEMM. Background elements were in the range $[-1, 1]$, with the 'salt' values in the range $[90, 110]$ or $[9990, 10010]$.

faster than running DGEMM on the CPU for a matrix of size 10048×10048, and is even faster for larger matrices. This represents an upper bound for the speedups we can hope to obtain with MGEMM for such matrices. Leveraging the GPU for small matrices is not effective, due to well-known overheads such as memory transfer and access latencies.

In contrast, the MGEMM speedups are strongly dependent on the fraction f_{salt}, which determines how much of the calculation is done in double-precision on the CPU. For $f_{\text{salt}} = 10^{-4}$, the speedups are approximately 10x, but for $f_{\text{salt}} = 10^{-3}$, speedups of approximately 2x relative to CPU DGEMM are observed. Indeed, for $f_{\text{salt}} = 10^{-2}$, MGEMM is actually slower than CPU DGEMM.

Next we consider the accuracy enhancement when using MGEMM compared to SGEMM. In Figure 5, we show the root mean square (RMS) errors of each matrix element relative to CPU DGEMM for different matrix sizes. As above, all the matrices were initialized with uniform random values in the range $[-1, 1]$, but now the salting sizes were grouped into two ranges: $[90, 110]$ and $[9990, 10010]$. Results are shown for SGEMM and MGEMM for various salting fractions.

As expected, SGEMM produces substantial errors. With a fraction of salted values, $f_{\text{salt}} = 1\%$, in the range $[90, 110]$, the errors are of $O(0.01)$ for the medium-sized matrices. In contrast, the errors are more than 2 orders of magnitude smaller when using MGEMM, and are the same regardless of the fraction or size of the salted elements. The limiting MGEMM errors are the same as the SGEMM errors for a pair of *unsalted* random matrices on $[-1, 1]$ because the MGEMM algorithm guarantees that all the salted contributions are computed on the CPU. Indeed, if the salts were larger or more numerous, the SGEMM errors would be even larger, but the MGEMM errors would be unchanged. This is in stark contrast to the behaviour of the bitwise MGEMM algorithm.

C. Comparison of MGEMM schemes

The behaviour of the two MGEMM algorithms is very different. First, the speed of the bitwise MGEMM depends only on the size of N, not on the elements of $\mathbf{A}$ or $\mathbf{B}$. Moreover, it is only 3 times faster than CPU DGEMM in the best case. In contrast, the speed of the heterogeneous MGEMM algorithm strongly depends on the fraction of 'large' elements. While it is up to 10 times faster than CPU DGEMM for a 0.01% fraction of large elements, for a 1.0% fraction, it is actually slower.

Concerning the accuracy of the two algorithms, consider the following cases. First, suppose the elements of $\mathbf{A}$ and $\mathbf{B}$ are random numbers on the interval $[1, 2]$. Mean errors produced by bitwise MGEMM are approximately 2 to 3 orders of magnitude smaller than SGEMM, depending on the matrix size (c.f. Figure 2). In contrast, it is obvious that no value of δ can be chosen to make the heterogeneous MGEMM useful. At the other extreme, consider two matrices with a vast majority of elements in the range $[0, 1]$, and a small minority of scattered elements of unlimited size. Now, the heterogeneous MGEMM will be extremely useful, while the bitwise algorithm is likely to be quite ineffective due to the problems previously explained.

For this research, we are interested in accelerating quantum chemistry. It turns out that the relevant $\mathbf{A}$ and $\mathbf{B}$ for the RI-MP2 method have large N and very large W, suggesting that the heterogeneous algorithm should be the method of choice.

V. RI-MP2 ACCELERATION BENCHMARKS

We now show how the tools described above can accelerate full RI-MP2 quantum chemistry calculations on real molecules. To achieve this, we accelerate the evaluation of eqn. (20) and eqn. (21) using MGEMM running on a GPU. We compare to the use of standard DGEMM BLAS on the CPU, and CUBLAS SGEMM on the GPU. For all these benchmarks, we used an AMD Athlon 5600+ CPU clocked at 2.8 GHz, combined with an NVIDIA Tesla C1060 GPU with 4 GiB of RAM. The matrix cleaver and MGEMM were implemented in a modified version of the Q-Chem 3.1 RI-MP2 code described elsewhere [3].

As anticipated in the previous section, the bitwise MGEMM library does not give useful improvements compared to a standard CPU DGEMM. In the evaluation of eqn. (20), we observed no significant improvement in accuracy using bitwise MGEMM, and an enhancement of only 2.5 for eqn. (21). Thus we decided not to study the use of bitwise MGEMM further here. The results of applying the heterogeneous MGEMM algorithm (just MGEMM, from now on) were much more encouraging.

For our test systems we chose a set of linear alkanes (C_8H_{18}, $C_{16}H_{34}$, $C_{24}H_{50}$, $C_{32}H_{66}$, $C_{40}H_{82}$) as well as two molecules of pharmaceutical interest, the anti-cancer drugs taxol ($C_{47}H_{51}NO_{14}$) and valinomycin

($C_{54}H_{90}N_6O_{18}$). We used the cc-pVDZ and cc-pVTZ [10] atomic orbital basis sets throughout.

	Speedup		SGEMM energy error
Molecule	SGEMM	DGEMM	(kcal mol^{-1})
C_8H_{18}	2.1	1.9	-0.05616
$C_{16}H_{34}$	4.5	3.7	-0.12113
$C_{24}H_{50}$	6.9	5.2	-0.62661
$C_{32}H_{66}$	9.0	6.4	-0.75981
$C_{40}H_{82}$	11.1	7.2	-1.12150
Taxol	11.3	7.1	-6.26276
Valinomycin	13.8	7.8	-9.99340

TABLE II: Speedups using CUBLAS SGEMM and DGEMM and total energy errors relative to CPU DGEMM for various molecules in a cc-pVDZ basis.

First, in Table II we benchmark the reference case of using either CUBLAS SGEMM or DGEMM for each test molecule using the double-ζ basis set. The table shows the speedup in computing the RI-MP2 correlation energy and the error relative to a standard CPU calculation (the DGEMM errors are negligible). The speedups and SGEMM errors are greater for the larger molecules, with the largest speedups observed for valinomycin: 13.8x and 7.8x, using SGEMM and DGEMM, respectively. However, while CUBLAS DGEMM gives essentially no loss of accuracy, the SGEMM error is approximately -10.0 kcal mol^{-1}, which is well beyond what is generally accepted as chemical accuracy.

Quantum chemistry generally aims to achieve a target accuracy of 1.0 kcal mol^{-1}. In Table III, we explore the performance of MGEMM using a constant cutoff value of δ=1.0 to try and reduce the SGEMM errors in Table II. The results show speedups and total energy errors for each molecule in both the double-ζ and triple-ζ basis sets. In this particular case, we have limited the GPU to use only 256 MiB of RAM to mimic the capability of older cards and emphasize the use of the MGEMM cleaver. This will naturally result in a loss of speedup compared to utilizing a larger GPU memory. In the case of taxol the reduction is approximately 20%.

	Speedup		Energy error (kcal mol^{-1})	
Molecule	Double-ζ	Triple-ζ	Double-ζ	Triple-ζ
C_8H_{18}	1.9	2.7	-0.01249	-0.03488
$C_{16}H_{34}$	3.8	5.6	-0.00704	-0.04209
$C_{24}H_{50}$	5.8	8.2	-0.14011	-0.33553
$C_{32}H_{66}$	7.9	9.2	-0.08111	-0.29447
$C_{40}H_{82}$	9.4	10.0	-0.13713	-0.51186
Taxol	9.3	10.0	-0.50110	-1.80076
Valinomycin	10.1	-	-1.16363	-

TABLE III: MGEMM speedups and total energy errors with respect to CPU DGEMM for various molecules in a cc-pVDZ and cc-pVTZ basis.

Looking at Table III, the trends are the same as in Table II, but the MGEMM errors are approximately an order of magnitude less than the SGEMM errors (for the larger molecules). For valinomycin in the cc-pVDZ basis, the SGEMM speedup is reduced from 13.8x to 10.1x using MGEMM, but the error in the total energy is also reduced from -10.0 kcal mol^{-1} to -1.2 kcal mol^{-1}, which is now very close to chemical accuracy. Moreover, while CUBLAS DGEMM clearly has the advantage (when available) of high accuracy, if -1.2 kcal mol^{-1} is deemed an acceptable accuracy, MGEMM could be favoured since the DGEMM speedup is only 7.8x compared to 10.1x. Note that the errors are larger for the triple-ζ basis simply because it requires more computation, which leads to greater error accumulation; this is a general issue not only for higher quality basis sets, but also for larger molecules using lower quality basis sets.

VI. CONCLUSION

We have demonstrated how computational quantum chemistry can benefit from leveraging the power of graphical processing units in a very simple way. Our new tools are easy to incorporate into existing legacy codes where matrix multiplications involve a substantial fraction of the overall computational cost. Clearly, more efficient use of the GPU can be achieved in special cases by devoting time to rewriting and redesigning the algorithms for GPU architectures. However, this is often not a feasible option in practice, especially for a mature discipline such as quantum chemistry where we typically employ large program suites representing years or decades of coding effort.

In summary, our contribution has been the development, testing and benchmarking of a general black-box approach for the efficient GPU acceleration of matrix-matrix multiplications. In particular, our new library, which we call MGEMM [11], works for matrices of arbitrary size, even if too large for the whole computation to be held in the GPU's onboard memory. Moreover, while assuming only single-precision operations to be available on the GPU, we have demonstrated two algorithms whereby orders of magnitude greater accuracy can be achieved within the context of a mixed-precision approach.

To illustrate the utility of MGEMM for quantum chemistry, we combined it with the Q-Chem 3.1 [9] program package and computed the MP2 correlation energy of the 168-atom valinomycin molecule in a cc-pVDZ basis set. Using SGEMM, MGEMM and (GPU) DGEMM, we observed speedups of 13.8x, 10.1x and 7.8x, respectively, while MGEMM was an order of magnitude more accurate than SGEMM, thus achieving 'chemical accuracy'.

Traditionally, GPUs have only had single-precision (SP) support and it remains true that the vast majority of GPU cards worldwide currently do not have double precision (DP) capability. Indeed, we are interested in using commodity GPUs within a grid-computing environment,

such as that promoted by the Berkeley Open Infrastructure for Network Computing (BOINC) [12]. GPUs in a typical BOINC client do not have DP support, yet comprise a formidable resource worldwide.

Nevertheless, DP devices and co-processors are now available. Moreover, the trend seems to be towards the release of more advanced DP support in the future, such as the next-generation GPU from NVIDIA, currently code-named Fermi. Fermi will reportedly have a DP peak performance which is only a factor of 2 less than the SP performance. (For NVIDIA's C1060, the ratio is approximately 12).

It is therefore valid to question the potential of mixed-precision algorithms for next-generation GPU architectures. We believe this is an open issue. For example, practical calculations on GPUs are very often bound by memory bandwidth, rather than raw operation count. While DP cards are getting faster, this I/O bottleneck is likely to remain. In these cases, the transfer and processing of only SP data could effectively double the performance compared to naive DP calculations. Overall, we believe that mixed-precision algorithms could remain important for applications were the highest performance is a priority.

VII. ACKNOWLEDGMENTS

The authors would like to thank L. Vogt for helpful discussions and her contributions to the numerical benchmarks. Financial support was provided by the NSF "Cyber-Enabled Discoveries and Innovations" (CDI) Initiative Award PHY-0835713, as well as NVIDIA. R.O.A. wishes to thank CONACyT and Fundación Harvard en México for additional financial support. Technical support by the High Performance Technical Computing Center at the Faculty of Arts and Sciences of Harvard University and the National Nanotechnology Infrastructure Network Computation project was appreciated.

[1] K. Yasuda, J. Comput. Chem. **29**, 334 (2008), URL `http://dx.doi.org.ezp-prod1.hul.harvard.edu/10.1002/jcc.20779`.

[2] I. S. Ufimtsev and T. J. Martinez, J. Chem. Theory Comput. **4**, 222 (2008), ISSN 1549-9618, URL `http://pubs3.acs.org/acs/journals/doilookup?in\_doi=10.1021/ct700268q`.

[3] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, and A. Aspuru-Guzik, J. Phys. Chem. A **112**, 20497 (2008), ISSN 1089-5639, URL `http://pubs3.acs.org/acs/journals/doilookup?in\_doi=10.1021/jp0776762`.

[4] R. Olivares-Amaya, M. A. Watson, R. G. Edgar, L. Vogt, Y. Shao, and A. Aspuru-Guzik, Journal of Chemical Theory and Computation p. 091214115953059 (2009), ISSN 1549-9618, URL `http://pubs.acs.org/doi/abs/10.1021/ct900543q`.

[5] T. Helgaker, P. Jørgensen, and J. Olsen, *Molecular Electronic-Structure Theory* (Wiley, Chichester, 2000).

[6] M. Feyereisen, G. Fitzgerald, and A. Komornicki, Chem. Phys. Lett. **208**, 359 (1993).

[7] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, et al., ACM Transactions on Mathematical Software **28**, 152 (2002).

[8] Y. Hida, X. S. Li, and D. H. Bailey, in *ARITH '01: Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (IEEE Computer Society, Washington, DC, USA, 2001), p. 155.

[9] Y. Shao, L. Fusti-Molnar, Y. Jung, J. Kussmann, C. Ochsenfeld, S. T. Brown, A. T. B. Gilbert, L. V. Slipchenko, S. V. Levchenko, D. P. O'Neill, et al., Phys. Chem. Chem. Phys. **8**, 3172 (2006).

[10] T. Dunning Jr., J. Chem. Phys. **90**, 1007 (1989).

[11] *scigpu-gemm v0.8, http://scigpu.org.*

[12] J. Bohannon, Science **308**, 310 (2005).