High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs

John E. Stone
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801
johns@ks.uiuc.edu

Jan Saam
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801
saam@ks.uiuc.edu

David J. Hardy
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801
dhardy@ks.uiuc.edu

Kirby L. Vandivort
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801
kvandivo@ks.uiuc.edu

Wen-mei W. Hwu
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801
w-hwu@uiuc.edu

Klaus Schulten*
Department of Physics
University of Illinois at Urbana-Champaign
Urbana, IL 61801
kschulte@ks.uiuc.edu

ABSTRACT

The visualization of molecular orbitals (MOs) is important for analyzing the results of quantum chemistry simulations. The functions describing the MOs are computed on a three-dimensional lattice, and the resulting data can then be used for plotting isocontours or isosurfaces for visualization as well as for other types of analyses. Existing software packages that render MOs perform calculations on the CPU and require runtimes of tens to hundreds of seconds depending on the complexity of the molecular system.

We present novel data-parallel algorithms for computing lattices of MOs on modern graphics processing units (GPUs) and multi-core CPUs. The fastest GPU algorithm achieves up to a 125-fold speedup over an optimized CPU implementation running on one CPU core. We also demonstrate possible benefits of dynamic GPU kernel generation and just-in-time compilation for MO calculation. We have implemented these algorithms within the popular molecular visualization program VMD, which can now produce high quality MO renderings for large systems in less than a second, and achieves the first-ever interactive animations of quantum chemistry simulation trajectories using only on-the-fly calculation.

*Corresponding author.

Categories and Subject Descriptors

J.2 [**Computer Applications**]: Physical Sciences and Engineering—*Chemistry*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Algorithms, Design, Performance

Keywords

GPU computing, GPGPU, CUDA, molecular orbital

1. INTRODUCTION

A molecular orbital (MO) represents a stationary state in which an electron can be found in a molecule, where the MO's spatial distribution is correlated to the probability density for the electron. Visualization of MOs is an important task for understanding the chemistry of molecular systems. MOs appeal to the chemist's intuition, and inspection of the MOs aids in explaining chemical reactivities. For instance, showing the orbital dynamics of a system along a reaction coordinate provides insight into the mechanism of a chemical reaction. Some existing software tools with these capabilities include MacMolPlt [2], Molden [17], and Molekel [15].

The calculations required for visualizing MOs are computationally demanding, and existing quantum chemistry visualization programs are only fast enough to interactively compute MOs for small molecules on a relatively coarse lattice. Existing visualization tools perform molecular orbital calculations on the CPU and require runtimes of tens to hundreds of seconds, depending on the complexity of the molecular system and spatial resolution of the MO discretization and subsequent surface plots. Few of these packages are parallelized for multi-core CPUs, and none for graphics processing units (GPUs), so a great opportunity exists to improve upon their capabilities in terms of interactivity, visual display quality, and scalability to larger and more complex molecular systems.

The advent of classical molecular dynamics (MD) simulations marked a fundamental change in the understanding of the function of complex biomolecular systems, providing the ability to view atomic motions hitherto unavailable to experimental techniques. A key to this understanding has been the ability to interactively animate MD trajectories at rates of 10 frames per second or faster, providing a lively view of their dynamical behavior. After decades of classical simulations and movies of classical motions, we are now entering the era of visualizing quantum mechanical dynamics behavior. With the new capability to interactively animate MOs, researchers will have a much better intuition for what the electrons are "doing" during a reaction. In combined quantum mechanics/molecular mechanics (QM/MM) simulations, the orbital dynamics can now be computed and displayed at the same rate as the motion of the atomic nuclei.

The recent shift of computer architecture away from high speed serial processors toward multi-core CPUs and massively parallel devices such as GPUs has led researchers to begin redesigning key computational kernels to exploit the performance of modern hardware. GPUs have emerged as a revolutionary technological opportunity due to their tremendous floating point capability, low cost, and ubiquitous presence in commodity computer systems. In particular, the fields of computational chemistry and biology have enjoyed early successes in exploiting GPUs to accelerate computationally demanding tasks [20, 14, 1, 21]. The CUDA GPU programming toolkit enabled many of these prior efforts and is the basis of the work described in this paper. Nickolls et al. provide a detailed discussion of the CUDA programming model [13]. In this paper we present high performance data-parallel algorithms for computing MOs on multi-core CPUs and GPUs and details of their implementation within the molecular visualization program VMD [9]. We evaluate our algorithms on several representative test cases, compare their performance with two popular quantum chemistry visualization tools, and describe opportunities for further improvement.

2. BACKGROUND

We can offer here only a brief introduction to MOs and basis sets; more details can be found in computational chemistry textbooks and reviews [4, 5]. MOs are solutions of the Schrödinger equation. Numerous quantum chemistry packages exist (e.g., GAMESS [18], Gaussian [7] or NWChem [3]) that solve the electronic Schrödinger equation $H\Psi = E\Psi$ for a given system; this text discusses the visualization of these solutions. Examples are shown in Fig. 1. MOs are the eigenfunctions Ψ_ν of the molecular wavefunction Ψ, with H the Hamiltonian operator and E the system energy. The wavefunction determines molecular properties. For instance, the one-electron density is $\rho(\mathbf{r}) = |\Psi(\mathbf{r})|^2$. The scheme to evaluate the wavefunction presented here can be used with only minor modifications to compute other molecular properties like the charge density, the molecular electrostatic potential, or multipole moments.

Each MO Ψ_ν can be expressed as a linear combination

shell	l	m	basis function $\Phi_{l,m}$
s	0	0	$\Phi_{0,0} = N_{0,0}\exp(-\zeta R^2)$
p$_x$	1	0	$\Phi_{1,0} = N_{1,0}\exp(-\zeta R^2)\,x$
p$_y$		1	$\Phi_{1,1} = N_{1,1}\exp(-\zeta R^2)\,y$
p$_z$		2	$\Phi_{1,2} = N_{1,2}\exp(-\zeta R^2)\,z$
d$_{xx}$	2	0	$\Phi_{2,0} = N_{2,0}\exp(-\zeta R^2)\,x^2$
d$_{xy}$		1	$\Phi_{2,1} = N_{2,1}\exp(-\zeta R^2)\,xy$
d$_{xz}$		2	$\Phi_{2,2} = N_{2,2}\exp(-\zeta R^2)\,xz$
d$_{yy}$		3	$\Phi_{2,3} = N_{2,3}\exp(-\zeta R^2)\,y^2$
d$_{yz}$		4	$\Phi_{2,4} = N_{2,4}\exp(-\zeta R^2)\,yz$
d$_{zz}$		5	$\Phi_{2,5} = N_{2,5}\exp(-\zeta R^2)\,z^2$

Table 1: Basis functions for s, p and d shells. Each shell has a certain number of components (such as p$_x$, p$_y$, p$_z$) associated with functions representing different angular momenta.

over a set of K basis functions Φ_κ,

$$\Psi_\nu = \sum_{\kappa=1}^{K} c_{\nu\kappa}\Phi_\kappa, \tag{1}$$

where $c_{\nu\kappa}$ are coefficients provided in the output of the QM package and used as input to our calculation. The basis functions used by the vast majority of quantum chemical calculations are atom-centered functions that approximate the solution of the Schrödinger equation for a single hydrogen atom with one electron, hence they are called "atomic orbitals." Fig. 2 depicts the shapes of a few common atomic orbitals.

Because they are not suitable for fast computation, functions representing the *exact* solutions of the Schrödinger equation for the hydrogen atom are not used as basis functions in modern quantum chemistry packages. Instead, they are modeled with more computationally efficient, so-called Gaussian type orbitals (GTOs):

$$\Phi^{\mathrm{GTO}}_{i,j,k}(\mathbf{R},\,\zeta) = N_{\zeta ijk}\,x^i\,y^j\,z^k\exp(-\zeta R^2). \tag{2}$$

Here, the exponential factor ζ is a part of the basis set definition, and i, j, and k, are used to modulate the functional shape. $N_{\zeta ijk}$ is a normalization factor deduced from the basis set definition, as presented in the next section. The vector $\mathbf{R} = \{x, y, z\}$ of length $R = |\mathbf{R}|$ connects the nucleus on which this basis function is centered to the current point in space.

The exponential term in Eq. (2) produces the radial decay of the function. To accurately describe the radial behavior of the atomic orbital (e.g., higher shells have radial nodes), linear combinations of GTOs are used as basis functions. These composite basis functions are called contracted GTOs (CGTOs), and the P individual GTOs are referred to as primitives,

$$\Phi^{\mathrm{CGTO}}_{i,j,k}(\mathbf{R},\,\{c_p\},\{\zeta_p\}) = \sum_{p=1}^{P} c_p\Phi^{\mathrm{GTO}}_{i,j,k}(\mathbf{R},\,\zeta_p). \tag{3}$$

The set of contraction coefficients $\{c_p\}$ and the set of exponents $\{\zeta_p\}$ defining the CGTO are output from the QM package.

CGTOs are classified into different *shells* based on the sum $l = i + j + k$ of the exponents of the x, y, and z fac-

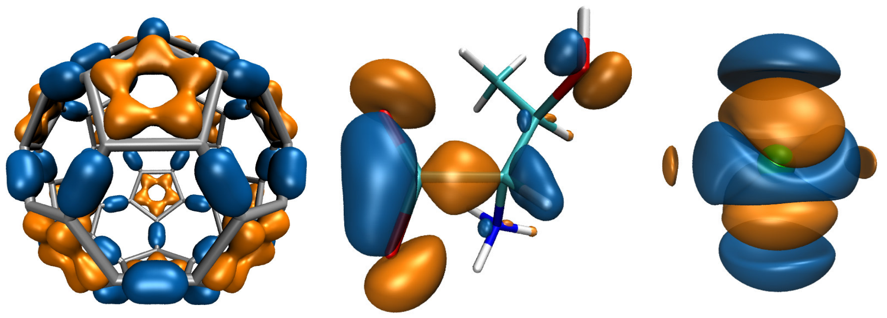

Figure 1: Examples of MO isovalue surfaces resulting from the lattice of wavefunction amplitudes computed for each molecule: Carbon-60, the amino acid threonine, and the element krypton. Isosurfaces of positive values are shown in blue, and negative valued isosurfaces are shown in orange.

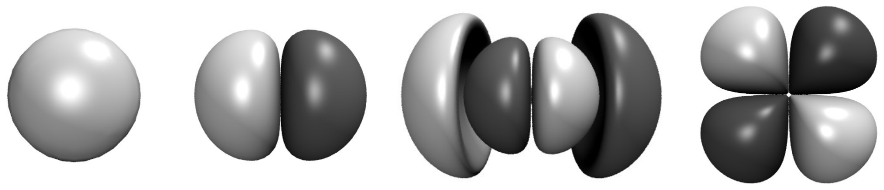

Figure 2: Shape of 1s, 2p$_x$, 3p$_x$ and 3d$_{xy}$ atomic orbitals.

tors. The shells are designated by letters s, p, d, f, and g for $l = 0, 1, 2, 3, 4$, respectively, where we explicitly list here the most common shell types but note that higher-numbered shells are occasionally used. The set of indices for a shell are also referred to as the different *angular momenta* of that shell. We establish an alternative indexing of the angular momenta based on the shell number l and a systematic indexing m over the possible number of sums $l = i + j + k$, where $M_l = \binom{l+2}{l}$ counts the number of combinations and $m = 0, \dots, M_l - 1$ references the set $\{(i, j, k) : i + j + k = l\}$. For example, the angular momenta for the $l = 2$ shell can be listed by $(i, j, k) \mapsto$ $(2, 0, 0)$, $(1, 1, 0)$, $(1, 0, 1)$, $(0, 2, 0)$, $(0, 1, 1)$, $(0, 0, 2)$. The angular momenta of the s, p, and d shells are listed in Table 1.

The linear combination defining the MO Ψ_ν must also sum contributions by each of the N atoms of the molecule and the L_n shells of each atom n. The entire expression, now described in terms of the data output from a QM package, for an MO wavefunction evaluated at a point $\mathbf{r}$ in space then becomes

$$\begin{aligned}\Psi_\nu(\mathbf{r}) &= \sum_{\kappa=1}^{K} c_{\nu\kappa}\Phi_\kappa \\ &= \sum_{n=1}^{N}\sum_{l=0}^{L_n-1}\sum_{m=0}^{M_l-1} c_{\nu nlm}\Phi_{n,l,m}^{\mathrm{CGTO}}(\mathbf{R}_n, \{c\}, \{\zeta\}), \qquad (4)\end{aligned}$$

where we have replaced $c_{\nu\kappa}$ by $c_{\nu nlm}$, with the vectors $\mathbf{R}_n = \mathbf{r} - \mathbf{r}_n$ connecting the position $\mathbf{r}_n$ of the nucleus of atom n to the desired spatial coordinate $\mathbf{r}$. We have dropped the subscript p from the set of contraction factors $\{c\}$ and exponents $\{\zeta\}$ with the understanding that each CGTO requires an additional summation over the primitives, as expressed in Eq. (3).

3. ALGORITHMIC DETAILS

For the purpose of visualization, an MO is essentially the 3-D scalar field of the wavefunction amplitude. There are several ways to depict such a field. For a single planar slice through the scalar field, one can display a height field or contour lines on a plane. In three dimensions, MOs may be depicted through isosurfaces or direct volume rendering of the scalar field. Isosurfaces are the most popular visualization choice among quantum chemists, presumably due to the comparatively low computational cost. Common to all visualization techniques used is the computation of the wavefunction amplitude on a lattice, which is a discretization of Eq. (4). Since the wavefunction has diminished to zero beyond a few Angstroms from the molecule, the lattice size is determined by the boundaries of the molecule plus some margin. We have developed a series of multi-core CPU and GPU algorithms for computing MOs on regularly spaced lattices. The strategies for achieving high performance on CPUs and GPUs are quite different due to their unique architectural strengths, and we highlight the differences between our solutions.

3.1 Preparing and optimizing the computation

Two sets of data are needed for the evaluation of MOs: the basis set, comprised of the contraction coefficients and exponents for each primitive, and the wavefunction coefficients. The number of shells can vary between atoms and the number of primitives and angular momenta can be different for each shell. Hence, the basis set and wavefunction would be most conveniently stored in the form of hierarchical data structures. However, for increased locality of reference and peak traversal performance on the GPU (due to global memory coalescing requirements), we craft packed coefficient arrays and then compose indexing arrays with the number of shells per atom and with the number of primitives per shell.

The output of a QM package typically lists the basis set for each atom independently, even though the basis functions for atoms of the same chemical element are identical. Our implementation preprocesses the loaded data, resulting in a unique basis set with only one entry for each chemical element type. The elimination of redundant basis set data boosts CPU performance by increasing cache hit rates, but it particularly benefits the GPU by conserving scarce on-chip memory. By sorting atoms according to their chemical element type and processing them in sorted order, the basis set data can be reused multiple times without the need to re-fetch it from (slow) main memory. This provides a performance benefit on both the CPU and GPU in cases where molecules contain many atoms of the same chemical element type.

The normalization factor $N_{\zeta ijk}$ appearing in Eq. (2) can be factored into a first part $\eta_{\zeta l}$ that depends on both the exponent ζ and shell type $l = i+j+k$ and a second part η_{ijk} ($= \eta_{lm}$ in terms of our alternative indexing) that depends only on the angular momentum,

$$N_{\zeta ijk} = \left(\frac{2\zeta}{\pi}\right)^{\frac{3}{4}} \sqrt{(8\zeta)^l} \cdot \sqrt{\frac{i!\,j!\,k!}{(2i)!\,(2j)!\,(2k)!}} = \eta_{\zeta l} \cdot \eta_{ijk}. \quad (5)$$

The separation of the normalization factor allows us to factor the summation over the primitives from the summation over the wavefunction array. Combining Eqs. (2)–(4) and rearranging terms gives

$$\Psi_\nu(\mathbf{r}) = \sum_{n=1}^{N} \sum_{l=0}^{L_n-1} \left(\sum_{m=0}^{M_l-1} \underbrace{c_{\nu nlm}\eta_{lm}}_{c'_{\nu nlm}}\ \omega_{lm} \right) \times \left(\sum_{p=1}^{P_{nl}} \underbrace{c_p \eta_{\zeta l}}_{c'_p} \exp(-\zeta_p R_n^2) \right). \quad (6)$$

We define $\omega_{lm} = x^i\, y^j\, z^k$ using our alternative indexing over l and m explained in the previous section. We reduce both data storage and operation count by defining $c'_{\nu alm} = c_{\nu alm}\eta_{lm}$ and $c'_p = c_p \eta_{\zeta l}$. The number of primitives P_{nl} depends on both the atom n and the shell number l. Fig. 3 shows the organization of the basis set and wavefunction coefficient arrays listed for a small example molecule.

Pseudo-code for the evaluation of Eq. (6) is shown in Algorithm 1, providing the foundation for both the CPU and GPU kernels described below. For high performance, wavefunction coefficients for the angular momenta are sorted in a preprocessing step and stored in the order they will be referenced within the loop over shell types. Since basis sets may involve arbitrarily high shell types, a looping construct is required to evaluate the angular momenta. The algorithm shows how we eliminate a loop over the third index by exploiting $k = l - i - j$. Extra evaluations of the `pow()` library function to compute powers of x, y, and z are avoided by using successive multiplications that follow the increments to the i, j, and k indexes. One significant performance optimization not shown in the algorithm is the use of *switch* case statements to augment the loop body that processes the angular momenta for the most commonly occurring shell types. Each shell type up to g is evaluated with a hand-coded unrolling of the angular momenta loops, such as in Table 1. By unrolling these loops and precomputing the most frequently used powers of x, y, and z in the outermost loop over atoms, a 20% to 50% overall performance increase is achieved versus the loop-based implementation shown here.

Algorithm 1 Calculate MO value $\Psi_\nu(\mathbf{r})$ at lattice point $\mathbf{r}$ for given wavefunction and basis set coefficient arrays.

```
1: $\Psi_\nu \Leftarrow 0.0$
2: $ifunc \Leftarrow 0$ {index array of wavefunction coefficients}
3: $shell\_counter \Leftarrow 0$ {index array of shell numbers}
4: for $n = 1$ to $N$ do {loop over atoms}
5:   $(x, y, z) \Leftarrow \mathbf{r} - \mathbf{r}_n$ {$\mathbf{r}_n$ is position of atom $n$}
6:   $R^2 \Leftarrow x^2 + y^2 + z^2$
7:   $prim\_counter \Leftarrow atom\_basis[n]$ {index arrays of basis set data}
8:   for $l = 0$ to $num\_shells\_per\_atom[n] - 1$ do {loop over shells}
9:     $\Phi^{\mathrm{CGTO}} \Leftarrow 0.0$
10:    for $p = 0$ to $num\_prim\_per\_shell[shell\_counter] - 1$ do {loop over primitives}
11:      $c'_p \Leftarrow basis\_c[prim\_counter]$
12:      $\zeta_p \Leftarrow basis\_zeta[prim\_counter]$
13:      $\Phi^{\mathrm{CGTO}} \Leftarrow \Phi^{\mathrm{CGTO}} + c'_p * exp(-\zeta_p * R^2)$
14:      $prim\_counter \Leftarrow prim\_counter + 1$
15:    end for
16:    for all $i$ such that $0 \le i \le shell\_type[shell\_counter]$ do {loop over angular momenta}
17:      $jmax \Leftarrow shell\_type[shell\_counter] - i$
18:      for all $j$ such that $0 \le j \le jmax$ do
19:        $k \Leftarrow jmax - j$
20:        $c' \Leftarrow wavefunction[ifunc]$
21:        $\Psi_\nu \Leftarrow \Psi_\nu + c' * \Phi^{\mathrm{CGTO}} * x^i * y^j * z^k$
22:        $ifunc \Leftarrow ifunc + 1$
23:      end for
24:    end for
25:    $shell\_counter \Leftarrow shell\_counter + 1$
26:  end for
27: end for
28: return $\Psi_\nu$
```

Visualization of an MO is accomplished by repeated evaluations of Algorithm 1 to compute a three-dimensional map of Ψ_ν on a finely spaced lattice. Letting μ be the number of lattice points, we see that the overall computational complexity for calculating this map is $O\big(\mu N L (M+P)\big)$, increasing quadratically as the product of the number of lattice points and number of atoms N, with a significant operation constant bounded by $L(M+P)$, where $L = \max\{L_n\}$, $M = M_{(L-1)}$, and $P = \max\{P_{nl}\}$.

3.2 Fast approximation of Gaussians

Due to the computational expense of evaluating e^x by calling `expf(x)` on the CPU, we have investigated the possibility of accelerating the overall calculation by replacing `expf()` with a less costly approximation. The demands of visualizing MOs permit reduced precision if we maintain the general shape and smoothness of the Gaussians.

For the approximation of Gaussians, we take advantage of the domain restriction of e^x to $x \le 0$. Since floating point powers of 2 are cheap to calculate by directly manipulating the bits of the exponent, we convert the exponential employing $e^x = 2^{n-d}$, whence $n - d = x \log_2 e$, with n an integer and $0 \le d < 1$. We exactly calculate 2^n as the scaling factor for an interpolation of 2^{-d}. Sufficient accuracy

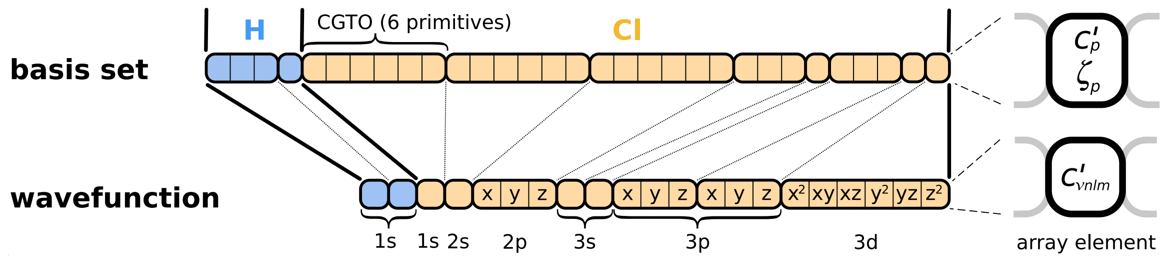

Figure 3: Structure of the basis set and the wavefunction coefficient arrays for HCl using the 6-31G* basis [6]. Each rounded box symbolizes the data pertaining to a shell. In the basis set array, an element in a box denotes both the exponent ζ_p and contraction coefficient c'_p for one primitive. In the wavefunction array the elements signify linear combination coefficients $c'_{\nu nlm}$ for the basis functions. Despite the differing angular momenta, all basis functions of a shell (marked by x, y, and z) use the same linear combination of primitives (see lines relating the two arrays). For example, the 2p shell in Cl is associated with 3 angular momenta that all share the exponents and contraction coefficients of the same 6 primitives. There can be more than one basis function for a given shell type (brackets below array).

is obtained by using a linear blending of third degree Taylor polynomials of 2^y expanded about $y = -1$ and $y = 0$, providing a continuously differentiable interpolation. The maximum relative error per interval is 0.131%, with a maximum absolute error of -0.000716. The entire calculation requires just six multiplies, six adds/subtracts, one bit shift, and one `floorf`() evaluation. We also accelerate the overall computation by using a cutoff value of -10, beyond which we truncate the approximation to 0 to avoid extra calculation. Manual testing indicates that our fast approximation works fine in practice for isosurface values larger than 10^{-4}, below which artifacts appear. The algorithm is amenable to Streaming SIMD Extensions (SSE) instructions that perform four evaluations simultaneously, and both standard and SSE kernels are tested in the following section and show marked speedup compared to the use of `expf`().

Beyond the approximation of e^x for calculating Gaussians, an even greater performance improvement may be available through interpolating the entire linear combination of GTO primitives for each contracted GTO, shown in Eq. (3). Unlike our approximation of e^x, this would require interpolation from a lookup table of precomputed values, but with the benefit of eliminating an inner loop of the calculation.

3.3 CPU algorithm details

It is a straightforward task to implement a sequential algorithm for computing MOs on the CPU. Modern CPU caches are large enough that, for many small molecules, the MO coefficient data will fit in the largest L3 or L2 cache. Since MO lattice values can be computed completely independently, the series of nested loops for processing the lattice points can be easily decomposed into separate planes, slabs, or blocks for parallel computation on multiple CPU cores. The preprocessing optimizations described above make the algorithm cache-friendly, so each CPU core spends most of its time working from local cache, and main memory bandwidth is not a significant bottleneck, even for a multi-core kernel.

With the previously described preprocessing and arithmetic optimizations in place, the dominant component of CPU runtime is the exponential computation for each basis set primitive. Most CPUs do not include dedicated instructions for computing exponentials, so they are provided by an external math library. The cost of calling an external C math library `expf()` function can be hundreds to over a thousand clock cycles, depending on the quality of the implementation and the processor hardware architecture. The use of an inlined function or a fast compiler intrinsic can often provide a substantial performance boost by eliminating function call overhead. For the CPU kernel, we use an inlined `expf()` function based on the Cephes math library by Steven Moshier [11]. Further performance gains are achieved by replacing the use of `expf()` with an approximation specifically tailored to the MO computation, as described above.

Further performance gains can be achieved by computing multiple lattice values at a time using SIMD instructions such as SSE. In theory, the use of 4-way SSE instructions would yield a 4x performance increase, but this comes at the cost of hand-coding the MO processing loop using SSE compiler intrinsics or direct assembly language, as existing compilers are incapable of automatically vectorizing the exponential function. The 4-way SSE kernels required careful design since the SSE instruction set does not allow divergent branching or scatter-gather memory operations. Due to SSE restrictions on memory alignment and partial loads and stores, the lattice is padded to an even multiple of four elements in the x-dimension. The innermost loop over lattice points in the x-dimension is then modified to process groups of four lattice points at a time.

3.4 GPU algorithm details

The authors have previously explored several GPU algorithms for computing Coulomb potentials on regularly spaced lattices, which is a problem with similarities to the MO computation presently of interest. In our previous work, we demonstrated GPU-accelerated algorithms capable of outperforming a single CPU core by factors of 26x to 44x for a single GPU and approximately linear scaling on multiple GPUs [20, 14, 16, 8]. One of the unique attributes of the MO algorithm, as compared with our past work with other spatially evaluated functions, is the comparatively large operand and operation count per lattice point and the increased complexity of control flow. Since the GPU provides dedicated exponential arithmetic instructions, the relative cost of evaluating e^x by calling `expf()` or `__expf()`, or evaluating 2^x via `exp2f()`, is much lower than on the CPU. According

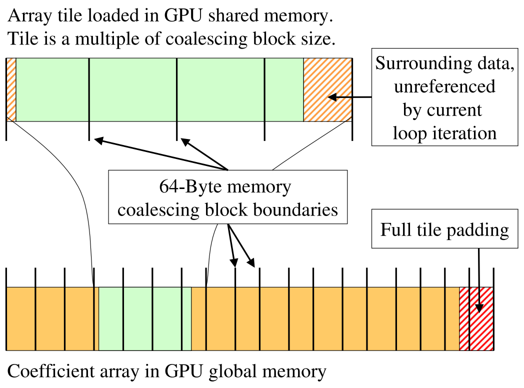
Array tile loaded in GPU shared memory.
Tile is a multiple of coalescing block size.
Surrounding data, unreferenced by current loop iteration
64-Byte memory coalescing block boundaries
Full tile padding
Coefficient array in GPU global memory

Figure 4: Schematic representation of the tiling strategy used to load subsets of large arrays, stored in GPU global memory, into small regions of the high performance on-chip shared memory.

to the CUDA documentation, the `exp2f()` function takes only 32 clock cycles and has a maximum of 2 ulp error over the full range of output values. Due to the high performance and bounded worst case error, our GPU kernels pre-multiply the incoming basis function exponent coefficients by $\log_2 e$ so that `exp2f()` may be used. Compared to CPU kernels, the use of CUDA's `exp2f()` provides a very high level of performance, even before taking into account the massive parallelism available on the GPU. The main algorithmic consideration becomes the effective use of GPU on-chip caches and shared memory to keep the arithmetic units supplied with operands.

The MO algorithm accesses several variably-sized arrays of operands, any one of which could exceed the capacity of on-chip caches and shared memory. We follow two strategies to achieve high performance. In cases where all operands will fit within the 64-kB constant memory, we use a direct implementation of the basic MO algorithm, broadcasting operands from constant memory to all threads in the thread block (at near register speed). In cases where operands will not fit entirely in constant memory, we use an algorithm that collectively loads blocks or tiles of operands into shared memory prior to entering performance critical loops. The coefficients in a shared memory tile are accessed by all threads within a thread block, reducing global memory accesses by a factor of 64 (the number of threads in a thread block). Infrequently accessed coefficients in outer loops are loaded in a simple blocked structure which packs groups of operands together (with appropriate padding) in a single 64-byte memory block, guaranteeing coalesced global memory access. By packing multiple disparate operands into the same memory block, the number of global memory reads is significantly reduced. This simple blocking strategy is used for the atom coordinates, basis set indices, and number of shells per atom in the outermost atom loop, and for the primitive count and shell type data in the loop over shells. In the case of the innermost loops, global memory reads are minimized by loading large tiles immediately prior to the loop over basis set primitives and the loop over angular momenta, respectively. For high performance, tiles must be a multiple of the 64-byte read/write size required for coalesced global memory transactions. Power-of-two tile sizes are preferable, as they simplify addressing arithmetic. Since the number of elements referenced within the two key inner loops is variable, tiles must be large enough to guarantee that all of the data required for execution of the entire loop will be resident in shared memory. Figure 4 illustrates the relationship between coalesced memory block sizes, the portion of a loaded array that will be referenced during the next pass of the innermost loops, and global memory padding and unreferenced data that exist to simplify addressing arithmetic and to guarantee coalesced global memory accesses.

Since the highest performance memory systems on the GPU have very limited capacity and because the MO computation requires many memory references in the innermost loops, our algorithms are optimized by constraining the size of two key arrays referenced in the innermost loops, the basis set and wavefunction coefficient arrays. The size of the basis set array is limited by the maximum degree-of-contraction (number of Gaussian primitives per shell) supported by existing quantum chemistry packages, and in practice is limited to that found in existing basis sets. The program Gaussian [7] supports a maximum degree-of-contraction of 100. The basis set with the largest degree of contraction found in the Basis Set Exchange Library (EMSL) [19] is WTBS [10] with 30 primitives per shell for cesium and some of the lanthanoids. Commonly used basis sets typically contain fewer than 15 primitives per shell. We selected a maximum degree-of-contraction limit of 64 for our CUDA kernels, double that of the largest published basis set. The program GAMESS supports shell types up to and including g, NWChem goes one shell type further to h shells, and Gaussian supports shells of arbitrary angular momenta. The tiled-shared kernel is limited to l shells due to practical limitations in shared memory tile size. Though very rarely used in practice, shell types higher than l could be handled with slightly reduced performance by creating a new kernel variation with angular momenta tile loading tests relocated to the inside of the angular momenta loop. This would add a small amount of additional overhead to each loop iteration, but would handle arbitrary shell types (more precisely, an unlimited number of angular momenta for a single shell).

We have just started to explore a significant optimization opportunity that dynamically generates a molecule-specific GPU kernel when a molecule is initially loaded, and the kernel may be reused from then on. Since Algorithm 1 is very data-dependent, we observe that instructions for loop control and conditional execution can be eliminated for a given molecule. The generation and just-in-time (JIT) compilation of kernels at runtime has associated overhead that must be considered when determining how much code to convert from data-dependent form into a fixed sequence of operations. The GPU MO kernel is dynamically generated by emitting the complete arithmetic sequence normally performed by looping over shells, primitives, and angular momenta for each atom type. This on-demand kernel generation scheme eliminates the overhead associated with loop control instructions (greatly increasing the arithmetic density of the resulting kernel) and allows the GPU to perform much closer to its peak floating point arithmetic rate. At present, CUDA lacks a mechanism for runtime compilation of C-language source code, but provides a mechanism for runtime compilation of the PTX intermediate pseudo-assembly language through a driver-level interface. We could not evaluate the current implementation of the CUDA dynamic compilation feature since our target application VMD uses the CUDA runtime API, and intermixing of driver and runtime APIs is not presently allowed. To evaluate the technique, we implemented a code generator within VMD and saved the dynamically generated kernel source code to a text file. The standard batch mode CUDA compilers were then used to recompile VMD incorporating the generated CUDA kernel. While not a viable solution for actual use, this methodology allowed us to run performance tests and verify the correctness of the results produced by the dynamically generated kernel. We expect future versions of CUDA to make the dynamic compilation functionality available through the runtime API, possibly also supporting compilation from C source code. The OpenCL [12] standards document indicates support for runtime compilation of kernels written in C, ideally providing a vendor-neutral mechanism for dynamic generation and execution of optimized MO kernels.

4. PERFORMANCE EVALUATION

The performance of the MO algorithm implementations was evaluated on several hardware platforms with multiple compilers, using several datasets. The test datasets were selected to be representative of the range of quantum chemistry simulation data that researchers often work with, and to exercise the limits of our algorithms, particularly in the case of the GPU. The benchmarks were run on a Sun Ultra 24 workstation containing a 2.4 GHz Intel Core 2 Q6600 quad core CPU running 64-bit Red Hat Enterprise Linux version 4 update 6. The CPU code was compiled using the GNU C compiler (gcc) version 3.4.6 or Intel C/C++ Compiler (icc) version 9.0. GPU benchmarks were performed using the NVIDIA CUDA programming toolkit version 2.0, running on GeForce 8800 GTX (G80) and GeForce GTX 280 (GT200) GPUs.

4.1 Comparison of CPU and GPU performance for carbon-60

All of the MO kernels presented have been implemented in an experimental version of the molecular visualization program VMD [9]. For comparison of the CPU and GPU implementations, a computationally demanding carbon-60 test case was selected. The C_{60} system was simulated with GAMESS, resulting in a log file containing all of the wavefunction coefficients, basis set, and atomic element data, which was then loaded into VMD. The MO was computed on a lattice with a 0.075 Å spacing, with lattice sample dimensions of 172 x 173 x 169. The C_{60} test system contained 60 atoms, 900 wavefunction coefficients, 15 unique basis set primitives, and 360 elements in the per-shell primitive count and shell type arrays. The performance results listed in Table 2 compare the runtime for computing the MO lattice on both single and multiple CPU cores, and on two generations of GPUs using a variety of kernels. The CPU kernels labeled with "gcc" and "icc" were compiled using GNU C/C++ and Intel C/C++, respectively.

The "cephes" labeled tests used an inlined implementation of the `expf()` routine derived from the Cephes [11] mathematical library. The "icc-sse-cephes" test cases benchmark a SIMD-vectorized SSE adaptation of the scalar Cephes `expf()` routine, hand-coded using compiler intrinsics that are translated directly into x86 SSE machine instructions. The "approx" test cases replace the use of `expf()` with a fast exponential approximation algorithm tailored for the domain of values encountered in the MO computation, with error acceptable for the purposes of MO visualization. The "icc-sse-approx" test case refers to a SIMD-vectorized SSE implementation of the same algorithm, hand-coded using compiler intrinsics.

The CUDA "const-cache" test cases store all of the MO coefficients entirely within the small 64-kB GPU constant memory. The "const-cache" kernel is only applicable to data sets that fit within the fixed-size arrays on constant memory, as defined at compile-time, so it represents a best-case performance scenario for the GPU. The "const-cache-jit" test case evaluates the performance of a dynamically generated CUDA kernel resulting from completely unrolling the loops over shells, primitives, and angular momenta for each atom type, and storing all coefficients in constant memory. The dynamic kernel generation greatly increases the arithmetic intensity of the resulting kernel compared to the fully general loop-based kernels. The CUDA "tiled-shared" test cases perform coalesced global memory reads of tiled data blocks into high-speed on-chip shared memory, capable of processing problems of arbitrary size. All of the benchmark test cases were small enough to reside within the GPU constant memory after preprocessing removed duplicate basis sets, so the "tiled-shared" test cases were conducted by overriding the runtime dispatch heuristic, forcing execution using the tiled-shared CUDA kernel.

The results listed in Table 2 illustrate the tremendous speedups achievable by computing MOs with the use of a GPU, or multi-core SSE SIMD kernels on the CPU. The "icc-sse-cephes" kernel running on a single CPU core was selected as the basis for normalizing performance results because it represents the best-case single-core CPU performance, with a full-precision kernel based on the Cephes `expf()` routine. By benchmarking on a single core, there is no competition for limited CPU cache and main memory bandwidth, and one can more easily extrapolate performance for an arbitrary number of cores. Most computers used for scientific visualization and analysis tasks now contain at least four cores, so the four-core CPU results for each kernel are representative of a typical use-case today. The "gcc-approx" kernel performs very well despite the fact that it does not take advantage of SSE SIMD vectorization. The four-core CPU timings indicated scaling results of 3.97 for "icc-sse-cephes" and 3.90 for "icc-sse-approx" versus their respective single-core runs.

The CUDA kernels achieve excellent performance results, due to the massive array of floating point units and high-bandwidth memory systems on the GPU. The full-precision kernels achieve 10% to 15% higher performance than the "approx" variants because the GPU has dedicated hardware for evaluation of special functions like `expf()` and `exp2f()`. The "const-cache" CUDA kernels take full advantage of the 64-kB constant memory on the GPU and bring it to good effect in keeping the floating point units supplied with operands. The "const-cache-jit" test cases demonstrate that a customized kernel can improve performance by a significant margin. In the case of the carbon-60 test case, the elimination of loop control overhead yields a 40% performance increase over the fastest loop-based implementation. The speedup for a dynamically generated CUDA kernel depends on the molecular system, where the largest performance boost occurs in cases

Kernel	cores/GPUs	Runtime (s)	Speedup
Intel Q6600, gcc-cephes	1	200.22	0.23
Intel Q6600, gcc-cephes	4	51.52	0.90
Intel Q6600, icc-sse-cephes	1	46.58	1.00
Intel Q6600, icc-sse-approx	1	14.82	3.14
Intel Q6600, icc-approx	4	13.13	3.55
Intel Q6600, icc-sse-cephes	4	11.74	3.97
Intel Q6600, gcc-approx	4	10.21	4.56
Intel Q6600, icc-sse-approx	4	3.76	12.4
CUDA 8800 GTX (G80), tiled-shared-approx	1	1.05	44.4
CUDA 8800 GTX (G80), tiled-shared	1	0.89	52.0
CUDA 8800 GTX (G80), const-cache-approx	1	0.63	73.9
CUDA 8800 GTX (G80), const-cache	1	0.57	81.7
CUDA 8800 GTX (G80), const-cache-jit	1	0.41	114.
CUDA GTX 280 (GT200), tiled-shared-approx	1	0.54	85.6
CUDA GTX 280 (GT200), tiled-shared	1	0.46	100.
CUDA GTX 280 (GT200), const-cache-approx	1	0.41	114.
CUDA GTX 280 (GT200), const-cache	1	0.37	126.
CUDA GTX 280 (GT200), const-cache-jit	1	0.27	173.

Table 2: Comparison of MO kernels and hardware performance for the carbon-60 test case.

where the loops over primitives and angular momenta have a very low trip count, with the loop control contributing a greater percentage to the total cost. For the carbon-60 test case, we see that the loop control and array indexing instructions were responsible for approximately 28% of the runtime on both the G80- and GT200-based GPUs. These early results illustrate the benefits of dynamic code generation, motivating further investigations into the application of this technique.

The "tiled-shared" kernel performs competitively given the amount of additional control logic and barrier synchronizations that must be executed in each pass through the MO shell loop. The cost of the additional control logic and the need for global memory references in all of the outermost loops slow the performance of the tiled-shared kernel, increasing its runtime by 57% compared to the const-cache kernel on the GeForce 8800 GTX GPU and by 24% on GeForce GTX 280. The GeForce GTX 280 gains a 54% increase in performance for the const-cache kernel but a 62% increase in memory bandwidth (140 GB/sec vs. 86 GB/sec) relative to the GeForce 8800 GTX. The increased memory bandwidth of the GeForce GTX 280 is likely one of the dominant factors explaining the disparity in the performance drop observed for the tiled-shared kernel on GeForce 8800 GTX as compared to the GeForce GTX 280. The 57% performance loss on G80 may also be partially attributed to inadequate hiding of global memory latency. The G80 GPUs have half the number of registers of GT200, making it more difficult for GPU kernels with large register counts to fully hide global memory latency due to limitations in the number of thread blocks that can be scheduled simultaneously (known as *occupancy*). The low GPU occupancy is the result of the large number of registers (28) and the large amount of shared memory consumed by the tiled-shared kernel. The tiled-shared kernel achieves only 33.3% occupancy on G80 (8 co-scheduled warps of a possible 24), compared with 37.5% (12 warps of a possible 32) on GT200.

The approximation-based algorithms tested here, although slower for CUDA kernels, will still prove useful in constructing cutoff distance algorithms for calculating an MO. The exponents ζ that describe the width of the GTOs in Eq. (2) can vary widely in magnitude for an MO. Replacing `expf()` with an approximation that decreases smoothly to zero will guarantee the truncation of the Gaussian terms beyond fixed spheres of lattice points about each atom without introducing visual artifacts. This idea suggests investigation of a new set of faster algorithms that employ cutoff distances and have computational complexity that is linear in the number of atoms. The performance benefit to CPU implementations could be substantial, and the design of suitable GPU kernels would be based on our prior work [16]. These faster algorithms would enable the visualization of MOs for much larger systems of atoms than is presently feasible.

4.2 Comparison with other software packages

To evaluate the practical merits of our approach we compared the performance of VMD using CPU and GPU implementations of our algorithms with that of the popular quantum chemistry visualization packages MacMolPlt and Molekel. In order to gain a broad perspective of performance characteristics of each package/kernel, performance was assessed over a range of test structures listed in Table 3, and shown in Fig. 1. It was impractical to benchmark exactly the same calculation for each package/kernel combination since each package places limits on the size and spacing of the MO lattice, and performance levels ranged over two orders of magnitude. We measured performance in terms of lattice points computed per second, selecting a large enough lattice in each case to provide execution timings accurate to 5% or better. All of the CPU tests were allowed to use all four cores except Molekel, which is a sequential code. The GPU tests were performed using a single CPU core and a single GPU. All of the test molecules were small enough to fit within the GPU constant memory; accordingly, the constant-cache kernel was used in all of these benchmarks.

The performance results presented in Table 4 show the practical merit of the algorithms in this paper in actual use. In all but one case, VMD outperformed the other two packages. The exceptional "Kr-a" test case is noteworthy as a very small single-atom Krypton model with only 19 unique basis functions. In this test, MacMolPlt outperformed VMD by a very small margin. We attribute this to the added cost of basis set indirection used as part of the elimination of redundant basis functions in our implementation. For

	system	atoms	basis set	basis functions (unique)
C60-a	carbon-60	60	STO-3G	300 (5)
C60-b	carbon-60	60	6-31Gd	900 (15)
Thr-a	threonine	17	STO-3G	49 (16)
Thr-b	threonine	17	6-31+Gd	170 (59)
Kr-a	krypton	1	STO-3G	19 (19)
Kr-b	krypton	1	cc-pVQZ	84 (84)

Table 3: List of test systems and their respective attributes. Three test systems were selected, each with two variants containing the same number of atoms, but using a basis set with a smaller (a) and larger (b) number of unique basis functions.

Program/Kernel	cores	C60-a	C60-b	Thr-a	Thr-b	Kr-a	Kr-b
Molekel CPU	1	39	25	175	108	617	138
MacMolPlt CPU	4	97	66	361	265	2668	632
VMD gcc-cephes	4	126	100	518	374	2655	892
VMD icc-sse-cephes	4	658	429	2428	1366	10684	2968
VMD gcc-approx	4	841	501	2641	1828	11055	4060
VMD icc-sse-approx	4	2314	1336	8829	5319	33818	9631
VMD CUDA 8800 GTX	1	14166	8565	45015	32614	104576	61358
VMD CUDA GTX 280	1	21540	13338	62277	45498	119167	78884

Table 4: Comparison of MO computation performance for MacMolPlt, Molekel, and VMD. The performance values are given in units of 10^3 lattice points per second. Higher numbers indicate better performance.

all of the larger test cases, VMD outperformed the other packages even with the non-SSE kernel compiled with GNU C/C++. The performance differences among the VMD kernels mirror the results in Table 2, but with two noteworthy exceptions. The “Kr-a” test case showed an unusually small difference in performance between the GeForce 8800 GTX (G80) and the GeForce GTX 280. It seems likely that the high cache locality occurring for the Kr-a model neutralizes the benefit of some architectural improvements of the GT200-based GPU versus the G80-based GPU. The “C60-b” test case (with the largest number of basis functions) shows the largest performance gap between G80 and GT200, likely indicating that GT200 cache performs better on large data than G80. The performance levels obtained for VMD using the CUDA kernels enable on-the-fly computation and animation of molecules up to the size of the C_{60} test cases, with typical lattice sizes.

5. CONCLUSION

We have presented new molecular orbital electron density algorithms that exploit the unique architectural strengths of GPUs to achieve performance levels far beyond those achievable with conventional CPUs, enabling for the first time fully interactive visualization and animation of quantum chemistry simulation trajectories. The early results presented for dynamic GPU kernel generation and just-in-time compilation demonstrate the potential benefits of this technique, motivating further work in this area. We have implemented these algorithms within the popular molecular visualization tool VMD [9] and expect to apply our computational techniques to all of the related, spatially evaluated functions used for visualization of quantum chemistry data.

Acknowledgments

This work was supported by the National Institutes of Health under grant P41-RR05969. Performance experiments were made possible by a generous hardware donation by NVIDIA.

6. REFERENCES

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Chem. Phys.*, 227(10):5342–5359, 2008.

[2] B. M. Bode and M. S. Gordon. MacMolPlt: a graphical user interface for GAMESS. *J. Mol. Graph. Model.*, 16(3):133–138, June 1998.

[3] E. J. Bylaska et al. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1.* Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA, 2007.

[4] C. J. Cramer. *Essentials of Computational Chemistry.* John Wiley & Sons, Ltd., Chichester, England, 2004.

[5] E. R. Davidson and D. Feller. Basis set selection for molecular calculations. *Chem. Rev.*, 86:681–696, 1986.

[6] M. M. Francl, W. J. Pietro, W. J. Hehre, J. S. Binkley, M. S. Gordon, D. J. DeFrees, and J. A. Pople. Self-consistent molecular orbital methods. XXIII. A polarization-type basis set for second-row elements. *J Chem Phys*, 77:3654–3665, 1982.

[7] M. J. Frisch et al. Gaussian 03 (Revision B.05). Gaussian, Inc., Pittsburgh, PA, 2003.

[8] D. J. Hardy, J. E. Stone, and K. Schulten. Multilevel summation of electrostatic potentials using graphics processing units. *J. Paral. Comp.*, 2009. In press.

[9] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J. Mol. Graphics*, 14:33–38, 1996.

[10] S. Huzinaga and M. Klobukowski. Well-tempered Gaussian basis sets for the calculation of matrix Hartree-Fock wavefunctions. *Chem. Phys. Lett.*, 212:260–264, 1993.

[11] S. L. Moshier. Cephes Mathematical Library Version 2.8, June 2000. http://www.moshier.net/#Cephes.

[12] A. Munschi. OpenCL Specification Version 1.0, Dec. 2008. http://www.khronos.org/registry/cl/.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[14] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proc. IEEE*, 96:879–899, 2008.

[15] S. Portmann and H. P. Lüthi. Molekel: An interactive molecular graphics tool. *CHIMIA*, 54:766–770, 2000.

[16] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W.-M. W. Hwu. GPU acceleration of cutoff pair potentials for molecular modeling applications. In *CF'08: Proceedings of the 2008 conference on Computing Frontiers*, pages 273–282, New York, NY, USA, 2008. ACM.

[17] G. Schaftenaar and J. H. Nooordik. Molden: a pre- and post-processing program for molecular and electronic structures. *J. Comp.-Aided Mol. Design*, 14(2):123–134, 2000.

[18] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. J. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. General atomic and molecular electronic structure system. *J. Comp. Chem.*, 14:1347–1363, 1993.

[19] K. L. Schuchardt, B. T. Didier, T. Elsethagen, L. Sun, V. Gurumoorthi, J. Chase, J. Li, and T. L. Windus. Basis set exchange: A community database for computational sciences. *J. Chem. Inf. Model.*, 47:1045–1052, 2007.

[20] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *J. Comp. Chem.*, 28:2618–2640, 2007.

[21] I. Ufimtsev and T. Martinez. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *J. Chem. Theor. Comp.*, 4(2):222–231, 2008.