JCTC Journal of Chemical Theory and Computation

pubs.acs.org/JCTC Article

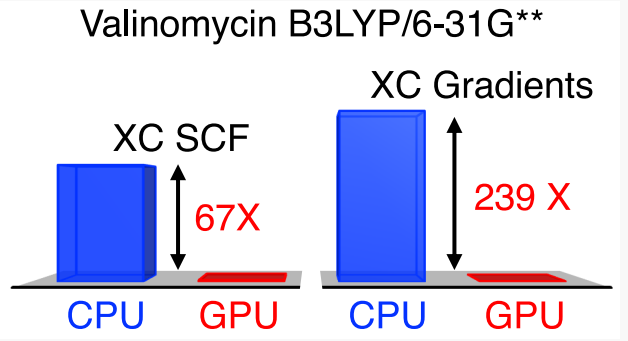
Parallel Implementation of Density Functional Theory Methods in the Quantum Interaction Computational Kernel Program

Madushanka Manathunga, Yipu Miao, Dawei Mu, Andreas W. Götz,* and Kenneth M. Merz, Jr.*

Cite This: *J. Chem. Theory Comput.* 2020, 16, 4315−4326 Read Online

ACCESS | Metrics & More | Article Recommendations | Supporting Information

ABSTRACT: We present the details of a graphics processing unit (GPU) capable exchange correlation (XC) scheme integrated into the open source QUantum Interaction Computational Kernel (QUICK) program. Our implementation features an octree based numerical grid point partitioning scheme, GPU enabled grid pruning and basis and primitive function prescreening, and fully GPU capable XC energy and gradient algorithms. Benchmarking against the CPU version demonstrated that the GPU implementation is capable of delivering an impressive performance while retaining excellent accuracy. For small to medium size protein/organic molecular systems, the realized speedups in double precision XC energy and gradient computation on a NVIDIA V100 GPU were 60−80-fold and 140−500-fold, respectively, as compared to the serial CPU implementation. The acceleration gained in density functional theory calculations from a single V100 GPU significantly exceeds that of a modern CPU with 40 cores running in parallel.



1. INTRODUCTION

Although graphics processing units (GPUs) were originally introduced for rendering computer graphics, they have become essential devices to enhance the performance of scientific applications over the past decade. Examples of GPU accelerated applications span from bioinformatics software that help solving genetic mysteries in biology to data analysis tools aiding gravitational wave detection in astrophysics.[1,2] The availability of powerful general purpose GPUs at a reasonable cost, convenient computing and programming environments such as Compute Unified Device Architecture (CUDA),[3] and especially, the fact that GPUs can perform trillions of floating point operations per second in combination with a high memory bandwidth, outperforming desktop central processing units (CPUs), are the main reasons for this trend.

GPUs are also known to deliver outstanding performance in traditional computational chemistry applications, particularly in classical molecular dynamics (MD) simulations[4−12] and *ab initio* quantum chemical calculations.[13−30] In the latter context, Hartree−Fock (HF)[13,14,23−28] and post-HF energy and gradient implementations[15−17,29−31] on GPUs have displayed multifold speedups for molecular systems containing a few to large number of atoms. Nevertheless, a majority of the computational chemistry community is unable to enjoy such performance benefits due to the unavailability of an open-source, user-friendly, GPU enabled quantum chemistry software. Toward this end, we have been developing a quantum chemical code named the QUantum Interaction Computational Kernel (QUICK) to fill this void.[27,28] As reported previously, QUICK is capable of efficiently computing HF energies and gradients. For instance, the speedup realized for moderate size molecular systems on a Kepler type GPU was about 10−20 times in comparison to a single CPU core while retaining an excellent accuracy. With high angular momentum basis functions (d and f functions), the realized speedup remained 10−18-fold. Such performance gain was primarily due to GPU accelerated electron repulsion integral (ERI) calculations. In our ERI engine, integrals are calculated using vertical and horizontal recurrence relations algorithms[32,33] reported by Obara and Saika and Head-Gordon and Pople. The integrals are calculated on the fly, and the Fock matrix is assembled on the GPU using an efficient direct self-consistent field (SCF) scheme. However, the accuracy of the HF method is insufficient, if not totally unsuitable, to study many chemical problems; but having a GPU enabled HF code paves the way toward an efficient post-HF or density functional theory (DFT) package. In the present work, we have undertaken the task of implementing the latter type of methods in QUICK. In fact, given the vast number of research

Received: March 25, 2020
Published: June 8, 2020

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

articles published using DFT methods over the past decades,[34] incorporation of such methods into our package was essential.

In the context of GPU parallelization of Gaussian based DFT calculations, a few publications have appeared in the literature.[35−37] Among these, Yasuda's work[35] is the earliest. His exchange correlation (XC) quadrature scheme consisted of several special features aimed at maximizing performance on GPU hardware. These include partitioning numerical grid points in 3-dimensional space, basis function prescreening, and preparation of basis function lists for grid point partitions. Only the electron densities and their gradients and matrix elements of the XC potential were calculated on the GPU. The hardware available at the time limited his algorithm to single precision calculations; but the reported accuracy of benchmark tests was about 10^{-5} au. A similar algorithm has been implemented by Martinez and co-workers in their GPU capable Terachem software.[36] In addition to grid point partitioning, this algorithm performs prescreening at the level of primitive functions and excludes certain partitions from the calculation based on a sorting procedure. In both of the above implementations, the values of the density functionals at the grid points are calculated on the CPU. We note another DFT package where the XC scheme is GPU enabled; however, the ERI calculations are performed on the CPU.[37]

The features of the XC quadrature scheme reported in the current work include grid point partitioning using an octree algorithm, prescreening and grid pruning based on the value of primitive Gaussian functions, and fully GPU enabled XC energy and gradient calculations where not only the electron density and its derivatives but also the XC functional values are computed on the GPU. The next sections of this paper are organized as follows. In section 2, we give an overview of the underlying theory of our XC scheme, which was originally documented by Pople and co-workers.[38] The details of the computational implementation are then presented in section 3. Here we first discuss important aspects of data parallel GPU programming. The GPU version of the XC scheme is then implemented following these considerations. Information regarding the parallel CPU implementation using the message passing interface (MPI)[39] is also presented. Section 4 is devoted to benchmark tests and discussion. The tests include performance comparisons between QUICK and the GAMESS GPU version[24,25,40] and accuracy and performance comparisons between the QUICK GPU and CPU versions. Finally, we conclude our discussion by exploring future directions for further improvement.

2. THEORY

The Kohn−Sham formulation of DFT differs from the HF method by its treatment of the exchange and correlation contributions to the electronic energy. The total electronic energy (E) is given by[38]

$$E = E^{\mathrm{T}} + E^{\mathrm{V}} + E^{\mathrm{J}} + E^{\mathrm{xc}} \tag{1}$$

where E^{T} and E^{V} stand for kinetic and electron−nuclear interaction energies, E^{J} is the Coulomb self-interaction of the electron density, and E^{xc} is the remaining electronic exchange and correlation energies. The electron density (ρ) is a summation of α and β electron densities (ρ_α and ρ_β, respectively) that can be expressed by choosing sets of orthonormal spin orbitals ψ_i^α (i = 1, ..., n_α) and ψ_i^β (i = 1, ..., n_β) as

$$\rho_\alpha = \sum_i^{n_\alpha} |\psi_i^\alpha|^2, \quad \rho_\beta = \sum_i^{n_\beta} |\psi_i^\beta|^2 \tag{2}$$

where n_α and n_β are the number of occupied orbitals.

E^{T}, E^{V}, and E^{J} can be written as

$$E^{\mathrm{T}} = \sum_i^{n_\alpha} \left\langle \psi_i^\alpha \middle| -\frac{1}{2}\nabla^2 \middle| \psi_i^\alpha \right\rangle + \sum_i^{n_\beta} \left\langle \psi_i^\beta \middle| -\frac{1}{2}\nabla^2 \middle| \psi_i^\beta \right\rangle \tag{3}$$

$$E^{\mathrm{V}} = -\sum_A^{\mathrm{nucl}} Z_A \int \rho(r)|r - r_A|^{-1}\, \mathrm{d}r \tag{4}$$

$$E^{\mathrm{J}} = \frac{1}{2} \iint \rho(r_1)|r_1 - r_2|^{-1}\rho(r_2)\, \mathrm{d}r_1\, \mathrm{d}r_2 \tag{5}$$

E^{xc}, within the generalized gradient approximation (GGA), may be given by a functional f that depends on electron densities and their gradient invariants,

$$E^{\mathrm{xc}} = \int f(\rho_\alpha, \rho_\beta, \gamma_{\alpha\alpha}, \gamma_{\alpha\beta}, \gamma_{\beta\beta})\, \mathrm{d}r \tag{6}$$

$$\gamma_{\alpha\alpha} = |\nabla\rho_\alpha|^2, \quad \gamma_{\alpha\beta} = \nabla\rho_\alpha . \nabla\rho_\beta, \quad \gamma_{\beta\beta} = |\nabla\rho_\beta|^2 \tag{7}$$

In practical computational implementations, one expresses molecular orbitals as a linear combination of atomic orbitals, and ρ_α in eq 2 becomes,

$$\rho_\alpha = \sum_\mu^N \sum_\nu^N \sum_i^{n_\alpha} (C_{\mu i}^\alpha) C_{\nu i}^\alpha \phi_\mu \phi_\nu = \sum_{\mu\nu} P_{\mu\nu}^\alpha \phi_\mu \phi_\nu \tag{8}$$

Here φ_μ (μ = 1, ..., N) are the atomic orbitals, and $P_{\mu\nu}^\alpha$ is the density matrix. Furthermore, the gradient of ρ_α can be written as

$$\nabla\rho_\alpha = \sum_{\mu\nu} P_{\mu\nu}^\alpha \nabla(\phi_\mu \phi_\nu) \tag{9}$$

A similar expression can be written for ρ_β. Substituting eqs 8 and 9 into eqs 2−7 and into the energy eq 1 and minimizing with respect to coefficients $C_{\mu i}^\alpha$ and $C_{\nu i}^\alpha$, one obtains a series of algebraic equations similar to the conventional HF procedure. The resulting Fock-type matrix (hereafter Kohn−Sham matrix) for ρ_α is

$$F_{\mu\nu}^\alpha = H_{\mu\nu}^{\mathrm{core}} + J_{\mu\nu} + F_{\mu\nu}^{\mathrm{XC}_\alpha} \tag{10}$$

where $H_{\mu\nu}^{\mathrm{core}}$ is the one electron Hamiltonian matrix. $J_{\mu\nu}$ is the Coulomb matrix, which can be written in the conventional form as

$$J_{\mu\nu} = \sum_{\lambda\sigma}^N P_{\lambda\sigma}(\mu\nu|\lambda\sigma), \quad P_{\lambda\sigma} = P_{\lambda\sigma}^\alpha + P_{\lambda\sigma}^\beta \tag{11}$$

The XC contribution to the Kohn−Sham matrix ($F_{\mu\nu}^{\mathrm{XC}_\alpha}$) is given by

$$F_{\mu\nu}^{\mathrm{XC}_\alpha} = \int \left[\frac{\partial f}{\partial \rho_\alpha}\phi_\mu\phi_\nu + \left(2\frac{\partial f}{\partial \gamma_{\alpha\alpha}}\nabla\rho_\alpha + \frac{\partial f}{\partial \gamma_{\alpha\beta}}\nabla\rho_\beta \right) \cdot \nabla(\phi_\mu\phi_\nu) \right] \mathrm{d}r \tag{12}$$

and a similar expression can be written for $F_{\mu\nu}^{\mathrm{XC}_\beta}$.

The gradient with respect to the position of nucleus A is,

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

$$\nabla_A E = \sum_{\mu\nu}^{N} P_{\mu\nu}(\nabla_A H_{\mu\nu}^{core}) + \frac{1}{2}\sum_{\mu\nu\lambda\sigma}^{N} P_{\mu\nu}P_{\lambda\sigma}\nabla_A(\mu\nu|\lambda\sigma) - \sum_{\mu\nu}^{N} W_{\mu\nu}(\nabla_A S_{\mu\nu}) - 2\sum_{\mu}{}' \sum_{\nu}^{N} P_{\mu\nu}^{\alpha} \int\left[\frac{\partial f}{\partial \rho_\alpha}\phi_\nu\nabla\phi_\mu + X_{\mu\nu}\left(2\frac{\partial f}{\partial \gamma_{\alpha\alpha}}\nabla\rho_\alpha + \frac{\partial f}{\partial \gamma_{\alpha\beta}}\nabla\rho_\beta\right)\right] dr - 2\sum_{\mu}{}' \sum_{\nu}^{N} P_{\mu\nu}^{\beta} \int\left[\frac{\partial f}{\partial \rho_\beta}\phi_\nu\nabla\phi_\mu + X_{\mu\nu}\left(\frac{\partial f}{\partial \gamma_{\alpha\beta}}\nabla\rho_\alpha + 2\frac{\partial f}{\partial \gamma_{\beta\beta}}\nabla\rho_\beta\right)\right] dr \quad (13)$$

where primed sums denote that μ is centered on nucleus A, $S_{\mu\nu}$ is the overlap matrix and the energy weighted density matrix, $W_{\mu\nu}$, is given by

$$W_{\mu\nu} = \sum_{i}^{n_\alpha} \epsilon_i^\alpha C_{\mu i}^\alpha C_{\nu i}^\alpha + \sum_{i}^{n_\beta} \epsilon_i^\beta C_{\mu i}^\beta C_{\nu i}^\beta \quad (14)$$

and the matrix element $X_{\mu\nu}$ is

$$X_{\mu\nu} = \phi_\nu\nabla(\nabla\phi_\mu)^t + (\nabla\phi_\mu)(\nabla\phi_\mu)^t \quad (15)$$

Note that for hybrid functionals, the energy in eq 1 will also include a contribution from the HF exchange energy and eqs 10 and 13 will be adjusted accordingly. Due to the complex form of the XC functionals f, the analytical calculation of the integrals required for the XC energy, XC potential, and its gradients in eqs 6, 12, and 13 is impossible; hence, this is usually achieved through a numerical procedure. The key steps of such a procedure involve the formation of a numerical grid (also called XC quadrature grid) with quadrature weights assigned to each grid point, calculation of the electron density and the gradients of the density at each grid point, calculation of the value of the density functional and the derivatives of the functional, and calculation of the XC energy and the contribution to Kohn−Sham matrix (called matrix elements of the XC potential hereafter). In order to compute the nuclear gradients of the XC energy, one must compute second derivatives of the basis functions and values of the two integral terms in eq 13 and add them to the total gradient vector. Finally, due to the involvement of a quadrature weighing scheme in the numerical procedure, the derivatives of the quadrature weights with respect to nuclear displacements must be computed and added to the total gradient.

3. IMPLEMENTATION

3.1. Key Considerations in GPU Programming. GPUs are ideal devices for data parallel computations, that is, computations that can be performed on numerous data elements simultaneously. They allow massive parallelization in comparison to traditional CPU platforms but at the expense of programming complexity and flexibility. Therefore, a proper understanding of the GPU architecture and the memory hierarchy is essential for writing an efficient code that exploits the full power of this hardware class. GPUs perform tasks according to a single instruction multiple data (SIMD) model, which simply means that they execute the same instructions for a given chunk of data and the same amount of work is expected to be performed for each piece of data. Hence, a programmer should organize the data and assign work to threads that are then processed by the GPU in batches known as thread blocks. The number of threads in a block is up to the programmer; however, there exists a maximum limit allowed by the GPU architecture. For instance, the NVIDIA Volta architecture permits a maximum of 1024 threads per block. NVIDIA GPUs execute threads on streaming multiprocessors (SMs) in warps whose size, 32 for recent architectures, is solely determined by the architecture. Each SM (for example, the V100 GPU has 80 SMs, each with 64 CUDA cores for a total of 5120 cores[41]) executes the same set of instructions for all threads in a warp during a given clock cycle. Therefore, it is essential to minimize the branching in GPU codes (device kernels) to avoid instruction divergence.

A GPU possesses its own physical memory that is distinct from the host (CPU accessible) memory. The main memory called global memory or dynamic random-access memory (DRAM) is relatively large (for example, 32 GB or 16 GB for the V100 and 12 GB for Titan V) and accessible by all threads in streaming multiprocessors. However, global memory transactions suffer from relatively high memory latency. A small secondary type of GPU memory called shared memory is also available on each SM. This type of memory transaction is faster but local, meaning that shared memory of a given SM is only accessible by threads within a thread block that is currently executing on this SM. In addition to these two types of memory, threads in a warp have access to a certain number of registers, and this is useful to facilitate the communication between threads of the same warp. GPUs also contain constant and texture memories, which are read-only and capable of delivering a higher performance than global memory in specific applications. If threads in a warp read from the same memory location, constant memory is useful. Texture memory is beneficial when the threads read from physically adjacent memory locations. To maximize the performance of device kernels, careful handling of memory is essential. The key considerations for engineering an efficient GPU code include minimizing the warp divergence, minimizing frequent global memory transactions, maintaining a coherent global memory access pattern by adjacent threads in a block (coalesced memory access), minimizing random memory access patterns, or simultaneously accessing a certain memory location by multiple threads. Furthermore, constant and texture memories should be employed where applicable. Our existing ERI and direct SCF scheme in QUICK was developed in adherence to this philosophy. As detailed below, we implement our XC scheme following the same practices.

3.2. Grid Generation and Pruning. The selected grid system for our work is the Standard Grid-1 (SG-1) reported by Pople and co-workers.[42] This grid consists of 50 radial grid points and 194 angular grid points for each atom of a molecule. The radial grid point generation is performed using the Euler−Maclaurin scheme, and for the angular grid points, Lebedev grids are used. Following the grid generation, we compute weights for each grid point based on the scheme reported by Frisch et al.[43] We then perform grid pruning in two stages. First, all points with weight less than a threshold (usually 10^{-10}) are eliminated. The remaining points are pruned at a second stage based on the value of atom centered basis and primitive functions at each grid point. As described in section 3.3, we make use of an octree algorithm for this purpose.

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

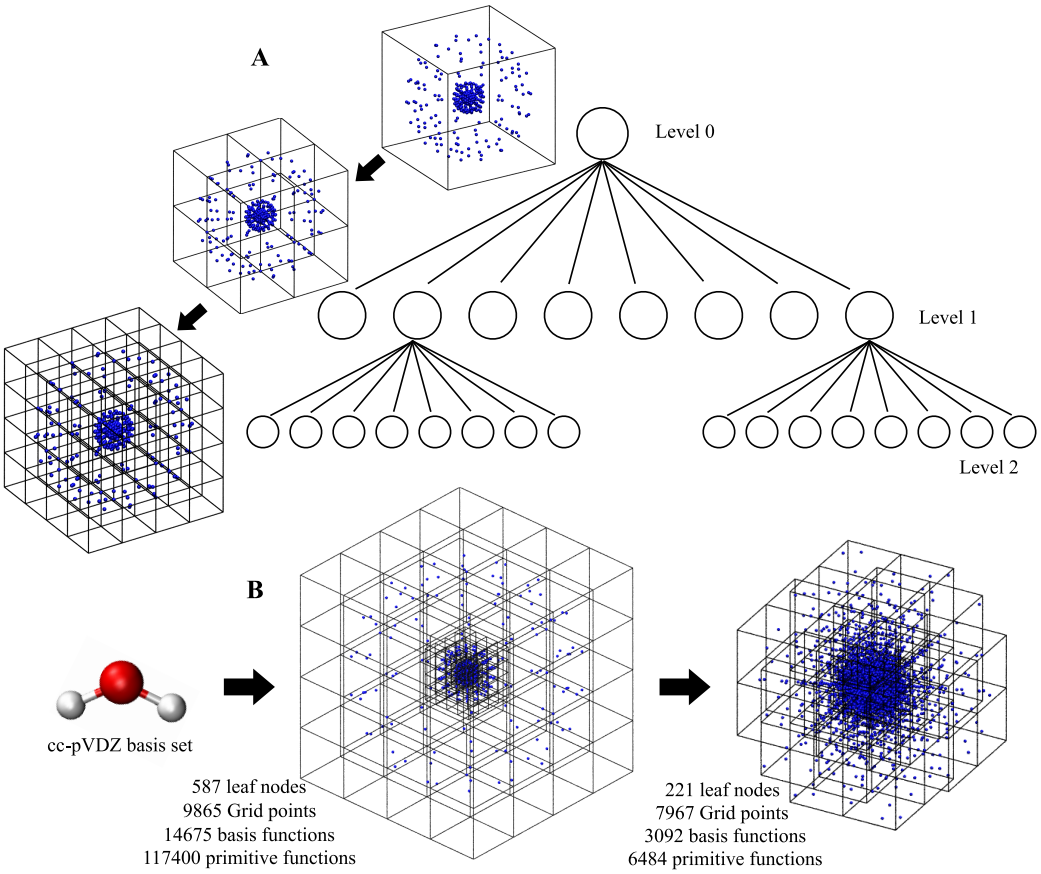


Figure 1. Partitioning of numerical grid points of a water molecule using the octree algorithm. (A) A cell containing all grid points (level 0) is recursively subdivided into octets (levels 1 and 2). (B) Fully partitioned bins or leaf nodes of the same example containing all grid points from pruning in stage 1 (middle) undergoes primitive function based grid pruning. The resulting bins or leaf nodes and grid points are shown at the bottom right.

3.3. Octree Based Grid Pruning, Preparation of Basis and Contracted Function Lists. In our octree algorithm, the grid points in space are partitioned as follows. First, the lower and upper bounds of the grid are determined, and a single cell (node) containing all grid points is formed in 3-dimensional space (see Figure 1A). This node is then divided into eight child nodes. The child nodes whose grid point count is greater than a user specified threshold are recursively subdivided into octets until the point count of each child node falls below the threshold. The nodes that are not further divisible (leaf nodes, also termed bins below) are considered to be fully partitioned. We then go through each grid point of the leaf nodes and compute the values of the basis and primitive functions at their positions. If a basis or primitive function satisfies the condition

$$|(\zeta_{ij} + \nabla\zeta_{ij})| > \tau \tag{16}$$

for a given grid point, it is considered to be significant for that particular point. We use $\tau = 10^{-10}$ as default threshold, which leads to numerical results that are indistinguishable from the reference without pruning. For basis function based prescreening, j in eq 16 becomes μ and ζ_{ij} stands for the value of basis function φ_μ at grid point g_i. Similarly, for primitive function based prescreening, ζ_{ij} represents the value of $c_{\mu_p\chi_p}$ at grid point g_i, where χ_p is the pth primitive function of φ_μ and c_{μ_p} is the corresponding contraction coefficient. Once the basis function values at each grid point are computed, the points that do not have at least one significant basis function are omitted from the corresponding bin (see Figure 1B). Furthermore, bins without any grid points are also discarded. At this stage, lists of significant basis and primitive function IDs are prepared for each remaining bin of grid points, significantly reducing the number of basis function values and derivatives that have to be evaluated at each grid point during the SCF and gradient computations. Using this algorithm, the number of primitive Gaussian basis function evaluations is reduced from 117400 to 6484 for H_2O with a cc-pVDZ basis set (Figure 1B).

3.4. Grid Point Packing and Kernel Launch. Following the two-stage grid pruning and the preparation of basis and primitive function ID lists, the grid points are prepared (hereafter grid point packing) to be uploaded to the GPU. Here we add dummy points into each bin and set the total grid point count in each bin to a threshold value used in the octree run. It is worth noting that the threshold we choose is a multiple of the warp size 32, usually 256. By doing so, we are able to pack true grid points into one corner of a block of an array (see Figure S1A) and this helps us to minimize the warp divergence when we launch GPU threads using the same value for the block size. More specifically, for subsequent calculations (i.e., density, XC energy, and XC gradients), we will launch a total of $256n_{\text{bin}}$ threads (where n_{bin} = number of significant bins) where each thread block contains at least one true grid point and a set of dummy points. The threads launched for dummy points should not perform any computation, and these are differentiated from true points by an assigned integer flag. As mentioned previously, the launched thread blocks are

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

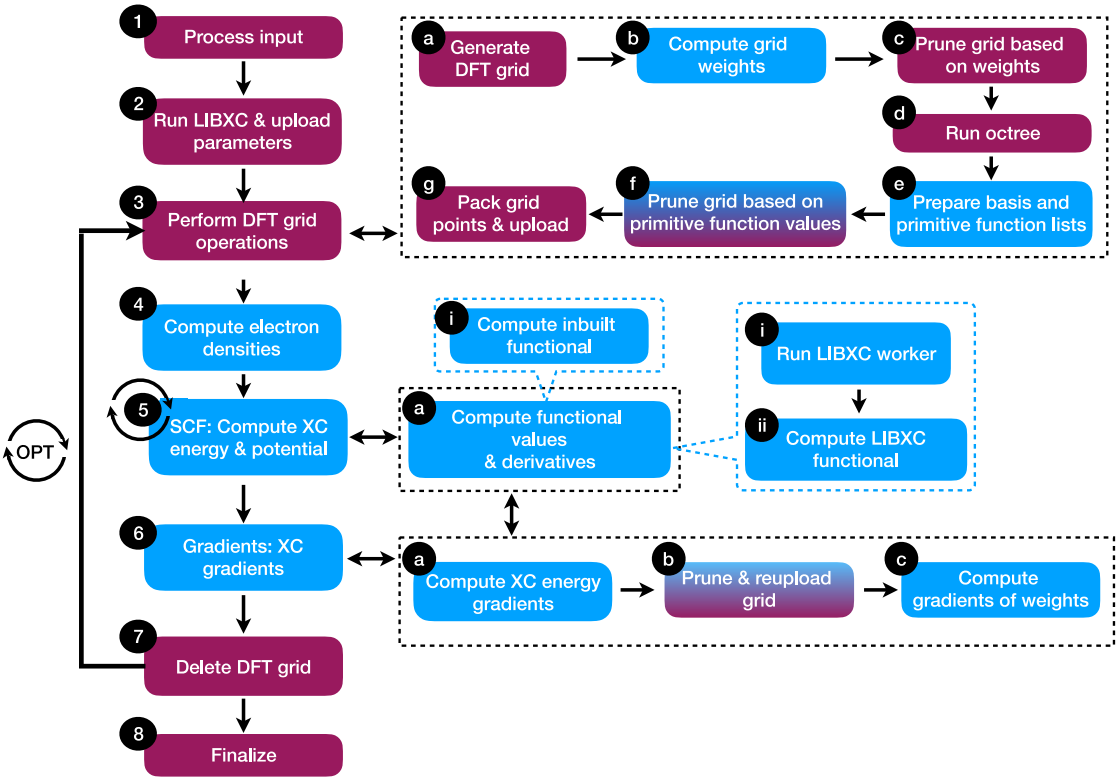


Figure 2. Flowchart depicting the workflow of a DFT geometry optimization calculation. Magenta, blue, and mixed color boxes indicate steps performed on CPU, GPU, and mixed CPU/GPU, respectively. Round arrows represent cyclic procedures. A double headed arrow indicates that a step performs suboperations enclosed by dashed boxes. Callout boxes with blue dashed borders indicate that enclosed operations are optional and can be chosen at users will. OPT stands for geometry optimization. The CPU remains idle during the GPU computations.

submitted to streaming multiprocessors as warps of 32 by the warp scheduler during runtime. This ensures that at most one warp per thread block suffers from warp divergence and threads of the remaining warps would perform the same task. Once the grid points are packed and basis and primitive function lists have been prepared, these are uploaded to the global memory of the GPU. The corresponding array pointers are stored in constant memory, and these are used in launching the kernel and the execution of device kernels during density, SCF, and gradient calculations.

3.5. Computing Electron Densities on the GPU. As apparent from eqs 8 and 9, the calculation of electron densities and their gradients require looping over basis and primitive functions at each grid point. This task is performed on the GPU, and each thread working on a single grid point only loops through basis and primitive function lists assigned to the corresponding thread block. The retrieval of correct lists from global memory is achieved by using two locator arrays (see Figure S1B). Note that we use the term "ID" when referring to basis and primitive functions and "array index" for array locations. The first, called basis function locator array, holds the array index ranges of the basis function ID array. Each thread accesses the former based on their block index, obtains the corresponding array index range of the latter, and picks the basis function IDs. Second, the primitive function locator array holds the array index ranges for accessing the primitive function ID array. Each thread picks elements from the primitive function locator array using basis function array indices, obtains the relevant array index range of the latter, and takes the primitive function IDs. This retrieval strategy allows us to maintain a coalesced memory access pattern.

3.6. Computing the XC Energy and the Matrix Elements of the XC Potential. As reported elsewhere,[27,28] our existing SCF implementation assembles the Fock matrix and computes the energy on the GPU. Therefore, our goal is to calculate the XC energy and associated derivatives on the GPU, which highlights the necessity to have density functionals implemented as device kernels. Needless to say, coding the numerous available functionals is a cumbersome process, and the best solution is to make use of an existing density functional library. Nevertheless, to the best our knowledge, such an appropriate GPU capable library is still not available, but a few stable CPU based density functional libraries have been reported and used in many DFT packages.[44−46] As discussed below, we selected one of these libraries and modified the source code for execution on GPUs via CUDA to achieve our goal.

The chosen density functional library, named LIBXC,[46] is open source and provides about 400 functionals that are based on the local density approximation (LDA), the generalized gradient approximation (GGA), and the meta-generalized gradient approximation (MGGA). This library is well organized and has a user-friendly interface, and above all, the source code is written in the C language. Such factors make LIBXC easily re-engineerable as a GPU capable library with relatively modest coding effort. In LIBXC, the functionals are organized in separate source files and are handled by several workers (each assigned to LDA, GGA exchange, GGA correlation, MGGA exchange, and MGGA correlation functional types) with the aid of a series of intermediate files. When the user provides the necessary input with a desired functional name, an assigned functional ID is selected, and if necessary, parameters

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

Table 1. Comparison of the Average SCF Time and Time to Compute ERI and XC Gradients Using GPU Versions of the QUICK and GAMESS Quantum Chemical Packages[a]

		average SCF time (s) (total number of iterations)		ERI gradient time (s)		XC gradient time (s)	
molecule (atom number)	basis set (function number)	QUICK	GAMESS	QUICK	GAMESS	QUICK	GAMESS
morphine (40)	6-31G (227)	0.6 (18)	3.6 (25)	3.8	9.6	2.3	11.3
	6-31G* (353)	1.4 (19)	5.7 (24)	12.9	60.7	3.8	19.6
	6-31G** (410)	1.8 (22)	6.3 (24)	16.0	72.0	4.4	24.4
Gly_{12} (87)	6-31G (517)	1.5 (22)	12.1 (26)	8.4	23.6	1.7	112.9
	6-31G* (811)	4.1 (24)	22.7 (27)	27.2	164.6	3.4	208.6
	6-31G** (925)	5.4 (24)	26.7 (27)	34.9	201.4	3.6	246.9
valinomycin (168)	6-31G (882)	12.8 (29)	68.2 (27)	85.1	183.2	10.4	686.6
	6-31G* (1350)	28.6 (20)	120.0 (30)	243.0	975.7	18.7	1167.3
	6-31G** (1620)	40.5 (20)	151.9 (29)	317.7	1291.5	19.3	1571.9

[a]BLYP functional is used in all calculations. NVIDIA V100-SXM2 GPU accompanied by Intel Xeon (R) Gold 6138 CPU.

required by the workers to call or send to functionals are obtained from the intermediate files. The functionals are then called from the workers. To make use of LIBXC in our GPU code, we initiate LIBXC and obtain functional ID and necessary intermediate parameters through a dry run immediately after grid point packing but before starting the SCF procedure. In order to do so, we made several minor modifications to the interface and intermediate files. The obtained information is then uploaded to the GPU. We also implemented device kernels for LDA and GGA workers to call functionals during device kernel execution. The corresponding functional source files were also modified with compiler preprocessor directives, and during the compilation time, these are included in the CUDA source files and compiled as device kernels. As reported previously, use of compiler directives for porting complex scientific applications to GPUs is rewarding in terms of application performance and implementation effort.[47] Note that our current XC implementation is not capable of computing kinetic energy densities, and for this reason, we have not made any changes to the MGGA workers and related source files in LIBXC.

During the SCF procedure, the device kernel versions of LIBXC workers are called with prestored functional IDs and other parameters. The worker then calls the appropriate functional kernel. Here we have used C function pointers rather than conditional statements due to potential performance penalties introduced by the latter. Following the computation of XC energy density on grid points, we compute the matrix elements of the potential and update the Kohn−Sham matrix using the CUDA atomic add function. In the past, we employed atomic add in our direct SCF scheme and noted that it is not a critical performance bottleneck.[27,28]

3.7. Computing XC Energy Nuclear Gradients. Most of the implementation strategy described above for XC energy and potential holds for our gradient algorithm. More specifically, this is a two-step process consisting of calculating the XC energy gradients and grid weight gradients. While computing the former involves a majority of steps discussed for the energy calculation (i.e., computing values of the basis functions and derivatives and functional values and associated derivatives), additionally, we compute second derivatives of basis functions. The device kernel performing this task is similar to the one that calculates basis function values and their gradients at a given grid point in the sense of accessing basis and primitive function IDs. The grid weight gradient calculation is only required for grid points whose weight is not equal to unity. Therefore, we prune our grid for the third time removing all points that do not meet this criterion. The resulting grid points are packed without dummy points and uploaded to the GPU, and the appropriate device kernel is called to compute the gradients. The calculated gradient contribution from each thread is added to the total gradient vector using the CUDA atomic add function, consistent with our existing ERI gradient scheme.

3.8. CPU Analog of the GPU Implementation. In order to perform a fair comparison with our GPU capable DFT implementation, we implemented a parallel CPU version using MPI. In this version, we make use of the standard LIBXC code base and omit the LIBXC dry run from our workflow (step 2 in Figure 2). Furthermore, the grid generation and octree run are performed in serial, but the weight computation, prescreening, and preparation of basis and primitive functions lists are all performed in parallel. Moreover, the packing of grid points (step 3g in Figure 2) is omitted. Computation of electron densities, XC energy, potential, and gradients (steps 4−6 in Figure 2) are performed in parallel. More specifically, once the grid operations are completed, partitioned bins are distributed among slave CPU tasks. The corresponding grid weights and basis and primitive function lists are also broadcast. All CPU tasks retrieve basis and primitive function IDs using locators as discussed in section 3.5, but with the block index now replaced by the bin index. When computing the XC energy and matrix elements of the potential, each CPU task initializes LIBXC through a standard Fortran 90 interface and computes the required functional values. The slave CPU tasks then send the computed energy and Kohn−Sham matrix contributions to the master CPU task. The implementation of the XC gradient scheme is very similar to the above.

4. BENCHMARK RESULTS AND DISCUSSION

4.1. Benchmarking the GPU Implementation. We now present the benchmarking results of our DFT implementation. First, the performance of the QUICK GPU version is compared against the GPU version of GAMESS.[24,25,40] Then a similar comparison between the QUICK CPU and GPU versions is performed. For the former test, we obtained a precompiled copy of GAMESS (the 17.09-r2-libcchem version) in a singularity image from the NVIDIA GPU cloud webpage. In order to make a fair comparison, the QUICK code was compiled using the comparable GNU and CUDA compilers with optimization level 2 (-O2). Note that performance trade-off between running a CPU/GPU applica-

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

Table 2. Comparison of the Accuracy and Performance of B3LYP Calculations Performed Using QUICK CPU (serial/single core) and GPU Versions[a]

molecule (atom number)	basis set (function number)	second SCF iteration (s)			XC (s)			total energy (au)	
		CPU	GPU	speedup	CPU	GPU	speedup	CPU	DGPU[b]
$(H_2O)_{32}$ (96)	6-311G (608)	153.9	2.9	54	44.5	0.5	82	−2444.98724960	-1.09×10^{-7}
	6-311G** (992)	367.2	7.7	48	68.7	1.0	70	−2445.85046773	-1.12×10^{-7}
	cc-pVDZ (800)	428.3	5.5	78	53.9	0.8	67	−2445.00925604	-1.09×10^{-7}
Taxol (110)	6-311G (940)	687.0	14.9	46	131.0	1.7	75	−2927.02879410	-6.20×10^{-8}
	6-311G** (1453)	2314.4	44.2	52	223.2	3.4	66	−2927.98372502	-3.10×10^{-8}
	cc-pVDZ (1160)	3537.4	38.4	92	198.7	3.1	63	−2927.38471110	-5.10×10^{-8}
valinomycin (168)	6-311G (1284)	1451.1	36.9	39	225.9	2.9	78	−3793.79224954	-1.28×10^{-7}
	6-311G** (2022)	4628.3	105.6	44	364.2	5.2	69	−3795.12274932	-1.10×10^{-7}
	cc-pVDZ (1620)	6625.2	81.4	81	304.8	4.7	65	−3794.17889687	-9.20×10^{-8}
3_{10}-helix acetyl$(Ala)_{18}NH_2$ (189)	6-311G (1507)	1680.8	43.7	38	211.3	2.7	77	−4660.64280944	-7.40×10^{-8}
	6-311G** (2356)	5914.4	134.4	44	356.1	5.2	68	−4662.21706599	-5.60×10^{-8}
	cc-pVDZ (1885)	8789.8	108.9	81	309.0	4.8	65	−4661.20248530	-6.80×10^{-8}
α-helix acetyl$(Ala)_{18}NH_2$ (189)	6-311G (1507)	1881.6	52.3	36	257.2	3.3	78	−4660.97909929	-1.55×10^{-7}
	6-311G** (2356)	6186.0	157.4	39	408.9	5.9	69	−4662.53814845	-2.10×10^{-7}
	cc-pVDZ (1885)	8907.1	119.8	74	355.9	5.4	65	−4661.54162676	3.10×10^{-8}
β-strand acetyl$(Ala)_{18}NH_2$ (189)	6-311G (1507)	921.2	24.5	38	100.3	1.3	76	−4660.89912485	-6.90×10^{-8}
	6-311G** (2356)	3451.9	82.1	42	187.8	2.8	67	−4662.48158890	-1.45×10^{-7}
	cc-pVDZ (1885)	5094.9	63.4	80	160.6	2.5	63	−4661.46843955	2.09×10^{-6}
α-conotoxin MII (PDB ID 1M2C) (220)	6-31G* (1964)	4627.1	108.3	43	373.6	4.9	77	−7142.85690266	-6.04×10^{-6}
	6-31G** (2276)	5601.1	138.1	41	407.1	6.2	66	−7143.06700681	-5.34×10^{-6}
	6-311G (1852)	3390.0	103.5	33	424.9	5.5	78	−7142.57919438	-7.30×10^{-8}
olestra (453)	6-31G* (3181)	5787.1	143.7	40	258.9	4.1	63	−7540.95874486	-2.14×10^{-6}
	6-31G** (4015)	9881.3	277.6	36	310.1	4.5	70	−7541.35299607	-2.19×10^{-6}
	6-311G (3109)	5511.5	164.3	34	276.0	3.7	74	−7540.71669525	-5.85×10^{-7}

[a]NVIDIA V100-SXM2 GPU, Intel Xeon (R) Gold 6148 CPU. [b]DGPU energy column shows the deviation of the energy with respect to the corresponding CPU calculation.

Table 3. Comparison of gradient Times between QUICK CPU (Serial/Single Core) and GPU Versions[a]

molecule (atom number)	basis set (function number)	ERI gradients (s)			XC gradients (s)		
		CPU	GPU	speedup	CPU	GPU	speedup
$(H_2O)_{32}$ (96)	6-31G (647)	218.3	5.3	41	1196.7	3.7	325
	6-31G** (992)	1015.1	23.7	43	1261.3	5.4	235
Taxol (110)	6-31G (647)	1676.5	38.2	44	1893.4	9.4	201
	6-31G** (1160)	7479.7	149.1	50	2094.2	14.7	142
valinomycin (168)	6-31G (882)	3339.8	85.0	39	6156.7	19.4	318
	6-31G** (1620)	14579.7	319.3	46	6457.1	27.0	239
3_{10}-helix acetyl$(Ala)_{18}NH_2$ (189)	6-31G (608)	4281.0	97.6	44	8611.6	20.6	418
	6-31G** (992)	20009.4	395.9	51	8907.9	27.6	323
α-helix acetyl$(Ala)_{18}NH_2$ (189)	6-31G (608)	4592.9	109.7	42	8777.5	23.0	381
	6-31G** (992)	20114.2	422.8	48	9073.9	32.5	279
β-strand acetyl$(Ala)_{18}NH_2$ (189)	6-31G (608)	2307.7	45.7	51	8398.9	16.5	509
	6-31G** (992)	11247.6	187.8	60	8601.1	22.4	383
α-conotoxin MII (PDB ID 1M2C) (220)	6-31G (1268)	8371.3	232.6	36	13868.9	17.8	780
	6-31G** (2276)	33798.1	813.3	42	14349.2	28.8	498

[a]B3LYP functional is employed in all calculations. NVIDIA V100-SXM2 GPU, Intel Xeon (R) Gold 6148 CPU.

tion native or through a container is reported to be negligible[48,49] and we assume that the container overhead has no significant impact on our GAMESS timings. The selected test cases include morphine, $(glycine)_{12}$, and valinomycin BLYP[50−52] gradient calculations using the 6-31G, 6-31G*, and 6-31G** basis sets. In GAMESS input files, the SG-1 grid system and direct SCF were requested, and the density matrix convergence threshold was set to 10^{-8}. Default values were used for all the other options. All tests were carried out on a NVIDIA Volta V100-SXM2 GPU (32 GB) accompanied by Intel Xeon (R) Gold 6138 CPU (2.10 GHz) with 190 GB memory.

The performance comparison between QUICK and GAMESS suggests that the former is significantly faster than the latter (see Table 1). For ERI gradients, the observed speedup is roughly 2−5-fold suggesting that our ERI engine is more efficient, in particular for basis sets with higher angular momentum quantum numbers. Furthermore, QUICK displays higher speedups (~60- or 80-fold in some cases) for the XC gradients; however, this result has to be interpreted carefully

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

Table 4. Comparison of Accuracy and Performance between LIBXC CPU (Serial/Single Core) and GPU Versions for Taxol (110 Atoms) with the 6-31G Basis Set (647 Contracted Basis Functions)[a]

		XC (s)			total energy (au)	
functional	number of SCF iterations	CPU	GPU	speedup	CPU	GPU
BLYP-native	21	491.2	21.7	23	−2925.30524021	-3.65×10^{-7}
BLYP	21	486.6	21.7	22	−2925.30524021	-3.65×10^{-7}
B3LYP-native	20	398.4	21.7	18	−2926.28595424	-2.41×10^{-7}
B3LYP	17	399.1	17.6	23	−2926.28595424	-2.41×10^{-7}
PBE0[54,55]	17	397.6	17.6	23	−2923.03725363	-1.50×10^{-8}
B3PW91[56]	17	466.2	20.8	22	−2925.19518104	-1.10×10^{-8}
XVWN[57,58]	21	498.3	21.7	23	−2902.07631292	-2.70×10^{-8}
LP_A[59]	21	498.7	21.7	23	−2930.72703131	-2.80×10^{-8}

[a]NVIDIA V100-SXM2 GPU, Intel Xeon (R) Gold 6148 CPU.

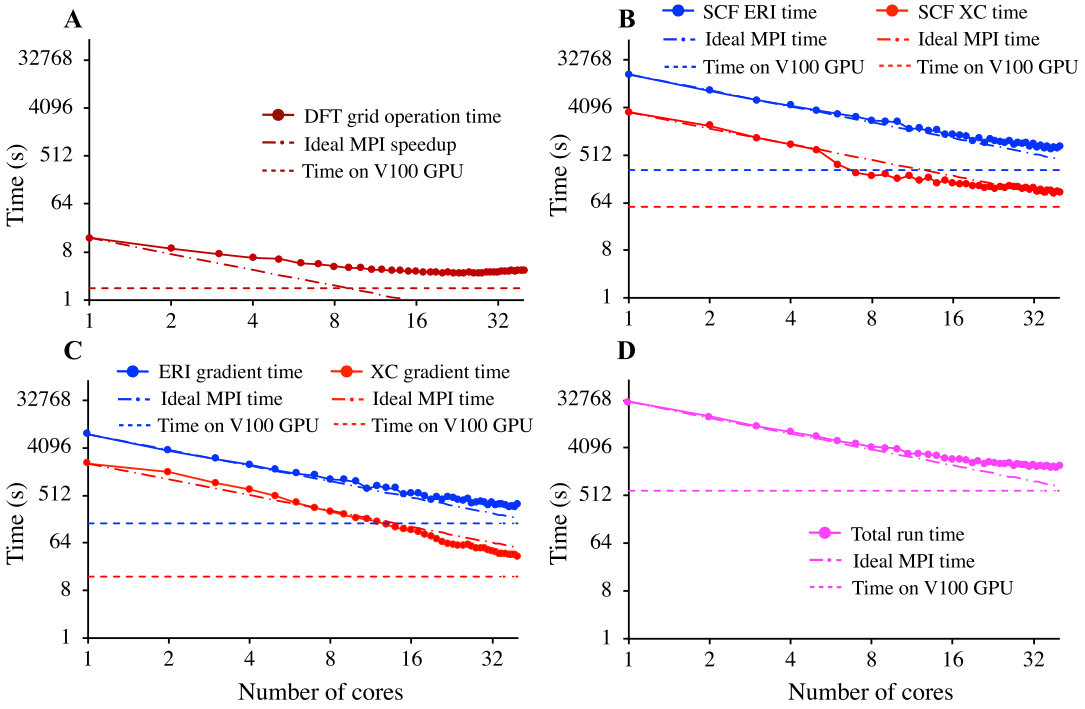


Figure 3. Comparison of performance between QUICK GPU and MPI implementations using Taxol (110 atoms) gradient calculations performed at B3LYP/6-31G** (1160 contracted basis functions) level of theory. Comparison of DFT grid operation time (A), total ERI and XC correlation times for 17 SCF iterations (B), ERI and XC gradient times (C), and total run times (D) between GPU and MPI versions. NVIDIA V100-SXM2 GPU accompanied by 2 Intel Xeon (R) Gold 6148 CPUs.

since the XC implementation of GAMESS appears to be not GPU capable. Note that in larger test cases, the GAMESS XC gradient time surpasses the ERI gradient time. Based on the average SCF times, the QUICK SCF scheme also outperforms GAMESS. Again, slow performance of the latter may be partially attributed to its CPU based XC implementation. A comparison of separate timings for ERI, XC energy, and potential calculations during each SCF iteration was not possible since timings for these individual tasks were not reported by GAMESS. Nevertheless, HF single point calculations carried out for the same systems (see Table S1) suggest that QUICK ERI calculations are faster, consistent with the gradient results documented above.

We now compare the accuracy and performance between the QUICK CPU and GPU versions using a series of B3LYP[53] energy calculations. As is apparent from Table 2, energies computed by the QUICK GPU version agree with the CPU version up to 10^{-6} au or better. Furthermore, the GPU version of the XC implementation delivers at least a 60-fold speedup over the serial CPU version (also see Table S2). In both versions, less than 30% of the SCF step time is spent on computing the XC energy and contribution to Kohn−Sham matrix. This suggests that ERI calculation remains the performance bottleneck of QUICK DFT energy calculations. As a result, the speedup observed for the SCF iteration in most cases is somewhat lower than the XC speedup.

In Table 3, we report a comparison of B3LYP/6-31G and B3LYP/6-31G** gradient calculation times between the QUICK CPU and GPU versions. The selected test cases are a subset of the molecular systems chosen for the SCF tests. The key observations from Table 3 include (1) the GPU version delivers significant speedups for both ERI and XC gradient calculations, (2) the speedup observed for ERI gradients resembles the ones realized for energy calculation (~50 times), and (3) the speedup delivered for XC gradients is in few hundreds range (~100−800 times). Similar speedups

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

observed for ERI energy and gradient calculations can be explained by the fact that their implementations are similar. Indeed, computing the gradients of a given basis function involves the cost of computing the value of the higher angular momentum basis function. Understanding the details of the observed speedups for XC energy and gradient computations requires further investigation. As mentioned in section 3.7, we implemented XC energy gradient and grid weight gradient calculations in two separate kernels. Careful examination of kernel run times suggests that the former consumes more than 90% of the gradient time and gains a better speedup on a GPU. However, the structure of this kernel is similar to that of the XC energy with the exception of computing second derivatives. Therefore, the observed higher speedup must be due to the inefficiency of computing the second derivatives on the CPU.

4.2. Performance of LIBXC Functionals. As mentioned above, we have integrated the original and modified LIBXC library versions into QUICK CPU and GPU codes. It is important to document the performance of these functionals within our XC scheme. In Table 4, we report a series of Taxol energy calculations at the DFT/6-31G level of theory using various functionals. The selected functionals include hand-coded (native) BLYP and B3LYP and their LIBXC versions and, additionally, representative LDA, GGA, and hybrid-GGA functionals. Comparison of native BLYP and B3LYP total energies against the corresponding LIBXC versions suggests that the latter are as accurate as the former on both CPU and GPU platforms. Furthermore, the reported CPU and GPU XC times remain very similar, and the acceleration delivered by the GPU remains substantially the same. In fact, the ca. 20-fold speedup realized on the GPU platform is common for all LDA and GGA functionals. Note that computing the value of density functionals is relatively inexpensive with respect to other operations performed in XC energy or gradient calculations, and such a similar speedup is expected. Finally, the fact that we maintain a similar speedup among native and LIBXC functionals suggests that minor modifications performed to density functional source code in the GPU version has not introduced performance penalties.

4.3. Comparison of GPU versus Parallel CPU Implementations. We now compare the performance between GPU and parallel CPU (MPI) implementations using Taxol gradient calculations at the B3LYP/6-31G** level of theory. The selected platform for testing is a single computer node equipped with a NVIDIA V100-SXM2 GPU (32 GB) and 40 Intel Xeon (R) Gold 6148 cores (2 sockets with 20 cores on each, 2.40 GHz clock speed) with 367 GB memory. The QUICK MPI version was compiled using the GNU 7.2.0 compiler and MPICH 3.2.1. The GPU version was compiled using the same GNU compiler with CUDA version 10.0.130. As is apparent from Figure 3A, the DFT grid operation time for the same calculation is reduced by ca. 4 times (14.8 vs 3.6 s) when going from single to 40 cores in the MPI version. The realized speedup can be mainly attributed to parallelized grid weight computation, grid pruning, and basis function prescreening. The corresponding V100 GPU time for the same task is 1.7 s with approximately 1 s spent on the CPU based octree run. We tested the necessity of implementing the octree algorithm on the GPU by measuring the time to partition numerical grid points of larger molecular systems. The results suggest that our existing CPU implementation is sufficiently efficient to handle such systems. For example, partitioning grid points of olestra (453 atoms, ∼4.4 million grid points) only took 3.0 s, while computing grid weights and prescreening consumed 12.7 and 0.6 s, respectively.

In Figure 3B, we report the total time spent by the ERI and XC calculations during 19 SCF iterations. The maximum speedup observed for the former and latter in the MPI version are ca. 23- and 33-fold, respectively. The speedups from the GPU version for the same tasks are ca. 65- and 62-fold. For ERI gradient calculations (see Figure 3C), the speedup gained from the MPI version is similar to that of the SCF (ca. 22-fold), but significantly less than the corresponding GPU speedup (ca. 50-fold). Furthermore, the MPI version delivers about 59-fold speedup for the XC gradient calculations; however, this is below the acceleration achieved by a V100 GPU (ca. 142-fold). Finally, as evident from Figure 3D, the total speedup gained from the GPU is two times over using 40 cores. It is also important to comment on the linearity of our MPI plots in Figure 3B,C. Both ERI SCF and gradient calculations scale well with the number of cores and almost achieve the ideal MPI speedup since we distribute atomic shells among MPI tasks (CPU cores). However, in the XC implementation, we distribute bins containing varying number of grid points. Therefore, the workload is not optimally balanced and plots of the XC MPI timings display a nonregular speedup (not linear) unlike in the ERI case. The linear scaling computation of XC contributions on CPU platforms has been documented in past.[60,61] Such work has also reported nonlinear and superlinear speedups.

4.4. Performance of QUICK on Different GPU Architectures. For all the aforementioned benchmarks, we have used a NVIDIA V100 data center GPU. It is also necessary to document how the current QUICK GPU implementation performs on other available devices. In Table 5, we report the performance of several important kernels on different workstation GPUs (V100, Titan V, and P100) and a gaming GPU (RTX2080Ti). The selected test case for this purpose is the Taxol gradient calculation at the B3LYP/6-31G level of theory. As is apparent from Table 5, all kernels display their best performance on the V100 GPU. Surprisingly, ERI kernels show their slowest times on the P100 rather than the gaming GPU, suggesting that their performance is not limited by double precision (FP64) operations. Note that the P100 GPU has a higher FP64 capability in comparison to the RTX2080Ti.[62,63] A detailed examination of the two kernels revealed that their performance is bound by high register usage and memory bandwidth. In contrast, XC kernels displayed

Table 5. Comparison of the Performance of Different Kernels Using Different GPU Architectures[a]

	total time for each task (s)			
task	V100 (32 GB)	Titan V (12 GB)	P100 (16 GB)	RTX2080Ti (11 GB)
SCF ERI[b]	80.4	107.6	159.2	125.2
SCF XC[b]	22.7	26.1	59.8	114.3
ERI gradients	38.0	40.7	66.9	55.1
XC gradients	6.9	7.9	8.5	13.4

[a]The selected tests case is Taxol gradient calculation at B3LYP/6-31G** (1160 contracted basis functions) level of theory. Reported kernel times were measured using the NVIDIA visual profiler available in the CUDA toolkit. [b]Reported time is the summation over 22 SCF cycles.

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

their highest timings on the RTX2080Ti GPU, and therefore, these are bound by FP64 operations. Further examination of the XC energy gradient kernel indicated that its performance is also limited by atomic operations. Overall, excellent performance is achieved on both data center GPUs with Pascal and Volta architectures and gaming GPUs with Turing architecture.

5. CONCLUSIONS

We have reported the details of the MPI parallel CPU and the GPU enabled DFT implementation of the QUICK quantum chemical package. Our implementation consists of features such as octree-based grid point partitioning, GPU assisted grid pruning and basis and primitive function prescreening, and fully GPU enabled XC energy and gradient computations. The CPU and GPU versions mainly differ from each other as follows. In the GPU version, device kernels are used for computing grid weights, preparing basis and primitive function lists and grid pruning based on primitive function values, computing electron densities, calculation of functional values and their derivatives, computing XC energy, and assembling matrix elements of the XC potential into the Kohn−Sham matrix. Computing XC energy gradients, grid pruning based on the grid weights, and computing grid weight gradients are also done using device kernels. In the CPU version, all these steps are performed using host functions.

Performance comparison with the GAMESS GPU version demonstrates that DFT calculations with QUICK are significantly faster. The accelerations observed for the XC energy and gradient computation in the QUICK GPU version with respect to the serial CPU version are impressive. The speedups realized on a V100 GPU for the former and latter are approximately 60−80-fold and 100−800-fold, respectively. Such speedups are out of reach with the MPI parallel CPU version even if one uses 40 cores in parallel. The recommended device for the latest QUICK version (v20.03) is the NVIDIA V100 data center GPU, but the code runs very well also on gaming GPUs.

The profiling of ERI and XC kernels has shown that there exists room for further performance improvement. In the former context, reimplementing large ERI kernels into smaller kernels may be a viable strategy. This is expected to reduce the register pressure and enhance the performance of the ERI engine. The performance of XC kernels on gaming GPUs may be improved by implementing a mixed precision scheme. Additionally, strategies such as storing the gradient vector in shared memory and latency hiding may be helpful to reduce the computational cost associated with atomic operations in the XC energy gradient kernel. Performance of XC kernels on data center GPUs may be further improved by increasing the kernel occupancy, which is currently about 9% on Volta type GPUs.

Currently, we are integrating QUICK as a library into the AMBER molecular dynamics package[64] to enable fully GPU enabled quantum mechanics/molecular mechanics (QM/MM) simulations. Furthermore, the current version is incapable of using more than one GPU and to this end, we are developing a multi GPU version. Finally, QUICK version 20.03 can be freely downloaded from http://www.merzgroup.org/quick.html under the Mozilla public license.

ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge at https://pubs.acs.org/doi/10.1021/acs.jctc.0c00290.

Grid point packing and retrieval of basis and primitive function indices, comparison of HF performance between the QUICK vs GAMESS GPU versions, and comparison of B3LYP performance between the QUICK CPU and GPU versions using last SCF iteration (PDF)

Cartesian coordinates of the test molecules (ZIP)

AUTHOR INFORMATION

Corresponding Authors

Andreas W. Götz − *San Diego Supercomputer Center, University of California, San Diego, La Jolla, California 92093-0505, United States;* orcid.org/0000-0002-8048-6906; Email: agoetz@sdsc.edu

Kenneth M. Merz, Jr. − *Department of Chemistry and Department of Biochemistry and Molecular Biology, Michigan State University, East Lansing, Michigan 48824-1322, United States;* orcid.org/0000-0001-9139-5893; Email: merz@chemistry.msu.edu

Authors

Madushanka Manathunga − *Department of Chemistry and Department of Biochemistry and Molecular Biology, Michigan State University, East Lansing, Michigan 48824-1322, United States;* orcid.org/0000-0002-3594-8112

Yipu Miao − *Facebook, Menlo Park, California 94025, United States*

Dawei Mu − *National Center for Supercomputing Applications, University of Illinois at Urbana−Champaign, Urbana, Illinois 61801, United States*

Complete contact information is available at: https://pubs.acs.org/10.1021/acs.jctc.0c00290

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

We thank Jonathan Lefman and Peng Wang from NVIDIA for their useful comments on technical aspects of our GPU code. M.M. and K.M. are grateful to the Department of Chemistry and Biochemistry and high performance computer center (iCER HPCC) at the Michigan State University. M.M. and A.G. thank San Diego Supercomputer Center for granted computer time. This research was supported by the National Science Foundation Grant OAC-1835144. This work also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by the National Science Foundation (Grant Number ACI-1053575, resources at the San Diego Supercomputer Center through award TG-CHE130010 to A.G.).

REFERENCES

(1) Nobile, M. S.; Cazzaniga, P.; Tangherloni, A.; Besozzi, D. Graphics Processing Units in Bioinformatics, Computational Biology and Systems Biology. *Briefings Bioinf.* **2016**, *18*, 870−885.

(2) NVIDIA. Seeing Gravity in Real-Time with Deep Learning https://images.nvidia.com/content/pdf/ncsa-gravity-group-success-story.pdf (accessed Feb 25, 2020).

(3) NVIDIA. CUDA C++ Programming Guide https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (accessed Feb 25, 2020).
(4) Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. Accelerating Molecular Dynamic Simulation on Graphics Processing Units. *J. Comput. Chem.* **2009**, *30*, 864−872.
(5) Götz, A. W.; Williamson, M. J.; Xu, D.; Poole, D.; Le Grand, S.; Walker, R. C. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 1. Generalized Born. *J. Chem. Theory Comput.* **2012**, *8*, 1542−1555.
(6) Salomon-Ferrer, R.; Götz, A. W.; Poole, D.; Le Grand, S.; Walker, R. C. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 2. Explicit Solvent Particle Mesh Ewald. *J. Chem. Theory Comput.* **2013**, *9*, 3878−3888.
(7) Biagini, T.; Petrizzelli, F.; Truglio, M.; Cespa, R.; Barbieri, A.; Capocefalo, D.; Castellana, S.; Tevy, M. F.; Carella, M.; Mazza, T. Are Gaming-Enabled Graphic Processing Unit Cards Convenient for Molecular Dynamics Simulation? *Evol. Bioinf. Online* **2019**, *15*, 117693431985014.
(8) Lee, T. S.; Cerutti, D. S.; Mermelstein, D.; Lin, C.; Legrand, S.; Giese, T. J.; Roitberg, A.; Case, D. A.; Walker, R. C.; York, D. M. GPU-Accelerated Molecular Dynamics and Free Energy Methods in Amber18: Performance Enhancements and New Features. *J. Chem. Inf. Model.* **2018**, *58*, 2043−2050.
(9) Harvey, M. J.; Giupponi, G.; De Fabritiis, G. ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *J. Chem. Theory Comput.* **2009**, *5*, 1632−1639.
(10) Le Grand, S.; Götz, A. W.; Walker, R. C. SPFP: Speed without Compromise - A Mixed Precision Model for GPU Accelerated Molecular Dynamics Simulations. *Comput. Phys. Commun.* **2013**, *184*, 374−380.
(11) Kutzner, C.; Páll, S.; Fechner, M.; Esztermann, A.; De Groot, B. L.; Grubmüller, H. Best Bang for Your Buck: GPU Nodes for GROMACS Biomolecular Simulations. *J. Comput. Chem.* **2015**, *36*, 1990−2008.
(12) Eastman, P.; Swails, J.; Chodera, J. D.; McGibbon, R. T.; Zhao, Y.; Beauchamp, K. A.; Wang, L. P.; Simmonett, A. C.; Harrigan, M. P.; Stern, C. D.; et al. OpenMM 7: Rapid Development of High Performance Algorithms for Molecular Dynamics. *PLoS Comput. Biol.* **2017**, *13*, No. e1005659.
(13) Ufimtsev, I. S.; Martínez, T. J. Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation. *J. Chem. Theory Comput.* **2008**, *4*, 222−231.
(14) Ufimtsev, I. S.; Martinez, T. J. Quantum Chemistry on Graphical Processing Units. 2. Direct Self-Consistent-Field Implementation. *J. Chem. Theory Comput.* **2009**, *5*, 1004−1015.
(15) Fales, B. S.; Levine, B. G. Nanoscale Multireference Quantum Chemistry: Full Configuration Interaction on Graphical Processing Units. *J. Chem. Theory Comput.* **2015**, *11*, 4708−4716.
(16) Asadchev, A.; Gordon, M. S. Fast and Flexible Coupled Cluster Implementation. *J. Chem. Theory Comput.* **2013**, *9*, 3385−3392.
(17) DePrince, A. E.; Hammond, J. R. Coupled Cluster Theory on Graphics Processing Units I. The Coupled Cluster Doubles Method. *J. Chem. Theory Comput.* **2011**, *7*, 1287−1295.
(18) Luehr, N.; Ufimtsev, I. S.; Martínez, T. J. Dynamic Precision for Electron Repulsion Integral Evaluation on Graphical Processing Units (GPUs). *J. Chem. Theory Comput.* **2011**, *7*, 949−954.
(19) Liu, F.; Sanchez, D. M.; Kulik, H. J.; Martínez, T. J. Exploiting Graphical Processing Units to Enable Quantum Chemistry Calculation of Large Solvated Molecules with Conductor-like Polarizable Continuum Models. *Int. J. Quantum Chem.* **2019**, *119*, No. e25760.
(20) Götz, A. W.; Wölfle, T.; Walker, R. C. Quantum Chemistry on Graphics Processing Units. *Annu. Rep. Comput. Chem.* **2010**, *6*, 21−35.
(21) Walker, R. C.; Götz, A. W. In *Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics*; Walker, R. C., Götz, A. W., Eds.; Wiley: Chichester, U.K., 2016.
(22) Andrade, X.; Aspuru-Guzik, A. Real-Space Density Functional Theory on Graphical Processing Units: Computational Approach and Comparison to Gaussian Basis Set Methods. *J. Chem. Theory Comput.* **2013**, *9*, 4360−4373.
(23) Ufimtsev, I. S.; Martinez, T. J. Quantum Chemistry on Graphical Processing Units. 3. Analytical Energy Gradients, Geometry Optimization, and First Principles Molecular Dynamics. *J. Chem. Theory Comput.* **2009**, *5*, 2619−2628.
(24) Asadchev, A.; Allada, V.; Felder, J.; Bode, B. M.; Gordon, M. S.; Windus, T. L. Uncontracted Rys Quadrature Implementation of up to G Functions on Graphical Processing Units. *J. Chem. Theory Comput.* **2010**, *6*, 696−704.
(25) Asadchev, A.; Gordon, M. S. New Multithreaded Hybrid CPU/GPU Approach to Hartree-Fock. *J. Chem. Theory Comput.* **2012**, *8*, 4166−4176.
(26) Yasuda, K. Two-Electron Integral Evaluation on the Graphics Processor Unit. *J. Comput. Chem.* **2008**, *29*, 334−342.
(27) Miao, Y.; Merz, K. M. Acceleration of Electron Repulsion Integral Evaluation on Graphics Processing Units via Use of Recurrence Relations. *J. Chem. Theory Comput.* **2013**, *9*, 965−976.
(28) Miao, Y.; Merz, K. M. Acceleration of High Angular Momentum Electron Repulsion Integrals and Integral Derivatives on Graphics Processing Units. *J. Chem. Theory Comput.* **2015**, *11*, 1449−1462.
(29) Hohenstein, E. G.; Luehr, N.; Ufimtsev, I. S.; Martínez, T. J. An Atomic Orbital-Based Formulation of the Complete Active Space Self-Consistent Field Method on Graphical Processing Units. *J. Chem. Phys.* **2015**, *142*, 224103.
(30) Fales, B. S.; Martínez, T. J. Efficient Treatment of Large Active Spaces through Multi-GPU Parallel Implementation of Direct Configuration Interaction. *J. Chem. Theory Comput.* **2020**, *16*, 1586.
(31) Isborn, C. M.; Luehr, N.; Ufimtsev, I. S.; Martínez, T. J. Excited-State Electronic Structure with Configuration Interaction Singles and Tamm-Dancoff Time-Dependent Density Functional Theory on Graphical Processing Units. *J. Chem. Theory Comput.* **2011**, *7*, 1814−1823.
(32) Obara, S.; Saika, A. Efficient Recursive Computation of Molecular Integrals over Cartesian Gaussian Functions. *J. Chem. Phys.* **1986**, *84*, 3963−3974.
(33) Head-Gordon, M.; Pople, J. A. A Method for Two-electron Gaussian Integral and Integral Derivative Evaluation Using Recurrence Relations. *J. Chem. Phys.* **1988**, *89*, 5777−5786.
(34) Jones, R. O. Density Functional Theory: Its Origins, Rise to Prominence, and Future. *Rev. Mod. Phys.* **2015**, *87*, 897.
(35) Yasuda, K. Accelerating Density Functional Calculations with Graphics Processing Unit. *J. Chem. Theory Comput.* **2008**, *4*, 1230−1236.
(36) Luehr, N.; Sisto, A.; Martinez, T. J. Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics. In *Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics*; Walker, R. C., Götz, A. W., Eds.; John Wiley & Sons, Ltd: Chichester, UK, 2016; pp 1−342.
(37) Nitsche, M. A.; Ferreria, M.; Mocskos, E. E.; González Lebrero, M. C. GPU Accelerated Implementation of Density Functional Theory for Hybrid QM/MM Simulations. *J. Chem. Theory Comput.* **2014**, *10*, 959−967.
(38) Pople, J. A.; Gill, P. M. W.; Johnson, B. G. Kohn—Sham Density-Functional Theory within a Finite Basis Set. *Chem. Phys. Lett.* **1992**, *199*, 557−560.
(39) Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (accessed Mar 17, 2020).
(40) Schmidt, M. W.; Baldridge, K. K.; Boatz, J. A.; Elbert, S. T.; Gordon, M. S.; Jensen, J. H.; Koseki, S.; Matsunaga, N.; Nguyen, K. A.; Su, S.; et al. General Atomic and Molecular Electronic Structure System. *J. Comput. Chem.* **1993**, *14*, 1347−1363.

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326

(41) NVIDIA. NVIDIA Tesla V100 GPU Architecture https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (accessed Feb 25, 2020).
(42) Gill, P. M. W.; Johnson, B. G.; Pople, J. A. A Standard Grid for Density Functional Calculations. *Chem. Phys. Lett.* **1993**, *209*, 506−512.
(43) Stratmann, R. E.; Scuseria, G. E.; Frisch, M. J. Achieving Linear Scaling in Exchange-Correlation Density Functional Quadratures. *Chem. Phys. Lett.* **1996**, *257*, 213−223.
(44) Density Functional Repository http://www.cse.scitech.ac.uk/ccg/dft/ (accessed Feb 28, 2020).
(45) Ekström, U.; Visscher, L.; Bast, R.; Thorvaldsen, A. J.; Ruud, K. Arbitrary-Order Density Functional Response Theory from Automatic Differentiation. *J. Chem. Theory Comput.* **2010**, *6*, 1971−1980.
(46) Lehtola, S.; Steigemann, C.; Oliveira, M. J. T.; Marques, M. A. L. Recent Developments in LIBXC — A Comprehensive Library of Functionals for Density Functional Theory. *SoftwareX* **2018**, *7*, 1−5.
(47) Lapillonne, X.; Fuhrer, O. Using Compiler Directives to Port Large Scientific Applications to GPUs: An Example from Atmospheric Science. *Parallel Process. Lett.* **2014**, *24*, 1450003.
(48) Xu, P.; Shi, S.; Chu, X. Performance Evaluation of Deep Learning Tools in Docker Containers. In *International Conference on Big Data Computing and Communications*; Institute of Electrical and Electronics Engineers Inc., 2017; pp 395−403.
(49) Torrez, A.; Priedhorsky, R.; Randles, T. HPC Container Runtime Performance Overhead: At First Order, There Is None https://sc19.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost227.html (accessed Feb 25, 2020).
(50) Becke, A. D. Density-Functional Exchange-Energy Approximation with Correct Asymptotic Behavior. *Phys. Rev. A: At., Mol., Opt. Phys.* **1988**, *38*, 3098−3100.
(51) Lee, C.; Yang, W.; Parr, R. G. Development of the Colle-Salvetti Correlation-Energy Formula into a Functional of the Electron Density. *Phys. Rev. B: Condens. Matter Mater. Phys.* **1988**, *37*, 785−789.
(52) Miehlich, B.; Savin, A.; Stoll, H.; Preuss, H. Results Obtained with the Correlation Energy Density Functionals of Becke and Lee, Yang and Parr. *Chem. Phys. Lett.* **1989**, *157*, 200−206.
(53) Stephens, P. J.; Devlin, F. J.; Chabalowski, C. F.; Frisch, M. J. Ab Initio Calculation of Vibrational Absorption and Circular Dichroism Spectra Using Density Functional Force Fields. *J. Phys. Chem.* **1994**, *98*, 11623−11627.
(54) Adamo, C.; Barone, V. Toward Reliable Density Functional Methods without Adjustable Parameters: The PBE0Model. *J. Chem. Phys.* **1999**, *110*, 6158−6170.
(55) Ernzerhof, M.; Scuseria, G. E. Assessment of the Perdew-Burke-Ernzerhof Exchange-Correlation Functional. *J. Chem. Phys.* **1999**, *110*, 5029−5036.
(56) Becke, A. D. Density-Functional Thermochemistry. III. The Role of Exact Exchange. *J. Chem. Phys.* **1993**, *98*, 5648−5652.
(57) Dirac, P. A. M. Note on Exchange Phenomena in the Thomas Atom. *Math. Proc. Cambridge Philos. Soc.* **1930**, *26*, 376−385.
(58) Vosko, S. H.; Wilk, L.; Nusair, M. Accurate Spin-Dependent Electron Liquid Correlation Energies for Local Spin Density Calculations: A Critical Analysis. *Can. J. Phys.* **1980**, *58*, 1200−1211.
(59) Lee, C.; Parr, R. G. Exchange-Correlation Functional for Atoms and Molecules. *Phys. Rev. A: At., Mol., Opt. Phys.* **1990**, *42*, 193−200.
(60) Challacombe, M. Linear Scaling Computation of the Fock Matrix. V. Hierarchical Cubature for Numerical Integration of the Exchange-Correlation Matrix. *J. Chem. Phys.* **2000**, *113*, 10037−10043.
(61) Gan, C. K.; Challacombe, M. Linear Scaling Computation of the Fock Matrix. VI. Data Parallel Computation of the Exchange-Correlation Matrix. *J. Chem. Phys.* **2003**, *118*, 9128−9135.
(62) NVIDIA. NVIDIA Tesla P100 https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (accessed Mar 17, 2020).
(63) NVIDIA. NVIDIA Turing GPU Architecture https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (accessed Feb 25, 2020).
(64) Case, D. A.; Ben-Shalom, I. Y.; Brozell, S. R.; Cerutti, D. S.; Cheatham, T. E., III; Cruzeiro, V. W. D.; Darden, T. A.; Duke, R. E.; Simmerling, C. L.; et al. *AMBER 2018*; University of California: San Francisco, CA, 2018.

https://dx.doi.org/10.1021/acs.jctc.0c00290
J. Chem. Theory Comput. 2020, 16, 4315−4326