THE JOURNAL OF PHYSICAL CHEMISTRY A
A JOURNAL OF THE AMERICAN CHEMICAL SOCIETY

pubs.acs.org/JPCA
Article

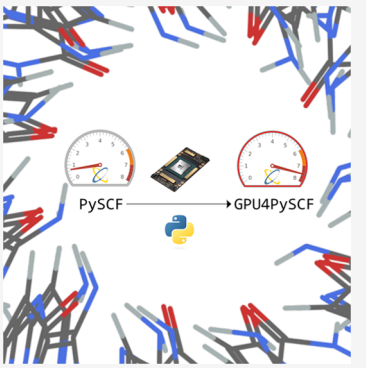
Introducing GPU Acceleration into the Python-Based Simulations of Chemistry Framework

Published as part of The Journal of Physical Chemistry A special issue "Quantum Chemistry Software for Molecules and Materials".

Rui Li,[§] Qiming Sun,[§] Xing Zhang, and Garnet Kin-Lic Chan*

Cite This: J. Phys. Chem. A 2025, 129, 1459−1468
Read Online
ACCESS | Metrics & More | Article Recommendations

ABSTRACT: We introduce the first version of GPU4PySCF, a module that provides GPU acceleration of methods in PySCF. As a core functionality, this provides a GPU implementation of two-electron repulsion integrals (ERIs) for contracted basis sets comprising up to *g* functions using the Rys quadrature. As an illustration of how this can accelerate a quantum chemistry workflow, we describe how to use the ERIs efficiently in the integral-direct Hartree−Fock build and nuclear gradient construction. Benchmark calculations show a significant speedup of 2 orders of magnitude with respect to the multithreaded CPU Hartree−Fock code of PySCF and the performance comparable to other open-source GPU-accelerated quantum chemical packages, including GAMESS and QUICK, on a single NVIDIA A100 GPU.

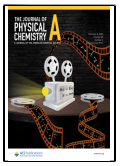


I. INTRODUCTION

The rapid advances in the capabilities of graphics processing units (GPUs) has significantly impacted many fields, including graphics rendering, gaming, and artificial intelligence.[1,2] The massively parallel architecture of GPUs offers drastically more computational throughput than traditional central processing units (CPUs), making them well-suited for computationally intensive tasks such as dense matrix multiplication and tensor contraction.[3] Consequently, GPUs have evolved into powerful tools for scientific computation on high-performance computing (HPC) platforms. For instance, at the National Energy Research Scientific Computing Center, GPUs deliver a maximum compute performance of 119.8 PFLOPS compared to only 11.6 PFLOPS from the associated CPUs.[4] However, leveraging GPUs for substantial performance gains over CPUs typically requires a significant redesign of the underlying algorithms.

In the field of quantum chemistry, GPUs have been extensively explored to accelerate the Hartree−Fock (HF) and density functional theory (DFT) methods. Particular attention has been paid to evaluating two-electron repulsion integrals (ERIs), a key computational primitive, and their subsequent use in the Fock builds of the HF and DFT equations. Over the past 15 years, various GPU algorithms for ERI evaluation and Fock builds have been proposed. Yasuda[5] implemented the first such algorithm, along with the construction of the Coulomb matrix using the J engine method.[6,7] At the same time, Ufimtsev and Martínez[8,9] developed a GPU implementation for the HF method, which included building the full Fock matrix (both Coulomb and exchange matrices), with ERIs evaluated using the McMurchie−Davidson (MD) algorithm.[10] Both implementations were initially limited to Gaussian basis sets containing only *s* and *p* functions, although very recent work from the same group has added support for up to *f* functions[11,12] by using code generation. Later, Asadchev and Gordon[13] developed a Fock build algorithm using the Rys quadrature method[14,15] for ERI evaluation, allowing for the use of uncontracted basis sets with up to *g* functions. In addition, Miao and Merz[16] employed the Head−Gordon−Pople (HGP) algorithm[17] to reduce the number of floating-point operations (FLOPs) required for computing ERIs with contracted basis functions. Recently, Barca et al. introduced a distinct implementation of the HGP algorithm and an improved ERI digestion (the contraction between ERIs and the density matrix to form the Fock matrix) scheme.[18] This was subsequently extended to run on multiple GPUs.[19,20] Barca et al. further combined their HGP-based

Received: August 30, 2024
Revised: December 30, 2024
Accepted: January 2, 2025
Published: January 23, 2025



https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

algorithm in one package, which excels for smaller systems, and Martínez et al.'s MD-based algorithm, which is advantageous for large systems. The hybrid implementation was shown to outperform most previous multi-GPU implementations.[21] Nevertheless, significant performance drops were observed for ERIs involving basis functions with higher angular momenta, such as *d* functions. To address this issue, Asadchev and Valeev[22,23] developed a matrix-based formulation of the MD algorithm, leveraging extensive use of dense matrix multiplication kernels. Their approach achieved significant speedups over the reference CPU implementation, particularly for high angular momentum ERIs, including those involving *i* functions.

In this work, we describe our implementation of four-center ERIs on the GPU within the GPU4PySCF module. As a core computational routine, this was the first feature to be developed. At the time of writing, GPU4PySCF also contains many additional features, including those developed using GPU-accelerated density fitting ERIs[24] (which are computed using an adaptation of the four-center ERI algorithm). However, to limit the scope of this paper to the work of the current set of authors as well as to present the chronological development of the package, this work describes only the algorithm for four-center ERIs and the subsequent Fock build routines that use them.

Our ERI implementation is based on Rys quadrature. One advantage of this technique is that it features a small memory footprint, making it well-suited for mainstream commodity GPUs with limited, fast on-chip memory. Additionally, it offers simple recurrence relations, facilitating straightforward extensions to high angular momentum and ERI derivatives. Within this framework, we utilize several algorithmic optimizations to enhance the performance of both energy and nuclear gradient ERI evaluation with the latest compute unified device architecture (CUDA). The resulting ERI routines support contracted basis sets comprising up to *g* functions.

GPU4PySCF is designed to operate primarily within the Python environment. Consequently, in addition to the custom CUDA kernels for the ERIs, it utilizes NumPy[25]-like packages (such as CuPy[26]) to accelerate the computationally expensive tensor contractions and linear algebra operations on the GPU. The Python-based nature of GPU4PySCF allows for seamless integration with other Python-based workflows, particularly those in machine learning applications. We envision that this choice of ecosystem for quantum chemistry GPU acceleration will allow GPU4PySCF to achieve the same type of interdisciplinary impact that its parent package PySCF has become known for.[27]

The review is organized as follows. Section II provides a brief review of the Rys quadrature method. Sections III and 4 detail our GPU-accelerated Hartree−Fock (HF) implementation, focusing on the algorithms for Fock build and nuclear gradients, respectively. The performance of our method is examined in Section V. Finally, we draw some conclusions and describe our general outlook for GPU4PySCF in Section VI.

II. RYS QUADRATURE METHOD

In the Rys quadrature method,[14,15,28] the six-dimensional ERI is expressed as a product of three two-dimensional (2D) integrals (I_x, I_y, and I_z), evaluated exactly by an *N*-point Gaussian quadrature with weights (w_n) and roots (t_n) of the Rys polynomial:

$$[ab|cd] = \sum_{n}^{N} w_n I_x(t_n) I_y(t_n) I_z(t_n) \tag{1}$$

In eq 1, the ERI is computed for Cartesian primitive Gaussian functions (PGFs)

$$|a] \equiv \phi_a(\mathbf{r}) = (x - A_x)^{a_x}(y - A_y)^{a_y}(z - A_z)^{a_z} e^{-\alpha|\mathbf{r}-\mathbf{A}|^2} \tag{2}$$

which are centered at nuclear positions $\mathbf{A} = (A_x, A_y, A_z)$, with exponents α and angular momenta $a = a_x + a_y + a_z$. The number of quadrature points is related to the angular momenta of the four PGFs as

$$N = \left\lfloor \frac{a + b + c + d}{2} \right\rfloor + 1 \tag{3}$$

The 2D integrals I_x, I_y, and I_z are evaluated for each primitive shell quartet (denoted as $[ab|cd]$, where bold letters indicate a shell of basis functions), using the recurrence and transfer relations (RRs).[14,15] Each of the 2D integral tensors has a size of $(a + 1)(b + 1)(c + 1)(d + 1)$ for each quadrature point. Finally, for contracted Gaussian functions (CGFs), which are linear combinations (with contraction order *K*) of PGFs

$$|i) = \sum_{a}^{K} C_a^i |a] \tag{4}$$

the contracted ERI can be written as

$$(ij|kl) = \sum_{abcd} C_a^i C_b^j C_c^k C_d^l [ab|cd] \tag{5}$$

Modern GPUs offer high computational throughput but often suffer from significant memory latency. They are well-suited for tasks with high arithmetic intensity [defined as the ratio of FLOPs to data movement (in bytes)], such as dense matrix multiplications, where for every number stored in the slow VRAM memory, there are dozens of associated floating-point operations, meaning that the latency can be effectively masked. These kernels are regarded as compute-bound, i.e., the peak FLOP rate of the GPU is the bottleneck for the performance of these kernels. Kernels with low arithmetic intensity, such as matrix addition, are memory-bound and are constrained by the VRAM memory bandwidth. The peak FLOP rate to VRAM memory bandwidth ratio of the GPU defines the boundary between the two types. ERI evaluation using the Rys quadrature method may or may not fall into this compute-bound category of tasks, depending on the feasibility of data caching in fast memory. We can roughly estimate its arithmetic intensity by considering the most computationally expensive step, i.e., eq 1. If no data are cached, the arithmetic intensity is approximately $\frac{3}{16}$FLOP/byte. This intensity is significantly below the peak FLOP rate to memory bandwidth ratio of $\frac{9.7\text{TFLOP/s}}{1.6\text{TB/s}} = 6.1$FLOP/byte for the NVIDIA A100 GPU used in this work. According to the Roofline model,[29] it suggests that the corresponding implementation will be memory-bound and likely inefficient. On the other hand, if the ERIs and 2D integrals can be cached completely, the arithmetic intensity becomes $\frac{3NK}{8}$FLOP/byte (assuming we are storing the results in slow memory). This value can be higher than the previous ratio for large *N* and *K*, indicating a compute-bound character. However, GPUs typically have

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

limited fast memory (e.g., registers and shared memory) and hence require careful algorithmic design to achieve optimal performance.

The Rys quadrature method features a low memory footprint[13] and high data locality,[30] which allows for more effective data caching. For instance, to compute ERIs with $N \leq 3$, the necessary data can almost entirely fit into the registers. Specifically, for the integral class (**pp**|**pp**), the required intermediates (only considering the contracted ERIs and 2D integrals) amount to $3^4 + 3 \times 2^4 = 129$ FP64 words (equivalent to 258 FP32 words). This is just above the maximum register file size allowed for an NVIDIA GPU thread, which is 255 FP32 words (for the microarchitectures since Kepler). However, for larger N values, the data size will inevitably exceed the available resources of fast memory. As a result, the implementation must minimize access to slow memory (e.g., local and global memory). In practice, we incorporate the following designs:

1. For ERIs with small N values, the RRs are unrolled to fully utilize registers.
2. In general cases, we perform ERI digestion before contraction and introduce novel intermediates to minimize global memory access.
3. When computing the nuclear gradients, double contractions between the ERI gradients and the density matrix are performed to directly obtain the energy gradients, thereby avoiding the need to store the Fock matrix gradients.

These are detailed in Sections 3 and 4.

III. FOCK BUILD

Algorithm 1 illustrates the workflow for building the Fock matrix. (The actual implementation incorporates vectorization and accounts for the 8-fold permutation symmetry of the ERIs.) Strategies similar to those employed by Barca et al.[18] are used for integral screening and workload partitioning. The algorithm starts by grouping the shells of CGFs that share the same angular momentum and contraction order, forming sets of shells denoted as $|\mathbf{a}, K_a\}$ (line 1). Shell pairs are then constructed using Cartesian products between the shells in each group, resulting in $|\mathbf{ab}, K_aK_b\} = |\mathbf{a}, K_a\} \otimes |\mathbf{b}, K_b\}$ (line 6). These pairs are further "binned" into $n_{\mathbf{ab}}$ batches indexed by the size parameter $s_{\mathbf{ab}}$ (line 8), defined as

$$s_{\mathbf{ab}} = \begin{cases} \left\lfloor \dfrac{\log_{10} I_{\mathbf{ab}}}{\log_{10} \tau} n_{\mathbf{ab}} \right\rfloor, & I_{\mathbf{ab}} < 1 \\ 0, & I_{\mathbf{ab}} \geqslant 1 \end{cases} \tag{6}$$

where the labels for contraction orders are omitted for clarity (similarly hereafter). In eq 6, τ is a positive integral accuracy threshold smaller than 1, $n_{\mathbf{ab}}$ is a heuristic parameter determined such that each bin contains roughly 128 shell pairs, and

$$I_{\mathbf{ab}} = \max_{a \in \mathbf{a},\, b \in \mathbf{b}} |(ab|ab)|^{1/2} \tag{7}$$

is the conventional Cauchy−Schwarz bound factor.[31] Note that the shell pairs are prescreened based on the condition $I_{\mathbf{ab}} > \tau$ before the binning process (line 7), ensuring that at most $n_{\mathbf{ab}}$ bins are generated.

The main computational loops are executed over the sets of "bra-ket" shell quartets, namely, $\{\mathbf{ab}|\mathbf{cd}\} = \{\mathbf{ab}| \otimes |\mathbf{cd}\}$ (lines 10 and 11). For each set, a loop over the $n_{\mathbf{cd}}$ batches of the ket shell pairs is further carried out (line 12), within which the significant bra shell pairs are selected according to the criterion $I_{|\mathbf{ab}\}}[s_{\mathbf{ab}}] I_{|\mathbf{cd}\}}[s_{\mathbf{cd}}] P_{\{\mathbf{ab}|\mathbf{cd}\}} > \tau$ (line 14), where

$$I_{|\mathbf{ab}\}}[s_{\mathbf{ab}}] = 10^{((s_{\mathbf{ab}}/n_{\mathbf{ab}})\log_{10}\tau)} \tag{8}$$

is the upper bound of the Cauchy−Schwarz bound factor for shell pairs in the $s_{\mathbf{ab}}$th batch, and

$$P_{\{\mathbf{ab}|\mathbf{cd}\}} = \max_{\substack{\mathbf{i},\mathbf{j} \in \{\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}\} \\ \mathbf{i} \neq \mathbf{j}}} |P_{\{\mathbf{i}|\mathbf{j}\}}| \tag{9}$$

is the maximum element across the corresponding sub-blocks of the density matrix (e.g., $P_{\{\mathbf{a}|\mathbf{b}\}}$ represents the blocks with bra and ket basis functions belonging to $|\mathbf{a}\}$ and $|\mathbf{b}\}$ shell sets, respectively). This screening procedure is performed at the level of batches of shell pairs, which preserves the continuous layout of the shell pair data and facilitates efficient coalesced memory access by the GPU threads. Finally, the GPU kernel (`jk_kernel`) is dispatched to compute the Coulomb (**J**) and exchange (**K**) matrices by using the screened shell quartets (line 16).

Algorithm 1. Workflow of building the Fock matrix

```
1  shl_sets = [|a}, |b}, ...]
2  shlpr_sets = [ ]
3  for |a} in shl_sets:
4      for |b} in shl_sets:
5          |ab} = [[ ], ] * n_ab
6          for |ab) in |a} ⊗ |b}:
7              if I_ab > τ:
8                  |ab}[s_ab].append(|ab))
9          shlpr_sets.append(|ab})
10 for |ab} in shlpr_sets:
11     for |cd} in shlpr_sets:
12         for s_cd in range(n_cd):
13             for s_ab in range(n_ab):
14                 if I_|ab}[s_ab] I_|cd}[s_cd] P_{ab|cd} < τ:
15                     break
16             jk_kernel(|ab}[: s_ab] ⊗ |cd}[s_cd]) #on GPU
```

The Fock build is parallelized over a 2D GPU thread grid, with the bra shell pairs distributed in one dimension and the ket shell pairs distributed in the other. Each thread evaluates the ERIs of a shell quartet, contracts them with the density matrix, and accumulates the results into the **J**/**K** matrix. Workload balance among the threads is ensured, given that the shell quartets are of the same type (with respect to angular momenta and contraction orders).

As mentioned in Section II, GPU kernel `jk_kernel` has two distinct designs depending on the value of N. For $N \leq 3$ (see Algorithm 2), we use metaprogramming to unroll the loops involved in the evaluation of the RRs (line 6), thereby explicitly storing the 2D integrals (I_x, I_y, and I_z) and other intermediates in registers to minimize the memory latency. In addition, the primitive ERIs are first contracted (line 7) before being digested (lines 8−13), as sufficient registers are available to hold the contracted ERIs. (Note that the contraction coefficients and the Rys quadrature weights have been

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

absorbed into I_x, I_y, and I_z in Algorithm 2.) Similarly, loops associated with the ERI digestion are also unrolled, with the final results accumulated into the **J**/**K** matrix (stored in global memory) by using the atomic operation (`atomicAdd`), which avoids explicit thread synchronizations.

Algorithm 2. `jk_kernel` for $N \leqslant 3$

```
1  (ab|cd) = 0
2  for |ab] in |ab):
3      for |cd] in |cd):
4          t, w = rys_roots<N>() #Rys roots/weights
5          for i in range(N):
6              I_x, I_y, I_z = rr<N>(t[i], w[i]) #unroll RRs
7              (ab|cd) += I_x * I_y * I_z
   #For each element of J/K:
8  atomicAdd(J_ab, (ab|cd) * P_cd)
9  atomicAdd(J_cd, (ab|cd) * P_ab)
10 atomicAdd(K_ac, (ab|cd) * P_bd)
11 atomicAdd(K_bd, (ab|cd) * P_ac)
12 atomicAdd(K_ad, (ab|cd) * P_bc)
13 atomicAdd(K_bc, (ab|cd) * P_ad)
```

For larger N values, `jk_kernel` adopts a general implementation, where the key difference is that the ERI digestion now occurs before the contraction. This can be seen from Algorithm 3 in which the **J**/**K** matrix is updated (e.g., line 23) within the loops over the basis functions. Because the contracted ERIs no longer fit into registers or fast memory, the contraction-then-digestion procedure will result in storing to and loading from global memory, which may significantly hinder the performance. Due to the same reason, the 2D integrals can only be stored in local memory. They are computed once for all of the Rys roots (line 5) to avoid increasing the number of updates to the **J**/**K** matrix. Furthermore, to reduce global memory loads for retrieving the density matrix, reusable strides (i.e., `P_ac` and `P_ad`) are cached in local memory, potentially benefiting from optimal L1 and L2 cache utilization. The same strategy applies to temporary stores of the potential matrix (i.e., `V_ac` and `V_ad`). Additional scalar intermediates are also introduced (e.g., `P_cd` and `V_cd`), which use registers for data loading and stores. While shared memory (part of the L1 cache) could be used for data caching, our experiment showed that it did not lead to better performance. Since the cached intermediates are streaming rather than persistent data, the compiler may optimize their memory usage more effectively than through manual manipulation.

Finally, it should be noted that the demand for local memory in Algorithm 3 increases rapidly with an increase in angular momentum. For example, the integral class of (**ii**|**ii**) requires 731 KB of storage for the 2D integrals, which exceeds the maximum allowed local memory of 512 KB per thread on the NVIDIA A100 GPU used here. Therefore, our present implementation supports only ERIs with up to g functions.

IV. NUCLEAR GRADIENT

The nuclear gradient of the electronic energy in the Hartree−Fock method is expressed as

$$\nabla_{\mathbf{R}} E = \nabla_{\mathbf{R}}(E_{\text{core}} + E_J + E_K) - \sum_{ab} W_{ab} \nabla_{\mathbf{R}} S_{ab} \tag{10}$$

Algorithm 3. `jk_kernel` for $N > 3$

```
1  get idx[...] #precomputed ERI to I_x, I_y, I_z mapping
2  for |ab] in |ab):
3      for |cd] in |cd):
4          t, w = rys_roots<N>() #Rys roots/weights
5          I_x, I_y, I_z = rr<N>(t, w) #RRs for all roots
           #15 is the size of a g shell
6          allocate V_ac[15], V_ad[15], P_ac[15], P_ad[15]
7          for |d) in |d):
8              V_ad[...] = 0
9              P_ad[...] = P_(a|d)
10             for |c) in |c):
11                 V_ac[...] = V_cd = 0
12                 P_ac[...] = P_(a|c)
13                 P_cd = P_cd
14                 for |b) in |b):
15                     V_bc = V_bd = 0
16                     P_bc = P_bc
17                     P_bd = P_bd
18                     for |a) in |a):
19                         α, β, γ = idx[a, b, c, d]
20                         g = 0
21                         for i in range(N):
22                             g += I_x[i, α]*I_y[i, β]*I_z[i, γ]
23                         atomicAdd(J_ab, g * P_cd)
24                         V_ac[a] += g * P_bd
25                         V_ad[a] += g * P_bc
26                         V_bc += g * P_ad[a]
27                         V_bd += g * P_ac[a]
28                         V_cd += g * P_ab
29                     atomicAdd(K_bc, V_bc)
30                     atomicAdd(K_bd, V_bd)
31                 for |a) in |a):
32                     atomicAdd(K_ac, V_ac[a])
33                 atomicAdd(J_cd, V_cd)
34             for |a) in |a):
35                 atomicAdd(K_ad, V_ad[a])
```

where E_{core} represents the energy associated with the one-electron core Hamiltonian, E_J and E_K denote the Coulomb and exchange energies, respectively, **W** is the orbital energy weighted density matrix,[32] and **S** is the overlap matrix. In this work, only the computationally intensive Coulomb and exchange energy gradients are evaluated on the GPU. The contributions from the one-electron integrals are computed on the CPU (with the exception of the nuclear-electron Coulomb attraction, which is computed on the GPU using a three-center integral code that is introduced in ref 24), allowing for concurrent execution with the GPU calculations.

A general mth order ERI derivative using the Rys quadrature method can be straightforwardly evaluated as follows:[28]

$$\nabla^m_{\mathbf{R}}\text{ERI} = \sum_n w_n \mathcal{F}_x[I_x(t_n)]\mathcal{F}_y[I_y(t_n)]\mathcal{F}_z[I_z(t_n)] \tag{11}$$

where $\mathcal{F}$ stands for a linear function. Therefore, our GPU implementation of the nuclear gradients for the Coulomb and exchange energies closely aligns with the implementation of `jk_kernel` as described in Section III. For example, computing the nuclear gradient for the integral class (**ss**|**ss**) is similar to computing the energy for the integral class (**ps**|**ss**).

Nevertheless, the presence of the gradient operator adds an additional dimension (i.e., nuclear coordinates **R**) to the ERIs.

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

It also increases the total angular momentum by 1. Moreover, while up to 8-fold permutation symmetry can be utilized in energy evaluation, at most 2-fold symmetry can be exploited for the ERI gradient. Consequently, this leads to significantly higher register usage and memory footprint for evaluating the RRs and the ERI gradient. Additionally, more atomic operations are required if the Fock matrix gradient is to be computed and stored.

In order to overcome these bottlenecks, we double-contracted the ERI gradient with the density matrix to directly obtain the energy gradient. Specifically, this involves computing the following unique contributions on the fly:

$$\nabla_{\mathbf{R}_a} E_J = \sum_{abcd} (\nabla_{\mathbf{R}_a} ab|cd) P_{ab} P_{cd} \tag{12}$$

$$\nabla_{\mathbf{R}_b} E_J = \sum_{abcd} (a\nabla_{\mathbf{R}_b} b|cd) P_{ab} P_{cd} \tag{13}$$

$$\nabla_{\mathbf{R}_a} E_K = \sum_{abcd} (\nabla_{\mathbf{R}_a} ab|cd)(P_{ac} P_{bd} + P_{ad} D_{bc}) \tag{14}$$

$$\nabla_{\mathbf{R}_b} E_K = \sum_{abcd} (a\nabla_{\mathbf{R}_b} b|cd)(P_{ac} P_{bd} + P_{ad} D_{bc}) \tag{15}$$

A general implementation of the GPU kernel (`ejk_grad_kernel`) for computing E_J and E_K gradients is demonstrated in Algorithm 4 for $N > 2$. (Note that N is determined after applying the gradient operator to the ERIs.) Similar to Algorithm 3, the 2D integral gradients are computed once for all Rys roots and stored in the local memory (line 5). However, the gradients of E_J and E_K for the two nuclear centers $\mathbf{R}_a$ and $\mathbf{R}_b$ comprise only 12 scalar numbers, which are cached in registers (line 1) and accumulated (lines 18 and 19) within the loops over the basis functions. Notably, atomic operations are no longer required within these loops. Instead, they are performed at the end of the kernel to write the results into global storage (lines 20 and 21), totaling 12 operations for each shell quartet. This can lead to significant performance gains compared with building the Fock matrix gradient. Finally, for $N \leq 2$, we cache the 2D integral gradients and other intermediates in registers to minimize memory latency.

Algorithm 4. `ejk_grad_kernel` for $N > 2$

```
1  J_αχ = K_αχ = 0 #α ∈ {a,b}  χ ∈ {x,y,z}
2  for |ab] in |ab):
3      for |cd] in |cd):
4          t, w = rys_roots<N>() #Rys roots/weights
5          I_αχ = rr1<N>(t, w) #gradient RRs
6          for |d) in |d):
7              for |c) in |c):
8                  P_cd = P_cd
9                  for |b) in |b):
10                     P_bc = P_bc
11                     P_bd = P_bd
12                     for |a) in |a):
13                         PJ = P_ab * P_cd
14                         PK = P_ac * P_bd + P_ad * P_bc
15                         g_αχ = 0
16                         for i in range(N):
17                             g_αχ +=
                                   F_αχ(I_αχ[i]) * I_αν[i] * I_αζ[i]
18                         J_αχ += g_αχ * PJ
19                         K_αχ += g_αχ * PK
20 atomicAdd(E'_J[α,χ], J_αχ)
21 atomicAdd(E'_K[α,χ], K_αχ)
```

V. RESULTS AND DISCUSSION

In this section, we present the performance of our GPU-accelerated HF method implemented within the GPU4PySCF module. All GPU calculations were performed on a single NVIDIA A100 GPU with 40 GB of VRAM. For comparison, the CPU calculations were performed using the AMD EPYC 7763 CPUs with 32 threads.

First, we compare the wall times for restricted HF (RHF) energy and nuclear gradient calculations using GPU4PySCF with those of other GPU-accelerated HF codes, including GAMESS[18−20] and QUICK.[33,34] Additionally, we provide results from the multithreaded CPU code in PySCF as a reference. The test set from ref 19 was used, which includes polyglycine (Gly_n) and RNA (RNA_n) molecules at various sizes with 213−843 and 131−1155 atoms, respectively, using the STO-3G, 6-31G, and 6-31G(d) basis sets. The integral threshold τ was set to 1×10^{-10}.

We present the results in Table 1 and Figure 1. It is evident that GPU4PySCF outperforms QUICK in both energy and nuclear gradient calculations, achieving speedups of over a factor of 2. For energy evaluations of the polyglycine systems, similar timings were observed when comparing GPU4PySCF to GAMESS. However, for the RNA systems, GAMESS outperforms GPU4PySCF, especially with a minimal basis set. Furthermore, we also compare the computational scalings for different codes in Table 2. Both GPU4PySCF and QUICK exhibit approximately quadratic scaling, whereas GAMESS approaches linear scaling. Finally, GPU4PySCF is 1−2 orders of magnitude more efficient than PySCF, highlighting its practical usefulness.

We note that our benchmark is not comprehensive. Although we have not timed against proprietary or unreleased implementations in this work, recent benchmarking of the EXESS package developed by Barca et al. shows that it can outperform GPU4PySCF by a factor of 2−3 for Gly_n with the 6-31G(d) basis set.[21] It is also indicated in the same work that Terachem is faster than GPU4PySCF by a factor of 2. GPU4PySCF lacks some of the optimizations in other works, but given the simple algorithm and implementation, we consider it promising that it is within a factor of ∼2 of the efficiency of these more mature implementations.

Next, we analyze the FLOP performance of the two GPU kernels (i.e., `jk_kernel` and `ejk_grad_kernel`) for various integral classes and N values using the roofline model. Profiling was performed on a water cluster system consisting of 32 water molecules using the cc-pVQZ basis set, which includes up to g functions. The results are displayed in Figures 2 and 3, respectively. We note that kernels with $N > 5$ typically involve f orbitals, and $N > 7$ kernels involve g orbitals.

The roofline (solid blue line) represents the performance bound of the NVIDIA A100 GPU, which includes a ceiling derived from the peak memory bandwidth (diagonal line) and the processor's peak FLOP rate (horizontal line). The dashed black line indicates the peak FLOP/peak memory ratio for the A100 GPU (6.1FLOP/byte). Kernels with an arithmetic intensity smaller than this ratio are considered memory-bound, while those with an arithmetic intensity greater than it are compute-bound.

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

Table 1. Wall Times (in Seconds) for 10 Self-Consistent Field (SCF) Iterations and Nuclear Gradient Calculations for Various Molecules and Basis Sets at the RHF Level of Theory

			10 SCF iterations				nuclear gradient		
system	basis set	N_{basis}	GPU4PySCF	GAMESS[a]	QUICK	PySCF	GPU4PySCF	QUICK	PySCF
Gly_{30}	STO-3G	697	2.4	2.3	3.5	74.2	3.7	6.6	84.3
	6-31G	1273	6.4	15.2	12.4	238.9	7.8	18.2	288.9
	6-31G(d)	1878	17.4	34.4	44.1	477.4	29.1	61.2	579.6
Gly_{40}	STO-3G	927	3.4	3.4	5.9	130.1	5.7	11.3	154.4
	6-31G	1693	10.4	19.2	23.0	430.1	12.4	31.1	576.6
	6-31G(d)	2498	28.9	45.9	80.2	880.7	49.7	107.0	1148.2
Gly_{50}	STO-3G	1157	5.2	4.7	9.3	213.2	8.7	17.5	262.5
	6-31G	2113	16.0	24.3	38.0	686.8	19.8	49.2	1032.3
	6-31G(d)	3118	44.1	61.2	130.4	1444.4	74.8	168.1	2006.6
Gly_{60}	STO-3G	1387	7.2	6.1	13.4	306.1	13.0	25.7	404.8
	6-31G	2533	21.3	31.1	57.6	1035.0	28.0	72.5	1688.3
	6-31G(d)	3738	61.2	80.8	190.9	2194.6	107.3	241.3	3278.1
Gly_{70}	STO-3G	1617	9.4	8.0	18.5	421.7	17.3	35.8	614.5
	6-31G	2953	28.8	39.7	81.2	1439.1	37.6	101.4	2636.0
	6-31G(d)	4358	82.9	103.7	263.9	3145.9	144.3	343.8	5033.1
Gly_{80}	STO-3G	1847	11.9	10.1	24.2	555.6	22.6	46.7	832.3
	6-31G	3373	35.9	48.4	109.0	1976.3	48.9	135.6	3959.9
	6-31G(d)	4978	107.0		352.4	4368.1	188.1	474.8	7461.2
Gly_{90}	STO-3G	2077	15.0	12.5	31.3	713.0	28.5	58.9	1134.5
	6-31G	3793	45.0	58.5	140.7	2591.7	62.2	182.0	5750.5
	6-31G(d)	5598	134.8		554.3	5865.5	238.3	671.6	10 723.6
Gly_{100}	STO-3G	2307	18.7	14.9	39.7	908.5	40.4	75.5	1527.3
	6-31G	4213	56.3	71.4	213.1	3334.7	74.0	242.0	8093.5
	6-31G(d)	6218	172.8		668.1	7689.4	269.8	623.5	14 977.0
Gly_{110}	STO-3G	2537	21.8	17.9	49.4	1111.5	42.6	92.3	1992.5
	6-31G	4633	65.5	83.9	226.0	4217.0	92.7	283.6	11 131.9
	6-31G(d)	6838	194.2		804.9	9977.5	326.5	1020.2	20 468.9
Gly_{120}	STO-3G	2767	26.0	20.8	58.5	1350.3	50.5	120.7	2569.5
	6-31G	5053	78.1		311.3	5295.4	111.0	375.4	15 027.9
	6-31G(d)	7458	240.2		962.8	12 702.1	427.3	1184.4	27 402.3
RNA_1	STO-3G	491	3.7	2.4	4.3		6.6	8.0	
	6-31G	880	13.3	21.2	18.0		16.6	23.3	
	6-31G(d)	1310	30.7	47.0	68.1		47.2	75.8	
RNA_2	STO-3G	975	15.7	7.0	21.1		28.3	33.7	
	6-31G	1747	46.2	36.6	102.9		60.2	95.8	
	6-31G(d)	2602	120.4	95.8	343.7		192.8	315.4	
RNA_3	STO-3G	1459	36.1	14.3	53.3		65.7	77.5	
	6-31G	2614	98.9	64.5	263.8		130.2	219.7	
	6-31G(d)	3894	318.2	184.2	880.0		508.9	717.2	
RNA_4	STO-3G	1943	67.1	24.5	101.8		124.2	140.4	
	6-31G	3481	169.7	107.2	505.7		228.5	405.8	
	6-31G(d)	5186	584.7		1625.5		865.8	1328.7	
RNA_5	STO-3G	2427	102.5	37.3	171.6		189.4	223.9	
	6-31G	4348	268.6	166.3	1008.2		358.5	689.2	
	6-31G(d)	6478	931.8		3162.4		1490.1	2451.7	
RNA_6	STO-3G	2911	150.7	53.9	267.5		291.2	342.5	
	6-31G	5215	382.0		1454.6		514.5	1061.6	
	6-31G(d)	7770	1317.3		4612.8		2172.0	3655.8	
RNA_7	STO-3G	3395	206.7	71.2	374.9		358.8	485.1	
	6-31G	6082	509.8		2050.0		659.2	1430.6	
	6-31G(d)	9062	1790.7		6258.1		2735.3	4680.6	
RNA_8	STO-3G	3879	283.0	97.2	480.2		515.8	683.8	
	6-31G	6949	702.2		2845.2		893.2	2025.6	
	6-31G(d)	10 354	2513.8		8953.9		3793.4	7149.2	
RNA_9	STO-3G	4363	344.5	118.3	647.8		679.0	855.5	
	6-31G	7816	837.0		3560.8		1163.6	2485.7	
	6-31G(d)	11 646	3008.3		11 095.9		4624.3	8687.9	

[a]Results are obtained from ref 19.

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

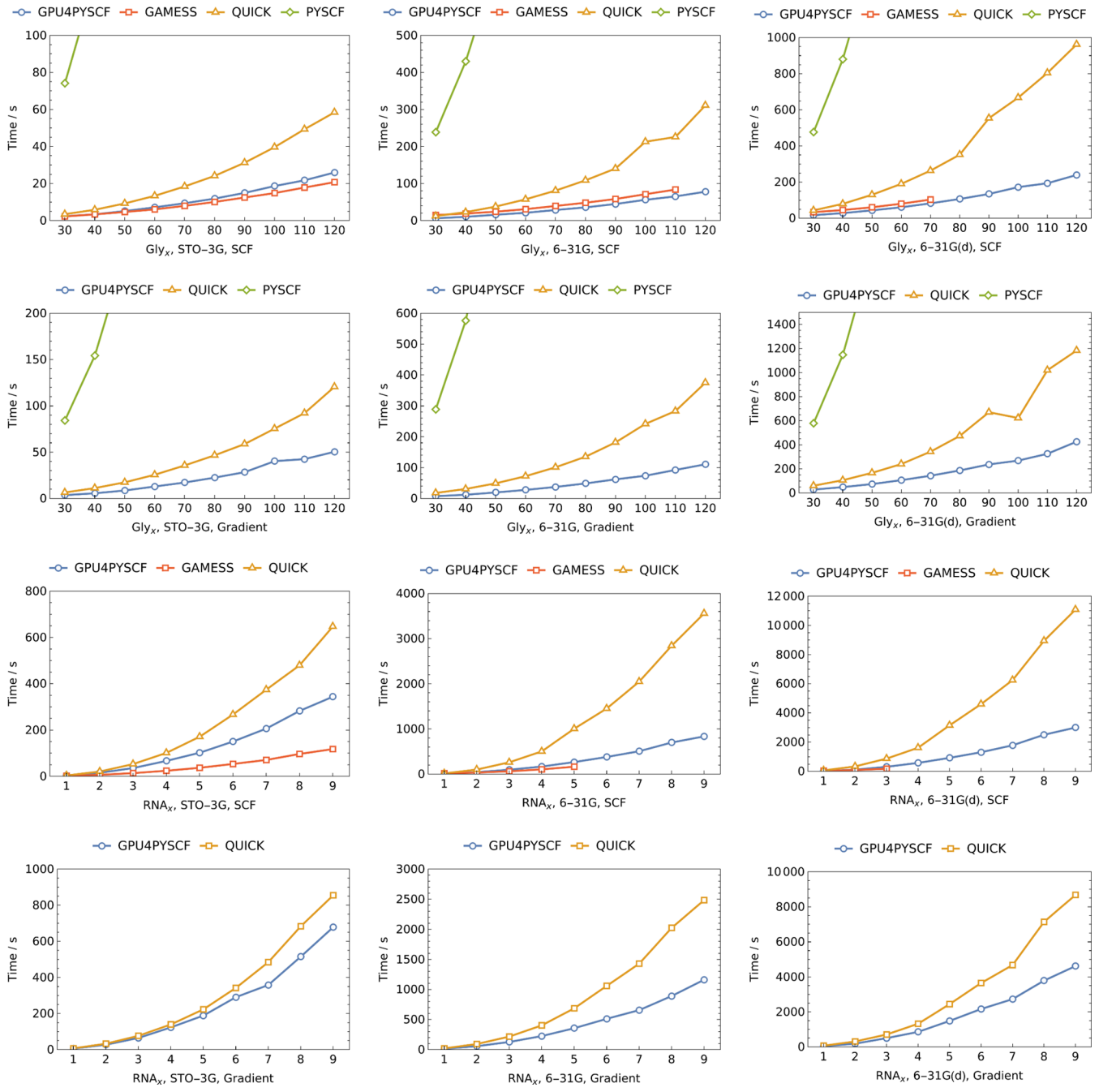


Figure 1. Graphic representation of Table 1.

Table 2. Observed Computational Scalings [a in $O(N_{\text{basis}}^{a})$] for Energy and Nuclear Gradient Calculations using Different RHF Codes

		10 SCF iterations				nuclear gradient		
system	basis set	GPU4PySCF	GAMESS	QUICK	PySCF	GPU4PySCF	QUICK	PySCF
Gly_x	STO-3G	1.77	1.62	2.07	2.16	1.96	2.08	2.49
	6-31G	1.81	1.35	2.32	2.29	1.94	2.20	2.88
	6-31G(d)	1.91	1.31	2.28	2.42	1.91	2.15	2.81
RNA_x	STO-3G	2.08	1.81	2.29		2.10	2.14	
	6-31G	1.91	1.28	2.43		1.94	2.16	
	6-31G(d)	2.12	1.23	2.35		2.12	2.19	

From Figure 2, we observe that for most integral classes with $N \leq 3$ [e.g., (**ss**|**ss**), (**ps**|**ss**), (**ds**|**ss**), (**fd**|**ss**), and (**pp**|**pp**)], `jk_kernel` is compute-bound and achieves an impressive FLOP rate ranging from 2TFLOP/s to 5TFLOP/s. However, there are exceptions, such as the integral class (**dp**|**pp**), which exhibits memory-bound character and limited FLOP perform-

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

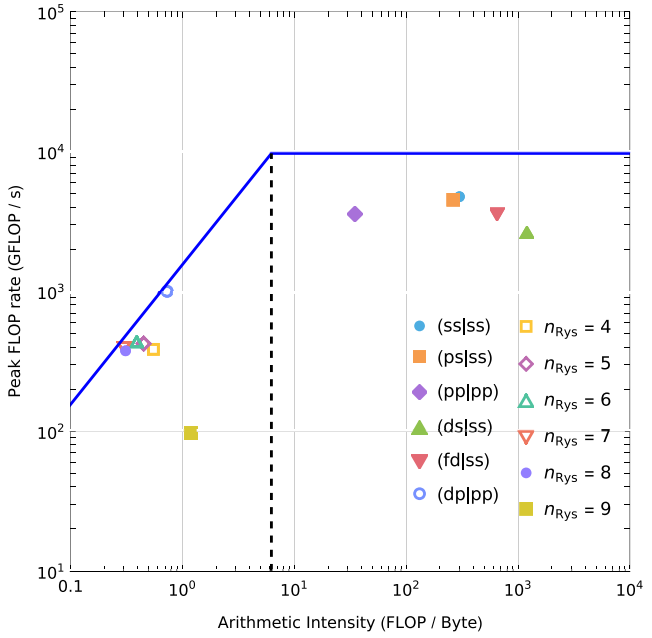


Figure 2. FLOP performance of the GPU kernels `jk_kernel` analyzed using the roofline model on the NVIDIA A100 GPU. The solid blue line represents the official peak (FP64) FLOP rate of 9.7TFLOP/s with no bandwidth constraint (horizontal) and the peak FP64 FLOP rate constrained by the peak memory bandwidth of 1.6TB/s (diagonal). The dashed black line indicates the theoretical arithmetic intensity, where the peak FLOP rate is no longer constrained by the memory bandwidth, 6.1FLOP/byte. The calculations were performed for a water cluster system consisting of 32 water molecules at the RHF/cc-pVQZ level of theory.

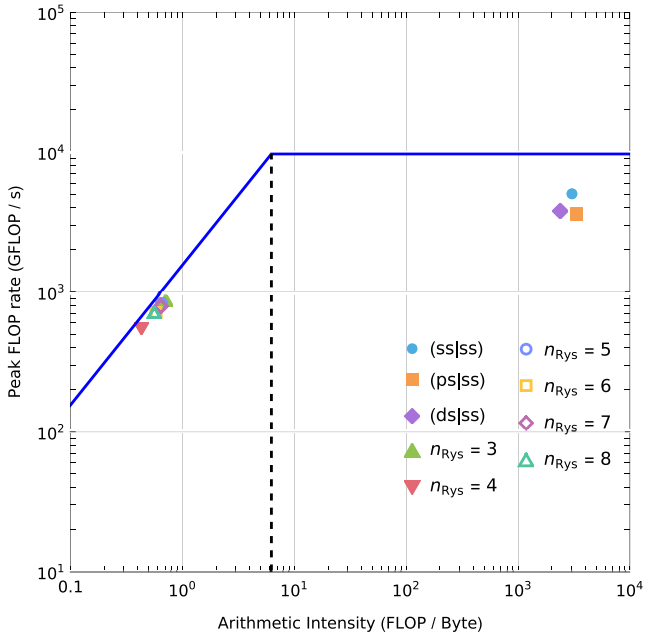


Figure 3. Same as Figure 2, but for the GPU kernels `ejk_grad_kernel`.

ance. This is due to the need to cache more than 234 FP64 words for intermediates per GPU thread, which exceeds the maximum number of registers (255 FP32 words) each thread can use by nearly a factor of 2. Consequently, intermediates that cannot fit into registers are likely stored in slow memory (known as register spilling), resulting in significant memory latency when accessed frequently.

For $N > 3$, `jk_kernel` is always memory-bound due to the use of local memory for storing intermediates. Nonetheless, the kernel generally utilizes the GPU hardware efficiently, as indicated by data points lying close to the roofline. An exception is the integral class (**gg|gg**) (with $N = 9$), which shows a potential loss of parallelization. This is mainly because of the insufficient workload to fully occupy the streaming multiprocessors (SMs), as only the O atoms contain g shells, and each O atom contains only one shell of g functions.

Similarly, `ejk_grad_kernel` shows a remarkable FLOP performance of over 3TFLOP/s for integral classes with $N \leq 2$, where intermediates can be cached in registers. For $N > 2$, the kernel is again memory-bound due to the use of local memory. However, all data points in Figure 3 lie close to the roofline, indicating efficient utilization of GPU hardware. Notably, even for $N = 7$, a FLOP rate of 0.8TFLOP/s is achieved, outperforming its `jk_kernel` counterpart for Fock builds by a factor of 8. This can be attributed to our integral-direct approach, as shown in Algorithm 4. It eliminates the need to compute the Fock matrix gradient, which would otherwise be stored in global memory. As a result, significantly fewer atomic operations and slow memory accesses are performed, enhancing cache utilization. In addition, the workload involved in gradient calculations is greater than that in Fock builds [e.g., the integral class (**ff|fd**) also corresponds to $N = 7$ when evaluating its gradient], which keeps more GPU threads active and helps hide latency more effectively. $N = 9$ kernels (e.g., (**gg|gg**)) could not be profiled on the testing platform, as they require 47628 FP64 words per thread for intermediates, which quickly consume the 40 GB memory of the A100 GPU.

VI. CONCLUSIONS

In this work, we introduced the GPU4PySCF module and, in particular, the core ERI CUDA kernels that form the starting point for accelerating quantum chemistry calculations. As an example of their use, we described a GPU-accelerated HF method for energy and nuclear gradient calculations, including the detailed optimizations required to achieve high GPU efficiency.

The GPU acceleration of quantum chemistry is integral not only to advancing traditional quantum chemistry calculations but also to bringing quantum chemical methods and data into new disciplines, such as machine learning. We hope that by providing a community-based, open-source implementation of GPU-accelerated quantum chemistry algorithms, we can help the growth of quantum chemistry in these areas. Indeed, we note that as a living open-source code, at the time of writing GPU4PySCF already contains new contributions targeted at these directions.[24]

AUTHOR INFORMATION

Corresponding Author

Garnet Kin-Lic Chan − *Division of Chemistry and Chemical Engineering, California Institute of Technology, Pasadena, California 91125, United States;* orcid.org/0000-0001-8009-6038; Email: gkc1000@gmail.com

Authors

Rui Li − *Division of Chemistry and Chemical Engineering, California Institute of Technology, Pasadena, California 91125, United States*

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468

Qiming Sun − *Quantum Engine LLC, Lacey, Washington 98516, United States*
Xing Zhang − *Division of Chemistry and Chemical Engineering, California Institute of Technology, Pasadena, California 91125, United States*

Complete contact information is available at: https://pubs.acs.org/10.1021/acs.jpca.4c05876

Author Contributions

[§]R.L. and Q.S. contributed equally to this work.

Notes

The authors declare the following competing financial interest(s): GKC is part owner of QSimulate, Inc.

■ ACKNOWLEDGMENTS

We acknowledge the generous contributions of the open-source community, in particular Xiaojie Wu of Bytedance US, to the GPU4PySCF module. Work carried out by Q.S. (development of initial ERI code and Fock build) was performed as a part of a software contract with GKC through the California Institute of Technology, funded by internal funds. R.L. (development of gradient ERIs and gradient code) and G.K.-L.C. (project supervision) were supported by the US Department of Energy, Office of Science, through Award No. DE-SC0023318. X.Z. (additional data analysis) was supported by the Center for Molecular Magnetic Quantum Materials, an Energy Frontier Research Center funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences under Award No. DE-SC0019330. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award ERCAP-0024087. G.K.-L.C. is a Simons Investigator in Physics.

■ REFERENCES

(1) Paszke, A.; Gross, F.; Massa, A.; Lerer, J.; Bradbury, G.; Chanan, T.; Killeen, Z.; Lin, N.; Gimelshein, L.; Antiga, A.; Desmaison, A.; Kopf, E.; Yang, Z.; DeVito, M.; Raison, A.; Tejani, S.; Chilamkurthy, B.; Steiner, L.; Fang, J. B.; Chintala, S. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32*; Curran Associates, Inc., 2019; pp 8024−8035.
(2) Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mane, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viegas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; Zheng, X. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2016, arXiv:1603.04467. arXiv.org e-Print archive. https://arxiv.org/abs/1603.04467.
(3) Springer, P.; Yu, C.-H. In *cuTENSOR: High-Performance CUDA Tensor Primitives*, NVIDIA GPU Technology Conference 2019, 2019.
(4) Nersc Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/.
(5) Yasuda, K. Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.* **2008**, *29*, 334.
(6) White, C. A.; Head-Gordon, M. Aj matrix engine for density functional theory calculations. *J. Chem. Phys.* **1996**, *104*, 2620−2629.
(7) Challacombe, M.; Schwegler, E. Linear scaling computation of the Fock matrix. *J. Chem. Phys.* **1997**, *106*, 5526−5536.
(8) Ufimtsev, I. S.; Martínez, T. J. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral valuation. *J. Chem. Theory Comput.* **2008**, *4*, 222.
(9) Ufimtsev, I. S.; Martínez, T. J. Quantum chemistry on graphical processing units. 2. direct self-consistent-field implementation. *J. Chem. Theory Comput.* **2009**, *5*, 1004.
(10) McMurchie, L. E.; Davidson, E. R. One- and two-electron integrals over cartesian gaussian functions. *J. Comput. Phys.* **1978**, *26*, 218−231.
(11) Titov, A. V.; Ufimtsev, I. S.; Luehr, N.; Martinez, T. J. Generating efficient quantum chemistry codes for novel architectures. *J. Chem. Theory Comput.* **2013**, *9*, 213−221.
(12) Wang, Y.; Hait, D.; Johnson, K. G.; Fajen, O. J.; Zhang, J. H.; Guerrero, R. D.; Martínez, T. J. Extending GPU-accelerated Gaussian integrals in the TeraChem software package to f type orbitals: Implementation and applications. *J. Chem. Phys.* **2024**, *161*, No. 174118.
(13) 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.
(14) Dupuis, M.; Rys, J.; King, H. F. Evaluation of molecular integrals over gaussian basis functions. *J. Chem. Phys.* **1976**, *65*, 111−116.
(15) Rys, J.; Dupuis, M.; King, H. F. Computation of electron repulsion integrals using the rys quadrature method. *J. Comput. Chem.* **1983**, *4*, 154−157.
(16) Miao, Y.; Merz, K. M. J. Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations. *J. Chem. Theory Comput.* **2013**, *9*, 965−976.
(17) 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.
(18) Barca, G. M. J.; Galvez-Vallejo, J. L.; Poole, D. L.; Rendell, A. P.; Gordon, M. S. High-performance, graphics processing unit-accelerated fock build algorithm. *J. Chem. Theory Comput.* **2020**, *16*, 7232−7238.
(19) Barca, G. M. J.; Alkan, M.; Galvez-Vallejo, J. L.; Poole, D. L.; Rendell, A. P.; Gordon, M. S. Faster self-consistent field (scf) calculations on gpu clusters. *J. Chem. Theory Comput.* **2021**, *17*, 7486−7503.
(20) Zahariev, F. P.; Xu, B. M.; Westheimer, S.; Webb, J.; Galvez Vallejo, A.; Tiwari, V.; Sundriyal, M.; Sosonkina, J.; Shen, G.; Schoendorff, M.; Schlinsog, T.; Sattasathuchana, K.; Ruedenberg, L. B.; Roskop, A. P.; Rendell, D.; Poole, P.; Piecuch, B. Q.; Pham, V.; Mironov, J.; Mato, S.; Leonard, S. S.; Leang, J.; Ivanic, J.; Hayes, T.; Harville, K.; Gururangan, E.; Guidez, I. S.; Gerasimov, C.; Friedl, K. N.; Ferreras, G.; Elliott, D.; Datta, D. D. A.; Cruz, L.; Carrington, C.; Bertoni, G. M. J.; Barca, M.; Alkan; Gordon, M. S. The general atomic and molecular electronic structure system (gamess): Novel methods on novel architectures. *J. Chem. Theory Comput.* **2023**, *19*, 7031−7055.
(21) Palethorpe, E.; Stocks, R.; Barca, G. M. J. Advanced Techniques for High-Performance Fock Matrix Construction on GPU Clusters. 2024, arXiv:2407.21445. arXiv.org e-Print archive. https://arxiv.org/abs/2407.21445.
(22) Asadchev, A.; Valeev, E. F. High-performance evaluation of high angular momentum 4-center gaussian integrals on modern accelerated processors. *J. Phys. Chem. A* **2023**, *127*, 10889−10895.
(23) Asadchev, A.; Valeev, E. F. 3-Center and 4-Center 2-Particle Haussian ao Integrals on Modern Accelerated Processors. 2024, arXiv:2405.01834. arXiv.org e-Print archive. https://arxiv.org/abs/2405.01834.
(24) Wu, X.; Sun, Q.; Pu, Z.; Zheng, T.; Ma, W.; Yan, W.; Yu, X.; Wu, Z.; Huo, M.; Li, X.; Ren, W.; Gong, S.; Zhang, Y.; Gao, W. Python-Based Quantum Chemistry Calculations with GPU Acceleration. 2024, arXiv:2404.09452. arXiv.org e-Print archive. https://arxiv.org/html/2404.09452v1.

(25) Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del Río, J. F.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; Oliphant, T. E. Array programming with NumPy. *Nature* **2020**, *585*, 357−362.
(26) Okuta, R.; Unno, Y.; Nishino, D.; Hido, S.; Loomis, C. In *CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations*, Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS), 2017.
(27) Sun, Q.; Zhang, X.; Banerjee, S.; Bao, P.; Barbry, M.; Blunt, N. S.; Bogdanov, N. A.; Booth, G. H.; Chen, J.; Cui, Z.-H.; Eriksen, J. J.; Gao, Y.; Guo, S.; Hermann, J.; Hermes, M. R.; Koh, K.; Koval, P.; Lehtola, S.; Li, Z.; Liu, J.; Mardirossian, N.; McClain, J. D.; Motta, M.; Mussard, B.; Pham, H. Q.; Pulkin, A.; Purwanto, W.; Robinson, P. J.; Ronca, E.; Sayfutyarova, E. R.; Scheurer, M.; Schurkus, H. F.; Smith, J. E. T.; Sun, C.; Sun, S.-N.; Upadhyay, S.; Wagner, L. K.; Wang, X.; White, A.; Whitfield, J. D.; Williamson, M. J.; Wouters, S.; Yang, J.; Yu, J. M.; Zhu, T.; Berkelbach, T. C.; Sharma, S.; Sokolov, A. Y.; Chan, G. K.-L. Recent developments in the pyscf program package. *J. Chem. Phys.* **2020**, *153*, No. 024109.
(28) Flocke, N.; Lotrich, V. Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations. *J. Comput. Chem.* **2008**, *29*, 2722−2736.
(29) Williams, S.; Waterman, A.; Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **2009**, *52*, 65−76.
(30) Sun, Q. Libcint: An efficient general integral library for gaussian basis functions. *J. Comput. Chem.* **2015**, *36*, 1664−1671.
(31) Whitten, J. L. Coulombic potential energy integrals and approximations. *J. Chem. Phys.* **1973**, *58*, 4496−4501.
(32) Pople, J. A.; Krishnan, R.; Schlegel, H. B.; Binkley, J. S. Derivative studies in hartree-fock and møller-plesset theories. *Int. J. Quantum Chem.* **2009**, *16*, 225−241.
(33) Manathunga, M.; Shajan, A.; Smith, J.; Miao, Y.; He, X.; Ayers, K.; Brothers, E.; Götz, A. W.; Merz, K. M. Quick-23.08 University of California, San Diego and Michigan State University, East Lansing, 2023.
(34) Manathunga, M.; Miao, Y.; Mu, D.; Götz, A. W.; Merz, K. M. J. Parallel implementation of density functional theory methods in the quantum interaction computational kernel program. *J. Chem. Theory Comput.* **2020**, *16*, 4315−4326.

https://doi.org/10.1021/acs.jpca.4c05876
J. Phys. Chem. A 2025, 129, 1459−1468