JCTC Journal of Chemical Theory and Computation

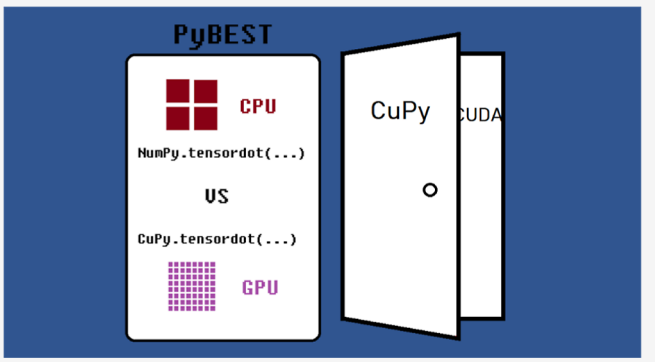
pubs.acs.org/JCTC Article

Accelerating Pythonic Coupled-Cluster Implementations: A Comparison Between CPUs and GPUs

Maximilian H. Kriebel,* Paweł Tecmer, Marta Gałyńska, Aleksandra Leszczyk, and Katharina Boguslawski*

Cite This: J. Chem. Theory Comput. 2024, 20, 1130−1142 Read Online

ABSTRACT: In this work, we benchmark several Python routines for time and memory requirements to identify the optimal choice of the tensor contraction operations available. We scrutinize how to accelerate the bottleneck tensor operations of Pythonic coupled-cluster implementations in the Cholesky linear algebra domain, utilizing a NVIDIA Tesla V100S PCIe 32GB (rev 1a) graphics processing unit (GPU). The NVIDIA compute unified device architecture API interacts with CuPy, an open-source library for Python, designed as a NumPy drop-in replacement for GPUs. Due to the limitations of video memory, the GPU calculations must be performed batch-wise. Timing results of some contractions containing large tensors are presented. The CuPy implementation leads to a factor of 10−16 speed-up of the bottleneck tensor contractions compared to computations on 36 central processing unit (CPU) cores. Finally, we compare example CCSD and pCCD-LCCSD calculations performed solely on CPUs to their CPU−GPU hybrid implementation, which leads to a speed-up of a factor of 3−4 compared to the CPU-only variant.



1. INTRODUCTION

A critical job of a graphics card is to compute projections of three-dimensional objects to a 2D surface using linear algebra. These calculations can be performed in parallel effectively, meaning that multiple small mathematical operations such as multiplication and addition can be performed simultaneously. For this reason, graphics processing unit (GPU) development mainly focuses on massively increasing the number of computing cores. While a central processing unit (CPU) may have up to 128 cores on the high end, GPUs already have up to 16 000 cores (like NVIDIA RTX 4090).

In scientific computing, the distribution of calculations among multiple CPU cores and multiple nodes is a standard practice.[1−5] Due to its inherently parallel structure, linear algebra operations can be calculated in parallel and, therefore, efficiently offloaded to the GPU.[6−10] The first to report independently that the internal hugely parallel structure of GPUs can be misused, not to compute graphics, but to be utilized in quantum chemistry were Yasuda[11] and Ufimtsev and Martinez,[12] respectively. Today, the potential of GPUs for nongraphics-related computations is widely understood and often used to accelerate quantum chemistry methods like density functional approximations,[13] Hartree−Fock theory,[14,15] Møller−Plesset perturbation theory,[16−18] coupled cluster (CC) theory,[3,8,19−21] and the evaluation of effective core potentials[22] to name a few. The NVIDIA compute unified device architecture (CUDA) API offers a C++ interface to utilize GPU computing power.[23] A relatively quick way to interact with this API is, for example, CuPy,[24] a Python library which internally uses CUDA routines for the utilization of NVIDIA GPUs and since release 9.0 also ROCm for the utilization of AMD GPUs. CuPy brings a lot of attention from Python-based software developers as its interface is highly compatible with NumPy[25] and allows even for a drop-in replacement in some particular cases. Although some fine control is sacrificed when exploiting libraries like CuPy, third-party libraries are quickly accessible without the necessity of in-detail backend control and, therefore, a very convenient and efficient way to probe graphics processor utilization in the first place.

In this work, we present benchmark results comparing the timings of the bottleneck tensor contractions present in CC calculations restricted to at most double excitations, where Cholesky vectors approximate the electron repulsion integrals.[26] Specifically, we perform several flavors of these

Received: October 6, 2023
Revised: January 12, 2024
Accepted: January 13, 2024
Published: February 2, 2024

bottleneck contractions on a CPU by using Python libraries and benchmark their resource requirements. These contractions are calculated with, for instance, NumPy's[25] tensordot and einsum routines or opt_einsum[27] on CPUs. Moreover, we elaborate on exporting these operations on a GPU exploiting CuPy's tensordot routine.[24] All the calculations are performed in the open-source PyBEST software platform.[28−32] The unique features of PyBEST include pCCD-based methods for ground- and excite-state calculations,[33] a quantum entanglement and correlation analysis,[34−38] and the design of a modular tensor contraction engine (TCE).[28−32] In the following, we focus on the TCE and its interface to the CuPy library. These characteristics make PyBEST different from other Python-based quantum chemistry software packages.[39−45]

This work is organized as follows. In Section 2, we briefly discuss the main bottleneck operations in CC calculations. Section 3 scrutinizes several Python-based strategies to compute the CC vector function. In Section 4, we examine the PyBEST tensor contraction engine. A GPU implementation exploiting the CuPy library is summarized in Section 5. Numerical results and the assessment of the GPU to CPU performance are presented in Section 6. Finally, we conclude in Section 7.

2. THE CC ANSATZ

Our starting point is the CC ansatz[46−51]

$$|\Psi\rangle = e^{\hat{T}}|\Phi_0\rangle \tag{1}$$

where $\hat{T}$ is the cluster operator and $|\Phi_0\rangle$ some reference wave function like the Hartree−Fock determinant. In this work, we will consider, at most, double excitations in the cluster operator, that is, $\hat{T} = \hat{T}_2$. We do not consider single excitations explicitly as the bottleneck operations are due to the $\hat{T}_2$ excitation operator. Furthermore, we will work in the spin-free representation, with spin-free amplitudes, and the CC equations are spin summed. In this picture, the double excitation operator takes on the form

$$\hat{T}_2 = \frac{1}{2}\sum_{ij}^{\text{occ}}\sum_{ab}^{\text{virt}} t_{ij}^{ab}\hat{E}_{ai}\hat{E}_{bj} \tag{2}$$

with the CCD amplitude t_{ij}^{ab} and $\hat{E}_{ai}$ being the singlet excitation operator

$$\hat{E}_{ai} = \hat{a}^{\dagger}\hat{i} + \hat{\bar{a}}^{\dagger}\hat{\bar{i}} \tag{3}$$

where $\hat{a}^{\dagger}$ ($\hat{\bar{a}}^{\dagger}$) indicate electron creation operators for α (β) electrons, while $\hat{i}$ ($\hat{\bar{i}}$) are the corresponding annihilation operators. The above sum runs over all occupied (occ) and virtual (virt) orbitals for the chosen reference determinant $|\Phi_0\rangle$.

The scaling-determining step in the CCD amplitude equations is associated with the following term

$$0 = \ldots + \sum_{cd}\langle ab|cd\rangle t_{ij}^{cd} + \ldots \tag{4}$$

Summation over the indices i, j, a, b is implied, which results in the formal scaling of the CCD equations of $O(o^2v^4)$. In the above equation, $\langle ab|cd\rangle$ are the electron-repulsion integrals (ERI) in Physicist's notation. To reduce the storage of the full ERI, they can be approximated by Cholesky decomposition[26,52]

$$\langle ab|cd\rangle \approx \sum_{x} L_{ac}^{x}L_{bd}^{x} \tag{5}$$

where x indicates the summation over the elements of the Cholesky vectors. Since we work with real orbitals with eightfold permutational symmetry of the ERI, both Cholesky vectors L_{ac}^{x} and L_{bd}^{x} are identical. Then, the CC excitation amplitudes are optimized iteratively by rewriting the rate-determining step of the vector function $\tilde{t}_{ij}^{ab}$ of iteration n of the optimization procedure

$$\tilde{t}_{ij}^{ab} = \ldots + \sum_{xcd} L_{ac}^{x}L_{bd}^{x}t_{ij}^{cd} + \ldots \tag{6}$$

including a third summation index running over the Cholesky vectors (ignoring the summation over the fixed indices i, j, a, b). Thus, formally, the complexity increases to $O(xo^2v^4)$. However, the formal scaling can be reduced to $O(xo^2v^3)$ (if $x > o^2$) or $O(o^2v^4)$ (if $v > o^2$) by defining suitable intermediates. While the former scheme creates an intermediate of size xo^2v^2, the latter one features an intermediate array of size v^4. Note that the dimension of x depends on the chosen threshold of Cholesky decomposition. For decent to tight thresholds (around 10^{-6}), we have $x \approx 5(o + v)$.

3. PYTHONIC IMPLEMENTATIONS OF THE CC VECTOR FUNCTION

To solve the CC amplitudes iteratively, we must evaluate the vector function of eq 6 in each iteration including all of its terms. Thus, we will focus on the bottleneck operations of the corresponding CCD vector function. Within a Python-based implementation, we can utilize various Python libraries to perform summations efficiently through an implementation which minimizes function calls.[28,29,39,40,44,45,53] To that end, Python-based implementations are becoming more popular among scientists. We should stress, however, on the fact that implementations based on compiled languages (such as C++ or FORTRAN) have more potential in delivering efficient computer codes. However, it comes at the cost of more laborious implementations. Based on the chosen routines, these summations (in the following called contractions) can be performed in one shot or sequentially, creating several intermediates to boost efficiency and reduce resource requirements or even enabling out-of-the-box parallelization. In the following, we will scrutinize different variants to evaluate eq 6 as Pythonically as possible, focusing solely on Python features and libraries. Specifically, we focus on the NumPy routines einsum and tensordot, the opt_einsum package, and a GPU implementation exploiting CuPy's tensordot feature.

3.1. einsum and opt_einsum. The possibly easiest and most straightforward way of avoiding nested for-loop implementations when dealing with tensor contractions is to refer to NumPy's einsum routine, which evaluates the Einstein summation convention on a sequence of operands. Using numpy.einsum, many—albeit not all—linear algebraic operations on multidimensional arrays can be represented in a simple and intuitive language. With an increasing version number, additional features and improvements have been incorporated into the numpy.einsum function. One crucial parameter is the optimize argument, which allows control over

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

intermediate optimization. If set to "optimal", an optimal path of the contraction in question will be performed. Another possibility is to exploit the numpy.einsum_path function to steer the order of the individual contractions in the most optimal way.

An effort to improve the performance of the original numpy.einsum routine led to the development of the opt_einsum package.[27] It offers several features to optimize numpy.einsum significantly, reducing the overall execution time of einsum-like expressions. For instance, it automatically optimizes the order of the underlying contraction and exploits specialized routines or BLAS.[54] opt_einsum can also handle various arrays from different frameworks such as NumPy, Dask, PyTorch, Tensorflow, or CuPy, to name a few. Furthermore, the optimization of numpy.einsum has been passed upstream of the original numpy.einsum project. Some of opt_einsum's features can hence be utilized by numpy.einsum modifying the optimization option. Since opt_einsum features more up-to-date algorithms for complex contractions, we will focus on the opt_einsum.contract function to evaluate the Einstein summation convention on a sequence of operands, typically containing three multidimensional input arrays. opt_einsum.contract represents a replacement for numpy.einsum, where the order of the contraction is optimized to reduce the overall scaling (and hence increase the computational speed-up) at the cost of several intermediate arrays. To steer the memory limit and prevent the generation of intermediates that are too large, opt_einsum.contract offers the memory_limit parameter to provide an upper bound of the largest intermediate array built during the tensor contraction.

Thus, the bottleneck contraction in eq 6 can be straightforwardly evaluated using, for instance, opt_einsum.contract as follows

```
t_new[:] += opt_einsum.contract("xac,xbd,icjd->iajb", L_0, L_1, t_old)
```

In the above code snippet, t_new indicates the vector function of iteration $n + 1$, t_old the current approximate solution (iteration n) of the CCD amplitudes, and L_0 (L_1) is the Cholesky vector of eq 5. Note that $\tilde{t}_{ij}^{ab}$ are stored as a four-dimensional NumPy array t_new[i,a,j,b].

3.2. tensordot. An alternative routine to perform a tensor contraction is the tensordot function offered in NumPy[55] and CuPy.[24] It efficiently computes the summation of one (or more) given index (indices). It allows for saving memory by de-allocating intermediate arrays and explicitly defining the path of the complete tensor contraction. Furthermore, tensordot makes use of the BLAS[54] API and features a multithreaded implementation when linked against the proper libraries like OpenBLAS,[56] MKL,[57] or ATLAS.[58] A contraction along one axis of two arrays A and B

$$C_{ij} = \sum_{n} A_{ni} B_{nj}$$

translates into

```
C = tensordot(A, B, axes=([0, 0]))
```

Similarly, tensordot can contract (that is, sum over) two or more axes in one function call, where

$$C_{ijkl\ldots} = \sum_{nm\ldots} A_{nimj\ldots} B_{nklm\ldots}$$

translates into

```
C = tensordot(A, B, axes=([0,2], [0,3]))
```

Note, however, that tensordot allows for contracting only two operands at a time. Thus, to evaluate the term in eq 6, a sequence of tensordot calls must be performed where suitable intermediates are created. One possibility is to contract the Cholesky vectors to create an intermediate of dimension v^4, which is then passed to a second tensordot call generating the desired output.

```
# first intermediate generates the dense ERI[a,c,b,d]
eri_acbd = tensordot(L_0, L_1, axes=([0, 0]))
# second contraction results in t_new[i,j,a,b]
t_new[:] += tensordot(t_old, eri_acbd, axes=([1,3], [1,3])).transpose
    ((0,2,1,3))
```

Note that tensordot does not reorder the axis. In that case, we need to transpose the intermediate result to match the shape of the output array (the vector function). However, generating a v^4 intermediate of the ERI might be prohibitive regarding memory requirements for larger systems. Other possibilities lead to even larger intermediates. For instance, contracting L_0 with t_old yields a multidimensional array of size xo^2v^3, which is smaller than the eri_acbd intermediate if $xo^2 \ll v$, where x is a prefactor depending on the threshold of the Cholesky-decomposed ERI. This prefactor is typically challenging to determine a priori. Nonetheless, for computationally feasible problems, the condition $xo^2 \ll v$ is rarely satisfied, making the first contraction path computationally more efficient, in terms of memory.

A less elegant, albeit computationally cheaper, way to use all of the benefits of the tensordot function is to introduce one for-loop to iterate over one axis. If we choose to loop over the second axis of L_0, we generate intermediates of at most v^3

```
for a in range(L_0.shape[1]):
    # first contract two Cholesky vectors
    eri_bdc = numpy.tensordot(L_1, L_0[:, a, :], axes=([0, 0]))
    # contract t_old[i,c,j,d] with eri_bdc (sum over c and d)
    t_new[:, a, :, :] += tensordot(t_old, eri_bdc, axes=([1, 3], [2, 1]))
```

The following will refer to the contraction path above as our numpy.tensordot routine. Note, however, that this is not purely a numpy.tensordot computation, but an iterative call of the numpy.tensordot method to calculate eq 6 to prevent the creation of v^4 intermediate tensors.

4. A MODULAR IMPLEMENTATION OF TENSOR CONTRACTIONS

We have implemented and benchmarked the performance of the above-mentioned tensor contraction routines in PyBEST.[28] Specifically, PyBEST is designed as a modular toolbox where the wave function-specific implementations are decoupled from the linear algebra operations.[28,29] In an actual calculation, the logic in choosing the optimal tensor-contraction scheme is as follows

```
try:
    cupy-accelerated contraction
except NotImplementedError:
    try:
        numpy.tensordot contraction
    except ArgumentError:
        opt_einsum if not available then numpy.einsum(..., optimize="
optimal")
```


https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

The first try statement enforces that the selected tensor contractions are performed on the GPU, if available. If bottleneck-specific contractions are not supported or a CUDA-ready GPU is unavailable, a numpy.tensordot call is performed. Since numpy.tensordot supports only specific contractions featuring nonrepeating indices (that is, repeated indices have to be summed over in numpy.tensordot), an opt_einsum call is performed or, if opt_einsum is not available, an optimized numpy.einsum function call is made instead. Another faster possibility is to default to opt_einsum calls instead of (batched) numpy.tensordot routines based on the expected memory requirements and available resources. The corresponding tensor contraction operation is written using the Einstein-summation convention of the numpy.einsum module; that is, all mathematical operations use an input and output string, where repeated indices are summed over. That allows for one unique notation of mathematical operations independent of the underlying representation of the tensors, especially the ERI. As an example, the bottleneck contraction in eq 6 translates to the string "abcd,ecfd−>eafb", where the first part (abcd) corresponds to the ERI (in their dense representation, which PyBEST internally translates to xac,xbd if Cholesky-decomposed ERI are used; vide infra), the second part (ecfd) to the doubles amplitudes, while the output string (eafb) indicates the order of the output indices of the vector function. Internally, this string is further decoded according to the notation used in the employed LinalgFactory instance, a dense or Cholesky-decomposed representation. If a dense representation of tensors is chosen, the string remains as is. For Cholesky-decomposed ERI, the input argument associated with the Cholesky instance (here "abcd") is translated to the internal Cholesky representation, that is, "xac,xbd."

Since numpy.tensordot only supports a summation over two multidimensional arrays at a time, we need to divide a tensor contraction containing more than two operands into appropriate intermediates. Such partitioning can be fully automated, exploiting the numpy.einsum_path function. It proposes a contraction order of the lowest possible cost for an einsum expression, taking into account the creation of intermediate arrays. The resulting tensordot_helper function has the following logic

```
def tensordot_helper(subscripts, *operands):
    # split subscripts in input and output strings
    inscripts, outscript = split subscripts wrt "->"

    # sanity check
    if contraction cannot be performed:
        raise ArgumentError(error message)

    # get the optimial path
    path, _ = numpy.einsum_path(subscripts, *operands)

    # sometimes einsum_path does not return a list of two-element tuples
    if no optimized path has been found:
        path = provide some path

    # loop over all steps in the path
    for step in path[1:]:
        # take 1st and 2nd operands and corresponding subscripts
        op0, op1 = take the proper arrays from the operands list
        # Find summation axes
        axis0 = list containing axis indices to sum over for op0
        axis1 = list containing axis indices to sum over for op1
        axis = (axis0, axis1)
        # find default outscript
        outscript_ = store shape of intermediate
        # do contraction using numpy.tensordot
        outmat = numpy.tensordot(op0, op1, axes=axis)
        # update the list of operands and subscripts used in the next
    interation
        operands.append(outmat)
        scripts.append(outscript_)
    # do transposition if required.
    if outscript does not have the proper shape:
        trans = tuple how to transpose the output array
        outmat = outmat.transpose(trans)
    # return result of the tensor contraction
    return outmat
```

The subscripts argument is a string specifying the contraction using the numpy.einsum notation, while all multidimensional input arrays are stored in the operands argument. We assume that the ERI corresponds to the leading subscripts. If a tensor contraction cannot be performed, we transition to opt_einsum or numpy.einsum(..., optimize = "optimal") and an ArgumentError is raised. In the case of intermediates that are too large, the tensordot_helper function is replaced by a function call containing selected hand-optimized tensor contraction operations. The implemented flow of contraction operations (CuPy−numpy.tensordot−opt_einsum/numpy.einsum) allows for optimal usage of computational resources, hardware, and multithreaded implementations. The reasons for the proposed operational flow are scrutinized in Section 6. When the performance of the libraries used is improved in future releases, the order of the contraction flavors can be adjusted to maximize efficiency without significantly interfering with the underlying source code in each wave function module.

In the following, all benchmark calculations adhere to the contraction flow mentioned above, if not stated otherwise. That is, most tensor contractions are performed using the numpy.tensordot routine, while the bottleneck contractions are outsourced to the GPU.

5. CUPY AND BATCH-WISE COMPUTATIONS

While, of course, the most performance one could achieve by designing a method specifically for the hardware to be

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

calculated on, we focus on accelerating the mathematical bottleneck operations by exporting the contractions in question to the GPU. As proof of the concept, we exploit CuPy[24] for GPU-accelerated computing. Specifically, it is written as a drop-in replacement for NumPy.[55] For medium- or large-sized molecules, we have to consider the size of the multidimensional arrays present in eq 6 and, therefore, the amount of data that needs to be processed and transferred to the video RAM (VRAM). The principle of memory and processor communication is shown in Figure 1. Due to their size, the underlying multidimensional arrays might be too big to be transferred, processed on the GPU, and transferred back. Thus, a generic implementation, where a NumPy implementation is recycled as a CuPy implementation by replacing, for instance, numpy.einsum with cupy.einsum and taking into account host to device and device to host operations, is impracticable or even impossible.

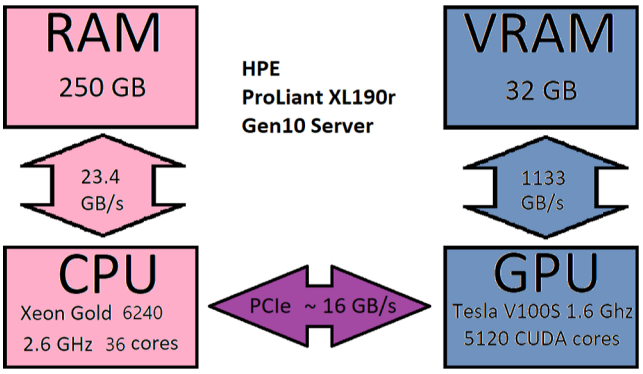


Figure 1. Schematic picture of RAM, VRAM, GPU, and CPU and the data transfer rate between the components.

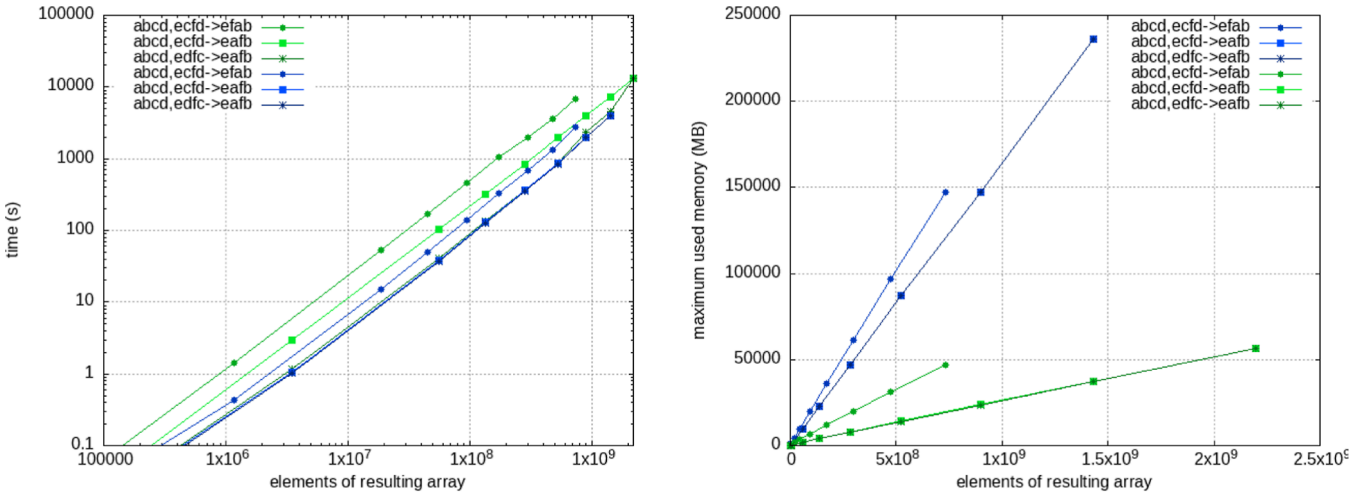
To calculate contractions on the GPU for realistic molecules and large basis sets, it is necessary to do the computations in batches. In our batch-wise computing approach, the arrays on which the operations will be performed must be copied to the VRAM in chunks. To maximize performance and utilize the massive number of CUDA cores, the size of the chunks has to be chosen as big as possible so that as few as possible cycles are needed. To achieve reasonably sized chunks of data to be processed in batches, we multiply the number of elements with their corresponding element size (in bytes) and sum up the required storage space for each multidimensional array involved. If too large, the arrays are split, and the necessary memory is checked again. This process is repeated until the split arrays fit into the VRAM. An example that shows the splitting, allocations, and de-allocations is shown in the following code example for the contraction xac,xbd,ecfd−>eafb

```
t_new = numpy.zeros((t_old.shape[0], L_0.shape[1], t_old.shape[2], L_1.shape
    [1]))
start_e = 0
end_e = 0
# parts_d is the number of parts in which the dense array will be split
for e in range(0, parts_d):
    end_e += dense_e_chunk_lengths[e]
    start_a = 0
    end_a = 0
    # parts_c is the number of parts in which the Cholesky array will be
    split
    for a in range(0, parts_c):
        end_a += chol_chunk_lengths_0[a]
        start_b = 0
        end_b = 0
        # parts_c is the number of parts in which the Cholesky array will be
    split
        for b in range(0, parts_c):
            end_b += chol_chunk_lengths_1[b]

            # Batches of Cholesky arrays are copied to GPU Memory (VRAM)
            chol_0 = cupy.array(numpy.array_split(L_0, parts_c, axis=1)[a])
            chol_1 = cupy.array(numpy.array_split(L_1, parts_c, axis=1)[b])

            # calculating batch of xac,xbd->acbd on GPU
            result_temp = cupy.tensordot(chol_0, chol_1, axes=(0, 0))

            # deallocation
            del chol_0, chol_1
            cupy.get_default_memory_pool().free_all_blocks()

            # Batches of the vector function array is copied to VRAM
            operand = cupy.array(numpy.array_split(t_old, parts_d, axis=0)[e
])

            # calculating batch of acbd,ecfd->abef on GPU
            result_temp_2 = cupy.tensordot(
                result_temp, operand,axes=([1, 3], [1, 3])
            )

            # deallocation
            del operand, result_temp
            cupy.get_default_memory_pool().free_all_blocks()

            # calculating batch of the transposition abef->eafb on GPU
            result_part = cupy.transpose(result_temp_2, axes=(2, 0, 3, 1))

            # deallocation
            del result_temp_2
            cupy.get_default_memory_pool().free_all_blocks()

            # result arrays are copied back to the memory (RAM)
            t_new[
                start_e:end_e, start_a:end_a, :, start_b:end_b
            ] = result_part.get()

            # deallocation
            del result_part
            cupy.get_default_memory_pool().free_all_blocks()

            start_b = end_b
        start_a = end_a
    start_e = end_e
```

6. NUMERICAL RESULTS

6.1. Software Specifications and Computational Details. All calculations were performed on the CentOS 7 operating system using, if not mentioned otherwise, CUDA compilation tools v12.1.105, intelpython v3.9 (referred to as Python in the following), Intel OneAPI v2023.1, and CuPy v12.0.0. Furthermore, we employed NumPy v1.23.5, opt_einsum v3.3.0, and PyBEST v1.4.0dev, which is available on Zenodo[30] or PyPi[32] (the released v1.3.1 includes the CuPy features presented in this work). The hardware on which the

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142



(a) Timings (b) Memory usage

Figure 2. Timings (a) and memory usage (b) of numpy.tensordot and opt_einsum with respect to the size (number of elements) of the largest input array. In both plots, the greenish colors show tensordot and bluish colors show opt_einsum computations using 1 CPU core.

computations were performed is gathered in Section S1 of the Supporting Information. Note that for the benchmark data shown in Figure 2, we used Intel Python 3.7, Intel OneAPI v2021.3, PyBEST v1.3.0, and NumPy v1.21.2.

The molecular structure of the L0 dye was optimized in the ORCA 5.0.3 software package[59−62] using the B3LYP[63,64] exchange−correlation functional and the cc-pVTZ basis set.[65] The resulting structural parameters are provided in Section S2 of the Supporting Information. That molecular structure is subsequently used in orbital-optimized pair CC doubles (pCCD[33,66−70]) calculations, pCCD augmented with the linearized CC singles and doubles (pCCD-LCCSD) correction,[71] and in the conventional CC singles and doubles (CCSD) approach, as implemented in the PyBEST software package.[28,29] In all PyBEST calculations for the L0 dye, we utilized the Cholesky linear algebra factory with a threshold of 10^{-5} for the ERI. In all benchmark calculations concerning timings and memory requirements, the Cholesky vectors are random arrays, where we assume a size of $5K^3$ with K corresponding to the number of basis functions. This roughly corresponds to a Cholesky cutoff threshold of 10^{-5} in the actual molecular calculations.

6.2. Comparison Between CPU- and GPU-Accelerated Implementations. To be able to make assumptions about the benefit of offloading computations to the GPU, it is reasonable to study how effectively different functions described in Section 3 perform compared to each other. In the following, the computation times of numpy.tensordot, opt_einsum, and their CPU multicore processing behavior are investigated and compared with CuPy's tensordot. As mentioned above, a very handy way for implementing tensor contractions is numpy.einsum or opt_einsum, which features a similar syntax. We should note that although numpy.einsum and opt_einsum have similar performance if two operands are contracted with each other, this is not the case anymore if the list of operands contains several multidimensional arrays. In the latter case, opt_einsum may be superior to numpy.einsum in terms of computing time by several orders of magnitude, primarily due to dynamical optimization of contraction paths, the parallelization of the underlying lower-level opt_einsum routines, and efficient batch operations.[72]

Thus, we only show benchmark results for opt_einsum in this work. Furthermore, we investigate three tensor contractions, namely "abcd,ecfd−>eafb", "abcd,edfc−>eafb", and "abcd,ecfd−>efab". The first contraction ("abcd,ecfd−>eafb") corresponds to the bottleneck term of eq 6, while the second one is the associated exchange term, which reads

$$\sum_{xcd} L^{x}_{ad} L^{x}_{bc} t^{cd}_{ij} \tag{7}$$

Although the exchange part of the formal bottleneck contraction is not present when working in a spin-free representation, we benchmark this contraction for reasons of completeness, in case a spin-dependent implementation is sought. The third contraction "abcd,ecfd−>efab" recipe is used in two additional terms of the CCSD vector function. These are one term involving an intermediate $\tilde{t}_{ijkl}$

$$\tilde{t}_{ijkl} = \sum_{xcd} L^{x}_{kc} L^{x}_{ld} t^{cd}_{ij} \tag{8}$$

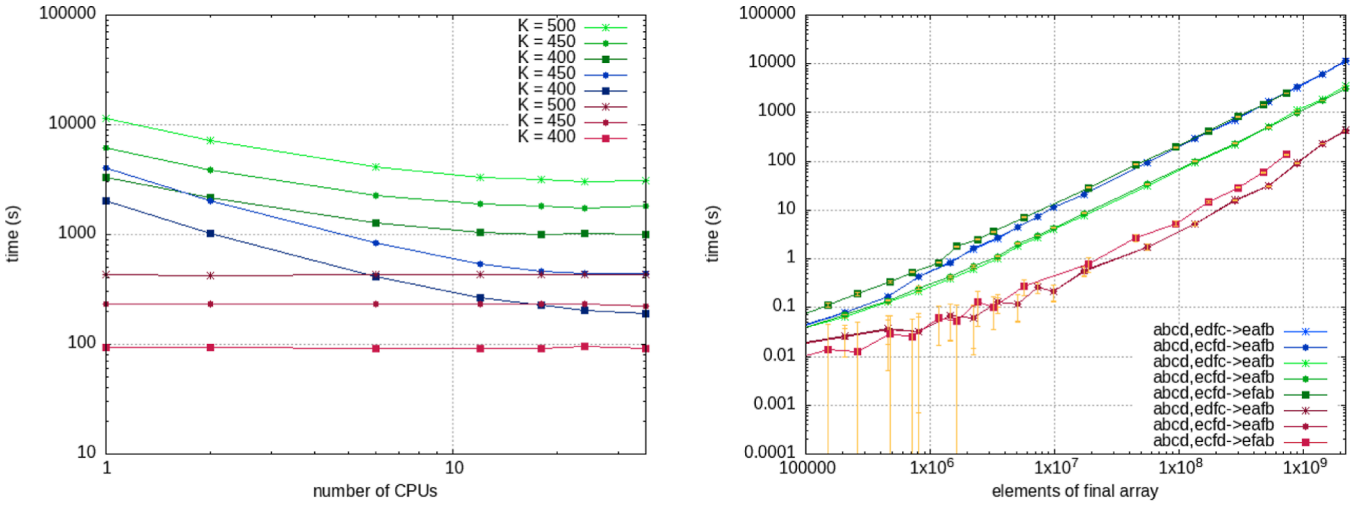
which is an operation of $O(xo^4v)$ complexity, and another one comprising the ERI of $O(ov^3)$

$$\tilde{t}_{ijbk} = \sum_{xcd} L^{x}_{bd} L^{x}_{kc} t^{cd}_{ij} \tag{9}$$

with computational complexity of $O(xo^3v^2)$. Note that exporting eq 8 to the GPU is merely a byproduct of eq 9 as both contractions can be written using the same subscript.

Compared to the for-loop version of numpy.tensordot, the internal optimization leads to a speed-up of a factor of 2 as indicated by Figure 2a. The reason we chose numpy.tensordot as the workhorse of all tensor contractions instead of opt_einsum (or numpy.einsum), is the severe disadvantage of opt_einsum regarding memory efficiency. This drawback becomes evident in the peak memory usage displayed in Figure 2b. The internal optimization and the creation of intermediate arrays (for speed-up) lead to *a* factor of 3 faster-growing memory consumption. That disqualifies opt_einsum as a generic option primarily because we do not want to and often cannot constrain the code to smaller problem sizes. Therefore, we employ numpy.tensordot as the default contraction flavor if

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142



(a) abcd, ecfd → eafb (b) `cupy.tensordot` vs. `numpy.tensordot`

Figure 3. Computing times for CPU and GPU-accelerated implementations. (a) Timings for the bottleneck contraction abcd,ecfd−>eafb. The greenish colors show numpy.tensordot, the bluish colors correspond to opt_einsum, while reddish colors indicate cupy.tensordot results. K is the total number of basis functions. The CuPy results are shown as a guide for the eye and use all available GPU cores. (b) Comparison of GPU and CPU computing times for different basis set sizes K. The numpy.tensordot results for calculations with 1 CPU core are shown in bluish colors and for 36 CPU cores in greenish colors. The reddish colors show the cupy.tensordot results. The error bars in orange are determined for an average of 10 runs and show the standard deviation. For the number of elements of the final array, see the description in the main text and Table 1.

Nvidia CUDA is not available. We should note that opt_einsum features other arguments that limit the memory peak to a specific size. However, this feature comes at the cost of computing time. Specifically, a user-defined limit of the memory peak significantly deteriorates the speed of the numerical operations, which renders opt_einsum impractical for large-scale tensor contractions.

Figure 3a summarizes timings for computations of the contraction "abcd,ecfd−>eafb" (or "xac,xbd,ecfd−>eafb" if the Cholesky vectors are explicitly mentioned) plotted over the number of CPU cores for various tensor contraction flavors and problem sizes N. The greenish colors show numpy.tensordot, the bluish colors opt_einsum, and reddish colors indicate cupy.tensordot timing results for different numbers of CPU cores. Specifically for Figure 3a, the problem size is given by $N = o^2v^2$ because we investigate the following problem setup with dimensions $(xv^2, xv^2, o^2v^2 \rightarrow o^2v^2)$, where v and o are the number of virtual and occupied orbitals, respectively. Their sizes have been set according to the relation $v = \mathrm{int}\left(\frac{3}{4}K\right)$ and $o = K - v$, where K is the total number of basis functions. We should mention that the data for $K = 500$ using opt_einsum could not be obtained due to technical problems, as the memory consumption/memory peak exceeds the available physical memory of the computing node. Overall, the opt_einsum calculations are a factor 3 faster than the corresponding numpy.tensordot variants but limited by their memory peak. However, the for-loop-based numpy.tensordot variant is roughly a factor of 2 slower compared to the contraction scheme where the v^4 intermediate is formed. Thus, opt_einsum and numpy.tensordot are comparable in performance, with the former being modestly faster. Note that the PyBEST project is constantly improved and tested, which means that we are also looking for a more efficient numpy.tensordot CPU implementation, utilizing what is learned from the cupy.tensordot implementation for the GPU, and vice versa. Efficient batching for better memory usage and more effective parallelization are two examples.

Figure 3a further highlights that the benefit of performing a computation on a larger number of cores shrinks very rapidly. The timings reach a plateau at about 8 to 10 cores. This data points to a decreased parallel efficiency of the CPU-based numpy.tensordot and opt_einsum routines. Note, however, that the investigated numpy tensordot scheme comprises one outer for loop, which additionally reduces parallel efficiency. Judging by the numbers, the contraction functions do not really benefit from the usage of more than 10 cores, with 4 cores being the most reasonable amount, assuming the computing time is limited. The bigger the basis set or problem size, the more benefit one gets from utilizing more cores. The cupy.tensordot timings are also shown in Figure 3a for a direct comparison (using all GPU cores). We should note that they are independent of the number of CPU cores (as the mathematical operations are performed on the GPU) and faster than both the NumPy and opt_einsum alternatives. Figure 3b compares the required computing times of CPU and GPU-accelerated contractions. The results corresponding to GPU and CPU timings are taken as an average of over 10 runs. Furthermore, the CPU data was obtained from computations exploiting 1 and 36 CPU cores, respectively. Figure 3b contains three different data sets, namely, one for each investigated contraction "abcd,ecfd−>eafb", "abcd,edfc−>eafb", and "abcd,ecfd−>efab". Note that the notation is internally translated to the Cholesky vectors by replacing the "abcd" part of the string with "xac,xbd". Thus, we always have three input operands in each tensor contraction operation. We should stress that the timings corresponding to all three contraction subscripts are very close to each other, making them almost indistinguishable from each other on the plot. For reasons of comparability, the x-axis shows the problem size, which is given by the size of the resulting (output) tensor. Table 1 summarizes the different problem sizes with respect to the contraction subscript, namely, $N =$

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

Table 1. Problem Sizes N Determined for Different Basis Set Sizes K for the Selected Contractions Benchmarked in This Work[a]

	N			
K	300	400	500	
abcd,ecfd−>eafb	284 765 625	900 000 000	2 197 265 625	o^2v^2
abcd,edfc−>eafb	284 765 625	900 000 000	2 197 265 625	o^2v^2
abcd,ecfd−>efab	94 921 875	300 000 000	732 421 875	o^3v

[a]The last column indicates the size of the output array, while $v = \mathrm{int}\left(\frac{3}{4}K\right)$ and $o = \mathrm{int}\left(\frac{1}{4}K\right)$.

o^2v^2 for "abcd,ecfd−>eafb" and "abcd,edfc−>eafb" and $N = o^3v$ for the contraction "abcd,ecfd−>efab", respectively. The standard deviation for the timings of larger problem sizes is about 1% of the average value, while it increases to approximately 10% for smaller problem sizes. The elements of the arrays used to perform the benchmark calculations were randomly generated. The larger standard deviation for smaller problem sizes and very short time measurements in the order of 10^{-2} are often attributed to small changes in the workload of the operating system.[73] Calculation time of 0.1% is used for host-to-device transfer; therefore, we do not consider PCIe a bottleneck. All in all, we observe an order of magnitude reduction of computing time for the 3 different bottleneck tensor contractions encountered in the CCSD working equation when utilizing the CuPy implementation computed on the NVIDIA Tesla V100S PCIe 32GB (rev 1a) compared to our NumPy implementation computed on 36 CPU cores.

6.3. A Case Study—A Sensitizer Molecule. To check the overall performance in actual chemical problems, we perform example calculations with and without GPU acceleration. Still, the focus is on bottleneck operations. Specifically, we will scrutinize the impact on the overall computing time when just a given set of contractions are offloaded to the GPU. 2-Cyano-3-(4-*N*,*N*-diphenylaminophenyl)-*trans*-acrylic acid, commonly referred to as L0 dye (see Figure 4),[74] has been chosen as the candidate for this performance test. This organic dye was designed to be a potential sensitizer in dye-sensitized solar cells, making it a viable alternative to ruthenium-type dyes. The size of the L0 dye allows us to benchmark the Python-based hybrid CPU−GPU implementation for a chosen parameter set.

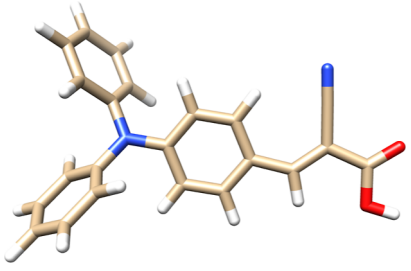


Figure 4. Visualization of molecular structure of the L0 dye relaxed at the B3LYP/cc-pVTZ level of theory.[75]

We tested our CCSD and pCCD-LCCSD implementations by exploiting a cc-pVDZ basis set in our benchmark calculations. All calculations were performed with 36 CPU cores. We should stress that only the 3 bottleneck operations mentioned above are offloaded to the GPU, while the remaining terms in the CC vector function (or the CC iterations step) are performed on the CPU. Table 2 compares different CPU timings in seconds for cupy.tensordot to the timings of our numpy.tensordot variant. As highlighted in Table 2, one iteration step of the vector function takes around 2600 s on the CPU, while the corresponding function requires only 770 s to be evaluated in the case of the hybrid CPU−GPU variant. Note that the average iteration step time shown in Table 2 includes the evaluation of the vector function, the update of the CC amplitudes (using a quasiperturbation-based Newton step exploiting the DIIS algorithm), and the evaluation of the CC energy expression, which is similar for the CPU and hybrid CPU−GPU implementation as those operations are not offloaded to the GPU. The difference in the computing times between the CPU and CPU−GPU implementation of the vector function is the time used by the contractions that were offloaded to the GPU, namely, around 1950 s for the bottleneck contractions per CC iteration step compared to the pure CPU variant. In contrast, the bottleneck contractions offloaded to the GPU require only about 123 s per CC (vector function) iteration step, which is a speed-up of approximately a factor of 16. Comparing the resulting iteration times for the vector function evaluation of the CPU-based numpy.tensordot implementation to the

Table 2. Timings [s] and Speed-Up Factors for the CPU and Hybrid CPU−GPU Implementation for Selected CC Calculations of 2-cyano-3-(4-*N*,*N*-diphenylaminophenyl)-*trans*-acrylic acid (Referred to as "L0") Using a cc-pVDZ Basis Set (444 Basis Functions) with 89 Occupied Orbitals[a]

		CCSD		pCCD-LCCSD	
		NumPy	CuPy	NumPy	CuPy
timings	average iteration step (CPU + GPU)	2754.21	918.25	2692.91	813.80
	average iteration step (CPU)	2754.21	795.28	2692.91	680.00
	average iteration step (GPU)		122.98		133.80
	vector function step (CPU + GPU)	2600.68	766.98	2541.80	662.38
	vector function step (CPU)	2600.68	644.00	2541.80	528.58
	vector function step (GPU)		122.98		133.80
	bottleneck contractions	1956.68	122.98	2013.23	133.80
speed-up	average iteration step	3.0	3.3		
	bottleneck contractions	16.0	15.0		
	vector function step	3.4	3.8		

[a]Iteration step time indicates the time [s] required for one CC iteration step. It contains the evaluation of the vector function, the update of the CC amplitudes, and the evaluation of the CC energy expression. All timings correspond to differences in epoch times. Average iteration step: mean value for the time of one CC iteration averaged over four steps. Vector function step: time of the CPU/GPU part of the vector function averaged over four steps. Bottleneck contractions: time for all bottleneck contractions investigated in this work, that is, those exported to the GPU, averaged over 4 steps.

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

CPU−GPU hybrid variant, 2600.68/766.98 ≈ 3.4, we obtain a total speed-up of approximately a factor of 3. Furthermore, while the bottleneck contractions require about 74% of the computing time per iteration step on the CPU, it drops to approximately 16% of the computing time per iteration using a CPU−GPU hybrid approach.

We observe similar speed-ups for the pCCD-LCCSD method. We should note that our generic GPU implementation also offloads the contraction "abcd,cedf−>aebf" to the GPU in addition to the bottleneck operation "abcd,ecfd−>eafb". The third bottleneck operation "abcd,ecfd−>efab" corresponds to disconnected $\hat{T}_2$ terms and does not show up in the pCCD-LCCSD vector function. This additional tensor contraction corresponds to a term of $O(o^4v^2)$. As expected, the evaluation time of the pCCD-LCCSD vector function takes less time compared to the CCSD implementation, as we exclude the majority of the disconnected terms. Offloading to the GPU reduces the computing time for the selected bottleneck operations from about 2000 to 133 s, which corresponds to a speed-up factor of approximately 15. The overall speed-up for one evaluation of the pCCD-LCCSD vector function drops to a factor of 2541.80/662.38 ≈ 3.8. On average, the bottleneck contractions take about 20% of the computing time of the vector function per iteration step for the CPU−GPU hybrid implementation, while the corresponding time increases to 80% for the CPU variant, which is similar to our CCSD example.

We conclude that offloading the slowest contraction of the form of eq 6 to the GPU already leads to a significant acceleration, shifting the bottleneck to another set of terms in the CC working equations. Additional speed-up can be obtained by offloading other speed-determining tensor contractions to the GPU.

6.4. Comparison to the Bare CUDA Implementation.

In this subsection, we assess the performance of our Python-based GPU acceleration exploiting the CuPy library for a direct CUDA implementation. Specifically, we compare our PyBEST's GPU implementation with the TeraChem[76] benchmark data for the $(H_2O)_{10}$ and $(mU)_2H_2O$ molecules.[77] The molecular structures of the investigated systems are listed in Figure 5. Table 3 collects the numerical data from PyBEST and

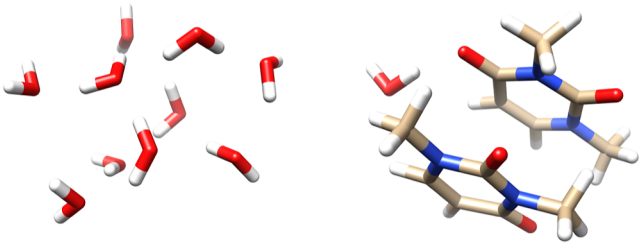


Figure 5. Molecular structures of $(H_2O)_{10}$ (left) and $(mU)_2H_2O$ (right) visualized with UCSF Chimera.[75] XYZ coordinates taken from ref 77.

TeraChem. We should stress that PyBEST uses the Cholesky decomposed integrals and a Tesla V100S, while TeraChem was tested against a Tesla V100. Despite that, we can still make a fair comparison. PyBEST's CPU−GPU implementation is 3.7 and 2.9 times faster than its CPU counterparts, for the $(H_2O)_{10}$ and $(mU)_2H_2O$ molecule, respectively. Based on the bare GPU times, our CuPy implementation is comparable to

Table 3. Average Timings [s] of a CCSD Iteration Step Compared to Computations with a CUDA Implementation[77] and Average GPU Time Used in One Iteration[a]

software	AOs	CPU cores	GPU	CCSD time	average GPU time
$(H_2O)_{10}$, cc-pVDZ, 30 atoms					
Psi4/DF	240	16		16 s	
PyBEST/CD	240	36		337 s	
PyBEST/CD	240	36	Tesla V100S	92 s	4.4 s
TeraChem/CD	240	1	Tesla V100	10 s	10 s
$(mU)_2H_2O$, 6-31+G**, 39 atoms					
PyBEST/CD	468	36		96.5 m	
PyBEST/CD	468	36	Tesla V100S	33.2 m	4.3 m
TeraChem/CD	489	1	Tesla V100	2.5 m	2.5 m

[a]PyBEST results are shown for CPU and CPU−GPU hybrid variants. CD denotes the Cholesky Decomposition and DF density fitting. The Psi4 data is given as a comparison.

the CUDA variant. For the water cluster $(H_2O)_{10}$ (with fewer basis functions), all operations in PyBEST fit into the GPU VRAM. Thus, the batched variant does not feature any Python-based for loops to distribute the numpy.ndarray to the VRAM. Assuming that a GPU-only CuPy version features similar bottleneck operations like the CPU counterpart, the averaged CuPy-based GPU time would be comparable to the CUDA implementation (assuming that the bottleneck operations take 50% of the total computing time to evaluate the vector function). For the larger system $(mU)_2H_2O$, a batching algorithm must be applied. The batching process utilizes Python-based loops, which are generally slow. Thus, PyBEST's CPU−GPU drop in performance for the larger system and the increased average GPU time can be attributed to batching. Based on the water cluster calculation, we can anticipate that the batched CuPy implementation, and hence the for loops, introduce a cost factor of approximately 2−4 compared to the CUDA variant.

7. CONCLUSIONS AND OUTLOOK

Current trends in scientific programming heavily rely on Python-based implementations,[78] which are easy to code but need to be more easily scalable to high-performance computing architectures. At the same time, the potential end-users of these codes would like to work on supercomputers to solve problems as large as possible without profound knowledge of high-performance optimizations. To meet those needs for quantum chemistry problems, we analyzed the limitations of quantum chemistry methods written in Python, where some trade-off between the memory and CPU time has to be made. We showed how to use the existing Python libraries to speed up quantum chemical calculations and provided numerical evidence, including comparisons between CPU and GPU.

A common and reasonable practice to speed up an implementation is to identify the so-called bottleneck operations and focus on optimizing these routines. In this work, we focused on the bottlenecks of CCSD-type methods, which are a given set of tensor contractions and their translation into Python code by using various third-party libraries. Specifically, we scrutinized computing timings and memory consumption, comparing numpy.tensordot and opt_einsum calculations. We found that opt_einsum computes

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

about a factor 2.5 faster than a modified version of numpy.tensordot on a CPU but is limited due to tremendous memory peaks and therefore is not a universal candidate when large system sizes are considered. Furthermore, we have rewritten selected routines based on numpy.tensordot, imposing one for loop to prevent the construction of large intermediate arrays. In general, this for-loop implementation is responsible for the drop in efficiency (speed) of numpy.tensordot compared to opt_einsum by approximately a factor of 2. Nonetheless, the gain in memory reduction outweighs the decrease in speed. Although third-party libraries are convenient to interface and easy to use, special attention has to be paid to possible intermediates created during the evaluation of, for instance, tensor contractions. Both opt_einsum and various NumPy linear algebra routines may create several intermediate arrays to increase performance. These memory outbursts may impede a black-box implementation that is computationally feasible for large-scale problems.

A promising alternative for Pythonic large-scale computing is GPUs. We implemented this set of bottleneck tensor contractions to be calculated on the GPU using CuPy, an application programming interface (API) to Nvidia CUDA for Python. Due to the size of the arrays, it is necessary to perform the computations on the GPU batch-wise by splitting the overall tensor contractions into smaller-sized problems that fit into the VRAM. Utilizing this implementation and performing benchmark calculations of the bottleneck contractions showed that a single GPU already leads to a factor of about 10 speed-up compared to our NumPy-based methods using 36 CPU cores. This factor 10 (or 15 in actual production runs) speed-up of the contraction routines only translates to an overall speed-up of factor 3 as the bottleneck of the CC vector function evaluation has shifted to a collection of terms of similar scaling, which are still evaluated on the CPU. Most importantly, our timing benchmark results are encouraging and again prove the potential of GPU utilization in Python-based computational chemistry. The assessment of our Python-based GPU acceleration against a CUDA implementation indicates a drop in performance because of the need for batching due to insufficient GPU VRAM. The anticipated cost of a batched algorithm (or the introduction of for loops in Python) is a factor of 4.

Subsequently, we will identify the "new" bottleneck operations in tensor contractions and export them to the GPU for additional potential performance or a generic GPU implementation using CuPy. Better utilization of the tensor cores could also lead to improvement, as good speed-ups are reported in developer forums, where FP32 input/output data in deep learning frameworks and HPC can be accelerated, running ten times faster than V100 FP32 FMA operations.[79] Yet, additional speed-up for quantum chemistry problems is expected on the NVIDIA A100 hardware,[80] which has more GPU cores and up to 80 GB of memory. Generally, direct back-end CUDA implementation in C++ also offers plenty of room for improvements, such as optimization of the array structure or concurrent computing. Alternative routes to export Python code to the GPU are, for instance, offered by the Intel oneAPI toolkits.[81] Finally, multi-GPU utilization[82] (for instance, with the NVIDIA NVLink, allowing up to 640 GB of NVRAM) could improve the already tremendous speed-up of factor 10 for tensor contractions.

■ ASSOCIATED CONTENT

Data Availability Statement

The data underlying this study are available in the published article and its Supporting Information and are openly available on Zenodo at https://zenodo.org/records/10069179 and on PyPI at https://pypi.org/project/pybest/.

Supporting Information

The Supporting Information is available free of charge at https://pubs.acs.org/doi/10.1021/acs.jctc.3c01110.

Hardware specifications and *xyz* coordinates of the L0 dye relaxed at the B3LYP/cc-pVTZ level of theory.PyBEST inputs/outputs (PDF)

(ZIP)

■ AUTHOR INFORMATION

Corresponding Authors

Maximilian H. Kriebel − *Institute of Physics, Faculty of Physics, Astronomy, and Informatics, Nicolaus Copernicus University in Toruń, 87-100 Toruń, Poland*; Email: maximilian.kriebel@web.de

Katharina Boguslawski − *Institute of Physics, Faculty of Physics, Astronomy, and Informatics, Nicolaus Copernicus University in Toruń, 87-100 Toruń, Poland*; orcid.org/0000-0001-7793-1151; Email: k.boguslawski@fizyka.umk.pl

Authors

Paweł Tecmer − *Institute of Physics, Faculty of Physics, Astronomy, and Informatics, Nicolaus Copernicus University in Toruń, 87-100 Toruń, Poland*; orcid.org/0000-0001-6347-878X

Marta Gałyńska − *Institute of Physics, Faculty of Physics, Astronomy, and Informatics, Nicolaus Copernicus University in Toruń, 87-100 Toruń, Poland*

Aleksandra Leszczyk − *Institute of Physics, Faculty of Physics, Astronomy, and Informatics, Nicolaus Copernicus University in Toruń, 87-100 Toruń, Poland*

Complete contact information is available at: https://pubs.acs.org/10.1021/acs.jctc.3c01110

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

The research leading to these results has received funding from the Norway Grants 2014−2021 via the National Centre for Research and Development. P.T. acknowledges financial support from the OPUS research grant from the National Science Centre, Poland (grant no. 2019/33/B/ST4/02114). P.T. acknowledges the scholarship for outstanding young scientists from the Ministry of Science and Higher Education. M.G. acknowledges financial support from a Ulam NAWA—Seal of Excellence research grant (no. BPN/SEL/2021/1/00005). Calculations have been carried out using resources provided by Wroclaw Centre for Networking and Supercomputing (http://wcss.pl), grant No. 412.

■ REFERENCES

(1) Schmidt, M. W. The Development of Parallel GAMESS. In *High Performance Computing Systems and Applications*, 1998, pp 243−244.

(2) von Arnim, M.; Ahlrichs, R. Performance of parallel TURBOMOLE for density functional calculations. *J. Comput. Chem.* **1998**, *19*, 1746−1757.
(3) Peng, C.; Calvin, J. A.; Pavosevic, F.; Zhang, J.; Valeev, E. F. Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using TiledArray framework. *J. Phys. Chem. A* **2016**, *120*, 10231−10244.
(4) Garniron, Y.; Applencourt, T.; Gasperich, K.; Benali, A.; Ferté, A.; Paquier, J.; Pradines, B.; Assaraf, R.; Reinhardt, P.; Toulouse, J.; Barbaresco, P.; Renon, N.; David, G.; Malrieu, J.-P.; Véril, M.; Caffarel, M.; Loos, P.-F.; Giner, E.; Scemama, A. Quantum package 2.0: An open-source determinant-driven suite of programs. *J. Chem. Theory Comput.* **2019**, *15*, 3591−3609.
(5) Kowalski, K.; Bair, R.; Bauman, N. P.; Boschen, J. S.; Bylaska, E. J.; Daily, J.; de Jong, W. A.; Dunning, T. J.; Govind, N.; Harrison, R. J.; Keçeli, M.; Keipert, K.; Krishnamoorthy, S.; Kumar, S.; Mutlu, E.; Palmer, B.; Panyala, A.; Peng, B.; Richard, R. M.; Straatsma, T. P.; Sushko, P.; Valeev, E. F.; Valiev, M.; van Dam, H. J. J.; Waldrop, J. M.; Williams-Young, D. B.; Yang, C.; Zalewski, M.; Windus, T. L. From NWChem to NWChemEx: Evolving with the computational chemistry landscape. *Chem. Rev.* **2021**, *121*, 4962−4998.
(6) Ufimtsev, I. S.; Martinez, T. J. Graphical processing units for quantum chemistry. *Comput. Sci. Eng.* **2008**, *10*, 26−34.
(7) Luehr, N.; Ufimtsev, I. S.; Martinez, T. J. Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). *J. Chem. Theory Comput.* **2011**, *7*, 949−954.
(8) 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.
(9) Pham, B. Q.; Alkan, M.; Gordon, M. S. Porting fragmentation methods to graphical processing units using an OpenMP application programming interface: Offloading the Fock build for low angular momentum functions. *J. Chem. Theory Comput.* **2023**, *19*, 2213−2221.
(10) Manathunga, M.; Aktulga, H. M.; Götz, A. W.; Merz, K. M. Quantum mechanics/molecular mechanics simulations on NVIDIA and AMD graphics processing units. *J. Chem. Inf. Model.* **2023**, *63*, 711−717.
(11) Yasuda, K. Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.* **2008**, *29*, 334−342.
(12) 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.
(13) Yasuda, K. Accelerating density functional calculations with graphics processing unit. *J. Chem. Theory Comput.* **2008**, *4*, 1230−1236.
(14) Asadchev, A.; Gordon, M. S. New multithreaded hybrid CPU/GPU approach to Hartree−Fock. *J. Chem. Theory Comput.* **2012**, *8*, 4166−4176.
(15) Barca, G. M.; 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.
(16) Vogt, L.; Olivares-Amaya, R.; Kermes, S.; Shao, Y.; Amador-Bedolla, C.; Aspuru-Guzik, A. Accelerating Resolution-of-the-Identity Second-Order Møller−Plesset Quantum Chemistry Calculations with Graphical Processing Units. *J. Phys. Chem. A* **2008**, *112*, 2049−2057.
(17) Song, C.; Martinez, T. J. Atomic orbital-based SOS-MP2 with tensor hypercontraction. I. GPU-based tensor construction and exploiting sparsity. *J. Chem. Phys.* **2016**, *144*, 174111.
(18) Pham, B. Q.; Carrington, L.; Tiwari, A.; Leang, S. S.; Alkan, M.; Bertoni, C.; Datta, D.; Sattasathuchana, T.; Xu, P.; Gordon, M. S. Porting fragmentation methods to GPUs using an OpenMP API: Offloading the resolution-of-the-identity second-order Møller−Plesset perturbation method. *J. Chem. Phys.* **2023**, *158*, 164115.
(19) Ma, W.; Krishnamoorthy, S.; Villa, O.; Kowalski, K. GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *J. Chem. Theory Comput.* **2011**, *7*, 1316−1327.
(20) Asadchev, A.; Gordon, M. S. Fast and flexible coupled cluster implementation. *J. Chem. Theory Comput.* **2013**, *9*, 3385−3392.
(21) Pototschnig, J. V.; Papadopoulos, A.; Lyakh, D. I.; Repisky, M.; Halbert, L.; Gomes, A. S. P.; Jensen, H. J. A.; Visscher, L. Implementation of relativistic coupled cluster theory for massively parallel GPU-accelerated computing architectures. *J. Chem. Theory Comput.* **2021**, *17*, 5509−5529.
(22) Song, C.; Wang, L.-P.; Sachse, T.; Preiß, J.; Presselt, M.; Martinez, T. J. Efficient implementation of effective core potential integrals and gradients on graphical processing units. *J. Chem. Phys.* **2015**, *143*, 014114.
(23) *CUDA Documentation*, https://docs.nvidia.com/cuda/ (accessed Dec 12, 2023).
(24) Okuta, R.; Unno, Y.; Nishino, D.; Hido, S.; Loomis, C. *CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations*, 2017.
(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) Aquilante, F.; Boman, L.; Boström, J.; Koch, H.; Lindh, R.; de Merás, A. S.; Pedersen, T. B. Cholesky decomposition techniques in electronic structure theory. In *Linear-Scaling Techniques in Computational Chemistry and Physics: Methods and Applications*, 2011; Vol *13*, pp 301−343..
(27) Smith, D. G. A.; Gray, J. opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *J. Open Source Softw.* **2018**, *3*, 753.
(28) Boguslawski, K.; Leszczyk, A.; Nowak, A.; Brzęk, F.; Żuchowski, P. S.; Kędziera, D.; Tecmer, P. Pythonic Black-box Electronic Structure Tool (PyBEST). An open-source Python platform for electronic structure calculations at the interface between chemistry and physics. *Comput. Phys. Commun.* **2021**, *264*, 107933.
(29) Boguslawski, K.; Brzęk, F.; Chakraborty, R.; Cieślak, K.; Jahani, S.; Leszczyk, A.; Nowak, A.; Sujkowski, E.; Świerczyński, J.; Ahmadkhani, S.; Kędziera, D.; Kriebel, M. H.; Żuchowski, P. S.; Tecmer, P. PyBEST: improved functionality and enhanced performance. *Comput. Phys. Commun.* **2024**, *297*, 109049.
(30) *PyBESTv1.3.1 on Zenodo* https://zenodo.org/records/10069179 (accessed Dec 12, 2023).
(31) *PyBEST Webpage* http://fizyka.umk.pl/~pybest (accessed Dec 12, 2023).
(32) *PyBESTc1.3.1 on PyPI* https://libraries.io/pypi/pybest (accessed Dec 12, 2023).
(33) Tecmer, P.; Boguslawski, K. Geminal-based electronic structure methods in quantum chemistry. Toward a geminal model chemistry. *Phys. Chem. Chem. Phys.* **2022**, *24*, 23026−23048.
(34) Boguslawski, K.; Tecmer, P. Orbital entanglement in quantum chemistry. *Int. J. Quantum Chem.* **2015**, *115*, 1289−1295.
(35) Boguslawski, K.; Tecmer, P. Erratum: Orbital entanglement in quantum chemistry. *Int. J. Quantum Chem.* **2017**, *117*, No. e25455.
(36) Tecmer, P.; Boguslawski, K.; Ayers, P. W. Singlet ground state actinide chemistry with geminals. *Phys. Chem. Chem. Phys.* **2015**, *17*, 14427−14436.
(37) Nowak, A.; Legeza, O.; Boguslawski, K. Orbital entanglement and correlation from pCCD-tailored coupled cluster wave functions. *J. Chem. Phys.* **2021**, *154*, 084111.
(38) Leszczyk, A.; Dome, T.; Tecmer, P.; Kedziera, D.; Boguslawski, K. Resolving the π-assisted U−N σ f-bond formation using quantum information theory. *Phys. Chem. Chem. Phys.* **2022**, *24*, 21296−21307.
(39) 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.;

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142

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*, 024109.
(40) Rehn, D. R.; Rinkevicius, Z.; Herbst, M. F.; Li, X.; Scheurer, M.; Brand, M.; Dempwolff, A. L.; Brumboiu, I. E.; Fransson, T.; Dreuw, A.; Norman, P. Gator: A Python-driven program for spectroscopy simulations using correlated wave functions. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2021**, *11*, No. e1528.
(41) *Lightspeed Project* https://github.com/robparrish/lightspeed (accessed Dec 12, 2023).
(42) Turney, J. M.; Simmonett, A. C.; Parrish, R. M.; Hohenstein, E. G.; Evangelista, F. A.; Fermann, J. T.; Mintz, B. J.; Burns, L. A.; Wilke, J. J.; Abrams, M. L.; Russ, N. J.; Leininger, M. L.; Janssen, C. L.; Seidl, E. T.; Allen, W. D.; Schaefer, H. F.; King, R. A.; Valeev, E. F.; Sherrill, C. D.; Crawford, T. D. Psi4: an open-source ab initio electronic structure program. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2012**, *2*, 556−565.
(43) Smith, D. G. A.; Burns, L. A.; Simmonett, A. C.; Parrish, R. M.; Schieber, M. C.; Galvelis, R.; Kraus, P.; Kruse, H.; Di Remigio, R.; Alenaizan, A.; James, A. M.; Lehtola, S.; Misiewicz, J. P.; Scheurer, M.; Shaw, R. A.; Schriber, J. B.; Xie, Y.; Glick, Z. L.; Sirianni, D. A.; O'Brien, J. S.; Waldrop, J. M.; Kumar, A.; Hohenstein, E. G.; Pritchard, B. P.; Brooks, B. R.; Schaefer, H. F.; Sokolov, A. Y.; Patkowski, K.; DePrince, A. E.; Bozkaya, U.; King, R. A.; Evangelista, F. A.; Turney, J. M.; Crawford, T. D.; Sherrill, C. D. PSI4 1.4: Open-source software for high-throughput quantum chemistry. *J. Chem. Phys.* **2020**, *152*, 184108.
(44) Kim, T. D.; Richer, M.; Sánchez-Díaz, G.; Miranda-Quintana, R. A.; Verstraelen, T.; Heidar-Zadeh, F.; Ayers, P. W. Fanpy: A python library for prototyping multideterminant methods in ab initio quantum chemistry. *J. Comput. Chem.* **2023**, *44*, 697−709.
(45) Lemmens, L.; De Vriendt, X.; Tolstykh, D.; Huysentruyt, T.; Bultinck, P.; Acke, G. GQCP: The Ghent Quantum Chemistry Package. *J. Chem. Phys.* **2021**, *155*, 084802.
(46) Cizek, J. On the Correlation Problem in Atomic and Molecular Systems. Calculation of Wavefunction Components in Ursell-Type Expansion Using Quantum-Field Theoretical Methods. *J. Chem. Phys.* **1966**, *45*, 4256−4266.
(47) Cizek, J.; Paldus, J. Correlation problems in atomic and molecular systems III. Rederivation of the coupled-pair many-electron theory using the traditional quantum chemical methodst. *Int. J. Quantum Chem.* **1971**, *5*, 359−379.
(48) Bartlett, R. J. Many-Body Perturbation Theory and Coupled Cluster Theory for Electron Correlation in Molecules. *Annu. Rev. Phys. Chem.* **1981**, *32*, 359−401.
(49) Bartlett, R. J.; Musiał, M. Coupled-cluster theory in quantum chemistry. *Rev. Mod. Phys.* **2007**, *79*, 291−352.
(50) Helgaker, T.; Jørgensen, P.; Olsen, J. *Molecular Electronic-Structure Theory*; Wiley: New York, 2000.
(51) Shavitt, I.; Bartlett, R. J. *Many-Body Methods in Chemistry and Physics*; Cambridge University Press: New York, 2009.
(52) Koch, H.; Sánchez de Merás, A.; Pedersen, T. B. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.* **2003**, *118*, 9481−9484.
(53) Smith, D. G. A.; Burns, L. A.; Sirianni, D. A.; Nascimento, D. R.; Kumar, A.; James, A. M.; Schriber, J. B.; Zhang, T.; Zhang, B.; Abbott, A. S.; Berquist, E. J.; Lechner, M. H.; Cunha, L. A.; Heide, A. G.; Waldrop, J. M.; Takeshita, T. Y.; Alenaizan, A.; Neuhauser, D.; King, R. A.; Simmonett, A. C.; Turney, J. M.; Schaefer, H. F.; Evangelista, F. A.; DePrince, A. E. I.; Crawford, T. D.; Patkowski, K.; Sherrill, C. D. Psi4NumPy: An interactive quantum chemistry programming environment for reference implementations and rapid development. *J. Chem. Theory Comput.* **2018**, *14*, 3504−3511.
(54) Lawson, C. L.; Hanson, R. J.; Kincaid, D. R.; Krogh, F. T. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math Software* **1979**, *5*, 308−323.
(55) *NumPy Developers Blog* https://numpy.org (accessed Jan 11, 2024).
(56) *OpenBLAS Releases*, https://github.com/OpenMathLib/OpenBLAS/releases (accessed Jan 11, 2024).
(57) *Intel Math Kernel Library Release Notes and New Features*, https://www.intel.com/content/www/us/en/developer/articles/release-notes/onemkl-release-notes.html (accessed Jan 11, 2024).
(58) Whaley, R. C.; Petitet, A.; Dongarra, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* **2001**, *27*, 3−35.
(59) Neese, F. The ORCA program system. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2012**, *2*, 73−78.
(60) Neese, F. Software update: the ORCA program system, version 4.0. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2018**, *8*, No. e1327.
(61) Neese, F.; Wennmohs, F.; Becker, U.; Riplinger, C. The ORCA quantum chemistry program package. *J. Chem. Phys.* **2020**, *152*, 152.
(62) Neese, F. Software update: The ORCA program system—Version 5.0. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2022**, *12*, No. e1606.
(63) 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.
(64) Becke, A. D. Density-functional thermochemistry. III. The role of exact exchange. *J. Chem. Phys.* **1993**, *98*, 5648−5652.
(65) Dunning, T. Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. *J. Chem. Phys.* **1989**, *90*, 1007−1023.
(66) Limacher, P. A.; Ayers, P. W.; Johnson, P. A.; De Baerdemacker, S.; Van Neck, D.; Bultinck, P. A New Mean-Field Method Suitable for Strongly Correlated Electrons: Computationally Facile Antisymmetric Products of Nonorthogonal Geminals. *J. Chem. Theory Comput.* **2013**, *9*, 1394−1401.
(67) Boguslawski, K.; Tecmer, P.; Ayers, P. W.; Bultinck, P.; De Baerdemacker, S.; Van Neck, D. Efficient description of strongly correlated electrons with mean-field cost. *Phys. Rev. B* **2014**, *89*, 201106.
(68) Stein, T.; Henderson, T. M.; Scuseria, G. E. Seniority Zero Pair Coupled Cluster Doubles Theory. *J. Chem. Phys.* **2014**, *140*, 214113.
(69) Boguslawski, K.; Tecmer, P.; Bultinck, P.; De Baerdemacker, S.; Van Neck, D.; Ayers, P. W. Non-Variational Orbital Optimization Techniques for the AP1roG Wave Function. *J. Chem. Theory Comput.* **2014**, *10*, 4873−4882.
(70) Limacher, P. A.; Kim, T. D.; Ayers, P. W.; Johnson, P. A.; De Baerdemacker, S.; Van Neck, D.; Bultinck, P. The influence of orbital rotation on the energy of closed-shell wavefunctions. *Mol. Phys.* **2014**, *112*, 853−862.
(71) Boguslawski, K.; Ayers, P. W. Linearized Coupled Cluster Correction on the Antisymmetric Product of 1-Reference Orbital Geminals. *J. Chem. Theory Comput.* **2015**, *11*, 5252−5261.
(72) *Opteinsum Documentation*, https://optimized-einsum.readthedocs.io/en/stable/ (accessed Dec 12, 2023).
(73) Psarras, C.; Karlsson, L.; Li, J.; Bientinesi, P. *The Landscape of Software for Tensor Computations*; arXiv preprint 2021, arXiv, 2103, p 13756.
(74) Kitamura, T.; Ikeda, M.; Shigaki, K.; Inoue, T.; Anderson, N. A.; Ai, X.; Lian, T.; Yanagida, S. Phenyl-Conjugated Oligoene Sensitizers for TiO_2 Solar Cells. *Chem. Mater.* **2004**, *16*, 1806−1812.
(75) Pettersen, E. F.; Goddard, T. D.; Huang, C. C.; Couch, G. S.; Greenblatt, D. M.; Meng, E. C.; Ferrin, T. E. UCSF Chimera—a visualization system for exploratory research and analysis. *J. Comput. Chem.* **2004**, *25*, 1605−1612.
(76) Seritan, S.; Bannwarth, C.; Fales, B. S.; Hohenstein, E. G.; Isborn, C. M.; Kokkila-Schumacher, S. I. L.; Li, X.; Liu, F.; Luehr, N.; Snyder, J. W.; Song, C.; Titov, A. V.; Ufimtsev, I. S.; Wang, L.-P.; Martínez, T. J. TeraChem: A graphical processing unit-accelerated electronic structure package for large-scale ab initio molecular dynamics. *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2021**, *11*, No. e1494.
(77) Fales, B. S.; Curtis, E. R.; Johnson, K. G.; Lahana, D.; Seritan, S.; Wang, Y.; Weir, H.; Martínez, T. J.; Hohenstein, E. G.

Performance of coupled-cluster singles and doubles on modern stream processing architectures. *J. Chem. Theory Comput.* **2020**, *16*, 4021−4028.
(78) Ryzhkov, F. V.; Ryzhkova, Y. E.; Elinson, M. N. Python in Chemistry: Physicochemical Tools. *Processes* **2023**, *11*, 2897.
(79) *NVIDIA Developers Blog* https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/ (accessed Jan 11, 2024).
(80) Hohenstein, E. G.; Fales, B. S.; Parrish, R. M.; Martinez, T. J. Rank-reduced coupled-cluster. III. Tensor hypercontraction of the doubles amplitudes. *J. Chem. Phys.* **2022**, *156*, 156.
(81) *Intel OneAPI Toolkit Documentation*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-documentation.html (accessed Dec 12, 2023).
(82) Johnson, K. G.; Mirchandaney, S.; Hoag, E.; Heirich, A.; Aiken, A.; Martínez, T. J. Multinode multi-GPU two-electron integrals: Code generation using the regent language. *J. Chem. Theory Comput.* **2022**, *18*, 6522−6536.

https://doi.org/10.1021/acs.jctc.3c01110
J. Chem. Theory Comput. 2024, 20, 1130−1142