Accelerating Quantum Chemistry with Vectorized and Batched Integrals

Hua Huang
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
huangh223@gatech.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia
echow@cc.gatech.edu

***Abstract*—This paper presents the first quantum chemistry calculations using a recently developed vectorized library for computing electron repulsion integrals. To lengthen the SIMD loop and thus improve SIMD utilization, the approach used in this paper is to batch together multiple integrals that can be computed using the same code path. The standard approach is to compute integrals one at a time, and thus a batching procedure had to be developed. This paper shows proof-of-concept and demonstrates the performance gains possible when the batched approach is used. Batching also enables certain optimizations when the integrals are used to compute the Fock matrix. We further describe several other optimizations that were needed to obtain up to a 270% speedup over the non-batching version of the code, making a compelling case for adopting the presented techniques in quantum chemistry software.**

***Index Terms*—electron repulsion integral, vectorization, batching, Hartree-Fock, quantum chemistry**

I. Introduction

Quantum chemistry is a mature area of computational science with many methods and codes developed that are used across chemistry, biochemistry, and materials science. The main kernels in quantum chemistry are electron repulsion integral (ERI) calculations, eigenvalue computations, and tensor contractions. Among these three kernels, ERI calculations are the most irregular and challenging to optimize. In particular, the ERI algorithms are recursive and loops have small trip counts, making efficient vectorization very difficult.

Previous attempts to improve the vector performance of ERI libraries took the approach of restructuring loops to make them vectorizable, annotating code to help the compiler auto-vectorizer, as well as hand-coding with intrinsics [1], [2]. However, the resulting vectorization efficiency was still poor, because the recursions and short loops of the ERI algorithms could not be addressed without a drastic code transformation or rewrite.

Recently, a library for ERI calculations, called Simint [3], was developed with the goal obtaining high vector efficiency. Simint is based on the idea of simultaneously computing the components of an ERI, called *primitive integrals*, that have the same code path. Benchmarks showed superior performance over existing ERI libraries and this performance improvement could be directly attributed to improved vector performance.

However, ERIs may only have a single or small number of primitive integrals, which would give Simint little or no advantage over other ERI libraries in this case. The solution here is to batch together the computation of multiple ERIs, simultaneously computing primitive integrals for these multiple ERIs, as long as the computation of these primitive integrals share the same code path. With enough primitive integrals, the vector loop computing the primitive integrals can become much more efficient.

This paper makes the following contributions:

1) In Section IV, we present optimizations for Simint, including a non-intuitive way to perform vector reductions, and a sorting procedure to improve vector utilization in the presence of primitive screening (neglect of small primitive integrals).
2) In Section V, we present an algorithm for batching ERIs and in Section VII we demonstrate the overall performance improvement that this batching approach can have in a quantum chemistry calculation.
3) In Section VI, we present efficient techniques for constructing the Fock matrix in the context of multiple threads updating a shared matrix. One of these techniques arises from the fact that the ERIs are batched.
4) Previous performance studies of Simint used microbenchmarks that only timed the ERI calculation. Our tests in Section VII use Simint in a Hartree-Fock application, giving a more realistic picture of performance, while allowing a wider range of optimizations.

II. Background

The solution to a quantum chemistry problem can be expressed in terms of basis functions ϕ_i, with $1 \leq i \leq n$ for n basis functions. "Gaussian" basis functions are most common, particularly for molecular systems, and are the target of this work, although other types of basis functions, such as plane waves and wavelets [4], are also used, particularly for materials systems.

An ERI describes the Coulombic interaction between two electrons in terms of four basis functions

$$(AB|CD) = \int \phi_A(\vec{x}_1)\phi_B(\vec{x}_1)\frac{1}{r_{12}}\phi_C(\vec{x}_2)\phi_D(\vec{x}_2)d\vec{x}_1 d\vec{x}_2$$

where the $(AB|CD)$ is notation denoting a specific ERI and r_{12} is the distance between the two electrons at $\vec{x}_1$ and $\vec{x}_2$.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

Each basis function is generally a linear combination (or "contraction") of known *primitive functions*, $\chi_{A\mu}$, i.e.,

$$\phi_A = \sum_{\mu}^{K_A} c_{A\mu}\chi_{A\mu}$$

(similarly for the other functions ϕ_B, ϕ_C, and ϕ_D) and thus the integral, which can be called a *contracted integral* for clarity, is the result of a four-fold sum,

$$(AB|CD) = \sum_{\mu}^{K_A}\sum_{\nu}^{K_B}\sum_{\lambda}^{K_C}\sum_{\sigma}^{K_D} c_{A\mu}c_{B\nu}c_{C\lambda}c_{D\sigma}[\chi_{A\mu}\chi_{B\nu}|\chi_{C\lambda}\chi_{D\sigma}]$$

where $[\chi_{A\mu}\chi_{B\nu}|\chi_{C\lambda}\chi_{D\sigma}]$ in square brackets denotes a *primitive integral*. It is these primitive integrals involving recursive calculations that are difficult to vectorize. In the above example, the contracted integral is the sum of $K_AK_BK_CK_D$ primitive integrals. Note that, nominally, n^4 ERIs must be computed, a potentially very large number. Neglect of small ERIs, called *screening*, and exploiting symmetries such as $(AB|CD) = (BA|CD) = (CD|BA)$ are necessary in practical implementations.

A *basis set* is a specification of the basis functions to use for different atomic species in a molecule. Some basis sets are *highly contracted*, meaning its basis functions are sums of many primitive functions (up to a dozen or more), whereas others are *lightly contracted*, meaning its basis functions are mostly represented by a single primitive function.

A basis function is characterized by its *angular momentum* (AM), which can take on values $0,1,2,3,4,\ldots$, denoted as $s,p,d,f,g,\ldots$, respectively. The primitive functions comprising a basis function have the same AM as the basis function. Since an integral is associated with four basis functions, an integral is also associated with four AM numbers, e.g., $(ss|pd)$, where the bracket notation has been overloaded, and denotes the AM *class* of an integral. Important for this paper, primitive integrals of the same class can be computed by the same code path, and can thus be computed simultaneously in vectorized fashion.

The Simint library vectorizes the computation of primitive integrals of the same class. For the computation of the specific ERI $(AB|CD)$, the number of primitive integrals is $K_AK_BK_CK_D$. Three cases are illustrated in Figure 1. For lightly contracted basis sets (a), the number of primitive integrals may be as small as 1, and thus no vectorization is possible. Even for moderately contracted basis sets (b), SIMD utilization may be poor because a large proportion of the integrals is calculated outside the SIMD loop. For highly contracted basis sets (c), SIMD utilization is good. Thus, for good performance for all basis sets, particularly lightly contracted basis sets, we need to batch together *contracted integrals* of the same class, computing their primitive integrals in the same SIMD loop.

Basis functions related in a specific way are grouped into sets called *shells*. Four shells M, N, P, Q, are called a *shell quartet* and define a set of integrals $(MN|PQ)$. Integral libraries always compute ERIs in these shell quartet sets in order to reuse intermediate computations, but computation of the ERI in these sets is not naturally vectorizable.

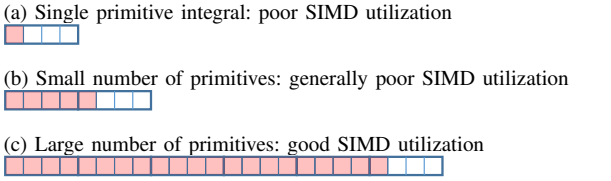

Fig. 1. Different numbers of primitive integrals calculated using SIMD vectorization using the example of 4 doubles per SIMD word. (Primitive integrals occupying lanes of a SIMD word are shown with red shading.) For high SIMD utilization, a large number of primitive integrals of the same AM class are necessary.

III. Related Work

Vectorization of ERI calculations has been considered since the era of pipelined vector supercomputers. On the CRAY-1, Saunders and Guest [5] proposed vectorizing primitive integral calculations for low AM cases, where the shell quartet structure, which complicates vectorization, could be ignored without too high a cost. For the high AM cases, it was suggested to vectorize the contractions. On the Alliant FX-8, Gill, Head-Gordon, and Pople [6] also proposed vectorizing primitive integrals. Here, contractions could be performed early or late in the algorithm depending on AM class. In the mid 1990s, Wolinski et al. [7] proposed batching contracted integrals in the TEXAS code to improve performance in the same vein as using BLAS constructs.

The above programs are legacy Fortran codes using common blocks. The TEXAS integral module, however, was recently converted to be used in a multithreaded environment [8] and is still the default integral library in NWChem [9]. Despite its design, however, Shan et al. [1] found vectorization performance to be very poor: "Via substantial programming effort, we obtained a vectorized version running approximately 25% faster compared to non-vectorization mode on the MIC and BG/Q platforms."

Libint2 [10], one of the integral libraries used in GAMESS [11], has experimental capability to vectorize across contracted integrals, but does not appear to be used this way. Vectorizing across contracted integrals poses several challenges, including the implementation of primitive integral screening, much larger working set size than vectorizing across primitive integrals, and needing to not only batch integrals with the same AM class but also the same contraction pattern, K_A, K_B, K_C, K_D.

Recently developed integral libraries have not focused on vectorization. Sun [12] provides comprehensive functionality although an expensive inner product in the Rys quadrature method was vectorized. Zhang [13] uses tree search to find efficient recursions and also uses a meta-algorithm approach, applying the best algorithms for each AM class.

In the last decade, there has also been extensive interest in computing ERIs on GPUs [14] [15] [16] [17] [18] [19]. Here, research addressed the use of single precision, the host-device bottleneck, and whether to vectorize primitive or contracted integrals. Some of the codes developed are limited to low AM functions as limited device memory makes simultaneous computation of large numbers of high AM integrals very

challenging. These works also appeared to ignore practical issues such as integral screening and exploitation of symmetry, which would introduce thread divergence.

Ramdas and co-authors [20]–[22] discuss the advantages of batching ERIs of the same AM class, such as instruction and data cache exploitation, and propose batching algorithms that can be implemented on FPGAs, but there are few details on actual implementations.

IV. Simint Performance Optimizations

Simint is a vectorized implementation of the Obara-Saika (OS) algorithm [6], [23]–[27] for computing ERIs. In the OS algorithm, an integral with AM class $(ij|kl)$ can be computed using recurrence relations involving *auxiliary* integrals with lower AM class, i.e., involving functions with lower AM. To give a conceptual but not precise description below, we denote an auxiliary integral of class $(ij|kl)$ as $\Theta^{(N)}_{ijkl}$ where the index N equals 0 for the desired target integral.

To compute $\Theta^{(0)}_{ijkl}$, multi-dimensional recurrence relations are used, where the base of the recursions are $\Theta^{(N)}_{0000}$ for any value of N, which are called Boys functions of order N, and which can be computed directly. The recurrence relations can be organized into vertical recurrence relations (VRR) and horizontal recurrence relations (HRR) and can be used as follows.

To compute $\Theta^{(0)}_{ijkl}$, one begins by computing $\Theta^{(m)}_{i+j,0,0,0}$ via a recurrence relation known as a bra-side (i.e., left-hand side) VRR, and then computing $\Theta^{(m)}_{i+j,0,k+l,0}$ via a ket-side (i.e., right-hand side) VRR, followed by computing $\Theta^{(m)}_{i,j,k+l,0}$ via a bra-side HRR, and finally by computing $\Theta^{(m)}_{i,j,k,l}$ by a ket-side HRR. For reference, these recurrence relations are:
Bra-side VRR:

$$\Theta^{(N)}_{i+1,0,0,0} = X_{PA}\Theta^{(N)}_{i,0,0,0} - \frac{\alpha}{p}X_{PQ}\Theta^{(N+1)}_{i,0,0,0} + \frac{i}{2p}\left(\Theta^{(N)}_{i-1,0,0,0} - \frac{\alpha}{p}\Theta^{(N+1)}_{i-1,0,0,0}\right)$$

Ket-side VRR:

$$\Theta^{(N)}_{i,0,k+1,0} = X_{QC}\Theta^{(N)}_{i,0,k,0} - \frac{\alpha}{q}X_{PQ}\Theta^{(N+1)}_{i,0,k,0} + \frac{k}{2q}\left(\Theta^{(N)}_{i,0,k-1,0} - \frac{\alpha}{q}\Theta^{(N+1)}_{i,0,k-1,0}\right) + \frac{i}{2(p+q)}\Theta^{(N+1)}_{i-1,0,k,0}$$

Bra-side HRR:

$$\Theta^{(N)}_{i,j+1,k,l} = \Theta^{(N)}_{i+1,j,k,l} + X_{AB}\Theta^{(N)}_{i,j,k,l}$$

Ket-side HRR:

$$\Theta^{(N)}_{i,j,k,l+1} = \Theta^{(N)}_{i,j,k+1,l} + X_{CD}\Theta^{(N)}_{i,j,k,l}.$$

In the above, X_{PA}, etc., are parameters that depend on the atomic coordinates of the molecular system, and p, q, α, are parameters of the basis functions. The index m above denotes an appropriate range of indices for auxiliary integrals that must be computed.

Listing 1. General bra-side HRR function

```
void HRR_J_f_d(const int ncart, double *hAB,
  double *fdXX, double *gpXX, double *fpXX) {
  for (int iket = 0; iket < ncart; ++iket) {
    fdXX[0*ncart+iket] = gpXX[0*ncart+iket]
                + hAB[0]*fpXX[0*ncart+iket];
    fdXX[1*ncart+iket] = gpXX[3*ncart+iket]
                + hAB[1]*fpXX[0*ncart+iket];
    fdXX[2*ncart+iket] = gpXX[6*ncart+iket]
                + hAB[2]*fpXX[0*ncart+iket];
    fdXX[3*ncart+iket] = gpXX[4*ncart+iket]
                + hAB[1]*fpXX[1*ncart+iket];
    // more calculations (no simple pattern)
  }
}
```

Simint applies the VRRs to SIMD words rather than regular double-precision words, i.e., to 8 doubles at a time in the case of AVX-512. One could also apply the HRRs to SIMD words, however, since the HRRs do not involve parameters that depend on the basis functions, the primitive integrals can be contracted at this point, before the HRRs, thus saving computation and storage. Thus the HRRs are not perfectly vectorized, but some auto-vectorization of the bra-side HRR is possible. Consequently, the vector efficiency of Simint depends on the AM class, with classes involving more VRRs than HRRs being more efficient.

A. Optimizing General Functions for High AM Integrals

Simint code is generated, with separate functions for computing integrals of each AM class. For low AM classes, the recursive operations are expanded explicitly and inlined. For high AM classes, in order to reduce code size, general functions are used to implement the VRRs and HRRs involving high AM functions. The following two optimizations are simple, although they could be easily missed, and give some performance improvement.

1) General Bra-side HRR Functions: As an example, the Simint function *HRR_J_f_d()* computes auxiliary integrals of class $(fd|**)$ from those of classes $(fp|**)$ and $(gp|**)$ using a bra-side HRR. These functions can be auto-vectorized by the compiler, but timings show that some AM classes with only VRRs and bra-side HRRs have poor vectorization speedup, and that the general bra-side HRR functions consume most of the time. These general bra-side HRR functions accept a parameter, *ncart*, to specify the number of basis functions in a shell on the ket side, and this parameter determines how data is accessed (see Listing 1). If this parameter value is known at compile time, auto-vectorization could be improved.

The parameter *ncart* only takes on a small set of values. Therefore, we have created specialized versions of *HRR_J_f_d()* and other functions for specific values of *ncart*. These specialized versions are called from a wrapper function. The general version is kept for large values of *ncart*, which are rare. As a result, the compiler can compute all the offsets of output auxiliary integrals and know the length of the loop at compile time, which reduces the cost of the loop and leads to better vectorization.

Listing 2. contract_all() implementation for AVX-512

```
inline void contract_all(int ncart,
    __m512d const *src, double *dest) {
    for (int np = 0; np < ncart; np++)
        dest[np] += _mm512_reduce_add_pd(src[np]);
}
```

2) General VRR Functions: In Simint, the function *ostei_general_vrr_K()* implements a ket-side VRR. Profiling found that this function is a hotspot when the basis set has many high AM shells. This function is implemented in a general way to open algorithmic options: it can compute $(i,j|k+1,l)$ for any given $i,j,k,l \geq 0$. However, the ket-side VRR in the OS algorithm assumes targets of the form $(i,0|k+1,0)$. By specializing this function for this case, branches that determine which recursions to take can be reduced from 16 to 4 and can also be moved from the inner-most loop to an outer loop. The original general function is retained for use with different algorithmic options.

B. Optimizing Contractions for AVX-512

The contraction operation sums primitive auxiliary integrals to form a contracted auxiliary integral. In Simint, the *contract_all()* function performs, for a set of SIMD words, a horizontal reduction of double-precision words in a SIMD word, as shown in Listing 2. Each SIMD word corresponds to a different auxiliary integral, not the same integral. The *contract_all()* function was measured to be a computational hotspot.

In the AVX-512 case shown in Listing 2, *contract_all()*, naturally uses the *_mm512_reduce_add_pd()* intrinsic function to sum the components of a SIMD vector. However, analyzing the assembly code for *contract_all()* reveals two problems: (1) *_mm512_reduce_add_pd()* does not have a corresponding CPU instruction, but is emulated by 8 instructions; (2) when $ncart > 10$, the compiler does not unroll the loop.

Considering that the AVX-512 instruction set has new shuffle instructions and that the Intel compilers can utilize these instructions efficiently [28] [29], we use a novel approach to accelerate the *contract_all()* function. For a 8×8 block of double-precision words (8 SIMD words), we first transpose the block, then perform vectorized add for the transposed vertical vectors. Listing 3 is the new, optimized implementation. Lines 10 - 12 transpose a 8×8 block, and line 13 effectively hints to the compiler that the transposed data will be used immediately and should be kept in registers. As a quick comparison, for a 8×8 block, the optimized implementation needs 47 CPU instructions, while the original approach needs $16 \times 8 = 128$ CPU instructions. Since *ncart* is not always a multiple of 8, we use the original approach for the remainder part.

We also make a special optimization for AM class $(ss|ss)$, since *contract_all()* for $(ss|ss)$ is a large portion of the runtime when using basis sets with few high AM functions. The subroutine for computing $(ss|ss)$ calls *contract_all()* with $ncart = 1$, so the remainder part in Listing 3 would be used. Instead, to use the new approach, we gather 8 SIMD words

Listing 3. Optimized contract_all() implementation for AVX-512

```
inline void contract_all(int ncart,                                  1
    __m512d const *src, double *dest) {                              2
    int ntrans   = ncart / 8;                                        3
    int np_start = ntrans * 8;                                       4
    double tmp[64];                                                  5
    __m512d dst[8];                                                  6
    // Transpose-Add part                                            7
    for (int it = 0; it < ntrans; it++) {                            8
        double *src_ptr = (double*)src + it * 64;                    9
        for (int i = 0; i < 8; i++) {                                10
            for (int j = 0; j < 8; j++)                              11
                tmp[i * 8 + j] = src_ptr[j * 8 + i];                 12
            dst[i] = _mm512_load_pd(tmp + i * 8);                    13
        }                                                            14
        __m512d res = _mm512_loadu_pd(dest + it * 8);                15
        for (int i = 0; i < 8; i++)                                  16
            res = _mm512_add_pd(res, dst[i]);                        17
        _mm512_storeu_pd(dest + it * 8, res);                        18
    }                                                                19
    // Remainder part                                                20
    for (int np = np_start; np < ncart; np++)                        21
        dest[np] += _mm512_reduce_add_pd(src[np]);                   22
}                                                                    23
```

corresponding to the same contracted integral, then store the results separately. It should be noted that this gather-contract-store approach can also be used for other AM classes, but it would make the code much more complicated, and the cost of extra operations in these cases may cancel out the saved time.

C. Sorting for Primitive Screening

Primitive screening is the concept of neglecting the computation of primitive integrals when they are known to be small. Since the computation of the primitive integrals is vectorized, primitive screening creates "holes" (screened primitives) in the SIMD words, which are not handled efficiently. Simint only neglects the computation of primitive integrals if all primitives in a SIMD word are screened. To improve vectorization efficiency, we sort all primitives in a shell pair in descending order according to an upper bound based on the Cauchy-Schwarz inequality. If a primitive involving a shell pair is screened, all primitives behind it will also be screened. As a result, all neglected primitives will be placed together, which increases the probability that all primitives in a SIMD word are screened. Figure 2 shows the mechanism of sorting for primitive screening in Simint.

V. Batching ERI Calculations to Improve Vectorization

In distributed memory quantum chemistry codes, the set of shell quartets to be computed is partitioned statically or dynamically among the compute nodes. Within a node, each thread computes and consumes the integrals for a shell quartet, one shell quartet at a time. As described in Section II, SIMD utilization in Simint for computing ERIs may be poor when shell quartets are computed one at a time. In this Section, we describe a batching procedure for shell quartets of the same AM class that is executed on each node. Each thread will thus compute the integrals for multiple shell quartets as one unit

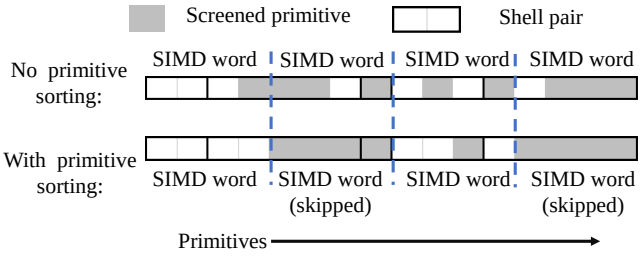

Fig. 2. Sorting primitive integrals for primitive screening. Heavy lines delineate primitive integrals corresponding to a ket-side shell pair. An example is shown for a sequence of 5 ket-side shell pairs with 2, 5, 1, 3, and 5 primitive integrals, respectively. Without sorting, no SIMD words can be skipped. With sorting, two of the four SIMD words can be skipped. All four primitives in a SIMD word are screened (shown in gray) when a SIMD word can be skipped.

of work and thus improve SIMD utilization, particularly for lightly contracted basis sets.

The batching procedure has several requirements:

1) Only unique shell quartets are computed, i.e., $(MN|PQ)$ is symmetric with 7 other shell quartets, for example $(NM|QP)$, and only one of these should be computed.
2) Shell quartets containing integrals that all are small in magnitude are neglected and the shell quartet is not computed; this is called *shell quartet screening* and is different from and used together with primitive integral screening.
3) Batching cannot precompute a list of all shell quartets with the same AM class, as the large number of shell quartets makes this infeasible.
4) Batching for a given AM class should be handled by a single thread; this maximizes the number of shell quartets that can be batched together.

A. ERI Batching Scheme

Our solution is to use a dynamic procedure where queues for each AM class are maintained. Shell quartets are added to the appropriate queue depending on their AM class, and a full queue constitutes a batch of shell quartets with integrals to be computed.

Algorithm 1 shows the procedure. For simplicity, we assume an appropriate partitioning of the shell quartets among the nodes and that the indices M, N, P, Q are only over the indices for quartets that are assigned to a node. The M,N loop (line 2) is parallelized with OpenMP multithreading. Each thread maintains its private shell quartet queues, pushing all valid shell quartets it encounters to a queue according to the AM of the P and Q shells (lines 4 - 6). A batch of shell quartets will be submitted to Simint when a queue is full, then the ERI results are consumed, and this queue is reset (lines 7 - 10). When a thread has looped over all P,Q pairs for a given M,N pair, it submits all its non-empty queues to Simint and resets these queues (lines 13 - 16). We note that the procedure is thread-aware but lock-free.

The shell quartet queues in Algorithm 1 should be long enough to provide good SIMD efficiency, but very long queues are not necessary. In the experimental tests (see Table V later in this paper), we found a queue length of 16 to give good performance.

Algorithm 1 Batched ERI computation

```
1:  Each thread initializes its private queues
2:  for shell pairs M,N in parallel do
3:    for shell pairs P, Q do
4:      if (MN|PQ) is unique and not screened then
5:        Compute bra-side shell pair id: id = (P,Q)
6:        Push (MN|PQ) to queue q[id]
7:        if queue q[id] is full then
8:          Compute shell quartets in queue q[id]
9:          Reset queue q[id]
10:       end if
11:     end if
12:   end for
13:   for a non-empty queue q[i] do
14:     Compute shell quartets in queue q[i]
15:     Reset queue q[i]
16:   end for
17: end for
```

B. ERI Library Issues for Batched Computation

To compute the integrals for a batch of shell quartets, Algorithm 1 calls Simint with the handle of shell pair (M,N), and a set of handles to shell pairs $\{(P_i,Q_i)\}$ such that all shells P_i have the same AM and all shells Q_i have the same AM. This should be considered the basic unit of work for a vectorized integral library.

The computation of the ERIs in a shell quartet $(MN|PQ)$ uses some quantities that only depend on *shell pairs* (M,N) and (P,Q). These quantities, called *shell pair data*, can be precomputed and stored because they are used for multiple shell quartets. If we store shell pair data, we must gather the appropriate data into a continuous buffer for batched ERI computation, since the vectorized computation needs the data in this format. Alternatively, shell pair data can be regenerated every time they are needed.

Let p_1 and p_2 be the number of primitive functions for two shells in a shell pair. The store-reuse approach needs to copy $12 \times p_1 \times p_2$ double-precision words for this shell pair, where 12 is the number of arrays in its shell pair data. Regeneration of the shell pair data only needs to read the coordinates and coefficients data for the two shells, which is $6+2\times(p_1+p_2)$ double-precision words. With primitive screening without primitive sorting, the regeneration approach could be a little faster than the store-reuse approach in some cases.

To see which approach is more efficient, we tested both these approaches with different workloads. Experiments showed that storing and reusing all shell pair data is much faster in most cases. If we disable sorting for primitive screening, recomputing shell pair data is a little bit faster in a few cases. Therefore, we use the store-reuse approach.

Finally, we note a caveat when using batching with a dynamic distribution of shell quartets to the compute nodes. When dynamic distribution is used, a task containing some number of shell quartets is assigned to a node that is free. When ERI computations are batched, one must make sure that the tasks contain enough shell quartets such that large enough batches of the same AM class can be formed.

VI. Optimizing Multithreaded Fock Matrix Accumulation

Once ERIs are computed, they are used to form the Fock matrix in the Hartree-Fock (HF) method [30], density functional theory (DFT) [31], and other methods. In the case of HF, blocks of the Fock matrix F have the form

$$F_{MN} = H_{MN}^{core} + \sum_{PQ} D_{PQ} \{2(MN|PQ) - (MP|NQ)\},$$

where $(MN|PQ)$ is a shell quartet of ERIs, D is the density matrix, and H^{core} is a known, fixed matrix. The Fock matrix, however, is not constructed one block at a time as the above equation suggests. Instead, because of the high cost of computing the ERIs, a unique shell quartet $(MN|PQ)$ is computed once (which is equivalent to computing $(PQ|MN)$ and 6 other symmetric shell quartets), and then contributions to F_{MN}, F_{MP}, F_{NP}, F_{PQ}, F_{MQ}, F_{NQ} are accumulated into F. These six blocks are all the ways of choosing pairs of the 4 shell indices, M, N, P, Q. Note also that F is symmetric so F_{NM}, etc., do not need to be computed.

Since multiple threads are computing shell quartets and accumulating them into a shared Fock matrix F, a thread-safe Fock matrix accumulation algorithm is needed. Previously, codes have used a combination of atomic operations and private copies of F that are summed at the end [2], [8], [32]. In the past, the performance of Fock matrix accumulation has not been critical since the ERI calculations are dominant. However, with more efficient ERI calculations and large numbers of threads, we find that the execution time for Fock matrix accumulation can be significant, especially when using basis sets with small numbers of basis functions per atom.

After a thread computes a shell quartet $(MN|PQ)$, the basic Fock matrix accumulation procedure executed by the thread is shown in Algorithm 2. In the Algorithm, *ERI* denotes the 4-D array storing the ERIs in the shell quartet, with dimensions $dimM \times dimN \times dimP \times dimQ$. A four-fold loop iterates over each of the elements of the shell quartet and computes contributions to F.

Assuming one copy of F for some set of threads, for thread safety, atomic operations are used to update F_{PQ}, F_{MQ}, and F_{NQ} (lines 9-11), resulting in $3 \times dimM \times dimN \times dimP \times dimQ$ atomic operations. Updates to F_{MN}, F_{MP} and F_{NP} can be accumulated in registers, with atomic operations used to accumulate these register values outside the iQ loop to update F. The total number of atomic operations needed here is

$$AO_1 = 3 \times dimM \times dimN \times dimP \times dimQ + 2 \times dimM \times dimN \times dimP + dimM \times dimN.$$

Algorithm 2 Fock matrix accumulation, given shell quartet $(MN|PQ)$ with dimensions $dimM \times dimN \times dimP \times dimQ$.

```
1:  for iM = 0 to dimM-1 do
2:    for iN = 0 to dimN-1 do
3:      for iP = 0 to dimP-1 do
4:        for iQ = 0 to dimQ-1 do
5:          I = ERI(iM,iN,iP,iQ)
6:          Update F_MN(iM,iN) with D_PQ(iP,iQ),   I
7:          Update F_MP(iM,iP)  with D_NQ(iN,iQ),  I
8:          Update F_NP(iN,iP)  with D_MQ(iM,iQ), I
9:          Update F_PQ(iP,iQ)  with D_MN(iM,iN), I
10:         Update F_MQ(iM,iQ)  with D_NP(iN,iP),  I
11:         Update F_NQ(iN,iQ)  with D_MP(iM,iP), I
12:       end for
13:     end for
14:   end for
15: end for
```

Our paramount consideration is reducing the usage of atomic operations, which we find to limit the performance of this approach. Thus, an obvious alternative is to split Algorithm 2 into six four-fold loops such that each four-fold loop only updates one block of the Fock matrix. The order of the loops can be exchanged so that atomic operations can be placed in the second nested loop. Algorithm 3 is an example for F_{MQ}. The calculation and update of other Fock matrix blocks are similar. After splitting, the total number of atomic operations becomes

$$AO_2 = dimM \times (dimN + dimP + dimQ) + dimN \times (dimP + dimQ) + dimP \times dimQ.$$

AO_2 equals the sum of the sizes of blocks F_{MN}, F_{PQ}, F_{MP}, F_{MQ}, F_{NP} and F_{NQ}, so it is the lower bound for the number of atomic operations needed.

Algorithm 3 Updating F_{MQ} with a separate four-fold loop

```
1:  for iM = 0 to dimM-1 do
2:    for iQ = 0 to dimQ-1 do
3:      register f_MQ = 0
4:      for iN = 0 to dimN-1 do
5:        for iP = 0 to dimP-1 do
6:          I = ERI(iM,iN,iP,iQ)
7:          Update f_MQ with D_NP(iN,iP) and I
8:        end for
9:      end for
10:     atomic_add(F_MQ(iM,iQ), f_MQ)
11:   end for
12: end for
```

As the price of reducing the number of atomic operations, the performance of this new approach is bounded by memory access: it needs to read the entire *ERI* array six times and has discontinuous memory access to *ERI* when updating F_{MQ}, F_{NQ} and F_{PQ}. We want to find a way that uses only AO_2 atomic operations while reading *ERI* continuously only once.

We observe that AO_2 is relatively small in most cases. If the maximum AM of the shells in a given problem is 4, which is not uncommon, then the maximum size of any dimension of the *ERI* array is 15, and thus $AO_2 \leq 15^2 \times 6 = 1350$ doubles. Each thread can use a thread-private buffer to accumulate the updates of the six Fock submatrices and add them to the shared F matrix at the end. Therefore, we do not need to split the four-fold loop in Algorithm 2 and we can avoid reading *ERI* multiple times. The buffer for each thread is small enough ($<$ 11 KB for maximum AM = 4) to fit in the L1 data cache of most processors.

When the ERIs are computed in batched fashion, another optimization allows us to eliminate about half of the atomic operations. In the batched ERI algorithm, line 3 of Algorithm 1 actually contains two loops: the outer loop iterates over shell indices P and the inner loop iterates over shell indices Q. As a result, all shell quartets in a batch have the same M and N shells and are likely to have only a small number of different P shells.

Suppose that all shell quartets in a batch have the same M, N and P shells. In this case, all shell quartets in the batch update the same F_{MN}, F_{MP}, F_{NP}. Therefore, the accumulation of F_{MN}, F_{MP}, F_{NP} can be performed in the thread-local buffers without atomics until the end of the batch. Atomics are only used at the end of the batch to accumulate these thread-local buffers into the shared F. As before, the accumulation of F_{PQ}, F_{MQ}, F_{NQ} needs atomics for each shell quartet processed. Thus the number of atomic operations can be reduced to

$$AO_3 = dimQ \times (dimM + dimN + dimP)$$

per shell quartet other than the last quartet in a batch. We present this optimized Fock matrix accumulation procedure in Algorithm 4. We should note that after optimization, the performance of this procedure is still bounded by two factors: (1) the flop-per-byte ratio of Algorithm 4 is still low, and (2) $dimQ$ is usually very small (basis sets usually have more low AM shells than high AM shells), which leads to a significant vectorization overhead in Algorithm 4.

Algorithm 4 Optimized Fock matrix accumulation, for shell quartet $(MN|PQ)$

```
1: if MN has changed since last call then
2:    Initialize thread-local buffer f_MN to 0
3: end if
4: if MP and NP have changed since last call then
5:    Initialize thread-local buffers f_MP, f_NP to 0
6: end if
7: Initialize thread-local buffers f_MQ, f_NQ, f_PQ to 0
8: for iM = 0 to dimM-1 do
9:    for iN = 0 to dimN-1 do
10:      for iP = 0 to dimP-1 do
11:         for iQ = 0 to dimQ-1 do
12:            I = ERI(iM,iN,iP,iQ)
13:            Update f_MN(iM,iN) with D_PQ(iP,iQ),   I
14:            Update f_MP(iM,iP) with D_NQ(iN,iQ),   I
15:            Update f_NP(iN,iP) with D_MQ(iM,iQ),   I
16:            Update f_PQ(iP,iQ) with D_MN(iM,iN),   I
17:            Update f_MQ(iM,iQ) with D_NP(iN,iP),   I
18:            Update f_NQ(iN,iQ) with D_MP(iM,iP),   I
19:         end for
20:      end for
21:   end for
22: end for
23: Update F_MQ, F_NQ, F_PQ with f_MQ, f_NQ, f_PQ
24: if last MP and NP then
25:    Update F_MP, F_NP with f_MP, f_NP
26: end if
27: if last MN then
28:    Update F_MN with f_MN
29: end if
```

VII. Performance Results

To test the performance of vectorized and batched ERI computation in a quantum chemistry application, we use the GTFock code [2], [33], [34] which implements the Hartree-Fock method for distributed memory computers. GTFock originally used an optimized version of the ERD integral library, called OptERD [2], which has a $2\times$ performance advantage over the original ERD library [35] developed for the ACES III quantum chemistry package [36]. ERD and OptERD use the Rys quadrature [37] method for computing ERI. We updated GTFock to use Simint to compute ERIs in vectorized fashion. Our implementation calls Simint one shell quartet at a time, or with batches of shell quartets, using our batching procedure described in Section V.

We have also modified the multithreaded Fock matrix accumulation routine in GTFock using the optimized procedure described in Section VI. Further, our tests use our optimizations for Simint described in Section IV.

The Hartree-Fock method is based on self-consistent field (SCF) iterations. The vast majority of the execution time of each iteration is spent building the Fock matrix ("Fock build") which itself consists of computing the ERIs and forming and accumulating the blocks of the Fock matrix ("Fock accum"). The eigenvalue calculations in each SCF iteration require very little time, although they do require relatively more time in DFT calculations where large molecular or materials systems are simulated with different approximations than used in Hartree-Fock. All reported timings for a Fock build or Fock accum are averaged over the SCF iterations needed for a Hartree-Fock calculation.

The tolerance for screening of shell quartets implemented in GTFock is 10^{-11}. The tolerance for primitive integral screening is 10^{-14} used for Simint and OptERD. These are commonly used values for these parameters.

Tests were performed using the Intel Xeon Phi 7250 (Knights Landing, or KNL) nodes on the Cori supercomputer at NERSC. Each of these nodes has 68 cores running at 1.4 GHz with 4 hyperthreads per core and 16 GB MCDRAM. Since January 2018, the Cori KNL nodes are fixed to use

TABLE I
TEST SYSTEMS FOR PERFORMANCE EVALUATION

Test System	*Basis Set*	*Atoms*	*Shells*	*Basis Functions*
protein_28	aug-cc-pVTZ	30	350	1230
protein_28	cc-pVDZ	30	138	310
protein_28	ANO-DZ	30	214	526
1hsg_28	cc-pVDZ	122	549	1220
1hsg_45	cc-pVDZ	554	2430	5330

quadrant clustering mode and MCDRAM is fixed to use cache mode. Most tests below were performed on a single KNL node but some tests were performed with 9 or 64 nodes. Tests on a single node used 9 MPI processes and 28 threads per process (total of 252 threads). Tests on 64 nodes used 4 MPI processes per node and 68 threads per process (total of 272 threads per node). Tests with 9 nodes used varying configurations described below. Codes were compiled with Intel C/C++/Fortran compiler version 17.0.4 and Cray MPI version 7.6.2.

Tests were performed using basis sets with different amounts of contraction:

- aug-cc-pVTZ: A lightly contracted basis set,
- cc-pVDZ: A moderately contracted basis set that has few high AM shells,
- ANO-DZ: A heavily contracted basis set.

The test molecular systems are derived from a protein-ligand complex consisting of a human immunodeficiency virus (HIV) drug molecule bound to HIV II protease. The atomic configuration comes from the protein data bank (code 1HSG). Small test systems, called protein_28, consist of just the binding pocket portion of the protein. Larger test systems, called 1hsg_28 and 1hsg_45 consist of the drug molecule and a portion of its protein environment. See Table I for additional details of these test systems. Tests in Sections VII-A to VII-E using a single KNL node use protein_28. The larger molecular systems are used for other tests.

A. Overall Results

Four cases are of primary interest for comparison. These are the implementations using (1) OptERD, (2) scalar Simint without batching, (3) vectorized Simint without batching, (4) vectorized Simint with batching. Scalar Simint is a version of Simint compiled without vector instructions and is useful to quantify the effectiveness of Simint's vectorization. Table II shows the timings for one Fock build when different basis sets are used. Table III shows the component of these timings that is due to ERI calculation. In all cases, low-level optimizations and primitive sorting are enabled in Simint. Our Fock matrix accumulation optimization is also enabled, including in the OptERD case.

Overall, we observe that vectorized Simint with batching gives a large speedup compared to using OptERD in all cases.

A surprising result is that, without batching, vectorized Simint is only slightly better than scalar Simint for the moderately contracted basis set cc-pVDZ and is actually worse

TABLE II
FOCK BUILD TIMINGS (IN SECONDS)

Basis Set	*OptERD*	*Scalar Simint w/o Batching*	*Vectorized Simint w/o Batching*	*Vectorized Simint w/ Batching*
aug-cc-pVTZ	345	221	250	92.5
cc-pVDZ	2.98	1.66	1.56	0.68
ANO-DZ	3578	1722	427	436

TABLE III
ERI CALCULATION TIMINGS (IN SECONDS)

Basis Set	*OptERD*	*Scalar Simint w/o Batching*	*Vectorized Simint w/o Batching*	*Vectorized Simint w/ Batching*
aug-cc-pVTZ	251	128	143	39.4
cc-pVDZ	2.03	0.75	0.67	0.33
ANO-DZ	3511	1677	393	406

than scalar Simint for the lightly contracted basis set aug-cc-pVTZ. This is due to extremely short SIMD loop lengths in these two cases when batching is not used, as will be shown below. This poor performance was not noticed in the original Simint paper [3] which only used a microbenchmark to measure ERI calculation time in a way that is divorced from how Simint would actually be called from a quantum chemistry code.

For highly contracted basis sets like ANO-DZ, Simint has a large speedup of approximately $4\times$ due to vectorization even without batching. For highly contracted basis sets, the vast majority of the computation is spent on well-vectorized VRRs rather than the poorly-vectorized HRRs that take place after the primitive integrals are contracted. The overhead of gathering shell pair data for batching actually has a small negative impact on the Fock build performance in this case.

For the lightly and moderately contracted basis sets, aug-cc-pVTZ and cc-pVDZ, the speedup of using vectorized Simint with batching vs. without batching is significant: 2.70 and 2.29, respectively. Therefore, batching for Simint is essential when using lightly and moderately contracted basis sets.

To support these observed results of batching on vectorization efficiency, we compare the average length of the SIMD loop in Simint with and without ERI batching. Table IV shows that without batching, the average SIMD loop length when using aug-cc-pVTZ and cc-pVDZ basis sets is very small, which means that the SIMD utilization is low. Very low SIMD utilization can make vectorized Simint slower than scalar Simint, as we saw above. When using the ANO-DZ basis set, the average SIMD loop length before batching is already large enough to obtain high vectorization efficiency.

Table V shows timings for different queue lengths used in Algorithm 1. The results justify our choice of 16 for the queue length, with longer lengths not giving significant performance improvement at the cost of additional storage.

B. Effect of Sorting for Primitive Screening

The goal of sorting for primitive screening is to increase the fraction of SIMD words that can be neglected by trying

TABLE IV
AVERAGE SIMD LOOP LENGTH FOR EACH CALL TO SIMINT, WITH AND WITHOUT BATCHING

Basis Set	*w/o Batching*	*w/ Batching*	*Ratio*
aug-cc-pVTZ	2.7	40.0	14.81
cc-pVDZ	7.4	71.5	9.66
ANO-DZ	79.3	1184.8	14.94

TABLE V
FOCK BUILD TIMINGS (IN SECONDS) WITH DIFFERENT QUEUE LENGTHS

Basis Set	*32*	*24*	*16*	*12*	*8*	*4*
aug-cc-pVTZ	91.2	91.8	92.5	97.7	99.7	122.2
cc-pVDZ	0.66	0.67	0.68	0.71	0.74	0.84
ANO-DZ	436	440	436	431	430	432

to group together neglected primitive integrals in a SIMD word. Table VI shows the percentage of primitive integrals that can be neglected, and the percentage of SIMD words that can be neglected, with and without sorting. In these tests, the ERI calculation is batched, and our Simint optimizations are enabled.

Compared to no sorting, primitive sorting increases the fraction of neglected SIMD primitives for ANO-DZ but is not as effective for aug-cc-pVTZ. This is reflected in the speedup results also shown in the table.

C. Effect of Low-level Simint Optimizations

Table VII shows the Fock build runtime with and without Simint low-level optimizations discussed in Sections IV-A and IV-B. In these tests, the ERI calculation is batched, and primitive sorting and Fock accumulation optimizations are enabled. We observe some speedup in each case. The speedup for the aug-cc-pVTZ basis set mainly reflects the effort of optimizing for high AM functions, since aug-cc-pVTZ is lightly contracted but has many such high AM functions. The speedup for the cc-pVDZ and ANO-DZ basis sets mainly reflects the effort of optimizing the contraction operation for AVX-512, given that these two basis sets have few high AM integrals, ANO-DZ is highly contracted, and the test molecule with cc-pVDZ has many integrals that belong to the $(ss|ss)$ AM class.

D. Effect of Fock Accumulation Optimization

Table VIII shows the average runtime of unoptimized and optimized Fock accumulation in one SCF iteration. The runtime for batched ERI calculation with our Simint optimizations is also shown for comparison. We observe that the runtime of unoptimized Fock accumulation when using aug-cc-pVTZ and cc-pVDZ basis sets is larger than the runtime of batched ERI calculation. Therefore, reducing the runtime of Fock accumulation is essential.

The optimized Fock accumulation algorithm greatly reduces the runtime of the Fock accumulation procedure for all tested basis sets and helps reduce the Fock build runtime. For the ANO-DZ basis set, ERI calculation dominates the runtime of Fock build, and optimizing the Fock accumulation only has a small impact on the Fock build runtime. Table IX justifies these statements, showing the Fock build time with and without Fock accumulation optimizations.

TABLE VI
REDUCTION OF SIMD WORDS WITH PRIMITIVES AND FOCK BUILD SPEEDUP VIA PRIMITIVE SORTING

Basis Set	*Percent Neglected Primitives*	*Percent Neglected SIMD Words*		*Fock Build Speedup via Sorting*
		w/o Sorting	*w/ Sorting*	
aug-cc-pVTZ	36.2	19.6	25.0	1.00
cc-pVDZ	62.0	41.6	51.3	1.05
ANO-DZ	91.3	64.6	78.4	1.33

TABLE VII
FOCK BUILD TIMINGS (IN SECONDS) WITH AND WITHOUT SIMINT LOW-LEVEL OPTIMIZATIONS. FOCK BUILD SPEEDUP AND ERI CALCULATION SPEEDUP DUE TO OPTIMIZATIONS ARE ALSO SHOWN.

Basis Set	*Fock Build w/o Opt.*	*Fock Build w/ Opt.*	*Fock Build Speedup*	*ERI Calc. Speedup*
aug-cc-pVTZ	97.1	92.5	1.05	1.11
cc-pVDZ	0.74	0.68	1.09	1.12
ANO-DZ	485	436	1.11	1.11

TABLE VIII
FOCK ACCUMULATION TIMINGS (IN SECONDS) AND SPEEDUP AFTER OPTIMIZATION

Basis Set	*Batched ERIs*	*Fock Accum. w/o Opt.*	*Fock Accum. w/ Opt.*	*Fock Accum. Speedup*
aug-cc-pVTZ	39.4	136.3	36.5	3.73
cc-pVDZ	0.336	0.432	0.197	2.19
ANO-DZ	405.5	13.3	6.15	2.16

E. Multi-thread Efficiency

We also tested the multi-thread efficiency of the optimized Fock build procedure. Here, the tests are performed on 9 KNL nodes with 1 MPI process per node. For 1 to 68 threads per MPI process, each core is bound to a physical core. For 136 / 272 threads per MPI processes, we use 2 / 4 hyperthreads on each physical core.

For the aug-cc-pVTZ and ANO-DZ basis sets, the test molecule is protein_28, as used in the previous tests. However, for the cc-pVDZ basis set, which uses a relatively small number of basis functions, we use the larger test molecule 1hsg_28 (see Table I).

Figure 3 shows timings for the Fock build, the batched ERI calculation, and the Fock accumulation. We can see that these three parts have very good multithreading scalability

TABLE IX
FOCK BUILD TIMINGS (IN SECONDS) WITH AND WITHOUT FOCK ACCUMULATION OPTIMIZATION

Basis Set	*Unoptimized Fock Accum.*	*Optimized Fock Accum.*	*Speedup of Fock Build*
aug-cc-pVTZ	204	92.5	2.20
cc-pVDZ	0.97	0.68	1.42
ANO-DZ	444	436	1.02

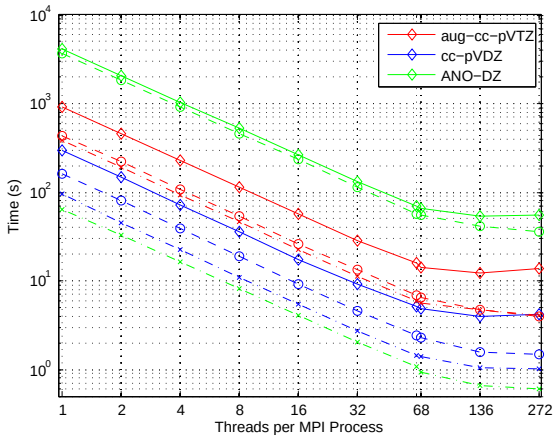

Fig. 3. Multithreading runtime of Fock build. Different colors identify different basis sets. Diamond marks are Fock build data, circle marks are ERI calculation data, and cross marks are Fock accumulation data. The speedup is almost perfect up to 68 threads.

TABLE X
FOCK BUILD TIMINGS (IN SECONDS) ON 9 KNL NODES FOR DIFFERENT NUMBERS OF MPI PROCESSES (P) AND THREADS PER PROCESS (T).

Basis Set	*9P × 272T*	*36P × 68T*	*81P × 30T*	*144P × 17T*
aug-cc-pVTZ	13.6	12.1	15.2	18.1
cc-pVDZ	4.21	3.81	6.73	7.55
ANO-DZ	55.3	51.7	57.0	80.5

for all basis sets. In these test cases, the maximum AM is 3, which means that the size of the thread-local buffer for Fock accumulation is $10^2 \times 6 \times 8$ Bytes $= 4800$ Bytes. Using 4 hyperthreads on a physical core only requires 18.75 KBytes. Therefore, we can see that using 4 hyperthreads per core can also give speedup to Fock accumulation in these cases.

Table X shows timing results for different combinations of number of MPI processes and OpenMP threads for 9 KNL nodes. The results show that 4 MPI threads per node gives the best result among the configurations tested. This may be due to being able to more fully exploit the inter-node memory bandwidth available when multiple MPI processes per node are used.

F. Multi-node Test with and without Batching

Finally, we demonstrate our performance optimizations on a large Hartree-Fock calculation, running GTFock with Simint on 64 KNL nodes on the Cori supercomputer. The test system is 1hsg_45 with the cc-pVDZ basis set (see Table I). Based on the results in the previous subsection, we use 4 MPI processes per node and 68 OpenMP threads per process, thus fully utilizing all hyperthreads on the hardware.

For calculations with multiple nodes, GTFock uses Global Arrays [38] and MPI. To accelerate convergence, the SCF iterations use the method of commutator-based direct inversion of the iterative subspace [39]. The initial guess for the density matrix uses superposition of atomic densities [40].

We tested both the batched and non-batching version of our code, with all our optimizations enabled. The Hartree-Fock calculation for the system converged in 16 SCF iterations to an energy of -12286.9243511005 Hartrees with the energy changing less than 10^{-10} Hartrees at convergence. The average Fock build time for one SCF iteration of the non-batching and batched version is 59.1 and 33.3 seconds, respectively, giving an average speedup of 1.77 when batching is used.

VIII. CONCLUSION

In this paper, we have described and demonstrated several performance optimizations related to the computation and use of ERIs in a quantum chemistry code. Adapting existing codes to batch shell quartets with the same AM class can follow the procedure described in this paper.

Our performance evaluation shows that batching shell quartets is significant for improving vectorization efficiency of a vectorized ERI library and can greatly improve the performance of ERI calculation when using lightly and moderately contracted basis sets. After accelerating ERI calculation, accumulating the blocks of the Fock matrix became a new performance bottleneck. This was resolved by using thread-private accumulation buffers and reordering the accumulation procedure. Multi-thread tests show that our optimized code has very good multithreading scalability, suggesting that the code does not stress shared resources such as L1 D-cache.

The batching procedure and improvements to Simint and GTFock have been incorporated into their code bases and are released in open-source form at https://github.com/simint-chem and https://github.com/gtfock-chem, respectively.

Although our work has focused on quantum chemistry, we found certain performance optimization principles to be particularly useful, and which are applicable to other domains. First, many of our optimizations arose from understanding the behavior of the code for expected and common input data. In other words, instead of optimizing code based on control flow alone, we also considered patterns or properties of the data. Second, compiler optimization reports were very useful for identifying memory alignment and vectorization issues as well as telling whether the compiler is optimizing the code as we expected. Third, we focused our primary attention on memory access patterns and the memory hierarchy, which is well-accepted but performance issues here can be complex and thus not always adequately addressed.

In future work, we plan to benchmark batched and vectorized ERI computation in the NWChem [9] and PSI4 [41] packages, including with density fitting methods where 3-center integrals are needed. We also plan to benchmark and optimize our techniques for Intel Skylake processors.

ACKNOWLEDGMENT

The authors wish to thank Benjamin Pritchard and Jeff R. Hammond for assistance and helpful discussions. Funding from an Intel Parallel Computing Center grant and the National Science Foundation grant ACI-1147843 are gratefully acknowledged. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] H. Shan, B. Austin, W. De Jong, L. Oliker, N. Wright, and E. Aprà, "Performance tuning of Fock matrix and two-electron integral calculations for NWChem on leading HPC platforms," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13) held as part of SC13*, 2013.

[2] E. Chow, X. Liu, M. S., M. Dukhan, M. Smelyanskiy, J. R. Hammond, Y. Du, X.-K. Liao, and P. Dubey, "Scaling up Hartree–Fock calculations on Tianhe-2," *International Journal of High Performance Computing Applications*, vol. 30, pp. 85–102, 2016.

[3] B. P. Pritchard and E. Chow, "Horizontal vectorization of electron repulsion integrals," *Journal of Computational Chemistry*, vol. 37, no. 28, pp. 2537–2546, 2016.

[4] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M.-J. Yvonne Ou, J. Pei, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Álvaro Vázquez-Mayagoitia, N. Vence, T. Yanai, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S123–S142, 2016.

[5] V. R. Saunders and M. F. Guest, "Applications of the CRAY-1 for quantum chemistry calculations," *Computer Physics Communications*, vol. 26, no. 3, pp. 389 – 395, 1982.

[6] P. M. W. Gill, M. Head-Gordon, and J. A. Pople, "Efficient computation of two-electron - repulsion integrals and their nth-order derivatives using contracted Gaussian basis sets," *The Journal of Physical Chemistry*, vol. 94, no. 14, pp. 5564–5572, 1990.

[7] K. Wolinski, R. Haacke, J. F. Hinton, and P. Pulay, "Methods for parallel computation of SCF NMR chemical shifts by GIAO method: Efficient integral calculation, multiFock algorithm, and pseudodiagonalization," *Journal of Computational Chemistry*, vol. 18, no. 6, pp. 816–825, 1997.

[8] H. Shan, S. Williams, W. de Jong, and L. Oliker, "Thread-level parallelization and optimization of NWChem for the Intel MIC architecture," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '15. New York, NY, USA: ACM, 2015, pp. 58–67.

[9] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. Van Dam, D. Wang, J. Nieplocha, E. Aprà, T. Windus, and et al., "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[10] E. F. Valeev, "A library for the evaluation of molecular integrals of many-body operators over Gaussian functions," http://libint.valeyev.net/, 2014.

[11] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, and et al., "General atomic and molecular electronic structure system," *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[12] Q. Sun, "Libcint: An efficient general integral library for Gaussian basis functions," *Journal of Computational Chemistry*, vol. 36, no. 22, pp. 1664–1671, August 2015.

[13] J. Zhang, "libreta: Computerized optimization and code synthesis for electron repulsion integral evaluation," *Journal of Chemical Theory and Computation*, vol. 14, no. 2, pp. 572–587, 2018.

[14] K. Yasuda, "Two-electron integral evaluation on the graphics processor unit," *Journal of Computational Chemistry*, vol. 29, no. 3, pp. 334–342, 2007.

[15] I. S. Ufimtsev and T. J. Martinez, "Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.

[16] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus, "Uncontracted Rys quadrature implementation of up to g functions on graphical processing units," *Journal of Chemical Theory and Computation*, vol. 6, no. 3, pp. 696–704, 2010.

[17] N. Luehr, I. S. Ufimtsev, and T. J. Martinez, "Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs)," *Journal of Chemical Theory and Computation*, vol. 7, no. 4, pp. 949–954, 2011.

[18] K. A. Wilkinson, P. Sherwood, M. F. Guest, and K. J. Naidoo, "Acceleration of the GAMESS-UK electronic structure package on graphical processing units," *Journal of Computational Chemistry*, vol. 32, no. 10, pp. 2313–2318, 2011.

[19] Y. Miao and K. M. Merz, "Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations," *Journal of Chemical Theory and Computation*, vol. 9, no. 2, pp. 965–976, 2013.

[20] T. Ramdas, G. K. Egan, D. Abramson, and K. K. Baldridge, "On ERI sorting for SIMD execution of large-scale Hartree Fock SCF," *Computer Physics Communications*, vol. 178, no. 11, pp. 817 – 834, 2008.

[21] ——, "Uniting extrinsic vectorization and shell structure for efficient simd evaluation of electron repulsion integrals," *Chemical Physics*, vol. 349, no. 13, pp. 147 – 157, 2008, electron Correlation and Molecular Dynamics for Excited States and Photochemistry.

[22] ——, "ERI sorting for emerging processor architectures," *Computer Physics Communications*, vol. 180, no. 8, pp. 1221 – 1229, 2009.

[23] T. Helgaker, P. Jørgensen, and J. Olsen, *Molecular Electronic-Structure Theory*. John Wiley & Sons, LTD, 2000.

[24] S. Obara and A. Saika, "General recurrence formulas for molecular integrals over Cartesian Gaussian functions," *The Journal of Chemical Physics*, vol. 89, no. 3, pp. 1540–1559, 1988.

[25] ——, "Efficient recursive computation of molecular integrals over Cartesian Gaussian functions," *The Journal of Chemical Physics*, vol. 84, no. 7, pp. 3963–3974, 1986.

[26] R. Lindh, U. Ryu, and B. Liu, "The reduced multiplication scheme of the Rys quadrature and new recurrence relations for auxiliary function based two-electron integral evaluation," *The Journal of Chemical Physics*, vol. 95, no. 8, pp. 5889–5897, 1991.

[27] T. P. Hamilton and H. F. Schaefer, "New variations in two-electron integral evaluation in the context of direct SCF procedures," *Chemical Physics*, vol. 150, no. 2, pp. 163 – 171, 1991.

[28] Colfax Research, "Capabilities of Intel AVX-512 in Intel Xeon Scalable processors (Skylake)," https://colfaxresearch.com/skl-avx512/.

[29] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier, 2016.

[30] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover Publications, 2006.

[31] J. A. Pople, P. M. Gill, and B. G. Johnson, "Kohn-Sham density-functional theory within a finite basis set," *Chemical Physics Letters*, vol. 199, no. 6, pp. 557–560, 1992.

[32] V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky, and M. S. Gordon, "An efficient MPI/OpenMP parallelization of the Hartree-Fock method for the second generation of Intel Xeon Phi processor," *SC 17': Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[33] X. Liu, A. Patel, and E. Chow, "A new scalable parallel algorithm for Fock matrix construction," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[34] E. Chow, X. Liu, M. Smelyanskiy, and J. R. Hammond, "Parallel scalability of Hartree-Fock calculations," *The Journal of Chemical Physics*, vol. 142, no. 10, p. 104103, 2015.

[35] N. Flocke and V. Lotrich, "Efficient electronic integrals and their generalized derivatives for object oriented implementations of electronic structure calculations," *Journal of Computational Chemistry*, vol. 29, no. 16, pp. 2722–2736, 2008.

[36] V. Lotrich, N. Flocke, M. Ponton, A. D. Yau, A. Perera, E. Deumens, and R. J. Bartlett, "Parallel implementation of electronic structure energy, gradient, and Hessian calculations," *The Journal of Chemical Physics*, vol. 128, no. 19, p. 194104, 2008.

[37] J. Rys, M. Dupuis, and H. F. King, "Computation of electron repulsion integrals using the Rys quadrature method," *Journal of Computational Chemistry*, vol. 4, no. 2, pp. 154–157, 1983.

[38] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the Global Arrays shared memory programming toolkit," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[39] P. Pulay, "Improved SCF convergence acceleration," *Journal of Computational Chemistry*, vol. 3, no. 4, pp. 556–560, 1982.

[40] J. H. Van Lenthe, R. Zwaans, H. J. J. Van Dam, and M. F. Guest, "Starting SCF calculations by superposition of atomic densities," *Journal of Computational Chemistry*, vol. 27, no. 8, pp. 926–932, 2006.

[41] J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, N. J. Russ, M. L. Leininger, C. L. Janssen, E. T. Seidl, W. D. Allen, H. F. Schaefer, R. A. King, E. F. Valeev, C. D. Sherrill, and

T. D. Crawford, "PSI4: an open-source *ab initio* electronic structure program," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 2, pp. 556–565, 2012.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: ACCELERATING QUANTUM CHEMISTRY WITH VECTORIZED AND BATCHED INTEGRALS

A. Abstract

This description contains the information needed to compile and launch the computational experiments in the SC18 paper "Accelerating Quantum Chemistry with Vectorized and Batched Integrals". More precisely, we describe how to compile the optimized Simint and GTFock programs and run the examples in Section VII. The results in Section VII can be reproduced and validated.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Hartree-Fock, Obara-Saika
- **Program:** GTFock, Simint
- **Compilation:** Intel compilers version 16 or 17 and MPI-3 supported MPI library
- **Binary:** MPI executable
- **Data set:** Included in GTFock
- **Run-time environment:** Linux environment with MPI
- **Hardware:** Intel Xeon CPU or Intel Xeon Phi 7200 series CPU
- **Output:** On-screen output: runtime and system energy of each iteration; optional file output: wavefunction matrix
- **Experiment workflow:** Clone and compile Simint code generator, generate and compile the source code of Simint, clone and compile libcint and GTFock, run the binaries, observe the results
- **Publicly available?:** Yes.

2) How delivered: Intel Parallel Studio XE including Intel C/C++/Fortran compiler, Intel MPI and Intel MKL. Licenses can be applied with education email account or for trial. ARMCI-MPI and Global Arrays can be downloaded from their websites. OptERD, libcint, Simint code generator and GTFock can be cloned from https://github.com/gtfock-chem.

3) Hardware dependencies: None. For reproducibility, we used Intel Xeon Phi 7200 series CPU.

4) Software dependencies: Simint requires CMake 3.0 or higher version and a C++11 compliant compiler which can also use intrinsic functions for AVX, AVX2 and AVX-512 instruction sets. We tested the Intel compilers in Intel Parallel Studio XE 2017 update 4, which is the version we suggest to use. ARMCI-MPI requires an MPI-3 compliant MPI library. We tested Intel MPI 2017 on a small cluster with Intel Xeon CPU and Cray MPI 7.6.2 on Cori supercomputer. We suggest using Intel MPI 2017 for small clusters and using vendor-customized MPI implementations on supercomputers.

5) Datasets: The datasets in the paper are contained in the GTFock repository on GitHub. The corresponding filenames for testing are given in Table XI.

C. Installation

Set *WORK_TOP* environment variable to the directory you will install the libraries and GTFock.

TABLE XI
FILENAMES AND RELATIVE PATH OF DATASETS

Type	Name	Relative Path
basis set	aug-cc-pVTZ	data/aug-cc-pvtz/aug-cc-pvtz.gbs
basis set	cc-pVDZ	data/cc-pvdz/cc-pvdz.gbs
basis set	ANO-DZ	data/ano-dz.gbs
molecule	protein_28	data/1hsg/protein_28.xyz
molecule	1hsg_28	data/1hsg/1hsg_28.xyz
molecule	1hsg_45	data/gb/1hsg_45.xyz

1) Compiling OptERD: OptERD is no longer supported in GTFock. It should be used for performance comparison only. Skip this part if you do not need the performance data of GTFock with OptERD.

```
cd $WORK_TOP
git clone https://github.com/gtfock-chem\
/OptErd_Makefile.git
cd OptErd_Makefile
# Adjust the make.in according to your system
make -j4
```

2) Generating Simint Source Code and Compiling Simint: To compile Simint, you need to compile the Simint code generator and use it to generate the source code of Simint. To disable the low-level optimizations for Simint in this paper, execute `git checkout 3257a93` at the top directory of simint-generator to switch to the old version.

```
cd $WORK_TOP
git clone https://github.com/gtfock-chem\
/simint-generator.git
cd simint-generator
mkdir build
cd build
CC=icc CXX=icpc cmake ../
make -j16
```

Then, use a python script to generate all Simint source code. The python script can be executed with python 2 or python 3:

```
cd $WORK_TOP/simint-generator
python create.py -g build/generator/ostei \
-l 4 -p 4 -d 0 -ve 4 -he 4 \
-vg 5 -hg 5 gtfock-simint
mv gtfock-simint ../
```

Finally, compile Simint for a target CPU architecture. Please read the README file in the top directory of Simint to see how to specify the target CPU architecture. The following commands will compile Simint for KNL platforms.

```
cd $WORK_TOP/gtfock-simint
mkdir build-avx512
cd build-avx512
CC=icc CXX=icpc cmake ../ \
-DSIMINT_VECTOR=micavx512 \
-DCMAKE_INSTALL_PREFIX=./install
make -j32 install
```

To disable primitive sorting, comment line 575 and line 576 in the file *$WORK_TOP/gtfock-simint/simint/shell/shell.c*.

3) Compiling libcint: libcint is a small library that serves as an interface for calling either OptERD or Simint from GTFock.

```
cd $WORK_TOP
git clone https://github.com/gtfock-chem\
/libcint.git
cd libcint
# Adjust the Makefile according to the
# directory you compiled Simint
make libcint.a
```

4) Compiling ARMCI-MPI: ARMCI-MPI needs to be compiled with an MPI library. The following commands use Intel compilers and Intel MPI. You may need to replace the MPI compiler wrapper if you use other compilers and/or MPI libraries. For the Cori supercomputer, you should always use Cray MPI instead of Intel MPI.

```
cd $WORK_TOP
git clone git://git.mpich.org/armci-mpi.git
cd armci-mpi
git checkout mpi3rma
./autogen.sh
# Recommend using a build directory
mkdir build
cd build
../configure CC=mpiicc \
--prefix=$WORK_TOP/ARMCI-MPI
make -j16 install
```

5) Compiling Global Arrays: We suggest that you compile Global Arrays with an MPI library and ARMCI-MPI. The following commands use Intel compilers and Intel MPI. You may need to replace the MPI compiler wrapper if you use other compilers and/or MPI libraries. For the Cori supercomputer, you should always use Cray MPI instead of Intel MPI. Please make sure that the compiler and MPI environment are the same as that of compiling ARMCI-MPI.

```
cd $WORK_TOP
# GA v5.3 is the only version we tested
wget http://hpc.pnl.gov/globalarrays\
/download/ga-5-3.tgz
tar xzf ga-5-3.tgz
cd ga-5-3
# Carefully set the mpi executables
# Recommend using a build directory
mkdir build
cd build
../configure CC=mpiicc MPICC=mpiicc \
CXX=mpiicpc MPICXX=mpiicpc \
F77=mpiifort MPIF77=mpiifort \
--with-mpi --with-armci=$WORK_TOP/ARMCI-MPI \
--prefix=$WORK_TOP/GAlib
make -j16 install
```

6) Compiling GTFock: The source code of GTFock can be cloned from https://github.com/gtfock-chem/gtfock.git. To compile GTFock, you need to modify the *make.in* file in the top directory of GTFock. File *make.in.Cori* and *make.in.KNL5* are the templates for compiling GTFock on the Cori supercomputer and a KNL node respectively. If you are using Intel Parallel Studio XE, the math libraries (BLAS, BLACS, LAPACK and ScaLAPACK) are provided by Intel MKL and integrated with Intel MPI. If you are compiling on the Cori supercomputer, the math libraries are provided by Cray SciLib and combined with Cray MPI. The example SCF program is *$WORK_TOP/gtfock/pscf/scf*.

GTFock cannot work with Simint and OptERD at the same time. To test GTFock with OptERD, execute `git checkout 67382ed` at the top directory of GTFock to switch to the old version of GTFock.

To test the non-batching version of GTFock, uncomment line 136 and comment line 137 in file *pfock/fock_task.c*.

D. Experiment workflow

For single node execution or launching on clusters without a job scheduling system, the following command should run on most platforms:

```
mpirun -np <nprocs> $WORK_TOP/gtfock/pscf/scf \
<basis> <xyz> <nprow> <npcol> \
<np2> <ntasks> <niters>
```

Where:

- `nprocs`: Number of MPI process
- `basis`: Gaussian94 format basis set file
- `xyz`: Cartesian coordinate file of the chemistry system
- `nprow`: Number of MPI process per row
- `npcol`: Number of MPI process per column
- `np2`: Number of MPI process per dimension (of cube) for purification
- `ntasks`: Each MPI process has `ntasks` × `ntasks` tasks
- `niters`: Maximum number of SCF iterations

Note:

- `nprow` × `npcol` mush be equal to `nprocs`
- `np2` × `np2` × `np2` should be close to `nprocs`
- Suggested values for `ntasks`: 3, 4, 5

For clusters and supercomputers with job scheduling systems like slurm, you need to write job scripts for launching GTFock on multiple nodes. File *Cori-KNL-*N.pbs* are the example job scripts for using KNL nodes on the Cori supercomputer.

E. Evaluation and expected result

In the screen output, lines starting with "fock build takes" show the elapsed wall-clock time of the Fock matrix construction in each SCF iteration; lines starting with "energy" show the energy of the current solution, which is used for checking convergence and correctness.

For the test system used in Section VII-F, GTFock with batched ERI calculation and other optimizations should have an average Fock matrix construction time of 33.3 seconds when running on 64 KNL nodes with 68 threads per MPI process and 4 MPI processes per KNL node on the Cori supercomputer.