Horizontal Vectorization of Electron Repulsion Integrals

Benjamin P. Pritchard and Edmond Chow*

We present an efficient implementation of the Obara–Saika algorithm for the computation of electron repulsion integrals that utilizes vector intrinsics to calculate several primitive integrals concurrently in a SIMD vector. Initial benchmarks display a 2–4 times speedup with AVX instructions over comparable scalar code, depending on the basis set. Speedup over scalar code is found to be sensitive to the level of contraction of the basis set, and is best for $(l_A l_B|l_C l_D)$ quartets when $l_D = 0$ or $l_B = l_D = 0$, which makes such a vectorization scheme particularly suitable for density fitting. The basic Obara–Saika algorithm, how it is vectorized, and the performance bottlenecks are analyzed and discussed. © 2016 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.24483

Introduction

In *ab initio* computational chemistry, the computation of electron repulsion integrals (ERI) is often a bottleneck of calculations. Computation of these integrals scales (formally) as the fourth power of the number of basis functions, although this cost is ameliorated somewhat by screening and eightfold permutational symmetry. Because of the expense of calculating these integrals, there has been much interest in the development of fast and efficient methods to calculate ERI, and this interest has spanned several decades. Different algorithms have been developed, such as the McMurchie–Davidson scheme,[1] variations of the Obara–Saika (OS) scheme,[2–7] Rys quadrature,[8–10] and the Accompanying Coordinate Expansion method.[11] In addition, improvements in computer hardware have led to specialized implementations, for example on GPUs.[12–16]

Many modern processor architectures and microarchitectures have single-instruction, multiple-data (SIMD) instructions, which are capable of operating on several data words simultaneously. Utilization of these instructions can result in speedups of 2-4x (or higher), depending on the size of the SIMD vectors. Previous work on vectorizing the Rys quadrature method[8–10] has been carried out by Sun,[17] who vectorized an expensive inner product in Rys quadrature. The LIBINT library, which utilizes OS recurrences, also contains experimental support for vectorization.[18] Some work has been performed in vectorizing existing codebases;[19,20] however, this retrofitting usually results in speedups that are less than would be expected from a new library that is written from scratch with vectorization in mind.

One scheme of vectorization algorithms is to first sort ERI into classes based on a selection criteria; commonly, just the angular momentum of the four shells would be used; however. the degree of contraction may also be included.[21–23] For example, all $(pp|ps)$ integrals would be batched together.[21] This type of approach is sometimes called *horizontal vectorization*. As the code path taken for any particular class is identical, code for a class could be vectorized such that several integrals could be computed concurrently using SIMD. Investigations into issues relating to sorting ERI have taken place over many years,[21–26] although not much information can be found with respect to actual implementations of horizontally vectorized integral algorithms.

Here, we present a method by which efficient calculation of ERI may be achieved by utilizing horizontal SIMD vectorization of the OS algorithm at several points. Specifically, calculation of primitive integrals is vectorized manually via the use of compiler intrinsics, with other parts of the integral calculation vectorized automatically by the compiler. Code for a specific class is created for a given microarchitecture via a generator. The generated code also allows for additional efficiency by consecutively computing primitive integrals for several contracted shells, resulting in better vector utilization, particularly for uncontracted shells. We first present the main features of the OS algorithm, followed by a discussion of the scheme by which OS may be vectorized. Benchmarks are given showing the speedup due to vectorization. Bottlenecks preventing efficient vectorization of some parts of the OS algorithm are analyzed.

OS Algorithm for ERI

For calculation of ERI, our new code, SIMINT, follows the typical OS method. What follows is a brief overview – we direct the reader to the literature for a more thorough discussion.[2–7]

The goal is to calculate the ERI of a quartet of contracted Gaussian basis functions ϕ

$$(\phi_A\phi_B|\phi_C\phi_D) = \int \phi_A(\boldsymbol{r}_1)\phi_B(\boldsymbol{r}_1)\frac{1}{r_{12}}\phi_C(\boldsymbol{r}_2)\phi_D(\boldsymbol{r}_2)d\boldsymbol{r}_1 d\boldsymbol{r}_2 \quad (1)$$

where ϕ_A represents a linear combination of primitive Gaussian functions χ centered on $\boldsymbol{A} = (A_x, A_y, A_z)$

B. P. Pritchard, E. Chow
School of Computational Science and Engineering, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, Georgia, 30332-0765
E-mail: echow@cc.gatech.edu

Contract grant sponsor: Intel Parallel Computing Center grant

© 2016 Wiley Periodicals, Inc.

$$\phi_A = \sum_i c_{Ai}\chi_{Ai} \tag{2}$$

and with identical equations for ϕ_B, ϕ_C, and ϕ_D. The resulting contracted ERI is then rewritten as a fourfold summation over primitive ERI

$$(\phi_A\phi_B|\phi_C\phi_D) = \sum_{\mu\nu\lambda\sigma} c_{A\mu}c_{B\nu}c_{C\lambda}c_{D\sigma}\left[\chi_{A\mu}\chi_{B\nu}|\chi_{C\lambda}\chi_{D\sigma}\right] \tag{3}$$

where parentheses denote contracted quartets and square brackets represent primitive quartets. Primitives χ_A, χ_B, χ_C, and χ_D represent Gaussian basis functions (centered on ***A***, ***B***, ***C***, and ***D***, respectively), defined as

$$\chi_A = (r_x - A_x)^{i_x}(r_y - A_y)^{i_y}(r_z - A_z)^{i_z}e^{-ar^2} \tag{4}$$

$$\chi_B = (r_x - B_x)^{j_x}(r_y - B_y)^{j_y}(r_z - B_z)^{j_z}e^{-br^2} \tag{5}$$

$$\chi_C = (r_x - C_x)^{k_x}(r_y - C_y)^{k_y}(r_z - C_z)^{k_z}e^{-cr^2} \tag{6}$$

$$\chi_D = (r_x - D_x)^{l_x}(r_y - D_y)^{l_y}(r_z - D_z)^{l_z}e^{-dr^2} \tag{7}$$

with exponents a, b, c, d and with total angular momentum quantum numbers $l_A = i_x + i_y + i_z$, etc. Often, the bracket notation is recycled to refer to the angular momentum instead. That is, $[l_Al_B|l_Cl_D]$ and $(l_Al_B|l_Cl_D)$ would refer to "generic" primitive and contracted integrals, respectively, of a particular angular momentum class.

The first step of the OS algorithm is to generate auxiliary primitive integrals $[00|00]^{(m)}$ for integers $0 \le m \le L$ and with total angular momentum $L = l_A + l_B + l_C + l_D$. These integrals can be computed via evaluation of the Boys function, defined as

$$F_m(x) = \int_0^1 t^{2m}e^{-xt^2}dt \tag{8}$$

with the $[00|00]^{(m)}$ integrals being calculated as

$$[00|00]^{(m)} = \frac{2\pi^{\frac{5}{2}}}{pq\sqrt{p+q}}F_m(\rho R) \tag{9}$$

The various parameters depend on the coordinates and exponents given by the four basis functions

$$R = |\boldsymbol{P} - \boldsymbol{Q}|^2 \tag{10}$$

$$\rho = \frac{pq}{p+q} \tag{11}$$

$$p = a + b \tag{12}$$

$$q = c + d \tag{13}$$

$$\boldsymbol{P} = \frac{a\boldsymbol{A} + b\boldsymbol{B}}{p} \tag{14}$$

$$\boldsymbol{Q} = \frac{c\boldsymbol{C} + d\boldsymbol{D}}{q} \tag{15}$$

Next, primitive auxiliary integrals of the form $[(l_A + l_B)\,0|00]^{(m)}$ and then $[(l_A + l_B)\,0\,|\,(l_C + l_D)\,0]^{(m)}$ are formed through use of the vertical recurrence relations (VRR). For simplification, we shall adopt the notation used by Helgaker et al.[2] An auxiliary integral (primitive or contracted) can be written as

$$\Theta^{(N)}_{i_x,j_x,k_x,l_x:i_y,j_y,k_y,l_y:i_z,j_z,k_z,l_z} = \left[\chi_{A\mu}\chi_{B\nu}|\chi_{C\lambda}\chi_{D\sigma}\right]^{(N)} \tag{16}$$

with $N = 0$ referring to the actual target integrals. A single recurrence is generally concerned with increasing or decreasing the angular momentum of a single Cartesian component, and this notation is shortened to $\Theta^{(N)}_{ijkl}$ with the Cartesian index (x, y, z) inferred from the context. The VRR for incrementing i_x (with $j = l = 0$) is given as

$$\begin{aligned}\Theta^{(N)}_{i+1,0,k,0} &= X_{PA}\Theta^{(N)}_{i0k0} - \frac{\alpha}{p}X_{PQ}\Theta^{(N+1)}_{i0k0} \\ &+ \frac{i}{2p}\left(\Theta^{(N)}_{i-1,0,k,0} - \frac{\alpha}{p}\Theta^{(N+1)}_{i-1,0,k,0}\right) + \frac{k}{2(p+q)}\Theta^{(N+1)}_{i,0,k-1,0}\end{aligned} \tag{17}$$

where $X_{PA} = P_x - A_x$ and the reduced exponent $\alpha = pq/(p+q)$. Identical equations for incrementing i_y or i_z can be obtained by replacing X_{PA} with Y_{PA} or Z_{PA}, respectively. In addition, increments of k_x can be obtained (again with $j = l = 0$) via

$$\begin{aligned}\Theta^{(N)}_{i,0,k+1,0} &= X_{QC}\Theta^{(N)}_{i0k0} - \frac{\alpha}{q}X_{PQ}\Theta^{(N+1)}_{i0k0} \\ &+ \frac{k}{2q}\left(\Theta^{(N)}_{i,0,k-1,0} - \frac{\alpha}{q}\Theta^{(N+1)}_{i,0,k-1,0}\right) + \frac{i}{2(p+q)}\Theta^{(N+1)}_{i-1,0,k,0}\end{aligned} \tag{18}$$

with $X_{QC} = Q_x - C_x$. Again, similar equations can be derived for incrementing k_y and k_z in the same manner. Incrementing j_x and l_x can be achieved by replacing X_{PA} with X_{PB} and X_{QC} with X_{QD}, respectively, leaving the rest of the right-hand sides of eqs. (17) and (18) the same.

Alternatively, an electron transfer (ET) step may be used instead to generate $[(l_A + l_B)\,0\,|\,(l_C + l_D)\,0]$ from $[(l_A + l_B + l_C + l_D)\,0\,|00]$, rather than the method outlined above. While performance was satisfactory, it was found that there exists an inherent loss of precision in the ET equations, where a subtraction exists between two floating point numbers of similar magnitude (see Supporting Information). Overcoming this would be very difficult, particularly in vectorized code. Therefore, eq. (18) was used instead.

Final integrals $[l_Al_B|l_Cl_D]$ or $(l_Al_B|l_Cl_D)$ are then generated by the horizontal recurrence relations (HRR)

$$\Theta^{(N)}_{i,j+1,k,l} = \Theta^{(N)}_{i+1,j,k,l} + X_{AB}\Theta^{(N)}_{i,j,k,l} \tag{19}$$

$$\Theta^{(N)}_{i,j,k,l+1} = \Theta^{(N)}_{i,j,k+1,l} + X_{CD}\Theta^{(N)}_{i,j,k,l} \tag{20}$$

with $X_{AB} = A_x - B_x$ and $X_{CD} = C_x - D_x$ (and similarly for y and z). Note that reversing the direction of the relation involves only changing the sign on X_{AB} and X_{CD}. Calculation of the final integrals via HRR can proceed via several different paths; we have chosen the simple algorithm given by Makowski, which seeks to minimize the number of intermediates required to be calculated and stored.[27]

There are a few important features of this algorithm to keep in mind. As the HRR steps do not depend on any variables specific to primitives, the HRR can occur outside of the

```
loop A : shells
loop B : shells
loop C : shells
loop D : shells
    loop i : primitives(A)
    loop j : primitives(B)
    loop k : primitives(C)
    loop l : primitives(D)
        p = exp(A,i) + exp(B,j)
        q = exp(C,k) + exp(D,l)
        rho = (p*q)/(p+q)
         ....
        contraction/accumulation
    end loop l
    end loop k
    end loop j
    end loop i

    ...HRR...

end loop D
end loop C
end loop B
end loop A
```

(a)

```
loop AB : shellpairs
loop CD : shellpairs
    loop ij: primitivepairs(AB)
        p = expsum(AB,ij)

        loop kl : primitivepairs(CD)
            q = expsum(CD,kl)
            rho = (p*q)/(p+q)
             ...
            contraction/accumulation
        end loop kl
    end loop ij

    ...HRR...

end loop CD
end loop AB
```

(b)

Figure 1. Different possible loop structures for calculating ERI. The calculation of ρ within the primitive loop is shown as an example. a) shows a traditional fourfold loop, first over shells then over primitives of those shells. b) flattens the fourfold loops over shells and primitives into twofold loops over pairs of shells and pairs of primitives. The values within the shell pair data structures are calculated prior to entering the ERI code. exp represents the exponents of the basis functions on the centers, and expsum represents the sum of exponents of two Gaussian basis functions.

primitive loop. The penalty for doing so, however, is the contraction of not only $[(l_A+l_B)0|(l_C+l_D)0]$ but also of any intermediates required by eqs. (19) and (20). For highly contracted basis sets, this is generally beneficial as the gain in efficiency of performing the HRR outside the primitive loop is more than that lost by having to perform several contractions, although this depends on the level of contraction of the basis set.

Lastly, for basis sets without general contractions (or if general contractions are not implemented), contraction coefficients [c from eq. (2)] may be introduced early (typically immediately after computation of the Boys function). Doing so reduces the cost of contractions later.

Optimization

Loop structure and shell pairs

Figure 1 shows two possible ways to structure the loops within an ERI calculation. Figure 1a shows a typical, straightforward loop structure, with loops over all four shell indicies occurring before loops over primitives within the shell. As vectorization is, as a general rule, best implemented on the innermost loop, this structure was considered a poor candidate as the innermost loop (over the number of primitives in a single shell, indexed by l) would often be too short to obtain meaningful improvements with vectorization. The structure in Figure 1b is considered more conducive to vectorization than Figure 1a. The innermost loop is now a flattened loop over all possible combinations of primitives from shells C and D. This flattening increases the number of iterations within this innermost loop, allowing for better vectorization performance. As a further optimization, values that can be computed from just a pair of shells (such as p, $\boldsymbol{P}$, X_{PA}, etc.), can be precomputed and stored ahead of time. This data is often referred to as "shell pair" data. In addition, a prefactor for the shell pair can be calculated that includes the product of the contraction coefficients as well as some pieces of the prefactor in eq. (9), slightly reducing the expense for ERI with very low L. The loop structure in Figure 1b was implemented in SIMINT, with the innermost loop as the target of vectorization.

Despite flattening, it is not uncommon for the primitive kl loop to still have only one iteration, as would occur with basis sets that contain uncontracted shells. As these shells often have higher values of L (therefore, requiring lots of work in this loop), the resulting unfilled vectors may seriously reduce the benefits of vectorization. The solution to this, implemented in SIMINT, is to allow for the ERI function to compute multiple shell quartets (of the same class) consecutively in one function call, and flatten the shell pair data of multiple shell pairs into a single structure. In that case, the shell pair information is best laid out in what is often referred to as a "structure of arrays" (SOA) style (Fig. 2). The data held within this structure is still calculated prior to entering the ERI function; however, when laid out in this fashion the innermost loop will often have more iterations than before.

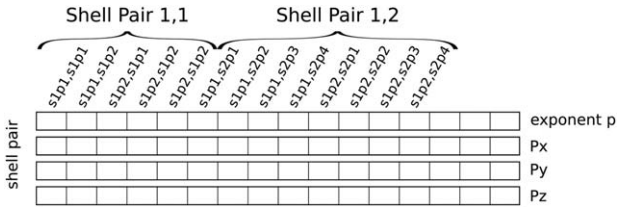

Figure 2. Storage of shell pair information within SIMINT. A single structure holds data from multiple pairs of shells sequentially within an array. In this example, shell s1 contains two primitives p1 and p2. Shell s2 contains four primitives p1–p4.

Vectorization

There are a few different methods by which the OS algorithm may be vectorized. A simple replacement of the scalar arithmetic types with vector types, with no change to the interface or algorithm, results in a calculation that is vectorized in such a way that each lane of a SIMD vector register computes a single *contracted* integral (Fig. 3). While straightforward to implement, this method is inefficient as different contracted shell quartets may contain very different numbers of primitive integrals, resulting in incompletely filled vectors within the primitive loop. This would be an increasing concern as the vector length increases, as the number of stored SIMD words must effectively be the maximum number of primitives of all contracted quartets in the vector.

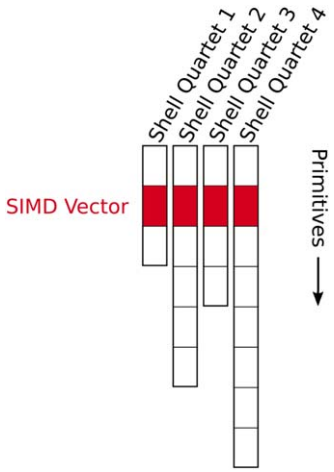

Figure 3. Example of the simple approach to vectorizing ERI. A SIMD lane represents a primitive from a single contracted shell quartet or contracted integral (for example, the first lane always maps to the first contracted integral). In this figure, the number of primitives differs for each contracted integral, resulting in unfilled vectors as the SIMD vector moves downward. In memory, the index of the contracted integral is the fastest index. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

In SIMINT, another approach was taken. Here, the loop over *primitives* was the (main) target of vectorization, as many modern basis sets utilize at least moderate levels of contraction in shells with low to moderate angular momentum. Under this scheme, the entire primitive loop (except for the Boys function) is completely vectorized, with lanes within a SIMD vector corresponding to primitive integrals of any contracted shell quartet (see Fig. 4). The algorithm uses the fourfold loop structure outlined in Figure 1b; the pseudocode for the vectorized version is listed in Figure 5. The first loop is a straightforward loop over individual (precomputed) contracted shell pairs from the bra side of the requested ERI. The second is a similar loop over the shell pairs of the ket side, the difference being that this second loop combines several of these contracted ket shell pairs into a batch of a preset size. In this case, a vector in Figure 4 may contain primitives from different contracted shell quartets. This allows for efficient computation of uncontracted or lightly contracted quartets, with the batch size controlling the memory required to store contracted intermediates.

Figure 6 shows the layout of a shell pair structure containing data from multiple contracted shells and its use in vectorized SIMINT. The third loop in Figure 5 runs over single primitive shell pairs (combining primitives from the first and second center), which are then broadcast within a SIMD word. The fourth loop will read in a SIMD word from the ket shell pair. Vectorization is enhanced when a shell pair structure contains information from multiple contracted shell pairs in the SOA format (Loop structure and shell pairs section). In this manner, the VRR steps are performed in SIMD fashion, with all variables in eqs. (17) and (18) substituted with vectors with each lane representing a single primitive quartet (as in Fig. 4).

As many shell quartets may be performed within a single function call, memory management becomes important. To reduce memory usage and increase the likelihood of data being in the processor cache, contracted integrals are computed in batches (the second loop), such that intermediates for only a few contracted integrals required for HRR need to be held at any point. Primitive integrals are computed, and then contracted into this intermediate workspace. To facilitate this, padding is inserted between each batch within the shell pair structure to allow for aligned loading beginning at the start of the fourth loop.

Contraction

To obtain the best SIMD vector utilization (by reduction of the number of unfilled vectors), a single vector may contain

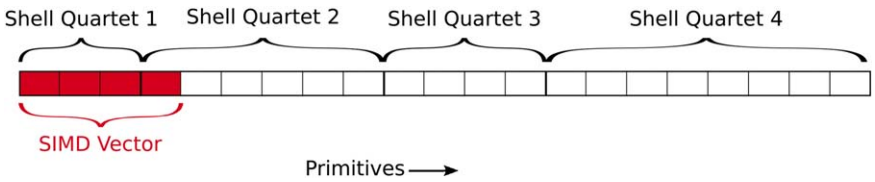

Figure 4. Example of primitive vectorization in SIMINT. A single SIMD vector stores primitive integrals from any contracted shell quartet as it moves left to right. In memory, the index of the primitive integral is the fastest index. Only one vector (4 primitives) for each integral need to be stored at one time within the primitive loop. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

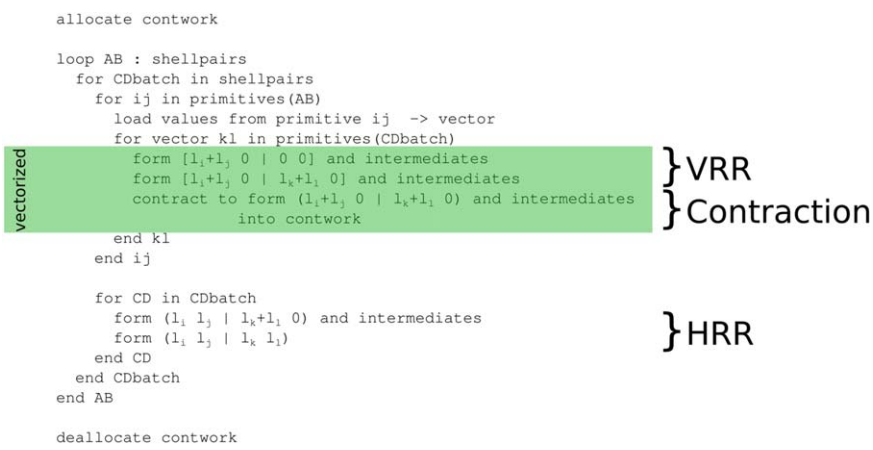

Figure 5. Basic pseudocode and loop structure for OS with a vectorized primitive loop, as implemented in SIMINT. Intermediates required for HRR are stored in contwork. The index CDbatch counts the batches of contracted integrals being done consecutively in the primitive loop, and l_i, l_j, l_k, and l_l are the angular momentum of the four centers. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

primitives from more than one contracted shell quartet. Vectorizing the primitive loop in this way will also have benefits for uncontracted quartets, as the single primitive from multiple contracted quartets may occupy the same vector. Due to this requirement, however, contraction at the end of the primitive loop becomes relatively complex and inefficient, as it requires logic to handle cases where contracted shell boundaries occur within a SIMD vector (see Fig. 7). In some cases, a somewhat efficient algorithm involving SIMD swizzle operations can be used. In particular, if the vector contains primitives from a single contracted shell quartet, a combination of horizontal addition instructions may be used to alleviate some of the expense of breaking the vector into individual elements and summing them.

For the most part, however, this accumulation is relatively expensive, particularly for high values of L. This is exacerbated by the need to contract all of the intermediates required by HRR. This is expected to scale poorly with respect to both the value of L and the length of the SIMD vector. Despite this, benchmarks (Benchmark Results section) show that it is still beneficial to perform the calculations within the primitive loop in SIMD fashion.

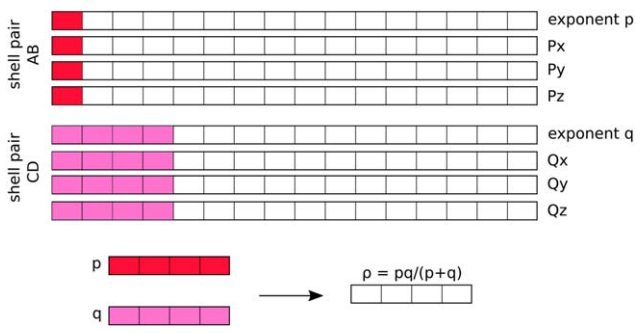

Figure 6. Storage and loading of shell pair information within SIMINT. Single values are taken from shell pair AB and broadcast within a vector. Multiple values are taken from CD. Intermediate values are then created from these vectors, with each lane representing a primitive integral (shell quartet). At the bottom is an example of how two vectors (containing exponents) are combined to form ρ in eq. (11). [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Horizontal recurrence

The HRR steps are performed outside of the primitive loop. As currently implemented, the ordering of the contracted integrals (Fig. 7) is not conducive to vectorization of the horizontal recurrence step in manner similar to that of the primitive loop. That is, horizontal vectorization of the HRR (one contracted shell per SIMD lane) would require the index of the contracted shell quartet be the fastest index. Implementation of this is possible, but would require permutation of the "final" integrals to the more commonly accepted ordering, where the shell quartet is the slowest index.

We have not explicitly vectorized the HRR steps as it can be mostly vectorized automatically by the compiler. Specifically, HRR in the bra part of the quartet [eq. (19)] may be automatically vectorized. To see this, consider the indexing of the Cartesian components. For a quartet $(I_A I_B|I_C I_D)$, with angular momentum I_A containing $N_A=(I_A+1)(I_A+2)/2$ Cartesian components (and similarly for I_B, I_C, and I_D), the index for a specific Cartesian ERI $(ij|kl)$, with i,j,k,l being indicies of the Cartesian component of a shell, is calculated as

$$\text{idx}=iN_BN_CN_D+jN_CN_D+kN_D+l \tag{21}$$

or, combining the indicies within the bra and ket

$$\text{idx}=IN_{\text{ket}}+K \tag{22}$$

$$N_{\text{ket}}=N_CN_D \tag{23}$$

$$I=iN_B+j \tag{24}$$

$$K=kN_D+l \tag{25}$$

Given that the ket is identical for all terms in eq. (19), one particularly efficient way to perform the bra HRR is to carry out all required steps within a loop over the combined ket index K over the range $[0,N_{\text{ket}})$. As the integrals are stored with k (and therefore K) effectively being the fastest index, the compiler may efficiently vectorize the loop, performing the bra HRR for several values of K at once. There does not appear to be such a beneficial layout for ket HRR, however.

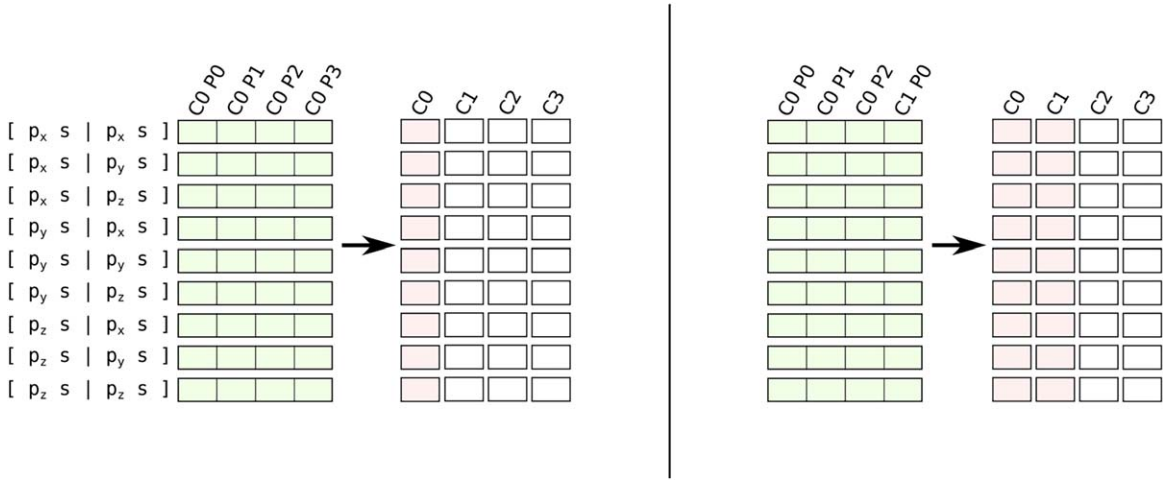

Figure 7. Example of contraction of vectors at the end of the primitive loop. On the left, all lanes of the SIMD vector belong to the same contracted integral. On the right, the last lane belongs to a different contracted integral than the others. Note the memory layout of the contracted integrals versus the vectorized primitive integrals. In the labels, C is the contracted quartet index, P is the primitive quartet index within the contracted quartet. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Code Structure

SIMINT consists of a flexible generator written in C++ which is used to generate pure C source code. The generator will create a function and source file per quartet class $(I_A I_B | I_C I_D)$ that utilizes intrinsics based on the given CPU flags of the target processor or microarchitecture. Generally, the functions are generated for $I_A \leq I_B$, $I_C \leq I_D$, and $I_A + I_B \leq I_C + I_D$. An experimental feature is available where the generator will create source files without restriction to the order of the angular momentum. The integrals in this case are generated directly via VRR and HRR as before; in that case, SIMINT does not permute the basis functions and the final integrals. The Cartesian ordering on output follows the default order of LIBINT.

Currently, SIMINT supports only the SSE and AVX instruction sets, and must be compiled with the Intel compiler (2015 and above). Support for other compilers and instruction sets is planned and should be straightforward to implement.

Benchmark Results

Benchmarks were performed on a machine containing an Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) processor. These processors are capable of AVX instructions; therefore, 256-bit intrinsics were used when possible. The thread affinity was set to "scatter" via the KMP_AFFINITY environment variable. See the Supporting Information for the compiler flags used. The benchmarks were performed on benzene (C_6H_6, experimental structure[28,29]) with the aug-cc-pVTZ[30] and the triple-zeta atomic natural orbital (ANO-TZ) basis sets. Basis functions were sorted into their respective classes prior to benchmarking; this sorting is not included in the timings.

Sections of code were timed by counting processor ticks via the rdtsc processor instruction. Reported timings are the number of processor ticks required to compute all integrals of the given quartet for the given molecule and basis set.

Comparisons were made to two common ERI libraries: LIBINT[18] (v2.0.5) and LIBERD.[31] LIBINT and LIBERD were compiled with AVX flags, allowing for compiler-generated vectorization; explicit vectorization was not enabled in LIBINT.† Comparisons were also made to a scalar version of SIMINT, which was generated without intrinsics and with explicit disabling of compiler vectorization. Validation of accuracy of computed integrals was carried out by comparing the values for contracted ERI across all libraries, and by comparison with values generated via the validation functionality distributed with LIBINT.

Both SIMINT and LIBINT were developed to take advantage of fused multiply-add instructions, and LIBINT was compiled with this functionality enabled. Benchmarks for SIMINT utilize FMA when compiled with vectorization, but not when compiled in scalar mode. LIBINT is neither responsible for evaluating the Boys function nor for calculating many of the prerequisite values [i.e., eqs. (10–15)] required for ERI calculation. The benchmark results for LIBINT include the time to compute the Boys function via the same method used by SIMINT (interpolation via Taylor series or utilization of the long range approximation); therefore, care must be taken in interpreting the results for quartets with very low L, where evaluating these prerequisites is expected to be a significant portion of the overall computation time. In particular, the $(ss|ss)$ quartet does not require calling any LIBINT code at all. Therefore, the timing for $(ss|ss)$ should be roughly equivalent to that of the scalar version of SIMINT.

The time required to compute shell pair information was measured and was found to be negligible (generally less than 0.1% of the total time for a given quartet class). Nevertheless, this time is included in the timings for SIMINT and LIBINT. The same shell pair routines from SIMINT were used to calculate the prerequisite data for LIBINT, although the data required copying from the SIMINT format to the LIBINT structures. The time required for this copy is not included in the timings for LIBINT.

†Some initial benchmarks were obtained with explicit vectorization in LIBINT; however, as this functionality is still considered experimental, results are not included here. LIBINT utilizes the simplified vectorization scheme (Fig. 3), and therefore, performance is expected to be poor.

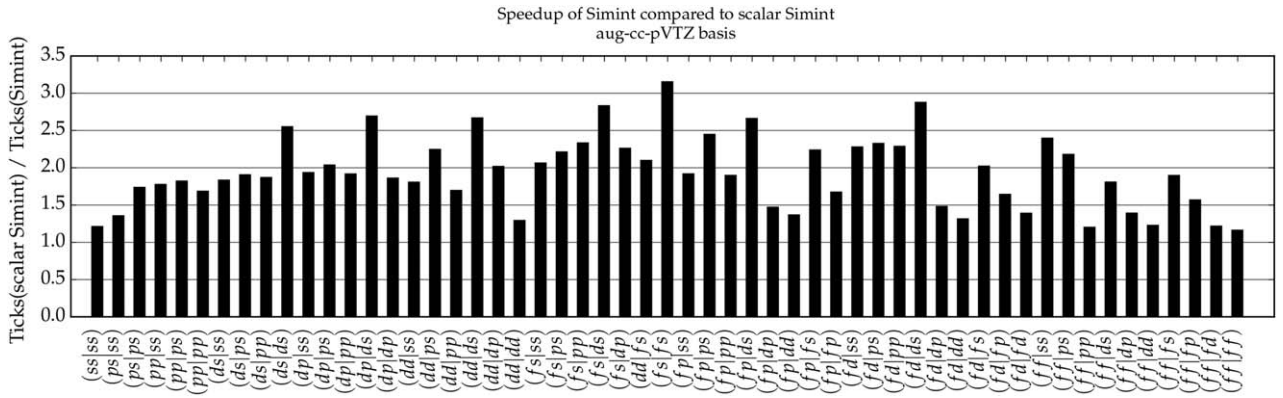

Figure 8. Speedup of SIMINT compared to scalar SIMINT. Benchmarks performed with benzene with the aug-cc-pVTZ basis.

Any required permutation or spherical transformation of the basis functions or final integrals (as would be needed in production computational chemistry software) are not included in any benchmark. Permutational symmetry within individual shell quartets was also not taken into account, and screening was not performed. Any general contractions within the basis set were converted to additional segmented shells.

Speedup with the aug-cc-pVTZ basis

Speedup is calculated as the ratio of time for the baseline calculation versus the time for the optimized calculation. Figure 8 shows the speedup of vectorized SIMINT over its scalar counterpart with the aug-cc-pVTZ basis. The speedup over scalar is generally in the range of 1.5 to 3.0, with quartets of type $(fs|fs)$ performing the best. The worst performing quartets are $(ss|ss)$ and several quartets of high L, which are only slightly faster with vectorized code than with scalar code. Only 13 quartet types have speedup below 1.5, and all are at least as fast as the scalar code. In general, quartets in which most computation is performed in the primitive loop display the best speedup. These are generally any quartets in which eq. (17) must be evaluated with $I_A, I_C \neq 0$. Quartets that contain HRR computation only in the bra of the quartet (i.e., $I_A, I_B, I_C \neq 0$) or where very little HRR is required in the ket part ($I_D = 1$) also tend to have very good speedup over the scalar code. This advantage comes from the fact that there is a lot of work performed in the primitive loop in forming the intermediates required for HRR, and from the fact that the HRR is effectively auto-vectorized by the compiler in these cases (see Horizontal recurrence section). This auto-vectorization was explicitly disabled for the scalar code.

Quartets with the highest values of total angular momentum L demonstrate a speedup of between 1.0 and 1.5, and have especially poor speedup when $I_D \neq 0$. For example, speedups are measured as 2.7, 2.0, and 1.3 for $(dd|ds)$, $(dd|dp)$, and $(dd|dd)$, respectively. One reason for this that the increasing amount of computation being performed in the unvectorized ket HRR loops. Another factor for the poorer speedup is the expense of the contraction step at the end of the primitive loop. This is worst with quartets of high L value, where many intermediates must be contracted before exiting the primitive loop. For example, contractions are much more expensive for $(dd|dd)$ than for $(ds|ds)$; as there is no HRR required for $(ds|ds)$, only primitive $[ds|ds]$ integrals must be contracted. The HRR for $(dd|dd)$ requires contraction of $[ds|ds]$, $[ds|fs]$, $[ds|gs]$, $[fs|ds]$, $[fs|fs]$, $[fs|gs]$, $[gs|ds]$, $[gs|fs]$, and $[gs|gs]$ primitives (all of which are obtained via VRR). Inefficiency in contraction due to vectors containing primitive integrals from multiple contracted quartets (which requires accessing individual elements of the vector) becomes exacerbated due to the need to contract all these different quartets. Therefore, for the aug-cc-pVTZ basis, the expense of the contractions is worsened by the uncontracted nature of the d and f quartets in benzene. For entirely uncontracted shell quartets, each lane within a vector is from a different contracted quartet, which is particularly slow. How to efficiently handle these cases is this scheme still an open question, and may involve logic to determine when it is more efficient to perform the HRR within the primitive loop rather than performing the expensive contraction, similar to the PRISM algorithm.[32]

Some quartets with very small L, such as $(ss|ss)$ and $(ps|ss)$, also have somewhat low speedup. These quartets spend much of their time in calculation of the Boys function, which contains poorly vectorized table lookups. In addition, calculation of divisions and square roots, also needed in evaluation of the Boys function but also for the prefactor in eq. (9), are also a greater fraction of the computational expense for these quartet types. On the Haswell microarchitecture, these instructions execute on the same port, are not pipelined, and are not fully vectorized.[33] As L increases, the square root and division become a smaller fraction of the expense, and as expected the speedup of the vectorized code over the scalar code increases.

Figure 9 shows the speedup of SIMINT over LIBINT (compiled with vectorization applied automatically by the compiler). Overall, the speedup is generally slightly better (around 1.5 to 3.5) than over scalar SIMINT; this is greater than the speedup over scalar SIMINT, indicating that the generated scalar code is somewhat faster than LIBINT as well. Quartets with high L and requiring much work in the HRR portion of the code again perform the worst. Quartets that perform particularly well when compared to LIBINT are also those that perform well against scalar SIMINT, namely the $(fs|fs)$, $(ff|fs)$, and $(fd|ss)$

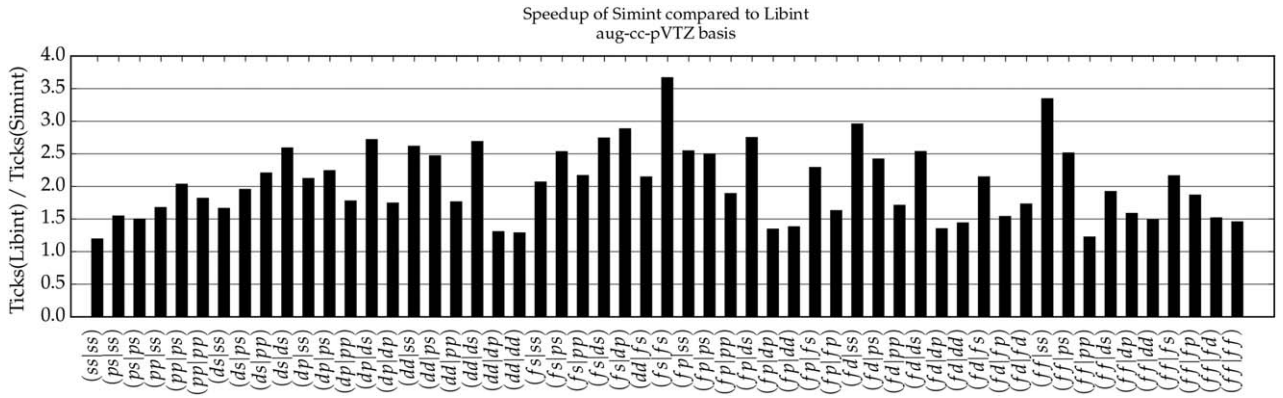

Figure 9. Speedup of SIMINT compared to LIBINT. Benchmarks performed with benzene with the aug-cc-pVTZ basis.

quartets. This is expected, as LIBINT also uses OS recurrence relations in calculation of its integrals. (It should be noted again that LIBINT is not responsible for calculation of the Boys function.) In addition, for quartets with very low L, a relatively large portion of time is spent in the preparation stage prior to calling LIBINT. At this stage, several factors are calculated without explicit vectorization.[‡]

Figure 10 shows the comparison of LIBERD with SIMINT. With this molecule and basis set, SIMINT is often much faster than LIBERD. For very low angular momentum, SIMINT has a speedup of only 2.0 to 5.0, again likely due to the bottleneck in divisions, square roots, and the Boys function. For high angular momentum, LIBERD performance improves, likely as the efficiency of Rys quadrature in this regime begins to take over. For the moderate values of angular momentum, SIMINT displays a speedup of between 4.0 and 11.0, particularly for quartets where HRR is not performed in the ket part ($I_D = 0$). Once again, SIMINT performs relatively poorly for quartets such as $(dd|dd)$ due to the required contractions and the unvectorized ket HRR sections taking a larger amount of time.

Speedup with the ANO-TZ basis

As seen in the previous section, the speedup of SIMINT has a dependence on the amount of contraction needed to be performed at the end of the primitive loop. The efficiency of this step depends on whether or not the vectors contain primitive quartets from more than one contracted shell—as mentioned in Contraction section, it is more efficient to perform the contractions if the vector contains primitive quartets from only one contracted shell. It is also expected that the speedup would also depend on the relative amount of work performed in the (vectorized) primitive loop versus outside the loop (i.e., HRR). Taking these into account, the uncontracted nature of several of the shells in aug-cc-pVTZ provide a worst-case scenario for SIMINT, and the code is expected to perform better with a more highly contracted basis set.

[‡]Also at this stage, data is copied from shell-pair data structures to LIBINT internal data structures. This copying may be particularly expensive, as the layout does not lend itself well to optimization or vectorization (by the compiler) nor to simple memory operations (such as memcpy). The timing of this copy, however, is not included in the timings here.

Figure 11 shows the speedup of SIMINT over the corresponding scalar SIMINT code when using the ANO-TZ basis set. This basis set has the same number of contracted integrals as aug-cc-pVTZ; however, it contains a much greater level of contraction and therfore a much greater number of primitive integrals. In this case, SIMINT exhibits a speedup of between 2.0 and 4.0, with 23 quartets having a speedup greater than 3. This is substantially better than with the aug-cc-pVTZ basis, as predicted. Once again, quartets with $I_D = 0$ (or even $I_D = 1$) perform much better than other quartets with similar L, and the worst-performing quartets tend to have relatively little computation done within the vectorized primitive loop. Figure 12 shows the speedup of vectorized SIMINT over LIBINT, where SIMINT shows a similar or better speedup over LIBINT than over scalar SIMINT, with 19 quartets showing a speedup of between 2 and 3, and 31 having a speedup greater than 3.0. The patterns seen in Figure 11 with regards to the efficiency of quartets with $I_D = 0$ is still evident in Figure 12. The advantage of SIMINT over LIBINT is similar for both the ANO-TZ and aug-cc-pVTZ bases.

The speedup of SIMINT over LIBERD for the ANO-TZ basis (Fig. 13) is not as good as for the aug-cc-pVTZ basis, although it is still substantial, with most quartets showing a speedup of at least 3.0, and many showing a speedup greater than 5.0. For very large L, SIMINT and LIBERD are once again on par, and it is expected that LIBERD will become faster for $L > 12$, even with the vectorization performed in SIMINT.

Conclusions and Future Work

We have presented a scheme whereby the primitive loop of the calculation of ERI (via OS recurrence relations) is vectorized via compiler intrinsics. Additionally, some calculations performed in the HRR step are efficiently auto-vectorized by the compiler. This scheme was implemented in a new code, SIMINT. Benchmarks against other ERI codes show favorable results on the Haswell microarchitecture, which is capable of AVX instructions. In particular, SIMINT displays very good speedup over its scalar counterparts when calculating quartets of the form $(I_A I_B | I_C 0)$, due to the heavy computation done in the primitive loop and the auto-vectorization of HRR. This efficiency may be very advantageous when combined with density fitting,[34–37] as only

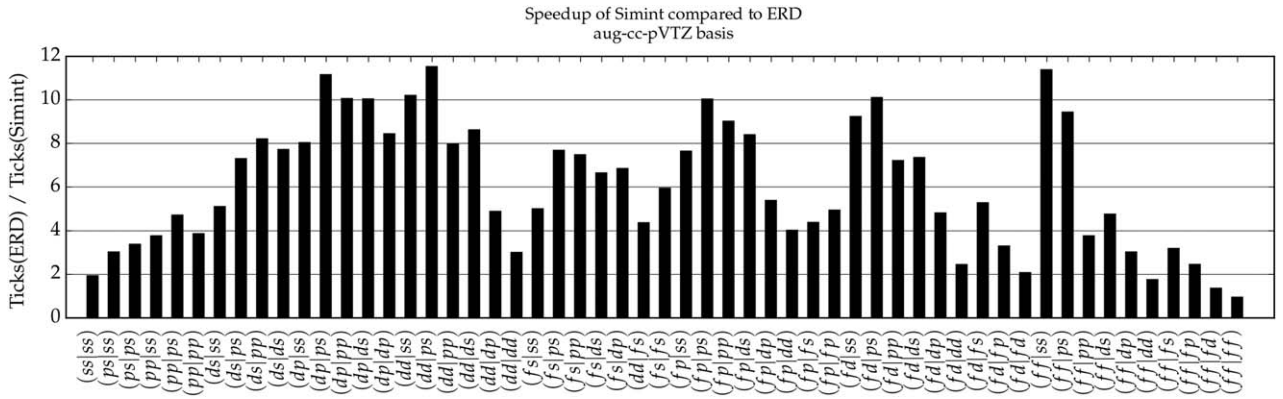

Figure 10. Speedup of SIMINT compared to LIBERD. Benchmarks performed with benzene with the aug-cc-pVTZ basis.

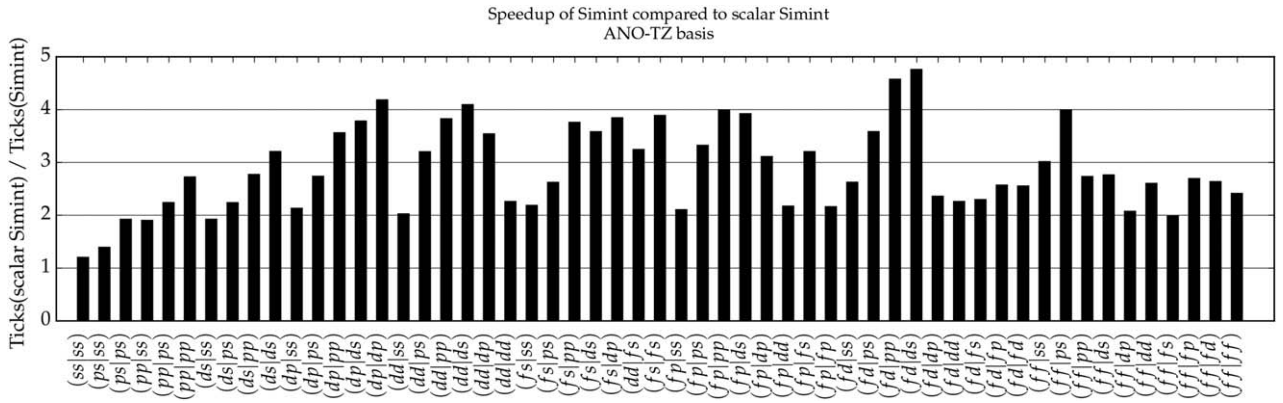

Figure 11. Speedup of SIMINT compared to scalar SIMINT. Benchmarks performed with benzene with the ANO-TZ basis.

three-center and two-center integrals need to be computed, which may be done via $(I_A I_B|I_C 0)$ and $(I_A 0|I_C 0)$ functionality, respectively.

Vectorizing ERI for CPUs in the way outlined above has some benefits over previous GPU implementations. One such advantage is that data can be explicitly shared within the SIMINT scheme, whereas data must be copied to separate thread blocks in a GPU. Another is that the contracted integrals can be computed only a few at a time and then accumulated, whereas the implementation in Ref. 12 requires storage (at one point) of all primitive integrals being calculated by the GPU, which can be expensive for high angular momentum shell quartets.

Future work will focus on improving the vectorization efficiency of the contraction and HRR steps which, as shown in Benchmark Results section, are major sources of inefficiency. One possibility is to generate separate functions for different levels of contraction – this is essentially the idea behind the PRISM algorithm.[32] This would allow for skipping the expensive contractions and performing the HRR within the primitive loop when the contraction level is low. As these uncontracted

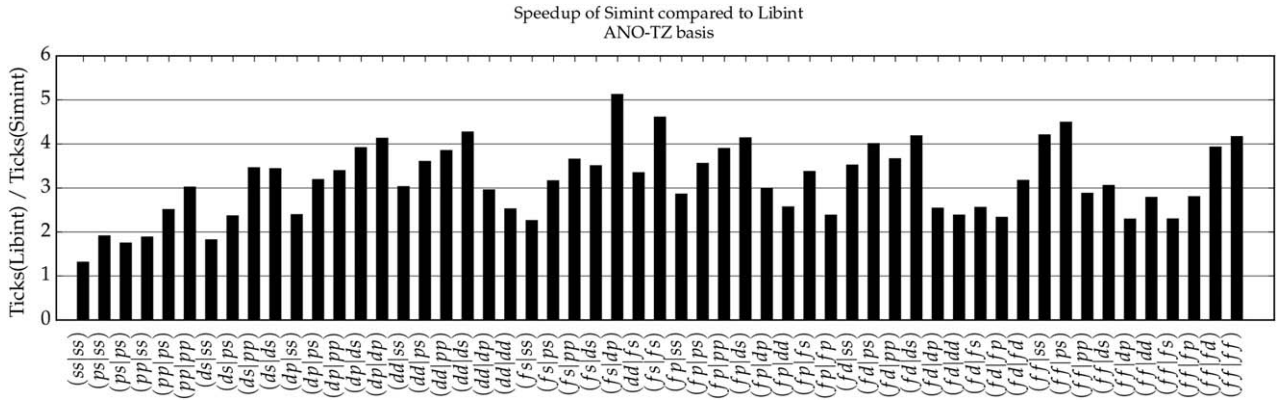

Figure 12. Speedup of SIMINT compared to LIBINT. Benchmarks performed with benzene with the ANO-TZ basis.

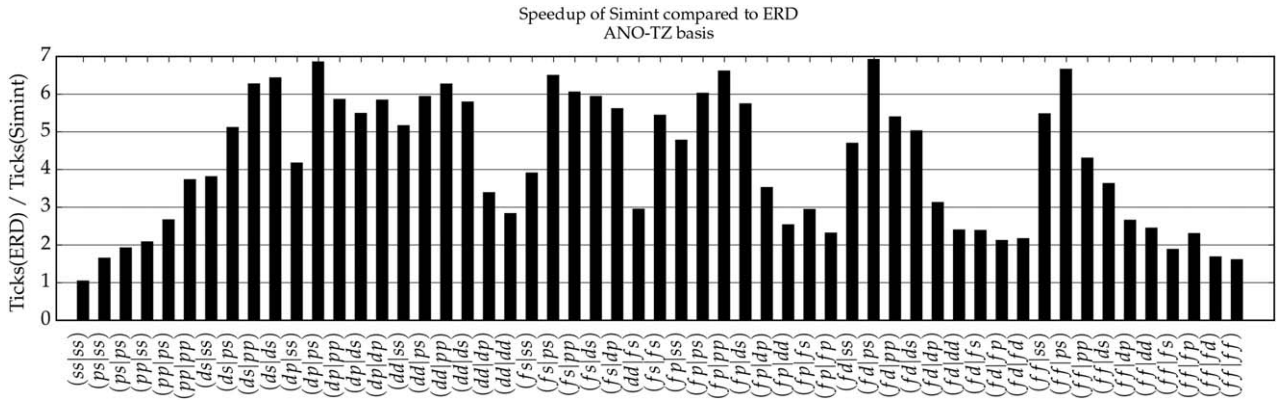

Figure 13. Speedup of SIMINT compared to LIBERD. Benchmarks performed with benzene with the ANO-TZ basis.

shells are often of higher angular momentum, another possibility is the implementation of Rys quadrature[8–10] for those shells.

We are currently investigating how to best use SIMINT with screening in such a way that calculation of some integrals may be skipped. Screening is the responsibility of the calling program; however, it may be needed to be performed differently to pass multiple shell quartets to SIMINT.

It seems likely that this scheme will scale relatively well with increasing vector lengths—work is proceeding on implementing this scheme on hardware with longer vector widths. In particular, investigations are being performed into utilizing the scheme on the Intel Xeon Phi, which contains 512-bit vectors capable of working with eight doubles.

Keywords: electron repulsion integrals · two-electron integrals · vectorization · Obara–Saika · horizontal vectorization · SIMD

How to cite this article: B.P. Pritchard, E. Chow. *J. Comput. Chem.* **2016**, *37*, 2537–2546. DOI: 10.1002/jcc.24483

Additional Supporting Information may be found in the online version of this article.

[1] L. E. McMurchie, E. R. Davidson, *J. Comput. Phys.* **1978**, *26*, 218.
[2] T. Helgaker, P. Jørgensen, J. Olsen, Molecular Electronic-Structure Theory; Chichester: Wiley, **2000**.
[3] S. Obara, A. Saika, *J. Chem. Phys.* **1988**, *89*, 1540.
[4] S. Obara, A. Saika, *J. Chem. Phys.* **1986**, *84*, 3963.
[5] M. Head-Gordon, J. A. Pople, *J. Chem. Phys.* **1988**, *89*, 5777.
[6] R. Lindh, U. Ryu, B. Liu, *J. Chem. Phys.* **1991**, *95*, 5889.
[7] P. Tracy, T. P. Hamilton, H. F. Schaefer, *Chem. Phys.* **1991**, *150*, 163.
[8] M. Dupuis, J. Rys, H. F. King, *J. Chem. Phys.* **1976**, *65*, 111.
[9] J. Rys, M. Dupuis, H. F. King, *J. Comput. Chem.* **1983**, *4*, 154.
[10] M. Dupuis, A. Marquez, *J. Chem. Phys.* **2001**, *114*, 2067.
[11] K. Ishida, *Int. J. Quantum Chem.* **1996**, *59*, 209.
[12] I. S. Ufimtsev, T. J. Martinez, *J. Chem. Theory Comput.* **2008**, *4*, 222.
[13] K. Yasuda, *J. Comput. Chem.* **2008**, *29*, 334.
[14] Y. Miao, K. M. Merz, *J. Chem Theory Comput.* **2013**, *9*, 965.
[15] A. Asadchev, V. Allada, J. Felder, B. M. Bode, M. S. Gordon, T. L. Windus, *J. Chem. Theory Comput.* **2010**, *6*, 696.
[16] K. A. Wilkinson, P. Sherwood, M. F. Guest, K. J. Naidoo, *J. Comput. Chem.* **2011**, *32*, 2313.
[17] Q. Sun, *J. Comput. Chem.* **2015**, *36*, 1664.
[18] E. F. Valeev, A library for the evaluation of molecular integrals of many-body operators over Gaussian functions; **2014**. Available at: http://libint.valeyev.net/. Last accessed: Feb 5, 2016.
[19] H. Shan, B. Austin, W. De Jong, L. Oliker, N. Wright, E. Apra, Performance tuning of fock matrix and two-electron integral calculations for NWChem on leading HPC platforms, *Lecture Notes in Computer Science*, Vol *8551*, pp. 261–280, **2014**.
[20] E. Chow, X. Liu, S. Misra, M. Dukhan, M. Smelyanskiy, J. R. Hammond, Y. Du, X. K. Liao, P. Dubey, *Int. J. High Perform. Comput. Appl.* **2016**, *30*, 85.
[21] T. Ramdas, G. K. Egan, D. Abramson, K. K. Baldridge, *Comput. Phys. Commun.* **2008**, *178*, 817.
[22] T. Ramdas, G. K. Egan, D. Abramson, K. K. Baldridge, *Chem. Phys.* **2008**, *349*, 147.
[23] T. Ramdas, G. K. Egan, D. Abramson, K. K. Baldridge, *Comput. Phys. Commun.* **2009**, *180*, 1221.
[24] M. F. Guest, S. Wilson, In Supercomputers in Chemistry; P. Lykos, I. Shavitt, Eds.; Washington, DC: American Chemical Society, **1981**; Vol. *173*; Chapter 2, pp. 1–37.
[25] V. R. Saunders, M. F. Guest, *Comput. Phys. Commun.* **1982**, *26*, 389.
[26] R. Ernenwein, M. M. Rohmer, M. Benard, *Comput. Phys. Commun.* **1990**, *58*, 305.
[27] M. Makowski, *Int. J. Quantum Chem.* **2007**, *107*, 30.
[28] I. Johnson, D. Russell NIST Computational Chemistry Comparison and Benchmark Database, Release 17b; NIST Standard Reference Database Number 101, **2015**. Available at: http://cccbdb.nist.gov/. Last accessed: Feb 5, 2016.
[29] G. Herzberg, Electronic Spectra and Electronic Structure of Polyatomic Molecules; New York: D. Van Nostrand Company, **1966**.
[30] T. H. Dunning, *J. Chem. Phys.* **1989**, *90*, 1007.
[31] N. Flocke, V. Lotrich, *J. Comput. Chem.* **2008**, *29*, 2722.
[32] P. M. W. Gill, J. A. Pople, *Int. J. Quantum Chem.* **1991**, *40*, 753.
[33] Intel Corporation, Intel 64 and IA-32 Architectures Optimization Reference Manual; **2015**.
[34] H. J. Werner, F. R. Manby, P. J. Knowles, *J. Chem. Phys.* **2003**, *118*, 8149.
[35] B. I. Dunlap, J. W. D. Connolly, J. R. Sabin, *Int. J. Quantum Chem.* **1977**, *12*, 81.
[36] B. I. Dunlap, J. W. D. Connolly, J. R. Sabin, *J. Chem. Phys.* **1979**, *71*, 4993.
[37] J. L. Whitten, *J. Chem. Phys.* **1973**, *58*, 4496.

Received: 5 February 2016
Revised: 25 July 2016
Accepted: 6 August 2016
Published online on 13 September 2016