OPTIMIZING COMPUTATIONAL KERNELS IN QUANTUM CHEMISTRY

Approved by:

Dr. Sherrill, Advisor
School of Chemistry and Biochemistry, School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. McDaniel
School of Chemistry and Biochemistry
Georgia Institute of Technology

Date Approved: April 23, 2018,
This thesis is dedicated to my parents.
ACKNOWLEDGEMENTS

I would like to thank my parents, Mike and Trish, for their love and support throughout my life. Thank you so much for instilling the curiosity and dedication in me which made this possible. My sisters, brothers, nieces, nephews, and cousins deserve my appreciation as well.

I would like to sincerely thank my supervisor, Prof. Sherrill, for his guidance and support. His belief in me provided encouragement throughout this entire process. I would also like to thank Dr. Chow for co-advising me. The insights I learned from his courses made much of this work possible. Also, I would like to thank Dr. Smith, who, without his mentorship and guidance, none of this would have been possible. Collaborating with him was truly inspiring.

I greatly appreciate the support of groupmates within the Sherrill Lab. Brandon, thank you for your superb automation skills and willingness to run tests for me. Lori, thank you for letting me bug you often about how to compile Psi4. Thank you Dom, Assim, Constance, Carlos, and Yi for the support and fun times.

Last but not least, I would like to thank my girlfriend, Andrea, whose emotional and intellectual support kept me above water throughout graduate school.

This research was supported by NSF grant ACI-1609842.
## TABLE OF CONTENTS

**Acknowledgments** ................................................................. v

**List of Tables** ................................................................. viii

**List of Figures** ................................................................. x

**Chapter 1: Introduction and Background** ............................. 1

**Chapter 2: Utilizing Spatial Sparsity** ................................. 6
  2.1 Sparsity Masks ............................................................... 6
  2.2 Integral Construction ....................................................... 8
  2.3 Integral Transformation ................................................... 9
  2.4 Results ........................................................................ 10

**Chapter 3: Optimizing Integral Transformations** .................... 14
  3.1 A Note on Disk-bound Index Blocking ............................... 14
  3.2 Memory Layout for Sparsity-utilized Transformations ............ 17
  3.3 Context Dependent Workflows ......................................... 19
  3.4 Intermediate Recycling ................................................... 22
  3.5 Results ........................................................................ 22
    3.5.1 Parallel Scaling of Transformations ............................ 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.2</td>
<td>Performance Crossover Through Number of Transformations</td>
<td>26</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Superior Workflows in Practice</td>
<td>28</td>
</tr>
<tr>
<td><strong>Chapter 4: Evaluating Coulomb and Exchange Matrices</strong></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Coulomb and Exchange Evaluations</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Results</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Multi-core Results</td>
<td>38</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Manycore Results</td>
<td>42</td>
</tr>
<tr>
<td><strong>Chapter 5: Conclusions</strong></td>
<td></td>
<td>44</td>
</tr>
<tr>
<td><strong>References</strong></td>
<td></td>
<td>46</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

2.1 Characteristics of benzene stack systems. $N$ and $N_{aux}$ refer to the number of primary and auxiliary basis functions, respectively. Mask sparsity refers to the percentage of significant AO function pairs in the sparsity mask. Mask sparsity increases with additional benzenes added to the stack. 11

2.2 Speedups obtained from sparsity screening at ten benzenes from data in Figure 2.2. 12

3.1 Total memory required for sparse three-index AO integrals for adenine-thymine dimer across various basis sets. The JK type auxiliary basis sets were used. 14

3.2 Speedups obtainable via pre-transforming the 3-center integrals prior to metric contraction for common occupied-virtual transformations. $N_{AO}$ and $N_{i}$ denote the number of atomic orbitals and occupied molecular orbitals, respectively. 20

3.3 Characteristics of organoboron catalyst. $N_{AO}$ and $N_{aux}$ refer to the number of primary and auxiliary basis functions, respectively. Mask sparsity refers to the percentage of significant AO function pairs in the sparsity mask. 23

3.4 Characteristics of organoboron catalyst systems across the cc-pVDZ, cc-pVTZ, and cc-pVQZ basis sets. 27

3.5 Characteristics of systems for Store vs Direct algorithm comparisons. 29

3.6 Computational times comparing the Direct and Store algorithms for three-index integral construction and transformations. 30

3.7 Total procedure wall clock times comparing the Direct and Store algorithms. 30
4.1 Total execution times for Self-Consistent Field procedures for various systems using Algorithms 10 and 11 and Algorithms 13 and 14 for exchange and Coulomb matrix evaluations, respectively. Total wall times include wall time for the entire program to execute. J and K compute time indicates the total time spent in the Coulomb and exchange matrix evaluation kernels, respectively. System descriptions including number of AO basis functions, $N_{AO}$, number of auxiliary basis functions, $N_{aux}$, basis, and number of atoms, $N_{at}$, are included. All rows are sorted by the product $N_{aux}N_{AO}$. A speedup column is included for total wall time and is calculated as the total procedure time spent using Algorithms 11 and 14 divided by the total procedure time spent using Algorithms 10 and 13. The experiments were carried out using one node consisting of an Intel Core i7-5930K processor (6 cores at 3.50GHz) and 64GB DRAM.

4.2 Total execution times for Self-Consistent Field procedures for various systems using Algorithms 10 and 11 and Algorithms 13 and 14 for exchange and Coulomb matrix evaluations, respectively. Total wall times include wall time for the entire program to execute. J and K compute time indicates the total time spent in the Coulomb and exchange matrix evaluation kernels, respectively. System descriptions including number of AO basis functions, $N_{AO}$, number of auxiliary basis functions, $N_{aux}$, basis, and number of atoms, $N_{at}$, are included. All rows are sorted by the product $N_{aux}N_{AO}$. A speedup column is included for total wall time and is calculated as the total procedure time spent using Algorithms 11 and 14 divided by the total procedure time spent using Algorithms 10 and 13. The experiments were carried out using one node consisting of two Intel Xeon E5-2640 processors (10 cores at 2.40GHz) and 256GB DRAM.

4.3 Total procedure execution times for a Self-Consistent Field procedure using both Algorithms 10 and 13 and Algorithms 11 and 14 for exchange and Coulomb matrix evaluations, respectively. The programs were executed on one Knights Landing processor using a development version of Psi4. A relative speedup is listed.

4.4 Total exchange matrix evaluation times within a Self-Consistent Field procedure using both Algorithms 10 and 11. The programs were executed on one Knights Landing processor using a development version of Psi4. A relative speedup is listed.
LIST OF FIGURES

2.1 Schwarz sparsity masks used for benzene stacks containing 1, 5, and 10 benzenes spaced 3Å apart. Sparsity masks were obtained by evaluating $(\mu\nu|\mu\nu) < \frac{\tau^2}{(\mu\nu|\mu\nu)_{max}}$. ................................................................. 8

2.2 Comparison of execution times using sparsity screening (blue) against no sparsity screening (orange). Execution time is plotted against number of benzenes in a benzene stack from one to ten benzenes. Transformations involved computing $(ib|Q)$, where $i$ and $b$ denote occupied and virtual indices, respectively. Cutoff refers to the Schwarz screening threshold. (a) Computing the integrals. (b) Contracting AO integrals with the fitting metric. (c) First transformation times only. (d) Sum of first and second transformation times. ......................................................... 12

3.1 Transition state for organoboron addition to trifluoroacetone. Taken from Ref. [18] ................................................................. 23

3.2 Speedup and execution time plots obtained using our optimized memory layout, $B_{\mu P\nu\mu}$, for sparsity screened 3-center integrals. Execution times involve computing three common transformation classes: $(ij|Q)$, $(ib|Q)$, and $(ab|Q)$, where $i, j$ and $a, b$ denote occupied and virtual indices, respectively. Graphs (a), (c), and (e) include speedups for constructing the integrals (orange), transforming (blue), total time (red), and ideal (black). Graphs (b), (d), and (f) plot total execution times. Construction times include both integral computations and metric contractions. Problem sizes were increased by increasing basis set size using cc-pVXZ, X = D,T,Q. ........................................ 25

3.3 Comparison of total execution times for the Store and Direct algorithms to complete $(ij|Q)$, $(ib|Q)$, and $(ab|Q)$ transformations across the cc-pVDZ, cc-pVTZ, and cc-pVQZ basis sets. A scan from one to ten transformations was performed. In each case, a crossover occurs as the Direct algorithm becomes more expensive. The crossover occurs in fewer iterations for transformations involving larger MO spaces. With increasing basis set size, the crossover point is shifted to the right for the $(ij|Q)$ and $(ib|Q)$ transformations and it is shifted slightly to the left for the $(ab|Q)$ transformation. ................................. 27
3.4 Systems used for context dependent investigation of the Store and Direct workflows. (a) Hexatriene. (b) Benzene and toluene in 20 water solvent molecules.
SUMMARY

Density fitting is a rank reduction technique popularly used in quantum chemistry in order to reduce the computational cost of evaluating, transforming, and processing the 4-center electron repulsion integrals (ERIs). By utilizing the resolution of the identity technique, density fitting reduces the 4-center ERIs into a 3-center form. Doing so not only alleviates the high storage cost of the ERIs, but it also reduces the computational cost of operations involving them. Still, these operations can remain as computational bottlenecks which commonly plague quantum chemistry procedures. The goal of this thesis is to investigate various optimizations for density-fitted version of computational kernels used ubiquitously throughout quantum chemistry. First, we detail the spatial sparsity available to the 3-center integrals and the application of such sparsity to various operations, including integral computation, metric contractions, and integral transformations. Next, we investigate sparse memory layouts and their implication on the performance of the integral transformation kernel. Next, we analyze two transformation algorithms and how their performance will vary depending on the context in which they are used. Then, we propose two sparse memory layouts and the resulting performance of Coulomb and exchange evaluations. Since the memory required for these tensors grows rapidly, we frame these discussions in the context of their in-core and disk performance. We implement these methods in the Ps14 electronic structure package and reveal the optimal algorithm for the kernel should vary depending on whether a disk-based implementation must be used.
Electron repulsion integrals (ERIs) and operations involving them pose a fundamental computational problem for quantum chemistry. These 2-electron, 4-center integrals take the form:

\[
(\mu\nu|\lambda\sigma) = \int \int \mu(r_1)\nu(r_1)r_{12}^{-1}\lambda(r_2)\sigma(r_2)d\mathbf{r}_1d\mathbf{r}_2, \tag{1.1}
\]

where \(\mu, \nu, \lambda, \sigma\) are atomic orbital (AO) basis functions and \(r_1\) and \(r_2\) are electron coordinates [1]. For simplicity, we have assumed the orbitals are real functions, as is usually the case in practice. Immediately, one should note the \(O(N_{AO}^4)\) storage costs (where \(N_{AO}\) is the number of AO basis functions), which quickly limits the size of in-core investigations. Moreover, various fundamental operations involving ERIs are also costly. Here we focus on two types of operations. The first is the construction of the Coulomb and exchange matrices:

\[
J_{\mu\nu} = (\mu\nu|\lambda\sigma)D_{\lambda\sigma} \tag{1.2}
\]

\[
K_{\mu\nu} = (\mu\lambda|\nu\sigma)D_{\lambda\sigma}, \tag{1.3}
\]

where \(J_{\mu\nu}, K_{\mu\nu}, \) and \(D_{\lambda\sigma}\) are the Coulomb, exchange, and density matrices, respectively. Note that the use of the Einstein summation notation implies summations over indices. Evaluating the 4-center ERIs according to (1.2) and (1.3) requires \(O(N_{AO}^4)\) operations, which is a heavy computational burden that plagues even the most basic quantum chemistry procedures. For example, the Fock matrix for Restricted Hartree-Fock is built in the AO
basis as:

\[ F_{\mu\nu} = H_{\mu\nu} + 2J_{\mu\nu} - K_{\mu\nu} \]  \hspace{1cm} (1.4)

Hartree-Fock is the simplest quantum chemistry procedure, also known as the Self-Consistent Field approach, because it involves iteratively reconstructing the Fock matrix until convergence. Except for very large systems with large degrees of spatial sparsity, the most computationally demanding operation for this method is the evaluation of the Coulomb and exchange matrices.

The second fundamental operation considered here involves the transformation of the ERIs from the AO basis to the molecular orbital (MO) basis. This operation is written as:

\[ (pq|rs) = C_{\mu p}C_{\nu q}(\mu\nu|\lambda\sigma)C_{\lambda r}C_{\sigma s}, \]  \hspace{1cm} (1.5)

where \( p, q, r, s \) denote MO indices and each MO is expanded as a linear combination of AO’s as \( \phi_p(r_2) = \sum_{\mu} C_{\mu p}\phi_p(r_2) \). These integral transformations scale as \( O(N_{AO}^4N_p) \) (where \( N_p \) is the size of the smallest subset of MO’s required in the target integrals). The transform serves as a major bottleneck for perturbative correlation methods like second-order Möller-Plesset perturbation theory (MP2).

To overcome the drastic cost of evaluation, transforming, and processing the 4-center ERIs, various approximations and screening techniques have been developed. Density fitting [2, 3, 4, 5, 6] is one method for alleviating these costs that has been gaining popularity in electronic structure theory [7, 8, 9, 10, 11]. By utilizing the resolution of the identity technique, density fitting reduces the rank of the 4-center ERIs and approximates them.
using a 3-center form:

\[(\mu\nu|P) = \int \int \mu(r_1)\nu(r_1)r_{12}^{-1}P(r_2)d\mathbf{r}_1d\mathbf{r}_2 \quad (1.6)\]

\[[J]_{PQ} = \int \int P(r_1)r_{12}^{-1}Q(r_2)d\mathbf{r}_1d\mathbf{r}_2 \quad (1.7)\]

\[(\mu\nu|\lambda\sigma) \approx (\mu\nu|P)[J]_{PQ}^{-1}(Q|\lambda\sigma) \quad (1.8)\]

Here we have introduced an auxiliary basis \(P, Q\) and the Coulomb fitting metric \([J]_{PQ}\) [12]. In practice, permutational symmetry is almost always used so that only one set of these 3-center integrals need be built:

\[B^P_{\mu\nu} = (\mu\nu|Q)[J]_{PQ}^{-\frac{1}{2}} \quad (1.9)\]

\[(\lambda\sigma|\mu\nu) \approx B^P_{\mu\nu}B^P_{\lambda\sigma} \quad (1.10)\]

In this work, tensors labeled \(B\) indicate integrals that have been contracted with the fitting metric. The rank-reduced 3-center ERIs have a storage requirement of \(O(N_{aux}N_{AO}^2)\), which allows many more investigations to be run in-core compared to the original 4-center ERIs. Using the 3-center ERIs, the Coulomb and exchange matrices are evaluated as:

\[J_{\mu\nu} = B^P_{\mu\nu}B^P_{\lambda\sigma}D_{\lambda\sigma} \quad (1.11)\]

\[K_{\mu\nu} = B^P_{\mu\lambda}B^P_{\nu\sigma}D_{\lambda\sigma} \quad (1.12)\]

Using (1.11), the Coulomb matrix is evaluated in \(O(N_{aux}N_{AO}^2)\) operations, which is a great success for reducing computational complexity. On the other hand, (1.12) still requires \(O(N_{aux}N_{AO}^3)\) operations. Since \(N_{aux}\) is always larger than \(N_{AO}\), evaluating (1.12) is actually more expensive than evaluating (1.3). This leaves the exchange matrix evaluation as the major bottleneck for most density-fitted procedures. One method for alleviating this bottleneck involves employing knowledge of the form of the density matrix. In Hartree-
Fock theory, the density matrix is

\[ D_{\lambda \sigma} = C_{\lambda i} C_{\sigma i}, \]  

(1.13)

where \( i \) indicates an occupied MO, and there are far fewer occupied MO’s than there are AO basis functions. Since MO spaces are often much smaller than AO spaces, it is possible to lower the complexity prefactor when building the exchange matrix. Decomposing the density matrix and rewriting the Coulomb and exchange matrix evaluations, we get:

\[ J_{\mu \nu} = B_{\mu \nu}^P B_{\lambda \sigma}^P C_{\lambda i} C_{\sigma i} \]  

(1.14)

\[ K_{\mu \nu} = B_{\mu \lambda}^P B_{\nu \sigma}^P C_{\lambda i} C_{\sigma i} \]  

(1.15)

Both (1.14) and (1.15) requires \( \mathcal{O}(N_{\text{aux}} N_{\lambda \sigma}^2 N_i) \) operations. This technique is not beneficial for Coulomb matrix evaluations, since the complexity of (1.14) is actually higher than that of (1.11). However, (1.15) reduces the complexity of exchange matrix evaluation by \( \frac{N_{\lambda \sigma}}{N_i} \), where \( N_i \) denotes the number of occupied orbitals, which provides moderate speedups in practice.

Moreover, density fitting also improves the cost of integral transformations. Applying (1.10) to (1.5), we get:

\[ B_{pq}^P = B_{\mu \nu}^P C_{\mu p} C_{\nu q} \]  

(1.16)

\[ (pq|rs) \approx B_{pq}^P B_{rs}^P \]  

(1.17)

The formal computational scaling of this operation is \( \mathcal{O}(N_{\text{aux}} N_p N_q N_r N_s) \). However, in practice, algorithms based on density fitting often involve contracting the transformed three-index integrals, \( B_{pq}^P \), with other terms, rather than explicit formulation of \( (pq|rs) \) as actual intermediates. Therefore, optimizing these procedures will primarily focus on optimizing the computation of \( B_{pq}^P \) via (1.16).
Although density fitting makes considerable progress towards reducing the storage and computation complexity when using the ERIs, there is still work to be done. Importantly, (1.15) and (1.16) still serve as major computational bottlenecks for iterative and perturbative methods, respectively. In my research, I have sought out and implemented various optimizations to further improve the density fitting regime. The essence of my work involves the union of sparsity screening, good parallel scaling, minimizing disk I/O, and optimal contraction paths. In Chapter 2, I discuss the spatial sparsity of the three-index integrals and the utilization of such sparsity for integral construction, metric contractions, and integral transformations. In Chapter 3, I focus on (1.15) by discussing blocking procedures, sparse memory layouts, disk implications, and workflow optimizations. Chapter 4 focuses on (1.14) by formulating exchange matrix builds based on different sparse memory layouts and details the scalability of the resulting algorithms on multi-core and manycore architectures. Finally, Chapter 5 summarizes the results and makes suggestions for future work.
For all but the smallest molecules, spatial sparsity is very important for achieving computational savings in quantum chemistry. There exist two primary forms of spatial sparsity in the 3-center integrals $\langle \mu \nu | P \rangle$, Eq. (1.6). The first form results from the Gaussian product $\mu(r_1)\nu(r_1)$ diminishing with the overlap of the two AO Gaussians. This product diminishes rapidly with distance between the Gaussian function centers and is also highly sensitive to their degree of locality. If the overlap between two AO Gaussians is insignificant, then it becomes unnecessary to compute any shell triplet featuring this AO Gaussian pair. Utilizing this pairwise sparsity requires only an estimation of the significant AO Gaussian pairs. Once this spatial sparsity is applied and insignificant AO function pairs are screened, the complexity of computing the 3-center integrals becomes $O(N_{\text{aux}} N_{AO}^{1-2})$, where the lower bound is achieved for sufficiently sparse systems. Another type of spatial sparsity in (1.6) involves the auxiliary function, which is more complicated to utilize; see [11]. In this thesis, I focus on the former, pairwise sparsity which is featured in the product $\mu(r_1)\nu(r_1)$. Utilizing this sparsity not only speeds up the computation of the three-index integrals, but when the integrals are stored in a sparse format, then this sparsity can be utilized in later operations such as the fitting metric contraction and basis transformations.

2.1 Sparsity Masks

The pairwise sparsity available to the 3-center integrals can be determined via an estimation of significant AO function pairs. An implementation will require a two dimensional sparsity mapping structure which keeps track of the significant pairs. We refer to these structures as sparsity masks, which are $N_{AO}$ by $N_{AO}$ Boolean matrices denoted as $S_{b\mu\nu}$. The superscript $b$ reminds the reader that the structure is of Boolean type. The element $S_{b\mu\nu}^b$ will be true if
the product $\mu(r)\nu(r)$ is estimated to be significant and false otherwise.

For efficiency, it is common to compute batches of density fitted integrals in shell triplets:

$$(MN|Q) = \{(\mu\nu|P), \mu \in M, \nu \in N, P \in Q\},$$

(2.1)

where $M$ and $N$ denote shells of AO functions, and $P$ denotes a shell of auxiliary basis functions. A shell is simply a collection of all angular momentum components of a given radial function, e.g., $p_x, p_y, p_z$ for $p$ shell. An individual integral $(\mu\nu|P)$ is bounded by the Schwarz inequality $(\mu\nu|P) \leq \sqrt{(\mu\nu|\mu\nu)(P|P)}$. We can eliminate entire shells of integrals, $(MN|Q)$, by computing and storing the maximum $(\mu\nu|\mu\nu)$ that occurs for a given shell $\mu \in M, \nu \in N$. We use a relative cutoff tolerance $\tau$ and neglect shells for which

$$(MN)_{max} < \frac{\tau^2}{(\mu\nu|\mu\nu)_{max}},$$

(2.2)

where $(MN)_{max}$ is the maximum $(\mu\nu|\mu\nu)$ within a shell pair $(M, N)$ and $(\mu\nu|\mu\nu)_{max}$ is the global maximum $(\mu\nu|\mu\nu)$. In addition, even for shell pairs that are not screened out, we can apply function-level screening and avoid computation of $(\mu\nu|P)$, where

$$(\mu\nu|\mu\nu) < \frac{\tau^2}{(\mu\nu|\mu\nu)_{max}},$$

(2.3)

Using this strategy, a sparsity mask can be easily constructed. Figure 2.1 includes illustrations of sparsity masks for benzene stacks with 1, 5, and 10 benzenes spaced 3 Å apart. As sparsity increases, the number of significant AO function pairs contained in these masks becomes $O(N_{AO}^{1-2})$, where the lower bound is obtained for sufficiently sparse systems.
Figure 2.1: Schwarz sparsity masks used for benzene stacks containing 1, 5, and 10 benzenes spaced 3Å apart. Sparsity masks were obtained by evaluating $\langle \mu \nu | \mu \nu \rangle < \frac{\pi^2}{\langle \mu \nu | \mu \nu \rangle_{\text{max}}}$. 

2.2 Integral Construction

Now, we are equipped to discuss the resulting algorithms when applying the integral sparsity to important operations such as integral construction and transformation. To write more readable algorithms, we employ some tensor notation and write $A^P_{\mu \nu}$ to represent the pre-contracted 3-center integrals $\langle \mu \nu | P \rangle$. To prune these integrals using sparsity, the following algorithm can be employed:

**Algorithm 1** Prune $A^P_{\mu \nu}$ using sparsity

```plaintext
Require: AO integrals: $A^P_{\mu \nu}$, screening mask: $S^b_{\mu \nu}$
for $\mu = 0$ to $\mu = N_{\text{AO}} - 1$ do
    $A^P_{\mu \nu} S^b_{\mu \nu} \rightarrow A^P_{\mu \nu}$
end for
return $A^P_{\mu \nu}$
```

Here we have introduced the symbol $\nu^\mu$, which indicates that $\nu$ is restricted to AO functions which are spatially close enough to $\mu$ to survive the Schwarz screening process. The superscript in $\nu^\mu$ indicates a dependence of $\nu$ according to $\mu$. Algorithm 1 is purely pedagogical, as one would never build the full $A^P_{\mu \nu}$ integrals and then prune them for sparsity. Rather, sparsity screening would be applied as the integrals are constructed and insignificant function triplets are never computed. The computation and storage of $A^P_{\mu \nu}$ scales as
Moreover, the costly metric contraction becomes

\[ B_{\mu\nu}^Q = A_{\mu\nu}^P [J]_{PQ}^{-\frac{1}{2}} \]  

(2.4)

and scales as \(O(N_{aux}^2 N_{AO}^{1/2})\), where the lower bound is achieved for sufficiently sparse systems.

### 2.3 Integral Transformation

The transformation of \(A_{\mu\nu}^P\) (or \(B_{\mu\nu}^Q\)) from the AO basis to the MO basis is an essential operation for many quantum chemistry procedures. The operation requires two MO matrices: \(C_{\mu p}\), \(C_{\nu q}\), where \(p, q\) are MO space indices. These matrices will be identical if \(p\) and \(q\) run over the full space of MOs, but could be different if \(p\) and \(q\) belong to two different subsets of MOs (e.g., occupied orbitals and unoccupied orbitals). A first contraction \(A_{\mu\nu}^P = A_{\mu\nu}^P C_{\mu p}\) will half-transform the integrals and costs \(O(N_{aux} N_{AO}^2 N_p)\). The second contraction \(A_{\mu\nu}^P = A_{\mu\nu}^P C_{\nu q}\) will cost \(O(N_{aux} N_{AO} N_p N_q)\). Note the first contraction should involve the smaller of \(N_p\) and \(N_q\) in order to reduce complexity. Also, the first contraction is comparably more expensive than the latter since the size of the AO space is larger than any MO space. Therefore, reducing the cost of the first contraction would alleviate a bottleneck overall. Thankfully, we can exploit the sparsity of \(A_{\mu\nu}^P\). To do so, we carry out a looping DGEMM through the \(\mu\) index and apply the sparsity mask to the orbital matrix \(C_{\mu p}\). Algorithm 2 illustrates the resulting process:
Algorithm 2 Transform sparse integrals $A^P_{\mu\nu}$ to MO spaces.

Require: Sparse AO integrals: $A^P_{\mu\nu}$, orbital matrices: $C_{\mu p}, C_{\nu q}$, screening mask: $S^b_{\mu\nu}$

for $\mu = 0$ to $\mu = N_{AO} - 1$ do
    $C_{\nu q} S^b_{\mu\nu} \rightarrow C_{\nu q}$
    $A^P_{\mu\nu} C_{\nu q} \rightarrow A^P_{\mu q}$
end for

$A^P_{\mu q} C_{\mu p} \rightarrow A^P_{pq}$

return $A^P_{pq}$

2.4 Results

All methods were implemented in the Psi4 electronic structure software package [13]. The parallelism in Psi4 relies on the shared memory programming model using OpenMP and carries out matrix multiplications using Intel’s Math Kernel Library.

We demonstrate the performance of our Schwarz screening implementation by measuring the performance when computing the 3-center integrals, contracting the fitting metric, and transforming the integrals into an MO basis. The experiment employed an ideal sparse system: stacked benzenes. Execution times were recorded for each successive benzene added to the stack, from one to ten benzenes, with each benzene spaced 5 Å apart. Transformation time involved the wall time required to carry out the common occupied-virtual transformation, as would be required by density-fitted MP2:

$$ \langle ib|Q \rangle = (\lambda\sigma|Q)C_{\sigma i}C_{\lambda b} $$ (2.5)

Where $i, j$ and $a, b$ denote occupied and virtual spaces, respectively. Algorithm 2 was implemented and used to carry out these transformations. The cc-pVTZ and cc-pVTZ-jkfit basis sets were used for primary and auxiliary basis sets, respectively. The characteristics of each system are listed in Table 2.1. The mask sparsity listed in Table 2.1 refers to
the percentage of non-significant AO function pairs appearing in the sparsity mask. The experiment was carried out using one node consisting of an Intel Core i7-5930K processor (6 cores at 3.50GHz) and 50GB DRAM. The results are plotted in Figure 2.2. Figure 2.2 (a), (b), (c), and (d) involve time to compute the 3-index integrals, contract the fitting metric, perform the first transformation step: $(\mu\nu|Q)C_{ij\lambda} \rightarrow (iv|Q)$, and total transform time, respectively. Note that our novel contribution involves the application of the sparsity screening in the transformation step.

Table 2.1: Characteristics of benzene stack systems. $N$ and $N_{aux}$ refer to the number of primary and auxiliary basis functions, respectively. Mask sparsity refers to the percentage of significant AO function pairs in the sparsity mask. Mask sparsity increases with additional benzenes added to the stack.

<table>
<thead>
<tr>
<th>Benzenes</th>
<th>$N$</th>
<th>$N_{aux}$</th>
<th>Mask Sparsity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>264</td>
<td>654</td>
<td>2.6</td>
</tr>
<tr>
<td>2</td>
<td>528</td>
<td>1308</td>
<td>24.7</td>
</tr>
<tr>
<td>3</td>
<td>792</td>
<td>1962</td>
<td>43.6</td>
</tr>
<tr>
<td>4</td>
<td>1056</td>
<td>2616</td>
<td>55.3</td>
</tr>
<tr>
<td>5</td>
<td>1320</td>
<td>3270</td>
<td>63.1</td>
</tr>
<tr>
<td>6</td>
<td>1584</td>
<td>3924</td>
<td>68.6</td>
</tr>
<tr>
<td>7</td>
<td>1848</td>
<td>4578</td>
<td>72.7</td>
</tr>
<tr>
<td>8</td>
<td>2112</td>
<td>5232</td>
<td>75.9</td>
</tr>
<tr>
<td>9</td>
<td>2376</td>
<td>5886</td>
<td>78.4</td>
</tr>
<tr>
<td>10</td>
<td>2640</td>
<td>6540</td>
<td>80.4</td>
</tr>
</tbody>
</table>
Figure 2.2: Comparison of execution times using sparsity screening (blue) against no sparsity screening (orange). Execution time is plotted against number of benzenes in a benzene stack from one to ten benzenes. Transformations involved computing $\langle ib | Q \rangle$, where $i$ and $b$ denote occupied and virtual indices, respectively. Cutoff refers to the Schwarz screening threshold. (a) Computing the integrals. (b) Contracting AO integrals with the fitting metric. (c) First transformation times only. (d) Sum of first and second transformation times.

The cost of the first contraction becomes $O(N_{aux} N_{AO}^{1-2} N_p)$. The remaining sparsity in the half-transformed $A_{\nu q}^P$ is unrelated to the original sparsity mask.

Table 2.2: Speedups obtained from sparsity screening at ten benzenes from data in Figure 2.2.
Clearly, Figure 2.2 reveals significant time reductions for all procedures measured. Table 2.2 reinforces that operations with higher complexity scaling have the largest time reduction, with the caveat that total transformation time includes portions without sparsity utilization. In the integral computations (Figure 2.2 (a)), we construct the sparse integrals using function screening; however, we still compute them in shells for efficiency. Therefore the acquired speedup is slightly less dramatic compared to the metric contraction and transformation procedures (Figure 2.2 (b) and (c), respectively) due to select functions being screened in cases where the entire shell is not screened. Note that the metric contraction is by far the most expensive operation, which should in one part highlight the boon of sparsity utilization and in another part illustrate the pertinence of our workflow investigation in Chapter 3 of this thesis. Lastly, our proposal outlined in Algorithm 2 for applying sparsity screening to integral transformations is proven viable by Figure 2.2 (c). Note that significant reductions are obtained for the first transformation; however, the sparsity mask constructed is not helpful for the second transformation step. With sparsity utilization, the first step scales as $\mathcal{O}(N_{aux} N^1_{AO} \cdot p)$, whereas the second transformation step will still scale as $\mathcal{O}(N_{aux} N_p N_q)$. In this work we have not gone on to consider sparsity of the MO indices, which would typically require transformation to local orbitals.
CHAPTER 3
OPTIMIZING INTEGRAL TRANSFORMATIONS

3.1 A Note on Disk-bound Index Blocking

The size of tensors grows rapidly in quantum chemistry. The 3-center integrals of density fitting are no exception. Often, the size of the AO three-index integrals will exceed 64GB of RAM for small systems when a large basis set such as aug-cc-pVQZ is used, even with sparsity screening. Table 3.1 illustrates this point by listing the total memory required for sparse three-index AO integrals for the adenine-thymine dimer across various basis sets. Once the memory required exceeds what is available, it is necessary for any implementation to begin reading and writing these tensors to and from disk-based memory. For any field this can be a major slowdown, but it is especially critical to performance when high-dimensional data is involved.

Table 3.1: Total memory required for sparse three-index AO integrals for adenine-thymine dimer across various basis sets. The JK type auxiliary basis sets were used.

<table>
<thead>
<tr>
<th>Basis</th>
<th>Memory Required</th>
<th>( N_{AO} )</th>
<th>( N_{aux} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc-pVDZ</td>
<td>1.0GB</td>
<td>321</td>
<td>1583</td>
</tr>
<tr>
<td>aug-cc-pVDZ</td>
<td>4.4GB</td>
<td>536</td>
<td>1986</td>
</tr>
<tr>
<td>cc-pVTZ</td>
<td>5.4GB</td>
<td>724</td>
<td>1831</td>
</tr>
<tr>
<td>aug-cc-pVTZ</td>
<td>22.0GB</td>
<td>1127</td>
<td>2482</td>
</tr>
<tr>
<td>cc-pVQZ</td>
<td>24.3GB</td>
<td>1375</td>
<td>2575</td>
</tr>
<tr>
<td>aug-cc-pVQZ</td>
<td>91.4GB</td>
<td>2026</td>
<td>3534</td>
</tr>
<tr>
<td>cc-pV5Z</td>
<td>93.3GB</td>
<td>2334</td>
<td>3687</td>
</tr>
<tr>
<td>aug-cc-pV5Z</td>
<td>316.1GB</td>
<td>3293</td>
<td>5014</td>
</tr>
</tbody>
</table>

To illustrate this issue, we will introduce an adapted tensor notation which better in-
icates memory layout. We denote an \( n \)-dimensional tensor as \( T_{a_{b...n}} \), where the indices from left to right go from the slowest-running to the fastest-running indices. Here, \( a \) is the slowest-running index, \( b \) is the next slowest-running index, and \( n \) is the fastest-running index. The choice of memory layout plays a crucial role when indices are being accessed. Iterating through the slowest-running index, \( a \), would require the largest memory strides whereas adjacent elements of the fastest-running index, \( n \), are contiguous in memory.

Now, we can consider two possible forms for the three-center integrals: \( A_{P\mu\nu} \) or \( A_{\mu P\nu} \). If these tensors are too large to fit into memory, we must read and write pieces of them to and from disk-based memory. To accomplish this, we must choose an index to block across. For example, if we choose to block across the \( P \) index, then we will partition the basis of \( P \) into discrete blocks: \( P_i \in \{P\} \). Then, we will read and write only those blocks of \( P \) along with all of \( \mu \) and \( \nu \) for that given block. The latency of these operations is bounded by the movement of a physical read-write head, so it is critically important to ensure that read and writes are as contiguous as possible. In the case of \( P \) blocking, the \( A_{P\mu\nu} \) tensor is far superior to \( A_{\mu P\nu} \) since the former will yield entirely contiguous operations whereas the latter will require strided operations.

Unfortunately, within the density fitting regime, different operations are optimal under different blocking schemes. For example, consider the construction of the full three-index AO integrals according to (1.9). To accomplish this, the initial integrals, \( A^{P}_{\mu\nu} \), are computed and then contracted with the fitting metric. If the AOs are too large to fully fit in-core, then we must choose an index, \( P \) or \( \mu \), to block across. Consider the following two algorithms that block across either index, respectively:
Algorithm 3 Construct the full AO integrals \( B_{\mu\nu}^P \) by blocking across the \( P \) index.

**Require:** Coulomb metric: \( [J]_{PQ}^{-\frac{1}{2}} \)

Initialize: \( B_{\mu\nu}^Q = 0 \)

for block \( P_i \in \{P\} \) do

Compute: \( A_{\mu\nu}^{P_i} \)

Contract: \( A_{\mu\nu}^{P_i}[J]_{P_iQ}^{-\frac{1}{2}} \rightarrow B_{\mu\nu}^Q \)

Write: \( B_{\mu\nu}^Q += B_{\mu\nu}^Q \)

end for

return \( B_{\mu\nu}^Q \)

Algorithm 4 Construct the full AO integrals \( B_{\mu\nu}^P \) by blocking across the \( \mu \) index.

**Require:** Coulomb metric: \( [J]_{PQ}^{-\frac{1}{2}} \)

for block \( \mu_i \in \{\mu\} \) do

Compute: \( A_{\mu\nu}^{P} \)

Contract: \( A_{\mu\nu}^{P}[J]_{P_iQ}^{-\frac{1}{2}} \rightarrow B_{\mu\nu}^Q \)

Write: \( B_{\mu\nu}^Q \)

end for

return \( B_{\mu\nu}^Q \)

Of the two algorithms, only Algorithm 4 would respect the memory constraints of a blocking procedure. Note that after the contraction \( A_{\mu\nu}^{P_i}[J]_{P_iQ}^{-\frac{1}{2}} \rightarrow B_{\mu\nu}^Q \) in Algorithm 3, the full 3-dimensional quantity \( B_{\mu\nu}^Q \) is returned, which would immediately violate memory constraints. For this operation, only one blocking method is possible. If we are constrained to blocking across the \( \mu \) index, then the tensor form \( A_{\mu\nu}^{P} \) will yield superior disk performance, as it would allow for completely contiguous read operations. The purpose of this illustration is to remind the reader that for large enough systems, disk performance is crucially important, therefore memory layout should be considered carefully.
3.2 Memory Layout for Sparsity-utilized Transformations

Algorithm 2 revealed a technique that can be used to utilize sparsity when carrying out three-index integral transformations. Although utilizing sparsity is imperative for cost reduction, an optimal implementation must be tailored to fully exploit modern computing hardware. Multi-core processors consisting of upwards of ten cores are found commonly both at the desk of computational chemists and in commodity computing clusters. Moreover, the birth of Intel’s manycore architecture and the advent of Graphics Processing Units (GPUs) further necessitate that scientific computing exploits every means of parallelism. Although the density fitting technique has been shown to be challenging to exploit in massively parallel algorithms [14], it remains an essential technique for accelerating computations of small to intermediate size chemical systems. The communication overhead that hinders large scale parallelism for density fitting is nominal for single node investigations, but even a single multi-core processor contains viable parallelism that can be challenging to fully exploit. Coupling the cost reduction of sparsity approximations with a finely tuned parallel code is crucial to performance.

Maximizing parallelism in Algorithm 2 will require careful implementation design. The choice of memory layout for the sparse 3-dimensional integral tensors will affect both algorithmic complexity and parallel scalability. For the three-center integrals, we use $A_{P\mu\nu}$ to denote a memory layout with the auxiliary index $P$ as the slowest-running index. The $A_{P\mu\nu}$ layout is intuitive, as it allows looping through $P$ and direct application of the sparsity mask for each submatrix. The resulting sparse form $A_{P\mu\nu}$ contains submatrices of identical structure. However, another form, $A_{\mu P\nu}$, must be considered. The sparse form $A_{\mu P\nu}$ results in submatrices of differing sizes. However, some advantages may be ascertained. We sought to determine which of these two sparse memory layouts, $A_{P\mu\nu}$ or $A_{\mu P\nu}$, is optimal for transforming the integrals into an MO basis. Algorithms 5 and 6 illustrate the difference:
Algorithm 5 Transforming sparse integrals using $A_{P\mu\nu}$ form.

Require: Sparse AO integrals: $A_{P\mu\nu}$, orbital matrices: $C_{\mu p}, C_{\nu q}$, screening mask: $S^{b}_{\mu\nu}$

for $P = 0$ to $P = N_{aux} - 1$ do
    for $\mu = 0$ to $\mu = N_{AO} - 1$ do
        Trim from dense to sparse: $C_{\nu q}S^{b}_{\mu\nu} \rightarrow C_{\nu q}^{\mu}$
        $A_{P\mu\nu}C_{\nu q} \rightarrow A_{P\mu p}$
    end for
end for

return $A_{P\mu p}$

Final transform: $A_{P\nu q}C_{\mu p} \rightarrow A_{Ppq}$

return $A_{Ppq}$

Algorithm 6 Transforming sparse integrals using $A_{\mu P\nu}$ form.

Require: Sparse AO integrals: $A_{\mu P\nu}$, orbital matrices: $C_{\mu p}, C_{\nu q}$, screening mask: $S^{b}_{\mu\nu}$

for $\mu = 0$ to $\mu = N_{AO} - 1$ do
    Trim from dense to sparse: $C_{\nu q}S^{b}_{\mu\nu} \rightarrow C_{\nu q}^{\mu}$
    $A_{\mu P\nu}C_{\nu q} \rightarrow A_{\mu Pq}$
end for

return $A_{\mu Pq}$

Final transform: $A_{\mu Pq}C_{\mu p} \rightarrow A_{Ppq}$

return $A_{Ppq}$

To carry out the first step of the transformation, both algorithms must loop through the slowest-running index of the integrals. Operations within this loop should be parallelized. The number of iterations for this step are greater in Algorithm 5 than in Algorithm 6 since $N_{aux}$ will always be larger than $N_{AO}$. Conversely, the matrix-matrix multiplications occurring in Algorithm 6 are larger. As a result, Algorithm 6 will benefit from delegating larger problem sizes to highly optimized level 3 BLAS routines.
However, the crucial difference is ascertained when one considers which index, \( \mu \) or \( P \), would be most appropriate to block across for a disk-bound implementation. If we choose to block across \( \mu \), the result of the final transformation, \( A_{pq}^P \), would be incomplete, and a full cumulative disk write of \( \mathcal{O}(N_{aux} N_p N_p) \) size would be necessary for each block of \( \mu \). Moreover, there is a chance this operation could be altogether impossible if memory constraints would be violated by having a full \( A_{pq}^P \) tensor in memory. On the other hand, blocking across the \( P \) index would not pose this problem, as no contractions occur across the \( P \) index. Therefore, blocking across \( P \) is the only scalable solution.

Now, if it is necessary to block across \( P \), consider the implications for Algorithms 5 and 6. For Algorithm 5, this means that if the blocks over \( P \) are very small, e.g. \(< 10\), then the parallelized loops will have very few iterations. Typically, the fewer the iterations the worse the parallel scalability as the workloads are much more susceptible to being unbalanced. To the contrary, Algorithm 6 will not suffer this drawback and is therefore the better option. Since Algorithm 6 contains higher concurrency and utilizes larger matrix-matrix multiplications, we propose that it will yield enhanced parallel scaling.

### 3.3 Context Dependent Workflows

Equation (1.9) demonstrates the necessity of the fitting metric when using the 3-center density-fitted integrals. Unfortunately, the metric contraction in equation (1.9) comes at the heavy price of \( \mathcal{O}(N_{aux}^2 N_{AO}^2) \) operations. Considering that \( N_{aux} \) can be 2-3x larger than \( N_{AO} \), this can be an extremely costly operation. One remedy for cost reduction is to transform the 3-center integrals prior to contracting them with the metric:

\[
A_{pq}^Q = A_{\mu \nu}^Q C_{\mu p} C_{\nu q} \quad (3.1)
\]

\[
B_{pq}^P = [J]^\frac{1}{2} P_{pq} A_{pq}^Q \quad (3.2)
\]

Since \( N_p \) and \( N_q \) are often much smaller than \( N_{AO} \), the speedup of \( \mathcal{O}(\frac{N_{AO}}{N_p N_q}) \) can be sub-
stantial. Table 3.2 illustrates the potential benefit for some commonly used transformed integrals using occupied and virtual spaces. Orbital indices $i,j$ denote occupied spaces and $a,b$ denote virtual spaces.

Table 3.2: Speedups obtainable via pre-transforming the 3-center integrals prior to metric contraction for common occupied-virtual transformations. $N_{AO}$ and $N_i$ denote the number of atomic orbitals and occupied molecular orbitals, respectively.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>$\frac{N_{AO}}{N_i} = 2$</th>
<th>$\frac{N_{AO}}{N_i} = 5$</th>
<th>$\frac{N_{AO}}{N_i} = 10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(Q</td>
<td>ij)$</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>$(Q</td>
<td>ia)$</td>
<td>4</td>
<td>6.25</td>
</tr>
<tr>
<td>$(Q</td>
<td>ab)$</td>
<td>4</td>
<td>1.56</td>
</tr>
</tbody>
</table>

The speedups in Table 3.2 are undoubtedly beneficial and this technique is commonly used in practice. However, we propose that this technique will be a disadvantageous in certain contexts. Namely, applying this workflow to methods using many transformations will increase cost unnecessarily. Using this method, a metric contraction is necessary for each transformation, whereas only one contraction is required if this technique is not used. If many transformations occur, the cost of contracting the metric for each transformation will eventually outweigh the speedups attainable in Table 3.2. Therefore, both workflows must be considered when carrying out 3-center integral transformations. Algorithm 7 and 8 illustrate the corresponding workflows:
Algorithm 7 The "Store" algorithm - contract metric then transform.

Require: AO integrals: $A^P_{\mu\nu}$, fitting metric: $[J]^{-\frac{1}{2}}_{PQ}$, orbital matrices: $C_{\mu p}, C_{\nu q}$

Compute: $A^Q_{\mu\nu}$

Contract metric: $A^P_{\mu\nu}[J]^{-\frac{1}{2}}_{PQ} \rightarrow B^Q_{\mu\nu}$

Save: $B^Q_{\mu\nu}$

for all transformation spaces: $C_{\mu p}, C_{\nu q}$ do

Transform: $B^Q_{\mu\nu}C_{\mu p}C_{\nu q} \rightarrow B^Q_{pq}$

end for

return $B^Q_{pq}$

Algorithm 8 The "Direct" algorithm - transform then contract metric.

Require: AO integrals: $A^P_{\mu\nu}$, fitting metric: $[J]^{-\frac{1}{2}}_{PQ}$, orbital matrices: $C_{\mu p}, C_{\nu q}$

Compute: $A^Q_{\mu\nu}$

for all transformation spaces: $C_{\mu p}, C_{\nu q}$ do

Transform: $A^P_{\mu\nu}C_{\mu p}C_{\nu q} \rightarrow A^P_{pq}$

Contract metric: $A^P_{pq}[J]^{-\frac{1}{2}}_{PQ} \rightarrow B^Q_{pq}$

end for

return $B^Q_{pq}$

Hereafter we refer to the Store algorithm being the workflow that contracts the metric and then transforms the integrals. Conversely, we refer to the Direct algorithm as the workflow that transforms the integrals then contracts the metric.

Depending on the context, either the Store or Direct algorithm may be superior. We propose the Store algorithm will be superior for procedures requiring many transformations. This includes contexts which iteratively recompute transformations, such as density-fitted multiconfigurational self-consistent field (DFMCSCF) [15], as well as contexts involving a large number of transformations spaces, such as density-fitted symmetry-adapted perturbation theory (SAPT) [16, 17]. Conversely, the Direct algorithm will be superior in contexts
requiring a small number of transformations. Most notably, this includes density-fitted second-order Møller-Plesset perturbation theory (DFMP2) [10].

3.4 Intermediate Recycling

Quantum chemistry procedures can require numerous integral transformations. For example, an unrestricted Hartree-Fock (UHF) based SAPT procedure will employ 24 unique integral transformations. Since all transformations are carried out at the same stage of the computation, one should build an implementation that queues the transformations, gathers information, and deploys strategic contraction paths. As mentioned previously, the first contraction of these integral transformations \( A_{p \nu}^P = A_{\mu \nu}^P C_{\mu p} \) should always be carried out on the smallest MO index \( p \) possible. Thereafter, transformations using the same intermediate, \( A_{p \nu}^P \), should all occur at the same time to avoid recomputing \( A_{p \nu}^P \). For example, suppose two sets of transformed integrals are required, \( A_{u \nu}^P \) and \( A_{u p}^P \), where \( u, v << N \) are indices restricted to some small "active" subspace of MOs and \( p \) is a general MO index where \( N_p \approx N_{AO} \). For both transformations, the first step will be:

\[
A_{\mu \nu}^P C_{\mu u} \rightarrow A_{u \nu}^P
\]  

(3.3)

If this is recognized beforehand, then this operation need only be carried out once and the intermediate \( A_{u \nu}^P \) can be recycled for both transformations. The speedup of doing so is \( O(2N_{AO}+N_v+N_p) \), which as \( N_p \rightarrow N_{AO} \) approaches 50%. Although the benefit is modest, it must be considered for optimized procedures.

3.5 Results

All methods were implemented in the PSI4 electronic structure software package [13]. The parallelism in PSI4 relies on the shared memory programming model using OpenMP and carries out matrix multiplications using Intel’s Math Kernel Library.
3.5.1 Parallel Scaling of Transformations

To measure parallel scaling, we performed integral transformations for the boron catalyst system shown in Figure 3.1. We varied the problem size by adjusting the $\zeta$ level for the Dunning correlation-consistent basis sets with $\zeta = D, T, Q$. The characteristics of these systems are included in Table 3.3. Note that while parallel scaling typically improves for larger systems (i.e. due to larger workloads), this is not guaranteed in a sparsity regime. Larger systems may contain more sparsity; more sparsity will result in more striding, copying, and irregular sizing, which will hinder parallel scaling. Nonetheless, our method outlined in Algorithm 2 is designed to utilize sparsity while also obtaining maximum parallel efficiency.

![Figure 3.1: Transition state for organoboron addition to trifluoroacetone. Taken from Ref. [18]](image)

Table 3.3: Characteristics of organoboron catalyst. $N_{AO}$ and $N_{aux}$ refer to the number of primary and auxiliary basis functions, respectively. Mask sparsity refers to the percentage of significant AO function pairs in the sparsity mask.

<table>
<thead>
<tr>
<th>Basis</th>
<th>$N_{AO}$</th>
<th>$N_{aux}$</th>
<th>Mask Sparsity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc-pVDZ</td>
<td>671</td>
<td>3277</td>
<td>29.6</td>
</tr>
<tr>
<td>cc-pVTZ</td>
<td>1566</td>
<td>3856</td>
<td>41.1</td>
</tr>
<tr>
<td>cc-pVQZ</td>
<td>3040</td>
<td>5593</td>
<td>50.2</td>
</tr>
</tbody>
</table>
For each system, we performed the three common transformations:

\[
(ij|Q) = (\lambda \sigma |Q) C_{\sigma i} C_{\lambda j}, \tag{3.4}
\]

\[
(ib|Q) = (\lambda \sigma |Q) C_{\sigma i} C_{\lambda b}, \tag{3.5}
\]

\[
(ab|Q) = (\lambda \sigma |Q) C_{\sigma a} C_{\lambda b}. \tag{3.6}
\]

where \(i\) and \(j\) refer to occupied orbitals, and \(a\) and \(b\) refer to unoccupied orbitals. The experiment was carried out using one node consisting of an Intel Xeon E5-2630 processor (10 cores at 2.20GHz) and using 24GB DRAM. Figure 3.2 includes plots of both speedups and execution times for each system.
Figure 3.2: Speedup and execution time plots obtained using our optimized memory layout, $B\mu P\nu\mu^*$, for sparsity screened 3-center integrals. Execution times involve computing three common transformation classes: $(ij|Q)$, $(ib|Q)$, and $(ab|Q)$, where $i, j$ and $a, b$ denote occupied and virtual indices, respectively. Graphs (a), (c), and (e) include speedups for constructing the integrals (orange), transforming (blue), total time (red), and ideal (black). Graphs (b), (d), and (f) plot total execution times. Construction times include both integral computations and metric contractions. Problem sizes were increased by increasing basis set size using cc-pVXZ, X = D,T,Q.

At ten cores, speedups for total computation time were recorded as 9.09, 9.56, and
8.93 for $\zeta = D, T, Q$, respectively. The improvement in scaling between $\zeta = D$ to $\zeta = T$ may be attributed to larger system sizes. The sizes of the sparsity screened AO integrals were 10.38GB and 55.70GB for these systems, respectively. In the latter case, the 24GB of memory at the compute node was fully used and work for each thread was increased to maximal levels. Conversely, when the system size was increased again using $\zeta = Q$, the memory constraint did not allow for further increase in work per thread. The hindrance in scaling from $\zeta = T$ to $\zeta = Q$ is explained by the workload imbalance incurred by the increase in sparsity.

3.5.2 Performance Crossover Through Number of Transformations

In this section, we reveal the contexts in which either the Store or Direct algorithms are superior. First, we analyzed performance when applying either algorithm to carry out each of the three common integral transformations: $(Q|ij)$, $(Q|ib)$, and $(Q|ab)$. Doing so reveals the crossover in computational complexity that occurs between the two algorithms. For few transformations, the Direct algorithm will be superior as it benefits from the speedups given in Table 3.2. However, if many transformations occur, we propose the Store algorithm will become superior as it avoids the costly metric contraction for each transformation.

We applied both algorithms to each transformation using the same boron catalyst system in Figure 3.1. To reveal the crossover in computational work between the two algorithms, the execution times to carry out one to ten transformations were recorded. To reveal additional trends, we varied the system size by adjusting the basis set size using $\zeta = D, T, Q$. The characteristics of these systems are described in Table 3.3. The experiments were carried out using one node consisting of an Intel Core i7-5930K processor (6 cores at 3.50GHz) and 50GB DRAM. The results are plotted in Figure 3.3.
Table 3.4: Characteristics of organoboron catalyst systems across the cc-pVDZ, cc-pVTZ, and cc-pVQZ basis sets.

<table>
<thead>
<tr>
<th>Basis</th>
<th>(N_{AO})</th>
<th>(N_{aux})</th>
<th>(N_{occ})</th>
<th>(N_{virt})</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc-pVDZ</td>
<td>671</td>
<td>3277</td>
<td>129</td>
<td>542</td>
</tr>
<tr>
<td>cc-pVTZ</td>
<td>1566</td>
<td>3856</td>
<td>129</td>
<td>1437</td>
</tr>
<tr>
<td>cc-pVQZ</td>
<td>3040</td>
<td>5593</td>
<td>129</td>
<td>2911</td>
</tr>
</tbody>
</table>

Figure 3.3: Comparison of total execution times for the Store and Direct algorithms to complete \((ij|Q)\), \((ib|Q)\), and \((ab|Q)\) transformations across the cc-pVDZ, cc-pVTZ, and cc-pVQZ basis sets. A scan from one to ten transformations was performed. In each case, a crossover occurs as the Direct algorithm becomes more expensive. The crossover occurs in fewer iterations for transformations involving larger MO spaces. With increasing basis set size, the crossover point is shifted to the right for the \((ij|Q)\) and \((ib|Q)\) transformations and it is shifted slightly to the left for the \((ab|Q)\) transformation.

If only one transformation occurs, then the speedups of Table 3.2 enable the Direct algorithm to be superior. However, the Direct algorithm must carry out expensive metric
contractions for every transformation. As the number of transformations increase, the expense of these contractions overtakes the speedups of pre-transforming the integrals. Note that a crossover between the two algorithms occurs in each system. This finding supports our conjecture that the Store algorithm is advantageous in contexts where many transformations occur. This includes iterative methods where the transformations are carried out in each iteration, such as MCSCF, as well as methods which require many transformations, such as SAPT. Conversely, the Direct algorithm is advantageous for methods requiring few transformations, such as DFMP2.

Additionally, the crossover point shifts to the left as the transformation spaces get larger \((ij, ia, ab)\), occurring in fewer transformations. This finding is supportive of the proposed speedups in Table 3.2. Therefore procedures using anything larger than a \((ib|Q)\) transformation will receive nominal benefits from employing the Direct algorithm and may incur slowdowns if many transformations occur. Last, the crossover point shifts to the right for the \((ij|Q)\) and \((ib|Q)\) transformations as larger basis sets are used. This will allow for continued benefits with more transformations. Conversely, the crossover shifts to the left for the \((ab|Q)\) transformation for larger basis sets. Either of these findings are elucidated by the increasing ratio of \(\frac{N_{aux}}{N_{AO}}\).

### 3.5.3 Superior Workflows in Practice

In the previous section, we determined the Direct algorithm will be superior in methods such as DFMP2, while the Store algorithm will be superior in methods such as MCSCF. After determining the contexts in which either algorithm prevail, we sought to reveal their benefits when applied in practice. To do so, we employed either algorithm in the contexts of different procedures and systems. For procedures, we tested DFMCSCF and DFMP2. We ran these procedures on the systems included in Figure 3.4. Figure 3.4 (a) is a hexatriene molecule. Figure 3.4 (b) is benzene and toluene solvated by 20 water molecules. Table 3.4 lists the characteristics of each system, which includes the basis set, the number of primary
and auxiliary basis functions, and the mask sparsity.

Figure 3.4: Systems used for context dependent investigation of the Store and Direct work-flows. (a) Hexatriene. (b) Benzene and toluene in 20 water solvent molecules.

Table 3.5: Characteristics of systems for Store vs Direct algorithm comparisons.

<table>
<thead>
<tr>
<th>System</th>
<th>Primary Basis</th>
<th>Auxiliary Basis</th>
<th>$N_{aux}$</th>
<th>$N_{virt}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexatriene</td>
<td>cc-pVQZ</td>
<td>cc-pVQZ-jkfit</td>
<td>570</td>
<td>1044</td>
</tr>
<tr>
<td>Hexatriene</td>
<td>cc-pVQZ</td>
<td>cc-pVQZ-rifit</td>
<td>570</td>
<td>1232</td>
</tr>
<tr>
<td>Benzene-Toluene</td>
<td>jun-cc-pVDZ</td>
<td>jun-cc-pVDZ-jkfit</td>
<td>867</td>
<td>3849</td>
</tr>
<tr>
<td>Benzene-Toluene</td>
<td>jun-cc-pVDZ</td>
<td>jun-cc-pVDZ-rifit</td>
<td>867</td>
<td>2901</td>
</tr>
</tbody>
</table>

The experiments were carried out using one node consisting of an Intel Core i7-5930K processor (6 cores at 3.50GHz) and 60GB DRAM. The results are included in Tables 3.6 and 3.7. Table 3.6 includes the total computation time spent in operations involving the three-index integrals. These times will reflect the algorithmic benefits illustrated in Figure 3.3. Table 3.7 includes the total time required to execute the program. These times reflect total procedure times, which include many operations extraneous to the three-index integrals.
Table 3.6: Computational times comparing the Direct and Store algorithms for three-index integral construction and transformations.

<table>
<thead>
<tr>
<th>System</th>
<th>Procedure</th>
<th>DIRECT</th>
<th>STORE</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexatriene</td>
<td>DFMCSCF</td>
<td>42.3s</td>
<td>7.8s</td>
<td>5.4x</td>
</tr>
<tr>
<td>Hexatriene</td>
<td>DFMP2</td>
<td>2.9s</td>
<td>6.0s</td>
<td>2.1x</td>
</tr>
<tr>
<td>Benzene-Toluene</td>
<td>DFMCSCF</td>
<td>438.1s</td>
<td>104.8s</td>
<td>4.2x</td>
</tr>
<tr>
<td>Benzene-Toluene</td>
<td>DFMP2</td>
<td>31.4s</td>
<td>61.2s</td>
<td>1.9x</td>
</tr>
</tbody>
</table>

Table 3.7: Total procedure wall clock times comparing the Direct and Store algorithms.

<table>
<thead>
<tr>
<th>System</th>
<th>Procedure</th>
<th>DIRECT</th>
<th>STORE</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexatriene</td>
<td>DFMCSCF</td>
<td>173.0s</td>
<td>145.6s</td>
<td>1.2x</td>
</tr>
<tr>
<td>Hexatriene</td>
<td>DFMP2</td>
<td>35.1s</td>
<td>40.5s</td>
<td>1.2x</td>
</tr>
<tr>
<td>Benzene-Toluene</td>
<td>DFMCSCF</td>
<td>5138.7s</td>
<td>4811.8s</td>
<td>1.1x</td>
</tr>
<tr>
<td>Benzene-Toluene</td>
<td>DFMP2</td>
<td>852.7s</td>
<td>856.0s</td>
<td>1.0x</td>
</tr>
</tbody>
</table>

Both Tables 3.6 and 3.7 reveal that the Direct algorithm is superior for DFMP2 whereas the Store algorithm is superior for DFMCSCF. The computational speedups can be substantial, reaching 5.4x for DFMCSCF with the hexatriene system. However, Table 3.7 reveals these speedups are considerably dampened for the overall procedure time. This is true because the operations involving the three-index integrals are not the most expensive computations occurring within these procedures.
4.1 Coulomb and Exchange Evaluations

From Chapter 1, we wrote the most efficient evaluations of the three-index ERIs to build the Coulomb and exchange matrices as:

\[ J_{\mu\nu} = B_{\mu\rho}^P B_{\lambda\sigma}^P D_{\lambda\sigma} \]  
\[ K_{\mu\nu} = B_{\mu\lambda}^P B_{\nu\sigma}^P C_{\lambda\rho} C_{\sigma\rho}, \]

where (4.1) and (4.2) are evaluated in \( O(N_A^2 N_{aux}) \) and \( O(N_A^2 N_{aux} N_p) \) operations, respectively. In self-consistent-field methods like Hartree-Fock and density functional theory, the density matrix is constructed by summing over only occupied molecular orbitals (\( N_p = N_{occ} \)). The exchange matrix evaluation is a major bottleneck within the density fitting regime. The focus of this chapter is on analyzing and improving exchange builds to take advantage of sparsity, optimize parallel scaling, and to minimize disk operations.

Thankfully, it turns out that Algorithm 6 is easily extendable to (4.2). In fact, if one uses orbital matrices to build the exchange matrix as in (4.2), the half-transformed intermediate generated during integral transformations, \( B_{\mu\rho Pq} \), is actually identical to the intermediate when building the exchange matrix. Moreover, the memory layout of \( B_{\mu\rho Pq} \) is ideal for the subsequent contraction to form \( K \). We will consider the possibility that we want to build "generalized" matrices, where perhaps two different matrices \( C \) are used (this is not relevant for self-consistent-field energies, but can arise in other circumstances, for example, in SAPT). A generalized Coulomb matrix can be constructed as in (4.1), but where the density matrix is not an SCF density matrix, but a generalized density matrix:

31
\[ D_{\lambda \sigma} = C_{\lambda p} C'_{\sigma p} \]  

(4.3)

and likewise a generalized exchange matrix can be written

\[ K_{\mu \nu} = B_{\mu \lambda} B_{\nu \sigma} C_{\lambda p} C'_{\sigma p}. \]  

(4.4)

Again, the contraction index \( p \) does not necessarily run over all MO’s, but is typically restricted to some subset, like occupied orbitals. The following exchange evaluation algorithm results:

**Algorithm 9** Building the \( K \) matrix.

**Require:** Sparse AO integrals: \( B_{\mu \nu \mu} \), orbital matrices: \( C_{\mu p}, C'_{\nu p} \), screening mask: \( S_{\mu \nu}^b \)

**for** \( \mu = 0 \) to \( \mu = N_{AO} - 1 \) **do**

Trim from dense to sparse: \( C'_{\nu p} S_{\mu \nu}^b \rightarrow C'_{\nu p} \)

\[ B_{\mu \nu \mu} C_{\nu p} \rightarrow T_{\mu P p} \]

**if** \( C'_{\nu p} \neq C_{\nu p} \) **then**

Trim from dense to sparse: \( C'_{\nu p} S_{\mu \nu}^b \rightarrow C'_{\nu p} \)

\[ B_{\mu \nu \mu} C'_{\nu p} \rightarrow T'_{\mu P p} \]

**else**

\[ T'_{\mu P p} = T_{\mu P p} \]

**end if**

**end for**

\[ T_{\mu P p} T'_{\nu P p} \rightarrow K_{\mu \nu} \]

**return** \( K_{\mu \nu} \)

We have used \( T \) tensors to indicate intermediates. Algorithm 9 is an ideal candidate for
building the $K$ matrix, as it both utilizes sparsity and maximizes concurrency. However, while it may be assumed that both $J$ and $K$ can always fit within in-core memory constraints, the same is not true for the three-index AO integrals. Therefore, we must consider the behavior of Algorithm 9 when the in-core memory is constrained such that it becomes necessary to perform blocking operations across the $P$ index. Then, the algorithm becomes:

**Algorithm 10 Building the $K$ matrix using $B_{\mu P\nu\mu}$, blocking across $P$**

**Require:** Sparse AO integrals: $B_{\mu P\nu\mu}$, orbital matrices: $C_{\mu p}, C'_{\nu p}$, screening mask: $S^b_{\mu\nu}$

```plaintext
for block $P_i \in \{P\}$ do
    Read from disk: $B_{\mu P_i\nu\mu}$
    for $\mu = 0$ to $\mu = N_{AO} - 1$ do
        Trim from dense to sparse: $C_{\nu p}S^b_{\mu\nu} \rightarrow C_{\nu p}$
        $B_{\mu P_i\nu\mu}C_{\nu p} \rightarrow T_{\mu P_p}$
        if $C'_{\nu p} \neq C_{\nu p}$ then
            Trim from dense to sparse: $C'_{\nu p}S^b_{\mu\nu} \rightarrow C'_{\nu p}$
            $B_{\mu P_i\nu\mu}C'_{\nu p} \rightarrow T'_{\mu P_p}$
        else
            $T'_{\mu P_p} = T_{\mu P_p}$
        end if
    end for
    $K_{\mu\nu} = K_{\mu\nu} + T_{\mu P_p}T'_{\nu P_p}$
end for
return $K_{\mu\nu}$
```

Unfortunately, this algorithm will require strided disk operations when reading the $B_{\mu P_i\nu\mu}$ tensor into memory. Since poor disk I/O can drastically decrease performance, it is often better to force disk operations to be contiguous and transpose any tensors as necessary in core memory. Since it is best to block across the $P$ index (as discussed in Chapter 3), a
better memory layout for the sparse integrals is the $B_{P^\mu\nu}$ form, since this form will allow for completely contiguous reads from disk memory. Another possible algorithm can then be formulated:

**Algorithm 11** Building the $K$ matrix using $B_{P^\mu\nu}$, blocking across $P$

**Require:** Sparse AO integrals: $B_{P^\mu\nu}$, orbital matrices: $C_{\mu p}, C'_{\nu p}$, screening mask: $S_{\mu \nu}^b$

for block $P_i \in \{P\}$ do

Read from disk: $B_{P^\mu\nu}$

for $\mu = 0$ to $\mu = N_{AO} - 1$ do

Copy: $B_{P^\mu\nu} \rightarrow b_{P^\mu\nu}$

Trim from dense to sparse: $C_{\nu p} S_{\mu \nu}^b \rightarrow C_{\nu p}$

$b_{P^\mu\nu} C_{\nu p} \rightarrow T_{\mu P_i p}$

if $C'_{\nu p} \neq C_{\nu p}$ then

Trim from dense to sparse: $C'_{\nu p} S_{\mu \nu}^b \rightarrow C'_{\nu p}$

$b_{P^\mu\nu} C_{\nu p} \rightarrow T'_{\mu P_i p}$

else

$T'_{\mu P_i p} = T_{\mu P_i p}$

end if

end for

$K_{\mu \nu} = K + T_{\mu P_i p} T'_{\nu P_i p}$

end for

return $K_{\mu \nu}$

Here, we used a buffer, $b_{P^\mu\nu}$, to transpose pieces of $B_{P^\mu\nu}$ while looping over $\mu$. Although this algorithm may involve a strided copy and additional memory usage, the disk reads for the $B_{P^\mu\nu}$ tensor are completely contiguous. For smaller investigations, the three-index AO integrals can be fit completely in-core and Algorithm 10 should yield superior performance. However, for larger systems, the strided disk reads in Algorithm 10 may cause performance degradation to the point that Algorithm 11 will become superior.
Although the exchange matrix evaluation is the primary computational bottleneck, it is important to note how the above integral formats will affect Coulomb matrix evaluations. To build the Coulomb matrix, the sparsity mask can be applied directly to the density matrix. The following algorithm results:

**Algorithm 12** Building the $J$ matrix.

**Require:** Sparse AO integrals: $B^{P}_{\mu\nu\mu}$, density matrix: $D_{\mu\nu}$, screening mask: $S^b_{\mu\nu}$

Trim from dense to sparse: $D_{\mu\nu}S^b_{\mu\nu} \rightarrow D^{P}_{\mu\nu}$

$B^{P}_{\mu\nu\mu}D^{P}_{\mu\nu\mu} \rightarrow T^P$

$T^P B^{P}_{\mu\nu\mu} \rightarrow J^{P}_{\mu\nu\mu}$

Unpack from sparse to dense: $J^{P}_{\mu\nu\mu} \rightarrow J_{\mu\nu}$

**return** $J_{\mu\nu}$

Moreover, the corresponding disk-based implementations for the $B_{\mu\mu\mu}$ and $B_{P\mu\mu}$ integral formats are listed in Algorithms 13 and 14, respectively.
Algorithm 13 Building the $J$ matrix using $B_{\mu P\nu}$, blocking across $P$

**Require:** Sparse AO integrals: $B_{\mu P\nu}$, density matrix: $D_{\mu\nu}$, screening mask: $S_{\mu\nu}^b$

for block $P_i \in P$ do
  Read from disk: $B_{\mu P_i\nu}$
  Initialize: $T_{P_i} = 0$
  for $\mu = 0$ to $\mu = N_{AO} - 1$ do
    Copy to sparse: $D_{\mu\nu}S_{\mu\nu}^b \rightarrow d_{\nu\mu}$
    $T_{P_i} = T_{P_i} + B_{\mu P_i\nu}d_{\nu\mu}$
  end for
  $T_{P_i} B_{P_i\mu\nu} \rightarrow J_{\mu\nu}$
  Unpack from sparse to dense: $J_{\mu\nu} \rightarrow J_{\mu\nu}^{(P_i)}$
  $J_{\mu\nu} = J_{\mu\nu} + J_{\mu\nu}^{(P_i)}$
end for
return $J_{\mu\nu}$

Algorithm 14 Building the $J$ matrix using $B_{P\mu\nu}$, blocking across $P$

**Require:** Sparse AO integrals: $B_{P\mu\nu}$, density matrix: $D_{\mu\nu}$, screening mask: $S_{\mu\nu}^b$

for block $P_i \in P$ do
  Read from disk: $B_{P_{P_i}\mu\nu}$
  Trim from dense to sparse: $D_{\mu\nu}S_{\mu\nu}^b \rightarrow D_{\mu\nu}$
  $B_{P_i\mu\nu}D_{\mu\nu} \rightarrow T_{P_i}$
  $T_{P_i} B_{P\mu\nu} \rightarrow J_{\mu\nu}$
  Unpack from sparse to dense: $J_{\mu\nu} \rightarrow J_{\mu\nu}^{(P_i)}$
  $J_{\mu\nu} = J_{\mu\nu} + J_{\mu\nu}^{(P_i)}$
end for
return $J_{\mu\nu}$

Here, we have used the superscript in $J_{\mu\nu}^{(P_i)}$ to indicate the contribution to $J_{\mu\nu}$ correspond-
ing to the $P_i$ block. Note that Algorithm 13 is likely inferior to Algorithm 14, with both necessary loops and copies as well as non-contiguous disk reads. However, since Coulomb matrix evaluation requires considerably less compute time than the exchange matrix evaluations, the time required by Algorithms 13 and 14 will be nearly trivial compared to the exchange form. Moreover, note that the disk reads in Algorithms 10 and 13 as well as in Algorithms 11 and 14 will be occurring simultaneously. The $J$ and $K$ computational kernel will generally take this form:

**Algorithm 15** Coulomb and exchange matrix evaluation kernel.

**Require:** Sparse AO integrals: $B_{\mu P \nu}^P$, density matrix: $D_{\mu \nu}$, orbital matrices: $C_{\mu p}, C'_{\nu p}$, screening mask: $S_{\mu \nu}^b$

Initialize temps.

**for** block $P_i \in P$ **do**

Read from disk: $B_{\mu P \nu}^P$

$J_{\mu \nu} = J_{\mu \nu} + \text{ComputeJ}(B_{\mu P \nu}^P, S_{\mu \nu}^b, D_{\mu \nu})$

$K_{\mu \nu} = K_{\mu \nu} + \text{ComputeK}(B_{\mu P \nu}^P, S_{\mu \nu}^b, C_{\mu p}, C'_{\nu p})$

**end for**

**return** $J_{\mu \nu}, K_{\mu \nu}$

An implementation of this kernel will involve using the $B_{\mu P \nu}^P$ integral form with Algorithms 10 and 13 or the $B_{\mu P \nu}^P$ integral form with Algorithms 11 and 14. In the next section, we discuss the performance of these choices in practice.

### 4.2 Results

All methods were implemented in the Psi4 electronic structure software package [13]. The parallelism in Psi4 relies on the shared memory programming model using OpenMP and carries out matrix multiplications using Intel’s Math Kernel Library. Currently, the state of the art for exchange matrix evaluations in Psi4 is Algorithm 11. However, we conjecture
that Algorithm 10 could provide considerable speedups as it eliminates entirely a strided, level 1 BLAS copy.

4.2.1 Multi-core Results

First, we reveal the performance of the above algorithms on multi-core processors. We implemented Algorithms 10 and 13 and incorporated them into a development version of Ps14. Then, we used Ps14’s Self-Consistent Field procedure to produce energies for various systems and basis set combinations. The systems used involved a protein-drug complex, where the drug molecule is omitted and the atoms of the protein are added in a series according to distance from the center of the drug molecule.

The experiments were carried out on two separate architectures. The first includes one node consisting of an Intel Core i7-5930K processor (6 cores at 3.50GHz) and 64GB DRAM. The results for this architecture are included in Table 4.1. The second architecture includes one node consisting of two Intel Xeon E5-2640 processors and 256GB DRAM. The results for this architecture are included in Table 4.2.
Table 4.1: Total execution times for Self-Consistent Field procedures for various systems using Algorithms 10 and 11 and Algorithms 13 and 14 for exchange and Coulomb matrix evaluations, respectively. Total wall times include wall time for the entire program to execute. J and K compute time indicates the total time spent in the Coulomb and exchange matrix evaluation kernels, respectively. System descriptions including number of AO basis functions, \( N_{AO} \), number of auxiliary basis functions, \( N_{aux} \), basis, and number of atoms, \( N_{at} \), are included. All rows are sorted by the product \( N_{aux}N_{AO} \). A speedup column is included for total wall time and is calculated as the total procedure time spent using Algorithms 11 and 14 divided by the total procedure time spent using Algorithms 10 and 13. The experiments were carried out using one node consisting of an Intel Core i7-5930K processor (6 cores at 3.50GHz) and 64GB DRAM.

<table>
<thead>
<tr>
<th>( N_{AO} )</th>
<th>( N_{aux} )</th>
<th>Basis</th>
<th>( N_{at} )</th>
<th>Total Wall Time</th>
<th>J Compute Time</th>
<th>K Compute Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg. 10 &amp; 13</td>
<td>Alg. 11 &amp; 14</td>
<td>spdup</td>
<td>Alg. 14</td>
<td>Alg. 13</td>
<td>Alg. 10</td>
<td>Alg. 11</td>
</tr>
<tr>
<td>147</td>
<td>721</td>
<td>DZ</td>
<td>15</td>
<td>3.8</td>
<td>4.7</td>
<td>1.2</td>
</tr>
<tr>
<td>247</td>
<td>912</td>
<td>aDZ</td>
<td>15</td>
<td>5.8</td>
<td>7.7</td>
<td>1.3</td>
</tr>
<tr>
<td>338</td>
<td>842</td>
<td>TZ</td>
<td>15</td>
<td>7.5</td>
<td>9.9</td>
<td>1.3</td>
</tr>
<tr>
<td>294</td>
<td>1442</td>
<td>DZ</td>
<td>30</td>
<td>12.4</td>
<td>18.1</td>
<td>1.5</td>
</tr>
<tr>
<td>529</td>
<td>1154</td>
<td>aTZ</td>
<td>15</td>
<td>18.7</td>
<td>29.9</td>
<td>1.6</td>
</tr>
<tr>
<td>375</td>
<td>1837</td>
<td>DZ</td>
<td>39</td>
<td>25.4</td>
<td>36.8</td>
<td>1.4</td>
</tr>
<tr>
<td>650</td>
<td>1205</td>
<td>QZ</td>
<td>15</td>
<td>24.7</td>
<td>43.8</td>
<td>1.8</td>
</tr>
<tr>
<td>494</td>
<td>1824</td>
<td>aDZ</td>
<td>30</td>
<td>35.3</td>
<td>54.8</td>
<td>1.6</td>
</tr>
<tr>
<td>676</td>
<td>1684</td>
<td>TZ</td>
<td>30</td>
<td>46.0</td>
<td>74.7</td>
<td>1.6</td>
</tr>
<tr>
<td>522</td>
<td>2558</td>
<td>DZ</td>
<td>54</td>
<td>72.3</td>
<td>109.2</td>
<td>1.5</td>
</tr>
<tr>
<td>631</td>
<td>3238</td>
<td>aDZ</td>
<td>39</td>
<td>81.5</td>
<td>132.9</td>
<td>1.6</td>
</tr>
<tr>
<td>603</td>
<td>2953</td>
<td>DZ</td>
<td>63</td>
<td>117.2</td>
<td>166.9</td>
<td>1.4</td>
</tr>
<tr>
<td>866</td>
<td>2150</td>
<td>TZ</td>
<td>39</td>
<td>109.3</td>
<td>176.6</td>
<td>1.6</td>
</tr>
<tr>
<td>1058</td>
<td>2308</td>
<td>aTZ</td>
<td>30</td>
<td>145.5</td>
<td>278.9</td>
<td>1.9</td>
</tr>
<tr>
<td>756</td>
<td>3696</td>
<td>DZ</td>
<td>81</td>
<td>231.3</td>
<td>328.3</td>
<td>1.4</td>
</tr>
<tr>
<td>878</td>
<td>3240</td>
<td>aDZ</td>
<td>54</td>
<td>232.0</td>
<td>394.4</td>
<td>1.7</td>
</tr>
<tr>
<td>1300</td>
<td>2410</td>
<td>QZ</td>
<td>30</td>
<td>218.0</td>
<td>376.1</td>
<td>1.7</td>
</tr>
<tr>
<td>1204</td>
<td>2992</td>
<td>TZ</td>
<td>54</td>
<td>332.5</td>
<td>504.1</td>
<td>1.5</td>
</tr>
<tr>
<td>1015</td>
<td>3744</td>
<td>aDZ</td>
<td>63</td>
<td>403.1</td>
<td>640.8</td>
<td>1.6</td>
</tr>
<tr>
<td>1357</td>
<td>2953</td>
<td>DZ</td>
<td>63</td>
<td>385.9</td>
<td>726.9</td>
<td>1.4</td>
</tr>
<tr>
<td>974</td>
<td>4766</td>
<td>DZ</td>
<td>103</td>
<td>218.0</td>
<td>376.1</td>
<td>1.7</td>
</tr>
<tr>
<td>1394</td>
<td>3458</td>
<td>QZ</td>
<td>54</td>
<td>3680.4</td>
<td>5241.3</td>
<td>1.4</td>
</tr>
<tr>
<td>1670</td>
<td>3089</td>
<td>aDZ</td>
<td>30</td>
<td>1153.5</td>
<td>1993.4</td>
<td>0.8</td>
</tr>
<tr>
<td>1275</td>
<td>4698</td>
<td>dTZ</td>
<td>81</td>
<td>1379.0</td>
<td>2379.0</td>
<td>1.5</td>
</tr>
<tr>
<td>1758</td>
<td>4341</td>
<td>QZ</td>
<td>81</td>
<td>1372.3</td>
<td>1839.0</td>
<td>1.3</td>
</tr>
<tr>
<td>1886</td>
<td>4108</td>
<td>aTZ</td>
<td>54</td>
<td>3811.0</td>
<td>5764.0</td>
<td>0.6</td>
</tr>
<tr>
<td>1641</td>
<td>6050</td>
<td>aDZ</td>
<td>103</td>
<td>4406.9</td>
<td>6052.3</td>
<td>0.8</td>
</tr>
<tr>
<td>2320</td>
<td>4294</td>
<td>QZ</td>
<td>54</td>
<td>4280.4</td>
<td>5741.3</td>
<td>0.6</td>
</tr>
<tr>
<td>2185</td>
<td>4754</td>
<td>aTZ</td>
<td>63</td>
<td>5744.9</td>
<td>7459.0</td>
<td>0.8</td>
</tr>
<tr>
<td>2258</td>
<td>5589</td>
<td>QZ</td>
<td>103</td>
<td>5474.8</td>
<td>7101.6</td>
<td>0.9</td>
</tr>
<tr>
<td>2690</td>
<td>4973</td>
<td>QZ</td>
<td>63</td>
<td>6892.9</td>
<td>8581.9</td>
<td>0.7</td>
</tr>
<tr>
<td>2760</td>
<td>5988</td>
<td>aTZ</td>
<td>81</td>
<td>11294.6</td>
<td>9350.0</td>
<td>0.8</td>
</tr>
<tr>
<td>3405</td>
<td>6276</td>
<td>QZ</td>
<td>81</td>
<td>14075.7</td>
<td>11354.3</td>
<td>0.8</td>
</tr>
<tr>
<td>3542</td>
<td>7696</td>
<td>aTZ</td>
<td>103</td>
<td>28136.3</td>
<td>23132.6</td>
<td>0.8</td>
</tr>
</tbody>
</table>

† Indicates a disk-based implementation was used.
Table 4.2: Total execution times for Self-Consistent Field procedures for various systems using Algorithms 10 and 11 and Algorithms 13 and 14 for exchange and Coulomb matrix evaluations, respectively. Total wall times include wall time for the entire program to execute. J and K compute time indicates the total time spent in the Coulomb and exchange matrix evaluation kernels, respectively. System descriptions including number of AO basis functions, \(N_{\text{AO}}\), number of auxiliary basis functions, \(N_{\text{aux}}\), basis, and number of atoms, \(N_{\text{at}}\), are included. All rows are sorted by the product \(N_{\text{aux}} \cdot N_{\text{AO}}\). A speedup column is included for total wall time and is calculated as the total procedure time spent using Algorithms 11 and 14 divided by the total procedure time spent using Algorithms 10 and 13. The experiments were carried out using one node consisting of two Intel Xeon E5-2640 processors (10 cores at 2.40GHz) and 256GB DRAM.

<table>
<thead>
<tr>
<th>(N_{\text{AO}})</th>
<th>(N_{\text{aux}})</th>
<th>Basis</th>
<th>(N_{\text{at}})</th>
<th>Total Wall Time</th>
<th>J Compute Time</th>
<th>K Compute Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10 &amp; 13</td>
<td>11 &amp; 14</td>
<td>spdup</td>
</tr>
<tr>
<td>147</td>
<td>721</td>
<td>DZ</td>
<td>15</td>
<td>2.9</td>
<td>3.4</td>
<td>1.2</td>
</tr>
<tr>
<td>247</td>
<td>912</td>
<td>aDZ</td>
<td>15</td>
<td>3.8</td>
<td>5.2</td>
<td>1.4</td>
</tr>
<tr>
<td>338</td>
<td>842</td>
<td>TZ</td>
<td>15</td>
<td>5.0</td>
<td>6.5</td>
<td>1.3</td>
</tr>
<tr>
<td>294</td>
<td>1442</td>
<td>DZ</td>
<td>30</td>
<td>7.5</td>
<td>9.4</td>
<td>1.3</td>
</tr>
<tr>
<td>529</td>
<td>1154</td>
<td>aTZ</td>
<td>15</td>
<td>8.8</td>
<td>13.4</td>
<td>1.5</td>
</tr>
<tr>
<td>375</td>
<td>1837</td>
<td>DZ</td>
<td>39</td>
<td>10.5</td>
<td>17.7</td>
<td>1.7</td>
</tr>
<tr>
<td>650</td>
<td>1205</td>
<td>QZ</td>
<td>15</td>
<td>11.4</td>
<td>18.7</td>
<td>1.6</td>
</tr>
<tr>
<td>494</td>
<td>1824</td>
<td>aDZ</td>
<td>30</td>
<td>14.7</td>
<td>22.9</td>
<td>1.6</td>
</tr>
<tr>
<td>676</td>
<td>1684</td>
<td>TZ</td>
<td>30</td>
<td>17.3</td>
<td>30.6</td>
<td>1.8</td>
</tr>
<tr>
<td>522</td>
<td>2558</td>
<td>DZ</td>
<td>54</td>
<td>21.8</td>
<td>39.3</td>
<td>1.8</td>
</tr>
<tr>
<td>631</td>
<td>2328</td>
<td>aDZ</td>
<td>39</td>
<td>27.2</td>
<td>50.2</td>
<td>1.8</td>
</tr>
<tr>
<td>962</td>
<td>1668</td>
<td>aQZ</td>
<td>15</td>
<td>27.2</td>
<td>45.5</td>
<td>1.7</td>
</tr>
<tr>
<td>603</td>
<td>2953</td>
<td>DZ</td>
<td>63</td>
<td>29.1</td>
<td>57.0</td>
<td>2.0</td>
</tr>
<tr>
<td>866</td>
<td>2150</td>
<td>TZ</td>
<td>39</td>
<td>34.5</td>
<td>60.5</td>
<td>1.8</td>
</tr>
<tr>
<td>1058</td>
<td>2308</td>
<td>aTZ</td>
<td>30</td>
<td>45.5</td>
<td>86.8</td>
<td>1.9</td>
</tr>
<tr>
<td>756</td>
<td>3696</td>
<td>DZ</td>
<td>81</td>
<td>50.4</td>
<td>102.6</td>
<td>2.0</td>
</tr>
<tr>
<td>878</td>
<td>3240</td>
<td>aDZ</td>
<td>54</td>
<td>62.4</td>
<td>119.8</td>
<td>1.9</td>
</tr>
<tr>
<td>1300</td>
<td>2410</td>
<td>QZ</td>
<td>30</td>
<td>58.4</td>
<td>105.1</td>
<td>1.8</td>
</tr>
<tr>
<td>1204</td>
<td>2992</td>
<td>TZ</td>
<td>54</td>
<td>71.0</td>
<td>133.6</td>
<td>1.9</td>
</tr>
<tr>
<td>1015</td>
<td>3744</td>
<td>aDZ</td>
<td>63</td>
<td>88.0</td>
<td>169.5</td>
<td>1.9</td>
</tr>
<tr>
<td>1357</td>
<td>2954</td>
<td>aTZ</td>
<td>39</td>
<td>96.2</td>
<td>181.1</td>
<td>1.9</td>
</tr>
<tr>
<td>974</td>
<td>4766</td>
<td>DZ</td>
<td>103</td>
<td>112.8</td>
<td>226.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1394</td>
<td>3458</td>
<td>TZ</td>
<td>63</td>
<td>103.2</td>
<td>217.8</td>
<td>2.1</td>
</tr>
<tr>
<td>1670</td>
<td>3089</td>
<td>QZ</td>
<td>39</td>
<td>119.9</td>
<td>221.4</td>
<td>1.8</td>
</tr>
<tr>
<td>1275</td>
<td>4698</td>
<td>aDZ</td>
<td>81</td>
<td>175.7</td>
<td>345.9</td>
<td>2.0</td>
</tr>
<tr>
<td>1924</td>
<td>3336</td>
<td>aQZ</td>
<td>30</td>
<td>184.9</td>
<td>346.0</td>
<td>1.9</td>
</tr>
<tr>
<td>1758</td>
<td>4341</td>
<td>TZ</td>
<td>81</td>
<td>207.3</td>
<td>386.2</td>
<td>1.9</td>
</tr>
<tr>
<td>1886</td>
<td>4108</td>
<td>aTZ</td>
<td>54</td>
<td>262.0</td>
<td>471.2</td>
<td>1.8</td>
</tr>
<tr>
<td>1641</td>
<td>6050</td>
<td>aDZ</td>
<td>103</td>
<td>416.8</td>
<td>798.6</td>
<td>1.9</td>
</tr>
<tr>
<td>2320</td>
<td>4294</td>
<td>QZ</td>
<td>54</td>
<td>328.4</td>
<td>646.8</td>
<td>2.0</td>
</tr>
<tr>
<td>2185</td>
<td>4754</td>
<td>aTZ</td>
<td>63</td>
<td>399.4</td>
<td>704.1</td>
<td>1.8</td>
</tr>
<tr>
<td>2474</td>
<td>4284</td>
<td>aQZ</td>
<td>39</td>
<td>481.1</td>
<td>814.1</td>
<td>1.7</td>
</tr>
<tr>
<td>2258</td>
<td>5589</td>
<td>TZ</td>
<td>103</td>
<td>495.3</td>
<td>884.8</td>
<td>1.8</td>
</tr>
<tr>
<td>2690</td>
<td>4973</td>
<td>QZ</td>
<td>63</td>
<td>495.4</td>
<td>837.5</td>
<td>1.7</td>
</tr>
<tr>
<td>3405</td>
<td>6276</td>
<td>QZ</td>
<td>81</td>
<td>1042.8</td>
<td>1713.8</td>
<td>1.6</td>
</tr>
</tbody>
</table>

† Indicates a disk-based implementation was used.
Note that the † symbol is used to indicate when a disk-based implementation is required. For Algorithm 11, it is possible to store up to half of the AO function pairs by applying permutational symmetry. The necessary copy, $B_{P_{i\mu}P_{\nu}} \rightarrow b_{P_{i\mu}P_{\nu}}$, allows for this. Doing so effectively halves the memory requirement for Algorithm 11 with respect to Algorithm 10. For this reason, Algorithm 10 is forced to resort to a disk-based implementation sooner than Algorithm 11.

The results in Table 4.1 and Table 4.2 confirm our analysis of Algorithms 10 and 11. For small enough investigations, procedures using Algorithm 10 will always be more efficient. Total time spent in the exchange matrix evaluation kernel is always smaller for Algorithm 10. Moreover, the speedups obtained on the two-processor architecture (Table 4.2) are larger than what was obtained on the one-processor architecture (Table 4.1). This result reveals the importance of memory locality and data movement on non-uniform memory access (NUMA) architectures. Algorithm 11 requires a strided level 1 BLAS operation and this additional data movement especially hinders performance on NUMA architectures. Only when the procedures switch to a disk-based implementation do strided reads of the $B_{\mu P_{\nu}}$ tensor result in a performance switch when using the two integral memory layouts. Moreover, since the scaling of the Coulomb matrix evaluation is an entire factor smaller than the exchange matrix evaluation, its required time is almost trivial overall.

Lastly, note that the total time spent in the $J$ and $K$ evaluations does not entirely account for the differences in performance. The remaining difference involves the computation of $B_{\mu \nu}^{P}$, which with a metric contraction scaling as $O(N_{aux}^{2}N_{AO}^{2})$, can be the most expensive operation of an SCF procedure. Even though Algorithm 10 does not apply permutational symmetry, it can be more efficient for this operation as both the contractions and disk writes are contiguous.
4.2.2 Manycore Results

Here we examine the performance of the above algorithms on Intel’s Knights Landing manycore processor. We implemented Algorithms 10 and 13 and incorporated them into a development version of Ps14. Then, we used Ps14’s Self-Consistent Field procedure to produce energies for the system in Figure 3.1 using the cc-pVTZ basis set. The experiments were carried out using one node consisting of one Knights Landing processor. Procedures were repeated using varying numbers of cores. Tables 4.3 and 4.4 include total procedure times and exchange matrix evaluation times, respectively.

Table 4.3: Total procedure execution times for a Self-Consistent Field procedure using both Algorithms 10 and 13 and Algorithms 11 and 14 for exchange and Coloumb matrix evaluations, respectively. The programs were executed on one Knights Landing processor using a development version of Ps14. A relative speedup is listed.

<table>
<thead>
<tr>
<th>Cores</th>
<th>10 &amp; 13</th>
<th>11 &amp; 14</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3881.67</td>
<td>8828.52</td>
<td>2.3</td>
</tr>
<tr>
<td>2</td>
<td>2278.7</td>
<td>4833.24</td>
<td>2.1</td>
</tr>
<tr>
<td>4</td>
<td>1289.95</td>
<td>2454.99</td>
<td>1.9</td>
</tr>
<tr>
<td>8</td>
<td>703.25</td>
<td>1290.01</td>
<td>1.8</td>
</tr>
<tr>
<td>16</td>
<td>421.24</td>
<td>735.1</td>
<td>1.7</td>
</tr>
<tr>
<td>32</td>
<td>302.46</td>
<td>507.31</td>
<td>1.7</td>
</tr>
<tr>
<td>64</td>
<td>280.56</td>
<td>467.03</td>
<td>1.7</td>
</tr>
</tbody>
</table>
Table 4.4: Total exchange matrix evaluation times within an Self-Consistent Field procedure using both Algorithms 10 and 11. The programs were executed on one Knights Landing processor using a development version of Psi4. A relative speedup is listed.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Alg. 10</th>
<th>Alg. 11</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1695.519</td>
<td>4217.851</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>979.497</td>
<td>2368.849</td>
<td>2.4</td>
</tr>
<tr>
<td>4</td>
<td>477.094</td>
<td>1159.278</td>
<td>2.4</td>
</tr>
<tr>
<td>8</td>
<td>242.393</td>
<td>572.458</td>
<td>2.4</td>
</tr>
<tr>
<td>16</td>
<td>125.677</td>
<td>291.958</td>
<td>2.3</td>
</tr>
<tr>
<td>32</td>
<td>73.627</td>
<td>188.214</td>
<td>2.6</td>
</tr>
<tr>
<td>64</td>
<td>64.841</td>
<td>180.905</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Again, Algorithm 10 is shown to be superior to Algorithm 11 for in-core procedures. Tables 4.3 and 4.4 reveal the most substantial speedups yet, which can be attributed to the increased NUMA issues for Algorithm 11 on the Knights Landing processor. Moreover, the parallel scaling of Algorithm 10 is superior to Algorithm 11. However, the implementation using Algorithms 10 and 13 is shown to scale worse overall.
CHAPTER 5
CONCLUSIONS

The work in this thesis sought to optimize the computational kernels within the density fitting technique in quantum chemistry. Chapter 2 introduced the use of the Schwarz sparsity screening method for integral computations, metric contractions, and integral transformations. The corresponding implementation and comparison against a non-screening version revealed the drastic importance of utilizing sparsity in quantum chemistry. Although the speedups obtained are considerable, future work should investigate tighter screening techniques involving the spatial sparsity between auxiliary functions. Chapter 3 detailed various optimizations for the integral transformations kernel. First, we identified an ideal sparse memory layout to optimize the parallel scalability and revealed its efficacy on multicore processors. Then, we performed an analysis of the Direct and Store workflows and revealed its applications in practice, showing that the Direct workflow is superior in the context of DFMP2 whereas the Store workflow is superior in the context of DFMCSCF. For future work, we recommend that this analysis be applied further to other contexts, such as SAPT. In chapter 4 we discussed two sparse memory layouts, $B_{\mu\nu}$ and $B_{\mu\nu\nu}$, and their corresponding evaluation algorithms. Importantly, we showed that procedures using Algorithm 10 will always outperform Algorithm 11 if the implementation is entirely in-core. If memory is too constrained and a disk-based implementation is necessary, then procedures using Algorithm 11 will become faster as it yields more ideal disk operations. The crossover in performance between the two algorithms was revealed via an extensive investigation on multi-core and manycore processors across various systems and basis sets. For future work, we recommend investigating optimizations of the disk reads in Algorithm 10. For example, it may be possible to avoid the strided disk reads in Algorithm 10 and 13 if the tensor blocks are preemptively known and stored in separate files.
REFERENCES


