OPTIMIZING COMPUTATIONAL KERNELS IN QUANTUM CHEMISTRY

A Thesis
Presented to
The Academic Faculty

By

Matthew C. Schieber

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computational Science and Engineering

Georgia Institute of Technology

March 2018

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. Three
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: January 11, 2000

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 curiousity 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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 their involving operations. Still, these operations remain as computational bottlenecks which commonly plague quantum chemistry procedures. The goal of this thesis is to investigate various optimizations for these 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 PSI4 electronic structure package and reveal that the exchange matrix evaluation kernel should vary depending on whether a disk-based implementation must be used.

CHAPTER 1
INTRODUCTION AND BACKGROUND

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:

$$(\mu\nu|\lambda\sigma) = \int\int \mu(\mathbf{r}_1)\nu(\mathbf{r}_1)r_{12}^{-1}\lambda(\mathbf{r}_2)\sigma(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2, \tag{1.1}$$

where $\mu, \nu, \lambda, \sigma$ are atomic orbital (AO) basis functions and $\mathbf{r}_1$ and $\mathbf{r}_2$ 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 $\mathcal{O}(N_{AO}^4)$ storage costs (where N_{AO} is the number of AO basis functions), which quickly limits the size of in-core investigations. Moreover, the two fundamental operations involving ERIs are also costly. The first operation involves their evaluation to build 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. Evaluating the 4-center ERIs according to (1.2) and (1.3) requires $\mathcal{O}(N_{AO}^4)$ operations, which is a heavy computational burden that plagues even the most basic quantum chemistry procedures. For example, the AO Fock matrix is built as:

$$F_{\mu\nu} = H_{\mu\nu} + 2J_{\mu\nu} - K_{\mu\nu} \tag{1.4}$$

The simplist quantum chemistry procedure, known as the Self-Consistent Field approach,

involves iteratively reconstructing the Fock matrix until convergence. The most compuatationally demanding operation for this method is by far the evaluating of the Coulomb and exchange matrices.

The second fundamental operation when using ERIs involves their transformation from the AO basis to the molecular orbital (MO) basis. This operation is written as:

$$(pq|rs) = C_{p\mu}C_{q\nu}(\mu\nu|\lambda\sigma)C_{r\lambda}C_{s\sigma}, \tag{1.5}$$

where p, q, r, s denote MO indices and orbital matrices, C, were introduced. These integral transformations scale as $\mathcal{O}(N_{AO}^4 N_p)$ (where N_p is the size of the denoted MO basis) and serve as a major bottleneck for perturbative correlation methods like second-order Møller-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 is one method for alleviating these costs which has been gaining popularity in electronic structure theory. 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:

$$(\mu\nu|P) = \int\int \mu(\mathbf{r}_1)\nu(\mathbf{r}_1)r_{12}^{-1}P(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2 \tag{1.6}$$

$$[J]_{PQ} = \int\int P(\mathbf{r}_1)r_{12}^{-1}Q(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2 \tag{1.7}$$

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

Here we have introduced an auxiliary basis P, Q and the Coloumb fitting metric $[J]_{PQ}$ [2]. In practice, permuational symmetry is almost always used so that only one set of these

3-center integrals need be built:

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

$$(\lambda\sigma|\mu\nu) \approx B^P_{\mu\nu}B^P_{\lambda\sigma} \tag{1.10}$$

The B tensors indicate integrals that have been contracted with the fitting metric. The rank-reduced 3-center ERIs yield a space complexity of $\mathcal{O}(N_{aux}N^2_{AO})$, which allows many more investigations to be run in-core. Using the 3-center ERIs, the Coulomb and exchange matrices are evaluated as:

$$J_{\mu\nu} = B^P_{\mu\nu}B^P_{\lambda\sigma}D_{\lambda\sigma} \tag{1.11}$$

$$K_{\mu\nu} = B^P_{\mu\lambda}B^P_{\nu\sigma}D_{\lambda\sigma} \tag{1.12}$$

Using (1.11), the Coulomb matrix is evaluated in $\mathcal{O}(N_{aux}N^2_{AO})$ operations, which is a great success for reducing computational complexity. On the other hand, (1.12) still requires $\mathcal{O}(N_{aux}N^3_{AO})$ 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 bottleck for most density-fitted procedures. One method for alleviating this bottleneck involves employing knowledge of the form of the density matrix:

$$D_{\lambda\sigma} = C_{p\lambda}C_{p\sigma} \tag{1.13}$$

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:

$$J_{\mu\nu} = B^P_{\mu\nu}B^P_{\lambda\sigma}C_{p\lambda}C_{p\sigma} \tag{1.14}$$

$$K_{\mu\nu} = B^P_{\mu\lambda}B^P_{\nu\sigma}C_{p\lambda}C_{p\sigma} \tag{1.15}$$

Evaluating both (1.14) and (1.15) requires $\mathcal{O}(N_{aux}N_{AO}^2N_p)$ operations. This technique is not benificial 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 $\frac{N_{aux}}{N_p}$, which provides moderate speedups in practice.

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

$$B_{pq}^P = B_{\mu\nu}^P C_{p\mu} C_{q\nu} \tag{1.16}$$

$$(pq|rs) \approx B_{pq}^P B_{rs}^P \tag{1.17}$$

The formal computational scaling of this operation is $\mathcal{O}(N_{aux}N_pN_qN_rN_s)$. 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 space 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 essense of my work involves the marriage of sparsity screening, parallel scaling, minimizing disk IO, and optimal contraction paths. This thesis details the progress made via my research. 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. Finally, Chapter 5 summarizes the results and makes suggestions for future work.

CHAPTER 2

UTILIZING SPATIAL SPARSITY

For all but the smallest molecules, spatial sparsity is very important for achieving computational savings in quantum chemistry. There exists two primary forms of spatial sparsity in (1.6). The first form results from the Gaussian product $\mu(\mathbf{r}_1)\nu(\mathbf{r}_1)$ 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 unecessary 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 $\mathcal{O}(N_{aux}N_{AO}^{1-2})$, 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 take advantage of. see Ref. [3]. In this thesis, I focus on the former, pairwise sparsity which is featured in the product $\mu(\mathbf{r}_1)\nu(\mathbf{r}_1)$. Utilizing this sparsity not only speeds up the computation of the three-index integrals, but 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 $S^b_{\mu\nu}$ 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 integrals in *shell triples*:

$$(MN|Q) = \{(\mu\nu|P), \mu \in M, \nu \in N, Q \in P\} \tag{2.1}$$

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 , $(MN|P)$, where M and N denote AO shells, 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 τ and neglect shells for which

$$MN_{max} < \frac{\tau^2}{(\mu\nu|\mu\nu)_{max}}, \tag{2.2}$$

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

$$(\mu\nu|\mu\nu) < \frac{\tau^2}{(\mu\nu|\mu\nu)_{max}}. \tag{2.3}$$

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 3Å apart. As sparsity increases, the number of significant AO functin pairs contained in these masks becomes $\mathcal{O}(N_{AO}^{1-2})$, where the lower bound is obtained for sufficiently sparse systems.

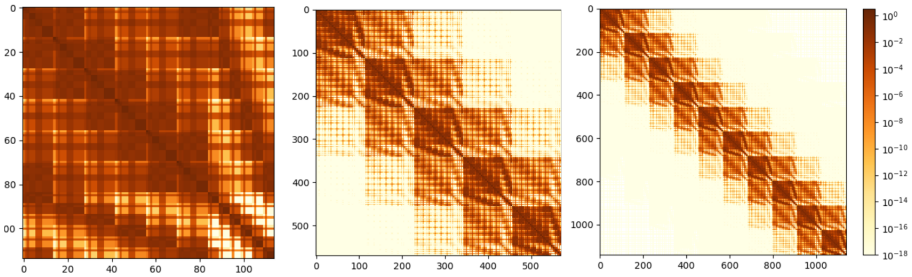


Figure 2.1: Schwarz sparsity masks used for benzene stacks containing 1, 5, and 10 benzenes spaced 3Å apart. Sparsity masks were obtained by evaluating $(\mu\nu|\mu\nu) < \frac{\tau^2}{(\mu\nu|\mu\nu)_{max}}$.

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 readible algorithms, we employ some tensor notation and write $A^P_{\mu\nu}$ to represent the pre-contracted 3-center integrals $(\mu\nu|P)$. To prune these integrals using sparsity, the following algorithm can be employed:

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

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

Here we have introduced the symbol ν^μ, which indicates that ν is restricted to AO functions which are spatially close enough to μ to survive the Schwarz screening process. The superscript in ν^μ indicates a dependence of ν according to μ. 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 $A^P_{\mu\nu^\mu}$ scales as $\mathcal{O}(N_{aux} N_{AO}^{1-2})$. Moroever, the costly metric contraction becomes

$$B^Q_{\mu\nu^\mu} = A^P_{\mu\nu^\mu} [J]^{-\frac{1}{2}}_{PQ} \tag{2.4}$$

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

2.3 Integral Transformation

The transformation of $A^P_{\mu\nu}$ (or $B^Q_{\mu\nu^\nu}$) 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 general MO space indices. A first contraction $A^P_{p\nu} = A^P_{\mu\nu} C_{\mu p}$ will half-transform the integrals and costs $\mathcal{O}(N_{aux} N^2_{AO} N_p)$. The second contraction $A^P_{pq} = A^P_{p\nu} C_{\nu q}$ will cost $\mathcal{O}(N_{aux} N_{AO} N_p N_q)$. 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 most MO spaces. Therefore, reducing the cost of the first contraction would alleviate a bottleneck overall. Thankfully, we can exploit the sparsity of $A^P_{\mu\nu^\mu}$. To do so, we carry out a looping DGEMM through the μ index and apply the sparsity mask to the orbital matrix $C_{\mu p}$. Algorithm 2 illustrates the resulting process:

Algorithm 2 Transform sparse integrals $A^P_{\mu\nu^\mu}$ to MO spaces.

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

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

$C_{\nu q} S^b_{\mu\nu} \rightarrow C_{\nu^\mu q}$

$A^P_{\mu\nu^\mu} C_{\nu^\mu q} \rightarrow A^P_{\mu q}$

end for

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

return A^P_{pq}

2.4 Results

All methods were implemented in the PSI4 electronic structure software package. 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 measur-

ing 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 5Å apart. Transformation time involved the wall time required to carry out the common occupied-virtual transformation, as would be required by density-fitted MP2:

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

Where i, j and a, b denote occupied and virtual spaces, respectively. Algorithm 4 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: $(\mu\nu|Q)C_{i\mu} \rightarrow (i\nu|Q)$, 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.

Benzenes	N	N_{aux}	**Mask Sparsity (%)**
1	264	654	2.6
2	528	1308	24.7
3	792	1962	43.6
4	1056	2616	55.3
5	1320	3270	63.1
6	1584	3924	68.6
7	1848	4578	72.7
8	2112	5232	75.9
9	2376	5886	78.4
10	2640	6540	80.4

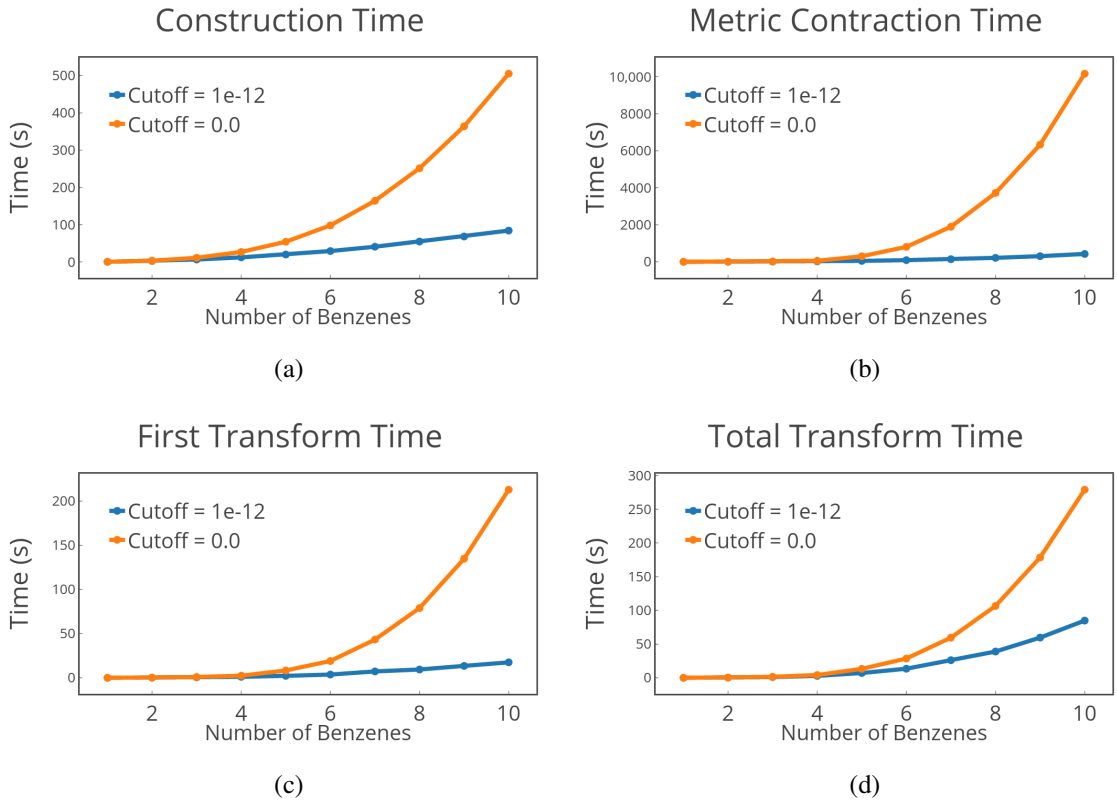


Figure 2.2: Comparison of execution times using sparsity screening (blue) against no sparsity screening (orange). Execution time is plotted against number of benzenes in a benzene stack from one to ten benzenes. Transformations involved computing $(ib|Q)$, where i and b denote occupied and virtual indices, 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) Total transformation times.

The cost of the first contraction becomes $\mathcal{O}(N_{aux}N_{AO}^{1-2}N_p)$. 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.

Operation	**Speedup at 10 benzenes**
Construction	10.6
Metric Contraction	37.8
First Transformation	25.8
Total Transformation	5.5

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 shell triplets for effeciency. 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. Last, 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 thereafter is unrelated to the initial sparsity mask. With sparsity utilization, the first step scales as $\mathcal{O}(N_{aux}N_{AO}^{1-2}p)$, whereas the second transformation step will still scale as $\mathcal{O}(N_{aux}N_pN_q)$. 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 grow 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 moderate 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 adenine-thymine dimer accross various a 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. To illustrate this issue, we will introduce an adapted tensor notation which better indicates memory layout.

Table 3.1: Total memory required for sparse three-index AO integrals for adenine-thymine dimer accross various basis sets.

Basis	**Memory Required**	N_{AO}	N_{aux}
cc-pVDZ	1.0GB	321	1583
aug-cc-pVDZ	4.4GB	536	1986
cc-pVTZ	5.4GB	724	1831
aug-cc-pVTZ	22.0GB	1127	2482
cc-pVQZ	24.3GB	1375	2575
aug-cc-pVQZ	91.4GB	2026	3534
cc-pV5Z	93.3GB	2334	3687
aug-cc-pV5Z	316.1GB	3293	5014

We denote an n-dimensional tensor as $T_{ab...n}$, 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 the 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 $A_{\mu P\mu}$. 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 accross. Primarily, this will involve either the P or μ index. For example, if we choose to block accross 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 μ and ν. 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 $A_{\mu P\mu}$ 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 μ, to block accross. Consider the following two algorithms that block accross either index, respectively:

Algorithm 3 Construct the full AO integrals $B^P_{\mu\nu}$ by blocking accross the P index.

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

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

Compute: $A^{P_i}_{\mu\nu}$

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

Reduction Write: $B^Q_{\mu\nu}$

end for

return $B^Q_{\mu\nu}$

Algorithm 4 Construct the full AO integrals $B^P_{\mu\nu}$ by blocking accross the μ index.

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

for block $\mu_i \in \mu$ **do**

Compute: $A^P_{\mu^i\nu}$

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

Write: $B^Q_{\mu_i\nu}$

end for

return $B^Q_{\mu\nu}$

Of the two algorithms, only Algorithm 4 would respect the memory constraints of a blocking procedure. Note that after the contraction $A^{P_i}_{\mu\nu}[J]^{-\frac{1}{2}}_{P_iQ} \rightarrow B^Q_{\mu\nu}$ in Algorithm 3, the full 3-dimensional quantity $B^Q_{\mu\nu}$ is returned, which would immediately violate memory constraints. For this operation, only one blocking method is *possible*. If we are constrained to blocking accross the μ index, then the tensor form $A_{\mu P\nu}$ 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 imperitive for cost reduction, an optimal implementation must be tailored to fully exploit modern computing hardware. Multicore 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 further necessitates that scientific computing exploit every means of parallelism. Although the density fitting technique has been shown to be challenging to exploit in massively parallel algorithms [4], 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 multicore 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 directly apply a sparsity mask for each submatrix. The resulting sparse form $A_{P\mu\nu\mu}$ contains submatrices of identical structure. However, another form, $A_{\mu P\nu}$, must be considered. The sparse form $A_{\mu P\nu\mu}$ results in submatrices of differing sizes. However, some advantages may be ascertained. We sought to determine which of these two sparse memory layouts, $A_{P\mu\nu\mu}$ or $A_{\mu P\nu\mu}$, is optimal for transforming the integrals into an MO basis. Algorithms 5 and 6 illustrate the difference:

Algorithm 5 Transforming sparse integrals using $A_{P\mu\nu^\mu}$ form.

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

for $P = 0$ to $P = N_{aux} - 1$ **do**

Trim from dense to sparse: $C_{\nu q} S^b_{\mu\nu} \rightarrow C_{\nu^\mu q}$

$A_{P\mu\nu^\mu} C_{\nu^\mu p} \rightarrow A_{P\mu p}$

end for

return $A_{P\mu p}$

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

return A_{Ppq}

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

Require: Sparse AO integrals: $A_{\mu P\nu^\mu}$, 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: $C_{\nu q} S^b_{\mu\nu} \rightarrow C_{\nu^\mu q}$

$A_{\mu P\nu^\mu} C_{\nu^\mu q} \rightarrow A_{\mu Pq}$

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 3 than in Algorithm 4 since N_{aux} will always be larger than N_{AO}. Conversely, the matrix-matrix multiplications occuring in Algorithm 4 are larger. As a result, Algorithm 4 will benefit from delegating larger problem sizes to highly optimized level 3 BLAS routines.

However, the crucial difference is acertained when one considers which index, μ or P, would be most appropriate to block accross for a disk-bound implementation. If we choose

to block accross μ, the result of the final transformation, A_{pq}^{P}, would be incomplete, and a full cummulative disk write of $\mathcal{O}(N_{aux}N_pN_p)$ size would be necessary for each block of μ. 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 accross the P index would not pose this problem, as no contractions occur accros the P index. Therefore, blocking accross P is the only scalable solution.

Now, if it is necessary to block accross 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 succeptible 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 $\mathcal{O}(N_{aux}^2N^2)$ 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:

$$A_{pq}^{Q} = A_{\mu\nu}^{Q}C_{\mu p}C_{\nu q} \tag{3.1}$$

$$B_{pq}^{P} = [J]_{PQ}^{-\frac{1}{2}}A_{pq}^{Q} \tag{3.2}$$

Since N_p and N_q are often much smaller than N_{AO}, the speedup of $\mathcal{O}(\frac{N_{AO}^2}{N_pN_q})$ can be substantial. 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.

Transformation	$\frac{N_{AO}}{N_i} = 2$	$\frac{N_{AO}}{N_i} = 5$	$\frac{N_{AO}}{N_i} = 10$
$(Q\|ij)$	4	25	100
$(Q\|ia)$	4	6.25	11.1
$(Q\|ab)$	4	1.56	1.23

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 disadvatageous in certain contexts. Namely, applying this workflow to methods using many transformations will increase cost unecessarily. 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 outweight 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: $C_{\mu p}, C_{\nu q}$

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 which contracts the metric and then transforms the integrals. Conversely, we refer to the Direct algorithm as the workflow which 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 DFMCSCF [**??**], as well as contexts involving a large number of transformations spaces, such as FS-APT and USAPT [5] [6]. Conversely, the Direct algorithm will be superior in contexts requiring a small number of transformations. Most notably, this includes DFMP2 [7].

3.4 Intermediate Recycling

Most often, quantum chemistry procedures will require numerous integral transformations. For example, a USAPT procedure will employ upwards of 24 unique integral transformations. Since all transformations are carried out at the same time, one should build an implementation that queues the transformations, gathers information, and deploys strategic contraction paths. As mentioned prevously, the first contraction of these integral transformations $A^P_{p\nu} = A^P_{\mu\nu} C_{\mu p}$ should always be carried out on the smallest MO index p possible. Thereafter, transformations using the same intermediate, $A^P_{p\nu}$, should all occur at the same

time to avoid recomputing $A^P_{p\nu}$. For example, if two sets of transformed integrals are required A^P_{uv}, A^P_{up} where $u, v << N$ are active indices and p is a general MO index where $N_p \approx N_{AO}$. For both transformations, the first step will be:

$$A^P_{\mu\nu} C_{\mu u} \rightarrow A^P_{u\nu} \tag{3.3}$$

If this is recognized beforehand, then this operation need only be carried out once and the intermediate $A^P_{u\nu}$ can be recycled for both transformations. The speedup of doing so is $\mathcal{O}(\frac{2N_{AO}+N_v+N_p}{N_{AO}+N_v+N_p})$, 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. 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 ζ level for the Dunning basis sets with ζ = D, T, Q. The characteristics of these systems are included in Table 4. Note that while parallel scaling typically improves for larger systems (i.e. due to larger workloads), this is not gauranteed 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.

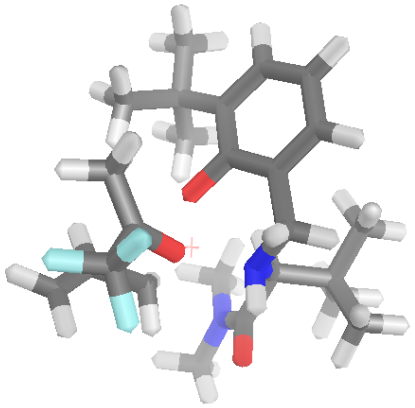


Figure 3.1: Transition state for organoboron addition to trifluooroacetone. Taken from Ref. [**???Lec:2016:768**]

Table 3.3: Characteristics of organoboron catalyst. 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.

Basis	N	N_{aux}	**Mask Sparsity (%)**
cc-pVDZ	671	3277	29.6
cc-pVTZ	1566	3856	41.1
cc-pVQZ	3040	5593	50.2

For each system, we performed the three common transformations:

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

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

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

The experiment was carried out using one node consisting of an Intel Xeon E5-2630 processor (10 cores at 2.20GHz) and 24GB DRAM. Figure 4 includes plots of both speedups and execution times for each system.

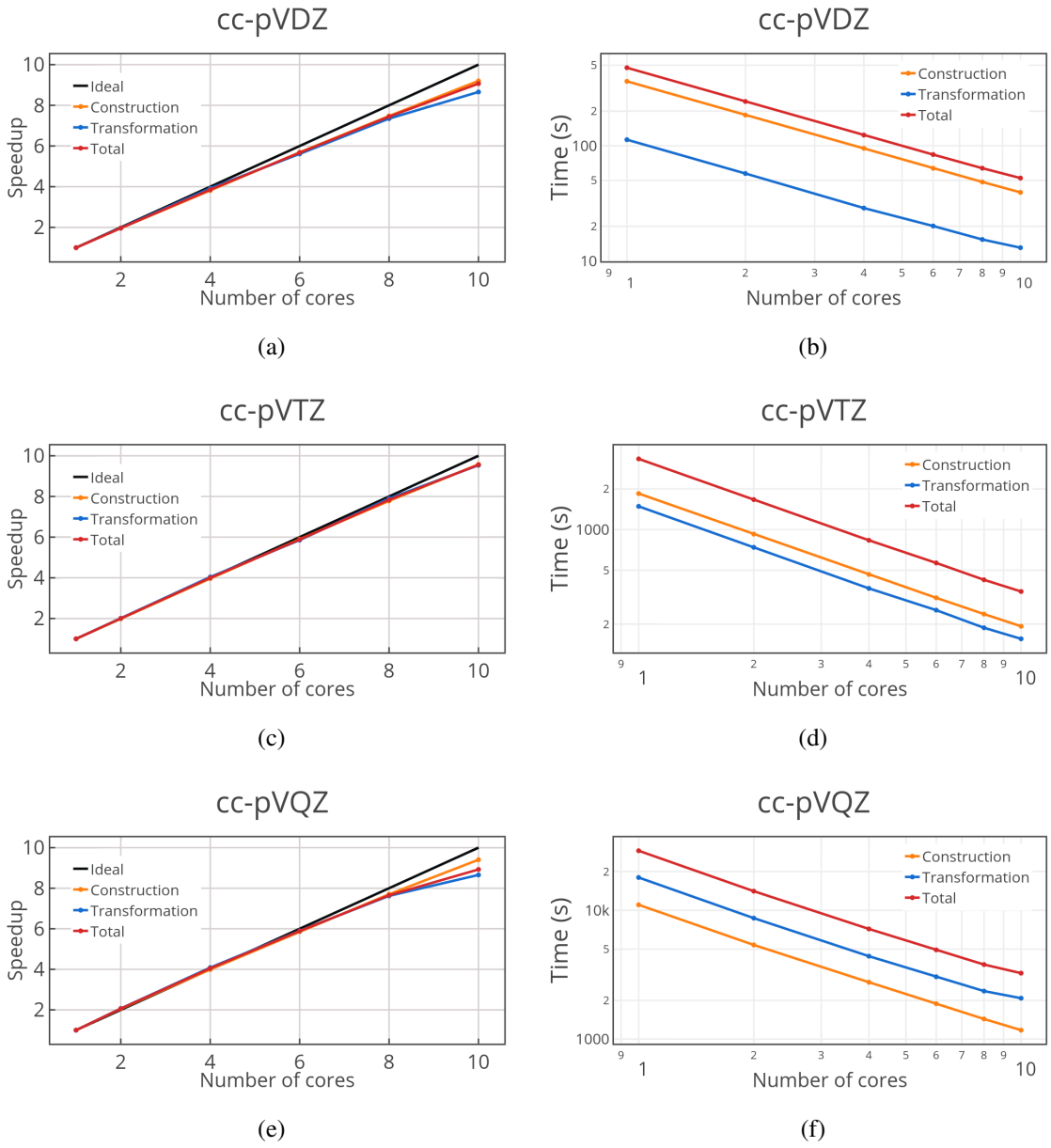


Figure 3.2: Speedup and execution time plots obtained using our optimized memory layout, $B_{\mu P \nu^{\nu}}$, for sparsity screened 3-center integrals. Execution times involve computing three common transformations: $(ij|Q)$, $(ib|Q)$, and $(ab|Q)$, where i, j and a, b denote occupied and virtual indices, 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. 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 ζ = D, T, Q, respectively. The improvement in scaling between ζ = D to ζ = 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 ζ = Q, the memory contraint did not allow for further increase in work per thread. The hindrance in scaling from ζ = T to ζ = Q is explained by the workload inbalance 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 ζ = 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.

Basis	N	N_{aux}	N_{occ}	N_{virt}
cc-pVDZ	671	3277	129	542
cc-pVTZ	1566	3856	129	1437
cc-pVQZ	3040	5593	129	2911

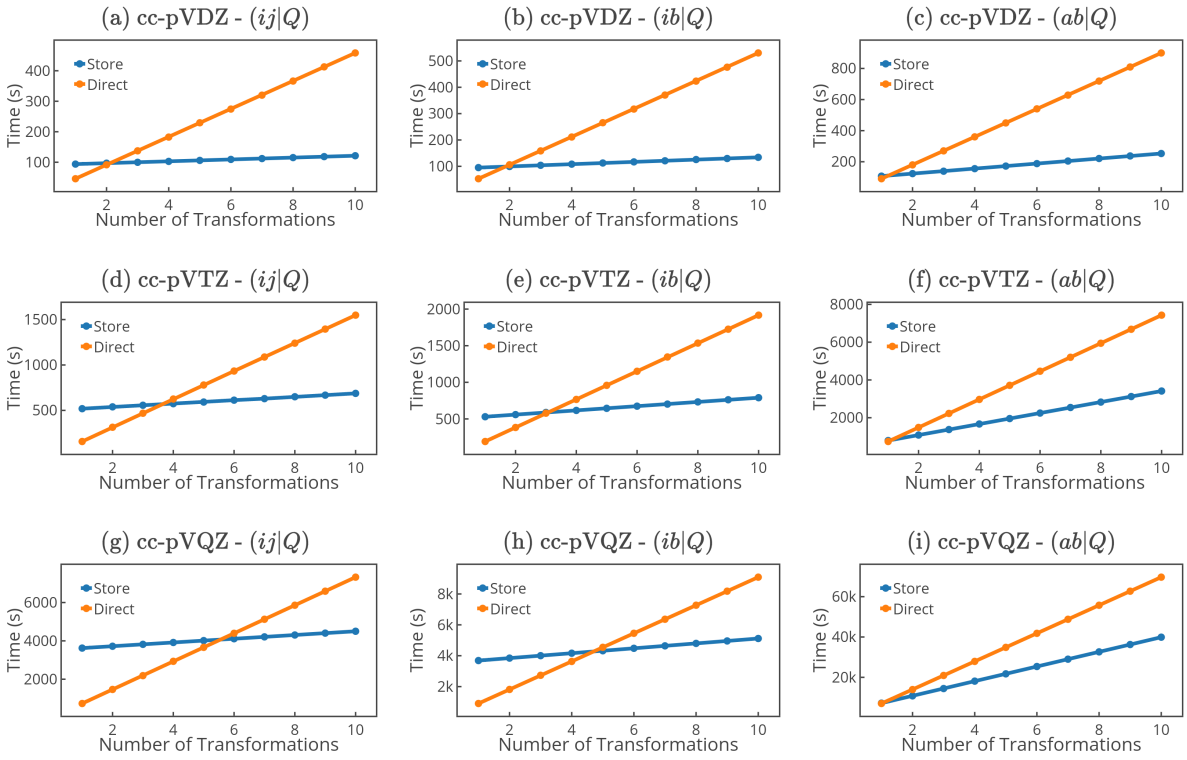


Figure 3.3: Comparison of execution times for the Store and Direct algorithms to complete $(ij|Q)$, $(ib|Q)$, and $(ab|Q)$ transformations across the cc-pVDZ, cc-pVTZ, and cc-pVQZ basis sets. A scan from one to ten transformations was performed. In each case, a crossover occurs as the Direct algorithm 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

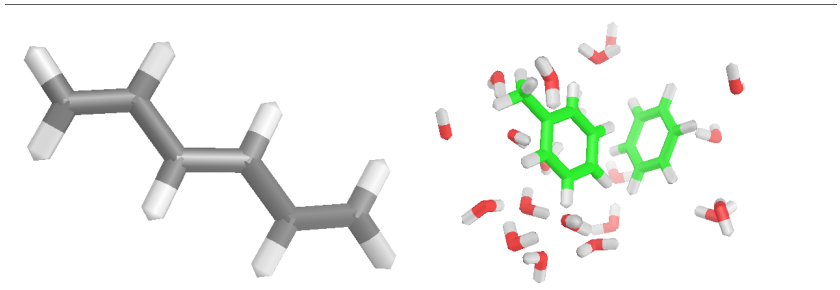
contractions for every transformation. As the number of transformations increase, the expense of these contractions overtakes the speedups of pre-transforming the integrals. Note that a crossover between the two algorithms occurs in each system. This finding supports our conjecture that the Store algorithm 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 FSAPT and USAPT. 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), occuring in fewer transforamtions. This finding is supportive of the proposed speedups in Table 3.2. Therefore procedures using anything larger than a $(ib|Q)$ transformation will recieve 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 determing 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.



(a) (b)

Figure 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.

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

System	**Primary Basis**	**Auxiliary Basis**	N_{aux}	N_{virt}
Hexatriene	cc-pVQZ	cc-pVQZ-jkfit	570	1044
Hexatriene	cc-pVQZ	cc-pVQZ-rifit	570	1232
Benzene-Toluene	jun-cc-pVDZ	jun-cc-pVDZ-jkfit	867	3849
Benzene-Toluene	jun-cc-pVDZ	jun-cc-pVDZ-rifit	867	2901

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 benifits 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: Computatoinal times comparing the Direct and Store algorithms for three-index integral construction and transformations.

System	Procedure	DIRECT	STORE	Speedup
Hexatriene	DFMCSCF	42.3s	7.8s	5.4x
Hexatriene	DFMP2	2.9s	6.0s	2.1x
Benzene-Toluene	DFMCSCF	438.1s	104.8s	4.2x
Benzene-Toluene	DFMP2	31.4s	61.2s	1.9x

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

System	Procedure	DIRECT	STORE	Speedup
Hexatriene	DFMCSCF	173.0s	145.6s	1.2x
Hexatriene	DFMP2	35.1s	40.5s	1.2x
Benzene-Toluene	DFMCSCF	5138.7s	4811.8s	1.1x
Benzene-Toluene	DFMP2	852.7s	856.0s	1.0x

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.

CHAPTER 4

EVALUATING COULOMB AND EXCHANGE MATRICES

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:

$$J_{\mu\nu} = B^P_{\mu\nu} B^P_{\lambda\sigma} D_{\lambda\sigma} \tag{4.1}$$

$$K_{\mu\nu} = B^P_{\mu\lambda} B_{\nu\sigma} C_{p\lambda} C_{p\sigma}, \tag{4.2}$$

where (4.1) and (4.2) are evaluated in $\mathcal{O}(N^2_{AO} N_{aux})$ and $\mathcal{O}(N^2_{AO} N_{aux} N_p)$ operations, respectively. 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 extendible 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, $B_{\mu Pq}$, is actually identical to the intermediate when building the exchange matrix. Moreover, the memory layout of $B_{\mu Pq}$ is ideal for the subsequent contraction to form K. The following algorithm results:

Algorithm 9 Building the K matrix.

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

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

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

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

if $C_{\mu p}! = C_{\nu p}$ **then**

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

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

else

$T_{\nu P p} = T_{\mu P p}$

end if

end for

$T_{\mu P q} T_{\nu P q} \rightarrow K_{\mu\nu}$

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 accross the P index. Then, the algorithm becomes:

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

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

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: $C_{\nu p} S^b_{\mu\nu} \rightarrow C_{\nu^\mu p}$

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

if $C_{\mu p} != C_{\nu p}$ **then**

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

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

else

$T_{\nu P_i p} = T_{\mu P_i p}$

end if

end for

$K_{\mu\nu} = K_{\mu\nu} + T_{\mu P_i q} T_{\nu P_i q}$

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 IO 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 accross the P index (as discussed in Chapter 3), a better memory layout for the sparse integrals is the $B_{P \mu\nu^\mu}$ form, since this form will allow for completely contiguous reads from disk memory. Another possible algorithm can then be formualted:

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

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

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

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

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

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

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

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

if $C_{\mu p}! = C_{\nu p}$ **then**

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

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

else

$T_{\nu P_i p} = T_{\mu P_i p}$

end if

end for

$K = K + T_{\mu P_i p}T_{\mu P_i p}$

end for

return K

Here, we used a buffer, $b_{P^i\nu^\mu}$, to transpose pieces of $B_{P^i\mu\nu^\mu}$ while looping over μ. Although this algorithm may involve a strided copy and additional memory usage, the disk reads for the $B_{P^i\mu\nu^\mu}$ 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 degredation 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: $D_{\mu\nu}S^b_{\mu\nu} \rightarrow D_{\mu\nu\mu}$

$B^P_{\mu\nu\mu}D_{\mu\nu\mu} \rightarrow T^P$

$T^P B^P_{\mu\nu\mu} \rightarrow J_{\mu\nu\mu}$

Unpack from sparse to dense: $J_{\mu\nu\mu} \rightarrow J_{\mu\nu}$

return $J_{\mu\nu}$

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

Algorithm 13 Building the J matrix using $B_{\mu P\nu\mu}$, blocking accross P

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

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

Read from disk: $B_{\mu P_i\nu\mu}$

Initialize: $T_{P_i} = 0$

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

Copy to sparse: $D_{\mu\nu}S^b_{\mu\nu} \rightarrow d_{\nu\mu}$

$T_{P_i} = T_{P_i} + B_{\mu P_i\nu\mu}d_{\nu\mu}$

end for

$T_{P_i}B_{P_i\mu\nu\mu} \rightarrow J_{\mu\nu\mu}$

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

$J_{\mu\nu} = J_{\mu\nu} + J^{\{P_i\}}_{\mu\nu}$

end for

return $J_{\mu\nu}$

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

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

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

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

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

$B_{P_i\mu\nu^\mu}D_{\mu\nu^\mu} \rightarrow T^{P_i}$

$T^{P_i}B_{P_i\mu\nu^\mu} \rightarrow J_{\mu\nu^\mu}$

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

$J_{\mu\nu} = J_{\mu\nu} + J^{\{P_i\}}_{\mu\nu}$

end for

return $J_{\mu\nu}$

Here, we have used the superscript in $J^{\{P_i\}}_{\mu\nu}$ to indicate the contribution to $J_{\mu\nu}$ corresponding 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 performance of Algorithms 13 and 14 will be nearly trivial. Moreover, note that the disk reads in Algorithms 10 and 13 as well as in Algorithms 11 and 14 will be occuring simulataneously. The J and K computational kernel will generally take this form:

Algorithm 15 Coulomb and exchange matrix evaluation kernel.

Require: Sparse AO integrals: $B^P_{\mu\nu^\mu}$, density matrix: $D_{\mu\nu}$, orbital matrices: $C_{\mu p}, C_{\nu p}$, screening mask: $S^b_{\mu\nu}$

Initialize temps.

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

Read from disk: $B^{P_i}_{\mu\nu^\mu}$

J = ComputeJ($B^{P_i}_{\mu\nu^\mu}$, $S^b_{\mu\nu}$, $D_{\mu\nu}$)

K = ComputeK($B^{P_i}_{\mu\nu^\mu}$, $S^b_{\mu\nu}$, $C_{\mu p}, C_{\nu p}$)

end for

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

An implementation of this kernel will involve using the $B_{\mu P\nu^\mu}$ integral form with Algorithms 10 and 13 or the $B_{\mu P\nu^\mu}$ 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. 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.

We implemented Algorithms 10 and 13 and incorporated them into a development version of PSI4. Then, we used PSI4'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 ommitted and the atoms of the protien are added in a series according to distance from the center of the drug molecule.

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 Table 4.1:

Table 4.1: Total execution times for SCF procedures accross various systems using Algorithms 10 and 11 for exchange matrix evaluations. The corresponding Coulomb matrix evaluations were performed using Algorithms 13 and 14, 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 in the leftmost columns. 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 prcedure time spent using Algorithms 11 and 14 dived by the total procedure time spent using Algorithms 10 and 13.

				Total Wall Time			**J Compute Time**		**K Compute Time**	
N_{AO}	N_{aux}	**Basis**	$N_{at.}$	**10 & 13**	**11 & 14**	**spdup**	**Alg. 10**	**Alg. 11**	**Alg. 13**	**Alg. 14**
147	721	DZ	15	3.8	4.7	1.2	0.1	0.0	0.5	0.6
247	912	aDZ	15	5.8	7.7	1.3	0.2	0.1	1.4	2.1
338	842	TZ	15	7.5	9.9	1.3	0.2	0.2	2.3	3.3
294	1442	DZ	30	12.4	18.1	1.5	0.3	0.3	5.6	7.1
529	1154	aTZ	15	18.7	29.9	1.6	0.8	0.8	7.4	13.1
375	1837	DZ	39	25.4	36.8	1.4	0.5	0.5	13.3	16.6
650	1205	QZ	15	24.7	43.8	1.8	1.2	1.1	11.1	19.3
494	1824	aDZ	30	35.3	54.8	1.6	1.0	0.9	18.1	25.7
676	1684	TZ	30	46.0	74.7	1.6	1.2	1.2	28.2	38.6
522	2558	DZ	54	72.3	109.2	1.5	1.1	0.9	48.8	56.8
631	2328	aDZ	39	81.5	132.9	1.6	2.1	2.0	46.8	65.1
603	2953	DZ	63	117.2	166.9	1.4	1.3	1.1	83.2	92.4
866	2150	TZ	39	109.3	176.6	1.6	2.5	2.4	74.7	97.6
1058	2308	aTZ	30	145.5	278.9	1.9	4.6	4.7	91.0	138.6
756	3696	DZ	81	231.3	328.3	1.4	1.8	1.6	176.3	190.9
878	3240	aDZ	54	232.0	394.4	1.7	4.6	4.4	161.9	206.3
1300	2410	QZ	30	218.0	376.1	1.7	5.6	5.8	148.7	198.3
1204	2992	TZ	54	332.5	504.1	1.5	4.9	4.7	256.8	302.4
1015	3744	aDZ	63	403.1	640.8	1.6	6.3	6.1	305.2	361.9
1357	2954	aTZ	39	385.9	726.9	1.9	10.5	12.3	259.2	376.6
974	4766	DZ	103	629.3	845.3	1.3	3.6	3.2	520.0	549.7
1394	3458	TZ	63	601.7	850.5	1.4	6.6	6.2	492.8	551.2
1670	3089	QZ	39	1153.5†	933.4	0.8	12.8†	13.2	395.5†	504.7
1275	4698	aDZ	81	898.9	1379.0	1.5	10.6	10.7	711.5	813.2
1758	4341	TZ	81	1372.3	1839.0	1.3	10.2	9.8	1176.8	1269.1
1886	4108	aTZ	54	3811.0†	2164.0	0.6	25.8†	26.2	931.2†	1183.4
1641	6050	aDZ	103	4406.9†	3528.3	0.8	23.9†	24.6	1958.4†	2161.9
2320	4294	QZ	54	4280.4†	2741.3	0.6	26.3†	25.0	1377.2†	1611.1
2185	4754	aTZ	63	5744.9†	4594.0†	0.8	37.3†	37.0†	1727.3†	2063.1†
2258	5589	TZ	103	5474.8†	4710.6	0.9	23.5†	20.2	3228.0†	3381.7
2690	4973	QZ	63	6892.9†	4581.9	0.7	39.5†	33.7	2598.4†	2893.4
2760	5988	aTZ	81	11294.6†	9350.0†	0.8	64.6†	63.4†	4003.6†	4669.9†
3405	6276	QZ	81	14075.7†	11354.3†	0.8	70.6†	54.7†	6318.9†	6829.2†
3542	7696	aTZ	103	28136.3†	23132.6†	0.8	144.4†	124.7†	11201.2†	12464.4†

† 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 permuatational symmetry. The necessary copy, $B_{P_i\mu\nu^{\mu}} \rightarrow b_{P_i\nu^{\mu}}$, 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 confirms our analysis of Algorithms 10 and 11. For small enough investigations, Algorithm 10 will always be more efficient. Total time spent in the exchange matrix evaluation kernel is always smaller for Algorithm 10. Only when Algorithm 10 switches to a disk-based implementation that requires strided disk reads does it become slower than Algorithm 11 overall. 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.

Last, 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 $B^P_{\mu\nu^{\mu}}$, which with a metric contraction scaling as $\mathcal{O}(N^2_{aux}N^2_{AO})$, can be the most expensive operation of an SCF procedure. Even though Algorithm 10 does not apply permuational symmetry, it can be more efficient for this operation as both the contractions and disk writes are contiguous.

CHAPTER 5
CONCLUSIONS

The work in this thesis sought to optimize the computational kernels within the density fitting technique. Chapter 2 introduced the use of the Schwarz sparsity screening method for integral computations, metric contactions, and integral transformations. The corresponding implementation and comparison against a non-screening version revealed the drastic importance of utilizing sparsity in quantum chemistry. 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 multi-core 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 analyis be applied further to other contexts, such as USAPT and FSAPT. In chapter 4 we discussed two sparse memroy layouts, $B_{P\mu\nu^\mu}$ $B_{\mu P\nu^\mu}$, and their corresponding evaluation algorithms. Importantly, we showed that 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 Algrotihm 11 will become faster it yields more ideal disk operations. The crossover in performance between the two algorithms was revealed via an extensive investigation on multi-core proccesors accross various systems and basis sets. For future work, we recommend investigating optimizations of the disk reads in Algorithm 10.

REFERENCES

[1] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry*. Dover Publications, Inc., 1988.

[2] A. E. DePrince and C. D. Sherrill, "Accuracy and efficiency of coupled-cluster theory using density fitting/cholesky decomposition, frozen natural orbitals, and a t_1-transformed hamiltonian," vol. 9, pp. 2687–2696, 2013.

[3] H.-J. Werner, F. R. Manby, and P. J. Knowles, "Fast linear scaling second-order Møller-Plesset perturbation theory (MP2) using local and density fitting approximations," vol. 118, no. 18, pp. 8149–8160, 2003.

[4] M. Katouda and T. Nakajima, "MPI/OpenMP hybrid parallel algorithm of resolution of identity second-order Møller-Plesset perturbation calculation for massively parallel multicore supercomputers," vol. 9, pp. 5373–5380, 2013.

[5] J. F. Gonthier and C. D. Sherrill, "Density-fitted open-shell symmetry-adapted perturbation theory and application to π-stacking in benzene dimer cation and ionized DNA base pair steps," vol. 145, p. 134 106, 2016.

[6] R. M. Parrish, T. M. Parker, and C. D. Sherrill, "Chemical assignment of symmetry-adapted perturbation theory interaction energy components: The functional-group sapt partition," vol. 10, pp. 4417–4431, 2014.

[7] D. E. Bernholdt and R. J. Harrison, "Large-scale correlated electronic structure calculations: The ri-mp2 method on parallel computers," vol. 250, pp. 477–484, 1996.