Design of efficient block-sparse data structures and associated tensor multiplications for applications in Cluster in Molecule electronic structure calculations

by

Haobo Liu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Chemistry

Waterloo, Ontario, Canada, 2022

© Haobo Liu 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Cluster-in-molecule approach (CiM) is one of the most popular methods in electronic structure calculations for medium to large molecules and systems. The Nooijen group is currently developing a new CiM approach using the range-separated coulomb potential developed by M. Lecours, however the progress does not reach our expectations as we encountered performance bottlenecks from the two-electron three-index integrals. To deal with this problem, we have implemented two block-sparse data structures named the Tile and the Tile Master to provide sparse matrix storage formats and efficient matrix multiplication algorithms benefiting from the high sparsity of the data. The Tile structure focuses on the efficiency of Sparse-matrix dense-matrix multiplication (SpMM), while the Tile Master emphasizes solving the three-index integral problem using the block-sparse structure and compressed three-dimensional array format. Both the Tile structure and the Tile Master are made highly efficient and able to achieve multi-threading under high parallel structures, including the new Intel KNL structure Xeon processors and any Graphic processing unit (GPU) using the Nvidia Compute unified device architecture cores (CUDA) structures. The benchmarking result indicates that the Tile structure is averaging around 2 to 5 times faster than the NumPy dot algorithm, and up to 30 times faster than our previous Compressed sparse row format (CSR) multiplication routine. The Tile Master on the other hand can compress three-index quantities down almost 95% in storage space using the block-sparse structure, and could handle the calculations efficiently using different dense and sparse calculation routines determined by the Atomic orbital (AO) geometries. To sum up, the Tile structure and the Tile Master will provide useful tools to solve the complex three-index integral problem in our CiM approach, as well as other scientific calculation problems with sparse matrices involved.

Keywords: cluster-in-molecule approach, range-separated coulomb potential, sparse-matrix dense-matrix multiplication, block-sparse structure, multi-threading programming, object-oriented programming

Acknowledgements

First of all, I would like to acknowledge my supervisor Dr. Marcel Nooijen for his support and guidance throughout my whole graduate degree. This would not have been possible without all the help and encouragement from professor Nooijen, especially during the period of this global pandemic. I am very grateful to professor Nooijen for providing good advice on my academic research and plans for the future.

I would also like to acknowledge Dr. Michael James Lecours, who has been very helpful during the early stages of my graduate studies. I would like to appreciate the support and friendship from Ph.D. candidate Songhao Bao since we were both undergraduate students in the Nooijen group. I would like to thank Dr. Ondrej Demel and Master candidate Nan Song for working on our CIM approach together. I would like to thank all of my colleagues in the Nooijen group, who have made my graduate experience second to none.

I would also like to appreciate the support and friendship from Ph.D. candidate Neil Raymond, who has been giving me many valuable suggestions on my project. I would like to thank all the people who have provided their help to me throughout my graduate journey.

Finally, I would like to thank my family for their love and support throughout my degree. My mother, professor Yongmei Cui, and my father Senhua Liu, have always been supporting me kindly, and I will not be able to finish my study in Canada without their love. I would also like to thank my girlfriend Erqian Gao, for being my partner since 2015. She has always been supporting me throughout any difficult times in my life and has made me who I am today.

Table of Contents

List of Tables viii

List of Figures x

List of Abbreviations xv

1 Introduction 1

1.1 The cluster-in-molecule approach . . . 2

1.2 The pair natural orbitals method . . . 4

1.3 Unique properties . . . 5

1.4 Future potentials . . . 7

2 The CiM approach 9

2.1 The CiM approach by the Nooijen group . . . 10

2.2 The range-separated Coulomb potential . . . 12

2.3 The Laplace Second-order Møller-Plesset perturbation theory (MP2) method 14

2.4 The orbital selection scheme in the CiM approach . . . 15

2.5 The new data type with the Hartree-Fock exchange algorithm . . . 16

2.6 The sparse-to-sparse and sparse-to-dense matrices multiplication . . . 18

2.7 The efficient three-index integral implementation . . . 19

3 The Tile structure 21
3.1 Background information 22
3.1.1 How two-electron integrals become a problem 23
3.1.2 How common is the two-electron integrals problem 24
3.2 Designing mindset 26
3.2.1 The sparse matrix and the matrix multiplication algorithm 26
3.2.2 Storing the sparse matrix 29
3.3 The Tile structure 32
3.3.1 Proof of concept 33
3.3.2 A closer look at the data 36
3.3.3 The automated data type conversion 38
3.3.4 The fast heuristic matrix screening method 47
3.3.5 The efficient self-pruning method 49
3.4 Performance and potentials 52

4 The Tile Master 56
4.1 Another data structure constructed using the Tile structure 57
4.2 The Tile Master 60
4.2.1 Determining the variable block size using geometry characteristics . 60
4.2.2 Creating 3-D sparse arrays using the Tile Master 61
4.2.3 Special dense and sparse calculation routines using sparsity arrays . 62
4.2.4 Solving three-index integral problem 63
4.3 Performance and potentials 65

5 Summary 68

References 70

APPENDICES 74

A Selected source codes from the Tile structure **75**

A.1 The fast heuristic matrix screening method 75

A.2 The efficient self-pruning method 78

A.3 The initiative method inside the Tile strcture 80

A.4 Overloaded arithmetic operators inside the Tile strcture 82

A.5 Auto-determination of the Block compressed sparse row format (BSR) block size from SciPy.sparse [27] 93

List of Tables

3.1 **Possible inputs for the two-electron three-center integrals.** 26

3.2 **Performance comparsion between the dense matrix multiplication and the sparse matrix multiplication algorithms. Part 1.** The dense algorithm is the dot operation from the NumPy library. The sparse algorithm is the CSR matrix multiplication algorithm from the SciPy Sparse library. 33

3.3 **Performance comparsion between the dense matrix multiplication and the sparse matrix multiplication algorithms. Part 2.** The dense algorithm is the dot operation from the NumPy library. The sparse algorithm is the CSR matrix multiplication algorithm from the SciPy Sparse library. 34

3.4 **Sparsity of the density matrix of selected conjugated alkene chains in different basis sets.** Values inside the matrix below 10^{-5} are pruned. 36

3.5 **Sparsity of the overlap matrix of selected conjugated alkene chains in different basis sets.** Values inside the matrix below 10^{-5} are pruned. 37

3.6 **Rules for simulating the actual intermediate matrices from the calculation.** 37

3.7 **Finding the second conversion threshold using the testing environment size of 500 by 500.** The conversion threshold is determined using the overall average efficiency of the Tile structure for better optimization with other available functions. The CSR conversion has been switched off. 46

3.8 **Finding the first conversion threshold after acquiring the $Threshold_{BSR} = 0.7$ using the testing environment size of 500 by 500.** The conversion threshold is determined using the overall average efficiency of the Tile structure for better optimization with other available functions. 46

3.9 **Available sparsity estimation methods from the Tile structure.** . 47

3.10 **Determine the heuristic estimate threshold using the average estimation error and the overall efficiency of the Tile structure.** The Tile structure has the highest overall efficiency of 361% when the $threshold_{screening} = 0.001$, which means any estimation converging under 0.1% will end the heuristic process with an error rate of sparsity in ±0.51125%. 49

3.11 **The hardware configuration used for the benchmarking.** 52

3.12 **The overall average performance of using the Tile structure under different sizes, compared to the NumPy dot routine.** The thresholds are: $threshold_{csr} = 0.95$, $threshold_{bsr} = 0.7$, $threshold_{screening} = 0.001$ and $threshold_{cutoff} = 10^{-5}$. 53

3.13 **The overall average performance of using the Tile structure under different sizes, compared to the previous CiM routine implementation.** The thresholds are: $threshold_{csr} = 0.95$, $threshold_{bsr} = 0.7$, $threshold_{screening} = 0.001$ and $threshold_{cutoff} = 10^{-5}$. 53

4.1 **Macro-operations versus Micro-operations in the block-sparse data structure.** Macro-operations are completed by the Tile Master, and the Tile structure will provide Micro-operations supports to the Tile Master. . 59

4.2 **The dense and sparse calculation strategy developed in the Tile Master using an external sparsity array.** The determination of the range and the threshold is inside the integral prescreening algorithm [10], which will not be covered in this project. 63

4.3 **Rules for simulating the actual three-index quantity for the calculation.** 65

4.4 **Space efficiency when using the Tile and Tile Master.** For three-index quantities, a 1000 by 1000 by 1000 array would take up to 8Gb of memory and while the Tile Master can keep the three-index sparse compressed to 0.47Gb. 66

4.5 **Time efficiency when using the Tile and Tile Master.** For three-index quantities, the maximum overall size for numpy is 1000 and the memory overflow happened at the maximum overall size. To illustrate the benefit from using the special dense and sparse calculation routines, a CSR calculation routine has been implemented using the compression method. The CSR represents the previous implementation attempt. 67

List of Figures

1.1 **Central processing unit (CPU) times for MP2 calculations compared to the CiM approach.** During the calculation, chain-like molecules like alkanes, C_nH_{2n+2} (n = 16,24,32,40), and water clusters, $(H_2O)_n$ (n = 16,32,40,48), are calculated under the same basis set 6-31+G** using the same system [12]. 2

1.2 **Comparison between the full Coupled-cluster doubles method (CCD) correlation calculation and the CiM approach.** The percentage under the local column displays the relative size of amplitudes during the CiM calculation compared to the full CCD calculation. The size of amplitudes is proportional to the computational efforts, which indicates that the CiM has much better efficiency. All calculations have an accuracy greater than 98.5% [11]. 3

1.3 **Runtime scaling for each step of the conventional Coupled-cluster theory (CC) calculations versus the local CC in their CiM approach.** Here, L represents the system size and n_b,n_o are the number of basis sets and the number of occupied orbitals. The advantages of the CiM approach can be easily observed as the linear scaling property holds for the most expensive CC steps [22]. 4

1.4 **The average number of correlating orbitals produced by the Domain-based local pair natural orbitals approach (PNO) method compared to Canonical expansions.** All calculations were done using the $(gly)_3$ molecule. Three thresholds are used by the calculation to control the size of the internal orbitals; they are $TCutPNO = 10_{-6}$; $TCutPairs = 0E_h$ and $TCutMKN = 10_{-3}$. [16]. 5

1.5 **Comparison between the canonical RI-CCSD(T)-F12 and the PNO version of the local DLPNO-CCSD(T)-F12.** The near linear scaling property can be observed from the PNO method [18]. 6

1.6 **The 'divide and conquer' illustration from the CiM approach.** Only a few localized molecular orbitals are selected to form a calculation subset within many molecule orbitals. Red represents the orbital center; blue represents the localized orbitals that are needed for the calculation; Grey represents the ignored orbitals [15]. 7

2.1 **Common three-index integrals.** In the first equation, a and b are general molecular orbitals (Molecular orbital (MO)), while i and j are occupied MOs. x,y are indices representing an auxilliary basis set which are used in density fitting approximations to two-electron integrals. The first equation is part of the orbitals selection scheme, which helps the CiM approach divide the orbital spaces. The second and third equations construct the exchange matrix K, where μ is localized occupied MOs and β, γ are atomic orbitals (AO). The exchange matrix is later used as an input for the CiM approach [10]. 10

2.2 **The notation of labels in the CiM approach.** 12

2.4 **The range-separated Coulomb potential.**[10] 12

2.3 **The Coulomb potential partitioned into short- and long-range potentials.**[10] 13

2.5 **Shortened algorithm for the short-range Density Fitting Fock contribution.**[10] 14

2.6 **The final representation of the MP2 energy.** This equation is obtained undergoing a Laplace transformation and then the use of the resolution of identity [4]. 14

2.7 **A piece of the three-index integral inside the calculation of the MP2 energy.** The label q represents the auxiliary basis x retrieved from the density fitting. It could also be replaced with the label g if this is a Fourier transformation term depending on the AO labels [4]. 15

2.8 **The proposed orbital selection scheme algorithm.** Provided by the Nooijen group. 16

2.9 **The cProfile result from running the straight-chain alkane $C_{20}H_{42}$ using the range-separated potential method.** Except for the first few lines of the matrix multiplication method, the function called '*getints*2*c*' at line 44 takes up exponentially more time than other methods. The '*getints*2*c*' is a function that calculates the two-electron integrals using the LibCint library, from which the majority of the computational cost is from the three-index integrals. [10, 23]. 17

2.10 **A partially dense matrix representation.** The 's' represents a sparse submatrix, while the 'D' represents a dense submatrix. 19

3.1 **An example of the profile tool provided by the Python library.** [10, 23] 22

3.2 **The repersentation of the Laplace MP2 energy solution.** The simplification steps will transform the two-electron four-center integrals into multiple three-center integrals for faster calculation time. Where q could be the auxiliary basis x retrieved from the density fitting, or g from the Fourier transform depending on the AO labels [4]. 25

3.3 **The algorithm for the short range density fit calculation.** From line 4 and 5, the algorithm divides the α into batches of A and calculate the corresponding two-eletron integral on the go [10]. 27

3.4 **The Coordinate list format (COO) for storing sparse matrices.** 29

3.5 **The Compressed sparse row format (CSR) for storing sparse matrices.** 30

3.6 **The Block compressed sparse row format (BSR) for storing sparse matrices.** 31

3.7 **The Relative Efficiency of the CSR routine compared to the NumPy dot routine in size of 500 by 500 matrices.** The higher the peak, the more efficient the algorithm routine is. The matrix density is the opposite of the matrix sparsity. The average performance of the CSR algorithm is measured at 12.3%. 35

3.8 **The Sparse-matrix dense-matrix multiplication (SpMM) and the Sparse-matrix vector multiplication (SpMV) problem visualized.** [28] 39

3.9 **One solution to the SpMM problem using their Blocked-Ellpack storage format (Blocked-ELL) structure provided by the Nvidia.** By converting the sparse matrix into block-wise dense matrix will provide efficient matrix multiplication performance. [28] 39

3.10 **The Relative Efficiency of the BSR routine compared to the NumPy dot routine in size of 500 by 500 matrices.** The average performance of the BSR algorithm is measured at 14.0%, higher than the 12.3% from the CSR routine. Also, the BSR routine outperforms the CSR in sparsity regions above 70%. 41

3.11 **The Relative Efficiency of the Intel MKL SpMM performance compared to the SciPy CSR and BSR routine in size of 500 by 500 matrices.** Under the same computing system, the Intel MKL provides a nearly 10 times efficient over the CSR routine and 8 times efficient over the BSR routine in the SciPy.sparse library. 42

3.12 **Illustrations on the threshold determination from the automated data type conversion.** The red curve represents the efficiency of the CSR routine versus the increasing matrix density, while the blue represents the BSR routine and the green represents the dense matrix routine. The Tile structure will automatically switch between different matrix data types for the best calculation efficiency. 44

3.13 **The flowchart of the fast heuristic matrix screening method inside the Tile structure.** The random sampling method will ensure the screening algorithm obtains an accurate estimation even with heterogeneous distributed data arrays. 48

3.14 **The overall average performance of using the Tile structure in size of 500 by 500 matrices, compared to the NumPy dot routine implementation.** The thresholds are: $threshold_{csr} = 0.95$, $threshold_{bsr} = 0.7$, $threshold_{screening} = 0.001$ and $threshold_{cutoff} = 10^{-5}$. The average performance of the Tile structure is measured at 355.64%. 54

4.1 **The Tile Master compared to conventional block-wise structures.** 58

4.2 **Assigning AO gauges centers in the integral prescreening algorithm developed by Mike Lecours [10].** The Tile Master will retrieve the geometry information from this algorithm and determine the most appropriate block size. 60

4.3 **The three-dimensional sparse quantity storing format in the Tile Master data structure.** The Tile Master uses three different indices to map the three-index integral inside each grid calculation [10]. 62

4.4 **The flowchart of solving one of the three-index integral problems (calculation of the exchange matrices) using the Tile Master and the Tile structure.** To prevent the data from exceeding the Dynamic random-access memory (DRAM) limitation, every three-dimensional quantity is stored in compressed sparse array format. During the multiplication routine, the self-screening feature inside the Tile structure has been turned off, and the sparsity information will be coming from the provided external sparsity array. 64

List of Abbreviations

AO Atomic orbital iii, xi–xiii, 10, 11, 15, 18, 23, 25, 26, 56, 60–62

Blocked-ELL Blocked-Ellpack storage format xii, 39

BSR Block compressed sparse row format vii, viii, xii, xiii, 29, 31, 32, 39–46, 51, 57, 59, 93, 94

CC Coupled-cluster theory x, 1, 3, 4, 11

CCD Coupled-cluster doubles method x, 3

CCSD Coupled-cluster single-double method 5

CCSD(T) Coupled-cluster single-double and perturbative triple method 5, 6, 14

CEPAs Coupled-electron pair approximations 4

CI Full configuration interaction 1

CiM Cluster-in-molecule approach iii, v, vi, ix–xi, 1–12, 14–16, 18–21, 24–26, 32–34, 36, 38, 47, 49, 50, 53, 56, 60, 61, 65, 69

COO Coordinate list format xii, 29–31, 33, 34, 36, 61

CPFs Coupled-pair functionals 4

CPU Central processing unit x, 2, 8, 40, 41, 43, 52

CSR Compressed sparse row format iii, viii, ix, xii, xiii, 29–36, 38, 40–42, 44–47, 50, 51, 53, 57, 67

CUDA Compute unified device architecture cores iii, 8, 42, 43, 59

DFT Density functional theory 1

DLPNO Domain-based local pair natural orbital formulation 5

DRAM Dynamic random-access memory xiv, 9, 10, 15, 21, 24, 26, 34, 38, 40, 43, 49, 52, 61, 64, 65

FEP Finite element problems 28

GPU Graphic processing unit iii, 8, 40–43, 52, 59, 68

HDD Hard disk drive 24, 49

LMOs Occupied orthogonal localized molecular orbitals 3, 5, 8, 14, 18, 26

MO Molecular orbital xi, 10

MP2 Second-order Møller-Plesset perturbation theory v, x, xi, 1, 2, 4, 11, 14, 15, 18

OOP Object-oriented programming 32, 38, 43, 47, 52, 55

PNO Domain-based local pair natural orbitals approach x, 1, 4–6, 8

PySCF Python module for quantum chemistry platform 8, 21, 26, 32

spGEMM Sparse general matrix-matrix multiplication 34, 41, 46

SpMM Sparse-matrix dense-matrix multiplication iii, xii, xiii, 38–43, 45–47, 52, 57, 63, 68

SpMV Sparse-matrix vector multiplication xii, 39, 46

Chapter 1

Introduction

The ability to perform accurate and efficient electronic structure calculations in large molecules and systems has been the ultimate goal of computational chemistry. For the past decades, the Density functional theory (DFT) has been considered the prevalent method with lower computational cost than traditional methods. However, there are challenges for the DFT to calculate the correlation energy for strongly correlated systems, excited states, or the inclusion of van der Waals interactions. [2] To further improve the calculation accuracy, many wave function-based approaches are developed for better calculation schemes in electronic exchange and correlation energy.

Two of the significant routines are the MP2 [20] and the CC [1]. The MP2 can be used to approximate the correlation energy for molecules, but the performance can be erratic under transition structures [21]. On the other hand, the CC rapidly converges towards the Full configuration interaction (CI), producing very accurate results within reasonable computational costs [6].

The MP2 and the CC theory are both considered accurate and efficient when the system size is small to medium, where most of the molecular orbitals are delocalized. However, when a larger molecule is given to the calculation, the delocalized orbitals are likely to cause errors and reduce efficiency due to their high computational scaling concerning the system size [3, 9]. A general solution to the problem is to localize the occupied orbitals, which could reduce the number of orbitals needed during the calculation. Two approaches have been developed based on the idea; one is called the PNO, while the other is called the CiM [14, 16]. These two methods are often put together for comparison, and CiM is usually regarded as a better approach. To further explain the advantage of CiM, a review will be given according to their history, unique properties, and future potentials.

1.1 The cluster-in-molecule approach

The CiM approach was first introduced by the Li group in 2002 and is well known for its ability to achieve linearly scaling when the system size gets bigger [11]. It also emphasizes the importance of dividing a large molecular into many small subsets containing limited occupied and unoccupied localized molecule orbitals [13, 14]. This is considered a significant improvement compared to some of the conventional electron correlation methods (Figure 1.1). For example, the MP2 is widely considered as the cheapest correlation method, while it still scales exponentially as $O(N^5)$ [8].

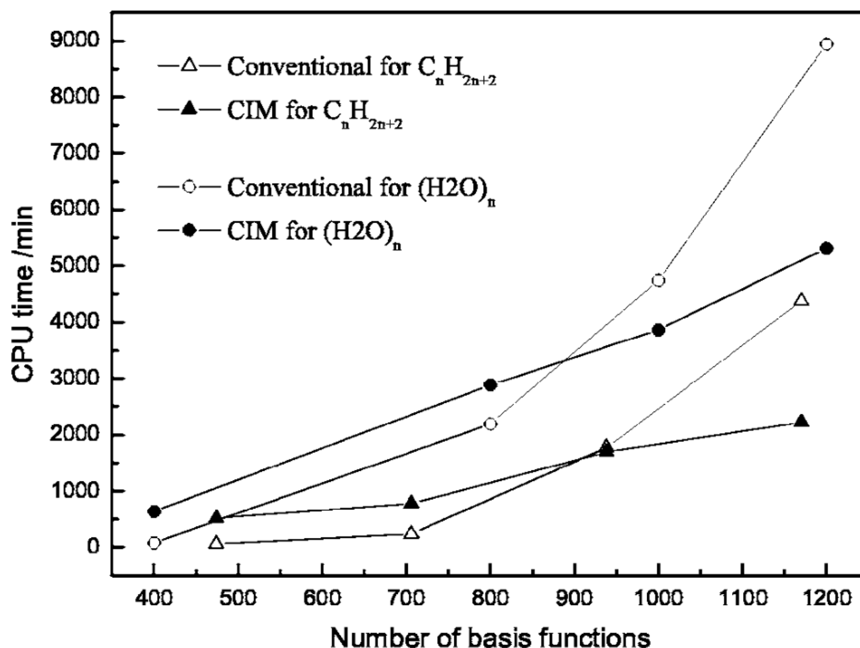

Figure 1.1: **CPU times for MP2 calculations compared to the CiM approach.** During the calculation, chain-like molecules like alkanes, C_nH_{2n+2} (n = 16,24,32,40), and water clusters, $(H_2O)_n$ (n = 16,32,40,48), are calculated under the same basis set 6-31+G** using the same system [12].

However, each subset of localized orbitals inside the CiM approach requires a separate MP2 calculation for the subset, which will require an extra cost in integral transformations and orbital selections. Therefore, the conventional method could yield faster results than the CiM approach at the beginning (Figure 1.1). According to their research, two approxi-

mations are used to reduce the amplitude of the calculation for the CCD resulting in better computing efficiency. The first approximation is by restricting excitations of some closely ranged Occupied orthogonal localized molecular orbitals (LMOs) into especially associated virtual LMOs. As a result, the calculation effort for large molecular can be reduced significantly and scales only linearly with molecular size (Figure 1.2). [11] The second approximation is the dividing of the system. The equation set is divided into various subsystems when solving the amplitude equations. Each subsystem is made up of an orbital cluster and the local environmental domain of the cluster. With these two improvements, this calculation still can recover more than 98.5% correlation energy compared to former methods [11].

Molecule	Basis Set[a]	E_{SCF}	Number of Amplitudes	
			Full	Local[b]
1	6-31G (82)	−235.271544	1.49	0.55 (36.6%)
2	6-31G (108)	−313.308413	4.53	0.73 (16.1%)
3	6-31G (134)	−391.345282	10.78	0.93 (8.6%)
4	6-31G (70)	−231.725501	0.82	0.32 (38.7%)
5	6-31G (92)	−308.586690	2.46	0.47 (19.0%)
6	6-31G (114)	−385.447962	5.83	0.65 (11.1%)
7	6-31G (88)	−307.412445	2.08	0.50 (23.9%)
8	6-31G (93)	−290.383341	2.54	0.88 (34.8%)
9	6-31G* (49)	−188.762309	0.14	0.14 (100.0%)
10	6-31G* (98)	−377.548971	2.32	0.58 (25.0%)

Figure 1.2: **Comparison between the full CCD correlation calculation and the CiM approach.** The percentage under the local column displays the relative size of amplitudes during the CiM calculation compared to the full CCD calculation. The size of amplitudes is proportional to the computational efforts, which indicates that the CiM has much better efficiency. All calculations have an accuracy greater than 98.5% [11].

Another very good example on the CiM approach is from the Kallay group. They have been developing a general-order local CC method based on the CiM approach under the inspiration of the work of Li [11, 22]. They also evaluated the runtime scaling for commonly used algorithms like the conventional CC calculations versus the local CC in their CiM approach, which could provide more solid support for the excellent performance of the CiM approach and show the linear scaling ability. (Figure 1.3)

HF-SCF	n_b^4	L^4
Boys localization	n_o^3	L^3
Integral transformation from the AO to the MO basis	n_b^5	L^5
Calculation of the density matrices	$n_o^2 n_v^3$	L^5
Integral transformation from the MO basis to the local subspaces	$n_o n_l n_b^4$	L^5
Canonicalization in the local subspaces	$n_o n_l^5$	L
Calculation of local CCSD correlation energy	$n_o n_{lo}^2 n_{lv}^4$	L
Calculation of local CCSD(T) correlation energy	$n_o n_{lo}^3 n_{lv}^4$	L
Calculation of local CCSDT correlation energy	$n_o n_{lo}^3 n_{lv}^5$	L
Calculation of local CCSDT(Q) correlation energy	$n_o n_{lo}^4 n_{lv}^5$	L
Any local CC calculation		L^5
Conventional CCSD calculation	$n_o^2 n_v^4$	L^6
Conventional CCSD(T) calculation	$n_o^3 n_v^4$	L^7
Conventional CCSDT calculation	$n_o^3 n_v^5$	L^8
Conventional CCSDT(Q) calculation	$n_o^4 n_v^5$	L^9

Figure 1.3: **Runtime scaling for each step of the conventional CC calculations versus the local CC in their CiM approach.** Here, L represents the system size and n_b,n_o are the number of basis sets and the number of occupied orbitals. The advantages of the CiM approach can be easily observed as the linear scaling property holds for the most expensive CC steps [22].

1.2 The pair natural orbitals method

The PNO method was developed based on two popular theories in the 1970s and 1980s. The first one is called the Coupled-electron pair approximations (CEPAs), while the second is the Coupled-pair functionals (CPFs). According to Neese et al. [16], both methods have better accuracy than the MP2 and other popular density functional theory-based approaches. These methods were precursors of CC, and today the PNO method is used mainly in conjunction with the coupled-cluster (CC). In addition, the PNO method makes extensive use of the resolution of the identity technique to speed up the integral transformation, creating a closed shell CEPAs and CPFs algorithm [16]. They can approximate the external space by compressing the orbitals and minimizing the size of their localized internal orbitals. Eventually, the number of correlating orbitals that are needed for the calculations will be exceptionally smaller than the canonical expansion like MP2 (Figure 1.4) [16].

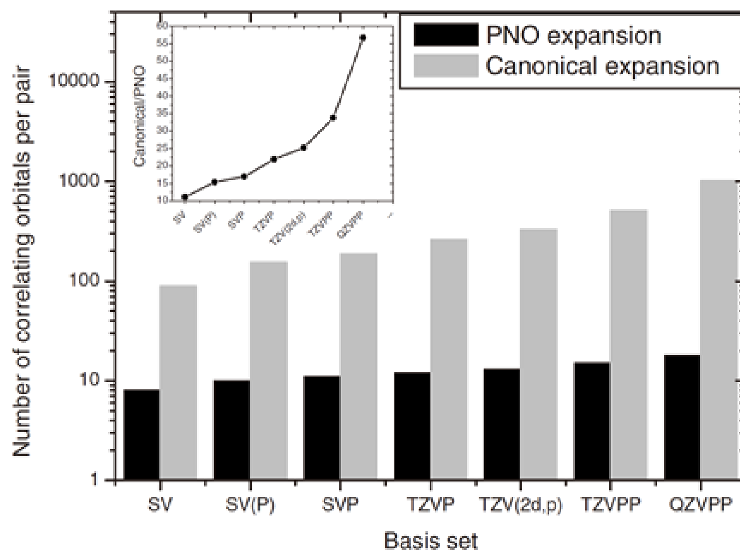

Figure 1.4: **The average number of correlating orbitals produced by the PNO method compared to Canonical expansions.** All calculations were done using the $(gly)_3$ molecule. Three thresholds are used by the calculation to control the size of the internal orbitals; they are $TCutPNO = 10_{-6}$; $TCutPairs = 0E_h$ and $TCutMKN = 10_{-3}$. [16].

In their latest findings, they combine the Coupled-cluster single-double and perturbative triple method (CCSD(T)) with the pair natural orbitals (PNO) methods. The use of PNO helps to reduce the size of the unoccupied space and generate more compact amplitudes, which the explicitly correlated Coupled-cluster single-double method (CCSD) further uses [18]. This method is called the Domain-based local pair natural orbital formulation (DLPNO), and the performance can be found in Figure 1.5.

1.3 Unique properties

The critical difference between the CiM approach and the PNO method is the way they reduce their correlating orbitals. In the case of the PNO method, the Neese group defines a set of specific virtual orbitals to correlate each pair of localized molecular orbitals (LMOs). The size of the virtual orbitals can be relatively small; however, it might lead

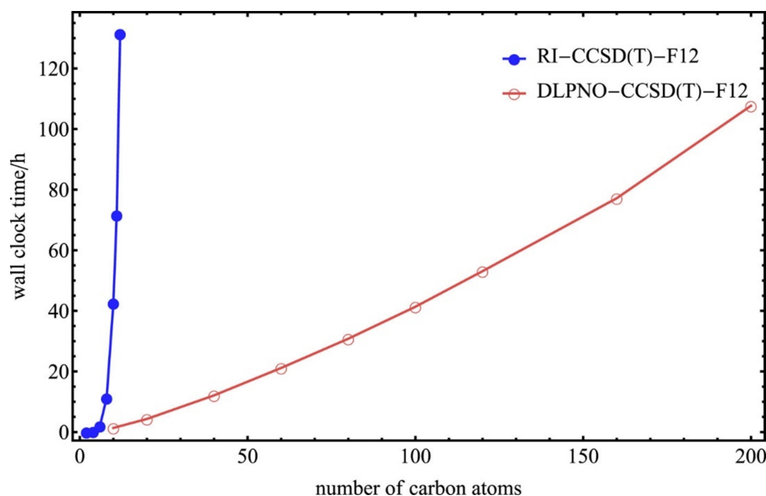

Figure 1.5: **Comparison between the canonical RI-CCSD(T)-F12 and the PNO version of the local DLPNO-CCSD(T)-F12.** The near linear scaling property can be observed from the PNO method [18].

to complex equations due to the different orbitals for each pair. In other words, PNO can solve equations for the complete molecular system and truncate their contributions simultaneously. Instead of solving the entire calculation, the Li group adapts the coupled-cluster single-double and perturbative triple (CCSD(T)) method to solve a large number of small calculations that each has a subset of its orbitals (Figure 1.6) [13, 14, 16, 17].

This quickly becomes another advantage of the CiM approach because each small calculations are parallel and can be beneficial when implemented to multi-core processors, where calculation can be run in parallelization well. Despite having similar accuracy, the PNO approach has to keep track of the set of orthonormal orbitals that will be used throughout the calculations. More problems arise as the system size increases, such as memory and storage space limitations. The CiM approach can further reduce the amount of space taken during each batch of the calculation, which grants the ability for smaller machines to run larger calculations. Therefore, compared to using certain thresholds to control the number of virtual orbitals during the PNO approach, the CiM approach projects a much better way of controlling the size of a system during the calculation [13, 16, 22].

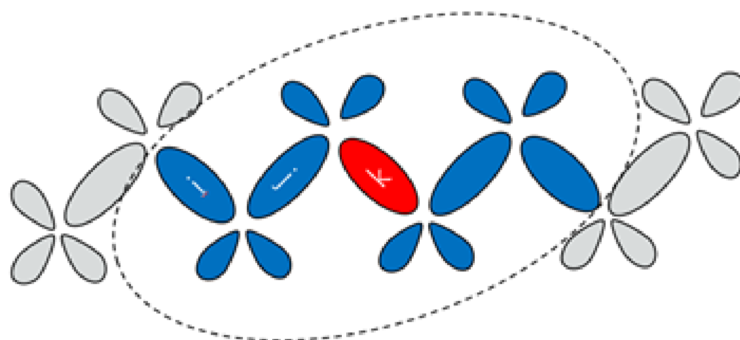

Figure 1.6: **The 'divide and conquer' illustration from the CiM approach.** Only a few localized molecular orbitals are selected to form a calculation subset within many molecule orbitals. Red represents the orbital center; blue represents the localized orbitals that are needed for the calculation; Grey represents the ignored orbitals [15].

1.4 Future potentials

When comparing different methods in theoretical chemistry, taking the potential for future developments into account is usually very important. As the processing power and the computational capability proliferate every day, it is crucial to plan ahead and adapt to the new tools. To maintain the linear scaling time complexity for the CiM approach, a highly efficient integral transformation and orbital selection scheme could play crucial roles in the future development of the CiM method. Using the unique property of the pivoted Cholesky decomposition algorithm, the program can quickly screen out the occupied orbitals. In the Nooijen group the orbital localization and orbital selection schemes in (future) CiM implementations use a pivoted Cholesky approach. Work is in progress to make this approach efficient for block sparse matrices. This Block sparse aspect is also a crucial ingredient for further efficicencies and will be discussed in detail in the thesis.

Currently CiM is used exclusively for ground state calculations. A major attraction of CiM is that one can use conventional programs to run the subsystem calculations. The goal is to extend the CiM approach to calculations for excited states and multireference situations. In the Nooijen group efficient Transform and Diagonalize methodologies are developed that in the final step require a highly compact diagonalization of a transformed effective hamiltonian. Using CiM ideas the effective Hamiltonian can be constructed piecemeal using subsystem calculations, and the final diagonalization is not a bottleneck. Another exciting direction of research based on the inherent parallelization of the CiM approach is implementing the theory with massively paralleled computing devices like a

graphic processing unit (Graphic processing unit (GPU)) [19]. Using the python package 'TensorFlow', they are able to implement a set of molecule orbitals coefficients on their central processing unit (CPU) and distribute the computation assignments to many CUDA, thus speeding up the process. The CiM approach has a much higher potential than any other method by increasing the processing power and keeping the algorithm highly efficient.

In summary the broad goal of the research is both to make ground state calculations easier and more efficient, to provide interfaces with various packages like ACESii, Python module for quantum chemistry platform (PySCF) and ORCA, and to extend the CiM ideas to excited states.

To sum up, the cluster-in-molecular (CiM) approach is one the most popular method used in electronic structure calculations for its efficiency and accuracy. Compared to the pair natural orbitals (PNO) method, the CiM approach uses the idea of divide-and-conquer to partition an extensive system. It keeps the calculation running smaller and parallel [13, 16]. Unlike the PNO method finding individual virtual orbitals for each pair of LMOs, the CiM approach will reduce its computation cost by determining its localized subset of orbitals in each parallel subroutine. Moreover, the CiM approach will have more potential in future research for its inherent parallelization ability as the processing power of today continues to proliferate.

In the next Chapter, we will start by explaining the CiM approach from the Nooijen group to give a broad overview. Then, some essential knowledge and existing methods and techniques required for further research will be given. At the end of Chapter 2, the idea of a new datatype will be proposed with the expected features. In Chapter 3, we will present the new Tile structure from the designing mindset to the covering of the important features. At the end of Chapter 3, we will present the benchmarking result and the performance evaluation for the Tile structure. In Chapter 4, we will explain the Tile Master structure and our solution to the two-electron three-index integral problems or any general tensor multiplication problems. At the end of Chapter 4, the benchmarking result for the Tile Master will be presented, and a final discussion will be made over the capabilities and the potential of these new data structures.

Chapter 2

The CiM approach

Building an accurate and efficient cluster-in-molecular (CiM) algorithm can have many challenges. Among them all, the most significant challenge is choosing a reasonable balance point between accuracy and speed. Therefore, the ultimate goal is to use as many approximations as possible while maintaining a reasonable accuracy like $error < 10^{-7}$. However, using many approximations complicates the routine of calculation, which further increases the difficulty in code implementations. As the code gets complicated, many other problems can arise as the number of subroutines and data size are significant. For example, the density matrix of a water molecule in $cc - pVTZ$ has a size of 58 by 58. It takes a few seconds to generate the density matrix for water, whereas it takes almost 3 hours to generate a density matrix for the Buckminsterfullerene, or C_{60} under the same basis set $cc - pVTZ$. Also, the density matrix for C_{60} has a size of 2,668 by 2,668, which is considerably enormous to handle during the calculation routine [23, 24, 25]. The size of the DRAM in a computer is limited, and it is vital to manage the storage space wisely during the calculation.

Another major challenge when designing a CiM algorithm is handling the three-index integrals. It is one of the most common calculation subroutines that is being used iteratively throughout the whole process. For example, the following three equations (Figure 2.1) are retrieved from a CiM procedure being developed by the Nooijen's group [10].

Due to the unique 'divide and conquer' property of the CiM approach, the subset for each subroutine needs to be determined separately [13, 14]. To determine the orbitals, a transformation of integrals from atomic orbitals to projected orthonormal orbital space of interest has to be accomplished first. The three-index integrals are required during this process, which means it will run iteratively throughout the CiM approach until every subset

$$(ai|bj) = \sum_{g} (ai|g)(bj|g)^{*} \qquad (1)$$

$$I_k(\mu\beta, x) \; += \sum_{y} (\mu\beta, y) m_{y,x} \qquad (2)$$

$$K_{\beta\gamma} \; += \sum_{\mu x} I_K(\mu\beta, x) I_K(\mu\gamma, x) \qquad (3)$$

Figure 2.1: **Common three-index integrals.** In the first equation, a and b are general molecular orbitals (MO), while i and j are occupied MOs. x,y are indices representing an auxilliary basis set which are used in density fitting approximations to two-electron integrals. The first equation is part of the orbitals selection scheme, which helps the CiM approach divide the orbital spaces. The second and third equations construct the exchange matrix K, where μ is localized occupied MOs and β, γ are atomic orbitals (AO). The exchange matrix is later used as an input for the CiM approach [10].

of the subroutine is determined. Therefore, the three-index integrals' efficiency contributes a significant portion to the overall efficiency.

Sometimes, different implementations of the same algorithm can result in different efficiencies and accuracies. Therefore, it is critical to find the best way to run a calculation and reduce the runtime as much as possible. In most cases, the algorithm needs to be explicitly developed in a ground-up fashion. Inside the electronic structure calculations, the matrix multiplications are used extensively since most variables are matrices. However, a large population of the matrices are made of 'sparse matrices', in which few of its elements are non-zero [7]. Therefore, the matrix multiplication algorithm inside the CiM approach could be redesigned and optimized to handle multiplications between different types of matrices. By doing so, the CiM approach can reduce the required DRAM space and the total runtime.

2.1 The CiM approach by the Nooijen group

The cluster-in-molecular (CiM) approach has been proven to be one of the most accurate and efficient post-Hartree-Fock methods. Inspired by the work of Kallay and Li, the

Nooijen group is currently developing a general method that could be viewed as a CiM approach. The method starts with a Hartree-Fock calculation to obtain a set of localized orbitals, where later they are separated into multiple domains in a 'divide and conquer' fashion. To be more specific, each domain contains a subset of occupied and virtual orbitals derived from the central set of orbitals, and they are optimized to perform conventional calculation routines efficiently. The size of the subset during the calculation can be determined within the program and self-adjusted using a new developing data structure called 'the Tile structure', which is capable of highly parallel calculations. At the end of the method, the quantities of interests within each subset can be easily assembled.

Our developing method has many innovations compared to other present available CiM approaches. One of the most distinctive features is that the Coulomb interactions and the AO electron repulsion integrals are partitioned based on their range [10]. This method is called the range-separated coulomb potential, and it is developed by the Ph.D. student Michael James Lecours. The implementation of this method has displayed near-linear scaling efficiency for solving integrals for two-electron Coulomb interaction [10]. Unfortunately, the method is relatively slow at this moment due to the use of entirely sparse matrix multiplication throughout. A primary goal is to improve this by using a combination of sparse and dense matrix multiplication to improve efficiency. Implementing the block-sparse matrix multiply is a primary goal of this research project and will be discussed further below. Further details will be explained in the subsection below. Another innovative feature is the use of the Laplace MP2 method based on the range-separated Coulomb potential. The Laplace MP2 energy is later used as the reference energy for the entire system and can be used to calculate the long-range energies [4]. The subsection below will give a review of the unique Laplace MP2 algorithm.

The third distinctive feature is the developing orbital selection scheme. The orbital selection plays a crucial role in connecting the Laplace MP2 with the CC equations, constructing the small localized orbital domains based on the exchange matrix retrieved from the Hartree-Fock code. See the subsection below for more information.

The fourth innovative feature is the emphasis of this paper, which is the new Hartree-Fock exchange algorithm explicitly developed for our CiM approach. It is developed based on a new primary data type named 'the Tile structure', which will be used throughout the approach to optimize the performance of the three-index integrals. The exchange algorithm is also capable of handling the special sparse-to-sparse and sparse-to-dense matrices multiplication, replacing the existing sparse matrices multiplication algorithm. The code development will strictly follow the objective-orientated programming structure, expecting to provide a helpful library to assist any further implementation.

There are still many other innovative features in our CiM approach, however, the space is limited and only the essence will be covered in this paper.

2.2 The range-separated Coulomb potential

There are many ways to represent the orbital labels in the CiM approach. In Figure 2.2, a list of notations has been given to ensure consistency throughout the paper.

$\alpha, \beta, \gamma, \delta$	Atomic Orbital labels
i, j, k, l	Occupied Canonical Molecular Orbital labels
a, b, c, d	Virtual Canonical Molecular Orbital labels
μ, ν, ξ, o	Localized Occupied Molecular Orbital labels
x, y	Auxiliary Basis
$C_{\alpha\lambda}C_{\beta\lambda}$.	Cholesky Orbitals
p, q, r, s	General Molecular Orbital labels

Figure 2.2: **The notation of labels in the CiM approach.**

The range-separated Coulomb potential makes the calculation of the two-electron integrals efficient from small to large molecules. This is done by separating the correlation interaction into short-range and long-range (Figure 2.3) allows the algorithm to use different methods and approximations for the fastest calculation time.

The complete representation of the Coulomb potential energy is presented in Figure 2.4.

$$
\begin{aligned}
(\alpha\beta|\gamma\delta)_{2range} = & \sum_{xy}(\alpha\beta|x)_{sr} M_{xy}^{-1}(y|\gamma\delta)_{sr} \\
& + \sum_{\mathbf{g}}(\alpha\beta|\mathbf{g})\Theta(\mathbf{R}_{12})\eta(\mathbf{g})(\gamma\delta|\mathbf{g})^{*} \\
& + \sum_{mn}(\alpha\beta|m)f^{mn}(\mathbf{R}_{12})\bar{\Theta}(\mathbf{R}_{12})(\gamma\delta|n)
\end{aligned}
$$

Figure 2.4: **The range-separated Coulomb potential.**[10]

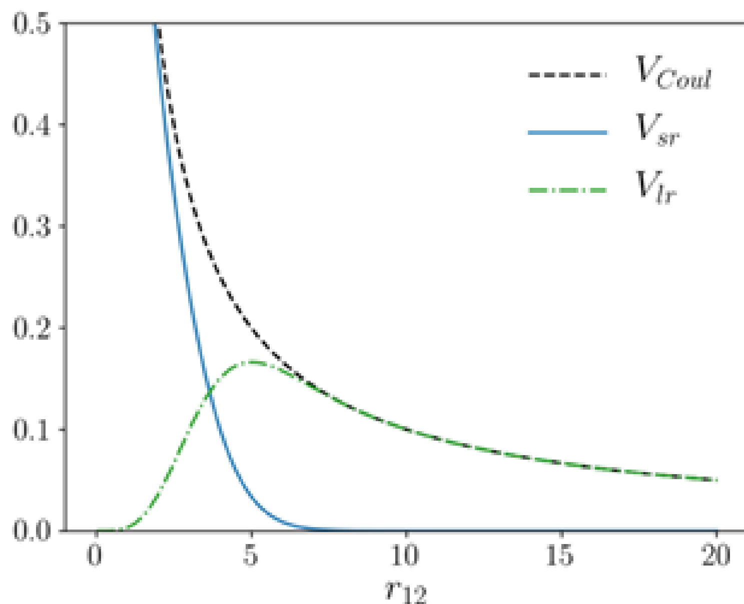

Figure 2.3: **The Coulomb potential partitioned into short- and long-range potentials.**[10]

In the short-range potential, conventional density fitting methods are used due to the limited computational cost. On the long-range side, two methods are used to optimize the accuracy. The Fourier transformation can handle the medium-range to long-range potentials, whereas the Cartesian multipole expansion is used when two electrons far apart. A threshold is implemented in the method to switch between the Fourier transformation and the multipole expansion, which can further improve the accuracy.

The short-range potential is worth mentioning since it is the major contributor to the exchange contribution, and it handles the most extensive calculation throughout the method due to the close distance between two electrons.

A shortened version of the algorithm for the short-range potential has been given in Figure 2.5. From running the cProfile tool, it has been shown that the most time-consuming step in this calculation of the two-electron three-index integral, which occurs multiple times when batching through the local orbitals μ. In the previously running version of the code, the atomic orbitals A are manually partitioned and batched using imperative programming commands. In addition, evidence shows that the performance can be erratic if the system size keeps increasing, therefore prohibiting the expected linear scaling timing. This problem motivates the desire to create a new data type capable of handling the three-index integral.

for M=1 in nM **do** ▷ Batch of local orbitals μ
  for $A \in M$ **do** ▷ Defined by $max(L_{\alpha_A,\mu}) > T_{density}$
    Obtain the Integral Batch $(\alpha_A\beta|y)$ ▷ $B \in A, Y \in AB$
    $(\mu\beta|y) += \sum_{\alpha_A}(\alpha_A\beta|y)L_{\alpha_A\mu}$
  $I_K(\mu\beta,x) += \sum_{y}(\mu\beta|y)m_{y,x}$
  $K_{\beta\gamma} += \sum_{\mu k} I_K(\mu\beta,x)I_K(\mu\gamma,x)$ ▷ Exchange Contribution

Figure 2.5: **Shortened algorithm for the short-range Density Fitting Fock contribution.**[10]

2.3 The Laplace MP2 method

A new Laplace MP2 method based on the range-separated Coulomb potential is implemented by Dr. Ondrej Demel [4], providing a more accurate and efficient version of the general second-order Møller-Plesset perturbation (MP2) method in the first place. It is also expected to be used in our CiM approach to generate low-level correlation energies for the entire system. The long-range correlation energy is adequately described by MP2 and the short-range contributions will be treated by CCSD(T) CiM calculations. The method begins by using the localized molecular orbitals (LMOs) obtained from the exchange matrice in the range-separated Coulomb potential. Later, the Laplace transformation was applied to remove the complicated denominator in the closed-shell restricted MP2 energy equation. The derivation can be tedious, but the results are awarding. The final MP2 energy can be written as Figure 2.6.

$$E(MP2) = \sum_{\tau} w_{\tau} \sum_{ijab} \sum_{\alpha\beta\gamma\delta} \{2(\alpha\beta|\gamma\delta) - (\alpha\delta|\gamma\beta)\}(\alpha\beta|\gamma\delta)_{\tau}$$

Figure 2.6: **The final representation of the MP2 energy.** This equation is obtained undergoing a Laplace transformation and then the use of the resolution of identity [4].

The Laplace MP2 has an excellent performance in long-range calculations, especially calculating the Van der Waals forces for long-range. However, inside the calculation, the algorithm can run into the same problem of the range-separated potential since they both contain extensive three-index integrals. Here in Figure 2.7, we extract a piece of the three-index integral that comes from the final representation of the MP2 energy equation.

$$
\begin{aligned}
(\underline{\mu}\alpha|q) &= \sum_{\kappa} G_{\mu,\kappa}(\tau/2,-)(\kappa\alpha|q) \\
&= \sum_{\kappa,\beta} G_{\mu,\kappa}(\tau/2,-)C_{\kappa}^{\beta}(\beta\alpha|q) \\
&\equiv \sum_{\beta} G_{\mu,\beta}(\tau/2,-)(\beta\alpha|q)
\end{aligned}
$$

Figure 2.7: **A piece of the three-index integral inside the calculation of the MP2 energy.** The label q represents the auxiliary basis x retrieved from the density fitting. It could also be replaced with the label g if this is a Fourier transformation term depending on the AO labels [4].

A closer look at the problem indicates the trouble comes inside the current routine of the three-center integral, whereas the tensor entity is too large to store in the DRAM and has to be stored on disk. Furthermore, the other intermediate quantities are calculated using a particular sparse matrix format, which slows down the overall efficiency. This problem again motivates the need for a new data type that can handle the three-index integral.

2.4 The orbital selection scheme in the CiM approach

The orbital selection scheme is the essence of the cluster-in-molecule (CiM) approach, which uses the 'divide and conquer' method. This method will select localized orbitals for large molecules in batches to create input for the final coupled cluster steps, and this could be completed in a parallel manner. During the CiM calculation, the orbital selection method will be called iteratively, therefore making the method efficient is crucial to keep the linear scaling performance.

Inside the proposed orbital selection scheme algorithm, there are two most common subroutines. The first is the pivoted Cholesky procedure, while the second is the Lowdin orthonormalization. The algorithm begins with constructing the exchange matrix K retrieved from the range-separated potential method. Then the pivoted Cholesky can be used to screen out the necessary localized molecules. (Figure 2.8) In the first step, the construction of the exchange matrix appears in a very familiar, three-index integral fashion. During this step, a large number of matrix multiplications would occur, and many of the

1. Construct the exchange matrix $K_{\alpha\beta} = \sum_{i \in P} (\alpha i|\beta i)$

2. Perform a pivoted sparse Cholesky decomposition of the projection matrix $\mathbf{Q}$ and use Löwdin orthonormalization to define a localized orthonormal basis $q_{\alpha p}$. This matrix is similar as before, but now it is a sparse matrix, and it represents a localized basis. This is the third scheme to obtain $\mathbf{q}$ discussed before in the general considerations.

3. Transform $\mathbf{K}$ to the orthonormal orbital space of interest $\tilde{\mathbf{K}} = \mathbf{q}^T \mathbf{K}\, \mathbf{q}$

4. Use a pivoted sparse Cholesky decomposition of the matrix $\tilde{\mathbf{K}}$ to obtain Cholesky vectors $\tilde{X}_{pk}$ above a threshold η.

5. Transform the kept eigenvectors to the AO basis $\tilde{X}_{\alpha k} = \sum_p q_{\alpha p} \tilde{X}_{pk}$

6. Define $\mathbf{M} = \tilde{\mathbf{X}}^T \mathbf{S} \tilde{\mathbf{X}}$, and using Löwdin orthogonalization define the final domain orbitals as $\mathbf{X} = \tilde{\mathbf{X}} \mathbf{M}^{-1/2}$

Figure 2.8: **The proposed orbital selection scheme algorithm.** Provided by the Nooijen group.

intermediate quantities are sparse matrices. Moreover, these sparse matrices tend to have a partially dense form (Figure 2.10). Designing efficient multiplication strategies for such operations with iterative nature is crucial, as it can contribute significantly to the overall CiM performance.

Besides the three-index integrals, a special sparse matrix multiplication algorithm can be helpful when running the pivoted Cholesky decomposition. This will help determine new features in the proposed new data type, which will be used among the CiM routine.

2.5 The new data type with the Hartree-Fock exchange algorithm

As mentioned in the above sections, it is crucial to find a solution to the problems preventing the current CiM approach from achieving linear scaling. The research process starts from analyzing the code structure, using profiling tools like cProfile. The result in Figure 2.9 is retrieved from running the range-separated potential method, where the number of function calls and their cumulative time could indicate which step is most time-consuming.

```
30          311083 function calls (295753 primitive calls) in 3.478 seconds
31
32     Ordered by: internal time
33
34        ncalls  tottime  percall  cumtime  percall filename:lineno(function)
35             1    0.727    0.727    0.727    0.727 linalg.py:1336(eigh)
36   23015/13249    0.668    0.000    2.069    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
37             1    0.539    0.539    0.539    0.539 linalg.py:476(inv)
38          3145    0.474    0.000    0.773    0.000 prescreen.py:212(extij)
39             7    0.193    0.028    0.193    0.028 {built-in method gc.collect}
40          6290    0.154    0.000    0.170    0.000 prescreen.py:243(expi)
41            10    0.094    0.009    0.095    0.009 moleintor.py:758(make_cintopt)
42          9317    0.057    0.000    0.104    0.000 linalg.py:2363(norm)
43          3880    0.055    0.000    0.061    0.000 fourier.py:234(cart)
44             8    0.054    0.007    0.057    0.007 moleintor.py:421(getints2c)
45             1    0.039    0.039    2.982    2.982 prescreen.py:9(prescreen)
46         31702    0.034    0.000    0.034    0.000 {built-in method numpy.zeros}
47          3236    0.030    0.000    0.030    0.000 {method 'reduce' of 'numpy.ufunc' objects}
48       5529/21    0.026    0.000    0.065    0.003 arrayprint.py:751(recurser)
49             1    0.026    0.026    0.026    0.026 {built-in method scipy.sparse._sparsetools.coo_tocsr}
50             1    0.022    0.022    0.100    0.100 fourier.py:34(getKpts)
51             1    0.021    0.021    0.021    0.021 {method 'nonzero' of 'numpy.ndarray' objects}
52         17211    0.019    0.000    0.019    0.000 {built-in method numpy.asarray}
53             1    0.013    0.013    0.045    0.045 coo.py:126(__init__)
54             1    0.010    0.010    0.010    0.010 molObj.py:182(buildMaps)
55            16    0.010    0.001    0.010    0.001 {method 'split' of 're.Pattern' objects}
56          1016    0.009    0.000    0.011    0.000 arrayprint.py:980(__call__)
57          3207    0.009    0.000    0.037    0.000 fromnumeric.py:69(_wrapreduction)
58          4984    0.008    0.000    0.011    0.000 arrayprint.py:701(_extendLine)
59   55090/55075    0.008    0.000    0.008    0.000 {built-in method builtins.len}
```

Figure 2.9: **The cProfile result from running the straight-chain alkane $C_{20}H_{42}$ using the range-separated potential method.** Except for the first few lines of the matrix multiplication method, the function called '*getints*2*c*' at line 44 takes up exponentially more time than other methods. The '*getints*2*c*' is a function that calculates the two-electron integrals using the LibCint library, from which the majority of the computational cost is from the three-index integrals. [10, 23].

As mentioned above in the range-separated potential section, the LibCint library is not designed and optimized for handling large matrix multiplication. Therefore, the atomic orbitals have been manually partitioned into many sub-matrices named 'tiles' using an imperative programming fashion to prevent the calculation from running out of memory space. This is a typical linear algebra technique when handling large matrix multiplications. However, the size of the matrix used in the calculation is highly dependent on the system size. Furthermore, due to the nature of the imperative programming, the thresholds and other parameters can not be adjusted during the calculation, which causes the overall erratic performance and inconsistent timings.

Aside from the range-separated potential method, the same problem occurs in the

Laplace MP2 development and the orbital selection scheme. They can be using different quantities during the calculation, for example, the localized molecular orbitals (LMOs) and the atomic orbitals (AOs) for the potential calculations; the auxiliary basis used by the Laplace MP2 calculation, or the Cholesky index obtained during the orbital selection. Therefore, it is necessary to create a new data type designed and optimized for explicitly handling the Hartree-Fock exchange algorithm to aid the three-index integral performance.

2.6 The sparse-to-sparse and sparse-to-dense matrices multiplication

Before the broad overview of the Hartree-Fock exchange algorithm is given, we would like to provide another handy tool that could further increase the efficiency of the overall CiM approach. The sparse matrix multiplication has been a long-existing problem in Computer Science, and there are many available open-source libraries capable of handling sparse matrix multiplication. However, these libraries are mostly not adequate to fulfill the requirement of running multiplication in the fastest way. The reason behind this is largely related to the unique structure, which we called the 'partially dense matrix', that applies to most density matrices and AO integrals in quantum chemistry. Along with developing the range-separated potential code, we discovered that inside a sparse matrix, there might be a small block of submatrix that is relatively dense compared to the rest of the submatrices. For example, Figure 2.10 is a partially dense matrix with a dense block forming a 'tile center'. In regular calculation routine, this matrix is considered as a sparse matrix and the sparse matrix multiplication algorithm has been applied throughout the calculation. However, this is considered inefficient since the sparse part will not contribute much to the calculation and can be ignored.

In the previous code, both sparse and dense matrices were considered sparse matrices and sparse matrix multiplication algorithms are used throughout the method. During this process, the conversions between dense matrices and sparse matrices take up a lot of processing power. Here, we propose a special sparse-to-sparse and sparse-to-dense matrices multiplication algorithm capable of handling the multiplication between these two different types of matrices and embedded in our 'tile' structure in the three-index integral algorithms.

$$\begin{vmatrix} s & s & s & s & s & s \\ s & D & D & s & s & s \\ s & D & D & s & s & s \\ s & s & s & s & s & s \\ s & s & s & s & s & s \\ s & s & s & s & s & s \end{vmatrix}$$

Figure 2.10: **A partially dense matrix representation.** The 's' represents a sparse submatrix, while the 'D' represents a dense submatrix.

2.7 The efficient three-index integral implementation

An efficient three-index integral implementation starts with the functional programming design. The goal of the implementation is to create a compact data class that can be used to replace every three-index integral subroutine in the CiM approach. Based on the 'divide and conquer' idea, we will use linear algebra and a customized sparse matrix multiplication algorithm to partition three-index integral or intermediate as well as many blocked-sparse matrices, which are being calculated in the general two-electron three-index integral fashion. There will be other works in transitioning the remaining imperative programming to functional programming, which could help achieve the linear scaling performance for the overall approach.

Since the space is limited, we propose the expecting features from the new data structure to have these features:

1. Able to control the tile size and batching properties in different routines to prevent data over-size from happening.

2. Combined with the customized sparse matrix multiplication algorithm, the algorithm is capable of doing self-screening, self datatype conversion with controllable thresholds

3. Able to control the size of the batch depending on the type of the calculation (two index matrices or 3-index tensors)

4. The algorithm should try to eliminate global thresholds and parameters as much as possible, determine necessary numbers in the local scopes as much as possible to accelerate the calculation further

5. Able to remove near-zero matrix elements using controllable thresholds

Implementing the three-index integral algorithm requires a detailed step-by-step plan, including multiple stages of testing, before it is finally implemented into the final CiM approach. The final work is expected to optimize the overall efficiency achieving the ultimate goal, running CiM in linear scaling performance with respect to the system size.

Chapter 3

The Tile structure

Tile, is a noun used to describe the most fundamental building unit for any enormous constructions. Given the same name, the Tile structure and the associated datatype library have a similar purpose, which is becoming the basic building block inside any complicated and extensive electronic structure calculations. When developing the CiM approach, we realized that the existing computational chemistry packages are neither adequate in functionality, nor satisfactory in the performance of the data handling. This chapter will begin by explaining the call in need of the new datatype, then followed by the designing mindset and a useful superstructure embedded in the Tile structure library named the 'Tile Master.' As complicated as the datatype goes, selected key features, such as the automated data type conversion, the fast heuristic matrix screening method and the efficient self-pruning method, will be covered. The Tile structure has been implemented into the PySCF program to further demonstrate the advantages of the sparse-dense matrix multiplication algorithm and also provide benchmarking results validates the efficiency of the bottom-up built data structure.

The Tile structure is a highly independent data structure that could be used by the Tile Master, which is our answer to the problem of manipulations involving the massive and time-consuming three-center integral. The Tile Master is able to partition large tensor or matrix objects that are impossible to fit in the DRAM, and feed the segmentations once at a time for the Tile structure to run the actual calculations. Hence, the Tile structure has to prove its capability to handle any matrix-matrix multiplication in the most efficient way.

3.1 Background information

To begin with, we need to define the problem. When running the previous Range separated Columb potential algorithms, we noticed that the timings from running the same calculation routines under the same environment are inconsistent. On the other hand, the performance of the algorithm is expected to be better than conventional routines, and the algorithm is expected to achieve linear scaling for large systems [10]. However, the results are not promising during our testing phase. A detailed investigation needs to be conducted to discover problems that cause the program to slow down.

The 'cProfile', is one of the profiling tools from the Python library. The profile tool, is often used to retrieve information and statistics inside a function call, which includes how often a subroutine has been called and how long does a subroutine take to complete [26]. To be specific, in figure 3.1, an example of the profile will contain number of calls *ncalls*, total run time *tottime*, run time per call *percall*, accumulate time *cumtime*, and accumulate time per call *percall*.

```
30          311083 function calls (295753 primitive calls) in 3.478 seconds
31
32      Ordered by: internal time
33
34      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
35           1    0.727    0.727    0.727    0.727 linalg.py:1336(eigh)
36 23015/13249    0.668    0.000    2.069    0.000 {built-in method numpy.core._multiarray_umath.implement_array_function}
37           1    0.539    0.539    0.539    0.539 linalg.py:476(inv)
38        3145    0.474    0.000    0.773    0.000 prescreen.py:212(extij)
39           7    0.193    0.028    0.193    0.028 {built-in method gc.collect}
40        6290    0.154    0.000    0.170    0.000 prescreen.py:243(expi)
41          10    0.094    0.009    0.095    0.009 moleintor.py:758(make_cintopt)
42        9317    0.057    0.000    0.104    0.000 linalg.py:2363(norm)
43        3880    0.055    0.000    0.061    0.000 fourier.py:234(cart)
44           8    0.054    0.007    0.057    0.007 moleintor.py:421(getints2c)
45           1    0.039    0.039    2.982    2.982 prescreen.py:9(prescreen)
46       31702    0.034    0.000    0.034    0.000 {built-in method numpy.zeros}
47        3236    0.030    0.000    0.030    0.000 {method 'reduce' of 'numpy.ufunc' objects}
48     5529/21    0.026    0.000    0.065    0.003 arrayprint.py:751(recurser)
49           1    0.026    0.026    0.026    0.026 {built-in method scipy.sparse._sparsetools.coo_tocsr}
50           1    0.022    0.022    0.100    0.100 fourier.py:34(getKpts)
51           1    0.021    0.021    0.021    0.021 {method 'nonzero' of 'numpy.ndarray' objects}
52       17211    0.019    0.000    0.019    0.000 {built-in method numpy.asarray}
53           1    0.013    0.013    0.045    0.045 coo.py:126(__init__)
54           1    0.010    0.010    0.010    0.010 molObj.py:182(buildMaps)
55          16    0.010    0.001    0.010    0.001 {method 'split' of 're.Pattern' objects}
56        1016    0.009    0.000    0.011    0.000 arrayprint.py:980(__call__)
57        3207    0.009    0.000    0.037    0.000 fromnumeric.py:69(_wrapreduction)
58        4984    0.008    0.000    0.011    0.000 arrayprint.py:701(_extendLine)
59 55090/55075    0.008    0.000    0.008    0.000 {built-in method builtins.len}
```

Figure 3.1: **An example of the profile tool provided by the Python library.** [10, 23]

After running the profiling tool under different scenario settings, an entry caught our attention. The 'prescreen.py' takes up a majority portion of the runtime. Inside this python script, we discovered that this subroutine is directly linked to a few functions that run heavy matrix multiplication operations and matrix tensor multiplications from the two-electron integrals. Moreover, another noticeable entry is the calculation of the exchange matrix J and K. The process also involves handling the full density matrix and the three-index tensor.

3.1.1 How two-electron integrals become a problem

If we take a step away from the statistic of the profiling, and take a point of view from the mathematical aspect, it's not hard to discover why the program struggles to run the four center integrals. Two types of integrals are commonly used inside the Hartree-fock calculation routine, they are one-eletron integrals and two-electron integrals. Start from the one-electron integrals, they are depend on the one-electron operator $\hat{O}_1$, and could be used in the calculation of the kinetic energy $\hat{T}_e$ and the potential energy $\hat{V_{Ne}}$. Similarly, the two-eletron integrals are depend on the two-eletron operator $\hat{O}_2$ and could be used in the calculation of the electron repulsion energy. Instead of doing the whole derivative, here we will give the final representation of the integrals to make things easier for analysis.

$$\int \Psi_i^*(\overrightarrow{\chi_1})(-\frac{1}{2}\nabla_1^2 - \sum_n \frac{z_n}{r_{n,1}})\Psi_j(\overrightarrow{\chi_1})dV_1 \tag{3.1}$$

Equation 3.1 shows the one-electron integrals, they are considered to be cheap since their time efficiency is $O(n^2)$, where n is the number of the electrons. However, things will get complicated when we are taking two-electrons into our consideration.

$$\iint \frac{\Psi_i^*(\overrightarrow{\chi_1})\Psi_j(\overrightarrow{\chi_1})\Psi_k^*(\overrightarrow{\chi_2})\Psi_l(\overrightarrow{\chi_2})}{|\overrightarrow{\chi_1} - \overrightarrow{\chi_2}|}dV_1 dV_2 \tag{3.2}$$

$$\langle ij|kl\rangle \tag{3.3}$$

Equation 3.2 shows the two-electron integrals, they can also be written in a more simple term shown in the equation 3.3. As the number of variables increases, the effort to calculate the integral will get expotentially bigger. The time complexity on a two-eletron integral will be $O(n^4)$, where $n = N_{AO}^4$.

In general quantum chemistry approaches, a conversion will take place to the four-center integrals into three-center integrals before the calculation. However, it will still be quite troublesome to handle the two-electron integrals as another problem quickly arises. As the system gets bigger, the number of electrons will get larger as well. The overall size of the two-electron integrals quickly exceed the size of a reasonable amount of DRAM in the system.

In order to solve this problem, there are two options to choose from at this point. The conventional methods will find a place to store the pre-calculated two-electrons integrals, which is the Hard disk drive (HDD). This can result in slow writing and reading speed due to the physical limitation of the HDD compared to the DRAM. An HDD can have a sequential read and write speed of 100 Mb/s, while the DRAM can have a sequential read and write speed of 10 Gb/s, which is almost 100 times faster than the HDD in every aspect like random throughput, latency and so on. However, the integral can only be calculated once and there are many data compression options available, only at a huge cost of time running the calculations. Unfortunately, as the size of the two eletron integrals grows exponentially, the HDD will eventually run out of space and the problem still exists.

Another approach is called the 'integral direct method', the most significant difference compared to the conventional method is that the two-electron integrals are only calculated if needed. This will result in the integral being calculated multiple times than it needs to be, but there will be no limitation on the system size since the size of the batch portion for the two-electron integrals can be adjusted accordingly to the size of the DRAM. This will benefit the overall performance since everything is stored in the DRAM instead of the HDD. However, it will require more routines on the calculation and a more careful design as well.

3.1.2 How common is the two-electron integrals problem

Since the problem has been defined, we can simplify the question by creating a notation for the two-electron three-center integral in equation 3.4.

$$(\alpha\beta|\chi) \tag{3.4}$$

The handling and the processing of the three-center integral is one of the most crucial steps over multiple procedures inside our CiM approach. Unfortunately, the tensor object is too large to fit in the DRAM, and it would be very challenging to complete a multiplication with any matrix. The problem was first found by investigating the Compact sparse

Coulomb integrals project from Mike Lecours [10], causing the program to slow down. Later, we discovered similar problems in the Laplace MP2 using Range separated coulomb potential project from Ondrej Demel [4]. In the final representation of the MP2 energy derived from the Laplace transformation and the resolution of identity, can be simplified into many three-center integrals terms.

$$\begin{aligned}(\underline{\mu}\alpha|q) &= \sum_{\kappa} G_{\mu,\kappa}(\tau/2,-)(\kappa\alpha|q) \\ &= \sum_{\kappa,\beta} G_{\mu,\kappa}(\tau/2,-)C_{\kappa}^{\beta}(\beta\alpha|q) \\ &\equiv \sum_{\beta} G_{\mu,\beta}(\tau/2,-)(\beta\alpha|q)\end{aligned}$$

Figure 3.2: **The repersentation of the Laplace MP2 energy solution.** The simplification steps will transform the two-electron four-center integrals into multiple three-center integrals for faster calculation time. Where q could be the auxiliary basis x retrieved from the density fitting, or g from the Fourier transform depending on the AO labels [4].

$$(\mu\beta|y) \mathrel{+}= \sum_{\alpha_A} (\alpha_A\beta|y) L_{\alpha_A\mu} \tag{3.5}$$

If we compared the last step from figure 3.2, where the three-center integral is $(\beta\alpha|q)$, to part of the exchange algorithm from the Compact sparse Coulomb integrals shown in equation 3.5, we could immediately notice that the first two terms are the same and only the third term is different.

However, the problem continues to exist in other part of our CiM approach. In the later development of our CiM approach, during the orbital selection procedure, the algorithm constructs the exchange matrix $K_{\alpha\beta} = (\alpha\beta|i)$ as its first step. Where i could be the auxiliary basis or the Cholesky index retrieved from the pivoted Cholesky algorithm from the orbital selection procedure. The same first two terms are found inside the three-center integral from constructing the exchange matrix.

From table 3.1, we summarized all possible inputs for a two-electron three-center integral. At this moment, we realized that there are many projects and programs that are currently suffering from the inefficient integral algorithm. Solving the problem becomes the number one goal in this project.

Input source	**Notation**
Localized molecular orbitials(LMOs)	μ
Atomic orbitals(AO)	α, β
Auxiliary basis(aux)	x, y
Cholesky Index / Meteric inputs(m)	z

Table 3.1: **Possible inputs for the two-electron three-center integrals.**

3.2 Designing mindset

After understanding the problem, the investigation begins by analyzing the current running algorithms. In our CiM approach, all projects are written in Python with the extensive use of the PySCF package. The PySCF, also known as the Python-based Simulations of Chemistry Framework, provides many useful tools and libraries that are being used inside the CiM approach. Among many libraries, one particular library named 'Libcint' is in charge of solving the two-electron integrals. In the previous section, two functions named 'int2e3c' and 'int2e4c' have been found to be the most time-consuming step in the calculation. They are provided from the 'Libcint' library in Python [24, 23].

The 'Libcint' library, written in C++, is capable of handling the calculation of most Gaussian integrals in the conventional way. This means that as the system increase, the calculation may exceed the DRAM limitation and cause the problem. From the Compact sparse Coulomb integrals project, an algorithm in figure 3.3 is made to divide the AO basis into small batches and calculate the two-electron integral on the go to achieve the 'integral direct' method [10].

However, the current algorithm can only batch through one of the items in the three-center integrals. What's more, the number of batches is given manually before the calculation starts, which requires further inputs from the user to find a proper value to optimize the algorithm.

3.2.1 The sparse matrix and the matrix multiplication algorithm

The most encountered data type during the electron structure calculation is the matrix. Usually, a matrix is a two-dimensional array filled with numbers. In Quantum Chemistry, one of the most common usages of the matrices is the density matrix, which is a positive semi-definite Hermitian matrix, that is used regularly to represent quantum states and store information. A matrix can undergo many different arithmetic operations with other

1: $MapA = False\ \forall A$ ▷ Vector to keep track of which A's have been summed into $I_2(x)$
2: **for** M=1 in nM **do** ▷ Batch of local orbitals μ
3: **for** $A \in M$ **do** ▷ Defined by $max(L_{\alpha_A,\mu}) > T_{density}$
4: Obtain the Integral Batch $(\alpha_A\beta|y)$ ▷ $B \in A, Y \in AB$
5: $(\mu\beta|y) += \sum_{\alpha_A}(\alpha_A\beta|y)L_{\alpha_A\mu}$
6: **if** $MapA \neq True$ **then** ▷ Intermediate used for J, only do this once
7: $I_1(\alpha_A,\beta,x) += \sum_{y}(\alpha_A\beta|y)M^{-1}_{xy}$
8: $I_2(x) += \sum_{\alpha_A\beta} I_1(\alpha_A,\beta,x)D_{\alpha_A\beta}$
9: $MapA$ at $A = True$
10: **end if**
11: **end for**
12: $I_K(\mu\beta,x) += \sum_{y}(\mu\beta|y)m_{y,x}$
13: $K_{\beta\gamma} += \sum_{\mu k} I_K(\mu\beta,x)I_K(\mu\gamma,x)$ ▷ Exchange Contribution
14: **end for**
15: **for** A **do**
16: Obtain the Integral Batch $(\alpha_A\beta|x)$ ▷ $B \in A, X \in AB$
17: $J_{\alpha_A\beta} = \sum_{x}(\alpha_A\beta|x)I2(x)$ ▷ Direct Contribution
18: **end for**

Figure 3.3: **The algorithm for the short range density fit calculation.** From line 4 and 5, the algorithm divides the α into batches of A and calculate the corresponding two-eletron integral on the go [10].

matrices, such as addition and subtraction from two matrices with the same shape. Multiplications and divisions will also work on matrices that are applicable under the rules of linear algebra. The divisions between matrices are always converted to multiplying the inversion of the matrices; therefore, we will be only considering the multiplication. Compared to the multiplications and divisions, additions and subtractions take much less effort to compute, and they are not usually the problem.

For two matrices A and B that have a size of $n * n$, the time complexity for addition, as shown in equation 3.6, is $O(n^2)$. On the other hand, the time complexity for multiplication, as shown in equation 3.7, is $O(n^3)$.

$$(A+B)_{ij} = A_{ij} + B_{ij} \tag{3.6}$$

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{3.7}$$

Another significant property observed from the data used inside the electron structure calculation is that the numbers inside the matrices are very small and close to zero. There are also many zeros inside the matrix; to be specific, the number of zeros exceeds the number of non-zero elements. A matrix with most of its elements zero has the name of 'sparse matrix'. During the electron structure calculation, the presence of the sparse matrix is ubiquitous. It is also prevalent in many other scientific computing problems like Finite element problems (FEP).

$$\begin{vmatrix} 0.1 & 0.05 & 0 & 0 & 0 & 0 & 0 \\ 0.05 & 0.1 & 0.05 & 0 & 0 & 0 & 0 \\ 0 & 0.05 & 0.1 & 0.05 & 0 & 0 & 0 \\ 0 & 0 & 0.05 & 0.1 & 0.05 & 0 & 0 \\ 0 & 0 & 0 & 0.05 & 0.1 & 0.05 & 0 \\ 0 & 0 & 0 & 0 & 0.05 & 0.1 & 0.05 \\ 0 & 0 & 0 & 0 & 0 & 0.05 & 0.1 \end{vmatrix} \tag{3.8}$$

A sample sparse matrix is given in figure 3.8, there are 19 non-zero elements and 30 zero elements. Given by definition, the sparsity of a sparse matrix can be determined by equation 3.9.

$$Sparsity = \frac{Number\ of\ zero\ elements}{Total\ number\ of\ matrix\ elements} \tag{3.9}$$

Another significant value to describe a sparse matrix is the density of the matrix, and it is very similar to the sparsity.

$$Density = \frac{Number\ of\ non\text{-}zero\ elements}{Total\ number\ of\ matrix\ elements} = 1 - Sparsity \tag{3.10}$$

Therefore, we could determine the sparsity and the density of the sample sparse matrix as shown in figure 3.8 are 61.2% and 38.8%. As sparsity increases, more zeros are stored

inside the matrix and there are fewer elements that are non-zero. On the contrary, matrices that have most of the elements non-zero are named dense matrices. Usually, the matrix multiplication algorithms are designed for dense matrices rather than sparse matrices. Based on the fact that most elements inside sparse matrices are zero, specialized algorithms can be designed to improve the efficiency of sparse matrix multiplication and can be used to save more space in the memory.

3.2.2 Storing the sparse matrix

It is essential to store as few zeros as possible to save space during the computation. Many different formats have been developed to store sparse matrices, however the performance of accessing and modifying the elements are quite different for each different storing format. In this project, we will focus on three crucial sparse matrix formats that are intensively used in the project, they are the Coordinate list format (COO), the CSR and the BSR.

The most straightforward format to store a matrix in sparse format is to use the Coordinate list format (COO) format. The COO will store each non-zero element into three different pieces of information: the row index, the column index and the data itself. Here in figure 3.4, an example of storing a sparse matrix in COO format is given. Using this format, a full matrix can be represented with three arrays of numbers. For example, the non-zero element located in $(1, 1)$ is 2, therefore its row and column indices are recorded in the corresponding position in the row array and the column array.

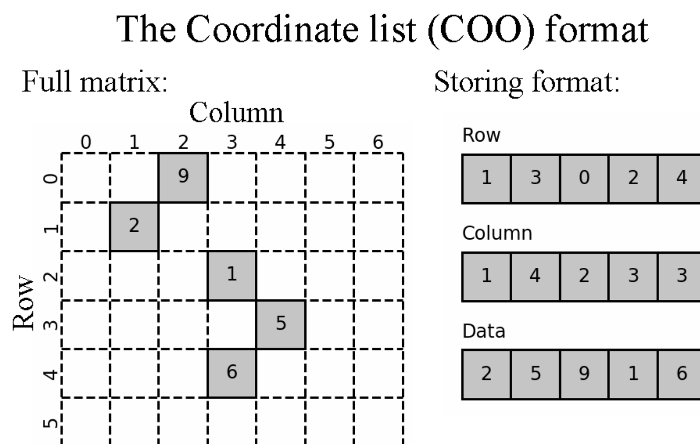

Figure 3.4: **The Coordinate list format (COO) for storing sparse matrices.**

However, there are advantages and disadvantages to use the COO format. Starting with the advantages, all the zero elements are discarded and no space is wasted during the storage. It would also be more space-efficient when the sparsity is high. Another characteristic of the COO format is that there is no order in storing elements. This could be a double-edged sword since the operation to create a new sparse matrix or add elements will cost little time in $O(1)$. At the same time, it would not be efficient when doing a reading or searching operation. Because there is no order in storing the elements, it would require a complete search inside the three arrays to look up one single element. This will lead to a considerable disadvantage when the program is trying to read a specific column or row of data.

Another downside of using the COO format is that in order to store one single element from the matrix, two other indices have to be created and stored as well. There is a problem that too many duplicated indices have been stored in the row and column array, and it could be solved by introducing another sparse matrix format called the Compressed sparse row format (CSR) format.

The CSR format uses three arrays to store information; they are the index pointers, the indices and the data itself. The index pointers record the row information using the number of non-zero elements. The position inside the index pointers represents the row number in the full matrix, and its value represents the total number of non-zero elements starting from the beginning to this current row.

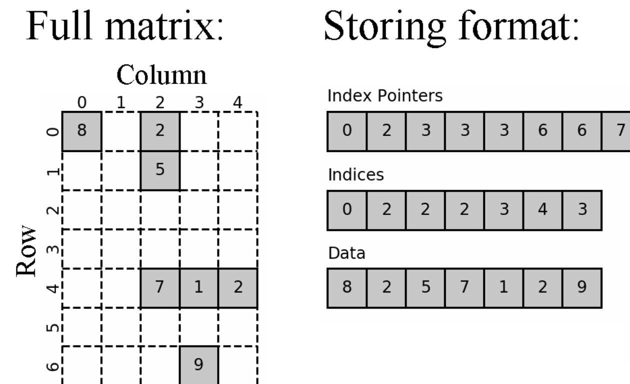

Figure 3.5: **The Compressed sparse row format (CSR) for storing sparse matrices.**

In order to retrieve the number of non-zero elements, the adjacent position inside the index pointers has to be accessed. For example, if the program wants to know how many elements are recorded in the first row, the program will access positions 0 and 1 of the index pointers. Then, the value of position 1 subtracted by the value of position 0 indicates the total number of non-zero elements in the first row, which is 2. By looking at the value in position 0, the program can know where to access the other two arrays to retrieve the column information and the data. The indices array and the data array have the same length, and a single non-zero element is stored in the same corresponding position among them. For example, using the value in position 0 from the index pointers, the program can locate the first element in the first row by accessing the "0" position in the indices array and the data array, which is an 8 in position $(0, 0)$ in the full matrix.

In this way, all the non-zero elements inside the full matrix are stored from left to right and top to bottom, which will provide huge benefits in data access. Instead of searching for the corresponding index from the COO format, the CSR can quickly locate the row index and work on a local search within a relatively small range using the number of non-zero elements. However, inserting an element into the CSR format can be really difficult and inefficient. There will be massive operations to update the index pointer array, and it is usually not encouraged to do so.

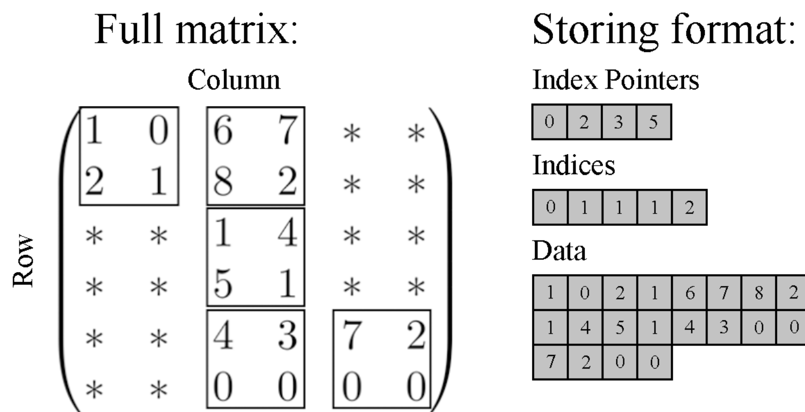

Figure 3.6: **The Block compressed sparse row format (BSR) for storing sparse matrices.**

The Block compressed sparse row format (BSR) is another sparse matrix format that

is very similar to the CSR format, the BSR store block matrices instead of individual elements like the CSR. One of the constraints for the BSR is that the dimensions of the block matrices have to be the same, and they must evenly divide the overall dimension of the full matrix. For example, in figure 3.6, the block size is $(2, 2)$, and the overall dimension of the full matrix is $(6, 6)$. The BSR will require an additional parameter to store the block size, and it can not be modified.

3.3 The Tile structure

The Tile structure, or the "Tile", is a fundamental datatype developed to provide automated data type conversion and efficient sparse matrix multiplication aiming to solve the three-index integral problem. The Tile structure, developed using the Object-oriented programming (OOP), also provides a new heuristic matrix screening function that is made available to external uses outside the library, which is one of the most critical fundamental subroutines in the later Tile Master structure. Additionally, the Tile structure has efficient self-pruning methods inside the data structure, which could remove matrix elements smaller than a given threshold in the most efficient way.

Currently, most of our CiM routine is developed based on the PySCF package. The PySCF provides many useful functions like setting up the geometries and the mean-field function; however, it will store every intermediate quantity using the NumPy ndarray from the NumPy library. This means that every quantity is stored in a dense matrix format, and there is no sparse matrix multiplication algorithm implemented. The advantages of using the sparse matrix data type has been discussed in the previous chapter 3.2.1, which could save more space in memory and reduce the total number of arithmetic operations in the calculation, thus resulting in a faster calculation. Previous attempts have been made to use the sparse matrix data type from the SciPy Sparse library; however, the improvements are quite limited due to the limited implementations in an imperative fashion and the incorrect multiplication algorithm used between dense and sparse entities.

Here, quantitative comparisons will be given to provide essential proof of concept and reveal the true potential of using sparse matrix multiplication algorithms. The standard testing procedures and the benchmark environment will also be introduced in the following section.

3.3.1 Proof of concept

In conventional calculation routines, few or no measurements are implemented to choose the most efficient data type for intermediate steps. This problem is caused by knowing insufficient data information in the calculation. To retrieve more detailed information like the sparsity and blocked structure, one will need to put tremendous effort into an imperative implementation. In our earlier implementation of the CiM approach, the COO and the CSR matrix format has been used to replace the full density matrix for a faster calculation. However, the drawback of this technique is evident, that the efficiency is highly dependent on the sparsity of the density matrix. As different basis sets are applied, the sparsity of the density matrix (and other intermedia matrices) can vary vastly. This will introduce inconsistency to the performance as it requires extra effort to convert dense matrices into sparse matrices and apply sparse matrix multiplication algorithms to them.

Matrix size	Sparsity	Dense time	Sparse time	Conversion time	Sparse total	Sparse efficiency
100	0.95	0.0065	0.00012	0.00043	0.00055	1193%
200	0.95	0.0070	0.00012	0.00026	0.00076	929%
300	0.95	0.0068	0.00070	0.0016	0.0023	292%
500	0.95	0.0081	0.0028	0.0038	0.0066	123%
1000	0.95	0.017	0.015	0.015	0.031	56%
5000	0.95	0.65	0.82	0.39	1.21	54%
10000	0.95	4.2961	6.1193	1.5764	7.6957	56%

Table 3.2: **Performance comparsion between the dense matrix multiplication and the sparse matrix multiplication algorithms. Part 1.** The dense algorithm is the dot operation from the NumPy library. The sparse algorithm is the CSR matrix multiplication algorithm from the SciPy Sparse library.

How much faster is the sparse matrix multiplication than the dense matrix multiplication? To start with, we will generate some random sparse matrices with specified sparsities in different sizes. Then, the performance of different multiplication algorithms will be compared by recording their runtime under the same environment. In Table 3.2, sparse matrices that have a sparsity of 95% in various sizes have been tested for sparse efficiency from a specified CSR sparse matrix multiplication algorithm. For matrices that have sizes under 300 by 300, the CSR sparse multiplication algorithms are very efficient. However, when the matrix size increases, the conversion time starts to slow down the total runtime of the sparse multiplication algorithm. When the matrix size exceeds 1000 by

1000, the sparse efficiency stables around 50% even under the same sparsity. The reason behind this could be complicated, as the calculation may soon run out of space using the memory. Many other studies focused on solving large-scale sparse matrix multiplication algorithms, called the Sparse general matrix-matrix multiplication (spGEMM) problems. However, in our CiM approach, we will keep the matrix size well under 500 since we want to keep the tensor calculation solely inside the DRAM.

However, the proof of concept does not end here. Many researchers believe that replacing the dense matrix with the CSR or COO sparse matrix format and using the NumPy dot operation on them would provide some efficiency. They believe that without knowing any information from the matrix, converting the matrices that they 'believe' to be sparse into sparse matrices is a good idea. This soon turns out to be the other way around as our testing continues. In table 3.3, the sparse efficiency drops below 100% quickly as the sparsity for each matrix is lowered to 50%. Matrices with sizes greater than 300 will suffer huge efficiency loss using the CSR matrix multiplication algorithm. If the sparsity keeps increasing, the turning point of the 100% will also get smaller. To sum up, the CSR sparse matrix multiplication works well on matrices with high sparsity. The exact value will be determined later in the chapter.

Matrix size	Sparsity	Dense time	Sparse time	Conversion time	Sparse total	Sparse efficiency
100	0.50	0.0097	0.00046	0.00050	0.00096	1011%
200	0.50	0.0092	0.0027	0.0015	0.0042	222%
300	0.50	0.010	0.0085	0.0029	0.011	90%
500	0.50	0.0088	0.038	0.0078	0.045	19%
1000	0.50	0.013	0.30	0.029	0.33	4%
200	0.15	0.0027	0.0068	0.0015	0.0084	32%
200	0.30	0.0019	0.0048	0.0015	0.0063	30%

Table 3.3: **Performance comparsion between the dense matrix multiplication and the sparse matrix multiplication algorithms. Part 2.** The dense algorithm is the dot operation from the NumPy library. The sparse algorithm is the CSR matrix multiplication algorithm from the SciPy Sparse library.

In order to have a more comprehensive and quantitative comparison, we will use two random matrices with controlled sparsity ranging between 0 to 1. Also, we will record the time that it takes to convert the dense matrices into sparse matrices, the time for using the sparse matrix multiplication algorithm, and the time for converting the result back to the dense format. On the other hand, the time that it takes to run a basic NumPy

dot product with two dense matrices will be recorded and used as a reference. Finally, the relative efficiency shown in the graph is derived using the total time of the NumPy dot routine divided by the total time of the CSR sparse multiplication algorithm. If the relative efficiency is greater than 1, then it means that converting these two dense matrices into CSR format is faster than calculating them directly using the dense format. On the contrary, if the relative efficiency falls below 1, the efforts to convert dense matrices into sparse matrices are not worthwhile.

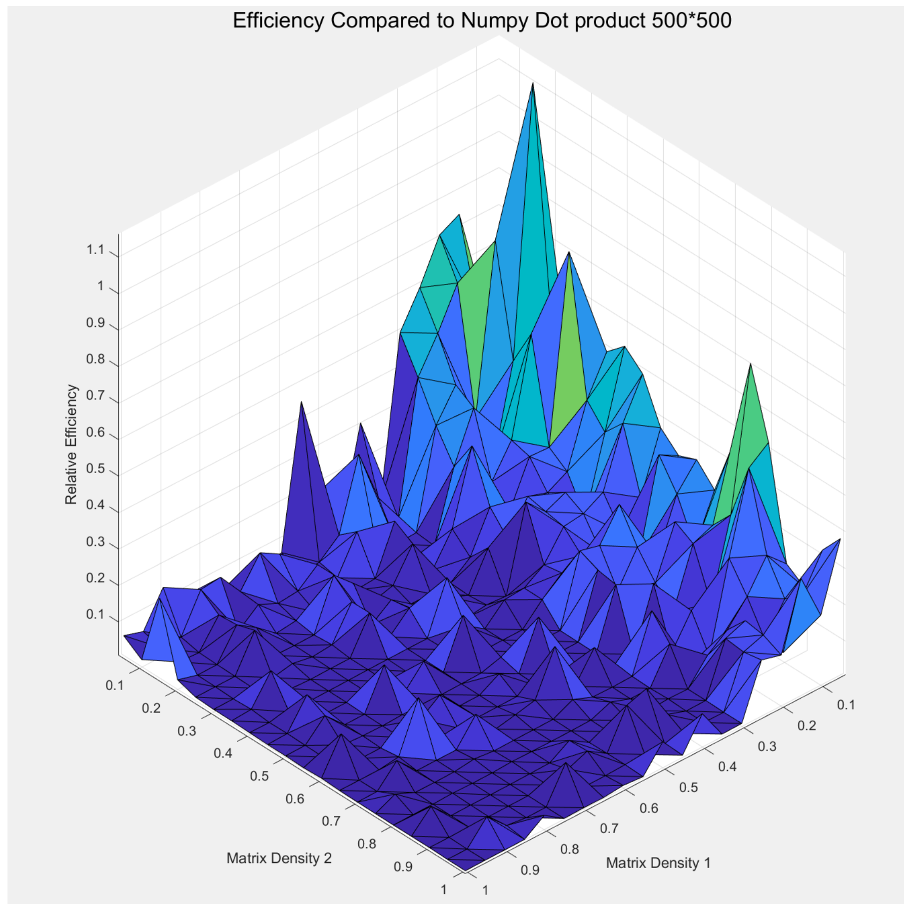

Figure 3.7: **The Relative Efficiency of the CSR routine compared to the NumPy dot routine in size of 500 by 500 matrices.** The higher the peak, the more efficient the algorithm routine is. The matrix density is the opposite of the matrix sparsity. The average performance of the CSR algorithm is measured at 12.3%.

In figure 3.7, two randomly generated matrices in size of 500 by 500 are used to illustrate

the CSR routine. Four hundred different cases are collected and coloured in the figure to represent a generalized scenario. However, only a few cases exceed the 100% efficiency and most of the cases are below even 50% efficiency. To evaluate the overall effectiveness of the CSR sparse multiplication algorithm, we can calculate the average based on all the testing cases since they are uniformly distributed. The average performance of the CSR algorithm is measured at 12.3%. Another noticeable detail is that the surface is not smooth, indicating that the CSR algorithm contains some inconsistency in the timing. This could be due to the memory allocating during the calculation, which is a hardware-related problem that is difficult to deal with.

In our previous CiM program, the use of the sparse matrix multiplication algorithm is limited to only the CSR routine. Inside the previous code, many procecures convert an intermediate matrix into the COO or the CSR sparse format, and use the CSR routine to calculate the results. Later, the results are converted into dense format matrices again for further calculations. It is unsurprising to see such a bad performance in the outcome when replacing dense matrices directly with sparse matrices. Also, the inconsistency from the CSR routine also answers the inconsistent timing issue from the previous CiM approach.

3.3.2 A closer look at the data

In order to design and optimize a sparse matrix multiplication algorithm for electronic structure calculations, we need to take a closer look at the intermediate data first. During the calculation of the CiM approach, the matrices we collected do not always have a sparsity randomly distributed between 0 and 1. For example, the density matrix of a C_4H_6 carbon chain in 3-21g has a sparsity of 37.8%. On the other hand, the overlap matrix of the same C_4H_6 carbon chain in the same basis set has a sparsity of 43.4%. From Table 3.4 and Table 3.5, we are able to notice that as the molecule gets bigger or the basis size increases, the sparsity of the density matrix and the overlap matrix will get higher.

	C_4H_6	C_8H_{10}	$C_{12}H_{14}$	$C_{16}H_{18}$	$C_{20}H_{22}$
3-21g	0.38	0.29	0.31	0.35	0.40
ccpv-dz	0.47	0.42	0.44	0.48	0.52
ccpv-tz	0.55	0.56	0.61	0.66	0.69

Table 3.4: **Sparsity of the density matrix of selected conjugated alkene chains in different basis sets.** Values inside the matrix below 10^{-5} are pruned.

In practice, retrieving the actual intermediate matrices from the calculation turns out to be inefficient and time-consuming. To develop and optimize the Tile structure, we

	C_4H_6	C_8H_{10}	$C_{12}H_{14}$	$C_{16}H_{18}$	$C_{20}H_{22}$
3-21g	0.43	0.60	0.70	0.76	0.80
ccpv-dz	0.53	0.65	0.73	0.78	0.82
ccpv-tz	0.60	0.72	0.79	0.83	0.86

Table 3.5: **Sparsity of the overlap matrix of selected conjugated alkene chains in different basis sets.** Values inside the matrix below 10^{-5} are pruned.

need to be able to simulate the intermediate matrices that could represent a wide range of molecules. Certain properties differ the density matrix from a randomly generated matrix; for example, the diagonal elements of a density matrix are always dense. Furthermore, since the density matrices are Hermitian, their eigenvalues should be real and they will also be non-negative. Another example of intermediate matrices is that they are usually partially dense and have a blocked sparse structure due to the atoms' unique geometry. For two atoms that are further apart, their two-electron integrals are going to be more sparse.

Matrix size	From 100 to 1000
Matrix sparsity	From 0 to 1
NumPy.random generator mode	default_rng
Target matrix data type	numpy.ndarray
Matrix element absolute value upper limit	1 to 10
Is matrix Hermitian?	Yes or No
Is matrix symmetric?	Yes or No
Does matrix contain negetive values?	Yes or No
Does matrix contain large diagonals?	Yes or No
Is matrix blocked sparse?	Yes or No

Table 3.6: **Rules for simulating the actual intermediate matrices from the calculation.**

To create a testing matrix, it has to follow the set of rules listed in Table 3.6 to ensure it is similar to the actual matrix retrieved from the calculation. Also, it has to have some randomness to ensure the testing cases are adequate for evaluations. After the testing matrix is generated, the test suite needs to ensure it is the same matrix that undergoes different matrix multiplication algorithms. Otherwise, the relative efficiency would become nonsense.

3.3.3 The automated data type conversion

In conventional imperative programming methods, the data type conversion can only happen if only there is specific instruction made to follow. As mentioned above, this will require a tremendous amount of work to implement. However, this problem could be dealt with using the OOP method. Since the most common data type used inside most calculations is the numpy.ndarray, it would be better to create a new data type that can automatically convert the data type to take advantage of the sparse matrix format only when needed. The benefits of using the sparse matrix format have been discussed in chapter 3.2: saving space in the DRAM and speeding up the calculation. However, this would bring up another question: Which sparse matrix format to use? Also, when to convert the dense matrix into the sparse matrix?

Throughout our testing phase, the data matrix tends to have a sparsity greater than 30% but lower than 80%. The sparsity can get larger as the system size increases, but we keep a relatively medium to small size system during development and benchmarking. This is due to the final application of the Tile structure will be inside the CiM approach, where we use divide-and-conquer to make sure everything fits inside the DRAM. In the early chapter, we discovered that only using the CSR multiplication algorithm would have a bad performance for anything above 30% density. It is crucial to turn to the other sparse matrix formats for a more efficient multiplication algorithm.

In the meantime, developers from Nvidia also discovered a similar finding: sparse linear algebra would not provide adequate performance to speed up calculation when the sparsity is below 95%. They concluded that algorithms are inefficient due to the 'irregular computation and scattered memory' in their article [28]. This again matches our finding in section 3.3.1, where we discovered that the timing inconsistency might come from the memory allocation. In the later part of the Nvidia article, they introduced a new method to overcome the limitations, which they named 'cuSPARSE Block-SpMM' [28].

The blocked sparse structure

To further understand the 'cuSPARSE Block-SpMM', we will have to introduce the blocked sparse structure first. The blocked sparse structure, is developed initially to solve the SpMM problem. In early testing of the CSR routine, we noticed that if one of the multiplying matrices is dense and the other is sparse, there will be a considerable performance penalty. This can be verified in figure 3.7, where the peaks are distributed around the line where two matrices' density equals. Many studies have been done to solve the SpMM

problem, as it is applicable not only in scientific computing but also in many other fields like deep learning, neural network and visual analytics.

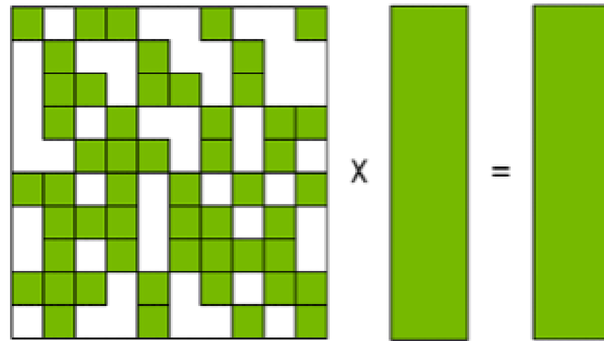

Figure 3.8: **The Sparse-matrix dense-matrix multiplication (SpMM) and the Sparse-matrix vector multiplication (SpMV) problem visualized.** [28]

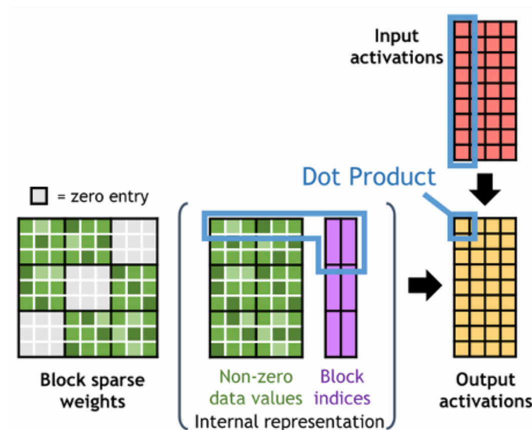

Figure 3.9: **One solution to the SpMM problem using their Blocked-ELL structure provided by the Nvidia.** By converting the sparse matrix into block-wise dense matrix will provide efficient matrix multiplication performance. [28]

The currently best solution to the SpMM problem is to convert the sparse matrix into a block-wise dense matrix structure, like the Block compressed sparse row format (BSR) or the Blocked-ELL shown in Figure 3.9. Once the block-wise dense matrix is obtained, the algorithm can proceed to perform dense matrix multiplication between the targeted blocks. In other words, using the blocked structure will divide the sparse matrix into many

blocks, and only the dense one will be calculated using the dense matrix multiplication algorithm.

In one attempted implementation from Eberhardt, they experimented with many different implementations on different system architectures like the CPU and GPU. In their case, the storing sparse matrix format is the BSR. They provided some inspirational methods which could process the BSR with the order of row-per-thread, which could speed up the calculation up to 4 times compared to the Nvidia cuSPARSE (2016) and up to 147 times faster than the Intel MKL library (2016) [5]. Their work might no longer be practical to us as the Nvidia and the Intel MKL library have been constantly updated throughout the past six years, but their experiments are still significant to our research by pointing out that the Block compressed sparse row format (BSR) is the way to go to deal with the SpMM problem.

The BSR algorithm and available external libraries

To see how effective the BSR sparse format is, we need to conduct the same testing procedure as the CSR to give a fair comparison. Here in figure 3.10, the BSR sparse algorithm has been compared to the NumPy dot algorithm using the same 500 by 500 size. At first glance, the highest peak from using BSR algorithm is over 500% efficient, whereas the highest peak from using the CSR routine is around 110%. The turning point for the BSR to fall behind 100% efficiency is around 70% sparsity, whereas the CSR has a very fast descending turning point around 95% sparsity. The average overall efficiency for the BSR routine is 14%, which is slightly higher than the 12.3% from the CSR.

Why does the BSR reaches a higher efficiency than the CSR when both matrices are very sparse? Theoretically, the CSR should require fewer steps to complete the multiplication since the CSR contains fewer number of zeros compared to the BSR format. However, this is related to the time complexity and the space complexity. An algorithm which requires fewer arithmetic operations might spend longer time allocating memories and moving data around, compared to an algorithm that could allocate memories very quickly with more arithmetic operations. However, in order to allocate memories quickly, the algorithm usually requires a lot of space. In our case, the BSR stores more zeros than the CSR and outperforms the CSR by using more space in the DRAM. Inside the Tile structure, both time complexity and the space complexity for each algorithm have to be taken into account since the calculation is tight on available DRAM space.

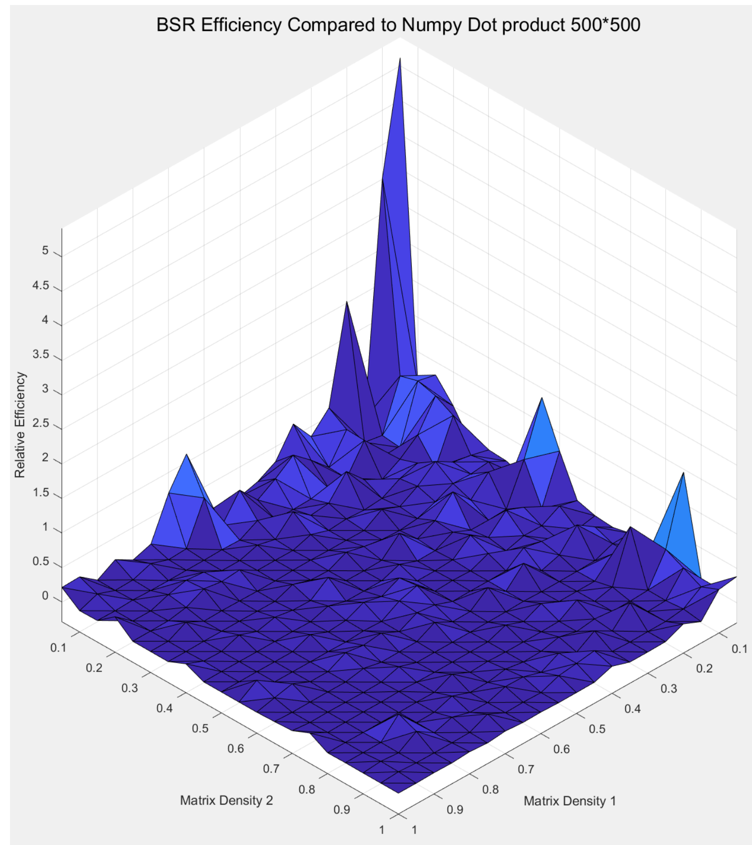

Figure 3.10: **The Relative Efficiency of the BSR routine compared to the NumPy dot routine in size of 500 by 500 matrices.** The average performance of the BSR algorithm is measured at 14.0%, higher than the 12.3% from the CSR routine. Also, the BSR routine outperforms the CSR in sparsity regions above 70%.

Thus far, the CSR and the BSR algorithms are provided from the SciPy.sparse library and they are implemented in single-thread only. This means that even their algorithms might not be much distinct from other SpMM approaches, but there is a massive difference in the output performance. To be clear, the performance is highly dependent on the computing capabilities as most of the recent spGEMM libraries emphasize the use of the multi-threading on multicore architectures, which could be either CPU or GPU.

One of the available external SpMM libraries is the Intel Math Kernel Library, also known as the Intel MKL. It has been one of the most popular kernel libraries and is often used as a standard reference for performance. Necessary implementations have been

done to enable the direct function calls from Python to the MKL library solely using cytpes and another python package named sparse-dot-mkl. At the time of writing, the tile data structure handles the sparse-sparse matrix multiplication and the sparse-dense matrix multiplication entirely on the Intel MKL library and takes full advantage of the multi-threading technique.

To further demonstrate the power of the multi-threading technique, we have run the testing procedures for the 500 by 500 matrices again using the same computing system with the Intel MKL library. From figure 3.11, an impressive 952% improvement in overall average efficiency and up to 2011% improvement in some instances from the CSR routine. At the same time, there is a 771% improvement on the BSR routine, with up to 1998% improvement in some instances.

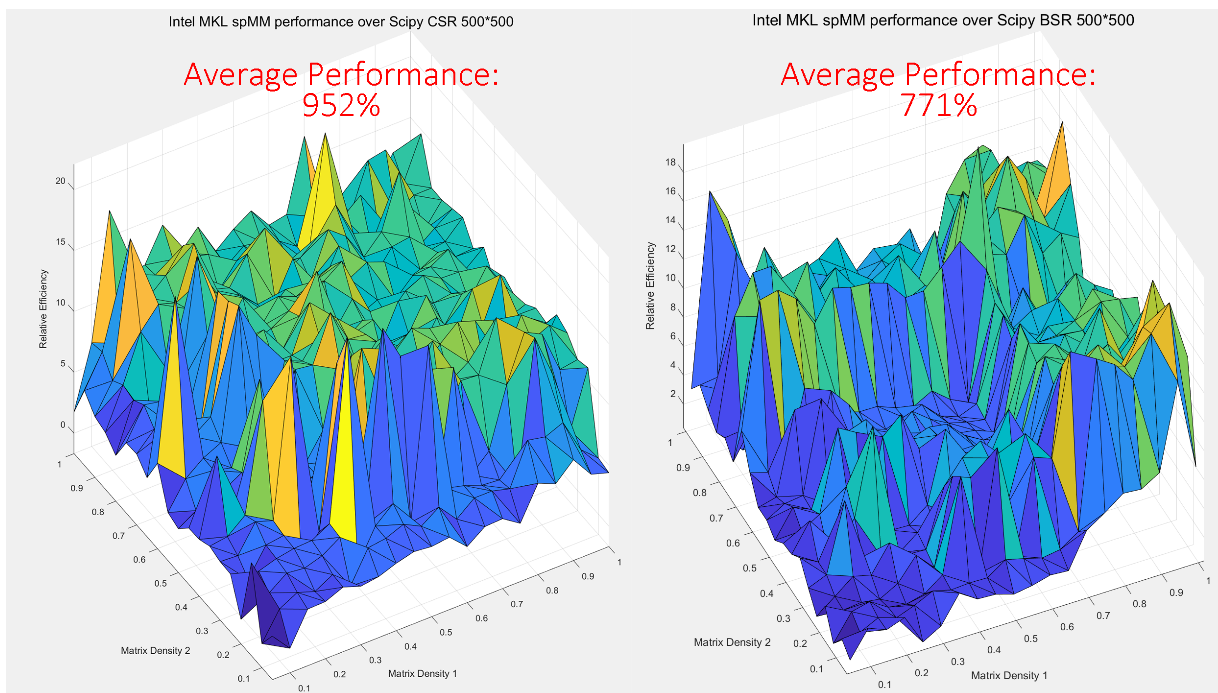

Figure 3.11: **The Relative Efficiency of the Intel MKL SpMM performance compared to the SciPy CSR and BSR routine in size of 500 by 500 matrices.** Under the same computing system, the Intel MKL provides a nearly 10 times efficient over the CSR routine and 8 times efficient over the BSR routine in the SciPy.sparse library.

Another implementation attempt is based on the GPU using the CUDA structure with the CuPy package. However, this GPU orientated implementation requires two extra steps

compared to any CPU orientated routine. In computing systems, the GPU can not read or write directly from the DRAM. One would have to imperatively copy the matrices into the GPU memory and collect the results from the GPU memory to the DRAM to able other access from the CPU. Since GPU is a highly parallelized structure, it is very optimal to divide the SpMM problem into many column-wise or row-wise subroutines, which could benefit the runtime in exponential orders and overcome the loss of the data transfer time. However, this implementation is not easily applicable inside the Tile structure as the data type is OOP.

The CuPy package is a wrapper program to the Nvidia CUDA library, where it provides accelerated tensor calculations from the cuTensor library and accelerated SpMM solvers from the cuSPARSELt library. However, the sparse matrix multiplication from the cuSPARSELt library does not currently support the calculations with the BSR matrix format. What's more, the performance of the CUDA calculation is highly dependent on the GPU devices which can not be compared to the results generated by the CPU. Hence, there will be no visualization of the performance improvement from the GPU SpMM libraries.

Auto-determination of the BSR block size from SciPy.sparse

When using the BSR data format, it is crucial to determine a proper block size to reach the maximum possible efficiency in both storage and multiplication. The most efficient block size is highly dependent on the data itself, and there is no general answer for every BSR matrix. Inside the SciPy library, a function named *estimate_blocksize* is created to determine the most efficient block size for a BSR matrix [27]. Every BSR will go through this function when it is being created to determine its proper block size.

We have no contribution to this function, but it is worth mentioning since it is crucial to the overall efficiency of the Tile structure. In short, the algorithm will eventually return a set of block size (r, c), where the number of non-zero elements of the BSR matrix is the smallest among a selection of available block sizes [27]. The detailed algorithm has been quoted in Appendix A.5, and the estimation method uses the trial and error method. Currently, the algorithm is only able to test and return a block size of $(1, 1)$, $(2, 2)$, $(3, 3)$, $(4, 4)$ or $(6, 6)$. The actual determination criteria is the fraction value shown in equation 3.11, and the default efficiency is 0.7.

$$\frac{\text{Number of non-zero elements for the matrix in a block size of } (1,1)}{\text{Number of non-zero elements for the matrix in a block size of } (r,c)} > \text{efficiency} \tag{3.11}$$

Inside the Tile structure, we wish to keep the block size relatively small and the calculation as efficient as possible. All the BSR matrices are using this auto-determination method to retrieve their best block size for storage and calculation. However, we soon encountered new problems with BSR multiplications between two BSR matrices with different block sizes. Inside the SciPy.sparse library, it is possible to multiply two BSR matrices with different block sizes, and the result BSR will keep the smaller block size between the two [27]. However, when using the external Intel MKL library, the operation quickly becomes illegal and the block size has to be manually adjusted before the BSR matrices are sent to the library.

Determination of the conversion thresholds

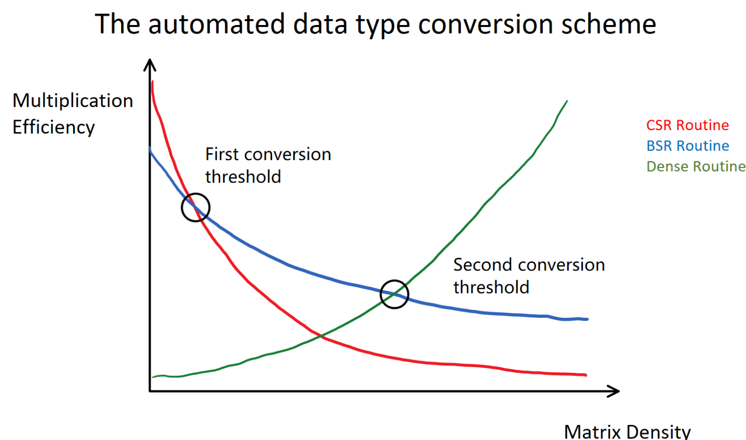

Figure 3.12: **Illustrations on the threshold determination from the automated data type conversion.** The red curve represents the efficiency of the CSR routine versus the increasing matrix density, while the blue represents the BSR routine and the green represents the dense matrix routine. The Tile structure will automatically switch between different matrix data types for the best calculation efficiency.

In the previous sections, we have discussed a few different sparse matrix storing formats and evaluated their performance and efficiency with different matrix densities. In figure 3.12, an illustration can be drawn to demonstrate the automated data type conversion scheme from the three available calculation routines. There are many contributing factors to the multiplication efficiency, and the values tend to change as the matrix size changes.

Hence, this illustration will be inexact and is qualitative only to aid in finding the two conversion thresholds.

We will separate the full range of matrix density into three regions for further discussions on the details. The first region starts from the zero density and ends where the matrix density is below the first conversion threshold, which represents the very sparse matrices. In this case, the dense of the dense routine would be inefficient, and lots of the memory space would be wasted by storing too many zeros. From our early observations, the time complexity of the BSR routine should be better than the CSR routine. However, when the matrix is very sparse, it usually worth the trade-off to sacrifice some time complexity to save more space during the calculation, as the space complexity of the CSR is much better than the BSR routine. Therefore, the CSR routine has the highest multiplication efficiency within the region, and the Tile structure will automatically convert any matrix that has a density below this first conversion threshold.

The second region lies between the first conversion threshold and the second conversion threshold. The efficiency of the CSR routine fades away quickly as the matrix density increases, and the efficiency of the dense matrix multiplication is not yet adequate to overtake the plate from the CSR routine without a significant performance penalty. Here, the Tile structure will rely on the blocked sparse structure using the latest SpMM solutions. The Block compressed sparse row format (BSR) can compress the sparse entries as blocks and perform a dense-like multiplication, giving a much higher performance over the CSR and the dense routine.

The third region is where the matrix density is above the second conversion threshold. Here the Tile structure will determine that it is inefficient to convert the dense format into any other sparse matrix format and will stick to the dense matrix multiplication algorithms.

Finding the first and the second conversion threshold can be done in a few tries using our testing environment. A recommended method is to start from the back, searching for the second conversion threshold for BSR first with the CSR routine switched off. In table 3.7, we can observe that the second conversion thresholds are around the when using different external libraries. The second conversion threshold can be determined as 0.7, meaning that any matrix with a sparsity greater than 70% will be converted to the BSR format and undergoes SpMM routines.

	Using Intel MKL library	Using Spicy.sparse library
$Threshold_{BSR} = 0.5$	180.13 %	123.68 %
$Threshold_{BSR} = 0.6$	202.38 %	159.40 %
$Threshold_{BSR} = 0.7$	238.54 %	194.50 %
$Threshold_{BSR} = 0.8$	187.47 %	175.14 %
$Threshold_{BSR} = 0.9$	215.10 %	181.87 %
BSR switched off, dense only	127.07%	127.59 %

Table 3.7: **Finding the second conversion threshold using the testing environment size of 500 by 500.** The conversion threshold is determined using the overall average efficiency of the Tile structure for better optimization with other available functions. The CSR conversion has been switched off.

	Using Intel MKL library	Using Spicy.sparse library
$Threshold_{CSR} = 0.8$	186.37 %	145.13 %
$Threshold_{CSR} = 0.9$	193.63 %	160.35 %
$Threshold_{CSR} = 0.95$	211.94 %	170.72 %
CSR switched off, BSR and dense only	238.54%	194.50 %

Table 3.8: **Finding the first conversion threshold after acquiring the** $Threshold_{BSR} = 0.7$ **using the testing environment size of 500 by 500.** The conversion threshold is determined using the overall average efficiency of the Tile structure for better optimization with other available functions.

After acquiring the first conversion threshold, the second conversion threshold for the CSR routine can also be determined using similar methods. In table 3.8, the optimal threshold for the CSR routine is determined as 0.95. This result validates the research from the Nvidia group as they found that most CSR routines work for matrix sparsity above 95% [28].

To sum up, the automated data type conversion contributes a majority of the gained efficiency of the Tile structure. With the aid of the blocked sparse structure and much help from current spGEMM, SpMM and Sparse-matrix vector multiplication (SpMV) research, we are able to provide a significant improvement using the Tile structure. In order to achieve better efficiency, much work has been done in the background to ensure the data type conversion is thread-safe and capable of running in a multi-threading fashion. Also,

using OOP enables the user to change the targeted math library to suit different calculation environments and adapt to future external library updates.

3.3.4 The fast heuristic matrix screening method

The screening method is one of the most critical features that differ the Tile structure from any other data type implementation. In order to calculate the sparsity, the number of non-zeros is always needed. The only way to get the answer from NumPy is to use the *count_nonzero* function, and from SciPy is to use the *nnz* (number of non-zeros) attributes. Both methods will provide an exact answer to the user since the functions are set to go over every single entry, which could impact the performance significantly as the size of matrices or tensors gets very high.

The Tile structure will need to retrieve the sparsity from the data to perform the automated data type conversion and benefit from the SpMM. Retrieving the sparsity from the sparse matrix formats is straightforward and efficient since there are not a lot of stored zeros, and it will take $O(1)$ for access in the CSR. On the other hand, the screening will become a problem for the dense matrix. Unless there is any external sparsity array input scheme from the Tile Master structure, the Tile structure will perform its screening method for the data input. However, the sparsity answer does not need to be exact, as accessing the entire dense array can be very costly. Currently, six sparsity estimation methods are available from the Tile structure shown in table 3.9 to help reduce the screening efforts.

$sparsityEstimationMethod$	Sparisity estimation method
1	Diagonal elements only
2	Column elements only
3	Row elements only
4	X style elements only
5	+ style elements only
6	The fast heuristic matrix screening method

Table 3.9: **Available sparsity estimation methods from the Tile structure.**

Method 1 to 5 are some commonly used matrix screening methods in linear algebra. However, even though the cost is cheap, none of the methods could yield an acceptable estimation compared to method 6. The fast heuristic matrix screening method, is developed to deal with the heterogeneous distributed data array coming from the CiM approach or any

other scientific computing program. A flowchart is provided in figure 3.13 to demonstrate the random sampling procedures inside the fast heuristic matrix screening method. The source code is also made available in appendix A.1 for a better understanding.

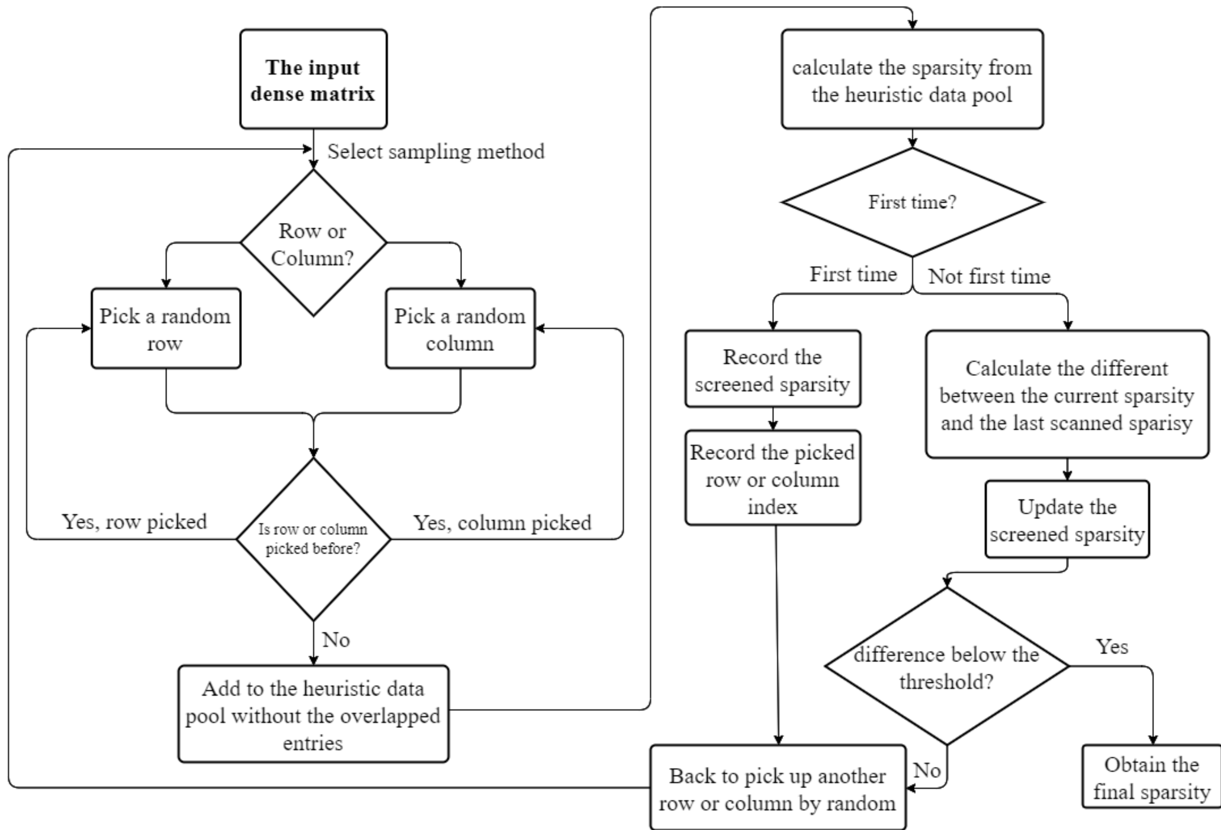

Figure 3.13: **The flowchart of the fast heuristic matrix screening method inside the Tile structure.** The random sampling method will ensure the screening algorithm obtains an accurate estimation even with heterogeneous distributed data arrays.

Inside the fast heuristic matrix screening method, a threshold is used to end the recursive random sampling. We are also able to further optimize the algorithm behaviour by adjusting the threshold to find a balance between the accuracy and the performance. Table 3.10 displays the average estimation error retrieved under different heuristic screening thresholds. However, a more accurate estimation can't be justified to be efficient only when not evaluating the overall tile structure efficiency. An inaccurate estimation will shift the first and second conversion threshold between different data types, and the final per-

formance could be influenced. Therefore, it is more appropriate to determine the heuristic screening threshold using the overall efficiency.

$heuristicEstimateThreshold$	Average estimation error	Overall efficiency of the Tile structure
0.1	±0.00962	3.279364962
0.05	±0.00928	3.202434383
0.01	±0.008866667	3.095969525
0.005	±0.00822419	3.348356564
0.001	±0.00511225	3.610377249
0.0005	±0.005525857	3.436049345
0.0001	±0.003336978	3.367356811

Table 3.10: **Determine the heuristic estimate threshold using the average estimation error and the overall efficiency of the Tile structure.** The Tile structure has the highest overall efficiency of 361% when the $threshold_{screening} = 0.001$, which means any estimation converging under 0.1% will end the heuristic process with an error rate of sparsity in ±0.51125%.

3.3.5 The efficient self-pruning method

During the CiM electronic structure calculation, it is vital to manage the use of the DRAM wisely. Using the sparse matrix format to free up the storage space from storing zeros is one attempt to reduce memory usage, avoiding the use of the HDD. Besides using the sparse matrix, another feasible and commonly used technique is to improve the overall efficiency and reduce the calculation size by removing the insignificant small matrix entries. This is called the matrix pruning method.

In our CiM approach, we have already determined a threshold to remove insignificant matrix entries below 10^{-5}. However, it is often considered a complicated task to complete since the insignificant entries would appear after each arithmetic operation. Moreover, it is nearly impossible to remember pruning after each command in the imperative coding style. We will once again benefit from our new data type, the Tile structure, by implementing the pruning operation along with the arithmetic operation. When using the Tile structure, the self-pruning procedure is operated whenever the Tile class constructor is called. There are several different scenarios for a Tile class constructor to be called; for example, when initializing a new Tile structure using any of the NumPy.ndarray, the python array or copying directly from another Tile structure. An attribute named $cutOffStatus$ inside

the Tile will keep track of the pruning operation, which can avoid executing duplicated self-pruning operations.

To be specific, the self-pruning operation is embedded inside the overloaded Tile class arithmetic operators, which accept any of the Tile structure, the NumPy.ndarray, the python array, or any sparse matrix format from the SciPy.sparse library. In addition, the Tile class arithmetic operators will also override all the NumPy arithmetic operators using an array priority rank of 15. The source code of the Tile class constructor and the overloader arithmetic operators can be found in appendix A.3 and A.4 for a better understanding.

While implementing the self-pruning method, we noticed that there is no available function that could complete the task efficiently from the SciPy.sparse library and the NumPy library. In our previous CiM procedures, sparse matrices have been converted to dense matrices before the pruning operation, and the pruned matrices are converted back to sparse matrix format to save some space. In the early sections, we already acknowledged that the conversion time between the dense and sparse matrix format is terrible when the matrix sparsity is low. Hence, the best way to carry out a prune operation is by pruning the sparse matrix format directly.

Algorithm 1 The efficient self-pruning method for the SciPy CSR format.

for $indexPointPositions$ **in** all available index pointer positions **do**
  $currentNNZ \leftarrow$ number of non-zeros from current index pointer position
  $nextNNZ \leftarrow$ number of non-zeros from the next adjacent index pointer position
  $NNZThisRow \leftarrow nextNNZ - currentNNZ$
  for $dataThisRow$ **in** $NNZThisRow$ **do**
    if absolute of the CSR data $[dataPositions + dataThisRow] > threshold$ **then**
      $newData \leftarrow newData$ + CSR data $[dataPositions + dataThisRow]$
      $newIndices \leftarrow newIndices$ + CSR indices $[dataPositions + dataThisRow]$
      $newNextNNZ \leftarrow newNextNNZ + 1$
    end if
  end for
  $dataPositions \leftarrow dataPositions + NNZThisRow$
  $newIndexPointers \leftarrow newIndexPointers + newNNZ$
end for
$newIndexPointers \leftarrow newIndexPointers + newNNZ$
CSR data $\leftarrow newData$ ▷ Update the previous CSR
CSR indices $\leftarrow newIndices$
CSR IndexPointers $\leftarrow newIndexPointers$

In algorithm 1 and in appendix A.2, we presented an efficient self-pruning algorithm by only accessing the CSR entries once. The time complexity of this algorithm is $O(n)$, where n is the total number of non-zeros elements. Compared to the previous attempt by converting back to the dense matrix, the time complexity is $O(N^2)$, where N is the matrix size and $n \ll N$ in most cases.

Algorithm 2 The efficient self-pruning method for the SciPy BSR format.

```
for indexPointPositions in all available index pointer positions do
    currentNNZ ← number of non-zeros from current index pointer position
    nextNNZ ← number of non-zeros from the next adjacent index pointer position
    NNZThisRow ← nextNNZ − currentNNZ
    for dataThisRow in NNZThisRow do
        for  each BSR data entry in the block do
            if absolute of the BSR data entry [dataPositions+dataThisRow] > threshold
then
                newData ← newData + BSR data [dataPositions + dataThisRow]
                newIndices ← newIndices + BSR indices [dataPositions+dataThisRow]
                newNextNNZ ← newNextNNZ + 1
            end if
        end for
    end for
    dataPositions ← dataPositions + NNZThisRow
    newIndexPointers ← newIndexPointers + newNNZ
end for
newIndexPointers ← newIndexPointers + newNNZ
BSR data ← newData                                  ▷ Update the previous BSR
BSR indices ← newIndices
BSR IndexPointers ← newIndexPointers
```

The efficient self-pruning method for the BSR is shown in algorithm 2 as well. Since the BSR and the CSR share a very similar data structure, the procedures are almost identical, except the data block from the BSR has to be examined one entry at a time. Being able to perform the pruning operation efficiently is critical to the overall efficiency of the Tile structure due to the fact that it is being frequently called.

3.4 Performance and potentials

The Tile structure, is designed to replace the NumPy.ndarray class with better performance on matrix multiplication and more functionalities like automated data type conversion and self-pruning. With the aid of many external libraries, the Tile structure is able to solve the SpMM problem efficiently using either CPU or GPU architecture with the ability of multi-threading. Furthermore, a specially designed heuristic matrix screening algorithm helps the Tile structure accurately estimate the sparsity from large matrices at a minimal computational cost. Implemented using the OOP method, the Tile structure can be used to speed up any matrix multiplication application with little implementation effort.

To evaluate the performance of the Tile structure in solving two-electron three index integral problems, we will run through simulated chemistry matrices made for our testing and project the final average efficiency along with the highest performance boost. The performance might differ under different computing environments, and the hardware configuration used for the benchmarking has been provided in table 3.11.

CPU	AMD Ryzen 9 5950x 16-Core Processor @ 4.50Ghz
GPU	NVIDIA GeForce RTX 3090
DRAM	32.0 GB DDR4 @ 3600 MHz
SSD	two PCIE 4.0 Solid State Drives in RAID 0

Table 3.11: **The hardware configuration used for the benchmarking.**

In table 3.12 and table 3.13, the simulated chemistry matrices ranging from 100 by 100 in size to 1000 by 1000 in size have been tested to provide a comprehensive review of the Tile structure. Starting form the table 3.12, the Tile structure provides around four times more efficient than the regular NumPy dot routine, while giving peak performance boosts over 30 times faster than the NumPy dot. However, as the matrix size increases, the efficiency of the Tile structure decays slowly to 1.9 times faster than the NumPy dot at the maximum matrix size of 1000 by 1000. For a better comparison, figure 3.14 displays the overall average performance of using the Tile structure compared to the NumPy dot routine. At the same time, figure 3.7 and figure 3.10 yield only around 10% of the overall efficiency, while the Tile structure has a 355.64% in overall efficiency. The result confirms that the Tile structure is capable of replacing the NumPy.ndarray to speed up matrix multiplications.

Input size	Overall average efficiency	Highest preformance boost
100	5.65	105.62
200	4.78	159.59
300	6.03	33.14
400	4.19	26.46
500	3.56	23.62
600	3.14	16.20
700	2.54	15.33
800	2.47	14.78
900	2.09	9.73
1000	1.90	8.48

Table 3.12: **The overall average performance of using the Tile structure under different sizes, compared to the NumPy dot routine.** The thresholds are: $threshold_{csr} = 0.95$, $threshold_{bsr} = 0.7$, $threshold_{screening} = 0.001$ and $threshold_{cutoff} = 10^{-5}$.

Input size	Overall average efficiency	Highest preformance boost
100	3.87	27.03
200	12.29	60.53
300	15.57	93.36
400	24.63	89.63
500	24.81	87.92
600	24.39	94.31
700	26.02	113.23
800	27.76	112.58
900	29.79	112.69
1000	30.93	123.15

Table 3.13: **The overall average performance of using the Tile structure under different sizes, compared to the previous CiM routine implementation.** The thresholds are: $threshold_{csr} = 0.95$, $threshold_{bsr} = 0.7$, $threshold_{screening} = 0.001$ and $threshold_{cutoff} = 10^{-5}$.

In table 3.13, the Tile structure is used to compare against the previous implementation on our CiM approach, which solely uses the SciPy CSR multiplication algorithm. The result is even more persuasive as the Tile structure reaches more than 30 times faster than the

previous attempt with the aid of external libraries. Under certain circumstances, the Tile structure can be over 120 times more efficient in matrix multiplications.

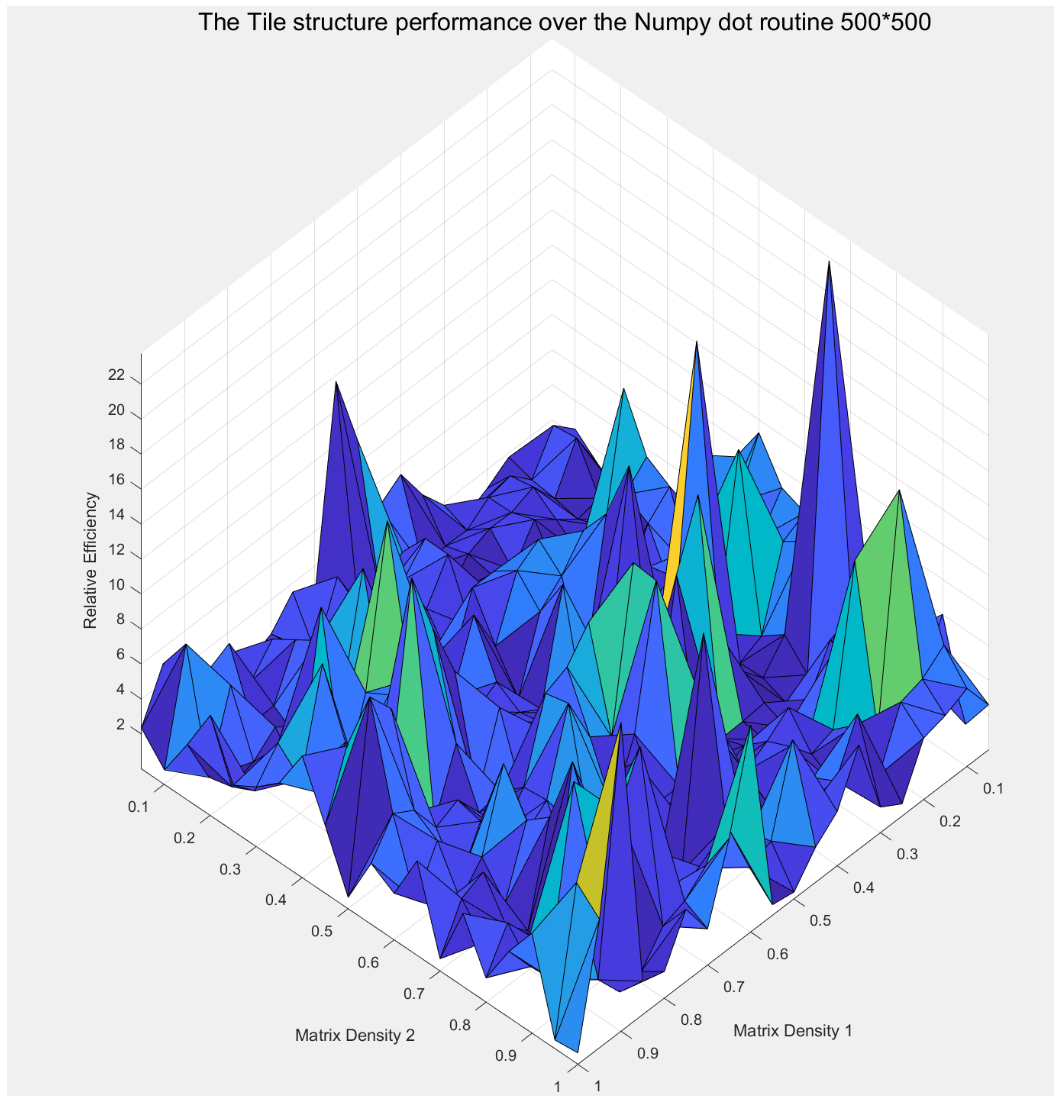

Figure 3.14: **The overall average performance of using the Tile structure in size of 500 by 500 matrices, compared to the NumPy dot routine implementation.** The thresholds are: $threshold_{csr} = 0.95$, $threshold_{bsr} = 0.7$, $threshold_{screening} = 0.001$ and $threshold_{cutoff} = 10^{-5}$. The average performance of the Tile structure is measured at 355.64%.

Since the Tile structure is capable of multi-threading, we assume that the Tile structure can be even more efficient on highly paralleled computing architectures. However, the time is limited during this project, and future work is expected for the confirmation.

The true potential of the Tile structure is far from only accelerating the matrix-matrix multiplications and saving space during the calculation. The OOP style implementation allows the Tile structure to retrieve information from other Tiles, therefore, the calculation between two or more Tiles can also be done in the most efficient way.

When used with the Tile Master, the Tile structure can focus on seeking the most efficient calculation routine, while the Tile Master can redirect the tensor objects from the two-electron three index integrals in a block-wise structure made of the Tile structures. By retrieving the data information from the Tile structures, the Tile Master can decide on a reasonable block size and monitor the performance throughout the calculation. We will cover the capabilities of the Tile Master by making using the Tile structure in the next chapter.

Chapter 4

The Tile Master

The Tile Master, a superstructure built solely for the CiM approach, is our answer to the problem of manipulations involving the massive and time-consuming three-center integral. With the most powerful tool ever, the Tile structure, we are able to go autopilot on the matrix-matrix multiplications and focus on our final destinations, solving the three-index integral problem. The most significant difference between the Tile Master and any other block-wise data structure is that the Tile Master can retrieve estimated sparsity information coming from Atomic orbital (AO) calculations and geometry analysis, before the data is even constructed.

In this chapter, we will first start with explaining what is the Tile Master and why the Tile Master is different from the Tile structure and other conventional block-wise data structure. Then, we can illustrate the capabilities of creating a three-dimensional 'sparse matrix' using the Tile Master and the Tile structure together, which is a vital data structure to use when handling tensors and three-index objects. Later, a specially designed dense and sparse calculation routine for the three-index quantities will be covered to enhance the efficiency of using the blocked sparse structure in 3-D. Finally, we will present an example of solving the three-index integral problems using the Tile Master, followed by more benchmarking and performance analysis for further validation.

At the very end, we will provide an overview of this project and its future potential in helping the CiM approach and many other scientific calculations.

4.1 Another data structure constructed using the Tile structure

What is the Tile Master? What's the difference between the Tile Master and the Tile structure? Why not use one data structure instead of two data structures? Unfortunately, there are no simple answers to these questions, and more information is needed for clarification. Hence, we will answer the questions one at a time and give necessary background information simultaneously.

The Tile Master, is a block-sparse data structure that is designed to solve the two-electron three-center integral problem using the Tile structure. The Tile Master is different from the Tile structure in its purpose and usage, but they all adopt a similar block-sparse structure to accelerate the Sparse-matrix dense-matrix multiplication (SpMM). In the previous chapter, we explained why using the blocked sparse structure can increase the calculation efficiency. Also, in Chapter 3.3.3, we also discovered that there is a block size limitation inside the SciPy library, preventing the individual block size exceeds six. Last but not least, the SciPy.sparse library cannot run three-index array multiplication in sparse format at the time of writing, so there is no solution using existing libraries. The Tile Master is explicitly created to fill in the blanks of calculating three-index quantity multiplications, and it adopts a unique block-wise structure, unlike any conventional block-wise data structure.

A typical block-wise data structure presented by Nvidia is limited to only accept in two-dimensional arrays or matrices [28], which prevents us from the direct use of the three-dimensional objects. Another significant difference between any conventional block-wise structure is the storing method and its associated multiplication algorithms. For example, in Figure 4.1, a regular block-wise structure will only store empty entries and dense matrices. However, the Tile Master can store the data block in a more dedicated way by distributing the data format handling to the Tile structure. Hence, there could be four different data formats stored in the Tile Master structure: The dense matrix, the BSR, the CSR and the empty entry.

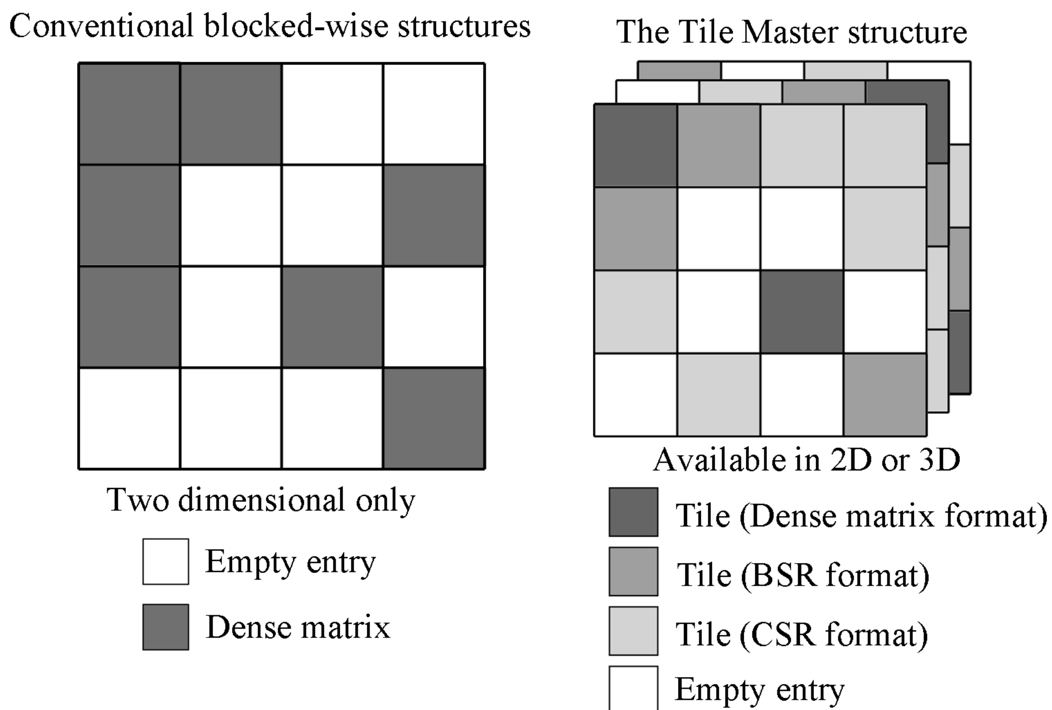

Figure 4.1: **The Tile Master compared to conventional block-wise structures.**

What's the difference between the Tile Master and the Tile structure? At the beginning stage of this project, we came across the idea of building a nested Tile structure, which allows the Tile structure to store another Tile structure as an entry inside these different sparse formats. Instead of having an array of float numbers, the nested Tile structure can have arrays of Tiles or numbers. However, this idea was later abandoned due to the complexity of the tree structure, and it was inefficient in the row or column search operations. At this point, we realized that the best solution to handle the three-index integral problem is to separate the macro-operations and the micro-operations into two different but connected data structures.

Macro-operations (Tile Master)	Micro-operations (Tile)
Determine an appropriate block-size under the current system size	Block-size for BSR is self-determined locally just for multiplication efficiency
Divide three-index quantities into matrices and pass them to Micro	Receive the input matrices from Macro and calculate the results
Choose dense or sparse multiplication strategies for blocks	Choose the most efficient matrix multiplication algorithm
Control and keep track of the block-wise multiplication	Screening matrices for sparsity to provide feedback for Macro
Sum up the final result	Remove array elements that are below the threshold

Table 4.1: **Macro-operations versus Micro-operations in the block-sparse data structure.** Macro-operations are completed by the Tile Master, and the Tile structure will provide Micro-operations supports to the Tile Master.

Conventional block-wise data structure usually handles both the Macro-operations and the Micro-operations at the same time, while the Tile Master does the Macro and the Tile structure does the Micro shown in Table 4.1. This strategy allows the low-level calculation to be separated from the high-level data flow, which has multiple advantages over the traditional arrangement. Firstly, avoiding getting caught between the low-level calculation helps the beginner-level user of this data structure free of frustration. We did not discuss this in Chapter 3, but we found in practice that the Tile structure can significantly improve the readability of the code and make programing much easy with them.

Secondly, being able to separate different levels of calculation enable the capablity of multi-threading and makes it suitable for any high-performance high-parallelization computing structure. This includes the new Intel KNL structure Xeon processors and any Graphic processing unit (GPU) using the Nvidia CUDA structures. Third but not least, using two data structures instead of one helps the future developers focus on either the Macro or the Micro side. For example, when designing a new calculation routine or translating an existing calculation, the developer only needs to modify or add one function to the Tile Master. On the other hand, if one wants to modify the lower-level calculation routines like switching the external runtime libraries, the developer will only need to modify the existing function in the Tile structure, and it will not interfere with the Tile Master.

4.2 The Tile Master

With the help of the Tile structure, the Tile Master can achieve things that could never be easily implemented before. In the following sections, selected features such as determining the variable block size using geometry characteristics, creating the three-dimensional sparse "matrix" and selecting between dense and sparse calculation routines using an external sparsity array. With all these added functions, the Tile Master is finally capable of solving the three-index integral problem and benchmark results will be given at the end of this chapter.

4.2.1 Determining the variable block size using geometry characteristics

The Tile Master can automatically determine an appropriate block size when given enough geometry characteristics in advance. The geometry information can be the number of atoms, the number of AO basis functions, or fitting basis functions, or some other dimension. This idea comes from the CiM approach and the divide-and-conquer technique. Specifically, the necessary information for the Tile Master can be retrieved when the gauge centers are assigned during the integral prescreening phase [10]. Assigning the gauge center is our way of dividing the whole molecular into subsets of atoms, and during this process the prescreening algorithm developed by Mike will determine an appropriate "grid size", which limits the number of AO inside the grid [10]. For convenience, we could retrieve the number of AO for each grid and make it the corresponding block size.

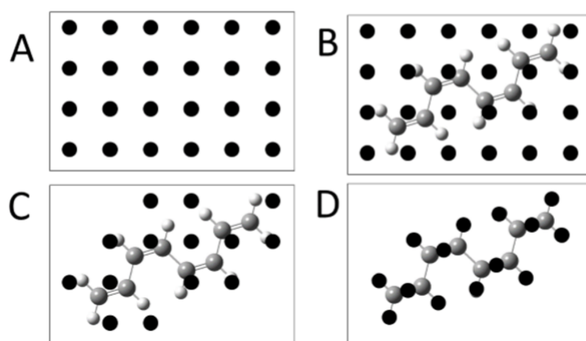

Figure 4.2: **Assigning AO gauges centers in the integral prescreening algorithm developed by Mike Lecours [10].** The Tile Master will retrieve the geometry information from this algorithm and determine the most appropriate block size.

In figure 4.2, an example of the integral prescreening algorithm is given by Mike to present the whole process of assigning gauge centers. In the code by M. Lecours gauge centers were used to calculate the final assigned gauge centers to calculate the long-range potentials, and as the centers closely follow the atoms [10]. We repurpose these gauge centers as cell centers associated with a small group of atoms, therefore the distance between the orbital pairs α, β can be easily calculated. This piece of geometry information will be used later to determine the external sparsity array that allows the Tile Master to handle sparse or dense calculation routines, which will be discussed later.

Another benefit of using the geometry information from the prescreening routine is that, after each grid has been created and subsets of AO have been determined, the CiM can only distribute the necessary AO to each subroutine of calculations. This will minimize the number of overlapping AO used in long-range potential calculations. Also, this will not be possible if only the Tile structure is used, since the Tile structure can only accept block sizes that are no more than six. In optimal cases, we will keep the number of atoms below five and the block size around 100, which is also a optimal size to input for the Tile structure.

4.2.2 Creating 3-D sparse arrays using the Tile Master

Another critical feature of the Tile Master is that we can create and store the vast three-index quantity in a sparse format. Currently, there is no three-dimensional sparse matrix support in any of the available libraries, including the NumPy and the SciPy. The only way to deal with the three-index quantity is to store them in dense arrays and perform dense multiplication algorithms. There are attempts to speed up the calculation process by arranging the memories for faster accessing time, however the major limitation is still the DRAM size. By creating a three-dimensional block-sparse structure, we could remove most of the zero in the three-index quantity and run more calculations simultaneously.

Unlike the traditional Coordinate list format (COO) way of storing the non-zero elements, the Tile Master uses a new storing format specially designed for multiplication efficiency. Figure 4.3 illustrates the mapping relationship between the three-index integral and the Tile Master storing format. The range of the auxiliary index is limited, and it would be appropriate to loop through the auxiliary first. Later, we can observe the block sparsity changes as the distance between the orbital pair α, β, and it would be best to use the block-sparse structure calculation routine in a two-dimensional way.

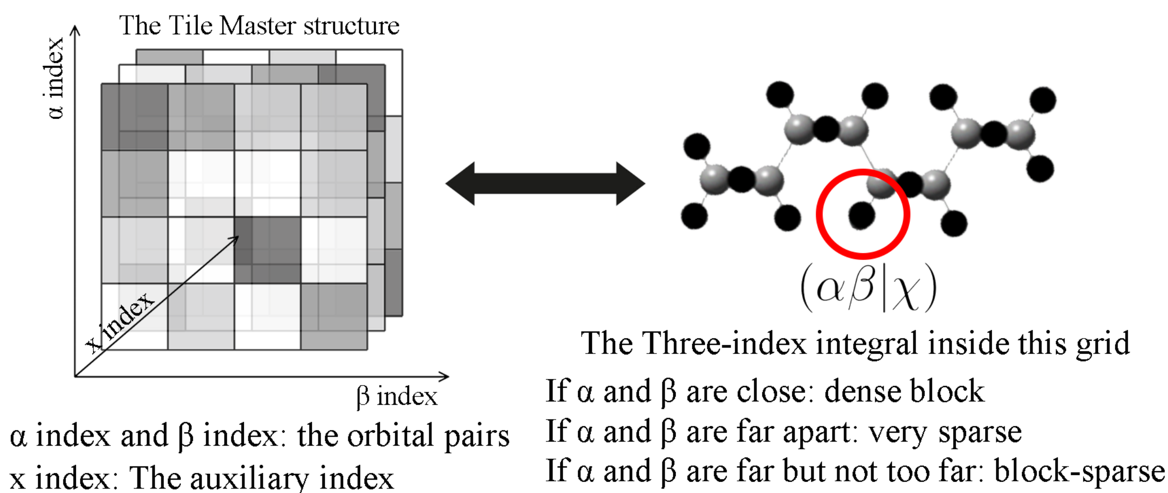

Figure 4.3: **The three-dimensional sparse quantity storing format in the Tile Master data structure.** The Tile Master uses three different indices to map the three-index integral inside each grid calculation [10].

4.2.3 Special dense and sparse calculation routines using sparsity arrays

Compared to the multiplication between two 2D matrices, there will be one more index for the three-index integral problem to deal with. The extra index is the auxiliary index, and it has a relatively limited range compared to the AO indices. This means that we could bypass the calculation when the auxiliary index gets off the threshold determined by the prescreening method [10]. What's more, we also know that if the α, β orbital pairs are separate and very far apart, their contribution to the overall matrix can be ignored. Therefore, we can further reduce the calculation effort using this approximation.

The approximation process is done by associating with the assignment of the gauge center inside the prescreening routine, and the Tile Master can retrieve necessary information from the prescreening routine using an external sparsity array. The external sparsity array contains an estimation of the sparsity in the current block position. For example, if the α and the β are very far apart and their distance beyond the threshold, the sparsity estimation for this block should be 1, which is completely empty. In this case, the Tile Master will not create any Tile in this location, and the calculation will bypass any associating operation referring to this location. On the other hand, if the orbital pair

distance is close enough to generate block-sparse matrices, the sparsity can be estimated, and the Tile Master can execute the sparse matrix multiplication routine based on the estimated sparsity without further screening. For the dense calculation routine, it will happen when the external sparsity array indicates the orbital pair is close to each other, and dense matrices will be produced. Therefore, the Tile Master can switch between the dense and the sparse calculation routine using the geometry and the grid information from the prescreening procedure. This could further increase the efficiency of the three-index integral multiplication since the self-screening procedure can be bypassed inside the Tile structure.

Dense calculation routine (stored in dense, use dense matrix dot product)	When auxiliary index is in range, and α, β are close
Sparse calculation routine (stored in block-sparse, use Tile SpMM support)	When auxiliary index in range, and α, β are not far
Skip calculation routine (does not store data, bypass any reference)	When auxiliary index not in range, or α, β are very far

Table 4.2: **The dense and sparse calculation strategy developed in the Tile Master using an external sparsity array.** The determination of the range and the threshold is inside the integral prescreening algorithm [10], which will not be covered in this project.

However, there are difficulties in designing a perfect sparsity array estimation algorithm as the sparsity of the data is highly dependent on many parameters, like the size of the basis set and the size of the grid. Future works are required to collect more three-index quantities from the actual calculation and establish the relationship between the geometry factors and the resulted sparsity behaviour. The time is limited in this project, and we only implemented the receiving functions inside the Tile Master. When an external sparsity array is given, the Tile Master will turn off the self-screening method inside the Tile structure and take over the control of the SpMM.

4.2.4 Solving three-index integral problem

Thus far, we have implemented two unique data structures, the Tile structure and the Tile Master, that are capable of handling both macro-operations and micro-operations for block-sparse data handling. We have created a new three-dimensional sparse array format and designed dense and sparse calculation routines using the sparsity arrays as the geometry information input from the integral prescreening algorithm. Now, it is time to see how we can deal with the significant and complex three-index integral problem.

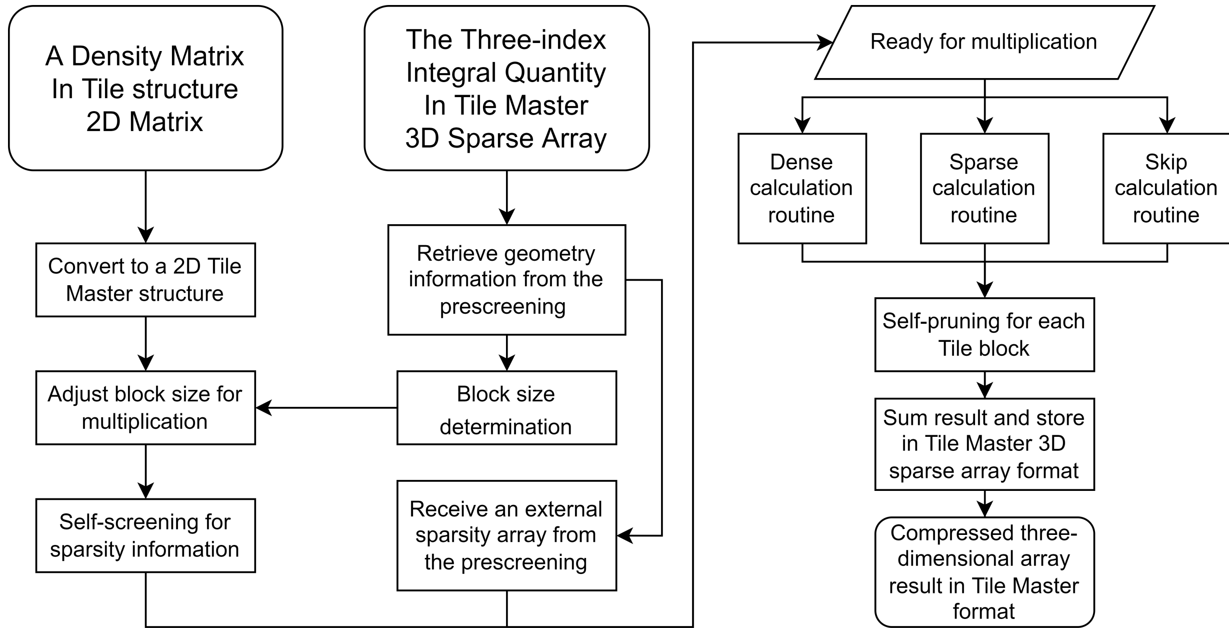

Figure 4.4: **The flowchart of solving one of the three-index integral problems (calculation of the exchange matrices) using the Tile Master and the Tile structure.** To prevent the data from exceeding the DRAM limitation, every three-dimensional quantity is stored in compressed sparse array format. During the multiplication routine, the self-screening feature inside the Tile structure has been turned off, and the sparsity information will be coming from the provided external sparsity array.

In figure 4.4, a flowchart illustrates our solution to one of the common three-index integral problems, which is the calculation of the exchange matrix. The calculation starts with the density matrix stored in the Tile structure, and the three-index quantity is stored in Tile Master format. The first step is to collect geometry information from the integral prescreening function, where the Tile Master can use them to determine the appropriate block size. After the block size is determined, we will convert the density matrix into a two-dimensional Tile Master with the same block size. The three-index quantity will also obtain a 3D external sparsity array from the prescreening as well. On the other hand, the 2D object will have to use the self-screening feature inside the Tile structure to build its own internal sparsity array.

Once both sparsity arrays are obtained, the multiplication process will begin starting by looping from the auxiliary index. Three different calculation strategies will handle the high-level multiplication using the sparsity arrays, while the low-level multiplication

between 2D matrices is still completed by the Tile structure. In the meantime, the Tile structure will no longer be doing a self-screening procedure, and all the necessary sparsity information is provided by the Tile Master. After the multiplication, the self-pruning procedure is executed before the sum happens for the final result. The final multiplication result is stored in the compressed three-dimensional array format as well.

4.3 Performance and potentials

The Tile Master can handle the three-index integral problem very efficiently using the block-sparse structure and the geometry information coming from the integral prescreen algorithm. Benefiting from the high data sparsity, the new compressed three-dimensional array format can keep the memory usage exponentially smaller than running in dense format. However, the integral prescreening algorithm is still in the development phase, and there will be much work to fully implement the Tile structure and the Tile Master into our CiM approach. To provide some benchmarking results and validate the functionality of the Tile Master, we simulated the potential external sparsity array and geometry array for conjugated alkene chains.

Three-index quantity overall size	From 1000 to 10000
Three-index quantity total number of elements	From 10^9 to 10^{12}
Block size	From 100 to 500
Tile Master 2D grid size	From 10 by 10 to 20 by 20
Overall sparsity	Above 70%
The accepted auxiliary index range	below 15%
Estimated dense calculation routine percentage	42%
Estimated block-sparse calculation routine percentage	18%
Estimated skip calculation routine percentage	40%
Maximum DRAM size	32Gb

Table 4.3: **Rules for simulating the actual three-index quantity for the calculation.**

During our testing, we pushed our Tile Master and Tile structure to the absolute limit.

In Table 4.4, a simple comparison of the memory usage between the NumPy dense array and the Tile Master validates the superior off using the compressed three-dimensional array. This format will reduce around 95% of the storage space from a three-index quantity, which will significantly increase the computing capacity of the system.

Quantity type	Memory taken using NumPy dense (Mb)	Memory taken using Tile and Tile Master (Mb)	Space efficiency
Two-dimentional matrix	3.8	3.4	1.12
Three-dimensional array	7629	477	15.99

Table 4.4: **Space efficiency when using the Tile and Tile Master.** For three-index quantities, a 1000 by 1000 by 1000 array would take up to 8Gb of memory and while the Tile Master can keep the three-index sparse compressed to 0.47Gb.

Followed by Table 4.5, the Tile structure and the Tile Master are able to complete calculations that are impossible for the NumPy dense routine. As the data size increases, the Tile Master can still handle the massive data using its block-sparse data structure. On the other hand, we stopped testing the NumPy as it would only present error messages like “Unable to allocate 59.6 GiB for an array with shape (2000, 2000, 2000) and data type float64”.

Input size	NumPy dense dot calculation time (s)	Tile Master calculation time (s)	CSR sparse multiplication routine with compressions (s)
1000 by 1000 times 1000 by 1000 by 150	89.3	0.77	4.68
1000 by 1000 times 1000 by 1000 by 1000	Memory overflowed, DNF	5.62	30.95
1500 by 1500 times 1500 by 1500 by 1500	Memory overflowed, DNF	23.99	130.09
2000 by 2000 times 2000 by 2000 by 2000	Memory overflowed, DNF	86.89	Memory overflowed, DNF

Table 4.5: **Time efficiency when using the Tile and Tile Master.** For three-index quantities, the maximum overall size for numpy is 1000 and the memory overflow happened at the maximum overall size. To illustrate the benefit from using the special dense and sparse calculation routines, a CSR calculation routine has been implemented using the compression method. The CSR represents the previous implementation attempt.

Chapter 5

Summary

To sum up, the Tile Master can handle the complex three-index integral problems with the help of the Tile structure and the external sparsity array. We started our comparison with the NumPy and SciPy libraries, however eventually, the Tile structure and the Tile Master surpassed them both in memory arrangement and calculation efficiency.

In this project, we tried to combine two completely different subjects together, the Sparse-matrix dense-matrix multiplication (SpMM) problem from computer science and the electron structure calculation from chemistry. Despite the differences, there are many similar problems like dealing with the sparsity during the matrix multiplication, finding an efficient sparse matrix compression format and utilizing the sparse matrix multiplication algorithm. During the project, we tested many models and solutions from other publications. However, many of them did not meet our expectations and therefore did not appear in this project.

We hope that the Tile structure and the Tile Master can be viewed as useful tools inside any scientific calculations. Not only can they solve the complex two-electron three-index integral problems, but also the sparse-block structure can be beneficial to any large sparse matrix multiplication applications like artificial intelligence or image processing.

As the computing power grows rapidly, there are new products coming out every day, changing the way of scientific computing step by step. We found huge potential in solving the matrix-matrix multiplication using the GPU, and this could become a reality for computational chemistry in the near future. The bottleneck that stops the implementation of the GPU is the memory transfer speed between the memory and the GPU memory. The anticipation is that this problem could be easily dealt with using newer hardware.

There is much work that still needs to be done to finish our Cluster-in-molecule approach (CiM). In the meantime, the Tile structure and the Tile Master will become powerful tools during future development. It will take some time to develop and upgrade the previous code in our CiM program since the program is complicated and massive. However, this will provide much more opportunity for future developers and make it easier to maintain in the long run.

References

[1] Jiří Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *The Journal of Chemical Physics*, 45(11):4256–4266, 1966.

[2] Aron J Cohen, Paula Mori-Sánchez, and Weitao Yang. Challenges for density functional theory. *Chemical reviews*, 112(1):289–320, 2012.

[3] Dieter Cremer. Møller–plesset perturbation theory: from small molecule methods to methods for thousands of atoms. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(4):509–530, 2011.

[4] Ondřej Demel, Michael J Lecours, Richard Habrovskỳ, and Marcel Nooijen. Toward laplace mp2 method using range separated coulomb potential and orbital selective virtuals. *The Journal of Chemical Physics*, 155(15):154104, 2021.

[5] Ryan Eberhardt and Mark Hoemmen. Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures. pages 663–672, 2016.

[6] RC Fortenberry and TD Crawford. Annual reports in computational chemistry, 2011.

[7] Krassimir Georgiev and Zahari Zlatev. Implementation of sparse matrix algorithms in an advection–diffusion–chemistry module. *Journal of computational and applied mathematics*, 236(3):342–353, 2011.

[8] Yang Guo, Ute Becker, and Frank Neese. Comparison and combination of "direct" and fragment based local correlation methods: Cluster in molecules and domain based local pair natural orbital perturbation and coupled cluster theories. *The Journal of Chemical Physics*, 148(12):124117, 2018.

[9] Kasper Kristensen, Ida-Marie Høyvik, Branislav Jansik, Poul Jørgensen, Thomas Kjærgaard, Simen Reine, and Jacek Jakowski. Mp2 energy and density for large molecular systems with internal error control using the divide-expand-consolidate scheme. *Physical Chemistry Chemical Physics*, 14(45):15706–15714, 2012.

[10] Michael Lecours. Compact sparse coulomb integrals using a range-separated potential. 2021.

[11] Shuhua Li, Jing Ma, and Yuansheng Jiang. Linear scaling local correlation approach for solving the coupled cluster equations of large systems. *Journal of computational chemistry*, 23(2):237–244, 2002.

[12] Shuhua Li, Jun Shen, Wei Li, and Yuansheng Jiang. An efficient implementation of the "cluster-in-molecule" approach for local electron correlation calculations. *The Journal of chemical physics*, 125(7):074109, 2006.

[13] Wei Li, Zhigang Ni, and Shuhua Li. Cluster-in-molecule local correlation method for post-hartree–fock calculations of large systems. *Molecular Physics*, 114(9):1447–1460, 2016.

[14] Wei Li, Piotr Piecuch, Jeffrey R Gour, and Shuhua Li. Local correlation calculations using standard and renormalized coupled-cluster approaches. *The Journal of chemical physics*, 131(11):114109, 2009.

[15] Péter R Nagy, Gyula Samu, and Mihály Kállay. Optimization of the linear-scaling local natural orbital ccsd (t) method: Improved algorithm and benchmark applications. *Journal of Chemical Theory and Computation*, 14(8):4193–4215, 2018.

[16] Frank Neese, Frank Wennmohs, and Andreas Hansen. Efficient and accurate local approximations to coupled-electron pair approaches: An attempt to revive the pair natural orbital method. *The Journal of chemical physics*, 130(11):114108, 2009.

[17] Zhigang Ni, Wei Li, and Shuhua Li. Fully optimized implementation of the cluster-in-molecule local correlation approach for electron correlation calculations of large systems. *Journal of Computational Chemistry*, 40(10):1130–1140, 2019.

[18] Fabijan Pavošević, Chong Peng, Peter Pinski, Christoph Riplinger, Frank Neese, and Edward F Valeev. Sparsemaps—a systematic infrastructure for reduced scaling electronic structure methods. v. linear scaling explicitly correlated coupled-cluster method with pair natural orbitals. *The Journal of chemical physics*, 146(17):174108, 2017.

[19] Johann V Pototschnig, Anastasios Papadopoulos, Dmitry I Lyakh, Michal Repisky, Loïc Halbert, Andre Severo Pereira Gomes, Hans Jørgen Aa Jensen, and Lucas Visscher. Implementation of relativistic coupled cluster theory for massively parallel gpu-accelerated computing architectures. *Journal of chemical theory and computation*, 17(9):5509–5529, 2021.

[20] Peter Pulay. Localizability of dynamic electron correlation. *Chemical physics letters*, 100(2):151–154, 1983.

[21] George D Purvis III and Rodney J Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *The Journal of Chemical Physics*, 76(4):1910–1918, 1982.

[22] Zoltán Rolik and Mihály Kállay. A general-order local coupled-cluster method based on the cluster-in-molecule approach. *The Journal of chemical physics*, 135(10):104111, 2011.

[23] Qiming Sun. Libcint: An efficient general integral library for g aussian basis functions. *Journal of computational chemistry*, 36(22):1664–1671, 2015.

[24] Qiming Sun, Timothy C Berkelbach, Nick S Blunt, George H Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D McClain, Elvira R Sayfutyarova, Sandeep Sharma, et al. Pyscf: the python-based simulations of chemistry framework. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 8(1):e1340, 2018.

[25] Qiming Sun, Xing Zhang, Samragni Banerjee, Peng Bao, Marc Barbry, Nick S Blunt, Nikolay A Bogdanov, George H Booth, Jia Chen, Zhi-Hao Cui, et al. Recent developments in the pyscf program package. *The Journal of chemical physics*, 153(2):024109, 2020.

[26] Guido vanRossum. Python reference manual. *Department of Computer Science [CS]*, (R 9525), 1995.

[27] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0

Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[28] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88:106848, 2020.

APPENDICES

Appendix A

Selected source code from the Tile structure

A.1 The fast heuristic matrix screening method

```
1        elif self.sparsityEstimationMethod == 6:
2          # heuristic estimate method
3          # it will randomly select row or column and add it into its
   dataset, until it thinks its good enough for an estimation
4          # part of the heuristic algorithm
5          # you can define the threshold to converge
6
7          heuristicDataSet = []
8          heuristicEstimationValue = 0
9          heuristicEstimationLastValue = 0
10         heuristicEstimationDif = 0
11         availableColumnIndex = [*range(self.dataArray.shape[1])] #
   argument-unpacking operator *
12         availableRowIndex = [*range(self.dataArray.shape[0])] #argument
   -unpacking operator *
13
14         # first attempt
15         if random.randint(0, 1) == 0:
16           # pick a row
17           rowIndex = random.choice(availableRowIndex)
18           heuristicDataSet = np.append(heuristicDataSet, self.dataArray
   [rowIndex, :])
19           availableRowIndex.remove(rowIndex)
```

```
20          else:
21            # pick a column
22            columnIndex = random.choice(availableColumnIndex)
23            heuristicDataSet = np.append(heuristicDataSet, self.dataArray
    [:, columnIndex])
24            availableColumnIndex.remove(columnIndex)
25          heuristicEstimationLastValue = 1 - (count_nonzero(
    heuristicDataSet) / len(heuristicDataSet))
26
27          # second attempt
28          if random.randint(0, 1) == 0:
29            # pick a row
30            rowIndex = random.choice(availableRowIndex)
31            heuristicDataSet = np.append(heuristicDataSet, self.dataArray
    [rowIndex, :])
32            availableRowIndex.remove(rowIndex)
33          else:
34            # pick a column
35            columnIndex = random.choice(availableColumnIndex)
36            heuristicDataSet = np.append(heuristicDataSet, self.dataArray
    [:, columnIndex])
37            availableColumnIndex.remove(columnIndex)
38          heuristicEstimationValue = 1 - (count_nonzero(heuristicDataSet)
     / len(heuristicDataSet))
39          heuristicEstimationDif = abs(heuristicEstimationLastValue -
    heuristicEstimationValue)
40
41          if heuristicEstimationDif <= self.heuristicEstimateThreshold:
42            self.sparsity = heuristicEstimationValue
43            self.sparsityEstimation = True
44            return self.sparsity
45          else:
46            while(not self.sparsityEstimation):
47              if len(availableRowIndex) > 0 and len(availableColumnIndex)
     > 0:
48                # both row and column are availble
49                if random.randint(0, 1) == 0:
50                  # pick a row
51                  rowIndex = random.choice(availableRowIndex)
52                  heuristicDataSet = np.append(heuristicDataSet, self.
    dataArray[rowIndex, :])
53                  availableRowIndex.remove(rowIndex)
54                else:
55                  # pick a column
56                  columnIndex = random.choice(availableColumnIndex)
```

```
57                  heuristicDataSet = np.append(heuristicDataSet, self.
   dataArray[:, columnIndex])
58                  availableColumnIndex.remove(columnIndex)
59
60              elif len(availableRowIndex) == 0 and len(
   availableColumnIndex) > 0:
61                # no row index available
62                rowIndex = random.choice(availableRowIndex)
63                heuristicDataSet = np.append(heuristicDataSet, self.
   dataArray[rowIndex, :])
64                availableRowIndex.remove(rowIndex)
65
66              elif len(availableRowIndex) > 0 and len(
   availableColumnIndex) == 0:
67                # no column index available
68                columnIndex = random.choice(availableColumnIndex)
69                heuristicDataSet = np.append(heuristicDataSet, self.
   dataArray[:, columnIndex])
70                availableColumnIndex.remove(columnIndex)
71
72              elif len(availableRowIndex) == 0 and len(
   availableColumnIndex) == 0:
73                # wait, you just scaned all the elements? you think the
   heuristic algorithm is a joke to me?
74                self.sparsity = heuristicEstimationValue
75                self.sparsityEstimation = True
76                return self.sparsity
77
78              heuristicEstimationLastValue = heuristicEstimationValue
79              heuristicEstimationValue = 1 - (count_nonzero(
   heuristicDataSet) / len(heuristicDataSet))
80              heuristicEstimationDif = abs(heuristicEstimationLastValue -
    heuristicEstimationValue)
81              if heuristicEstimationDif <= self.
   heuristicEstimateThreshold:
82                self.sparsity = heuristicEstimationValue
83                self.sparsityEstimation = True
84                return self.sparsity
```

Listing A.1: Python example for the fast heuristic matrix screening method

A.2 The efficient self-pruning method

```
def cutOffElements(self, threshold = globalSparseThreshold):
# cutOffElements will remove the values that are smaller than the
 global sparse threshold
# will be used before the conversion and after the multiplications
# this will work with the dense matrix and the csr matrix
# for csr matrix, it will delete the zeros at the end
# this does delete the elements in the data array but will not update
 the memory taken due to the running time
# bug fixed: now the cut off will keep negetive numbers
  if self.pruneSwitch:
    # this will bypass the prune behavior
    if (self.storeMethod in (None, 'dense')) and not self.cutOffStatus:
      self.dataArray = np.where(abs(self.dataArray) < threshold, 0,
 self.dataArray)
      self.cutOffStatus = True
    elif self.storeMethod == 'scipy_csr' and not self.cutOffStatus:


      # csr prune algorithm (no such thing in numpy, scipy and Intel
 MKL)
      # this will be the most efficient pruning algorithm
      # build new csr array, only scan once
      dataPositions = 0
      newData = []
      newIndices = []
      newIndexPointers = []
      newNextNNZ = 0
      newNNZ = 0
      # loop from index pointers
      for indexPointPositions in range(len(self.dataArray.indptr) - 1):
        currentNNZ = self.dataArray.indptr[indexPointPositions]
        nextNNZ = self.dataArray.indptr[indexPointPositions + 1]
        # calculate how many non-zero elements are here
        NNZThisRow = nextNNZ - currentNNZ
        for dataThisRow in range(NNZThisRow):
          # if this item is above threshold
          if abs(self.dataArray.data[dataPositions + dataThisRow]) >
 threshold:
            # add this data to the new csr
            newData += [self.dataArray.data[dataPositions + dataThisRow
 ]]
            newIndices += [self.dataArray.indices[dataPositions +
 dataThisRow]]
```

```
37                 newNextNNZ += 1
38             # record the non-zero element positions
39             dataPositions += NNZThisRow
40             # record the index pointers
41             newIndexPointers += [newNNZ]
42             newNNZ = newNextNNZ
43         # for the last index pointers
44         newIndexPointers += [newNNZ]
45         # update the csr - error the scipy will not recognize this csr
   anymore.
46         # self.dataArray.data = newData
47         # self.dataArray.indices = newIndices
48         # self.dataArray.indptr = newIndexPointers
49 
50         # making up a new csr and replace it will have huge performance
   penalty - this is only working way from scipy.
51         # sadly we would abandon python one day
52         makeupCSR = sparse.csr_matrix((newData, newIndices,
   newIndexPointers), shape=self.dataArray.shape)
53         self.dataArray = makeupCSR
54 
55         # they dont want me to update the nnz
56         # self.dataArray.nnz = newNNZ
57         # self.dataArray.data = np.where(abs(self.dataArray.data) <
   threshold, 0, self.dataArray.data)
58         # self.dataArray.eliminate_zeros()
59         # self.cutOffStatus = True
60       elif self.storeMethod == 'scipy_bsr' and not self.cutOffStatus:
61         self.dataArray.data = np.where(abs(self.dataArray.data) <
   threshold, 0, self.dataArray.data)
62         self.dataArray.eliminate_zeros()
63         self.cutOffStatus = True
```

Listing A.2: Python example for the efficient self-pruning method

A.3 The initiative method inside the Tile strcture

```
def initiative(self, givenArray):
        # initiative method design specifically for the __init__ method
        # propose: 1. complete an array search, identify the array
  properties
    # change log: add memory calculation by-pass

    if type(givenArray) is np.ndarray:
      # check if the array is in numpy array format, if not, convert the
  array into an numpy array object
      # in this case, it is an np.ndarray
      self.dataArray = givenArray
      if not self.memoryByPass:
        self.memoryTaken = self.dataArray.nbytes / pow(1024, 2)
      self.storeMethod = 'dense'

    elif isinstance(givenArray, sparse.csr_matrix):
      # check if the array is in scipy csr format
      self.dataArray = givenArray
      if not self.memoryByPass:
        self.memoryTaken = getsizeof(self.dataArray) / pow(1024, 2)
      self.storeMethod = 'scipy_csr'

    elif isinstance(givenArray, sparse.bsr_matrix):
      # check if the array is in scipy bsr format
      self.dataArray = givenArray
      if not self.memoryByPass:
        self.memoryTaken = getsizeof(self.dataArray) / pow(1024, 2)
      self.storeMethod = 'scipy_bsr'

    else:
      try:
        self.dataArray = np.array(givenArray)
        if not self.memoryByPass:
          self.memoryTaken = self.dataArray.nbytes / pow(1024, 2)
        self.storeMethod = 'dense'
      except:
        exit("Error 01 - The input object is not an numpy array and can
  not be converted to one.")


    if isinstance(givenArray, sparse.csr_matrix):
      self.originalSize = self.dataArray.shape
      self.totalElements = self.originalSize[0] * self.originalSize[1]
```

```
41        self.dimensions = self.dataArray.ndim
42        self.nonzeroElements = self.dataArray.nnz
43
44      elif isinstance(givenArray, sparse.bsr_matrix):
45        self.originalSize = self.dataArray.shape
46        self.totalElements = self.originalSize[0] * self.originalSize[1]
47        self.dimensions = self.dataArray.ndim
48        self.nonzeroElements = self.dataArray.nnz
49
50      else:
51        # retrieve some attributes from the numpy ndarray
52        self.originalSize = self.dataArray.shape
53        self.totalElements = self.dataArray.size
54        self.dimensions = self.dataArray.ndim
55
56      # get sparsity for further analysis
57      self.getSparsity()
58
59      # if the sparsity is high, auto convert the data type to csr
60      # since the bsr is better when handling sparse matrices with dense
    sub matrices, we will use bsr for lower sparsity is presentated
61      if self.storeMethod == 'dense' and self.sparsity >=
    CSRconversionThrehold:
62        self.convertDenseToCSR()
63      elif self.storeMethod == 'dense' and self.sparsity >=
    BSRconversionThrehold:
64        self.convertDenseToBSR()
```

Listing A.3: Python example for the initiative method inside the Tile strcture

A.4 Overloaded arithmetic operators inside the Tile strcture

```
1  # the addition operator overloading (used for tile master strucutre and
     anything else)
2  # __add__: Tile Any -> Tile
3  # the Tile class can add other Tiles, numpy arrays, regular arrays
4  def  __add__(self, other):
5    # Tile Tile addition
6    if isinstance(other, Tile):
7      if self.dimensions != other.dimensions:
8        exit("Error 04 - The two inputs are not at the same dimension.")
9      else:
10       if self.originalSize != other.originalSize:
11         exit("Error 03 - The two inputs are not at the same size.")
12       else:
13         return Tile(self.dataArray + other.dataArray)
14
15   # Tile numpy array addition
16   elif isinstance(other, np.ndarray):
17     # check if the addition is legal
18     if other.ndim > self.dimensions:
19       exit('Error 07 - The second argument of the addition does not
   have the correct dimensions.')
20     # ignore the 3 dimensional matrix addition at this moment
21     elif self.dimensions == 2:
22       if self.originalSize[0] != other.shape[0] or self.originalSize[1]
    != other.shape[1]:
23         exit('Error 08 - The second argument of the addition does not
   have the correct size.')
24       else:
25         return Tile(self.dataArray + other)
26
27   # Tile array addition
28   elif isinstance(other, list):
29     try:
30       givenArray = np.array(other)
31     except:
32       exit("Error 06 - The second argument of the addition can not be
   converted to an numpy array.")
33
34     # continue to check if the addition is legal
35     if givenArray.ndim > self.dimensions:
36       exit('Error 07 - The second argument of the addition does not
```

```
  have the correct dimensions.')
37      # ignore the 3 dimensional matrix addition at this moment
38      elif self.dimensions == 2:
39        if self.originalSize[0] != givenArray.shape[0] or self.
  originalSize[1] != givenArray.shape[1]:
40          exit('Error 08 - The second argument of the addition does not
  have the correct size.')
41        else:
42          return Tile(self.dataArray + givenArray)
43
44    # Tile number addition - will cause error
45    elif isinstance(other, numbers.Number):
46      if isinstance(other, bool):
47        exit("Error 09 - The second argument of the Tile addition can not
   be booleans.")
48      else:
49        if other == 0:
50          return self
51        else:
52          exit("Error 09 - The second argument of the Tile addition can
  not be a single number.")
53    else:
54      exit("Error 10 - Invalid second argument given for Tile addition.")
55
56
57  # the right handed addition operator overloading (used for tile master
   strucutre and anything else)
58  # __radd__:Any Tile -> Tile
59  # the Tile class can add other Tiles, numpy arrays, regular arrays
60  def  __radd__(self, other):
61    # Tile Tile addition
62    if isinstance(other, Tile):
63      if self.dimensions != other.dimensions:
64        exit("Error 04 - The two inputs are not at the same dimension.")
65      else:
66        if self.originalSize != other.originalSize:
67          exit("Error 03 - The two inputs are not at the same size.")
68        else:
69          return Tile(other.dataArray + self.dataArray)
70
71    # Tile numpy array addition
72    elif isinstance(other, np.ndarray):
73      # check if the addition is legal
74      if other.ndim > self.dimensions:
75        exit('Error 07 - The second argument of the addition does not
```

```
have the correct dimensions.')
76        # ignore the 3 dimensional matrix addition at this moment
77        elif self.dimensions == 2:
78          if self.originalSize[0] != other.shape[0] or self.originalSize[1]
 != other.shape[1]:
79            exit('Error 08 - The second argument of the addition does not
have the correct size.')
80          else:
81            return Tile(other + self.dataArray)
82
83      # Tile array addition
84      elif isinstance(other, list):
85        try:
86          givenArray = np.array(other)
87        except:
88          exit("Error 06 - The second argument of the addition can not be
converted to an numpy array.")
89
90        # continue to check if the addition is legal
91        if givenArray.ndim > self.dimensions:
92          exit('Error 07 - The second argument of the addition does not
have the correct dimensions.')
93        # ignore the 3 dimensional matrix addition at this moment
94        elif self.dimensions == 2:
95          if self.originalSize[0] != givenArray.shape[0] or self.
originalSize[1] != givenArray.shape[1]:
96            exit('Error 08 - The second argument of the addition does not
have the correct size.')
97          else:
98            return Tile(givenArray + self.dataArray)
99
100     # Tile number addition - will cause error
101     elif isinstance(other, numbers.Number):
102       if isinstance(other, bool):
103         exit("Error 09 - The second argument of the Tile addition can not
 be booleans.")
104       else:
105         if other == 0:
106           return self
107         else:
108           exit("Error 09 - The second argument of the Tile addition can
not be a single number.")
109     else:
110       exit("Error 10 - Invalid second argument given for Tile addition.")
111
```

```
112
113   # the subtraction operator overloading (used for tile master strucutre
      and anything else)
114   # __sub__: Tile Any -> Tile
115   # the Tile class can subtract other Tiles, numpy arrays, regular arrays

116   def __sub__(self, other):
117     # Tile Tile subtraction
118     if isinstance(other, Tile):
119       if self.dimensions != other.dimensions:
120         exit("Error 04 - The two inputs are not at the same dimension.")
121       else:
122         if self.originalSize != other.originalSize:
123           exit("Error 03 - The two inputs are not at the same size.")
124         else:
125           return Tile(self.dataArray - other.dataArray)
126
127     # Tile numpy array subtraction
128     elif isinstance(other, np.ndarray):
129       # check if the subtraction is legal
130       if other.ndim > self.dimensions:
131         exit('Error 07 - The second argument of the subtraction does not
      have the correct dimensions.')
132       # ignore the 3 dimensional matrix subtraction at this moment
133       elif self.dimensions == 2:
134         if self.originalSize[0] != other.shape[0] or self.originalSize[1]
       != other.shape[1]:
135           exit('Error 08 - The second argument of the subtraction does
      not have the correct size.')
136         else:
137           return Tile(self.dataArray - other)
138
139     # Tile array subtraction
140     elif isinstance(other, list):
141       try:
142         givenArray = np.array(other)
143       except:
144         exit("Error 06 - The second argument of the subtraction can not
      be converted to an numpy array.")
145
146       # continue to check if the subtraction is legal
147       if givenArray.ndim > self.dimensions:
148         exit('Error 07 - The second argument of the subtraction does not
      have the correct dimensions.')
149       # ignore the 3 dimensional matrix subtraction at this moment
```

```
150       elif self.dimensions == 2:
151         if self.originalSize[0] != givenArray.shape[0] or self.
    originalSize[1] != givenArray.shape[1]:
152           exit('Error 08 - The second argument of the subtraction does
    not have the correct size.')
153         else:
154           return Tile(self.dataArray - givenArray)
155 
156     # Tile number subtraction - will cause error
157     elif isinstance(other, numbers.Number):
158       if isinstance(other, bool):
159         exit("Error 09 - The second argument of the Tile subtraction can
    not be booleans.")
160       else:
161         if other == 0:
162           return self
163         else:
164           exit("Error 09 - The second argument of the Tile subtraction
    can not be a single number.")
165     else:
166       exit("Error 10 - Invalid second argument given for Tile subtraction
    .")
167 
168 
169   # the right handed subtraction operator overloading (used for tile
    master strucutre and anything else)
170   # __sub__: Any Tile-> Tile
171   # the Tile class can subtract other Tiles, numpy arrays, regular arrays

172   def __rsub__(self, other):
173     # Tile Tile subtraction
174     if isinstance(other, Tile):
175       if self.dimensions != other.dimensions:
176         exit("Error 04 - The two inputs are not at the same dimension.")
177       else:
178         if self.originalSize != other.originalSize:
179           exit("Error 03 - The two inputs are not at the same size.")
180         else:
181           return Tile(other.dataArray - self.dataArray)
182 
183     # Tile numpy array subtraction
184     elif isinstance(other, np.ndarray):
185       # check if the subtraction is legal
186       if other.ndim > self.dimensions:
187         exit('Error 07 - The second argument of the subtraction does not
```

```
have the correct dimensions.')
188      # ignore the 3 dimensional matrix subtraction at this moment
189      elif self.dimensions == 2:
190        if self.originalSize[0] != other.shape[0] or self.originalSize[1]
 != other.shape[1]:
191          exit('Error 08 - The second argument of the subtraction does
not have the correct size.')
192        else:
193          return Tile(other - self.dataArray)
194
195    # Tile array subtraction
196    elif isinstance(other, list):
197      try:
198        givenArray = np.array(other)
199      except:
200        exit("Error 06 - The second argument of the subtraction can not
be converted to an numpy array.")
201
202      # continue to check if the subtraction is legal
203      if givenArray.ndim > self.dimensions:
204        exit('Error 07 - The second argument of the subtraction does not
have the correct dimensions.')
205      # ignore the 3 dimensional matrix subtraction at this moment
206      elif self.dimensions == 2:
207        if self.originalSize[0] != givenArray.shape[0] or self.
originalSize[1] != givenArray.shape[1]:
208          exit('Error 08 - The second argument of the subtraction does
not have the correct size.')
209        else:
210          return Tile(givenArray - self.dataArray)
211
212    # Tile number subtraction - will cause error
213    elif isinstance(other, numbers.Number):
214      if isinstance(other, bool):
215        exit("Error 09 - The second argument of the Tile subtraction can
not be booleans.")
216      else:
217        if other == 0:
218          return Tile(self.dataArray * -1)
219        else:
220          exit("Error 09 - The second argument of the Tile subtraction
can not be a single number.")
221    else:
222      exit("Error 10 - Invalid second argument given for Tile subtraction
.")
```

```
223
224
225   # the multiplication operator (example: myTile * 2)
226   # __mul__: Tile Any -> Tile or NumPy Array
227   # the Tile class can multiply other Tiles, numpy arrays, regular arrays
        , and constants
228   def __mul__(self, other):
229     # Tile Tile multiplication
230     if isinstance(other, Tile):
231       if self.storeMethod == 'dense':
232         if other.storeMethod == 'dense':
233           return self.multiTwoDense(other)
234
235         elif other.storeMethod == 'scipy_csr':
236           return self.multiDenseCSR(other)
237
238         elif other.storeMethod == 'scipy_bsr':
239           return self.multiDenseBSR(other)
240
241       elif self.storeMethod == 'scipy_csr':
242         if other.storeMethod == 'dense':
243           return self.multiCSRDense(other)
244
245         elif other.storeMethod == 'scipy_csr':
246           return self.multiTwoCSR(other)
247
248         elif other.storeMethod == 'scipy_bsr':
249           return self.multiCSRBSR(other)
250
251       elif self.storeMethod == 'scipy_bsr':
252         if other.storeMethod == 'dense':
253           return self.multiBSRDense(other)
254
255         elif other.storeMethod == 'scipy_csr':
256           return self.multiBSRCSR(other)
257
258         elif other.storeMethod == 'scipy_bsr':
259           return self.multiTwoBSR(other)
260
261       else:
262         exit('Error 05 - The second argument of the multiplication does
        not have the correct store method.')
263
264     # Tile numpy array multiplication
265     elif isinstance(other, np.ndarray):
```

```
266       # check if the multiplication is legal
267       if other.ndim > self.dimensions:
268         exit('Error 07 - The second argument of the multiplication does
    not have the correct dimensions.')
269       # ignore the 3 dimensional matrix multiplication at this moment
270       elif self.dimensions == 2:
271         if self.originalSize[1] != other.shape[0]:
272           exit('Error 08 - The second argument of the multiplication does
     not have the correct size.')
273         else:
274           if self.storeMethod == 'dense':
275             if other.ndim == 1:
276               return np.dot(self.dataArray, other)
277             elif other.ndim == 2:
278               return Tile(np.matmul(self.dataArray, other))
279           elif self.storeMethod == 'scipy_csr':
280             if other.ndim == 1:
281               return sparse.csr_matrix.dot(self.dataArray, other)
282             elif other.ndim == 2:
283               return Tile(sparse.csr_matrix.dot(self.dataArray, other))
284           elif self.storeMethod == 'scipy_bsr':
285             if other.ndim == 1:
286               return sparse.bsr_matrix.dot(self.dataArray, other)
287             elif other.ndim == 2:
288               return Tile(sparse.bsr_matrix.dot(self.dataArray, other))
289
290     # Tile array multiplication
291     elif isinstance(other, list):
292       try:
293         givenArray = np.array(other)
294       except:
295         exit("Error 06 - The second argument of the multiplication can
    not be converted to an numpy array.")
296
297       # continue to check if the multiplication is legal
298       if givenArray.ndim > self.dimensions:
299         exit('Error 07 - The second argument of the multiplication does
    not have the correct dimensions.')
300       # ignore the 3 dimensional matrix multiplication at this moment
301       elif self.dimensions == 2:
302         if self.originalSize[1] != givenArray.shape[0]:
303           exit('Error 08 - The second argument of the multiplication does
     not have the correct size.')
304         else:
305           if self.storeMethod == 'dense':
```

```
306             if givenArray.ndim == 1:
307               return np.dot(self.dataArray, givenArray)
308             elif givenArray.ndim == 2:
309               return Tile(np.matmul(self.dataArray, givenArray))
310           elif self.storeMethod == 'scipy_csr':
311             if givenArray.ndim == 1:
312               return sparse.csr_matrix.dot(self.dataArray, givenArray)
313             elif givenArray.ndim == 2:
314               return Tile(sparse.csr_matrix.dot(self.dataArray,
    givenArray))
315           elif self.storeMethod == 'scipy_bsr':
316             if givenArray.ndim == 1:
317               return sparse.bsr_matrix.dot(self.dataArray, givenArray)
318             elif givenArray.ndim == 2:
319               return Tile(sparse.bsr_matrix.dot(self.dataArray,
    givenArray))
320
321       # Tile number multilication
322       elif isinstance(other, numbers.Number):
323         if isinstance(other, bool):
324           exit("Error 09 - The second argument of the multiplication can
    not be booleans.")
325         else:
326           if self.storeMethod == 'dense':
327             return Tile(np.dot(self.dataArray, other))
328           elif self.storeMethod == 'scipy_csr':
329             return Tile(sparse.csr_matrix.dot(self.dataArray, other))
330           elif self.storeMethod == 'scipy_bsr':
331             return Tile(sparse.bsr_matrix.dot(self.dataArray, other))
332
333       else:
334         exit("Error 10 - Invalid second argument given for Tile
    multiplication.")
335
336
337     # the right multiplication operator (example: 2 * myTile)
338     # __mul__: Any Tile -> Tile
339     # the Tile class can multiply other Tiles, numpy arrays, regular arrays
     , and constants
340     # this is designed due to the uniqueness of the linear algebra AB != BA

341     # there is no Tile Tile multiplication implemented in the right
     multiplication operator
342     # When Python attempts to multiply two objects, it first tries to call
     the left object's __mul__() method
```

```
  def __rmul__(self, other):
    #print(other)
    # Tile numpy array multiplication
    if isinstance(other, np.ndarray):
      # check if the multiplication is legal
      if other.ndim > self.dimensions:
        exit('Error 07 - The first argument of the right-handed
  multiplication does not have the correct dimensions.')
      # ignore the 3 dimensional matrix multiplication at this moment
      elif self.dimensions == 2:
        # different here
        if self.originalSize[0] != other.shape[1]:
          exit('Error 08 - The second argument of the right-handed
  multiplication does not have the correct size.')
        else:
          if self.storeMethod == 'dense':
            if other.ndim == 1:
              return np.dot(other, self.dataArray)
            elif other.ndim == 2:
              return Tile(np.matmul(other, self.dataArray))
          elif self.storeMethod == 'scipy_csr':
            if other.ndim == 1:
              return sparse.csr_matrix.dot(other, self.dataArray)
            elif other.ndim == 2:
              return Tile(sparse.csr_matrix.dot(other, self.dataArray))
          elif self.storeMethod == 'scipy_bsr':
            if other.ndim == 1:
              return sparse.bsr_matrix.dot(other, self.dataArray)
            elif other.ndim == 2:
              return Tile(sparse.bsr_matrix.dot(other, self.dataArray))

    # Tile array multiplication
    elif isinstance(other, list):
      try:
        givenArray = np.array(other)
      except:
        exit("Error 06 - The second argument of the right-handed
  multiplication can not be converted to an numpy array.")

      # continue to check if the multiplication is legal
      if givenArray.ndim > self.dimensions:
        exit('Error 07 - The second argument of the right-handed
  multiplication does not have the correct dimensions.')
      # ignore the 3 dimensional matrix multiplication at this moment
      elif self.dimensions == 2:
```

```
384         # different here
385         if self.originalSize[0] != givenArray.shape[1]:
386           exit('Error 08 - The second argument of the right-handed
    multiplication does not have the correct size.')
387         else:
388           if self.storeMethod == 'dense':
389             if givenArray.ndim == 1:
390               return np.dot(givenArray, self.dataArray)
391             elif givenArray.ndim == 2:
392               return Tile(np.matmul(givenArray, self.dataArray))
393           elif self.storeMethod == 'scipy_csr':
394             if givenArray.ndim == 1:
395               return sparse.csr_matrix.dot(givenArray, self.dataArray)
396             elif givenArray.ndim == 2:
397               return Tile(sparse.csr_matrix.dot(givenArray, self.
    dataArray))
398           elif self.storeMethod == 'scipy_bsr':
399             if givenArray.ndim == 1:
400               return sparse.bsr_matrix.dot(givenArray, self.dataArray)
401             elif givenArray.ndim == 2:
402               return Tile(sparse.bsr_matrix.dot(givenArray, self.
    dataArray))
403
404     # Tile number multilication
405     elif isinstance(other, numbers.Number):
406       if isinstance(other, bool):
407         exit("Error 09 - The second argument of the right-handed
    multiplication can not be booleans.")
408       else:
409         if self.storeMethod == 'dense':
410           return Tile(np.dot(other, self.dataArray))
411         elif self.storeMethod == 'scipy_csr':
412           return Tile(sparse.csr_matrix.dot(other, self.dataArray))
413         elif self.storeMethod == 'scipy_bsr':
414           return Tile(sparse.bsr_matrix.dot(other, self.dataArray))
415
416     else:
417       exit("Error 10 - Invalid second argument given for right-handed
    Tile multiplication.")
```

Listing A.4: Python example for overloaded arithmetic operators inside the Tile strcture

A.5 Auto-determination of the BSR block size from SciPy.sparse [27]

The following code is quoted directly from the SciPy library to illustrate the functionality of the auto-determination of the block size. I have no contribution in these function and I have used this library under the liberal BSD license. Citation is here: [27]

```
def estimate_blocksize(A,efficiency=0.7):
    """Attempt to determine the blocksize of a sparse matrix
    Returns a blocksize=(r,c) such that
        - A.nnz / A.tobsr( (r,c) ).nnz > efficiency
    """
    if not (isspmatrix_csr(A) or isspmatrix_csc(A)):
        A = csr_matrix(A)

    if A.nnz == 0:
        return (1,1)

    if not 0 < efficiency < 1.0:
        raise ValueError('efficiency must satisfy 0.0 < efficiency < 1.0'
    )

    high_efficiency = (1.0 + efficiency) / 2.0
    nnz = float(A.nnz)
    M,N = A.shape

    if M % 2 == 0 and N % 2 == 0:
        e22 = nnz / (4 * count_blocks(A,(2,2)))
    else:
        e22 = 0.0

    if M % 3 == 0 and N % 3 == 0:
        e33 = nnz / (9 * count_blocks(A,(3,3)))
    else:
        e33 = 0.0

    if e22 > high_efficiency and e33 > high_efficiency:
        e66 = nnz / (36 * count_blocks(A,(6,6)))
        if e66 > efficiency:
            return (6,6)
        else:
            return (3,3)
    else:
        if M % 4 == 0 and N % 4 == 0:
```

```
37              e44 = nnz / (16 * count_blocks(A,(4,4)))
38          else:
39              e44 = 0.0
40
41          if e44 > efficiency:
42              return (4,4)
43          elif e33 > efficiency:
44              return (3,3)
45          elif e22 > efficiency:
46              return (2,2)
47          else:
48              return (1,1)
49
50  def count_blocks(A,blocksize):
51      """For a given blocksize=(r,c) count the number of occupied
52      blocks in a sparse matrix A
53      """
54      r,c = blocksize
55      if r < 1 or c < 1:
56          raise ValueError('r and c must be positive')
57
58      if isspmatrix_csr(A):
59          M,N = A.shape
60          return csr_count_blocks(M,N,r,c,A.indptr,A.indices)
61      elif isspmatrix_csc(A):
62          return count_blocks(A.T,(c,r))
63      else:
64          return count_blocks(csr_matrix(A),blocksize)
```

Listing A.5: Python example for the auto-determination of the BSR block size from SciPy.sparse [27].