SISSA – ICTP

MASTER IN HIGH PERFORMANCE COMPUTING

Master Thesis

IMPROVING PERFORMANCE OF BASIS-SET-FREE HARTREE-FOCK CALCULATIONS THROUGH GRID-BASED MASSIVELY PARALLEL TECHNIQUES

Author:
Edwin Fernando Posada Correa, MSc

Supervisor:
Giuseppe Piero Brandino, PhD

ACADEMIC YEAR 2014/2015

ACKNOWLEDGEMENTS

The author would like to thank to all MHPC staff, in particular to Giuseppe Piero Brandino as his advisor, to Ivan Girotto from ICTP and to MHPC coordinator Stefano Cozzini.

This work and the permanence in Trieste were supported by the ICTP TRIL Programme, grant SMR 2352.

CONTENTS

1 INTRODUCTION 6
2 BACKGROUND 8
2.1 Multi-center Numerical Integrator 8
2.1.1 Single-center subintegrations 9
2.2 Calculation of Coulomb integrals 10
2.2.1 Solution of Poisson's equation 11
3 NAPMO PACKAGE 13
3.1 nAPMO structure 14
3.1.1 Python Interface 14
3.1.2 C library 15
3.2 Results 17
3.2.1 Multi-center integration 17
3.2.2 Two electron Coulomb integration 17
4 PERFORMANCE AND PARALLELIZATION 20
4.1 Testing configuration and environment 20
4.1.1 System 20
4.1.2 Compilers 20
4.2 Multi-center integrator 21
4.2.1 Serial performance 22
4.2.2 OpenMP implementation 22
4.2.2.1 Grids construction 22
4.2.2.2 Becke weights 23
4.2.2.3 Calculation of the Integrand 25
4.2.2.4 Results 25
4.2.3 CUDA implementation 27
4.2.3.1 Results 28
4.3 Two-electronic Coulomb integration 32
4.3.1 Serial performance 33
4.3.2 OpenMP performance 34
5 SUMMARY AND CONCLUSIONS 36
5.1 Future work 37
6 BIBLIOGRAPHY 38

LIST OF FIGURES

Figure 1 Relative nuclear weights, the color scale shows the value of the function $w_n(\mathbf{r})$ . . . 9
Figure 2 Structure of nAPMO package . . . 14
Figure 3 Averaged error calculating $\int \rho(\mathbf{r})$ for O_2 molecule with different grid sizes. Color in log scale. . . . 18
Figure 4 Algorithm to perform multi-center numerical integration . . . 21
Figure 5 Geometrical description for Becke-weight calculation of point i in a system with centers A and B . . . 23
Figure 6 Time and speedup vs number of threads for the integration of $\rho(\mathbf{r})$ for O_2 molecule using different grid sizes. Basis-set used 6−31+G**. . . . 26
Figure 7 Time and speedup vs Grid size for the integration of $\rho(\mathbf{r})$ for H_2 and O_2 on CUDA device. Basis-set used 6−31+G**. Grid size, angular × radial points. . . . 30
Figure 8 Time and speedup vs number of threads for the integration of $\rho(\mathbf{r})$ for H_2 molecule using different grid sizes. The dashed line corresponds to OpenMP and solid to CUDA. Basis-set used 6−31G**. . . . 31
Figure 9 Algorithm to perform two-electron Coulomb numerical integration. . . . 32
Figure 10 Time and speedup vs number of threads for two-coulomb integration of H_2 STO−3G basis-set at different grid sizes. . . . 35

LIST OF TABLES

Table 1 Requisites to compile and run nAPMO 14

Table 2 Numerical integration of $\rho(\mathbf{r})$ for different diatomic molecules. 110 angular points were used in this calculation. 18

Table 3 Two-electron Coulomb integration over STO-3G basis-set for H_2 molecule. 19

Table 4 Intel VTune amplifier OpenMP (8 threads) analysis for the integration of $\rho(\mathbf{r})$ molecule O_2. 6$-$31+G** basis-set. Grid sizes (angular $\times$ radial points): a: 1202$\times$100, b: 1202$\times$500, c: 1202$\times$1000, d: 5810$\times$1000. . . 25

Table 5 Percentage of several NVIDIA profiler metrics for calculation of $\int \rho(\mathbf{r})$ for O_2 molecule. $6-31$+G** basis-set used. Grid size: 5810 angular by 1000 radial points. 29

ACRONYMS

ICTP International Center for Theoretical Physics

SISSA Scuola Internazionale Superiore di Studi Avanzati

MHPC Master in High Performance Computing

HF Hartree-Fock Molecular Orbital

SCF Self-Consistent Field

MPI Message Passing Interface

OpenMP Open Multi-Processing

CUDA Compute Unified Device Architecture

GPU Graphics Processing Unit

GTO Gaussian Type Orbital

NAPMO Numerical Any Particle Molecular Orbital

1

INTRODUCTION

For many years the performance of scientific softwares has been one of the keys to expand the frontiers of science. The case of Computational Chemistry is not an exception. Quantum chemists all around the world have worked intensively to produce faster and more efficient software in order to be able to study bigger and more complex systems.

Hartree-Fock (HF) molecular orbital theory is one of the fundamental pillars of Quantum Chemistry, and as such, it has been in development for many years. This constant development includes improved algorithms to accelerate the self consistent field (SCF) convergence, more efficient algorithms to perform integration [21], and lately, the implementation of all those algorithms using parallel techniques, such as, MPI (Message Passing Interface), OpenMP (Open Multi-Processing), CUDA, among others. Nowadays, HF can be applied to molecules containing hundreds of atoms in commodity computers.

Discussion on HF theory is not the main goal of this work, but let us to recall some of the HF equations that are needed to illustrate the work done here. More detailed information on HF theory can be found in reference [20].

The single-particle HF equation is defined as

$$F(1)|\chi_a(1)\rangle = \varepsilon_a|\chi_a(1)\rangle \tag{1}$$

For particle (1) with single particle wave-function χ_a and Fock operator F expressed as

$$F(1) = \hat{T}(1) + \hat{V}(1) + \hat{J}(1) - \hat{K}(1) \tag{2}$$

where $\hat{T}(1)$ is the kinetic energy operator, $\hat{V}(1)$ the electron-nucleus potential energy operator and $\hat{J}(1) - \hat{K}(1)$ are the coulomb and exchange operators respectively. This HF equation can be solved iteratively through the SCF procedure [20].

In this work we will focused in the Coulomb operator $\hat{J}$ which is defined as

$$\hat{J}_b(1) = \sum_{b \neq a} \int \frac{|\chi_b(2)|^2}{\mathbf{r}_{12}} d^3\mathbf{r}_2 \tag{3}$$

The expectation value of $\hat{\mathcal{J}}_b(1)$ leads to the two-electron Coulomb integral

$$\langle \chi_a(1) | \hat{\mathcal{J}}_b(1) | \chi_a(1) \rangle = I_{aa,bb} = \int \frac{\chi_a(1)\chi_a(1)\chi_b(2)\chi_b(2)}{\mathbf{r}_{12}} d^3\mathbf{r}_1 d^3\mathbf{r}_2 \tag{4}$$

which is known to be the biggest bottleneck in HF implementations. Finally, χ_i is expressed as a linear combination of basis functions ϕ_μ, such as,

$$\chi_i = \sum_{\mu=1}^{k} C_{\mu i} \phi_\mu \tag{5}$$

A popular choice of basis functions ϕ_μ are the Gaussian type orbitals (GTO) since they allow an efficient evaluation of molecular integrals such as the two-electron Coulomb integral. Efficient strategies for evaluating this kind of integrals over GTOs have been developed, including, screening and density fitting schemes [19] [21] [10] [8].

Despite their success, GTO basis-sets have disadvantages, such as the poor description of cusps and linear dependence of large basis-sets which can lead to poor conditioned equations, loss of precision, and poor convergence in the iterative procedure. Additionally, GTO basis-set also causes the so called basis-set superposition error, which can impact in the reliability of the outcome[4].

The use of alternative basis functions results in more complex and expensive integration procedures, that could not have been afforded for scientist for many years. Nowadays with the growth of computational power and the use of accelerators, such as GPUs or Xeon PHI in computational sciences, the use of this alternatives starts to be feasible[16].

The basis-set-free methods allow a free choice of the basis functions since they offer numerical integration for any arbitrary integrand by discretization of the molecular space on spherical grids, avoiding nuclear singularities and offering, depending on the basis function, a correct description of the nuclear cusp.

In this work we have implemented the basis-set free calculation of the Coulomb operator (two-electron Coulomb integrals) and the numerical integration for an arbitrary integrand $F(\mathbf{r})$ over the molecular domain including OpenMP parallelization and massively parallelization on CUDA-capable devices.

2

BACKGROUND

In this chapter we will discuss about the fundamental concepts used in this work.

2.1 MULTI-CENTER NUMERICAL INTEGRATOR

The following describes briefly the multi-center numerical integration method proposed by Becke et. al. [2].

The aim of this method is to approximate integrals of the type

$$I = \int F(\mathbf{r}) d^3\mathbf{r} \tag{6}$$

where $F(\mathbf{r})$ is an arbitrary integrand, by a discrete numerical summation of the form

$$I = \sum_i A_i F(\mathbf{r}_i) \tag{7}$$

where $\mathbf{r}_i$ and A_i are the grid points and their respective integration weights. In a multi-center system such as a polyatomic molecule, the integrand $F(\mathbf{r})$ may be decomposed into single-center components $F_n(\mathbf{r})$

$$F_n(\mathbf{r}) = w_n(\mathbf{r}) F(\mathbf{r}) \tag{8}$$

such that

$$F(\mathbf{r}) = \sum_n F_n(\mathbf{r}) \tag{9}$$

so that the integral of eq. 6 is reduced to a sum of single-center integrations I_n over each nuclei in the system.

$$I = \sum_n I_n \tag{10}$$

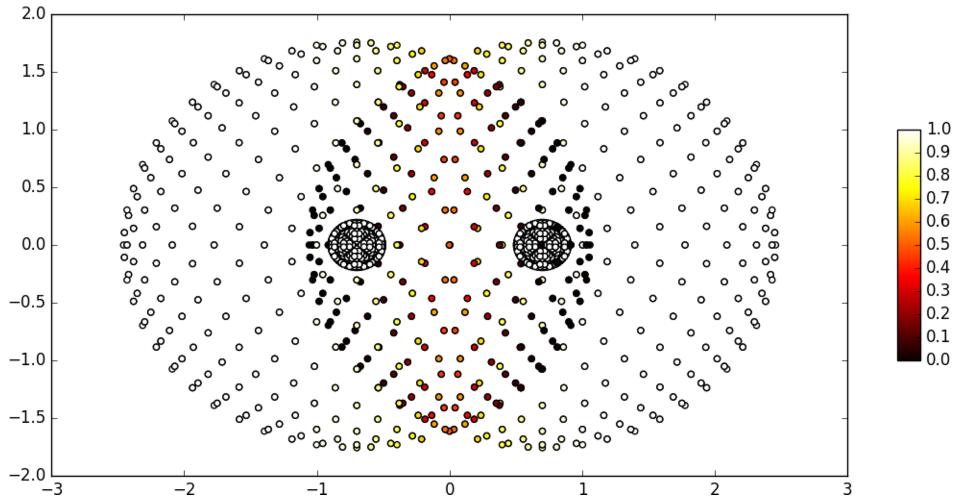

Figure 1: Relative nuclear weights, the color scale shows the value of the function $w_n(\mathbf{r})$

where

$$I_n = \int F_n(\mathbf{r}) d^3\mathbf{r} \tag{11}$$

The relative weight function $w_n(\mathbf{r})$ is assigned to each nucleus n such that, for all $\mathbf{r}$

$$\sum_n w_n(\mathbf{r}) = 1 \tag{12}$$

and such that each $w_n(\mathbf{r})$ has the value of 1 on the vicinity of its own nucleus, but vanishes uniformly near to other nucleus. Figure 1 illustrates the behavior of the relative weight function.

The definition of function $w_n(\mathbf{r})$ depends on the atomic size, the internuclear separation and the distance between point $\mathbf{r}$ and nuclei n. Further details on the derivation of the relative nuclear weights can be found in reference [2].

2.1.1 *Single-center subintegrations*

The procedure for computing the single-center integrals I_n involves both radial and angular quadrature for a spherical polar system (r, ϑ, φ) on nucleus n. The volume integral for a single center component $F_n(\mathbf{r})$ is therefore expressed as

$$I_n = \int\int\int F_n(r, \vartheta, \varphi) r^2 \sin\vartheta dr d\vartheta \varphi \tag{13}$$

This three-dimensional integral can be calculated using Gauss-type quadrature for two dimensional integration on the surface of the unit sphere and therefore the integral of eq. 13 can be rewritten as

$$I_n = \int\int F_n(r,\Omega) r^2 dr d\Omega \tag{14}$$

where Ω denotes the solid angle.

The integration over solid angle Ω is carried out using Lebedev grids [15]. Lebedev grid is characterized by its rank ℓ_{quad} and is designed to integrate spherical harmonics $Y_{\ell m}$ exactly up to $\ell = \ell_{quad}$.

Finally radial integration is performed using Gauss-Chebyshev quadrature of second kind. However, the standard Gauss-Chebyshev integration interval $-1 < x < +1$ must be mapped into the semi-infinite interval $0 < r < \infty$. This mapping can be done through the following coordinate transformation

$$r = r_m \frac{1+x}{1-x} \tag{15}$$

where r_m corresponds to the half of the atomic radius for all atoms [5] except for the hydrogen atom in which case the factor $1/2$ is not applied.

2.2 CALCULATION OF COULOMB INTEGRALS

We now briefly discuss the methodology for the calculation of the Coulomb integral of equation 4, 4 using the methodology described in reference [3] and [18].

Let us recall the Poisson's equation

$$\nabla^2 V = -4\pi\rho \tag{16}$$

for the potential $V(\mathbf{r})$ of an arbitrary distribution $\rho(\mathbf{r})$ in a multi-center i.e. polyatomic system. Solving equation 16 is equivalent to solve the integral

$$V(\mathbf{r}_1) = \int \frac{\rho(\mathbf{r}_2)}{r_{12}} d^3\mathbf{r}_2 \tag{17}$$

for all points $\mathbf{r}_1$. Combining the solution of Poisson's equation and the multi-center numerical integrator described in Section 2.1 the two-electron coulomb integral

$$I_{\alpha\beta,\gamma\nu} = \int\int \frac{\phi_\alpha(1)\phi_\beta(1)\phi_\gamma(2)\phi_\nu(2)}{r_{12}} d^3r_1 d^3r_2 \tag{18}$$

for basis functions ϕ_α can be calculated by solving the Poisson's equation for potential $V_{\alpha\beta}$ of the charge distribution $\phi_\alpha\phi_\beta$ and then evaluate the integral

$$I_{\alpha\beta,\gamma\nu} = \int V_{\alpha\beta}\phi_\gamma\phi_\nu d^3r \tag{19}$$

which can be solved using the quadratures explained on Section 2.1.

2.2.1 *Solution of Poisson's equation*

As explained above, any arbitrary integrand can be decomposed in single-center functions, so the charge distribution $\rho(\mathbf{r})$ can be decomposed as

$$\rho(\mathbf{r}) = \sum_n \rho_n(\mathbf{r}) \tag{20}$$

$$\rho_n(\mathbf{r}) = w_n(\mathbf{r})\rho(\mathbf{r}) \tag{21}$$

such that each single-center component of the charge distribution leads to a n single-center Poisson's equation of the form

$$\nabla^2 V^{(n)} = -4\pi\rho_n \tag{22}$$

so that the total potential V can be calculated by summing all $V^{(n)}$ potentials.

To solve the single-center Poisson's equations 22 one may express the single-center charge distribution as a multi-polar expansion of the form

$$\rho_n(r,\vartheta,\varphi) = \sum_{\ell m} \rho_{\ell m}(r) Y_{\ell m}(\vartheta,\varphi) \tag{23}$$

where $Y_{\ell m}(\vartheta,\varphi)$ corresponds to a real spherical harmonics functions and $\rho_{\ell m}(r)$ are functions of the r variable only. $\rho_{\ell m}(r)$ functions can be calculated through orthogonality of the spherical expansion as follows

$$\rho_{\ell m}(r) = \int_\Omega \rho_n(r,\vartheta,\varphi) Y_{\ell m}(\vartheta,\varphi) d\Omega \tag{24}$$

such integrals can be calculated using the already mentioned Lebedev quadratures [15]. In principle, the multi-polar expansion is infinite, however the truncation order of the expansion $l_{\max}$ has been chosen to be the half of the order of the Lebedev quadrature l_{quad}

$$l_{\max} = l_{quad}/2 \tag{25}$$

In a similar fashion the potential $V^{(n)}$ can be expanded as

$$V^{(n)}(r,\vartheta,\varphi) = \sum_{\ell m}^{l_{\max}} r^{-1} U_{\ell m}(r) Y_{\ell m}(\vartheta,\varphi) \tag{26}$$

Replacing those expansions into eq. 22 we obtain

$$\frac{\partial^2}{\partial r^2} U_{\ell m} - \frac{l(l+1)}{r^2} U_{\ell m} = -4\pi r \rho_{\ell m} \tag{27}$$

The equations for $U_{\ell m}$ are solved by a finite-difference method.

In order to simplify the calculation of the differential equations 27 we map the r points of equation 15 into a uniform grid z with the following coordinate change

$$z = \frac{\arccos\left(\frac{r-r_m}{r+r_m}\right)}{\pi} \tag{28}$$

Using chain rule we can convert equation 27 from the r variable to the z variable, as

$$\frac{\partial^2}{\partial z^2} U_{\ell m} \left(\frac{\partial z}{\partial r}\right)^2 + \frac{\partial}{\partial z} U_{\ell m} \frac{\partial^2 z}{\partial r^2} - \frac{l(l+1)}{r^2} U_{\ell m} = -4\pi r \rho_{\ell m} \tag{29}$$

with r equal to

$$r = r_m \frac{1+\cos(\pi z)}{1-\cos(\pi z)} \tag{30}$$

We now discuss the boundary conditions. At $r = \infty$ ($z = 0$) all $U_{\ell m}$ has the value of zero except for U_{00} in which case

$$U_{00}(r \to \infty) = \sqrt{4\pi q_n} \tag{31}$$

where q_n is the total charge of the single-center source density

$$q_n = \int \rho_n d^3\mathbf{r} \tag{32}$$

On the other hand at $r = 0$ ($z = 1$) all the $U_{\ell m}$ functions take the value of zero without exception.

Finally, after solving equation 27 for $U_{\ell m}$ equations, the potential $V(\mathbf{r})$ can be reconstructed using equation 26.

3

NAPMO PACKAGE

The computational implementation of the theory explained in Chapter 2 is called *nAPMO* (numerical Any Particle Molecular Orbital) which is intended to be the numerical version of *LOWDIN* package [9]. In this section we describe the implementation of the package developed in this work.

nAPMO code was written mainly in Python and C languages. The use of Python allows easier development and scripting, as well as offering different tools to visualize and to customize the code. However pure python code has poor performance for numerical calculations. That is why C coding become important due to the fact that is faster and it can be easily parallelized.

The overall code was written from scratch during the MHPC program. To this day nAPMO contains more than 10.000 lines of code, the following is the outcome of *cloc* command

```
http://cloc.sourceforge.net v 1.64
-------------------------------------------
Language      files  blank  comment    code
-------------------------------------------
C                10    644      832    6045
JSON              6      1        0    2097
Python           35    722      861    1609
CUDA              4     96       62     309
C/C++ Header     13    124      221     294
make              6     55       39     197
-------------------------------------------
SUM:             74   1642     2015   10551
-------------------------------------------
```

The code can be compiled using the standard *make* command in three different ways; SERIAL, OMP (OpenMP), and CUDA. Table 1 summarizes the requisites to compile and run nAPMO.

Regardless of the compilation flavor, the user interface is made such that the OMP and CUDA implementations are totally transparent from the user perspective. Now we will discuss the structure of the program.

Requisites	SERIAL	OMP	CUDA
C compiler (GCC, INTEL)	√	√	√
SciPy	√	√	√
libGSL	√	√	√
libCBLAS	√	√	√
OpenMP		√	√
CUDA ToolKit			√

Table 1: Requisites to compile and run nAPMO

3.1 NAPMO STRUCTURE

The program is divided in two main parts; the C library called *napmo_library* and the Python interface. The figure 2 shows the file-system structure of the package.

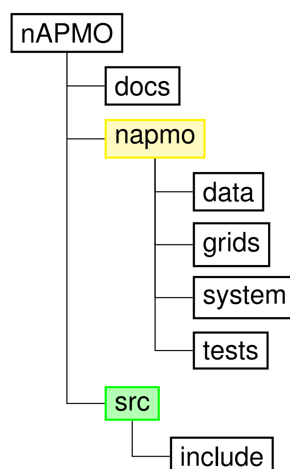

Figure 2: Structure of nAPMO package

3.1.1 *Python Interface*

The Python interface consists of 3 modules, *system*, *grids* and *data*. The *system* module manages the molecular system interface. For instance, the following code creates a `MolecularSystem` object for H_2 molecule.

```
from napmo.system.molecular_system import MolecularSystem

molecule = MolecularSystem ('H2')
molecule.add_atom('H', [0.0,0.0,0.371])
molecule.add_atom('H', [0.0,0.0,0.371])
molecule.show()
```

```
==================================================
Object: MolecularSystem
--------------------------------------------------
AtomicElement:  atoms  number of e-:  2
Symbol  Z     origin (Bohr)              Basis-set
H       1     [ 0.    0.     0.69919862]    None
H       1     [ 0.    0.    -0.69919862]    None
--------------------------------------------------
```

The *data* module handles all information related to atomic elements constants, elementary particles and fundamental constants. It also contains different kind of basis-sets. Additional basis-sets and supplementary information can be added using *json* format.

Finally, the module *grids* contains the implementation of Gauss-Chebyshev grids (`radial`), Lebedev grids (`angular`) and different level of abstraction of grids such as '`atomic`' for atoms and '`becke`' for molecules. Additionally the module *grids* contains a Poisson solver to calculate Coulomb Potentials. As an example, the following code builds the grid represented in figure 1.

```
from napmo.grids.becke import BeckeGrid

angularPoints = 590
radialPoints = 2
grid = BeckeGrid(molecule, radialPoints, angularPoints)
```

Once created, the grid object contains all the data required to perform numerical integration or to solve Poisson's equation for a given function.

3.1.2 *C library*

The C library is designed to work along with python objects through *ctypes* library. This means that the corresponding structures on C were created to match the tuple list `_fields_` coded in the class definition on Python interfaces. For instance, the definition

```
class RadialGrid(Structure):
    _fields_ = [
    ('_size', c_int),
    ('_points', POINTER(c_double)),
    ('_weights', POINTER(c_double)),
]
```

corresponds to the C structure

```
struct _radial {
  int size;         // number of grid points.
  double *points;   // points of the grid
  double *weights; // weights of the grid
```

```
};
typedef struct _radial RadialGrid;
```

The memory allocation is done in Python through *numpy* arrays [22]. This means that for each `'_member'` `POINTER(c_type)` variable in `_fields_` list, exist one `self.member` variable of the type *numpy.ndarray*. The ctypes pointers are set to point the data of its correspondent numpy array. The following code illustrates the allocation and instantiation of the C pointer.

```python
import numpy as np

def __init__(self, size, atomic_symbol):
    self.points = np.empty(self.size, dtype=np.float64)
    self._points = np.ctypeslib.as_ctypes(self.points)
```

Note that the memory allocation is done only once, in the initialization of the object, and that the `self._points` pointer is pointing to the data of the `self.points` numpy array. This strategy of memory allocation allows to access the data in a numpy fashion and to take advantage of the huge *SciPy* library [14] or visualization tools such as *Matplotlib* [13]

New functions can be added to the C library and be available on Python interfaces by importing the library in the following manner

```python
from napmo.system.cext import napmo_library
```

However a Python function prototype may be needed to call the new C function. For example, for the C function,

```c
void angular_spherical_expansion(AngularGrid *grid,
                                 const int lmax,
                                 const int size_f,
                                 double *f,
                                 double *output);
```

the corresponding Python function prototype looks like

```python
import numpy.ctypeslib as npct

array_1d_double = npct.ndpointer(dtype=np.double, ndim=1,
flags='CONTIGUOUS')

array_2d_double = npct.ndpointer(dtype=np.double, ndim=2,
flags='CONTIGUOUS')

c_func = napmo_library.angular_spherical_expansion

c_func.restype = None
c_func.argtypes = [POINTER(AngularGrid),
                   c_int,
                   c_int,
```

```
array_1d_double,
array_2d_double]
```

So the function can be called as

```
c_func(byref(self.angular_grid), lmax, size, f, output)
```

Note the special case of the `array_2d_double` type. Even if it is a two dimensional numpy array, this will be used in C code as 1D array. This procedure is only needed if the new functions from the C library need to be called from Python.

Additional details on the C implementation and parallelization will be given in Chapter 4 and additional details on the application programming interface (API) can be found in ref [17].

3.2 RESULTS

In order to check the correctness of the implementation, several test cases have been designed for both numerical multi-center integration and for two electron Coulomb integration.

3.2.1 *Multi-center integration*

As a first test case we choose the integrand

$$F(\mathbf{r}) = \rho \tag{33}$$

for diatomic systems H_2, Li_2, Be_2, B_2, C_2, N_2 and O_2 at their equilibrium internuclear separation.

For convenience the total density ρ has been modeled as the sum of free atomic densities optimized at HF level of theory [9] using 6−31+G** basis-set [12].

Table 2 shows the results on the calculation using small grids.

Even though small grids were used in this test, the integration provides an accuracy of five significant figures. Increasing the grid size will produce more accurate results.

In order to explore the error among different grid sizes, the integration of ρ was performed for the system O_2 using all possible combination among radial and angular points, the results can be seen in the figure 3.

3.2.2 *Two electron Coulomb integration*

As a second test, we consider the diatomic molecule of H_2. In particular, the two-electron Gaussian integral over a minimal basis STO−3G [11]. Two-electron integration over GTOs can

System	Exact	Radial Points		
		20	25	30
H_2	2.0	1.99995	1.99998	2.00000
Li_2	6.0	6.00011	5.99999	6.00005
Be_2	8.0	7.99931	7.99975	8.00001
B_2	10.0	9.99813	9.99967	10.00002
C_2	12.0	11.99808	11.99991	12.00003
N_2	14.0	13.99942	14.00034	14.00010
O_2	16.0	16.00007	16.00007	15.99991

Table 2: Numerical integration of $\rho(\mathbf{r})$ for different diatomic molecules. 110 angular points were used in this calculation.

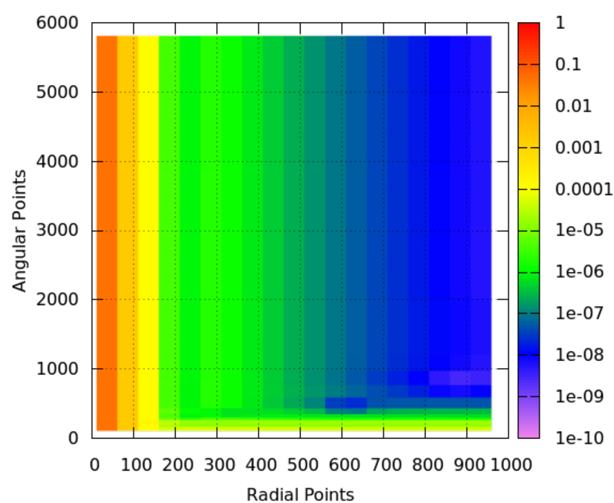

Figure 3: Averaged error calculating $\int \rho(\mathbf{r})$ for O_2 molecule with different grid sizes. Color in log scale.

Mesh[a]	Coulomb Integral (a.u.)
40 × 50	0.77561
60 × 110	0.77506
100 × 146	0.77477
180 × 170	0.77465
280 × 194	0.77462
Exact[b]	0.77461

[a] Radial × angular points per atom.
[b] Calculated with LIBINT

Table 3: Two-electron Coulomb integration over STO−3G basis-set for H_2 molecule.

be performed analytically, so the exact value of the integration can be calculated by using well known libraries. In our case we have calculated the analytical integrals using LIBINT library [21].

The results are presented in table 3. As shown in the table, the accuracy of the integration is around 10^{-5}, for small grids. As in the previous case of ρ integrand, the accuracy can be improved by increasing the number of grid points.

Once we have checked the correctness of nAPMO results, we proceed to discuss the performance and the parallelization strategy in the next chapters.

4

PERFORMANCE AND PARALLELIZATION

So far, we have discussed the theory behind nAPMO, and the philosophy used to implement the code. In this Chapter we will discuss about the strategy adopted to achieve an optimal parallelization with OpenMP (OMP) and CUDA.

Fist to all let us to describe the configuration used to perform all the calculations and benchmarking.

4.1 TESTING CONFIGURATION AND ENVIRONMENT

4.1.1 *System*

All the test were performed in the Ulysses-SISSA cluster, Intel Xeon E5−2680 V2 2.80 GHz Ivy Bridge processor 10 cores 20 Threads.

CUDA benchmarks were carried out on a NVIDIA GeForce GTX 860M Maxwell compute capability 5.0 (5 Multiprocessors, 128 CUDA Cores/MP), and on a NVIDIA Tesla K20m compute capability 3.5 (13 Multiprocessors, 192 CUDA Cores/MP).

4.1.2 *Compilers*

The C part of the code is compiler-independent, that means that the code can be compiled with any C compiler. GNU GCC compiler from version 4.4.7 to 5.0 and Intel compiler version 16.0.1 were tested. However, the results reported in this work are those obtained with the binary generated with GCC 4.4.7 compiler along with the following flags:

- −O2 level of optimization.
- −fPIC to produce position-independent code.
- −ffast-math
- −fompenmp for OpenMP support in OMP and CUDA.
- −shared for the linking process to generate the shared library.

The CUDA code was compiled with NVCC compiler version 7.5 with the same flags used on GCC compiler and the following additional flags:

- -arch=sm_50 for Maxwell card
- -arch=sm_35 for Tesla card

4.2 MULTI-CENTER INTEGRATOR

In this section we will analyze the performance of the multi-center numerical integrator, and its subsequent parallelization with OpenMP and CUDA. To start this analysis, let us first describe the general algorithm adopted to perform the integration. Figure 4 shows the algorithm implemented.

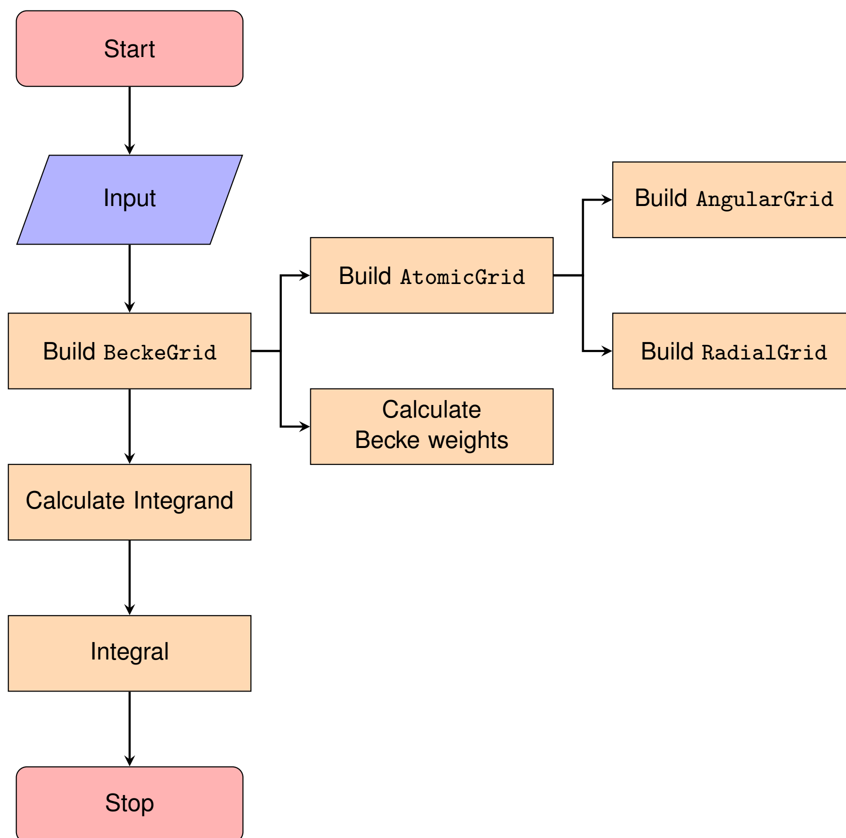

Figure 4: Algorithm to perform multi-center numerical integration

As can be seen in the flowchart, the integration is composed of four main steps; load data, build the molecular grid (BeckeGrid), calculate the integrand $F(\mathbf{r})$ in all points of the grid and, finally, perform the reduction of the integral (Integrate).

Having explained the implemented strategy to perform the integration, we will analyze the performance of the serial code in order to identify suitable spots were the code should be parallelized to increase the performance.

4.2.1 *Serial performance*

The test to analyze the performance of the serial code is the calculation of $\int \rho(\mathbf{r})$ for the diatomic molecule of H_2 using 6−31+G** basis-set. The size of the grid is 1202 angular by 1000 radial points. The following is the profiling of $\rho(\mathbf{r})$ integration:

```
85036 function calls (82212 primitive calls) in 2.575 sec

Ordered by: internal time
ncalls  tottime  cumtime  filename:lineno(function)
    1    1.986    1.987   density.py:31(density_full_...)
    1    0.155    0.155   becke.py:85(_becke_weights)
    1    0.124    0.356   becke.py:41(__init__)
    2    0.074    0.078   atomic.py:29(__init__)
    2    0.057    0.119   atomic.py:89(integrate)
```

`tottime` is the total time spent in the given function (and excluding time made in calls to sub-functions), while `cumtime` is the cumulative time spent in this and all sub-functions (from invocation till exit).

As expected, the most time consuming part is the calculation of the integrand, which for this particular test is `density.py`. Subsequently, another important amount of time is spent in the the calculation of Becke weights and in the construction of the molecular and atomic grids.

The most obvious target for parallelization is the calculation of the integrand. However, since the integrator must be generic, i.e. for any arbitrary integrand, the construction of the grids and the calculation of Becke weights as well will be parallelized.

4.2.2 *OpenMP implementation*

Having delimited the scope of the parallelization, we will start to explain the implementation of the parallelization and its results.

4.2.2.1 *Grids construction*

For the molecular grid construction, as observed in figure 4, the algorithm includes the construction of `AngularGrid` and `RadialGrid` objects which compose the `AtomicGrid` instance. The pseudo code to build the `AtomicGrid` object is

```
loop from i = 0 to AngularGrid.npoints
    loop from j = 0 to RadialGrid.npoints
        each grid coordinate is:
        AngularGrid.point[i] * RadialGrid.point[j]
            + atomic origin
        each grid point weight is
        AngularGrid.weight[i] + RadialGrid.weight[j]
```

so the OpenMP directive `#pragma omp parallel for` can be added above the loop over the number of angular points, which is typically greater than the number of radial points in most of the cases,

```
#pragma omp parallel for default(shared) private(i, j, ...)
loop from i = 0 to AngularGrid.npoints
    loop from j = 0 to RadialGrid.npoints
        each grid coordinate is:
        AngularGrid.point[i] * RadialGrid.point[j]
            + atomic origin
        each grid point weight is
        AngularGrid.weight[i] + RadialGrid.weight[j]
```

The calculation of Gauss-Chebyshev points (`RadialGrid`) has been parallelized in a similar way, over the number of radial points, as well as the Lebedev grid (`AngularGrid`), but of course, the latter over the number of angular points.

4.2.2.2 *Becke weights*

The calculation of the Becke grids is an expensive calculation since for each grid point all centers in the system must be taken into account. Figure 5 offers a graphical description of the geometry used to calculate the weights. A and B are centers, i.e. atoms, and i corresponds to a grid point. r_{iA} and r_{iB} corresponds to the distance from i to centers A and B respectively, while r_{AB} is the inter-nuclear distance.

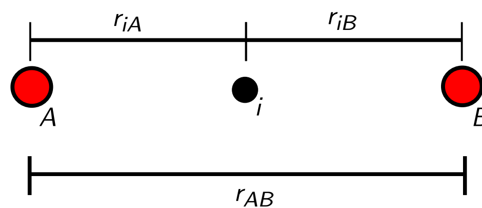

Figure 5: Geometrical description for Becke-weight calculation of point i in a system with centers A and B

The quantities r_{iA}, r_{iB} and r_{AB} are used to calculate the coordinate μ_{AB} in the following manner

$$\mu_{AB} = \frac{r_{iA} - r_{iB}}{r_{AB}} \tag{34}$$

This coordinate is then passed to a function called `step function`, which returns the contributions needed to calculate the weights. More detailed description of the form of the `step function` can be found in reference [2].

The algorithm to calculate those μ_{AB} coordinates and in consequence the Becke weights is

```
// Calculate internuclear distances
for i = 0 to number of atoms
    for j = 0 to i - 1
        R_ij[counter] = atomic origin[i] - atomic origin[j]

// Calculate the Becke weights
for atom = 0 to number of atoms
    for point = 0 to number of grid points (npoints)
        for i = 0 to number of atoms
            for j = 0 to number of atoms
                if i == j: continue
                r_i = distance point - i
                r_j = distance point - j
                calculate mu_ij
                calculate s = s(mu_ij) //step function
                cell_function *= s
            sum += cell_function
            if iatom == atom: P = cell_function
        weight[atom * npoints + point] = P / sum
```

Three of the four loops are over the number of centers (atoms) in the system, which of course depends on the system size, while the other one is over the number of grid points of a given center. Regardless of the system size, we can say that the loop over the number of grid points is greater than the number of centers in the system. In consequence, the loop over the number of grid points has been chosen for parallelization. The OpenMP directive added to achieve this parallelization is

```
#pragma omp parallel for default(shared)                  \
    firstprivate(atom, npoints, idx) private(point, i, j, \
    r_i, r_j, mu_ij, s, cell_function, sum, P)
```

4.2.2.3 *Calculation of the Integrand*

As said before, the parallelization strategy for the calculation of the integrand must be independent of the integrand itself. Because of this, the problem reduces to the calculation of any arbitrary function for each point i in the grid. Such as in the previous case of Becke weights calculation, the parallelization will be over the number of grid points.

From the above discussion it can be concluded that, in order to calculate efficiently integrals with this method, any new integrand must be implemented and parallelized. Details on the implementation of new functions inside the nAPMO package can be found in Chapter 3.

4.2.2.4 *Results*

We now show the performance improvements provided by the OpenMP parallelization described in the previous sections. As in the serial performance analysis, the integrand is $\rho(\mathbf{r})$ but in this case for the diatomic molecule of O_2. To explore the scaling across different grid sizes, four different grids were chosen. Figure 6 shows the scaling of the time and speedup versus the number of threads for each grid.

As can be seen in the plot, linear scaling is reported up to 20 threads for the biggest grid. However, for smaller and more realistic grids, the speedup scales linearly up to 8 or 4 threads for the smallest one. The reason of this behavior is related with the ratio between the serial and parallel zones and its dependency on the grid size.

Table 4 shows the OpenMP analysis performed with Intel VTune amplifier [6] for difference grid sizes. As can be seen in the table, the serial part of the code becomes more relevant for small grid sizes, affecting the expected speedup. The theoretical speedup S has been calculated using the Amdahl's Law [1]

$$S = \frac{1}{r_s + \frac{r_p}{n}} \tag{35}$$

where $r_s + r_p = 1$ and r_s and r_p represents the ratio of the sequential and parallel portion in one program executed over n threads.

	Grid Size			
OpenMP analysis	a	b	c	d
Serial Time (%)	69.2	35.9	23.5	13.0
Parallel Region Time (%)	30.8	64.1	76.5	87.0
Theoretical speedup	1.37	2.28	3.02	4.19
Achieved speedup	1.77	2.66	2.90	3.47

Table 4: Intel VTune amplifier OpenMP (8 threads) analysis for the integration of $\rho(\mathbf{r})$ molecule O_2. 6−31+G** basis-set. Grid sizes (angular × radial points): a: 1202×100, b: 1202×500, c: 1202×1000, d: 5810×1000.

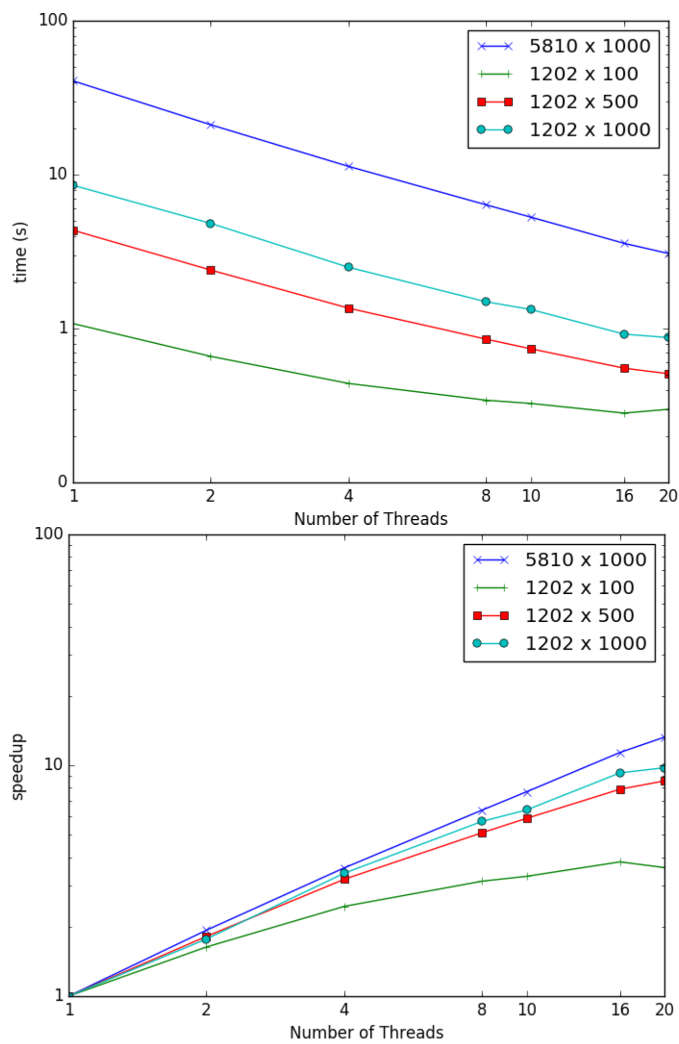

Figure 6: Time and speedup vs number of threads for the integration of $\rho(\mathbf{r})$ for O_2 molecule using different grid sizes. Basis-set used 6−31+G**.

In the case of small grids (*a* and *b*) the speedup is effected by caching effects, explaining why the the achieved speedup is higher than the theoretical.

4.2.3 *CUDA implementation*

Consistently with the profiling results for the serial code, Section 4.2.1, the calculation of Becke weights, the calculation of the integrand and the integration were chosen to be parallelized with CUDA. The implementation has been done in double precision to preserve accuracy.

Since memory management is critical in CUDA programming model [7], the data structure used in CUDA implementation is different to the C/OpenMP version. The first consideration is that device's global memory is accessed via 32−, 64−, or 128−byte memory transactions, so the data types used must meet those alignment conditions (i.e must be multiple of those transaction sizes) to avoid memory transfer overhead.

CUDA toolkit provides a set of data structures that meet with the aforementioned requirements. One of those data structure is `double2`. The following code illustrates this data structure.

```
struct __align__(16) {
    double x;
    double y;
};
```

That kind of structures can improve the efficiency of memory access as fewer accesses are needed for the same amount of data handled. Along `double2` another built-in vector types are available, such as `int2`, `float4`, among others.

The proposed data structure for `BeckeGrid` using built-in vector CUDA data types is as follows

```
struct _becke_cuda {
      int2 gridDim;    // Dim of the grid (radial, angular)
      double2 *xy;     // coord x and y.
      double2 *zw;     // coord z and weights.
};
```

Additional memory management considerations were taken into account, such as the use of shared memory. For instance, in the case of atomic additions, the performance improves dramatically by doing the reduction among the threads within a block using a shared array and then perform the reduction among the blocks through global memory. The following code illustrates the described two-step reduction.

```
const unsigned int i = __umul24(blockIdx.x, blockDim.x)
                          + threadIdx.x;
__shared__ double temp[THREADS_PER_BLOCK];
__shared__ double sum_block;
```

```
temp[threadIdx.x] = 0.0;
sum_block = 0.0;

if (i < size) {
temp[threadIdx.x] += work[i] * weights[i];
}
__syncthreads();
atomicAdd(&sum_block, temp[threadIdx.x]);

__syncthreads();
if (threadIdx.x == 0) {
atomicAdd(integral, sum_block);
}
```

The shared memory was also used to reduce the global memory load operations of intensively used data or data used for all threads simultaneously. For example,

```
__shared__ double buffer[THREADS_PER_BLOCK];

function_value = 0.0;
for (int i = 0; i < n_cont; ++i) {
  for (int j = threadIdx.x; j < n_cont; j+=THREADS_PER_BLOCK) {
    buffer[j] = dens[i * n_cont + j];
  }
  __syncthreads();
  temp_val = 0.0;
  for (int j = 0; j < n_cont; ++j) {
    temp_val += basis_val[j] * buffer[j];
  }
```

Finally, splitting the code into smaller kernels was done to reduce the amount of registers needed for each kernel in order to improve the occupancy on the device. In the following, we show the metrics on the performance of this implementation for the particular case of multi-center numerical integration.

4.2.3.1 *Results*

The system used for this calculations is the usual integration of $\rho(\mathbf{r})$ for the diatomic molecules H_2 and O_2. Different grid sizes were used. Three global kernels are our main focus here, `becke_weights_kernel`, `atomic_grid_integrate_kernel` and `density_gto_kernel` which corresponds to Becke weights calculation, integration and calculation of the integrand.

Before showing the results on timing and speedup for the integration, we will show some detailed information about the performance of the kernels in the device. Table 5 contains several performance metrics for all kernels.

Metric	Kernel		
	Weights	Integrate	Density
Multiprocessor Activity	99.99	99.97	99.99
Achieved Occupancy	73.01	61.77	95.20
Global Memory Load Efficiency	70.00	97.73	84.87
Global Memory Store Efficiency	100.00	100.00	100.00

Table 5: Percentage of several NVIDIA profiler metrics for calculation of $\int \rho(\mathbf{r})$ for O_2 molecule. $6-31+$G** basis-set used. Grid size: 5810 angular by 1000 radial points.

Occupancy refers to the utilization of the device in terms of the number of concurrent blocks that are being executed by each multiprocessor. Optimize occupancy can impact performance since greater occupancy means improving the utilization of the computing capacity of the device. An occupancy greater than 50% is considered to be acceptable. From table 5 it is clear that all kernels have high occupancy level.

Additionally, The load/store efficiency from global memory is perhaps the most important metric to be optimized. Latency of the global memory is so high that any optimization on the memory throughput will improve performance. The greater efficiency the better performance. The lowest memory load efficiency was evidenced in the Becke weights kernel because of precomputed data that can not be accessed in a coalesced way. In other kernels the memory efficiency is high, which means that, the kernels offer a performance near to the actual device limit.

Figure 7 shows the time and speedup reached by the implementation for several grid sizes and two different systems. It can be seen that the speedup depends on the problem size, that is, the smaller grid size the smaller speedup. In the case of medium (1202×500) to big grid sizes (5810 $\times$ 1000 $-$ 2000) it can be reported a speedup near to 30$\times$.

Compared to OpenMP, figure 8 shows that CUDA speedup is greater than OpenMP's for medium to large grid sizes. For the particular case of small grids (not in the plot), such as, 194 $\times$ 50 grid, the performance is better in OpenMp than in CUDA because of the overhead of data transfer which can not be compensated by the computational speedup.

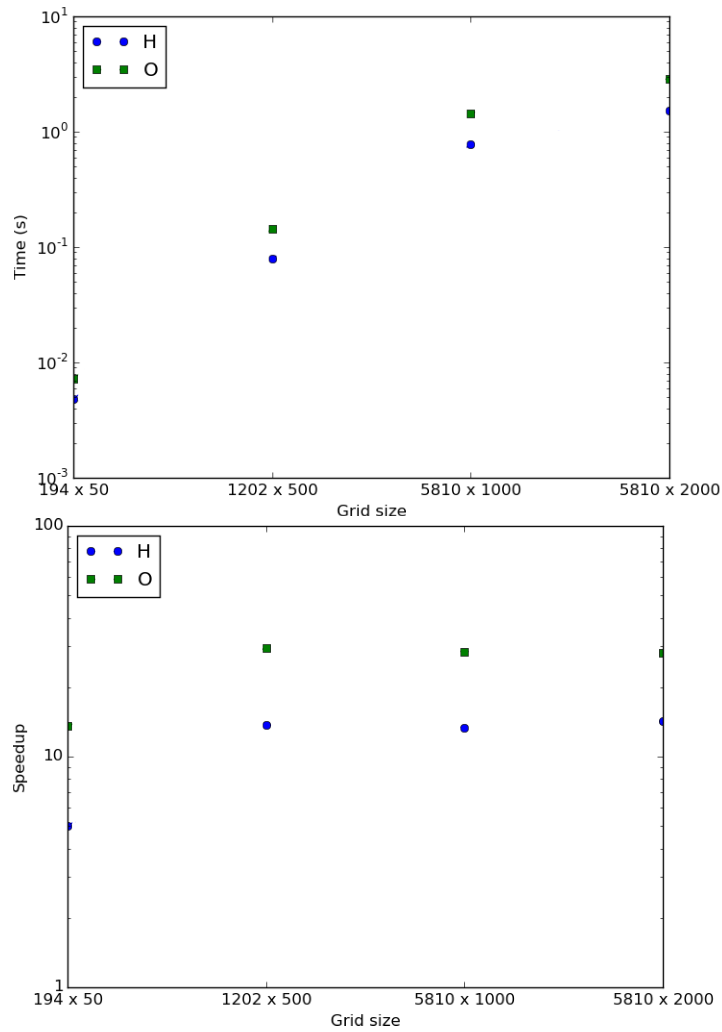

Figure 7: Time and speedup vs Grid size for the integration of $\rho(\mathbf{r})$ for H_2 and O_2 on CUDA device. Basis-set used 6−31+G**. Grid size, angular × radial points.

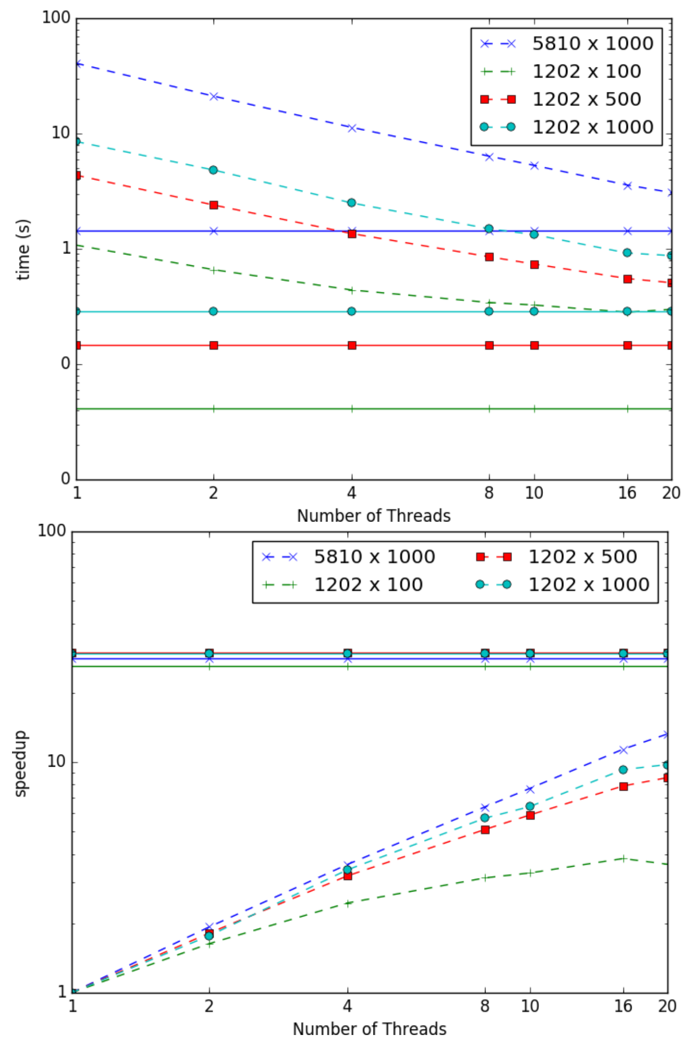

Figure 8: Time and speedup vs number of threads for the integration of $\rho(\mathbf{r})$ for H_2 molecule using different grid sizes. The dashed line corresponds to OpenMP and solid to CUDA. Basis-set used 6−31G**.

4.3 TWO-ELECTRONIC COULOMB INTEGRATION

In this section we will discuss about the performance of the calculation of two-electron coulomb integrals. As explained in Chapter 2 the calculation of this particular integral includes the solution of Poisson's equation for the potential of a given charge distribution. The algorithm to calculate the two-electron Coulomb integrals is shown in Figure 9.

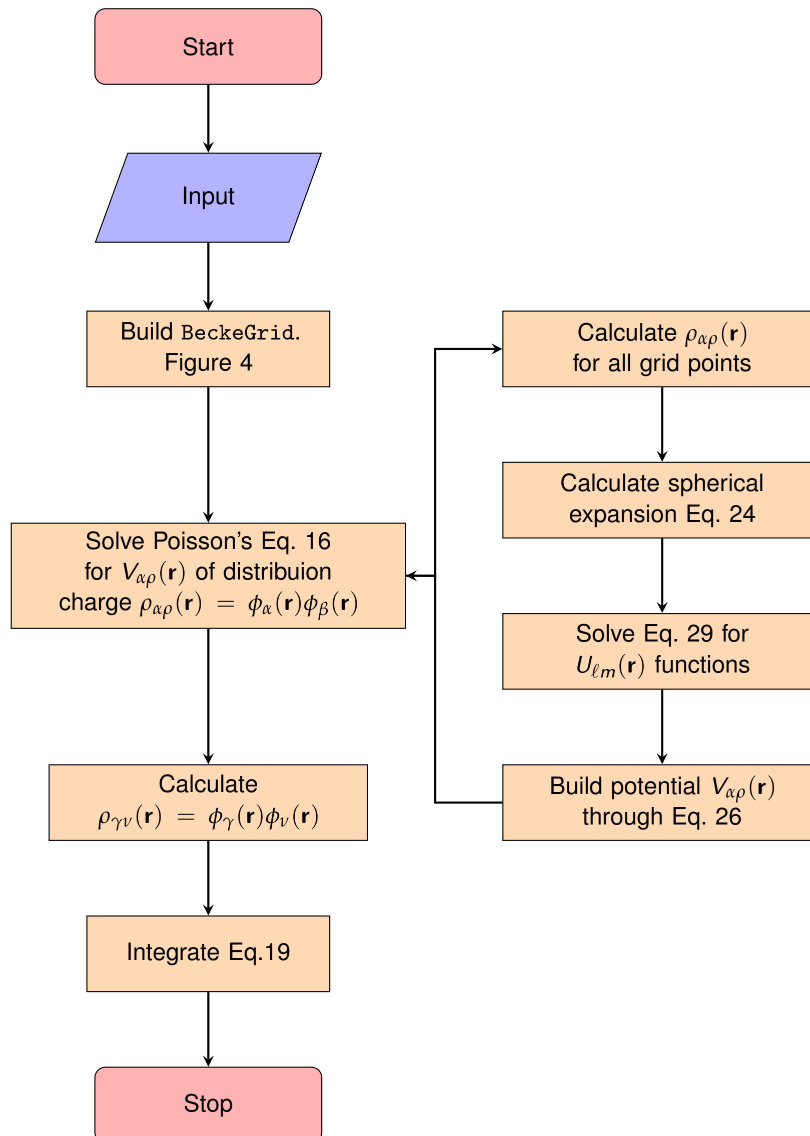

Figure 9: Algorithm to perform two-electron Coulomb numerical integration.

As can be seen, the two-electron coulomb integration follows the same procedure of multi-center numerical integration. In other words, the procedure called *calculate the integrand* in

Figure 4 is in this case the calculation of $V_{\alpha\beta}$ plus the calculation of $\rho_{\gamma\nu}(\mathbf{r})$.

It is important to point out that the solution of eq. 29 for $U_{\ell m}(\mathbf{r})$ functions is done by using the finite differences method. This method involves the solution of a linear system $Ax = b$ for x values ($U_{\ell m}(\mathbf{r})$) using a tridiagonal matrix A of coefficients. The matrix A is sparse, so direct or iterative sparse solvers can be used to improve the performance.

4.3.1 *Serial performance*

We will focus or attention in the two-electron Coulomb integral

$$I_{\alpha\beta,\gamma\nu} = \int\int \frac{\phi_\alpha(1)\phi_\beta(1)\phi_\gamma(2)\phi_\nu(2)}{r_{12}} d^3r_1 d^3r_2 \tag{36}$$

of the H_2 molecule. (i) refers to electron i. C_μ and ϕ_μ correspond to the μth coefficient and function of the linear combination

$$\chi_i = \sum_{\mu=1}^{k} C_{\mu i}\phi_\mu \tag{37}$$

of the one-electron wave-function χ_i. In this case the basis-set used is the STO$-$3G. Different grid sizes have been used. The following is the profiling using a grid of 194 angular $\times$ 500 radial points.

```
328557 function calls (322104 primitive calls) in 0.500 sec

Ordered by: internal time
ncalls  tottime cumtime filename:lineno(function)
66/56    0.056   0.062  {built-in method _imp.create_dynamic}
  360    0.037   0.037  {built-in method marshal.loads}
  144    0.034   0.034  {built-in method scipy.sparse.linalg}
    1    0.028   0.028  atomic.py:52(spherical_expansion)
    6    0.023   0.024  primitive_gaussian.py:71(compute)
```

The `scipy.sparse.linalg` function refers to the sparse solver from the SciPy library. We remind that both integration procedure, and the construction of the grids have been already parallelized, see previous section.

From the previous profiling results it can be seen that there is not any advisable bottleneck. However, let us try again with a bigger grid (1202 $\times$ 500).

```
548071 function calls (540106 primitive calls) in 3.948 sec
```

```
Ordered by: internal time
ncalls  tottime cumtime filename:lineno(function)
    1    1.595   1.595  atomic.py:71(evaluate_expansion)
    1    1.002   1.002  atomic.py:52(spherical_expansion)
  900    0.177   0.177  {built-in method scipy.sparse.linalg}
    6    0.143   0.143  primitive_gaussian.py:71(compute)
66/56    0.056   0.062  {built-in method _imp.create_dynamic}
```

In this case there are two identifiable hotspots suitable for parallelization, `evaluate_expansion` which corresponds to build the potential through eq. 26, and the `spherical_expansion`.

4.3.2 *OpenMP performance*

Proper OpenMP parallelization was done in the `evaluate_expansion` and `spherical_expansion` routines. Figure 10 shows the result of the performance.

The implementation reports scalability up to 16 cores, were the serial part of the code starts to become more and more relevant. Additionally it can be seen a speedup from 2.5 to 4.0 depending on the grid size.

Further optimizations, such as, the utilization of a external sparse solver with OpenMP and CUDA support can improve the performance reported here.

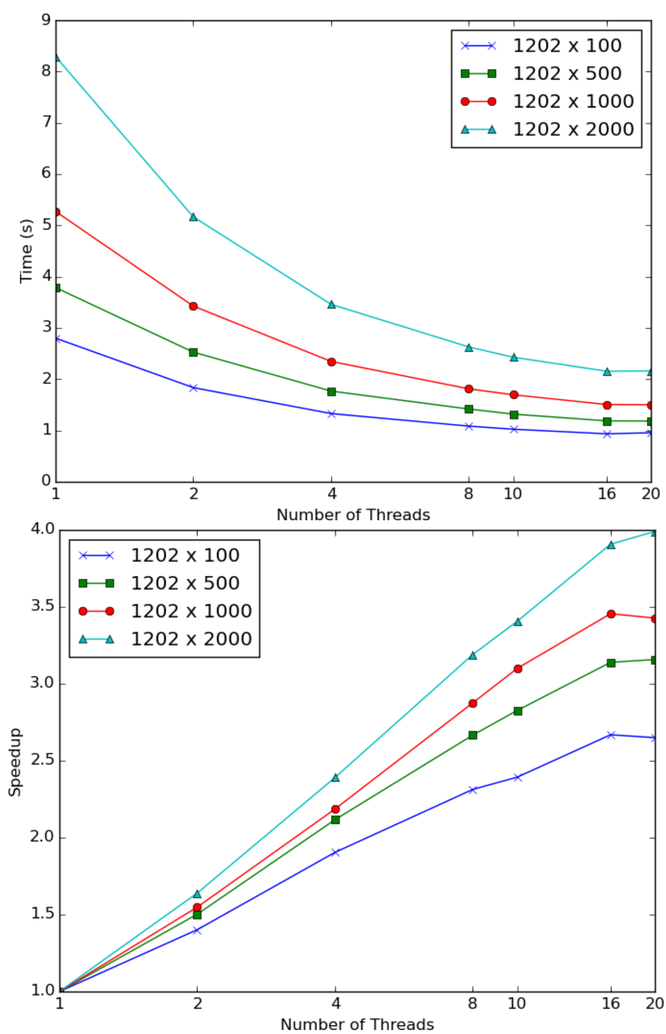

Figure 10: Time and speedup vs number of threads for two-coulomb integration of H_2 STO−3G basis-set at different grid sizes.

5

SUMMARY AND CONCLUSIONS

nAPMO package has been developed as a platform to develop a basis-set-free suite for Quantum chemical calculations. With the aim to develop a basis-set-free Hartree-Fock implementation, we have developed the multi-center numerical integrator and the solver of the Poisson's equation for the calculation of two-electron Coulomb integrals.

To check the correct implementation of the code, the integral $\int \rho(\mathbf{r})$ has been calculated for diatomic molecules from $Z = 1$ to $Z = 8$ using 6$-$31+G** basis-set. Additionally the calculation of two-electron integrals has been calculated for the STO$-$3G basis-set of the Hydrogen molecule. The calculation of the two-electron coulomb integrals has been compared against the analytic result obtained with LIBINT library [21].

Python language was chosen to be the main language of the program due to the high scripting facilities and tools that this language offers. The effects of Python overhead is addressed with the implementation of a C library which contains most of the numerical procedures.

The C library, called `libnapmo` has been parallelized with OpenMP and CUDA. The OpenMP parallelization reports a speedup of 3 to 20$\times$ for the integration of $\rho(\mathbf{r})$ and up to 4$\times$ for the two-electron coulomb integration. The CUDA implementation reports a speedup from 4 to 30$\times$ depending on the atom and the grid size, for the integration of density ρ.

All performance calculations were obtained using medium to big grid sizes. In the case of two-electron Coulomb integrals, small grids are enough to get a precision of 10^{-5}. The time spent for this kind of calculations is of the order of 10^{-2} seconds, reason for which further parallelization with CUDA was not taken into account.

Additional optimizations can be implemented, including the use of a high performance sparse linear algebra solver, and implementation in CUDA of the calculation of real spherical harmonics. This implementation will be done in future work.

5.1 FUTURE WORK

This work is also part of a PhD thesis which has among his goals the implementation of the basis-set-free Hartre-Fock molecular orbital theory. To achieve this the following work is missing.

- Implementation in CUDA of spherical harmonics.
- Use of a external solver for the linear algebra problem (already done, but not properly tunned)
- Implementation of the self-consistent field with the integrals implemented here.
- Further optimizations and parallelizations.

6

BIBLIOGRAPHY

[1] Gene M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, 2013.

[2] Axel D. Becke. A multicenter numerical integration scheme for polyatomic molecules. *The Journal of Chemical Physics*, 88(4):2547, 1988.

[3] Axel D. Becke and Ross M. Dickson. Numerical solution of Poissons equation in polyatomic molecules. *The Journal of Chemical Physics*, 89(5):2993, 1988.

[4] Florian a. Bischoff and Edward F. Valeev. Low-order tensor approximations for electronic wave functions: Hartree-Fock method with guaranteed precision. *The Journal of chemical physics*, 134(10):104104, March 2011.

[5] Beatriz Cordero, Verónica Gómez, Ana E Platero-Prats, Marc Revés, Jorge Echeverría, Eduard Cremades, Flavia Barragán, and Santiago Alvarez. Covalent radii revisited. *Dalton Transactions*, (21):2832, 2008.

[6] Intel Corporation. Intel VTune Amplifier 2016. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`. Accessed November 24, 2015.

[7] NVIDIA Corporation. CUDA Toolkit Documentation v7.5. `http://docs.nvidia.com/cuda/index.html`. Accessed November 25, 2015.

[8] N Flocke and V Lotrich. Efficient Electronic Integrals and their Generalized Derivatives for Object Oriented Implementations of Electronic Structure Calculations. *Journal of computational chemistry*, 2008.

[9] Roberto Flores-Moreno, Edwin Posada, Félix Moncada, Jonathan Romero, Jorge Charry, Manuel Díaz-Tinoco, Sergio A. González, Néstor F. Aguirre, and Andrés Reyes. LOWDIN: The any particle molecular orbital code. *International Journal of Quantum Chemistry*, 114(1):50–56, jan 2014.

[10] Martin Head-Gordon and John A Pople. A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations. *The Journal of Chemical Physics*, 89(9):5777, 1988.

[11] W J Hehre. Self-Consistent Molecular-Orbital Methods. I. Use of Gaussian Expansions of Slater-Type Atomic Orbitals. *The Journal of Chemical Physics*, 51(6):2657, 1969.

[12] W J Hehre. SelfConsistent Molecular Orbital Methods. XII. Further Extensions of GaussianType Basis Sets for Use in Molecular Orbital Studies of Organic Molecules. *The Journal of Chemical Physics*, 56(5):2257, 1972.

[13] J.D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, May 2007.

[14] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Accessed December 1, 2015].

[15] V. I. Lebedev and D. N. Laikov. A Quadrature Formula for the Sphere of the 131st Algebraic Order of Accuracy. *Doklady. Mathematics*, 59(3):477–481.

[16] Sergio a. Losilla, Mark a. Watson, Alán Aspuru-Guzik, and Dage Sundholm. Construction of the Fock Matrix on a Grid-Based Molecular Orbital Basis Using GPGPUs. *Journal of Chemical Theory and Computation*, 11(5):2053–2062, May 2015.

[17] Edwin Posada. nAPMO's Documentation. `http://efposadac.github.io/nAPMO/index.html`. Accessed November 18, 2015.

[18] Toru Shiozaki and So Hirata. Grid-based numerical Hartree-Fock solutions of polyatomic molecules. *Physical Review A - Atomic, Molecular, and Optical Physics*, 76(4):040503, oct 2007.

[19] Alex Sodt, Joseph E Subotnik, and Martin Head-Gordon. Linear scaling density fitting. *The Journal of Chemical Physics*, 125(19), 2006.

[20] Attila Szabo and Neil S. Ostlund. *Modern quantum chemistry : introduction to advanced electronic structure theory*. Dover Publications, New York, reprinted edition, 1996.

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

[22] S. van der Walt, S.C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.