UNIVERSITY OF COPENHAGEN

Massively lockstep-parallel algorithms for full-isomer space quantum chemistry

Masters

Anton M. Nielsen & Asmus Tørsleff

Supervised by Assoc. Professor, Ph.D. James Avery
Co-supervised by Professor, Ph.D., Dr. Scient. Kurt V. Mikkelsen

Department of Computer Science
Quantum Information Science

August 15, 2025

Masters

Massively lockstep-parallel algorithms for full-isomer space quantum chemistry

By Anton M. Nielsen & Asmus Tørsleff

Supervised by Assoc. Professor, Ph.D. James Avery
Co-supervised by Professor, Ph.D., Dr. Scient. Kurt V. Mikkelsen

Date of submission: August 15, 2025

University of Copenhagen
Faculty of Science
Department of Computer Science
Quantum Information Science
Universitetsparken 5
2100 Copenhagen Ø

Acknowledgements

We would like to thank our supervisor Assoc. Professor, Ph.D. James Avery for guidance and advise during this project.

We would like to thank our co-supervisor Professor, Ph.D., Dr. Scient. Kurt V. Mikkelsen for advise and sharing his chemistry expertise.

We would like to thanks Jonas..

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Individual Contributions to the Project

Table of Contents

Part I

Thesis

1 Introduction

As part of James E. Avery's efforts to develop an efficient screening pipeline for fullerenes and potentially fulleroids we will in this report detail our efforts in porting parts of the xtb program by Grimme et al to SYCL code. The goal is a highly optimised and fully lockstep-parallel implementation of the electronic structure calculations from the GFN2-xTB method. A previous thesis by la Cour provides an efficient lockstep-parallel implementation of a forcefield method for computing geometric structures of fullerenes, which we will take to be our input.

The mentioned screening pipeline would enable the search of entire isomer spaces for fullerenes with certain properties such as a low lowest energy state indicating a stable isomer.

A fullerene is a molecule consisting only of carbon atoms connected in 12 pentagons and enough hexagons to create a hollow structure. As we increase the number of atoms the isomer space quickly grows leading to very slow search times. We aim to provide a quick and relatively accurate method for discarding large unpromising parts of the isomer space before searching with more accurate methods.

Fulleroids are essentially an extension to fullerenes where we allow n-gons instead of only penta- and hexagons, as long as we can still create a closed shape.

After a literature review we settled on the Geometry, Frequency, Noncovalent, extended Tight Binding (GFNn-xTB) family of methods as they are relatively accurate and quite fast at predicting electronic structures to a reasonable accuracy. We hope that this inherent speed will aid in getting good though-put after the transformation to a lockstep-parallel version. The GFNn-xTB metods come in iterations. GFN1 is the first and lays the ground work for the later iterations. It does however rely on element pairwise specific constants. In GFN2 this has been changed in favour of only element specific parameters. GFN0 is a more approximate and faster version of GFN2. And GFN-FF takes this trade-off further as this is a forcefield method which is parametrised using the insights (and parameters) gained from the other GFN iterations.

Forcefield methods save on computing all the pairwise interactions between atoms in a molecule and instead use efficient rules to lump atoms together in predictable clumps which then interact with other clumps. This can save tremendous effort.

Specifically GFN2 seemed most promising for our purposes as it is more accurate than GFN0 and simpler than GFN1, and if it is not fast enough would be relatively easy to then implement GFN0. GFN-FF was not considered suitable due to us wanting to see if it could be fast enough without defaulting to a forcefield method.

Lockstep-parallelisation is a paradigm best suited for GP-GPU. It takes advantage of the fact that GPUs operate more efficiently when all the cores are doing the same operations in a predictable fashion. This essentially is a step beyond data parallelism. We are not only operating on the same data across cores, but also doing the exact same steps. This means no conditionals with a data dependent evaluation. It is fine to have a loop that runs five times, opposed to say `data[coreId]` times.

Code Structure

2

The code repository can be found at: `https://github.com/AceMouse/xTB-math`.

```
xTB-math/
├── bin2xyz/
│   ├── bin2xyz.cpp ..... converter from float64 coords into xyz format
│   └── C200_10000_fullerenes.float64 ..... 3d coords for 10k fullerenes
├── flake.lock
├── flake.nix ..... conventional structure for Nix inputs and outputs
├── nix/ ..... Nix package definitions for xtb and its dependencies
│   ├── cpx.nix ..... CPCM-X - a solvation model
│   ├── numsa.nix ..... solvent accessible surface area calculation
│   ├── patches/ ..... patches for extracting data for validation
│   │   ├── dftd4/ ..... dispersion correction
│   │   │   ├── log_args_and_outputs.patch
│   │   │   └── use_gfn2.patch
│   │   └── xtb/
│   │       ├── log_args_and_outputs.patch
│   │       ├── log_electro.patch
│   │       └── log_utils.patch
│   └── xtb.nix ..... extended tight-binding program
├── README.md
├── report/
└── xtb-python/ ..... Python port of xTB paper and Fortran impl
    ├── basisset.py
    ├── blas.py
    ├── cmp_impls.py ..... validation tests against Fortran impl
    ├── data/
    │   ├── C200.xyz
    │   └── caffeine.xyz
    ├── dftd4.py ..... computation for dispersion correction
    ├── dftd4_reference.py ..... constants for dftd4
    ├── energy.py ..... various energy computations
    ├── fock.py ..... Fock matrix computation
    ├── gfn2.py ..... GFN2-xTB specific constants
    ├── lapack.py ..... Facade for LAPACK functions
    ├── scc.py ..... computation for self-consistent charges
    ├── slater.py ..... computation for slater determinants
    ├── util.py
    └── xyz_reader.py
```

The code repository can be found at: `https://github.com/Maroka-chan/xtb-gpu`.

```
xtb-gpu/
├── flake.lock
├── flake.nix
├── nix/
│   ├── nvhpc.nix ..... patching nvhpc to make nvfortran work on Nix
│   └── xtb.nix ..... xtb version 6.4.0 compiled with nvfortran
├── README.md
└── sycl/
    ├── build_SDQH0.cpp ..... incomplete SYCL impl for build_SDQH0
    ├── data_caffeine/ ..... test data for electro.cpp computed from caffeine
    │   ├── atomicGam.txt
    │   ├── dqsh.txt
    │   ├── dq.txt
    │   ├── H0.txt
    │   ├── jmat.txt
    │   ├── P.txt
    │   ├── shellGam.txt
    │   └── shift.txt
    ├── data_c200/ ..... test data for electro.cpp computed from a C200 fullerene
    │   ├── atomicGam.txt
    │   └── ...
    └── electro.cpp ..... SYCL impl for computing electrostatic energy
```

3 Theory

3.1 GFN2-xTB

GFN2-xTB is a part of the GFNn-xTB (Geometry, Frequency, Non covalent, eXtended Tight Binding) family of semi-empirical methods for computational chemistry.

The method gives good approximations for molecular geometries, vibrational frequencies, and non-covalent interaction energies and also does well on a variety other properties. Overall it strives to hit a balance between being accurate, close to the physics, general over a wide range of elements and not too computationally expensive.

This is achieved by approximating a true quantum mechanical simulation, using carefully chosen approximations and parameters. In contrast to forcefield methods that often operate on the level of atoms or even functional groups interacting, GFN2-xTB still treats the calculations at the level of individual orbitals in many places.

GFN2-xTB uses a Self-Consistent Charges (SCC) approach i.e. it makes an initial guess at a density matrix and an energy which it then iteratively refines until both have converged. We are mostly interested in the final energy.

3.1.1 Energy terms

The terms in GFN2-xTB that determine the energy are as follows

$$E = E_{rep} + E_{disp}^{D4'} + E_{EHT} + E_{\gamma} + E_{AES} + E_{AXC} + E_{\Gamma} + E_{Fermi} \tag{3.1}$$

We then have to also factor in the SCC. Let us consider the following code sketch in figure 3.1 which we will fill out in this chapter. The chapter will go over each term and add code snippets illustrating the mathematics in a more concrete way.

```
def get_GFN2_energy(atoms: list[int], positions : list[list[float]]) -> float:
    density = density_initial_guess(atoms)
    overlap, dipol_dipol, charge_quadrupol = overlap(atoms, positions)
    initial_hamiltonian = H0(atoms, positions)
    huckel_theory_matrix = H_EHT(atoms, positions, overlap, initial_hamiltonian)
    charges = mulliken_population_analysis(density,atoms)
    eigen_values = diagonalize(initial_hamiltonian, overlap) # HC=SCe
    E_repulsion = repulsion(atoms, positions)
    E_dispersion = D4Prime(charges, atoms, positions)
    E_huckel = Huckel(density, extended_huckel_theory_matrix)
    E_anisotropic = AES(charges, overlap, dipol_dipol, charge_quadrupol,
    ↪  positions)
    E_isotropic = IES(charges, positions)
    E_fermi = Fermi(...)
    E = E_repulsion + E_dispersion + E_huckel + E_anisotropic + E_isotropic +
    ↪  E_Fermi
    while not (energy_converged and densities_converged):
        # change densities along the gradients and
        # update everything using the new density
        # ...
        energy_converged = (E-E_new)**2 < tolerance
        densities_converged = error_squared(density, new_density) < tolerance
        E = E_new
        density = new_density
```

Figure 3.1.: Python code illustrating the main loop of a GFN2-xTB implementation. Line 2-16 computes the energy in a non SCC manner, line 15-23 iteratively improves the energy using SCC. Another way to describe this is that the zeroth iteration of SCC happens outside the loop as there is some setup required.

Let us define a small helper function to decrease the amount of indentation in the later code examples. This will allow us to easily loop over the orbitals and know which atom and shell each belongs to.

```
def get_orbitals(atoms: list[int]) -> list[tuple[int]]:
    orbitals = []
    for atom_idx,atom in enumerate(atoms):
        for subshell in range(number_of_subshells[atom]):
            l = angular_momentum_of_subshell[atom][subshell]
            for orbital in range(l*2+1):
                orbitals.append((atom_idx,atom,subshell,orbital))
    return orbitals
```

Figure 3.2.: Python code for generating a convenient list of orbitals to iterate though.

And some functions used when we want to signal to the reader that the python list we just created will stay a fixed size. Additionally we get to contain the python list comprehension syntax here.

```
def vector(n: int) -> list[float]:
    return [0.0 for _ in range(n)]

def square_matrix(n: int) -> list[list[float]]:
    return [[0.0 for _ in range(n)] for _ in range(n)]

def square_matrix_of_vectors(n: int, v:int) -> list[list[list[float]]]:
    return [[[0.0 for _ in range(v)] for _ in range(n)] for _ in range(n)]
```

Figure 3.3.: Python code for creating a n long vector, $n \times n$ matrix and $n \times n \times v$ tensor.

For the sake of clarity we will define the order of direction as follows:

```
x = 0
y = 1
z = 2
xx = 0
yy = 1
zz = 2
xy = 3
xz = 4
yz = 5
```

Figure 3.4.: Constants for the order of directions.

3.1.2 Constructing the initial density matrix

Many of the terms use the density matrix as part of their calculation and it is central to the SCC method. In the GFN2 paper the initial density matrix guess is formulated as a superposition of neutral atomic reference densities $P_0 = \sum_A P_{A_0}$. In simpler terms this means that we let P_0 be a diagonal matrix that is n by n where n is the total number of orbitals across the whole molecule. The values on the diagonal are the fractional number of electrons in the orbitals, the fractional occupations. They are reference densities so we get them in a table we can index into per subshell, thus we just have to divide the electrons evenly between the orbitals in the subshell. The number of orbitals in a subshell is $2l + 1$ where l is the quantum number corresponding to the angular momentum of the subshell.

```python
def density_initial_guess(atoms: list[int]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    fractional_occupations = square_matrix(len(orbitals))
    for orbital_idx,(_,atom,subshell,_) in enumerate(orbitals):
        l = angular_momentum_of_subshell[atom][subshell]
        orbitals_in_subshell = l*2+1
        electrons_in_subshell = reference_occupations[atom][subshell]
        electrons_per_orbital = electrons_in_subshell/orbitals_in_subshell
        fractional_occupations[orbital_idx][orbital_idx] = electrons_per_orbital
    return fractional_occupations
```

Figure 3.5.: Python code for computation of the initial density matrix P_0

For fullerenes the guess is simply all ones on the diagonal as `number_of_subshells[C] = 2`, `angular_momentum_of_subshell[C] = [0,1,0]` and `reference_occupations[C] = [1.0,3.0,0.0]`. Here `C=5` is the index for carbon. Thus `fractional_occupations` will contain a repeating series of $\frac{1}{0\cdot2+1}, \frac{3}{1\cdot2+1}, \frac{3}{1\cdot2+1}, \frac{3}{1\cdot2+1}$ on the diagonal.

3.1.3 Constructing the overlap matrix

We also need the overlap matrix, S, for several of the terms. An element in the matrix is computed in the following way.

$$S_{\nu\mu} = \langle\psi_\nu|\psi_\mu\rangle \qquad \forall\nu \in l \in A, \mu \in l' \in B \tag{3.2}$$

This is an integral between two Slater type orbitals (STOs). When we write $l \in A$ we mean to iterate though the subshells in A and take their angular momentum. $\nu \in l$ means that we iterate though the $2l+1$ orbitals in the subshell with quantum number $m \in \{-l,..,0,..,l\}$. See figure 3.5 for a code example.

```python
def overlap(atoms: list[int], positions: list[list[float]]) ->
    tuple[list[list[float]],list[list[list[float]]],list[list[list[float]]]]:
orbitals = get_orbitals(atoms)
S = square_matrix(len(orbitals))
D = square_matrix_of_vectors(len(orbitals),3)
Q = square_matrix_of_vectors(len(orbitals),6)
for idx_A, (atom_idx_A,atom_A,subshell_A,orbital_A) in enumerate(orbitals):
    for idx_B, (atom_idx_B,atom_B,subshell_B,orbital_B) in
        enumerate(orbitals):
        R_A = positions[atom_idx_A]
        R_B = positions[atom_idx_B]
        l_A = angular_momentum_of_subshell[atom_A][subshell_A]
        l_B = angular_momentum_of_subshell[atom_B][subshell_B]
        s,d,q = compute_STO_integral(...)
        S[idx_A][idx_B] = s
        for dir in [x,y,z]:
            D[idx_A][idx_B][dir] = d[dir]
        for dir in [xx,yy,zz,xy,xz,yz]:
            Q[idx_A][idx_B][dir] = q[dir]
return S,D,Q
```

Figure 3.6.: Python snippet illustrating construction of the overlap, dipol-dipol and charge-quadrupol matricies S, D and Q.

To compute the integral we need to remember that in GFN2-xTB an STO is approximated as a linear combination of gaussian type orbitals:

$$\begin{aligned}
|\psi_\nu\rangle =& \psi(\zeta_{A,l}, r - R_A) \\
=& N_{STO,l} |r - R_A|^{n-1} e^{-\zeta_{A,l}|r-R_A|} Y_l^m(r - R_A) \\
\approx& \sum_i^{N_{A,l}} c_{i,\nu} N_{GTO,l} (r_x - R_{Ax})^{l_x} (r_y - R_{Ay})^{l_y} (r_z - R_{Az})^{l_z} e^{-\alpha_{i,\nu}|r-R_A|^2} \\
=& \sum_i^{N_{A,l}} c_{i,\nu} \phi(\alpha_{i,\nu}, r - R_A) \\
=& \sum_i^{N_{A,l}} c_{i,\nu} |\phi_i\rangle
\end{aligned} \tag{3.3}$$

Where $N_{A,l}$ is an element and shell dependant constant, it is the number of GTOs used to approximate the STO. ζ and α are the slater and gaussian exponents. The contraction coefficient c is a fitted value, it is fitted with the assumption that $\zeta = 1$ and to get the value we will actually be using, we need to scale the fitted value by ζ^2. R_A is the position of atom A. $N_{STO,l}$ and $N_{GTO,l}$ are

normalisation terms. We will refer to the terms $(r_u - R_{Au})^{l_u} \quad \forall u \in x, y, z$ as polynomial prefactors. The approximation in terms of GTOs means we can express our overlap in terms of GTOs integrals instead:

$$S_{\nu\mu} = \sum_{i}^{N_{A,l}} \sum_{j}^{N_{B,l'}} c_{i,\nu} c_{j,\mu} \langle \phi_i | \phi_j \rangle \tag{3.4}$$

```
def compute_STO_integral(...) -> tuple[float,list[float],list[float]]:
    number_of_gaussians_A = number_of_gaussians[atom_A][subshell_A]
    number_of_gaussians_B = number_of_gaussians[atom_B][subshell_B]
    slater_exponent_A = slater_exponents[atom_A][subshell_A]
    slater_exponent_B = slater_exponents[atom_B][subshell_B]
    overlap = 0
    dipol = vector(3)
    quadrupol = vector(6)
    for gaussian_i in range(number_of_gaussians_A):
        for gaussian_j in range(number_of_gaussians_B):
            exponent_i = normalised_gaussian_exponent(...)
            exponent_j = normalised_gaussian_exponent(...)
            contraction_i = normalised_contraction_coeficient(...)
            contraction_j = normalised_contraction_coeficient(...)
            s,d,q = compute_GTO_integral(...)
            overlap += contraction_i*contraction_j*s
            for dir in [x,y,z]:
                dipol[dir] += contraction_i*contraction_j*d[dir]
            for dir in [xx,yy,zz,xy,xz,yz]:
                quadrupol[dir] += contraction_i*contraction_j*q[dir]
    return overlap,dipol,quadrupol
```

Figure 3.7.: Python snippet illustrating construction of the STO integral $\langle \psi_\nu | \psi_\mu \rangle$.

For an s-orbital the normalisation terms would be:

$$N_{STO,0} = \sqrt{\frac{\zeta_{A,l}^3}{\pi}} \tag{3.5}$$

$$N_{GTO,0} = \left(\frac{2\alpha_{i,\nu}}{\pi}\right)^{3/4} \tag{3.6}$$

We only need the GTO ones and can compute them in the following way:

```python
def normalised_gaussian_exponent(atom:int, subshell:int, gaussian:int,
↪  slater_exponent:float) -> float:
    normalisation_factor = slater_exponent**2
    return gaussian_exponents[atom][subshell][gaussian]*normalisation_factor

def normalised_contraction_coefficient(atom:int, subshell:int, gaussian:int,
↪  gaussian_exponent:float) -> float:
    l = angular_momentum_of_subshell[atom][subshell]
    normalization_factor = (((2.0*gaussian_exponent)/pi)**(3/4)) *
    ↪  (sqrt(4*gaussian_exponent)**l) / sqrt(double_factorial(l))
    return contraction_coeficients[atom][shell][gaussian]*normalisation_factor

def double_factorial(n:int) -> int:
    if n <= 1:
        return 1
    return n*double_factorial(n-2)
```

Figure 3.8.: Python snippet illustrating normalisation of the gaussian exponents and contraction coefficients.

As the GTOs in the integral are real valued functions we can drop the conjugation when we write them out.

$$\begin{aligned}\langle\phi_i|\phi_j\rangle &= \int \phi^*(\alpha_{i,\nu}, r - R_A)\phi(\alpha_{j,\mu}, r - R_B)\,dr \\ &= \int \phi(\alpha_{i,\nu}, r - R_A)\phi(\alpha_{j,\mu}, r - R_B)\,dr\end{aligned} \tag{3.7}$$

Now we see that we have to compute the products of GTOs. The product of two Gaussians is a new Gaussian centred at a point between the two R_P.

$$\alpha = \alpha_{i,\nu} + \alpha_{j,\mu} \tag{3.8}$$

$$K_{AB} = \left(\frac{2\alpha_{i,\nu}\alpha_{j,\mu}}{\alpha\pi}\right)^{\frac{3}{4}} e^{-\frac{\alpha_{i,\nu}\alpha_{j,\mu}}{\alpha}|R_A - R_B|^2} \tag{3.9}$$

$$R_P = \frac{\alpha_{i,\nu}R_A + \alpha_{j,\mu}R_B}{\alpha} \tag{3.10}$$

$$\phi(\alpha_{i,\mu}, r - R_A)\phi(\alpha_{j,\nu}, r - R_B) = K_{AB}\phi(\alpha, r - R_P) \tag{3.11}$$

This rewrite only works if we also shift the polynomial prefactors to be relative to the product centre R_P, for all $u \in x, y, z$:

$$\begin{aligned}(r_u - R_{Au})^{l_{Au}} &= \sum_{m_i=0}^{l_{Au}} \binom{l_{Au}}{m_i} (R_{Pu} - R_{Au})^{(l_{Au}-m_i)} (r_u - R_{Pu})^{l_{Au}} \\ &= \sum_{m_i=0}^{l_{Au}} v_{m_i} (r_u - R_{Pu})^{l_{Au}}\end{aligned} \tag{3.12}$$

```
def shift_polynomial(l_dim, difference_to_P_dim) -> list[float]:
    poly_coefficients = vector(l_dim+1)
    for m in range(l_dim+1):
        poly_coefficients[m] = comb(l_dim, m)*difference_to_P_dim**(l_dim-m)
    return poly_coefficients
```

Figure 3.9.: Python code illustrating computation of the shifted polynomial coefficients.

If we multiply these shifted prefactors we get the prefactors for the product.

$$\begin{aligned}(r_u - R_{Au})^{l_{Au}} (r_u - R_{Bu})^{l_{Bu}} &= \sum_{m_i=0}^{l_{Au}} \sum_{m_j=0}^{l_{Bu}} v_{m_i} v_{m_j} (r_u - R_{Pu})^{l_{Au}+l_{Bu}} \\ &= \sum_{t}^{l_{Au}+l_{Bu}} v_t (r_u - R_{Pu})^t\end{aligned} \tag{3.13}$$

In the code we compute the values of v_t as a convolution:

```
def convolute(coef_A, coef_B) -> float:
    max_t = len(coef_A)+len(coef_B)
    poly_coefficients = vector(max_t+1)
    for i,ci in enumerate(coef_A):
        for j,cj in enumerate(coef_B):
            poly_coefficients[i+j] += ci*cj
    return poly_coefficients
```

Figure 3.10.: Python code illustrating computation of the R_P polynomial coefficients via a convolution.

To compute the integral over the GTOs we can split it in terms of the 3 dimensions we are integrating over, x,y and z.

$$\begin{aligned}\int \phi(\alpha_{i,\nu}, r - R_A)\phi(\alpha_{j,\mu}, r - R_B)\,dx\,dy\,dz &= K_{AB}\int \phi(\alpha, r - R_P)\,dx\,dy\,dz \\ &= N_{GTO,l_A}N_{GTO,l_B}K_{AB}\int \phi(\alpha, r_x - R_{Px})\,dx \\ &\quad\times \int \phi(\alpha, r_y - R_{Py})\,dy \\ &\quad\times \int \phi(\alpha, r_z - R_{Pz})\,dz\end{aligned} \tag{3.14}$$

Each of these integrals can now be expanded using the analytical solution to gaussian integrals of the form $\int x^t e^{-ax^2}\,dx$[**<empty citation>**]:

$$\begin{aligned}\int \phi(\alpha, r_u - R_{Pu})\,du &= \int \sum_{t=0}^{l_{Au}+l_{Bu}} v_t (r_u - R_{Pu})^t e^{-\alpha|r_u - R_{Pu}|^2}\,du \\ &= \sum_{t=0}^{l_{Au}+l_{Bu}} v_t \int (r_u - R_{Pu})^t e^{-\alpha|r_u - R_{Pu}|^2}\,du \\ &= \sum_{\{t\in 1,..,l_{Au}+l_{Bu}|odd(t)\}} v_t \frac{\left(\frac{t-1}{2}\right)!}{2a^{\frac{t+1}{2}}} + \sum_{\{t\in 0,..,l_{Au}+l_{Bu}|even(t)\}} v_t \frac{(t-1)!!}{2^{\frac{t}{2}+1}a^{\frac{t+1}{2}}}\sqrt{\frac{\pi}{\alpha}}\end{aligned} \tag{3.15}$$

this is something we can compute. Only v_t is dependant on the direction so we can precompute the other terms.

```
def compute_gaussian_integral_factors(alpha:float, l_A_dim:int, l_B_dim:int)->
↪  list[float]:
    factors = vector(l_A_dim+l_B_dim+3)
    for t in range(l_A_dim+l_B_dim+3):
        if t % 2 == 0:
            factors[t] = factorial((t-1)//2)/(2*alpha*((t+1)//2))
        else:
            factors[t] = (double_factorial(t-1)/(2**(t/2+1)*alpha**((t+1)//2))) *
            ↪  sqrt(pi/alpha)
    return factors
```

Figure 3.11.: Python code illustrating computation of the integral factors for even and odd powers

```
def compute_GTO_integral(...)-> tuple[float,list[float],list[float]]:
    alpha = exponent_i + exponent_j
    exponents = exponent_i * exponent_j
    K_AB = ((2*exponents)/(alpha*pi))**(3/4) * e**(-(exponents/alpha)*distance)
    integral_factors = compute_gaussian_integral_factors(alpha, l_A_dim, l_B_dim)
    zeroth_moment = vector(3)
    first_moment = vector(3)
    second_moment = vector(3)
    for dim in [x,y,z]:
        gaussian_product_center = (exponent_i*R_A[dim]+exponent_j*R_B[dim])/alpha
        center_relative_to_A = gaussian_product_center-R_A[dim]
        center_relative_to_B = gaussian_product_center-R_B[dim]
        l_A_dim = angular_momentum_in_dimension[l_A][orbital_A][dim]
        l_B_dim = angular_momentum_in_dimension[l_B][orbital_B][dim]
        l_max_dim = max(l_A_dim, l_B_dim)
        vmis = shift_polynomial(l_A_dim, center_relative_to_A)
        vmjs = shift_polynomial(l_B_dim, center_relative_to_B)
        vts = convolute(vmis, vmjs)
        for t, vt in enumerate(vts):
            for moment in [0,1,2]:
                for m in range(moment+1):
                    zeroth_moment[dim] += comb(moment,m) *
                    ↪ gaussian_product_center**(moment-m) * vt *
                    ↪ integral_factors[t+m]

    overlap = zeroth_moment[x]*zeroth_moment[y]*zeroth_moment[z]

    dipol = vector(3)
    dipol[x] = first_moment[x]*zeroth_moment[y]*zeroth_moment[z]
    dipol[y] = zeroth_moment[x]*first_moment[y]*zeroth_moment[z]
    dipol[z] = zeroth_moment[x]*zeroth_moment[y]*first_moment[z]

    quadrupol = vector(3)
    quadrupol[xx] = second_moment[x]*zeroth_moment[y]*zeroth_moment[z]
    quadrupol[yy] = zeroth_moment[x]*second_moment[y]*zeroth_moment[z]
    quadrupol[zz] = zeroth_moment[x]*zeroth_moment[y]*second_moment[z]
    quadrupol[xy] = first_moment[x]*first_moment[y]*zeroth_moment[z]
    quadrupol[xz] = first_moment[x]*zeroth_moment[y]*first_moment[z]
    quadrupol[yz] = zeroth_moment[x]*first_moment[y]*first_moment[z]
    return overlap, dipol, quadrupol
```

Figure 3.12.: Python code illustrating computation of the integral $\langle\phi_i|\phi_j\rangle$ using the dimension-wise decomposition.

NOTE(Asmus) Explain D and Q as well

3.1.4 Building the Hückel matrix

As a part of computing the Hückel energy we need to construct the Hückel matrix. The Hückel matrix comes from extended Hückel theory and is calculated as follows.

$$\begin{aligned} H_{\mu\nu}^{EHT} &= \frac{1}{2} K_{AB}^{ll'} S_{\mu\nu} (H_{\mu\mu} + H_{\nu\nu}) \\ &\cdot X(EN_A, EN_B) \\ &\cdot \Pi(R_{AB}, l, l') \\ &\cdot Y(\zeta_l^A, \zeta_{l'}^B), \forall \mu \in l \in A, \nu \in l' \in B \end{aligned} \tag{3.16}$$

$$H_{\nu\nu} = H_A^l - H_{CN_A} CN'_A \qquad \forall \nu \in l \in A \tag{3.17}$$

$$CN'_A = \sum_{B \neq A} \left(1 + e^{-10\left(\frac{4\left(R_{A,cov}+R_{B,cov}\right)}{3R_{AB}} - 1\right)} \right)^{-1} \left(1 + e^{-20\left(\frac{4\left(R_{A,cov}+R_{B,cov}+2\right)}{3R_{AB}} - 1\right)} \right)^{-1} \tag{3.18}$$

where μ and ν are AO indecies, l and l' index shells. Both AO's are associated with an atom labled A and B. $K_{AB}^{ll'}$ is a element and shell specific fitted constant however, in GFN2 it only depends on the shells. $S_{\mu\nu} = \langle \phi_\mu | \phi_\nu \rangle$ is just the overlap of the orbitals. In GFN2 $H_{\kappa\kappa} = h_A^l - \delta h_{CN'_A}^l CN'_A$ where CN'_A is the modified GFN2-type Coordinate Number for the element of atom A.

$$\begin{aligned} CN_A^{'} = \sum_{B \neq A}^{N_{\text{atoms}}} &(1 + e^{-10(4(R_{A,\text{cov}}+R_{B,\text{cov}})/3R_{AB}-1)})^{-1} \\ &\times (1 + e^{-20(4(R_{A,\text{cov}}+R_{B,\text{cov}}+2)/3R_{AB}-1)})^{-1} \end{aligned} \tag{3.19}$$

h_A^l and $\delta h_{CN'_A}^l$ are both fitted constants. EN_A is the electronegativity of the element of atom A, given in the original `xtb` code.

$$X(EN_A, EN_B) = 1 + k_{EN} \Delta EN_{AB}^2 \tag{3.20}$$

$$k_{EN} = 0.02 \text{ in GFN2} \tag{3.21}$$

$$\Delta EN_{AB}^2 = (EN_A - EN_B)^2 \tag{3.22}$$

$$\Pi(R_{AB}, l, l') = \left(1 + k_{A,l}^{\text{poly}} \left(\frac{R_{AB}}{R_{\text{cov},AB}}\right)^{\frac{1}{2}}\right)\left(1 + k_{B,l'}^{\text{poly}} \left(\frac{R_{AB}}{R_{\text{cov},AB}}\right)^{\frac{1}{2}}\right) \tag{3.23}$$

$R_{\text{cov},AB}$ are the summed covalent radii ($R_{\text{cov},A} + R_{\text{cov},B}$), e.g. $R_{\text{cov},H} = 0.32$, $R_{\text{cov},C} = 0.75$ are given in the original `xtb` code. $k_{A,l}^{\text{poly}}$ and $k_{B,l'}^{\text{poly}}$ are element and shell specific constants.

$$Y(\zeta_l^A, \zeta_{l'}^B) = \left(\frac{2\sqrt{\zeta_l^A \zeta_{l'}^B}}{\zeta_l^A + \zeta_{l'}^B}\right)^{\frac{1}{2}} \tag{3.24}$$

Here, ζ_l^A are the STO exponents of the GFN2-xTB AO basis.
Slater Type Orbitals are defined as such:

$$\chi_{\zeta,n,l,m}(r, \theta, \varphi) = N Y_{l,m}(\theta, \varphi) r^{n-1} e^{-\zeta r} \tag{3.25}$$

N is a normalisation constant, Y are spherical harmonic funtions, n, l, m are the quantum numbers for the AO. r, θ, φ are polar 3D coordinates. ζ determines the radial extent of the STO, a large value gives rise to a function that is "tight" around the nucleus and a small value gives a more "diffuse" function. This ζ is the one mentioned in the Y term of E_{EHT} and is a value fitted when constructing the basis set, thus it is given to us.

```
def huckel_matrix(atoms: list[int], positions: list[list[float]], overlap:
↪  list[list[float]]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    H_EHT = get_square_matrix(len(orbitals))
    CN = get_coordination_numbers(atoms,positions)
    for orbital_idx, (atom_idx,atom,subshell,orbital) in enumerate(orbitals):
        CN_A = CN[atom_idx]
        H_A = self_energy[atom][subshell] # constant
        H_CN_A = GFN2_H_CN_A[atom][subshell] # constant
        H_EHT[orbital_idx][orbital_idx] = H_A - H_CN_A*CN_A

    for idx_A,(atom_A_idx,atom_A,subshell_A,_) in enumerate(orbitals):
        l_A = angular_momentum_of_subshell[atom_A][subshell_A]
        EN_A = electro_negativity[atom_A]
        R_A = positions[atom_A_idx]
        Rcov_A = covalent_radii[atom_A]
        k_poly_A = k_poly[atom_A][l_A]
        for idx_B,(atom_B_idx,atom_B,subshell_B,_) in enumerate(orbitals):
            if idx_A == idx_B:
                continue
            l_B = angular_momentum_of_subshell[atom_B][subshell_B]
            EN_B = electro_negativity[atom_B]
            R_B = positions[atom_B_idx]
            Rcov_B = covalent_radii[atom_B]
            k_poly_B = k_poly[atom_B][l_B]
            K_ll = GFN2_K_AB[l_A][l_B]
            delta_EN_squared = (EN_A-EN_B)**2
            k_EN = 0.02
            X = 1+k_EN*delta_EN_squared
            R_AB = euclidean_distance(R_A,R_B)
            Rcov_AB = Rcov_A + Rcov_B
            PI = (1+k_poly_A*sqrt(R_AB/Rcov_AB)) *
            ↪  (1+k_poly_B*sqrt(R_AB/Rcov_AB))
            slater_exp_A = slater_exponent[atom_A][l_A]
            slater_exp_B = slater_exponent[atom_B][l_B]
            Y = sqrt((2*sqrt(slater_exp_A*slater_exp_B)) /
            ↪  (slater_exp_A+slater_exp_B))
            H_nn = H_EHT[idx_A][idx_A]
            H_mm = H_EHT[idx_B][idx_B]
            S_nm = overlap[idx_B][idx_B]
            H_EHT[idx_A][idx_B] = k_ll*(1/2)*(H_nn+H_mm)*S_nm*Y*X*PI
    return H_EHT
```

Figure 3.13.: Python like code illustrating construction of H_{KK}

3.2 High Performance Parallel Computing

Now that we have taken a look at the computations performed by the xTB method, we can begin to see that the steps are the same for all molecules, and that the exact amount of steps are identical between isomers. Isomers are molecules that are identical in their composition of atoms, that is, they have the same molecular formula, and differ only in the relative positions between atoms or the bonds between them. This project focuses on large isomer spaces of fullerenes in the range $C_{20}, \cdots, C_{200}$ which is a total of 2157751423 molecules, or in other words in the order of 10^9. A fullerene is a molecule consisting only of carbon atoms where each atom has three neighbors and is arranged such that the molecule produces a mesh of exactly 12 pentagonal faces and the rest being hexagonal faces.

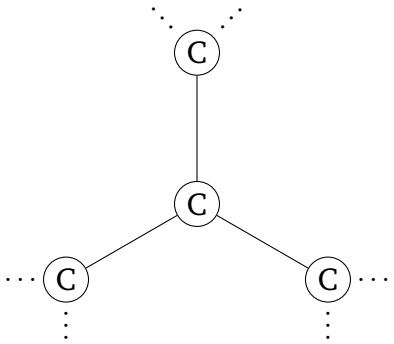

Figure 3.14.: Each carbon atom in a fullerene has exactly 3 neighboring carbon atoms.

For each isomerspace we essentially have two matrices, one for the coordinates of each carbon atom, and one containing the three neighbors for each of the atoms. The first matrix as seen in Figure 3.15, is of size $N \times 3 \times M$ where N is the amount of atoms for each of the fullerenes within the same isomerspace, 3 is the three coordinates for each atom, and M is the amount of fullerenes within the isomerspace. The second matrix as seen in Figure 3.16, is the exact same, but instead of coordinates it holds the indices for the three neighboring carbon atoms. The xTB method only needs the positions for each atom, so we are not concerned with the second matrix.

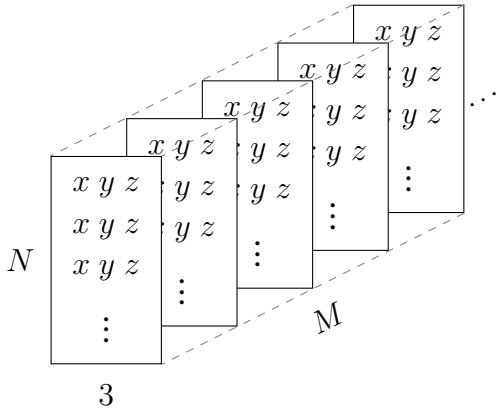

Figure 3.15.: Matrix with the atom coordinates for each fullerene within an isomerspace. The 3-dimensional matrix holds the 3 coordinates for each of the N carbon atoms for all M fullerenes in a given isomerspace.

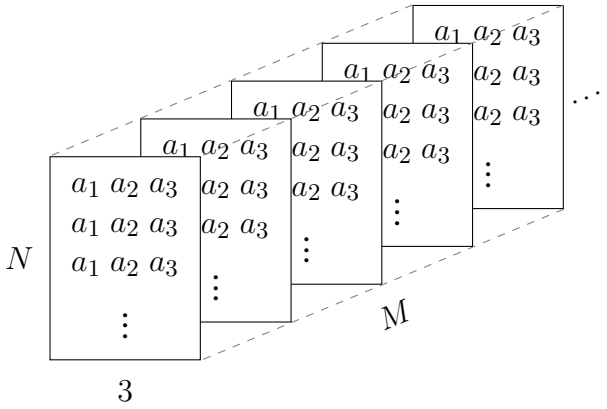

Figure 3.16.: Matrix holding an adjacency list of 3 neighbors for each carbon atom for all fullerenes in a given isomerspace.

The xtb program takes a single molecule as input, but the fact that fullerenes in the same isomerspace executes the same series of instructions, makes a great case for lockstep parallelism. Running computations in lockstep means doing the exact same instructions in parallel, and if we can do this, then we can efficiently utilize the single instruction, multiple threads (SIMT) execution model. The SIMT model is used when a warp scheduler dispatches a single instruction to all threads within a warp. All of the threads in the warp will then execute the instruction simultaniously, which means that they are synchronized at a hardware level. This perfect as it means that we can use warps to execute instructions for multiple fullerenes in lockstep. With this in mind, instead of feeding the program a single fullerene at a time, we want to give it a complete isomerspace so that we can utilize all threads on one or even multiple GPUs. The largest isomerspace we consider is C_{200}, which is 214127742 fullerenes, or in other words in the order of 10^8.

It takes around 30 seconds for a 12th generation Intel mobile processor to run xtb on a single C_{200} fullerene. Going through the whole isomerspace on this CPU would take about 204 years to complete. This is a bit long to wait for a failed run, and by that time we will probably have achieved more efficient ways of computation, or the world might simply have more pressing matters than keeping this computer alive. The degree of lockstep parallelism scales with the isomerspace, and since there are no interdependencies between the molecules, this makes it a perfect case for horizontal scaling by using hundreds or even thousands or GPUs. The CPU used has 12 cores and 24 threads, 12 of which can run in parallel. An NVIDIA GPU has hundredthousands of threads and tenthousands of CUDA cores, though these cores are not equivalent to CPU cores, and we can in general not compare the relative performance between the two by looking at this. Something that is typically compared though, is the theoretical throughput of floating point operations per second

(FLOPS). The top of the line AMD EPYC 9965 CPU has a theoritical throughput of 27.648 TeraFLOPS (TFLOPS) for 32-bit floating point values[1], while the NVIDIA A100 has 156 TFLOPS[2]. From these observations we can say that the GPU is much more capable of massive parallelism, which is what we seek for these massive isomerspaces.

3.2.1 Case Study

To get an idea of the scale of parallelism we can achieve on a single GPU, we will take a look at the memory specifications of the NVIDIA A100 GPU. We hope to use this to get an idea of how many fullerenes can fit into memory.

Using all the levels of memory on a GPU efficiently is crucial to achieve high performance in parallel computing tasks. Let us therefore take a moment to familiarize ourselves with the different types typically found in the memory hiearchy on a GPU. Below you will find an overview of the main memory levels with short explanations for each of them:

- Global - Accessible from all threads on the GPU. This is the largest but also the slowest pool of memory in terms of latency.
- Shared - Tied to a thread block (or workgroup), so it can be accessed by the threads in the same block. This pool of memory is smaller but faster to access than global memory.
- Register Memory - Each thread on a GPU has private access to a number of registers. This is the fastest type of memory used to store local variables and intermediate results.

We will specifically look at the NVIDIA A100 GPU [2], which has 40 GB of memory and 108 SMs each with 64 FP32 CUDA cores and 4 tensor cores resulting in a total of 6912 CUDA cores and 432 tensor cores.

To find out how many fullerenes worth of coordinate data can fit into global memory, we can use the following equation:

$$GlobalMemFullereneCapacity = \left\lfloor \frac{GlobalMemInBits}{FullereneAtomCount \times 3 \times 32} \right\rfloor \tag{3.26}$$

Figure 3.17.: The amount of 3-dimensional fullerene coordinate data that fits in global memory. The 3×32 comes from the three 32-bit floating point numbers that represents the 3-dimensional coordinates. This equation does not consider other data such as results or intermediate values.

$$MinimumBatchesToFitIsomerspace = \left\lceil \frac{IsomerspaceSize}{GlobalMemFullereneCapacity} \right\rceil \tag{3.27}$$

Figure 3.18.: The minimum number of batches required to process a complete isomerspace in global memory.

Be aware that this equation does not consider any space needed for the results or any intermediate values, but we can use this equation to calculate the minimum amount of batches needed to process the whole isomerspace. If we want to process the whole isomerspace at once, then we might need to add additional GPUs. For example, the C_{200} isomerspace needs a minimum of 13 GPUs with 40 GB of global memory each.

We are also interested in knowing how much non-global memory is available to each xTB computation. Knowing this will tell us how many of the threads, warps, and blocks we can utilize on the GPU. To figure this out, we first need to know how much of the various memory levels are available to a single thread if distributed over all threads.

Each SM has up to 164 KB of shared memory and is divided into four partitions, each containing a 64 KB register file, an one warp scheduler, 16 CUDA Cores dedicated for processing FP32 operations, 16 CUDA Cores for processing INT32 operations, 8 CUDA Cores for processing FP64 operations, and a tensor core, which is specialized for matrix operations. See Figure 3.19 for a figure of the SM architecture.

Figure 3.19.: The NVIDIA A100 Architecture for a streaming-multiprocessor. [2, Figure 5]

There are a maximum of 64 warps per SM, so each warp scheduler has 16 warps allocated to them. Each warp has 32 threads, so if we distribute a register file over the 512 threads for an SM, then each of them have 320 bytes (see Equation 3.28). If we need to allocate more memory per thread, then we can utilize fewer of the threads.

To get the amount of shared memory available to each thread when distributed evenly, we can use Equation 3.29. There is 164 KB of shared memory per SM, so each thread has 80 bytes to work with.

$$RegisterMemoryPerThread = \frac{RegisterFile}{WarpsPerPartition \times ThreadsPerWarp} \tag{3.28}$$

$$SharedMemoryPerThread = \frac{SharedMemoryPerSM}{WarpsPerSM \times ThreadsPerWarp} \tag{3.29}$$

Depending on the upperbound on intermediate data required by the GFN2-xTB algorithm, we can tweak the amount of non-global memory available to each fullerene by using fewer threads, warp, and blocks. For example, using only 16 of the 32 threads in each warp will double the available register and shared memory for each thread. There is a point of diminishing returns as a single thread is limited to a maximum of 255 32-bit registers. In the case that the non-global memory alone is not enough to hold the intermediate data of a molecule, then we will need to also utilize global memory. Shared memory has a latency of 20 to 30 cycles while global memory has a latency of 200 to 1000 cycles [3]. Global memory is slow to access, so when used it becomes especially important to place data strategically to allow for coalesced access.

3.2.2 Data Coalescing

Our goal is an xTB implementation capable of working with data for several molecules in parallel. This opens the opportunity for coalesced data access if we can structure the data in a way that achieves spacial locality. Spacial locality is achieved when related data is close together in memory such that it can be accessed as a contiguous block. This minimizes cache misses by eliminating random access across cache lines. One way to achieve spacial locality is to store data as structures of arrays (SOA) instead of arrays of structures (AOS) (see Figure 3.20). In order to allow coalesced access to data in device memory, we can essentially realign the data to match the access patterns of our kernels.

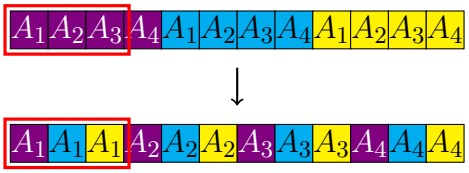

Figure 3.20.: Transforms arrays of structures where the memory for each structure is laid out contiguously, into a flattened structure of arrays where data relevant for a computation is close together. This way of grouping related data close together achieves spatial locality and allows for coalesced access of the data.

In contrast to a task-parallel model where different instructions are executed in parallel, the threads in a warp is optimized around the single instruction multiple threads model (SIMT). This means that all 32 threads in a warp can execute the exact same instructions in lockstep, thus being synchronized on a hardware level. To run in lockstep, data access has to complete for all the threads before the warp can advance, so to best utilize the SIMT model we need to fetch the data efficiently in a coalesced manner to minimize latency and maximize throughput.

This project is part of a larger pipeline, so the fullerene data that is fed to the GFN2-xTB algorithm is already in device memory when we get it. The xTB implementation therefore never moves any

data between the host and device. As such, data coalescing is something we consider when data is moved from higher to lower levels of memory on a device.

3.2.3 Introduction to a Parallel Lockstep Algorithm

We want to ensure that all threads within a warp can work on different fullerenes in parallel. To achieve this we need to remove branching in the code and place upper bounds on all loops to make sure the exact same work is performed by the threads. Luckily, since we are only working with one type of atom, we can simply replace branches with the case that applies for carbon.

Normally we would have to set upper bounds on all loops to ensure instructions are executed the same amount of times for all atoms. This is not necessary in our case since we only have carbon atoms, because all of them have the same amount of shells, orbitals, electrons and so on.

3.2.4 Orbitals as a constant

This following helper function would never be called in the actual implementation, but instead be computed on the fly. This section will show however, that we never need to compute this at all with fullerenes and can instead store it as a compact constant array.

```
def get_orbitals(atoms: list[int]) -> list[tuple[int]]:
    orbitals = []
    for atom_idx,atom in enumerate(atoms):
        for subshell in range(number_of_subshells[atom]):
            l = angular_momentum_of_subshell[atom][subshell]
            for orbital in range(l*2+1):
                orbitals.append((atom_idx,atom,subshell,orbital))
    return orbitals
```

Figure 3.21.: The original helper function for computing the orbitals for each atom in a molecule.

The orbitals for each atom will always be the same since we are only working with carbon atoms, so we can simply compute the orbitals for the first atom and then use the count of atoms as a bound for how many times we can read the array. This means that we only return an array with the orbitals for the two subshells in the first carbon atom, and we can thus remove the outermost loop.

```python
def get_orbitals() -> list[int]:
    orbitals = []

    # We only work with carbon atoms
    carbon = 6-1 # -1 because we index from 0
    # There are always 2 subshells in a carbon atom
    subshell_count = number_of_subshells[carbon]

    for subshell in range(subshell_count):
        # There are always two subshells and l will always be 0 for the first
        ↪ and 1 for the second
        l = angular_momentum_of_subshell[carbon][subshell]
        for orbital in range(l*2+1):
            # We do not need atom_idx or atom because all are carbon
            orbitals.append((subshell, orbital))

    return orbitals
```

Figure 3.22.: Helper function for computing the amount of orbitals for a single carbon atom. This function only considers carbon atoms, so only the first atom is computed and we can therefore remove the outer loop over atoms.

We can simplify this even further by unrolling the loops, since they have a constant amount of iterations. Also, since we only access carbon, we can remove values for other atoms from `number_of_subshells` and `angular_momentum_of_subshell`. These changes gives us the following:

```
def get_orbitals() -> list[int]:
    orbitals = []

    angular_momentum_of_subshell = [0, 1]
    l1 = angular_momentum_of_subshell[0] # Always 0
    l2 = angular_momentum_of_subshell[1] # Always 1

    orbitals_in_subshell1 = l1*2+1 # Always 1
    orbitals_in_subshell2 = l2*2+1 # Always 3

    # If we unroll the loops then we get:

    # Iteration 1
    # [subshell_0, orbital_0]
    orbitals.append_all([0, 0])

    # Iteration 2
    # [subshell_0, orbital_0, orbital_1, orbital_2]
    orbitals.append_all([1, 0, 1, 2])

    return orbitals # Always [0,0,1,0,1,2]
```

Figure 3.23.: Helper function for computing the amount of orbitals a single carbon atom. A carbon atom always has 4 orbitals in total, so this function unrolls the loop over orbtials and show that we always get the same constant array as result.

As noted in the code, this always results in the array `[0,0,1,0,1,2]`, so we can just store this as a constant instead of doing the computation. This array is only for a single carbon atom, but instead of replicating this by the amount of atoms, we can save the space by indexing into this array for all atoms and bound the amount of reads to the count of atoms in the fullerene.

3.2.5 Simplifying the initial density guess

The initial density matrix guess can also be simplified when we only consider carbon atoms.

```python
def density_initial_guess(atoms: list[int]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    occs = get_square_matrix(len(orbitals))
    for idx,(_,atom,subshell,_) in enumerate(orbitals):
        l = angular_momentum_of_subshell[atom][subshell]
        orbitals_in_subshell = l*2+1
        electrons_in_subshell = reference_occupations[atom][subshell]
        electrons_per_orbital = electrons_in_subshell/orbitals_in_subshell
        occs[idx][idx] = electrons_per_orbital
    return occs
```

Figure 3.24.: The original simplified code for the initial density matrix guess.

We just simplified the orbitals to a constant, but we are only using the length of the array in this function, which is no longer representative of the total amount of orbitals. Instead we can `atom_count*4` as the range, since each atom has four orbitals. Below we show the iteration for the first atom to show that there is always 1 electron per orbital when distributing them evenly, which means we can just create a matrix with 1's along the main diagonal.

```python
def density_initial_guess(atom_count: int) -> list[list[float]]:
    row_length = atom_count*4 # There are 4 orbitals for each atom

    orbital_occupations = [0] * row_length**2

    angular_momentum_of_subshell = [0, 1]
    l1 = angular_momentum_of_subshell[0] # Always 0
    l2 = angular_momentum_of_subshell[1] # Always 1

    orbitals_in_subshell1 = l1*2+1 # Always 1
    orbitals_in_subshell2 = l2*2+1 # Always 3

    reference_occupations = [1.0, 3.0] # Occupations for carbon
    electrons_in_subshell1 = reference_occupations[0] # Always 1.0
    electrons_in_subshell2 = reference_occupations[1] # Always 3.0

    electrons_per_orbital_in_subshell1 =
    ↪  electrons_in_subshell1/orbitals_in_subshell1 # 1/1 = 1
    electrons_per_orbital_in_subshell2 =
    ↪  electrons_in_subshell2/orbitals_in_subshell2 # 3/3 = 1

    # Subshell 1
    orbital_occupations[0] = electrons_per_orbital_in_subshell1 # Always 1
    # Subshell 2
    for i in range(1, orbitals_in_subshell2 + 1):
        orbital_occupations[i * row_length + i] =
        ↪  electrons_per_orbital_in_subshell2 # Always 1

    return orbital_occupations

    # This should be repeated atom_count amount of times so we get the full
    ↪  (atom_count*4)**2 density matrix.
    # Orbital_occupations are always 1 we can just return the (atom_count*4)**2
    ↪  with 1s along the main diagonal.

    # The above computations can therefore be ignored and we can instead just
    ↪  do the following instead:

    density_matrix = [0] * row_length**2
    for i in range(row_length):
        density_matrix[i * row_length + i] = 1

    return density_matrix
```

Figure 3.25.: Function for computing the initial density matrix guess. This code only shows an unrolled version of the first iteration to show that we can simply make an array with 1's along the main diagonal, since there is always 1 electron per orbital when distributing them evenly.

3.2.6 Computing overlap in lockstep

```
def overlap(atoms: list[int]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    S = get_square_matrix(len(orbitals))
    for idx_A, (_,atom_A,subshell_A,orbital_A) in enumerate(orbitals):
        for idx_B, (_,atom_B,subshell_B,orbital_B) in enumerate(orbitals):
            if idx_A == idx_B:
                S[idx_A][idx_B] = 1
            else:
                S[idx_A][idx_B] = compute_integral(...)
    return S
```

Figure 3.26.: The original simplified overlap function.

```
def overlap(atom_count: int) -> list[list[float]]:
    row_length = atom_count*4 # There are 4 orbitals for each atom
    S = [0] * row_length**2

    for i in range(len(S)):
        column = i \% row_length
        row = i // row_length
        is_diag = int((column ^ row) == 0)
        is_diag_negated = 1 - is_diag
        # We mask the input to compute_integral so it does not real work for
        ↪ the main diagonal
        S[row * row_length + column] = is_diag + compute_integral(is_diag_negated
        ↪ * input_data...)
        # Otherwise we can also just run it always and not count it like this
        S[row * row_length + column] = is_diag + is_diag_negated *
        ↪ compute_integral(input_data...)
        # We should also be able to keep the if statement in this case because
        ↪ row == column for all threads at the same time, thus causing no
        ↪ warp divergence

    return S
```

Figure 3.27.: This version of the overlap function combines the loops and shows three options for getting rid of the conditional to avoid warp divergence.

3.3 Introduction to quantum algorithmic approaches

In this section we will construct quantum algorithms for calculating two of the GFN2-xTB[4] energy terms: E^{Γ} and E^{γ}. We will showcase two very different approaches to doing a calculation as a building block of a larger circuit.

The conceptually simplest approach is to directly translate classical mathematical circuits to the quantum world using ancillary qubits to ensure reversibility. Here most of the computation happens in the state, and the result is readable in the bits of the measurement output. This approach has seen some development beyond this simple translation resulting in some very qubit efficient primitives for multiplication and addition in particular[5, 6]. This approach will be applied to both the E^{Γ} and E^{γ} terms, and be referred to as Quantum Digital Arithmetic in this report.

Our second approach is Quantum Amplitude Arithmetic[7]. In this approach we try to prepare the desired result not as a easily read measurement result, but as part of the amplitude of a state. We will use this approach for the E^{Γ} term to prepare a qubit in the state $w|0\rangle + \alpha|0\rangle$ where we can choose α to be proportional to the E^{Γ} of the molecule. Alternatively we can choose α to be proportional to $E^{\Gamma} - E^{\Gamma}_{H}$ where E^{Γ}_{H} is the E^{Γ} energy for some known high energy isomer. This is not something we imagine being a common thing to want, however it is something which we want for the total energy. The issue that is solved by subtracting a known high energy is the following: We want large relative differences between stable isomers and unstable ones. This is to ensure that if we run a superposition of isomers though the circuits required for the complete GFN2-xTB method we are able to efficiently sample the low energy isomers in the isomer space. Say we know all the energies; if all the energies lie between -100 and -101 (units not important), which may make a large difference, the relative difference is not large. If we subtract the energy of a known high energy isomer of say -100.1 we get much larger relative differences where the low energy isomers will have a much larger amplitude on 1, α, than the high energy isomers.

For both of these approaches we will assume that we have access to some intricately prepared states. We will not go into how the are prepared other than the fact that classically we can generate the approximate geometries for entire isomer spaces without any other information. As any classical computation in theory also can be applied to a quantum computer after modifications to make it reversible it is a possibility to prepare these states. A sketch of the preparation could be to generate all the IDs of the isomers, create a superposition and then run the algorithms for determining the geometry on this superposition of IDs.

As a final note leading in to the implementations Quantum Amplitude Amplification is not to be confused with Quantum Amplitude Amplification, both shortened to QAA some times. We will be very explicit in which we are discussing in this thesis.

3.3.1 Calculating E^{Γ} using Quantum Digital Arithmetic

The GFN2-xTB E^{Γ} term has the following form[8]

$$E^{\Gamma} = \frac{1}{3}\sum_{A}\sum_{\mu\in A}(q_{A,\mu})^3\Gamma_{A,\mu}, \tag{3.30}$$

where $q_{A,\nu} = \sum_B \sum_{\nu\in B} P_{\mu\nu}S_{\mu\nu}$ is the Mulliken partial charge of shell μ associated with atom A. P, S are the density and overlap matrices. $\Gamma_{A,\mu} = \Gamma_A K_\mu$ is just the product of an element specific constant and a shell specific constant, for our purposes the element is always carbon and the shell is either the first or second in GFN2 thus we have 2 numbers $\Gamma_{\text{Carbon},0(1)}$ henceforth referred to as $\mathit{\Gamma}_{0(1)}$.

Let us first rewrite the inner expression a bit given our new definition and knowledge of the atoms we are working with.

$$\sum_{\mu\in A}(q_{A,\mu})^3\Gamma_{A,\mu} = \sum_{\mu\in\{0,1\}}(q_{A,\mu})^3\mathit{\Gamma}_{\mu} \tag{3.31}$$

To implement this equation as a circuit we want to be able to add the terms in the inner expression to an accumulator. Thus we need a unitary which computes this function on a given state $|\mathit{\Gamma}_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |acc\rangle_E \to |\mathit{\Gamma}_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |acc + (q_{A,\mu})^3\mathit{\Gamma}_\mu\rangle_E$. The subscripts on the kets refer to the quantum register they represent. Consider having access to the following fused multiply add unitary $|A\rangle|B\rangle|acc\rangle \to |A\rangle|B\rangle|acc + A*B\rangle$, call it $\text{MADD}_{A,B,C}$. Let us to our initial Γ, Q, E registers add two ancillary registers, W_1, W_2. We can now apply the following unitary

$$E^{\Gamma}_{i\ (\Gamma,Q,W_1,W_2,E)} = \text{MADD}^{\dagger}_{\Gamma,Q,W_1}\text{MADD}^{\dagger}_{Q,W_1,W_2}\text{MADD}_{Q,W_2,E}\text{MADD}_{Q,W_1,W_2}\text{MADD}_{\Gamma,Q,W_1} \tag{3.32}$$

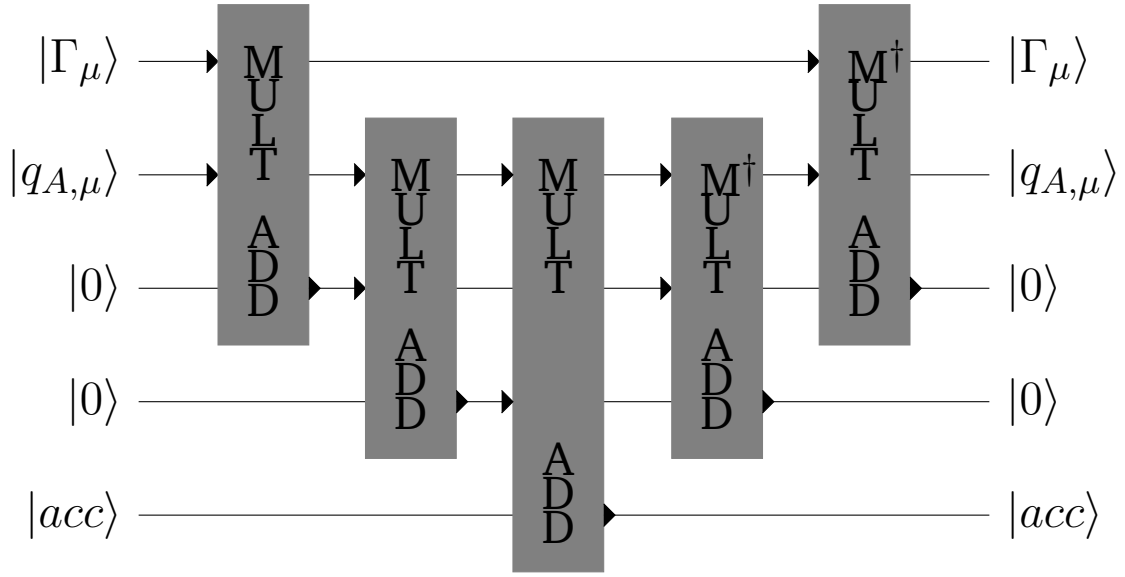

Figure 3.28.: Visualization of the circuit presented in equation 3.32. The arrows going into a block symbolise that qubit being used for the calculation happening in a given block. The arrows leaving a block symbolise the block changing something about that qubit. † is the complex conjugate used for doing an operation in reverse.

Let us follow the process:

$$E_i^{\Gamma}{}_{(\Gamma,Q,W_1,W_2,E)} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |0\rangle_{W_1} |0\rangle_{W_2} |acc\rangle_E \tag{3.33}$$

$$= \text{MADD}^\dagger_{\Gamma,Q,W_1} \text{MADD}^\dagger_{Q,W_1,W_2} \text{MADD}_{Q,W_2,E} \text{MADD}_{Q,W_1,W_2} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |0\rangle_{W_2} |acc\rangle_E \tag{3.34}$$

$$= \text{MADD}^\dagger_{\Gamma,Q,W_1} \text{MADD}^\dagger_{Q,W_1,W_2} \text{MADD}_{Q,W_2,E} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |\Gamma_\mu (q_{A,\mu})^2\rangle_{W_2} |acc\rangle_E \tag{3.35}$$

$$= \text{MADD}^\dagger_{\Gamma,Q,W_1} \text{MADD}^\dagger_{Q,W_1,W_2} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |\Gamma_\mu (q_{A,\mu})^2\rangle_{W_2} |acc + \Gamma_\mu (q_{A,\mu})^3\rangle_E \tag{3.36}$$

$$= \text{MADD}^\dagger_{\Gamma,Q,W_1} |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |\Gamma_\mu q_{A,\mu}\rangle_{W_1} |0\rangle_{W_2} |acc + \Gamma_\mu (q_{A,\mu})^3\rangle_E \tag{3.37}$$

$$= |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |0\rangle_{W_1} |0\rangle_{W_2} |acc + \Gamma_\mu (q_{A,\mu})^3\rangle_E \tag{3.38}$$

$$\tag{3.39}$$

We see that already in eq. 3.36 we have the result we want in the accumulation register. We continue with the uncomputation of the W_1, W_2 registers purely to be able to reuse them in the remaining calculations. This saves the qubits required for having 2 ancillary registers for every calculation. The 'fused multiply add' gates here could be implementing using QFT multipliers[6] in

which case we wouldn't need any ancillaries beyond the two mentioned. If we decompose our QFT multiplier into its components it is essentially a chain of QFT additions[5, 6] and multiplications by a constant power of two. These additions are built up of two components: (inverse) Fourier transforms and conditional rotations. When we chain them together like this however many of the transforms can be taken out as they are always followed or preceded by their inverse except for in the beginning and end.

Let us say we are given a circuit, SDA, for encoding a molecule from its ID in the following manner, and a circuit DA $= \prod_A \prod_{\mu\in\{0,1\}} E_i^{\Gamma}{}_{(\Gamma_\mu, Q_{A,\mu}, W_1, W_2, E)}$. Then

$$\begin{aligned}
&\text{DA SDA}\,|ID\rangle_{ID}\,|0\rangle \rightarrow \\
&\quad \text{DA}\,|ID\rangle_{ID}\left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_{A} |q_{A,\mu}\rangle_{Q_{A,\mu}}\right)|0\rangle_{W_1}\,|0\rangle_{W_2}\,|0\rangle_{E} \rightarrow \\
&\quad\quad |ID\rangle_{ID}\left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_{A} |q_{A,\mu}\rangle_{Q_{A,\mu}}\right)|0\rangle_{W_1}\,|0\rangle_{W_2}\,|E^{\Gamma}\rangle_{E}
\end{aligned} \tag{3.40}$$

will give us the E^{Γ} energy term in the E register corresponding to the ID in the ID register. Let us look at what happens as we are given a superposition of molecules as input instead. We will use a method introduced in the same paper[7] as Quantum Amplitude Arithmetic for sampling the low energy candidates. This will lead us gently into the later section where we use Quantum Amplitude Arithmetic to construct a circuit for calculating E^{Γ}.

3.3.2 Sampling using Quantum Amplitude Arithmetic

Assume that we are given an equal superposition of all the canonical IDs of the fullerenes in an isomer-space. We can apply SDA to set up the encoding and then apply DA. We now have computed the E^{Γ} energies for every isomer. However we can only sample once. Let us say that we are interested in the isomers with the lowest energies. We then would like the probability of sampling an isomer to be proportional to $|E^{\Gamma}|$.

Wang et al. use their introduced addition and multiplication primitives to construct a probabilistic circuit which transforms the state $|x\rangle_D\,|0\rangle_C\,|0\rangle_W \rightarrow \left[\frac{1}{2}\frac{1}{2^n}\right]x\,|x\rangle_D\,|0\rangle_C\,|1\rangle_W + \alpha\,|\omega\rangle_{D\otimes C\otimes W}$ where α is some normalization factor, and $|\omega\rangle$ is some state with no overlap with the state containing all 0's in the control register, C, and 1 in the work register, W. This property makes it very easy to check if the circuit succeeded, we can just use the measurement output that collapses the state to see if we

measured the correct pattern of all zeroes and a one. The term in the brackets is the probability of success, α is then related to the probability of failure as the result should be normalised.

When using this circuit we can treat the E register containing our resulting E^Γ term as the data register, D. We can reuse the W_1, W_2 registers as the control and work registers. Let us take a look at that.

$$\begin{aligned}&\sum_{ID\in\text{isomers}} |ID\rangle_{ID} \left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_A |q_{A,\mu}\rangle_{Q_{A,\mu}} \right) |0\rangle_{W_1} |0\rangle_{W_2} |E^\Gamma\rangle_E \rightarrow \\ &\sum_{ID\in\text{isomers}} |ID\rangle_{ID} \left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_A |q_{A,\mu}\rangle_{Q_{A,\mu}} \right) \left(\frac{1}{2} \frac{E^\Gamma_{ID}}{2^n} |0\rangle_{W_1} |1\rangle_{W_2} |E^\Gamma_{ID}\rangle_E + \alpha_{ID} |\omega_{ID}\rangle \right)\end{aligned} \tag{3.41}$$

If we now sample from this superposition and postselect for $W_1 = 0$ and $W_2 = 1$ we are more likely to sample a low energy fullerene. The likelihood of sampling a given canonical fullerene ID is proportional with E^Γ for that fullerene. Let us now look at what it would take to prepare the E^Γ energy directly in the amplitude using Quantum Amplitude Arithmetic.

3.3.3 Calculating E^Γ with Quantum Amplitude Arithmetic.

In this section we will encode a molecule as follows

$$\begin{aligned}&\text{SAA} |ID\rangle_{ID} |0\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_w,\Gamma_{1,\ldots,n},q_{1,\ldots,n},C_{1,\ldots,\lceil log(n+1)\rceil}} \\ &= |ID\rangle_{ID} |0\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_w} \bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_{1,\ldots,n}} \bigotimes_A |q_{A,\mu}\rangle_{q_{1,\ldots,n}} |0\rangle_{C_{1,\ldots,\lceil log(n+1)\rceil}}\end{aligned} \tag{3.42}$$

Again we will not go into detail with the SAA circuit.

We now present a circuit for computing $E^\Gamma - E^\Gamma_H$ on a one atom molecule, with 4 bits of precision, with a single shell and therefore Γ value. It is trivial to increase precision as it only involves straight forwards extension of the crimson region to use more qubits for q, Γ and C.

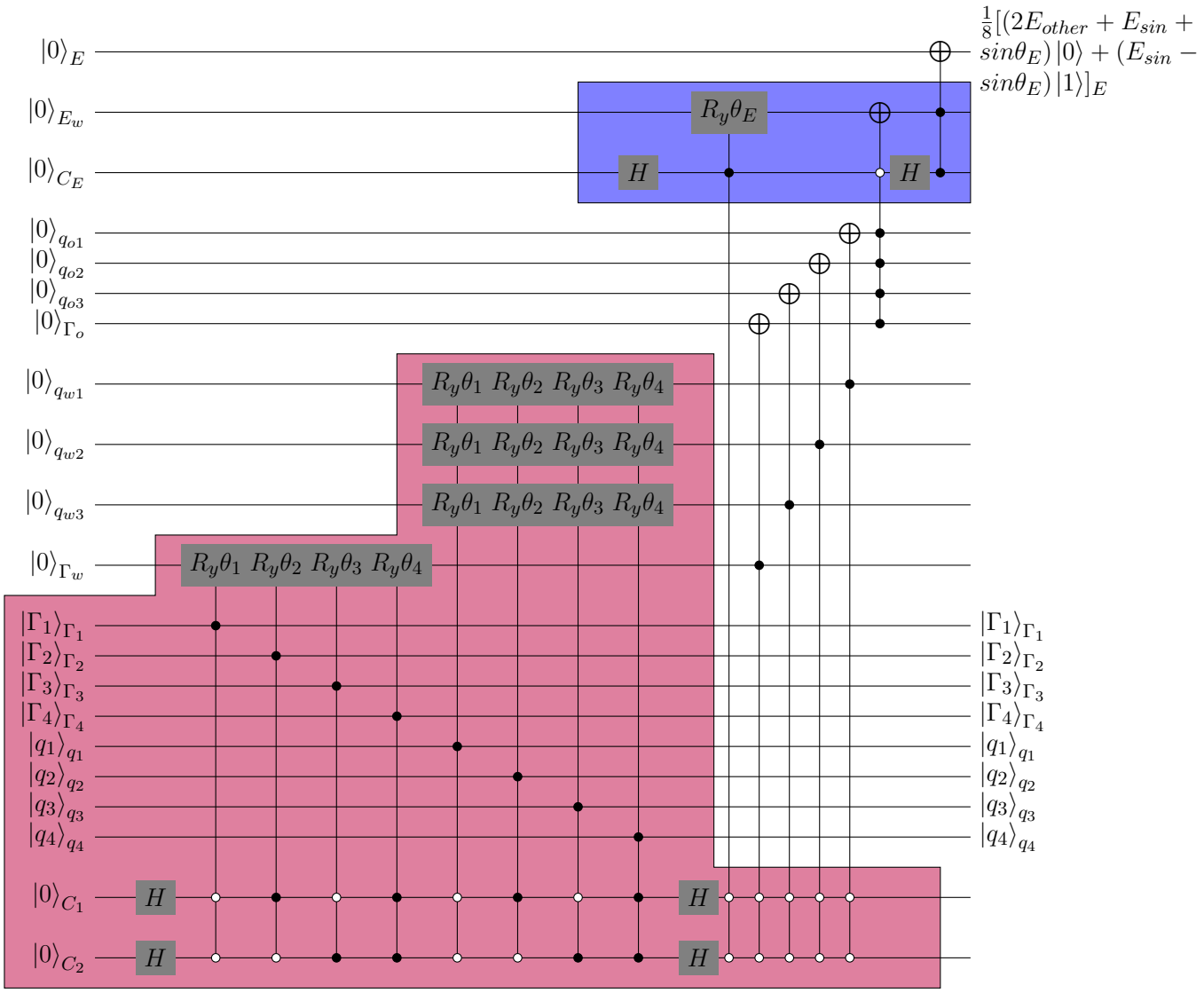

Figure 3.29.: Circuit for computing $E^{\Gamma} - E_H^{\Gamma}$ on a one atom molecule where the charge q and Γ are described using 4 bites. The crimson shaded area scales with the number of bits. The blue shaded area is for subtracting E_H^{Γ} and is completely optional.

This circuit is built from the addition and multiplication primitives introduced in the QA-Arithmetic paper[7]. We also do a slight modification to get subtraction.

If using more than one atom and shells the additional q's and Γ's can be added below and the result added to the final E_w register using an extra addition. C_E should be scaled appropriately as the base 2 logarithm of the number of q, Γ pairs plus 1.

Let us go though the mathematics of our 4 bit example to show that A.) it is sound and B.) make the areas that will be extended more obvious.

The controlled gate notation here is the following, t is the target register and $c1, c2, c3, \ldots$ are the control registers. $a, b, c, \ldots$ are all 1 except if there is a bar over the corresponding $c1, c2, c3, \ldots$ in which case it is 0.

$$CU_t^{c1,c2,c3,\ldots} = (U_t - I_t) \otimes |a\rangle \langle a|_{c1} \otimes |b\rangle \langle b|_{c2} \otimes |c\rangle \langle c|_{c3} \otimes \cdots + \sum_{\alpha,\beta,\zeta,\cdots \in \{0,1\}} I_t \otimes |\alpha\rangle \langle \alpha|_{c1} \otimes |\beta\rangle \langle \beta|_{c2} \otimes |\zeta\rangle \langle \zeta|_{c3} \otimes \ldots. \tag{3.43}$$

As we do the calculation we neglect writing out the $q_{1,2,3,4}, \Gamma_{1,2,3,4}$ as they never change throughout, we also neglect the registers outside of the crimson region for now. We begin by applying the Hadamard gates.

$$\begin{aligned} H_{C1} H_{C2} |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |00\rangle_{C_{(1,2)}} \rightarrow \\ \frac{1}{2}(|0000\rangle_{q_{w(1,2,3)},\Gamma_w} |00\rangle_{C_{(1,2)}} + \\ |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |01\rangle_{C_{(1,2)}} + \\ |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |10\rangle_{C_{(1,2)}} + \\ |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |11\rangle_{C_{(1,2)}}) \end{aligned} \tag{3.44}$$

When we apply the conditional rotation gates to a register such as Γ_w we do the following

$$\begin{aligned} CRy_{\Gamma_w}^{\Gamma_4,C_1,C_2}(2\theta_4) CRy_{\Gamma_w}^{\Gamma_3,\bar{C_1},C_2}(2\theta_3) CRy_{\Gamma_w}^{\Gamma_2,C_1,\bar{C_2}}(2\theta_2) CRy_{\Gamma_w}^{\Gamma_1,\bar{C_1},\bar{C_2}}(2\theta_1) \\ \frac{1}{2} \sum_{x_1,x_2 \in \{0,1\}} |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |x_1 x_2\rangle_{C_{(1,2)}} \rightarrow \\ \frac{1}{2}(CRy_{\Gamma_w}^{\Gamma_1}(2\theta_1) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |00\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_2}(2\theta_2) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |01\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_3}(2\theta_3) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |10\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_4}(2\theta_4) |0000\rangle_{q_{w(1,2,3)},\Gamma_w} |11\rangle_{C_{(1,2)}}) \rightarrow \end{aligned} \tag{3.45}$$

$$\begin{aligned} \frac{1}{2}(|000\rangle [\Gamma 1 (cos\theta_1 |0\rangle + sin\theta_1 |1\rangle) + (1 - \Gamma_1) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |00\rangle + \\ |000\rangle [\Gamma 2 (cos\theta_2 |0\rangle + sin\theta_2 |1\rangle) + (1 - \Gamma_2) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |01\rangle + \\ |000\rangle [\Gamma 3 (cos\theta_3 |0\rangle + sin\theta_3 |1\rangle) + (1 - \Gamma_3) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |10\rangle + \\ |000\rangle [\Gamma 4 (cos\theta_4 |0\rangle + sin\theta_4 |1\rangle) + (1 - \Gamma_4) |0\rangle]_{q_{w(1,2,3)},\Gamma_w} |11\rangle) \end{aligned}$$

To make our computations fit more easily on the page let us adopt the notation $|\Psi_i^t\rangle = t(cos\theta_i\,|0\rangle + sin\theta_i\,|1\rangle) + (1-t)\,|0\rangle$ before redoing the application using our new notation. We also apply the rotation gates for the q_w registers:

$$
\begin{aligned}
&CRy^{\Gamma_4,C_1,C_2}_{\Gamma_w}(2\theta_4)CRy^{\Gamma_3,\bar{C_1},C_2}_{\Gamma_w}(2\theta_3)CRy^{\Gamma_2,C_1,\bar{C_2}}_{\Gamma_w}(2\theta_2)CRy^{\Gamma_1,\bar{C_1},\bar{C_2}}_{\Gamma_w}(2\theta_1)\\
&CRy^{q_4,C_1,C_2}_{q_{w1}}(2\theta_4)CRy^{q_3,\bar{C_1},C_2}_{q_{w1}}(2\theta_3)CRy^{q_2,C_1,\bar{C_2}}_{q_{w1}}(2\theta_2)CRy^{q_1,\bar{C_1},\bar{C_2}}_{q_{w1}}(2\theta_1)\\
&CRy^{q_4,C_1,C_2}_{q_{w2}}(2\theta_4)CRy^{q_3,\bar{C_1},C_2}_{q_{w2}}(2\theta_3)CRy^{q_2,C_1,\bar{C_2}}_{q_{w2}}(2\theta_2)CRy^{q_1,\bar{C_1},\bar{C_2}}_{q_{w2}}(2\theta_1)\\
&CRy^{q_4,C_1,C_2}_{q_{w3}}(2\theta_4)CRy^{q_3,\bar{C_1},C_2}_{q_{w3}}(2\theta_3)CRy^{q_2,C_1,\bar{C_2}}_{q_{w3}}(2\theta_2)CRy^{q_1,\bar{C_1},\bar{C_2}}_{q_{w3}}(2\theta_1)\\
&\qquad\qquad\frac{1}{2}\sum_{x_1,x_2\in\{0,1\}}|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|x_1x_2\rangle =\\
&\frac{1}{2}(CRy^{q_1}_{q_{w1}}(2\theta_1)CRy^{q_1}_{q_{w2}}(2\theta_1)CRy^{q_1}_{q_{w3}}(2\theta_1)CRy^{\Gamma_1}_{\Gamma_w}(2\theta_1)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|00\rangle +\\
&\quad CRy^{q_2}_{q_{w1}}(2\theta_2)CRy^{q_2}_{q_{w2}}(2\theta_2)CRy^{q_2}_{q_{w3}}(2\theta_2)CRy^{\Gamma_2}_{\Gamma_w}(2\theta_2)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|01\rangle +\\
&\quad CRy^{q_3}_{q_{w1}}(2\theta_3)CRy^{q_3}_{q_{w2}}(2\theta_3)CRy^{q_3}_{q_{w3}}(2\theta_3)CRy^{\Gamma_3}_{\Gamma_w}(2\theta_3)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|10\rangle +\\
&\quad CRy^{q_4}_{q_{w1}}(2\theta_4)CRy^{q_4}_{q_{w2}}(2\theta_4)CRy^{q_4}_{q_{w3}}(2\theta_4)CRy^{\Gamma_4}_{\Gamma_w}(2\theta_4)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|11\rangle) \rightarrow\\
&\qquad\frac{1}{2}(|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle_{q_{w(1,2,3)},\Gamma_w}\,|00\rangle +\\
&\qquad\quad|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle_{q_{w(1,2,3)},\Gamma_w}\,|01\rangle +\\
&\qquad\quad|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle_{q_{w(1,2,3)},\Gamma_w}\,|10\rangle +\\
&\qquad\quad|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle_{q_{w(1,2,3)},\Gamma_w}\,|11\rangle)
\end{aligned}
\tag{3.46}
$$

Let us now apply the second set of Hadamard gates:

$$
\begin{aligned}
H_{C_1}H_{C_2}\frac{1}{2}(&|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle|00\rangle+\\
&|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle|01\rangle+\\
&|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle|10\rangle+\\
&|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle|11\rangle)\rightarrow\\
\frac{1}{4}(&|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle[|00\rangle+|01\rangle+|10\rangle+|11\rangle]+\\
&|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle[|00\rangle-|01\rangle+|10\rangle-|11\rangle]+\\
&|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle[|00\rangle+|01\rangle-|10\rangle-|11\rangle]+\\
&|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle[|00\rangle-|01\rangle-|10\rangle+|11\rangle])=\\
\frac{1}{4}&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle+|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle+|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle+|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|00\rangle+\\
\frac{1}{4}\Bigg(&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle-|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle+|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle-|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|01\rangle+\\
&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle+|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle-|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle-|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|10\rangle+\\
&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle-|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle-|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle+|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|11\rangle\Bigg)\\
=\frac{1}{4}&\left[|\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{q_1}\Psi_1^{\Gamma_1}\rangle+|\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{q_2}\Psi_2^{\Gamma_2}\rangle+|\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{q_3}\Psi_3^{\Gamma_3}\rangle+|\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{q_4}\Psi_4^{\Gamma_4}\rangle\right]|00\rangle+|M\rangle\\
=&|N\rangle+|M\rangle
\end{aligned}
\tag{3.47}
$$

Before the next step let us define:

$$q_{sin}=q_1 sin\theta_1+q_2 sin\theta_2+q_3 sin\theta_3+q_4 sin\theta_4 \tag{3.48}$$

$$q_{other}=q_1 cos\theta_1+q_2 cos\theta_2+q_3 cos\theta_3+q_4 cos\theta_4+4-q_1-q_2-q_3-q_4 \tag{3.49}$$

$$\Gamma_{sin}=\Gamma_1 sin\theta_1+\Gamma_2 sin\theta_2+\Gamma_3 sin\theta_3+\Gamma_4 sin\theta_4 \tag{3.50}$$

$$\Gamma_{other}=\Gamma_1 cos\theta_1+\Gamma_2 cos\theta_2+\Gamma_3 cos\theta_3+\Gamma_4 cos\theta_4+4-\Gamma_1-\Gamma_2-\Gamma_3-\Gamma_4 \tag{3.51}$$

$$E_{sin}=\Gamma_{sin}(q_{sin})^3 \tag{3.52}$$

$$
\begin{aligned}
E_{other}=&\Gamma_{other}(q_{other}^3+3q_{other}^2 q_{sin}+3q_{other}q_{sin}^2+q_{sin}^3)\\
&+\Gamma_{sin}(q_{other}^3+3q_{other}^2 q_{sin}+3q_{other}q_{sin}^2)
\end{aligned}
\tag{3.53}
$$

$$|\sigma_t\rangle=t_{other}|0\rangle+t_{sin}|1\rangle \tag{3.54}$$

$$\tag{3.55}$$

Let us now add in the $_o$ registers and apply the first 4 conditional not gates:

$$
\begin{aligned}
&CX_{q_{o1}}^{q_{w1},\bar{C}_1,\bar{C}_2} CX_{q_{o2}}^{q_{w2},\bar{C}_1,\bar{C}_2} CX_{q_{o3}}^{q_{w3},\bar{C}_1,\bar{C}_2} CX_{\Gamma_o}^{\Gamma_w,\bar{C}_1,\bar{C}_2} |0000\rangle_{E,q_{o(1,2,3)},\Gamma_o}(|N\rangle + |M\rangle) \rightarrow \\
&\qquad \left(CX_{q_{o1}}^{q_{w1}} CX_{q_{o2}}^{q_{w2}} CX_{q_{o3}}^{q_{w3}} CX_{\Gamma_o}^{\Gamma_w} |0000\rangle |N\rangle \right) + |0000\rangle |M\rangle \rightarrow \\
&\qquad\qquad |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |N\rangle + |0000\rangle |M\rangle
\end{aligned}
\tag{3.56}
$$

Now let us disregard everything in the crimson region except the C_1, C_2 control registers and do the final gates involving the $_o$ registers:

$$
\begin{aligned}
&H_{C_E} CX_{E_w}^{\bar{C}_E,q_{o1},q_{o2},q_{o3},\Gamma_o} CRy_{E_w}^{C_E,\bar{C}_1,\bar{C}_2}(\theta_E) H_{C_E} |000\rangle_{E,E_w,C_E} \frac{1}{4} \Big(|\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |0000\rangle_{q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Big) \rightarrow \\
&H_{C_E} \frac{1}{4} \Big(|0\rangle \frac{1}{\sqrt{2}} \Big[CX_{E_w}^{q_{o1},q_{o2},q_{o3},\Gamma_o} |0\rangle |0\rangle + Ry_{E_w}(\theta_E) |0\rangle |1\rangle \Big] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |00+0000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Big) \rightarrow \\
&H_{C_E} \frac{1}{4} \Big(|0\rangle \frac{1}{\sqrt{2}} \Big[|\sigma_E\rangle_{E_w} |0\rangle_{C_E} + (cos\theta_E |0\rangle + \sin\theta_E |1\rangle)_{E_w} |1\rangle_{C_E} \Big] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |00+0000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Big) \rightarrow \\
&\quad \frac{1}{4} \Big(|0\rangle \frac{1}{\sqrt{2}} \Big[|\sigma_E\rangle_{E_w} |+\rangle_{C_E} + (cos\theta_E |0\rangle + \sin\theta_E |1\rangle)_{E_w} |-\rangle_{C_E} \Big] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |000000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2} \Big)
\end{aligned}
\tag{3.57}
$$

Now we can neglect the $q_{o(1,2,3),\Gamma_o,C_1,C_2}$ registers too and do some preparatory manipulations before applying the final conditional not gate.

$$
\begin{aligned}
&\frac{1}{4}\Bigg(|0\rangle\frac{1}{\sqrt{2}}\Big[|\sigma_E\rangle_{E_w}|+\rangle_{C_E}+(cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|-\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\quad\frac{1}{4}\Bigg(|0\rangle\frac{1}{2}\Big[|\sigma_E\rangle_{E_w}|0\rangle_{C_E}+|\sigma_E\rangle_{E_w}|1\rangle_{C_E}+(cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|0\rangle_{C_E}\\
&\quad-(cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|1\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\quad\frac{1}{4}\Bigg(|0\rangle\frac{1}{2}\Big[(|\sigma_E\rangle+cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|0\rangle_{C_E}\\
&\qquad+(|\sigma_E\rangle-cos\theta_E|0\rangle-\sin\theta_E|1\rangle)_{E_w}|1\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[|\sigma_E\rangle+cos\theta_E|0\rangle+\sin\theta_E|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[|\sigma_E\rangle-cos\theta_E|0\rangle-\sin\theta_E|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[E_{other}|0\rangle+E_{sin}|1\rangle+cos\theta_E|0\rangle+\sin\theta_E|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[E_{other}|0\rangle+E_{sin}|1\rangle-cos\theta_E|0\rangle-\sin\theta_E|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[(E_{other}+cos\theta_E)|0\rangle+(E_{sin}+sin\theta_E)|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[(E_{other}-cos\theta_E)|0\rangle+(E_{sin}-sin\theta_E)|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\quad\frac{1}{8}\Bigg((E_{other}+cos\theta_E)|000\rangle+(E_{sin}+sin\theta_E)|010\rangle\\
&\qquad\quad+(E_{other}-cos\theta_E)|001\rangle+(E_{sin}-sin\theta_E)|011\rangle+6|000\rangle\Bigg)
\end{aligned}
\tag{3.58}
$$

In the last step we just did, see that we can 'select' either addition or subtraction of $sin\theta_E$ just by controlling our next gate on 010 or 011. We now apply the final conditional not gate:

$$\begin{aligned}
CX_E^{E_w,C_E}\frac{1}{8}\Big(&(E_{other}+cos\theta_E)\,|000\rangle+(E_{sin}+sin\theta_E)\,|010\rangle\\
&+(E_{other}-cos\theta_E)\,|001\rangle+(E_{sin}-sin\theta_E)\,|011\rangle+6\,|000\rangle\Big)\rightarrow\\
\frac{1}{8}\Big(&(E_{other}+cos\theta_E)\,|000\rangle+(E_{sin}+sin\theta_E)\,|010\rangle\\
&+(E_{other}-cos\theta_E)\,|001\rangle+X_E(E_{sin}-sin\theta_E)\,|011\rangle+6\,|000\rangle\Big)\rightarrow\\
\frac{1}{8}\Big(&(E_{other}+cos\theta_E)\,|000\rangle+(E_{sin}+sin\theta_E)\,|010\rangle\\
&+(E_{other}-cos\theta_E)\,|001\rangle+(E_{sin}-sin\theta_E)\,|111\rangle+6\,|000\rangle\Big)
\end{aligned}\tag{3.59}$$

After applying those gates we see that the amplitude on $|1\rangle_E$ across the whole state is

$$\tfrac{1}{8}(E_{sin}-sin\theta_E)=(\Gamma_1 sin\theta_1+\Gamma_2 sin\theta_2+\Gamma_3 sin\theta_3+\Gamma_4 sin\theta_4)(q_1 sin\theta_1+q_2 sin\theta_2+q_3 sin\theta_3+q_4 sin\theta_4)^3-sin\theta_E.$$

Let us say we know the E^Γ energy of some high energy molecule in the isomer space

$$E_H^\Gamma=(0b0.\Gamma_H)(0b0.q_H)^3$$

If we specify

$$\theta_i=arcsin\frac{1}{2^i},\quad \theta_E=arcsin[(0b0.\Gamma_H)(0b0.q_H)^3]$$

we get that

$$E_{sin}=(\frac{\Gamma_1}{2}+\frac{\Gamma_2}{2^2}+\frac{\Gamma_3}{2^3}+\frac{\Gamma_4}{2^8})(\frac{q_1}{2}+\frac{q_2}{2^2}+\frac{q_3}{2^3}+\frac{q_4}{2^8})^3=(0b0.\Gamma_1\Gamma_2\Gamma_3\Gamma_4)(0b0.q_1q_2q_3q_4)^3$$

and that

$$\frac{1}{8}(E_{sin}-sin\theta_E)=\frac{1}{8}[(0b0.\Gamma_1\Gamma_2\Gamma_3\Gamma_4)(0b0.q_1q_2q_3q_4)^3-(0b0.\Gamma_H)(0b0.q_H)^3]$$

. This is proportional to $E^\Gamma-E_H^\Gamma$ and thus the circuit is sound.

We will now take a look at the E^γ term.

3.3.4 Calculating E^{γ} using Quantum Digital Arithmetic

The E_{γ} term is formulated as follows:

$$\eta_{AB,ll'} = \frac{1}{2}\left[\eta_A(1+K_A^l)+\eta_B(1+K_B^{l'})\right] \tag{3.60}$$

$$R_{AB}^2 = (A_x-B_x)^2+(A_y-B_y)^2+(A_z-B_z)^2 \tag{3.61}$$

$$\gamma_{AB,ll'} = \frac{1}{\sqrt{R_{AB}^2+\eta_{AB,ll'}^{-2}}} \tag{3.62}$$

$$E_{\gamma} = \frac{1}{2}\sum_{A,B}\sum_{l\in A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'} \tag{3.63}$$

$$\tag{3.64}$$

For fullerenes we can view $\eta_{AB,ll'}$ as only dependant on the angular momenta l and l', so there are 4 configurations as there are only 2 shells for carbon in GFN2-xTB and the rest of the terms are constants. Additionally $l=0,l'=1$ and $l=1,l'=0$ are equivalent. Thus we don't actually have to compute η during the quantum circuit and can just bake in those 3 configurations as constants. A small circuit for calculating R_{AB}^2 could be made up of 3 repetitions of the DIFF^2 circuit described bellow which for 2 numbers a,b and an accumulator acc computes $acc+(a-b)^2$:

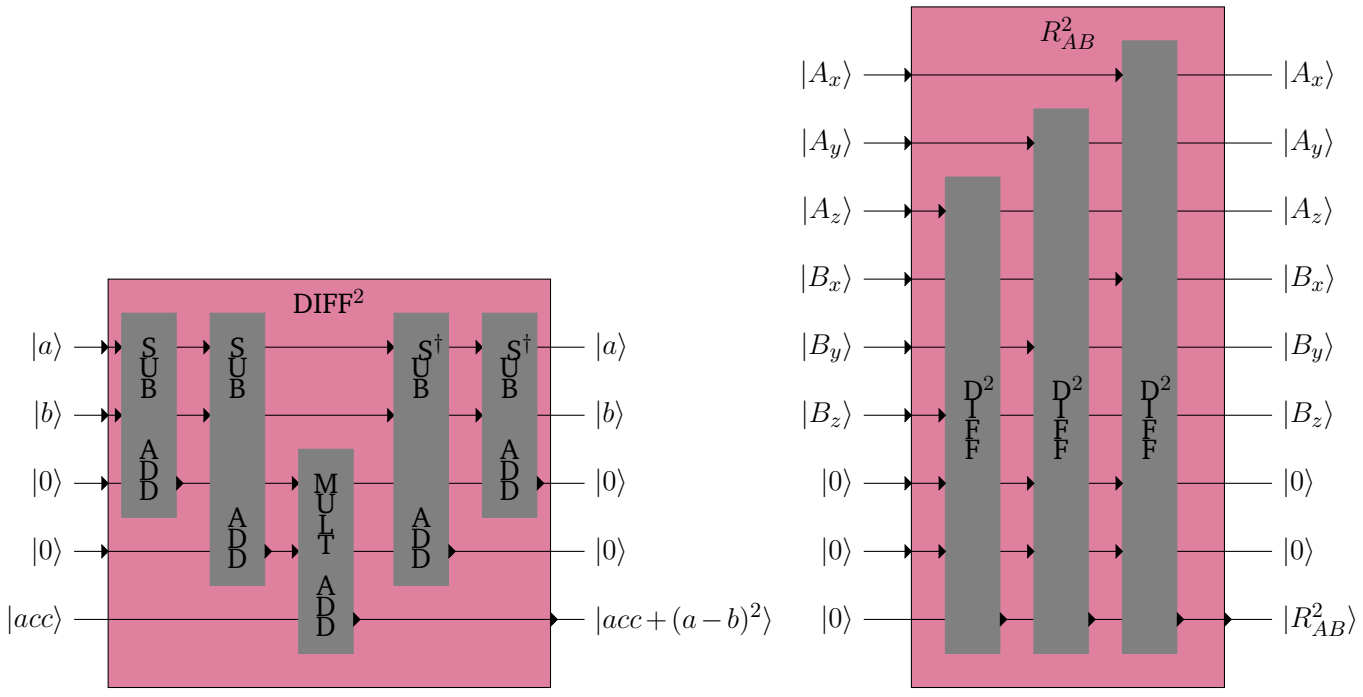

Figure 3.30.: Circuits for calculating $|a\rangle|b\rangle|acc\rangle \to |a\rangle|b\rangle|acc+(a-b)^2\rangle$ and $|A\rangle|B\rangle|0\rangle \to |A\rangle|B\rangle|R_{AB}^2\rangle$

Using this circuit along with a constant addition of $\eta_{AB,ll'}$ and the use of an inverse square root circuit[9] we get a fixed point approximation of $\gamma_{AB,ll'}$. We can use this with a QFT fused multiply add operation to get the inner sum of the E^γ term. There could potentially be a lot of pairs, however we can skip almost half with the following rewrite:

$$\begin{aligned} E^\gamma &= \frac{1}{2}\sum_{A,B}\sum_{l\in A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'} \\ &= \sum_{A}\sum_{l\in A}\left[\sum_{l'\in A} q_l q_{l'}\gamma_{AA,ll'} + \sum_{B>A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'}\right] \end{aligned} \tag{3.65}$$

This rearrangement is also something they do in the xtb software. Let us take take a brief look at the complexity of these quantum algorithms.

3.3.5 Complexity

The first circuit uses 5 QFT fused multiplication addition circuits. Each with $O(b^3)$[6] gates and no additional qubits, where b is the number of bits used to encode the numbers. Thus if we have encoded $\Gamma_\mu, q_{A_n,\mu}, \mu \in \{0,..,l\}, A_n \in \{0,..,o\}$ each using b bits we will need to perform $5lo$ multiplications, resulting in $5lo \cdot O(b^3) = O(lob^3)$ gates, on $m+b+bl+lob = O(m+lob)$ qubits. We can do trade-offs in circuit depth and amount of ancillary qubits by having multiple running sums e.g. in a binary tree pattern instead of immediately summing them all in the same register, resulting in a depth of $O(log_2(lo)b^3)$.

If we add in the sampling step we then have to run the state preparation circuit which adds $O(\log b)$ qubits and $O(b \log b)$ gates[7], however that is not enough to change the asymptotic runtime further.

For the second circuit we use $o(2b+1)$ multiple controlled rotation gates, at most $o \cdot log_2(b) + log_2(o) + 1$ Hadamard gates, and $5o+1$ multiple controlled not gates. This gives a circuit depth on the order of $O(ob)$.

TODO: E^γ

These complexities are not very impressive for a single computation, in which case we would get better results just running it classically. However if we have an isomer space with x isomers we would have to run it on n of them to have a probability of $\frac{n}{x}$ of having found the lowest energy isomer. For the quantum circuits this dependence on x is less extreme, and considering that x

is on the order of $O(c^9)$ for the C_c space, that enables us to potentially run this on much larger isomer spaces and still have a high probability of getting some of the lowest energy candidates. This works only because we took care to sample in such a way that the probability of getting a specific candidate is proportional to how different it is energetically to a known high energy isomer.

TODO: see if we can say how much better.

4 Related Work

4.1 xtb version 6.4.0

The xtb project has basic GPU support through the Nvidia Fortran compiler 'nvfortran', but this compiler fails on newer versions of the project. To get a version that has been officially tested with nvfortran, we have to go all the way back to version 6.4.0 from Feburary 2021. We have successfully made a reproducable build for this version and managed to run it, but the output seems much different from newer versions. There has been released 7 versions since, so doing any benchmarking comparisons with this older version likely would not be fair or representative.

4.2 dxtb

The dxtb project is a re-implementation of the xTB methods written in Python. This implementation has much better comments that actually explain the functions, but we discovered the project late into the process, and as such it has not been hugely beneficial for us. The project is no substitute for a GPU implementation, but it could have helped us with our own Python re-implementation.

5 Methodology

5.1 Porting the Reference Implementation

With all the equations from the xTB paper now in place, the next step is to implement them in code. Writing another Python re-implementation might seem rendundant, but the purpose of doing so is to start with a sequential version that is hopefully easier to reason about, and which is structured in a way that can more easily be translated to parallel GPU code.

We started implementing the equations from the paper directly, but found that it gave results different from the Fortran implementation. It is not that the Fortran code does it differently from what is described in the paper, but rather that there are details that might appear obvious to a chemist, which is not explicitly described in the paper. In [4] it is not specified in equation 9 for repulsion energy (5.1), that A and B cannot be the same atom index. They cannot be the same because R_{AB} is the distance between the two atoms, so if A and B is the same, then the distance is 0, which will result in division by zero. This makes sense intuitively because an atom does not repulse itself, but this is not always immediately apparent to someone outside the field.

$$E_{rep} = \frac{1}{2} \sum_{A,B} \frac{Z_A^{eff} Z_B^{eff}}{R_{AB}} e^{-\sqrt{\alpha_A \alpha_B}(R_{AB})^{(k_f)}} \tag{5.1}$$

In [10] they write that they use SCC, but they do not mention which method they use. It turns out that they use Broyden's method. The unit convertions between angström and bohr that are applied to the input geometry data, does also not seem to be mentioned anywhere.

Suffice to say, at the time we were not aware of all these details, so for the sake of correctness it was safer to follow the code rather than the papers. This approach came with its own challenges as we had to become acquainted with Fortran as a programming language while deciphering all its unintuitive quirks. Here is an overview of different challenges, inconsistencies, and confusing language quirks we have encountered while working with the xtb project.

- The project is written in object-oriented Fortran, which makes the execution flow difficult to follow.

- Arrays in Fortran start at 1 by default, but it is possible to specify that it should start at any other index. We have encountered a wide mix of start indices including 0, 1, 5, and 11.

- Variable names in Fortran are case-insensitive, and we encountered a variable that was defined with uppercase but referenced with lowercase.

- Fortran uses column-major indexing, so we have swapped the indices in Python.

- The goto statement exists in Fortran, and has been used inconsistently for something that should have been a simple for loop.

- Interfaces in Fortran are a way to overload a method, and it can be difficult to know which function it calls without digging into the subtle type differences of the arguments.

- In Fortran 'pure elemental' functions are pure functions that can be given an array in place of any single value argument of the same type. The function will essentially unpack the array and call the function with each element.

- Fortran uses parenthesis to index into an array. A subroutine is called by using the 'call' keyword, but pure functions does not. This makes it unclear whether we are calling a pure function or indexing into an array.

In addition to these confusing language specifics, there are also numerous structural oddities within the code, some of which we present below.

- The code authors often use symmetric pairing functions which are functions that map some multi-dimensional index into a flat 1-dimensional index. Specifically, in this code it maps pairs of indices (x, y) and (y, x) to the same index. This would be easier to understand with a 2-dimensional matrix, but the reason they do this is to save memory on symmetric matrices where only one of the triangular halves are needed. This understanding is difficult to come up with from the code, because the pairing function is named '`lin`', which says little about its effect, and there are no comments explaining what it is.

- The non-descriptive variable name leads us to the next point, being that this is a reoccuring problem caused by the fact that most of the names are abbreviated with three letters.

This makes it impossible to know what the values actually are and makes the codebase unapproachable for new contributors.

Some of the command-line arguments have also caused some confusion. We noticed that the program caches previous computations for a molecule, so running it twice on say a C200 molecule, will make the second run much faster. This is undesirable when we need to know what the total computational time is, and so to disable this behavior, the program has the flag seen in Figure 5.1.

```
1  --[no]restart
2      restarts calculation from xtbrestart (default = true)
```

Figure 5.1.: The xtb command-line flag that controls whether to continue the computation from the last run.

The description is ambiguous as it can both mean that we recompute the data stored in `xtbrestart`, or that we reuse the value and continue the computation from there. At first we thought that we would specify `--restart` because we want to restart the whole computation without any caching. In reality what we need is `--norestart` because it should not use `xtbrestart` which is the cache from the previous run. We believe that all of these points are worth taking note of for anyone new to the codebase.

When porting code from an implementation and a language with so many quirks and pitfalls, it becomes exceedingly important to have tests in place to catch these errors. Having a test suite that enables us to compare our implementation to the original has been indispensible to us for a fast iteration loop that gives insight into the mismatches and any potential patterns in the discrepancies.

5.2 Testing

The xtb codebase has a large footprint, and there is a lot of overlap between the xTB methods, so even focusing on just one of them still requires a great amount of code. The large amount of computations we have to deal with when porting the code makes proper validation especially important, as it becomes increasingly easy to make mistakes.With the discrepancies between the paper and the code, in addition to details that seem to go unexplained, we realized that the obvious approach forward was to lean towards the existing implementation rather than the paper.

One of the hurdles from testing against an existing program becomes the lack of transparency regarding the logic that takes place between the initial input and the final output presented to the user. Thankfully, the source code is publicly available, which allows for easy manipulation of the original flow of execution, thus avoiding the hassle of testing against a black box or the need to resort to methods of reverse engineering.

On behalf of these considerations we decided to write patches that allow to intercept the arguments and results of arbitrary functions just by running the program as normal. This gave us the ability to implement and test abritrary functions in an isolated manner, by eliminating the need to compute their arguments.

5.2.1 Towards Reproducability with Nix

An important part of any software is reproducability, and applying certain patches in certain scenarios is something that should preferably be automated, reproducable, and optimally also portable. This is especially important for this approach to validation as it requires a way to reproduce a specific version of xtb linked to the same versions of dependencies. Essentially an exact copy of the original shell environment to ensure that patches work, results are the same, and no new bugs appear in the program itself or its dependencies. All of this should be achievable without having to add, remove, downgrade or upgrade system packages on your system.

The well-known contenders for this is any of the numerous containerization solutions on Linux, such as Docker, Podman, LXC etc. There are some problems with these options though, one being that it can be difficult to truly reproduce package versions without saving the resulting container image, another being that it does not solve the problem of having multiple versions of the same package installed. Some other notable limitations are that it limits the process to run within the container and passing in a GPU or other hardware can be nefariously difficult. A container also does not have access to the X or Wayland session needed to run GUI applications, though that is not currently relevant in this case.

Another approach which has been growing in popularity in recent years are tools that take unique approaches to package management in order to make not only packages reproducable, but also shell environments, system configurations and other forms of "outputs". Two such popular package managers are Nix from the Nix team and Guix from the GNU foundation. Nix is arguably the more popular option and it is also the solution that has been chosen for this project.

Nix is an umbrella term that can refer to either the Nix functional programming language, the package manager, or the Nix based Linux distribution NixOS. The language and package manager go hand in hand and can be used on any Linux distribution. As such, NixOS is not required for the needs of this project and will not be mentioned going forward.

Nix does not follow the Unix Filesystem Hierarchy Standard (FHS), which brings with it some challenges, but this fundamental difference from other package managers is a major part of what makes Nix so powerful. Rather than installing packages into the usual system paths like '/bin', '/lib' etc. Nix installs everything into a read-only path called the Nix store under '/nix/store'. Everything in the Nix store is a result of a core concept in Nix called a derivation, which is essentially a build task to produce some output of files into the Nix store. All outputs into the store is marked with a custom hash in the filename called a NAR hash. These fundamental ideas fix some common problems such as circular dependencies and allow having multiple versions of the same package installed as they will simply coincide in the Nix store with different NAR hashes.

The typical binary on Linux is dynamically linked against the FHS compliant paths and it is not uncommon to have them hardcoded either. To make use of the packages in the Nix store, it is required to either recompile the program against the store paths, or in the case of proprietary software, patching the ELF header is needed to change the path to the interpreter and to dynamically linked libraries. Thankfully the Nix package repository 'NixPkgs' is the largest and freshest out there[1], so as a typical user doing this is rarely needed. Nixpkgs is a version-controlled repository on GitHub, so using older versions of packages even alongside newer ones, is fairly trivial as it simply requires fetching multiple revision of the repository.

This along with the previously mentioned features have allowed a greatly simplified process of not only running the newest version 6.7.1 of the xtb program, but also running the much older nvfortran compatible version 6.4.0 alongside it. NixPkgs is also a collection of library functions, and the helper functions for making derivations called 'mkDerivation' make it easy to define all the stages of packaging a program including unpacking, patching, building, checking, and installing the files. With this, the whole pipeline of patching, compiling, running, and passing the data over to the Python validation tests can be achieved with a single shell command.

```
> nix run .#cmp-impls
```

This command takes the form '`nix run <path to flake>#<output>`'. Path to flake refers to a file-tree whose root directory contains a file called 'flake.nix'. Nix flakes is an experimental but

[1] `https://repology.org/repositories/graphs`

widely adopted feature, which provides a standard way to write Nix expressions and a way to manage their dependencies through a version-pinned lock file. The 'flake.nix' file follows a uniform naming schema for declaring inputs and outputs, where inputs are the dependencies, and outputs are Nix expressions to be exposed. The new Nix command-line interface needed to interact with flakes is naturally also an experimental feature that has to be enabled explicitly. The run command instructs Nix to build and run the derivation 'cmp-impls', which is defined as an app in the flake outputs.

```
{
  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/nixos-unstable";
  };

  outputs = { self, nixpkgs, ... }: {
    apps."x86_64-linux" = let
      pkgs = nixpkgs.legacyPackages."x86_64-linux";
      ...
    in {
      "cmp-impls" = let
        python = (pkgs.python3.withPackages (python-pkgs: with python-pkgs; [
          numpy scipy cvxopt
        ]));
      in {
        type = "app";
        program = toString (pkgs.writeShellScript "cmp-impls" ''
          PYTHONPATH=${pkgs.lib.cleanSource ./xtb-python} exec
          ↪ ${python}/bin/python \
            ${./xtb-python/cmp_impls.py} ${xtb_test_data}
        '');
      };
    };
    ...
  };
}
```

Figure 5.2.: This is a snippet of our Nix flake which shows the app declaration for running our test suite. The app depends on the derivation that generates the test data.

Python is declared with the required packages and is then used in the app to call the `cmp_impls.py` script. The script is called with the test data acquired from running the xtb program. This data comes from another derivation which executes patched versions of xtb and dftd4 on a C_{200} fullerene to get the relevant function arguments and results as binary files.

```
xtb_test_data = builtins.derivation {
  name = "xtb-test-data";
  system = "x86_64-linux";
  builder = "${pkgs.bash}/bin/bash";
  src = ./xtb-python/data/C200.xyz;
  args = ["-c" ''
    PATH=$PATH:${pkgs.coreutils}/bin
    mkdir -p ./calls/{build_SDQH0,coordination_number,\
      dim_basis,dtrf2,electro,form_product,get_multiints,\
      h0scal,horizontal_shift,multipole_3d,newBasisset, olapp}
    ${xtb}/bin/xtb $src
    ${dftd4}/bin/dftd4 $src
    mv calls $out
  ''];
};
```

Figure 5.3.: This is the Nix derivation for generating the test data from our patched versions of xtb and dftd4.

The directories for the binary files are created in advance as checking whether they exist when writing the binary files has a large overhead. This derivation in turn uses derivations for xtb and dftd4. Luckily dftd4 is already in NixPkgs, but it still needs to be patched in order to extract the required data for validation. Thankfully the `mkDerivation` function used in NixPkgs makes overriding and patching a package very straightforward.

```
dftd4 = (pkgs.dftd4.overrideAttrs (finalAttrs: previousAttrs: {
  src = pkgs.fetchFromGitHub {
    owner = "dftd4";
    repo = "dftd4";
    rev = "502d7c59bf88beec7c90a71c4ecf80029794bd5e";
    hash = "sha256-FEABtBAZK0xQ1P/Pbj5gUuvKf8/ZLITXaXYB+btAY/8=";
  };
  buildInputs = [ multicharge ] ++ previousAttrs.buildInputs;
  doCheck = false;
  patches = previousAttrs.patches ++ [
    ./nix/patches/dftd4/use_gfn2.patch
    ./nix/patches/dftd4/log_args_and_outputs.patch
  ];
}));
```

Figure 5.4.: This is the Nix expression for overriding the dftd4 package derivation. We essentially update dftd4 to a newer revision and add patches to log arguments and results to a binary file, and another patch to use GFN2.

The version is bumped by overriding the source, and the multicharge project is added from NixPkgs and also bumped as a requirement of this newer version. Some of the tests were timing out, so they have been disabled by setting `doCheck` to false. Lastly the patches are applied by providing the relevant patch files.

The xtb project and two of its dependencies, namely CPCM-X and numsa are not in NixPkgs, so we had to package them ourselves. This is one of our contributions, which makes xtb easily available to all Linux distributions. It is a reproducable package that works the same on different machines regardless of their respective system packages. We have also packaged the old version 6.4.0 of xtb, which requires various old dependencies and is the only way currently to run xtb on a GPU through nvfortran. We have shared this achievement on the xtb GitHub repository, and others have since mentioned our GitHub issue to highlight the challenges of getting the GPU version to run.

5.2.2 Patching xtb

Now we have a way to apply patches and repoduce our tests. Next we will dive into what the patches do and how they are used. All the patches follow the structure seen in Figure 5.5 where the original function is prefixed with a 'g', such that the new wrapper function will be called instead. The wrapper function writes the function arguments to a binary file before calling the actual function before finally writing the result of the function to the same binary file. Writing a file for each call to a function is a bit excessive and will produce a very large amount of files, so a threshold has been used to create an upperbound on the number of files that can be created for each function.

```
+   logical :: hit_threshold
+   integer :: u
+   character(len=200) :: path
+
+   hit_threshold = testfile_path('electro', path)
+   if (.not.hit_threshold) then
+     open(newunit=u, file=trim(path), form='unformatted', access='stream')
+     write(u) nbf
+     write(u) size(H0), H0
+     write(u) size(P, 1), size(P, 2), P
...
+     if (allocated(ies%thirdOrder%atomicGam)) then
+       write(u) size(ies%thirdorder%atomicgam), ies%thirdorder%atomicgam
+     else
+       write(u) 0
+     end if
...
+     write(u) size(ies%jmat, 1), size(ies%jmat, 2), ies%jmat
+     write(u) size(ies%shift), ies%shift
+   end if
+
+   call gelectro(n,at,nbf,nshell,ies,H0,P,dq,dqsh,es,scc)
+
+   if (.not.hit_threshold) then
+     write(u) es
+     write(u) scc
+     close(u)
+   end if
```

Figure 5.5.: This is a diff file for the electro energy function. A diff file reflects the changes between two files and can be used to patch code by applying these changes. All our patches follow this structure of writing the arguments to a file, then running the original function before finally writing the result to the same binary file.

5.2.3 Implementing the Tests

With the binary files containing the arguments and results, we now have all the data necessary to compare against our Python implementation. We have made a test suite in the file `cmp-impls.py` where all tests follow these same steps:

1. Load an deserialize a binary file for the appropriate function

2. Call the corresponding Python function with the deserialized arguments

3. Compare the result against the deserialized Fortran result

4. Repeat until there are no more binary files for this function

```python
def test_electro():
    fn_name = "electro"
    for i, file_path in enumerate(glob.glob(f'{directory}/{fn_name}/*.bin')):
        with open(file_path, 'rb') as f:
            def read_ints(n=1):
                return np.fromfile(f, dtype=np.int32, count=n)

            nbf = read_ints(1)[0]
            H01 = read_ints(1)[0]
            H0 = np.fromfile(f, dtype=np.float64, count=H01)
            m, n = read_ints(2)
            P = np.fromfile(f, dtype=np.float64, count=m * n).reshape((n, m))
            ...
            atomicGam1 = read_ints(1)[0]
            atomicGam = None  if atomicGam1 == 0
                              else np.fromfile(f, dtype=np.float64,
                              ↪  count=atomicGam1)
            ...
            es_res, scc_res = read_reals(2)
            es, scc = electro(nbf, H0, P, dq, dqsh, atomicGam, shellGam, jmat,
            ↪  shift)

            is_equal(es, es_res, "es", fn_name)
            is_equal(scc, scc_res, "scc", fn_name)

    print(f"matches! [{fn_name}]")
```

Figure 5.6.: This is some of the code from the test for the electro function. It shows how we iterate through each binary file, deserialize its data, call the corresponding Python implementation, and then compare the results.

6 Results

6.1 Our Contributions

In this section we will present and reiterate the contributions we have made as a result of this project.

We have contributed with a simplified Python version of the original GFN2-xTB Fortran implementation. The implementation and validation of the dispersion term is not complete, and we have not implemented the self-consistent charges that is used to iteratively rerun the computations to improve the accuracy of the result. Given the size of the reference implementation we are quite satisfied with this. The Python implementation should hopefully make the GFN2-xTB algorithm more approachable for our successors and give a good foundation for making the lockstep parallel GPU implementation.

We have contributed a testing framework for comparing results against the original Fortran implementation. We utilize Nix to ensure tests that are reproducible, regardless of Linux distribution and system configuration. In connection to this, we have also contributed a reproducible and easy way to build, run, and patch xtb. This also includes its dependencies, and even the older GPU compatible version 6.4.0 of xtb known from GitHub issues to be difficult to run.

This report contributes a walkthrough of the GFN2-xTB algorithm with code snippets that directly links to the equations in a way that should be more easily digestible for a computer scientist. As an extension of this, we have presented insights and ideas on how to approach a massively lockstep implementation by optimizing code snippets for our problem domain, and transforming them such that they adhere to the requirements of a lockstep kernel.

We have also presented a case study with the NVIDIA A100 datacenter GPU to show how GPU architectures are structured and how to use the technical specifications to estimate hardware requirements.

6.2 Validation

All of our tests pass indicating that our results match the reference implementation. We had to introduce a threshold for some of the comparisons as they came extremely close. Figure 6.1 shows the maximum squared deviations of all tests for the electrostatic term. This term returns the isotropic electrostatic energy and the self-consistent charges, so there are two plots to show both.

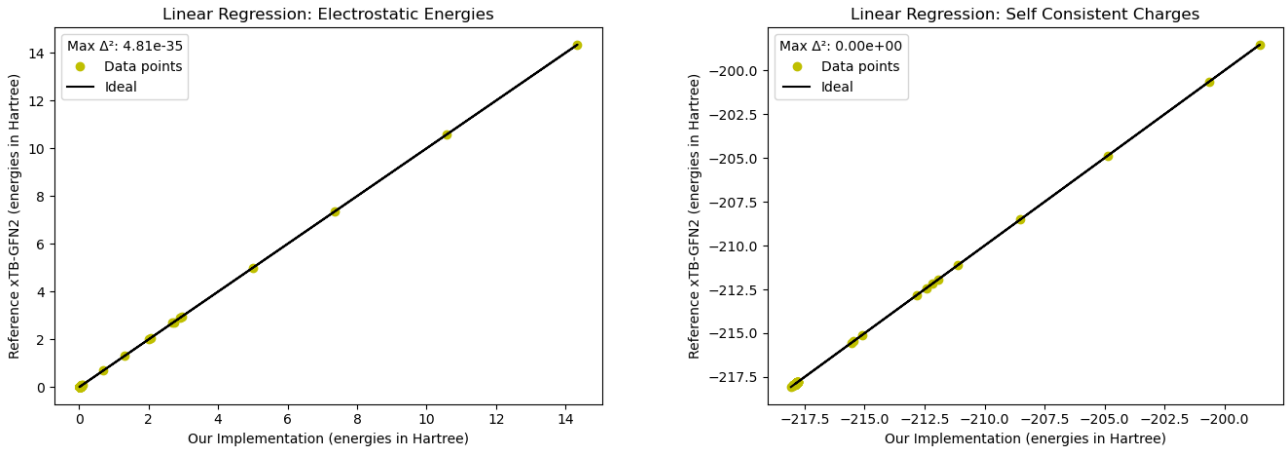

Figure 6.1.: Linear regressions showing how much the Python implementation for the electrostatic term deviates from the Fortran results.

6.3 Benchmarks

Reflections

7

8 Future Work

9 Extended Hückel Theory Matrix for GFN2-xTB

$$
\begin{aligned}
H_{\mu\nu}^{EHT} = & \frac{1}{2} K_{AB}^{ll'} S_{\mu\nu} (H_{\mu\mu} + H_{\nu\nu}) \\
& \cdot X(EN_A, EN_B) \\
& \cdot \Pi(R_{AB}, l, l') \\
& \cdot Y(\zeta_l^A, \zeta_{l'}^B), \forall \mu \in l(A), \nu \in l'(B)
\end{aligned}
\tag{9.1}
$$

where μ and ν are AO indecies, l and l' index shells. Both AO's are associated with an atom labled A and B. $K_{AB}^{ll'}$ is a element and shell specific fitted constant however, in GFN2 it only depends on the shells. $S_{\mu\nu} = \langle \phi_\mu | \phi_\nu \rangle$ is just the overlap of the orbitals. In GFN2 $H_{\kappa\kappa} = h_A^l - \delta h_{CN_A'}^l CN_A'$ where CN_A' is the modified GFN2-type Coordinate Number for the element of atom A.

$$
\begin{aligned}
CN_A^{'} = & \sum_{B \neq A}^{N_{\text{atoms}}} (1 + e^{-10(4(R_{A,\text{cov}} + R_{B,\text{cov}})/3R_{AB} - 1)})^{-1} \\
& \times (1 + e^{-20(4(R_{A,\text{cov}} + R_{B,\text{cov}} + 2)/3R_{AB} - 1)})^{-1}
\end{aligned}
\tag{9.2}
$$

h_A^l and $\delta h_{CN_A'}^l$ are both fitted constants. EN_A is the electronegativity of the element of atom A, given in the original `xtb` code.

$$X(EN_A, EN_B) = 1 + k_{EN} \Delta EN_{AB}^2 \tag{9.3}$$

$$k_{EN} = 0.02 \text{ in GFN2} \tag{9.4}$$

$$\Delta EN_{AB}^2 = (EN_A - EN_B)^2 \tag{9.5}$$

$$\Pi(R_{AB}, l, l') = \left(1 + k_{A,l}^{\text{poly}} \left(\frac{R_{AB}}{R_{\text{cov},AB}}\right)^{\frac{1}{2}}\right) \left(1 + k_{B,l'}^{\text{poly}} \left(\frac{R_{AB}}{R_{\text{cov},AB}}\right)^{\frac{1}{2}}\right) \tag{9.6}$$

$R_{\text{cov},AB}$ are the summed covalent radii ($R_{\text{cov},A} + R_{\text{cov},B}$), e.g. $R_{\text{cov},H} = 0.32$, $R_{\text{cov},C} = 0.75$ are given in the original `xtb` code. $k^{\text{poly}}_{A,l}$ and $k^{\text{poly}}_{B,l'}$ are element and shell specific constants.

$$Y(\zeta^A_l, \zeta^B_{l'}) = \left(\frac{2\sqrt{\zeta^A_l \zeta^B_{l'}}}{\zeta^A_l + \zeta^B_{l'}} \right)^{\frac{1}{2}} \tag{9.7}$$

Here, ζ^A_l are the STO exponents of the GFN2-xTB AO basis.
Slater Type Orbitals are defined as such:

$$\chi_{\zeta,n,l,m}(r,\theta,\varphi) = NY_{l,m}(\theta,\varphi) r^{n-1} e^{-\zeta r} \tag{9.8}$$

N is a normalisation constant, Y are spherical harmonic funtions, n, l, m are the quantum numbers for the AO. r, θ, φ are polar 3D coordinates. ζ determines the radial extent of the STO, a large value gives rise to a function that is "tight" around the nucleus and a small value gives a more "diffuse" function. This ζ is the one mentioned in the Y term of E_{EHT} and is a value fitted when constructing the basis set, thus it is given to us.

9.1 Fock Matrix for GFN2-xTB

$$F^{GFN2-xTB}_{\mu\nu} = H^{EHT}_{\mu\nu} + F^{IES+IXC}_{\mu\nu} + F^{AES}_{\mu\nu} + F^{AXC}_{\mu\nu} + F^{D4}_{\mu\nu}, \quad \forall \mu \in A, \nu \in B \tag{9.9}$$

9.1.1 Isotropic Electrostatic and Exchange-correlation contribution

$$\begin{aligned} F^{IES+IXC}_{\mu\nu} = &-\frac{1}{2} S_{\mu\nu} \sum_C \sum_{l''} (\gamma_{AC,ll''} + \gamma_{BC,l'l''}) q_{C,l''} \\ &-\frac{1}{2} S_{\mu\nu} (q^2_{A,l} \Gamma_{A,l} + q^2_{B,l'} \Gamma_{B,l'}) \end{aligned} \tag{9.10}$$

l, l', l'' being the angular momenta of the orbitals μ, ν and each of C's orbitals.

$$\Gamma_{A,l} = K^{\Gamma}_l \Gamma_A \tag{9.11}$$

K_l^Γ is a shell specific constant common for all elements and Γ_A is an element specific constant.

$$\gamma_{AB,ll'} = \frac{1}{\sqrt{R_{AB}^2 + \eta_{AB,ll'}^{-2}}} \tag{9.12}$$

$$\eta_{AB,ll'} = \frac{1}{2}\left[\eta_A(1+k_A^l) + \eta_B(1+k_B^{l'})\right] \tag{9.13}$$

q_l is a partial Mulliken charge. η_A and η_B are element-specific fit parameters, while k_A^l and $k_B^{l'}$ are element-specific scaling factors for the individual shells ($k_A^l = 0$ when $l = 0$).

$$GAP_A = \sum_{l \in A} q_{A,l} \tag{9.14}$$

$$q_{A,l} = \sum_{l' \in B} P_{ll'} S_{ll'} = GOP_l \tag{9.15}$$

9.1.2 Anisotropic Electrostatic and Exchange-correlation contribution

$$\begin{aligned} F_{\mu\nu}^{AES} + F_{\mu\nu}^{AXC} &= \frac{1}{2} S_{\mu\nu} \left[V_S(\boldsymbol{R}_B) + V_S(\boldsymbol{R}_C)\right] \\ &+ \frac{1}{2} \boldsymbol{D}_{\mu\nu}^T \left[\boldsymbol{V}_D(\boldsymbol{R}_B) + \boldsymbol{V}_D(\boldsymbol{R}_C)\right] \\ &+ \frac{1}{2} \sum_{\alpha,\beta \in \{x,y,z\}} Q_{\mu\nu}^{\alpha\beta} \left[V_Q^{\alpha\beta}(\boldsymbol{R}_B) + V_Q^{\alpha\beta}(\boldsymbol{R}_C)\right] \end{aligned} \tag{9.16}$$

$$\boldsymbol{D}_{\mu\nu}^T = \begin{pmatrix} D_{\mu\nu}^x & D_{\mu\nu}^y & D_{\mu\nu}^z \end{pmatrix} \tag{9.17}$$

(9.18)

$$\begin{aligned} V_S(\boldsymbol{R}_C) = \sum_A \Bigg\{ & \boldsymbol{R}_C^T \Big[f_5(R_{AC}) \boldsymbol{\mu}_A R_{AC}^2 - \boldsymbol{R}_{AC} 3 f_5(R_{AC}) (\boldsymbol{\mu}_A^T \boldsymbol{R}_{AC}^2) \\ & - f_3(R_{AC}) q_A \boldsymbol{R}_{AC} \Big] - f_5(R_{AC}) \boldsymbol{R}_{AC}^T \boldsymbol{\Theta}_A \boldsymbol{R}_{AC} - f_3(R_{AC}) \boldsymbol{\mu}_A^T \boldsymbol{R}_{AC} \\ & + q_A f_5(R_{AC}) \frac{1}{2} \boldsymbol{R}_C^2 \boldsymbol{R}_{AC}^2 - \frac{3}{2} q_A f_5(R_{AC}) \sum_{\alpha\beta} \alpha_{AB} \beta_{AB} \alpha_C \beta_C \Bigg\} \\ & + 2 f_{XC}^{\mu_C} \boldsymbol{R}_C^T \boldsymbol{\mu}_C - f_{XC}^{\Theta_C} \boldsymbol{R}_C^T \Big[3\boldsymbol{\Theta}_C - \mathrm{Tr}(\boldsymbol{\Theta}_C) \boldsymbol{I} \Big] \boldsymbol{R}_C \end{aligned} \tag{9.19}$$

QUESTION: Should this not be $R_C^{2,T}$, in line 3, term 1?

$$
\begin{aligned}
V_D(\boldsymbol{R}_C) = \sum_A \Bigg[& \boldsymbol{R}_{AC} 3 f_5(R_{AC})(\boldsymbol{\mu}_A^T \boldsymbol{R}_{AC}) - f_5(R_{AC}) \boldsymbol{\mu}_A R_{AC}^2 + f_3(R_{AC}) q_A \boldsymbol{R}_{AC} \\
& - q_A f_5(R_{AC}) \boldsymbol{R}_C R_{AC}^2 + 3 q_A f_5(R_{AC}) \boldsymbol{R}_{AC} \sum_\alpha \alpha_C \alpha_{AC} \Bigg] \\
& - 2 f_{XC}^{\mu_C} \boldsymbol{\mu}_C - 2 f_{XC}^{\Theta_C} \Big[3 \boldsymbol{\Theta}_C - \mathrm{Tr}(\boldsymbol{\Theta}_C) \boldsymbol{I} \Big] \boldsymbol{R}_C
\end{aligned}
\tag{9.20}
$$

$$
\begin{aligned}
V_Q^{\alpha\beta}(\boldsymbol{R}_C) = & - \sum_A q_A f_5(R_{AC}) \left[\frac{3}{2} \alpha_{AC} \beta_{AC} - \frac{1}{2} R_{AB}^2 \right] \\
& - f_{XC}^{\Theta_C} \left[3 \boldsymbol{\Theta}_C^{\alpha\beta} - \delta_{\alpha\beta} \sum_\alpha \boldsymbol{\Theta}_C^{\alpha\alpha} \right]
\end{aligned}
\tag{9.21}
$$

$\boldsymbol{\mu}_A$ is the cumulative atomic dipole moment of atom A and $\boldsymbol{\Theta}_A$ is the corresponding traceless quadrupole moment. Traceless simply means that the sum of the diagonal elements is 0. The curly braces and brackets are used in the same way as normal parenthesis for showing order of operations. q_A is the atomic charge of atom A.

$$\Theta_A^{\alpha\beta} = \frac{3}{2} \theta_A^{\alpha\beta} - \frac{\delta_{\alpha\beta}}{2} \left(\theta_A^{xx} + \theta_A^{yy} + \theta_A^{zz} \right) \tag{9.22}$$

$$\theta_A^{\alpha\beta} = \sum_{l' \in A} \sum_l P_l \left(\alpha_A D_{ll'}^\beta + \beta_A D_{ll'}^\alpha - \alpha_A \beta_A S_{ll'} - Q_{ll'}^{\alpha\beta} \right) \tag{9.23}$$

$$q_A = Z_A - GAP_A \tag{9.24}$$

$$\mu_A^\alpha = \sum_{l' \in A} \sum_l P_{l'l} \left(\alpha_A S_{l'l} - D_{l'l}^\alpha \right) \tag{9.25}$$

$$D_{ll'}^\alpha = \langle \phi_l | \alpha_i | \phi_{l'} \rangle = \langle \phi_l(\alpha_i) | \alpha_i | \phi_{l'}(\alpha_i) \rangle = \int \alpha_i \phi_l^*(\alpha_i) \phi_{l'}(\alpha_i) d\alpha_i \tag{9.26}$$

$$Q_{ll'}^{\alpha\beta} = \langle \phi_l | \alpha_i \beta_i | \phi_{l'} \rangle = \langle \phi_l(\alpha_i) | \alpha_i \beta_i | \phi_{l'}(\beta_i) \rangle = \int \int \alpha_i \beta_i \phi_l^*(\alpha_i) \phi_{l'}(\beta_i) d\alpha_i d\beta_i \tag{9.27}$$

α and β are Cartesian components labled $(x, y, z)^T$ with atom A being centered in $\boldsymbol{R}_A = (x_i, y_i, z_i)^T$ where i is a form of pointer/label dereferencing. $\delta_{\alpha\beta}$ is just the delta function, i.e is is 1 if α and β are the same label and 0 otherwise, this serves to include the term only for the diagonal.

$$\boldsymbol{\Theta}_A = \begin{pmatrix} \Theta_A^{xx} & \Theta_A^{xy} & \Theta_A^{xz} \\ \Theta_A^{yx} & \Theta_A^{yy} & \Theta_A^{yz} \\ \Theta_A^{zx} & \Theta_A^{zy} & \Theta_A^{zz} \end{pmatrix} \tag{9.28}$$

$$\boldsymbol{\mu}_A = \begin{pmatrix} \mu_A^x & \mu_A^y & \mu_A^z \end{pmatrix}^T \tag{9.29}$$

$$\boldsymbol{R}_{AB} = \boldsymbol{R}_A - \boldsymbol{R}_B \tag{9.30}$$

$$R_{AB} = \sqrt{(\boldsymbol{R}_{AB}^x)^2 + (\boldsymbol{R}_{AB}^y)^2 + (\boldsymbol{R}_{AB}^z)^2} \tag{9.31}$$

$$f_n(R_{AB}) = \frac{f_{damp}(a_n, R_{AB})}{R_{AB}^n} = \frac{1}{R_{AB}^n} \frac{1}{1 + 6\left(\frac{R_0^{AB}}{R_{AB}}\right)^{a_n}} \tag{9.32}$$

$$R_0^{AB} = 0.5(R_0^{A\prime} + R_0^{B\prime}) \tag{9.33}$$

$$R_0^{A\prime} = \begin{cases} R_0^A + \frac{R_{max} - R_0^A}{1 + exp[-4(CN_A' - N_{val} - \Delta_{val})]} & \text{if } N_{val} \text{ is given} \\ 5.0 \text{ bohrs} & \text{otherwise} \end{cases} \tag{9.34}$$

$$R_{max} = 5.0 \text{ bohrs} \tag{9.35}$$

$$\Delta_{val} = 1.2 \tag{9.36}$$

R_0^A is a fitted value for 12 elements and 5.0 for the rest. a_n are adjusted global parameters. Where $f_{XC}^{\mu_A}$ and $f_{XC}^{\Theta_A}$ are fitted values.

9.1.3 Dispersion contribution

$$F_{\mu\nu}^{D4} = -\frac{1}{2} S_{\mu\nu}(d_A + d_B), \forall \mu \in A, \nu \in B \tag{9.37}$$

$$\begin{aligned} d_A = \sum_r^{N_{A,ref}} \frac{\partial \xi_A^r(q_A, q_{A,r})}{\partial q_A} \sum_B \sum_s^{N_{B,ref}} \sum_{n=6,8} \\ W_A^r(CN_{cov}^A, CN_{cov}^{A,r}) W_B^s(CN_{cov}^B, CN_{cov}^{B,s}) \xi_B^s(q_B, q_{B,s}) \times \\ s_n \frac{C_n^{AB,ref}}{R_{AB}^n} f_n^{damp,BJ}(R_{AB}) \end{aligned} \tag{9.38}$$

The dispersion coefficient for two reference atoms $C_n^{AB,\text{ref}}$ is evaluated at the reference points, i.e., for $q_A = q_r$, $q_B = q_s$, $CN^A_{\text{cov}} = CN^r_{\text{cov}}$, and $CN^B_{\text{cov}} = CN^s_{\text{cov}}$.

The Gaussian weighting for each reference system is given by:

$$W^r_A(CN^A_{cov}, CN^{A,r}_{cov}) = \sum_{j=1}^{N_{gauss}} \frac{1}{\mathcal{N}} \exp\left[-6j \cdot (CN^A_{cov} - CN^{A,r}_{cov})^2\right] \tag{9.39}$$

with

$$\sum_{r}^{N_{A,ref}} W^r_A(CN^A_{cov}, CN^{A,r}_{cov}) = 1 \tag{9.40}$$

$\mathcal{N}$ is a normalization constant.

$$\mathcal{N} = \sum_{A,\text{ref}=1}^{N^{A,\text{ref}}} \exp\left[-6j \cdot (CN^A - CN^{A,\text{ref}})^2\right] \tag{9.41}$$

// Write r or ref? CN with or without cov?

The number of Gaussian function per reference system N_{gauss} is mostly one, but equal to three for $CN^{A,r}_{cov} = 0$ and reference systems with similar coordination number.

C_6^{AB} is the pairwise dipole-dipole dispersion coefficients calculated by numerical integration via the Casimir-Polder relation.

$$C_6^{AB} = \frac{3}{\pi} \sum_j w_j \overline{\alpha}_A(i\omega_j, q_A, CN^A_{cov}) \overline{\alpha}_B(i\omega_j, q_B, CN^B_{cov}) \tag{9.42}$$

w_j are the integration weights, which are derived from a trapeziodal partitioning between the grid points $j \in \{2, \ldots, 22\}$.

The isotropically averaged, dynamic dipole-dipole polarizabilites $\overline{\alpha}$ at the jth imaginary frequency $i\omega_j$ are obtained from the self-consistent D4 model; i.e., they are depending on the covalent coordination number and are also charge dependent.

$$\overline{\alpha}_A(i\omega_j, q_A, CN^A_{cov}) = \sum_r^{N_{A,ref}} \xi^r_A(q_A, q_{A,r}) \overline{\alpha}_{A,r}(i\omega_j, q_{A,r}, CN^{A,r}_{cov}) W^r_A(CN^A_{cov}, CN^{A,r}_{cov}) \tag{9.43}$$

$$\overline{\alpha}_{A,r}(i\omega_j, q_{A,r}, CN^{A,r}_{cov}) = \sum_{A,\text{ref}=1}^{N^{A,\text{ref}}} \alpha^{A,\text{ref}}(i\omega, q_A) W^r_A \tag{9.44}$$

The charge-dependent atomic dynamic polarizability for a single reference system of atom A is given by the product of $\alpha^{A,\text{ref}}(i\omega)$ and its scaling function as:

$$\alpha^{A,\text{ref}}(i\omega, q_A) = \alpha^{A,\text{ref}}(i\omega) \xi^r_A(q_A, q_{A,r}) \tag{9.45}$$

$$\alpha^{A,\text{ref}}(i\omega) = \frac{1}{m}\left[\alpha^{AmXn}(i\omega) - \frac{n}{l}\alpha^{X_l}(i\omega)\xi^r_A(q_X, q_{X,r})\right] \tag{9.46}$$

// The effective nuclear charges $z^{X,\text{ref}}$ entering equation 9.46 are constant values determined once for the respective reference system. (Find out how to get them)

The charge-dependency is included via the empirical scaling function ξ^r_A.

$$\xi^r_A(q_A, q_{A,r}) = \exp\left[3\left\{1 - \exp\left[4\eta_A\left(1 - \frac{Z^{eff}_A + q_{A,r}}{Z^{eff}_A + q_A}\right)\right]\right\}\right] \tag{9.47}$$

where η_A is the chemical hardness taken from ref 98.

Z^{eff}_A is the effective nuclear charge of atom A.

C^{AB}_8 is calculated recursively from the lowest order C^{AB}_6 coefficients.

$$C^{AB}_8 = 3C^{AB}_6\sqrt{\mathcal{Q}^A\mathcal{Q}^B} \tag{9.48}$$

$$\mathcal{Q}^A = s_{42}\sqrt{Z^A}\frac{\langle r^4\rangle^A}{\langle r^2\rangle^A} \tag{9.49}$$

$\sqrt{Z^A}$ is the ad hoc nuclear charge dependent factor.

From the original xTB program we can see that s_{42} is 0.5, and Z^A is the atomic number of A.

$$\sqrt{0.5\left(\frac{r^4}{r^2}\sqrt{Z^A}\right)} \tag{9.50}$$

$\langle r^4\rangle$ and $\langle r^2\rangle$ are simple multipole-type expectation values derived from atomic densities which are averaged geometrically to get the pair coefficients.

CN^A_{cov} is the covalent coordination number for atom A.

q is the atomic charge, so q_A is the atomic charge for atom A.

The scaling parameters in the dispersion model are:

$$a1 = 0.52 \quad | \quad a2 = 5.0 \quad | \quad s6 = 1.0 \quad | \quad s8 = 2.7$$

BJ = Becke-Johnson

$$f_n^{damp,BJ}(R_{AB}) = \frac{R_{AB}^n}{R_{AB}^n + (a_1 \times R_{AB}^{crit} + a_2)^6} \tag{9.51}$$

$$R_{AB}^{crit} = \sqrt{\frac{C_8^{AB}}{C_6^{AB}}} \tag{9.52}$$

$$f_9^{damp,zero}(R_{AB},R_{AC},R_{BC}) = \left(1+6\left(\sqrt{\frac{R_{AB}^{crit}R_{BC}^{crit}R_{CA}^{crit}}{R_{AB}R_{BC}R_{CA}}}\right)^{16}\right)^{-1} \tag{9.53}$$

9.2 Total Energy for GFN2-xTB

$$\begin{aligned} E_{GFN2-xTB} &= E_{rep}^{(0)} + E_{disp}^{(0,1,2)} + E_{EHT}^{(1)} + E_{IES+IXC}^{(2)} + E_{AES+AXC}^{(2)} + E_{IES+IXC}^{(3)} \\ &= E_{rep} + E_{disp}^{D4'} + E_{EHT} + E_{\gamma} + E_{AES} + E_{AXC} + E_{\Gamma}^{GFN2} \end{aligned} \tag{9.54}$$

9.2.1 Repulsion Energy

$$E_{rep} = \frac{1}{2}\sum_{A,B}\frac{Z_A^{eff}Z_B^{eff}}{R_{AB}}e^{-\sqrt{a_A a_B}(R_{AB})^{(k_f)}} \tag{9.55}$$

$$k_f = \begin{cases} 1 & if A,B \in \{\mathrm{H},\mathrm{He}\} \\ \frac{3}{2} & otherwise \end{cases} \tag{9.56}$$

Z^{eff} and a are variables fitted for each element. A,B are the labels of atoms. Since we only have C and H in our systems we can simplify this quite a bit in code. R_{AB} is the distance between the A and B atoms.

9.2.2 Extended Hückel Theory Energy

$$E_{EHT} = \sum_{\mu\nu} P_{\mu\nu}H_{\mu\nu}^{EHT} \tag{9.57}$$

$$P_{\mu\nu} = P_{\mu\nu}^0 + \delta P_{\mu\nu} \tag{9.58}$$

$$P^0 = \sum_A P_A^0 \tag{9.59}$$

$\delta P_{\mu\nu} = ??$ comes from the iteration, can be skipped for now (9.60)

Where P_A^0 is the neutral atomic reference density of A. This is known as Superposition of Atomic Densities or SAD.

9.2.3 Isotropic electrostatic and Exchange-correlation energy

Second order

$$E_\gamma = \frac{1}{2}\sum_{A,B}^{N_{atoms}}\sum_{l\in A}\sum_{l'\in B} q_{A,l}q_{B,l'}\gamma_{AB,ll'} \tag{9.61}$$

Third order

$$E_\Gamma^{GFN2} = \frac{1}{3}\sum_{A}^{N_{atoms}}\sum_{l\in A}(q_{A,l})^3\Gamma_{A,l} \tag{9.62}$$

9.2.4 Anisotropic electrostatic energy

$$\begin{aligned} E_{AES} &= E_{q\mu} + E_{q\Theta} + E_{\mu\mu} \\ &= \frac{1}{2}\sum_{A,B}\{f_3(R_{AB})[q_A(\boldsymbol{\mu}_B^T\boldsymbol{R}_{BA}) + q_B(\boldsymbol{\mu}_A^T\boldsymbol{R}_{AB})] \\ &\quad + f_5(R_{AB})[q_A\boldsymbol{R}_{AB}^T\boldsymbol{\Theta}_B\boldsymbol{R}_{AB} + q_B\boldsymbol{R}_{AB}^T\boldsymbol{\Theta}_A\boldsymbol{R}_{AB} \\ &\quad - 3(\boldsymbol{\mu}_A^T\boldsymbol{R}_{AB})(\boldsymbol{\mu}_B^T\boldsymbol{R}_{AB}) + (\boldsymbol{\mu}_A^T\boldsymbol{\mu}_B)R_{AB}^2]\} \end{aligned} \tag{9.63}$$

9.2.5 Anisotropic XC energy

$$E_{AXC} = \sum_A (f_{XC}^{\mu_A}|\boldsymbol{\mu}_A|^2 + f_{XC}^{\Theta_A}||\boldsymbol{\Theta}_A||^2) \tag{9.64}$$

What norms are these?

9.2.6 Dispersion Energy

$$
\begin{aligned}
E_{disp}^{D4'} = &- \sum_{A>B} \sum_{n=6,8} s_n \frac{C_n^{AB}(q_A, CN_{cov}^A, q_B, CN_{cov}^B)}{R_{AB}^n} f_{damp,BJ}^{(n)}(R_{AB}) \\
&- s_9 \sum_{A>B>C} \frac{(3cos(\theta_{ABC})cos(\theta_{BCA})cos(\theta_{CAB})+1)C_9^{ABC}(CN_{cov}^A, CN_{cov}^B, CN_{cov}^C)}{(R_{AB}R_{AC}R_{BC})^3} \\
&\times f_{damp,zero}^{(9)}(R_{AB}, R_{AC}, R_{BC}).
\end{aligned}
\tag{9.65}
$$

The term in the second line is the three-body Axilrod– Teller–Muto (ATM) (What is this??????) term and the last line is the corresponding zero-damping function for this term.

The damping and scaling parameters in the dispersion model are:

$$
s6 = 1.0 \quad | \quad s8 = 2.7 \quad | \quad s9 = 5.0
$$

C_9^{ABC} is the triple-dipole constant[1]:

$$
C_9^{ABC} = \frac{3}{\pi} \int_0^\infty \alpha^A(i\omega)\alpha^B(i\omega)\alpha^C(i\omega)d\omega \tag{9.66}
$$

The three-body contribution is typically $< 5 - 10\%$ of E_{disp}, so it is small enough that we can reasonably approximate the coefficients by a geometric mean as[1]:

$$
C_9^{ABC} \approx -\sqrt{C_6^{AB} C_6^{AC} C_6^{BC}} \tag{9.67}
$$

θ_{ABC} is the angle between the two edges going from B to the other two atoms. θ_{BCA} is the angle between the edges going from C to the other two and so on.

[1] https://www.researchgate.net/publication/43347348_A_Consistent_and_Accurate_Ab_Initio_Parametrization_of_Density_Func D_for_the_94_Elements_H-Pu

9.2.7 SAD - Superposition of Atomic Densities

The superposition of atomic densities(SAD) is an approach to obtain a good approximation of a collection of atoms, to be used as an initial guess for solving the self-consistent field(SCF) equation.

As originally implemented in DISCO, the molecular electron density can be obtained by adding the densities of all the constituting atoms.

This is how we get the density matrix for an isolated atom? equation 15 from: (https://sci-hub.box/10.1002/jcc.540030314)

$$D_{ij} = \sum_{a}^{occ} c_{ia} c_{ja} \tag{9.68}$$

To get the coefficients we need to solve SCF for each atom? this is supposedly cheap, but idk how to do it. (https://sci-hub.box/10.1002/jcc.20393) Though the math for Direct SCF Approach is given in this paper at equation 10: (https://sci-hub.box/10.1002/jcc.540030314). This is probably how.

The SAD method is then the sum of all of these?

Equation 2 in the GFN2 paper talks about "superposition of (neutral) atomic reference densities". Is this relevant?

Direct SCF Approach

$$\begin{aligned}
\Delta F_{ab} =& (c_{ia}c_{jb} + c_{ja}c_{ib}) \\
& \Delta F_{ij} + (c_{ia}c_{kb} + c_{ka}c_{ib}) \\
& \Delta F_{ik} + (c_{ia}c_{lb} + c_{la}c_{ib}) \\
& \Delta F_{il} + (c_{ja}c_{kb} + c_{ka}c_{jb}) \\
& \Delta F_{jk} + (c_{ja}c_{lb} + c_{la}c_{jb}) \\
& \Delta F_{jl} + (c_{ka}c_{lb} + c_{la}c_{kb}) \Delta F_{kl} \\
& = l_{ijkl}(4E^{ab}_{ij}D_{kl} + 4D_{ij}E^{ab}_{kl} - E^{ab}_{ik}D_{jl} - D_{ik}E^{ab}_{jl} - E^{ab}_{il}D_{jk} - D_{il}E^{ab}_{jk})
\end{aligned} \tag{9.69}$$

where

$$E^{ab}_{ij} = c_{ia}c_{jb} + c_{ja}c_{ib} \tag{9.70}$$

Equation 18 from (https://sci-hub.box/https://doi.org/10.1021/acs.chemrev.5b00584) uses ρ_0 which is the superposition of neutral atom densities:

$$\rho_0 = \sum_A \rho_0^A \tag{9.71}$$

10 Conclusion

We have explained why massive lockstep parallelization on a GPU is a good fit for the problem domain, and what makes it better than regular task parallelization. We have explained necessary considerations to make this approach possible and effective.

AI Declaration

11

look at what needs to be in the AI section somewhere on KU's website.

Part II

Appendicies

An appendix

A

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

Bibliography

[1] R. Hormuth. *Leadership HPC Performance with 5th Generation AMD EPYC Processors*. 2025. URL: https://www.amd.com/en/blogs/2025/leadership-hpc-performance-with-5th-generation-amd.html (visited on Aug. 11, 2025).

[2] N. Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. V1.0. NVIDIA.

[3] N. A. S. (Division. *Basics on NVIDIA GPU Hardware Architecture*. 2025. URL: https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html (visited on Aug. 10, 2025).

[4] C. Bannwarth, S. Ehlert, and S. Grimme. "GFN2-xTB—An Accurate and Broadly Parametrized Self-Consistent Tight-Binding Quantum Chemical Method with Multipole Electrostatics and Density-Dependent Dispersion Contributions". In: *Journal of Chemical Theory and Computation* 15.3 (Mar. 2019), pp. 1652–1671. ISSN: 1549-9618. DOI: 10.1021/acs.jctc.8b01176.

[5] T. G. Draper. *Addition on a Quantum Computer*. 2000. arXiv: quant-ph/0008033 [quant-ph].

[6] L. Ruiz-Perez and J. C. Garcia-Escartin. "Quantum arithmetic with the quantum Fourier transform". In: *Quantum Information Processing* 16.6 (Apr. 2017). ISSN: 1573-1332. DOI: 10.1007/s11128-017-1603-1.

[7] S. Wang, Z. Wang, G. Cui, L. Fan, S. Shi, R. Shang, W. Li, Z. Wei, and Y. Gu. *Quantum Amplitude Arithmetic*. 2020. arXiv: 2012.11056 [quant-ph].

[8] C. Bannwarth, E. Caldeweyher, S. Ehlert, A. Hansen, P. Pracht, J. Seibert, S. Spicher, and S. Grimme. "Extended tight-binding quantum chemistry methods". In: *WIREs Computational Molecular Science* 11.2 (2021), e1493. DOI: https://doi.org/10.1002/wcms.1493. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1493.

[9] T. Häner, M. Roetteler, and K. M. Svore. *Optimizing Quantum Circuits for Arithmetic*. 2018. arXiv: 1805.12445 [quant-ph].

[10] S. Grimme, C. Bannwarth, and P. Shushkov. “A Robust and Accurate Tight-Binding Quantum Chemical Method for Structures, Vibrational Frequencies, and Noncovalent Interactions of Large Molecular Systems Parametrized for All spd-Block Elements (Z = 1–86)”. In: *Journal of Chemical Theory and Computation* 13.5 (2017). PMID: 28418654, pp. 1989–2009. DOI: 10.1021/acs.jctc.7b00118. eprint: https://doi.org/10.1021/acs.jctc.7b00118.

[11] A. Gilyén, Y. Su, G. H. Low, and N. Wiebe. “Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics”. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 193–204. ISBN: 9781450367059. DOI: 10.1145/3313276.3316366.

[12] NVIDIA Corporation. *Ada Tuning Guide*. Release 12.9. NVIDIA. 2025.

[13] N. Corporation. *NVIDIA ADA GPU ARCHITECTURE*. V2.02. NVIDIA.

Part III

Articles