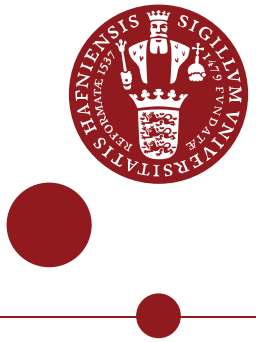
UNIVERSITY OF COPENHAGEN



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

subtitle

Masters

Anton M. Nielsen & Asmus Tørsleff

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

Department of Computer Science
Department of 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 <James title> 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
Department of Quantum Information Science
Universitetsparken 5
2100 Copenhagen Ø

Acknowledgements

I would like thank my supervisor Professor, Ph.D., Dr. Scient. Kurt V. Mikkelsen for guidance and advise during my masters degree.

I would like to thank Kurt's research group for their discussions and advice.

I would like to thank my friends and family for supporting me and reminding me of a world outside of chemistry.

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.

Contributions to papers

This masters thesis is partly based on the following paper which is attached in the articles appendix.

In the paper "**??**", Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Table of Contents

I Thesis **1**

0.1 Introduction . . . 1

1 Theory **3**

1.1 GFN2-xTB . . . 3

1.2 High Performance Parallel Computing . . . 8

1.2.1 Memory Types and Current Hardware . . . 10

1.2.2 Optimizing GPU Configuration for Isolated Molecules . . . 12

1.3 Introduction to quantum algorithmic approaches . . . 13

1.3.1 Calculating E^{Γ} using Quantum Digital Arithmetic . . . 15

1.3.2 Sampling using Quantum Amplitude Arithmetic . . . 17

1.3.3 Calculating E^{Γ} with Quantum Amplitude Arithmetic. . . . 18

1.3.4 Calculating E^{γ} using Quantum Digital Arithmetic . . . 26

1.3.5 Complexity . . . 27

1.3.6 Cleaning up ω . . . 28

1.3.7 Concentrating the probabilities on the best candidates . . . 28

1.3.8 Discussion . . . 29

2 Related Work **30**

2.1 xTB version 6.4.0 . . . 30

2.2 dxTB . . . 30

3 Methodology **31**

3.1 Porting the Reference Implementation . . . 31

3.2 Testing . . . 32

3.2.1 Towards Reproducability with Nix . . . 32

3.2.2 Patching xTB . . . 37

3.2.3 Implementing the Tests . . . 38

4 Code Structure **40**

5 Challenges **42**

5.1 Implementation Deviating from Paper . . . 42

6 Results **43**
6.1 Validation . . . 43
6.2 Benchmarks . . . 43

7 Reflections **44**

8 Future Work **45**

9 Extended Hückel Theory Matrix for GFN2-xTB **46**
9.1 Fock Matrix for GFN2-xTB . . . 47
9.1.1 Isotropic Electrostatic and Exchange-correlation contribution . . . 47
9.1.2 Anisotropic Electrostatic and Exchange-correlation contribution . . . 48
9.1.3 Dispersion contribution . . . 50
9.2 Total Energy for GFN2-xTB . . . 54
9.2.1 Repulsion Energy . . . 54
9.2.2 Extended Hückel Theory Energy . . . 54
9.2.3 Isotropic electrostatic and Exchange-correlation energy . . . 55
9.2.4 Anisotropic electrostatic energy . . . 55
9.2.5 Anisotropic XC energy . . . 55
9.2.6 Dispersion Energy . . . 56
9.2.7 SAD - Superposition of Atomic Densities . . . 57

10 AI Declaration **59**

II Appendicies **60**

A An appendix **61**

Bibliography **62**

III Articles **63**

Part I

Thesis

0.1 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.

1 Theory

1.1 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 but also does well on a variety other properties. Over all 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 the method is still treating the calculations at the level of individual shells in many places. GFN2-xTB uses A Self-Consistent Charges approach i.e. it makes an initial guess at a density matrix and an energy which it then iteratively refines until both have converged.

```python
def get_GFN2_energy(atoms: list[int], positions : list[list[float]]) -> float:
    density = P(atoms)
    overlap = S(atoms, positions)
    dipol_dipol = D(atoms, positions, overlap)
    charge_quadrupol = Q(atoms, positions, overlap, dipol_dipol)
    initial_hamiltonian = H0(atoms, positions)
    huckel_theory_matrix = H_EHT(atoms, positions, overlap, initial_hamiltonian)
    charges = mulliken_population_analysis(density,atoms)
    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 = E_repulsion + E_dispersion + E_huckel + E_anisotropic + E_isotropic
    # ...
    while not (energy_converged and densities_converged):
        eigen_values = diagonalize(hamiltonian, overlap) # HC=SCe
        new_density = compute_density_matrix_from_fermi(eigen_values,...)
        # ... update everything with new density
        energy_converged = (E-E_new)**2 < tolerance
        densities_converged = error_squared(density,new_density) < tolerance
        E = E_new
        density = new_density
```

Figure 1.1.: Python like code illustrating the main loop of a GFN2-xTB implementation. Line 2-14 computes the energy in a non SCC manner, line 15-23 iteratively improves the energy using SCC.

Let us define a small helper function to decrease the amount of indentation in the later code examples.

```python
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 1.2.: Python like code for generating a convenient list of orbitals to iterate though.

And a helper function to create a square matrix.

```
def get_square_matrix(n: int) -> list[list[float]]:
    matrix = []
    for _ in range(n):
        row = []
        for _ in range(n):
            row.append(0.0)
        matrix.append(row)
    return matrix
```

Figure 1.3.: Python like code for generating a square matrix.

The initial density matrix guess is formulated as a superposition of neutral atomic reference densities $P_0 = \sum_A P_{A_0}$. 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. The shells are filled from the nucleus and outwards.

```
def density_initial_guess(atoms: list[int]) -> list[list[float]]:
    orbitals = get_orbitals(atoms)
    occs = get_square_matrix(len(orbitals))
    for idx,(_,atom,subshell,orbital) 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 1.4.: Python like code illustrating computation of 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]. Thus occ will be a repeating series of $[\frac{1.0}{1}, \frac{3.0}{3}, \frac{3.0}{3}, \frac{3.0}{3}]$. We also need the overlap matrix, S, which is computed in the following way.

$$S_{\nu\mu} = \langle \nu | \mu \rangle \qquad \forall \nu \in l \in A, \mu \in l' \in B \tag{1.1}$$

```python
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
def compute_integral(atom_A:int, atom_B:int, shell_A:int, shell_B:int,
↪   orbital_A:int, orbital_B:int)-> float:
    #...
```

Figure 1.5.: Python like code illustrating construction of S

NOTE(Asmus) Explain D and Q as well

The huckel matrix is calculated using extended huckel theory. This is the calculation.

$$H_{\nu\nu} = H_A^l - H_{CN_A} CN'_A \qquad \forall \nu \in l \in A \tag{1.2}$$

$$CN'_A = \sum_{B \neq A} \left(1 + e^{-10\left(\frac{4\left(R_{A,cov}+R_{B,cov}\right)}{3R_{AB}}-1\right)}\right)^{-1} \left(1 + e^{-20\left(\frac{4\left(R_{A,cov}+R_{B,cov}+2\right)}{3R_{AB}}-1\right)}\right)^{-1} \tag{1.3}$$

```
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 1.6.: Python like code illustrating construction of H_{KK}

1.2 High Performance Parallel Computing

The xTB program uses a specified method of the GFN-xTB algorithm to compute various energies for a molecule. By default the program is set to use GFN2-xTB, which is also the method this project focuses on.

For a molecule on the smaller scale such as a caffeine molecule which has 24 atoms, the Fortran xTB program takes around 100ms on a 12th gen intel mobile processor, while a fullerene with 200 carbon atoms takes about 23 seconds to compute. The computional time for even small molecules begins to be noticable, when the problem size grows to thousands or millions of molecules.

This project aims to compute the energies of fullerenes in the isomerspaces $C_{20}, \cdots, C_{200}$. For isomerspaces of this scale, rather than improving the performance for individual molecules, what is truly interesting for this problem domain is speeding up largely concurrent xTB computations by running them in parallel on general purpose graphic processing units (GPGPUs).

The highest level of parallelization here is to compute all the energy terms of a molecule in the same kernel. There are no data dependencies between the computations of multiple molecules, and this makes it a perfect case for massive parallelization by distributing these isolated workloads across the thousands of threads supported on modern GPGPUs. Within the area of computing, the idea of running the same operations in parallel is known as a lockstep system. With a focus on fullerenes, which consists exclusively of carbon atoms, this type of lockstep parallelization is exactly what we want to create, namely a fast and constant flow of data for the broader pipeline that this project is part of.

Since the executions of the xTB algorithm on each fullerene are completely isolated workloads, this means that the level of parallelization for a given isomer group, such as C20, scales with the amount of isomers in that group. This means that a much larger isomer group like C200 will also have a much greater level of parallelization.

The streaming multiprocessors (SMs) on a GPU are slower and simpler than the cores on a CPU. They have no branch prediction or other smart optimization techniques, but instead an SM has more threads it can execute in parallel in comparison to a CPU core which can only execute threads concurrently. The difference is that SMs can truly run its threads simultaniously, while CPU cores rely on context switching to make it seem like proccesses are running simultaniously. With SMs, working only on a few fullerenes will have a massive overhead from spinning up a kernel and

copying data from the host(CPU) to the device(GPU), but the problem size for this project makes SMs a great fit.

The current Fortran implementation of the xTB program only takes a single molecule at a time, but when doing lockstep parallization it would be interesting to have an implementation that takes multiple molecules. This would avoid the overhead of starting multiple processes, and the program will have the data for all molecules, which gives opportunity for data coalescing by aligning the data as a structure of arrays (SOA) instead of an array of structures (AOS). It can also make copying data from the host to device more efficient since the data for multiple molecules can be moved together by the same instruction. To allow for coalesced access to the data on the device, we can essentially realign the data so that parts of the molecules that are accessed together, are also close together in memory.

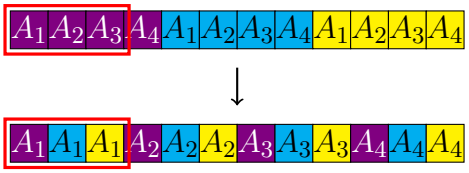


Figure 1.7.: Transforms arrays of structures where the memory for each structure is laid out contiguously, into a structure of arrays where data relevant for a computation is close together. This way of grouping related data close together is known as spatial locality and allows for coalesced access of the data.

An example of turning an AOS into an SOA can be seen in Figure 1.7 where the data is realigned to allow for coalesced data access. If the code accesses only a few atoms of a molecule at a time, then it is a waste if the memory page contains the rest of a molecule as well. Instead of copying whole molecules at a time, the data should be structured such that only the required data of a molecule is copied as it allows the kernel to get that data for more molecules at a time, thus reducing unnecessary copies and speeding up the computation.

The geometric data for the fullerenes that are fed into the xTB implementation that our project focuses on, is already on the device and is never meant to leave the device before the final result is reached. The xTB implementation should therefore never move any data between the host and device. As such, memory coalescing is something to consider mainly when talking about the data transfer between the global and shared memory on a device. When we compute the energies for multiple molecules in the same SM, then we would like to move the necessary data for the current operation for all of the molecules in the SM together. The whole molecule will not fit in the lower memory levels, so we need to interweave the data of the molecules such that the necessary parts of the molecules can be moved together in as few operations as possible.

1.2.1 Memory Types and Current Hardware

The types of memory typically found in the memory hiearchy on a GPU are global, shared, local, and register memory. Using these levels of memory properly is crucial for achieving high performance in parallel computing tasks. Here is an overview of the various levels:

- Global - Accessible from all threads on the GPU. This is the largest but also the slowest pool of memory.
- Shared - Tied to a thread block (or workgroup), so it can be accessed by the threads in the same thread block. This pool of memory is smaller but faster 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.
- Local Memory - Also private to each thread. This type is slower than register memory and is usually used when there is insufficient space for variables on the registers of a thread.

The NVIDIA Ada Lovelace GPU architecture[1] has 80 GB of global memory, 100 KB of shared memory per SM, and $64,000$ 32-bit registers per SM, which gives us 256 KB of register memory per SM.

$$\begin{aligned} 64000 \cdot 32 &= 2048000 \text{ bits} \\ &= 256000 \text{ bytes} \\ &= 256 \text{ KB} \end{aligned} \tag{1.4}$$

Each thread has a maximum of 255 registers, and each SM has a maximum of 24 blocks and a maximum of 48 warps. This adds up such that full utilization can be achieved when each block has 2 warps allocated. A warp has 32 threads, so an SM has a total of 1536 threads when each of its 24 blocks has two warps. Distributing the $64,000$ registers evenly over these threads gives each of them 41 registers, which is considerably lower than the maximum of 255, but allows all of the threads to be used evenly. By using Equation 1.8 we can see that this configuration utilizes 251.9 KB of the register memory available to each of the SMs.

$$WarpsPerBlock = \left\lfloor \frac{WarpsPerSM}{BlocksPerSM} \right\rfloor \tag{1.5}$$

$$ThreadsPerBlock = WarpsPerBlock \times ThreadsPerWarp \tag{1.6}$$

$$RegistersPerThread = \left\lfloor \frac{RegistersPerSM}{BlocksPerSM \times ThreadsPerBlock} \right\rfloor \tag{1.7}$$

$$\begin{aligned} RegisterMemoryUtilizedPerSM = \\ & BlocksPerSM \\ & \times ThreadsPerBlock \\ & \times RegistersPerThread \\ & \times BitsPerRegister \end{aligned} \tag{1.8}$$

BlocksPerSM and BitsPerRegister are constants found in the specification for the GPU architecture.

With this, each block in an SM has 10.496 KB of register memory to work with, and thus each thread has 164 bytes.

$$\frac{251.904}{24} = 10.496 \text{ KB} \tag{1.9}$$

$$\frac{251904}{1536} = 164 \text{ bytes} \tag{1.10}$$

The shared memory capacity per SM is 100 KB and a single block can have a maximum of 99 KB. This gives each block about 4.16 KB of shared memory, meaning that the 64 threads in a block will have 65 bytes each.

$$\frac{100}{24} = 4.166 \text{ KB} \tag{1.11}$$

$$\frac{4166}{64} = 65.09 \text{ bytes} \tag{1.12}$$

The Ada Lovelace architecture has a total of 128 SMs and it also has a 98304 KB L2 cache. This results in a total of 3072 blocks that each has 32 KB of the L2 Cache. All of this combined gives each block a total of 46.656 KB of non-global memory.

$$10.496 \text{ KB} + 4.166 \text{ KB} + 32 \text{ KB} = 46.662 \text{ KB} \tag{1.13}$$

Distributing the global memory across the blocks gives each of them 26.042 MB. Combined with the non-global memory, this results in each block having a total of 26.088 MB of memory.

1.2.2 Optimizing GPU Configuration for Isolated Molecules

Computing xTB-GFN2 for a molecule is completely isolated, so if we do the whole xTB computation in lockstep, then we only have to concern ourselves with how many warps and blocks a single molecule needs in order to keep all its data in device memory. When we know these details, then we can simply scale up the number of molecules to be computed in lockstep until all the resources on the GPU are saturated. With this approach, expanding to multiple GPUs should be rather trivial as there are no interdependencies.

The exact space needed to compute the GFN2 method of xTB for a single fullerene is not yet clear to us, and it will also vary based on the size of the fullerene. To make finding the most optimal

GPU configuration easier when optimizing for most possible parallel xTB computations, we have developed a script that computes such a configuration based on the space requirement of a single molecule.

```
def compute(bytes_per_molecule):
  mb_per_molecule = bytes_per_molecule / 1_000_000

  threads_per_molecule = compute_threads_per_molecule(mb_per_molecule)
  warps_per_molecule = compute_warps_per_molecule(threads_per_molecule)
  molecules_per_block = compute_molecules_per_block(warps_per_molecule)

  print_utilization(warps_per_molecule, molecules_per_block)

  print(f"MB per molecule: {mb_per_molecule} MB")
  print(f"Threads per molecule: {threads_per_molecule}")
  print(f"Warps per molecule: {warps_per_molecule}")
  print(f"Molecules per block: {molecules_per_block}")
```

Figure 1.8.: This is the top-level function for computing the optimal number of warps needed per molecule when optimizing for most possible parallel xTB computations on a single GPU specifically for the Ada Lovelace architecture.

The output of the script has information on how many threads are required for a single molecule, how many warps this fits within. It also includes general information about the utilization of the GPUs resources with the configuration presented. An example of the full output can be seen in Figure 1.9.

In Figure 1.9 each molecule is 8.3 MB and thus requires the memory of 18 threads. Each warp has 32 threads, so in this case a single molecule fits within a single warp. With 24 blocks and 48 warps per SM, all blocks can have 2 warps each, which in this scenario means that 2 molecules can be computed in parallel within a single block. This space requirement for a molecule is just an example and in no way guaranteed to be representative of the actual requirements of fullerenes of any size.

1.3 Introduction to quantum algorithmic approaches

In this section we will attempt to construct quantum algorithms for calculating three of the GFN2-xTB[2] energy terms: E^{Γ}, E^{γ} and E^{EHT}. We will showcase three different approaches to doing a calculation as a building block of a larger circuit.

```
| Global memory | L2 cache | Total SMs | Threads per warp |
|    80.0 GB    | 98304 KB |    128    |        32        |
| Threads Available | Threads Used | Blocks Available | Blocks Used |
|      196608       |    196608    |       3072       |    3072.0   |

################################### Per SM ###################################
| Warps | Blocks | Registers | Shared Memory | Threads Available | Threads Used |
|  48   |   24   |   64000   |    100 KB     |       1536        |     1536     |
| Blocks Available | Blocks Used |
|        24        |     24.0    |

############################ Per Block ############################
| L2 Cache | Register Mem | Shared Mem | Global Mem | Total mem |
| 32.0 KB  |  10.496 KB   |  4.167 KB  |  26.042 MB | 26.088 MB |

MB per molecule: 8.3 MB
Threads per molecule: 18
Warps per molecule: 1
Molecules per block: 2
```

Figure 1.9.: This is the output of our script for computing an optimal GPU configuration when a single molecule needs 8.3 MB of memory.

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[3, 4]. This approach will be applied to 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[5]. 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. Say we know all the energies, and we want to do something with those states that have a low energy. 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 a known high energy of say 100.9 we

get much larger relative differences where the low energy isomers will have a much larger α than the high energy isomers.

The final algorithmic approach we will explore uses quantum singular value transformations[6]. In this approach the calculations are being carried out in the singular values of block encoded matrices. We will use this approach to calculate the E^{ETH} term, as it involves a lot of elementwise matrix multiplications. This is well suited for this approach.

For all of these approaches we will assume that we have access to some pretty intricately prepared states. We will not go into how the are prepared other than the fact that classically we can generate the geometries and so for entire isomer spaces without any other information. As any classical computation in theory also can be applied to a quantum computer after modifications it is a possibility to prepare these states.

1.3.1 Calculating E^{Γ} using Quantum Digital Arithmetic

The GFN2-xTB E^{Γ} term has the following form[7]

$$E^{\Gamma} = \frac{1}{3}\sum_{A}\sum_{\mu\in A}(q_{A,\mu})^3\Gamma_{A,\mu}, \tag{1.14}$$

where $q_{A,\nu} = \sum_B \sum_{\nu\in B} P_{\mu\nu}S_{\mu\nu}$ is the 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 $\Gamma_{0(1)}$.

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

$$\sum_{\mu\in A}(q_{A,\mu})^3\Gamma_{A,\mu} = \sum_{\mu\in\{0,1\}}(q_{A,\mu})^3\Gamma_{\mu} \tag{1.15}$$

We now need a unitary which computes this function on a given state $|\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |acc\rangle_E \rightarrow |\Gamma_\mu\rangle_\Gamma |q_{A,\mu}\rangle_Q |acc + (q_{A,\mu})^3\Gamma_\mu\rangle_E$. The subscripts on the kets refer to the quantum register they represent. Consider having access to the following fused multiply add unitary $|A\rangle|B\rangle|acc\rangle \rightarrow$

$|A\rangle|B\rangle|acc + A * B\rangle$. Let us to our initial Γ, Q, E registers add two ancillary registers, W_1, W_2. We can now apply the following unitary

$$E_i^{\Gamma}{}_{(\Gamma,Q,W_1,W_2,E)} = \text{MADD}^{\dagger}_{\Gamma,Q,W_1}\text{MADD}^{\dagger}_{Q,W_1,W_2}\text{MADD}_{Q,W_2,E}\text{MADD}_{Q,W_1,W_2}\text{MADD}_{\Gamma,Q,W_1} \quad (1.16)$$

Let us follow the process:

$$E_i^{\Gamma}{}_{(\Gamma,Q,W_1,W_2,E)}|\Gamma_\mu\rangle_\Gamma|q_{A,\mu}\rangle_Q|0\rangle_{W_1}|0\rangle_{W_2}|acc\rangle_E \quad (1.17)$$

$$=\text{MADD}^{\dagger}_{\Gamma,Q,W_1}\text{MADD}^{\dagger}_{Q,W_1,W_2}\text{MADD}_{Q,W_2,E}\text{MADD}_{Q,W_1,W_2}|\Gamma_\mu\rangle_\Gamma|q_{A,\mu}\rangle_Q|\Gamma_\mu q_{A,\mu}\rangle_{W_1}|0\rangle_{W_2}|acc\rangle_E \quad (1.18)$$

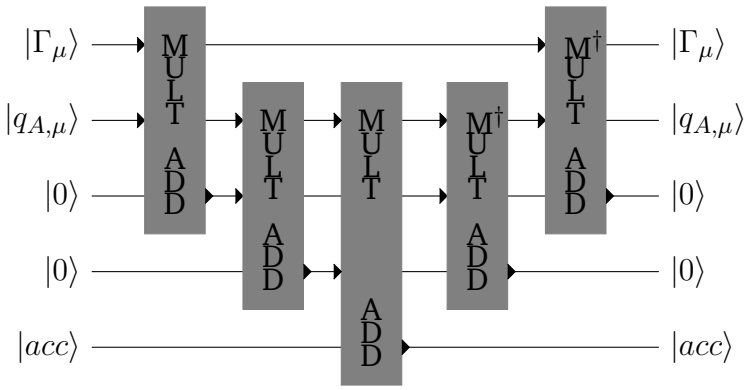
$$=\text{MADD}^{\dagger}_{\Gamma,Q,W_1}\text{MADD}^{\dagger}_{Q,W_1,W_2}\text{MADD}_{Q,W_2,E}|\Gamma_\mu\rangle_\Gamma|q_{A,\mu}\rangle_Q|\Gamma_\mu q_{A,\mu}\rangle_{W_1}|\Gamma_\mu(q_{A,\mu})^2\rangle_{W_2}|acc\rangle_E \quad (1.19)$$

$$=\text{MADD}^{\dagger}_{\Gamma,Q,W_1}\text{MADD}^{\dagger}_{Q,W_1,W_2}|\Gamma_\mu\rangle_\Gamma|q_{A,\mu}\rangle_Q|\Gamma_\mu q_{A,\mu}\rangle_{W_1}|\Gamma_\mu(q_{A,\mu})^2\rangle_{W_2}|acc+\Gamma_\mu(q_{A,\mu})^3\rangle_E \quad (1.20)$$

$$=\text{MADD}^{\dagger}_{\Gamma,Q,W_1}|\Gamma_\mu\rangle_\Gamma|q_{A,\mu}\rangle_Q|\Gamma_\mu q_{A,\mu}\rangle_{W_1}|0\rangle_{W_2}|acc+\Gamma_\mu(q_{A,\mu})^3\rangle_E \quad (1.21)$$

$$=|\Gamma_\mu\rangle_\Gamma|q_{A,\mu}\rangle_Q|0\rangle_{W_1}|0\rangle_{W_2}|acc+\Gamma_\mu(q_{A,\mu})^3\rangle_E \quad (1.22)$$

(1.23)



We see that already in eq. 1.20 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 a 2 ancillary registers for every calculation. The MADD gates here could be implementing using QFT multipliers[4] in which case we wouldn't need any additional ancillaries. If we decompose our QFT multiplier into its components it is essentially a chain of QFT additions[3, 4] 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

$$\begin{aligned}
&\text{DA SDA}\,|ID\rangle_{ID}\,|0\rangle \rightarrow \\
&\quad \text{DA}\,|ID\rangle_{ID}\left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_{A} |q_{A,\mu}\rangle_{Q_{A,\mu}}\right)|0\rangle_{W_1}\,|0\rangle_{W_2}\,|0\rangle_E \rightarrow \\
&\quad\quad |ID\rangle_{ID}\left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_{A} |q_{A,\mu}\rangle_{Q_{A,\mu}}\right)|0\rangle_{W_1}\,|0\rangle_{W_2}\,|E^{\Gamma}\rangle_E
\end{aligned} \tag{1.24}$$

will give us the E^{Γ} energy term in the E register corresponding to the ID in the ID register.

1.3.2 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^{Γ}. We can achieve this using Quantum Amplitude Arithmetic[5], not to be confused with Quantum Amplitude Amplification, both shortened as QAA but in this writing as QA-Arithmetic and QA-Amplification.

Wang et al. use their introduced addition and multiplication primitives to construct a circuit which transforms the state $|x\rangle_D\,|0\rangle_C\,|0\rangle_W \rightarrow \frac{1}{2}\frac{x}{2^n}\,|x\rangle_D\,|0\rangle_C\,|1\rangle_W + \alpha\,|\omega\rangle_{D\otimes C\otimes W}$ 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.

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.

$$\begin{aligned}&\sum_{ID\in\text{isomers}} |ID\rangle_{ID} \left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_{A} |q_{A,\mu}\rangle_{Q_{A,\mu}} \right) |0\rangle_{W_1} |0\rangle_{W_2} |E^\Gamma\rangle_E \rightarrow \\ &\sum_{ID\in\text{isomers}} |ID\rangle_{ID} \left(\bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_\mu} \bigotimes_{A} |q_{A,\mu}\rangle_{Q_{A,\mu}} \right) \left(\frac{1}{2} \frac{E^\Gamma_{ID}}{2^n} |0\rangle_{W_1} |1\rangle_{W_2} |E^\Gamma_{ID}\rangle_E + \alpha_{ID} |\omega_{ID}\rangle \right)\end{aligned} \tag{1.25}$$

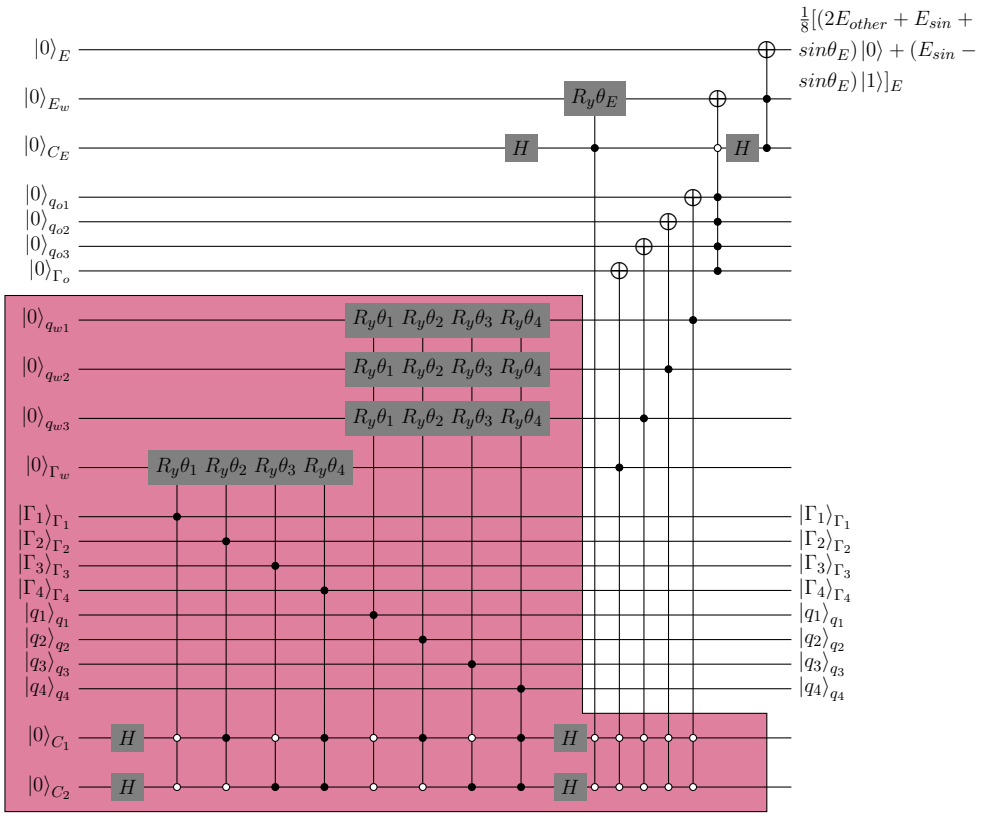
If we now sample from this superposition and postselect for $W_1 = 0$ and $W_2 = 1$ we are more likely to sample a low energy fullerene. The likelihood of sampling a given canonical fullerene ID is proportional with E^Γ for that fullerene.

1.3.3 Calculating E^Γ with Quantum Amplitude Arithmetic.

An alternative strategy would be to go all in on QA-Arithmetic and do all the arithmetic in the amplitudes. Here we would encode a molecule as follows

$$\begin{aligned}&\text{SAA} |ID\rangle_{ID} |0\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_w,\Gamma_{1,\ldots,n},q_{1,\ldots,n},C_{1,\ldots,\lceil log(n+1)\rceil}} \\ &= |ID\rangle_{ID} |0\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_w} \bigotimes_{\mu\in\{0,1\}} |\Gamma_\mu\rangle_{\Gamma_{1,\ldots,n}} \bigotimes_{A} |q_{A,\mu}\rangle_{q_{1,\ldots,n}} |0\rangle_{C_{1,\ldots,\lceil log(n+1)\rceil}}\end{aligned} \tag{1.26}$$

Let us apply the following example circuit to our encoding. Here we focus on one pair of q and Γ.



This circuit is built from the addition and multiplication primitives introduced in the QA-Arithmetic paper[5]. We also do a trivial modification to get subtraction. The diagram is for a n=4 bit example i.e. q and Γ are encoded as 4 bit numbers. The crimson region in the diagram is the only part which needs to be scaled up if using larger n.

If using more than one q and Γ the contribution to the final E_w register should include those as well which would just need 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.

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.

$$CU_t^{c1,c2,c3,\ldots} = (U_t - I_t) \otimes |a\rangle\langle a|_{c1} \otimes |b\rangle\langle b|_{c2} \otimes |c\rangle\langle c|_{c3} \otimes \cdots + \sum_{\alpha,\beta,\zeta,\cdots \in \{0,1\}} I_t \otimes |\alpha\rangle\langle\alpha|_{c1} \otimes |\beta\rangle\langle\beta|_{c2} \otimes |\zeta\rangle\langle\zeta|_{c3} \otimes \ldots. \tag{1.27}$$

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

$$\begin{aligned} H_{C1}H_{C2}\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|00\right\rangle_{C_{(1,2)}} \rightarrow \\ \frac{1}{2}(\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|00\right\rangle_{C_{(1,2)}} + \\ \left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|01\right\rangle_{C_{(1,2)}} + \\ \left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|10\right\rangle_{C_{(1,2)}} + \\ \left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|11\right\rangle_{C_{(1,2)}}) \end{aligned} \tag{1.28}$$

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

$$\begin{aligned} CRy_{\Gamma_w}^{\Gamma_4,C_1,C_2}(2\theta_4)CRy_{\Gamma_w}^{\Gamma_3,\bar{C_1},C_2}(2\theta_3)CRy_{\Gamma_w}^{\Gamma_2,C_1,\bar{C_2}}(2\theta_2)CRy_{\Gamma_w}^{\Gamma_1,\bar{C_1},\bar{C_2}}(2\theta_1) \\ \frac{1}{2}\sum_{x_1,x_2\in\{0,1\}}\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|x_1x_2\right\rangle_{C_{(1,2)}} \rightarrow \\ \frac{1}{2}(CRy_{\Gamma_w}^{\Gamma_1}(2\theta_1)\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|00\right\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_2}(2\theta_2)\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|01\right\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_3}(2\theta_3)\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|10\right\rangle_{C_{(1,2)}} + \\ CRy_{\Gamma_w}^{\Gamma_4}(2\theta_4)\left|0000\right\rangle_{q_{w(1,2,3)},\Gamma_w}\left|11\right\rangle_{C_{(1,2)}}) \rightarrow \\ \frac{1}{2}(\left|000\right\rangle\left[\Gamma1(cos\theta_1\left|0\right\rangle + sin\theta_1\left|1\right\rangle) + (1-\Gamma_1)\left|0\right\rangle\right]_{q_{w(1,2,3)},\Gamma_w}\left|00\right\rangle + \\ \left|000\right\rangle\left[\Gamma2(cos\theta_2\left|0\right\rangle + sin\theta_2\left|1\right\rangle) + (1-\Gamma_2)\left|0\right\rangle\right]_{q_{w(1,2,3)},\Gamma_w}\left|01\right\rangle + \\ \left|000\right\rangle\left[\Gamma3(cos\theta_3\left|0\right\rangle + sin\theta_3\left|1\right\rangle) + (1-\Gamma_3)\left|0\right\rangle\right]_{q_{w(1,2,3)},\Gamma_w}\left|10\right\rangle + \\ \left|000\right\rangle\left[\Gamma4(cos\theta_4\left|0\right\rangle + sin\theta_4\left|1\right\rangle) + (1-\Gamma_4)\left|0\right\rangle\right]_{q_{w(1,2,3)},\Gamma_w}\left|11\right\rangle) \end{aligned} \tag{1.29}$$

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_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)\\
&CRy_{q_{w1}}^{q_4,C_1,C_2}(2\theta_4)CRy_{q_{w1}}^{q_3,\bar{C_1},C_2}(2\theta_3)CRy_{q_{w1}}^{q_2,C_1,\bar{C_2}}(2\theta_2)CRy_{q_{w1}}^{q_1,\bar{C_1},\bar{C_2}}(2\theta_1)\\
&CRy_{q_{w2}}^{q_4,C_1,C_2}(2\theta_4)CRy_{q_{w2}}^{q_3,\bar{C_1},C_2}(2\theta_3)CRy_{q_{w2}}^{q_2,C_1,\bar{C_2}}(2\theta_2)CRy_{q_{w2}}^{q_1,\bar{C_1},\bar{C_2}}(2\theta_1)\\
&CRy_{q_{w3}}^{q_4,C_1,C_2}(2\theta_4)CRy_{q_{w3}}^{q_3,\bar{C_1},C_2}(2\theta_3)CRy_{q_{w3}}^{q_2,C_1,\bar{C_2}}(2\theta_2)CRy_{q_{w3}}^{q_1,\bar{C_1},\bar{C_2}}(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_{w1}}^{q_1}(2\theta_1)CRy_{q_{w2}}^{q_1}(2\theta_1)CRy_{q_{w3}}^{q_1}(2\theta_1)CRy_{\Gamma_w}^{\Gamma_1}(2\theta_1)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|00\rangle +\\
&\quad CRy_{q_{w1}}^{q_2}(2\theta_2)CRy_{q_{w2}}^{q_2}(2\theta_2)CRy_{q_{w3}}^{q_2}(2\theta_2)CRy_{\Gamma_w}^{\Gamma_2}(2\theta_2)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|01\rangle +\\
&\quad CRy_{q_{w1}}^{q_3}(2\theta_3)CRy_{q_{w2}}^{q_3}(2\theta_3)CRy_{q_{w3}}^{q_3}(2\theta_3)CRy_{\Gamma_w}^{\Gamma_3}(2\theta_3)\,|0000\rangle_{q_{w(1,2,3)},\Gamma_w}\,|10\rangle +\\
&\quad CRy_{q_{w1}}^{q_4}(2\theta_4)CRy_{q_{w2}}^{q_4}(2\theta_4)CRy_{q_{w3}}^{q_4}(2\theta_4)CRy_{\Gamma_w}^{\Gamma_4}(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{1.30}
$$

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{1.31}
$$

Before the next step let us define:

$$q_{sin}=q_1 sin\theta_1+q_2 sin\theta_2+q_3 sin\theta_3+q_4 sin\theta_4 \tag{1.32}$$

$$q_{other}=q_1 cos\theta_1+q_2 cos\theta_2+q_3 cos\theta_3+q_4 cos\theta_4+4-q_1-q_2-q_3-q_4 \tag{1.33}$$

$$\Gamma_{sin}=\Gamma_1 sin\theta_1+\Gamma_2 sin\theta_2+\Gamma_3 sin\theta_3+\Gamma_4 sin\theta_4 \tag{1.34}$$

$$\Gamma_{other}=\Gamma_1 cos\theta_1+\Gamma_2 cos\theta_2+\Gamma_3 cos\theta_3+\Gamma_4 cos\theta_4+4-\Gamma_1-\Gamma_2-\Gamma_3-\Gamma_4 \tag{1.35}$$

$$E_{sin}=\Gamma_{sin}(q_{sin})^3 \tag{1.36}$$

$$
\begin{aligned}
E_{other}=&\Gamma_{other}(q_{other}^3+3q_{other}^2 q_{sin}+3q_{other}q_{sin}^2+q_{sin}^3)\\
&+\Gamma_{sin}(q_{other}^3+3q_{other}^2 q_{sin}+3q_{other}q_{sin}^2)
\end{aligned}
\tag{1.37}
$$

$$|\sigma_t\rangle=t_{other}|0\rangle+t_{sin}|1\rangle \tag{1.38}$$

$$\tag{1.39}$$

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

$$
\begin{aligned}
&CX_{q_{o1}}^{q_{w1},\bar{C}_1,\bar{C}_2} CX_{q_{o2}}^{q_{w2},\bar{C}_1,\bar{C}_2} CX_{q_{o3}}^{q_{w3},\bar{C}_1,\bar{C}_2} CX_{\Gamma_o}^{\Gamma_w,\bar{C}_1,\bar{C}_2} |0000\rangle_{E,q_{o(1,2,3)},\Gamma_o}(|N\rangle+|M\rangle) \rightarrow \\
&\qquad \left(CX_{q_{o1}}^{q_{w1}} CX_{q_{o2}}^{q_{w2}} CX_{q_{o3}}^{q_{w3}} CX_{\Gamma_o}^{\Gamma_w} |0000\rangle |N\rangle\right) + |0000\rangle |M\rangle \rightarrow \\
&\qquad\qquad |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |N\rangle + |0000\rangle |M\rangle
\end{aligned}
\tag{1.40}
$$

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:

$$
\begin{aligned}
&H_{C_E} CX_{E_w}^{\bar{C}_E,q_{o1},q_{o2},q_{o3},\Gamma_o} CRy_{E_w}^{C_E,\bar{C}_1,\bar{C}_2}(\theta_E) H_{C_E} |000\rangle_{E,E_w,C_E} \frac{1}{4}\Big(|\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |0000\rangle_{q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2}\Big) \rightarrow \\
&H_{C_E} \frac{1}{4}\Big(|0\rangle \frac{1}{\sqrt{2}}\Big[CX_{E_w}^{q_{o1},q_{o2},q_{o3},\Gamma_o} |0\rangle |0\rangle + Ry_{E_w}(\theta_E) |0\rangle |1\rangle\Big] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |00+0000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2}\Big) \rightarrow \\
&H_{C_E} \frac{1}{4}\Big(|0\rangle \frac{1}{\sqrt{2}}\Big[|\sigma_E\rangle_{E_w} |0\rangle_{C_E} + (cos\theta_E |0\rangle + \sin\theta_E |1\rangle)_{E_w} |1\rangle_{C_E}\Big] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |00+0000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2}\Big) \rightarrow \\
&\quad \frac{1}{4}\Big(|0\rangle \frac{1}{\sqrt{2}}\Big[|\sigma_E\rangle_{E_w} |+\rangle_{C_E} + (cos\theta_E |0\rangle + \sin\theta_E |1\rangle)_{E_w} |-\rangle_{C_E}\Big] |\sigma_q\sigma_q\sigma_q\sigma_\Gamma\rangle_{q_{o(1,2,3)},\Gamma_o} |00\rangle_{C_1,C_2} \\
&\quad + |000000\rangle_{E,E_w,C_E,q_{o(1,2,3)},\Gamma_o} [|01\rangle + |10\rangle + |11\rangle]_{C_1,C_2}\Big)
\end{aligned}
\tag{1.41}
$$

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.

$$
\begin{aligned}
&\frac{1}{4}\Bigg(|0\rangle\frac{1}{\sqrt{2}}\Big[|\sigma_E\rangle_{E_w}|+\rangle_{C_E}+(cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|-\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\quad\frac{1}{4}\Bigg(|0\rangle\frac{1}{2}\Big[|\sigma_E\rangle_{E_w}|0\rangle_{C_E}+|\sigma_E\rangle_{E_w}|1\rangle_{C_E}+(cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|0\rangle_{C_E}\\
&\quad-(cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|1\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\quad\frac{1}{4}\Bigg(|0\rangle\frac{1}{2}\Big[(|\sigma_E\rangle+cos\theta_E|0\rangle+\sin\theta_E|1\rangle)_{E_w}|0\rangle_{C_E}\\
&\qquad+(|\sigma_E\rangle-cos\theta_E|0\rangle-\sin\theta_E|1\rangle)_{E_w}|1\rangle_{C_E}\Big]+3|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[|\sigma_E\rangle+cos\theta_E|0\rangle+\sin\theta_E|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[|\sigma_E\rangle-cos\theta_E|0\rangle-\sin\theta_E|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)= \qquad (1.42)\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[E_{other}|0\rangle+E_{sin}|1\rangle+cos\theta_E|0\rangle+\sin\theta_E|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[E_{other}|0\rangle+E_{sin}|1\rangle-cos\theta_E|0\rangle-\sin\theta_E|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\frac{1}{8}\Bigg(|0\rangle[(E_{other}+cos\theta_E)|0\rangle+(E_{sin}+sin\theta_E)|1\rangle]_{E_w}|0\rangle_{C_E}\\
&\qquad+|0\rangle[(E_{other}-cos\theta_E)|0\rangle+(E_{sin}-sin\theta_E)|1\rangle]_{E_w}|1\rangle_{C_E}+6|000\rangle\Bigg)=\\
&\qquad\quad\frac{1}{8}\Bigg((E_{other}+cos\theta_E)|000\rangle+(E_{sin}+sin\theta_E)|010\rangle\\
&\qquad\quad+(E_{other}-cos\theta_E)|001\rangle+(E_{sin}-sin\theta_E)|011\rangle+6|000\rangle\Bigg)
\end{aligned}
$$

We now apply the final conditional not gate:

$$
\begin{aligned}
CX_E^{E_w,C_E}\frac{1}{8}\Big(&(E_{other}+cos\theta_E)\,|000\rangle+(E_{sin}+sin\theta_E)\,|010\rangle\\
&+(E_{other}-cos\theta_E)\,|001\rangle+(E_{sin}-sin\theta_E)\,|011\rangle+6\,|000\rangle\Big)\rightarrow\\
\frac{1}{8}\Big(&(E_{other}+cos\theta_E)\,|000\rangle+(E_{sin}+sin\theta_E)\,|010\rangle\\
&+(E_{other}-cos\theta_E)\,|001\rangle+X_E(E_{sin}-sin\theta_E)\,|011\rangle+6\,|000\rangle\Big)\rightarrow\\
\frac{1}{8}\Big(&(E_{other}+cos\theta_E)\,|000\rangle+(E_{sin}+sin\theta_E)\,|010\rangle\\
&+(E_{other}-cos\theta_E)\,|001\rangle+(E_{sin}-sin\theta_E)\,|111\rangle+6\,|000\rangle\Big)
\end{aligned}
\tag{1.43}
$$

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 $E^\Gamma-E_H^\Gamma$!

1.3.4 Calculating E^{γ} using Quantum Digital Arithmetic

The E_{γ} term is formulated as follows:

$$\eta_{AB,ll'} = \frac{1}{2}\left[\eta_A(1+K_A^l)+\eta_B(1+K_B^{l'})\right] \tag{1.44}$$

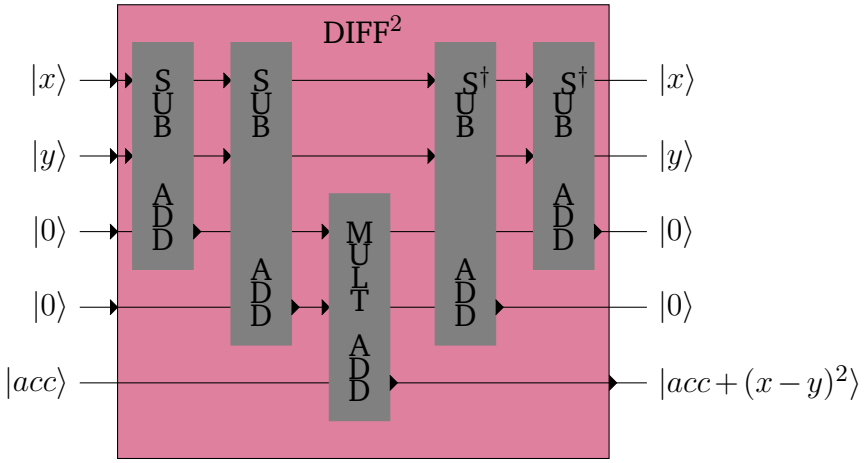
$$R_{AB}^2 = (A_x-B_x)^2+(A_y-B_y)^2+(A_z-B_z)^2 \tag{1.45}$$

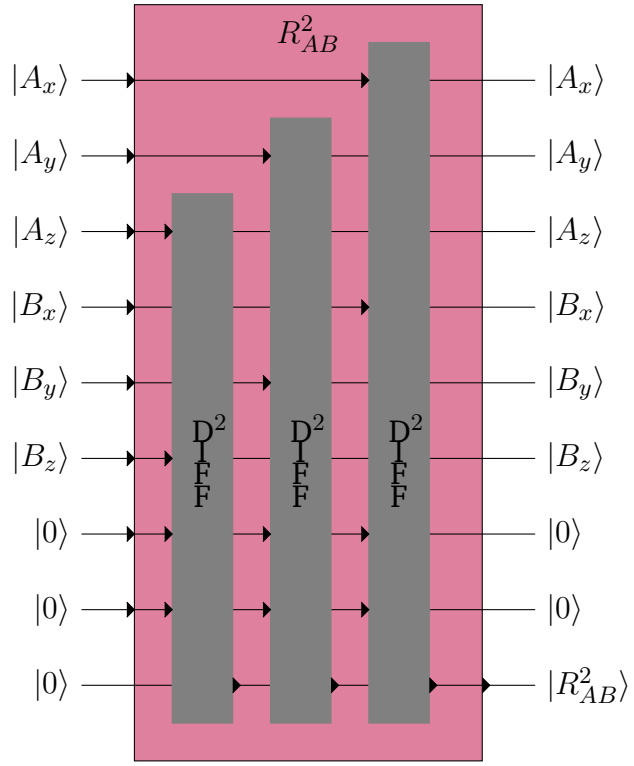
$$\gamma_{AB,ll'} = \frac{1}{\sqrt{R_{AB}^2+\eta_{AB,ll'}^{-2}}} \tag{1.46}$$

$$E_{\gamma} = \frac{1}{2}\sum_{A,B}\sum_{l\in A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'} \tag{1.47}$$

$$\tag{1.48}$$

For fullerenes we can view $\eta_{AB,ll'}$ as only dependant on the values of l and l', so there are 4 configurations as there are only 2 shells for carbon in GFN2-xTB. A small circuit for calculating R_{AB}^2 could be made up of the $diff^2$ circuit described bellow:





So using this circuit along with a constant addition of $\eta_{AB,ll'}$ and the use of an inverse square root circuit[8] we get a fixed point approximation of $\gamma_{AB,ll'}$. We can use this with our QFT fused multiply add operation to get the inner sum of the E^γ term. When repeating this over all the pairs it is a good optimisation to see that saving a little under half the computation.

$$\begin{aligned} E^\gamma &= \frac{1}{2}\sum_{A,B}\sum_{l\in A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'} \\ &= \sum_{A}\sum_{l\in A}\left[\sum_{l'\in A} q_l q_{l'}\gamma_{AA,ll'} + \sum_{B>A}\sum_{l'\in B} q_l q_{l'}\gamma_{AB,ll'}\right] \end{aligned} \tag{1.49}$$

1.3.5 Complexity

The second algorithm has, in terms of big O notation, the same complexity as the state preparation introduced in the QA-Arithmetic paper, as it is simply 4 applications of this circuit. The state preparation can be achieved using $O(\log n)$ extra qubits and $O(n\log n)$ Toffoli gates where n is the number of bits used to represent $\Gamma_\mu, q_{A_n,\mu}$. Thus if we have a m bit canonical fullerene ID we end up using on the order of $m+2n+4O(\log n) = O(n+m)$ qubits and $4O(n\log n) = O(n\log n)$ Toffoli gates.

The first circuit on the other hand uses 4 mulitplication circuits, 2 squaring circuits which could just as well be multiplication circuits and an addition circuit. A QFT addition (multiplication) circuit uses $O(n^{2(3)})$ gates and no additional qubits. Thus if we have encoded $\Gamma_\mu, q_{A_n,\mu}, \mu \in \{0,..,l\}, A_n \in \{0,..,o\}$ each using n bits we will need to perform $6lo$ multiplications and lo additions, resulting in $6lo \cdot O(n^3) + lo \cdot O(n^2) = O(lon^3)$ gates, on $m+n+nl+lon = O(m+lon)$ qubits. Additionally we then have to run the state preparation circuit which adds $O(\log n)$ qubits and $O(n \log n)$ gates, however that is not enough to change the asymptotic runtime further.

1.3.6 Cleaning up ω

We would like to get rid of the $|\omega\rangle$ term in both algorithms to avoid having to post select. We can achieve this with amplitude amplification. To do amplitude amplification we first need to define what a 'good' state is, in our case we know all good states have $|0\rangle_C |1\rangle_W$. Second we need an oracle in terms of a unitary which flips the sign of the good state, i.e. reflecting the state around the bad state, this would be $I - 2|0\rangle_C \langle 0|_C |1\rangle_W \langle 1|_W$, which can be easily implemented with controlled rotations. We also need a circuit which would reflect around the initial state by flipping the sign of it, given that we have a circuit U for preparing the initial state that would be $I - 2U|0\rangle\langle 0|U^\dagger$. In our case the angle between the bad and initial states are $\theta_{DA} = arcsin(\frac{1}{2}\frac{E^\Gamma}{2^n}), \theta_{AA} = arcsin(\frac{1}{2^4}\frac{E^\Gamma}{2^{n4}})$ for the two algorithms. We have to do $\lfloor \frac{\pi}{4\theta} \rfloor$ repetitions to maximize the probability of measuring a good state.

1.3.7 Concentrating the probabilities on the best candidates

We now have a superposition where the probability of sampling a fullerene is proportional to the energy of that fullerene. But is that ideal? The energies might be quite close to each other in absolute terms. Therefore we would like to exaggerate the difference between them and then sample according to that difference. If we knew what the highest energy was we could just subtract that from every energy calculation thus getting probabilities proportional to how much lower an energy we are working with. Another option is if we expect the energies to be within $100(1-x)\%$ we could subtract ex from every energy calculation where e is the energy from a random fullerene in the isomer-space. This is of cause not as good but quite achievable. In both algorithms we can encode ex in a register and then use a digital subtraction or do a QAA state preparation addition but with all the R_y gates inverted, resulting in a counter-clockwise rotation, in effect subtracting ex.

1.3.8 Discussion

From the asymptotic resource use the second algorithm is clearly superior, even if some of the multiplications and additions can be run in parallel in the first one. We do have a factor 8 lower chance of getting a useful state out, but this again does not change the asymptotics, as we can just repeat it. QA-Amplification might be possible since we have a very clear "good" state in both algorithms. This would reduce the need for postselection and repetitions. Preparing the initial encoding of the isomer space seems less straight forwards in the second approach than in the first unfortunately.

2 Related Work

2.1 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.

2.2 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.

3 Methodology

3.1 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 details that might appear obvious to a chemist is not explicitly described in the paper. For example, it is not specified in the equation for repulsion energy (3.1), that A and B cannot be the same atom index. The reason for this is that 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.

$$E_{rep} = \frac{1}{2} \sum_{A,B} \frac{Z_A^{eff} Z_B^{eff}}{R_{AB}} e^{-\sqrt{a_A a_B}(R_{AB})^{(k_f)}} \tag{3.1}$$

Another thing is that they do not explain clearly that the covalent radii of the atoms are different for equation 18 and 19 in the paper. For equation 19, the covalent radii is apparently computed from the dispersion model.

They do not mention they use Broyden, but just say scc. They also do not mention unit convertions, angstrom. They don't mention that repulsion should not be computed when the atom index is the same, because the distance from an atom to itself is 0, so we end up dividing by zero. There is no repulsion happening between an atom an itself.

3.2 Testing

The xTB algorithm computes a lot of different energies, corrections, and other additions to the energy terms. There is a lot of overlap between the different variants of xTB, so even focusing on just one of them still requires a great amount of code. Dealing with large computations with so many small parts makes proper validation especially important, as it becomes increasingly easy to make mistakes. In the case of the xTB algorithm, the original Fortran implementation becomes an important point of reference for comparison. Throughout the coding process it became apparent that the xTB implementation by the Grimme research group does not match the equations described in the original xTB paper by the same authors.

With this realization the obvious approach forward was to lean towards the existing implementation rather than the paper. This choice would allow us to continue doing validation against the existing code as a reference.

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 allowing 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 it was decided to write patches that allow to intercept the arguments and results of arbitrary functions just by running the program as normal. This meant that smaller parts could be implemented without the need to implement all the code needed to compute its arguments.

3.2.1 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

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

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 widely adopted feature, which provides a standard way to write Nix expressions and a way to manager 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 3.1.: 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 Fortran xTB program. This data comes from another derivation which executes patched versions of xTB and DFT-D4 on a C200 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 3.2.: 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 DFT-D4. Luckily DFT-D4 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 3.3.: This is the Nix expression for overriding the dftd4 package derivation. We essentially update dftd4 to a newer revision and add patches to logs 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.

xTB and two of its dependencies, namely CPCM-X and numsa are not in NixPkgs and had to be packaged from scratch.

3.2.2 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 3.4 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 3.4.: 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.

3.2.3 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 3.5.: 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.

Code Structure

4

```
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 ..... xTB-GFN2 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
```

```
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/ ..... test data for electro.cpp computed from caffeine
│       ├── atomicGam.txt
│       ├── dqsh.txt
│       ├── dq.txt
│       ├── H0.txt
│       ├── jmat.txt
│       ├── P.txt
│       ├── shellGam.txt
│       └── shift.txt
└── electro.cpp ..... SYCL impl for computing electrostatic energy
```

5 Challenges

5.1 Implementation Deviating from Paper

6 Results

6.1 Validation

6.2 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:

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

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 $\boldsymbol{R}_C^{2,T}$, 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.

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

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

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

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

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

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

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:

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

with

$$\sum_{r}^{N_{A,ref}} W_A^r(CN_{cov}^A, CN_{cov}^{A,r}) = 1 \tag{9.40}$$

$\mathcal{N}$ 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 $CN_{cov}^{A,r} = 0$ 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.

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

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:

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

AI Declaration

10

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

[2] 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.

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

[4] 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.

[5] 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].

[6] 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.

[7] 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.

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

Part III

Articles