

Accelerating fundamental components of a quantum chemical multilevel workflow by means of GPU offloading

Von der Fakultät für Mathematik und
Naturwissenschaften der RWTH Aachen University
zur Erlangung des akademischen Grades eines Doktors
der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Pit Steinbach
aus Esch-sur-Alzette,
Luxemburg

Berichter:	Jun.-Prof. Dr. Christoph Bannwarth
Berichter:	Univ.-Prof. Dr. Arne Lüchow
Datum der mündlichen Prüfung:	18.02.2026

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek verfügbar.

*I embrace my desire to
Feel the rhythm
To feel connected
Enough to step aside and
Weep like a widow
To feel inspired
To fathom the power
To witness the beauty
To bathe in the fountain
To swing on the spiral
Of our divinity and
Still be a human*

Lateralus by TOOL

Eidesstattliche Erklärung

Ich, Pit Steinbach, erkläre hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt:

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Promovierender dieser Fakultät und Universität angefertigt;
2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;
3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;
4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;
5. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;
6. Ein Teil oder Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in:
 - P. Steinbach, C. Bannwarth, Phys. Chem. Chem. Phys. **2024**, 26, 16567–16578.
 - P. Steinbach, C. Bannwarth, J. Chem. Theory Comput. **2025**, 21, 7335–7351.

Aachen, den 11.2025

Zusätzlich wurden während der Zeit der Promotion folgende Arbeiten veröffentlicht an denen der Autor dieser Dissertation mitgewirkt hat:

- T. Kratz, P. Steinbach, S. Breitenlechner, G. Storch, C. Bannwarth, T. Bach, J. Am. Chem. Soc. **2022**, *144*, 10133–10138
- P. Pracht, S. Grimme, C. Bannwarth, F. Bohle, S. Ehlert, G. Feldmann, J. Gorges, M. Müller, T. Neudecker, C. Plett, S. Spicher, P. Steinbach, P. A. Wesolowski, F. Zeller, J. Chem. Phys. **2024**, *160*, 114110.
- D. Kolarski, P. Steinbach, C. Bannwarth, K. Klaue, S. Hecht, Angew. Chem. - Int. Ed. **2024**, *63*, e202318015.
- A. Katbashev, M. Stahn, T. Rose, V. Alizadeh, M. Friede, C. Plett, P. Steinbach, S. Ehlert, J. Phys. Chem. A **2025**, *129*, 2667–2682.

Die folgenden Manuskripte sind zusätzlich aktuell in Ausarbeitung und eine Veröffentlichung derer ist geplant. Falls Teile dieser Arbeiten genutzt werden, wird dies durch den Autor eindeutig kenntlich gemacht.

- P. Steinbach, C. Bannwarth, *Atomistic machine learning features from semiempirical electronic structure theory and its application in reranking of conformer ensembles, in preparation.*
- P. Steinbach, M. Heezen, C. Bannwarth, *LAHVA: Linear Algebra on Heterogeneous/Vectorized Architecture - a library to simplify GPU acceleration, in preparation.*
- G. Feldmann, P. Steinbach, C. Bannwarth, *MLH: An ML Model for nuclear Hessians based on GFN2-xTB derived atomistic features, in preparation.*

Acknowledgments

I would like to express my gratitude towards the following people that have supported and helped throughout my academic career and my doctorate years:

Christoph Bannwarth. I would like to thank you for all the confidence you have put in me. You gave me the freedom to pursue projects and studies that are located on the intersection between computer science, numerics and theoretical chemistry. This allowed me to find my passion in understanding and fostering parallel programming and numerical applications. In the process of developing your group at RWTH Aachen, you gave me the opportunity to construct a compute cluster from scratch. This allowed me to gain a vast set of soft skills in this area. Your passion and your attention to detail are a source of inspiration to me.

Arne Lüchow. I thank you for the nice discussions, we had during all my years at RWTH Aachen. I admire your passion for teaching new students and introducing them to the more mathematical and theoretical side of chemistry. I would also like to thank you for co-supervising my work.

Mike Pauls, Linda Sophie Reitz. I would like to thank you for the discussions around this work and for proposing ways to express what seems straightforward to me in a way “that most chemists can understand”. I thank you for the support in discussing figures and making me aware of the little details I might have missed.

Mark Heezen. I would like to thank you for the contributions to the software projects that are part of this work.

Ying Chen, Christopher Zurek, Nils van Staaldunin, Gereon Feldmann, Daria Babushkina, Denise Scharz, Martin Schäfer, Sonja Keimes and all students that have been part of our group at some point. Thanks for the great time that I got to spend with you in the office, on conferences or other events. I am thankful for the numerous fruitful discussions, the cake and the laughter that we shared. My friends. I would like to thank all my friends who have been a support throughout my time at RWTH Aachen University. Whether they went to concerts with me or organized them with me, whether they climbed mountains with or feasted at BBQs with me. Those were key moments that helped me refocus on my research and this work.

I am very grateful to my parents Diane and Claude and my sister Liz for their love and support. They gave me the freedom and the means to pursue any studies. They have supported me financially and emotionally through all the phases of my studies in Aachen and abroad.

Abstract

The dynamic rearrangement of a molecule at finite temperature is not captured by a single static input geometry to a quantum chemical calculation. Multilevel workflows have established themselves as standard procedures to capture the dependence of the molecule's configuration on computed properties. Each step within this workflow uses an electronic structure theory method that features the optimal computational time to accuracy ratio. For exploratory steps the computational expense of a method needs to be minimal as several thousand energy and nuclear gradient evaluations are needed. Semiempirical tight binding methods, such as GFN2-xTB are often used for such purposes. However, these methods usually lack the accuracy to rank conformers relative to each other within chemical accuracy (2 kcal/mol). Therefore, so-called "low-cost" DFT methods are commonly used for geometry optimizations and conformer ranking. The combination of (semi)empirical correction models with small basis sets leads to good accuracy at moderate computational cost. However, even these highly efficient methods reach the limits of feasibility for very large systems (> 200 atoms). This work proposes solutions to partially alleviate those restrictions and allow routine calculations of these system sizes outside of large high-performance computing facilities. The key to unlock those calculations is to combine tailored algorithmic developments with consumer-grade GPUs.

A GPU-enabled linear algebra library is developed to accelerate further development and make the entry to GPU-accelerated software development easier. To demonstrate the ease of implementation, the addition of Mulliken-approximated exchange to GFN2-xTB will be shown. In most semiempirical methods, the general eigenvalue solver algorithm becomes the computational bottleneck, dominating the walltime with increasing system size. A density matrix purification algorithm is tailored to give up to six-fold acceleration of GFN2-xTB calculations using a mixed-precision approach. Lastly, 3c-methods are added to the GPU-accelerated TeraChem software package. In combination with the "mixed-precision" scheme for computing the electron repulsion integrals, this addition decreases the runtime of DFT calculations by up to an order of magnitude on mid-level consumer hardware.

The overall goal of these projects is to make GPU-accelerated calculations in quantum chemistry more available. This achieve by exposing algorithmic developments that are performant on reasonably priced hardware and making GPU computing more accessible to non-experts.

Contents

Acronyms	viii
1 Motivation and state of the art	2
2 Methodological background	7
2.1 Electronic structure theory methods	7
2.1.1 Hartree Fock	8
2.1.2 Density functional theory	11
2.1.3 Semiempirical electronic structure methods	12
2.1.4 GFN2-xTB	14
2.1.5 SCF procedure in GFN2-xTB	16
2.2 GPU hardware and programming using the Compute Unified Device Architecture (CUDA) model	18
3 Introduction into GPU implementation in quantum chemistry	20
3.1 Computation of Electron repulsion integrals on GPUs	20
3.2 Floating point representations and mixed precision strategies	24
3.3 Half-precision splitting schemes	26
4 LAHVA: Linear algebra on heterogeneous/vectorized architecture - a li- brary to simplify GPU acceleration	29
4.1 Motivation and library functionality	29
4.2 Introduction	31
4.3 Computational details	33
4.4 Key features and timing results	34
4.4.1 Templated tensor classes	35
4.4.2 Using LAHVA's tensor classes to implement a base class for Mul- liken exchange	38
4.4.3 Switching compute hardware with one parameter	39
4.4.4 Half precision splitting schemes	43
4.4.5 Asynchronous calculations on a single GPU	47
5 Acceleration of semiempirical electronic structure theory calculations on consumer-grade GPUs using mixed-precision density matrix purification	49
5.1 Introduction	49
5.2 Theory	53
5.2.1 Density matrix purification methods	53

5.2.2	Mixed precision approximations	55
5.3	Computational details	58
5.3.1	Implementation details	59
5.4	Numerical accuracy evaluation	60
5.5	Timings for single-point calculations and geometry optimizations	64
5.5.1	Single-point calculations	64
5.5.2	Geometry Optimization	67
5.6	Acceleration of a conformer sampling protocol	69
6	Combining low-cost electronic structure theory and low-cost parallel com- puting architecture	72
6.1	Introduction	72
6.1.1	Low-cost 3c composite methods with Fock exchange	73
6.2	Computational details	75
6.3	Element availability of 3c-methods in TeraChem	76
6.4	Comparing SCF walltimes for implemented 3c-methods	80
6.5	Price-to-performance comparison of GPU models	83
7	Conclusion & outlook	86
8	Appendix	102
8.1	Computational hardware	102
8.2	Combining low-cost electronic structure theory and low-cost parallel computing architecture	102
8.2.1	Implementation details and element availability	102

Acronyms

API application programming interface

CAMM cumulative atomic multipole moment

COSX chain-of-spheres for HF exchange

CREST conformer rotamer ensemble searching tool

DFA density functional approximation

DFT density functional theory

DFTB density functional tight binding

DMM density matrix minimization

DMP density matrix purification

DMPE density matrix polynomial expansion method

ECD electronic circular dichroism

ECP effective core potential

ERI electron repulsion integral

FP32 single precision floating point

FP64 double precision floating point

GGA generalized gradient approximation

GPT Gaussian product theorem

GPU graphics processing unit

HF Hartree Fock

HGP Head-Gordon-Pople

LDA local density approximation

MAD mean absolute deviation

MD molecular dynamics

MMD McMurchie-Davidson

MO molecular orbital

MSRP manufacturer's suggested retail price

MTD metadynamics

PES potential energy surface

RIJ resolution-of-the-identity for the Coulomb integral

RMSD root mean square deviation

SCC self-consistent charge

SCF self-consistent field

SQM semiempirical quantum mechanical

xTB extended tight binding

1 Motivation and state of the art

For finite temperatures, a molecule rearranges itself, occupying many geometries over time. The number of potential structures grows exponentially with the number of rotatable bonds. Even with constant connectivity, the arrangement of a molecule's atom impacts its physical and chemical properties. Therefore, it is crucial to consider the flexibility of a molecule for simulations that aim to come close to experimental values. However, computing all possible geometries becomes unfeasible for large molecules, moreover only those with the lowest energy contribute significantly to the overall property. Hence, a systematic workflow to explore all thermally accessible geometries is needed. To achieve this goal, a multilevel workflow is introduced that combines various computational steps to reduce the number of potential conformer structures with every step, see fig. 1.1.

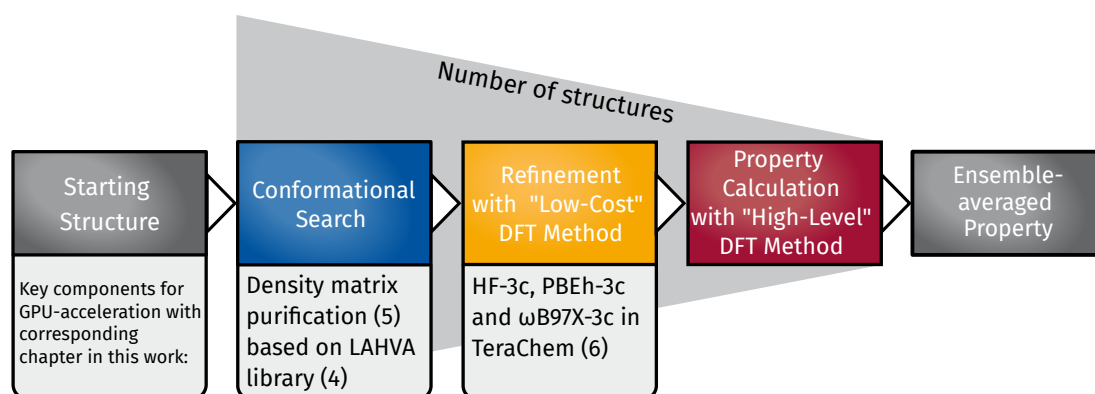
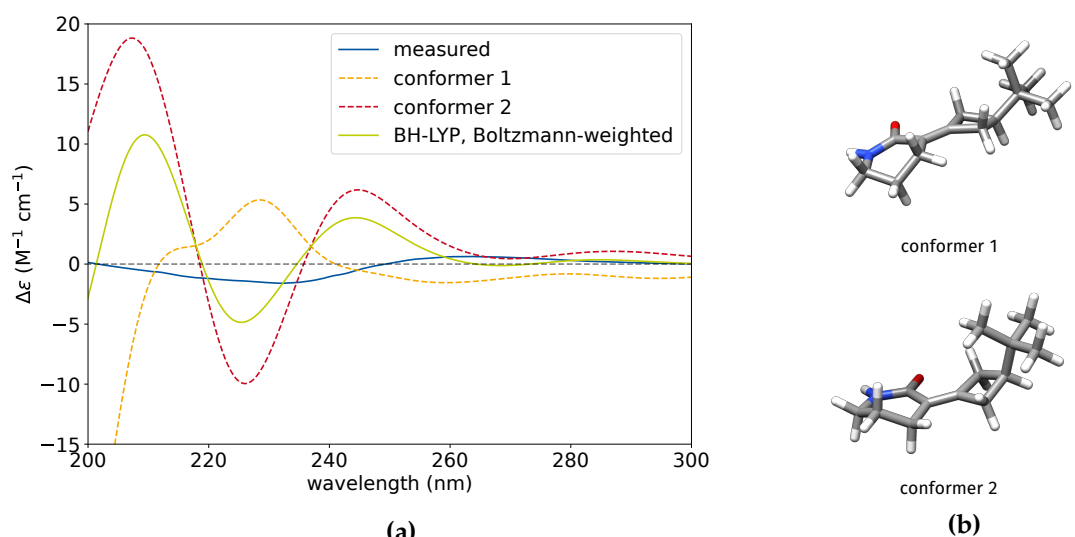


Fig. 1.1: The multilevel workflow is composed of three main computational steps with filtering steps after each step. Based on a single input structure, the purpose of the conformational search steps is to produce a set of potentially relevant/thermally accessible conformers. After retaining the most relevant structures at a fast but rather inaccurate level of theory, a more accurate level of theory is used to optimize the geometries to local minima. The energy and property of the thermally accessible conformers is then computed using highly accurate methods.

To underline the importance of this approach, we want to present an application that demonstrates the need for such a multilevel workflow to achieve a reliable prediction of experimental properties. In a cooperation with the group of Prof. Bach, we determined the absolute configuration of a photochemical reaction product by means of simulations.^[1] Electronic circular dichroism (ECD) is directly linked to the absolute configuration of a molecule and allows to discern between enantiomers. We applied the multilevel workflow to the molecule, resulting in two conformers with very similar

energies. In fig. 1.2, we show the ECD spectra of both conformers gained by averaging over the snapshots of molecular dynamics (MD) simulations started from their respective minima. We also show the Boltzmann-weighted average of the two spectra and the experimental spectra.



conformer	TD-BHLYP/def2-TZVPD (° mL (dm g) ⁻¹)	TD-PBE0/def2-TZVPD (° mL (dm g) ⁻¹)	Boltzmann-weights PW6B95-D4/def2-QZVPP
1	-22.1	-23.3	0.653
2	29.8	26.6	0.347
Boltzmann-average	-4.1	-6.0	exp. value: -2 ° mL (dm g) ⁻¹

Fig. 1.2: (a) Simulated electronic circular dichroism spectrum at the sTD-BH-LYP/def2-TZVPD level compared to the experimental spectrum.^[2-5] For the full computational procedure, we direct the reader to ref. [1]. (b): Depiction of the ground state minimum geometries of the involved conformers, computed at the PBEh-3c^[6] level. (c): Specific angle of rotation computed with the TD-BHLYP and TD-PBE0/def2-TZVP levels of theory for conformers 1 and 2. The experimentally determined value is also given. **Reproduced with permission from: Thilo Kratz, Pit Steinbach, Stefan Breitenlechner, Golo Storch, Christoph Bannwarth, and Thorsten Bach Journal of the American Chemical Society 2022 144 (23), 10133-10138 DOI: 10.1021/jacs.2c02511. Copyright 2022 American Chemical Society.**

However, both conformers show opposing signs for some peaks in the spectrum. We can see that neither conformer alone would have been able to allow certain determination of the absolute configuration. Although, the ECD spectrum of conformer 2 is qualitatively in accordance with the experiment, the computed specific angle of rotation of that conformer alone is on the wrong order of magnitude. Without exploring the conformational space first, a prediction would not have been possible with confidence.

After giving a short motivation for a multilevel workflow as a default strategy for quantum chemical calculations in the ground-state, we will present the individual steps of a workflow variant mainly based on the works of Grimme and coworkers^[6–11] which is widely used as a standard procedure.^[1,12–15] We have identified the potential to accelerate multiple computational steps within this workflow.

The workflow starts with an input geometry, since this geometry is usually not fully relaxed, we first perform a geometry optimization. For this purpose we need an electronic structure theory method with a good cost to accuracy ratio. During the last decade, there has been a “renaissance” of semi-empirical methods, which were developed to serve this purpose.^[9,16,17,17–21] They are based on physical models and foundations, however, they use empirical parameters in combination with simplified energy expressions to improve their computational scaling compared to “ab initio” methods. In the upcoming chapter, we will further focus on the derivation of such models. For these methods, computing the one particle density matrix is commonly the computational bottleneck. Generally, the density matrix is built using the coefficient matrix obtained from solving the Roothaan-Hall (see eq. 2.13) eigenvalue equations. Accelerating this step would have an effect on the computational time of every calculation using this method and is therefore a perfect target for this work.

The goal of the first step of the multilevel workflow is to identify the most promising conformers. This task leads to a combinatorial challenge as a systematic exploration based on dihedral angles would lead to exponential scaling with the number of rotatable bonds. Alternatives to this combinatorial approach have been proposed, such as the conformer rotamer ensemble searching tool (CREST) workflow.^[7,8,22] In this approach, MD simulations are extended with a bias potential based on the history of conformers previously sampled.^[22] These metadynamics (MTD) calculations push the simulation towards unseen regions of the potential energy surface (PES). Another approach, recently developed by Zurek *et al.* is the so called “TTConf” method.^[23] Here, a tensor train-based representation of the torsion angle space is used to reduce the scaling in exploring the energetically relevant conformer space. TTConf thereby eliminates the use of time-consuming MD or MTD calculations to explore the PES, instead relying on ensemble geometry optimizations. In both approaches, semi-empirical methods are used to compute the energy and nuclear gradient of the molecule. Hence, a method to accelerate the density matrix build would also speed-up these workflows.

The result of conformational sampling is an ensemble of structures with their associated energies. To determine the most relevant structures according to Boltzmann statistics, very accurate energies are necessary. However, semi-empirical methods are generally not accurate enough to determine the relative energies of the conformers with such high accuracy. Therefore, we need to reoptimize the most relevant conformers using a more accurate method, such as density functional theory (DFT). Similar to our previous steps, we have the possibility to use empirically parametrized methods to speed up the computation time. In 2013, Grimme introduced the concept of using

Tab. 1.1: Launch prices of consumer grade and general purpose (Tesla) GPUs, and their performance for various floating point representations.^[30] FLOPS is the unit of performance and refers to floating point operations per second. The launch prices of the V100 and A100 are tentative as they are nearly only sold in complete systems. The definition of the floating point standards FP32 and FP64 can be found in sec. 3.2.

GPU	Price [USD]	FP64 [TFLOPS]	FP32 [TFLOPS]	GPU RAM [GB]
GeForce RTX 3070	499	0.31	20.3	8
GeForce RTX3080Ti	1199	0.53	34.1	12
GeForce RTX4090	1599	1.29	82.6	24
Quadro RTX5000	2299	0.35	11.1	16
Tesla V100	~8999	8.18	16.35	16
Tesla A100	~15000	9.75	19.5	40

an “ab-initio” level of theory in combination with a small basis set and empirically parameterized corrections, e.g. in the form of interatomic potentials.^[24] These composite methods are called “3c-methods”.^[6,11,24,25] As for other ab-initio methods, the evaluation of the electron repulsion integrals (ERIs) is the computational bottleneck. Therefore, we would need to address the acceleration of these integrals with graphics processing units (GPUs) to speed up this step.

The last step of the multilevel workflow is to compute the target property, which can be the Gibbs free energy or also commonly spectroscopic properties. For this step, usually a DFT or wavefunction calculation with a large basis set is used. These calculations should only be run for the most relevant structures, previously identified in the workflow. Accelerating this step comes with even more challenges, than the previous step. Most schemes to compute the ERIs rely on recursion relations, leading to a steep scaling of the number of intermediates necessary to compute high angular momenta. This rapidly increasing complexity and memory requirement for high angular momentum ERIs is a challenge for GPU implementations. Approximations that reduce the formal scaling of the ERI computation, like resolution-of-the-identity for the Coulomb integral (RIJ)^[26] and chain-of-spheres for HF exchange (COSX)^[27–29] are slowly being integrated in GPU ERI implementations. The acceleration of these methods with GPUs is an active field of research and beyond the scope of this work.

We will present several algorithmic developments that are tailored towards a specific type of hardware. We have focussed on consumer-grade GPUs, which refers to the hardware initially developed for the purpose of rendering graphics simulations, particularly, in computer games. This task can be trivially parallelized since every pixel can be computed independently, and the same operation is executed for every pixel. This type of parallelization is called “SIMD” (single instruction multiple data) and is the basis for the GPU architecture. This model lends itself well to the acceleration of linear algebra operations, which are frequently used in quantum chemistry

software. However, the computation of pixels only needs a limited precision, since only a moderate amount of colors can be displayed by a monitor. Hence, the architecture of consumer-grade GPUs was optimized from the ground up for single precision floating point (FP32) numbers, see ch. 3.2.^[31,32] The main quantity computed in quantum chemical applications is the total electronic energy computed in the Born-Oppenheimer approximation, corresponding to the energy gained by associating free, non-interacting electrons to the nuclei from prior infinite distance. This quantity grows linearly in numerical value with electron count, however, the difference in energy due to changes in chemical bonds are far smaller than the overall electronic energy. Hence, for many applications 8 or more significant digits are necessary to accurately calculate relative energies that correspond to chemical reactions. The double precision floating point (FP64) standard offers up to 15 digits of accuracy making it the default precision for quantum chemical calculations. Manufacturers have introduced new hardware designed for scientific computing, which are referred to as general-purpose GPUs, that is tailored towards FP64 performance. These specialized accelerators come with a much higher cost than their consumer-grade counterparts, see tab. 1.1. This price restricts the access of researchers to GP-GPUs, especially, if the access to high performance computing clusters is limited. Hence, by targeting CG-GPU hardware, we are making an effort to “democratize” the access to GPU acceleration in quantum chemistry. Additionally, the use of lower grade hardware challenges us to develop new algorithms that save compute time on high-grade and lower-grade GPUs alike.

The following dissertation will be structured into a chapter introducing the methodological background, followed by an introduction into GPU implementations in quantum chemistry. In the first section of the upcoming chapter, the electronic structure theory methods used in this work will be recapitulated. The following section briefly introduces the GPU hardware and how the programming model is a consequence of the hardware. The hardware limitations presented in that section partially influence how the SCF workflow is implemented for GPUs, which is the focus of the first section of the ch. 3. The second section of ch. 3 finishes with a review of mixed numerical precision strategies. The main projects that led to an important acceleration of the multilevel workflow are introduced in the next three chapters. This dissertation will be rounded off by a resume of what has been achieved by this work and which areas could be tackled in further publications.

2 Methodological background

2.1 Electronic structure theory methods

In the following chapter, we will give a short introduction into the electronic structure theory methods relevant to this work. For a complete derivation, especially beyond mean-field methods, consider the following refs. [33,34].

A system of quantum-mechanical particles, for which relativistic effects and spin are not considered, can be described by the (time-dependent) Schrödinger equation.^[35]

$$i\hbar \frac{\partial}{\partial t} \Psi(\mathbf{r}, \mathbf{R}, t) = \hat{H} \Psi(\mathbf{r}, \mathbf{R}, t) \quad (2.1)$$

$\Psi(\mathbf{r}, \mathbf{R}, t)$ is the many-body wave function of the system of electrons and nuclei. The electron positions are collectively referred to by $\mathbf{r}$, while $\mathbf{R}$ is used for the nuclei positions. $\Psi(\mathbf{r}, \mathbf{R}, t)$ is complex-valued and evolves in time t . The wave function fully characterizes the system, however $\Psi(\mathbf{r}, \mathbf{R}, t)$ itself has no direct physical interpretation. The absolute square of that function $|\Psi(\mathbf{r}, \mathbf{R}, t)|^2$ gives the probability density function, which is an observable. $\hat{H}$ is the Hermitian Hamilton operator, and is given as a sum of the kinetic $\hat{T}$ and the Coulomb interaction-based potential $\hat{V}$ energy operators: $\hat{H} = \hat{T} + \hat{V}$. For relativistic particles, the formulation of the Hamiltonian operator and the wave function becomes vastly more complicated, which is beyond the scope of this work.

For systems where the potential energy operator $\hat{V}$ is independent of the time, we can simplify eq. 2.1 through the separation of time and space variables, yielding eventually the time-independent Schrödinger equation (eq. 2.2). The solutions to the time-independent Schrödinger equation are stationary states.

$$\hat{H} \Psi(\mathbf{r}, \mathbf{R}) = E(\mathbf{r}, \mathbf{R}) \Psi(\mathbf{r}, \mathbf{R}) \quad (2.2)$$

Commonly, in chemistry we consider a collection of atoms as our system of interest. We know, that the mass of electrons m_e is far smaller than that of the nuclei (i.e. protons and neutrons) $2000m_e \approx m_p$. Hence, we can assume that the electrons move much faster and will instantly adapt to a redistribution of the nuclei, which motivates the separation of nuclear and electronic coordinates. In the following equations, we present the consequences of this separation on the previously seen quantities, we

denote by the suffix e properties of the electrons and by n those of the nuclei:

$$\hat{H} = \hat{H}_e + \hat{T}_n$$

$$\text{where: } \hat{H}_e = \hat{T}_e + \hat{V}_{ee} + \hat{V}_{en} + \hat{V}_{nn} \quad (2.3)$$

$$\Psi(\mathbf{r}, \mathbf{R}) = \Psi_n(\mathbf{R})\Psi_e(\mathbf{r}; \mathbf{R}) \quad (2.4)$$

$$\hat{H}_e\Psi_e(\mathbf{r}; \mathbf{R}) = E_e(\mathbf{R})\Psi_e(\mathbf{r}; \mathbf{R}) \quad (2.5)$$

The potential energy contributions have been separated into electron-electron $\hat{V}_{ee}$, electron-nuclei $\hat{V}_{en}$ and nuclei-nuclei interactions $\hat{V}_{nn}$. The total wave function $\Psi(\mathbf{r})$ can be separated in a product of the nuclear $\Psi_n(\mathbf{R})$ and the electronic $\Psi_e(\mathbf{r}; \mathbf{R})$ wave function. By neglecting the coupling between the electronic and nuclear velocities, we work under the assumption of electrons moving in a field of fixed nuclei. Thereby, $\Psi_e(\mathbf{r}; \mathbf{R})$ depends parametrically on the coordinates of the nuclei. The change to parametric dependence is shown by ; in contrast to the previous use of , in eq. 2.2 for example. The separation of variables gives the electronic Schrödinger eq. 2.5. This separation is known as the “Born-Oppenheimer” approximation. The electronic Hamiltonian operator $\hat{H}_e$ acts on $\Psi_e(\mathbf{r}; \mathbf{R})$ giving the electronic energy $E_e(\mathbf{R})$ as its eigenvalues. $E_e(\mathbf{R})$ is now a function of the nuclear coordinates, creating the familiar description of the internal nuclear degrees of freedom that define a PES on which we can manipulate the nuclear coordinates. So far, we have not introduced a specific formulation of $\Psi_e(\mathbf{r}; \mathbf{R})$ nor $\hat{H}_e$, these will be the focus of the upcoming sections.

2.1.1 Hartree Fock

The first “ab initio” electronic structure theory used on a wide scale is Hartree Fock (HF) theory. Ab initio is used in this field to refer to methods that are derived from physical principles without the need for “empirical” parameters. A short summary of the major aspects of the HF method is presented here, for a complete theoretical derivation please consider the following reference [33].

Starting from eq. 2.3 and setting aside $\hat{V}_{nn}$, for a system of N electrons in a field of M nuclei with a nuclear charge of Z_A , the electronic Hamiltonian $\hat{H}_e$ is given (in atomic units) by:[33]

$$\hat{H}_e = \hat{T}_e + \hat{V}_{ne} + \hat{V}_{ee} \quad (2.6)$$

$$= \sum_{i=1}^N -\frac{1}{2} \nabla_i^2 - \sum_{i=1}^N \sum_{A=1}^M \frac{Z_A}{|\mathbf{r}_i - \mathbf{R}_A|} + \sum_{i=1}^N \sum_{j>i}^N \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|} \quad (2.7)$$

$$\text{with: } \nabla_i = \left(\frac{\partial}{\partial x_i}, \frac{\partial}{\partial y_i}, \frac{\partial}{\partial z_i} \right)$$

$\hat{V}_{nn}$ is a constant value for each distribution of nuclei and can be added after solving for the electronic structure. The position of the electron $\mathbf{r}_i$ and the nucleus $\mathbf{R}_A$

are expressed as vectors in Cartesian coordinates. For the remainder of this chapter the parametric dependence on $\mathbf{R}$ will be neglected. Another fundamental property of electrons is their spin ω , which was previously not considered explicitly. Therefore, we introduce a new variable $\mathbf{x}$, which is a vector containing the Cartesian position $\mathbf{r}$ and spin of the electrons ω .

To compute the electronic energy $E_e(\mathbf{R})$, we need an ansatz for the electronic many-particle wave function $\Psi_e(\mathbf{x}; \mathbf{R})$. For the remainder of this chapter the parametric dependence on $\mathbf{R}$ will be neglected. The many-particle wavefunction $\Psi_e(\mathbf{x})$ is derived by combining single-particle wavefunctions $\psi(\mathbf{x})$. These single-particle wavefunctions $\psi(\mathbf{x})$, also known as spin orbitals, are defined as a product of the spatial orbital $\varphi(\mathbf{r})$ and a spin function $\sigma(\omega)$. For fermions, such as electrons, the many-particle wavefunction $\Psi_e(\mathbf{x})$ must fulfill the anti-symmetry rule. This rule states that the interchange of the coordinates $\mathbf{x}$ of two particles must lead to a change of the sign of the many-particle wavefunction. This condition is fulfilled by Slater determinants. Equation 2.9 shows how spin orbitals $\psi_i(\mathbf{x}_j)$, single electron wave functions, are combined to form an anti-symmetric many-electron wave function $\Psi(\mathbf{x})$. In the case of molecular systems, $\varphi_i(\mathbf{x}_j)$ is also often referred to as molecular orbital (MO). In HF, a single Slater determinant is used for Ψ .^[33]

$$\psi_i(\mathbf{x}_j) = \varphi_i(\mathbf{r}_j)\sigma(\omega) \quad (2.8)$$

$$\Psi_{\text{SD}}(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) = \Psi_{\text{SD}}(\mathbf{x}) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(\mathbf{x}_1) & \psi_2(\mathbf{x}_1) & \cdots & \psi_n(\mathbf{x}_1) \\ \psi_1(\mathbf{x}_2) & \psi_2(\mathbf{x}_2) & \cdots & \psi_n(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\mathbf{x}_n) & \psi_2(\mathbf{x}_n) & \cdots & \psi_n(\mathbf{x}_n) \end{vmatrix} \quad (2.9)$$

However, the Slater determinant $\Psi_{\text{SD}}(\mathbf{x})$ is not an exact eigenfunction of the electronic Hamiltonian $\hat{H}_e$ (see eq. 2.5). The Rayleigh-Ritz variation principle allows us to formulate an expectation value for the electronic energy E_e . When using orthonormal orbitals to build a normalized Slater determinant, the expression can be simplified as the denominator $\int \Psi_{\text{SD}}(\mathbf{x})^* \Psi_{\text{SD}}(\mathbf{x}) d\tau$ becomes 1.

$$E_e = \frac{\int \Psi_{\text{SD}}^*(\mathbf{x}) \hat{H} \Psi_{\text{SD}}(\mathbf{x}) d\tau}{\int \Psi_{\text{SD}}^*(\mathbf{x}) \Psi_{\text{SD}}(\mathbf{x}) d\tau} = \int \Psi_{\text{SD}}^*(\mathbf{x}) \hat{H} \Psi_{\text{SD}}(\mathbf{x}) d\tau \quad (2.10)$$

In the previous equation, $d\tau$ is used as short notation for the integration over all variables. The expectation value for the energy of the HF approximation can be determined by careful integration and its expression can be simplified by accounting for the orthonormal orbitals and spin integration. After these steps, which can be found among others in the following ref. [33,34], we get the following energy for a restricted

closed-shell system:

$$E_e[\psi] = 2 \sum_{i=1}^{N/2} \int \varphi_i(\mathbf{r}_1)^* \hat{h} \varphi_i(\mathbf{r}_1) d\mathbf{r}_1 + \sum_i \sum_{j>i}^{N/2} 2 (\varphi_i \varphi_i | \varphi_j \varphi_j) - (\varphi_i \varphi_j | \varphi_j \varphi_i) \quad (2.11)$$

$$\text{with: } \hat{h} = -\frac{1}{2} \nabla_1^2 - \sum_{A=1}^M \frac{Z_A}{|\mathbf{r}_1 - \mathbf{R}_A|}$$

note: the Mulliken notation is used for two electron integrals:

$$(\varphi_i \varphi_j | \varphi_k \varphi_l) = \iint \varphi_i(\mathbf{r}_1) \varphi_j(\mathbf{r}_1) \varphi_k(\mathbf{r}_2) \varphi_l(\mathbf{r}_2) d\mathbf{r}_1 d\mathbf{r}_2$$

The first term of the energy expression is often referred to as “one-electron” Hamiltonian or “core” Hamiltonian. The second term depends on two electrons and expresses the interaction of electrons. We refer to the first double integral as Coulomb term and the second as exchange term. Starting from expression 2.11, the Lagrange formalism can be used to minimize the energy under the constraint of orthonormal MOs. This derivation yields the canonical Hartree-Fock equations. At this point we are introducing the basis set approximations as a tool to systematically improve our MOs. The MO $\varphi_i(\mathbf{r})$ is described as a linear combination of K_i atomic basis functions χ_μ : $\varphi_i(\mathbf{r}) = \sum_{\mu}^{K_i} C_{\mu i} \chi_\mu(\mathbf{r})$. We are using the indices i and j to refer to the MO orbitals, whereas we use μ , ν , κ and λ to refer to individual basis set functions. The coefficients $C_{\mu i}$ of these atomic orbitals (AOs) can then be optimized while keeping the basis functions constant. Therefore, we are introducing a new quantity the one-electron density matrix (in the AO basis), the element of that matrix for the AOs μ and ν is: $D_{\mu\nu} = \sum_j^N C_{\mu j} C_{\nu j}$. In eq. 2.12, we define the elements $F_{\mu\nu}$ of the effective one-electron Fock operator in the AO basis. Eq. 2.13 takes the form of a generalized eigenvalue problem, which can be solved by diagonalizing $\mathbf{F}$. The result of this operation are the eigenvectors $\mathbf{C}$, used to build a new density matrix $\mathbf{D}$, and the eigenvalues ε , corresponding to the orbital energies. From the definition of Fock matrix element for the AOs μ and ν , we can see that the second term of $F_{\mu\nu}$ depends on a prior solution of the electronic structure via $D_{\kappa\lambda}$. Hence an iterative solution of the problem is necessary. This iterative variation procedure is called self-consistent field (SCF) and is converged using pre-defined thresholds, based on the change in energy and density for example.

$$F_{\mu\nu} = \int \chi_\mu^* \hat{h} \chi_\nu d\mathbf{r}_1 + \sum_{\kappa}^{N_{bf}} \sum_{\lambda}^{N_{bf}} D_{\kappa\lambda} \left\{ (\chi_\mu \chi_\nu | \chi_\kappa \chi_\lambda) - \frac{1}{2} (\chi_\mu \chi_\kappa | \chi_\lambda \chi_\nu) \right\} \quad (2.12)$$

$$\mathbf{FC} = \mathbf{SC}\varepsilon \quad (2.13)$$

The HF method gives the “lowest possible energy” solution under the restriction of using a single Slater determinant as wave function. However, this approximation leads to the neglect of explicit correlation effects between electrons, which is an important effect especially in bond breaking situations. The electron correlation can be regained by incorporating additional Slater determinants in the calculations, as done in methods such as configuration interaction, perturbation theory-based methods such as Møller-Plesset and coupled cluster. This leads to a steep increase in computational complexity

and memory requirements. This increase in computation time is so severe that it limits the use of these methods for the application in a multilevel workflow (see fig. 1.1). The steep increase in necessary memory capacity makes the computation of such methods on CG-GPUs nearly impossible, as only limited memory resources are present on these devices. Therefore, we will not be focusing on these methods and their theoretical derivation in this work.

2.1.2 Density functional theory

An alternative formulation to partially describe electron correlation is DFT, the foundation of which has been proposed by Hohenberg and Kohn in 1964.^[36] The central quantity in their derivation is the electron density ρ in contrast to the wave function Ψ . The first theorem they propose states that in a system with N electrons, the external potential acting on the electrons uniquely determines the ground state electron density ρ_0 . Further, they show that the ground state energy E_0 is a functional of ρ_0 and that, similar to HF, the variational principle holds for $E_0[\rho_0]$. Conceptually, this makes DFT an elegant theory, as ρ_0 is the only variable to be optimized and hence no extensive integral calculations over all occupied orbitals would be necessary as compared to HF for example. However, the authors could also show that the exact functional that incorporates all interactions is unknown and cannot be deduced in a systematic fashion. The kinetic energy contribution turned out to be a problematic term in the derivation of an energy functional $E[\rho]$, as the expression for the uniform electron gas turned out to be useless when applied to molecules.^[37]

In 1965, Kohn and Sham proposed a solution to this problem: the introduction of orbitals into DFT.^[38] The kinetic energy can thus be computed in a similar fashion to HF theory. Formally, this corresponds to the kinetic energy of non-interacting electrons T_0 , however, the difference to the kinetic energy of interacting electrons is considered small.^[37] Equation 2.14 is used to derive the electron density $\rho(\mathbf{r})$ from these Kohn-Sham-orbitals φ_i . f_i denotes the occupation number of the orbital i .

$$\rho(\mathbf{r}) = \sum_{i=1}^N f_i |\varphi_i(\mathbf{r})|^2 \quad (2.14)$$

The Kohn-Sham DFT energy term is the sum of T_0 , the electron-nucleus $E_{\text{ne}}[\rho]$, the classical interelectronic Coulomb interaction $J[\rho]$, and the exchange-correlation term $E_{\text{XC}}(\rho)$, see eq. 2.15. The main development task in DFT is the formulation of $E_{\text{XC}}(\rho)$, as it describes exchange and correlation effects, i.e., the difference for the Coulomb energy of a system of interacting electrons $E_{\text{ee}}[\rho]$ and the classical mean-field interelectronic Coulomb energy $J[\rho]$. In theory, it should also correct the assumption of non-interacting electrons in the computation of the kinetic energy, see eq. 2.16.

$$E_{\text{KS-DFT}} = T_0[\rho] + E_{\text{ne}}[\rho] + J[\rho] + E_{\text{XC}}[\rho] \quad (2.15)$$

$$E_{\text{XC}}[\rho] = (T[\rho] - T_0[\rho]) + (E_{\text{ee}}[\rho] - J[\rho]) = E_{\text{X}}[\rho] + E_{\text{C}}[\rho] \quad (2.16)$$

Commonly, the exchange and correlation functional formulations are independent from each other and are computed separately. The exchange-correlation functionals are also called density functional approximation (DFA) and are characterized by their functional dependence on the electron density. The most rudimentary approach is to derive the exchange-correlation energy from a uniform electron gas. This approximation is known as local density approximation (LDA), and only the local value of ρ is used to compute the local energy, which is integrated over the entire space to yield the total XC energy. The generalized gradient approximation (GGA) takes the gradient $\nabla\rho(\mathbf{r})$ into consideration, making it more suitable to describe systems with nonuniform ρ as in molecules for example. DFAs using the second derivative of the electron density or kinetic energy density are referred to as meta-GGAs. In the previous section we have seen the exchange term in eq. 2.11, which describes the exchange effect exactly for single Slater determinant. In 1993, Becke proposed mixing the non-local HF-type exchange with the local exchange from the aforementioned DFAs.^[4] These so-called hybrid DFT methods turned out to be very successful, especially, for excited states calculations and barrier heights.^[39] One of the key parameters in the development of a hybrid DFA is the ratio a_x of HF exchange E_X^{HF} used.

$$E_X = a_x \cdot E_X^{\text{HF}} + (1 - a_x) \cdot E_X^{\text{DFT}} \quad (2.17)$$

The use of non-local HF-exchange already corrected the asymptotic behavior of the exchange potential from exponential decay to $-\frac{a_x}{r}$. The correct decay would be $-\frac{1}{r}$, which would imply 100 % HF-exchange and no electron density-based local exchange. A distance-based scaling of the HF-exchange has been proposed by Hirao *et al.*^[40] Head-Gordon and Chai then proposed a range of so called “range-separated” hybrid functionals that have been very popular both for excited states chemistry and thermochemistry applications.^[39,41–43]

2.1.3 Semiempirical electronic structure methods

In the previous two subsections, we have given an overview on HF and DFT theory, which are mostly based on physical principles and do not introduce any, or at best (for DFT) very few empirical parameters. However, implementations of these theories come with a substantial computational cost. Already before the advent of computers, more approximate electronic structure theories have been derived, such as the Hückel method^[44,45] for π -electrons or its extension to arbitrary electrons (extended Hückel) by Hoffmann.^[46] Such methods are commonly characterized as “semiempirical”, as they present a physical foundation but also employ empirical parameters while approximating, for example, the explicit computation of integrals.^[16,17,47–51] Moreover, they delimit themselves from classical force field methods that do not solve for the electronic structure but are only based on the nuclei coordinates, classical interatomic energy terms, and empirical parameters.

The semiempirical quantum mechanical (SQM) methods are classified based on their derivation from HF or DFT theory.^[16,17] Semiempirical molecular orbital theory is

based on the former, it is not further studied in this work and we point the reader to a comprehensive review.^[16] In this work, we are focussing on methods based on density functional tight binding (DFTB). This type of SQM method was pioneered by Seifert, Elstner and coworkers around the turn of the century.^[18,19,52,53] The derivation of DFTB methods starts from the energy expression of KS-DFT using the LDA approximation for the exchange-correlation functional, see eq. 2.18.

$$E_{\text{tot}}^{\text{KS-DFT}} = E_{\text{nn}} + \sum_i^{N_{\text{MO}}} f_i \int \psi_i^*(\mathbf{r}) \left[\hat{T}(\mathbf{r}) + V_{\text{ne}}(\mathbf{r}) + \varepsilon_{\text{XC}}^{\text{LDA}}[\rho(\mathbf{r})] + \frac{1}{2} \int \left(\frac{1}{|\mathbf{r} - \mathbf{r}'|} \right) \rho(\mathbf{r}') d\mathbf{r}' \right] \psi_i(\mathbf{r}) d\mathbf{r} \quad (2.18)$$

Based on this expression the tight-binding approximation is introduced, which conceptually states that the converged electron density ρ is that of a reference density ρ_0 that is only slightly changed by $\Delta\rho$. The reference density ρ_0 is defined by the superposition of neutral atomic reference densities ρ_{A_0} : $\rho_0 = \sum_A^{N_{\text{atoms}}} \rho_{A_0}$. This motivates an expansion of the total energy in terms of density fluctuations $\delta\rho$ relative to ρ_0 , which gives an expression for $E[\rho]$:

$$E[\rho] = E^0[\rho_0] + E^1[\rho_0, \delta\rho] + E^2[\rho_0, (\delta\rho)^2] + E^3[\rho_0, (\delta\rho)^3] + \dots \quad (2.19)$$

with: $\rho = \rho_0 + \delta\rho$

This equation will give the exact KS-DFT energy, if the expansion is not truncated. Since the goal is to speed-up calculations through sensible approximations, this series is commonly truncated. The DFTB family of methods is directly classified by the highest order of $\delta\rho$ considered: DFTB1,^[18] DFTB2^[53] and DFTB3.^[19] However, these methods are based on element pair-wise parametrization, which makes the methods flexible in their application but also leads to an explosion in necessary parameters. This deficit in usability and element (combination) availability was addressed by Grimme and coworkers with their extended tight binding (xTB) methods.^[9,20,21] Primarily by replacing the parametrization imposed by the Slater-Koster integrals with an extended Hückel Hamiltonian & classical, short-range repulsive potential.

When choosing methods for a multilevel workflow (see fig. 1.1), the scaling of the computational cost is an important quantity to consider. For HF and DFT-theory, the component with the highest scaling is the computation of the ERIs. Without any approximations, the computational workload scales as the fourth power of the number of basis functions N_{bf} . In computer science, the so-called “big O” notation is used, i.e. HF scales as $(O(N_{\text{bf}}^4))$. Hence for a constant basis set, if the system size is doubled, such as going from C_{60} to C_{120} , the computational cost will be 16 times higher.

SQM methods are frequently used in exploratory stages of the multilevel workflow due to their low scaling. This relatively low scaling comes from a combination of strategies. On one hand, SQM methods frequently use terms with summations over atoms or over orbital shells instead of orbitals. These summation bounds are smaller than the number of basis functions, leading to a lower scaling. For many SQM methods, the computational component with the highest scaling is the diagonalization of

the Fock matrix, scaling $O(N_{\text{bf}}^3)$. This bottleneck also partially motivates the use of valence-only minimal basis sets for SQM methods. The errors introduced by the use of these reduced basis sets are partially remedied in the parametrization of these methods.

This strategy of deliberately lowering the basis set size and compensating the errors by fitting correction terms via empirical models, has been used by Grimme during the conception of the “3c”-methods. Starting with HF-3c^[24] in 2013, Grimme combined *ab-initio* electronic structure methods with specifically fitted correction terms. The correction terms are based on the dispersion correction models DFT-D3^[54,55] and DFT-D4^[56,57] and the geometric counterpoise correction^[58] model. The concept of the 3c-methods was extended to DFT, leading to a method for every level of DFA approximation: GGA: B97-3c,^[25] hybrid GGA: PBEh-3c,^[6] mGGA : r2-SCAN-3c^[10] and range-separated hybrid ω B97X-3c.^[11] Due to their favorable computational cost-accuracy ratio, they are often used in the second step of a multilevel workflow, where geometries are optimized on an *ab-initio* level.

2.1.4 GFN2-xTB

A family of semiempirical methods developed by Grimme and coworkers is called GFNn-xTB ($n \in 0, 1, 2$). This name was chosen to underline the special purpose nature, of this method which was fitted for geometries, frequencies and noncovalent interactions.^[9,20] In contrast, their latest xTB method is intended as a general-purpose method and is called g-xTB.^[59]

Slightly different from the original DFTB methods, the derivation of GFN-xTB methods, formally, starts from a DFT energy expression with the addition of a non-local dispersion term. The reference density is constructed via reference charges q_0^A for these methods. In the following section, we will be focusing on GFN2-xTB as the other GFN methods are not used extensively in this work. Regarding the parameterization strategy, GFN2-xTB uses element- and shell-specific parameters as well as some global parameters. This strategy leads to overall less parameters than would be needed for a DFTB3 model with the same coverage of the periodic table. We will structure our introduction to the GFN2-xTB method around the energy term of the method:

$$E_{\text{GFN2-xTB}} = E_{\text{rep}} + E_{\text{disp}} + E_{\text{EHT}} + E_{\text{IES+IXC}} + E_{\text{AES}} + E_{\text{AXC}} + G_{\text{Fermi}} \quad (2.20)$$

The first term is the repulsion energy E_{rep} , which is constructed as an atom pairwise potential using element-specific parameters Y_A^{eff} and α_A , and the general parameter k_{rep} .

$$E_{\text{rep}} = \sum_A^M \sum_B^M \frac{Y_A^{\text{eff}} Y_B^{\text{eff}}}{R_{AB}} e^{(\alpha_A \alpha_B)^{0.5} (R_{AB})^{k_{\text{rep}}}} \quad (2.21)$$

$$\text{where: } R_{AB} = |\mathbf{R}_A - \mathbf{R}_B| \quad (2.22)$$

The dispersion energy E_{disp} is modeled by the self-consistent version of the DFT-D4 model.^[56,57] E_{disp} can be separated in a two- and three-body term, the specific equations can be found in the original publication.^[9,56,57] The extended Hückel term E_{EHT} is mainly responsible for the attractive interaction leading to covalent bond formation. GFN2-xTB uses a minimal valence-only basis set constituted of atom-centered Gaussian orbitals χ_i . This small basis set and the frozen-core approximation also reduce the computational demands of the method. We show the construction of the diagonal Hamiltonian term $H_{\kappa\kappa}$ in eq. 2.24 to highlight the strategy of using shell-specific parameters H_A^l , $H_{CN_A}^l$, where l is the index of a shell on atom A . The latter parameter is scaled by a modified version of the D3 coordination number.^[54] Both shell-specific parameters, H_A^l and $H_{CN_A}^l$, are not precomputed by “ab-initio” calculations but are fitted during the parameterization step. The off-diagonal values $H_{\kappa\lambda}$ are computed based on the diagonal elements $H_{\kappa\kappa}$ and other quantities. The detailed form of this term can be found in the original publication.^[9] The construction of the EHT term allows the model to adapt to varying bonding situations without the need for pairwise parametrization.

$$E_{\text{EHT}} = \sum_i^N f_i \langle \psi_i | \hat{H}_0 | \psi_i \rangle = \sum_{\kappa}^{N_{\text{bf}}} \sum_{\kappa}^{N_{\text{bf}}} \sum_i^N f_i c_{\kappa i} c_{\lambda i} \langle \chi_{\kappa} | \hat{H}_0 | \chi_{\lambda} \rangle = \sum_{\kappa}^{N_{\text{bf}}} \sum_{\kappa}^{N_{\text{bf}}} D_{\kappa\lambda} H_{\kappa\lambda} \quad (2.23)$$

$$H_{\kappa\kappa} = H_A^l - H_{CN_A}^l C N_A' \quad (\kappa \in l \in A) \quad (2.24)$$

GFN2-xTB is the first method to include multipole interaction terms up to second order. The multipoles are derived from the Mulliken population analysis,^[60] giving the so-called cumulative atomic multipole moment (CAMP).^[61] Due to the frozen core approximation, the nuclear charge Z_A is corrected by the number of core electrons and gives the screened nuclear charge Z_A' , see eq. 2.25.

$$q_A = Z_A' - \sum_{\kappa \in A}^{N_{\text{bf}}} \sum_{\lambda}^{N_{\text{bf}}} D_{\kappa\lambda} \underbrace{\langle \phi_{\lambda} | \phi_{\kappa} \rangle}_{S_{\lambda\kappa}} \quad (2.25)$$

$$\mu_A^{\alpha} = \sum_{\kappa \in A}^{N_{\text{bf}}} \sum_{\lambda}^{N_{\text{bf}}} D_{\kappa\lambda} (\alpha_A S_{\lambda\kappa} - M_{\lambda\kappa}^{\alpha}) \quad (2.26)$$

$$\theta_A^{\alpha\beta} = \sum_{\kappa \in A}^{N_{\text{bf}}} \sum_{\lambda}^{N_{\text{bf}}} D_{\kappa\lambda} (\alpha_A M_{\lambda\kappa}^{\beta} + \beta_A M_{\lambda\kappa}^{\alpha} - \alpha_A \beta_A S_{\lambda\kappa} - M_{\lambda\kappa}^{\alpha\beta}) \quad (2.27)$$

The atomic dipole moment μ_A and atomic quadrupole moment θ_A are tensorial quantities, its components are specified using $\alpha, \beta \in x, y, z$. μ_A^{α} is computed as the sum of the product of dipole integral $M_{\lambda\kappa}^{\alpha}$ and density matrix $D_{\kappa\lambda}$, and the distribution of the atomic charges. In a similar fashion, $\theta_A^{\alpha\beta}$ is the sum of the charge distribution, the atomic dipole distribution and the product of quadrupole integral $M_{\lambda\kappa}^{\alpha\beta}$ and $D_{\kappa\lambda}$. Based on these multipole moments, the remaining terms $E_{\text{IES+IXC}}$, E_{AES} , and E_{AXC} can be defined. The isotropic part of this interaction is characterized by the shell-wise partial charges $q_{A,l}$ computed using eq. 2.25 with a summation over shells. An exchange-correlation type on-site term E_{IXC} can also be formulated and can be assigned to a

third order perturbation in $\delta\rho$.

$$E_{\text{IES+IXC}} = \frac{1}{2} \sum_{A,B} \sum_{l \in A} \sum_{l' \in B} q_{A,l} q_{B,l'} \gamma_{AB,ll'} + \frac{1}{3} \sum_A \sum_{l \in A} \Gamma_{A,l} q_{A,l}^3 \quad (2.28)$$

The term $\gamma_{AB,ll'}$ is computed using a generalized form of the Ohno-Klopman formula.^[9] It depends on the chemical hardness of the corresponding shells and elements and their interatomic distance to allow damping of the Coulomb-type interaction. The third-order term uses another quantity $\Gamma_{A,l}$, which is computed based on shell- and element-specific parameters.

GFN2-xTB improves upon the isotropic description of interaction charges by using the CAMMs to model anisotropic effects as well.^[9] The anisotropic electrostatic interaction E_{AES} is a sum of the monopole-dipole $E_{q\mu}$, monopole-quadrupole $E_{q\theta}$, and dipole-dipole $E_{\mu\mu}$ interactions. Damping functions $f_m(R_{AB})$ ($m \in 3, 5$) are used to describe the distance dependence of those interactions. Similar, to the isotropic counterpart E_{AXC} is an on-site correction, see eq. 2.30.

$$E_{\text{AES}} = E_{q\mu} + E_{q\theta} + E_{\mu\mu} = \frac{1}{2} \sum_A^M \sum_A^M \left\{ f_3(R_{AB}) \left[q_A (\boldsymbol{\mu}_B^T \mathbf{R}_{BA}) + q_B (\boldsymbol{\mu}_A^T \mathbf{R}_{AB}) \right] \right. \\ \left. + f_5(R_{AB}) \left[q_A \mathbf{R}_{AB}^T \boldsymbol{\Theta}_B \mathbf{R}_{AB} + q_B \mathbf{R}_{AB}^T \boldsymbol{\Theta}_A \mathbf{R}_{AB} - 3 (\boldsymbol{\mu}_A^T \mathbf{R}_{AB}) (\boldsymbol{\mu}_B^T \mathbf{R}_{AB}) + (\boldsymbol{\mu}_A^T \boldsymbol{\mu}_B) R_{AB}^2 \right] \right\} \quad (2.29)$$

$$E_{\text{AXC}} = \sum_A^M \left(f_{\text{XC}}^{\mu_A} |\boldsymbol{\mu}_A|^2 + f_{\text{XC}}^{\Theta_A} \|\boldsymbol{\Theta}_A\|^2 \right) \quad (2.30)$$

The traceless quadrupole moment $\boldsymbol{\Theta}_A$ for atom A is given by:

$$\Theta_A^{\alpha\beta} = \frac{3}{2} \theta_A^{\alpha\beta} - \frac{\delta_{\alpha\beta}}{2} (\theta_A^{xx} + \theta_A^{yy} + \theta_A^{zz}) \quad (2.31)$$

$f_{\text{XC}}^{\mu_A}$ and $f_{\text{XC}}^{\Theta_A}$ in eq. 2.30 are element-specific fit parameters.

In eq. 2.20, G_{Fermi} is the entropic contribution to the energy term due to Fermi smearing.

2.1.5 SCF procedure in GFN2-xTB

Based on the previous discussion, we can see that GFN2-xTB needs to be solved in a self-consistent manner. However, all self-consistent terms depend on the multipole moments. Hence, our optimization problem shifts from the coefficients of the atomic orbitals (SCF) to that of shell-wise multipole moments. This process is therefore called self-consistent charge (SCC). Commonly, for the SCF procedure elaborate acceleration techniques are used to reduce the number of cycles needed to reach convergence.

One of the most prominent acceleration schemes is the direct inversion of the iterative subspace (DIIS) method.^[62] For GFN2-xTB and the SCC, the use of the comparatively simple Broyden^[63] method has proven very successful and compute time efficient. In alg. 2.1, a resume of the most important steps in the SCC loop are given.

Alg. 2.1: Pseudocode presenting the important steps involved in a direct SCF code.

```

iscf = 0
while (.not. converged .and. iscf < max_cycles)
  ! the first cycle uses the guess charges
  if (iscf > 0) then
    call broyden_next() ! generate next multipoles
    call broyden_get_charges() ! return the computed multipoles
  end if

  ! Compute self-consistent contributions to the Fock matrix
  call get_potential()
  call add_pot_to_hcore()
  ! store current charges in broyden
  call broyden_set_charges()
  ! generate new density by diagonalization
  call get_next_density()
  ! compute new partial charges, dipole and quadrupole moments
  based on new density
  call compute_charges_dipole_quadrupole()
  ! store difference in multipoles
  call broyden_diff_charges()
  ! compute all energy contributions
  call compute_energies()
  iscf = iscf + 1
end while

```

Similar to the SCF procedure of ab-initio methods, the potential of the first SCC cycle is computed based on guess charges. The self-consistent part of the Fock matrix is added to the extended Hückel Hamiltonian to build the full Fock matrix. The values of the charges are stored in the Broyden mixer. Solving the generalized eigenvalue problem by means of diagonalization gives a coefficient matrix and the orbital energies. GFN2-xTB uses Fermi-smearing^[64] to allow the fractional occupation of the MO orbitals. This technique has proven useful as GFN2-xTB generally features rather small gaps that could lead to convergence issues especially for unrelaxed geometries. Due to the fractional occupation, the resulting density matrices may no longer fulfill the idempotency condition: $\mathbf{P} = \mathbf{P}^2$ (where $\mathbf{P}$ is the density matrix in the orthogonal basis). Based on the new density matrix, new multipole moments can be computed using eqs. 2.25, 2.26, and 2.27. Newton-Raphson-based acceleration schemes, like Broyden, use the gradient to compute the values of a new iteration. This gradient is computed by the `broyden_diff_charges` function. In a last step, the energy of this cycle is computed. If the convergence threshold has not been reached, the new cycle will start with the generation of new charges based on the previous cycle's gradient.

2.2 GPU hardware and programming using the Compute Unified Device Architecture (CUDA) model

During the advent of personal computers, the need for processors designed to efficiently compute graphics rose. The graphics workload essentially consists of computing the pixel values of an image with a certain resolution. This resolution defines a matrix of pixels that has to be computed using similar or the same instructions for all data points. Hence, the GPU must be good at applying one instruction to many data units, referred to as SIMD, see fig. 2.1b. In contrast, CPUs have been optimized to be versatile in the kind of instructions that they are handling, see fig. 2.1a. Therefore, in GPUs more transistors have been dedicated to data manipulation compared to CPUs.^[65]

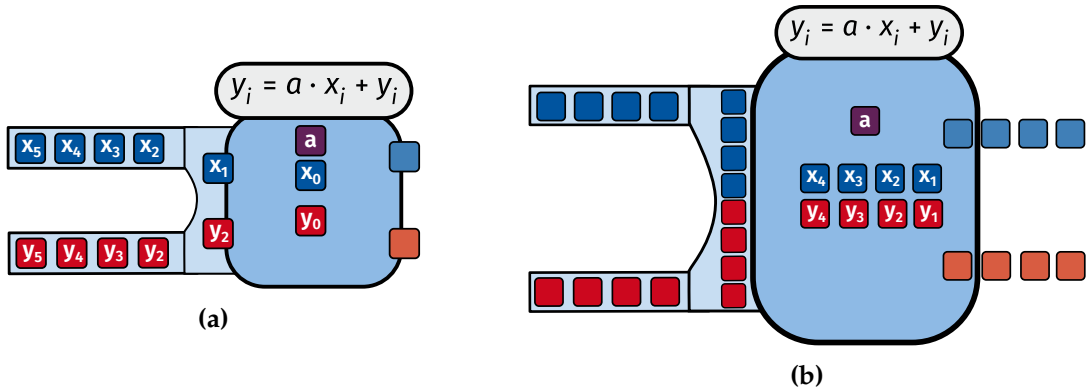


Fig. 2.1: (a) Schematic illustration of a processor applying an instruction to two data pairs in a serial fashion. (b) Schematic illustration of a processor applying a single instruction to multiple data pairs in a SIMD fashion.

In 2007, NVIDIA published the CUDA software package, allowing developers to address the vast computing resources of GPUs directly.^[65] CUDA also introduced a programming model that is based on the hardware found on their devices and was created with scalability in mind. In figure 2.2, we show a schematic representation of the CUDA programming model with its various layers of organization and their relation to hardware. The lowest abstraction unit is a thread, which corresponds to addressing a single CUDA core. A group of 32 threads are organized in a warp, which are scheduled together. A set of warps is then combined into a thread block, which are executed on one streaming multiprocessor (SM). Threads in one thread block can cooperate easily using the fast shared memory pool. Additionally, also synchronization between the threads of a block or warp can be forced through CUDA. An SM refers to a collection of processing cores, various memory levels, and schedulers. For each generation of GPUs, usually the SM architecture remains constant and GPU models differ in the number of SMs that are available. The highest level of abstraction is the grid, which is defined as the number of thread blocks that are executed on a device.

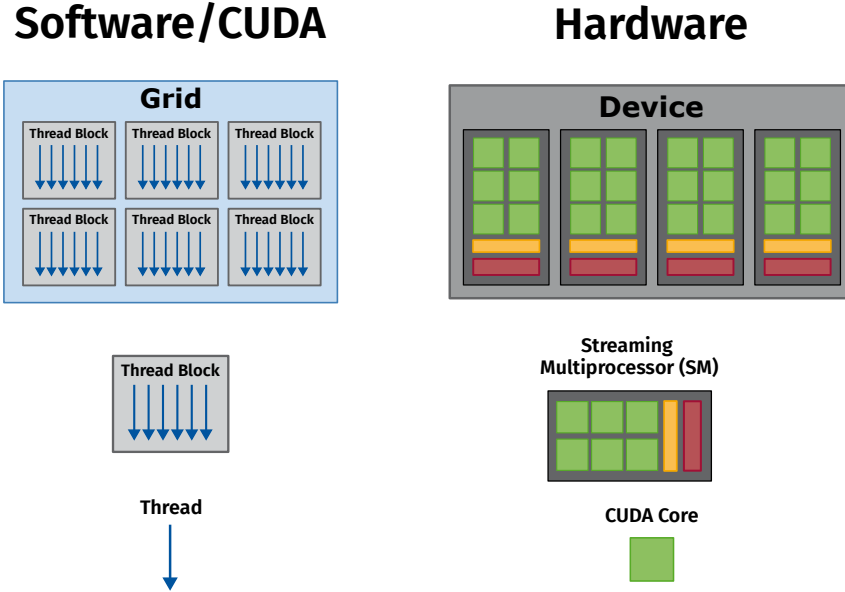


Fig. 2.2: Representation of the three abstraction layers introduced by CUDA in the programming model. The corresponding hardware and software unit are shown on a horizontal level.

In the context of GPUs, we refer to a function that is executed on the GPU as *kernel*. Upon calling a kernel, we have to provide the configuration of thread block size and grid size used to execute the kernel. This configuration should be chosen in a way to cover all data that should be manipulated by a kernel, in the sense that $\text{threadBlockSize} \times \text{gridSize} \geq n_{\text{data}}$. The thread blocks are then launched in waves to fill as many SMs as available. Once a thread block is completed, another one is launched until the grid has been cleared.

As previously indicated, the data type of operations and quantities is more important in GPU programming than in CPU programming. The SMs are built with processing cores that are specific to a data type in contrast to a CPU that is more versatile. Therefore, the presence or absence of these compute cores has a crucial impact on the performance for that data type. The GPUs that have been designed for graphical simulations will be referred to as consumer grade GPUs (CG-GPUs) in this work. For the workload of rendering graphics, the representation of real numbers as 32 bit floating point numbers is sufficient. Hence, processing cores for larger floating point representations (64-bit) have not been included in their hardware design. The CUDA platform allows the computation of data with that representation. Since the corresponding specific cores are not present in CG-GPU SMs, this leads to a performance penalty. Strategies to address this deficit with clever numerical algorithms will be discussed after introducing the challenges of implementing the SCF method on GPUs in the next chapter.

3 Introduction into GPU implementation in quantum chemistry

In the previous section [2.1.1](#) on the Hartree-Fock theory, we briefly introduced the Coulomb and exchange terms which depend on four orbital indices. These two terms can be summarized under the name electron repulsion integrals (ERIs) and represent the computational bottleneck in HF calculations and other theories that explicitly compute two-electron integrals. As discussed before, the hardware architecture of GPUs influences software design and algorithmic development. The challenges related to an ERI GPU implementation will be the topic of the upcoming section. We will present the main challenges related to this field and its implementation on GPUs.

An aspect of great importance in GPU computations is the numerical representation of real numbers and its implications on the computing performance of these devices. Hence, we will shortly give an introduction on standardized floating point representations of real numbers. We will then discuss strategies that aim at reducing the need for large floating point representations, while preserving the numerical accuracy.

3.1 Computation of Electron repulsion integrals on GPUs

Let us revisit the structure of ERIs. We take the Coulomb term in eq. [3.1](#) in the AO basis as an example.

$$(\mu\nu|\kappa\lambda) = \iint \chi_\mu(\mathbf{r}_1)\chi_\nu(\mathbf{r}_1) \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \chi_\kappa(\mathbf{r}_2)\chi_\lambda(\mathbf{r}_2) d\mathbf{r}_1 d\mathbf{r}_2 \quad (3.1)$$

The atomic orbitals χ_μ are modeled in the vast majority of implementations by atom-centered Gaussian functions. Based on the above Coulomb term, we can see that the formation of products such as $\chi_\mu(\mathbf{r}_1)\chi_\nu(\mathbf{r}_1)$ is an elementary step. The regrouping of atomic orbitals $\chi_\mu^l(\mathbf{r})$ in terms of their shell or angular momentum l is often used in ERI evaluation schemes. This is motivated by the fact that the complexity of ERI computations is mainly dependent on the involved angular momenta. The pair $\chi_\mu(\mathbf{r}_1)\chi_\nu(\mathbf{r}_1)$ of Gaussians can be computed using the position vectors and exponents of both Gaussian

functions using the Gaussian product theorem (GPT). This also marks the advantage of using Gaussian functions for χ . However, it is hard to describe the cusp of the wavefunction expected for $\mathbf{r} = (0.0, 0.0, 0.0)$ with a single Gaussian function. Hence, "primitive" Gaussian functions $\phi_{\mu,a}^l(\mathbf{r})$ are used to form "contracted" Gaussians $\chi_\mu^l(\mathbf{r})$ according to eq. 3.2.

$$\begin{aligned}\chi_\mu^l(\mathbf{r}) &= \sum_a^{K_\mu} k_a \phi_{\mu,a}^l(\mathbf{r}) \\ &= \sum_a^{K_\mu} k_a (x - A_x)^{l_x} (y - A_y)^{l_y} (z - A_z)^{l_z} e^{-\lambda_a |\mathbf{r} - \mathbf{A}|^2} = |\mu\rangle\end{aligned}\quad (3.2)$$

The number of primitive functions K_μ is also referred to as the degree of contraction. Each primitive Gaussian is characterized by its contraction coefficient k_a , the center of the function $\mathbf{A} = (A_x, A_y, A_z)$, the angular momentum vector $\mathbf{l} = (l_x, l_y, l_z)$ and the exponent λ_a .

After establishing these fundamental quantities, we will shortly introduce the general algorithmic workflow used to solve the electronic structure. Barca *et al.*^[66] have given a very good overview on the challenges dominating a (multi-)GPU implementation of an ERI code. We will base our discussion on a similar structure, restricting ourselves to single GPU implementations.

In eq. 2.13, we introduced the Roothaan-Hall equation, which is solved iteratively to compute the electronic structure. The pseudocode in alg. 3.1 shows how the SCF procedure is generally implemented in computer programs. The variant, where the ERI integrals are recomputed on every iteration, is referred to as direct SCF procedure and was introduced by Almlöf.^[67] A common strategy is to sort basis functions into shells with constant total angular momentum. For simplicity, we will focus on contracted Gaussian shells $|m\rangle$, which refers to a set of p contracted Gaussians $|m\rangle = |\mu_1\rangle, \dots, |\mu_p\rangle$ with the same total angular momentum and the same primitive Gaussian functions.

The shells are then combined into shell pairs by taking the tensor product and using the GPT. A side effect of the GPT is that the numerical value of products of Gaussian type basis functions decreases with the distance between the nuclei. Hence, a first screening of significant shell pairs can be performed via the GPT prefactor. Based on the significant shell pairs $|mn\rangle$, the overlap $S_{\mu\nu}$ and core Hamiltonian $h_{\mu\nu}$ elements can be computed based on the primitive Gaussians $|\mu\nu\rangle$. To compute the ERIs contributions to the Fock matrix $F_{\mu\nu}$, a density matrix $\mathbf{D}$ is needed. A guess density $\mathbf{D}_0$ can be formed by diagonalizing the core Hamiltonian for example. The workflow then proceeds to compute the ERIs. Without any approximations this step scales as N_{bf} to the power of four, where N_{bf} is the number of basis functions. If the molecules become large enough, this scaling is effectively reduced to $O(N_{\text{bf}}^2)$ when screening based on Gaussian-type orbitals is used.^[68] Hence, an effective screening procedure is needed to limit the amount of integrals that are computed. The remaining significant ERIs are computed using the recurrence relations of Cartesian Gaussian functions.

Alg. 3.1: Pseudocode presenting the important steps involved in a restricted direct SCF code.

```

Form list of shells  $|m\rangle$ ;
Screening 1: form list of significant shell pairs  $|mn\rangle$ ;
for all  $|mn\rangle$ 
    for all  $|\mu\nu\rangle \in |mn\rangle$ 
        Compute  $S_{\mu\nu}$ ;
        Compute  $h_{\mu\nu}$ ;
    end
end
Initial Guess  $D_0$ ;
repeat
     $F = h$ ;
    ComputeERI( $D, F, |mn\rangle$ );
    Generate Density  $D$  from  $F$ ;
    Check convergence;
until converged;

function ComputeERI( $D, F, |mn\rangle$ ):
for all  $|mn\rangle$ 
    for all  $|\mu\nu\rangle \in |mn\rangle$ 
        for all  $|kl\rangle$ 
            for all  $|\kappa\lambda\rangle \in |kl\rangle$ 
                Screening 2: Skip computation of all insignificant ERIs;
                Compute ERI:  $(\mu\nu|\kappa\lambda) \in (mn|kl)$ ;
                 $F_{\mu\nu} + = \sum_{\kappa\lambda} D_{\kappa\lambda}[(\mu\nu|\kappa\lambda)]$ 
            end
        end
    end
end
end

```

The first challenge for GPU implementation arises from the fact that the computation of ERIs is an intrinsically inhomogeneous computational workload. For example, the evaluation of an $(ss|ss)$ type integral needs way less operations than that of a $(sp|pp)$ integral. Computing these $(ss|ss)$ and $(sp|pp)$ integrals together in one thread block would be very inefficient as the execution time would be determined by the integral with higher angular momentum. This would obstruct the execution of further thread blocks and lead to a low GPU utilization. Therefore, a scheme needs to be used to sort ERIs into batches of integrals with similar computation time requirements. Such a scheme has been introduced already in 2008 by Ufimtsev and Martinez.^[69] The shell quartets are ordered according to their total angular momentum and the total degree of contraction.^[66] This allows uniform batches of ERI evaluation, which ensures that the divergence between threads is minimized. Additionally, the batches are ordered according to their expected numerical value.

A strict upper bound to the numerical value of ERIs is also needed to perform the screening and minimize the computation of numerically irrelevant integrals. The Cauchy-Schwarz inequality is the most commonly used upper bound for ERI screening. Various implementations also use the product of the ERI and the density matrix elements in the screening process.^[69–72] This is especially efficient for large molecules as $\mathbf{D}$ has been proven to decay exponentially with distance, making it increasingly

sparse for large molecules.^[73,74] The value for D_{kl} is chosen by taking the maximum density matrix element for the combinations within the shell-shell pair $|kl\rangle$. Similarly, the factor I_{mn} is also determined by taking the largest numerical value of the square root of the integral $(\mu\nu|\mu\nu)$ from the shell-shell pair combinations $|mn\rangle$.

$$|(mn|kl)| \leq I_{mn}I_{kl} \text{ or: } |(mn|kl)D_{kl}| \leq I_{mn}I_{kl}|\max_{\kappa,\lambda \in |kl\rangle}(D_{\kappa\lambda})| \quad (3.3)$$

$$\text{where: } I_{mn} = \max_{\mu,\nu \in |mn\rangle}(\mu\nu|\mu\nu)^{\frac{1}{2}} \quad (3.4)$$

The computation of I_{mn} also allows the previously mentioned sorting of integral batches according to their expected numerical value. The implementation by Ufimtsev and Martinez builds the exchange and Coulomb contributions to the Fock Matrix separately,^[69,71] as found in the software suite TeraChem.^[75,76] In the case of exchange builds, the ERI is contracted against density matrix elements situated on two distinct atoms, thereby its numerical value decreases faster than the Coulomb part. A separate build of both contributions means that numerically insignificant ERIs can be screened more effectively.^[70] Depending on the ERI evaluation scheme used, the screening is performed on a primitive or contracted Gaussian level.

Choosing an ERI evaluation scheme that reduces the number of floating point operations is a common challenge to CPU and GPU implementations. There are four main schemes currently in use for this procedure: Obara-Saika,^[77,78] McMurchie-Davidson (MMD),^[79] Rys-Quadrature^[80,81] and Head-Gordon-Pople (HGP).^[82] To our knowledge, all but Obara-Saika have been used in GPU-based ERI codes. However, this scheme is the basis for the Head-Gordon-Pople scheme as it uses the recurrence relations of Obara-Saika while adding more recurrence relations. The McMurchie-Davidson algorithm is based on expressing the integrals using Hermite functions. Lastly, in the Rys-quadrature scheme, the integral is modeled using a numerical quadrature based on Rys-polynomials. This scheme is known to perform very efficiently for high angular momenta as less intermediate quantities are needed.^[69,83,84] This is especially relevant for GPU implementations, since their global memory pool is usually smaller than that of the CPU main memory. Another relevant consideration in GPU implementations is that the reuse of common intermediates might involve synchronization steps that can also deteriorate the computational efficiency. These hardware limitations need to be considered when using code generators to write optimized GPU kernels for various ERI combinations.

The computation of the ERI contribution to the Fock matrix can in theory also be reduced by exploiting the 8 fold permutation symmetry of these integrals. However, certain implementations sacrifice the use of this symmetry relationship as it would lead to necessary synchronization steps or non optimal memory access patterns.^[69,71] In 2021, Barca *et al.*^[66] managed to incorporate full symmetry. In that publication, the HGP scheme is used for ERI evaluation. The recently published GPU4pySCF package uses Rys-quadrature for all ERI evaluations.^[83] TeraChem uses a MMD-based implementation.^[85]

The publication of GPU-based SCF codes or ERI computation schemes can be loosely separated into two generations. The early works after the introduction of the CUDA

software stack.^[69,86–89] After those initial contributions, further work was dedicated in refining those strategies.^[66,73,84,85,90–93] Various software implementations also reported multi-GPU implementations for ERI evaluation.^[66,85,94–97] For the most recent comparison of the fastest GPU implementations consider the following ref. [85], this work also presents optimized versions of Ufimtsev’s and Barca’s ERI implementations.

However, most recent GPU implementations use 64-bit floating point number arithmetics, thereby targeting expensive high performance GPUs. In 2011 Luehr *et al.*, proposed an algorithm that minimizes the number of operations done in 64-bit precision without sacrificing the overall numerical precision.^[72] In the following section, we will discuss the design strategies behind this implementation and other strategies involving multiple numerical precisions.

3.2 Floating point representations and mixed precision strategies

Fractional and real numbers can only be represented with limited precision on computers. Their standard representation on modern computers is that of floating point numbers as defined by various IEEE standards.^[31,32] The following equation can be used to transform between the binary form of a floating point number $b_0, \dots, b_{n_{\text{tot}}}$ and its normal numerical value r .

$$r = (-1)^{b_{n_{\text{tot}}-1}} \cdot 2^{E-B} \cdot \left(1 + \sum_{i=1}^{n_{\text{significant}}} b_{n_{\text{significant}}-i} \cdot 2^{-i} \right) \quad (3.5)$$

$$\text{where: } E = \sum_{j=0}^{n_{\text{exponent}}-1} b_{n_{\text{significant}}+j} \cdot 2^j$$

The floating point representation is composed of a single sign bit, n_{exponent} exponent bits and $n_{\text{significant}}$ bits. The standard defines the number of exponent bits and significant bits, and also the exponent bias B which is an integer value. In fig. 3.1, the bit composition for the double precision FP64, single precision FP32 and half precision FP16 is represented.

We can see that the number of bits in the significand decreases faster than the number of exponent bits. The FP64 standard allows to store 15-16 significant digits, while the FP32 standard allows 7-8 significant digits. Due to the above shown construction, the highest resolution for floating point numbers can be found around $r = 1.0$.

In quantum chemistry, we compute electronic energies as a result of the Schrödinger equation. However, the absolute values are less relevant in a chemical context as they can not be compared to an experiment and are usually implementation and method specific. The difference in energy can be directly compared with experimentally accessible quantities. It is convenient to compute the energy using atomic units, which results in the unit of Hartree for the energy. Commonly, differences in energy are on

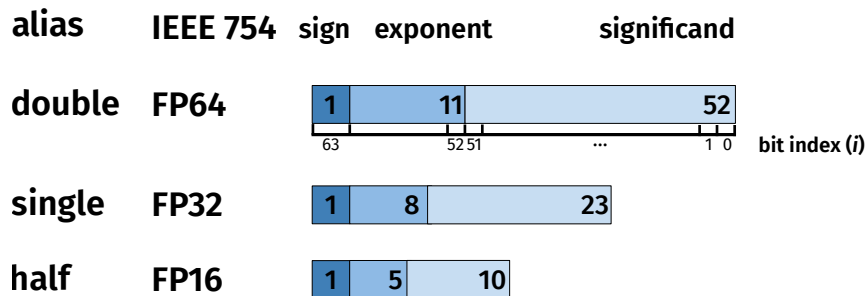


Fig. 3.1: Bit-wise composition of floating point representations FP64, FP32, and FP16 according to IEEE standard 754.^[31,32] The length of each bit group is not chosen in proportion to its number of bits. The index i of bits b , used to construct the decimal number according to eq. 3.5, is shown for the FP64 standard. Next to the standard name we also give the frequently use alias: double, single, half precision.

the scale of milli Hartree (mH), 1 mH is equal to 2.6 kJ/mol or 0.62 kcal/mol. Common energy thresholds for the SCF procedure, are on the order of $10^{-6} - 10^{-8}$ H. For reference, the total energy for a small molecule like water using the PBEh-3c^[6] composite method is -74.6600 H or -3.2890 H using GFN2-xTB. A larger molecule like Buckminster fullerene (C_{60}) has total energies of -2281.1041 H and 128.3255 H for PBEh-3c and GFN2-xTB, respectively. These numerical requirements mean that in quantum chemistry most if not all calculations are performed in FP64. For implementations on CPUs or high-performance general purpose GPUs, the penalty for using FP64 is a factor of 2 for both computation time and memory. Therefore, FP64 is still the “de facto” standard, although some work has been published recently investigating the need for FP64 in very strong scaling theories such as coupled cluster.^[98] However, when trying to use the more affordable CG-GPUs, the penalty for using FP64 instead of FP32 is commonly a factor of 64, see also tab. 1.1.^[30] When trying to reduce FP64 usage, there are two main strategies: separating the computation of large and small numerical values or combining multiple lower precision representations to compute numerically similar values to a high precision representation. In general, we refer to mixed precision applications when more than one numerical precision is used to represent or compute values.

The goal of reducing FP64 computations can be achieved more readily by using an incremental Fock matrix build as proposed by Häser and Ahlrichs^[70]. The idea is to compute and then store the full Fock matrix for a reference iteration cycle $\mathbf{F}_{\text{ref}}$ and the density matrix $\mathbf{D}_{\text{ref}}$ used to form that Fock matrix. For every iteration i after that reference, the Fock matrix is incremented $\mathbf{F}_i = \mathbf{F}_{\text{ref}} + \Delta\mathbf{F}_i$. $\Delta\mathbf{F}_i$ is computed by contracting the ERIs against the difference density matrix $\Delta\mathbf{D}_i = \mathbf{D}_i - \mathbf{D}_{\text{ref}}$. The choice of the reference cycle and the number of cycles after which a full rebuild of the Fock matrix is performed are specific to each implementation. In eq. 3.3, we briefly mentioned that some screening implementations use a density-weighted version of the Cauchy-Schwarz inequality. In that case and when using an incremental Fock build, the number of numerically significant ERIs is decreasing after the full build, leading to fewer integral evaluations. In general, the numerical value of the increments is smaller

than that of the full Fock matrix, paving the way for the use of reduced numerical precision.

Coming back to GPU implementations, Luehr *et al.*^[72] proposed a mixed and dynamic precision runmode for the TeraChem codebase. Their work introduces an additional numerical threshold $\epsilon_{\text{FP64-FP32}}$ to be used in the Schwarz screening next to ϵ_{screen} , see fig. 3.2. The latter value defines the cutoff for ERI computation. All J and K-matrix elements with a density-weighted upper bound between $\epsilon_{\text{FP64-FP32}}$ and ϵ_{screen} are computed using FP32 arithmetics. Values with an upper bound above $\epsilon_{\text{FP64-FP32}}$ are computed in FP64. To be more efficient this scheme is combined with an incremental Fock build, reducing the number of elements evaluated in FP64 for later cycles. In fig. 3.2, we schematically show how ERI elements are sorted according to their Cauchy-Schwarz upper bound value. The technique of building numerically large quantities using an incremental approach can be transferred to other applications as we will demonstrate in ch. 5.

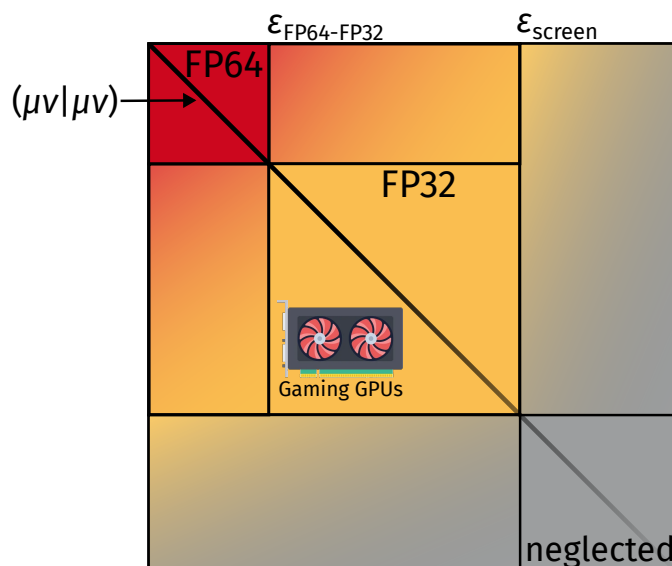


Fig. 3.2: Schematic graphical representation of ERI values sorted by numerical value. The numerically largest ERIs are computed in FP64 and are colored in red. ERIs below ϵ_{screen} are neglected and not computed at all. Integrals with values between $\epsilon_{\text{FP64-FP32}}$ and ϵ_{screen} are computed using FP32 operations. Weighting with the density matrix, which is not shown here, further reduces the number of ERI computations. **The following figure has been previously published in Ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

3.3 Half-precision splitting schemes

In 2017, NVIDIA introduced a specialized processing core named “Tensor Core” as part of the V100 GPU. This core can perform fused multiply-add operations on tiles of matrices. That operation is the key arithmetic step in matrix-matrix multiplications for example. The first generation tensor cores could perform operations on 4 by 4

tiles, with input tensors up to FP16 and output tensors up to FP32. NVIDIA reports a relative speed-up of factor 8 for these tensor cores compared to peak FP32 performance on the V100 platform.^[100] This speed-up is even more pronounced for newer hardware.^[101] The recent development of GPU hardware has been driven by optimizing the performance for artificial intelligence applications, who benefit greatly from reduced precision arithmetics. This hardware-based trend, has lead to various mixed precision schemes, where a high precision tensor is split into multiple lower precision tensors.^[102–104] We will be focusing on two schemes that have also been published for chemical applications. The Markidis^[102] scheme focusses on the splitting of an FP32 tensor A into two FP16 tensors $A^{(h)}, A^{(l)}$ according to eq. 3.6. The $\text{fl}_{\text{FPX}}(Y)$ operator refers to casting the tensor Y into the floating precision representation FPX.

$$\begin{aligned} A^{(h)} &= \text{fl}_{\text{FP16}}(A) \\ A^{(l)} &= \text{fl}_{\text{FP16}}(A - \text{fl}_{\text{FP32}}(A^{(h)})) \end{aligned} \tag{3.6}$$

The most significant part of the FP32 number is stored in $A^{(h)}$, whereas the difference between this reduced representation and the actual FP32 value is stored in $A^{(l)}$. This scheme has been used by Niklasson, Finkelstein and their coworkers in various numerical algorithms relevant to chemical applications.^[105–108] Another scheme that has also been applied to chemistry^[101] more recently is the Ozaki scheme.^[103,104] The Ozaki scheme has a more general approach that can be extended to other floating precisions as well, most relevant for this work to FP64. The splitting function used in this scheme is more elaborate than that used in the Markidis scheme and allows in principle to replicate the results of high-precision matrix-matrix multiplications exactly.

In general, there are three main stages to those FP16 splitting schemes as displayed in fig. 3.3. In the first step, the high precision matrix is split into a series of lower precision matrices. For the aforementioned Markidis scheme, we directly apply the conversion from high to low precision to the elements of our matrix, see eq. 3.6. This procedure is straightforward and easy to implement. However, if the numerical values to be computed are outside of the exponent range that can be represented by half precision ($10^{-14} - 10^{15}$), this scheme fails.

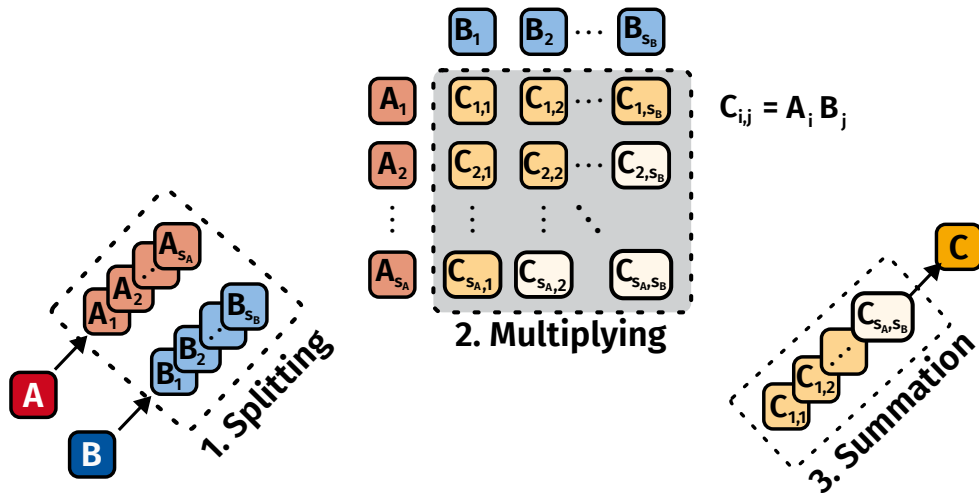


Fig. 3.3: Schematic drawing showing how the input matrices **A** and **B** are split s_A - and s_B -times into FP16 matrices $A_1 \cdots A_{s_A}$ and $B_1 \cdots B_{s_B}$ respectively. The combination of those split-matrices gives then at most $s_A \cdot s_B$ GEMMs to be computed. Strategies to reduce the number of GEMMs by neglecting the light orange combinations for example are also in use.^[101] Special mixed precision GEMMs are used, that take FP16 input matrices ($A_i B_j$) and accumulate the products in FP32. The resulting FP32 matrices $C_{i,j}$ are then summed up at the desired final precision to give **C**.

4 LAHVA: Linear algebra on heterogeneous/vectorized architecture - a library to simplify GPU acceleration

The contents of this chapter have not been published in a journal at the time of writing. However, we intend to publish this work in the future. The majority of code that is presented in this work has been written by the author of this thesis. Mark Heezen has contributed to the minor parts to this project: the fortran package manager build system, the addition the `IndexMaxFromVector` and `FrobeniusInnerProduct` functions. The conceptualization, the vast majority of code writing, software design, and documentation writing were carried out fully by the author of this thesis.

4.1 Motivation and library functionality

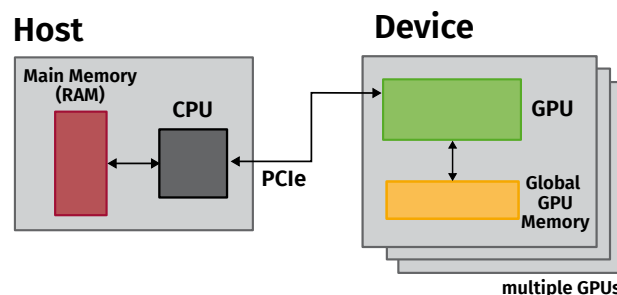


Fig. 4.1: Schematic drawing of a heterogeneous computing device, focussing on the distinction of a host and device system and its interconnect. The host system is composed of one CPU and its random access memory (RAM) and is connected to a device via a peripheral component interconnect express (PCIe) bus. A host system can be connected to multiple devices.

In the previous chapter, the challenges, to be considered when implementing an SCF program on a GPU, have been presented. These additional considerations are partially related to the hardware of the heterogeneous computing system (see fig. 4.1) and its software environment, see fig. 2.2. For example, the CPU system, often referred to as host, has a different memory pool than the GPU (device). Therefore, in order to pass

data from host to device, the developer first needs to allocate memory in the device memory pool (in yellow). Then the information contained in the host memory (red) range is copied to the device code. The CUDA programming environment exposes an application programming interface (API) to the developer that implements the allocation of memory on the device and host, as well as copying between both.

Although, NVIDIA offers an extensive documentation platform and some tutorials, developing a GPU-accelerated application comes with a high initial hurdle for developers. This time investment is especially considerable for researchers without any formal training in computer science. Programming languages like Python offer user-friendly interfaces to the CUDA API as well as easy to use objects like tensors, matrices, and other linear algebra classes.^[109] However, the performance of these python implementations comes from the use of C/C++ libraries as backend. Friede *et al.*^[110] have shown that their implementation of the GFN1-xTB Hamiltonian using the pytorch module^[109] is still slower than the FORTRAN implementation in tblite.^[111] Although, a FORTRAN API to CUDA exists through a custom compiler, this compiler often struggles to compile newer features of the language in our testings. The most comprehensive access to the CUDA programming environment is given through C++. Additionally, C++ is a performance language in which numerous quantum chemical packages are written.^[75,76,112-115]

Based on these considerations, our goal is to develop a library that hides the cumbersome CUDA API and exposes easy-to-use C++ objects to the developers. The library focusses on linear algebra operations as standardized by the BLAS system.^[116-118] The formulation of models in terms of BLAS operations is generally seen as good practice, because hardware vendors usually publish highly optimized BLAS implementations and update them for new hardware. Similarly, NVIDIA has also published their implementation of the BLAS functions called cuBLAS.^[119] Hence, where possible we will use the API to cuBLAS and to a CPU BLAS library to optimize performance. The target user for the LAHVA library are researchers that have some previous knowledge in C++ programming but little in GPU programming.

Offloading calculations to a GPU or any accelerator connected to the host via PCIe bus or other protocols comes with an overhead. For NVIDIA GPUs, this overhead is formed by the setup time of the CUDA runtime (ca. 100 ms), the allocation and transfer of data from host to device and back. The consequence is that offloading calculations to the GPU is not always beneficial for the overall runtime. The problem size for which this investment pays off depends on the workload and the hardware in use, but in general the GPU favors larger system sizes. Not only for small problem sizes but also for timing purposes, it is desirable to have a CPU-only implementation along the GPU-accelerated version. Hence, another key proposition of our library is easy switching between CPU and GPU BLAS operations based on a single template parameter of an encompassing class.

As discussed in sec. 3.2, choosing the right floating point precision for a task is important to optimize the performance in GPU applications. Therefore, we use the template feature of the C++ language to create tensor objects with varying data types. As a result of mixed-precision implementations, kernels that perform conversion or BLAS-like operations need to be written by the developer. A collection of such operations are also provided by LAHVA. Lastly, mixed precision schemes such as the Ozaki scheme should be available to developers without exposing the underlying splitting and merging kernels.

So far, the GPU has been introduced as a tool excelling at tasks that require data parallelism, referring to the distribution of data among many processors executing a single function. In contrast to this form of parallelism, task parallelism describes the concurrent execution of potentially varying tasks on multiple processors. For task parallelism, on the other hand, we want to have tasks that are mutually independent of each other. The CUDA programming model also enables the use of task parallelism through multiple schemes. Recent GP-GPUs allow for the GPU to be partitioned into multiple independent entities. However, this feature is not found on CG-GPUs or older models. Therefore, we are relying on CUDA's "stream"-based task parallelism, which is again hidden to the end user of LAHVA.

Before discussing similar projects that try to simplify the use of GPUs, we want to summarize the functionality that we want to include in LAHVA:

- Implement tensor classes, that hide pointer semantics where possible, and use templates for data type flexibility.
- Facilitate switching between the execution of operations on the CPU and GPU, resulting in a high degree of code reusability.
- Provide special kernels for mixed-precision implementations and provide splitting schemes down to half precision.
- Allow asynchronous concurrent calculations

4.2 Introduction

The idea of separating linear algebra objects and operations from the actual implementation of an electronic structure code or a solver library has been proposed several times before.^[120–123] This strategy aligns with the current trend of building more reusable and modular software in quantum chemistry and adjacent fields.^[124] The advantage of modular libraries in contrast to large monolithic codebases is the ease of implementing new algorithms and maintaining the current codebase. Additionally,

computing hardware is constantly evolving. This creates a challenge that can be tackled effectively by using a linear algebra library that exposes a common and constant interface to application code.

The basic matrix library (BML)^[120,125] project is one example for such an effort. It is written in C and implements matrices as C structs, encoding the most important information of the matrix such as its shape, data type, and so on. The BML focusses on sparse matrix representations and their operations for double and single precision floats as well as double and single precision complex data types. Most of the operations are then handled by subprojects like cuBLAS, cuSolver, MAGMA^[126,127] and the AMD GPU equivalents. The choice for C is motivated by the good language interoperability, especially towards FORTRAN implementations.^[120]

An effort, also heavily based on the template feature of the C++ language, is the matrix template library (MTL).^[128] However, this library appears to be no longer maintained as the last commit has been 4 years ago and a GPU interface could not be readily found. An initiative partially led by intel is the oneMATH^[129] project unifying the interface to all major GPU accelerator vendors. This interoperability is based on the SYCL programming language that acts as an abstraction layer for heterogeneous computing platforms. However, this exposes application developers to a new programming language, although based on C++ 17.

Our goal with LAHVA is to provide an interface that is highly modular through the use of C++ templates and that can be easily used by developers familiar with C++ but without any previous experience in GPU programming. A common interface for CPU and GPU operations simplifies software developing. Our current effort does not allow offloading to AMD GPUs, however an extension would be straightforward as we provide abstract base classes that define the interface.

4.3 Computational details

The timing results shown in this chapter have been generated on hardware of type 1, see tab. 8.1. We have modified the tblite codebase^[111,130] to include exchange contributions to the Fock matrix and energy. Those changes can be found in the “new-cpp-exchange” branch of the tblite fork of the author of this thesis (commit dd6c898). The semiempirical Mulliken-exchange implementation has been added to the GAMBITs codebase^[131] (branch: “mulliken-exchange”, commit ee71b716). Although, it does not directly belong to the functionality of that library it allowed fast testing as GAMBITs is already linked to the tblite codebase. For all single-point calculations, the standard SCF convergence threshold of $1.0 \cdot 10^{-6}$ H and an electronic Fermi temperature of 0 K is used. The addition of an exchange interaction in xTB made the use of the direct inversion of the iterative subspace (DIIS)^[62,132] acceleration model necessary. This model has been implemented by Mark Heezen in the GAMBITs codebase. The LAHVA version used in this chapter is from the branch “half-precision-mp”, commit: 6ba6c211. To speed-up the computation, a GPU-accelerated version of the diagonalization algorithm (DSYGVD) is used. All calculations are based on the parameter set of the GFN2-xTB method, a full range exchange fraction of 20 % is added to the otherwise unchanged Hamiltonian. To compute the γ values, the Mataga-Nishimoto formalism is used together with the arithmetic averaging scheme for the chemical hardness.

4.4 Key features and timing results

In the following subsection, the software design choices that were made to achieve the desired functionality for LAHVA will be presented. As the target user base for LAHVA are scientific application developers, we will restrict the discussion of abstract software design choices to a minimum. Instead, the implications and advantages of using LAHVA to implement a previously published model will be presented. For this purpose, we chose the semiempirical Mulliken-approximated exchange as presented by Bannwarth and Martínez.^[133] It is a prime example for the use case that LAHVA was written for. Bannwarth has already shown that a mixed precision incremental scheme can be used to efficiently accelerate the computation of this exchange term using CG-GPUs.

The general form to compute the Mulliken-approximated exchange contribution $\tilde{\mathbf{K}}$ to the Fock matrix is as follows:

$$\tilde{\mathbf{K}} = \frac{1}{4} \{ \gamma \circ (\mathbf{SDS}) + \mathbf{S} [(\mathbf{DS}) \circ \gamma] + [\gamma \circ (\mathbf{SD})] \mathbf{S} + \mathbf{S}(\mathbf{D} \circ \gamma) \mathbf{S} \} \quad (4.1)$$

We use $\circ$ to denote the element-wise product also known as Hadamard product. The Coulomb-type interaction terms are computed per shell using the Mataga-Nishimoto-Ohno-Klopman formalism described by Bannwarth.^[133] These shell-wise values are then mapped to the corresponding AO orbitals giving the symmetric matrix γ with the dimension N_{bf} by N_{bf} . For this general form, 6 general matrix-matrix multiplications (GEMMs) are needed. If the density matrix $\mathbf{D}$ is symmetric, $\mathbf{D} = \mathbf{D}^T$, the term simplifies to:

$$\tilde{\mathbf{K}}_{\text{sym}} = \frac{1}{4} [\mathbf{B} + \mathbf{B}^T] \equiv \tilde{\mathbf{K}}_{\text{sym}}[\mathbf{S}] \quad (4.2)$$

$$\mathbf{B} = \frac{1}{2} \gamma \circ (\mathbf{SDS}) + \left[\frac{1}{2} \mathbf{S}(\mathbf{D} \circ \gamma) + \gamma \circ (\mathbf{SD}) \right] \mathbf{S} \quad (4.3)$$

For the symmetric case, only 4 GEMMs need to be computed, accounting for the use of intermediate quantities.

When using an incremental Fock build, as briefly described in section 3.2, the numerical values of the contributions to the Fock matrix get smaller. Therefore, Bannwarth developed a mixed-precision (MP) incremental scheme, where the increment to the exchange contributions $\Delta\tilde{\mathbf{K}}_{\text{MP,sym}}$ are:

$$\begin{aligned} \Delta\tilde{\mathbf{K}}_{\text{MP,sym}} = & \underbrace{\Delta\mathbf{D} \circ \gamma}_{\text{FP64}} \\ & + \underbrace{\frac{1}{2} [(\Delta\mathbf{D}\bar{\mathbf{S}}) \circ \gamma + (\Delta\mathbf{D} \circ \gamma)\bar{\mathbf{S}}] + \frac{1}{2} [(\Delta\mathbf{D}\bar{\mathbf{S}}) \circ \gamma + (\Delta\mathbf{D} \circ \gamma)\bar{\mathbf{S}}]^T + \tilde{\mathbf{K}}_{\text{sym}}[\bar{\mathbf{S}}]}_{\text{FP32}} \end{aligned} \quad (4.4)$$

with: $\bar{\mathbf{S}} = \mathbf{S} - \mathbf{I}$

For the details on the derivation of this scheme, please refer to the supporting information of ref. [133].

For our tutorial implementation, we will restrict ourselves to the symmetric density matrix case and only consider contributions to the Fock matrix and no nuclear gradients. Hence, we need to write methods to compute $\tilde{\mathbf{K}}_{\text{sym}}$ using FP64 and FP32 arithmetics. The former is mainly used for reference, while the latter is needed to compute $\tilde{\mathbf{K}}_{\text{sym}}[\tilde{\mathbf{S}}]$ or $\tilde{\mathbf{K}}_{\text{sym}}[\mathbf{S}]$. Additionally, for the MP scheme, we need to compute the remaining terms from eq. 4.4. However, by reusing the intermediates from the computation of $\tilde{\mathbf{K}}_{\text{sym}}[\tilde{\mathbf{S}}]$ the number of necessary GEMMs remains 4.

4.4.1 Templated tensor classes

Whereas other projects like the BML are putting an emphasis on sparse matrix representations, we are focussing on dense linear algebra for single node applications. A later extension of the functionality towards multi-node or sparse representations is straightforward due to the modular design of the library. In LAHVA, we currently implemented vector (`Vector`), matrix (`Matrix`), and lower triangular matrix (`LowTriMatrix`) classes. These classes are implemented as pure CPU classes bearing control over a CPU memory pointer and a CPU+GPU class handling pointers to both memory spaces.

One source of inspiration for our implementation is the C++ standard library. In the vector header file it defines a vector container: `std::vector<T, Allocator = std::allocator<T>>`. `T` and `Allocator` are template parameter. `T` is the datatype stored in the vector, which can be elementary types such as `double` (FP64) or `float` (FP32) or more complex structures such as user-defined classes. A class definition is written using `T` as placeholder for the actual datatype. Upon requesting an instance of a vector with a certain data type, a class with that specific type will be created by the compiler. Hence, `T` must be known on compile time. This flexibility comes at a cost, the time needed for compilation of a project heavily relying on templates is longer than one with constant type classes. However such an implementation allows us to provide our classes for any data type requested, which is especially beneficial for mixed-precision applications.

The second template parameter of the `std::vector` is the `Allocator`, which is a class that serves to allocate and deallocate memory for any number of `T`-type objects. The `Allocator = std::allocator<T>` statement in the class definition means that if the developer is not providing a specific allocator, the one defined by the standard library will be used. The benefit of this second template parameter might not be obvious at this point, however it will become clearer when attempting to implement asynchronous GPU applications. In general, the decoupling of the allocator and the container allows an even greater level of flexibility and also performance optimization. Writing custom allocators is an active field of research, we will only give a few

references^[134,135] as this field is not the focus of this work. In contrast to the standard library's containers, dynamic memory allocation is not implemented for LAHVA classes. This means that our tensor classes are constructed with a fixed size and a new instance needs to be created upon changing the required size.

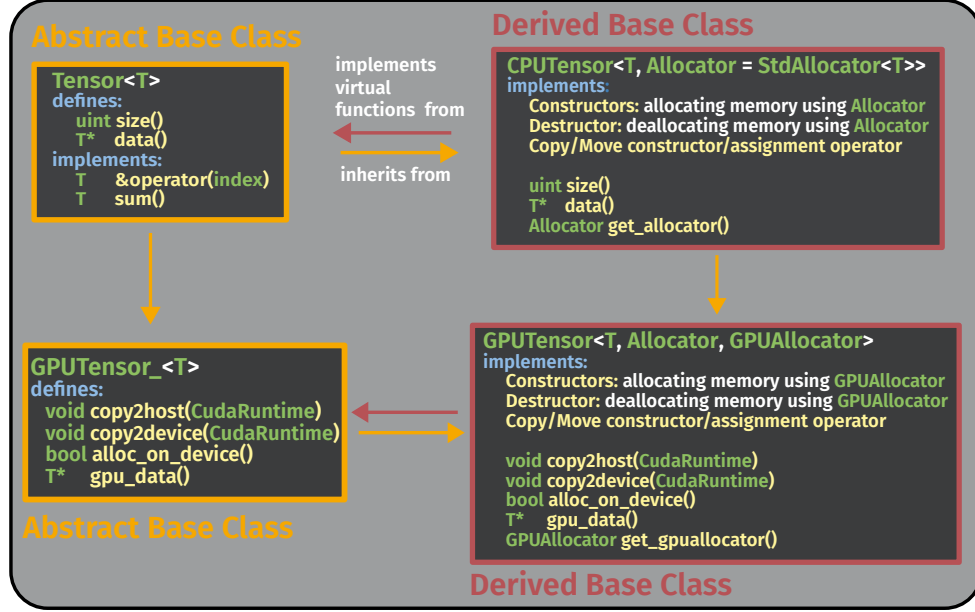


Fig. 4.2: Inheritance relationships between the CPU and GPU abstract base classes and the their derived classes. A selection of member functions is shown to demonstrate how polymorphism is used in the LAHVA project.

Allocating memory to store a certain count of T instances using an Allocator is a functionality common to all tensor classes. We could, in theory, write this functionality for every individual class. However, this would be very tedious and could easily lead to errors. Additionally, such a software design would lead to more code that is harder to maintain. Therefore, we are relying heavily on the design principles called inheritance and polymorphism.

Inheritance refers to the ability of a class to derive or inherit methods and attributes from another class. The inheritance functionality is often used to implement polymorphism for classes. One way would be to define a base class, that defines the interface and thereby the desired functionality of a family of classes. In fig. 4.2, we show how we applied those principles to our tensor classes. In general, classes ending in “_” are abstract classes that cannot be instantiated and are meant for library-specific purposes. In a similar fashion, class member functions and member variables, also called attributes, ending in _ are not meant for direct manipulation by the user, but are private variables.

For the CPU and CPU+GPU family of classes, we define an abstract base class (in yellow), Tensor and GPUTensor_ respectively. Those abstract base classes define the

interface of the methods that are common to all tensor classes. All tensor classes need to have a method to get the size of the tensor and to access the pointer holding their values in memory. Based on those methods, we can implement an access operator and a sum method. The `CPUTensor` class inherits from the `Tensor` class and needs to implement the member functions defined by it. Therefore, the purpose of the `CPUTensor` class is to allocate the right amount of CPU memory for all tensor classes that will be based on it. Additionally, it will implement the method `size()` and `data()` for all tensor classes as well. To extend this design strategy to tensor classes that handle CPU and GPU pointers, we let the `GPUSensor_` class inherit from the `Tensor` class and the `GPUSensor` class from the `CPUTensor` class. The effect will be that the `GPUSensor` class only needs to define the allocation of GPU memory, the allocation of CPU memory will be readily available through inheritance. For the GPU+CPU class, a method to copy from device to host and back is also essential to all tensor classes, see fig. 4.1.

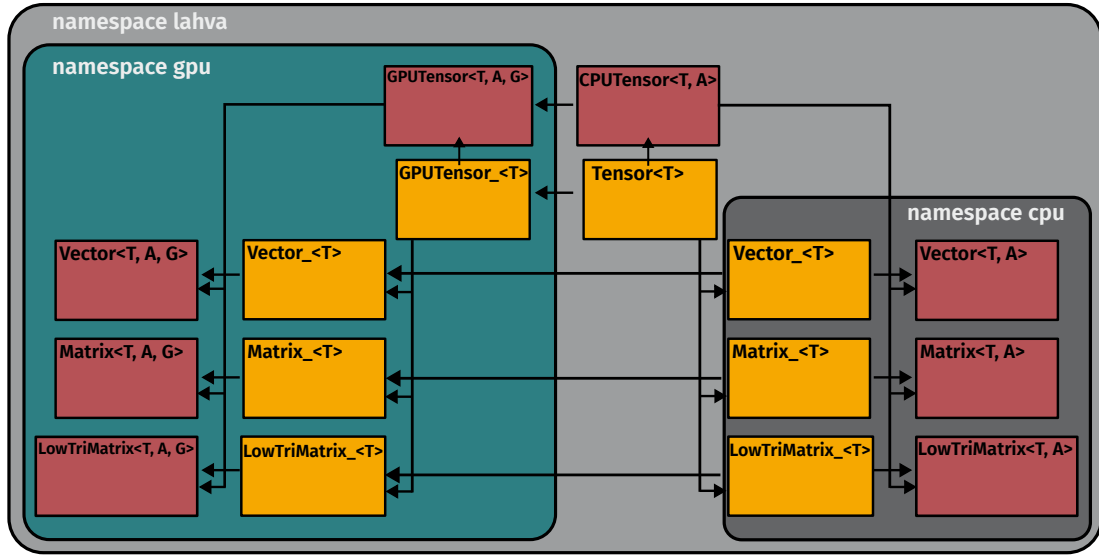


Fig. 4.3: Overview on the **derived** and **abstract** classes implemented in the LAHVA project. The arrows present the inheritance relationship between classes. The namespaces are used to organize the classes.

To further organize our codebase, we are introducing separate namespaces for code designated for execution on GPU and CPU. This also allows us to use the same names (`Vector`, `Matrix`, `LowTriMatrix`) for CPU and CPU+GPU classes, leading to a cleaner codebase. In fig. 4.3, an overview on the **derived** and **abstract** classes implemented in LAHVA is shown. From figs. 4.2 and 4.3, we can see that the abstract classes are devoid of allocator template parameters. This was done to make function definitions for BLAS operations independent of the allocators in use. Including the allocators would have meant to give to the compiler all possible combinations of template parameters that could be used as requested by the C++ standard. This would break the premise of having a flexible implementation, where new allocators could be easily added. From fig. 4.3 we can also observe that the abstract CPU classes are parents of the abstract GPU classes. This allows the use of CPU operations on CPU+GPU classes

independent of the GPU buffer or in situations where the CPU operation is faster. After this general introduction to the LAHVA tensor classes, the implementation for the Mulliken-exchange application will be discussed.

4.4.2 Using LAHVA's tensor classes to implement a base class for Mulliken exchange

```

1  template <typename ioprec, typename blas_impl, typename wprec>
2  class ExchangeBase : public ExchangeModel
3  {
4  protected:
5      template <typename U>
6      using Vector = typename TensorFactory<blas_impl>::template Vector<U>;
7      template <typename U>
8      using Matrix = typename TensorFactory<blas_impl>::template Matrix<U>;
9      using RunTime = typename TensorFactory<blas_impl>::RunTime;
10     using Indexer = SQM_Indexer;
11
12     Matrix<wprec> Overlap_;
13     Matrix<wprec> Density_;
14     Matrix<wprec> gamma_;
15
16     Indexer *indexer_;
17     bool allowsingle_;
18     bool incremental_;
19
20     wprec half = 0.5;
21     wprec one = 1.0;
22
23     std::shared_ptr<RunTime> runtime_;
24     std::shared_ptr<Timer> timer_;
25 };

```

Fig. 4.4: Class definition of the templated exchange base class. The datatype of the `Overlap_`, `Density_` and `gamma_` matrix class is defined by the `wprec` template parameter of the exchange class. The type of matrix and runtime class are taken from the `TensorFactory` class, which is the focus of fig. 4.6.

Similar to the strategy for our tensor classes, we also implement classes for our exchange application. First, an abstract base class is defined, holding all attributes necessary for implementations. In fig. 4.4, we can see that the `ExchangeBase` class has 3 template parameters `ioprec`, `blas_impl` and `wprec`. `blas_impl` is the crucial parameter to switch easily between GPU and CPU BLAS operations and will become more important in the next subsection. The other two parameters define the the working precision (`wprec`), i.e. the numerical precision in which the eq. 4.1 or 4.2 are evaluated. That numerical precision can differ from the precision (`ioprec`) in which values are provided to our library and returned to the calling program afterwards. The former parameter is used to define the floating point precision of the class attributes `Overlap_`, `Density_`, and `gamma_`. This setup allows the reuse of this class definition for any combination of `ioprec` and `wprec`. For example, for an incremental Fock scheme, we can create one

class to compute the the full Fock contribution using `ioprec=double`, `wprec=double` and a second to compute the increments (`ioprec=double`, `wprec=float`). The tedious part of writing the same functionality for the `float` precision is done by the compiler. This mechanism will work for other tensor classes as well. In the next subsection, the `blas_impl` template parameter switching between GPU-accelerated classes and CPU-only classes will be further explained.

4.4.3 Switching compute hardware with one parameter

In order to easily switch the compute hardware used for a BLAS operation, the function calls to the CPU and GPU operations need to be implemented. Similar to fig. 4.3, the function definitions for CPU and GPU are separated into the namespaces `cpu` and `gpu`. The signatures of GPU and CPU functions differ by the abstract tensor class and the runtime type used. Fig. 4.5 shows the function definitions for the BLAS AXPY kernel, that adds tensor `x` scaled by `a` to tensor `y`. For the CPU function a runtime object is not needed as there are no additional steps involved when computing on the CPU. However, to keep the signature similar to the GPU function, we introduce a dummy object `CPURuntime`, which is not forwarded to the actual function call. The GPU function call is similar, but instead of the abstract CPU base class `Tensor`, the abstract base class `GPUTensor_` is used. Since `Tensor` and `GPUTensor_` are the base classes for all tensor classes, we can use the AXPY kernel with `Vector`, `Matrix` and `LowTriMatrix` objects.

```

1  namespace lahva
2  {
3      namespace cpu
4      {
5          /// @brief Wrapper to the BLAS Level 1 DAXPY kernel
6          void AddVectors(const double a, const Tensor<double>& x, Tensor<double>& y);
7          /// @brief Wrapper to the BLAS Level 1 SAXPY kernel
8          void AddVectors(const float a, const Tensor<float>& x, Tensor<float>& y);
9          /// @brief Wrapper to the BLAS Level 1 AXPY kernel with dummy runtime argument
10         template <typename... Args>
11         void AddVectors(const CPURuntime &rt_, Args &&...args)
12         {
13             (AddVectors(args...));
14         }
15     } // namespace cpu
16     namespace gpu
17     {
18         /// @brief Wrapper to the BLAS Level 1 DAXPY kernel
19         void AddVectors(const CudaRuntime& cudart, double a, const GPUTensor_<double>& x,
20         GPUTensor_<double>& y);
21         /// @brief Wrapper to the BLAS Level 1 SAXPY kernel
22         void AddVectors(const CudaRuntime& cudart, float a, const GPUTensor_<float>& x,
23         GPUTensor_<float>& y);
24     } // namespace gpu
25 } // namespace lahva

```

Fig. 4.5: Example code showing the definition of linear algebra operations in the LAHVA codebase. Functions executed on the CPU are collected in the `cpu` namespace, whereas those executed on the GPU are placed in the `gpu` namespace.

To implement the desired “one-template-switching” mechanism, a method to switch the runtime and tensor objects between CPU and CPU+GPU is needed. The C++ standard allows classes, structs or datatypes as template parameters but not namespaces. Hence, we cannot simply use the previously defined namespaces to switch between CPU-only and CPU+GPU classes. Therefore, when wrapping the lahva namespaces to our project, a new namespace `cpuBLAS` resolving the `lahva::cpu` namespace and containing an empty struct `cpu_BLAS` is created. The use of this struct allows the compiler to find the function of the respective namespace more easily. Additionally, this struct is used as a template parameter for a class referred to as `TensorFactory`, see fig. 4.6. This class uses conditional functions from the C++ standard library to switch between the CPU or CPU+GPU Tensor classes and runtime classes. This mechanism is used in class definitions such as `ExchangeBase` (see fig. 4.4). Together with the `blas_impl` template parameter, the switching mechanism for the tensor attributes can be realized. Although, the exchange application only shows this concept for the `Matrix` class, it can be applied to other tensor classes as well.

```

1  namespace gpuBLAS
2  {
3      using namespace lahva::gpu;
4      struct gpu_BLAS{};
5  };
6  namespace cpuBLAS
7  {
8      using namespace lahva::cpu;
9      struct cpu_BLAS{};
10 };
11
12
13 template<typename blas_impl>
14 class TensorFactory
15 {
16     public:
17         template<typename T>
18         using CPUAll = CudaHostAllocator<T>;
19         template<typename T>
20         using GPUAll = CudaDeviceAsyncAllocator<T>;
21
22         template<typename T>
23         using GPUMatrix = gpu::Matrix<T,CPUAll<T>,GPUAll<T>>;
24
25         template<typename T>
26         using Matrix = typename std::conditional<std::is_same<blas_impl, cpuBLAS::cpu_BLAS>::value,
27             cpu::Matrix<T>, GPUMatrix<T>>::type;
28
29         using Timer = typename std::conditional<std::is_same<blas_impl, cpuBLAS::cpu_BLAS>::value,
30             CPUTimer, GPUTimer>::type;
31         using RunTime = typename std::conditional<std::is_same<blas_impl, cpuBLAS::cpu_BLAS>::value,
32             CPURuntime, CudaRuntime>::type;
33 };

```

Fig. 4.6: Definition of the `TensorFactory` class regrouping all types needed to switch between CPU and GPU execution. For example, the `Matrix` alias is defined based on the value of the `blas_impl` template parameter. This alias definition is then used in the class definition of groups that allow execution hardware switching like `ExchangeBase`, see fig. 4.4.

Having implemented the switching mechanism to the tensor classes, it needs to be extended to the implementation of member functions. The Mulliken-approximated exchange model has been transferred to a BLAS-operation based pseudocode by Ban-

nwarth and is implemented in the TeraChem software package.^[133] In fig. 4.7, we show the member function implementation for the symmetric density matrix case (see eq. 4.2). In a first step, we use the CPU copy operation to copy the data from the calling program to our matrix objects. Currently the library implementing this exchange type is linked to a CPU-based program, however the transfer of GPU buffers can easily be realized. The Fock matrix is treated separately, as its initialization depends on the use of an incremental scheme. To minimize the amount of matrix-matrix multiplications, two temporary buffers are used to store intermediates. The hardware executing the remaining functions is then controlled by the `blas_impl` template parameter. The last step involves copying back the Fock matrix to the buffer of the calling program. This results in a very clean and concise implementation without obfuscating the underlying physical model. All copying and initialization instructions for the GPU buffer are hidden in LAHVA and only copying from the device to the host must be performed explicitly (in the `FockOutput` function).

```

1  using namespace cpuBLAS;
2  using namespace gpuBLAS;
3  template<typename ioprec, typename blas_impl, typename wprec>
4  void SymSQMExchange<ioprec, blas_impl, wprec>::ComputeFock(const ioprec* Overlap,
5                    const ioprec* Density, const ioprec* gamma, ioprec* Fock)
6  {
7      //copy values to work buffers
8      CopyVectors(Overlap_.size(), Overlap, Overlap_.data());
9      CopyVectors(Density_.size(), Density, Density_.data());
10     CopyVectors(gamma_.size(), gamma, gamma_.data());
11     Matrix<wprec> Fock_;
12     this->FockInput(Fock_, Fock);
13
14     //create temporary matrices
15     Matrix<wprec> A(indexer_>nao, indexer_>nao);
16     Matrix<wprec> B(indexer_>nao, indexer_>nao);
17
18     MatrixMatrixProduct(*runtime_, Overlap_, Density_, A);
19     MatrixMatrixProduct(*runtime_, A, Overlap_, B, half);
20
21     HadamardProduct(*runtime_, gamma_, B);
22     HadamardProduct(*runtime_, gamma_, A);
23     HadamardProduct(*runtime_, gamma_, Density_);
24
25     MatrixMatrixProduct(*runtime_, Overlap_, Density_, A, half, one);
26     MatrixMatrixProduct(*runtime_, A, Overlap_, B, one, one);
27     AddMatrices(*runtime_, B, B, Fock_, quarter, quarter, "N", "T");
28
29     //copy back to output Fock matrix
30     this->FockOutput(Fock_, Fock);
31 };
```

Fig. 4.7: Implementation of the symmetric density matrix variant of the Mulliken-approximated exchange according to the pseudocode from ref. [133]. Using LAHVA, we can define this function in a general way as a function of the template parameters. By creating a class using `blas_impl=gpu_BLAS` the main computational steps (lines 18-27) are performed on the GPU.

In order to show the effect of GPU-offloading with LAHVA, we will be comparing timings for the computation of the Mulliken-approximated exchange contributions to

the Fock matrix on CPU and GPU. To assess the scaling of the computation time with the number of basis functions, a set of fullerene molecules^[136] is used. The details of the computational setup have been presented in section 4.3. The recorded walltimes include the setup function, computation of the γ values, and the exchange contribution to the Fock matrix. “fullKsymm” refers to the exchange contributions according to eq. 4.2, computed in FP64. “incrKsymm” refers to one fullKsymm calculation followed by incremental contributions according to eq. 4.4. For incrKsymm, the fullKsymm calculation is performed in FP64, whereas the incremental contribution is computed according the numerical precisions in eq. 4.4. The latter portion is the function block shown in fig. 4.7. The total runtime of these program sections is summed and the average over 5 consecutive runs is shown. We show the speed up gained from running the calculations on the GPU as opposed to the CPU in FP64. Additionally, the walltime of the incremental MP scheme by Bannwarth and the associated speed-up compared to the CPU FP64 values is also shown. The calculations in fig. 4.8 have been recorded on a type 1 system (see tab. 8.1).

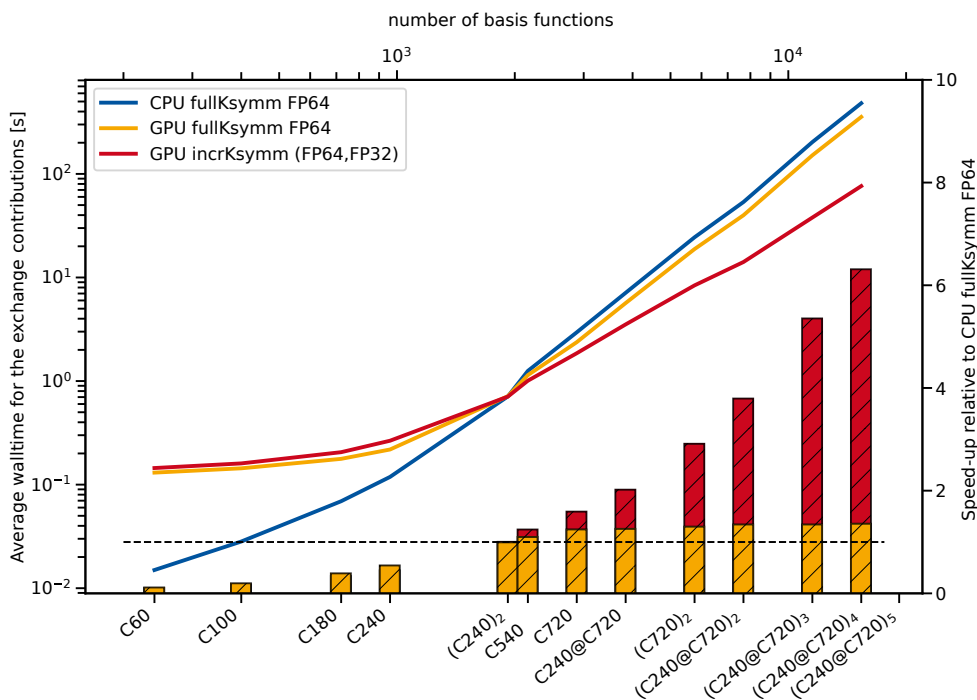


Fig. 4.8: Comparison of the walltime needed to compute the exchange contributions to the Fock matrix (according to eqs. 4.2 and 4.4 for fullKsymm and incrKsymm respectively). The setup of the C++ classes and the computation of the γ values is also included. System 1 from tab. 8.1 is used for all computations. 16 CPU threads are used for all CPU and GPU configurations. One RTX 4090 is used for the GPU configurations. The second y-axis shows the speed-up of the GPU configuration relative to the CPU full build using FP64 arithmetics.

Fig. 4.8 reveals that for systems with less than 1000 basis functions, the walltime of the GPU-accelerated code is dominated by the time necessary to setup the GPU and its runtime (approx. 100 ms). It has to be noted that for this example, we only report the walltime for 16 CPU threads on an empty node. If less CPU threads are used or the node is occupied by other CPU calculations, the number of basis functions might

change for which GPU acceleration pays off. For the presented setup, the calculation of the exchange contribution to the Fock matrix will offer a speed-up for molecules with more than 3.000 basis functions, if the incremental Fock build MP is not used. By using the incremental Fock build and the tailored MP scheme (see eq. 4.2), the payoff size can be lowered to 2.000 basis functions. The speed-up associated with using the incr. Fock build and MP scheme grows faster with system size than the FP64 GPU implementation, reaching a maximum of about factor 6 for the investigated systems. Using the splitting techniques presented in the introduction section, the number of FP64 operations can be lowered for the full Fock build FP64 and incr. MP scheme. The details of these techniques and how we implemented them to work seamlessly in our LAHVA environment are the subject of the next section.

4.4.4 Half precision splitting schemes

As already discussed in sec. 3.3, splitting schemes that are based on half precision are very promising for modern hardware. Although, most performance upgrades are released on the HPC GPUs, CG-GPUs also benefit from splitting schemes as they allow to minimize the amount of FP64 calculations. In our preliminary studies, we observed that the Markidis scheme is not always able to reproduce polynomials of the density matrix well. Additionally, there is no direct formulation of the Markidis scheme starting from a FP64 matrix. Therefore, we decided to focus on the Ozaki scheme.^[103]

In its original formulation, the Ozaki scheme is used for inner products. The vectors are then each split into either a fixed number of lower precision tensors or until the largest element equals 0.0. The latter option has been proven to give a result that is numerically equivalent to the original values.^[103,104] The inner product mechanism can then be generalized for matrix-matrix multiplications, where each column or row is split and later rescaled. The original Ozaki scheme is therefore tedious to implement and a multitude of handwritten kernels are needed. However, our preliminary numerical tests showed that the scheme can be simplified significantly if the splitting scheme is applied to the whole matrix at once. This leads to alg. 4.1. The main alteration to the original publication is in the computation of π . Instead of using the number of elements in a row, we always use the constant 8 in the last term of π . Using this splitting setup, we generate `n_split` FP16 matrices `X_split` and a vector `c` containing the scaling coefficients with `max_split` entries. In the next phase, we compute the matrix-matrix multiplication using the FP16 split matrices, see fig. 3.3. Due to the construction of the splitting matrices, the largest numerical values are captured by the split matrices with smaller indices. Therefore, the multiplication phase can be approximated by only taking the combinations of the upper triangle, which are colored in a darker orange shade. This simplification has also been used by Dawson et al.^[101] to accelerate their splitting scheme with minimal impact on numerical accuracy. When using four split matrices to represent FP64 values, the neglect of the smallest contri-

butions would reduce the number of GEMMs from 16 to 10. This setup will be used throughout this work. Similar to other implementations we use the `cublasGemmEx` or `cublasGemmBatchedEx` function for this second phase. We found that the batched version gives lower numerical divergences compared to the non-batched version. This behavior needs further research, especially regarding the cuBLAS release 11.8 that was used. In a last step, the computed C split matrices are summed and scaled to give the final C matrix. A mixed precision (FP32/FP64) version of the AXPY kernel was written for this step.

Alg. 4.1: Pseudocode illustrating a simplified version of the Ozaki splitting scheme,^[103] that only uses one scaling factor per split matrix. The main change is found in the first line of the function. The code in the for loop is presumed to be parallelised over all elements i of the matrix X . `n_split` is the number of splits, this number depends on the input floating point precision. However, an alternative formulation could be to perform splits as long as $\mu \neq 0.0$.

```
void SplitMatrix(max_split, n, X, X_split[0:n_split], c[0:n_split])
{
     $\pi = \text{ceil}(\log_2(\mathbf{u}_{\text{FP64}}^{-1}) - (\log_2(\mathbf{u}_{\text{FP32}}^{-1}) - \log_2(8))/2)$ ;
    j = 0;
    for (size_t j = 0; j < n_split; j++)
    {
        // find largest absolute value in X array
         $\mu = \max_{0 \leq i \leq n}(|X_i|)$ ;
         $c[j] = \tau = \text{ceil}(\log_2(\mu))$ ;
         $\sigma = 2^{\pi+\tau} \cdot 0.75$ ;
        // GPU parallel region
        idx = blockIdx.x * blockDim.x + threadIdx.x;
        // thread private variable x_tmp
        x_tmp = flFP64((X[idx] +  $\sigma$ ) -  $\sigma$ );
        X[idx] = flFP64( X[idx] - x_tmp);
        X_split[j][idx] = flFP16( flFP64(  $2^{-\tau}$  x_tmp) );
        // end GPU parallel region
    }
}
```

Based on the previous discussion, we can see that a lot of operations and additional buffers are needed to use such a splitting scheme. In the spirit of providing a developer-friendly implementation, we organized all this functionality in a new class: `MixedPrecisionMatrix<T, Allocator, GPUAllocator>`. This class directly inherits from the `Matrix` class, where the template parameter T determines the high accuracy representation of the MP matrix. By inheriting from `Matrix`, we can use all other functions in LAHVA with the high precision buffer and only specialize the matrix-matrix multiplication operation. This makes the new class essentially a drop-in alternative to the `Matrix` class. The FP16 accelerated matrix-matrix multiplication can be used with FP64 or FP32 high precision values. The function signature is similar to the general case with the addition of a temporary buffer that can be used to split A and B. This is necessary as the splitting mechanism in use changes the values of the original buffer, however according to function definition A and B should remain unchanged.

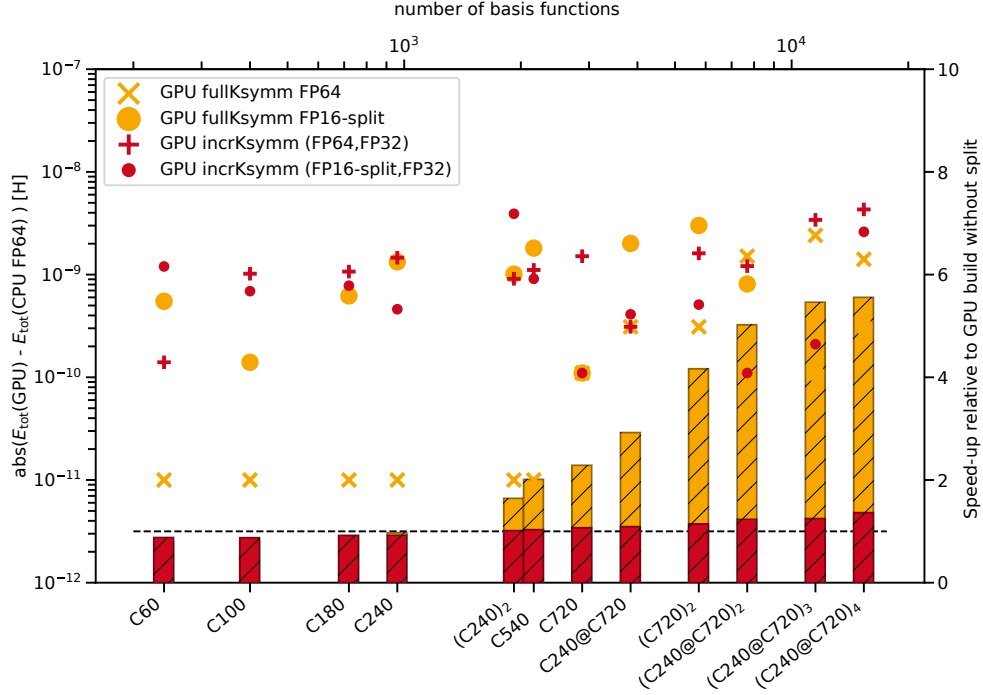


Fig. 4.9: Absolute difference in total energy for the GPU implementations of the Mulliken-approximated exchange compared to the CPU FP64 implementation. Eq. 4.2 corresponds to fullKsymm either computed in pure FP64 or using the FP16-split scheme from sec. 4.4.4. Eq. 4.4 corresponds to incrKsymm respectively, where the first FP precision refers to the precision in which the first iteration using eq. 4.2 for a full build is performed. A value of 10^{-11} H is added to all differences to make numerically identical values visible on the figure. The computational setup of these calculations is detailed in sec. 4.3, hardware setup 1 from tab. 8.1 is used. The speed-up gained from using the FP16 splitting scheme compared to the non-split version is shown, i.e. GPU full build FP64 vs. GPU full build FP64 FP16-split. For the FP16 splitting scheme only the 10 combinations with the largest numerical contributions are computed.

The effect of using this modified Ozaki scheme for our Mulliken-approximated exchange will be investigated for the full build FP64 and incr. FP64/FP32 scheme (from eq. 4.2). For the later method, only the first FP64 full Fock build will be accelerated by the FP16 splitting scheme. The use of a FP16 splitting scheme for FP32 matrix-matrix multiplications is in most cases not beneficial on CG-GPUs. On CG-GPUs, like the RTX 4090, the theoretical performance of FP16 with FP32 accumulate GEMMs is only a factor 2 faster than FP32 GEMM. For the FP64 GEMMs, a 4-fold split is used for the **A** and **B** matrices. Out of the 16 potential combinations in the second step, only the 10 combinations with the largest contributions are computed, i.e. the darker orange combinations in fig. 3.3. To assess the numerical accuracy of the energies computed using this split scheme, the total energy of the FP16 split variants of the full build and the incremental MP build is compared to the full build CPU reference values in fig. 4.9. Additionally, the speed-up realized by using this FP16 split variant compared to the GPU implementation without split is also shown.

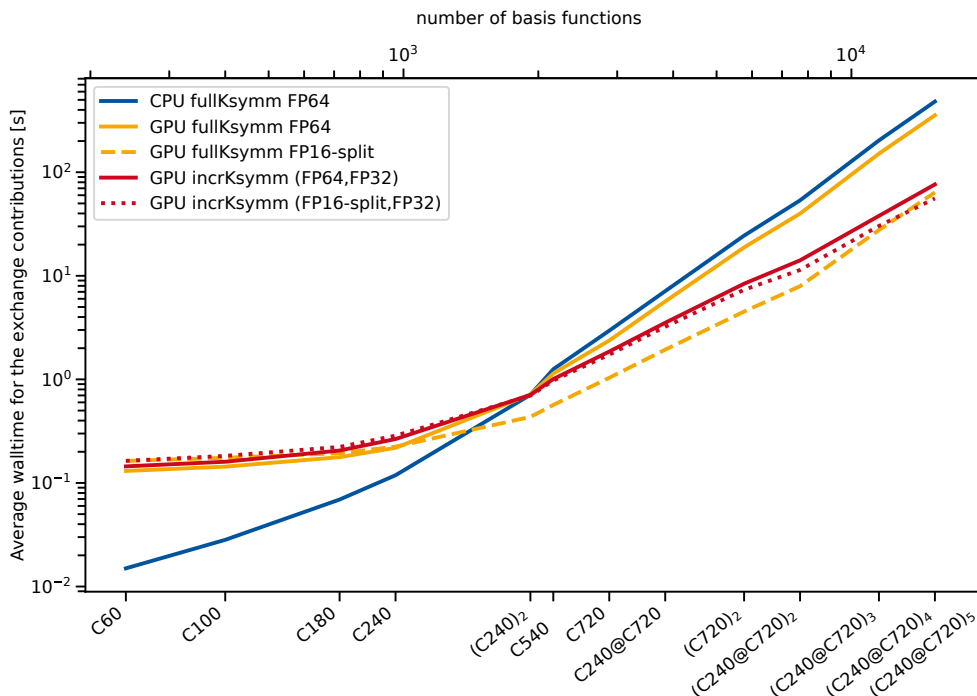


Fig. 4.10: Comparison of the walltime needed to compute the exchange contributions to the Fock matrix, the setup of the C++ classes and the computation of the γ values. Eq. 4.2 corresponds to fullKsymm either computed in pure FP64 or using the FP16-split scheme from sec. 4.4.4. Eq. 4.4 corresponds to incrKsymm respectively, where the first FP precision refers to the precision in which the first iteration using eq. 4.2 for a full build is performed. System 1 from tab. 8.1 is used for all computations. 16 CPU threads are used for all CPU and GPU configurations. One RTX 4090 is used for the GPU configurations. For the FP16 splitting scheme only the 10 combinations with the largest numerical contributions are computed.

The absolute energy differences shown in fig. 4.9 are all below the energy threshold of the SCC of $1.0 \cdot 10^{-6}$ H. The deviations in absolute energy are increasing for all implementations with the considered system size. This can be attributed to the increase in the numerical value of the total energy, which in turn leads to an increase in significant digits needed to represent the same relative accuracy. Hence for a relative accuracy of 10^{-10} H, a total of 14 significant digits are needed for molecules larger than C_{240} as the magnitude of the total energy exceeds 1000. These numerical values are on the limit of what can be represented by FP64. Further increasing the size also leads to more operations being executed and therefore more potential rounding errors. However, our FP16 splitting scheme is not introducing a significant amount of accuracy loss. For the full Fock build variant, we can see that for some molecules the error in the total energy becomes even smaller than the FP64 GPU without splitting. This is probably connected to error cancellation during the rounding in the summation step of the splitting scheme. Using the splitting scheme leads to an increase in computation time for systems smaller than 1000 basis functions, however as shown in fig. 4.8, GPU acceleration is not advantageous for this region in the first place. In summary, for the present hardware setup, the use of the splitting scheme is advantageous for all system sizes, where the use of the GPU pays off in the first place.

In fig 4.10, the absolute walltimes with and without the FP16 splitting are compared. The speed-up of the full build using the FP16 splitting scheme is so significant, that it is faster than the incr. scheme without splitting for all system sizes above 1000 basis functions. This speed-up over the incrKsymm (FP16-split,FP32) scheme can be explained by the additional logic and additional computations needed. The most computational time intensive additional operation is the computation of $tr(\mathbf{PS})$ every cycle. This operation is performed on the CPU as a loop of N inner products of the rows and columns on the diagonal.

The computation time saving from the splitting scheme for the incr. Fock build is less pronounced and gets larger with system size. For the two largest systems in the fullerene set, the incr. MP FP16-split scheme surpasses the splitted full build.

These results show that the FP16 scheme is able to accelerate FP64 GEMMs independent of the underlying physical model and can be applied universally. However, it can also be combined with a tailored model based mixed-precision like the incrKsymm model and achieve an even greater speed-up.

4.4.5 Asynchronous calculations on a single GPU

In the previous sections we have only presented applications, where we are using the data parallelism capacities of the GPU. However, with the introduction of streams it became possible to also use task parallelism techniques with the GPU. In fig. 4.11, we illustrate how stream based parallelism can work on a GPU. In total, there are three scheduling engines, on most GPUs, of which two are memory engines. One engine coordinates all host to device memory transfers whereas the other orchestrates the other direction of transfer. The third engine schedules the execution of kernels across the GPU's SMs. In our example we show the execution of 4 subtasks in one stream, the default serial stream 0. The lower representation shows how the subtasks can be associated with a stream each and execute in a concurrent fashion on the GPU.

Besides the definition of streams and the association of kernels with that stream, several other prerequisites are necessary for asynchronous execution. All copying instructions from host to device and vice-versa need to be issued using the asynchronous copy function, provided by CUDA. As alluded to previously, we decided to use allocators for our tensor classes. In the case of GPU memory allocators, they also include the copying instructions. Therefore, we defined an abstract GPUAllocator base class that defines the desired interface. Based on that class, an asynchronous allocator has been implemented, it uses the stream defined in the CudaRuntime object for memory transfer and memory allocation. The default GPUAllocator uses the serial memory transfer and allocation functions. To ensure proper asynchronous memory transfer, the CPU buffer in use must be allocated using so-called "pinned" or page-locked memory. This allocation functionality is implemented in a separate CPUAllocator. Allocating host memory in this way takes a substantial amount of time compared to regular memory,

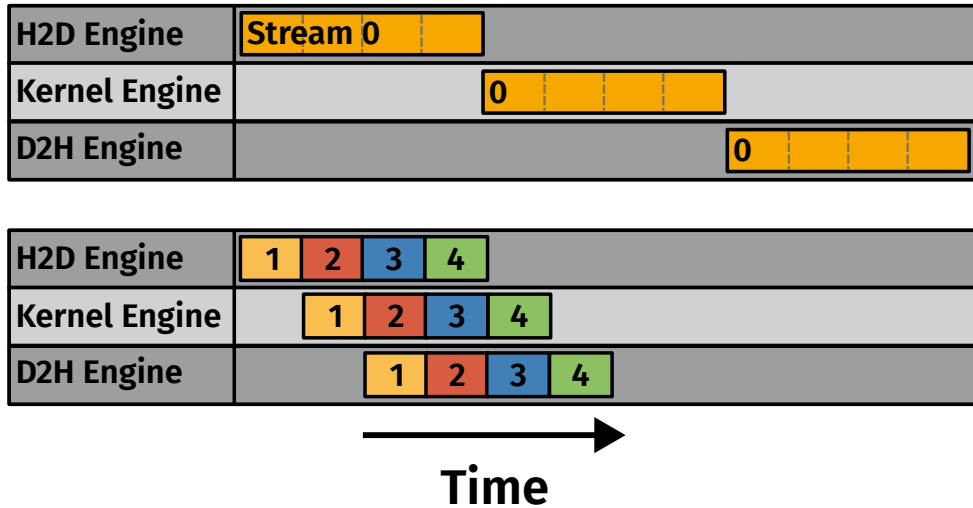


Fig. 4.11: Schematic illustration of the execution of four mutually independent subtasks as a function of time. In the upper execution model all subtasks are running in serial fashion in stream 0. The lower execution model shows how the subtasks can be run in a concurrent fashion using four non-zero streams. The model GPU used for this illustration has two memory engines, for the direction host to device and vice versa, abbreviated as H2D and D2H, respectively.

it must hence be seen as an investment to allow asynchronous execution. Lastly, we need to consider that the allocation of host memory via the CUDA runtime library leads to an implicit synchronization of the GPU. Therefore, we found a few guidelines when developing software that is using multiple concurrent streams: all CPU allocation should be done first, in the constructor for example; all tensor objects should be members of classes in order to persist during an execution. This limits the number of CPU memory allocations, especially during the concurrent part of the execution.

Coming back to our application example: in its current form, the function implemented in fig. 4.7 cannot be divided into independent tasks. However, in the case of unpaired electrons, it would be possible to compute the contribution of the α -spin channel independent of the β -spin channel. This is currently not implemented in `tblite` and would need additional implementation steps. In the following chapter, we are presenting an application of this asynchronous execution model by running multiple SCC cycles in parallel in the CREST software suite.

5 Acceleration of semiempirical electronic structure theory calculations on consumer-grade GPUs using mixed-precision density matrix purification

Parts of the content of the following chapter have been previously published as P. Steinbach, C. Bannwarth, J. Chem. Theory Comput. **2025**, *21*, 7335–7351. The figures that have been reprinted from the original publication are marked as such. Most figures have been reworked since the FP16 split method has not yet been available in the published article, but was now added in the present work. Accordingly, the discussion has also been modified compared to ref. [137].

5.1 Introduction

In fig. 5.1, the main computational steps for an SCF procedure in the orthogonal basis are summarized. In a first step, the Fock matrix F_i is computed based on the density matrix of the previous cycle P_{i-1} . The left side (in gray) shows the commonly used strategy of solving the eigenvalue equation for the MO coefficient C_i and the orbital energies ε . The new density matrix P_i is built from C_i and used to compute the energy $E(P_i)$. As mentioned before, the computation of the ERIs and therefore the Fock matrix F_i is the computational bottleneck in ab-initio methods. The computational scaling of SQM methods is usually limited by the $O(N^3)$ scaling of the diagonalization process. However, SQM methods commonly use minimal valence-only basis sets, reducing the number of basis functions drastically. This choice means that the walltime of the diagonalization step only becomes noticeable for rather large systems (> 200 atoms).

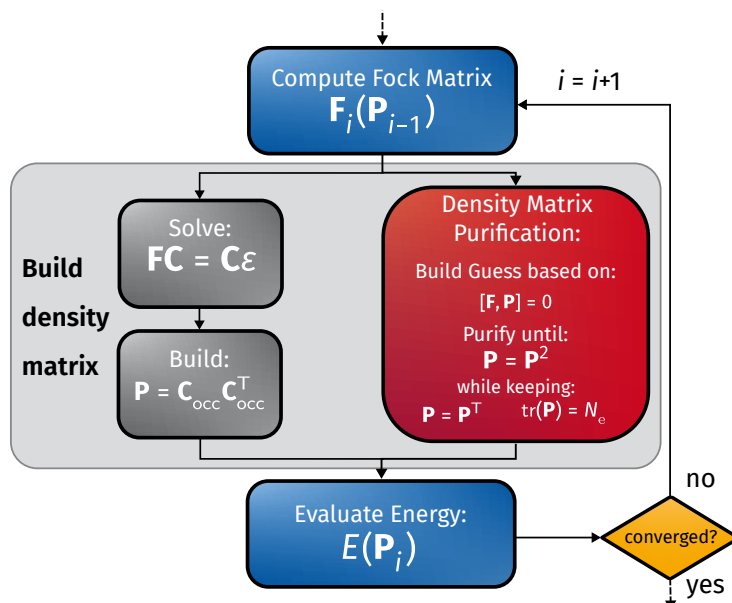


Fig. 5.1: Simplified workflow showing the options in a simplified SCF iteration loop. The density matrix from the previous iteration $\mathbf{P}_{i-1}$ is used to build the Fock matrix $\mathbf{F}_i$ of the current iteration i . From this, a new density matrix $\mathbf{P}_i$ is determined that provides the correct orbital occupation and used to compute the energy. In most cases, the Roothaan Hall equations are solved for the orbitals and the density matrix is constructed from these. Alternatively, density matrix purification techniques solve for the density matrix directly by iteratively improving a suitable guess matrix until idempotency is achieved. For brevity, the use of SCF convergence acceleration algorithms is neglected and the orthogonal basis representation is used throughout. For the same reason, the index i has been neglected in the density matrix build section. **Reproduced with permission from: Pit Steinbach, Christoph Bannwarth, and Thorsten Bach Journal of the American Chemical Society 2025, 21, 7335–7351. DOI: 10.1021/acs.jctc.5c00262. Copyright 2025 American Chemical Society.**

Already before the inventions of computers, Löwdin and McWeeny proved that the solution of the eigenvalue can be circumvented. Realizing that $\mathbf{P}$ is sufficient to compute the energy and Fock matrix, they found that solving for $\mathbf{C}$ is not required. Density matrix purification as stated on the right side (in red) of fig. 5.1 is one method to directly solve for $\mathbf{P}$. The rest of this section will introduce further theories used to solve the SCF problem without using diagonalization. Comprehensive reviews on the subject of these theories can be found under the following References [138–141].

Valid solutions to the SCF procedure must with integer orbital occupations fulfill two necessary and sufficient conditions: the idempotency relation ($\mathbf{P}^2 = \mathbf{P}$) must hold, and the commutator of $\mathbf{F}_i$ and $\mathbf{P}_i$ must be zero, see fig. 5.1 right.^[142] Based on these conditions, methods can be derived to compute $\mathbf{P}$ without solving for the orbitals first. Density matrix minimization (DMM) refers to direct optimization of $\mathbf{P}$ while enforcing the aforementioned constraints on the density-based energy expression.^[143] The number of electrons N_e in the density matrix needs to be constrained during the optimization process.^[143] However, only constraining the number of particles might lead to unphysical solutions as occupations could become negative or larger than 1, violating

the Pauli principle. Li, Nunes and Vanderbilt^[143] and Daw (LNVD)^[144] pioneered the field by combining a Lagrange multiplier-based process with the use of a purification polynomial. McWeeny was the first to propose such a purification polynomial,^[142] it refers to a function that can be used to update a trial density matrix $\mathbf{P}$ which is nearly idempotent to $\mathbf{\Pi}$ which is more idempotent. This purification polynomial of 3rd order is given in eq. 5.2.

$$\Omega(\mathbf{P}) = \text{tr}(\mathbf{\Pi}\mathbf{F}) - \mu(\text{tr}(\mathbf{P}) - N_e) \quad (5.1)$$

$$\text{with: } \mathbf{\Pi} = 3\mathbf{P}\mathbf{P} - 2\mathbf{P}\mathbf{P}\mathbf{P} \quad (5.2)$$

The process starts from a guess density, for which the gradient of the Lagrangian is computed. Minimization algorithms such as steepest descent, conjugate gradient or others are then used to propose a new $\mathbf{P}$.^[145,145,146] The formulation of the guess density also has an effect on the performance, for example Challacombe proposes a DMM algorithm where the commutation of $\mathbf{F}$ and $\mathbf{P}$ is conserved from the guess density.^[147] Lastly, changes to the purification polynomials and application process have also been reported.^[145,148] As seen in eq. 5.1, the chemical potential μ must also be known. For a canonical ensemble of electrons, μ is determined such that $N_e = \text{tr}(\mathbf{P})$. Although, the foundations of DMM methods have been published around the turn of the century, the field has also seen recent developments.^[149–152]

An alternative to minimizing the energy functional, is the use of purification polynomials, such as eq. 5.2, to construct the density matrix. These methods are summarized under the term density matrix polynomial expansion method (DMPE). The core idea is to reproduce the Fermi-Dirac distribution function f using polynomials of the scaled Fock matrix $\bar{\mathbf{F}}$, see eq. 5.3.

$$\mathbf{P} = f\left(\frac{\bar{\mathbf{F}} - \mu}{k_B T}\right) \quad (5.3)$$

The Fock matrix $\mathbf{F}$ is rescaled such that its eigenvalues fall in the interval $[-1, 1]$. The largest and smallest eigenvalues of $\mathbf{F}$ are needed for this rescaling step. However, diagonalization would be needed to compute the exact eigenvalues rendering the use of DMPE to circumvent diagonalization useless. Instead, Gershgorin’s circle theorem^[153] can be used to compute well-defined bounds of the eigenvalue spectrum. k_B is the Boltzmann constant and T the absolute temperature of the system.

In general, there are two strategies for DMPE methods, either the polynomial is determined based on an accuracy threshold or a fixed form polynomials is applied recursively until numerical convergence. For the first variant, many methods based on Chebyshev polynomials have been published.^[154–157] Methods from the second variant are also referred to as density matrix purification (DMP) techniques. However, most of them are restricted to $T_{\text{Fermi}} = 0 \text{ K}$, although recently adaptations allowing fractional occupation have been published.^[158] McWeeny’s purification polynomial in eq. 5.2 is the first to be used in DMP methods. The role of purification polynomials is to alter the orbital occupation of the guess density $\mathbf{P}_0$, which are real-valued in the interval $[0, 1]$, until they are numerically indistinguishable from 0 and 1. A disadvantage of the

McWeeny polynomial is its inflection point at 0.5, see fig. 5.2a. This inflection point makes the use of a rescaling step necessary to keep the desired number electrons.^[138] The chemical potential is also needed, unless Palser and Manolopoulos’ canonical ensemble version^[159] is used. Other polynomials avoiding the inflection point at 0.5 have also been proposed.^[160–164] In this work we will focus on the Niklasson’s 2nd-order trace correcting algorithm (SP2).^[160] He proposes a set of two purification polynomials that can be used in order to adapt the density matrix to the correct (user-specified) electron number. The details of this method will be the topic of the upcoming section. Variations of the SP2 method include a spin-unrestricted adaptation^[165] and an accelerated version.^[166,167] In ref. [137], the SP2 and SP2-accelerated and TRS4 DMP methods have been implemented, however we will focus ourselves on the SP2 method for this work.

The previously presented DMPE methods are very well suited for execution on GPUs. This can be motivated by the abundant use of matrix-matrix multiplications, for example when applying second order polynomials recursively. Dense matrix-matrix multiplications are compute-heavy workloads, which are particularly well suited for GPUs. Diagonalization, in contrast, is dominated by memory operations, leaving some of the GPU’s compute performance underused.^[108,120] Hence, the combination of DMP methods like SP2 with GPU offloading is a viable and proven strategy to lower the effective scaling of density build step in SCF workflows. Several GPU-accelerated version of the SP2 method, that use GP-GPUs, have been reported before.^[120,168] However, our focus in the context of this work is to derive algorithms that perform well on CG-GPUs. This can be achieved by reducing the number of high-precision (FP64) operations while ensuring the numerical fidelity.

Another approach to bypass the diagonalization in the SCF process is to use “second-order” methods. This class of methods starts from an expression of the energy as a function of orbital rotation parameters, which is expanded in a Taylor series and then truncated after the second term.^[169–173] This approach has the advantage that the orbital information is conserved in contrast to methods that directly solve for the density matrix. A GPU implementation that combines this approach with a quasi-Newton SCF acceleration scheme has been recently published.^[173] In that work, the so-called “Q-Next” approach outperforms diagonalization plus DIIS on V100 GP-GPUs.^[173] However, an open-source publication of this method is not available and, hence, a comparison with our approach is hard to realize. Besides that, we are proposing the use of SP2 DMP with xTB Hamiltonians, which do not need acceleration schemes based on the entire density matrix for the SCF process, as discussed in previous sections.

In the upcoming section, the theory of the SP2 method will be shortly presented followed by an introduction to our customized DMP workflow and a tailored MP scheme. The implementation presented here is realized as a library that is supposed to be interfaced to an arbitrary electronic structure program. For the tests and applications presented in this work, the library is linked to the tblite codebase,^[111] allowing access to the GFN2- and GFN1-xTB Hamiltonians. Besides presenting the numerical accuracy and timings for single-point calculations, the asynchronous capabilities im-

plemented in LAHVA (see sec. 4.4.5) are used to accelerate CREST^[7,8,174] conformational search protocols.

5.2 Theory

5.2.1 Density matrix purification methods

In the following section, we will focus on the theory of the SP2 DMP method. As mentioned before, other DMP methods have also been implemented, however, in our preliminary studies we found that SP2 gives the best ratio of reliability and computational cost. The DMP process is based on the properties of the Fock and density matrix in the orthogonal basis. Hence, a transformation of the GFN2-xTB Fock matrix to the orthogonal basis and of the density matrix back to the AO basis is necessary. The basis transformation workflow will be the topic of the following subsection.

At each SCF cycle i , the guess density $\mathbf{P}_{0,i}$ is computed based on the Fock matrix $\mathbf{F}_i$ of the current iteration. The density matrix guess based on the work of Palser and Manolopoulos^[159] is used. The occupation ratio θ is used to adapt the guess procedure to any number of electrons N_e ,^[149] see equation 5.4.

$$\mathbf{P}_0 = \theta \mathbf{I} + \min\{\beta, \bar{\beta}\} (\mu \mathbf{I} - \mathbf{F}) \quad (5.4)$$

$$\text{with: } \beta = \frac{\theta}{\tilde{\epsilon}_{\max} - \mu}, \bar{\beta} = \frac{1 - \theta}{\mu - \tilde{\epsilon}_{\min}}, \theta = \frac{N_e}{N_{\text{bf}}}$$

Gershgorin's disk theorem^[153] is used to compute estimates of the eigenvalue bounds $\tilde{\epsilon}_{\min}, \tilde{\epsilon}_{\max}$ of $\mathbf{F}$. These bounds together with θ and the chemical potential μ are used to compute the scaling parameters β and $\bar{\beta}$. μ is approximated by the arithmetic mean of the eigenvalues of $\mathbf{F}$, see eq. 5.5.

$$\mu = \frac{\text{tr}(\mathbf{F})}{N_{\text{bf}}} \quad (5.5)$$

As suggested by fig. 5.1, this guess density commutes with $\mathbf{F}$ and contains N_e . The eigenvalue spectrum of this guess density falls in the spectrum $[0, 1]$, hence purification polynomials can be used to alter the occupation of $\mathbf{P}$ until the eigenvalues are numerically indistinguishable from 0 or 1. The details of the purification process are explained in the remaining subsection, the original publications can be considered for more details.

The purification polynomials proposed by Niklasson in his SP2 method^[160] are presented in fig. 5.2a. For comparison, the McWeeny polynomial is also shown. The parabola-shaped $f(x) = x^2$ polynomial will always lower values in the range $[0, 1]$ until reaching the minimum of the function at 0. The other polynomial $g(x) = 2x - x^2$ increases the eigenvalues until reaching its maximum at 1. However, both polynomi-

als keep values at $x = 0$ and $x = 1$ constant. The effect of the recursive application of these polynomials can be seen in fig. 5.2b. The order in which the polynomials $f(x)$ and $g(x)$ are applied influences the location of the inflection point of the stepping function. Using this mechanism, the SP2 DMP method can adapt to various Fermi levels μ . This effect can be seen visually by comparing the red and the green function, resulting from different combinations of SP2 polynomials (see fig. 5.2b).

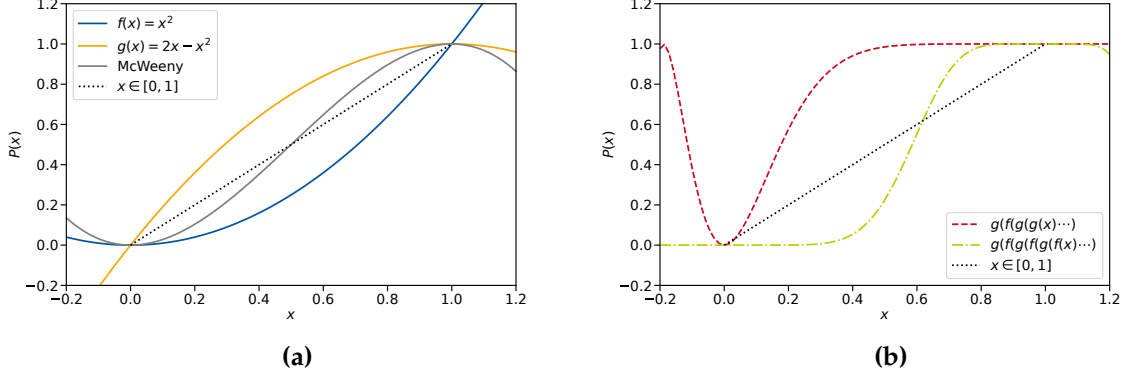


Fig. 5.2: (a) Purification Polynomials used in the SP2 DMP method^[160] and McWeeny polynomial.^[142] (b) Two combinations of the polynomials f, g presented in (a), resulting in stepping functions with different inflection points.

The resulting density matrix will be idempotent to a specified numerical threshold τ : $|\text{tr}(\mathbf{P}) - \text{tr}(\mathbf{P}^2)| < \tau$. An implementation of this theoretical description can be seen in the pseudocode in alg. 5.1. Only the main loop used to build the recursive projection polynomial is shown. The polynomial $f(x)$ is applied upon using the LowerOccup function, and RaiseOccup uses $g(x)$. If traceP is greater than the number of electrons, than the LowerOccup function is called, if P is electron deficient then RaiseOccup is used.

Alg. 5.1: Recursive build of the projection polynomial for the 2nd-order trace-correcting spectral projection.^[160]

```
// Check for convergence with respect to idempotency
while (abs(tracePP-traceP) > thresh):
{
    // compute P2 and its trace
    tracePP = computePP();
    // are we above or below the desired electron number?
    if (traceP < elnum)
    {
        // increase electron count in P
        // P = 2P - P2; also sets traceP
        RaiseOccup();
    }
    else
    {
        // decrease electron count in P
    }
}
```

```

    // P = P2; also sets traceP
    LowerOccup();
}
}

```

For electronic structure methods operating in the AO basis, the transformation to the non-orthogonal must be applied first. Then the density matrix can be used identically to one build from coefficient matrices.

5.2.2 Mixed precision approximations

The focus of this work is to derive algorithms that run competitively and efficiently on CG-GPUs. To achieve this goal, two main MP techniques are used to optimize the performance of the SP2 DMP scheme on CG-GPUs. The computational workload of the main loop in DMP methods is dominated by matrix-matrix multiplications. They represent the computational bottleneck of DMP methods as they scale $O(N^3)$ in dense linear algebra. However, since density matrices have larger numerical values on their diagonal entries than their off-diagonal, this motivates separating both parts and representing them in different numerical precisions. An original density matrix stored in FP64 can be approximated as the sum of its diagonal values stored in FP64 $\mathbf{p}_{\text{FP64}}$ and its off-diagonal values as a FP32 matrix (see eq. 5.6). Taking the square of this sum yields 4 terms according to eq. 5.7.

$$\mathbf{P}_{\text{FP64}}^2 \approx \left(\mathbf{p}_{\text{FP64}} + \bar{\mathbf{P}}_{\text{FP32}} \right)^2 \quad (5.6)$$

$$\approx \underbrace{(\mathbf{p}^2)_{\text{FP64}}}_{O(N)} + \underbrace{(\mathbf{p}\bar{\mathbf{P}} + \bar{\mathbf{P}}\mathbf{p})_{\text{FP32}}}_{O(N^2)} + \underbrace{(\bar{\mathbf{P}}^2)_{\text{FP32}}}_{O(N^3)} \quad (5.7)$$

The first and numerically largest term is the element-wise product of $\mathbf{p}$ with $\mathbf{p}$. This term scales linearly with the number of basis functions and is computed using FP64 arithmetics. The second and third term arise from the cross term, a kernel implementing eq. 5.8 has been published in the context of ref. [137]. From eq. 5.8, we can see that this operation ensures the symmetry of the resulting matrix if the input matrices are distinct.

$$\begin{aligned} & (\mathbf{m}_1 \bar{\mathbf{M}}_2 + \bar{\mathbf{M}}_1 \mathbf{m}_2) \\ &= \mathbf{R} \begin{cases} R_{ij} = \frac{1}{2}(m_{1,j} + m_{1,i}) \cdot M_{2,ij} + \frac{1}{2}(m_{2,j} + m_{2,i}) \cdot M_{1,ij} & \text{if } \mathbf{M}_1 \neq \mathbf{M}_2 \\ R_{ij} = (m_j + m_i) \cdot M_{ij} & \text{if } \mathbf{M}_1 = \mathbf{M}_2 = \mathbf{M} \end{cases} \quad (5.8) \end{aligned}$$

The last term and also computational bottleneck is the matrix-matrix multiplication of the off-diagonal matrix $\bar{\mathbf{P}}$. The GEMM operation followed by an explicit symmetrization is found to be faster than the symmetric variant (SYMM) for the used cuBLAS release (11.8). After executing these steps to approximate a FP64 matrix-matrix mul-

tiplication and before entering the next purification cycle, the off-diagonal part $\bar{\mathbf{P}}_{\text{FP32}}$ must be reset to a diagonal of 0 and the values accumulated on its diagonal are added to $\mathbf{p}_{\text{FP64}}$. This approximation reduces the number of FP64 operations while trying to conserve the numerical accuracy.

As mentioned before, the DMP method starts from a Fock matrix in the orthogonal basis. Most SQM methods, including GFN2-xTB, build Fock matrices in the non-orthogonal AO basis $\tilde{\mathbf{F}}$. The basis transformation from $\tilde{\mathbf{F}}$ to the orthogonal basis Fock matrix $\mathbf{F}$ is described by the following equation:

$$\mathbf{F} = \mathbf{X}^T \tilde{\mathbf{F}} \mathbf{X} \quad (5.9)$$

Various transformation matrices $\mathbf{X}$ can be used, in this application we use the coefficient matrix $\mathbf{C}$ from solving the Roothaan-Hall equations in the first SCF cycle. The details of how this coefficient is computed and used are the subject of sec. 5.3.1. After converging the density matrix by means of a DMP scheme, $\mathbf{P}$ needs to be transformed to the non-orthogonal AO basis variant $\mathbf{D}$.

$$\mathbf{D} = \mathbf{X} \mathbf{P} \mathbf{X}^T \quad (5.10)$$

In summary, the basis transformation steps require another 4 GEMMs in FP64. After implementing the MP scheme to compute polynomials of $\mathbf{P}$ as shown above, preliminary results showed that the transformation was at that stage the computational bottleneck. A change to FP32 to compute this step is not viable as the Fock matrix presents too large numerical values. However, the numerical values can be reduced by using an incremental scheme, inspired by the incremental Fock build.^[70] Therefore, reference Fock matrices in the non-orthogonal AO basis $\tilde{\mathbf{F}}_{\text{ref}}$ and orthogonal basis $\mathbf{F}_{\text{ref}}$ are defined. The Fock matrix $\tilde{\mathbf{F}}_i$ of the current SCF iteration i can therefore be defined as a sum of the reference matrix $\tilde{\mathbf{F}}_{\text{ref}}$ and its increment $\Delta\tilde{\mathbf{F}}_i$ (see eq. 5.11). Using this partition of the AO Fock matrix, the transformation into the orthogonal basis can be separated into the transformation of the reference and its increment, see eq. 5.12. By storing the reference orthogonal Fock matrix $\mathbf{F}_{\text{ref}}$, only the transformation of the increment needs to be computed every SCF cycle.

$$\tilde{\mathbf{F}}_i = \tilde{\mathbf{F}}_{\text{ref}} + \Delta\tilde{\mathbf{F}}_i \quad (5.11)$$

$$\mathbf{F}_i = \mathbf{X}^T \tilde{\mathbf{F}}_i \mathbf{X} = \mathbf{X}^T \tilde{\mathbf{F}}_{\text{ref}} \mathbf{X} + \mathbf{X}^T \Delta\tilde{\mathbf{F}}_i \mathbf{X} \quad (5.12)$$

$$= \mathbf{F}_{\text{ref}} + \Delta\mathbf{F}_i \quad (5.13)$$

$$\text{where: } \tilde{\mathbf{F}}_{\text{ref}} = \tilde{\mathbf{F}}_1 + \Delta\tilde{\mathbf{F}}_2 + \cdots + \Delta\tilde{\mathbf{F}}_{i-1}$$

$$\text{and: } \mathbf{F}_{\text{ref}} = \mathbf{F}_1 + \Delta\mathbf{F}_2 + \cdots + \Delta\mathbf{F}_{i-1}$$

Updating the reference matrices every cycle with the increment as shown above, ensures that the numerical values of the current cycle's increment remain small. A treatment of these small numerical values using FP32 arithmetics is not affecting the overall numerical accuracy considerably. Hence, only the first transformation is computed in

FP64 and all subsequent increment transformations are performed in FP32:

$$\mathbf{F}_{\text{FP64},i} \approx (\mathbf{X}^T \tilde{\mathbf{F}}_1 \mathbf{X})_{\text{FP64}} + (\mathbf{X}^T \Delta \tilde{\mathbf{F}}_2 \mathbf{X})_{\text{FP32}} + (\mathbf{X}^T \Delta \tilde{\mathbf{F}}_3 \mathbf{X})_{\text{FP32}} + \cdots + (\mathbf{X}^T \Delta \tilde{\mathbf{F}}_i \mathbf{X})_{\text{FP32}} \quad (5.14)$$

To preserve the numerical accuracy, only the increments are stored using the FP32 representation, while the reference values remain in FP64. For brevity, we refer to the transformation from AO to orthogonal basis as AO-MO transform and MO-AO transform refers to the transformation from the orthogonal to the non-orthogonal basis.

$$\Delta \tilde{\mathbf{F}}_{\text{FP32},i} = \tilde{\mathbf{F}}_{\text{FP64,ref}} - \tilde{\mathbf{F}}_{\text{FP64},i} \xrightarrow[\text{FP32}]{\text{AO-MO Transform}} \mathbf{F}_{\text{FP64},i} = \mathbf{F}_{\text{FP64,ref}} + \Delta \mathbf{F}_{\text{FP32},i} \quad (5.15)$$

The same incremental scheme is applied to the MO-AO transformation of the density matrix from $\mathbf{P}_i$ to $\mathbf{D}_i$:

$$\mathbf{P}_i = \mathbf{P}_{\text{ref}} + \Delta \mathbf{P}_i \quad (5.16)$$

$$\mathbf{D}_i = \mathbf{X} \mathbf{P}_i \mathbf{X}^T = \mathbf{X} \mathbf{P}_{\text{ref}} \mathbf{X}^T + \mathbf{X} \Delta \mathbf{P}_i \mathbf{X}^T \quad (5.17)$$

$$= \mathbf{D}_{\text{ref}} + \Delta \mathbf{D}_i \quad (5.18)$$

$$\text{where: } \mathbf{P}_{\text{ref}} = \mathbf{P}_1 + \Delta \mathbf{P}_2 + \cdots + \Delta \mathbf{P}_{i-1}$$

$$\text{and: } \mathbf{D}_{\text{ref}} = \mathbf{D}_1 + \Delta \mathbf{D}_2 + \cdots + \Delta \mathbf{D}_{i-1}$$

Also here, the lowering of numerical values based on the incremental scheme motivates the transformation of the increments using FP32 arithmetics.

$$\mathbf{D}_{\text{FP64},i} \approx (\mathbf{X} \mathbf{P}_1 \mathbf{X}^T)_{\text{FP64}} + (\mathbf{X} \Delta \mathbf{P}_2 \mathbf{X}^T)_{\text{FP32}} + (\mathbf{X} \Delta \mathbf{P}_3 \mathbf{X}^T)_{\text{FP32}} + \cdots + (\mathbf{X} \Delta \mathbf{P}_i \mathbf{X}^T)_{\text{FP32}} \quad (5.19)$$

Adding the increments to the reference matrices and to the final $\mathbf{D}_{\text{FP64},i}$ is performed in FP64.

$$\Delta \mathbf{P}_{\text{FP32},i} = \mathbf{P}_{\text{FP64,ref}} - \mathbf{P}_{\text{FP64},i} \xrightarrow[\text{FP32}]{\text{MO-AO Transform}} \mathbf{D}_{\text{FP64},i} = \mathbf{D}_{\text{FP64,ref}} + \Delta \mathbf{D}_{\text{FP32},i} \quad (5.20)$$

This incremental AO-MO scheme reduces the number of FP64 operations greatly. The effect of both MP approximations on the numerical accuracy of single-point total energies and nuclear gradients will be evaluated in sec. 5.4.

Finkelstein^[105] and Dawson^[101] already showed how FP16 split schemes (see sec. 3.3) can be combined with the SP2 method to shorten the walltime while conserving numerical accuracy. Based on the recent developments in LAHVA shown in sec. 4.4.4, the FP16 split method presented therein can be combined with SP2. Our original SP2 publication^[137] concluded that the initial AO-MO and MO-AO transforms (in FP64) are contributing significantly (> 10 %) to the overall runtime. This computational step is, hence, a good target for acceleration using FP16 splitting techniques. The splitting parameters used in sec. 4.4.4 are used in this chapter as well. Hence, the FP64 matrices will be split into four FP16 matrices and the ten combinations with the largest numerical contributions are considered. Finkelstein *et al.* also applied the Markidis

scheme during the recursive polynomial build step of SP2. They proposed to neglect the smallest numerical contributions from the splitting scheme, i.e. A_2xB_2 using our notation from fig. 3.3. To counteract the loss of accuracy introduced by this approximation, a refinement step corresponding to the combination of two recursive cycles in FP32 or FP64 is introduced. We applied the same strategy as Finkelstein using our adapted Ozaki scheme (see sec. 4.4.4) and MP approximation for the matrix-matrix-product (see eq. 5.7). Thereby, the off-diagonal FP32 matrix $\bar{P}_{\text{FP32}}$ is split into two FP16 matrices using our Ozaki-type scheme from sec. 4.4.4. Instead of using a refinement step, the idempotency error $|\text{tr}(\mathbf{P}) - \text{tr}(\mathbf{P}^2)|$ is used as an indication of how close the DMP cycle is to convergence. For idempotency errors larger than 1.0, the smallest combination of FP16 matrices, i.e. A_2xB_2 , is also neglected. If the idempotency error gets smaller, the polynomial is constructed using plain FP32 GEMM routines without FP16 splitting. These approximations may also affect the numerical accuracy of the energies and gradients. This impact will also be investigated in sec. 5.4.

5.3 Computational details

The DMP library is bundled as a larger package called GAMBITS: (Gaming)GPU-Accelerated Matrix-Based Iterative Solvers.^[131] LAHVA is the foundation of the speed and portability of this package, especially regarding the asynchronous runtime shown in sec. 5.6. LAHVA and GAMBITS are linked against the CUDA 11.8 runtime, cuBLAS as well as the cuSOLVER library. Hand-written kernels are compiled with `nvcc` from the same CUDA release.

GAMBITS is developed as reusable library, that can be used as a drop-in alternative to LAPACK-based solvers. For the applications shown in this work, GAMBITS is linked to `tblite`,^[111,130] giving access to the GFN2-xTB Hamiltonian. The `tblite` version used for this work is commit 1346c78 of the “tc-purification” branch of the author’s public `tblite` fork. The GAMBITS version is commit ff807c8a of the “half-precision” branch. Lastly, commit 3b69eb04 of the “half-precision” branch of LAHVA is used.

For use in conformational sampling, this altered `tblite` version is then used in a development version of CREST^[8,174] based on v.3.0.2 as a subproject. Minor alterations to CREST are then needed to select the proper solver and optimize the parallel execution. Those changes can be found as commit 5f4c9121 in the “purification” branch on the BannwathLab CREST fork.^[175]

For assessment of the numerical performance and the computational efficiency, we use differently sized fullerene systems taken from Ref. 136. For our single-point timings, the system sizes considered range from rather small systems like C_{42} over medium-sized C_{240} to C_{720} . For the conformer sampling timings with CREST, alanine polypeptides with N-methylated and acetylated termini are used. Though the DMP formalism can be combined with arbitrary basis sets, we used the minimal valence-only basis

set^[21] in combination with GFN2-xTB as a practical use case. The basis set for GFN2-xTB uses 4 basis functions for C, N and O and 1 for H. Only using monomers would limit us to about 2880 basis functions for C_{720} . Hence, we constructed different multimers of C_{720} and C_{240} with $(C_{240}@C_{720})_5$ being the largest system amounting to a total of 19,200 basis functions.

5.3.1 Implementation details

The workflow of the MP DMP implementation will be covered in this subsection. Fig. 5.3 summarizes the individual steps. As discussed in the previous sections, the DMP algorithm operates in the orthogonal basis. Several transformation matrices X can be used to perform the basis transformation, a common choice is the Löwdin orthogonalization, i.e. $S^{-1/2} = X$. Another choice is the coefficient matrix $\tilde{C}$, which is also obtained solving the Roothaan-Hall equations via diagonalization of a generalized eigenvalue problem. However, solving the Roothaan-Hall equations also gives the energy and a new density matrix as opposed to Löwdin orthogonalization, thus, allowing to move one step further in the SCF iterations. Aiming for faster SCF convergence, the approach of using the coefficient matrix has been chosen. GAMBITS also implements an interface to the generalized eigenvalue equation solver DSYGVD for GPUs via NVIDIA’s cuSOLVER library. Depending on the system size, the first SCF cycle is solved by the CPU or GPU DSYGVD call. In both cases, the coefficient matrix is transferred via `tblite` to the GAMBITS library. The second SCF cycle is then the first where the density is built using a DMP algorithm. This includes the first AO-MO transform that is executed using FP64 arithmetics or approximated using the FP16-split algorithm. The reference Fock matrices in the AO basis $\tilde{F}_{FP64,ref}$ and MO basis $F_{FP64,ref}$ are stored. The transformed Fock matrix is then used to generate the guess density by means of eq. 5.4. According to the approximation in eq. 5.6 the density is split in its diagonal and off-diagonal parts. In the main recursive loop, eq. 5.7 is used to build purification polynomials. The converged density matrix P undergoes a transformation back to the non-orthogonal AO basis to be used in `tblite`. The workflow for the following SCF cycles is similar, except for the basis transformations, which use the incremental scheme introduced in sec. 5.2.2. If the DMP algorithm does not converge within 50 recursive cycles, the DSYGVD algorithm is used as a failsafe alternative solver for the remaining SCF.

The software design philosophy presented in the LAHVA chapter is inherited to GAMBITS. Thereby, an object-oriented approach is used for implementation, creating an object for every major operation in the workflow. These operations are: basis transformation, generating the guess density, the recursive main loop of the DMP method and computing polynomials of the density matrix. By separating these functionalities various floating point representations can be tested easily, as switching between

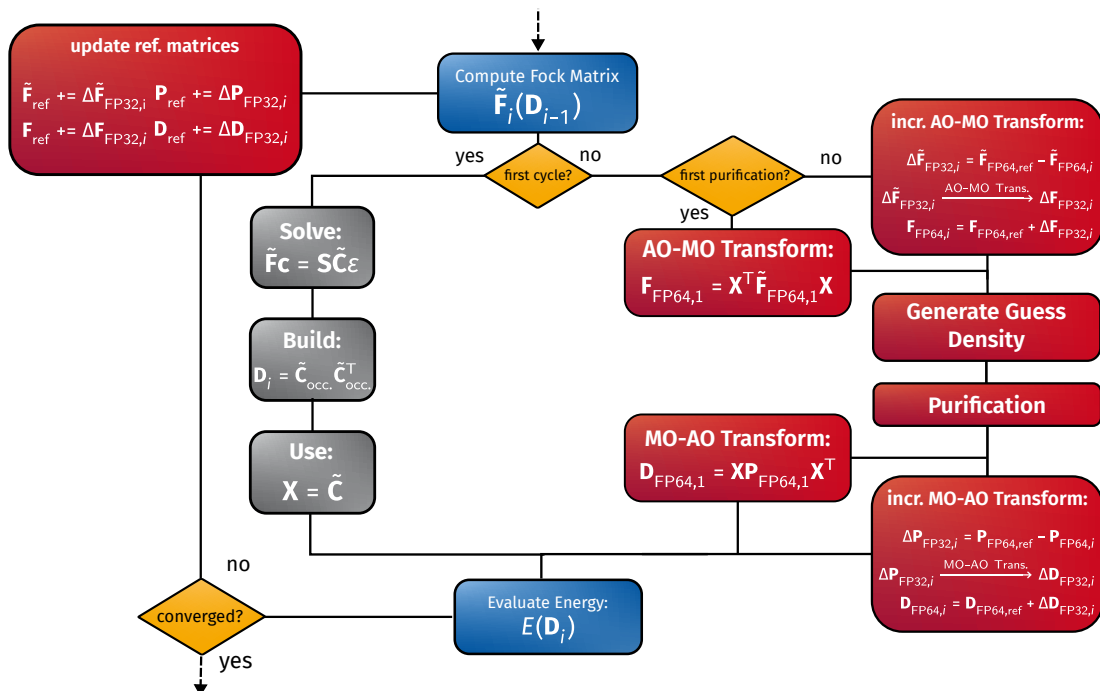


Fig. 5.3: Core workflow to compute a new density from the Fock matrix, using a hybrid approach of one diagonalization combined with our mixed-precision SP2 DMP implementation and incremental mixed-precision basis transformations. For brevity, we refer to “AO-MO Transform” as the basis transformation from the non-orthogonal AO basis to the orthogonal basis and vice versa for “MO-AO Transform”. It should be stressed that does not imply a transformation to or from the canonical orbital basis. **Reproduced with permission from: Pit Steinbach, Christoph Bannwarth, and Thorsten Bach Journal of the American Chemical Society 2025, 21, 7335–7351. DOI: 10.1021/acs.jctc.5c00262. Copyright 2025 American Chemical Society.**

FP64, FP32 and MP is straightforward due to LAHVA. LAHVA also simplified the implementation of an asynchronous non-blocking runtime necessary to run several SCF calculations in parallel.

5.4 Numerical accuracy evaluation

Since multiple approximations have been introduced in the course of this chapter, the influence of these on the numerical accuracy must be investigated. The DSYGVD method from LAPACK is used as a reference method for the total energy and nuclear gradients. From our previous experiences the Intel® Math Kernel Library (MKL) implementation is the fastest on Intel® hardware. Hence, it also serves as a reference for the timing comparisons shown in the next subsection. As eluded to in the introduction section of this chapter, DMP methods give the density matrix for a Fermi temperature of 0 K. Therefore, all GFN2-xTB calculations shown in this chapter have been performed at 0 K.

In fig. 5.4, the absolute difference in single-point energies between SP2 DMP and DSYGVD calculations for fullerene systems of various sizes is compared. For the SP2 GPU FP32 and FP64 values, the basis transform calculations are performed in FP64 and without the use of increments. For the SP2 GPU MP scheme the incremental scheme is used for the AO-MO transform and vice versa. The first transform is executed using FP64 arithmetics. The polynomial builds are realized using FP32 GEMM calls. The SP2 GPU MP (FP16) model uses the same overall workflow as the MP version, however the first transform is approximated using our Ozaki-style FP16 split version. Similarly, the polynomial building process is also approximated using a FP16 splitting scheme ignoring the smallest contribution until the idempotency error is below 1.0 when FP32 GEMM calls are used again.

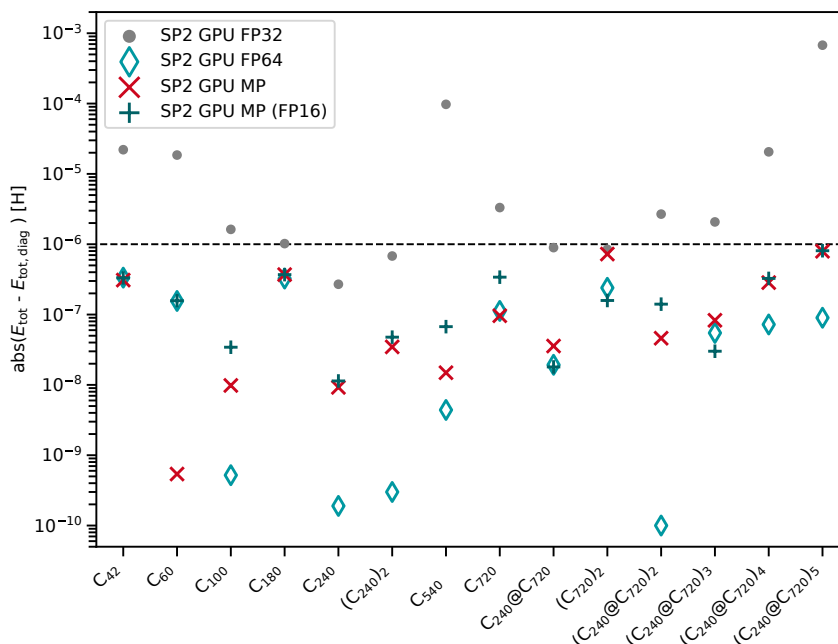


Fig. 5.4: Absolute difference between the total energy computed with various precision schemes and hardware and the total energy computed using the MKL DSYGVD algorithm. For those values where the result is identical within the numerical precision displayed by tblite a small value of 10^{-10} is added to make them visible for completeness.

For the comparison shown in fig. 5.4, the default energy threshold of $1.0 \cdot 10^{-6}$ H is used in the SCF procedure. The SP2 FP32 variant shows large deviations from the DSYGVD reference energies especially for the largest systems. The full double precision version shows the lowest deviations from DSYGVD for most test molecules. However, the computation of these values using FP64 arithmetics takes a lot of computation time on CG-GPUs. The accuracy of the MP methods is located between FP32 and FP64 methods and allows the computation of the energy within the SCF threshold. For the largest system $(C_{240}@C_{720})_5$, the deviation is close to the energy threshold and identical for both MP models. For this system size, we were not able to use the FP16 splitting scheme as there is not enough memory to apply the scheme in its current version. In general, the FP16 values are close to the other MP values, sometimes even lower devi-

ations are reported these can probably be linked to error cancellation in the numerical splitting procedure.

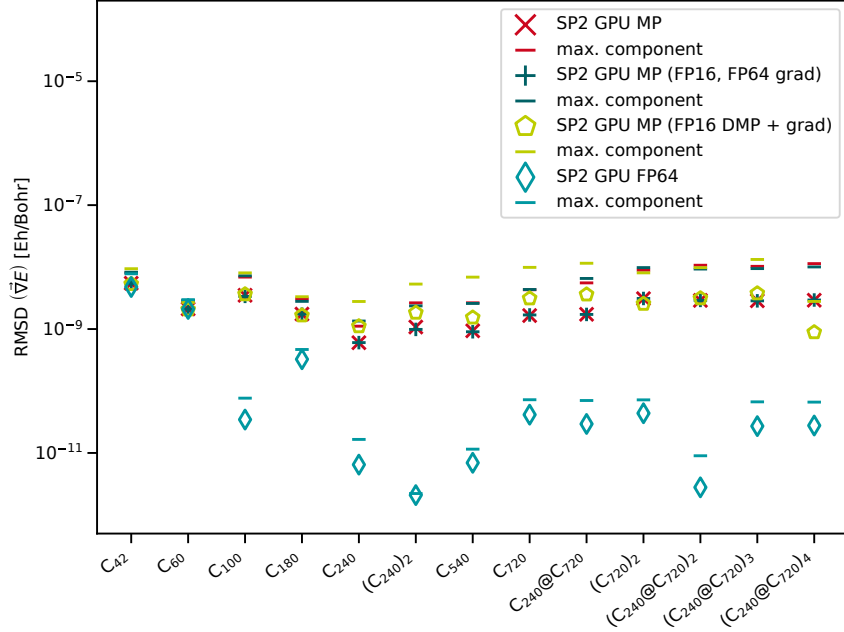


Fig. 5.5: Root mean square deviation (RMSD) of the nuclear gradients computed using SP2 with various precision schemes and hardware relative to the nuclear gradients computed using the MKL DSYGVD algorithm. Additionally, the component that deviates the most is shown.

After establishing that both MP schemes present lower deviations in energy than a pure FP32 treatment, the effect on nuclear gradients is investigated. The nuclear gradients are generally computed based on the orbital energies, orbital coefficients, and the orbital occupations. However, since DMP methods construct the density matrix without solving for the orbitals first, this construction of the nuclear gradients cannot be used. Instead, the so-called energy-weighted density matrix $\mathbf{W}$ is computed according to the following equation:

$$\mathbf{W} = \mathbf{X}\mathbf{X}^T\tilde{\mathbf{F}}\mathbf{D} \quad (5.21)$$

DSYGVD values are used as a reference to compute the root mean square deviation (RMSD) of the gradient. In addition to the RMSD, the largest deviation of a gradient component is also shown in fig. 5.5.

The SP2 FP32 results are not shown as they are several orders of magnitude larger than the MP and FP64 values as demonstrated in ref. [137]. Instead, the focus is shifted to the MP scheme variants. The construction of the gradient is performed using three FP64 GEMMs, however these can also be approximated with the Ozaki-style FP16 splitting scheme presented in sec. 4.4.4. Using the FP16 splitting only in the DMP process leads to no noticeable deviation of the gradients from the MP non-split values

(see fig. 5.5). Further approximating the gradient build using FP16 splitting together with the Fock and density matrix build from DMP with FP 16 as well leads to negligible deviations for a few molecules. However, especially for large molecules, no large deviations can be noted. In the remaining subsection, the effect on longer running MD simulations will be investigated. Additionally, the FP16 DMP + grad variant will be tested in a geometry optimization workload.

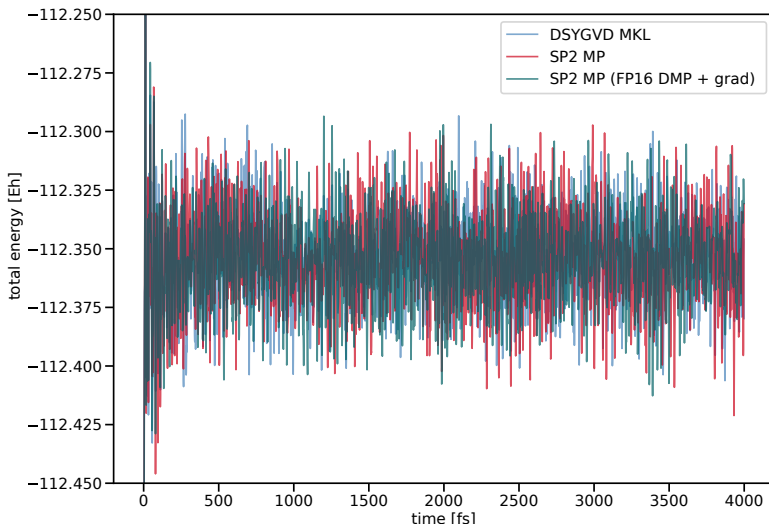


Fig. 5.6: Total energy along an NVE MD simulation of 2.0 ps with 0.5 fs timesteps using GFN2-xTB. ALPB for water is used as a solvent model to stabilize the triazolium-based photoswitch with a charge of +1. All NVE MDs are started from the same restart file of an NVT MD (using DSYGVD MKL) used for equilibration.

In fig. 5.6, the total energy of a triazolium-based photoswitch^[12] (charge +1) is plotted along a NVE MD trajectory (2 ps with 0.5 fs time steps). The molecule was equilibrated by means of a NVT MD simulation at 298.15 K for 1 ps. The final snapshot from that NVT MD run is used as initial condition in all NVE MDs. The analytical linearized Poisson-Boltzmann (ALPB) solvation model^[176] for water is added to stabilize the charged molecule.

Tab. 5.1: Mean values $\bar{E}_{\text{tot}}$ and Bessel-corrected standard deviations $\sigma(E_{\text{tot}})$ of the average total energy over 30 NVE MD runs (2.0 ps with 0.5 fs timesteps). All MDs are started from the same restart file of an NVT MD (using DSYGVD MKL) used for equilibration. SP2 MP FP16 is referring to using the FP16 splitting scheme for energy and gradient calculations.

Solver	$\bar{E}_{\text{tot}}$ [H]	$\sigma(E_{\text{tot}})$ [H]
MKL DSYGVD	-112.082341	0.003152
SP2 MP	-112.083069	0.003907
SP2 MP FP16	-112.083232	0.005033

The values in fig. 5.6 do not show a noticeable shift in total energy between the methods. However, to compare the methods quantitatively and also capture the non-deterministic effect of MDs 30 consecutive MD runs based on the same initial conditions are computed. In tab. 5.1, the mean value of the average total energy and its Bessel-corrected standard deviations are reported.

The mean total energies of both SP2 MP variants are very close, while the difference to the MKL DSYGVD mean value is larger. However, all values lie within the respective standard deviations. The standard deviation of the SP2 MP FP16 solver is larger than for the other methods. However, for the metadynamics runs, used in CREST’s conformer searching algorithm, or NVT MD simulations this error is irrelevant. In the next subsection, the performance of our MP methods in terms of computational time savings will be explored.

5.5 Timings for single-point calculations and geometry optimizations

5.5.1 Single-point calculations

After ensuring the numerical fidelity of our MP DMP methods, the speed-up realized by these methods compared to CPU diagonalization is investigated. The same fullerene systems as in subsection 5.4 are used to explore the scaling of the computation time with the number of basis functions. In ref. [137], the number of CPU cores was also varied, however, we will use calculations with 16 threads and focus on comparing the MP scheme with and without FP16 split. Since tblite uses an elaborate caching system, we repeat the calculation 6 times and average the runtimes over the last 5 runs. It should be noted that this work only accelerates the density build process and all other operations remain unchanged from the main branch of tblite. Therefore, we report the speed-up of the density build and the overall speed-up separately, see fig. 5.7 and 5.8.

For all GPU implementations, the 100 ms setup time dominates the runtime for small molecules up to 1000 basis functions. Around that point, a speed-up for offloading calculations to the GPU can be recorded. The smallest speed-up is observed for the cuSolver DSYGVD implementation. This can be related to the use of a CG-GPU for computation, which features suboptimal FP64 performance. For the investigated systems and hardware, a maximal speed-up of factor 3.5 can be reported for $(C_{240}@C_{720})_4$ with cuSolver DSYGVD. All SP2 implementations presented are faster than NVIDIA’s highly optimized cuSolver routine.

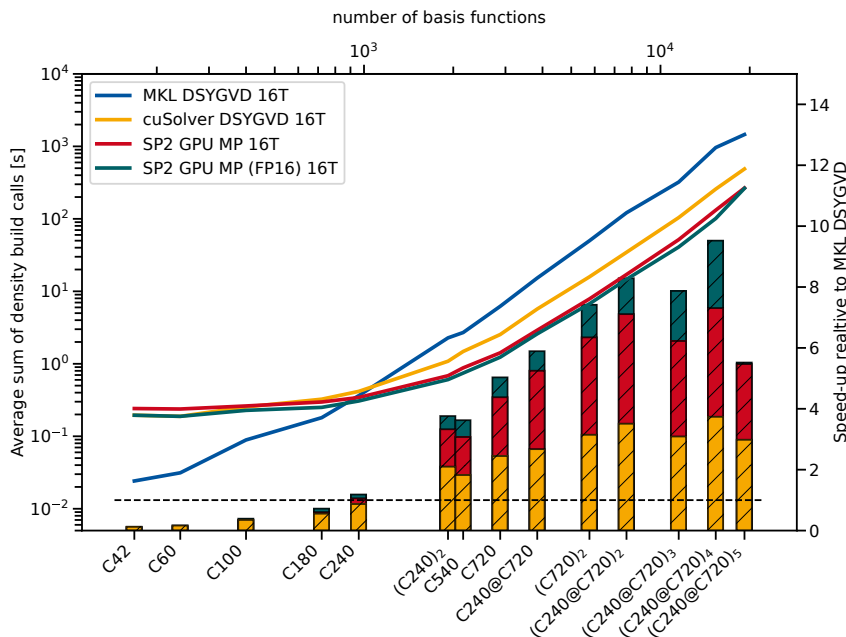


Fig. 5.7: Sum of density build walltimes over all SCF cycles of a GFN2-xTB single-point calculation for the shown molecule averaged over 5 consecutive runs. The GPU walltimes also include basis transformations, the initial diagonalization at the first iteration, as well as setup and shutdown of the purification C++ object. The MKL DSYGVD runtime serves as a reference for the relative speed-up and is compared to the cuSolver GPU DSYGVD solver, the SP2 MP scheme with and without FP16 splitting. All calculations are performed on 16 threads and 1 RTX 4090 on a system of type 1 (see tab. 8.1). Only one CUDA stream is used for the GPU calculations.

The addition of the FP16 Ozaki-style splitting scheme is advantageous for all system sizes. For $(C_{240}@C_{720})_4$ the reduction in walltime compared to the “regular” SP2 MP scheme is the largest. Underlining that this splitting scheme is a powerful tool to partially alleviate the CG-GPUs weak FP64 performance, especially, for large system sizes. For small system sizes, the overhead of splitting the matrices and summing them up again is a substantial computational cost. The highest speed-up up to an order of magnitude is reported for $(C_{240}@C_{720})_4$ compared to MKL DSYGVD in the density build. The drop after that molecule can be related to the necessity to copy back some matrices to the host as the GPU’s memory is too small to store all of them. For $(C_{240}@C_{720})_5$ we are currently unable to use the FP16 split scheme as the memory requirements then exceed the capacity of the GPU used. However, we check if the matrices fit before potential FP16 GEMM calls and if there is not enough memory the corresponding non-FP16 GEMM call is used instead. This is also why a datapoint is shown for $(C_{240}@C_{720})_5$ in the SP2 MP FP16 dataset in fig. 5.7 and why it is identical to the SP2 MP value.

The speed-up seen in the density build timings in fig. 5.7 is not fully transferred to the overall runtime as the remaining unaltered program makes up a significant portion of the runtime (see fig. 5.8). For the overall runtime, the largest improvement of SP2 MP FP16 over the already very good speed-up with SP2 MP is also found for $(C_{240}@C_{720})_4$, going from factor 5 to factor 6. Achieving a further speed-up would re-

quire to move the Hamiltonian build to the GPU as well.

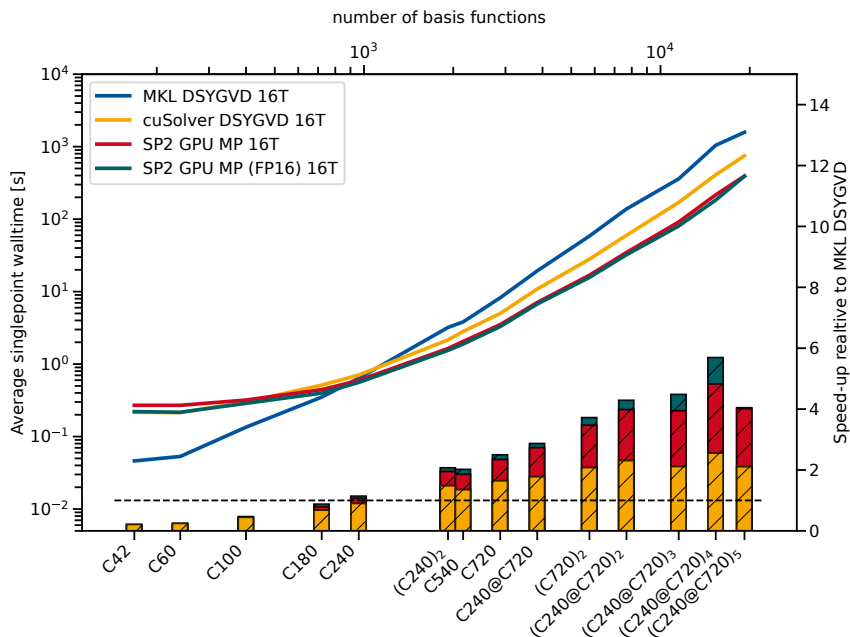


Fig. 5.8: Total walltimes of a GFN2-xTB single-point calculation for the shown molecule averaged over 5 consecutive runs. The total walltime as determined by `tblite` is used for various molecules. All calculations are performed on 16 threads and 1 RTX 4090 on a system of type 1 (see tab. 8.1).

In ref. [137], we could show that the number of SCF cycles for the SP2 and DSYGVD solver are the same except for the plain FP32 version. For the latter implementation, more SCF steps are needed than for the MP variant with the default threshold of 10^{-6} H. The use of the FP16 splitting scheme has also no effect on the number of SCF cycles needed for convergence.

In fig. 5.9, the percentage of runtime spent in a certain subsection of the DMP library is shown for the SP2 MP scheme without (fig. 5.9a) and with FP16-split (fig. 5.9a).

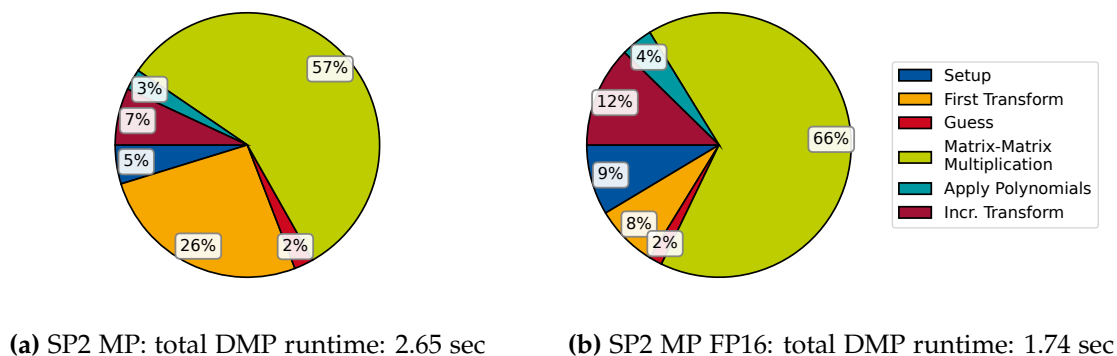


Fig. 5.9: Runtime composition among various algorithmic sections of the SP2 DMP algorithm. The total DMP runtime refers to the sum of all DMP cycles of one single-point calculation for C_{720} on system 1 hardware using 8 CPU threads, see tab. 8.1. (a) SP2 MP implementation without FP16 split. (b) SP2 MP with Ozaki-type FP16 splitting.

As expected, the total runtime is lower for the calculation with FP16 split. This is mainly the result of a reduced runtime for the first AO-MO transform, performed at FP64 precision for fig. 5.9a. The use of the FP16 splitting scheme reduces the absolute runtime for this step by 0.5 seconds. The remaining reduction comes from the use of the FP16 scheme for the symmetric matrix-matrix product P^2 neglecting the contribution the smallest split matrices (see sec. 5.2.2).

After optimizing the SP2 MP runtime even further with the FP16 scheme, the effect of that optimization for the geometry optimization of a protein is investigated in the next section.

5.5.2 Geometry Optimization

The previous section showed that the target size for our SP2 MP scheme are large molecules or clusters thereof with more than 1000 basis functions. A common application for this size range is the geometry optimization of proteins. Hence, the geometry of the photoactive yellow protein^[177] (4922 basis functions) is relaxed using the GFN2-xTB Hamiltonian. Due to the high charge of -4, the implicit solvation model ALPB^[176] for water is also added. `tblite` only provides functionality to compute the energy and gradients using GFN-xTB methods. Therefore, CREST is used as a driver for the geometry optimization using a modified `tblite` version that is, in turn, connected to GAMBITS. The default convergence thresholds of CREST and an electronic temperature of 0 K are used.

In table 5.2, the runtimes for various solver implementations are summarized. The MKL DSYGVD algorithm is used as a reference to report the relative speed-up of the GPU-accelerated solvers. In addition to the first AO-MO transform and the matrix-matrix multiplication of the density matrix, the FP16 split scheme can also be used to compute the energy-weighted density (see eq. 5.21). In this step, the 3 FP64 GEMMs

can be substituted by the FP16 split scheme equivalent using 4 splits per input matrix, while keeping only the 10 combinations with the highest numerical values (see sec. 4.4.4). The effect of this substitution is reported separately. The number of optimization steps needed for convergence shows slight deviation with the solver. However, the optimization process is not fully deterministic, and even for cuSolver DSYGVD the number of steps deviates slightly from MKL DSYGVD. The speed-up reported is not further scaled with number of cycles.

Tab. 5.2: Comparison of geometry optimization runtimes of the photoactive yellow protein (4922 basis functions) using the GFN2-xTB Hamiltonian with the implicit ALPB solvation model for water. Default numerical convergence thresholds of CREST are used. The calculations are executed on 8 CPU threads and one RTX 4090 of a system 2 hardware, see tab. 8.1. The walltime is reported with the relative speed-up to the MKL DSYGVD CPU solver. The speed-up has not been scaled in relation to the number of optimization steps.

Solver variation	Total walltime [min]	Number of optimization steps	speed-up relative to MKL DSYGVD
MKL DSYGVD	350	655	1.00
cuSolver DSYGVD	156	659	2.24
SP2 MP	136	665	2.57
SP2 MP (FP16 only energy)	130	665	2.69
SP2 MP (FP16 energy+grad)	120	657	2.92

The use of the GPU-accelerated DSYGVD algorithm already leads to a significant speed-up of factor 2.24. Using the SP2 MP scheme further reduces the walltime to give a speed-up factor of 2.57. Substituting GEMMs with the FP16 split scheme leads to further reduction over the “normal” SP2 MP scheme by 6 minutes and by further 10 minutes if the gradient calculation is also accelerated by FP16 splitting. The fastest solver implementation is thereby 2.92 times faster than the CPU-only MKL DSYGVD algorithm.

In fig. 5.10, the distribution of the computational time over the individual algorithmic steps is shown for the SP2 MP FP16 (energy+gradient) variant. The largest contribution is the computation of the non-selfconsistent dispersion energy and its nuclear gradients with 30 % of the overall runtime. Hence, the density build is no longer the computational bottleneck for this system and hardware in combination with the developed SP2 implementation. Another significant contribution is the Hamiltonian build time, which mainly refers to the EHT model, and its gradient. During every SCC cycle, the potential and the charges have to be computed, accounting for 12 % and 9 % of the overall walltime, respectively.

The conclusion of this runtime composition analysis is that the potential for accelera-

tion in the density build section has been nearly completely exhausted. The most promising remaining improvement would be to eliminate the necessity for the first diagonalization altogether. Numerical procedures to determine a transformation matrix, inspired by the work of Niklasson and coworkers,^[178] have been tested. However, the combination of this transformation matrix, derived from the overlap matrix, with the SP2 MP scheme presented in this work revealed numerical instabilities. Further testing could improve the stability, but the potential of offloading other parts of the energy and gradient calculation to the GPU are larger. Based on the presented analysis, the acceleration of the Hamiltonian build, particularly, the D4 dispersion model and its gradient would be the next targets for acceleration. Instead of targeting individual contributions to the Fock matrix and gradient, it would be beneficial to offload all components to the GPU by making use of the LAHVA package from ch. 4.

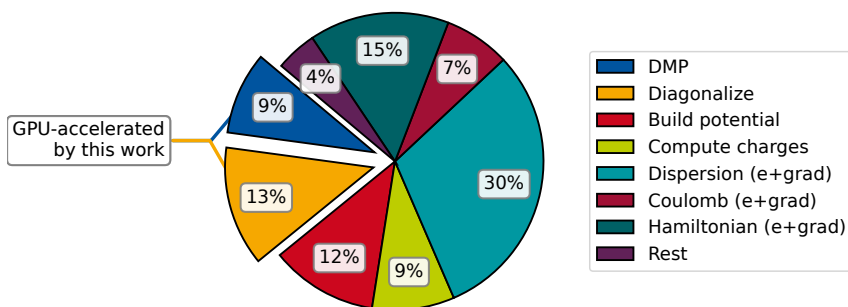


Fig. 5.10: Runtime composition of the tblite library for a geometry optimization of the photoactive yellow protein^[177] (charge -4) using GFN2-xTB/ALPB(water). 8 CPU threads and one RTX 4090 in a system of type 2, see tab. 8.1 The SP2 MP FP16 (energy+gradient) implementation is shown in this example. The DMP section includes all subsections of the DMP algorithm, such as the basis transforms, guess density calculation and iterative polynomial application. The sections dispersion, Coulomb, and Hamiltonian capture the time necessary to compute the non-selfconsistent part of the energy and the contribution to the nuclear gradients. The contributions to the self-consistent energy computation and the ALPB interaction are summarized as “Rest”.

The geometry optimization is the bottleneck of CREST’s conformer exploration algorithm iMTD-GC.^[7] Hence, this workload is a perfect example to demonstrate the acceleration provided by the SP2 MP algorithm over the default DSYGVD solver. Since, several SCCs are run in parallel during this workload, it allows us to present the asynchronous scheduling capabilities of LAHVA as presented in sec. 4.4.5.

5.6 Acceleration of a conformer sampling protocol

As discussed in the motivation, one of the key propositions of the multilevel workflow is that it captures the conformational flexibility of a system. This core step in the mul-

tilelevel workflow is commonly performed with CREST’s iMTD-GC workflow. However, alternatives also based on the GFN2-xTB Hamiltonian to evaluate the energy and gradient have been published.^[23,179] The implementation presented in this section can be readily transferred to those algorithms as well.

The key workflow segments of the iMTD-GC algorithm are a combination of a Cartesian atomic position RMSD-based bias potential MTDs^[22] with subsequent geometry optimizations and a genetic crossing step. With the CREST 3.0 release,^[8] the program directly interfaces to tblite for energy and gradient evaluation without the need to write and read files compared to earlier versions. Several SCFs are run asynchronously in parallel to reduce the overall runtime. This step is orchestrated by the OMP task directive, which provides separate data environments for every task. To achieve this execution behavior with the GPU-accelerated SP2 MP solver, each OMP task is executed in a separate CUDA stream, see sec. 4.4.5. This allows the GPU scheduler to combine each SCFs calculation in an optimal fashion. This has been made easily accessible through the LAHVA library.

Besides these developments in LAHVA, a few adaptations are made in tblite and

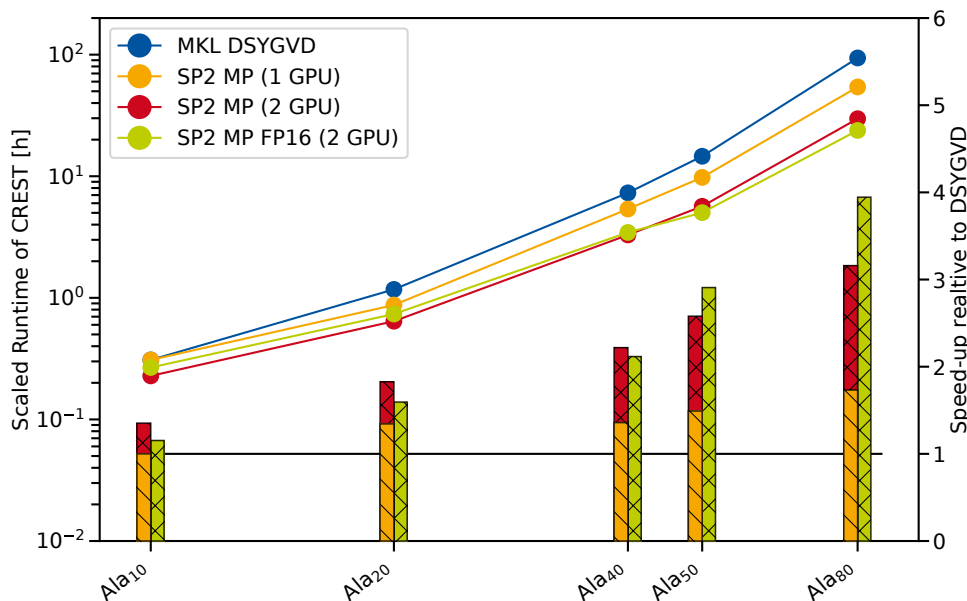


Fig. 5.11: Walltime of a CREST iMTD-GC run for the shown molecule with a fixed MTD of 5 ps. MP SP2 computed on the GPU is compared against the MKL DSYGVD algorithm with 32 SCF jobs with 2 threads each. The number of GPUs (1 or 2 RTX 3080Ti) is varied as well as the use of the FP16 splitting scheme within the SP2 MP algorithm. The speed-up of MP SP2 (FP16) compared to MKL DSYGVD is also shown. Due to memory constraints, the calculations for Ala₈₀ using SP2 MP and one GPU had to be limited to 16 SCFs and SP2 MP FP16 (2 GPUs) to 28 SCFs.

CREST to improve the performance of the asynchronous execution. The changes focus on minimizing the times that the DMP C++ object is deleted and recreated, since that would lead to synchronization of the entire GPU. Compared to ref. [137], LAHVA now also supports the execution on multiple GPUs. The ID of the OMP thread creating the LAHVA runtime instance is used to distribute the DMP and cuSolver DSYGVD instances equally among the GPUs. The number of GPUs can be easily regulated

through the use of the `CUDA_VISIBLE_DEVICES` environment variable, which is also commonly set by workload schedulers.

With this code infrastructure in place, the desired SCF solver model and the number of SCF tasks run in parallel can be set using CREST’s toml input. The scaling of the overall walltime is investigated with alanine-based polypeptides with *N*-methylated and acetylated termini and a chain length varying from 10 to 80 units (227-1977 basis functions). These calculations are run on a system 3 machine, see tab. 8.1. Ref. [137] showed that the number of SCF tasks should be chosen as high as possible, within the memory limits, for the best result. Therefore, 32 SCF calculations are run at most, during the MTD phase only 14 SCFs are running, while the maximal number is reached only during the geometry optimization phase. Besides testing for 1 and 2 GPUs we are also testing if the previously shown FP16 split leads to further acceleration in this setup.

To limit the runtime, the MTD length is fixed to 5 ps. The iMTD-GC algorithm is not fully deterministic, in order to compensate for this, the runtimes are rescaled according to the number of energy and gradient calls counted by CREST. The largest number of energy and gradient calls for one system size is selected and the runtimes are scaled to match this maximum number of calls. For the largest polypeptide Ala₈₀, the memory on one RTX 3080Ti is not sufficient for 32 SCFs, hence that value has been recorded with 16 SCFs each with 4 threads. The FP16 algorithm needs more memory due to the splitting, therefore only 28 SCFs in parallel could be sustained. The reference values for MKL DSYGVD are recorded with 32 SCFs with 2 threads each.

As expected based on the single-point results (see fig. 5.8), the speed-up relative to MKL DSYGVD increases with system size. Similar to ref. [137], there is no speed-up when using one GPU for the smallest polypeptide Ala₁₀. However, a relatively small speed-up can be observed when distributing the calculations on 2 GPUs. When increasing the system size, the speed-up for the one GPU runtime remains about constant, indicating that the GPU is saturated with the 32 parallel SCFs. This conclusion is further strengthened by the increased advantage of using 2 GPUs. For Ala₅₀, one GPU leads to a speed-up of about 1.5, whereas using 2 GPUs leads to a speed-up of about 2.8. Hence, a nearly ideal doubling of the speed-up with GPU count is observed. The use of the Ozaki-style FP16 split is only beneficial over the regular SP2 MP scheme for molecules larger than Ala₄₀. However, the overall highest speed-up is recorded for Ala₈₀ using two GPUs and the FP16 splitting. Hence, for this hardware setup the use of FP16 splitting techniques is paying off for molecules with about 500 atoms. With the addition of multi-GPU capability and FP16 splitting, the size of molecules that can be investigated with CREST’s iMTD-GC has been successfully extended. Additionally, other conformers-searching tools such as TTConf^[23] and GOAT^[179] can easily adopt this speed-up when using tblite to compute energy and gradients.

6 Combining low-cost electronic structure theory and low-cost parallel computing architecture

The contents of the following chapter have been previously published as P. Steinbach, C. Bannwarth, *Phys. Chem. Chem. Phys.* **2024**, 26, 16567–16578. The figures, and tables in this chapter have been reprinted and have been marked accordingly. The results are identical to the ones in the publication, the text has been adapted.

6.1 Introduction

After demonstrating how single-point and geometry optimizations can be accelerated by DMP schemes executed on CG-GPUs. Those individual computation steps have been combined in an asynchronous workflow in the CREST software that allows a faster exploration of the conformational space. The next step in the multilevel workflow (see fig. 1.1) is the reoptimization of the conformer ensemble candidates with more accurate methods. The semi-empirical methods used in the exploratory phase commonly fall behind in accuracy compared to DFT methods for example.^[39] However, since a significant number of structures needs to be optimized, a method with a good accuracy/computational cost ratio is needed. In sec. 2.1.3, we already introduced the 3c-methods by Grimme and coworkers that have been developed as “low-cost ab-initio” methods. The low computational cost comes from the use of small basis sets, while accuracy is retained through empirical correction methods that have been fitted for the combination of method and basis set explicitly.

This scheme was first applied to the Hartree-Fock method, where a minimal basis-set is combined with DFT-D3^[54] and gCP^[58] correction terms to give HF-3c.^[24] DFT-based variants followed soon covering every step of Jacob’s ladder with a GGA B97-3c,^[25] a hybrid-GGA PBEh-3c,^[6] a meta-GGA r2-SCAN-3c,^[10] and a range separated hybrid ω B97X-3c.^[11] The components of the methods implemented in this work will be discussed in detail in the following section. These composite methods have become very popular in well-established CPU-based quantum chemistry packages like ORCA^[112,113] and TURBOMOLE.^[180,181] These CPU-based quantum chemistry codes often rely on

approximations such as the RIJ^[26] to lower the scaling of the Coulomb-type ERIs (see sec. 2.1.1) and accelerate calculations. However, the computation of the exact non-local exchange (see sec. 2.1.1) is the computational bottleneck for hybrid functionals. The resolution-of-the-identity approximation cannot be applied to this term to lower the formal scaling. Instead, semi-numerical integration schemes have been derived to accelerate the computation of the exchange integrals.^[27,182–185] Most GPU-accelerated integral libraries make no use of these “standard” approximations. As discussed in section 3.1, load balancing and communication between GPU threads are important to the computational performance. The addition of approximation schemes increases the complexity of the individual ERI evaluations, which makes effective load balancing more complex. Therefore, many GPU implementations focus on reducing the effective scaling by using effective screening techniques as pointed out before. The integral library in the TeraChem codebase^[75,76] is a prime example for this strategy. Building on this infrastructure, the exchange-inclusive 3c-methods (HF-3c, PBEh-3c, and ω B97X-3c) were implemented in this quantum chemistry package in the course of the presented work.

Consistent with previous chapters, we target the computation on CG-GPUs instead of the significantly more expensive GP-GPUs (see tab. 1.1). The key technique to enable competitive ERI evaluations on CG-GPUs is the use of the “mixed precision” ERI computation handling introduced by Luehr *et al.*^[72] The details of this technique have been discussed in sec. 3.2. In short, a density-weighted Schwarz bound is used to estimate the numerical value of the Fock element. ERIs with smaller estimated numerical significance are computed using FP32 arithmetics. The numerical value of the density-weighted ERI elements is kept small by using the incremental Fock scheme, which employs the numerically smaller difference density. However, as pointed out before, most GPU integral codes are limited to basis functions of a certain angular momentum. For the TeraChem release used as the basis of this work, the maximal angular momentum is d . Furthermore, CG-GPUs usually offer less memory capacity, which is also a limiting factor for the maximum system size that can be computed. In summary, this chapter focusses on combining low-cost heterogeneous computing architecture with the low-cost Fock exchange-inclusive 3c methods.

6.1.1 Low-cost 3c composite methods with Fock exchange

In this work, we implemented the 3c-methods HF-3c, PBEh-3c, and ω B97X-3c in the TeraChem codebase. The parent methods HF, PBE0^[186] and ω B97X-V^[187] had to be modified slightly to create the composite methods. In tab. 6.1, these adaptations have been summarized:

Since all implemented methods are mean-field methods without non-local correlation kernels, London dispersion interactions are not captured in the electronic energy and

Tab. 6.1: Components of the 3c methods presented earlier and implemented in this work. **The following table has been previously published in Ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

Method	HF-3c ^[24]	PBEh-3c ^{[6], a)}	ω B97X-3c ^[11]
Basis	MINIX	def2-mSVP	vDZP
Fock Exchange [%]	100	42	16.7-100
Dispersion	D3(BJ) ^[54,55]	D3(BJ) ^[54,55]	D4 ^[57]
gCP ^[58]	✓	✓	x
^{a)} Additionally, three functional parameters were modified (w.r.t. the PBE functional).			

specifically fitted D3 and D4 dispersion correction models are used. HF-3c uses a minimal basis set, leading to basis set incompleteness and basis set superposition errors (BSSE). These errors are reduced by applying a geometrical counterpoise correction (gCP).^[58] The name 3c-methods was coined by this first method and its 3 correction terms. Although, future methods presented less correction terms, the add-on 3c is used throughout.

Moving to DFT-based methods, PBEh-3c is introduced as a variation of the well known PBE0^[186] functional with a high amount of non-local Fock exchange (42 %). Additionally, some parameters of the exchange and correlation functionals are adapted.^[186,188] A D3 dispersion correction including the three-body term and short-ranged damped gCP correction are added. A split valence basis set, mainly based on the def2-SVP set^[5] is used.

The lowest number of empirical correction terms is used by the latest 3c-method: ω B97X-3c. A newly developed valence-only double-zeta basis set with large-core effective core potential (ECP) is utilized. This basis set is further characterized by a large number of primitive basis functions and a low BSSE compared to other double-zeta valence basis sets.^[11] A correction using a fitted gCP model is thereby not necessary for this method. Furthermore, an adaptation of functional or range-separating parameters is not necessary, but the ECPs have been tailored to this basis set.

After introducing the computational details of our work, we will go over some implementation details and give an overview on the elements available in our implementation. Next, the average wall time per SCF cycle of our implementation compared to state-of-the-art CPU quantum chemical packages, ORCA^[112,113] and TURBO-MOLE^[180,181] is discussed.

6.2 Computational details

The TeraChem implementation of the 3c-methods is verified against the ORCA^[112,113] and TURBOMOLE^[180,181] programs. The performance evaluation is also carried out against those CPU codes. All ORCA calculations are performed with version 5.0.4 using defgrid3 and the default options if not specified otherwise. The TURBOMOLE 7.6.1 version is used with the dynamic grid m4 option. The symmetry is set to C1 for all molecules, furthermore the generation of the guess density by the define program is not included in the walltime reported for TURBOMOLE calculations. The 3c-methods implementation is added to TeraChem version 1.9. All calculations use dftgrid1, the superposition of atomic densities guess (guess sad), and the spherical basis representation unless stated otherwise.

TeraChem is built with compilers and linked against libraries from the CUDA v11.6 release, the Intel Classic Compiler, and its Math Kernel Library (Intel oneAPI 2023.0).

6.3 Element availability of 3c-methods in TeraChem

The angular momentum for ERI evaluation in the TeraChem version used in this work is restricted to $l < 3$. The maximum angular momentum for ECPs is g . These restrictions limit the availability of the 3c-methods for certain elements in TeraChem. The implementation, realized in this work is numerically validated against the CPU programs ORCA and TURBOMOLE. These programs were used in the context of the original publications of these methods. The electronic energies and nuclear gradients of a selection of small molecules covering all implemented elements is computed and compared to the references. The numerical thresholds used for this procedure can be found in the appendix in sec. 8.2.1.^[99] Since DFT calculations depend largely on the chosen numerical grid, the following grids are used in each respective software suite: “dftgrid5” in TeraChem, “defgrid3” in ORCA and “grid m4” in TURBOMOLE.

Figs. 6.1-6.3 summarize which elements can be made accessible in TeraChem, given the angular momentum constraints, and, hence, match the original publication or implementation.^[6,11,24]

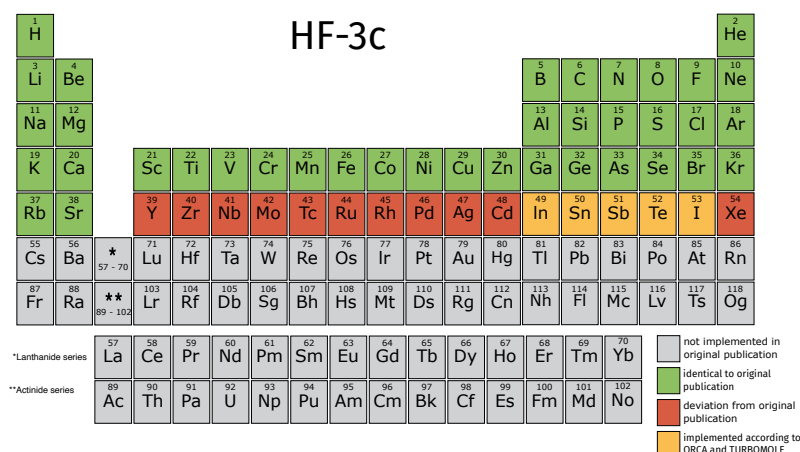


Fig. 6.1: Overview of elements for which HF-3c is implemented in TeraChem. The elements conforming to the original publication are colored in green. Elements that do not conform with the ORCA and TURBOMOLE implementation, due to unsupported basis functions with f angular momentum, are colored red. The elements colored in orange conform with the ORCA and TURBOMOLE implementation, but differ from information provided in the original publication. The following figure has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.

The definition of the basis set used with the HF-3c method differs for some elements between the HF-3c definition, and ORCA/TURBOMOLE implementation. Whereas ref. [24] defines that the elements from Y to Xe should use the def2-SV(P), the reference implementations in ORCA and TURBOMOLE use def2-SVP. Additionally, further inconsistencies for the ECPs have been recorded: Y to Cd use the def2-ECPs,^[189] whereas the def-ECPs (modified version of ref. [190]) are used for the main group elements from In to Xe. Due to the angular momentum restriction in TeraChem, the

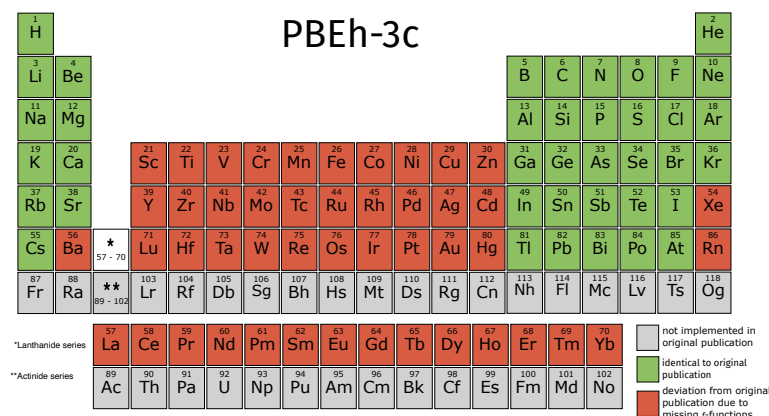


Fig. 6.2: Overview of elements for which PBEh-3c is implemented in TeraChem. The elements conforming to the original publication are colored in green. Elements that don't conform to the original publication are colored in red. **The following figure has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

transition metals from Y to Cd cannot be computed according to the ORCA and TURBOMOLE implementation. For the elements In to I, the ORCA and TURBOMOLE implementation is adopted, however, this divergence from ref. [24] is shown by using a different color in fig. 6.1.

The PBEh-3c publication defines the def2-mSVP basis set, which is a modified version of the def2-SVP (despite a typographical error in ref. [6] indicating the use of def2-SV(P)). The main modifications concern the elements of the first two periods. The def2-SVP basis functions includes f -functions for all transition metals and lanthanides, which are, therefore, not available in this TeraChem implementation.

At the time of implementing these methods into TeraChem, ω B97X-3c was not yet available in the TURBOMOLE software suite. Hence in this section and the following TeraChem is only compared to ORCA for this method. In contrast to PBEh-3c, the basis set of ω B97X-3c uses fewer f -functions for transition metals, making its implementation in TeraChem very appealing. However, ω B97X-3c relies heavily on ECPs to lower the computational cost without sacrificing accuracy. Some of these ECPs feature very high angular momenta, for example the sixth period (as well as K and Xe) use ECPs with h angular momentum. This surpasses TeraChem's ECP capability, and, hence, these elements are not available. Despite this, ω B97X-3c, being the most accurate 3c method, has the highest element coverage of the 3c-methods implemented in TeraChem, see fig. 6.3.

The empirical correction models needed for the composite methods were provided by the Grimme group as FORTRAN code. Compatibility with TeraChem's C++ architecture was achieved via the `iso_c_binding` module for the gCP program.^[191] The

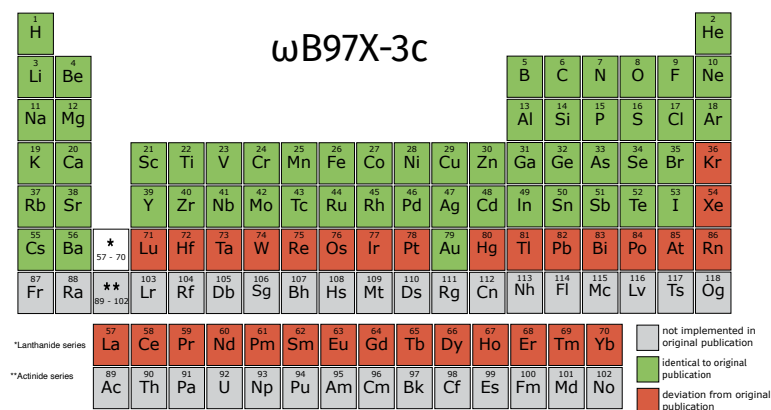


Fig. 6.3: Overview of the elements for which $\omega\text{B97X-3c}$ is implemented in TeraChem. The elements conforming to the original publication are colored in green. Elements that don't conform to the original publication are colored red. For the lanthanides, this is due to the lack of an ERI implementation for f -functions. The other elements are unavailable, because they use ECPs with unsupported angular momenta. **The following figure has been published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

DFT-D4 source code implements a C API that could be directly used to gain access to the dispersion model for $\omega\text{B97X-3c}$. The D3 model, on the other hand, has been implemented by Christoph Bannwarth in TeraChem previously and could be reused for HF-3c and PBEh-3c.

The mixed precision (MP) ERI handling scheme of TeraChem (see sec. 3.2) makes the use of CG-GPUs for *ab-initio* methods viable. However, the influence of the MP scheme on the numerical accuracy needs to be evaluated. The GMTKN55^[39] benchmark set can be used to compare relative energies for ORCA, TeraChem MP, and TeraChem double precision (DP). As discussed before, the $\omega\text{B97X-3c}$ method has the best coverage of the periodic table, hence it is a good candidate for this accuracy test. The GMTKN55 is used to evaluate the thermochemistry of the main block elements, using 55 benchmark sets grouped into 5 subsets. Each subset tries to capture a certain property, such as: reaction energies, barrier heights, basic properties, and non-covalent interactions. Due to the varying magnitudes of the respective relative energetics in these subsets, a weighted mean absolute deviation scheme (WTMAD-2) is established to be used for this set. However, the performance of the $\omega\text{B97X-3c}$ method is not of interest as it is covered in the original publication.^[11] Rather, the numerical deviations between TeraChem's precision models and ORCA are investigated. Tab. 6.2 shows the mean absolute deviation (MAD) relative to the reference values of the GMTKN55 per subset and for the weighted MAD (WTMAD-2) scheme, see eq. 8.4 in the appendix.^[39]

The largest deviation of 0.005 kcal/mol can be recorded between the TeraChem MP and ORCA values in the intramolecular subcategory, which is negligibly small and within the SCF convergence and DFT grid errors. To identify potential outliers, the Bessel-corrected standard deviation is also shown. The difference in standard deviation

tions is also rather small, with the largest divergence being 0.004 kcal/mol. The DP TeraChem calculations show deviations from the ORCA calculations in a similar order of magnitude than the MP version. This supports the argument that the errors mainly stem from SCF convergence and grid differences rather than insufficient accuracy. In summary, the MP scheme is not introducing a significant amount of inaccuracy compared to ORCA and can be safely used instead of the more computationally-costly DP mode.

Tab. 6.2: Mean absolute deviation (MAD) and standard deviation (STD) values in kcal/mol for the subcategories of the GMTKN55 benchmark set computed with the ω B97X-3c composite method. Due to the element restrictions of the TeraChem implementation, the benchmark sets HEAVY28, HEAVYSB11 and RG18 are not included in this analysis. The weighting scheme remains unchanged, the weighted MAD/STD for all examined sets is dubbed GMTKN52. DP refers to the double precision and MP to the mixed precision ERI handling schemes in TeraChem, and NCI to non-covalent interactions. **The following table has been published in Ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

Program (Numerical precision)	MAD of subcategories weighted according to WTMAD-2 (kcal/mol)					
	Basic prop.	Reaction energies	Barrier heights	intermol. NCIs	intramol. NCIs	GMTKN52
ORCA DP	4.749	7.933	5.154	4.887	5.698	5.480
TeraChem DP	4.745	7.935	5.154	4.883	5.701	5.479
TeraChem MP	4.746	7.934	5.153	4.883	5.703	5.479
Program (Numerical precision)	WTSTD-2: STD of subcategories weighted according to WTMAD-2 (kcal/mol)					
	Basic prop.	Reaction energies	Barrier heights	intermol. NCIs	intramol. NCIs	GMTKN52
ORCA DP	5.180	7.471	5.963	4.766	5.515	5.608
TeraChem DP	5.184	7.475	5.963	4.762	5.518	5.609
TeraChem MP	5.184	7.475	5.963	4.762	5.519	5.610

Although, the performance of ω B97X-3c is not the focus of this section, it should be noted that it performs very similar to more costly methods such as ω B97X-D4/def2-QZVP.^[192] The latter method has a WTMAD-2 for the whole set of 4.34 kcal/mol, which is about 1 kcal/mol less than the TeraChem ω B97X-3c MP implementation. For some subsets, both methods are even closer such as C60ISO, for which ω B97X-D4/def2-QZVP scores 14.17 kcal/mol and ω B97X-3c 14.64 kcal/mol. This is especially remarkable considering the former method uses a quadruple zeta basis, which is computationally far more demanding. Hence, the implementation of ω B97X-3c adds a composite method to TeraChem with fairly accurate thermochemistry that runs effectively on CG-GPUs. The speed-up associated with this implementation compared to state-of-the-art CPU implementations will be evaluated in the next section.

6.4 Comparing SCF walltimes for implemented 3c-methods

In this section, the average walltimes per SCF cycle are compared between the CPU-only codes and TeraChem. This quantity has the advantage that it is independent of convergence thresholds or other settings that influence the number of SCF iterations. Since this work focusses on inexpensive workstations as target hardware, we choose to run these timings with 8 CPU threads. As discussed before, most CPU implementations of ERI codes use elaborate and/or memory-intensive approximation schemes to lower the formal scaling of the ERI computation. The RIJ^[26,193–196] approximation is available in both CPU programs ORCA and TURBOMOLE. The COSX approximation^[27–29] is only implemented in ORCA, the seminumerical exchange approximation in TURBOMOLE^[183] is not used in this work. The *cefine*^[197] tool distributed by the Grimme group is used to prepare TURBOMOLE calculations. When using PBEh-3c or HF-3c, *cefine* removes the basis functions with *g*-angular momentum from the auxiliary basis set used in RIJ. TeraChem features none of the formal scaling reduction approximations. For a more thorough comparison, the timings are given with and without these schemes for the CPU programs. The average SCF walltime shown in this work is computed as the overall walltime of the program divided by the number of SCF iterations. This ensures that all setup times and other parts of the program are captured and a fair comparison among different programs is possible.

The scaling of the walltime with system size is evaluated based on a set of 164 carbon fullerenes taken from ref. [136]. The geometries are used as provided by the reference and have not undergone geometry optimization. Since several structures with the same number of basis functions are present in the set, the walltime is averaged over all structures.

The timings for the HF-3c composite method are shown in fig. 6.4.

ORCA generates quite large temporary files that contain intermediates to be reused in the RIJCOSX-based ERI evaluations. For the largest fullerenes in this set, the intermediate files are too large for the 500 GB disk space available on the used hardware. Hence, these datapoints are missing in the figures. For small molecules, the walltime for ORCA shows no clear scaling with system size. This behavior could be linked to program-specific overheads, but no further research was undertaken as the scaling becomes regular again for larger molecules. The benefit of using the RIJ and COSX approximations is very small for the minimal basis HF-3c method. For TURBOMOLE a small speed-up can be observed with using the RIJ approximation, however this speed-up is more prominent for the other methods. This observation can be explained by the minimal basis set used in HF-3c, which lowers the advantage created by these approximations. The DP scheme in TeraChem leads to an acceleration compared to the CPU programs for molecules with more than 200 basis functions. The use of the MP scheme leads to a larger speed-up even for the smallest molecules. A speed-up of up to one order of magnitude can be recorded for the HF-3c in MP runmode in TeraChem compared to ORCA or TURBOMOLE.

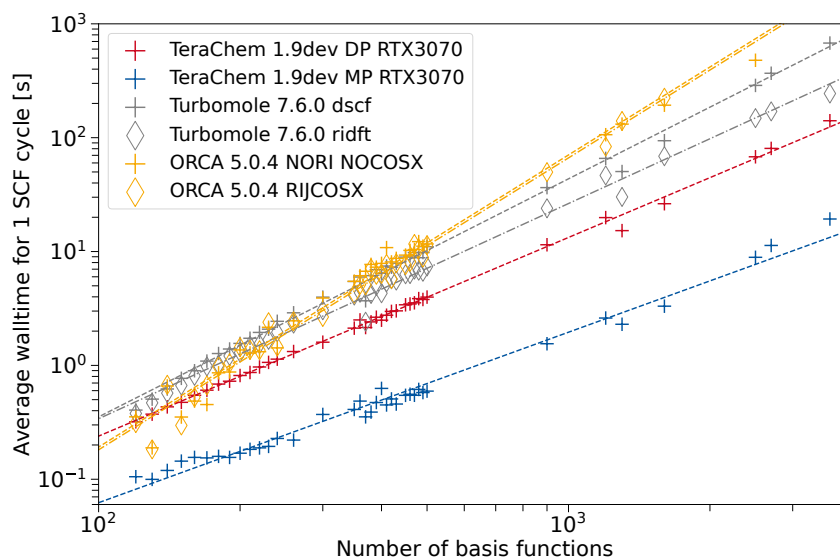


Fig. 6.4: Average walltime per SCF cycle for the HF-3c composite method as a function of the number of basis functions. For TeraChem, the mixed precision (MP) and double precision (DP) computation schemes are considered. For TURBOMOLE (ridft, dscf) and ORCA (NORI, RIJ), single-point calculations with and without the resolution of the identity approximation (and seminumerical COSX scheme in ORCA) are shown. A computer with type 2 setup is used, see tab. 8.1. **This figure has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

Comparing DFT implementations is inherently more difficult as the size of the numerical grid has a strong influence on the computational cost. However, the computation of the non-local Fock exchange is generally the dominant factor for hybrid DFT functionals, like PBEh-3c and ω B97X-3c. Hence, the dependence on the DFT grid should be less prominent. For PBEh-3c, the *cefine* tool adds the following parameter to the TURBOMOLE control file “\$extol 2.5”. This parameter is used in various screening thresholds and lowers the number of integrals to be computed in the exact Fock exchange computation routine in TURBOMOLE’s *ridft* program.^[198] The influence of this parameter on the walltime is also studied in fig. 6.5. The datapoints for fullerenes larger than C260 (3900 basis functions) in ORCA are missing since the program ran out of disk space to write temporary files.

For the used hardware and parameter setup, ORCA is always a bit slower than TURBOMOLE with and without integral approximations. However, it should be noted again that the *g*-functions of the auxiliary basis have been removed for TURBOMOLE but not for ORCA. A drastic reduction in computation time has been recorded when using “\$extol 2.5” with TURBOMOLE for small systems. With this setting, TURBOMOLE is faster than the TeraChem DP for molecules with less than 3.000 basis functions. Nevertheless, the GPU-accelerated MP scheme clearly yields the fastest computation method evaluated in this work, being an order of magnitude faster com-

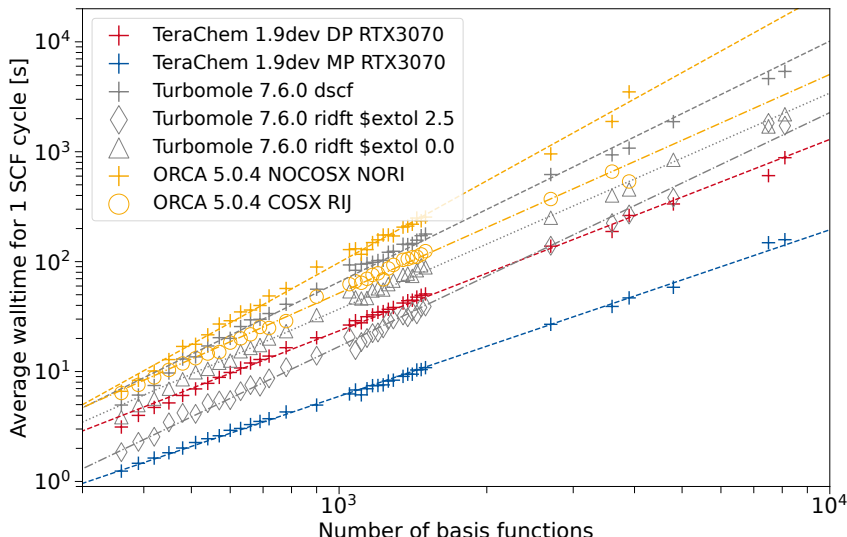


Fig. 6.5: Average walltime per SCF cycle for the PBEh-3c composite method as a function of the number of basis functions. For TeraChem, the mixed precision (MP) and double precision (DP) computation schemes are considered. For TURBOMOLE (ridft, dscf) and ORCA (NORI, RIJ), single-point calculations with and without the resolution of the identity approximation (and seminumerical COSX scheme in ORCA) are shown. “\$extol” is a parameter that is added to the exponential value of screening parameters in the Fock build routine of ridft.^[198] A computer with type 2 setup is used, see tab. 8.1. **This figure has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

pared to the second fastest scheme (TeraChem DP and Turbomole, respectively).

As stated before, at the time of publishing the present research, a TURBOMOLE implementation of ω B97X-3c has not yet been published. Hence, the comparison is restricted to TeraChem and ORCA, see fig. 6.6. Besides the disk space restriction, we also encountered difficulties computing fullerenes larger than 1300 basis functions with ORCA. The calculation failed repeatedly while setting up the model potential of the guess calculation.

The ω B97X-3c basis vDZP features a large number of primitive Gaussians compared to basis sets like def2-mSVP or MINIX. TeraChem uses the MMD ERI evaluation scheme, which performs the contraction step only at the end after building the necessary angular momentum. Hence, this scheme is not the most efficient for basis sets such as vDZP. The COSX and RIJ approximations appear to be more efficient for the ω B97X-3c composite method compared to the other 3c-methods investigated in this work. This can be linked to the fact that those schemes compute and store intermediates that can be reused in future SCF cycles. This is especially compute time-saving for highly contracted basis sets such as vDZP combined with late-contraction ERI schemes such as MMD used in ORCA’s SHARK.^[123]

ORCA with RIJ and COSX approximation presents about the same average walltime as the DP scheme in TeraChem. However, the TeraChem MP scheme still outperforms all other ERI implementations and leads to nearly an order of magnitude of speed-

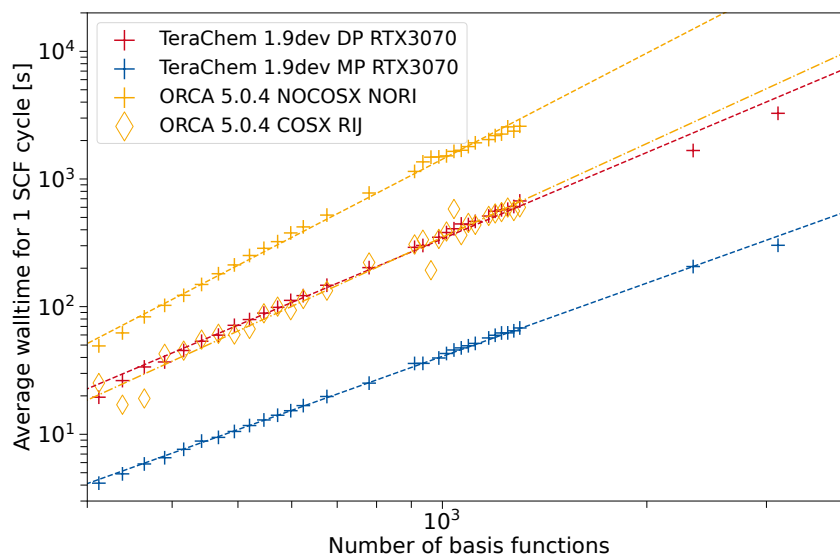


Fig. 6.6: Average walltime per SCF cycle for the composite method ω B97X-3c as a function of the number of basis functions. For TeraChem, the mixed precision (MP) and double precision (DP) computation schemes are considered. For ORCA single point calculations with and without the resolution of the identity and chain of spheres approximation are shown (NOCOSX NORI, RIJCOSX, respectively). A computer with type 2 setup is used, see tab. 8.1. **This figure has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

up compared to ORCA with formal low-scaling approximations. The results shown in this section have been recorded on an “off-the-shelf” workstation with a mid-tier GPU. This section thereby shows that an outstanding degree of acceleration can be achieved without the need for very expensive specialized hardware. To underline this point, varying GPU models are compared in the upcoming section to elucidate the price to performance ratio.

6.5 Price-to-performance comparison of GPU models

In the previous section, the discussion focussed on the performance of the TeraChem GPU implementation versus existing CPU implementations with and without favorably scaling ERI approximations. In most publications, the ratio of price to performance is not taken into account when publishing GPU implementations. However, especially if this software is tailored to GP-GPUs, the price of the hardware is limiting the access to this software for many users. Therefore, we investigate the performance of various GPUs from NVIDIA, to check if the higher price tag for QUADRO GPUs leads to better performance for our application. The prices shown in this section are manufacturer’s suggested retail price (MSRP) and not current market prices.

Whereas, GeForce cards have a FP64/FP32 theoretical performance ratio of 1/64, Quadro cards have a 1/32 ratio. However, in recent generations the Quadro line has been discontinued and instead variations of the CG-GPU chips with more memory are sold as workstation GPUs. In tab. 6.3, the theoretical performance of the GPUs used in this comparison is presented.

Tab. 6.3: Comparison of the theoretical performance of GPUs for FP32 and FP64 operations. Values are taken from ref.[30]. The following table has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.

GPU	Theoretical performance	
	FP64 [GFLOPS]	FP32 [TFLOPS]
GeForce RTX 3070	317.4	20.31
GeForce RTX 3080Ti	532.8	34.10
Quadro RTX 5000	348.5	11.15

In fig. 6.7, these selected GPUs are compared against each other using PBEh-3c calculations in TeraChem using the DP and MP runtime. The same fullerene model systems as in the previous section are used, and the average walltime per SCF section is compared. The computers with RTX 3080Ti (system 4) and Quadro RTX 5000 (system 5) GPUs share the same CPU model, as see in tab. 8.1. The systems with RTX 3070 GPUs are equipped with other CPUs. However, the influence of the GPUs on the overall walltime is assumed to be more important and, hence, the non-matching CPUs are neglected. The plots for the other implemented 3c-methods can be found in the SI of the original publication, but the trend for all investigated methods is similar.

As expected from the theoretical performance, the RTX 3070 and RTX 5000 Quadro perform very similar for the DP scheme. The higher FP32 performance of the 3070 leads to lower walltimes per SCF for the MP scheme especially for larger systems. Considering the substantially lower price of the RTX 3070 compared to the Quadro card, there is no argument in favor of the latter card apart from its higher memory capacity. The RTX 3080Ti shows the lowest walltimes for all considered GPUs for both precision runtimes and should be chosen over the RTX 5000 Quadro. The choice between the 3080Ti and 3070 is mainly driven by the targeted application, since for the price of one 3080Ti two 3070 GPUs could be purchased. This comparison shows that CG-GPUs in combination with mixed precision implementations are a viable strategy to speed-up quantum chemical calculations. Given that such “consumer-grade” cards are comparable in price to modern CPU chips, while providing much better performance in electronic structure calculations (by a factor of ~ 10 with the MP scheme, see section 6.4), one can expect roughly a five-fold speed-up in the calculations per invested dollar. With the implementation of the 3c-methods in TeraChem, a key component of the multilevel workflow, the reoptimization of the conformer ensemble, can be accelerated effectively. In particular, this chapter showed that the comparatively cheap CG-GPUs perform very well at this task if combined with smart algorithmic implementations such as TeraChem’s mixed precision scheme.

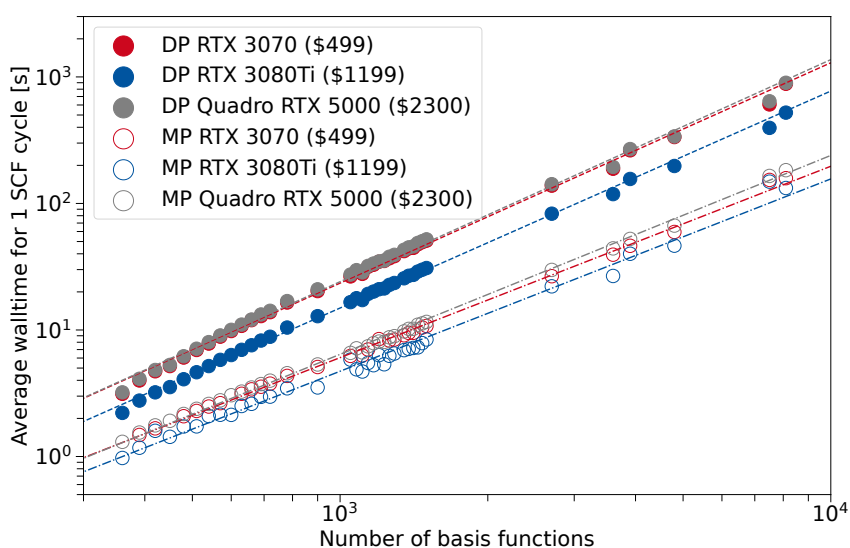


Fig. 6.7: Average wall time for 1 SCF cycle for the composite method PBEh-3c as a function of the number of basis set functions. For TeraChem, the mixed precision (MP) and double precision (DP) computation schemes are compared for different GPU models. The indicated prices are MSRP according to ref. [30]. **This figure has been previously published in ref. [99] under the CC BY 3.0 license allowing its reproduction free of charge.**

7 Conclusion & outlook

In this chapter, a summary is drawn of the projects that have been presented in this work and how their impact affects future developments.

LAHVA is provided to the community as open source code to simplify the development of GPU-accelerated code without the need for expert knowledge. In this work, the software design philosophy behind this project has been introduced. Besides delivering this abstract theoretical explanation, the implementation of a scientific model based on LAHVA is also shown. The straightforward use and performance of the library have been demonstrated by computing the Mulliken-approximated exchange contribution to the Fock matrix. The mixed-precision incremental Fock build proves LAHVA's outstanding capability to unify several numerical accuracies in a single model. This unlocks the efficient use of CG-GPUs for scientific calculations with high accuracy demands. To go even further, a FP16 splitting scheme has been developed, that allows to reduce the number of FP64 operations even further. The impact of this splitting scheme is shown impressively by accelerating the full build of the exchange contributions to the Fock matrix by factor 5.5 compared to FP64 calculations. This acceleration on CG-GPUs is so significant and efficient that it allows a faster computation than the FP64/FP32 incremental build scheme, while presenting a similar accuracy. This speed-up from the FP16 splitting scheme is available to the implementing software developer by simply switching the object to a `MixedPrecisionMatrix`.

With these developments in LAHVA the foundation has been laid to quickly transfer further terms of an electronic structure theory method to the GPU. Although, the focus of this work has been on using CG-GPUs, the performance of on GP-GPUs is also a matter of interest. Due to the recent increase in the computation of artificial intelligence models, the hardware developers have boosted the performance of low-precision floating point numbers such as FP16 enormously. Modern GP-GPUs by NVIDIA like the H100 offer up to 989 TFLOPS of theoretical performance for FP16 tensor core operations.^[199] With the addition of the FP16 scheme, LAHVA's scope has also been extended to GP-GPUs.

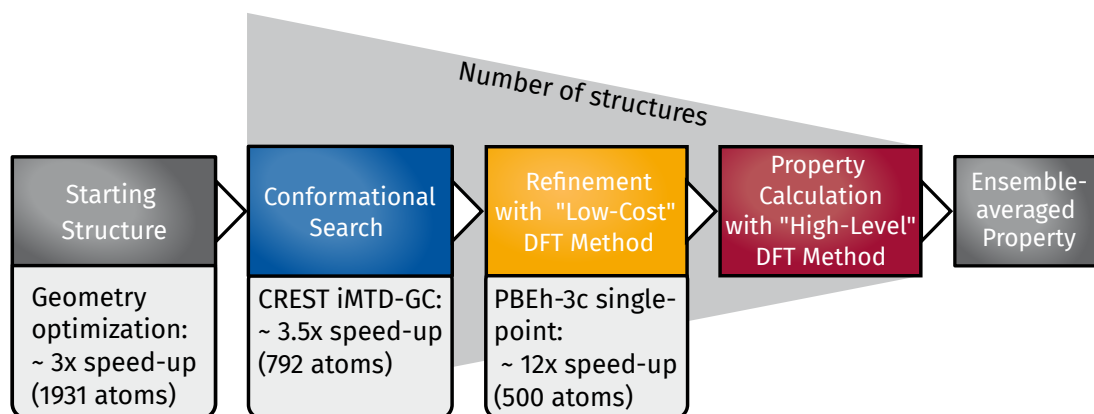


Fig. 7.1: Overview of the speed-up realized by the projects presented in this work. The acceleration for the first two steps is relative to diagonalization with CPU, while using an RTX 4090 (system 1). For the PBEh-3c calculation, the runtime of TeraChem MP is compared to the Turbomole calculation with the RIJ approximation and “\$extol 2.5”. The speed-up of ω B97X-3c is reported comparing TeraChem MP with ORCA (with RIJ and COSX approximations). The PBEh-3c and ω B97X-3c timings are recorded from a system 4 machine, see tab. 8.1.

Moving to the individual stages of the multilevel workflow introduced in the motivation chapter (see ch. 1), the acceleration realized in every step can be seen in fig. 7.1. If the workflow starts with a manually built geometry or a file from the protein database for example. In a first step the geometry would be relaxed on a GFN2-xTB PES. Especially for biology-related applications, these structures could have significant number of atoms. In sec. 5.5.2, the structure of the photoactive yellow protein (1931 atoms) is optimized. The use of the SP2 DMP algorithm with the previously mentioned FP16 split leads to a decrease in computation time of about factor 3. This speed-up is realized on consumer-grade hardware. Hence, proteins structures can now routinely be optimized on local workstation machine in only 2 hours for nearly 2000 atoms. This also unlocks calculations of even larger systems in a sensible timeframe. It is worth mentioning that these algorithms are provided open-source to the community and we are currently working on distributing pre-compiled versions.

However, the runtime analysis also showed that the density build is with our SP2 MP implementation no longer the computational bottleneck. Therefore, other components of the Hamiltonian need to be GPU-accelerated as well, as already done in the “SQM-Box” effort by Bannwarth and Martinez.^[133] This implementation resides in a closed source project and is part of a monolithic software architecture. In order to have a larger impact and be more readily available, an open-source implementation would be the logical next step. With the exchange model shown in this work, we have proven that this is can be easily achieved with LAHVA. The addition of new semiempirical methods such as g-xtb^[59] should also be straightforward.

Based on the relaxed geometry, the thermally accessible conformers should be identified. For this step, the SP2 MP algorithm can also be used. The computational bottleneck for this step at least when using the CREST’s is the geometry optimization

of the conformer candidates. Due to this, the acceleration should be equal to that for a geometry optimization for the corresponding molecule size. However, the overall utilization can be improved by scheduling the GPU calculations asynchronously. Additionally, the GPU calculations can also be distributed among several GPUs. By doing so an acceleration for the iMTD-GC run of factor 4.0 can be reached for a polypeptide with 792 atoms with the newly developed SP2 MP FP16 algorithm. This acceleration of computing time can be easily adopted by other workflows that use `tblite` for energy and gradient calculations. Computing the Hamiltonian on the GPU would also lead to an even larger speed-up. However, moving more calculations to the GPU would mean that the GPU is already saturated for fewer SCFs than currently. At that point it might become necessary to support distributed computing schemes to increase the number of concurrently sustained SCFs.

The next steps of the multilevel workflow are currently mostly performed using DFT calculations. As DFT calculations, especially with (semi)empirical correction schemes, present a good accuracy to computational cost ratio in the realm of “ab-initio” methods. In this work, the implementation of three “3c” methods in the GPU-accelerated TeraChem software is reported: HF-3c, PBEh-3c and ω B97X-3c. GPU ERI implementations decrease in efficiency with increasing angular momentum. These methods are very well suited for use in GPU-based SCF codes as they are based on small, double zeta basis sets. The “mixed-precision” runmode implemented in TeraChem allows the fast computation of integrals on CG-GPUs without a significant loss in accuracy. A rigorous timings comparison between TeraChem in mixed and double precision mode and state-of-the-art CPU implementations on various hardware is presented. Based on these results speed-ups of more than an order of magnitude can be reported on mid to high-tier CG-GPUs. A GPU comparison also shows that the increased cost of NVIDIA’s workstation GPUs does not lead to increased performance, for the models investigated in this work.

For applications, these developments allow to decrease the runtime of the refinement step by more than an order of magnitude. This not only means that less computational resources are needed but also that results are computed faster allowing researchers to get faster feedback from calculations. Additionally, these developments also increase the size of systems that can be routinely treated by DFT calculations. The computation of large systems such as C_{500} is easily feasible, whereas CPU programs are already facing their limits. With the ω B97X-3c method, it might also be possible to substitute large basis set calculations with this method based on a split-valence basis. This allows users to compute adequately-accurate Boltzmann weights for large systems, where DFT calculations might not longer be feasible.

Further improvements of GPU ERI implementations could be based on implementing low-scaling approximations such as RIJ or exchange equivalents. The idea of the mixed-precision runtime to compute ERI elements in various numerical precisions could be extended to use FP16 splitting methods as well. The number of splits could be chosen based on the expected numerical value determined by the Schwarz bound.

In summary, the projects presented in this work shift the limit of the molecule size that can be simulated by multilevel workflows. This acceleration is achieved by combining tailored algorithmic developments with affordable consumer grade hardware. Thereby, steps have been undertaken to reach overarching goal of “democratizing” GPU-usage in quantum-chemical calculations.

Bibliography

- [1] T. Kratz, P. Steinbach, S. Breitenlechner, G. Storch, C. Bannwarth, T. Bach, *J. Am. Chem. Soc.* **2022**, *144*, 10133–10138.
- [2] S. Grimme, *J. Chem. Phys.* **2013**, *138*, 244104.
- [3] C. Bannwarth, S. Grimme, *Comput. Theo. Chem.* **2014**, *1040–1041*, 45–53.
- [4] A. D. Becke, *J. Chem. Phys.* **1993**, *98*, 1372–1377.
- [5] F. Weigend, R. Ahlrichs, *Phys. Chem. Chem. Phys.* **2005**, *7*, 3297–3305.
- [6] S. Grimme, J. G. Brandenburg, C. Bannwarth, A. Hansen, *J. Chem. Phys.* **2015**, *143*, 054107.
- [7] P. Pracht, F. Bohle, S. Grimme, *Phys. Chem. Chem. Phys.* **2020**, *22*, 7169–7192.
- [8] P. Pracht, S. Grimme, C. Bannwarth, F. Bohle, S. Ehlert, G. Feldmann, J. Gorges, M. Müller, T. Neudecker, C. Plett, S. Spicher, P. Steinbach, P. A. Wesolowski, F. Zeller, *J. Chem. Phys.* **2024**, *160*, 114110.
- [9] C. Bannwarth, S. Ehlert, S. Grimme, *J. Chem. Theory Comput.* **2019**, *15*, 1652–1671.
- [10] S. Grimme, A. Hansen, S. Ehlert, J.-M. Mewes, *J. Chem. Phys.* **2021**, *154*, 064103.
- [11] M. Müller, A. Hansen, S. Grimme, *J. Chem. Phys.* **2023**, *158*, 014103.
- [12] D. Kolarski, P. Steinbach, C. Bannwarth, K. Klaue, S. Hecht, *Angew. Chem. - Int. Ed.* **2024**, *63*, e202318015.
- [13] Z. Liu, M. Li, Q. Xie, Y. Liu, J. Huang, Q. Zeng, X. Li, K. Rao, J. Ning, M. Zhao, B. Li, F. Li, H. Liu, S. Zhou, B. Shu, B. Yang, J. Zheng, Y. Liao, *Nat. Commun.* **2025**, *16*, 6499.
- [14] M. Colaço, J. Ewert, J.-S. von Glasenapp, U. Pischel, R. Herges, N. Basílio, *J. Am. Chem. Soc.* **2025**, *147*, 734–745.

- [15] A. J. Lee, E. E. DeLuca, E. B. Kelly, J. M. Taylor, L. R. Weddle, H. Chen, C. Park, C. K. Loeb, T. Chan, C. P. Kubiak, *J. Am. Chem. Soc.* **2025**, *147*, 7411–7422.
- [16] W. Thiel, *WIREs Comput. Mol. Sci.* **2014**, *4*, 145–157.
- [17] A. S. Christensen, T. Kubař, Q. Cui, M. Elstner, *Chem. Rev.* **2016**, *116*, 5301–5337.
- [18] D. Porezag, Th. Frauenheim, Th. Köhler, G. Seifert, R. Kaschner, *Phys. Rev. B* **1995**, *51*, 12947–12957.
- [19] M. Gaus, Q. Cui, M. Elstner, *J. Chem. Theory Comput.* **2011**, *7*, 931–948.
- [20] S. Grimme, C. Bannwarth, P. Shushkov, *J. Chem. Theory Comput.* **2017**, *13*, 1989–2009.
- [21] C. Bannwarth, E. Caldeweyher, S. Ehlert, A. Hansen, P. Pracht, J. Seibert, S. Spicher, S. Grimme, *WIREs Comput. Mol. Sci.* **2021**, *11*, e1493.
- [22] S. Grimme, *J. Chem. Theory Comput.* **2019**, *15*, 2847–2862.
- [23] C. Zurek, R. A. Mallaev, A. C. Paul, N. van Staaldin, P. Pracht, R. Ellerbrock, C. Bannwarth, *J. Chem. Theory Comput.* **2025**, *21*, 1459–1475.
- [24] R. Sure, S. Grimme, *J. Comput. Chem.* **2013**, *34*, 1672–1685.
- [25] J. G. Brandenburg, C. Bannwarth, A. Hansen, S. Grimme, *J. Chem. Phys.* **2018**, *148*, 064104.
- [26] O. Vahtras, J. Almlöf, M. Feyereisen, *Chem. Phys. Lett.* **1993**, *213*, 514–518.
- [27] F. Neese, F. Wennmohs, A. Hansen, U. Becker, *J. Chem. Phys.* **2009**, *356*, 98–109.
- [28] B. Helmich-Paris, B. de Souza, F. Neese, R. Izsák, *J. Chem. Phys.* **2021**, *155*, 104109.
- [29] R. Izsák, F. Neese, W. Klopper, *J. Chem. Phys.* **2013**, *139*, 094111.
- [30] <https://www.techpowerup.com/>, (last accessed: 20.10.2025).
- [31] American National Standards Institute, *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*, **1985**.
- [32] I. S. Association, et al., *IEEE Standard* **2008**, 754–2008.
- [33] A. Szabo, N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, Courier Corporation, **2012**.

- [34] F. Jensen in *Introduction to Computational Chemistry 2nd ed.*, John Wiley & Sons, **2007**, chapter 15.2, pp. 80–131.
- [35] F. Jensen in *Introduction to Computational Chemistry 2nd ed.*, John Wiley & Sons, **2007**, chapter 15.2, pp. 1–21.
- [36] P. Hohenberg, W. Kohn, *Phys. Rev.* **1964**, *136*, B864–B871.
- [37] F. Jensen in *Introduction to Computational Chemistry 2nd ed.*, John Wiley & Sons, **2007**, chapter 15.2, pp. 492–494.
- [38] W. Kohn, L. J. Sham, *Phys. Rev.* **1965**, *140*, A1133–A1138.
- [39] L. Goerigk, A. Hansen, C. Bauer, S. Ehrlich, A. Najibi, S. Grimme, *Phys. Chem. Chem. Phys.* **2017**, *19*, 32184–32215.
- [40] H. Iikura, T. Tsuneda, T. Yanai, K. Hirao, *J. Chem. Phys.* **2001**, *115*, 3540–3544.
- [41] J.-D. Chai, M. Head-Gordon, *Phys. Chem. Chem. Phys.* **2008**, *10*, 6615–6620.
- [42] J.-D. Chai, M. Head-Gordon, *J. Chem. Phys.* **2008**, *128*, 084106.
- [43] Y.-S. Lin, G.-D. Li, S.-P. Mao, J.-D. Chai, *J. Chem. Theory Comput.* **2013**, *9*, 263–272.
- [44] E. Hückel, *Z. Phys.* **1931**, *70*, 204–286.
- [45] E. Hückel, *Z. Phys.* **1933**, *83*, 632–668.
- [46] R. Hoffmann, *J. Chem. Phys.* **1963**, *39*, 1397–1412.
- [47] W. Weber, W. Thiel, *Theo. Chem. Acc.* **2000**, *103*, 495–506.
- [48] M. J. S. Dewar, W. Thiel, *Theoret. Chim. Acta* **1977**, *46*, 89–104.
- [49] M. J. S. Dewar, W. Thiel, *J. Am. Chem. Soc.* **1977**, *99*, 4899–4907.
- [50] M. J. S. Dewar, E. G. Zoebisch, E. F. Healy, J. J. P. Stewart, *J. Am. Chem. Soc.* **1985**, *107*, 3902–3909.
- [51] J. J. P. Stewart, *J. Mol. Model* **2007**, *13*, 1173–1213.
- [52] G. Seifert, D. Porezag, Th. Frauenheim, *Int. J. Quantum Chem.* **1996**, *58*, 185–192.
- [53] M. Elstner, D. Porezag, G. Jungnickel, J. Elsner, M. Haugk, Th. Frauenheim, S. Suhai, G. Seifert, *Phys. Rev. B* **1998**, *58*, 7260–7268.

- [54] S. Grimme, J. Antony, S. Ehrlich, H. Krieg, *J. Chem. Phys.* **2010**, *132*, 154104.
- [55] S. Grimme, S. Ehrlich, L. Goerigk, *J. Comput. Chem.* **2011**, *32*, 1456–1465.
- [56] E. Caldeweyher, C. Bannwarth, S. Grimme, *J. Chem. Phys.* **2017**, *147*, 034112.
- [57] E. Caldeweyher, S. Ehlert, A. Hansen, H. Neugebauer, S. Spicher, C. Bannwarth, S. Grimme, *J. Chem. Phys.* **2019**, *150*, 154122.
- [58] H. Kruse, S. Grimme, *J. Chem. Phys.* **2012**, *136*.
- [59] T. Froitzheim, M. Müller, A. Hansen, S. Grimme, *G-xTB: A General-Purpose Extended Tight-Binding Electronic Structure Method For the Elements H to Lr (Z=1–103)*, **2025**.
- [60] R. S. Mulliken, *J. Chem. Phys.* **1955**, *23*, 1833–1840.
- [61] A. M. Köster, M. Leboeuf, D. R. Salahub in *Molecular Electrostatic Potentials: Concepts and Applications*, Vol. 3, J. S. Murray, K. Sen (Eds.), Elsevier, **1996**, chapter In *Molecular Electrostatic Potentials*, pp. 105–142.
- [62] P. Pulay, *Chem. Phys. Lett.* **1980**, *73*, 393–398.
- [63] C. G. Broyden, *Math. Comput.* **1965**, *19*, 577–593.
- [64] N. D. Mermin, *Phys. Rev.* **1965**, *137*, A1441–A1443.
- [65] <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, (last accessed: 20.10.2025).
- [66] G. M. J. Barca, M. Alkan, J. L. Galvez-Vallejo, D. L. Poole, A. P. Rendell, M. S. Gordon, *J. Chem. Theory Comput.* **2021**, *17*, 7486–7503.
- [67] J. Almlöf, K. Faegri Jr., K. Korsell, *J. Comput. Chem.* **1982**, *3*, 385–399.
- [68] V. Dyczmons, *Theoret. Chim. Acta* **1973**, *28*, 307–310.
- [69] I. S. Ufimtsev, T. J. Martínez, *Comput. Sci. Eng.* **2008**, *10*, 26–34.
- [70] M. Häser, R. Ahlrichs, *J. Comput. Chem.* **1989**, *10*, 104–111.
- [71] I. S. Ufimtsev, T. J. Martinez, *J. Chem. Theory Comput.* **2009**, *5*, 1004–1015.
- [72] N. Luehr, I. S. Ufimtsev, T. J. Martínez, *J. Chem. Theory. Comput.* **2011**, *7*, 949–954.

- [73] J. Kussmann, C. Ochsenfeld, *J. Chem. Theory Comput.* **2017**, *13*, 3153–3159.
- [74] P. E. Maslen, C. Ochsenfeld, C. A. White, M. S. Lee, M. Head-Gordon, *J. Phys. Chem. A* **1998**, *102*, 2215–2222.
- [75] S. Seritan, C. Bannwarth, B. S. Fales, E. G. Hohenstein, C. M. Isborn, S. I. L. Kokkila-Schumacher, X. Li, F. Liu, N. Luehr, J. W. Snyder Jr., C. Song, A. V. Titov, I. S. Ufimtsev, L.-P. Wang, T. J. Martínez, *WIREs Comput. Mol. Sci.* **2021**, *11*, e1494.
- [76] S. Seritan, C. Bannwarth, B. S. Fales, E. G. Hohenstein, S. I. L. Kokkila-Schumacher, N. Luehr, Jr. Snyder, James W., C. Song, A. V. Titov, I. S. Ufimtsev, T. J. Martínez, *J. Chem. Phys.* **2020**, 152.
- [77] S. Obara, A. Saika, *J. Chem. Phys.* **1986**, *84*, 3963–3974.
- [78] S. Obara, A. Saika, *J. Chem. Phys.* **1988**, *89*, 1540–1559.
- [79] L. E. McMurchie, E. R. Davidson, *J. Comput. Phys.* **1978**, *26*, 218–231.
- [80] M. Dupuis, J. Rys, H. F. King, *J. Chem. Phys.* **1976**, *65*, 111–116.
- [81] R. Lindh, U. Ryu, B. Liu, *J. Chem. Phys.* **1991**, *95*, 5889–5897.
- [82] M. Head-Gordon, J. A. Pople, *J. Chem. Phys.* **1988**, *89*, 5777–5786.
- [83] R. Li, Q. Sun, X. Zhang, G. K.-L. Chan, *Introducing GPU-acceleration into the Python-based Simulations of Chemistry Framework*, **2024**.
- [84] A. Asadchev, E. F. Valeev, *J. Chem. Phys.* **2024**, *160*, 244109.
- [85] E. Palethorpe, R. Stocks, G. M. J. Barca, *J. Chem. Theory Comput.* **2024**, *20*, 10424–10442.
- [86] I. S. Ufimtsev, T. J. Martínez, *J. Chem. Theory Comput.* **2008**, *4*, 222–231.
- [87] K. Yasuda, *J. Chem. Theory Comput.* **2008**, *4*, 1230–1236.
- [88] A. Asadchev, M. S. Gordon, *J. Chem. Theory Comput.* **2012**, *8*, 4166–4176.
- [89] Y. Miao, K. M. J. Merz, *J. Chem. Theory Comput.* **2013**, *9*, 965–976.
- [90] J. Kalinowski, F. Wennmohs, F. Neese, *J. Chem. Theory Comput.* **2017**, *13*, 3160–3170.

- [91] A. Asadchev, E. F. Valeev, *J. Phys. Chem. A* **2023**, 127, 10889–10895.
- [92] Y. Tian, B. Suo, Y. Ma, Z. Jin, *J. Chem. Phys.* **2021**, 155, 034112.
- [93] G. M. J. Barca, C. Bertoni, L. Carrington, D. Datta, N. De Silva, J. E. Deustua, D. G. Fedorov, J. R. Gour, A. O. Gunina, E. Guidez, T. Harville, S. Irle, J. Ivanic, K. Kowalski, S. S. Leang, H. Li, W. Li, J. J. Lutz, I. Magoulas, J. Mato, V. Mironov, H. Nakata, B. Q. Pham, P. Piecuch, D. Poole, S. R. Pruitt, A. P. Rendell, L. B. Roskop, K. Ruedenberg, T. Sattasathuchana, M. W. Schmidt, J. Shen, L. Slipchenko, M. Sosonkina, V. Sundriyal, A. Tiwari, J. L. Galvez Vallejo, B. Westheimer, M. Włoch, P. Xu, F. Zahariev, M. S. Gordon, *J. Chem. Phys.* **2020**, 152, 154102.
- [94] M. Manathunga, C. Jin, V. W. D. Cruzeiro, Y. Miao, D. Mu, K. Arumugam, K. Keipert, H. M. Aktulga, K. M. J. Merz, A. W. Götz, *J. Chem. Theory Comput.* **2021**, 17, 3955–3966.
- [95] M. Manathunga, H. M. Aktulga, A. W. Götz, K. M. J. Merz, *J. Chem. Inf. Model.* **2023**, 63, 711–717.
- [96] R. Stocks, E. Palethorpe, G. M. J. Barca, *J. Chem. Theory Comput.* **2024**, 20, 7503–7515.
- [97] D. B. Williams-Young, A. Asadchev, D. T. Popovici, D. Clark, J. Waldrop, T. L. Windus, E. F. Valeev, W. A. de Jong, *J. Chem. Phys.* **2023**, 158, 234104.
- [98] P. Pokhilko, E. Epifanovsky, A. I. Krylov, *J. Chem. Theory Comput.* **2018**, 14, 4088–4096.
- [99] P. Steinbach, C. Bannwarth, *Phys. Chem. Chem. Phys.* **2024**, 26, 16567–16578.
- [100] <https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf>, (last accessed: 20.10.2025).
- [101] W. Dawson, K. Ozaki, J. Domke, T. Nakajima, *J. Chem. Theory Comput.* **2024**.
- [102] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, J. S. Vetter in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531.
- [103] D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura in *High Performance Computing*, Springer International Publishing, Cham, pp. 230–248.
- [104] D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura in *Proceedings of the 50th Inter-*

- national Conference on Parallel Processing*, Association for Computing Machinery, New York, NY, USA, of ICPP '21, pp. 1–11.
- [105] J. Finkelstein, J. S. Smith, S. M. Mniszewski, K. Barros, C. F. A. Negre, E. H. Rubensson, A. M. N. Niklasson, *J. Chem. Theory Comput.* **2021**, 17, 2256–2265.
 - [106] J. Finkelstein, J. S. Smith, S. M. Mniszewski, K. Barros, C. F. A. Negre, E. H. Rubensson, A. M. N. Niklasson, *J. Chem. Theory Comput.* **2021**, 17, 6180–6192.
 - [107] J. Finkelstein, E. H. Rubensson, S. M. Mniszewski, C. F. A. Negre, A. M. N. Niklasson, *J. Chem. Theory Comput.* **2022**, 18, 4255–4268.
 - [108] J. Finkelstein, C. F. A. Negre, J.-L. Fattebert, *J. Chem. Phys.* **2023**, 159, 101101.
 - [109] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmairson, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhersch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, S. Chintala in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, Association for Computing Machinery, New York, NY, USA, Vol. 2 of ASPLOS '24, pp. 929–947.
 - [110] M. Friede, C. Hölzer, S. Ehlert, S. Grimme, *J. Chem. Phys.* **2024**, 161, 062501.
 - [111] A. Katbashev, M. Stahn, T. Rose, V. Alizadeh, M. Friede, C. Plett, P. Steinbach, S. Ehlert, *J. Phys. Chem. A* **2025**, 129, 2667–2682.
 - [112] F. Neese, *WIREs Comput. Mol. Sci.* **2012**, 2, 73–78.
 - [113] F. Neese, *WIREs Comput. Mol. Sci.* **2018**, 8, e1327.
 - [114] J. P. Unsleber, T. Dresselhaus, K. Klahr, D. Schnieders, M. Böckers, D. Barton, J. Neugebauer, *Journal of Computational Chemistry* **2018**, 39, 788–798.
 - [115] N. Niemeyer, P. Eschenbach, M. Bensberg, J. Tölle, L. Hellmann, L. Lampe, A. Massolle, A. Rikus, D. Schnieders, J. P. Unsleber, J. Neugebauer, *WIREs Computational Molecular Science* **2023**, 13, e1647.
 - [116] C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, *ACM Trans. Math. Softw.* **1979**, 5, 308–323.

- [117] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, *ACM Trans. Math. Softw.* **1988**, *14*, 1–17.
- [118] J. J. Dongarra, J. Du Croz, S. Hammarling, I. S. Duff, *ACM Trans. Math. Softw.* **1990**, *16*, 1–17.
- [119] <https://docs.nvidia.com/cuda/cublas/contents.html>, (last accessed: 20.10.2025).
- [120] J.-L. Fattebert, C. F. A. Negre, J. Finkelstein, J. Mohd-Yusof, D. Osei-Kuffuor, M. E. Wall, Y. Zhang, N. Bock, S. M. Mniszewski, *J. Chem. Phys.* **2024**, *160*, 122501.
- [121] U. Borštnik, J. VandeVondele, V. Weber, J. Hutter, *Parallel Comput.* **2014**, *40*, 47–58.
- [122] O. Schütt, P. Messmer, J. Hutter, J. VandeVondele in *Electronic Structure Calculations on Graphics Processing Units*, John Wiley & Sons, Ltd, **2016**, chapter 8, pp. 173–190.
- [123] F. Neese, *J. Comput. Chem.* **2023**, *44*, 381–396.
- [124] S. Lehtola, *J. Chem. Phys.* **2023**, *159*, 180901.
- [125] <https://github.com/lanl/bml>, **2025**, (last accessed: 20.10.2025).
- [126] C. Brown, A. Abdelfattah, S. Tomov, J. Dongarra in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7.
- [127] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki in *Numerical Computations with GPUs*, V. Kindratenko (Ed.), Springer International Publishing, Cham, **2014**, pp. 3–28.
- [128] <https://github.com/simunova/mtl4>, **2025**, (last accessed: 20.10.2025).
- [129] Unified Acceleration (UXL) Foundation, **2025**, (last accessed: 20.10.2025).
- [130] <https://github.com/tblite/tblite>, (last accessed: 20.10.2025).
- [131] <https://git.rwth-aachen.de/bannwarthlab/gambits>, (last accessed: 20.10.2025).
- [132] P. Pulay, *J. Comput. Chem.* **1982**, *3*, 556–560.
- [133] C. Bannwarth, T. J. Martínez, *J. Chem. Phys.* **2023**, *158*.
- [134] J. L. Risco-Martín, J. M. Colmenar, D. Atienza, J. I. Hidalgo, *Microprocessors and Microsystems* **2011**, *35*, 755–765.

- [135] J. Chang, E. Gehringer, *IEEE Transactions on Computers* **1996**, 45, 357–366.
- [136] D. Tomanek, N. Frederick, <https://nanotube.msu.edu/fullerene/fullerene-isomers.html>, (last accessed: 20.10.2025).
- [137] P. Steinbach, C. Bannwarth, *J. Chem. Theory Comput.* **2025**, 21, 7335–7351.
- [138] A. D. Daniels, G. E. Scuseria, *J. Chem. Phys.* **1999**, 110, 1321–1328.
- [139] J. Kussmann, M. Beer, C. Ochsenfeld, *WIREs Comput. Mol. Sci.* **2013**, 3, 614–636.
- [140] D. R. Bowler, T. Miyazaki, *Rep. Prog. Phys.* **2012**, 75, 036503.
- [141] E. Rudberg, E. H. Rubensson, *J. Phys.: Condens. Matter* **2011**, 23, 075502.
- [142] R. McWeeny, *Phys. Rev.* **1962**, 126, 1028–1034.
- [143] X.-P. Li, R. W. Nunes, D. Vanderbilt, *Phys. Rev. B* **1993**, 47, 10891–10894.
- [144] M. S. Daw, *Phys. Rev. B* **1993**, 47, 10895–10898.
- [145] J. M. Millam, G. E. Scuseria, *J. Chem. Phys.* **1997**, 106, 5569–5577.
- [146] A. D. Daniels, J. M. Millam, G. E. Scuseria, *J. Chem. Phys.* **1997**, 107, 425–431.
- [147] M. Challacombe, *J. Chem. Phys.* **1999**, 110, 2332–2342.
- [148] D. R. Bowler, M. J. Gillan, *Comput. Phys. Comm.* **1999**, 120, 95–108.
- [149] L. A. Truflandier, R. M. Dianzinga, D. R. Bowler, *J. Chem. Phys.* **2016**, 144, 091102.
- [150] J. VandeVondele, J. Hutter, *J. Chem. Phys.* **2003**, 118, 4365–4369.
- [151] Y. Shao, C. Saravanan, M. Head-Gordon, C. A. White, *J. Chem. Phys.* **2003**, 118, 6144–6151.
- [152] T. Helgaker, P. Jorgensen, J. Olsen, *Molecular Electronic-Structure Theory*, John Wiley & Sons, **2013**.
- [153] S. A. Gershgorin, *Bull. Russ. Acad. Sci.* **1931**, 749–754.
- [154] S. Goedecker, L. Colombo, *Phys. Rev. Lett.* **1994**, 73, 122–125.
- [155] S. Goedecker, M. Teter, *Phys. Rev. B* **1995**, 51, 9455–9464.

- [156] R. Baer, M. Head-Gordon, *J. Chem. Phys.* **1997**, *107*, 10003–10013.
- [157] H. Röder, R. N. Silver, D. A. Drabold, J. J. Dong, *Phys. Rev. B* **1997**, *55*, 15382–15385.
- [158] S. M. Mniszewski, R. Perriot, E. H. Rubensson, C. F. A. Negre, M. J. Cawkwell, A. M. N. Niklasson, *J. Chem. Theory Comput.* **2019**, *15*, 190–200.
- [159] A. H. R. Palser, D. E. Manolopoulos, *Phys. Rev. B* **1998**, *58*, 12704–12711.
- [160] A. M. N. Niklasson, *Phys. Rev. B* **2002**, *66*, 155115.
- [161] A. M. N. Niklasson, C. J. Tymczak, M. Challacombe, *J. Chem. Phys.* **2003**, *118*, 8611–8620.
- [162] A. Holas, *Chem. Phys. Lett.* **2001**, *340*, 552–558.
- [163] D. A. Mazziotti, *Phys. Rev. E* **2003**, *68*, 066701.
- [164] A. M. N. Niklasson, C. J. Tymczak, H. Röder, *Phys. Rev. B* **2002**, *66*, 155120.
- [165] H. J. Xiang, W. Z. Liang, J. Yang, J. G. Hou, Q. Zhu, *J. Chem. Phys.* **2005**, *123*, 124105.
- [166] E. H. Rubensson, *J. Chem. Theory Comput.* **2011**, *7*, 1233–1236.
- [167] E. H. Rubensson, A. M. N. Niklasson, *SIAM J. Sci. Comput.* **2014**, *36*, B147–B170.
- [168] M. J. Cawkwell, E. J. Sanville, S. M. Mniszewski, A. M. N. Niklasson, *J. Chem. Theory Comput.* **2012**, *8*, 4094–4101.
- [169] R. Seeger, J. A. Pople, *J. Chem. Phys.* **1976**, *65*, 265–271.
- [170] G. B. Bacskay, *Chem. Phys.* **1981**, *61*, 385–404.
- [171] M. Head-Gordon, J. A. Pople, *J. Phys. Chem.* **1988**, *92*, 3063–3069.
- [172] R. Shepard, *Theoret. Chim. Acta* **1993**, *84*, 343–351.
- [173] C. Seidl, G. M. J. Barca, *J. Chem. Theory Comput.* **2022**, *18*, 4164–4176.
- [174] <https://github.com/grimme-lab/crest>, (last accessed: 20.10.2025).
- [175] <https://git.rwth-aachen.de/bannwarthlab/crest>, (last accessed: 20.10.2025).

- [176] S. Ehlert, M. Stahn, S. Spicher, S. Grimme, *J. Chem. Theory Comput.* **2021**, *17*, 4250–4261.
- [177] J. Seibert, C. Bannwarth, S. Grimme, *J. Am. Chem. Soc.* **2017**, *139*, 11682–11685.
- [178] A. Habib, J. Finkelstein, A. M. N. Niklasson, *J. Chem. Theory Comput.* **2024**, *20*, 7102–7112.
- [179] B. de Souza, *Angew. Chem. - Int. Ed.* **2025**, *64*, e202500393.
- [180] S. G. Balasubramani, G. P. Chen, S. Coriani, M. Diedenhofen, M. S. Frank, Y. J. Franzke, F. Furche, R. Grotjahn, M. E. Harding, C. Hättig, A. Hellweg, B. Helmich-Paris, C. Holzer, U. Huniar, M. Kaupp, A. Marefat Khah, S. Karbalaei Khani, T. Müller, F. Mack, B. D. Nguyen, S. M. Parker, E. Perlt, D. Rapoport, K. Reiter, S. Roy, M. Rückert, G. Schmitz, M. Sierka, E. Tapavicza, D. P. Tew, C. van Wüllen, V. K. Voora, F. Weigend, A. Wodyński, J. M. Yu, *J. Chem. Phys.* **2020**, *152*, 184107.
- [181] Y. J. Franzke, C. Holzer, J. H. Andersen, T. Begušić, F. Bruder, S. Coriani, F. Della Sala, E. Fabiano, D. A. Fedotov, S. Fürst, S. Gillhuber, R. Grotjahn, M. Kaupp, M. Kehry, M. Krstić, F. Mack, S. Majumdar, B. D. Nguyen, S. M. Parker, F. Pauly, A. Pausch, E. Perlt, G. S. Phun, A. Rajabi, D. Rapoport, B. Samal, T. Schrader, M. Sharma, E. Tapavicza, R. S. Treß, V. Voora, A. Wodyński, J. M. Yu, B. Zerulla, F. Furche, C. Hättig, M. Sierka, D. P. Tew, F. Weigend, *J. Chem. Theory Comput.* **2023**, *19*, 6859–6890.
- [182] R. A. Friesner, *Chem. Phys. Lett.* **1985**, *116*, 39–43.
- [183] P. Plessow, F. Weigend, *J. Comput. Chem.* **2012**, *33*, 810–816.
- [184] F. Liu, J. Kong, *Chem. Phys. Lett.* **2018**, *703*, 106–111.
- [185] H. Laqua, T. H. Thompson, J. Kussmann, C. Ocsenfeld, *J. Chem. Theory Comput.* **2020**, *16*, 1456–1468.
- [186] C. Adamo, V. Barone, *J. Chem. Phys.* **1999**, *110*, 6158–6170.
- [187] N. Mardirossian, M. Head-Gordon, *Phys. Chem. Chem. Phys.* **2014**, *16*, 9904–9924.
- [188] J. P. Perdew, K. Burke, M. Ernzerhof, *Phys. Rev. Lett.* **1996**, *77*, 3865–3868.
- [189] D. Andrae, U. Haeussermann, M. Dolg, H. Stoll, H. Preuss, *Theor. Chim. Acta* **1990**, *77*, 123–141.

- [190] A. Bergner, M. Dolg, W. Küchle, H. Stoll, H. Preuß, *Mol. Phys.* **1993**, *80*, 1431–1441.
- [191] <https://github.com/grimme-lab/gcp>, (last accessed: 20.10.2025).
- [192] A. Najibi, L. Goerigk, *J. Comput. Chem.* **2020**, *41*, 2562–2572.
- [193] J. L. Whitten, *J. Chem. Phys.* **1973**, *58*, 4496–4501.
- [194] K. Eichkorn, O. Treutler, H. Öhm, M. Häser, R. Ahlrichs, *Chemical Physics Letters* **1995**, *242*, 652–660.
- [195] K. Eichkorn, F. Weigend, O. Treutler, R. Ahlrichs, *Theor. Chem. Acc.* **1997**, *97*, 119–124.
- [196] F. Neese, *J. Comput. Chem.* **2003**, *24*, 1740–1747.
- [197] <https://github.com/grimme-lab/cefine>, (last accessed: 20.10.2025).
- [198] U. Huniar, *Private Communication*, **2023**.
- [199] <https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet>, (last accessed: 20.10.2025).

8 Appendix

8.1 Computational hardware

The computers used to record walltimes and other timings are deployed in a temperature-controlled room at 22°C. All machines are running under Ubuntu 20.4 LTS. All tasks are running via the SLURM workload scheduler. During timings no other user-controlled tasks are running on the machines. The following table gives an overview on the various hardware setups used in this work:

Tab. 8.1: Hardware components of the computers used in this dissertation for timing purposes.

Setup	CPU	N _{Threads}	CPU RAM	GPU
1	Intel® Xeon® w7-2495X	48	256 GB	NVIDIA® RTX 4090 (24 GB VRAM)
2	Intel® i7-11700	16	32 GB	NVIDIA® RTX 3070 (8 GB VRAM)
3	AMD® Threadripper 3975WX	64	256 GB	NVIDIA® RTX 3080Ti (12 GB VRAM)
4	Intel® Xeon® w7-2255	20	64 GB	NVIDIA® RTX 3080Ti (12 GB VRAM)
5	Intel® Xeon® w7-2255	20	64 GB	NVIDIA® Quadro RTX 5000 (16 GB VRAM)

8.2 Combining low-cost electronic structure theory and low-cost parallel computing architecture

8.2.1 Implementation details and element availability

Statistical measures:

- x_i denotes the computed/sample value

- r_i denotes the reference value
- n denotes the sample size
- Mean deviation (MD) : $MD = \frac{\sum_i^n x_i - r_i}{n}$
- Mean absolute deviation (MAD) : $MAD = \frac{\sum_i^n |x_i - r_i|}{n}$
- Root mean square deviation (RMSD) : $RMSD = \sqrt{\frac{\sum_i^n (x_i - r_i)^2}{n}}$
- Bessel corrected standard deviation (STD) : $STD = \sqrt{\frac{\sum_i^n (x_i - r_i - MD)^2}{n-1}}$

To check the method implementation in this work, we control the energies and gradients against the corresponding ORCA and TURBOMOLE implementations. We consider the energy as equivalent, when the absolute difference in electronic energies E_{elec} is smaller than the threshold defined in eq. 8.1. For the gradients $d\mathbf{X}$, we take the absolute difference between two quantum chemical programs and average over the number of atoms N_{atoms} in that molecule, see Equation 8.2. This measure $\Delta d\mathbf{X}$ is averaged over the Cartesian coordinates x, y, z , then computed and compared to a threshold, see eq. 8.3.

$$|E_{\text{elec,TC}} - E_{\text{elec,ORCA,TM}}| \leq 2 \cdot 10^{-4} \text{ Eh} \quad (8.1)$$

$$\sum_{\alpha \in (x,y,z)} \frac{\Delta d\mathbf{X}^\alpha}{3} \leq 1 \cdot 10^{-4} \text{ atomic units} \quad (8.2)$$

$$\text{where: } \Delta d\mathbf{X} = \sum_{i=0}^{N_{\text{atoms}}} |d\mathbf{X}_{i,\text{TC}} - d\mathbf{X}_{i,\text{ORCA,TM}}| \cdot \frac{1}{N_{\text{atoms}}} \quad (8.3)$$

The WTMAD-2 weighting scheme is defined in eq. . For this work, HEAVY28, HEAVYSB11 and RG1 could not be included due to element availability restrictions in TeraChem. This means that the summation is only performed over 52 benchmarking sets, leaving the value of $56.84 \text{ kcal mol}^{-1}$ unaltered.

$$\text{WTMAD-2} = \frac{1}{\sum_{i=1}^{55}} \sum_{i=1}^{55} N_i \frac{56.84 \text{ kcal mol}^{-1}}{|\Delta E_i| \cdot \text{MAD}_i} \quad (8.4)$$