LUT University

ON GRAPHICS PROCESSING UNITS FOR SIMULATION OF MASS SPECTRA

Lappeenranta-Lahti University of Technology LUT

Master's Program in Computational Engineering, Master's Thesis

2025

Tornike Onoprishvili

Examiners:	Professor Lasse Lensu
	Professor Ernst-Jan Camiel Wit

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Engineering Science
Computational Engineering

Tornike Onoprishvili

On Graphics Processing Units For Simulation Of Mass Spectra

Master's thesis
2025
66 pages, 22 figures, 2 tables, 0 appendices
Examiners: Professor Lasse Lensu and Professor Ernst-Jan Camiel Wit

Keywords: mass spectrometry, computational chemistry, graphics processing unit

Mass spectrometry (MS) is an important technique in analytical chemistry for identifying molecules. Accurate identification often relies on high-quality datasets of experimental mass spectra. Unfortunately, the existing datasets are not large enough to capture the full diversity of biological molecules. Expanding these datasets with new experiments is a slow and expensive process. An alternative to experimental data is the simulation of mass spectra *in silico* using computational quantum chemistry. These approaches are often computationally intensive, which limits their use to only the most well-equipped computational chemistry labs. In recent years, the mass adoption of Artificial intelligence (AI) programs has led to a soaring demand for high-performance graphics processing units (GPUs). This study demonstrates that offloading parts of the *in silico* mass spectra calculations to a GPU can lead to substantial speedups. In these settings, it is shown that a single state-of-the-art NVIDIA GPU can provide an equivalent computational power of over 300x central processing units (CPUs). In this way, it is demonstrated that the hardware typically used for AI inference can also be effectively utilized for the simulation of mass spectra.

ACKNOWLEDGEMENTS

This work would not have been possible without the unwavering support and insightful comments of my spouse Ani Talakhadze. I am deeply grateful for the steady guidance and encouragement of my academic supervisors:

- Professor Lasse Lensu
- Professor Ernst-Jan Camiel Wit
- D.Sc. (Tech.) Henri Petrow
- Professor Satu-Pia Reinikainen

AI was employed to detect misspellings, punctuation errors, and to provide critical feedback on my writing in the form of reviewer questions. Specifically, I supplied sections of my written work to the AI and asked it to critique my work. This AI-driven feedback loop prompted me to expand and improve several key sections of this thesis.

Lugano, November 3, 2025

Tornike Onoprishvili

LIST OF ABBREVIATIONS

AI	Artificial intelligence
ALU	Arithmetic logic unit
BLAS	Basic Linear Algebra Subprograms
CC	Compute Capability
CGTO	Contracted Gaussian-type orbital
CID	Collsion-induced dissociation
CLI	Command-line interface
CPU	Central processing unit
DFT	Density functional theory
DFTB	Density functional based tight binding
DNA	Deoxyribonucleic acid
DRAM	Dynamic random-access memory
EI	Electron ionization
ESI	Electrospray ionization
FLOPS	floating point operations per second
GDDR	Graphics double data rate
GNPS	Global Natural Product Social Molecular Networking
GPU	Graphics processing unit
GTO	Gaussian-type orbital
GUI	Graphical user interface
HBM	High bandwidth memory
HF	Hartree-Fock
LAPACK	Linear Algebra Package
LC	Liquid chromatography
LC-MS/MS	Liquid chromatography-tandem mass spectrometry
MD	Molecular dynamics
ML	Machine learning
MoNa	MassBank of North America
MS	Mass spectrometry
MS/MS	Tandem mass spectrometry
NCU	NVIDIA Nsight Compute
NIST	National Institute of Standards and Technology
PDB	Protein Data Bank
PES	Potential energy surface
RT	Retention time
SCC	Self-consistent cycle

SCF	Self-consistent field
SIMD	Single instruction, multiple data
SIMT	Single instruction, multiple threads
SM	Streaming multiprocessor
SMEM	Shared memory
SQM	Semi-empirical quantum mechanical
SRAM	Static random-access memory
STO	Slater-type orbital
TBlite	Light-weight tight-binding framework
vdW	van der Waals
xTB	Extended tight-binding

Nomenclature

m/z	Mass-to-charge ratio	Dalton per elementary charge
Da	Dalton	$1.660\,539\,068\,92 \times 10^{-27}\,\mathrm{kg}$
eV	Electron volt	$1.602\,176\,634 \times 10^{-19}\,\mathrm{J}$
GB	Gigabyte	2^{30} bytes = 1 073 741 824 bytes
h	Planck constant	$6.626\,070\,15 \times 10^{-34}\,\mathrm{J\,s}$
K	Kelvin (temperature)	K
kB	Kilobyte	2^{10} bytes = 1024 bytes
MB	Megabyte	2^{20} bytes = 1 048 576 bytes
ppm	Parts per million	10^{-6} (dimensionless)
TB	Terabyte	2^{40} bytes = 1 099 511 627 776 bytes
W	Watt	$1\,\mathrm{J\,s^{-1}}$

CONTENTS

1 INTRODUCTION

1.1 Background

MS is an analytical chemistry technique for identifying the molecular composition of a chemical substance. Used correctly, it allows identifying environmental pollutants, drugs and small biological molecules. Coupled with Liquid chromatography (LC), MS allows molecular analysis of complex biological products, like plant extracts and animal-derived products [1, 2].

The ultimate goal of MS is identifying the molecular structure of a sample, given a set of high-resolution mass spectra. A typical approach to deducing the structure relies on the fact that similar molecules yield similar mass spectra. By comparing the given mass spectra to a high-quality curated dataset of known molecules, researchers are able to narrow down the structure hypothesis of the sample. MS datasets like National Institute of Standards and Technology (NIST) [3], MassBank of North America (MoNa) [4] have historically been limited in size, owing to the high cost and labor associated with operating mass spectrometers. The size of the available annotated data is a major bottleneck in identification of novel chemical structures in common biological samples.

Various computation-based solutions have been developed in order to alleviate this bottleneck. There have been attempts at using machine learning based approaches for predicting mass spectra from structures. Examples include graph neural network [5], support vector machine [6], and transformer [7], to name a few. Crucially, all these data-centric methods are bound by the availability of training data, severely limiting their ability to generalize to unseen data. Adequately representing the interesting chemical space of the target molecules is considered to be still beyond the reach of these methods.

Compared to data-centric approaches, there has been a relatively little research activity on approaching this problem *in silico* with computational quantum chemistry. QCxMS [8], a FORTRAN software for simulating the physical processes inside a mass spectrometer is currently the only such example. QCxMS has been used successfully to simulate various modes of MS [9–11]. However, a significant limitation of QCxMS is its computational intensity. A typical computational requirement of QCxMS is measured in CPU-days, per single spectrum. This limits the applicability of QCxMS to only researchers with access to high-end computational hardware.

Driven by the increasing use of AI and the demands of video games, recent years have seen explosive growth in the performance of Graphics processing units (GPUs). Modern hardware is capable of performing massive numbers of floating point operations per second (FLOPS). Certain algorithms achieve speedups of several orders of magnitude on this hardware compared to conventional Central processing units (CPUs). Efforts have been made to leverage this processing power for quantum chemistry calculations, as demonstrated by GPU4PySCF [12]. However, no GPU-based software for the *in silico* generation of mass spectra has been identified in the existing literature. The focus of this study is illustrated in Figure 1.

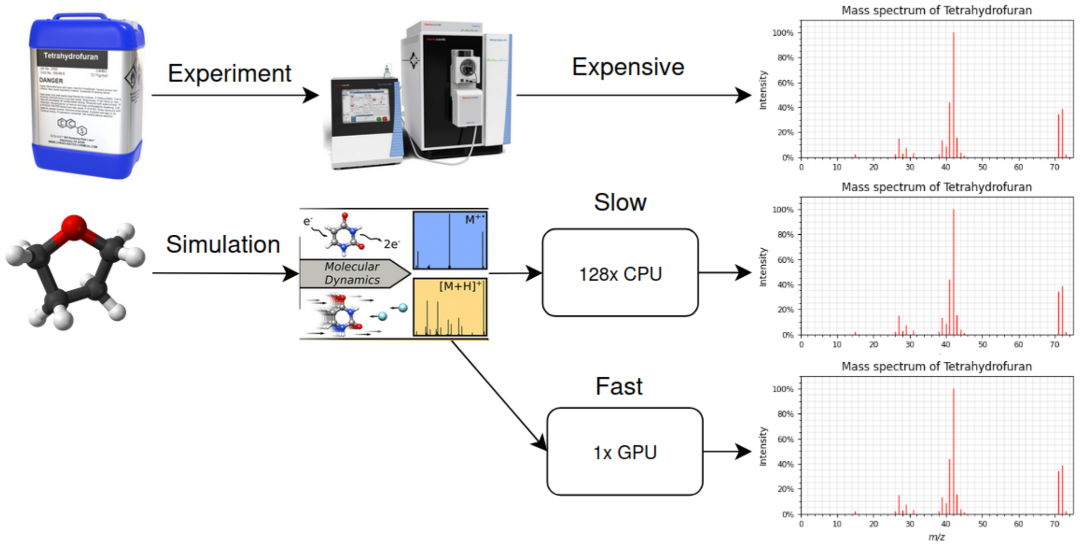


Figure 1. Mass spectrometry (MS) is an analytical chemistry technique that allows identification of chemical substances via mass spectra. Computational chemistry can simulate mass spectra *in silico* but typically require super-computing resources. This work focuses on using Graphics processing unit (GPU) to simulate mass spectra, without using large CPU clusters. [8]

1.2 Objectives and delimitations

This study explores how the computational power of modern, widely available GPUs, including those found in gaming laptops and desktop computers, can be harnessed to efficiently calculate *in silico* mass spectra. Specifically, it demonstrates that key steps in generating *in silico* mass spectra can be accelerated using such GPUs. By rewriting performance-critical calculation routines to utilize GPU acceleration, the objective of this work is to make *in silico* mass spectra calculations faster and more accessible to researchers worldwide.

The main objectives of this study are as follows:

- Literature review of the computational methods related to the *in silico* generation of mass spectra.
- The computational profile analysis of QCxMS and identification of performance bottlenecks.
- Efficient GPU implementation of performance-critical segments of QCxMS.
- Performance evaluation of the implementation on several commonly-used GPUs.

It is important to note that the aim of this study is *not* to develop new theoretical approaches for *in silico* mass spectra generation, but rather to redesign existing software to take advantage of modern GPU hardware, thereby improving accessibility and performance.

1.3 Structure of the thesis

This thesis is organized into six main sections, each addressing a specific aspect of the work. Section 1 introduces the background, motivation, objectives, and delimitations of the study, outlining the challenges in MS and the motivation for GPU acceleration. Section 2 reviews the essential concepts in chemistry, MS, computational quantum chemistry, and GPU programming, and surveys the state-of-the-art methods and software relevant to *in silico* mass spectra generation. Section 3 describes the profiling of QCxMS, identification of computational bottlenecks, and the design and implementation of GPU-accelerated routines, detailing the technical challenges, development workflow, and validation strategies. Section 4 presents the experimental setup, validation of the GPU implementation, and performance benchmarks on various molecular systems and hardware platforms, analyzing computational limits and discussing the impact of GPU acceleration. Section 5 interprets the results, discusses the implications and limitations of the work. Finally, Section 6 summarizes the main contributions and results of the thesis and outlines potential directions for future research and development.

2 RELATED WORK

This section introduces important concepts from chemistry, MS, computational quantum chemistry and GPU programming. Each of these fields contains a lifetime's worth of learning material, making it infeasible to describe them comprehensively here. The reader is encouraged to consult the referenced literature for an in-depth understanding of each field.

2.1 Chemistry

Atoms are composed of positively charged nuclei and negatively charged clouds of electrons. Atoms that share some of their electrons are said to form chemical bonds. Collections of atoms connected by chemical bonds form molecules. Each atom in a molecule shares some of its electrons with the others, resulting in a uniform electronic cloud across the molecule. Electrons that form bonds between atoms are called valence electrons [13].

Molecules are three-dimensional entities characterized by both a specific spatial arrangement its atoms (shape) and a measurable mass. The mass of a molecule is often expressed in Dalton (Da), where 1 Da is equivalent to one-twelfth the mass of a carbon-12 atom [1]. The mass of a molecule is approximately equal to the sum of the masses of its atomic nuclei and electrons. Electrons are extremely light, with a mass of about 0.0005 Da each.

The shape of small molecules is mainly dictated by electrostatic interactions between atoms, which arise from the repulsion and attraction between electrons and nuclei [14]. However, molecular shape is also influenced by van der Waals (vdW) forces, which are weak interactions arising quantum mechanical fluctuations in electron clouds. Unlike electrostatic forces, vdW forces can be attractive instead of repulsive [15]. In this study, molecule identification refers to determining the shape of a molecule, which is influenced by both electrostatic and vdW interactions.

Molecules can gain or lose an electric charge through a process known as ionization, which involves acquiring or losing a charge carrier. A charge carrier can be an ion itself or an electron. When the charge carrier is an added ion, the product is called an adduct. For example, the adduct [M+H] refers to a molecule with one attached proton and has +1 charge. The energy changes involved in these processes are often measured in electron volts (eV), a unit that represents the amount of kinetic energy gained or lost by a single electron moving through an electric potential difference of one volt. Ions can be manipulated using electric and magnetic fields, a property that is fundamental to MS [1].

2.2 Mass spectrometry

Mass spectrometry (MS) is an analytical tool used in chemistry to identify molecules [1]. The device employed in this technique is called a mass spectrometer. A mass spectrometer uses electric and magnetic fields to focus a beam of ions into a perpendicular magnetic field, causing the ions to arc based on their mass-to-charge ratio (m/z). A detector surface then measures the m/z values and the number of ions striking its surface, producing a mass spectrum of the molecule. The charge of an ion in MS is typically +1 or -1. In this case, m/z is equivalent to the mass of the ion m. A visual schematic of a mass spectrometer is given in Figure 2.

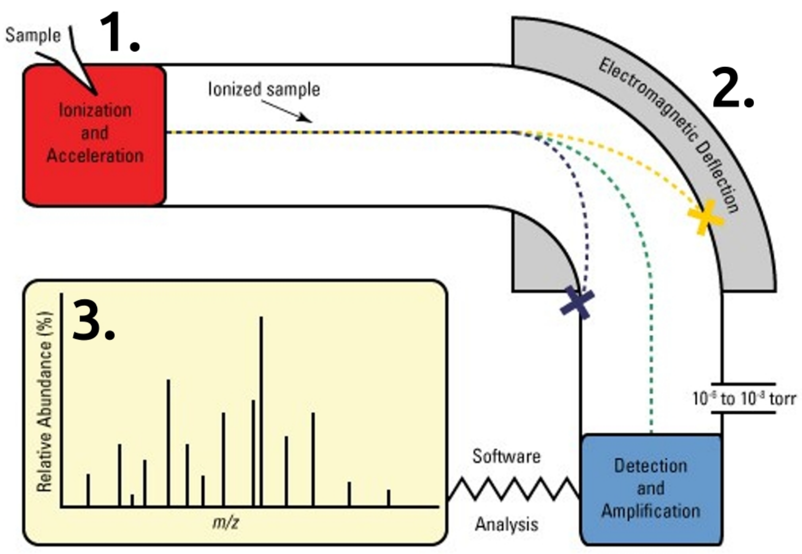
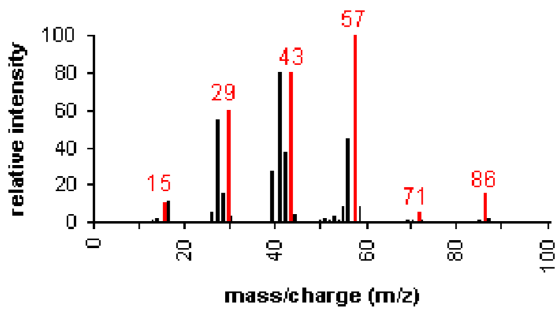


Figure 2. Schematic diagram of a mass spectrometer, showing main parts. A sample is first injected into the device and ionized (1). The ionized sample is then separated into ion beams of different mass-to-charge (m/z) ratio using a sector electromagnet (2). The device allows the selection of a range of m/z values that are directed towards a detector for recording (dashed green line). On-board software analyses the raw detector data and outputs a mass spectrum (3). The device requires a strong vacuum to operate. [16]

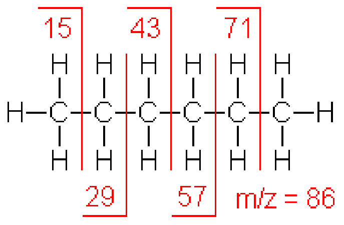
When a sample contains a mix of ions from different molecules, the resulting mass spectrum becomes complex, complicating molecule identification. This is referred to as contamination and it poses a significant challenge in MS [1]. Since biological samples are typically mixtures, LC is often used prior to MS to separate the mixture into groups of identical molecules based on their Retention time (RT). Retention time is the duration between injecting the sample into the LC inlet and detecting its output. Molecules with the same retention time have the same shape.

Notably, LC cannot ionize the molecules, which is required for MS [1]. As a result, liquid chromatography is usually combined with MS in a method called Liquid chromatography-tandem mass spectrometry (LC-MS/MS). LC-MS/MS systems incorporate an ionization source between the LC and MS stages. Common ionization sources include Electron ionization (EI) and Electrospray ionization (ESI). The ions generated are then directed into the mass spectrometer using electromagnetic fields.

As mentioned before, mass spectrometer can only measure the mass-to-charge ratio of ions and not their structure [1]. This makes structure elucidation with MS very challenging. To gain additional information about the structure,Tandem mass spectrometry (MS/MS) is often used. In MS/MS, ions within a narrow band around m/z are selected. These ions are referred to as the precursor ions and this m/z value is referred to as precursor m/z. After this selection, the ions undergo fragmentation. Fragmentation can happen in two ways: ions are either bombarded with a beam of electrons or ions are accelerated and collided with gas molecules. The former is called Electron ionization (EI) and the latter is Collsion-induced dissociation (CID). The resulting fragments of molecules are then fed into a coupled mass spectrometer to produce the tandem mass spectrum. An example of a tandem mass spectrum of a simple alkane (C_6H_{14}) is shown in Figure 3.



(a)



(b)

Figure 3. Mass spectrum (a) of a simple molecule like hexane (b) is a histogram of m/z values that can be used to identify the molecule. For example, the rightmost red peak with m/z of 86 and intensity of ≈20% corresponds an intact hexane *ion* hitting the mass spectrometer detector since the mass of hexane is known to be 86 Da. Similarly, a peak at m/z of 43 corresponds to a hexane ion that was fragmented in half. Black, unannotated peaks in (a) correspond to fragments that have lost one or more of hydrogen atoms. [17]

Mass spectrometers have an inherent measurement error, known as tolerance, typically expressed in parts per million (ppm). Tolerance determines whether two m/z masses are close enough to be considered practically equivalent. To compare mass spectra, a similarity measure is required. One commonly used metric is the *dot product*, also known as the *cosine similarity* or *cosine score* [18]. The dot product between two spectra ranges from 0 to 1, where values above 0.7 generally indicate strong similarity.

Dot product similarity is frequently used in molecule identification. Molecule identification using mass spectral similarity is based on the idea that inside a mass spectrometer, molecules undergo fragmentation in a reproducible manner, producing a fragment peak 'fingerprint.' Matching against a spectral library of experimental mass spectra with known chemicals allows researchers to narrow down structural hypotheses [19, 20].

Modern tandem mass spectrometers are high-throughput devices [21]. An LC-MS/MS experiment can process thousands of mass spectra per hour. Commercial mass spectrometers typically support multiple modes and levels of operation, each of which can yield vastly different results for the same molecule. For example, ESI spectrum is usually not easily comparable to EI spectrum of the same molecule, as illustrated in Figure 4.

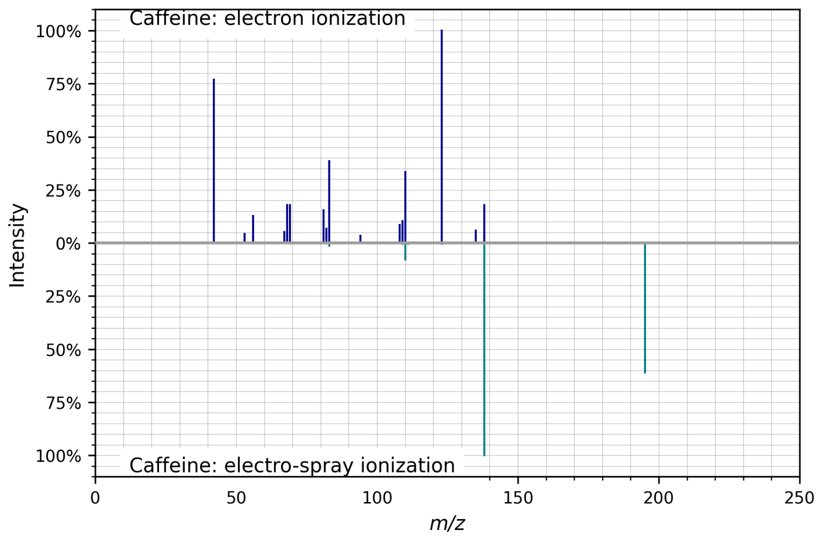


Figure 4. Mass spectrum of a caffeine molecule in Electron ionization mode (top) and Electrospray ionization mode (bottom).

Likewise, Collsion-induced dissociation (CID) at different collision energies, for example 30 eV and 70 eV can be quite dissimilar. Additionally, the contamination of samples, physical measurement inaccuracies, and device defects at any stage of the LC-MS/MS process can significantly affect the spectra obtained.

2.3 Computational quantum chemistry

Computational chemistry is a study of computational methods for simulation of various chemical phenomena. It has nearly endless applications ranging from materials science to industrial chemistry to drug design [22–24].

Computational chemistry encompasses several branches, each with distinct practical applications and frequent overlap. The branches most relevant to this work include:

- **Computational quantum chemistry** focuses on solving the Schrödinger equation for molecular systems [13]. Common methods include Hartree-Fock (HF), Density functional theory (DFT), and Density functional based tight binding (DFTB).
- **Molecular dynamics (MD)** studies the movement of atoms and molecules over time [25]. Molecular dynamics (MD) treats nuclei as classical point particles, neglecting quantum effects. MD is an efficient simulation method for molecular systems where effects can be safely ignored. It is used in biochemistry, drug design, and materials science.
- **Computational thermodynamics** predicts chemical reactions and reaction energies [26], with applications in industrial process optimization.
- **Cheminformatics** involves storing and analyzing chemical data for data-driven models, AI, and Machine learning (ML) to predict chemical properties and interactions [27]. It is used in drug discovery, material design, and reaction prediction.

This work is concerned with simulating Mass spectrometry (MS) which, as mentioned in the previous section, consists of three main parts: ionization of a sample, excitation of an ion, and subsequent dissociation of the ion into fragments. In principle, all of these processes can be simulated using quantum chemistry to achieve highly accurate results [13]. In reality, the required computational quantum chemistry methods are extremely computationally intensive, making exact simulations impractical for all but the simplest systems [28]. As a result, a variety of approximation methods have been developed to make these simulations tractable.

At the heart of quantum chemistry lies the time-dependent **Schrödinger equation**, which governs the evolution of quantum systems such as electrons, atoms, and molecules [13],

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \hat{H}\Psi(\mathbf{r},t) \tag{1}$$

where $\Psi(\mathbf{r},t)$ is the wavefunction, $\hat{H}$ is the Hamiltonian operator (total energy), i is the imaginary unit, and $\hbar$ is the reduced Planck constant. For stationary states, the time-independent form is used:

$$\hat{H}\Psi(\mathbf{r}) = E\Psi(\mathbf{r}) \tag{2}$$

where E is the energy eigenvalue. The **Hamiltonian** for a molecular system includes kinetic energy of electrons and nuclei, electron-nucleus attraction, electron-electron repulsion, and nucleus-nucleus repulsion.

As mentioned above, closed-form solutions to Eq. 2 exist for only the simplest systems (e.g., the hydrogen atom), while more complex systems, like molecules require approximate computational approaches. Computational quantum chemistry employs a number of such approximations [13,29]. These can be broadly divided into two categories: ***ab initio*** methods, which solve the Schrödinger equation from first principles without empirical parameters, and **Semi-empirical quantum mechanical (SQM)** methods, which introduce experimental (empirical) data or further approximations to simplify calculations and reduce computational cost. *Ab initio* methods are generally more accurate but much slower than SQMs.

The **Born–Oppenheimer approximation** is a foundational concept in computational quantum chemistry that takes advantage of the fact that electrons are much lighter and move much faster than nuclei. In this approximation, the nuclei are treated as completely fixed while solving the electronic Schrödinger equation. This allows the electronic structure to be determined for a given nuclear configuration, yielding the **Potential energy surface (PES)**, which can then be used to study nuclear dynamics. The main computational effort lies in solving the electronic Schrödinger equation for the fixed nuclear configuration [13].

One of the earliest and most important *ab initio* methods is the **Hartree-Fock (HF)** theory [13]. HF uses an independent-particle model, where each electron is described by an **orbital**, and the total wavefunction is constructed as a Slater determinant to enforce the Pauli exclusion principle.

Electrons obey Pauli exclusion principle. This means that no two electrons can occupy the same quantum state. This requirement is conveniently enforced by representing the combined electron wavefunction as a determinant called **Slater determinant** [13]. The

Slater determinant for N electrons can be written as:

$$\Psi(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(\mathbf{x}_1) & \psi_2(\mathbf{x}_1) & \cdots & \psi_N(\mathbf{x}_1) \\ \psi_1(\mathbf{x}_2) & \psi_2(\mathbf{x}_2) & \cdots & \psi_N(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\mathbf{x}_N) & \psi_2(\mathbf{x}_N) & \cdots & \psi_N(\mathbf{x}_N) \end{vmatrix} \tag{3}$$

where Ψ is the combined wavefunction, and $\psi_i(\mathbf{x}_j)$ are the single-electron orbitals, with i labeling the orbital and j labeling the electron. Swapping any two electrons (i.e., exchanging two rows) changes the sign of the full wavefunction Ψ, thus ensuring the antisymmetric nature required for fermions. Additionally, if any two electrons occupy the same orbital, two rows in the determinant become identical, causing the wavefunction to vanish, which directly enforces the exclusion principle.

In practice, the electronic wavefunction is expanded in a basis set, most commonly composed of **Gaussian-type orbitals (GTOs)**. A Gaussian-type orbital (GTO) is a mathematical function of the form $e^{-\alpha r^2}$, where α controls the width and r is the distance from the nucleus. GTOs are favored because they allow efficient computation of the integrals required in electronic structure methods. However, real atomic orbitals are better described by **Slater-type orbitals (STOs)**, which have the form $e^{-\zeta r}$ and exhibit the correct cusp at the nucleus and exponential decay at large distances. To better approximate STO, **Contracted Gaussian-type orbital (CGTO)** is formed as a fixed linear combination of several primitive GTOs, typically between 3 and 6 per orbital [13]. An example of one such approximation is shown in Figure 5.

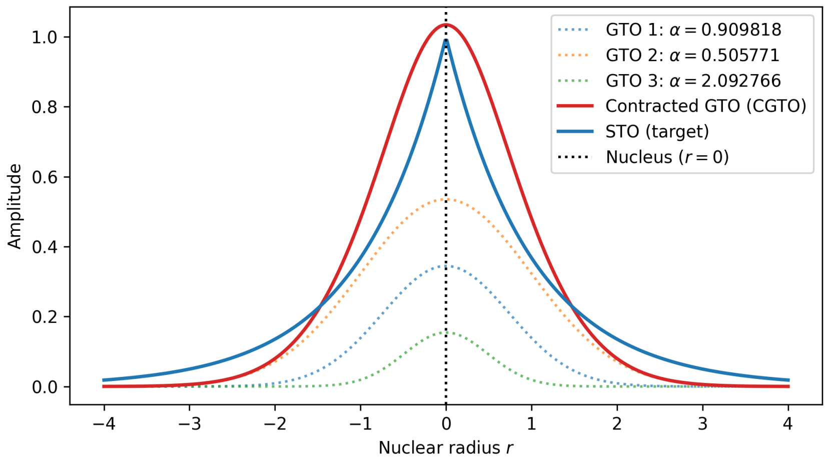


Figure 5. Slater-type orbital (STO) and its Contracted Gaussian-type orbital (CGTO) approximation consisting of primitive Gaussian-type orbitals (GTOs).

With the molecular orbitals expressed as linear combinations of these (contracted) Gaussian-type basis functions, the HF equations can be reformulated as a matrix eigenvalue problem. This leads to the **Roothaan–Hall equations**:

$$\mathbf{FC} = \varepsilon\mathbf{SC} \tag{4}$$

where **F** is the one-electron **effective Hamiltonian** (also called the Fock matrix), **C** contains the molecular orbital coefficients, **S** is the overlap matrix between basis functions, and ε is a diagonal matrix of orbital energies. These equations are solved by repeated diagonalization of the Fock matrix until self-consistency is achieved, a process known as the **Self-consistent field (SCF)** or **Self-consistent cycle (SCC)** procedure. While SCF is essential for meaningful solutions, it is computationally expensive and may not always converge, especially for complex systems, like metal complexes and transition structures [13].

HF theory serves as a branching point for further developments. It can be improved by including additional determinants (post-HF methods) to approach the exact solution, or simplified by neglecting certain integrals, leading to semi-empirical methods. For example, SQMs often neglect integrals involving more than two electrons to further reduce computational cost [13].

A major advance beyond HF is **Density Functional Theory (DFT)**, which models the many-body effects of electron correlation as a functional of the electron density rather than the wavefunction. The central equation of DFT is the Kohn–Sham equation:

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{eff}}(\mathbf{r})\right]\psi_i(\mathbf{r}) = \epsilon_i\psi_i(\mathbf{r}) \tag{5}$$

where $V_{\text{eff}}(\mathbf{r})$ is the effective potential, including external, Hartree, and exchange-correlation contributions, and $\psi_i(\mathbf{r})$ are the Kohn–Sham orbitals. DFT is computationally comparable to HF but typically yields much better results for a wide range of systems. It is widely used for predicting molecular geometries, reaction energies, and electronic properties in chemistry and materials science. However, standard DFT methods struggle to accurately describe **dispersion** interactions, which are important for weakly bound systems. To address this, empirical dispersion corrections have been developed, such as those proposed by Grimme, leading to DFT-D (DFT with dispersion) methods [13, 15]. The DFT-D energy is typically written as:

$$E_{\text{DFT-D}} = E_{\text{DFT}} + E_{\text{disp}} \tag{6}$$

where E_{disp} is an empirical correction term accounting for dispersion interactions.

To further increase efficiency, especially for large systems, SQM methods based on DFT have been developed. One prominent example is **Density functional based tight binding (DFTB)**, which simplifies DFT by neglecting three- and four-center integrals and considering only valence electrons explicitly. DFTB and its extensions, such as Extended tight-binding (xTB), use empirical parameters and are capable of simulating systems with thousands of atoms, making them particularly useful for large-scale or molecular dynamics simulations [13, 30]. While these methods sacrifice some accuracy compared to full DFT, they enable the study of much larger systems. There is a readily available open-source library called **Light-weight tight-binding framework (TBlite)** that implements the xTB SQM method [31].

In summary, the field of computational quantum chemistry has evolved a hierarchy of methods, from the fundamental Schrödinger equation to highly efficient SQM methods, each balancing accuracy and computational cost. The choice of method depends on the size and type of the system and the required level of accuracy, with DFT-D and DFTB/xTB representing state-of-the-art compromises for large, complex molecular systems.

2.4 Graphics processing unit

A GPU is a special-purpose computer originally designed to accelerate on-screen graphics and video rendering. GPU development was primarily driven by the demands of the video gaming industry, which required increasingly sophisticated hardware for rendering complex 3D graphics [32]. However, the architectural design of GPUs has proven invaluable for compute-intensive tasks in many scientific high-performance computing applications.

The fundamental difference between GPU and CPU lies in the types of computational problems they address most efficiently. CPUs are fast, general-purpose computers used for everyday tasks such as running operating systems, enabling internet access, and managing user data. In contrast, GPUs are purpose-built for massively parallel calculations and data processing [33]. A GPU cannot run an operating system, or interact with the internet, or access the file system on its own. Instead, the CPU performs these tasks on behalf of the GPU. The CPU then delegates a computational task (also known as **kernel**) to be executed by the GPU. The GPU executes the kernel and communicates the results back to the CPU.

It is important to understand that both CPU and GPU chips rely on the same semiconductor fabrication technology and, to some degree, consist of the same building blocks, arranged differently [34]. In all cases, manufacturers of both types of chips must decide how to divide a limited chip area among various building blocks. In general, these blocks are: Control, Cache, and Arithmetic logic unit (ALU). The Control unit decodes instructions and uses other blocks for computation. The Cache can be thought of as scratchpad memory. The ALU is roughly analogous to a calculator and performs mathematical operations such as addition, multiplication, and exponentiation. Dynamic random-access memory (DRAM) is used for storing program data; it is not part of the chip, but is instead connected to it [33].

Because CPUs and GPUs serve different purposes, their designers allocate chip area differently. A CPU has a few relatively large memory caches and a single control unit. In contrast, a GPU dedicates a much larger area to many small ALUs, and relatively less area to caches [33]. These architectural differences are shown in Figure 6.

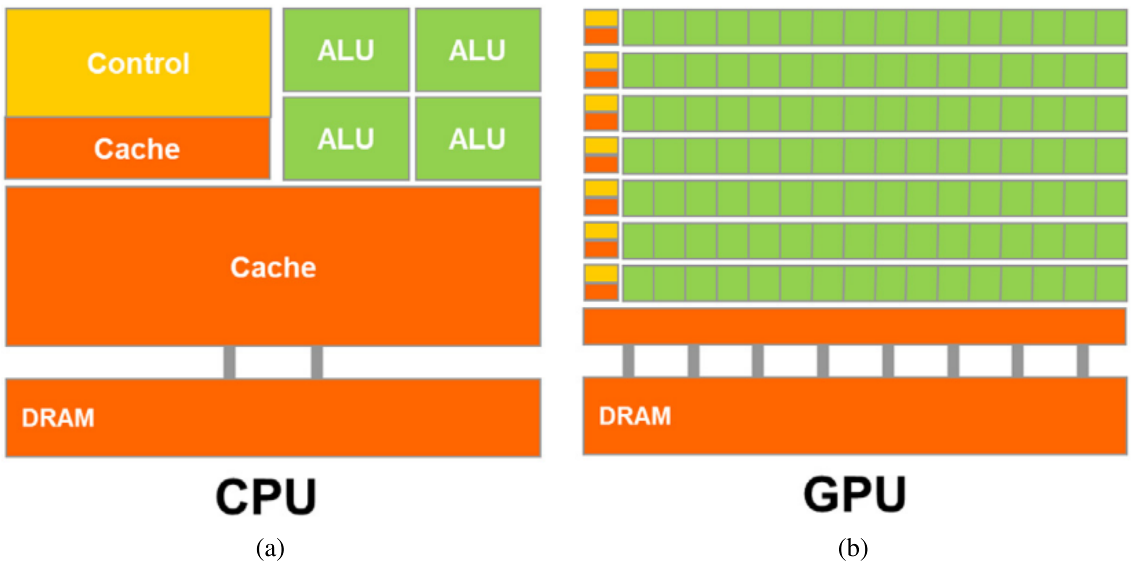


Figure 6. CPU and GPU have a fundamentally different design philosophy: (a) CPU design is latency-oriented; (b) GPU design is throughput-oriented. [33]

The evolution of early GPUs into parallel compute accelerators they are today began with the creative use of graphics programs and texture rendering pipelines [35]. Recognizing this trend, NVIDIA Corporation introduced CUDA in 2006, providing a parallel computing platform that enables direct programming of NVIDIA's GPUs using a C-like programming language called CUDA C++ [36]. The primary use of CUDA C++ is to define a GPU kernel which is executed on an NVIDIA GPU.

As mentioned above, GPU is a special-purpose computer. The GPUs specialize in executing computational tasks that exhibit one or more of the following properties:

- **Data Parallelism:** The same operation needs to be performed on many independent data elements. For example, applying a filter to every pixel in an image or adding two large arrays element-wise.
- **High Arithmetic Intensity:** The computation involves many arithmetic operations for each memory access. A good example is matrix multiplication, where each output element requires many multiplications and additions.
- **Regular Memory Access Patterns:** Memory is accessed in a predictable, sequential way, allowing efficient use of memory bandwidth. For instance, reading a large array from start to finish, such as summing all elements in a vector.

Finally, the role of the kernel engineer (i.e. the person writing a GPU kernel) can not be understated. Creating a GPU kernel with CUDA C++ is a relatively easy and straightforward task. Optimizing the GPU kernel to make good use of the GPU hardware is, however, a challenging endeavor. Optimization of a kernel requires a thorough understanding of not only the C++ and CUDA C++ programming, but also of the memory and computational architecture of the GPU [37].

In summary, GPUs are special-purpose computers that are faster than CPUs only for specific computational tasks. This GPU performance advantage does not stem from any inherent superiority of GPU technology, but rather from the fact that GPUs are better suited to perform certain calculations at the expense of all others [33]. However, this advantage in special computational tasks also heavily depends on the GPU programmer's ability to effectively parallelize the problem and utilize the GPU hardware.

2.4.1 Kernel engineering

This section provides a brief overview of GPU programming, focusing on the key concepts needed to understand the design and engineering decisions in the following sections. The aim is to give readers a foundation for how GPU programming with an NVIDIA GPU works in scientific computing contexts. For more advanced details, consult the official NVIDIA CUDA C++ Programming Guide [37], which serves as a comprehensive reference for much of the material discussed here.

A program is a sequence of instructions. The **instruction cycle** is the process by which the processor advances to the next instruction. Instructions can broadly be categorized as arithmetic instructions and memory instructions. Different instructions require appropriate chip circuitry to be executed. The **clock rate** of the processor is the number of instructions processed per second. Higher clock rates allow processors to be faster, but can also lead to lower power efficiency and overheating. **Kernel** is a program that executes on a GPU [33].

Instruction throughput is the total number of low-level instructions (e.g., `load`, `store`, `add`) executed per second [33]. **Bandwidth** refers to the maximum rate at which data can be transferred between two components in a computer system, such as between memory and a processor. **Memory latency** is the average time spent loading or storing a byte in memory. A thread can experience a **memory stall** when it attempts to `load` or `store` a memory address that is not in the **cache**. This causes a **cache miss**; conversely, if the requested address is in the cache, it is a **cache hit**. Modern processors have multiple levels of cache, named L1, L2, and sometimes even L3. Each cache level is both larger and slower than the previous one. When the cache hit rate is high, the program typically runs observably faster, without any changes to the inputs or outputs.

Programs require memory for computation, and there are different memory types. The concept of **memory hierarchy** refers to the practical reality that faster memory is generally less available. Several types of storage technologies are used in modern CPUs and GPUs, each with different bandwidth and capacity. Static random-access memory (SRAM) is a small, fast, on-chip memory. The GPU SRAM is divided into the register file and the Shared memory (SMEM). **High bandwidth memory (HBM)** and Graphics double data rate (GDDR) are types of large, off-chip memory that implement the CUDA global memory. CPU memory is the largest, but also the slowest to access. The memory hierarchy can thus be visualized as a pyramid of increasing bandwidth and decreasing capacity, as shown in Figure 7.

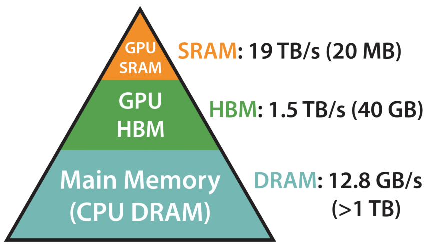


Figure 7. GPU memory hierarchy visualized as a pyramid with three levels of memory. Each successive level is both smaller than the previous one and provides higher bandwidth (numbers are examples). [38]

A **global memory** in a GPU is the largest available storage space. Different GPUs can have different types of global memory. For example, the two most common types are GDDR and HBM. GDDR memory is the cheaper of the two, requires less power, and can be found in many small GPUs in laptops and workstations. GDDR offers a moderate amount of bandwidth suitable for graphics and general-purpose workloads, typically in the range of several hundred GB/s. HBM is used in data centers and server GPUs, providing larger bandwidth, often exceeding 1 TB/s [39]. The type of memory and its bandwidth have a large impact on GPU kernel performance: higher bandwidth allows more data to be moved per unit time, reducing stalls and improving overall throughput. For a memory-bound kernel, maximizing memory bandwidth utilization is critical for achieving optimal performance [34].

Computational problems can broadly be classified as either **compute-bound** or **memory-bound** [34]. A computational problem is said to be memory-bound when the time required to complete it is determined primarily by the memory bandwidth. For example, element-wise addition of two double-precision vectors is typically memory-bound: the operation `c[i] = a[i] + b[i]` requires reading two 8-byte values and writing one, for a total of 24 bytes moved per operation, but only performs a single arithmetic addition. This results in a low **arithmetic intensity** of 1 floating-point operation per 24 bytes. Increasing arithmetic power in this case provides little benefit unless memory bandwidth is also improved [34]. Conversely, the time to complete a compute-bound computational problem is primarily determined by the number of elementary computation steps, such as additions. For example, the multiplication of large square matrices is a compute-bound problem, since it requires $O(N^2)$ memory accesses and $O(N^3)$ arithmetic operations [34].

Numerical precision refers to the binary representation of real numbers in computer systems. A common standard for floating-point arithmetic is IEEE 754, which defines formats for representing and manipulating real numbers in binary [40]. This standard specifies several levels of precision, with lower precision formats typically offering faster computations at the expense of accuracy. For example, IEEE 754 double precision, also known as FP64, encodes a real number using 64 bits (8 bytes), divided into a sign bit, exponent, and fraction (mantissa), as illustrated in Figure 8.

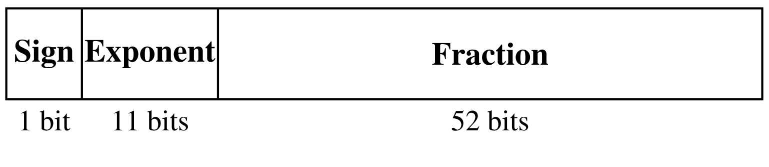


Figure 8. IEEE 754 double-precision (FP64) floating-point format: 1 sign bit, 11 exponent bits, 52 fraction bits.

Roofline model is a visual model that can be used to understand the performance limitations of a computational task; a sample roofline model is shown in Figure 9.

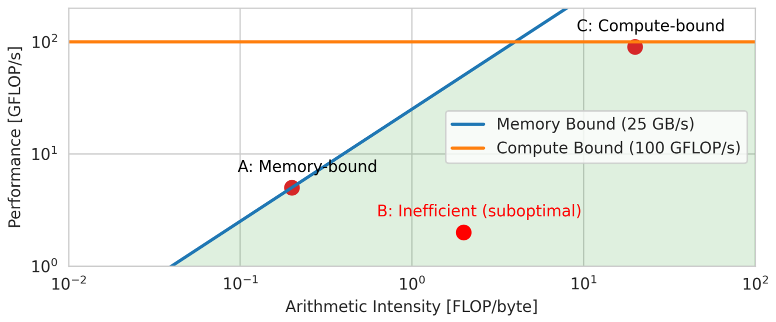


Figure 9. Example roofline model showing three applications, each exhibiting a different performance limitation.

Single instruction, multiple data (SIMD) is a parallel computing model where a single instruction operates on multiple data points simultaneously. For example, adding two arrays element-wise in a single instruction is SIMD. CUDA uses a more general model called Single instruction, multiple threads (SIMT) [37], where each thread executes the same instruction but can follow its own control flow. An example of SIMT is a CUDA kernel where each thread processes a different element of an array.

GPU programming is based on a heterogeneous computing model that consists of one or more CPU and GPU. In this model, the CPU and GPU cooperate to solve computational problems. The CPU manages resources, orchestrates data transfers, and launches computational tasks (kernels) on the GPU. The GPU, in turn, executes these kernels in parallel, leveraging its computational hardware to accelerate suitable programs. This separation of roles allows each processor to play to its strengths: the CPU handles complex logic and control flow, while the GPU excels at executing large numbers of simple, data-parallel operations [33, 37].

A key enabler of this computing model is CUDA, a proprietary parallel computing platform created by NVIDIA in 2006 [37]. The CUDA programming model provides two primary tools for programming a GPU: a C-like programming language called CUDA C++, and a runtime called the CUDA runtime. The former allows programmers to write the program logic; the latter allows executing this logic directly on NVIDIA GPU hardware. CUDA C++ is a general-purpose language, capable, in principle, of expressing any computational algorithm. However, the typical use case for CUDA C++ is to write a high-performance GPU kernel for executing computational tasks in parallel. Note that it is possible, but not practical, to perform non-parallel computational tasks on a GPU. NVIDIA supports a FORTRAN-based language called CUDA FORTRAN that can also be used for GPU programming [41].

Central to the CUDA programming model is the organization of computation into a hierarchy of grids, blocks, and threads [37]. It is helpful to present the CUDA computation organization and the three level grid, block, and thread hierarchy visually, as shown in Figure 10.

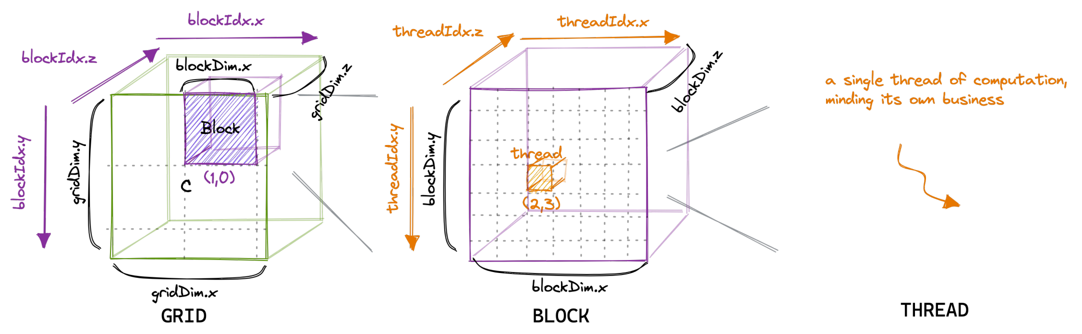


Figure 10. CUDA programming model organization can be visualized as a 3D grid that contains many identical blocks. Each block can itself consist of up to 1024 individual threads. [42]

When a kernel is launched, the programmer specifies the shape of the grid and the shape of each block. Each block contains multiple threads, and each thread executes the same kernel. All blocks must have the same shape [37]. This hierarchy can be summarized as follows:

- **Grid:** A 3D array of thread blocks. The grid defines the total scope of parallel work for a kernel launch.
- **Block:** A 3D array of threads, also called a Cooperative Thread Array (CTA). Threads within a block can cooperate via fast Shared memory (SMEM) and can synchronize with each other.
- **Thread:** The smallest unit of execution; a thread is essentially a few variables and a pointer to the current program instruction. Threads within a block are grouped into *warps* (typically 32 threads), which execute instructions in lockstep.

The GPU hardware is divided into many identical **Streaming multiprocessors (SMs)** [33]. Each SM is designed for executing parallel computational tasks and is roughly analogous to a multi-core CPU. Each SM typically executes multiple thread blocks in parallel, while each thread block is assigned to one and only one SM. Each SM contains two main types of storage: **Shared memory (SMEM)** and the **register file**. The SMEM is shared between threads in a single block, while the register file storage is private to each thread. Each SM also contains up to four warp schedulers. **Warp schedulers** issue instructions to warps. An SM cannot directly issue instructions to individual threads; instructions are instead issued to individual warps. Each warp can advance independently from other warps [37]. The organization of a single SM is shown in Figure 11.

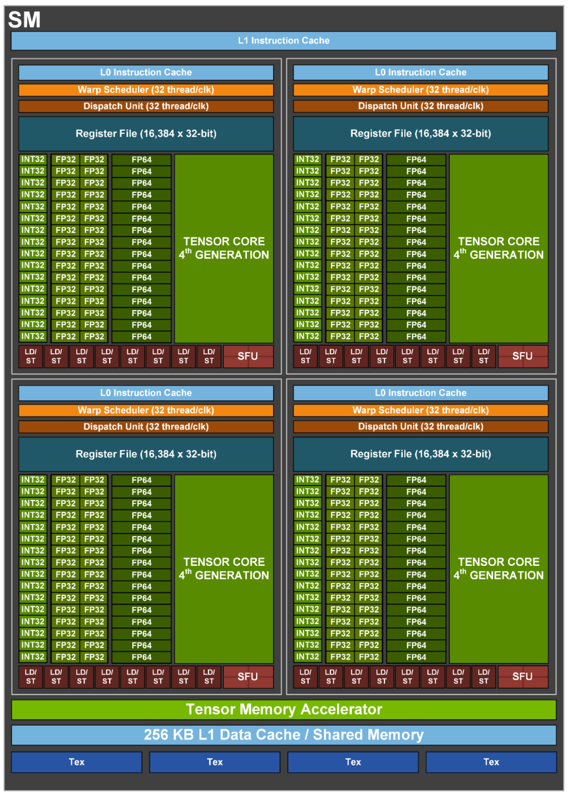

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

Figure 11. Diagram of a single H100 Streaming multiprocessor. [39]

A GPU can efficiently execute a very large number of threads in parallel. For example, NVIDIA's H100 SXM GPU has 132 SMs. Each SM has four warp schedulers, and each warp scheduler can process 32 threads per clock cycle, allowing an H100 to execute over 16,000 threads in parallel [39]. Note that, unlike CPU cores, all SMs execute the same kernel throughout execution. Additionally, it is not possible to assign a particular thread block to a specific SM, or to specify the order in which the warps inside a block advance. These decisions trade flexibility for better power efficiency and parallelism [33].

An important aspect of NVIDIA GPU architecture is the concept of **Compute Capability (CC)**, which is roughly analogous to a version number assigned to an SM and determines the features and resources available to the kernel [37]. For instance, an H100 consists of

SMs with CC of 9.0, while an RTX 4090 has CC of 8.9. CC additionally specifies which hardware-supported instructions are available to the programmer. For instance, CC determines the supported types of atomic operations. **Atomic operations** are operations that facilitate inter-thread cooperation. For example, adding two numbers from two threads to a shared memory location typically requires an atomic addition [37]. The GPU kernels developed in this thesis require a compute capability of at least 6.0 due to their reliance on double-precision atomic addition. Note that CC does not determine the type and size of global memory of the GPU, nor does it specify the number of SMs. These differences can lead to different performance characteristics for GPUs that have the same CC [37].

A GPU is more **power-efficient** than a CPU for suitable tasks. To illustrate the difference in parallelism and efficiency, consider this: An AMD EPYC 9965 CPU has 192 cores, each capable of running two threads, for a total of 384 concurrent threads at about 1.25 W per thread (500 W total) [43]. In contrast, an NVIDIA H100 SXM GPU draws up to 700 W, but can run over 16,000 threads in parallel, with each thread consuming only about 0.05 W [39]. This massive parallelism enables a GPU to outperform a CPU for suitable workloads, while using less overall power.

A GPU is designed to maximize **instruction throughput**, whereas a CPU is designed to minimize **memory latency**. To understand this distinction, consider the basic operation of adding two numbers and storing the result. The slowest part of this operation is typically not the computation itself, but the three memory accesses required: two `load`s and one `store`; once the data is loaded, the addition is usually performed very quickly. This effect is often called the **memory wall** [34]. Both the GPU and the CPU aim to overcome the memory wall in order to speed up operations, but they do so using different strategies.

A CPU relies on large, elaborate cache to minimize memory latency [33]. The CPU tries to retain as much of the required memory as possible to avoid stalling when accessing data. A CPU typically does not have enough registers to keep all assigned threads in registers. As a result, this leads to relatively slow thread switching, but excellent single-thread performance. Another downside of this approach is suboptimal use of registers on the CPU. A single CPU core typically cannot share its resources efficiently between many parallel threads. Finally, because the CPU cache occupies a large part of the available silicon die area, there is less room for implementing arithmetic units.

A GPU achieves high instruction throughput by maintaining a large number of concurrent warps and by rapidly switching between warps that are idle and warps that are ready for the next instruction [33]. This rapid switching is possible because an SM, unlike a

CPU, divides the register file among many warps. By holding many warps from multiple thread blocks in registers, an SM can quickly switch to a warp that is ready to execute the next instruction. In this way, a GPU "hides" the latency of memory accesses. Of course, this switching requires that a there are a sufficiently large number of warps per SM. Then, when a warp encounters a memory access that would cause it to stall, the SM can immediately switch to another warp that is ready for execution, keeping the hardware busy and minimizing idle time. An SM also contains many compute-related facilities, such as FP32 calculation units (also called **CUDA FP32 cores**) [37]. The downside of this architecture is that, on average, individual global memory access instructions are much slower than those of a CPU. The fact that many threads can *simultaneously* stall until a load or store is complete is referred to as "**latency hiding**". Given enough threads, latency can be fully hidden, leading to substantial parallel processing speed.

A key challenge in GPU programming is managing **register pressure**. Each SM has only a limited amount of fast SRAM storage, which must be divided among all active warps and blocks assigned to it [37]. Register pressure arises when a kernel uses too many registers per warp; as a result, the number of warps that can be active on an SM is reduced, limiting performance. **Occupancy** is defined as the ratio of active warps on an SM to the maximum number supported by the hardware. Increasing occupancy can lead to better performance. Balancing register usage, SMEM allocation, and thread count is critical for achieving high occupancy and performance [33]. Techniques to reduce register pressure include rewriting code to reduce the number of live variables, loop unrolling, and variable reuse where possible.

In summary, efficient kernel engineering on an NVIDIA GPU requires a deep understanding of the CUDA programming model, memory hierarchy, and hardware architecture. By carefully managing resources such as registers and shared memory, optimizing memory access patterns, and balancing thread and block configurations, developers can maximize parallelism and throughput. These principles are essential for achieving high performance in scientific computing applications, where both computational and memory-bound tasks must be tuned to fully exploit the capabilities of modern GPUs.

2.5 Computational mass spectrometry

The practical goal of Mass spectrometry (MS) research is to enable the accurate annotation of molecular structures based on mass spectral data [1]. This challenging task has led to the development of three major approaches – forward annotation, inverse annotation, and similarity search:

- **Forward Annotation:** Forward annotation involves predicting mass spectra from known molecular structures and experimental conditions [44, 45]. This approach is particularly useful for generating reference spectra, which can be used to build spectral libraries or validate experimental data. Forward methods often rely on computational quantum chemistry, machine learning models, or hybrid techniques that combine empirical rules with theoretical calculations and are thus mainly limited via computational efforts required.
- **Inverse Annotation:** Inverse annotation aims to infer molecular structures or elemental compositions from observed mass spectra [46, 47]. This problem is inherently more complex, as a single spectrum can correspond to multiple plausible structures. Methods in this category often employ combinatorial algorithms, deep learning, or probabilistic frameworks to narrow down potential candidates. Methods in this category are often limited by the availability of the training data.
- **Similarity Search:** Similarity search focuses on comparing pairs of mass spectra to estimate the structural similarity of their parent molecules [20, 48]. This approach plays a key role in database searching, clustering, and classification tasks, where the goal is to group or identify molecules based on shared spectral features. Similarity search methods are only bounded by the availability of reference spectra – which also plagues the inverse annotation methods.

The **forward annotation** is the only approach that is primarily limited by computational resources rather than data availability. Specifically, methods that employ computational quantum chemistry to simulate tandem mass spectra of small biological molecules (with molecular weights less than 1000 Da) can be improved with more computational investment. Forward annotation using quantum chemistry not only enables the generation of *in silico* mass spectra, but also provides valuable insights into precursor ion fragmentation patterns and molecular rearrangement reactions.

Grimme [49] created a FORTRAN package called QCEIMS for automatic forward annotation of small biological molecules. In the original version, QCEIMS supported only the electron ionization fragmentation mode. QCEIMS was later extended by Koopman and Grimme [8] to additionally support Collsion-induced dissociation (CID) with ESI. The updated package was renamed to QCxMS. Finally, the most recent work by Gorges and Grimme [50] introduced a different approach for modeling EI fragmentation called QCxMS2. The corresponding paper is, as of writing this, under review.

QCEIMS package uses molecular dynamics (MD) simulations to model molecules, with the forces calculated using approximate quantum mechanical methods [49]. By explicitly simulating the electron beam bombardment of a precursor ion and the subsequent dissociation of chemical bonds, QCEIMS obtains a set of fragment ions. By repeatedly simulating the fragmentation starting from a large number of different initial conditions, QCEIMS creates a mass spectrum *in silico*. A conceptual framework for how QCEIMS works is shown in Figure 12

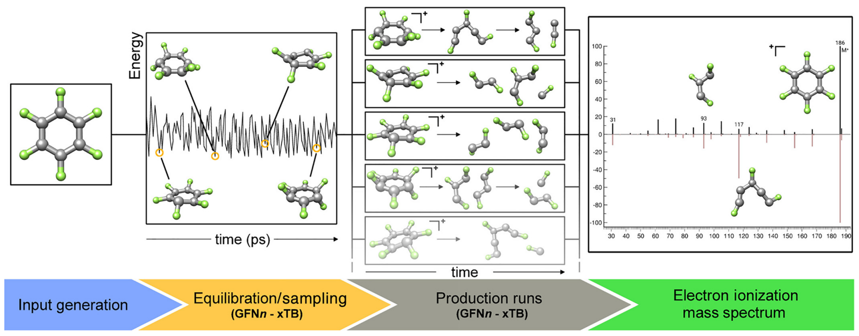

Energy
time (ps)
time
Input generation
Equilibration/sampling (GFNn - xTB)
Production runs (GFNn - xTB)
Electron ionization mass spectrum

Figure 12. Conceptual framework for QCEIMS, that shows the steps involved in obtaining computational EI spectra of a molecule. [49]

QCxMS package builds on this idea by adding the fragmentation mode [8]. Similar to EI mode, CID and ESI are also explicitly modeled. Specifically, after the initial conditions are set, a random number of gas atoms are accelerated into the precursor ion. If fragmentation happens, simulation stops and the fragments are recorded. A large number of simulations are performed and the resulting histogram of fragment m/z values are created. All simulations after the initial condition generation step are independent and can be done in a fully parallel fashion on a cluster of CPUs.

QCxMS2 is a recently-published FORTRAN package [50]. QCxMS2 takes a different approach to simulate fragmentation. Instead of explicit simulation of molecular dynamics and bond breakage, QCxMS2 immediately samples a set of possible end-results of a fragmentation of the precursor. For each fragmentation, QCxMS2 then calculates the minimum energy path from the precursor ion. The required energies for each fragmentation are then used for calculating the fragment intensities in the final spectrum.

QCEIMS, QCXMS, and QCxMS2 are compute-intensive programs. It has been shown that there is a strong positive correlation between the amount of computation (in CPU-hours) invested and the accuracy of the simulation [8]. Thus, the primary limitations of QCxMS are computational. These limitations can be summarized as follows:

- Inside a mass spectrometer, fragmentations occur on microsecond timescales, while the simulations can only feasibly simulate nanosecond timescales. This means that many components of generated mass spectra are completely missing, due to the computational infeasibility of generating them.
- The underlying quantum physical reality is provably hard to model on computers. Existing quantum models trade off accuracy and runtime requirements. Even for medium-sized molecules (30-50 atoms) the practically applicable quantum models accumulate a large amount of error throughout the simulation. Applying more accurate models is impractical without greatly extending CPU resources.

Even when using the fastest and the lowest-quality simulation methods, calculating just a single CID spectrum for a molecule with 30 atoms can take multiple CPU-days [8]. This steep computational cost makes it impractical to generate useful amounts of *in silico* mass spectra.

Several different studies have used QCEIMS and QCxMS to explore *in silico* generation of mass spectra. For instance, Wang *et al*. [9] evaluate the accuracy of QCEIMS by generating 451 mass spectra and comparing the results to the experimental National Institute of Standards and Technology (NIST) mass spectral dataset [3]. The authors used only the electron-ionization (EI) spectra for 43 different light (< 358 Da) molecules. The authors used dot product similarity [51] as a measure of accuracy between the predicted and experimental spectra. The study concluded that QCEIMS yielded moderately accurate mass spectra. Based on the results, 47% of generated spectra were in good agreement with the experimental data and 20% had excellent scores. Further, the authors investigated failure modes of QCEIMS. The authors found that QCEIMS tends to perform poorly for oxygen-containing organic compounds and for flexible molecules. Finally, the authors noted that the computation time exponentially increased with the number of atoms of the molecule. In one calculation instance, a molecule with 50 atoms was noted to take more than 14 hours on a system with 66 CPUs.

Yeo *et al*. [10] tested the accuracy of QCxMS by generating 2,708 synthetic CID mass spectra of small natural products at the GFN2-xTB [52] simulation level. The spectra were calculated across three different collision energies, with both positive and negative ionization modes. Overall, the authors found good agreement (mean dot-product score 0.68) between QCxMS-generated mass spectra and experimental spectra from Global Natural Product Social Molecular Networking (GNPS) [53]. They observed that higher-energy CID simulations tend to produce better-matching spectra in terms of dot-product scores. Additionally, they found that considering only the lowest-energy parent molecule does not significantly impact output quality while reducing computation time. The study also showed no significant difference in accuracy between [M+H] and [M-H] ionization modes, whereas other ionization modes performed significantly worse, with a mean dot-product score of 0.28.

The authors further reported that the presence of aromatic rings had a positive effect on accuracy. In line with the findings of Wang *et al.* [9], they also noted that the presence of oxygen and molecular flexibility were negatively correlated with the accuracy of *in silico* mass spectra. The authors do not comment on the total CPU-hours used, but they cite support from the A*STAR Computational Resource Center, which provided access to high-performance computing resources.

Hecht *et al.* [11] introduced an open-source workflow for large-scale prediction of EI spectra, by leveraging QCxMS software and using the extended tight-binding semi-empirical quantum method GFN1-xTB [54]. In total, authors tested 56 molecular classes comprising 356 total different molecules within the mass range of 108 to 715 Da and used an in-house reference mass spectral dataset for evaluation.

The workflow is divided into three steps. First, the 3D conformer generation creates energetically optimized 3D positions of atomic nuclei for a given chemical. This step requires simplified molecular input line entry system (SMILES) as input and uses GAMESS 6-31G [55] level of theory. This level of optimization can be considered to be very precise. Second, the fragmentation run is performed with QCxMS, with default parameters: 70 eV collision energy, excess energy per atom of 0.6 eV, initial temperature of 500 kelvin and a total simulation time of 10 ps. The simulation is conducted using GFN1-xTB method. This method is less precise but faster than the GAMESS 6-31G. Finally, post-processing of the results of the parallel execution of QCxMS fragmentation aggregates the data into a commonly used MS file format `.msp`. Surprisingly, authors chose to additionally filter the synthetically generated spectra at 1% level - by removing peaks that were less intense than 1% of the most intense peak. This is surprising given that synthetic spectra, by their very nature, should not be affected by contamination or measurement noise.

Authors reported the dot-product similarity measure between synthetic and experimental spectra. Authors found that the degree of molecular flexibility of the input is negatively correlated with accuracy of QCxMS. Similarly to other authors, Hecht *et al.* found that the molecules with carboxylic acid groups were the hardest to predict accurately with QCxMS [11]. Occurrence of sulfur and phosphorus atoms was also negatively correlated with accuracy. Authors used a total of 43,201 CPU-days, and a total of 2 TB storage for the analyses.

Aside from QCxMS, only a few other computational approaches for simulating mass spectra have been developed. Lee *et al.* [56] introduced CIDMD, a molecular dynamics library for simulating molecular collisions to predict CID-MS/MS spectra. The study investigated 12 metabolites with molecular weights under 205 Da to validate the accuracy of CIDMD. CIDMD is a collection of Python scripts that rely on `TeraChem` [57], a proprietary GPU-based quantum chemistry calculation package, for molecular dynamics simulations.

In total, the authors generated 261 *in silico* mass spectra using CIDMD. Many of the predicted spectra showed high similarity to their corresponding experimental spectra from the NIST library. The authors concluded that CIDMD predictions significantly outperformed those of CFM-ID [6], a widely used rule-based mass spectra prediction tool. However, CIDMD accuracy comes at the cost of computational efficiency. For example, predicting a single CID mass spectrum for a molecule with 31 atoms required approximately 123 hours on a GeForce GTX 980 Ti GPU. By utilizing a grid of 32 GPUs, this time was reduced to approximately 11.5 hours. The authors emphasized that no current tool effectively combines accuracy with speed. As a result, predicting high-quality MS/MS spectra for even a small portion of the chemical space remains a significant challenge.

Beyond computationally generating mass spectra, the applications of computational quantum chemistry are numerous. In this field, there are several established open-source tools that are relevant to the current study. Sun *et al.* [58] introduced PySCF, a Python-based electronic structure calculation library that supports simulation of molecules at multiple levels of theory. Rui *et al.* [12] subsequently introduced GPU4PySCF that extends PySCF with GPU acceleration. GPU4PySCF targets modern CUDA-enabled GPU hardware.

GPU4PySCF supports a wide range of *ab initio* methods, including HF, DFT. Although most functions have an interface in Python, the performance-critical modules are written in optimized C++ and CUDA kernels. Authors highlight the ability to efficiently explore the PES of molecules as one of the strengths of GPU4PySCF. Notably, efficient exploration of PES is a key challenge for accelerating QCxMS [8]. There has been no prior work applying GPU4PySCF to accelerate *in silico* generation of mass spectra.

It is worth noting that GPU4PySCF does not explicitly support semi-empirical methods such as GFN2-xTB. Semi-empirical methods are usually significantly faster than *ab initio* approaches like DFT, however they also are less accurate. Exploring the integration of semi-empirical methods with GPU acceleration could represent a valuable future research direction.

2.6 Summary

The reviewed works highlight significant progress and challenges in predicting and simulating mass spectra using computational approaches. Quantum chemistry-based methods like QCxMS provide robust mass spectra predictions, but their high computational cost makes them impractical for high-throughput applications. Other approaches have attempted to improve efficiency or accuracy but still struggle to combine both at scale. Benchmarking studies, like those by Yeo *et al.* [10], demonstrate the value of leveraging quantum chemistry tools while underlining the limitations of current computational methods.

This study focuses on accelerating QCxMS simulations by integrating GPU programming to enable faster *in silico* generation of mass spectra, without sacrificing accuracy. By combining the computational power of a GPU with QCxMS, this work aims to bridge the gap between accuracy and speed in computational MS.

3 PROPOSED METHODS

3.1 Profiling and bottleneck identification

Recall that QCxMS enables *in silico* simulation of mass spectra, but this process is computationally expensive [8]. Additionally, QCxMS depends on several other computational chemistry tools to function, most importantly TBlite. Accelerating this non-trivial software required a multi-step approach.

The first step in accelerating QCxMS was systematic code-base profiling and identification of computational bottlenecks. QCxMS and its dependencies are written in FORTRAN, comprising 44,217 lines of code and 3,099 functions, with extensive use of multidimensional arrays and external libraries such as Linear Algebra Package (LAPACK) and Basic Linear Algebra Subprograms (BLAS). Profiling was performed using `gcov`, `lcov`, and `gprof`, which provided initial insights into both line coverage and runtime performance.

Initial profiling focused on the Electron ionization (EI) mode, which involves 3,338 lines of code and 272 functions. The analysis revealed that, although many routines are theoretically parallelizable (for example, simulating different molecular trajectories in parallel), practical constraints such as GPU memory limitations make it infeasible to accelerate the entire workflow in parallel on the GPU. As a result, the focus shifted to identifying individual functions with the highest computational cost and parallelization potential. Figure 13 shows how much time is spent in different parts of the single-point evaluation for a sample molecule of aspirin that consists of 21 atoms.

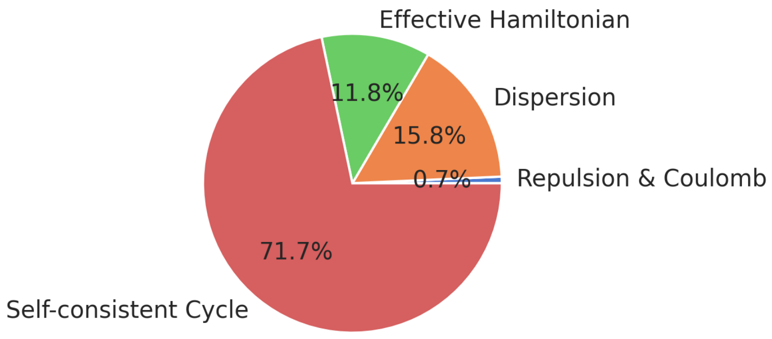


Figure 13. Time spent in parts of TBlite for single-point evaluation of aspirin.

As shown in Figure 13, the Self-consistent cycle (SCC) procedure is the most time-consuming part of the single-point evaluation. As discussed in Subsection 2.3, SCC is inherently iterative, requiring repeated diagonalization of the Fock matrix until convergence is achieved. The number of iterations needed cannot be determined in advance, and in some cases, such as for distorted geometries (e.g., transition structures) and metal complexes, convergence may not be reached at all [13]. To take advantage of GPU acceleration, the algorithm must be reformulated for parallel computation. However, the extent of the necessary modifications and the complexity of implementing a generalized eigenvalue solver for the Roothaan–Hall equations are well beyond the scope of this study. Therefore, despite its significant impact on performance, SCC is not the focus of GPU acceleration in this work.

Recall that dispersion refers to long-range attractive interactions between atoms or molecules. Dispersion interactions are important for accurately describing the stability and structure of molecular systems, especially in non-covalently bound complexes, biomolecules, and materials. For isolated small molecules, dispersion corrections are typically negligible because there are few non-bonded atom pairs at distances where dispersive forces are significant. For this reason, dispersion is also not the focus of GPU acceleration in this work.

The effective Hamiltonian calculation subroutine computes overlap integrals between atomic orbitals and is a prerequisite for SCC calculation. Specifically, it computes:

- **The overlap matrix**: A real, symmetric matrix containing the overlap integrals between pairs of atomic orbitals. These values quantify how much two orbitals spatially overlap and are fundamental for constructing the electronic structure.
- **The dipole moment integrals**: A real tensor that describes how the electron distribution responds to electric fields.
- **The quadrupole moment integrals**: A real tensor representing the quadrupole moment contributions. These are important for describing higher-order charge distributions and interactions with external fields.
- **The effective Hamiltonian matrix**: A real, symmetric matrix that encodes the total electronic energy contributions, including kinetic energy, nuclear attraction, and inter-electronic interactions. The effective Hamiltonian is central to solving the electronic structure problem.

The effective Hamiltonian subroutine loops over all pairs of atoms and their basis functions (including periodic images if present), computes the required integrals for each pair, and fills the matrices. The effective Hamiltonian elements are built using atomic self-energies, distance-dependent scaling, and polynomial enhancements. The effective Hamiltonian calculation is the primary target for GPU acceleration in this thesis and is the central focus of this work.

3.2 Linking FORTRAN and CUDA C++

Recall that QCxMS and all its dependencies are written in FORTRAN [8], a language widely used for scientific computation. In contrast, NVIDIA GPU applications are typically developed in CUDA C++, a C-like language. Modifying FORTRAN code to leverage GPU acceleration is not straightforward. One approach is to switch to the CUDA FORTRAN compiler to utilize the GPU from FORTRAN with minimal code modifications. Unfortunately, many FORTRAN language constructs in the codebase are unsupported by CUDA FORTRAN, making this method nonviable. Another approach is to rewrite the entire MS simulation in CUDA C++; however, the amount and complexity of the source code make this approach impractical. A more feasible solution is to identify the most computationally intensive part of the simulation, implement it in CUDA C++, and then link the GPU-accelerated component to the existing FORTRAN code. This can be achieved by compiling the FORTRAN and CUDA C++ code separately and linking the FORTRAN binary executable to the CUDA C++ binary. This approach is illustrated in Figure 14.

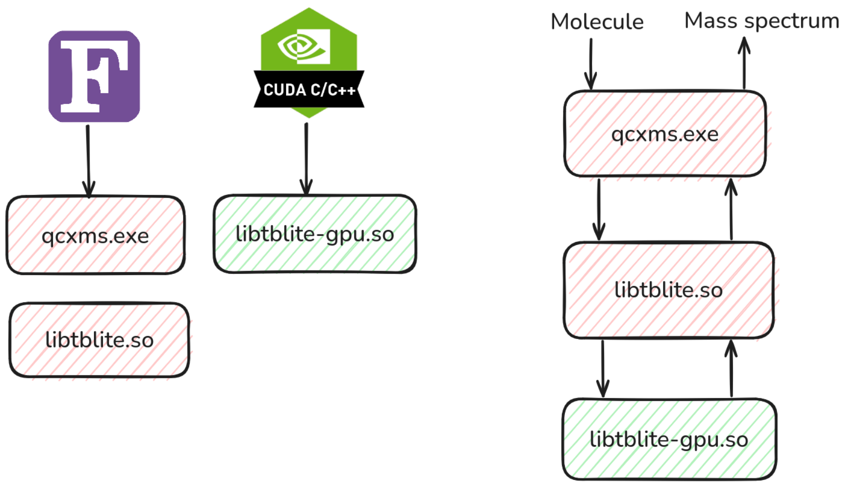


Figure 14. FORTRAN and CUDA C++ sources are compiled separately (left). The full application (right) consists of a QCxMS executable file that is linked to the TBlite library. TBlite library is itself linked to a CUDA binary that allows using a GPU.

Integrating CUDA C++ kernels into the FORTRAN-based QCxMS code-base requires an interface between the two languages. Since FORTRAN 2003, the `iso_c_binding` module provides a standardized way to interoperate between FORTRAN and C/C++ [59]. Fortunately, this module allows connecting FORTRAN to CUDA C++ just as well. This allows correctly passing data such as arrays and scalars between FORTRAN and CUDA C++ using C-compatible pointers. Arrays are passed as pointers to avoid copying data. For example, a FORTRAN array `real(c_double) :: hamiltonian(:,:)` can be passed to CUDA C++ as `double *hamiltonian` pointer, with the array dimensions communicated in separate scalar values.

Complex types and structures in FORTRAN, especially those containing `allocatable` FORTRAN arrays, are not easily transferable to CUDA C++ [59]. This limitation is addressed by flattening all types and passing each underlying array as a separate argument. As a result, the CUDA C++ link function has 94 separate parameters. To facilitate this translation, on the CUDA C++ side these parameters are re-packaged into FORTRAN-like types from which they originated. For example, N-dimensional arrays are wrapped in `tensorNd_t` types, which provide N-dimensional indexing, built-in array boundary checking, and memory management. These types greatly simplify development and translation efforts, at the cost of small performance overhead.

Using `iso_c_binding`, a bridge was implemented that allows FORTRAN code to invoke the CUDA-accelerated effective Hamiltonian kernel. This bridge was integrated into the existing TBlite test suite, enabling benchmarking of the CUDA kernel from the existing TBlite FORTRAN tools, such as molecule loading and basis creation. The FORTRAN code was compiled with GNU FORTRAN (`gfortran`) version 13.3.0, while the CUDA C++ code was compiled using NVIDIA's `nvcc` compiler version 12.4.

3.3 FORTRAN to CUDA C++ translation

The program for calculating the integrals for the effective Hamiltonian matrix is a non-trivial computational algorithm. It must correctly read and write multidimensional arrays with up to four dimensions and implement an iteration that is nested eight for-loops deep. Complicating the implementation is the fact that CUDA C++ does not inherently support multidimensional dynamic array indexing; all arrays are represented as flat 1D pointers to the first element. It is up to the CUDA C++ programmer to develop efficient bounds-checking and memory access methods [37]. For instance, a 3x4 array in C++ would simply be `double *arr`, placing the burden of correctly implementing, for example, `arr[2][1]` access on the programmer. This differs from FORTRAN, where multidimensional array operations and bounds checking are intrinsic and supported by the language.

Translation is further complicated by the fact that the indexing and memory layout of arrays differ significantly between FORTRAN and C++. FORTRAN uses 1-based indexing and stores arrays in column-major order, while C++ uses 0-based indexing and stores arrays in row-major order. To bridge this gap, all multidimensional arrays are passed from FORTRAN to C++ as a flat pointer, along with their dimensions listed in reverse order. For example, a FORTRAN array declared as `real(c_double) :: arr(3,4)` (3 rows, 4 columns) is passed to C++ as `arr, 4, 3`, where 4 is the size of the last dimension and 3 is the size of the second-to-last. On the C++ side, this is interpreted as `double arr[4][3]`, with the first index corresponding to the FORTRAN column and the second to the row.

When translating array accesses, a FORTRAN operation such as `arr(j, i)` (with indices starting from 1) becomes `arr[i-1][j-1]` in C++ (with indices starting from 0). This convention is applied consistently for all arrays, ensuring that the data layout and indexing semantics are preserved across the language boundary. By always passing array dimensions in reverse order and adjusting indices, the translation of FORTRAN array operations to C++ becomes more tractable.

To simplify the translation of complex FORTRAN array operations to C++, a powerful C++ feature, operator overloading, is used. An operator for multidimensional array access was developed with built-in bounds checking and support for up to 4D arrays. For instance, this operator allows writing expressions like `arr(i,j) += 1.0` with the usual meaning of incrementing element a_{ij} by 1, where A is a 4D tensor and i and j are 0-based, meaning the first element is a_{00}.

Even with these expressions, several routines proved too difficult to translate directly. To enable a one-to-one correspondence between FORTRAN and C++ code, an alternative indexing operator was developed that allowed copying a mostly-unchanged FORTRAN code to CUDA C++. This operator can be selected by supplying `arr(i,j,'f') += 1.0`. In this case, i and j are interpreted as 1-based indices, allowing more straightforward translation of FORTRAN to C++ with no impact on performance. These are used extensively for the complicated p-orbital Spherical-to-Cartesian transformation routines.

Template programming in C++ is a language feature that allows declaration of a parameterized family of functions or classes. A template in C++ is comparable to a blueprint for creating a generic class or function. For instance, `template <typename T> void foo(T item);` declares a function `foo` that works with any type of item, whether it is a `float item`, a `double item`, or any other type. Template programming was used to parameterize the maximum angular momentum of the basis. The maximum angular momentum of the basis dictates the required memory size for calculating overlap integrals (this is also called integral scratchpad). Since the integral scratchpad resides in the high-speed but limited SMEM of the GPU, knowing the required memory size ahead of time allows the CUDA runtime to optimize kernel launch configuration for a given GPU hardware. By encoding the maximum angular momentum as a template parameter, the compiler can generate specialized kernel variants for each supported value of the maximum angular momentum, ensuring that only the required amount of SMEM is allocated and that memory accesses are as efficient as possible. This approach not only minimizes wasted SMEM but also enables the CUDA runtime to maximize occupancy and throughput, as the exact memory requirements of each kernel instance are known in advance. As a result, the kernel can be tuned for better performance and can support a range of basis sets without sacrificing efficiency.

3.4 Kernel design

The effective Hamiltonian kernel calculates the atomic orbital interactions in parallel using the CUDA programming model. As mentioned previously, in CUDA programming model consists of a program (kernel) that is executed in parallel over many blocks of threads. Figure 15 shows a summary of the chosen parallelization scheme.

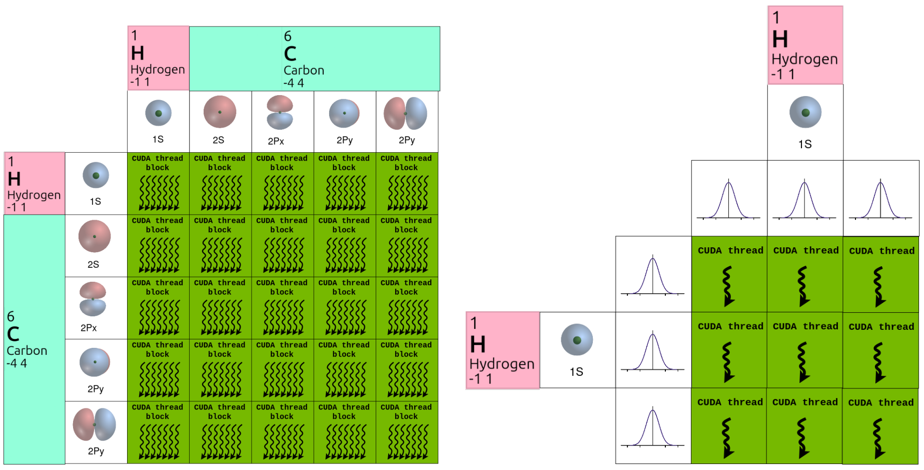


Figure 15. GPU effective Hamiltonian parallelization scheme, showing the grid of blocks (left) and one block of threads (right) used for the effective Hamiltonian calculation for a hypothetical system of hydrogen and carbon.

In implementation, the CUDA grid is three-dimensional. The X, Y, and Z axes of the grid correspond to the number of atoms, the maximum number of neighbors per atom, and number of atomic orbitals respectively. The neighbors of an atom are defined as all atoms within a certain distance. This includes valence atoms and possibly other nearby atoms. For example, when simulating an aspirin molecule, the CUDA grid dimensions are (21, 20, 2): 21 atoms in aspirin, a maximum of 20 neighbors for any atom (indicating that at least one atom is close enough to interact with all others), and 2 for the Z axis, which represents the number of atomic orbitals for carbon.

Each thread block contains 32 threads, matching the CUDA warp size for optimal performance. Within each block, threads cooperate to load basis function coefficients into shared memory, perform overlap and integral calculations, and write results back to global memory.

The GPU kernel for effective Hamiltonian calculation consists of two distinct CUDA kernels that are launched in succession:

- The first kernel calculates atomic orbital interactions between different atoms (inter-atomic).
- The second kernel calculates atomic orbital interactions within the same atom (intra-atomic).

The computational effort for the first kernel scales as $O(N^2)$, where N is the total number of atoms in the molecule, while the second kernel scales as $O(N)$. Consequently the first kernel much more compute-intensive than the second one for most molecular systems.

The parallelization scheme is similar between the two kernels. Recall that in the CUDA programming environment, there are two levels of parallelism: the grid of blocks (each block is independent) and the block of threads (threads within a block can cooperate). For the effective Hamiltonian calculation, the CUDA runtime is used to launch enough blocks to cover all possible interactions between all orbitals of each atom. Each block is assigned a unique `blockIdx` (with x, y, z indices) to determine the atom-pair and orbital-pair for the given block. Each block computes at most one orbital-to-orbital interaction, and each interaction is assigned to exactly one block. Typically, more blocks are launched than strictly necessary; any extra blocks exit quickly if they find no work to perform. This over-scheduling is an engineering compromise. In CUDA, the number of blocks must be specified before launching the kernel. However, it is not feasible to efficiently calculate the exact number of required thread blocks before kernel execution, because the exact number depends on the adjacency of atoms, which cannot be calculated before the kernel launch.

Recall that each thread in CUDA has access to private variables and can progress in parallel with other threads in the same block. Crucially, all threads in a block have access to SMEM, a limited, high-speed memory space shared among the threads. Accessing shared memory is several orders of magnitude faster than accessing global GPU memory. Threads in each block use this memory extensively to perform all instructions needed for calculating orbital-to-orbital interactions. Threads are distinguished by the intrinsic CUDA C++ variable `threadIdx.x`, which allows each thread to identify itself within the block.

Each thread in the 32-thread block performs the following steps in parallel:

1. Locate the first atom and its orbital for calculation.
2. Locate the second atom and its orbital for calculation.
3. Calculate the distance between the nuclei of the two atoms.
4. Load basis data for the first and second orbitals from global memory to shared memory.
5. Allocate shared memory scratchpad space for integral calculation.
6. Calculate overlap, dipole, and quadrupole integrals between one pair of CGTOs.
7. Move the results from shared memory back to global memory.

The calculation of overlap, dipole, and quadrupole integrals is itself a multi-step process and a major target of optimization. All steps are executed by the 32 threads in parallel. A single thread performs these steps:

1. Initialize Spherical integral accumulation scratchpads in shared memory.
2. Select one pair of the primitive Gaussians making up the orbital.
3. Calculate the overlap integral between the two primitive Gaussians.
4. Atomically sum the overlap integrals into shared memory scratchpads.

5. Transform one row of the overlap integral from Spherical to Cartesian coordinates.
6. Remove one diagonal element from quadrupole integrals.

As a result, the cumulative effort by all threads leads to efficient calculation of the effective Hamiltonian, overlap, dipole, and quadrupole integrals between the two given atomic orbitals. As mentioned above, the transformation between Spherical and Cartesian coordinates is currently implemented only for s and p type atomic orbitals. d and higher orbitals are omitted, but their implementation is straightforward and would require efficient general matrix-matrix and matrix-vector multiplication (`gemm` and `gemv`) as well as transpose operations. It is noted that these `gemm` operations could significantly benefit from the utilization of tensor cores on GPUs that support them.

In addition to replicating the effective Hamiltonian logic for GPU parallelism, the potential of batched effective Hamiltonian calculations was evaluated. Batched effective Hamiltonian refers to calculating the effective Hamiltonian for N different molecules (or molecular orientations) in one shot. This mode is motivated by the fact that QCxMS relies on statistically converging many molecular fragments to their corresponding mass spectra. This requires performing a large number of single-point evaluations on the same structure, each with slightly different initial conditions or orientations. It is possible to substantially improve the singlepoint calculation throughput by executing these calculations in parallel, inside a single GPU kernel. This batching approach offers little benefit on CPUs, where each core can only process one item at a time, but GPUs excel at processing large parallelized calculations. In the current implementation, the batched calculation allows each item in the batch to have different XYZ atomic coordinates (and thus variable self-energies), but all other parameters remain the same. For example, for a molecule like glutamine (29 atoms, 71 orbitals), a batched calculation with batch size B would use a $(B, 29, 3)$ tensor for XYZ coordinates and return a $(B, 71, 71)$ tensor for overlap integrals. The molecular connectivity and basis information are shared across the batch. Performance is estimated by extending the CUDA grid's first dimension to be `batch_size` × `number_of_atoms`. While this batched mode is still experimental and not yet ready for general use, it demonstrates the scalability of the kernel and the advantages of GPU-based batching for high-throughput scenarios.

The second kernel (intra-atomic orbital interactions) follows the same steps and grid-block-thread layout as the first, but only computes interactions between each atom's orbitals, pairwise. As mentioned previously, this kernel is typically an order of magnitude faster than the first.

Atomic operations are slow and have been used sparingly. Only one segment of code requires atomic operations: the accumulation of specific pairs of overlap integrals inside shared memory in the `multipole_cgto` subroutine. The global write to the final outputs is fully parallel, without any use of atomics, since each thread writes to a unique location in global memory.

Almost all calculations are performed in double precision to ensure numerical accuracy. The innermost "hot path" of the `multipole_3d` calculation is performed in single precision (`float32`), as this was empirically found to improve performance without impacting the final results.

3.5 Development tools and workflow

The project was developed with reproducibility and modularity in mind. The build system is **Meson** [60], which allows combining QCxMS, its dependency TBlite, and the GPU kernel (`tblite-gpu`) into a single executable. Meson's subproject feature allows each component to be developed and tested independently.

Visual Studio Code was used as the primary integrated development environment.

Python notebooks were used for:

- Generating figures using `matplotlib` [61].
- Automating benchmarks and collecting performance data.
- Procedurally generating test molecules.
- Post-processing and validating output data.

Version control was managed with **Git**. All code, scripts, input files, and performance logs are publicly available on GitHub:

- GPU kernel development repository [62].
- QCxMS performance profiles [63].
- Modifications to TBlite and figures used in this work [64].

Detailed commit messages and tags allow easy navigation of the development timeline.

Containerization: To ensure reproducibility of the code-base across different machines and environments, `Dockerfile` is provided for building and running the software. Docker enables encapsulation of all dependencies, compilers, and libraries, making it straightforward to deploy the application on any compatible system. Since GPU acceleration is essential for the workflow, it relies on NVIDIA's Docker runtime (`nvidia-docker`), which allows a Docker container to access the host's GPU resources. This is particularly useful for benchmarking on large cloud-based GPUs, as it guarantees a consistent environment regardless of the underlying hardware or operating system.

Profiling: Performance analysis and kernel optimization were guided by NVIDIA Nsight Compute (NCU), a comprehensive profiling tool for CUDA applications. Nsight Compute provides both Command-line interface (CLI) and Graphical user interface (GUI) (`ncu-ui`) for collecting and visualizing detailed performance metrics [65]. The CLI is used to profile kernel execution, capturing aggregate statistics such as compute throughput, kernel duration, and memory bandwidth, as well as fine-grained measurements like source-line counters and instruction-level analysis. The resulting reports can be explored in the GUI, which also features an integrated expert system that offers targeted optimization recommendations based on the collected data. These insights are useful for identifying performance bottlenecks in the kernel code and guiding optimization efforts.

4 RESULTS

4.1 Validation and performance suite

A simple validation and benchmarking suite was created to check both the correctness and speed of the GPU effective Hamiltonian kernel. Initial tests ensured that arrays and numbers were passed correctly between FORTRAN and CUDA C++, helping catch any programming errors with array shapes, indexing, or memory layout. Following this, the TBlite test suite was extended to check the accuracy of the GPU kernel. The extension directly compared GPU kernel results the existing CPU subroutine results from TBlite. A set of representative molecules was selected for comparison, including H_2, LiH, S_2, and SiH_4.

To test GPU kernel, the kernel was benchmarked on a range of increasingly large, complex biomolecules. These biomolecules were: glutamine (29 atoms), cocaine (42 atoms), a partial helix of a Deoxyribonucleic acid (DNA) (387 atoms), hen egg white lysozyme (1102 atoms) [66], and Tequatrovirus T4 lysozyme (1404 atoms) [67]. Structures of glutamine and cocaine were accessed via from PubChem [68]. Lysozyme and DNA structures were accessed via Protein Data Bank (PDB). Additionally, a set of straight-chain alkanes from 32 up to 4096 carbon atoms were procedurally generated and using a Python script. These large biomolecules were used both for testing both the performance and the accuracy of the GPU kernel.

The TBlite test suite was extended to measure both the performance and accuracy of the effective Hamiltonian kernel. Specifically, the extended tests followed this procedure for each molecule:

1. The effective Hamiltonian, overlap, dipole, and quadrupole integrals were computed independently using both the CPU (reference TBlite implementation) and the CUDA C++ GPU kernel.

2. The resulting matrices and tensors were compared element-wise. Any absolute or relative difference exceeding double-precision machine epsilon ($\sim 10^{-15}$) was flagged.

3. Automated scripts summarized discrepancies, if any, and generated detailed reports for further investigation.

4. This process was repeated after every kernel modification, ensuring continuous validation during development and optimization.

For each molecule, the GPU and CPU results were compared element-wise to ensure they matched within floating-point epsilon. TBlite, by default, uses IEEE double-precision floating-point values, with a machine epsilon on the order of 10^{-15}. Notably, the molecules tested contain only elements up to Argon (Z=18); elements beyond Argon require d-orbital Spherical transformations, which are currently unsupported in the GPU kernel.

Several different CPU and GPU hardware platforms have been tested. Ground truth CPU results were produced on a 32-core Intel i9-14900HX [69] system with 32GB memory. GPU benchmarks used a single GPU, including an NVIDIA RTX 4060 Laptop GPU for development, and for larger benchmarks, Tesla V100 SXM 32GB, RTX 4090, and H100 SXM 80GB GPUs. These GPUs were accessed via an online GPU marketplace [70]. Unless otherwise specified, the 32-core CPU is used as a comparison baseline instead of a single core.

4.2 Implementation validation

Across all tested molecules and hardware platforms, the GPU kernel outputs were numerically indistinguishable from the CPU implementation within machine epsilon. No discrepancies, edge cases, or anomalous behaviors were observed, even for the largest bio-molecular systems tested. In particular:

- All matrix and tensor elements matched to at least 14 significant digits.
- No systematic or random deviations were detected, confirming the absence of hardware-specific or race condition-induced errors.
- The validation suite also confirmed correct handling of memory layout, indexing, and data transfer between FORTRAN and CUDA C++.

Notably, the automated validation tests facilitated rapid kernel development and optimization. By running the validation test suite after each code modification, all regressions and software bugs were detected early. This enabled more aggressive performance tuning while maintaining full validation accuracy. The suite also makes it easy to reproduce the results of this research and can be extended to benchmark new molecules.

4.3 Performance evaluation

Performance of the kernel on the GPU has been exhaustively benchmarked. All molecules in the test suite were tested against all previously listed GPU hardware. Initially, the parallel performance of CPUs was examined to create a baseline. Figure 16 shows how the CPU-only system scales with each added CPU core. The left plot shows the total speedup achieved when using N parallel CPUs. The right plot shows the parallel efficiency, which measures the marginal contribution of the Nth CPU. Parallel CPU scaling performance for the effective Hamiltonian demonstrates that even with 32 CPUs, speedup plateaus at approximately five times that of single-threaded performance. Adding extra CPUs is therefore an inefficient way to gain additional speedup.

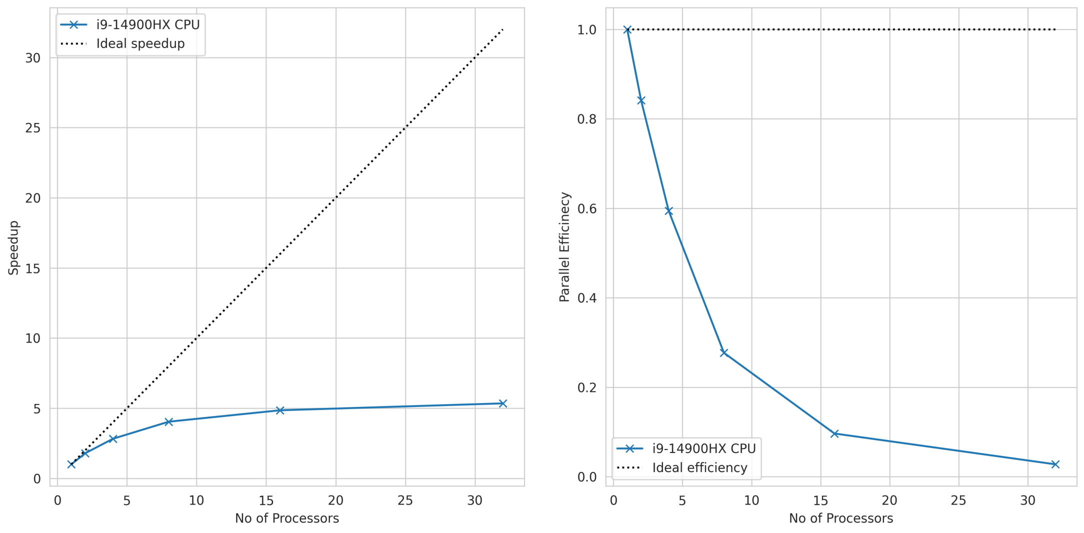


Figure 16. Parallel speedup (left) and efficiency (right) of multi-CPU system for effective Hamiltonian calculation.

Compared to the 32-CPU baseline, the GPU implementation showed substantially better scaling, with the state-of-the-art H100 GPU achieving over a 28-fold speedup compared to 32 CPUs for the 1404-atom lysozyme system, as shown in Figure 17. For highly regular systems such as linear alkanes, GPU speedup exceeded 250-fold. Notably, the performance of the older V100 and the newer RTX4090 GPUs was similar, which could be attributed to their comparable memory bandwidths (898 GB/s and 1010 GB/s for the V100 and RTX4090, respectively [71, 72]). This suggests the kernel is heavily memory-bound. Still, since GPUs typically provide orders of magnitude higher memory bandwidth than contemporary CPUs, GPUs show a large speedup.

The observed GPU speedup, as detailed in Table 1, is most pronounced for systems that

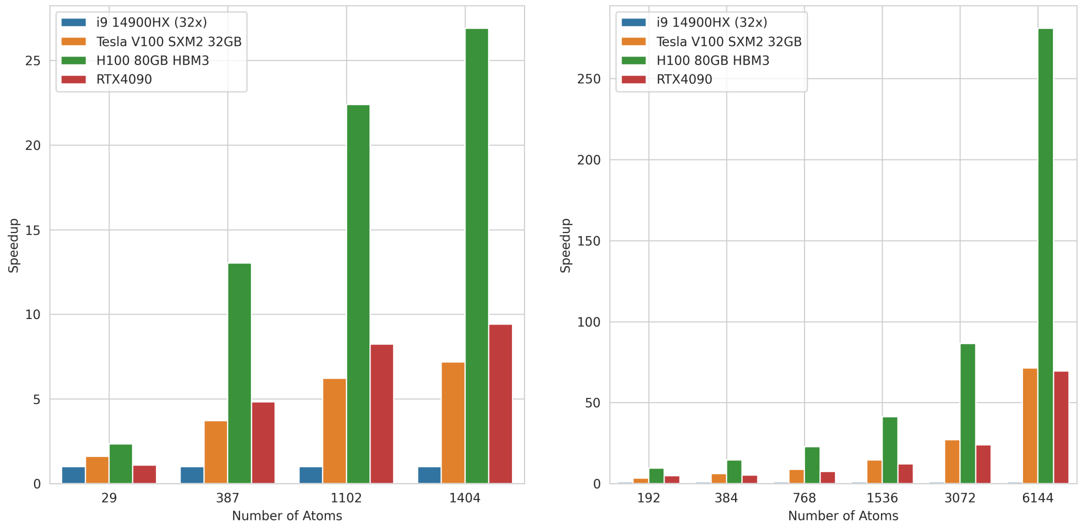


Figure 17. GPU speedup for effective Hamiltonian calculation for proteins (left) and alkanes (right) compared to a 32-core Intel i9-14900HX CPU.

Table 1. Molecules and different execution times on evaluated systems, time shown in seconds. Note that `i9 14900HX` refers to a full 32-core CPU system.

Structure	Atoms	i9 14900HX	Tesla V100	RTX4090	H100
Glutamine	29	0.01293	0.00800	0.01192	0.00553
64 alkane	192	0.00754	0.00222	0.00156	0.00079
128 alkane	384	0.01287	0.00209	0.00243	0.00088
DNA fragment	387	0.05703	0.01535	0.01182	0.00438
256 alkane	768	0.02858	0.00325	0.00380	0.00125
1LYZ Lysozyme	1102	0.23149	0.03726	0.02809	0.01034
103L Lysozyme	1404	0.32456	0.04523	0.03448	0.01207
512 alkane	1536	0.08313	0.00569	0.00679	0.00201
1024 alkane	3072	0.28517	0.01052	0.01193	0.00330
2048 alkane	6144	1.45793	0.02042	0.02098	0.00519

are repetitive. This is due to two main factors. First, GPUs process interactions in batches using high-speed shared memory, requiring only an initial single slow transfer from global to shared memory before performing computations rapidly. In contrast, CPUs, despite having longer cache lines, lack an equivalent shared memory architecture and cannot easily process data in similar chunks, resulting in lower cache efficiency and slower performance. This architectural difference explains the substantial performance gap observed between GPUs and multi-core CPU systems.

Batched calculation of many orientations of the same molecular structure is useful in QCxMS, as the software relies on statistically converging many molecular fragments to their corresponding mass spectra. This requires performing a large number of single-point evaluations on the same structure, each with slightly different initial conditions or orientations. To assess the performance of batched calculations, the GPU kernel was extended so that the first dimension of the CUDA grid corresponds to the batch size multiplied by the number of atoms. In this configuration, only the XYZ coordinates and self-energy of the atoms may differ between batch items, while all other parameters remain constant. Although a more flexible batching scheme—allowing for different atom counts or types in each batch—would be desirable, it is beyond the scope of this study.

Figure 18 shows the performance of batched effective Hamiltonian calculations using a cocaine molecule as a test case. The results show that an H100 GPU achieves an order of magnitude speedup over a 32-core i9 CPU system, even as the batch size increases. This demonstrates the scalability and efficiency of the GPU kernel for high-throughput scenarios, such as generating large synthetic spectral libraries or performing ensemble simulations. Efficient batched computation is particularly important for applications that require statistical sampling or uncertainty quantification, as it enables rapid evaluation of many molecular configurations in parallel.

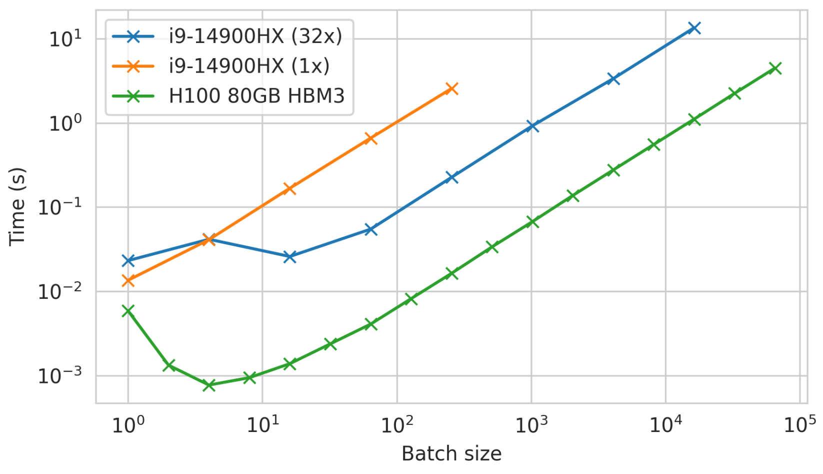


Figure 18. Batched effective Hamiltonian calculation performance comparison between a single CPU, 32 parallel CPUs, and an H100 GPU.

CPU-to-GPU data transfer is a significant bottleneck. The transfer typically takes longer than the CPU-only calculation, as shown in Figure 19. This problem highlights the need for an end-to-end GPU workflow that would eliminate the need for such transfers.

The cause of this bottleneck can be inferred from Table 2, which shows that the size of the system in gigabytes and the time it takes to complete the transfer scale approximately as $O(N^2)$, where N is the number of atoms:

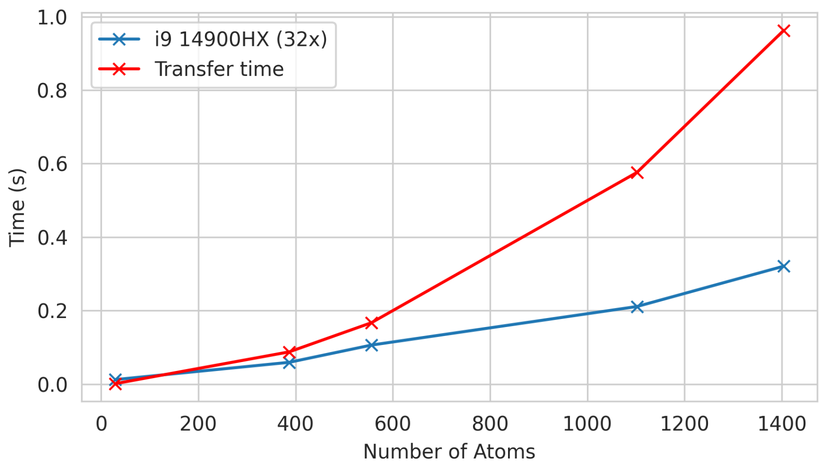


Figure 19. Comparison of CPU-to-GPU transfer time (red) and 32x CPU calculation time (blue) as a function of atom count.

Table 2. CPU-to-GPU transfer times, sizes of transferred systems (in GB), and CPU processing times.

Structure	System size (GB)	CPU processing time (s)	Transfer time (s)
Glutamine	0.00090	0.01293	0.00122
64 alkane	0.02600	0.00754	0.00948
128 alkane	0.10391	0.01287	0.03803
256 alkane	0.41544	0.02858	0.15521
DNA fragment	0.46387	0.05703	0.18663
512 alkane	1.66135	0.08313	0.58872
1LYZ Lysozyme	3.49879	0.23149	1.24432
103L Lysozyme	5.63159	0.32456	2.00886
1024 alkane	6.64458	0.28517	2.39134
2048 alkane	26.57672	1.45793	9.39111

The cause of this lies in the memory-bound nature of the effective Hamiltonian calculation. For each molecular system, a substantial amount of data, scaling as $O(N^2)$, must be transferred from CPU DRAM to GPU memory. GPUs offer memory bandwidth several orders of magnitude higher than CPUs, allowing them to outpace CPUs. However, the transfer bandwidth between a CPU and GPU on the systems tested was slower than the bandwidth of either CPU or GPU. As a result, the time required to transfer the system to the GPU can exceeded the time needed for the CPU to perform the calculation on its own. This overhead presents a major obstacle to realizing the full performance benefits of GPU acceleration for simulation of mass spectra.

4.4 Analysis of computational limits

In the previous subsection, the kernel's performance was evaluated externally by measuring execution time for each step. In this section, the kernel's internal computational limits are presented using analysis via NVIDIA Nsight Compute (NCU) tool. While most benchmarks were performed on state-of-the-art hardware (e.g., the H100 GPU), the analyses in this subsection use a laptop-grade RTX 4060 GPU (compute capability 8.9). This is because in-depth profiling on an H100 requires privileged access to the GPU hardware, which is restricted on cloud platforms due to security concerns [73, 74]. Consequently, detailed profiling results are provided using the local laptop-grade GPU.

The kernel's computational limits were assessed in terms of warp occupancy, register and shared memory usage, and arithmetic intensity. The kernel achieved a warp occupancy of 16%, limited by both register usage (120 registers per thread) and shared memory allocation (5.3 kB per block). Reducing register usage could increase occupancy up to 32%. Figure 20 shows that register usage is the main limiter and that increasing shared memory up to 12 kB per block is unlikely to further reduce occupancy.

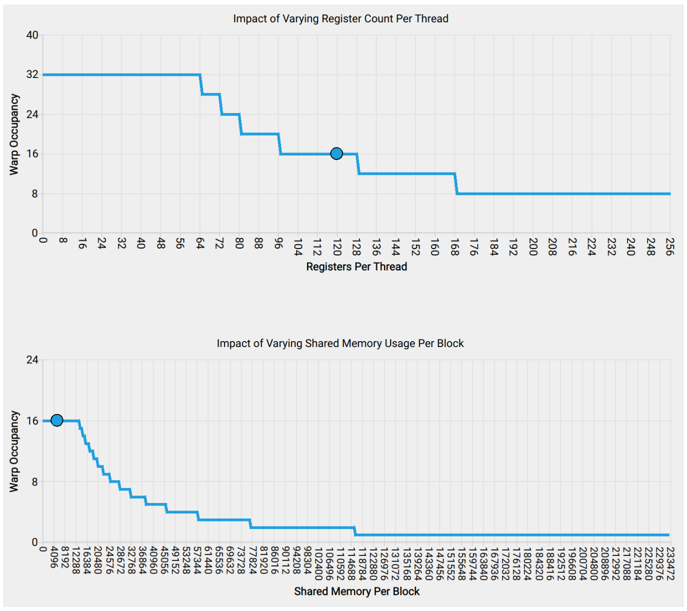


Figure 20. Warp occupancy as a function of registers per thread (top) and shared memory per thread block (bottom).

Next, NCU was used to profile the kernel using a T4 lysozyme molecule as a test case. The NCU throughput section reports achieved compute and memory throughput as a percentage of the theoretical maximum. Here, the kernel achieved 23.31% compute throughput and 13.01% memory throughput, both well below the recommended 60% threshold. This level of low throughput indicates latency-related kernel issues, suggesting that the kernel is limited by memory access latency and suboptimal memory access patterns.

Given the RTX 4060's theoretical peak performance (181.4 GFLOPS FP64, 256.0 GB/s bandwidth) [75], a roofline model was constructed using the kernel's measured double-precision arithmetic intensity, as shown in Figure 21. The kernel's performance reached neither the memory nor the compute peak, indicating that further optimization is possible. The performance analysis showed that the kernel achieved less than 1% of the device's single-precision peak and only 2% of its peak double-precision performance.

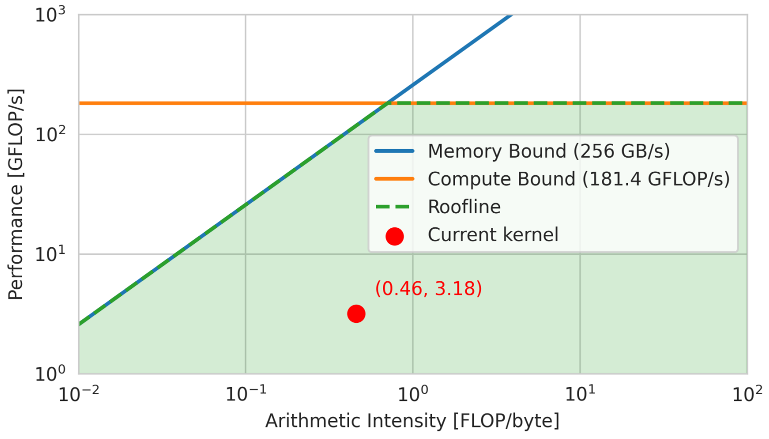


Figure 21. Roofline model for double precision on the RTX 4060 GPU. The kernel's performance is shown as a red dot.

Examining the memory throughput chart (Figure 22), an L1 cache hit rate of only 35.25% was observed, which likely contributed to the low memory throughput. On GPUs, memory accesses are most efficient when data is accessed sequentially, allowing the hardware to effectively use the cache lines (128 bytes) [37]. However, when accessing arrays with large strides or in higher dimensions, cache lines are frequently invalidated and reloaded, leading to poor cache utilization. Inside the kernel, many 2D, 3D, and 4D arrays are accessed with non-contiguous patterns, resulting in inefficient use of the L1 cache. For double-precision data (8 bytes per element), each L1 cache line can hold 16 elements, but with a 35.25% hit rate, most accesses require fetching new cache lines from slower memory. Ideally, perfectly aligned access would yield a near-100% L1 hit rate.

Source-level analysis in Nsight Compute revealed that the kernel suffers from both uncoalesced shared and global memory accesses. The most significant uncoalesced global accesses occur when reading the 4D enhancement factor (used to scale effective Hamiltonian elements) and when writing the 3D quadrupole integral matrix from shared to global

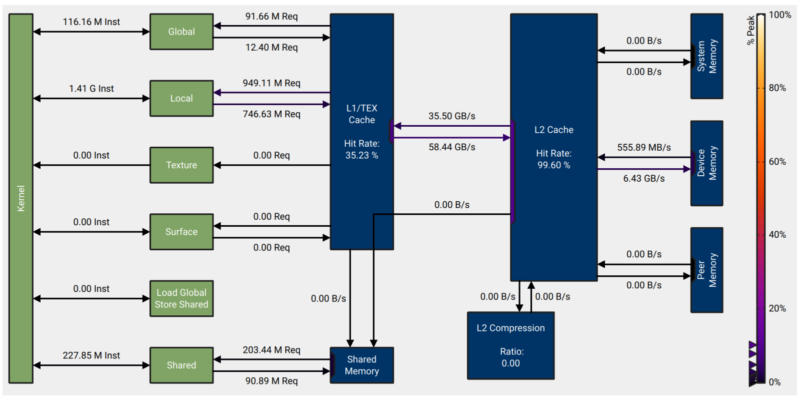


Figure 22. Memory chart showing logical (green) and physical (blue) memory resources. Arrows indicate read (left) and write (right) operations; arrow color reflects percentage of peak achievable rate.

memory. These could be improved by moving relevant data to shared memory or optimizing write patterns. Addressing these issues could yield an estimated 25.33% speedup.

Uncoalesced shared memory accesses are mainly due to custom access operators used for 3D integral matrices, especially during atomic accumulation of primitive Gaussian pair contributions. Improving striding and resolving bank conflicts could yield an additional 23.87% speedup.

In summary, the GPU kernel achieved a significant acceleration over the 32-CPU baseline. The GPU kernel's performance is limited by register pressure and suboptimal memory access patterns that lead to a low cache hit rate. Despite these problems, the GPU kernel showed up to two orders of magnitude speedup for large molecules over the CPU baseline. The batched calculation of small molecules also showed comparable speedups.

5 DISCUSSION

5.1 Current study

This study demonstrated that targeted GPU acceleration can deliver substantial speedups for computationally intensive and parallelizable tasks in MS/MS simulation workflows such as QCxMS. By porting the effective Hamiltonian kernel to CUDA C++ and integrating it with the existing FORTRAN code-base, speedups of up to two orders of magnitude were achieved for large and regular molecular systems, while maintaining full numerical accuracy. The integration of FORTRAN and CUDA C++ was accomplished using standardized language bindings, and the resulting GPU kernel was validated on a diverse set of molecular systems and NVIDIA GPUs.

The main objectives outlined in Section 1.2 were addressed as follows:

- **Literature review:** A comprehensive review of computational methods for *in silico* mass spectra generation was conducted, highlighting the computational challenges and the lack of GPU-accelerated solutions for SQM methods.
- **Profiling and bottleneck identification:** Detailed profiling of QCxMS identified the effective Hamiltonian calculation as a computationally expensive and parallelizable component suitable for GPU acceleration.
- **GPU implementation:** The effective Hamiltonian kernel was implemented in CUDA C++ and integrated with the FORTRAN code-base using standardized language bindings, enabling smooth data transfer and workflow integration.
- **Performance evaluation:** The GPU-accelerated kernel was validated and benchmarked on a range of molecular systems and NVIDIA GPUs, achieving up to 250x speedup for large, regular systems compared to a 32-core CPU baseline.

Limitations: The current GPU acceleration is limited to the effective Hamiltonian calculation for the semi-empirical TBlite method. Other critical components of the QCxMS workflow, such as the SCC and eigenvalue solvers, remain CPU-bound and are not yet ported to the GPU. The workflow is also constrained by CPU-to-GPU data transfer overhead. Additionally, the current implementation supports only elements up to Argon (Z=18), as d-orbital transformations are not yet available. These limitations mean that, while significant speedups are achieved in micro-benchmarks, full end-to-end acceleration of the entire workflow will require further development.

Comparison to Prior Work: This approach is not directly comparable to GPU4PySCF or other general-purpose GPU-accelerated quantum chemistry packages. The method here focuses exclusively on accelerating a single, performance-critical part (the effective Hamiltonian) of the xTB semi-empirical method within QCxMS. It is not a replacement for the existing TBlite or GPU4PySCF packages. Instead, this work demonstrated a targeted GPU-acceleration strategy that could be integrated into an existing FORTRAN code-base to address computational bottlenecks.

Reproducibility: All code, figure-generation scripts, and performance log files are publicly available on GitHub [62–64]. The code-bases are designed for modular integration using the Meson subproject system, with detailed documentation and example workflows to support reproducibility and further development.

Summary of Main Contributions:

- **GPU-accelerated effective Hamiltonian:** CUDA kernel for the effective Hamiltonian calculation, achieving up to 250x speedup for large, regular systems.
- **FORTRAN/CUDA C++ integration:** Standardized language bindings for smooth data transfer and workflow integration.
- **Validation and benchmarking suite:** Automated tests ensuring numerical accuracy within machine precision.
- **Open-source and reproducible workflow:** All code, scripts, and logs available on GitHub; modular integration via Meson.
- **Analysis of limitations:** Identification of remaining CPU-bound bottlenecks and data transfer overheads.

A key insight from profiling and benchmarking is that not all components of the QCxMS workflow are equally amenable to GPU acceleration. While the effective Hamiltonian and, potentially, the dispersion calculations are highly parallelizable, the SCC remains a major challenge due to its iterative nature and reliance on complex linear algebra routines. The current implementation accelerates only the effective Hamiltonian calculation, and as the experiments show, the overall workflow is still limited by CPU-to-GPU data transfer overhead and the non-accelerated SCC. This highlights the importance of a holistic approach to GPU porting: accelerating isolated kernels can yield impressive speedups in micro-benchmarks, but end-to-end workflow acceleration requires minimizing data movement and porting as much computation as possible to the GPU.

The use of modern C++ features such as operator overloading, templates, and custom array wrappers proved very useful for translating complex FORTRAN logic and managing multidimensional data structures. This not only facilitated the porting process but also improved code maintainability and correctness. However, the translation process also revealed limitations in language interoperability, particularly when dealing with complex or dynamically allocated data structures. Flattening and repackaging data for the C++ interface introduced some overhead but allowed for easier development and debugging.

The experiments with batched calculations further underscore the strengths of GPU acceleration for high-throughput scenarios, such as generating large synthetic spectral libraries or performing ensemble simulations. The ability to process many molecular configurations in parallel is particularly valuable for statistical sampling and uncertainty quantification in computational chemistry.

Overall, this work demonstrates the feasibility and benefits of GPU acceleration for quantum chemistry simulations, but also highlights the engineering challenges involved in porting complex scientific software to new architectures. The lessons learned here regarding profiling, kernel design, language interoperability, and workflow integration are likely to be relevant to other efforts in computational science seeking to leverage GPU hardware.

5.2 Future work

While the current GPU kernel achieves notable speedups for some parts of the MS simulation, several limitations and opportunities for further development remain. One important direction is kernel and memory optimization: reducing register usage, improving memory coalescing, and leveraging shared memory more effectively could further increase throughput. Exploring the use of lower-precision types in critical parts of code may also lead to improved performance.

Currently, only a subset of the single-point evaluation is GPU-accelerated. Porting additional components, such as the SCC and eigenvalue solvers, to the GPU would reduce reliance on CPU resources and minimize data transfer overhead. Achieving a workflow in which all critical data remains on the GPU throughout the simulation is key to unlocking the full performance potential and avoiding bottlenecks caused by limited CPU-to-GPU bandwidth.

The current implementation supports elements up to Argon (Z=18), as d-orbital transformations are not yet available. Extending support to heavier elements by implementing d-orbital handling and optimizing related memory usage would broaden the applicability of the software to a wider range of chemical systems.

Enabling more flexible batched calculations and high-throughput workflows would allow the simulation of large libraries of molecules in parallel, leading to better utilization of GPU hardware. Improving the modularity and interoperability of the code-base would also allow integration with other computational chemistry tools and workflows.

Finally, using a GPU-accelerated simulation workflow to generate high-quality synthetic mass spectra for training machine learning models represents a promising research direction. A dataset generation approach like this could improve the accuracy of molecular identification. Addressing these areas could enhance both the performance and flexibility of mass spectrometry machine learning models, as well as scientific applications in computational chemistry.

6 CONCLUSION

This thesis set out to address the challenge of accelerating *in silico* mass spectra simulation by leveraging GPU hardware. The main objectives were to identify computational bottlenecks in the QCxMS workflow, develop and integrate a GPU-accelerated kernel for the most intensive components, and validate the performance and accuracy of the resulting implementation.

By profiling QCxMS, the effective Hamiltonian calculation was identified as the computational bottleneck. This component was ported to CUDA C++ and integrated with the existing FORTRAN code-base using standardized language bindings. The resulting GPU kernel was validated on a diverse set of molecular systems and NVIDIA GPUs, achieving substantial speedups over a 32-core CPU baseline, up to 28x for large biomolecules and over 250x for long polymers, while preserving full numerical accuracy.

These results demonstrate that GPU acceleration can make *in silico* mass spectra generation significantly more accessible and efficient, paving the way for high-throughput computational workflows on widely available graphics hardware. While some workflow components remain CPU-bound and require further optimization, this work provides a foundation for future development and broader adoption of GPU acceleration for the simulation of mass spectra.

REFERENCES

[1] Edmond De Hoffmann and Vincent Stroobant. *Mass Spectrometry: Principles and Applications*. John Wiley & Sons, 2007.

[2] James J Pitt. Principles and Applications of Liquid Chromatography-Mass Spectrometry in Clinical Biochemistry. *The Clinical Biochemist Reviews*, 30(1):19, 2009.

[3] NIST Mass Spectrometry Data Center. NIST/EPA/NIH EI-MS LIBRARY. https://chemdata.nist.gov/dokuwiki/doku.php?id=chemdata:start, 2024. [Online; accessed November, 1, 2024].

[4] Hisayuki Horai, Masanori Arita, Shigehiko Kanaya, Yoshito Nihei, Tasuku Ikeda, Kazuhiro Suwa, Yuya Ojima, Kenichi Tanaka, Satoshi Tanaka, Ken Aoshima, et al. MassBank: A public repository for sharing mass spectral data for life sciences. *Journal of Mass Spectrometry*, 45(7):703–714, 2010.

[5] Michael Murphy, Stefanie Jegelka, Ernest Fraenkel, Tobias Kind, David Healey, and Thomas Butler. Efficiently predicting high resolution mass spectra with graph neural networks. In *International Conference on Machine Learning*, pages 25549–25562. PMLR, 2023.

[6] Fei Wang, Jaanus Liigand, Siyang Tian, David Arndt, Russell Greiner, and David S. Wishart. CFM-ID 4.0: More accurate ESI-MS/MS spectral prediction and compound identification. *Analytical Chemistry*, 93(34):11692–11700, 2021. PMID: 34403256.

[7] Adamo Young, Hannes Röst, and Bo Wang. Tandem mass spectrum prediction for small molecules using graph transformers. *Nature Machine Intelligence*, 6(4):404–416, 2024.

[8] Jeroen Koopman and Stefan Grimme. From QCEIMS to QCxMS: A tool to routinely calculate CID mass spectra using molecular dynamics. *Journal of the American Society for Mass Spectrometry*, 32(7):1735–1751, 2021.

[9] Shunyang Wang, Tobias Kind, Dean J Tantillo, and Oliver Fiehn. Predicting in silico electron ionization mass spectra using quantum chemistry. *Journal of cheminformatics*, 12(1):63, 2020.

[10] Naythan Yeo, Dillon Tay, and Shi Jun Ang. Benchmarking tandem mass spectra of small natural product molecules via ab initio molecular dynamics. https://doi.org/10.26434/chemrxiv-2023-7k5x8, 2023. This content is a preprint and has not been peer-reviewed.

[11] Helge Hecht, Wudmir Y Rojas, Zargham Ahmad, Aleš Krěnek, Jana Klánová, and Elliott J Price. Quantum chemistry-based prediction of electron ionization mass spectra for environmental chemicals. *Analytical Chemistry*, 96(33):13652–13662, 2024.

[12] Rui Li, Qiming Sun, Xing Zhang, and Garnet Kin-Lic Chan. Introducing GPU Acceleration into the Python-Based Simulations of Chemistry Framework. *The Journal of Physical Chemistry A*, 2025.

[13] Frank Jensen. *Introduction to computational chemistry*. John wiley & sons, 2017.

[14] Jonathan Clayden, Nick Greeves, and Stuart Warren. *Organic Chemistry*. Oxford University Press, USA, 2012.

[15] Stefan Grimme. Density functional theory with London dispersion corrections. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(2):211–228, 2011.

[16] Overview of Mass Spectrometry for Protein Analysis. https://www.thermofisher.com/ch/en/home/life-science/protein-biology/protein-biology-learning-center/protein-biology-resource-library/pierce-protein-methods/overview-mass-spectrometry.html, 2025. [Online; accessed 25-February-2025].

[17] Mass Spectrometry - Fragmentation Patterns, 2025. [Online; accessed 25-February-2025].

[18] Timothy MD Ebbels, Justin JJ van der Hooft, Haley Chatelaine, Corey Broeckling, Nicola Zamboni, Soha Hassoun, and Ewy A Mathé. Recent advances in mass spectrometry-based computational metabolomics. *Current opinion in chemical biology*, 74:102288, 2023.

[19] Wout Bittremieux, Mingxun Wang, and Pieter C Dorrestein. The critical role that spectral libraries play in capturing the metabolomics community knowledge. *Metabolomics*, 18(12):94, 2022.

[20] Tornike Onoprishvili, Jui-Hung Yuan, Kamen Petrov, Vijay Ingalalli, Lila Khederlarian, Niklas Leuchtenmuller, Sona Chandra, Aurelien Duarte, Andreas Bender, and Yoann Gloaguen. SimMS: a GPU-accelerated cosine similarity implementation for tandem mass spectrometry. *Bioinformatics*, 41(3):btaf081, 2025.

[21] Maria Emilia Dueñas, Rachel E Peltier-Heap, Melanie Leveridge, Roland S Annan, Frank H Büttner, and Matthias Trost. Advances in high-throughput mass spectrometry in drug discovery. *EMBO Molecular Medicine*, 15(1):e14850, 2023.

[22] Richard A Friesner and Michael D Beachy. Quantum mechanical calculations on biological systems. *Current opinion in structural biology*, 8(2):257–262, 1998.

[23] Yuriko Aoki, Yuuichi Orimoto, and Akira Imamura. *Quantum chemical approach for organic ferromagnetic material design*. Springer, 2017.

[24] Kaushik Raha, Martin B Peters, Bing Wang, Ning Yu, Andrew M Wollacott, Lance M Westerhoff, and Kenneth M Merz Jr. The role of quantum mechanics in structure-based drug design. *Drug discovery today*, 12(17-18):725–731, 2007.

[25] Scott A Hollingsworth and Ron O Dror. Molecular dynamics simulation for all. *Neuron*, 99(6):1129–1143, 2018.

[26] Tamás Turányi and Alison S Tomlin. *Analysis of kinetic reaction mechanisms*, volume 20. Springer, 2014.

[27] Steven Brown, Romà Tauler, and Beata Walczak. *Comprehensive chemometrics: chemical and biochemical data analysis*. Elsevier, 2020.

[28] C David Sherrill. An introduction to Hartree-Fock molecular orbital theory. *School of Chemistry and Biochemistry Georgia Institute of Technology*, 2000.

[29] Yasuaki Ito, Satoki Tsuji, Haruto Fujii, Kanta Suzuki, Nobuya Yokogawa, Koji Nakano, and Akihiko Kasagi. Introduction to computational quantum chemistry for computer scientists. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 273–282. IEEE, 2024.

[30] Christoph Bannwarth, Eike Caldeweyher, Sebastian Ehlert, Andreas Hansen, Philipp Pracht, Jakob Seibert, Sebastian Spicher, and Stefan Grimme. Extended tight-binding quantum chemistry methods. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 11(2):e1493, 2021.

[31] Light-weight tight-binding framework. https://github.com/tblite/tblite. [Online; accessed June, 27, 2025].

[32] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[33] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[34] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[35] Mark J Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Graphics Hardware*, volume 2002, pages 1–10, 2002.

[36] NVIDIA. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit, 2020.

[37] NVIDIA. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Online; accessed May, 30, 2025].

[38] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[39] NVIDIA H100 NVL GPU. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c?ncid=no-ncid. [Online; accessed June, 3, 2025].

[40] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[41] CUDA FORTRAN. https://developer.nvidia.com/cuda-fortran. [Online; accessed June, 26, 2025].

[42] How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog. https://siboehm.com/articles/22/CUDA-MMM, 2025. [Online; accessed 25-February-2025].

[43] AMD EPYC 9965. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html. [Online; accessed June, 3, 2025].

[44] Felicity Allen, Allison Pon, Michael Wilson, Russ Greiner, and David Wishart. CFM-ID: a web server for annotation, spectrum prediction and metabolite identification from tandem mass spectra. *Nucleic acids research*, 42(W1):W94–W99, 2014.

[45] Sebastian Böcker and Kai Dührkop. Fragmentation trees reloaded. *Journal of cheminformatics*, 8:1–26, 20156.

[46] Kai Dührkop, Markus Fleischauer, Marcus Ludwig, Alexander A Aksenov, Alexey V Melnik, Marvin Meusel, Pieter C Dorrestein, Juho Rousu, and Sebastian Böcker. SIRIUS 4: a rapid tool for turning tandem mass spectra into metabolite structure information. *Nature methods*, 16(4):299–302, 2019.

[47] Christoph Ruttkies, Emma L Schymanski, Sebastian Wolf, Juliane Hollender, and Steffen Neumann. MetFrag relaunched: incorporating strategies beyond in silico fragmentation. *Journal of cheminformatics*, 8:1–16, 2016.

[48] Mingxun Wang, Jeremy J Carver, Vanessa V Phelan, Laura M Sanchez, Neha Garg, Yao Peng, Don Duy Nguyen, Jeramie Watrous, Clifford A Kapono, Tal Luzzatto-Knaan, et al. Sharing and community curation of mass spectrometry data with Global Natural Products Social Molecular Networking. *Nature biotechnology*, 34(8):828–837, 2016.

[49] Stefan Grimme. Towards first principles calculation of electron impact mass spectra of molecules. *Angewandte Chemie International Edition*, 52(24), 2013.

[50] Johannes Gorges and Stefan Grimme. QCxMS2 - a program for the calculation of electron ionization mass spectra via automated reaction network discovery. https://arxiv.org/abs/2105.13979, 2025.

[51] Florian Huber, Stefan Verhoeven, Christiaan Meijer, Hanno Spreeuw, Efraín Manuel Villanueva Castilla, Cunliang Geng, Simon Rogers, Adam Belloum, Faruk Diblen, Jurriaan H Spaaks, et al. matchms - processing and similarity evaluation of mass spectrometry data. *Journal of Open Source Software*, 5(52):2411, 2020.

[52] Christoph Bannwarth, Sebastian Ehlert, and Stefan Grimme. Gfn2-xtb—an accurate and broadly parametrized self-consistent tight-binding quantum chemical method with multipole electrostatics and density-dependent dispersion contributions. *Journal of Chemical Theory and Computation*, 15(3):1652–1671, 2019.

[53] Mingxun Wang, Jeremy J Carver, Vanessa V Phelan, Laura M Sanchez, Neha Garg, Yao Peng, Don Duy Nguyen, Jeramie Watrous, Clifford A Kapono, Tal Luzzatto-Knaan, et al. Sharing and community curation of mass spectrometry data with Global Natural Products Social Molecular Networking. *Nature biotechnology*, 34(8):828–837, 2016.

[54] José Manuel Vicent-Luna, Sofia Apergi, and Shuxia Tao. Efficient computation of structural and electronic properties of halide perovskites using density functional tight binding: Gfn1-xtb method. *Journal of chemical information and modeling*, 61(9):4415–4424, 2021.

[55] Giuseppe MJ Barca, Colleen Bertoni, Laura Carrington, Dipayan Datta, Nuwan De Silva, J Emiliano Deustua, Dmitri G Fedorov, Jeffrey R Gour, Anastasia O Gunina, Emilie Guidez, et al. Recent developments in the general atomic and molecular electronic structure system. *The Journal of chemical physics*, 152(15), 2020.

[56] Jesi Lee, Dean Joseph Tantillo, Lee-Ping Wang, and Oliver Fiehn. Predicting Collision-Induced-Dissociation Tandem Mass Spectra (CID-MS/MS) Using Ab Initio Molecular Dynamics. *Journal of Chemical Information and Modeling*, 64(19):7470–7487, 2024.

[57] Ivan S Ufimtsev and Todd J Martinez. Quantum chemistry on graphical processing units. 3. analytical energy gradients, geometry optimization, and first principles molecular dynamics. *Journal of Chemical Theory and Computation*, 5(10):2619–2628, 2009.

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

[59] Michael Metcalf, John Reid, and Malcolm Cohen. *Modern Fortran Explained: Incorporating Fortran 2018.* Oxford University Press, 2018.

[60] Meson. https://github.com/mesonbuild/meson. [Online; accessed June, 27, 2025].

[61] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95, 2007.

[62] TBlite GPU. https://github.com/tornikeo/tblite-gpu. [Online; accessed June, 27, 2025].

[63] QCxMS. https://github.com/tornikeo/qcxms. [Online; accessed June, 27, 2025].

[64] Light-weight tight-binding framework. https://github.com/tblite/tblite. [Online; accessed June, 27, 2025].

[65] NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. [Online; accessed June, 3, 2025].

[66] R Diamond. Real-space refinement of the structure of hen egg-white lysozyme. *Journal of molecular biology*, 82(3):371–391, 1974.

[67] Dirk W Heinz, Walter A Baase, Frederick W Dahlquist, and Brian W Matthews. How amino-acid insertions are allowed in an α-helix of T4 lysozyme. *Nature*, 361(6412):561–564, 1993.

[68] PubChem. https://pubchem.ncbi.nlm.nih.gov/. [Online; accessed June, 25, 2025].

[69] Intel core i9 processor 14900HX. https://www.intel.com/content/www/us/en/products/sku/23599
core-i9-processor-14900hx-36m-cache-up-to-5-80-ghz/specifications.html. [Online; accessed June, 3, 2025].

[70] Vast.ai. https://cloud.vast.ai. [Online; accessed June, 27, 2025].

[71] NVIDIA Tesla V100 PCIe 32 GB. https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184. [Online; accessed June, 3, 2025].

[72] NVIDIA GeForce RTX 4090. https://www.techpowerup.com/gpu-specs/geforce-rtx-4090.c3889. [Online; accessed June, 3, 2025].

[73] NVIDIA Development Tools Solutions - ERR_NVGPUCTRPERM: Permission issue with Performance Counters. https://developer.nvidia.com/ERR_NVGPUCTRPERM. [Online; accessed June, 1, 2025].

[74] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2139–2153, 2018.

[75] NVIDIA GeForce RTX 4060. https://www.techpowerup.com/gpu-specs/geforce-rtx-4060.c4107. [Online; accessed June, 3, 2025].