

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
Faculty of Chemical and Food Technology

Reg. No.: FCHPT-5414-44240

Quantum-chemical computing on GPU

Master thesis

2015

Bc. Ján Minárik

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
Faculty of Chemical and Food Technology

Reg. No.: FCHPT-5414-44240

Quantum-chemical computing on GPU

Master thesis

Study programme: Automation and Information Engineering in Chemistry and Food Industry

Study field number: 2621

Study field: 5.2.14. Automation

Training workplace: Institute of Information Engineering, Automation and Mathematics

Thesis supervisor: Ing. Marián Gall, PhD.

Consultant: Lukáš Bučinský, Radovan Bast



ZADANIE DIPLOMOVEJ PRÁCE

Študent: **Bc. Ján Minárik**
ID študenta: 44240
Študijný program: automatizácia a informatizácia v chémii a potravinárstve
Študijný odbor: 5.2.14. automatizácia
Vedúci práce: Ing. Marián Gall, PhD.
Konzultant: Lukáš Bučinský, Radovan Bast

Názov práce: **Kvantovo-chemické výpočty na GPU**

Špecifikácia zadania:

Silná potreba výpočtového výkonu v oblasti vedy vedie k použitiu aj iných prostriedkov ako klasické využívanie jednojadrových CPU. Viecjadrové CPU, GPU a ďalšie akcelerátory sa používajú ako ko-procesory pre intenzívne aritmetické dátovo-paralelné výpočtové úkony. V súčasnej dobe existujú dva dominantné API rozhrania pre výpočty na GPU, CUDA a OpenCL. Cieľom tohto projektu je študovať možnosti týchto rozhraní API a ich využitie na urýchlenie základných výpočtov v kvantovej chémii na komerčných GPU kartách.

Riešenie zadania práce od: 16. 02. 2015
Dátum odovzdania práce: 24. 05. 2015

L. S.

Bc. Ján Minárik
študent

prof. Ing. Miroslav Fikar, DrSc.
vedúci pracoviska

prof. Ing. Miroslav Fikar, DrSc.
garant študijného programu

Žiadosť

o súhlas s vypracovaním a obhájením

záverečnej práce

v inom ako štátnom jazyku

meno a priezvisko, ID	Ján Minárik, 44 240
stupeň štúdia	2.
oddelenie	Ústav informatizácia, automatizácie a matematiky
študijný program	Automatizácia a informatizácia v chémii a potravinárstve
jazyk práce	Anglický
Názov práce v SJ	Kvantovo-chemicke výpočty na GPU
Názov práce v inom* jazyku	Quantum-chemical computing on GPU

*anglicky

Dátum: 3.5.2015

Podpis študenta

Vyjadrenie súhlasu/nesúhlasu (zdôvodnenie)

Vedúci práce

Garant ŠP:

dekan/prodekan:

I would like to express my gratitude to my supervisor Marián Gall from STU Bratislava, voluntary consultants Lukáš Bučický STU Bratislava and Radovan Bast from KTH Stockholm for their help, understanding and guidance.

In Bratislava 24.5.2015

Bc. Ján Minárik

Abstract

Computational chemistry has a wide range of applications. Most notable are predictions of real world experiments and estimates of a molecular properties. Problem of computational chemistry algorithm is that they are computationally demanding. Large amount of computer memory and processing power is required. Therefore computational acceleration is a natural choice. In last decade graphic cards evolved rapidly. Nowadays they are capable of floating-point arithmetics. This is not beneficial only for game enthusiasts but also for scientific community. While the former are enjoying near real world video experience the later are developing algorithms witch can benefit from large amount of single instruction processing units.

In my thesis I examine utilization of modern programming techniques for scientific calculation both on CPU and GPU architectures. I focus on GPU computation. To demonstrate its efficiency I wrote my own code for calculation of the exchange-correlation energy. The code is used in an experiment on a Neon atom and compared with pure C++ code and usage of optimized math arithmetics library. In the thesis is necessary theoretical background from computational chemistry. Followed by basic concepts of math kernel library and graphic card programming for scientific computing. The written code is explained in detail. Results from a computational study are presented and discussed.

Keywords: computational chemistry, parallel computing on GPU, Intel MKL, CUDA.

Abstrakt

Počítačová chémia má široký rozsah praktických aplikácií. Najznámejšie sú programy na predikciu experimentov a odhad vlastností nových molekúl. Problém algoritmov používaných v počítačovej chémii je ich vysoká výpočtová náročnosť. Vyžadujú obrovské množstvo počítačovej pamäte a obrovský výpočtový výkon. Preto sa používajú rôzne techniky zrýchlenia výpočtu. V poslednom desaťročí nastal obrovský rozmach grafických kariet (GPU). Dnešné GPU dokážu vykonávať operácie s plávajúcou desatinnou čiarkou. Táto vlastnosť nieje vítaná len vo svete hráčov počítačových hier, ale aj vo vedeckej komunite. Zatiaľčo prvá skupina sa teší z takmer realistického herného zážitku, tá druhá z veľkého množstva grafických procesorov.

V mojej práci skúmam efekty využitia moderných programových nástrojov na zefektívnenie vedecko-technických výpočtov na CPU aj GPU. Primárne sa venujem využitiu grafických kariet. Aby som ukázal aké zefektívnenie výpočtu ponúkajú naprogramoval som výpočet výmenno-korelačnej energie. Tento kód okrem štandardného C++ jazyka využíva vektorizované a optimalizované matematické knižnice. Takisto obsahuje funkcie na výpočty na grafickej karte. Porovnávam jednotlivé metódy na príklade atómu Neónu. V práci sa nachádza potrebný úvod do počítačovej chémie a zrýchľovania vedecko-technických výpočtov. Do detailov vysvetľujem svoj kód a na záver ukazujem zrýchlenie na príklade atómu Neónu.

Kľúčové slová: počítačová chémia, parallélne výpočty na GPU, Intel MKL, CUDA.

Contents

1	Introduction	4
2	Computational Chemistry	6
2.1	Computable qualities	6
2.1.1	Structure	6
2.1.2	Potential Energy Surfaces	6
2.1.3	Chemical Properties	7
2.2	Quantum Mechanics	7
2.2.1	The Schrödinger's equation	7
2.2.2	The Hamiltonian operator	8
2.2.3	The variational principle	9
2.2.4	The Born-Oppenheimer approximation	9
2.2.5	The LCAO basis sets	10
2.2.6	Hartree-Fock computation	11
2.2.7	Density Functional Theory	12
2.3	Software for computational chemistry	13
2.3.1	Commercial: Crystal	13
2.3.2	Open-source: XCint	14
3	Computation acceleration	17
3.1	Intel MKL	17
3.2	GPU Computing	18
3.2.1	OpenCL	19
3.2.2	CUDA	19
3.3	Computation study	20
4	Grid Calculation	23
4.1	Compilation	24
4.2	Input data parsing	24
4.3	Atomic orbital value calculation	25
4.4	Calculate electron density	25
4.4.1	CPU sequential	26
4.4.2	CPU screening	26
4.4.3	CPU batch	27
4.4.4	GPU sequential	28
4.5	Exchange-Correlation Energy	28

4.6	Print output	29
5	Conclusion	30

Chapter 1

Introduction

Computational chemistry uses computer programs to answer questions from theoretical chemistry such as [1]:

1. Which geometrical arrangements are possible stable molecules ?
2. What are their relative energies ?
3. What are their physical and chemical properties ?
4. What is the rate at which one stable molecule can transform into another one ?
5. What is the time dependence of molecular structures and properties?
6. How do molecules interact ?

if we know initial set of nuclei and electrons. With sufficient hardware and tailored software the answer may be found. But processing power is not infinite. Algorithms usually takes long time (i.e. hours to days) to perform and accuracy is limited [1, 2]. Usually more precise algorithm more time it takes to complete. Fast hardware development in recent years offers new possibilities. To utilize new computer components old algorithms have to be reprogrammed and adjust. Usually it means to develop a handy application programming interface (API) first. Therefore with each new generation of hardware architecture a new programming techniques are developed.

Graphic processing units (GPU) (i.e. graphic cards) were part of compute hardware for a long time. But only since 2003 they are capable of floating point arithmetics. This feature attracts scientists in last decade and questions such as

1. Could GPUs be utilized for scientific computing ?
2. If so, how much effort must be put to rewrite the old computer codes ?
3. Will be that effort worth it ?
4. How much faster would be the computations ?

5. What are the limitations ?

are asked. Numerous studies has been done to show improvement gained from using GPU for scientific computing [12, 13]. Major vendors also provide their optimized API. Two most dominant are CUDA for NVidias GPUs and OpenCL mainly used on ATI cards. In the chapter 3 I provide an overview about both APIs. If a new technology wants to be successful it must be competitive with the old ones. Same holds in programming. Therefore I provide a quick overview of Math Kernel Library (MKL) for vectorized and optimized arithmetics used on Intel CPUs (core processing units) in same chapter.

The thesis is focused on developing a code for a computational study of three approaches.

1. Pure C++ code.
2. Code using Intel MKL library.
3. CUDA code for GPU.

Then all three codes are compared on a test case atom of a Neon based on computational time. Whole code with data for the experimental atom can be downloaded from my Git repository [15] and run on any Linux operating system. On the repository web page is also the readme file with compilation instructions. Detailed explanation of the code and used algorithms is provided in the chapter 4. A description of the computational case study with the result is presented in the chapter 5.

Chapter 2

Computational Chemistry

2.1 Computable qualities

The postulates and theorems of quantum mechanics form the foundation for the prediction of observable chemical properties. But there is one important question which needs to be asked first. Which properties can be predicted ? Answer is simple. If we can measure it, we can predict it. Christopher Cramer in his book [2] divided molecular properties into three groups: structure, potential energy surfaces (PES) and chemical properties.

2.1.1 Structure

What is the best structure of a molecule ? To describe any molecule we need only its chemical formula, i.e. atoms from which it is composed. Because *the best* means that the interacting forces are zero at the given initial positions of atoms. It is a task of a structural optimization. Real world problem is that molecules are in forms of multiple discrete stereoisomers, tautomers, etc. Therefore great care is taken when comparing real world experiment and idealized theory.

2.1.2 Potential Energy Surfaces

First step in a simulation of a molecule is to consider not just one structure for a given chemical formula, but all possibilities. This characterizes Potential Energy Surface (PES) for the given chemical formula based on Born-Oppenheimer approximation. The PES is a hypersurface defined by the potential energy of a collection of atoms over all possible atomic arrangements [2]. Thus PES is defined as a vector \mathbb{X} :

$$\mathbb{X} = [x_1, y_1, z_1, \dots, x_N, y_N, z_N] \quad (2.1)$$

where x, y, z are Cartesian coordinates of atom i . Interesting points on PES are local minima and saddle points. Former correspond to optimal molecular structures and later are lowest energy barriers on paths connecting them. They are chemical concept of transition states. So a complete PES is map of all possible chemical structure states and all isomerisations paths interconnecting them.

Problem with PES is that they are hard to visualize because they involve many dimensions. Instead slices and projections like in figure 2.1 are used. There is desired chemical property visualized as function of one, or reduced number of coordinates. Sometimes structures can be grouped by a common symmetry.

2.1.3 Chemical Properties

In this group belongs single molecule properties as spectral quantities. Predicting nuclear magnetic resonance (NMR) chemical shifts and coupling constants, electron paramagnetic resonance (EPR) hyperfine coupling constants, absorption maxima and i.e. has many practical applications. For statistical reasons usually more molecules are considered.

It is possible to measure total energy of molecule. It is minimal energy necessary to separate it into nuclei and electrons. However in experiments focus is more on particular thermodynamic quantity such as enthalpy or free energy. This is used either before or after conducting real-world experiment. In former case it is used in design. In later to tune the experiment. In comparing chemists can observe many molecular and chemical reaction properties that were hidden during the experiment.

At last, there are computable properties that do not link directly to real physical properties. But they have great value in describing molecules. Good examples include aromaticity and partial atomic charge.

2.2 Quantum Mechanics

To fully understand complexity of computational chemistry one needs to have fundamental knowledge of quantum mechanics. This chapter provides reader with elementary methods without going into mathematical details. I refer reader to [1, 2] for more detail. The interacting gravitational force between two particles in classical mechanics is [1]:

$$\mathbf{V}(\mathbf{r}_{12}) = -C_{grav} \frac{m_1 m_2}{r_{12}} \quad (2.2)$$

Where m_1, m_2 are weights of particles and r_{12} is distance between particles. But in case of nucleus-electron interaction the only significant force which holds particles together is Coulomb interaction:

$$\mathbf{V}(\mathbf{r}_{12}) = \frac{q_1 q_2}{r_{12}} \quad (2.3)$$

with q_1, q_2 being charges. Coulomb force 2.3 is 10^{38} times larger than gravitational force 2.2.

2.2.1 The Schrödinger's equation

Because electrons have negligible weight and are moving close to speed of light ($3 \times 10^8 \text{ms}^{-1}$) traditional deterministic mechanics must be replaced by quantum mechanics [1]. With deterministic equations past and future position of particles can

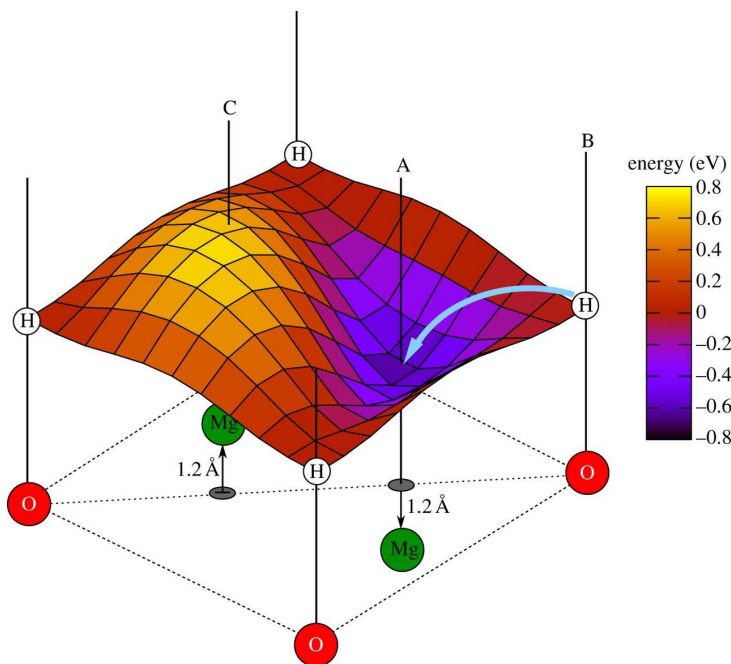


Figure 2.1: PES for Mg_2O_4 energy.

be calculated. Quantum mechanics on the other hand are probabilistic. They predict probability of particle being in chosen space. Fundamental equation of quantum mechanics is Schrödinger's equation. For bounded system time independent form is used 2.4.

$$\mathbf{H}\Psi = E\Psi \quad (2.4)$$

where \mathbf{H} is Hamiltonian operator, Ψ is wave-function and E is scalar value of system energy. Most important is that product of Ψ with it's complex conjugate pair ($|\Psi^*\Psi|$) has units of probabilistic density. Therefore the probability that chemical system will be found in multidimensional space is equal to integral of $|\Psi^2|$ over that space [2]. Thus there are constrains what can be a wave function. For a bound particle normalized integral of Ψ must be unite. Probability of finding it somewhere is one. In addition Ψ must be continuous, single-valued and quadratically integrable [2].

2.2.2 The Hamiltonian operator

The Hamiltonian operator from equation 2.4 typically represents five contributions to total energy of a system (molecule or atom). These are kinetic energy of electrons and nuclei, the attraction of electrons to nuclei, interelectronic and internuclear

repulsions. More complicated version of Hamiltonian can include presence of external electric and magnetic field or relativistic effects [2]. Mathematical form of Hamiltonian is:

$$\mathbf{H} = - \sum_i \frac{\hbar^2}{2m_e} \nabla_i^2 - \sum_k \frac{\hbar^2}{2m_k} \nabla_k^2 - \sum_i \sum_k k \frac{e^2 Z_k}{r_{ik}} + \sum_{i < j} \frac{e^2}{r_{ij}} + \sum_{k < l} \frac{e^2 Z_k Z_l}{r_{kl}} \quad (2.5)$$

where i, j iterates over electrons, k, l runs over nuclei, \hbar is reduced Planck's constant, m_e is mass of electron, m_k is mass of nucleus k , ∇^2 is Laplacian operator, e is charge of electron, Z is an atomic number and r_{ij} is distance between particles i and j . Thus Ψ is a function of $3n$ Cartesian coordinates with n being total number of particles (electrons and nuclei), i.e. x, y and z coordinate of each particle. In three-dimensional Cartesian space Laplacian has form of:

$$\nabla_i^2 = \frac{\partial^2}{\partial x_i^2} + \frac{\partial^2}{\partial y_i^2} + \frac{\partial^2}{\partial z_i^2} \quad (2.6)$$

2.2.3 The variational principle

If we have wave-function Ψ we can calculate other atomic observable physical properties by switching Hamiltonian operator \mathbf{H} . Problem is how to calculate molecular properties. It is clear that molecules are composed of atoms. Therefore it is possible to assume that an arbitrary function Φ exists which is function of individual electronic and nuclear coordinates operated upon by the Hamiltonian. Because set of orthonormal wave-functions Ψ_i is complete, the function Ω is a linear combination of Ψ s [2].

$$\Phi = \sum_i c_i \Psi_i \quad (2.7)$$

Because individual Ψ_i are unknown also coefficients are unknown. But they are constrained.

$$\begin{aligned} \int \Phi d\mathbf{r} &= 1 = \int \sum_i c_i \Psi_i \sum_j c_j \Psi_j d\mathbf{r} = \\ &= \sum_{ij} c_i c_j \int \Psi_i \Psi_j d\mathbf{r} = \\ &= \sum_{ij} c_i c_j \delta_{ij} \\ &= \sum_{ij} c_i c_j \end{aligned} \quad (2.8)$$

Values of coefficients c_i, c_j are calculated iteratively. In nature a system is stable in ground state if it has the lowest energy.

2.2.4 The Born-Oppenheimer approximation

Typically electrons are moving much faster than nuclei because they are about 1800 times lighter and mass appears in denominator in Hamiltonian equation 2.5.

Therefore it practical to decouple this two motions and compute electronic energies for fixed nuclear positions [2]. Formally decoupled Schrödinger's equation 2.4 is written as following:

$$(H_{el} + V_N)\Psi_{el}(\mathbf{q}_i; \mathbf{q}_k) = E_{el}\Psi_{el}(\mathbf{q}_i; \mathbf{q}_k) \quad (2.9)$$

where H_{el} includes only first, third and fourth term of 2.5, V_N is nuclear-nuclear repulsion energy and q_i are independent variables with q_k being nuclear coordinates parameters. The term V_N is constant for a fixed set of nuclear coordinates. Note that PES is calculated E_{el} over all possible nuclear coordinates.

2.2.5 The LCAO basis sets

As mentioned earlier any functions which has certain criteria can be a wave-function. Two wave-functions are compared based on energy calculated when using them. The one with lower is more suitable as a wave-function for calculating other properties by substituting Hamiltonian for another operator. Convenient functions are called 'basis sets'. For system with only one nucleus equation 2.9 can be solved exactly without guessing of wave-functions. Question is how can we construct a molecular wave-function ? Same as we in eq.2.7 we can construct it as a linear combination of known atomic wave-functions:

$$\phi = \sum_{i=1}^N a_i \varphi_i \quad (2.10)$$

with basis set of N φ_i function and associated coefficients a_i . This is Linear Combination of Atomic Orbitals (LCAO) approach. Usually these basis sets are centered on atomic nucleus but it is not a requirement.

There is one question to answer. What is the form of basis set function φ ? Slater-type orbitals (STOs) are describing real hydrogenic orbitals the best. They converge rapidly with increasing number of functions [2].

$$\varphi(r, \theta, \phi,) = NY(\theta, \phi)r^{n-1}e^{\zeta r} \quad (2.11)$$

where N is normalization constant, $Y(\theta, \phi)$ describe spherical harmonics, r is distance from nucleus, ζ is an exponent. All of these can be found in any quantum chemistry database. Limitation of using STOs is that there is no analytical solution for 4 index integral. It needs to be solved numerically which is computationally extensive operation. Therefore they cannot be used for any significant large molecule. Improvement was proposed by Boys in 1950. Exponent $e^{\zeta r}$ is changed to $e^{\zeta r^2}$ and Gaussian-type orbitals (GTOs) are formed. GTO in Cartesian coordinates has form of:

$$\varphi(x, y, z) = Nx^{l_x}y^{l_y}z^{l_z}e^{\zeta r^2} \quad (2.12)$$

where l_x, l_y, l_z are non-negative integer representing type of orbital. For example p-type of orbital is represented by three vectors of $[l_x, l_y, l_z]$ namely $[1, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$. GTOs have an analytical solution but there is a price for that. Near the nucleus GTO has a zero slope, therefor behavior is poorly represented and far away from nucleus GTOs converge too rapidly to zero. Also extra d-, f-, g-

functions from Cartesian representation may lead to linear dependence in large basis sets, therefore they are usually dropped. Linear combination of GTOs may overcome these difficulties [3]. Difference in converge is shown on figure 2.2.

At this point is the fundamental theoretical background covered. In next two section I describe a two popular computation principles Hartree-Fock theory (H3F) and Density Functional Theory (DFT).

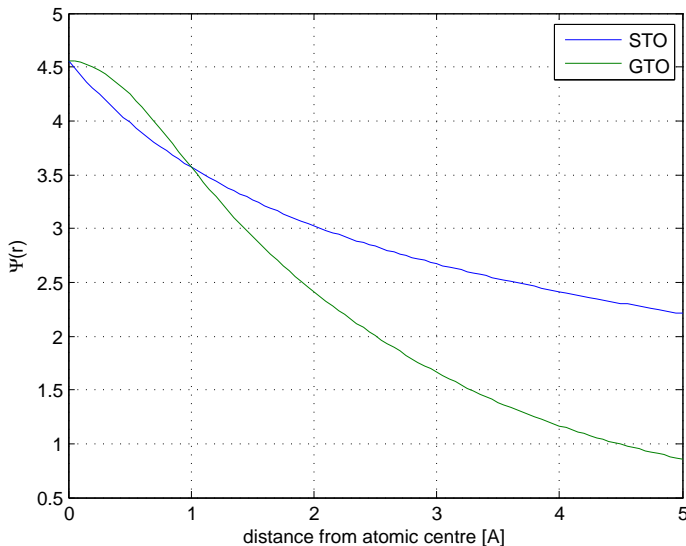


Figure 2.2: Difference between STO and GTO functional in molecule of H_2^+

2.2.6 Hartree-Fock computation

Many electron system such as a molecule has very complex dynamics which are hard to compute. A simplification is to use an independent particle model where motion of an electron is considered to be independent of the dynamics of all other electrons [1]. This method does not ignore interelectron interactions but use an approximation instead. Usually it is taken only the most important (strongest) one or an average into account as in Hartree-Fock (HF) theory. In HF model each electron is described by an orbital and total wave function is the product of all orbitals. The best set of orbitals is determined by variational principle (section 2.2.3). Because all other electrons are described by their own orbitals, HF equations depends on their own solution. Therefore they must be solved iteratively. During computation Fock matrix is constructed by expanding molecular orbitals into basis sets. Then a solution is found as a matrix eigenvalue problem. The elements in Fock matrix are integrals of one and two electron operators multiplied by density matrix. HF equations can be improved by adding additional determinants to converge to

exact solution of Schrödinger's equations. Or by adding additional approximations in semi-empirical methods. This is show in figure 2.3.

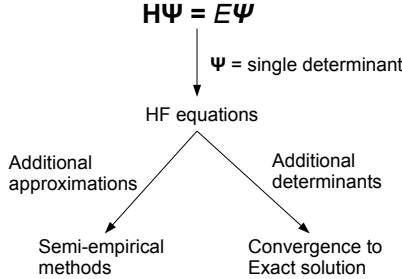


Figure 2.3: Improvement of Hartree-Fock method.

Because HF theory is the best single-determinant wave function approach it takes into account only average electron-electron interactions. Methods that cover electron correlation require a multi-determinant wave function [1]. Multi-determinant methods are computationally more extensive but systematically converge to exact solution of Schrödinger's equation (eq 2.4).

2.2.7 Density Functional Theory

Density Functional Theory (DFT) is an improvement to HF theory. It was proposed by Kohn-Sham [4]. The many electron correlation effect is modeled by a physical property. Because Hamiltonian is depended only on position and atomic number of nuclei and total number of electrons electron density ρ is natural choice for this physical quantity [2]. Electron density integrated over all space gives total number of electrons N . I.e.:

$$N = \int \rho(\mathbf{r}) d\mathbf{r} \quad (2.13)$$

This indicates that if we know electron density ρ we can construct Hamiltonian H , solve Schrödinger's equation and determine the wave functions and energy eigenvalues.

Kohn-Sham proposed to start with an fictional system where electrons does not interact but that have same density as the real system with interacting electrons [4]. Because density determines position and atomic numbers of the nuclei these quantities are same for both systems. Next energy functional is divided into several terms:

$$E[\rho(\mathbf{r})] = T_{ni}[\rho(\mathbf{r})] + V_{ne}[\rho(\mathbf{r})] + V_{ee}[\rho(\mathbf{r})] + \Delta T[\rho(\mathbf{r})] + \Delta V_{ee}[\rho(\mathbf{r})] \quad (2.14)$$

where individual term represent the kinetic energy of the non-interacting electrons, the nuclear-electron interaction, the classical electron-electron repulsion, the correction to the kinetic energy deriving from the interacting nature of the electrons, and all non-classical corrections to the electron-electron repulsion energy. The last

two terms are usually merged into Exchange-correlation energy (E_{xc}). The density ρ is simply sum of all electron orbital functionals:

$$\rho = \sum_{i=1}^N \langle \chi_i | \chi_i \rangle \quad (2.15)$$

E_{xc} represents not only quantum-mechanics exchange and correlation but also classical self-interaction energy and kinetic energy difference between real and fictional non-interacting system. Because this term is the most difficult to compute modern computational algorithms approximate it by introducing an empirical parameters ε_{xc} based on an experiment. Then E_{xc} is expressed as:

$$E_{xc}[\rho(\mathbf{r})] = \int \rho(\mathbf{r}) \varepsilon_{xc}[\rho(\mathbf{r})] d\mathbf{r} \quad (2.16)$$

The energy density ε_{xc} is a sum of individual exchange and correlation contributions. For example Slater exchange energy density is:

$$\varepsilon_{xc}[\rho(\mathbf{r})] = -\frac{9\alpha}{8} \left(\frac{3}{\pi} \right)^{1/3} \rho^{1/3}(\mathbf{r}) \quad (2.17)$$

Whole DFT calculation process is shown of figure 2.4. It is worth noting that DFT is computationally comparable with HF theory but yields more accurate results. Main disadvantage is that there is no systematic approach to improve convergence to exact solution of Schrödinger's equation.

2.3 Software for computational chemistry

There are many commercial quantum chemistry softwares used in practical applications and in academical spheres. A nice list with essential informations as license type, basis set and included methods can be found in [5]. To illustrate features of a commercial program I choose Crystal. The Crystal was chosen because we have a license at the university so I was able to examine its features. Because my thesis is focused on possibilities of computation acceleration in XCint open source package I describe basic XCint features in this section also.

2.3.1 Commercial: Crystal

Crystal does ab initio calculations in three dimensions (crystals), two dimensions (slabs) and 1 dimension (polymers). It is developed mainly by teams from University of Torino (Italy) and Computational Materials Science Group at the Daresbury Laboratory near Cheshire (England) [6]. Crystal has a rich functionality. Some of included the are:

1. The single particle potential:
 - Hartree-Fock theory, including restricted and unrestricted.
 - Density Functional theory for exchange and correlation.

- Spin Density Functional theory.
2. Algorithms:
 - Parallel processing.
 - Self Consistent Field.
 3. Structural editing:
 - Group symmetry.
 - Deformation of crystallographic cell.
 - Removal and substitution of atoms.
 - Displacement and rotation of atoms.
 - Cluster generation from 3D crystal.
 4. Properties:
 - Electronic charge density maps on 2D or 3D grid.
 - X-ray structure atom factors.
 - Electron momentum distribution.
 - First order density matrix.
 - Electrostatic potential, field and field gradients.
 - Density functional correlation energy.
 - Spontaneous polarization.
 - Piezoelectricity.

All these features are useful for theoretical chemists but for someone who is in development of computational algorithms there is limitation. Whole code is closed for public. Thus these developers must either code from ground zero or find an open-source alternative. Also license cost may be high if all features of the Crystal are not utilized.

2.3.2 Open-source: XCint

XCint is an open-source program that integrates the exchange-correlation (XC) energy E_{xc} and the elements of XC potential V_{xc} . Their derivatives in respect to electric field and geometric perturbations are also computed [7]. Whole project is developed by Radovan Bast from KTH Stockholm. XCint computes whole batch of points at a time. Second speed up for computation is usage of BLAS 3 libraries. More about BLAS is in chapter 3.1. For E_{xc} electron densities are required. They are computed in two steps.

$$X_{kb} = \sum_l D_{kl} \chi_{lb} \quad (2.18)$$

$$\mathbf{n}_b = \sum_k \chi^{bk} X_{kb} \quad (2.19)$$

Then E_{xc} is computed using XCFun library.

$$E_{xc} = \sum_b w_b \epsilon_{xc}(\mathbf{n}_b) \quad (2.20)$$

For the integration grid there are three approaches combined during computation in XCint.

1. Integration grid according to Becke [8].
2. Radial grid according to Lindh, Malmqvist, and Gagliardi [9].
3. Angular grid generated according to Lebedev and Laikov [10].

Radovan Bast put great effort into easy installation and running of XCint. After downloaded whole code can be compiled by running a configuration python script and make command in any Linux environment. There is included an unit test to test proper installation.

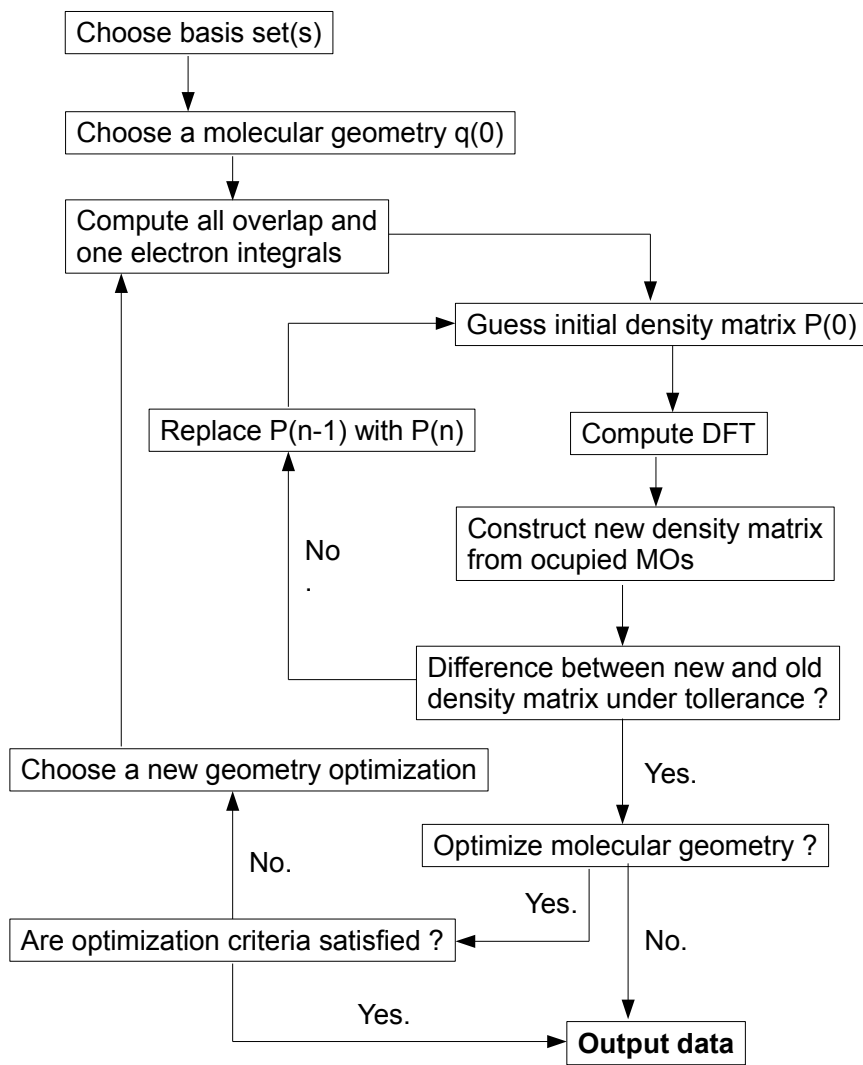


Figure 2.4: DFT calculation.

Chapter 3

Computation acceleration

In my thesis I focus on the computation accelerations. First one is usage of an optimized Basic Linear Algebra Subprogram (BLAS) libraries. Second one is porting part of code on Graphical Processing Units (GPU). For blas routines I use Intel's Math Kernel Library (MKL). For GPU computing I use NVidia's CUDA library. In this chapter I review both approaches. At the end of the chapter is a computation speed comparison of multiplication of two $[N \times N]$ sized matrices.

3.1 Intel MKL

Intel Math Kernel Library (MKL) is a library of highly vectorized and threaded math functions for scientific, engineering and financial applications. Thus using this library greatly increase program performance and development time. The only strong requirement is Intel processor and installed corresponding libraries for compiler. More general technical specifications are in table 3.1 The key areas where MKL library can be used are [11]:

- Linear algebra
- Fast Fourier Transforms
- Vector Math
- Statistics

In my thesis I used Basic Linear Algebra Subprogram (BLAS) functions for vector-matrix, matrix-matrix multiplication and vector data allocation. It is important to note, that BLAS routines are organized in three levels. Symbols α, β represents constants, A, B, C matrices and \mathbf{x}, \mathbf{y} are used for vectors.

- **Level 1:** vector manipulation such as dot product, norm and addition in general form:

$$\mathbf{y} = \alpha \mathbf{x} + \mathbf{y} \tag{3.1}$$

- **Level 2:** matrix and vector operations including general multiplication (GEMV):

$$\mathbf{y} = \alpha A\mathbf{x} + \beta \mathbf{y} \quad (3.2)$$

There is also a solver for equation:

$$T\mathbf{x} = \mathbf{y} \quad (3.3)$$

- **Level 3:** newest part of Intel MKL library includes functions for matrix-matrix operations. As well general matrix-matrix multiplication:

$$C = \alpha AB + \beta C \quad (3.4)$$

Hardware	Intel and compatible processors such as: Intel®Xeon, Intel®Core™ or Intel®Atom.
Operating system	Windows, Linux, OS X.
Development tools and environments	Compatible with compilers from vendors that follow platform standards (e.g. Microsoft, GCC, Intel).
Programming languages	natively supports C/C++ and Fortran.
System requirements	all commonly used compilers and OS distributions are supported.

Table 3.1: Technical specifications for Intel MKL library

3.2 GPU Computing

There was always need for strong computational performance in science and engineering. Today's Graphic Processing Units (GPU) are a valuable option for multiprocessing. They are one of the first common computational kernels that run faster than optimized CPU implementations [12]. This new era of parallel computing would not be possible if there were not developed various APIs that make applications programming easier. Most notable are:

- OpenCL
- CUDA
- OpenMP
- Thread Building Blocks
- OpenACC

I describe two APIs which are nowadays the most common, OpenCL and CUDA. The OpenCL is a hardware independent solution developed and maintained mostly by Khronos group. It is widely used on ATI graphic cards architectures. The CUDA is developed by NVidia group and is NVidia hardware dependent. In table 3.2 is comparison of terms used by both groups to describe similar concept [12]. There are numerous articles about computational performance analysis and comparison between CUDA and OpenCL algorithms, most notably [12] and [13].

NVidia term	ATI term
Scalar core	Stream core
Streaming multiprocessor (SM)	Compute unit (CU)
Shared memory	Local data store (LDS)
Warp	Wavefront
PTX	IL

Table 3.2: Comparison of terms used by ATI and NVIDIA to describe similar concepts.

3.2.1 OpenCL

Open Computing Language is a framework for writing code that executes on heterogeneous platforms consisting of host CPU and any attached OpenCL device. This device may share memory with host and typically has different instruction set [13]. OpenCL includes language based on C99 standard for programming the compute devices and API for control the platform and execute programs on these devices. The key programming features are functions for enumerating available target devices (independent of their type), managing data transfer on these devices and compiling and executing OpenCL programs. Also run-time compilation is supported. This feature eliminates dependency on target hardware and software architecture. It is convenient to a programmer because he is not limited for a specific scenario. Even his code can use devices which are unavailable during development.

OpenCL guarantees hardware portability, but it is not guaranteed that a particular kernel will achieve peak performance on every architecture. The term device mean CPU, GPU or various types of accelerators. In OpenCL there are defined four types of memory [13]. Large high-latency *global* which can be accessed from anywhere, small low-latency read-only *constant* memory. Shared *local* memory is accessible from all PEs on same compute unit and *private* also known as device registers are accessible only within the PE. An application can query device to determine its properties. Therefor available compute units and memory can be used effectively.

3.2.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing programming model and platform developed and maintained by NVIDIA corporation. Unlike OpenCL it is designed only for NVIDIA hardware but programming in this language is simpler. First graphic card build with a CUDA architecture was released in 2006 as GeForce 8800GX [14].

This included several new components designed for GPU computing. Thanks to this improvements CUDA removed many limitations that prevented GPUs to be usefull for a general-purpose computing. Most notably unified shader pipeline that allow each ALU to be managed by a program. Additional ALUs are build to support IEEE standardized single-precision floating-point arithmetics and use instruction set build for general computation instead of standard graphics. All

execution units on the device are allowed arbitrary read and write access to memory and software-managed cache *shared* memory.

For programming the CUDA C language is used [14]. It is the industrial standard C language with few additional keywords to harness maximum from NVIDIA's CUDA architecture. It is also the first language developed by GPU company for general purpose computing on graphic cards. There are also specialized hardware drivers to utilize CUDA architectures massive computing power. Users are not required to solve their problem as a computer graphic task nor they are required to have a knowledge about OpenGL or DirectX.

An additional drawback of CUDA C is usage of C++ compiler. Therefore valid C, but invalid C++ code can be fail to compile. There is also no support for exception handling. While OpenGL has access to registered CUDA memory, it is not possible the other way.

Main advantages of using CUDA are:

1. Code can be read from any address in memory,
2. Unified memory,
3. Usage of shared memory,
4. Faster downloads and readbacks to and from the GPU,
5. Full support for integer and bitwise operations.

3.3 Computation study

In this section I will show you trends for computational speed for sequential code, blas code and GPU code. Before getting to the results of the experiment it is important to note two facts. First not every procedure can be parallelized. Operations within an algorithm can be parallelized if and only if they are independent (i.e. they can be executed in any order). In my thesis I focus on Single Instruction Multiple Data (SIMD) parallelization technique. One kernel is executed simultaneously on multiple chunks of data as illustrated on figure 3.2. Second fact is that copying data from host to device memory takes processing time. While parallelization on multiple CPUs or GPUs faster whole computation, data transfer on the other hands slows down. Thus it is important to consider wisely which part of code can be ported. Often the gain is visible after porting and running the code and in more complex computations.

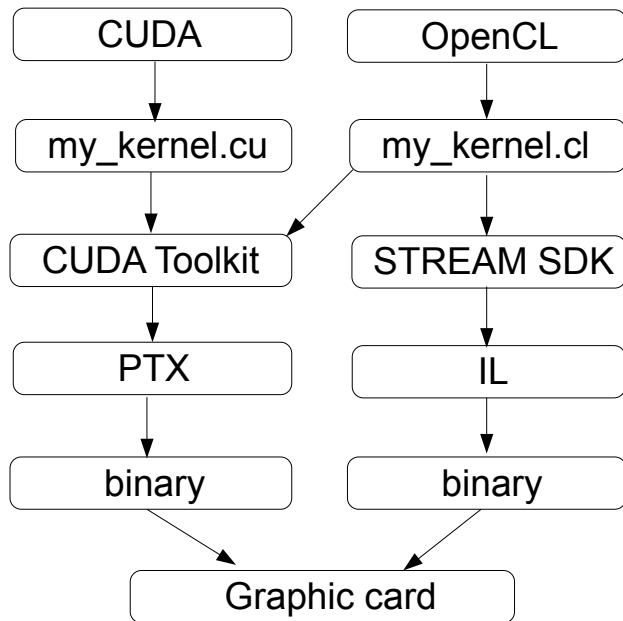


Figure 3.1: Compilation steps

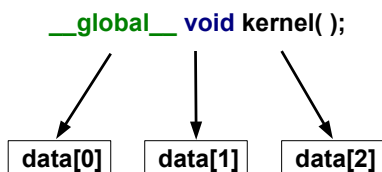


Figure 3.2: SIMD principle

In the table 5.2 are results from the experiment. All times are average of 10 consequential runs and in milliseconds (ms). There is included also data copying time in cuda columns. From measured times it is clear that data transfer takes majority of computation time in CUDA case. Despite that CUDA is still faster than pure sequential code. The fastest are the fully optimized BLAS3 functions from the Intel MKL library both on CPU and GPU processors. A matrix multiplication has complexity of $\mathcal{O}(N^3)$. If multiple processing units are working parallel it is reduces by number of units. In case of vectorized routine time reduction is even

greater. Therefore usage of these two techniques is of practical significance and this thesis aims are meaningful.

N	sequential	blas lvl 3	cuda gpu	cuda blas3
100	6	0.07	2	0
200	49	0.27	23	1
400	434	1.45	200	2
600	1584	5.13	665	4
800	4007	11.37	1621	7
1000	6655	21.82	3095	23

Table 3.3: Results of experiment.

Chapter 4

Grid Calculation

In this chapter I describe code I wrote for my diploma thesis. Whole code can be downloaded and run from my Git repository [15]. In following sections I describe whole computation step-by-step. A pseudocode, a part of code or an example output is provided if needed. Because I will reference to individual functions I provide structure of project as it can be downloaded. Functions are in *italics*. Important keywords as are marked **bold**. Folders are CAPITAL. Note that I list only the most important functions. Whole code was tested on Scientific Linux 6.4 operating system and instruction are provided only for Linux distributions. Because this project was created for purposes of scientific computing Windows and OS X distribution are not available.

- INPUT
 - NEON-DZ
 - * basis.txt
 - * dmat.txt
 - * grid.txt
 - BENZENE-DZ
 - * basis.txt
 - * dmat.txt
 - * grid.txt
- SRC
 - atom.h
 - atom.cpp
 - ccalc.h
 - ccalc.cu
 - * ***void*** *calcDensityCuda(int)*
 - * ***__global__ void*** *calcDens(int pts, int noAOs, double *cd_DM,*
*double *cd_gVal, double *cd_wght, double *cd_gDns)*

- grid.h
- grid.cpp
 - * protected: **int** errorCode
 - * protected: **struct atom** gridAtom
 - * public: **void** calcGrid(int, int)
 - * public: **void** printInfoGrid()
 - * public: **void** printFullGrid()
 - * protected: **double** getR(int)
 - * protected: **double** getValue(int)
 - * protected: **void** calcDensity()
 - * protected: **void** calcDensityScr()
 - * protected: **void** calcDensityBatch(int)
- **int** main
- makefile
- readme.txt

4.1 Compilation

For sequential code with utilizing Intel®MKL library compilation is done by typing command *make* in repository directory. Intel compiler and libraries must be installed on your Linux system. On university server cluster they are installed but needs to be loaded by command *module load intel/composer_xe_2013*. If Intel MKL libraries are not available user needs to change compiler option in makefile and set MKL to 0 in grid.cpp file. Cuda version of code can be used by setting CUDA to 1 and MKL to 0 in *grid.cpp* file and *main.cpp* file and running compilation command provided in readme.txt file.

4.2 Input data parsing

Three data files are provided. The first file *basis.txt* holding information about the coefficients and the exponents for the Gaussian basis sets and Cartesian positions of orbital centers. The second *dmat.txt* is carrying density matrix. In the third *grid.txt* is grid generated according to Beck [8]. There is also an option for radial grid implemented, but for in the experiment I will use "Becke" grid from text file. Names for all three files with number of atomic orbitals and contracted functions are provided at the beginning of the **int** main() function. Functions from grid.cpp take care of loading data and initializing variables.

4.3 Atomic orbital value calculation

In this step value for every atomic orbital χ_i is computed according to equation 4.1. With Gaussian being basis function as was mentioned in chapter 2.2.5.

$$\chi_j = \sum_{i=0}^N \alpha_i \exp^{\beta_i R^2} \quad (4.1)$$

Coefficients α_i s and β_i s are provided from input data files. R^2 is squared Cartesian distance.

$$R^2 = (x_a - x_p)^2 + (y_a - y_p)^2 + (z_a - z_p)^2 \quad (4.2)$$

x_a denotes x coordinate for atomic orbital center and x_p denotes grid point x coordinate. This step is performed at each grid point for each atomic orbital. Pseudocode for algorithm is provided in 1. Whole procedure consists of two private

Algorithm 1 Calculate atomic orbitals.

```

1: for all grid points do
2:   for all atomic orbitals do
3:      $value := 0.0$ 
4:     compute  $R^2$ 
5:     for  $i \leq n_{fnc}$  do
6:        $value += \alpha_i * \exp(\beta_i * R^2)$ 
7:     end for
8:     current el. orbital  $\leftarrow value$ 
9:   end for
10: end for
```

functions which are called by main *calcGrid(int, int)* during computation. The **double** *getR2(int)* for calculating R^2 and the **double** *getValue(int, double)* to compute value in each step. This step is done sequentially. Because it consists of a few mathematical operations between a large amount of data parallelization would not have desired effect. Most of processing time would be spent on data transfer between cores and host.

4.4 Calculate electron density

This is the most interesting part of the code and thus the part which was mainly focused in my diploma thesis. In fact it is the bottleneck of whole computation. In chapter 2.2.7 was mentioned that the physical quantity used for calculation is electron density. And in equation 2.15 that electron density (E) is calculated from atomic orbitals χ multiplied by Density Matrix D . Below is rewritten equation 2.15 in more mathematical form with index p denoting a grid point.

$$E_p = \chi_p D \chi_p \quad (4.3)$$

Currently there are four methods supported and fifth under development with

Algorithm 2 Calculate electron density.

```
1: N := 0.0
2: for all grid points do
3:   E =  $\chi D \chi$ 
4:   N += E
5: end for
```

expectation to be finished until end of May.

1. CPU sequential
2. CPU screening
3. CPU batch
4. GPU sequential
5. GPU batch is under development.

4.4.1 CPU sequential

The simplest way how to calculate product of atomic orbitals and density matrix. Because matrix D is diagonally symmetric (i.e. $D[i][j] = D[j][i]$ if $i \neq j$) a simplification can be used. It greatly reduces computational complexity. Instead of complexity scaling by $\mathcal{O}(N^3)$ the factor is only $\mathcal{O}(N^2)$. In my code it is done by private function **void** *calcDensity()*.

Algorithm 3 Sequential computation of ED

```
1: E := 0.0
2: for  $k \leq n_{ao}$  do
3:   for  $l < k$  do
4:     E +=  $2 * \chi_k * D[k][l] * \chi_l$ 
5:   end for
6:   E +=  $\chi_k * D[k][k] * \chi_k$ 
7: end for
```

4.4.2 CPU screening

In 4.1 is depicted sample output for Neon atom. Data are in order of $[x, y, z]$ coordinate and $[ao_1, ao_2, \dots, ao_{15}]$. As you can see many values for atomic orbitals are zeros, or very small close to zero. Though for smaller systems this may seem as unimportant, but in larger system computing these zeros is significant waste of time. Therefore a screening procedure may be used. First only left side of 4.3 is computed and if value is below a threshold instead of computing the rest, zero is taken as the result. In my code I took advantage of C language's pointers and Blas2 (section 3.1) vector-matrix arithmetics from Intel MKL library. Instead of working with memory directly, I use a pointer array and dereference positions


```

3xminarikj@one:~/NeverWhere/GridCalculation
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$ head out.txt
== SUCCESS ==
calculation time: 40.00000 ms
no Points: 16496
no AOs: 15
no electrons: 4.999998
9.96218e-07 0 0 16.7749 -3.93649 0.415422 9.29626e-06 0 0 4.96904e-07 0 0 1.12618e-11 0 0 0 0 4.53488e-17
-9.96218e-07 0 0 16.7749 -3.93649 0.415422 -9.29626e-06 0 0 -4.96904e-07 0 0 1.12618e-11 0 0 0 -0 0 4.53488e-17
0 9.96218e-07 0 16.7749 -3.93649 0.415422 0 9.29626e-06 0 0 4.96904e-07 0 0 1.12618e-11 0 0 0 0 4.53488e-17
0 -9.96218e-07 0 16.7749 -3.93649 0.415422 0 -9.29626e-06 0 0 -4.96904e-07 0 0 1.12618e-11 0 0 0 -0 4.53488e-17
0 0 9.96218e-07 16.7749 -3.93649 0.415422 0 0 9.29626e-06 0 0 4.96904e-07 0 0 1.12618e-11 0 0 0 4.53488e-17
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$
[3xminarikj@one GridCalculation]$

```

Figure 4.1: Atomic orbital values for Neon

that provide product below threshold. Threshold is defined as global static variable and is set to 10^{-12} . Algorithm is provided below, pointers are denoted by a \star . For implementation in code look for function *void calcDensityScr()*.

Algorithm 4 Computation of E with screening

```

1: E := 0.0
2:  $\chi^* \leftarrow \chi$ 
3:  $X^* \leftarrow \chi^D$ 
4: for  $X_i^*$  do
5:   if  $X_i^* < \text{threshold}$  then
6:      $X_i^* = \text{NULL}$ 
7:      $\chi_i^* = \text{NULL}$ 
8:   end if
9:    $E \leftarrow X^* \times \chi^*$ 
10: end for

```

4.4.3 CPU batch

Why to compute just one point at a time if there is a way to compute whole batch. If χ_p vectors are put one below another a matrix is constructed. Then there is a possibility to utilized matrix-matrix Blas3 routines. Lets denote a batch of n grid

points by b and construct a matrix C_b from n consecutive vectors χ_p .

$$C_b = \begin{bmatrix} \chi_p \\ \chi_{p+1} \\ \chi_{p+2} \\ \vdots \\ \chi_{p+n-1} \end{bmatrix} \quad (4.4)$$

Then equation 4.3 is rewritten to:

$$E_b = C_b \times D \cdot C_b \quad (4.5)$$

where \times denotes classical matrix-matrix product and \cdot denotes vector-wise matrix product. Whole process is described in algorithm below and implemented in function ***void calcDensityBatch(int)***.

Algorithm 5 Computation of E from batch of points.

- 1: $E_b := \text{zeros}[b \times n_{ao}]$
 - 2: Construct C_b
 - 3: $E_b \leftarrow C_b \times D$
 - 4: $Eb := E_b \cdot C_b$
 - 5: Distribute E_b to E_p .
-

4.4.4 GPU sequential

Because grid points are computed independently of each other they are natural choice for parallelization using Single Instruction Multiple Data (SIMD) principle. This procedure is done in following three steps:

1. Copy required data to GPU memory: Atomic orbital values, density matrix.
2. Compute E using algorithm 3.
3. Copy E to host memory.

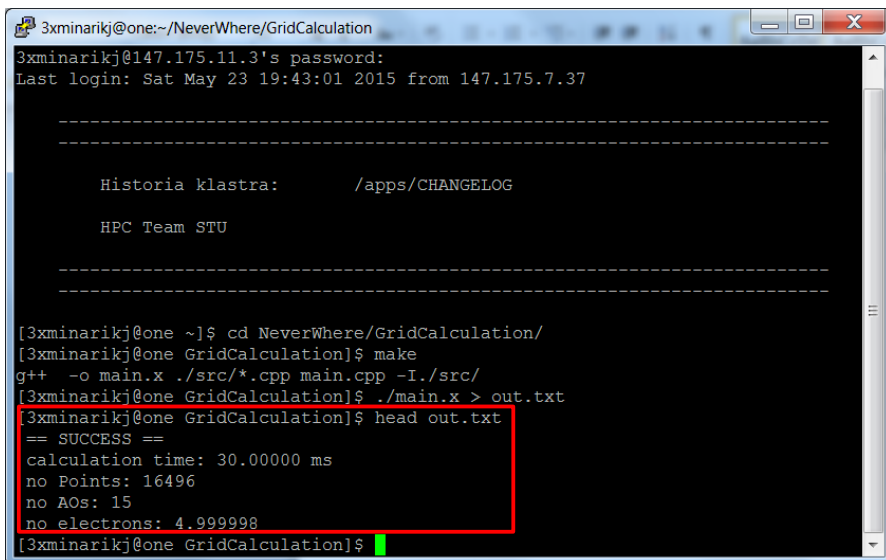
Because algorithm 3 has complexity of $\mathcal{O}(N^2)$ it is obvious that effectiveness of this method depends on number of atomic orbitals. Basically speed from using multiple cores must be greater than time spend on data movement.

4.5 Exchange-Correlation Energy

Exchange correlation energy is computed according to equations 2.16 and 2.17. Element df is weight of grid point. Because it is a fast computation this step is done on CPU for all four implemented methods.

4.6 Print output

Two functions are provided for user to print information about computation. The function ***void printInfoGrid()*** provides all important information about computation as status, number of electrons, total computation time, number of atomic orbitals and number of grid points. If user is interested in detailed information about electron density on individual grid points he can call function ***void printFullGrid()***. But note that there are thousands of points in grid, therefore this output is quite a long. And example short output is provided in figure 4.2. For developing purposes there is a compilation option `VERBOSITY` which prints additional info about computation.



```
3xminarikj@one:~/NeverWhere/GridCalculation
3xminarikj@147.175.11.3's password:
Last login: Sat May 23 19:43:01 2015 from 147.175.7.37

-----
Historia klastra:      /apps/CHANGELOG
HPC Team STU
-----

[3xminarikj@one ~]$ cd NeverWhere/GridCalculation/
[3xminarikj@one GridCalculation]$ make
g++ -o main.x ./src/*.cpp main.cpp -I./src/
[3xminarikj@one GridCalculation]$ ./main.x > out.txt
[3xminarikj@one GridCalculation]$ head out.txt
== SUCCESS ==
calculation time: 30.00000 ms
no Points: 16496
no AOs: 15
no electrons: 4.999998
[3xminarikj@one GridCalculation]$
```

Figure 4.2: Output from function *printInfoGrid()*

Chapter 5

Conclusion

The computational case study is carried on the Neon atom. It has 6 atomic orbitals occupied by 10 electrons and 25 contracted functions for GTO basis set. All input files for Neon are in the input folder of the Git repository [15]. The Becke grid containing 16496 points is in grid.txt file.

The study is run on a computational cluster of the Slovak University of Technology in Bratislava with parameters in table 5.1. Measured processing times are in table 5.2.

parameter	value
OS	Scientific Linux 6.4.
CPU	Intel Xeon X5670 2.93 GHz
RAM	48GB
GPU	NVIDIA Tesla M2050 448 cuda cores

Table 5.1: Parameters of STU BA computational cluster.

method	computational time [ms]
CPU Sequential	8.3
CPU Screening	6.5
CPU Batch	7.2
GPU Sequential	8.0

Table 5.2: Measured processing times.

The results provide an interesting conclusion. Each improved code is faster than pure sequential one. For molecule of this small size, vectorized and optimized functions from Math Kernel Library outran each other code by a significant amount (at least 10%). From the difference between screening and batch technique we can see that the construction of a matrix (done by reallocation of memory) are a slow down for a molecule of small size as Neon. The CUDA GPU code has major slow down cause by copying of data for all grids points. But parallel usage of 448 cores manages successful trade-off in terms of decreased overall computational time.

Thus rewriting traditional CPU code to new GPU code is a viable step forward. As was shown in motivational case at the end of chapter 4 in larger matrices this gain would be bigger. Therefor in case of a larger molecule utilizing GPUs would be a great benefit for computational chemists. Another scored points for CUDA will be with Blas3 code on GPUs which is expected to be finished in future weeks. In the future we can expect further improvement of graphic processing units and APIs for their programming. Especially from the NVidia corporation which is putting great effort to propagate their cards in scientific communities.

Resumé

V mojej práci sa venujem využitiu grafických kariet (GPU) na kvantovo-chemické výpočty. Aby som ukázal ich prínos naprogramoval som výpočet výmenno-korelačnej energie (E_{xc}) v programovacom jazyku C++. Moj program obsahuje štyri rôzne metódy na výpočet elektrónovej hustoty, ktorá je hlavným parametrom finálneho výpočtu E_{xc} . Menovite:

1. CPU sekvenčne
2. CPU s využitím screeningu
3. CPU po skupine bodov
4. GPU sekvenčne

Druhá a tretia metóda navyše využíva vektorizované funkcie z Math Kernel Library od spoločnosti Intel na ich procesory.

Celá práca je štrukturovaná aby čitateľ získal prehľad o použitých pojmoch a metódach z počítačovej chémie, ktoré sú vysvetlené v kapitole 2. V kapitole 3. sa venujem MKL knižnici od Intelu a dvom najpoužívanejším rozhraniám na programovanie grafických kariet CUDA a OpenCL. V tejto kapitole sa nachádza aj zhrnutie výhod oboch rozhraní.

Kedže pri použití GPU treba kopírovať všetky potrebné dáta do pamäte karty dochádza k spomaleniu celého výpočtu. Preto treba zvážiť ktorá časť aplikácie pobeží mimo hlavnej procesorovej jednotky. Zanalyzoval som jednotlivé časti výpočtu E_{xc} a preniesol len najkritickejšiu časť na GPU. Záver práce tvorí príklad výpočtu E_{xc} na atóme Neónu a vyhodnotenie výpočtového času všetkých štyroch metód. Ukázal som, že aj jednoduché upravenie pôvodného CPU kódu na GPU kód priniesie zlepšenie v celkovom výpočtovom čase, aj obrovskému množstvu prenesených dát. V budúcnosti môžeme očakávať len ďalší pokrok vo vývoji metód programovania GPU a čoraz väčšie množstvo vedeckých výpočtov na grafických kartách.

Bibliography

- [1] Frank Jensen. *Introduction to Computational Chemistry*. John Wiley & Sons, Ltd, West Sussex, England, 2007.
- [2] Christopher J. Cramer. *Essentials of Computational Chemistry*. John Wiley & Sons, Ltd, West Sussex, England, 2004.
- [3] Andrew S. Ichimura. *Computational chemistry lectures*. California Institute of Technology, California, USA, 2004.
- [4] Walter Kohn and Lu Jeu Sham. Self-consistent equations including exchange and correlation effects. *Physical Review*, 140:1133–1138, 1965.
- [5] Wikipedia. *List of quantum chemistry and solid-state physics software*. http://en.wikipedia.org/wiki/List_of_quantum_chemistry_and_solid-state_physics_software, 2015.
- [6] R. Dovesi, V.R. Saunders, C. Roetti, R. Orlando, C. M. Zicovich-Wilson, F. Pascale, and B. Civalleri and. *Crystal 06 User’s Manual*. University of Torino, Torino, Italy, 2006.
- [7] Radovan Bast. *XCint documentation*. <http://xcint.readthedocs.org/en/latest/purpose.html>, 2010.
- [8] A.D. Becke. A multicenter numerical integration scheme for polyatomic molecules. *The Journal of Chemical Physics*, 88, 1988.
- [9] Roland Lindh, Per Åke Malmqvist, and Laura Gagliardi. Molecular integrals by numerical quadrature. i. radial integration. *Theoretical Chemistry Accounts*, 106:178–187, 2001.
- [10] Lebedev and Laikov. A quadrature formula for the sphere of the 131st algebraic order of accuracy. *Russian Academy of Sciences Doklady Mathematics*, 59:477–481, 1999.
- [11] Intel Corporation. *Intel MKL official website*. <https://software.intel.com/en-us/intel-mkl>, 2015.
- [12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarr. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38:391–407, 2012.

- [13] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–72, 2011.
- [14] Jason Sanders and Edward Kandrod. *CUDA by Example*. Addison - Wesley, Boston, USA, 2011.
- [15] Jan Minarik. *NeverWhere* *Git repository*.
<https://github.com/JanoMinarik/NeverWhere>, 2014.