Institute of Computer Engineering and Computer Architecture
Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Pfaffenwaldring 47, 70569 Stuttgart

Master Project Nr. 3221

Implementing Density Functional Theory (DFT) Methods on Many-core GPGPU Accelerators

by

Bishwajit Mohan GOSSWAMI

M S C T H E S I S

in partial fulfillment of the requirements
for the degree of **Master of Science**

Supervisors :	Dipl.-Inform. Claus BRAUN
Examiner :	Prof. Dr. rer. nat. habil. Hans-Joachim WUNDERLICH
Start Date :	May 01, 2011
Submission Date :	November 14, 2011
CR Classification :	D.1.3, D.3.2, D.4.8, G.1.0, I.6.8

Universität Stuttgart

Abstract

Density Functional Theory (DFT) is one of the most widely used quantum mechanical methods for calculations of the electronic structure of molecules and surfaces, which achieves an excellent balance of accuracy and computational cost. However, for large molecular systems with few hundred atoms, the computational costs are become very high. Therefore, there is a fast growing demand for much more efficient implementations to utilize DFT for macro molecules. General Purpose Graphics Processors (GPUs) are highly parallel, multi-threaded, many-core processors with tremendous computational capability, which out-paces CPUs in terms of floating-point performance. They are particularly focused for computation intensive and highly data-parallel computations. This thesis will introduce the scope of fine grained parallelism with highly data-parallel GPU implementations of several algorithmic parts of DFT. Furthermore, experimental results and benchmarks will be presented in comparison with a current state of art DFT implementation (Molpro).

Keywords: Parallel Architecture, Parallel Algorithms, Many-core architecture, GPGPU, GPU, CUDA, Density Functional Theory (DFT), Molpro, Quantum Chemistry

Acknowledgments

This Master thesis and my degree would not have been possible without the support from many people. I would like to take this opportunity to express my gratitude to my mother **Bakul Banik**, my father **Bijoy Gosswami**, and my sister **Tapashi Gosswami** for their enormous support, patience, blessings throughout the whole period of my study in Stuttgart, and for always being with me. I am grateful to **Prof. Dr. Hans-Joachim Wunderlich** for giving me the opportunity to work on this project in his department. I am heartily thankful to **Dipl.-Inform. Claus Braun** for persevering with me as my supervisor and for his motivation and help through out the time of this research. I offer my regards to **Prof. Dr. Hans-Joachim Werner** for his valuable advice and guidance. He has made available his support in a number of ways. I am deeply indebted to **Dr. Toru Shiozaki** and **Kamruzzaman Tupa** for their valuable suggestions. I am particularly grateful to **Sipu Talukder** for her support, encouragements, dedication and patience throughout the thesis, as well as my whole study periods.

Contents

List of Figures

List of Tables

List of Abbreviations

AO	Atomic Orbital
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
BLYP	Becke-Lee-Yang-Parr
CATS	ClearSpeed Accelerated Tera-scale System
CGTO	Contracted Gaussian Type Orbitals
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DF-KS	Density Fitting Kohn Sham
DFMP2	Density Fitting Mϕller - Plesset Perturbations
DFT	Density Functional Theory
ERI	Electron Repulsion Integrals
FLOP	Floating Point Operations
GGA	Generalized Gradient Approximations
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
GTO	Gaussian Type Orbitals
HF	Hartree Fock
HPC	High Performance Computing
LCAO	Linear Combination of Atomic Orbital
LDA	Local Density Approximation
MO	Molecular Orbital
MPI	Message Passing Interface
NVCC	NVIDIA C Compiler
RDM	Reduced Density Matrix

SCF	Self Consistent Field
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
TFD	Thomas-Fermi-Dirac

Chapter 1

Introduction

Density Functional Theory (DFT) is an approach to describe the quantum behavior of atoms and molecules in settings of practical value. It is a well established quantum mechanical method for electronic structure calculations for molecules. Principally, in electronic structure calculations, a molecular system is described by a set of functions that depends on the coordinates of all the particles in the system. This set of functions is known as the wave function in quantum mechanics. The wave function defines the coordinates of the nuclei and the orbiting electrons. The energy is calculated from the wave functions through quantum mechanical operations and from the solution of these wave equations. The complex nature of the wave functions makes the evaluation of the full system very complex and highly computational expensive even for very small molecules. The Kohn-Sham formalism of DFT approximates this ab-initio approach and proposes electron density instead of the electronic coordinates wave function. The approximation simplifies the model of the electronic system and allows DFT to provide an excellent balance between computational accuracy and cost. Over the last years, this approach has rapidly grown as the cutting edge of quantum mechanical theory that is used regularly by large numbers of researchers in chemistry, physics, materials science and other disciplines. However, despite the approximate nature of the DFT, the computational demand becomes still very high for systems with few hundred atoms. For such a macromolecular system with a large number of atoms, the computational overhead now becomes a serious limitation for the considered system size.

In recent years, GPGPU (General Purpose Graphics Processing Unit) accelerators have raised the interest for massively parallel computing in the field of scientific calculations. This technology has rapidly evolved from graphics processing to programmable parallel streaming processing. These accelerators are specialized for computationally intensive and for highly data parallel computations. As a result, GPU accelerators are potentially several times faster than the CPUs (Central Processing Units) in data-parallel applications. However the architectures of GPUs are more centric to data processing rather than to instruction flow control and data caching. The excellent price to performance ratio and the low energy consumption per FLOP makes the GPGPU an attractive way to accelerate highly data parallel scientific computations. However, these GPU accelerators have some special characteristics. Often they are optimized in such a way that the performance has the higher priority rather than numerical accuracy. Most of the GPUs are only support single precision floating point numbers with very high performance. But only single precision accuracy is not sufficient in the quantum chemical simulations. Recently, the NVIDIA Fermi architecture supports double precision arithmetics, but with the expense of computational performance. Moreover, the amount of memory resources (registers, on-chip memory) per core is very small compared to CPU. In order to achieve high performance from GPU accelerators, the application needs to be very fine grained with highly data parallel algorithms, where each light weight thread can do the calculations using the ample resources allocated for it.

Applications with high arithmetic intensity are particularly well suited for GPU accelerators. Recently a lot of scientific application have been ported to the GPU. Thus there is an opportunity for high-throughput GPUs to play an active role on DFT calculations. In the domain of electronic structure calculation, only few works have been done in the context of GPGPU accelerators. For instance, the calculation of the exchange correlation term in a Gaussian-based DFT code has been implemented [1]. The 4-index electron repulsion integrals (ERI), which is one of the computational bottlenecks in the electronic structure method, was ported to GPUs [2]. A many-core hybrid CPU-GPU architecture of DFT was implemented [3], where the wavelet based transformation and convolution were used. Most of these implementations were performed with single precision arithmetic and with single GPU accelerator in single core system.

In this thesis, the scope of fine grained parallelism is analyzed for the integrated DFT module in the *Molpro* quantum chemistry package. The state of the art multi-threaded DFT architecture in Molpro forms the basis of the GPU mappings. A multi-GPU DFT architecture is proposed and implemented which can run on hybrid CPU-GPU clusters. Two main computationally expensive bottlenecks are identified and ported to the state of art NVIDIA Fermi GPU architecture, which supports double precision arithmetics. Additionally, a set of multi-GPU Fortran wrappers is implemented in this thesis to accelerate the density fitted Mϕller-Plesset Perturbation (DF-MP2) calculation to GPU.

The outline of this thesis is the following:

We will start in chapter 2 by providing the basic theoretical and mathematical background of the quantum chemistry which will help to understand the concepts of the DFT theory in more detail. In chapter 3, the state of the art is presented which gives an overview of the previous work done in this area. Chapter 4 presents the GPU architecture with the corresponding programming model for this device. Chapter 5 presents the architecture of the integrated DFT module within the Molpro quantum chemistry package. This DFT module forms the basis of the GPU mappings developed in this thesis. Chapter 6 explains the implementation of the two computationally most expensive DFT algorithmic parts in a multi-GPU environment. Chapter 7 provides the details of the multi-GPU Fortran wrapper implemented to accelerate the density fitted Mϕller-Plesset

Perturbation (DF-MP2) calculation on the GPU. Chapter 8 presents the results and performance evaluation of the ported GPU code and a brief discussion about the outcomes. Chapter 9 summarizes the thesis and provides the outlook for further improvements.

Chapter 2

Density Functional Theory

Contents

2.1 Introduction

Density functional theory (DFT) is now one of the most dominant computational procedures for molecular electronic structure calculations. The basic idea behind this method is that the energy can be calculated for any electronic system in terms of electron probability density [4]. Since electrons are very light particles they need to be described by a quantum mechanical approach. Thus basic knowledge of *quantum mechanics* is required to understand the procedure. In this chapter, we will go through some relevant definition of basic quantum mechanics, the quantum operators and their mathematical interpretations first. The remaining section about the electronic structures calculation is built on these quantum notations.

2.2 Basics of Quantum Mechanics

Atoms and molecules consist of positively charged nuclei and negatively charged electrons. Different atoms have different nuclear charges. Electrons carry a charge $-e$ (elementary charge) and the nuclear charges are multiples of e, i.e., Z_ke, where Z_k is the atomic number. So, in neutral molecules the number of electrons equals the sum of the atomic numbers of all nuclei in the molecule. For example, a diatomic molecule like, O_2 has 16 electrons [5]. Electrons are very light weight particles and they can not be described correctly by classical mechanics. So they have to be treated by quantum mechanics. Quantum Mechanics explains the behavior of matter and its interactions with *energy* on atomic scales. It provides the mathematical description of the *wave-particle* duality of matter and energy. The introduction of quantum mechanics led to a revolution in science and is regarded as the most profound scientific breakthrough of all time. In this section some very basic facts and concepts of quantum mechanics are summarized.

2.2.1 Terms and Notations

Operator An *observable* defines a variable that can be measured. An *operator* is a symbol for an instruction to carry out an operation on a function [4]. For example, one typical quantum operator is a differentiation with respect to an observable x and is represented by $\frac{d}{dx}$.

Linear Operator A *linear operator*, $\hat{\Omega}$ is of the form of

$$\hat{\Omega}(af) = a\hat{\Omega}f \tag{2.1}$$

where a is a constant and f is a function [4]. For instance, multiplication, differentiation, integrations are all linear operators. Most of the operators in quantum mechanics are linear.

Eigenfunctions and Eigenvalues A function f is an *eigenfunction* of an operator $\hat{\Omega}$ if it satisfies an equation of the form

$$\hat{\Omega}f = \omega f \tag{2.2}$$

where ω is a constant. This kind of equations is also called an *eigenvalue equation* [4]. In general, the outcome of an operation is the same function multiplied by a constant. The constant ω in an *eigenvalue equation* is known as the *eigenvalue* of the operator $\hat{\Omega}$ [4]. For example, the function e^{ax} is an eigenfunction of the operator $\frac{d}{dx}$, as

$$\frac{d}{dx}e^{ax} = ae^{ax} \tag{2.3}$$

where a is the constant and called eigenvalue of the operator $\frac{d}{dx}$.

A general function can be constructed in terms of all the *eigenfunctions* of an *operator*. The *eigenfunctions* used to construct a general function are called *basis functions*. It is also expressed as the *linear combinations* of *basis functions* to construct a general function [4]. For example,

$$g = \sum_n c_n f_n \tag{2.4}$$

where c_n are coefficients and the sum is over a complete set of basis functions f_n.

Commutation In general, *commutation* do not hold for operators. The outcome depends on the order of successive operations. That means, $\hat{A}\hat{B} \neq \hat{B}\hat{A}$. $\hat{A}\hat{B} - \hat{B}\hat{A}$ is known as the *commutator* of $\hat{A}$ and $\hat{B}$ and is represented by $[\hat{A}, \hat{B}]$ [4].

Representations All quantum operators can be constructed from the position operator $\hat{\zeta}$ and the linear momentum operator $\hat{p}_\zeta$, where $\hat{\zeta} = \{\hat{x}, \hat{y}, \hat{z}\}$ and $\hat{p}_\zeta = \{\hat{p}_x, \hat{p}_y, \hat{p}_z\}$. These must obey the commutation relations

$$[\hat{x}, \hat{p}_x]\psi(x) = i\hbar\psi(x) \text{ for any functions } \psi(x) \tag{2.5}$$

This does not define the operators uniquely. Different representations are possible. There are two common representations, one is the *position representation*, where the position operator is represented by multiplication by $\hat{x}$ (with the coordinates specified) and the *linear momentum*, $\hat{p}_x$, parallel to $\hat{x}$ is represented by differentiation with respect to $\hat{x}$ [4].

$$\hat{x} \to \hat{x}\times \tag{2.6}$$

$$\hat{p}_x \to \frac{\hbar}{i}\frac{\partial}{\partial x} \tag{2.7}$$

The other representation is the *momentum representation*, where the *linear momentum* parallel to $\hat{x}$ is represented by the multiplication by $\hat{p}_x$ and the *position operator* is represented by differentiation with respect to $\hat{p}_x$.

$$\hat{x} \to -\frac{\hbar}{i}\frac{\partial}{\partial x} \tag{2.8}$$

$$\hat{p}_x \to \hat{p}_x\times \tag{2.9}$$

Operator Constructions Operators can be constructed from the operators of position and momentum for any observable [4]. For example, the *kinetic energy* is related with the *linear momentum* by $\hat{T} = \frac{\hat{p}^2}{2m}$, where m is the mass of the particle and $\hat{p}^2$ is the operator that is applied two times in series. So in three dimensions, the *kinetic energy* operator in position representation is

$$\hat{T} = \frac{\hat{p}^2}{2m} = -\frac{\hbar^2}{2m}\left\{\frac{\partial^2}{\partial x} + \frac{\partial^2}{\partial y} + \frac{\partial^2}{\partial z}\right\} = -\frac{\hbar^2}{2m}\nabla^2 \tag{2.10}$$

the operator ∇^2 is called the *Laplacian*, which is the sum of all the three second derivatives.

Hamiltonian Operator The *Hamiltonian operator* describes the total energy of a system and is denoted by $\hat{H}$ [4].

$$\hat{H} = \hat{T} + \hat{V} \tag{2.11}$$

where $\hat{T}$ is the *kinetic energy*, explained above (2.10), and $\hat{V}$ denotes the *potential energy operator*. The potential energy per electron is defined by

$$\hat{V} = -\frac{Ze^2}{4\pi\varepsilon_0 r} \tag{2.12}$$

with atomic number Z, the electron charge $-e$ and the distance between the nucleus and electron, r.

Integral over Operators The integral over an operator $\hat{\Omega}$ has the form

$$I = \int f_m^* \hat{\Omega} f_n d\tau \tag{2.13}$$

where f_m^* is the complex conjugate of f_m and $d\tau$ is the volume element [4]. The integration is over the whole space ($-\infty$ to ∞).

Overlap Integrals The above integral (2.13) is called *overlap integral* if the operator $\hat{\Omega}$ is defined as the multiplication by 1 [4].

$$S = \int f_m^* f_n d\tau \tag{2.14}$$

Normalization Integral It is a special case of *overlap integral* (2.14) [4], where $m = n$. A function f_m is said to be normalized if

$$\int f_m^* f_m d\tau = 1 \tag{2.15}$$

Dirac Brackets For simplicity, the integrals are written in the *Dirac bracket* notation, as follows.

$$\langle m|\hat{\Omega}|n\rangle = \int f_m^* \Omega f_n d\tau \tag{2.16}$$

the symbol $\langle m|$ is called the *bra* and represents the complex conjugates of the function f_m^*. On the other hand, the symbol $|n\rangle$ is called *ket* and denotes the function f_n [4]. The *normalization integral* can be represented by the Dirac bracket notation as follows

$$\langle m|n\rangle = \int f_m^* f_n d\tau = \delta_{mn} \tag{2.17}$$

Matrix Notation A *matrix* is an array of numbers. Each number is called a *matrix elements* and is specified by the *row(r)* number and *column(c)* number. *Dirac brackets* are the elements of a matrix of the operator $\hat{\Omega}$ [4].

$$\langle m|\hat{\Omega}|n\rangle = \mathbf{\Omega}_{mn} \tag{2.18}$$

A *diagonal matrix* is also defined as of the form $\langle n|\hat{\Omega}|n\rangle$ where *bra* and *ket* referring to the same state.

Hermitian Operator A *Hermitian* operator satisfies the following relation

$$\int f_m^* \hat{\Omega} f_n d\tau = \left\{ \int f_n^* \hat{\Omega} f_m d\tau \right\}^* \tag{2.19}$$

for any functions of f_m and f_n [4]. The definition of *Hermiticity* in terms of the Dirac notation is as follows

$$\langle m|\hat{\Omega}|n\rangle = \langle n|\hat{\Omega}|m\rangle^* \tag{2.20}$$

Hermitian operators are very important for the quantum mechanics and have two important properties [4]:

1. The eigenvalues of hermitian operators are real.
2. Eigenfunctions of an Hermitian operator are orthogonal.

Functional A *function* is a prescription for producing a number from a set of *variables*. Similarly a *functional* is a prescription for producing a number from a function, which in turns depends on variables. For example, a wave function is a function, while the energy depending on a wave function is called a functional. Usually, a function is denoted by a set of depending variables with parentheses, $f(x)$. And a functional is denoted by the depending functions with brackets, $F[f]$ [6].

2.3 Electronic Structure

In quantum mechanics, the state of a system can be entirely described by a *wave function* Ψ, which depends on the co-ordinates of all particles of the system.

$$\Psi_n(x_1, x_2, ...x_N, R_1, R_1, ...R_M) \tag{2.21}$$

Here $x_i = (r_i, s_i)$ are the space-spin co-ordinates of the N electrons, where $r_i = (x_i, y_i, z_i)$ denotes the Cartesian co-ordinates of the electrons, s_i denotes the spin co-ordinates. $R_K = (X_K, Y_K, Z_K)$ are the Cartesian co-ordinates of the M nuclei. The *quantum number*, n denotes the different molecular states [5]. There are some certain conditions and properties that the wave function should obey, for instance, anti-symmetricity property:

$$\Psi_n((x_1, x_2, ...x_N) = -\Psi_n((x_2, x_1, ...x_N) \tag{2.22}$$

The wave functions are the solutions of the molecular *Schrödinger equation.*

$$\hat{H}\Psi_n = E_n\Psi_n \tag{2.23}$$

The *Hamiltonian* $\hat{H}$ is the quantum mechanical operator for the energy, its *eigenvalues* E_n are the energies of the stationary states and the *eigenfunctions* Ψ_n are the corresponding molecular wave functions. The *Schrödinger equation* is an *eigenvalue equation*, which in general has solutions only for discrete values, E_n. If the solutions of (2.23) are generated without reference to the experimental data, the method is normally called an *ab-initio* method. Where as *semi-empirical* methods referred to the experimental data in some extents [6]. If the wave function Ψ_n is known, any time-independent *observable* for state n can be computed by calculating the *expectation value*

$$\int \Psi_n^*(\tau)\hat{\Omega}\Psi_n(\tau)d\tau \tag{2.24}$$

where the integration is over the all spatial co-ordinates and over the full space τ. $\hat{\Omega}$ is the *Hermitian* (2.19) operator associated with the observable property. The wave function is assumed to be normalized (2.15) [5]. Usually the *ground state energy*, i.e., the state with lowest energy is of the main interest.

2.3.1 The Born-Oppenheimer Approximation

Due to the complexity of the wave function structure, the *Schrödinger equation* can not be solved analytically [4] [5] [6]. As the electrons are lighter than the nuclei (the mass of the lightest nucleus, the proton H^+, is 1837 times larger than that of an electron) [5], they can respond almost instantaneously to displacements of the nuclei and it is assumed that the charge distribution adjusts instantaneously to the slow motion of the nuclei. Therefore it is convenient to fix the nuclear positions and solve the Schrödinger equation for the electrons of a fixed molecular structure. This is known as the *Born-Oppenheimer approximation* [4]. If the nuclear co-ordinates are varied, the electronic energy can be obtained as a function of the nuclear co-ordinates, which is known as *potential energy surface* (PES) [5].

In the *Born-Oppenheimer approximation* the molecular wave function is approximated as a product function

$$\Psi_{n,v,J,\ldots}(x,R) = \psi^{(n)}(x,R)\chi^{(n)}_{n,v,J,\ldots}(R) \tag{2.25}$$

where the *electronic wave functions* $\psi^{(n)}(x,R)$ describe the electronic structure for a fixed nuclear geometry $R \equiv \{R_1, R_2, ..., R_M\}$ and the nuclear wave functions $\chi^{(n)}_{n,v,J,\ldots}(R)$ describe the nuclear motions [5]. So the *electronic Schrödinger equation* is solved first for fixed nuclear coordinates

$$\hat{H}_e\psi^{(n)}(x,R) = E_e^{(n)}(R)\psi^{(n)}(x,R) \tag{2.26}$$

resulting in the electronic wave functions $\psi(x,R)$ and *energies* $E_e^{(n)}$ (the PES). The n is called the electronic quantum number and denotes the individual electronic eigenstates [4]. Here $\hat{H}_e$ is the *electronic Hamiltonian operator* which is a sum of electron kinetic energy, T_e, nuclear-electron attraction, V_{ne}, electron -electron repulsion, V_{ee} and nuclear-nuclear repulsion, V_{nn}, as defined below [6].

$$\hat{H}_e = \hat{T}_e + V_{ne} + V_{ee} + V_{nn} \tag{2.27}$$

$$\hat{T}_e = -\sum_i^N \frac{1}{2}\nabla_i^2 \tag{2.28}$$

$$V_{ne} = -\sum_i^N \sum_a \frac{Z_a}{|R_a - r_i|} \tag{2.29}$$

$$V_{ee} = \sum_i^N \sum_{j>i}^N \frac{1}{|r_i - r_j|} \tag{2.30}$$

$$V_{nn} = \sum_a \sum_{b>a} \frac{Z_a Z_b}{|R_a - R_b|} \tag{2.31}$$

This is represented in atomic units: $\hbar = 1$, $e = 1$, $m_e = 1$ and $4\pi\varepsilon_0 = 1$. And the numclear Schrödinger equation is then written as:

$$[\hat{T}_n + E_e^n(R)]\chi_{v,J,\ldots}^{(n)}(R) = E_{v,J,\ldots}^{(n)}(R)\chi_{v,J,\ldots}^{(n)}(R) \tag{2.32}$$

2.3.2 Hartree-Fock Self-Consistent Field Method

The electronic Schrödinger equation can only be solved exactly for the H_2^+ molecule and for similar one electron systems [6]. But for the general case, like in many electron systems, it is not possible to solve the equation exactly. Additional approximations have to be used (numerical methods). By neglecting the relativistic effects, electron spin is introduced as an *ad hoc* quantum effect. Each electron has a spin quantum number of $\frac{1}{2}$ with two possible states, corresponding to alignment along or opposite to the external magnetic field. The spin functions are represented by α and β, with following *orthonormality* conditions.

$$\langle\alpha|\alpha\rangle = \langle\beta|\beta\rangle = 1 \tag{2.33}$$

$$\langle\alpha|\beta\rangle = \langle\beta|\alpha\rangle = 0 \tag{2.34}$$

The *variational principle* helps here to generate approximate solutions of the electronic Schrödinger equation. The variational principle states that for any trial wave function ψ, the energy expectation value E is always an upper bound to the exact ground state energy E_1 [5]. The energy of an approximate (or trial) wave function can be calculated as the *expectation value* of the Hamiltonian operator (except for Coupled Cluster (CC) method), divided by the norm of the wave function and is represented below with *Dirac bracket* notation [6]:

$$E_e = \frac{\langle\psi|H_e|\psi\rangle}{\langle\psi|\psi\rangle} \geq E_1 \tag{2.35}$$

For a normalized wave function the denominator of eq. (2.35) is equal to 1 (see eq. (2.15)). Considering the *Pauli principle*, which states that two electrons can not have all quantum numbers equal, the wave function must be antisymmetric. The antisymmetry holds if the wave functions change their sign for any interchange of any two electron co-ordinates. The antisymmetry of the wave function can be achieved by constructing the wave functions from *Slater Determinants* (SDs). The columns in a Slater determinant are single electron wave functions, *spin orbitals*, while the electron coordinates are along the rows. The one electron functions are known as *Molecular spin orbitals* (MOs) which are defined as a product of a *spatial orbitals* and a spin function (α or β) [6]. The Slater determinant of N electrons is given below:

$$\Phi_{SD} = \frac{1}{\sqrt{N!}} \begin{pmatrix} \psi_1(x_1) & \psi_2(x_1) & \cdots & \psi_N(x_1) \\ \psi_1(x_2) & \psi_2(x_2) & \cdots & \psi_N(x_2) \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \psi_1(x_N) & \psi_2(x_N) & \cdots & \psi_N(x_N) \end{pmatrix} \tag{2.36}$$

Here $\psi_i(x) = \phi_i(r)\varrho(s)$, where spin function $\varrho(s) = \{\alpha, \beta\}$ and $\phi_i(r)$ is the spatial function. Now, considering a trial wave function that consists of a single Slater determinant the variational principle can be used to derive the *Hartree-Fock* (HF) equations [5]. The HF equations is a kind of branching point, as explained in the figure 2.1, for either leading to semi-empirical methods with additional approximations, or Self-consistent Field method by adding additional determinants where the solutions converge towards the exact solution of the electronic *Schrödinger equation* [6].

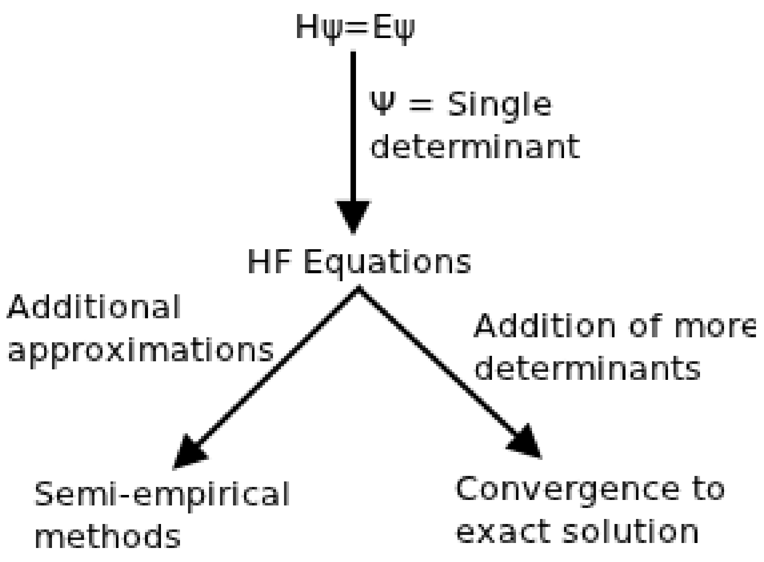

Hψ=Eψ
Ψ = Single determinant
HF Equations
Additional approximations
Addition of more determinants
Semi-empirical methods
Convergence to exact solution

Figure 2.1: The HF model, starting point of different computational methods [7]

2.3.3 Hartree-Fock Equations

To derive the Hartree-Fock equations, a further approximation is made. Considering a single Slater determinant as a trial wave function which is a sum of permutations over the diagonal of the determinant and is represented by Φ,

$$\Phi = A[\phi_1(1)\phi_2(2)\cdots\phi_N(N)] \tag{2.37}$$

$$A = \frac{1}{\sqrt{N!}}\sum_{p=0}^{N-1}(-1)^p P \tag{2.38}$$

where 1 operator is the *identity* and P generates all the possible permutations of electron coordinates [6]. In this case the operators mentioned in (2.27) can be formed according to the number of electron indices.

$$h_i = -\frac{1}{2}\nabla_i^2 - \sum_a \frac{Z_a}{|R_a - r_i|} \tag{2.39}$$

$$g_{ij} = \frac{1}{|r_i - r_j|} \tag{2.40}$$

$$H_e = \sum_{i=1}^{N} h_i + \sum_{i=1}^{N}\sum_{j>i}^{N} g_{ij} + V_{nn} \tag{2.41}$$

Here h_i is the one electron operator that describes the motion of electron i in the field of all the nuclei, g_{ij} is a two electron operator that describes the electron-electron repulsion and H_e defines the electronic Hamiltonian operator (For more elaboration, see [6], page 59- 63). The energy then can be written in a more symmetrical form as

$$E_{HF} = \sum_{i=1}^{N}\langle\phi_i|h_1|\phi_i\rangle + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=i}^{N}(J_{ij} - K_{ij}) + V_{nn} \tag{2.42}$$

$$J_{12} = \langle\phi_i(1)\phi_j(2)|g_{12}|\phi_i(1)\phi_j(2)\rangle \tag{2.43}$$

$$K_{12} = \langle\phi_i(1)\phi_j(2)|g_{12}|\phi_j(1)\phi_i(2)\rangle \tag{2.44}$$

where the factor of $\frac{1}{2}$ allows the double sum to run over all electrons and it can be shown that the Coulomb *self-interaction* J_{ii} is exactly canceled by the corresponding *exchange* element K_{ii} [6].

Based on the variational principle, the energy can be minimized or at least be made stationary with respect to the shape of the orbitals, subject to the conditions that the orbitals must remain orthonormal. This will give the best possible energy for a single Slater determinant. The orthonormality can be maintained by minimizing the *Lagrange functional*

$$L = E - \sum_{ij}^{N}\varepsilon_{ij}(\langle\phi_i|\phi_j\rangle - \delta_{ij}) \tag{2.45}$$

with respect to the arbitrary variation $\delta\phi_i$ of the orbitals. And the variation of the energy can be written in terms of *Fock operator* $\hat{F}_i$, which is the sum of usual one-electron operator, $\hat{h}_i$ and an average potential operator, $\hat{g}_i = \sum_j (\hat{J}_j - \hat{K}_j)$:

$$\hat{F} = \hat{h} + \sum_j^N (\hat{J}_j - \hat{K}_j) \tag{2.46}$$

The *Fock operator* is one-electron energy operator. It presents the *kinetic energy* of an electron and the attraction to all the nuclei by $\hat{h}_i$ and the repulsion to all the other electrons through the $\hat{J}$ and $\hat{K}$ operators. The $\hat{J}$ and $\hat{K}$ operators are defined as follows.

$$\hat{J}_j(1) = \int \phi_j^*(r_2) \frac{1}{r_{12}} \phi_j(r_2) dr_2 \tag{2.47}$$

$$\hat{K}_j(1) = \int \phi_j^*(r_2) \frac{1}{r_{12}} \phi_j(r_1) dr_2 \tag{2.48}$$

The Fock operator depends on the molecular orbitals, ϕ_i. If the molecular orbitals (MOs) are transformed by a unitary matrix transformation and the Lagrange multipliers are elements of a *Hermitian* matrix, the final Hartree-Fock equations will take the form of

$$\hat{F}\phi_i = \varepsilon_i \phi_i \tag{2.49}$$

which is a pseudo-eigenvalue equation. These transformed molecular orbitals, ϕ_i, are called *canonical MOs.* A specific Fock orbital can only be determined if all the other occupied orbitals are known. It is possible to use an iterative scheme in which in each iteration the Fock operator is kept fixed. Then the improved orbitals can be found by solving the eigenvalue equation. These orbitals can be used to construct a new Fock operator and this procedure is repeated until the convergence is achieved. This procedure is called the *Hartree-Fock self consistent-field* (SCF) method. It is not always guaranteed that this method converges, in that case, special convergence acceleration methods are required. Another important problem is how to parametrize the orbitals so that the eigenvalue equation can be solved efficiently [5]. The most commonly used method is to expand the molecular orbitals in terms of some basis set approximation. This is briefly described in the next section.

2.3.4 The Basis Set Approximation

The standard method to solve the Hartree-Fock equations is to expand the molecular orbitals (MOs) in terms of basis functions known as *atomic orbitals* (AOs). In principle, any type of basis functions can be used, like exponential, Gaussian, polynomial, etc. This MOs expansion is also known as, *Linear Combination of Atomic*

Orbitals (LCAO).

$$\phi_i = \sum_{\alpha}^{M} C_{\alpha i}\chi_\alpha \tag{2.50}$$

The Hartree-Fock equations then has the form of

$$\hat{F}\sum_{\alpha}^{M} C_{\alpha i}\chi_\alpha = \varepsilon_i \sum_{\alpha}^{M} C_{\alpha i}\chi_\alpha \tag{2.51}$$

This is known as Fock equations in the *atomic orbital basis.* The coefficients $C_{\alpha i}$ represent the orbitals in AO basis. This equation (2.51) has to be projected from the leftby $\langle\chi_\beta|$ to get a matrix *eigenvalue* equation.

$$\sum_{\alpha}\langle\chi_\beta|\hat{F}|\chi_\alpha\rangle C_{\alpha i} = \varepsilon_i \sum_{\alpha} C_{\alpha i}\langle\chi_\beta|\chi_\alpha\rangle \tag{2.52}$$

All the equations are collected in matrix notations which has the form of

$$\mathbf{FC} = \mathbf{SC}\varepsilon \tag{2.53}$$

$$F_{\beta\alpha} = \langle\chi_\beta|F|\chi_\alpha\rangle \tag{2.54}$$

$$S_{\beta\alpha} = \langle\chi_\beta|\chi_\alpha\rangle \tag{2.55}$$

The matrix **S** contains the overlap elements between basis functions and the matrix **F** contains Fock matrix elements [6]. The density $\rho(r)$ can be formed in terms of the basis functions as follows,

$$\rho(r) = \sum_{i=1}^{N}\phi_i\phi_i^* = \sum_{\alpha\beta}\chi_\alpha(r)\chi_\beta^* D_{\alpha\beta} \tag{2.56}$$

$$D_{\alpha\beta} = \sum_{i=1}^{N} C_{\alpha i}C_{\beta i}^* \tag{2.57}$$

The *Roothaan-Hall equation* is the determination of the eigenvalues of the Fock matrix. These equations are solved iteratively, known as SCF procedure, as the Fock matrix depends on it's own solutions. This is illustrated in the figure 2.2. The main steps in the SCF procedure [6] are:

1. Compute all one- and two-electron integrals
2. Prepare an acceptable guess for the MO coefficients
3. Build the initial density matrix
4. Build the Fock matrix
5. Diagonalize the Fock matrix
6. Calculate the new density matrix, check for sufficient convergence, if not converged repeat from step (4)

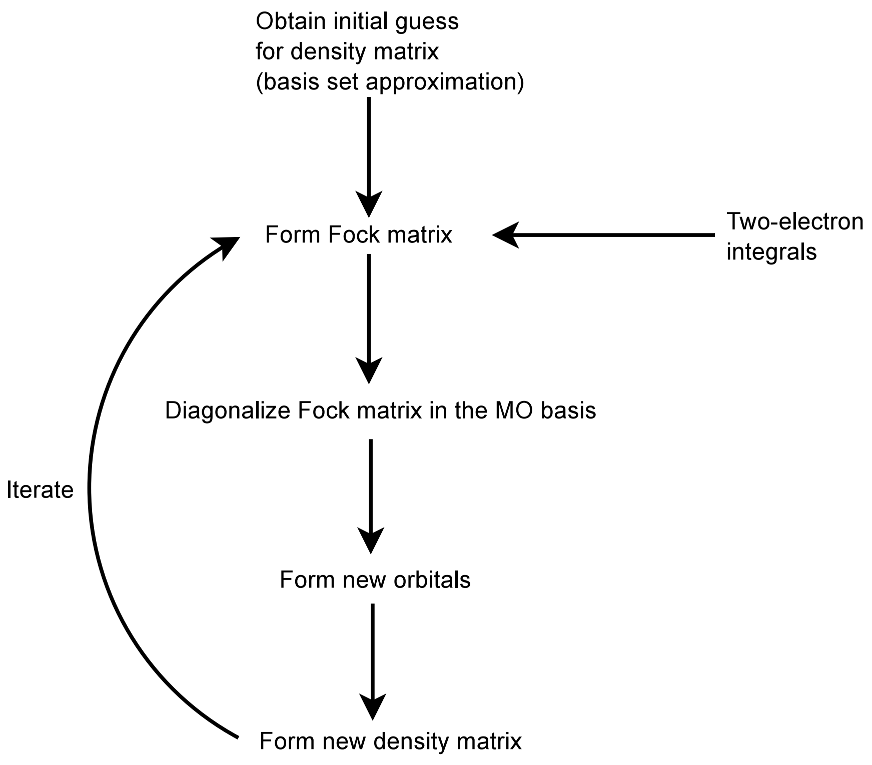


Figure 2.2: The SCF procedure
[7]

2.3.5 The Density Functional Theory

The *Density Functional Theory* (DFT) can be seen as an extension of the Hartree-Fock methods. DFT is one of the extensively used method in *computational chemistry*, due to its simplicity and good accuracy/cost ratio. The basis for DFT is based on the proof by Hohenberg and Kohn [8], that the ground-state electronic energy is determined completely by the *electron density, ρ*. The wave function for an N-electron systems contains $3N$ coordinates, three for each electron. The electron density only depends on three coordinates. It is calculated from the square of the wave functions that are integrated over $N-1$ electrons coordinates. The complexity of the wave function approach depends on the number of electrons. It increases with the increasing number of electrons. Whereas the complexity of electron density remains same independently of the system size. Thus the *density functional theory* is much simpler and cheaper compared to the wave function approach. But the crucial point is that the exact energy functional in terms of the density is unknown. However, there are many different approximate functionals proposed in the literature for different kind of molecular systems or different molecular properties. The results are depending on the used functionals.

2.3.5.1 The Hohenberg-Kohn Theorem

Density Functional Theory is based on two fundamental theorems proved by Hohenberg and Kohn. The first theorem states that the exact ground sate energy can be expressed as functional of electron density $\rho(r)$.

$$\rho(r_1) = N \int_{R^3} dr_2 \int_{R^3} dr_3 \cdots \int_{R^3} dr_N \Psi^*(r_1, r_2, \cdots, r_N)\Psi(r_1, r_2, \cdots, r_N) \quad (2.58)$$

where N is the number of electrons, the ground state wave function Ψ is assumed to be normalized and the spin coordinate is omitted. Since the electrons are indistinguishable, the electronic index can be omitted and the density can be simply represented by $\rho(r)$. $\rho(r)dr$ is the *probability* of finding any electron in the volume element dr at position r, independent of the positions of the other electrons. The density must fulfill the *N-representability* condition

$$\int_{R^3} \rho(r)\, dr = N \quad (2.59)$$

The energy can be expressed as a functional of the density.

$$E[\rho(r)] = \int_{R^3} v_{ext}(r)\rho(r)\, dr + F[\rho(r)] \quad (2.60)$$

where $v_{ext}(r)$ is the external potential of the nuclear charges

$$v_{ext}(r) = -\frac{e^2}{4\pi\varepsilon_0} \sum_{k=1}^{M} \frac{Z_k}{|R_k - r|} \quad (2.61)$$

The energy functional $F[\rho(r)]$ is universal and independent of the external potential. It explains the pure electronic energy as a function of the density. The second theorem states that, if the energy functional $F[\rho(r)]$ is known, the exact ground-state energy E_{GS} can only be obtained with the exact density $\rho_{GS}(r)$; for all other approximate densities $\rho(r)$ the energy is higher than the exact energy, i.e.,

$$E[\rho(r)] \geq E_{GS} \quad (2.62)$$

So in principle, the exact density can be obtained my minimizing the energy functional with respect to ρ. *Thomas-Fermi-Dirac* (TFD) approximate the electron as a non-interacting uniform electron gas to determine the unknown energy functional $F[\rho(r)]$. In this case, the energy can be written as,

$$E_{TFD}(\rho) = T(\rho) + V_{ext}(\rho) + J(\rho) + K(\rho) \quad (2.63)$$

where $V_{ext}(\rho)$ and $J(\rho)$ are the classical Coulomb interaction energies and are defined as:

$$V_{ext}(\rho) = \int_{R^3} \rho(r) v_{ext}(r) dr \quad (2.64)$$

$$J(\rho) = \frac{1}{2} \int_{R^3} dr_1 \int_{R^3} dr_2 \rho(r_1) \frac{e^2}{4\pi\varepsilon_0 r_{12}} \rho(r_2) \quad (2.65)$$

This assumption of a non-interacting uniform electron gas does not hold very well for atomic and molecular systems. The TFD approximation can not predict bonding and molecules simply do not exists in this case [6].

2.3.5.2 Kohn-Sham Theory

The main problem in *Thomas-Fermi-Dirac* model is that the *kinetic energy* is represented very poorly. The basic idea in the *Kohn and Sham* (KS) formalism is splitting the *kinetic energy functional* in to two parts, one of which can be calculated exactly and and a small correction term. The Kohn-Sham formalism treats the system as an independent particle model, like the *Hartree-Fock* theory. The *kinetic energy*, the *external potential* and the average *Coulomb interaction* are treated exactly as in the *Hartree Fock*, based on antisymmetrized product of spin-orbitals [5]. Since the exact density matrix is unknown, the (approximate) density is defined in terms of a set of auxiliary one-electron functions (orbitals) as

$$\rho(r) = \sum_{i=1}^{N} |\phi_i(r)|^2 \tag{2.66}$$

The kinetic energy for independent particles is computed exactly as

$$T_s\left[\rho\right] = -\frac{\hbar^2}{m_e}\sum_{i=1}^{N}\langle\phi_i|\nabla^2|\phi_i\rangle \tag{2.67}$$

It is a functional of density, ρ, because the one electron orbitals depend on ρ [5]. The *external potential,* $V_{ext}(\rho)$, and the *Coulomb-repulsion* energy, $J(\rho)$ are defined as like in equation (2.64) using the density calculated as in equation (2.66). The difference between the exact kinetic energy functional, $T(\rho)$, and the approximate kinetic energy for non-interacting electrons, $T_s(\rho)$, and the remaining exchange and correlations contributions are collected in an unknown *exchange correlation functional* called $F_{XC}\left[\rho\right]$. So the energy as a functional of density is rewritten as

$$E\left[\rho\right] = T_s\left[\rho\right] + V_{ext}(\rho) + J(\rho) + F_{XC}\left[\rho\right] \tag{2.68}$$

Minimization of this functional with respect to ρ yields the *Kohn-Sham equation*

$$\hat{h}^{KS}\phi_i = \varepsilon_i\phi_i \tag{2.69}$$

Here $\hat{h}^{KS}$ is the *Kohn-Sham* Hamiltonian and the one-electron *Kohn-Sham* Hamiltonian is defined as:

$$\hat{h}^{KS}(1) = -\frac{\hbar^2}{2m_e}\nabla_1^2 + v_{ext}(r_1) + j(r_1) + v^{xc}(r_1) \equiv \hat{h}(1) + j(r_1) + v^{xc}(r_1) \tag{2.70}$$

where $j(r)$ and $v^{xc}(r)$ are the Coulomb potential and the exchange correlation potential respectively [5]:

$$j(r_1) = \frac{e^2}{4\pi\varepsilon_0}\int_{R^3}\frac{\rho(r_2)}{r_{12}}dr_2 \tag{2.71}$$

$$v^{xc}(r_1) = \frac{\delta F_{XC}\left[\rho(r_1)\right]}{\delta\rho(r)} \tag{2.72}$$

There are various approximations for the *exchange correlation potential*, like *local density approximation (LDA)* , *generalized gradient approximations (GGA)*, etc.

Local Density Approximation (LDA) In the *local density approximation* it is assumed that the density can be treated locally as a uniform electron gas. That means that the density is a slowly varying function [6]. The *exchange correlation potential* is local in the sense that it only depends on $\rho(r)$ at the same position [5]. The exchange energy is then written by

$$E_x^{LDA}[\rho] = -C_x \int \rho^{\frac{4}{3}}(r)dr \tag{2.73}$$

$$\varepsilon_x^{LDA}[\rho] = -C_x \rho^{\frac{1}{3}} \tag{2.74}$$

This approximation is reasonably simple, however not very accurate in practice [6].

Generalized Gradient Approximation (GGA) In this approximation, the exchange and correlation energies depend not only on the electron density, but also on derivatives of the density, i.e., $v^{xc}(r) \equiv v^{xc}[\rho(r), \nabla\rho(r)]$. GGA methods are also sometimes referred as *non-local* methods. There are various gradient corrected functional forms for the correlation energy. For instance, one popular functional form is Becke, Lee, Yang and Parr (LYP) [9] [10].

2.3.5.3 Solution to the Kohn-Sham Equation

Similar to the Hartree-Fock equations Kohn-Sham equations need to be solved iteratively until self consistency. In practice, the Kohn-Sham equations are solved in the LCAO approximation 2.3.4, just like the HF equations. So the Kohn-Sham matrix in the AO basis needs to be formed

$$h_{\alpha\beta}^{KS} = \langle\alpha|\hat{h}|\beta\rangle + \langle\alpha|\hat{j}|\beta\rangle + \langle\alpha|v^{xc}|\beta\rangle = h_{\alpha\beta} + j_{\alpha\beta} + v_{\alpha\beta}^{xc} \tag{2.75}$$

the integrals $v_{\mu\vartheta}^{xc}$ are approximated numerically on grid as

$$v_{\alpha\beta}^{xc} = \sum_{\lambda} w_\lambda v_{xc}(r_\lambda)\chi_\alpha(r_\lambda)\chi_\beta(r_\lambda) \tag{2.76}$$

Here, λ labels grid points r_λ with weights w_λ. $\chi_\alpha(r_\lambda)$ is the value of the basis function χ_α at the grid point r_λ. The density on the grid is computed by

$$\rho(r_\lambda) = \sum_{\alpha,\beta} D_{\alpha\beta}\chi_\alpha(r_\lambda)\chi_\beta(r_\lambda) \tag{2.77}$$

where the $D_{\alpha\beta}$ is the density matrix in AO basis. The *grid* [10] for a molecule is the union of atomic grids that are spherically symmetric around the nuclei [5].
In this chapter, the theoretical background of the DFT theory was briefly explained. Chapter 3 will shortly explain the related work done on the parallel DFT implementation. This will lead us to mapping of DFT calculation to the GPU architectures.

CHAPTER 3

State of the Art

Contents

3.1 Introduction

Chapter 2 explains a brief theoretical overview and the related mathematical background of the density functional theory (DFT) calculation in a quantum mechanical approach. DFT is one of the most widely used quantum mechanical methods for electronic structure calculations. Though it provides a balance between computational accuracy and the computational cost, but for larger molecular systems DFT still has very high computational cost. Therefore, there is a growing demand for much more efficient implementations of DFT. Several works have already been done to make an efficient parallel implementation. This chapter provides an overview about related works that has been done in DFT parallelization, which includes from multicore architecture to some extents of numerical accelerators. These related works has an influence on the implementation proposed in this thesis.

3.2 Related Work

DFT parallelization with ClearSpeed

A massively multi core parallelization of Kohn-Sham theory was introduced by Brown et. al. in their paper published in 2008 [11]. They presented a heterogeneous approach to accelerate DFT, where They combined ClearSpeed's low-power 64-bit accelerator technology in parallel with the host CPU. The ClearSpeed accelerated tera-scale system (CATS) was used in their implementations. ClearSpeed is a kind of dedicated numerical accelerator that can perform hundreds of floating point operations in parallel. ClearSpeed consists of SIMD (Single Instruction Multiple Data) array processors, each containing 96 processing elements (PEs). Typically, it has 96 GFLOPS of single/double precision peak performance. In the CATS system, where the DFT was accelerated, there were 2304 processing elements (PE) available.

A very fine grain parallelization is required in the ClearSpeed architecture. The algorithms need to be redesigned to fit into this paradigm. Initially, the two main bottlenecks of the DFT calculation for a macro molecular system ($\sim$50 Atoms) were identified: which involves the evaluation of the Coulomb matrix and to evaluate the exchange correlation contribution to the Fock matrix. The numerical quadrature to the exchange correlation contribution was parallelized in a straightforward way by distributing batches of integration points between processing elements of the ClearSpeeds. Screening over the basis function evaluation showed to have a large effect on the exchange correlation contribution to the Fock matrix, it reduces the complexity of the exchange correlation contribution matrix significantly. However, the Coulomb term was very problematic since it requires four-index ERIs (Electron Repulsion Integrals) which are hard to keep in the small local memories of ClearSpeed PEs. The authors proposed a solution idea to avoid this ERI calculation completely and used a combination of density fitting and Poisson equation. The exchange correlation quadrature and the density fitting Poisson method was implemented in a hybrid CPU - ClearSpeed fashion based on the Molpro [12] serial code. This hybrid implementation achieved a speedup of an order of magnitude, with good scaling over thousands of PEs. With large basis sets, speedup of $4\times$ is observed compared to the standard DF-KS (Density Fitting Kohn-Sham) calculation. However, the screening was not implemented, so the accelerators were performing more work than the equivalent host. The gradients were also not implemented on that architecture. Later this algorithm was introduced for implementation on standard shared memory parallel architectures, like multi core machines [13].

Multicore Kohn-Sham DFT

A multi-core implementation of the Kohn-Sham density functional theory (DFT) was introduced by Woods et. al. [13], which is optimized for modern commodity processors. Typically many scientific applications were developed and designed to use only a single CPU thread. They could only run on a single core of the system and can not automatically scale to the additional cores that are available on the system. A significant redesign of the application is also required to port the application to the multi-core architecture. However, this multi-core platform has an advantage over the numerical accelerators as they can be programmed using portable and well established languages like *OpenMP* and *MPI*. Woods et al, first identify the computational bottlenecks of the calculation and adapt those to run over multiple cores. The three bottlenecks in Kohn-Sham DFT theory for their target system are:

1. The evaluation of the Kohn-Sham density
2. The quadrature based evaluation of exchange correlation matrix, and
3. The evaluation of the Coulomb contribution.

These calculations involve computations of values of all the numbers of independent quadrature grid points. Hence, the program was designed in such a way that the outer loop was running over the grid points and the parallelization was achieved by dividing batches of grid points between processor cores. So that the parallelization was gained from the batch of grid points in per processor cores [13]. Distributing batches of grid points among processors cores was done by *OpenMP*. These computationally expensive modules from *Molpro* were rewritten to C++ to exploit the features parallelization using OpenMP. The orbital screening was also implemented which in turns reduced the scaling of computational work. The use of the numerical quadrature has resulted in linear scaling for each three computationally expensive parts up to 16 threads [13]. For the quad-quad Xeon platforms has resulted linear scaling up to 8 threads for the exchange correlation and the Coulomb parts, and for the density evaluation linear scaling maintained up to 16 threads. This paper also observed that the four thread OpenMP implementation was faster than using a ClearSpeed accelerator card [13].

Electron Repulsion Integrals on GPU

Several groups have investigated the scope of graphics processing units (GPUs) to use for quantum chemistry calculations. Ufimtsev and Martinez et. al., studied the scope of GPUs to calculate two-electron repulsion integrals (ERIs) [2]. ERIs calculation is one of the computational bottleneck in many electronic structure methods. Three different algorithms to evaluate 4-index ERI by GPU over the contracted basis functions are proposed in [14]. Based on the organization of the contracted ERIs, Ufimtsev and Martinez et. al. proposed three different mappings of the computational work to CUDA thread blocks [2].

1. A thread for each contracted ERI

2. A thread block for each contracted ERI

3. A thread for each primitive ERI

The grain of parallelism and the degree of load balancing differed in all three cases. All three schemes were implemented on the 32-bit precision NVIDIA G80 series, which does not support double-precision floating point arithmetic. The single precision operations are not always sufficient for quantum chemistry applications. The chemical accuracy is typically considered to be 10^{-3} atomic units [2]. So a significant effort is required to minimize the errors for this lack of double precision support. The performance comparison of this GPU implementation of ERI with GAMESS (a general purpose quantum chemistry programs) shows a significant speedups of $\sim 30\times$ to $\sim 50\times$ with single precision accuracy [2].

Exchange Correlation Terms on the GPU

Apart from this promising results from Ufimtsev and Martinez in [2], Yasuda proposed an algorithm for GPUs to evaluate the exchange correlation terms [1]. The evaluation of the exchange correlation term is reported as the most time consuming steps in the density functional calculation. Yasuda profiled the density functional calculation in *Gaussian03* program (a program for electronic structure modeling). The profiling identified that the $90 - 95\%$ of the total DFT computation time was used for self-consistent field (SCF) equations. Inside of each SCF, 10% of time was required to evaluate the coulomb potential, 85% time to evaluate the exchange-correlation potential and 5% time was required to diagonalize the fock matrix [1]. Yasuda proposed an algorithm to evaluate the exchange correlation term on GPU. This algorithm was implemented on a *NVIDIA GeForce 8800 GTX*, which supports only single precision operations. Yasuda evaluated the exchange correlation term by using three dimensional quadrature points and their weights. Four distinct steps were identified:

1. The quadrature points and weights are generated.
2. The electron density and the gradient on these points are calculated.
3. The energy functionals are generated.
4. The exchange-correlation potential is then evaluated.

It was noticed that most of the computational effort was spent on the evaluation of electron density and the matrix elements of the potential. The computational cost are proportional to the number of grid points and to the square of the number of atomic orbitals (AOs) [1]. These time consuming steps are implemented by Yasuda [1] on a GPU and then linked with a modified version of the *Gaussian03* program. Due to the lack of double precision support in that GPU, a significant efforts were made to minimize the errors in the calculation. The computational time to calculate the exchange correlation terms were compared with the execution time of Gaussian03 program and the GPU calculation is noticed to be five to ten times faster than the commodity CPU calculation [1].

3.3 Beyond the State of the Art

In the previous sections we have seen some related work on the parallelization of density functional theory. It has been seen that, many scientific applications are typically designed as a sequential program architecture due to the simplicity of coding. So they often can only executed on a single threaded core and cannot automatically scale to the available commodity cores or to any numerical accelerator. There are two methods to resolve this issue. One method is to completely design from the scratch to make the program parallel over multiple cores or over numerical accelerators. This may result in highly parallel designs with a good performance over the existing sequential implementation. However, this may come out with expensive design in case of very large, complex applications. And it is sometimes very complicated in case of quantum chemical calculations. Another approach is to analyze the existing code and identify the computationally expensive bottlenecks and adapt only those bottlenecks to multi-core architecture or numerical accelerators. The parallelization of the bottlenecks can be an effective strategy for any complex and large systems [15]. This second approach is taken in this thesis to map the density functional theory to GPU accelerators. The recent development of the GPU architectures, that now support double precision arithmetic with a good amount of on-chip memory, gives a big motivation to map this density functional theory calculation to such accelerators. The state of the art multi-core DFT implementation in the Molpro [12] quantum chemistry package has been analyzed in detail for fine grain parallelization scope. As explained before, Yasuda mapped exchange correlation terms of DFT to the GPU architecture, but it was implemented and compared with the Gaussian03 program. In this thesis, a multi-GPU architecture mapped with multi-core DFT module is proposed. The two main bottlenecks of this DFT modules in Molpro [12] have been ported to the state of the art NVIDIA Fermi architecture.

Chapter 4 will briefly explain the state of the art of GPU architectures and the corresponding programming model for these devices. The DFT architecture in Molpro [12] and the proposed implementation will be discussed in the subsequent chapters.

Chapter 4

GPU Architecture

Contents

4.1 Introduction

GPGPU stands for General-Purpose computation on Graphics Processing Units. The definition implies using Graphics Processing Units (GPU) to do general purpose scientific computing. It is also known as GPU computing. The GPU computing model uses a Central Processing Unit (CPU) and GPU together in a heterogeneous co-processing computing model. In this model, the sequential part of any applications runs typically on the CPU and computationally-intensive part is accelerated by the GPU [7]. The overall applications performance is gained from the high performance of the GPU. The developments of super computers observed in recent years shows that the future designs of large high performance computing systems will be heterogeneous in nature [16].

4.2 GPU Architecture

GPUs are massively parallel many-core processors with ample computational resources. Over the last few years, GPUs evolved from fixed-function special-purpose processors into full-fledged parallel programmable processors with additional fixed-function special purpose functionality. GPUs are built for different application demands than the CPU: large, parallel computation requirements with an emphasis on throughput rather than latency. As a consequence, the architecture of the GPUs is progressed in a different direction than CPU [17]. In the following the evolution of the GPU architecture will be briefly explained with a basic overview of the typical graphics pipeline.

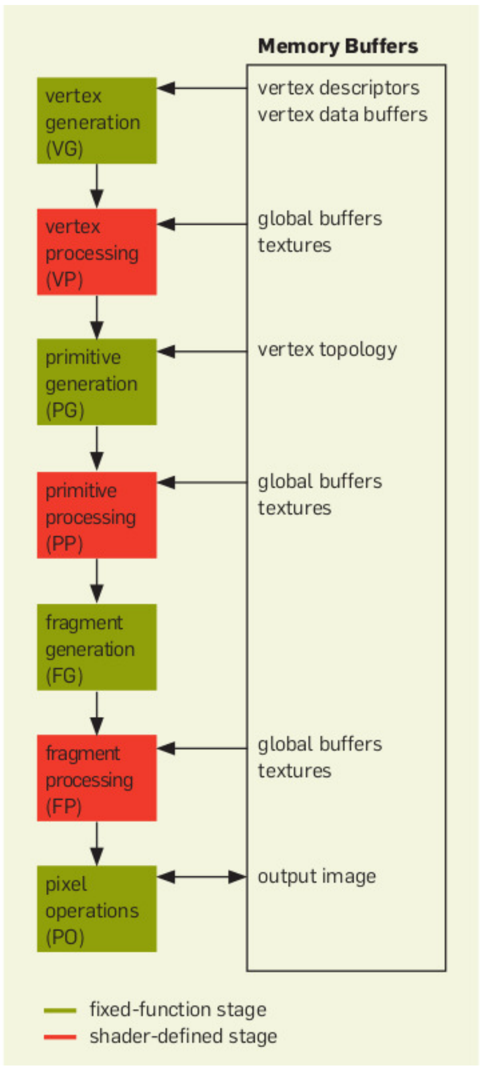


Figure 4.1: Graphics pipeline
[18]

4.2.1 The Graphics Pipeline

Typical input for a graphics data processing is a list of 3-D coordinates, known as geometric primitives. Through different steps. these primitives are shaded and mapped on the screen to create the final image. The different pipeline stages are shown in the figure 4.1.

Vertex Operations Input primitives are created from the individual vertices. Each vertex is transformed into screen space (*vertex generation*, see Fig: 4.1) and shaded by computing their interaction (*vertex processing*, see Fig: 4.1) with the lights in the scene. A typical scene has tens to hundreds of thousands of vertices, and each vertex can be computed independently. This stage is well suited for parallel hardware.

Primitive Assembly The vertices are assembled together to form triangles. Recent GPU architectures support these fundamental primitives. It includes the steps *Primitive Generation*, (Fig: 4.1), to generate primitives from vertex and *Primitive Processing*, (Fig: 4.1), to produce more output primitives.

Rasterization Rasterization is a process to determine the screen-space pixel locations that are covered by each triangle. Each triangle generates a fragment at each screen-space pixel location that it covers. Because many triangles may overlap at any pixel location, each pixels color value is computed from several fragments. It is also known as *fragment generation*, (Fig: 4.1).

Fragment Processing Each fragment is shaded and determines its final color using color information from the vertices. If required, it can fetch additional data from global memory. Each fragment can be computed in parallel. This stage is one of the computationally most intensive stage in graphics pipeline.

Pixel Operations A final image is assembled later with one color per pixel.

Traditionally, the operations at the vertex and fragment stages were configurable but not programmable. As they are not programmable, it is also known as fixed-function pipeline. So the next step was replacing this fixed-function operations with user-specified programs, which allows more sophisticated operations for complex effects. Over the years, these vertex programs and fragments programs have become more capable with larger limits on their size and resource consumption. Initially they had separate instruction sets for vertex and fragment operations. Later GPU's started supporting the Unified Shader Model 4.0 [17].

4.2.2 Evolution of Modern GPU Architecture

In Modern GPUs the ideas of parallel computation are emphasized. Hence, the GPU architecture progress in a different direction than CPUs. The pipeline is divided into time scale in the modern CPUs, that means in turn each pipeline stage will use all processor resources. Whereas, GPU pipeline is divided into space, the processor resources are divided among the pipeline stages. Hence, the programmable stage and components replace the fixed function stage and components, respectively [17]. In GPU, the latency of any given operation is very long. For instance, for any given operation to enter and leave the CPU pipeline may take 20 cycles, where as, a GPU operations may take thousands of cycles. However, a very high throughput is achieved form the data parallelism across pipeline stages [17]. Figure 4.2 shows

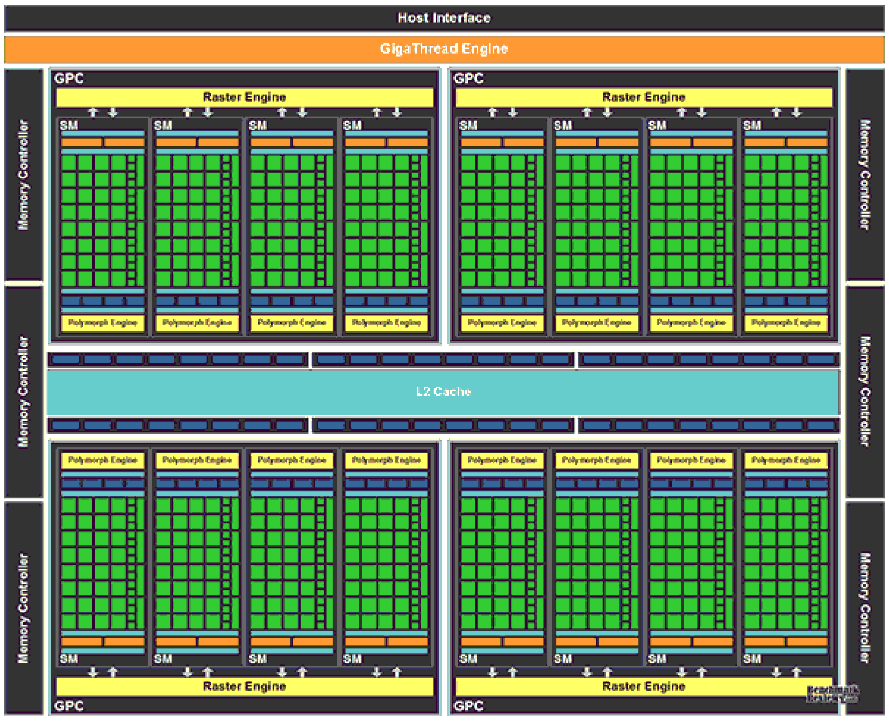


Figure 4.2: CUDA architecture
[19]

the state of the art NVIDIA CUDA Fermi architecture. Fermi present 512 CUDA cores, that are organized in 16 Streaming Multiprocessors (SMs) of 32 cores each. It features six 64-bit memory partitions supporting a total of 6GB DRAM memory. The *GigaThread* engine distributes thread block. The Fermi architecture is known as the third generation streaming multiprocessors [19].

4.2.3 Application Programming Interface

There are two major commercial standards for GPGPU development. One is ATI's StreamSDK and the other is the "Compute Unified Device Architecture" (CUDA) represented by NVIDIA. The High Performance Computing (HPC) market adopts CUDA developing platform more extensively than StreamSDK.

4.3 Programming Model

The GPU computing model is known as the heterogeneous CPU - GPU programming model. In this model, the sequential parts of an application are running on CPU. The data parallel, computationally expensive, parts are running on the GPU. Together the CPU-GPU model accelerates the overall performance. In the early days of GPGPU programming, applications had to be programmed by graphics APIs, like *openGL*. The general applications have to be structured and mapped in

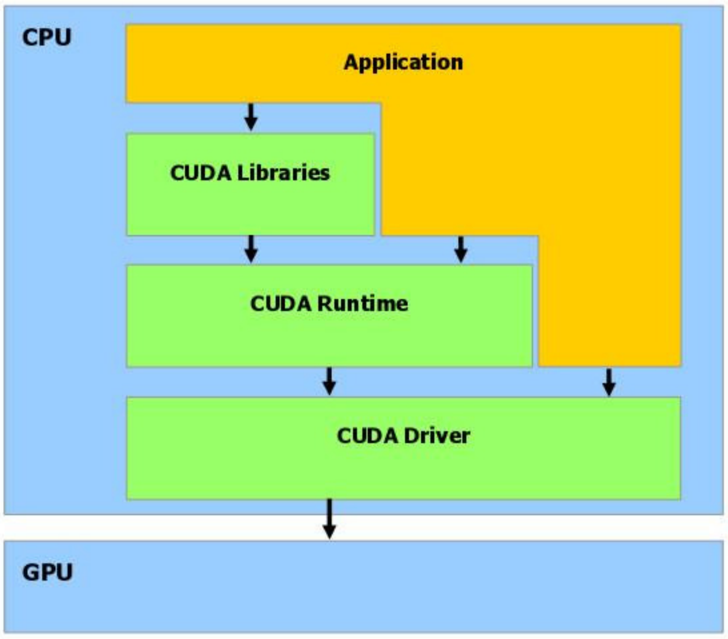


Figure 4.3: CUDA SDK structure
[20]

terms of graphics pipeline, although the task may not related with the graphics. Today, in recent programming models, these kind of obstacles are gone as there are more natural, direct and non-graphic interfaces to the programmable units of the device are introduced. Now, GPU computing applications are structured in the following way.

1. The computation domain is defined by the programmer as a structured grid of threads
2. A SIMD (Single Instruction Multiple Data) program computes the result from each thread.
3. The result from each thread is computed by a combination of operations and both read from and write to the device memory.
4. The resulting data in the device memory can be used as an input in future computation [17].

This programming model is a powerful for it's simplicity and generality. The application can optimally exploit the massive parallelism of the hardware. However, There are some restrictions too, like restrictions on branching, restrictions on data communications between elements and between kernels. This is maintained to ensure good performance. Consequently, this also allows writing more general algorithms. To maintain this programming structure it is important to know about the device memory hierarchy, thread structures, kernel and the overall compilation flow inside the graphics device.

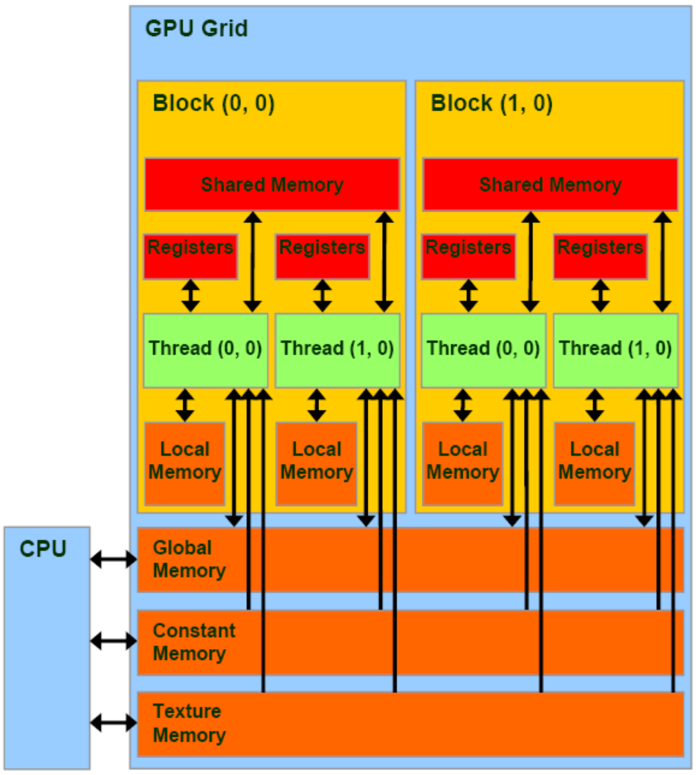


Figure 4.4: CUDA memory model
[7]

4.3.1 Device Memory Hierarchy

In the *CUDA programming model*, any system is composed of a host and a device, each with their own separate memory. A CUDA kernel only operates on device memory. To allocate, deallocate and copy to device memory, CUDA runtime provides several functions. It also provides functions to transfer data between host memory and device memory. Device memory is allocated as *linear memory*. Linear memory exists on the device in a 32-bit address space for device of compute capability 1.x and 40-bit address space for device of compute capability 2.x. These memory entities can reference one another via pointers. There are multiple memory spaces resides in the CUDA device. The figure 4.4 explains the memory hierarchy structure.

Each CUDA threads can access data from these memory spaces as required. Each thread has its own local private memory. Additionally, each thread block has *shared memory* that is only visible to all the threads of that block. There is a *global* memory space that can be accessed by all the running CUDA threads of any application. There are two additional read-only memory spaces - *constant memory* and *texture memory* - that are accessible by all the threads. The *global*, *constant* and *texture memory* spaces are persistent during the lifetime of the application unless explicitly cleared [7].

4.3.1.1 Global Memory

Global memory is the physical memory that resides on the device. This is the amount of memory alloted in the graphics device. This memory can be accessed by all the CUDA threads via 32-, 64- or 128 byte memory transactions. An associated host (CPU) thread has read and write access to the global memory. Global memory is generally allocated using *cudaMalloc()* and freed using *cudaFree()*. The data transfer between host memory and device memory are done using *cudaMemcopy* (please refer to the *CUDA C Programming Guide* [7] for the other API functions). All the CUDA threads can synchronize with each other through *global memory* [7].

4.3.1.2 Shared Memory

Shared memory is equivalent to a user managed cache. A GPU consists of multiprocessors and each multi processor has a small amount of shared memory, typically in the order of about 16KB. This memory is only accessible by the threads in a single thread block, as shown in figure 4.5. Shared memory is generally used as a very quick working space for threads within a block. Shared memory is allocated using a variable type qualifier *shared* and it has a life time of the thread block. As it is on-chip, the shared memory space is faster than the global and local memory space. Accessing shared memory is fast for all thread of a warp as long as there is no bank conflicts between the threads [7].

4.3.1.3 Texture and Surface Memory

The *texture* and *surface memory* resides in the *device memory*, figure 4.4. This memory space is used by the GPU for graphics. CUDA supports a subset of texturing hardware to use for general purpose calculation. Using texture or surface memory is sometimes advantageous as it costs only one memory read from device memory in case of any cache miss [7].

4.3.1.4 Constant Memory

The constant memory space resides on device memory, as shown in figure 4.4 and is cached in the constant cache. This memory is writable from the associated host (CPU) thread and only readable from the CUDA threads [7].

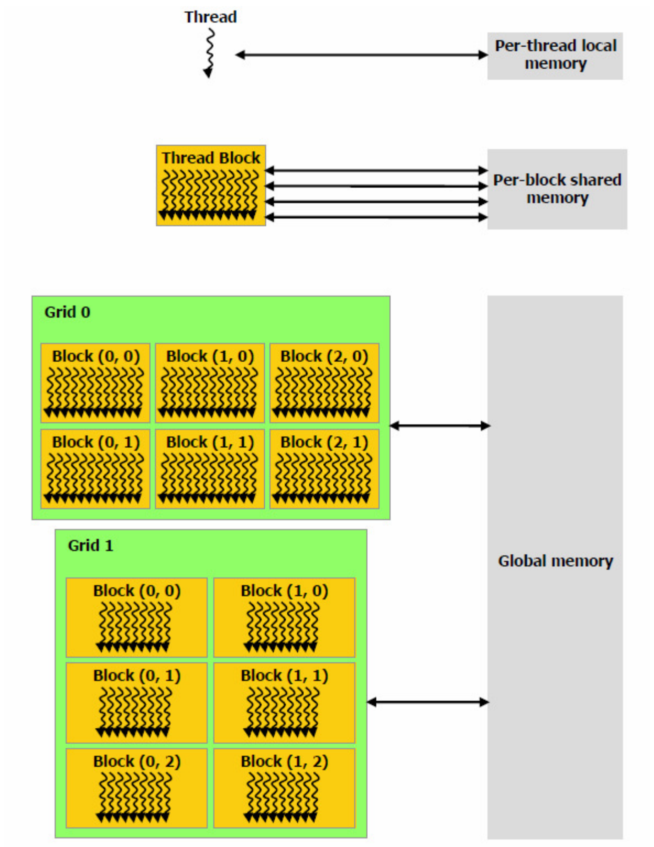


Figure 4.5: CUDA memory hierarchy
[7]

4.3.2 Thread Hierarchy

A heterogeneous CUDA program consists of one or more phases that are executed on either the host (CPU) or in GPU device. The NVIDIA C Compiler (NVCC) separates these phases into two, the host code and device code. The host code is straight C code. The code is compiled with the host's standard C compilers. The device code is written using C extended with keywords. The keyword is used for labeling data-parallel functions, called *kernels*, and the corresponding data structures. The *kernels functions*, or simply *kernels*, typically generate a large number of *threads* to exploit data parallelism [7]. In order to do efficient programming in CUDA, it is very important to know the *threads* hierarchy.

The execution starts on the host. The execution is moved to a GPU device when the application invokes a kernel function. To take the advantage of data parallelism, a large number of threads are generated. All the threads are collectively known as a grid. Figure 4.6 shows the execution of two Girds of threads. The grid is terminated, when all the threads of that kernel completed their execution. The

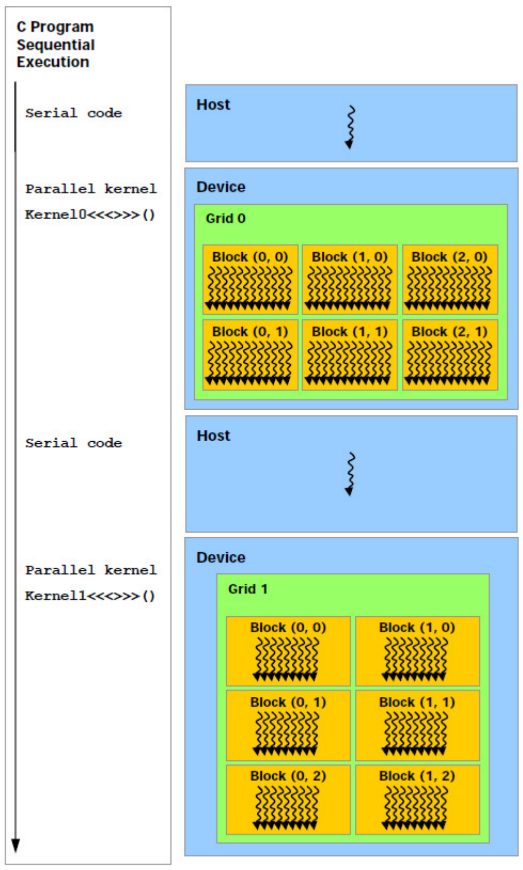


Figure 4.6: CUDA programming model
[7]

application continues its execution on the host until another kernel is invoked. However, on the new Fermi devices, multiple kernels can be executed in parallel on the same GPU [7].

CUDA Threads *CUDA threads* are very light weight and takes few cycles to generate. Each thread is identified by a variable known as *threadIdx*. The *threadIdx* is a 3-component vector and is represented by one-dimensional, two-dimensional or three-dimensional thread index.

- threadIdx.x
- threadIdx.y
- threadIdx.z

Half-Warp A half-warp is a group of 16 consecutive threads. Half-warp threads are generally executed together and they are aligned. For example, Threads 0 to 15 will be in the same half-warp, 16 to 31, and so on.

Warp A warp of threads is a group of 32 consecutive threads. Typically, threads 0 to 31 will be in the same warp, 32 to 63 and so on.

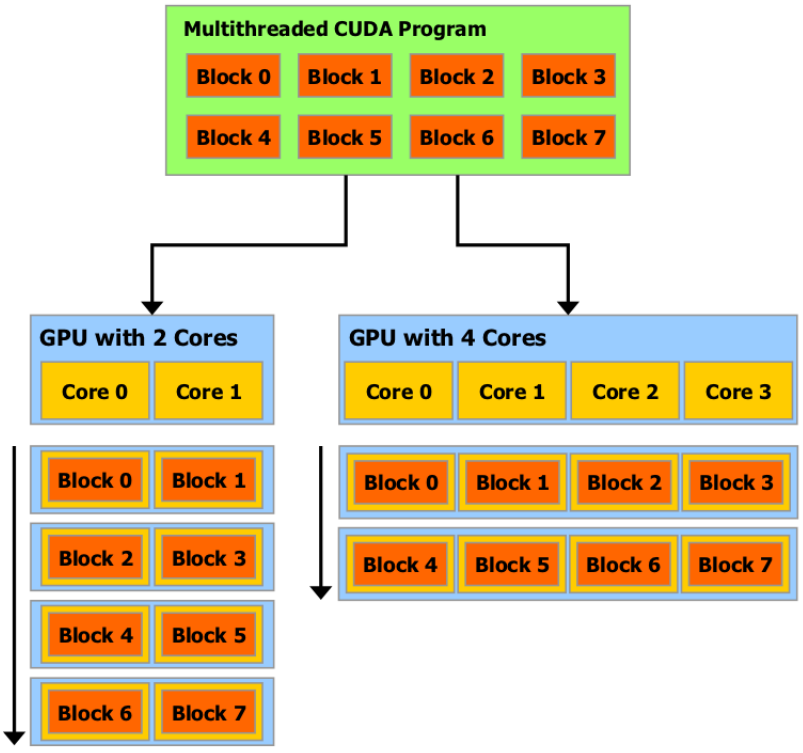


Figure 4.7: Automatic scalability
[7]

Block A *block* is a collection of threads. Threads are organized into a block of one-dimensional, two-dimensional or three-dimensional. This provides a natural way to map computation across the elements of a *vector*, *matrix* or *volume*. As all the threads in a thread block reside on the same processor core and share the limited memory resources of that core, there is a limit in the number of threads in each thread block. Typically, a block can contain up to 512 threads, however in the recent GPU's a thread block is allowed to have 1024 threads [7]. A thread block size of 16×16 is a common choice, however it depends on the required number of variables to compute, number of registers, amount of shared memory space in that block. The dimension of a thread block is accessible within the kernel through the built-in *blockDim* variable. Threads within the same block can synchronize and quickly communicate with each other through *shared memory*.

Thread blocks are executed independently and it is also possible to execute them in any order, in parallel or in series. This independency allows thread blocks to be scheduled in any order among all the cores as explained in the figure 4.7. As a result the CUDA code can be scaled to any number of cores [7].

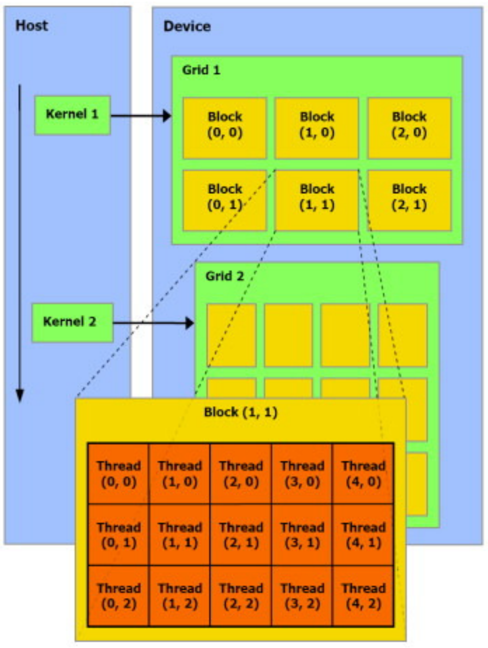


Figure 4.8: CUDA threads structure
[7]

Grid A grid is a collection of thread blocks. The blocks are organized into a one-dimensional or two-dimensional structure. The number of thread blocks in the grid is usually determined by the size of the data being processed. Each *thread block* within the grid is identified by a one-dimensional or one dimensional index accessible through the built-in *blockIdx* variable. Blocks can not synchronize with each other, and therefore synchronization is not possible among the threads of different blocks. The figure 4.8 explains this structure more clearly.

4.3.3 Kernel and Device Functions

In CUDA, a *kernel* function specifies the code to be executed by all threads of a parallel phase. The *__global__* qualifier is used to declare a function as kernel. The kernel function is only callable from associated host thread and executed on the device. When a kernel is called it generates a grid of threads as per the specification(from the grid dimension and block dimension) of the kernel. A call to kernel function is asynchronous, that means it returns before the device has completed its execution.

The *__device__* qualifier declares the *device function.* The device function is executed on the device and only callable from the device thread. Host thread is not allowed to call device routine. By default, this function is in-lined [7].

Chapter 5

Molpro - a package of ab initio programs

Contents

5.1 Introduction

In chapter 2, the density functional theory (DFT) was introduced. DFT is an essential component of *Molpro* [12], one of the most sophisticated general purpose quantum chemical packages available. In this chapter, an overview of Molpro is given and the architecture of the integrated DFT module is briefly described. This architecture formed the basis for the GPU mappings developed in this thesis. A detailed profiling of the Molpro DFT has been performed to identify the computationally most intensive algorithmic part.

5.2 Molpro - A Package of ab-initio Programs

Molpro is a general-purpose quantum chemical program for molecular electronic structure calculations [12]. The package is designed and maintained by H.-J. Werner and P. J. Knowles, and contains contributions from a number of other authors. Apart from other commonly used quantum chemistry packages, the original focus of Molpro was on *high-accuracy* wave function calculations for small molecules, with extensive treatment of the electron correlation problem through the multi-configuration reference CI, coupled cluster and associated methods. By using *local electron correlation methods*, Molpro significantly reduce the increase of the computational cost with molecular size [12]. *Ab initio* calculations can be performed more accurately for much larger molecules [21]. Molpro includes a state of the art implementation of density functional theory.

Density functional theory (DFT) is implemented in *Molpro* for spin-restricted Kohn-Sham (KS) and spin-unrestricted Kohn -Sham (UKS) cases. DFT calculations can be performed either using precomputed integrals, direct integrals or density fitting. Many different functionals with or without exact exchange are available [21]. The functionals are represented in their mathematical form written by the syntax of the *Maple15* symbolic algebra system [22], which is used to generate both executable Fortran code and documentation. There are few recent improvements of the DF-KS code as well, which includes faster integral evaluation, integral caching, and faster evaluation of the exchange-correlation potential [21]. In this thesis, the DFT formed the basis for the exploration of fine grain parallelism for GPGPU accelerators.

5.3 DFT Architecture in Molpro

The *Kohn-Sham* density functional theory is designed with an iterative procedure until the energy converged within a certain threshold. The $df - ks$ *Molpro* input command initiate the DFT procedure. The energy is computed accurately by using three dimensional quadrature (grid) points and with their weights. One can define the parameters to generate the *grid points*, however the default parameter is usually sensibly sufficient. The three dimensional grids can be obtained by a subroutine called by DFT. The whole grid space is divided into chunks of grid points. This number of spatial integration points together is treated as a grid block in the DFT integration routines (default is 128 grid points per grid block). The size of each grid block is easily configurable from the *Molpro* input commands. Increasing the grid block size sometimes enhance the efficiency, specially for vector architecture machines, but leads to increased memory usage. Each grid block is assigned to a job and this job evaluates all the necessary DFT computations. A block of grid points, together with other parameters, like the basis sets, etc., is treated as the input to the DFT job. The results of all the jobs are later accumulated to form the final energies of each Self Consistent Field (SCF) iteration [12].

The steps for each SCF iteration is summarized below:

– The grid space is divided in to a number of job chunks.

– Each chunk has the size of the grid block size. By default, the grid block size is 128, but it can be freely configured.

– Each job chunk runs all the required DFT operations.

– Each job chunk is fully independent from the others and no communication is required among them.

– The results from all the job chunks are accumulated at the end.

There are four major algorithmic steps identified for a single job:

1. Evaluation of the basis set functions.
2. Calculation of the density matrix
3. Computation of density functionals
4. Accumulation of the exchange correlation matrix

The figure 5.1 illustrates the schematic architecture of a single SCF iteration of the density functional code. In the following, the four major algorithmic parts are explained briefly.

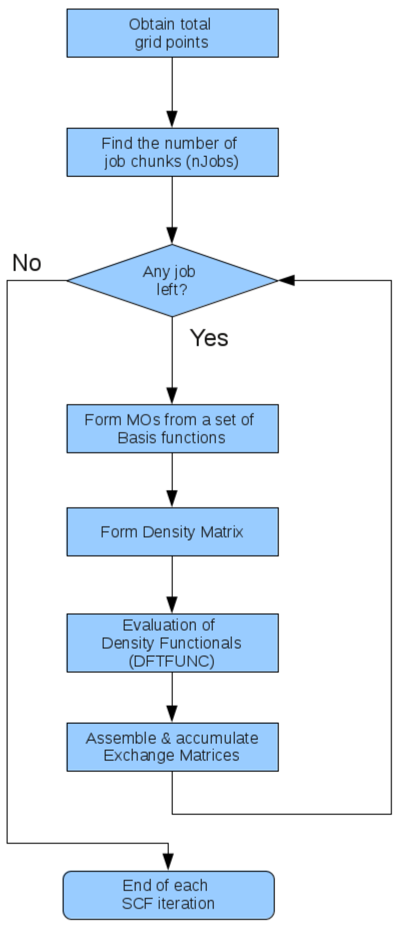

Obtain total grid points
Find the number of job chunks (nJobs)
No
Any job left?
Yes
Form MOs from a set of Basis functions
Form Density Matrix
Evaluation of Density Functionals (DFTFUNC)
Assemble & accumulate Exchange Matrices
End of each SCF iteration

Figure 5.1: Single SCF iteration of DFT

Evaluation of basis functions This part computes the basis function values $\chi_\mu(r_\lambda)$, the *atomic orbitals* (AOs) depending on the basis set used at grid points r_λ. *Molpro* input parameters defines the type of basis functions to be used. The derivatives of the AOs on the grid points are also calculated. Typically, the contracted Cartesian Gaussian basis function centered at A is given by [23]

$$\chi_\mu(x_\lambda, y_\lambda, z_\lambda) = (x_\lambda - A_x)^{a_x}(y_\lambda - A_y)^{a_y}(z_\lambda - A_z)^{a_z}\sum_\alpha c_{\alpha\mu}exp(-a_\alpha R^2_{A_\lambda}) \tag{5.1}$$

$$R^2_{A_\lambda} = (x_\lambda - A_x)^2(y_\lambda - A_y)^2(z_\lambda - A_z)^2 \tag{5.2}$$

where $c_{\alpha\mu}$ is the *contraction coefficient*, a_α is the *primitive Gaussian exponent*, and A_x, A_y and A_z are non-negative integers. The sufficiently small terms are skipped based on a threshold. This extensive screening efficiently optimizes the code in terms of speed.

Computation of electron density and its gradients The electron density $\rho(r_\lambda)$ and the gradient of the density $\nabla\rho(r_\lambda)$ are computed for a given first-order reduced density matrix *(1-RDM)*:

$$\rho(r_\lambda) = \sum_{\alpha,\beta} D_{\alpha\beta}\chi_\alpha(r_\lambda)\chi_\beta(r_\lambda) \tag{5.3}$$

where $D_{\mu\vartheta}$ is the density matrix in the AO basis set. The summation runs over the significant AOs.

Evaluation of DFT Functionals on grid The electron density, the gradient and the laplacian of the density are then formed in to exchange correlation functionals and their derivatives. The total exchange correlation functionals are calculated. The accumulated energy contribution is split according to the functionals used. The functionals to be used here is determined by the specification of the *Molpro* input parameters. A list of functional that are supported by *Molpro* can be found [12].

Accumulation of exchange correlation contribution Finally the exchange correlation contribution to the Fock matrix is calculated. the integrals $v^{xc}_{\mu\vartheta}$ are calculated numerically on grid as

$$v^{xc}_{\alpha\beta} = \sum_\lambda w_\lambda v_{xc}(r_\lambda)\chi_\alpha(r_\lambda)\chi_\beta(r_\lambda) \tag{5.4}$$

Here, λ labels grid points r_λ with grid weights w_λ. $\chi_\mu(r_\lambda)$ is the value of the basis function χ_μ at the grid point r_λ. The sum is over the batch of grid points λ.

5.4 Profiling of the DFT Module in Molpro

From the algorithmic point of view, four different major parts of the DFT code are identified. Those have already been briefly explained. We need to know the timing

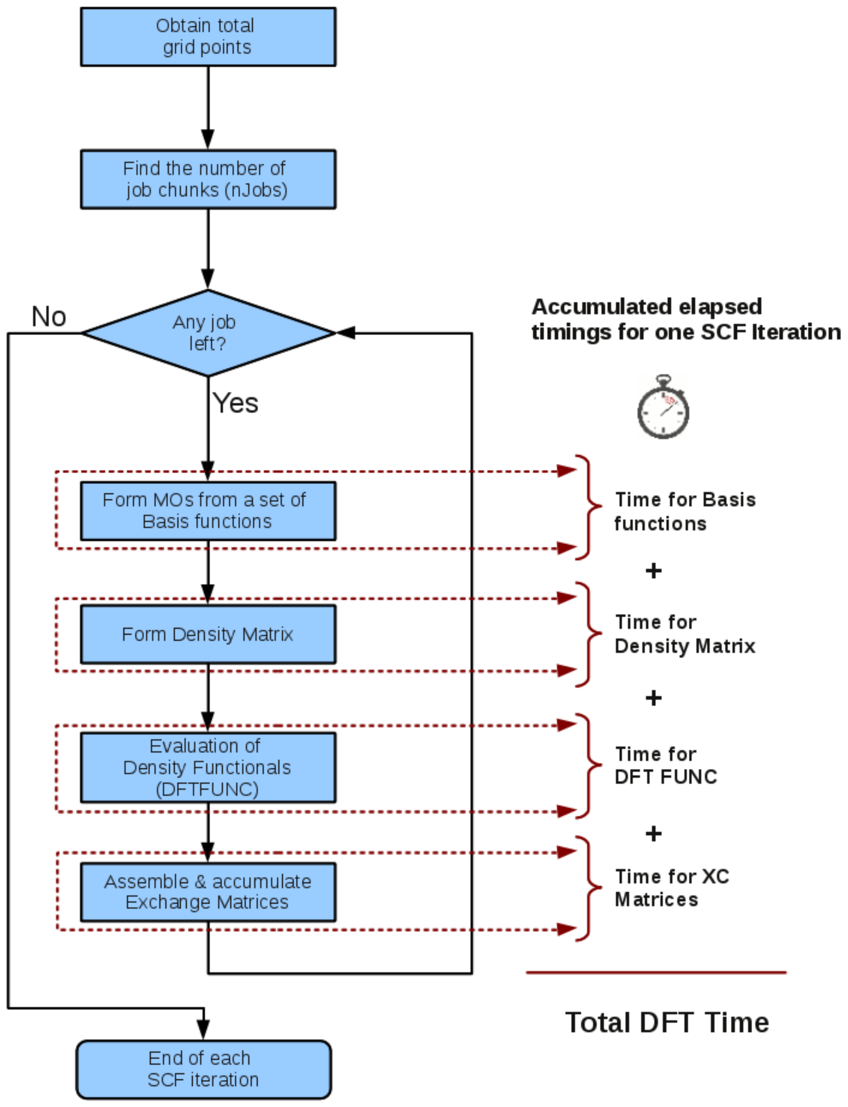

Obtain total grid points
Find the number of job chunks (nJobs)
Any job left?
No
Yes
Form MOs from a set of Basis functions
Form Density Matrix
Evaluation of Density Functionals (DFTFUNC)
Assemble & accumulate Exchange Matrices
End of each SCF iteration
Accumulated elapsed timings for one SCF Iteration
Time for Basis functions
+
Time for Density Matrix
+
Time for DFT FUNC
+
Time for XC Matrices
Total DFT Time

Figure 5.2: Single SCF iteration of DFT with timers

of each of these algorithmic parts to identify the most computationally expensive part of the code. A system timer is set wrapping all those four parts. This is explained in the figure 5.2.

The *Molpro* standard system timing function is used here, which produce timing results with sufficient resolution. The profiling is done for Polyvinyl fluorides (pv-20) $C_{40}F_{22}H_{62}$ which consists of 122 atoms. The cc-pVTZ orbital basis set (OBS) (2668 CGTOs) and TZVPP/JFIT (4228 GTOs, for BLYP functionals) basis set is used in this case. The profile benchmarking is run on a 12-core Intel(R) Xeon(R) X5680 processor *with single core* running with 3.33GHz. The total time required to finish the whole operations is 3477.44*sec*. This time includes the density fitting coulomb part, diagonalization, etc. apart from the DFT. The whole *df-ks* operation requires 13 SCF iterations to converge the result. The table 5.1

Table 5.1: DFT profiling - elapsed times for a single SCF iteration

Algorithmic Parts	**Elapsed Time**	**% of Total Elapsed Time**
Basis Set Evaluation	8.95	16.56
Density Matrices & Gradients	20.55	38.02
DFT Functionals	0.35	0.65
Accumulated XC Matrices	24.12	44.63
Time for DFT (total)	54.05	100.00

presents the elapsed time and the percentage of time consumed by the four major DFT parts in each single SCF iteration. The timing for only one single iteration is considered here. These timing results are almost same for all other SCF iterations.

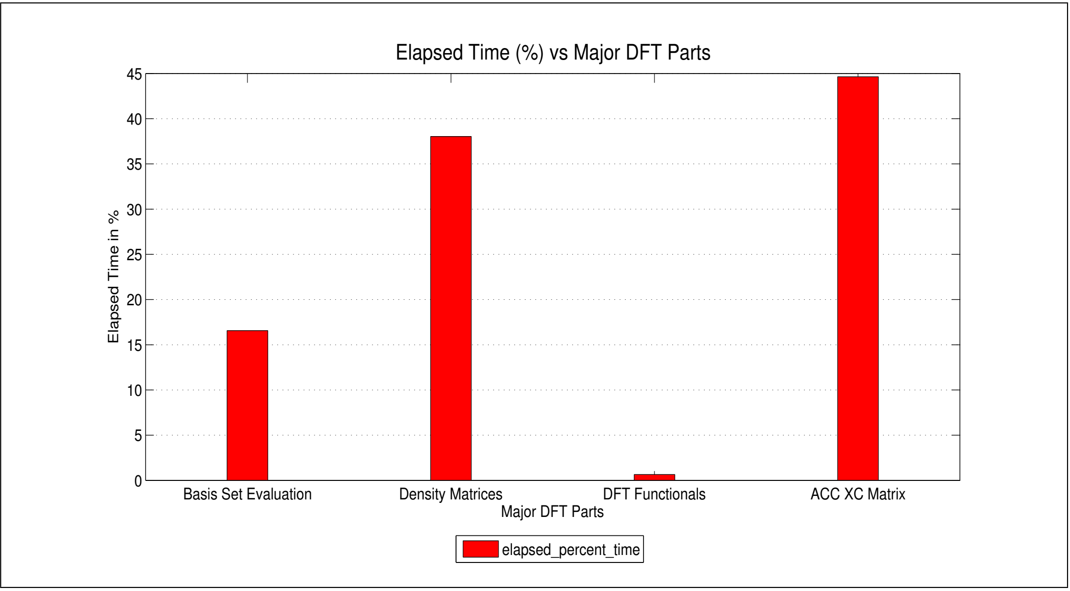


Figure 5.3: Percentage of elapsed time for each major algorithmic DFT parts

The figure 5.3 explains the timing behavior of each DFT part. The results show that the accumulated exchange correlation matrices (XC Matrices) and the evaluation of density matrices are the most time consuming steps. Here the exchange correlation matrices and the evaluation of density matrices take nearly 45% and 40% of total DFT time, respectively. The third expensive part is the evaluation of basis sets, which takes nearly 17% of total DFT time. The timing of the DFT Functionals is not trivial here as it takes less than 1% of computational time. This is almost the same case if a different functional, like *pbe* is used. The block size of 128 is used in this profiling. Several other profiling with bigger grid block sizes were performed, which suggested that the 128 block size is the optimum for running DFT calculations in one single CPU.

We take this results as a basis and continue the analysis of these most computationally expensive parts. Our intention now is to look the existing algorithm and the implementation in Molpro and prepare a suitable implementation in GPU. Next chapter will explain the details of the GPU implementation.

Chapter 6

GPU Implementation

Contents

6.1 Introduction

In this chapter, the algorithm of the two most computationally expensive DFT parts (density matrices and accumulated exchange correlation) will be analyzed and a fine grained parallel algorithm is proposed. Fine grained threads are the important characteristics of parallel execution in GPUs. These two DFT parts are then implemented with CUDA (Compute Unified Device Architecture) as a hybrid CPU-GPU structure.

6.2 Evaluation of Density Matrices

The electron density $\rho_(r_\lambda)$, the gradient of the density $\nabla\rho_(r_\lambda)$, the kinetic energy term τ_λ and the Laplacian of the electron density υ_λ are computed for a given first-order reduced density matrix *(1-RDM)* as explained in the previous chapter. Density matrices are formed for both closed and open shell systems in this code. An open shell configuration represents a partially filled valence shell with electrons, whereas closed shell configuration is completely filled valence shell [24]. This closed or open shell calculation depends on the molecular structure to be analyzed and is set in the *Molrpo* input parameter list. The existing density evaluation subroutine is redesigned and adapted to comply with the CUDA architecture. The main DFT routine calls this density subroutine as follows,

1. Initialize the data structure and prepare the memory for closed shell density calculation
2. Evaluate the closed shell density
3. Check if open shell calculation is required or not
    – Prepare memory and initialize the open shell data
    – Evaluate open shell density (just like the closed shell, with different arguments)
    – Check if there are any spin density (α/β) present
        – Transform spin (α/β) densities to closed/open shell densities
4. Keep the densities and their gradients in memory for future calculation and release other unnecessary memories.

The state of art code for evaluation of density matrices in *Molpro* is complex and the data used is distributed over several data structures (classes). This is inconvenient for the management of the memory addresses on the GPU. Therefore, a simple data structure is implemented. All the nested C++ classes are removed and only the required matrix data are kept and managed in a single C++ class, named *dftiCuda::FDftiJob*. This C++ class holds all the persistent information and functions for evaluating *dfti* on a single block of grid points. *dftiCuda* is the *namespace* used here to define a new code path along the standard code in *Molpro*. Keeping all the data and functions in a single C++ class, helps to keep track of the memory addresses both on the host and on the GPU. This reduces the complexity and improve the readability of the code as well.

The block diagram in figure 6.1 presents the structure of the routine to evaluate the density matrix. Depending on the required derivative order, whether to use orbitals or whether to make the *kinetic energy* term (*Tau*), τ, five different code paths with four different computationally expensive parts are identified. They are:

– *MakeBxRho*: Transforms the basis function values to contraction density matrix
– *MakePhi*: Transforms the basis function values and their gradients to occupied orbitals
– *AccN* operation: It is a BLAS *daxpy* operation to form density, gradient, etc., if no orbitals are used

– *DotN* operation: The N number of dot products to form density, gradient, etc., in case the orbitals are used

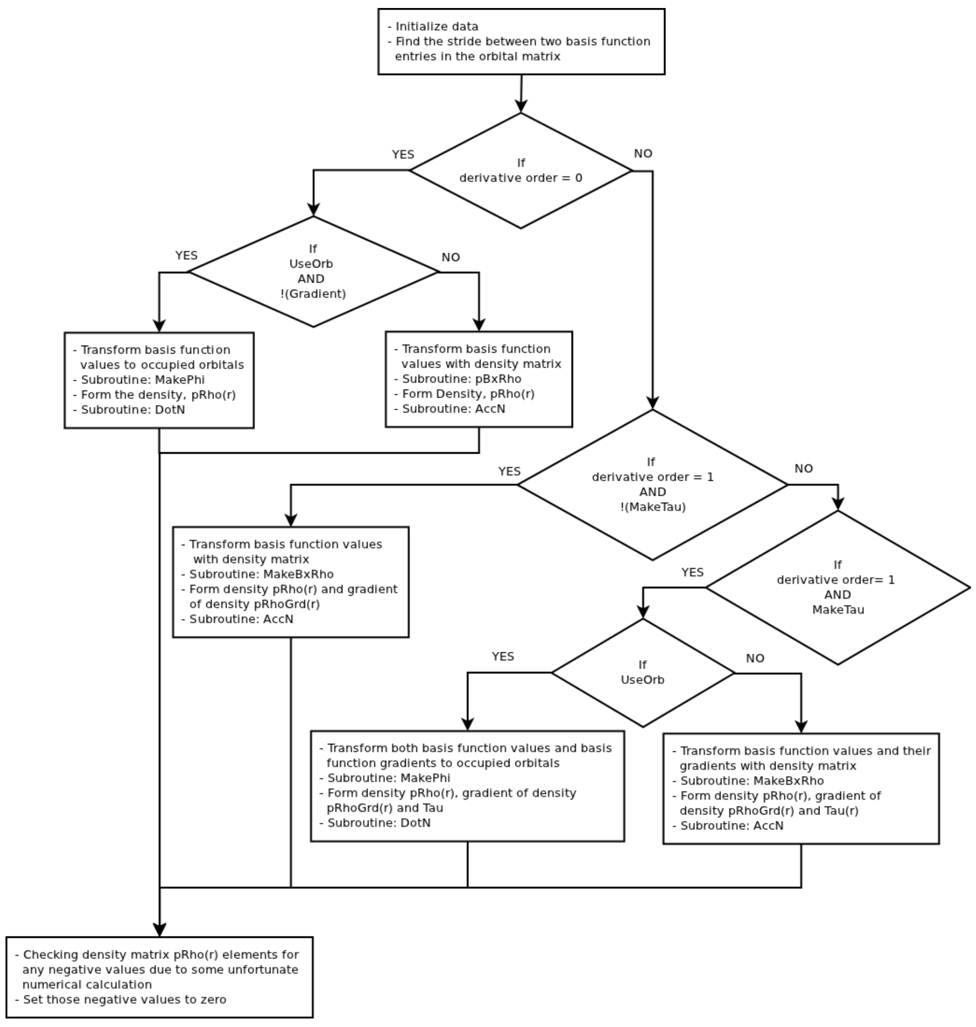
In this next section, we will analyze these parts in detail and find the scope for fine grain parallelism.

6.2.1 MakeBxRho

A contracted matrix is formed. If occupied orbitals are presented in during the total process, the contraction of matrix is formed from the occupied orbitals*(pOccOrb)*, otherwise it is created from the reduced density matrix *(pRdm)*. The steps for computing the contraction matrix are as follows:

1. If orbitals are not used in the computation process

   (a) Unpack the reduced density matrix and compress it.



- Initialize data
- Find the stride between two basis function entries in the orbital matrix
YES
If derivative order = 0
NO
YES
If UseOrb AND !(Gradient)
NO
- Transform basis function values to occupied orbitals
- Subroutine: MakePhi
- Form the density, pRho(r)
- Subroutine: DotN
- Transform basis function values with density matrix
- Subroutine: pBxRho
- Form Density, pRho(r)
- Subroutine: AccN
YES
If derivative order = 1 AND !(MakeTau)
NO
- Transform basis function values with density matrix
- Subroutine: MakeBxRho
- Form density pRho(r) and gradient of density pRhoGrd(r)
- Subroutine: AccN
YES
If derivative order= 1 AND MakeTau
YES
If UseOrb
NO
- Transform both basis function values and basis function gradients to occupied orbitals
- Subroutine: MakePhi
- Form density pRho(r), gradient of density pRhoGrd(r) and Tau
- Subroutine: DotN
- Transform basis function values and their gradients with density matrix
- Subroutine: MakeBxRho
- Form density pRho(r), gradient of density pRhoGrd(r) and Tau(r)
- Subroutine: AccN
- Checking density matrix pRho(r) elements for any negative values due to some unfortunate numerical calculation
- Set those negative values to zero

Figure 6.1: Evaluation of the density matrix

   (b) Contraction matrix is computed from this compressed reduced matrix.

2. If orbitals are present

   (a) Compress the occupied orbitals.

   (b) Form the contraction matrix from the occupied orbitals.

The complete implementation procedure is explained below.

6.2.1.1 If orbitals are not used:

If occupied orbital matrix is not present, the first order reduced density matrix (*pRdm*) is used for contraction.

Unpack and Compress Reduced Density Matrix Typically, this first order reduced density matrix (*pRdm*) is triangular and of dimension $(nBf \times nBf)$. Here, *nBf*, is the number of basis functions used. This input triangular matrix *pRdm* needs to be transformed from triangular to square matrix, known as unpacking of *pRdm*. As the extensive screening is used during the process of *evaluation of the basis function*, this square matrix is compressed to a matrix of dimension of $(nMap \times nMap)$. Here *nMap*, represents the number of basis functions kept after the screening. For a typical test case with cc-pVDZ basis set, number of basis function, nBf, is equal to 580 and if screening is applied on the molecules then $nMap$ typically is less than the nBf.

This transformation and compression can be efficiently done in CUDA. The required input matrix size is $(nBf \times nBf)$ which needs to be transfered to the GPU memory from the host. And a size of $(nMap \times nMap)$ output matrix *pRdm1_gpu*, is required to allocate in the GPU. This output matrix is kept on the GPU, as it is required in the next step. This eliminates the latency of further memory transfers. A subroutine, called *makeBxRho*, is written which acts like a wrapper from *Molpro* to call the CUDA kernel for these operations. An *unpackDensityMatrix* CUDA kernel is created, which is called from the *makeBxRho* subroutine. A number of $(nMap \times nMap)$ CUDA threads are required to map with the each element of compressed matrix. The number of CUDA threads are needed to be power of 2, as explained in the section *Programming Model* in chapter 4. The value of *nMap* is determined in runtime and it is not assured that the value of *nMap* is a power of 2. So an in-lined efficient small subroutine is written to find the next nearest value which is a power of 2. This is done by the right shift operator ($\gg$), listing 6.1. That means, the matrix dimension is padded to a value, which is a power of 2. The CUDA threads, that have numbers smaller than $nMap$ will work on the corresponding matrix elements. The rest of the threads are kept idle.

Listing 6.1: Pseudocode for calculating next power of 2 of x

```
inline int nextPow2( int x ) {
  --x;
  x |= x >> 1;
  x |= x >> 2;
  x |= x >> 4;
  x |= x >> 8;
  x |= x >> 16;
  return ++x;
};
```

Computing the contraction matrix (*pBxRho*) from compressed reduced matrix After getting the compressed matrix, a matrix multiplication is required to calculate contraction matrix (*pBxRho*). The state of the art *Molpro* uses a different *BLAS* library which is very efficient on single core and multi-core architecture. We have decided to use the *CUBLAS* library for any linear algebra calculation [25]. For the sake of numerical accuracy of molecular simulation, all the matrix elements are of double precision. The level-3 *CUBLAS* routine *cublasDgemm* is used to do the matrix-matrix multiplication. The output matrix is kept in the GPU memory for further calculation. The temporary memory spaces and the initial input reduced matrix is released from the GPU memory. The number of floating point operations (FLOPs) are then $2 \times nPts \times nMap \times nMap$. Here *nPts* is equal to the number of grid points times the number of derivative components of each element (*nComp*).

Listing 6.2: Pseudocode for creating contraction matrix without occupied orbitals

```
{
  ...
  int nMap_x = nextPow2(nMap);
  blocksPerGridX = (nMap_x + threadsPerBlockX - 1) / threadsPerBlockX;
  blocksPerGridY = (nMap_x + threadsPerBlockY - 1) / threadsPerBlockY;

  dim3 dimBlock(threadsPerBlockX, threadsPerBlockY);
  dim3 dimGrid(blocksPerGridX, blocksPerGridY);

  unpackDensityMatrix<<<dimGrid_1, dimBlock_1>>>(pRdm1_gpu, pRdm_gpu,
                                                  pMap_gpu, nMap);
  cublasDgemm ('n', 'n', nPts, nMap, nMap, 1.0, pOrbVal_gpu,
               Stride, pRdm1_gpu, nMap, 0 , pBxRho_gpu, nPts);
  ...
}
```

6.2.1.2 If orbitals are used:

If occupied orbital matrix is present, the contraction will be done on this.

Compress the occupied orbitals The occupied orbitals (*pOccOrb*) matrix has the dimension of $(nOcc \times nBf)$. *nOcc* represents the number of occupied orbitals and *nBf* is the number of basis functions, as explained before. Due to the screening phase in *evaluation of basis functions*, not all the nBf is used. $nMap$ represents the number of basis functions that are kept after the screening. Screening removes the certain basis functions that will not have any effect on the total energy. The screening threshold is set in the *Molpro* input parameters when running the DFT calculation. So this orbital matrix *pOccOrb* has to be compressed to ($nOcc \times nMap$) matrix. This also can be done very efficiently in CUDA. The input occupied orbitals (*pOccOrb*) matrix is transfered to GPU memory from host memory. A CUDA kernel, *compressInpOrbital* is created to do this compression on the GPU. The number of occupied orbitals, *nOcc*, and the number of kept basis functions, $nMap$, are only determined during the runtime. So the next nearest value, that is a power of 2, is calculated (6.1), i.e., $nOcc_x$ and $nMap_x$ for *nOcc* and $nMap$ respectively. The ($nOcc_x \times nMap_x$) CUDA threads are created to do the compression efficiently and can avoid race conditions among the threads. No floating point computations are done here but a lot of memory transformations are done inside.

Computing the contraction matrix (*pBxRho*) from the occupied orbitals
Matrix multiplication is used to compute the contraction matrix (*pBxRho*). Similarly, the level-3 *CUBLAS* routine *cublasDgemm* is used to calculate the matrix-matrix multiplication. The required transposition of the matrix is done through the *cublasDgemm* routine. Two *cublasDgemm* operations are required. The contraction matrix is kept on the GPU and all other temporary memory spaces and the occupied orbital matrices are released from the GPU explicitly. The total number of floating point operations (FLOPs) are then $4 \times nPts \times nMap \times nOcc$.

Listing 6.3: Pseudocode for creating contraction matrix with occupied orbitals

```
{
  ...
  int nMap_x = nextPow2(nMap);
  int nOcc_x = nextPow2(nOcc);
  blocksPerGridX = (nMap_x + threadsPerBlockX - 1) / threadsPerBlockX;
  blocksPerGridY = (nOcc_x + threadsPerBlockY - 1) / threadsPerBlockY;

  dim3 dimBlock(threadsPerBlockX, threadsPerBlockY);
  dim3 dimGrid(blocksPerGridX, blocksPerGridY);

  compressInpOrbital<<<dimGrid, dimBlock>>>(pOrb1_gpu, pOccOrb_gpu,
                                  pMap_gpu, (int)nMap, (int)nOcc);
  ...
  cublasDgemm ('n', 't', nPts, nOcc, nMap, 1.0, pOrbVal_gpu, Stride,
               pOrb1_gpu, nOcc, 0 , devPtr_pDummy, nPts);
  cublasDgemm ('n', 'n', nPts, nMap, nOcc, 1.0, devPtr_pDummy, nPts,
               pOrb1_gpu, nOcc, 0 , pBxRho_gpu, nPts);
  ...
}
```

6.2.2 MakePhi

This subroutine transforms the matrix that contains basis functions values to occupied orbital matrix. Typically, after the extensive screening phase in *basis function evaluation*, the dimension of the orbital matrix is then $(nPts \times nMap)$. Here, $nPts$ represents the total number of grid points (*nGridPt*) times the number of derivative components of each element (*nComp*). This routine transforms orbital matrix to a dimension of $(nOcc \times nPts)$. The whole procedure includes following steps

Compress the input occupied orbital matrix As explained before, the occupied orbitals (*pOccOrb*) matrix has the dimension of $(nOcc \times nBf)$. This matrix has to be compressed to $(nOcc \times nMap)$ matrix. This transformation is done very efficiently with the CUDA kernel *compressInpOrbital* as explained in page 52. An efficient mapping of the threads and the matrix elements are implemented to avoid race conditions among the threads.

Computing the orbital matrix ($\phi_i(r)$) Similarly matrix multiplication is required to calculate the *phi*(ϕ). The level-3 *CUBLAS* routine *cublasDgemm* is used to calculate the matrix-matrix multiplication. The required transposition of the matrix is done through the *cublasDgemm* routine. As further computation will be done on this matrix, this is kept in the GPU memory and all other temporary memory spaces are released explicitly from the GPU. The number of floating point operations (FLOPs) are then $2 \times nOcc \times nMap \times nPts$.

6.2.3 Form Density Matrix and its Gradient

The density matrix is then formed from the orbital matrix. The gradient, the laplacian of the density matrix is calculated, if it is required:

$$\rho(r) = \sum_i \phi_i(r)\phi_i(r) \tag{6.1}$$

$$\nabla\rho(r) = 2\sum_i \phi_i(r)(\nabla\phi_i(r)) \tag{6.2}$$

$$\tau(r) = \sum_i [\nabla\phi_i(r)]^2 \tag{6.3}$$

Two subroutines are written to do the calculations. The choice of subroutines depends on how the orbital matrix is formed.

6.2.3.1 AccN

This subroutine consists of a series of *BLAS daxpy* operations running over the basis functions and grid points. A new kernel *Add2-gpu* is created. This calculates the element-wise multiplication of two vectors and then the result is added with the elements of an output matrix that is calculated previously. This addition is run over all the grid points in a grid block size. As the grid block size is set as a power of 2, the mapping of the CUDA threads to the corresponding vector elements make efficient. Later this resultant matrix element is accumulated over the number of kept basis functions (*nMap*). The accumulation process can be included in *Add2-gpu* kernel. But this results race conditions among the CUDA threads. One option is to use the *AtomicAdd* operations to solve the race condition. But this mechanism keeps the CUDA threads waiting until other CUDA threads finish updating their values in GPU memory location. Another option is that, we keep this accumulation outside the kernel and call this kernel *nMap* times, which in turns aggregate the values together. This procedure increases the speed by a factor of 2, compared to the using *AtomicAdd* operation.

6.2.3.2 DotN

This subroutine consists of a series of dot products running over the basis functions and grid points. The dot products are done by *CUBLAS Ddot* subroutine, it computes the dot product of two double precision vectors. Later this result of the dot product is multiplied by a scalar factor and stored. This is running over all the kept basis functions (*nMap*). The dot products are running asynchronously in the GPU.

GPU memory requirements

In the following the total memory requirements for one SCF calculation of the density matrices and its gradients are summarized.

Derivative order = 0 Total memory requirement is

$$\begin{aligned} Mem &= [nGridPt(2 \times nMap + nOcc + 1) + \\ &\quad nBf(nBf + nOcc) + nMap(nMap + nOcc)] \times \\ &\quad sizeof(Double) + nMap \times sizeof(Integer) \end{aligned} \tag{6.4}$$

Derivative order > 0 Total memory requirement is

$$\begin{aligned} Mem &= [nGridPt(2 \times nMap \times nComp + nOcc + 4) + \\ &\quad nBf(nBf + nOcc) + nMap(nMap + nOcc)] \times \\ &\quad sizeof(Double) + nMap \times sizeof(Integer) \end{aligned} \tag{6.5}$$

Derivative order > 0 **and with Tau , Upsilon** Total memory requirement is

$$\begin{aligned} Mem &= [nGridPt(2 \times nMap \times nComp + nOcc + 6) + \\ &\quad nBf(nBf + nOcc) + nMap(nMap + nOcc)] \times \\ &\quad sizeof(Double) + nMap \times sizeof(Integer) \end{aligned} \tag{6.6}$$

Here

– nGridPt: Number of grid point in a grid block, grid block size. Typically for calculation in GPU it is set to 4096, 8192, etc.

– nBf: Number of basis functions. For cc-pVDZ basis set, it is typically 580.

– nMap: Number of basis functions kept after screening. Only determined during runtime, however less than nBf

– nComp: Number of components for different derivative order. For derivative order 0, 1 and 2, nComp is 1, 4 and 10 respectively.

– nOcc: Number of occupied orbitals

6.3 Evaluation of The Exchange Correlation Matrix

As explained in chapter 5, this routine calculates the *exchange correlation contributions* to the *Fock* matrix. The typical equation is:

$$v^{xc}_{\alpha\beta} = \sum_{\lambda} w_\lambda v_{xc}(r_\lambda)\chi_\alpha(r_\lambda)\chi_\beta(r_\lambda) \tag{6.7}$$

and this is calculated numerically on the grid, in practice. This calculation depends on the previous functionals used. Three different cases are handled here in this code:

1. LDA (*Local Density approximation*) case (with derivative order = 0)
   – For pure non-negative integrals, (i.e., from functionals).
   – For any negative integrals present.
2. GGA (*Generalized Gradient Approximation*) case with-out the kinetic term (τ).

The architecture is briefly explained in the figure 6.2. The whole implementation of the procedures is explained in the following section.

6.3.1 LDA (*Local Density approximation*) Case

In this LDA (*Local Density approximation*), case there are two different but yet similar calculations required. The procedure depends on the fact that whether there is any negative values present in the DFT Functionals output. This functional output is known in the code as $pVdRho$, which is, in fact the first derivative of the functionals with respect to the density $\rho(r)$. Initially every element of this $pVdRho$ is checked for any negative values. The number of elements in this $pVdRho$ is equal to the size of the grid block, which is represented by the value *nGridPt*. This inspection is done on the CPU side in the host code, as it is not worthy to do in the GPU.

With pure non-negative integrals With pure non-negative integrals, i.e., all the elements in *pVdRho* are positive scalar, the calculation of the exchange correlation contribution is very simple. The procedure includes four simple steps, explained below, and all of them are efficiently done on the GPU side:

1. Find the negative weighted density functional values for each grid point and then take the square root of each element of the weighted density functional values

   This can be parallelized very efficiently on the GPU. A kernel, called *makeSqrtWeightedDFTFUNC* is written, for this purpose. The number of elements in each vectors is equal to the *nGridPt*. *nGridPt* number of CUDA threads are created and each thread calculates the negative weight by multiplying the corresponding vector elements and then do the standard square root operations. The typical number of floating point operation is $24 \times nGridPt$.

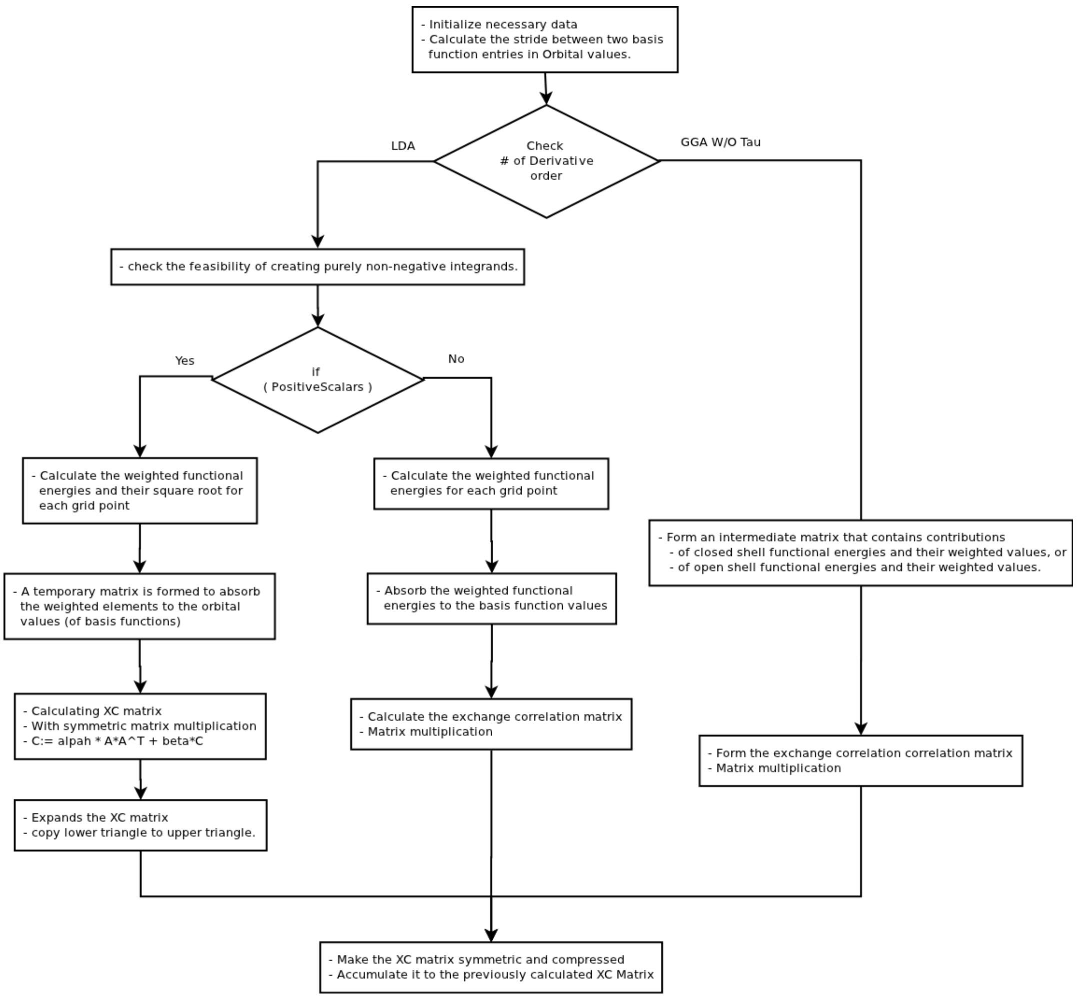

- Initialize necessary data
- Calculate the stride between two basis function entries in Orbital values.
Check # of Derivative order
LDA
GGA W/O Tau
- check the feasibility of creating purely non-negative integrands.
if (PositiveScalars)
Yes
No
- Calculate the weighted functional energies and their square root for each grid point
- Calculate the weighted functional energies for each grid point
- Form an intermediate matrix that contains contributions
- of closed shell functional energies and their weighted values, or
- of open shell functional energies and their weighted values.
- A temporary matrix is formed to absorb the weighted elements to the orbital values (of basis functions)
- Absorb the weighted functional energies to the basis function values
- Calculating XC matrix
- With symmetric matrix multiplication
- C:= alpah * A*A^T + beta*C
- Calculate the exchange correlation matrix
- Matrix multiplication
- Form the exchange correlation correlation matrix
- Matrix multiplication
- Expands the XC matrix
- copy lower triangle to upper triangle.
- Make the XC matrix symmetric and compressed
- Accumulate it to the previously calculated XC Matrix

Figure 6.2: Evaluation of the exchange correlation matrix

The specification for calling the GPU kernel depends on the number of grid point values, *nGridPt*. Typically, *nGridPt* is a value, which can be expressed as a power of 2, additional padding to the vectors is not necessary. The result is stored in a temporary matrix, named as *pFac*.

2. Absorb these square rooted elements to the orbital values (the values of the basis functions)

   This is also can be parallelized efficiently. A temporary matrix, (*pRhoZkWt*), is created to store the results. A GPU kernel, (*makeAbsorbKernel*) is written to execute it on the GPU. This is an element wise multiplication between the previously calculated weighted functional energies (*pFac*) and the basis function values, that are in the matrix (*pOrbVal*). The for loops of the CPU code are unrolled and mapped to GPU threads. The number of CUDA threads depends on the grid block size (*nGridPt*) and the number of kept basis functions (*nMap*) during the screening process of basis function evaluation. *nMap* is only known during the runtime and the value is not always equal to the power of 2. So $nMap_x$, the next closest power of 2 of *nMap* is computed. The total number of CUDA threads is $nMap_x \times nGridPt$ and distributed to a 2D thread structure. The total number of floating point operations is $nMap \times nGridPt$.

3. Calculating exchange matrix (pXC), by symmetric matrix multiplication

   In *Molpro*, this is done by a *BLAS* subroutine for symmetric rank k update, called *DSYRK*. Here, the corresponding *cublasDsyrk* subroutine is used. The *cublasDsyrk* does the following symmetric matrix multiplication.

$$C = \alpha * A * A^T + \beta * C \tag{6.8}$$

   Here, A^T represents the transpose of matrix A and α, β represents the double precision scalar multiplier. The previously calculated (*pRhoZkWt*) matrix is represented by the matrix A, in equation 6.8. In our case, the β is set to zero, so the number of FLOPs will be $2 \times nMap \times nGridPt \times nMap$.

4. Expand exchange matrix, by copying the lower triangle elements to the upper triangle in the output matrix.

   A GPU kernel, *makeExpand()* is written to parallely copy the lower triangle of a matrix to upper triangle. No floating point operation is necessary except memory copy (transformation) on the GPU device. The specification for this GPU kernel calling depends on the number of kept basis functions, $nMap$, and is only determined during runtime. As usual the next power of 2, $nMap_x$ is calculated and ($nMap_x \times nMap_x$) number of threads are created and the corresponding elements are copied from lower triangle to upper triangle of the exchange correlation matrix.

If negative integrals are present If there are any negative integrals presents, i.e., at least one element in *pVdRho* is a negative scalar, the calculation of the exchange correlation contribution is done by three steps that are explained below, and which are efficiently done on the GPU.

1. Calculate the weighted density functional values *pVdRhoWt* for each grid point

   This is an element wise multiplication of two vectors running over all the grid points. A GPU kernel *makeWeightedDFTFUNC* is written to do this multiplication in parallel on the GPU. *nGridPt* number of CUDA threads are created to do the multiplication. The total number of floating point operations is *nGridPt*.

2. Form *pVdRhoOrb* by multiplying the previously calculated weighted *pVdRhoWt* with the evaluated basis function values (*pOrbVal*).

   This is also an element wise multiplication that runs over all the grid points for each *iMap*. *iMap* contains the values from 0 to $(nMap-1)$. Another GPU kernel, called *makepVdRhoOrb()* is written to run this multiplication parallely on the GPU. *nMap* is padded to next power of 2 values, $nMap_x$. A total number of $(nGridpt \times nMapx)$ threads is created. Each thread will do one single multiplication, which yields total $(nGridpt \times nMap)$ floating point operations (FLOPs).

3. Exchange correlation matrix is formed

   This is formed by matrix matrix multiplication between *pVdRhoOrb* and *pOrbVal*. The *pVdRhoOrb* matrix needs to be transposed before the multiplication. *cublasDgemm* is used here. The number of floating point operation (FLOPs) is then $2 \times nMap \times nGridPt \times nMap$.

6.3.2 GGA (*Generalized Gradient Approximation*) Case

For this GGA case without the kinetic term (τ), the exchange correlation matrix becomes like as follows,

$$\begin{aligned} k_{\mu\nu}(r) &= \mu(r)zk(r)\nu(r) + 2\Xi(r)(\nabla\rho(r))(\nabla\mu(r)\nu(r)) \\ &= \mu(r)zk(r)\nu(r) + 2\Xi(r)(\nabla\rho(r))([\nabla\mu(r)]\nu(r) + \mu(r)[\nabla\nu(r)]) \end{aligned} \tag{6.9}$$

where $\Xi = \frac{dzk}{d\sigma}$. The last two terms, $[\nabla\mu(r)]\nu(r) + \mu(r)[\nabla\nu(r)\ ]$, are symmetrizing combinations of each other. So it is evaluated only for one of them and then later multiplied by two. The non-symmetric exchange matrix resulting from this is then explicitly symmetrized afterwards to account for the effect of the other term. The equation becomes

$$\begin{aligned} k^*_{\mu\nu}(r) &= \mu(r)zk(r)\nu(r) + 4\Xi(r)(\nabla\rho(r))([\nabla\mu(r)]\nu(r)) \\ k^*_{\mu\nu}(r) &= [\mu(r)zk(r) + 4\Xi(r)(\nabla\rho(r))[\nabla\mu(r)]]\nu(r) \end{aligned} \tag{6.10}$$

Here $k^*_{\mu\nu}(r)$, means that it needs explicit symmetrization later. In the open-shell case the above formulas have an additional term

$$\begin{aligned} \text{for the closed-shell exchange:} &\ + 2\Xi_{co}(\nabla \rho_o(r)) \\ \text{for the open-shell exchange:} &\ + 2\Xi_{co}(\nabla \rho_c(r)) \end{aligned} \tag{6.11}$$

The whole process is done in two steps.

1. Form an intermediate matrix
   - For closed shell case
     An intermediate matrix, called *pLmu*, is formed which represents $k^*_{\mu\nu}$ in equation 6.10. A GPU kernel (*makepLmuKernel*) is written to compute this intermediate matrix for the closed shell molecular system. *nGridPt* number of CUDA threads are created during this kernel call. Each thread will do 3 additions and 7 multiplications, in total 10 floating point operations. So the total number of FLOPs is $10 \times nGridPt$. The mapping between the threads and the matrices elements in the global memory is done carefully. A stride of *nGridPt* is used, which is a multiple of 16 (half warp size), to avoid bank conflicts.
   - For open shell case
     The intermediate matrix, *pLmu*, will have two additional terms, with the close shell matrix, as explained in equation 6.11. Similarly another GPU kernel (*makepLmuOpenShellKernel*) is written to include this additional term. *nGridPt* number of CUDA threads are created, and a stride of *nGridPt* is used here too, to avoid the bank conflicts. The resultant matrix is stored in the global memory for further operations. Each thread will execute 2 additions and 6 multiplications. So the total number of FLOPs is $8 \times nGridPt$.

2. Form the exchange correlation matrix
   The level 3 CUBLAS double precision matrix multiplication subroutine, *cublasDgemm*, is used here to formulate the exchange correlation terms. The *pLmu* matrix needs to be transposed before the multiplication. This matrix transposition is done explicitly during the multiplication process with out any computational overhead. The number of floating point operations (FLOPs) calculated is $2 \times nMap \times nGridPt \times nMap$.

6.3.3 Form the Symmetric Exchange Matrix

The calculated exchange matrix needs to be symmetric. Then it is compressed to triangular matrix and is accumulated to the the previously calculated Fock matrix. A GPU kernel *compressXCMatrix* is written to serve this purpose. The number of elements in the exchange matrix is $nMap \times nMap$. As *nMap* is not a power of 2, the matrix dimensions are padded to the next power of 2 $nMap_x$, from *nMap*.

This allows the mapping of each matrix element to a single thread. The previously calculated Fock matrices are kept in the GPU memory, so expensive memory transfers from host to GPU memory, and vice versa, are avoided. Each thread will do 2 additions and 1 multiplications, in total 3 floating point operations. The total number of FLOPs is $3 \times nMap \times nMap$.

GPU memory requirements

Total memory requirements for the complete calculation of the exchange matrix in one SCF iteration is

$$\begin{aligned} Mem &= [nMap \times nMap + nGridPt \times (nMap + 4)] \times \\ &\quad sizeof(Double) + nMap \times sizeof(Integer) \end{aligned} \tag{6.12}$$

Here

– nGridPt: Number of grid point in a grid block, grid block size. Typically for calculation in GPU it is set to 4096, 8192, etc.

– nMap: Number of basis functions kept after screening. Only determined during runtime, however less than nBf

6.4 Intermediate Routines

Apart from these two expensive parts, two other subroutines are ported to the GPU to accelerate the overall calculation.

6.4.1 Form Sigma σ

An intermediate sigma matrix σ_{ij} is formed only for the gradient case, i.e., the derivative order is greater than 0.

$$\sigma_{AB}^{ij} = \nabla\rho_A^{ij} * \nabla\rho_B^{ij} \tag{6.13}$$

This computation can also be mapped to the GPU. A kernel (*formSigmaGPU*) is created to do this calculation. No extra GPU memory space is required neither any host to GPU memory copy, since the density matrices are already in the GPU memory. The total number of floating point operations is equal to $5 \times nGridPt$, as each thread will do 3 multiplications and 2 additions.

6.4.2 Transformation of Spin Density

If the input orbitals for density matrix contains spin α/β orbitals, then a transformation is necessary from spin densities to closed/open shell densities. In the original CPU code, it is implemented with couple of for loops which are running through all the density matrix (or gradient of density matrix) elements. In the GPU code, those for loops are unrolled and each elements are assigned to the CUDA threads to do the necessary transformations. The number of CUDA threads depends on the number of elements of the matrix. For only the density matrix (without the gradient), the number of CUDA threads required is equal to the block size ($nGridPt$). For a gradient case, it is $4 \times nGridPt$. The number of Floating point operation is $2 \times nGridPt$ (for density only) and $2 \times 4 \times nGridPt$ for gradient case. No additional GPU memory space is required, as the corresponding data are already in the GPU.

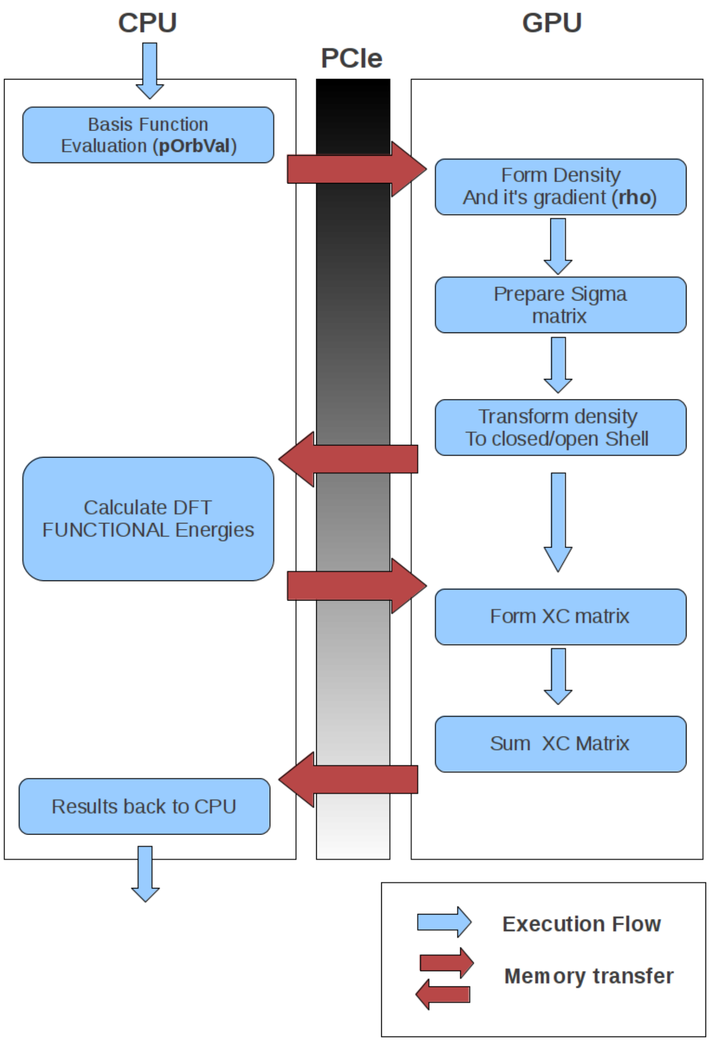

CPU
PCIe
GPU
Basis Function Evaluation (pOrbVal)
Form Density And it's gradient (rho)
Prepare Sigma matrix
Transform density To closed/open Shell
Calculate DFT FUNCTIONAL Energies
Form XC matrix
Sum XC Matrix
Results back to CPU
Execution Flow
Memory transfer

Figure 6.3: Hybrid CPU-GPU architecture

6.5 Hybrid (CPU-GPU) Implementation

Figure 6.3 explains the hybrid CPU-GPU implementation of the DFT code. The evaluation of the basis function is done on the CPU side and the calculated result is stored in a matrix, called *pOrbVal*. This matrix with other necessary data is transfered to the GPU memory. Then the density and their gradients (if necessary) are calculated on the GPU. The results are stored in the GPU memory. The sigma matrices (for closed/open shell) are formed from the density and gradient of the density matrices. If any transformation from spin to closed/open shell density is required, this is also done on the GPU. It eliminates the CPU-GPU memory transfer overhead. The calculation of the DFT functional energies is done on the CPU side. The resultant density matrices and the other corresponding data needs to be copied back to the CPU. A copy of this data remains in the GPU memory as they are required later. The energy functionals are efficiently computed on the CPU. Subsequently the calculated energy functionals are transfered to the GPU. This memory copy back and forth has a large overhead in the total computation process. Next the exchange correlation energy contribution to the Fock matrix are calculated on the GPU. This requires the density matrices, the energy functionals etc.. Keeping the density matrices, the grid points, etc, on the GPU, we could reduce the memory transfer overhead. Later the exchange matrix elements are summed together with the previous calculated exchange matrix. The accumulated exchange matrix is kept in the GPU memory and only transfered to the CPU after one full SCF iteration.

Chapter 7

A Multi-GPU Wrapper for Accelerated Density Fitting Mϕller - Plesset Perturbation Theory

Contents

7.1 Introduction

The *BLAS* (Basis Linear Algebra Subprograms) is a bundle of standard subroutines for basic vector and matrix operations [26]. *CUBLAS* is an implementation of the *BLAS* library for the NVIDIA CUDA architecture that uses the computational resources of NVIDIA GPUs. Basically, a C based CUDA toolchain is used in CUBLAS, which provides C-style API [25]. However, CUBLAS uses 1-based indexing and Fortran-style column major order for multidimensional arrays. This formation simplifies the CUBLAS interfacing to Fortran applications. However, the C-Fortran calling conventions differ from platform to platform, as they are not standardized. CUBLAS provides Fortran interface in the form of wrapper functions written in C [25]. There are two form of these wrapper functions:

1. Thunking wrapper and

2. Non-thunking or direct wrapper

The thunking wrappers can be used in existing Fortran applications without any changes to the applications. The GPU memory management, data copy between host and GPU and the calling of CUBLAS library functions are explicitly managed by the wrappers. This process produces a significant call overhead [25].

The non-thunking or the direct wrappers, are Fortran wrapper functions around the CUBLAS library routines. The memory management needs to be done manually in the applications. Existing applications are responsible to manage the data structures on the GPU. Most of the Molpro routines are written in Fortran. So existing Molpro modules need to be changed to cope with the non-thunking CUBLAS wrapper.

In this chapter, a set of GPU utility routines is described, The routines have been developed to enable Molpro to use the CUBLAS routines seamlessly. For maximum achievable performance, the developed wrappers support the use of multiple GPU devices. The routines help Molpro Fortran code to call and execute expensive matrix operations in a single or multiple GPUs concurrently. To show the efficiency of the developed wrappers, the DF-MP2 (second order Density Fitting MϕllerPlesset Perturbation) from Molpro has been chosen for acceleration. Nevertheless, the wrappers can be used actually by all Molpro modules which incorporate BLAS routines. Apart from the application of the developed wrappers, the energy calculation in DF-MP2 part is also done on the GPU through a CUDA kernel, specially written for it. These altogether accelerates the calculation significantly.

7.2 Wrapper Routines

The idea is to provide GPU wrappers around CUBLAS routines and CUDA API calls, that allow seamless invocation from Molpro Fortran routines. These wrappers must have the abilities to:

– Support memory management on the GPU

– Support Multi-GPU usage

– Support seamless invocation from Molpro

– Support *sm-20* and *sm-13* CUDA architecture with different code paths

So the utility wrappers are divided into two categories, CUDA API utility wrapper by using the standard CUDA API routines and CUBLAS library wrapper by using CUBLAS routines. The implementations of these wrapper routines are explained in the next section.

7.2.1 CUDA API Utility Wrapper

INIT_GPU This routine initializes a GPU device. It associates a host thread to an available GPU device. If there are N number of GPU devices in a system, typically they are numbered from 0 to $N-1$. This routine takes the host thread number as an argument and assigns the corresponding GPU to that thread. If the thread number is higher than the number of available GPUs, it can not associate any GPUs and sends an error message to the application. This mechanism needs to be strictly maintain from the application. Additionally, the routine initializes and prepares the CUBLAS library for the GPU associated with the host thread. This wrapper is implemented around the *cudaSetDevice()* routine. [27].

RELEASE_GPU This routine releases the assigned GPU device and dissociats it from the host thread. It explicitly cleans all runtime related GPU resources. It is a wrapper around *cudaDeviceReset()* in version 4.0 and around *cudaThreadExit()* routines in older versions [27].

GPU_MEM_INFO This routine returns the memory statistics for a single GPU, specifically from the associated GPU. It returns the total device memory and currently available memory at the point of calling. This wrapper is written around the CUDA routine *cudaMemGetInfo()* [27].

NUM_CUDA_DEVICE This interface returns the number of attached GPU capable devices in the system. This allows the application to know the number of available GPUs beforehand. It assigns and manages the host threads thereby. This wrapper is around CUDA routine *cudaGetDeviceCount()* [27].

GPU_DALLOC This routine allocates a double precision floating point array with n elements. It first checks the available memory on the associated GPU device. If the required space fits into the available GPU memory, it allocates and return the device pointer address to the application. This way Molpro Fortran subroutines can easily keep track of the device memory addresses. If there are no available spaces on the GPU, it returns an error to the application. Fortran does not support pointers, so the device pointers or addresses are stored in Fortran integer variables and are used as arguments for this routine. This is implemented around *cudaMalloc()* [27].

GPU_DRELEASE This routine will release the allocated memory space from the GPU. For the multi-GPU case it will only release memory from the associated GPU device of the host thread. This wrapper is implemented around *cudaFree()* [27].

GPU_DPUT This routine will copy an array of n double precision numbers from the host memory to the associated GPU memory space. The memory space on the

GPU needs to be allocated first through *GPU_DALLOC* calls. The data is copied to the GPU through the PCIe bus. This is a wrapper around *cudaMemcpy()* [27].

GPU_DGET This routine copies double precision arrays back to the CPU host memory from the GPU device memory. The memory space on the host must be allocated before through the Molpro standard memory allocation scheme. This is a synchronous operation, that means, the CPU thread will wait until the copy is finished. This wrapper is also around *cudaMemcpy()* [27].

7.2.2 CUBLAS Wrapper

Here, three interfaces to the CUBLAS double precision subroutines are written, as they are the most frequently used in the Molpro DF-MP2 module. The standard non-thunking routines can be invoked directly as well. However, in order to maintain the consistency of the interface names, the three following interfaces are rewritten. All the input arguments of these wrappers are identical to the corresponding CUBLAS routines.

GPU_DDOT This subroutine is a wrapper to the CUBLAS DDOT (cublasDdot) subroutine [25]. This computes the dot product of two double precision vectors [25] and returns the dot product.

GPU_DGEMM This wrapper is around the CUBLAS DGEMM (cublasDgemm) subroutine [25]. It performs the standard double precision matrix-matrix multiplication and is identical to the BLAS [26] DGEMM operation. Transposition of input matrices are performed as it is performed in the DGEMM routines. Typically this matrix-matrix multiplication is defined as follows.

$$C = \alpha * op(A) * op(B) + \beta * C \tag{7.1}$$

where $op(X) = X$ or $op(X) = X^T$, transpose of X [25]. If β is zero, then the multiplications with β and addition to C are skipped completely. The multiplication operation for alpha is always done, even if $\alpha = 1$, but these operations are quite negligible compared to the bulk of A*B computation. The FLOPS are usually approximated as $(2*m*n*k + 3*m*n)$ whenever $\beta \neq 0$, and $(2*m*n*k)$ for the simple case of $\beta = 0$. Here

m represents number of rows in matrix $op(A)$ and in matrix C

n represents number of columns in matrix $op(B)$ and in matrix C

k represents number of columns in matrix $op(A)$ and rows in matrix $op(B)$ [25]

GPU_DTRSM This is a wrapper around the CUBLAS DTRSM (cublasDtrsm) subroutine [25]. It performs the standard DTRSM operation between two matrices. The input arguments are the standard arguments as in standard BLAS routine. This routine is used to adapt the Fortran DTRTRS Lapack code by using *GPU_DTRSM* wrapper in Molpro.

7.3 Case Study: Application of the Wrappers in Molpro

In Molpro, DF-MP2 is implemented as a multi threaded routine written in Fortran and it consists of large number of matrix operations. The DF-MP2 code is ported to the GPU using the developed wrappers.

7.3.1 DF-MP2 Theory

The closed-shell MP2 energy is calculated by

$$E_{MP2} = \sum_{i \geqslant j} (2 - \delta_{ij}) \sum_{ab} (ai|bj)[2(ai|bj) - (bi|aj)], \tag{7.2}$$

Here $(ai|bj)$ are the 2-electron integrals in the molecular orbital (MO) basis and defined as

$$(ai|bj) = \int \rho_{ai}(r_1) \frac{1}{|r_1 - r_2|} \rho_{bj}(r_2) dr_1 dr_2, \tag{7.3}$$

$$\rho_{ai}(r_i) = \phi_a(r_i)\phi_i(r_i). \tag{7.4}$$

$\rho_{ai}r_i$ are the one-electron densities, that are products of the occupied orbitals $\phi_i(r_i)$ and virtual orbitals $\phi_a(r_i)$, where r_i denotes the coordinates of electron i. The distance between the electrons 1 and 2 is represented by $|r_1 - r_2|$ [28]. In the density fitting approximation, the orbital products $\rho_{ai}(r_i)$ are approximated by linear exapnsions

$$\rho_{ai}(\mathbf{r}) \approx \sum_A \chi_A(\mathbf{r}) D_{A.ai} \tag{7.5}$$

Here, $\chi_A(\mathbf{r})$ are fitting basis functions and $D_{A,ai}$ are the fitting coefficients. This later yields the linear equations

$$K_{A,ai} = \sum_B J_{AB} D_{B,ai} \tag{7.6}$$

This is a matrix multiplication and can be written in matrix notation as follows

$$\mathbf{K} = \mathbf{JD} \tag{7.7}$$

J and **K** are known as 2-index and 3-index 2-electron integrals, respectively and defined as [28]

$$[J]_{AB} = \int \chi_A(r_1) \frac{1}{|r_1 - r_2|} \chi_B(r_2) dr_1 dr_2 \equiv (A|B) \tag{7.8}$$

$$[K]_{A,ai} = \int \chi_A(r_1) \frac{\rho_{ai}(r_2)}{|r_1 - r_2|} \chi_B(r_2) dr_1 dr_2 \equiv (A|ai) \tag{7.9}$$

From this it follows

$$(ai|bj) = \sum_A D_{A,ai} K_{A,bj} = [\mathbf{D}^T \mathbf{K}]_{ai,bj} \tag{7.10}$$

$$= \sum_{A,B} K_{A,ai} J^{-1}_{AB} K_{B,bj} = [\mathbf{K}^T \mathbf{J}^{-1} \mathbf{K}]_{ai,bj} \tag{7.11}$$

A symmetric equation can be formed from equation (7.11), with

$$\tilde{\mathbf{D}} = \mathbf{J}^{-1/2} \mathbf{K} \tag{7.12}$$

So the new symmetric formula can be written in matrix notation as follows [28]

$$(ai|bj) = [\tilde{\mathbf{D}}^T \tilde{\mathbf{D}}]_{ai,bj} \tag{7.13}$$

In this case a diagonalization of **J** is needed to make $\mathbf{J}^{-1/2}$. An efficient and numerically stable Cholesky decomposition can be applied to decompose **J** into a product of lower triangle matrix and its transpose.

$$\mathbf{J} = \mathbf{L}\mathbf{L}^T \tag{7.14}$$

$$\mathbf{J}^{-1} = [\mathbf{L}^{-1}]^T \mathbf{L}^{-1} \tag{7.15}$$

$$\tag{7.16}$$

This decomposition helps to solve the linear equations using the triangular matrix **L** through DTRTRS Lapack subroutine.

$$\mathbf{K} = \mathbf{L}\tilde{\mathbf{D}} \tag{7.17}$$

As the orbitals ϕ_r are expanded in a basis χ_μ with LCAO approximation, the 3-index integrals $(\mu\nu|A)$ are first evaluated in AO (atomic orbital) basis $(\mu\nu)$ and then transformed into the MO (molecular orbital) basis (a, i). The number of occupied orbitals i, is much smaller than the number of virtual orbitals a. So that the transformation to the occupied orbitals is done first and then the transformation to the virtual orbitals are done [28] [12].

$$(\mu i|A) = \sum_\nu C_{\nu i}(\mu\nu|A) \qquad \text{first half transformation} \tag{7.18}$$

$$(ai|A) = \sum_\mu C_{\mu a}(\mu i|A) \qquad \text{second half transformation} \tag{7.19}$$

7.3.2 Implementation

The 2-electron integrals (7.13) need to be evaluated first to calculate the energy. So the quantities $D(A, a, i)$ needs to be loaded in to the GPU memory first. This is done by the GPU wrappers written to map with the Fortran code. Then it has to be assembled by one *cublasDgemm* operation for each pair of i and j.

$$(ai|bj) = K_{ab,ij} = \sum_A D_{A,ai} * D_{A,bj} \qquad \text{for} i \geq j \tag{7.20}$$

Here A is the number of fitting basis functions, a the number of virtual orbitals, and i the number of correlated orbitals. The $D_{A,ai}$ is loaded into the GPU memory in batches for as many i as possible and for one batch one j at a time (with $j < i$). So that each load of j, upto i number of matrix multiplication can be done in GPU with out any data transfer between CPU and GPU. All the matrix multiplications on the GPU are done through the wrapper *GPU_DGEMM*. The DF-MP2 code is written and ported to the GPU by Prof. Hans-Joachim Werner[1]. Later the energy (7.2) is calculated also on the GPU. A GPU kernel wrapper (*GPU_MP2EN*) is written to compute the energy as part of the thesis.

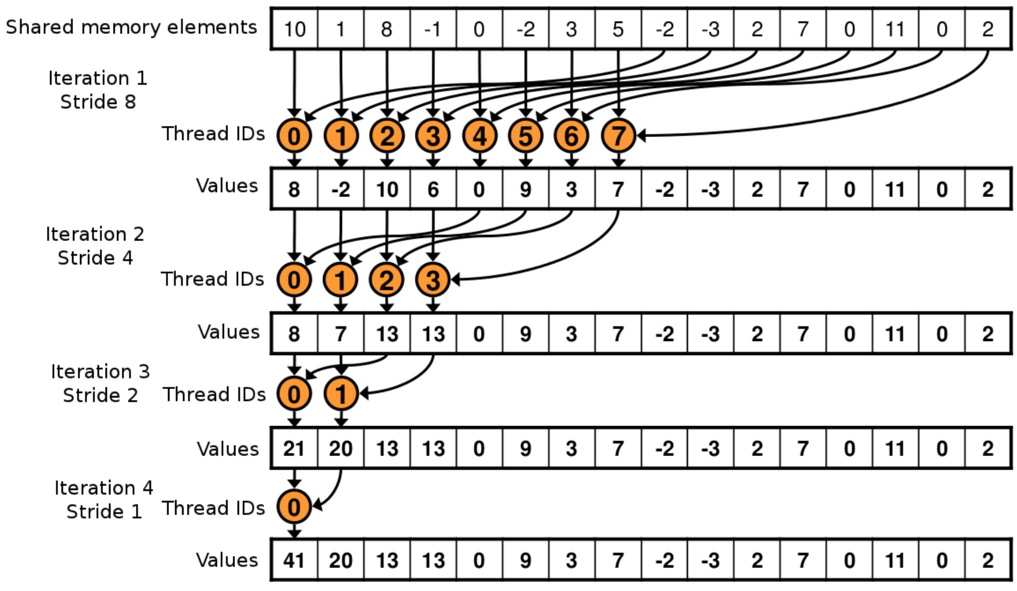


Figure 7.1: Parallel reduction with sequential addressing [29]

[1]Institut für Theoretische Chemie, Universität Stuttgart

7.3.3 GPU_MP2EN Wrapper

The summation of MP2 energy components (7.2) from the orbital energies is done by this wrapper. The orbital energies are calculated before(7.3.2) and stored in the GPU global memory. The summation of the energies on the GPU will reduce the memory transfer latency. Three energy components are calculated from the orbital energy matrix. The parallelization is done over all the matrix elements. A CUDA kernel (*dfmp2_en_kernel*) is written to calculate the energy and to do the block level reduction of the energy. This kernel is written with CUDA C-style tool chain and *GPU_MP2EN* is the Fortran wrapper around this kernel. The number of elements in the orbital matrix is known. To do the parallelization over all the matrix elements, the same number of CUDA threads is created. Each element of the matrix is mapped to one CUDA thread and the CUDA thread computes the energy components for single matrix elements and store them in the shared memory.

Listing 7.1: Pseudocode of parallel reduction

```
__global__ void dfmp2_en_kernel( ...  ...  ...)
{
  int s_i = threadIdx.x;
  int s_j = threadIdx.y;
  int s = s_getindex(s_i , s_j);
  // acc is the number of threads per block
  __shared__ double s_ders[acc];
  __shared__ double s_desr[acc];
  __shared__ double s_gn  [acc];
  ...
  //computation of energy components
  //store the results in shared memory
  ...
  __syncthreads();
  // reduction is done in shared mem
  // with strides index s_x
  for(unsigned int s_x= acc/2; s_x>0; s_x>>=1)
  {
          if (s < s_x)
          {
                  s_ders[s] += s_ders[s_getindex(s_i + s_x,s_j)];
                  s_desr[s] += s_desr[s_getindex(s_i + s_x,s_j)];
                  s_gn[s] += s_gn[s_getindex(s_i + s_x,s_j)];
          }
          __syncthreads();
  }
  if((s_i == 0) && (s_j == 0))
  {
    // write result for this block to global mem
    ...
    ...
  }
}
```

These results are accumulated later using a tree based approach within each thread block. Each thread block reduces (makes the summation) of a portion of the energy values. This scheme is also known as parallel reduction technique. The listing 7.1 shows the pseudocode of the tree based reduction in each thread block. The CUDA threads access the elements from the shared memory in a sequential addressing technique. The accumulation is done for each block and the results are stored in a vector in the global memory. The size of the vector is small and is equal to the number of thread blocks configured for this CUDA kernel call. Three vectors are created to store the intermediate reduction results of the three MP2 energy components. Later these vectors are transferred to the host thread to do the rest of the accumulation with a simple for loop. The required memory transfers from the GPU to the host are considerably reduced, since a large initial reduction is done on the GPU. The following figure 7.1 explains the parallel reduction technique using sequential shared memory addressing. The sequential shared memory addressing technique has the advantage that bank conflicts in shared memory can be avoid. However, it has an disadvantage too, for each iteration in each block, half of the threads are being idle. Fortunately, this does not have any effect.

CHAPTER 8

Results and Performance Evaluation

Contents

8.1 Introduction

In chapter 6 the implementation of two computationally expensive DFT algorithmic parts with a hybrid CPU-GPU architecture were presented. A set of GPU wrappers were also introduced in chapter 7. In this chapter, the evaluation of the performance and the efficiency of the developed DFT mapping is presented. The Fortran GPU wrappers are extensively used in the Density Fitting Mϕller-Plesset Perturbation (DF-MP2) calculation. The performance of this GPU DF-MP2 code is also evaluated for a large test case.

8.2 Performance of the Hybrid CPU-GPU DFT Code

The performance of the hybrid CPU-GPU DFT code is analyzed by running a large number of DFT calculations. The DFT calculations are running within the Molpro [12]. The *df-ks* input command initiates the density fitted Kohn-Sham calculations. Polyvinyl-fluoride $-(CH_2CHF)_n-$ is used for the performance benchmarking, here n represents the number of monomers. Polyvinyl-fluoride is a long chain with repeating vinyl fluoride units. The structure of vinyl fluoride is shown in the figure 8.1. Figure 8.2 represents the long chain of Vinyl-fluoride.

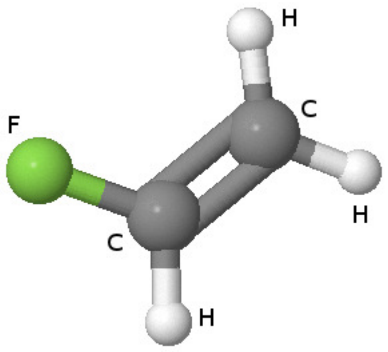


Figure 8.1: 3D structure of Vinyl-fluoride
[30]

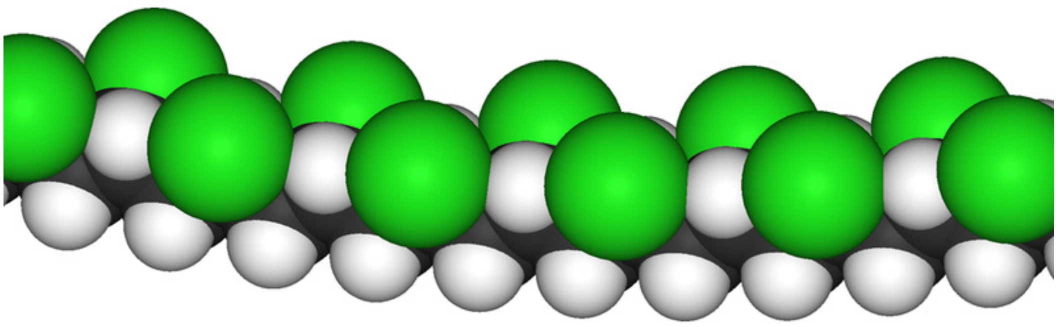


Figure 8.2: 3D structure of Polyvinyl-fluoride (long chain of Vinyl-fluoride)
[30]

This molecule is typically used in flammability-lowering coating of airplane interiors.

The Molpro code is configured using *cuda* and *mpp* flags to compile with the CUDA tool chain and MPP, respectively to support CUDA and many-core.

Listing 8.1: Configuring Molpro

```
./configure −mpp −auto−ga−mpich2 −cuda
```

The benchmarking is performed on a 12-core 3.33 GHz Intel(R) Xeon(R) X5680 machine with four NVIDIA Fermi GPUs. Table 8.1 summarizes the system used in our benchmarking. The GPU device properties are listed in the table 8.2. Molpro is a multi compiler applications. Table 8.3 lists compiler names with their version to compile the Molpro code.. The following performance benchmarks are done on Polyvinyl fluoride $-(CH_2CHF)_10-$, with $n = 10$, called here as *pv-10* molecule.

Table 8.1: System used in benchmarking

System Properties				
Vendor	**Model**	**Speed/GHz**	**Cores**	**Memory/GB**
Intel	Xeon(R) X5680	3.33	12	48

Table 8.2: GPU device used in benchmarking

GPU Device Properties				
Vendor	**Model**	**Cores**	**Memory/GB**	**Number of Device**
NVIDIA	Tesla C2070	448	6	4

Table 8.3: Compilers used with versions

Compiler	**Version**
C	GCC 4.3.4
Fortran	Intel Fortran 12.0.4
CUDA	3.20
Make	GNU Make 3.81

This molecule chain consists of 62 atoms, which contains 32 *Hydrogen*, 20 *Carbon* and 10 *Fluorine* atoms. The *aug-cc-pVDZ* orbital basis set (OBS) (with 978 CGTOs[1]) and TZVPP/JFIT (with 4228 GTOs, for B-LYP functionals) basis set is used in this case.

Not all the routines of the Molpro DFT code are ported to the GPU. As explained in chapter 6, the two main computational bottlenecks, the evaluation of the density matrix and the exchange correlation contribution to the Fock matrix are implemented on the GPU. The remaining parts of the DFT are executed by the CPU cores. So the main focus is on the DFT GPU parts and how well they scaled from the original CPU version. The speedup of the total DFT calculation executed in the hybrid approach is presented. The notation used on the charts are defined as:

– 0 GPU: With out any GPU accelerators but with single core

– 1 GPU: With single GPU accelerator and single core

– 2 GPU: With two GPU accelerators and two cores

– 4 GPU: With four GPU accelerators and four cores

[1]Contracted Gaussian Type Orbitals

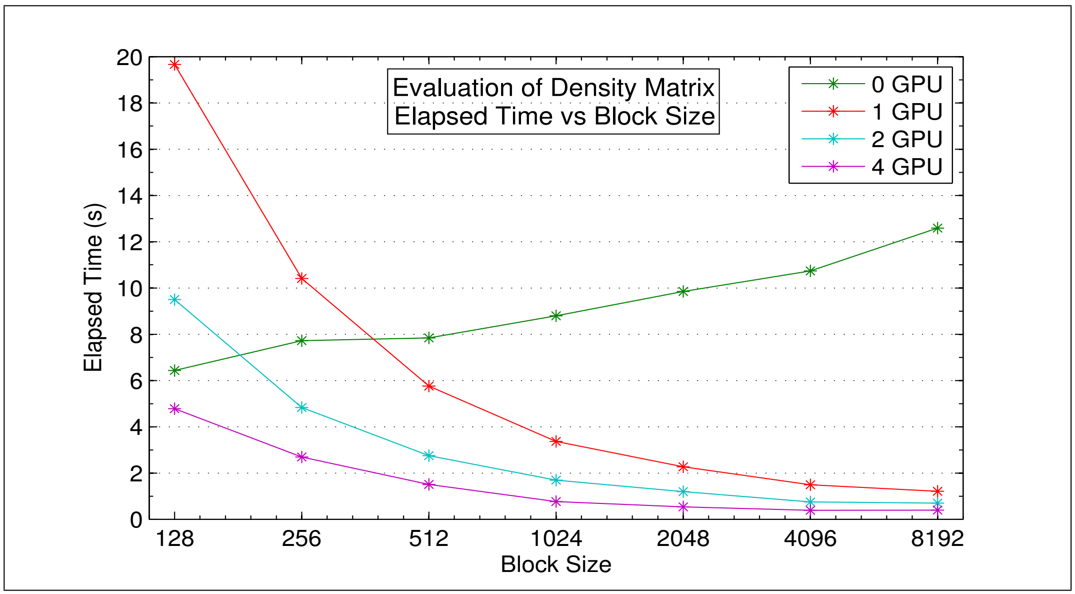


Figure 8.3: Elapsed time for evaluation of density matrix for different block size

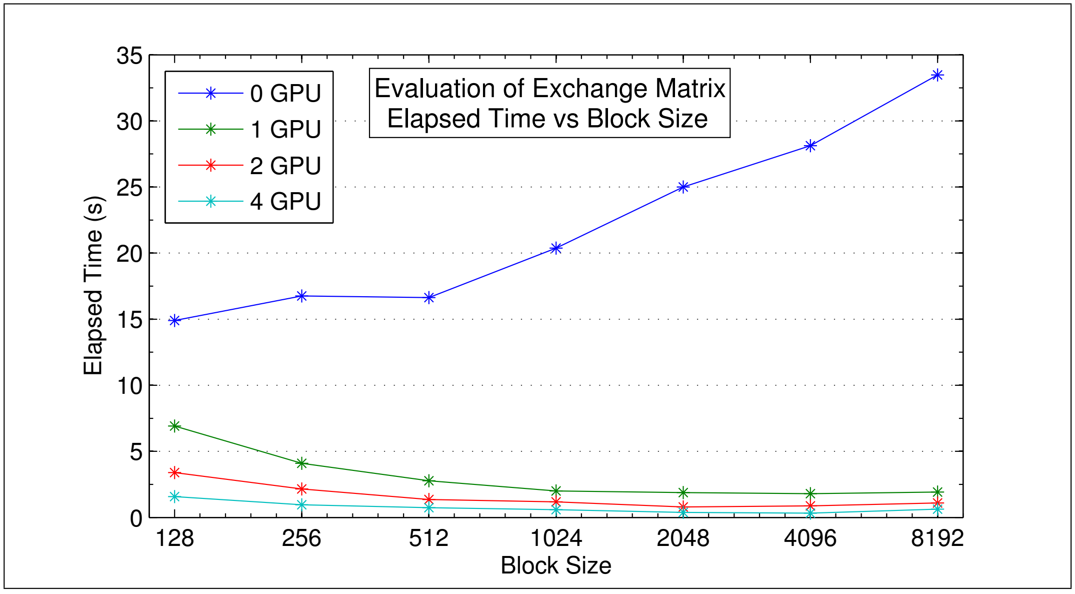


Figure 8.4: Elapsed time for evaluation of exchange correlation matrix for different block size

8.2.1 Selection of Optimum Grid Block Size

The original production CPU code and the GPU version of the DFT modules are executed for different grid block sizes to analyze the behavior of the execution time. In figure 8.3, the timing of the density matrix evaluation with respect to increasing grid block sizes for one SCF iteration has been shown. It is noticed that the grid block size of 128 is optimal for the CPU version of the density matrix routine. However, using a grid block size of 128 for the GPU is very expensive, and leads to a 3 times slower execution time compared to the CPU version. This is evident as the GPU requires a very large number of threads to minimize the memory access latency and GPU pipelining overhead. Increasing the grid block size reduces the elapsed time significantly for the GPU code. It has been seen that with block sizes of 4096 or 8192, the GPU version executed very efficiently. The same trend has been evident in the exchange matrix contribution to the Fock matrix calculation in figure 8.4.

The block size of 128 is optimal for the CPU code for the exchange correlation matrix. With this block size the GPU code works considerably better than the CPU version. However, the optimal block size for the GPU version is 4096, where it performs best in terms of elapsed time in single GPU as well as multi-GPU setups. Based on this outcome, the block size of 4096 is taken as an optimum for the GPU ported DFT code to do both the density and the exchange matrix calculations. Consequently, for all other benchmarking runs, the block size of 4096 is used for the GPU version and 128 for original CPU code. The performance of the GPU code will be compared with the performance of the optimized CPU versions.

8.2.2 Relative Timings for DFT Code

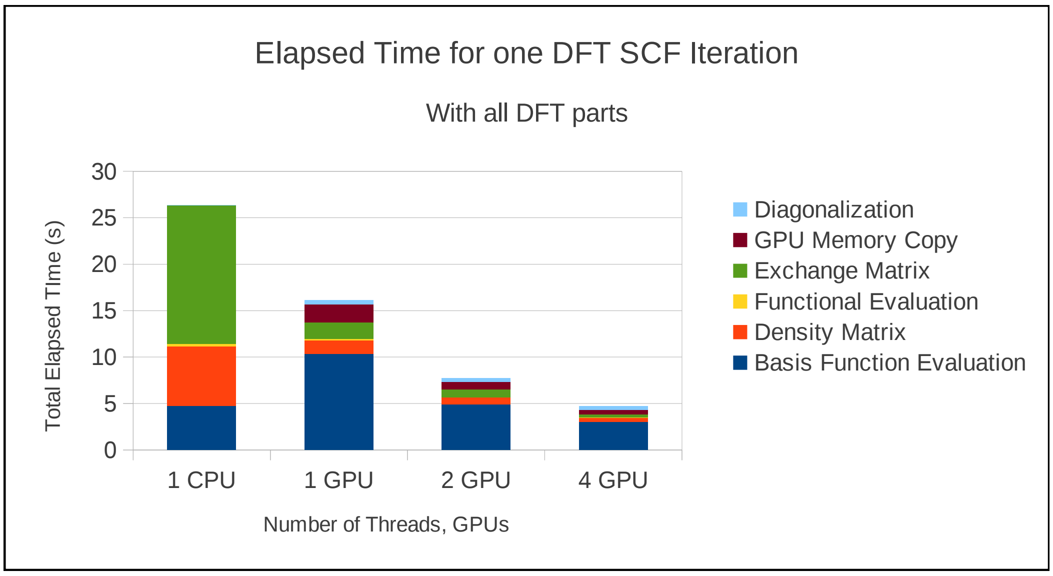


Figure 8.5: Relative timings for CPU and GPU code

In the figure 8.5, the relative elapsed time of the total DFT calculation for a single

SCF iteration is shown for single core to multi-GPU cases. As in the GPU case, the use of a large grid block size has an effect on the basis function evaluation. As the basis function evaluation is not ported to GPU and still executed on the CPU side, it now takes longer for large block sizes. The execution time of density matrices and the exchange correlations matrix show a promising speed up compared to the multi threaded CPU version. Although there is a notable overhead for CPU-GPU memory

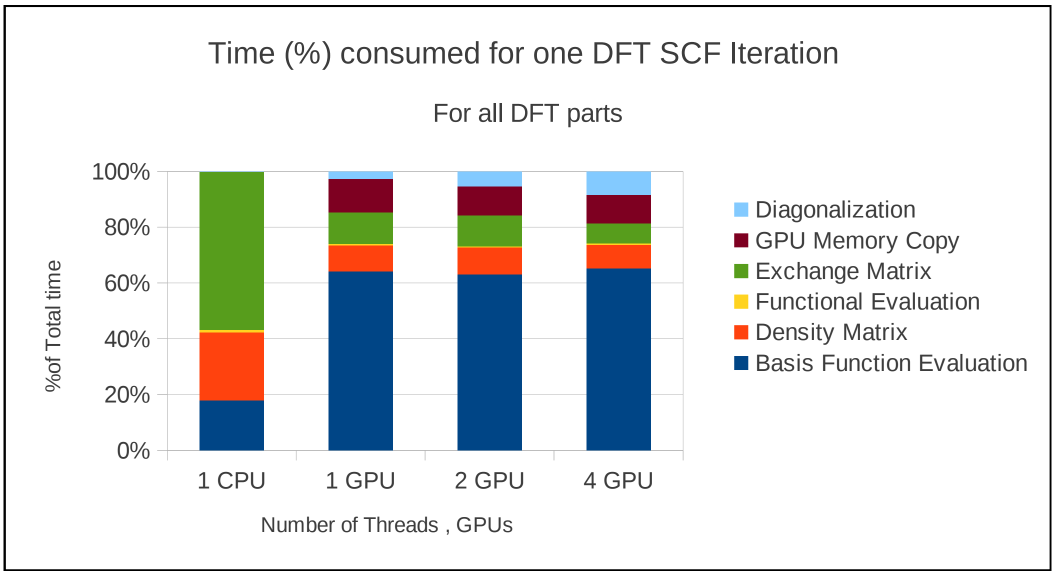


Figure 8.6: Percentage of relative timings for CPU and GPU code

data transfer in the GPU cases, the applications scaled very well in the multi-GPU system. From this it is clear that the preference of the density calculation and the exchange correlation matrix for larger block size out-competes the preference of the other bottlenecks for smaller block sizes. The best performance is obtained using a grid block size of 4096 or 8192. The percentage of relative timing for the DFT algorithmic parts are shown in this figure 8.6. In the GPU versions, the basis function evaluation is now becoming the computational bottleneck for large grid block size. The CPU-GPU memory data transfer is also taking significant portion of the total timings. The DFT functionals take a negligible amount of time to evaluate on the CPU. However, they cost a large overhead of CPU-GPU memory transfers (10% to 15% of total execution time). So, if the functional calculation will be additionally ported to GPU, it will reduce the CPU-GPU memory transfers significantly.

8.2.3 Effects on Basis Function Evaluation Timings

In the figure 8.7, the effects of the large grid block sizes in the execution of basis function evaluation are shown. This module is running on the CPU very efficiently with a grid block size of 128. As the size increases, the elapsed time for this module

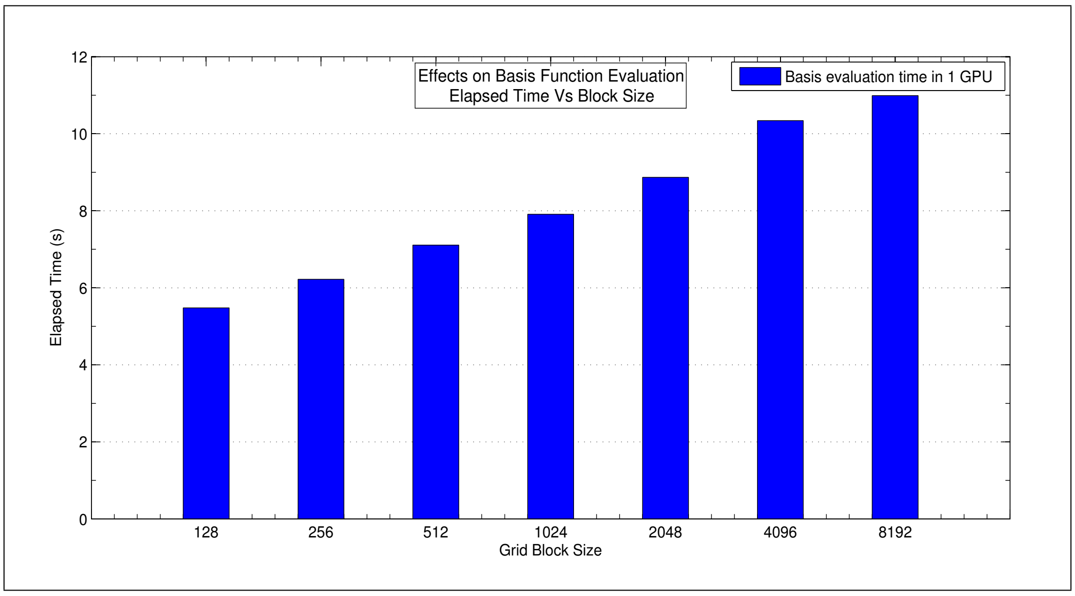


Figure 8.7: Effects on CPU basis function evaluation module for increasing grid block size

increases linearly. The GPU modules have to be executed with larger block sizes, hence the evaluation of the basis functions is growing as the new computational bottleneck for the total DFT applications.

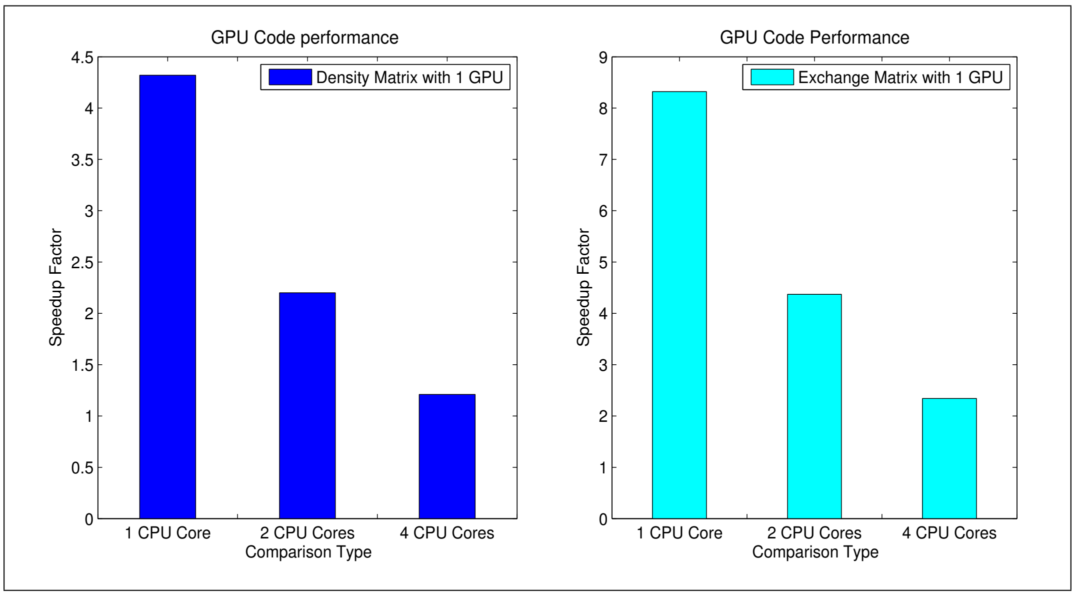


Figure 8.8: Performance of GPU routines in terms of relative speedup factor

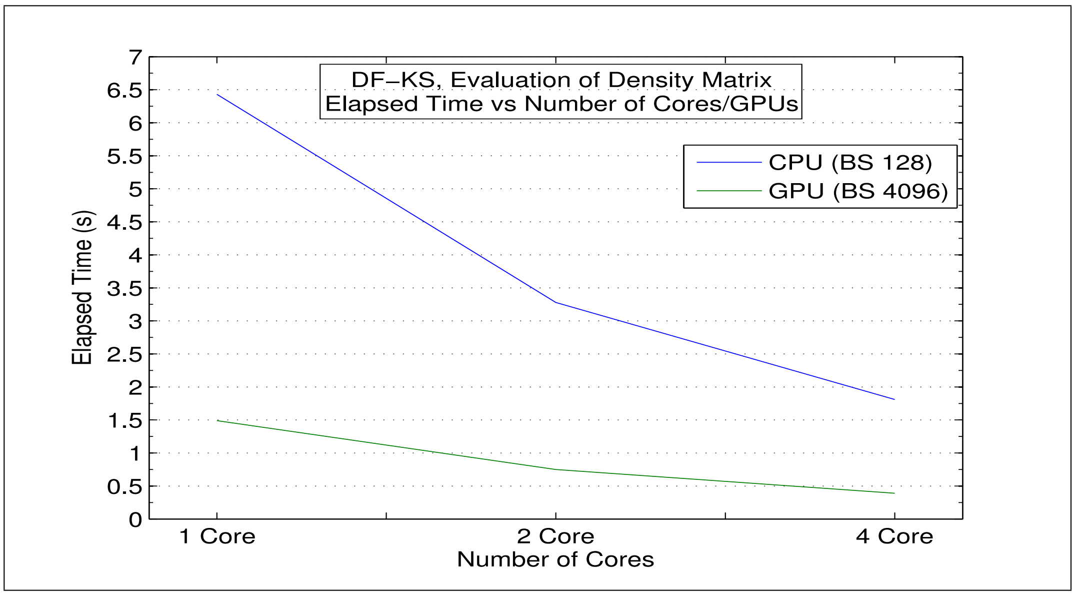


Figure 8.9: Performance of density matrix evaluation in multi-GPU environment

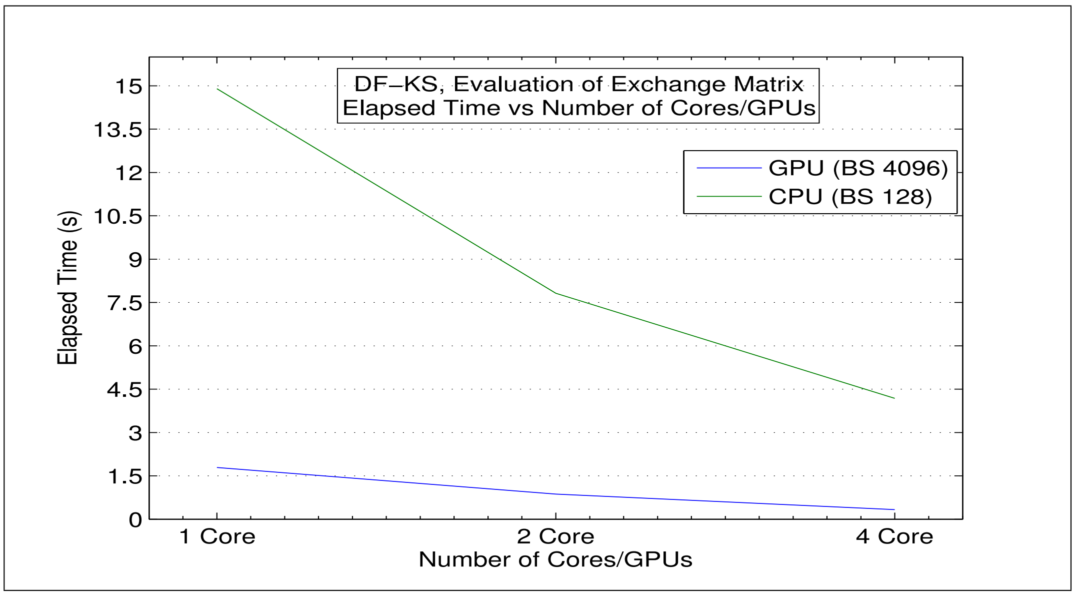


Figure 8.10: Performance of exchange correlation matrix evaluation in multi-GPU environment

8.2.4 Performance of the DFT Code

Figure 8.8 shows the performance of the modules that are ported to the GPU. This test is running on a single GPU with the same molecule and input parameters. The execution efficiency, in terms of the speedup factor, is analyzed, and compared

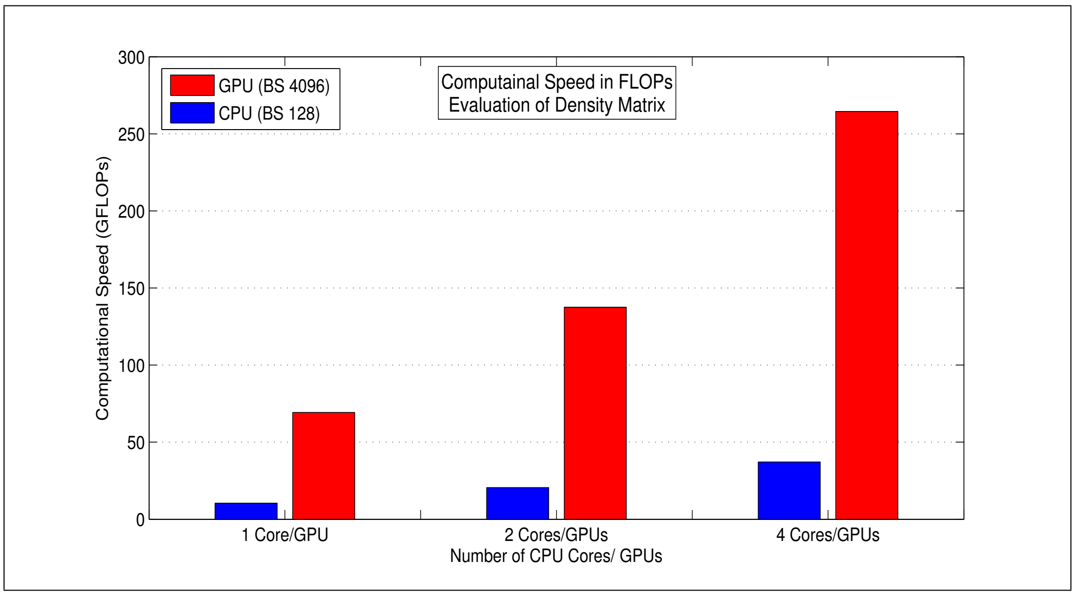


Figure 8.11: Computational speed for density matrix evaluation in terms of GFLOPs in multi-GPU environment

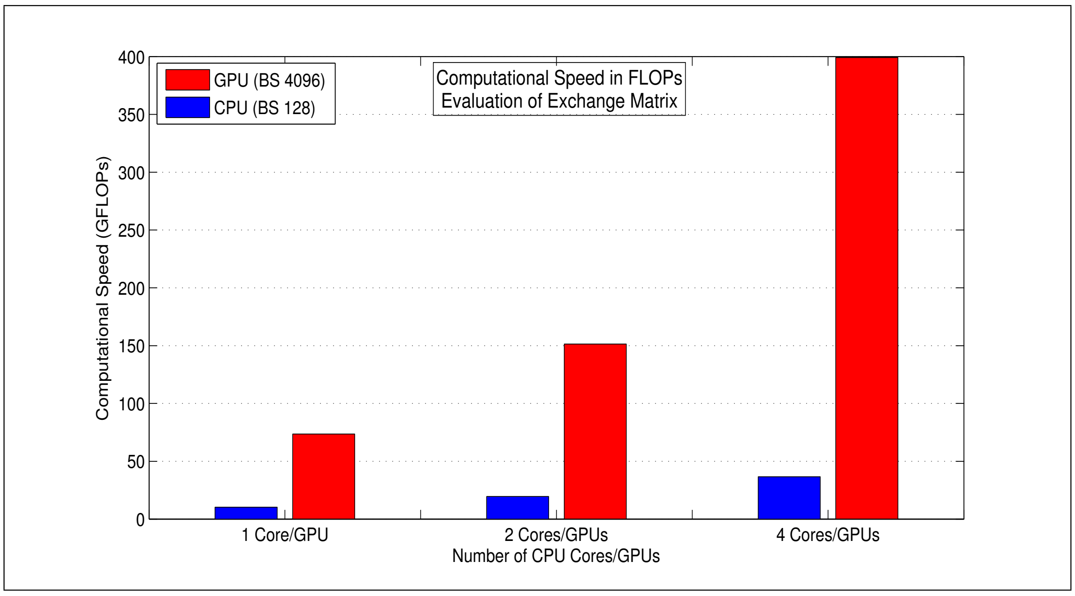


Figure 8.12: Computational speed of exchange correlation matrix evaluation in terms of GFLOPs in multi-GPU environment

with the single thread to multi thread execution. The density matrix code in GPU presents a speedup with a factor of nearly 4.5 and the exchange correlation contribution has a speedup with a factor of more than 8 compared to the single core machine. The graph 8.8 also shows a relatively very good scaling compared to the multiple CPUs.

Figure 8.9 shows the performance of the density matrix evaluation in a multi GPU environment. The CPU code is executed with the optimal block size of 128 over multiple cores. The GPU code is executed with the optimal block size of 4096 over multiple GPUs. In the GPU case, each thread is associated with one GPU. The figure 8.9 shows the comparison of the elapsed time for different multi threaded execution.

Figure 8.10 shows the performance of the exchange correlation matrix over the multi threaded multi-GPU environments. The elapsed time is reduced linearly for multi GPU case.

Figure 8.11 and the figure 8.12 show the computational speed in FLOPS for the density matrix evaluation and the exchange matrix evaluation, respectively.

The results exhibit very good performance over the multi-core CPU implementations. However, the GPU FLOPs performance is still far from the theoretical peak performance of the GPU device. The *Tesla C2070* has a peak performance with double precession of 515 GFLOPs. One of the reason for this difference is for the large number of memory transformations (like, compression, unpacking, etc of matrices), as explained in chapter 6.

8.2.5 Performance Comparison with Large Basis Sets

In the previous test cases, the *aug-cc-pVDZ* basis sets has been used. The following test uses few large basis sets for the evaluation of the GPU code. The name of the different basis sets with the number of contracted Gaussian Type orbitals (CGTOs) are listed in the table 8.4. Figure 8.13 shows the performance of the CPU and GPU

Table 8.4: Basis sets with the number of contractions

Basis Set	**Number of CGTOs**
cc-pVDZ	580
aug-cc-pVDZ	978
cc-pVTZ	1348
aug-cc-pVTZ	2116
cc-pVQZ	2610

code together with respect to the different basis sets. As the number of basis functions increases, each GPU thread needs to do more calculations, hence the elapsed time increases. The density matrix code scales well over the number of basis functions. Figure 8.14 shows the performance of the exchange correlation matrix for different basis functions. The GPU version of the exchange correlation matrix

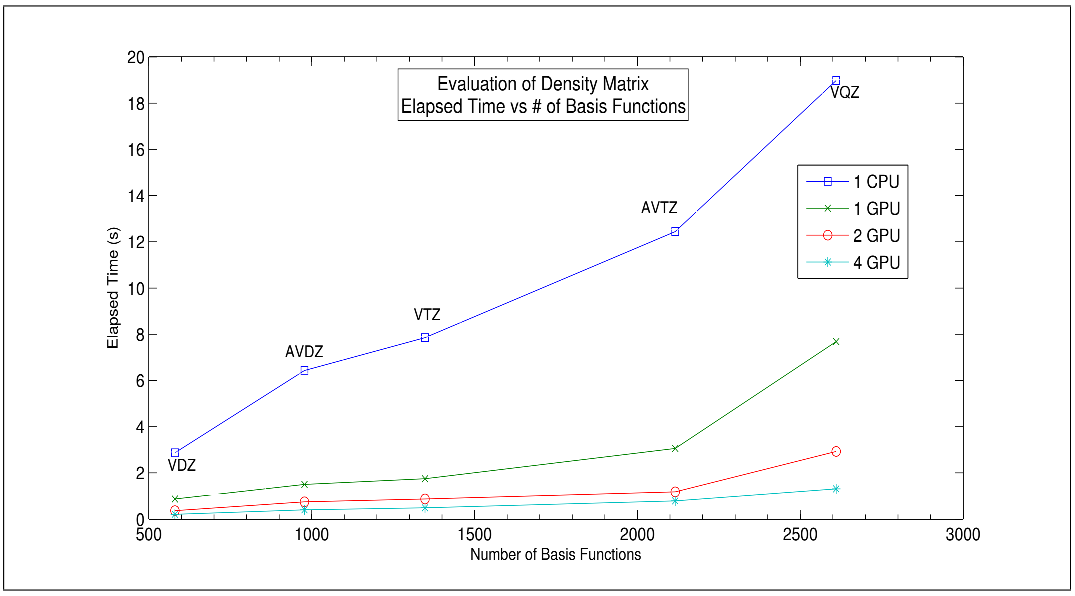


Figure 8.13: Performance of density matrix evaluation with different basis functions

outperforms the execution of the CPU version significantly. Figure 8.15 shows the

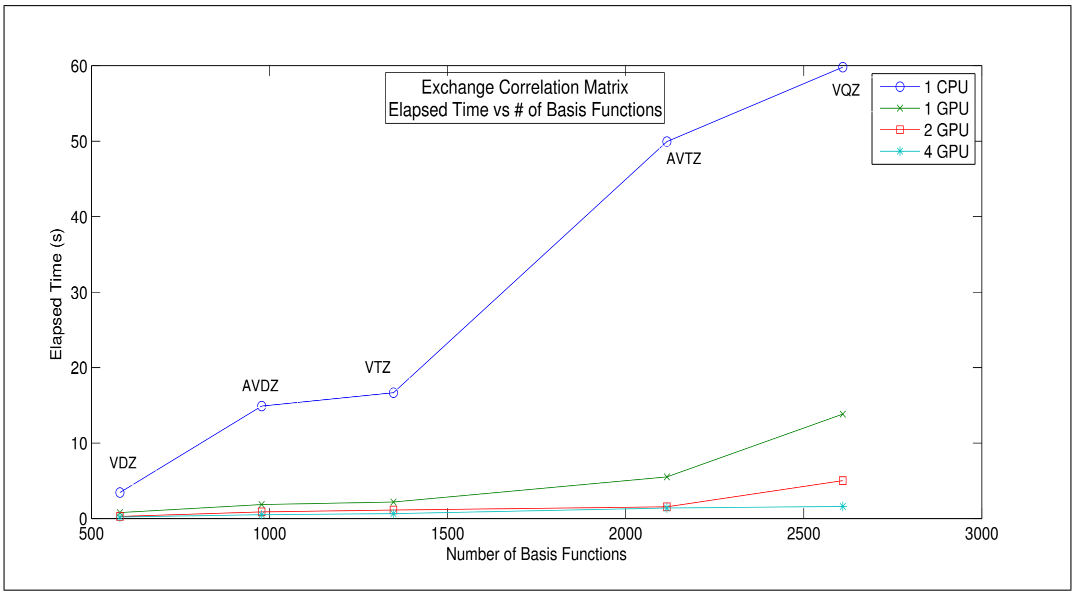


Figure 8.14: Performance of exchange correlation matrix evaluation with different basis functions

elapsed time for one SCF iteration of overall DFT calculation over the number of basis functions. The overall executional efficiency is similar to the earlier cases explained before except with the *cc-pVQZ*. Together with the very large number of contractions (2610 CGTOs) in this basis set along with the larger block size selected for the GPU code, the evaluation of the basis functions now becomes the main bottleneck of the calculation.

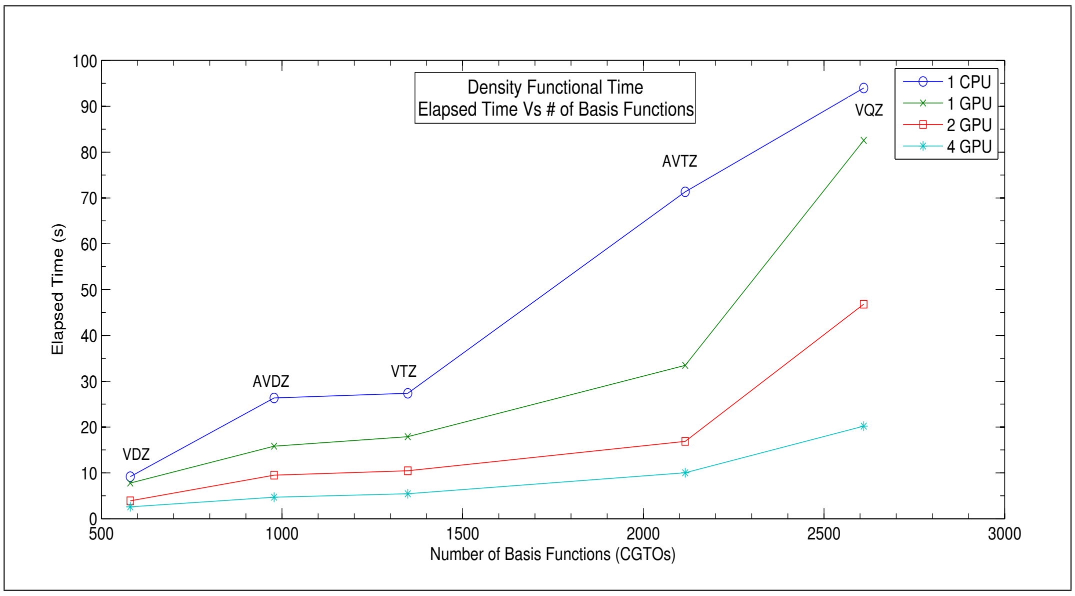


Figure 8.15: Performance of density functional theory calculation with different basis functions

8.2.6 Summary

From the benchmarks, it is evident that the smaller block size performs very good in the multi-core CPU implementation, whereas it is no longer valid for the many-core architectures, like the GPU accelerators. The GPU implementation requires a very large number of grid points to map with the GPU threads to increase the arithmetic intensity which will hide the memory access latency and will maximize the performance for parallel execution. Hence the maximum GPU utilization can be achieved. The evaluation of the basis functions becomes the new bottleneck. Due to the large block size selection, the basis function evaluation is now 3 to 4 times slower than the CPU code. The DFT functional takes a very small amount of time to calculate the functional energies. But it requires CPU-GPU memory data transfers in the middle of the DFT calculations, as explained in chapter 6, section 6.5, the hybrid CPU-GPU implementation. So it is clear that, if the basis function evaluation and the DFT functionals are ported to GPU, the total execution time can be significantly reduced.

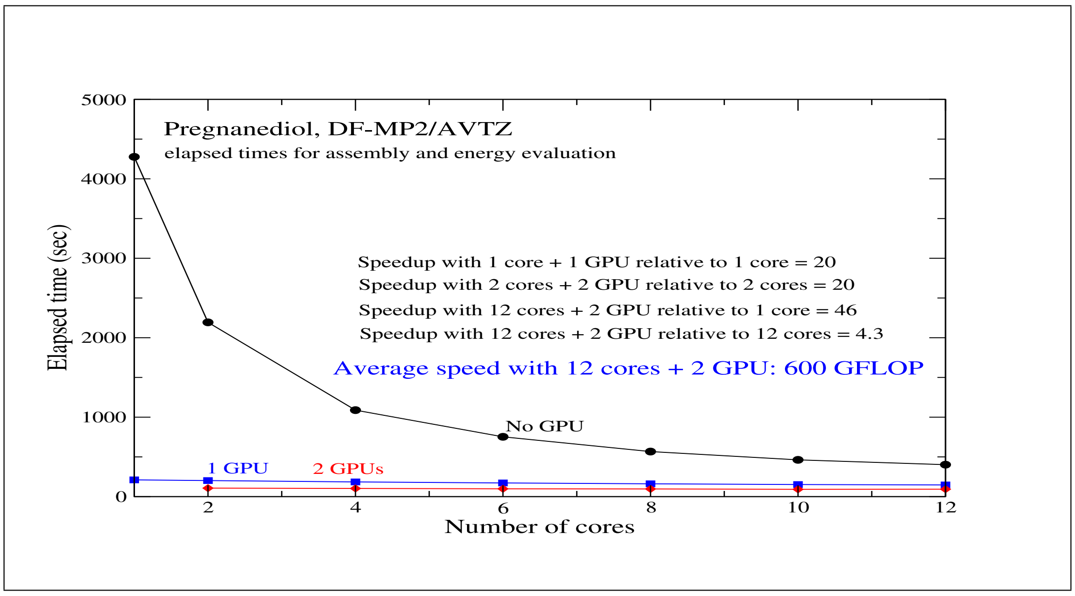


Figure 8.16: Performance of DF-MP2 calculation with energy evaluation in GPU [31]

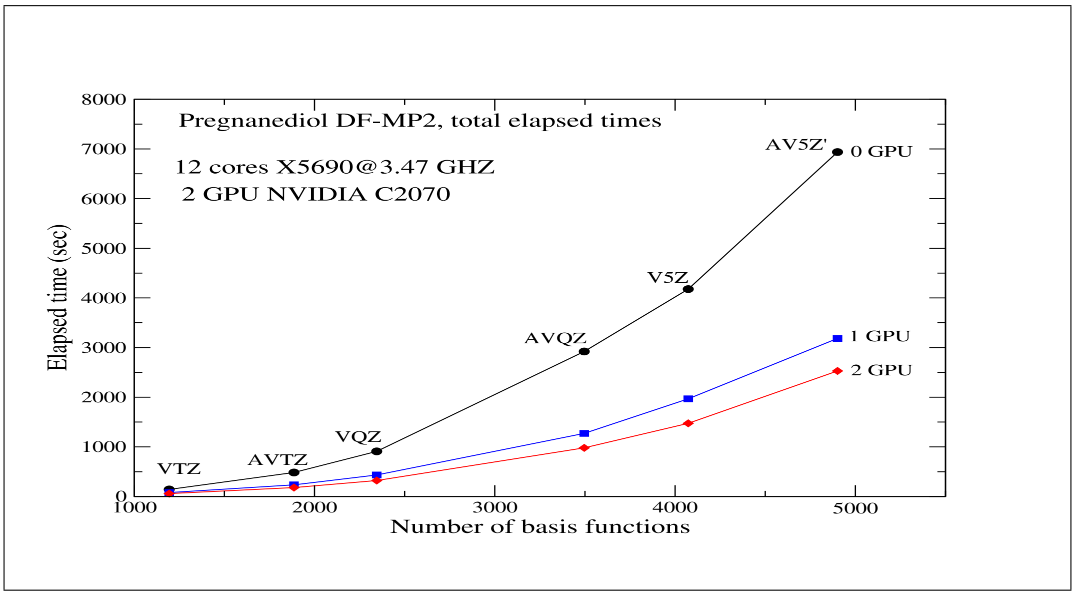


Figure 8.17: Performance of DF-MP2 calculation with different basis functions [31]

8.3 Performance of DF-MP2 Code

In this section, the performance of the DF-MP2 GPU code is presented. As explained in the chapter 7, the assembly and the MP2 energy evaluation parts are ported to the GPU. This two algorithmic parts heavily use the GPU Fortran wrappers. The Pregnanediol, $C_{21}O_2H_{36}$, which contains 59 atoms is used for this benchmarking. Figure 8.16 presents the total elapsed time for the assembly and the energy evaluation with respect to multi-core and multi GPU architectures.

The *aug-cc-pVDZ* basis set is used in this test case. It shows a factor of 20x speedup with respect to a single core execution. A factor of 46x speedup is achieved with 12 cores and 2 GPUs relative to a single core execution. Around 600 GFLOPS of computational performance is achieved from this calculation. Figure 8.17 shows the elapsed time of three different cases with respect to the different basis sets.

Chapter 9

Conclusion

9.1 Summary

In this thesis, the two principal computationally expensive algorithmic parts of the integrated density functional code in Molpro has been analyzed and mapped to the GPGPU accelerators. The proposed implementation will automatically scale over multiple GPUs. The production code from Molpro has very good performance and excellent efficiency in multi threaded parallel computation. The DFT code in Molpro supports very high systematic convergence properties. The proposed GPU implementations in this thesis maintain and respect these properties strictly. The GPU code sections are going to be inserted into the Molpro production code. The proposed GPU implementation codes have been implemented with double precision arithmetic supports. The proposed implementations also maintain the compatibility with the existing parallelization of the Molpro code. These Molpro embedded GPU codes are then tested for different molecules with varying number of atoms on a 12 Core Intel Xeon machines which includes four Tesla C2070 Fermi GPUs. Each of the ported GPU routines produces a speedup of a factor of 5x to 10x comparing with the Molpro multi-core parallel implementation. The GPU code also shows a linear scaling over multiple GPUs. The original aim of this thesis was to reduce the runtime of the DFT calculations that the calculation poses in a commodity processor system. This has been achieved in some extents for different molecular sizes with larger basis sets. The benchmarking for a molecule of size 62 atoms with aug-cc-pVTZ orbital basis sets (with 2116 CGTOs) and TZVPP/JFIT (with 4228 GTOs, for B-LYP functionals) basis shows that the hybrid DFT code with a single GPU took 33.6 sec where as a single core CPU required 71.3 sec for one SCF iterations.

Additionally, a set of multi GPU wrappers were implemented to use in the Molpro production Fortran code. These wrappers are implemented around CUDA APIs and CUBLAS subroutines. The wrappers help the existing Fortran code to seamlessly invoke CUDA functions and to easily port the large matrix operations to the GPU. These wrappers make the GPU memory management easier by invoking from the existing Molpro Fortran routines. They are developed in such a way that they are scalable to multiple GPUs. Later, these wrappers were introduced to the calculation of Molpro DF-MP2 calculation. These developments produce an overall speedup on the whole DF-MP2 code of a factor of around 20x relative to single core running. The calculation is also achieved the scalability over multiple GPUs.

9.2 Future Work

9.2.1 Basis Function Evaluation

It has been noticed that in the proposed development, the grid block size needs to be large enough to get the efficient performance from the GPU accelerators. It is found that for the developed GPU codes, the grid block size of 4096 or 8192 are the optimum choice in evaluating the density matrix and the exchange correlation contributions to the Fock matrix. However, this in turns increases the elapsed time for the evaluation of the basis functions. Since the CPU needs to do a lot more work for different basis sets. For larger basis sets, the performance of the basis sets evaluation decreases and causes a big overhead on the total DFT computation. So in the next step, the introduction of the basis function evaluation to the GPU accelerators should significantly improve the overall DFT performance. However, the screening of basis function values is required to have careful consideration.

9.2.2 Evaluation of the Functional Energies

From the profiling of the DFT integrated Molpro code in chapter 5, it has been identified that the functional energy calculation takes a very negligible amount of time in comparison with a full DFT calculation. This algorithmic part is running very fast in CPU for larger block sizes as well. However, this part is in between of the density matrix calculation and the evaluation of the exchange correlation contributions. The exchange correlation contribution matrix depends on the results from the DFT functionals, and the inputs of the DFT functionals depend on the density matrices. So a large number of CPU-GPU memory transactions is required, which creates a big memory latency on the DFT computations. The functional energies are calculated over the grid points and the functionals used, so this module can be exploited for a fine grain parallelization. This will significantly reduce the number of memory transactions.

9.2.3 CPU-GPU Load Balancing

The load balancing between the CPU and GPU needs to be improved further. Now in some extents, CPU has to wait for the calculations to be finished by the associated GPU.

9.2.4 Other Improvements

The analytical gradients in DFT integration are then a good choice for fine grain parallelization. The generation of grid points can also be one of the next step for analyzing the scope for fine grain parallelization.

Bibliography

[1] K. Yasuda, “Accelerating density functional calculations with graphics processing unit,” *Journal of Chemical Theory and Computation*, vol. 4, no. 8, pp. 1230–1236, 2008. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct8001046 (Cited on pages 2 and 24.)

[2] I. Ufimtsev and T. Martinez, “Graphical processing units for quantum chemistry,” *Computing in Science Engineering*, vol. 10, no. 6, pp. 26 –34, nov.-dec. 2008. (Cited on pages 2, 23 and 24.)

[3] L. Genovese, M. Ospici, T. Deutsch, J.-F. Mhaut, A. Neelov, and S. Goedecker, “Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures,” *Journal of Chemical Physics*, vol. 131, no. 3, 2009, cited By (since 1996) 13. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-67651156160&partnerID=40&md5=1c563bcaae6dc9aaad5a4a99e4aee019 (Cited on page 2.)

[4] P. Atkins and R. Friedman, *Molecular quantum mechanics*. Oxford University Press, 2011. [Online]. Available: http://books.google.com/books?id=yk7lRAAACAAJ (Cited on pages 5, 6, 7, 8, 9 and 10.)

[5] H.-J. Werner, “A short introduction to quantum chemistry,” January 2011, unpublished. (Cited on pages 5, 9, 10, 11, 12, 14, 18 and 19.)

[6] F. Jensen, *Introduction to Computational Chemistry*, 2nd ed. Wiley, Dec. 2006. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0470011874 (Cited on pages 9, 10, 11, 12, 13, 15, 18 and 19.)

[7] N. Corporation, *NVIDIA CUDA C Programming Guide*, version 3.2 ed., NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf (Cited on pages 12, 16, 27, 32, 33, 34, 35, 36 and 37.)

[8] P. Hohenberg and W. Kohn, “Inhomogeneous electron gas,” *Physical Review*, vol. 136, pp. B864–B871, Nov 1964. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRev.136.B864 (Cited on page 16.)

[9] C. Lee, W. Yang, and R. G. Parr, “Development of the colle-salvetti correlation-energy formula into a functional of the electron density,” *Physical Review B*, vol. 37, pp. 785–789, Jan 1988. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevB.37.785 (Cited on page 19.)

[10] A. D. Becke, “Density-functional exchange-energy approximation with correct asymptotic behavior,” *Physical Review A*, vol. 38, pp. 3098–3100, Sep 1988. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevA.38.3098 (Cited on page 19.)

[11] P. Brown, C. Woods, S. McIntosh-Smith, and F. R. Manby, “Massively multicore parallelization of kohnsham theory,” *Journal of Chemical Theory and Computation*, vol. 4, no. 10, pp. 1620–1626, 2008. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct800261j (Cited on page 21.)

[12] H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, M. Schütz, *et al.*, “Molpro, version 2010.1, a package of ab initio programs,” Cardiff, UK, 2010, see http://www.molpro.net. (Cited on pages 22, 25, 39, 40, 42, 68 and 73.)

[13] C. J. Woods, P. Brown, and F. R. Manby, “Multicore parallelization of kohnsham theory,” *Journal of Chemical Theory and Computation*, vol. 5, no. 7, pp. 1776–1784, 2009. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct900138j (Cited on pages 22 and 23.)

[14] I. S. Ufimtsev and T. J. Martinez, “Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation,” *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct700268q (Cited on page 23.)

[15] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*, ser. Software patterns series. Addison-Wesley, 2005. [Online]. Available: http://books.google.com/books?id=2ZpQAAAAMAAJ (Cited on page 25.)

[16] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010. [Online]. Available: http://books.google.com/books?id=49OmnOmTEtQC (Cited on page 27.)

[17] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, “Gpu computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879 –899, may 2008. (Cited on pages 27, 29 and 31.)

[18] K. Fatahalian and M. Houston, “A closer look at gpus,” *Communications of the ACM*, vol. 51, pp. 50–57, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1400181.1400197 (Cited on page 28.)

[19] NVIDIA, “Nvidias next generation cuda compute architecture: Fermi, Whitepaper,” 2011. (Cited on page 30.)

[20] K. B. Rita Borgo, “State of the art report on gpu,” *Visualization & Virtual Reality Research Group report*, 2009. (Cited on page 31.)

[21] H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, and M. Schtz, “Molpro: a general-purpose quantum chemistry program package,” *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2011. [Online]. Available: http://dx.doi.org/10.1002/wcms.82 (Cited on pages 39 and 40.)

[22] “Maple soft.” [Online]. Available: http://www.maplesoft.com/products/maple/index.aspx?L=E (Cited on page 40.)

[23] T. Helgaker, P. Jørgensen, and J. Olsen, *Molecular electronic-structure theory.* Wiley, 2000. [Online]. Available: http://books.google.com/books?id=2G8vAQAAIAAJ (Cited on page 42.)

[24] A. Burrows, A. Parsons, G. Price, J. Holman, and G. Pilling, *Chemistry3: Introducing Inorganic, Organic and Physical Chemistry.* Oxford University Press, 2009. [Online]. Available: http://books.google.com/books?id=MVflPQAACAAJ (Cited on page 47.)

[25] N. Corporation, *CUDA CUBLAS Library*, NVIDIA Corporation, August 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf (Cited on pages 51, 63, 64, 66 and 67.)

[26] “Netlib/blas.” [Online]. Available: http://www.netlib.org/blas/ (Cited on pages 63 and 66.)

[27] N. Corporation, *CUDA API Reference Manual*, ve ed., NVIDIA Corporation, February 2011. (Cited on pages 65 and 66.)

[28] H.-J. Werner, F. R. Manby, and P. J. Knowles, “Fast linear scaling second-order mϕller-plesset perturbation theory (mp2) using local and density fitting approximations,” *The Journal of chemical physics*, vol. 118, no. 18, pp. 8149–8160, 2003. [Online]. Available: http://dx.doi.org/10.1063/1.1564816 (Cited on pages 67 and 68.)

[29] M. Harris. Optimizing parallel reduction in cuda. CUDA Webinar 2. NVIDIA Developer Technology. (Cited on page 69.)

[30] “Chemical compounds database.” [Online]. Available: http://www.chembase.com/mol_6339.htm (Cited on page 74.)

[31] H.-J. Werner, “Df-mp2 calculations on gpu-accelerators.” (Cited on page 85.)

Declaration - Erklärung

Declaration

This is to certify that:

i. the thesis comprises only my original work towards the master degree

ii. due acknowledgment has been made in the text to all other material used

Bishwajit Mohan Gosswami
14. November 2011

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Bishwajit Mohan Gosswami
14. November 2011