TÉCNICO LISBOA

Accelerating Molecular Mechanics Applications

Intermediate Report

Sebastião Salvador de Miranda

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Júri

Presidente:	Doutor Nuno Cavaco Gomes Horta
Orientador:	Doutor Pedro Filipe Zeferino Tomás
Vogal:	Doutor Gabriel Falcão Paiva Fernandes

Janeiro 2014

Abstract

Molecular simulations play an important role in computational chemistry, computational biology and computer-aided drug design. Traditional single core implementations have very prolonged runs and do not exploit the intrinsic data and task parallelism of some of these methods. Following the recent trend of high performance heterogeneous computing, the purpose of this work is to accelerate a novel Perturbative Monte Carlo (PMC) mixed Molecular Mechanics (MM)/Quantum Mechanics (QM) simulation algorithm under development by the Institut für Physikalische Chemie, Georg-August-Universität Göttingen. The multi-platform OpenCL programming framework will be used, targeting parallel heterogeneous systems constituted by multi-core Central Processing Units (CPUs) and Graphical Processing Units (GPUs). Performance evaluation of the PMC MM/QM OpenCL implementations for several different heterogeneous systems will be assessed. Preliminary acceleration results for a one-CPU/one-GPU system are presented in this report, yielding speedups ranging from 16 to 28 times regarding the original single-core version, thus hinting that the PMC algorithm is a good candidate for parallelization. Future work will focus on improving the parallel implementation to efficiently scale to multiple heterogeneous devices.

Keywords

Molecular Mechanics, Quantum Mechanics, Monte Carlo Simulations, Heterogeneous Computing, OpenCL, GPGPU.

Resumo

As Simulações moleculares desempenham um papel crucial em várias áreas científicas como a química computacional, biologia computacional e o desenvolvimento de fármacos assistido por computador. No entanto, implementações tradicionais single-core têm execuções prolongadas e não aproveitam o paralelismo de dados e de tarefas intrinsecamente presente nalguns destes tipos de simulações. Seguindo a tendência recente de computação heterogénea, o objetivo deste trabalho é acelerar o algoritmo de simulação molecular Perturbative Monte Carlo (PMC) mixed Molecular Mechanics (MM)/ Quantum Mechanics (QM) que está a ser desenvolvido no Institut für Physikalische Chemie, Georg-August-Universität Göttingen. A ferramenta multiplataforma de programação heterogénea, OpenCL, será utilizada, com o objetivo de programar sistemas heterogéneos constituídos pelas arquiteturas multi-core Central Processing Units (CPUs) e Graphical Processing Units (GPUs). O desempenho de várias implementações OpenCL do algoritmo PMC MM/QM será avaliado em diferentes sistemas heterogéneos. Resultados preliminares de aceleração para um sistema composto por um CPU e um GPU são apresentados neste relatório, tendo sido obtidos speedups de 16 a 28 tendo em conta uma execução da versão original num sistema single-core, sugerindo que o algoritmo PMC é um bom candidato para paralelização. O trabalho futuro a ser desenvolvido irá incidir em melhorar a implementação paralela do algoritmo para escalar eficientemente para vários dispositivos heterógenos.

Palavras Chave

Mecânica Molecular, Mecânica Quântica, Simulações Monte Carlo, Computação Heterogénea, OpenCL, GPGPU.

Contents

List of Figures

List of Tables

List of Acronyms

MD	Molecular Dynamics
MC	Monte Carlo
MM	Molecular Mechanics
QM	Quantum Mechanics
PMC	Perturbative Monte Carlo
VDW	Van der Waals
CPU	Central Processing Unit
GPU	Graphical Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
ILP	Instruction Level Parallelism
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
SPMD	Single Program Multiple Data
PE	Processing Element
CU	Computing Unit
GPC	Graphics Processing Cluster
PC	Program Counter
LDS	Local Data Share
SI	Southern Islands
SMX	Streaming Multiprocessor
CC	Compute Capability

1

Introduction

Computer simulation of chemical processes is a modern alternative to traditional laboratory experiments, allowing a very detailed study of chemical reactions, while saving time and resources. Using methods from theoretical chemistry, where mathematics and physics are used to study chemical processes, computational chemistry is concerned with studying the properties of a chemical system, describing inter molecule interaction, geometrical arrangements and other chemically related problems. A particular case of molecular computer simulation is drug docking simulation, which is a method that predicts the preferred configurations of two molecules when bounding to each other and plays a crucial role in the lengthy process of computer-aided drug design [14], to which thousands of lives are tied.

In molecular computer simulations, execution time and memory scale rapidly with the size of the system being simulated, leading to prolonged runs (sometimes in the order of weeks or months) and resulting in wasted time and energy. Mature implementations of computational science software are usually highly optimized for traditional single core Central Processing Unit (CPU) architectures, and therefore are intrinsically limited by advances in single core execution time. To tackle this limitation, more recent High Performance Computing (HPC) solutions have been exploiting the advances in parallel and heterogeneous computing, using parallel hardware such as multi-core CPUs and many-core Graphical Processing Units (GPUs) and specialized accelerators such as Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs).

Two typical approaches to molecular simulation are Molecular Dynamics (MD) and Monte Carlo (MC). MD simulates the system by calculating the current forces in each iteration, computing the resulting particle velocities and apply those velocities to each particle, letting them move over a small increment of time. By following the system's time evolution, MD allows studying dynamic properties such as diffusion constants and viscosity. To obtain usable results, the time step between iterations has to be very small (in the order of the femtosecond) and this limits MD simulations to the order of microseconds. Conversely, MC samples the system in the ensemble space[1] rather than in time evolution, by generating a large set of system configurations and measuring the average of some thermodynamic property of interest. Contrary to MD, MC does not enable the computation of dynamical quantities but allows the study of processes which take a long time, where sampling in time would prove to be infeasible. Approaches

[1] The set of all possible system configurations.

based on the Metropolis MC [21] require the computation of system energy, which may be computed using traditional Molecular Mechanics (MM), Quantum Mechanics (QM) or mixed QM/MM methods. MM approaches simulate molecules using traditional ball and spring modelling and represent molecules as their atomic nuclei with point charges. However, for simulating particles with wave characteristics, MM representation does not suffice and a QM description must be used instead, which usually involves solving an approximated version of the Schrödinger equation. Because this is a very computational intensive process, mixed QM/MM simulate small portions of the system with great detail using QM, keeping the remainder represented by classical MM.

Collaboration between INESC-ID/IST, Instituto Superior Técnico, Universidade de Lisboa and Institut für Physikalische Chemie, Georg-August-Universität Göttingen, led to this work, with the objective of accelerating their novel algorithm for Perturbative Monte Carlo (PMC) mixed MM/QM simulation of periodic systems [30]. Generally, the purpose of this model is describing explicitly a system composed of a solvent and a solute. For the solvent, the faster and less precise classical MM force field methods are used, whereas the solute is represented by an electronic charge distribution in space and it's interaction with the solvent governed by QM. Ultimately, the purpose of the model is to simulate the docking of drugs in the active site of a protein using QM, taking the surrounding environment into account explicitly using MM. This is in contrast with models that don't have an explicit description of the environment (i.e molecules of solvent), using implicit descriptions instead (continuous model, or a function of some physical variable, such as permittivity, dielectric constant, etc.). While in MD the computational bottleneck is in each iteration, MC iterations are much lighter but thousands have to be executed in order to fully sample the system, leading to prolonged runs (as presented further, typical runs may take hundreds of days).

As will be discussed in this text, there are many opportunities for exploiting different types of parallelism in PMC MM/ QM, making it an excellent candidate for acceleration with heterogeneous parallel architectures. Most of the works done in parallelizing Computational Chemistry algorithms with GPUs use Nvidia's CUDA framework [23], as will be described in Section 1.2. Conversely, in this work, the multi-platform multi-paradigm parallel programming framework, OpenCL [15], will be used. The reason for this choice is that OpenCL allows targeting GPUs from other vendors (e.g. AMD, Intel), multi-core CPUs, DSPs and FPGAs [26]. Also, OpenCL enables writing heterogeneous code, providing mechanisms for synchronization and communication between an OpenCL Host (CPU) and the target OpenCL compute devices (e.g.: CPUs, GPUs, DSPs, FPGAs).

1.1 Perturbative Monte Carlo MM/QM

Contrary to Molecular Dynamics which follows the system's time evolution, Monte Carlo Molecular Mechanics samples the system in the ensemble space. As depicted in Figure 1.1, at each MC iteration

a perturbation is added to the MM system[2], by selecting, rotating and translating a random MM molecule (*chmol*). After this perturbation, the Van der Waals (VDW) and Coulomb energy of the system must be re-computed using Molecular Mechanics for the interactions between the *chmol* and the MM solvent and Quantum Mechanics between the *chmol* and the QM solute. MC iterations are accepted if the energy of the obtained configuration is lower than the previous configuration reference (which is the last accepted configuration) or accepted with a probability $e^{\frac{\Delta E}{K_B T}}$[3] if the energy of the system has risen. The output of the program is a file with the accepted system configurations, which correspond to the xzy Cartesian coordinates of the MM molecules and the different energy contributions of the system.

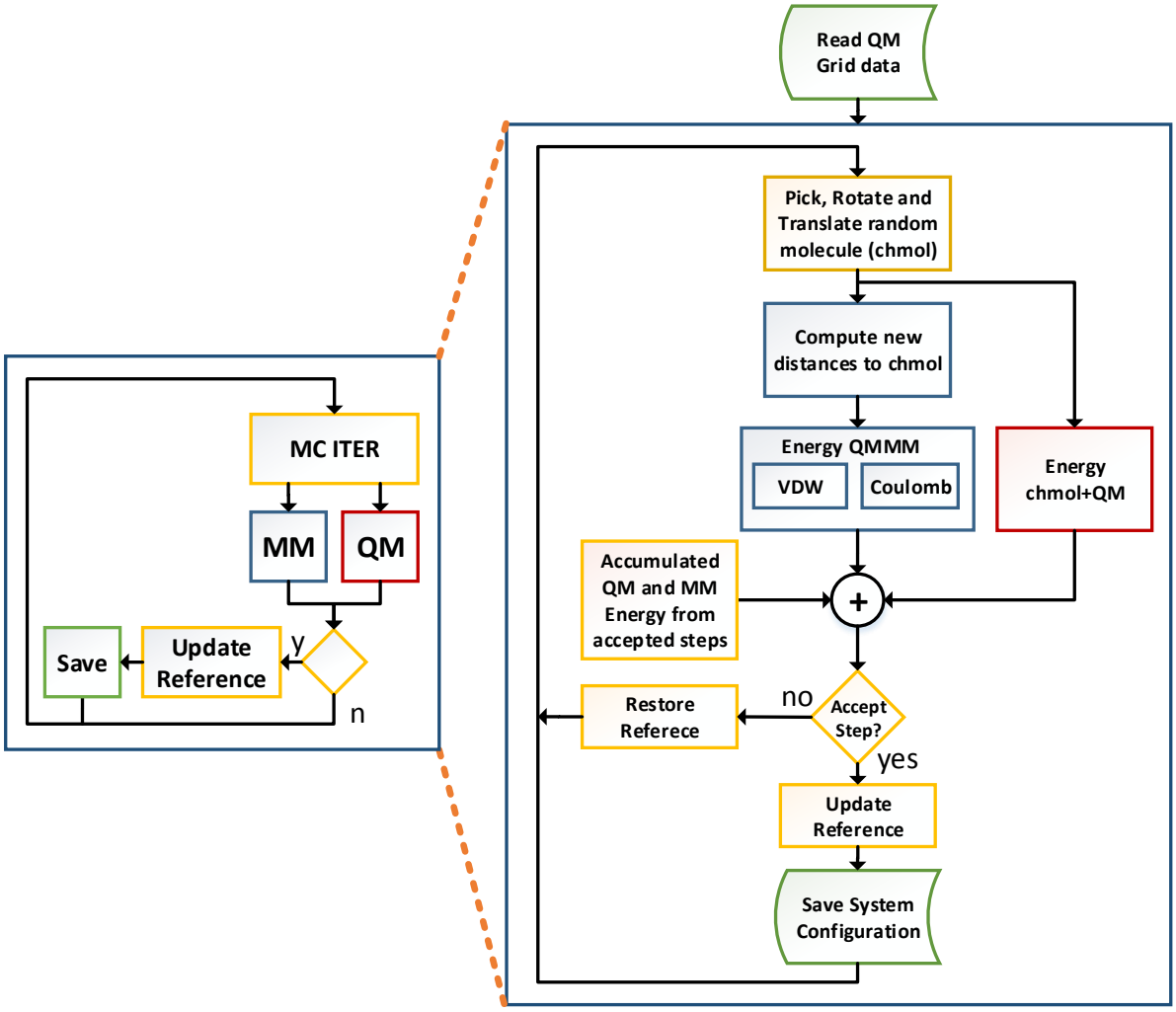

Figure 1.1: Perturbative Monte Carlo (right) along side with an abridged diagram (left). Arrows represent data dependencies.

The heaviest step in the algorithm is the computation of the energy contribution from the Coulomb interaction between the electronic density distribution that represents the QM molecule and the point

[2] In this text, MM system stands for the set of molecules that are part of the solvent (simulated with Molecular Mechanics) and QM molecule stands for the solute molecule.

[3] Boltzmann distribution, where K_B stands for the Boltzmann factor and T for temperature.

charges that represent the *chmol* (one per atomic nuclei), corresponding to the red QM procedure in Figure 1.1. The expression for this energy contribution is the integral in equation (1.1) for each atom j of *chmol* [20], where $\rho(\mathbf{r})$ is the electronic density function.

$$E_j = -\int \rho(\mathbf{r}) \frac{q_j}{r_j(\mathbf{r})} d\mathbf{r} \tag{1.1}$$

One approach to solve the integral in equation (1.1) discussed in [30] is to compute it numerically by describing the electronic charge distribution by a fine grid of points. Essentially, this reduces equation (1.1) to equation (1.2).

$$E_j = \sum_{i}^{N_{GRID}} \frac{q_i q_j}{r_{ij}} \tag{1.2}$$

The QM grid is constituted of N_{GRID} point charges and is read at the beginning of the PMC algorithm. To effectively sample the conformational space of the QM molecule, the QM grid must be updated every K iterations by another program which can be run in parallel. This execution flow is represented in Figure 1.2, where the abridged version of Figure 1.1 was used. A typical run simulates over ~ 1000 MM atoms interacting with each other and a ~ 280000 point charge grid, taking about $M = \sim 100k$ steps each comprising $K = \sim 10k$ PMC iterations. In fact, running the original version [30] in one core of an Intel i7 3820 CPUtook $\sim 230s$ per PMC iteration, so it would take ~ 266 days to finish the complete run.

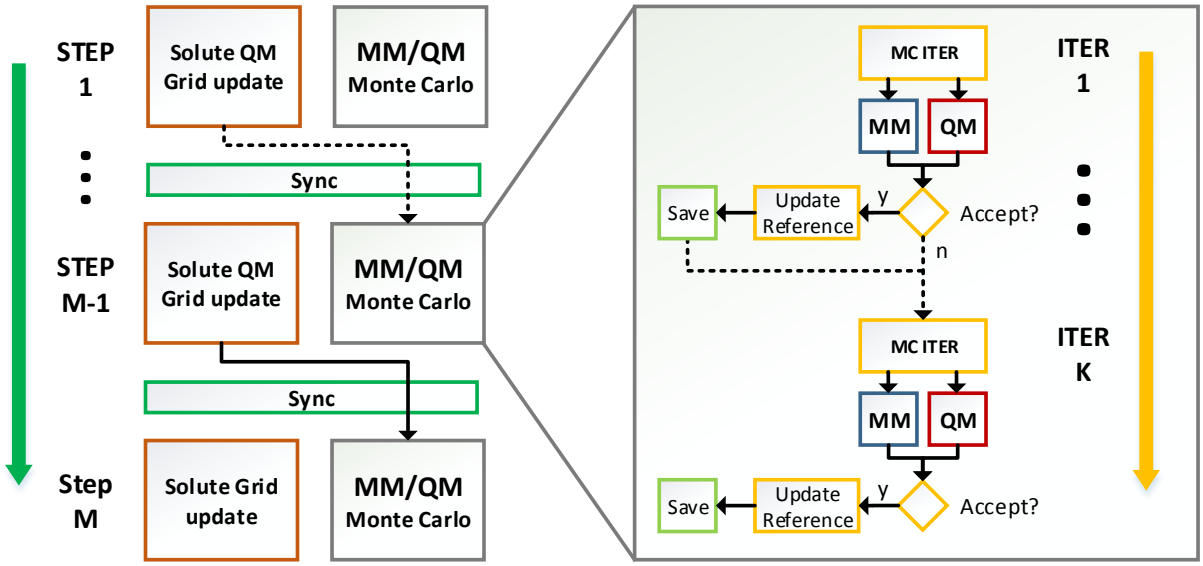

Figure 1.2: Complete algorithm flow for M steps of K PMC iterations each.

1.2 Related Work

There is extensive work on accelerating computational chemistry algorithms. In [5], a complete GPU implementation for molecular dynamics simulation is presented and a novel neighbour-list method,

the space-filling curve pack (SFCPACK), is introduced and discussed. This method, in contrast with traditional CPU particle partitioning algorithms (such as the optimized neighbour-list method introduced in [31]), is tuned for the GPU architecture and the considered MD routines. A neighbour-list is a data structure (e.g. Matrix or vector of Linked Lists) that records, for each particle, the particles that are close enough to it, according to a cutoff. This spares unnecessary computation over molecules which are already known to be outside the interaction cutoff.

In [29], the authors present a GPU-based Molecular Dynamics simulation for the study of fluid flows with anisotropic molecules. In particular, intermolecular force and torque computations are parallelized in the GPU, yielding a speedup of 50 in respect to the serial version. The authors concluded that using an optimized neighbour-list such as described in [5] added too much overhead and decided to used a cell-list method instead. In this method, the system is partitioned in cells, with dimension equal to the interaction cutoff and N_{MC} maximum molecules per cell. This way, interactions only need to be computed between each molecule and the molecules in their own cells and neighbour cells, but cells need to be updated every MD step according to the new coordinates of the molecules. They report results for two thread-block organization schemes: In the first one, memory access patterns are optimized by defining one thread-block per cell and making thread blocks of $N_{TB} = \lceil \frac{N_{MC}}{warpsize} \rceil$ threads (this results in thread redundancy if N_{MC} is not a multiple of $warpsize$). In the second one, they minimize thread redundancy by using a predefined number of threads per block N_{TB} and computing the number of blocks $N_B = \lceil \frac{N_{mol}}{N_{TB}} \rceil$, resulting in the sacrifice of the better memory access pattern and locality the first method benefited from, by having each cell in its own work-group. They conclude that using the first approach results in 10% to 40% speedup in respect to the first one. Although this text is about parallelizing MC MM/ QM instead of MD, the same principle of organizing particles in a neighbour-list or cell-list may improve the Coulomb QM step, as will be discussed further.

In [10], earlier work for paralelizing Monte Carlo algorithms for molecular mechanics in a CPU cluster environment is presented. Like the reference for the algorithm parallelized in this text [30], [10] builds over Metropolis Monte Carlo, the main difference being the latter does not consider Quantum Mechanics. The problem is benchmarked for a small network of 2 or 4 CPU nodes, working in a master-slaves relationship where the master is responsible for taking the MC step and broadcasting the changed molecule to the slave processes, each in charge of computing interactions between the changed molecule and their subset of molecules.

Specific acceleration of the PMC MM/QM algorithm introduced in [30] is not available, nor is any parallelization of other Monte Carlo mixed MM/ QM algorithm, to my knowledge. Also, in most publications the programming framework of choice is CUDA and the target hardware are Nvidia GPU's (an exception being for example [13], which also targets ATI boards using Brook stream programming language). Conversely, in this work, OpenCL will be used to allow multi-platform heterogeneous computing, as justified in the introduction of this report. Other frameworks have been developed to make programming for non-conventional architectures more accessible, such as IBM Lime [6] and StarPU [7]. An example of a

more application specific library is OpenMM [12] (for which the work in [13] has contributed), which offers software framework for molecular modelling applications. However, unlike those tools, OpenCL is a an open standard supported by large chip manufacturers (e.g. AMD, Intel) and is extensively documented and endorsed by a larger user base.

1.3 Objectives

The objective of this MsC thesis is to accelerate the execution of the PMC MM/QM algorithm introduced in [30] by designing a parallel heterogeneous implementation for target state-of-the-art hardware architectures, using the multi-platform multi-paradigm parallel programming framework, OpenCL. Performance will be assessed in several system configurations:

(i) **1xCPU+1xGPU**

(ii) **multi-core CPU+1xGPU**

(iii) **1xCPU+2xGPUs (equal)**

(iv) **1xCPU+2xGPUs (different)**

The purposed work is delineated in more detail in Section 3.5.

1.4 Document Outline

The remaining of this report is organized as follows: In chapter 2, a description of target state-of-the-art CPU and GPU hardware and the OpenCL framework will be presented, as well as programming considerations to be taken when targeting this architectures. In chapter 3, follows an analysis of the PMC MM/QM from a computational point of view accompanied by a serial benchmark of the algorithm and several possible acceleration approaches, finishing with the detailed work proposal for this MsC thesis. The current intermediate results are also reported and discussed. Finally, in chapter 4, the planning for the development of this MsC thesis is delineated and the major milestones listed.

2

Heterogeneous Computing

GPU and CPU architectures are considerably different and it is possible to take advantage of both to accelerate an application. The GPU architecture provides many cores, each substantially simpler than a CPU core, trading off single-thread for multi-thread performance. The GPU achieves high-throughput by hiding thread memory access latency with intensive arithmetic operations from other threads and by rapidly switching execution context between groups of threads. It should be noticed that context switching has very little cost in comparison to CPU threads because hundreds of thread contexts are stored on-chip[1]. Conversely, state-of-the-art multi-core CPU architectures offer a few but highly complex cores, using techniques to exploit Instruction Level Parallelism (ILP) and multiple levels of caches to accelerate main memory access. This higher complexity results in increased area and power consumption which means multi-core CPU's only include a small number of cores.

When targeting a CPU/GPU heterogeneous environment, the application must be partitioned to efficiently take advantage of both architectures. Code with intensive flow-control or low threaded data and functional parallelism should be kept on the CPU, whereas arithmetic intensive large data parallel code should be executed in the GPU. The partitioning should in general minimize communication between the CPU and the GPU, but the trade-off between heavier communication costs and faster execution (e.g. executing single threaded code on the GPU) is not always obvious and a complete study of the program must be done in order to find the best partitioning scheme. Work Partitioning schemes may be static or dynamic. Static partitioning is limited in many applications which have unpredictable Load Balancing, because it may cause idle devices waiting for the device which got an unexpectedly heavy code portion. OpenCL does dot offer any tools for dealing with dynamic scheduling between Devices, and so the programmer is responsible for writing the scheduler, if it so desires. Dynamic scheduling is specially important for the case of Heterogeneous systems, because devices have different performance depending on the type of parallelism that can be extracted from a given piece of work. This is the case of a system composed of both CPU's and GPU's. Several authors have studied dynamic scheduling to complement standard heterogeneous frameworks, such as libraries for the CUDA framework ([11], [1], [8]) and the Maestro library for the OpenCL framework[27].

[1] For the case of the NVIDIA GK104/GK110 architectures, the maximum number of resident threads per Multiprocessor is 2048 [22].

In this chapter, an overview of both state-of-the-art CPU and GPU is presented, followed by an introduction to the OpenCL programming framework.

2.1 Multi-core Central Processing Unit (CPU)

State-of-the-art mainstream multi-core CPU architectures offer a few but highly complex CPU cores. Very fast memory is available through the use of registers local to each core and access to the larger but slower main memory is made via several levels of caches. In Figure 2.1, a typical example for a multi-core CPU architecture with three layers of cache (2 private and 1 shared) is displayed. Several hardware techniques are employed to accelerate single-threaded execution, such as increasing the clock frequency through multi-stage hardware pipelining, resulting in higher instruction throughput. Furthermore, modern architectures exploit ILP using super-scalar and out-of-order instruction execution. While the former allows executing instructions in parallel in the available functional units (in case of no data, control or structural hazard), the latter enables switching the order of independent instructions to reduce processor stalls. Program flow control overhead is mitigated by branch prediction hardware and ultimately by allowing speculative execution. Latency caused by inevitable processor stalls may be further hidden by hardware multi-threading, which allows fast context switching between threads in the same processor.

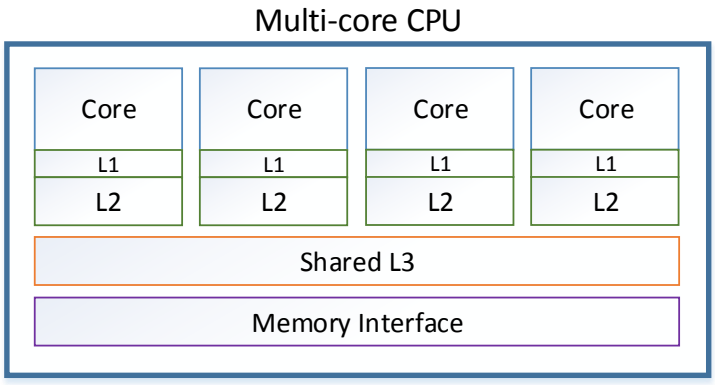

Figure 2.1: Generic CPU with 4 cores and 3 levels of cache.

Higher hardware complexity results in increased area and power consumption per core, which is the reason why multi-core CPUs only include a small number of cores relatively to GPUs. This means that the type of parallelism that can be extracted from a multi-core CPU architecture is also more coarse grained, typically leading to the application of the Multiple Instruction Multiple Data (MIMD) parallel programming paradigm. Furthermore, Communication between threads in different CPU cores is much more expensive than communication between threads in cores of the same thread-block in a GPU. In modern CPUs, each core offers vector instructions that enable the extraction of Single Instruction Multiple Data (SIMD) parallelism (ex Intel SSE/AVX instructions), making the multi-core CPU a very versatile parallel platform. Although it does not match the GPU in terms of floating point operations per

second for highly data parallel applications, it is more efficient for algorithms with complex control flow or very coarse grained parallel structure.

2.2 Graphical Processing Unit (GPU)

Graphical Processing Units are many core processors originally built to accelerate graphic computations. In the last few years, vendors have enabled their hardware to run general purpose applications, leading to a paradigm called GPGPU (GP stands for General Purpose). Nvidia and AMD released proprietary GPU programming languages, respectively CUDA and CTM, but lately AMD has embraced the open standard OpenCL, which is also supported in Nvidia GPUs, Intel CPUs and embedded GPUs, and a multitude of other devices. In this section, an overview of AMD's Southern Islands and Nvidia's Kepler architectures is presented.

2.2.1 AMD and Nvidia Architectures

The current high-end state-of-the-art AMD device architecture is Vulcanic Islands, although in this section the Southern Islands (SI) (HD7000 family) is described instead, due to wider availability of hardware and technical material. For the case of the SI-GPU device architecture depicted in Figure 2.2, the

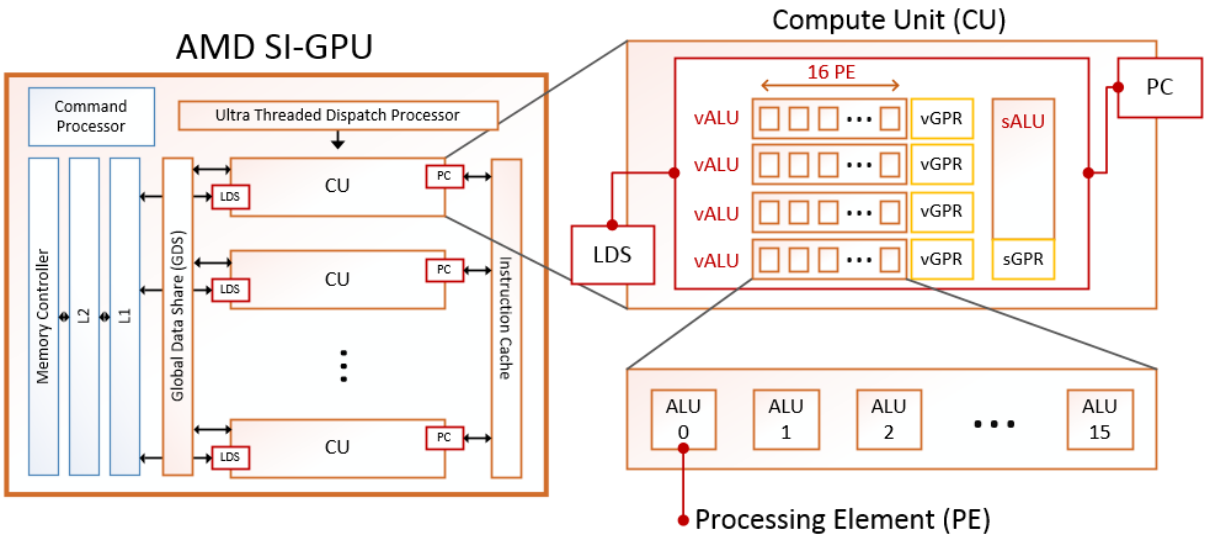

Figure 2.2: Organization of the AMD SI-GPU.

GPU is composed of several compute units. Each Computing Unit (CU) has one scalar unit and 4 vector units composed of an array of 16 processing elements (PEs) each. Local to each CU, there are also five banks of vector and scalar General Purpose Registers (vGPR/sGPR) and Local Data Share (LDS) memory. The instruction issue takes four cycles where the four 16-Processing Element (PE) arrays execute 64 work-items in total. The resulting 64 element vector is called a wavefront[2]. Processing elements

[2]Other AMD graphic card families may have different wavefront sizes.

within a compute unit execute in lock-step, whereas compute units execute independently in respect to each other. Lock-step execution might pose problems if work-items from the same wavefront fall on different branch paths, in which case all paths must be executed serially, thus reducing efficiency. This is because work-items from the same wavefront share the same Program Counter (PC). For the case of this device architecture family, the four arrays of 16-PE execute code from different wavefronts.

The current state-of-the-art of NVIDIA device architecture is *Kepler* (Compute Capability (CC) 3.X). An example device from this family is the GeForce GTX 680 GPU, which includes the Kepler chip GK104. This particular GPU is composed of 4 Graphics Processing Clusters (GPCs) each with 2 Streaming Multiprocessor (SMX), and 4 memory controllers. Each SMX is in turn composed of 192 CUDA cores (roughly equivalent to the AMD ALUs presented earlier). The warp of a GPU is a number of threads representing the finest grain of instruction execution of a multi-processor. For the Tesla, Fermi and Kepler device architectures, this number is 32, meaning that at least 32 threads must execute the same instruction. A warp is roughly the equivalent of a wavefront in AMD hardware. Each SMX contains 4 warp schedulers that dispatch two instructions per warp with active threads, every clock cycle. A thread

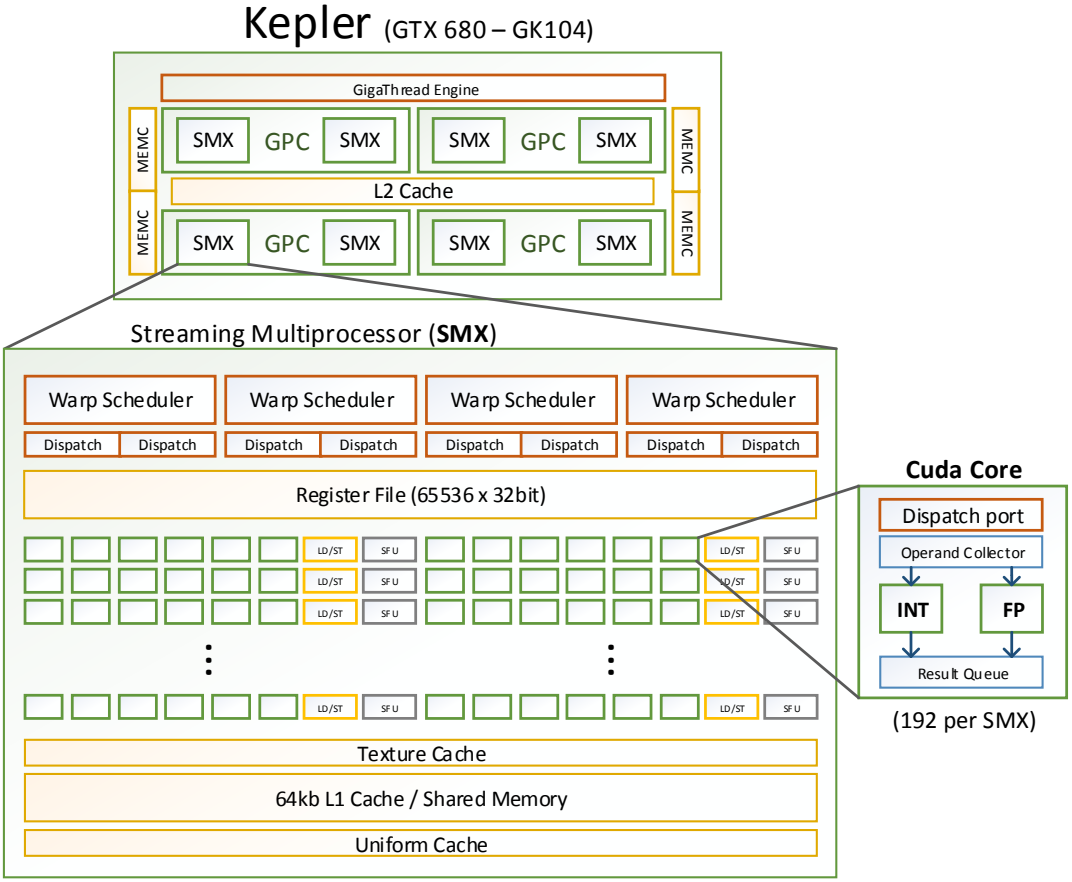

Figure 2.3: GTX 680 device Architecture.

is said to be active if it is on the warp's current execution path, otherwise it is inactive. When a threads in the same warp follow different execution paths, the warp is said to be diverging (the same thing happens

for wavefronts in AMD hardware). For example, in the case of a kernel with a branch where some threads fall on different branch sections, all sections must be executed serially, reducing parallel efficiency. To eliminate this problem, the programmer must try to align branch results with Warp/Wavefront boundaries, enforcing that threads from the same Warp/Wavefront always fetch the same instructions. This might not be possible at all if the branching is not predictable, or change the algorithm memory access patterns in such way it does not compensate the gains acquired by reducing divergence. In older GPU devices (i.e. Nvidia's CC 1.X), optimal memory access access patterns where restricted to a fairly reduced set, but in CC 3.X the allowed patterns that still offer optimal performance are much more relaxed [23].

2.3 OpenCL

As previously mentioned several paradigms exist for programming both CPUs and GPUs. However, unlike most alternatives, OpenCL is supported by several different platforms, such as GPUs from multiple vendors, multi-core CPUs, DSPs and FPGAs [26]. Also, OpenCL simplifies orchestrating multiple devices in an heterogeneous environment and allows writing portable code between different architectures. Thus, OpenCL has been chosen for performing the proposed work.

OpenCL is organized in a hierarchy of models [15]: Platform Model, Execution Model, Memory Model and Programming Model. Each of these models is explained in the following sections. The OpenCL framework includes the OpenCL compiler (OpenCL C), the OpenCL platform layer and OpenCL Runtime. Currently, the standard is in version 2.0, although in this project the previous version (1.2) is being used due to wider device support.

2.3.1 Platform Model

The Plaform Model defines how a program maps into the OpenCL platform, which is an abstract hardware representation of the underlying device. As depicted in Figure 2.4, the platform model is composed of a Host connected to one or multiple OpenCL devices. An OpenCL device is a collection of CU, which in turn are divided into one or more PE[3], where the computation is done. The code that runs on the host uses the OpenCL Runtime to interface with the OpenCL device, to which it may enqueue synchronization commands, data or kernels. A kernel is a function written in OpenCL C, and can be compiled before or during program execution. Within each CU, PEs can execute either in SIMD or Single Program Multiple Data (SPMD) fashion. In the former, PEs execute in lock-step, whereas in the latter PEs keep their own program counter and may follow independent execution paths.

2.3.2 Execution Model

An OpenCL program executes over an index space in two main components: host code running on the host device and kernel code that runs on each OpenCL Device. Kernel instances are called work-

[3]This structure (and naming) closely resembles the one for AMD devices, presented in section 2.2.1.

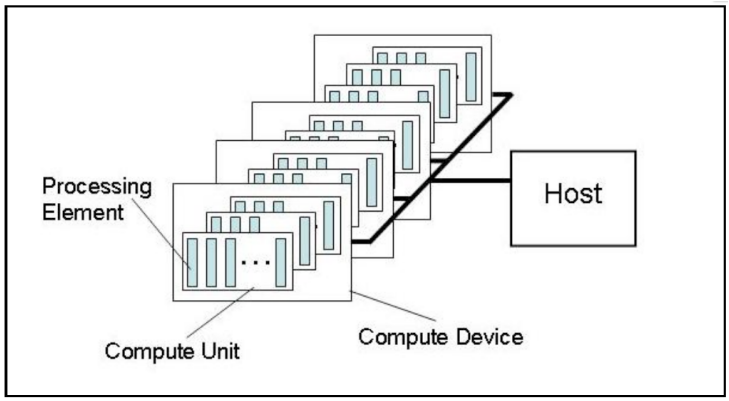

Figure 2.4: OpenCL Platform Model [15].

items and are further grouped into work-groups. Each work-item has a unique identifier in the global index space and in the local index space (local to each work-group). Index spaces are called NDRanges and can have 1,2 or 3 dimensions, thus, the local and global indexes are 1,2 or 3 dimensional tuples. In Figure 2.5, an example of this organization is depicted for 2 dimensions. For GPU devices the best performance should be attained when work-group-size is an integer multiple of warp-size (NVIDIA) or wavefront-size (AMD) because this is the minimum execution granularity supported. Failing to meet this criteria will cause running $work\text{-}group\text{-}size\%warp\text{-}size$ useless threads.

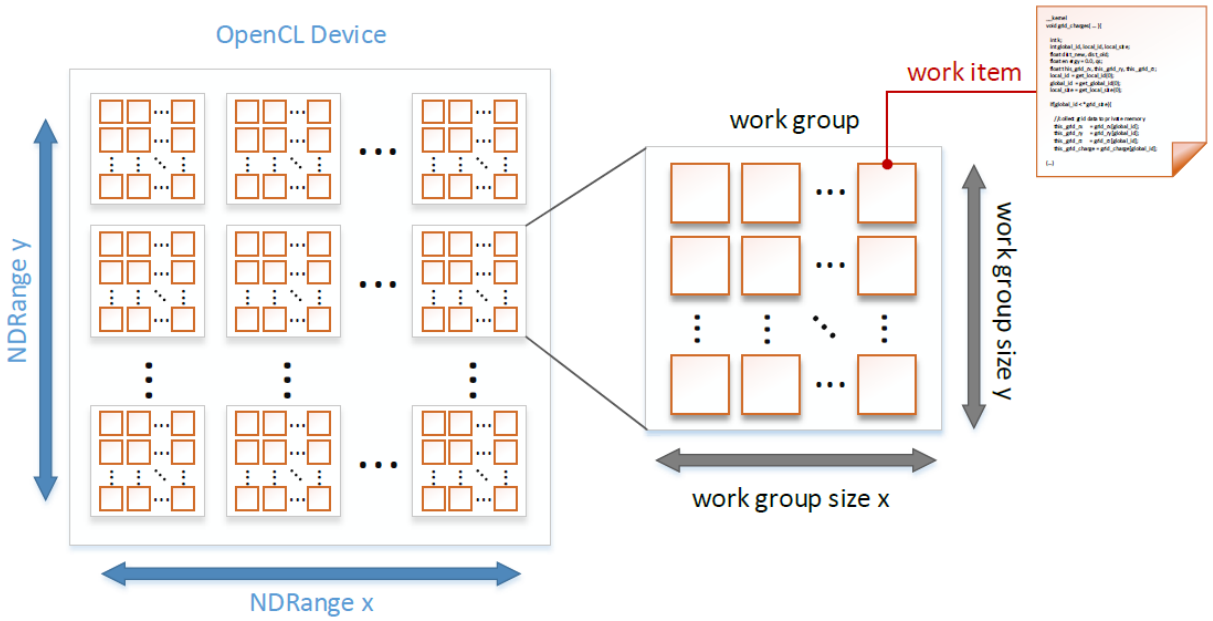

Figure 2.5: Partitioning of work-items into work-groups.

To support different devices with different thread management systems, OpenCL employs a relaxed synchronization and memory consistency model. This way, execution of work-items is not guaranteed to follow any specific order. Nevertheless, explicit work-group barrier instructions can be placed in the

kernel code to ensure execution synchronization between work-items of the same work-group. Synchronization of work-items belonging to different work-groups is not possible during the same kernel launch, a behaviour depicted in Figure 2.6. Memory consistency details are explored in Section 2.3.3.

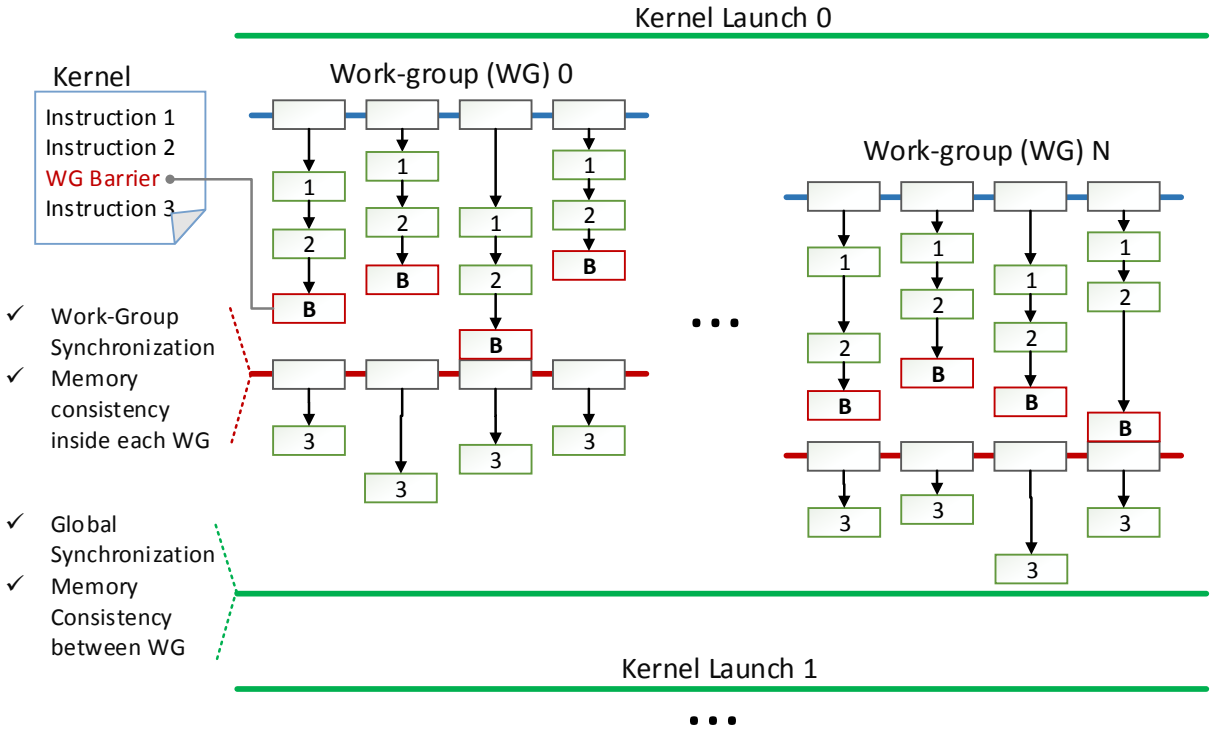

Figure 2.6: Partitioning of work-items into work-groups.

Another important concept is an OpenCL Context, which includes a collection of OpenCL Devices, a set of kernels, a set of Programs (source and compiled binary that implement the kernels) and a set of Memory Objects. Associated with a Context is one or more Command Queues, via which the host enqueues execution, memory and synchronization commands to the OpenCL Devices. This queue may be set as in-order or out-of-order, which defines if the order by which commands are enqueued must be respected or not.

2.3.3 Memory Model

The OpenCL standard defines four memory region types, each having different rules for access and allocation:

(i) **Global Memory:** Accessible by all work-items for read/write operations. The Host has read/write access and is responsible for memory allocation (Dynamic). This memory may be cached or not. AMD SI-GPU and NVIDIA CC 3.3 devices, for example, have global memory caches accessible by each CU. Between work-items of the same work-group, global memory consistency is only guaranteed if they encounter a global work-group barrier. Across different work-groups there is no

guarantee of memory consistency during the execution of a kernel. This behaviour is depicted in Figure 2.6.

(ii) **Constant Memory:** Memory Accessible by all work-items for read operations, remaining constant during the execution of a kernel. Like Global Memory, the Host has read/write access and is responsible for memory allocation (Dynamic). Constant memory is usually cachable and has a lower access latency than Global Memory.

(iii) **Local Memory:** This memory region is shared by work-items of the same work-group for read/write operations. Allocation can be done either statically by a kernel or dynamically by the Host, although the Host cannot access this memory region. It is usually implemented as dedicated memory in each CU, but in some devices it can also be mapped into Global Memory. In AMD SI-GPU, this memory is mapped into LDS (Figure 2.2), whereas in Nvidia's Kepler architecture it is mapped into the Shared Memory (Figure 2.3). Local memory is only consistent between work-items of the same work-group after they encounter a local work-group barrier, as depicted in Figure 2.6.

(iv) **Private Memory:** Memory region private to each work-item, for read/write access. Neither the Host nor other work-items can access this memory. It must be statically allocated in the kernel. It is usually implemented as registers in each CU.

2.3.4 Programming Model

The OpenCL standard supports two programming models: Data Parallel and the Task Parallel. In the Data Parallel programming model, parallelism is exploited by executing in parallel the same set of operations over a large collection of data. Considering computation over data in an array, each work-item executes an instance of the kernel in one array index (strictly data parallel model) or in more (relaxed data parallel model). Hierarchical partitioning of work-items into work-groups can be defined explicitly by the programmer or implicitly by the OpenCL implementation. Conversely, in the Task Parallel programming model, a single instance of the kernel is executed, where parallelism can be extracted by using vector types supported by the device or by enqueueing multiple tasks (different kernels) to the Device. Intel SSE/AVX/AVX2 vector instructions, for example, can be inferred by writing operations with OpenCL vector types(e.g. *float4, int4*).

3

Perturbative Monte Carlo MM/QM Acceleration

As introduced in Section 1.1, the Perturbative Monte Carlo algorithm is a mixed MM/QM molecular simulation program that outputs system configurations (energy and molecule xyz) according to the Metropolis MC sampling rule. At each MC iteration a perturbation is added to the MM system, the interaction energy of the MM and QM systems is recomputed and the step accepted or not, according to the sampling rule.

In this chapter, a description of the performance evaluation metrics that will be used followed by an analysis of the algorithm from a computational view point is presented, where the complexity of the major portions of the algorithm is derived and discussed. In particular, the following computations are analysed: QM Coulomb energy contributions, MM distance computation and MM VDW and Coulomb energy contributions. Following, an analysis of possible approaches to parallelizing those algorithm portions is presented, along side with the revision of the objectives for this thesis.

3.1 Measuring Performance

For measuring the gains of porting the original serial algorithm in [30] to an heterogeneous platform, the metric of choice will be execution time Speedup (3.1) in respect to a single core CPU execution.

$$Speedup = \frac{T_{\text{CPU}-single-core}}{T_{\text{CPU}/\text{GPU}}} \tag{3.1}$$

To have a meaningful comparison between implementations and devices, the CPU core used as a reference to run the serial version of the algorithm must be fixed. Both versions will be compiled with the Intel Compiler (ICC) and the Gnu Compiler (GCC) using the *-O3* optimization flag. Global execution time will be measured using the PAPI [9] library in the OpenCL Host CPU program. For evaluating kernel execution time and data buffer transfers to the GPU, OpenCL Profiling Events give a more accurate measurement.

The target architectures that will be used to evaluate Speedup will be those available at the SIPS Inesc-id research group, which include Intel i7 CPUs, Nvidia GPUs and AMD GPUs.

3.2 Algorithm Analysis and Benchmark

Before introducing the possible parallelization approaches to the problem, it is relevant to discuss a benchmark of the original algorithm implementation done in [30] running on a single core CPU, which is presented in Table 3.1 normalized to one Monte Carlo iteration. The execution time was measured according to Section 3.1, in one core of CPU Intel i7 3820.

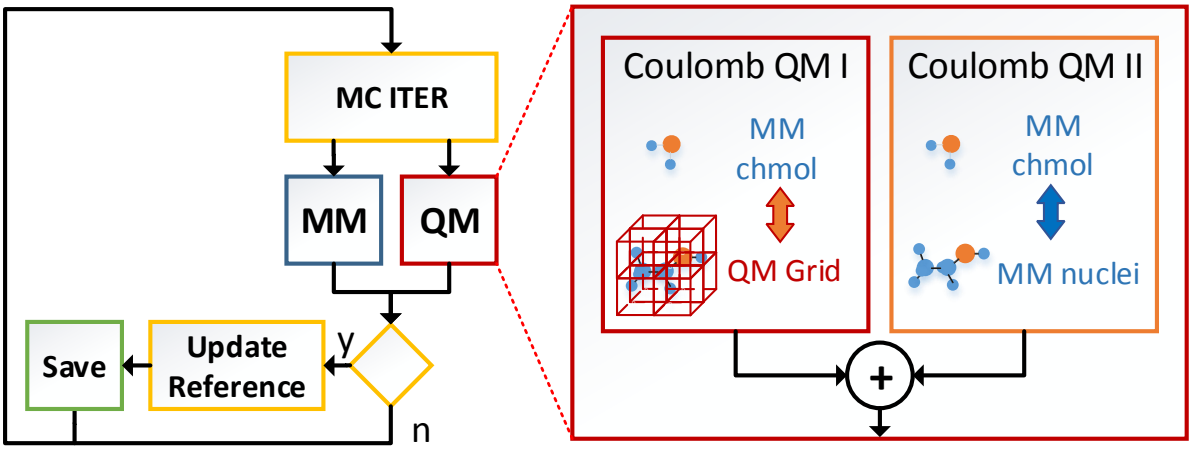

Figure 3.1: Abridged version of the PMC algorithm.

In Figure 3.1, the abridged version of the PMC algorithm (corresponding to the more complete diagram of Figure 1.1) is presented, highlighting an important division of the QM computations into two parts (*Coulomb QM I* and *II*), because they have very different computational weights: *Coulomb QM I* corresponds to interactions between the changed molecule and the QM point charge grid, whereas *Coulomb QM II* represents the interactions between the changed molecule and the QM molecule represented as nuclei charges (MM representation). The former, is the heaviest slice of each MC iteration, whilst the latter corresponds to very light calculations. The pseudocode for Coulomb QM I is presented in Algorithm 1. In each inner iteration, the distance between each atom of the *chmol* and each point from the QM charge grid is computed and then used to select one out of four possible energy contributions, according to two distance cutoffs in respect to the set of coordinates of the changed molecule in the old and new systems (before and after the MC step). Outside of both this cutoffs the energy contribution is regarded as zero.

An optimized serial version was developed to help pinpoint the best strategy to accelerate the code. Having in mind the fact that this is still an intermediate report, effort was not made to make the best serial version possible, just to apply some obvious serial optimizations. The results are presented in Table 3.2. In this optimized version it is even more evident that the most heavy code portion is *Coulomb QM I*, representing 98% of total execution cost. Parallelizing this code portion is discussed in Section 3.3.1.

In Eq. (3.2) and Eq. (3.3), the complexity for the major code sections is presented. N_{MM} stands

Code Section	time(us)	%(MC)
Distances (MM)	1058.13	4.57%
VDW/Coulomb (MM)	2072.40	8.95%
Coulomb I (QM)	**18825.60**	**81.22%**
Coulomb II (QM)	5.45	0.10%
Decision step	1194.79	5.16%
Total	23156.40	

Table 3.1: Original serial version of algorithm benchmark.

Code Section	time(us)	%(MC)
Distances (MM)	55.95	0.29%
VDW/Coulomb (MM)	49.27	0.26%
Coulomb I (QM)	**18805.10**	**98.00%**
Coulomb II (QM)	5.55	0.78%
Decision step	125.8	0.66%
Total	19041.70	

Table 3.2: Optimized serial version of algorithm benchmark.

Algorithm 1 Pseudo code for charge grid energy contribution.

Init: Energy = 0.0

for all atom in changed molecule **do**
  for all point in charge grid **do**
    $d_{old} = distance(atom, point)$ in old system
    $d_{new} = distance(atom, point)$ in new system
    $qs = -point_{charge} \times atom_{charge}$
    if $d_{new} < C$ and $d_{old} < C$ **then**
      $Energy \mathrel{+}= qs \times (\frac{1}{d_{new}} - \frac{1}{d_{old}} + C_{recsq} \times (d_{new} - d_{old}))$
    else if $d_{new} < C$ and $d_{old} \geq C$ **then**
      $Energy \mathrel{+}= qs \times (\frac{1}{d_{new}} - C_{rec} + C_{recsq} \times (d_{new} - C))$
    else if $d_{old} < C$ **then**
      $Energy \mathrel{-}= qs \times (\frac{1}{d_{old}} - C_{rec} + C_{recsq} \times (d_{old} - C))$
    else
      do nothing
    end if
  end for
end for

for the total number of MM molecules, $natom_{chmol}$ the number of atoms in the changed molecule, $natom_{MMj}$ the number of atoms of MM molecule j and N_{GRID} the number of grid point charges in the QM solute grid. The complexity of the distances, MM VDW and Coulomb energy computations is proportional to the sum of the number of atoms of the changed molecule times the number of atoms of each MM molecule. In typical runs the solvent is homogeneous, which means that $natom_{MMj}$ is the same for every j. Nevertheless the general expression for the complexity of each of this three code portions is presented in Eq. (3.2).

$$complexity_{MM} = O\left(\sum_{j}^{N_{MM}} natom_{chmol} \times natom_{MMj}\right) \tag{3.2}$$

On the other hand, the QM Coulomb I computation complexity is proportional to the number of atoms of the changed molecule times the number of points in the charge grid, as show in Eq. (3.3).

$$complexity_{QM} = O(natom_{chmol} \times N_{GRID}) \tag{3.3}$$

For a typical run with 1569 MM atoms (523 water molecules with 3 atoms each) and a QM solute represented by a grid of 268229 points, it can easily be seen that the QM portion, with complexity given by Eq. (3.3), is much more complex.

3.3 Exploiting parallelism in PMC

As will be discussed in this section, both MM and QM computations are extremely data parallel and suitable for the GPU architecture. The MC iteration structure and output file saving however, is a task more suitable to a CPU device and the data flow between all these components makes it unclear (without testing) if one architecture prevails over the other. Therefore, the proposed objective for this work is the performance evaluation of different parallelization approaches for several heterogeneous systems, as will be delineated at the end of this section.

In the following sections possible acceleration approaches for the PMC algorithm will be discussed, as well as some preliminary results. Finally, the objectives of this work are delineated.

3.3.1 Parallelizing QM computations

The charge grid energy contribution (depicted in Algorithm 1) can be partitioned by executing each iteration independently and executing a reduction step over the computed energy terms. Following the OpenCL execution model, a possible parallelization is depicted in figure 3.2. In this partitioning, each work-item computes the distances between one point of the charge grid and all the atoms in the changed molecule. It then proceeds to calculate the coulomb energy contribution for the grid point and to store it in local memory. Then, work-items of the same work-group begin reducing the computed energies, eventually accumulating all terms in one memory address, after $log_2(workgroupsize)$ iterations. After this, the first work-item of each work-group writes the work-group result to global memory and the reduction

kernel must be relaunched with global-size equal to the number of work-groups of the last launch. This process repeats itself until it is no longer worth to parallelize the reduction on the OpenCL device (e.g. GPU) and it may be finished in the Host device. An Example diagram for a CPU/GPU heterogeneous system is depicted in Figure 3.3. Assuming that the CPU host device is multi-core, the MM code can be parallelized among the remaining cores, which will be further discussed in Section 3.3.2.

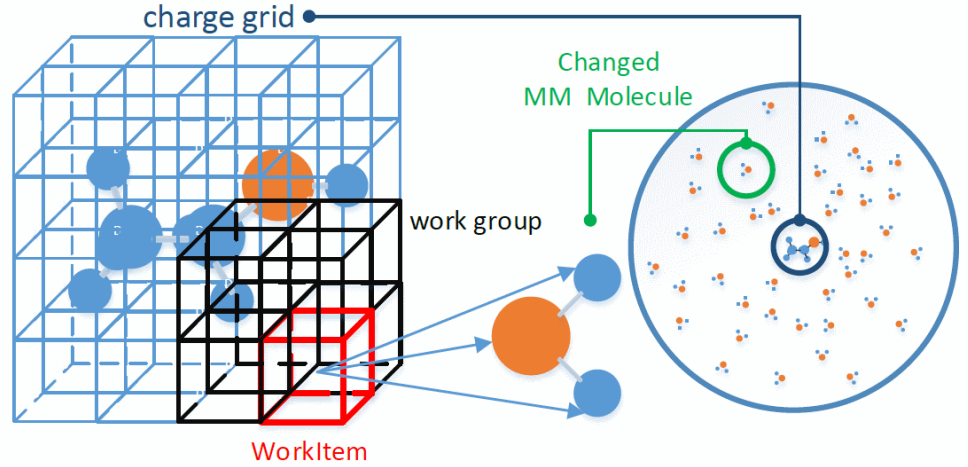

Figure 3.2: QM computation partitioning.

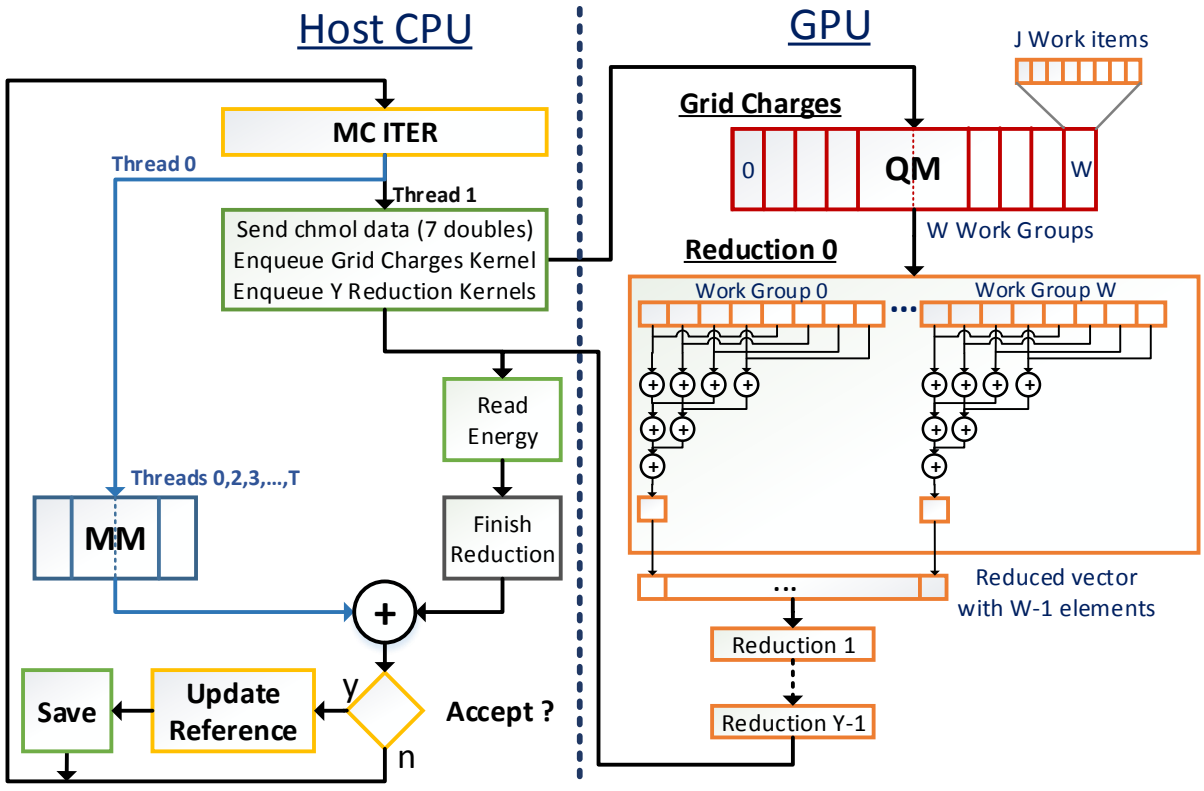

Figure 3.3: Heterogeneous PMC flowchart.

3.3.1.A Work-item divergence

In each iteration of Algorithm 1, corresponding to the OpenCL kernel code, there are four possible branches for work-items to follow, each with a different energy contribution formula. Unfortunately, the branching is only decided after computing the two set of distances, which in the described setting is done in the same kernel of the GPU Device. If work-items from the same work-group fall on different branch sections, then warp/wavefront divergence occurs, resulting in serialized branch execution and loss of parallelism. In practice, for a typical test set, the percentage of warps that diverge per MC iteration is plotted in Figure 3.4. In the 100 samples that were taken from $10k$ MC iterations, severe warp-divergence can be observed in a few cases, whereas in most cases it is below 25%. More concerning than this is that in a run comprising $10k$ MC iterations, on average 50% of work-items are falling in the branch section which computes no energy contribution, meaning that those work items could have been skipped in the first place.

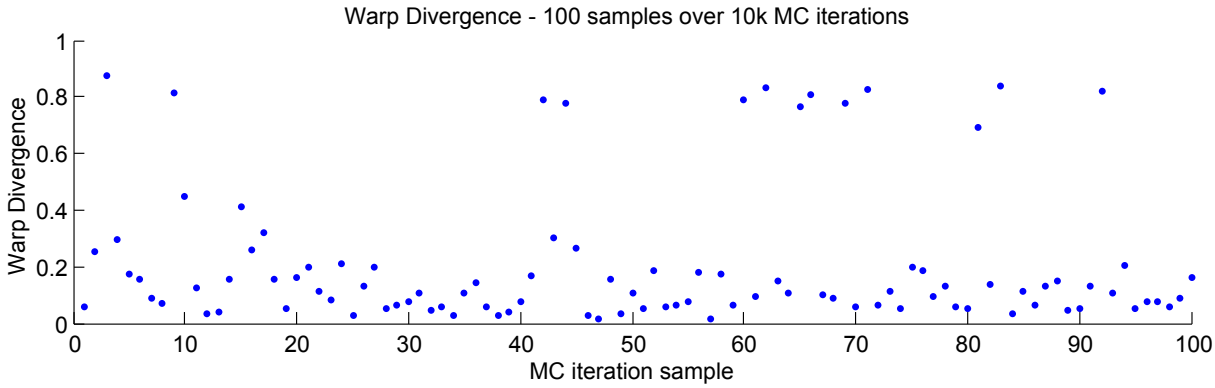

Figure 3.4: Warp divergence per MC iteration ($\frac{diverging\text{-}warps}{total\text{-}warps}$)

Separating distance computation from energy contribution computation in two different kernels would allow the OpenCL Host to predict the branching in the energy contribution, but would also mean the additional communication of two huge sets of distance values ($N_{GRID} \times natom_{chmol} \times sizeof(double)$) from the OpenCL device to the Host, worsening the overall execution time. An alternative would be for the host to employ some sort of calculation to predict what grid points will be inside or outside the cutoffs. In Figure 3.5, a simplified geometrical representation of this problem is presented, depicting the changed molecule in it's old and new state, the QM charge grid and the four branch regions according to the results of the distance computation for each $(atom, point)$ pair. For the sake of simplicity, in Figure 3.5, only one cutoff sphere is drawn per molecule when in reality there is one per atom. As devised in [30], the QM grid follows a logarithmic law with radial geometry: $r = -\alpha log_e(1 - x^m)$, centred on each atom of the QM molecule.

An idea worth to be explored would be to organize grid points in a few spherical clusters and compute the distance of the *chmol* atoms to each cluster before enqueuing the kernel. This way, points which are member of a cluster known to be entirely outside of the cutoffs (zone IV) could be discarded from the

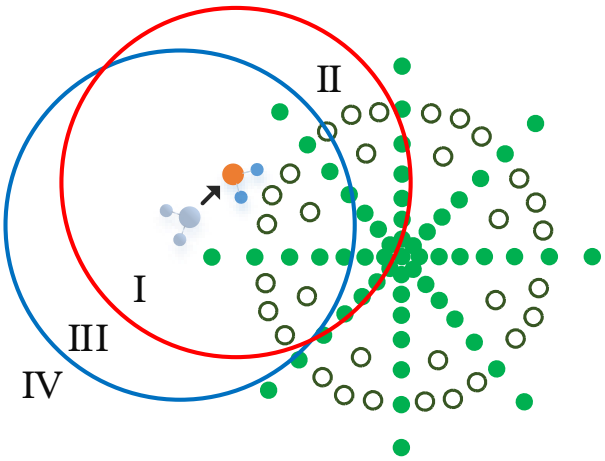

Figure 3.5: Cutoffs represented around the molecule before and after the MC step, along side with the QM point charge grid.

problem in that iteration and spare work-items. In order to preserve memory coalescence and simplify the attribution of work-item clusters to work-groups, it would be extremely helpful if the cluster size is a multiple of the preferred work-group size (e.g., multiple of 32, for an Nvidia Kepler Device). As discussed in Section 1.2, other authors have used neighbour-lists and cell-lists for slightly different problems. The challenge for the case of the PMC MM/QM is that for each QM grid configuration (an outer iteration in Figure 1.2) the PMC execution is already very fast ($\sim 8s$ for the OCL-GPU-Float version discussed in Section 3.4), meaning the overhead associated with creating and managing the lists must be low to compensate possible gains.

3.3.2 Paralellizing MM computations

MM distance computation involves calculating the Cartesian distance between the atoms of *chmol* and the atoms of all MM molecules, with the minor detail that distances are computed taking into account that the simulation box is periodic. This is easily parallelizable by tasking each work-item with computing the distances between one atom of the MM system and all the atoms of *chmol*. The resulting distances buffer size is proportional to the square of the number of atoms in the system ($natom^2$), which means transferring it between devices should be avoided. This buffer is required by the VDW and Coulomb MM energy computations (as suggested by the arrow dependency in Figure 1.1), so these computations should also be executed in the same device, if one wishes to avoid transferring a large buffer every MC iteration. VDW and Coulomb computation follow the same cutoff branching structure of Algorithm 1, and also correspond to an accumulation of energy terms (which may be mapped into a parallel reduction). The main difference is that the inner *for* cycle iterates over the MM atoms instead of the points of the charge grid (and for the case of the VDW energy computation the math is different). After executing these three operations, the resulting energy is just one floating point number, which can be lightly transferred around. In a system with a single multi-core CPU and a single GPU, the parallelization illustrated

in Figure 3.3 respects the above constraints. To take advantage of a system with two GPU devices for running both the MM and QM computations, the load balancing between the latter two has to be adjusted, by introducing a relaxation to the problem. This relaxation is discussed in Section 3.3.3. Also, while the MM system data (MM molecule coordinates and distances) may be kept in the GPU responsible for this computations, in a fraction of the accepted MC steps, the system coordinates and energy have to be transferred back to the CPU, to be written in the output file[1].

3.3.3 Problem Relaxations

As discussed with the research group in Institut für Physikalische Chemie, Georg-August-Universität Göttingen, the MC iteration structure can be relaxed in such way that k MM steps are computed in parallel with the QM step, as depicted in Figure 3.6. MM steps $N(0)$ to $N(k)$ use the QM energy computed in QM step N and are accepted/rejected accordingly. Following the heterogeneous approach in Figure 3.3, this relaxation can be used to allow more MM steps to be computed in the CPU while the GPU is still busy computing the QM step. Another way to take advantage of this relaxation is running several MM steps in another GPU device, in parallel.

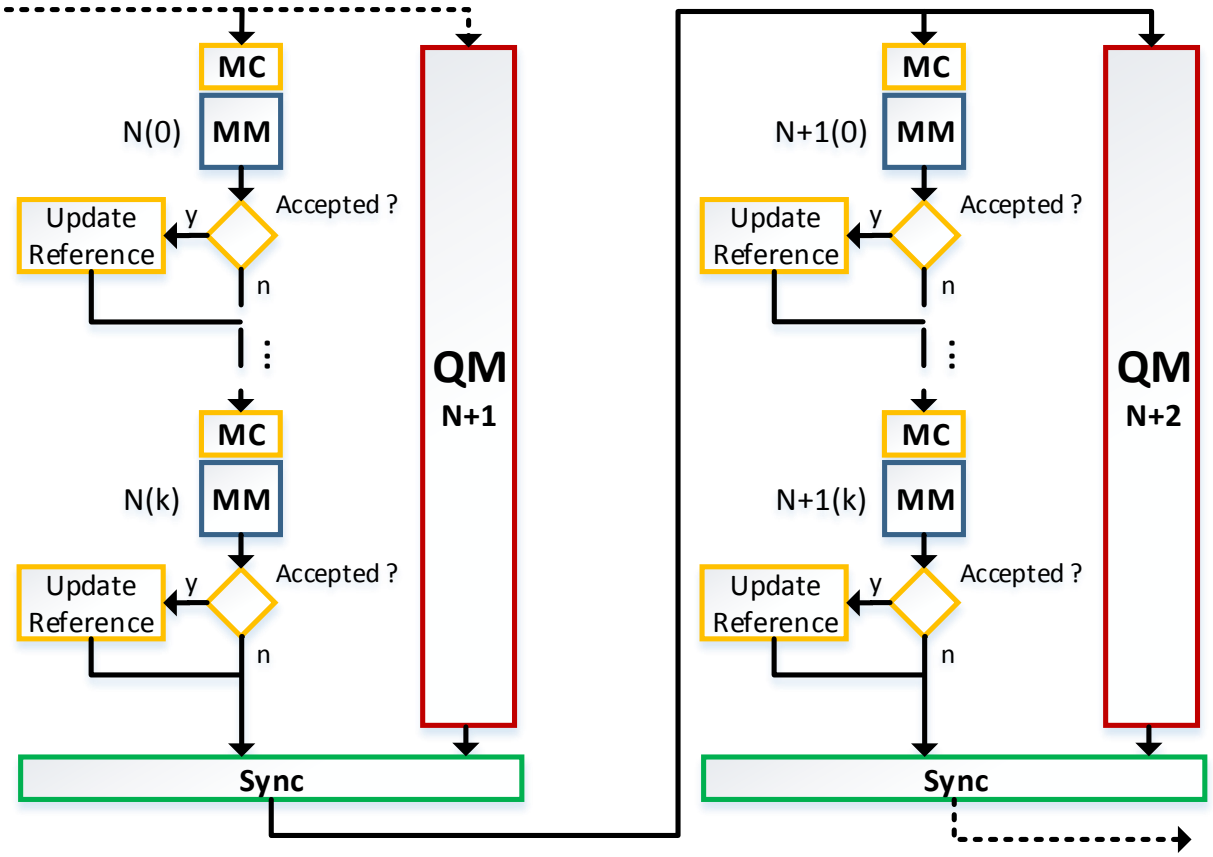

Figure 3.6: Problem relaxation.

[1] The fraction of the accepted steps which are written in the output file is a run parameter.

3.4 Intermediate Results

At the time of the writing of this report, a partial implementation of the system in Figure 3.3 has been carried out. The only thing that differs from that system is that in the CPU part only a single core is being used. Two OpenCL C kernels were written, one for the QM energy contribution (Algorithm 1) and first energy reduction and another for subsequent energy reductions. Results for a double and single precision version running in a system with an Intel i7 3820 CPU and an Nvidia GTX 660 Ti GPU are presented in tables 3.3 and 3.4, respectively. In the single-precision version, the floating-point precision has just been changed in *Coulomb QM I*, keeping the rest of the code in double precision. Due to time constraints, in this intermediate report a complete benchmark with different problem sizes was not made. Instead, the input system already discussed in previous sections (an MM solvent with 523 water molecules, each with 3 atoms and a QM solute, also a water molecule, represented by a grid of 268229 points) was used.

Code Section	time(us)	%(MC)
Distances (MM)	58.01	4.60%
VDW/Coulomb (MM)	49.66	3.94%
Coulomb I (QM)	**1020.05**	**80.96%**
Coulomb II (QM)	5.65	0.45%
Decision step	126.49	10.04%
Total	1259.86	

Table 3.3: Double-precision version with *Coulomb QM I* parallelized.

Code Section	time(us)	%(MC)
Distances (MM)	57.44	7.95%
VDW/Coulomb (MM)	49.50	6.85%
Coulomb I (QM)	**482.38**	**66.72%**
Coulomb II (QM)	5.55	0.78%
Decision step	128.3	17.70%
Total	722.90	

Table 3.4: Single-precision version with *Coulomb QM I* parallelized.

In Table 3.5, speedups between the two OpenCL versions presented in Tables 3.3 and 3.4 and the two serial versions discussed in Section 3.2, are displayed. The speedups obtained in the OpenCL versions result only from accelerating the *Coulomb QM I* portion, which is accelerated by a factor of ~ 39 when comparing the *OCL-GPU-Float* version to the *Serial-Optimized* version. According to Amdahl's law, the maximum achievable speedup by accelerating *Coulomb QM I* in *Serial-Optimized* is given by Eq. (3.4), where $\alpha(X)$ gives the fraction of the total execution time that X represents. Theoretically, from 39 to 50 there is still room for improvement, the most obvious reason being the warp/wavefront divergence discussed in Section 3.3.1.A. Limiting this maximum achievable speedup are several factors not accounted by Amdahl's law, such as kernel launch overhead, memory access time and communications between the Host CPU and the GPU.

$$S_{max} = \frac{1}{1 - \alpha(Coulomb\text{-}I\text{-}QM)} = \frac{1}{1 - 0.98} = 50 \tag{3.4}$$

In principle *Coulomb QM I* is already efficient in the serial implementation, and as mentioned in

Version	10k MC iters(s)	S(optimized)	S(original)
Serial-Original	231.98	0.86	1.00
Serial-Optimized	198.85	1.00	1.16
OCL-GPU-Double	13.83	14.38	16.77
OCL-GPU-Float	8.11	24.52	28.60

Table 3.5: Speedup of the developed program versions in respect to each other.

Section 3.2, there is still room for improving other sections of the serial code, which will have impact both on the serial and OpenCL versions. For the case of the OpenCL version it will increase the fraction *Coulomb QM I* takes of the total execution time (currently dropped to 66.72%, Table 3.4) and thus increase speedup.

3.5 Proposed Work

The proposed work for this MsC thesis is the performance evaluation of PMC MM/QM OpenCL implementations for several heterogeneous systems, optimizing OpenCL kernels and OpenCL Host code according to each system. The obtained Speedup in respect to the original serial version will be the major metric of interest. Following are the major considered systems:

(i) **1xCPU+1xGPU:** This is the system that was already targeted and the respective implementation discussed in Section 3.4, where QM computations are parallelized in the GPU and the rest of the algorithm is run in a single-core CPU.

(ii) **multi-core CPU+1xGPU:** Building upon the previously presented implementation, running MM computations using parallel CPU OpenCL kernels would complete the heterogeneous system depicted in Figure 3.3.

(iii) **1xCPU+2xGPUs (equal):** Two different approaches may be taken here. Either the QM computations are partitioned equally among each GPU, or one GPU device is tasked with several MM steps per MC iteration while the other is computing one QM step.

(iv) **1xCPU+2xGPUs (different):** In respect to the previous system, the main difference here is that the differences between each GPU architecture used should be exploited to take the most out of each one.

A system composed of a **CPU+GPU+FPGA** might also be targeted, which would be a great opportunity to evaluate the wide portability of the OpenCL framework. Also, slightly different combinations of the presented devices may be explored if it seems favourable. Furthermore, future work will include a detailed study of dynamic and static load balancing algorithms to tackle the load balancing problems that will arise when targeting the proposed heterogeneous systems.

4

Conclusions and Work Planning

In Chapter 2, an overview of current state-of-the-art hardware devices for heterogeneous computing was presented, as well as a brief introduction to the OpenCL framework. In Chapter 3, an analysis of the target algorithm was made, followed by a description of some possible acceleration approaches and results for a first implementation of one of the solutions. Accelerating the computation of the Coulomb I QM code portion by a factor of ~ 39 (version OCL-GPU-Float) led to an overall speedup of 28.6 in respect to the original serial version described in [30], proving the algorithm is in fact suitable for parallelization in a GPU device. This reduced the execution time of $10k$ PMC iterations from $\sim 230s$ to $\sim 8s$. As introduced in Chapter 1 (Figure 1.2), those $10k$ PMC iterations correspond to only one outer iteration of the complete algorithm. Details about the synchronization and load balancing between the PMC and the external program that updates the QM grid have yet to be discussed with the research group in Institut für Physikalische Chemie, Georg-August-Universität Göttingen, and as such the final impact of accelerating the PMC MM/ QM program in the complete algorithm cannot be analysed for the time being.

The Proposed Work for this MsC Thesis has been delineated in Section 3.5. For the major milestones, a time-line is depicted in Figure 4.1.

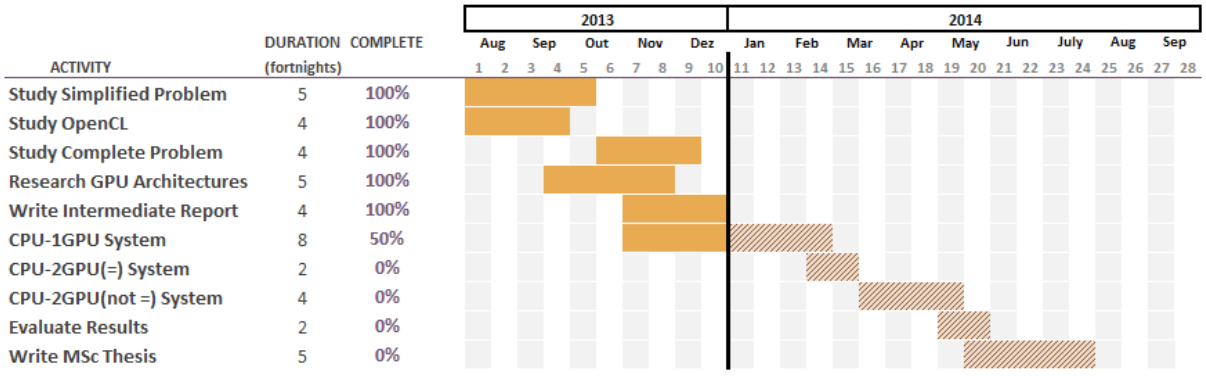

Figure 4.1: Thesis Planning.

Bibliography

[1] Alejandro Acosta, Robert Corujo, Vicente Blanco, and Francisco Almeida. Dynamic load balancing on heterogeneous multicore/multigpu systems. In High Performance Computing and Simulation (HPCS), 2010 International Conference on, pages 467–476. IEEE, 2010.

[2] AMD. OpenCL Optimization Case Study: Simple Reductions, 2010.

[3] AMD. Southern Islands Series Instruction Set Architecture Programming Guide, 2012.

[4] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide, 2013.

[5] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. Journal of Computational Physics, 227(10):5342–5359, 2008.

[6] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. ACM Sigplan Notices, 45(10):89–108, 2010.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.

[8] Alécio Pedro Delazari Binotto, Carlos Eduardo Pereira, Arjan Kuijper, Andre Stork, and Dieter W Fellner. An effective dynamic scheduling runtime and tuning system for heterogeneous multi and many-core desktop platforms. In High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on, pages 78–85. IEEE, 2011.

[9] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications, 14(3):189–204, 2000.

[10] Alfredo Palace Carvalho, José ANF Gomes, and M Natália DS Cordeiro. Parallel implementation of a monte carlo molecular simulation program. Journal of Chemical Information and Computer Sciences, 40(3):588–592, 2000.

[11] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. Dynamic load balancing on single-and multi-gpu systems. In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1–12. IEEE, 2010.

[12] Peter Eastman, Mark S Friedrichs, John D Chodera, Randall J Radmer, Christopher M Bruns, Joy P Ku, Kyle A Beauchamp, Thomas J Lane, Lee-Ping Wang, Diwakar Shukla, et al. Openmm 4: A reusable, extensible, hardware independent library for high performance molecular simulation. Journal of chemical theory and computation, 9(1):461–469, 2012.

[13] Mark S Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L Beberg, Daniel L Ensign, Christopher M Bruns, and Vijay S Pande. Accelerating molecular dynamic simulation on graphics processing units. Journal of computational chemistry, 30(6):864–872, 2009.

[14] George D Geromichalos. Importance of molecular computer modeling in anticancer drug development. Journal of BU ON.: official journal of the Balkan Union of Oncology, 12:S101, 2007.

[15] Khronos OpenCL Working Group. The OpenCL Specification version 1.2 revision 19, 2012.

[16] Mark Harris. General-Purpose Computation on Graphics Hardware, 2013.

[17] F. Jensen. Introduction to Computational Chemistry. Wiley, 2007.

[18] Jungwon Kim and Sangmin Seo. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. ICS '12 Proceedings of the 26th ACM international conference on Supercomputing, 2012.

[19] Victor W Lee. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. ISCA'10, June 19-23, 2010.

[20] Ricardo Mata. Electrostatics QMMM, presentation, 2013.

[21] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. The journal of chemical physics, 21:1087, 1953.

[22] NVIDIA. Kepler GK110, version 1.0, 2012.

[23] NVIDIA. CUDA C Programming Guide, version v5.5, 2013.

[24] Department of Computer Science and Engineering (DEI) Instituto Superior Técnico. Parallel and Distributed Computing slides, 2012.

[25] Benedict R.Gaster, Lee Howes, David R.Kaeli, Perhaad Mistry, and Dana Schaa. Heterogeneous Computing with OpenCL revised OpenCL 1.2 edition. Morgan Kaufmann, 2013.

[26] "Desh Singh, Tom Czajkowski, and Altera Corporation" Andrew Ling. Tutorial: Harnessing the Power of FPGAs using Altera's OpenCL Compiler, 2013.

[27] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for opencl devices. In Euro-Par 2010-Parallel Processing, pages 275–286. Springer, 2010.

[28] John E Stone, James C Phillips, Peter L Freddolino, David J Hardy, Leonardo G Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. Journal of computational chemistry, 28(16):2618–2640, 2007.

[29] Alfeus Sunarso, Tomohiro Tsuji, and Shigeomi Chono. Gpu-accelerated molecular dynamics simulation for study of liquid crystalline flows. Journal of Computational Physics, 229(15):5486–5497, 2010.

[30] Master's thesis submitted by Jonas Feldt. Entwicklung einer störungstheoretischen qm/mm monte carlo methode für die studie von molekülen in lösung, 2013.

[31] Zhenhua Yao, Jian-Sheng Wang, Gui-Rong Liu, and Min Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. Computer physics communications, 161(1):27–35, 2004.