Alma Mater Studiorum - Università di Bologna

Campus di Cesena

Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale
in
Ingegneria Biomedica

Parallelization of the Algorithm WHAM with NVIDIA CUDA

Tesi in

Bioingegneria Molecolare e Cellulare

Relatore:
Prof. Stefano Severi

Presentata da:
Nicolò Savioli

Correlatore:
Prof. Simone Furini

II Sessione
2012/2013

Contents

Introduction

Italian Language

Lo scopo di della mia tesi e' quello di parallelizzare Weighting Histogram Analysis Method (WHAM), che e' un popolare algoritmo usato per calcolare l'energia libera di un sistema molecolare in simulazioni di Molecular Dynamics. WHAM lavora in cooperazione con un altro algoritmo chiamato Umbrella Sampling.
Umbrella Sampling ha lo scopo di aggiungere un potenziale di energia (Biasing) al sistema al fine di forzarlo a campionare specifiche regioni nello spazio di configurazione. N diverse simulazioni indipendenti sono eseguite al fine di campionare tutte le regioni di interesse. In seguito, l'algoritmo WHAM e' usato per stimare l'originario sistema energetico partendo dalle N traiettorie atomiche. La parallelizzazione di WHAM e' stata fatta attraverso CUDA, un linguaggio che permette di lavorare nelle GPUs di schede grafiche NVIDIA, le quali presentano una architettura parallela. L' implementazione dovrebbe sensibilmente aumentare la velocita' dell' esecuzione di WHAM comparata con le sue precedenti implementazioni seriali in CPU. Tuttavia, il codice WHAM in CPU presenta delle criticita' temporali all' aumentare del numero di iterazioni.
L' algoritmo e' stato scritto in C++ ed eseguito in sistemi UNIX, purche' dotati di una scheda grafica NVIDIA. I risultati sono stati soddisfacenti, ottenendo un incremento di prestazioni quando il modello e' stato eseguito in schede grafiche con capacita' di elaborazione maggiore. Cio' nonostante, le GPUs usate, per testare l'algoritmo, sono molto vecchie e quindi non adatte al calcolo scientifico. E' probabile che si otterrebbe un ulteriore aumento di prestazione se l'algoritmo venisse eseguito su cluster di GPUs

ad alto livello di efficienza computazionale. La tesi e' organizzata nel seguente modo, In primo luogo descrivo la formulazione matematica dell' algoritmo Umbrella Sampling e WHAM con le loro applicazioni nello studio dei canali ionici e del Docking Molecolare (capitolo 1); poi, presentero' l'architettura CUDA utilizzata per implementare il modello (capitolo 2) e infine, saranno presentati i risultati ottenuti dal modello (capitolo 3).

English Language

The aim of my thesis is to parallelize the Weighting Histogram analysis Method (WHAM), which is a popular algorithm used to calculate the Free Energy of a molucular system in Molecular Dynamics simulations. WHAM works in post processing in cooperation with another algorithm called Umbrella Sampling.

Umbrella Sampling has the purpose to add a biasing in the potential energy of the system in order to force the system to sample a specific region in the configurational space. Several N independent simulations are performed in order to sample all the region of interest. Subsequently, the WHAM algorithm is used to estimate the original system energy starting from the N atomic trajectories. The parallelization of WHAM has been performed through CUDA, a language that allows to work in GPUs of NVIDIA graphic cards, which have a parallel achitecture. The parallel implementation may sensibly speed up the WHAM execution compared to previous serial CPU imlementations. However, the WHAM CPU code presents some temporal criticalities to very high numbers of interactions.

The algorithm has been written in C++ and executed in UNIX systems provided with NVIDIA graphic cards. The results were satisfying obtaining an increase of performances when the model was executed on graphics cards with compute capability greater. Nonetheless, the GPUs used to test the algorithm is quite old and not designated for scientific calculations. It is likely that a further performance increase will be obtained if the algorithm would be executed in clusters of GPU at high level of computational efficiency. The thesis is organized in the following way,

I will first describe the mathematical formulation of Umbrella Sampling and WHAM algorithm with their apllications in the study of ionic channels and in Molecular Docking (Chapter 1); then, I will present the CUDA architectures used to implement the model (Chapter 2); and finally, the results obtained on model systems will be presented (Chapter 3).

Chapter 1

Characterisation of Model

Biological membranes are very important structures and the most challenging targets in structural biology. In biological membrane some elements are of great importance: ion channels, protein molecules embedded in the lipid bilayer of the cell membrane. They play a key role in various physiological functions like nerve transmission, muscular contraction, and secretion. X-ray crystallography is the most important experimental technique to study ion channel structures; however, a small percentage of membrane has been obtained by nuclear magnetic resonance (NMR). A crystal structure is limited by the fact to be an average of many conformations of the crystal produced during the experiments, which represent the best fit of them, and so these three-dimensional models give only a partial representation of the dynamics of the system. Therefore, computational simulations are important to associate energy behaviour of the molecular system with its dynamic.

Figure 1.1 shows an interesting computational simulation of K^+ channel inserted in lipid bilayer. Unfortunately, even with large-scale computing resources and parallel codes it is still very difficult to simulate large biological time-scale. Firstly, the average time required for gating of a single ion is too long at time-scales of a classical algorithm of molecular dynamics. Secondly, the enormous quantity of atoms that constitutes a normal biological system impose limits to the length of simulations.

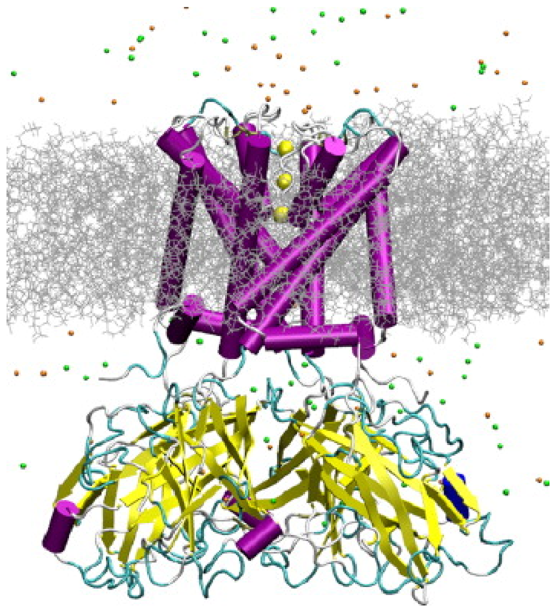

Figure 1-1: *The KirBac1.1 K+ channel structure (PDB 1P7B) embedded in a lipid bilayer with individual lipids rendered as gray chains. The transmembrane -helices are rendered in purple and the extracellular domain is colored according to its structure (-sheet: yellow; random coil: white; turns: cyan). K+ ions inside the selectivity filter of the channel are represented as yellow spheres. Other ions in solution are yellow and orange spheres "Jorgensen and Ravimohan, 1985 in Domene and Furini, 2009".*

1.1 Free-Energy Calculations

Computational simulations of atoms provide the description of the behaviour of biological systems at the atomic level. In molecular dynamics, interactions between the various atoms are described by empirical potential functions. Several characteristics of the trajectory in a system over time, structural and thermodynamic properties can be studied using the mathematical foundations of statistical mechanics, from which emerges the need to calculate the free energy (FE) of a system that represents a bond between mechanics and thermodynamics. The FE is the measure of the probability of finding an atomic system in a given state[1]. If the absolute energy of a system is

[1]It represents a crucial descriptor when the characteristics of a system can be related to the experiments or macroscopic properties of atomic system (Chandler, 1987; Hill, 1987).

scarsely calculated, the difference of free energy between two different states of the atomic system under consideration is easier to obtain. For example, in the field of *pharmacological research* it can be useful to use calculation of free energy to identify which molecule is the most powerful channel blocker. In this particular case the calculation of the FE is useful to understand how much affinity there is between drug molecule and the single channel to block. With the use of statistical thermodynamics, it is possible to calculate FE with MD. In fact, the difference FE, ΔA, between two configurations, 0 and 1, can be expressed as:

$$\Delta A = A_1 - A_0 = -K_B T \ln \frac{P_1}{P_0} \tag{1.1}$$

Where K_B is the Boltzmann's constants, T is the temperature, and P_O and P_1, respectively.

Configuration 0 is separated from its configuration 1 through a FE barrier. The stability of the system will be reached in the nearest space of configuration 0; while configuration 1 will never hit.

1.2 Thermodynamic Integration

In Thermodynamic Integration (TI) two states, linked by a path along the FE profile, are analyzed[2]. We consider the Hamiltonian as the sum of the kinetic and potential energy of a physical system or as the continuous function of parameter λ. As stated before, the states of interest of the physical system can be described with 0 and 1; thus, λ will be equal to 0 if the system is in configuration 0, while λ will be 1 if the system is in configuration 1. In particular, λ is modelled as a series of λ_i so that to for every λ_i point the system is completely relaxed. The FE difference between two states is expressed by the following relation:

[2]Constrained MD (den Otter and Briels, 1998) and adaptive biasing force MD (Darve and Pohorille, 2001) are variants of this technique as well as the popular steered MD method (Gulligsrund et al., 1999)

$$\Delta A^{TI}(0 \to 1) = \int_0^1 \left\langle \frac{\partial H}{\partial \lambda} \right\rangle_\lambda d\lambda \approx \sum_{i=1}^{n-1} \left\langle \frac{\partial H}{\partial \lambda} \right\rangle_{\lambda_i} \tag{1.2}$$

Where H is the Hamiltonian and $\langle\rangle_\lambda$ is the average value of λ. In fact, in TI calculation the calculation of averages are obtained by using molecular dynamics or Monte Carlo sampling[3]. Given that FE is a state function it does not depend on the path connecting the starting and final states, allowing the use of different, non physical paths as long as the beginning and the end remain constant. Many simulations of λ in the range 0-1 can be utilized to improve the numeric accuracy. To compare the values obtained by the evaluation of two FE series, two methods can be used: the forward sampling where the gradient is calculated as the increase of coefficient i that is moved by λ_i to λ_{i+1}, and the backward sampling where the estimation of coefficient i is obtained moving from λ_{i+1} to λ_i. Due to the presence of a small gap between λ_i and a quite long simulation, the difference of the integrals should be zero; practically, this will not happen in this case where the measure of the accuracy of calculation will be the difference, known as hysteresis error[4].

1.3 Free-Energy Perturbation

Another approach to calculate the FE is using *Zwanzig* equation.

$$\Delta A^{FEP}(0 \to 1) = A_1 - A_0 = -k_B T \ln \left\langle \exp(-\frac{H_1 - H_0}{k_B T}) \right\rangle_0 \tag{1.3}$$

Where T is the temperature, k_B is the Boltzmann's constant, H is the Hamiltonian. FE path is calculated as the sum of intermediate configurations; hence, we can write the Hamiltonian of intermediate state as:

[3]Due to fact that the FE is approximated by a sum over a discrete numbers errors can be present, even though with more λ_i points, errors are likely to decrease

[4]Jorgensen and Ravimohan, 1985 in Domene and Furini, 2009

$$H(\lambda_i) = \lambda_i H_1 + (1 - \lambda_i) H_0 \qquad 0 \leq \lambda_i \leq \lambda_{i+1} \leq 1 \tag{1.4}$$

This equation transform Eq.1.3 as follow:

$$\begin{aligned}\Delta A^{FEP}(0 \to 1) &= \sum_{i=0}^{n-1} \Delta A^{FEP}(\lambda_i \to \lambda_{i+1}) \\ &= \sum_{i=0}^{n-1} -k_B T \ln \left\langle \exp(-\frac{H(\lambda_{i+1}) - H(\lambda_i)}{k_B T}) \right\rangle_{\lambda_i} .\end{aligned} \tag{1.5}$$

Supposing that the sum of the series of FE is divided into intermediate states, it is possible to affirm that the difference between the Hamiltonian is small; therefore, we have to choose a better λ_i to obtain an overlap without doing unnecessary calculations. Since obtaining the ensemble for each value of λ_i is an independent process, the method can be trivially parallelized by running each window using different processors. One inconvenience of the technique is the fact that it is hard to know in advance what size of interval between the λ_i to choose. Pearlman's dynamically modified windows in some way mitigate the problem[5]. To solve the problem of representation of geometries of intermediate states that allow to transform a chemical substance into another, two approach have been analyzed: the single topology method and the dual topology method. In the first the change between the starting and final states of the system is represented by the alteration of atom types and internal coordinates; in the latter, two complete versions of the starting and final states can coexist. Due to the fact that more elaborated changes in topology can be better supported, the dual topology method is more advisable.

[5]Pearlman and Kollman, 1989 in Domene and Furini, 2009

1.4 Umbrella Sampling

Free energy calculations with the Thermodynamic integration or Free Energy Perturbation methods allows to calculate the difference in energy between two states but not the energy profile for the transformation from these two states. Umbrella Sampling is a powerful algorithm to calculate free energy profiles of chemical reactions. To understand the Umbrella Sampling (US) algorithm it is necessary to briefly introduce the canonical partition function Q.[6]

$$Q = \int \exp\left[-\beta E(r)\right] d^N r \tag{1.6}$$

Where $\beta = \frac{1}{k_B T}$, k_B is the Boltzmann's constant, T is the absolute temperature and N is the number of degrees of freedom of the system and E the potential energy. Therefore, we can obtain FE through the following relation: $A = \frac{-1}{\beta}\ln Q$, so that ΔA and ΔG, which is the gap of free energy, are numerically similar. The reaction coordinates describe the system state in the equation that allow us to derive $A(\xi)$. These ξ can be one or more dimensional, therefore, the probability distribution of the system along ξ can be evaluated by the following equation:

$$Q(\xi) = \frac{\int \delta[\xi(r) - \xi] \exp[(-\beta E) d^N r]}{\int \exp[(-\beta E) d^N r]} \tag{1.7}$$

$Q(\xi)d\xi$ can be seen as the probability of finding the system in a small interval $d\xi$ around ξ. As a result, this allows to calculate FE along the reaction coordinates as follow: $A(\xi) = \frac{-1}{\beta}\ln Q(\xi)$, which is also called Potential of Mean Force (PMF)[7]. In computer simulations, it is impossible to calculate the direct phase-space integrals present in Eq.(1.6) and (1.7); however, it is supposed that the system is ergodic. In fact, $P(\xi)$[8] is describes as in Eq. (1.8) if every point in phase space is touched during the simulation.

[6]If we consider the potential energy independent of the momentum, the integral in the phase space is a multiplicative constant to Q

[7]PMF was introduced for the first time by Kirkwood in 1935 and it is used for studying the behaviour of a molecular system.

[8]From now on, $P(\xi)$ will refer to the normalized frequency of finding the system in the state characterized by a given value of ξ

$$P(\xi) = \lim_{t \to +\infty} \frac{1}{t} \int_0^t \rho[\xi(t^*)] dt^* \tag{1.8}$$

Where $Q(\xi)$ is equal to the time average of $P(\xi)$ for infinite sampling, t is the time and ρ is the occurrence of ξ in a given interval (note that simulations are only run for finite time). After the end of the algorithm, the results of different simulations are integrated in a global FE profile $A(\xi)$.
Umbrella Sampling was developed by Torrie and Valleau in 1977. Basically, the region of interest are divided into R sets called windows, representing the points along of the collective variables referring to ξ. An additional term, called Bias, is included in the system to ensure the efficiency of the sampling along the coordinates of reaction. Hence, the bias potential ω_i of window i is another energy term depending only on the reaction coordinate:

$$E^b(r) = E^u(r) + \omega_i(\xi) \tag{1.9}$$

Where ‘b’ denotes the bias quantities and ‘u’ unbiased quantities.

To obtain unbiased FE $A_i(\xi)$, we have to consider the unbiased distribution as follow:

$$P_i^u(\xi) = \frac{\int \exp[-\beta E(r)] \delta[\xi^*(r) - \xi] d^N r}{\int \exp[-\beta E(r)] d^N r} \tag{1.10}$$

.

Modifying the previous equation by adding the biased potential, we obtain P_i^b.

Assuming an ergodic system:

$$P_i^b(\xi) = \frac{\int \exp\left(-\beta[E(r) + \omega_i(\xi^*(r))]\right) \delta[\xi^*(r) - \xi] d^N r}{\int \exp\left(-\beta[E(r) + \omega_i(\xi^*(r))]\right) d^N r} \tag{1.11}$$

.

Multiplying by β inside exp and gathering these outside, we obtain:

$$P_i^b(\xi) = \exp[-\beta \omega_i(\xi)] \times \frac{\int [-\beta E(r)] \delta[\xi^*(r) - \xi] d^N r}{\int \exp(-\beta[E(r) + \omega_i(\xi^*(r))]) d^N r} \tag{1.12}$$

.

Using Eq.(1.10) and Eq.(1.12):

$$P_i^u(\xi)\frac{\int \exp[-\beta E(r)]d^N r}{\int[-\beta E(r)]d^N r} = \delta[\xi^*(r) - \xi]d^N r \tag{1.13}$$

.

$$P_i^b(\xi) = \exp[-\beta\omega_i(\xi)] \times \frac{\int \exp\left[-\beta E(r)\right]\delta[\xi^*(r) - \xi]d^N r}{\int \exp[-\beta E(r)]\exp[-\beta\omega_i(\xi^*(r))]d^N r} \tag{1.14}$$

.

Substituting (1.13) in the equation (1.14), we have the following relationship:

$$P_i^u(\xi) = P_i^b(\xi)\exp[\beta\omega_i(\xi)]\left\langle \exp[-\beta\omega_i(\xi)]\right\rangle \tag{1.15}$$

.

P_i^b is obtained by the biased system simulations, $\omega_i(\xi)$ is obtained analytically and $F_i = -\frac{-1}{\beta}\ln\langle -\beta\omega_i(\xi)\rangle$ is independent of ξ and is given by an approximation of umbrella potentials $\omega_i(\xi)$:

$$A_i(\xi) = -\frac{1}{\beta}\ln P_i^b(\xi) - \omega_i(\xi) + F_i \tag{1.16}$$

.

If we want to calculate the FE of unbiased system, R simulations with different bias potential and the R different estimates of the unbiased probability $P_i^u(\xi)$ have to be combined; to obtain this, we use an algorithm called Weighted Histogram Analysis Method (WHAM), of which we present a parallelized version for NVIDIA CUDA Graphics Processing Unit (GPU).

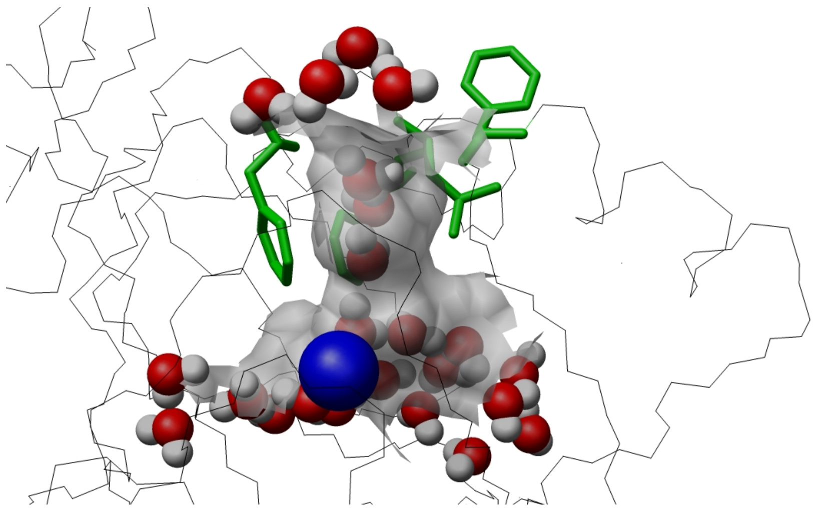

Figure 1-2: *Dioxygen channel (grey surface) constituted by four amino-acids (green). The copper centre and water molecules were respectively coloured in blue and grey/red. "Molecular simulation reveal a new entry site in Quercetin 2,3-Dioxygenase. A pathway for Dioxygen? S. Fiorucci;J. Golebiowski; D. Cabrol-Bass; S. Antonczak, Proteins, 2006, 64, 845-850. "*

1.5 Bias Potential

The bias potential is selected so that sampling along the entire range of reaction coordinate ξ is constant; for this reason the ideal bias potential is $\omega_{opt} = -A(\xi)$. This equation should give a uniform distribution $P_i^b(\xi)$. Nevertheless, $A(\xi)$ is surely unknown because the goal of Umbrella Sampling is to find it. Consequently, different families of potential are discovered, namely harmonic biases in a series of windows along ξ and an adaptive bias, which is realized to join $-A(\xi)$ in only one window covering the whole range of ξ. Often an harmonic bias of force constant k_i is expressed as follow:

$$\omega_i(\xi) = \frac{k_i}{2}(\xi - \xi_i^{ref})^2 \tag{1.17}$$

After each simulation, all FE curves are combined with the WHAM algorithm. We observe that the CPU time required to reach equilibrium increases with the number of simulations, but that in addition, the MD simulations are completely independent and can be executed in parallel. However, we must assume that k_i should be large enough to drive the system above barrier. Anyway, k_i too large can cause limited distribution $P_i^b(\xi)$ good overlap between the various distributions is required to perform for WHAM algorithm.

1.6 Weighted Histogram Analysis Method (WHAM)

After N_w simulations obtained with Umbrella Sampling through a potential $\omega_i(\xi)$ evaluating it in different ξ_i^{ref} and k_i, from each simulation, it is achieved an appropriate histogram $h_i(\xi_h)$ representing the probability of distribution P_i^b along the reaction coordinate ξ. Thus, in US simulations, bias is added to Hamiltonian of the atomic system in order to optimize the sampling along the coordinates of reactions. Therefore, the probability of finding the system in $h_i(\xi_h)$ of biased system is $P_i^b(\xi_h)$ where ξ_h is the simulation window. $P_i^b(\xi_h)$ is obtained by the following equation, derived forms Eq. (1.15):

$$P_i^b(\xi_h) = P^u(\xi_h)\mathrm{e}^{-\beta(W_i(\xi_h)-f_i)} \tag{1.18}$$

From Equation (1.16) we observe that the free energy of the system is obtain only by the knowledge of P_i^u; for this reason we should derive P_i^u from Eq. (1.18), given by N_w simulations; where $w(\beta_h)$ is the biasing potential, f_i is a constant, while $\beta = \frac{1}{k_B T}$, k_B is the Boltzmann's constant.

For obtaining a profile of the unbiased probability for the region of interest it is necessary to combine information of Umbrella Sampling from many different simulations. In the WHAM algorithm the unbiased probability is defined as a linear combination of N_w estimates to $P_i^u(\xi_h)$:

$$P_i^u(\xi_h) = \sum_{i=1}^{N_w} w_i(\xi_h) P_i^u(\xi_h) \tag{1.19}$$

Where $w_i(\xi_h)$ are the weights that allow to minimize the statistical uncertainty $\sigma^2[P^u(\xi_h)]$.

$$\begin{cases} P^u(\xi_h) = \sum_{i=1}^{N_w} \frac{\frac{\eta_i}{2\tau_i(\xi_h)}}{\frac{\eta_j}{2\tau_j(\xi_h)} \mathrm{e}^{-\beta(W_j(\xi_h)-f_j)}} \\ f_i = -\frac{1}{\beta} \ln \sum_h P^u(\xi_h) \mathrm{e}^{-\beta(W_i(\xi_h)} \end{cases} \tag{1.20}$$

Where n_i is the number of samples inside bin h_i and $\tau_i(\xi_h)$ is the integrated autocorrelation time.

Thereafter, in a easier way, we can express Eq. (1.15) as:

$$P_{ij} = \sum_j f_i c_{ij} P_j^* \tag{1.21}$$

Where P_{ij} is the unbiased probability and P_j^* biased probability, whilst f_i a constant and $c_{ij} = \exp[\beta\omega_i(\xi_h)]$. P_{ij} is chosen in such a way that $\sum_j P_{ij} = 1$ and then $f_i^{-1} = \sum_{j=1}^{M} c_{ij} P_j^*$. We can observe that an optimal estimation of P_j^* can be obtained from the following relationship:

$$P_j^* = \frac{\sum_{i=1}^{S} n_{ij}}{\sum_{i=1}^{S} N_i f_i c_{ij}} \tag{1.22}$$

Here n_{ij} is number of counts in the histogram $h_i(\xi_h)$ bin j for i simulations and N_i total number of samples generated in the previous simulations.
WHAM algorithm generates M + S non-linear equations that can be solved iteratively.

WHAM proceeds in this way:

1. Starts with an *arbitrary set of f_i* (for example $f_1 = f_2 = ... = f_S = 1$).

2. Use this f_i for *calculate* $P^u(\xi_h)$ with first equation of the system (1.20).

3. Use second equation of the system (1.20) *to calculate the new f_i*.

4. *Repeat the process* until convergence.

1.6.1 Traditional Derivation of the WHAM Equations

There are two ways to derive the following equations: the first, used by us, is the most simple and follows the work of *Ferrenberg and Swendsen and Kumar* based on a minimization of the estimated unbiased probability variance; the latter is more complex and follows the work of *Bartels and Karplus* and it uses the technique of a maximum likelihood.
By the equation (1.22) we know that the best estimate of unbiased probability, obtained from the j bin using the i simulations, is given by the following Equation:

$$P^u_{ij}(\xi_h) = \Omega_{ij} = \frac{n_{ij}}{N_i c_{ij} f_i} \tag{1.23}$$

We want to estimate the weights $w_i(\xi_h)$ of Eq. (1.19) that with S simulation and Ω_{ij} becomes:

$$P^*_j = P^u_{ij}(\xi_h) = \sum_{i=1}^{S} \omega_i \Omega_{ij} \tag{1.24}$$

What we hope to seek is the value w_i that minimizes the expected variance of P^*_j

$$\begin{aligned} var(p^*_j) = \langle (p^*_j - \langle p^*_j \rangle)^2 \rangle &= \langle (\sum_{i=1}^{S} \omega_i \Omega - ij - \langle \sum_{i=1}^{S} \omega_i \Omega_{ij} \rangle)^2 \rangle \\ &= \langle (\sum_{i=1}^{S} \omega_i (\Omega_{ij} - \langle \Omega_{ij} \rangle))^2 \rangle \end{aligned} \tag{1.25}$$

We take $\delta\Omega_{ij} = \Omega_{ij} - \langle \Omega_{ij} \rangle$ and then:

$$\begin{aligned} var(p^*_j) = \langle (\sum_{i=1}^{S} \omega_i \delta\omega_{ij})^2 \rangle &= \langle \sum_{i=1}^{S} \omega_i^2 (\delta\Omega_{ij})^2 + \sum_{k \neq l = 1}^{S} \omega_k \omega_l \delta\Omega_{kj} \delta\Omega_{lj} \rangle \\ &= \sum_{i=1}^{S} \omega_i^2 \langle (\delta\Omega_{ij})^2 \rangle + \sum_{k \neq l = 1}^{S} \omega_k \omega_l \langle \delta\Omega_{kj} \Omega_{lj} \rangle \end{aligned} \tag{1.26}$$

But we know that: $\langle (\delta\Omega_{ij})^2 \rangle = var(\Omega_{ij})$; if we assume that the different simulations k and l are uncorrelated we obtain: $\langle \delta\Omega_{kj}\delta\Omega_{lj}\rangle = 0$.

Therefore, the Eq.(1.26) becomes:

$$var(p_j^*) = \sum_{i=1}^{S} \omega_i^2 var(\Omega_{ij})) \tag{1.27}$$

We want to remind that: $\Omega_{ij} = \frac{n_{ij}}{N_i c_{ij} f_i}$ substituting this in $var(\Omega_{ij})$, we obtain the variance of n_{ij}; however, to obtain this, we must remember that the variance of the product of a constant $a = \frac{1}{N_i c_{ij} f_i}$ and $x = n_{ij}$ is given by:

$$var(ax) = \langle a^2x^2\rangle - \langle ax\rangle^2 = a^2\langle x^2\rangle - (a\langle x\rangle)^2 = a^2 var(x) \tag{1.28}$$

Then we get:

$$var(\Omega_{ij}) = \frac{var(n_{ij})}{N_i^2 c_{ij}^2 f_i^2} \tag{1.29}$$

If we replace this last Equation in Eq. (1.27) we obtain the following equation:

$$var(p_j^*) = \sum_{i=1}^{S} \frac{\omega_i^2 var(n_{ij})}{N_i^2 c_{ij}^2 f_i^2} \tag{1.30}$$

What is the $var(n_{ij})$? To answer this question, we pause to reflect: If we have N_i independent sampling, data from previous simulations, the probability of n_{ij} counts in a single bin of histogram is given by the Binomial distribution:

$$P(n) = \binom{N}{n} p^n (1-p)^{N-n} \tag{1.31}$$

If p is the probability of bin, then the mean and the variance of this Binomial distribution are respectively: np and $np(1-p)$. Of course, if we assume a large N and a small p, the Binomial distribution can be approximated by a Poisson distribution:

$$P(n) = \mathrm{e}^{-Np}\frac{(Np)^n}{n!} \tag{1.32}$$

As we know, the probability of bin j in the simulations i is:

$$p = f_i c_{ij} p_j^* \tag{1.33}$$

Knowing that the variance, in the limit of the Poisson approximation, is expressed as np, we have that the $var(n_{ij}) = (N_i) * (f_i c_{ij} p_j^*)$, and if inserted in the equation (1.30), we obtain:

$$var(p_j^*) = \sum_{i=1}^{S} \frac{w_i^2 p_j^*}{N_i c_{ij} f_i} \tag{1.34}$$

Now we need to minimize the function in respect of w_i under the condition of constraint $\sum_{j=1}^{S} w_i = 1$ using the Lagrange Multipliers [9].
Our function to minimize is:

$$Q = \sum_{i=1}^{S} \frac{w_i^2 p_j^*}{N_i c_{ij} f_i} + \lambda \sum_{i=1}^{S} w_i \tag{1.35}$$

Therefore, we derive this compared to w_i

$$\frac{\partial Q}{\partial w_k} = \frac{2 w_k p_j^*}{N_k c_{kj} f_k} + \lambda = 0 \tag{1.36}$$

Solving the equation we find that:

$$w_k = \frac{-N_k c_{kj}}{2 p_j^*}\lambda \tag{1.37}$$

Applying the boundary conditions we obtain:

[9]The Lagrange multipliers are the coefficients that are used to find the maximum/minimum of a function of several variables respect to a real-valued constraint. The constraint $g(x, y) = 0$ is a three-dimensional surface, which is detected in implicit form (ie as a place of zeros),the function $f(x, y)$ is a surface. The method involves introducing a new variable called the Lagrange multipliers, which we call λ and determined in accordance with the stationary points of a new function, the function of Lagrange is: $L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$

$$\sum_{k=1}^{S} w_i = \frac{-\lambda}{2p_j^*} \sum_{k=1}^{S} N_k c_{kj} = 1 \tag{1.38}$$

Thus, we can derive λ from equation (1.38):

$$\lambda = \frac{-2p_j^*}{\sum_{i=1}^{S} N_i c_{ij}} \tag{1.39}$$

In the end, we can get the optimal weights w_i

$$w_i = \frac{N_i c_{ij}}{\sum_{k=0}^{S} N_k c_{ik}} \tag{1.40}$$

If we substitute Eq. (1.40) in Eq. (1.24) we obtain Eq. (1.22).

1.7 Applications Of Free-Energy Methods

The *design of new drugs* involves the constant discovery of molecules that can effectively block proteins, in particular: receptors or enzymes that involve important physiological processes. Many ideas, applied to study the complex interactions such as linked friendships in social networks like *Facebook*, can be used to understand the metabolic networks and describe in detail the free energy required to start the biochemical reactions that involve the connection of a single drug molecule to the enzyme or cellular receptor.

In recent years new methods have been used to study the surface free energy for the protein folding; describing how a globular protein assumes its well-defined three-dimensional structure. Another important topic of study concerns the ionic channels that are integral membrane proteins having the role to control the passive diffusion along their electrochemical gradient. On the basis of these ionic species, the channel are classified into three categories Na^+, K^+ and Ca^{2+}. One of the first computational problems to be solved is the atomic structure of a K^+ channel. Its structure has inspired many computational studies that have found the conduction mechanism and selectivity of it. In recent years, the structure of Na^+ channel was found and through studies related to the free energy simulations it was possible to provide a possible conduction mechanism of Na^+ channel. Unlike what was observed for K^+ channels, ionic movements through the Na^+ channels appear highly uncorrelated.

1.7.1 Molecular Docking with Clustering and Cut-based FEP

Many techniques for the analysis of protein folding are based on a process of simulation of a drug molecule by performing a clustering according to the mean square distance between two snapshots of MD.

$$DRMS = [n^{-1} \sum_{ij}^{n} (d_{ij}^{a} - d_{ij}^{b})^2]^{1/2} \tag{1.41}$$

Where d_{ij} is the intermolecular distance between the hydrogen atoms of the drug molecule and the active site of the protein to block. The graph is formed by the nodes representing clustering while the transitions are its edges.

Krivov and Karplus have found an analogy between the kinetics of a complex process and equilibrium through a network graph developing an algorithm called free energy cut-based FEP[10]. The input of this algorithm is the transition network that is derived from the clustering of many simulations of the drug molecule on the active site of the protein to block[11]. For each node i in the transition network the partition function is described in Eq (1.42) and it is equal to the number of times that the node i is visited.

$$Z_i = \sum_{j} c_{ij} \tag{1.42}$$

Where c_{ij} is instead the number of direct transition from node i to node j. The individual transition probabilities can be calculated in this way:

[10]Krivov SV, Karplus M (2006) one-dimensional free-energy profiles of complex systems: Progress variables that preserve the barriers. J Phys Chem B 110: 1268912698.

[11]The Free Energy Landscape of Small Molecule UnbindingDanzhi Huang,Amedeo Caflisch.

$$p_{ij} = \frac{c_{ij}}{\sum_k c_{ik}} \tag{1.43}$$

For example, if the nodes of transition are partitioned into two groups: A and B in which A contains the reference node $Z_A = \sum_{i \in A} Z_i$ while B is $Z_B = \sum_{i \in B} Z_i$. The number of transitions from node A to node B is expressed by:

$$z_{AB} = \sum_{i \in A, j \in B} c_{ij} \tag{1.44}$$

The free energy of the transition is given by:

$$\Delta G = -kT \ln(\frac{Z_{AB}}{Z}) \tag{1.45}$$

Where Z is the partition function of the entire network.

Basically, the algorithm consists in three steps:

1. Nodes are sorted according to increasing values of mean first passage time (mfpt) which corresponds with the solution of the system of equations: $mfpt_i = \Delta t + \sum p_{ji} mfpt_j$ with boundary condition $mfpt = reference - node = 0$; $mfpt$ of a one node is defined as the sum of the time-step Δt equal to the saving frequency of 4 ps, and mfpt to adjacent node [12];

2. For each value of the progress variable are calculated relative partition function Z_A and Z_AB;

3. It is mapped a point ($x = \frac{Z_A}{Z}$,$y = -kT \ln(\frac{Z_{AB}}{Z})$) in the graph of energy profile.

[12]Apaydin M, Brutlag D, Guesttin C, Hsu D, Latombe J (2002) Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion. In International Conference on Computational Molecular Biology (RECOMB).

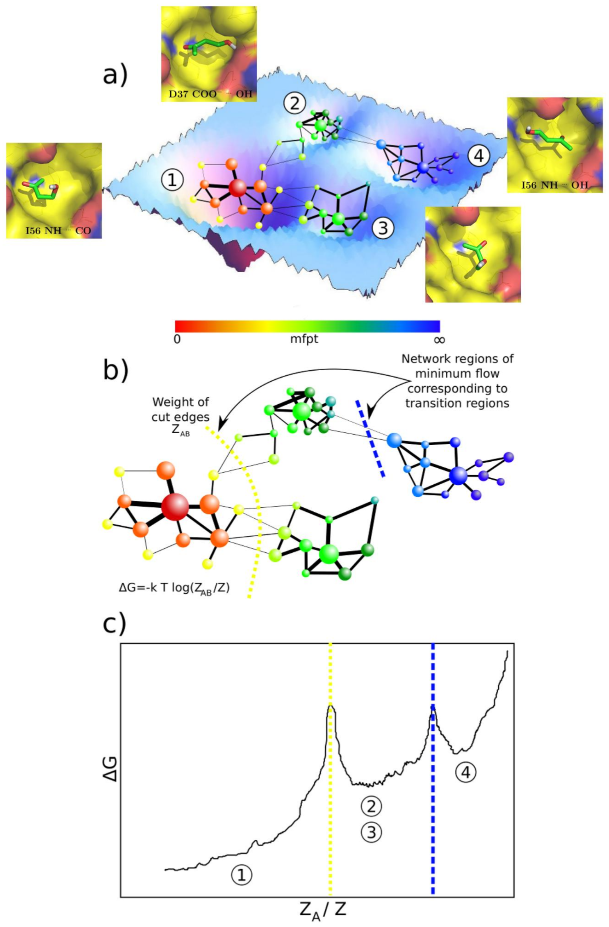
a)
D37 COO ··· OH
I56 NH ··· CO
I56 NH ··· OH
①
②
③
④
0
mfpt
∞
b)
Network regions of minimum flow corresponding to transition regions
Weight of cut edges Z_{AB}
$\Delta G = -k\,T\,\log(Z_{AB}/Z)$
c)
ΔG
①
②
③
④
Z_A / Z

Figure 1-3: *(a) The high-dimensional free-energy surface is coarse-grained into nodes of the network. Two nodes are linked if the system proceeds from one to the other along the considered timeseries. The mean first passage time (mfpt) is calculated for each node analytically (see text). (b) For each value of mfpt the set A of all nodes with a lower mfpt value is defined. The free-energy of the barrier between the two states formed by the nodes in A and the remainder of the network B can be calculated by the number of transitions between nodes of either set.(c) The cut-based FEP is a projection of the free-energy surface onto the relative partition function, which includes all pathways to the reference node. For each value of mfpt, the point is added to the FEP. The cut-based FEP projects the free-energy surface faithfully for all nodes to the left of the first barrier. After the first barrier, two or more basins overlap if they have the same kinetic distance from the reference node. "The Free Energy Landscape of Small Molecule Unbinding. Danzhi Huang,Amedeo Caflisch".*

1.7.2 Free Energy Profile in Misfolding Diseases

One of the most complex bioinformatics problems, still unresolved, is the kinetics of formation of the first ordered aggregates Amyloid fibrils which are the cause of many serious neurodegenerative diseases like *Alzheimer's*, *Parkinson's*, *Creutzfeld-Jacob's*. These fibrils are aggregates of polypeptide with a basic structure in β-sheets, in which the wires are perpendicular to the fibril axis. The initial phase of the formation of this fibril is of particular interest since it indicates the formation of precursors to soluble oligomers. Due to the transient nature of oligomeric precursors is difficult to understand the nature of their formation. Recent discoveries of amyloid functional in mammalian cells [13] lead to doubt that amyloid is always cytotoxic it is very likely that the kinetics of aggregation of pathogenic slower compared to healthy ones. Computational models of molecular dynamics have been used to understand the association irreversible of the polypeptide chain on the fibril but are not coherent with the experimental tests [14]. However, only through simulations with models that use a simplified representation of the geometry of the proteins has provided fundamental principles of protein aggregation. Moreover, it was also analysed by an energetic point of view. The computational model was obtained with an approximation based on a polypeptide with 125 monomers in a cubic box. Analysis of the system, from the point of view of the free energy profile, gives us very important information; inasmuch different aggregation scenarios are observed varying the energy parameter $dE = E_\pi - E_\beta$. The monomer has an internal flexibility with a profile of free energy with two minima: the first at the state amyloidcompetent-β and the second at the state amyloid protected-π, analyzed β-unstable ($dE \leq 2kcal/mol$) and β-stable ($dE \geq 0kcal/mol$) models. Infact, non-fibrillar aggregates are observed only in the unstable amyloid competent state with $(dE) \leq -2kcal/mol$. This confirms that: $\beta - unstable$ is more slower than $\beta - stable$ models [15].

[13] Functional amyloid formation within mammalian tissue. PLoS Biol. 4, e6. Fowler, D. M., Koulov, A. V., Alory-Jost, C., Marks,M. S., Balch, W. E. Kelly, J. W. (2006).

[14] O'Nuallain, B., Shivaprasad, S., Kheterpal, I. Wetzel,R. (2005). Thermodynamics of A(140) amyloid fibril elongation. Biochemistry, 44, 127091271

[15] Interpreting the Aggregation Kinetics of Amyloid Peptides Riccardo Pellarin,Amedeo Caflisch.

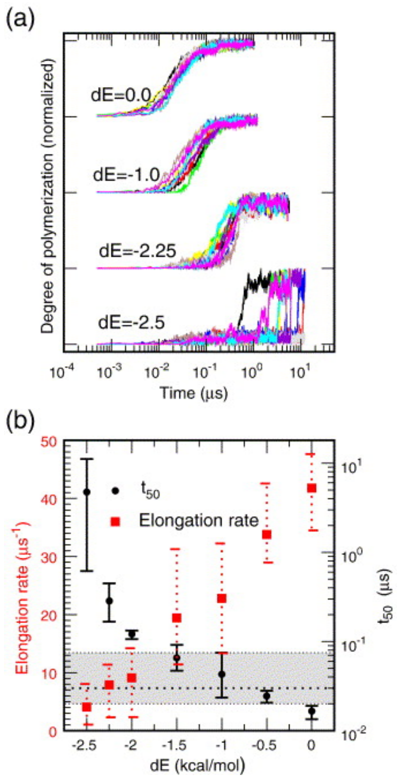

Figure 1-4: *Effect of relative stability of the amyloid-competent state on nucleation and elongation kinetics. (a) Time series of the fraction of ordered aggregation evaluated at four values of the protected state stability dE. Ten independent simulations are shown for each dE value. The degree of polymerization is normalized to the maximum for each curve. Note that the average value at the plateau is about 10% smaller for the dE = 2.5 kcal/mol than the dE = 0.0 kcal/mol model. It is not possible to directly compare the slopes of the curves (i.e. elongation rates) at different values of dE because of the logarithmic scale of the x-axis. (b) Influence of dE on the kinetics of the system. The time needed to reach 50% of the maximal amplitude t50 (black circles and y-axis legend on the right) and the elongation rate (red squares and y-axis legend on the left) are displayed for seven dE values. Symbols represent the average value of ten independent runs and the error bars are the maximum and minimum values. The broken line and the gray band indicate the average and the maxmin values for the time of micelle formation, respectively. All simulations were performed at a temperature of 310 K and a concentration of 8.5 mM."Interpreting the Aggregation Kinetics of Amyloid Peptides Riccardo Pellarin, Amedeo Caflisch".*

1.7.3 Conduction in Bacterial Na^+ Channel

From experimental observations, it is noted that Na^+ and K^+ share the same general structure. Therefore, we have a central pore where there is the passage of ions and delimited by two transmembrane helices S5, S6 connected together by an intervening loop. The data for the analysis are obtained from the crystallization of the channel Na_+ in the closed state. In the open state, the S6 helices bend and the cavity filled with water becomes a continuous with the intracellular solution [16]. The region that is responsible for selecting the ions in the incoming channel is localized in the loop between S5 and S6. We observe that the residues of Thr175 to Leu176 define the two rings of carbonyl oxygen atoms at the intracellular ingress of the filter. The molecules of water of hydration instead can interact through hydrogen bonds with these two layers of carbonyl oxygen atoms. Thus, being able to define two binding sites for Na^+ ions by means of the convention introduced by Payandeh[17] these sites of selectivity may relate with the following nomenclature: the first and the second S_{IN} and S_{CEN}, third as S_{HFS}. MD simulations of time scales of nanoseconds are not relevant for understanding the mechanism of conduction and selectivity in Na^+ channels. Therefore, in order to understand better the process of permeation, it was calculated the permeation free-energy profiles of Na^+ and K^+ channels with US[18]. By applying WHAM algorithm, it is observed that the energy profile of a single ion K^+ passing through a potassium channel it is different from the energy profile of a single ione through Na^+ channel.
This energy profile is constituted by two minima:

1. In the first we observe that minimum of the Na^+ is equal to K^+;
2. In contrast to Na+, no local minimum is present in the region between the carbonyl oxygen atoms of Leu176 and the side chain oxygen atoms of Glu177.

[16]Cuello LG, Romero JG, Cortes DM, Perozo E (1998) pH-Dependent gating in the Streptomyces lividans K+ channel. Biochem 37: 32293236.

[17]Payandeh J, Scheuer T, Zheng N, Catterall WA (2011)The crystal structure of a voltage-gated sodium channel. Nature 475: 353

[18]On Conduction in a Bacterial Sodium Channel Simone Furini, Carmen Domene Apr 05, 2012DOI: 10.1371/journal.pcbi.1002476

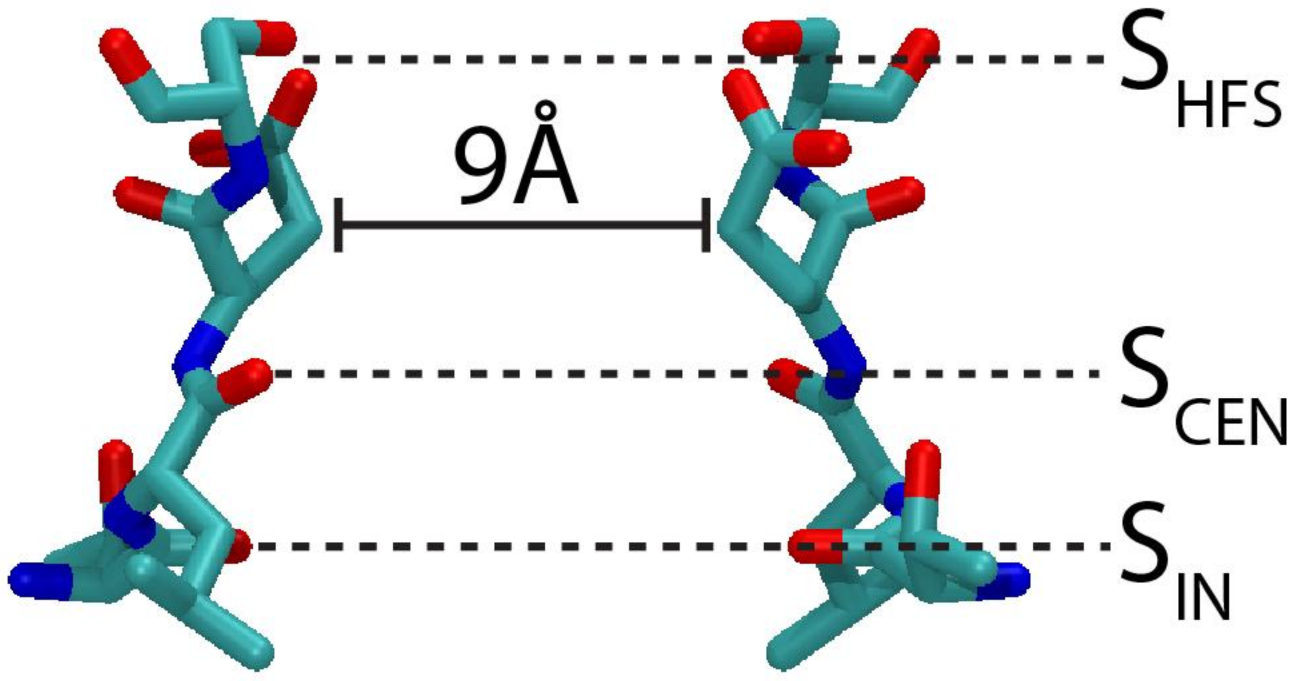

Figure 1-5: *Structure of the selectivity filter of NavAb. Residues 175 to 178 (TLES) of two opposite subunits are shown in licorice representation. SIN, SCEN and SHFS indicate possible ion binding sites. The distance between the C atoms of two Glu177 residues is indicated. doi:10.1371/journal.pcbi.1002476.g001.*

However, it is noted that position of an ion in a plane perpendicular to the axis of the pore is equal for both ions; furthermore it was also observed that a barrier higher than 8 kcal/mol prevents movement of the individual ions from S_{HFS} to extracellular solution.

Hence, it seems plausible that the selectivity filter is occupied on average by one or more positive ions and that conduction takes place during the movement to the ions already inside the pore[19]. To test this hypothesis were made maps of the free energy with two ions Na^+, we observe in the conditions of stability both S_{CEN} and S_{HFS} are occupied by ions Na^+.

[19]Berneche S, Roux B (2001) Energetics of ion conduction through the K+ channel. Nature 414: 7377. doi: 10.1038/35102067.

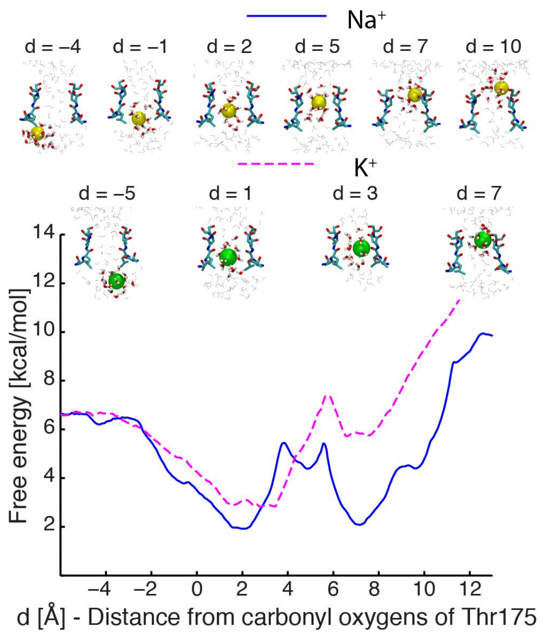

Figure 1-6: *Potential of Mean force for conduction events with one Na+ ion and one K+ ion. Values on the x-axis correspond to the distance between the permeating ion and the centre of the carbonyl oxygen atoms of residues Thr175 along the pore axis. The free-energy for a permeating Na+ ion is shown in a blue continuous line and K+ in a pink dashed line. Snapshots from the Umbrella Sampling trajectories are shown for some significant configurations. Residues 175 to 178 of two opposite subunits are depicted in licorice representation, together with the permeating ion in VDW spheres, yellow and green for Na+ and K+. Water molecules closer than 5 (licorice representation) and 15 (grey lines) from the ion are also shown. doi:10.1371/journal.pcbi.1002476.g002.*

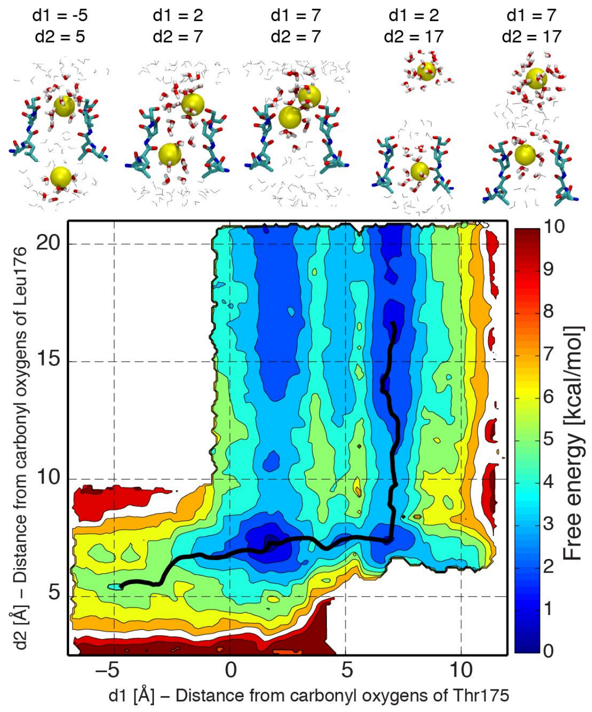

Figure 1-7: *Potential of Mean force for conduction events with two Na+ ions. Values on the x/y-axis correspond to distances along the pore axis between the permeating ions and the centre of the carbonyl oxygen atoms of residues Thr175 (d1) and Leu176 (d2) respectively. Counter lines are drawn every 1 kcal/mol. Snapshots from the Umbrella Sampling trajectories are shown for some significant configurations. Residues 175 to 178 of two opposite subunits are depicted in licorice representation, together with the permeating Na+ ion (yellow). Water molecules closer than 5 (licorice representation) and 15 (grey lines) to the ion are shown. The minimum energy path between a minimum with one ion in the cavity and a minimum with one ion in extracellular solution is shown as a black line. The free energy along the minimum energy path is shown in Figure S3.doi:10.1371/journal.pcbi.1002476.g003.*

Chapter 2

Description of CUDA Architecture

In recent years, new computational models have been developed in which new parallel architectures have allowed the improvement of computational abilities allowing numerical simulations to be more efficient and quicker. One of the strategies used to parallelize mathematical models is the use of GPGPUs (General Propose Computing on Graphics Processing Unit). These have been used in several sectors such as the image processing and also with efficiency in molecular dynamics. However, from 2007 on, it has been opened the possibility of programming GPUs with a specific language called CUDA (Compute Unified Device Architecture). This language allows to implement algorithms with a high-level programming language like C, which gives the possibility to control the whole architecture in order to exploit all the multi-core processors of GPU. It has to be said that in last years the computational capability of these architectures is increasing exponentially in comparison with normal processors called CPU (Central Processing Unit). Despite to the fact that the number of cores is increasing, the majority of the normal CPUs exploits part of their transistors for the logic control, while GPU cores, which are more in number, are simpler in architectures and are optimized for numerical calculations.

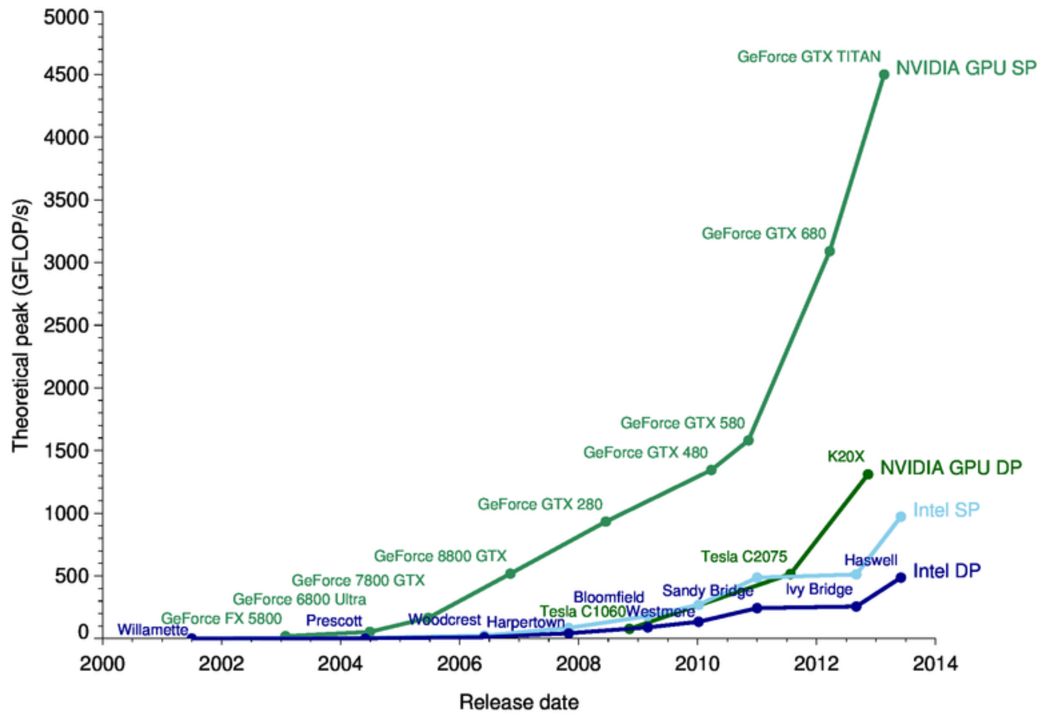

Figure 2-1: *GPU vs CPU performance from"web2.infn.it".*

2.1 GPU Architecture

The *GPU* is usually connected to a host through a *PCI-Express*. In addition, GPU has a memory containing several gigabytes and individual data are transferred between GPU and memory of the host using a programmable DMA which can operate in the two directions. One important thing to keep in mind is that many GPUs do not depend on a hierarchy of cache memory and also supports a deep width very high memory bandwidth, using a wide data path. Each NVIDIA [1] GPUs is constituted by a number of multiprocessors and each of these can run in parallel with the others. In turn each multiprocessor consists of 8 or 16 *Stream Processors*; for example, a Tesla [2] architecture has a group of 8 *Stream Processors*, while a Fermi [3] architecture has two groups of 16 *Stream Processors*. Since, if Tesla presents 30 Multiprocessors and

[1]*NVIDIA* is a registered trademark of *NVIDIA Corporation* http://www.nvidia.com/
[2]http://www.nvidia.com/object/tesla-supercomputing-solutions.html
[3]http://www.nvidia.com/object/fermi-architecture.html

8 Stream Processors, the total number of cores is 240, instead Fermi has two groups of 16 Stream Processors for 16 Multiprocessors for a total number of cores of 512. Each of these multiprocessors can perform mathematical operations such as addition, multiplication, subtraction, etc in single precision (32-bit) or double precision (64 bit). The architecture of the GPUs NVIDIA offers also the possibility of a *Shared Memory* accessible directly from all SM. The model of NVIDIA GPUs is *SMID* (Single Instruction, Multiple Data) composed of only a control unit that executes one instruction at a time by controlling more ALU that works in a synchronous manner. For every single step all elements perform the same identical scalar instruction (eg. sum) but each on different data; NVIDIA calls this SIMT (Single Instruction Multiple Thread) because in his model CUDA the same instructions are actually carried out from different Threads.
One basic thing to understand is that the decoding takes place every 4 clock cycles of the multiprocessor, but each of them can launch an instruction for every clock cycle. This means that each instruction decoded match 32 executions of the same clock cycles, as a matter of fact Tesla graphics card possesses 8 stream processors, and then 32 executions of the same instruction[4].

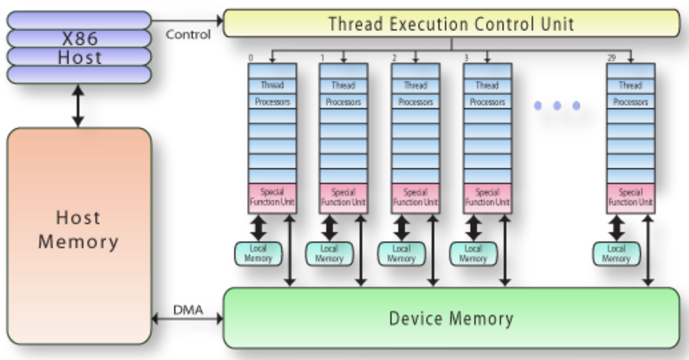

Figure 2-2: *NVIDIA Tesla Block Diagram from "http://www.pgroup.com" Copyright 2013 NVIDIA Corporation.*

[4]NVIDIAs Fermi: The First Complete GPU Computing Architecture A white paper by Peter N. Glaskowsky Prepared under contract with NVIDIA Corporation

2.2 CUDA Parallel Programming

In the CUDA programming model there are two parts:

1. parts of code executed in series by the host system;
2. parts of the code, called kernels, which run in parallel.

A kernel can be seen as a *grid* which is composed of *blocks*, assigned sequentially to the various CUDA multiprocessors. Inside each block there is the basic unit called *thread*. A Thread belongs to one block and for this reason is identified by a unique index. Each thread can even be assigned by three-dimensional index; for blocks we can only have two-dimensional indices.

Listing 2.1: *http://docs.nvidia.com/cuda/cuda-c-programming-guide*

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

As in code example 2.1 a single kernel is defined using the specifier __ *global* __ and the number of blocks and threads are specified using the specifier operator

⋘ ... ⋙, syntax can be of type *int* or *dim3*. As previously said, each thread that executes the kernel is defined by a unique thread ID that be accessed via the built-in variable *threadIdx*. Instead, to access the ID of a block is used *blockIdx* variable (values from 0 to gridDim-1), whose size can be obtained through its *blockDim* variable. Finally, the variable *gridDim* fives the grid size.

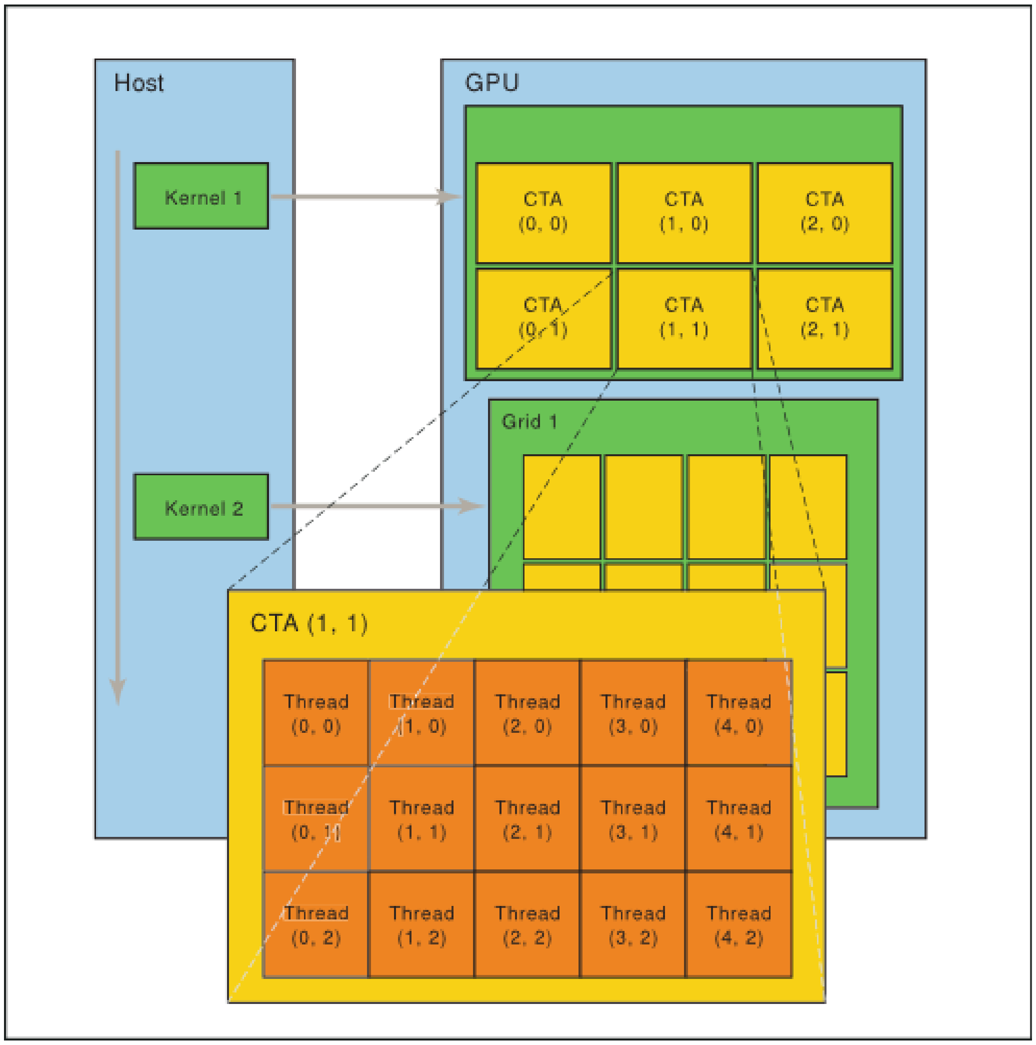

Figure 2-3: *Grid of Thread Blocks in Running from "http://docs.nvidia.com".*

For completeness, we extend the example 2.1 to deal *multiple blocks:*

Listing 2.2: *http://docs.nvidia.com*

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A very common choice is to take the block size equal to 16x16 (256 threads), which in this case is given in an arbitrary manner. In this example, the grid is then created to have a thread for each element of the array. However, we use the simplification that the number of threads per grid in each dimension is divisible by the number of threads of the block in that dimension[5]. Where each thread, that is the element of the matrix, is represented by the following relation *blockIdx.x * blockDim.x + threadIdx.x* for *i* and *blockIdx.y * blockDim.y + threadIdx.y* for *j* as represented in *Figure 2.4.*

[5]Example can be found at the following link: *http://docs.nvidia.com/cuda/cuda-c-programming-guide.*

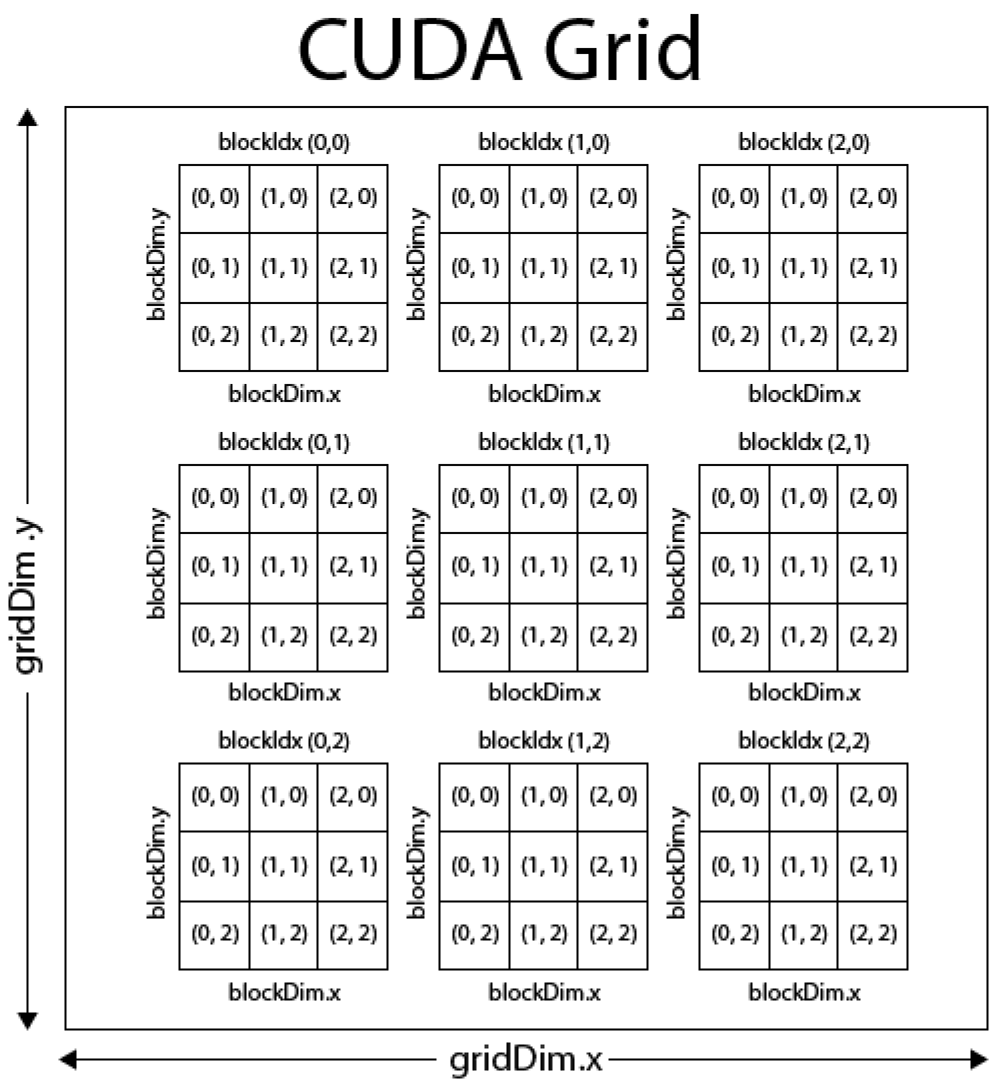

Figure 2-4: *Grid 2D of Thread Blocks from "http://3dgep.com".*

2.3 GPU Memory

In CUDA platform, knowledge of the memory and its use is of crucial importance to obtain optimum efficiency in the numerical simulator that we are going to implement. In GPUs we can distinguish six types of memory:

1. *Global Memory*

2. *Constant Memory*

3. *Texture Memory*
4. *Registers*
5. *Local Memory*
6. *Shared Memory*

2.3.1 Global Memory

The Global Memory, which resides in device DRAM, is the main available in GPUs. Depending on the model of the device, this memory ranges from 256MB to 1024MB. Its capacity is in contrast with its speed. Infact, it is the slowest available to the programmer, as it uses 400-600 cycles to access data[6]. There are two ways to declare this memory: through a declaration by the global scope, using declaration specifier *__device__* or dynamically allocated with *cudaMalloc()* as in the code 2.3.
In the previous sections we discussed how individual threads are clustered into blocks, which are subsequently assigned to multiprocessors. This means that during the running there is a grouping of treads for warp (block of threads divided into sub-blocks is called "warps"). Therefore, each multiprocessor in the GPU executes instructions per warp in SIMD fashion. The size of each warp of any GPUs is 32 threads, thus, device tries to coalesces Global Memory (*Global Memory Coalescing*) loads and stores then issued by threads of a warp into minimal transactions in order to minimize DRAM bandwidth[7].

Listing 2.3: *http://devblogs.nvidia.com*

```


// Using declaration specifier:
__device__ int globalArray[256];

```

[6]presentations NVIDIA GPU Computing Webinars Best Practises for OpenCL Programming,2011

[7]http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

```
void foo()
{
    ...

    int *myDeviceMemory = 0;
    //Using dynamically allocated:
    cudaError_t result = cudaMalloc(&myDeviceMemory, 256 * sizeof(int));

    ...
}
```

To get started with the Global Memory, it is needed allocate global space memory in the GPU to achieve this we use the following instructions:

```
int N = ...;
float * dev_A ;
cudaMalloc(&dev_A, N*sizeof(float)) ;
```

After allocating the memory the data need to be copied from GPU to CPU. This process can be done with the following instructions:

```
int N=...;
float  host_A = ( float* )malloc (N*sizeof(float)) ;
cudaMemcpy ( dev_A , host_A , N*sizeof(float), cudaMemcpyHostToDevice);
```

Finally, when our processing is complete, the data can be downloaded once again from Global Memory of our device to the CPU through the following statement.

```
cudaMemcpy(host_A,dev_A,N*sizeof(float), cudaMemcpyDeviceToHost) ;
```

2.3.2 Constant Memory

The Constant Memory is a memory used to load data in read-only, so it does not change during the execution of the kernel. The size of this memory does not exceed 64 KB of total space thus, every multiprocessor, has available 8 KB.
In some situations it may be useful to use this memory type instead of the Global Memory by reducing the bandwidth of memory required. This memory can only be written by the host and not from the device because it would lose the connotation of constant; this process is done through the function *cudaMemcpyToSymbol()*.

```
__constant__ float const [256] ;
float dataA [256] ;
cudaMemcpyToSymbol(const,dataA,sizeof(data));
cudaMemcpyFromSymbol(dataA,const,sizeof(data));
```

The method CUDA *MemcpyToSymbol()* allows to load data into Constant Memory, while CUDA *MemcpyFromSymbol()*, download them to the CPU.

```
__constant__ float const [256] ;
float dataA [256] ;
cudaMemcpyToSymbol(const,dataA,sizeof(data));
cudaMemcpyFromSymbol(dataA,const,sizeof(data));
```

2.3.3 Texture Memory

As for the Constant Memory, Texture Memory is cached in the chip. However, this type of memory is used for very frequent operations in computer graphics, such as: mapping, deformation, or implanting a texture 2D of polygonal model. The texture cache is then shared by all processors and provides a speed up in reading texture memory, implemented as a read-only memory region of the device[8]. CUDA devices have addition units called Texture Mapping Unit (TMU) that are not members of the

[8]Exploiting Graphical Processing Units for Data-Parallel Scientic Applications, A. Leist, D. P. Playne and K. A. Hawick Computer Science,Institute of Information and Mathematical Sciences, Massey University, Albany, Auckland, New Zealand,December 2008

physical processors and are able to rotate or resize a bitmap placed on an arbitrary plane of any given object as a 3D texture. The texture has an internal memory (on-chip) using a buffer that is capable of taking data from global memory. A variable of type texture is visible from both the kernel and the host code, and it must be declared with global scope. As for the variables allocated in memory constant, it has a duration equal to the life cycle of CUDA application. In order to use variables that reside in Memory Texture is therefore necessary to declare a reference to the texture, in this manner:

```
texture<float, cudaTextureType2D,cudaReadModeElementType> texRef;
```

And then perform the data binding that already reside in global memory:

```
cudaChannelFormatDesc channel= cudaCreateChannelDesc<float>();
cudaBindTexture (NULL, &deviceA,texref,&channel,size)
```

The data that are all inside of the kernel can be read via *tex1Dfetch*, if we made the binding of a zone of linear memory, otherwise the primitive *tex1D*, *tex2D* and *tex3D*, when an array is multidimensional. At the end of the application is proper to make unbundling to free up resources through the following statement:

```
cudaUnbindTexture(texref);
```

2.3.4 Registers

They are at the disposal a type of on-chip memory that is very fast, its size is 32 bits and is visible from a single thread. The parameters of the local kernel are automatically copied in the records without any specific operation by the developer.

2.3.5 Local Memory

The Local Memory is called this way because it is "local" in the scope of each thread. It is not a hardware component of a multiprocessor [9] This type of memory physically resides in global memory that is allocated by the compiler, so it will have the same performance than any other global memory. The compiler generally can choose to allocate our variables in local memory when:

1. *many variables occupy too many registers;*
2. *Structures and Arrays, which take excessive large space in the registers;*
3. *the compiler is unable to determine if an array has a constant size.*

2.3.6 Shared Memory

This type of memory is a hundred times faster than global memory and has the distinction of being shared between threads in the same block[10] and it can be used to carry out communications and synchronizations between the same thread. The Shared Memory is divided into 16 KB of memory per Streaming Multiprocessor but bearing in mind that every SM has a private L1 cache, then it shares a fixed 64KB of memory space with shared memory, so the memory on-chip RAM is divided into 16 KB L1 and 64 KB of shared memory and later, L2 cache is global on all the SM with a size of 768 KB[11]. Life cycle of the variables stored in the shared memory are connected with the life of the kernel when it ends its execution they are released. The creations of these variables in the shared memory is done by prefixing the type *__shared__*.

[9]http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/215900921
[10]http://www.sdsc.edu/us/training/assets/docs/NVIDIA-04-OptimizingCUDA.pdf
[11]Encyclopedia of Parallel Computing, David Padua, Springer Science LLC 2011

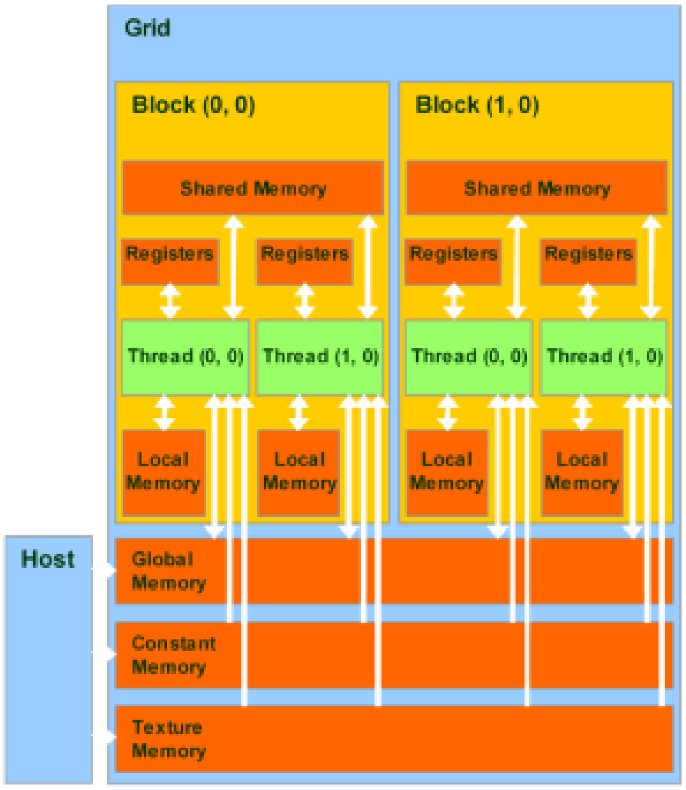

Figure 2-5: *CUDA Memory Model from "http://www.realworldtech.com".*

2.4 Threads and Blocks Managing

When designing kernels with high performance, we need to take into consideration not only a careful use of the internal memory, but also a suitable division of the various blocks in which our application works. If we take our graphics card with compute capability 1.1 we have the following hardware characteristics respect:

1. *maximum number threads to block = 512;*

2. *maximum number of blocks for SM = 8;*

3. *maximum number threads for SM = 768.*

From the information above, we have that each SM can assign up to $768/32 = 24$ warp, therefore, a very important rule to follow is that the number of threads per block must be divisible by 32, otherwise it may happen that we will have warp partially filled and then adds fictitious thread to reach 32. The number of warp should then be a divisor of 24 or we can risk not to achieve the maximum number of warp assignable to every single SM. A further important rule to follow is that each block must not be less than 24/8=3 warp threads 96 and therefore accordingly[12]. The total number of blocks and warps that can reside and can be processed together on the multiprocessor depends on the:

1. *amount of registers and Shared Memory used by multiprocessor;*
2. *amount of registers and Shared Memory used by the kernel.*

The formula to calculate the total number of warps W_{block} in a single block is as follows:

$$W_{block} = ceil(\frac{T}{W_{size}}, 1) \tag{2.1}$$

1. *T is the total number of threads per block;*
2. *W_{size} is the wrap size that is equal to 32;*
3. *$ceil(x, y)$ is equal to x rounded up to the nearby multiple of y.*

The total number of registers R_{block} allocated per block is provided by following equation:

For devices of compute capability 1.x

$$R_{block} = ceil(ceil(W_{block}, G_W) \times W_{size} \times R_k, G_T) \tag{2.2}$$

For devices of compute capability 2.x

$$R_{block} = ceil(R_k \times W_{size}, G_T) \times W_{block} \tag{2.3}$$

1. *G_W is warp allocation granularity(equal to 2);*

[12]Optimizing CUDA Applications, 3D Game Engine Programming, Execution Optimizations from http://3dgep.com/?p=2081

2. *R_k is the number of registers used by the kernel;*

3. *G_T is the thread allocation granularity(equal to 256 for devices of capability 1.0 and 1.1).*

The total amount of Shared Memory S_{block} in bytes for a single block allocation follows the following relationship:

$$S_{block} = ceil(S_k, G_S) \tag{2.4}$$

1. *S_k is the amount Shared Memory used by the kernel in bytes;*

2. *G_S is the shared memory allocation granularity (equal to 512 for device of compute capability 1.x).*

These formulas were obtained by "NVIDIA CUDA C Programming Guide Version 4.0".

2.5 Compilation Process

The compilation process is the initial step to run the code created for GPUs. CUDA files typically have a file extension *.cu* and can be compiled only with a specific compiler package provided by nvcc. In order to compile the innumerable CUDA files of the thesis project it was created a make file that can be compiled via shell command *make*.

2.6 Optimization

2.6.1 Coalescence

One of the fundamental processes to optimize the CUDA code is to gather more accesses, both in reading and in writing, in a single transition from the memory controller. More generally, the unit that we are referring is that of the half-warp, i.e. The memory accesses are not clustered in groups of 32 threads of the warp but in groups of 16 threads. If we are able to organize access in a half-warp we can get a performance boost through their coalescence. The guidelines to be followed vary according to the compute capabilities of the graphics card. In general, for those with compute capability 1.0 and 1.1 the accesses of a half warp are coalescing if:

1. *64 bytes - each thread reads a word: int,float, ...*
2. *128 byte - each thread reads a double-word: int2,float2,...*
3. *256 bytes - each thread reads a quad-word: int4,float4,...*

In addition, the following restrictions must be observed:

1. *The starting address of a region must be multiple of the size of the region;*
2. *The k-th thread in a half-warp must access the k-th element of a lock (read or written), that access must be perfectly aligned between thread.*

2.6.2 Synchronization

A limitation of the CUDA architecture is that the various threads of different blocks can not communicate between them and exchange messages. The threads belonging to a same block can do it through of a primitive call *__syncthreads()* that can be visualized as a point of stopping at a precise point which will start only when all threads of the same block have achieved.

Chapter 3

Code and Results

This chapter describes the details of the implemented code and the results obtained by running this in different architectures by comparing it with the same code implemented on the CPU.
All the code presented below is protected by *GPLv3*[1].

3.1 Code

The code consists of *11 files* invoked as external functions and of a main file in which is implemented variable initialization and the iterative algorithm. Each of these CUDA external function contributes to the estimate of the constant F_i. Then, at the end of the iterative cycle, results are recovered by means of a function, *clock*(), which also temporize the iterative algorithm.

[1]Copyright (C) 2013 Nicolò Savioli This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

3.1.1 Main

The main code runs the mathematical model using the same iterative steps described in *chapter 1 section 1.6*, with the aim of calculating the constant F_i. Each of the functions within the iterative algorithm (discussed below) run a single kernel in order to performe a parallel operation in the GPU. Initially, the allocated variables are used to load data in the global memory of the GPU[2], then through *malloc()* they are allocated in dynamics memory space[3]. The allocation of variables in the memory of the graphics chip is done through *cudaMalloc()*[4]. Thereafter, we need to fill variables allocated in the dynamic memory of the host in order to use them to allocate global memory of graphics card. These data are acquired from a class called histogram, whose builder is HIST, used to acquire the data of the model[5]. Finally, we arrive to the convergence loop, it consists of two conditions: the number of iterations and convergence condition[6] where *cudaMemcpy()*[7] is used to download the result in host memory for comparison with the value of tolerance.

Listing 3.1: Extract of Main.cc

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
 cudaDeviceReset();
 // Allocate var for GPU
   __constant__ float* dev_U;
  __constant__ float * dev_UU;
  __constant__ float * dev_Punnorm_result;
  float * dev_histmin;
  int * dev_numbinwin;
  float * dev_g;
  float * dev_F;
  float * dev_numwham;
```

[2]From line 10 to 34 of Main.cc
[3]From line 39 to 66 of Main.cc
[4]From line 69 to 95 of the Main.cc
[5]From line 97 to 123 of Main.cc
[6]From line 151 of Main.cc
[7]From line 172 of Main.cc

```
float * dev_center;
float * dev_harmrest;
float * dev_delta;
int * dev_step;
int * dev_numbin;
float * dev_sum_result;
float * dev_denwham;
float * dev_result_Punnorm;
float * dev_sum_normfactor_for_normprob_and_normcoef;
float * dev_P;
float * dev_A_result;
float * dev_P_old;
float * dev_rmsd_result;
float * dev_F_toBeNormalized;
float * dev_kT;
float * dev_sumP;
float * dev_P_normalized;
int * dev_numhist;
int * dev_numwin;
int * dev_numdim;
// Dimension:
int size =HIST.numdim*HIST.numwin;
int dimen = HIST.numhist*HIST.numwin;
// Allocate space for CPU
float * _histmin = (float*)malloc(sizeof(float)*HIST.numdim);
float * _harmrest = (float*)malloc(sizeof(float)*size);
float * _center = (float*)malloc(sizeof(float)*size);
float * _numwham = (float*)malloc(sizeof(float)*HIST.numhist);
float * _g = (float*)malloc(sizeof(float)*HIST.numwin);
float * _F = (float*)malloc(sizeof(float)*HIST.numwin);
float * _delta= (float*)malloc(sizeof(float)*HIST.numdim);
int * _numbinwin=(int*)malloc(sizeof(int)*HIST.numwin);
```

```
float * _denwham=(float*)malloc(sizeof(float)*HIST.numwin);
float * _sumP=(float*)malloc(sizeof(float)*HIST.numwin);
float * _U=(float*)malloc(sizeof(float)*dimen);
int * _numbin=(int*)malloc(sizeof(int)*HIST.numdim);
int * _step=(int*)malloc(sizeof(int)*HIST.numdim);
float * _dev_sum_result=(float*)malloc(sizeof(float)*HIST.numhist);
float * _Load_kT=(float*)malloc(sizeof(float));
float * cpu_result_Punnorm=(float *)malloc(HIST.numhist * sizeof(float));
float * cpu_F= (float *)malloc(HIST.numwin * sizeof(float));
float * cpu_P = (float *)malloc(HIST.numhist*sizeof(float));
float * cpu_A_result = (float *)malloc(HIST.numhist*sizeof(float));
float * cpu_rmsd_result=(float *)malloc(sizeof(float));
float * _cpu_P_old=(float *)malloc(HIST.numhist*sizeof(float));
float * cpu_U=(float *)malloc(HIST.numhist*HIST.numwin * sizeof(float));
float * cpu_Punnorm=(float *)malloc(HIST.numhist*sizeof(float));
float * cpu_sumP = (float *)malloc(HIST.numwin*sizeof(float));
float * cpu_UU = (float *)malloc(HIST.numhist*HIST.numwin*sizeof(float));
int * cpu_numhist=(int*)malloc(sizeof(int));
int * cpu_numwin=(int*)malloc(sizeof(int));
int * cpu_numdim=(int*)malloc(sizeof(int));
printf("Initialization Array...\n");
// Allocation space for GPU:
cudaMalloc(&(dev_histmin), HIST.numdim* sizeof(float));
cudaMalloc(&(dev_numbinwin), HIST.numwin* sizeof(int));
cudaMalloc(&(dev_g), HIST.numwin* sizeof(float));
cudaMalloc(&(dev_F), HIST.numwin* sizeof(float));
cudaMalloc(&(dev_numwham), HIST.numhist* sizeof(float));
cudaMalloc(&(dev_center), size* sizeof(float));
cudaMalloc(&(dev_harmrest), size* sizeof(float));
cudaMalloc(&(dev_delta), HIST.numdim* sizeof(float));
cudaMalloc(&(dev_step), HIST.numdim * sizeof(int));
cudaMalloc(&(dev_numbin), HIST.numdim* sizeof(int));
cudaMalloc(&(dev_U),dimen*sizeof(float));
cudaMalloc(&(dev_sumP),HIST.numwin*sizeof(float));
cudaMalloc(&(dev_result_Punnorm),HIST.numhist* sizeof(float));
cudaMalloc(&(dev_Punnorm_result),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_denwham),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_F_toBeNormalized),HIST.numwin*sizeof(float));
cudaMalloc(&(dev_sum_normfactor_for_normprob_and_normcoef),sizeof(float));
cudaMalloc(&(dev_P),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_P_normalized),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_kT),sizeof(float));
cudaMalloc(&(dev_A_result),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_P_old),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_rmsd_result),HIST.numhist*sizeof(float));
cudaMalloc(&(dev_UU),HIST.numwin*HIST.numhist*sizeof(float));
cudaMalloc(&(dev_numhist),sizeof(int));
cudaMalloc(&(dev_numwin),sizeof(int));
cudaMalloc(&(dev_numdim),sizeof(int));
 _Load_kT[0]=kT;
 cpu_numhist[0]=HIST.numhist;
 cpu_numwin [0]=HIST.numwin;
 cpu_numdim [0]=HIST.numdim;
 for(int i=0; i<size;i++){
 _center[i]= HIST.center[i];
 _harmrest[i]= HIST.harmrest[i];
 _U[i]=0.0;
 }
 for(int i=0; i<HIST.numhist;i++){
  _numwham[i]= HIST.numwham[i];
  _denwham[i]=0.0;
  _cpu_P_old[i]=0.0;
 }
  for(int i=0;i<HIST.numwin;i++){
```

```
  _g[i]=HIST.g[i];
  _F[i]=HIST.F[i];
  _numbinwin[i]=HIST.numbinwin[i];
  _sumP[i]=0.0;
}
for(int i=0;i<HIST.numdim;i++)
{
  _delta[i]=HIST.delta[i];
  _numbin[i]=HIST.numbin[i];
  _histmin[i]=HIST.histmin[i];
  _step[i]=HIST.step[i];
  }
printf(”Copy CPU var to GPU device ...\n”);
//Copy CPU sapce to GPU device :
cudaMemcpy(dev_numhist,cpu_numhist,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_numw in,cpu_numwin,sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_numdim,cpu_numdim,sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_histmin, _histmin,HIST.numdim* sizeof(float *),
                                                    cudaMemcpyHostToDevice);
cudaMemcpy(dev_numbinwin,_numbinwin,HIST.numwin* sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_g,_g,HIST.numwin* sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_F,_F,HIST.numwin* sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_numwham,_numwham,HIST.numhist* sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_center,_center,(HIST.numdim*HIST.numwin)* sizeof(float),
                                                   cudaMemcpyHostToDevice);
cudaMemcpy(dev_harmrest,_harmrest,(HIST.numdim*HIST.numwin)* sizeof(float),
                                                   cudaMemcpyHostToDevice);
cudaMemcpy(dev_delta,_delta,HIST.numdim* sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_step,_step,HIST.numdim* sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_numbin,_numbin,HIST.numdim* sizeof(int) cudaMemcpyHostToDevice);
cudaMemcpy(dev_sumP,_sumP,HIST.numwin* sizeof(float) cudaMemcpyHostToDevice);
cudaMemcpy(dev_denwham,_denwham,HIST.numhist* sizeof(float),cudaMemcpyHostToDevice);
```

```
 cudaMemcpy(dev_kT,_Load_kT,sizeof(float),cudaMemcpyHostToDevice);
 cudaMemcpy(dev_P_old,_cpu_P_old,HIST.numhist*sizeof(float),cudaMemcpyHostToDevice);
 printf("GPU Processing ...\n");
// -- Calculation of Potential --
initGPU=clock();
Bias(HIST.numhist, HIST.numwin,HIST.numdim,dev_numhist,dev_numdim,dev_histmin,dev_center,
                                        dev_harmrest, dev_delta,dev_step,dev_numbin,dev_U,dev_numwham);
while((it < numit)&&(!converged)){
//Calculating new probability
NewProbabilities(cpu_numhist[0],cpu_numwin[0],dev_numhist,dev_numwin,
dev_numbinwin,dev_g,dev_numwham,dev_U,dev_F,dev_denwham,dev_Punnorm_result);
//Calculating new Sum
summationP (cpu_numhist[0],cpu_numwin[0],
            dev_numhist,dev_numwin,dev_U,dev_UU,dev_numwham);
NewSum (dev_numhist,cpu_numwin[0],dev_sumP,dev_UU,dev_Punnorm_result,dev_numwham);
//Calculating new constant F
NewConstants(cpu_numhist[0],cpu_numwin[0],dev_U,dev_Punnorm_result,
                                                        dev_sumP,dev_F,dev_numwham);
//Calculating Normalization Constant
NormFactor(cpu_numhist[0],dev_Punnorm_result,
                         dev_sum_normfactor_for_normprob_and_normcoef,dev_numwham);
//Normalization of P
NormProbabilities (cpu_numhist[0],dev_sum_normfactor_for_normprob_and_normcoef,
                                                  dev_Punnorm_result,dev_P,dev_numwham);
//Normalization of F
NormCoefficient(cpu_numwin[0],dev_sum_normfactor_for_normprob_and_normcoef
                                                                  ,dev_F,dev_sumP);
//Convergence of the Math Model
CheckConvergence(cpu_numhist[0],dev_P,dev_P_old,HIST.numgood,
                                                        dev_rmsd_result,dev_numwham);
//Calculating Free Energy
ComputeEnergy(cpu_numhist[0],dev_P,dev_kT,dev_A_result,dev_P_old,dev_denwham);
```

```
    cudaMemcpy(cpu_rmsd_result,dev_rmsd_result,sizeof (float),cudaMemcpyDeviceToHost);
    if (cpu_rmsd_result[0] < tol)
    converged = true;//Is it converged ?
    it++;
    }
  finalGPU=clock()-initGPU;
```

3.1.2 Calculating Bias

After invoking a kernel from *extern "C" void Bias()*, we create a grid of size *numdim * numhist*, where *numdim* is the dimensionality of the grid *(1D,2D,3D)* while, *numhist* is the number of all bins. In addition to the grid blocks *numhist*numwin* are allocated, where *numwin* is the number of simulations. Inside the kernel we initialize the variables of interaction of the blocks, which allow to scroll the various blocks with *ihist* and *iwin*. Instead, $iter_i$ runs *harmrest* which is a vector that contains the constants K_i of the potential and vector center which instead contains the coordinates where are sampled molecules in space. After, calculating the potential in the line of code 18 that replicates the formula *1.17 of Chapter 1* where *pos* is the index that translates the simulation windows. Finally, each of these simulations is passed to a temporary array in global memory with $iter_j$.

In order to minimize the operations of the memory access, Bias is loaded into the array *dev_U* which contains the data calculated by *Bias.cu* on *constant memory* because has a faster access and it often used by iterative cycle. The same strategy was used for vector which contains $P_i^u(\xi)$, i.e. *dev_Punnorm_result*.

Listing 3.2: *Bias.cu*

```
1 //GPLv3, Copyright (C) 2013 Nicolo' Savioli
2 #include <assert.h>
3 #include <helper_cuda.h>
4 __global__ void BiasKernel(int * dev_numhist,int * dev_numdim,
5 float * histmin,float * center,float * harmrest,float * delta,
int * step,int* numbin,float * temp_U,float * dev_numwham){
  double pos=0.0;
  int ipos=0;
  int ihist=blockIdx.x;
  int iwin=blockIdx.y;
  int idim=threadIdx.x;
  int iter_i=idim+iwin*dev_numdim[0];
  int iter_j=ihist+iwin*dev_numhist[0];
  float U=0.0;
  if(ihist<dev_numhist[0]){
   if(dev_numwham[ihist]>1){
     ipos = (int)((ihist/step[idim]) % numbin[idim]);
     pos = histmin[idim] + ipos*delta[idim];
     U += 0.5 * harmrest[iter_i] *(pos - center[iter_i])*(pos - center[iter_i]);
     temp_U[iter_j] = U;
     __syncthreads();
    }}}
extern "C" void Bias (int numhist,int numwin,int numdim,
                      int * dev_numhist, int * dev_numdim,
                      float * histmin,float * center,
                      float * harmrest,float * delta,
                      int * step,int* numbin,float * temp_U,
                      float * dev_numwham){
   dim3 dimGrid(numdim,numhist);
   dim3 dimBlock(numhist,numwin);
   // Kernel invocation
   BiasKernel<<<dimBlock,dimGrid>>>(dev_numhist,dev_numdim,histmin,
                                    center,harmrest,delta,step,numbin,
                                    temp_U,dev_numwham);
   getLastCudaError("Cuda Bias execution failed\n");
}
```

3.1.3 Probability Update: Calculation of New Probability

This kernel in *NewProbabilities.cu* has the aim to calculate the *first equation of the system (1.20) of Chapter 1*, in order to perform this, the blocks are initialized with dimension *numhist*numwin* after several tests it was decided to use __expf to speed up the calculation of the exponential function.

Listing 3.3: *NewProbabilitie.cu*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>

__global__ void NewProbabilitiesKernel(int * dev_numhist,int * dev_numwin,
                                        int * numbinwin,float * g,float* numwham,
                                        float* U,float * F,float *denwham,
                                        float * sum_result){
  int ihist = blockIdx.x;
  int iwin = blockIdx.y;
  if((ihist<dev_numhist[0])&&(iwin<dev_numwin[0])){
   if(numwham[ihist]>1){
      denwham[ihist] += (numbinwin[iwin]/g[iwin]) *__expf(F[iwin]-U[ihist+iwin*dev_numhist[0]]);
      sum_result[ihist] =numwham[ihist]/denwham[ihist];
      __syncthreads();
    }}}
extern "C" void NewProbabilities (int numhist,int numwin,int * dev_numhist,
                                  int * dev_numwin,int * numbinwin,float * g,
                                  float* numwham,float* U,float * F,float * denwham,
                                  float * sum_result){
 dim3 Block(numhist,numwin);
 NewProbabilitiesKernel<<<Block,1>>>(dev_numhist,dev_numwin,numbinwin,
                                      g,numwham,U,F,denwham,sum_result);
 getLastCudaError("Cuda NewProbabilities execution failed\n");
}
```

3.1.4 Constants Update: Calculation of Constant F_i

The result of the previous kernel is used in order to update the constant F_i by means of *the second equation of the system (1.20) of Chapter 1.* In order to speed up the algorithm, the calculation of the kernel is divided into two part: the first kernel computes the most critical part of the operation $-\beta e^{W_i(\xi_h)}$; and the second kernel, performs a simple multiplication operation between $P_i^u(\xi)$, calculated with *NewProbabilities.cu*, and $-\beta e^{W_i(\xi_h)}$ previously calculated.

Listing 3.4: *summationP.cu First part of kernel used for the calculation of F_i*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>

__global__ void summationPKernel (int * dev_numhist,int * dev_numwin,float* U,
                                  float * dev_UU,float * dev_numwham){
  int ihist = blockIdx.y;
  int iwin = blockIdx.x;
  if((iwin<dev_numwin[0])&&(ihist<dev_numhist[0])){
     if(dev_numwham[ihist]>1)
        dev_UU[ihist+iwin*dev_numhist[0]] = __expf(-U[ihist+iwin*dev_numhist[0]]);
        __syncthreads();
   }}
extern "C" void summationP (int numhist,int numwin,int * dev_numhist,
                            int * dev_numwin,float * U,float * dev_UU,
                            float * dev_numwham){
  dim3 dimBlock(numwin,numhist);
  // Kernel invocation
  summationPKernel<<<dimBlock, 1>>>(dev_numhist,dev_numwin,U,dev_UU,dev_numwham);
  getLastCudaError("Cuda summationPexecution failed\n");
}
```

Listing 3.5: *NewSum.cu Second part of kernel used for the calculation of F_i*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>

__global__ void NewSumKernel (int * dev_numhist,
                              float * sumP,float * U,
                              float * Punnorm,
                              float * numwham){
  int iwin = blockIdx.x;
  for(int i=0;i<dev_numhist[0];i++)
   {
    if(numwham[i]>1)
     sumP[iwin] +=U[i+iwin*dev_numhist[0]]*Punnorm[i];
   }}
extern "C" void NewSum (int * dev_numhist,int numwin,
                        float * sumP,float * U,
                        float * Punnorm, float * numwham){
   dim3 dimBlock(numwin,1);
   // Kernel invocation
   NewSumKernel<<<dimBlock,1>>>(dev_numhist,sumP,U,Punnorm,numwham);
   getLastCudaError("Cuda NewSum execution failed\n");
}
```

3.1.5 Normalization

The normalization process involves both: $P_i^u(\xi)$ and F_i. This normalization, is made by two CUDA external function: the first, which normalizes $P_i^u(\xi)$, is *NormProbabilities.cu* while the second,which has instead the function of normalizing F_i, is *NormCoefficient.cu*. Them both these processes require a coefficient of normalization

which it is computed as the sum of all probabilities $P_i^u(\xi)$. In order to optimize the process of summation we used a technique called *sum reduction*.
This technique uses a tree-based approach for each thread of a block. Allowing it to be capable of processing large arrays and each thread block reduces a portion of the array therefore, if each thread of a block is synchronized and it can produce a single result that can be shared with another through *shared memory*; finally, the result of that block, is written to global memory. This normalization factor is calculated with the kernel of *NormFactor.cu* external function that implements the code below.

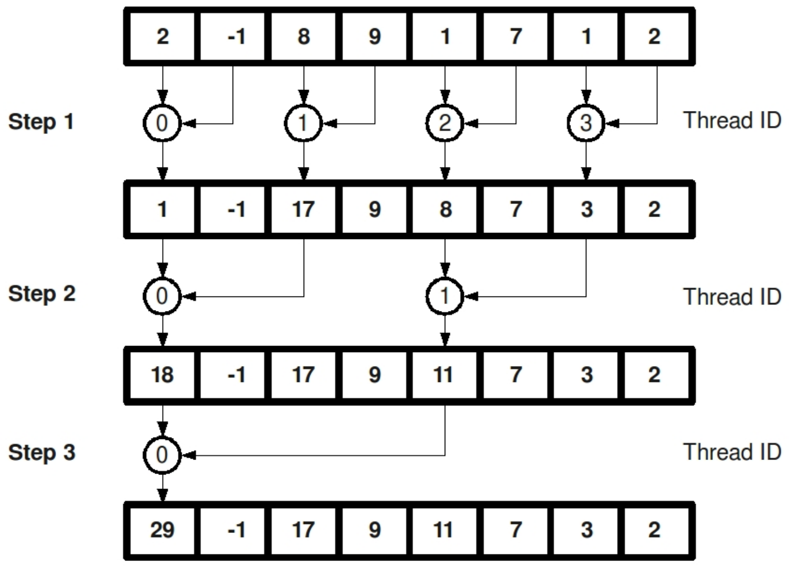

Figure 3-1: *Sum Reduction from "http://cs.anu.edu.au/".*

Listing 3.6: *NormFactor.cu*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>

__global__ void NormFactorKernel(int numhist,float* Punnorm,float* sum_result,float * dev_numwham){
  // use shared memory to speed up the operation of summation
  extern __shared__ float sdata[];
  int tx = threadIdx.x;
  float sum=0.0;

   if(tx<numhist){
    if (dev_numwham[tx]>1)
    sum=Punnorm[tx];
    sdata[tx] = sum;
     __syncthreads();
   }
  for(int offset = blockDim.x / 2;offset > 0;offset >>= 1){
   if(tx < offset){
    // add a partial sum upstream to our own
    sdata[tx] += sdata[tx + offset];
   }
   __syncthreads();
  }
   // finally, thread 0 writes the result
  if(threadIdx.x == 0){
   // note that the result is per-block
   // not per-thread
  sum_result[0] = sdata[0];
  }}

extern "C" void NormFactor (int numhist,float* Punnorm,float * sum_result,float * dev_numwham)
```

```
{
  int smem_sz = (256)*sizeof(float);
  dim3 Block(numhist,1);
  NormFactorKernel<<<Block,256,smem_sz>>>(numhist,Punnorm,sum_result,dev_numwham);
  getLastCudaError("Cuda NormFactor execution failed\n");
}
```

Normalization of $P_i^u(\xi)$ is done dividing by this norm factor as implemented in the external function *NormProbabilities.cu*.

Listing 3.7: *NormProbabilities.cu*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>

__global__ void NormProbabilitiesKernel(int numhist,float * sum_result,
                                        float * Punnorm,float * P,
                                        float * dev_numwham){
  int ihist = blockIdx.x;
  if(ihist<numhist){
   if(dev_numwham[ihist]>1)
   P[ihist]=Punnorm[ihist]/sum_result[0];
   __syncthreads();
  }}

extern "C" void NormProbabilities(int numhist,float * sum_result,
                                  float * Punnorm,float * P,
                                  float * dev_numwham){
  dim3 Block(numhist,1);
  NormProbabilitiesKernel<<<Block,1>>>(numhist,sum_result,Punnorm,P,dev_numwham);
  getLastCudaError("Cuda NormProbabilities execution failed\n");

}
```

The normalization of the constant F_i is done by adding the logarithm of the normalization factor as implemented in the kernel of the external function *NormCoefficient.cu*. In this function it is also initialized in global memory *sumP* which acts as a accumulator for the calculation of the constant F_i in the kernel of *NewSum.cu*.

3.1.6 Calculation of the Free Energy

The calculation of the free energy follows the *Equation 1.5 in Chapter 1*, then we go to calculate $-kT \log P_i^u(\xi)$ in the kernel of the external function *ComputeEnergy.cu*. In order to calculate the convergence, we insert the values of $P_i^u(\xi)$ in a state vector that at each iteration contains $P_i^u(\xi)$ values of the previous iteration.

Listing 3.8: *ComputeEnergy.cu*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>


__global__ void ComputeEnergyKernel(int numhist,float * P,float * kT,
                                    float * A_result,float * P_old,
                                    float * denwham){
  int ihist = blockIdx.x;
  if(ihist<numhist){
   A_result[ihist] = -kT[0]*__logf(P[ihist]);
   P_old[ihist]=P[ihist];
   denwham[ihist]=0.0;
   __syncthreads();
  }}
extern "C" void ComputeEnergy(int numhist,float * P,
                              float * kT,float * A_result,
                              float * P_old,float * denwham){
```

```
  dim3 Block(numhist,1);
  ComputeEnergyKernel<<<Block,1>>>(numhist,P,kT,A_result,P_old,denwham);
  getLastCudaError("Cuda ComputeEnergy execution failed\n");
}
```

3.1.7 Convergence Checking

The calculation of the stability of the model is a critical step since it allows to stop the iterative cycle when it is converged. The logic with which ensures that the algorithm has reached convergence is to check if the sum of the deviations between the calculated $P^{u}_{i-1}(\xi)$ at the iteration $i-1$ and the one at iteration i is less than a tolerance value. It is implemented through the kernel of the function *CheckConvergence.cu*, for better performances the computation is implemented a sum reduction.

Listing 3.9: *CheckConvergence.cu*

```
//GPLv3, Copyright (C) 2013 Nicolo' Savioli
#include <assert.h>
#include <helper_cuda.h>

__global__ void CheckConvergenceKernel(int numhist,float * P_new,
                                        float * P_old,int numgood,
                                        float * Check_result,float * dev_numwham){

   // use shared memory to speed up the operation of summation
  extern __shared__ float sdata[];

  int tx = threadIdx.x;
  float sum=0.0;

   if(tx<numhist){
    if(dev_numwham[tx])
    sum=(P_new[tx] - P_old[tx]) * (P_new[tx] - P_old[tx]);
    sdata[tx] = sum;
     __syncthreads();
   }

  for(int offset = blockDim.x / 2;offset > 0;offset >>= 1){
   if(tx < offset)
   {
    // add a partial sum upstream to our own
    sdata[tx] += sdata[tx + offset];
   }
   __syncthreads();
  }
   // finally, thread 0 writes the result
  if(threadIdx.x == 0){
   // note that the result is per-block
   // not per-thread
   Check_result[0] = sqrt(sdata[0]/numgood);
  }}

extern "C" void CheckConvergence(int numhist,float * P_new,
                                  float * P_old,int numgood,
                                  float * Check_result,float * dev_numwham){
 int smem_sz = (256)*sizeof(float);
 dim3 Block(numhist,1);
 CheckConvergenceKernel<<<Block,256,smem_sz>>>(numhist,P_new,P_old,
                                        numgood,Check_result,dev_numwham);
 getLastCudaError("Cuda CheckConvergence execution failed\n");
}
```

3.2 Hardware Specifications

The work of this thesis was mainly carried out in two different GPUs architectures in which we compared the running times of the code. What we want to prove is that the code has a performance increase when the computational capability of the GPU to increase. To this end we took two GPUs with two different level of *Compute Capability*: *GT 9500* with a Compute Capability of 1.1 and *GT 320M* of 1.0.
The GeForce 9500 GT has 32 CUDA cores with a graphics clock of 550 MHz and 800 MHz to GDDR3 Memory Clock with 500 MHz of DDR2, the *memory bandwidth* for GDDR3 is 25.6 GB/sec and 16.0 GB/s for DDR2[8].
The 320M is an integrated version of the GT 320 for desktop and it has lower hardware specific. The GT 320M is in possession of 24 CUDA cores with a graphics clock of 500 MHz. Instead, memory clock, has a clock of 790 MHz (1580 MHz data rate) with a memory interface of 128-bit.
The results were compared with two different CPU architecture: *Intel i5 series 3400*, which is a quad core processor where each core has a clock 64-bit of 2.24GHz with a cache size to 3072 KB, and an dual core *Athlon 64 bit X2* that each core has a clock of 1.8 GHz with cache size to 512 KB.

3.3 Results

It is to be noted that the software presented here was adapted for working on graphics cards with low compute capability and then with some limitations such as the ability to work in double precision. The main purpose of the test was to demonstrate that the performances of the software improve with the increase of the compute capability of the graphics card.The results are presented in the graphs below.

[8]http://www.geforce.com/hardware/desktop-gpus/geforce-9500-gt/specifications

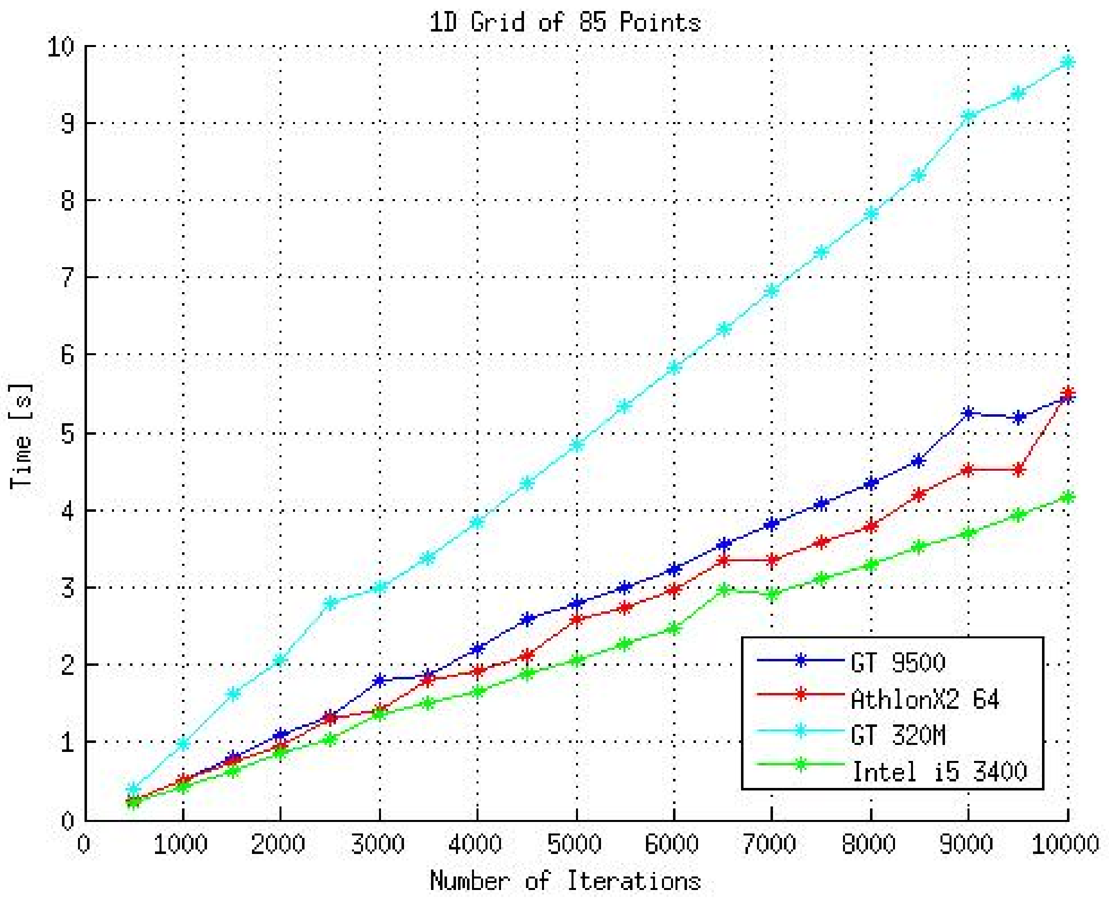

Figure 3-2: The figure compares the different architectures of CPUs and GPUs in a grid of 85 points. As expected, the code shows better performances when it was executed in graphics cards with a greater compute capability (1.1). In fact, as it can be seen from the graph, there is a decrease of coefficient angle of about -53.3% from a GPU with compute capability 1.0 to compute capability 1.1. However, we note the inability of GPUs at our disposal to overcome CPUs.

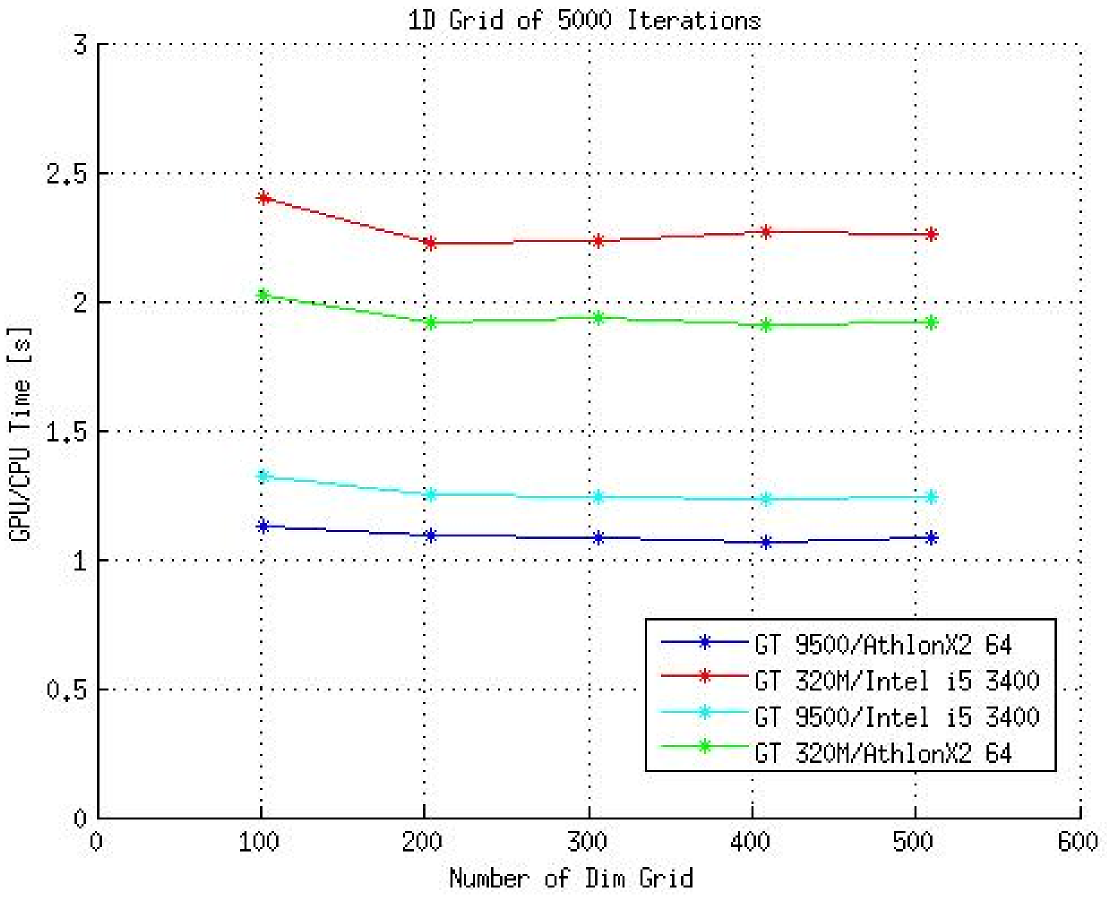

Figure 3-3: The figure describes that varying size of the grid, at equal iteration, the ratio between the GPUs and CPUs remained constant. It is possible to obtain better results if the ratio was made between a fast GPU and a slow processor CPU. It is clear that the relations between the GPU and CPU with the same number of iterations and variable grid tends to decrease gradually with the increase in the speed of the GPU, while the ratio remained constant.

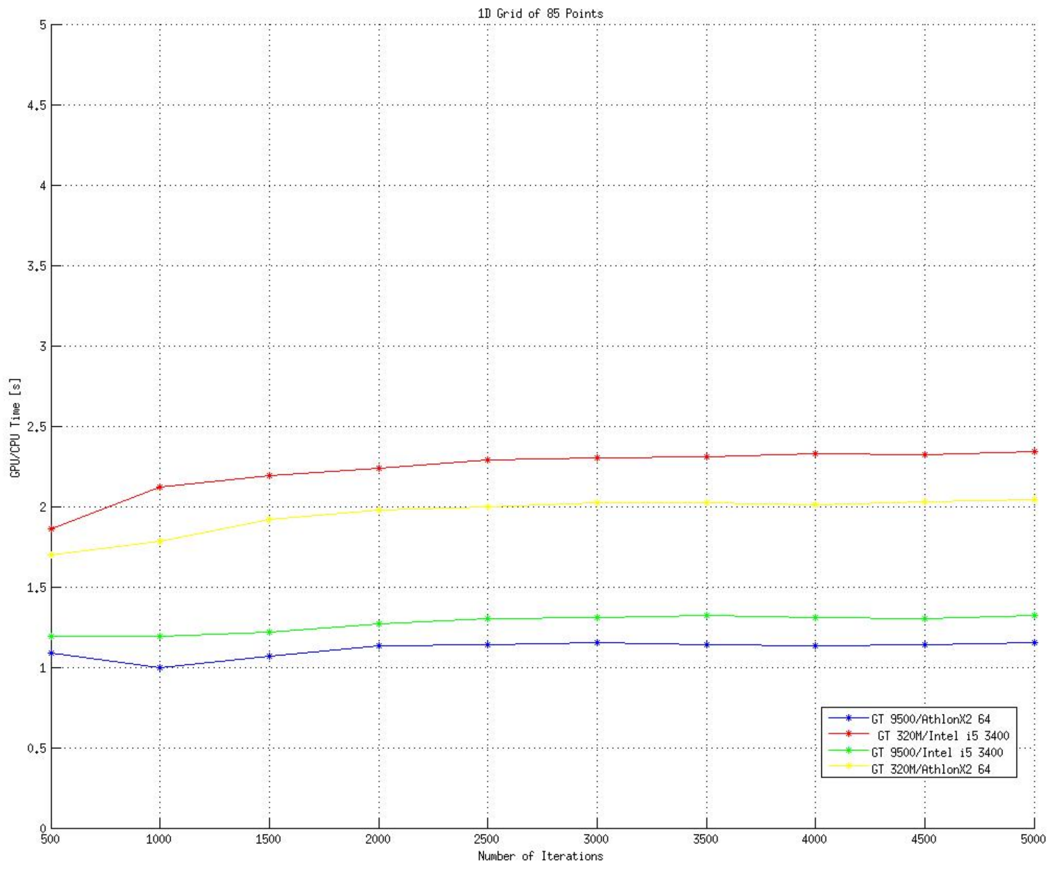

Figure 3-4: The figure describes the constancy of the ratio between the CPU and GPU when varying the number of iterations, where we see the same phenomenon seen before, i.e. the ratio drops to increase the speed of the GPU.

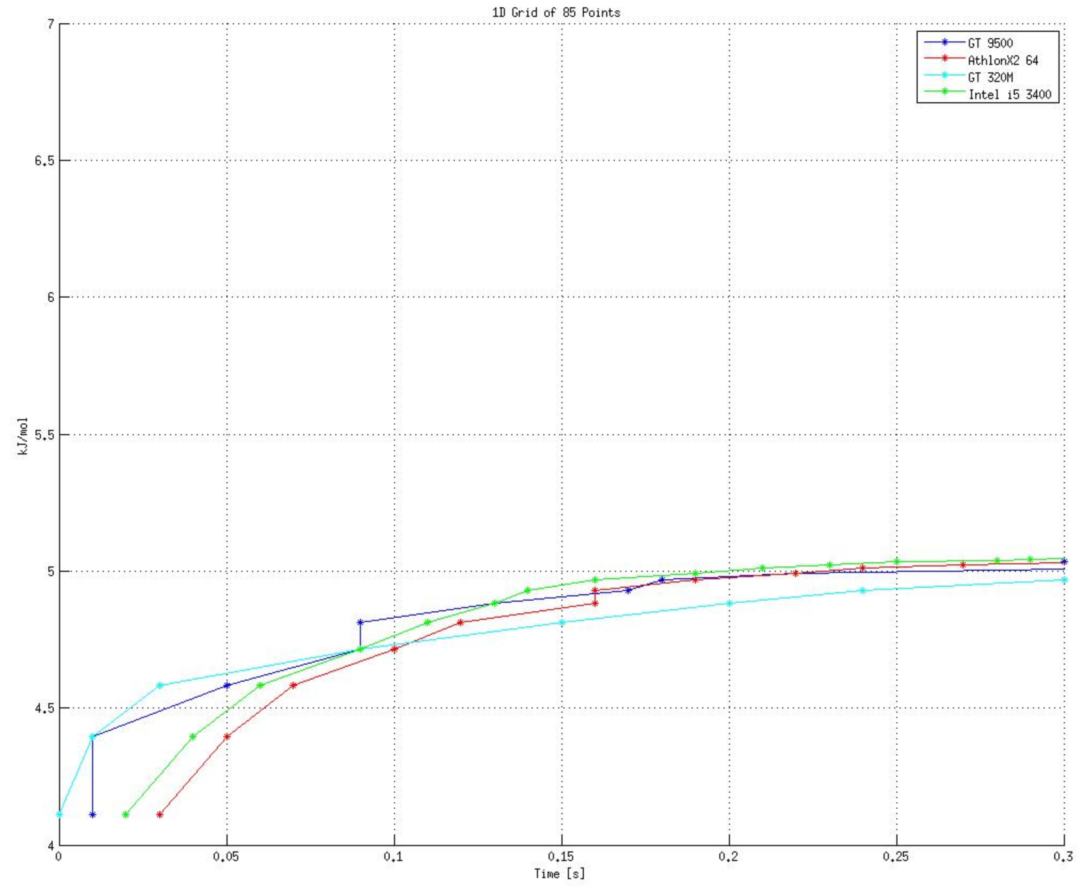

Figure 3-5: The figure, describes the convergence of the model analyzed in function of time. Observing the figure, it denotes a more rapid convergence to the GT 9500 with compute capability 1.1, compared GT 320 that has compute capability 1.0.

Conclusion

After having analysed the algorithm proposed in different GPUs architectures, it has been understood how the algorithm shows an improvement in the speed of execution when it is used by GPUs with higher compute capabilities. Furthermore, it has been noticed that the relations between GPUs and CPUs, where the algorithm was tested, remain constant. In addition, this relation decreases when the speed of the GPU increases and at the decrease of the speed of CPU; indeed, from the proposed graphs it has been shown that at equal number of iterations, varying the point of the grid (where it is calculated the free energy), the relation between the fastest GPU (GT 9500) and the slowest CPU (Athlon X2 64-bit) is the smallest above the entire ratios. This is confirmed by one of the charts that correlates this relation with the variation of number of iterations, showing that the smallest ratio is maintained constant from the use of the same GPU and CPU the previous graph. In conclusion, having considered the architectural limits where the simulation are executed, it is possible to understood how a use of GPUs with higher compute capabilities can decrease the times of execution of WHAM. The implemented model is limited because it can works only in one-dimensional grids. Indeed, due to an implementative choice, bi- or three-dimensional grids have been converted in 1D arrays. This choice brings to an incapacity to calculate grids with dimensions higher than one-dimension because the maximum dimension of the CPU grid is 512. To solve this problem, it should have been worked from the beginning with bi- or three- dimensional. However, the problem is partially avoidable if the calculation of the grid is made through several steps with size less than 512. Another use that can be done thank to this algorithm is that of distributing the calculations at the same time in GPU and CPU in the same

PC used, using one more resources to compute.
The main aim of all the thesis has been that of implementing a free energy method to speed up the calculation of free energy in GPU. It has been demonstrated how the implementation of free energy method in GPU, as WHAM algorithm, can improve the performances of calculation. This approach might be extended in other situations where the use of GPU present in graphic cards at high performance computing, could bring an advantage in the computational times without the use of expensive data centres, optimizing costs and benefits.

References

1. Carmen Domene, Simone Furini. Examining ion channel properties using free-energy methods.Physical and Theoretical Chemistry Laboratory, Department of Chemistry, University of Oxford, Oxford, United Kingdom. Methods in enzymology (Impact Factor: 1.9). 01/2009; 466:155-77. DOI:10.1016/S0076-6879(09)66007-9.

2. Johannes Kastner Umbrella sampling.(2011). Jhon Willey and Sons,Ltd Published Online: May 26 2011 DOI:10.1002/wcms.66.

3. Michael Andrec.The Weighted Histogram Analysis Method (WHAM). January 21, 2010.

4. XU Wei-Xin,LI Yang 1 , ZHANG John Z. H. Calculation of Collective Variable-based PMF by Combining WHAM with Umbrella Sampling State Key Laboratory of Precision Spectroscopy, Department of Physics, East China Normal University, Shanghai 200062 Institute of Theoretical and Computational Science, Institutes for Advanced Interdisciplinary Research, East China Normal University, Shanghai 200062 Department of Chemistry, New York University, New York, New York 10003, USA.

5. Krivov SV, Karplus M (2006) One-dimensional free-energy profiles of complex systems: Progress variables that preserve the barriers. J Phys Chem B 110: 1268912698. doi: 10.1021/jp060039b.

6. Danzhi Huang, Amedeo Caflisch.The Free Energy Landscape of Small Molecule Unbinding. Published: February 03, 2011DOI: 10.1371/journal.pcbi.1002002.

7. Apaydin M, Brutlag D, Guesttin C, Hsu D, Latombe J (2002) Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion. In International Conference on Computational Molecular Biology (RECOMB).

8. Fowler, D. M., Koulov, A.V., Alory-Jost, C., Marks, M. S., Balch, W. E. Kelly, J.W. (2006).Functional amyloid formation within mammalian tissue.Department of Chemistry, The Skaggs Institute of Chemical Biology, The Scripps Research Institute, La Jolla, California, USA.

9. O'Nuallain, B., Shivaprasad, S., Kheterpal, I. Wetzel,R. (2005). Thermodynamics of A(140) amyloid fibril elongation. Biochemistry, 44, 127091271.

10. Marino Convertino, Riccardo Pellarin, [...], and Amedeo Caflisch.(2009).0-Anthraquinone hinders - aggregation: How does a small molecule interfere with A-peptide amyloid fibrillation?Article first published online: 10 FEB 2009 DOI: 10.1002/pro.87 Copyright 2009 The Protein Society

11. http://www.nvidia.com/object/tesla-supercomputing-solutions.html

12. http://www.nvidia.com/object/fermi-architecture.html

13. Peter N. Glaskowsky NVIDIAs Fermi: The First Complete GPU Computing Architecture A white paper Prepared under contract with NVIDIA Corporation.

14. http://docs.nvidia.com/cuda/cuda-c-programming-guide

15. http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

16. A. Leist, D. P. Playne and K. A. Hawick. (December 2008). Exploiting Graphical Processing Units for Data-Parallel Scientic Applications, Computer Science,Institute of Information and Mathematical Sciences,Massey University, Albany, Auckland, New Zealand.

17. http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/215900921

18. http://www.sdsc.edu/us/training/assets/docs/NVIDIA-04-OptimizingCUDA.pdf

19. Encyclopedia of Parallel Computing, David Padua, Springer Science LLC 2011.

20. Optimizing CUDA Applications, 3D Game Engine Programming, Execution Optimizations from http://3dgep.com/?p=2081.